TOSHIBA

64-Bit TX System RISC
TX49/H2 Core Architecture

JAN. 2002

TOSHIBA CORPORATION

R4000/R4400/R5000 are a trademark of MIPS Technologies, Inc.

The information contained herein is subject to change without notice.

The information contained herein is presented only as a guide for the applications of our products. No responsibility is assumed by TOSHIBA for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of TOSHIBA or others.

The products described in this document contain components made in the United States and subject to export control of the U.S. authorities. Diversion contrary to the U.S. law is prohibited.

TOSHIBA is continually working to improve the quality and reliability of its products. Nevertheless, semiconductor devices in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress.
It is the responsibility of the buyer, when utilizing TOSHIBA products, to comply with the standards of safety in making a safe design for the entire system, and to avoid situations in which a malfunction or failure of such TOSHIBA products could cause loss of human life, bodily injury or damage to property.
In developing your designs, please ensure that TOSHIBA products are used within specified operating ranges as set forth in the most recent TOSHIBA products specifications.
Also, please keep in mind the precautions and conditions set forth in the “Handling Guide for Semiconductor Devices,” or “TOSHIBA Semiconductor Reliability Handbook” etc..

The Toshiba products listed in this document are intended for usage in general electronics applications (computer, personal equipment, office equipment, measuring equipment, industrial robotics, domestic appliances, etc.).
These Toshiba products are neither intended nor warranted for usage in equipment that requires extraordinarily high quality and/or reliability or a malfunction or failure of which may cause loss of human life or bodily injury (“Unintended Usage”). Unintended Usage include atomic energy control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, combustion control instruments, medical instruments, all types of safety devices, etc.. Unintended Usage of Toshiba products listed in this document shall be made at the customer’s own risk.

The products described in this document may include products subject to the foreign exchange and foreign trade laws.

Preface

Thank you for your new or continued patronage of Toshiba semiconductor products. This is the 1998 edition of the user's manual for the TX49 Family of 64-bit RISC microprocessors, entitled 64-Bit TX System RISC TX49/H2 Architecture.

This manual is written so as to be accessible to engineers who may be designing a Toshiba microprocessor into their products for the first time. No prior knowledge of these devices is assumed. The manual includes a review of the architecture of the processor family, a description of the TX49 instruction set, and sections dedicated to various other relevant topics, such as the Memory Management System (MMU) and CPU exceptions.

Toshiba continually updates its technical information. Your comments and suggestions concerning this and other Toshiba documents are sincerely appreciated and may be used in subsequent editions. For updates to this document or for additional information about the product, please contact your nearest Toshiba office or authorized Toshiba dealer.

January 2002

Contents

Handling Precautions

Handling Precautions

1. Using Toshiba Semiconductors Safely

TOSHIBA are continually working to improve the quality and the reliability of their products.

Nevertheless, semiconductor devices in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress. It is the responsibility of the buyer, when utilizing TOSHIBA products, to observe standards of safety, and to avoid situations in which a malfunction or failure of a TOSHIBA product could cause loss of human life, bodily injury or damage to property.

In developing your designs, please ensure that TOSHIBA products are used within specified operating ranges as set forth in the most recent products specifications. Also, please keep in mind the precautions and conditions set forth in the TOSHIBA Semiconductor Reliability Handbook.

2. Safety Precautions

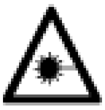
This section lists important precautions which users of semiconductor devices (and anyone else) should observe in order to avoid injury and damage to property, and to ensure safe and correct use of devices.

Please be sure that you understand the meanings of the labels and the graphic symbol described below before you move on to the detailed descriptions of the precautions.

[Explanation of labels]

⚠ DANGER	Indicates an imminently hazardous situation which will result in death or serious injury if you do not follow instructions.
⚠ WARNING	Indicates a potentially hazardous situation which could result in death or serious injury if you do not follow instructions.
⚠ CAUTION	Indicates a potentially hazardous situation which if not avoided, may result in minor injury or moderate injury.

[Explanation of graphic symbol]

Graphic symbol	Meaning
	Indicates that caution is required (laser beam is dangerous to eyes).

2.1 General Precautions regarding Semiconductor Devices

⚠CAUTION

Do not use devices under conditions exceeding their absolute maximum ratings (e.g. current, voltage, power dissipation or temperature).
This may cause the device to break down, degrade its performance, or cause it to catch fire or explode resulting in injury.

Do not insert devices in the wrong orientation.
Make sure that the positive and negative terminals of power supplies are connected correctly. Otherwise the rated maximum current or power dissipation may be exceeded and the device may break down or undergo performance degradation, causing it to catch fire or explode and resulting in injury.

When power to a device is on, do not touch the device's heat sink.
Heat sinks become hot, so you may burn your hand.

Do not touch the tips of device leads.
Because some types of device have leads with pointed tips, you may prick your finger.

When conducting any kind of evaluation, inspection or testing, be sure to connect the testing equipment's electrodes or probes to the pins of the device under test before powering it on.
Otherwise, you may receive an electric shock causing injury.

Before grounding an item of measuring equipment or a soldering iron, check that there is no electrical leakage from it.
Electrical leakage may cause the device which you are testing or soldering to break down, or could give you an electric shock.

Always wear protective glasses when cutting the leads of a device with clippers or a similar tool.
If you do not, small bits of metal flying off the cut ends may damage your eyes.

2.2 Precautions Specific to Each Product Group

2.2.1 Optical semiconductor devices

⚠ DANGER

When a visible semiconductor laser is operating, do not look directly into the laser beam or look through the optical system.
This is highly likely to impair vision, and in the worst case may cause blindness.
If it is necessary to examine the laser apparatus, for example to inspect its optical characteristics, always wear the appropriate type of laser protective glasses as stipulated by IEC standard IEC825-1.

⚠WARNING

Ensure that the current flowing in an LED device does not exceed the device’s maximum rated current.
This is particularly important for resin-packaged LED devices, as excessive current may cause the package resin to blow up, scattering resin fragments and causing injury.

When testing the dielectric strength of a photocoupler, use testing equipment which can shut off the supply voltage to the photocoupler. If you detect a leakage current of more than 100 μA, use the testing equipment to shut off the photocoupler’s supply voltage; otherwise a large short-circuit current will flow continuously, and the device may break down or burst into flames, resulting in fire or injury.

When incorporating a visible semiconductor laser into a design, use the device’s internal photodetector or a separate photodetector to stabilize the laser’s radiant power so as to ensure that laser beams exceeding the laser’s rated radiant power cannot be emitted.
If this stabilizing mechanism does not work and the rated radiant power is exceeded, the device may break down or the excessively powerful laser beams may cause injury.

2.2.2 Power devices

⚠ DANGER

Never touch a power device while it is powered on. Also, after turning off a power device, do not touch it until it has thoroughly discharged all remaining electrical charge.
Touching a power device while it is powered on or still charged could cause a severe electric shock, resulting in death or serious injury.

When conducting any kind of evaluation, inspection or testing, be sure to connect the testing equipment’s electrodes or probes to the device under test before powering it on.
When you have finished, discharge any electrical charge remaining in the device.
Connecting the electrodes or probes of testing equipment to a device while it is powered on may result in electric shock, causing injury.

⚠WARNING

Do not use devices under conditions which exceed their absolute maximum ratings (current, voltage, power dissipation, temperature etc.).
This may cause the device to break down, causing a large short-circuit current to flow, which may in turn cause it to catch fire or explode, resulting in fire or injury.

Use a unit which can detect short-circuit currents and which will shut off the power supply if a short-circuit occurs.
If the power supply is not shut off, a large short-circuit current will flow continuously, which may in turn cause the device to catch fire or explode, resulting in fire or injury.

When designing a case for enclosing your system, consider how best to protect the user from shrapnel in the event of the device catching fire or exploding.
Flying shrapnel can cause injury.

When conducting any kind of evaluation, inspection or testing, always use protective safety tools such as a cover for the device.
Otherwise you may sustain injury caused by the device catching fire or exploding.

Make sure that all metal casings in your design are grounded to earth.
Even in modules where a device’s electrodes and metal casing are insulated, capacitance in the module may cause the electrostatic potential in the casing to rise.
Dielectric breakdown may cause a high voltage to be applied to the casing, causing electric shock and injury to anyone touching it.

When designing the heat radiation and safety features of a system incorporating high-speed rectifiers, remember to take the device’s forward and reverse losses into account.
The leakage current in these devices is greater than that in ordinary rectifiers; as a result, if a high-speed rectifier is used in an extreme environment (e.g. at high temperature or high voltage), its reverse loss may increase, causing thermal runaway to occur. This may in turn cause the device to explode and scatter shrapnel, resulting in injury to the user.

A design should ensure that, except when the main circuit of the device is active, reverse bias is applied to the device gate while electricity is conducted to control circuits, so that the main circuit will become inactive.
Malfunction of the device may cause serious accidents or injuries.

⚠CAUTION

When conducting any kind of evaluation, inspection or testing, either wear protective gloves or wait until the device has cooled properly before handling it.
Devices become hot when they are operated. Even after the power has been turned off, the device will retain residual heat which may cause a burn to anyone touching it.

2.2.3 Bipolar ICs (for use in automobiles)

⚠CAUTION

If your design includes an inductive load such as a motor coil, incorporate diodes or similar devices into the design to prevent negative current from flowing in.
The load current generated by powering the device on and off may cause it to function erratically or to break down, which could in turn cause injury.

Ensure that the power supply to any device which incorporates protective functions is stable.
If the power supply is unstable, the device may operate erratically, preventing the protective functions from working correctly. If protective functions fail, the device may break down causing injury to the user.

3. General Safety Precautions and Usage Considerations

This section is designed to help you gain a better understanding of semiconductor devices, so as to ensure the safety, quality and reliability of the devices which you incorporate into your designs.

3.1 From Incoming to Shipping

3.1.1 Electrostatic discharge (ESD)

When handling individual devices (which are not yet mounted on a printed circuit board), be sure that the environment is protected against electrostatic electricity. Operators should wear anti-static clothing, and containers and other objects which come into direct contact with devices should be made of anti-static materials and should be grounded to earth via an 0.5- to 1.0-MΩ protective resistor.

Please follow the precautions described below; this is particularly important for devices which are marked “Be careful of static.”.

(1) Work environment

- When humidity in the working environment decreases, the human body and other insulators can easily become charged with static electricity due to friction. Maintain the recommended humidity of 40% to 60% in the work environment, while also taking into account the fact that moisture-proof-packed products may absorb moisture after unpacking.

- Be sure that all equipment, jigs and tools in the working area are grounded to earth.

- Place a conductive mat over the floor of the work area, or take other appropriate measures, so that the floor surface is protected against static electricity and is grounded to earth. The surface resistivity should be 10^4 to 10^8 Ω/sq and the resistance between surface and ground, 7.5×10^5 to 10^8 Ω

- Cover the workbench surface also with a conductive mat (with a surface resistivity of 10^4 to 10^8 Ω/sq, for a resistance between surface and ground of 7.5×10^5 to 10^8 Ω) . The purpose of this is to disperse static electricity on the surface (through resistive components) and ground it to earth. Workbench surfaces must not be constructed of low-resistance metallic materials that allow rapid static discharge when a charged device touches them directly.

- Pay attention to the following points when using automatic equipment in your workplace:

  (a) When picking up ICs with a vacuum unit, use a conductive rubber fitting on the end of the pick-up wand to protect against electrostatic charge.

  (b) Minimize friction on IC package surfaces. If some rubbing is unavoidable due to the device’s mechanical structure, minimize the friction plane or use material with a small friction coefficient and low electrical resistance. Also, consider the use of an ionizer.

  (c) In sections which come into contact with device lead terminals, use a material which dissipates static electricity.

  (d) Ensure that no statically charged bodies (such as work clothes or the human body) touch the devices.

(e) Make sure that sections of the tape carrier which come into contact with installation devices or other electrical machinery are made of a low-resistance material.

(f) Make sure that jigs and tools used in the assembly process do not touch devices.

(g) In processes in which packages may retain an electrostatic charge, use an ionizer to neutralize the ions.

- Make sure that CRT displays in the working area are protected against static charge, for example by a VDT filter. As much as possible, avoid turning displays on and off. Doing so can cause electrostatic induction in devices.

- Keep track of charged potential in the working area by taking periodic measurements.

- Ensure that work chairs are protected by an anti-static textile cover and are grounded to the floor surface by a grounding chain. (Suggested resistance between the seat surface and grounding chain is 7.5×10^5 to $10^{12}\Omega$.)

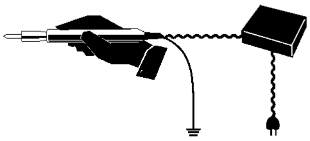
- Install anti-static mats on storage shelf surfaces. (Suggested surface resistivity is 10^4 to 10^8 Ω/sq; suggested resistance between surface and ground is 7.5×10^5 to 10^8 Ω.)

- For transport and temporary storage of devices, use containers (boxes, jigs or bags) that are made of anti-static materials or materials which dissipate electrostatic charge.

- Make sure that cart surfaces which come into contact with device packaging are made of materials which will conduct static electricity, and verify that they are grounded to the floor surface via a grounding chain.

- In any location where the level of static electricity is to be closely controlled, the ground resistance level should be Class 3 or above. Use different ground wires for all items of equipment which may come into physical contact with devices.

(2) Operating environment



- Operators must wear anti-static clothing and conductive shoes (or a leg or heel strap).

- Operators must wear a wrist strap grounded to earth via a resistor of about 1 MΩ.

- Soldering irons must be grounded from iron tip to earth, and must be used only at low voltages (6 V to 24 V).

- If the tweezers you use are likely to touch the device terminals, use anti-static tweezers and in particular avoid metallic tweezers. If a charged device touches a low-resistance tool, rapid discharge can occur. When using vacuum tweezers, attach a conductive chucking pat to the tip, and connect it to a dedicated ground used especially for anti-static purposes (suggested resistance value: 10^4 to 10^8 Ω).

- Do not place devices or their containers near sources of strong electrical fields (such as above a CRT).

- When storing printed circuit boards which have devices mounted on them, use a board container or bag that is protected against static charge. To avoid the occurrence of static charge or discharge due to friction, keep the boards separate from one other and do not stack them directly on top of one another.

- Ensure, if possible, that any articles (such as clipboards) which are brought to any location where the level of static electricity must be closely controlled are constructed of anti-static materials.

- In cases where the human body comes into direct contact with a device, be sure to wear anti-static finger covers or gloves (suggested resistance value: 10^8 Ω or less).

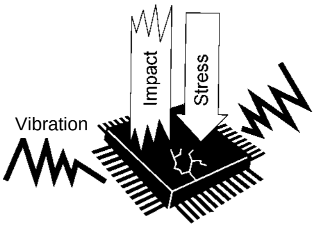
- Equipment safety covers installed near devices should have resistance ratings of 10^9 Ω or less.

- If a wrist strap cannot be used for some reason, and there is a possibility of imparting friction to devices, use an ionizer.

- The transport film used in TCP products is manufactured from materials in which static charges tend to build up. When using these products, install an ionizer to prevent the film from being charged with static electricity. Also, ensure that no static electricity will be applied to the product's copper foils by taking measures to prevent static occuring in the peripheral equipment.

3.1.2 Vibration, impact and stress

Handle devices and packaging materials with care. To avoid damage to devices, do not toss or drop packages. Ensure that devices are not subjected to mechanical vibration or shock during transportation. Ceramic package devices and devices in canister-type packages which have empty space inside them are subject to damage from vibration and shock because the bonding wires are secured only at their ends.



Plastic molded devices, on the other hand, have a relatively high level of resistance to vibration and mechanical shock because their bonding wires are enveloped and fixed in resin. However, when any device or package type is installed in target equipment, it is to some extent susceptible to wiring disconnections and other damage from vibration, shock and stressed solder junctions. Therefore when devices are incorporated into the design of equipment which will be subject to vibration, the structural design of the equipment must be thought out carefully.

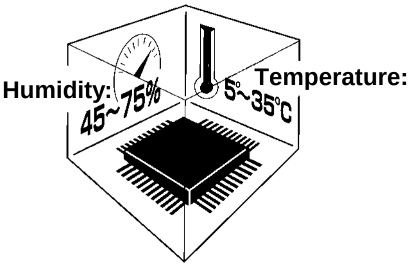
If a device is subjected to especially strong vibration, mechanical shock or stress, the package or the chip itself may crack. In products such as CCDs which incorporate window glass, this could cause surface flaws in the glass or cause the connection between the glass and the ceramic to separate.

Furthermore, it is known that stress applied to a semiconductor device through the package changes the resistance characteristics of the chip because of piezoelectric effects. In analog circuit design attention must be paid to the problem of package stress as well as to the dangers of vibration and shock as described above.

3.2 Storage

3.2.1 General storage

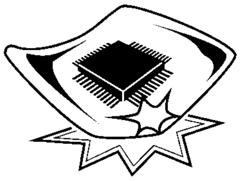
- Avoid storage locations where devices will be exposed to moisture or direct sunlight.
- Follow the instructions printed on the device cartons regarding transportation and storage.
- The storage area temperature should be kept within a temperature range of 5°C to 35°C, and relative humidity should be maintained at between 45% and 75%.
- Do not store devices in the presence of harmful (especially corrosive) gases, or in dusty conditions.



- Use storage areas where there is minimal temperature fluctuation. Rapid temperature changes can cause moisture to form on stored devices, resulting in lead oxidation or corrosion. As a result, the solderability of the leads will be degraded.
- When repacking devices, use anti-static containers.
- Do not allow external forces or loads to be applied to devices while they are in storage.
- If devices have been stored for more than two years, their electrical characteristics should be tested and their leads should be tested for ease of soldering before they are used.

3.2.2 Moisture-proof packing

Moisture-proof packing should be handled with care. The handling procedure specified for each packing type should be followed scrupulously. If the proper procedures are not followed, the quality and reliability of devices may be degraded. This section describes general precautions for handling moisture-proof packing. Since the details may differ from device to device, refer also to the relevant individual datasheets or databook.



(1) General precautions
Follow the instructions printed on the device cartons regarding transportation and storage.

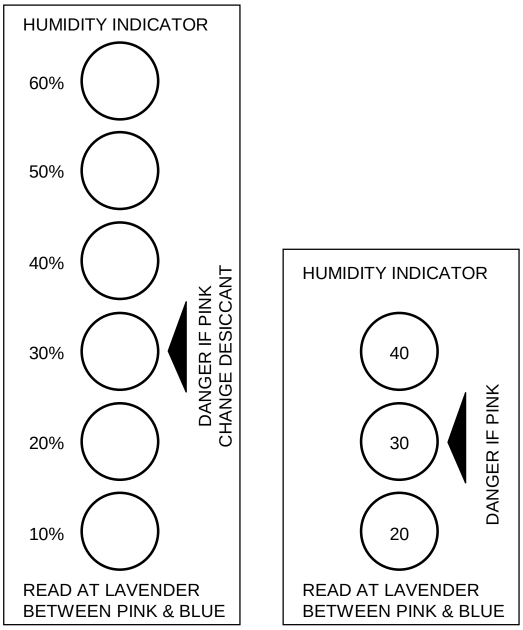
- Do not drop or toss device packing. The laminated aluminum material in it can be rendered ineffective by rough handling.
- The storage area temperature should be kept within a temperature range of 5°C to 30°C, and relative humidity should be maintained at 90% (max). Use devices within 12 months of the date marked on the package seal.

- If the 12-month storage period has expired, or if the 30% humidity indicator shown in Figure 1 is pink when the packing is opened, it may be advisable, depending on the device and packing type, to back the devices at high temperature to remove any moisture. Please refer to the table below. After the pack has been opened, use the devices in a 5°C to 30°C. 60% RH environment and within the effective usage period listed on the moisture-proof package. If the effective usage period has expired, or if the packing has been stored in a high-humidity environment, bake the devices at high temperature.

Packing	Moisture removal
Tray	If the packing bears the "Heatproof" marking or indicates the maximum temperature which it can withstand, bake at 125°C for 20 hours. (Some devices require a different procedure.)
Tube	Transfer devices to trays bearing the "Heatproof" marking or indicating the temperature which they can withstand, or to aluminum tubes before baking at 125°C for 20 hours.
Tape	Deviced packed on tape cannot be baked and must be used within the effective usage period after unpacking, as specified on the packing.

- When baking devices, protect the devices from static electricity.

- Moisture indicators can detect the approximate humidity level at a standard temperature of 25°C. 6-point indicators and 3-point indicators are currently in use, but eventually all indicators will be 3-point indicators.



(a) 6-point indicator

(b) 3-point indicator

Figure 1 Humidity indicator

3.3 Design

Care must be exercised in the design of electronic equipment to achieve the desired reliability. It is important not only to adhere to specifications concerning absolute maximum ratings and recommended operating conditions, it is also important to consider the overall environment in which equipment will be used, including factors such as the ambient temperature, transient noise and voltage and current surges, as well as mounting conditions which affect device reliability. This section describes some general precautions which you should observe when designing circuits and when mounting devices on printed circuit boards.

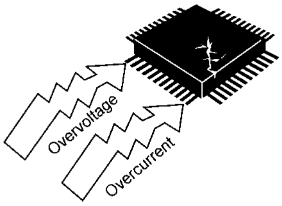
For more detailed information about each product family, refer to the relevant individual technical datasheets available from Toshiba.

3.3.1 Absolute maximum ratings

⚠CAUTION

Do not use devices under conditions in which their absolute maximum ratings (e.g. current, voltage, power dissipation or temperature) will be exceeded. A device may break down or its performance may be degraded, causing it to catch fire or explode resulting in injury to the user.

The absolute maximum ratings are rated values which must not be exceeded during operation, even for an instant. Although absolute maximum ratings differ from product to product, they essentially concern the voltage and current at each pin, the allowable power dissipation, and the junction and storage temperatures.



If the voltage or current on any pin exceeds the absolute maximum rating, the device's internal circuitry can become degraded. In the worst case, heat generated in internal circuitry can fuse wiring or cause the semiconductor chip to break down.

If storage or operating temperatures exceed rated values, the package seal can deteriorate or the wires can become disconnected due to the differences between the thermal expansion coefficients of the materials from which the device is constructed.

3.3.2 Recommended operating conditions

The recommended operating conditions for each device are those necessary to guarantee that the device will operate as specified in the datasheet.
If greater reliability is required, derate the device's absolute maximum ratings for voltage, current, power and temperature before using it.

3.3.3 Derating

When incorporating a device into your design, reduce its rated absolute maximum voltage, current, power dissipation and operating temperature in order to ensure high reliability.
Since derating differs from application to application, refer to the technical datasheets available for the various devices used in your design.

3.3.4 Unused pins

If unused pins are left open, some devices can exhibit input instability problems, resulting in malfunctions such as abrupt increase in current flow. Similarly, if the unused output pins on a device are connected to the power supply pin, the ground pin or to other output pins, the IC may malfunction or break down.

Since the details regarding the handling of unused pins differ from device to device and from pin to pin, please follow the instructions given in the relevant individual datasheets or databook.

CMOS logic IC inputs, for example, have extremely high impedance. If an input pin is left open, it can easily pick up extraneous noise and become unstable. In this case, if the input voltage level reaches an intermediate level, it is possible that both the P-channel and N-channel transistors will be turned on, allowing unwanted supply current to flow. Therefore, ensure that the unused input pins of a device are connected to the power supply (Vcc) pin or ground (GND) pin of the same device. For details of what to do with the pins of heat sinks, refer to the relevant technical datasheet and databook.

3.3.5 Latch-up

Latch-up is an abnormal condition inherent in CMOS devices, in which Vcc gets shorted to ground. This happens when a parasitic PN-PN junction (thyristor structure) internal to the CMOS chip is turned on, causing a large current of the order of several hundred mA or more to flow between Vcc and GND, eventually causing the device to break down.

Latch-up occurs when the input or output voltage exceeds the rated value, causing a large current to flow in the internal chip, or when the voltage on the Vcc (Vdd) pin exceeds its rated value, forcing the internal chip into a breakdown condition. Once the chip falls into the latch-up state, even though the excess voltage may have been applied only for an instant, the large current continues to flow between Vcc (Vdd) and GND (Vss). This causes the device to heat up and, in extreme cases, to emit gas fumes as well. To avoid this problem, observe the following precautions:

(1) Do not allow voltage levels on the input and output pins either to rise above Vcc (Vdd) or to fall below GND (Vss). Also, follow any prescribed power-on sequence, so that power is applied gradually or in steps rather than abruptly.

(2) Do not allow any abnormal noise signals to be applied to the device.

(3) Set the voltage levels of unused input pins to Vcc (Vdd) or GND (Vss).

(4) Do not connect output pins to one another.

3.3.6 Input/Output protection

Wired-AND configurations, in which outputs are connected together, cannot be used, since this short-circuits the outputs. Outputs should, of course, never be connected to Vcc (Vdd) or GND (Vss).

Furthermore, ICs with tri-state outputs can undergo performance degradation if a shorted output current is allowed to flow for an extended period of time. Therefore, when designing circuits, make sure that tri-state outputs will not be enabled simultaneously.

3.3.7 Load capacitance

Some devices display increased delay times if the load capacitance is large. Also, large charging and discharging currents will flow in the device, causing noise. Furthermore, since outputs are shorted for a relatively long time, wiring can become fused.

Consult the technical information for the device being used to determine the recommended load capacitance.

3.3.8 Thermal design

The failure rate of semiconductor devices is greatly increased as operating temperatures increase. As shown in Figure 2, the internal thermal stress on a device is the sum of the ambient temperature and the temperature rise due to power dissipation in the device. Therefore, to achieve optimum reliability, observe the following precautions concerning thermal design:

(1) Keep the ambient temperature (Ta) as low as possible.

(2) If the device's dynamic power dissipation is relatively large, select the most appropriate circuit board material, and consider the use of heat sinks or of forced air cooling. Such measures will help lower the thermal resistance of the package.

(3) Derate the device's absolute maximum ratings to minimize thermal stress from power dissipation.

$\theta ja = \theta jc + \theta ca$

$\theta ja = (Tj - Ta) / P$

$\theta jc = (Tj - Tc) / P$

$\theta ca = (Tc - Ta) / P$

in which θja = thermal resistance between junction and surrounding air (°C/W)

θjc = thermal resistance between junction and package surface, or internal thermal resistance (°C/W)

θca = thermal resistance between package surface and surrounding air, or external thermal resistance (°C/W)

Tj = junction temperature or chip temperature (°C)

Tc = package surface temperature or case temperature (°C)

Ta = ambient temperature (°C)

P = power dissipation (W)

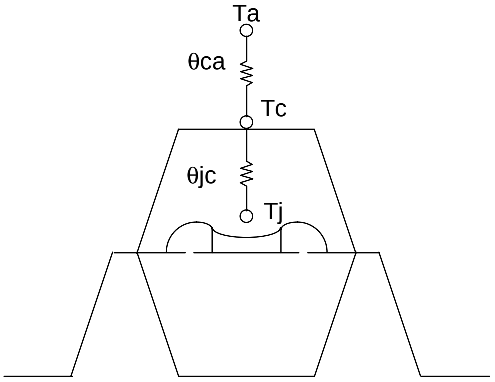


Figure 2 Thermal resistance of package

3.3.9 Interfacing

When connecting inputs and outputs between devices, make sure input voltage (VIL/VIH) and output voltage (VOL/VOH) levels are matched. Otherwise, the devices may malfunction. When connecting devices operating at different supply voltages, such as in a dual-power-supply system, be aware that erroneous power-on and power-off sequences can result in device breakdown. For details of how to interface particular devices, consult the relevant technical datasheets and databooks. If you have any questions or doubts about interfacing, contact your nearest Toshiba office or distributor.

3.3.10 Decoupling

Spike currents generated during switching can cause Vcc (Vdd) and GND (Vss) voltage levels to fluctuate, causing ringing in the output waveform or a delay in response speed. (The power supply and GND wiring impedance is normally 50 Ω to 100 Ω.) For this reason, the impedance of power supply lines with respect to high frequencies must be kept low. This can be accomplished by using thick and short wiring for the Vcc (Vdd) and GND (Vss) lines and by installing decoupling capacitors (of approximately 0.01 µF to 1 µF capacitance) as high-frequency filters between Vcc (Vdd) and GND (Vss) at strategic locations on the printed circuit board.

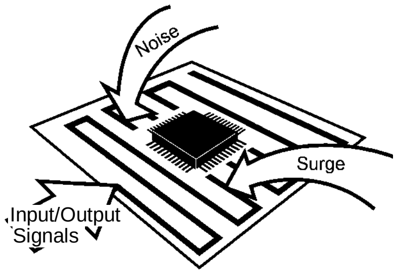
For low-frequency filtering, it is a good idea to install a 10- to 100-µF capacitor on the printed circuit board (one capacitor will suffice). If the capacitance is excessively large, however, (e.g. several thousand µF) latch-up can be a problem. Be sure to choose an appropriate capacitance value.

An important point about wiring is that, in the case of high-speed logic ICs, noise is caused mainly by reflection and crosstalk, or by the power supply impedance. Reflections cause increased signal delay, ringing, overshoot and undershoot, thereby reducing the device's safety margins with respect to noise. To prevent reflections, reduce the wiring length by increasing the device mounting density so as to lower the inductance (L) and capacitance (C) in the wiring. Extreme care must be taken, however, when taking this corrective measure, since it tends to cause crosstalk between the wires. In practice, there must be a trade-off between these two factors.

3.3.11 External noise

Printed circuit boards with long I/O or signal pattern lines are vulnerable to induced noise or surges from outside sources. Consequently, malfunctions or breakdowns can result from overcurrent or overvoltage, depending on the types of device used. To protect against noise, lower the impedance of the pattern line or insert a noise-canceling circuit. Protective measures must also be taken against surges.

For details of the appropriate protective measures for a particular device, consult the relevant databook.



3.3.12 Electromagnetic interference

Widespread use of electrical and electronic equipment in recent years has brought with it radio and TV reception problems due to electromagnetic interference. To use the radio spectrum effectively and to maintain radio communications quality, each country has formulated regulations limiting the amount of electromagnetic interference which can be generated by individual products.

Electromagnetic interference includes conduction noise propagated through power supply and telephone lines, and noise from direct electromagnetic waves radiated by equipment. Different measurement methods and corrective measures are used to assess and counteract each specific type of noise.

Difficulties in controlling electromagnetic interference derive from the fact that there is no method available which allows designers to calculate, at the design stage, the strength of the electromagnetic waves which will emanate from each component in a piece of equipment. For this reason, it is only after the prototype equipment has been completed that the designer can take measurements using a dedicated instrument to determine the strength of electromagnetic interference waves. Yet it is possible during system design to incorporate some measures for the prevention of electromagnetic interference, which can facilitate taking corrective measures once the design has been completed. These include installing shields and noise filters, and increasing the thickness of the power supply wiring patterns on the printed circuit board. One effective method, for example, is to devise several shielding options during design, and then select the most suitable shielding method based on the results of measurements taken after the prototype has been completed.

3.3.13 Peripheral circuits

In most cases semiconductor devices are used with peripheral circuits and components. The input and output signal voltages and currents in these circuits must be chosen to match the semiconductor device's specifications. The following factors must be taken into account.

(1) Inappropriate voltages or currents applied to a device's input pins may cause it to operate erratically. Some devices contain pull-up or pull-down resistors. When designing your system, remember to take the effect of this on the voltage and current levels into account.

(2) The output pins on a device have a predetermined external circuit drive capability. If this drive capability is greater than that required, either incorporate a compensating circuit into your design or carefully select suitable components for use in external circuits.

3.3.14 Safety standards

Each country has safety standards which must be observed. These safety standards include requirements for quality assurance systems and design of device insulation. Such requirements must be fully taken into account to ensure that your design conforms to the applicable safety standards.

3.3.15 Other precautions

(1) When designing a system, be sure to incorporate fail-safe and other appropriate measures according to the intended purpose of your system. Also, be sure to debug your system under actual board-mounted conditions.

(2) If a plastic-package device is placed in a strong electric field, surface leakage may occur due to the charge-up phenomenon, resulting in device malfunction. In such cases take appropriate measures to prevent this problem, for example by protecting the package surface with a conductive shield.

(3) With some microcomputers and MOS memory devices, caution is required when powering on or resetting the device. To ensure that your design does not violate device specifications, consult the relevant databook for each constituent device.

(4) Ensure that no conductive material or object (such as a metal pin) can drop onto and short the leads of a device mounted on a printed circuit board.

3.4 Inspection, Testing and Evaluation

3.4.1 Grounding

⚠CAUTION

Ground all measuring instruments, jigs, tools and soldering irons to earth. Electrical leakage may cause a device to break down or may result in electric shock.

3.4.2 Inspection Sequence

⚠CAUTION

① Do not insert devices in the wrong orientation. Make sure that the positive and negative electrodes of the power supply are correctly connected. Otherwise, the rated maximum current or maximum power dissipation may be exceeded and the device may break down or undergo performance degradation, causing it to catch fire or explode, resulting in injury to the user.

② When conducting any kind of evaluation, inspection or testing using AC power with a peak voltage of 42.4 V or DC power exceeding 60 V, be sure to connect the electrodes or probes of the testing equipment to the device under test before powering it on. Connecting the electrodes or probes of testing equipment to a device while it is powered on may result in electric shock, causing injury.

(1) Apply voltage to the test jig only after inserting the device securely into it. When applying or removing power, observe the relevant precautions, if any.

(2) Make sure that the voltage applied to the device is off before removing the device from the test jig. Otherwise, the device may undergo performance degradation or be destroyed.

(3) Make sure that no surge voltages from the measuring equipment are applied to the device.

(4) The chips housed in tape carrier packages (TCPs) are bare chips and are therefore exposed. During inspection take care not to crack the chip or cause any flaws in it.
Electrical contact may also cause a chip to become faulty. Therefore make sure that nothing comes into electrical contact with the chip.

3.5 Mounting

There are essentially two main types of semiconductor device package: lead insertion and surface mount. During mounting on printed circuit boards, devices can become contaminated by flux or damaged by thermal stress from the soldering process. With surface-mount devices in particular, the most significant problem is thermal stress from solder reflow, when the entire package is subjected to heat. This section describes a recommended temperature profile for each mounting method, as well as general precautions which you should take when mounting devices on printed circuit boards. Note, however, that even for devices with the same package type, the appropriate mounting method varies according to the size of the chip and the size and shape of the lead frame. Therefore, please consult the relevant technical datasheet and databook.

3.5.1 Lead forming

⚠CAUTION

① Always wear protective glasses when cutting the leads of a device with clippers or a similar tool. If you do not, small bits of metal flying off the cut ends may damage your eyes.

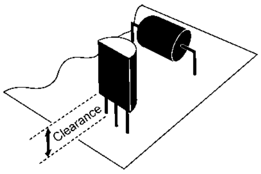
② Do not touch the tips of device leads. Because some types of device have leads with pointed tips, you may prick your finger.

Semiconductor devices must undergo a process in which the leads are cut and formed before the devices can be mounted on a printed circuit board. If undue stress is applied to the interior of a device during this process, mechanical breakdown or performance degradation can result. This is attributable primarily to differences between the stress on the device's external leads and the stress on the internal leads. If the relative difference is great enough, the device's internal leads, adhesive properties or sealant can be damaged. Observe these precautions during the lead-forming process (this does not apply to surface-mount devices):

(1) Lead insertion hole intervals on the printed circuit board should match the lead pitch of the device precisely.

(2) If lead insertion hole intervals on the printed circuit board do not precisely match the lead pitch of the device, do not attempt to forcibly insert devices by pressing on them or by pulling on their leads.

(3) For the minimum clearance specification between a device and a printed circuit board, refer to the relevant device's datasheet and databook. If necessary, achieve the required clearance by forming the device's leads appropriately. Do not use the spacers which are used to raise devices above the surface of the printed circuit board during soldering to achieve clearance. These spacers normally continue to expand due to heat, even after the solder has begun to solidify; this applies severe stress to the device.



(4) Observe the following precautions when forming the leads of a device prior to mounting.

- Use a tool or jig to secure the lead at its base (where the lead meets the device package) while bending so as to avoid mechanical stress to the device. Also avoid bending or stretching device leads repeatedly.
- Be careful not to damage the lead during lead forming.
- Follow any other precautions described in the individual datasheets and databooks for each device and package type.

3.5.2 Socket mounting

(1) When socket mounting devices on a printed circuit board, use sockets which match the inserted device's package.

(2) Use sockets whose contacts have the appropriate contact pressure. If the contact pressure is insufficient, the socket may not make a perfect contact when the device is repeatedly inserted and removed; if the pressure is excessively high, the device leads may be bent or damaged when they are inserted into or removed from the socket.

(3) When soldering sockets to the printed circuit board, use sockets whose construction prevents flux from penetrating into the contacts or which allows flux to be completely cleaned off.

(4) Make sure the coating agent applied to the printed circuit board for moisture-proofing purposes does not stick to the socket contacts.

(5) If the device leads are severely bent by a socket as it is inserted or removed and you wish to repair the leads so as to continue using the device, make sure that this lead correction is only performed once. Do not use devices whose leads have been corrected more than once.

(6) If the printed circuit board with the devices mounted on it will be subjected to vibration from external sources, use sockets which have a strong contact pressure so as to prevent the sockets and devices from vibrating relative to one another.

3.5.3 Soldering temperature profile

The soldering temperature and heating time vary from device to device. Therefore, when specifying the mounting conditions, refer to the individual datasheets and databooks for the devices used.

(1) Using a soldering iron

Complete soldering within ten seconds for lead temperatures of up to 260°C, or within three seconds for lead temperatures of up to 350°C.

(2) Using medium infrared ray reflow

- Heating top and bottom with long or medium infrared rays is recommended (see Figure 3).

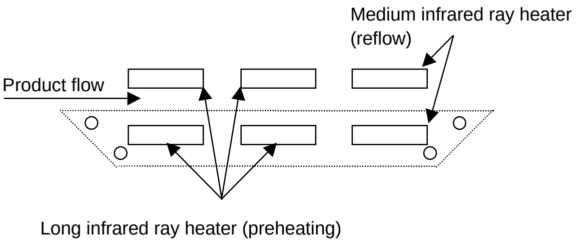

Medium infrared ray heater
(reflow)
Product flow
Long infrared ray heater (preheating)

Figure 3 Heating top and bottom with long or medium infrared rays

- Complete the infrared ray reflow process within 30 seconds at a package surface temperature of between 210°C and 240°C.
- Refer to Figure 4 for an example of a good temperature profile for infrared or hot air reflow.

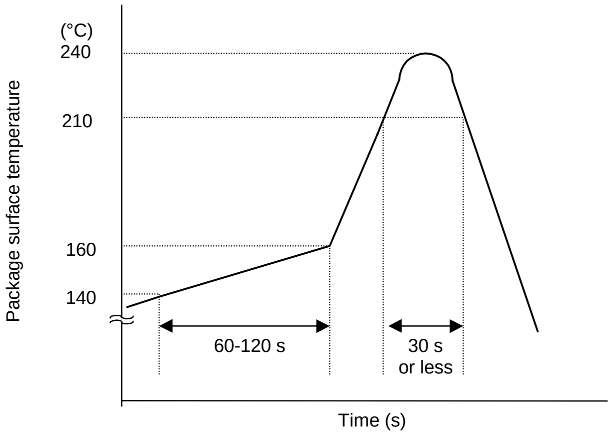


Figure 4 Sample temperature profile for infrared or hot air reflow

(3) Using hot air reflow

- Complete hot air reflow within 30 seconds at a package surface temperature of between 210°C and 240°C.
- For an example of a recommended temperature profile, refer to Figure 4 above.

(4) Using solder flow

- Apply preheating for 60 to 120 seconds at a temperature of 150°C.
- For lead insertion-type packages, complete solder flow within 10 seconds with the temperature at the stopper (or, if there is no stopper, at a location more than 1.5 mm from the body) which does not exceed 260°C.

- For surface-mount packages, complete soldering within 5 seconds at a temperature of 250°C or less in order to prevent thermal stress in the device.

- Figure 5 shows an example of a recommended temperature profile for surface-mount packages using solder flow.

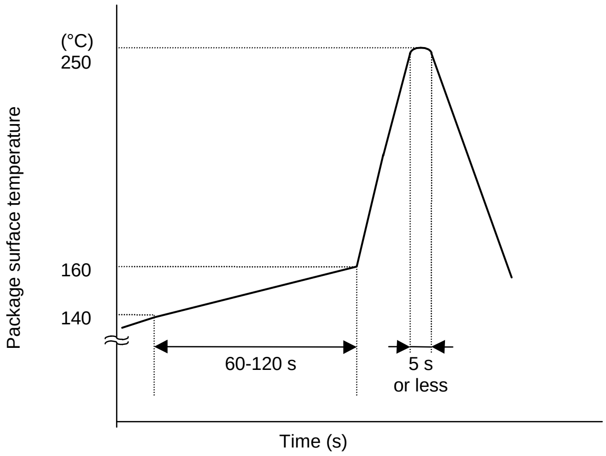


Figure 5 Sample temperature profile for solder flow

3.5.4 Flux cleaning and ultrasonic cleaning

(1) When cleaning circuit boards to remove flux, make sure that no residual reactive ions such as Na or Cl remain. Note that organic solvents react with water to generate hydrogen chloride and other corrosive gases which can degrade device performance.

(2) Washing devices with water will not cause any problems. However, make sure that no reactive ions such as sodium and chlorine are left as a residue. Also, be sure to dry devices sufficiently after washing.

(3) Do not rub device markings with a brush or with your hand during cleaning or while the devices are still wet from the cleaning agent. Doing so can rub off the markings.

(4) The dip cleaning, shower cleaning and steam cleaning processes all involve the chemical action of a solvent. Use only recommended solvents for these cleaning methods. When immersing devices in a solvent or steam bath, make sure that the temperature of the liquid is 50°C or below, and that the circuit board is removed from the bath within one minute.

(5) Ultrasonic cleaning should not be used with hermetically-sealed ceramic packages such as a leadless chip carrier (LCC), pin grid array (PGA) or charge-coupled device (CCD), because the bonding wires can become disconnected due to resonance during the cleaning process. Even if a device package allows ultrasonic cleaning, limit the duration of ultrasonic cleaning to as short a time as possible, since long hours of ultrasonic cleaning degrade the adhesion between the mold resin and the frame material. The following ultrasonic cleaning conditions are recommended:

Frequency: 27 kHz ~ 29 kHz

Ultrasonic output power: 300 W or less (0.25 W/cm^2 or less)

Cleaning time: 30 seconds or less

Suspend the circuit board in the solvent bath during ultrasonic cleaning in such a way that the ultrasonic vibrator does not come into direct contact with the circuit board or the device.

3.5.5 No cleaning

If analog devices or high-speed devices are used without being cleaned, flux residues may cause minute amounts of leakage between pins. Similarly, dew condensation, which occurs in environments containing residual chlorine when power to the device is on, may cause between-lead leakage or migration. Therefore, Toshiba recommends that these devices be cleaned. However, if the flux used contains only a small amount of halogen (0.05W% or less), the devices may be used without cleaning without any problems.

3.5.6 Mounting tape carrier packages (TCPs)

(1) When tape carrier packages (TCPs) are mounted, measures must be taken to prevent electrostatic breakdown of the devices.

(2) If devices are being picked up from tape, or outer lead bonding (OLB) mounting is being carried out, consult the manufacturer of the insertion machine which is being used, in order to establish the optimum mounting conditions in advance and to avoid any possible hazards.

(3) The base film, which is made of polyimide, is hard and thin. Be careful not to cut or scratch your hands or any objects while handling the tape.

(4) When punching tape, try not to scatter broken pieces of tape too much.

(5) Treat the extra film, reels and spacers left after punching as industrial waste, taking care not to destroy or pollute the environment.

(6) Chips housed in tape carrier packages (TCPs) are bare chips and therefore have their reverse side exposed. To ensure that the chip will not be cracked during mounting, ensure that no mechanical shock is applied to the reverse side of the chip. Electrical contact may also cause a chip to fail. Therefore, when mounting devices, make sure that nothing comes into electrical contact with the reverse side of the chip.
If your design requires connecting the reverse side of the chip to the circuit board, please consult Toshiba or a Toshiba distributor beforehand.

3.5.7 Mounting chips

Devices delivered in chip form tend to degrade or break under external forces much more easily than plastic-packaged devices. Therefore, caution is required when handling this type of device.

(1) Mount devices in a properly prepared environment so that chip surfaces will not be exposed to polluted ambient air or other polluted substances.

(2) When handling chips, be careful not to expose them to static electricity.
In particular, measures must be taken to prevent static damage during the mounting of chips. With this in mind, Toshiba recommend mounting all peripheral parts first and then mounting chips last (after all other components have been mounted).

(3) Make sure that PCBs (or any other kind of circuit board) on which chips are being mounted do not have any chemical residues on them (such as the chemicals which were used for etching the PCBs).

(4) When mounting chips on a board, use the method of assembly that is most suitable for maintaining the appropriate electrical, thermal and mechanical properties of the semiconductor devices used.

* For details of devices in chip form, refer to the relevant device’s individual datasheets.

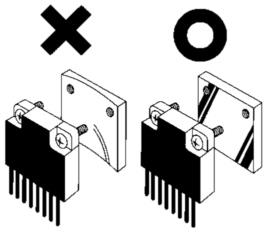
3.5.8 Circuit board coating

When devices are to be used in equipment requiring a high degree of reliability or in extreme environments (where moisture, corrosive gas or dust is present), circuit boards may be coated for protection. However, before doing so, you must carefully consider the possible stress and contamination effects that may result and then choose the coating resin which results in the minimum level of stress to the device.

3.5.9 Heat sinks

(1) When attaching a heat sink to a device, be careful not to apply excessive force to the device in the process.

(2) When attaching a device to a heat sink by fixing it at two or more locations, evenly tighten all the screws in stages (i.e. do not fully tighten one screw while the rest are still only loosely tightened). Finally, fully tighten all the screws up to the specified torque.



(3) Drill holes for screws in the heat sink exactly as specified. Smooth the surface by removing burrs and protrusions or indentations which might interfere with the installation of any part of the device.

(4) A coating of silicone compound can be applied between the heat sink and the device to improve heat conductivity. Be sure to apply the coating thinly and evenly; do not use too much. Also, be sure to use a non-volatile compound, as volatile compounds can crack after a time, causing the heat radiation properties of the heat sink to deteriorate.

(5) If the device is housed in a plastic package, use caution when selecting the type of silicone compound to be applied between the heat sink and the device. With some types, the base oil separates and penetrates the plastic package, significantly reducing the useful life of the device.
Two recommended silicone compounds in which base oil separation is not a problem are YG6260 from Toshiba Silicone.

(6) Heat-sink-equipped devices can become very hot during operation. Do not touch them, or you may sustain a burn.

3.5.10 Tightening torque

(1) Make sure the screws are tightened with fastening torques not exceeding the torque values stipulated in individual datasheets and databooks for the devices used.

(2) Do not allow a power screwdriver (electrical or air-driven) to touch devices.

3.5.11 Repeated device mounting and usage

Do not remount or re-use devices which fall into the categories listed below; these devices may cause significant problems relating to performance and reliability.

(1) Devices which have been removed from the board after soldering

(2) Devices which have been inserted in the wrong orientation or which have had reverse current applied

(3) Devices which have undergone lead forming more than once

3.6 Protecting Devices in the Field

3.6.1 Temperature

Semiconductor devices are generally more sensitive to temperature than are other electronic components. The various electrical characteristics of a semiconductor device are dependent on the ambient temperature at which the device is used. It is therefore necessary to understand the temperature characteristics of a device and to incorporate device derating into circuit design. Note also that if a device is used above its maximum temperature rating, device deterioration is more rapid and it will reach the end of its usable life sooner than expected.

3.6.2 Humidity

Resin-molded devices are sometimes improperly sealed. When these devices are used for an extended period of time in a high-humidity environment, moisture can penetrate into the device and cause chip degradation or malfunction. Furthermore, when devices are mounted on a regular printed circuit board, the impedance between wiring components can decrease under high-humidity conditions. In systems which require a high signal-source impedance, circuit board leakage or leakage between device lead pins can cause malfunctions. The application of a moisture-proof treatment to the device surface should be considered in this case. On the other hand, operation under low-humidity conditions can damage a device due to the occurrence of electrostatic discharge. Unless damp-proofing measures have been specifically taken, use devices only in environments with appropriate ambient moisture levels (i.e. within a relative humidity range of 40% to 60%).

3.6.3 Corrosive gases

Corrosive gases can cause chemical reactions in devices, degrading device characteristics.
For example, sulphur-bearing corrosive gases emanating from rubber placed near a device (accompanied by condensation under high-humidity conditions) can corrode a device’s leads. The resulting chemical reaction between leads forms foreign particles which can cause electrical leakage.

3.6.4 Radioactive and cosmic rays

Most industrial and consumer semiconductor devices are not designed with protection against radioactive and cosmic rays. Devices used in aerospace equipment or in radioactive environments must therefore be shielded.

3.6.5 Strong electrical and magnetic fields

Devices exposed to strong magnetic fields can undergo a polarization phenomenon in their plastic material, or within the chip, which gives rise to abnormal symptoms such as impedance changes or increased leakage current. Failures have been reported in LSIs mounted near malfunctioning deflection yokes in TV sets. In such cases the device’s installation location must be changed or the device must be shielded against the electrical or magnetic field. Shielding against magnetism is especially necessary for devices used in an alternating magnetic field because of the electromotive forces generated in this type of environment.

3.6.6 Interference from light (ultraviolet rays, sunlight, fluorescent lamps and incandescent lamps)

Light striking a semiconductor device generates electromotive force due to photoelectric effects. In some cases the device can malfunction. This is especially true for devices in which the internal chip is exposed. When designing circuits, make sure that devices are protected against incident light from external sources. This problem is not limited to optical semiconductors and EPROMs. All types of device can be affected by light.

3.6.7 Dust and oil

Just like corrosive gases, dust and oil can cause chemical reactions in devices, which will adversely affect a device's electrical characteristics. To avoid this problem, do not use devices in dusty or oily environments. This is especially important for optical devices because dust and oil can affect a device's optical characteristics as well as its physical integrity and the electrical performance factors mentioned above.

3.6.8 Fire

Semiconductor devices are combustible; they can emit smoke and catch fire if heated sufficiently. When this happens, some devices may generate poisonous gases. Devices should therefore never be used in close proximity to an open flame or a heat-generating body, or near flammable or combustible materials.

3.7 Disposal of Devices and Packing Materials

When discarding unused devices and packing materials, follow all procedures specified by local regulations in order to protect the environment against contamination.

4. Precautions and Usage Considerations

This section describes matters specific to each product group which need to be taken into consideration when using devices. If the same item is described in Sections 3 and 4, the description in Section 4 takes precedence.

4.1 Microcontrollers

4.1.1 Design

(1) Using resonators which are not specifically recommended for use

Resonators recommended for use with Toshiba products in microcontroller oscillator applications are listed in Toshiba databooks along with information about oscillation conditions. If you use a resonator not included in this list, please consult Toshiba or the resonator manufacturer concerning the suitability of the device for your application.

(2) Undefined functions

In some microcontrollers certain instruction code values do not constitute valid processor instructions. Also, it is possible that the values of bits in registers will become undefined. Take care in your applications not to use invalid instructions or to let register bit values become undefined.

64-Bit TX System RISC
TX49/H2 Core Architecture

I TX49/H2 Processor Core Specification

1. Introduction

The TX49/H2 Processor Core is a high performance and low-power 64-bit RISC microprocessor core developed by Toshiba which is well-suited to embedded applications such as networking, laser printer, STB (Set Top Box) and 3-D graphic.

In this manual, TX49/H2 is called “TX49” hereinafter.

2. Feature

- 64 bit operation
- 32 of 64 bit integer general purpose registers
- 32 of 64 bit floating point general purpose registers
- 64 GB physical address space
- Instruction Set
    - Upward compatible with MIPS I, MIPS II, and MIPS III ISA
    - MAC (Multiply and Accumulate) instructions
    - PREF (Prefetch) instruction
- Optimized 5 stage pipeline
- Instruction Cache
    - 8 KB/ 16 KB/ 32KB : Fixed in each products
    - Four-way set associative
    - Lock function support (Way1-Way3)
- Data cache
    - 8 KB/ 16 KB/ 32 KB: Fixed in each products
    - Four-way set associative
    - Lock function support (Way1-Way3)
    - Write policies

      Write-back

      Write-through-No-Write-Allocate-Snoop

      Write-through-Write-Allocate-Snoop
- MMU
    - 48-double-entry (even/odd) Joint TLB
    - 2-entry Instruction TLB
    - 4-entry Data TLB
- IEEE754 compatible single and double precision FPU
    - Single and double precision FPU in hardware
- Debug support (EJTAG)
    - Debug instructions
    - Real time debugging is supported by debug module logic
- Power management modes (Halt, Doze)
    - WAIT instruction

3. TX49 Block Diagram

Figure 3-1 shows the block diagram of TX49 Pure Core, MPU Core and MCU. TX49 Pure Core includes an instruction cache and a data cache. These cache are selectable by user system from among a variety of possible configurations. Cache size is predetermined for each ASSP product, however.

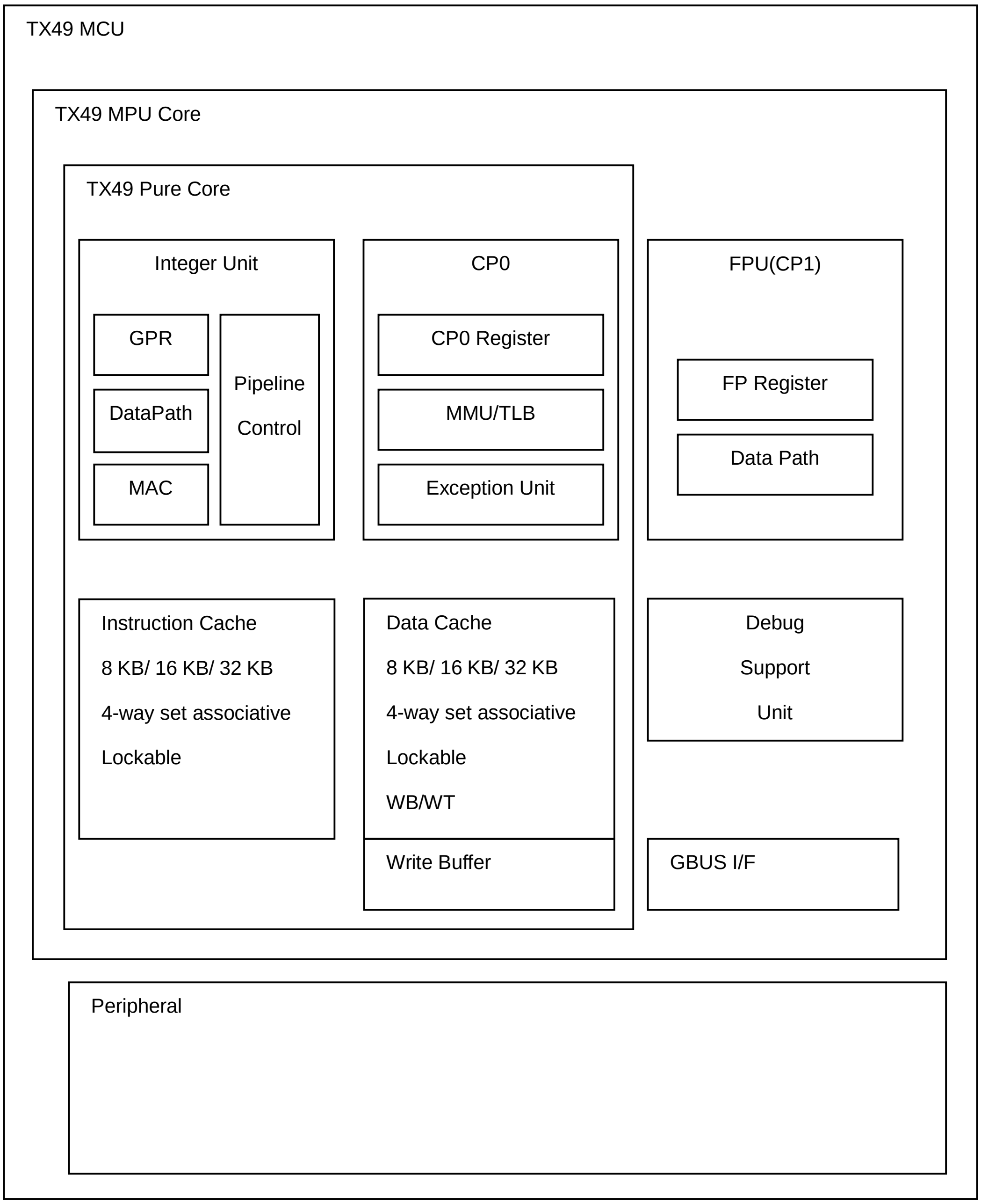


Figure 3-1 Block Diagram of the TX49

4. CPU Registers Overview

4.1 Introduction

The TX49 has the CPU registers for integer operation or address calculation and the CP0 registers for memory system or exception handling.

4.2 CPU Registers

The TX49 has the 64-bit CPU registers.

- 32 general-purpose registers
- 64-bit program counters
- HI/LO register for storing the result of multiply and divide operations

Figure 4-1 shows the configuration of these registers.

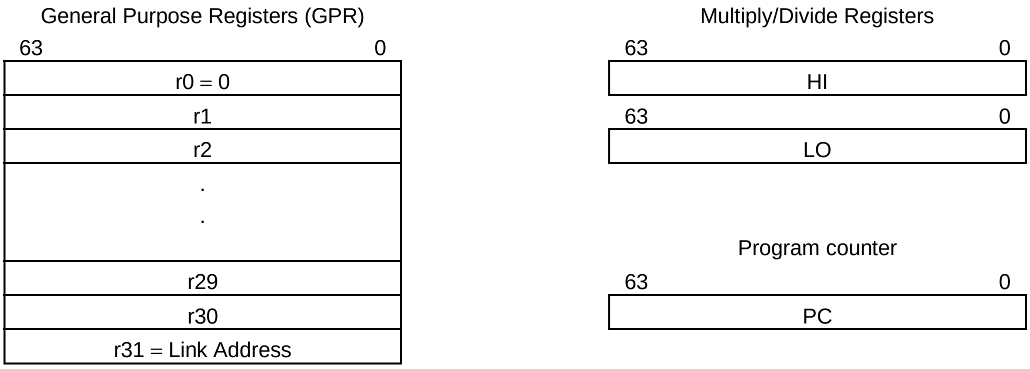


Figure 4-1 TX49 CPU registers

The r0 and r31 registers of GPR have special functions as follows.

- Register r0 always contains the value 0. It can be a target register of an instruction whose operation result is not needed. Or, it can be a source register of an instruction that requires a value of 0.
- Register r31 is the link register for the Jump and Link instruction. The address of the instruction after the delay slot is placed in r31.

The TX49 has the following some special registers that are used or modified implicitly by certain instructions.

- HI - Holds the high-order bits of the result of integer multiply operation or the remainder of integer divide operation.
- LO - Holds the low-order bits of the result of integer multiply operation or the quotient of integer divide operation.

These two registers are used to store that result of an integer multiplication or division. In multiplication, the 64 high-order bits of a 128-bit result are stored in the HI, and the 64 low-order bits are stored in the LO. In division, the resulting quotient is stored in the LO, and the remainder is stored in the HI.

- PC - Program Counter

The register contains the address of the currently executed instruction.

4.3 CP0 Registers

The TX49 has the 32-bit or 64-bit System control coprocessor(CP0) registers. These registers are used for memory system or exception handling. Table 4-1 lists the CP0 registers built into the TX49. The more detail information are described in Chapter 7.

Table 4-1 CP0 Registers

Register Name	Reg. No.	Register Name	Reg. No.
Index	Reg#0	Config	Reg#16
Random	Reg#1	LLAddr	Reg#17
EntryLo0	Reg#2	(Reserved) (Note 1)	Reg#18
EntryLo1	Reg#3	(Reserved) (Note 1)	Reg#19
Context	Reg#4	XContext	Reg#20
PageMask	Reg#5	(Reserved) (Note 1)	Reg#21
Wired	Reg#6	(Reserved) (Note 1)	Reg#22
(Reserved) (Note 1)	Reg#7	Debug (Note 2)	Reg#23
BadVAddr	Reg#8	DEPC (Note 2)	Reg#24
Count	Reg#9	(Reserved) (Note 1)	Reg#25
EntryHi	Reg#10	(Reserved) (Note 1)	Reg#26
Compare	Reg#11	(Reserved) (Note 1)	Reg#27
Status	Reg#12	TagLo	Reg#28
Cause	Reg#13	TagHi	Reg#29
EPC	Reg#14	ErrorEPC	Reg#30
PRId	Reg#15	DESAVE (Note 2)	Reg#31

Note 1: These registers are used to test the System Control Coprocessor (CP0) and should not be accessed by the user.

Note 2: These registers are exclusively used by external in-circuit emulators (ICE).

5. CPU Instruction Set Summary

5.1 Introduction

Each instruction is 32 bits long. These instructions are upward compatible with the MIPS I, II and III instruction set architecture and the TX39's instructions.

5.2 Instruction Format

There are three instruction formats: Immediate (I-type), Jump (J-type) and Register (R-type), as shown in Figure 5-1. Having just three instruction formats simplifies instruction decoding. If more complex functions or addressing modes are required, they can be produced with the compiler using combinations of the instructions.

Immediate (I-type)

31 26	25 21	20 16	15 0
op	rs	rt	immediate

Jump (J-type)

31 26	25 0
op	target

Register (R-type)

31 26	25 21	20 16	15 11	10 6	5 0
op	rs	rt	rd	sa	funct

op	Operation code (6 bits)
rs	Source register (5 bits)
rt	Target (source or destination) register, or branch condition (5 bits)
rd	Destination register (5 bits)
immediate	Immediate, branch displacement, address displacement (16 bits)
target	Branch target address (26 bits)
sa	Shift amount (5 bits)
funct	Function (6 bits)

Figure 5-1 Instruction formats and subfield mnemonics

5.3 Instruction Set Overview

5.3.1 Load and Store Instructions (Table 5-1)

Load and Store instructions move data between memory and general purpose registers, and are all I-type instructions. The only directly supported addressing mode is “base register plus 16-bit signed immediate offset”.

Table 5-1 CPU Instruction Set: Load and Store Instructions

Instruction	Description	Note
LB	Load Byte	MIPS I
LBU	Load Byte Unsigned	MIPS I
LH	Load Halfword	MIPS I
LHU	Load Halfword Unsigned	MIPS I
LW	Load Word	MIPS I
LWL	Load Word Left	MIPS I
LWR	Load Word Right	MIPS I
SB	Store Byte	MIPS I
SH	Store Halfword	MIPS I
SW	Store Word	MIPS I
SWL	Store Word Left	MIPS I
SWR	Store Word Right	MIPS I
LD	Load Doubleword	MIPS III
LDL	Load Doubleword Left	MIPS III
LDR	Load Doubleword Right	MIPS III
LL	Load Linked	MIPS II
LLD	Load Linked Doubleword	MIPS III
LWU	Load Word Unsigned	MIPS III
SC	Store Conditional	MIPS II
SCD	Store Conditional Doubleword	MIPS III
SD	Store Doubleword	MIPS III
SDL	Store Doubleword Left	MIPS III
SDR	Store Doubleword Right	MIPS III
SYNC	Sync	MIPS II

5.3.2 Computational Instructions (Table 5-2)

Computational instructions perform arithmetic, logical or shift operations on values in registers. This instruction format can be R-type or I-type. With R-type instructions, the one/two operands and the result are register values. With I-type instructions, one of the operands is 16-bit immediate data. Computational instructions can be classified as follows.

- ALU immediate
- Three-operand register-type
- Shift
- Multiply/Divide

Table 5-2 CPU Instruction Set: Computational Instructions

Instruction	Description	Note
	(ALU Immediate)	
ADDI	Add Immediate	MIPS I
ADDIU	Add Immediate Unsigned	MIPS I
SLTI	Set on Less Than Immediate	MIPS I
SLTIU	Set on Less Than Immediate Unsigned	MIPS I
ANDI	AND Immediate	MIPS I
ORI	OR Immediate	MIPS I
XORI	Exclusive OR Immediate	MIPS I
LUI	Load Upper Immediate	MIPS I
DADDI	Doubleword Add Immediate	MIPS III
DADDIU	Doubleword Add Immediate Unsigned	MIPS III
	(ALU 3-Operand, register type)	
ADD	Add	MIPS I
ADDU	Add Unsigned	MIPS I
SUB	Subtract	MIPS I
SUBU	Subtract Unsigned	MIPS I
SLT	Set on Less Than	MIPS I
SLTU	Set on Less Than Unsigned	MIPS I
AND	AND	MIPS I
OR	OR	MIPS I
XOR	Exclusive OR	MIPS I
NOR	NOR	MIPS I
DADD	Doubleword Add	MIPS III
DADDU	Doubleword Add Unsigned	MIPS III
DSUB	Doubleword Subtract	MIPS III
DSUBU	Doubleword Subtract Unsigned	MIPS III
	(Shift)	
SLL	Shift Left Logical	MIPS I
SRL	Shift Right Logical	MIPS I
SRA	Shift Right Arithmetic	MIPS I
SLLV	Shift Left Logical Variable	MIPS I
SRLV	Shift Right Logical Variable	MIPS I
SRAV	Shift Right Arithmetic Variable	MIPS I
DSLL	Doubleword Shift Left Logical	MIPS III
DSRL	Doubleword Shift Right Logical	MIPS III
DSRA	Doubleword Shift Right Arithmetic	MIPS III
DSLLV	Doubleword Shift Left Logical Variable	MIPS III
DSRLV	Doubleword Shift Right Logical Variable	MIPS III

Instruction	Description	Note
DSRAV	Doubleword Shift Right Arithmetic Variable	MIPS III
DSLL32	Doubleword Shift Left Logical +32	MIPS III
DSRL32	Doubleword Shift Right Logical +32	MIPS III
DSRA32	Doubleword Shift Right Arithmetic +32	MIPS III
	(Multiply and Divide)	
MULT	Multiply	MIPS I
MULTU	Multiply Unsigned	MIPS I
DIV	Divide	MIPS I
DIVU	Divide Unsigned	MIPS I
MFHI	Move From HI	MIPS I
MTHI	Move To HI	MIPS I
MFLO	Move From LO	MIPS I
MTLO	Move To LO	MIPS I
DMULT	Doubleword Multiply	MIPS III
DMULTU	Doubleword Multiply Unsigned	MIPS III
DDIV	Doubleword Divide	MIPS III
DDIVU	Doubleword Divide Unsigned	MIPS III

5.3.3 Jump and Branch Instructions (Table 5-3)

Jump and branch instructions change the control flow of a program. All jump and branch instructions occur with a delay of one instruction: that is, the instruction immediately following the jump or branch (this is known as the instruction in the delay slot) always executes while the target instruction is being fetched from storage. Branch-likely instructions are used for static branch prediction. The instruction in the delay slot is executed only when the branch is taken; the instruction in the delay slot is nullified if the branch is not taken.

Table 5-3 CPU Instruction Set: Jump and Branch Instructions

Instruction	Description	Note
J	Jump	MIPS I
JAL	Jump And Link	MIPS I
JR	Jump Register	MIPS I
JALR	Jump And Link Register	MIPS I
BEQ	Branch on Equal	MIPS I
BNE	Branch on Not Equal	MIPS I
BLEZ	Branch on Less Than or Equal to Zero	MIPS I
BGTZ	Branch on Greater Than Zero	MIPS I
BLTZ	Branch on Less Than Zero	MIPS I
BGEZ	Branch on Greater than or Equal to Zero	MIPS I
BLTZAL	Branch on Less Than Zero And Link	MIPS I
BGEZAL	Branch on Greater than or Equal to Zero And Link	MIPS I
BEQL	Branch on Equal Likely	MIPS II
BNEL	Branch on Not Equal Likely	MIPS II
BLEZL	Branch on Less Than or Equal to Zero Likely	MIPS II
BGTZL	Branch on Greater Than Zero Likely	MIPS II
BLTZL	Branch on Less Than Zero Likely	MIPS II
BGEZL	Branch on Greater Than or Equal to Zero Likely	MIPS II
BLTZALL	Branch on Less Than Zero And Link Likely	MIPS II
BGEZALL	Branch on Greater Than or Equal to Zero And Link Likely	MIPS II

5.3.4 Special Instructions (Table 5-4)

There are special instructions used for software trap. The instruction format is R-type for all two.

Table 5-4 CPU Instruction Set: Special Instructions

Instruction	Description	Note
SYSCALL	System Call	MIPS I
BREAK	Break	MIPS I

5.3.5 Exception Instructions (Table 5-5)

These instructions (R-type or I-type) cause a branch to the general exception handling vector based upon the result of a comparison.

Table 5-5 CPU Instruction Set: Exception Instructions

Instruction	Description	Note
TGE	Trap if Greater Than or Equal	MIPS II
TGEU	Trap if Greater Than or Equal Unsigned	MIPS II
TLT	Trap if Less Than	MIPS II
TLTU	Trap if Less Than Unsigned	MIPS II
TEQ	Trap if Equal	MIPS II
TNE	Trap if Not Equal	MIPS II
TGEI	Trap if Greater Than or Equal Immediate	MIPS II
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	MIPS II
TLTI	Trap if Less Than Immediate	MIPS II
TLTIU	Trap if Less Than Immediate Unsigned	MIPS II
TEQI	Trap if Equal Immediate	MIPS II
TNEI	Trap if Not Equal Immediate	MIPS II

5.3.6 Coprocessor Instructions (Table 5-6)

Coprocessor instructions invoke coprocessor operations. The format of these instructions depends on which coprocessor is used.

Table 5-6 CPU Instruction Set: Coprocessor Instructions

Instruction	Description	Note
LWCz	Load Word to Coprocessor z (z = 1,2)	MIPS I
SWCz	Store Word from Coprocessor z (z = 1,2)	MIPS I
MTCz	Move To Coprocessor z (z = 1,2)	MIPS I
MFCz	Move From Coprocessor z (z = 1,2)	MIPS I
CTCz	Move Control To Coprocessor z (z = 1,2)	MIPS I
CFCz	Move Control From Coprocessor z (z = 1,2)	MIPS I
COPz	Coprocessor Operation z (z = 1,2)	MIPS I
BCzT	Branch on Coprocessor z True (z = 0,1,2)	MIPS I
BCzF	Branch on Coprocessor z False (z = 0,1,2)	MIPS I
BCzTL	Branch on Coprocessor z True Likely (z = 0,1,2)	MIPS II
BCzFL	Branch on Coprocessor z False Likely (z = 0,1,2)	MIPS II
LDCz	Load Double Coprocessor z (z = 1,2)	MIPS III
SDCz	Store Double Coprocessor z (z = 1,2)	MIPS III
DMTCz	Doubleword Move To Coprocessor z (z = 1,2)	MIPS III
DMFCz	Doubleword Move From Coprocessor z (z = 1,2)	MIPS III

5.3.7 CP0 Instructions (Table 5-7)

Coprocessor 0 instructions are used for operations involving the system control coprocessor (CP0) registers, processor memory management and exception handling.

Table 5-7 Instruction Set: CP0 Instructions

Instruction	Description	Note
MTC0	Move To CP0	MIPS I
MFC0	Move From CP0	MIPS I
DMTC0	Doubleword Move To CP0	MIPS III
DMFC0	Doubleword Move From CP0	MIPS III
TLBR	Read Indexed TLB Entry	
TLBWI	Write Indexed TLB Entry	
TLBWR	Write Random TLB Entry	
TLBP	Probe TLB for Matching Entry	
CACHE	Cache	MIPS III
ERET	Exception Return	MIPS III
WAIT	Enter power management mode	

5.3.8 Multiply and Divide Instructions (Table 5-8)

Table 5-8 Extensions to the ISA: Multiply and Divide Instructions

Instruction	Description	Note
MULT	Multiply (3-operand)	
MULTU	Multiply Unsigned (3-operand)	
DMULT	Doubleword Multiply (3-operand)	
DMULTU	Doubleword Multiply Unsigned (3-operand)	
MADD	Multiply and ADD (3-operand)	
MADDU	Multiply and ADD Unsigned (3-operand)	

5.3.9 Debug Instructions (Table 5-9)

Table 5-9 Extensions to the ISA: Debug Instructions

Instruction	Description	Note
CTC0	Move Control To Coprocessor 0	
CFC0	Move Control From Coprocessor 0	
SDBBP	Software Debug Breakpoint	
DERET	Debug Exception Return	

5.3.10 Other Instructions (Table 5-10)

Table 5-10 Other Instructions

Instruction	Description	Note
PREF	Prefetch	

5.4 Instruction Execution Cycles

Because the TX49 employs the high-speed Multiply and Add Calculator (MAC), multiply instructions, such as MULT, MULTU, DMULT and DMULTU are executed faster. And, TX49 is improved the execution of divide instructions, too.

Instruction	Latency (2op/3op)	Repeat (2op/3op)
MULT 2/3 operand	4/4	1/3
MADD 2/3 operand	4/4	1/3
DMULT 2/3 operand	7/7	6/6
DIV	37	36
DDIV	69	68

5.5 Defining Access Types

Access type indicates the size of a TX49 processor data item to be loaded or stored, set by the load or store instruction opcode. Access types are defined in Table A-3.

Regardless of access type or byte ordering (endianness), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed doubleword (shown in Figure 5-2). Only the combinations shown in Figure 5-2 are permissible; other combinations cause address error exceptions. See Appendix A for individual descriptions of CPU load and store instructions.

Access Type Mnemonic (Value)	Low-Order Address Bits			Bytes Accessed															
				Big Endian (63-----------------31-----------------0) Byte								Little Endian (63-----------------31-----------------0) Byte							
	2	1	0																
Doubleword (7)	0	0	0	0	1	2	3	4	5	6	7	7	6	5	4	3	2	1	0
Septibyte (6)	0	0	0	0	1	2	3	4	5	6			6	5	4	3	2	1	0
	0	0	1		1	2	3	4	5	6	7	7	6	5	4	3	2	1	
Sextibyte (5)	0	0	0	0	1	2	3	4	5					5	4	3	2	1	0
	0	1	0			2	3	4	5	6	7	7	6	5	4	3	2		
Quintibyte (4)	0	0	0	0	1	2	3	4							4	3	2	1	0
	0	1	1				3	4	5	6	7	7	6	5	4	3			
Word (3)	0	0	0	0	1	2	3									3	2	1	0
	1	0	0					4	5	6	7	7	6	5	4				
Triplebyte (2)	0	0	0	0	1	2											2	1	0
	0	0	1		1	2	3									3	2	1	
	1	0	0					4	5	6			6	5	4				
	1	0	1						5	6	7	7	6	5					
Halfword (1)	0	0	0	0	1													1	0
	0	1	0			2	3									3	2		
	1	0	0					4	5					5	4				
	1	1	0							6	7	7	6						
Byte (0)	0	0	0	0															0
	0	0	1		1													1	
	0	1	0			2											2		
	0	1	1				3									3			
	1	0	0					4							4				
	1	0	1						5					5					
	1	1	0							6			6						
	1	1	1								7	7							

Figure 5-2 Byte Access within a Doubleword

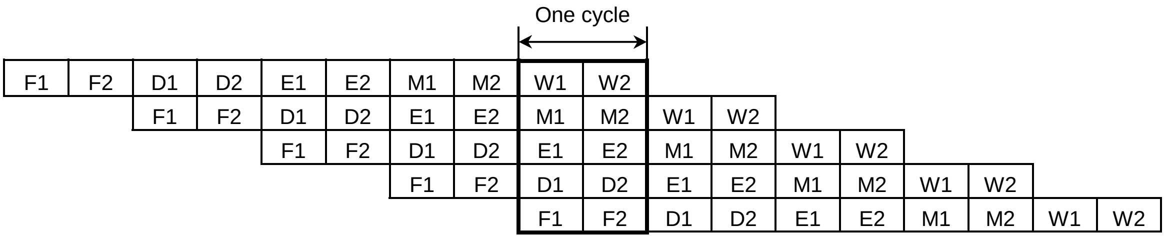
6. CPU Pipeline

6.1 Introduction

This chapter describes the operation of the TX49 pipeline. It explains the basic operation of the pipeline. And, it explains how the TX49 handled delay instructions; these are instructions that follow a branch or load instruction in the pipeline. A later section explains interruptions to the pipeline flow caused by interlocks and exceptions.

6.2 Basic Pipeline Operation

The TX49 executes instructions in an optimized 5 stage pipeline. Each pipeline stage is executed in one clock cycle. When the pipeline is fully utilized, five instructions are executed at the same time, resulting in an average instruction execution rate of one instruction par cycle as illustrated in Figure 6-1.



F1	F2	D1	D2	E1	E2	M1	M2	W1	W2								
		F1	F2	D1	D2	E1	E2	M1	M2	W1	W2						
				F1	F2	D1	D2	E1	E2	M1	M2	W1	W2				
						F1	F2	D1	D2	E1	E2	M1	M2	W1	W2		
								F1	F2	D1	D2	E1	E2	M1	M2	W1	W2

F1 - Instruction Fetch, Phase one
F2 - Instruction Fetch, Phase two
D1 - Instruction Decode, Phase one
D2 - Instruction Decode, Phase two
E1 - Execution, Phase one
E2 - Execution, Phase two
M1 - Memory Access, Phase one
M2 - Memory Access, Phase two
W1 - Write Back, Phase one
W2 - Write Back, Phase two

Figure 6-1 Pipeline stages for executing TX49 instructions

F1, F2 : Instruction Fetch

During the F1 phase the ITLB begins the virtual to physical address translation. And, during the F2 phase the instruction cache fetch and the virtual to physical address translation are completed.

D1, D2 : Instruction Decode

The instruction is decoded. Contents of the general-purpose registers are read. If the instruction involves a branch or jump, the target address is generated. The coprocessor condition signal is latched.

E1, E2 : Execution

Arithmetic, logical and shift operations are performed. The execution of multiple/divide instructions is begun.

For load and store instructions, the data virtual address is calculated, and virtual-to-physical address translation is begun.

M1, M2 : Memory Access

The data cache is accessed in the case of load and store instructions.

W1, W2 : Write Back

The result is written to a general register.

6.3 TX49 Pipeline Activities

Stage	F1	F2	D1	D2	E1	E2	M1	M2	W1	W2
Fetch		ICD	ICA	RF						
& Decode	ITLBM	ITLBR	ITC	IDEC						
ALU					ALU				WB	
Load/Store					DVA	DCAD	DCAA	DCLA		
						JTLB1	JTLB2			
							SA	DTC	WB	
									DCW	
Jump/Branch					BCMP					
				BAC		IVA				

ICD: Instruction cache address decode
ICA: Instruction cache array access
RF: Register fetch
ITLBM: Instruction address translation match
ITLBR: Instruction address translation read
ITC: Instruction tag match
IDEC: Instruction decode
ALU: ALU operation
WB: Write back to register file
DVA: Data virtual address calculation
DCAD: Data cache address decode
DCAA: Data cache array access
DCLA: Data cache load align
JTLB1: Address translation in JTLB stage1
JTLB2: Address translation in JTLB stage2
SA: Store align
DTC: Data cache tag check
DCW: Data cache write
BCMP: Branch compare
BAC: Branch address calculation
IVA: Generate instruction virtual address

6.4 Branch and Load Delay

Some TX49 instructions are executed with a delay of one instruction cycle. The cycle in which an instruction is delayed is called a delay slot. A delay occurs with load instruction and branch/jump instructions.

6.4.1 Delayed load

With load instructions, a one-cycle delay occurs while waiting for the data being loaded to become available for use by another instruction. The TX49 checks the instruction in the delay slot (the instruction immediately following the load instruction) to see if that instruction needs to use the load result; if so, it stalls the pipeline (see Figure 6-2).

LW r5, 0 (r26)	F	D	E	M	W		
ADDU r8, r7, r5		F	D	ES	E	M	W
				↑ Pipeline stall			

Figure 6-2 CPU Pipeline Load Delay

6.4.2 Delayed branching

Figure 6-3 shows the pipeline flow for jump/branch instructions. The branch target address that must be generated for these type of instructions does not become available unit the E stage - too late to be used by the instruction in the branch delay slot. The branch target instruction is fetched immediately after the branch delay slot cycle.

It is, however, possible to fetch a different instruction that would normally be executed prior to the branch instruction.

BEQ r1, r4, L1		F	D	E	M	W		
				Target addr				
subu r3, r5,r6	(delay slot)		F	D	E	M	W	
L1:addiu r7, r7, 1	(target)			F	D	E	M	W

Figure 6-3 CPU Pipeline Branch Delay

You can make effective use of the branch delay slot as follows.

- Since the instruction immediately following a branch instruction will be executed just prior to the branch, you can therefore place an instruction (that logically should be executed just before the branch) into delay slot following the branch instruction.
- The TX49 provides Branch Likely instructions in addition to the normal Branch instructions. If the branch condition of the Branch Likely instruction is met, the instruction in the delay slot is executed and the branch is taken. If the branch is not taken, the instruction in the delay is treated as a NOP.
  Therefore, Branch-Likely instructions allow the processor to execute the instruction immediately following the branch while the target instruction is being fetched.
- If no instruction is placed in the delay slot, a NOP is placed just after the branch instruction.

6.5 Non-blocking Load Function

The non-blocking load function prevents the pipeline from stalling when a cache miss occurs and a refill cycle is required to refill the data cache. Instructions after the load instruction that do not use registers affected by the load will continue to be executed. An example is shown in Figure 6-4. Here a cache miss occurs with the first load instruction. The two instructions following are executed prior to the load. The fourth instruction (ADD) must use a register that will be loaded by the load instruction, therefore the pipeline is stalled until the cache data becomes valid.

LW r3, 0(r0)	F	D	E	M	R	R	R	R	W		
ADD r6, r4, r2		F	D	E	M	W			r3		
ADD r7, r5, r2			F	D	E	M	W				
ADD r8,r9,r3				F	D	ES	ES	ES	E	M	W

R: Refill cycle, ES: Stall in E stage

Figure 6-4 Non-blocking load function

6.6 Interlock and Exception Handling

6.6.1 Overview of Interlock and Exception Handling

Smooth pipeline flow is interrupted when cache misses or exceptions occur, or when data dependencies are detected. Interruptions handled using hardware, such as cache misses, are referred to as interlocks, while those that are handled using software are called exceptions.

As shown in Figure 6-5, all interlock and exception conditions are collectively referred to as faults.

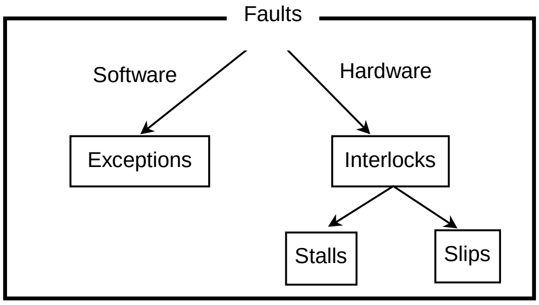


Figure 6-5 Interlocks, Exceptions, and Faults

These are two types of interlocks:

- stalls, which are resolved by halting the pipeline
- slips, which require one part of the pipeline to advance while another part of the pipeline is held static

At each cycle, exception and interlock condition corresponds to a particular pipeline stage, a condition can be traced to the particular instruction in the exception/interlock stage, as shown in Figure 6-6. For instance, an Illegal Instruction (II) exception is raised in the exception (EX) stage.

Table 6-1 and Table 6-2 describe the pipeline interlocks and exceptions listed in Figure 6-6.

State	Pipeline Stage				
	F	D	E	M	W
Stall	ITM	ICM		DCM	
				CPE	
Slip		LDI			
		MDSt			
		FCBsy			
Exception	ITLB	IBE	RI	DBE	
			Cun	NMI	
			BP	Reset	
			SC	OVF	
			DTLB	Trap	
			DTMod		
			Intr		

Figure 6-6 Correspondence of pipeline stage to interlock condition

Table 6-1 Pipeline Interlocks

Interlock	Description
ITM	Instruction TLB Miss
ICM	Instruction Cache Miss
CPE	Coprocessor Possible Exception
DCM	Data Cache Miss
LDI	Load Interlock
MDSt	Multiply / Divide Start
FCBsy	FP Coprocessor Busy

Table 6-2 Pipeline Exceptions

Exception	Description
ITLB	Instruction Translation or Address Exception
Intr	External Interrupt
IBE	Instruction Bus Error
RI	Reserved Instruction
BP	Breakpoint
SC	System Call
Cun	Coprocessor Unusable
OVF	Integer Overflow
FPE	FP Interrupt
ExTrap	EX Stage Traps
DTLB	Data Translation or Address Exception
TLBMod	TLB Modified
DBE	Data Bus Error
NMI	Nom-maskable Interrupt (or Soft Reset)
Reset	Reset

6.6.2 Exception Conditions

When an exception condition occurs, the relevant instruction and all those that follow it in the pipeline are cancelled. Accordingly, any stall conditions and any later exception conditions that may have referenced this instruction are inhibited; there is no benefit in servicing stalls for a cancelled instruction.

After instruction cancellation, a new instruction stream begins, starting execution at a predefined exception vector. System Control Coprocessor registers are loaded with information that identifies the type of exception and auxiliary information such as the virtual address at which translation exceptions occur.

6.6.3 Stall Conditions

Often, a stall condition is only detected after parts of the pipeline have advanced using incorrect data; this is called a pipeline overrun. When a stall condition is detected, all five instructions – each different stage of the pipeline – are frozen at once. In this stalled state, no pipeline stages can advance until the interlock condition is resolved. For example, when a cache miss occurs, the processor must refill the cache before it restarts the pipeline.

Once the interlock is removed, the restart sequence begins two cycles before the pipeline resumes execution. The restart sequence reverses the pipeline overrun by inserting the correct information into the pipeline.

6.6.4 External Stalls

External stall is another class of interlocks. An external stall originates outside the processor and is not referenced to a particular pipeline stage. This interlock is not affected by exceptions.

6.6.5 Interlock and Exception Timing

To prevent interlock and exception handling from adversely affecting the processor cycle time, the TX49 processor uses both logic and circuit pipeline techniques to reduce critical timing paths. Interlock and exception handling have the following effects on the pipeline:

- In some cases, the processor pipeline must be backed up (reversed and started over again from a prior stage) to recover from interlocks.
- In some cases, interlocks are serviced for instructions that will be aborted, due to an exception.

6.7 Multiply and Multiply/Add Instructions (MULT, MULTU, MADD, MADDU)

The TX49 can execute 32-bit multiply and multiply/add instructions of 2-operand continuously, and can use the results in the HI/LO registers in immediately following instructions, without pipeline stall as shown Figure 6-7. The TX49 requires three cycles to use the results of a general-purpose register as shown Figure 6-8.

MULT/MADD r3, r4	F	D	E1	E2	E3	M	W	
MULT/MADD r6, r7, r8		F	D	E1	E2	E3	M	W

Figure 6-7 MULT and MADD Instructions without data dependency (32-bit and 2-operand)

MULT/MADD r3, r4, r5	F	D	E1	E2	E3	M	W				
MULT/MADD r6, r3, r8		F	D	ES	ES	ES	E1	E2	E3	M	W

Figure 6-8 MULT and MADD Instructions with data dependency (32-bit and 3-operand)

6.8 Divide Instructions (DIV, DIVU)

Division starts from the pipeline E stage and takes 36 cycles.

Figure 6-9 shows an example of a divide instruction.

DIV/DIVU	F	D	E	M	W					
				V1	V2	V3	V4	…	V35	V36

Division stage1

Figure 6-9 DIV and DIVU Instructions

6.9 Streaming

During a cache refill operation, the TX49 can resume execution immediately after arrival of necessary data or instruction in cache even though cache refill is not completed. This is referred to as "streaming".

7. System Control Coprocessor, CP0

7.1 Introduction

The TX49 has a System Control Co-Processor (CP0). CP0 translates virtual addresses to physical addresses. CP0 manages exceptions and transitions between kernel, supervisor, and user states. CP0 also controls the cache sub-system, as well as providing diagnostic control and error recovery facilities.

7.2 CP0 Registers

This section is described about the bit field of each register. The term “coldreset” of tables shows the value of each bit when GCOLDRESET* signal is asserted. The reserved bits in description must be written the same value in reset, and return the same value when read.

7.2.1 Index register (Reg#0)

The Index register is a 32-bit read/write register containing six bits to index an entry in the TLB. The P bit of the register shows the success/failure of a TLB Probe (TLBP) instruction.

The Index register also specifies the TLB entry affected by TLB Read (TLBR) or TLB Write Index (TLBWI) instructions. Figure 7-1 shows the format of the Index register and Table 7-1 describes the Index register fields.

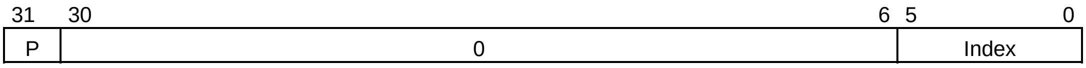


Figure 7-1 Index Register Format

Table 7-1 Index Register Field Descriptions

Bit	Field	Description	Cold Reset	Read/Write
31	P	Probe failure. Set to 1 when the previous TLB Probe (TLBP) instruction was unsuccessful.	Undefined	Read/Write
30~6	0	Reserved	0x0	Read
5~0	Index	Index to the TLB entry affected by the TLB Read and TLB Write Index instructions.	Undefined	Read/Write

7.2.2 Random register (Reg#1)

The Random register is a read only register containing six bits to index an entry in the TLB. This register decrements as each instruction executes. The values are as follows.

- A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the Wired register).
- An upper bound is set by the total number of TLB entries (47 maximum).

The Random register specifies the TLB entry affected by TLB Write Random (TLBWR) instruction. However the register doesn't need to be read for this purpose, it is readable to verify proper operation of the processor.

To simplify testing, the Random register is set to the value of the upper bound upon system reset. This register is also set to the upper bound when the Wired register is written.

Figure 7-2 shows the format of the Random register and Table 7-2 describes the Random register fields.

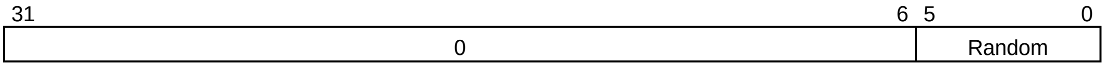


Figure 7-2 Random Register Format

Table 7-2 Random Register Field Descriptions

Bit	Field	Description	Cold Reset	Read/Write
31~6	0	Reserved.	0x0	Read
5~0	Random	TLB random index for TLBWR instruction.	Upper bound (47)	Read

7.2.3 EntryLo0 register (Reg#2) and EntryLo1 register (Reg#3)

The EntryLo register consists of two registers have identical formats:

- EntryLo0 is used for even virtual pages
- EntryLo1 is used for odd virtual pages

The EntryLo0 and EntryLo1 register are read/write register. These registers hold the physical page frame number (PFN) of the TLB entry for even and odd pages, respectively, when performing TLB read and write operations.

Figure 7-3 shows the format of the EntryLo0/EntryLo1 register and Table 7-3 describes the EntryLo0/EntryLo1 register fields.

63 32	31 30	29 6	5 3	2	1	0
0	WCE	PFN	C	D	V	G

Figure 7-3 EntryLo0/EntryLo1 Register Format

Table 7-3 EntryLo0/EntryLo1 Register Field Descriptions

Bit	Field	Description	Cold Reset	Read/Write
63~32	0	Reserved	0x0	Read
31~30	WCE	Usable for Win-CE	0x0	Read/Write
29~6	PFN	Page frame number.	Undefined	Read/Write
5~3	C	Specifies the TLB page coherency attribute. 0: Cacheable, noncoherent, write-through, no-WA 1: Cacheable, noncoherent, write-through, WA 2: Uncached 3: Cacheable,noncoherent,write-back,WA 4~7: Reserved	0x0	Read/Write
2	D	Dirty If this bit is set, the page is marked as dirty and, therefore, writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.	0	Read/Write
1	V	Valid If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS miss occurs.	0	Read/Write
0	G	Global If this bit is set in both EntryLo0 and EntryLo1, then the processor ignores the ASID during TLB lookup.	0	Read/Write

7.2.4 Context register (Reg#4)

The Context register is a read/write register containing the pointer to an entry in the page table entry (PTE) array. This array is an operating system data structure that stores virtual to physical address translations. When there is a TLB miss, the CPU loads the TLB with the missing translation from the PTE array. Normally, the operating system uses the Context register to address the current page map which resides in the kernel mapped segment,kseg3. However the contents of this register duplicates some information of the BadVAddr register, it is arranged in a form that is more useful for TLB exception handler by a software.

Figure 7-4 shows the formats of the Context register and Table 7-4 describes the Context register fields.

31 23	22 4	3 0
PTEBase	BadVPN2	0

(32-bit mode)

63 23	22 4	3 0
PTEBase	BadVPN2	0

(64-bit mode)

Figure 7-4 Context Register Formats

Table 7-4 Context Register Field Descriptions

32-bit mode

Bit	Field	Description	Cold Reset	Read/Write
31~23	PTEBase	Page table entry base pointer This field is for use by the operating system. It is normally written with a value that allows the operating system to use the Context register as a pointer into the current PTE array in memory.	Undefined	Read/Write
22~4	BadVPN2	Bad virtual address bits 31~13 This field is written by hardware on a miss. It contains the virtual page number (VPN) of the most recent virtual address that did not have a valid translation.	Undefined	Read
3~0	0	Reserved	0x0	Read

64-bit mode

Bit	Field	Description	Cold Reset	Read/Write
63~23	PTEBase	Page table entry base pointer	Undefined	Read/Write
22~4	BadVPN2	Bad virtual address bits 31~13	Undefined	Read
3~0	0	Reserved	0x0	Read

The 19-bit BadVPN2 field contains bits 31 to 13 of the virtual address that caused the TLB miss; bits 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format can directly address the pair-table of 8-byte PTEs. For other page size and PTE sizes, shifting and masking this value produces the appropriate address.

7.2.5 PageMask Register (Reg#5)

The PageMask register is a read/write register used for reading from/writing to the TLB. This register holds a comparison mask that sets the variable page size for each TLB entry.

TLB read and write operations use this register as either a source or a destination. When virtual addresses are presented for translation into physical address, the corresponding bits in the TLB identify which virtual address bits among bits 24~13 are used in the comparison. When the Mask field is not one of the values shown in Table 7-5, the operation of the TLB is undefined.

Figure 7-5 shows the format of the PageMask register and Table 7-5 describes the PageMask register fields.

31 … 25	24 … 13	12 … 0
0	MASK	0

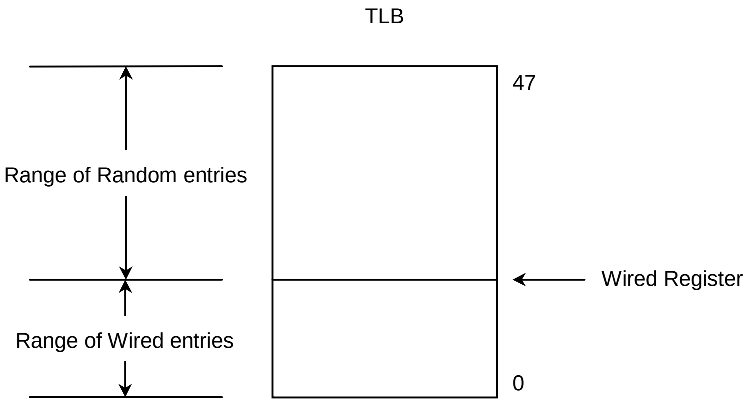
Figure 7-5 PageMask Register Format

Table 7-5 PageMask Register Field Descriptions

Bit	Field	Description	Cold Reset	Read/Write
31~25	0	Reserved	0x0	Read
24~13	MASK	Page comparison mask 000000000000: page size = 4 Kbytes 000000000011: page size = 16 Kbytes 000000001111: page size = 64 Kbytes 000000111111: page size = 256 Kbytes 000011111111: page size = 1 Mbytes 001111111111: page size = 4 Mbytes 111111111111: page size = 16 Mbytes	0x0	Read/Write
12~0	0	Reserved	0x0	Read

7.2.6 Wired Register (Reg#6)

The Wired register is a read/write register specifies the boundary between the wired and random entries of the TLB as follows. Wired entries are non-replaceable entries, which can not be overwritten by a TLB write random operation. Random entries can be overwritten.



The Wired register is set to 0 upon system reset. Writing this register also sets the Random register to the value of its upper bound. Figure 7-6 shows the format of the Wired register and Table 7-6 describes the Wired register fields.

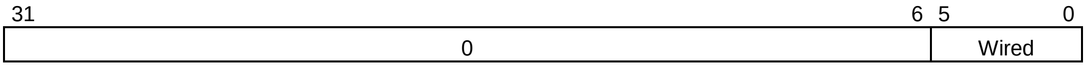


Figure 7-6 Wired Register

Table 7-6 Wired Register Filed Descriptions

Bit	Field	Description	Cold Reset	Read/Write
31~6	0	Reserved (Must be written as zeroes, and returns zeroes when read.)	0x0	Read
5~0	Wired	TLB Wired boundary.	0x0	Read/Write

7.2.7 BadVAddr Register (Reg#8)

The Bad Virtual Address (BadVAddr) register is a read only register that displays the most recent virtual address that cause one of the following exceptions; Address Error, TLB Invalid, TLB Modified and TLB Refill exceptions.

The processor does not write to this register when the EXL bit in the Status register is set to a 1. Figure 7-7 shows the formats of the BadVAddr register and Table 7-7 describes the BadVAddr register fields.

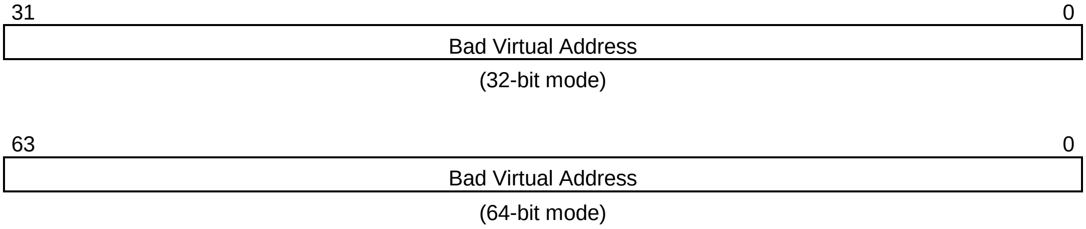


Figure 7-7 BadVAddr Register Formats

Table 7-7 BadVAddr Register Field Descriptions

32-bit mode

Bit	Field	Description	Cold Reset	Read/Write
31~0	BadVAddr	Bad Virtual address	Undefined	Read

64-bit mode

Bit	Field	Description	Cold Reset	Read/Write
63~0	BadVAddr	Bad Virtual address	Undefined	Read

7.2.8 Count Register (Reg#9)

The Count register is a read/write register. This register acts as a timer, incrementing at a constant rate (1/2 rate of CPUCLK) whether or not an instruction is executed, retired, or any forward progress is made through the pipeline.

This register can be also written for diagnostic purpose or system initialization. Figure 7-8 shows the format of the Count register and Table 7-8 describes the Count register field.

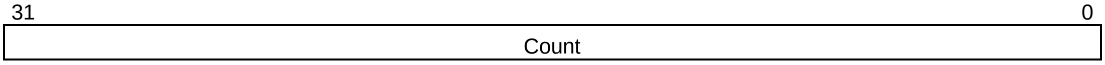


Figure 7-8 Count Register Format

Table 7-8 Count Register Field Description

Bit	Field	Description	Cold Reset	Read/Write
31~0	Count	32-bit timer, incrementing at half the maximum instruction issue rate (CPUCLK).	0x0	Read/Write

7.2.9 EntryHi Register (Reg#10)

The EntryHi is a read/write register, and holds the high-order bits of a TLB entry for TLB read and write operations. This register is accessed by the TLB Probe (TLBP), TLB Write Ransom (TLBWR), TLB Write Indexed (TLBWI), and TLB Read Indexed (TLBR) instructions.

When either a TLB refill, TLB invalid, or TLB modified exception occurs, this register is loaded with the virtual page number (VPN2) and the ASID of the virtual address that did not have a matching TLB entry. Figure 7-9 shows the formats of the EntryHi register and Table 7-9 describes the EntryHi register fields.

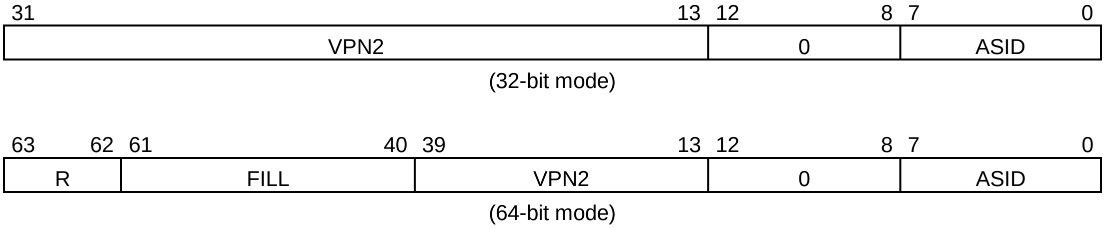


Figure 7-9 EntryHi Register Formats

Table 7-9 EntryHi Register Field Descriptions

32-bit mode

Bit	Field	Description	Cold Reset	Read/Write
31~1	VPN2	Virtual page number divided by two	Undefined	Read/Write
12~8	0	Reserved	0x0	Read
7~0	ASID	Address space ID field An 8-bit field that lets multiple processes share the TLB; each process has a distinct mapping of otherwise identical virtual page numbers.	Undefined	Read/Write

64-bit mode

Bit	Field	Description	Cold Reset	Read/Write
63~62	R	Region. Used to match vAddr63 and vAddr62. 00: user, 01: supervisor, 11: kernel	Undefined	Read/Write
61~40	Fill	Reserved. 0 on read. Ignored on write.	Undefined	Read
39~13	VPN2	Virtual page number divided by two	Undefined	Read/Write
12~8	0	Reserved	0x0	Read
7~0	ASID	Address space ID field.	Undefined	Read/Write

7.2.10 Compare Register (Reg#11)

The Compare register acts as a timer. When value of the Count register equals the value of the Compare register, interrupt bit IP (7) in the Cause register is set. This causes an interrupt exception as soon as the interrupt is enabled. Writing a value to this register, as a side effect, clears the timer interrupt.

For diagnostic purpose, this register is a read/write register. However, in normal operation this register is write only. Figure 7-10 shows the format of the Compare register and Table 7-10 describes the Compare register field.

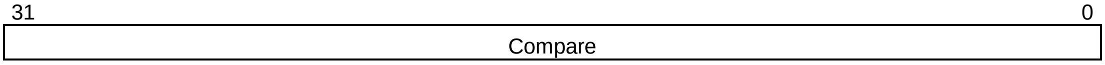


Figure 7-10 Compare Register Format

Table 7-10 Compare Register Field Description

Bit	Field	Description	Cold Reset	Read/Write
31~0	Compare	Acts as a timer; it maintains a stable value that does not change on its own.	0x0	Read/Write

7.2.11 Status Register (Reg#12)

The Status register is a read/write register that contains the operating mode, interrupt enabling, and diagnostic states of the processor. The more important Status register fields are as following;

- The Interrupt Mask (IM) field of 8 bits controls the enabling of eight interrupt conditions. Interrupt must be enabled before they can be asserted, and the corresponding bits are set in both the IM field of this register and the Interrupt Pending field of the Cause register.
- The Coprocessor Usability (CU) field of 4 bits controls the usability of four possible coprocessors. Regardless of the CU0 bit setting, CP0 is always usable in Kernel mode.
- The Diagnostic Status (DS) field of 9 bits is used for self-testing, and checks the cache and virtual memory system.
- The Reverse Endian (RE) bit reverses the endianness. The processor can be configured as either little/big-endian at reset; reverse-endian selection is used in Kernel and Supervisor modes, and in the User mode when the RE bit is 0. Setting the RE bit to 1 inverts the User mode endianness.

Figure 7-11 shows the format of the Status register and Table 7-11 describes the Status register field.

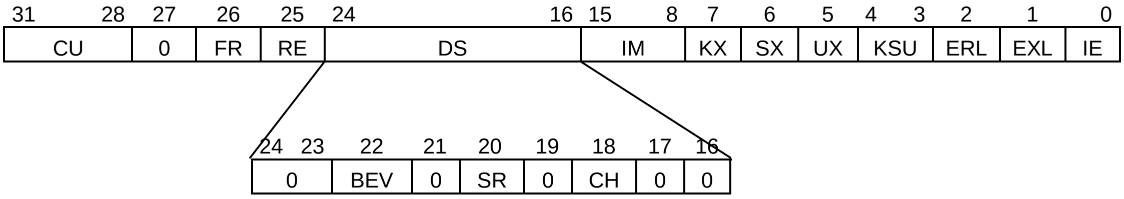


Figure 7-11 Status Register Format

Table 7-11 Status Register Field Descriptions

Bit	Field	Description	Cold Reset	Read/Write
31~28	CU (3,2,1,0)	Controls the usability of each of the four coprocessor unit numbers. CP0 is always usable when in Kernel mode, regardless of the setting of the CU0 bit. 0: unusable, 1: usable	0000	Read/Write
27	0	Reserved	0	Read
26	FR	Enables additional floating-point registers. 0: 16 registers, 1: 32 registers	0	Read/Write
25	RE	Reverse-Endian bit, valid in User mode.	0	Read/Write
24~23	0	Reserved	0x0	Read
22	BEV	Controls the location of TLB refill and general exception vectors. 0: normal, 1: bootstrap	1	Read/Write

Bit	Field	Description	Cold Reset	Read/Write
21	0	Reserved	0	Read
20	SR	1: Indicates a soft reset or NMI has occurred.	0	Read/Write
19	0	Reserved	0	Read
18	CH	“Hit” or “miss” indication for last CACHE Hit Invalidate, Hit Write Back Invalidate, Hit Write Back for a primary cache. 0: miss, 1: hit.	0	Read/Write
17~16	0	Reserved	0x0	Read
15~8	IM	Interrupt Mask Controls the enabling of each of the external, internal and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the IM field of the Status register and the IP field of the Cause register. 0: disabled, 0: enabled	0x0	Read/Write
7	KX	Enables 64-bit addressing in Kernel mode. The extended-addressing TLB refill exception is used for TLB misses on kernel addresses. 0: 32-bit, 1: 64-bit	0	Read/Write
6	SX	Enables 64-bit addressing and operations in Supervisor mode. The extended-addressing TLB refill exception is used for TLB misses on supervisor addresses. 0: 32-bit, 1: 64-bit	0	Read/Write
5	UX	Enables 64-bit addressing and operations in User mode. The extended-addressing TLB refill exception is used for TLB misses on user addresses. 0: 32-bit, 1: 64-bit	0	Read/Write
4~3	KSU	Mode. 10: user, 01: supervisor, 00: kernel.	0x0	Read/Write
2	ERL	Error Level. 0: normal, 1: error.	1	Read/Write
1	EXL	Exception Level. 0: normal, 1: exception.	0	Read/Write
0	IE	Interrupt Enable. 0: disable, 1: enable.	0	Read/Write

Status Register Modes and Access States

Fields of the Status register set the modes and access states described in the section that follow.

- Interrupt Enable: Interrupts are enabled when all of the following conditions are met:
  - IE = 1
  - EXL = 0
  - ERL = 0

  If these conditions are met, the settings of the IM bits enable the interrupt.

- Operation Modes: The following CPU Status register bit settings are required for User, Kernel and Supervisor modes (see Section 8.3, *Operation Modes*, for more information about operating modes).
  - The processor is in User mode when KSU = 10_2, EXL = 0, and ERL = 0.
  - The processor is in Supervisor mode when KSU = 01_2, EXL = 0 and ERL = 0.
  - The processor is in Kernel mode when KSU = 00_2, or EXL= 1, or ERL =1.

- 32- and 64-bit Modes: The following CPU Status register settings select 32- or 64-bit operation for User, Kernel, and Supervisor operating modes. Enabling 64-bit operation permits the execution of 64-bit opcodes and translation of 64-bit addresses. 64-bit operation for User, Kernel and Supervisor modes can be set independently.
  - 64-bit addressing for Kernel mode is enabled when KX = 1. 64-bit operations are always valid in Kernel mode.
  - 64-bit addressing and operations are enabled for Supervisor mode when SX = 1.
  - 64-bit addressing and operations are enabled for User mode when UX = 1.

- Kernel Address Space Accesses: Access to the kernel address space is allowed when the processor is in Kernel mode.

- Supervisor Address Space Accesses: Access to the supervisor address space is allowed when the processor is in Kernel or Supervisor mode, as described above in the section above titled *Operating Modes*.

- User Address Space Accesses: Access to the user address is allowed in any of the three operating modes.

Status Register Reset

The contents of the Status register are undefined at reset, except for the following bits in the Diagnostic Status field:

- ERL and BEV = 1

The SR bit distinguishes between the Reset exception and the Soft Reset exception (caused by Nonmaskable Interrupt [NMI]).

7.2.12 Cause Register (Reg#13)

The Cause register holds the cause of the most recent exception. This register is read-only, except for the IP[1~0] bits. Figure 7-12 shows the format of the Cause register and Table 7-12 describes the Cause register field.

31	30	29 28	27 16	15 8	7	6 2	1 0
BD	0	CE	0	IP	0	ExcCode	0

Figure 7-12 Cause Register Format

Table 7-12 Cause Register Field Descriptions

Bit	Field	Description	Cold Reset	Read/Write
31	BD	Indicates whether or not the last exception was taken while executing in a branch delay slot. 0: normal, 1: delay slot.	0	Read
30	0	Reserved	0	Read
29~28	CE	Indicates the coprocessor unit number referenced when a coprocessor unusable exception is taken. 00: coprocessor 0, 01: coprocessor 1, 10: coprocessor 2, 11: coprocessor 3.	0x0	Read
27~16	0	Reserved	0x0	Read
15~10	IP [7~2]	Indicates whether an interrupt is pending. 0: not pending, 1: pending.	INT[5:0]	Read
9~8	IP [1~0]	Software interrupts. 0: reset, 1: set.	0x0	Read/Write
7	0	Reserved	0	Read
6~2	ExcCode	Exception Code field. 0: Int: Interrupt. 1: Mod: TLB modification exception. 2: TLBL: TLB exception (load or instruction fetch) 3: TLBS: TLB exception (Store) 4: AdEL: Address error exception (load or instruction fetch) 5: AdES: Address error exception (store) 6: IBE: Bus error exception (instruction fetch) 7: DBE: Bus error exception (data reference: load or Store) 8: Sys: Syscall exception 9: Bp: Breakpoint exception 10: RI: Reserved instruction exception 11: CpU: Coprocessor Unusable exception 12: Ov: Arithmetic Overflow exception 13: Tr: Trap exception 14: Reserved: 15: FPE: Floating-Point exception 16-31: Reserved :	0x0	Read
1~0	0	Reserved	0x0	Read

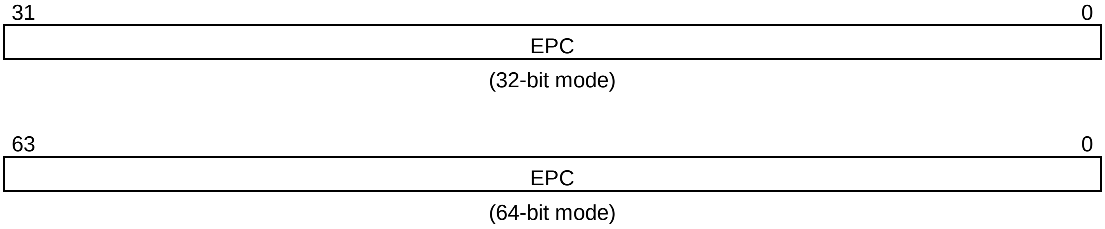
7.2.13 EPC Register (Reg#14)

The Exception Program Counter (EPC) register is a read/write register. This register contents the address at which processing resumes after an exception has been serviced.

For synchronous exceptions, this register contains either;

- the virtual address of the instruction that was the direct cause of the exception.
- the virtual address of the immediately preceding branch or jump instruction (when the instruction is in a branch delay slot, and the Branch Delay bit in the Cause register is set).

The processor does not write to the EPC register when EXL bit in the Status register is set to 1. Figure 7-13 shows the formats of the EPC register and Table 7-13 describes the EPC register field.



(32-bit mode)

(64-bit mode)

Figure 7-13 EPC Register Formats

Table 7-13 EPC Register Field Description

32-bit mode

Bit	Field	Description	Cold Reset	Read/Write
31~0	EPC	Exception program counter	Undefined	Read/Write

64-bit mode

Bit	Field	Description	Cold Reset	Read/Write
63~0	EPC	Exception program counter	Undefined	Read/Write

7.2.14 PRId Register (Reg#15)

The Processor Revision Identifier (PRId) register is a read-only register. This register contents information identifying the implementation and revision level of the CPU and CP0. Figure 7-14 shows the format of the PRId register and Table 7-14 describes the PRId register field.

31 16	15 8	7 0
0	Imp	Rev

Figure 7-14 PRId Register Format

Table 7-14 PRId Register Field Descriptions

Bit	Field	Description	Cold Reset	Read/Write
31~16	0	Reserved	0x0	Read
15~8	Imp	Implementation number 0x2d means “TX49 family”.	0x2d	Read
7~0	Rev	Revision number +.	+	Read

+ Value is shown in product sheet

7.2.15 Config Register (Reg#16)

The Config register is a read-only register; except for HALT, ICE#, DCE# and K0 fields. This register specifies various configuration options selected on the TX49.

EC, BE, IC, DC, IB and DB fields are set by the hardware during reset and are included in this register as read-only status bits for the software to access. Figure 7-15 shows the format of the Config register and Table 7-15 describes the Config register field.

31	30 28	27 24	23 19	18	17	16	15	14 13	12	11 9	8 6	5	4	3	2 0
0	EC	0	0	HALT	ICE#	DCE#	BE	1	0	IC	DC	IB	DB	0	K0

Figure 7-15 Config Register Format

Table 7-15 Config Register Field Descriptions

Bit	Field	Description	Cold Reset	Read/Write
31	0	Reserved	0	Read
30~28	EC	GBUS clock rate: 0: processor clock frequency divided by 2 1: processor clock frequency divided by 3 2: processor clock frequency divided by 4 7: processor clock frequency divided by 2.5 3, 4, 5, 6 : reserved	pin	Read
27	0	Reserved	pin	Read/Write
26~24	0	Reserved	pin	Read
23~19	0	Reserved	0	Read
18	HALT	Wait mode. 0: Halt 1: Doze Indicates the power-down behavior of the TX49 when WAIT instruction is executed. The TX49 stalls the pipeline both in halt and doze mode. Cache snoops are possible during Doze mode but not possible during Halt mode. Halt mode reduces power consumption to a greater extent than Doze mode.	0	Read/Write
17	ICE#	Instruction Cache Enable 0: Instruction cache enable 1: Instruction cache disable	0	Read/Write
16	DCE#	Data Cache Enable 0: Data cache enable 1: Data cache disable	0	Read/Write
15	BE	Big Endian 0: Little Endian 1: Big Endian	pin	Read
14~13	1	Reserved	11	Read
12	0	Reserved	0	Read

Bit	Field	Description	Cold Reset	Read/Write
11~9	IC	Instruction cache size. In the TX49, this is set to 8 KB (001), 16 KB (010) or 32 KB (011).	001, 010 or 011	Read
8~6	DC	Data cache size. In the TX49, this is set to 8 KB (001), 16 KB (010) or 32 KB (011).	001, 010 or 011	Read
5	IB	Primary I-Cache line Size 1:32 bytes (8 words)	1	Read
4	DB	Primary D-cache line Size 1:32 bytes (8 words)	1	Read
3	0	Reserved	0	Read
2~0	K0	kseg0 coherency algorithm 0: Cacheable, noncoherent, write-through, no-WA 1: Cacheable, noncoherent, write-through, WA 2: Uncached 3: Cacheable, noncoherent, write-back, WA 4-7: Reserved	0x0	Read/Write

7.2.16 LLAddr Register (Reg#17)

The Load Linked Address (LLAddr) register is a read/wirte register, and contains the physical address read by the most recent Load Linked (LL/LLD) instruction. This register is for diagnostic purposes only, and serves no function during normal operation. Figure 7-16 shows the format of the LLAddr register and Table 7-16 describes the LLAddr register field.

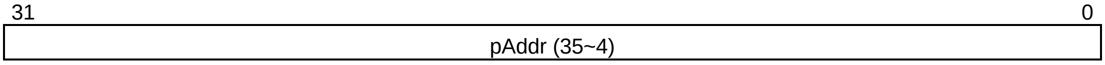


Figure 7-16 LLAddr Register Format

Table 7-16 LLAddr Register Field Description

Bit	Field	Description	Cold Reset	Read/Write
31~0	pAddr	Physical address bits 35~4	0x0	Read/Write

7.2.17 XContext Register (Reg#20)

The XContext register is a read/write register, and contains a pointer to an entry in the page table entry (PTE) array, an operating system data structure that stores virtual to physical address translations. When there is a TLB miss, the operating system software loads the TLB with the missing translation from the PTE array. However the contents of this register duplicates some information of the BadVAddr register, it is arranged in a form that is more useful for TLB exception handler by a software. This register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for operating system use. The operating system sets the PTE base field in the register, as needed. Normally, the operating system uses this register to address the current page map which resides in the Kernel mapped segment, kseg3.

The BadVPN2 field of 27 bits has bit [39~13] of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4 KByte page size, this format may be used directly to access the pair-table of 8 Byte PTEs. For other page sizes and PTE sizes, shifting and masking this value produces the appropriate address.

Figure 7-17 shows the format of the XContext register and Table 7-17 describes the XContext register field.

63 33	32 31	30 4	3 0
PTEBase	R	BadVPN2	0

Figure 7-17 XContext Register Format

Table 7-17 XContext Register Field Description

Bit	Field	Description	Cold Reset	Read/Write
63~33	PTEBase	Page table entry base pointer This field is normally written with a value that allows the operation system to use the Context register as a pointer into the current PTE array in memory.	Undefined	Read/Write
32~31	R	The Region field contains bits 63 to 62 of the virtual address. 00: user, 01: supervisor, 11: kernel	Undefined	Read/Write
30~4	BadVPN2	Bad virtual page number divided by two. This field is written by hardware on a miss. It contains the VPN of the most recent invalidly translated virtual address.	Undefined	Read
3~0	0	Reserved	0x0	Read

7.2.18 Debug Register (Reg#23)

The Debug register is a read-only; except for TLF, BsF, SSt and JtagRst fields. This register holds the information for debug handler. Figure 7-18 shows the format of the Debug register and Table 7-18 describes the Debug register field.

31	30	29 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DBD	DM	0	NIS	TRS	OES	TLF	BsF	0	SSt	JtagRst	0	DINT	DIB	DDBS	DDBL	DBp	DSS

Figure 7-18 Debug Register Format

Table 7-18 Debug Register Field Descriptions

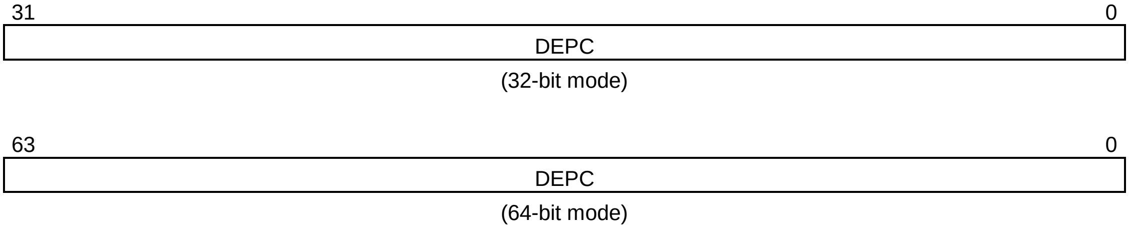
Bit	Field	Description	Cold Reset	Read/Write
31	DBD	Debug Branch Delay; When a debug exception occurs while an instruction in the branch delay slot is executing, this bit is set to 1.	0	Read
30	DM	Debug Mode; It indicates that a debug exception has taken place. This bit is set when a debug exception is taken, and is cleared upon return from the exception (DERET). While this bit is set all interrupts, including NMI, TLB exception , BUS error exception, and debug exception are masked and cache line locking function is disabled. 0: Debug handler not running. 1: Debug handler running.	0	Read
29~15	0	Reserved	0x0	Read
14	NIS	Non-maskable Interrupt Status; When this bit is set indicating that a non-maskable interrupt has occurred at the same time as a debug exception. In this case the Status, Cause, EPC, and BadVAddr registers assumes the usual status after occurrence of a non-maskable interrupt, but the address in DEPC is not the non-maskable exception vector address (0xbfc0 0000). Instead, 0xbfc0 0000 is put in DEPC by the debug handler software after which processing returns directly from the debug exception to the non-maskable interrupt handler.	0	Read
13	TRS	TLB Miss Status; When this bit is set indicating the Debug Exception and TLB/XTLB refill exception has occurred at the same time. In this case the Status, Cause, EPC, and BadVAddr registers assumes the usual status after occurrence of TLB/XTLB refill. The address in the DEPC is not the other exception vector address. Instead, 0xbfc0 0200 (if BEV = 1) in case of TLB refill exception and 0xbfc0 0280 (if BEV = 1) in case of XTLB refill exception or 0x8000 0000 (if BEV = 0) in case of TLB refill exception and 0x8000 0080 (if BEV = 0) in case of XTLB refill exception is put in DEPC by the debug exception handler software, after which processing returns directly from the debug exception to the other exception handler.	0	Read

Bit	Field	Description	Cold Reset	Read/Write
12	OES	Other Exception Status; When this bit is set indicates exception other than reset, NMI, or TLB/XTLB refill has occurred at the same time as a debug exception. In this case the Status, Cause, EPC, and BadVAddr registers assume the usual status after occurrence of such an exception, but the addressing the DEPC is not the other exception Vector address. Instead, 0xbfc0 0380 (if BEV = 1) or 0x8000 0180 (if BEV = 0) is put in DEPC by the debug exception handler software, after which processing returns directly from the other exception handler.	0	Read
11	TLF	TLB Exception Flag; This bit is set to 1 when TLB related exception occurs for immediately preceding load or store instruction while a debug exception handler is running (DM = 1). TLB exception will set this bit to 1 regardless of writing zero. It is cleared by writing 0 and writing 1 is ignored.	0	Read/Write
10	BsF	Bus Error Exception Flag; This bit is set to 1 when a bus error exception occurs for a load or store instruction while a debug exception handler is running (DM = 1). Bus error exception will set this bit to 1 regardless of writing zero. It is cleared by writing 0 and writing 1 is ignored.	0	Read/Write
9	0	Reserved	0	Read
8	SSt	Single Step; Set to 1 indicates the single step debug function is enable (1) or disabled (0). The function is disable when the DM bit is set to 1 while the debug exception is running.	0	Read/Write
7	JtagRst	JTAG Reset; When this bit is set to 1 the processor reset the JTAG unit.	0	Read/Write
6	0	Reserved	0	Read
5	DINT	Debug Interrupt Break Exception Status; set to 1 when debug interrupts occurs.	0	Read
4	DIB	Debug Instruction Break Exception Status; Set to 1 on instruction address break.	0	Read
3	DDBS	Debug Data Break Store Exception Status; Set to 1 on data address break at store operation.	0	Read
2	DDBL	Debug Data Break Load Exception Status; Set to 1 on data address break at load operation.	0	Read
1	DBp	Debug Breakpoint Exception Status; This bit is set when executing SDBBP instruction.	0	Read
0	DSS	Debug Single Step Exception Status; Set to 1 indicate Single Step Exception.	0	Read

7.2.19 DEPC Register (Reg#24)

The DEPC register holds the address where processing resumes after the debug exception routine has finished. The address that has been loaded in the DEPC register is the virtual address of the instruction that caused the debug exception. If the instruction is in the branch delay slot, the virtual address of the immediately preceding branch or jump instruction is placed in this register. Execution of the DERET instruction causes a jump to the address in the DEPC. If the DEPC is both written from software (by MTC0) and by hardware (debug exception) then the DEPC is loaded by the value generated by the hardware.

Figure 7-19 shows the formats of the DEPC register and Table 7-19 describes the DEPC register field.



(32-bit mode)

(64-bit mode)

Figure 7-19 DEPC Register Formats

Table 7-19 DEPC Register Field Description

32-bit mode

Bit	Field	Description	Cold Reset	Read/Write
31~0	DEPC	Debug exception program counter.	Undefined	Read/Write

64-bit mode

Bit	Field	Description	Cold Reset	Read/Write
63~0	DEPC	Debug exception program counter.	Undefined	Read/Write

7.2.20 TagLo Register (Reg#28) and TagHi Register (Reg#29)

The TagLo and TagHi registers are a read/write registers. These registers hold the primary cache tag for cache lock function or cache diagnostics. These registers are written by the CACHE/MTC0 instruction. Figure 7-20 shows the formats of the TagLo and TagHi registers and Table 7-20 describes the TagLo and TagHi registers field.

31 8	7 6	5 3	2	1	0
PTagLo	PState	RWNT	Lock	F0	0

(TagLo)

31	30	29 0
F1	PtagLo1	0

(TagHi)

Figure 7-20 TagLo and TagHi Register Formats

Table 7-20 TagLo and TagHi Register Field Descriptions

TagLo

Bit	Field	Description	Cold Reset	Read/Write
31~8	PTagLo	Bits 35~12 of the physical address	0x0	Read/Write
7~6	PState	Specifies the primary cache state 0: Invalid 1: Reserved 2: Reserved 3: Valid	0x0	Read/Write
5~3	RWNT	Read/Write bits required for Windows NT	0x0	Read/Write
2	Lock	Lock bit (0: not locked, 1: locked)	0	Read/Write
1	F0	FIFO Replace bit 0 (indicates the set to be replaced)	0	Read/Write
0	0	Reserved	0	Read

TagHi

Bit	Field	Description	Cold Reset	Read/Write
31	F1	FIFO Replace bit 1 (indicates the set to be replaced)	0	Read/Write
30	PTagLo1	Bit 11 of the physical address	0	Read/Write
29~0	0	Reserved	0x0	Read

F1 and F0 are concatenated and indicate the set to be replaced.

F1 || F0

0 0 : way0

0 1 : way1

1 0 : way2

1 1 : way3

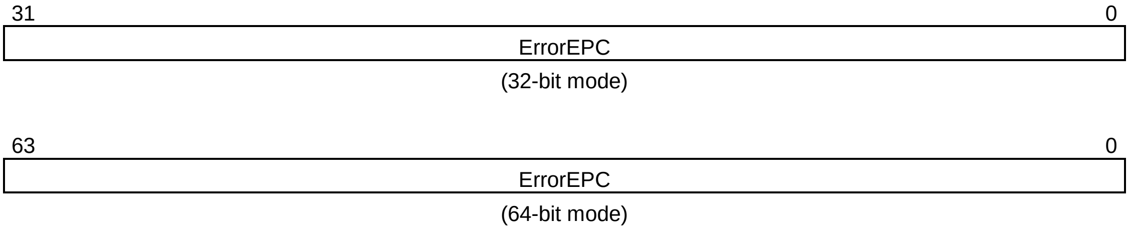
7.2.21 ErrorEPC Register (Reg#30)

The ErrorEPC is a read/write register, and is similar to the EPC register. This register is used to store the program counter (PC) on ColdReset, SoftReset and NMI exceptions.

This register contains the virtual address at which instruction processing can resume after servicing an error. This address can be;

- The virtual address of the instruction that caused the exception
- The virtual address of the immediately preceding branch or jump instruction, when this address is in a branch delay slot.

There is no branch delay slot indication for this register. Figure 7-21 shows the formats of the ErrorEPC register and Table 7-21 describes the ErrorEPC register field.



(32-bit mode)

(64-bit mode)

Figure 7-21 ErrorEPC Register Formats

Table 7-21 ErrorEPC Register Field Descriptions

32-bit mode

Bit	Field	Description	Cold Reset	Read/Write
31~0	ErrorEPC	Error Exception Program Counter.	Undefined	Read/Write

64-bit mode

Bit	Field	Description	Cold Reset	Read/Write
63~0	ErrorEPC	Error Exception Program Counter.	Undefined	Read/Write

7.2.22 DESAVE Register (Reg#31)

This register is used by the debug exception handler to save one of the GPRs, that is then used to save the rest of the context to a pre-determined memory are, e.g. in the processor probe. This register allows the safe debugging of exception handlers and other types of code where the existence of a valid stack for context saving cannot be assumed.

Figure 7-22 shows the formats of the DESAVE register and Table 7-22 describes the DESAVE register field.

Note: This register can use for ICE system only.

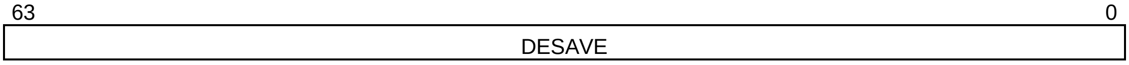


Figure 7-22 DESAVE Register Format

Table 7-22 DESAVE register Field Description

32/64-bit mode

Bit	Field	Description	Cold Reset	Read/Write
63~0	DESAVE	Save one of the GPRs	Undefined	Read/Write

7.2.23 The Initialization of CP0 Registers in SoftReset Exception

Table 7-23 shows the values of the registers that be initialized by SoftReset exception.

Table 7-23 The Initial Value by SoftReset Exception

Register	Bit	Field	SoftRest	Description
Status (Reg#12)	22	BEV	1	Same value as ColdReset
	20	SR	1	ColdReset has priority over SoftReset
	2	ERL	1	Same value as ColdReset

8. Memory Management System

8.1 Introduction

The TX49 provides a full-featured memory management unit (MMU) which uses an on-chip translation look aside buffer (TLB) to translate virtual addresses into physical addresses.

8.2 Address Space Overview

The TX49 physical address space is 64 Gbyte using a 36-bit address. The virtual address is either 64 or 32 bits wide depending on whether the processor is operating in 64- or 32-bit mode. In 32-bit mode, addresses are 32-bits wide and the maximum user process size is 2 Gbyte (2**31). In 64-bit mode, addresses are 64-bit wide and the maximum user process is 1 Tbyte (2**40). The virtual address is extended with an Address Space Identifier (ASID) to reduce the frequency of TLB flushing when switching context. The size of the ASID field is 8 bits. The ASID is contained in the CP0 EntryHi register.

8.2.1 Virtual Address Space

The processor virtual address can be either 32 or 64 bits wide, depending on whether the processor is operating in 32-bit or 64-bit mode.

- In 32-bit mode, addresses are 32 bits wide.

  The maximum user process size is 2 gigabytes (2^{31}).
- In 64-bit mode, addresses are 64 bits wide.

  The maximum user process size is 1 terabyte (2^{40}).

Figure 8-1 shows the translation of a virtual address into a physical address.

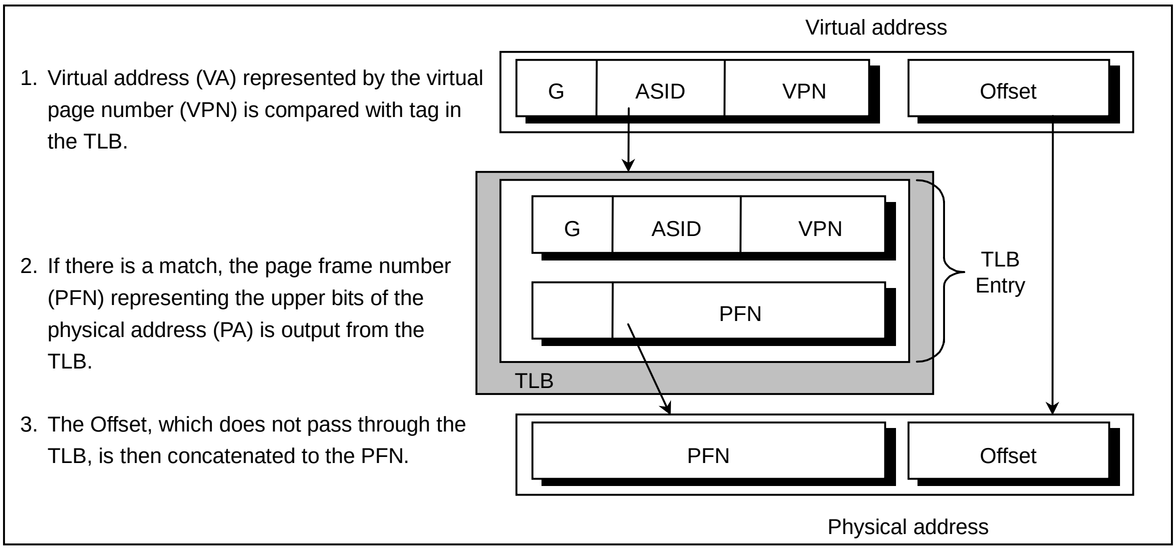

Virtual address
1. Virtual address (VA) represented by the virtual page number (VPN) is compared with tag in the TLB.
G
ASID
VPN
Offset
G
ASID
VPN
TLB Entry
2. If there is a match, the page frame number (PFN) representing the upper bits of the physical address (PA) is output from the TLB.
PFN
TLB
3. The Offset, which does not pass through the TLB, is then concatenated to the PFN.
PFN
Offset
Physical address

Figure 8-1 Overview of a Virtual-to-Physical Address Translation

As shown in Figure 8-2 and Figure 8-3, the virtual address is extended with an 8-bit address space identifier (ASID), which reduces the frequency of TLB flushing when switching contexts. This 8-bit ASID is in the CP0 *EntryHi* register, described later in this chapter. The *Global* bit (G) is in the *EntryLo0* and *EntryLo1* registers, described later in this chapter.

8.2.2 Physical Address Space

Using a 36-bit address, the processor physical address space encompasses 64 Gbytes. The section following describes the translation of a virtual address to a physical address.

8.2.3 Virtual-to-Physical Address Translation

Converting a virtual address to a physical address begins by comparing the virtual address from the processor with the virtual addresses in the TLB; there is a match when the virtual page number (VPN) of the address is the same as the VPN field of the entry, and either:

- the Global (G) bit of the TLB entry is set, or
- the ASID field of the virtual address is the same as the ASID field of the TLB entry.

This match is referred to as a *TLB hit*. If there is no match, a TLB Miss exception is taken by the processor and software is allowed to refill the TLB from a page table of virtual/physical addresses in memory.

If there is a virtual address match in the TLB, the physical address is output from the TLB and concatenated with the *Offset*, which represents an address within the page frame space. The *Offset* does not pass through the TLB.

Virtual-to-physical translation is described in greater detail throughout the remainder of this chapter; Figure 8-8 is a flow diagram of the process shown at the end of this chapter. The next two sections describe the 32-bit and 64-bit address translations.

8.2.4 32-bit Mode Address Translation

Figure 8-2 shows the virtual-to-physical-address translation of a 32-bit mode address. This figure illustrates two of the possible page sizes: a 4-Kbyte page (12 bits) and a 16-Mbyte page (24 bits).

- The top portion of Figure 8-2 shows a virtual address with a 12-bit, or 4-Kbyte, page size, labeled *Offset*. The remaining 20 bits of the address represent the VPN, and Index the 1M-entry page table.
- The bottom portion of Figure 8-2 shows a virtual address with a 24-bit, or 16-Mbyte, page size, labeled *Offset*. The remaining 8 bits of the address represent the VPN, and index the 256-entry page table.

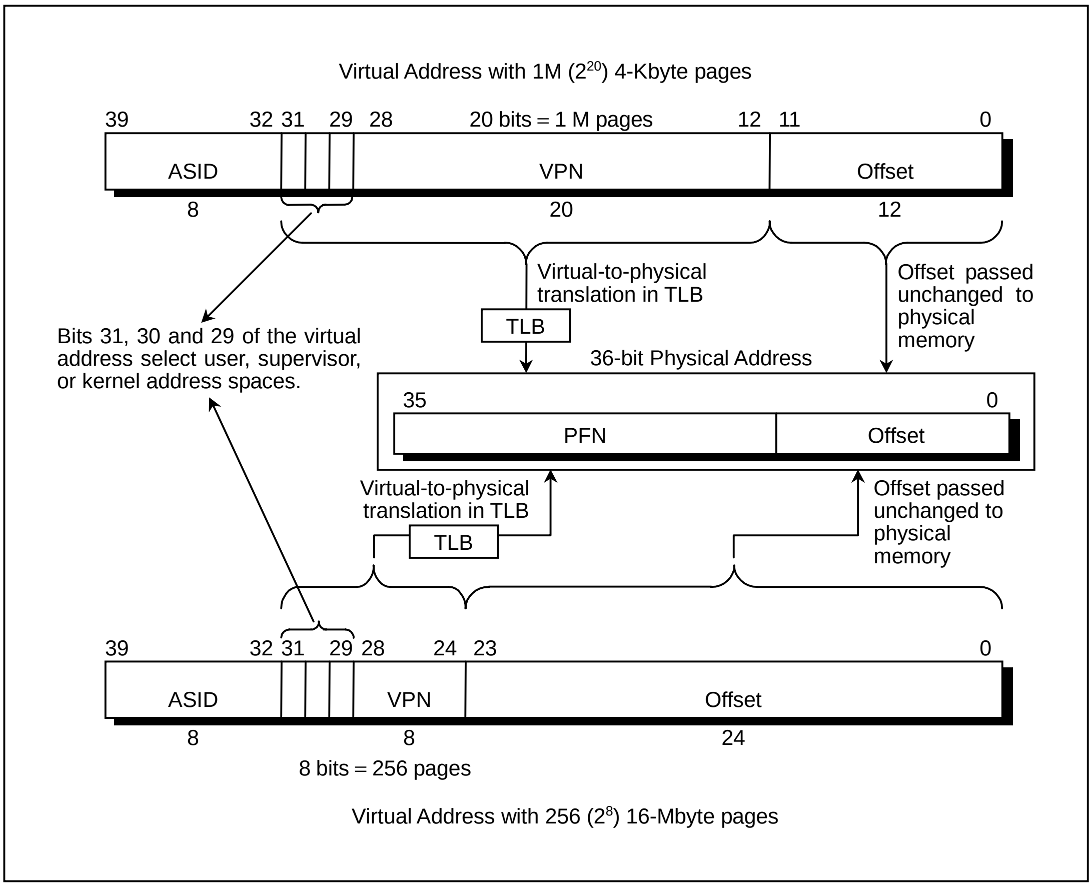

Virtual Address with 1M (2^{20}) 4-Kbyte pages
39
32
31
29
28
20 bits = 1 M pages
12
11
0
ASID
VPN
Offset
8
20
12
Virtual-to-physical translation in TLB
Offset passed unchanged to physical memory
Bits 31, 30 and 29 of the virtual address select user, supervisor, or kernel address spaces.
TLB
36-bit Physical Address
35
0
PFN
Offset
Virtual-to-physical translation in TLB
TLB
Offset passed unchanged to physical memory
39
32
31
29
28
24
23
0
ASID
VPN
Offset
8
8
24
8 bits = 256 pages
Virtual Address with 256 (2^8) 16-Mbyte pages

Figure 8-2 32-bit Mode Virtual Address Translation

8.2.5 64-bit Mode Address Translation

Figure 8-3 shows the virtual-to-physical-address translation of a 64-bit mode address. This figure illustrates two of the possible page sizes: a 4-Kbyte page (12 bits) and a 16-Mbyte page (24 bits).

- The top portion of Figure 8-3 shows a virtual address with a 12-bit, or 4-Kbyte, page size, labelled *Offset*. The remaining 28 bits of the address represent the VPN, and index the 256M-entry page table.
- The bottom portion of Figure 8-3 shows a virtual address with a 24-bit, or 16-Mbyte, page size, labelled *Offset*. The remaining 16 bits of the address represent the VPN, and index the 64K-entry page table.

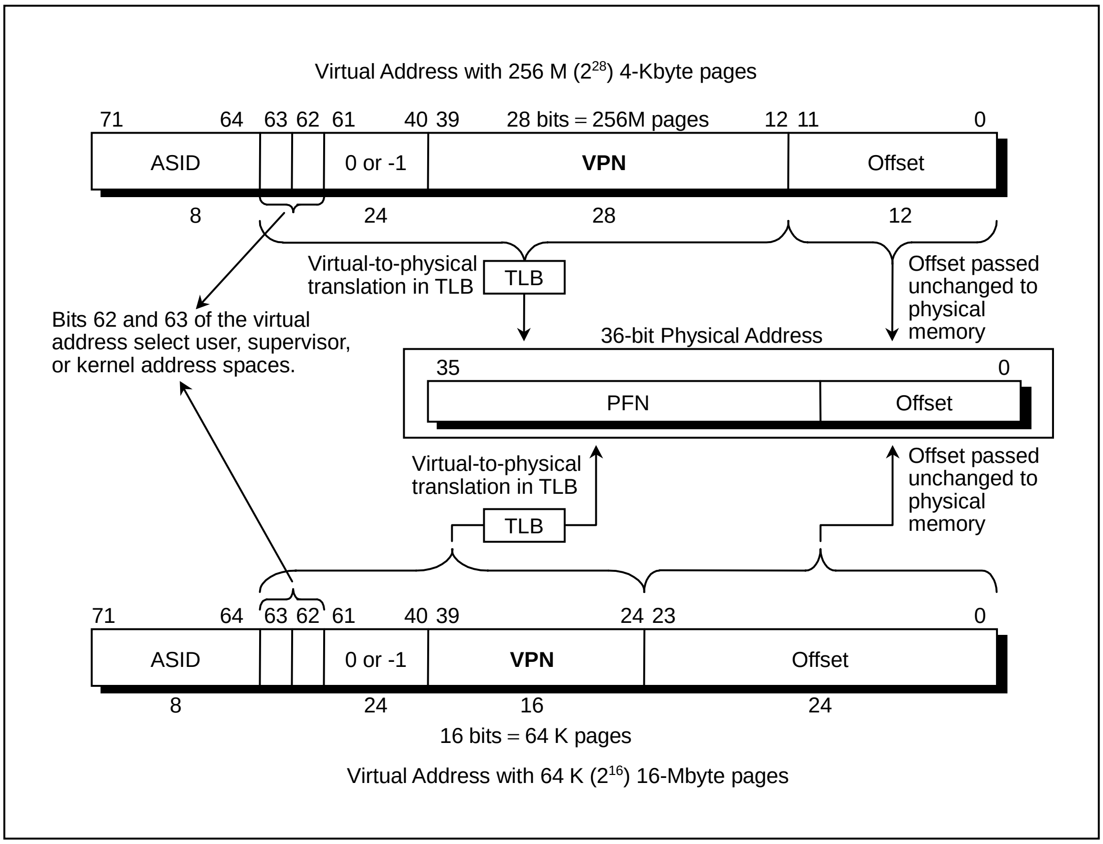

Virtual Address with 256 M (2^{28}) 4-Kbyte pages
71
64
63
62
61
40
39
28 bits = 256M pages
12
11
0
ASID
0 or -1
VPN
Offset
8
24
28
12
Virtual-to-physical translation in TLB
TLB
Offset passed unchanged to physical memory
Bits 62 and 63 of the virtual address select user, supervisor, or kernel address spaces.
36-bit Physical Address
35
0
PFN
Offset
Virtual-to-physical translation in TLB
TLB
Offset passed unchanged to physical memory
71
64
63
62
61
40
39
24
23
0
ASID
0 or -1
VPN
Offset
8
24
16
24
16 bits = 64 K pages
Virtual Address with 64 K (2^{16}) 16-Mbyte pages

Figure 8-3 64-bit Mode Virtual Address Translation

8.3 Operating Modes

The TX49 has the three operating modes, User mode, Supervisor mode and Kernel mode, for 32- and 64-bit operation. The KSU, EXL and ERL bit in the Status register select User, Supervisor or Kernel mode. The UX, SX and KX bit in the Status register select 32- or 64-bit addressing in user, supervisor and kernel mode respectively.

KSU	EXL	ERL	UX	SX	KX	Mode
10	0	0	0	-	-	32-bit addressing in user mode
10	0	0	1	-	-	64-bit addressing in user mode
01	0	0	-	0	-	32-bit addressing in supervisor mode
01	0	0	-	1	-	64-bit addressing in supervisor mode
00	-	-	-	-	0	32-bit addressing in kernel mode
-	1	-	-	-	0	32-bit addressing in kernel mode
-	-	1	-	-	0	32-bit addressing in kernel mode
00	-	-	-	-	1	64-bit addressing in kernel mode
-	1	-	-	-	1	64-bit addressing in kernel mode
-	-	1	-	-	1	64-bit addressing in kernel mode

8.3.1 User Mode Operations

In User mode, a single, uniform virtual address space-labelled User segment-is available; its size is:

- 2 Gbytes (2^{31} bytes) in 32-bit mode (*useg*)
- 1 Tbyte (2^{40} bytes) in 64-bit mode (*xuseg*)

Figure 8-4 shows User mode virtual address space.

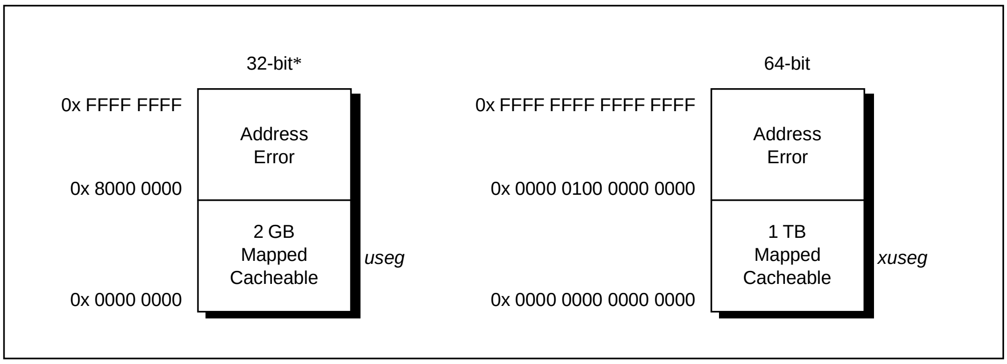


Figure 8-4 User Mode Virtual Address Space

*Note: In 32-bit mode, bit 31 is sign-extended through bits 63~32. Failure results in an address error exception.

The User segment starts at address 0 and the current active user process resides in either *useg* (in 32-bit mode) or *xuseg* (in 64-bit mode). The TLB identically maps all references to *useg*/*xuseg* from all modes, and controls cache accessibility.

The processor operates in User mode when the Status register contains the following bit-values:

- *KSU* bits = 10_2
- *EXL* = 0
- *ERL* = 0

In conjunction with these bits, the *UX* bit in the Status register selects between 32- or 64-bit User mode addressing as follows:

- when $UX = 0$, 32-bit useg space is selected and TLB misses are handled by the 32-bit TLB refill exception handler
- when $UX = 1$, 64-bit xuseg space is selected and TLB misses are handled by the 64-bit TLB refill exception handler

Table 8-1 lists the characteristics of the two user mode segments, useg and xuseg.

Table 8-1 32-bit and 64-bit User Mode Segments

Address Bit Values	Status Register Bit Values				Segment Name	Address Range	Segment Size
	KSU	EXL	ERL	UX			
32-bit A (31) = 0	10_2	0	0	0	*useg*	0x0000 0000 through 0x7FFF FFFF	2 Gbyte (2^{31} bytes)
64-bit A (63~40) = 0	10_2	0	0	1	*xuseg*	0x0000 0000 0000 0000 through 0x0000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)

32-bit User Mode (*useg*)

In User mode, when $UX = 0$ in the Status register, User mode addressing is compatible with the 32-bit addressing model shown in Figure 8-4, and a 2-Gbyte user address space is available, labelled useg.

All valid User mode virtual addresses have their most-significant bit cleared to 0; any attempt to reference an address with the most-significant bit set while in User mode causes an Address Error exception.

The system maps all references to *useg* through the TLB, and bit settings within the TLB entry for the page determine the cacheability of a reference.

64-bit User Mode (*xuseg*)

In User mode, when $UX = 1$ in the Status register, User mode addressing is extended to the 64-bit model shown in Figure 8-4 . In 64-bit User mode, the processor provides a single, uniform address space of 2^{40} bytes, labelled xuseg.

All valid User mode virtual addresses have bits 63~40 equal to 0; an attempt to reference an address with bits 63~40 not equal to 0 causes an Address Error exception.

The system maps all reference to *xuseg* through the TLB, and bit settings within the TLB entry for the page determine the cacheability of a reference.

8.3.2 Supervisor Mode Operations

Supervisor mode is designed for layered operating systems in which a true kernel runs in TX49 Kernel mode, and the rest of the operating system runs in Supervisor mode.

The processor operates in Supervisor mode when the Status register contains the following bit-values:

- $KSU = 01_2$
- $EXL = 0$
- $ERL = 0$

In conjunction with these bits, the SX bit in the Status register selects between 32- or 64-bit Supervisor mode addressing:

- when $SX = 0$, 32-bit supervisor space is selected and TLB misses are handled by the 32-bit TLB refill exception handler
- when $SX = 1$, 64-bit supervisor space is selected and TLB misses are handled by the 64-bit XTLB refill exception handler

The system maps all references through the TLB, and bit settings within the TLB entry for the page determine the cacheability of a reference.

Figure 8-5 shows Supervisor mode address mapping. Table 8-2 lists the characteristics of the supervisor mode segments; descriptions of the address spaces follow.

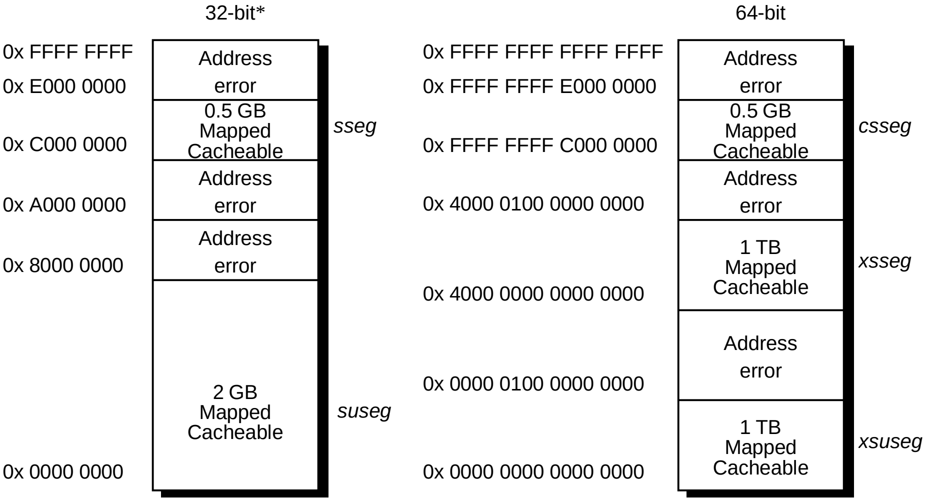


Figure 8-5 Supervisor Mode Address Space

*Note: In 32-bit mode, bit31 is sign-extended through bits 63~32. Failure results in an address error exception.

Table 8-2 32-bit and 64-bit Supervisor Mode Segments

Address Bit Values	Status Register Bit Values				Segment Name	Address Range	Segment Size
	KSU	EXL	ERL	SX			
32-bit A (31) = 0	01_2	0	0	0	*suseg*	0x0000 0000 through 0x7FFF FFFF	2 Gbyte (2^{31} bytes)
32-bit A (31~29) = 110_2	01_2	0	0	0	*ssseg*	0xC000 0000 through 0xDFFF FFFF	512 Mbytes (2^{29} bytes)
64-bit A (63~62) = 00_2	01_2	0	0	1	*xsuseg*	0x0000 0000 0000 0000 through 0x0000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)
64-bit A (63~62) = 01_2	01_2	0	0	1	*xsseg*	0x4000 0000 0000 0000 through 0x4000 00FF FFFF FFFF	1 Tbyte (2^{40} bytes)
64-bit A (63~62) = 11_2	01_2	0	0	1	*csseg*	0xFFFF FFFF C000 0000 through 0xFFFF FFFF DFFF FFFF	512 Mbytes (2^{29} bytes)

32-bit Supervisor Mode, User Space (*suseg*)

In Supervisor mode, when $SX = 0$ in the *Status* register and the most-significant bit of the 32-bit virtual address is set to 0, the *suseg* virtual address space is selected; it covers the full 2^{31} bytes (2 Gbytes) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. This mapped space starts at virtual address 0x0000 0000 and runs through 0x7FFF FFFF.

32-bit Supervisor Mode, Supervisor Space (*sseg*)

In Supervisor mode, when $SX = 0$ in the *Status* register and the three most-significant bits of the 32-bit virtual address are 110_2, the *sseg* virtual address space is selected; it covers 2^{29} bytes (512 Mbytes) of the current supervisor address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. This mapped space begins at virtual address 0xC000 0000 and runs through 0xDFFF FFFF.

64-bit Supervisor Mode, User Space (*xsuseg*)

In Supervisor mode, when $SX = 1$ in the *Status* register and bits 63:62 of the virtual address are set to 00_2, the *xsuseg* virtual address space is selected; it covers the full 2^{40} bytes (1 Tbyte) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. This mapped space starts at virtual address 0x0000 0000 0000 0000 and runs through 0x0000 00FF FFFF FFFF.

64-bit Supervisor Mode, Current Supervisor Space (*xsseg*)

In Supervisor mode, when $SX = 1$ in the *Status* register and bits 63~62 of the virtual address are set to 01_2, the *xsseg* current supervisor virtual address space is selected. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. This mapped space begins at virtual address 0x4000 0000 0000 0000 and runs through 0x4000 00FF FFFF FFFF.

64-bit Supervisor Mode, Separate Supervisor Space (*csseg*)

In Supervisor mode, when $SX = 1$ in the *Status* register and bits 63~62 of the virtual address are set to 11_2, the *csseg* separate supervisor virtual address space is selected. Addressing of the csseg is compatible with addressing *sseg* in 32-bit mode. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. This mapped space begins at virtual address 0xFFFF FFFF C000 0000 and runs through 0xFFFF FFFF DFFF FFFF.

8.3.3 Kernel Mode Operations

The processor operates in Kernel mode when the Status register contains one or more of the following values:

- $KSU = 00_2$
- $EXL = 1$
- $ERL = 1$

In conjunction with these bits, the *KX* bit in the Status register selects between 32- or 64-bit Kernel mode addressing:

- when $KX = 0$, 32-bit kernel space is selected and all TLB misses are handled by the 32-bit TLB refill exception handler
- when $KX = 1$, 64-bit kernel space is selected and all TLB misses are handled by the 64-bit XTLB refill exception handler

The processor enters Kernel mode whenever an exception is detected and it remains in Kernel mode until an Exception Return (ERET) instruction is executed and results in ERL and/or EXL = 0. The ERET instruction restores the processor to the mode existing prior to the exception.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 8-6. Table 8-3 lists the characteristics of the 32-bit kernel mode segments, and Table 8-4 lists the characteristics of the 64-bit kernel mode segments.

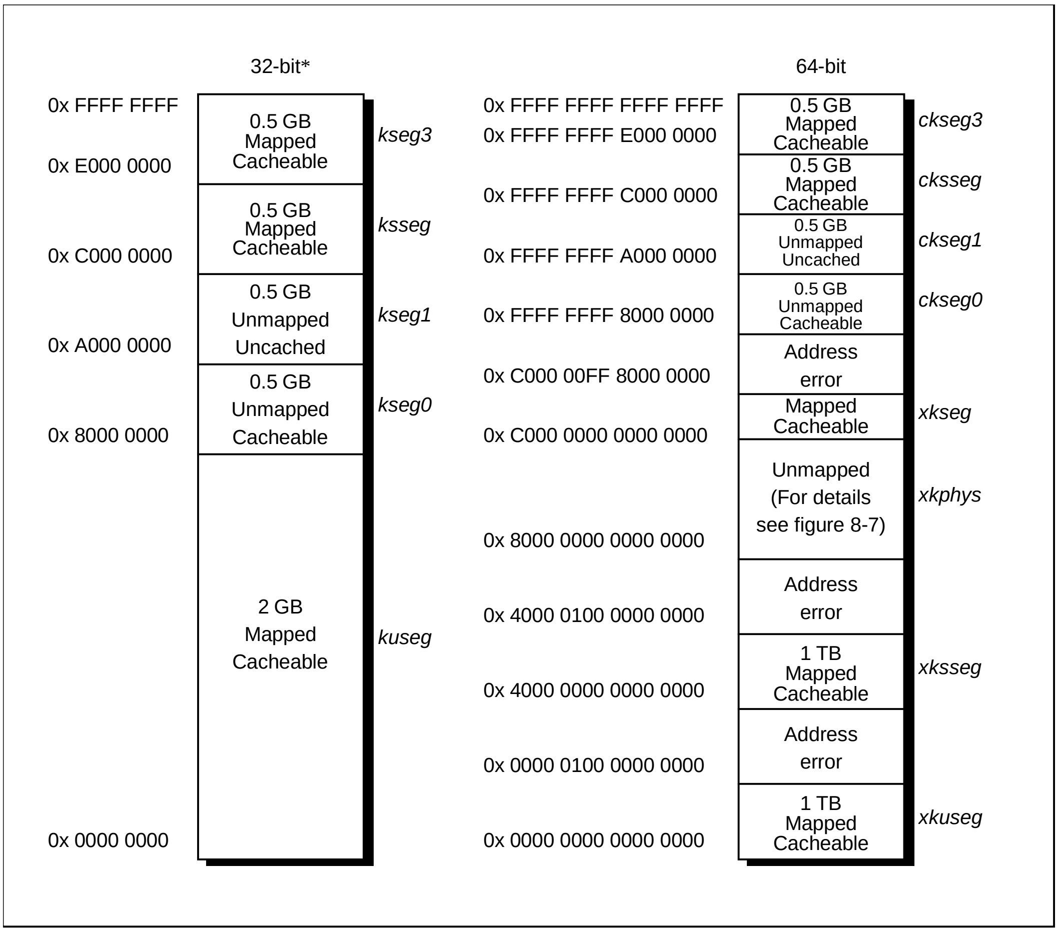


Figure 8-6 Kernel Mode Address Space

*Note 1: In 32-bit mode, bit 31 is sign-extended through bits 63~32. Failure results in an address error exception.

*Note 2: 0xff00_0000 through 0xff3f_ffff in 32-bit mode and 0xffff_ffff_ff00_0000 through 0xffff_ffff_ff3f_ffff in 64-bit mode are reserved (unmapped, uncached) for use by registers in the Debug Support Unit and TX49 MCU peripherals.

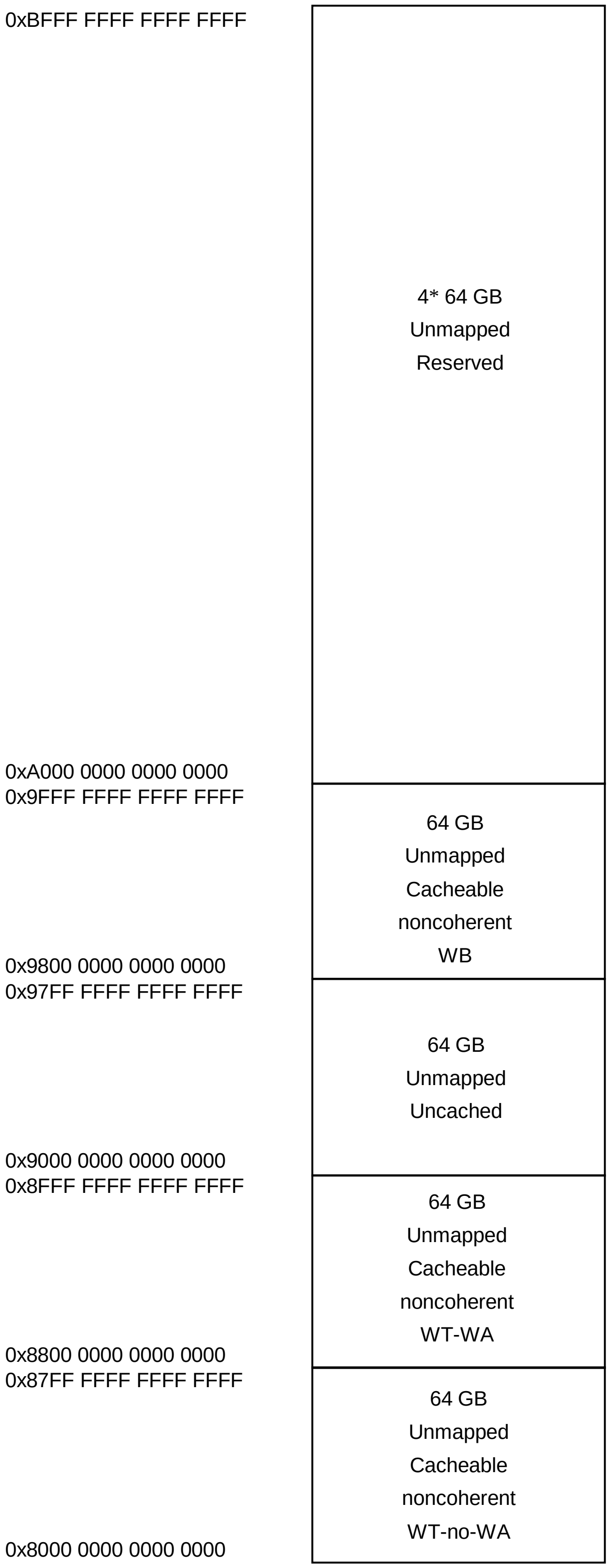


Figure 8-7 xkphys Address Space

Table 8-3 32-bit Kernel Mode Segments

Address Bit Values	Status Register Is One Of These Values				Segment Name	Address Range	Segment Size
	KSU	EXL	ERL	KX			
A (31) = 0	KSU = 00_2 or EXL = 1 or ERL = 1			0	*Kuseg*	0x0000 0000 through 0x7FFF FFFF	2 Gbyte (2^{31} bytes)
A (31~29) = 100_2				0	*Kseg0*	0x8000 0000 through 0x9FFF FFFF	512 Mbytes (2^{29} bytes)
A (31~29) = 101_2				0	*Kseg1*	0xA000 0000 through 0xBFFF FFFF	512 Mbytes (2^{29} bytes)
A (31~29) = 110_2				0	*Ksseg*	0xC000 0000 through 0xDFFF FFFF	512 Mbytes (2^{29} bytes)
A (31~29) = 111_2				0	*Kseg3*	0xE000 0000 through 0xFFFF FFFF	512 Mbytes-4 Mbytes (2^{29} bytes)
				0	(*Reserved*)	0xFF00 0000 through 0xFF3F FFFF	4 Mbytes

32-bit Kernel Mode, User Space (*kuseg*)

In Kernel mode, when *KX* = 0 in the *Status* register, and the most-significant bit of the virtual address, A31, is cleared, the 32-bit *kuseg* virtual address space is selected; it covers the full 2^{31} bytes (2 Gbytes) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. When ERL = 1 in the *Status* register, the user address region becomes a 2^{31} bytes unmapped (that is, mapped directly to physical addresses) uncached address space.

32-bit Kernel Mode, Kernel Space 0 (*kseg0*)

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the virtual address are 100_2, 32-bit *kseg0* virtual address space is selected; it is the 2^{29} bytes (512 Mbyte) kernel physical space. References to *kseg0* are not mapped through the TLB; the physical address selected is defined by subtracting 0x8000 0000 from the virtual address. The *K0* field of the *Config* register, described in this chapter, controls cacheability and coherency.

32-bit Kernel Mode, Kernel Space 1 (*kseg1*)

In Kernel mode, when *KX* = 0 in the Status register and the most-significant three bits of the 32-bit virtual address are 101_2, 32-bit *kseg1* virtual address space is selected; it is the 2^{29} bytes (512 Mbyte) kernel physical space. References to *kseg1* are not mapped through the TLB; the physical address selected is defined by subtracting 0xA000 0000 from the virtual address. Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

32-bit Kernel Mode, Supervisor Space (*ksseg*)

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are 110_2, the *ksseg* virtual address space is selected; it is the current 2^{29} bytes (512 Mbyte) supervisor virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

32-bit Kernel Mode, Kernel Space 3 (*kseg3*)

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit vital address are 111_2, the *kseg3* virtual address space is selected; it is the current 2^{29} bytes (512 Mbyte-4 Mbyte) kernel virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

Note: These is the 4 Mbytes Reserved area, begin at virtual address 0xFF00_0000 and runs through 0xFF3F_FFFF.

Table 8-4 64-bit Kernel Mode Segments

Address Bit Values	Status Register Is One Of These Values				Segment Name	Address Range	Segment Size
	KSU	EXL	ERL	KX			
A (63~62) = 00_2	KSU = 00_2 or EXL = 1 or ERL = 1			1	*xkuseg*	0x0000 0000 0000 0000 through 0x0000 00FF FFFF FFFF	1 Tbytes (2^{40} bytes)
A (63~62) = 01_2				1	*xksseg*	0x4000 0000 0000 0000 through 0x4000 00FF FFFF FFFF	1 Tbytes (2^{40} bytes)
A (63~62) = 10_2				1	*xkphys*	0x8000 0000 0000 0000 through 0xBFFF FFFF FFFF FFFF	$8*2^{32}$ bytes
A (63~62) = 11_2				1	*xkseg*	0xC000 0000 0000 0000 through 0xC000 00FF 7FFF FFFF	$2^{40}-2^{31}$ bytes
A (63~62) = 11_2 A (61~31) = -1				1	*ckseg0*	0xFFFF FFFF 8000 0000 through 0xFFFF FFFF 9FFF FFFF	512 Mbytes (2^{29} bytes)
A (63~62) = 11_2 A (61~31) = -1				1	*ckseg1*	0xFFFF FFFF A000 0000 through 0xFFFF FFFF BFFF FFFF	512 Mbytes (2^{29} bytes)
A (63~62) = 11_2 A (61~31) = -1				1	*cksseg*	0xFFFF FFFF C000 0000 through 0xFFFF FFFF DFFF FFFF	512 Mbytes (2^{29} bytes)
A (63~62) = 11_2 A (61~31) = -1				1	*ckseg3*	0xFFFF FFFF E000 0000 through 0xFFFF FFFF FFFF FFFF	512 Mbytes -4 Mbyte
				1	(*Reserved*)	0xFFFF FFFF FF00 0000 through 0xFFFF FFFF FF3F FFFF	4 Mbytes

64-bit Kernel Mode, User Space (*xkuseg*)

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63~62 of the 64-bit virtual address are 00_2, the *xkuseg* virtual address space is selected; it covers the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

When ERL = 1 in the *Status* register, the user address region becomes a 2^{31} bytes unmapped (that is, mapped directly to physical addresses) uncached address space.

64-bit Kernel Mode, Current Supervisor Space (*xksseg*)

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63~62 of the 64-bit virtual address are 01_2, the *xksseg* virtual address space is selected; it is the current supervisor virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

64-bit Kernel Mode, Physical Spaces (*xkphys*)

In Kernel mode, when $KX = 1$ in the *Status* register and bits 63~62 of the 64-bit virtual address are 10_2, one of the two unmapped *xkphys* address spaces are selected, either cached or uncached. Accesses with address bits 58~36 not equal to 0 cause an address error.

References to this space are not mapped; the physical address selected is taken from bits 35~0 of the virtual address. Bits 61~59 of the virtual address specify the cacheability and coherency attributes, as shown in Table 8-5.

Table 8-5 Cacheability and Coherency Attributes

Value(61~59)	Cacheability and Coherency Attributes	Starting Address
0	Cacheable, non-coherent, write-through, no write allocate	0x8000 0000 0000 0000
1	Cacheable, non-coherent, write-through, no write allocate	0x8800 0000 0000 0000
2	Uncached	0x9000 0000 0000 0000
3	Cacheable, non-coherent	0x9800 0000 0000 0000
4-7	Reserved	0xA000 0000 0000 0000

64-bit Kernel Mode, Kernel Space (*xkseg*)

In Kernel mode, when $KX = 1$ in the *Status* register and bits 63~62 of the 64-bit virtual address are 11_2, the address space selected is one of the following:

- kernel virtual space, *xkseg*, the current kernel virtual space; the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address
- one of the four 32-bit kernel compatibility spaces, as described in the next section.

64-bit Kernel Mode, Compatibility Spaces (*ckseg1~0, cksseg, ckseg3*)

In Kernel mode, when $KX = 1$ in the *Status* register, bits 63~62 of the 64-bit virtual address are 11_2, and bits 61~31 of the virtual address equal-1, the lower two bytes of address, as shown in Figure 8-6, select one of the following 512 Mbytes compatibility spaces.

- *ckseg0*. This 64-bit virtual address space is an unmapped region, compatible with the 32-bit address model *kseg0*. The *K0* field of the *Config* register, described in this chapter, controls cacheability and coherency.
- *ckseg1*. This 64-bit virtual address space is an unmapped and uncached region, compatible with the 32-bit address model *kseg1*.
- *cksseg*. This 64-bit virtual address space is the current supervisor virtual space, compatible with the 32-bit address model *ksseg*.
- *ckseg3*. This 64-bit virtual address space is kernel virtual space, compatible with the 32-bit address model *kseg3*.

8.4 Translation Lookaside Buffer

8.4.1 Joint TLB

The TX49 has a fully associative TLB which maps 48 pairs (odd/even entry) of virtual pages to their corresponding physical addresses.

8.4.2 TLB Entry format

32-bit addressing

127 121	120 109	108 96
0	MASK	0

95 77	76	75 72	71 64
VPN2	G	0	ASID

63 62	61 38	37 35	34	33	32
0	PFN	C	D	V	0

31 30	29 6	5 3	2	1	0
0	PFN	C	D	V	0

64-bit addressing

255 217	216 205	204 192
0	MASK	0

191 190	189 168	167 141	140	139 136	135 128
R	0	VPN2	G	0	ASID

127 94	93 70	69 67	66	65	64
0	PFN	C	D	V	0

63 30	29 6	5 3	2	1	0
0	PFN	C	D	V	0

MASK : Page comparison mask. This field sets the variable page size for each TLB entry.

VPN2 : Virtual page number divided by two (maps to two pages)

ASID : Address space ID field.

R : Region. (00: user, 01: supervisor, 11: kernel) used to match Vaddr63~62.

PFN : Page frame number; upper bits of the physical address.

C : Specifies the cache algorithm to be used (see the “C” field of the EntryLo0, 1).

D : Dirty. If this bit is set, the page is marked as dirty and therefore, writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.

V : Valid. If this bit is set, it indicates that the TLB entry is valid. If a cache hit occurs through a TLB entry when this bit is cleared, a TLB invalid exception occurs.

G : Global. If this bit is set in both Lo0 and Lo1, then ignore the ASID during TLB lookup.

0 : Reserved. Returns zeroes when read.

8.4.3 Instruction-TLB

The TX49 has a 2-entry instruction TLB (ITLB). Each ITLB entry is a subset of any single JTLB entry. The ITLB is completely invisible to software.

8.4.4 Data-TLB

The TX49 has a 4-entry data TLB (DTLB). Each DTLB entry is a subset of any single JTLB entry. The DTLB is completely invisible to software.

8.5 Virtual-to-Physical Address Translation Process

During virtual-to-physical address translation, the CPU compares the 8-bit ASID (if the Global bit, G, is not set) of the virtual address to the ASID of the TLB entry to see if there is a match. One of the following comparisons are also made:

- In 32-bit mode, the highest 7 to 19 bits (depending upon the page size) of the virtual address are compared to the contents of the TLB VPN2 (virtual page number divided by two).
- In 64-bit mode, the highest 15 to 27 bits (depending upon the page size) of the virtual address are compared to the contents of the TLB VPN2 (virtual page number divided by two).

If a TLB entry matches, the physical address and access control bits (*C*, *D*, and *V*) are retrieved from the matching TLB entry. While the *V* bit of the entry must be set for a valid translation to take place, it is not involved in the determination of a matching TLB entry.

Figure 8-8 illustrates the TLB address translation process.

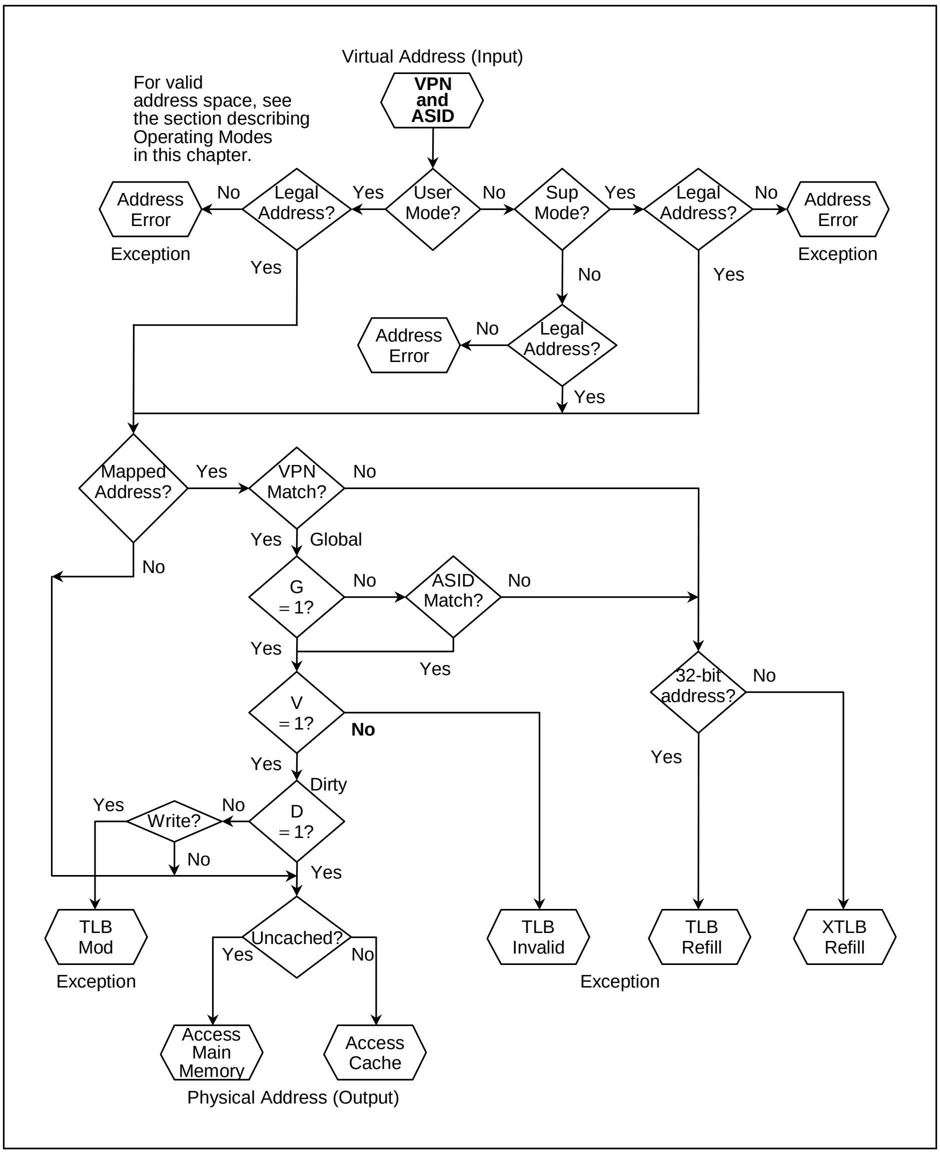


Figure 8-8 TLB Address Translation

TLB Misses

If there is no TLB entry that matches the virtual address, a TLB refill exception occurs. (TLB refill exceptions are described in Chapter 11.) If the access control bits (D and V) indicate that the access is not valid, a TLB modification or TLB invalid exception occurs. If the C bits equal 010_2, the physical address that is retrieved accesses main memory, bypassing the cache.

TLB Instructions

Table 8-6 lists the instructions that the CPU provides for working with the TLB. See Appendix A for a detailed description of these instructions.

Table 8-6 TLB Instructions

Op Code	Description of Instruction
TLBP	Translation Lookaside Buffer Probe
TLBR	Translation Lookaside Buffer Read
TLBWI	Translation Lookaside Buffer Write Index
TLBWR	Translation Lookaside Buffer Write Random

9. Cache Organization

9.1 Introduction

This chapter describes the cache memory of TX49. This processor has two on-chip primary caches for instruction and data. Both caches are configured as either 8 K-byte, 16 K-byte or 32 K-byte in size.

9.2 Instruction Cache (I-Cache)

The TX49 primary I-cache has the following characteristics:

- Cache size: 8 KB/ 16 KB/ 32 KB (fixed in each products)
- Four-way set associative
- FIFO replacement
- Indexed with a virtual address
- Checked with a physical tag
- Block (line) size: 8 words (32 bytes)
- Burst refill size: 8 words (32 bytes)
- Lockable on a per-line basis (way1, way2 and way3)
- All valid bits, lock and FIFO bits are cleared by a Reset exception

9.2.1 Instruction Cache Address Field

Figure 9-1 shows the instruction cache address field. When 4-KB page size is used in 32 KB Instruction cache, the bit 12 of the Physical Address and the Virtual Address must be same value.

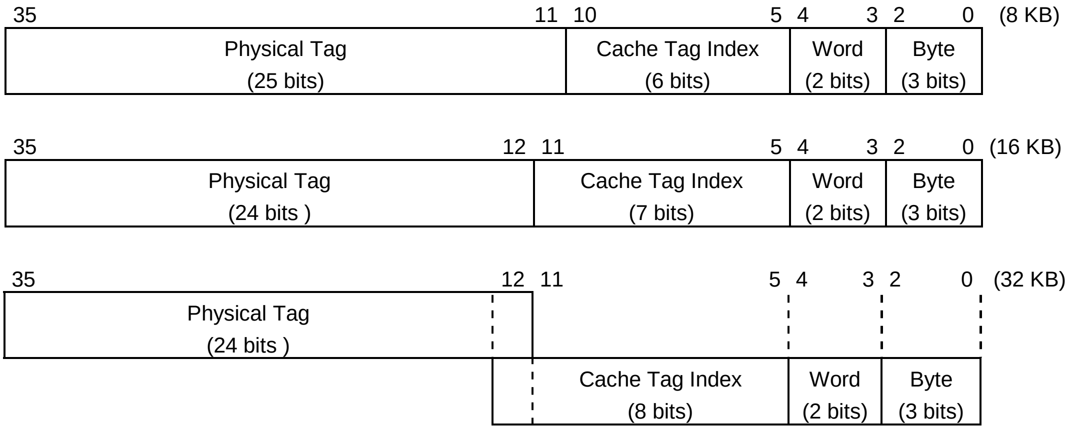
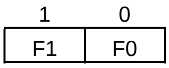


Figure 9-1 Instruction Cache Address Field

9.2.2 Instruction Cache Configuration

Each line in the 4 ways of the instruction cache share FIFO replacement bits. Figure 9-2 shows the format of replacement bits. These bits are shared by way0, way1, way2 and way3 for 8 KB/ 16 KB/ 32 KB cache, and indicate next set to which replacement will be directed; when lock bit is set to 1, indicate this set is not locked.

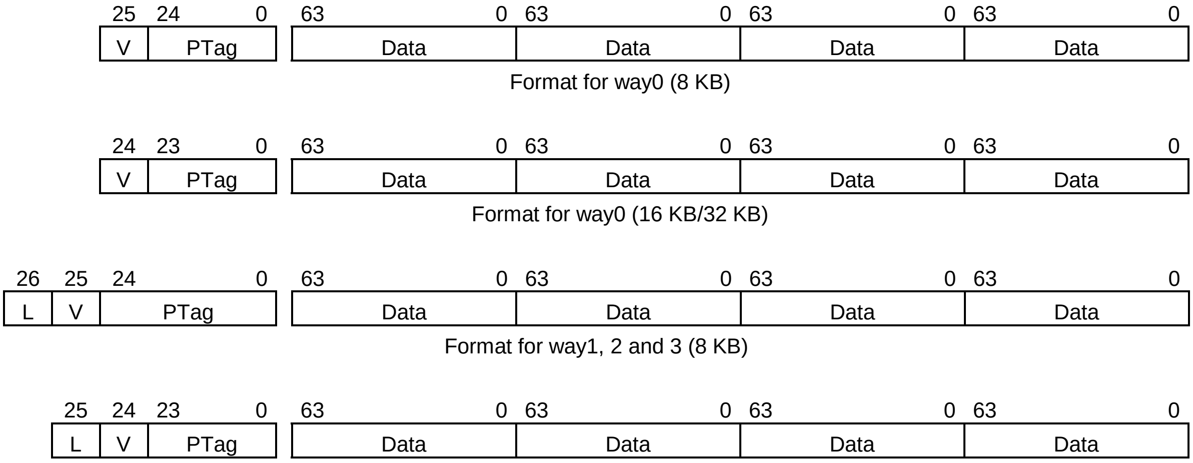
Each line of instruction cache data has an associated 27-bit (8 KB)/26-bit (16 KB/32 KB) tag that contains a 25-bit (8 KB)/24-bit (16 KB/32 KB) physical address, a single Lock bit and a single valid bit, except for the line in way0, which has an 26-bit (8 KB)/25-bit (16 KB/32 KB) tag that excludes a lock bit. Figure 9-3 shows the formats of tag and data pair.



F0: FIFO replace bit 0

F1: FIFO replace bit 1

Figure 9-2 Format of Replacement Bits



Format for way0 (8 KB)

Format for way0 (16 KB/32 KB)

Format for way1, 2 and 3 (8 KB)

Format for way1, 2 and 3 (16 KB/32 KB)

L: Lock bit (1: enable, 0: disable)

V: Valid bit (1: valid, 0: invalid)

PTag: Physical tag (bit 35~12 of the physical address)

Data: Instruction cache data

Figure 9-3 Format of Tag and Data Pair for I-cache

9.3 Data Cache

The TX49 primary D-cache has the following characteristics:

- Cache size: 8 KB/ 16 KB/ 32 KB (fixed in each products)
- Four-way set associative
- FIFO replacement
- Indexed with a virtual address
- Checked with a physical tag
- Block (line) size: 8 words (32 bytes)
- Burst refill size: 8 words (32 bytes)

- Lockable on a per-line basis (way1, way2 and way3)
- Store buffer
- Selectable write-back and write-through on a page basic
- All W, CS, FIFO and Lock bits are cleared by a Reset exception

9.3.1 Data Cache Address Field

Figure 9-4 shows the data cache address field. When 4-KB page size is used in 32 KB Instruction cache, the bit 12 of the Physical Address and the Virtual Address must be same value.

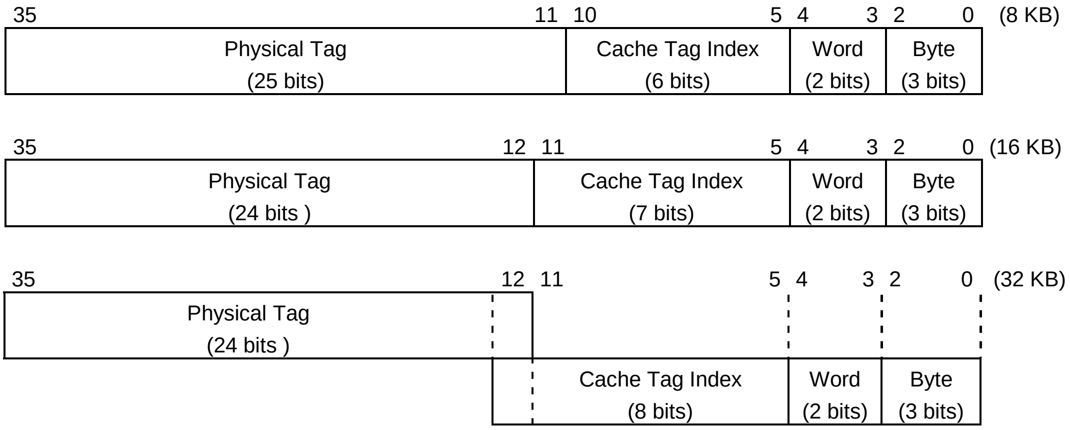
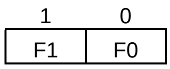


Figure 9-4 Data Cache Address Field

9.3.2 Data Cache Configuration

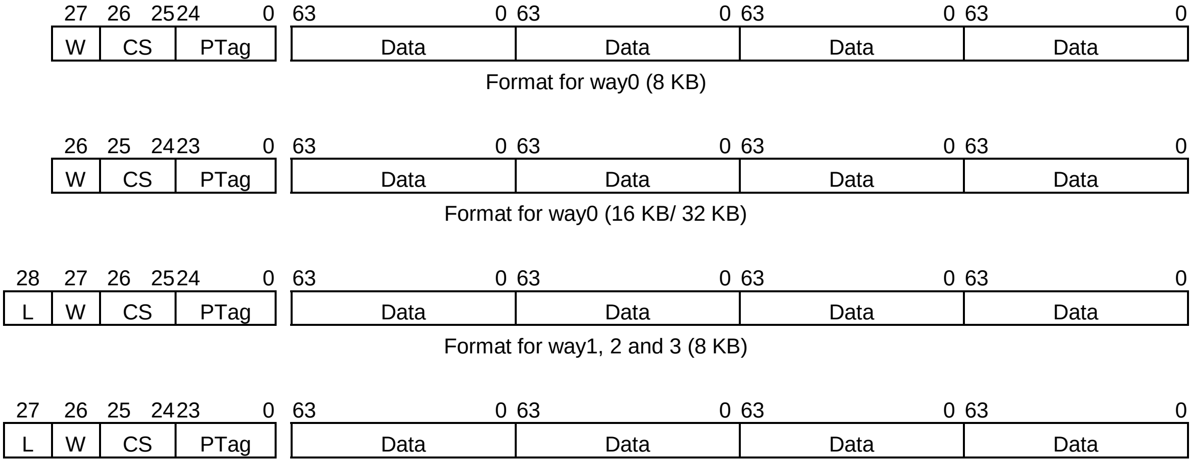
Each line in the 4 ways of the data cache share F1, F0 replacement bits. Figure 9-5 shows the format of replacement bits. These bits are shared by way0, way1, way2 and way3 for 8 KB/ 16 KB/ 32 KB cache, and indicate next set to which replacement will be directed; when lock bit is set to 1, indicate this set is not locked.

Each line of data cache data has an associated 29-bit/28-bit tag that contains a 25-bit/24-bit physical address, a single Lock bit, a single write-back bit and a 2-bit cache state, except for the line in way0, which has an 28-bit/27-bit tag that excludes a Lock bit. Figure 9-6 shows the formats of tag and data pair.



F0: FIFO replace bit 0
F1: FIFO replace bit 1

Figure 9-5 Format of Replacement Bits


27 26 25 24 0 63 0 63 0 63 0 63 0
W | CS | PTag | Data | Data | Data | Data
Format for way0 (8 KB)
26 25 24 23 0 63 0 63 0 63 0 63 0
W | CS | PTag | Data | Data | Data | Data
Format for way0 (16 KB/ 32 KB)
28 27 26 25 24 0 63 0 63 0 63 0 63 0
L | W | CS | PTag | Data | Data | Data | Data
Format for way1, 2 and 3 (8 KB)
27 26 25 24 23 0 63 0 63 0 63 0 63 0
L | W | CS | PTag | Data | Data | Data | Data
Format for way1, 2 and 3 (16 KB/ 32 KB)

L: Lock bit (1: enable, 0: disable)
W: Write-back bit (set if cache line has written)
CS: Primary cache state
(0: Invalid, 1: Reserved, 2: Reserved, 3: Valid)
PTag: Physical tag (bit 35~12 of the physical address)
Data: Data cache data

Figure 9-6 Format of Tag and Data Pair for D-cache

In the TX49, the W (write-back) bit, not the cache state, indicates when the primary cache contents modified data that must be written back to memory. The states Invalid and Valid are used to describe the cache line. That is, there is no hardware support for cache coherency.

9.3.3 Data Cache Policies

The TX49 provides three write policy options for the data cache: two write-through modes and one write-back mode. Selection of a write policy is done by the K0 bit in the Config register for the kseg0 segment and the C bit within each TLB entry for the other segments. For a description of the K0 bit, see Table 7-15; for a description of the C bit, see Table 7-3.

The write policy should not be changed once the cache is initialized; otherwise, the contents of the data cache are not guaranteed.

a) Write-through modes (write allocate/no write allocate)
In write-through, the data is written to cache and to main memory at the same time. On a cache store miss, a write-through without write-allocate causes data to be sent only to main memory, whereas a write-through with write-allocate causes the relevant cache line to be replaced before being sent to the data cache and main memory.

b) Write-back mode
In the write-back policy, a copy of the data is written to cache by the processor, but not to main memory. The data will be written to main memory only if cache's copy is about to be replaced.

9.4 FIFO Replacement Algorithm

The TX49 uses the FIFO (first in, first out) policy when overwriting the blocks of data in its instruction and data caches.

- Typically, data items in way0, way1, way2 and way3 are replaced in this order.
- The FIFO[1:0] bits do not point at locked and valid lines.
- Invalid lines, if any, are replaced first.
- The FIFO replacement bits are altered when external data is written to the cache or via the CACHE instruction.

Figure 9-7 shows several examples of how the FIFO replacement bits change due to cache line replacements.

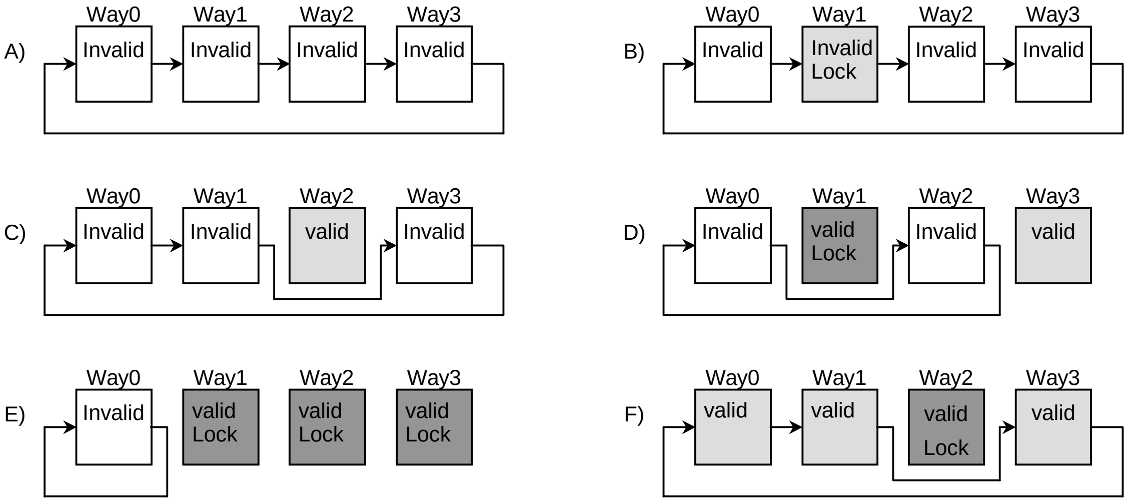


Figure 9-7 FIFO Replacement Policy

9.5 Lock function

The lock function can be used to locate critical instruction/data in one instruction/data cache set and they are not replaced when the lock bit is set.

9.5.1 Lock bit setting and clearing

Setting the Lock bit in each line cache enable the instruction/data cache lock function. When the lock function is enabled, the instruction/data in the valid line is locked and never be replaced. The set to be locked is pointed by FIFO bit. Refilled instruction/data during the lock function is enabled is locked. When a store miss occurs for the write-through data cache without write allocate, the store data is not written to the cache and will therefore not be locked.

The lock function is disabled by clearing the Lock bit in each line.

In order to clear or set the Lock bit in the cache, Cache instructions (Index store I-cache /D-cache Tag) can be used, and in order to load the instruction/data to cache from memory, another Cache instructions (Fill I-cache/D-cache) can be used (refer to Cache instruction).

Clear the lock bit as follows when data written to a locked line should be stored in main memory.

(1) Read the locked data from cache memory

(2) Clear the lock bit

(3) Store the data that was read

9.5.2 Operation During Lock

After the lock bit is set for a line, the line can be replaced only when it's line state is invalid. The locked valid line can never be replaced. FIFO bit should point only to the set of locked invalid line or unlocked line.

A write access to a locked valid line takes place only to the cache not to the memory at Write Back mode. Both of the cache and the memory are replaced at Write Through mode.

9.5.3 Example of Data Cache Locking

During the load operation to the locked line of the cache, any interrupt should be disabled in order to avoid to lock the wrong data.

To lock data cache lines, the following sequence of codes could be used.

```
.......................             /* Disable the interrupt */
mtc0    t0, TagLo                   /* Load data into TagLo reg */
cache   2 (D), offset (base)        /* Invalidate and lock line in desired set using
                                       Index_Store_Tag cache instruction */
cache   7 (D), offset (base)        /* Fill the cache line from desired memory location */
.......................             / Enable the interrupt */
```

9.5.4 Example of Instruction Cache Locking

To lock instruction cache lines, the following sequence of codes could be used:

```
.......................             /* Disable the interrupt */
mtc0    t0, TagLo                   /* Load data into TagLo reg */
cache   2 (I), offset (base)        /* Invalidate and lock line in desired set using
                                       Index_Store_Tag cache instruction */
cache   5 (I), offset (base)        /* Fill the cache line from desired memory location */
.......................             /* Enable the interrupt */
```

9.6 The Primary Cache Accessing

Figure 9-8 shows the virtual address (VA) index to the primary cache. Each instruction and data cache size is 8 KB, 16 KB or 32 KB. The virtual address bits be used to index into the primary cache decided by the cache size.

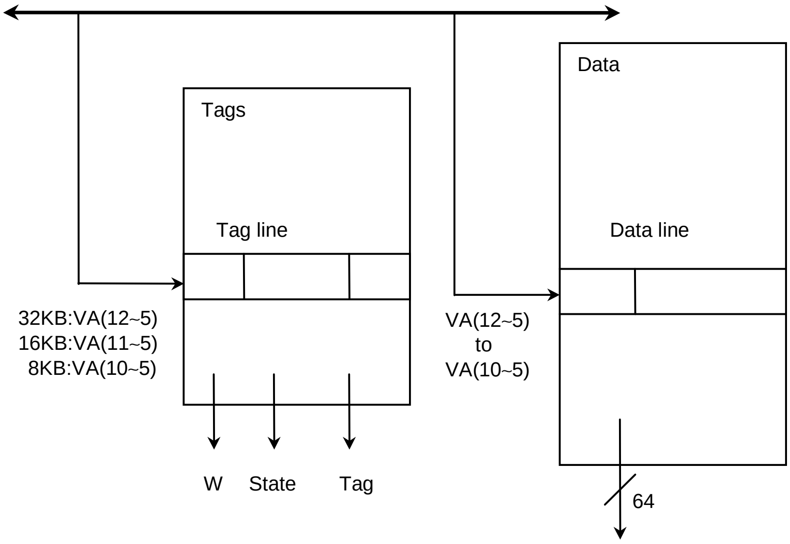


Figure 9-8 Primary Cache Data and Tag Organization

9.7 Cache States

The section describes about the state of a cache line. The cache line in the TX49 is in one of states described in Table 9-1.

The I-Cache line is in one of the following states:

- invalid
- valid

The D-Cache line is in one of the following states:

- invalid
- valid

Table 9-1 Cache States

Cache line State	Description
Invalid	A cache line that does not contain valid information must be marked invalid, and cannot be used. A cache line in any other state than invalid is assumed to contain valid information.
Valid	A Valid cache line contains valid information. The cache line may or not be consistent with memory and is owned by the processor (see Cache Line Ownership in this chapter).

9.8 Cache Line Ownership

The TX49 becomes the owner of a cache line after it writes to that cache line (that is, by entering the Valid), and is responsible for providing the contents of that line on a read request. There can only be one owner for each cache line.

9.9 Cache Multi-Hit Operation

The TX49 is not guaranteed the operation for the multi-hit of primary cache.

Thus, in case of locking the specified program/data in the primary cache, the program/data must be used after locked in the cache by Fill instruction.

Such as the previous description the cache multi hit does not guarantee in the TX49.

9.10 Cache Test Function

9.10.1 Cache Disabling

The Config register bits ICE# (Instruction Cache Enable) and DCE# (Data Cache Enable) are used to enable and disable the instruction and data cache, respectively.

When a cache is disabled, all cache accesses are misses and there is no refill (nor is there any burst bus cycle; this is the same as accessing a non-cacheable area). The Valid bit (V) or Cache State bit (CS) for each entry cannot be modified.

Notes:
When the instruction cache is disabled:

- Every instruction fetch causes a cache miss, and external memory accesses are performed using single-read bus cycles.
- The CACHE instruction can still operate on the instruction cache.

Notes:
When the data cache is disabled:

- Every load or store instruction causes a cache miss. Data cache refills are disabled, and external memory accesses occur using single-read or single-write transactions.
- The CACHE instruction can still operate on the data cache.

Notes:
How to disable the instruction cache:

- When disabling the instruction cache, instruction streaming should be discontinued by placing a jump instruction following the MTC0 instruction.

```
Example:  MTC0   Rn, Config  (Set the ICE# bit to 1)
          J      L1          (Jump to L1 and disable instruction streaming)
          NOP                (Branch delay slot)
L1:       CACHE IndexIncaliate, offset (base)
```

9.10.2 Cache Flushing

Both the instruction and data cache are flushed when a ColdReset/SoftReset exception is raised (all valid bits are cleared to 0).

The instruction cache is flushed by the CACHE instruction Index_Invalidate /Hit_Invalidate. The data cache is flushed by the CACHE instruction IndexWriteBackInvalidate/HitInvalidate/HitWriteBackInvalidate.

The processor writes the cache line back to main memory during the execution of Index Writeback Invalidate, Hit Writeback Invalidate or Hit Writeback CACHE instruction or when the modified cache line is replaced. In write-back mode, software is responsible for ensuring cache coherency.

10. Write Buffer

The TX49 contains a write buffer to improve the performance of writes to the external memory. Every write to external memory uses this on-chip write buffer. The write buffer holds up to four 64-bit address and data pairs.

For a cache miss write-back, the entire buffer is used for the write-back data and allows the processor to proceed in parallel with the memory update. For uncached and write-through stores, the write buffer uncouples the CPU from the write to memory. If the write buffer is full, additional stores will stall until there is room for them in the write buffer.

The TX49 processor core might issue a read request while the write buffer is performing a write operation. Multiple read/write operations are serviced in the following order:

- If there is only a write request, the data in the write buffer is written to an external device.
- If there is only a read request, a read operation is performed to bring in data from an external device.
- If a read request and a write request occur simultaneously, the read request is serviced first, except for the following cases:
  - when the processor issues a read request to the target address of one of the write buffer entries
  - when the processor issues an uncacheable read reference while the write buffer has uncacheable write data

The BC0T and BC0F instructions can be used to determine whether any data is present in the write buffer:

If there is data in the write buffer, the coprocessor condition signal is false (0).

If there is no data in the write buffer, the coprocessor condition signal is true (1).

Following is the assembly language code to freeze the processor until the write buffer becomes empty.

```
      SW
      NOP
      NOP
Loop: BC0F Loop
      NOP
```

The following sequence of instructions also causes the TX49 to perform the same action. Appended to a store instruction, the SYNC instruction ensures that the store instruction initiated prior to this instruction is completed before any instruction after this instruction is allowed to start.

```
SW
SYNC
```

11. CPU Exception

11.1 Introduction

This chapter describes the explanation of CPU exception processing. The chapter concludes with a description of each exception's cause, together with the manner in which the CPU processes and services these exceptions.

11.2 Exception Vector Locations

Exception vector addresses are stored in an area of kseg0 or kseg1 except for Debug exception vector. The vector address of the ColdReset, SoftReset and NMI exception is always in a non-cacheable area of kseg1. Vector addresses of the other exceptions depend on the BEV bit of Status register. When BEV is 0, these exceptions are vectored to a cacheable area of kseg0. When BEV is 1, all vector addresses are in a non-cacheable area of kseg1.

Table 11-1 shows the list of the exception vector locations.

Table 11-1 Exception Vector Locations

Exception	TX49 Vector Address (virtual address)	
	(BEV = 0)	(BEV = 1)
ColdReset, SoftReset, NMI	0xffff_ffff_bfc0_0000	0xffff_ffff_bfc0_0000
TLB refill, EXL = 0	0xffff_ffff_8000_0000	0xffff_ffff_bfc0_0200
XTLB refill, EXL = 0 (X = 64 bit TLB)	0xffff_ffff_8000_0080	0xffff_ffff_bfc0_0280
Others (common exception)	0xffff_ffff_8000_0180	0xffff_ffff_bfc0_0380

Exception	TX49 Vector Address (physical address)	
	(BEV = 0)	(BEV = 1)
ColdReset, SoftReset, NMI	0x0_1fc0_0000	0x0_1fc0_0000
TLB refill, EXL = 0	0x0_0000_0000	0x0_1fc0_0200
XTLB refill, EXL = 0 (X = 64 bit TLB)	0x0_0000_0080	0x0_1fc0_0280
Others (common exception)	0x0_0000_0180	0x0_1fc0_0380

The cache error exception is not occurred because the TX49 does not have the parity bit into the primary cache. Debug exception needs the care, it has the special address. (See 14.9.5) Table 11-2 shows the list of the debug exception vector locations.

Table 11-2 Debug Exception Vector Locations

Exception	TX49 Debug Exception Vector Address (virtual address)	
	(ProbEnb = 0)	(ProbEnb = 1)
Debug	0xffff_ffff_bfc0_0400	0xffff_ffff_ff20_0200

Exception	TX49 Debug Exception Vector Address (physical address)	
	(ProbEnb = 0)	(ProbEnb = 1)
Debug	0x0_1fc0_0400	0xf_ff20_0200

11.3 Priority of Exception

More than one exception may be raised for the same instruction, in which case only the exception with the highest priority is reported. The TX49 Processor Core instruction exception priority is shown in Table 11-3.

Table 11-3 Priority of Exception

Priority	Exception		Mnemonic
	Cold Reset		
	Soft Reset		
High	NMI		
	Address error	Inst. Fetch	AdEL
	TLB refill	Inst. Fetch	TLBL
	TLB invalid	Inst. Fetch	TLBL
	Bus error	Inst. Fetch	IBE
	Integer overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unusable, or Floating-Point Exception		Ov, Tr, Sys, Bp, RI, CpU, FPE
	Address error	Data access	AdEL/AdES
	TLB refill	Data access	TLBL/TLBS
	TLB invalid	Data access	TLBL/TLBS
Low	TLB modified	Data write	Mod
	Bus error	Data access	DBE
	Interrupt		Int

General exceptions (i.e., exceptions other than debug exceptions) are prioritized as follows:

1. If more than one exception condition occurs for a signal instruction, only the exception with the highest priority is reported, as shown in Table 11-3 (from highest to lowest priority).
2. If two instructions cause exception conditions in the M and E stages of the pipeline simultaneously, the instruction in the M stage causes the processor to take an exception.
3. When 64-bit instructions are executed in 32-bit mode, the Reserved Instruction (RI) exception can occur simultaneous with other exception, as shown below. In that case, the RI exception is given precedence.
   - RI and CpU
   - RI and Ov
   - RI and AdEL/S (data)
   - RI and TLBL/S (data)

General and debug exceptions are prioritized as follows:

1. If a general exception condition and a debug exception condition occur for a single instruction, the debug exception is serviced first, and then the general exception is serviced.
2. If two instructions cause exception conditions in the M and E stages of the pipeline simultaneously, only the instruction in the M stage generates an exception.

For details on debug exceptions, see Section 14.9.

11.4 ColdReset Exception

11.4.1 Cause

This ColdReset exception occurs when the GCOLDRESET* signal is asserted and then deasserted. This exception is not maskable.

11.4.2 Processing

A special interrupt vector that resides in an unmapped and uncached area is used. It is therefore not necessary for hardware to initialize TLB and cache memory in order to process this exception. The vector location of this exception is;

- In 32 bit mode, 0xbfc0 0000 (virtual address), 0x0_1fc0_0000 (physical address)
- In 64 bit mode, 0xffff ffff bfc0 0000 (virtual address), 0x0_1fc0_0000 (physical address)

The most register's contents are cleared when this exception occurs. The values of these bits are listed into the table of Section 7.

Valid bits, Lock bits and FIFO replacement bits in the instruction cache are all cleared to 0. W bits, CS bits, Lock bits and FIFO replacement bits in the data cache are all cleared to 0.

If a ColdReset exception occurs during bus cycle, the current bus cycle is aborted and an exception is taken.

11.4.3 Servicing

The ColdReset exception is serviced by;

- initializing all registers, coprocessor registers, caches and the memory system
- performing diagnostic tests
- bootstrapping the operating system

11.5 SoftReset Exception

11.5.1 Cause

This SoftReset exception occurs when the GRESET* signal is asserted and then deasserted. This exception is not maskable.

11.5.2 Processing

A special interrupt vector that resides in an unmapped and uncached area is used. It is therefore not necessary for hardware to initialize TLB and cache memory in order to process this exception. The vector location of this exception is;

- In 32 bit mode, 0xbfc0 0000 (virtual address), 0x0_1fc0_0000 (physical address)
- In 64 bit mode, 0xffff ffff bfc0 0000 (virtual address), 0x0_1fc0_0000 (physical address)

All register contents are retained except for the following.

- ErrorEPC register, which contains the restart PC
- ERL, SR and BEV bits of Status register, which are set to “1”

Because Soft-reset exception can abort cache and bus operations, cache and memory state is undefined when this exception occurs.

11.5.3 Servicing

The SoftReset exception is serviced by saving the current processor state for diagnostic purposes, and reinitializing for the ColdReset exception.

11.6 NMI (Non-maskable Interrupt) Exception

11.6.1 Cause

The NMI (Non-maskable Interrupt) exception occurs at the falling edge of the GNMI* signal. This interrupt is not maskable, and occurs regardless of the EXL, ERL and IE bits of the Status register.

11.6.2 Processing

The same special interrupt vector as for Cold-reset/Soft-reset exception (0xbfc0_0000/ 0xffff_ffff_bfc0_0000). This vector is located within unmapped and uncached area so that the cache and TLB need not be initialized to process this exception. When this exception occurs, the SR bit of Status register is set.

Because NMI exception can occur in the midst of another exception, it is not normally possible to continue program execution after servicing NMI exception.

Unlike the Cold-reset/Soft-reset exception, but like other exceptions, this exception occurs at an instruction boundary. The state of the primary cache and memory system are preserved by this exception.

All register contents are retained except for the following.

- ErrorEPC register, which contains the restart PC
  If the exception-causing instruction is in a branch delay slot, the ErrorEPC register points at the preceding branch instruction, and the BD bit of the Cause register is set as indication.
- ERL, SR and BEV bits of the Status register, which is set to 1.

11.6.3 Servicing

The NMI exception is serviced by saving the current processor state for diagnostic purposes, and reinitializing the system for the ColdReset exception.

11.7 Address Error Exception

11.7.1 Cause

The Address Error exception occurs when an attempt is made to execute one of the following.

- load or store a doubleword that is not aligned on a doubleword boundary
- load, fetch or store a word that is not aligned on a word boundary
- load or store a halfword that is not aligned on a halfword boundary
- reference Kernel mode address while in User or Supervisor mode
- reference Supervisor mode address while in User mode

This exception is not maskable.

11.7.2 Processing

The common exception vector is used. ExcCode AdEL or AdES in Cause register is set depending on whether the memory access attempt was a load or store. When this exception is raised, the misalign virtual address causing the exception, or the protected virtual address that was illegally referenced, is placed in BadVAddr register. The contents of the VPN field of Context and EntryHi registers are undefined, as are the contents of EntryLo register.

If EXL bit of Status register is only set to 0, the following operation is executed. EPC register points to the address of the instruction causing the exception. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in EPC register and BD bit of Cause register is set to “1”.

11.7.3 Servicing

The process executing at the time is handed a segmentation violation signal. This error is usually fatal to the process incurring the exception.

11.8 TLB Refill Exception

11.8.1 Cause

The TLB refill exception occurs when there is no TLB entry to match a reference to a mapped address. This exception is not maskable.

11.8.2 Processing

There are two special exception vectors for this exception; one for references to 32-bit virtual address, and one for references to 64-bit virtual address. The KX, SX and UX bits of Status register determine whether the User, Supervisor or Kernel address referenced are 32-bit mode or 64-bit mode. When EXL bit of Status register is set to “0”, all references use these vectors. When this exception occurs, TLBL or TLBS code is set in the ExcCode field of Cause register. This code indicates whether the instruction, as shown by EPC register and BD bit of Cause register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs;

- BadVAddr, Context, XContext and EntryHi registers hold the virtual address failed address translation
- EntryHi register contains ASID from which the translation fault occurred, too
- A valid address in which to place the replacement TLB entry is contained into Random register
- The contents of EntryLo register are undefined

If EXL bit of Status register is only set to 0, the following operation is executed. EPC register points to the address of the instruction causing the exception. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in EPC register and BD bit of Cause register is set to “1”.

11.8.3 Servicing

To service this exception, the contents of the Context or XContext register are used as a virtual address to fetch memory locations containing the physical page frame and access control bits for a pair of TLB entries. The two entries are placed into the EntryLo0/EntryLo1 register; the EntryHi and EntryLo registers are written into the TLB.

It is possible that the virtual address used to obtain the physical address and access control information is on a page that is not resident in the TLB. This condition is processed by allowing a TLB refill exception in the TLB refill handler. This second exception goes to the common exception vector because the EXL bit of the Status register is set.

11.9 TLB Invalid Exception

11.9.1 Cause

The TLB Invalid exception occurs when a virtual address reference matches a TLB entry that is marked invalid (TLB valid bit cleared). This exception is not maskable.

11.9.2 Processing

The common exception vector is used for this exception. When this exception occurs, TLBL or TLBS code is set in the ExcCode field of Cause register. This code indicates whether the instruction, as shown by EPC register and BD bit of Cause register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs;

- BadVAddr, Context, XContext and EntryHi registers hold the virtual address failed address translation
- EntryHi register contains ASID from which the translation fault occurred, too
- A valid address in which to place the replacement TLB entry is contained into Random register
- The contents of EntryLo register are undefined

If EXL bit of Status register is only set to 0, the following operation is executed. EPC register points to the address of the instruction causing the exception. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in EPC register and BD bit of Cause register is set to “1”.

11.9.3 Servicing

A TLB entry is typically marked invalid when one of the following is true;

- a virtual address does not exist
- the virtual address exists, but is not in main memory (a page fault)
- a trap is desired on any reference to the page (for example, to maintain a reference bit or during debug)

After servicing the cause of a TLB Invalid exception, the TLB entry is located with TLB Probe (TLBP) instruction, and replaced by an entry with that entry’s Valid bit set.

11.10 TLB Modified Exception

11.10.1 Cause

The TLB Modified exception occurs when a store operation virtual address reference to memory matches a TLB entry that is marked valid but is not dirty and therefore is not writable. This exception is not maskable.

11.10.2 Processing

The common exception vector is used for this exception, and Mod code in Cause register is set.

When this exception occurs;

- BadVAddr, Context, XContext and EntryHi registers hold the virtual address failed address translation
- EntryHi register contains ASID from which the translation fault occurred, too
- The contents of EntryLo register are undefined

If EXL bit of Status register is only set to 0, the following operation is executed. EPC register points to the address of the instruction causing the exception. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in EPC register and BD bit of Cause register is set to 1.

11.10.3 Servicing

The kernel uses the failed virtual address or virtual page number to identify the corresponding access control information. The page identified may or may not permit write accesses; if writes are not permitted, a write protection violation occurs.

If write accessed are permitted, the page frame is marked dirty/writable by the kernel in its own data structures. The TLB Probe (TLBP) instruction places the index of the TLB entry that must be altered into the Index register. The EntryLo register is loaded with a word containing the physical page frame and access control bits (with the D bit set), and the EntryHi and EntryLo registers are written into the TLB.

11.11 Bus Error Exception

11.11.1 Cause

The Bus Error exception occurs when GBUSERR* signal is asserted during a memory read bus cycle. This exception is raised by board-level circuitry for events such as bus time-out, backplane bus parity errors, and invalid physical memory addresses or access types. This occurs during execution of the instruction causing the bus error. The memory bus cycle ends upon notification of a bus error. When a bus error is raised during a burst refill, the following refill is not performed. A bus error request made by asserting GBUSERR* signal will be ignored if TX49 is executing a cycle other than a bus cycle. It is therefore not possible to raise a Bus Error exception in a write access using a write buffer. A general interrupt must be used instead. This exception is not maskable.

11.11.2 Processing

The common interrupt vector is used for a Bus Error exception. The IBE or DBE code in the ExcCode field of the Cause register is set, signifying whether the instruction (as indicated by the EPC register and BD bit in the Cause register) caused the exception by an instruction reference, load operation, or store operation.

The EPC register contains the address of the instruction that caused the exception, unless it is in a branch delay slot, in which case the EPC register contains the address of the preceding branch instruction and the BD bit of the Cause register is set.

11.11.3 Servicing

The physical address at which the fault occurred can be computed from information available in the CP0 registers.

- If the IBE code in the Cause register is set (indicating an instruction fetch reference), the virtual address is contained in the EPC register (or 4+ the contents of the EPC register if the BD bit of the Cause register is set).
- If the DBE code is set (indicating a load or store reference), the instruction that caused the exception is located at the virtual address contained in the EPC register (or 4+ the contents of the EPC register if the BD bit of the Cause register is set).

The virtual address of the load and store reference can then be obtained by interpreting the instruction. The physical address can be obtained by using the TLB Probe (TLBP) instruction and reading the EntryLo register to compute the physical page number.

The process executing at the time of this exception is handed a bus error signal, which is usually fatal.

11.12 Integer Overflow Exception

11.12.1 Cause

The Integer Overflow exception occurs when ADD, ADDI, SUB, DADD, DADDI or DSUB instruction results in a 2's complement overflow. This exception is not maskable.

11.12.2 Processing

The common exception vector is used for this exception, and the Ov code in Cause register is set.

If EXL bit of Status register is only set to 0, the following operation is executed. EPC register points to the address of the instruction causing the exception. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in EPC register and BD bit of Cause register is set to 1.

11.12.3 Servicing

The process executing at the time of the exception is handed a floating-point exception/integer overflow signal. This error is usually fatal to the current process.

11.13 Trap Exception

11.13.1 Cause

The Trap exception occurs when TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEIU, TLTI, TLTIU, TEQI or TNEI instruction results in a TRUE condition. This exception is not maskable.

11.13.2 Processing

The common exception vector is used for this exception, and the Tr code in Cause register is set.

If EXL bit of Status register is only set to 0, the following operation is executed. EPC register points to the address of the instruction causing the exception. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in EPC register and BD bit of Cause register is set to 1.

11.13.3 Servicing

The process executing at the time of a Trap exception is handed a floating-point exception/integer overflow signal. This error is usually fatal.

11.14 System Call Exception

11.14.1 Cause

The System Call exception occurs during an attempt to execute the SYSCALL instruction. This exception is not maskable.

11.14.2 Processing

The common exception vector is used for this exception, and the Sys code in Cause register is set.

If EXL bit of Status register is only set to 0, the following operation is executed. EPC register points to the address of the SYSCALL instruction. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in EPC register.

If the SYSCALL instruction is in a branch delay slot, BD bit of Status register is set, otherwise this bit is cleared.

11.14.3 Servicing

When this exception occurs, control is transferred to the applicable system routine.

To resume execution, the EPC register must be altered so that the SYSCALL instruction does not re-execute; this is accomplished by adding a value of 4 to the EPC register (EPC register + 4) before returning.

If a SYSCALL instruction is in a branch delay slot, a more complicated algorithm, beyond the scope of this description, may be required.

11.15 Breakpoint Exception

11.15.1 Cause

The Breakpoint exception occurs when an attempt is made to execute the BREAK instruction. This exception is not maskable.

11.15.2 Processing

The common exception vector is used for this exception, and the Bp code in Cause register is set.

If EXL bit of Status register is only set to 0, the following operation is executed. EPC register points to the address of the BREAK instruction. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in EPC register.

If the BREAK instruction is in a branch delay slot, BD bit of Status register is set, otherwise this bit is cleared.

11.15.3 Servicing

When the Breakpoint exception occurs, control is transferred to the applicable system routine. Additional distinctions can be mode by analyzing the unused bits of the BREAK instruction (bits 25~6), and loading the contents of the instruction whose address the EPC register contains. A value of 4 must be added to the contents of the EPC register (EPC register + 4) to locate the instruction if it resides in a branch delay slot.

To resume execution, the EPC register must be altered so that the BREAK instruction does not re-execute; this is accomplished by adding a value of 4 to the EPC register (EPC register + 4) before returning.

If a BREAK instruction is in a branch delay slot, interpretation of the branch instruction is required to resume execution.

11.16 Reserved Instruction Exception

11.16.1 Cause

The Reserved Instruction exception occurs when one of the following condition occurs:

- an attempt is made to execute an instruction with an undefined major opecode (bit 31~26)
- an attempt is made to execute a SPECIAL instruction with an undefined minor opcode (bit 5~0)
- an attempt is made to execute a REGIMM instruction with an undefined minor opcode (bit20~16)
- an attempt is made to execute 64-bit operations in 32-bit mode when in User or Supervisor modes
- an attempt is made to execute a COPz rs instruction with an undefined minor opcode (bit25~21)
- an attempt is made to execute a COPz rt instruction with an undefined minor opcode (bit20~16)

64-bit operations are always valid in Kernel mode regardless of the value of the KX bit in Status register. This exception is not maskable.

11.16.2 Processing

The common exception vector is used for this exception, and the RI code in Cause register is set.

If EXL bit of Status register is only set to 0, the following operation is executed. EPC register points to the address of the instruction causing the exception. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in EPC register and the BD bit of Cause register is set to 1.

11.16.3 Servicing

No instruction in the MIPS ISA are currently interpreted. The process executing at the time of this exception is handed an illegal instruction/reserved operand fault signal. This error is usually fatal.

11.17 Coprocessor Unusable Exception

11.17.1 Cause

The Coprocessor Unusable exception occurs when an attempt is made to execute a coprocessor instruction for either.

- attempting to execute a coprocessor CPz instruction when its corresponding CUz bit in Status register.
- in User or Supervisor mode attempting to execute a CP0 instruction when CU0 bit is cleared to “0”. (In Kernel mode, an exception is not raised when a CP0 instruction is issued , regardless of the CU0 bit setting)
- an attempt is made to execute a FPU instruction in TX49 without FPU

11.17.2 Processing

The common exception vector is used for this exception, and the CpU code in Cause register is set. The coprocessor number referred to at the time of the exception is stored in Cause register CE (Coprocessor Error) field.

If EXL bit of Status register is only set to 0, the following operation is executed. EPC register points to the address of the instruction causing the exception. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in EPC register and BD bit of Cause register is set to 1.

11.17.3 Servicing

The coprocessor unit to which an attempted reference was mode is identified by the Coprocessor Usage Error field, which results in one of the following situations:

- If the process is entitled access to the coprocessor, the coprocessor is marked usable and the corresponding user state is restored to the coprocessor.
- If the process is entitled access to the coprocessor, but the coprocessor does not exist or has failed, interpretation of the coprocessor instruction is possible.
- If the BD bit is set in the Cause register, the branch instruction must be interpreted; then the coprocessor instruction can be emulated and execution resumed with the EPC register advanced past the coprocessor instruction.
- If the process is not entitled access to the coprocessor, the process executing at the time is handed an illegal instruction/privileged instruction fault signal. This error is usually fatal.

11.18 Floating-Point Exception

11.18.1 Cause

The Floating-Point exception is used by the floating-point coprocessor. This exception is not maskable.

11.18.2 Processing

The common exception vector is used for this exception, and the FPE code in Cause register is set. The contents of the Floating-Point Control/Status register indicate the cause of this exception.

If EXL bit of Status register is only set to 0, the following operation is executed. EPC register points to the address of the instruction causing the exception. If, however, the affected instruction was in the branch delay slot (for execution during a branch), the immediately preceding branch instruction address is retained in EPC register and the BD bit of Cause register is set to 1.

11.18.3 Servicing

This exception is cleared by clearing the appropriate bit in the Floating-Point Control/Status register.

For an unimplemented instruction exception, the kernel should emulate the instruction; for other exceptions, the kernel should pass the exception to the user program that caused the exception.

11.19 Interrupt Exception

11.19.1 Cause

The Interrupt exception is raised by any of eight interrupts (two software and six hardware). A hardware interrupt is raised when GINT* signal goes active. A software interrupt is raised by setting the IP[1]/IP[0] bit in Cause register. The significance of these interrupts is dependent upon the specific system implementation.

Each of the eight interrupts can be masked individually by clearing its corresponding bit in the IM(Interrupt Mask) field of Status register, and all interrupts can be masked at once by clearing IE bit of Status register to "0".

If the GTINTDIS is low when a Reset exception occurred, GINT[5]* is disabled and the timer exception is enabled.

11.19.2 Processing

The common exception vector is used as following;

- In 32 bit mode, 0x8000 0180 (BEV = 0)
  0xbfc0 0380 (BEV = 1)
- In 64 bit mode, 0xffff ffff 8000 0180 (BEV = 0)
  0xffff ffff bfc0 0380 (BEV = 1)

11.19.3 Servicing

If the interrupt is caused by one of the two software-generated exceptions (SW1 or SW0), the interrupt condition is cleared by setting the corresponding Cause register bit to 0.

If the interrupt is hardware-generated, the interrupt condition is cleared by correcting the condition causing the interrupt pin to be asserted.

If the timer interrupt is caused, the interrupt condition is cleared by changing the value of the Compare register or setting the corresponding Cause register bit (IP[7]) to 0.

Interrupts are not acceptable when the settings of the Status register are EXL = 1 and ERL = 1.

Note: due to the write buffer, a store to an external device will not necessary occur until after other instructions in the pipeline finish. Thus, the user must ensure that the store will occur before the return from exception instruction (ERET) is executed otherwise the interrupt may be serviced again even though there should be no interrupt pending.

11.20 Exception Handling and Servicing Flowcharts

The remainder of this chapter contains flowcharts for the following exceptions and guidelines for their handlers:

- general exceptions and their exception handler
- TLB/XTLB miss exception and their exception handler
- Cold Reset, Soft Reset and NMI exceptions, and a guideline to their handler.

Generally speaking, the exceptions are handled by hardware (HW); the exceptions are then serviced by software (SW).

Exceptions other than Reset, Soft Reset, NMI or first-level miss

Note: Interrupts can be masked by IE or IMs

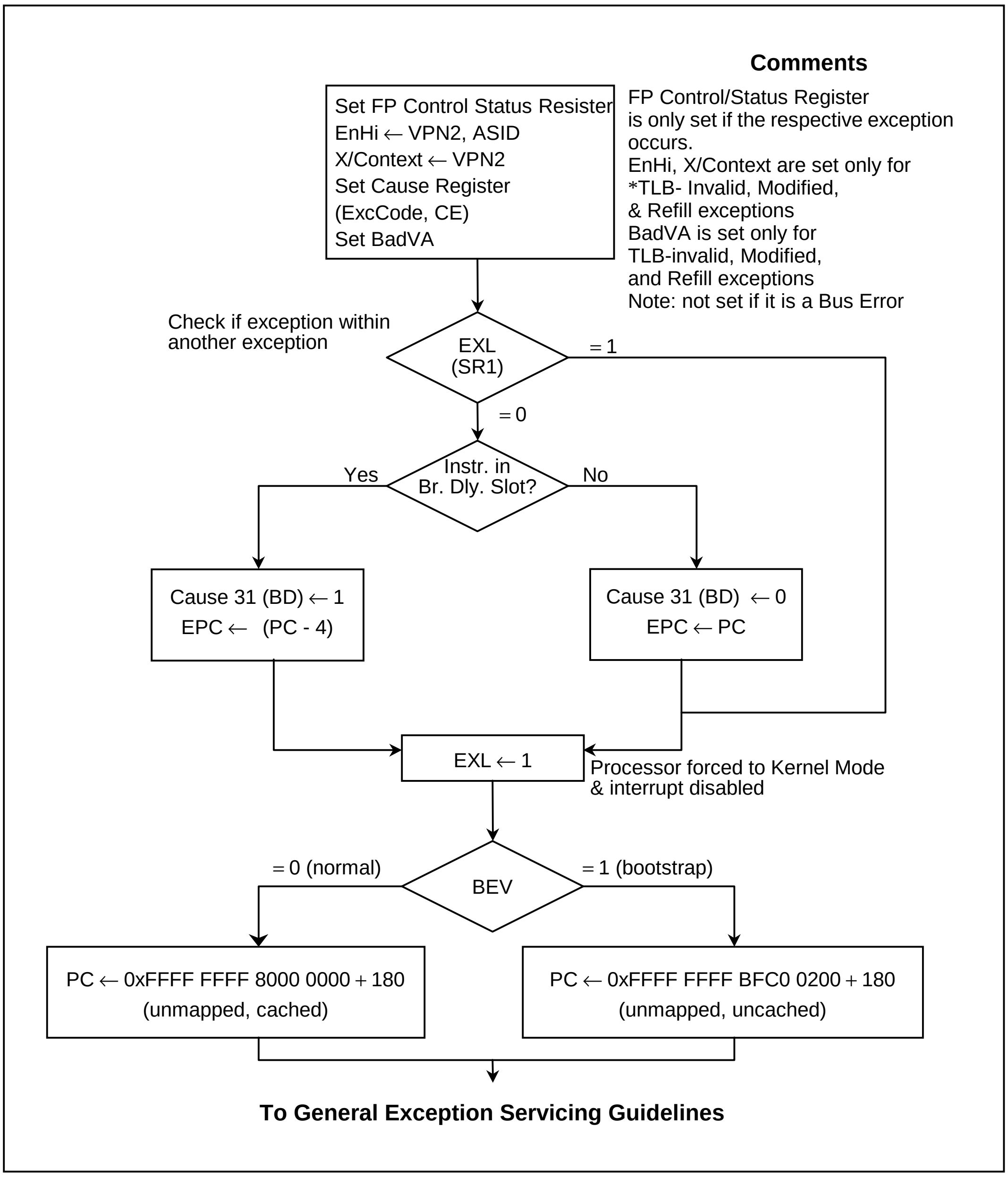


Figure 11-1 General Exception Handler (HW)

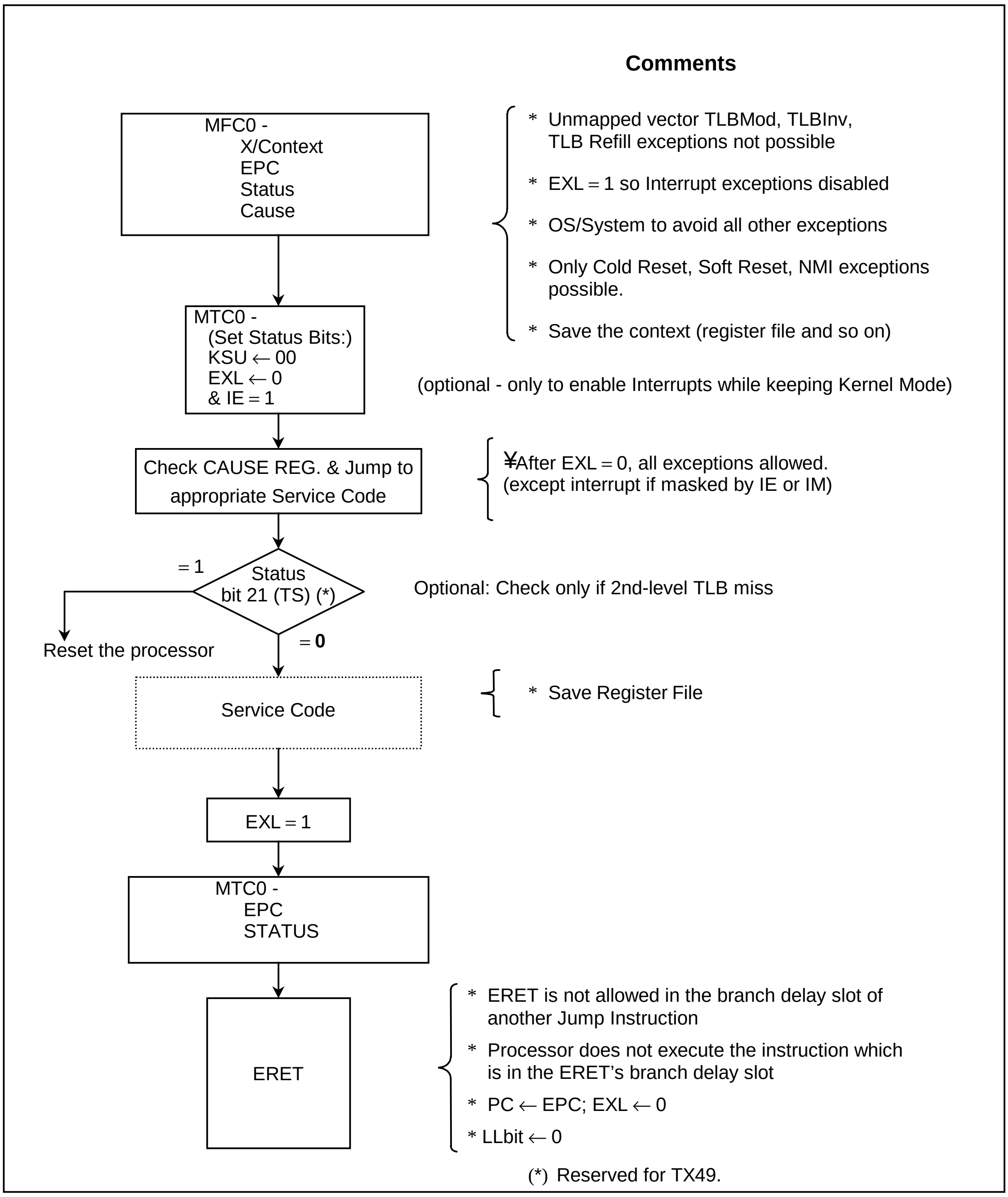

Comments
MFC0 -
X/Context
EPC
Status
Cause
* Unmapped vector TLBMod, TLBInv, TLB Refill exceptions not possible
* EXL = 1 so Interrupt exceptions disabled
* OS/System to avoid all other exceptions
* Only Cold Reset, Soft Reset, NMI exceptions possible.
* Save the context (register file and so on)
MTC0 -
(Set Status Bits:)
KSU ← 00
EXL ← 0
& IE = 1
(optional - only to enable Interrupts while keeping Kernel Mode)
Check CAUSE REG. & Jump to appropriate Service Code
¥After EXL = 0, all exceptions allowed.
(except interrupt if masked by IE or IM)
= 1
Status bit 21 (TS) (*)
Optional: Check only if 2nd-level TLB miss
Reset the processor
= 0
Service Code
* Save Register File
EXL = 1
MTC0 -
EPC
STATUS
ERET
* ERET is not allowed in the branch delay slot of another Jump Instruction
* Processor does not execute the instruction which is in the ERET's branch delay slot
* PC ← EPC; EXL ← 0
* LLbit ← 0
(*) Reserved for TX49.

Figure 11-2 General Exception Servicing Guidelines (SW)

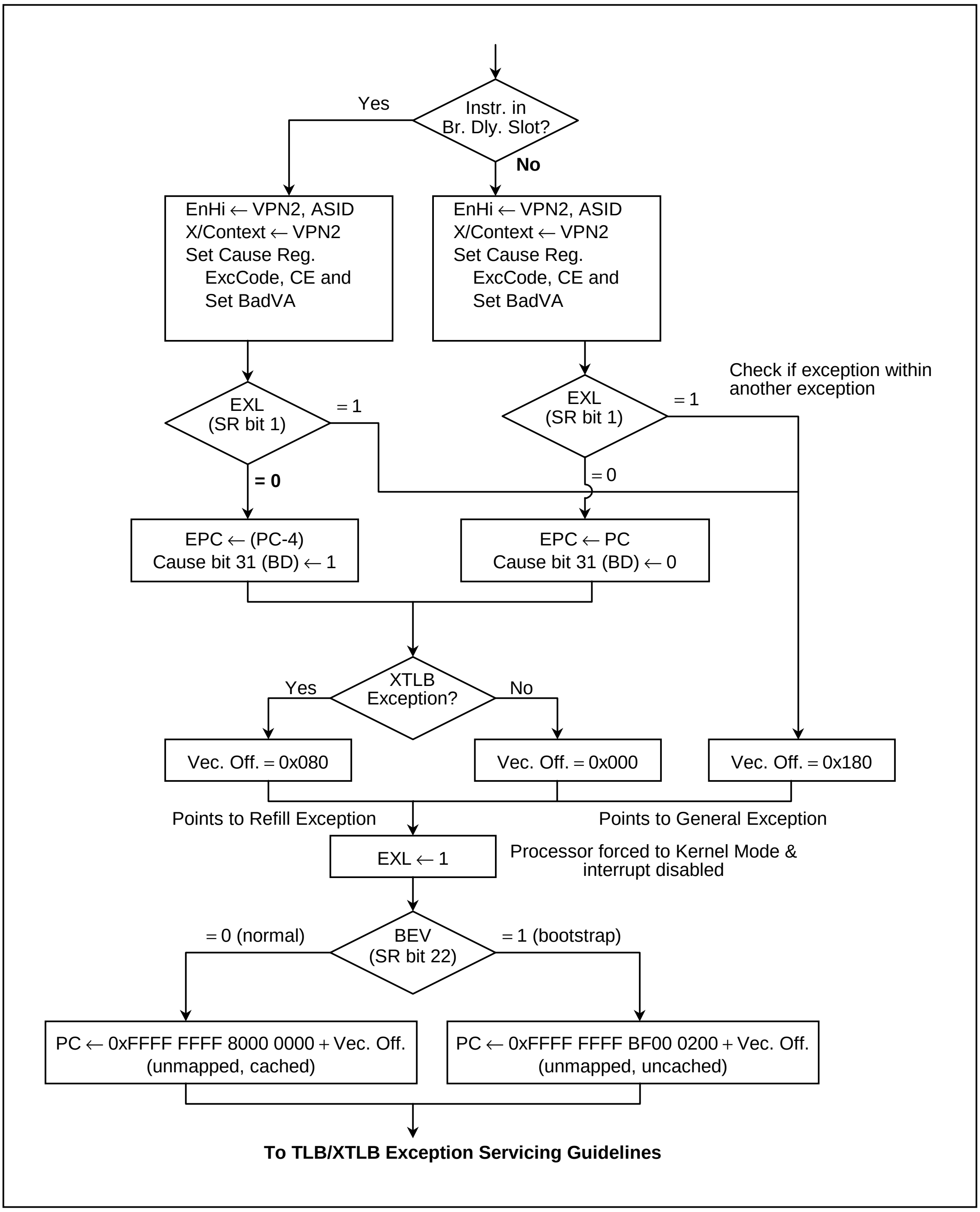
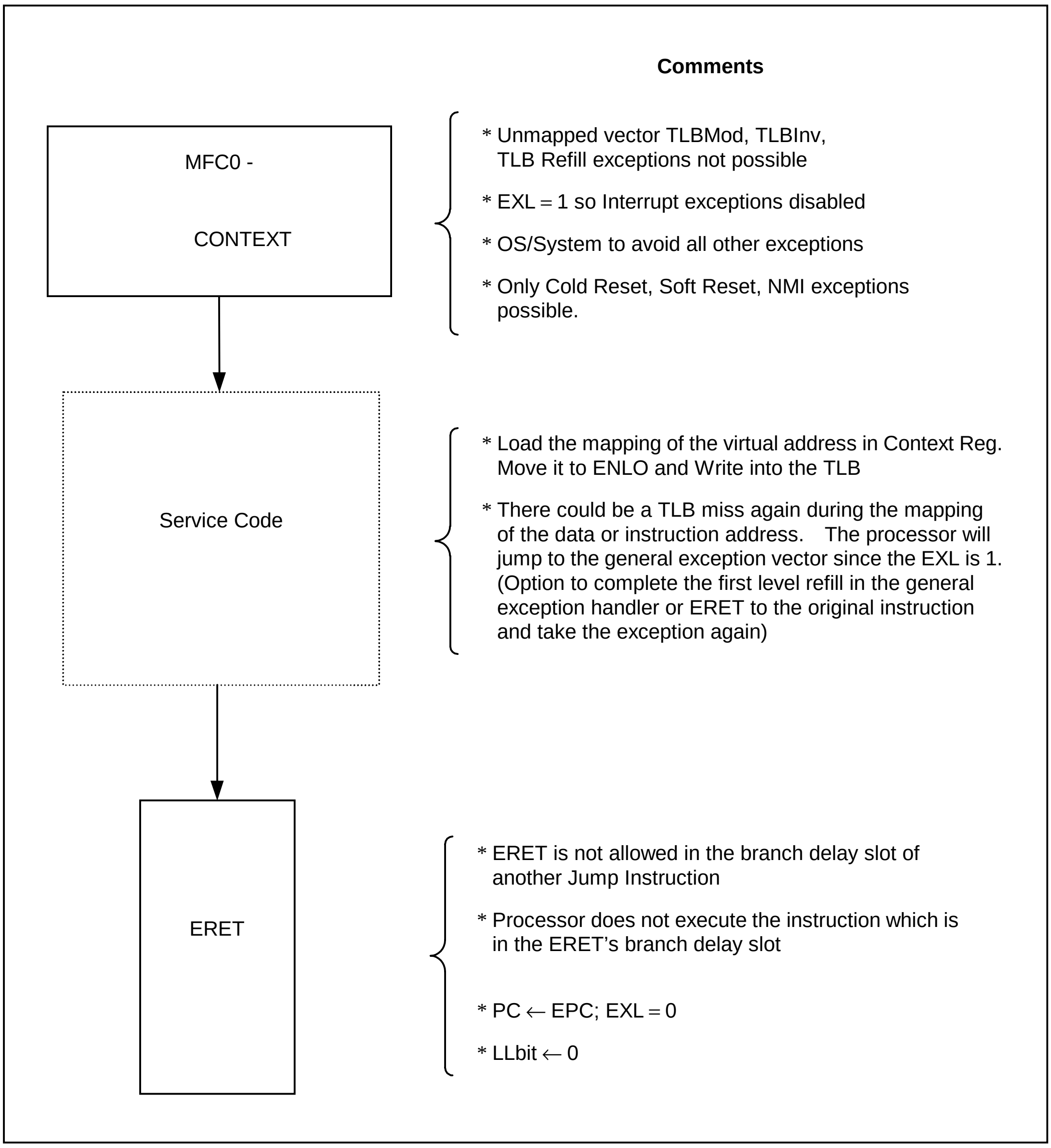


Figure 11-3 TLB/XTLB Miss Exception Handler (HW)



Comments

- * Unmapped vector TLBMod, TLBInv, TLB Refill exceptions not possible
- * EXL = 1 so Interrupt exceptions disabled
- * OS/System to avoid all other exceptions
- * Only Cold Reset, Soft Reset, NMI exceptions possible.

- * Load the mapping of the virtual address in Context Reg. Move it to ENLO and Write into the TLB
- * There could be a TLB miss again during the mapping of the data or instruction address. The processor will jump to the general exception vector since the EXL is 1. (Option to complete the first level refill in the general exception handler or ERET to the original instruction and take the exception again)

- * ERET is not allowed in the branch delay slot of another Jump Instruction
- * Processor does not execute the instruction which is in the ERET’s branch delay slot
- * PC ← EPC; EXL = 0
- * LLbit ← 0

Figure 11-4 TLB/XTLB Exception Servicing Guidelines (SW)

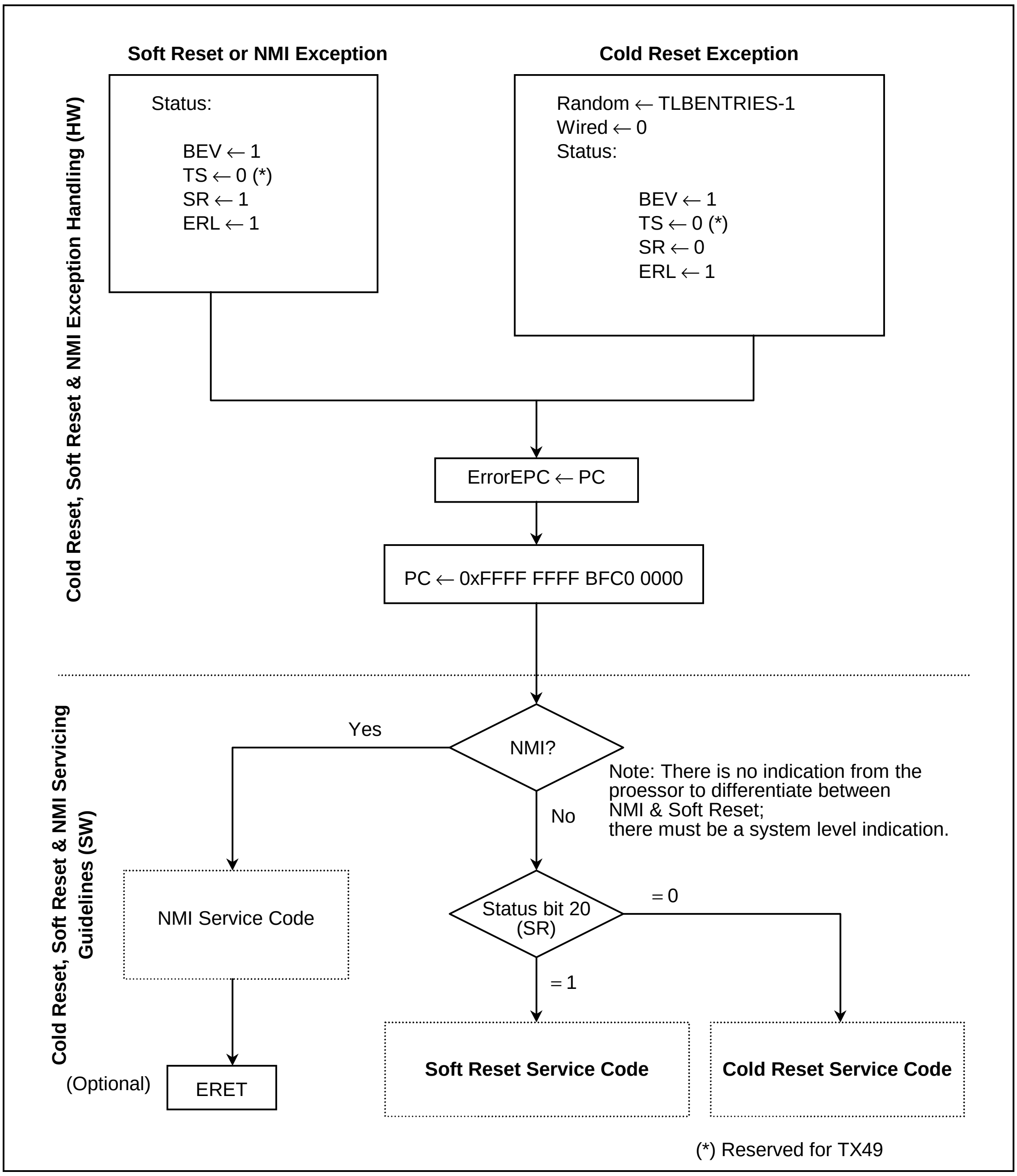

Cold Reset, Soft Reset & NMI Exception Handling (HW)
Soft Reset or NMI Exception
Status:
BEV ← 1
TS ← 0 (*)
SR ← 1
ERL ← 1
Cold Reset Exception
Random ← TLBENTRIES-1
Wired ← 0
Status:
BEV ← 1
TS ← 0 (*)
SR ← 0
ERL ← 1
ErrorEPC ← PC
PC ← 0xFFFF FFFF BFC0 0000
Cold Reset, Soft Reset & NMI Servicing Guidelines (SW)
NMI?
Yes
No
Note: There is no indication from the proessor to differentiate between NMI & Soft Reset; there must be a system level indication.
NMI Service Code
(Optional)
ERET
Status bit 20 (SR)
= 0
= 1
Soft Reset Service Code
Cold Reset Service Code
(*) Reserved for TX49

Figure 11-5 Cold Reset, Soft Reset & NMI Exception Handling (HW) and Servicing Guidelines (SW)

12. Floating-Point Unit, CP1

This chapter describes the floating-point operations, including the programming model, instruction set and formats.

The floating-point operations fully conform to the requirements of ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic.*

12.1 Overview

All floating-point instructions, as defined in the MIPS ISA for the floating-point coprocessor, CP1, are processed by the other hardware unit that executes integer instructions.

The execution of floating-point instructions can be disabled by the coprocessor usability *CU* bit defined in the CP0 *Status* register.

12.2 Floating Point Register

12.2.1 Floating-Point General Registers (FGRs)

CP1 has a set of *Floating-Point General Purpose registers (FGRs)* that can be accessed in the following ways:

- As 32 general purpose registers (32 FGRs), each of which is 32 bits wide when the *FR* bit in the *CPU* Status register equals 0; or as 32 general purpose registers (32 FGRs), each of which is 64-bits wide when FR equals 1. The CPU accesses these registers through MOVE, LOAD, and STORE instructions.
- As 16 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the FR bit in the CPU *Status* register equals 0. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to adjacently numbered FGRs as shown in Figure 12-1.
- As 32 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the *FR* bit in the CPU *Status* register equals 1. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to an FGR as shown in Figure 12-1.

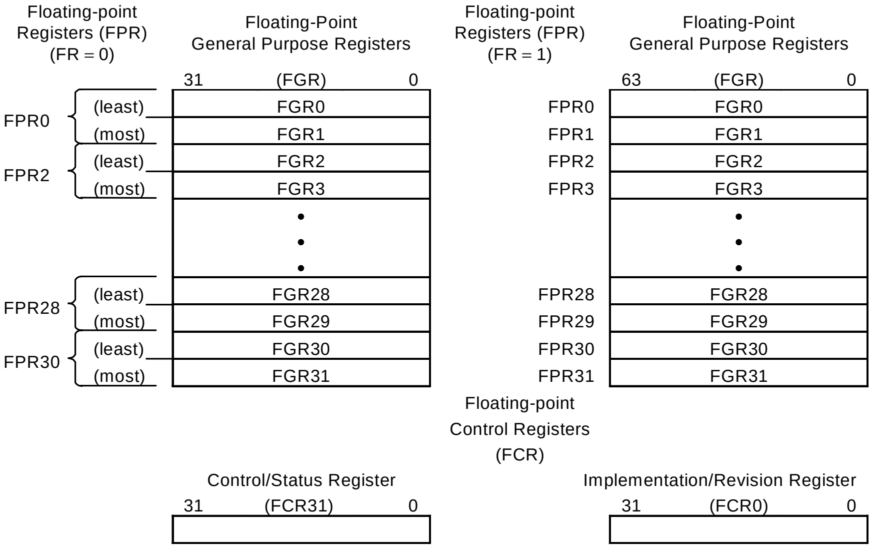


Figure 12-1 FP Registers

12.2.2 Floating-Point Control Registers

The MIPS RISC architecture defines 32 floating-point control registers (*FCRs*); the TX49 processor implements two of these registers: *FCR0* and *FCR31*. These *FCRs* are described below:

- The *Implementation/Revision* register *(FCR0)* holds revision information.
- The *Control/Status* register *(FCR31)* controls and monitors exceptions, holds the result of compare operations, and establishes rounding modes.
- *FCR1* to *FCR30* are reserved.

Table 12-1 lists the assignments of the FCRs.

Table 12-1 Floating-Point Control Register Assignments

FCR Number	Use
FCR0	Coprocessor implementation and revision register
FCR1 to FCR30	Reserved
FCR31	Rounding mode, cause, trap enables, and flags

Implementation and Revision Register, (FCR0)

The read-only *Implementation and Revision* register *(FCR0)* specifies the implementation and revision number of CP1. This information can determine the coprocessor revision and performance level, and can also be used by diagnostic software.

Figure 12-2 shows the layout of the register; Table 12-2 describes the *Implementation and Revision* register *(FCR0)* fields.

Implementation/Revision Register (FCR0)

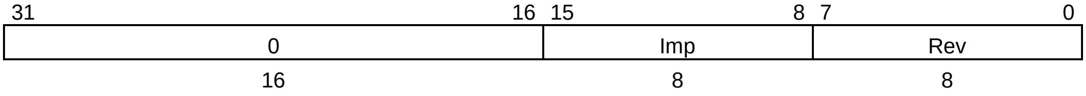


Figure 12-2 Implementation/Revision Register

Table 12-2 FCR0 Fields

Field	Description
Imp	Implementation number
Rev	Revision number in the form of y. x
0	Reserved. Returns zeroes when read.

The revision number is a value of the form *y. x*, where:

- *y* is a major revision number held in bits 7:4.
- *x* is a minor revision number held in bits 3:0.

Control/Status Register (FCR31)

The *Control/Status* register *(FCR31)* contains control and status information that can be accessed by instructions in either Kernel or User mode. *FCR31* also controls the arithmetic rounding mode and enables User mode traps, as well as identifying any exceptions that may have occurred in the most recently executed floating-point instruction, along with any exceptions that may have occurred without being trapped.

Figure 12-3 shows the format of the *Control/Status* register, and Table 12-3 describes the *Control/Status* register fields. Figure 12-4 shows the *Control/Status* register *Cause, Flag,* and *Enable* fields.

Control/Status Register (FCR31)

31 25	24	23	22 18	17 12	11 7	6 2	1 0
0	FS	C	0	Cause EVZOUI	Enables VZOUI	Flags VZOUI	RM
7	1	1	5	6	5	5	2

Figure 12-3 FP Control/Status Register Bit Assignments

Table 12-3 Control/Status Register Fields

Field	Description
FS	When set, denormalized results can be flushed instead of causing an unimplemented operation exception.
C	Condition bit. Stores the result of compare instruction. See description of *Control/Status* register *Condition* bit.
Cause	Cause bits. These bits identify the exceptions raised by the most recently executed floating-point instruction. See Figure 12-4 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Enables	Enable bits. When set, these bits trap any floating-point exceptions to indicate that they have been passed to the CPU. See Figure 12-4 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Flags	Flag bits. These bits indicate that an exception was raised. See Figure 12-4 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
RM	Rounding mode bits. See Table 12-5 and the description of *Control/Status* register *Rounding Mode Control* bits.

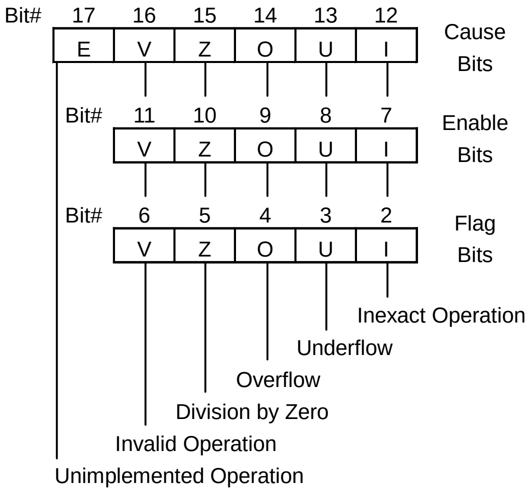


Figure 12-4 Control/Status Register Cause, Flag, and Enable Fields

Control/Status Register FS Bit

The *FS* bit enables the flushing of denormalized values. When the *FS* bit is set and the Underflow and Inexact *Enable* bits are not set, denormalized results are flushed instead of causing an Unimplemented Operation exception. Results are flushed either to 0 or the minimum normalized value, depending upon the rounding mode (see Table 12-4 below), and the Underflow and Inexact *Flag* and *Cause* bits are set.

Table 12-4 Flush Values of Denormalized Results

Denormalized Result	Flushed Result Rounding Mode			
	RN	RZ	RP	RM
Positive	+0	+0	$+2^{Emin}$	+0
Negative	-0	-0	-0	-2^{Emin}

Control/Status Register Condition Bit

When a floating-point Compare operation takes place, the result is stored at bit 23, the *Condition* bit. The *C* bit is set to 1 if the condition is true; the bit is cleared to 0 if the condition is false. Bit 23 is affected only by compare and CTC1 instructions.

The BC1T and BC1F instructions test the C bit to decide whether or not to cause a branch.

Control/Status Register Cause, Flag, and Enable Fields

Figure 12-4 illustrates the *Cause, Flag,* and Enable fields of the *Control/Status* register. The *Cause* and *Flag* fields are updated by all conversion, computational (except MOV. fmt), CTC1, reserved, and unimplemented instructions. All other instructions have no affect on these fields.

Cause Bits

Bits 17:12 in the *Control/Status* register contain *Cause* bits, as shown in Figure 12-4, which reflect the results of the most recently executed floating-point instruction. The *Cause* bits are a logical extension of the CP0 *Cause* register; they identify the exceptions raised by the last floating-point operation. If the corresponding *Enable* bit is set at the time of the exception a floating-point exception and interrupt is raised. If more than one exception occurs on a single instruction, each appropriate bit is set.

The *Cause* bits are updated by most floating-point operations. The Unimplemented Operation *(E)* bit is set to 1 if software emulation is required, otherwise it remains 0. The other bits are set to 0 or 1 to indicate the occurrence or non-occurrence (respectively) of an IEEE 754 exception. Within the set of floating-point instructions that update the *Cause* bits, the *Cause* field indicates the exceptions raised by the most-recently-executed instruction.

When a floating-point exception is taken, no results are stored, and the only state affected is the *Cause* bit. Therefore, software emulation routines can use the original values to emulate the exception-causing floating-point operation.

Enable Bits

A floating-point exception is generated any time a *Cause* bit and the corresponding *Enable* bit are set. A floating-point operation that sets an enabled *Cause* bit forces an immediate floating-point exception, as does setting both *Cause* and *Enable* bits with CTC1. Software can also emulate above.

There is no enable for Unimplemented Operation *(E)*. An Unimplemented exception always generates a floating-point exception.

Before returning from a floating-point exception, software must first clear the enabled *Cause* bits with a CTC1 instruction to prevent a repeat of the interrupt. Thus, User mode programs can never observe enabled *Cause* bits set; if this information is required in a User mode handler, it must be passed somewhere other than the *Status* register.

For a floating-point operation that sets only unenabled *Cause* bits, no floating-point exception occurs and the default result defined by IEEE 754 is stored. In this case, the exceptions that were caused by the immediately previous floating-point operation can be determined by reading the *Cause* field.

Flag Bits

The *Flag* bits are cumulative and indicate the exceptions that were raised by the operations that were executed since the bits were explicitly reset. *Flag* bits are set to 1 if an IEEE 754 exception is raised, otherwise they remain unchanged. The *Flag* bits are never cleared as a side effect of floating-point operations; however, they can be set or cleared by writing a new value into the *Status* register, using a CTC1 instruction.

Control/Status Register Rounding Mode Control Bits

Bits 1 and 0 in the *Control/Status* register constitute the *Rounding Mode (RM)* field.

As shown in Table 12-5, these bits specify the rounding mode that CP1 uses for all floating-point operations.

Table 12-5 Rounding Mode Bit Decoding

Rounding ModeRM (1:0)	Mnemonic	Description
0	RN	Round result to nearest representable value; round to value with least-significant bit 0 when the two nearest representable values are equally near.
1	RZ	Round toward 0: round to value closest to and not greater in magnitude than the infinitely precise result.
2	RP	Round toward +∞: round to value closest to and not less than the infinitely precise result.
3	RM	Round toward –∞: round to value closest to and not greater than the infinitely precise result.

12.2.3 Accessing the FP Control and Implementation/Revision Registers

The *Control/Status* and the *Implementation/Revision* registers are read by a Move Control From Coprocessor 1 (CFC1) instruction.

The bits in the *Control/Status* register can be set or cleared by writing to the register using a Move Control To Coprocessor 1 (CTC1) instruction. The *Implementation/Revision* register is a read-only register. There are no pipeline hazards (between any instructions) associated with floating-point control registers.

12.3 Floating-Point Formats

CP1 performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit single-precision format has a 24-bit signed-magnitude fraction field *(f+s)* and an 8-bit exponent *(e)*, as shown in Figure 12-5.

31	30 — 23	22 — 0
s Sign	e Exponent	f Fraction
1	8	23

Figure 12-5 Single-Precision Floating-Point Format

The 64-bit double-precision format has a 53-bit signed-magnitude fraction field *(f+s)* and an 11-bit exponent, as shown in Figure 12-6.

63	62 — 52	51 — 0
s Sign	e Exponent	f Fraction
1	11	52

Figure 12-6 Double-Precision Floating-Point Format

As shown in the above figures, numbers in floating-point format are composed of three fields:

- sign field, s
- biased exponent, $e = E + bias$
- fraction, $f = b_1 b_2 \ldots . b_{p-1}$

The range of the unbiased exponent E includes every integer between the two values E_{min} and E_{max} inclusive, together with two other reserved values:

- $E_{min} - 1$ (to encode 0 and denormalized numbers)
- $E_{max} + 1$ (to encode ∞ and NaNs [Not a Number])

For single-and double-precision formats, each representable nonzero numerical value has just one encoding. For single-and double-precision formats, the value of a number, *v*, is determined by the equations shown in Table 12-6.

Table 12-6 Equations for Calculating Values in Single and Double-Precision Floating-Point Format

No.	Equation
(1)	if $E = E_{max}+1$ and $f \neq 0$, then *v* is NaN, regardless of s
(2)	if $E = E_{max}+1$ and $f = 0$, then $v = (-1)s\infty$
(3)	if $E_{min} \leq E \leq Emax$, then $v = (-1)^s 2^E (1.f)$
(4)	if $E = E_{min}-1$ and $f \neq 0$, then $v = (-1)^s 2^{Emin} (0.f)$
(5)	if $E = E_{min}-1$ and $f = 0$, then $v = (-1)^s 0$

For all floating-point formats, if *v* is NaN, the most-significant bit of *f* determines whether the value is a signaling or quiet NaN: *v* is a signaling NaN if the most-significant bit of *f* is set, otherwise, *v* is a quiet NaN.

Table 12-7 defines the values for the format parameters; minimum and maximum floating-point values are given in Table 12-8.

Table 12-7 Floating-Point Format Parameter Values

Parameter	Format	
	Single	Double
E$_{max}$	+127	+1023
E$_{min}$	–126	–1022
Exponent *bias*	+127	+1023
Exponent width in bits	8	11
Integer bit	hidden	hidden
Fraction width in bits	23†	52†
Format width in bits	32	64

† Excluding the sign bit.

Table 12-8 Minimum and Maximum Floating-Point Values

Type	Value
Single-precision Minimum	1.40129846e^{-45}
Single-precision Minimum Norm	1.17549435e^{-38}
Single-precision Maximum	3.40282347e^{+38}
Double-precision Minimum	4.9406564584124654e^{-324}
Double-precision Minimum Norm	2.2250738585072014e^{-308}
Double-precision Maximum	1.7976931348623157e^{+308}

12.4 Binary Fixed-Point Format

Binary fixed-point values are held in 2's complement format. Unsigned fixed-point values are not directly provided by the floating-point instruction set. Figure 12-7 illustrates binary single fixed-point format and Figure 12-8 illustrates binary long fixed-point format; Table 12-9 lists the binary fixed-point format fields.

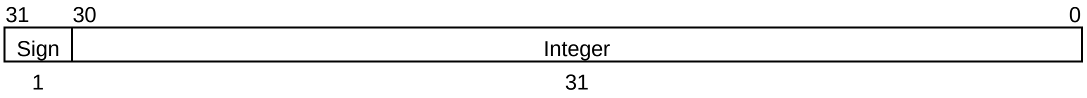


Figure 12-7 Binary Single Fixed-Point Format

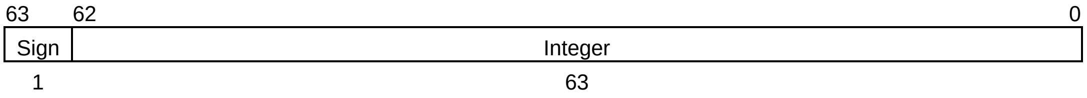


Figure 12-8 Binary Long Fixed-Point Format

Field assignments of the binary fixed-point format are:

Table 12-9 Binary Fixed-Point Format Fields

Field	Description
sign	sign bit
integer	integer value (2’s complement)

12.5 Floating-Point Instruction Set Summary

Each instruction is 32 bits long, and aligned on a word boundary. This section describes the overview of instructions for floating-point unit. A detailed description of each instruction is provided in Appendix B.

12.5.1 Load, Move and Store Instructions (Table 12-10)

Load and Store instructions move data between memory and FPU general purpose registers, and Move instructions move data directly between CPU and FPU general purpose registers. These instructions are not perform format conversions and therefore never cause floating-point exceptions. The instruction immediately following a load can use the contents of the loaded register. However, in such case the hardware interlocks, requiring additional real cycles. Thus, the scheduling of load delay slots is required to avoid the interlocking.

Data Alignment

All processor loads and stores reference the following aligned data items:

- For word loads and stores, the access type is always WORD, and the low-order 2 bits of the address must always be 0.
- For doubleword loads and stores, the access type is always DOUBLEWORD, and the low-order 3 bits of the address must always be 0.

Endian

Regardless of byte-numbering order (endianness) of the data, the address specifies the byte that has the smallest byte address in the addressed field. For a big-endian system, it is the leftmost byte; for a little-endian system, it is the rightmost byte.

Table 12-10 FPU Instruction Set (Optional): Load, Move and Store Instruction

Instruction	Description	Note
LWC1	Load Word to FPU (coprocessor 1)	MIPS I
SWC1	Store Word from FPU (coprocessor 1)	MIPS I
MTC1	Move Word to FPU (coprocessor 1)	MIPS I
CTC1	Move Control Word to FPU (coprocessor 1)	MIPS I
MFC1	Move Word from FPU (coprocessor 1)	MIPS I
CFC1	Move Control Word from FPU (coprocessor 1)	MIPS I

12.5.2 Conversion Instructions (Table 12-11)

Conversion instructions perform conversion operations between the various data formats such as single- or double-precision, fixed- or floating-point formats. Table 12-11 list conversion instructions.

Table 12-11 FPU Instruction Set(Optional): Conversion Instruction

Instruction	Description	Note
CVT.S.fmt	Floating-Point Convert to Single FP Format	MIPS I
CVT.W.fmt	Floating-Point Convert to Single Fixed-Point Format	MIPS I
ROUND.W.fmt	Floating-point Round	MIPS II
TRUNC.W.fmt	Floating-point Truncate	MIPS II
CEIL.W.fmt	Floating-point Ceiling	MIPS II
FLOOR.W.fmt	Floating-point Floor	MIPS II

12.5.3 Computational Instructions (Table 12-12)

Computational instructions perform arithmetic operations on floating-point values in the FPU registers. These are two categories of computational instructions:

- 3-Operand Register-Type instructions, which perform floating-point addition, multiplication, division, and square root operations
- 2-Operand Register-Type instructions, which perform floating-point absolute value, move, negate, and square root operation.

Table 12-12 FPU Instruction Set(Optional): Computational Instruction

Instruction	Description	Note
ADD.fmt	Floating-point Add	MIPS I
SUB.fmt	Floating-point Subtract	MIPS I
MUL.fmt	Floating-point Multiply	MIPS I
DIV.fmt	Floating-point Divide	MIPS I
ABS.fmt	Floating-point Absolute Value	MIPS I
MOV.fmt	Floating-point Move	MIPS I
NEG.fmt	Floating-point Negate	MIPS I
SQRT.fmt	Floating-point Square root	MIPS II

12.5.4 Compare and Branch Instructions (Table 12-13)

Compare instructions perform comparisons of the contents of registers and set a conditional bit based on the results. Branch on FPU Condition instructions perform a branch to the specified target if the specified coprocessor condition is met.

Table 12-13 FPU Instruction Set(Optional): Compare and Branch Instruction

Instruction	Description	Note
C.cond.fmt	Floating-point Compare	MIPS I
BC1T	Branch on FPU True	MIPS I
BC1F	Branch on FPU False	MIPS I
BC1TL	Branch on FPU True Likely	MIPS II
BC1FL	Branch on FPU False Likely	MIPS II

The floating-point compare (C.fmt.cond) instructions interpret the contents of two FPU registers (fs, ft) in the specified format (fmt) and arithmetically compare them. A result is determined based on the comparison and conditions (cond) specified in the instruction.

Table 12-4 lists the mnemonics for the compare instruction conditions.

Table 12-14 Mnemonics and Definitions of Compare Instruction Conditions

Mnemonic	Definition	Mnemonic	Definition
F	False	T	True
UN	Unordered	OR	Ordered
EQ	Equal	NEQ	Not Equal
UEQ	Unordered or Equal	OLG	Ordered or Less than or Greater than
OLT	Ordered Less Than	UGE	Unordered or Greater than or Equal
ULT	Unordered or Less Than	OGE	Ordered Greater than or Equal
OLE	Ordered Less Than or Equal	UGT	Unordered or Greater Than
ULE	Unordered or Less than or Equal	OGT	Ordered Greater Than
SF	Signaling False	ST	Signaling True
NGLE	Not Greater than or Less than or Equal	GLE	Greater than, or Less than or Equal
SEQ	Signaling Equal	SNE	Signaling Not Equal
NGL	Not Greater than or Less than	GL	Greater than or Less Than
LT	Less Than	NLT	Not Less Than
NGE	Not Greater than or Equal	GE	Greater than or Equal
LE	Less than or Equal	NLE	Not Less than or Equal
NGT	Not Greater Than	GT	Greater Than

13. Floating-Point Exception

13.1 Introduction

This chapter describes floating-point exceptions, including FPU exception type, exception trap processing, exception flags, saving and restoring state when handling an exception, and trap handlers for IEEE Standard 754 exceptions.

13.2 Exception Types

The FP Control/Status register described in Chapter 12 contains an Enable bit for each exception type; exception Enable bits determine whether an exception will cause the FPU to initiate a trap or set a status flag.

- If a trap is taken, the FPU remains in the state found at the beginning of the operation and a software exception handling routine executes.
- If no trap is taken, an appropriate value is written into the FPU destination register and execution continues.

The FPU supports the five IEEE Standard 754 exceptions:

- Inexact (I)
- Underflow (U)
- Overflow (O)
- Division by Zero (Z)
- Invalid Operation (V)

Cause bits, Enables, and Flag bits (status flags) are used.

The FPU adds a sixth exception type, Unimplemented Operation (E). This exception indicates the use of a software implementation. The Unimplemented Operation exception has no Enable or Flag bit; whenever this exception occurs, an unimplemented exception trap is taken.

Figure 13-1 shows the Control/Status register bits that support exceptions.

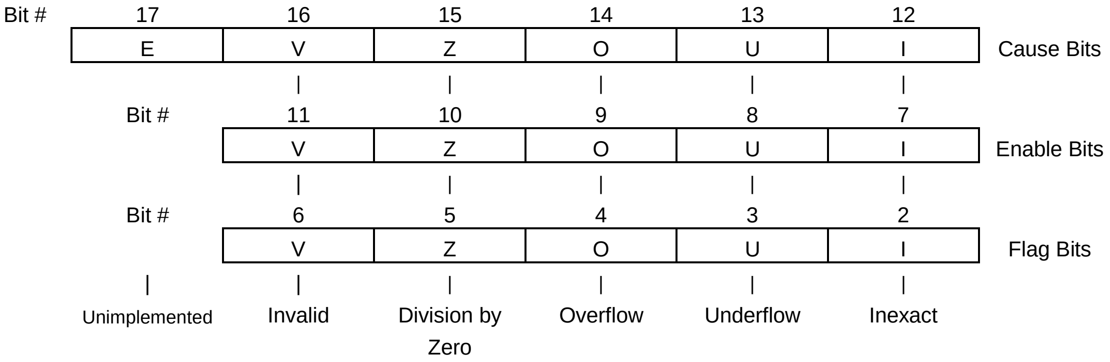


Figure 13-1 Control/Status Register Exception/Flag/Trap/Enable Bits

13.3 Exception Trap Processing

When a floating-point exception trap is taken, the Cause register indicates the floating-point coprocessor is the cause of the exception trap.

The Floating-Point Exception (FPE) code is used, and the Cause bits of the floating-point Control/Status register indicate the reason for the floating-point exception. These bits are, in effect, an extension of the system coprocessor Cause register.

13.4 Flags

A Flag bit is provided for each IEEE exception. This Flag bit is set to a 1 on the assertion of its corresponding exception, with no corresponding exception trap signaled.

When no exception trap is signaled, floating-point coprocessor takes a default action, providing a substitute value for the exception-causing result of the floating-point operation. The particular default action taken depends upon the type of exception. Table 13-1 lists the default action taken by the FPU for each of the IEEE exceptions.

Table 13-1 Default FPU Exception Actions

Field	Description	Rounding Mode	Default Action
I	Inexact exception	ANY	Supply a rounded result.
U	Underflow exception	ANY	Supply a rounded result.
O	Overflow exception	RN	Modify overflow values to ∞ with the sign of the intermediate result.
		RZ	Modify overflow values to the format’s largest finite number with the sign of the intermediate result.
		RP	Modify negative overflows to the format’s most negative finite number; modify positive overflows to + ∞
		RM	Modify positive overflows to the format’s largest finite number; modify negative overflows to – ∞
Z	Division by zero	ANY	Supply a properly signed ∞
V	Invalid operation	ANY	Supply a quiet Not a Number (NaN).

The FPU detects the eight exception causes internally. When the FPU encounters one of these unusual situations, it causes either an IEEE exception or an Unimplemented Operation exception (E).

Table 13-2 lists the exception-causing situations and contrasts the behavior of the FPU with the requirements of the IEEE Standard 754.

Table 13-2 FPU Exception-Causing Conditions

FPA Internal Result	IEEE Standard 754	Trap Enable	Trap Disable	Notes
Inexact result	I	I	I	Loss of accuracy
Exponent overflow	O, I*	O, I	O, I	Normalized exponent > Emax
Division by zero	Z	Z	Z	Zero is (exponent = Emin – 1, mantissa = 0)
Overflow on convert	V	V	E	Source out of integer range
Signaling NaN source	V	V	E	Quiet NaN result generated from quiet NaN source
Invalid operation	V	V	E	0/0, etc.
Exponent underflow	U	E	E	Normalized exponent < Emin
Denormalized or QNaN	None	E	E	Denormalized is (exponent = Emin – 1 and mantissa < > 0)

* The IEEE Standard 754 specifies an inexact exception on overflow only if the overflow trap is disabled.

13.5 FPU Exceptions

The following sections describe the conditions that cause the FPU to generate each of its exceptions, and details the FPU response to each exception-causing condition.

Inexact Exception (I)

The FPU generates the Inexact exception if one of the following occurs:

- the rounded result of an operation is not exact, or
- the rounded result of an operation overflows, or
- the rounded result of an operation underflows and both the Underflow and Inexact Enable bits are not set and the *FS* bit is set.

Trap Enabled Results: If Inexact exception traps are enabled, the result register is not modified and the source registers are preserved.

Trap Disabled Results: The rounded or overflowed result is delivered to the destination register if no other software trap occurs.

Invalid Operation Exception (V)

The Invalid Operation exception is signaled if one or both of the operands are invalid for an implemented operation. When the exception occurs without a trap, the MIPS ISA defines the result as a quiet Not a Number (qNaN). The invalid operations are:

- Addition or subtraction: magnitude subtraction of infinities, such as: (+ ∞) + (–∞) or (–∞) – (–∞)
- Multiplication: 0 times ∞, with any signs
- Division: 0/0, or ∞/∞, with any signs
- Comparison of predicates involving ‘<’ or ‘>’ without ‘?’, when the operands are unordered
- Any arithmetic operation, when one or both operands is a signaling NaN. A move (MOV) operation is not considered to be an arithmetic operation, but absolute value (ABS) and negate (NEG) are.
- Comparison or a Convert From Floating-point Operation on a signaling NaN.
- Square root: , where x is less than zero.

Software can simulate the Invalid Operation exception for other operations that are invalid for the given source operands. Examples of these operations include IEEE Standard 754-

specified functions implemented in software, such as Remainder: *x* REM *y*, where *y* is 0 or *x* is infinite; conversion of a floating-point number to a decimal format whose value causes an overflow, is infinity, or is NaN; and transcendental functions, such as ln (−5) or $\cos^{-1}$ (3). Refer to Appendix B for examples or for routines to handle these cases.

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: A quiet NaN is delivered to the destination register if no other software trap occurs.

Divide-by-Zero Exception (Z)

The Division-by-Zero exception is signaled on an implemented divide operation if the divisor is zero and the dividend is a finite nonzero number. Software can simulate this exception for other operations that produce a signed infinity, such as In (0), sec (π/2), csc (0), or 0^{-1}

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is a correctly signed infinity.

Overflow Exception (O)

The Overflow exception is signaled when the magnitude of the rounded floating-point result, with an unbounded exponent range, is larger than the largest finite number of the destination format. (This exception also signals an Inexact exception.)

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result (as listed in Table 12-1).

Underflow Exception (U)

Two related events contribute to the Underflow exception:

- creation of a tiny nonzero result between $\pm 2^{Emin}$ which can cause some later exception because it is so tiny
- extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers.

IEEE Standard 754 allows a variety of ways to detect these events, but requires they be detected the same way for all operations.

Tininess can be detected by one of the following methods:

- after rounding (when a nonzero result, computed as though the exponent range were unbounded, would lie strictly between $\pm 2^{Emin}$)
- before rounding (when a nonzero result, computed as though the exponent range and the precision were unbounded, would lie strictly between $\pm 2^{Emin}$).

The MIPS architecture requires that tininess be detected after rounding.

Loss of accuracy can be detected by one of the following methods:

- denormalization loss (when the delivered result differs from what would have been computed if the exponent range were unbounded)
- inexact result (when the delivered result differs from what would have been computed if the exponent range and precision were both unbounded).

The MIPS architecture requires that loss of accuracy be detected as an inexact result.

Trap Enabled Results: If Underflow or Inexact traps are enabled, or if the *FS* bit is not set, then an Unimplemented exception (E) is generated, and the result register is not modified.

Trap Disabled Results: If Underflow and Inexact traps are not enabled and the *FS* bit is set, the result is determined by the rounding mode and the sign of the intermediate result (as listed in Table 12-1).

Unimplemented Instruction Exception (E)

Any attempt to execute an instruction with an operation code or format code that has been reserved for future definition sets the *Unimplemented* bit in the *Cause* field in the FPU *Control/Status* register and traps. The operand and destination registers remain undisturbed and the instruction is emulated in software. Any of the IEEE Standard 754 exceptions can arise from the emulated operation, and these exceptions in turn are simulated.

The Unimplemented Instruction exception can also be signaled when unusual operands or result conditions are detected that the implemented hardware cannot handle properly. These include:

- Denormalized operand, except for Compare instruction
- Quiet Not a Number operand, except for Compare instruction
- Denormalized result or Underflow, when either Underflow or Inexact *Enable* bits are set or the *FS* bit is not set.
- Reserved opcodes
- Unimplemented formats
- Operations which are invalid for their format (for instance, CVT.S.S)

Note: Denormalized and NaN operands are only trapped if the instruction is a convert or computational operation. Moves do not trap if their operands are either denormalized or NaNs.

The use of this exception for such conditions is optional; most of these conditions are newly developed and are not expected to be widely used in early implementations. Loopholes are provided in the architecture so that these conditions can be implemented with assistance provided by software, maintaining full compatibility with the IEEE Standard 754.

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: This trap cannot be disabled.

13.6 Saving and Restoring State

Sixteen doubleword[†] coprocessor load or store operations save or restore the coprocessor floating-point register state in memory. The remainder of control and status information can be saved or restored through CFC1/CTC1 instructions, and saving and restoring the processor registers. Normally, the *Control/Status* register is saved first and restored last.

When state is restored, state information in the *Control/Status* register indicates the exceptions that are pending. Writing a zero value to the *Cause* field of *Control/Status* register clears all pending exceptions, permitting normal processing to restart after the floating-point register state is restored.

13.7 Trap Handlers for IEEE Standard 754 Exceptions

The IEEE Standard 754 strongly recommends that users be allowed to specify a trap handler for any of the five standard exceptions that can compute; the trap handler can either compute or specify a substitute result to be placed in the destination register of the operation.

By retrieving an instruction using the processor *Exception Program Counter* (*EPC*) register, the trap handler determines:

- exceptions occurring during the operation
- the operation being performed
- the destination format

On Overflow or Underflow exceptions (except for conversions), and on Inexact exceptions, the trap handler gains access to the correctly rounded result by examining source registers and simulating the operation in software.

On Overflow or Underflow exceptions encountered on floating-point conversions, and on Invalid Operation and Divide-by-Zero exceptions, the trap handler gains access to the operand values by examining the source registers of the instruction.

The IEEE Standard 754 recommends that, if enabled, the overflow and underflow traps take precedence over a separate inexact trap. This prioritization is accomplished in software; hardware sets the bits for both the Inexact exception and the Overflow or Underflow exception.

[†] 32 doublewords if the FR bit is set to 1.

14. Debug Support Unit

14.1 Features

1. Utilizes JTAG interface compatible with IEEE Std. 1149.1.
2. Additional Status pins and debug clock in conjunction with JTAG pins provide Real-Time Trace information.
3. Processor access to external processor probe to execute from the external trace memory during debug exception and boot time. This is to eliminate system memory for debugging purpose.
4. Supports DMA access through JTAG interface to internal processor bus to access internal registers, host system peripherals and system memory.
5. Debug functions
   - Instruction Address Break
   - Data Bus break
   - Processor Bus Break
   - Hardware Debug Interrupt
   - Reset, NMI, Interrupt Mask
6. Instructions for Debug
   - SDBBP, DERET, CTC0, CFC0
7. CP0 Registers for Debug
   - Debug, DEPC, DESAVE

14.2 EJTAG interface

This interface consists of two modes of operation a Run Time Mode and a Real Time Mode. The Run Time mode provides functions such as processor Run, Stop, Single Step, and access to internal registers and system memory. The Real Time mode provides additional status pins used in conjunction with JTAG pins for Real Time Trace information.

Pins	In/Out	Description
GTCK	I	Test Clock Input
GDCLK	O	Debug Clock (1/3 CPU Clock)
GTDI/GDINT	I	Test Data Input (GTDI) at Run Time mode /Debug Interrupt Input (GDINT) at Real Time mode
GTDO/GTPC[0]	O	Test Data Output (GTDO) /PC Output (GTPC)
GTMS	I	Test Mode Select Input
GTRST*	I	Reset
GPCST[8~0]	O	PC Trace Status Information
GTPC[3~1]	O	PC Output

14.3 JTAG Interface

Standard JTAG interface is used for on chip debugging during Run Time mode. The TX49 Debug Support Unit has following registers.

- Instruction Register
- Bypass Register
- Boundary-Scan Register
- Device Identification Register
- Implementation Register
- JTAG_Data_register
- JTAG_Address_Register
- JTAG_Control_Register

14.4 Processor Access Overview

The core processor can access external processor probe for reading and writing to external monitor memory, registers and other external resources.

In addition the processor can execute from the external monitor memory located from 0xf_ff20 0000 to 0xf_ff2f ffff when the ProbEnb bit is set and the processor probe is turned ON. Any access to the monitor location from 0xf_ff20 0000 to 0xf_ff3f ffff are only allowed when the processor is in the debug mode (DM = 1).

14.5 Instruction

The instruction is a 8 bit field. Instructions for the TX49 Debug Support Unit are encoded between 0x80 and 0x9f and other codes are reserved for Toshiba Standard JTAG instructions (Includes EXTEST, SAMPLE/PRELOAD, INTEST, IDCODE and HI-Z) and so on. Instructions are decoded as follows.

Hex Value	Instruction	Description
0x83	EJTAG_ImpCode	Select Implementation Register
0x88	JTAG_ADDRESS_IR	Select JTAG_Address Register
0x89	JTAG_DATA_IR	Select JTAG_Data Register
0x8A	JTAG_CONTROL_IR	Select JTAG_Control Register
0x8B	JTAG_ALL_IR	Select JTAG_All Register
0x90	PCTRACE	PCTRACE Instruction

Any unused instruction between 0x80 and 0x9f defaulted to BYPASS instruction.

14.6 Debug Unit

14.6.1 Extended Instructions

- SDBBP
- DERET
- CTC0
- CFC0

14.6.2 Extended Debug Registers in CP0

- Debug Register
- Debug Exception PC (DEPC)
- Debug SAVE

14.7 Register Map

Address	Mnemonic	Description
0xf ff30 0000	DCR	Debug Control Register
0xf ff30 0008	IBS	Instruction Break Status
0xf ff30 0010	DBS	Data Break Status
0xf ff30 0018	PBS	Processor Break Status
0xf ff30 0100	IBA0	Instruction Break Address 0
0xf ff30 0108	IBC0	Instruction Break Control 0
0xf ff30 0110	IBM0	Instruction Break Address Mask 0
0xf ff30 0300	DBA0	Data Break Address 0
0xf ff30 0308	DBC0	Data Break Control 0
0xf ff30 0310	DBM0	Data Break Address Mask 0
0xf ff30 0318	DB0	Data Break Value 0
0xf ff30 0600	PBA0	Processor Bus Break Address 0
0xf ff30 0608	PBD0	Processor Bus Break Data 0
0xf ff30 0610	PBM0	Processor Bus Break Mask 0
0xf ff30 0618	PBC0	Processor Bus Break Control 0

14.8 Processor Bus Break Function

This function is to monitor the interface to core and provide debug interruption or trace trigger for a given physical address and data.

14.9 Debug Exception

Three kinds of debug exception are supported.

- Debug Single Step (DSS bit)
- Debug Breakpoint Exception (SDBBP Instruction)
- JTAG Break Exception (Jtagbrk bit in JTAG_Control_Register)

Note: During real time debugging, first two functions are disabled.

14.9.1 Debug Single Step (DSS)

When the debug register DSS bit is set, this exception has been raised each time one instruction is executed.

14.9.2 Debug Breakpoint exception (Dbp)

This exception is raised when SDBBP instruction is executed.

14.9.3 JTAG Break Exception

This exception is raised when JTAG unit set the Jtagbrk in JTAG_Control_Register.

14.9.4 Debug Exception Handling

Updates DEPC and Debug register.

Registers other than DEPC and Debug register retain their values.

14.9.5 Branching to debug handler

If the ProbEnb bit in JTAG_Control_Register[15] is set, the debug exception vector is located at

PC: 0xffff ffff ff20 0200.

If the ProbEnb bit in JTAG_Conctrol_Register[15] is cleared, the debug exception vector is located at

PC: 0xffff ffff bfc0 0400.

14.9.6 Exception handling when in Debug Mode (DM bit is set)

All interrupts including NMI are masked. When the NMI interrupt has occurred during Debug mode, it is stored internally and the NMI interrupt is taken after debug handler is finished (DM is clear).

14.10 Real Time PC TRACE Output

In real time mode non-sequential Program Counter and trace information are outputted on GTPC[3~0] and GPCST[8~0]. at 1/3 of the processor clock speed.

15. TX49 MPU Core Signal Descriptions

The TX49 MPU core has a 64-bit bus interface that is upward compatible with the TX39 G-bus interface.

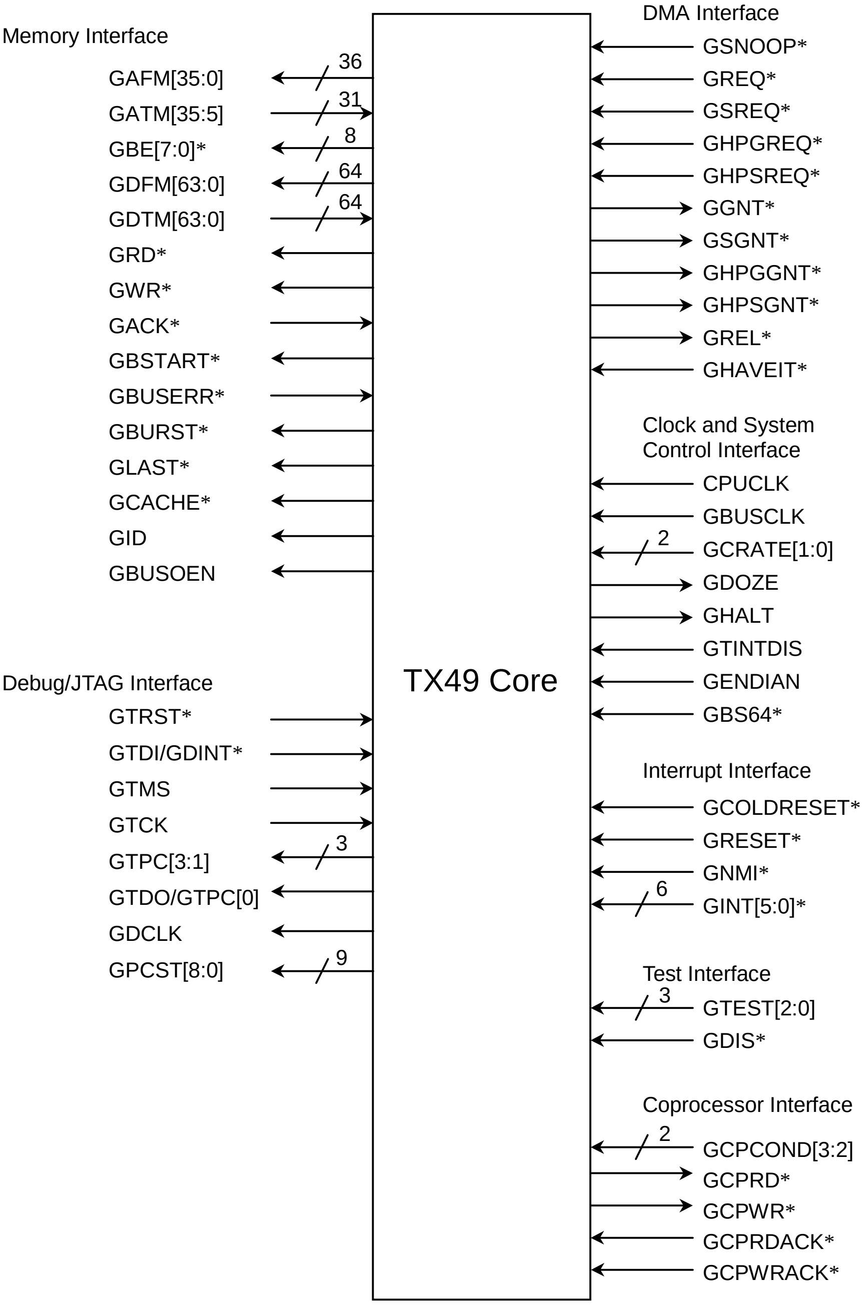


Figure 15-1 TX49 MPU Core Interface Signals

15.1 Signal Descriptions

15.1.1 Memory Interface Signals

Table 15-1 lists the memory interface signals.

Table 15-1 Memory Interface Signals

<table>
<tr><th>Signal Name</th><th>I/O</th><th>Active State</th><th>Description</th></tr>
<tr><td>GAFM[35:0]</td><td>O</td><td>–</td><td>Address From Bus (Output)
GAFM[35:0] is used as a 36-bit output address bus.</td></tr>
<tr><td>GATM[35:5]</td><td>I</td><td>–</td><td>Address To Bus (Input)
GATM[35:5] is a 31-bit address input bus used for data cache snooping.</td></tr>
<tr><td>GBE[7:0]*</td><td>O</td><td>Low</td><td>Byte Enable
GBE[7:0]* defines the valid data bytes within the 64-bit data bus. The correlation between the byte enable signals and data bytes is as follows:
<table>
<tr><th>Byte Enable</th><th>Corresponding Data Byte</th></tr>
<tr><td>GBE[7]*</td><td>GDFM[63:56], GDTM[63:56]</td></tr>
<tr><td>GBE[6]*</td><td>GDFM[55:48], GDTM[55:48]</td></tr>
<tr><td>GBE[5]*</td><td>GDFM[47:40], GDTM[47:40]</td></tr>
<tr><td>GBE[4]*</td><td>GDFM[39:32], GDTM[[39:32]</td></tr>
<tr><td>GBE[3]*</td><td>GDFM[31:24], GDTM[[31:24]</td></tr>
<tr><td>GBE[2]*</td><td>GDFM[23:16], GDTM[23:16]</td></tr>
<tr><td>GBE[1]*</td><td>GDFM[15:8], GDTM[15:8]</td></tr>
<tr><td>GBE[0]*</td><td>GDFM[7:0], GDTM[7:0]</td></tr>
</table></td></tr>
<tr><td>GDFM[63:0]</td><td>O</td><td>–</td><td>Data From Master (Output)
This data bus always acts as a 64-bit output.</td></tr>
<tr><td>GDTM[63:0]</td><td>I</td><td>–</td><td>Data to Master (Input)
This data bus always acts as a 64-bit input.</td></tr>
<tr><td>GRD*</td><td>O</td><td>Low</td><td>Read
GRD* is an output-only strobe that is asserted during a bus read operation.</td></tr>
<tr><td>GWR*</td><td>O</td><td>Low</td><td>Write
GWR* is an output-only strobe that is asserted during a bus write operation.</td></tr>
<tr><td>GACK*</td><td>I</td><td>Low</td><td>Read/Write Acknowledge
GACK* is sampled with the rising edge of GBUSCLK. The TX49 MPU core ends single-read and single-write operations in the next cycle after GACK* is recognized as asserted. During burst-read and burst-write operations, the TX49 MPU core increments the address at the next rising edge of GBUSCLK after GACK* is recognized as asserted. If GACK* is sampled as deasserted, a bus wait cycle is inserted.</td></tr>
<tr><td>GCACHE*</td><td>O</td><td></td><td>Cacheable
GCACHE* is an output signal that indicates whether the bus transfer in progress is being performed on a cached or uncached address space.
H: Uncached space
L: Cached space</td></tr>
<tr><td>GID</td><td>O</td><td></td><td>Instruction or Data
GID is an output signal that indicates the type of bus transfer being performed.
H: Instruction
L: Data</td></tr>
<tr><td>GBSTART*</td><td>O</td><td>Low</td><td>Bus Start
GBSTART* is an output signal that is asserted for one clock cycle to indicate that a bus operation has started.</td></tr>
<tr><td>GBUSERR*</td><td>I</td><td>Low</td><td>Bus Error
When GBUSERR* is asserted during a bus read operation, the TX49 MPU core immediately terminates the ongoing transaction and takes a Bus Error exception. GBUSERR* is valid only during bus read operations.</td></tr>
</table>

Signal Name	I/O	Active State	Description
GBURST*	O	Low	Burst GBURST* is an output-only strobe that is asserted during burst-read and burst-write operations.
GLAST*	O	Low	Last GLAST* is an output signal that indiates completion of a bus cycle. • During a single-read or single-write, GLAST* is asserted simultaneously with GBSTART*. • During a burst-read or burst-write, GLAST* is asserted when the TX49 MPU core has recognized a GACK* for the second last data read.
GBUSOEN*	O	Low	G-Bus Output Enable GBUSOEN* is the output enable control for the bus control signals: While the TX49 assumes bus mastershp: Low While the TX49 has released bus mastership: High While GDIS* is asserted: High

15.1.2 DMA Interface Signals

Table 15-2 lists the DMA interface signals.

Table 15-2 DMA Interface Signals

Signal Name	I/O	Active State	Description
GSNOOP*	I	Low	SNOOP The TX49 samples GNSOOP* with the rising edge of GBUSCLK. When GSNOOP* is recognized as asserted, the TX49 captures the address on GATM[35:5] and compares it to the addresses of all data items held in the on-chip data cache. If the snoop address hits in the data cache, the cache entry is invalidated. GSNOOP* is valid when either GHPSGNT* or GSGNT* is asserted.
GREQ*	I	Low	Normal Bus Request Alternate bus masters assert this signal to request bus mastershp as per ET concurrency protocols.
GSREQ*	I	Low	Snoop Bus Request Alternate bus masters assert this signal to request bus mastership as per ST concurrency protocols.
GHPGREQ*	I	Low	High-Priority Normal Bus Request In response to GHPGREQ*, the TX49 asserts GHPGGNT* to grant the bus to the requesting bus master as per ET concurrency protocols. GHPGREQ* has priority over GREQ* if both are asserted simultaneously.
GHPSREQ*	I	Low	High-Priority Snoop Bus Request In response to GHPSREQ*, the TX49 asserts GHPSGNT* to grant the bus to the requesting bus master as per ST concurrency protocols. GHPSREQ* has priority over GSREQ* if both are asserted simultaneously.
GGNT*	O	Low	Normal Bus Grant Assertion of GGNT* indicates that the TX49 has relinquished bus mastership in response to GREQ*.
GSGNT*	O	Low	Snoop Bus Grant Assertion of GSGNT* indicates that the TX49 has relinquished bus mastership in response to GSREQ*.
GHPGGNT*	O	Low	High-Priority Normal Bus Request Assertion of GHPGGNT* indicates that the TX49 has relinquished bus mastership in response to GHPGREQ*.
GHPSGNT*	O	Low	High-Priority Snoop Bus Grant Assertion of GHPSGNT* indicates that the TX49 has relinquished bus mastership in response to GHPSREQ*.
GREL*	O	Low	Release Request This output signal indicates to an external bus master that the TX49 wants to regain bus mastership. The TX49 asserts GREL* 1) when higher-priority GHPGREQ* is asserted while lower-priority GSGNT* is asserted and 2) when a bus request is generated from the TX49 processor core while GHPGGNT* is asserted.
GHAVEIT*	I	Low	Have IT This is a bus grant acknowledge signal used by an external bus master to indicate that it has assumed bus mastership. The external bus master can release the bus by asserting and deasserting GHAVEIT* while keeping a bus request signal asserted. In a single-bus-master system, GHAVEIT* may be tied high.

15.1.3 Coprocessor Interface Signals

Table 15-3 lists the coprocessor interface signals.

Table 15-3 Coprocessor Interface Signals

Signal Name	I/O	Active State	Description
GCPRD*	O	Low	Coprocessor Read GCPRD* is an output-only strobe that is asserted during a coprocessor read operation.
GCPWR*	O	Low	Coprocessor Write GCPWR* is an output-only strobe that is asserted during a coprocessor write operation.
GCPRDACK*	I	Low	Coprocessor Read Acknowledge A coprocessor asserts this signal to indicate to the TX49 processor core that the coprocessor read request has been acknowledged.
GCPWRACK*	I	Low	Coprocessor Write Acknowledge A coprocessor asserts this signal to indicate to the TX49 processor core that the coprocessor write request has been acknowledged.
GCPCOND[3:2]	I	–	Coprocessor Condition Coprocessor branch instructions use the GCPCOND[z] signal as the coprocessor z's condition signal: GCPCOND[3] is for CP3, and GCPCON[2] is for CP2.

15.1.4 Interrupt Interface Signals

Table 15-4 lists the interrupt interface signals.

Table 15-4 Interrupt Interface Signals

Signal Name	I/O	Active	Description
GCOLDRESET*	I	Low	Coldreset Assertion of this input signal initiates a cold reset and forces the TX49 to enter Cold Reset exception processing.
GRESET*	I	Low	Reset Assertion of this input signal initiates a soft reset and forces the TX49 to enter Soft Reset exception processing.
GNMI*	I	Low	Nonmaskable Interrupt Assertion of this input signal forces the TX49 to enter Nonmaskable Interrupt exception processing.
GINT[5:0]*	I	Low	Interrupt Assertion of any of these interrupt request inputs causes a general Interrupt exception unless the corresponding bit is masked in the Status register. GINT[5] can be configured for either a general interrupt input or a timer interrupt input during Reset exception processing. If the GTINTDIS input is zero during a reset sequence, GINT[5] is configured for the timer interrupt input.

15.1.5 Test Interface Signals

Table 15-5 lists the test interface signals.

Table 15-5 Test Interface Signals

Signal Name	I/O	Active State	Description
GTEST[2:0]	I	–	Test The GTEST[2:0] inputs are used to set up the TX49 in test mode. A value of 2'b000 at GTEST[2:0] puts the TX49 in normal operation mode.
GDIS*	I	–	Disable output This input must be tied high.

15.1.6 Debug Interface Signals

Table 15-6 lists the debug interface signals.

Table 15-6 Debug Interface Signals

Signal Name	I/O	Active State	Description
GTRST*	I	Low	Test Reset Input Assertion of this input initializes the on-chip Debug Support Unit (DSU).
GTDI/GDINT*	I	–	Test Data Input / Debug Interrupt Run-Time mode: Functions as a serial data input to the EJTAG instruction register. Real-Time mode: Switches the debug unit mode from Real-Time mode to Run-Time mode.
GTMS	I	–	Test Mode Select Input The GTMS input controls the transitions of the TAP controller in conjunction with the rising edge of GTCK.
GTCK	I	–	Test Clock Input GTCK is used to shift test data into or out of JTAG logic for EJTAG instructions. GTCK is independent of CPUCLK.
GTPC[3:1]	O	–	Trace PC Output. GTPC[3:1] provide non-sequential program counter output at the GDCLK speed.
GTDO/GTPC[0]	O	–	Test Data Output Run-Time mode: Shifts serial output data from the EJTAG data or instruction register. Real-Time mode: Provides a non-sequential program counter.
GPCST[8:0]	O	–	PC Trace Status The GPCST[8:0] outputs provide PC trace status information and serial monitor bus mode.
GDCLK	O	–	Debug Clock Output clock for EJTAG debug.

15.1.7 Clock and System Control Interface Signals

Table 15-7 lists the clock and system control interface signals.

Table 15-7 Clock and System Control Interface Signals

Signal Name	I/O	Active State	Description
CPUCLK	I	–	CPU Clock Input The TX49 processor core operates at the same frequency as CPUCLK.
GBUSCLK	I	–	GBUS Clock Input GBUSCLK is the clock input for the G-Bus interface. A divided-down clock must be applied to GBUSCLK according the value of GCRATE[1:0]. Otherwise, correct operation is not guaranteed.
GCRATE [1:0]	I	–	GBUS Clock Rate Input from External Pin GCRATE[1:0] selects the frequency at which the G-Bus interface runs with respect to the TX49 processor core. The frequency division factor can be one of the following; it must not be changed while the processor is running. GCRATE[1:0] 1/2 1/3 1/4 1/2.5
GDOZE	O	High	Doze GDOZE follows the state programmed into the Doze bit in the Config register. GDOZE=1 when the TX49 is in Doze mode.
GHALT	O	High	Halt GHALT follows the state programmed into the Halt bit in the Config register. GHALT=1 when the TX49 is in Halt mode.
GTINTDIS	I	–	Timer interrupt disable Input from External Pin GTINTDIS is specifies the pin function of GINT[5] during a reset sequence. H: Disables the timer interrupt function (i.e., configures the GINT[5] pin as a general interrupt request pin) L: Enables the timer interrupt function (i.e., configures the GINT[5] pin as a timer interrupt request pin.)
GENDIAN	I	–	Endianess Input from External Pin GENDIAN specifies byte ordering during a reset sequence. H: Big-endian L: Little-endian
GBS64*	I	–	System bus size. GBS64* specifies the G-Bus size during a reset sequence. H: 32-bit (GDTM[31:0] and GDFM[31:0] are valid.) L: 64-bit (GDTM[63:0] and GDFM[63:0] are valid.)

16. Low Power Consumption Modes

The TX49 can reduce its power consumption compared to the normal mode by controlling its internal clocks. The following two operation modes function as low power consumption modes of the TX49:

- Halt mode
- Doze mode

16.1 Halt mode

The halt mode reduces power consumption by halting TX49 operation. By setting the HALT bit of the Config register to 0 by the software and executing WAIT instruction, the TX49 mode shifts from the normal operation mode to the halt mode.

Therefore, as for bus control requests in the halt mode, a bus release request is responded to in cases of ET concurrency such as the GREQ* signal or the GHPGREQ* signal. However, the request is not responded to in cases of ST concurrency such as the GSREQ* signal or the GHPSREQ* signal. On the other hand, if WAIT instruction is executed while the bus is being released, the halt mode starts in cases of ET concurrency, but in cases of ST concurrency starts after bus ownership is granted and the GHALT signal is asserted.

If WAIT instruction is executed during a bus operation, the GHALT signal is asserted after the bus operation is completed.

If data remain in the write buffer, the write operation is executed even after shifting to the halt mode.

The internal halt bit is cleared by the assertion of the GINT[5~0]* signal, the GNMI* signal, the GRESET* signal or the GCOLDRESET* signal, and the TX49 return from the halt mode. If this is caused by the assertion of the GINT[5:0]* signal, the TX49 is released from the halt mode irrespective of the value in the IntMask field of the Status register. If the TX49 is brought back from the halt mode by the GCOLDRESET* signal, the GRESET* signal, the GNMI* signal, or a non-masked GINT[5~0]* signal, the initial instruction in the corresponding exception handler is executed. At this time, the EPC register is pointing to the instruction following the WAIT instruction. If it is recovered by a masked GINT[5~0]* signal, execution resumes from the instruction following the instruction that was being executed when it shifted to the halt mode.

As shown in Figure 16-1 the TX49 outputs the status of the internal halt bit on the GHALT signal. The memory interface output signals in the halt mode are maintained in the same status as when no bus operation was being executed.

Note: When the condition is brought back from the Power Consumption Modes are satisfied and WAIT instruction is executed, the TX49 does not shift to the mode.

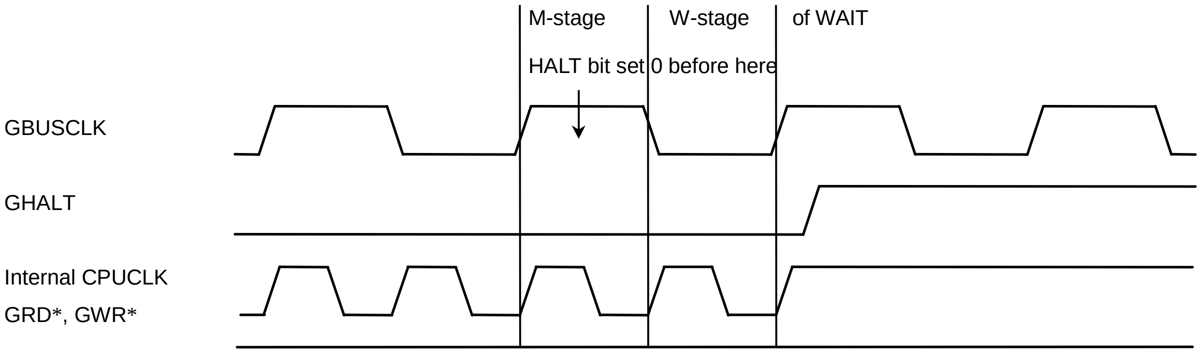


Figure 16-1 Halt Mode

16.2 Doze mode

The doze mode is also a mode which halts TX49 operation in order to lower power-consumption. However, the difference from the halt mode is that bus control requests (both ST concurrency and ET concurrency) from an external bus master can be responded to. Snooping operation of the data cache can also performed in ST concurrency. By setting the HALT bit of the Config register to 1 by the software and executing WAIT instruction, the TX49 mode shifts from the normal operation mode to the doze mode. Then, the TX49 Processor Core that is built into the TX49 halts operation while retaining the pipeline status.

As mentioned above, bus control requests are responded to while in the doze mode in cases of ET concurrency such as the GREQ* signal and the GHPGREQ* signal, and in cases of ST concurrency such as the GSREQ* signal and the GHPSREQ* signal. On the other hand, if WAIT instruction is executed while the bus is being released, the doze mode starts in cases of ET concurrency, but in cases of ST concurrency starts after bus ownership is granted and the GDOZE signal is asserted. If WAIT instruction is executed during a bus operation, the GDOZE signal is asserted after the bus operation is completed. The snooping of an external bus master is done by ST concurrency when the TX49 is in the doze mode. For the bus that is released by the assertion of the SGNT* signal or the GHPSGNT* signal, snooping of the data cache can be performed by the GSNOOP* signal and the GA[35~0] signal. When an external bus master deasserts the GSREQ* signal or the GHPSREQ* signal, the TX49 deasserts the GSGNT* signal or the GHPSGNT* signal.

By asserting the GINT[5~0]* signal, the GNMI* signal, the GRESET* signal or the GCOLDRESET* signal, the internal doze bit is cleared and the TX49 returns from the doze mode. If this is caused by the assertion of the GINT[5~0]* signal, the TX49 is released from the doze mode irrespective of the value in the IntMask field of the Status register. If the TX49 is brought back from the doze mode by the GCOLDRESET* signal, the GNMI* signal, or a non-masked GINT[5~0]* signal, the top instruction in the corresponding exception handler is executed. At this time, the EPC is pointing to the instruction following the WAIT instruction. If it is recovered by a masked GINT[5~0]* signal, execution resumes from the instruction following the instruction that was being executed when it shifted to the doze mode.

As shown in Figure 16-2, the TX49 outputs the status of the internal doze bit on the GDOZE signal. The memory interface output signals in the doze mode are maintained in the same status as when no bus operation was executed.

Note: When the condition is brought back from the Power Consumption Modes are satisfied and WAIT instruction is executed, the TX49 does not shift to the mode.

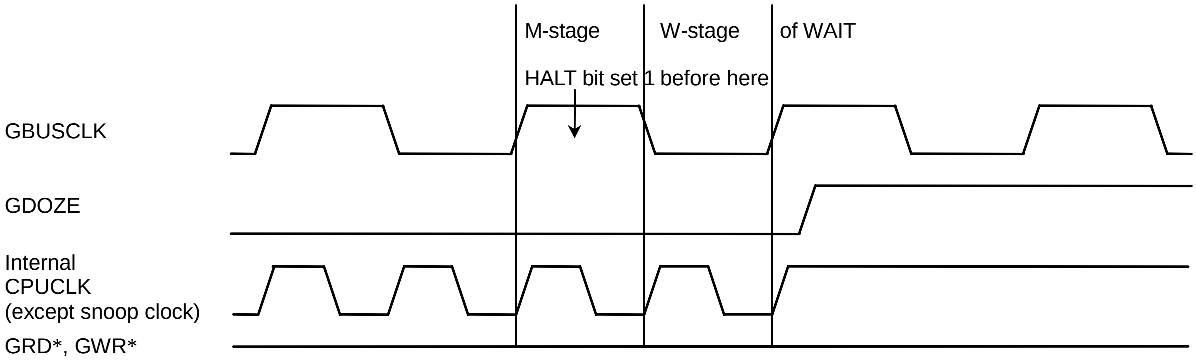


Figure 16-2 Doze Mode

16.3 Status Shifts

Figure 16-3 shows the status shifts in the operation mode of the TX49.

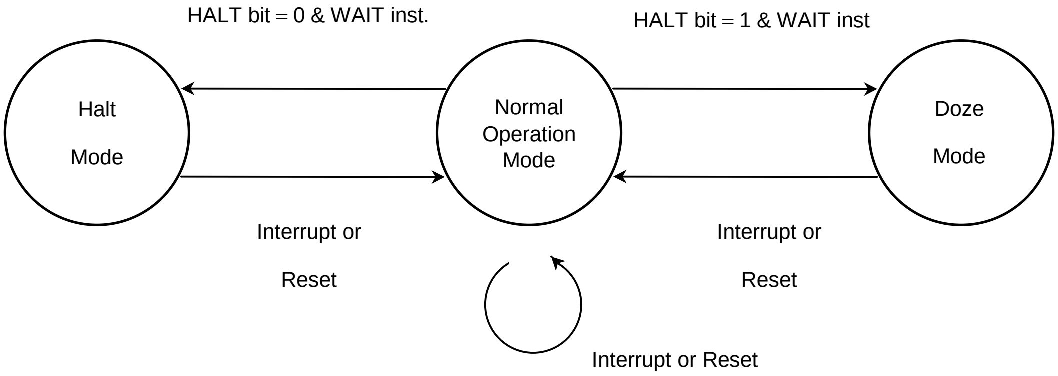


Figure 16-3 Status Shift Among Normal Operation Mode and Low Power Consumption Modes

When operation status shifts from the normal operation mode to the halt mode, it is returned to the normal operation mode by an interrupt or a reset. Similarly, when it shifts from the normal operation mde to the doze mode, it is returned to the normal operation mode by an interrupt or a reset. After a reset, the TX49 is initialized to the normal operation mode.

Appendix A: CPU Instruction Set Details

This appendix provides a detailed description of the operation of each TX49 instruction in both 32- and 64-bit modes. The instructions are listed in alphabetical order.

The exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. The description of the immediate causes and manner of handling exceptions is omitted from the instruction descriptions in this chapter.

Figures at the end of this appendix list the bit encoding for the constant fields of each instruction, and the bit encoding for each individual instruction is included with that instruction.

For a detailed description of the FPU instructions, refer to Appendix B.

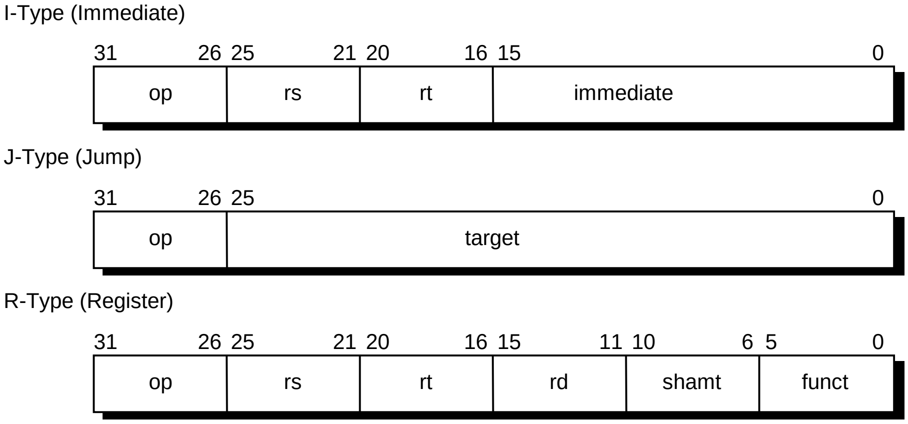
A.1 Instruction Classes

The TX49 has some classes of CPU instructions, as follows.

- Load and Store
- Computational
- Jump and Branch
- Coprocessor
- Special
- Exception
- Multiply and Divide
- Debug
- Others

A.1.1 Instruction Formats

Every instruction consists of a single word (32 bits) aligned on a word boundary. The main instruction formats are shown in Figure A-1.



where:

op	is a 6-bit operation code
rs	is a 5-bit source register specifier
rt	is a 5-bit target (source/destination) register or branch condition
immediate	is a 16-bit immediate, branch displacement or address displacement
target	is a 26-bit jump target address
rd	is a 5-bit destination register specifier
shamt	is a 5-bit shift amount
funct	is a 6-bit function field

Figure A-1 CPU Instruction Formats

A.1.2 Instruction Notation Conventions

In this appendix, all variable subfields in an instruction format (such as *rs*, *rt immediate*, etc.) are shown in lowercase names.

For the sake of clarity, we sometimes use an alias for a variable subfield in the formats of specific instructions. For example, we use *rs* = *base* in the format for load and store instructions. Such an alias is always lower case, since it refers to a variable subfield.

Figures with the actual bit encoding for all the mnemonics are located at the end of this Appendix, and the bit encoding also accompanies each instruction.

In the instruction descriptions that follow, the *Operation* section describes the operation performed by each instruction using a high-level language notation. The TX49 can operate as either a 32- or 64-bit microprocessor. The operation for both modes is included with the instruction description. Special symbols used in the notation are described in Table A-1.

Table A-1 CPU Instruction Operation Notations

Symbol	Meaning
←	Assignment.
||	Bit string concatenation.
x^y	Replication of bit value *x* into a *y*-bit string. Note: *x* is always a single-bit value.
$x_{y...z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation is always used. If *y* isess than *z*, this expression is an empty (zero length) bit string.
+	Two's complement or floating-point addition.
–	Two's complement or floating-point subtraction.
*	Two's complement or floating-point multiplication.
Div	Two's complement integer division.
Mod	Two's complement modulo.
/	Floating-point division.
<	Two's complement less than comparison.
And	Bitwise logic AND.
Or	Bitwise logic OR.
Xor	Bitwise logic XOR.
Nor	Bitwise logic NOR.
GPR[x]	General-Register *x*. The content of GPR[0] is always zero. Attempts to alter the content of GPR[0] have no effect.
CPR[z,x]	Coprocessor unit *z*, general register *x*.
CCR[z,x]	Coprocessor unit *z*, control register *x*.
COC[z]	Coprocessor unit *z* condition signal.
BigEndianMem	Big-endian mode as configured at reset (0 → Little, 1 → Big). Specifies the endianess of the memory interface (see LoadMemory and StoreMemory), and the endianess of Kernel and Supervisor mode execution.
ReverseEndian	Signal to reverse the endianess of load and store instructions. This feature is available in User mode only, and is effected by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR_{25} and User mode)
BigEndianCPU	The endianess for load and store instructions (0 → Little, 1 →Big). In User mode, this endianess may be reversed by setting SR_{25} Thus, BigEndianCPU may be computed as BigEndianMem XOR ReverseEndian.
Llbit	Bit of state to specify synchronization instructions. Set by LL, cleared by ERET and Invalidate and read by SC.
T + *i*:	Indicates the time steps between operations. Each of the statements within a time step are defined to be executed in sequential order (as modified by conditional and loop constructs). Operations which are marked *T + i*: are executed at instruction cycle *i* relative to the start of execution of the instruction. Thus, an instruction which starts at time *j* executes operations marked T + *i*: at time *i* + *j*. The interpretation of the order of excution between two instructions or two operations which execute at the same time should be pessimistic; the order is not defined.

A.1.3 Sign Extension and Zero Extension

With some instructions the bit length may be extended; for example, a 16-bit offset may be extended to 32 bits. This extension can take the from of either a sign extension or zero extension.

- Sign extension

The extended part is filled with the value of the most significant bit.

(example)

```
                1001100101011100    16 bit

11111111111111111001100101011100    32 bit
```

- Zero extension

The extended part is filled with zeros.

(example)

```
                1001100101011100    16 bit

00000000000000001001100101011100    32 bit
```

A.1.4 Instruction Notation Examples

The Following examples illustrate the application of some of the instruction notation conventions:

Example #1:
GPR[rt] $\leftarrow$ immediate $\|\|$ 0^{16}
Sixteen zero bits are concatenated with an immediate value (typically 16 bits), and the 32-bit string (with the lower 16 bits set to zero) is assigned to General-Purpose Register rt.
Example #2:
$(\text{immediate}_{15})^{16}$ $\|\|$ $\text{immediate}_{15\sim0}$
Bit 15 (the sign bit) of an immediate value is extended for 16 bit positions, and the result is concatenated with bits 15 through 0 of the immediate value to form a 32-bit sign extended value.

A.2 Load and Store Instructions

In the TX49 implementation, the instruction immediately following a load may use the contents of the register loaded. In such cases, the hardware interlocks, requiring additional real cycles, so scheduling load delay slots is still desirable, although not required for functional code.

Two special instructions are provided in the TX49 implementation of the MIPS ISA, Load Linked and Store Conditional. These instructions are used in carefully coded sequences to provide one of several synchronization primitives, including test-and-set, bit-level locks, semaphores, and sequencers / event counts.

In the load and store operation descriptions, the functions listed in Table A-2 are used to summarize the handling of virtual addresses and physical memory.

Table A-2 Load and Store Common Functions

Function	Meaning
AddressTranslation	Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the required translation is not present in the TLB.
LoadMemory	Uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order two bits of the address and the access type field indicates which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache.
StoreMemory	Uses the cache, write buffer, and main memory to store the word or part of word specified as data in the word containing the specified physical address. The low-order two bits of the address and the access type field indicates which of each of the four bytes within the data word should be stored.

The access type field indicates the size of the data item to be loaded or stored as shown in Table A-3. Regardless of access type or byte-numbering order (endianness), the address specifies the byte which has the smallest byte address of the bytes in the addressed field. For a Big-endian machine, this is the leftmost byte and contains the sign for a 2's-complement number; for a Little-endian machine, this is the rightmost byte and contains the lowest precision byte.

Table A-3 Access Type Specifications for Loads/Stores

Access Type Mnemonic	Value	Meaning
DOUBLEWORD	7	doubleword (64 bits)
SEPTIBYTE	6	seven bytes (56 bits)
SEXTIBYTE	5	six bytes (48 bits)
QUINTIBYTE	4	five bytes (40 bits)
WORD	3	word (32 bits)
TRIPLEBYTE	2	triple-byte (24 bits)
HALFWORD	1	halfword (16 bits)
BYTE	0	byte (8 bits)

The bytes within the addressed doubleword which are used can be determined directly from the access type and the three low-order bits of the address, as shown in Chapter 2.

A.3 Jump and Branch Instructions

All jump and branch instructions have an architectural delay of exactly one instruction. That is, the instruction immediately following a jump or branch (i.e., occupying the delay slot) is always executed while the target instruction is being fetched from storage. It is not valid for a delay slot to be occupied itself by a jump or branch instruction; however, this error is not detected, and the results of such an operation are undefined.

If an exception or interrupt prevents the completion of a legal instruction during a delay slot, the hardware sets the *EPC* register to point at the jump or branch instruction which precedes it. When the code is restarted, both the jump or branch instructions and the instruction in the delay slot are reexecuted.

Because jump and branch instructions may be restarted after exceptions or interrupts, they must be restartable. Therefore, when a jump or branch instruction stores a return link value, register 31 (the register in which the link is stored) may not be used as a source register.

Since instructions must be word-aligned, a Jump Register or Jump and Link Register instruction must use a register whose two low-order bits are zero. If these low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

A.4 Coprocessor Instructions

The MIPS architecture provides four coprocessor units, or classes. Coprocessors are alternate execution units, which have separate register files from the CPU. R-Series coprocessors have 2 register spaces, each with thirty-two 32-bit registers. The first space, *coprocessor general* registers, may be directly loaded from memory and stored into memory, and their contents may be transferred between the coprocessor and processor. The second, *coprocessor control* registers, may only have their contents transferred directly between the coprocessor and processor. Coprocessor instructions may alter registers in either space. Normally, by convention, *Coprocessor Control Register 0* is interpreted as a *Coprocessor Implementation And Revision* register. However, the system control coprocessor (CP0) uses *Coprocessor General Register 15* for the processor / coprocessor revision register. The register's low-order byte (bits 7~0) is interpreted as a coprocessor unit revision number. The second byte (bits 15~8) is interpreted as a coprocessor unit implementation descriptor. The revision number is a value of the form *y.x* where *y* is a major revision number in bits 7~4 and *x* is a minor revision number in bits 3~0.

The contents of the high-order halfword of the register are not defined (currently read as 0 and should be 0 when written).

A.5 System Control Coprocessor (CP0) Instructions

There are some special limitations imposed on operations involving CP0 that is incorporated within the CPU. Although load and store instructions to transfer data to and from coprocessors and move control to/from coprocessor instructions are generally permitted by the MIPS architecture, CP0 is given a somewhat protected status since it has responsibility for exception handling and memory management. Therefore, the move to/from coprocessor instructions are the only valid mechanism for reading from and writing to the CP0 registers.

Several coprocessor operation instructions are defined for CP0 to directly read, write, and probe TLB entries and to modify the operating modes in preparation for returning to User mode or interrupt-enabled states.

A.6 CPU Instructions

This appendix provides a detailed description of the operation of each TX49 instruction in both 32- and 64-bit modes.

Exceptions that may occur due to the execution of each instruction are listed after the description of each instruction.

For a detailed description of the exception of the exceptions, refer to Chapter 4.

ADD **Add** ADD

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6

Format:

ADD rd,rs,rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

An overflow exception occurs if the carries out of bits 30 and 31 differ (2's-complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

Operation:

```
32   T:   GPR[rd] ← GPR[rs] + GPR[rt]

64   T:   temp ← GPR[rs] + GPR[rt]
          GPR[rd] ← $(temp_{31})^{32}$ || $temp_{31\sim0}$
```

Exceptions:

Integer overflow exception

ADDI Add Immediate ADDI

31 26	25 21	20 16	15 0
ADDI 001000	rs	rt	immediate
6	5	5	16

Format:

ADDI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt*. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

An overflow exception occurs if carries out of bits 30 and 31 differ (2's-complement overflow). The destination register *rt* is not modified when an integer overflow exception occurs.

Operation:

```
32   T:   $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \mid\mid immediate_{15\sim0}$

64   T:   $temp \leftarrow GPR[rs] + (immediate_{15})^{48} \mid\mid immediate_{15\sim0}$
          $GPR[rt] \leftarrow (temp_{31})^{32} \mid\mid temp_{31\sim0}$
```

Exceptions:

Integer overflow exception

ADDIU Add Immediate Unsigned ADDIU

31 26	25 21	20 16	15 0
ADDIU 001001	rs	rt	immediate
6	5	5	16

Format:

ADDIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt*. No integer overflow exception occurs under any circumstances. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADDI instruction is that ADDIU never causes an overflow exception.

Operation :

```
32  T:  $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \,||\, immediate_{15\sim0}$

64  T:  $temp \leftarrow GPR[rs] + (immediate_{15})^{48} \,||\, immediate_{15\sim0}$
        $GPR[rt] \leftarrow (temp_{31})^{32} \,||\, temp_{31\sim0}$
```

Exceptions:

None

ADDU Add Unsigned ADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADDU 100001
6	5	5	5	5	6

Format:

ADDU rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*. No overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADD instruction is that ADDU never causes an overflow exception.

Operation:

```
32   T:   $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

64   T:   $temp \leftarrow GPR[rs] + GPR[rt]$
          $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp_{31\sim0}$
```

Exceptions:

None

AND **And** AND

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	AND 100100
6	5	5	5	5	6

Format:

AND rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical AND operation. The result is placed into general register *rd*.

Operation:

```
32   T:        GPR[rd] ← GPR[rs] + GPR[rt]

64   T:        GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

ANDI And Immediate ANDI

31 26	25 21	20 16	15 0
ANDI 001100	rs	rt	immediate
6	5	5	16

Format:

ANDI rt, rs, immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical AND operation. The result is placed into general register *rt*.

Operation:

```
32   T:    GPR[rt] ← $0^{16}$ || (immediate and GPR[rs]$_{15\sim0}$)

64   T:    GPR[rt] ← $0^{48}$ || (immediate and GPR[rs]$_{15\sim0}$)
```

Exceptions:

None

BCzF Branch On Coprocessor z False BCzF

31 26	25 21	20 16	15 0
COPz 0100xx*	BC 01000	BCF 00000	offset
6	5	5	16

Format:

BCzF offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If coprocessor z's condition signal (CpCond), as sampled during the previous instruction, is false, then the program branches to the target address with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

```
32   T-1:     condition ← not COC[z]
     T:       target ← $(offset_{15})^{14}$ || offset || $0^2$
     T + 1:   if condition then
                  PC ← PC + target
              endif
64   T-1      condition ← not COC[z]
     T:       target ← $(offset_{15})^{46}$ || offset || $0^2$
     T + 1:   if condition then
                  PC ← PC + target
              endif
```

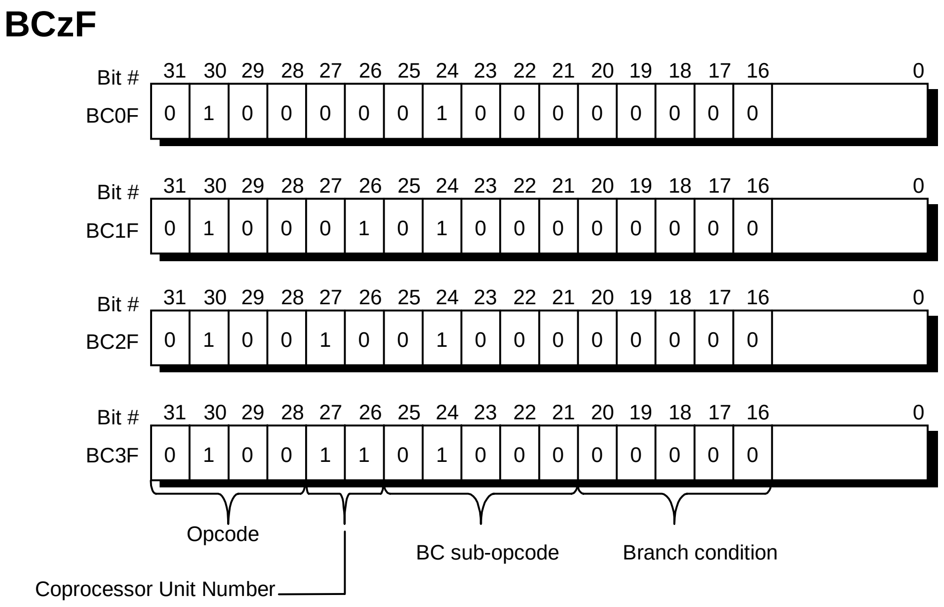
*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.

BCzF Branch On Coprocessor z False (continued) BCzF

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding



BCzF

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0F	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	
BC1F	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	
BC2F	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	
BC3F	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	

Note:

CpCond0 = Write Buffer Empty

(Empty → true (1), Not empty → false (0))

CpCond1 = FPU (See the Appendix B)

CpCond2 = External Pin condition (GCPCOND2)

CpCond3 = External Pin condition (GCPCOND3)

BCzFL Branch On Coprocessor z False Likely BCzFL

31 26	25 21	20 16	15 0
COPz 0100xx*	BC 01000	BCFL 00010	offset
6	5	5	16

Format:

BCzFL offset

Description :

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of coprocessor z's condition line, as sampled during the previous instruction, is false, the target address is branched to with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

BCzFL Branch On Coprocessor z False Likely (continued) BCzFL

Operation:

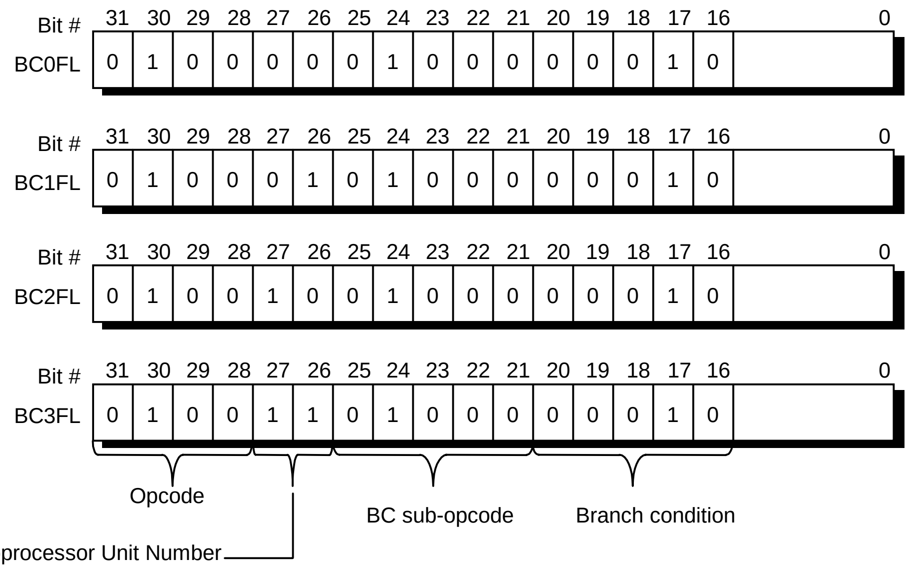
```
32   T-1:     condition ← not COC[z]
     T:       target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
     T + 1:   if condition then
                  PC ← PC + target
              else
                  NullityCurrentInstruction
              endif
64   T-1      condition ← not COC[z]
     T:       target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
     T + 1:   if condition then
                  PC ← PC + target
              else
                  NullifyCurrentInstruction
              Endif
```

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:

BCzFL



Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0FL	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	
BC1FL	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	
BC2FL	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	
BC3FL	0	1	0	0	1	1	0	1	0	0	0	0	0	0	1	0	

Note:

CpCond0 = Write Buffer Empty
(Empty → true (1), Not empty → false (0))

CpCond1 = FPU (See the Appendix B)

CpCond2 = External Pin condition (GCPCOND2)

CpCond3 = External Pin condition (GCPCOND3)

BCzT Branch On Coprocessor z True BCzT

31 26	25 21	20 16	15 0
COPz 0100XX*	BC 01000	BCT 00001	offset
6	5	5	16

Format:

BCzT offset

Description :

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the coprocessor z's condition signal (CpCond) is true, then the program branches to the target address, with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation :

```
32   T-1:     condition ← COC[z]
     T:       target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
     T + 1:   if condition then
                  PC ← PC + target
              endif
64   T-1      condition ← COC[z]
     T:       target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
     T + 1:   if condition then
                  PC ← PC + target

              Endif
```

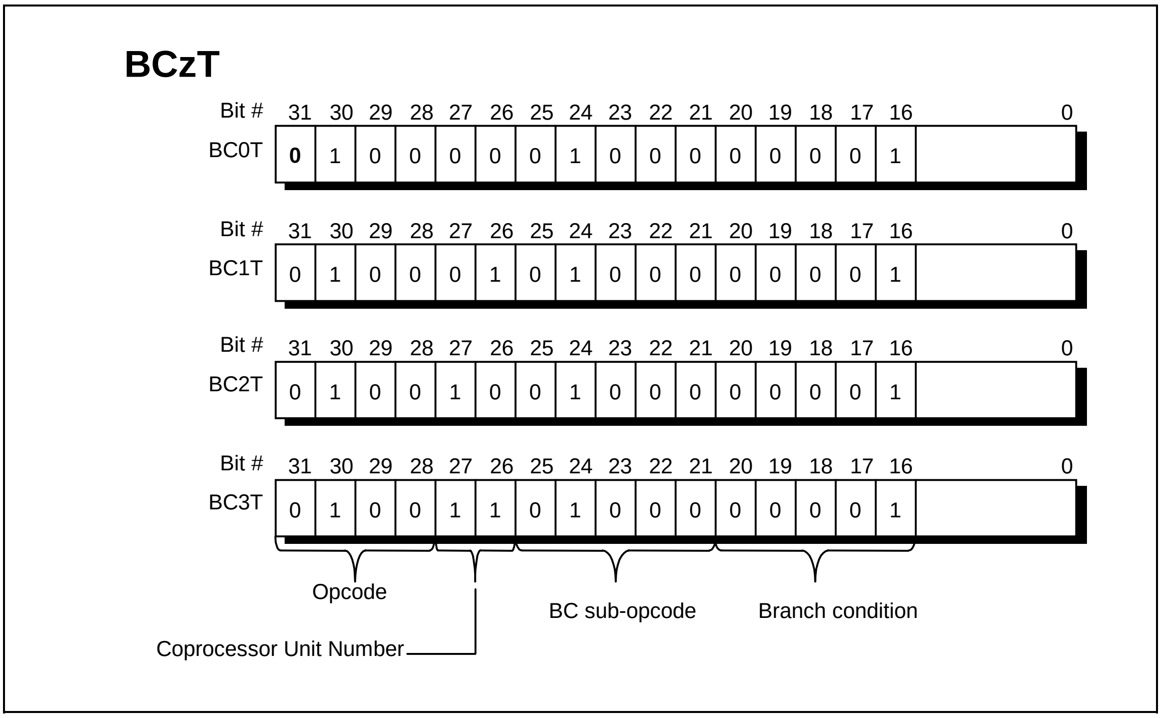
*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.

BCzT Branch On Coprocessor z True (continued) BCzT

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:



BCzT

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0T	**0**	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	
BC1T	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1	
BC2T	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	
BC3T	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	1	

Note:

CpCond0 = Write Buffer Empty

(Empty → true (1), Not empty → false (0))

CpCond1 = FPU (See the Appendix B)

CpCond2 = External Pin condition (GCPCOND2)

CpCond3 = External Pin condition (GCPCOND3)

BCzTL Branch On Coprocessor z True Likely BCzTL

31 26	25 21	20 16	15 0
COPz 0100XX*	BC 01000	BCTL 00011	offset
6	5	5	16

Format:

BCzTL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of coprocessor z's condition line, as sampled during the previous instruction, is true, the target address is branched to with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

```
32   T-1:     condition ← COC[z]
     T:       target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
     T + 1:   if condition then
              PC ← PC + target
              else
              NullifyCurrentInstruction
              endif
64   T-1      condition ← COC[z]
     T:       target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
     T + 1:   if condition then
              PC ← PC + target
              else
              NullifyCurrentInstruction
              endif
```

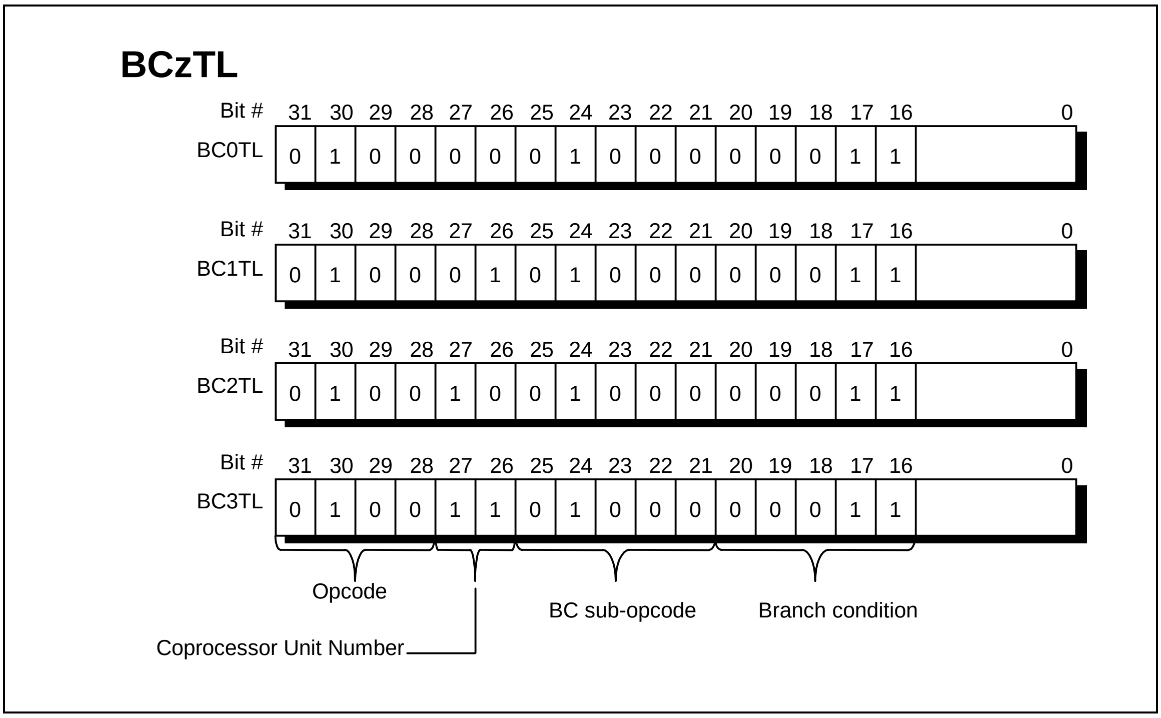
*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.

BCzTL **Branch On Coprocessor z True Likely (continued)** BCzTL

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:



BCzTL

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0TL	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	
BC1TL	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	1	
BC2TL	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	1	
BC3TL	0	1	0	0	1	1	0	1	0	0	0	0	0	0	1	1	

Note:

CpCond0 = Write Buffer Empty

(Empty → true (1), Not empty → false (0))

CpCond1 = FPU (See the Appendix B)

CpCond2 = External Pin condition (GCPCOND2)

CpCond3 = External Pin condition (GCPCOND3)

BEQ Branch On Equal BEQ

31 26	25 21	20 16	15 0
BEQ 000100	rs	rt	offset
6	5	5	16

Format:

BEQ rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* and the con-tents of general register *rt* are compared. If the two registers are equal, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32   T:       condition ← $(offset_{15})^{14}$ || offset || $0^2$
              condition ← (GPR[rs] = GPR[rt])
     T + 1:   if condition then
              PC ← PC + target
              endif
64   T:       target ← $(offset_{15})^{46}$ || offset || $0^2$
              condition ← (GPR[rs] = GPR[rt])
     T + 1:   if condition then
              PC ← PC + target
              endif
```

Exceptions:

None

BEQL Branch On Equal Likely BEQL

31 26	25 21	20 16	15 0
BEQL 010100	rs	rt	offset
6	5	5	16

Format:

BEQL rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are equal, the target address is branched to, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32   T:       target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
              condition ← (GPR[rs] = GPR[rt])
     T + 1:   if condition then
              PC ← PC + target
              else
              NullifyCurrentInstruction
              endif
64   T:       target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
              condition ← (GPR[rs] = GPR[rt])
     T + 1:   if condition then
              PC ← PC + target
              else
              NullifyCurrentInstruction
              endif
```

Exceptions:

None

BGEZ Branch On Greater Than Or Equal To Zero BGEZ

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZ 00001	offset
6	5	5	16

Format:

BGEZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32   T:       target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
              condition ← $(\text{GPR}[rs]_{31} = 0)$
     T + 1:   if condition then
              PC ← PC + target
              endif
64   T:       target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
              condition ← $(\text{GPR}[rs]_{63} = 0)$
     T + 1:   if condition then
              PC ← PC + target
              endif
```

Exceptions:

None

BGEZAL Branch On Greater Than Or Equal To Zero And Link BGEZAL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZAL 10001	offset
6	5	5	16

Format:

BGEZAL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31* . If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction is *not* tapped, however.

Operation:

```
32   T:       target ← $(offset_{15})^{14}$ || offset || $0^2$
              condition ← $(GPR[rs]_{31} = 0)$
              GPR[31] ← PC + 8
     T + 1:   if condition then
                  PC ← PC + target
              endif
64   T:       target ← $(offset_{15})^{46}$ || offset || $0^2$
              condition ← $(GPR[rs]_{63} = 0)$
              GPR[31] ← PC + 8
     T + 1:   if condition then
                  PC ← PC + target
              endif
```

Exceptions:

None

BGEZALL Branch On Greater Than Or Equal To Zero And Link Likely BGEZALL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZALL 10011	offset
6	5	5	16

Format:

BGEZALL rs, offset

Descriptions:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31* . If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction is *not* rapped, however. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

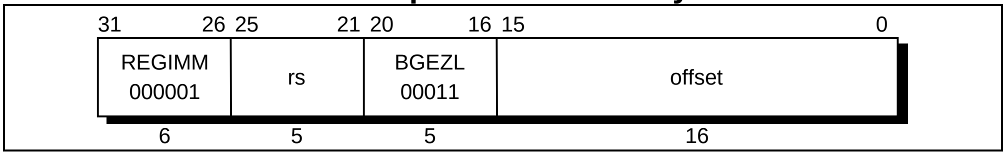
Operation:

```
32    T:       target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
               condition ← $(\text{GPR[rs]}_{31} = 0)$
               GPR[31] ← PC + 8
      T + 1:   if condition then
               PC ← PC + target
               Else
               NullifyCurrentInstruction
               Endif
64    T:       target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
               condition ← $(\text{GPR[rs]}_{63} = 0)$
               GPR[31] ← PC + 8
      T + 1:   if condition then
               PC ← PC + target
               Else
               NullifyCurrentInstruction
               Endif
```

Exceptions:

None

BGEZL Branch On Greater Than Or Equal To Zero Likely BGEZL



Format:

BGEZL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32   T:       $target \leftarrow (offset_{15})^{14} \,||\, offset \,||\, 0^2$
              $condition \leftarrow (GPR[rs]_{31} = 0)$
     T + 1:   if condition then
                  $PC \leftarrow PC + target$
              else
                  NullifyCurrentInstruction
              endif
64   T:       $target \leftarrow (offset_{15})^{46} \,||\, offset \,||\, 0^2$
              $condition \leftarrow (GPR[rs]_{63} = 0)$
     T + 1:   if condition then
                  $PC \leftarrow PC + target$
              else
                  NullifyCurrentInstruction
              endif
```

Exceptions:

None

BGTZ Branch On Greater Than Zero BGTZ

31 26	25 21	20 16	15 0
BGTZ 000111	rs	0 00000	offset
6	5	5	16

Format:

BGTZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* are compared to zero. If the contents of general register *rs* have the sign bit cleared and are not equal to zero, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32  T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← $(\text{GPR}[rs]_{31} = 0)$ and $(\text{GPR}[rs] \neq 0^{32})$
    T + 1:  if condition then
                PC ← PC + target
            endif
64  T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
            condition ← $(\text{GPR}[rs]_{63} = 0)$ and $(\text{GPR}[rs] \neq 0^{64})$
    T + 1:  if condition then
                PC ← PC + target
            endif
```

Exceptions:

None

BGTZL Branch On Greater Than Zero Likely BGTZL

31 26	25 21	20 16	15 0
BGTZL 010111	rs	0 00000	offset
6	5	5	16

Format:

BGTZL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* are compared to zero. If the contents of general register *rs* have the sign bit cleared and are not equal to zero, then the program branches to the target address, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32   T:      target ← $(\text{offset}_{15})^{14} \,||\, \text{offset} \,||\, 0^{2}$
             condition ← $(\text{GPR[rs]}_{31} = 0)$ *and* $(\text{GPR[rs]} \neq 0^{32})$
     T + 1:  if condition then
                 PC ← PC + target
             else
                 NullifyCurrentInstruction
             endif
64   T:      target ← $(\text{offset}_{15})^{46} \,||\, \text{offset} \,||\, 0^{2}$
             condition ← $(\text{GPR[rs]}_{63} = 0)$ *and* $(\text{GPR[rs]} \neq 0^{64})$
     T + 1:  if condition then
                 PC ← PC + target
             else
                 NullifyCurrentInstruction
             endif
```

Exceptions:

None

BLEZ Branch on Less Than Or Equal To Zero BLEZ

31 … 26	25 … 21	20 … 16	15 … 0
BLEZ 000110	rs	0 00000	offset
6	5	5	16

Format:

BLEZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* are compared to zero. If the contents of general register *rs* have the sign bit set, or are equal to zero, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32  T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← ($\text{GPR[rs]}_{31}$ = 1 ) or (GPR[rs] = $0^{32}$)
    T + 1:  if condition then
                PC ← PC + target
            endif
64  T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
            condition ← ($\text{GPR[rs]}_{63}$ = 1 ) or (GPR[rs] = $0^{64}$)
    T + 1:  if condition then
                PC ← PC + target
            endif
```

Exceptions:

None

BLEZL Branch on Less Than Or Equal To Zero Likely BLEZL

31 26	25 21	20 16	15 0
BLEZL 010110	rs	0 00000	offset
6	5	5	16

Format:

BLEZL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* is compared to zero. If the contents of general register *rs* have the sign bit set, or are equal to zero, then the program branches to the target address, with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32  T:      target ← $(offset_{15})^{14}$ || offset || $0^2$
            condition ← $(GPR[rs]_{31} = 1)$ or $(GPR[rs] = 0^{32})$
    T + 1:  if condition then
                PC ← PC + target
            else
                NullifyCurrentInstruction
            endif
64  T:      target ← $(offset_{15})^{46}$ || offset || $0^2$
            condition ← $(GPR[rs]_{63} = 1)$ or $(GPR[rs] = 0^{64})$
    T + 1:  if condition then
                PC ← PC + target
            else
                NullifyCurrentInstruction
            Endif
```

Exceptions:

None

BLTZ Branch On Less Than Zero BLTZ

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZ 00000	offset
6	5	5	16

Format:

BLTZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32    T:       target ← $(offset_{15})^{14}$ || offset || $0^2$
               condition ← $(GPR[rs]_{31} = 1)$
      T + 1:   if condition then
               PC ← PC + target
               endif
64    T:       target ← $(offset_{15})^{46}$ || offset || $0^2$
               condition ← $(GPR[rs]_{63} = 1)$
      T + 1:   if condition then
               PC ← PC + target
               endif
```

Exceptions:

None

BLTZAL Branch On Less Than Zero And Link BLTZAL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZAL 10000	offset
6	5	5	16

Format:

BLTZAL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31* . If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction.

General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction with register 31 specified as *rs* is *not* trapped, however.

Operation:

```
32    T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
              condition ← $(\text{GPR[rs]}_{31} = 1)$
              GPR[31] ← PC + 8
      T + 1:  if condition then
              PC ← PC + target
              endif
64    T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
              condition ← $(\text{GPR[rs]}_{63} = 1)$
              GPR[31] ← PC + 8
      T + 1:  if condition then
              PC ← PC + target
              endif
```

Exceptions:

None

BLTZALL Branch On Less Than Zero And Link Likely BLTZALL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZALL 10010	offset
6	5	5	16

Format:

BLTZALL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31* . If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction.

General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction with register *31* specified as *rs* is *not* trapped, however. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32      T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
                condition ← $(\text{GPR[rs]}_{31} = 1)$
                GPR[31] ← PC + 8
        T + 1:  if condition then
                PC ← PC + target
                else
                NullifyCurrentInstruction
                endif
64      T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
                condition ← $(\text{GPR[rs]}_{63} = 1)$
                GPR[31] ← PC + 8
        T + 1:  if condition then
                PC ← PC + target
                else
                NullifyCurrentInstruction
                endif
```

Exceptions:

None

BLTZL Branch On Less Than Zero Likely BLTZL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZL 00010	offset
6	5	5	16

Format:

BLTZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32    T:       target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
               condition ← $(\text{GPR[rs]}_{31} = 1)$
      T + 1:   if condition then
               PC ← PC + target
               else
               NullifyCurrentInstruction
               endif
64    T:       target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
               condition ← $(\text{GPR[rs]}_{63} = 1)$
      T + 1:   if condition then
               PC ← PC + target
               else
               NullifyCurrentInstruction
               endif
```

Exceptions:

None

BNE Branch On Not Equal BNE

31 26	25 21	20 16	15 0
BNE 000101	rs	rt	offset
6	5	5	16

Format:

BNE rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are not equal, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32      T:          target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
                    condition ← (GPR[rs] ≠ GPR[rt])
        T + 1:      if condition then
                    PC ← PC + target
                    endif
64      T:          target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
                    condition ← (GPR[rs] ≠GPR[rt])
        T + 1:      if condition then
                    PC ← PC + target
                    endif
```

Exceptions:

None

BNEL Branch On Not Equal Likely BNEL

31 26	25 21	20 16	15 0
BNEL 010101	rs	rt	offset
6	5	5	16

Format:

BNEL rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are not equal, then the program branches to the target address, with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

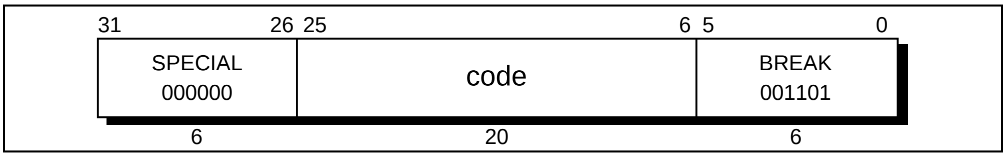
Operation:

```
32    T:       target ← $(offset_{15})^{14}$ || offset || $0^2$
               condition ← (GPR[rs] ≠ GPR[rt])
      T + 1:   if condition then
               PC ← PC + target
               else
               NullifyCurrentInstruction
               endif
64    T:       target ← $(offset_{15})^{46}$ || offset || $0^2$
               condition ← (GPR[rs] ≠GPR[rt])
      T + 1:   if condition then
               PC ← PC + target
               else
               NullifyCurrentInstruction
               endif
```

Exceptions:

None

BREAK Breakpoint BREAK



Format:

BREAK

Description:

A breakpoint trap occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64    T:        BreakpointException
```

Exceptions:

Breakpoint exception

CACHE Cache CACHE

31 26	25 21	20 16	15 0
CACHE 101111	base	op	offset
6	5	5	16

Format:

CACHE op, offset(base)

Description:

Generates a virtual address by sign-extending the 16-bit offset and adding the result to the contents of register base. The virtual address is translated to a physical address using the TLB, and the 5-bit sub-opecode designates the cache operation to be performed at that address.

If CP0 is unusable (in User or Supervisor mode), the CP0 enable bit in the Status register is cleared, and a Coprocessor Unusable Exception is raised. The behavior of this instruction for operation and cache combinations other than those listed in the table below, and when used with an uncached address, is undefined.

Cache index operations designate a cache block using part of the virtual address.

The memory address that specifies in cache instruction must be cacheable area. If uncachable area is specified, the operation is not guaranteed for TX49. If the instruction is issued for the line which this instruction itself exists, the following operation is not guaranteed.

The Index operation uses part of the virtual address to specify a cache block.

The each way is chosen by LSB (bit 0..1) of the virtual address.

Virtual Address bit (1:0)	Selected Way
00	Way 0
01	Way 1
10	Way 2
11	Way 3

The Hit operation accesses the specified cache as normal data references, and performs the specified operation if the cache block contains valid data with the specified physical address (a hit). If the cache block is invalid or contains a different address (a miss), no operation is performed. Write back from a cache goes to memory. The address to be written is specified by the cache tag and not the translated physical address. TLB Refill and TLB Invalid exceptions can occur on any operation. For Index operations (where the physical address is used to index the cache but need not match the cache tag) unmapped addresses may be used to avoid TLB exceptions. This operation never causes TLB Modified or Virtual Coherency exceptions. Bits 17~16 of the instruction specify the cache as follows:

Code	Name	Cache
0	I	Primary instruction
1	D	Primary data
2	-	reserved
3	-	reserved

CACHE **Cache (continued)** CACHE

Bits 20~18 of the instruction specify the operation as follows:

Code	Caches	Name	Operation
0	I	Index Invalidate	Set the cache state of the indexed block to invalid.
0	D	Index WriteBack Invalidate	Examine the cache state and W bit of the primary data cache block at the invalidate index specified by the virtual address. If the state is not invalid and the W bit is set, then write back the block to memory. The address to write is taken from the primary cache tag. Set cache state of primary cache block to invalid. LSB (bit 1 ~ 0) of VA select the way.
1	I / D	Index Load Tag	Read the tag for the cache block at the specified index and place it into the TagLo and TagHi CP0 registers. LSB (bit 1 ~ 0) of VA select the way.
2	I / D	Index Store Tag	Write the tag for the cache block at the specified index from the TagLo and TagHi CP0 registers. LSB (bit 1 ~ 0) of VA select the way.
3	I	Undefined	Undefined
3	D	Create Dirty Exclusive	This operation is used to avoid loading data needlessly from memory when writing new contents into an entire cache block. If the cacheblock does not contain the specified address, and the block is dirty, write it back to the memory. In all cases, set the cache block tag to the specified physical address, set the cache state to Dirty Exclusive.
4	I / D	Hit Invalidate	If the cache block contains the specified address, mark the cache block invalid. In case of multi-hit, lock bits of the specified line become ineffective and all way are invalidated.
5	I	Fill	Fill the primary instruction cache block from memory. LSB (bit 1 ~ 0) of VA select the way.
5	D	Hit WriteBack Invalidate	If the cache block contains the specified address, write back the data if it is dirty, and mark the cache block invalid.
6	I	Undefined	Undefined
6	D	Hit WriteBack	If the cache block contains the specified address, and the W bit is set, write back the data to memory, and clear the W bit.
7	I	Undefined	Undefined
7	D	Fill	Fill the primary data cache block from memory. LSB (bit 1 ~ 0) of VA select the way.

CACHE **Cache (continued)** CACHE

Operation:

```
32, 64   T:   vAddr ← $((offset_{15})^{48} \| offset15\sim0) + GPR[base]$
              (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
              CacheOp(op, cAddr, pAddr)
```

Exceptions:

Coprocessor unusable exception

TLB refill exception

TLB invalid exception

CFC0 Move Control From Coprocessor 0 CFC0

31 26	25 21	20 16	15 11	10 0
COP0 010000	CF 00010	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

CFC0 rt, rd

Description:

For ICE system only.

Loads the contents of Monitor memory into the general-purpose register rt.

Operation:

```
32    T:        data ← CCR[0,rd]
      T + 1:    GPR[rt] ← data
64    T:        data ← $(CCR[0,rd]_{31})^{32}$ || CCR[0, rd]
      T + 1:    GPR[rt] ← data
```

Exceptions:

Coprocessor Unusable exception

CFCz Move Control From Coprocessor CFCz

31 26	25 21	20 16	15 11	10 0
COPz 0100xx*	CF 00010	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

CFCz rt, rd

Description:

The contents of coprocessor control register *rd* of coprocessor unit *z* are loaded into general register *rt*.

Operation:

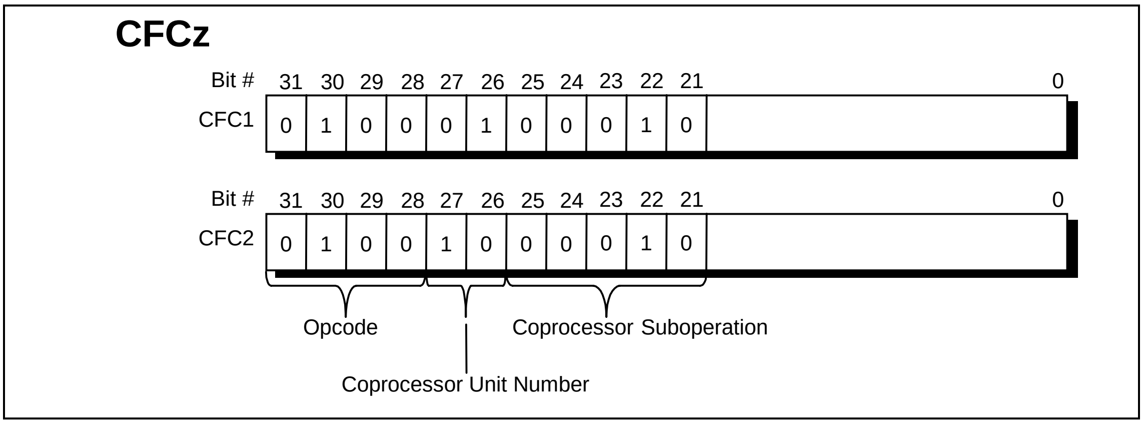
```
32    T:        data ← CCR[z,rd]
      T + 1:    GPR[rt] ← data
64    T:        data ← $(CCR[z,rd]_{31})^{32}$ || CCR[z, rd]
      T + 1:    GPR[rt] ← data
```

Exceptions:

Coprocessor unusable exception

Reserved Instruction exception (CFC3)

∗Opcode Bit Encoding:



CFCz

Bit #	31	30	29	28	27	26	25	24	23	22	21	0
CFC1	0	1	0	0	0	1	0	0	0	1	0	

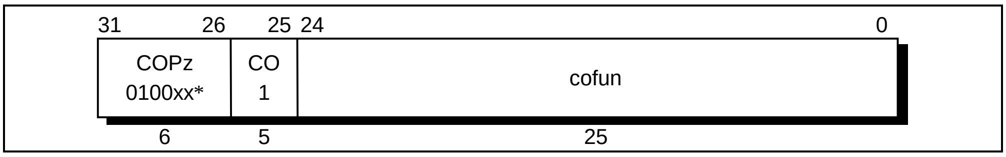
Bit #	31	30	29	28	27	26	25	24	23	22	21	0
CFC2	0	1	0	0	1	0	0	0	0	1	0	

Note:

CFC1 for FPU (See the Appendix B)

CFC2 for Coprocessor 2 (user define)

COPz Coprocessor z Operation COPz



Format:

COPz cofun.

Description:

A coprocessor operation is performed. The operation may specify and reference internal coprocessor registers, and may change the state of the coprocessor condition line, but does not modify state within the processor or the cache / memory system. Details of coprocessor 1 operations are contained in Appendix B.

Operation:

```
32, 64   T:          CoprocessorOperation(z, cofun)
```

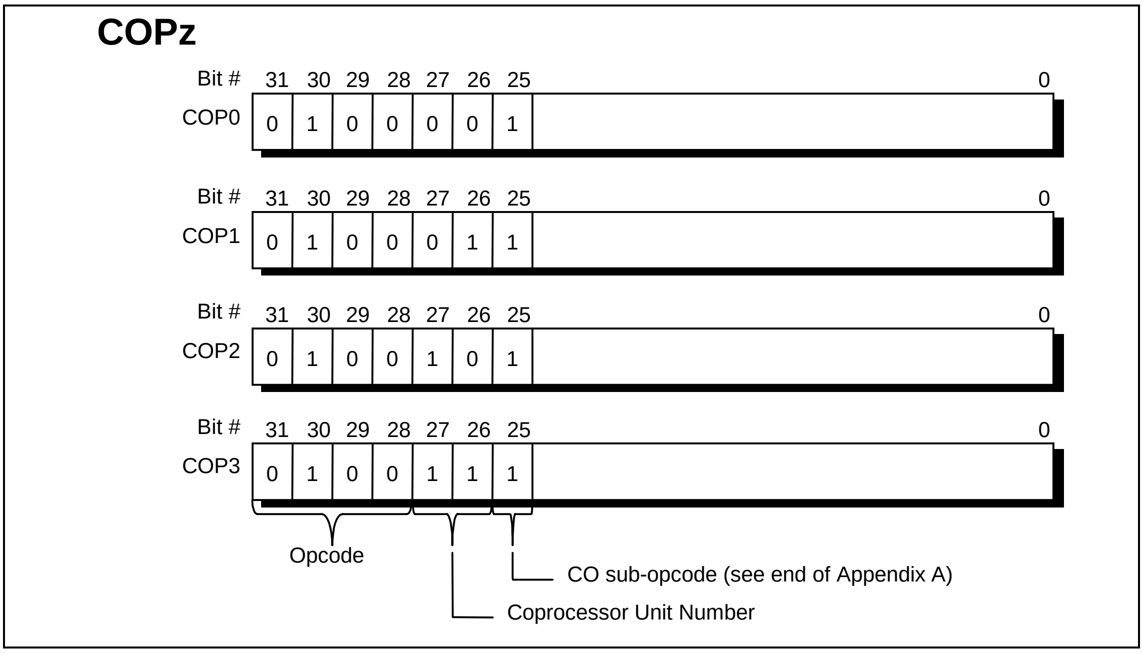
Exceptions:

Coprocessor unusable exception

Coprocessor interrupt or Floating-Point Exception (CP1 only)

Reserved Instruction exception (COP3)

∗Opcode Bit Encoding:


COPz
Bit # 31 30 29 28 27 26 25 0
COP0 0 1 0 0 0 0 1
Bit # 31 30 29 28 27 26 25 0
COP1 0 1 0 0 0 1 1
Bit # 31 30 29 28 27 26 25 0
COP2 0 1 0 0 1 0 1
Bit # 31 30 29 28 27 26 25 0
COP3 0 1 0 0 1 1 1
Opcode
CO sub-opcode (see end of Appendix A)
Coprocessor Unit Number

Note:

COP0 for ICE system

COP1 for FPU (See the Appendix B)

COP2 for Coprocessor 2 (user define)

CTC0 Move Control To Coprocessor 0 CTC0

31 26	25 21	20 16	15 11	10 0
COP0 010000	CT 00110	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

CTC0 rt, rd

Description:

For ICE system only.

Loads the contents of general-purpose register rt into the Monitor memory.

Operation:

```
32, 64   T:        data ← GPR[rt]
         T + 1:    CCR[0,rd] ← data
```

Exceptions:

Coprocessor Unusable exception

CTCz Move Control to Coprocessor z CTCz

31 26	25 21	20 16	15 11	10 0
COPz 0100xx*	CT 00110	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

CTCz rt, rd

Description:

The contents of general register *rt* are loaded into control register *rd* of coprocessor unit *z*.

Operation:

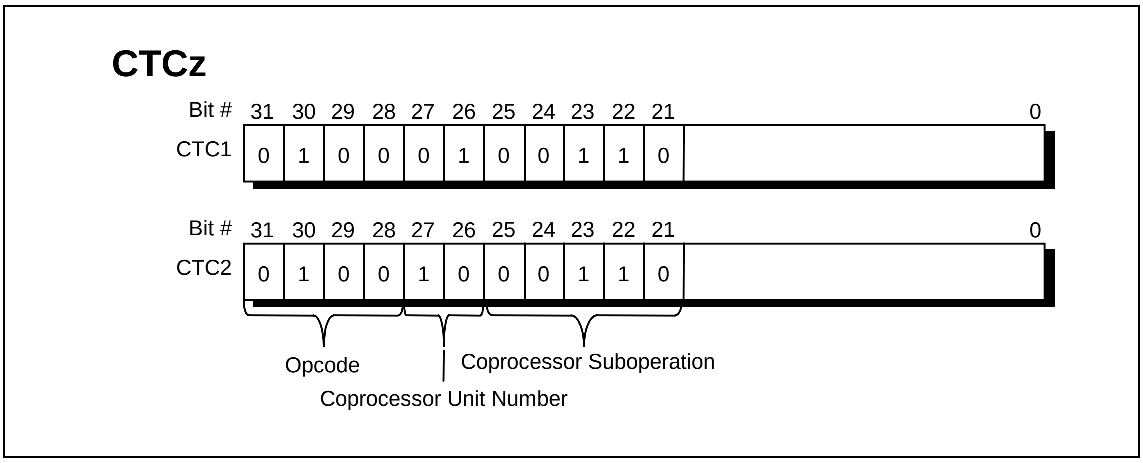
```
32, 64    T:       data ← GPR[rt]
          T + 1:   CCR[z,rd] ← data
```

Exceptions:

Coprocessor unusable

Reserved Instruction exception (CTC3)

* Opcode Bit Encoding:



CTCz

Bit #	31	30	29	28	27	26	25	24	23	22	21	0
CTC1	0	1	0	0	0	1	0	0	1	1	0	

Bit #	31	30	29	28	27	26	25	24	23	22	21	0
CTC2	0	1	0	0	1	0	0	0	1	1	0	

Note:

CTC1 for FPU (See the Appendix B)

CTC2 for Coprocessor 2 (user define)

∗See “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.

DADD Doubleword Add DADD

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DADD 101100
6	5	5	5	5	6

Format:

DADD rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*.

An overflow exception occurs if the carries out of bits 62 and 63 differ(2's-complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

Operation:

```
64    T:          GPR[rd] ← GPR[rs] + GPR[rt]
```

Exceptions:

Integer overflow exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DADDI Doubleword Add Immediate DADDI

31 26	25 21	20 16	15 0
DADDI 011000	rs	rt	immediate
6	5	5	16

Format:

DADDI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt*.

An overflow exception occurs if carries out of bits 62 and 63 differ (2's-complement overflow). The destination register *rt* is not modified when an integer overflow exception occurs.

Operation:

64	T:	$\text{GPR [rt]} \leftarrow \text{GPR[rs]} + (\text{immediate}_{15})^{48} \mid\mid \text{immediate}_{15\text{~}0}$

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Integer overflow exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DADDIU Doubleword Add Immediate Unsigned DADDIU

31 26	25 21	20 16	15 0
DADDIU 011001	rs	rt	immediate
6	5	5	16

Format:

DADDIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt*. No integer overflow exception occurs under any circumstances.

The only difference between this instruction and the DADDI instruction is that DADDIU never causes an overflow exception.

Operation:

64	T:	$\mathrm{GPR[rt]} \leftarrow \mathrm{GPR[rs]} + (\mathrm{immediate}_{15})^{48} \,||\, \mathrm{immediate}_{15\sim0}$

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DADDU Doubleword Add Unsigned DADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DADDU 101101
6	5	5	5	5	6

Format:

DADDU rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*.

No overflow exception occurs under any circumstances.

The only difference between this instruction and the DADD instruction is that DADDU never causes an overflow exception.

Operation:

```
64    T:        GPR [rd] ← GPR[rs] + GPR[rt]
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DDIV Doubleword Divide DDIV

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DDIV 011110
6	5	5	10	6

Format:

DDIV rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt*, treating both operands as 2's-complement values. No overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

This instruction is typically followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded into special register *LO*, and the remainder word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions.

Operation:

```
64   T-2:   LO      ← undefined
            HI      ← undefined
     T-1:   LO      ← undefined
            HI      ← undefined
     T:     LO      ← GPR[rs] div GPR[rt]
            HI      ← GPR[rs] mod GPR[rt]
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DDIVU Doubleword Divide Unsigned DDIVU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 000000 0000	DDIVU 011111
6	5	5	10	6

Format:

DDIVU rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt*, treating both operands as unsigned values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

This instruction is typically followed by additional instructions to check for a zero divisor.

When the operation completes, the quotient word of the double result is loaded into special register *LO*, and the remainder word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions.

Operation:

```
64   T-2:    LO        ← undefined
             HI        ← undefined
     T-1:    LO        ← undefined
             HI        ← undefined
     T:      LO        ← (0 || GPR[rs]) div (0 || GPR[rt])
             HI        ← (0 || GPR[rs]) mod (0 || GPR[rt])
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DERET Debug Exception Return DERET

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	DERET 011111
6	1	19	6

Format:

DERET

Description:

Execute a return a self-debug interrupt or exception. This instruction requires a branch delay slot like that of the branch or jump instructions, and executes with a delay of one instruction cycle. The DERET instruction itself cannot be put in the delay slot.

The return address stored in the DEPC register is copied to the PC, and processing returns to the original program.

Note: If a MTC0 instruction was used to set the return address in the DEPC register, a minimum of two instructions must be executed before executing DERET.

Operation:

```
32, 64   T:     temp ← DEPC
         T-1:   PC← temp
                $Debug_{30}$ ← 0
```

Exceptions:

Coprocessor unusable exception

DIV Divide DIV

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DIV 011010
6	5	5	10	6

Format:

DIV rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt*, treating both operands as 2's-complement values. No overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

This instruction is typically followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded into special register *LO*, and the remainder word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions.

Operation:

```
32   T-2:   LO     ← undefined
            HI     ← undefined
     T-1:   LO     ← undefined
            HI     ← undefined
     T:     LO     ← GPR[rs] div GPR[rt]
            HI     ← GPR[rs] mod GPR[rt]
64   T-2:   LO     ← undefined
            HI     ← undefined
     T-1:   LO     ← undefined
            HI     ← undefined
     T:     q      ← $GPR[rs]_{31\sim0}$ div $GPR[rt]_{31\sim0}$
            r      ← $GPR[rs]_{31\sim0}$ mod $GPR[rt]_{31\sim0}$
            LO     ← $(q_{31})^{32} \mid\mid q_{31\sim0}$
            HI     ← $(r_{31})^{32} \mid\mid r_{31\sim0}$
```

Exceptions:

None

DIVU Divide Unsigned DIVU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DIVU 011011
6	5	5	10	6

Format:

DIVU rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt*, treating both operands as unsigned values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero. In 64-bit mode, the operands must be valid sign-extended, 32-bit values. In 64-bitmode, the operands must be valid sign-extended, 32-bit values.

This instruction is typically followed by additional instructions to check for a zero divisor.

When the operation completes, the quotient word of the double result is loaded into special register *LO*, and the remainder word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions.

Operation:

```
32   T-2:   LO   ← undefined
            HI   ← undefined
     T-1:   LO   ← undefined
            HI   ← undefined
     T:     LO   ← (0 || GPR[rs]) div (0 || GPR[rt])
            HI   ← (0 || GPR[rs]) mod (0 || GPR[rt])
64   T-2:   LO   ← undefined
            HI   ← undefined
     T-1:   LO   ← undefined
            HI   ← undefined
     T:     q    ← $(0 \,||\, GPR[rs]_{31\sim0})$ div $(0 \,||\, GPR[rt]_{31\sim0})$
            r    ← $(0 \,||\, GPR[rs]_{31\sim0})$ mod $(0 \,||\, GPR[rt]_{31\sim0})$
            LO   ← $(q_{31})^{32} \,||\, q_{31\sim0}$
            HI   ← $(r_{31})^{32} \,||\, r_{31\sim0}$
```

Exceptions:

None

DMFC0 Doubleword Move From System Control Coprocessor DMFC0

31 26	25 21	20 16	15 11	10 0
COP0 010000	DMF 00001	rt	rd	0 000 0000 0000
6	5	5	5	5

Format:

DMFC0 rt, rd

Description:

The contents of coprocessor register rd of the CP0 are loaded into general register rt.

This operation is defined in kernel mode regardless of the setting of the Status. KX bit. Execution of this instruction with in supervisor mode with Status. SX = 0 or in user mode with UX = 0, causes a reserved instruction exception. All 64-bits of the general register destination are written from the coprocessor register source. The operation of DMFC0 on a 32-bit coprocessor 0 register is undefined.

Operation:

```
64    T:         data ← CPR[0,rd]
      T + 1:     GPR[rt] ← data
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Coprocessor unusable exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DMTC0 Doubleword Move TO System Control Coprocessor DMTC0

31 26	25 21	20 16	15 11	10 0
COP0 010000	DMT 00101	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

DMTC0 rt, rd

Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of the CP0.

This operation is defined for the R4000 operating in 64-bit mode or in 32-bit kernal mode. Execution of this instruction in 32-bit user or supervisor mode causes a reserved instruction exception. All 64-bits of he coprocessor 0 register are written from the general register source. The operation of DMTC0 on a 32-bit coprocessor 0 register is undefined.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load, store instructions and TLB operations immediately prior to and after this instruction are undefined.

Operation:

```
64   T:        data ← GPR[rt]
     T + 1:    CPR[0,rd] ← data
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Coprocessor unusable exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DMULT Doubleword Multiply DMULT

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DMULT 011100
6	5	5	10	6

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 0 0000	DMULT 011100
6	5	5	5	5	6

Format:

DMULT rs, rt

DMULT rd, rs, rt

Description:

The contents of general registers *rs* and *rt* are multiplied, heating both operands as 2's-complement values. No integer overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two other instructions.

Operation:

```
64   T-2:   LO        ← undefined
            HI        ← undefined
     T-1:   LO        ← undefined
            HI        ← undefined
     T:     t         ← GPR[rs]∗ GPR[rt]
            LO        ← $t_{63\sim0}$
            HI        ← $t_{127\sim64}$
            GPR[rd]   ← $t_{63\sim0}$
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DMULTU Doubleword Multiply Unsigned DMULTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DMULTU 011101
6	5	5	10	6

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 0 0000	DMULTU 011101
6	5	5	5	5	6

Format:

DMULTU rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are multiplied, treating both operands as unsigned values. No over-flow exception occurs under any circumstances.

When the operation completes, the low-order word of the double re-suit is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the re-suits of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two instructions.

Operation:

```
64   T-2:   LO        ← undefined
            HI        ← undefined
     T-1:   LO        ← undefined
            HI        ← undefined
     T:     t         ← (0 || GPR[rs]) * (0 || GPR[rt])
            LO        ← $t_{63\sim0}$
            HI        ← $t_{127\sim64}$
            GPR[rd]   ← $t_{63\sim0}$
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DSLL Doubleword Shift Left Logical DSLL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL 111000
6	5	5	5	5	6

Format:

DSLL rd, rt, sa

Description:

The contents of general register *rt* are shifted left by *sa* bits, inserting zeros into the low-order bits. The result is placed in register *rd*.

Operation:

```
64    T:    $s \leftarrow 0 \| sa$
            $GPR[rd] \leftarrow GPR[rt]_{(63-sa)\sim 0} \| 0^{s}$
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DSLLV Doubleword Shift Left Logical Variable DSLLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSLLV 010100
6	5	5	5	5	6

Format:

DSLLV rd, rt, rs

Description:

The contents of general register *rt* are shifted left by the number of bits specified by the low-order six bits contained as contents of general register *rs*, inserting zeros into the low-order bits. The result is placed in register *rd*.

Operation :

```
64    T:    $s \leftarrow GPR[rs]_{5\sim0}$
            $GPR[rd] \leftarrow GPR[rt]_{(63-s)\sim0} \| 0^{s}$
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DSLL32 Doubleword Shift Left Logical + 32 DSLL32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL32 111100
6	5	5	5	5	6

Format:

DSLL32 rd, rt, sa

Description:

The contents of general register *rt* are shifted left by 32 + *sa* bits, inserting zeros into the low-order bits. The result is placed in register *rd*.

Operation:

```
64    T:    $s \leftarrow 1 || sa$
            $GPR[rd] \leftarrow GPR[rt]_{(63-s)\ldots0} || 0^{s}$
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DSRA Doubleword Shift Right Arithmetic DSRA

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA 111011
6	5	5	5	5	6

Format:

DSRA rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, sign-ex-tending the high-order bits. The result is placed in register *rd*.

Operation:

```
64    T:    $s \leftarrow 0 || sa$
            $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} || GPR[rt]_{63\sim s}$
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DSRAV Doubleword Shift Right Arithmetic Variable DSRAV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSRAV 010111
6	5	5	5	5	6

Format:

DSRAV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order six bits of general register *rs*, sign-ex-tending the high-order bits. The result is placed in register *rd*.

Operation:

64 T: $s \leftarrow GPR[rs]_{5\sim0}$

$GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \, || \, GPR[rt]_{63\sim s}$

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DSRA32 Doubleword Shift Right Arithmetic + 32 DSRA32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA32 111111
6	5	5	5	5	6

Format:

DSRA32 rd, rt,sa

Description:

The contents of general register *rt* are shifted right by 32 + *sa* bits, sign-extending the high-order bits. The result us placed in register *rd*.

Operation:

```
64    T:    $s \leftarrow 1 || sa$
            $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} || GPR[rt]_{63\sim s}$
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DSRL Doubleword Shift Right Logical DSRL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRL 111010
6	5	5	5	5	6

Format:

DSRL rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, inserting zeros into the high-order bits. The result is placed in register *rd*.

Operation:

```
64    T:    $s \leftarrow 0 \| sa$
            $GPR[rd] \leftarrow 0^{s} \| GPR[rt]_{63\sim s}$
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DSRLV Doubleword Shift Right Logical Variable DSRLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSRLV 010110
6	5	5	5	5	6

Format:

DSRLV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order six bits of general register *rs*, inserting zeros unto the high-order bits. The result us placed in register *rd*.

Operation:

```
64    T:    $s \leftarrow GPR[rs]_{5\sim0}$
            $GPR[rd] \leftarrow 0^{s} \| GPR[rt]_{63\sim s}$
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DSRL32 Doubleword Shift Right Logical + 32 DSRL32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRL32 111110
6	5	5	5	5	6

Format:

DSRL32 rd, rt, sa

Description:

The contents of general register *rt* are shifted right by 32 + *sa* bits, inserting zeros into the high-order bits. The result is placed in register *rd*.

Operation:

```
64    T:    $s \leftarrow 1 \,||\, sa$
            $GPR[rd] \leftarrow 0^{s} \,||\, GPR[rt]_{63 \sim s}$
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DSUB **Doubleword Subtract** DSUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSUB 101110
6	5	5	5	5	6

Format:

DSUB rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd*.

The only difference between this instruction and the DSUBU instruction is that DSUBU never traps on overflow.

An integer overflow exception takes place if the carries out of bits 62and 63 differ (2's-complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

Operation :

```
64      T:          GPR[rd] ← GPR[rs] – GPR[rt]
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Integer overflow exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

DSUBU Doubleword Subtract Unsigned DSUBU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSUBU 101111
6	5	5	5	5	6

Format:

DSUBU rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd*.

The only difference between this instruction and the DSUB instruction is that DSUBU never taps on overflow. No integer overflow exception occurs under any circumstances.

Operation:

```
64    T:        GPR[rd] ← GPR[rs] – GPR[rt]
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

ERET Exception Return ERET

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	ERET 011000
6	1	19	6

Format:

ERET

Description:

ERET is the TX49 instruction for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, ERET does not execute the next instruction.

ERET must not itself be placed in a branch delay slot.

If the processor is servicing an error trap ($SR_2 = 1$), then load the PC from the *ErrorEPC* and clear the *ERL* bit of the *Status* register (SR_2). Otherwise ($SR_2 = 0$), load the PC from the *EPC*, and clear the *EXL* bit of the *Status* register (SR_1).

An ERET executed between a LL and SC also causes the SC to fail.

In case of this instruction is placed in the boundary of memory, it is necessary to keep the branch delay slot into same memory area.

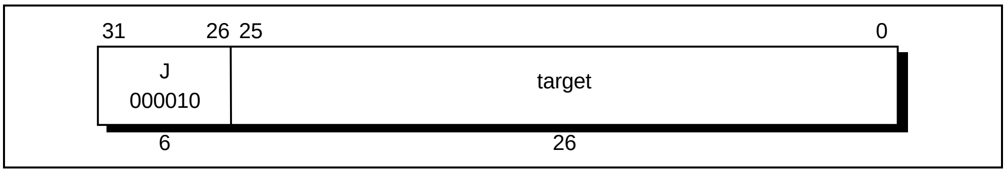
Operation:

```
32, 64   T:   if $SR_2 = 1$ then
                  PC ← ErrorEPC
                  SR ← $SR_{31\sim3}$ || 0 || $SR_{1\sim0}$
              else
                  PC ← EPC
                  SR ← $SR_{31\sim2}$ || 0 || $SR_0$
              endif
              LLbit ← 0
```

Exceptions:

Coprocessor unusable exception

J Jump J



Format:

J target

Description:

The 26-bit target address is shifted left two bits and combined with the high-order bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction.

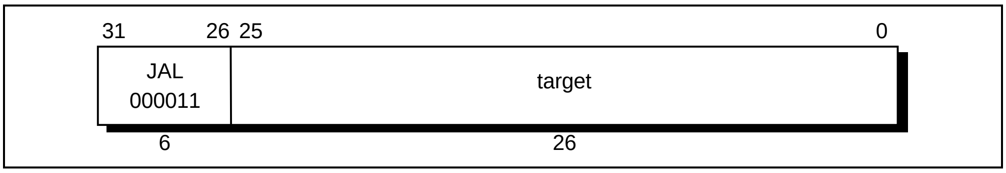
Operation:

```
32   T:       $temp \leftarrow target$
     T + 1:   $PC \leftarrow PC_{31\sim28} \,||\, temp \,||\, 0^2$
64   T:       $temp \leftarrow target$
     T + 1:   $PC \leftarrow PC_{63\sim28} \,||\, temp \,||\, 02$
```

Exceptions:

None

JAL Jump And Link JAL



Format:

JAL target

Description:

The 26-bit target address is shifted left two bits and combined with the high-order bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register, *r31*.

Operation:

```
32   T:      temp ← target
             GPR[31] ← PC + 8
     T + 1:  PC ← $PC_{31\sim28}$ || temp || $0^2$
64   T:      temp ← target
             GPR[31] ← PC + 8
     T + 1:  PC ← $PC_{63\sim28}$ || temp || $0^2$
```

Exceptions:

None

JALR Jump And Link Register JALR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	0 00000	rd	0 00000	JALR 001001
6	5	5	5	5	6

Format:

JALR rs

JALR rd, rs

Description:

The program unconditionally jumps to the address contained in general register rs, with a delay of one instruction. The address of the instruction after the delay slot is placed in general register *rd*. The default value of *rd*, if omitted in the assembly language instruction, is 31.

Register specifiers *rs* and *rd* may not be equal, because such an instruction does not have the same effect when reexecuted. However, an attempt to execute this instruction is *not* trapped, and the result of executing such an instruction is undefined.

Since instructions must be word-aligned, a *Jump and Link Register* instruction must specify a target register (rs) whose two low-order bits are zero. If these. low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

Operation:

```
32, 64  T:      temp ← GPR[rs]
                GPR[rd] ← PC + 8
        T + 1:  PC ← temp
```

Exceptions:

None

JR Jump Register JR

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000 0000 0000 0000	JR 001000
6	5	15	6

Format:

JR rs

Description:

The program unconditionally jumps to the address contained in general register *rs*, with a delay of one instruction.

Since instructions must be word-aligned, a *Jump Register* instruction must specify a target register (*rs*) whose two low-order bits are zero. If these low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

Operation:

```
32, 64  T:       temp ← GPR[rs]
        T + 1:   PC ← temp
```

Exceptions:

None

LB Load Byte LB

31 … 26	25 … 21	20 … 16	15 … 0
LB 100000	base	rt	offset
6	5	5	16

Format:

LB rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added tp the contents of general register *base* to form a virtual address. The contents of the byte at the memory location specified by the effective address are sign-extended and loaded unto general register *rt*.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15\sim0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1\sim3} \| (pAddr_{2\sim0}\ xor\ ReverseEndian^{3})$
          $mem \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2\sim0}\ \mathit{xor}\ BigEndianCPU^{3}$
          $GPR[rt] \leftarrow (mem_{7+8*byte})^{24} \| mem_{7+8*byte\sim8*byte}$
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15\sim0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1\sim3} \| (pAddr_{2\sim0}\ xor\ ReverseEndian^{3})$
          $mem \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2\sim0}\ \mathit{xor}\ BigEndianCPU^{3}$
          $GPR[rt] \leftarrow (mem_{7+8*byte})^{56} \| mem_{7+8*byte\sim8*byte}$
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LBU Load Byte Unsigned LBU

31 26	25 21	20 16	15 0
LBU 100100	base	rt	offset
6	5	5	16

Format:

LBU rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the byte at the memory location specified by the effective address are zero-extended and loaded into general register *rt*.

Operation :

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15\sim0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1\sim3} \| (pAddr_{2\sim0}$ xor $ReverseEndian^3)$
          $mem \leftarrow LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2\sim0}$ *xor* $BigEndianCPU^3$
          $GPR[rt] \leftarrow 0^{24} \| mem_{7+8*byte\sim8*byte}$
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15\sim0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1\sim3} \| (pAddr_{2\sim0}$ xor $ReverseEndian^3)$
          $mem \leftarrow LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2\sim0}$ *xor* $BigEndianCPU^3$
          $GPR[rt] \leftarrow 0^{56} \| mem_{7+8*byte\sim8*byte}$
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LD Load Doubleword LD

31 26	25 21	20 16	15 0
LD 110111	base	rt	offset
6	5	5	16

Format:

LD rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register base to form a virtual address. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded into general register *rt*.

If any of the three least-significant bits of the effective address are non-zero, an address error exception occurs.

Operation:

```
64    T:    vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15\sim0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

LDCz Load Doubleword To Coprocessor z LDCz

31 26	25 21	20 16	15 0
LDCz 1101xx*	base	rt	offset
6	5	5	16

Format:

LDCz rt, offset (base)

Description :

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The processor reads a double-word from the addressed memory location and makes the data available to coprocessor unit z. The manner in which each coprocessor uses he data is defined by the individual coprocessor specifications.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

This instruction is not valid for use with CP0.

This instruction is undefined when the least-significant bit of the *rt-field* is non-zero.

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.

LDCz Load Doubleword To Coprocessor z (continued) LDCz

Operation:

```
32   T:   vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15\sim0}) + \text{GPR[base]}$
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
          COPzLD(rt, mem)
64   T:   vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15\sim0}) + \text{GPR[base]}$
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
          COPzLD (rt, mem)
```

Exceptions:

TLB refill exception

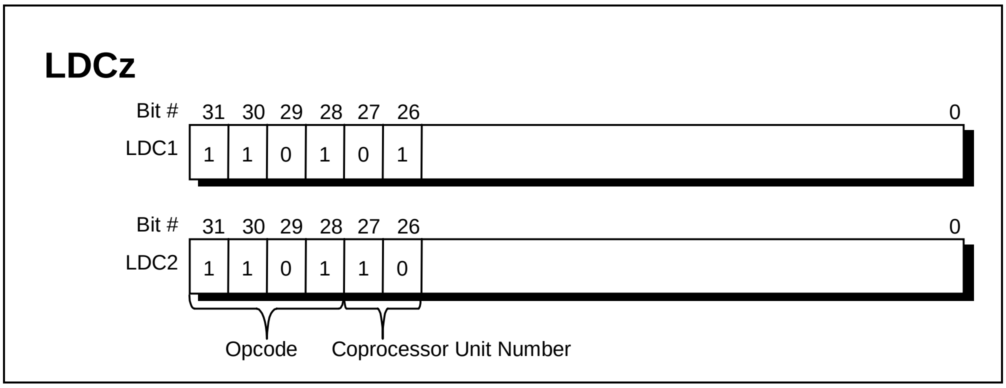
TLB invalid exception

Bus error exception

Address error exception

Coprocessor unusable exception

Opcode Bit Encoding:



LDL Load Doubleword Left LDL

31 26	25 21	20 16	15 0
LDL 011010	base	rt	offset
6	5	5	16

Format:

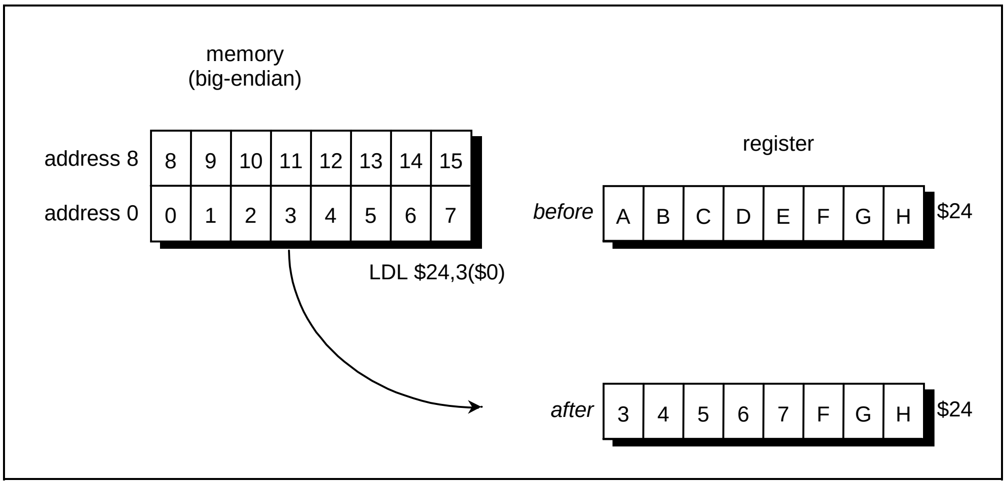
LDL rt, offset (base)

Description:

This instruction can be used in combination with the LDR instruction to load a register with eight consecutive bytes from memory, when the bytes cross a boundary between two doublewords. LDL loads the left portion of the register from the appropriate part of the high-order doubleword; LDR loads the right portion of the register from the appropriate part of the low-order doubleword.

The LDL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the doubleword in memory which contains the specified starting byte. From one to eight bytes will be loaded, depending on the starting byte specified.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it proceeds toward the low-order byte of the doubleword in memory and the low-order byte of the register, loading bytes from memory into the register until it reaches the low-order byte of the doubleword in memory. The least-significant (right-most) byte(s) of the register will not be changed.



LDL Load Doubleword Left (continued) LDL

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LDL (or LDR) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

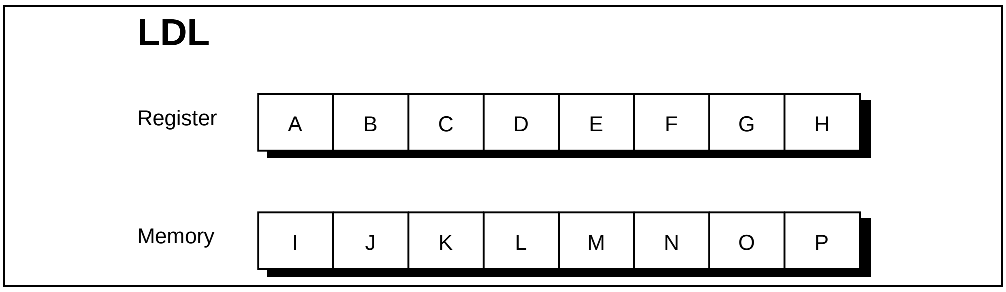
Operation:

```
64   T:   vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15\sim0})$ + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← $\text{pAddr}_{\text{PSIZE-1}\sim3} \,||\, (\text{pAddr}_{2\sim0}$ xor $\text{ReverseEncian}^{3})$
          if BigEndianMem = 0 then
          pAddr ← $\text{pAddr}_{\text{PSIZE-1}\sim3} \,||\, 0^{3}$
          endif
          byte ← $\text{vAddr}_{2\sim0}$ xor $\text{BigEndianCPU}^{3}$
          mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
          GPR[rt] ← $\text{mem}_{7+8*\text{byte}\sim0} \,||\, \text{GPR[rt]}_{55-8*\text{byte}\sim0}$
```

Note: It is also the same operation in the 32 bit kernel mode.

LDL — Load Doubleword Left (continued) — LDL

Given a doubleword in a register and a doubleword in memory, the operation of LDL us as follows:



$vAddr_{2\sim0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	type	offset		Destination	type	offset	
			LEM	BEM			LEM	BEM
0	P B C D E F G H	0	0	7	I J K L M N O P	7	0	0
1	O P C D E F G H	1	0	6	J K L M N O P H	6	0	1
2	N O P D E F G H	2	0	5	K L M N O P G H	5	0	2
3	M N O P E F G H	3	0	4	L M N O P F G H	4	0	3
4	L M N O P F G H	4	0	3	M N O P E F G H	3	0	4
5	K L M N O P G H	5	0	2	N O P D E F G H	2	0	5
6	J K L M N O P H	6	0	1	O P C D E F G H	1	0	6
7	I J K L M N O P	7	0	0	P B C D E F G H	0	0	7

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	AccessType sent to memory
Offset	$Addr_{2\sim0}$ sent to memory

Exceptions:

TLB refill exception

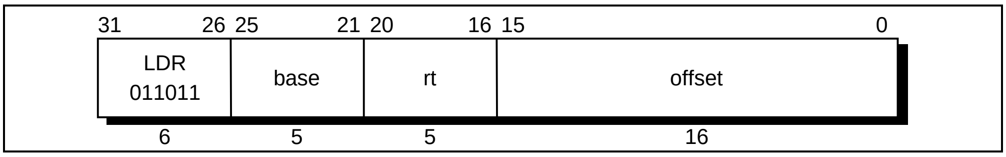
TLB invalid exception

Bus error exception

Address error exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

LDR Load Doubleword Right LDR



Format:

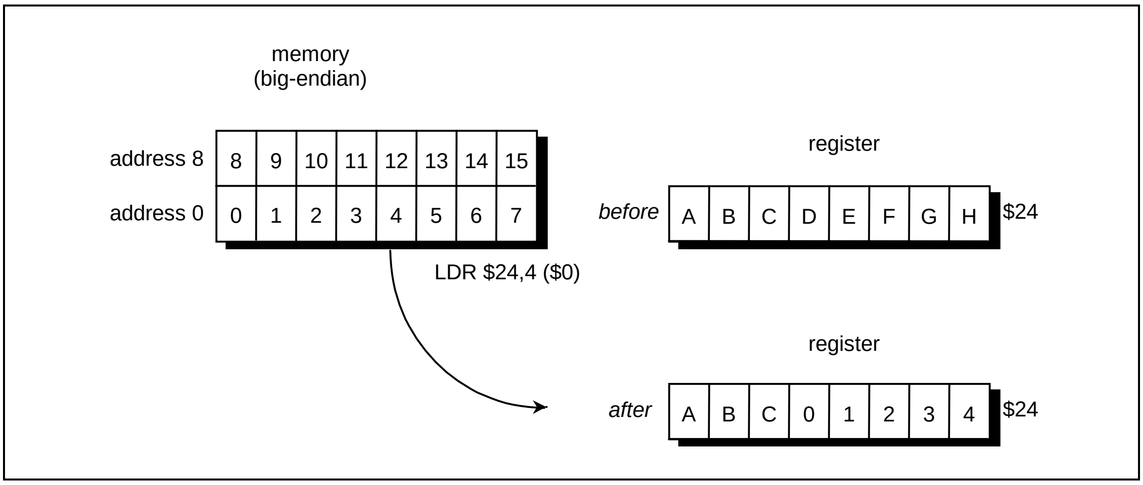
LDR rt, offset (base)

Description:

This instruction can be used in combination with the LDL instruction to load a register with eight consecutive bytes from memory, when the bytes cross a boundary between two doublewords. LDR loads the right portion of the register from the appropriate part of the low-order doubleword; LDL loads the left portion of the register from the appropriate part of the high-order doubleword.

The LDR instruction adds its sign-extended 16-bit *offset* to the con-tents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the doubleword in memory which contains the specified starting byte. From one to eight bytes will be loaded, depending on the starting byte specified.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it proceeds toward the high-order byte of the doubleword in memory and the high-order byte of the register, loading bytes from memory into the register until it reaches the high-order byte of the doubleword in memory. The most significant (left-most) byte (s) of the register will not be changed.



LDR Load Doubleword Right (continued) LDR

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LDR (or LDL) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

Operation:

```
64   T:   vAddr ← $((\text{offset}_{15})^{48} \mid\mid \text{offset}_{15\sim0}) + \text{GPR[base]}$
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← $\text{pAddr}_{\text{PSIZE-1}\sim3} \mid\mid (\text{pAddr}_{2\sim0}$ xor $\text{ReverseEncian}^{3})$
          if BigEndianMem = 1 then
          pAddr ← $\text{pAddr}_{31\sim3} \mid\mid 0^{3}$
          endif
          byte ← $\text{vAddr}_{2\sim0}$ *xor* $\text{BigEndianCPU}^{3}$
          mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
          GPR[rt] ← $\text{GPR[rt]}_{63\sim64-8*\text{byte}} \mid\mid \text{mem}_{63\sim8*\text{byte}}$
```

Note: It is also the same operation in the 32 bit kernel mode.

LDR Load Doubleword Right (continued) LDR

Given a doubleword in a register and a doubleword in memory, the operation of LDR is as follows:

LDR

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2\sim0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	type	offset		Destination	type	offset	
			LEM	BEM			LEM	BEM
0	I J K L M N O P	7	0	0	A B C D E F G I	0	7	0
1	A I J K L M N O	6	1	0	A B C D E F I J	1	6	0
2	A B I J K L M N	5	2	0	A B C D E I J K	2	5	0
3	A B C I J K L M	4	3	0	A B C D I J K L	3	4	0
4	A B C D I J K L	3	4	0	A B C I J K L M	4	3	0
5	A B C D E I J K	2	5	0	A B I J K L M N	5	2	0
6	A B C D E F I J	1	6	0	A I J K L M N O	6	1	0
7	A B C D E F G I	0	7	0	I J K L M N O P	7	0	0

LEM BigEndianMem = 0

BEM BigEndianMem = 1

Type AccessType sent to memory

Offset Addr$_{2\sim0}$ sent to memory

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

LH Load Halfword LH

31 26	25 21	20 16	15 0
LH 100001	base	rt	offset
6	5	5	16

Format:

LH rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the halfword at the memory location specified by the effective address are sign-extended and loaded into general register *rt*.

If the least-significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

```
32   T:   vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15\sim0}$) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ xor (ReverseEndian || 0))
          mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
          byte ← vAddr$_{2\sim0}$ *xor* (BigEndianCPU$^{2}$ || 0)
          GPR[rt] ← (mem$_{15+8*byte}$)$^{16}$ || mem$_{15+8*byte\sim8*byte}$
64   T:   vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15\sim0}$) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ xor (ReverseEndian || 0))
          mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
          byte ← vAddr$_{2\sim0}$ *xor* (BigEndianCPU$^{2}$ || 0)
          GPR[rt] ← (mem$_{15+8*byte}$)$^{16}$ || mem$_{15+8*byte\sim8*byte}$
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LHU Load Halfword Unsigned LHU

31 26	25 21	20 16	15 0
LHU 100101	base	rt	offset
6	5	5	16

Format:

LHU rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the halfword at the memory location specified by the effective address are zero-extended and loaded into general register *rt*.

If the least-significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

```
32   T:   vAddr ← $((offset_{15})^{16} || offset_{15\sim0}) + GPR[base]$
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← $pAddr_{PSIZE-1\sim3} || (pAddr_{2\sim0}$ xor $(ReverseEndian^2 || 0))$
          mem ← LoadMemory (uncached, HALFWORD, pAddr,vAddr, DATA)
          byte ← $vAddr_{2\sim0}$ *xor* $(BigEndianCPU^2 || 0)$
          GPR[rt] ← $0^{16} || mem_{15+8*byte\sim8*byte}$
64   T:   vAddr ← $((offset_{15})^{48} || offset_{15\sim0}) + GPR[base]$
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← $pAddr_{PSIZE-1\sim3} || (pAddr_{2\sim0}$ xor $(ReverseEndian^2 || 0))$
          mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
          byte ← $vAddr_{2\sim0}$ *xor* $(BigEndianCPU^2 || 0)$
          GPR[rt] ← $0^{48} || mem_{15+8*byte\sim8*byte}$
```

Exceptions:

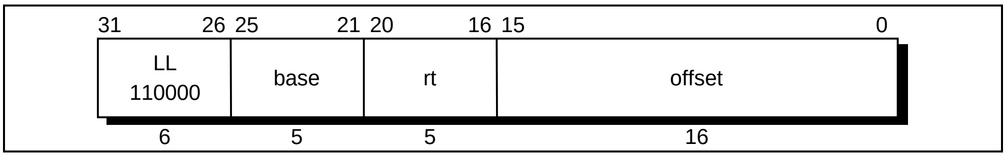
TLB refill exception

TLB invalid exception

Bus Error exception

Address error exception

LL Load Linked LL



Format:

LL rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register *rt*. In 64-bit mode, the loaded word is sign-extended.

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LLD Load Linke Doubleword LLD

31 26	25 21	20 16	15 0
LLD 110100	base	rt	offset
6	5	5	16

Format:

LLD rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the doubleword at the memory location specified by the effective address are loaded into general register *rt*.

The processor begins checking the accessed doubleword for modification by other processors and devices.

Load Linked Doubleword and Store Conditional Doubleword can be used to atomically update memory locations:

```
L1:
        LLD     T1, (T0)
        ADD     T2, T1, 1
        SCD     T2, (T0)
        BEQ     T2, 0, L1
        NOP
```

This atomically increments the word addressed by T0. Changing the ADD to an OR changes this to an atomic bit set.

LLD Load Linked Doubleword (continued) LLD

The operation of LLD is undefined if the addressed location is uncached and, for synchronization between multiple processors, the operation of LLD is undefined if the addressed location is noncoherent.

A cache miss that occurs between LLD and SCD may cause SCD to fail, so no load or store instruction should occur between LLD and SCD. Exceptions also cause SCD to fail, so persistent exceptions must be avoided.

This instruction is available in User mode, and it is not necessary for CP0 to be enabled.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

Operation:

```
64   T:   vAddr ← $((\text{offset}_{15})^{48} \| \text{offset}_{15\sim0}) + \text{GPR[base]}$
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          mem←LoadMemory (uncached,DOUBLE WORD,pAddr,vAddr,DATA)
          GPR[rt] ← mem
          LLbit ← 1
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

LUI Load Upper Immediate LUI

31 26	25 21	20 16	15 0
LUI 001111	0 00000	rt	immediate
6	5	5	16

Format:

LUI rt, immediate

Description:

The 16-bit *immediate* is shifted left 16 bits and concatenated to 16 bits of zeros. The result is placed into general register *rt*. In 64-bit mode, the loaded word is sign-extended.

Operation:

```
32   T:   $GPR[rt] \leftarrow immediate \,||\, 0^{16}$
64   T:   $GPR[rs] \leftarrow (immediate_{15})^{32} \,||\, immediate \,||\, 0^{16}$
```

Exceptions:

None

LW Load Word LW

31 26	25 21	20 16	15 0
LW 100011	base	rt	offset
6	5	5	16

Format:

LW rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register *rt*. In 64-bit mode, the loaded word is sign-extended.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

Operation:

```
32  T:  vAddr ← $((\text{offset}_{15})^{16} \| \text{offset}_{15\sim0}) + \text{GPR[base]}$
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         pAddr ← $\text{pAddr}_{\text{PSIZE-1}\sim3} \| (\text{pAddr}_{2\sim0}$ xor (ReverseEndian $\| 0^2$)
         mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
         byte ← $\text{vAddr}_{2\sim0}$ *xor* (BigEndianCPU $\| 0^2$)
         GPR[rt] ← $\text{mem}_{31+8*\text{byte}\sim8*\text{byte}}$
64  T:  vAddr ← $((\text{offset}_{15})^{48} \| \text{offset}_{15\sim0}) + \text{GPR[base]}$
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         pAddr ← $\text{pAddr}_{\text{PSIZE-1}\sim3} \| (\text{pAddr}_{2\sim0}$ xor (ReverseEndian $\| 0^2$)
         mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
         byte ← $\text{vAddr}_{2\sim0}$ *xor* (BigEndianCPU $\| 0^2$)
         GPR[rt] ← $(\text{mem}_{31+8*\text{byte}})^{32} \| \text{mem}_{31+8*\text{byte}\sim8*\text{byte}}$
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LWCz Load Word To Coprocessor z LWCz

31 26	25 21	20 16	15 0
LWXz 1100xx*	base	rt	offset
6	5	5	16

Format:

LWCz rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The processor reads a word from the addressed memory location, and makes the data available to coprocessor unit z. The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

This instruction is not valid for use with CP0.

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.

LWCz Load Word To Coprocessor z (continued) LWCz

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \mid\mid offset_{15\sim0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1\sim3} \mid\mid (pAddr_{2\sim0}\ xor\ (ReverseEndian \mid\mid 0^{2})$
          $mem \leftarrow LoadMemory\ (uncached, WORD, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2\sim0}\ xor\ (BigEndianCPU \mid\mid 0^{2})$
          COPzLW (byte, rt, mem)
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \mid\mid offset_{15\sim0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1\sim3} \mid\mid (pAddr_{2\sim0}\ xor\ (ReverseEndian \mid\mid 0^{2})$
          $mem \leftarrow LoadMemory\ (uncached, WORD, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2\sim0}\ xor\ (BigEndianCPU \mid\mid 0^{2})$
          COPzLW(byte, rt, mem)
```

Exceptions:

TLB refill exception

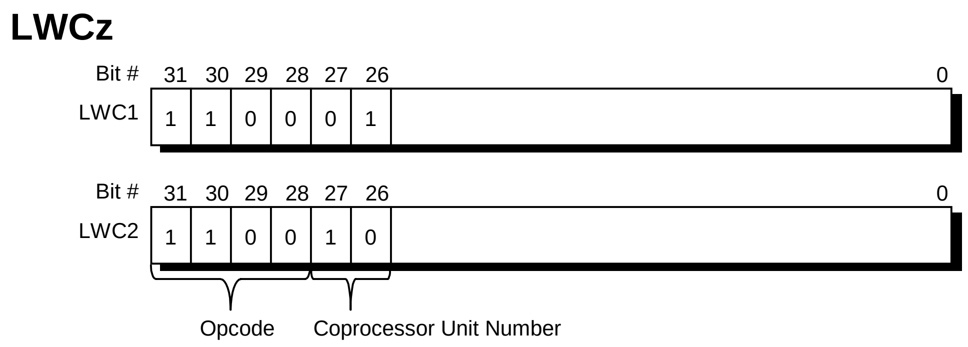
TLB invalid exception

Bus error exception

Address error exception

Coprocessor unusable exception

Opcode Bit Encoding:



LWCz

Bit #	31	30	29	28	27	26	0
LWC1	1	1	0	0	0	1	

Bit #	31	30	29	28	27	26	0
LWC2	1	1	0	0	1	0	

LWL Load Word Left LWL

31 26	25 21	20 16	15 0
LWL 100010	base	rt	offset
6	5	5	16

Format:

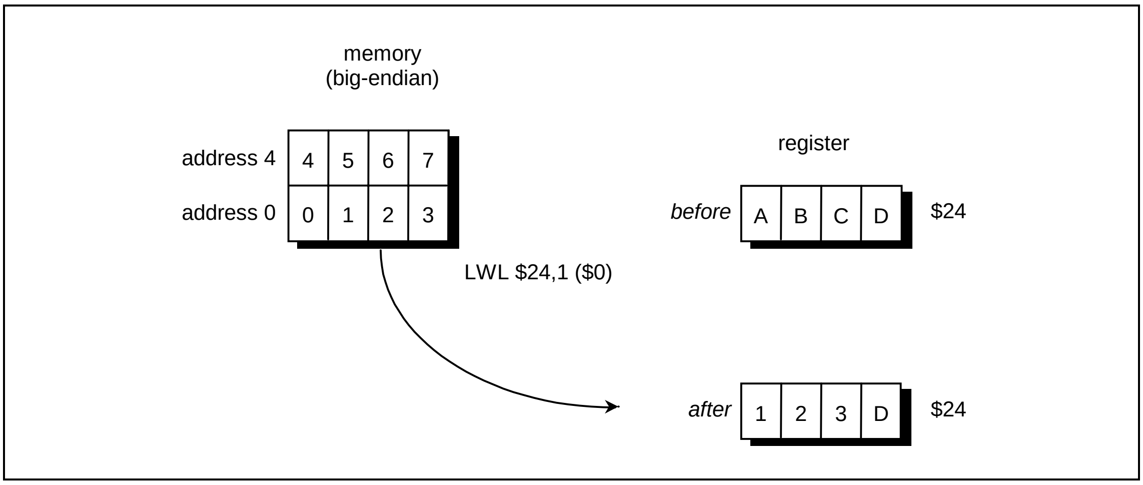
LWL rt, offset (base)

Description:

This instruction can be used in combination with the LWR instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words. LWL loads the left portion of the register from the appropriate part of the high-order word; LWR loads the right portion of the register from the appropriate part of the low-order word.

The LWL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified. In 64-bit mode, the loaded word is sign-extended.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it proceeds toward the low-order byte of the word in memory and the low-order byte of the register, loading bytes from memory into the register until it reaches the low-order byte of the word in memory. The least-significant (right-most) byte(s) of the register will not be changed.



LWL Load Word Left (continued) LWL

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWL (or LWR) instruction which also specifies register *rt*.

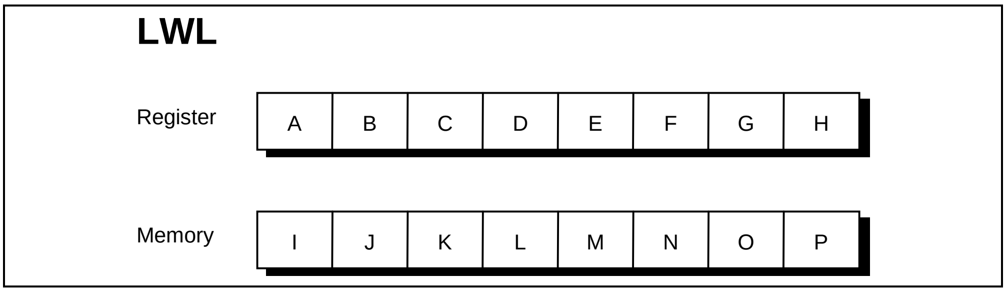
No address exceptions due to alignment are possible.

Operation:

```
32   T:     vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15\sim0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ xor (ReverseEncian$^{3}$)
            if BigEndianMem = 0 then
            pAddr ← pAddr$_{PSIZE-1\sim2}$ || 0$^{2}$
            endif
            byte ← vAddr$_{1\sim0}$ *xor* BigEndianCPU$^{2}$
            word ← vAddr$_{2}$ *xor* BigEndianCPU
            mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
            temp ← mem$_{32*word + 8*byte + 7\sim32*word}$ || GPR[rt]$_{23 - 8*byte\sim0}$
            GPR[rt] ← temp
64   T:     vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15\sim0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ xor (ReverseEncian$^{3}$)
            if BigEndianMem = 0 then
            pAddr ← pAddr$_{PSIZE-1\sim2}$ || 0$^{2}$
            endif
            byte ← vAddr$_{1\sim0}$ *xor* BigEndianCPU$^{2}$
            word ← vAddr$_{2}$ *xor* BigEndianCPU
            mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
            temp ← mem$_{32*word + 8*byte + 7\sim32*word}$ || GPR[rt]$_{23 - 8*byte\sim0}$
            GPR[rt] ← (temp$_{31}$)$^{32}$ || temp
```

LWL **Load Word Left (continued)** LWL

Given a doubleword in a register and a doubleword in memory, the operation of LWL is as follows:



$vAddr_{2\sim0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	type	offset		Destination	type	offset	
			LEM	BEM			LEM	BEM
0	S S S S P F G H	0	0	7	S S S S I J K L	3	4	0
1	S S S S O P G H	1	0	6	S S S S J K L H	2	4	1
2	S S S S N O P H	2	0	5	S S S S K L G H	1	4	2
3	S S S S M N O P	3	0	4	S S S S L F G H	0	4	3
4	S S S S L F G H	0	4	3	S S S S M N O P	3	0	4
5	S S S S K L G H	1	4	2	S S S S N O P H	2	0	5
6	S S S S J K L H	2	4	1	S S S S O P G H	1	0	6
7	S S S S I J K L	3	4	0	S S S S P F G H	0	0	7

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	AccessType (see Figure 2-2) sent to memory
Offset	$pAddr_{2\sim0}$ sent to memory
S	sign-extend of destination31

Exception:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LWR Load Word Right LWR

31 26	25 21	20 16	15 0
LWR 100110	base	rt	offset
6	5	5	16

Format:

LWR rt, offset (base)

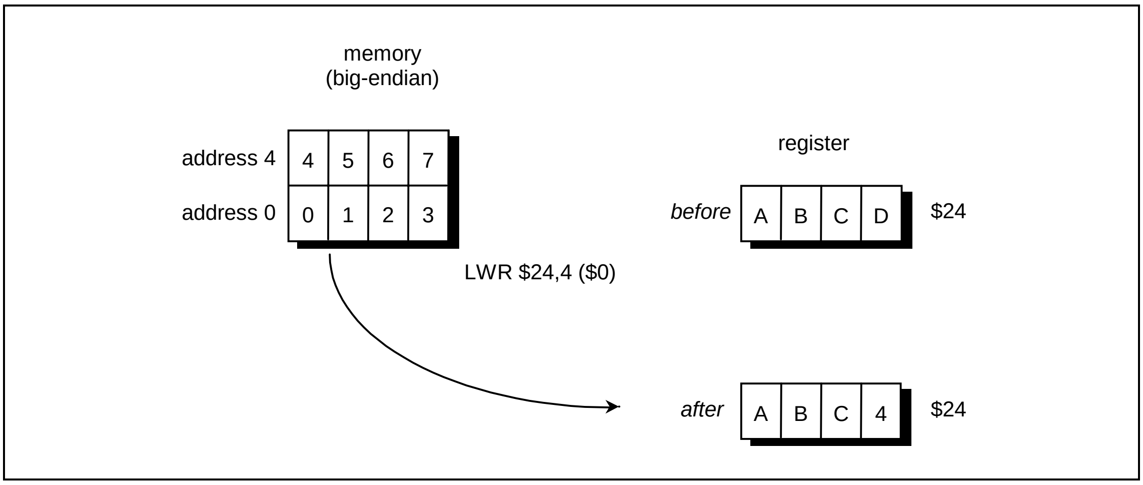
Description:

This instruction can be used in combination with the LWL instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words. LWR loads the right portion of the register from the appropriate part of the low-order word; LWL loads the left portion of the register from the appropriate part of the high-order word.

The LWR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified. In 64-bit mode, if bit 31 of the destination register is loaded, then the loaded word is sign-extended.

Conceptually, it starts at the specified byte in memory and loads. that byte into the low-order (right-most) byte of the register; then it proceeds toward the high-order byte of the word in memory and the high-order byte of the register, loading bytes from memory into the register until it reaches the high-order byte of the word in memory.

The most significant (left-most) byte(s) of the register will not be changed.



LWR Load Word Right (continued) LWR

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWR (or LWL) instruction which also specifies register *rt*.

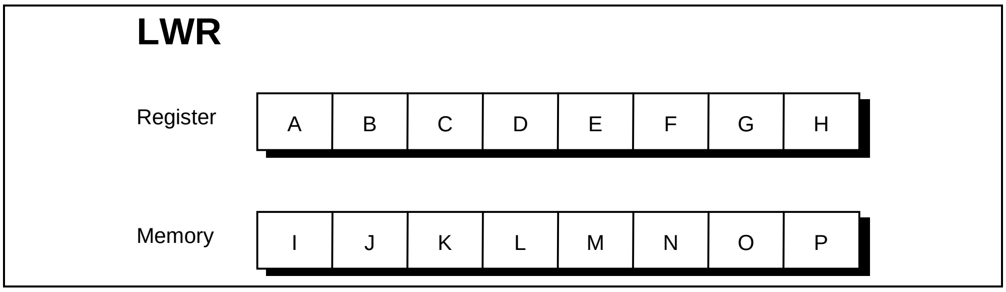
No address exceptions due to alignment are possible.

Operation:

```
32   T:   vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15\sim0}$) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ xor ReverseEndian$^{3}$)
          if BigEndianMem = 1 then
              pAddr ← pAddr$_{PSIZE-31\sim3}$ || 0$^{3}$
          endif
          byte ← vAddr$_{1\sim0}$ xor BigEndianCPU$^{2}$
          word ← vAddr$_{2}$ xor BigEndianCPU
          mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
          temp ← GPR[rt]$_{31\sim32-8*byte}$ || mem$_{31+32*word\sim32*word+8*byte}$
          GPR[rt] ← temp
64   T:   vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15\sim0}$) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ xor ReverseEndian$^{3}$)
          if BigEndianMem = 1 then
              pAddr ← pAddr$_{PSIZE-31\sim3}$ || 0$^{3}$
          endif
          byte ← vAddr$_{1\sim0}$ xor BigEndianCPU$^{2}$
          word ← vAddr$_{2}$ xor BigEndianCPU
          mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
          temp ← GPR[rt]$_{31\sim32-8*byte}$ || mem$_{31+32*word\sim32*word+8*byte}$
          GPR[rt] ← (temp$_{31}$)$^{32}$ || temp
```

LWR Load Word Right (continued) LWR

Given a word in a register and a word in memory, the operation of LWR is as follows:



$vAddr_{2\sim0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		Destination	type	offset	
			LEM	BEM			LEM	BEM
0	S S S S M N O P	0	0	4	X X X X E F G I	0	7	0
1	X X X X E M N O	1	1	4	X X X X E F I J	1	6	0
2	X X X X E F M N	2	2	4	X X X X E I J K	2	5	0
3	X X X X E F G M	3	3	4	S S S S I J K L	3	4	0
4	S S S S I J K L	0	4	0	X X X X E F G M	4	3	4
5	X X X X E I J K	1	5	0	X X X X E F M N	5	2	4
6	X X X X E F I J	2	6	0	X X X X E M N O	6	1	4
7	X X X X E F G I	3	7	0	S S S S M N O P	7	0	4

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	AccessType (see Figure 2-2) sent to memory
Offset	$pAddr_{2\sim0}$ sent to memory
S	sign-extend of $destination_{31}$

Exceptions:

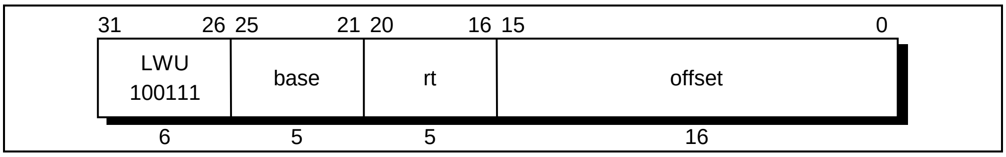
TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LWU Load Word Unsigned LWU



Format:

LWU rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register *rt*. The loaded word is zero-extended.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

Operation:

```
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15\sim0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1\sim3} \| (pAddr_{2\sim0}\ xor\ ReverseEndian \| 0^{2})$
          $mem \leftarrow LoadMemory\ (uncached, WORD, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2\sim0}\ \textit{xor}\ (BigEndianCPU \| 0^{2})$
          $GPR[rt] \leftarrow 0^{32} \| mem_{31+8*byte\sim8*byte}$
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisor mode)

MADD Multiply/Add MADD

31 26	25 21	20 16	15 6	5 0
MAC 011100	rs	rt	0 00 0000 0000	MADD 000000
6	5	5	10	6

31 26	25 21	20 16	15 11	10 6	5 0
MAC 011100	rs	rt	rd	0 00000	MADD 000000
6	5	5	5	11	

Format:

- MADD rs, rt
- MADD rd, rs, rt

Description:

Multiplies the contents of general registers rs and rt, treating both values as two's complement, and puts the double-word result in special registers HI and LO. An overview exception is never raised. The low-order word of the multiplication result is put in general register rd and in special register LO, whereas the high-order word of the reuslt is put in special register HI.

If rd is omitted in assembly language, 0 is used as the default value. To guarantee correct operation even if an interrupt occurs, neithe of the two instructions following MADD should be DIV or DIVU instructions which modify the HI and LO register contents.

Operation:

```
32, 64    T:    t ← (HI || LO) + GPR[rs]*GPR[rt]
                LO ← $t_{31\sim0}$
                HI ← $t_{63\sim32}$
                GPR[rd] ← $t_{31\sim0}$
```

Exception:

None

MADDU Multiply/Add Unsigned MADDU

31 26	25 21	20 16	15 6	5 0
MAC 011100	rs	rt	0 00 0000 0000	MADDU 000001
6	5	5	10	6

31 26	25 21	20 16	15 11	10 6	5 0
MAC 011100	rs	rt	rd	0 00000	MADDU 000001
6	5	5	5	11	

Format:

MADDU rs, rt

MADDU rd, rs, rt

Description:

Multiplies the contents of general registers rs and rt, treating both values as unsigned, and puts the double-word result in special registers HI and LO. An overview exception is never raised. The low-order word of the multiplication result is put in general register rd and in special register LO, whereas the high-order word of the reuslt is put in special register HI.

If rd is omitted in assembly language, 0 is used as the default value. To guarantee correct operation even if an interrupt occurs, neithe of the two instructions following MADDU should be DIV or DIVU instructions which modify the HI and LO register contents.

Operation:

```
32, 64    T:    $t \leftarrow (HI \,||\, LO) + (0 \,||\, GPR[rs]) + (0 \,||\, GPR[rt])$
                $LO \leftarrow t_{31\sim0}$
                $HI \leftarrow t_{63\sim32}$
                $GPR[rd] \leftarrow t_{31\sim0}$
```

Exception:

None

MFC0 **Move From System Control Coprocessor 0** MFC0

31 26	25 21	20 16	15 11	10 0
COP0 010000	MF 00000	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

MFC0 rt, rd

Description:

The contents of coprocessor register *rd* of the CP0 are loaded into general register *rt*.

May be used on both 32-bit and 64-bit CP0 registers.

Operation:

```
32   T:       data ← CPR[0,rd]
     T + 1:   GPR[rt] ← data
64   T:       data ← CPR[0,rd]
     T + 1:   GPR[rt] ← $(data_{31})^{32}$ || $data_{31\sim0}$
```

Exceptions:

Coprocessor unusable exception

MFCz Move From Coprocessor z MFCz

31 26	25 21	20 16	15 11	10 0
COPz 0100xx*	MF 00000	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

MFCz rt, rd

Description:

The contents of coprocessor register *rd* of coprocessor *z* are loaded into general register *rt*.

Execution of the instruction referencing coprocessor 3 causes a reserved instruction exception, not a coprocessor unusable exception.

Operation:

```
32  T:      data ← CPR[z,rd]
    T + 1:  GPR[rt] ← data
64  T:      if rd0 = 0
                data ← CPR[z,$rd_{4\sim1}$ || 0]$_{31\sim0}$
            else
                data ← CPR[z,$rd_{4\sim1}$ || 0]$_{63\sim32}$
            endif
    T + 1:  GPR[rt] ← $(data_{31})^{32}$||data
```

Exceptions:

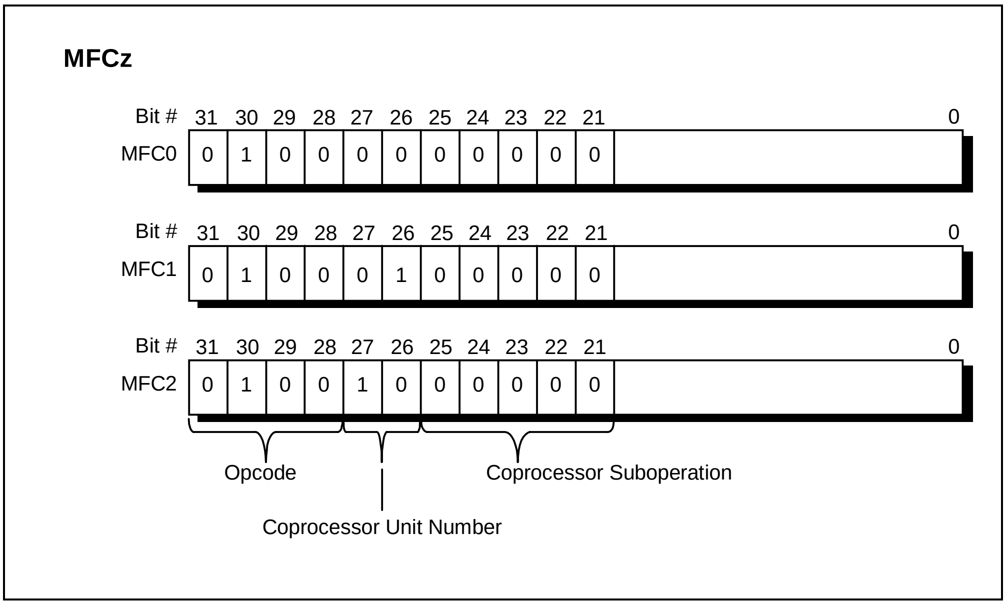
Coprocessor unusable exception

Reserved instruction exception (coprocessor 3)

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.

MFCz Move From Coprocessor z (continued) MFCz

Opcode Bit Encoding:



MFCz

Bit #	31	30	29	28	27	26	25	24	23	22	21	0
MFC0	0	1	0	0	0	0	0	0	0	0	0	
MFC1	0	1	0	0	0	1	0	0	0	0	0	
MFC2	0	1	0	0	1	0	0	0	0	0	0	

MFHI **Move From HI** MFHI

31 26	25 16	15 11	10 6	5 0
SPECIAL 000000	0 00 0000 0000	rd	0 00000	MFHI 010000
6	10	5	5	6

Format:

MFHI rd

Description:

The contents of special register *HI* are loaded into general register *rd*.

To ensure proper operation in the event of interruptions, the two instructions which follow a MFHI instruction may not be any of the instructions which modify the *HI* register: MULT, MULTU, DIV, DIVU, MTHI, DMULT, DMULTU, DDIV, DDIVU, MADD, MADDU.

Operation:

```
32, 64   T:        GPR[rd] ← HI
```

Exceptions:

None

MFLO Move From Lo MFLO

31 26	25 16	15 11	10 6	5 0
SPECIAL 000000	0 00 0000 0000	rd	0 00000	MFLO 010010
6	10	5	5	6

Format:

MFLO rd

Description:

The contents of special register *LO* are loaded into general register *rd*.

To ensure proper operation in the event of interruptions, the two instructions which follow a MFLO instruction may not be any of the instructions which modify the *LO* register: MULT, MULTU, DIV, DIVU, MTLO, DMULT, DMULTU, DDIV, DDIVU, MADD, MADDU.

Operation:

```
32, 64  T:      GPR[rd] ← LO
```

Exceptions:

None

MTC0 Move To System Control Coprocessor 0 MTC0

31 26	25 21	20 16	15 11	10 0
COP0 010000	MT 00100	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

MTC0 rt, rd

Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of the CP0.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load, store instructions and TLB operations immediately prior to and after this instruction are undefined.

Operation:

```
32, 64  T:      data ← GPR[rt]
        T + 1:  CPR[0,rd] ← data
```

Exceptions:

Coprocessor unusable exception

MTCz Move To Coprocessor z MTCz

31 26	25 21	20 16	15 11	10 0
COPz 0100xx*	MT 00100	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

MTCz rt, rd

Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of coprocessor z. Execution of the instruction referencing coprocessor 3 causes a reserved instruction exception, not a coprocessor unusable exception.

Operation:

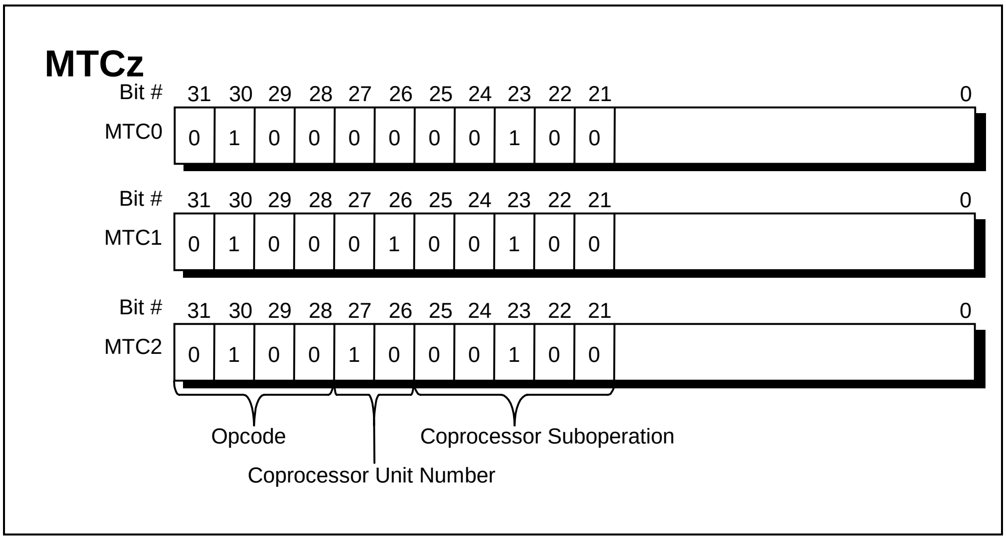
```
32  T:      data ← GPR[rt]
    T + 1:  CPR[z,rd] ← data
64  T:      data ← $GPR[rt]_{31\sim0}$
    T + 1:  if rd0 = 0
            $CPR[z,rd_{4\sim1} || 0] ← CPR[z,rd_{4\sim1} || 0]_{63\sim32} ||$ data
            else
            $CPR[z,rd_{4\sim1} || 0] ←$ data$||CPR[z, rd_{4\sim1} || 0]_{31\sim0}$
            endif
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception (coprocessor 3)

*Opcode Bit Encoding:



MTCz

Bit #	31	30	29	28	27	26	25	24	23	22	21	0
MTC0	0	1	0	0	0	0	0	0	1	0	0	
MTC1	0	1	0	0	0	1	0	0	1	0	0	
MTC2	0	1	0	0	1	0	0	0	1	0	0	

MTHI **Move To HI** MTHI

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000 0000 0000 0000	MTHI 010001
6	5	15	6

Format:

MTHI rs

Description:

The contents of general register *rs* are loaded into special register *HI*

If a MTHI operation is executed following a MULT, MULTU, DIV, DIVU, DMULT, DMULTU, DDIV, DDIVU, MADD, or MADDU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register *LO* are undefined.

Operation:

```
32, 64  T – 2:  HI ← undefined
        T – 1:  HI ← undefined
        T:      HI ← GPR[rs]
```

Exceptions:

None

MTLO Move To LO MTLO

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000 0000 0000 0000	MTLO 010011
6	5	15	6

Format:

MTLO rs

Description:

The contents of general register *rs* are loaded into special register *LO* If a MTLO operation is executed following a MULT, MULTU, DIV, DIVU, DMULT, DMULTU, DDIV, DDIVU, MADD, or MADDU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register *HI* are undefined.

Operation:

```
32, 64  T – 2:  LO ← undefined
        T – 1:  LO ← undefined
        T:      LO ← GPR[rs]
```

Exceptions:

None

MULT Multiply MULT

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 0 0000	MULT 011000
6	5	5	5	5	6

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	MULT 011000
6	5	5	10	6

Format:

MULT rs, rt

MULT rd, rs, rt

Description:

The contents of general registers *rs* and *rt* are multiplied, treating both operands as 32-bit 2's-complement values. No integer overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results is of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two other instructions.

Operation:

```
32   T – 2:   LO ← undefined
              HI ← undefined
     T – 1:   LO ← undefined
              HI ← undefined
     T:       t ← GPR[rs]* GPR[rt]
              LO ← $t_{31\sim0}$
              HI ← $t_{63\sim32}$
              GPR[rd] ← $t_{31\sim0}$
64   T – 2:   LO ← undefined
              HI ← undefined
     T – 1:   LO ← undefined
              HI ← undefined
     T:       t ← $GPR[rs]_{31\sim0}$* $GPR[rt]_{31\sim0}$
              LO ← $(t_{31})^{32}$ || $t_{31\sim0}$
              HI ← $(t_{63})^{32}$ || $t_{63\sim32}$
              GPR[rd] ← $(t_{31})^{32}$ || $t_{31\sim0}$
```

Exceptions:

None

MULTU Multiply Unsigned MULTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	MULTU 011001
6	5	5	10	6

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 0 0000	MULTU 011001
6	5	5	5	5	6

Format:

MULTU rs, rt

MULTU rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are multiplied, treating both operands as unsigned values. No overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two instructions.

Operation:

```
32   T – 2:   LO ← undefined
              HI ← undefined
     T – 1:   LO ← undefined
              HI ← undefined
     T:       t ← (0 || GPR[rs])* (0 || GPR[rt])
              LO ← $t_{31\sim0}$
              HI ← $t_{63\sim32}$
              GPR[rd] ← $t_{31\sim0}$
64   T – 2:   LO ← undefined
              HI ← undefined
     T – 1:   LO ← undefined
              HI ← undefined
     T:       t ← (0 || $GPR[rs]_{31\sim0}$)* (0 || $GPR[rt]_{31\sim0}$)
              LO ← $(t_{31})^{32}$ || $t_{31\sim0}$
              HI ← $(t_{63})^{32}$ || $t_{63\sim32}$
              GPR[rd] ← $(t_{31})^{32}$ || $t_{31\sim0}$
```

Exceptions:

None

NOR **Nor** NOR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	NOR 100111
6	5	5	5	5	6

Format:

NOR rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical NOR operation. The result is placed into general register *rd*.

Operation:

```
32, 64  T:      GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions:

None

OR **Or** OR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101
6	5	5	5	5	6

Format:

OR rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical OR operation. The result is placed into general register *rd*.

Operation:

```
32, 64   T:        GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None

ORI Or Immediate ORI

31 26	25 21	20 16	15 0
ORI 001101	rs	rt	immediate
6	5	5	16

Format:

ORI rt, rs, immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical OR operation. The result is placed into general register *rt*.

Operation:

32 T: $\text{GPR}[rt] \leftarrow \text{GPR}[rs]_{31\sim16} \,||\, (\text{immediate } \mathit{or} \text{ GPR}[rs]_{15\sim0})$

64 T: $\text{GPR}[rt] \leftarrow \text{GPR}[rs]_{63\sim16} \,||\, (\text{immediate } \mathit{or} \text{ GPR}[rs]_{15\sim0})$

Exceptions:

None

PREF Prefetch PREF

31 26	25 21	20 16	15 0
PREF 110011	base	hint	offset
6	5	5	16

Format :

PREF hint, offset (base)

Description :

PREF adds the 16-bit signed offset to the contents of GPR base to form an effective byte address. It advises that data at the effective address may be used in the near future.

If the hint field is 00000_2, this instruction prefetches a block of data from main memory into cache.

PREF is an advisory instruction. It may change the performance of the program. For all hint values and all effective addresses, it neither changes architecturally-visible state nor alters the meaning of the program.

PREF does not cause addressing-related exceptions. If it raises an exception condition, the exception conditions ignored. If an addressing-related exception is raised and ignored, no data will be prefetched, even if no data is prefetched in such a case, some action that is not architecturally-visible, such as writeback of a dirty cache line, might take place.

PREF will never generate a memory operation for a location with an uncached memory access type.

The defined hint values are shown in the table below. The TX49 only supports hint = 0. The hint table may be extended in future implementations.

hint field: Value

Value	Name	Data use and desired prefetch action
0	Load	Data is expected to be loaded (not modified). Fetch data as if for a load.
1-31	Reserved	Reserved

PREF Prefetch (continued) PREF

Programming Notes:

Prefetch can not prefetch data from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. It will not cause an exception to prefetch using an address pointer value before the validity of a pointer determined.

Operation :

```
32, 64    T:    vAddr ← GPR[base] = sign_extend (offset)
                (pAddr, uncached) ← Address Translation (vAddr, DATA, LOAD)
                Prefetch (uncached, pAddr, vAddr, DATA, hint)
```

Exception :

None

SB Store Byte SB

31 26	25 21	20 16	15 0
SB 101000	base	rt	offset
6	5	5	16

Format:

SB rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The least-significant byte of register *rt* is stored at the effective address.

Operation:

```
32   T:   vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15\sim0}$) + GPR[base]
          (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddrPSIZE$_{-1\sim3}$ || (pAddr$_{2\sim0}$ xor ReverseEndian$^3$)
          byte ← vAddr$_{2\sim0}$ xor BigEndianCPU$^3$
          data ← GPR[rt]$_{63-8*byte\sim0}$ || 0$^{8*byte}$
          StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
64   T:   vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15\sim0}$) + GPR[base]
          (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddrPSIZE$_{-1\sim3}$ || (pAddr$_{2\sim0}$ xor ReverseEndian$^3$)
          byte ← vAddr$_{2\sim0}$ xor BigEndianCPU$^3$
          data ← GPR[rt]$_{63-8*byte\sim0}$ || 0$^{8*byte}$
          StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

SC Store Conditional SC

31 26	25 21	20 16	15 0
SC 111000	base	rt	offset
6	5	5	16

Format:

SC rt, offset (base)

Description:

The 16-bit offset is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of general register *rt* are conditionally stored at the memory location specified by the effective address.

If an ERET instruction occurs between the Load Linked instruction and this store instruction, the store fails and is inhibited from taking place.

The success or failure of the store operation (as defined above) is indicated by the contents of general register *rt* after execution of the instruction. A successful store sets the contents of general register *rt* to1 ; an unsuccessful store sets it to 0.

The operation of Store Conditional is undefined when the address is different from the address used in the last Load Linked.

This instruction is available in User mode; it is not necessary for CP0 to be enabled.

If either of the two least-significant bits of the effective address is non-zero, an address error exception takes place.

SC Store Conditional (continued) SC

If this instruction should both fail and take an exception, the exception takes precedence.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15\sim0}) + GPR[base]$
          $(pAddr, unchached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddrPSIZE_{-1\sim3} \,||\, (pAddr_{2\sim0}\ \mathit{xor}\ ReverseEndian \,||\, 0^{2})$
          $data \leftarrow GPR[rt]_{63-8*byte\sim0} \,||\, 0^{8*byte}$
          if LLbit then
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
          endif
          $GPR[rt] \leftarrow 0^{31} \,||\, LLbit$
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15\sim0}) + GPR[base]$
          $(pAddr, unchached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddrPSIZE_{-1\sim3} \,||\, (pAddr_{2\sim0}\ \mathit{xor}\ ReverseEndian \,||\, 0^{2})$
          $data \leftarrow GPR[rt]_{63-8*byte\sim0} \,||\, 0^{8*byte}$
          if LLbit then
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
          endif
          $GPR[rt] \leftarrow 0^{63} \,||\, Llbit$
```

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

SCD Store Conditional Doubleword SCD

31 26	25 21	20 16	15 0
SCD 111100	base	rt	offset
6	5	5	16

Format:

SCD rt, offset (base)

Description:

The 16-bit offset is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of general register *rt* are conditionally stored at the memory location specified by the effective address.

If an ERET instruction occurs between the Load Linked Doubleword instruction and this store instruction, the store fails and is inhibited from taking place.

The success or failure of the store operation (as defined above) is indicated by the contents of general register *rt* after execution of the instruction. A successful store sets the contents of general register *rt* to1; an unsuccessful store sets it to 0.

The operation of Store Conditional Doubleword is undefined when the address is different from the address used in the last Load Linked Doubleword.

This instruction is available in User mode; it is not necessary for CP0 to be enabled.

If either of the three least-significant bits of the effective address is non-zero, an address error exception takes place.

If this instruction should both fail and take an exception, the exception takes precedence.

Operation:

```
64    T:    vAddr ← $((\text{offset}_{15})^{48} \| \text{offset}_{15\sim0}) + \text{GPR[base]}$
            (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
            data ← GPR[rt]
            If LLbit then
            StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
            endif
            GPR[rt] ← $0^{63} \|$ Llbit
```

Note: It is also the same operation in the 32 bit kernel mode.

SCD Store Conditional Doubleword (continued) SCD

Exceptions:

TLB refill exception

TLB invalid exception

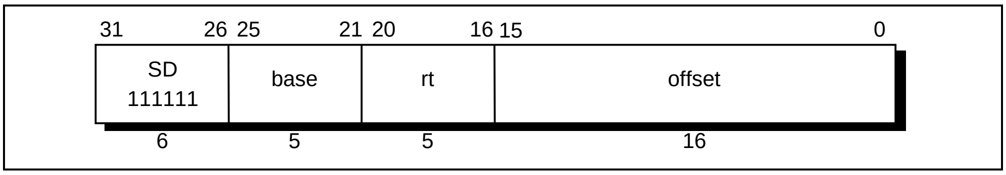
TLB modification exception

Bus error exception

Address error exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

SD Store Doubleword SD



Format:

SD rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of general register *rt* are stored at the memory location specified by the effective address.

If either of the three least-significant bits of the effective address are non-zero, an address error exception occurs.

Operation:

```
64    T:    vAddr ← $((offset_{15})^{48} || offset_{15\sim0}) + GPR[base]$
            (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
            data ← GPR[rt]
            StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Note: It is also the same operation, and the upper 32 bit is ignored when the virtual address is created in the 32 bit kernel mode.

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

SDBBP Store Debug Breakpoint SDBBP

31 26	25 6	5 0
SPECIAL 000000	Code	SDBBP 001110
6	20	6

Format:

SDBBP code

Description:

Raises a Debug Breakpoint exception, passing control to an exception handler. The code field can used for passing information to the exception handler, but the only way to have the code field retrived by the exception handler is to load the contents of the memory word containing this instruction using the DEPC register.

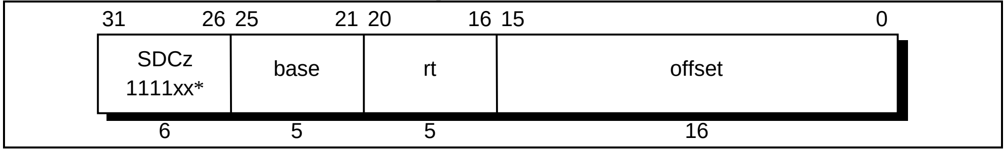
Operation:

```
32, 64    T:        Software DebugBreakpointException
```

Exception:

Debug Breakpoint exception

SDCz Store Doubleword From Coprocessor z SDCz



Format:

SDCz rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. Coprocessor unit *z* sources a doubleword, which the processor writes to the addressed memory location. The data to be stored is defined by individual coprocessor specifications.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

This instruction is not valid for use with CP0.

This instruction is undefined when the least-significant bit of the *rt-field* is non-zero.

*See the table, “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.

SDCz Store Doubleword From Coprocessor z (continued) SDCz

Operation:

```
32   T:   vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15\sim0}) + \text{GPR[base]}$
          (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
          data ← COPzSD (rt),
          StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
64   T:   vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15\sim0}) + \text{GPR[base]}$
          (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
          data ← COPzSD (rt),
          StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception

TLB invalid exception

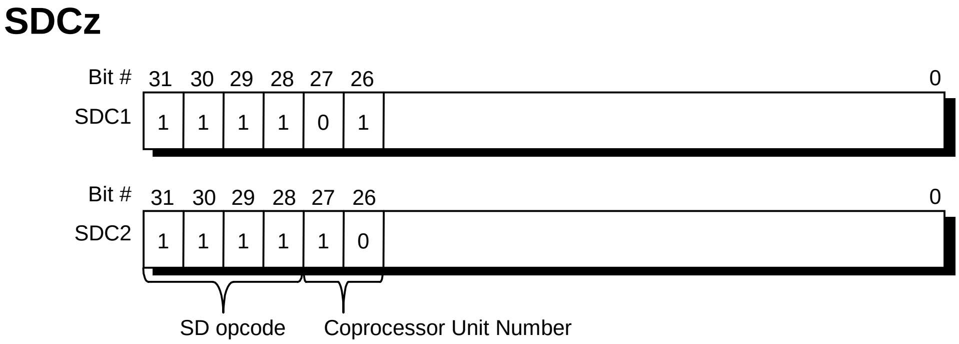
TLB modification exception

Bus error exception

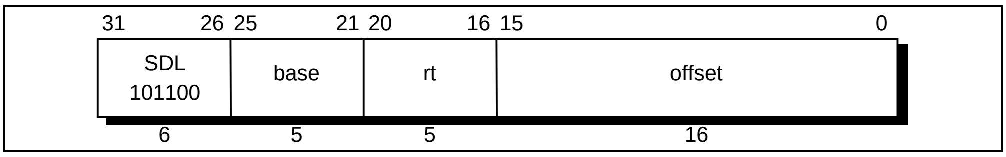
Address error exception

Coprocessor unusable exception

Opcode Bit Encoding:



SDL Store Doubleword Left SDL



Format:

SDL rt, offset (base)

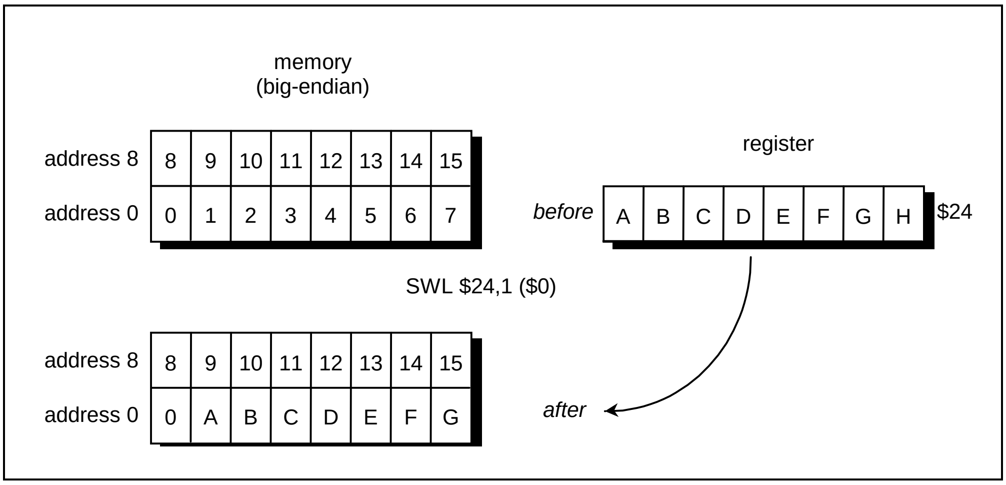
Description:

This instruction can be used with the SDR instruction to store the contents of a register into eight consecutive bytes of memory, when the bytes cross a boundary between two doublewords. SDL stores the left portion of the register into the appropriate part of the high-order doubleword of memory; SDR stores the right portion of the register into the appropriate part of the low-order doubleword.

The SDL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it proceeds toward the low-order byte of the register and the low-order byte of the word in memory, copying bytes from register to memory until it reaches the low-order byte of the word in memory.

No address exceptions due to alignment are possible.



SDL Store Doubleword Left (continued) SDL

This operation is only defined for the TX4300 operating in 64-bit mode nad 32-bit kernal mode.

Execution of this instruction in 32-bit user or supervisor mode causes a reserved instruction exception.

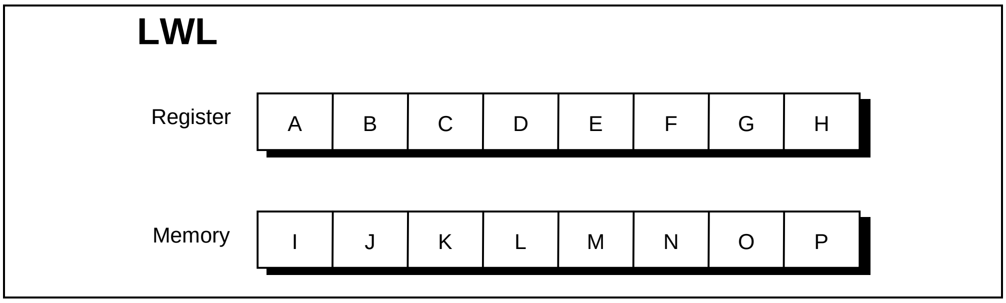
Operation:

```
64   T:   vAddr ← (($offset_{15}$)$^{48}$ || $offset_{15\sim0}$) + GPR[base]
          (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
          pAddr ← $pAddrPSIZE_{-1\sim3}$ || ($pAddr_{2\sim0}$ xor $ReverseEndian^3$)
          If BigEndianMem = 0 then
                pAddr ← $pAddr_{31\sim3}$ || $0^3$
          endif
          byte ← $vAddr_{2\sim0}$ xor $BigEndianCPU^3$
          data ← $0^{56-8*byte}$ || $GPR[rt]_{63\sim56-8*byte}$
          StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)
```

Note: It is also the same operation, and the upper 32 bit is ignored when the virtual address is created in the 32 bit kernel mode.

SDL Store Doubleword Left (continued) SDL

Given a doubleword in a register and a doubleword in memory, the operation of SWL is as follows:



LWL

Register A B C D E F G H

Memory I J K L M N O P

vAddr$_{2\sim0}$	BigEndianCPU = 0											BigEndianCPU = 1										
	destination								type	offset		destination								type	offset	
										LEM	BEM										LEM	BEM
0	I	J	K	L	M	N	O	A		0		A							H	7	0	0
1	I	J	K	L	M	N														6	0	1
2	I	J	K	L	M	A	B	C	2	0	5	I	J	A	B	C	D	E	F	5	0	2
3	I	J	K	L	A	B	C	D	3	0	4	I	J	K	A	B	C	D	E	4	0	3
4	I	J	K	A	B	C	D	E	4	0	3	I	J	K	L	A	B	C	D	3	0	4
5	I	J	A	B	C	D	E	F	5	0	2	I	J	K	L	M	A	B	C	2	0	5
6	I	A	B	C	D	E	F	G	6	0	1	I	J	K	L	M	N	A	B	1	0	6
7	A	B	C	D	E	F	G	H	7	0	0	I	J	K	L	M	N	O	A	0	0	7

LEM BigEndianMem = 0

BEM BigEndianMem = 1

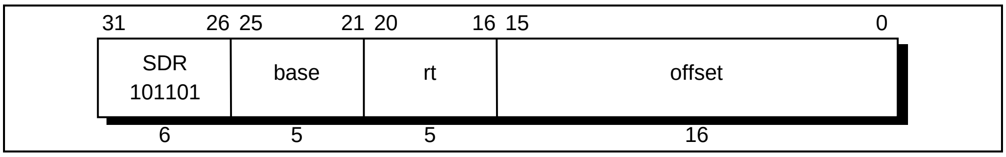
Type Access Type (see Figure 2-2) sent to memory

Offset pAddr$_{2\sim0}$ sent to memory

Exceptions:

- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
- Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

SDR Store Doubleword Right SDR



Format:

SDR rt, offset (base)

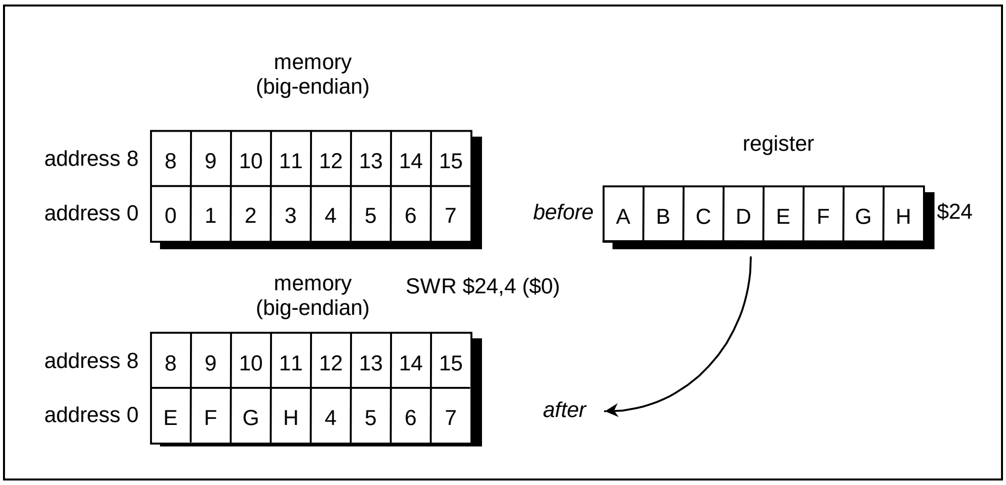
Description:

This instruction can be used with the SDL instruction to store the contents of a register into eight consecutive bytes of memory, when the bytes cross a boundary between two doublewords. SDR stores the right portion of the register into the appropriate part of the low-order doubleword; SDL stores the left portion of the register into the appropriate part of the low-order doubleword of memory.

The SDR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to eight bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then it proceeds toward the high-order byte of the register and the high-order byte of the word in memory, copying bytes from register to memory until it reaches the high-order byte of the word in memory.

No address exceptions due to alignment are possible.



SDR Store Doubleword Right (continued) SDR

This operation is only defined for the TX4300 operating in 64-bit mode and 32-bit kernal mode.

Execution of this instruction in 32-bit user or supervisor mode causes a reserved instruction exception

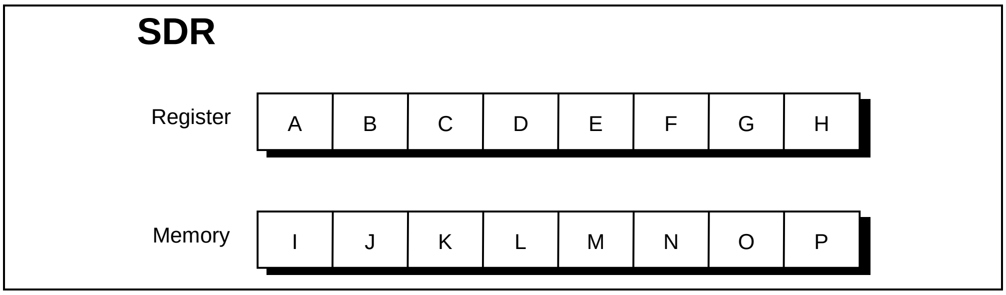
Operation:

```
64    T:    vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15\sim0}$) + GPR[base]
            (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE$_{-1\sim3}$ || (pAddr$_{2\sim0}$ xor ReverseEndian$^{3}$)
            if BigEndianMem = 0 then
                    pAddr ← pAddrPSIZE$_{-31\sim3}$ || 0$^{3}$
            endif
            byte ← vAddr$_{1\sim0}$ xor BigEndianCPU$^{3}$
            data ← GPR[rt]$_{63-8*byte\sim0}$ || 0$^{8*byte}$
            StoreMemory (uncached, DOUBLEWORD-byte, data, pAddr, vAddr,
```

Note: It is also the same operation, and the upper 32 bit is ignored when the virtual address is created in the 32 bit kernel mode.

SDR Store Doubleword Right (continued) SDR

Given a doubleword in a register and a doubleword in memory, the operation of SDR is as follows:



SDR

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2\sim0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	A B C D E F G H	7	0	0	H J K L M N O P	0	7	0
1	B C D E F G H P	6	1	0	G H K L M N O P	1	6	0
2	C D E F G H O P	5	2	0	F G H L M N O P	2	5	0
3	D E F G H N O P	4	3	0	E F G H M N O P	3	4	0
4	E F G H M N O P	3	4	0	D E F G H N O P	4	3	0
5	F G H L M N O P	2	5	0	C D E F G H O P	5	2	0
6	G H K L M N O P	1	6	0	B C D E F G H P	6	1	0
7	H J K L M N O P	0	7	0	A B C D E F G H	7	0	0

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	Access Type (see Figure 2-2) sent to memory
Offset	pAddr$_{2\sim0}$ sent to memory

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisior mode)

SH **Store Halfword** SH

31 2	25 21	20 16	15 0
SH 101001	base	rt	offset
6	5	5	16

Format:

SH rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address. The least-significant halfword of register *rt* is stored at the effective address. If the least-significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

```
32   T:   vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15\sim0}) + \text{GPR[base]}$
          (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
          pAddr ← $\text{pAddr}_{\text{PSIZE-1}\sim3} \,||\, (\text{pAddr}_{2\sim0}\ \textit{xor}\ (\text{ReverseEndian}^2 \,||\, 0))$
          byte ← $\text{vAddr}_{2\sim0}\ \textit{xor}\ (\text{BigEndianCPU}^2 \,||\, 0)$
          data ← $\text{GPR[rt]}_{63\text{-}8*\text{byte}\sim0} \,||\, 0^{8*\text{byte}}$
          StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)
64   T:   vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15\sim0}) + \text{GPR[base]}$
          (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
          pAddr ← $\text{pAddr}_{\text{PSIZE-1}\sim3} \,||\, (\text{pAddr}_{2\sim0}\ \textit{xor}\ (\text{ReverseEndian}^2 \,||\, 0))$
          byte ← $\text{vAddr}_{2\sim0}\ \textit{xor}\ (\text{BigEndianCPU}^2 \,||\, 0)$
          data ← $\text{GPR[rt]}_{63\text{-}8*\text{byte}\sim0} \,||\, 0^{8*\text{byte}}$
          StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

SLL **Shift Left Logical** SLL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SLL 000000
6	5	5	5	5	6

Format:

SLL rd, rt, sa

Description:

The contents of general register *rt* are shifted left by sa bits, inserting zeros into the low-order bits. The result is placed in register *rd*. In 64-bit mode, the 32-bit result is sign extended when placed in the destination register. It is sign-extended for all shift amounts, including zero; SLL with a zero shift amount truncates a 64-bit value to 32-bits and sign extends this 32-bit value. SLL, unlike nearly all other word operations, does not repuire and operand to be a properly sign-extended word value to produce a valid sign-extended word result.

Note: SLL with a shift amount of zero may be treated as a NOP by some assemblers at some optimization levels. If using SLL with zero shift to truncate 64-bit values, check the assembler being used.

Operation:

```
32  T:    $GPR[rd] \leftarrow GPR[rt]_{31-sa\sim0} \,||\, 0^{sa}$
64  T:    $s \leftarrow 0 \,||\, sa$
          $temp \leftarrow GPR[rt]_{31-s\sim0} \,||\, 0^{s}$
          $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$
```

Exceptions:

None

SLLV Shift Left Logical Variable SLLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 rs	rt	rd	0 00000	SLLV 000100
6	5	5	5	5	6

Format:

SLLV rd, rt, rs

Description:

The contents of general register *rt* are shifted left by the number of bits specified by the low-order five bits contained as contents of general register *rs*, inserting zeros into the low-order bits. The result is placed in register *rd*. In 64-bit mode, the 32-bit result is sign extended when placed in the destination register. It is sign-extended for all shift amounts, including zero; SLLV with a zero shift amount truncates a 64-bit value to 32-bits and sign extends this 32-bit value. SLLV, unlike nearly all other word operations, does not require the operand to be a properly sign-extended word value to produce a valid sign-extended word result.

Note: SLLV with a shift amount of zero may be treated as a NOP by some assemblers at some optimization levels. If using SLLV with zero shift to truncate 64-bit values, check the assembler being used.

Operation :

```
32    T:    $s \leftarrow GP[rs]_{4\sim0}$
            $GPR[rd] \leftarrow GPR[rt]_{(31-s)\sim0} \,||\, 0^{s}$
64    T:    $s \leftarrow 0 \,||\, GP[rs]_{4\sim0}$
            $temp \leftarrow GPR[rt]_{(31-s)\sim0} \,||\, 0^{s}$
            $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$
```

Exceptions:

None

SLT Set On Less Than SLT

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010
6	5	5	5	5	6

Format:

SLT rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register rs are less than the contents of general register *rt*, the result is set to one, otherwise the result is set to zero.

The result is placed into general register *rd*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32    T:    if GPR[rs] < GPR[rt] then
            GPR[rd] ← $0^{31}$ || 1
            else
            GPR[rd] ← $0^{32}$
            endif
64    T:    if GPR[rs] < GPR[rt] then
            GPR[rd] ← $0^{63}$ || 1
            else
            GPR[rd] ← $0^{64}$
            endif
```

Exceptions:

None

SLTI Set On Less Than Immediate SLTI

31 26	25 21	20 16	15 0
SLTI 001010	rs	rt	immediate
6	5	5	16

Format:

SLTI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general register *rs*. Considering both quantities as signed integers, if *rs* is less than the sign-extended immediate, the result is set to one, otherwise the result is set to zero. The result is placed into general register *rt*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32    T:    if GPR[rs] < $(\text{immediate}_{15})^{16} || \text{immediate}_{15\sim0}$ then
                GPR[rt] ← $0^{31} || 1$
            else
                GPR[rt] ← $0^{32}$
            endif
64    T:    if GPR[rs] < $(\text{immediate}_{15})^{48} || \text{immediate}_{15\sim0}$ then
                GPR[rt] ← $0^{63} || 1$
            else
                GPR[rt] ← $0^{64}$
            endif
```

Exceptions:

None

SLTIU Set On Less Than Immediate Unsigned SLTIU

31 26	25 21	20 16	15 0
SLTIU 001011	rs	rt	immediate
6	5	5	16

Format:

SLTIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general register *rs*. Considering both quantities as unsigned integers, if *rs* is less than the sign-extended immediate, the result is set to one, otherwise the result is set to zero. The result is placed into general register rt.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32    T:    if $(0 \| GPR[rs]) < (immediate_{15})^{16} \| immediate_{15-0}$ then
                $GPR[rt] \leftarrow 0^{31} \| 1$
            else
                $GPR[rt] \leftarrow 0^{32}$
            endif
64    T:    if $(0 \| GPR[rs]) < (immediate_{15})^{48} \| immediate_{15-0}$ then
                $GPR[rt] \leftarrow 0^{63} \| 1$
            else
                $GPR[rt] \leftarrow 0^{64}$
            endif
```

Exceptions:

None

SLTU Set On Less Than Unsigned SLTU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLTU 101011
6	5	5	5	5	6

Format:

SLTU rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are less than the contents of general register *rt*, the result is set to one, otherwise the result is set to zero.

The result is placed into general register *rd*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32   T:   if (0 || GPR[rs]) < 0 || GPR[rt] then
              GPR[rd] ← $0^{31}$ || 1
          else
              GPR[rd] ← $0^{32}$
          endif
64   T:   if (0 || GPR[rs]) < 0 || GPR[rt] then
              GPR[rd] ← $0^{63}$ || 1
          else
              GPR[rd] ← $0^{64}$
          endif
```

Exceptions:

None

SRA Shift Right Arithmetic SRA

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRA 000011
6	5	5	5	5	6

Format:

SRA rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, sign-extending the high-order bits. The result is placed in register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation :

```
32   T:   $GPR[rd] \leftarrow (GPR[rt]_{31})^{sa} || GPR[rt]_{31\sim sa}$
64   T:   $s \leftarrow 0 || sa$
          $temp \leftarrow (GPR[rt]_{31})^{s} || GPR[rt]_{31\sim s}$
          $GPR[rd] \leftarrow (temp_{31})^{32} || temp$
```

Exceptions:

None

SRAV Shift Right Arithmetic Variable SRAV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SRAV 000111
6	5	5	5	5	6

Format:

SRAV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order five bits of general register *rs*, sign-extending the high-order bits. The result is placed in register *rd*. In64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32   T:   $s \leftarrow GPR[rs]_{4\sim0}$
          $GPR[rd] \leftarrow (GPR[rt]_{31})^{s} \,||\, GPR[rt]_{31\sim sa}$
64   T:   $s \leftarrow GPR[rs]_{4\sim0}$
          $temp \leftarrow (GPR[rt]_{31})^{s} \,||\, GPR[rt]_{31\sim s}$
          $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$
```

Exceptions:

None

SRL Shift Right Logical SRL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRL 000010
6	5	5	5	5	6

Format:

SRL rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, inserting zeros into the high-order bits. The result is placed in register *rd*. In64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32   T:   $GPR[rd] \leftarrow 0^{sa} || GPR[rt]_{31\sim sa}$
64   T:   $s \leftarrow 0 || sa$
          $temp \leftarrow 0^{s} || GPR[rt]_{31\sim s}$
          $GPR[rd] \leftarrow (temp_{31})^{32} || temp$
```

Exceptions:

None

SRLV Shift Right Logical Variable SRLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SRLV 000110
6	5	5	5	5	6

Format:

SRLV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order five bits of general register *rs*, inserting zeros into the high-order bits. The result is placed in register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32   T:   $s \leftarrow GPR[rs]_{4\sim0}$
          $GPR[rd] \leftarrow 0^{s} \mid\mid GPR[rt]_{31\sim s}$
64   T:   $s \leftarrow GPR[rs]_{4\sim0}$
          $temp \leftarrow 0^{s} \mid\mid GPR[rt]_{31\sim s}$
          $GPR[rd] \leftarrow (temp_{31})^{32} \mid\mid temp$
```

Exceptions:

None

SUB **Subtract** SUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010
6	5	5	5	5	6

Format:

SUB rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd*. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the SUBU instruction is that SUBU never traps on overflow.

An integer overflow exception takes place if the carries out of bits 30 and 31 differ (2's-complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

Operation:

```
32   T:   GPR[rd] ← GPR[rs] – GPR[rt]
64   T:   temp ← GPR[rs] – GPR[rt]
          GPR[rd] ← $(temp_{31})^{32}$ || $temp_{31\sim0}$
```

Exceptions:

Integer overflow exception

SUBU Subtract Unsigned SUBU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUBU 100011
6	5	5	5	5	6

Format:

SUBU rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd*. In 64-bit mode, the operands must be valid sign-extended,32-bit values.

The only difference between this instruction and the SUB instruction is that SUBU never traps on overflow. No integer overflow exception occurs under any circumstances.

Operation:

```
32   T:   GPR[rd] ← GPR[rs] – GPR[rt]
64   T:   temp ← GPR[rs] – GPR[rt]
          GPR[rd] ← $(temp_{31})^{32}$ || $temp_{31\sim0}$
```

Exceptions:

None

SW Store Word SW

31 26	25 21	20 16	15 0
SW 101011	base	rt	offset
6	5	5	16

Format:

SW rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of general register *rt* are stored at the memory location specified by the effective address.

If either of the two least-significant bits of the effective address are non-zero, an address error exception occurs.

Operation:

```
32   T:   vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15\sim0}$) + GPR[base]
          (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ xor (ReverseEndian || 0$^2$)
          byte ← vAddr$_{2\sim0}$ xor (BigEndianCPU || 0$^2$)
          data ← GPR[rt]$_{63-8*byte}$ || 0$^{8*byte}$
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
64   T:   vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15\sim0}$) + GPR[base]
          (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ xor (ReverseEndian || 0$^2$)
          byte ← vAddr$_{2\sim0}$ xor (BigEndianCPU || 0$^2$)
          data ← GPR[rt]$_{63-8*byte}$ || 0$^{8*byte}$
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception

TUB invalid exception

TLB modification exception

Bus error exception

Address error exception

SWCz Store Word From Coprocessor z SWCz

31 26	25 21	20 16	15 0
SWCz 1110xx*	base	rt	offset
6	5	5	16

Format:

SWCz rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. Coprocessor unit z sources a word, which the processor writes to the addressed memory location.

The data to be stored is defined by individual coprocessor specifications. This instruction is not valid for use with CP0. If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

Execution of the instruction referencing coprocessor 3 causes a reserved instruction exception, not a coprocessor unusable exception.

Operation:

```
32    T:    $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15\sim0}) + GPR[base]$
            $(pAddr, unchached) \leftarrow AddressTranslation\ (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1\sim3} \| (pAddr_{2\sim0}\ \mathit{xor}\ (ReverseEndian \| 0^{2})$
            $byte \leftarrow vAddr_{2\sim0}\ \mathit{xor}\ (BigEndianCPU \| 0^{2})$
            $data \leftarrow COPzSW\ (byte, rt)$
            StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
64    T:    $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15\sim0}) + GPR[base]$
            $(pAddr, unchached) \leftarrow AddressTranslation\ (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1\sim3} \| (pAddr_{2\sim0}\ \mathit{xor}\ (ReverseEndian \| 0^{2})$
            $byte \leftarrow vAddr_{2\sim0}\ \mathit{xor}\ (BigEndianCPU \| 0^{2})$
            $data \leftarrow COPzSW\ (byte, rt)$
            StoreMemory (uncache, WORD, data, pAddr, vAddr, DATA)
```

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.

SWCz Store Word From Coprocessor z (Continued) SWCz

Exceptions:

TLB refill exception

TLB invalid exception

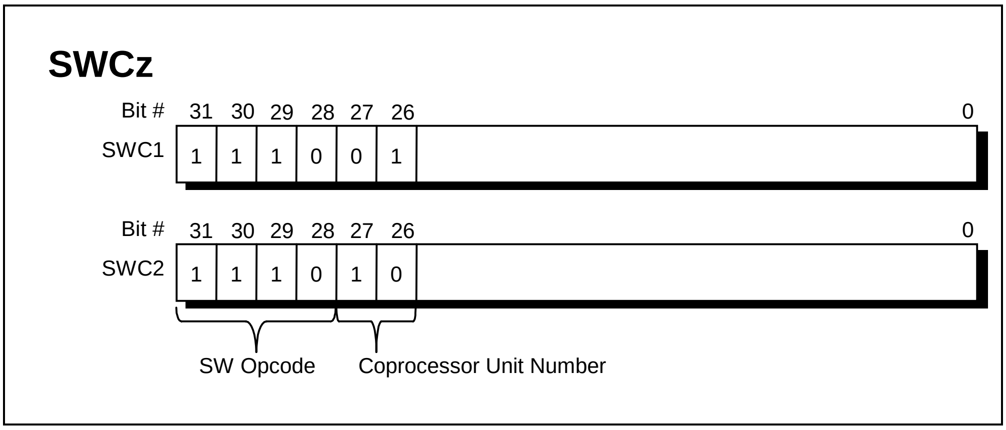
TLB modification exception

Bus error exception

Address error exception

Coprocessor unusable exception

Opcode Bit Encoding:



SWL Store Word Left SWL

31 26	25 21	20 16	15 0
SWL 101010	base	rt	offset
6	5	5	16

Format:

SWL rt, offset (base)

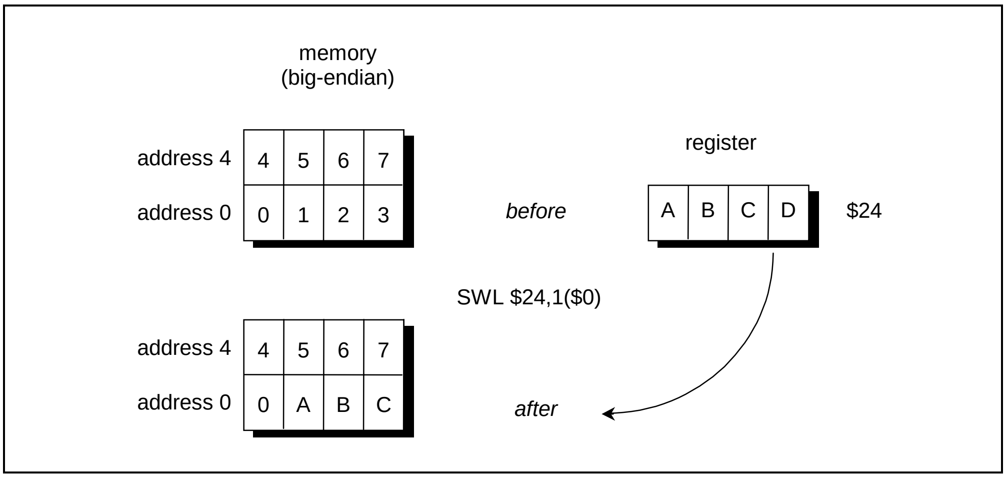
Description:

This instruction can be used with the SWR instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words. SWL stores the left portion of the register into the appropriate part of the high-order word of memory; SWR stores the right portion of the register into the appropriate part of the low-order word.

The SWL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the staring byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it proceeds toward the low-order byte of the register and the low-order byte of the word in memory, copying bytes from register to memory until it reaches the low-order byte of the word in memory.

No address exceptions due to alignment are possible.



SWL Store Word Left (Continued) SWL

Operation:

```
32  T:  vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15\sim0}$) + GPR[base]
        (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
        pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ xor ReverseEndian$^{3}$)
        if BigEndianMem = 0 then
        pAddr ← pAddr$_{31\sim2}$ || 0$^{2}$
        endif
        byte ← vAddr$_{1\sim0}$ xor BigEndianCPU$^{2}$
        if (vAddr$_{2}$ xor BigEndianCPU) = 0 then
        data ← 0$^{32}$ || 0$^{24-8*byte}$ || GPR[rt]$_{31\sim24-8*byte}$
        else
        data ← 0$^{24-8*byte}$ || GPR[rt]$_{31\sim24-8*byte}$ || 0$^{32}$
        endif
        StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)
64  T:  vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15\sim0}$) + GPR[base]
        (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
        pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ xor ReverseEndian$^{3}$)
        if BigEndianMem = 0 then
        pAddr ← pAddr$_{31\sim2}$ || 0$^{2}$
        endif
        byte ← vAddr$_{1\sim0}$ xor BigEndianCPU$^{2}$
        if (vAddr$_{2}$ xor BigEndianCPU) = 0 then
        data ← 0$^{32}$ || 0$^{24-8*byte}$ || GPR[rt]$_{31\sim24-8*byte}$
        else
        data ← 0$^{24-8*byte}$ || GPR[rt]$_{31\sim24-8*byte}$ || 0$^{32}$
        endif
        StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)
```

SWL Store Word Left (Continued) SWL

Given a doubleword in a register and a doubleword in memory, the operation of SWL is as follows:

SWL

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2\sim0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	type	offset		Destination	type	offset	
			LEM	BEM			LEM	BEM
0	I J K L M N O E	0	0	7	E F G H M N O P	3	4	0
1	I J K L M N E F	1	0	6	I E F G M N O P	2	4	1
2	I J K L M E F G	2	0	5	I J E F M N O P	1	4	2
3	I J K L E F G H	3	0	4	I J K E M N O P	0	4	3
4	I J K E M N O P	0	4	3	I J K L E F G H	3	0	4
5	I J E F M N O P	1	4	2	I J K L M E F G	2	0	5
6	I E F G M N O P	2	4	1	I J K L M N E F	1	0	6
7	E F G H M N O P	3	4	0	I J K L M N O E	0	0	7

LEM BigEndianMem = 0

BEM BigEndianMem = 1

Type AccessType (see Figure 2-2) sent to memory

Offset pAddr$_{2\sim0}$ sent to memory

Exceptions:

TLB refill exception

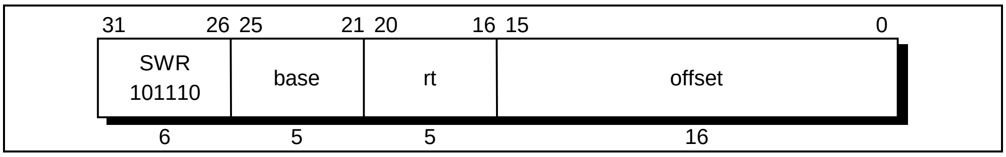
TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

SWR Store Word Right SWR



Format:

SWR rt, offset (base)

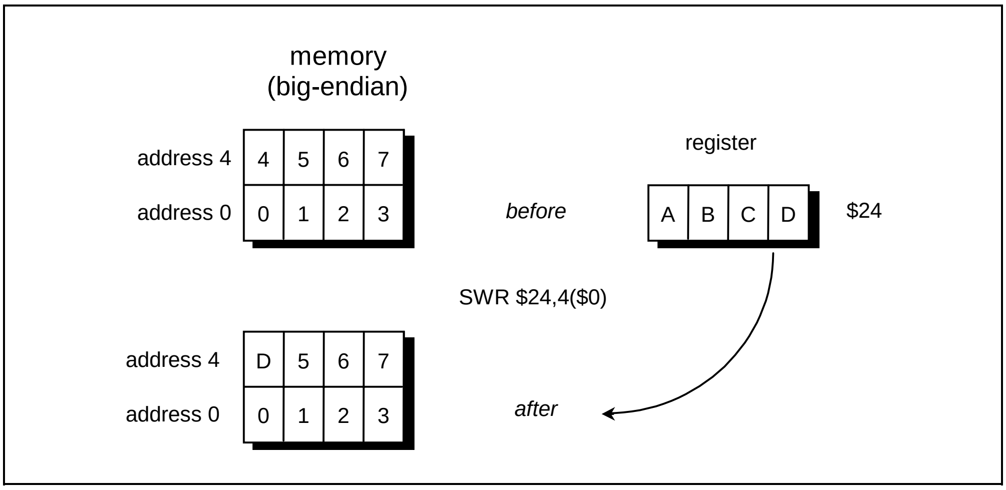
Description:

This instruction can be used with the SWL instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words. SWR stores the right portion of the register into the appropriate part of the low-order word; SWL stores the left portion of the register into the appropriate part of the low-order word of memory.

The SWR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then it proceeds toward the high-order byte of the register and the high-order byte of the word in memory, copying bytes from register to memory until it reaches the high-order byte of the word in memory.

No address exceptions due to alignment are possible.



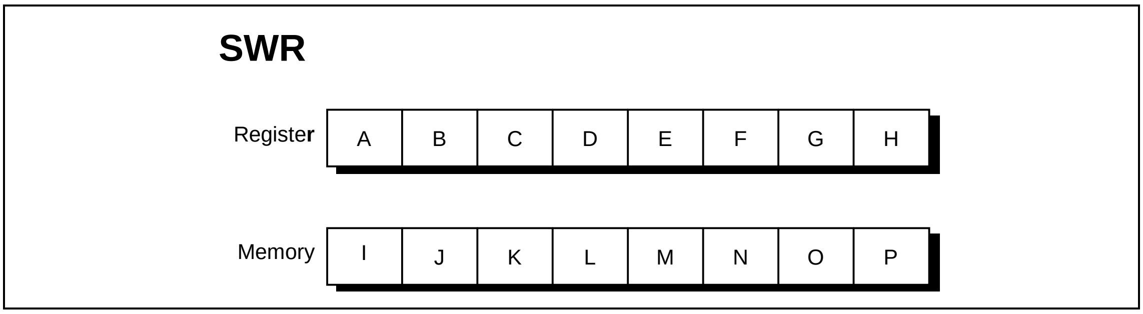
SWR Store Word Right (Continued) SWR

Operation:

```
32   T:   vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15\sim0}$) + GPR[base]
          (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ *xor* ReverseEndian$^{3}$)
          if BigEndianMem = 0 then
          pAddr ← pAddr$_{31\sim2}$ || 0$^{2}$
          endif
          byte ← vAddr$_{1\sim0}$ xor BigEndianCPU$^{2}$
          if (vAddr$_{2}$ xor BigEndianCPU) = 0 then
          data ← 0$^{32}$ || GPR[rt]$_{31-8*byte\sim0}$ || 0$^{8*byte}$
          else
          data ← GPR[rt]$_{31-8*byte\sim0}$ || 0$^{8*byte}$ || 0$^{32}$
          endif
          StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA)
64   T:   vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15\sim0}$) + GPR[base]
          (pAddr, unchached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddr$_{PSIZE-1\sim3}$ || (pAddr$_{2\sim0}$ *xor* ReverseEndian$^{3}$)
          if BigEndianMem = 0 then
          pAddr ← pAddr$_{31\sim2}$ || 0$^{2}$
          endif
          byte ← vAddr$_{1\sim0}$ xor BigEndianCPU$^{2}$
          if (vAddr$_{2}$ xor BigEndianCPU) = 0 then
          data ← 0$^{32}$ || GPR[rt]$_{31-8*byte\sim0}$ || 0$^{8*byte}$
          else
          data ← GPR[rt]$_{31-8*byte\sim0}$ || 0$^{8*byte}$ || 0$^{32}$
          endif
          StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA)
```

SWR Store Word Right (Continued) SWR

Given a doubleword in a register and a doubleword in memory, the operation of SWR is as follows:



$\text{vAddr}_{2\sim0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	type	offset		Destination	type	offset	
			LEM	BEM			LEM	BEM
0	I J K L E F G H	3	0	4	H J K L M N O P	0	7	0
1	I J K L F G H P	2	1	4	G H K L M N O P	1	6	0
2	I J K L G H O P	1	2	4	F G H L M N O P	2	5	0
3	I J K L H N O P	0	3	4	E F G H M N O P	3	4	0
4	E F G H M N O P	3	4	0	I J K L H N O P	0	3	4
5	F G H L M N O P	2	5	0	I J K L G H O P	1	2	4
6	G H K L M N O P	1	6	0	I J K L F G H P	2	1	4
7	H J K L M N O P	0	7	0	I J K L E F G H	3	0	4

LEM BigEndianMem = 0

BEM BigEndianMem = 1

Type AccessType (see Figure 2-2) sent to memory

Offset $\text{pAddr}_{2\sim0}$ sent to memory

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

BUS error exception

Address error exception

SYNC Synchronize SYNC

31 26	25 6	5 0
SPECIAL 000000	0 0000 0000 0000 0000 0000	SYNC 001111
6	20	6

Format:

SYNC

Description:

The SYNC instruction ensures that any loads and stores fetched *prior* to the present instruction are completed before any loads or stores *after* this instruction are allowed to start. Use of the SYNC instruction to serialize certain memory references may be required in multiprocessor environment for proper synchronization.

For example:

Processor A			Processor B	
SW	R1, DATA	1:	LW	R2, FLAG
LI	R2, 1		BEQ	R2, R0, 1B
SYNC			NOP	
SW	R2, FLAG		SYNC	
			LW	R1, DATA

The SYNC in processor A prevents DATA being written after FLAG, which could cause processor B to read stale data. The SYNC in processor B prevents DATA from being read before FLAG, which could likewise result in reading stale data. For processors which only execute loads and stores in order, with respect to shared memory, this instruction is a NOP.

LL and SC instructions implicitly perform a SYNC.

This instruction is allowed in User mode.

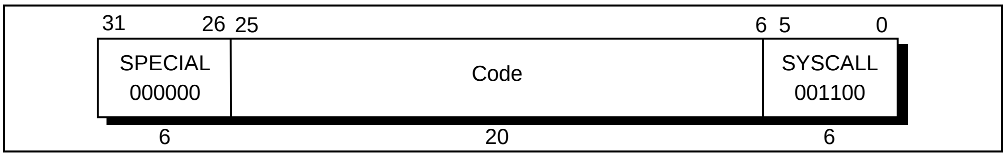
Operation:

```
32, 64  T:      SyncOperation()
```

Exceptions:

None

SYSCALL System Call SYSCALL



Format:

SYSCALL

Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:        SystemCallException
```

Exceptions:

System Call exception

TEQ Trap If Equal TEQ

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TEQ 110100
6	5	5	10	6

Format:

TEQ rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*.

If the contents of general register *rs* are equal to the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:    if GPR[rs] = GPR[rt] then
               TrapException
               endif
```

Exceptions:

Trap exception

TEQI **Trap If Equal Immediate** TEQI

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TEQI 01100	immediate
6	5	5	16

Format:

TEQI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. If the contents of general register *rs* are equal to the sign-extended *immediate*, a trap exception occurs.

Operation:

```
32   T:   if GPR[rs] ← $(immediate_{15})^{16}$ || $immediate_{15\sim0}$ then
          TrapException
          endif
64   T:   if GPR[rs] ← $(immediate_{15})^{48}$ || $immediate_{15\sim0}$ then
          TrapException
          endif
```

Exceptions:

Trap exception

TGE Trap If Greater Than Or Equal TGE

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TGE 110000
6	5	5	10	6

Format:

TGE rs, rt

Description:

The contents of general register *rt* are compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are greater than or equal to the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

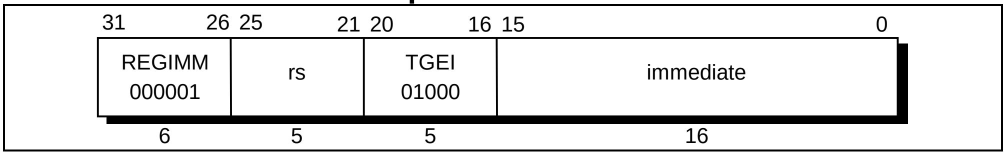
Operation:

```
32, 64  T:    if GPR[rs] ≥ GPR[rt] then
              TrapException
              endif
```

Exceptions:

Trap exception

TGEI Trap If Greater Than Or Equal Immediate TGEI



Format:

TGEI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are greater than or equal to the sign-extended *immediate*, a trap exception occurs.

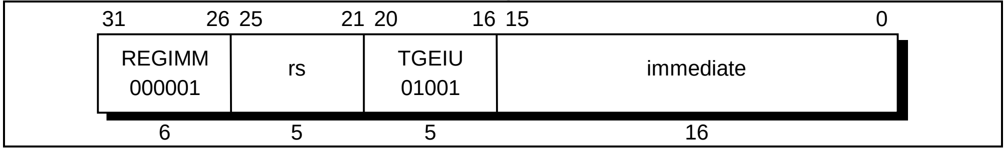
Operation:

```
32   T:   if GPR[rs] ≥ $(\text{immediate}_{15})^{16} || \text{immediate}_{15\sim0}$ then
              TrapException
          endif
64   T:   if GPR[rs] ≥ $(\text{immediate}_{15})^{48} || \text{immediate}_{15\sim0}$ then
              TrapException
          endif
```

Exceptions:

Trap exception

TGEIU Trap If Greater Than Or Equal Immediate Unsigned TGEIU



Format:

TGEIU rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are greater than or equal to the sign-extended *immediate*, a trap exception occurs.

Operation:

```
32   T:   if $(0 \| GPR[rs]) \geq (0 \| (immediate_{15})^{16} \| immediate_{15\sim0})$ then
              TrapException
          endif
64   T:   if $(0 \| GPR[rs]) \geq (0 \| (immediate_{15})^{48} \| immediate_{15\sim0})$ then
              TrapException
          endif
```

Exceptions:

Trap exception

TGEU — Trap If Greater Than Or Equal Unsigned — TGEU

31 … 26	25 … 21	20 … 16	15 … 6	5 … 0
SPECIAL 000000	rs	rt	code	TGEU 110001
6	5	5	10	6

Format:

TGEU rs, rt

Description:

The contents of general register *rt* are compared to the contents of general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are greater than or equal to the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64    T:    if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
                TrapException
                endif
```

Exceptions:

Trap exception

TLBP Probe TLB For Matching Entry TLBP

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBP 001000
6	5	19	6

Format:

TLBP

Description:

The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set.

The architecture does not specify the operation of memory references associated with the instruction immediately after a TLBP instruction, nor is the operation specified if more than one TLB entry matches.

Operation:

```
32   T:   Index ← 1 || $0^{25}$ || $\text{Undeficed}^{6}$
          for i in 0~TLBEntries-1
          if (TLB[i]$_{95\sim77}$ = EntryHi$_{31\sim12}$) and (TLB[i]$_{76}$ or
          (TLB[i]$_{71\sim64}$ = EntryHi$_{7\sim0}$)) then
          Index ← $0^{26}$ || $i_{5\sim0}$
          endif
          endfor
64   T:   Index ← 1 || $0^{25}$ || $\text{Undeficed}^{6}$
          for i in 0~TLBEntries-1
          if (TLB[i]$_{167\sim141}$ and not ($0^{15}$ || TLB[i]$_{216\sim205}$))
          = (EntryHi$_{39\sim13}$ and not ($0^{15}$ || TLB[i]$_{216\sim205}$)) and
          (TLB[i]$_{140}$ or (TLB[i]$_{135\sim128}$ = EntryHi$_{7\sim0}$)) then
          Index ← $0^{26}$ || $i_{5\sim0}$
          endif
          endfor
```

Exceptions:

Coprocessor unusable exception

TLBR Read Indexed TLB Entry TLBR

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBR 000001
6	5	19	6

Format:

TLBR

Description:

The *G* bit (controls ASID matching) read from the TLB is written into both *EntryLo0* and *EntryLo1*.

The *EntryHi* and *EntryLo* registers are loaded with the contents of the TLB entry pointed at by the contents of the TLB *Index* register. The operation is invalid (and the results are unspecified) if the contents of the TLB *Index* register are greater than the number of TLB entries in the processor.

Operation:

```
32  T:   PageMask ← TLB[Index$_{5\sim0}$]$_{127\sim96}$
         EntryHi ← TLB[Index$_{5\sim0}$]$_{95\sim64}$ and not TLB[Index$_{5\sim0}$]$_{127\sim96}$
         EntryLo1 ← TLB[Index$_{5\sim0}$]$_{63\sim32}$
         EntryLo0 ← TLB[Index$_{5\sim0}$]$_{31\sim0}$
64  T:   PageMask ← TLB[Index$_{5\sim0}$]$_{255\sim192}$
         EntryHi ← TLB[Index$_{5\sim0}$]$_{191\sim128}$ and not TLB[Index$_{5\sim0}$]$_{255\sim192}$
         EntryLo1 ← TLB[Index$_{5\sim0}$]$_{127\sim65}$ || TLB[Index$_{5\sim0}$]$_{140}$
         EntryLo0 ← TLB[Index$_{5\sim0}$]$_{63\sim1}$ || TLB[Index$_{5\sim0}$]$_{140}$
```

Exceptions:

Coprocessor unusable exception

TLBWI **Write Indexed TLB Entry** TLBWI

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBWI 000010
6	5	19	6

Format:

TLBWI

Description:

The *G* bit of the TLB is written with the logical AND of the *G* bits in *EntryLo0* and *EntryLo1*.

The TLB entry pointed at by the contents of the TLB *Index* register is loaded with the contents of the *EntryHi* and *EntryLo* registers.

The operation is invalid (and the results are unspecified) if the contents of the TLB *Index* register are greater than the number of TLB entries in the processor.

Operation:

```
32, 64  T:     TLB[Index5~0] ←
               PageMask || (EntryHi and not PageMask) || EntryLo1 || EntryLo0
```

Exceptions:

Coprocessor unusable exception

TLBWR Write Random TLB Entry TLBWR

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBWR 000110
6	5	19	6

Format:

TLBWR

Description:

The *G* bit of the TLB is written with the logical AND of the *G* bits in *EntryLo0* and *EntryLo1*.

The TLB entry pointed at by the contents of the TLB *Random* register is loaded with the contents of the *EntryHi* and *EntryLo* registers.

Operation:

```
32, 64   T:      TLB[Random5~0] ←
                 PageMask || (EntryHi and not PageMask) || EntryLo1 || EntryLo0
```

Exceptions:

Coprocessor unusable exception

TLT Trap If Less Than TLT

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TLT 110010
6	5	5	10	6

Format:

TLT rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*.

Considering both quantities as signed integers, if the contents of general register *rs* are less than the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:     if GPR[rs] < GPR[rt] then
                TrapException
                endif
```

Exceptions:

Trap exception

TLTI Trap If Less Than Immediate TLTI

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TLTI 01010	immediate
6	5	5	16

Format:

TLTI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are less than the sign-extended *immediate*, a trap exception occurs.

Operation:

```
32   T:    if GPR[rs] < $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15\sim0}$ then
           TrapException
           endif
64   T:    if GPR[rs] < $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15\sim0}$) then
           TrapException
           endif
```

Exceptions:

Trap exception

TLTIU **Trap If Less Than Immediate Unsigned** TLTIU

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TLTIU 01011	immediate
6	5	5	16

Format:

TLTIU rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are less than the sign-extended *immediate*, a trap exception occurs.

Operation:

```
32   T:   if $(0 \| GPR[rs]) < (0 \| (immediate_{15})^{16} \| immediate_{15\sim0})$ then
              TrapException
          endif
64   T:   if $(0 \| GPR[rs]) < (0 \| (immediate_{15})^{48} \| immediate_{15\sim0})$ then
              TrapException
          endif
```

Exceptions:

Trap exception

TLTU Trap If Less than Unsigned TLTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TLTU 110011
6	5	5	10	6

Format:

TLTU rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are less than the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:     if (0 || GPR [rs]) < (0 || GPR [rt]) then
                TrapException
                endif
```

Exceptions:

Trap exception

TNE Trap If Not Equal TNE

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TNE 110110
6	5	5	10	6

Format:

TNE rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*. If the contents of general register rs are not equal to the contents of general register *rt*, a tap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:     if GPR [rs] ≠ GPR [rt] then
                TrapException
                endif
```

Exceptions:

Trap exception

TNEI Trap If Not Equal Immediate TNEI

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TNEI 01110	immediate
6	5	5	16

Format:

TNEI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. If the contents of general register *rs* are not equal to the sign-extended *immediate*, a trap exception occurs.

Operation:

```
32   T:   if GPR[rs] $\neq$ (immediate$_{15}$)$^{16}$ || immediate$_{15\sim0}$ then
              TrapException
          endif
64   T:   if GPR[rs]$\neq$(immediate$_{15}$)$^{48}$ || immediate$_{15\sim0}$ then
              TrapException
          endif
```

Exceptions:

Trap exception

WAIT **Wait** WAIT

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	WAIT 100000
6	5	19	6

Format :

WAIT

Description :

The WAIT instruction is used to halt the internal pipeline and thus reduce the power consumption of the CPU. See Chapter 16.

Operation :

```
32, 64    T:        if G-bus is idle then
                    StopPipeline
                    Endif
```

Exceptions :

Coprocessor unusable exception

XOR Exclusive Or XOR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110
6	5	5	5	5	6

Format:

XOR rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical exclusive OR operation. The result is placed into general register *rd*.

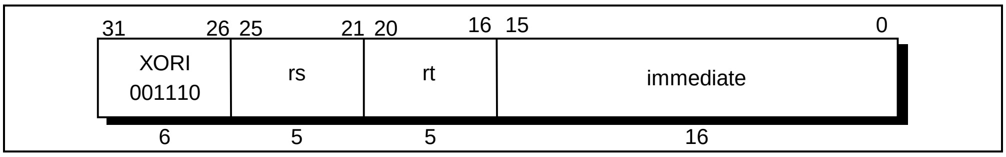
Operation:

32, 64 T: GPR [rd] ← GPR [rs] *xor* GPR [rt]

Exceptions:

None

XORI Exclusive OR Immediate XORI



Format:

XORI rt, rs, immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical exclusive OR operation. The result is placed into general register rt.

Operation:

32 T: GPR [rt] ← GPR [rs] *xor* (0^{16} || immediate)

64 T: GPR [rt] ← GPR [rs] *xor* (0^{48} || immediate)

Exceptions:

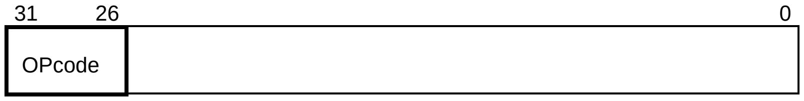
None

A.7 Bit Encoding of CPU Instruction OPcodes

The Table A-2 shows the bit codes for all TX49 CPU instructions(ISA and extended ISA)

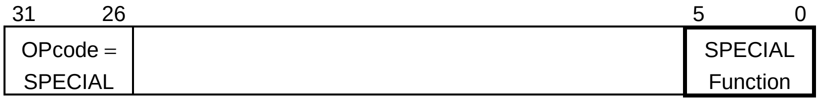
Table A-4 CPU Operation Code Bit Encoding

OPcode



	28~26							
31~29	0	1	2	3	4	5	6	7
0	SPECIA λ	REGIMM λ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0 α	COP1 α	COP2 α	COP3 α θ	BEQL	BNEL	BLEZL	BGTZL
3	DADDI ε	DADDIU ε	LDL ε	LDR ε	MAC λ	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWU ε
5	SB	SH	SWL	SW	SDL ε	SDR ε	SWR	CACHE
6	LL	LWC1 α	LWC2 α	PREF	LLD ε	LDC1 α	LDC2 α	LD ε
7	SC	SWC1 α	SWC2 α	*	SCD ε	SDC1 α	SDC2 α	SD ε

SPECIAL Function



	2~0							
5~3	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	SDBBP	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLV ε	*	DSRLV ε	DSRAV ε
3	MULT	MULTU	DIV	DIVU	DMULT ε	DMULTε	DDIV ε	DDIVU ε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADD ε	DADDU ε	DSUB ε	DSUBU ε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLL ε	*	DSRL ε	DSRA ε	DSLL32 ε	*	DSRL32 ε	DSRA32 ε

REGIMM rt

31 26		20 16	0
OPcode = REGIMM		REGIMM rt	

	18~16							
20~19	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

COPz rs

31 26	25 21	0
OPcode = COPz	COPz rs	

	23~21							
25~24	0	1	2	3	4	5	6	7
0	MF	DMF ε	CF	γ	MT	DMT ε	CT	γ
1	BC	γ	γ	γ	γ	γ	γ	γ
2	CO							
3								

COPz rt

31 26		20 16	0
OPcode = COPz		COPz rt	

	18~16							
20~19	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFL	BCTL	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

COP0 Function

31 26		5 0
OPcode = COP0		COP0 Function

	2~0							
5~3	0	1	2	3	4	5	6	7
0	ϕ	TLBR	TLBWI	ϕ	ϕ	ϕ	TLBWR	ϕ
1	TLBP	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
2	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
3	ERET	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	DERET
4	WAIT	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
5	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
6	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
7	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ

MAC Function

31 26		5 0
OPcode = MAC		MAC Function

	2~0							
5~3	0	1	2	3	4	5	6	7
0	MADD	MADDU	γ	γ	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ
4	γ	γ	γ	γ	γ	γ	γ	γ
5	γ	γ	γ	γ	γ	γ	γ	γ
6	γ	γ	γ	γ	γ	γ	γ	γ
7	γ	γ	γ	γ	γ	γ	γ	γ

Key :

*: This opcode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception.

γ: This opcode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception.

λ: This opecode indicates an instruction class. The instruction word must be further decoded by examining additional tables that show the values for another instruction field.

α: This opcode is a coprocessor operation, not a CPU operation. If the processor state does not allow access to the specified coprocessor, the instruction causes a Coprocessor Unusable exception. It is included in the table because it uses a primary opecode in the instruction encodeing map.

ϕ: This opcode is reserved for future use, but does not cause a Reserved Instruction exception in TX49 implementations. It is treated as "NOP".

θ: This opcode is valid when BC is only selected in COPz rs; In other case, it causes a Reserved Instruction exception .

ε: This opcode is valid when the processor is operating either in the Kernel mode or in the 64-bit non-Kernel (User or Supervisor) mode; In other case, it causes a Reserved Instruction exception .

Appendix B: FPU Instruction Set Details

This appendix provides a detailed description of the operation of each Floating-Point (FPU) instruction. The instructions are listed alphabetically. The exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. The description of the immediate causes and the manner of handling exceptions us omitted horn the instruction descriptions in this chapter. Refer to Chapter 6 for detailed descriptions of floating-point exceptions and handling.

Table B-5 lists the entire bit encoding for the constant fields of the Floating-Point instruction set; the bit encoding for each instruction is included with that individual instruction.

B.1 Instruction Formats

There are three basic instruction format types:

- I-Type, or Immediate instructions, which include load and store operations, M-Type, or Move instructions
- R-Type, or Register instructions, which include the two-and three-register Floating-Point operations.
- Branch instructions and Move instructions

The instruction description subsections that follow show how the three basic instruction formats are used by:

Load and store instructions,
Move instructions, and
Floating-Point Computational instructions.

Floating-point instructions are mapped onto the MIPS coprocessor instructions, defining coprocessor unit number one (CP1) as the floating-point unit.

Each operation is valid only for certain formats. Implementations may support some of these formats and operations only through emulation, but only need support combinations that are valid, which are marked with a V in Table B-1 below. Those combinations marked with a "R" are not currently specified by this architecture, causing an unimplemented instruction trap, to maintain compatibility with future architecture extensions.

Table B-1 Valid FPU Instruction Formats

Operation	Source Format			
	Single	Double	Word	Longword
ADD	V	V	R	R
SUB	V	V	R	R
MUL	V	V	R	R
DIV	V	V	R	R
SQRT	V	V	R	R
ABS	V	V	R	R
MOV	V	V		
NEG	V	V	R	R
TRUNC.L	V	V		
ROUND.L	V	V		
CEIL.L	V	V		
LOOR.L	V	V		
TRUNC.W	V	V		
ROUND.W	V	V		
CEIL.W	V	V		
FLOOR.W	V	V		
CVT.S		V	V	V
CVT.D	V		V	V
CVT.W	V	V		
CVT.L	V	V		
C	V	V	R	R

The coprocessor branch on condition true/false instructions can be used to logically negate any predicate. Thus, the 32 possible conditions require only 16 distinct comparisons, as shown in Table B-2 below.

Table B-2 Logical Negation of Predicates by Condition True/False

Condition			Relations				Invalid Operation exception if unordered
Mnemonic		Code	Greater Than	Less Than	Equal	Unordered	
True	False						
F	T	0	F	F	F	F	No
UN	OR	1	F	F	F	T	No
EQ	NEQ	2	F	F	T	F	No
UEQ	OGL	3	F	F	T	T	No
OLT	UGE	4	F	T	F	F	No
ULT	OGE	5	F	T	F	T	No
OLE	UGT	6	F	T	T	F	No
ULE	OGT	7	F	T	T	T	No
SF	ST	8	F	F	F	F	Yes
NGLE	GLE	9	F	F	F	T	Yes
SEQ	SNE	10	F	F	T	F	Yes
NGL	GL	11	F	F	T	T	Yes
LT	NLT	12	F	T	F	F	Yes
NGE	GE	13	F	T	F	T	Yes
LE	NLE	14	F	T	T	F	Yes
NGT	GT	15	F	T	T	T	Yes

B.1.1 Floating-Point Loads, Stores, and Moves

All movement of data between the floating-point coprocessor and memory is accomplished by coprocessor load and store operations, which reference the floating-point coprocessor's *General-Purpose* Registers. These operations are unformated; no format conversions are performed and, therefore, no floating-point exceptions occur due to these operations.

Data may also be directly moved between the floating-point coprocessor and the processor by move to coprocessor and move from coprocessor instructions. Like the floating-point load and store operations, move to/from operations perform no format conversions and never cause floating-point exceptions.

An additional pair of coprocessor registers are available, called *Floating-Point Control* registers for which the only data movement opera-lions supported are moves to and from processor *General-Purpose* Registers.

B.1.2 Floating-Point Operations

The floating-point unit's operation set includes floating-point add, subtract, multiply, divide, square root, convert between fixed-point and floating-point format, convert between floating-point formats, and floating-point compare. These operations satisfy IEEE Standard 754's requirements for accuracy. Specifically, these operations obtain a result which is identical to performing the result with infinite precision and then rounding to the specified format, using the current rounding mode.

Instructions must specify the format of their operands. Except for con-version functions, mixed-format operations are not provided.

B.2 Instruction Notational Conventions

In this appendix, all variable sub fields in an instruction format (such as *fs*, *ft*, immediate, and so on) are shown with lower-case names. The instruction name (such as ADD, SUB, and so on) is shown in upper-case.

For the sake of clarity, an alias is sometimes substituted for a variable subfield in the formats of specific instructions. For example, we use *rs* = *base* in the format for load and store instructions. Such an alias is always lower case, since it refers to a variable subfield.

In some instructions, however, the two instruction subfields *op* and *function* have constant 6-bit values. When reference is made to these instructions, upper-case mnemonics are used. In the floating-point instruction, for example, we use *op* = COP1 and *function* = FADD. In some cases, a single field has both fixed and variable subfields, so the name contains both upper and lower case characters. Actual bit encoding for mnemonics is shown in Figure B-5 at the end of this appendix, and are also included with each individual instruction.

In the instruction description examples that follow, the *Operation* section describes the operation performed by each instruction using a high-level language notation.

B.2.1 Instruction Notation Examples

Example #1:

$$\boxed{\text{GPR[ft]} \leftarrow \text{immediate} \,||\, 0^{16}}$$

Sixteen zero bits are concatenated with an immediate value (typically 16 bits),. and the 32-bit string (with the lower 16 bits set to zero) is assigned to GPR register *ft*.

Example #2:

$$\boxed{(\text{immediate}_{15})^{16} \,||\, \text{immediate}_{15\sim0}}$$

Bit 15 (the sign bit) of an immediate value is extended for 16 bit positions, and the result is concatenated with bits 15 through 0 of the immediate value to form a 32-bit sign-extended value.

B.3 Load and Store Instructions

In the MIPS ISA, all load operations have a delay of at least one instruction. That is, the instruction immediately following a load cannot use the contents of the register that will be loaded with the data being fetched from storage.

In the TX49, the instruction immediately following a load may use the contents of the register loaded. In such cases, the hardware will interlock, requiring additional real cycles, so scheduling load delay slots is still desirable, although not absolutely required for functional code.

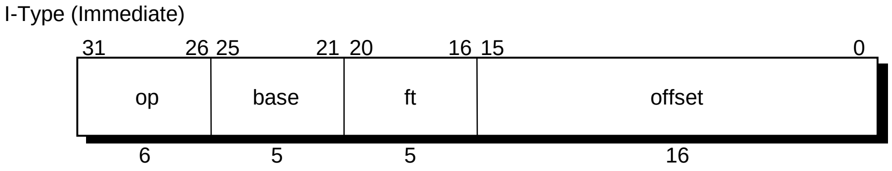
When the FR bit in the *Status* register equals zero, the *Floating-Point General Registers (FGR)* are 32-bits wide. When the FR bit in the Status register equals one, the *Floating-Point General Registers (FGR)* are 64-bits wide. The behavior of the load store insturctions in dependent on the width of the *FGRs*.

In the load/store operation descriptions, the functions listed in Table B-3 are used to summarize the handling of virtual addresses and physical memory.

Table B-3 Load/Store Common Functions

Function	Meaning
AddressTranslation	Uses the TLB to find the physical address given the virtualaddress. The function fails and an exception is taken if therequired translation is not present in the TLB.
LoadMemory	Uses the cache and main memory to find the contents of theword containing the specified physical address. The low-ordertwo bits of the address and the access type field indicates whichof each of the four bytes within the data word need to bereturned. If the cache is enabled for this access, the entire wordis returned and loaded into the cache.
StoreMemory	Uses the cache, write buffer and main memory to store the wordor part of word specified as data in the word containing thespecified physical address. The low-order two bits of theaddress and the access type field indicates which of each of thefour bytes within the data word should be stored.

Figure B-1 shows the I-Type instruction format used by load and store operations.



where:

op is a 6-bit operation code

base is the 5-bit base register specifier

ft is a 5-bit. source (for stores) or destination (for loads) FPA register specifier

offset is the 16-bit signed immediate offset

Figure B-1 Load and Store Instruction Format

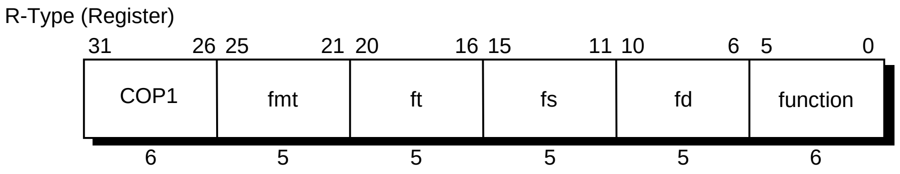
All coprocessor loads and stores reference aligned word data items. Thus, for word loads and stores, the access type field is always WORD, and the low-order two bits of the address must always be zero.

For double word loads and stores, the access type field is always DOUBLEWORD, and the low-order three bits of the address must always be zero.

Regardless of byte-numbering order (endianness), the address specifies that byte which has the smallest byte-address of all of the bytes in the addressed field. For a Big-endian machine, this is the leftmost byte; for a Little-endian machine, this is the rightmost byte.

B.4 Computational Instructions

Computational instructions include all of the arithmetic floating-point operations performed by the FPU. Figure B-2 shows the R-Type instruction format used for computational operations.



where:

COP1	is a 6-bit major operation code
fmt	is a 5-bit format specifier
fs	is a 5-bit source1 register
ft	is a 5-bit source2 register
fd	is a 5-bit destination register
function	is a 6-bit function field

Figure B-2 Computational Instruction Format

Each floating-point instruction can be applied to a number of operand formats. The operand format for an instruction is specified by the 4-bit *Format* field; decoding for this field is shown in Table B-4.

Table B-4 Format Field Decoding

Code	Mnemonic	Size	Format
16	S	single	Binary floating-point
17	D	double	Binary floating-point
18	Reserved		
19	Reserved		
20	W	single	Binary fixed-point
21	L	longword	64-bit binary fixed-point
22~31	-	-	Reserved

The *function* indicates which floating-point operation is to be performed. Table B-5 lists all floating-point instructions.

Table B-5 Floating-Point Instructions and Operations

Code (5~0)	Mnemonic	Operation
0	ADD	Add
1	SUB	Subtract
2	MUL	multiply
3	DIV	Divide
4	SQRT	Square root
5	ABS	Absolute value
6	MOV	Move
7	NEG	Negate
8	ROUND.L	Convert to single fixed-point, rounded to nearest/even
9	TRUNC.L	Convert to single fixed-point, rounded toward zero
10	CEIL.L	Convert to single fixed-point, rounded to +∞
11	FLOOR.L	Convert to single fixed-point, rounded to –∞
12	ROUND.W	Convert to single fixed-point, rounded to nearest/even
13	TRUNC.W	Convert to single fixed-point, rounded toward zero
14	CEIL.W	Convert to single fixed-point, rounded to +∞
15	FLOOR.W	Convert to single fixed-point, rounded to –∞
16~31	-	Reserved
32	CVT.S	Convert to single floating-point
33	CVT.D	Convert to double floating-point
34	-	Reserved
35	-	Reserved
36	CVT.W	Convert to binary fixed-point
37	CVT.L	Convert to 64-bit binary fixed-point
38~47	-	Reserved
48~63	C	Floating-point compare

In the following pages, the notation FGR refers to the FPU's 32 General-Purpose Registers FGRO through FGR31, and FPR refers to the FPU's Floating-Point Registers. When the FR bit in the *Status* register (SR$_{26}$) equals zero, only the even Floating-Point Registers are valid and the FPU's 32 General-Purpose Registers are 32-bits wide. When the FR bit in the *Status* register (SR$_{26}$) equals one, both odd and even Floating-Point Registers may be used and the FPU's 32 General-Purpose Registers are 64-bits wide.

The following routines are used in the description of the floating-point operations to get the value of an FPR or to change the value of an FGR:

```
32 Bit Mode
value < - - ValueFPR (fpr, fmt)
     /* undefined for odd fpr */
     case fmt of
     S, W:
             value < - - FGR[fpr + 0]
     D:
             /* undefined for fpr not even */
             value < - - FGR[fpr + 1] || FGR[fpr + 0]
     end

StoreFPR (fpr, fmt, value):
     /* undefined for odd fpr */
     case fmt of
     S, W:
             FGR[fpr + 1] < - - undefined
             FGR[fpr + 0] < - - value
     D:
             FGR[fpr + 1] < - - value$_{63\sim32}$
             FGR[fpr + 0] < - - value$_{31\sim0}$
     end
```

```
64 Bit Mode
value < - - ValueFPR (fpr, fmt)
     case fmt of
     S:
             value < - - FGR[fpr]$_{31\sim0}$
     D, L:
             value < - - FGR[fpr]
     W:
             value < - - FGR[fpr]
     end

StoreFPR (fpr, fmt, value):
     case fmt of
     S, W:
             FGR[fpr] < - - undefined$^{32}$ || value
     D, L:
             FGR[fpr] < - - value
     end
```

ABS.fmt Floating-Point Absolute Value ABS.fmt

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
COP1 010001	fmt	0 00000	fs	fd	ABS 000101
6	5	5	5	5	6

Format:

ABS.fmt fd, fs

Description:

The contents of the FPU register specified by *fs* are interpreted in thespecified format and the arithmetic absolute value is taken. The result is placed in the floating-point register specified by *fd*.

The absolute value operation is arithmetic; a NaN operand signals in-valid operation.

This instruction is valid only for single- and double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
32, 64    T:        StoreFPR (fd, fmt, AbsoluteValue (ValueFPR (fs, fmt)))
```

Exceptions:

Coprocessor unusable exception

Coprocessor exception tap

Coprocessor Exceptions:

Unimplemented operation exception

Invalid operation exception

ADD.fmt Floating-Point Add ADD.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	ADD 000000
6	5	5	5	5	6

Format:

ADD.fmt fd, fs, ft

Description

The contents of the FPU registers specified by *fs* and *ft* are interpreted in the specified format and arithmetically added. The result is round-ed as if calculated to infinite precision and then rounded to the specified format (*fmt*), according to the current rounding mode. The result is placed in the floating-point register (*FPR*) specified by *fd*.

This instruction is valid only for single- and double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
32, 64     T:          StoreFPR (fd, fmt, ValueFPR(fs, fmt) + ValueFPR (fl, fmt))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact exception

Overflow exception

Underflow exception

BC1F Branch On FPU False (coprocessor 1) BC1F

31 26	25 21	20 16	15 0
COP1 010001	BC 01000	BCF 00000	offset
6	5	5	16

Format:

BC1F offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the result of the last floating-point compare is false(zero), the program branches to the target address, with a delay of one instruction. There must be at least one instruction between C.cond. fmt and BC1F.

Operation:

```
32  T – 1:  condition ← not COC[1]
    T:      target ← $(offset_{15})^{14}$ || offset || $0^2$
    T + 1:  if condition then
            PC ← PC + target
            endif
64  T – 1   condition ← not COC[1]
    T:      target ← $(offset_{15})^{46}$ || offset || $0^2$
    T + 1:  if condition then
            PC ← PC + target
            endif
```

Exceptions:

Coprocessor unusable exception

BC1FL Branch On FPU False Likely (coprocessor 1) BC1FL

31 26	25 21	20 16	15 0
COP1 010001	BC 01000	BCFL 00010	offset
6	5	5	16

Format:

BC1FL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended.

If the result of the last floating-point compare is false(zero), the program branches to the target address, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified. There must be at least on instruction between C.cond. fmt and BC1FL.

Operation:

```
32   T – 1:   condition ← not COC[1]
     T:       target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
     T + 1:   if condition then
              PC ← PC + target
              Else
              NullifyCurrentInstruction
              Endif
64   T – 1:   condition ← not COC[1]
     T:       target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
     T + 1:   if condition then
              PC ← PC + target
              Else
              NullifyCurrentInstruction
              endif
```

Exceptions:

Coprocessor unusable exception

BC1T Branch On FPU True (coprocessor 1) BC1T

31 26	25 21	20 16	15 0
COP1 010001	BC 01000	BCT 00001	offset
6	5	5	16

Format:

BC1T offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the result of the last floating-point compare is true(one), the program branches to the target address, with a delay of one instruction. There must be at least one instruction between C.cond. fmt and BC1T.

Operation:

```
32  T – 1:  condition ← COC[1]
    T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
    T + 1:  if condition then
                PC ← PC + target
            endif
64  T – 1:  condition ← COC[1]
    T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
    T + 1:  if condition then
                PC ← PC + target
            endif
```

Exceptions:

Coprocessor unusable exception

BC1TL **Branch On FPU True Likely (coprocessor 1)** BC1TL

31 26	25 21	20 16	15 0
COP1 010001	BC 01000	BCTL 00011	offset
6	5	5	16

Format:

BC1TL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended.

If the result of the last floating-point compare is true(one), the program branches to the target address, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified. There must be at least one instruction between C.cond.fmt and BC1TL.

Operation:

```
32   T – 1:   condition ← COC[1]
     T:       target ← $(offset_{15})^{14}$ || offset || $0^2$
     T + 1:   if condition then
              PC ← PC + target
              else
              NullifyCurrentInstruction
              endif
64   T – 1:   condition ← COC[1]
     T:       target ← $(offset_{15})^{46}$ || offset || $0^2$
     T + 1:   if condition then
              PC ← PC + target
              else
              NullifyCurrentInstruction
              endif
```

Exceptions:

Coprocessor unusable exception

C.cond.fmt Floating-Point Compare C.cond.fmt

31 26	25 21	20 16	15 11	10 6	5 4	3 0
COP1 010001	fmt	ft	fs	0 00000	FC*	cond*
6	5	5	5	5	2	4

Format:

C.cond.fmt fs, ft

Description:

The contents of the floating-point registers specified by *fs* and *ft* are interpreted in the specified format and arithmetically compared.

A result is determined based on the comparison and the conditions specified in the instruction. If one of the values is a Not a Number (NaN), and the high-order bit of the *condition* field is set, an invalid operation exception is taken. After a one-instruction delay, the condition is available for testing with branch on floating-point coprocessor condition instructions. There must be at least one instruction between the conpare and branch.

Comparisons are exact and can neither overflow nor underflow. Four mutually exclusive relations are possible results: less than, equal, greater than, and unordered. The last case arises when one or both of the operands are NaN; every NaN compares unordered with every-thing, including itself. Comparisons ignore the sign of zero, so + 0 = –0.

This instruction is valid only for single- and double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

**See “FPU Instruction Opcode Bit Encoding” at the end of Appendix B.

C.cond.fmt **Floating-Point Compare (continued)** C.cond.fmt

Operation:

```
32, 64    T:        if NaN (ValueFPR(is, fmt)) *or* NaN (ValueFPR(it, fmt)) then
                      less $\leftarrow$ false
                      equal $\leftarrow$ false
                      unordered $\leftarrow$ true
                      if cond$_3$ then
                              signal InvalidOperationException
                      endif
                  else
                      less $\leftarrow$ ValueFPR (fs, fmt) $<$ ValueFPR (It, fmt)
                      equal $\leftarrow$ ValueFPR (fs, fmt) $=$ ValueFPR (it, fmt)
                      unordered $\leftarrow$ false
                  endif
                  condition $\leftarrow$ (cond$_2$ *and* less) or (cond$_1$ *and* equal) or
                      (cond$_0$ *and* unordered)
                  FCR[31]$_{23}$ $\leftarrow$ condition
                  COC[1] $\leftarrow$ condition
```

Exceptions:

Coprocessor unusable

Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception

Invalid operation exception

CEIL.L.fmt Floating-Point Ceiling to Long Fixed-Point Format CEIL.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CEIL.L 001010
6	5	5	5	5	6

Format:

CEIL.L.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in. the specified source format, *fmt*, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to + ∞ (2).

This instruction is valid only for conversion from single-, double-, extended or quad-precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined if bit 0 of the source register specification is set, since the register number specifies an aligned coprocessor general register. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity, NaN, or the correctly rounded integer result us outside of -2^{63} to 2^{63} -1, the Invalid operation exception us raised. If the Invalid operation is not enabled then no exception us taken and 2^{63} -1 is returned.

This instruction is not implemented on MIPS I or MIPS II processors, and Will cause an unimplemented operation exception to occur.

Operation:

```
32, 64    T:        StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisor mode)

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

Overflow exception

CEIL.W.fmt Floating-Point Ceiling to Single Fixed-Point Format CEIL.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CEIL.W 001110
6	5	5	5	5	6

Format:

CEIL.W.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to + ∞ (2).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to 2^{31}-1, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and 2^{31}-1 is returned.

Operation:

```
32, 64    T:        StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

Overflow exception

CFC1 Move Control Word From FPU (coprocessor 1) CFC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	CF 00010	rt	fs	0 000 0000 0000
6	5	5	5	11

Format:

CFC1 rt, fs

Description:

The contents of the FPU’s control register *fs* are loaded into general register *rt*.

This operation is only defined when *fs* equals 0 or 31.

The contents of general register *rt* are undefined for the instruction immediately following CFC1.

Operation:

```
32    T:       temp ← FCR[fs]
      T + 1:   GPR[rt] ← temp
64    T:       temp ← FCR[fs]
      T + 1:   GPR[rt] ← $(temp_{31})^{32}$ || temp
```

Exceptions:

Coprocessor unusable exception

CTC1 Move Control Word To FPU (coprocessor 1) CTC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	CT 00110	rt	fs	0 000 0000 0000
6	5	5	5	11

Format:

CTC1 rt, fs

Description:

The contents of general register *rt* are loaded into the FPU's control register *fs*. This operation is only defined when *fs* equals 0 or 31. Writing to *Control Register 31*, the floating-point *Control/Status* register, causes an interrupt or exception if any cause bit and its corresponding enable bit are both set. The register will be written before the exception occurs. The contents of floating-point control register fs are undefined for the instruction immediately following CTC1.

Operation:

```
32   T:      temp ← GPR[rt]
     T + 1:  FCR[fs] ← temp
             COC[1] ← $FCR[31]_{23}$
64   T:      temp ← $GPR[rt]_{31\sim0}$
     T + 1:  FCR[fs] ← temp
             COC[1] ← $FCR[31]_{23}$
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception

Invalid operation exception

Division by zero exception

Inexact exception

Overflow exception

Underflow exception

CVT.D.fmt Floating-Point Convert to Double Fixed-Point Format CVT.D.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.D 100001
6	5	5	5	5	6

Format:

CVT.D.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* is interpreted in the specified source format, *fmt*, and arithmetically converted to the double. binary floating-point format. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for conversions from single floating-pount format, 32-bit or 64-bit fixed-point format.

If the single floating-point or single fixed-point format is specified, the operation is exact. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
32, 64    T:        StoreFPR (fd, D, ConvertFmt (ValueFPR (fs, fmt), fmt, D))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

Overflow exception

Underflow exception

CVT.L.fmt Floating-Point Convert to Long Fixed-Point Format CVT.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.L 100101
6	5	5	5	5	6

Format:

CVT.L.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* is interpreted in the specified source format, *fmt*, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for conversions from single-, double-, extended- or quard-precision floating-point formats. If extended- or quad-precision format is specified, the operation is not defined if bit 0 of the source register specification is set, since the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of -2^{63} to 2^{63}-1, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and 2^{63}-1 is returned.

This instruction is not implemented on MIPS I or MIPS II processors, and will cause an unimplemented operation exception to occur.

The operation is not defined if bit 0 of any register specification is set and the FR bit in the status register epuals zero.

Operation:

```
32, 64    T:        StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisor mode)

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

Overflow exception

CVT.S.fmt Floating-Point Convert to Single Fixed-Point Format CVT.S.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.S 100000
6	5	5	5	5	6

Format:

CVT.S.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single binary floating-point format. The result is placed in the floating-point register specified by *fd*. Rounding occurs according to the currently specified rounding mode.

This instruction is valid only for conversions from double floating-point format, or from 32-bit or 64-bit fixed-point format. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
32, 64    T:        StoreFPR (fd, S, ConvertFmt (ValueFPR (fs, fmt), fmt, S))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

Overflow exception

Underflow exception

CVT.W.fmt Floating-Point Convert to Fixed-Point Format CVT.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.W 100100
6	5	5	5	5	6

Format:

CVT.W.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result us outside of -2^{31} to 2^{31}-1, an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and 2^{31}-1 is returned.

Operation:

```
32, 64    T:        StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

Overflow exception

DIV.fmt Floating-Point Divide DIV.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	DIV 000011
6	5	5	5	5	6

Format:

DIV.fmt fd, fs, ft

Description:

The contents of the floating-point registers specified by *fs* and *ft* are interpreted in the specified *format* and the value in *fs* is divided by the value in *ft*. The result is rounded as if calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. The result is placed in the floating-point register specified by *fd*.

This instruction is valid for only single or double precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
32, 64    T:         StoreFPR (fd, fmt, ValueFPR(fs, fmt)/ValueFPR(ft, fmt))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception

Invalid operation exception

Division-by-zero exception

Inexact exception

Overflow exception

Underflow exception

DMFC1 Doubleword Move From Floating-Point Coprocessor DMFC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	DMF 00001	rt	fs	0 000 0000 0000
6	5	5	5	11

Format:

DMFC1 rt, fs

Description:

The contents of register *fs* from the floating-point coprocessor is stored into processor register *rt*.

The contents of general register *rt* are undefined for the instruction immediately following DMFC1.

The *FR* bit in the *Status* register specifies whether all 32 register of the TX49 are addressable. When FR is clear, this instruction is not defined when the least significant bit of *fs* is non-zero. When *FR* is set, *fs* may specify either odd or even registers.

Operation:

```
64   T:     if $SR_{26}$ = 1 then /*64-bit wide FGRs*/
            data ←    FGR[fs]
            elseif $fs_0$ = 0 then /*valid specifier, 32-bit wide FGRs*/
            data ←    FGR[fs+1] || FGR[fs]
            else /*undefined for odd 32-bit reg #s */
            data ←    undefined$^{64}$
            endif
     T+1:   GPR[rt] ← data
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisor mode)

Coprocessor Exceptions:

Unimplemented operation exception

DMTC1 Doubleword Move To Floating-Point Coprocessor DMTC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	DMT 00101	rt	fs	0 000 0000 0000
6	5	5	5	11

Format:

DMTC1 rt, fs

Description:

The contents of general register *rt* are loaded into coprocessor register *fs* of the CP1.

The contents of floating-point register *fs* are undefined for the instruction immediately following DMTC1.

The *FR* bit in the *Status* register specifies whether all 32 register of the TX49 are addressable. When *FR* equals zero, this instruction is not defined when the least significant bit of *fs* is non-zero. When *FR* equals one, *fs* may specify either odd or even registers.

Operation:

```
64   T:      data ← GPR[rt]
     T + 1:  if $SR_{26}$ = 1 then /*64-bit wide FGRs*/
             FGR[fs] ← data
             elseif $fs_0$ = 0 then /*valid specifier, 32-bit wide valid FGRs*/
             FGR[fs + 1] ← $data_{63\sim32}$
             FGR[fs] ← $data_{31\sim0}$
             else /*undefined result for odd 32-bit reg #s */
             undefined_result
             endif
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisor mode)

Coprocessor Exceptions:

Unimplemented operation exception

FLOOR.L.fmt — Floating-Point Floor to Long Fixed-Point Format — FLOOR.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	FLOOR.L 001011
6	5	5	5	5	6

Format:

FLO0R.L.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conver-sion is rounded as if the current rounding mode is round to –∞ (3).

This instruction is valid only for conversion from single-, double-, extended or quad-precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined if bit 0 of the source register specification is set, since the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of -2^{63} to 2^{63}-1, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and 2^{63}-1 is returned. This instruction is not implemented on MIPS I or MIPS II processors, and will cause an unimplemented operation exception to occur.

Operation:

```
32, 64    T:        StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisor mode)

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

Overflow exception

FLOOR.W.fmt Floating-Point Floor to Single Fixed-Point Format FLOOR.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	FLOOR.W 001111
6	5	5	5	5	6

Format:

FLOOR.W.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to –∞ (RM = 3).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to 2^{31}-1, an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and 2^{31}-1 is returned.

Operation:

```
32, 64    T:        StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

Overflow exception

LDC1 Load Doubleword to FPU (coprocessor 1) LDC1

31 26	25 21	20 16	15 0
LDC1 110101	base	ft	offset
6	5	5	16

Format:

LDC1 ft, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address. In 32-bit mode, the contents of the doubleword at the memory location specified by the effective address is loaded into registers *ft* and *ft + 1* of the floating-point coprocessor. This instruction is not valid, and is undefined, when the least significant bit of *ft* is non-zero. In 64-bit mode, the contents of the doubleword at the memory location specified by the effective ad-dress are loaded into the 64-bit register *ft* of the floating point coprocessor. The *FR* bit of the *Status* register (SR$_{26}$) specifies whether all 32 registers of the TX49 are addressable. When FR = 0, this instruction is not defined when the least significant bit of *ft* is non-zero. When FR = 1, *ft* may specify either odd or even registers.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

LDC1 Load Doubleword to FPU (coprocessor 1) (continued) LDC1

Operation:

```
32   T:   vAddr $\leftarrow$ ((offset$_{15}$)$^{16}$ || offset$_{15\sim0}$) + GPR[base]
                  (pAddr, uncached) $\leftarrow$ Address Translation (vAddr, DATA)
          data $\leftarrow$ LoadMemory (uncached, DLUBLEWORD, pAddr, vAddr, DATA)
          if SR$_{26}$ = 1 then /*64-bit wide GFRs */
                  FGR[ft] $\leftarrow$ data
          elseif ft$_0$ = 0 then /*valid specifier, 32-bit wide FGRs */
                  FGR[ft + 1] $\leftarrow$ data$_{63\sim32}$
                  FGR[ft] $\leftarrow$ data$_{31\sim0}$
          else /*undefined result if odd */
                  undefined_result
          endif
64   T:   vAddr $\leftarrow$ ((offset$_{15}$)$^{48}$ || offset$_{15\sim0}$) + GPR[base]
                  (pAddr, uncached) $\leftarrow$ Address Translation (vAddr, DATA)
          data $\leftarrow$ LoadMemory (uncached, DLUBLEWORD, pAddr, vAddr, DATA)
          if SR$_{26}$ = 1 then /*64-bit wide GFRs */
                  FGR[ft] $\leftarrow$ data
          elseif ft$_0$ = 0 then /*valid specifier, 32-bit wide FGRs */
                  FGR[ft + 1] $\leftarrow$ data$_{63\sim32}$
                  FGR[ft] $\leftarrow$ data$_{31\sim0}$
          else /*undefined result if odd */
                  undefined_result
          endif
```

Exceptions:

Coprocessor unusable

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LWC1 Load Word to FPU (coprocessor 1) LWC1

31 26	25 21	20 16	15 0
LWC1 110001	base	ft	offset
6	5	5	16

Format:

LWC1 ft, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address. The contents of theword at the memory location specified by the effective address is loaded into register *ft* of the floating-point coprocessor.

The *FR* bit of the *Status* register specifies whether all 64-bit *Floating-Point Registers* are addressable. If *FR* equals zero, LWC1 loads eitherthe high or low half of the 16 even *Floating-Point Registers*. If *FR* equals one, LWC1 loads the low 32-bits of both even and odd *Floating-Point Registers*.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

LWC1 Load Word to FPU (coprocessor 1) (continued) LWC1

Operation:

```
32   T:   vAddr ← $((offset_{15})^{16} \,||\, offset_{15\sim0}) + GPR[base]$
                (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← $pAddr_{PSIZE-1\sim3} \,||\, (pAddr_{2\sim0} xor(ReverseEndian \,||\, 0^2))$
          mem ← LoadMemory(uncached, WORD, pAddr, vAddr, DATA)
          byte ← $vAddr_{2\sim0} xor(BigEndianCPU \,||\, 0^2)$
          /*“mem” is aligned 64-bits from memory. Pick out correct bytes. */
          if $SR_{26} = 1$ then */64-bit wide FRGs */
                FGR[ft] ← $undefined^{32} \,||\, mem_{31+8*byte\sim8*byte}$
          else /*32-bit wide FGRs */
                FGR[rf] ← $mem_{31+8*byte\sim8*byte}$
          endif
64   T:   vAddr ← $((offset_{15})^{48} \,||\, offset_{15\sim0}) + GPR[base]$
                (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← $pAddr_{PSIZE-1\sim3} \,||\, (pAddr_{2\sim0} xor(ReverseEndian \,||\, 0^2))$
          mem ← LoadMemory(uncached, WORD, pAddr, vAddr, DATA)
          byte ← $vAddr_{2\sim0} xor(BigEndianCPU \,||\, 0^2)$
          /*“mem” is aligned 64-bits from memory. Pick out correct bytes. */
          if $SR_{26} = 1$ then */64-bit wide FRGs */
                FGR[ft] ← $undefined^{32} \,||\, mem_{31+8*byte\sim8*byte}$
          else /*32-bit wide FGRs */
                FGR[rf] ← $mem_{31+8*byte\sim8*byte}$
          endif
```

Exceptions:

Coprocessor unusable

TLB-refill exception

TLB invalid exception

Bus error exception

Address error exception

MFC1 **Move From FPU (Coprocessor 1)** MFC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	MF 00000	rt	fs	0 000 0000 0000
6	5	5	5	11

Format:

MFC1 rt, fs

Description:

The contents of register *fs* from the floating-point coprocessor are stored into processor register *rt*.

The contents of register *rt* are undefined for time *T* of the instruction immediately following this load instruction.

The *FR* bit of the Status register specifies whether all 32 registers of the TX49 are addressable. If *FR* equals zero, MFC1 stores either the high or low half of the 16 even *Floating-Point Registers*. If *FR* equals one, MFC1 stores the low 32-bits of both even and odd *Floating-Point Registers*.

Operation:

```
32   T:       $\text{data} \leftarrow \text{FGR[fs]}_{31\sim0}$
     T + 1:   $\text{GPR[rt]} \leftarrow \text{data}$

64   T:       $\text{data} \leftarrow \text{FGR[fs]}_{31\sim0}$
     T + 1:   $\text{GPR[rt]} \leftarrow (\text{data}_{31})^{32} \,||\, \text{data}$
```

Exceptions:

Coprocessor unusable exception

MOV.fmt Floating-Point Move MOV.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	MOV 000110
6	5	5	5	5	6

Format:

MOV.fmt fd, fs

Description:

The contents of the FPU register specified by *fs* are interpreted in the specified *format* and are copied into the FPU register specified by *fd*. The move operation is non-arithmetic; no IEEE 754 exceptions occur as a result of the instruction.

This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
32, 64    T:        StoreFPR (fd, fmt, ValueFPR (fs, fmt))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception

MTC1 **Move To FPU (Coprocessor 1)** MTC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	MT 00100	rt	fs	0 000 0000 0000
6	5	5	5	11

Format:

MTC1 rt, fs

Description:

The contents of register *rt* are loaded into the FPU's general register at location *fs*.

The contents of floating-point register *fs* is undefined for the instruction immediately following MTC1.

The *FR* bit of the *Status* register specifies whether all 32 registers of the TX49 are addressable. If *FR* equals zero, MTC1 loads either the high or low half of the 16 even *Floating-Point Registers*. If *FR* equals one, MTC1 loads the low 32-bits of both even and odd *Floating-Point Registers*.

Operation:

```
32, 64  T:      data ← $GPR[rt]_{31\sim0}$
        T + 1:  if $SR_{26}$ = 1 then /* 64-bit wide FGRs */
                     FGR[fs] ← $undefined^{32}$ || data
                else /* 32-bit wide FGRs */
                endif
```

Exceptions:

Coprocessor unusable exception

MUL.fmt Floating-Point Multiply MUL.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	MUL 000010
6	5	5	5	5	6

Format:

MUL.fmt fd, fs, ft

Description:

The contents of the floating-point registers specified by *fs* and *ft* are interpreted in the specified *format* and arithmetically multiplied. The result is rounded as if calculated to infinite precision and then rounded to the specified *format*, according to the current rounding mode. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
32, 64    T:         StoreFPR (fd, fmt, ValueFPR (fs, fmt)* ValueFPR (ft, fmt))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact exception

Overflow exception

Underflow exception

NEG.fmt Floating-Point Negate NEG.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	NEG 000111
6	5	5	5	5	6

Format:

NEG.fmt fd, fs

Description:

The contents of the FPU register specified by *fs* are interpreted in the specified *format* and the arithmetic negation is taken (the polarity of the sign-bit is changed). The result is placed in the FPU register specified by *fd*.

The negate operation is arithmetic; an NaN operand signals invalid operation.

This instruction is valid only for single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
32, 64    T:        StoreFPR (fd, fmt, Negate (ValueFPR (fs, fmt)))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception

Invalid operation exception

ROUND.L.fmt — Floating-Point Round to Long Fixed-Point Format — ROUND.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ROUND.L 001000
6	5	5	5	5	6

Format:

ROUND.L.fmt fd, fs

Description :

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to nearest/even (0).

This instruction is valid only for conversion from single-, double-, extended or quad-precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined if bit 0 of the source register specification is set, since the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity , NaN, or the correctly rounded integer result is outside of -2^{63} to 2^{63}-1, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and 2^{63}-1 is returned.

This instruction is not implemented on MIPS I or MIPS II processors, and will cause an unimplemented operation exception to occur.

Operation:

```
32, 64    T:        StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisor mode)

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

Overflow exception

ROUND W.fmt **Floating-Point Round to Single Fixed-Point Format** ROUND W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ROUND.W 001100
6	5	5	5	5	6

Format:

ROUND.W.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to nearest/even (RM = 0).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to 2^{31}-1, an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and 2^{31}-1 is returned.

Operation:

```
32, 64      T:          StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

Overflow exception

SDC1 Store Doubleword from FPU (coprocessor 1) SDC1

31 26	25 21	20 16	15 0
SDC1 111101	base	ft	offset
6	5	5	16

Format:

SDC1 ft, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address.

In 32-bit mode, the contents of registers *ft* and *ft + 1* from the floating-point coprocessor are stored at the memory location specified by the effective address. This instruction is not valid, and is undefined, when the least significant bit of *ft* is non-zero.

In 64-bit mode, the 64-bit register *ft* is stored to the contents of the doubleword at the memory location specified by the effective address. The *FR* bit of the *Status* register (SR_{26}) specifies whether all 32 registers of the TX49 are addressable. When FR = 0, this instruction is not defined if the least significant bit of *ft* is non-zero. If FR = 1, *ft* may specify either odd or even registers.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

SDC1 Store Doubleword from FPU (coprocessor 1) (continued) SDC1

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15\sim0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          if $SR_{26} = 1$ /*64-bit wide FGRs */
                  $data \leftarrow FGR[ft]$
          elseif $ft_0 =$ then /* valid specifier, 32-bit wide FGRs */
                  $data \leftarrow FGR[ft + 1] \,||\, FGR[ft]$
          else /*undefined for odd 32-bit reg #s */
                  $data \leftarrow undefined^{64}$
          endif
          StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15\sim0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          if $SR_{26} = 1$ /*64-bit wide FGRs */
                  $data \leftarrow FGR[ft]$
          elseif $ft_0 =$ then /* valid specifier, 32-bit wide FGRs */
                  $data \leftarrow FGR[ft + 1] \,||\, FGR[ft]$
          else /*undefined for odd 32-bit reg #s */
                  $data \leftarrow undefined^{64}$
          endif
          StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

SQRT.fmt Floating-Point Square Root SQRT.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	SQRT 000100
6	5	5	5	5	6

Format:

SQRT.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified *format* and the positive arithmetic square root is taken. The result is rounded as if calculated to infinite precision and then rounded to the specified *format*, according to the current rounding mode. If the value of *fs* corresponds to –0, the result will be –0. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
32, 64    T:        StoreFPR (fd, fmt, SquareRoot (ValueFPR (fs, fmt)))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact exception

SUB.fmt **Floating-Point Subtract** SUB.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	SUB 000001
6	5	5	5	5	6

Format:

SUB.fmt fd,fs, ft

Description:

The contents of the floating-point registers specified by *fs* and *ft* are interpreted in the specified *format* and the value in *ft* is subtracted from the value in *fs*. The result is rounded as if calculated to infinite precision and then rounded to the specified *format*, according to the current rounding mode. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
32, 64    T:        StoreFPR (fd, fmt, ValueFPR (fs, fmt) – ValueFPR (ft, fmt))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact exception

Overflow exception

Underflow exception

SWC1 Store Word from FPU (coprocessor 1) SWC1

31 26	25 21	20 16	15 0
SWC1 111001	base	ft	offset
6	5	5	16

Format:

SWC1 ft, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address. The contents of register *ft* from the floating-point coprocessor are stored at the memory location specified by the effective address.

The *FR* bit of the *Status* register specifies whether all 64-bit *Floating-Point Registers* are addressable. If *FR* equals zero, SWC1 stores either the high or low half of the 16 even *Floating-Point Registers*. If *FR* equals one, SWC1 stores the low 32-bits of both even and odd *Floating-Point Registers*.

If either of the two least-significant bits of the effective address are non-zero, an address error exception occurs.

SWC1 Store Word from FPU (coprocessor 1) (continued) SWC1

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15\sim0}) + GPR[base]$
          (pAddr, uncached) $\leftarrow$ AddressTranslation (vAddr, DATA)
          $pAddr \leftarrow pAddr_{PSIZE-1\sim3} \,||\, (pAddr_{2\sim0}$ xor (RecerseEndian $||\, 0^2$))
          $byte \leftarrow vAddr_{2\sim0}$ xor (BigEndianCPU $||\, 0^2$)
          /* tne bytes of the word are put in the correct byte lanes in
           * “data” for a 64-bit path to memory */
          if $SR_{26} = 1$ then /*64-bit wide FGRs */
                $data \leftarrow FGR[ft]_{63-8*byte\sim0} \,||\, 0^{8*byte}$
          else /* 32-bit wide FGRs /*
                $data \leftarrow 0^{32-8*byte} \,||\, FGR[ft] \,||\, 0^{8*byte}$
          endif
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15\sim0}) + GPR[base]$
          (pAddr, uncached) $\leftarrow$ AddressTranslation (vAddr, DATA)
          $pAddr \leftarrow pAddr_{PSIZE-1\sim3} \,||\, (pAddr_{2\sim0}$ xor (RecerseEndian $||\, 0^2$))
          $byte \leftarrow vAddr_{2\sim0}$ xor (BigEndianCPU $||\, 0^2$)
          /* tne bytes of the word are put in the correct byte lanes in
           * “data” for a 64-bit path to memory */
          if $SR_{26} = 1$ then /*64-bit wide FGRs */
                $data \leftarrow FGR[ft]_{63-8*byte\sim0} \,||\, 0^{8*byte}$
          else /* 32-bit wide FGRs /*
                $data \leftarrow 0^{32-8*byte} \,||\, FGR[ft] \,||\, 0^{8*byte}$
          endif
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

TRUNC.L.fmt **Floating-Point Truncate to Long Fixed-Point Format** TRUNC.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	TRUNC.L 001001
6	5	5	5	5	6

Format:

TRUNC.L.fmt fd, fs

Description :

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round toward zero (1).

This instruction is valid only for conversion from single-, double-, ex-tended or quad-precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined if bit 0 of the source register specification is set, since the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of -2^{63} to 2^{63}-1, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and 2^{63}-1 is returned.

This instruction is not implemented on MIPS I or MIPS II processors, and will cause an unimplemented operation exception to occur.

Operation:

```
32, 64     T:         StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))
```

Note: It is also the same operation in the 32 bit kernel mode.

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Reserved Instruction exception (in the 32 bit user or 32 bit supervisor mode)

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

Overflow exception

TRUNC.W.fmt Floating-Point Truncate to Single Fixed-Point Format TRUNC.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	TRUNC.W 001101
6	5	5	5	5	6

Format:

TRUNC.W.fmt fd, fs

Description:

The contents of the FPU register specified by *fs* are interpreted in the specified source format *fmt* and arithmetically converted to the single fixed-point format. The result us placed in the FPU register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round toward zero (RM = 1).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of $–2^{31}$ to 2^{31}-1, an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and -2^{31} is returned.

Operation:

```
32, 64     T:         StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception

Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception

Unimplemented operation exception

Inexact exception

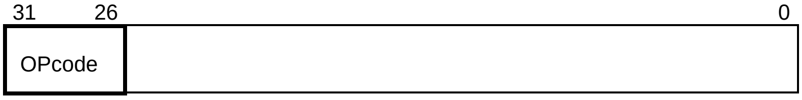
Overflow exception

B.5 Bit Encoding of FPU Instruction OPcodes

The Table B-6 shows the bit codes for all TX49 FPU instructions (ISA and extended ISA)

Table B-6 FPU Operation Code Bit Encoding

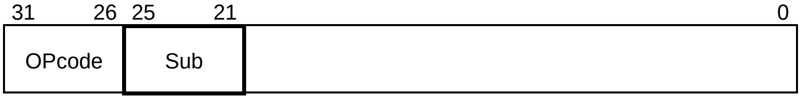
Opcode



28~26

31~29	0	1	2	3	4	5	6	7
0								
1								
2		COP1						
3								
4								
5								
6		LWC1				LDC1 θ		
7		SWC1				SDC1 θ		

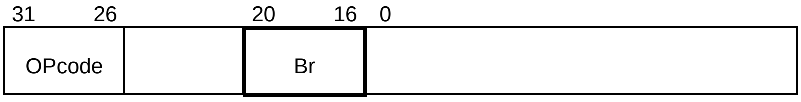
Sub



23~21

25~24	0	1	2	3	4	5	6	7
0	MF	DMF η θ	CF		MT	DMT η θ	CT	δ
1	BC	δ	δ	δ	δ	δ	δ	δ
2	S	D θ	δ	δ	W	L η θ	δ	δ
3	δ	δ	δ	δ	δ	δ	δ	δ

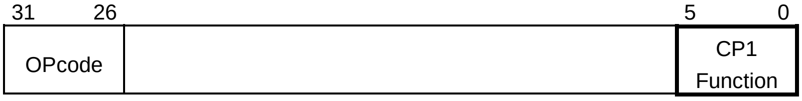
Br



18~16

20~19	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFL	BCTL	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

CP1 Function



2~0

5~3	0	1	2	3	4	5	6	7
0	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	ROUND.Lηθ	TRUNC.L η θ	CEIL.L η θ	FLOOR.L η θ	ROUND.W	TRUNC.W	CEIL.W	FLOORW
2	δ	δ	δ	δ	δ	δ	δ	δ
3	δ	δ	δ	δ	δ	δ	δ	δ
4	CVT.S	CVT.D θ	δ	δ	CVT.W	CVT.L η θ	δ	δ
5	δ	δ	δ	δ	δ	δ	δ	δ
6	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

Key:

γ: This opcode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception.

δ: Thie opcode is reserved for future use. An attempt to execute it causes a Unimplemented operation exceptions in all current implementations.

η: This opcode is valid only when MIPS III instructions are enabled. An attempt to execute these without MIPS III instruction enabled will cause an Unimplemented operation exception.

θ: This opcode is valid only when the TX49 has a double precision FPU in hardware. An attempt to execute these without it will cause an Unimplemented operation exception.

Note:

FPU Instructions are valid only when TX49 has with FPU(CP1). An attempt to execute these insturctions causes a Coprocessor Unusable exception, independent of C0_SR(bit 29)'s value.

Appendix C: Coprocessor 0 Hazards

C.1 Pipeline Interlock and Hazard in TX49

C.1.1 Interlock in Load Delay Slot

Pipeline control logic will interlock the pipeline when detecting a hazard condition and pipeline won't resume until the hazard is resolved.

An example is shown in Figure C-1. In this case, instruction in the load delay slot tries to read the destination register of the load instruction resulting in pipeline stall until the data is read from the cache.

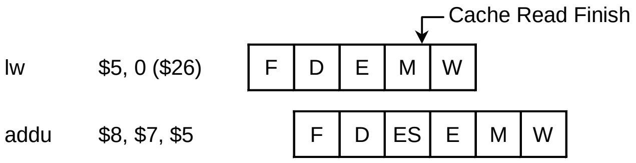


Figure C-1 Interlock in Load Delay Slot

Pipeline also interlocks when the cache miss occurs or when the data is loaded from uncached area (Figure C-2).

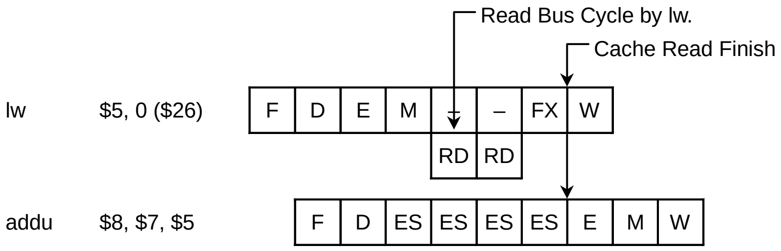


Figure C-2 Interlock in Cache Miss or in the Data Load from Non-cached Area

In this example where there is a register hazard between two consecutive instructions, ADDU will stall at E stage until the destination register of LW is written back.

However, if there is no data dependancy between LW and ADDU, execution of ADDU will complete without stall before the destination register of LW is written back. Pipeline interlock occurs at the first instruction that has the data dependency with the preceding load instruction (Figure C-3).

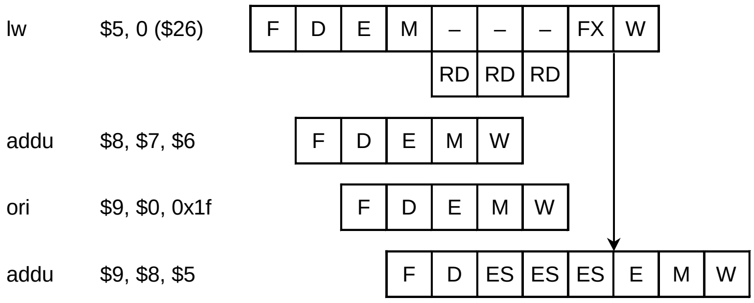


Figure C-3 Pipeline Interlock by Cache Miss

Pipeline also interlocks on write-after-write hazard which is illustrated in Figure C-4. Write-after-write hazard is detected when one of the instructions following a load has the destination register which is same as that of the load instruction. In this example, the ADDU instruction stalls at its E stage until the destination register ($1) of the load is written back.

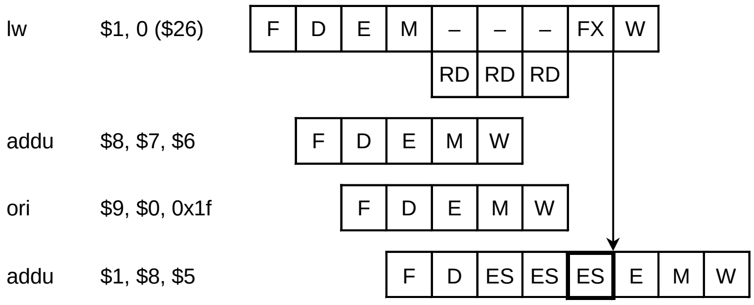


Figure C-4 Write-after-write Hazard by Load Instruction

A SYNC instruction may be placed right after a load instruction. This will cause pipeline stall until the bus cycle issued by the previous load instruction completes (Figure C-5). If the data is read from the cache, there is no bus cycle pending before the SYNC which results in no pipeline stall.

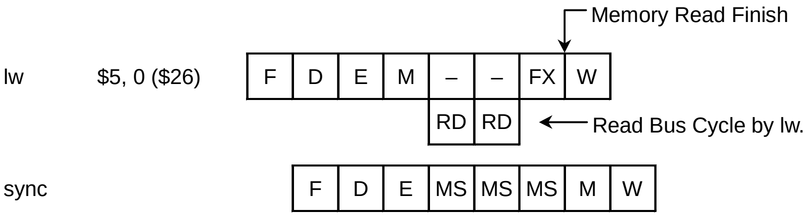


Figure C-5 SYNC Instruction After Load Instruction

C.1.2 Branch Delay Slot

Branch and jump instructions have a branch delay slot (Figure C-6). Also, DERET instruction has a branch delay slot. Note that the result is undefined when the branch/jump instruction is placed in the branch delay slot[1].

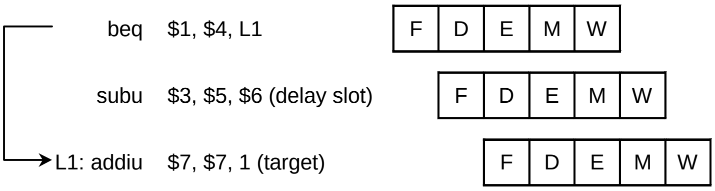


Figure C-6 Branch Delay Slot

[1] Instructions which cause exception, such as, SYSCALL, BREAK, and SDBBP may be placed in the branch delay slot.

C.1.3 Multiply, Multiply/Add and Division Instructions

This subsection explains the pipeline hazard/interlock caused by the combinations of multiply, multiply/add, division, and MTHI/MTLO/MFHI/MFLO instructions (Figure C-7). Basically, the pipeline hazard/interlock by these instructions can be summarized in this way:

- Pipeline interlocks when the data dependency exists.
- Pipeline interlocks when preceding 32-bit multiply or 32-bit multiply/add instruction has <rd> field.
- Pipeline interlocks when 32-bit instruction and 64-bit instruction are executed in sequence.
- HI/LO registers are in undefined state within two instructions before the division instruction, such as, DIV/DIVU/DDIV/DDIVU instruction[2].

		SUCCEEDING INSTRUCTION									
		MULT/ MULTU (2-operand)	MULT/ MULTU (3-operand)	MADD/ MADDU (2-operand)	MADD/ MADDU (3-operand)	MTHI/ MTLO	MFHI/ MFLO	DIV/ DIVU	DMULT/ DMULTU (2-operand)	DMULT/ DMULTU (3-operand)	DDIV/ DDIVU
PRECEEDING INSTRUCTION	MULT/MULTU (2-operand)	NO STALL	NO STALL	NO STALL	NO STALL	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK
	MULT/MULTU (3-operand)	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK
	MADD/MADDU (2-operand)	NO STALL	NO STALL	NO STALL	NO STALL	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK
	MADD/MADDU (3-operand)	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK
	MTHI/MTLO	NO STALL	NO STALL	NO STALL	NO STALL	NO STALL	NO STALL	NO STALL	NO STALL	NO STALL	NO STALL
	MFHI/MFLO	NO STALL	NO STALL	NO STALL	NO STALL	NO STALL	NO STALL	*	NO STALL	NO STALL	*
	DIV/DIVU	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK
	DMULT/DMULTU (2-operand)	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK
	DMULT/DMULTU (3-operand)	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK
	DDIV/DDIVU	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK	INTERLOCK

*: HI/LO registers are in undefined state within two instructions before division instruction

Figure C-7 MAC pipeline hazard/interlock

In the following sections, the pipeline hazards/interlocks caused by the possible combinations of the instructions related multiply, multiply/add, division and both 32-bit and 64-bit operations are illustrated in detail. The Figures in the following sections classifies the cases in such a way that:

A The preceding instruction is immediately followed by 32-bit multiply or multiply/add instruction

B The preceding instruction is immediately followed by MFHI or MFLO intstruction

C The preceding instruction is immediately followed by MTHI or MTLO intstruction

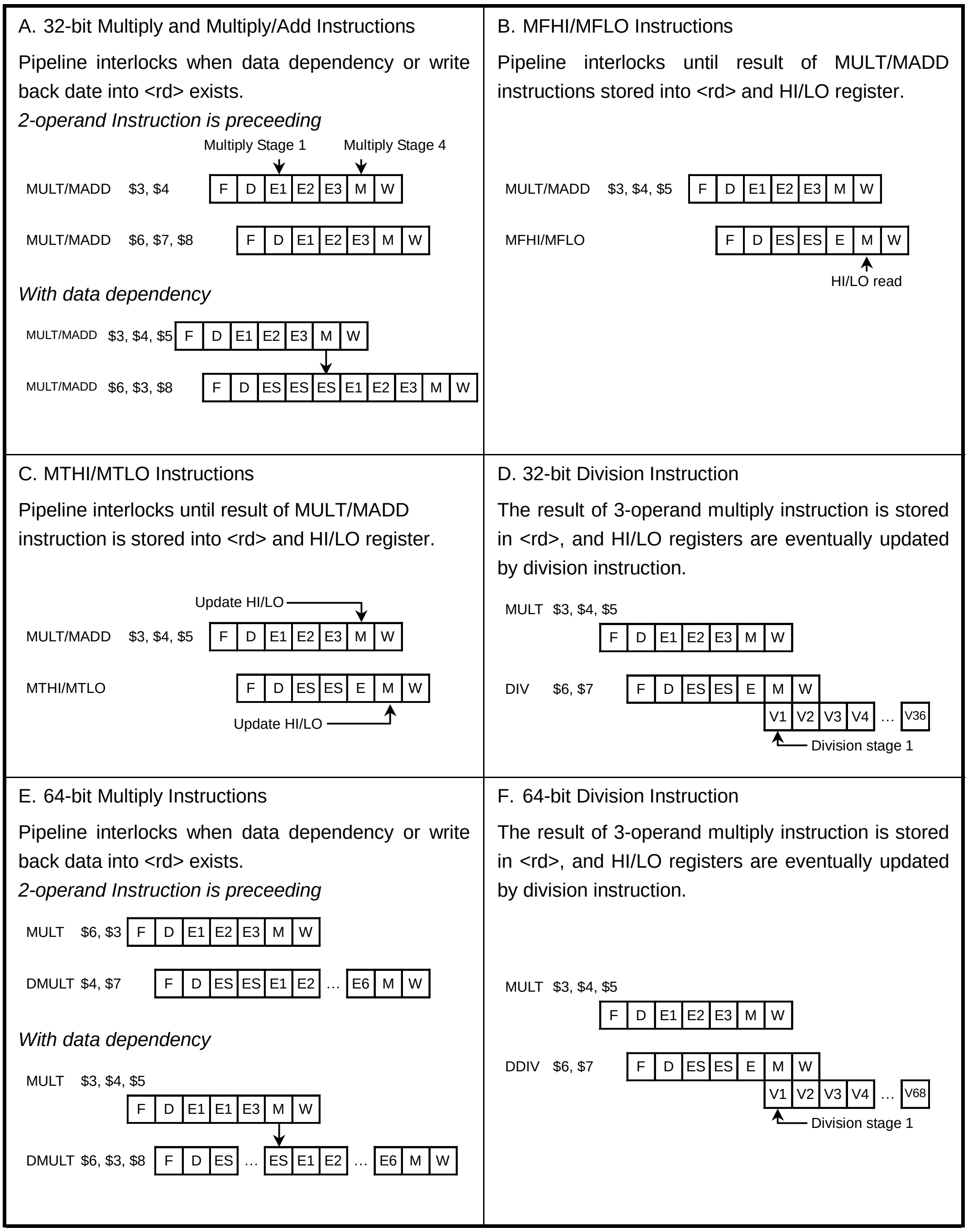
D The preceding instruction is immediately followed by 32-bit division instruction

E The preceding instruction is immediately followed by 64-bit multiply instruction

F The preceding instruction is immediately followed by 64-bit division instruction

[2] In the original R3000, this can be applied to MULT, MULTU, MTHI, and MTLO instructions.

Case 1: Preceding Instruction Is 32-bit Multiply or 32-bit Mutiply/Add Instruction



A. 32-bit Multiply and Multiply/Add Instructions

Pipeline interlocks when data dependency or write back date into <rd> exists.

2-operand Instruction is preceeding

Multiply Stage 1
Multiply Stage 4
MULT/MADD $3, $4 F D E1 E2 E3 M W
MULT/MADD $6, $7, $8 F D E1 E2 E3 M W

With data dependency

MULT/MADD $3, $4, $5 F D E1 E2 E3 M W
MULT/MADD $6, $3, $8 F D ES ES ES E1 E2 E3 M W

B. MFHI/MFLO Instructions

Pipeline interlocks until result of MULT/MADD instructions stored into <rd> and HI/LO register.

MULT/MADD $3, $4, $5 F D E1 E2 E3 M W
MFHI/MFLO F D ES ES E M W
HI/LO read

C. MTHI/MTLO Instructions

Pipeline interlocks until result of MULT/MADD instruction is stored into <rd> and HI/LO register.

Update HI/LO
MULT/MADD $3, $4, $5 F D E1 E2 E3 M W
MTHI/MTLO F D ES ES E M W
Update HI/LO

D. 32-bit Division Instruction

The result of 3-operand multiply instruction is stored in <rd>, and HI/LO registers are eventually updated by division instruction.

E. 64-bit Multiply Instructions

Pipeline interlocks when data dependency or write back data into <rd> exists.

2-operand Instruction is preceeding

MULT $6, $3 F D E1 E2 E3 M W
DMULT $4, $7 F D ES ES E1 E2 … E6 M W

With data dependency

F. 64-bit Division Instruction

The result of 3-operand multiply instruction is stored in <rd>, and HI/LO registers are eventually updated by division instruction.

Figure C-8 Pipeline Hazard/Interlock by 32-bit Multiply or 32-bit Multiply/Add Instruction

Note that in the category A of the Figure C-8, pipeline interlocks for *any* instruction immediately after the multiply or multiply/add instruction when it has the data dependency regarding the general purpose registers. Thus, in the category D, the DIV instruction stalls at the E stage for three cycles when the division instruction has the data dependency with the preceding multiply instruction.

Also note that in the category D of the Figure C-8, Because the division instruction overwrites the HI/LO registers, the HI/LO registers as the result of the 2-operand multiply instruction is undefined. The result of the multiply instruction, as in this figure, is correctly stored in the <rd> register. If the preceding multiply or multiply/add instruction had a <rd> field, pipeline interlocks due to the resource conflict.

Case 2: Preceding Instruction Is MFHI/MFLO Instruction

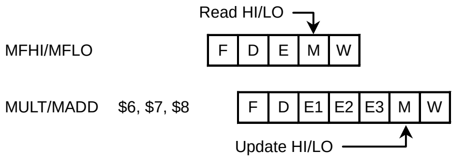
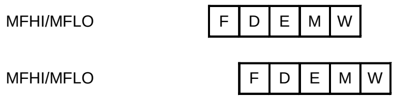
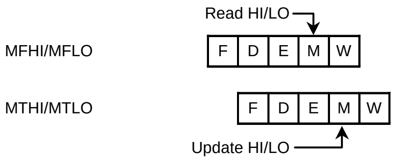
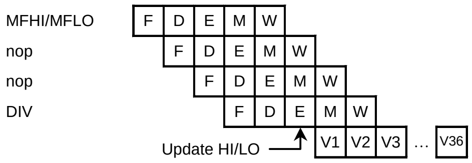
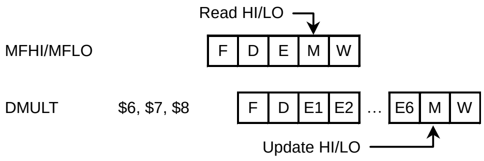
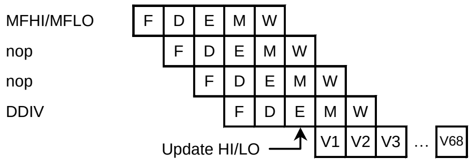
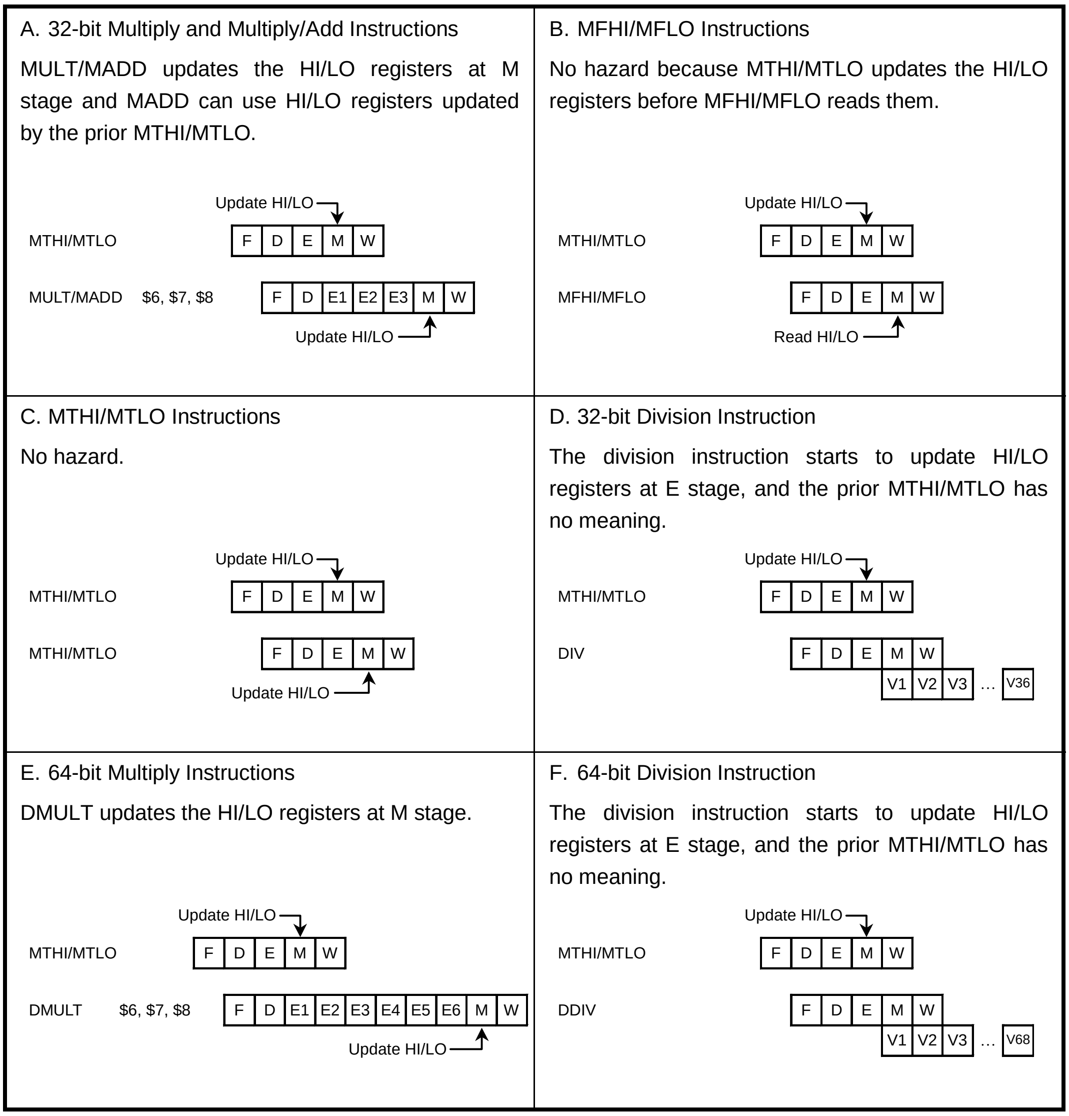
A. 32-bit Multiply and Multiply/Add Instructions MULT/MADD updates the HI/LO registers at M stage and the prior MFHI/MFLO can read the HI/LO registers before the update.  Read HI/LO MFHI/MFLO F D E M W MULT/MADD $6, $7, $8 F D E1 E2 E3 M W Update HI/LO	B. MFHI/MFLO Instructions No hazard. 
C. MTHI/MTLO Instructions No hazard because MTHI/MTLO updates HI/LO resisters at M stage.  Read HI/LO MFHI/MFLO F D E M W MTHI/MTLO F D E M W Update HI/LO	D. 32-bit Division Instruction It is necessary to insert at least two instructions between MFHI/MFLO and DIV.  MFHI/MFLO F D E M W nop F D E M W nop F D E M W DIV F D E M W Update HI/LO V1 V2 V3 … V36
E. 64-bit Multiply Instructions DMULT updates the HI/LO registers at M stage and the prior MFHI/MFLO can read the HI/LO registers before the update.  Read HI/LO MFHI/MFLO F D E M W DMULT $6, $7, $8 F D E1 E2 … E6 M W Update HI/LO	F. 64-bit Division Instruction It is necessary to insert at least two instructions between MFHI/MFLO and DDIV.  MFHI/MFLO F D E M W nop F D E M W nop F D E M W DDIV F D E M W Update HI/LO V1 V2 V3 … V68

Figure C-9 Pipeline Hazard/Interlock by MFHI/MFLO Instructions

Case3: Preceding Instruction Is MTHI/MTLO Instruction



A. 32-bit Multiply and Multiply/Add Instructions

MULT/MADD updates the HI/LO registers at M stage and MADD can use HI/LO registers updated by the prior MTHI/MTLO.

B. MFHI/MFLO Instructions

No hazard because MTHI/MTLO updates the HI/LO registers before MFHI/MFLO reads them.

C. MTHI/MTLO Instructions

No hazard.

D. 32-bit Division Instruction

The division instruction starts to update HI/LO registers at E stage, and the prior MTHI/MTLO has no meaning.

E. 64-bit Multiply Instructions

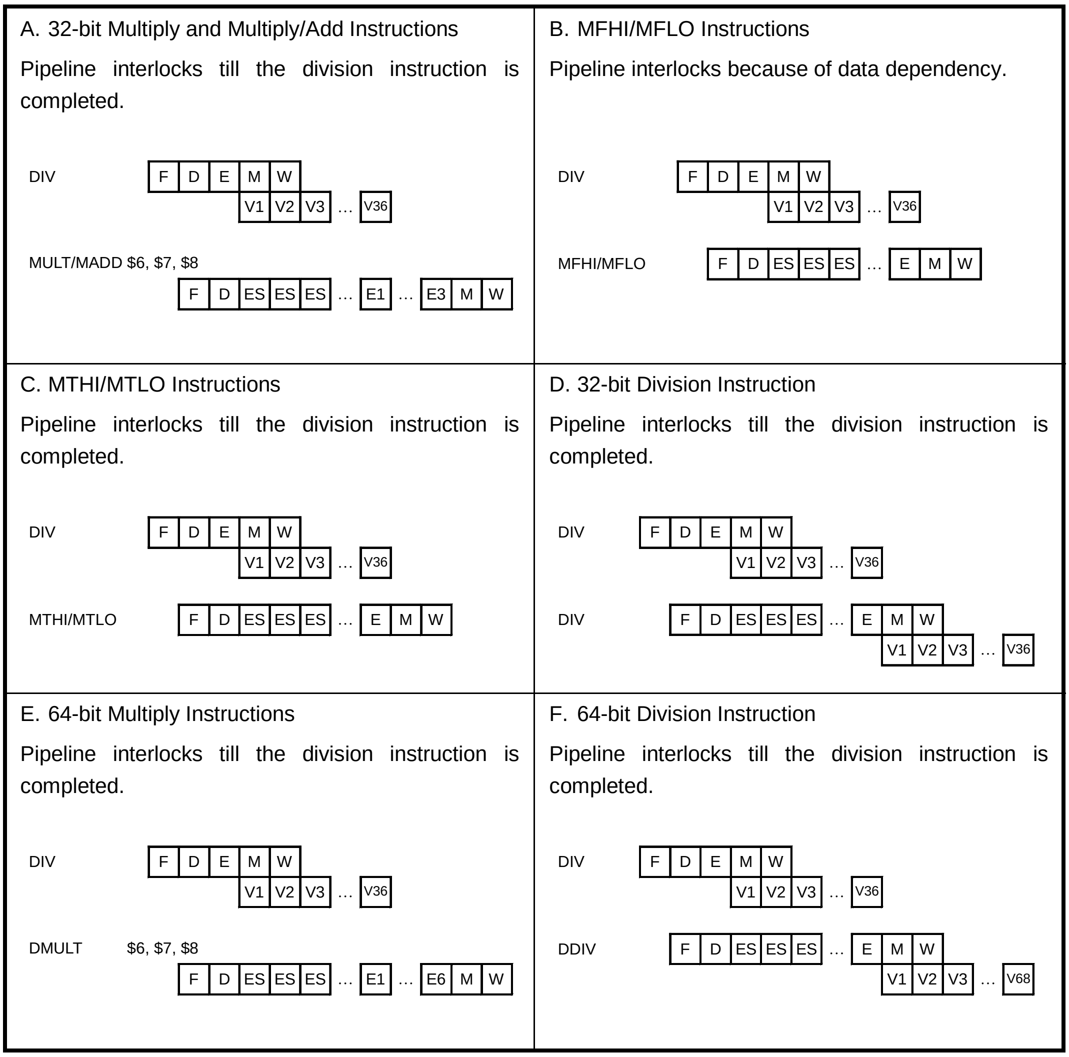
DMULT updates the HI/LO registers at M stage.

F. 64-bit Division Instruction

The division instruction starts to update HI/LO registers at E stage, and the prior MTHI/MTLO has no meaning.

Figure C-10 Pipeline Hazard/Interlock by MTHI/MTLO Instructions

Case 4: Preceding Instruction Is 32-bit Division Instruction



A. 32-bit Multiply and Multiply/Add Instructions

Pipeline interlocks till the division instruction is completed.

B. MFHI/MFLO Instructions

Pipeline interlocks because of data dependency.

C. MTHI/MTLO Instructions

Pipeline interlocks till the division instruction is completed.

D. 32-bit Division Instruction

Pipeline interlocks till the division instruction is completed.

E. 64-bit Multiply Instructions

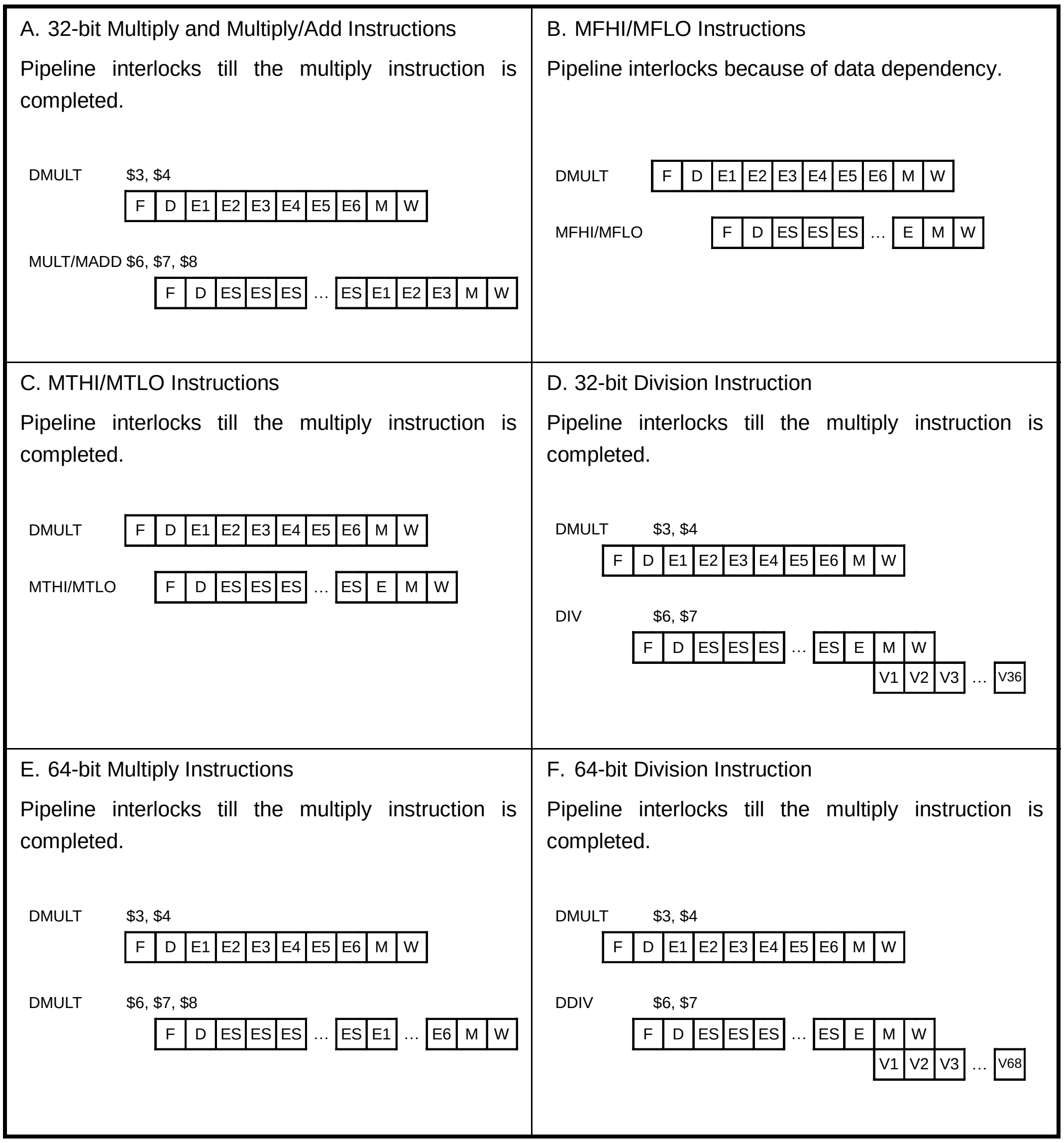
Pipeline interlocks till the division instruction is completed.

F. 64-bit Division Instruction

Pipeline interlocks till the division instruction is completed.

Figure C-11 Pipeline Hazard/Interlock by Division Instructions

Case 5: Preceding Instruction Is 64-bit Multiply Instruction



A. 32-bit Multiply and Multiply/Add Instructions

Pipeline interlocks till the multiply instruction is completed.

DMULT $3, $4
F D E1 E2 E3 E4 E5 E6 M W
MULT/MADD $6, $7, $8
F D ES ES ES … ES E1 E2 E3 M W

B. MFHI/MFLO Instructions

Pipeline interlocks because of data dependency.

DMULT
F D E1 E2 E3 E4 E5 E6 M W
MFHI/MFLO
F D ES ES ES … E M W

C. MTHI/MTLO Instructions

Pipeline interlocks till the multiply instruction is completed.

DMULT
F D E1 E2 E3 E4 E5 E6 M W
MTHI/MTLO
F D ES ES ES … ES E M W

D. 32-bit Division Instruction

Pipeline interlocks till the multiply instruction is completed.

E. 64-bit Multiply Instructions

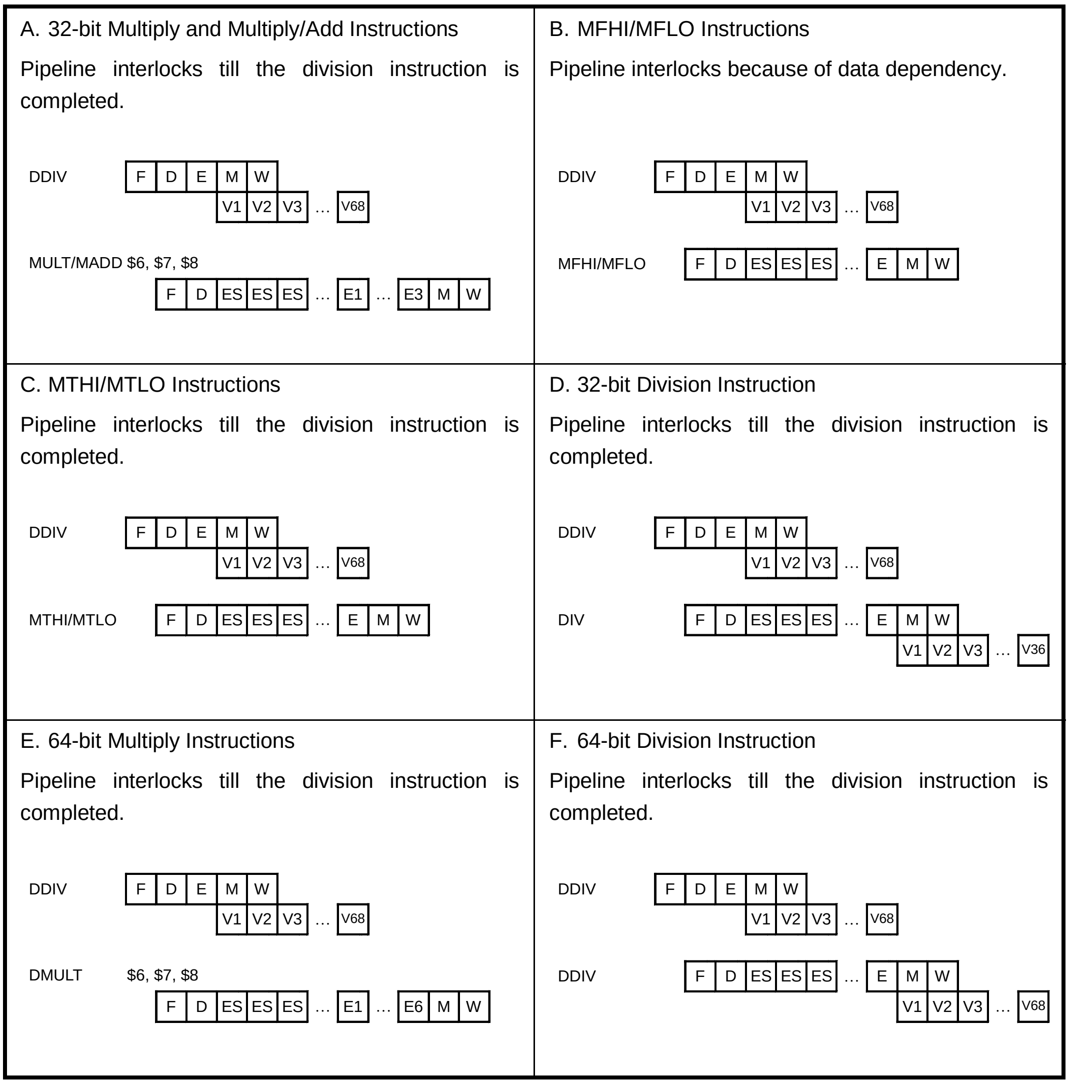
Pipeline interlocks till the multiply instruction is completed.

F. 64-bit Division Instruction

Pipeline interlocks till the multiply instruction is completed.

Figure C-12 Pipeline Hazard/Interlock by Division Instructions

Case 6: Preceding Instruction Is 64-bit Division Instruction



A. 32-bit Multiply and Multiply/Add Instructions

Pipeline interlocks till the division instruction is completed.

B. MFHI/MFLO Instructions

Pipeline interlocks because of data dependency.

C. MTHI/MTLO Instructions

Pipeline interlocks till the division instruction is completed.

D. 32-bit Division Instruction

Pipeline interlocks till the division instruction is completed.

E. 64-bit Multiply Instructions

Pipeline interlocks till the division instruction is completed.

F. 64-bit Division Instruction

Pipeline interlocks till the division instruction is completed.

Figure C-13 Pipeline Hazard/Interlock by Division Instructions

C.1.4 Instructions regarding System Control Co-processor (CP0)

C.1.4.1 MFC0 and MTC0 Instructions

Pipeline interlocks when the MFC0 instruction is followed by the instruction that reads the destination register of MFC0 instruction (Figure C-14).

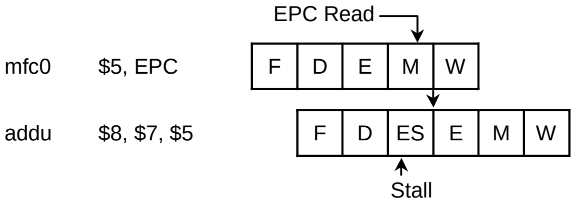


Figure C-14 Pipeline Interlock by MFC0 Instruction

No pipeline hazards occur when the MTC0 instruction is followed by MFC0 instruction because MTC0 writes the destination register in the M stage and MFC0 reads it also in the M stage (Figure C-15).

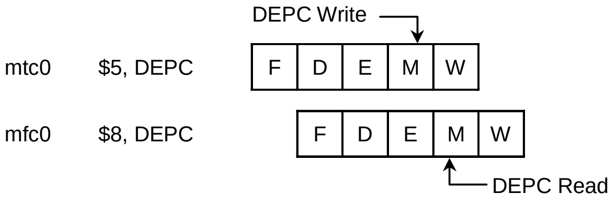


Figure C-15 MTC0 Instruction Followed by MFC0 Instruction

C.1.4.2 ERET Instruction

Unlike a branch or jump instruction, ERET does not execute the next instruction. The changed EPC becomes effective at the second instruction after the MTC0 instruction (Figure C-16).

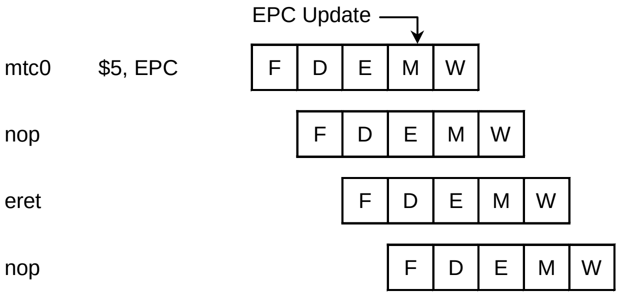


Figure C-16 MTC0 Instruction Followed by ERET Instruction

C.1.4.3 DERET Instruction

The DERET instruction has a branch delay slot, and the debug exception mode is effective till the delay slot instruction[3]. The instruction in the delay slot of DERET must be NOP instruction. Single step exception is disabled till the instruction to which DERET returns the control.

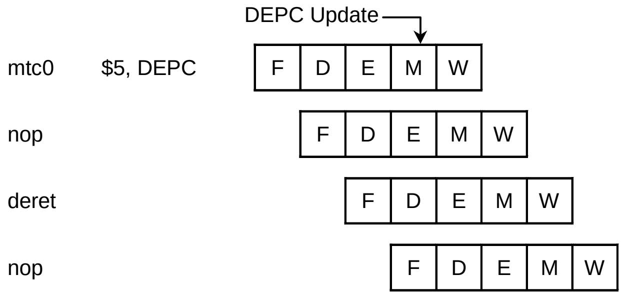


Figure C-17 MTC0 Instruction Followed by DERET Instruction

[3] i.e. DM bit stays one (1) and interrupts and exceptions stay disabled.

C.1.5 Control Bits Change in CP0 Registers by MTC0 Instruction

The following sections describe the timings when the control bits change by the MTC0 instruction become effective.

C.1.5.1 Status Register

CU Bits: Because the co-processor instructions refer the CU bit in the D stage, if either of the two following instructions of the MTC0 instruction is the co-processor instruction, then its result is undefined because the CU bit is undefined (Figure C-18).

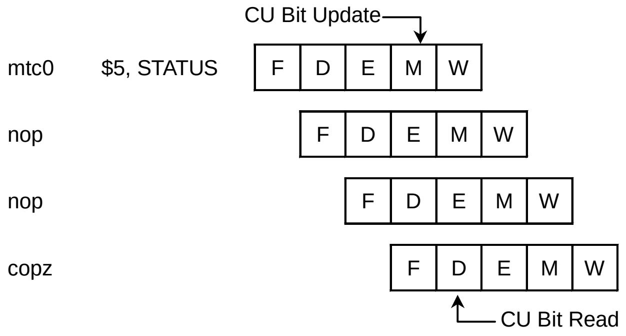


Figure C-18 Hazard regarding the CU Bits

Note that even if the CU bit is changed by the MTC0 instruction during the co-processor bus cycles of the preceding co-processor instruction, this gives no effect on the co-processor instruction currently being executed.

RE Bit: Because the load/store instructions refer the RE bit in the E stage, the change becomes effective at the second instruction after the MTC0 instruction. The result of the load/store instructions immediately after the MTC0 instruction is undefined (Figure C-19).

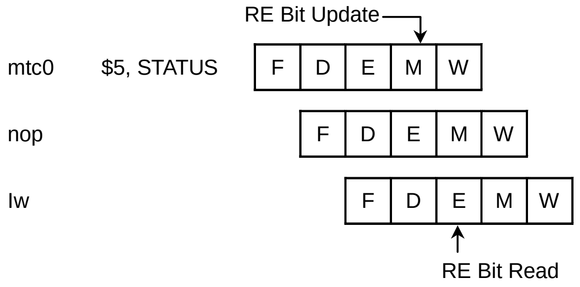


Figure C-19 Hazard regarding the RE Bits

Note that even if the RE bit is changed by the MTC0 instruction during the bus cycles of the preceding load/store instruction, this gives no effect on the load/store instruction currently being executed.

BEV Bit: For the exceptions that occur in the E stage, such as, the address error (AdEL) or the TLB miss (TLBL) exceptions which occurs in the instruction fetch stage, the exception vector base address designated by the changed BEV becomes effective at the second instruction after the MTC0 instruction. If these exceptions occur in the instruction immediately after the MTC0 instruction, the referred value of the BEV bit is undefined[4] (Figure C-20).

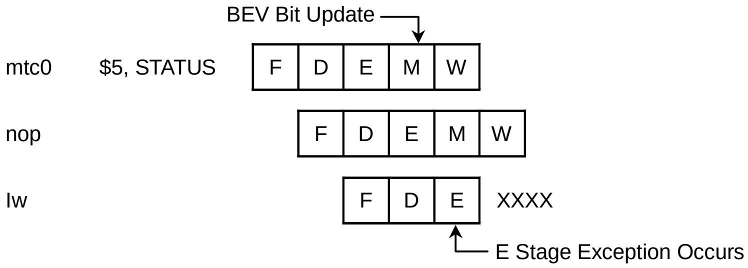


Figure C-20 Hazard regarding the BEV Bits (1)

For the exceptions that occur in the M stage, such as, IBE, DBE, NmI, CpU, Ov, Sys, Bp, RI, AdEL (data), TLBL (data), and TLBS, Mod, and Int, the exception vector base address designated by the changed BEV becomes effective at the instruction immediately after the MTC0 instruction (Figure C-21).

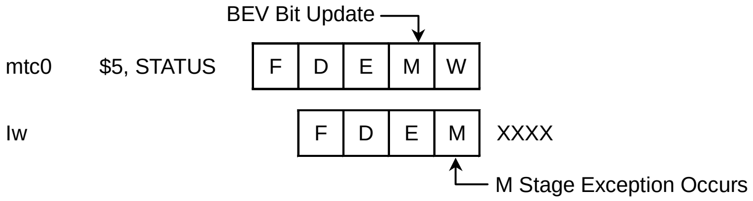


Figure C-21 Hazard regarding the BEV Bits (2)

Note that because the interrupts and the Bus Error exception occurs asynchronously with the instruction execution, the BEV bit value for them is the value which is hold in the BEV bit when they occurs.

IntMask Bits and IE Bit:
When the MTC0 instruction enables the interrupts by changing these bit, then the corresponding interrupts become enabled at the second instruction after the MTC0 instruction[5] (Figure C-22).

On the other hand, when the MTC0 instruction disables the interrupts, the corresponding interrupts become disabled at the instruction immediately after the MTC0 instruction (Figure C-23).

FR Bit: Because the FR bit is changed in the M stage of the MTC0 instruction, new FR bit becomes effective at the third instruction after the MTC0 instruction (Figure C-24).

[4] The new exception vector base address may be effective because of pipeline stall.

[5] They may become enable at the instruction immediately after the MTC0 instruction because of pipeline stall.

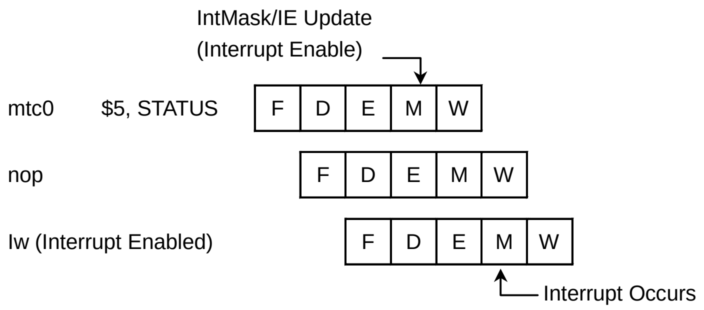


Figure C-22 Hazard regarding the IntMask Bits and IE Bit (1)

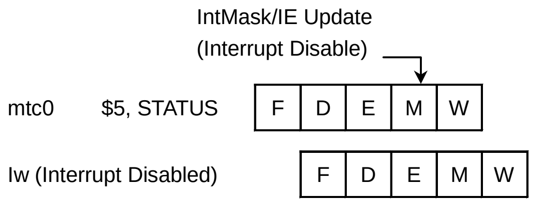


Figure C-23 Hazard regarding the IntMask Bits and IE Bit (2)

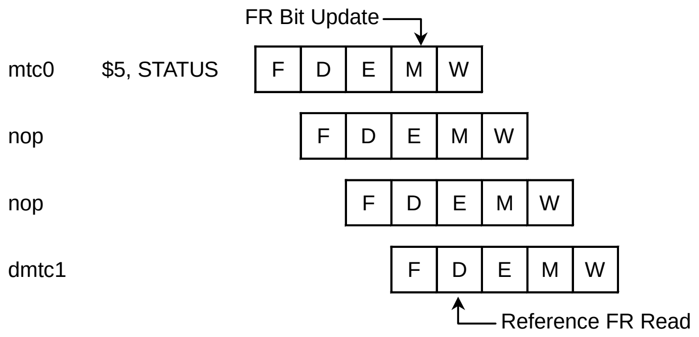


Figure C-24 Hazard regarding the FR Bit

EXL, ERL, KX, SX, UX, KSU Bit:

The modification of these bits become effective at the forth instruction after the MTC0 instruction. On the other hand, new addressing mode for a load/store instruction which is accessing the address in Kernel/Supervisor space or accessing in 64-bit addressing is effective at the second instruction after the MTC0 instruction. If either of the two instructions after the MTC0 instruction is co-processor instruction, result of the instruction is undefined (Figure C-25).

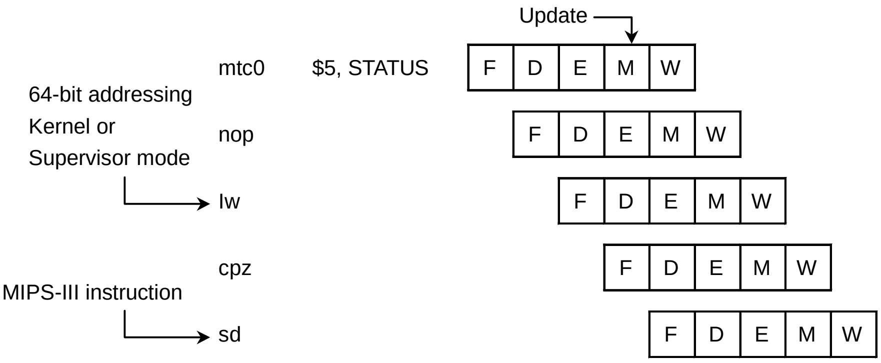


Figure C-25 EXL, ERL, KX, SX, UX, KSU Bit

C.1.5.2 Config Register

ICE# Bit: The MTC0 instruction may change the ICE# bit during the instruction cache streaming. In this case, the old ICE# bit are effective for the instructions during the streaming (Figure C-26).

```
      mtc0  $5, Config       ; update ICE# bit
      nop
      beq   $0, $0, L1       ; stop instruction streaming
      nop
L1:   lw    $2, 0 ($0)       ; new ICE# bit is effective
```

Figure C-26 ICE# Bit update

DCE# Bit: The changed DCE# becomes effective at the second instruction after the MTC0 instruction. The DCE# bit is undefined at the instruction immediately after the MTC0 instruction. Note that the MTC0 instruction may change the DCE# bit during the data cache refill. In this case, the hardware interlock waits updating the DCE# bit till the data cache refill finishes.

K0 Bit: The modification of these bits becomes effective at the forth instruction after the MTC0 instruction, the result of the instruction in Kseg0 address space is undefined if they executed as first, second or third instruction after the MTC0 instruction. On the other hand, the modification of these bits are effective at the third instruction after MTC0 instruction. New addressing mode for a load/store instruction accessing the Kseg0 address space is undefined if the instruction executed as first or second instruction after MTC0 instruction.

C.2 Pipeline Behavior on Cache Miss

This section describes the pipeline behavior on cache miss.

C.2.1 Instruction Cache Miss

Instruction cache miss is detected in F stage and it is immediately followed by a cache refill cycle (Figure C-27).

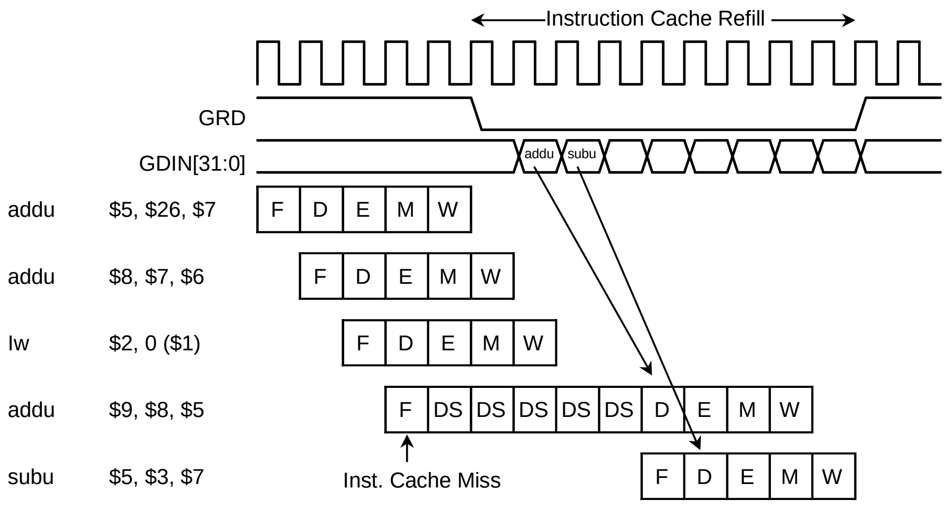


Figure C-27 Streaming on Instruction Cache Refill Cycle in 32-bit GBus mode

On cache miss, the fetched instructions are immediately decoded and executed before completion of refill cycle so that the pipeline resumes the execution of instruction stream as shown in Figure C-27. This is so called *streaming*[6] and its refill cycle is called *stream cycle*.

When the branch or jump instruction is executed during the stream cycle, streaming will be terminated which means refill cycle will completed but the fetched instructions after the branch delay slot won't be executed. The pipeline will stall until the instruction at the branch or jump target is fetched. (Figure C-28).

[6] No streaming in 64-bit GBus mode with 1:1 of GBus clock rate. TX49 executes one instruction per clock cycle even if two instructions are fetched in one cycle. In this case, fetched instruction won't be executed until the refill cycle completes.

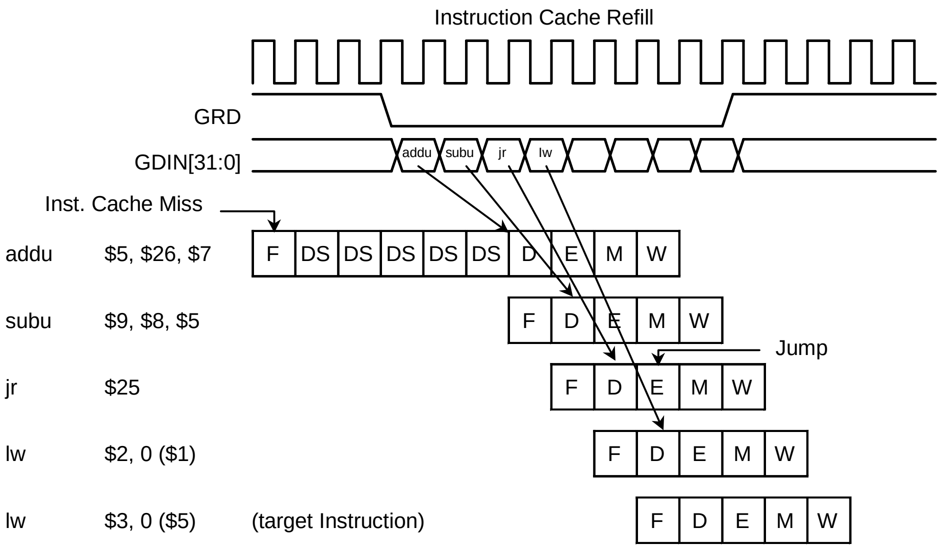


Figure C-28 Branch/Jump Instruction during Stream Cycle in GBus 32-bit Mode

C.2.2 Data Cache Miss

The data cache miss is detected in the M stage of load instruction and it is immediately followed by a cache refill cycle. Non-blocking load mechanism implemented in TX49 data cache allows the following instruction stream to be executed without waiting for the completion of data cache refill if there is no data dependancy between the load and the following instructions.

The pipeline will stall at E-stage of the instruction which use the refilled data as its source until the data is loaded. (Figure C-29).

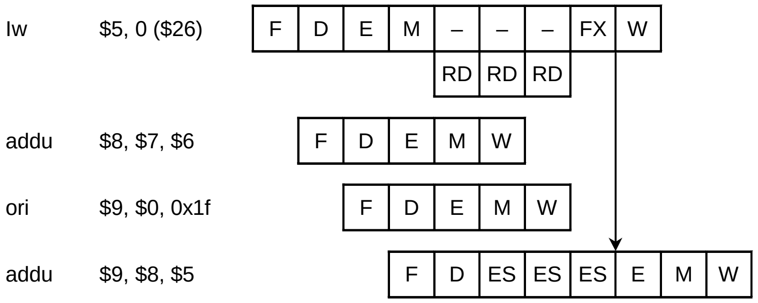


Figure C-29 Pipeline Interlock by Cache Miss

The pipeline also interlocks when a load/store instruction is issued during the data cache refill cycle because of the resource (i.e. data cache) conflict (Figure C-30).

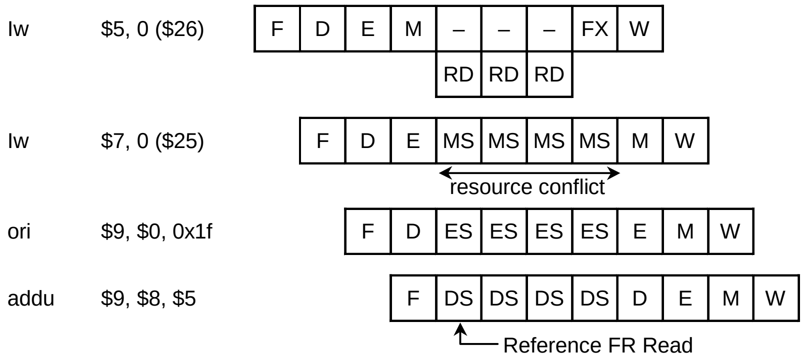

lw $5, 0 ($26) F D E M – – – FX W
RD RD RD
lw $7, 0 ($25) F D E MS MS MS MS M W
resource conflict
ori $9, $0, 0x1f F D ES ES ES ES E M W
addu $9, $8, $5 F DS DS DS DS D E M W
Reference FR Read

Figure C-30 Load Instruction during the Data Cache Refill Cycle

It is possible that the conflict at W-stage occurs between load instruction and one of the following instructions if the load instruction causes cache refill cycle. This situation is shown in Figure C-31.

In this case, W-stage of load instruction takes precedence resulting in one cycle stall at M-stage of the addu instruction.

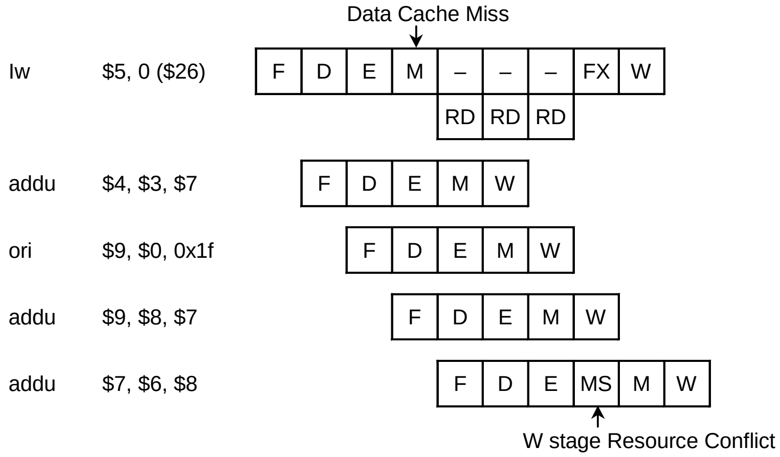

Data Cache Miss
lw $5, 0 ($26) F D E M – – – FX W
RD RD RD
addu $4, $3, $7 F D E M W
ori $9, $0, 0x1f F D E M W
addu $9, $8, $7 F D E M W
addu $7, $6, $8 F D E MS M W
W stage Resource Conflict

Figure C-31 W stage Pipeline Register Conflict

If the instruction fetch cycle is requested during the data cache refill cycle, the data cache refill completes first followed by the instruction fetch cycle (Figure C-32).

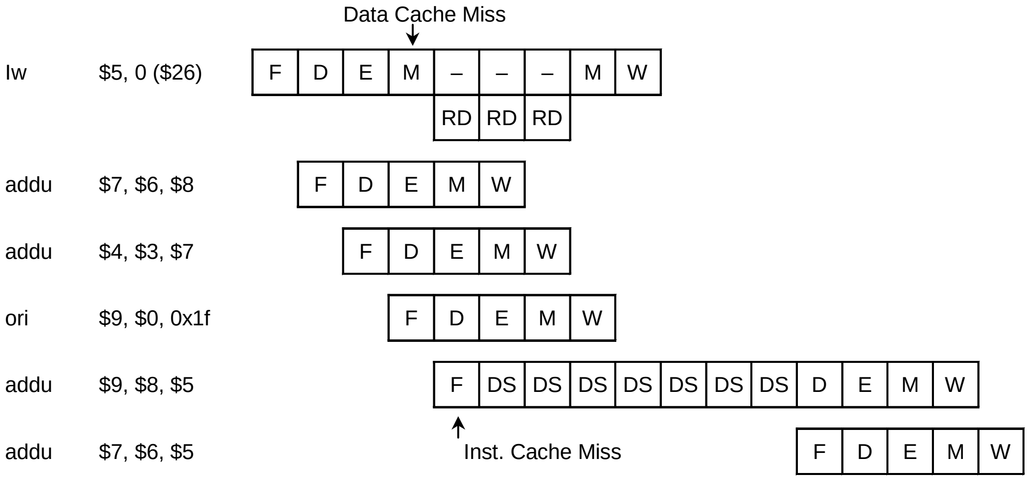

Data Cache Miss
lw $5, 0 ($26) F D E M – – – M W
RD RD RD
addu $7, $6, $8 F D E M W
addu $4, $3, $7 F D E M W
ori $9, $0, 0x1f F D E M W
addu $9, $8, $5 F DS DS DS DS DS DS DS D E M W
addu $7, $6, $5 F D E M W
Inst. Cache Miss

Figure C-32 Instruction Cache Miss during the Data Cache Refill Cycle

C.3 Pipeline Behavior in Uncached Area

The pipeline behavior regarding the memory access to an uncached area is similar to that of refill cycle sequence caused by the cache miss.

C.3.1 Data Read from Uncached Area

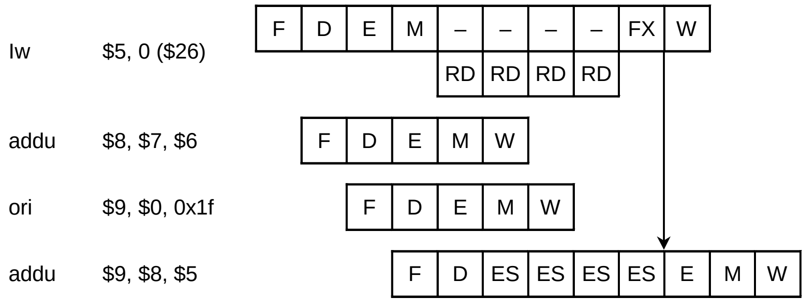


Figure C-33 Data Read from Uncached Area

C.3.2 Instruction Fetch from Uncached Area

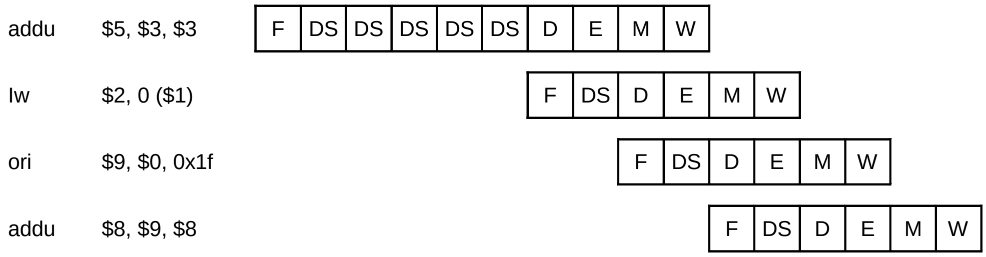


Figure C-34 Instruction Fetch from Uncached Area

C.3.3 Data Write to Uncached Area

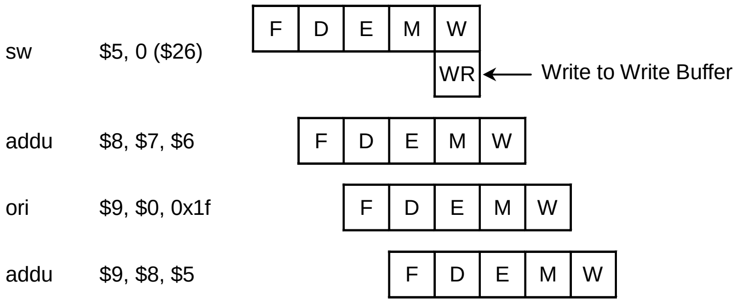


Figure C-35 Data Write to Uncached Area

C.4 Timings on the Exception Handling

This section describes the detail pipeline behavior on exception. When an exception takes place, the instruction on which the exception occurs is aborted. All instructions immediately after that instruction are also aborted and the processor passes the control to the exception handler.

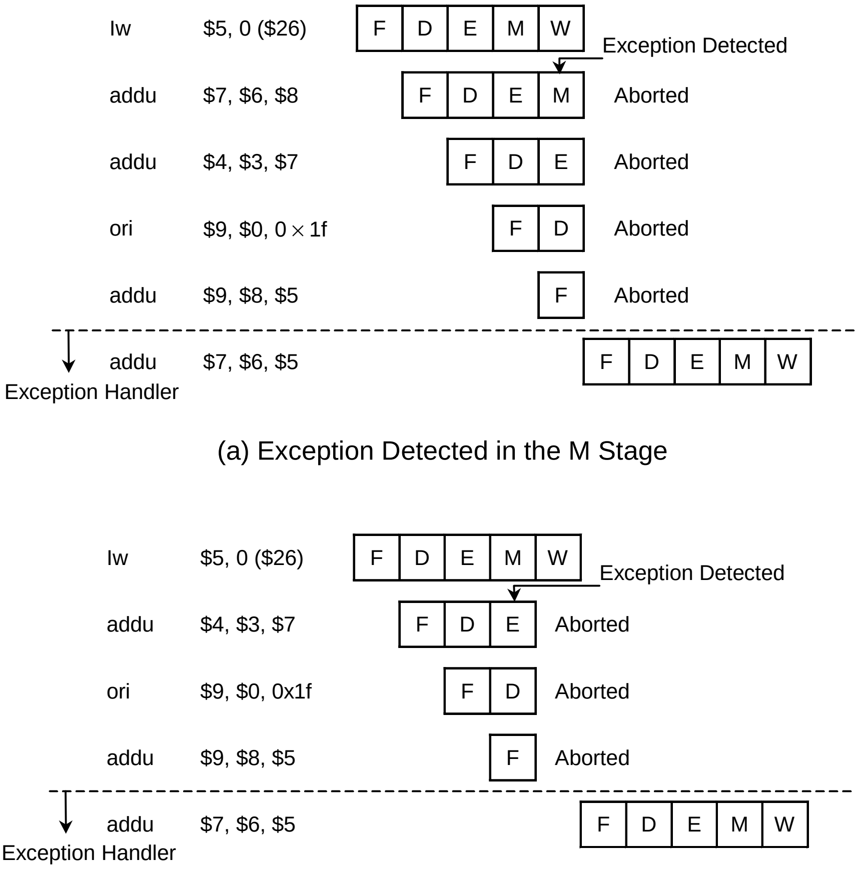
The exceptions normally occur in the M stage, but some of the exceptions occur in the E stage. The exceptions which occur in the E stage are:

- Debug Single Step (DSS)
- Debug Instruction Break (DIB)
- Address Error on Instruction Fetch (AdEL)
- TLB Refill/Invalid on Instruction Fetch (TLBL)

Note that the Reset/Soft Reset Exceptions occur in any stage.

C.4.1 Basic Pipeline Behavior When Exceptions Occur

The following Figure illustrates the pipeline behavior when an exception occurs.


lw $5, 0 ($26) F D E M W
Exception Detected
addu $7, $6, $8 F D E M Aborted
addu $4, $3, $7 F D E Aborted
ori $9, $0, 0 × 1f F D Aborted
addu $9, $8, $5 F Aborted
addu $7, $6, $5 F D E M W
Exception Handler
lw $5, 0 ($26) F D E M W
Exception Detected
addu $4, $3, $7 F D E Aborted
ori $9, $0, 0x1f F D Aborted
addu $9, $8, $5 F Aborted
addu $7, $6, $5 F D E M W
Exception Handler

(a) Exception Detected in the M Stage

(b) Exception Detected in the E Stage

Figure C-36 Pipeline Behavior in Case of Exception

C.4.2 Exceptions during the Execution of Multi-cycle Instructions

As described in the section entitle Multiply, Multiply/Add and Division Instructions, multi-cycle instructions which do not have a destination register file, such as DIV, and the following instructions will be executed in parallel if they do not have data dependency.

If an exception takes place at the instruction being executed in parallel with this type of multi-cycle instructions, the preceding multi-cycle instruction is completed while the instructions after the exception are aborted and the control is passed to the exception handler.

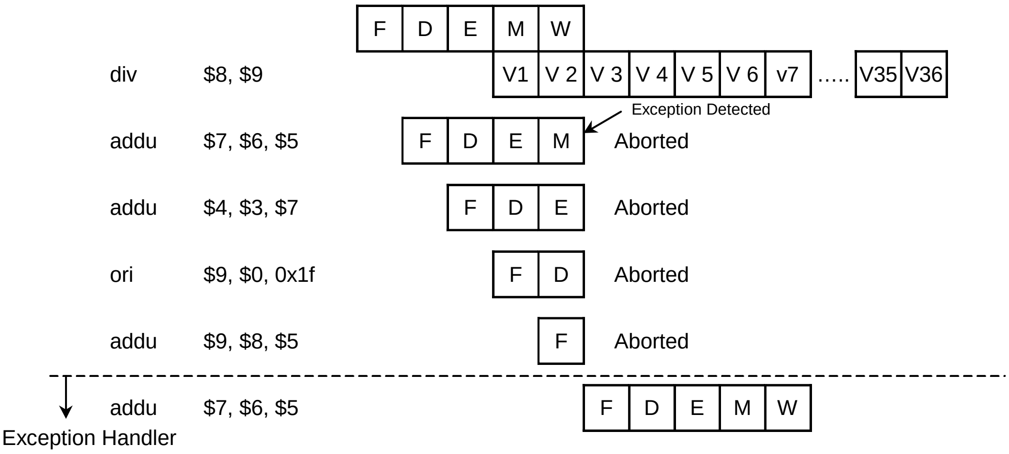

F D E M W
div $8, $9
V1 V 2 V 3 V 4 V 5 V 6 v7 ..... V35 V36
Exception Detected
addu $7, $6, $5 F D E M Aborted
addu $4, $3, $7 F D E Aborted
ori $9, $0, 0x1f F D Aborted
addu $9, $8, $5 F Aborted
addu $7, $6, $5 F D E M W
Exception Handler

Figure C-37 Exception during the Execution of Division Instruction

C.4.3 Exceptions during the Data Cache Refill Cycle

When one of the exceptions occurs at the instruction which is being executed in parallel with data cache refill, the data cache refill cycle is completed while the instructions after the exception are aborted and the control is passed to the exception handler.

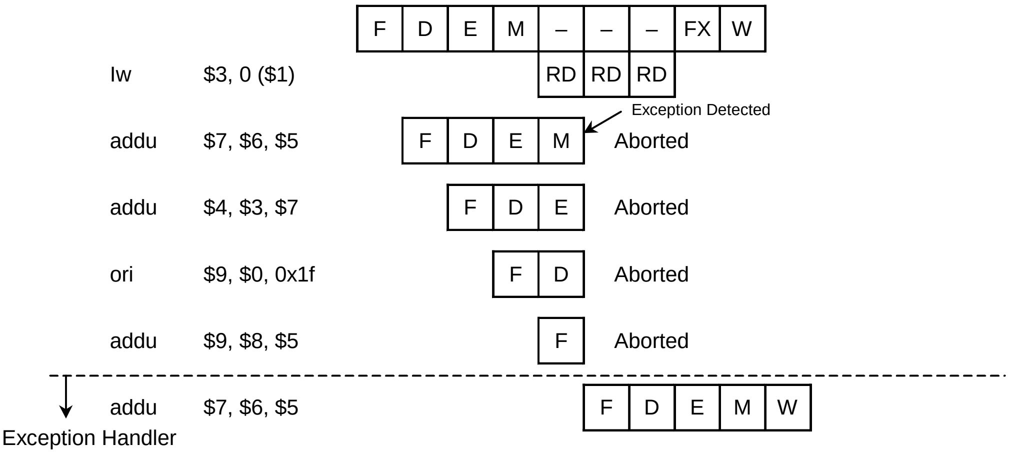

F D E M – – – FX W
lw $3, 0 ($1)
RD RD RD
Exception Detected
addu $7, $6, $5 F D E M Aborted
addu $4, $3, $7 F D E Aborted
ori $9, $0, 0x1f F D Aborted
addu $9, $8, $5 F Aborted
addu $7, $6, $5 F D E M W
Exception Handler

Figure C-38 Exceptions during the Data Cache Refill Cycle (1)

However, when one of the fatal exceptions, such as Bus Error or Reset occurs, the refill cycle is also aborted and the control is passed to the exception handler.

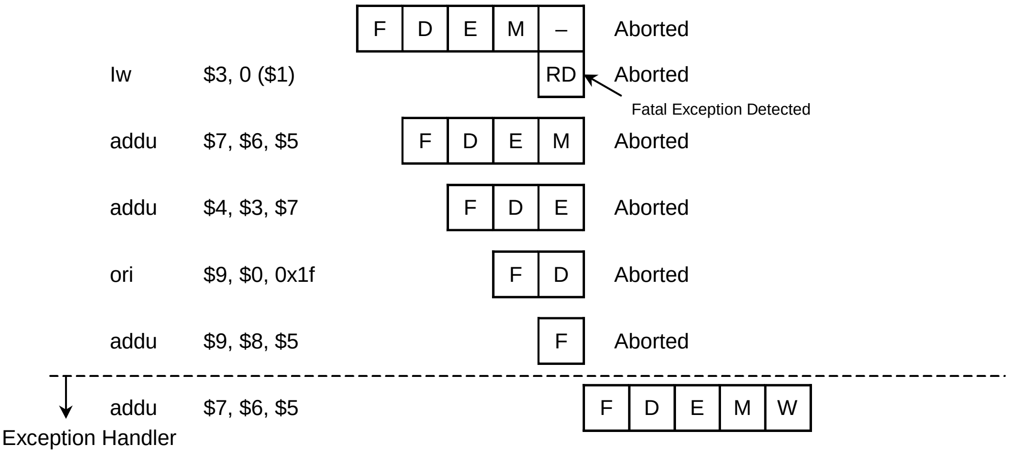


Figure C-39 Exception during Data Cache Refill Cycle (2)

Appendix D: G-Bus Overview

D.1 G-Bus Operation

The G-Bus has a 36-bit address bus and a 64-bit data bus. Byte and halfword transfers can occur in any byte lane, depending on how GBE[7:0]* are driven.

The G-Bus speed can be divided by 2, 2.5, 3 or 4 relative to the CPU full speed. Selection of which G-Bus speed to use is determined by the value of GCRATE[1:0] while GCOLDRESET is asserted. Correct operation is not guaranteed if GCRATE[1:0] changes while the TX49 is running.

The TX49 supports four different types of bus transactions: single-read, burst-read, single-write and burst-write. When a bus transaction starts, GBSTART* is asserted for one GBUSCLK cycle, regardless of the type of the transaction. Peripheral logic must sample GBSTART* to recognize the beginning of a bus cycle. It should be noted that when multiple read or write transactions occur back-to-back, GRD* or GWR* remains asserted until the last transaction is completed; therefore, GRD* and GWR* can not be used to detect the beginning of a bus cycle.

During a read operation, the TX49 samples GACK* with the rising edge of GBUSCLK. When it is detected as asserted, the TX49 captures the data on GDTM at the next rising edge of GBUSCLK. If the bus transaction is a burst-read, the TX49 also automatically increments the address value.

During a write operation, the TX49 samples GACK* with the rising edge of GBUSCLK. When it is detected as asserted during a single-write, the TX49 terminates the current bus transaction at the next rising edge of GBUSCLK. If the bus transaction is a burst-write, the TX49 goes ahead with the next write, automatically incrementing the address value.

GLAST* indicates the completion of a bus cycle. Peripheral logic must sample GLAST* to terminate a bus transaction.

D.2 Types of G-Bus Arbitration

One important feature of the TX49 is its enhanced bus arbitration flexibility. This section introduces two types of bus arbitration: Snoop & Transfer (ST) concurrency and Execute & Transfer (ET) concurrency. ST concurrency causes the TX49 to stall the processor pipeline while allowing the internal data cache to be snooped during DMA transfers. In contrast, ET concurrency allows the processor core to continue execution out of the internal cache during external bus mastership; ET concurrency does not allow data cache snooping.

D.2.1 Snoop & Transfer (ST) Concurrency

In systems in which main memory is accessed by DMA, it must be ensured that the internal data cache of the TX49 always has the most recent data and is not in possession of stale data. In other words, if the data in main memory has been changed by DMA, the matching cache entries in the TX49 must be marked as "modified" (i.e., invalidated). ST concurrency allows the TX49 to "snoop" DMA's access to main memory and check for a matching data cache entry. Figure D-1 illustrates this feature. During an ST concurrency operation, the TX49 stalls the processor pipeline.

An alternate bus master asserts either GHPSREQ* or GSREQ* to request bus mastership for an ST concurrency operation. Once GHPSREQ* or GSREQ* is detected, the TX49 will flush the internal write buffer before granting the bus to the requesting master; GHPSGNT* or GSGNT* is asserted to indicate that the bus has been granted.

While GHPSGNT* or GSGNT* is asserted, the TX49 continually samples GSNOOP* with the rising edge of GBUSCLK. When GSNOOP* is recognized as asserted, the TX49 captures the address on GATM[35:5] and compares it to the addresses of all data items held in the data cache. If the snoop address hits in the data cache, the cache entry is invalidated. GSNOOP* is valid only when either GHPSGNT* or GSGNT* is asserted.

The internal data cache of the TX49 can employ either the write-through or write-back policy. The write-back data cache does not provide support for snooping. When the write-back option is selected, GHPSREQ* and GSREQ* can not be used.

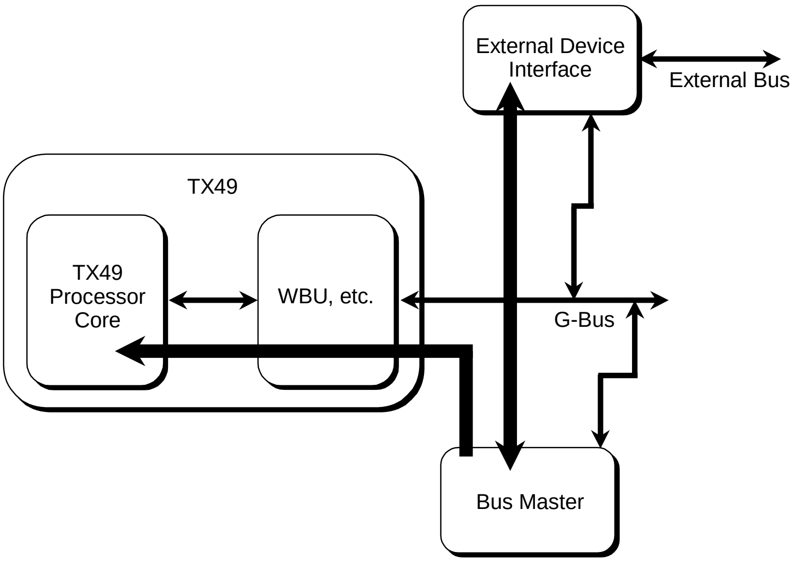


Figure D-1 ST Concurrency

D.2.2 Execute & Transfer (ET) Concurrency

Figure D-2 illustrates ET concurrency. Whereas ST concurrency causes the TX49 to stall the processor pipeline, ET concurrency allows the processor to continue execution out of the internal cache during external bus mastership. However, it does stall when there is a need for a cache refill. Also, if the write buffer is full, additional stores will stall until there is room for them in the write buffer.

ET concurrency is recommended for the following cases:

- when the internal data cache is programmed for write-back mode
- when performing DMA transfers to an uncached address space even if the internal data cache is programmed for write-through mode

An alternate bus master asserts either GHPGREQ* or GREQ* to request bus mastership for an ET concurrency operation. Once GHPGREQ* is detected and the bus is free, the TX49 will grant the bus to the requesting master. GHPGGNT* or GREQ* is asserted to indicate that the bus has been granted to the master. If the bus is busy, the TX49 will relinquish the bus after it completes the current bus cycle. GHPGREQ* and GREQ* are sampled with the rising edge of GBUSCLK.

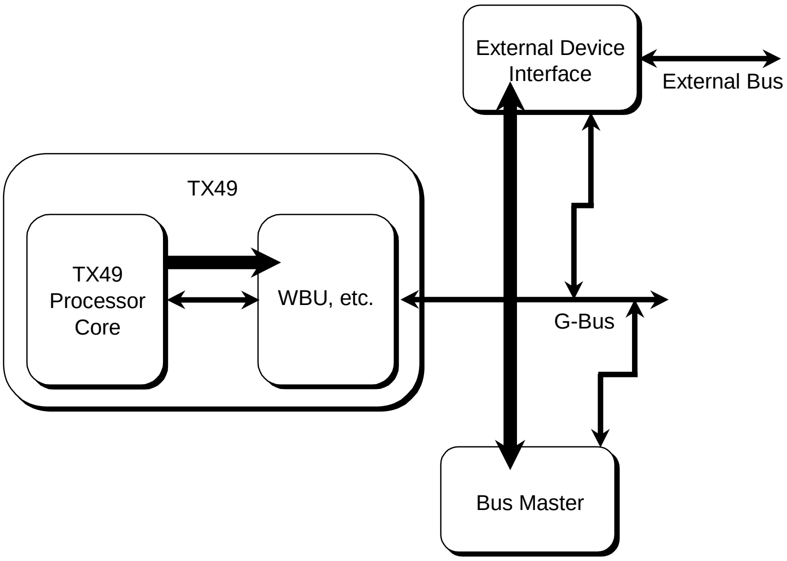


Figure D-2 ET Concurrency

Table D-1 summarizes the differences between ST and ET concurrency.

Table D-1 ST Concurrency vs. ET Concurrency

	ST Concurrency	ET Concurrency
Handshake Signals	Bus request signal: GHPSREQ* Bus grant signal: GHPSGNT* Bus request signal: GSREQ* Bus grant signal: GSGNT*	Bus request signal: GHPGREQ* Bus grant signal: GHPGGNT* Bus request signal: GREQ* Bus grant signal: GGNT*
Data Cache Snooping	Accepted by assertion of GSNOOP* (Not supported in write-back mode)	Not Supported
Stores to the Write Buffer	Disabled	Enabled
Usage Example	When an external bus master performs store operations to a memory space mapped to the data cache (i.e., When data cache snooping is necessary)	• When an external bus master transfers data over the G-Bus without performing a snoop operation. • When the data cache employs the write-back policy
Maximum Bus Control (Request-to-Grant) Latency	Remaining current bus cycle + write buffer flushing* + dta read bus cycle already issued internally + instruction fetch bus cycle already issued internally	

* During an ET concurrency operation, the write buffer is flushed only when the write buffer contains uncached store data which has not yet been written to memory and the TX49 issues an uncached read request to the target address of one of the write buffer entries.

Appendix E: Differences From TX4955A,TX4300 and TX4600

Item	TX4955A	TX4300	TX4600
Datapath	64	64	64
ISA	MIPS I, II, III +MADD, +Debug +PREF	MIPS I, II,III	MIPS I, II, III
Pipeline	5	5	5
MMU	TLB	TLB	TLB
JointTLB	48 double	32 double	48 double
I-TLB	2 entry	2 entry	2 entry
D-TLB	4 entry	No	4 entry
Page Size	4 K-16 MB	4 K-16 MB	4 K-16 MB
Shutdown	No-TS	Yes	No-TS
V.A. Size	40	40	40
P.A. Size	36	32	36
I-cache			
Size	32 KB	16 KB	16 KB
Associate.	4-way	Dir.-map	2-way
Lock	Yes	No	No
Snoop	No	No	No
Index	V	V	V
Tag	P	P	P
Line	32 B	32 B	32 B
Parity	No	No	Yes
D-cache			
Size	32 KB	8 KB	16 KB
Associate.	4-way	Dir.-map	2-way
Lock	Yes	No	No
Write Policy	W.-back/-through	W.-back	W.-back/-through
Snoop	No	No	No
Index	V	V	V
Tag	P	P	P
Line	32 B	16 B	32 B
Parity	No	No	Yes

Item	TX4955A	TX4300	TX4600
WriteBuffer	4A/D pairs	4A/D pairs	4A/D pairs
FPU (CP1)	FPU Hard Single Double	Shared w/ IU Single Double	FPU Hard Shared w/ I-mul/div Single Double
Debug Support Unit	Yes	No	No
MPU Bus I/F	SysAD 32-bit A/D multiplexed	SysAD 32-bit A/D multiplexed	SysAD 64-bit A/D multiplexed
Sys.Clock Ratio:			
1:1	No	No	No
2:1	Yes	Yes	Yes
2.5:1	Yes	No	No
3:1	Yes	Yes	Yes
4:1	Yes	No	Yes
5:1	No	No	Yes
6:1	No	No	Yes
7:1	No	No	Yes
8:1	No	No	Yes
JTAG	Yes	Yes(No func.)	No
Power Sup.	Internal: 1.5 V External: 3.3 V	3.3 V	3.3 V
Power down Mode	-WAIT Inst. (Halt/Doze)	-Status. Reg. (1/4 PClock)	-WAIT Inst. (Stand-by)
Package	PQFP-160	PQFP-120	PGA-179 HSQFP-208