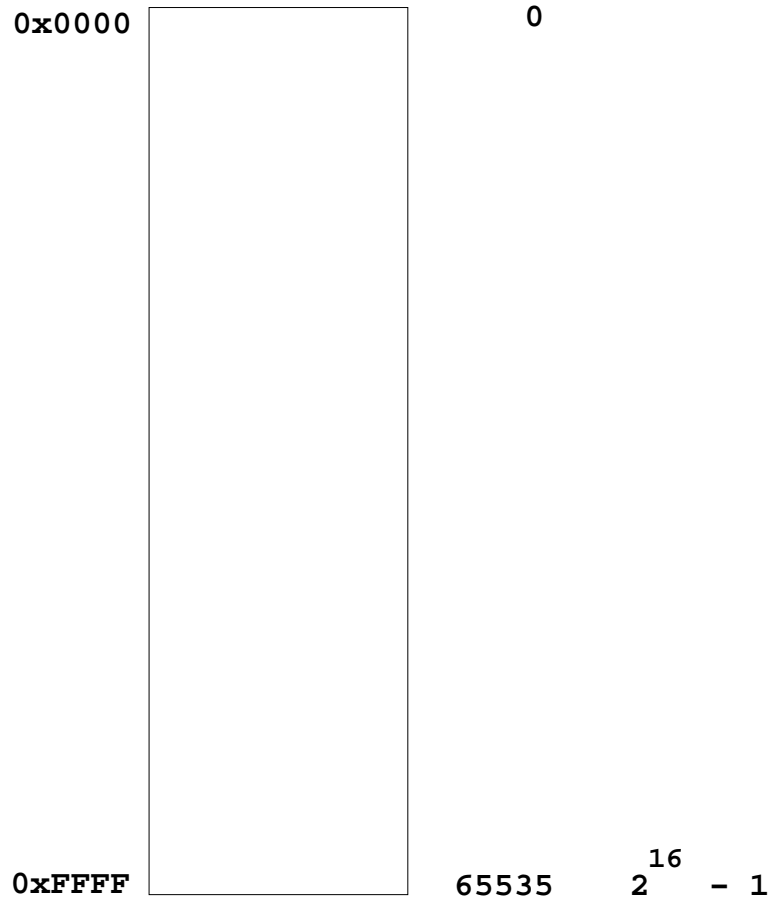


68HCS12 Address Space

- 68HCS12 has 16 address lines
- 68HCS12 can address 2^{16} distinct locations
- For 68HCS12, each location holds one byte (eight bits)
- 68HCS12 can address 2^{16} bytes
- $2^{16} = 65536$
- $2^{16} = 2^6 \times 2^{10} = 64 \times 1024 = 64 \text{ KB}$
- ($1\text{K} = 2^{10} = 1024$)
- 68HCS12 can address 64 KB

68HCS12 Address Space

- Lowest address: $0000000000000000_2 = 0000_{16} = 0_{10}$
- Highest address: $1111111111111111_2 = FFFF_{16} = 65535_{10}$



MEMORY TYPES

- RAM: Random Access Memory (can read and write)
- ROM: Read Only Memory (programmed at factory)
- PROM: Programmable Read Only Memory
(Program once at site)
- EPROM: Erasable Programmable Read Only Memory
(Program at site, can erase using UV light and reprogram)
- EEPROM: Electrically Erasable Programmable Read Only Memory
(Program and erase using voltage rather than UV light)

68HCS12 has: 12 K RAM

4 KB EEPROM

Can erase and reprogram any byte using normal 5V power

256 KB Flash EEPROM

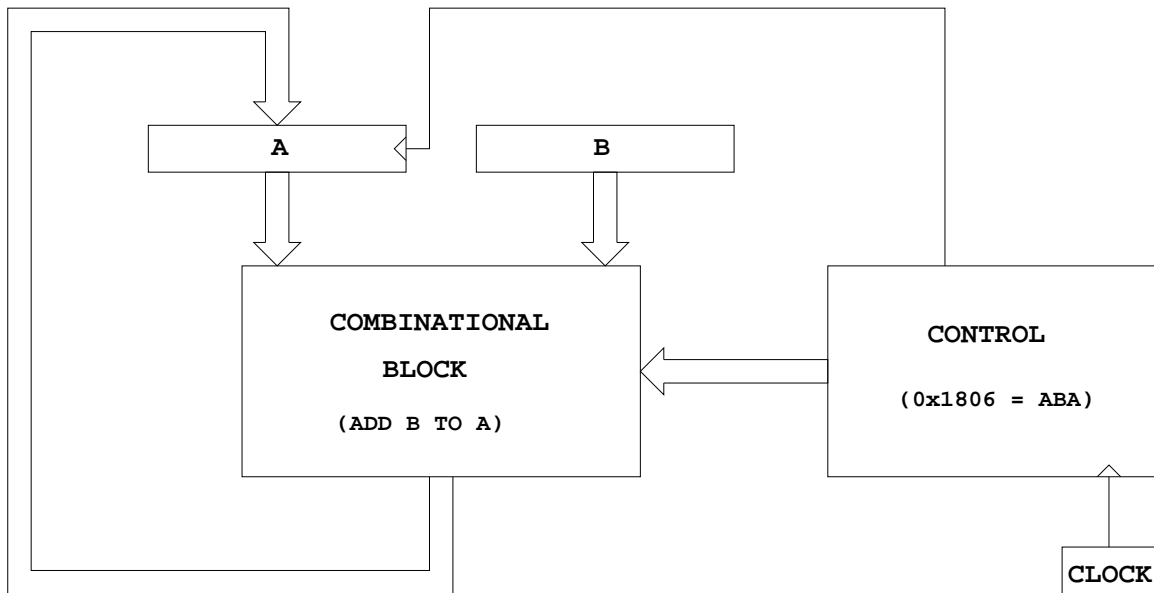
Can erase using external 12V power supply

68HCS12 Address Space

0x0000	Registers	1 K Bytes
0x03FF		
0x0400		3 K Bytes EEPROM
0x0FFF		
0x1000		
0x3BFF	User RAM	11 K
0x3C00		
0x3FFF	D-Bug 12 RAM	1 K
0x4000		
0xFFFF	D-Bug 12 Flash EEPROM	

68HCS12 ALU

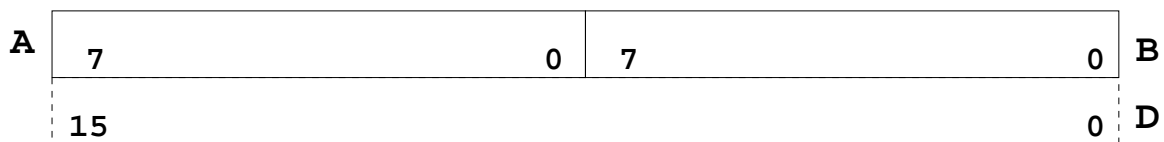
- Arithmetic Logic Unit (ALU) is where instructions are executed.
- Examples of instructions are arithmetic (add, subtract), logical (bitwise AND, bitwise OR), and comparison.
- 68HCS12 has two 8-bit registers for executing instructions. These registers are called **A** and **B**.
- For example, the HCS12 can add the 8-bit number stored in **B** to the eight-bit number stored in **A** using the instruction **ABA** (add B to A):



When the control unit sees the sixteen-bit number $0x1806$, it tells the ALU to add **B** to **A**, and store the result into **A**.

68HCS12 Programming Model

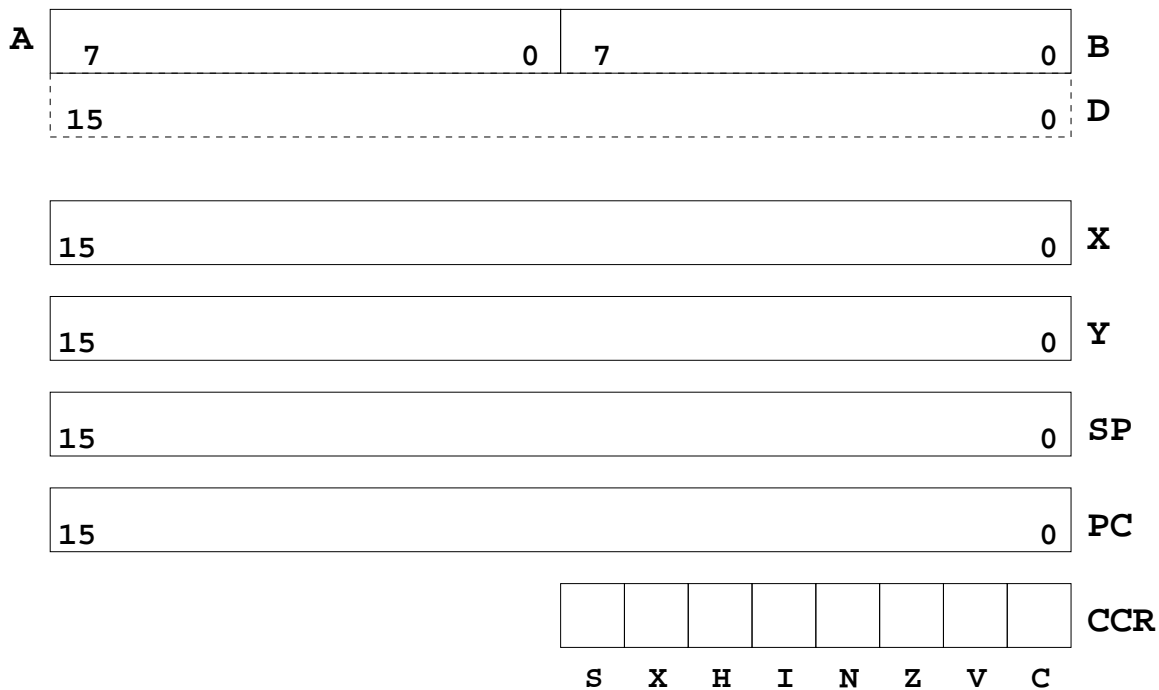
- A Programming Model details the registers in the ALU and control unit which a programmer needs to know about to program a microprocessor.
- Registers **A** and **B** are part of the programming model. Some instructions treat **A** and **B** as a sixteen-bit register called **D** for such things as adding two sixteen-bit numbers. Note that **D** is the same as **A** and **B**.



- The HCS12 can work with 8-bit numbers (bytes) and 16-bit numbers (words).
- The size of word the HCS12 uses depends on the instruction. For example, the instruction LDAA (Load Accumulator A) puts a byte into **A**, and LDD (Load Double Accumulator) puts a word into **D**.

68HCS12 Programming Model

- The 68HCS12 has a sixteen-bit register which tells the control unit which instruction to execute. This is called the Program Counter (**PC**). The number in **PC** is the address of the next instruction the HC12 will execute.
- The 68HCS12 has an eight-bit register which tells the HCS12 about the state of the ALU. This register is called the Condition Code Register (**CCR**). For example, one bit (**C**) tells the HCS12 whether the last instruction executed generated a carry. Another bit (**Z**) tells the HCS12 whether the result of the last instruction was zero. The **N** bit tells whether the last instruction executed generated a negative result.
- There are three other 16-bit registers – **X**, **Y**, **SP** – which we will discuss later.



Some HCS12 Instructions Needed for Lab 1

LDAA	address	Put the byte contained in memory at address into A
STAA	address	Put the byte contained in A into memory at address
CLRA		Clear A ($0 \rightarrow \mathbf{A}$)
INCA		Add 1 to A ($(\mathbf{A}) + 1 \rightarrow \mathbf{A}$)
ABA		Add B to A , store the result in A
ASRA		Shift A right by one bit (keep the MSB the same) This divides a signed byte by 2
LSRA		Shift A right by one bit (put 0 into MSB) This divides an unsigned byte by 2
NEGA		Negate A ($-(\mathbf{A}) \rightarrow \mathbf{A}$)
TAB		Transfer A to B ($(\mathbf{A}) \rightarrow \mathbf{B}$)
SWI		Software Interrupt (Used to end all our HCS12 programs)

A Simple HCS12 Program

- All programs and data must be placed in memory between address 0x1000 and 0x3BFF. For our short programs we will put the first instruction at 0x1000, and the first data byte at 0x1100
- Consider the following program:

```

ldaa $1113 ; Put contents of memory at 0x1113 into A
inca      ; Add one to A
staa $1114 ; Store the result into memory at 0x1114
swi      ; End program

```

- If the first instruction is at address 0x1000, the following bytes in memory will tell the HCS12 to execute the above program:

Address	Value	Instruction
0x1000	B6	ldaa \$1113
0x1001	11	
0x1002	13	
0x1003	42	inca
0x1004	7A	staa \$1114
0x1005	11	
0x1006	14	
0x1007	3F swi	

- If the contents of address 0x1113 were 0xA2, the program would put an 0xA3 into address 0x1114.

A Simple Assembly Language Program.

- It is difficult for humans to remember the numbers (op codes) for computer instructions. It is also hard for us to keep track of the addresses of numerous data values. Instead we use words called mnemonics to represent instructions, and labels to represent addresses, and let a computer program called an assembler to convert our program to binary numbers (machine code).
- Here is an assembly language program to implement the previous program:

```
prog      equ      $1000   ; Start program at 0x1000
data      equ      $1113   ; Data value at 0x1113
result    equ      $1114   ; Result at 0x1114
          org      prog
CODE:     section   .text
          ldaa     data
          inca
          staa     result
          swi
```

- We would put this code into a file and give it a name, such as `test.s`
- Note that `equ`, `org`, and `section` are not instructions for the HCS12 but are directives to the assembler which make it possible for us to write assembly language programs. They are called assembler directives or pseudo-ops. For example the pseudo-op `org` tells the assembler that the starting address (origin) of our program should be `0x1000`.

Assembling an Assembly Language Program

- A computer program called an assembler can convert an assembly language program into machine code.
- The assembler we use in class is from a company called Cosmic.
- To assemble the above program using the Cosmic assembler, we must first create a file called `test.lkf` which tells the assembler where to put things in memory. Our `test.lkf` file would look like this:

```
# Link file for test program
+seg .text -b 0x1000 -n .text # program start address
+seg .data -b 0x1100 -n .data # data start address
test.o # application program
```

- To assemble the program, use the following commands:

```
ca6812 -a -l -xx -pl test.s
clnk -o test.h12 -m test.map test.lkf
chex -o test.s19 test.h12
```

- This will produce a file called `test.s19` which we can load into the 68HCS12.