• **Introduction to the 9S12 Microcontroller**
  - Harvard architecture and Princeton architecture
  - Memory map for a Princeton architecture microprocessor
  - 68HC12 Address Space
  - 68HC12 ALU
  - 68HC12 Programming Model
  - Some 9S12 Instructions Needed for Lab 1
  - A Simple Assembly Language Program
  - Assembling an Assembly Language Program
PRINCETON (VON NEUMAN) ARCHITECTURE

MICROPROCESSOR

MEMORY

IN

OUT

ALU

DATA

INSTRUCTION

CONTROL & ADDR

CONTROL

STATUS

CLOCK

MEMORY MAP

(Princeton Architecture)

Function of memory determined by programmer

PROGRAM

DATA
MC9S12 Address Space

- MC9S12 has 16 address lines
- MC9S12 can address $2^{16}$ distinct locations
- For MC9S12, each location holds one byte (eight bits)
- MC9S12 can address $2^{16}$ bytes
  - $2^{16} = 65536$
  - $2^{16} = 2^6 \times 2^{10} = 64 \times 1024 = 64$ KB
  - (1K = $2^{10} = 1024$)
- MC9S12 can address 64 KB
- Lowest address: 0000000000000000₂ = 0000₁₆ = 0₁₀
- Highest address: 1111111111111111₂ = FFFF₁₆ = 65535₁₀
MEMORY TYPES

**RAM**: Random Access Memory (can read and write)

**ROM**: Read Only Memory (programmed at factory)

**PROM**: Programmable Read Only Memory
(Programmed 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)

**MC9S12** has:

12 KB RAM

4 KB EEPROM (Normally can only access 3 KB)

256 KB Flash EEPROM (Can access 16 KB at a time)
## MC9S12 Address Space

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Registers (Hardware)</td>
<td>1 K Byte</td>
</tr>
<tr>
<td>0x03FF</td>
<td>User EEPROM</td>
<td>3 K Bytes</td>
</tr>
<tr>
<td>0x0400</td>
<td>User RAM</td>
<td>11 K Bytes</td>
</tr>
<tr>
<td>0x0FFF</td>
<td>D-Bug 12 RAM</td>
<td>1 K Bytes</td>
</tr>
<tr>
<td>0x1000</td>
<td>Fixed Flash EEPROM</td>
<td>16k Bytes</td>
</tr>
<tr>
<td>0x3BFF</td>
<td>Banked Flash EEPROM</td>
<td>16k Bytes</td>
</tr>
<tr>
<td>0x3C00</td>
<td>Fixed Flash EEPROM</td>
<td>16k Bytes</td>
</tr>
<tr>
<td>0x3FFF</td>
<td>(D-Bug 12)</td>
<td></td>
</tr>
</tbody>
</table>
MC9S12 ALU

- Arithmetic Logic Unit (ALU) is where instructions are executed.

- Examples of instructions are arithmetic (add, subtract), logical (bitwise AND, bitwise OR), and comparison.

- MC9S12 has two 8-bit registers for executing instructions. These registers are called A and B.

- For example, the MC9S12 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.
MC9S12 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 MC9S12 can work with 8-bit numbers (bytes) and 16-bit numbers (words).

• The size of word the MC9S12 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**.

• The MC9S12 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 MC9S12 will execute.

• The MC9S12 has an eight-bit register which tells the MC9S12 about the state of the ALU. This register is called the **Condition Code Register** (CCR). For example, one bit (C) tells the MC9S12 whether the last instruction executed generated a carry. Another bit (Z) tells the MC9S12 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 MC9S12 Instructions Needed for Lab 1

LDAA address  puts the byte contained in memory at address into A
STAA address  puts the byte contained in A into memory at address
STAB address  puts the byte contained in B into memory at address
ADDA address  adds the byte in memory address to A, and save result in A
CLRB          clears B (0 ⇒ B)
INCA          adds 1 to A ((A) + 1 → A)
DECB          decrements B by 1 ((B) - 1 → B)
LSRA          shifts A right by one bit (puts 0 into MSB)
               This divides an unsigned byte by 2
ASRA          shifts A right by one bit (keep MSB the same)
               This divides a signed byte by 2
SWI           Software Interrupt (Used to end all our MC9S12 programs)

A Simple MC9S12 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 0x2000, and the first data byte at 0x1000.

• Consider the following program:

  ldaa $1000     ; Put contents of memory at 0x1000 into A
  inca          ; Add one to A
  staa $1001     ; Store the result into memory at 0x1001
  swi           ; End program
• If the first instruction is at address 0x2000, the following bytes in memory will tell the MC9S12 to execute the above program:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000</td>
<td>B6</td>
<td>ldaa $1000</td>
</tr>
<tr>
<td>0x2001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>0x2002</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>0x2003</td>
<td>42</td>
<td>inca</td>
</tr>
<tr>
<td>0x2004</td>
<td>7A</td>
<td>staa $1001</td>
</tr>
<tr>
<td>0x2005</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>0x2006</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>0x2007</td>
<td>3F</td>
<td>swi</td>
</tr>
</tbody>
</table>

• If the contents of address 0x1000 were 0xA2, the program would put a 0xA3 into address 0x1001.

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 programmer called an assembler to convert our program to binary numbers (machine code).

• Here is an assembly language program to implement the previous program:

```assembly
prog:  equ $2000  ; Start program at 0x2000
data:  equ $1000  ; Data value at 0x1000
org    prog
lda prog
inca   input
staa   result
swi

org    data          ; Start of data
input: dc.b $A2
result: ds.b 1
```
• We would put this code into a file and give it a name, such as `main.asm` (assembly language programs usually have the extension .s or .asm).

• Note that `equ`, `org`, `dc.b` and `ds.b` (define constant byte and define storage byte) are not instructions for the MC9S12 but are directives to the assembler which makes 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 0x2000.

**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 a commercial compiler from Freescale called CodeWarrior.

• How to use CodeWarrior is discussed in Lab 1 and in Huang (3.8 Using CodeWarrior).

• The assembler will produce a file called `main.lst`, which shows the machine code generated.

### Freescale HC12-Assembler
(c) Copyright Freescale 1987-2009

<table>
<thead>
<tr>
<th>Abs.</th>
<th>Rel.</th>
<th>Loc</th>
<th>Obj. code</th>
<th>Source line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0000 2000</td>
<td>prog equ</td>
<td>$2000 ; Start program at 0x2000</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0000 1000</td>
<td>data equ</td>
<td>$1000 ; Data value at 0x1000</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td>org prog</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>a002000 B610 00</td>
<td>ldaa input</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>a002003 42</td>
<td>inca</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>a002004 7A10 01</td>
<td>staa result</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>a002007 3F</td>
<td>swi</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td></td>
<td>org data</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>a001000 A2</td>
<td>input: dc.b $A2</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>a001001</td>
<td>result: ds.b 1</td>
<td></td>
</tr>
</tbody>
</table>
This will produce a file called Project.abs.s19.

S06B0000433A5C446F63756D656E74732061642053657474696773
S1051000A20048
S10B2000B61000427A10013F02
S9030000FC

We can load into the MC9S12.

S1051000A20048
S10B2000B61000427A10013F02
S9030000FC

- The first line of the S19 file starts with a S0: the **S0** indicates that it *is the first line*.
- The last line of the S19 file starts with a S9: the **S9** indicates that it *is the last line*.
- The other lines begin with a S1: the **S1** indicates these lines are data to be loaded into the MC9S12 memory.
- Here is the second line (with some spaces added):
  
  S1 0B 2000 B6 1000 42 7A 1001 3F 02

- On the second line, the S1 if followed by a **0B**. This tells the loader that there this line has 11 (0x0B) bytes of data follow.
- The count 0B is followed by **2000**. This tells the loader that the data (program) should be put into memory starting with address 0x2000.
- The next 16 hex numbers B61000427A10013F are the 8 bytes to be loaded into memory. You should be able to find these bytes in the *main.lst* file.
- The last two hex numbers, **0x02**, is a one byte checksum, which the loader can use to make sure the data was loaded correctly.
What will program do?

- **ldaa input**: Load contents of 0x1000 into A
  (0xA2 into A)

- **inca**: Increment A
  (0xA2 + 1 = 0xA3 -> A)

- **staa result**: Store contents of A to address 0x1001
  (0xA3 -> address 0x1001)