Chapter 1

Introduction to 68HC12

Assembly Notation :

	Intel	Motorola
binary	`10010011b`	`%10010011`
decimal	`1478`	`1478`
hexadecimal	`5678h`	`$5678`

SYSC-2001 Review : Computer Hardware Organization

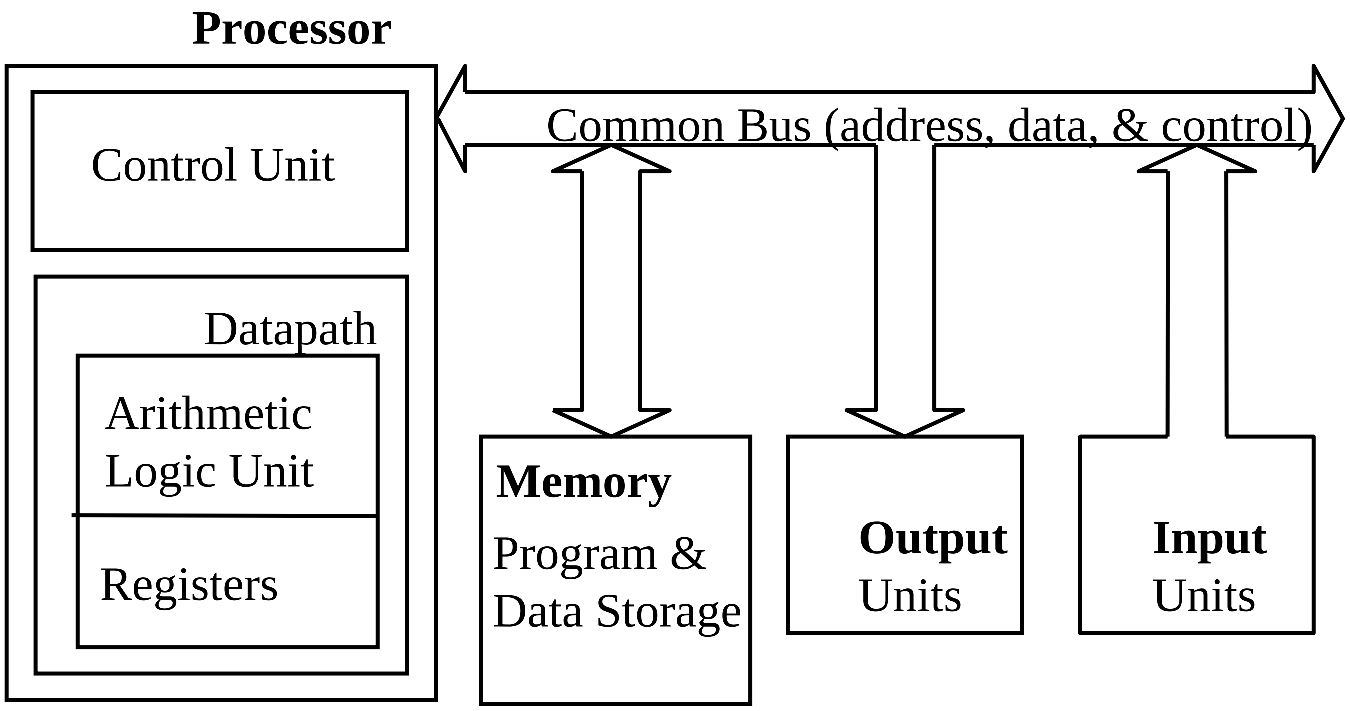


Figure 1.1 Computer Organization

CPU: Microprocessor versus Microcontroller

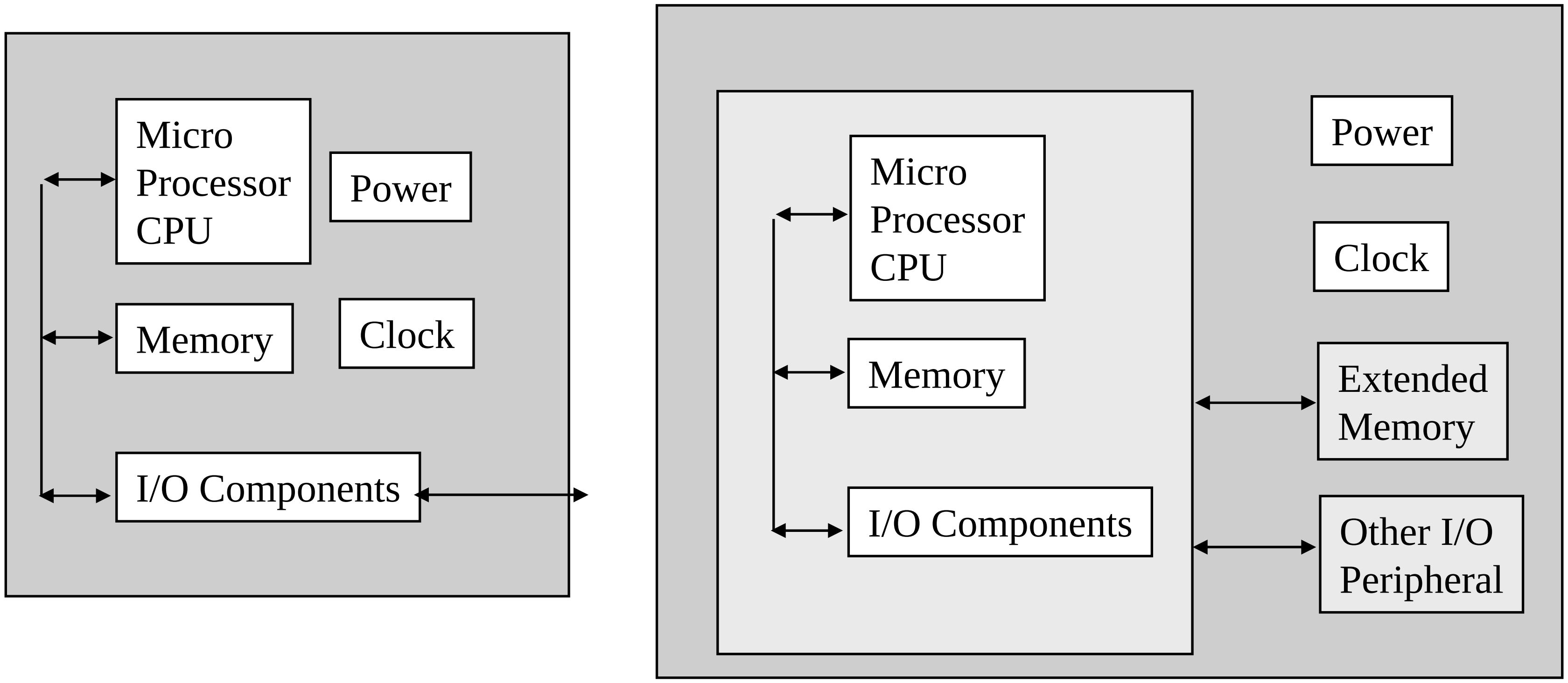
- Microprocessor: processor on a single integrated chip
  - Intel 80x86, Motorola 680x0 families...

- Microprocessors evolved in two general directions:
  - Performance: processing power, data storage
  - Integration: amount of circuits on one chip

- Microcomputer: computer using microprocessor as CPU
  - Combines microprocessor/peripheral chips -> computer *system*
  - i.e.: PC => Intel 80x86 + memory + timer + keyboard + modem (etc.) chips

- Microcontroller: *computer system on a chip*
  - Microprocessor AND peripheral functions implemented on one chip
  - Built-in memory and interface circuits (I/O units)

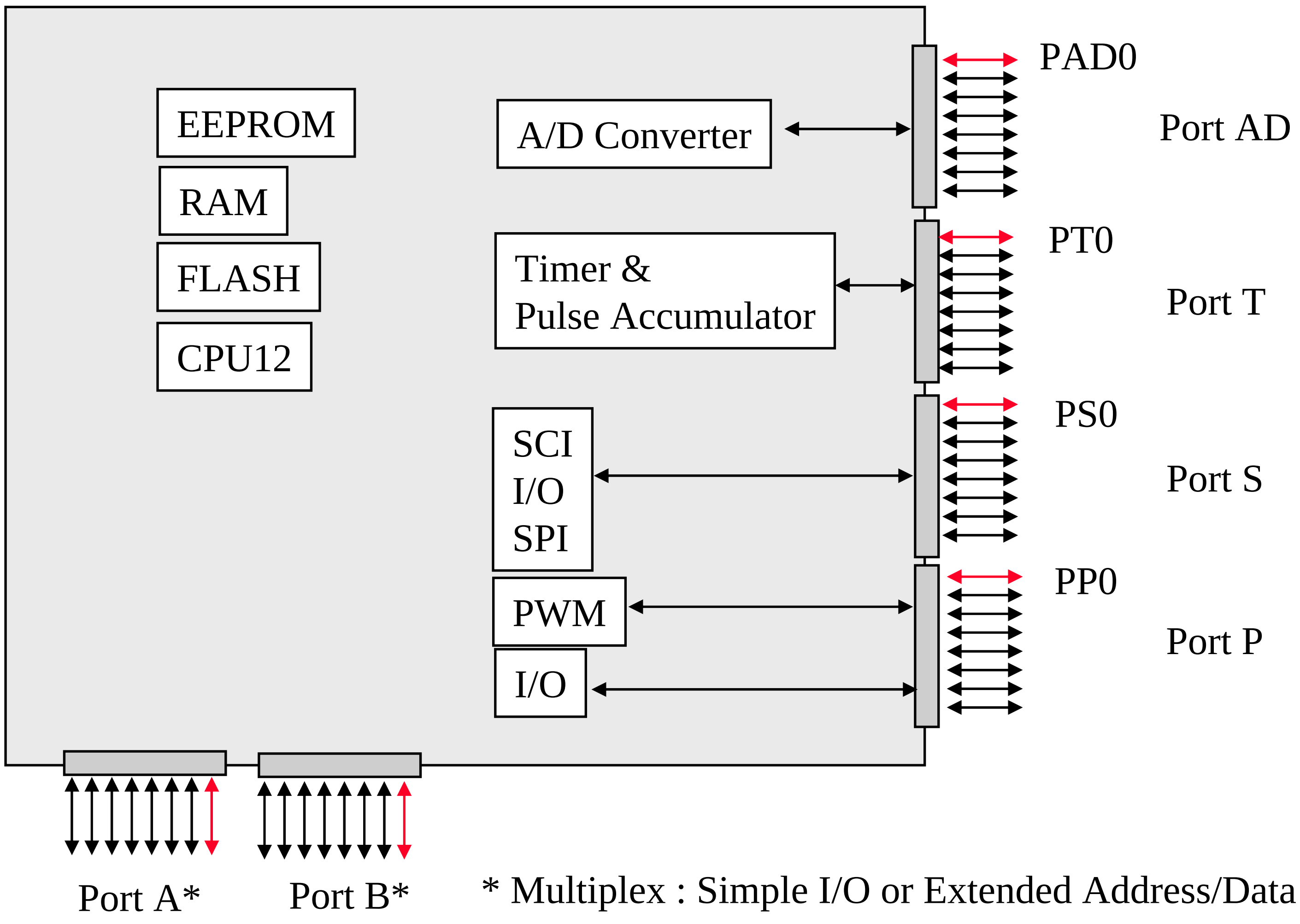
MicroComputer versus MicroController System



Features of the 68HC12 Microcontroller

- 16-bit CPU
- 64 Kb memory space
- 768 bytes to 4 Kb of EEPROM
- 1 Kb to 12 Kb of on-chip SRAM
- 32 Kb to 128 Kb flash memory
- Sophisticated timer functions including: input capture, output compare, pulse accumulators, real-time interrupt, COP timer
- Serial communication interfaces: SCI, SPI, CAN, BDLC
- Background debug mode (BDM)
- 8-bit or 10-bit A/D converter
- Instructions for supporting fuzzy logic function

Block Diagram of MC68HC12



* Multiplex : Simple I/O or Extended Address/Data

The 68HC12 Family

- 68HC12 Family Members
  - Different memory and number/type of on-chip peripheral functions
  - Vary in fabrication methods (different power requirements, environmental tolerances, cost)

Huang Text : **CME-12BC32**

- Made from socket parts
- One SCI channel
- 1 A/D Converter

Our lab : **CML-9S12DP256**

- Made from surface-mount parts
- Two SCI Channels
- 2 A/D Converters

Concepts : Use Text
Specifics : Use Hardware Reference Manuals (posted on website)

HC12 Programmer's Model

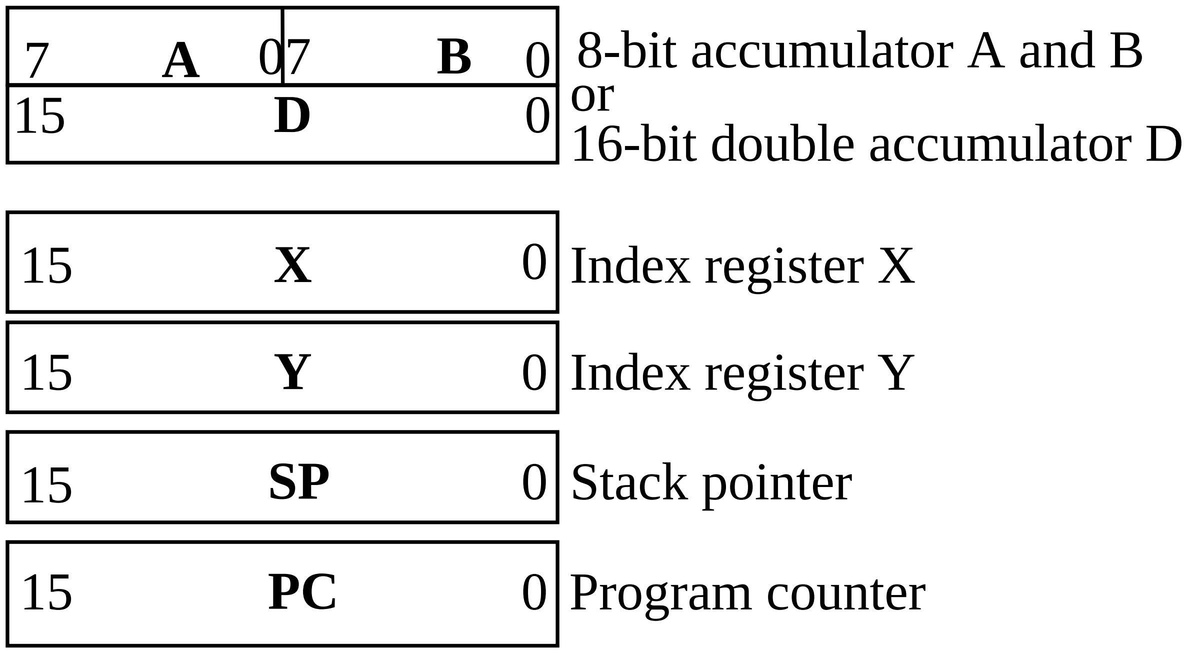


Figure 1.3 MC68HC12 CPU registers.

Condition Code Register

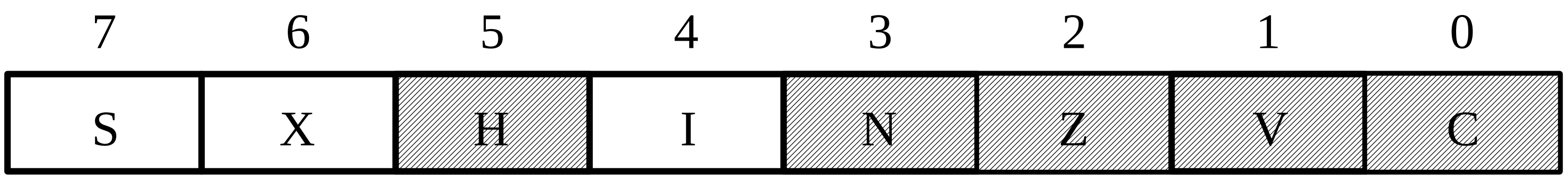


Figure 2.8 Condition code register

C : Carry Flag
Set if carry generated as a result of an operation

V : Overflow Flag
Sets if result of 2’s complement arithmetic operation is out of range

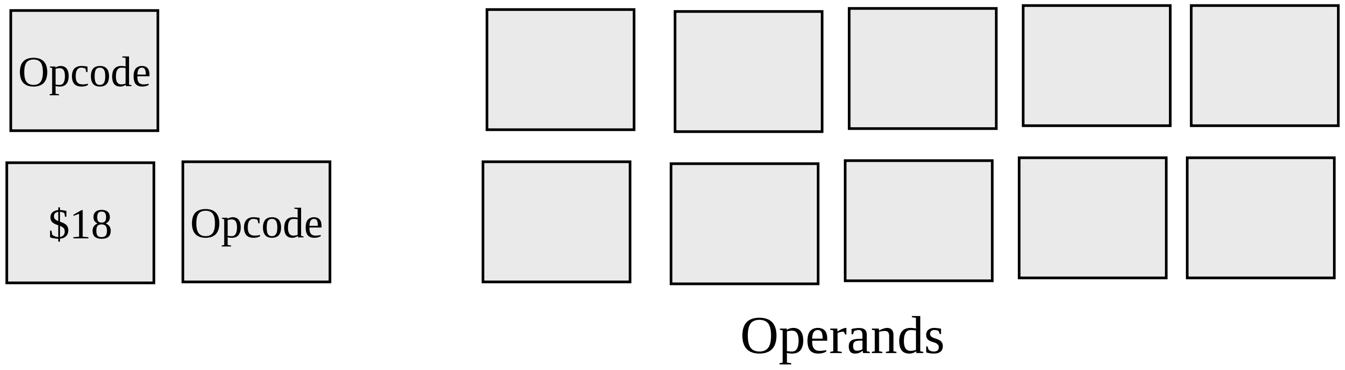
Z : Zero Flag
Set if result of an operation is zero

N : Negative Flag
Set if MSB of the result of an operation = 1

H : Half-carry Flag
Set if carry from lower four bits to upper four bits as a result of an operation

Instruction Set

- 68HC12 instructions: one/two bytes -> **opcode;** zero-five bytes -> **operand** (addressing information).



- **Opcode:** operation to be performed. First byte of two-byte opcode always $18.

- Instruction (opcodes) classified into three groups
    1. Data Transfer
    2. Data Manipulation (Arithmetic and Logic)
    3. Control Flow

Source Code

- Program written in assembly or high-level language

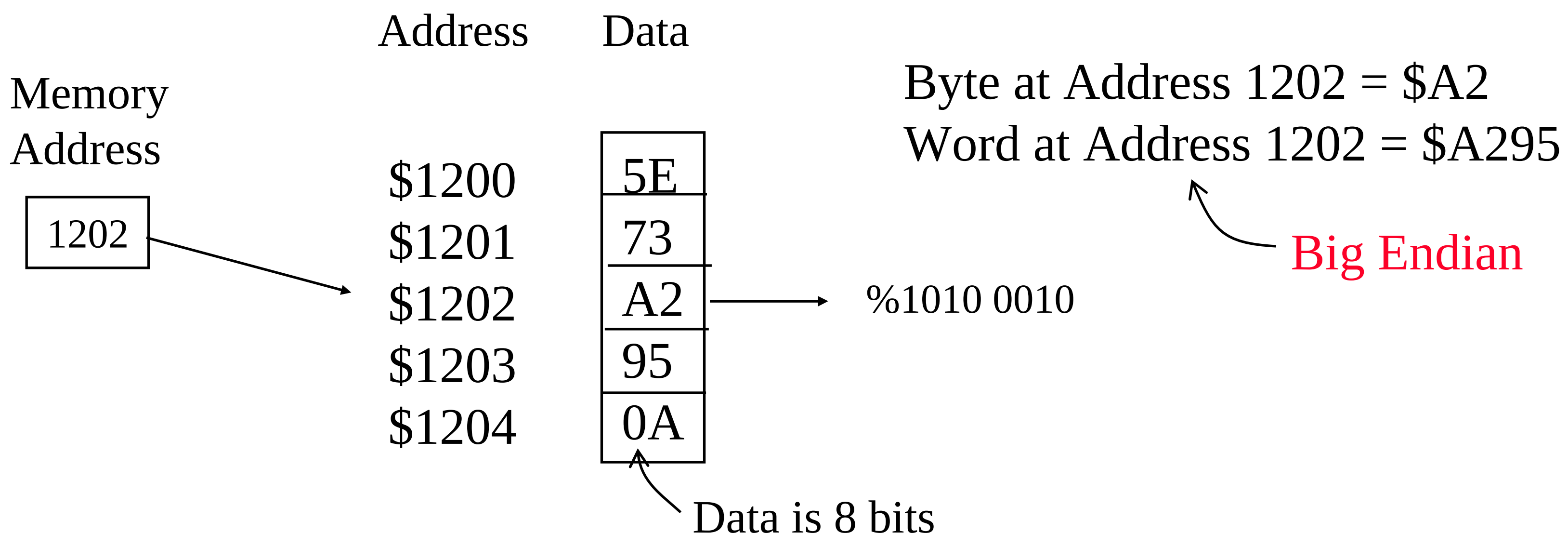
Object Code

- Output of an assembler or compiler
- Executable program in binary format (machine instructions)

```
line     addr.   machine code        (Assembly) source code
------------------------------------------------------------
 1:              = 00001000          org    $1000
 2:      1000    B6 0800             ldaa   $800
 3:      1003    BB 0801             adda   $801
 4:      1006    BB 0802             adda   $802
 6:      1009    7A 0900             staa   $900
                                     end
```

Memory Format of the 68HC12 family

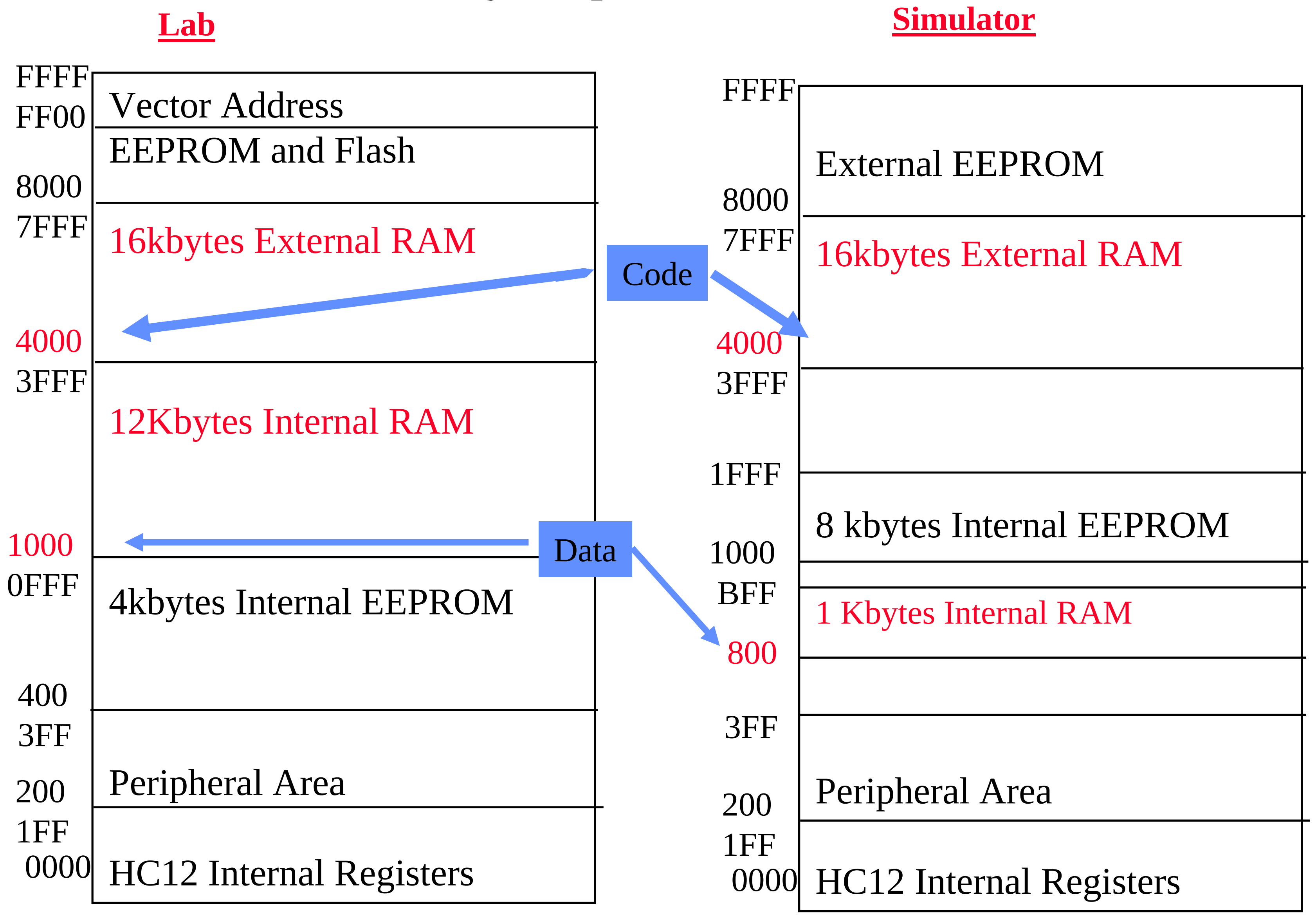
- HC12: 16-bit processor. It can read 16-bit words organized as 8-bit bytes.
  - On Motorola's, words accessed with Big Endian format



MC68HC12 Memory Map

- Different members of the HC12 family: different amounts of on-chip SRAM, EEPROM and flash memory
- Education boards: external SRAM and EEPROM to facilitate program downloading/debugging
- HC12 can only access 64Kbytes of memory (16-bit processor). Larger memory spaces: special “bank-switching” techniques
- Memory mapped to allocate different kinds of data and instructions (single-chip; expanded mode)
- More information about memory mapping: board’s user manual.

MC68HC12 Memory Map for DP256



HC12 Simulator

- Microcontroller simulator: tool (program) that replicates the operation of a microcontroller
  - Learn about/ develop code without having the hardware
- HC12 simulator mimics operation of HC12 microcontroller
  - Displays and changes “registers”
  - Displays and changes “memory”
  - “Executes” code, changing contents of simulated memory/ registers according to the semantics of instructions executed

HC12 Simulator

We will use a freeware simulator available at
http://www.electronikladen.de/hc12

This simulator is a Java program, so you will need to have a JDK on your machine
www.sun.org

Instructions for running the simulator are provided in a note on the course webpage.

Simulator Demo

I/O Ports and I/O Addresses

Enable exchange of data between chip and external I/O devices
- A port for a particular device identified by I/O Address

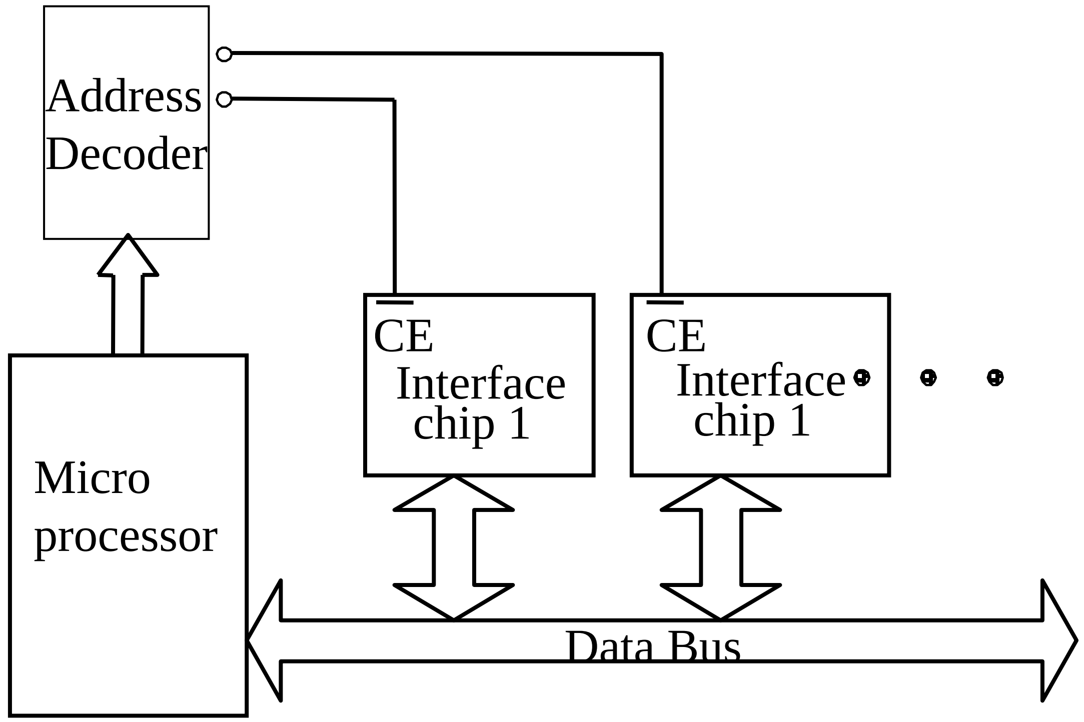
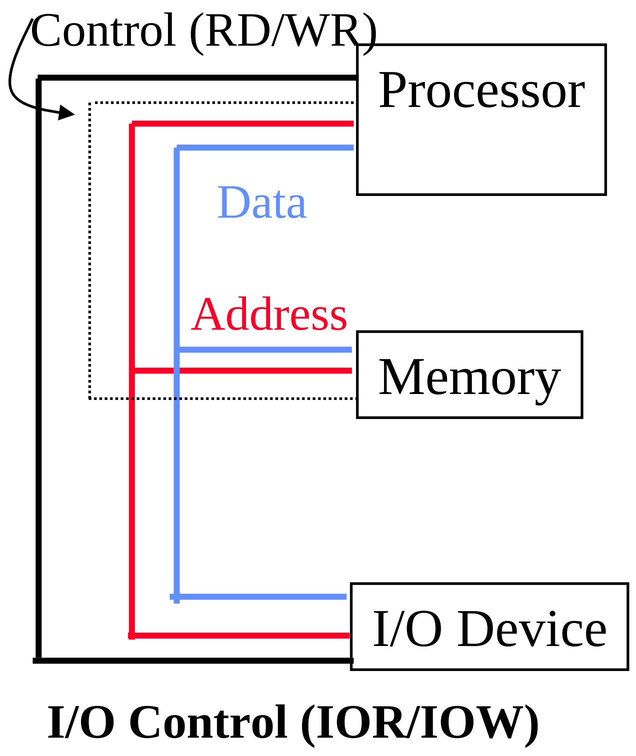
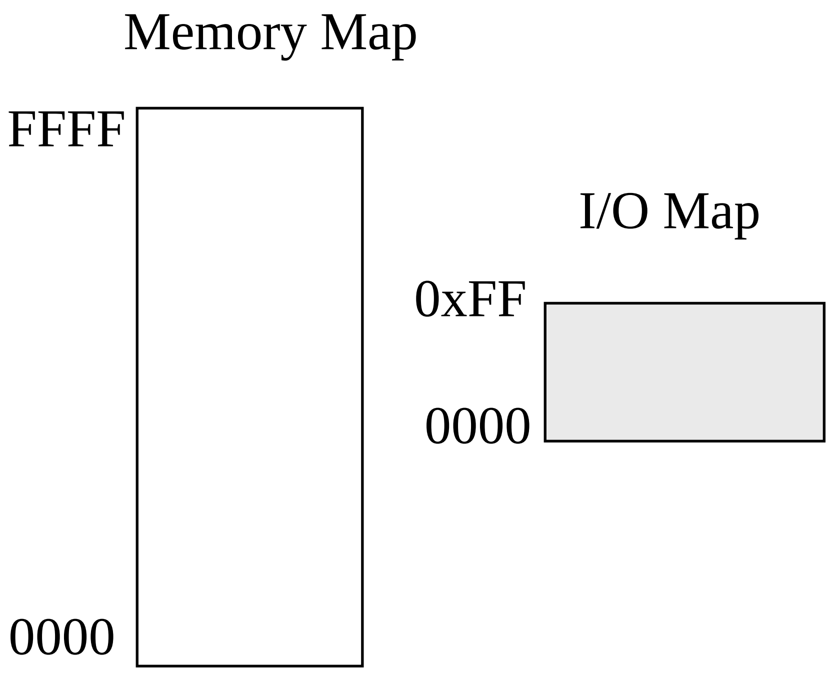


Figure 7.1 Interface chip, I/O devices, and microprocessor

I/O Addressing Schemes: Isolated I/O

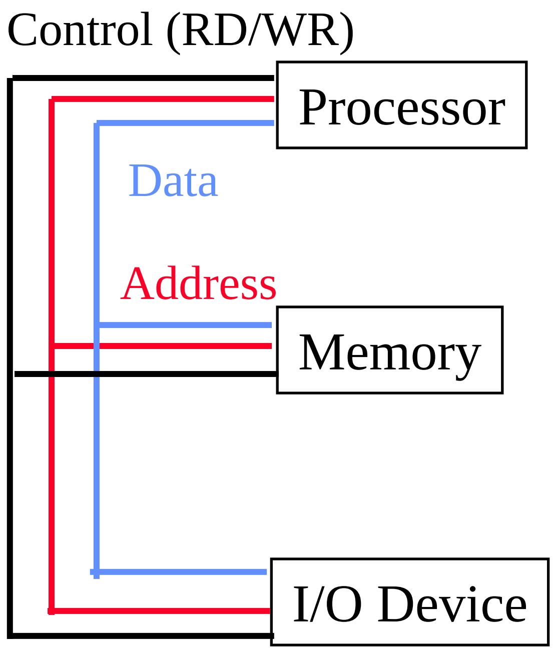
- Based on a separate address space for I/O devices.
- Programming: use dedicated instructions for I/O operations.

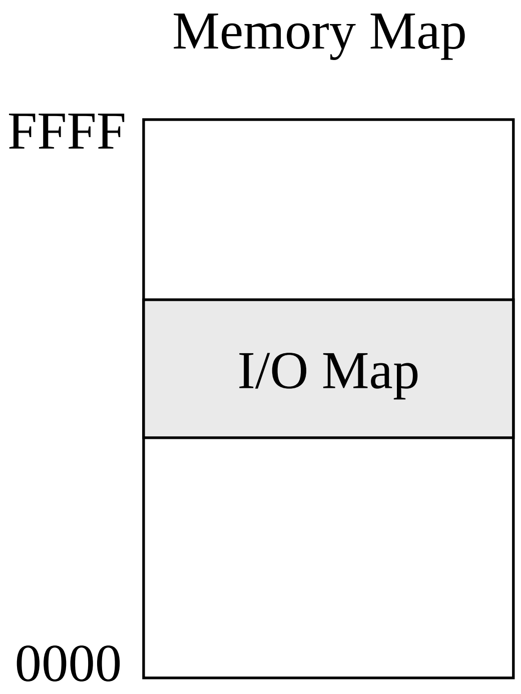




I/O Addressing Scheme : Memory Mapped I/O

- Both I/O and memory shared same memory space.
- Programmer’s: same instructions for both memory and I/O





Programming MicroControllers

Using a High-level Language

- Syntax: closer to human languages
- Translator (Compiler): convert program in high-level language into machine language
- Allow the users to work on the program logic at higher level.

Machine Language (executable code)

- Sequence of binary digits that can be executed by the processor

Assembly Language (mnemonic code)

- Assembly instruction: representation of a machine instruction
  (1 assembly instruction = 1 machine instruction)
- Hard to understand, program, and debug for humans

Real-Time Embedded Systems programming

- High level languages not adequate: performance issues
- Memory space (Java exe >> C exe >> Assembly exe = machine exe)

68HC12 Instruction Examples: Load and Store

- LOAD: copies contents of operand into CPU register.
- STORE: save contents of CPU register into memory location.
- Execution affects certain flags
    - N and Z flags of the CCR automatically updated; V flag cleared.

Learning HC12 Instructions : (1) Built-in Operands

Mnemonic	Function	Operation
STAA	Store A	$(A) \Rightarrow M$
STAB	Store B	$(B) \Rightarrow M$
STD	Store D	$(A) \Rightarrow M, (B) \Rightarrow M+1$
STS	Store SP	$(SP) \Rightarrow M, M+1$
STX	Store X	$(X) \Rightarrow M:M+1$
STY	Store Y	$(Y) \Rightarrow M:M+1$

Learning New HC12 Instructions (1) : Built-in Operands

Mnemonic	Function	Operation
LDAA	Load A	(M) $\Rightarrow$ A
LDAB	Load B	(M) $\Rightarrow$B
LDD	Load D	(M:M+1) $\Rightarrow$(A:B)
LDS	Load SP	(M:M+1) $\Rightarrow$SP
LDX	Load index register X	(M:M+1) $\Rightarrow$ X
LDY	Load index register Y	(M:M+1) $\Rightarrow$ X
LEAS	Load effective address into SP	Effective address$\Rightarrow$SP
LEAX	Load effective address into X	Effective address$\Rightarrow$X
LEAY	Load effective address into Y	Effective address$\Rightarrow$Y

Flags = SXHINZVC

Learning New HC12 Instructions (2) : Addressing Modes

Source	Operation	Mode	Coding	Flags
LDAA #opr8i	(M)=>A	IMM	86 ii	----xx0-
LDAA opr8a	Load Accumulator A	DIR	96 ii	
LDAA opr16a		EXT	B6 hh ll	
LDAA oprx0_xysp		IDX	A6 xb	
LDAA oprx9, xysp		IDX1	A6 xb ff	
LDAA oprx16,xysp		IDX2	A6 xb ee ff	
LDAA [D, xysp]		[D,IDX]	A6 xb	
LDAA [oprx16,xysp]		[IDX2]	A6 xb ee ff	

Addressing Modes

Inherent Addressing Mode

- Instructions do not use extra bytes for operands: instructions either do not need operands or all operands are CPU registers.

- Operands are implied by the opcode.

- Examples:

```
NOP
INX
DECA
```

Immediate **Addressing Mode**

- Operands included in the instruction.
- CPU does not access memory for operands.
- Examples:

```
LDAA   #$55

LDX    #$800
```

Operand Compatibility :

- What if we now have `LDX #$55` ?

Instruction encoding :

```
• LDAA #$55              86  55
• LDX #$55               CE  00 55
```

- What's wrong with this ? `LDAA #$1234`

Direct **Addressing Mode**

- Can only specify memory locations in the range of 0– 255.
- Uses only one byte to specify the operand address.
- Examples:

```
      LDAA  $20
      LDAB  $40
```

Extended **Addressing Mode**

- Full 16-bit address provided in the instruction.
- Examples:

```
      LDAA  $4000
      LDX   $FE60
```

Indexed **Addressing Mode(s)**

- Operand tells how to calculate *effective address* of the data

- Many forms calculate address as sum of parts
  - Parts = registers and/or constants.

- Effective address = Sum of index register (X, Y, PC, or SP) and offset to specify address of an operand.
  - Offset can be 5-bit, 9-bit, and 16-bit signed value or value in accumulator A, B, or D

Hence, the many different forms …

The Various Forms of Indexed Addressing

1. 5-bit Constant Offset (Indexed Addressing)

   Examples : `ldaa 0,X` `stab -8,X`

   The range of the offset : -16 to +15.

2. 9-bit Constant Offset

   Examples : `ldaa $FF,X` `ldab -20,Y`

   The range of the offset : -256 to +255

3. 16-bit Constant Offset

   Examples: `ldaa 2000,X` `staa 4000,Y`

   Allows access any location in the 64-KB range

4. Accumulator Offset

   Examples: `ldaa B,X` `stab B,Y`

   The offset is contained in either A, B or the 16-bit accumulator D.

16-bit Constant Indirect Indexed Addressing Mode

- 16-bit offset added to index register to form address of memory location containing a pointer to memory location affected by the instruction.

- Square brackets distinguish this addressing mode from 16-bit constant offset indexing. For example,

```
LDAA  [10,X]
STAA  [20,Y]
```

Compare :

```
LDAA  10, X
LDAA  [10,X]
```

16-bit Constant Indirect Indexed Addressing (example)

```
1000
…
100A          10
100B          22
…
1020          2f
1021          10
1022          ff

LDAA 10, X              ; Suppose X=1000
                        ; A <- m(000A+1000) = 10

LDAA [10, X]            ; A <- m[m(000A+1000)] = m(1022) = ff
```

Transfer and Exchange Instructions

- Copy the contents of a CPU register or accumulator into another CPU register or accumulator.

1. **Universal Transfer** Instruction: `TFR abcdxys, abcdxys`

   `TFR A, B` Transfers contents of A to B

   - TFR does not affect any flags.

2. **Specific Transfer** Instructions

   `TAB` Transfers contents of A to B

   `TBA` Transfer contents of B to A

   - Both affect the N, Z, and V condition code bits.

Transfer and Exchange Instructions

3. **Exchange** contents of a pair of registers or accumulators.

`EXG`	`A, B`	Swap contents of A and B

4. **Sign-extend** 8-bit two's complement number into a 16- bit number

`SEX`	`A,X`	Move contents of A into LSB of X, and sign extend

- Purpose: To use an 8-bit value in 16-bit signed operations.

Move Instructions

- Move data bytes or words from a source to a destination in memory.

- Six combinations of immediate, extended, and index addressing modes allowed to specify source/destination addresses:

  IMM $\Rightarrow$ EXT, IMM $\Rightarrow$ IDX, EXT $\Rightarrow$ EXT,

  EXT $\Rightarrow$ IDX, IDX $\Rightarrow$ EXT, IDX $\Rightarrow$ IDX

- Examples:

```
MOVB  $100,$800
MOVW  0,X, 0,Y
```

Add and Subtract Instructions

Flags = SXHINZVC

• Destinations of these instructions are always a CPU register or accumulator.

Source	Operation	Mode	Coding	Flags
ADDA #opr8i	(A)+(M)=>A	IMM	8B ii	- - x – x x x x
ADDA opr8a	Add w/o Carry to A	DIR	9B ii	
ADDA opr16a		EXT	BB hh ll	
ADDA oprx0_xysp		IDX	AB xb	
ADDA oprx9, xysp		IDX1	AB xb ff	
ADDA oprx16,xysp		IDX2	AB xb ee ff	
ADDA [D, xysp]		[D,IDX]	AB xb	
ADDA [oprx16,xysp]		[IDX2]	AB xb ee ff	

Variations of the Same Theme

Adding TO OTHER registers

```
adda $800          ; A ⇐ [A] + [$800]
addb $800          ; B ⇐ [B] + [$800]
addd $800          ; D ⇐ [D] + [$800]
```

WORD!

Adding TWO registers

```
aba                ; A ⇐ [A] + [B]
abx                ; X ⇐ [B] + [X]
aby                ; Y ⇐ [B] + [Y]
```

Notice : Order of operands

Addition Overflow

Problem: Fixed width registers have limited range.

Overflow: result of an operation **outside the range** that can be represented

<u>8-bit Unsigned Integer Example</u>:

$$\begin{array}{rcl} 255_{10} & = & 1111\ 1111_2 \\ +\quad 1_{10} & = & 0000\ 0001 \\ \hline 256\ ??\quad 0_{10} & = & (1)\ 0000\ 0000_2 \end{array}$$

Need 9 bits to represent result

In this (unsigned) case, with fixed 8-bits: OVERFLOW OCCURRED!

- A carry @ msb is important in the INTERPRETATION of the result.

Addition Overflow

Is that the only interpretation of the example?

$$
\begin{array}{rr}
 & 1111\ 1111_2 \\
+ & \underline{0000\ 0001_2} \\
(1) & 0000\ 0000_2
\end{array}
$$

Result is correct if the values are interpreted as 2's complement signed integers! (– 1 + 1 = 0)

- In that case, no overflow, and carry at MSB not important!

Addition Overflow

OVERFLOW DEPENDS ON THE INTERPRETATION OF VALUES!

Another example:

	unsigned	**signed**
$0111\ 1111_2$	127	127
$+\ 0000\ 0001_2$	+ 1	+ 1
$1000\ 0000_2$	128	– 128

UNSIGNED OVERFLOW Did NOT occur

- The C flag is cleared.

SIGNED OVERFLOW did occur! (even though there is no carry outside of fixed width!)

- The V flag is set.

Example: Suppose that A contains $73

Execute: `ADDA #$40`

$**73** + $**40** = ? Overflow ?

Results: **A** := $**B3** (= %1011 0011)

Z := 0 result ≠ 0

N := 1 result is negative (signed)

C := 0 (no carry out of msbit)

V := 1 +ve + +ve = –ve