# Chapter 1: Introduction to the HCS12 Microcontrollor

## Basic Computer Concepts

A computer is made up of ***hardware*** (from EE/CompE) and ***software*** (from CSC).

1. ***processor*** (= CPU = core) ← ENGG 32A + ENGG 36
   - the brain of a computer system
   - performs = arithmetic and logic computations
   - single processor (core) vs. multiple processors (cores)

2. ***memory*** ← ENGG 32A
   - stores programs and data
   - semiconductor chips: RAM, ROM, PROM, EPROM EEPROM, flash, etc
   - magnetic and/or optical materials: CD-R, DVD-R, hard drive, etc

3. ***input devices***

4. ***output devices***

Learned about a basic CPU and memory devices in ENGG 32A. [see next 3 pages.]

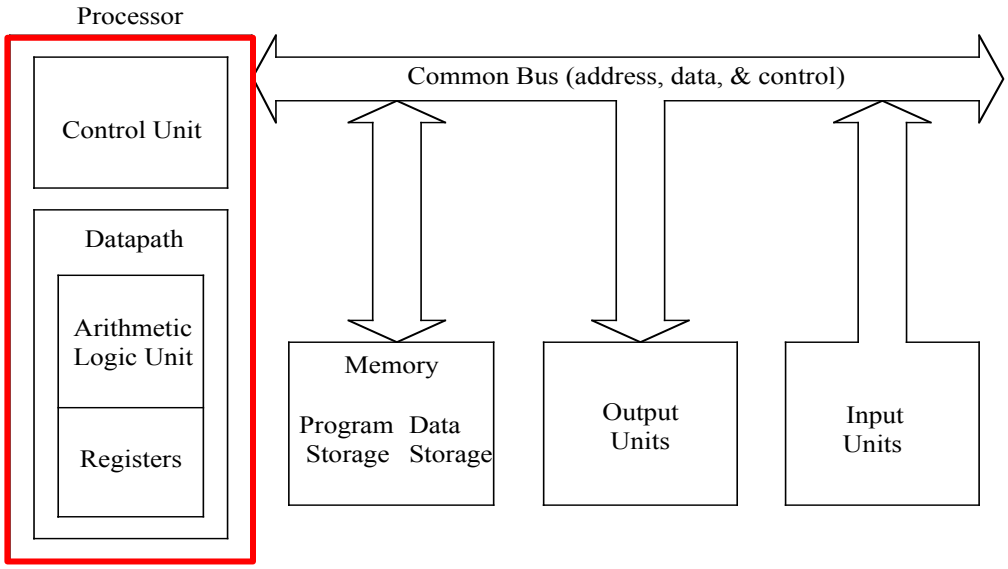



Figure 1.1 Computer Organization

# The Processor

A microprocessor is a processor fabricated on a single integrated circuit (IC).

Classifying the microprocessors by using the **number of bits** (or word length) that a microprocessor can manipulate in one operation. For example, a 32-bit microprocessor can only work on up to 32 bits of data in one operation.

Divided into two major parts: ***Datapath*** and ***Control Unit***.

**Datapath** consists of a ***register file*** and an ***ALU***. [see p. 3]

- Register file = multiple registers for faster data access inside CPU.
- A register is a storage location in the CPU, holding data or a memory address. For example, MAR, PC, MBR, DR.
- The register file makes program execution more efficient because accessing data in registers is much faster than accessing data in memory outside the CPU.

- ALU performs all of the arithmetic and logic computations.
- ALU receives data from main memory or registers, performs a computation, and writes the result back to main memory or registers.

**Control Unit** contains the ***hardware instruction logic***. [see p. 4]

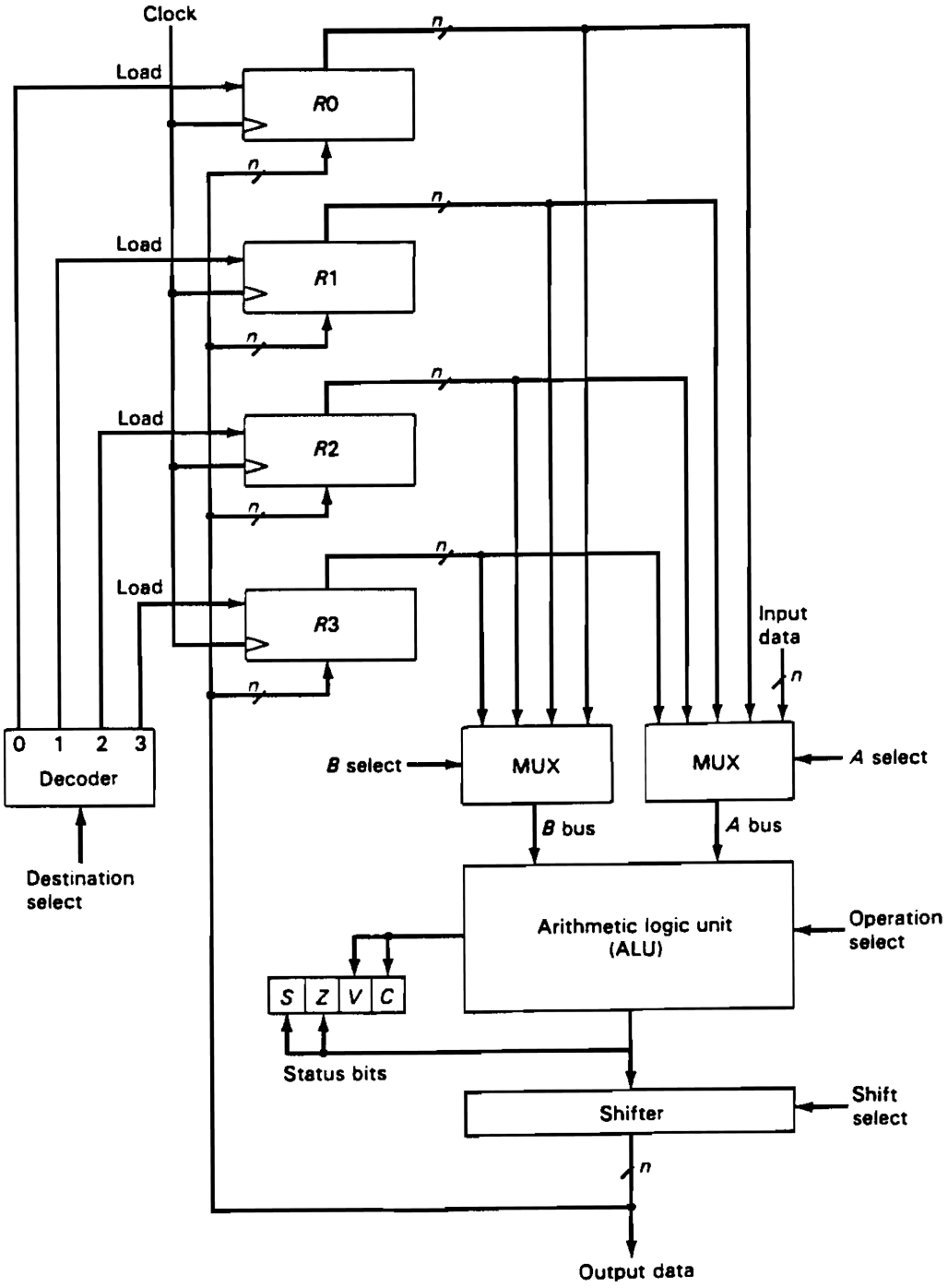
- Decodes and monitors the execution of instruction. (instructions = control words in ENGG 32A)
- Coordinates requests of using CPU resources from various parts of the computer system.
- Operation is synchronized by the system clock.
- Maintains a register called ***program counter*** (PC), which sets the memory address of the next instruction to be executed. → IR = ***instruction register***
- A ***status register*** is used to store the presence of overflow, an addition carry, a subtraction borrow, zero/negative, intterrupt enable, etc. The content of this register is then used for program control and decision making.

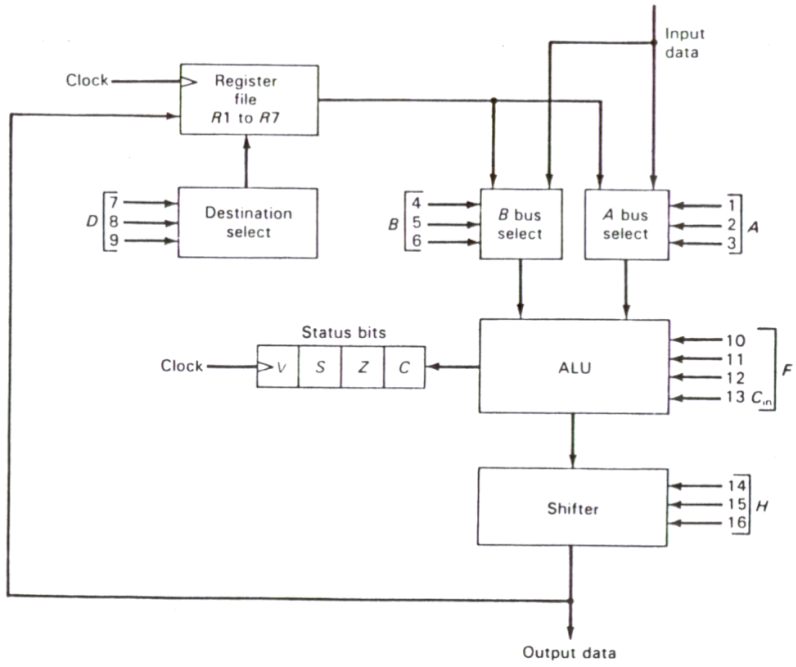
32-bit and 64-bit microprocessors incorporate on-chip memory (so-called **Cache** memory) to enhance performance by prefetching next several instructions in advance.

Microprocessors interface to input/output devices through peripheral chips.

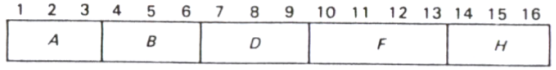
For example, i8255 interface a parallel device such as a printer or 7-segment display with the Intel 8-bit microprocessor 8085.

## A Basic CPU


Clock
Load
R0
R1
R2
R3
n
0 1 2 3
Decoder
Destination select
B select
MUX
MUX
A select
Input data
B bus
A bus
Arithmetic logic unit (ALU)
Operation select
S Z V C
Status bits
Shifter
Shift select
Output data




(a) Block diagram




(b) Control word

**FIGURE 7-20**
Processor Unit with Control Variables

**Encoding of Control Word for Processor Unit**

| Binary code | Function of selection fields | | | | | |
|---|---|---|---|---|---|---|
| | A | B | D | F with $C_{in} = 0$ | F with $C_{in} = 1$ | H |
| 000 | Input | Input | None | $F = A$ | $F = A + 1$ | No shift |
| 001 | $R1$ | $R1$ | $R1$ | $F = A + \overline{B}$ | $F = A + B + 1$ | SHL |
| 010 | $R2$ | $R2$ | $R2$ | $F = A + \overline{B}$ | $F = A - B$ | SHR |
| 011 | $R3$ | $R3$ | $R3$ | $F = A - 1$ | $F = A$ | Bus = 0 |
| 100 | $R4$ | $R4$ | $R4$ | $F = A \wedge B$ | — | — |
| 101 | $R5$ | $R5$ | $R5$ | $F = A \vee B$ | — | ROL |
| 110 | $R6$ | $R6$ | $R6$ | $F = \overline{A} \oplus B$ | — | ROR |
| 111 | $R7$ | $R7$ | $R7$ | $F = \overline{A}$ | — | — |

| Field: | A | B | D | F | H |
|---|---|---|---|---|---|
| Symbol: | $R2$ | $R3$ | $R1$ | $F = A - B$ | No shift |
| Control word: | 010 | 011 | 001 | 0101 | 000 |

**Examples of Microoperations for the Processor**

| Microoperation | Symbolic designation | | | | | Control word | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | D | F | H | A | B | D | F | H |
| $R1 \leftarrow R2 - R3$ | $R2$ | $R3$ | $R1$ | $F = A - B$ | No shift | 010 | 011 | 001 | 0101 | 000 |
| $R4 \leftarrow$ shr $(R5 + R6)$ | $R5$ | $R6$ | $R4$ | $F = A + B$ | SHR | 101 | 110 | 100 | 0010 | 010 |
| $R7 \leftarrow R7 + 1$ | $R7$ | — | $R7$ | $F = A + 1$ | No shift | 111 | 000 | 111 | 0001 | 000 |
| $R1 \leftarrow R2$ | $R2$ | — | $R1$ | $F = A$ | No shift | 010 | 000 | 001 | 0000 | 000 |
| Output $\leftarrow R3$ | $R3$ | — | None | $F = A$ | No shift | 011 | 000 | 000 | 0000 | 000 |
| $R4 \leftarrow$ rol $R4$ | $R4$ | — | $R4$ | $F = A$ | ROL | 100 | 000 | 100 | 0000 | 101 |
| $R5 \leftarrow 0$ | — | — | $R5$ | — | Bus = 0 | 000 | 000 | 101 | 0000 | 011 |

# Limitations of Microprocessors

- Require external memory to execute programs.
- Cannot directly interface with input/output devices; require peripheral chips.
- Need additional logic (such as address decoders and buffers) to interconnect external memory and peripheral interface chips with the microprocessor.
- Such a microprocessor-based system cannot be made as small as an IC chip because it is ususlaly mounted on ***printed circuit board*** (= motherboard).

→ Lead to the development of **microcontrollers**.

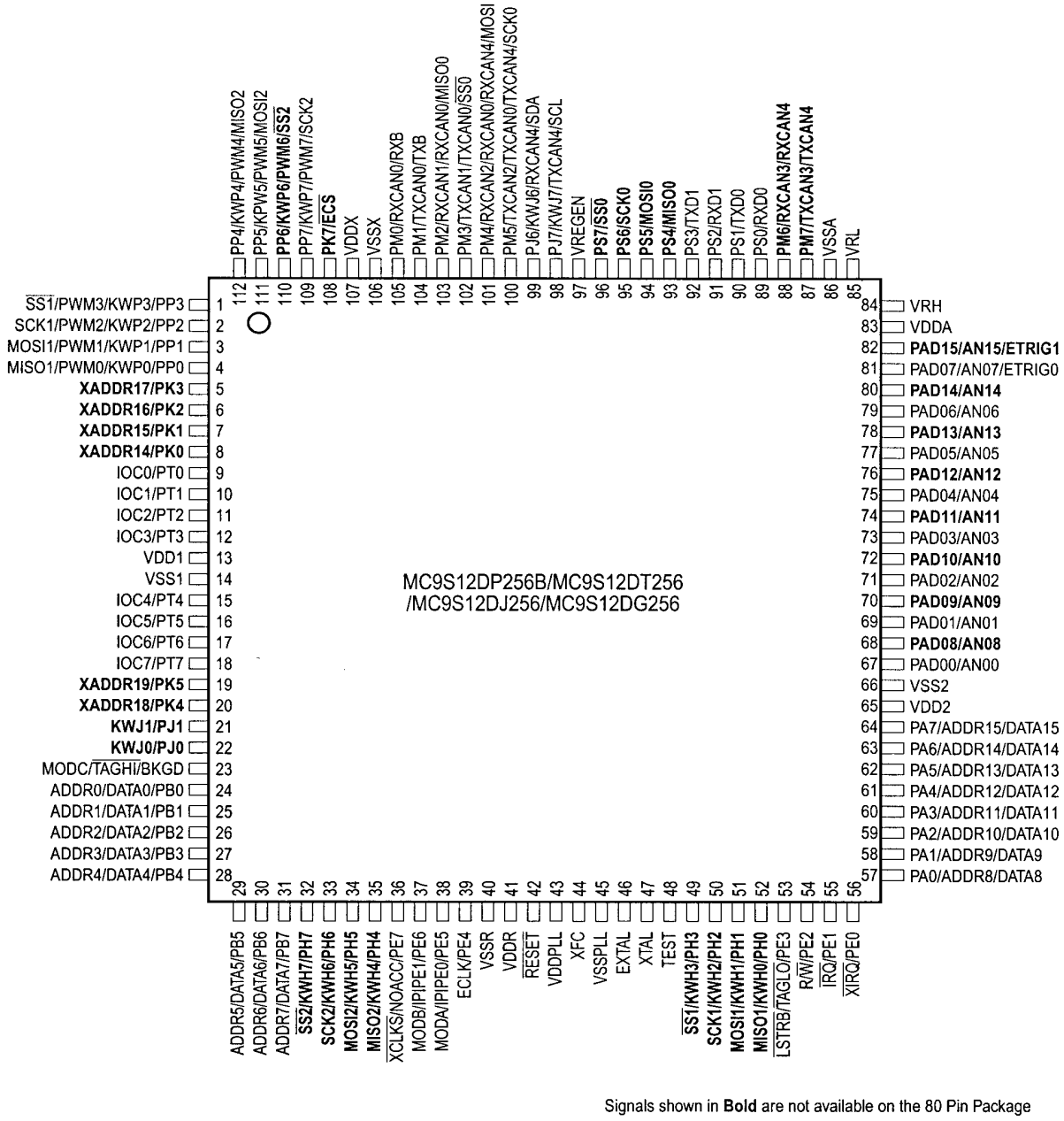
# The Microcontrollers

A computer implemented on a single "very large scale integration" (**VLSI**) chip.
[will learn about VLSI chip design in ENGG 172]

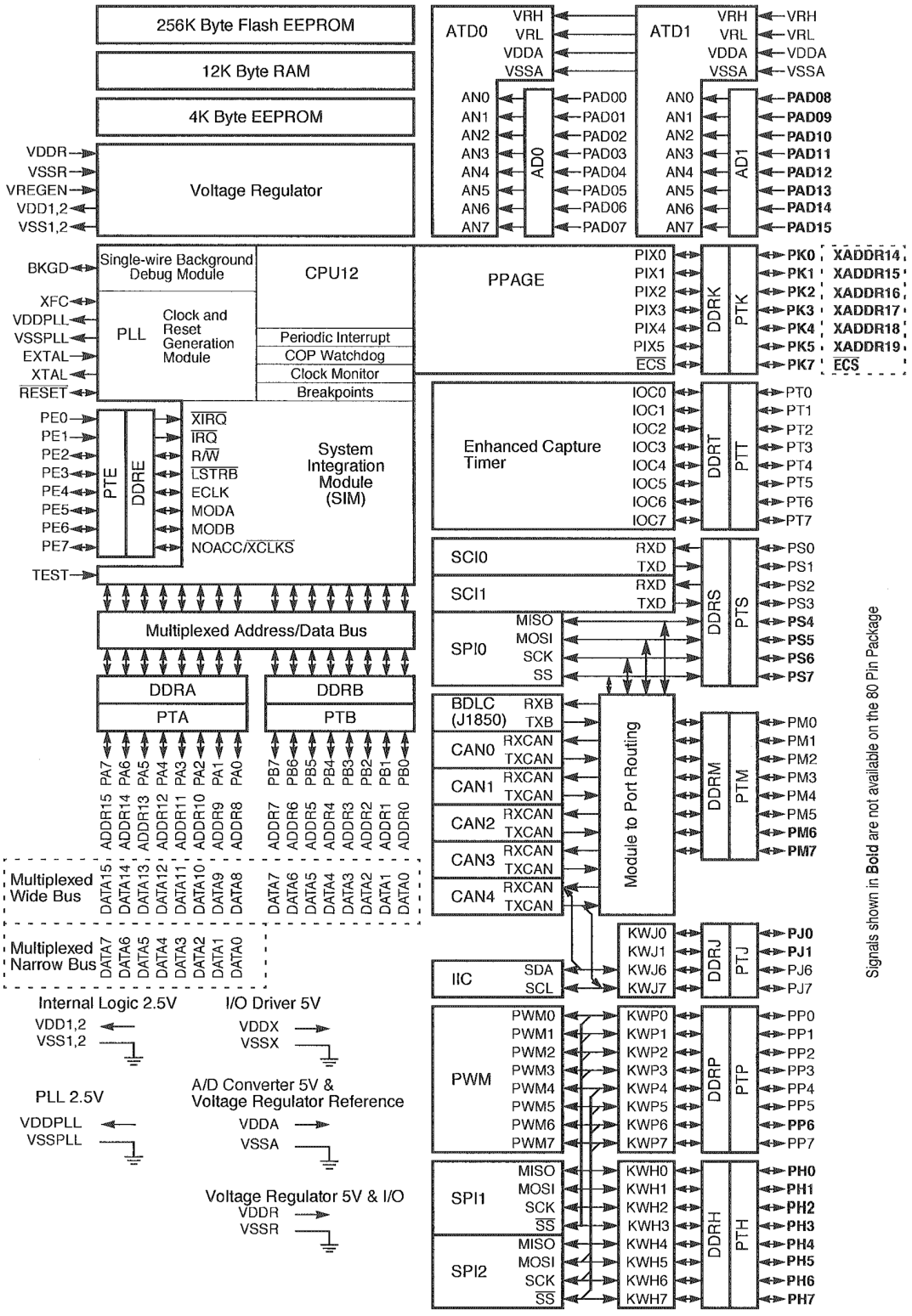
A microcontroller contains everything a microprocessor contains plus one or more of the following components:

- ***Memory*****:** stores data and programs.
- ***Timer*****:** includes input capture, output compare, counter, pulse accumulator, pulse-width modulation (PWM) modules, real-time interrupt, etc. It can be used to measure frequency, period, pulse width, and duty cycle, create time delay, and generate waveforms. → oscilloscope
- ***Analog-to-Digital Converter*** **(ADC):** often used with a sensor, converting certain non-electric quantities (temperature, weight, pressure) into an electric voltage and then into a digital value. → digital thermometer
- ***Digital-to-Analog Converter*** **(DAC):** convert a digital value into a voltage output; e.g., control motor speed, adjust brightness of a light bulb. → waveform generator
- ***Direct memory access*** **(DMA)** ***controller*****:** A DMA is a data transfer method that requires the CPU to perform initial setup but does not requires the CPU to execute instructions to control the data transfer. → burn a DVD
- ***Parallel I/O interface*****:** Parallels ports are used to drive parallel I/O devices. → printer
- ***Asynchronous serial I/O interface*****:** transmits data bits serially without using a clock signal to synchronize the sender and receiver. → keyboard
- ***Synchronous serial I/O interface*****:** transmits data bits serially with a clock signal to synchronize sender and receiver. → computer-to-computer communications
- ***Memory component interface circuitry*****:** generates appropriate control signals to operate the memory chips.
- ***Digital signal processing*** **(DSP)** → FFT function in oscilloscope

We are using the Freescale **HC9S12DG256C** microcontroller on the Wytec **Dragon12-Plus2** Trainer.




Pin assignment of the MC9S12DG256 MCU




Block diagram of the MC9S12DG256 MCU.

# Features of the HCS12 Microcontroller Family

- 80 to 112 pins VLSI package
- 16-bit CPU
- 64 Kbytes memory space
  - 768 bytes to 4 Kbytes of on-chip EEPROM
  - 1 to 12 Kbytes of on-chip SRAM
  - 32 to 128 Kbytes of on-chip flash memory
- 8-bit or 10-bit A/D converter
- Sophisticated timer functions: input capture, output compare, pulse accumulator functions, real-time interrupt
- Pulse-width modulation (PWM)
- Multiple parallel input-output (I/O) ports
- Serial communication interfaces: asynchronous serial communication interface (SCI), synchronous peripheral interface (SPI), controller area network (CAN) bus interface, byte data link communication interface (BDLC)
- Computer operating properly (COP) watchdog timer
- Background debug mode (BDM)

# Embedded Systems

A system that uses multiple microcontrollers as controller(s) is called an ***embedded system***. For example, cell phones, home security systems, and modern automobiles.

Each microcontroller has its own responsibilities. These microcontrollers may need to exchange information or even coordinate their operations.

# Machine Instructions

Programs = software = a set of **instructions** that the computer hardware can execute.

A program is stored in the computer's memory in the form of binary numbers called ***machine instructions*** = a sequence of binary digits executed by the CPU
→ Control Words (in ENGG 32A).

Writing programs in machine language is extremely difficult and inefficient. Hard to understand, program, or debug for human being; also, CPU-architecture dependent!

| ***Machine instructions*** | ***Mircrooperations*** |
|---|---|
| 0001 1000 0000 0001 | register A ← register A + register B |
| 0001 0011 0001 0001 | register A ← register A – 1 |
| 1000 0111 0010 1000 | register B ← 6 |
| control words | |

# Assembly Language

- Invented to simplify the programming job.
- Significant improvement over machine language programming.
- Consists of assembly instructions: mnemonic representations of machine instructions.
- Programmers need not to scan through 0s and 1s in order to identify what instructions are in the program.
- For example,

ABA = add the content of register B to register A
DECA = decrement the content of register A by 1

| | | ***object code*** ← ***assembler*** ← | assembly instructions |
|---|---|---|---|
| *Line* | ***address*** | ***machine code*** | ***source code*** |
| 1: | | 00002000 | org $2000 |
| 2: | 2000 | B6 0800 | ldaa $800 |
| 3: | 2003 | BB 0801 | adda $801 |
| 4: | 2006 | BB 0802 | adda $802 |
| 5: | 2009 | 7A 0900 | staa $900 |
| 6: | | | end |

↑ -- Each line of instruction is stored in one or more memory addresses.

- The assembly program is called source program or ***source code***.
- A software program, called an ***assembler***, is used to translate the source code in assembly language into machine instructions.
- The output (i.e., machine instructions) of the assembler is called ***object code***.

Drawbacks of Assembly Language:

- Programmers must be very familiar with the hardware organization of the microcontroller → CPU-architecture dependent.
- The programs are not easily transferrable to other CPU architectures.
- Still difficult to understand for anyone other than the author.
- Productivity is not satisfactory for large programming projects.

# High-level Language

√ Examples: C, C++, JAVA.
√ Syntax of a high-level language is similar to English, easier to understand.
√ Another software, called a translator or ***compiler***, is required to translate the program written in a high-level language.
√ One statement in high-level language often needs to be implemented by tens or even hundreds of assembly instructions.
√ High-level languages allow the user to work on the program logic at higher level, thus **higher productivity**.
× Resulting machine code **cannot run as fast** as its equivalent in assembly language due to the inefficiency of the compilation process.

Time-critical programs (e.g., arcade video games) are still written in assembly language.

A complier is a software program to translate the source program (= source code) into machine instructions (i.e., object code).

# The HCS12 CPU Registers

For temporary storage inside the CPU, this family of microcontrollers has many registers: ***CPU registers*** and ***I/O registers***.

**I/O registers:**

- Configure the operations of **peripheral** functions.
- Hold data transferred in and out of the peripheral subsystem.
- Record the status of I/O operations.
- Classified into ***data***, ***data direction***, ***control***, and **status** registers. They are treated as **memory locations** when they are accessed.

**CPU registers** perform general-purpose operations such as arithmetic, logic, and program flow control.

## CPU Registers

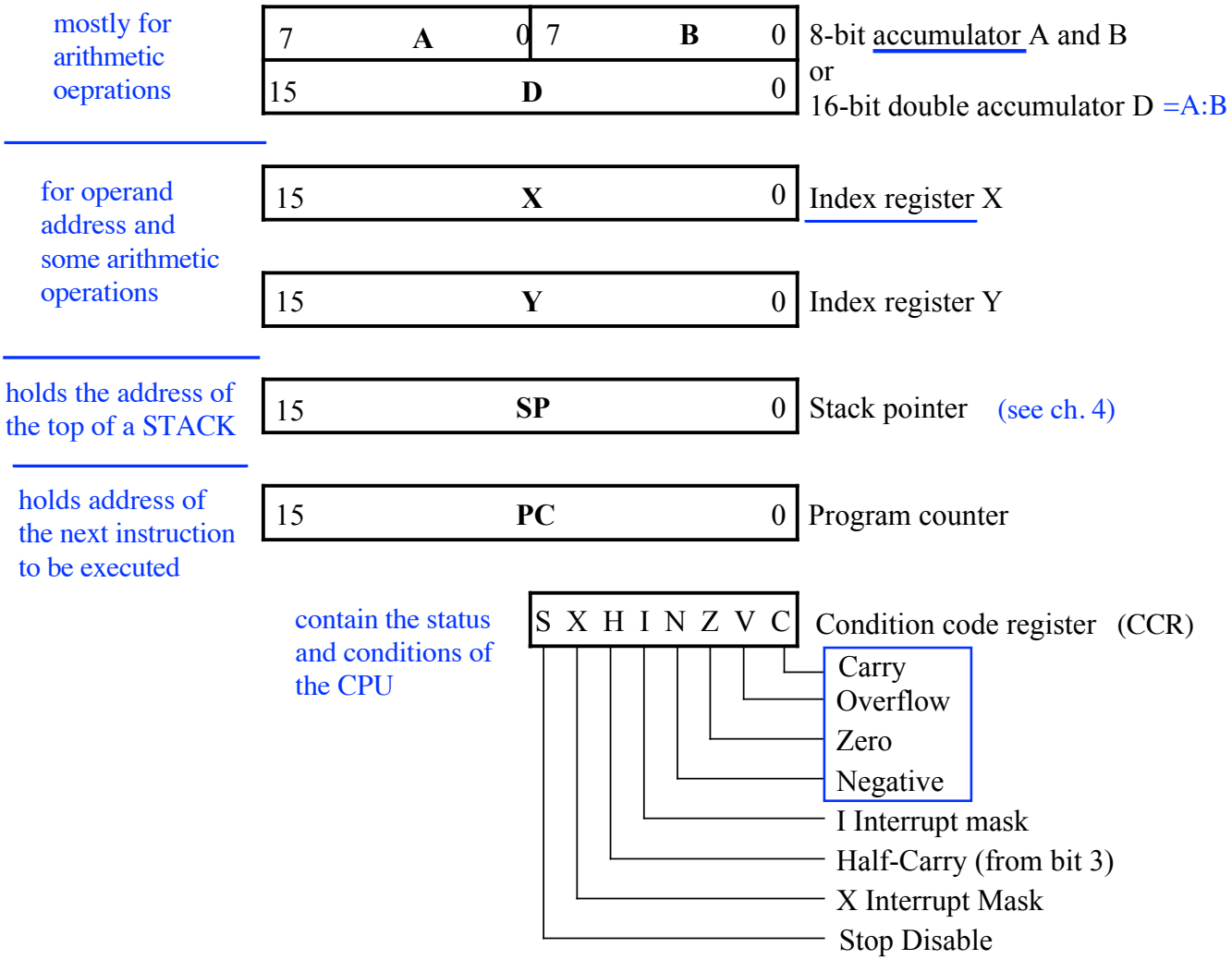



Figure 1.3 MC68HC12 CPU registers.

Condition Code Register = CCR

**General-Purpose Accumulator A and B**

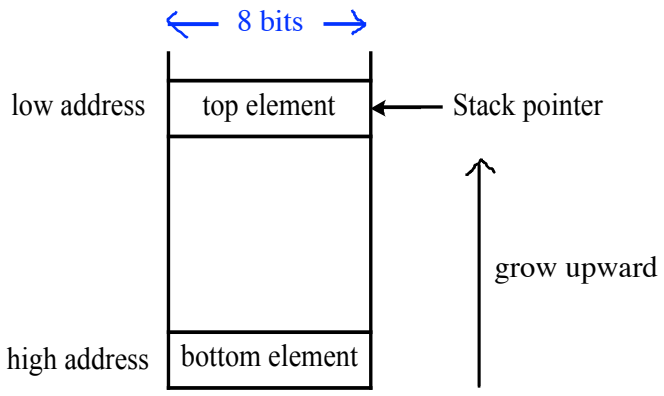
- Both are ***8-bit*** registers.
- Most arithmetic functions are performed on these two registers.
- They can also be concatenated to form a single ***16-bit*** accumulator, referred to as the **D accumulator**. **D = A:B**

**Index Registers X and Y**

- Used mainly in forming **operand addresses** during the instruction execution process.
- Also used in several arithmetic operations.

**Stack Pointer (SP)**

- A stack is a first-in-last-out (FILO) data structure.
- The 68HC12 has a ***16-bit*** SP to the top byte of the stack. The stack grows toward lower address.
- For dynamic data storage. [see ch. 4]




Memory sructure of a **stack**

**Program Counter (PC)**

- A ***16-bit*** register.
- Holds the address of the next instruction to be executed.
- Incremented by the number of bytes of the executed instruction after the instruction is executed.

**Condition Code Register (CCR)**

- A ***8-bit*** register
- Keep track of the program execution status, control the execution of conditional instructions, and enable/disable the interrupt handling.

# Types of Data in HCS12 MCU

- Baisc unit: bits
- 5-bit signed integers: formed during addressing mode computations
- 8-bit signed and unsigned integers
- 8-bit 2-digit BCD numbers
- 9-bit signed integers: formed during addressing mode computations
- 16-bit signed and unsigned integers
- 16-bit effective addresses: formed during addressing mode computations
- 32-bit singed and unsigned integers: used by extended division, extended multiply, and extended multiply-and-accumulate instructions

Negative numbers are represented in the **2's-complement** form.

16-bit or 32-bit integer is stored in memory from **most significant to least significant bytes**, starting from **low to higher addresses**.

A number can be represented in binary, octal, decimal, or hexadecimal format by adding an appropriate prefix in front of the number to indicate its base.

Prefixes for number bases inside assembly programs. (8 bits = 1 byte = 2 hex digits)

| base | prefix | example |
|---|---|---|
| binary | % | %1000 1101 |
| octal | @ | @215 |
| decimal | | 141 |
| hexadecimal (hex) | $ | $8D |

# Memory Addressing

- Memory consists of a sequence of directly addressable locations, referred to as an information unit.
- A memory location can be used to store **data**, **instruction**, and the **status** of peripheral devices.
- A memory location has two components: an **address** and its **contents**.

1 address contains 8 bits (=1 byte)

$2^{16}$ = 64K addresses
can store 64K bytes

addresses: $0000 to $FFFF

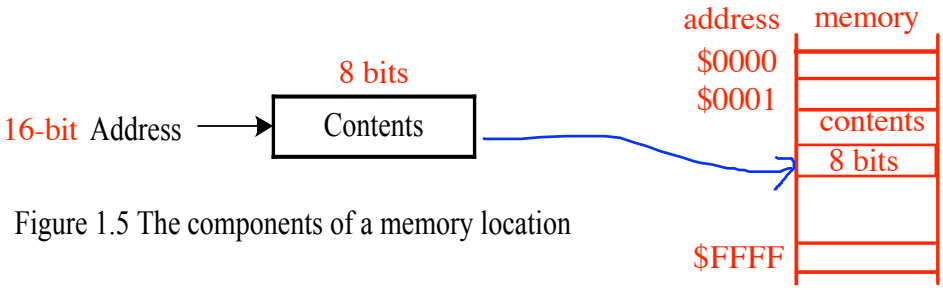
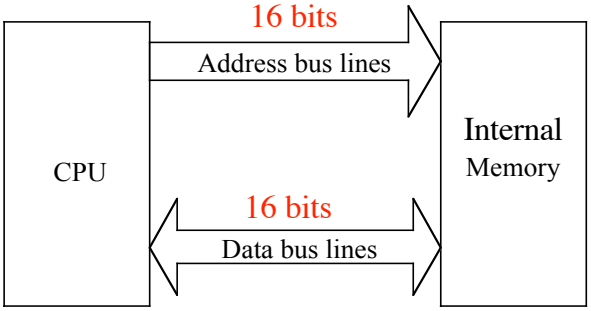
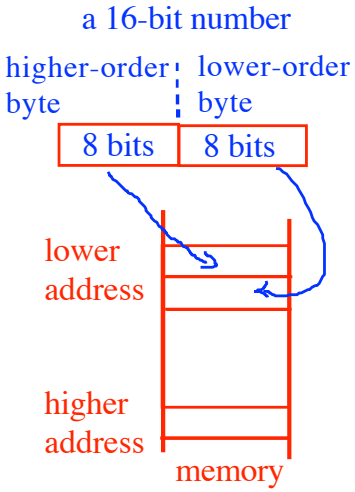



Figure 1.5 The components of a memory location

- Data transfers between the CPU and the memory are done over the common buses: ***address bus*** and ***data bus***.
- The CPU communicates with memory by first identifying the location's address and then passing this address on the address bus.
- The data are transferred between memory and the CPU along the data bus.




Components of a memory location.

Memory size is measure in **bytes** (i.e., groups of **8 bits**)

K = $2^{10}$ = 1,024
M = $2^{20}$ = 1024 x 1024 = 1,048,576
G = $2^{30}$ = 1024 x 1024 x 1024

T (tera) = $2^{40}$ = 1024 G
P (peta) = $2^{50}$ = 1024 T

The HCS12 has a 16-bit address bus and a 16-bit data bus.

- The 16-bit data bus allows the access of **16-bit data** in one operation.
- The 16-bit address bus allows the access of directly up to $2^{16}$ (i.e., 65,536 or **64K**) different memory locations.

Certain HCS12 members use paging techniques to allow user programs to access more than 64Kbytes.

**m[addr]** represents the contents of a memory location. For example, **m[$20]** refers to the contents of memory location at $0020 (in hex) or 32 (in decimal).

**[reg]** refers to the contents of a register. For example, **[A]** refers to the contents of accumulator A.

# Addressing Modes

A 68HC12 instruction consists of

- one or two bytes of **opcode** (= oepration)
- zero to five bytes of **operand** addressing information

Opcode bytes specify the **operation** to be performed by the CPU and the **addressing modes** to be used to access the operand(s).

**Addressing modes** determines how the CPU access memory locations to be acted upon.

HCS12 MCU addressing mode

Table 1.21 M68HC12 addressing mode summary

| | Addressing mode | Source format | Abbre. | Description |
|---|---|---|---|---|
| 1. | Inherent | **INST** (no externally supplied operands) | INH | Operands (if any) are in CPU registers |
| 2. | Immediate | **INST** #opr8i or **INST** #opr16i | IMM | Operand is included in instruction stream. 8- or 16-bit size implied by context |
| 3. | Direct | **INST** opr8a | DIR | Operand is the lower 8 bits of an address in the range $0000-$00FF |
| | Extended | **INST** opr16a | EXT | Operand is a 16-bit address |
| 4. | Relative | **INST** rel8 or **INST** rel16 | REL | An 8-bit or 16-bit relative offset from the current PC is supplied in the instruction |
| 5. | Indexed (5-bit offset) | **INST** oprx5,xysp | IDX | 5-bit signed constant offset from x,y,sp, or pc |
| | Indexed (pre-decrement) | **INST** oprx3,-xys | IDX | Auto pre-decrement x, y, or sp by 1 ~ 8 |
| | Indexed (pre-increment) | **INST** oprx3,+xys | IDX | Auto pre-increment x, y, or sp by 1 ~ 8 |
| | Indexed (post-decrement) | **INST** oprx3,xys- | IDX | Auto post-decrement x, y, or sp by 1 ~ 8 |
| | Indexed (post-increment) | **INST** oprx3,xys+ | IDX | Auto post-increment x, y, or sp by 1 ~ 8 |
| | Indexed (accumulator offset) | **INST** abd,xysp | IDX | Indexed with 8-bit (A or B) or 16-bit (D) accumulator offset from x, y, sp, or pc |
| | Indexed (9-bit offset) | **INST** oprx9,xysp | IDX1 | 9-bit signed constant offset from x, y, sp, or pc (lower 8-bits of offset in one extension byte) |
| | Indexed (16-bit offset) | **INST** oprx16,xysp | IDX2 | 16-bit constant offset from x, y, sp, or pc (16-bit offset in two extension bytes) |
| 6. | Indexed-Indirect (16-bit offset) | **INST** [oprx16,xysp] | [IDX2] | Pointer to operand is found at 16-bit constant offset from (x, y, sp, or pc) |
| | Indexed-Indirect (D accumulator offset) | **INST** [**D**,xysp] | [D,IDX] | Pointer to operand is found at x, y, sp, or pc plus the vlaue in D |

Note: INST = the instruction mnemonic; opr8i = 8-bit immediate value; opr16i = 16-bit immediate value; opr8a = 8-bit address; opr16a = 16-bit address; oprx3 = 3-bit value (amount to be incremented or decremented.)

**1. Inherent Mode** (INH)

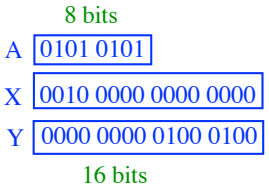
- Instructions do not uses extra bytes to specify operands because the instructions either do **not need operands** or all operands are internal CPU registers.
- Operands are implied by the opcode.

```
NOP     ; no operation (this instruction has no operands)
INX     ; increment index register X by 1
DECA    ; decrement accumulator A by 1
```

**2. Immediate Mode** (IMM)

- Operands are included in the instruction. (operands = numbers.)
- CPU does **not need to access memory** for operands.
- An immediate value (i.e., operand) can be **8 bits or 16-bits**, preceded by a "#".

```
LDAA #$55     ; load hex value $55 to accumulator A
LDX  #$2000   ; load hex value $2000 to index register X
LDY  #$44     ; load hex value $0044 to index register Y
```




(Although an 8-bit values was supplied in this LDY instruction, the assembler will generate the 16-bit value $0044 because the CPU expects a 16-bit value in this kind of register.)

How about LDAA #$0044?

**3. Direct Mode** (DIR)

- This mode can only specify memory locations in the range of 0-255 (i.e., $0000 to $00FF in hexadecimal).
- Uses only **one byte** (= 8 bits) to specify the operand address to save space.

```
LDAA   $20   ; A ← m[$0020]; load the contents in memory to A
LDAB   $40   ; B ← m[$0040]   (8 bits)
LDX    $20   ; XH ← m[$0020],  XL ← m[$0021]   (16 bits)
```

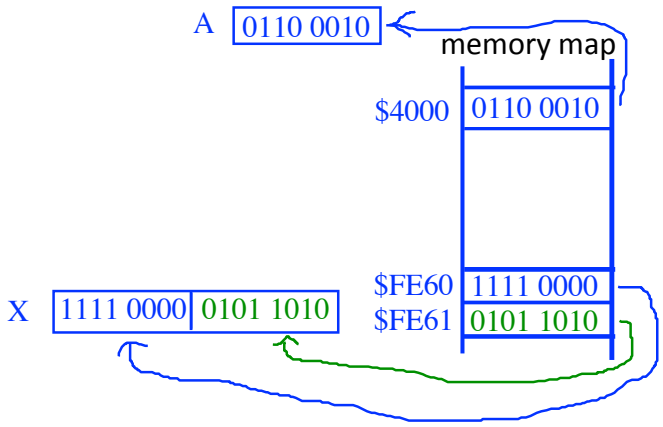
[The LDX instruction places the contents of memory locations at $0020 and $0021 in the higher and lower bytes ($X_H$ and $X_L$) of the index register X, respectively.]

→ see example after Extened Mode in next page

**Extended Mode** (EXT)

- Allows full **16-bit** address of a memory location to be operated on.
- Can be used to access any location in the 64-Kbyte memory map.

```
LDAA   $4000   ; A ← m[$4000]
LDX    $FE60   ; XH ← m[$FE60], XL ← m[$FE61]
```




**4. Relative Mode** (REL)

- Used only by ***branch instructions*** in form of conditional statements.
- Branching versions of bit manipulation instructions (BRCLR and BRSET) can also use relative addressing mode to specify branch target.
- A **short branch** instruction consists of an 8-bit opcode and a **signed 8-bit** offset, specifying a range of -128 ~ +127 (or $80 ~ $7F in hex).
- A **long branch** instruction consists of an 8-bit opcode and a **signed 16-bit** offset, specifying a range of -32768 ~ +32767 (or $8000 ~ $7FFF in hex).
- Both 8-bit and 16-bit offsets are **signed 2's complement** numbers to support branching forward and backward in memory.
- Branch offset is often specified using **a label**, rather than a numeric value, to specify the branch target and the assembler will calculate the actual branch offset (distance) from the instruction that follows branch instruction.

Instruction examples: BRCLR = branch if all selected bits are cleared to 0
BRSET = branch if all selected bits are set to 1

**5. Indexed Modes** (IDX)

- There are several variations for the indexed addressing mode.
- Uses a postbyte plus 0, 1, or 2 extension bytes after the instruction opcode.
- Uses the sum of an index register (X, Y, PC, or SP) and an offset to specify the address of an operand.
- The offset can be a 5-bit, 9-bit, and 16-bit signed value or the value in accumulator A, B, or D.
- Automatic pre- or post-increment or pre- or post-decrement by -8 to +8 are options.
- Indirect indexing with 16-bit offset or accumulator D as the offset is supported.

**memory address = content of a 16-bit register + offset**

Table 1.3 Summary of indexed operations

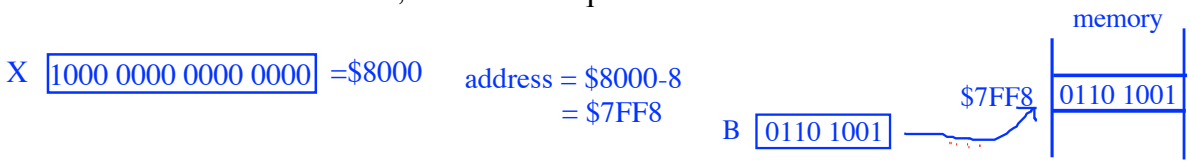
| Postbyte code (xb) | source code syntax | **Comments**<br>rr: 00 = X, 01 = Y, 10 = SP, 11 = PC |
|---|---|---|
| rr0nnnnn | r<br>n,r<br>-n,r | **5-bit constant** offset n = -16 to +15<br>r can be X, Y, SP, or PC |
| 111rr0zs | n,r<br>-n,r | **Constant offset** (9- or 16-bit signed)<br>z: 0 = 9-bit with sign in LSB of postbyte (s) -256< n < 255<br>1 = 16-bit 0 < n < 65535<br>if z = s = 1, 16-bit offset indexed-indirect (see below)<br>r can be X, Y, SP, or PC |
| 111rr011 | [n,r] | **16-bit offset indexed-indirect** 0 < n < 65536<br>rr can be X, Y, SP, or PC |
| rr1pnnnn | n,-r<br>n,+r<br>n,r-<br>n,r+ | **Auto pre-decrement/increment or auto post-decrement/increment** ;<br>p = pre-(0) or post-(1), n = -8 to -1 or +1 to +8<br>r can be X, Y, or SP (PC not a valid choice)<br>+8 = 0111<br>...<br>+1 = 0000<br>-1 = 1111<br>....<br>-8 = 1000 |
| 111rr1aa | A,r<br>B,r<br>D,r | **Accumulator offset** (unsigned 8-bit 0r 16-bit)<br>aa: 00 = A<br>01 = B<br>10 = D (16-bit)<br>11 = see accumulator D offset indexed-indirect<br>r can be X, Y, SP, or PC |
| 111rr111 | [D,r] | **Accumulator D offset indexed-indirect**<br>r can be X, Y, SP, or PC |

**5-bit Constant Offset Indexed Addressing**

- This mode adds a 5-bit signed offset, which is included in the instruction postbyte to the base index register to form the effective address.
- The base index register can be X, Y, SP, or PC.
- The range of the offset is from **-16 to +15**.
- Examples:

```
LDAA    0, X    ; loads the contents of the memory location pointed to
                ; by index register X with zero offset into A.

STAB    -8, X   ; stores the contents of B into the memory location with
                ; the address equal to the contents of X minus 8.
```




**9-bit Constant Offset Indexed Addressing**

- This mode uses a 9-bit signed offset, which is added to the base index register to form the effective address of the memory location.
- The base index register can be X, Y, SP, or PC.
- The range of the offset is from **-256 to +255**.

```
LDAA    $FF, X      ; loads the contents of the memory location with
                    ; the address equal to the value in X plus 255.

LDAB    -20, Y      ; loads the contents of the memory location with
                    ; the address equal to the value in Y minus 20.
```

**16-bit Constant Offset Indexed Addressing**

- This mode uses a 16-bit signed offset, which is added to the base index register to form the effective address of the memory location,
- The base index register can be X, Y, SP, or PC.
- This mode allows access any location in the **64-Kbyte** range.

```
LDAA    $2000, X

STAA    $4000, Y
```

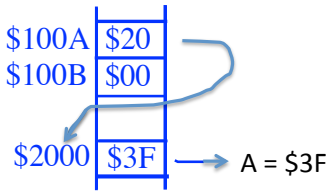
**16-bit Constant Indirect Indexed Addressing**

- A **16-bit** offset is added to the base index register to form the address of a memory location that contains a pointer to the desired memory location.
- The square brackets distinguish this addressing mode from the constant offset indexing.

```
LDAA    [10, X]     ; [ ] = an address pointer
                    ; X holds the base address of a table of pointers.
```

Let X=$1000

address=$1000+10=$100A



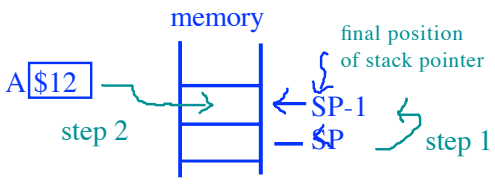

What will be the content of A if there is no "[ ]"?

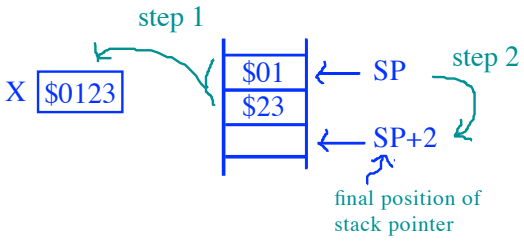
**Auto Pre/Post Decrement/Increment Indexed Addressing**

- Four ways to change the value in a base index register (X, Y, or SP) as a part of instruction execution
- The index register can be incremented or decremented by an integer value either before or after indexing taking place, in the ranges **-8 through -1** or **1 through 8**.
- **Pre-decrement** and **pre-increment** indexed addressing modes adjust the value of the index register before accessing the memory location, and the index register retains the changed value
- **Post-decrement** and **post-increment** indexed addressing modes use the initial value in the index register to access the memory location, and then change the value of the index register
- Used to perform operations on a series of data structure of memory

```
STAA    1, -SP      ; pre-decrement by 1        +SP = pre-increment
```

stores the contents of A at the memory location with the address equal to the value of stack pointer minus 1. After the store operation, the contents of SP are decremented by 1.




```
LDX     2, SP+      ; post-increment by 2       SP- = post-decrement
```

loads the contents of the memory locations pointed to by the stack pointer. After the instruction execution, the value of SP in incremented by 2.

**Accumulator Offset Indexed Addressing**

- The effective address of the operand is the sum of the accumulator (8-bit A, 8-bit B, or 16-bit D) and the base index register (X, Y, SP, or PC).
- The base register value is not changed.
- The accumulator provides an **unsigned** 8-bit or 16-bit offset.

```
LDAA        B, X        ; address = content of B + content of X
```

loads accumulator A with the contents of the memory location, whose address is given by the sum of the values of accumulator B and index register X.

**Accumulator D Indirect Indexed Addressing**

- The value in accumulator D is added to the base index register (X, Y, SP, or PC) to form the address of a memory location that contains the address (so-called a ***pointer***) to the memory location affected by the instruction.
- The square brackets distinguish this addressing mode from accumulator D offset indexing.

```
JMP         [D, PC]     ; jump to M[D+PC] = an address poiner
```

content of D + content of PC = an address of an address

# Instruction Examples

(See **Appendix A** of the textbook for the complete list of instructions.)

**1. LOAD and STORE Instructions**

- **LOAD** copies the contents of a memory location or places an immediate value into an accumulator or a CPU register.
- **STORE** copies the contents of a CPU register into a memory location.
- N and Z flags of the condition code register **CCR** are automatically updated and the V flag is cleared.
- All except for the relative addressing mode can be used to select the memory location or value to be loaded into an accumulator or CPU register.
- All except for the relative and immediate addressing modes can be used to select memory location to store contents of the CPU register.
- For example,

```
LDAA    0, X        ; A ← M[0+X]                move 1 byte
```
loads the contents of the memory location pointed to by index register X with zero offset into accumulator A.

```
LDAB    $1004       ; B ← M[$1004]              move 1 byte
```
loads the contents of the memory location at $1004 into accumulator B.

```
STAA    $20         ; M[$0020] ← A              move 1 byte
```
stores the contents of accumulator A into the memory location at $0020.

```
STX     $8000       ; M[$8000:$8001] ← X        move 2 bytes
```
stores the contents of index register X into the memory locations at $8000 and $8001.

```
LDD     #1000       ; A:B ← 1000                move a value
```
loads the decimal value 1000 into accumulator D.

What are the actual contents in D, A, and B?

Note: Each memory location (or address) has only **8 bits**.

LOAD and STORE instructions

| Mnemonic | Function | Operation | |
|---|---|---|---|
| LDAA | Load A | (M) ⇒A | 8 bits |
| LDAB | Load B | (M) ⇒ B | |
| LDD | Load D | (M:M+1) ⇒ (A:B) | |
| LDS | Load SP | (M:M+1) ⇒ SP | 16 bits |
| LDX | Load index register X | (M:M+1) ⇒ X | |
| LDY | Load index register Y | (M:M+1) ⇒ Y | |
| LEAS | Load effective address into SP | Effective address ⇒ SP | |
| LEAX | Load effective address into X | Effective address ⇒ X | |
| LEAY | Load efective address into Y | Effective address ⇒ Y | |

Store Instructions

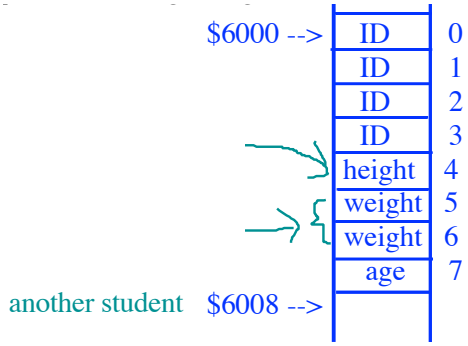
| Mnemonic | Function | Operation | |
|---|---|---|---|
| STAA | Store A | (A) ⇒ M | 8 bits |
| STAB | Store B | (B) ⇒ M | |
| STD | Store D | (A) ⇒ M, (B) ⇒ M+1 | |
| STS | Store SP | (SP) ⇒ M, M+1 | |
| STX | Store X | (X) ⇒ M:M+1 | 16 bits |
| STY | Store Y | (Y) ⇒ M:M+1 | |

**Record**

- A complex data structure.
- Usually use an index register or SP to point to the beginning address of the data structure.
- Example: A record of students contains four fields: ID number (4 bytes), height in inches (1 byte), weight in pounds (2 bytes), and age in years (1 byte).

  Assume this record is stored in memory location starting at $6000. The following instruction sequence accesses the height and weight fields of the first student:

```
LDX     #$6000    ; X points to the beginning
                  ; of the data structure
LDAB    4, X      ; copy height into B
LDD     5, X      ; copy weight into D
```



## 2. Transfer and Exchange Instructions

**Transfer Instructions**

| Mnemonic | Function | Operation | |
|---|---|---|---|
| TAB | Transfer A to B | (A) ⇒ B | affect N, Z, V flags |
| TAP | Transfer A to CCR | (A) ⇒ CCR | |
| TBA | Transfer B to A | (B) ⇒ A | affect N, Z, V flags |
| TFR | Transfer register to register | (A, B, CCR, D, X, Y, or SP) ⇒ A, B, CCR, D, X, Y, or SP | |
| TPA | Transfer CCR to A | (CCR) ⇒ A | |
| TSX | Transfer SP to X | (SP) ⇒ X | |
| TSY | Transfer SP to Y | (SP) ⇒ Y | |
| TXS | Transfer X to SP | (X) ⇒ SP | |
| TYS | Transfer Y to SP | (Y) ⇒ SP | |

8 bits

universal

16 bits

between two same size registers

**Exchange Instructions**

| Mnemonic | Function | Operation |
|---|---|---|
| EXG | Exchange register to register | (A, B, CCR, D, X, Y, or SP) ⇔ A, B, CCR, D, X, Y, or SP |
| XGDX | Exchange D with X | (D) ⇔ X |
| XGDY | Exchange D with Y | (D) ⇔ Y |

**Sign Extension Instructions**

| Mnemonic | Function | Operation |
|---|---|---|
| SEX | Sign extend 8-bit operand | (A, B, CCR) ⇒ X, Y, or SP |

8 bits to 16 bits

- **TFR** is the universal transfer instruction.
- **TFR** does not affect any condition code bits.

```
TFR   D, X      ; (D) ⇒ X; move content of D to X      (16 bits)
TFR   A, B      ; (A) ⇒ B; move content of A to B      (8 bits)
```

How to exchange X and Y?

- **TAB** and **TBA** affect the N, Z, and V condition code bits.
- Example: if we want to compute the squared value of A

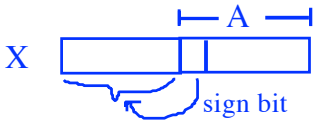
```
TAB             ; (A) ⇒ B
MUL             ; (A)x(B) ⇒ A:B; store the 16-bit product into D
```

- **EXG** exchanges the contents of a pair of registers or accumulators.

```
EXG   A, B        ; exchange the contents of A and B      (8 bits)
EXG   D, X        ; exchange the contents of D and X      (16 bits)
```

- **SEX** sign-extends an 8-bit 2's complement number into a 16-bit number so that it can be used in 16-bit signed operations.
- The 8-bit number is copied from A, B, or CCR to D, X, Y, or SP.
- All the bits in the upper byte of the 16-bit result are given the value of the most significant bit of the 8-bit number.

```
SEX   A, X
```




copies the contents of A to the lower bye of X and duplicates the bit 7 (= sign bit) of A to every bit of upper byte of X.

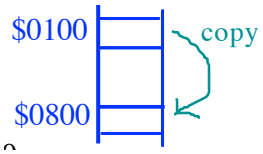
How to trasnfer 16 bits to 8 bits?

### 3. Move Instructions

- Move data bytes or words from a source address(es) to a destination address(es) in memory.
- Six combinations of immediate, extended, and index addressing modes are allowed to specify the source and destination addresses:

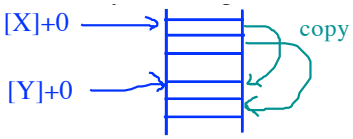
| | | |
|---|---|---|
| IMM ⇒ EXT | IMM ⇒ IDX | EXT ⇒ EXT |
| EXT ⇒ IDX | IDX ⇒ EXT | IDX ⇒ IDX |

```
MOVB      $0100, $0800      ; move 1 byte
```




copies the contents of the memory location at $0100 to the memory location at $0800.

```
MOVW      0, X, 0, Y        ; move 2 bytes
```




copies the 16-bit word pointed to by X with zero offset to the memory location pointed by Y with zero offset.

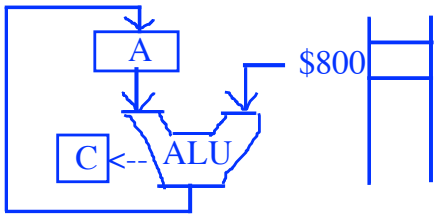
Table 1.6 Move instructions

| Transfer Instructions | | |
|---|---|---|
| Mnemonic | Function | Operation |
| MOVB | Move byte (8-bit) | (M1) ⇒ M2 |
| MOVW | Move word (16-bit) | $(M:M+1_1) \Rightarrow M:M+1_2$ |

What is MOVW 2,X+,2,Y+ ? What is MOVW X,Y ?

## 4. Add and Subtract Instructions

- Perform fundamental arithmetic operations.
- The destinations of these instructions are always a CPU register or accumulator.
- There are two-operand and three-operand versions of these instructions.
- Three-operand ADD or SUB instructions always include the C flag as one of the operand.



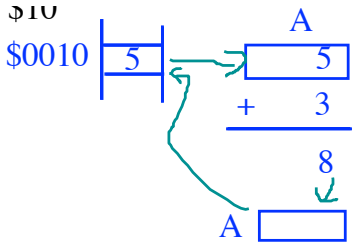

```
ADDA    $0800    ; A ← A + [$0800]
ADCA    $0800    ; A ← A + [$0800] + C        C = carry
SUBA    $0802    ; A ← A - [$0802]
SBCA    $0800    ; A ← A - [$0800] – C        C = borrow
```

Table 1.7 Add and subtract instructions

| Add Instructions | | | |
|---|---|---|---|
| Mnemonic | Function | Operation | |
| ABA | Add B to A | (A) + (B) ⇒ A | |
| ABX | Add B to X | (B) + (X) ⇒ X | |
| ABY | Add B to Y | (B) + (Y) ⇒ Y | |
| ADCA | Add with carry to A | (A) + (M) + C ⇒ A | |
| ADCB | Add with carry to B | (B) + (M) + C ⇒ B | |
| ADDA | Add without carry to A | (A) + (M) ⇒ A | |
| ADDB | Add without carry to B | (B) + (M) ⇒ B | |
| ADDD | Add without carry to D | (A:B) + (M:M+1) ⇒ A:B | 16 bits |
| **Subtract Instructions** | | | |
| Mnemonic | Function | Operation | |
| SBA | Subtract B from A | (A) - (B) ⇒ A | |
| SBCA | Subtract with borrow from A | (A) - (M) - C ⇒ A | |
| SBCB | Subtract with borrow from B | (B) - (M) - C ⇒ B | |
| SUBA | Subtract memory from A | (A) - (M) ⇒ A | |
| SUBB | Subtract memory from B | (B) - (M) ⇒ B | |
| SUBD | Subtract memory from D | (D) - (M:M+1) ⇒ D | 16 bits |

Example: Add 3 to the memory location at $10. What is the final value in $10?

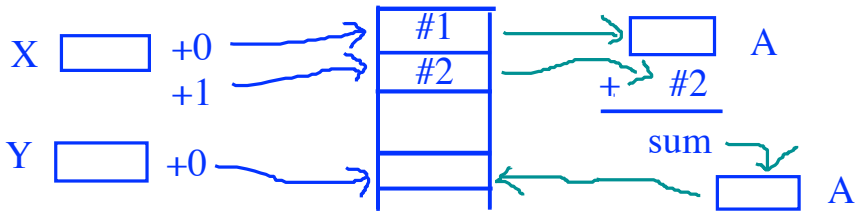
```
LDAA    $10     ; copy the contents at location $0010 into A
ADDA    #3      ; add 3 to A
STAA    $10     ; store the sum back to memory location $0010
```




What happens without the #?

Example: Add the byte in the memory location ***pointed*** to by index register X with the following byte, and place the sum at the memory location ***pointed*** to by index register Y.

```
LDAA    0, X    ; put the byte in the memory location pointed to by X in A
ADDA    1, X    ; add the following byte to A
STAA    0,Y     ; store the sum at the memory location pointed to by Y
```




Example: Add the numbers stored at $0800 and $0801 and store the sum at $0804.

```
LDAA    $0800   ; copy the number at location $0800 to A
ADDA    $0801   ; add the second number to A
STAA    $0804   ; copy the sum to $0804
```

End of Chapter 1