Introduction

The computer is made up of hardware and software. The hardware part of it consists of the following major components:

- The Processor.

  Control Unit, ALU, and Registers

- The Memory.

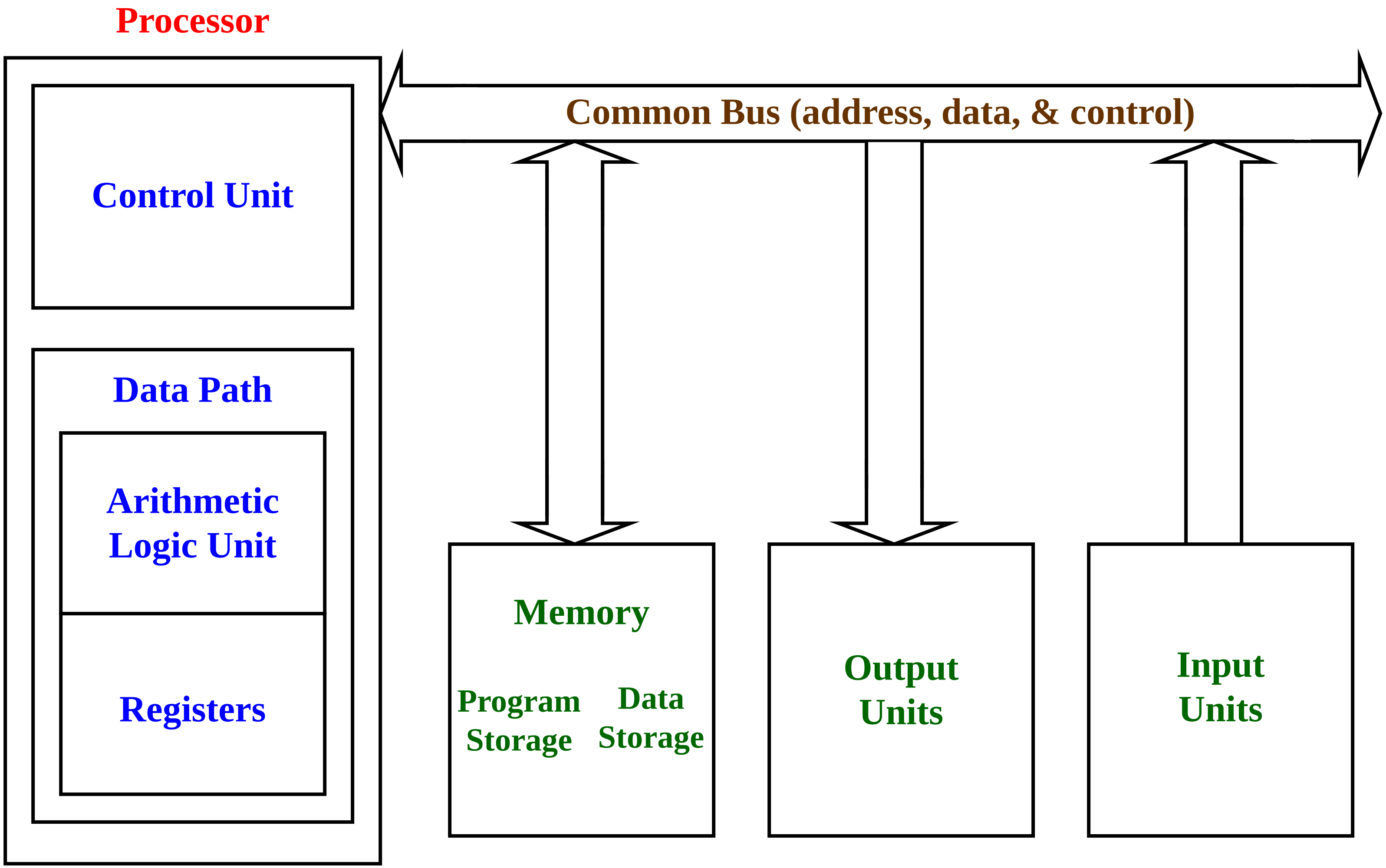
  SRAM, DRAM, ROM, EEPROM, FLASH, etc.

- Input Devices.

  Switches, Keypad, Keyboard, Mouse, Microphone, etc.

- Output Devices.

  CRT, LEDs, 7-segment display, LCD, Printers, etc.



Computer Organization

Introduction *continued …*

- Recent advances in electronic semiconductor technology have resulted in the development of highly integrated *microprocessors* and *microcontrollers*.
- *Microprocessors* are used not only in many desktop personal computers but also in many more embedded applications such as fax and copying machine, laser printers, and communication controllers.
- To build a complete system based on a microprocessor, the designer needs external memories, peripheral interface chips and other logic circuits.
- A *Microcontroller* incorporates an adequate amount of memory, timers, and other peripheral functions in one chip along with the CPU.

Introduction *continued …*

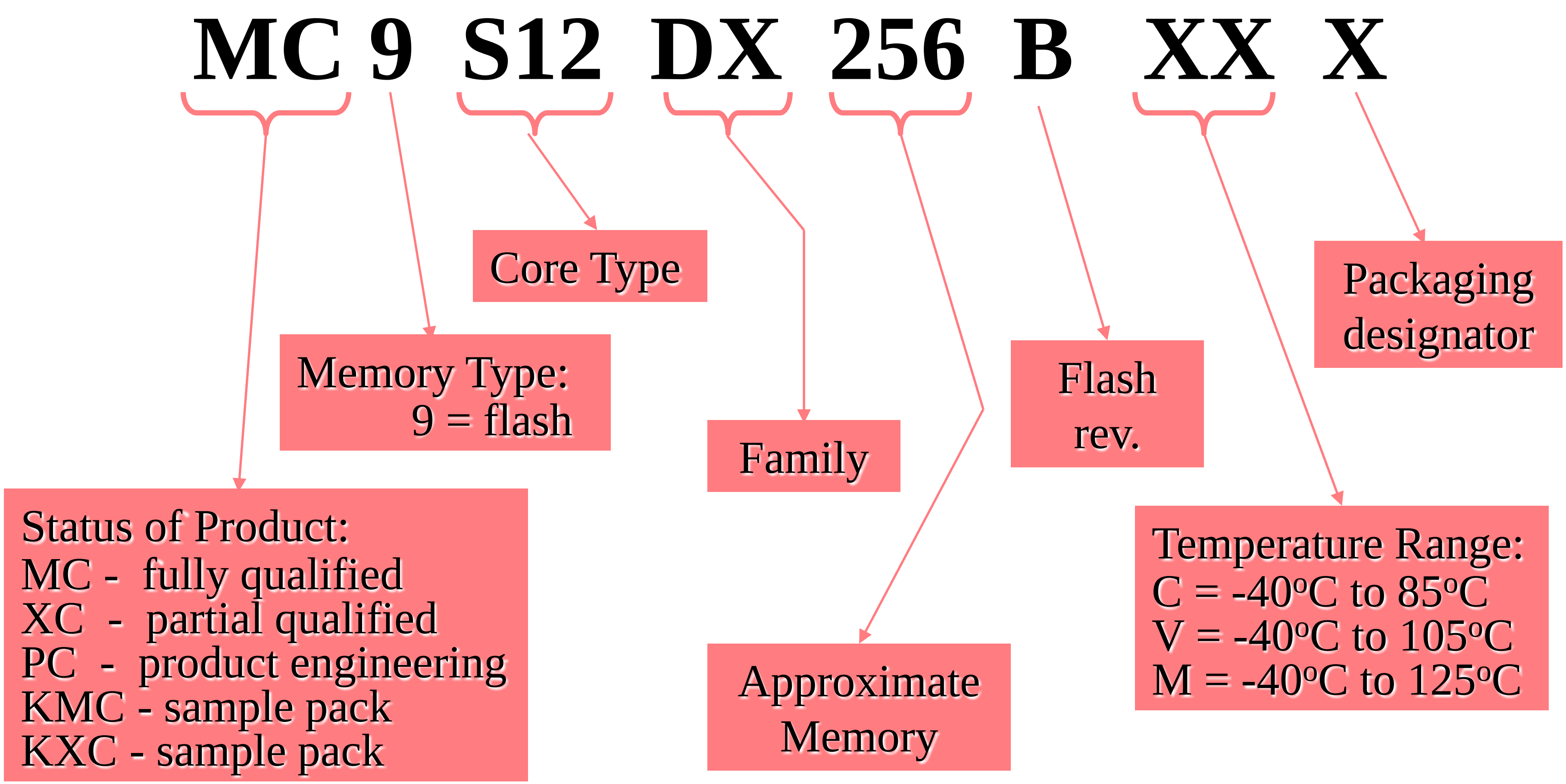
- These extra resources on board of a microcontroller are adequate for many embedded applications.

- Microcontrollers not only simplify the design of many embedded products but also have the advantage of smaller size and lower power consumption.

- This course is intended to teach how to use a microcontroller in the design of an instrument or any embedded product.

- To achieve this goal, the Motorola MC9S12DP256 microcontroller will be used to introduce various concepts of interfacing and the design of embedded systems.

Motorola Product Numbering System for the HCS12



MC 9 S12 DX 256 B XX X

Core Type

Memory Type:
9 = flash

Family

Flash rev.

Packaging designator

Status of Product:
MC - fully qualified
XC - partial qualified
PC - product engineering
KMC - sample pack
KXC - sample pack

Approximate Memory

Temperature Range:
C = -40^{o}C to 85^{o}C
V = -40^{o}C to 105^{o}C
M = -40^{o}C to 125^{o}C

- The Motorola 68HC12 family microcontrollers were introduced in 1996 as an upgrade for the 68HC11 microcontrollers.

- The HCS12 family was later designed to improve the performance of the 68HC12 family.

- The HCS12 family supports the same instruction set and addressing modes but provides much higher external bus speed.

- The HCS12 family also differs in the amount of on-chip memory and peripheral functions.

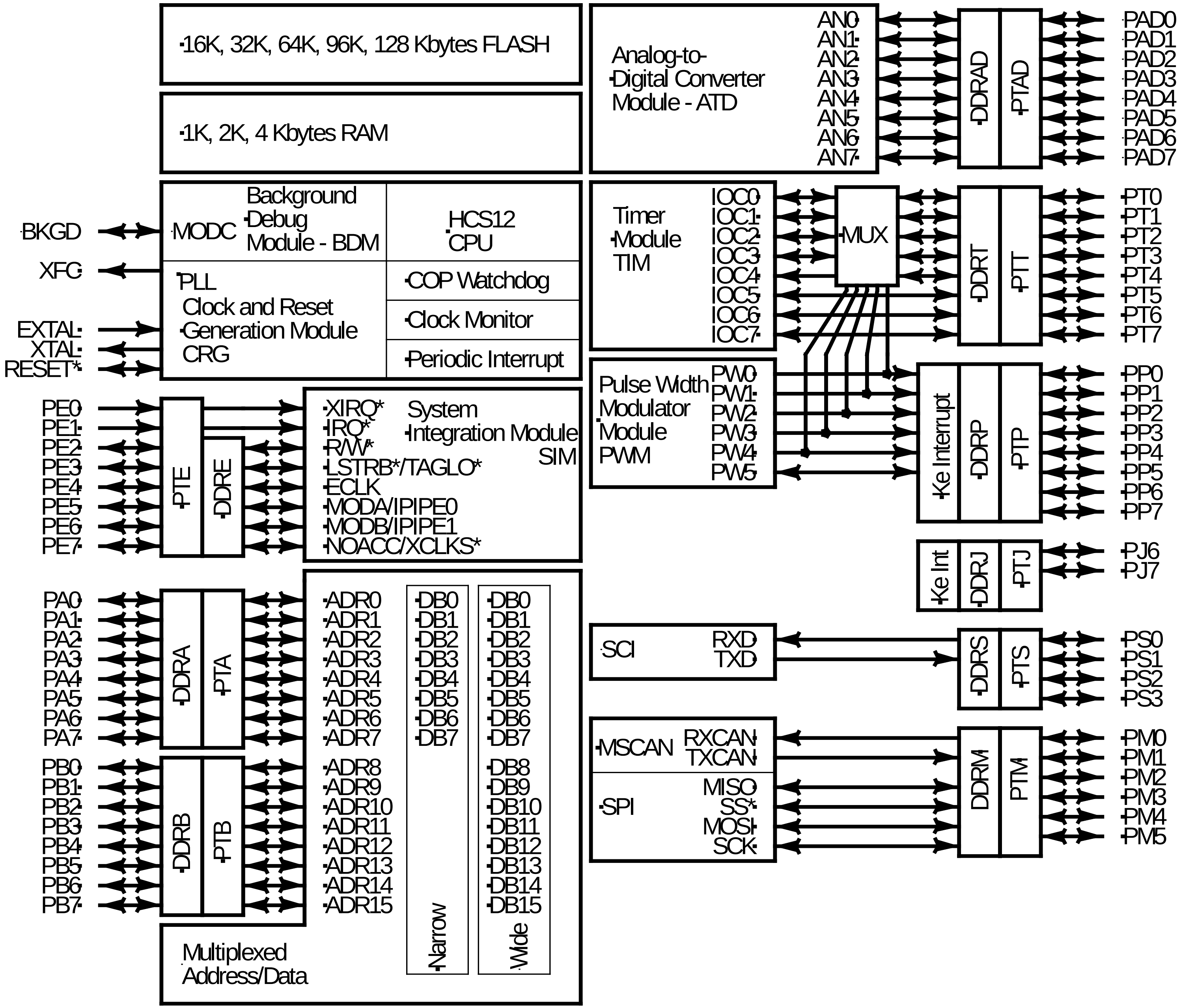
- The HCS12 family has the following features:

Introduction *continued …*

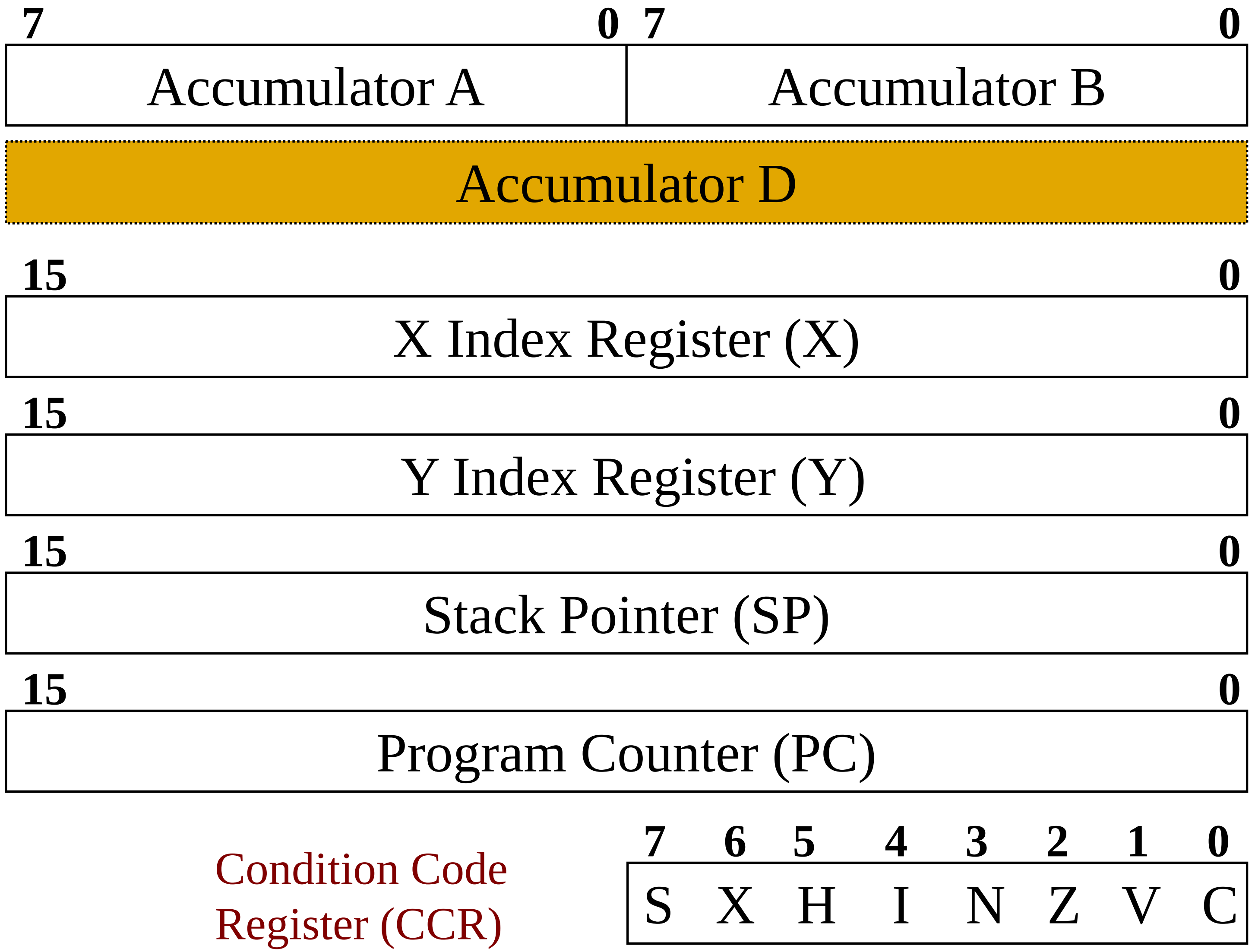
- 16-bit CPU
- Multiplexed (add. & data) bus
- 2 to 14 KB of on-chip SRAM
- 16 to 512 KB of on-chip Flash
- Pulse-width modulation (PWM)
- Inter-Integrated Circuit interface
- Byte data link controller (BDLC)
- Computer Operating Properly
- Instruction for Fuzzy Logic support

- Standard 64 KB address space
- 0 to 4 KB of on-chip EEPROM
- 10-bit A/D converter
- Timer module (IC, OC, PA)
- Synchronous Peripheral Interface
- Asynchronous serial comm.
- Controller Area Network (CAN)
- Background Debug Mode (BDM)

Introduction *continued …*



HCS12 Programmer’s Model



7 0 7 0

Accumulator A

Accumulator B

Accumulator D

15 0

X Index Register (X)

15 0

Y Index Register (Y)

15 0

Stack Pointer (SP)

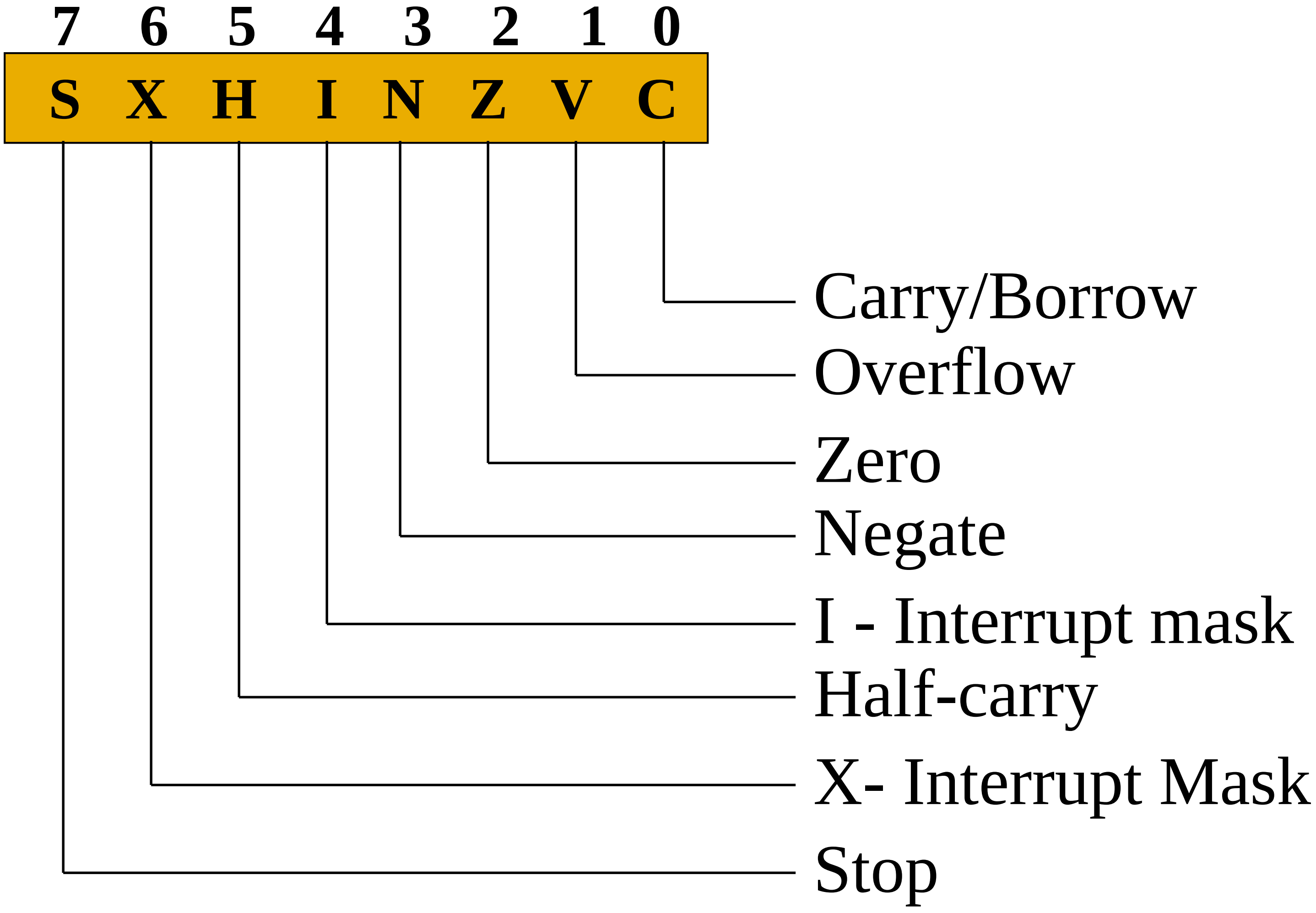
15 0

Program Counter (PC)

Condition Code Register (CCR)

7	6	5	4	3	2	1	0
S	X	H	I	N	Z	V	C

Condition-Code Register



General-purpose Accumulators A & B:

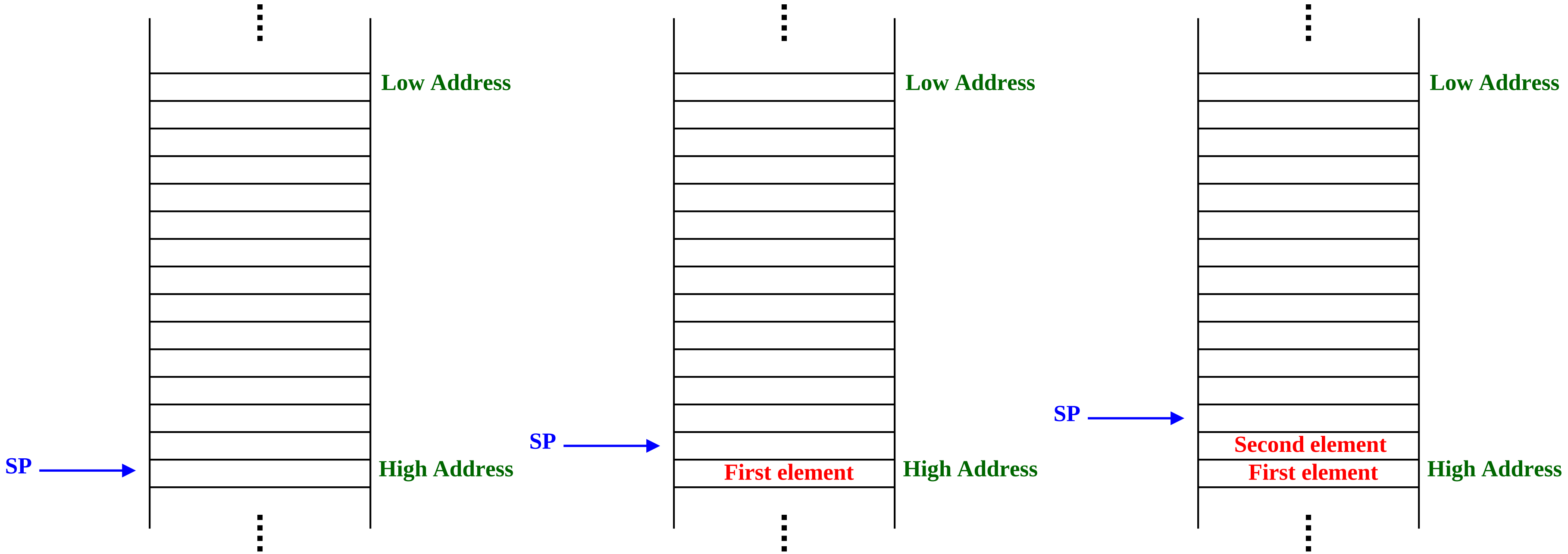
- Both A and B are 8-bit registers. Most arithmetic functions are performed on these two registers. These two accumulators can also be concatenated to form a single, 16-bit accumulator referred to as the D accumulator.

Indexed registers X & Y:

- These two registers are used mainly in forming operand addresses during the instruction execution process. However, they are also used in several arithmetic operations.

Stack pointer (SP):

- The stack is a last-in-first-out (LIFO) structure. The HCS12 has a 16-bit stack pointer that points to the top byte of the stack. The stack grows toward lower addresses.



HCS12 Stack Structure

Program counter (PC):

- The 16-bit PC holds the address of the next instruction to be executed. After the execution of an instruction, the PC is incremented by the number of bytes of the executed instruction.

Condition code register (CCR):

- The 8-bit register is used to keep track of the program execution status, control the execution of conditional instructions, and enable/disable the interrupt handling.

The HCS12 supports the following types of data:

- Bits
- 8-bit signed & unsigned integer
- 9-bit signed integer
- 16-bit effective addresses
- 5-bit signed integer
- 8-bit binary-coded-decimal BCD
- 16-bit signed & unsigned integer
- 32-bit signed & unsigned integer

➢ Negative numbers are represented in *2's complement* format.

➢ 5-bit, 9-bit signed integers and 16-bit effective addresses are formed during addressing mode computations.

- 32-bit integer dividends are used by extended division instructions.

- Extended multiply and extended multiply-and-accumulate instructions produce 32-bit products.

- A multi-byte integer (16-bit or 32-bit) 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.

- An appropriate prefix is added in front of the number to indicate its base:

Base	Prefix	Example
Binary	%	%10001010
Octal	@	@1234567
Decimal		28749026
Hexadecimal	$	$3A67B0CD
ASCII characters	‘	‘Hi There!’

The Computer's Software

- Programs are known as *software*. A *program* is a set of instructions that computer hardware can execute.
- A program is stored in the computer memory in the form of binary numbers called *machine instructions*.
- For example, the HCS12 machine instruction

  0001 1000 0000 0110

  adds the content of accumulator B and accumulator A together and leaves the sum in accumulator A.
- The machine instruction

  0100 0011

  Decrements the content of accumulator A by 1.
- Writing programs in machine language is extremely difficult and inefficient.

- *Program entering ………*
- *Program debugging ………*
- *Program maintenance ………*

- *Assembly language* was invented to simplify the programming job.
- An *assembly program* consists of assembly instructions.
- An *assembly instruction* is the mnemonic representation of a machine instruction.

0001 1000 0000 0110 → ABA (A + B → A)

- The assembly program that programmer enters is called *source program* or *source code*.
- A software program called an *assembler* is then invoked to translate the program written in *assembly language* into *machine instructions*.
- The output of the assembly process is called *object code*.

The Computer’s Software *continued …*

- A *native assembler* or simply assembler runs on a computer and generate machine instructions to be executed by the machines that have the same instruction set.
- It is a common practice to use a *cross assembler* to translate assembly programs, which is an assembler that runs on one computer but generate machine instructions to be run at the different computer with a totally different instruction set.
- The freeware **as12** is a *cross assembler* that runs on an IBM PC and generates machine code that can be downloaded into a HC12-based computer for execution.
- There are drawbacks to assembly language programs.
  - Needs familiarity with hardware organization of computer.
  - Without good documentation, it is hard to follow.
  - Programming productivity is low for large programs.

The Computer’s Software *continued …*

- There are advantages to assembly programming as well.
  - The resulting machine code is very efficient and runs fast.
  - Assembly programming gives access to details of the computer hardware (bit access).
  - Most drivers written for different peripherals are either entirely in assembly language or good portion of it is written in assembly (subroutine calls).

Number Systems

There are two fundamental elements to number system:

1. The base or radix of the system
2. The value assigned to the position of a digit.

Example:

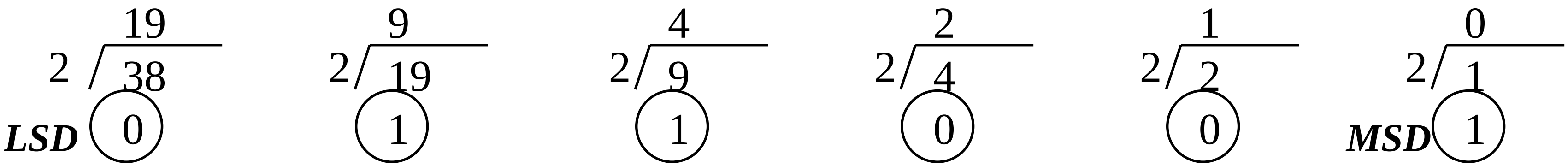
$$(103)_4 = 3\times4^0 + 0\times4^1 + 1\times4^2 = 3 + 16 = 19_{10}$$

$$(257)_8 = 7\times8^0 + 5\times8^1 + 2\times8^2 = 7 + 40 + 128 = 175_{10}$$

How do we go from base 10 to base 2?

Example: $38_{10} = (\ ?\ )_2$

$38_{10} = (100110)_2$



How do we take care of decimal point?

Ex.: $124.078_{10} = (\quad ? \quad)_2$

$124_{10} = (1111100)_2$

$.078 \times 2 = 0.156$	0
$.156 \times 2 = 0.312$	0
$.312 \times 2 = 0.624$	0
$.624 \times 2 = 1.248$	1
$.248 \times 2 = 0.496$	0
$.496 \times 2 = 0.992$	0
$.992 \times 2 = 1.984$	1
$.984 \times 2 = 1.968$	1
$.968 \times 2 = 1.936$	1
.	.

MSD after binary point

$124.078_{10} = 1111100.000100111_2$

Number System *continued …*

Ex.: $109.163_{10} = (\quad ? \quad)_2$

$$109_{10} = (1101101)_2$$

$.163 \times 2 = 0.326$	0
$.326 \times 2 = 0.652$	0
$.652 \times 2 = 1.304$	1
$.304 \times 2 = 0.608$	0
$.608 \times 2 = 1.216$	1
$.216 \times 2 = 0.432$	0
$.432 \times 2 = 0.864$	0
$.864 \times 2 = 1.728$	1
$.728 \times 2 = 1.456$	1
⋮	⋮

MSD after decimal point ↓

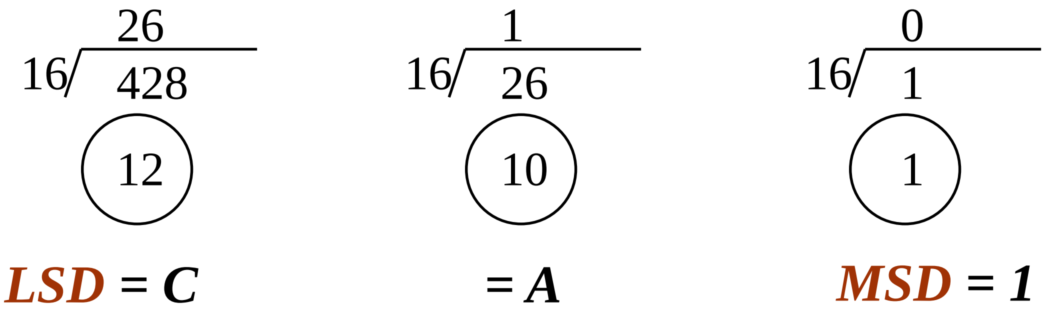
$$109.163_{10} = 1101101.001010011_2$$

How do we go from base 10 to base 16 (Hexadecimal)?

Ex.: $428_{10} = (\ ?\ )_{16}$

Note! Base 16 needs sixteen symbols:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F



26
16/ 428
12

LSD* = *C

1
16/ 26
10

= A

0
16/ 1
1

MSD* = *1

$428_{10} = (1AC)_{16}$

How to convert Binary to Hexadecimal:

Simply partition binary digits into groups of 4-bit starting from right.

$$\underbrace{110}_{\mathbf{6}}\underbrace{0111}_{\mathbf{7}}\underbrace{0110}_{\mathbf{6}}\underbrace{0010}_{\mathbf{2}}\underbrace{1100}_{\mathbf{C}} = 6762C_{16}$$

How to convert Binary to Octal:

Simply partition binary digits into groups of 3-bit starting from right.

$$\underbrace{101}_{2}\underbrace{001}_{4}\underbrace{011}_{5}\underbrace{100}_{4}\underbrace{111}_{7} = 24547_8$$

The Binary-Coded Decimal System (BCD):

Each decimal digit is represented by 4-bit binary number.

0 →	0000	4 →	0100	8 →	1000
1 →	0001	5 →	0101	9 →	1001
2 →	0010	6 →	0110		
3 →	0011	7 →	0111		

Example: Represent 28_{10} in both binary and BCD using 8-bits.

Binary → 00011100 BCD → 00101000

Binary Addition:

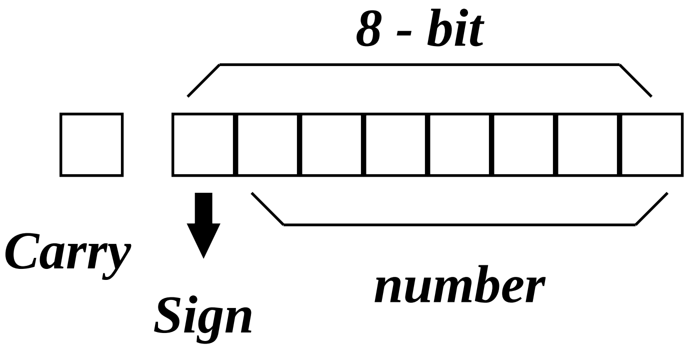
Lets add 29_{10} and 11_{10} in binary 6-bit word;

$$\begin{array}{rr} & 0\,1\,1\,1\,0\,1 \\ + & 0\,0\,1\,0\,1\,1 \\ \hline & 1\,0\,1\,0\,0\,0 \end{array} \quad = 40_{10}$$

Ex.: Add 171_{10} to 202_{10} in binary 8-bit word.

$$\begin{array}{rr} & 1\,0\,1\,0\,1\,0\,1\,1 \\ + & 1\,1\,0\,0\,1\,0\,1\,0 \\ \hline & \underset{\swarrow\ \textbf{\textit{Carry bit}}}{1}\,\underbrace{0\,1\,1\,1\,0\,1\,0\,1}_{\textbf{\textit{8 - bit}}} \end{array}$$

Binary Subtraction:



Using ***1's Complement***:

Ex. 1) $13_{10} - 10_{10} = ?$

$$13_{10} \longrightarrow 0\ 0001101 \xrightarrow{\textit{1's Comp}} 0\ 0001101$$

$$-10_{10} \longrightarrow 1\ 0001010 \xrightarrow{\textit{1's Comp}} 1\ 1110101$$

Now add them;

end around carry

$$\begin{array}{r} 0\ 0001101 \\ 1\ 1110101 \\ \hline (1)\ 0\ 0000010 \\ 1 \\ \hline 0\ 0000011 = +3_{10} \end{array}$$

Ex. 2) $10_{10} - 13_{10} = \ ?$

$$10_{10} \longrightarrow 0\ 0001010 \xrightarrow{\textit{1's Comp}} 0\ 0001010$$

$$-13_{10} \longrightarrow 1\ 0001101 \xrightarrow{\textit{1's Comp}} 1\ 1110010$$

Now add them;

$$\begin{array}{r l}
0\ 0001010 & \\
1\ 1110010 & \\
\hline
1\ 1111100 & = -3_{10}
\end{array}$$

Ex. 3) $-13_{10} - 10_{10} = \ ?$

$$-13_{10} \longrightarrow 1\ 0001101 \xrightarrow{\textit{1's Comp}} 1\ 1110010$$

$$-10_{10} \longrightarrow 1\ 0001010 \xrightarrow{\textit{1's Comp}} 1\ 1110101$$

end around carry

$$\begin{array}{r l}
1\ 1110101 & \\
\hline
(1)\ 1\ 1100111 & \\
1 & \\
\hline
1\ 1101000 & = -23_{10}
\end{array}$$

Using *2's Complement*:

Ex. 1) $13_{10} - 10_{10} = ?$

$$13_{10} \longrightarrow 0\ 0001101 \xrightarrow{\textit{2's Comp}} 0\ 0001101$$

$$-10_{10} \longrightarrow 1\ 0001010 \xrightarrow{\textit{2's Comp}} 1\ 1110101$$

add 0ne $+ \quad 1$

$$1\ 1110110$$

Now add them;

$$\begin{array}{r} 0\ 0001101 \\ 1\ 1110110 \\ \hline (1)\ 0\ 0000011 = +3_{10} \end{array}$$

In 2's Complement always disregard the carry bit

Note! There is a short cut in finding 2’s complement of a negative number. Start from most right to the left till you encounter first 1, up to there (including first 1) stays the same from there on complement.

Ex. 2) $10_{10} - 13_{10} = ?$

$10_{10} \longrightarrow 0\ 0001010 \xrightarrow{\textit{2's Comp}} 0\ 0001010$

$-13_{10} \longrightarrow 1\ 0001101 \xrightarrow{\textit{2's Comp}} 1\ 1110011$

Now add them;

$$\begin{array}{r} 0\ 0001010 \\ 1\ 1110011 \\ \hline 1\ 1111101 \end{array} = -3_{10}$$

Ex. 3) $-13_{10} - 10_{10} = \ ?$

$$-13_{10} \longrightarrow 1\ 0001101 \xrightarrow{\textbf{\textit{2's Comp}}} 1\ 1110011$$

$$-10_{10} \longrightarrow 1\ 0001010 \xrightarrow{\textbf{\textit{2's Comp}}} 1\ 1110110$$

Now add them;

$$\begin{array}{r} 1\ 1110011 \\ 1\ 1110110 \\ \hline (1)\ 1\ 1101001 \end{array} = -23_{10}$$

disregard

Ex.: Use 1's and 2's complement to find the result of the following statement: -38 - 55 = ?

1's Comp:

$$-38_{10} \longrightarrow 1\ 0100110 \xrightarrow{\text{1's Comp}} 1\ 1011001$$

$$-55_{10} \longrightarrow 1\ 0110111 \xrightarrow{\text{1's Comp}} 1\ 1001000$$

$$\begin{array}{r} 1\ 1011001 \\ \underline{1\ 1001000} \\ (1)\ 1\ 0100001 \\ \underline{1} \\ 1\ 0100010 = -93_{10} \end{array}$$

end around carry

2's Comp:

$$-38_{10} \longrightarrow 1\ 0100110 \xrightarrow{\text{2's Comp}} 1\ 1011010$$

$$-55_{10} \longrightarrow 1\ 0110111 \xrightarrow{\text{2's Comp}} 1\ 1001001$$

$$\begin{array}{r} 1\ 1011010 \\ \underline{1\ 1001001} \\ (\cancel{1})\ 1\ 0100011 = -93_{10} \end{array}$$

Disregard carry

Memory Addressing

- Memory consist of a sequence of directly addressable “locations.”
- A memory location can be used to store *data*, *instruction*, and the *status of a peripheral device*.
- Each memory unit has two components: its *address* and its *contents*.
- Each location in memory has an address that must be supplied before its contents can be accessed.
- 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 CPU along the *data bus*.
- The number of bits that can be transferred on the data bus at once is called *data bus width* of the processor.
- The size of memory is measured in bytes; a *byte* consists of 8-bits.
- A 4-bit quantity is called a *nibble*; a 16-bit quantity is called a *word*.

Memory Addressing *continued …*

- To simplify quantification of memory, the unit *kilobyte* (*kB*) is often used. *k* is given by the following formula

$$k = 2^{10} = 1024$$

- Another frequently used unit is *megabyte* (*MB*), which is given by the following formula

$$M = k^2 = 2^{20} = 1024 \times 1024 = 1048576$$

- The HCS12 has a 16-bit address bus and 16-bit data bus. This will allow the HCS12 to access 16-bit data in one operation.

- The 16-bit address bus enables the HCS12 to address directly up to 2^{16} (65536) different memory locations.

- Most HCS12 members use *paging techniques* to allow user programs to access more than 64 *kB*.

- In this course we will use the notation **m[addr]** and **[reg]** to refer to the contents of a memory location at **addr** and the content of the register, **reg**, respectively.

The Instruction

Operation performed by MicroProcessor Unit (MPU) can be described in two different ways:

Statistically - as a collection of bits stored in memory or
a line of a program.

Dynamically - as a sequence of action by controller.

Controller will send commands to memory or any other parts of the computer to effectively carry out the intentions of the programmer.

To execute the program, the MPU controller repeatedly executes the ***instruction cycle*** or ***fetch/execute cycle***:

1. **Read** the next instruction from the memory.
2. **Execute** the instruction.

In order to accomplish this, manufacturers design into their instruction set a number of different ***operational code*** (*op-codes*) and ***addressing modes***.

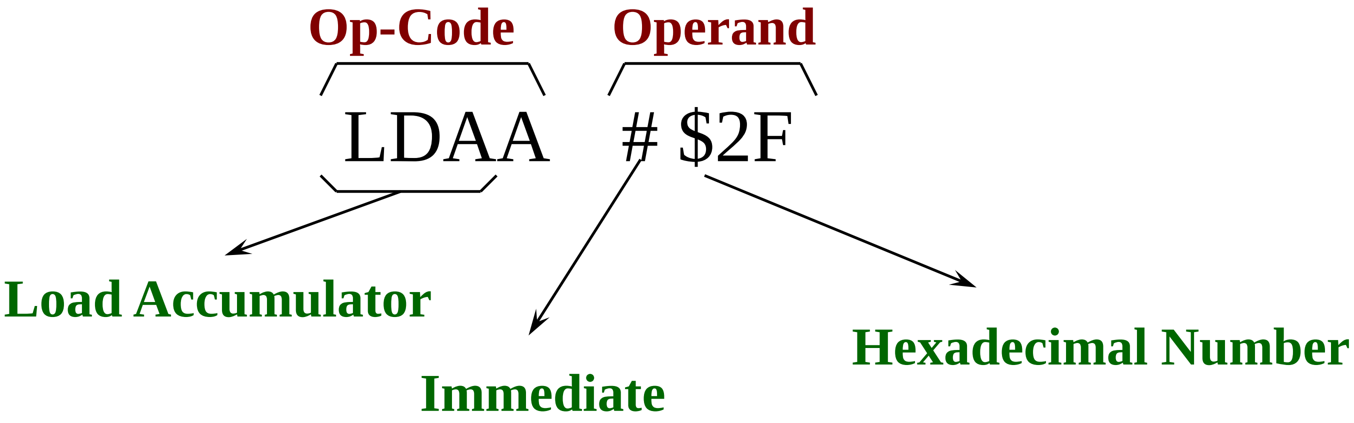
The Instruction *continued …*

- The ***op-code*** “tells” the CPU what to do;
- The ***addressing mode*** “tells” the CPU how to obtain the data.

Addressing modes providing the programmer with various ways of handling data to speed up the operation process and/or reduce the number of instructions required to write the program.

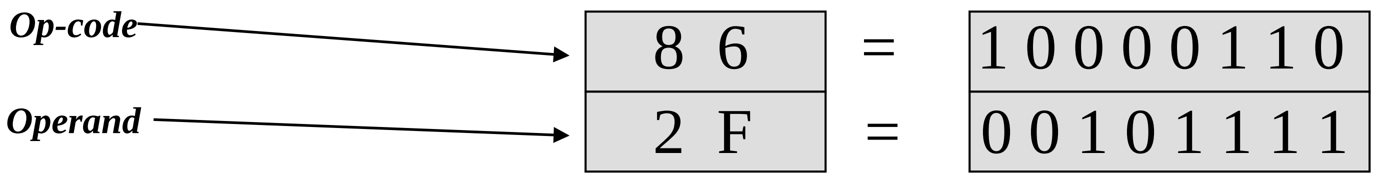
Note! Any time we are reading an instruction from memory it is called “***fetch***”. Otherwise it is “***read***” or “***recall***”.

Lets put a specific number, say $2F, in accumulator A using ***Immediate Addressing*** mode.



The Instruction *continued …*

The instruction is stored in memory as a two consecutive bytes.



Op-code →	8 6	=	1 0 0 0 0 1 1 0
Operand →	2 F	=	0 0 1 0 1 1 1 1

This is *statistical* presentation of an instruction.

The *Dynamic* Operation of Fetch/Execute:

1. Fetch the first byte of the instruction from memory,
2. Increment the PC by one,
3. Decode the ***Op-code*** that was fetched in step 1,
4. Repeat steps 1 and 2 to fetch all bytes of the instruction,
5. Calculate the ***effective address*** to access memory, if needed,
6. Recall the operand from memory, if needed,
7. Execute the instruction, which may include writing the result into memory.

Instruction Cycle: The number of clock cycles that takes to carry out the instruction.

Clock Cycle Activities for “LDAA #$2F”

This instruction needs **2** clock cycles. Following table shows the activities in each clock cycle.

Clock Cycle One	PC	into	Address Bus
	PC + 1	into	PC
	[memory location]	into	Data Bus
	Data Bus	into	Instruction Register
Clock Cycle Two	PC	into	Address Bus
	PC + 1	into	PC
	[memory location]	into	Data Bus
	Data Bus	into	Accumulator A

Addressing Modes

- ➢ The *effective address* is the address where the data is located.

- ➢ How the effective address of certain memory location is determined is called the *addressing mode*.

- ➢ HCS12 has **6** addressing modes:

  ❶ Inherent

  ❷ Immediate

  ❸ Direct

  ❹ Extended

  ❺ Relative

  ❻ Indexed

Inherent Addressing Mode:

- Instructions that use this addressing mode either have no operand or all operands are in internal CPU registers.

- The CPU does not need to access any memory locations to complete the instruction.

```
NOP             ;this instruction has no operand

INX             ;operand is a CPU register
```

Immediate Addressing Mode:

- Operand for immediate mode instructions are included in the instruction stream.

- The CPU does not access memory when this type of instruction is executed.

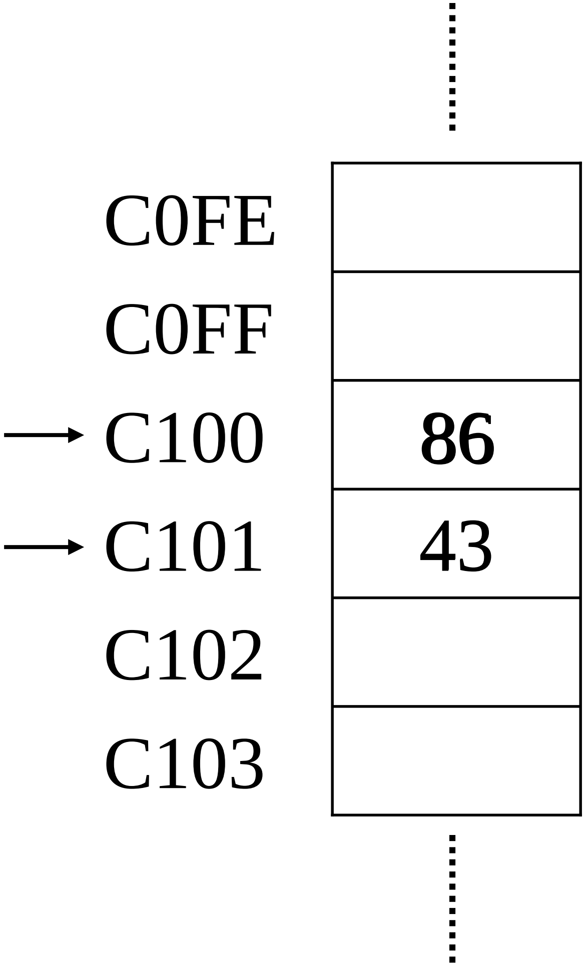
- An immediate value can be 8-bit or 16-bit depending on the context of instruction and preceded by a # character in the assembly instruction.

Example of Immediate Addressing Mode

Location	Op-code	Operand
…	…	…
C100	LDAA	#’C’
…	…	…

Instruction Register:

A:



C0FE	
C0FF	
C100	86
C101	43
C102	
C103	

Direct Addressing Mode:

- This addressing mode sometimes called zero-page addressing because it is used to access operands in the address range of \$0000 - \$00FF.

- Since these addresses begin with \$00, only the eight low-order bits of the address needs to be included in the instruction, which saves program space and execution time.

$\text{LDAA} \quad \$20 \qquad ; A \leftarrow m[\$20]$

$\text{LDX} \quad \$1B \qquad ; X_H \leftarrow m[\$1B], X_L \leftarrow m[\$1C]$

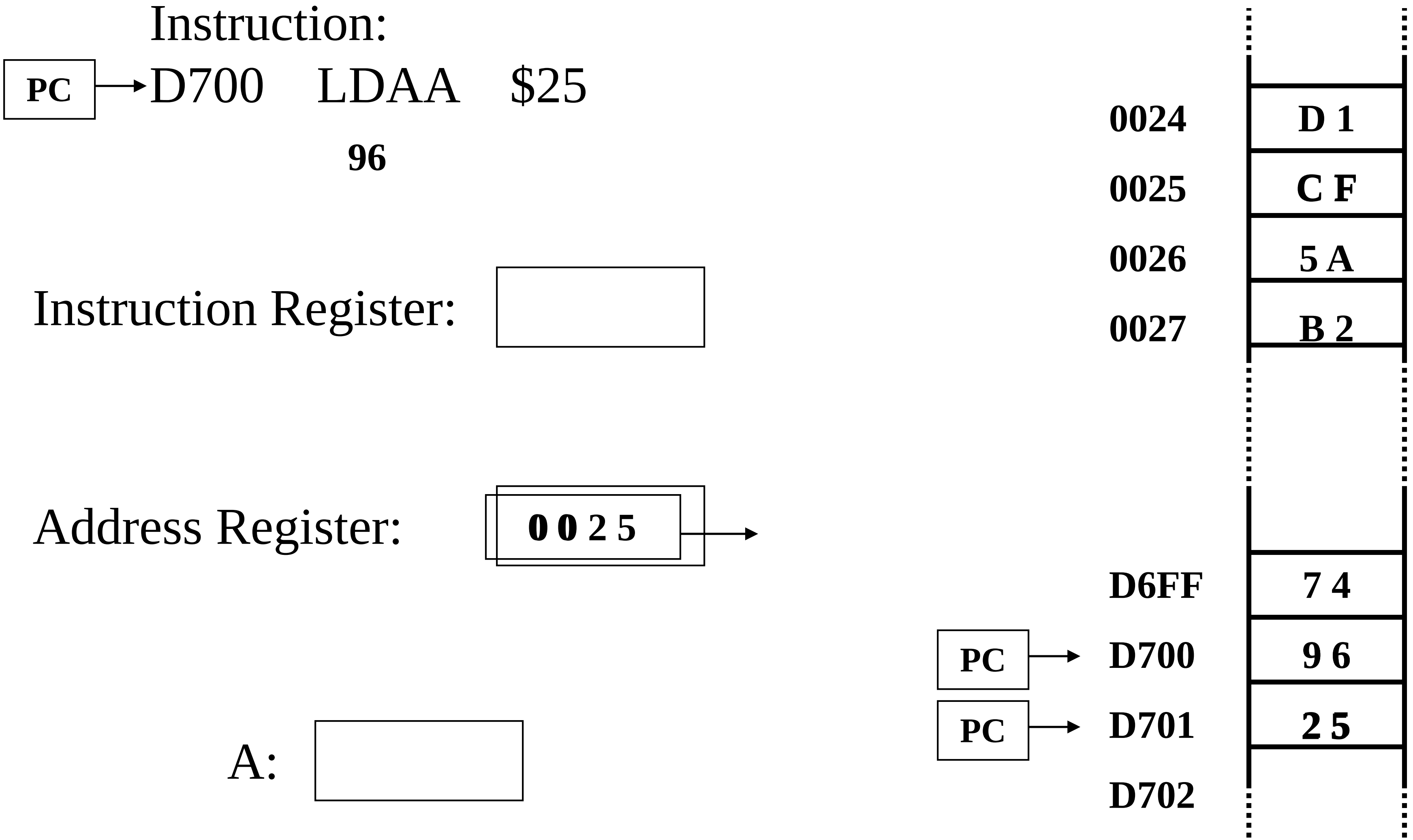
Extended Addressing Mode:

- In this addressing mode, the full 16-bit address of memory location to be operated on is provided in the instruction.

$\text{LDAB} \quad \$2000 \qquad ; B \leftarrow m[\$2000]$

$\text{LDY} \quad \$2500 \qquad ; Y_H \leftarrow m[\$2500], Y_L \leftarrow m[\$2501]$

Example of Direct Addressing Mode



Instruction:

PC → D700 LDAA $25

96

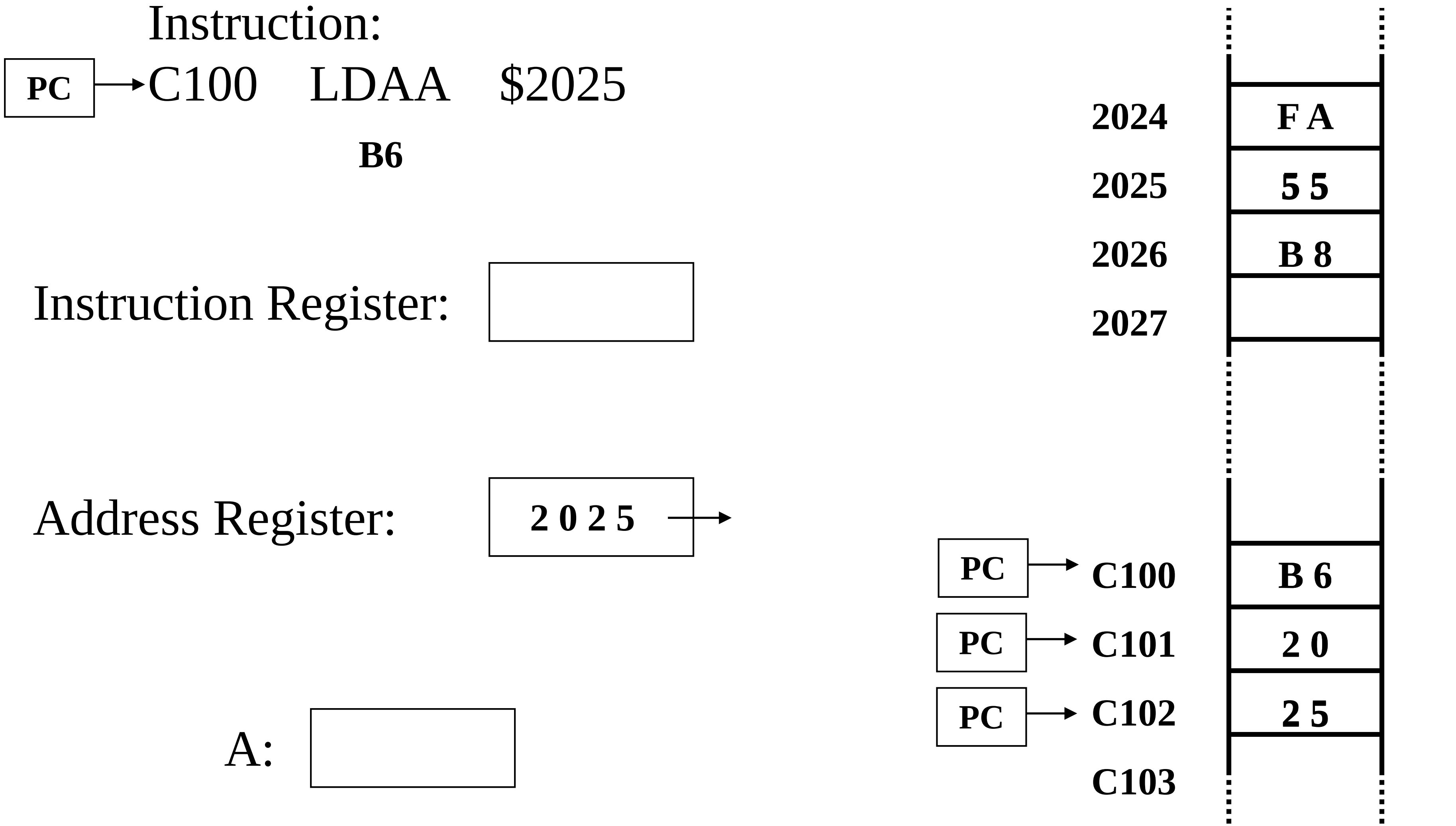
Instruction Register:

Address Register: 0 0 2 5

A:

0024	D 1
0025	C F
0026	5 A
0027	B 2
D6FF	7 4
D700	9 6
D701	2 5
D702	

Example of Extended Addressing Mode



Instruction:

PC → C100 LDAA $2025

B6

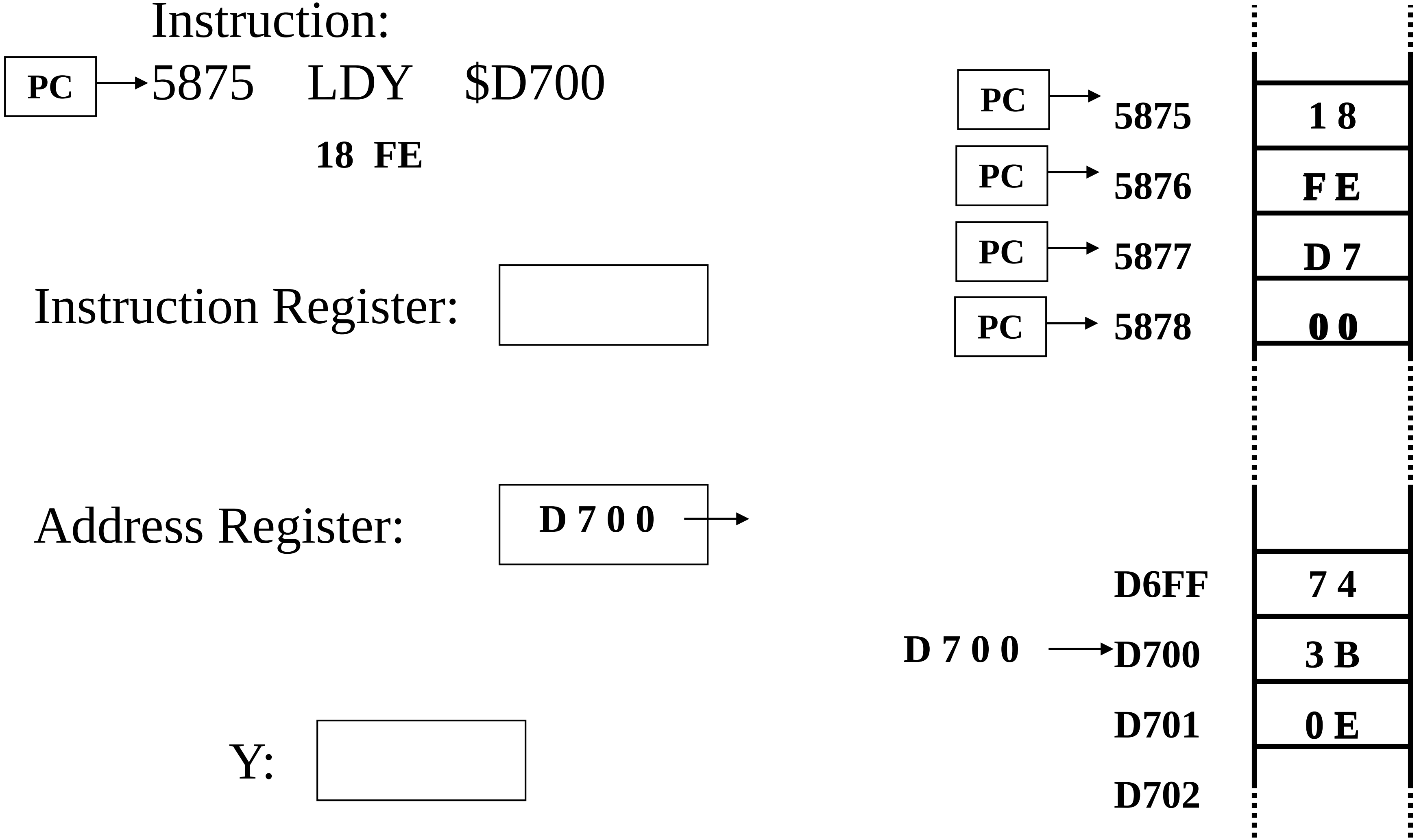
Instruction Register:

Address Register: 2 0 2 5 →

A:

	2024	F A
	2025	5 5
	2026	B 8
	2027	
PC →	C100	B 6
PC →	C101	2 0
PC →	C102	2 5
	C103	

Example of Extended Addressing Mode



Instruction:

PC → 5875 LDY $D700

18 FE

Instruction Register:

Address Register: D 7 0 0

Y:

PC	5875	1 8
PC	5876	F E
PC	5877	D 7
PC	5878	0 0
	D6FF	7 4
D 7 0 0	D700	3 B
	D701	0 E
	D702	

Relative Addressing Mode:

- The relative addressing mode is only used by branch instructions.

- A short branch instruction consists of an 8-bit op-code and a signed 8-bit offset contained in the byte that follows the op-code.

- Long branch instructions consist of an 8-bit pre-byte, an 8-bit op-code, and a signed 16-bit offset contained in two bytes that follow the op-code.

- Each conditional branch instruction tests certain status bits in the condition code register.

- Both 8-bit and 16-bit offsets are signed two's complement numbers to support branching forward or backward in memory.

- The offset is added to the address of the next memory location after the offset to form an effective address, and execution continues at that address.

```
BPL     msg1    ; operand is 8-bit signed (-128 to 127)
LBPL    msg2    ; operand is 16-bit signed (-32768 to
                ;                               32767)
```

Example of Short-Relative Addressing

Instruction:

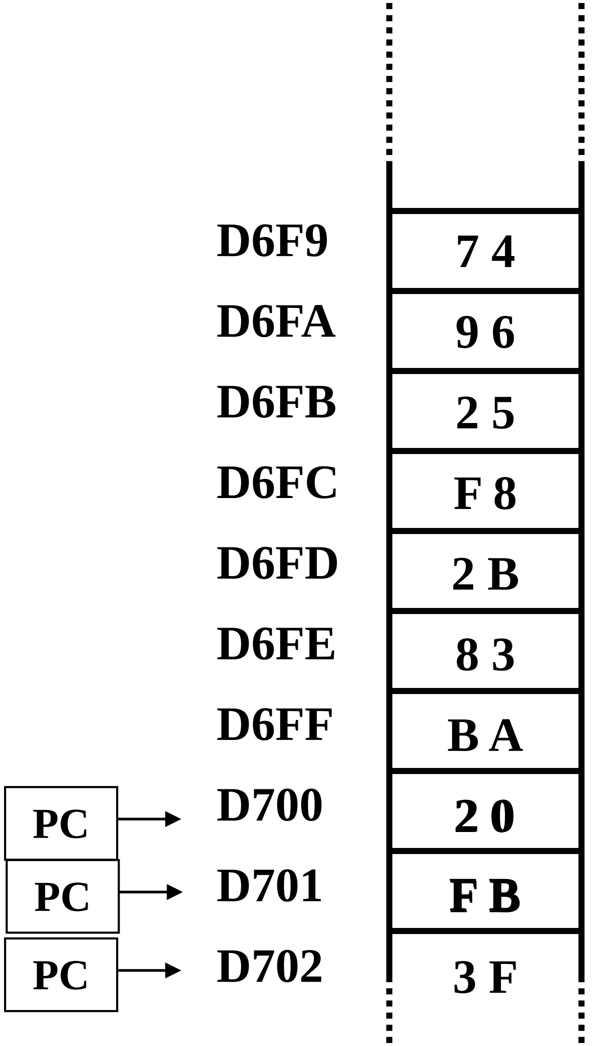
PC → D700 BRA $FB

20

Instruction Register:

Program Counter:

~~1~~ D 6 F D



D6F9	7 4
D6FA	9 6
D6FB	2 5
D6FC	F 8
D6FD	2 B
D6FE	8 3
D6FF	B A
PC → D700	2 0
PC → D701	F B
PC → D702	3 F

Examples of Short-Relative Addressing

Find the content of the Program Counter (PC) after execution of the following instructions:

C28A BRA $FA

2’s complement of -6

[PC] = [PC] + 2 + offset

[PC]+2 in 2’s complement

```
  1100 0010 1000 1100
  1111 1111 1111 1010
-----------------------
1 1100 0010 1000 0110
```

[PC] = $C286

C800 BRA $DB

2’s complements of -37

[PC] = [PC] + 2 + offset

[PC]+2 in 2’s complement

```
  1100 1000 0000 0010
  1111 1111 1101 1011
-----------------------
1 1100 0111 1101 1101
```

[PC] = $C7DD

Example of Long-Relative Addressing

Instruction:

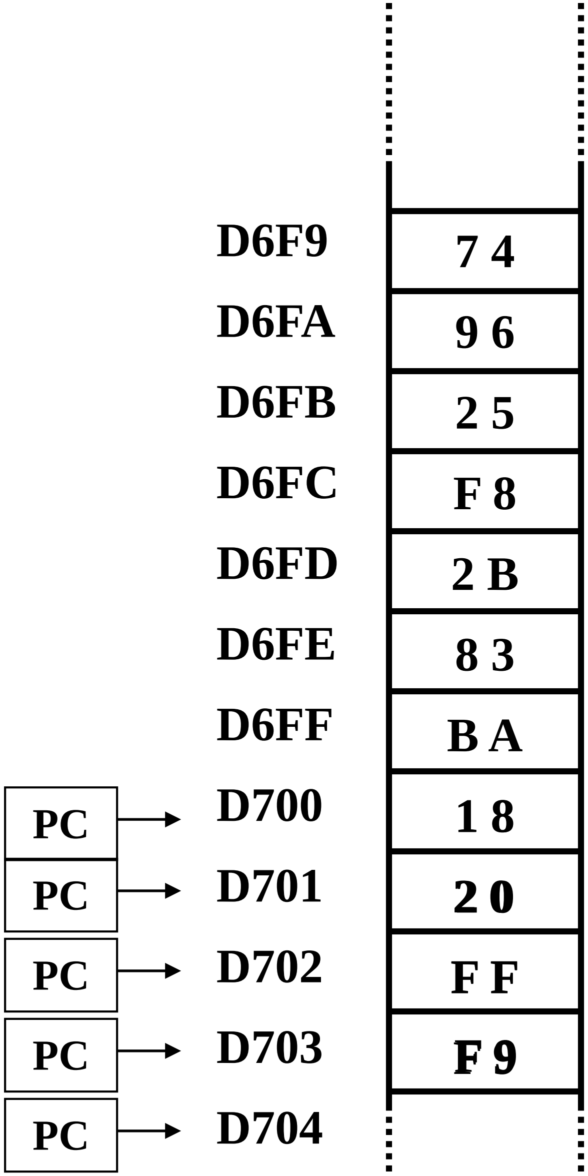
PC → D700 LBRA $FFF9

18 20

Instruction Register:

Program Counter: D704

~~1~~D6FD



D6F9	7 4
D6FA	9 6
D6FB	2 5
D6FC	F 8
D6FD	2 B
D6FE	8 3
D6FF	B A
PC → D700	1 8
PC → D701	2 0
PC → D702	F F
PC → D703	F 9
PC → D704	

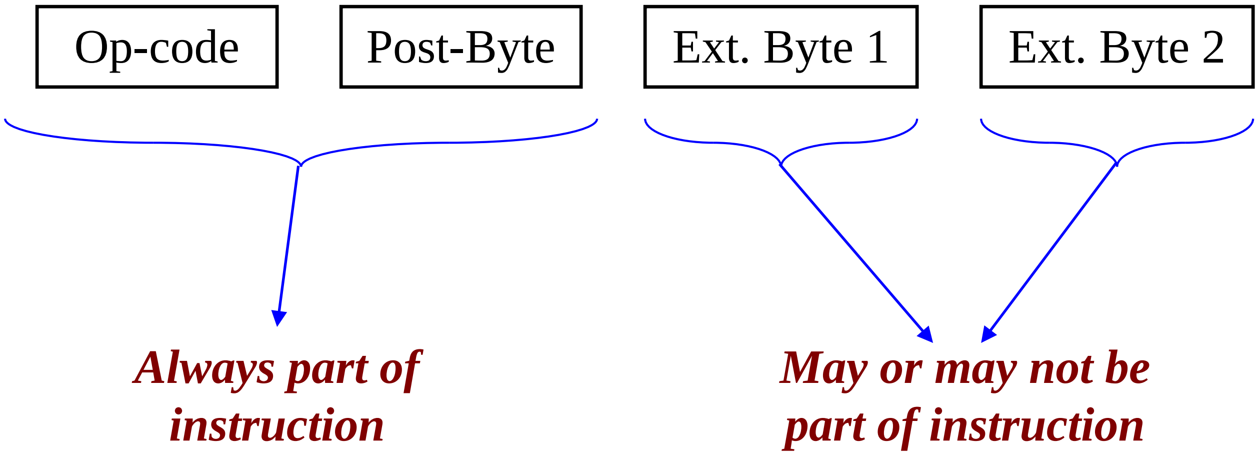
Indexed Addressing Mode:

- There are quite a few variations of indexed addressing scheme.
- The indexed addressing uses *post-byte* plus zero, one, or two extension bytes after the instruction op-code.
- The post-byte and extensions implement the following functions:
  - Specify which indexed register is used.
  - Determine whether a value in an accumulator is used as an offset.
  - Enable automatic pre- or post- increment or decrement.
  - Specify the size of increment or decrement.
  - specify the use of 5-, 9-, or 16-bit signed offset.
- The indexed addressing scheme allows:
  - The stack pointer to be used as an indexed register in all indexed operation.

Indexed Addressing Mode *continued …*

- The program counter to be used as an indexed register in all but autoincrement and autodecrement modes.
- The value in accumulator A, B, or D to be used as an offset.
- Automatic pre- or post- increment or decrement by -8 to +8
- A choice of 5-, 9-, or 16-bit signed constant offsets.

Indexed Addressing Instruction Format:



Op-code

Post-Byte

Ext. Byte 1

Ext. Byte 2

Always part of instruction

May or may not be part of instruction

Indexed Addressing Mode *continued …*

➢ Indexed addressing mode has **10** different variations:

❶ 5-bit offset

❷ Pre-decrement

❸ Pre-increment

❹ Post-decrement

❺ Post-increment

❻ Accumulator offset

❼ 9-bit offset

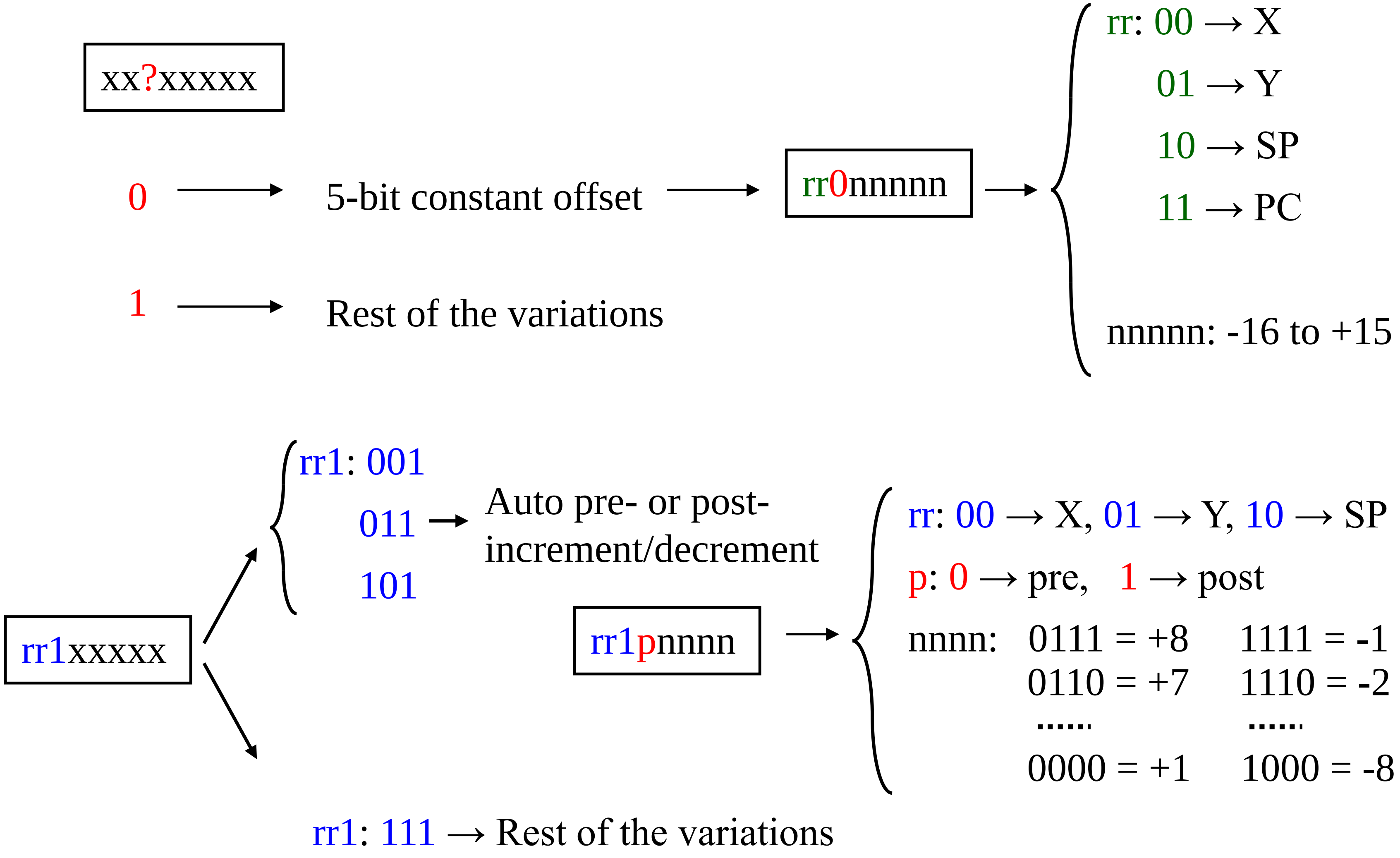
❽ 16-bit offset

❾ Indexed-Indirect 16-bit offset

❿ Indexed-Indirect D acc. offset

Indexed Addressing Mode *continued …*

Post-Byte Format:



xx?xxxxx

0 → 5-bit constant offset → rr0nnnnn →

rr: 00 → X
01 → Y
10 → SP
11 → PC

nnnnn: -16 to +15

1 → Rest of the variations

rr1xxxxx

rr1: 001
011 → Auto pre- or post-increment/decrement
101

rr1pnnnn →

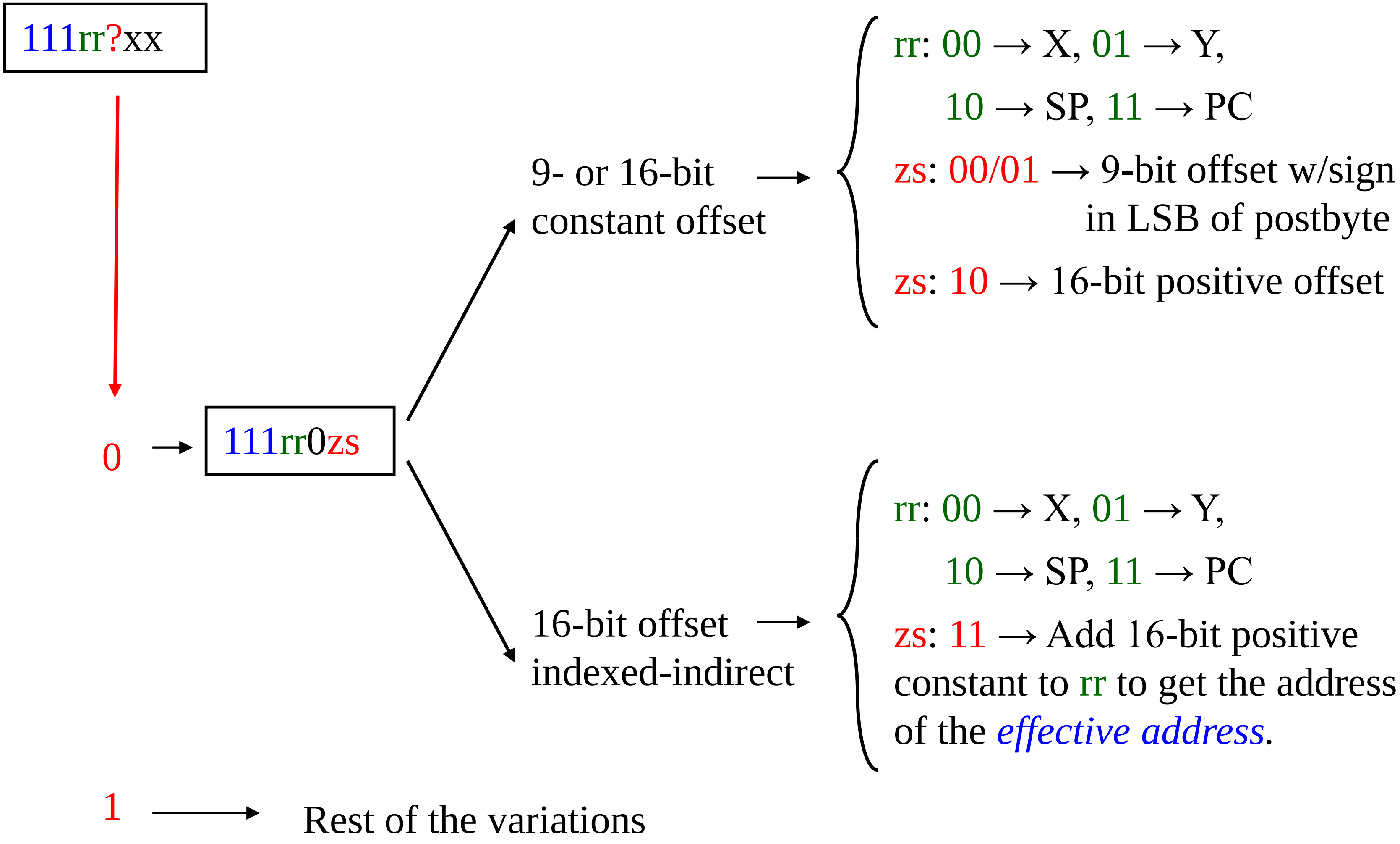
rr: 00 → X, 01 → Y, 10 → SP

p: 0 → pre, 1 → post

nnnn: 0111 = +8 1111 = -1
0110 = +7 1110 = -2
…… ……
0000 = +1 1000 = -8

rr1: 111 → Rest of the variations

Indexed Addressing Mode *continued …*



111rr?xx

0 → 111rr0zs

9- or 16-bit constant offset →

rr: 00 → X, 01 → Y,
10 → SP, 11 → PC

zs: 00/01 → 9-bit offset w/sign in LSB of postbyte

zs: 10 → 16-bit positive offset

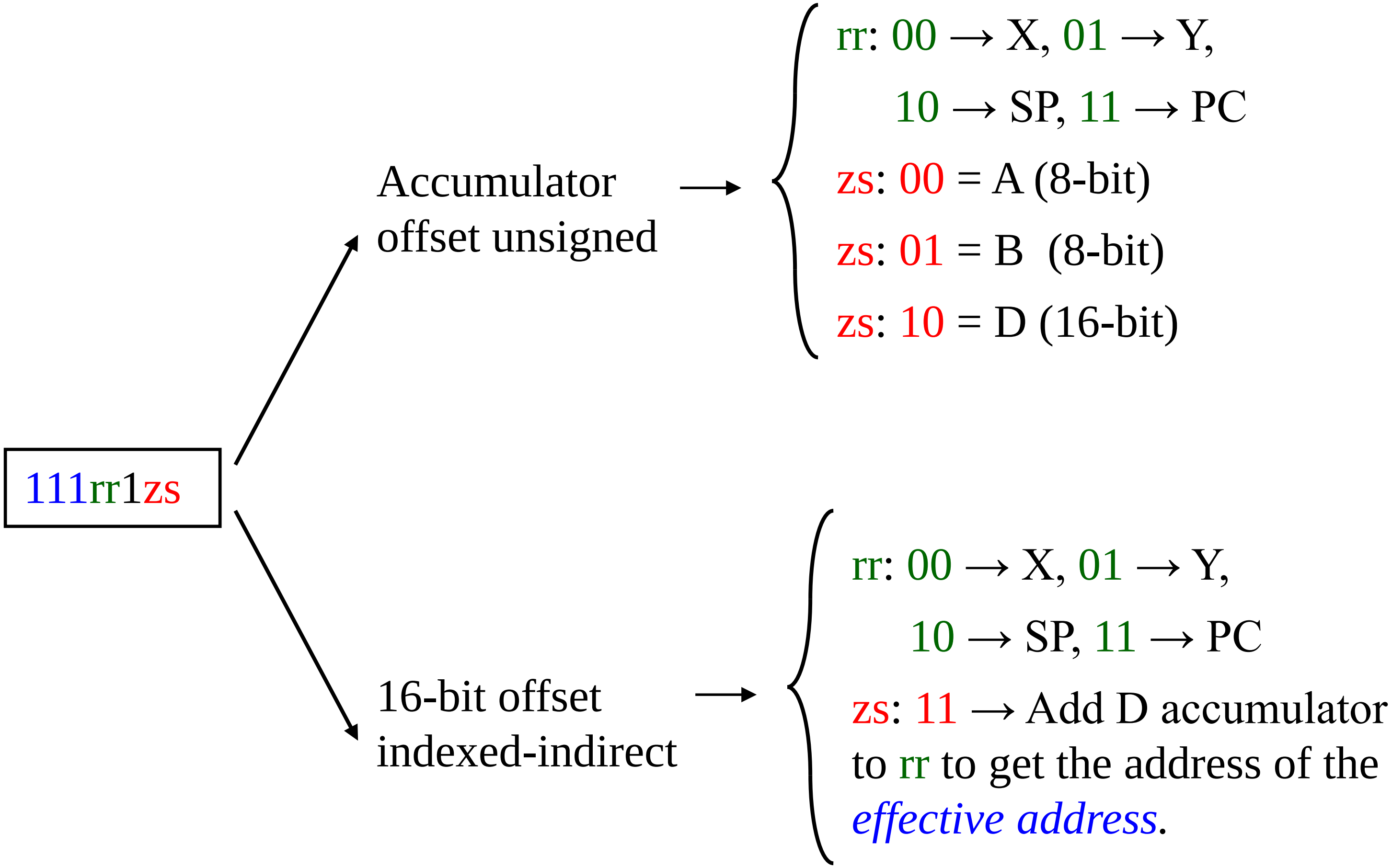
16-bit offset indexed-indirect →

rr: 00 → X, 01 → Y,
10 → SP, 11 → PC

zs: 11 → Add 16-bit positive constant to rr to get the address of the *effective address*.

1 → Rest of the variations

Indexed Addressing Mode *continued …*



111rr1zs

Accumulator offset unsigned →

rr: 00 → X, 01 → Y,
10 → SP, 11 → PC

zs: 00 = A (8-bit)

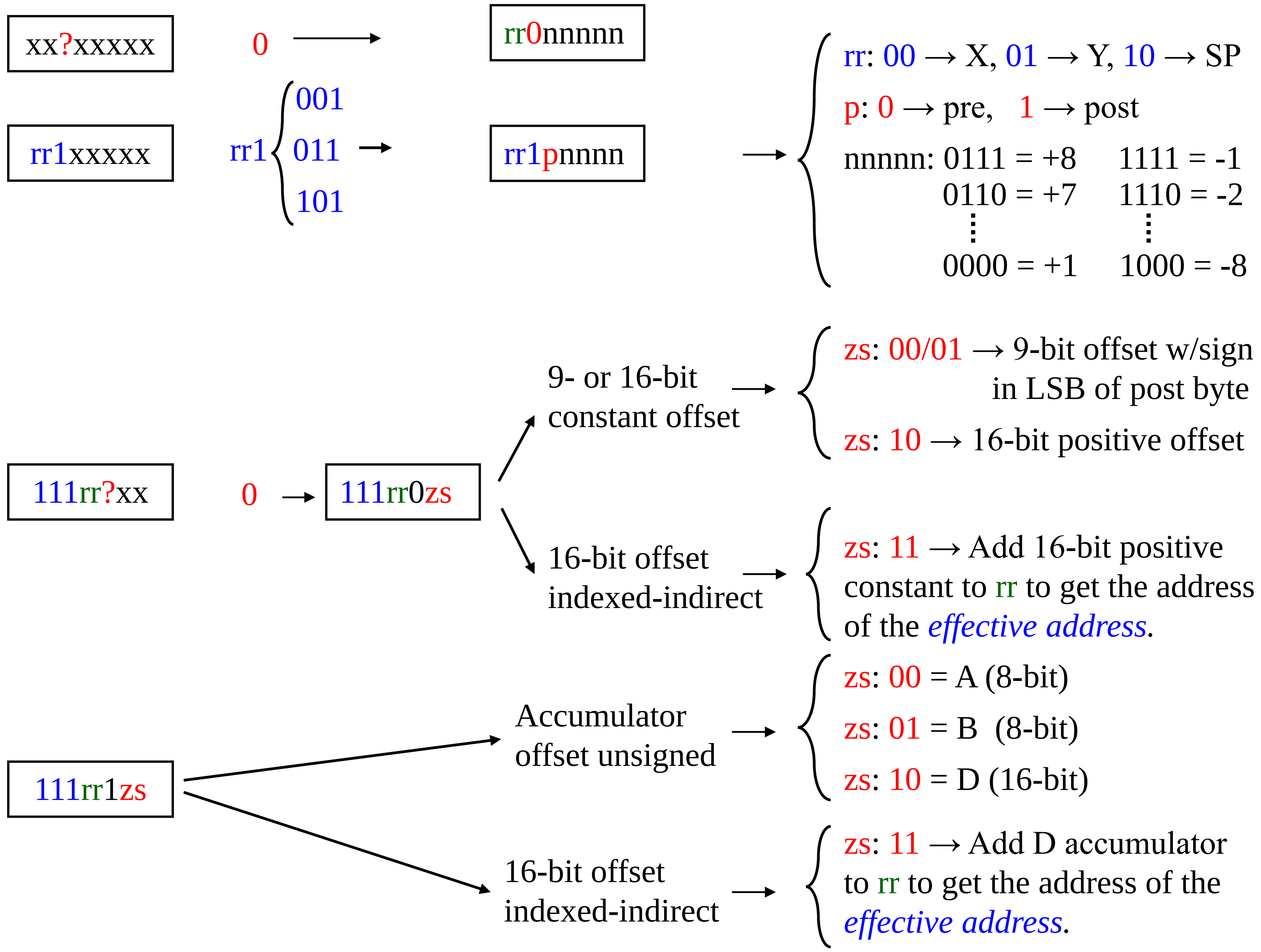
zs: 01 = B (8-bit)

zs: 10 = D (16-bit)

16-bit offset indexed-indirect →

rr: 00 → X, 01 → Y,
10 → SP, 11 → PC

zs: 11 → Add D accumulator to rr to get the address of the *effective address*.



xx?xxxxx 0 → rr0nnnnn

rr1xxxxx rr1 {001, 011, 101} → rr1pnnnn →

rr: 00 → X, 01 → Y, 10 → SP

p: 0 → pre, 1 → post

nnnnn: 0111 = +8 1111 = -1
0110 = +7 1110 = -2
⋮ ⋮
0000 = +1 1000 = -8

111rr?xx 0 → 111rr0zs

9- or 16-bit constant offset →

zs: 00/01 → 9-bit offset w/sign in LSB of post byte

zs: 10 → 16-bit positive offset

16-bit offset indexed-indirect →

zs: 11 → Add 16-bit positive constant to rr to get the address of the *effective address*.

111rr1zs

Accumulator offset unsigned →

zs: 00 = A (8-bit)

zs: 01 = B (8-bit)

zs: 10 = D (16-bit)

16-bit offset indexed-indirect →

zs: 11 → Add D accumulator to rr to get the address of the *effective address*.

Summary of Indexed Operations

Postbyte Code *(xb)*	**Source Code Syntax**	**Comments** rr: 00 = X, 01 = Y, 10 = SP, 11 = PC
rr0nnnnn	**n, r**	**5-bit Constant Offset n = -16 to +15** **r can be X, Y, SP, or PC**
rr1pnnnn	**n, -r** **n, +r** **n, r-** **n, r+**	**Auto Pre-decrement/increment or auto Post-decrement/increment** **p = 0 → (pre-) or p = 1 → (post-)** **r can be X, Y, or SP (PC not a valid choice)** **n = 0111 = +8 n = 1111 = -1** ⋮ ⋮ **n = 0000 = +1 n = 1000 = -8**
111rr0zs	**n, r** **-n, r**	**Constant Offset** **zs = 00 or 01 (9-bit with sign in LS-Bit of postbyte; -256 < n < 255)** **zs = 10 (unsigned 16-bit; 0 ≤ n ≤ 65,535)**
111rr011	**[n, r]**	**16-bit Offset Indexed-Indirect** **r can be X, Y, SP, or PC; 0 < n < 65536**
111rr1aa	**A, r** **B, r** **D, r**	**Accumulator Offset (unsigned 8-bit or 16-bit)** **aa: 00 = A, 01 = B, 10 = D (16-bit)** **r can be X, Y, SP, or PC**
111rr111	**[D, r]**	**Accumulator D Offset Indexed-Indirect** **r can be X, Y, SP, or PC**

5-Bit Constant Offset Indexed Addressing:

- It adds a 5-bit signed offset that is included in the instruction *postbyte* to the base index register to form the effective address.

```
STAB    -8,X              ; MC: 6B 18     rr0nnnnn
LDY     5,SP              ; MC: ED 85     rr0nnnnn
```

9-Bit Constant Offset Indexed Addressing:

- It uses a 9-bit signed offset, which is added to the base index register (X, Y, SP, or PC) to form the effective address of the memory location affected by the instruction.

```
LDD     160,Y             ; MC: EC E8 A0     111rr00s
JMP     -40,X             ; MC: 05 E1 D8     111rr00s
```

16-Bit Constant Offset Indexed Addressing:

- This indexed addressing mode specifies a 16-bit offset to be added to the base index register (X, Y, SP, or PC) to form the effective address of the memory location affected by the instruction.
- This allows access to any location in the 64 kB address space.

```
INC     $2000,X           ; MC: 62 E2 20 00     111rr010
```

16-Bit Constant Indirect Indexed Addressing:

- This indexed addressing mode adds a 16-bit offset to the base index register to form the address of a memory location that contains a pointer to the memory location affected by the instruction (*address of address*).

```
LDAA   [$1000,Y]          ; MC: A6 EB 10 00     111rr011
```

if [Y] = $1A00, then Effective address = [$2A00]

```
CPD     [128,X]           ; MC: AC E3 00 80     111rr011
```

if [X] = $1000, then Effective address = [$1080]

Auto Pre/Post-Decrement/Increment Indexed Addressing:

- This indexed addressing mode provides four ways to automatically change the value in a base index register as a part of instruction execution.

- The base index register may be X, Y, or SP.

```
STAB   2,X+    ; MC: 6B 31              rr1pnnnn
DEC    1,-Y    ; MC: 63 6F              rr1pnnnn
```

Accumulator Offset Indexed Addressing:

- In this indexed addressing mode, the effective address is the sum of the values in the base index register and an unsigned offset in one of the accumulators.

- The value in the base index register itself does not change and accumulator can be A, B, or D.

```
STY    B,X     ; MC: 6D E5              111rr1aa
LDAA   A,Y     ; MC: A6 EC              111rr1aa
```

Accumulator D Indirect Indexed Addressing:

- This indexed addressing mode adds the value in accumulator D to the value in the base index register to form the address of the memory location affected by the instruction (*address of address*).

```
JMP     [D,PC]          ; MC: 05 FF          111rr111
```

Addressing more than 64 KB:

- The HCS12 devices incorporate hardware that supports addressing ae larger memory space than the standard 64 KB.
- The expanded memory system is accessed by using the bank-switching scheme.
- The HCS12 treats the 16 KB of memory space from $8000 to $BFFF as a program memory window.
- The HCS12 has an 8-bit program page register (PPAGE), which allows up to 256 16-KB program memory pages to be switched into and out of the program memory window.
- This provides up to 4 MB of paged program memory.

A Few Examples

LDX #$1500
MC: **CE 15 00**

Reg. X : **15 00**

LDY 2,X+
MC: **ED 31**

Reg. X : **15 02**
Reg. Y : **2A 00**

LDAA 34,Y
MC: **A6 E8 22**

Reg. Y : **2A 00**
Acc. A : **10**

LDAB $23,Y
MC: **E6 E8 23**

Reg. Y : **2A 00**
Acc. B : **02**

LDD [$1516,X]
MC: **EC E3 15 16**

Add. Of Add.: 2A 18

EA: 10 17

Reg. X : **15 02**
Acc. D : **BC EF**

Partial Memory Dump

```
$1000  2A    $1500  2A    $2A0F  25
$1001  56    $1501  00    $2A10  58
$1002  CD    $1502  03    $2A11  AD
$1003  9E    $1503  27    $2A12  4E
$1004  33    $1504  20    $2A13  F4
$1005  00    $1505  05    $2A14  1A
$1006  2A    $1506  0B    $2A15  94
$1007  15    $1507  39    $2A16  F0
$1008  15    $1508  81    $2A17  77
$1009  0F    $1509  A9    $2A18  10
$100A  2A    $150A  3F    $2A19  17
$100B  22    $150B  55    $2A1A  2A
$100C  67    $150C  97    $2A1B  27
$100D  B2    $150D  AB    $2A1C  75
$100E  0C    $150E  6C    $2A1D  52
$100F  8A    $150F  62    $2A1E  64
$1010  B0    $1510  19    $2A1F  0C
$1011  CC    $1511  99    $2A20  35
$1012  15    $1512  B5    $2A21  AA
$1013  0D    $1513  4C    $2A22  10
$1014  58    $1514  DD    $2A23  02
$1015  BF    $1515  0F    $2A24  2B
$1016  76    $1516  17    $2A25  1A
$1017  BC    $1517  88    $2A26  61
$1018  EF    $1518  EE    $2A27  00
```