uOttawa

L'Université canadienne
Canada's university

CEG3136
Computer Architecture II

Module 2 – CPU Registers, Condition Codes, and Addressing Modes

Notes for
Dr. Voicu Groza

Université d'Ottawa | University of Ottawa

www.uOttawa.ca

Topics of discussion

- **Introduction to the CPU**
  - CPU Registers
  - MC68HC12 Programmer's Model
  - CPU Condition Codes
- Memory Addressing
  - Memory Architecture
  - Addressing Modes
- Reading Assignment: Sections 4.1-4.2, 4.5-4.6

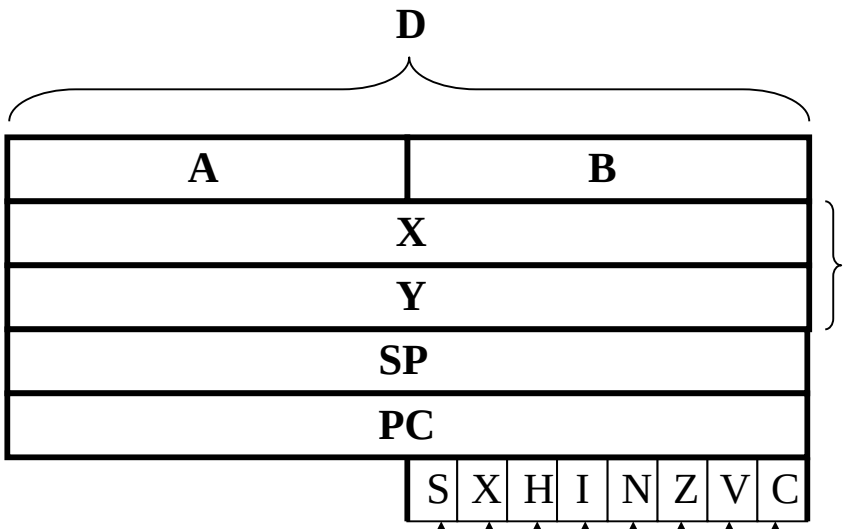
CPU Registers

- Accumulators: Accumulate answers
  - Can also serve as source registers for data operands
- General-purpose registers: Hold data
  - Source and destination operands for data transfer instructions,
  - Source for ALU operations
- Doubled registers: double size of registers
  - Combine 2 8-bit registers to form 16 bit register
- Pointer Registers: address memory
  - Used to point to a memory address location

CPU Registers (continued)

- Stack Pointer Register: dedicated to addressing memory
  - Used to implement subroutines
- Index registers: Address memory
  - Add an offset value to the register contents to generate the *effective* address
- Segment registers: Address memory
  - Addresses segments of memory (an offset is then used to address the memory location in the segment)
- Condition Code Register: Contains flags and status
  - Holds condition code bits generated by the processor during execution of instructions
  - Some bits are used for control – ex: interrupt mask

CPU12 Programmer's Model



Accumulators

Index registers

Stack Pointer

Program Counter

Condition Code Register (CCR)

Carry =1 if unsigned carry/borrow out of/into msb

o**V**erflow in 2's complement

Zero = 1 if every bit of ALU result is 0

Negative flag = bit 7 after ALU operation

Interrupt = global interrupt mask

Half-carry flag (BCD arithmetic) = 1 if carry out of bit 3->4

X = pseudo-nonmaskable interrupt mask

Stop Disable = 1 => STOP = NOP, else STOP clocks

CPU 12 Programmer's Model

- Accumulators A, B, D
  - Two 8-bit accumulators
  - D is the concatenation of A and B to form 16-bit accumulator
    - Accumulator A is most significant byte in D
- Index Registers X and Y
  - Primarily for indexed addressing
  - Are also used in some arithmetic operations
- Stack Pointer
  - Points to the bottom of the memory stack
  - Need initialisation before use
- Program Counter
  - For addressing instructions in memory
- Condition Code Register
  - Status flags

Data types used by MC68HC12

- Bit
- 5-bit signed integers (used as offset for indexed addressing)
- 8-bit signed and unsigned integers
- 9-bit signed integers (used as offset for indexed addressing)
- 16-bit signed and unsigned integers
- 16-bit effective addresses
- 32-bit signed and unsigned integers (used for extended-multiply, extended division and extended multiply-and-accumulate instructions)

The Condition Control Register

M68HC12 condition code register bits

Bits modified by various instructions

Bit	Flag	Conditions for setting
0	C	If a carry or borrow occurs.
1	V	If two's-complement overflow occurs.
2	Z	If the result is zero.
3	N	If the most significant bit of the result is set.
5	H	This is the half-carry bit and is set if a carry or borrow out of bit three of the result occurs.

Bits associated with M68HC12 control

Bit	Flag	Use
4	I	Interrupt mask.
6	X	X interrupt mask.
7	S	Stop disable.

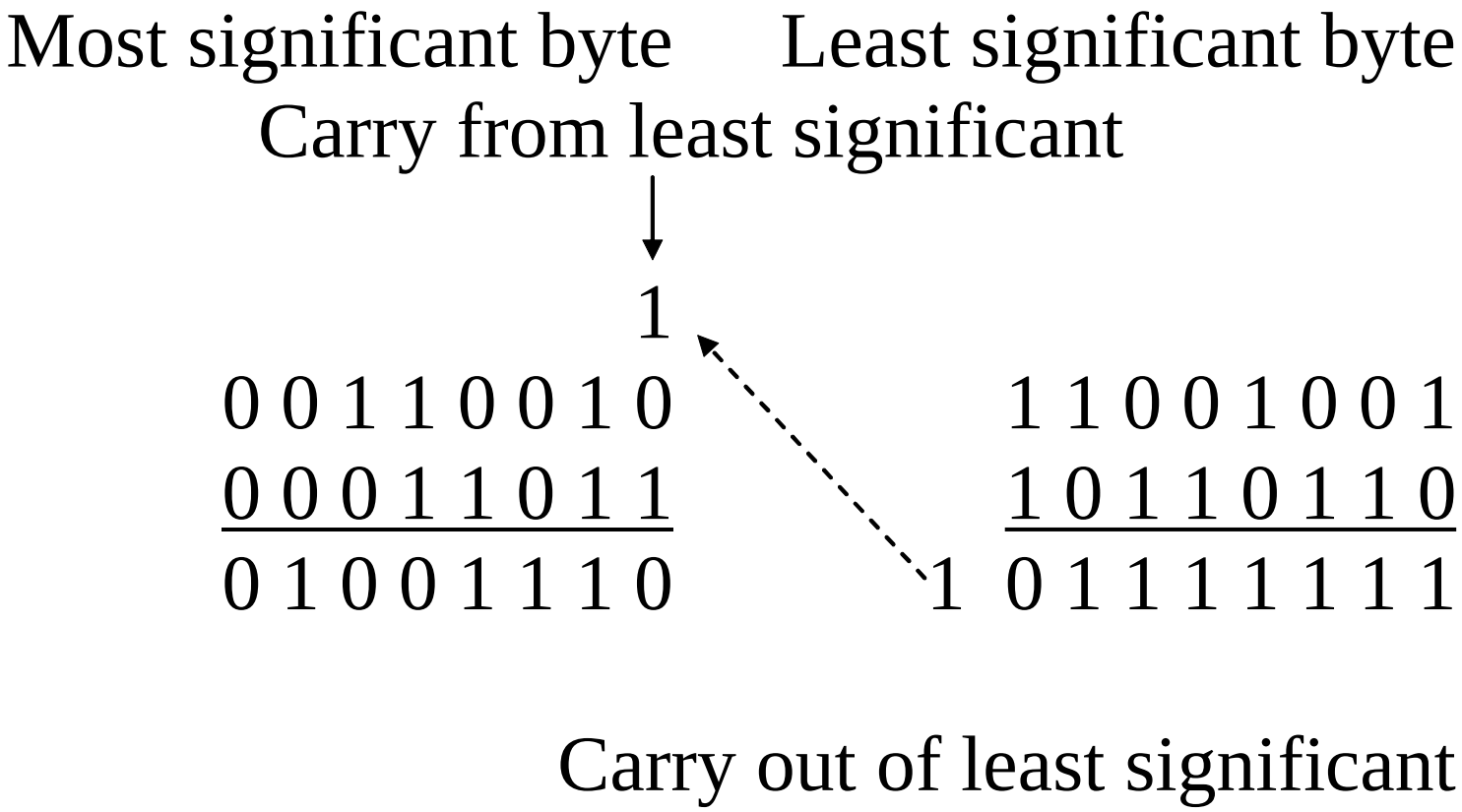
The Carry Bit (C)

- Definition: Overflow
  - The result of addition or subtraction is too large or too small to be represented by number of bits available.
- Carry bit set when overflow occurs during addition/subtraction of unsigned numbers
  - Sometimes referred to as the carry/borrow bit
  - Set (=1) when an *overflow* occurs: an addition produces a carry
    - a subtraction requires a borrow (also called underflow)
  - Can be used for multiple-byte addition or subtraction

Addition or subtraction of unsigned numbers

Addition			Subtraction		
147	1 0 0 1 0 0 1 1		147	1 0 0 1 0 0 1 1	
+179	+ 1 0 1 1 0 0 1 1		-179	- 1 0 1 1 0 0 1 1	
326	1 0 1 0 0 0 1 1 0		- 32	1 1 1 1 0 0 0 0 0	
	↑			↑	
	Carry			Borrow	

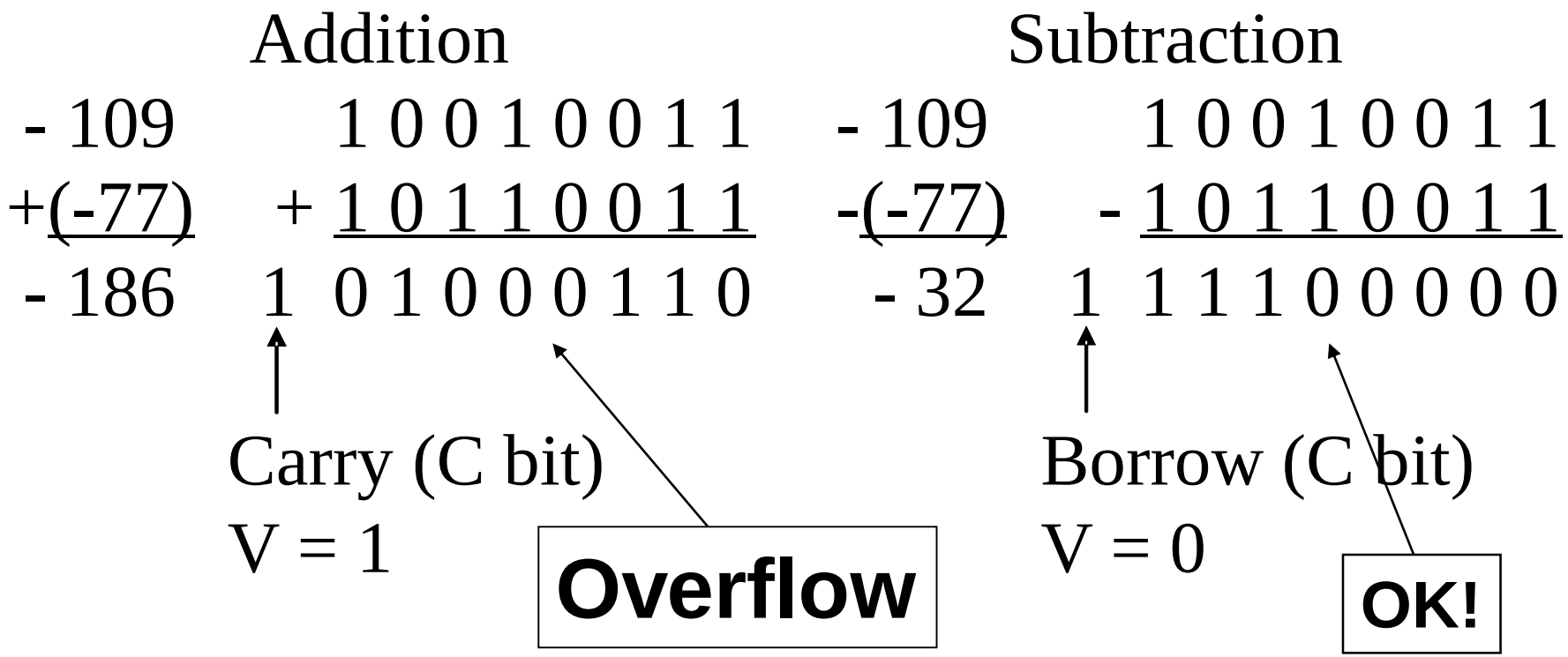
Multi-byte Addition



Two's complement Overflow Bit (V)

- Will the carry bit detect an overflow when numbers are treated as 2's complement code (i.e. *2's complement representation* of signed numbers)?
- Two's complement overflow bit required
  - Algorithm: Two's complement overflow occurs when:
    - Two operands are of the same sign AND
    - The sign of the result is different
    - Two's complement overflow cannot occur when operands are of different signs
  - For subtraction, consult the SUBA instruction in the CPU12 Reference Manual.
- Note that the same hardware (and thus computer instructions) is used to add both signed and unsigned numbers

Addition and subtraction of two's complement numbers (signed)



Sign, Zero, and Half-carry Bits

- Sign Bit (N)
  - Set to the most significant bit of the results
  - Indicates sign only if result treated as a signed number (two's complement)
- Zero bit (Z)
  - Set (=1) when the result of the operation is zero
- Half-carry bit (H)
  - Set (=1) when carry out of bit 3 occurs
  - Only used by the DAA (Decimal adjust) instruction

Example

- Add the following numbers, give the results and show how the operations affect the C, V, S, and Z bits.

1 0 1 0 1 1 0 1	1 0 1 0 1 1 0 1	1 0 1 0 1 1 0 1
1 0 1 1 0 0 1 0	0 1 0 0 1 1 0 1	0 1 0 1 0 0 1 1

Bits Associated with Control

- I Interrupt Mask (I)
  - Used to globally mask (disable) and unmask (enable) interrupts
- X Interrupt Mask (X)
  - Used to mask (disable) and unmask (enable) for the XIRQ interrupt input
- Stop Disable (S)
  - Allows or disallows the STOP instruction

Using the CCR (Condition Control Register) bits

- CCR bits set or reset by the execution of certain instructions
- Conditional branch instructions can then use the bits to react to the results
  - Consider a BCS (branch if carry set)
  - Consider program that rings a bell when an overflow occurs for an unsigned addition
    - Load A accumulator with first byte (LDAA)
    - Load B accumulator with the second byte (LDAB)
    - Add byte in B to byte in A and store results in A (ABA) (Carry bit is set if an overflow occurs)
    - Branch if carry bit is set to section of the program that rings the bell (BCS)
    - Otherwise continue

Topics of discussion

- **Introduction to the CPU**
  - CPU Registers
  - MC68HC12 Programmer's Model
  - CPU Condition Codes
- Memory Addressing
  - Memory Architecture
  - Addressing Modes

Memory Terminology

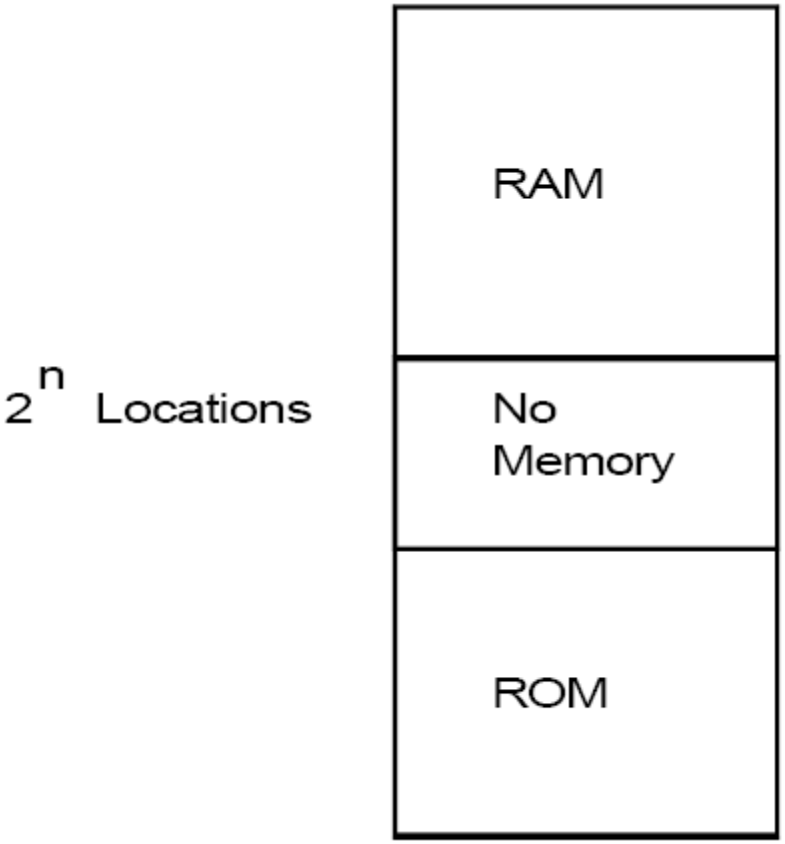
- Physical Address: address applied to physical memory
- Segment Address: Location of block or segment of address (smaller than full physical memory)
- Offset (or Relative) Address: Calculated from the start of segment or block
- Logical Address: logical address used by program mapped to the physical address
  - For example:
    - Segment number + offset = logical address
    - Segment address added to offset gives physical address
    - Segment number used to get address of segment from table

Memory Terminology (continued)

- Effective address: address calculated by the processor
  - May be physical or logical address
- Auto-increment and auto-decrement
  - Registers (for example index registers) may be incremented/decremented automatically
- RAM: Random access memory - volatile
- ROM: Read only memory - permanent
- EEPROM (Flash Memory): permanent memory that is erasable.
- Memory and I/O **maps**: show how all memory addresses are used
  - Region may contain RAM
  - Other region contains ROM
  - Some regions contain nothing

Memory Architecture – Linear addressing

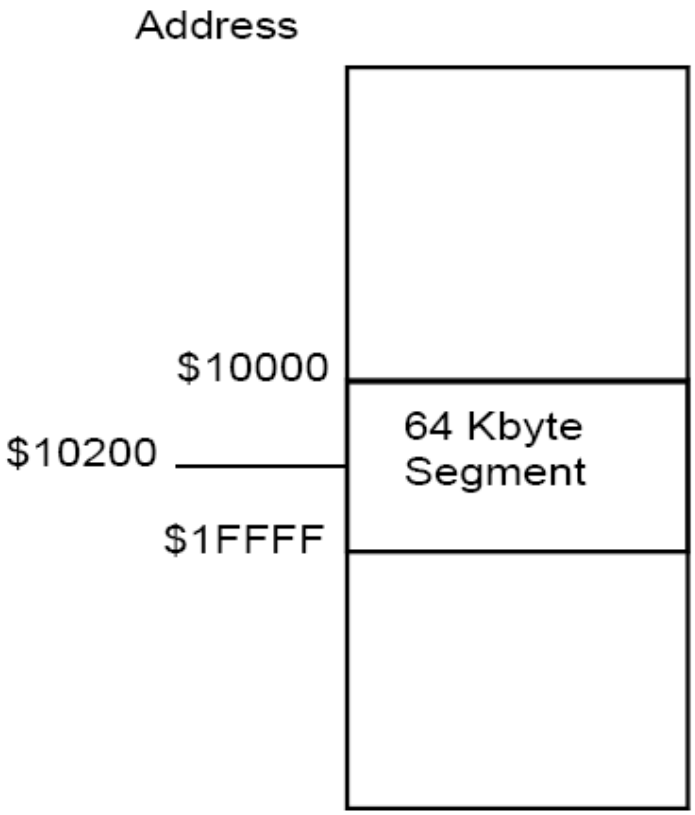
- Instructions specify full physical address
  - Motorola microprocessors use linear addressing
- MC68HC12 uses 16-bit addresses
  - Address 64 Kbyte address space



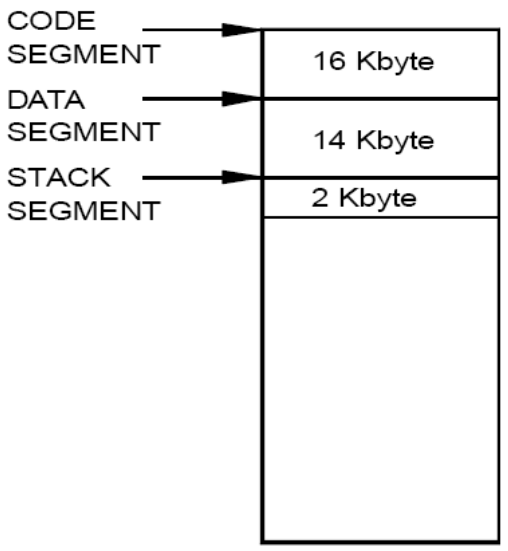
Segmented Addressing

- Address memory using segment and offset
  - Segment address stored in register
  - Offset specified in instruction
    - Reduces size of address in instruction
- Intel 80xxx microprocessors used segmented addressing
- Consider addressing in the 8086
  - 16-bit segment registers
  - 16-bit offsets (segments occupy up to 64 Kbytes)
  - Shift address in segment register by 4 bits and add offset
  - Segments can start at any 16-byte boundary
  - Results in 20-bit physical address, i.e. 1 Mbyte memory address space

Segmented Addressing Example



$	1	0	0	0	Segment Register
$	1	0	0	0	0 Segment shifted left 4 bits
$	0	2	0	0	Offset Address
$	1	0	2	0	0 Physical Address = Segment*16 + Offset

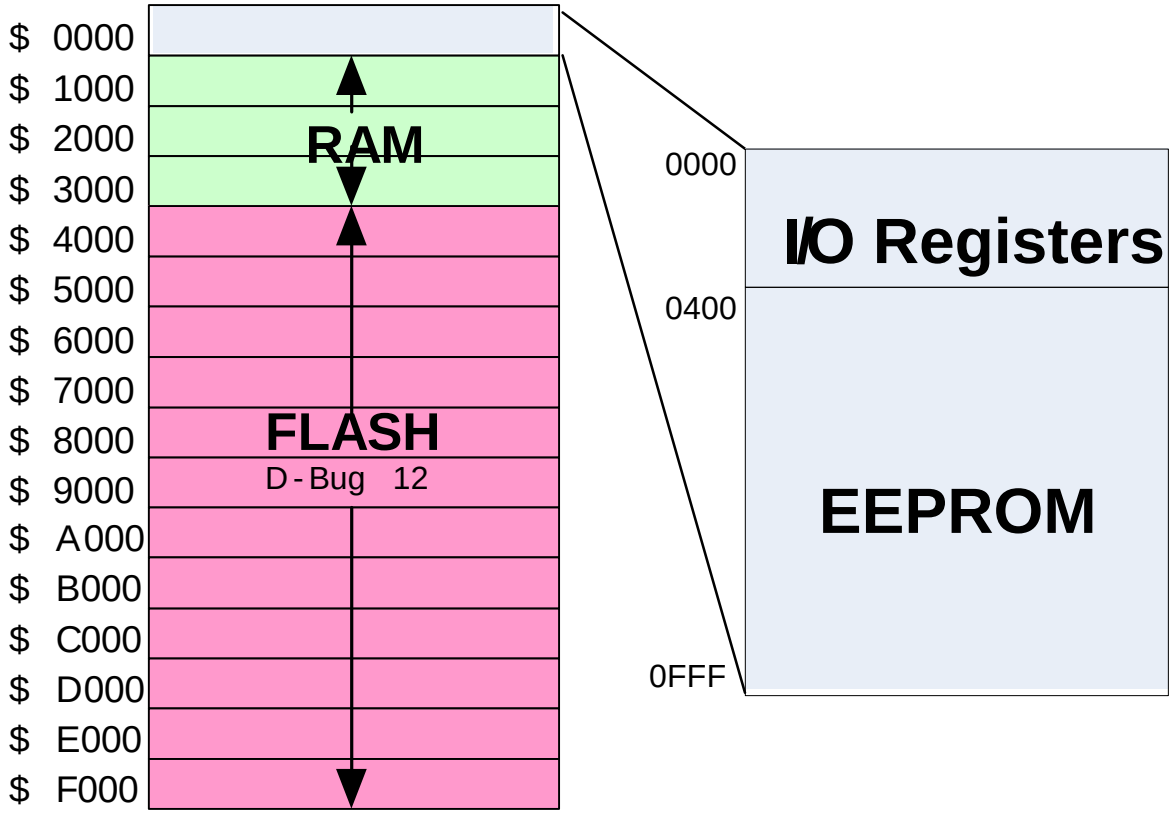


MC9S12DG256 Paging

- Although a 16 bit address is used within the HCS12 MCU, it is possible to expand memory using the Program Page Index Register (PPAGE)
  - 6 bits are used in PPAGE used to access up to 64 – 16Kbyte pages of external memory
  - The pages are accessed at addresses $8000-$BFFF within the 64kByte memory map of the MCU
  - Use of PPAGE is restricted to certain instructions, see CALL and RTC
  - This allows expansion to an additional 1Mbyte memory
  - Similar to segmentation

MC9S12DG256 Memory Map

- Motorola products use linear addressing
- In single chip mode (no external addressing), addresses:



- On-chip registers to I/O interfaces ($0000 to $03FF)
- 3 KBytes EEPROM ($0400 to $0FFF)
- 12 KByte RAM ($1000 to $3FFF)
  - $3C00 to $3FFF is used by the D-Bug12 Monitor
- 48 KByte EEPROM Memory ($4000 to $FFFF)

Addressing Modes

- Inherent (CPU Register) Addressing
  - Operands reference internal registers
  - Also called *inherent addressing* (Motorola)
- Immediate Addressing
  - Operand is constant and *immediately* follows the opcode
- Direct Addressing
  - The Operand that follows the opcode contains the memory address
- Indexed addressing
  - Finds the data address using an index (an offset)
- Indirect Addressing
  - Instruction specifies where the address of the data is located
- Relative Addressing
  - Add offset to current value of the PC
- Stack Addressing
  - Addressing memory reserved for temporary storage used on a last-in first-out basis. (the stack).

Notation used in Comments

- Register Name: Indicates a register and its contents
  - Example: A refers to the contents of accumulator A
- → (->) Right arrow indicates data transfer operation (<-> indicates an exchange of data)
  - Example: A -> B indicates the contents of A are transferred (copied) to B
- (…) Contents of a memory location
  - Example: ($1234) -> B indicates that the contents of memory location $1234 are transferred to B
- ((…)) Indirect addressing – inner parentheses specifies a memory address that contains the data address
  - Example: A ->(($1234)) indicates that the contents of A are transferred to a memory location whose address is found in $1234:$1235.

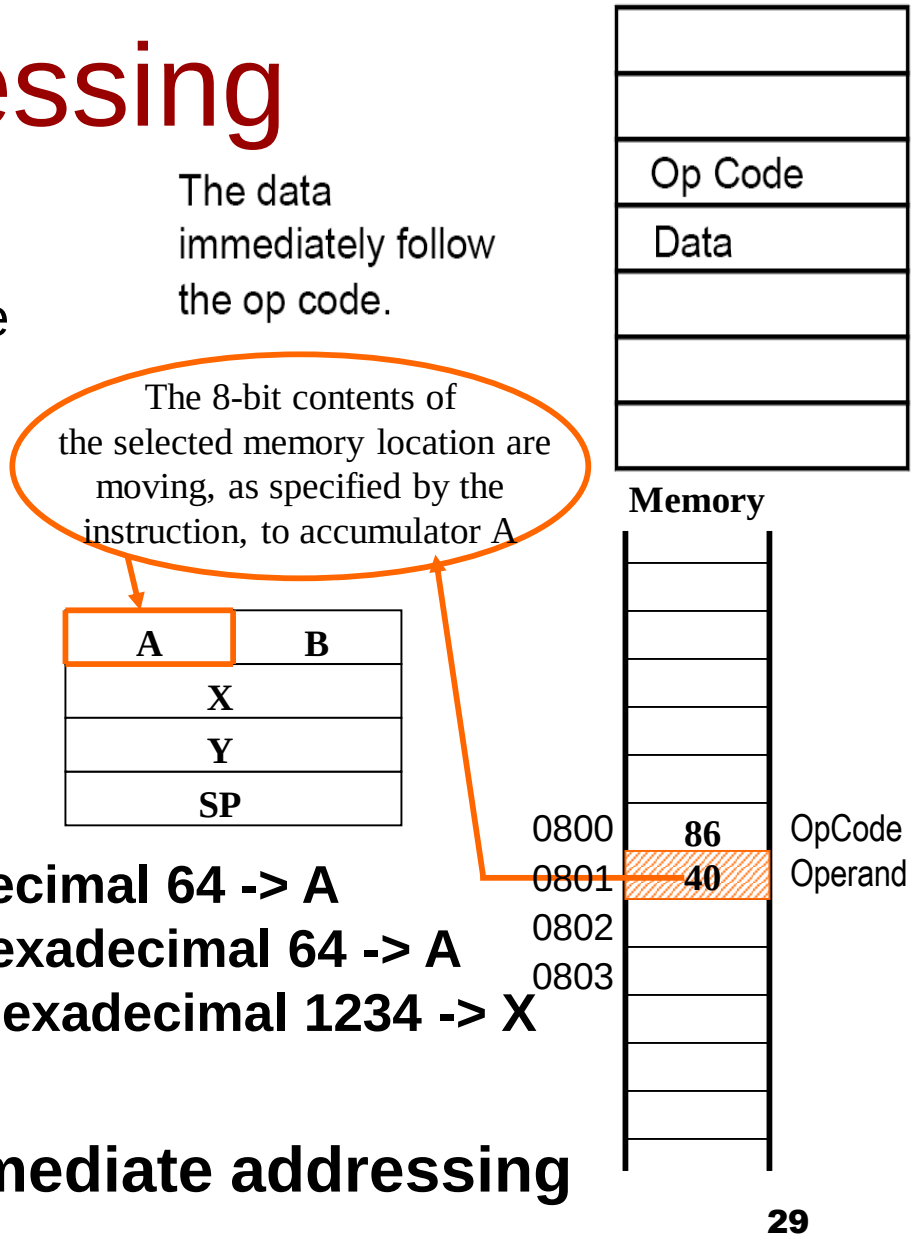
Inherent Addressing (Register Addressing)

- All data for instructions are in the CPU
- Instructions with inherent addressing
  - Are among the fastest to execute
  - Coded with the least amount of bits
- Also called **Register Addressing** or **Implied Addressing**

MACHINE CODE		ASSEMBLY LANGUAGE			
Memory Addr	Memory content				
0000	**1806**	**1**	**aba**		**; A+B -> A**
0002	**08**	**2**	**inx**		**; X+1 -> X**
0003	**B781**	**3**	**exg**	**a,b**	**; A <-> B**

Immediate Addressing

An operand is contained in the byte(s) immediately following the opcode. The number of bytes following the opcode matches the size of the register or memory location being operated on. The effective address is the address of the byte following the instruction.



Immediate Addressing Examples

```
0800 8640      1   ldaa   #64      ; Decimal 64 -> A
0802 8664      2   ldaa   #$64     ; Hexadecimal 64 -> A
0804 CE1234    3   ldx    #$1234   ; Hexadecimal 1234 -> X
```

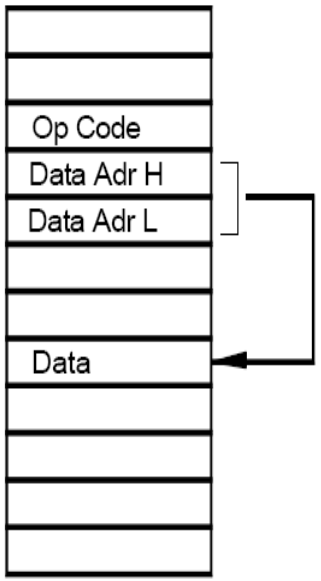
- **The # symbol indicates immediate addressing**
- **It is easy to forget the #**

Direct Memory Addressing

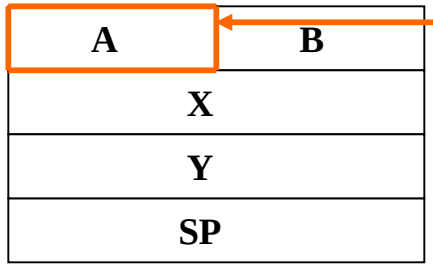
- Instruction contains the data **address**
  - Single-level addressing

The low-order byte of the operand address is contained in a single byte following the opcode, and the high-order byte of the address is assumed to be $00. Addresses $00–$FF are thus accessed directly, using 2-byte instructions. In many microprocessors, this 256-byte area is reserved for frequently referenced data; the following example refers to an HC12 I/O device.

The data address is in the two bytes following the op code.

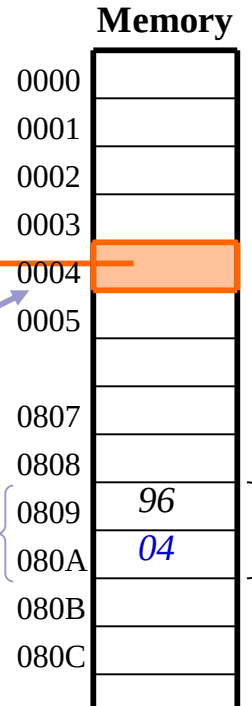


Example: `ldaa 4 ;($0004)-> A`



#2
The 8-bit contents of the selected memory location are moving, as specified by the instruction, to the accumulator A.

#1
The 8-bit contents *(04)* of the last byte of the instruction represents the address of this memory location



CPU12 Direct Memory Addressing

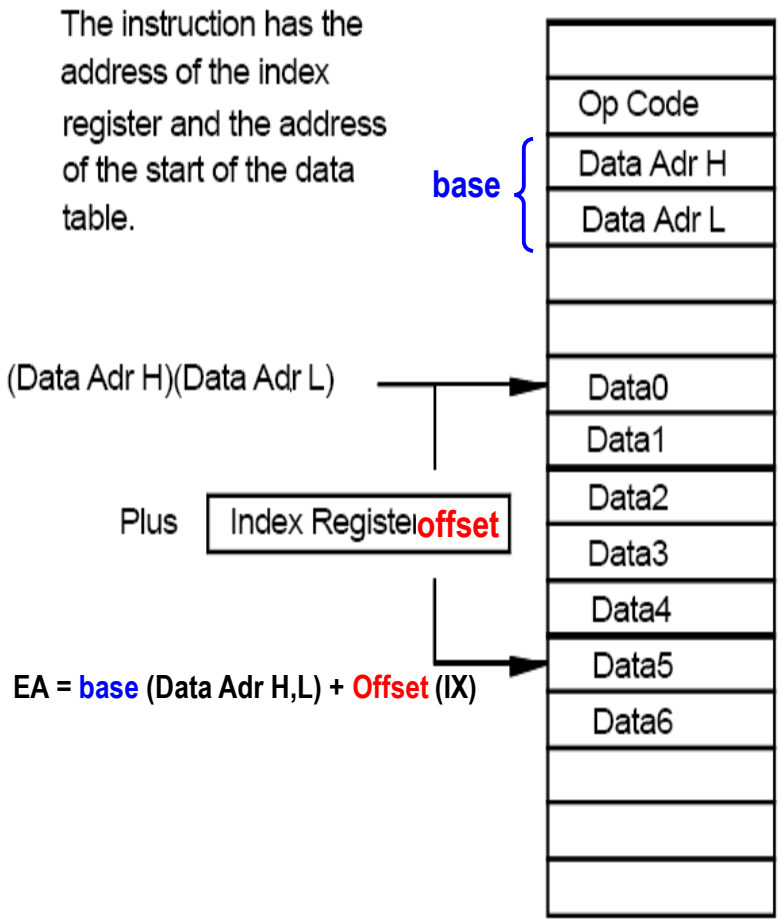
- CPU12 defines two modes of direct memory addressing
  - Direct Addressing – uses 8-bit memory address ($0000 to $00FF, i.e. 0 to 255)
  - Extended Addressing – uses 16-bit address to address all 64 K addresses

```
; Direct Addressing
0800 9664      1   ldaa   100     ; ($0064) -> A, i.e., PORTAD -> A
0802 5BFF      2   stab   $FF     ; B -> ($00FF)
0804 DE0A      3   ldx    10      ; ($000A:000B) -> X
; Extended Addressing
0806 B61234    1   ldaa   $1234   ; ($1234) -> A
0809 FC1234    2   ldd    $1234   ; ($1234:1235) -> D
080C 7EC000    3   stx    $c000   ; X -> ($C000:C001)
```

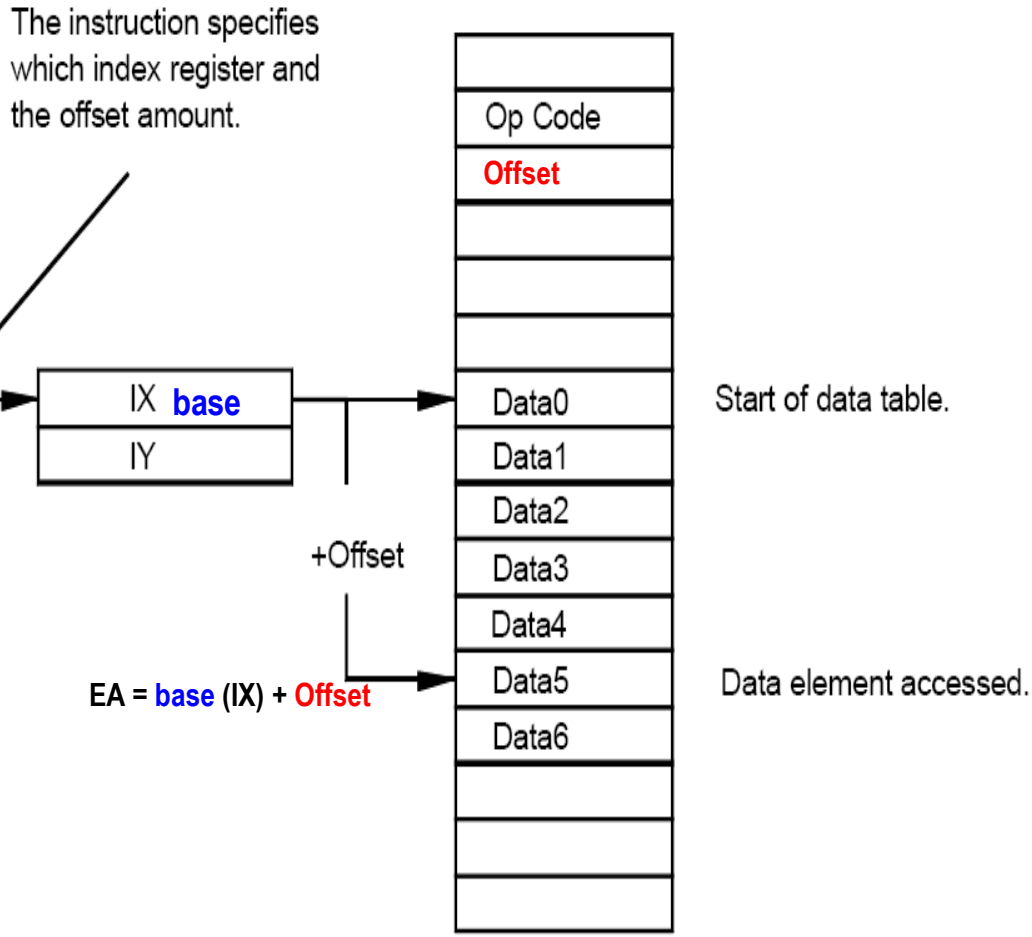
Indexed addressing

- Calculates an *effective address* by summing a starting address (**base address)** to an **offset (index)**

Indexed addressing
Index Register = **offset**

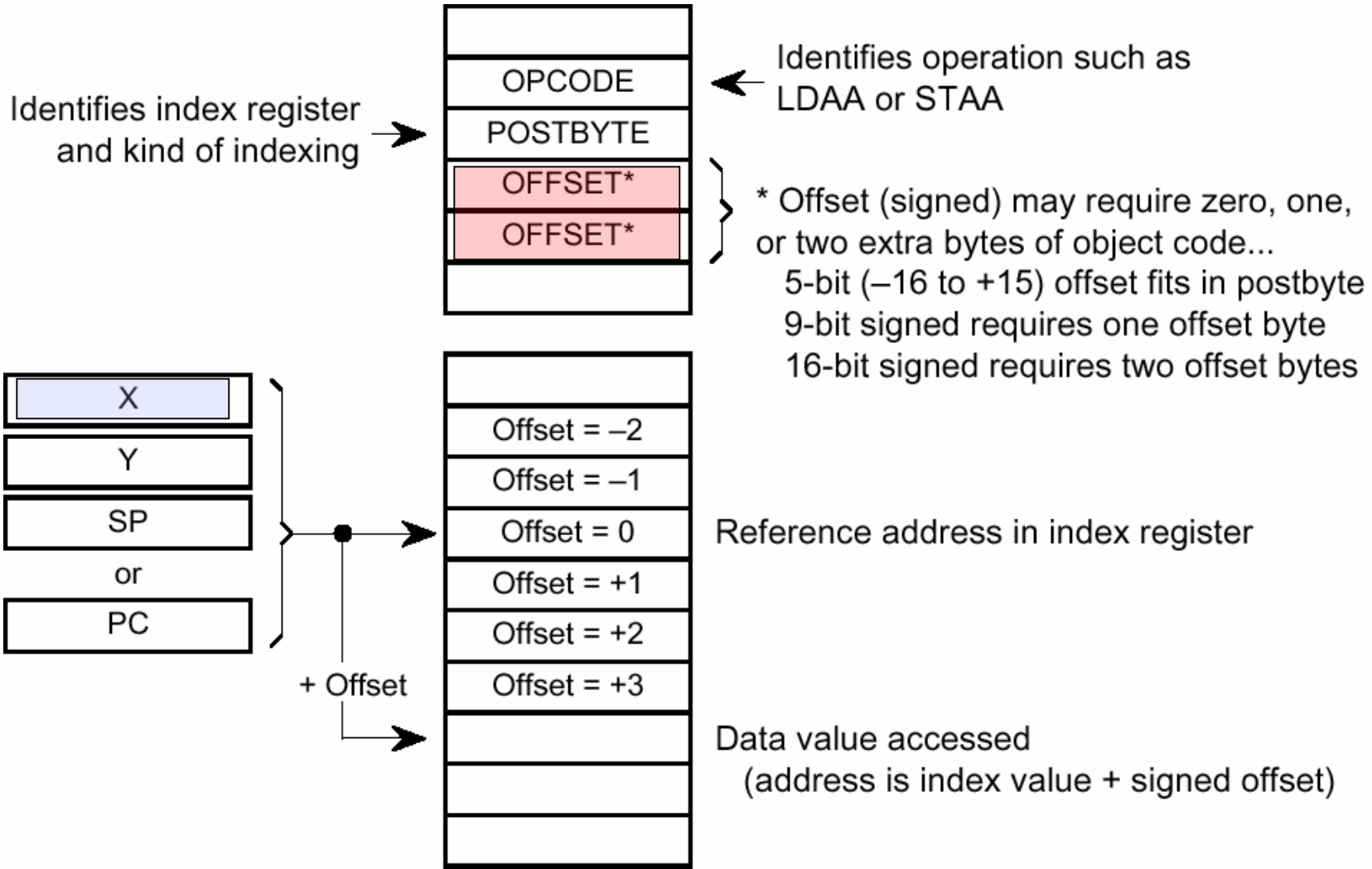


Based addressing
Index Register = **base**



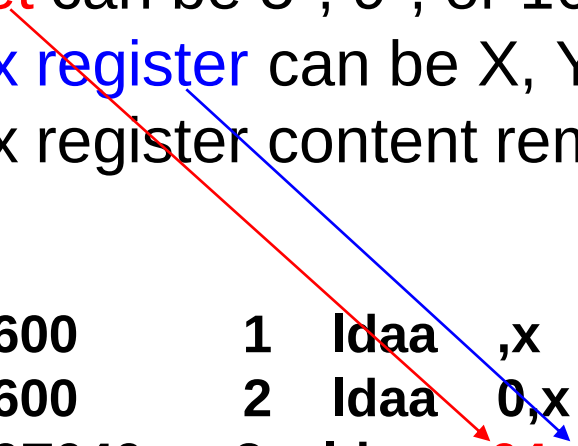
HCS12 Indexed Addressing =

(called Based Addressing in other systems)



HCS12 Indexed Addressing **Using Constant Offset**

- Effective address = offset + contents of index register
  - Offset can be 5-, 9-, or 16-bit offset
  - Index register can be X, Y, SP, or PC **only**! (not memory location!)
  - Index register content remains unchanged!



```
0800 A600       1   ldaa   ,x          ; (X+0) 5-bit offset -> A
0802 A600       2   ldaa   0,x         ; (X+0) 5-bit offset -> A
0804 A6E040     3   ldaa   64,x        ; (X+64) 9-bit offset -> A
0807 A6E9C0     4   ldaa   -64,y       ; (Y-64) 9-bit offset -> A
080A 6A9F       5   staa   -1,SP       ; A -> (SP-1) 5-bit offset
080C A6FA1388   6   ldaa   5000,PC     ; (PC+5000) 16-bit offset -> A
```

Indexed Addressing with Automatic **Incrementing** and **Decrementing**

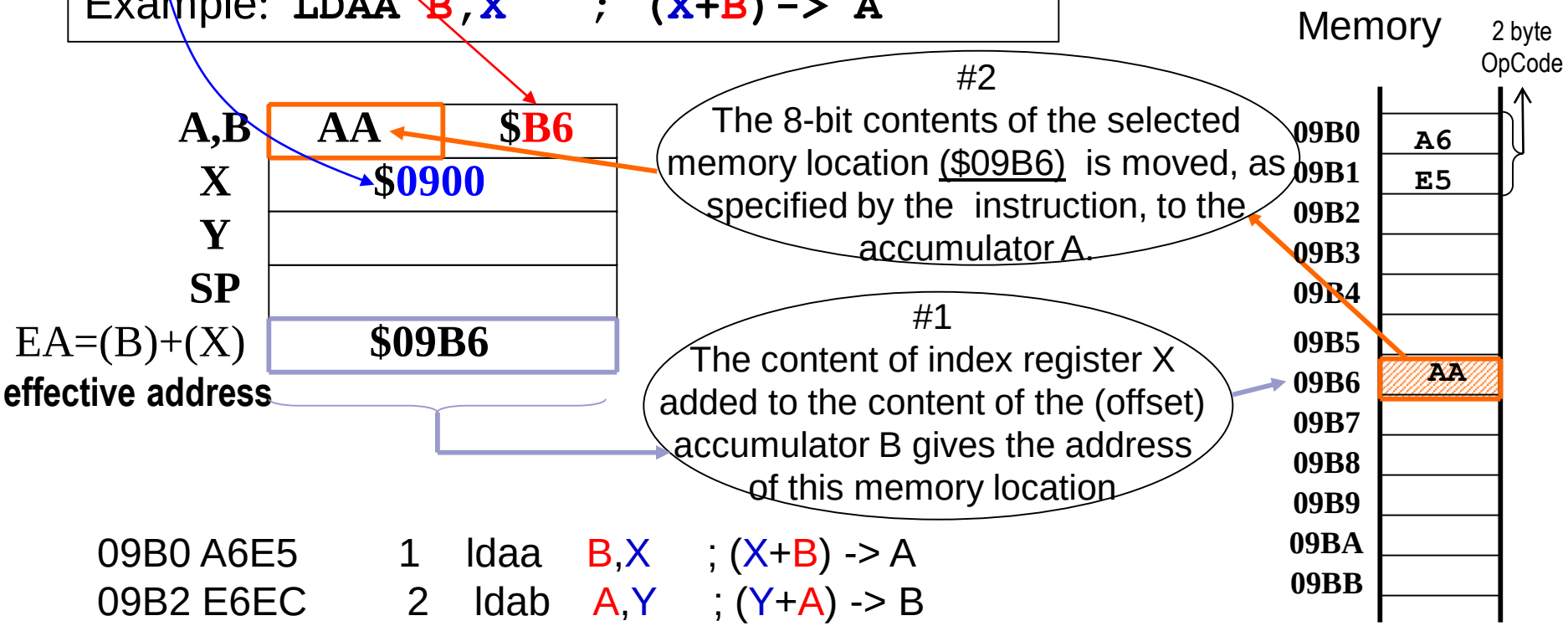
- Use + and – in instruction to include automatic incrementing and decrementing of index register
  - Position of +/- indicates to modify index register before (pre) or after (post) the data value was accessed
  - **Offset value (-8,+8)** is added/subtracted to/from index register (*index register is **changed***)

```
                    1  ; Pre-decrement
0900 A629           2    ldaa   7,-X   ; X-7 -> X, (X) -> A
                    3  ; Post-decrement
0902 A63E           4    ldaa   2,X-   ; (X) -> A, X-2 -> X
                    5  ; Pre-increment
0904 A620           6    ldaa   1,+X   ; X+1 -> X, (X) -> A
                    7  ; Post-increment
0906 A630           8    ldaa   1,X+   ; (X) -> A, X+1 -> X
```

Accumulator Offset
Indexed Addressing

An **offset** contained in the accumulator (A, B, or D), is added to the value contained in an **index register** (X, Y, SP or PC). Note: **memory locations** cannot be used as **offset** or **index.** The sum is the effective address; the registers remain unchanged. This addressing mode allows referencing any memory location in the 64-KB address space.

Example: `LDAA B,X ; (X+B)-> A`




```
09B0 A6E5      1   ldaa   B,X    ; (X+B) -> A
09B2 E6EC      2   ldab   A,Y    ; (Y+A) -> B
09B4 EDE6      3   ldy    D,X    ; (X+D:X+D+1) -> Y
```

Summary of HCS12 Indexed Addressing

Operand Syntax	Comments
ldaa ,r ldaa n,r	5-, 9- or 16-bit, signed, constant offset n = -16 to +15 for 5-bit n = -256 to +255 for 9-bit n = -32,768 to 32,767 for 16-bit r can be X, Y, SP, or PC; r is not changed by the instruction.
ldaa n,-r	Automatic pre-decrement n = 1 to 8 and is subtracted from the contents of register r before the data value is fetched. r can be X, Y, or SP (not PC); r is modified by the instruction.
ldaa n,+r	Automatic pre-increment n = 1 to 8 and is added to the contents of register r before the data value is fetched. r can be X, Y, or SP (not PC); r is modified by the instruction.

Summary of HCS12 Indexed Addressing

Operand Syntax	Comments
ldaa n,r-	Automatic post-decrement n = 1 to 8 and is subtracted from register r after the data value is fetched. r can be X, Y, or SP (not PC or memory); r is modified by the instruction
ldaa n,r+	Automatic post-increment n = 1 to 8 and is added to register r after the data value is fetched. r can be X, Y or SP (not PC or memory); r is modified by the instruction.

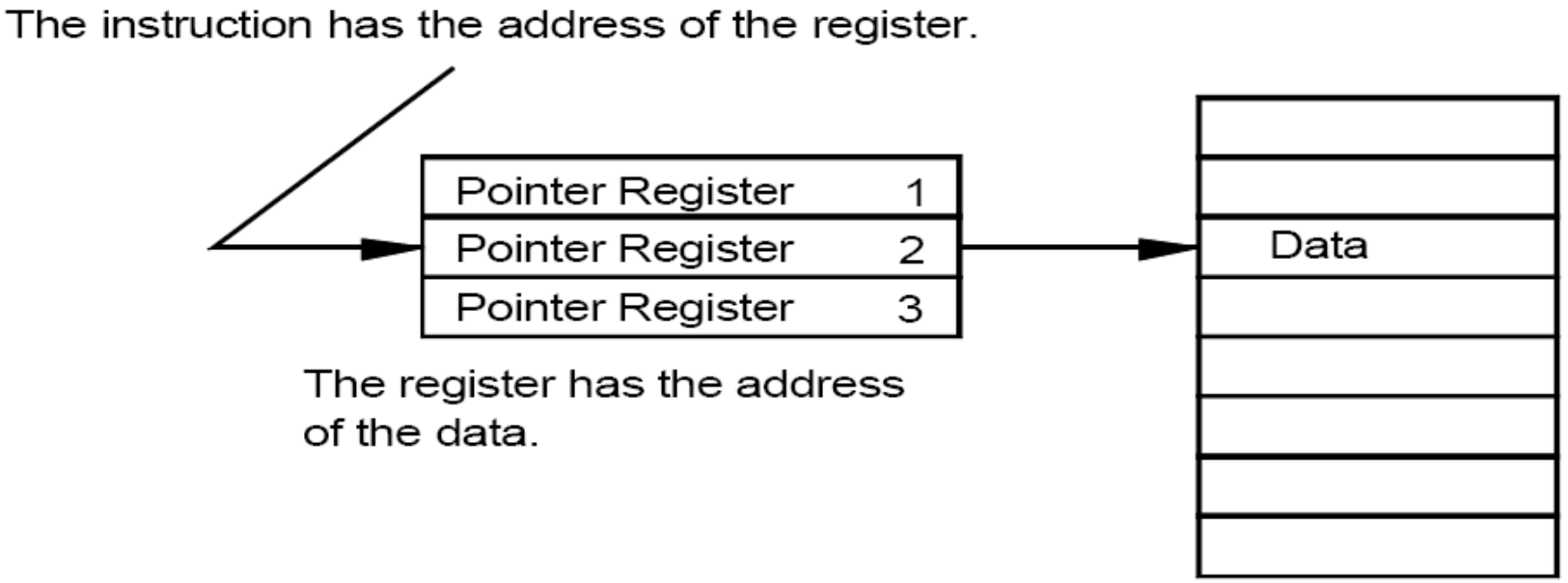
ldaa A,r
ldaa B,r
ldaa D,r
	Accumulator offset
	The contents of A, B, or D are used as a 16-bit, unsigned offset.
	r can be X, Y, SP, or PC; r is not changed by the instruction.

Indirect Addressing

- Two level addressing mechanism
- First level provided by instruction gives address of memory containing an address
- Second level is the address that specifies where the data is located
- **Register Indirect**: Register contains the data address
- **Register Indirect with auto-increment or auto-decrement**: register automatically incremented/decremented
- **Memory Indirect**: Data address found in memory

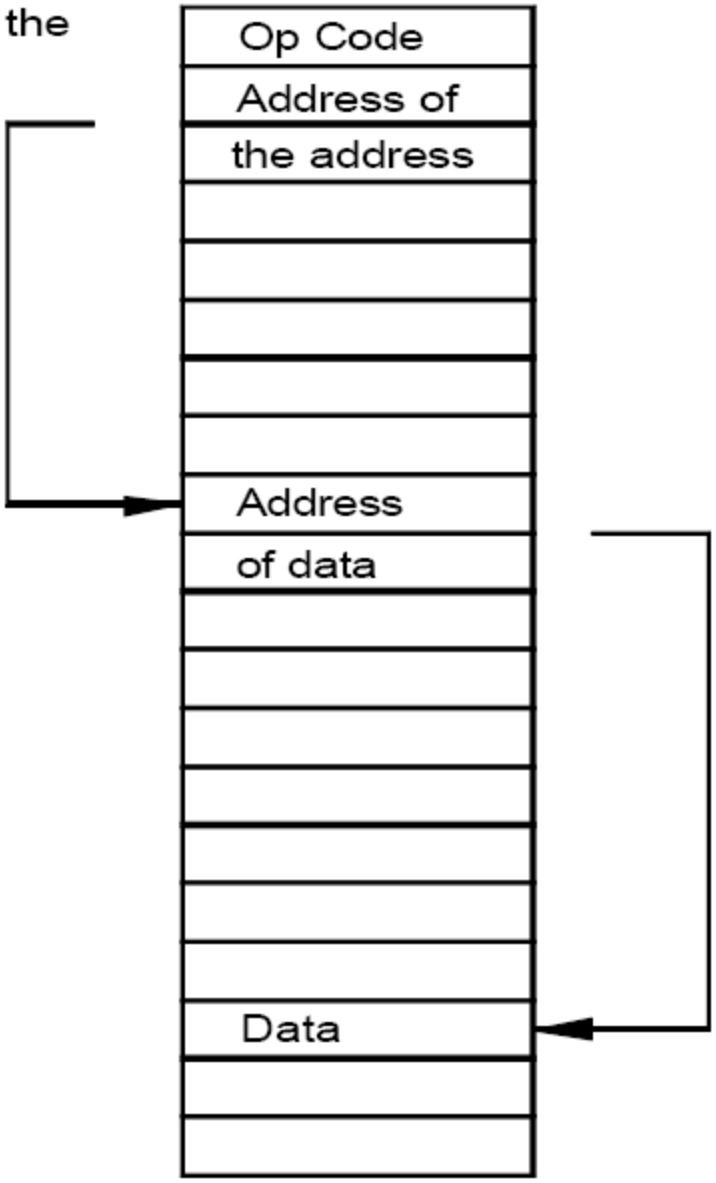
Register Indirect Addressing

- Also called *pointer register addressing*
- Instruction contains address of register holding the data address
- Data address can be calculated at run time
- Registers may also be incremented or decremented automatically



Memory Indirect Addressing

The instruction has the address of the address.

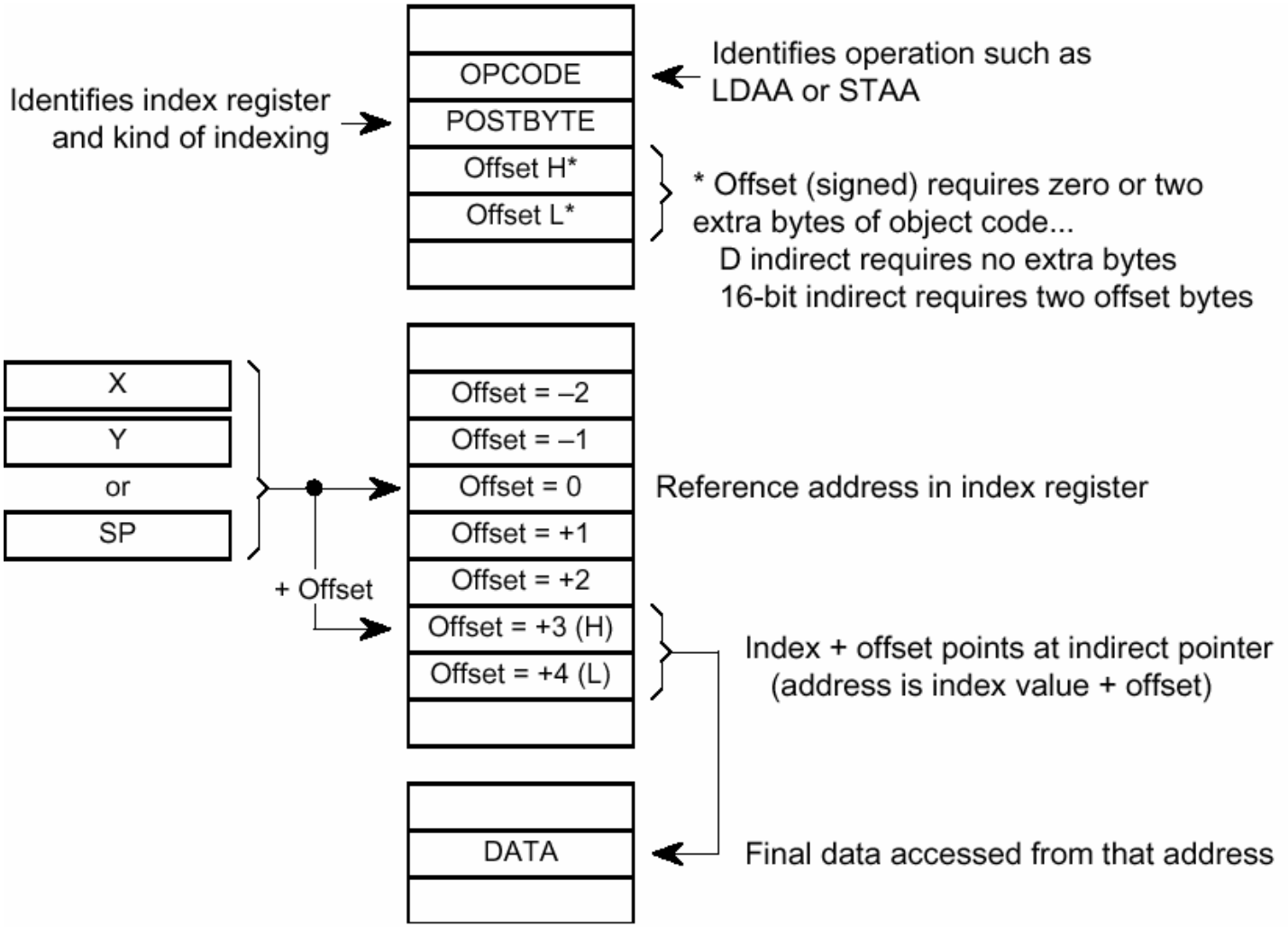


The address in memory points to the data.

HCS12 Indexed-**Indirect** Addressing Modes

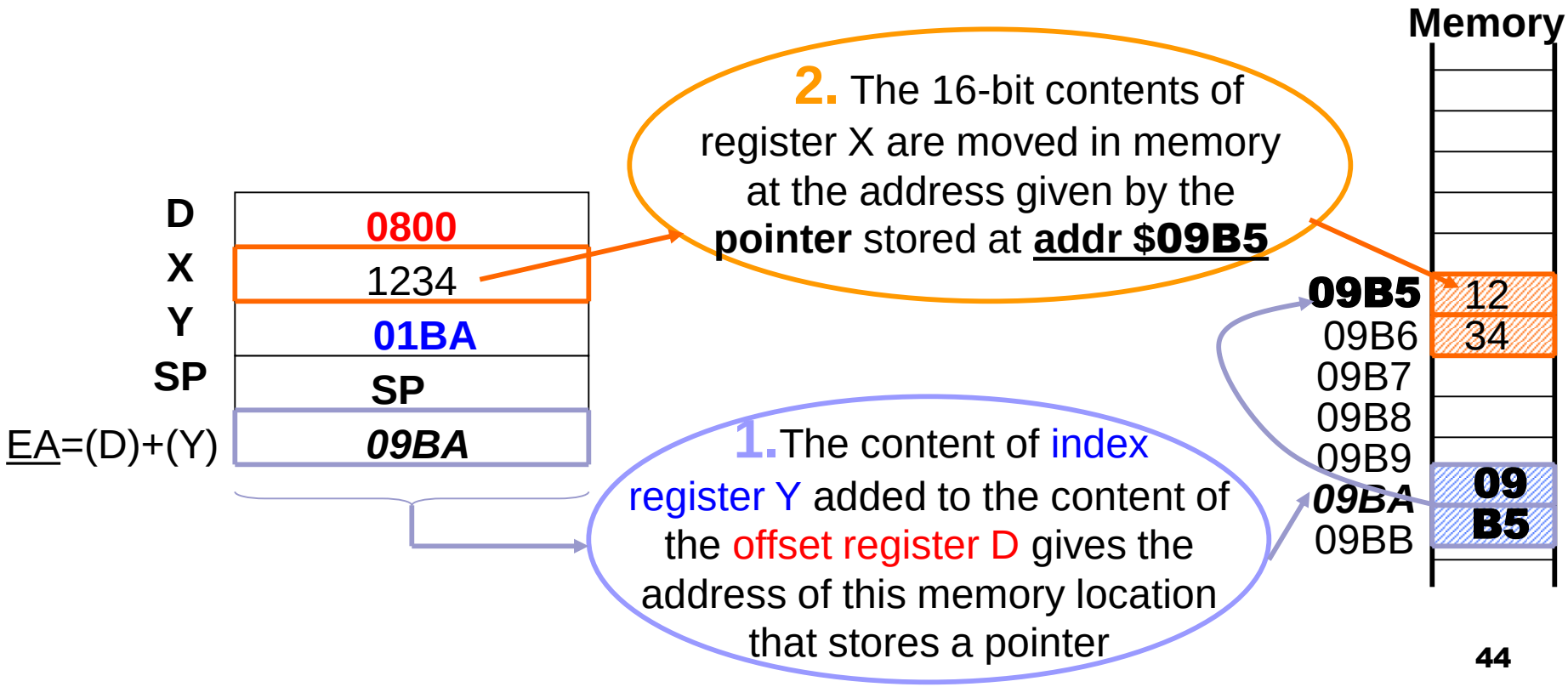
- Provides memory_indexed-indirect addressing using
  1. a constant offset
     - Instruction provides a 16-bit offset
  2. accumulator offset
     - D register contains the 16-bit offset
- 16-bit offset is added to the index register to obtain the effective address to the data address
  - The address found at the effective address is then used to address the data

HCS12 16-bit Constant Indexed-**Indirect** Addressing



HCS12 Accumulator D Indexed **Indirect** Addressing

Example: **`stx [D,Y]`** **`;X <- m[m(D+Y)]`**



HCS12 Indexed-Indirect Instructions

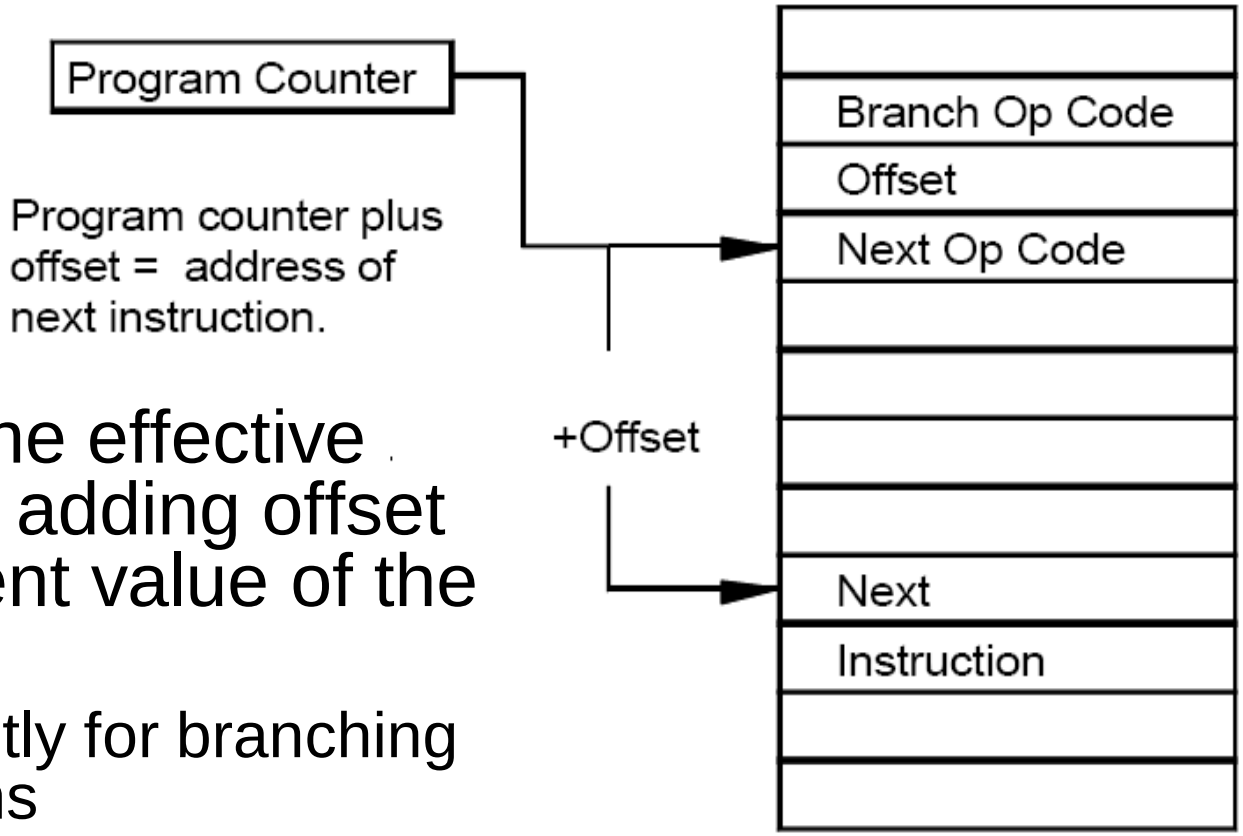
- Instruction Format
  1. Operation [Offset, Index_register]
     - 16-bit Offset
  2. Operation [D, Index_register]
  - Index_register can be X, Y, SP, PC

```
;16-bit Constant Indexed-Indirect Addressing
0000 CE5000        1   ldx    #$5000        ; $5000 -> X, Initialize X
0003 A6E30064      2   ldaa   [$64,X]      ; (($5064)) -> A
0007 6AE3FFFF      3   staa   [-1,X]       ; A -> (($4fff))
;Indexed-Indirect Addressing using Accumulator D
0000 CE5000        1   ldx    #$5000        ; $5000 -> X, initialize X
0003 CC0064        2   ldd    #$064         ; $0064 -> D, initialize D
0006 EFE7          3   lds    [D,X]         ; (($5064)) -> SP
```

Summary of HCS12 Indexed-Indirect Addressing

Operand Syntax	**Comments**
ldaa [n,r]	16-bit constant offset **indirect** indexed r can be X, Y, SP, or PC; r is not changed by the instruction.
ldaa [D,r]	Accumulator D offset **indirect** indexed r can be X, Y, SP, or PC; r is not changed by the instruction.

Relative Addressing



- Calculate the effective address by adding offset to the current value of the PC
  - Used mostly for branching instructions
  - Normally offset is calculated from the address of the next op-code

HCS12 Relative Addressing

- Short branching instructions: 8-bit offsets (offset ranges between -128 to +127)
- Long branching instructions: 16-bit offsets (offset ranges between -32,768 to +32,767)
- Loop primitive instructions that use 9-bit offsets (range between -256 to +255)

```
0000 2002        1  THERE:       bra    WHERE   ; Forward branch
0002 A7          2                nop
0003 A7          3                nop
0004 22FA        4  WHERE:       bhi    THERE   ; Conditional branch back
0006 18260256    5                lbne   LONG_BRANCH
000A             6                DS     256     ; Simulate instructions
                 7  LONG_BRANCH:
0260 A7          8                nop
```

HCS12 Addressing Modes

Each addressing mode except inherent mode generates a 16-bit effective address (EA) which is used during the memory reference portion of the instruction. Effective address computations do not require extra execution cycles.

Addressing Mode	Operand	CPU12	Syntax	Operand, EA
IMPLIED (register)		**Inherent:**	opcode	Operands (if any) are in CPU registers
IMMEDIATE	data	**Immediate:**	opcode #opr8i	opr8i= One byte of immediate data
opcode data			opcode #opr16i	opr16i= High/low-order byte of 16-bit immediate data
ABSOLUTE	(addr)	**Direct**	opcode opr8a	opr8a = 8-bit direct address ($0000–$00FF) => it's about I/O registers
opcode addr	EA = addr	**Extended:**	opcode opr16a	opr16a=high/low-order byte of 16-bit extended addr.
RELATIVE	(EA)	**Relative**	opcode rel8	EA = (PC)+rel8 (for branch instructions)
opcode offset			opcode rel16	EA = (PC)+rel16 (for branch: PC=(PC)+offset)
INDEXED	(EA)	**Constant offset**	opcode oprx,xysp	EA=xysp+oprx
		Pre-de/increment	opcode oprx3,–/+xys	xys=(xys)-/+oprx3 then EA=xys
		Post-de/increment	opcode oprx3,xys–/+	EA=xys then xys=(xys)-/+oprx3
		Register offset	opcode offsetREG, indexREG	EA=(offsetREG)+(indexREG)
INDIRECT	((EA))	**Indexed Indirect**	opcode [offset, indexREG]	EA=[(offset)+(indexREG)]
			opcode [offsetREG, indexREG]	EA=[(offsetREG)+(indexREG)]

Stack Addressing

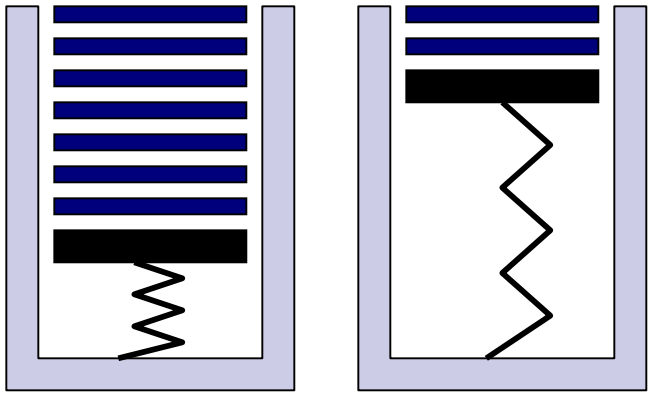
- Stack is an area of memory (RAM) reserved as temporary storage area
  - Operates on a last-in first-out (LIFO) basis
- Stack Pointer (SP) register
  - Used to store and retrieve data on the stack
  - Also used to store return addresses when calling subroutines
  - Also used to store CPU register data when processing interrupts
  - Can either point to last information put on the stack or the next available memory location
  - Must be initialised to point to memory before using stack instructions

Stack Operations

- Push and Pull operations
  - Push operation places data on the stack
  - Pull (also called pop) operation retrieves data from the stack
  - Each operation affects the value of the SP
- Subroutine Call and Return Operations
  - When branching to a subroutine (BSR), the return address is stored on the stack
    - Return address correspond to the address of the instruction following the branch to subroutine instruction
  - At the end of the subroutine, the return instruction (RTS) pulls the return address from the stack
    - PC is updated with the return address
- Interrupts and Interrupt Return Operations
  - Similar operation to subroutine calls and return, except that all CPU registers are stored on the stack (will study later)

The Stack is…

- a special area of RAM
- used for
  - Temporary data storage
  - Parameter and control information storage for subroutines
  - General data storage



Analogy:

the plate rack (used in some cafeterias)

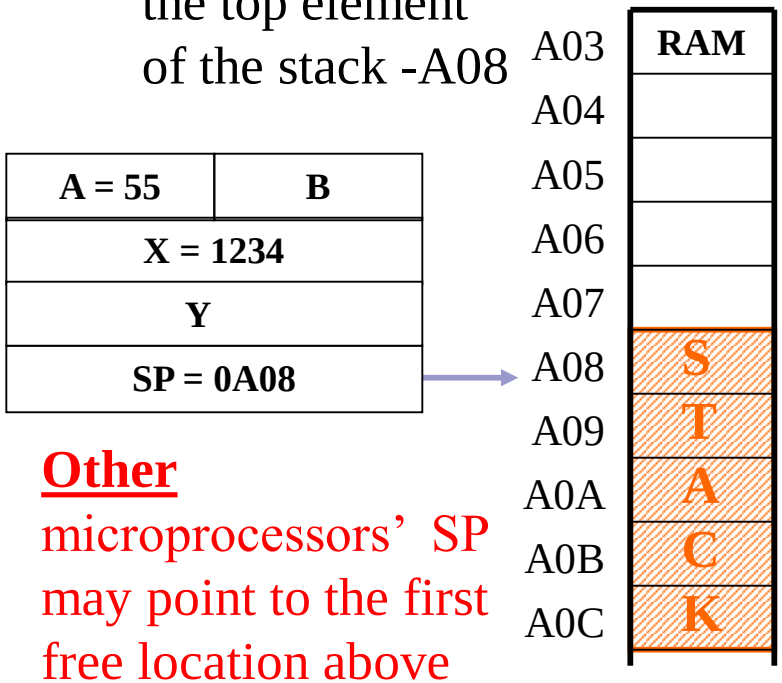
- –Each customer takes a plate off the top of the rack
- –A mechanism moves the plates upward, so that the next plate is at the top
- –When more clean plates are put into the rack, the mechanism moves downwards

It's a **LIFO** (last-in first-out) device

The Stack Pointer (SP) – HCS12

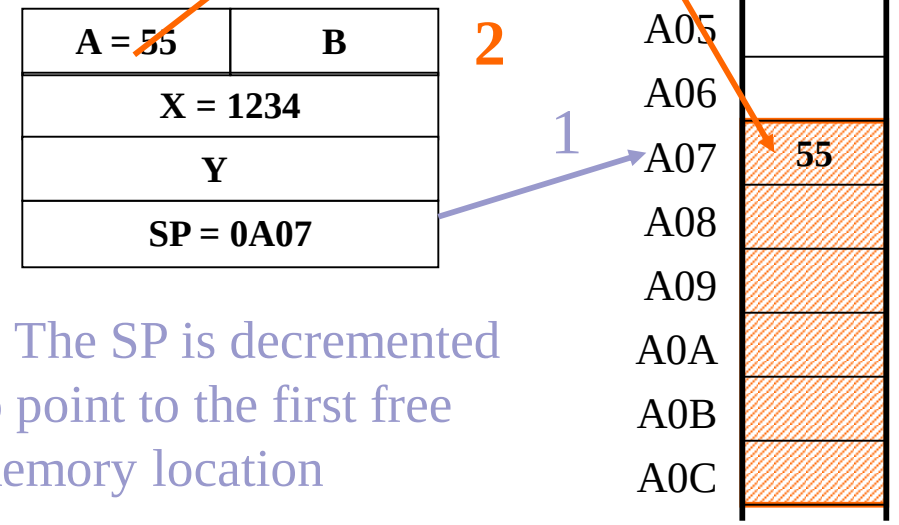
SP = stack's address register

68HC12 SP points to the top element of the stack -A08



Other microprocessors' SP may point to the first free location above the top element of the stack (A07)!!!!

PUSH a new element to the stack => the SP grows up (to smaller addresses) pointing to the address of the new elem.



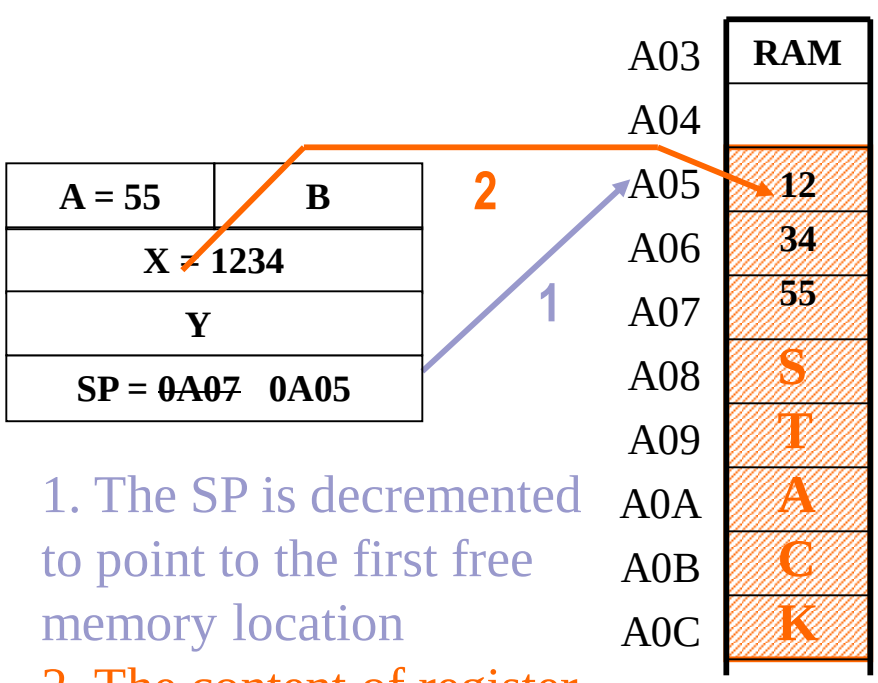
1. The SP is decremented to point to the first free memory location

2. The content of accumulator A is copied to the STACK to this free memory location

Example: `psha`

```
;1. SP <- SP-1 , i.e., SP = A07
;2. m(SP) <- A , i.e., m(A07)=55
```

PUSH - PULL

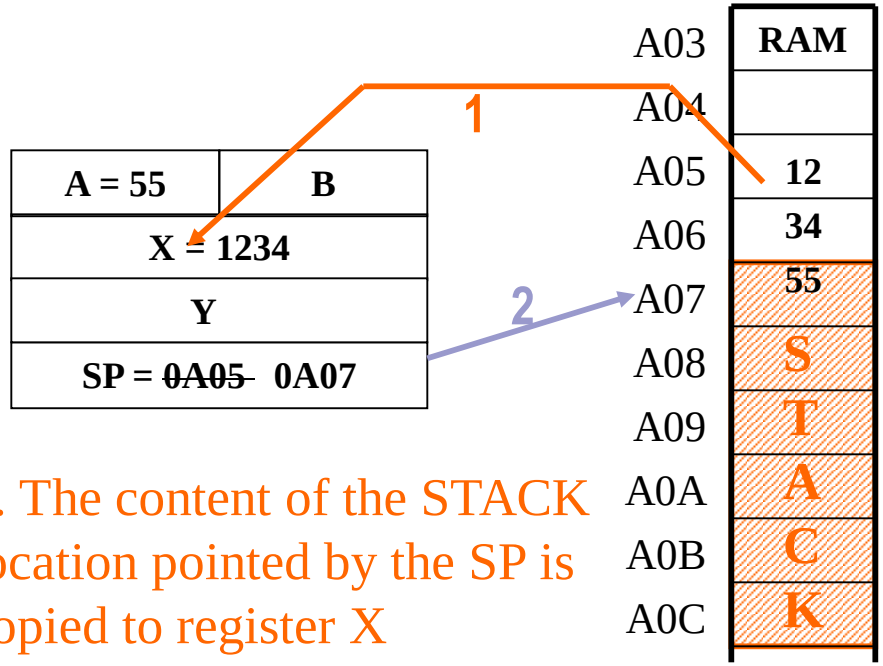


1. The SP is decremented to point to the first free memory location
2. The content of register X is copied to the STACK

Example: pshx

;1. SP <- SP-2, i.e., SP = A05

;2. m(SP, SP+1) <- X



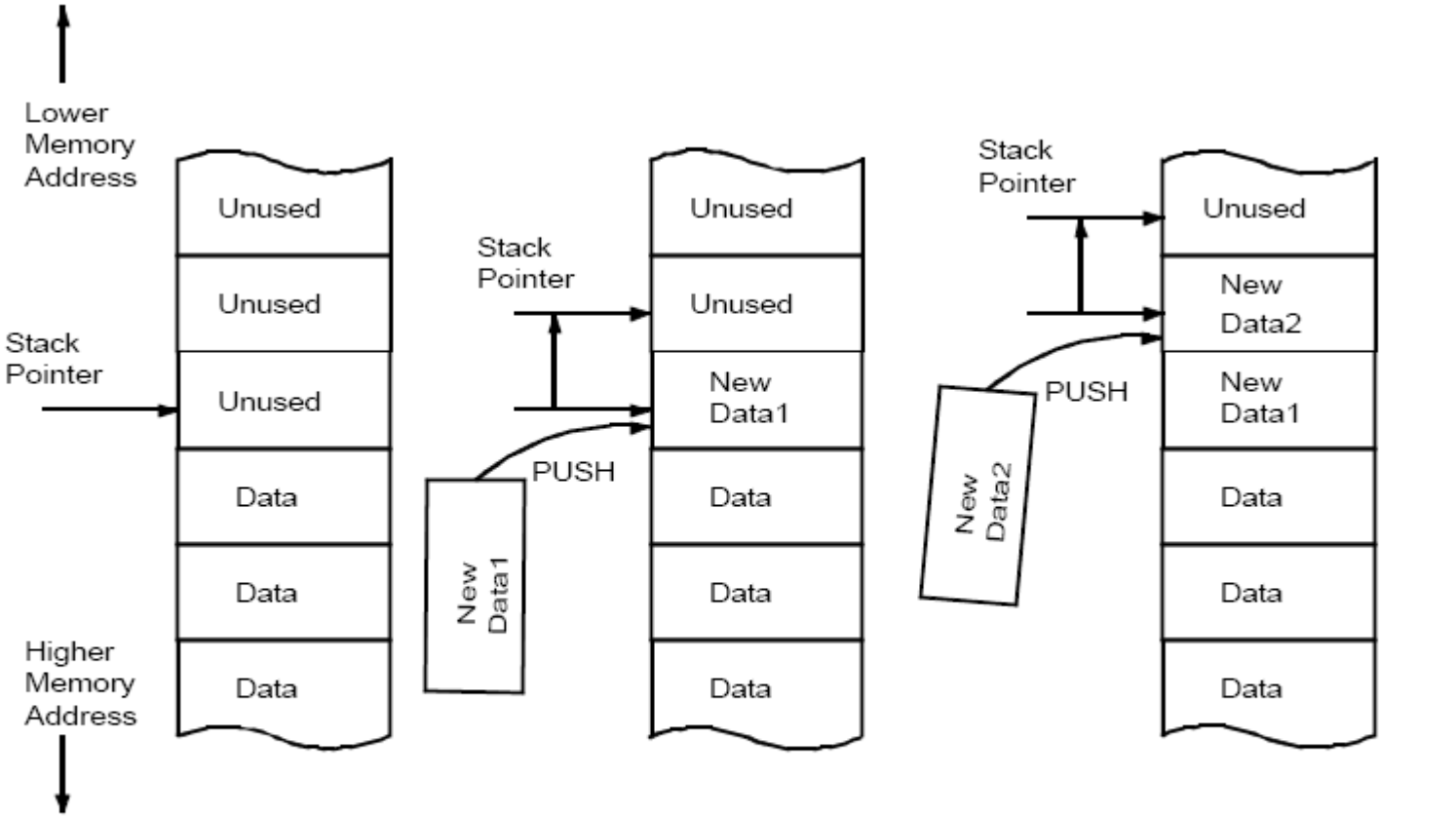
1. The content of the STACK location pointed by the SP is copied to register X
2. The SP is incremented to free the STACK, i.e., to point to the next element

Example: pulx

;1. X <- m(SP, SP+1), i.e., X=[m(A05),m(A06)]

;2. SP <- SP+2, i.e., SP = A07

Push Operation – non HCS12 µP

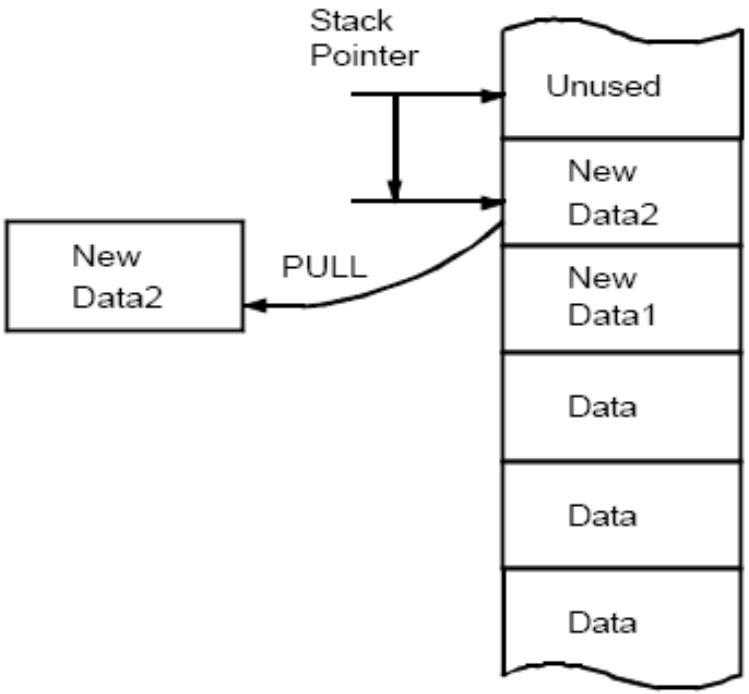


(a) Stack pointer before stack operations.

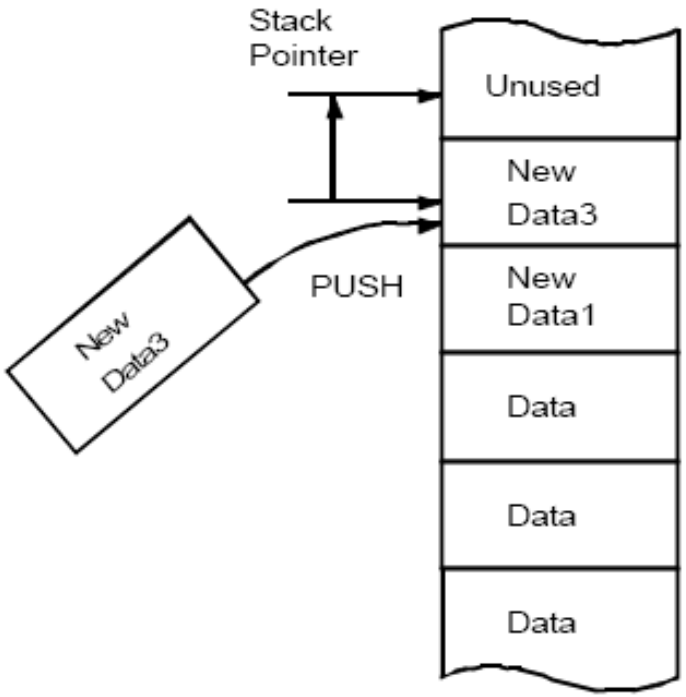
(b) Stack pointer after a a push operation.

(c) Stack pointer after second push operation.

Pull Operation – non HCS12 µP

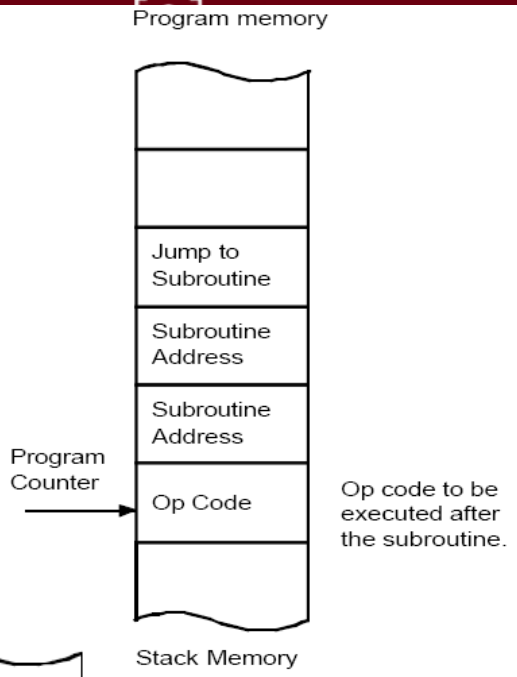


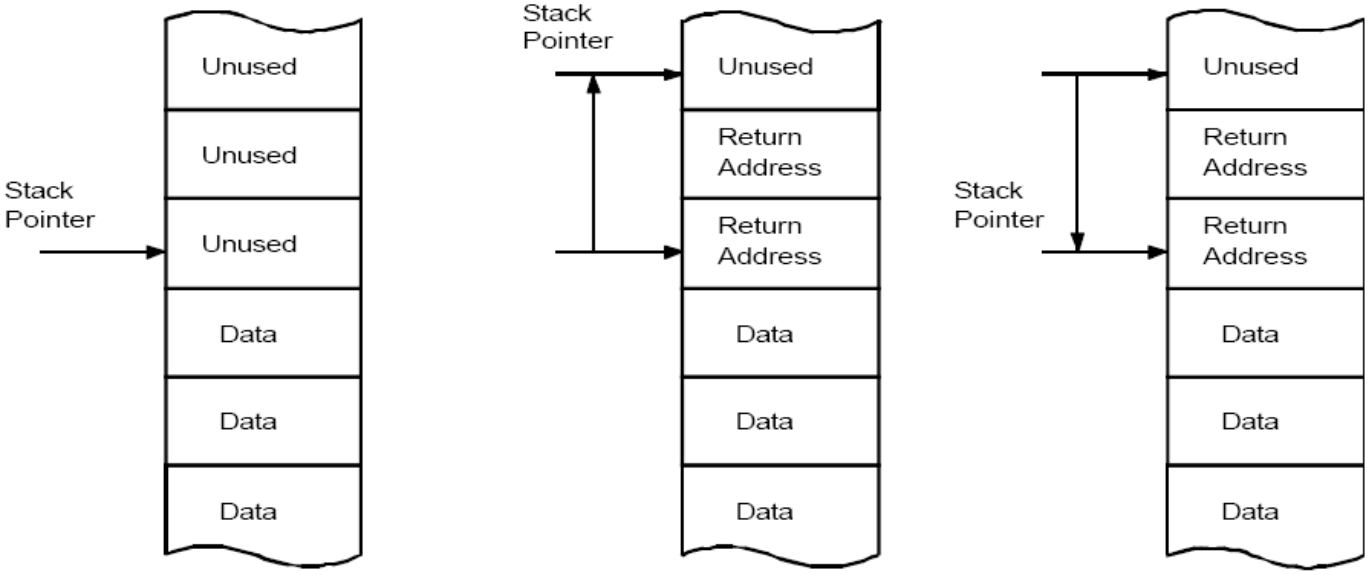
(d) Stack pointer after a pull operation.



(e) Stack pointer after third push operation.

Stack Operations during Subroutine Call and Return – non HCS12 µP



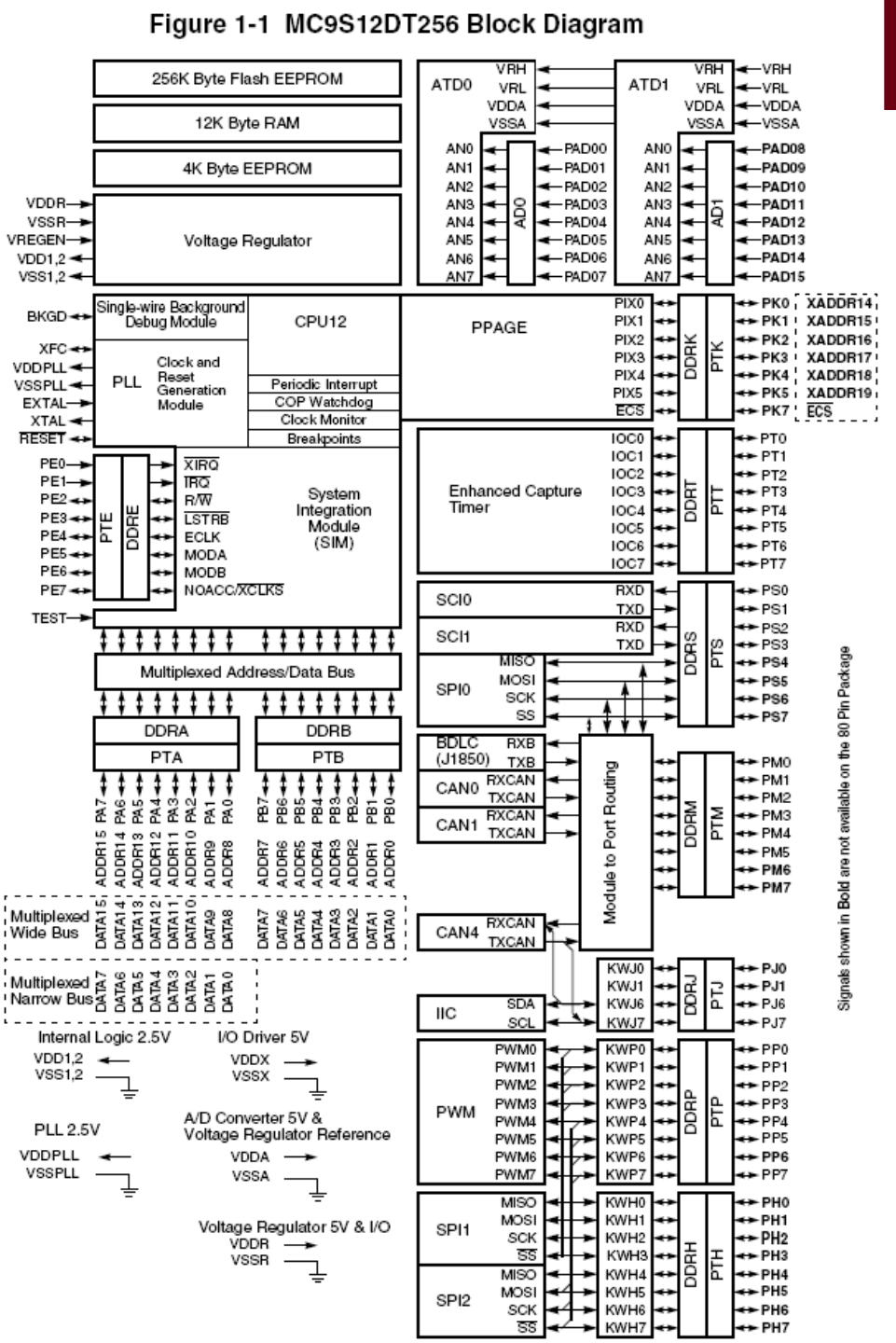


(a) Stack pointer before subroutine branch.

(b) Stack pointer after a jump to subroutine instruction.

(c) Stack pointer after return from subroutine.

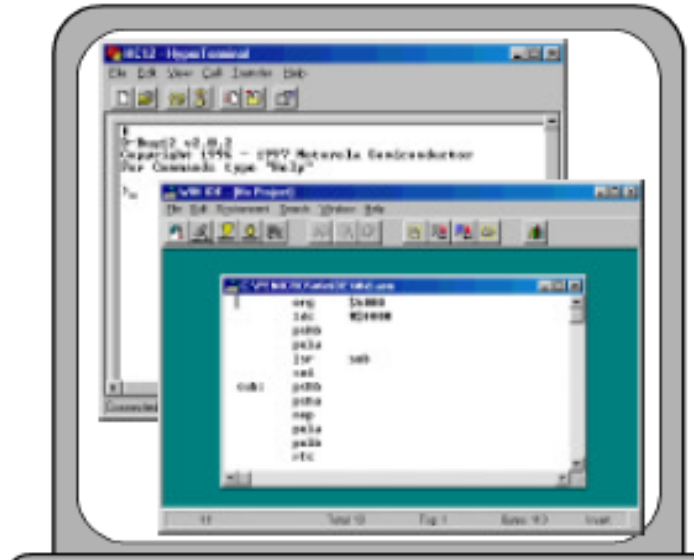
MC9S12DG256 Block Diagram



References

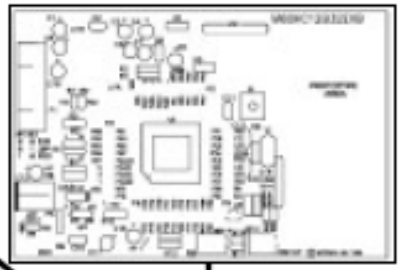
- Fredrick M. Cady, Software and Hardware Engineering: Assembly and C Programming for the Freescale HCS12 Microcontroller
- S12CPUV2, Rev. D, Reference Manual, Freescale Semiconductor Inc.
- MC9S12DT256 Device User Guide (covers also MC9S12DG256), V03.07, Freescale Semiconductor Inc.
- Reference Guide for D-Bug12 Version 4.x.x, Gordon Dougham
- Above are sources for most of the figures, tables and examples in the course notes

Lab Environment – Assembly Language



Serial port (com 2)

MC**68HC912**B32**EVB** development board with



PC with serial connection:

- **Hyperterminal**
- **WinIDE**

D-Bug12 **monitor** / debugger program

Instructions Needed for Lab 1

Load Instructions:

LDAA address	Put the byte contained in memory at address into **A** (N,Z)
LDX address	Load Index Register X (N,Z)

Decrement and Increment Instructions

DECA	Subtract one from the content of accumulator A. (N,Z,V)
INX	Add one to index register X. (Z)

Short Branch Instructions

BNE address	Tests the Z status bit and branches if Z = 0.
NOP	

Stack Operation Instructions

PSHA	The stack pointer is decremented by one. The content of A is then stored at the address the SP points to.
PULA	Accumulator A is loaded from the address indicated by the SP The SP is then incremented by one.

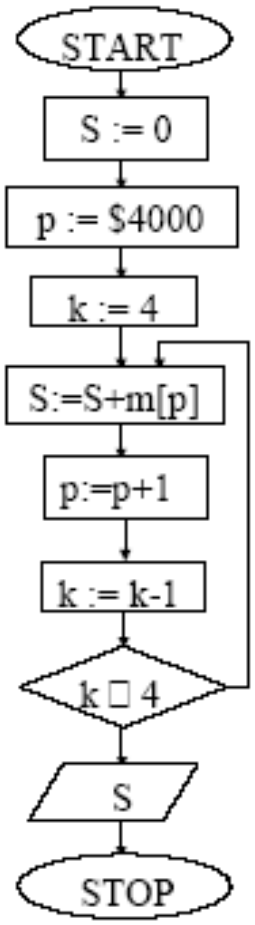
Subroutine Instruction

JSR address

- Sets up conditions to return to normal program flow, then transfers control to a subroutine. Uses the address of the instruction following the JSR as a return address.
- Decrements the SP by two to allow the two bytes of the return address to be stacked.
- Stacks the return address. The SP points to the high order byte of the return address.
- Calculates an effective address according to the rules for extended, direct, or indexed addressing.
- Jumps to the location determined by the effective address.
- Subroutines are normally terminated with an RTS instruction, which restores the return address from the stack.

Interrupt Instruction

SWI Software Interrupt (Used to end all our HC12 programs)



Variable	Register
S	A
p	X
k	B

A Simple Program

A 4-element (1-dimensional) array is stored in memory starting from address $4000.
Write a program to add all array's elements and put the sum in A.

Instruction		Operation	Comments	Address	Memory Content
LDAA	#0	A ← 0	S:=0	$800	86
					0
LDX	#$4000	X ← $4000	Pointer to the first element	802	CE
					40
					00
LDAB	#$02	B ← 4	Init. Counter (k := 4)	$805	C6
					02
ADDA	0,X	A ← (A) + m[(X)]	S:=S+m[p]	$807	AB
					00
INX		X ← (X)+1	Point to next element	$809	08
DECB		B ← (B) - 1	Decrement counter	$80A	53
BNE	$807	PC ← (PC) - 5	Redo loop if more to be added	809	26
					FA
SWI		SW Interrupt	Back to D-Bug12	080B	3F

Example: Identify the destination of the data operand and the value of the data operand in the last instruction of the following sequence.

PULA
INCA
STAA -6,SP