uOttawa

L'Université canadienne
Canada's university

CEG3136 Computer Architecture II

Module 4 – Instruction Set Architecture (ISA)

Notes for
Dr. Voicu Groza

Université d'Ottawa | University of Ottawa

Topics of discussion

- CISC and RISC Architectures
  - Reference: RISC – Wikipedia article (http://en.wikipedia.org/wiki/RISC)
- CPU12 Instruction Set
  - Instruction Set Categories
  - Move Data Categories
  - Modify Data Categories
  - Decision Making Categories
  - Flow Control Categories
  - Other Categories
- Reading: Chapter 7 in text, CPU12RG – CPU12 Reference Guide (for HCS12 and original M68HC12)
- Bring the Reference Guide to class

Processor Architectures

- The processor is the functional brain of the computer
  - Sometimes referred to as the Central Processing Unit (CPU)
  - Can be classified by functional characteristics, operational characteristics or even timing characteristics
  - Most important feature is: performance!
- We have to increase the amount of work the processor does in a given period of time. How ?
  - Execute instructions in less time OR
  - Make each instruction do more work….
- Real-life analogy: riding a bike
  - Use a low gear and pedal very quickly OR
  - Use a high gear and push harder

CISC Architecture

- Architecture refers to a microprocessor instruction set architecture (ISA)
- Early CPU ISA design included complex instructions
  - In the early days, memory was expensive, small and slow
  - Compiler technology did not exist
  - Programming was done in assembly language (machine code)
  - Attitude: hardware design easier than software design
- The term CISC (complex instruction set computer)
  - RISC (reduced instruction set computer) processors were designed to simplify ISAs
  - The term CISC was coined later to distinguish the complex ISAs form the simpler RISC ISAs
  - Although RISC processors provided advantages over CISC processors, they did not gain the popularity as expected, particularly with desktop PCs

CISC Architecture

- CPU design was influence by factors in the previous slide
  - With more complex instructions, programs were reduced in size
  - Simplified the task of the assembler programmer
  - Few internal registers
    - Internal memory was expensive
    - Large number of internal registers increased the instruction size that increased the program size
  - One instruction did a lot of work: for example, load data into registers, add numbers, and store result into memory
    - Made available all addressing modes to all instructions
- Examples of CISC processors: VAX, PDP-11, Motorola 68000 processors, and Intel x86/Pentium CPUs
- The HCS12 contains a CISC CPU

CISC Processors

- CISC = Complex Instruction Set Computer
  - Reduce the amount of instructions required to do work
  - Considered more superior for a long time (memory was very expensive)
- CISC defining characteristics
  - Higher clock rate (1 GHz and above)
  - Much more than 100 executable instructions
  - Many addressing modes
  - Variable instruction format (16- to 64-bit words)
  - High CPI (15-20 for certain instructions)

RISC Architecture - Motivation

- IBM research showed in the late 70's that majority of addressing modes were ignored
  - Side effect of using compilers instead of assemblers
  - Compilers had limited ability to take advantage of CISC CPU features
- Some CISC complex operations were slower than smaller number of simpler operations
  - Example: VAX INDEX instruction ran slower than a loop implementing the same code
- CPUs started to run faster than memory
  - Wanted more internal registers (later becomes caches)
- CISC CPUs were shown to be over designed
  - Example: Andrew Tanenbaum showed that 98% of constants fit in 13 bits
  - By reducing number of bits to represent instructions, could reduce the length of instructions by reducing the number of operand bytes (even include operands with opcode bits)

RISC Design Philosophy

- Name RISC (reduced instruction set computer) resulted from small number of addressing modes and instructions
  - Actually size of the instruction set could be quite large
  - Real difference was increasing the size in number of internal registers where the work was done
  - Required loading and storing data – design referred to as *load-store*.
- Smaller opcodes
  - Leaves more room for including the operands directly in the instruction
- Disadvantages
  - Series of instructions needed to complete even simple tasks
    - Leads to larger programs and more I/O with memory
    - Not clear that the RISC design would provide a net gain in performance

Improving performance

- In late 1980's, number of ideas improved performance dramatically
- Pipeline was technique that broke down instruction into steps
  - Could work on steps from different instructions at the same time
- Another approach was to use several processing elements to process instructions in parallel
  - Difficult to do as some instructions depend on results from others
- These techniques added complexity to layout of CPU
  - Space on CPU chips was limited
  - RISC CPUs being simpler could quickly adopt these techniques
  - Soon outperforming CISC CPUs
  - But CISC CPUs were able to eventually incorporate these techniques

Using RISC CPU Chip Space

- RISC core logic required fewer transistors
- Designers could use the additional space
  - Increase the size of the register set and add large caches
  - Implement measures to increase internal parallelism
  - Add other functionality such as I/O or timers (microcontrollers)
  - Do nothing and use chip in battery-constrained or size-limited applications.

RISC Processors

- RISC = Reduced Instruction Set Computer
  - Invented in 1974 in IBM research
  - “Computer only uses 20% of instructions”
  - 801 prototype (1975) was first RISC microprocessor, which used less transistors, as less instructions were handled
  - ‘RISC’, the term, was coined by none other than David Patterson!
- RISC defining characteristics
  - Driven by hardwired logic (usually)
  - Lower clock rate (500 MHz)
  - Less than 100 executable instructions (most of them register-based)
  - Few addressing modes (usually less than 8)
  - Fixed instruction format (MIPS = 32-bit words)
  - Very low CPI (< 2 for certain instructions)

General RISC Features

- Uniform Instruction Encoding
  - For example, op-code in same bit position in each instruction that is always one word long
  - Allows faster decoding
- Homogeneous register set
  - Any register can be used in any context
- Simple addressing modes
  - Complex addressing modes replaced with simple arithmetic instructions
- Few data types
  - Some CISC CPUs dealt with byte strings, polynomials, complex numbers, etc.

RISC vs CISC Processors

- Which one is better ? More popular ?
  - No real better solution
  - Most of the CISC instructions are underutilized (20-80 rule)
  - Eliminating those instructions takes us back to RISC
  - Both needed in their own domains
    - RISC for dedicated processing (e.g. file and web servers)
    - CISC for general-purpose processing
- Differing Notes
  - RISC has smaller number of instructions, but larger programs
  - Can execute instructions 4x or 5x faster!
  - However, clock rate is not high!
  - CISC should be more expensive than RISC, but popularity (and competition) of design architecture has lowered prices
  - AMD, Cyrix, Intel (CISC) vs. Apple (RISC)

RISC versus CISC

- Common RISC processors include ARM, DEC Alpha, SPARC, MIPS, PIC and PowerPC
- Sun's Microsystems SPARC made its way into the workstation market
- MIPS Computer Systems created the MIPS chip R2000 that were used in the PlayStation and Nintendo 64 game consoles
- IBM created the PowerPC now used in all Apple Macintosh machines and commonly used in automotive applications
- Most vendors joined in creating RISC CPUs including Intel (i860 and i960 in the late 1980s) and Motorola (88000 and eventually joined IBM to create the PowerPC)
- But RISC has made few inroads into the desktop PC and commodity server markets

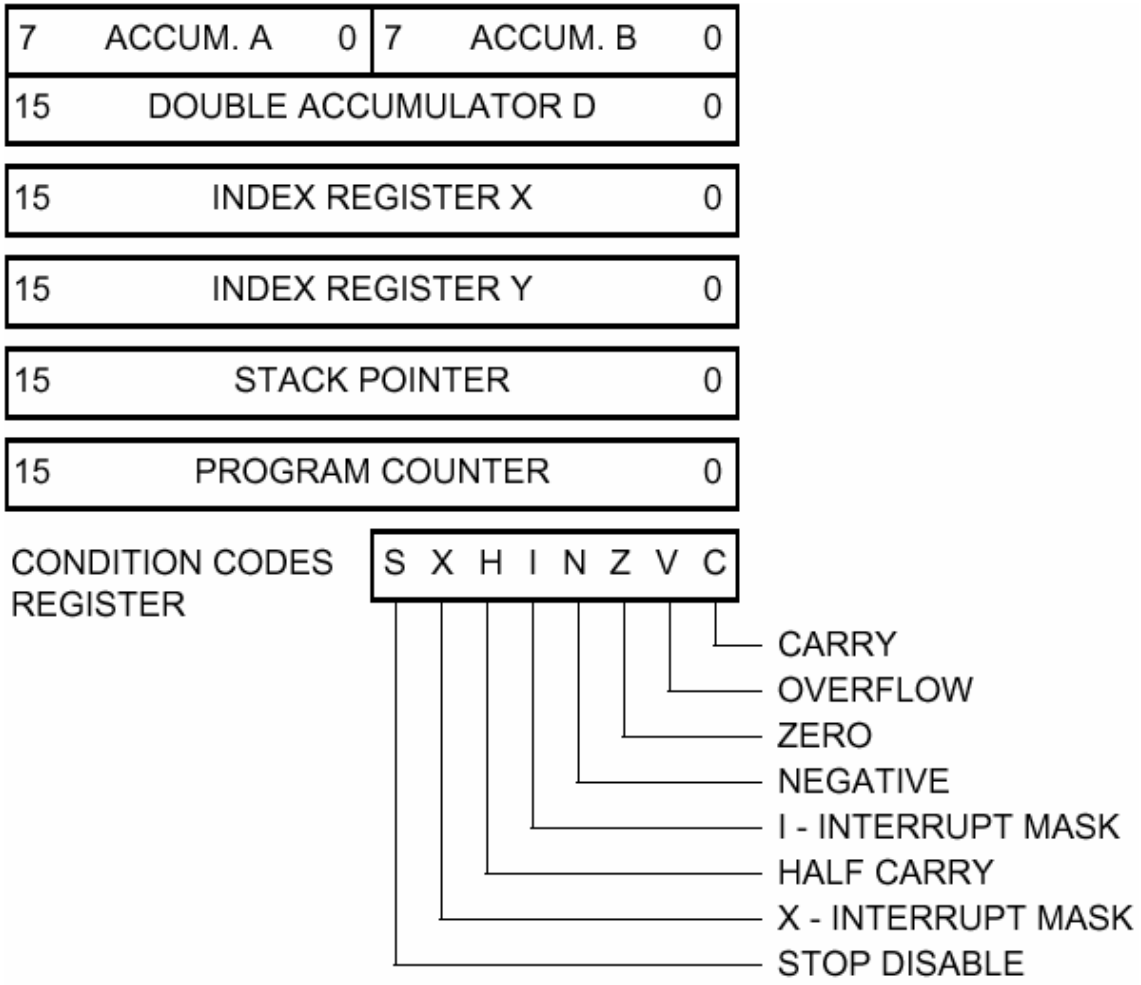
RISC versus CISC

- Intel's x86 platform remains dominant processor architecture in the PC market for three main reasons
  - The x86 has a large base of proprietary applications
  - Intel was able to throw all necessary funds to create CISC CPUs that could outperform RISC CPUs
  - Intel (and AMD) started to integrate RISC design philosophies into their CPUs
    - For example the PentiumPro CPUs implement CISC instructions by using simpler RISC operations internally.
- As of 2004, the x86 chips are the faster CPUs

Topics of discussion

- CISC and RISC Architectures
  - Reference: RISC – Wikipedia article (http://en.wikipedia.org/wiki/RISC)
- CPU12 Instruction Set
  - Instruction Set Categories
  - Move Data Categories
  - Modify Data Categories
  - Decision Making Categories
  - Flow Control Categories
  - Other Categories

CPU12 CPU Programmer's Model

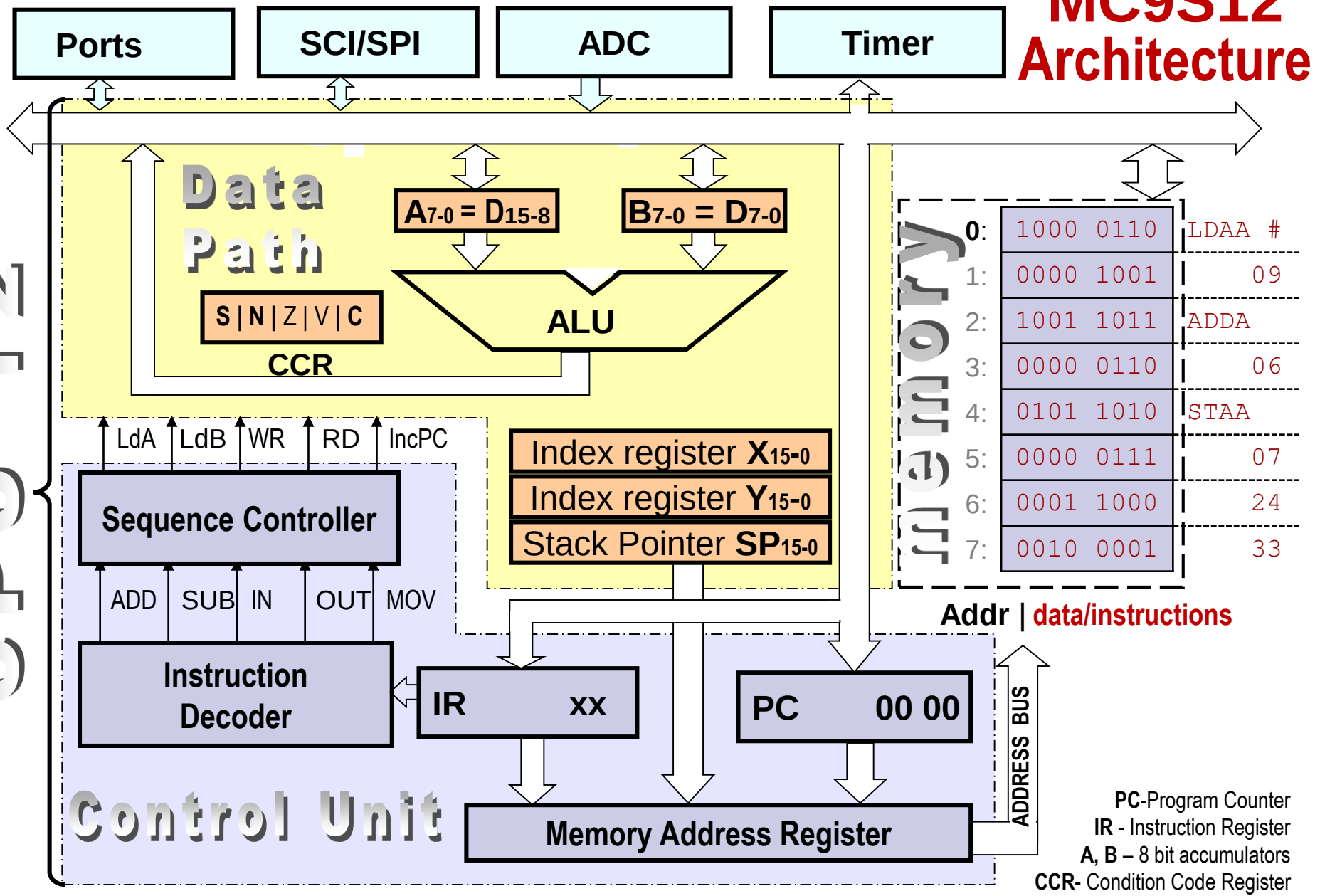


The CPU12 Instruction Set

- Contains over 1000 instructions
- Instructions include 188 operations
- Can be divided into 18 different categories
- Programming consists of:
  - Determine instruction category according to desired operation
  - Select the addressing mode
  - Must manage the resources in the CPU (programmer's model)
- Consult Table 7-2 and Appendix B in textbook for summary of all instructions
- The Freescale *CPU12 Reference Manual* (CPU12RM) provides complete descriptions of the instruction set.
- The Motorola *CPU12 Reference Guide* provides a more compact version of the instruction set (CPU12RG).

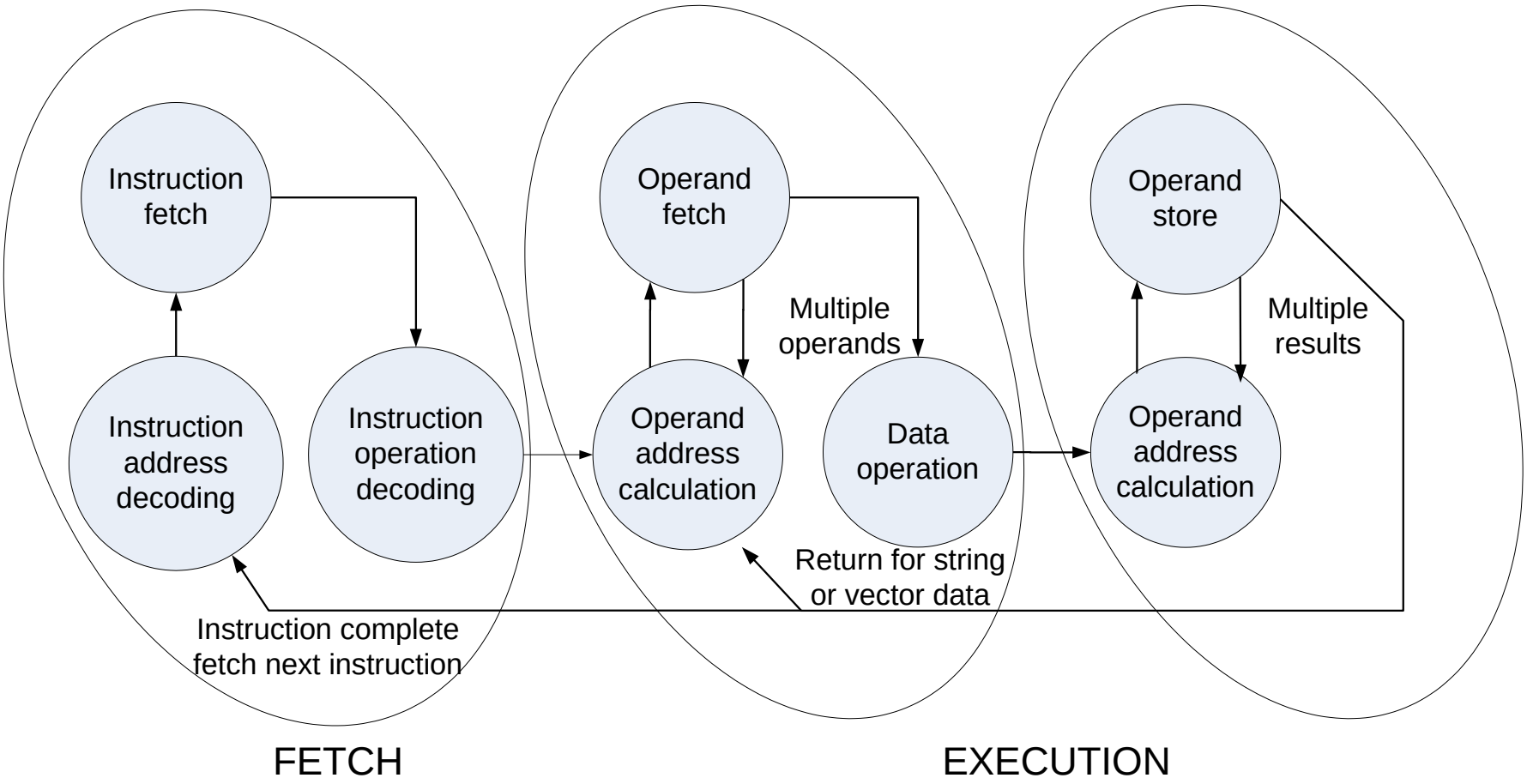
uOttawa

MC9S12 Architecture

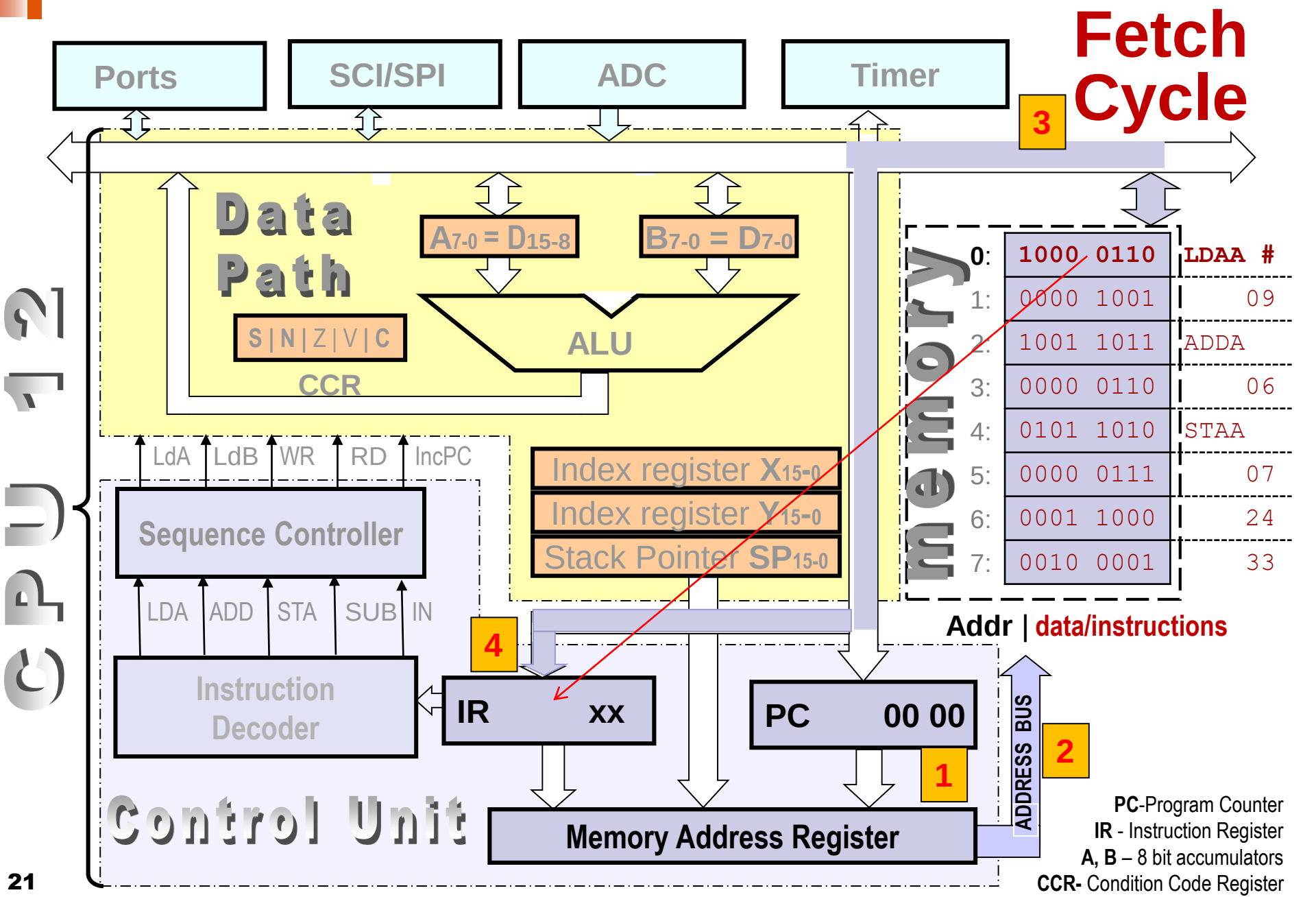


PC-Program Counter
IR - Instruction Register
A, B – 8 bit accumulators
CCR- Condition Code Register

Instruction Cycle – State Diagram

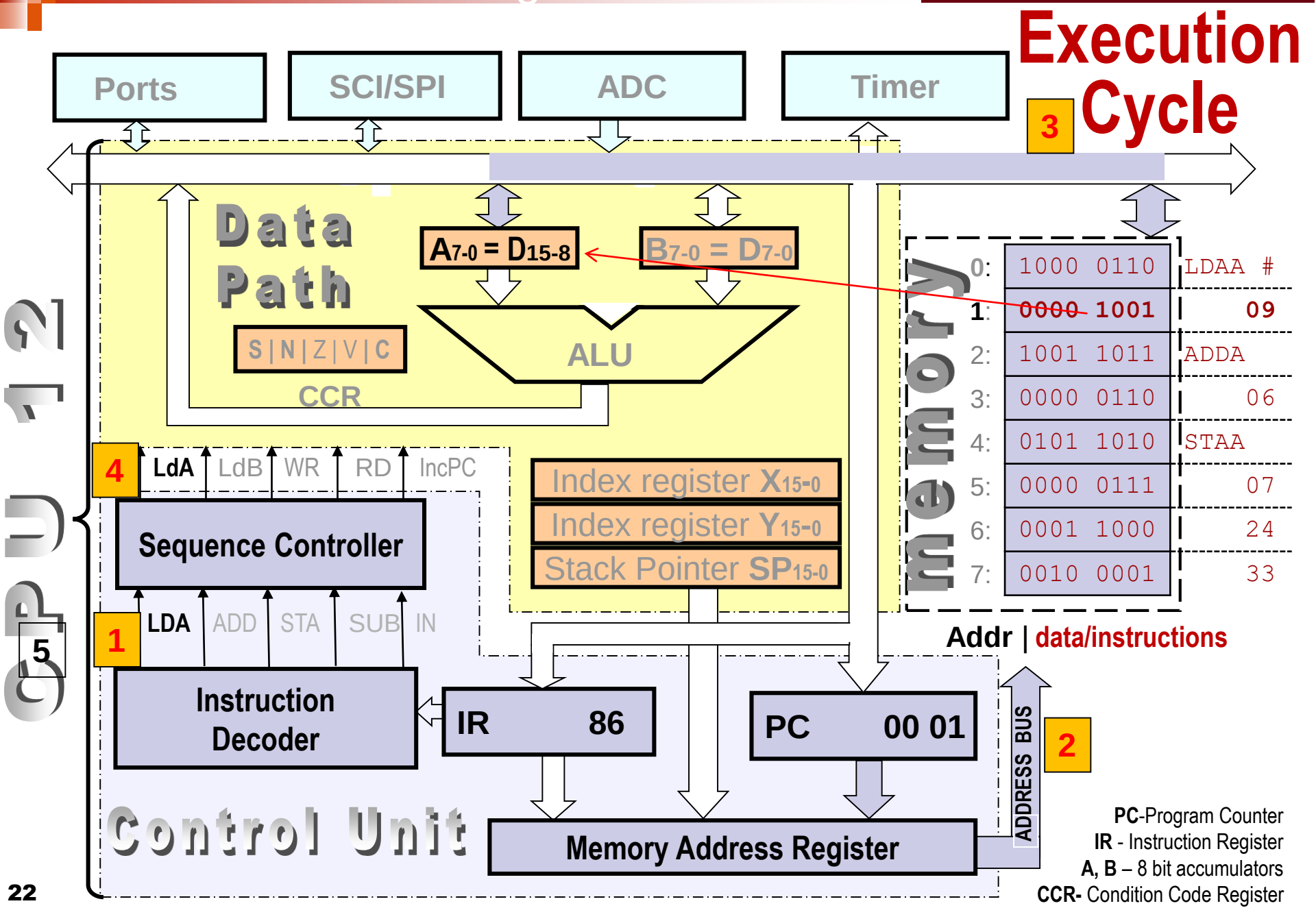


MC9S12 Block Diagram



MC9S12 Block Diagram

uOttawa



PC-Program Counter
IR - Instruction Register
A, B – 8 bit accumulators
CCR- Condition Code Register

Recall MC68HCS12 Addressing Modes

Immediate Addressing	opcode #opr8i opcode #opr16i
Direct Addressing	opcode opr8a
Extended Addressing	opcode opr16a
Indexed addressing with 5-bit, 9-bit or 16-bit signed offset	opcode oprx5,xysp opcode oprx9,xysp opcode oprx16,xysp
Indexed addressing with register offset	opcode abd,xysp
Indexed addressing with increment or decrement	opcode oprx3,-yxs+
16-bit or D register indexed-indirect addressing	opcode [oprx16,xysp] opcode [D,xysp]
Relative Addressing	opcode rel8 opcode rel16

Not all instructions can be employed in every mode!

Operand Syntax

- ***opr8i*** and ***opr16i***: 8-bit and 16-bit immediate data
- ***opr8a*** and ***opr16a***: 8-bit and 16-bit address
- ***oprx5***, ***oprx9*** and ***oprx16***: 5-bit, 9-bit, and 16-bit constant offset for indexed addressing
- ***oprx3***: 3-bit increment or decrement value
- ***xysp***: Either X, Y, SP, or PC register
- ***xys***: Either X, Y, or SP register
- ***abd***: Either A, B, or D register
- ***rel8, rel9*** and ***rel16***: 8-bit, 9-bit and 16 bit offset from PC for relative addressing
- In the reference guide, **oprx0_xysp**, represents,
  - oprx3, ± xys; oprx3,yxs±; oprx5,xysp; abd,xysp

Topics of discussion

- CISC and RISC Architectures
  - Reference: http//en.wikipedia.org/wiki/RISC
- M68HC12 Instruction Set
  - Instruction Set Categories
  - Move Data Categories
  - Modify Data Categories
  - Decision Making Categories
  - Flow Control Categories
  - Other Categories

Instruction Set Categories(18)

1. Move Data Categories
   - Load Registers
   - Store Registers
   - Transfer/Exchange Registers
   - Move Memory Contents
2. Modify Data Categories
   - Decrement/Increment
   - Clear/Set
   - Rotates/Shifts
   - Arithmetic
   - Logic
   - Condition Code

Instruction Set Categories

3. Decision Making Categories
   - Data Test
   - Conditional Branch
   - Branch if Bit Set or Clear
   - Loop Primitive
4. Flow Control Categories
   - Jump and Branch
   - Interrupt
5. Other Categories
   - Fuzzy Logic and Specialized Math Category (Will not study this category in this course)
   - Miscellaneous

Topics of discussion

- CISC and RISC Architectures
  - Reference: RISC – Wikipedia article (http://en.wikipedia.org/wiki/RISC)
- M68HC12 Instruction Set
  - Instruction Set Categories
    1. Move Data Categories
    2. Modify Data Categories
    3. Decision Making Categories
    4. Flow Control Categories
    5. Other Categories

Summary of Data Transfer Instructions

- Load Register Category
  - Transferring data from memory to a CPU register
  - Transferring data from the stack to a CPU register
  - Can also load effective addresses into X, Y, and SP registers
- Store Register Category
  - Transferring data from a CPU register to memory
  - Transferring data from a CPU register to the stack
- Transfer/Exchange Registers Categories
  - Transferring and Exchanging data between two registers
- Move Memory Category
  - Moving data from one memory location to another

Load Register Instructions

- First two instructions load 8 bit data into either accumulator
- Other instructions load 16 bit data
  - Note that order of bytes store in memory

LDAA	Load A	(M) ⇒ A
LDAB	Load B	(M) ⇒ B
LDD	Load D	(M : M + 1) ⇒ (A:B)
LDS	Load SP	(M : M + 1) ⇒ SP_H:SP_L
LDX	Load index register X	(M : M + 1) ⇒ X_H:X_L
LDY	Load index register Y	(M : M + 1) ⇒ Y_H:Y_L

Load Instruction Addressing Modes

Mnemonic	Operand Syntax					CCR			
	Immediate	Direct	Extended	Indexed	Indexed-Indirect	N	Z	V	C
LDAA, LDAB	#opr8i	opr8a	opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	⇳	⇳	0	-
LDD, LDX, LDY, LDS	#opr16i	opr8a	opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	⇳	⇳	0	-

Store Register Instructions

STAA	Store A	(A) ⇒ M
STAB	Store B	(B) ⇒ M
STD	Store D	(A) ⇒ M, (B) ⇒ M + 1
STS	Store SP	$(SP_H:SP_L) \Rightarrow M : M + 1$
STX	Store X	$(X_H:X_L) \Rightarrow M : M + 1$
STY	Store Y	$(Y_H:Y_L) \Rightarrow M : M + 1$

Mnemonic	Operand Syntax					CCR			
	Immediate	Direct	Extended	Indexed	Indexed-Indirect	N	Z	V	C
STAA, STAB, STD, STX, STY, STS		opr8a	opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	⇳	⇳	0	-

Load Effective Address

All ”load/store/transfer/move” instructions handle **data** specified by an *effective address*.
Only LEA(S,X,Y) handle *effective* ***addresses*** (not data), calculated in indexed addressing mode from the content of one of the registers S,X,Y.

LEAS	Load effective address into SP	Effective address ⇒ SP
LEAX	Load effective address into X	Effective address ⇒ X
LEAY	Load effective address into Y	Effective address ⇒ Y

Mnemonic	Operand Syntax					CCR			
	Immediate	Direct	Extended	Indexed	Indexed-Indirect	N	Z	V	C
LEAS, LEAX, LEAY				oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+		-	-	-	-

Example of Load Effective Address Instructions

- Assume X = $1234, Y = $1000 and SP = $0A00. Give the contents of each affected register after the following instructions are executed:

Instruction	Result
LEAX 10,X	X = X + 10_{10} = $1234 + $000A = $123E
LEAX $10,Y	X = Y + $10 = $1000 + $0010 = $1010
LEAS -10,SP	SP = SP - 10_{10} = $0A00 - $000A = $09F6

Use of the Stack Instructions

- Recall the use of the Stack as a temporary storage area for data
  - Save data from registers (often used within a subroutine to preserve the register contents)
  - Saving the return address when calling a subroutine
  - Allocating space for passing arguments to the subroutine
  - Allocating space for temporary variables in the subroutine
  - Preserving the CPU registers during an interrupt
  - Will study these techniques later
- The stack pointer must point to RAM and stack operations must be balanced
  - Need to initialize the stack pointer

STACK Push Instructions

- For pushing register contents onto the stack
  - Uses inherent addressing mode
  - No operands required
  - Does not affect the CCR bits
  - Note that SP is decremented **BEFORE** the transfer operation
  - Push operations are usually balanced with pull operations

PSHA	Push A	$(SP) - 1 \Rightarrow SP; (A) \Rightarrow M_{(SP)}$
PSHB	Push B	$(SP) - 1 \Rightarrow SP; (B) \Rightarrow M_{(SP)}$
PSHC	Push CCR	$(SP) - 1 \Rightarrow SP; (A) \Rightarrow M_{(SP)}$
PSHD	Push D	$(SP) - 2 \Rightarrow SP; (A : B) \Rightarrow M_{(SP)} : M_{(SP+1)}$
PSHX	Push X	$(SP) - 2 \Rightarrow SP; (X) \Rightarrow M_{(SP)} : M_{(SP+1)}$
PSHY	Push Y	$(SP) - 2 \Rightarrow SP; (Y) \Rightarrow M_{(SP)} : M_{(SP+1)}$

STACK Pull Instructions

- For pulling contents from the stack to a register
  - Uses inherent addressing mode
  - No operands required
  - Does not affect the CCR bits
  - Note that the SP is incremented **AFTER** the transfer operation
  - Pull operations are balanced with push operations
  - Pulls must be in the reverse order of pushes

PULA	Pull A	$(M_{(SP)}) \Rightarrow A; (SP) + 1 \Rightarrow SP$
PULB	Pull B	$(M_{(SP)}) \Rightarrow B; (SP) + 1 \Rightarrow SP$
PULC	Pull CCR	$(M_{(SP)}) \Rightarrow CCR; (SP) + 1 \Rightarrow SP$
PULD	Pull D	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow A : B; (SP) + 2 \Rightarrow SP$
PULX	Pull X	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow X; (SP) + 2 \Rightarrow SP$
PULY	Pull Y	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y; (SP) + 2 \Rightarrow SP$

Example of PSH and PUL instructions

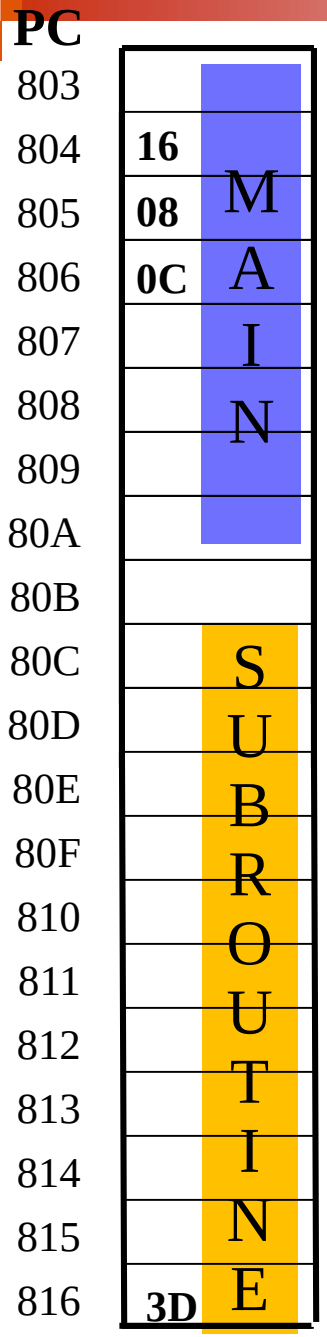
- Assembler ASM12 V1.22 Build

```
1:    =00000A00     STACK:    EQU $0a00    ; Equate the stack pointer
2:                                          ; initialization value.
3:     0000 CF 0A00           lds    #STACK  ; Init the stack pointer.
4:                      ;        ---
5:     0003 36                psha          ; Put the A register on stack
6:     0004 34                pshx          ; Put the X register on stack
7:                      ;        ---
8:     0005 30                pulx          ; Must pull the data in the
9:     0006 32                pula          ; reverse order
```

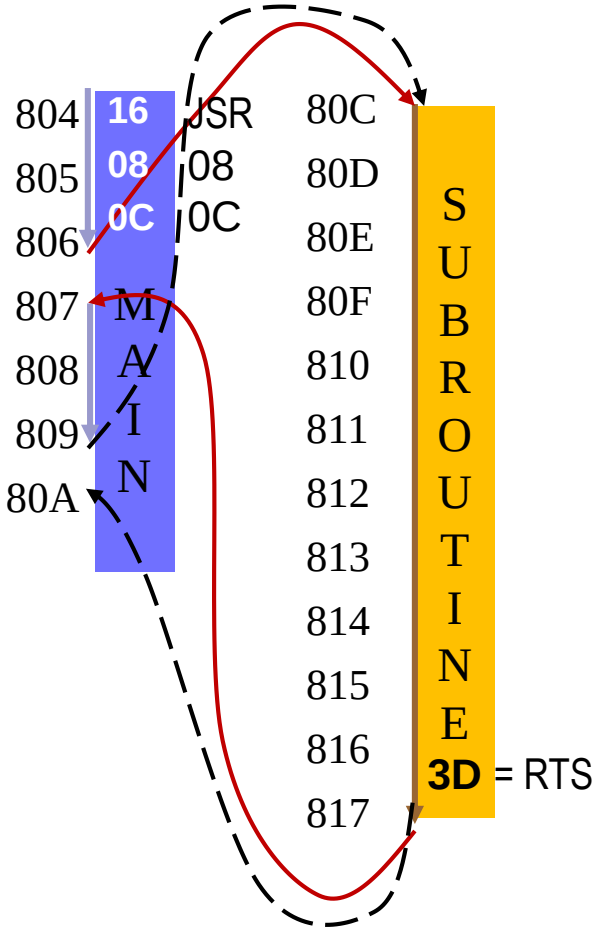
The subroutine

- A sequence of instructions that can be called from various points in a program
- Allows the same operation with different parameters
- Simplifies programming using modular structure programming (will study more closely later).

HOW DOES IT WORK?

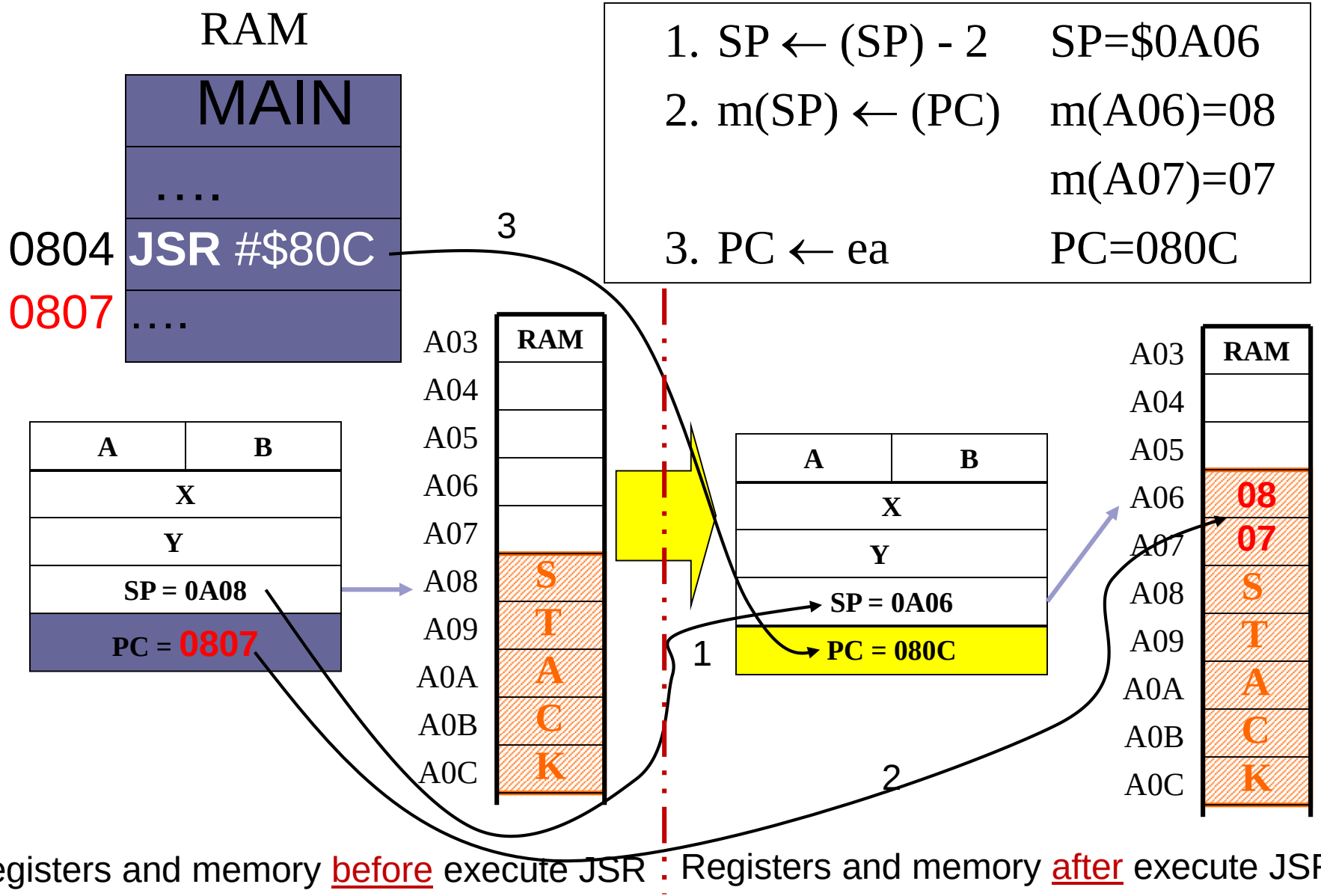


- A subroutine is a program module that is independent of the main program
- To use it, the main program transfers control to the subroutine
- The subroutine performs its function and then returns control to the main program



uOttawa

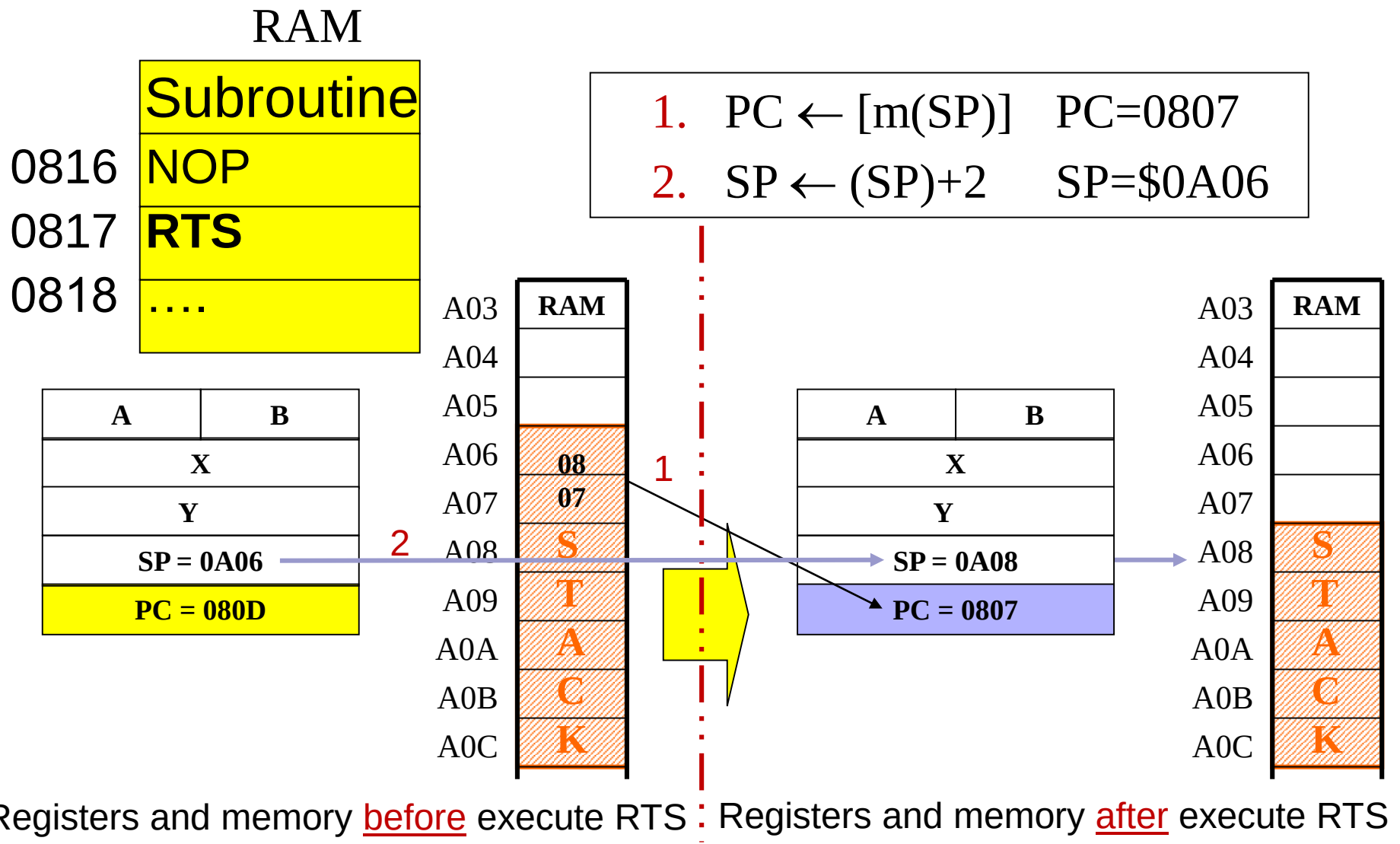
MAIN CALLS SUBROUTINE



Registers and memory before execute JSR

Registers and memory after execute JSR

SUBROUTINE RETURNS CONTROL TO MAIN



Subroutines and the Return Address

- Instructions “JSR SUB” or “BSR SUB” pushes the PC onto the stack
  - Effect is to provide the address of the instruction following the JSR or BSR
- RTS Instruction pulls the return address into the PC
  - What does this mean if stack operations are used in the subroutine?

Subroutine Example

- What is wrong with the following subroutine?

```
SUB:  pshx    ; Save the registers
      psha
      ;       - - -
      ;       - - -
      pshb    ; Temp save some data
      ;       - - -
      ;       - - -
      pula    ; Restore the registers
      pulx
      rts
```

Transfer and Exchange Register Instructions (BACK to Move Data Categories)

TAB	Transfer A to B	(A) ⇒ B
TBA	Transfer B to A	(B) ⇒ A
TFR	Transfer register to register	(A, B, CCR, D, X, Y, or SP) ⇒ A, B, CCR, D, X, Y, or SP
EXG	Exchange register to register	(A, B, CCR, D, X, Y, or SP) ⇔ (A, B, CCR, D, X, Y, or SP)

Mnemonic	**Operand Syntax**	**CCR**			
	Inherent	**N**	**Z**	**V**	**C**
TAB, TBA		⇕	⇕	0	-
TAP		⇕	⇕	⇕	⇕
TFR, EXG	(abd or ccr or xys), (abd or ccr or xys)	-	-	-	- *

* When the CCR is the destination register, all flags will be set accordingly

Transfer and Exchange Register Instructions - Notes

- When transferring (TFR) contents of 8-bit register to 16-bit registers, sign is extended
  - Can use alternative mnemonic SEX
  - For example, SEX A,X is the same as TFR A,X
- When exchanging the contents between an 8-bit register and a 16-bit registers:
  - Exchange low order byte of 16-bit register with 8-bit register
  - High order byte of 16-bit register set to $00
- When transferring contents of 16-bit register to an 8-bit register, only lower order byte is transferred

MC68HCS12 Transfer Register Mnemonics

HCS12 Operation	Opcode	Symbolic Operation	MC68HC12 Instruction
Transfer SP to X	TSX	(SP) ⇨ X	TFR SP,X
Transfer SP to Y	TSY	(SP) ⇨ Y	TFR SP,Y
Transfer X to SP	TXS	(X) ⇨ SP	TFR X,SP
Transfer Y to SP	TYS	(Y) ⇨ SP	TFR Y,SP
Exchange D and X	XGDX	(D) ⇔ (X)	EXG D,X
Exchange D and Y	XGDY	(D) ⇔ (Y)	EXG D,Y

Transfer Instructions - Examples

- What is in the A, B, NZVC bits after the following sequence is executed?

```
LDAA    #$AA
TAB
```

- What is in the stack pointer and the Y register after the following sequence is executed?

```
LDX     #$1234
TXS
TSY
```

Move Instructions

- Transfer content of a memory location to another memory location
- Important with CPU with small register set
- See Example 7-14 - Reversing the order of data in a 100-byte table

MOVB	Move byte (8-bit)	$(M_1) \Rightarrow M_2$
MOVW	Move word (16-bit)	$(M : M+1_1) \Rightarrow M : M+1_2$

Mnemonic	Operand Syntax					
	Imm → Ext	**Imm → Ind**	**Ext → Ext**	**Ext → Ind**	**Ind → Ext**	**Ind → Ind**
MOVB	#opr8i, opr16a	#opr8i, oprx0_xysp	Opr16a, opr16a	opr16a, oprx0_xysp	oprx0_xysp, opr16a	oprx0_xysp, oprx0_xysp
MOVW	#opr16i, opr16a	#opr16i, oprx0_xysp	Opr16a, opr16a	opr16a, oprx0_xysp	oprx0_xysp, opr16a	oprx0_xysp, oprx0_xysp

oprx0_xysp represents one of the following indexing modes: *oprx5,xysp* *oprx3,-xys+* *abd,xysp*

COPY 10 elements of a vector SOURC from address A000 to the vector COPY at address A100

```
 1:        =0000A000          SOURC   EQU $A000
 2:        =0000A100          COPY    EQU $A100
 3:        =0000000A          NBELEM  EQU 10
 4:        =00000800                  ORG    $800
 5:    0800 CE A000                   LDX    #SOURC
 6:    0803 CD A100                   LDY    #COPY
 7:    0806 C6 0A                     LDAB   #NBELEM
 8:    0808 180A 30 70         LOOP   MOVB 1,X+,1,Y+
 9:    080C 53                        DECB
10:    080D 26 F9                     BNE    LOOP
11:    080F 3F                        SWI
```

Symbols:

copy	*0000a100
loop	*00000808
nbelem	*0000000a
sourc	*0000a000

source address	X
copy address	Y
counter	B

Topics of discussion

- CISC and RISC Architectures
  - Reference: RISC – Wikipedia article (http://en.wikipedia.org/wiki/RISC)
- M68HC12 Instruction Set
  - Instruction Set Categories
  - Move Data Categories
  - Modify Data Categories
  - Decision Making Categories
  - Flow Control Categories
  - Other Categories

Modify Data Categories

- Decrement/Increment
  - Decrement/Increment registers by 1
- Clear/Set
  - Clear memory and registers
  - Bit set and clear memory and registers
- Rotates/Shifts
  - Moving data 1 bit position in memory and registers
- Arithmetic
  - 8 and 16 bit addition and subtraction, and decimal adjust
  - 8 and 16 bit multiply and division (both signed and unsigned)
  - Negation
- Logic
  - AND, OR, XOR, and 1's-complement operations
- Condition Codes
  - Operations for manipulating the bits in the CCR

Decrement/Increment Instructions

Decrement Instructions		
DEC	Decrement memory	(M) – $01 ⇒ M
DECA	Decrement A	(A) – $01 ⇒ A
DECB	Decrement B	(B) – $01 ⇒ B
DES	Decrement SP	(SP) – $0001 ⇒ SP
DEX	Decrement X	(X) – $0001 ⇒ X
DEY	Decrement Y	(Y) – $0001 ⇒ Y
Increment Instructions		
INC	Increment memory	(M) + $01 ⇒ M
INCA	Increment A	(A) + $01 ⇒ A
INCB	Increment B	(B) + $01 ⇒ B
INS	Increment SP	(SP) + $0001 ⇒ SP
INX	Increment X	(X) + $0001 ⇒ X
INY	Increment Y	(Y) + $0001 ⇒ Y

Decrement/Increment Instructions

Mnemonic	Operand Syntax			CCR			
	Extended	Indexed	Indexed-Indirect	N	Z	V	C
INC,DEC	opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	⇳	⇳	⇳	-
DECA, DECB, INCA, INCB				⇳	⇳	⇳	-
DEX, DEY, INX, INY				-	⇳	-	-

Increment and Decrement

- Carry bit not affected to allow use of instruction with multi-precision arithmetic
- DES and INS translate to LEAS -1,SP and LEAS 1,SP respectively
- Example – Using memory to implement a counter

```
1:        =0000001A          COUNT:  EQU     26
2:                           ; Initialize the counter in memory
3:     0000 180B 1A 000A              movb    #COUNT,Counter
4:     0005                  LOOP:
5:                           ;              - - -
6:                           ;              - - -
7:     0005 73 000A                    dec     Counter
8:     0008 26 FB                      bne     LOOP
9:                           ;              - - -
10:     000A +0001           Counter:  DS      1       ; 8-bit counter
```

Clear and Set Instructions

- Clear instruction zeros the register or memory location
- BCLR and BSET can clear/set individual bits in memory
  - Uses an 8-bit mask
  - Note the operation of the BCLR, how do you interpret the bits in the mask?
  - Useful for setting bits of the output ports to control peripherals – see the LED example in Figure 7-4.

CLR	Clear memory	$00 ⇒ M
CLRA	Clear A	$00 ⇒ A
CLRB	Clear B	$00 ⇒ B

BCLR	Clear bits in memory	(M) • ($\overline{mm}$) ⇒ M
BSET	Set bits in memory	(M) + (mm) ⇒ M

Clear and Set Instructions

Mnemonic	Operand Syntax				CCR			
	Direct	Extended	Indexed	Indexed-Indirect	N	Z	V	C
CLR		opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	0	1	0	0
BSET, BCLR	opr8a,msk8	opr16a,msk8	oprx5,xysp,msk8 oprx9,xysp,msk8 oprx16,xysp,msk8 abd,xysp.msk8 oprx3,-xys+,msk8		⇳	⇳	0	-
CLRA, CLRB					0	1	0	0

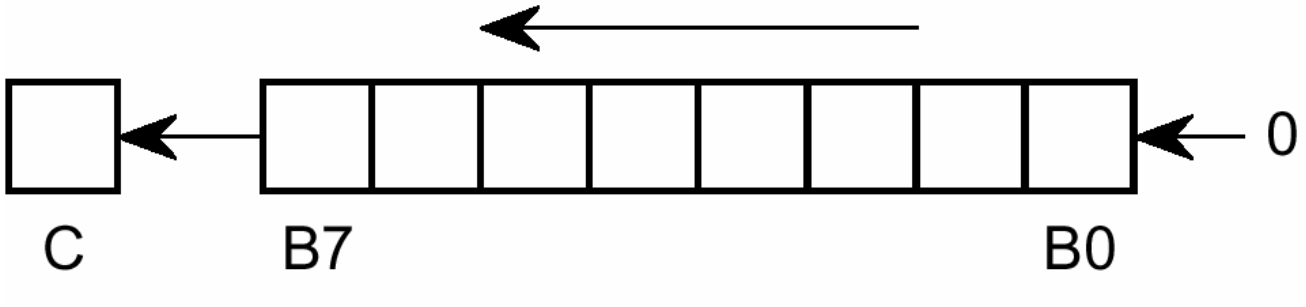
The operand msk8 is an 8-bit mask.

Shift and Rotate Instructions

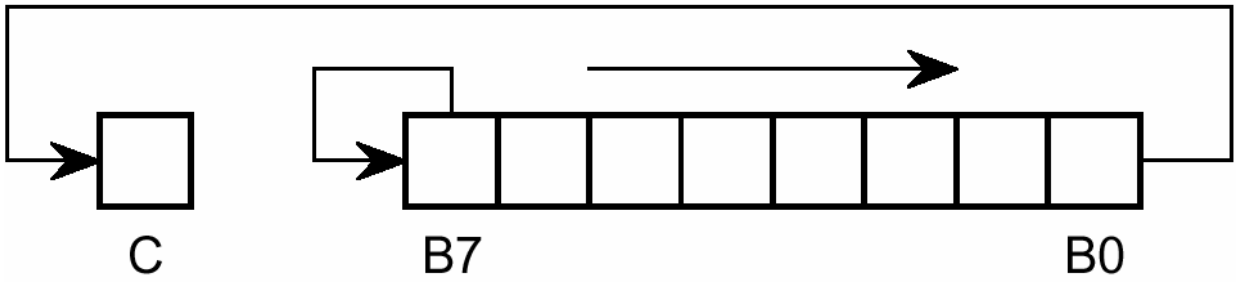
- Two types of shift instructions
  - Arithmetic
    - Shift Left: ASL, ASLA, ASLB, ASLD
    - Shift Right: ASR, ASRA, ASRB, ASRD
  - Logical
    - Shift Left: LSL, LSLA, LSLB, LSLD (assembled as arithmetic shift left)
    - Shift Right: LSR, LSRA, LSRB, LSRD
- Rotate instructions
  - Rotate Left: ROL, ROLA, ROLB
  - Rotate Right: ROR, RORA, RORB
- All instructions affect all bits in the condition code register
  - Carry bit has data shifted into it

Arithmetic Shift Instructions

- Arithmetic Shift Left

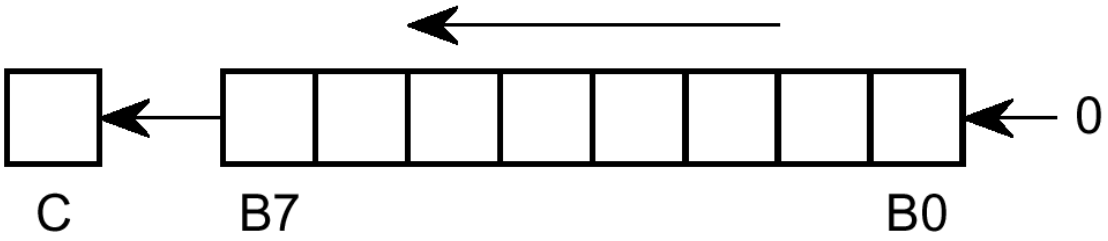


- Arithmetic Shift Right

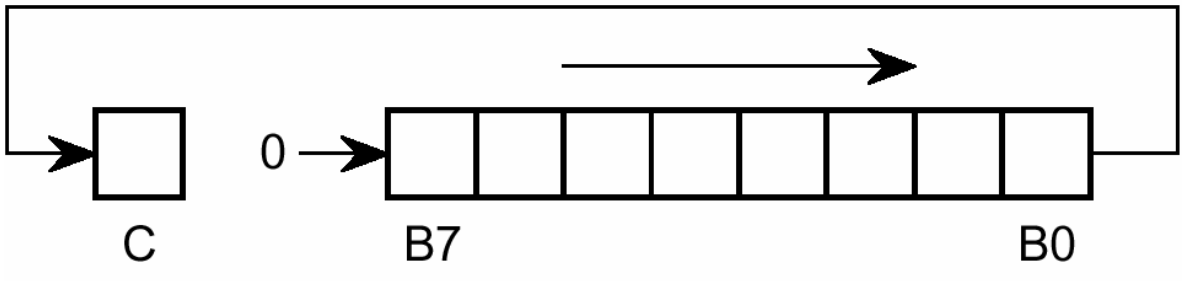


Logical Shift Instructions

- Logical Shift Left

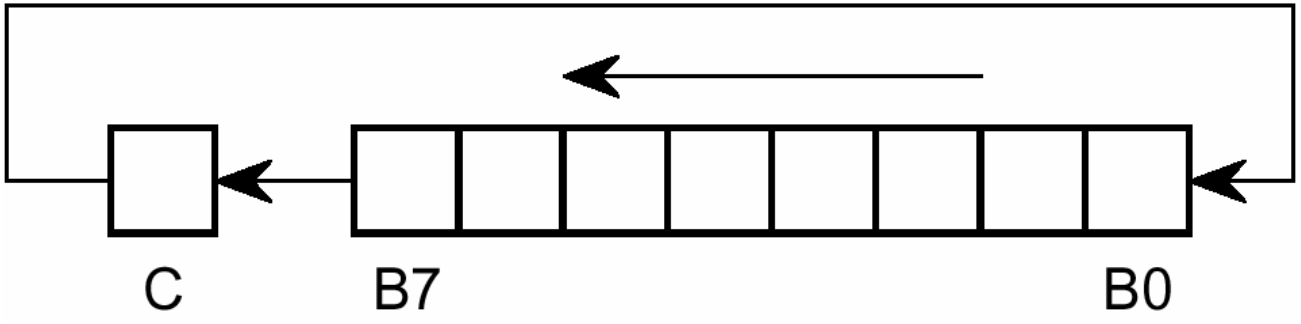


- Logical Shift Right

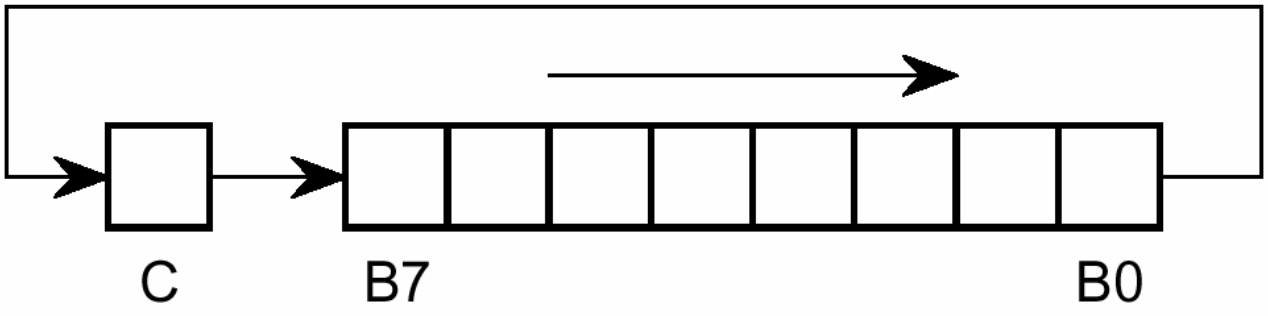


Rotate Instructions

- Rotate Left

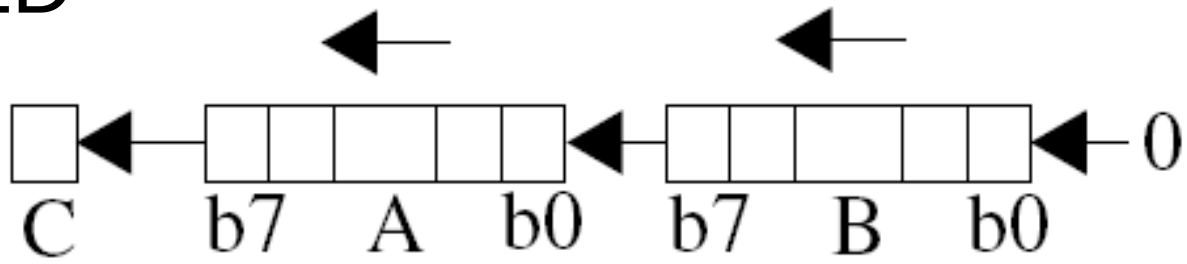


- Rotate Right

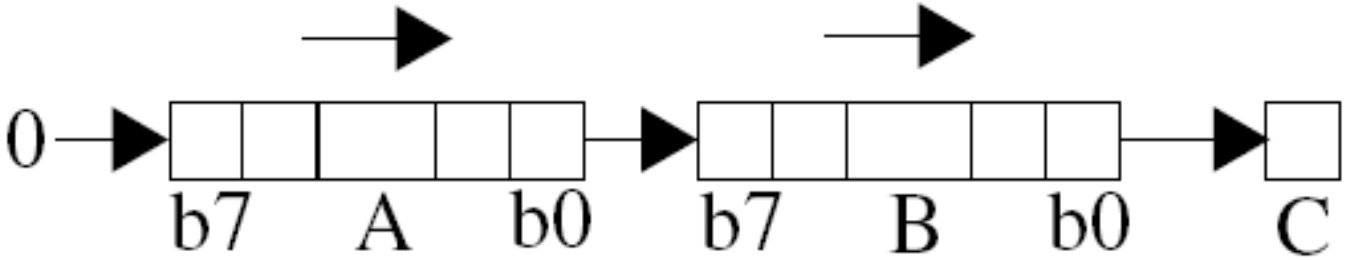


16-bit Shift Instructions

- ASLD



- LSRD



Operands for ASL, ASR, LSL, LSR, ROL, ROR

Mnemonic	Operand Syntax				CCR			
	Direct	Extended	Indexed	Indexed-Indirect	N	Z	V	C
ASL, ASR, ROL, ROR		opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	⇳	⇳	⇳	⇳
LSL, LSR		opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	0	⇳	⇳	⇳

Examples of Shift and Rotate Operations

Assume the A value is $A9. What is the result of each of the following instructions? ASLA, ASRA, LSLA, LSRA, ROLA, RORA.

		After		
Before		C	A Register	Comments
10101001	ASLA	1	01010010	Zero shifted into bit-0
10101001	ASRA	1	11010100	Sign bit is preserved
10101001	LSLA			
10101001	LSRA			
10101001	ROLA			
10101001	RORA			

Multiplying with Shift Instructions

Multiply by 10 a multiplicand passed through register D:

```
1:    0000 5C 07          std     TEMP; Save multiplicand @ location TEMP
2:    0002 59             asld          ; x 2
3:    0003 59             asld          ; x 2 again = x 4
4:    0004 D3 07          addd  TEMP ; Add the original.  Now it's x 5
5:    0006 59             asld          ; x 2 = x 10
6:
7:    0007 +0002   TEMP:   DS      2      ; Temp storage
```

Arithmetic Instructions

- All arithmetic operations are executed in the accumulator D (or parts of it A or B), except multiplication and division which may also use index registers X and Y
- Add and Subtract
  - 8-bit add/subtract: ABA, ADDA, ADDB, SBA, SUBA, SUBB
    - With the carry bit: ADCA ADCB SBCA SBCB
  - 16-bit add/subtract: ADDD, SUBD
    - LEAX, LEAY, LEAS can be view as 16-bit additions
- Decimal Arithmetic: DAA (for handling BCD codes)
- Negating and sign extension: NEG, NEGA, NEGB, SEX
- Multiplication
  - 8-bit unsigned multiply: MUL
  - 16-bit signed and unsigned multiply: EMUL, EMULS
- Division
  - Unsigned and signed 32/16-bit division: EDIV, EDIVS
  - Unsigned and signed 16/16-bit division: IDIV, IDIVS
  - Fractional division: FDIV

Add and Subtract Instructions

ABA	Add B to A	(A) + (B) ⇒ A
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 to D	(A:B) + (M : M + 1) ⇒ A : B
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 (A:B)	(D) – (M : M + 1) ⇒ D

Load effective address (LEAS, LEAX, and LEAY) instructions could also be considered as specialized addition and subtraction instructions.

Add/Subtract Operands

- All arithmetic operations involves the accumulator D (or parts of it A or B) the accumulator
- 8-bit operations also sets the H bit

Mnemonic	Operand Syntax					CCR			
	Imm	Direct	Ext	Indexed	Indexed-Indirect	N	Z	V	C
ADDA, ADDB, SUBA, SUBB, ADCA, ADCB, SBCA, SBCB, ADDD, SUBD	#opr8i	opr8a	opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	⇳	⇳	⇳	⇳

Multi-Precision Adding

```
1:                      ; Add the least significant bytes first
2:    0000 96 0D          ldaa   DATA1+1 ; Get least sig byte of
3:                                       ; 16-bit DATA1
4:    0002 9B 0F          adda   DATA2+1 ; Add in the least sig byte
5:                                       ; of 16-bit DATA2
6:    0004 5A 11          staa   DATA3+1 ; Save it
7:                      ; The carry bit now has a carry out of the least
8:                      ; significant byte that must be
9:                      ; added in to the most significant byte addition.
10:                      ; Note that STAA does not change the carry bit.
11:    0006 96 0C         ldaa   DATA1   ; ldaa does not affect
12:                                      ; the carry bit
13:    0008 99 0E         adca   DATA2   ; Add the most significant
14:                                      ; byte plus the carry
15:    000A 5A 10         staa   DATA3
16:
17:    000C +0002            DATA1:  DS     2      ; 16-bit Storage areas
18:    000E +0002            DATA2:  DS     2
19:    0010 +0002            DATA3:  DS     2
```

Decimal Arithmetic

- DAA instruction adjusts the binary add results for BCD numbers
  - ABA, ADDA, ADCA sets the H bit in the CCR and leaves the results of a binary addition in accumulator A
  - If the operands of the addition were BCD (Binary Coded Decimal) values, the result is incorrect
  - DAA will correct the binary result in A

Decimal	BCD Code	
34	0011 0100	
29	0010 1001	
	0101 1101	(not a BCD code)
	0000 0110	Correction added by DAA
63	0110 0011	

Negating

NEG	Two's complement memory	$00 – (M) ⇒ M or $(\overline{M}) + 1 ⇒ M$
NEGA	Two's complement A	$00 – (A) ⇒ A or $(\overline{A}) + 1 ⇒ A$
NEGB	Two's complement B	$00 – (B) ⇒ B or $(\overline{B}) + 1 ⇒ B$

Mnemonic	Operand Syntax					CCR			
	Imm	Direct	Ext	Indexed	Indexed-Indirect	N	Z	V	C
NEG			opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	⇳	⇳	⇳	⇳
NEGA, NEGB						⇳	⇳	⇳	⇳

Example of Negating

- Assume A contains the following data before the M68HC12 executes the NEGA instruction. What is the result in A and the N, Z, V and C bits for the negation of each byte?
  - A = $00, $7F, $01, $FF, $80

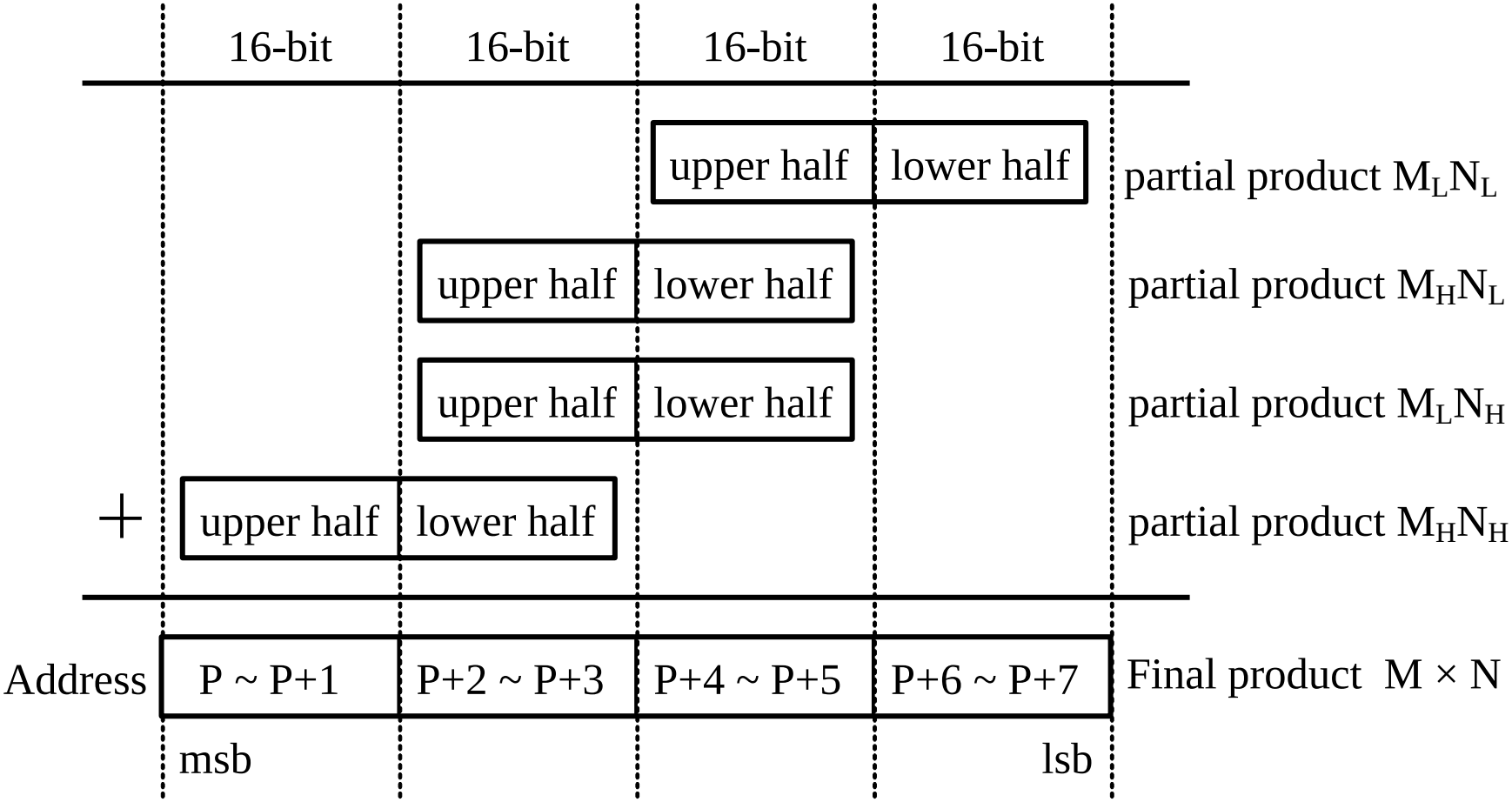
	After		
Before	A Register	N Z C V	Comment
$00	$00	0 1 0 0	Negating zero gives us zero
$7F	$81		
$01	$FF		
$FF	$01		
$80	$80		

Multiplication

- Carry bit can be used for rounding
  - MUL sets C=1 when the bit 7 of the results = 1
  - EMUL, and EMULS sets C=1 when bit 15 of results = 1
- EMUL and EMULS also set the N and Z bits in the CCR
- Note there is no 8 bit signed multiply instructions
  - How can we perform an 8-bit signed multiply?

EMUL	16 by 16 multiply (unsigned)	(D) × (Y) ⇒ Y : D
EMULS	16 by 16 multiply (signed)	(D) × (Y) ⇒ Y : D
MUL	8 by 8 multiply (unsigned)	(A) × (B) ⇒ A : B

32-bit Multiplication



Division

EDIV	32 by 16 divide (unsigned)	(Y : D) ÷ (X) ⇒ Y Remainder ⇒ D
EDIVS	32 by 16 divide (signed)	(Y : D) ÷ (X) ⇒ Y Remainder ⇒ D
FDIV	16 by 16 fractional divide	(D) ÷ (X) ⇒ X Remainder ⇒ D
IDIV	16 by 16 integer divide (unsigned)	(D) ÷ (X) ⇒ X Remainder ⇒ D
IDIVS	16 by 16 integer divide (signed)	(D) ÷ (X) ⇒ X Remainder ⇒ D

Division

- Signed divide instructions:
  - EDIV**S** = 32-bit Extended Divide 32-Bit by 16-Bit: (Y : D) ÷ (X) ⇒ Y; Remainder ⇒ D
  - IDIV**S** = 16-bit - Integer Divide: (D) ÷ (X) ⇒ X; Remainder ⇒ D
  - If divide by 0, then C is set to 1
  - Produces a quotient & a remainder
- EDIV, IDIV are unsigned operations
  - If divide by 0, then C is set to 1
  - In the case of IDIV, X (quotient) will contain $FFFF
  - Produces a quotient and a remainder
- FDIV provides unsigned fractional division, numerator is assumed to be less than the denominator
  - If divide by 0, then C is set to 1
  - V is set to 1 when numerator larger than denominator
  - Radix in quotient is to the right of the MSB when radix is in the same positions for the numerator and denominator
  - FDIV corresponds to the 32bit/16bit division operation: numerator * 2^{16} / denominator (i.e. the EDIV)

IDIV Example

- Assume D contains 176_{10} and X = 10_{10}. What is in A, B and X before and after an IDIV instruction?

- *Solution:*

(D) ÷ (X) ⇒ X; Remainder ⇒ D

Before the IDIV instruction:

D = 176_{10} = $00B0, therefore A = $00, B = $B0

X = 10_{10} = $000A

After the IDIV instruction:

$176_{10}/10_{10} = 17_{10}$ with a remainder of 6_{10}, therefore

X = 17_{10} = $0011, D = 6_{10} = $0006, i.e., A = $00, B = $06.

Fractional Arithmetic

```
 0.50     .1000     0.75   .1100
+0.25    +.0100    -0.25  -.0100
 0.75     .1100     0.50   .1000
```

```
  0.50    .1000
x 0.25   x.0100
  0.125    0000
           0000
          1000
           0000
         .00100000
```

0.375/0.5 = 0.75

```
.0110
.1000
```

```
          .1100
1000|0110.0000
```

FDIV Example

- Assume D contains 100_{10} and X = 300_{10}. What is in A, B and X before and after an FDIV instruction?

Solution:

(D) ÷ (X) ⇒ X; Remainder ⇒ D

Before the FDIV instruction:

D = 100_{10} = $0064, therefore A = $00, B = $64

X = 300_{10} = $012C

After the FDIV instruction:

$100_{10}/300_{10} = 0.333328247070_{10}$ with a remainder of 0000, therefore

X = 0.33332838 = $5555, D = $0064, A = $00, B = $64.

Rounding with Fractional Arithmetic

- The MUL instruction produces a 16 bit result
  - To round to an 8-bit result (ignore the least significant byte) the Carry bit can be used
  - C bit set to value of bit 7, thus can use the ADCA #0 to round the most significant byte
- The carry bit can also be used to round the EMUL and EMULS results

Logic Instructions

ANDA	AND A with memory	(A) • (M) ⇒ A
ANDB	AND B with memory	(B) • (M) ⇒ B
ANDCC	AND CCR with memory (clear CCR bits)	(CCR) • (M) ⇒ CCR
EORA	Exclusive OR A with memory	(A) ⊕ (M) ⇒ A
EORB	Exclusive OR B with memory	(B) ⊕ (M) ⇒ B
ORAA	OR A with memory	(A) + (M) ⇒ A
ORAB	OR B with memory	(B) + (M) ⇒ B
ORCC	OR CCR with memory (set CCR bits)	(CCR) + (M) ⇒ CCR

COM	One's complement memory	$FF – (M) ⇒ M or $(\overline{M}) \Rightarrow M$
COMA	One's complement A	$FF – (A) ⇒ A or $(\overline{A}) \Rightarrow A$
COMB	One's complement B	$FF – (B) ⇒ B or $(\overline{B}) \Rightarrow B$

Converting BCD to ASCII

```
LS_MASK: EQU          %00001111 ; Least sig nibble mask
;
         tfr      a,b      ; Save the BCD number in B
; Need to print the most significant nibble first
         lsra              ; Shift 4 bits to right
         lsra
         lsra
         lsra
         oraa     #$30     ; Convert to ASCII
         jsr      PRINT    ; Go print it
         tfr      b,a      ; Get the original back
         anda     #LS_MASK ; Set most sig bits to 0
         oraa     #$30     ; Convert to ASCII
         jsr      PRINT    ; Print it
;        - - -
; Dummy subroutine
PRINT: rts
```

Condition Code Instructions

ANDCC	AND CCR with memory (clear CCR bits)	(CCR) • (M) ⇒ CCI
ORCC	OR CCR with memory (set CCR bits)	(CCR) + (M) ⇒ CC
CLC	Clear C bit in CCR	0 ⇒ C
CLV	Clear V bit in CCR	0 ⇒ V
SEC	Set C bit	1 ⇒ C
SEV	Set V bit	1 ⇒ V

- Have already seen ANDCC and ORCC
- CLC translates to ANDCC #$FE
- CLV translates to ANDCC #$FD
- SEC translates to ORCC #$01
- SEV translates to ORCC #$02

Topics of discussion

- CISC and RISC Architectures
  - Reference: RISC – Wikipedia article (http://en.wikipedia.org/wiki/RISC)
- M68HC12 Instruction Set
  - Instruction Set Categories
  - Move Data Categories
  - Modify Data Categories
  - Decision Making Categories
  - Flow Control Categories
  - Other Categories

Decision Making Categories

- Category - Data Test
  - Instructions used to test registers and memory – only CCR bits are changed
- Category - Conditional Branch
  - Branching according to how condition code bits are set
- Category - Branch if Bit Set or Clear
  - Branching based on bits set in memory byte
- Category - Loop Primitive
  - Complex instruction that modifies a register and branches based on result of modification

Data Test Instructions

- These instructions do not affect the contents of registers but simply modify the CCR bits according to corresponding operation.

BITA	Bit test A	(A) • (M)
BITB	Bit test B	(B) • (M)
CBA	Compare A to B	(A) – (B)
CMPA	Compare A to memory	(A) – (M)
CMPB	Compare B to memory	(B) – (M)
CPD	Compare D to memory (16-bit)	(A : B) – (M : M + 1)
CPS	Compare SP to memory (16-bit)	(SP) – (M : M + 1)
CPX	Compare X to memory (16-bit)	(X) – (M : M + 1)
CPY	Compare Y to memory (16-bit)	(Y) – (M : M + 1)
TST	Test memory for zero or minus	(M) – $00
TSTA	Test A for zero or minus	(A) – $00
TSTB	Test B for zero or minus	(B) – $00

Data Test Instructions

Mnemonic	Operand Syntax					CCR			
	Imm	Direct	Ext	Indexed	Indexed-Indirect	N	Z	V	C
BITA, BITB	#opr8i	opr8a	opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	⇳	⇳	0	-
CBA						⇳	⇳	⇳	⇳
CMPA, CMPB, CPD, CPX, CPY, CPS	#opr8i	opr8a	opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	⇳	⇳	⇳	⇳
TST			opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]	⇳	⇳	0	0
TSTA, TSTB						⇳	⇳	0	0

Example of Data Test Instruction

```
1:          =00000002    BIT_1:  EQU     %00000010 ; Mask for Bit-1
2:          =00000024    PORTH:  EQU     $24     ; Offset to Port H
3:                       ;               - - -
4:                       ; IF BIT_1 is zero
5:      0000 86 02                       ldaa    #BIT_1
6:      0002 95 24                       bita    PORTH          ; Test Bit-1, Port H
7:      0004 26 02                       bne     do_if_one      ; Do the one part
8:                       ; THEN
9:                       ; This is the code to do if the bit is a zero
10:                        ;               - - -
11:     0006 20 00                       bra     end_if         ; Skip the next part
12:     0008               do_if_one:
13:                        ; This is the code to do if the bit is a one
14:                        ;               - - -
15:     0008               end_if:
```

Conditional Branch Instructions

- Branch Instructions have the format:
  - Short Branches: OPCODE rel8
    - Offsets range from -128 ($80) to 127 ($7F)
  - Long Branches: OPCODE rel16
    - Offset ranges from -32,768 ($8000) to 32,767 ($7FFF)
  - Does not affect the CCR bits
- Branches taken when a condition is true
  - See following slides
- If branch is taken, offset is added to PC value with address following the branch instruction
  - Recall previous example (data test instruction)

Simple Branching

Mnemonic Long Branch	Short Branch	Operation	Condition
LBCC	BCC	Branch if carry clear	C = 0
LBCS	BCS	Branch if carry set	C = 1
LBEQ	BEQ	Branch if equal	Z = 1
LBMI	BMI	Branch if minus	N = 1
LBNE	BNE	Branch if not equal	Z = 0
LBPL	BPL	Branch if plus	N = 0
LBVC	BVC	Branch if overflow clear	V = 0
LBVS	BVS	Branch if overflow set	V = 1

Branching useful after instructions that affect the specific condition code

Example: BCS, BCC: after CMPA, CMPB, etc.

BMI, BPL, BEQ, BNE: after TSTA, TSTB, etc.

Unsigned and Signed Branching

- Unsigned

LBHI	BHI	Branch if higher	R > M	C + Z = 0
LBHS	BHS	Branch if higher or same	R ≥ M	C = 0
LBLO	BLO	Branch if lower	R < M	C = 1
LBLS	BLS	Branch if lower or same	R ≤ M	C + Z = 1

- Signed

LBGE	BGE	Branch if greater than or equal	R ≥ M	N ⊕ V = 0
LBGT	BGT	Branch if greater than	R > M	Z + (N ⊕ V) = 0
LBLE	BLE	Branch if less than or equal	R ≤ M	Z + (N ⊕ V) = 1
LBLT	BLT	Branch if less than	R < M	N ⊕ V = 1

Using long branches

```
 1:                          ;           - - -
 2:    0000 81 FF                      cmpa    #$FF     ; Compare and Set CCR
 3:
 4:    0002 1826 00FE                  lbne    DO_NOT_EQUAL
 5:    0006                      DO_EQUAL:
 6:                            ; This is the code for the DO IF EQUAL part.  The
 7:                            ; DS 250 simulates more than 127 bytes of code.
 8:    0006 +00FA                       DS      250
 9:                            ;
10:    0100 1820 00FA                  lbra    OVER_NEXT
11:    0104                      DO_NOT_EQUAL:
12:                             ; This is the code to be done if A does not equal
13:                             ; $FF.  The DS 250 simulates more than 127 bytes
14:                             ; of code.
15:    0104 +00FA                       DS      250
16:    01FE                      OVER_NEXT:
```

Branch if Bit Set or Bit Clear

BRCLR	Branch if selected bits clear	(M) • (mm) = 0
BRSET	Branch if selected bits set	$(\overline{M})$ • (mm) = 0

- A mask, *mm.* is supplied in the instruction
  - E.g. **BRSET opr8, msk8, rel8**
  - opr8 gives the direct address of M
  - msk8 is the mask mm
  - rel8 gives the relative address for branching

Example 7-42:

```
BIT_7:     EQU   %1000000  ; mask for bit 7
PORTT:   EQU    $240         ; address for port T
 ..............
wait_for _7:  brclr  PORTT,BIT_7, wait_for_7 ;while bit 7 is zero, wait for it to become one
```

Loop Primitive Instructions

- Decrement/Increment any of the registers and branch if condition is TRUE
- Instruction format: OPCODE abdxys,rel9
  - A three byte instruction that includes a 9-bit offset (ranges from -256 to 256)
  - Does not affect the CCR bits

Loop Primitive Instructions

Mnemonic	Function	Equation or Operation
DBEQ	Decrement counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	(counter) − 1⇒ counter If (counter) = 0, then branch; else continue to next instruction
DBNE	Decrement counter and branch if ≠ 0 (counter = A, B, D, X, Y, or SP)	(counter) − 1⇒ counter If (counter) not = 0, then branch; else continue to next instruction
IBEQ	Increment counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	(counter) + 1⇒ counter If (counter) = 0, then branch; else continue to next instruction
IBNE	Increment counter and branch if ≠ 0 (counter = A, B, D, X, Y, or SP)	(counter) + 1⇒ counter If (counter) not = 0, then branch; else continue to next instruction
TBEQ	Test counter and branch if = 0 (counter = A, B, D, X,Y, or SP)	If (counter) = 0, then branch; else continue to next instruction
TBNE	Test counter and branch if ≠ 0 (counter = A, B, D, X,Y, or SP)	If (counter) not = 0, then branch; else continue to next instruction

Example of Loop Primitive Instruction

```
 1:                          ; Comparing the loop primitive and "normal"
 2:                          ; decrement and branch instructions
 3:                          ;
 4:        =000000FF     COUNT:  EQU     255     ; Counter value
 5:
 6:     0000 C6 FF             ldab    #COUNT  ; Initialize counter
 7:                          ;               - - -
 8:     0002                  LOOP:                   ; Here is the repetitive code
 9:                          ;               - - -
10:     0002 04 31 FD         dbne    b,LOOP  ; Using the DBNE instruction
11:                          ; The alternative is to do the following:
12:     0005 53               decb            ; But this sets the CCR bits
13:     0006 26 FA            bne     LOOP
```

Topics of discussion

- CISC and RISC Architectures
  - Reference: RISC – Wikipedia article (http://en.wikipedia.org/wiki/RISC)
- M68HC12 Instruction Set
  - Instruction Set Categories
  - Move Data Categories
  - Modify Data Categories
  - Decision Making Categories
  - Flow Control Categories
  - Other Categories

Unconditional Jumps and Branches

- JMP – uses an absolute 16-bit address
- JSR – also uses an absolute 16-bit address
  - Also pushes onto the stack a return address (following the JSR instruction)
- BSR – uses an 8-bit signed offset
  - Pushes onto the stack a return address
  - No LBSR exists, use “JSR opr16a,PC”
- Use RTS instruction to return from subroutine
- BRA,LBRA (Branch always) uses 8-, 16-bit offset
- BRN,LBRN (Branch never) never branches
- CALL and RTC used for calling subroutines when using paged memory
- Rules for using subroutines:
  - Stack should always be initialised to RAM
  - ***Never, ever* JMP to a subroutine**
  - ***Never, ever* JMP out of a subroutine**.

Jump and Jump to Subroutine Instructions (Flow Control Categories)

Short branch
Relative addressing

Very long branch

Long branch
Direct, indexed, indirect

Mnemonic	Function	Operation
BSR	Branch to subroutine	$SP - 2 \Rightarrow SP$ $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ Subroutine address $\Rightarrow$ PC
CALL	Call subroutine in expanded memory	$SP - 2 \Rightarrow SP$ $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 1 \Rightarrow SP$ $(PPAGE) \Rightarrow M_{(SP)}$ Page $\Rightarrow$ PPAGE Subroutine address $\Rightarrow$ PC
JMP	Jump	Address $\Rightarrow$ PC
JSR	Jump to subroutine	$SP - 2 \Rightarrow SP$ $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ Subroutine address $\Rightarrow$ PC
RTC	Return from call	$M_{(SP)} \Rightarrow$ PPAGE $SP + 1 \Rightarrow SP$ $M_{(SP)} : M_{(SP+1)} \Rightarrow PC_H : PC_L$ $SP + 2 \Rightarrow SP$
RTS	Return from subroutine	$M_{(SP)} : M_{(SP+1)} \Rightarrow PC_H : PC_L$ $SP + 2 \Rightarrow SP$

Jump and Jump to Subroutine Operand Syntax (Flow Control Categories)

Mnemonic	Operand Syntax				
	Relative	Direct	Ext	Indexed	Indexed-Indirect
JMP			opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]
JSR		opr8a	opr16a	oprx5,xysp oprx9,xysp oprx16,xysp abd,xysp oprx3,-xys+	[oprx16,xysp] [D,xysp]
BSR,BRA,BRN	rel8				
LBRA, LBRN	rel16				
CALL			opr16a, page	oprx5,xysp, page oprx9,xysp, page oprx16,xysp, page abd,xysp, page oprx3,-xys+, page	[oprx16,xysp] [D,xysp]

BAD Example – jumping into a subroutine

- What is **wrong** with the following code:

```
2:    0800 16 0804          jsr SUB     ; Go to the subroutine
3:    0803 A7          BACK: nop         ; The next op code
4:                     ;       - - -
5:                     ;       - - -
6:                     ;       - - -
7:    0804 A7          SUB:  nop         ; This is the subroutine
8:                     ;       - - -
9:    0805 06 0803          jmp BACK    ; Go back to main program
```

Interrupt Instructions

Mnemonic	Function	Operation
RTI	Return from interrupt	$(M_{(SP)}) \Rightarrow CCR; (SP) + \$0001 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + \$0002 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + \$0004 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) + \$0002 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + \$0004 \Rightarrow SP$
SWI	Software interrupt	$SP - 2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 1 \Rightarrow SP; CCR \Rightarrow M_{(SP)}$
TRAP	Unimplemented opcode interrupt	$SP - 2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 1 \Rightarrow SP; CCR \Rightarrow M_{(SP)}$

Interrupt Instructions

STOP	Stop	$SP - 2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 1 \Rightarrow SP; CCR \Rightarrow M_{(SP)}$ Stop CPU clocks
WAI	Wait for interrupt	$SP - 2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 1 \Rightarrow SP; CCR \Rightarrow M_{(SP)}$

- CLI and SEI are used to clear and set the interrupt bit in the CCR (translated to ANDCC #$EF and ORCC #$10 respectively)
- Study these instructions more closely when considering interrupts

Topics of discussion

- CISC and RISC Architectures
  - Reference: RISC – Wikipedia article (http://en.wikipedia.org/wiki/RISC)
- M68HC12 Instruction Set
  - Instruction Set Categories
  - Move Data Categories
  - Modify Data Categories
  - Decision Making Categories
  - Flow Control Categories
  - Other Categories

Other Categories

- Fuzzy Logic and Specialized Math Category (Will not study this category in this course)
  - If interested, consult Chapter 13
- Miscellaneous
  - NOP – no operation
    - For timing – one clock cycle is expended
    - Add NOPs into code during debugging so that re-assembling is not necessary
    - BRN and LBRN are essentially no-operation instructions as well
  - BGND – enables debugging feature (see chapter 20 for details)

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.
- *CPU12 Reference Guide (for HCS12 and original M68HC12), CPU12RG/D Rev. 2, 11/2001*

- Above are sources for most of the figures, tables and examples in the course notes.
- RISC and CISC:
  - http//en.wikipedia.org/wiki/RISC