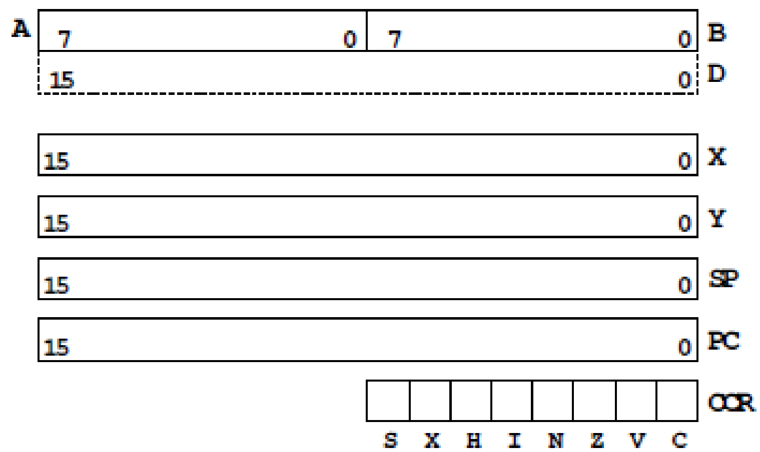
Review for Test 1

You may use any of the handouts from the Freescale data books, and one page of notes. No calculators allowed.

Programing Model



• Registers **A** and **B** are part of the programming model. Some instructions treat **A** and **B** as a sixteen-bit register called D for such things as adding two sixteen-bit numbers.

• The MC9S12 has a sixteen-bit register which tells the control unit which instruction to execute. This is called the **Program Counter** (PC). The number in PC is the address of the next instruction the MC9S12 will execute.

• The MC9S12 has an eight-bit register which tells the MC9S12 about the state of the ALU. This register is called the **Condition Code Register** (CCR). One bit (C) tells the MC9S12 whether the last instruction executed generated a carry. Another bit (Z) tells the MC9S12 whether the result of the last instruction was zero. The (N) bit tells whether the last instruction executed generated a negative result.

• Registers **X** and **Y** are 16-bit registers and are used mostly for indexing arrays. **SP** are a register used to point to the stack, and **PC** is the register that holds the program counter. part of the programming model.

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Convert Binary to Decimal

1111011_2
$1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
$1 \times 64 + 1 \times 32 + 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$
123_{10}

Convert Hex to Decimal

$82D6_{16}$
$8 \times 16^3 + 2 \times 16^2 + 13 \times 16^1 + 6 \times 16^0$
$8 \times 4096 + 2 \times 256 + 13 \times 16 + 6 \times 1$
33494_{10}

A Simple Assembly Language Program

```
prog:   equ    $2000          ; Start program at 0x2000
data:          equ    $1000   ; Data value at 0x1000

               org    prog
               ldaa   input
               inca
               staa   result
               swi

               org    data            ; Start of data
input:  dc.b   $A2
result: ds.b   1
```

Assembling an Assembly Language Program

```
Freescale HC12-Assembler
(c) Copyright Freescale 1987-2009

Abs.   Rel.   Loc      Obj. code     Source line
----   ----   ------   ---------     -----------
1      1
2      2      0000 2000              prog   equ     $2000 ; Start program at 0x2000
3      3      0000 1000              data   equ     $1000 ; Data value at 0x1000
4      4
5      5                             org     prog
6      6
7      7      a002000 B610 00        ldaa    input
8      8      a002003 42             inca
9      9      a002004 7A10 01        staa    result
10     10     a002007 3F             swi
11     11
12     12                            org     data
13     13     a001000 A2             input:  dc.b   $A2
14     14     a001001                result: ds.b   1
```

The MC9S12 has 6 addressing modes

Most of the HC12's instructions access data in memory
There are several ways for the HC12 to determine which address to access

Effective address:
Memory address used by instruction

Addressing mode:
How the MC9S12 calculates the effective address

MC9S12 ADDRESSING MODES:

INH Inherent:

Instructions which work only with registers inside ALU

IMM Immediate:

Value to be used is a part of the instruction

DIR Direct:

Instructions which give 8 LSB of address

EXT Extended:

Instructions which give the 16-bit address to be accessed

REL Relative (used only with branch instructions):

The relative addressing mode is used only in branch and long branch instructions

IDX Indexed:

Effective address is obtained from X or Y register (or SP or PC)

Summary of HCS12 addressing modes

Name	Example	Op Code	Effective Address
INH Inherent	ABA	18 06	None
IMM Immediate	LDAA #$35	86 35	PC + 1
DIR Direct	LDAA $35	96 35	0x0035
EXT Extended	LDAA $2035	B6 20 35	0x2035

IDX Indexed
IDX1
IDX2
	LDAA 3,X
	LDAA 30,X
	LDAA 300,X
		A6 03
		A6 E0 13
		A6 E2 01 2C
			X + 3
			X + 30
			X + 300

IDX Indexed Postincrement	LDAA 3,X+	A6 32	X (X+3 -> X)
IDX Indexed Preincrement	LDAA 3,+X	A6 22	X+3 (X+3 -> X)
IDX Indexed Postdecrement	LDAA 3,X-	A6 3D	X (X-3 -> X)
IDX Indexed Predecrement	LDAA 3,-X	A6 2D	X-3 (X-3 -> X)
REL Relative	BRA $1050 LBRA $1F00	20 23 18 20 0E CF	PC + 2 + Offset PC + 4 + Offset

Hand Assembling a Program

To hand-assemble a program, do the following:

1. Start with the org statement, which shows where the first byte of the program will go into memory (e.g., **org $2000** will put the first instruction at address **$2000**.)

2. Look at the first instruction. Determine the addressing mode used.
(e.g., **ldab #10** uses IMM mode.)

3. Look up the instruction in the **MC9S12 S12CPUV2 Reference Manual**, find the appropriate Addressing Mode, and the Object Code for that addressing mode. (e.g., **ldab IMM** has object code **C6 ii**.)

- **Table A.1 of S12CPUV2 Reference Manual** has a concise summary of the instructions, addressing modes, op-codes, and cycles.

4. Put in the object code for the instruction, and put in the appropriate operand. Be careful to convert decimal operands to hex operands if necessary. (e.g., **ldab #10** becomes **C6 0A**.)

5. Add the number of bytes of this instruction to the address of the instruction to determine the address of the next instruction (e.g., **$2000 + 2** = **$2002** will be the starting address of the next instruction.)

```
          org $2000
          ldab #10
    loop: clra
          dbne b,loop
          swi
```

```
Abs. Rel.   Loc    Obj. code     Source line
----  ----   ------  ---------     -----------
   1    1
   2    2      0000 2000          prog:  equ    $2000
   3    3                                org   prog
   4    4  a002000 C60A                  ldab #10
   5    5  a002002 87                loop: clra
   6    6  a002003 0431 FC               dbne b,loop
   7    7  a002006 3F                    swi
```

MC9S12 Cycles

- 68HC12 works on **48 MHz clock**

- Each processor cycle takes **41.7 ns** (1/24 MHz) to execute

- You can determine how many cycles an instruction takes by looking up the CPU cycles for that instruction in the S12CPUV2 Core Users Guide.

  – For example, **LDAA** using the **IMM** addressing mode shows one CPU cycle.

  – **LDAA** using the **EXT** addressing mode shows three CPU cycles.

```
2000                org $2000     ; Inst    Mode    Cycles
2000  C6 0A         ldab #10      ; LDAB    (IMM)     1
2002  87       loop: clra         ; CLRA    (INH)     1
2003  04 31 FC      dbne b,loop   ; DBNE    (REL)     3
2006  3F            swi           ; SWI               9
```

Total number of cycles:

$1 + 10 \times (1 + 3) + 9 = 50$

50 cycles = 50×41.7 ns/cycle = 2.08 μs

Using X and Y as Pointers

• Registers X and Y are often used to point to data.

• To initialize pointer use

ldx #table NOT **ldx table**

• For example, the following loads the address of table ($1000) into X; i.e., X will point to table:

ldx #table *; Address of table ⇒ X*

The following puts the first two bytes of table ($0C7A) into X. **X will not point to table**:

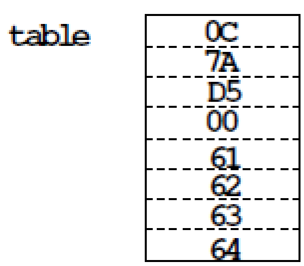
ldx table *; First two bytes of table ⇒ X*

• To step through table, need to increment pointer after use

ldaa 0,x
inx

OR

ldaa 1,x+




```
        org   $900
table:  dc.b  12,122,-43,0
        dc.b  'a','b','c','d'
```

Disassembly of an HC12 Program

• It is sometimes useful to be able to convert *HC12 op codes* into *mnemonics*.

For example, consider the hex code:

ADDR DATA
\---
1000 C6 05 CE 20 00 E6 01 18 06 04 35 EE 3F

• To determine the instructions, use Table A-2 of the HCS12 Core Users Guide.

– Use Sheet 1 & 2 of Table A.2.

\- Use Table A3. For Indexed addressing mode.

\- Use Table A.6 for loop instructions to determine whether the branch is positive (forward) or negative (backward).

C6 05 ⇒ LDAB #$05	LDAB, IMM addressing mode
CE 20 00 ⇒ LDX #$2000	LDX, IMM addressing mode
E6 01 ⇒ LDAB 1,X	LDAB, IDX addressing mode
18 06 ⇒ ABA	ABA, INH addressing mode
04 35 EE ⇒ DBNE X,(-18)	DBNE X,negative branch
3F ⇒ SWI	SWI, INH addressing mode

Signed Number Representation in 2's Complement Form:

If the most significant bit (MSB) is 0 (most significant hex digit 0−7), then the number is positive.

Example for 8−bit number:

3A $_{16}$ −> + (3 x 16^1 + 10 x 16^0) $_{10}$
+ (3 x 16 + 10 x 1) $_{10}$
+ 58 $_{10}$

If the most significant bit is 1 (most significant hex digit 8−F), then the number is negative.

Example for 8−bit number:

A3 $_{16}$ −> - (5D) $_{16}$
- (5 x 16^1 + 13 x 16^0) $_{10}$
- (5 x 16 + 13 x 1) $_{10}$
- 93 $_{10}$

One's complement table makes it simple to finding 2's complements

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

Addition of Hexadecimal Numbers

ADDITION:

C bit set when result does not fit in word

V bit set when P + P = N or
N + N = P

N bit set when MSB of result is 1

Z bit set when result is 0

Subtraction of Hexadecimal Numbers

SUBTRACTION:

C bit set on borrow (when the magnitude of the subtrahend is greater than the minuend

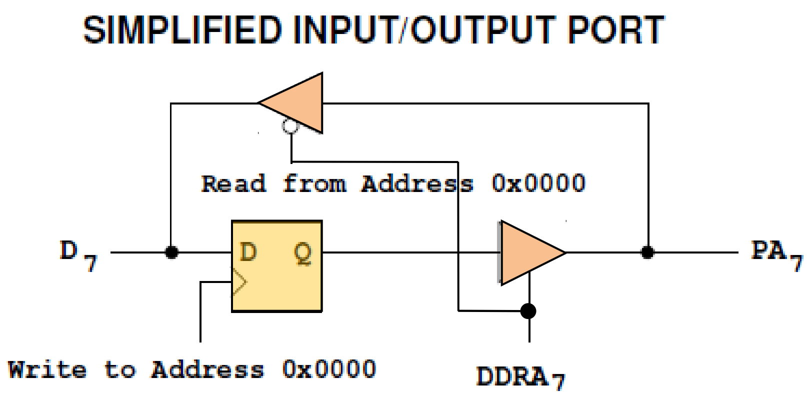
V bit set when N - P = P or
P - N = N

N bit set when MSB is 1

Z bit set when result is 0

Input and Output Ports

• Most I/O ports on MC9S12 can be configured as either input or output



• PORTA is accessed by reading and writing address $0000.
  - DDRA is accessed by reading and writing address $0002.

• PORTB is accessed by reading and writing address $0001.
  - DDRB is accessed by reading and writing address $0003.

• PTJ is accessed by reading and writing address $0268.
  - DDRJ is accessed by reading and writing address $026A.

• PTP is accessed by reading and writing address $0258.
  - DDRP is accessed by reading and writing address $025A.

• On the Dragon12, eight LEDs and four seven-segment LEDs are connected to PORTB

```
;A simple program to make PORTA output and PORTB
; input, then read the signals on PORTB and write these
; values out to PORTA

prog:   equu    $2000

PORTA:      equ     $00
PORTB:      equ     $01
DDRA:       equ     $02
DDRB:       equ     $03

            org     prog
            movb  #$ff,DDRA     ; Make PORTA output
            movb   #$00,DDRB    ; Make PORTB input

            ldaa    PORTB
            staa    PORTA
            swi
```

The Stack and the Stack Pointer

• When we use subroutines and interrupts it will be essential to have the storage region **the *Stack***.

• The **Stack Pointer** (SP) register is used to indicate the location of the last item put onto the stack.

• When you put something onto the stack (**push onto the stack**), the SP is decremented before the item is placed on the stack.

• When you take something off of the stack (**pull from the stack**), the SP is incremented after the item is pulled from the stack.

• Before you can use a stack **you have to initialize the Stack Pointer** to point to one value higher than the highest memory location in the stack. Use **LDS** to initialize the stack pointer.

Subroutines

• A subroutine is a section of **code which performs a specific task**, usually a task which needs to be executed by different parts of a program.

• When you call a subroutine, your code saves the address where the subroutine should return to. It does this by saving the return address on the stack.

- This is done automatically for you when you get to the subroutine by using the **JSR** (Jump to Subroutine) or **BSR** (Branch to Subroutine) instruction. This instruction **pushes the address** of the instruction following the **JSR**/**BSR** instruction **on the stack**.

- After the subroutine is done executing its code it needs to return to the address saved on the stack when **RTS** is used.