EEL 4744C: Microprocessor Applications

Lecture 5

68HC12 Instruction Set
Reading Assignment

• Software and Hardware Engineering (Old version) Chapter 4

Or

• Software and Hardware Engineering (New version) Chapter 7

And

• CPU12 Reference Manual Chapter 5
Some Tips

• 68HC12 has >1000 instructions!
  – They are grouped into a few (17) functional categories
  – Besides operation, variance w.r.t. effect on CCR, available addressing modes, etc
  – Details found in book as well as Motorola CPU Ref. Guide (short) and CPU Ref. Manual (long)
M68HC12 Instruction Set Categories

- Load registers
- Store registers
- Transfer/Exchange Registers
- Move memory contents
- Decrement/Increment
- Clear/Set
- Arithmetic
- Logic

- Rotates/Shifts
- Data test
- Fuzzy logic & Specialized math
- Conditional branch
- Loop primitive
- Jump and branch
- Condition code
- Interrupt
- Miscellaneous
Load and Store Instructions

- 8-bit load and store instructions (LDAA, LDAB, STAA, STAB)
- 16-bit load and store instructions (LDD, LDS, LDX, LDY, STD, STS, STX, STY)

MSB $\leftrightarrow$ lower address (EA)
LSB $\leftrightarrow$ higher address (the next location)
Endianness (Byte Order)

• **Big endian**: the most significant byte of multibyte data is stored at the lowest memory address
  – Sun's SPARC, Motorola's 68K, and the PowerPC families

• **Little endian**: the least significant byte of multibyte data is stored at the lowest memory address
  – Intel's 80x86

```c
char c1 = 1;
char c2 = 2;
short s = 255;  // 0x00FF
long l = 0x44332211;
```

A little-endian memory dump
## Load Register Instructions

### Load Registers

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAA</td>
<td>(M)\rightarrow A</td>
<td>LDAB</td>
<td>(M)\rightarrow B</td>
</tr>
<tr>
<td>LDD</td>
<td>(M:M+1)\rightarrow D</td>
<td>LDS</td>
<td>(M:M+1)\rightarrow SP</td>
</tr>
<tr>
<td>LDX</td>
<td>(M:M+1)\rightarrow X</td>
<td>LDY</td>
<td>(M:M+1)\rightarrow Y</td>
</tr>
<tr>
<td>LEAS</td>
<td>EA\rightarrow SP</td>
<td>LEAX</td>
<td>EA\rightarrow X</td>
</tr>
<tr>
<td>LEAY</td>
<td>EA\rightarrow Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load memory data to Reg.

Load effective address to Reg.

Load stack element to Reg.

See Freescale manual for supported addressing mode and impact on CCR

---

Dr. Tao Li
## Store Register Instructions

(see Section 4.4)

### Store Registers

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAA</td>
<td>A $\uparrow$(M)</td>
<td>STAB</td>
<td>B $\uparrow$(M)</td>
</tr>
<tr>
<td>STD</td>
<td>D $\uparrow$(M:M+1)</td>
<td>STS</td>
<td>SP $\uparrow$(M:M+1)</td>
</tr>
<tr>
<td>STX</td>
<td>X $\uparrow$(M:M+1)</td>
<td>STY</td>
<td>Y $\uparrow$(M:M+1)</td>
</tr>
<tr>
<td>PSHA</td>
<td>A $\uparrow$(SP)</td>
<td>PSHB</td>
<td>B $\uparrow$(SP)</td>
</tr>
<tr>
<td>PSHD</td>
<td>D $\uparrow$(SP:SP+1)</td>
<td>PSHC</td>
<td>CCR $\uparrow$(SP)</td>
</tr>
<tr>
<td>PSHY</td>
<td>Y $\uparrow$(SP:SP+1)</td>
<td>PSHX</td>
<td>X $\uparrow$(SP:SP+1)</td>
</tr>
</tbody>
</table>

- **Store Reg. data to memory**
- **Store Reg. data to stack**
- **Save CCR to stack**

See Freescale manual for supported addressing mode and impact on CCR
What is Wrong with this Program?

COUNT: EQU !8 ; Loop counter

ldab #COUNT ; Initialize loop counter

LOOP:

decb ; Decrement the B register and
; branch to LOOP if B register is
; not zero

lda a #$64 ; Load the A register with some
; data

bne LOOP

decb sets Z bit in CCR
bne detects Z bit in CCR
lda a (accidentally) alters CCR
Stack Instructions

- Use **LDS** to initialize; access via PSHA, PSHB, PSHX, etc., PULA, PULB, PULX, etc.)

- Access is normally balanced (i.e. matching pushes and pulls, **JSRs** and **RTSs**)

- e.g. to pass several parameters to subroutine via stack, we can push them before **JSR**, within subroutine we pull off **RA** from stack (and keep), then pull off parameters (in reverse order), then restore RA before we get to the RTS; e.g. consider passing input and output parameter via stack
Stack Instructions (2)

Push RA

- **PSHX**; push parameter on stack
- **JSR** **SUBR**; call subroutine

...  

SUBR

- **PULY**; pop off RA from stack (and keep)
- **PULD**; pop input parameter from stack

...; do some work...

- **PSHA**; push output parameter on stack
- **PSHY**; push RA back on stack
- **RTS**; return to calling pgm via RA on stack

Pull RA

To obtain SUBR output parameter, caller can execute **PULA**
Load Effective Address Instructions

• LEA instructions (LEAX, LEAY, LEAS) to save computed EA in 16-bit pointer register
  – An Effective Address (EA) is the memory address from or to which data are transferred
  – e.g. if Y=$1234 and instruction LEAX $10,Y executed, then stores EA=$1244 in X
  – Used for calculating memory addresses at run-time
  – They Do NOT modify CCR contents
  – LEA instructions is useful if we want to change X, Y, SP register by more than one (e.g. LEA_ vs. IN_)

Dr. Tao Li
Example: LEA_ Instructions

Adds 10 bytes passed on the stack and returns the sum on the A register

NUM:      EQU     !10
PROG:     EQU     $0800
DATA:     EQU     $0900
STACK:    EQU     $0a00
ORG     PROG

lds     #STACK
ldab    #NUM
ldx     #BUF
LoadLoop:
ldaa    0,x
psha
inx
dbne    b,LoadLoop
ldab    #NUM
jsr     CalcSum
leas    NUM,SP
swi

CalcSum:
leas    2,SP
pula
decb
add_loop:
adda    1,SP+
dbne    b,add_loop
leas    {0-NUM-2},SP
rts

ORG     DATA
BUF:      DB      1,2,3,4,5,6,7,8,9,!10
Transfer Register Instructions

• Transfer instructions from one register to another (e.g. TAB, TBA, TFR reg, reg)
  – If 8-bit to 16-bit transfer, upper formed as sign-extension of lower
  – If 16-bit to 8-bit transfer, low byte transferred from source

• Exchange instructions swap contents between registers (e.g. EXG reg, reg)
  – If 8-bit and 16-bit swap, low bytes exchanged and high byte of 16-bit reg. set to $00

Dr. Tao Li
Move Instructions

• Transfer from one memory location to another w/o using CPU registers (MOVB or MOVW)
  – Valuable for a CPU with only a few registers
  – Index addressing /w 9- and 16- bit constant offsets and indexed-indirect addressing are not allowed

• Copies data from memory specified by first operand to memory specified by second (no CPU register involved)
  
  MOVB #$64,DATA1 ; initializes byte in memory
  MOVW DATA2,DATA3 ; copies word from address DATA2 to DATA3
  MOVB 0,Y,1,X+ ; copies byte from (Y+0) in memory to (X); then X←X+1
### MOV Instructions: Example 1

Write a program segment to reverse the order of data in a 100-byte table

<table>
<thead>
<tr>
<th>LEN:</th>
<th>EQU</th>
<th>!100</th>
<th>; Length of the table</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDX</td>
<td>#TABLE</td>
<td></td>
<td>; Point to start of the table</td>
</tr>
<tr>
<td>LDY</td>
<td>#TABLE+LEN-1</td>
<td></td>
<td>; Point to the end byte</td>
</tr>
<tr>
<td>LDAB</td>
<td>#{LEN/2}</td>
<td></td>
<td>; Init counter</td>
</tr>
</tbody>
</table>

**LOOP:**

- **LDAA** 0,X ; Get a byte out of the way
- **MOVB** 0,Y,1,X+ ; Get from bottom, put in top and increment the top pointer
- **STAA** 1,Y- ; Put top in bottom and decrement the bottom pointer
- **DBNE** B,LOOP ; Decrement counter and branch if not done

**TABLE:**

- **DS** LEN

---

Dr. Tao Li
Transfer data from one buffer to another

LDX   #BUFF1 ; initialize pointer to first buffer
LDY   #BUFF2 ; initialize pointer to second buffer
LDAA  #LEN  ; initialize loop counter to length of buffer

LOOP: MOVB 1,X+,1,Y+ ; copy element from first to second buffer
DBNE  A,LOOP ; decrement loop counter and branch if not zero

Or a faster way by moving a word at a time

LDAA  #(LEN/2) ; initialize counter to length of buffer

LOOP: MOVW 2,X+,2,Y+ ; copy element from first to second buffer
DBNE  A,LOOP ; decrement loop counter and branch if not zero
Decrement and Increment Instructions

- Subtract 1 from register or memory location specified (e.g. DEC, DECA, DECB, DEX, etc.)
- Add 1 to register or memory location specified (e.g. INC, INCA, INCB, INX, etc.)
- All but DES and INS affect CCR bits

DES ⇔ LEAS -1, S
INS ⇔ LEAS 1, S
Memory Based Counter

- Declare counter using memory location
  - Does not occupy register
  - Can have more bits (e.g. >8 bits)
- The following (buggy) code segment declares and decrements a 16 bit memory based counter

```assembly
BIG:    EQU     !1000  ; !1000 == $03E8

movw    #BIG,Cnter ; Initialize the counter in memory

LOOP:   
    - - -
    dec    Cnter
    bne    LOOP
    - - -

Cnter:DS   2    ; 16-bit counter
```

• Declare counter using memory location
  - Does not occupy register
  - Can have more bits (e.g. >8 bits)

• The following (buggy) code segment declares and decrements a 16 bit memory based counter
Memory Based Counter

• The bug free code segment

```assembly
BIG:    EQU     !1000

movw    #BIG,Cnter  ; Initialize the counter in memory

LOOP:
   - - -
   pshx
   ldx    Cnter
   dex    Cnter
   stx    Cnter
   pulx   LOOP
   bne
   - - -

Cnter:DS  2       ; 16-bit counter
```

This instruction is not harmful

pshx and pulx do not affect CCR

Dr. Tao Li
Clear and Set Instructions

• Clear and Set bits
  – CLR, CLRA, and CLRB to clear a byte in memory, or A or B registers
  – BCLR and BSET for bitwise clear/set bits in byte of memory via mask byte (where 1=affected, 0=not)
    • BCLR Operand, Mask    BSET Operand, Mask
      – Operand: memory location specified using direct addressing, extended and indexed addressing
    • Useful for controlling external devices one bit at a time
    • For example, with LEDs attached to output port, can turn on/off each individually
Example: Clear & Set Instructions

Use BCLR and BSET to turn on and off LEDs

Port H is mapped to memory address $0024

Use buffers such as 74HC241 for higher current LED's

Dr. Tao Li
Example: Clear & Set Instructions

BIT0: EQU %00000001 ; Mask for bit 0
...
BIT7: EQU %10000000 ; Mask for bit 7
ALL: EQU %11111111
REGS: EQU $0000 ; Start of the I/O regs
PORTH: EQU REGS+$24 ; Offset for Port H
DDRH: EQU REGS+$25 ; Data dir register

bset DDRH,ALL ; Make all bits outputs
LOOP: bset PORTH,ALL ; Turn out all LEDs (NOTE: active-low)
bclr PORTH,BIT0 ; Turn on bit 0
...
bclr PORTH,BIT7 ; Turn on bit 7
bra LOOP ; Do it forever

Of course, LEDs will appear on the whole time since mP is much faster than our eyes!
Shift and Rotate Instructions

- Shifts come in two kinds, arithmetic and logical. Shifts and rotates move left or right by one bit

  • Logical (LSL) and arithmetic (ASL) shift left are identical; 0 fed in (LSB), and out (MSB) goes to C bit in CCR

  • Same for logical shift right (LSR), i.e. 0 fed in (MSB), and out (LSB) goes to C bit in CCR

  • For arithmetic shift right (ASR), copy of sign bit fed in (MSB), out (LSB) goes to C bit
Logic Shift Instructions

- **LSL: Logic Shift Left**

- **LSR: Logic Shift Right**
Arithmetic Shift Instructions

- **ASL: Arithmetic Shift Left**

- **ASR: Arithmetic Shift Right**
Arithmetic Shift Instructions

- Arithmetic shifts left can serve as fast multiplication by powers of two; same for right as division

; multiply 16-bit number in D by 10 using arithmetic left shifts

std TEMP ; Save in location TEMP
asld ; X2
asld ; X2 again = X4
addd TEMP ; Add the original. Now X5
asld ; X2 = X10
Rotate Instructions

• Rotate left (ROL) and rotate right (ROR) instructions rotate through the C bit
  – i.e. w/ ROL, all move left, LSB updated from C, then C updated by old MSB

• See textbook a complete list of shift and rotate instructions
Rotation Instructions

- **ROL: Rotation Left Instruction**
  - Carry bit is involved in rotation

- **ROR: Rotate Right Instruction**
  - Carry bit is involved in rotation
LED Example using Rotate Instructions

COUNT: EQU 8 ; Going to do 8 bits
ALLBITS: EQU %11111111 ; Spec all bits
FIRST: EQU %11111110 ; Turn on bit 0
PORTH: EQU $24 ; Address of Port H

; Turn off all bits
OUTER:
    bset PORTH,ALLBITS
    ldaa #FIRST ; Initialize for bit-0
    ldab #COUNT
LOOP:
    staa PORTH ; Turn on a bit
    jsr Delay ; Delay for a while
    clc ; clear carry bit to rotate
    rola ; Shift the ACCA left
    dbne b,LOOP ; Do it for 8 bits
    bra OUTER ; Do it forever

; Dummy subroutine
Delay: rts

Dr. Tao Li
Arithmetic Instructions

• See textbook for a complete list of arithmetic instructions

• Binary addition and subtraction
  – Add register to register (ABA for A+B→A) or memory to register (ADDA, ADDB, ADDD)
  – Same for subtraction (SBA) and (SUBA, SUBB, SUBD)
  – Add with carry input (ADCA, ADCB) for adding memory + C flag into accumulator
• Binary addition and subtraction (contd.)

–useful for multi-byte arithmetic w/o size limit, performed in stages; for example:

Add two 16-bit numbers stored in NUM1:NUM1+1 and NUM2:NUM2+1

LDAA NUM1+1 ; load lower byte of first number
ADDA NUM2+1 ; add lower byte of second number
STAA NUM3+1 ; store lower byte of result
LDAA NUM1 ; load upper byte of first number
ADCA NUM2 ; add upper byte of second number
STAA NUM3 ; store upper byte of result

Of course, here we could more easily just add into D register in one step!
Binary addition and subtraction (contd.)

- Same for subtraction (SBCA, SBCB); e.g. SBCA means $A-(M)-C \rightarrow A$

- Other registers (e.g. X, Y) not available

- The LEA instructions previously consider could be considered as 16-bit arithmetic instructions
Decimal (packed BCD) addition

• We sometimes use pBCD code to input, use, or store data (one byte contains two digits)

• DAA instruction used immediately after a binary byte addition to adjust for proper pBCD format

• DAA automatically determines correction factor to add; for example:

    NUM1  DC.B  $12 ; represents decimal 12 in pBCD  00010010
    NUM2  DC.B  $09 ; represents decimal 9 in pBCD  00001001
    NUM    DS.B  1 ; sum to go here will be 21           ____________
    ...                                      00011011
    ldaa  NUM1 ; load first pBCD value
    adda  NUM2 ; add second pBCD value as if they’re binary
    daa   ; adjust back to pBCD format  00100001
    staa NUM3 ; store pBCD result

Dr. Tao Li
Negation and Sign Extension

– Can negate (i.e. two’s complement) memory location (NEG) or register (NEGA, NEGB)

– Sign-extension instruction (SEX) is useful for converting signed bytes to words

– SEX is simply another mnemonic for any 8-bit to 16-bit register transfer, such as TFR B,D; creates upper byte by replicating sign bit of lower byte (a.o.t. padding with zeros as we would do for unsigned extension)

– May have A, B, or CCR registers as source and D, X, Y, or SP as destination
Multiplication

• Signed or unsigned binary may be multiplied (or divided), but separate instructions since signed multiplication works differently than unsigned

• Registers involved are implicit operands, and we have but three choices:

  MUL ; 8-bit unsigned A×B→D; C=1 if bit 7 of result = 1 to ;allow rounding for fractional #s (more later)

  EMUL ; 16-bit unsigned D×Y→Y:D; C=1 if bit 15 of result=1

  EMULS ; 16-bit signed D×Y→Y:D; C=1 if bit 15 of result=1
Example: Multiplication

; 8-bit×8-bit unsigned multiply
ldaa DATA1 ; Get the multiplier
ldab DATA2 ; Get the multiplicand
mul ; The product is in D
std DATA3

; 8-bit × 8-bit signed multiply
ldaa DATA1 ; Get the multiplicand
sex a,y ; Sign extend into Y
ldaa DATA2 ; Get the multiplier
tfr a,d ; Same as SEX A,D
emuls ; Extended multiply Y*D
        ; 32-bit product is in Y:D
        ; The 16 bits we need are in D
std DATA3
Fractional Number Arithmetic

- Arithmetic instructions (e.g. add, subtract and multiply) can also be used for fractional numbers; e.g.:
  - \(0.50 + 0.25 = .1000_2 + .0100_2 = .1100_2 = 0.75\)
  - \(0.75 - 0.25 = .1100_2 - .0100_2 = .1000_2 = 0.50\)
  - \(0.50 \times 0.25 = .1000_2 \times .0100_2 = .00100000_2 = 0.125\)
  - \(0.375 \div 0.50 = .0110_2 \div .1000_2 = .1100_2 = 0.75\)

4-bit by 4-bit mult. above yields 8-bit product
When multiplying with MUL on 68HC12, 8-bit fractional multiply yields 16-bit fractional product; sometimes convenient to discard lesser half and round-up to upper half; accomplished by using C update feature of MUL see below:

; Fractional multiplication with rounding
ldaa DATA1 ; 8-bit fraction
ldab DATA2 ; 8-bit fraction
mul ; 16-bit fraction result (AxB=>D)
adca #0 ; Increment A if B is 0.5 or greater
staa DATA3 ; 8-bit rounded result

Dr. Tao Li
Division

• **Unsigned** division with **IDIV** (16-bit), **FDIV** (16-bit fractional), and **EDIV** (32-bit); signed with **IDIVS** (16-bit) and **EDIVS** (32-bit)

• **IDIV** does \( \frac{D}{X} \) with quotient to \( X \) and remainder to \( D \).
  – if denominator was 0, sets \( C=1 \) and \( X=\text{FFFF} \)

• **FDIV** for fractional divide, where **numerator** (in \( D \)) assumed less than **denominator** (in \( X \)) and both assumed to have same radix point; thus, result is binary-weighted fraction; quotient to \( X \) and remainder to \( D \)
  – Divide by 0 as before, and \( V=1 \) if denominator \( \leq \) numerator

• **EDIV** and **EDIVS** do \( \frac{Y:D}{X} \) with quotient to \( Y \) and remainder to \( D \)
Example: Division

; Extended, unsigned multiply and divide (here: DATA1*DATA2/100)

ldd DATA1 ; load first unsigned number
ldy DATA2 ; load second unsigned number
emul ; 32-bit product in Y:D
ldx #100 ; load divisor
ediv ; do the division
sty DATA3 ; save quotient
std DATA4 ; save remainder
Logic Instructions

- Bit-wise logical operations of **AND** (good to clear bits), **OR** (set bits), and **EOR** (bit-wise XOR)

- Also one’s-complement operation also available

- AND instructions (ANDA, ANDB, ANDCC), OR instructions (ORAA, ORAB, ORCC), EOR instructions (EORA, EORB), and COM (i.e. complement) instructions (COM, COMA, COMB)
  - CCR variants to clear (AND) or set (OR) individual bits in CCR
  - e.g. **ANDA #$0F** will clear upper half of A, **ORAA #$F0** will set upper half
; Convert BCD number in register A to ASCII and print (using PRINT subroutine).

tfr a,b ; Save the BCD number in B
lsra ; Shift 4 bits to right
lsra
lsra
lsra
ora #30 ; Convert to ASCII
jsr PRINT ; Go print it
tfr b,a ; Get the original back
anda #0F ; Set upper-half to 0
ora #30 ; Convert to ASCII
jsr PRINT ; Go print it

$30 => 0011 0000
Data Test Instructions

• Used to modify CCR without changing the operands

• BITA and BITB instructions work like ANDA and ANDB w/o storing result; e.g.
  ```
  LDAA  #\%10000000
  BITA  PORTH       ; test Bit-7 on PORTH
  BNE   DO_IF_ONE   ; branch if Bit-7 was set, else fall thru
  ```

• CBA, CMPA, CMPB, CPD, CPX, CPY, and CPS work like subtraction but w/o storing result

• TST, TSTA, and TSTB work like subtraction with 0 (e.g. does A-0) but w/o storing result
  – To test if memory location, A, or B is zero or negative
Conditional Branch Instructions

• These instructions test bits in CCR

• Short (PC relative, **8-bit** displacement) or long branches (PC relative, **16-bit** displacement)
  – Long designated by **L** prefix in mnemonic (e.g. BNE versus LBNE)

• Different conditional branches for **signed** and **unsigned** data, since e.g. $FF$ might be large or small

• See textbook for detailed definition of various branches and their symbolic operations
Loop Primitive Instructions

- Decrement/increment counter in registers A, B, D, X, Y, or SP, then branch if the counter equals (or does not equal) to zero

- DBNE, DBEQ, IBNE, IBEQ, TBNE, and TBEQ (latter two tests register w/o inc. or dec.)

- By contrast to regular inc. and dec. instructions, CCR not affected

- Instead of 8-bit offset (-128~127) of short branches, uses 9-bit offset (-256~255)
Unconditional Jump & Branch Instructions

- Branches used **9-bit relative offset**, while jumps & calls support **wide array of addressing modes**

- Unconditional jumps (JMP) and branches (BRA, LBRA) **always** go to the target

- Call a subroutine via branch (BSR), jump (JSR), or call in expanded memory (CALL)
  - Expansion memory of up to 4MB program space and 1MB data space on some devices

- Returns from a subroutine via RTS or (if expanded memory) via RTC

Dr. Tao Li
Never Jump out of a Subroutine

```
jsr SUB
BACK:
    nop
    - - -   Main program
    ;
    ;
    ;
SUB:
    nop
    - - -   Subroutine
    ;
    jmp BACK
    ;
```

Dr. Tao Li
• CCR instructions: ORCC and ANDCC used to set/clear C and V bits; useful when returning Boolean result from subroutine (i.e. need not waste register for result; just indicate correct outcome or error, and let calling program branch accordingly to handle the outcome)

• Interrupt instructions, BGND (background debug), NOP, STOP (stop all clocks and puts device in power-save mode, later awakened by interrupt)
Example: Using the Carry Bit for Boolean Information Transfer

**Initialization**

```
STACK:    EQU     $0c00   ;Stack location
CARRY:    EQU     %00000001 ;Bit 0 is carry
lds      #STACK     ;Init stack pointer
```

```
;Branch to subroutine that checks if a variable is within a set range
bsr      check_range

;C=0 for variable in range
bcc      IN_RANGE

OUT_OF_RANGE:
; Print an error message if out of range
;
IN_RANGE:
; Continue with the process
;
IN_RANGE:

```

**Main Program**

```
; Subroutine to check if a variable is in range. If it is, clear the carry bit, otherwise set the carry bit and return
check_range:
;
OK:         clc         ; Clear carry bit
bra      DONE

NOT_OK:     sec         ; Set carry bit
DONE:       rts         ; Return with the bit clear or set
```

Dr. Tao Li