# ECET 310-001
# Chapter 1 Concluded

W. Barnes, 9/2006, rev'd 9/07

**Ref. Huang, Han-Way, *The HCS12/9S12: An Introduction to Software and Hardware Interfacing,* Thomson/Delmar**.

## In this set of Slides:

1. Last of the Six Basic Addressing Modes
2. Introduction to Instructions
   - Three Types of Move/Copy
   - Add/Subtract
3. Instruction Execution & Queue
4. Three Simple Example Programs

## Six Basic Addressing Modes cont'd

**6. Indexed, which has 5 basic sub-types**

1. Constant (signed) offset (5, 9, 16 bits)

   ldaa 10,x ***; loads a with m[x+10]***

2. Indirect constant or reg. D to create a POINTER to ADDRESS of operand

   ldaa [10,x]
   ***; loads A with m[ ], which is pointed to by contents of m[x+10,11]& [x+11]***

   ex. If X = \$1000, M[\$1010 & \$1011] = \$20 & \$50, Then, A ← M[\$2050]

1. Auto pre OR post increment or decrement of index register (**N.B.** the number given with this type of addressing is the amount of incr or dec, not an offset)

   ldaa 1,-SP ***;decrements SP by 1 and then loads A***
   staa 2, X+ ***;stores A and then increments X by 2***

***Questions:***
***In #1, what determines if the offset is 5, 9, or 16 bits?***
***What distinguishes #2 from #1 above?***
***In #3, how can you indicate whether instruction executes or inc/decr first?***

**Indexed Addressing cont'd on next slide**

## Sixth Basic Addressing Mode cont'd

### [5 basic types of Indexed Mode cont'd]

4. Accumulator Offset Indexed Addressing
   - The accumulator can be the 8-bit A or B or the 16-bit accumulator D.
   - The base register can be X, Y, SP, or PC.

```
ldaa  B,X  ;loads A with m[B+X]
stab  A,Y  ;stores B in m[A+Y]
```

**Question:**

The ***effective*** address is the sum of the index register plus the unsigned number in the accumulator. Therefore, if X = 2A 04 and B = 4C in the first example, where in memory will the uctlr get the number to load into A?

**Indexed Addressing cont'd on next slide**

# Sixth Basic Addressing Mode Concluded

[5 basic types of Indexed Mode concluded]

**5. Accumulator D Indirect Indexed Addressing** Value in D is added to the value in the base index register to form the address of the memory location that contains the address to the memory location affected by the instruction. Square brackets distinguish this addressing mode from accumulator D offset indexing (type #4, on previous slide).

Example using Computed GOTO

```
1.            jmp     [D,PC]   ;D previously loaded with 0,2,or 4
2.  go1       dc.w    target1  ; note these assembler
3.  go2       dc.w    target2  ;    directives store
4.  go3       dc.w    target3  :       2-byte addresses here
     …
5.  target1   …
              .
6.  target2   …
              .
7.  target3   …
```

Suppose D = 4 on reaching jmp instruction, then 4 is added to PC, which will now point to go3 and thus target3 will be used in the jmp instruction. This results in the instructions starting at target3 to be executed.

**Question: in line 2 of the example, how many labels are used and what are they?**

# Introduction to Instructions

**<u>What to keep in mind when using instructions:</u>**

**How does the instruction affect registers and/or memory?**

**How does the instruction affect the CCR?**

**Is it clear where the input numbers are and where the results (destination) should go?**

**Is the program using signed numbers?**

**What kind of addressing modes are available for a particular instruction?**

## **Introduction to Instructions Cont'd.,** <u>3 basic types</u> of move/copy:

1. Load / Store registers from / to memory
   - updates the N and Z flags, clear V flag
   - See table 1.4, p.19 (next slide)
   - **Examples:** ldaa 0,X ; staa $20 ; stx $8000 ; ldd #100

2. Transfer, Exchange, Sign Extend (registers only)
   - See table 1.5, p.20, (slide after next)
   - **Examples :** tab ; TAB, tfr A,X ; exg D, X ; sex A, X

3. Move (mem ↔ mem, I/O register ↔ mem)
   - Also move immediate values into mem
   - See table 1.6, p. 22 (second slide after next)
   - **EXs:** movb #0, $1500 ; movb $100,$800 ; movw 0,X, 0,Y

**Questions:**

**How does the Appendix A show that flags are being changed (affected) by a load or store instruction?**

**How are memory and registers affected by the above examples?**

Table 1.4 Load and store instructions

| Mnemonic | Function | Operation |
|---|---|---|
| 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 |
| LDX | Load index register X | (M:M+1) ⇒ X |
| LDY | Load index register Y | (M:M+1) ⇒ X |
| LEAS | Load effective address into SP | Effective address ⇒ SP |
| LEAX | Load effective address into X | Effective address ⇒ X |
| LEAY | Load efective address into Y | Effective address ⇒ Y |

Store Instructions

| Mnemonic | Function | Operation |
|---|---|---|
| STAA | Store A | (A) ⇒ M |
| STAB | Store B | (B) ⇒ M |
| STD | Store D | (A) ⇒ M, (B) ⇒ M+1 |
| STS | Store SP | (SP) ⇒ M, M+1 |
| STX | Store X | (X) ⇒ M:M+1 |
| STY | Store Y | (Y) ⇒ M:M+1 |

## Table 1.5, Xfer, Exchange, and Sign Extend Instructions

| **Transfer** | | |
|---|---|---|
| TAB | Xfer A to B | (A) → B |
| TBA | Xfer B to A | (B) → A |
| TAP | Xfer A to CCR | (A) → CCR |
| TPA | Xfer CCR to A | (CCR) → A |
| TFR src, obj | Xfer reg. to reg. | (src) → obj |
| TSX | Xfer SP to X | (SP) → X |
| TXS | Xfer X to SP | (X) → SP |
| TSY | Xfer SP to Y | (SP) → Y |
| TYS | Xfer Y to SP | (Y) → SP |
| **Exchange** | | |
| EXG reg1, reg2 | Exchange two registers | (A) ↔ (B), etc. |
| XGDX | Exchange X and D | (X) ↔ (D) |
| XGDY | Exchange Y and D | (Y) ↔ (D) |
| **Sign Extension** | | |
| SEX | Sign extend 8-bit operand | (A,B,CCR) → X,Y, SP |

Questions: In TAB, what happens to register A?
What is at least one difference between Load and Transfer instructions?
What is a difference between Transfer and Exchange instructions?

## Move instructions

(Used within memory, not registers, but can also use to move immediate number into memory)

Table 1.6 Move instructions

| Transfer Instructions | | |
|---|---|---|
| Mnemonic | Function | Operation |
| MOVB<br>MOVW | Move byte (8-bit)<br>Move word (16-bit) | $(M1) \Rightarrow M2$<br>$(M{:}M{+}1_1) \Rightarrow M{:}M{+}1_2$ |

## Introduction to Instructions cont'd

- **Add/Sub Instructions**
  - Destinations are a CPU register or accumulator.
  - Three-operand ADD or SUB instructions always include the C flag as an operand and are used to perform multi-precision addition or subtraction
  - See table 1.7, p.22 (next slide)
    - adca $1000 ; A ⇐ [A] + [$1000] + C
    - suba $1002 ; A ⇐ [A] - [$1002]
    - sbca $1000 ; A ⇐ [A] - [$1000] - C
    - adda $1000 ; A ⇐ [A] + [$1000]

Question: How can you change the second example to subtract the number $44 from register A? What kind of addressing will be needed?

Table 1.7 Add and subtract instructions

| Add Instructions | | |
|---|---|---|
| Mnemonic | Function | Operation |
| ABA | Add B to A | (A) + (B) ⇒ A |
| ABX | Add B to X | (B) + (X) ⇒ X |
| ABY | Add B to Y | (B) + (Y) ⇒ Y |
| 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 without carry to D | (A:B) + (M:M+1) ⇒ A:B |
| **Subtract Instructions** | | |
| Mnemonic | Function | Operation |
| 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 | (D) - (M:M+1) ⇒ D |

## Instruction Execution Cycle

- One or more read cycles to fetch instruction opcode bytes and addressing information
- One or more read cycles to fetch the memory operand (s) (optional)
- Perform the operation specified by the opcode
- One or more write cycles to write back the result to either a register or a memory location (optional)

## Instruction Queue

- The HCS12 executes one instruction at a time and many instructions take several clock cycles to complete.
- When the CPU is performing the operation, it does not need to access memory.
  - The HCS12 prefetches instructions when the CPU is not accessing memory to speedup the instruction execution process.
  - There are two 16-bit queue stages and one 16-bit buffer. Unless buffering is required, program information is first queued in stage 1, and then advanced to stage 2 for execution.

## Simple Example Programs (*NOTE: see the Tracing Programs handout*)

```
;webex1, 8/4/06
;program adds (- 50) to a number and stores result
num        equ    -50   ; $CE = -50
           org    $1800
testval    db     60    ; $3C = +60
answer     rmb    1

           org    $2000
           ldaa   #num
           adda   testval   ; -50 + 60 = 10 = $0A
           staa   answer
           swi
           end
```

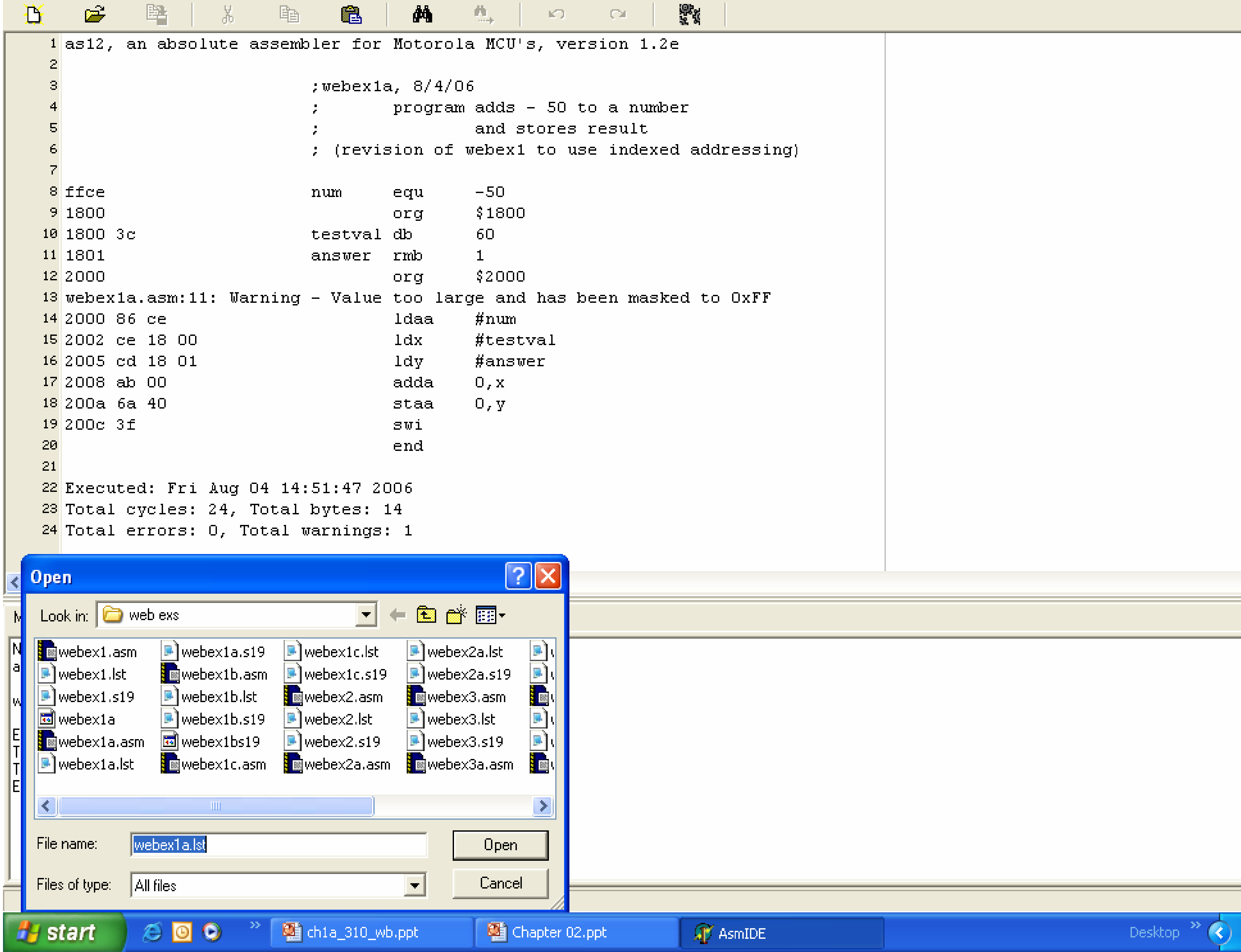
## Simple Example Programs Cont’d.

```
;webex1a, 8/4/06
;       program adds (- 50) to a number and stores result
; (revision of webex1 to use indexed addressing)
num             equu    -50
                org     $1800
testval         db      60
answer          rmb     1           ;reserve memory byte at answer

                org     $2000
                ldaa    #num
                ldx     #testval
                ldy     #answer
                adda    0,x
                staa    0,y
                swi
                end
```

## Next Slide:

- a list file for webex1a
- inset showing list of <u>all</u> (asm, lst, s19) files in directory



```
as12, an absolute assembler for Motorola MCU's, version 1.2e

                        ;webex1a, 8/4/06
                        ;       program adds - 50 to a number
                        ;               and stores result
                        ; (revision of webex1 to use indexed addressing)

ffce                    num     equ     -50
1800                            org     $1800
1800 3c                 testval db      60
1801                    answer  rmb     1
2000                            org     $2000
webex1a.asm:11: Warning - Value too large and has been masked to 0xFF
2000 86 ce                      ldaa    #num
2002 ce 18 00                   ldx     #testval
2005 cd 18 01                   ldy     #answer
2008 ab 00                      adda    0,x
200a 6a 40                      staa    0,y
200c 3f                         swi
                                end

Executed: Fri Aug 04 14:51:47 2006
Total cycles: 24, Total bytes: 14
Total errors: 0, Total warnings: 1
```

**Open**

Look in: web exs

| | | | |
|---|---|---|---|
| webex1.asm | webex1a.s19 | webex1c.lst | webex2a.lst |
| webex1.lst | webex1b.asm | webex1c.s19 | webex2a.s19 |
| webex1.s19 | webex1b.lst | webex2.asm | webex3.asm |
| webex1a | webex1b.s19 | webex2.lst | webex3.lst |
| webex1a.asm | webex1bs19 | webex2.s19 | webex3.s19 |
| webex1a.lst | webex1c.asm | webex2a.asm | webex3a.asm |

File name: webex1a.lst

Files of type: All files

# Simple Example Programs Cont’d.

```
;webex1b, 8/4/06, program adds (- 50) to a number and stores result
; (revision of webex1 with a test for invalid results:
;       if result exceeds 8-bit limits for signed numbers
;       store FF in location [valid], otherwise store 00)
; Program also includes a conditional branch
num             equu  -50
                org   $1800
testval         db    60
answer          rmb   1             ;reserve memory byte at answer
valid           rmb   1

                org   $2000
                clr   valid         ;make default 00
                ldaa  #num
                adda  testval
                bvc   good   ;skip if results ok (checks overflow flag, table 2.2, p. 54)
                com   valid         ;make FF
good            staa  answer
                swi
                end
```

# Simple Example Programs Cont'd.

```
;webex1c, 9/13/06,;          program adds - 50 to a number and stores result
; (revision of webex1 with a test for invalid results: see web1b for details)
; and use of indexed  w/  normal and auto incr.
num               equ    -50
                  org    $1800
data                           db     60, -200
results                        rmb     2
valid                          rmb     1
                  org    $2000
                  clr    valid                ;make default 00
                  ldab   #num
                  ldx    #data
                  ldy    #results
                  ldaa   1,x+
                  aba                         ;add (-50) to first number
                  bvc    good                 ;skip if results ok (overflow flag clear)
                  movb   #$FF,valid   ;   otherwise ...
good              staa   1,y+
                               ldaa    0,x
                  aba                          ;add (-50) to second number
                  bvc    good2                ;skip if results ok
                  movb   #$FF,valid
good2             staa   0,y
                  swi
                  end
```