ECET 310-001
Chapter 1 Concluded

W. Barnes, 9/2006, rev’d 9/07
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)
   
   ```assembly
   ldaa 10, x ; loads a with m[x+10]
   ```

2. **Indirect** constant or reg. D to create a POINTER to ADDRESS of operand
   
   ```assembly
   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)
   
   ```assembly
   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.
   
   ```
   ldax 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 directives store
3. go2 dc.w target2 ;
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

What to keep in mind when using instructions:

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?
3 basic types 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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAA</td>
<td>Load A</td>
<td>(M) ⇒ A</td>
</tr>
<tr>
<td>LDAB</td>
<td>Load B</td>
<td>(M) ⇒ B</td>
</tr>
<tr>
<td>LDD</td>
<td>Load D</td>
<td>(M:M+1) ⇒ (A:B)</td>
</tr>
<tr>
<td>LDS</td>
<td>Load SP</td>
<td>(M:M+1) ⇒ SP</td>
</tr>
<tr>
<td>LDX</td>
<td>Load index register X</td>
<td>(M:M+1) ⇒ X</td>
</tr>
<tr>
<td>LDY</td>
<td>Load index register Y</td>
<td>(M:M+1) ⇒ X</td>
</tr>
<tr>
<td>LEAS</td>
<td>Load effective address into SP</td>
<td>Effective address ⇒ SP</td>
</tr>
<tr>
<td>LEAX</td>
<td>Load effective address into X</td>
<td>Effective address ⇒ X</td>
</tr>
<tr>
<td>LEAY</td>
<td>Load effective address into Y</td>
<td>Effective address ⇒ Y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Store Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mnemonic</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>STAA</td>
</tr>
<tr>
<td>STAB</td>
</tr>
<tr>
<td>STD</td>
</tr>
<tr>
<td>STS</td>
</tr>
<tr>
<td>STX</td>
</tr>
<tr>
<td>STY</td>
</tr>
</tbody>
</table>
# Table 1.5, Xfer, Exchange, and Sign Extend Instructions

<table>
<thead>
<tr>
<th>Transfer</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAB</td>
<td>Xfer A to B</td>
<td>(A) → B</td>
</tr>
<tr>
<td>TBA</td>
<td>Xfer B to A</td>
<td>(B) → A</td>
</tr>
<tr>
<td>TAP</td>
<td>Xfer A to CCR</td>
<td>(A) → CCR</td>
</tr>
<tr>
<td>TPA</td>
<td>Xfer CCR to A</td>
<td>(CCR) → A</td>
</tr>
<tr>
<td>TFR src, obj</td>
<td>Xfer reg. to reg.</td>
<td>(src) → obj</td>
</tr>
<tr>
<td>TSX</td>
<td>Xfer SP to X</td>
<td>(SP) → X</td>
</tr>
<tr>
<td>TXS</td>
<td>Xfer X to SP</td>
<td>(X) → SP</td>
</tr>
<tr>
<td>TSY</td>
<td>Xfer SP to Y</td>
<td>(SP) → Y</td>
</tr>
<tr>
<td>TYS</td>
<td>Xfer Y to SP</td>
<td>(Y) → SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exchange</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXG reg1, reg2</td>
<td>Exchange two registers</td>
<td>(A) ↔ (B), etc.</td>
</tr>
<tr>
<td>XGDX</td>
<td>Exchange X and D</td>
<td>(X) ↔ (D)</td>
</tr>
<tr>
<td>XGDY</td>
<td>Exchange Y and D</td>
<td>(Y) ↔ (D)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sign Extension</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEX</td>
<td>Sign extend 8-bit operand</td>
<td>(A,B,CCR) → X,Y, SP</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVB</td>
<td>Move byte (8-bit)</td>
<td>(M1) ⇒ M2</td>
</tr>
<tr>
<td>MOVW</td>
<td>Move word (16-bit)</td>
<td>(M:M+1_1) ⇒ M:M+1_2</td>
</tr>
</tbody>
</table>
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 \leftarrow [A] + [$1000] + C
  • suba $1002 ; A \leftarrow [A] - [$1002]
  • sbca $1000 ; A \leftarrow [A] - [$1000] - C
  • adda $1000 ; A \leftarrow [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?
### Add Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABA</td>
<td>Add B to A</td>
<td>(A) + (B) ⇒ A</td>
</tr>
<tr>
<td>ABX</td>
<td>Add B to X</td>
<td>(B) + (X) ⇒ X</td>
</tr>
<tr>
<td>ABY</td>
<td>Add B to Y</td>
<td>(B) + (Y) ⇒ Y</td>
</tr>
<tr>
<td>ADCA</td>
<td>Add with carry to A</td>
<td>(A) + (M) + C ⇒ A</td>
</tr>
<tr>
<td>ADCB</td>
<td>Add with carry to B</td>
<td>(B) + (M) + C ⇒ B</td>
</tr>
<tr>
<td>ADDA</td>
<td>Add without carry to A</td>
<td>(A) + (M) ⇒ A</td>
</tr>
<tr>
<td>ADDB</td>
<td>Add without carry to B</td>
<td>(B) + (M) ⇒ B</td>
</tr>
<tr>
<td>ADDD</td>
<td>Add without carry to D</td>
<td>(A:B) + (M:M+1) ⇒ A:B</td>
</tr>
</tbody>
</table>

### Subtract Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBA</td>
<td>Subtract B from A</td>
<td>(A) - (B) ⇒ A</td>
</tr>
<tr>
<td>SBCA</td>
<td>Subtract with borrow from A</td>
<td>(A) - (M) - C ⇒ A</td>
</tr>
<tr>
<td>SBCB</td>
<td>Subtract with borrow from B</td>
<td>(B) - (M) - C ⇒ B</td>
</tr>
<tr>
<td>SUBA</td>
<td>Subtract memory from A</td>
<td>(A) - (M) ⇒ A</td>
</tr>
<tr>
<td>SUBB</td>
<td>Subtract memory from B</td>
<td>(B) - (M) ⇒ B</td>
</tr>
<tr>
<td>SUBD</td>
<td>Subtract memory from D</td>
<td>(D) - (M:M+1) ⇒ D</td>
</tr>
</tbody>
</table>
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)

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

7. org $2000
8. ldaa #num
9. adda testval ; -50 + 60 = 10 = $0A
10. staa answer
11. swi
12. end
Simple Example Programs Cont’d.

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

8. org $2000
9. ldaa #num
10. ldx #testval
11. ldy #answer
12. adda 0,x
13. staa 0,y
14. swi
15. end
Next Slide:

- a list file for webex1a
- inset showing list of all (asm, lst, s19) files in directory
asm12, an absolute assembler for Motorola MCU's, version 1.2e

; webexia, 04/06
;     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 $0FF

2000 86 ce  ldas  #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

22 Executed: Fri Aug 04 14:51:47 2006
23 Total cycles: 24, Total bytes: 14
24 Total errors: 0, Total warnings: 1
Simple Example Programs Cont’d.

1. ;webex1b, 8/4/06, program adds (-50) to a number and stores result
2. ; (revision of webex1 with a test for invalid results:
3. ;   if result exceeds 8-bit limits for signed numbers
4. ;   store FF in location [valid], otherwise store 00)
5. ; Program also includes a conditional branch
6. num    equ   -50
7. testval db    60
8. answer rmb   1 ;reserve memory byte at answer
9. valid rmb   1
10. org    $2000
11. clr     valid ;make default 00
12. ld # num
13. adda testval
14. bvc good ;skip if results ok (checks overflow flag, table 2.2, p. 54)
15. com valid ;make FF
16. staa answer
17. swi
18. end
Simple Example Programs Cont’d.

1. ;webex1c, 9/13/06,; program adds -50 to a number and stores result
2. ; (revision of webex1 with a test for invalid results: see web1b for details)
3. ; and use of indexed w/ normal and auto incr.
4. num equ -50
5. org $1800
6. data db 60, -200
7. results rmb 2
8. valid rmb 1
9. org $2000
10. clr valid; make default 00
11. ldab #num
12. ldx #data
13. ldy #results
14. ld aa 1,x+
15. aba; add (-50) to first number
16. bvc good; skip if results ok (overflow flag clear)
17. movb #$FF,valid; otherwise ...
18. good staa 1,y+
19. l d aa 0,x
20. aba; add (-50) to second number
21. bvc good2; skip if results ok
22. movb #$FF,valid
23. good2 staa 0,y
24. swi
25. end