In This Set of Slides:

1. An Instruction’s (or Directive’s) Four Fields
2. Assembler Directives
3. SW Development Process
4. RISC vs. CISC Instruction Sets
5. Simple Arithmetic Programs
6. Using the carry flag
7. Multi-byte Addition & Subtraction
8. Multiplication and Division
9. Binary $\rightarrow$ BCD/ASCII
Assembly Language Program Structure

• Programs consist of:
  Assembler Directives, Instructions, Comments

• Instruction format has four fields:

  Label : Operation Operand ;Comment

  ↑↑ (optional but, if used, label doesn’t have to start in col. 1)
  Must be in col. 1 even if using a colon (some assemblers more flexible)
Examples of the four fields of an instruction

• **loop**   **adda** #$40 ; add 40 to accumulator A

(1) “loop” is a label, may use a colon as in loop: but must start in column 1
(2) “ADDA” is an instruction mnemonic
(3) “#$40” is the operand
(4) “;add #$40 to accumulator A” is a comment

• **movb**   0,X,0,Y ; memory to memory copy

(1) no label field,
(b) “movb” is an instruction mnemonic and cannot start in column 1
(c) “0,X,0,Y” is the operand field
(d) “; memory to memory copy” is a comment
Examples of the four fields of an assembler directive

• **count equ 25 ; assigns 25 to count**
  (1) “count” is a label and must start in column 1
  (2) “equ” is a directive mnemonic
  (3) “25” is the operand
  (4) “; assigns 25 to count” is a comment

• **org $1800 ; set location counter to $1800**
  (1) no label field,
  (b) “org” is a directive instruction mnemonic, and can’t start in col. 1
  (c) “$1800” is the operand field
  (d) “; set location …” is a comment
Assembler Directive Examples

• **dc.b (define constant byte), db (define byte), fcb (form constant byte)**
  - These three directives define the value of a byte or bytes that will be stored.
  - Often preceded by the **org** directive.
  - For example,

    ```
    org $800
    array       dc.b $11,$22,$33,$44       ;stores these numbers at $800 thru $803
    ```

• **dc.w (define constant word), dw (define word), fdb (form double bytes)**
  - These three directives define the value of a word or words that will be stored.
  - For example,

    ```
    org $900
    vec_tab       dc.w $1234, %11 1110 1111 11000 ;results below
    ```

<table>
<thead>
<tr>
<th>location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>09 00</td>
<td>12</td>
</tr>
<tr>
<td>09 01</td>
<td>34</td>
</tr>
<tr>
<td>09 02</td>
<td>3E</td>
</tr>
<tr>
<td>09 03</td>
<td>F4</td>
</tr>
</tbody>
</table>
Assembler Directive Examples cont’d.

• **fcc (form constant character)**
  - Used to define a string of characters (a message)
  - The first character (and the last character) is used as the delimiter and must be the same (usually “ ” )
  - The delimiter must not appear in the string and can’t be space
  - Each character is represented by its ASCII code.

**EXAMPLE**

```
org $1500

; storage shown below
```

```
greeting: “hello”
greeting: “hello"
```

- 15 00 $68 h
- 15 01 $65 e
- 15 02 $6C l
- 1503 $6C l
- 1504 $6F o
Assembler Directive Examples cont’d.

• **fill value, count**

```
org $1800
space_line: fill $20,40 ; fill 40 locations w/$20 starting at $1800
```

Zeros: fill 0 , 20 ;fill 20 locations w/0 starting at $1800 + 40
(can also store zeros using zmb XX or bsz XX)

• **ds (define storage), rmb (reserve memory byte), ds.b (define storage bytes)**

```
buffer ds 100 ; reserves 100 bytes
outbuf rmb 100 ; same (Preferred)
```

• **ds.w (define storage word), rmw (reserve memory word)**

```
dbuf ds.w 20 ;reserves 20 words (40 bytes)
lnbuf rmw 10 ;reserves 10 words (20 bytes)
```
Assembler Directive Examples cont’d.

• **equ (equate)**
  This directive assigns a value to a label and makes a program more readable.

```assembly
arr_cnt    equ    100     ; arr_cnt is now a constant with a value of decimal 100
oc_cnt     equ    50
```

• Loc discussed in book but **not** recommended for use
Assembler Directive Examples cont’d.

**Macro:** A name assigned to a group of instructions
Use `macro` and `endm` to define a macro.

**Example of Defining and Invoking a macro**

```
sumOf3 macro arg1,arg2,arg3 ; this line defines name and no. of arguments
ldaa arg1 ; these three lines are the actual code of macro
adda arg2
adda arg3
endm ; tells assembler this is the end of the macro
```

```
sumOf3 $1000,$1001,$1002 ; invoking the macro in the program
ldaa $1000 ;assembler replaces the invocation with these lines
adda $1001 ; when program is assembled
adda $1002
```

**Notes:** (1) each time macro is invoked, the assembler inserts the code
(2) compare and contrast a macro with a subroutine
Software Development Process

1. Problem definition: Identify what should be done.
   - Develop the algorithm.
     • Algorithm is the overall plan for solving the problem at hand.
     • Next is a step by step approach (or pseudo code) and/or flow chart

2. Convert the pseudo code or flowchart into programs.

3. Program testing
   - simulation (CodeWarrior)
   - Downloading into DeBug12 and execution

4. Program maintenance (revisions, additions, etc.)
Flowchart Symbols
(not particularly useful for large programs)

Figure 2.1 Flowchart symbols used in this book
RISC vs. CISC

• Reduced Instruction Set Computer (RISC)
  – Minimal instruction set for fast execution
  – PIC 16F877 has 35 instructions

• Complex Instruction Set Computer (CISC)
  – Number of instructions in hundreds
  – More complex, costly, but more flexible
  – HSC12 operand can be two bytes
    • The first byte of a two-byte opcode is always $18.
    • Thus, $2 \times (256) = 512$ possible instructions
Simple Arithmetic Programs
(actually using “snippets”: parts of programs that are not assembly ready)

Example 2.4  Write a program to add the values of memory locations at $1000, $1001, and $1002, and save the result at $1100.

Solution: noting we cannot add numbers in memory, following is the step-by-step pseudo code

Step 1
A ⇐ m[$1000]

Step 2
A ⇐ A + m[$1001]

Step 3
A ⇐ A + m[$1002]

Step 4
$1100 ⇐ A

The snippet is:

```
org $1500 ;start program at this location
ldaa $1000 ;assuming sum will not exceed 8 bits & numbers present in memory
adda $1001
adda $1002
staa $1100
end
```

EXERCISE: draw a flow chart for the above snippet (next slide has a revised program’s fc).
Simple Arithmetic Programs cont’d.

- Example 2.4A Revise ex. 2.4 to add contents of locations $1000$ and $1002$ and subtract contents of $1005$. The results are to be stored in location $1010$.

```
org  $1500
ldaa $1000
adda $1002
suba $1005
staa $1010
end
```

Figure 2.2 Logic flow of program 2.4
• **Example 2.6** Write a program to add two 16-bit numbers that are stored at $1000$-$1001$ and $1002$-$1003$ and store the sum at $1100$-$1101$.

**Solution:**

```
org  $1500
ldd $1000 ; D ← m[$1000:$1001]
add $1002 ; D ← [D] + [$1002:$1003]
std $1100 ; $1100:$1101 ← [D]
end
```

NOTE: MS Byte is in the **lower** address.

2A 15 + 49 E0 = 73 F5

Q: What if a carry was generated?
Using the carry flag

1. Located in bit 0 of the CCR register
2. Useful in multi-precision arithmetic
3. Will be set to 1 if the addition operation produces a carry, otherwise cleared
4. Set to 1 when the subtraction operation produces a borrow, otherwise cleared
5. Carry/borrow flag is affected by both 8-bit addition/subtraction (registers A or B gets result) and 16-bit addition/subtraction (register D gets result)
6. Carry can be included only in 8-bit addition/subtraction, therefore:
   - Add with carry or borrow available with A & B
   - Add with carry or borrow not available with D
7. Note that, because of # 5 and # 6, for multi-byte operations we usually start with D but then continue with A and B.
8. See examples on subsequent slides.
Multi-byte addition

Example 2.7 Write a program to add two 4-byte numbers that are stored at $1000-$1003 and $1004-$1007, and store the sum at $1010-$1013.

Solution: (Addition must start with the LSB and proceed toward MSB!)

1. org $1500
2. ldd $1002 ; add and save the least significant two bytes (words)
3. addd $1006 ;
4. std $1012 ;
   ; now we start working with one byte at a time while using the C flag
5. ldaa $1001 ; add and save the second most significant bytes
6. adca $1005 ;
7. staa $1011 ;
8. ldaa $1000 ; add and save the most significant bytes
9. adca $1004 ;
10. staa $1010 ;
11. end

Notes: (1) stdd (and staa) and lda instructions do not affect the carry flag, so we can depend on the fact that ‘C’ flag in line 6 still reflects condition created by line #3, etc.
(2) Create a table to show how contents of memory are affected
Multi-byte Subtraction

• **Example 2.8** Write a program to subtract the hex number stored at $1004$-$1007$ from the hex number stored at $1000$-$1003$ and save the result at $1100$-$1103$.

• **Solution:** (The subtraction starts from the LSBs and proceeds toward the MSBs.)

```
1. org $1500
2. ldd $1002 ; subtract and save the least significant two bytes
3. subd $1006 ; “
4. std $1102 ; “

; now we start working with one byte at a time while using the C flag

5. ldaa $1001 ; subtract and save the difference of the second to most
6. sbca $1005 ; significant bytes
7. staa $1001 ; “
8. ldaa $1000 ; subtract and save the difference of the most significant
9. sbca $1004 ; bytes
10. staa $1100 ; “
11. end
```

**Note:** recall that sta (and staa) and lda instructions do not affect the carry flag so ‘C’ flag in line 6 still reflects condition created by line #3, etc.
Multiplication and Division

[Pay Attention To: (a) signed/unsigned, (b) 8bit/16bit, (c) locations of factors and results]

Table 2.1 Summary of HCS12 multiply and divide instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>emul</td>
<td>unsigned 16 by 16 multiply</td>
<td>(D) × (Y) → Y:D</td>
</tr>
<tr>
<td>emuls</td>
<td>signed 16 by 16 multiply</td>
<td>(D) × (Y) → Y:D</td>
</tr>
<tr>
<td>mul</td>
<td>unsigned 8 by 8 multiply</td>
<td>(A) × (B) → A:B</td>
</tr>
<tr>
<td>ediv</td>
<td>unsigned 32 by 16 divide</td>
<td>(Y:D) ÷ (X) quotient → Y remainder → D</td>
</tr>
<tr>
<td>edivs</td>
<td>signed 32 by 16 divide</td>
<td>(Y:D) ÷ (X) quotient → Y remainder → D</td>
</tr>
<tr>
<td>fdiv</td>
<td>16 by 16 fractional divide</td>
<td>(D) ÷ (X) → X remainder → D</td>
</tr>
<tr>
<td>idiv</td>
<td>unsigned 16 by 16 integer divide</td>
<td>(D) ÷ (X) → X remainder → D</td>
</tr>
<tr>
<td>idivs</td>
<td>signed 16 by 16 integer divide</td>
<td>(D) ÷ (X) → X remainder → D</td>
</tr>
</tbody>
</table>
Multiplication and Division cont’d.
(examples using actual numbers on slide 22 & 23)

- **Example 2.10** Write a snippet to multiply the 16-bit numbers stored at $1000$-$1001$ and $1002$-$1003$ and store the 32-bit product at $1100$-$1103$.

  **Solution:**
  
  ```
  ldd $1000  ; load first word
  ldy $1002  ; load second word
  emul        ; Y:D ← D*Y
  sty $1100  ; store MSW at $1100:$1101
  std $1102  ; store LSW at $1102:$1103
  ```

- **Example 2.11** Write a snippet to divide the 16-bit number stored at $1020$-$1021$ into the 16-bit number stored at $1005$-$1006$ and store the 16-bit quotient and 16-bit remainder at $1100$ and $1102$, respectively.

  **Solution:**
  
  ```
  ldd $1005
  ldx $1020
  idiv        ; X ← D/X and D ← Rem
  stx $1100  ; store the quotient at $1100:$1101
  std $1102  ; store the remainder at $1102:$1103
  ```
Multiplication and Division cont’d.
(examples using actual numbers on slide 22 & 23)

• **Example 2.10A** Write an instruction sequence (snippet) to multiply the **signed** 16-bit numbers stored at $1000$-$1001$ and $1002$-$1003$ and store the 32-bit product at $1100$-$1103$.

  **Solution:**
  
  ```
  ldd $1000 ; load first word
  ldy $1002 ; load second word
  emuls ; Y:D ← D*Y (Only change)
  sty $1100 ; store MSW at $1100:$1101
  std $1102 ; store LSW at $1102:$1103
  ```

• **Example 2.11A** Write a snippet to divide the **signed** 16-bit number stored at $1020$-$1021$ into the **signed** 16-bit number stored at $1005$-$1006$ and store the 16-bit quotient and 16-bit remainder at $1100$ and $1102$, respectively.

  **Solution:**
  
  ```
  ldd $1005
  ldx $1020
  idivs ; X ← D/X and D ← Rem (Only change)
  stx $1100 ; store the quotient at $1100:$1101
  std $1102 ; store the remainder at $1102:$1103
  ```
Multiplication and Division cont’d.

Complete the following table for examples 2.10 and 2.10A and discuss in terms of decimal numbers

<table>
<thead>
<tr>
<th>Example 2.10</th>
<th>Before</th>
<th>After</th>
<th>Example 2.10A</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1000</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td>FF</td>
</tr>
<tr>
<td>$1001</td>
<td>D8</td>
<td></td>
<td></td>
<td></td>
<td>D8</td>
</tr>
<tr>
<td>$1002</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td>FF</td>
</tr>
<tr>
<td>$1003</td>
<td>FD</td>
<td></td>
<td></td>
<td></td>
<td>FD</td>
</tr>
<tr>
<td>$1100</td>
<td>XX</td>
<td></td>
<td></td>
<td></td>
<td>XX</td>
</tr>
<tr>
<td>$1101</td>
<td>XX</td>
<td></td>
<td></td>
<td></td>
<td>XX</td>
</tr>
<tr>
<td>$1102</td>
<td>XX</td>
<td></td>
<td></td>
<td></td>
<td>XX</td>
</tr>
<tr>
<td>$1103</td>
<td>XX</td>
<td></td>
<td></td>
<td></td>
<td>XX</td>
</tr>
</tbody>
</table>
Multiplication and Division cont’d.

Complete the following table for examples 2.11 and 2.11A and discuss in terms of decimal numbers

<table>
<thead>
<tr>
<th>Example 2.11</th>
<th>Before</th>
<th>After</th>
<th>Example 2.11A</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1005</td>
<td>FF</td>
<td></td>
<td>FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1006</td>
<td>D8</td>
<td></td>
<td>D8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1020</td>
<td>00</td>
<td></td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1021</td>
<td>FD</td>
<td></td>
<td>FD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1100</td>
<td>XX</td>
<td></td>
<td>XX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1101</td>
<td>XX</td>
<td></td>
<td>XX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1102</td>
<td>XX</td>
<td></td>
<td>XX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1103</td>
<td>XX</td>
<td></td>
<td>XX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BCD

• **Binary Coded Decimal**
  – Useful in I/O operations
  – Cumbersome in arithmetic operations (only addition is worthwhile)
  – Each decimal digit is replaced by a four digit binary value, usually two packed into a byte

  – Example:
    • Given the decimal number 57
    • In packed BCD: 0101 0111 or $57
    • In binary: 0011 1001 or $39
BCD Numbers and Addition

- Two 4-bit digits are packed into one byte
- The addition of two BCD numbers requires binary addition and the DAA instruction, which makes use of the H flag.
- DAA can be applied after the instructions ADDA, ADCA, and ABA.
- Simplifies I/O conversion
- For example, the instruction sequence
  - LDAA $1000 ;get first packed BCD number
  - ADDA $1001 ;add the second BCD number
  - DAA ;adjust for errors created
  - STAA $1002 ;store new packed BCD sum
- Show what happens for the case of 27 + 45
Converting a Binary Number to BCD/ASCII

- **Algorithm**
  - Use repeated division by 10, saving the remainders to form the BCD digits, but also keeping each new quotient for the next division.
  - The first division generates the LSD, the second division by 10 obtains the second LSD, and so on.
  - The largest 16-bit binary number is 65535 which has five decimal digits.
  - Add $30 to each BCD digit forming its ASCII value.
### Code for Binary → BCD/ASCII (1st part)

#### Example 2.13: Convert a 16 bit number (in \$1000~\$1001\) to BCD and store in 5 bytes at \$1010~\$1014.

1. org $1000
2. data dc.w 12345 ; data to be tested
3. org $1010
4. result ds.b 5 ; reserve bytes to store the result
5. org $1500
6. ldd data
7. ldy #result
8. ldx #10
9. idiv ;X ← D/10, D ← Rem (won’t exceed B, thus
10. addb #$30 ; convert first digit into ASCII code
11. stab 4,Y ; save the least significant digit
12. xgdx ; get new quotient into D
13. ldx #10

**Note** the offset for Y, so that we end up with MSD at lowest address.
Code for Binary $\rightarrow$ BCD/ASCII (2\textsuperscript{nd} part)

1. idiv ; create second digit
2. adcb #$30 ;
3. stab 3,Y ; save the second to least significant digit
4. xgdx
5. ldx #10
6. idiv
7. addb #$30
8. stab 2,Y ; save the middle digit
9. xgdx
10. ldx #10
11. idiv
12. addb #$30
13. stab 1,Y ; save the second most significant digit
14. xgdx
15. addb #$30
16. stab 0,Y ; save the most significant digit
17. end
Example of Binary → BCD

Note: 12345 = $3039

<table>
<thead>
<tr>
<th>Location</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1000</td>
<td>$30</td>
<td></td>
</tr>
<tr>
<td>$1001</td>
<td>$39</td>
<td></td>
</tr>
<tr>
<td>$1010</td>
<td>XX</td>
<td></td>
</tr>
<tr>
<td>$1011</td>
<td>XX</td>
<td></td>
</tr>
<tr>
<td>$1012</td>
<td>XX</td>
<td></td>
</tr>
<tr>
<td>$1013</td>
<td>XX</td>
<td></td>
</tr>
<tr>
<td>$1014</td>
<td>XX</td>
<td></td>
</tr>
</tbody>
</table>