Lecture 9
February 6, 2012

Writing Assembly Language Programs

• Use flow charts to lay out structure of program
• Use common flow structures
  – if-then
  – if-then-else
  – do-while
  – while
• Plan structure of data in memory
• Top-down Design
  – Plan overall structure of program
  – Work down to more detailed program structure
  – Implement structure with instructions
• Optimize program to make use of instruction efficiencies
• Do not sacrifice clarity for efficiency or speed

Input and Output Ports

• How to get data into and out of the MC9S12
Example Program: Divide a table of data by 2

Problem: Start with a table of data. The table consists of 5 values, with the first value at $1000. Each value is between 0 and 255. Create a new table whose contents are the original table divided by 2. Start the new table immediately after the original table.

1. Determine where code and data will go in memory.
   Code at $2000, data at $1000.

2. Determine type of variables to use.
   Because data will be between 0 and 255, can use unsigned 8-bit numbers.

3. Draw a picture of the data structures in memory:
4. Strategy: Because we are using a table of data, we will need pointers to each table so we can keep track of which table element we are working on.
   Use the X and Y registers as pointers to the tables.
5. Use a simple flow chart to plan structure of program.
6. Need a way to determine when we reach the end of the table. One way: Use a counter (say, register A) to keep track of how many elements we have processed.
7. Add code to implement blocks:

```
START
  - Init Counter
    - LDAA #COUNT
  - Init Pointers
    - LDX #TABLE1
    - LDY #TABLE2
  - Get Entry
    - LDAB 0,X
    - STAB 0,Y
    - INX
    - INY
    - DECA
    - LSRB ; unsigned divide
  - Divide by 2
  - Store Result
    - STAB 0,Y
  - Inc Pointers
    - INX
    - INY
  - Dec Counter
    - DECA
  - More?
    - YES
      - More?
    - NO
      - STOP
  - BNE L1
STOP
```
8. Write program:

; Program to divide a table by two
; and store the results in memory

prog:   equ $2000
data:   equ $1000

count:  equ 5

org    prog ; set program counter to 0x1000
lda    #count ; Use A as counter
ldx    #table1 ; Use X as data pointer to table1
ldy    #table2 ; Use Y as data pointer to table2
l1:    ldab 0,x ; Get entry from table1
lsrb   ; Divide by two (unsigned)
stab   0,y   ; Save in table2
inx     ; Increment table1 pointer
iny     ; Increment table2 pointer
deca    ; Decrement counter
bne    l1   ; counter != 0 => more entries to divide
swi     ; Done

org    data

table1: dc.b $07,$c2,$3a,$68,$f3
table2: ds.b count
9. Advanced: Optimize program to make use of instructions set efficiencies:

; Program to divide a table by two
; and store the results in memory

prog:   equ   $1000
data:   equ   $2000

count:   equ   5

org     prog   ;set program counter to 0x1000
lda     #count   ;Use B as counter
ldx     #table1   ;Use X as data pointer to table1
ldy     #table2   ;Use Y as data pointer to table2
l1:     ldab  1,x+   ;Get entry from table1; then inc pointer
        lsrb
        stab 1,y+   ;Save in table2; then inc pointer
        dbne a,l1   ;Decrement counter; if not 0, more to do
        swi

org     data

table1: dc.b  $07,$c2,$3a,$68,$f3

table2: ds.b  count
TOP-DOWN PROGRAM DESIGN

- PLAN DATA STRUCTURES IN MEMORY
- START WITH A LARGE PICTURE OF PROGRAM STRUCTURE
- WORK DOWN TO MORE DETAILED STRUCTURE
- TRANSLATE STRUCTURE INTO CODE
- OPTIMIZE FOR EFFICIENCY —
  DO NOT SACRIFICE CLARITY FOR EFFICIENCY
Input and Output Ports

• How do you get data into a computer from the outside?

**SIMPLIFIED INPUT PORT**

Any read from address $0000$
gets signals from outside

`LDA A $00`

Puts data from outside
into accumulator A.

Data from outside looks
like a memory location

A Tri-State Buffer acts like a switch

If TRI is active, the switch is closed
OUT will be the same as IN

If TRI is not active, the switch is open
OUT will not be driven by IN
Some other device can drive OUT
• How do you get data out of computer to the outside?

**SIMPLIFIED OUTPUT PORT**

Any write to address $01 latches data into flip-flops, so data goes to external pins

```
MOV #$AA,$01
```

goes $AA on the external pins

When a port is configured as output and you read from that port, the data you read is the data which was written to that port:

```
MOV #$AA,$01
LDAA $01
```

Accumulator A will have $AA after this
• Most I/O ports on MC9S12 can be configured as either input or output

SIMPLIFIED INPUT/OUTPUT PORT

A write to address 0x0000 writes data to the flip-flop
A read from address 0x0000 reads data on pin

If Bit 7 of DDRA is 0, the port
is an input port. Data written to
flip-flop does not get to pin
through tri-state buffer

If Bit 7 of DDRA is 1, the port
is an output port. Data written to
flip-flop does get to pin
through tri-state buffer

DDRA (Data Direction Register A) is located at 0x0002
Figure 1-1 MC9S12DP256B Block Diagram
Ports on the MC9S12

- How do you get data out of computer to the outside?

- A Port on the MC9S12 is a device the MC9S12 uses to control some hardware.

- Many of the MC9S12 ports are used to communicate with hardware outside of the MC9S12.

- The MC9S12 ports are accessed by the MC9S12 by reading and writing memory locations $0000$ to $03FF$.

- Some of the ports we will use in this course are PORTA, PORTB, PTJ and PTP

  - 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-Plus EVB, eight LEDs and four seven-segment LEDs are connected to PTB.
  
  - Before you can use the eight individual LEDs or the seven-segment LEDs, you need to enable them.

  - Bit 1 of PTJ must be low to enable the eight individual LEDs
    * To make Bit 1 of PTJ low, you must first make Bit 1 of PTJ an output by writing a 1 to Bit 1 of DDRJ.
    * Next, write a 0 to Bit 1 of PTJ.

  - Bits 3-0 of PTP are used to enable the four seven-segment LEDs
To use the seven-segment LEDs, first write 1’s to Bits 3-0 of DDRP to make Bits 3-0 of PTP outputs.

* A low PTP0 enables the left-most (Digit 3) seven-segment LED
* A low PTP1 enables the second from the left (Digit 2) seven-segment LED
* A low PTP2 enables the third from the left (Digit 1) seven-segment LED
* A low PTP3 enables the right-most (Digit 0) seven-segment LED

To use the eight individual LEDs and turn off the seven-segment LEDs, write ones to Bits 3-0 of PTP, and write a 0 to Bit 1 of PTJ:

\[
\begin{align*}
\text{BSET} & \quad \text{DDRP},#\$0F \quad ; \quad \text{Make PTP3 through PTP0 outputs} \\
\text{BSET} & \quad \text{PTP},#\$0F \quad ; \quad \text{Turn off seven-segment LEDs} \\
\text{BSET} & \quad \text{DDRJ},#\$02 \quad ; \quad \text{Make PTJ1 output} \\
\text{BCLR} & \quad \text{PTJ},#\$02 \quad ; \quad \text{Turn on individual LEDs}
\end{align*}
\]

- On the DRAGON12-Plus EVB, the LCD display is connected to PTK
- When you power up or reset the MC9S12, PORTA, PORTB, PTJ and PTP are input ports.
- You can make any or all bits of PORTA, PORTB, PTP and PTJ outputs by writing a 1 to the corresponding bits of their Data Direction Registers.

- You can use DBug-12 to manipulate the IO ports on the MC9S12.
  * To make PTB an output, use MM to change the contents of address $0003 \text{ (DDRB)} to an $FF$.
  * You can now use MM to change contents of address $0001 \text{ (PORTB)}$, which changes the logic levels on the PORTB pins.
  * If the data direction register makes the port an input, you can use MD to display the values on the external pins.
Using Port A of the MC9S12

To make a bit of Port A an output port, write a 1 to the corresponding bit of DDRA (address 0x0002). To make a bit of Port A an input port, write a 0 to the corresponding bit of DDRA.

On reset, DDRA is set to $00, so Port A is an input port.

<table>
<thead>
<tr>
<th>DDA7</th>
<th>DDA6</th>
<th>DDA5</th>
<th>DDA4</th>
<th>DDA3</th>
<th>DDA2</th>
<th>DDA1</th>
<th>DDA0</th>
<th>$0002</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0002</td>
</tr>
</tbody>
</table>

RESET 0 0 0 0 0 0 0 0

For example, to make bits 3–0 of Port A input, and bits 7–4 output, write a 0xf0 to DDRA. To send data to the output pins, write to PORTA (address 0x0000). When you read from PORTA input pins will return the value of the signals on them (0 => 0V, 1 => 5V); output pins will return the value written to them.

<table>
<thead>
<tr>
<th>PA7</th>
<th>PA6</th>
<th>PA5</th>
<th>PA4</th>
<th>DP3</th>
<th>PA2</th>
<th>PA1</th>
<th>PA0</th>
<th>$0000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000</td>
</tr>
</tbody>
</table>

RESET _ _ _ _ _ _ _ _

Port B works the same, except DDRB is at address 0x0003 and PORTB is at address 0x0001.
; A simple program to make PORTA output and PORTB input,
; then read the signals on PORTB and write these values
; out to PORTA

prog:       equ  $1000
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

• Because DDRA and DDRB are in consecutive address locations, you could
make PORTA and output and PORTB and input in one instruction:

movw  #$ff00,DDRA ; FF -> DDRA, 00 -> DDRB
GOOD PROGRAMMING STYLE

1. Make programs easy to read and understand.
   • Use comments
   • Do not use tricks

2. Make programs easy to modify
   • Top-down design
   • Structured programming – no spaghetti code
   • Self contained subroutines

3. Keep programs short BUT do not sacrifice items 1 and 2 to do so

TIPS FOR WRITING PROGRAMS

1. Think about how data will be stored in memory.
   • Draw a picture

2. Think about how to process data
   • Draw a flowchart

3. Start with big picture. Break into smaller parts until reduced to individual instructions
   • Top-down design

4. Use names instead of numbers
Another Example of an Assembly Language Program

- Find the average of the numbers in an array of data.
- The numbers are 8-bit unsigned numbers.
- The address of the first number is $E000$ and the address of the final number is $E01F$. There are 32 numbers.
- Save the result in a variable called `answer` at address $2000$.

Start by drawing a picture of the data structure in memory:

```
FIND AVERAGE OF NUMBERS IN ARRAY FROM 0xE000 TO 0xE01f
Treat numbers as 8-bit unsigned numbers
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>1</th>
<th>8</th>
<th>6</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Start with the big picture

FIND AVERAGE OF 8-BIT NUMBERS IN ARRAY FROM 0xE000 TO 0xE01f

```
START
Init
Process Entries
Save Answer
Done
```

```
0xE000
4
5
1
8
6
11

0xE01F
```
Add details to blocks

SUM ODD 8-BIT NUMBERS IN ARRAY FROM 0xE000 TO 0xE01f

START
Init
Addr -> Pointer
Process Entries
0 -> Sum
Save Answer
Done

4
5
1
8
6
11

0xE000

0xE01F
Decide on how to use CPU registers for processing data

**FIND AVERAGE OF 8-BIT NUMBERS IN ARRAY FROM 0xE000 TO 0xE01f**

- **Pointer:** X or Y -- use X
- **Sum:** 16-bit register
  - D or Y
  - No way to add 8-bit number to D
  - Can use ABY to add 8-bit number to Y
FIND AVERAGE OF 8-BIT NUMBERS IN ARRAY FROM 0xE000 TO 0xE01f
More details: How to tell when program reaches end of array

FIND AVERAGE OF 8-BIT NUMBERS IN ARRAY FROM 0xE000 TO 0xE01f

How to check if more to do?  
If X < 0x0E20, more to do. 

BLT or BLO?  
Addresses are unsigned, so BLO

How to find average?  Divide by LEN 
To divide, use IDIV

TFR Y,D ; dividend in D
LDX #LEN ; divisor in X
IDIV

<table>
<thead>
<tr>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addresses</th>
<th>0xE000</th>
<th>0xE01F</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Convert blocks to assembly code

FIND AVERAGE OF 8-BIT NUMBERS IN ARRAY FROM 0xE000 TO 0xE01F

START
  Init
    Addr -> Pointer
    LDY #0
  Process Entries
    0 -> Sum
    LDX #ARRAY
    Find Average
    Save Answer
    Done

Process Entries
  Get Num
    LDAB 0,X
    ABY
  Add Num to Sum
    INX
  Inc Pointer
    CMPX #ARRAY_END
    BLO loop
    Done

std Answer
  Save Answer
  Done

TFR Y,D
LDX #LEN
IDIV

X ->

4
5
1
8
6
11

0xE000 ARRAY

0xE01F ARRAY_END
Write program

;Program to average 32 numbers in a memory array

prog:   equ $2000
data:   equ $1000
array:  equ $E000
len:    equ 32

org prog

ldx #array ; initialize pointer
ldy #0 ; initialize sum to 0
loop:  ldab 0,x ; get number
aby ; odd - add to sum
inx ; point to next entry
cpx #(array+len) ; more to process?
blo loop ; if so, process

tfr y,d ; To divide, need dividend in D
ldx #len ; To divide, need divisor in X
idiv ; D/X quotient in X, remainder in D
stx answer ; done -- save answer
swi

org data

answer: ds.w 1 ; reserve 16-bit word for answer

• Important: Comment program so it is easy to understand.
The assembler output for the above program

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

<table>
<thead>
<tr>
<th>Abs.</th>
<th>Rel.</th>
<th>Loc</th>
<th>Obj. code</th>
<th>Source line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>;Program to average 32 numbers in a memory array</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0000</td>
<td>2000</td>
<td>prog: equ $2000</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0000</td>
<td>1000</td>
<td>data: equ $1000</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0000</td>
<td>E000</td>
<td>array: equ $E000</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0000</td>
<td>0020</td>
<td>len: equ 32</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td></td>
<td></td>
<td>org prog</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>a002000</td>
<td>CEE0 00</td>
<td>ldx   #array  ; initialize pointer</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>a002003</td>
<td>CD00 00</td>
<td>ldy   #0  ; initialize sum to 0</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>a002006</td>
<td>E600</td>
<td>loop:  ldab 0,x  ; get number</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>a002008</td>
<td>19ED</td>
<td>aby  ; odd - add to sum</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>a00200A</td>
<td>08</td>
<td>inx  ; point to next entry</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>a00200B</td>
<td>8EE0 20</td>
<td>cpx  #(array+len)  ; more to process?</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>a00200E</td>
<td>25F6</td>
<td>blo  loop  ; if so, process</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>a002010</td>
<td>B764</td>
<td>tfr  y,d  ; To divide, need dividend</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>a002012</td>
<td>CE00 20</td>
<td>ldx   #len  ; To divide, need divisor</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>a002015</td>
<td>1810</td>
<td>idiv  ; D/X quotient in X, rem</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>a002017</td>
<td>7E10 00</td>
<td>stx   answer  ; done -- save answer</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>a00201A</td>
<td>3F</td>
<td>swi</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td></td>
<td></td>
<td>org  data</td>
</tr>
<tr>
<td>26</td>
<td>26</td>
<td>a001000</td>
<td></td>
<td>answer: ds.w 1  ; reserve 16-bit word for</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And here is the .s19 file:

```
S11E2000CE000CD0000E60019ED088EE02025F6B764CE002018107E10003FAB
S9030000FC
```