Lecture 12
February 13, 2012

More on using the Stack and the Stack Pointer
Introduction to Programming the MC9S12 in C

• Examples of using the stack
• Including "derivative.inc" in an assembly language program
• Using a mask in an assembly language program
• Using the DIP switches on the Dragon12
• Putting a program into the MC9S12 EEPROM
• Displaying patterns from a table on the Dragon12 LEDs
• Comparison of C and Assembly language programs
Examples of Using the Stack

Consider the following:

2000          org     $2000
2000 cf 20 00  lds     #$2000
2003 ce 01 23  ldx     #$0123
2006 cc ab cd  ldd     #$abcd
2009 34        pshx
200a 36        psha
200b 37        pshb
200c 07 04     bsr     delay
200e 33        pulb
200f 32        pula
2010 30        pulx
2011 3f        swi

2012 34  delay: pshx
2013 ce 03 e8  ldx #1000
2016 04 35 fd  loop: dbne x,loop
2019 30        pulx
201a 3d        rts

The following does not work; the RTS goes to the wrong place

2000          org     $2000
2000 cf 20 00  lds     #$2000
2003 ce 01 23  ldx     #$0123
2006 cc ab cd  ldd     #$abcd
2009 34        pshx
200a 36        psha
200b 37        pshb
200c 07 04     bsr     delay
200e 33        pulb
200f 32        pula
2010 30        pulx
2011 3f        swi

2012 34  delay: pshx
2013 ce 03 e8  ldx #1000
2016 04 35 fd  loop: dbne x,loop
2019 3d        rts
Using Registers in Assembly Language

- The DP256 version of the MC9S12 has lots of hardware registers
- To use a register, you can use something like the following:

  PORTB   equ $0001

- It is not practical to memorize the addresses of all the registers
- Better practice: Use a file which has all the register names with their addresses

```
#include "derivative.inc"
```

- Here is some of derivative.inc

```c
;*** PORTA - Port A Register; 0x00000000 ***
PORTA: equ $0000 ;*** PORTA - Port A Register; 0x0000 ***
;*** PORTB - Port B Register; 0x0001 ***
PORTB: equ $0001 ;*** PORTB - Port B Register; 0x0001 ***
;*** DDRA - Port A Data Direction Register; 0x0002 ***
DDRA: equ $0002 ;*** DDRA - Port A Data Direction Register; 0x0002 ***
;*** DDRB - Port B Data Direction Register; 0x0003 ***
DDRB: equ $0003 ;*** DDRB - Port B Data Direction Register; 0x0003 ***
```
Using DIP switches to get data into the MC9S12

- DIP switches make or break a connection (usually to ground)

**DIP Switches on Breadboard**
• To use DIP switches, connect one end of each switch to a resistor
• Connect the other end of the resistor to +5 V
• Connect the junction of the DIP switch and the resistor to an input port on the MC9S12
• The Dragon12-Plus has eight dip switches which are already connected to Port H (PTH).
• The four least significant bits of PTH are also connected to push-button switches.
  – If you want to use the push-button switches, make sure the DIP switches are in the OFF position.

• When the switch is open, the input port sees a logic 1 (+5 V)
• When the switch is closed, the input sees a logic 0 (0.22 V)
Looking at the state of a few input pins

- Want to look for a particular pattern on 4 input pins
  - For example want to do something if pattern on PH3-PH0 is 0110
- Don’t know or care what are on the other 4 pins (PH7-PH4)
- Here is the wrong way to do it:
  
  ```
  ldaa PTH
  cmpa #$06
  beq task
  ```

  - If PH7-PH4 are anything other than 0000, you will not execute the task.
- You need to mask out the Don’t Care bits before checking for the pattern on the bits you are interested in
  
  - To mask out don’t care bits, AND the bits with a mask which has 0’s in the don’t care bits and 1’s in the bits you want to look at.
  
  ```
  ldaa PTH
  anda #$0F
  cmpa #$06
  beq task
  ```

- Now, whatever pattern appears on PH7-4 is ignored
Using an MC9S12 output port to control an LED

- Connect an output port from the MC9S12 to an LED.

**Using an output port to control an LED**

When a current flows through an LED, it emits light.

Resistor, LED, and ground connected internally inside breadboard
Making a pattern on a seven-segment LED

- Want to generate a particular pattern on a seven-segment LED:

- Determine a number (hex or binary) which will generate each element of the pattern
  - For example, to display a 0, turn on segments a, b, c, d, e and f, or bits 0, 1, 2, 3, 4 and 5 of PTB. The binary pattern is 00111111, or $3f$.
  - To display 0 2 4 6 8, the hex numbers are $3f$, $5b$, $66$, $7d$, $7f$.

- Put the numbers in a table
- Go through the table one by one to display the pattern
- When you get to the last element, repeat the loop
Flowchart to display a pattern of lights on a set of LEDs

```
START
ldaa #$ff
staa DDRB
ldx #table
ldaa 0,x
staa PORTB

11: Point to first entry

12: Get entry

Output to PORTB

Wait

Inc Pointer

X < end?
```

```
PORTB Output

Table:

0x3f
0x5b
0x66
0x7d
0x7f
```

```
lx #table
ldaa 0,x
staa PORTB

bsr delay

inx

cpx #table_end
bra 11
```
as12, an absolute assembler for Motorola MCU’s, version 1.2h

; Program to display a pattern on a seven-segment LED display

#include "hcs12.inc"

2000 prog: equ $2000
1000 data: equ $1000
2000 stack: equ $2000

0005 table_len: equ (table_end-table)

2000 org prog

2000 cf 20 00 lds #stack ; initialize stack pointer
2003 86 ff ldaa #$ff ; Make PORTB output
2005 5a 03 staa DDRB ; 0xFF -> DDRB
2007 ce 10 00 l1: ldx #table ; Start pointer at table
200a a6 00 l2: ldaa 0,x ; Get value
200c 5a 01 staa PORTB ; Update LEDs
200e 07 08 bsr delay ; Wait a bit
2010 08 inx ; point to next
2011 8e 10 05 cpx #table_end ; More to do?
2014 25 f4 blo 12 ; Yes, keep going through table
2016 20 ef bra l1 ; At end; reset pointer

2018 36 delay: psha
2019 34 pshx
201a 86 64 ldaa #100
201c ce 1f 40 loop2: ldx #8000
201f 04 35 fd loop1: dbne x,loop1
2022 04 30 f7 dbne a,loop2
2025 30 pulx
2026 32 pula
2027 3d rts

1000 org data

1000 3f table: dc.b $3f
1001 5b dc.b $5b
1002 66 dc.b $66
1003 7d dc.b $7d
1004 7f dc.b $7F
1005 table_end:
Putting a program into EEPROM on the Dragon12-Plus

- EEPROM from 0x400 to 0xFFF
- Program will stay in EEPROM memory even after power cycle
  - Data will not stay in RAM memory
- If you put the above program into EEPROM, then cycle power, you will display a sequence of patterns on the seven-segment LED, but the pattern will be whatever junk happens to be in RAM
- To make sure you retain your patterns, put the table in the text part of your program, not the data part
- If you use a variable which needs to be stored in data, be sure you initialize that variable in your program and not by using \texttt{dc.b}.
- The Dragon12 board uses an 8 MHz clock. The MC9S12 has an internal phase-locked loop which can change the clock speed. DBug12 increases the clock speed from 8 MHz to 48 MHz. When you run a program from EEPROM, DBug12 does not run, so your program will run six times slower than it would using DBug12. The Lab has instructions on how to increase the MC9S12 clock from 8 MHz to 48 MHz so your program will run with the same speed as under DBug12.
• Here is the above program with table put into EEPROM

• Also, I have included a variable var which I initialize to $aa in the program
  
  – I don’t use var in the program, but included it to show you how to use a RAM-based variable

```
#include "hcs12.inc"
prog: equ $0400
data: equ $1000
stack: equ $2000

table_len: equ (table_end-table)

org prog

lds #stack ; initialize stack pointer
moveb #$aa,var ; initialize var
ldaa #$ff ; Make PORTB output
staa DDRB ; 0xFF -> DDRB

l1: ldx #table ; Start pointer at table
l2: ldaa 0,x ; Get value
staa PORTB ; Update LEDs
brr delay ; Wait a bit
inx ; point to next
cpx #table_end ; More to do?
blo 12 ; Yes, keep going through table
bra 11 ; At end; reset pointer

delay: psha
pshx
ldaa #100

loop2: ldx #8000
loop1: dbne x,loop1
dbne a,loop2
pulx
pula
rts

table: dc.b $3f
dc.b $5b
dc.b $66
dc.b $7d
dc.b $7F

org data

var: ds.b 1 ; Reserve one byte for var
Programming the MC9S12 in C

• A comparison of some assembly language and C constructs

<table>
<thead>
<tr>
<th>Assembly</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>; Use a name instead of a num</td>
<td>/* Use a name instead of a num */</td>
</tr>
<tr>
<td>COUNT: EQU 5</td>
<td>#define COUNT 5</td>
</tr>
<tr>
<td>; start a program</td>
<td>/* To start a program */</td>
</tr>
<tr>
<td>org $2000</td>
<td>main()</td>
</tr>
<tr>
<td>lds #0x2000</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

• Note that in C, the starting location of the program is defined when you compile the program, not in the program itself.

• Note that C always uses the stack, so C automatically loads the stack pointer for you.

<table>
<thead>
<tr>
<th>Assembly</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>; allocate two bytes for</td>
<td>/* Allocate two bytes for</td>
</tr>
<tr>
<td>; a signed number</td>
<td>* a signed number */</td>
</tr>
<tr>
<td>i: ds.w 1</td>
<td>int i;</td>
</tr>
<tr>
<td>j: dc.w $1A00</td>
<td>int j = 0x1a00;</td>
</tr>
<tr>
<td>; allocate two bytes for</td>
<td>/* Allocate two bytes for</td>
</tr>
<tr>
<td>; an unsigned number</td>
<td>* an unsigned number */</td>
</tr>
<tr>
<td>i: ds.w 1</td>
<td>unsigned int i;</td>
</tr>
<tr>
<td>j: dc.w $1A00</td>
<td>unsigned int j = 0x1a00;</td>
</tr>
<tr>
<td>; allocate one byte for</td>
<td>/* Allocate one byte for</td>
</tr>
<tr>
<td>; an signed number</td>
<td>* an signed number */</td>
</tr>
<tr>
<td>i: ds.b 1</td>
<td>signed char i;</td>
</tr>
<tr>
<td>j: dc.b $1F</td>
<td>signed char j = 0x1f;</td>
</tr>
</tbody>
</table>
• In C, the construct *(num) says to treat num as an address, and to work with the contents of that address.

• Because C does not know how many bytes from that address you want to work with, you need to tell C how many bytes you want to work with. You also have to tell C whether you want to treat the data as signed or unsigned.

  – i = *(unsigned char *) 0xE000; tells C to take one byte from address 0xE000, treat it as unsigned, and store that value in variable i.
  – j = *(int *) 0xE000; tells C to take two bytes from address 0xE000, treat it as signed, and store that value in variable j.
  – *(char *) 0xE000 = 0xaa; tells C to write the number 0xaa to a single byte at address 0xE000.
  – *(int *) 0xE000 = 0x00aa; tells C to write the number 0x00aa to two bytes starting at address 0xE000.
Here is a simple program written in C and assembly. It simply divides 16 by 2. It does the division in a function.

```
org $1000
i: ds.b 1

org $1000
main()
lds #$2000
ldaa #16
jsr div
staa i
swi

div: asra
rts

signed char i;
signed char div(signed char j);

i = div(16);
```