CPE 390 Microprocessor Systems

Lab1: Introduction to the EVB

This lab is intended for the first-time user of the CML-12C32 evaluation board (referred to as the EVB). The setup and operation of the EVB is explained. An introduction to the D-Bug12 monitor program and its memory and register management commands is provided.

1.1 Equipment

- CML-12C32 evaluation board containing D-Bug12 monitor program in Flash EEPROM. **(Please do not change any settings on the EVB. Alert your TA if you need assistance with the board)**
- Personal Computer containing the MiniIDE Integrated Development Environment software
- Power Supply

1.2 Introduction

The EVB contains a Freescale MC9S12C32 microcontroller and associated I/O and memory devices. The EVB allows the user to easily test systems and software designed for the MC9S12 microcomputer. The board includes Flash EEPROM containing the D-Bug12 monitor program. The monitor program begins executing on power-up or reset. This program controls communication between the EVB and the serial port on the PC. It allows the user to examine or modify the contents of memory. In conjunction with the MiniIDE software, it allows a user to assemble, debug and execute programs.

1.3 EVP Memory Map

Address	Memory Type	Application
$C000 - $FFFF	FLASH EEPROM	D-Bug12 and utility firmware
$8000 - $BFFF	FLASH EEPROM	User Autostart Program Memory Space
$1000 - $7FFF	External RAM	Expanded Mode User Program Memory
$0F80 - $0FFF	Internal RAM	RAM interrupt vector table
$0E00 - $0F8B	Internal RAM	Monitor reserved RAM memory
$0800 - $0DFF	Internal RAM	User internal RAM memory
$0400 - $04FB	External Memory	Expended Mode Development memory
$0000 - $03FF	HCS12 registers	Internal control and I/O port registers

All memory locations are expressed as 16-bit hexadecimal numbers. Memory values are expressed as 8-bit hexadecimal numbers. The D-Bug12 monitor software resides in FLASH EEPROM in the top 16K bytes of memory space. It is recommended that you use memory locations $4000 - $7FFF for your programs and data.

2. D-Bug12 Monitor Environment

2.1 Setting up the Hardware

Connect the "wall wart" power supply to the EVB, The green power LED on the board will illuminate. Now connect the EVB to the PC with the serial cable. Start the *MiniIDE* software by double-clicking on its icon. To communicate with the EVB, select the **Terminal** pull-down menu and select both **Show Terminal Window** and **Connected**. This brings up a terminal sub-window through which you can communicate with the EVB.

Press the reset button on the EVB. This starts the D-Bug12 software. D-Bug12 prints the following sign-on message in the terminal window:

```
AXIOM MANUFACTURING CML-12C32 v1.3

Press 'E' for external memory
Press any other key for single chip mode
```

Place your cursor in the terminal window and hit the 'E' key on the keyboard. The D-Bug12 software will then confirm that you are running in Expanded Mode (meaning you use the external memory installed in the board) and provide you with a ">" prompt. The software is now ready to receive commands.

More detailed information on setting up the MiniIDE and EVB can be found in Sections 3.5.1 and 3.5.2 of the text-book. If you are having trouble getting started, please consult your TA.

2.2 Examining and Modifying Memory

The md or "memory display" command is used to examine the contents of memory as shown in the following example. The bold-faced text indicates text that is typed by the user. Non-bold text is the response from the EVB.

>md 0800 0820

```
0800 D3 AB 1E 64 69 32 E8 D7 46 8F DA 9D BE 87 3B 18    di2  F     ;
0810 CF BD 06 EC E8 DC 4E B7 21 6A F8 09 45 23 EC 1D       N !j  E#
0820 F6 C4 FF F4 A1 E2 ED 02 85 A6 EF 7D 60 15 8B 14             `
```

This command displays memory from locations $0800 to $0820. *(Note that the data contained in your memory may be different from that displayed above)* The command displays memory contents a line at a time, where each line contains 16 bytes. Note that it always 'rounds' to display a whole number of lines. In addition to printing out the value of each byte in hex format, it also prints out the ascii character equivalent of each byte on the right-hand side of the display. In the example above, memory location $0803 contains the value $64 which is the ascii code for the letter 'd'. Memory location $0804 contains the value $69 which is the ascii code for the letter

'i'. This is useful when looking for a text string stored in memory. Note that many values do not have a "printable" ascii equivalent. If the "**md**" command specifies only a starting address, the monitor will display 9 lines of data starting at that address.

The mm or "memory modify" command can be used to alter the values stored in memory locations. The user specifies a single memory address. The monitor prints out the address followed by the value currently stored at that address. It then waits for a response from the user. There are three possible responses:

(a) Hit the return (ENTER) key. This will leave the memory location at its current value. The monitor will move on to the next memory address and, once again, wait for a user response.
(b) Type a new hex value to be stored into the current location followed by a return (ENTER). This will change the value currently stored in this location to the new value that was entered. The monitor will then move on to the next memory address and, once again, wait for a user response.
(c) Type a single period '.' This will exit the memory modify mode and return the user to the command prompt.

Use of these commands can be seen in the following example.

>**mm 0800**
0800 D3 **24**
0801 AB
0802 1E **35**
0803 64 **.**
>

In this example, memory location $800 was changed to $24 and location $802 was changed to $35. Locations $801 and $803 were left unchanged. You can verify the changes you have made by using the "**md**" command.

The bf or "block fill" command is used to fill a number of contiguous memory locations with the same value. For example, it may be desirable to zero a section of the on-chip RAM memory. The following command will set all locations from $6000 to $602F in the user on-chip RAM memory to zero.

>**bf 6000 602F 00**

Try this command and then use the "**md**" command to display that section of memory

The mov or "move memory" command copies entire sections of memory from one location to another. The user needs to specify three memory addresses: the starting address of the source, the ending address of the source and the starting address of the destination. For example,

>**mov 0800 08FF 6000**

will copy the data in the 256 locations from address $0800~$08FF to locations $6000~$60FF. Try this command and then use the "**md**" command to check the result.

2.3 Examining and Modifying Registers

The MC9S12 CPU has seven registers: the program counter (PC), the stack pointer (SP), two index registers (X and Y), two 8-bit accumulators and the condition code register (CCR).

The rd or "register display" command displays the current values of all registers.

```
>rd
P-0123   Y-0000   X-0000   A-00   B-00   C-D0   S-3E5F   PG-00
```

Note that it also displays the value of the *page* register (PG). This register is used to extend the memory address range of the processor beyond 64k bytes. We will not be using the PG register in these lab sessions.

The rm or "register modify" command allows the user to modify the value of one register. The command takes a single argument – the name of the register (P, Y, X, A, B, C or S). The PG register cannot be modified. In response to the **"rm"** command, the monitor prints out the name of the register and it current value. The user can either enter a new value or leave the current value unchanged by simply typing return (ENTER).

```
>rm B
B-00  5A
>
```

This example will change the contents of accumulator B from $00 to $5A.

2.4 Laboratory Reports

All exercises must be conducted during the assigned laboratory period. A description of each experiment, your code solution and results should be submitted with your lab. report.

3. Entering and Running a Simple Program

The following assembly language code is a simple program to get you started. The code loads the number 5 into accumulator A of the MC9S12, copies that value from accumulator A to accumulator B and then stores the contents of accumulator B into memory location $4000.

```
ORG   $7000      ; starting address of the program in memory
ldaa  #$05       ; loads accumulator A with value 5
tfr   A, B       ; transfers contents of A to B
stab  $4000      ; stores contents of B into memory location $4000
swi              ; returns control to the monitor program
```

For all your exercises, you will be using the Minim IDE to create and assemble the program on the PC. You can then download binary machine instructions to the EVB where you can execute your program. The use of the MiniIDE and the downloading of programs is discussed in the textbook in sections 3.5.4 through 3.5.6. A brief description of the process is as follows:

1. Generate your assembly language program using the MiniIDE editor. Store your code in a file with an ".asm" extension. Let's assume you call it *lab_code.asm*
2. Assemble the program using the **Build** pull-down menu and clicking on **Build lab_code.asm.** An output window will appear and inform you of any syntax errors found in your code. Correct any errors and re-assemble. Once the program is error free, the assembler will generate an S-record file (with a *.s19* extension - in this case *lab_code.s19*). This file contains the raw binary machine code instructions that will be executed by the microprocessor.
3. Download the program into the EVB. This is done by returning to the D-Bug12 window and typing **load** followed by a return (ENTER). There is no immediate response from the monitor. Now hit the *F8* key on your keyboard. A browsing window will pop up. Select the name of your binary file (lab_code.s19) and click **open**. The D-Bug12 software will load the binary instructions into the microprocessor memory. It will print out a few periods to let you know it is downloading and then return with the standard prompt.
4. Once your program has loaded, you can execute your code with the **go** command followed by the (hex) starting address of your program. This **go** command simply jumps to the starting address you provide. The "swi" instruction at the end of your program is a software interrupt instruction which returns control to the D-Bug12 monitor.

3.1 Exercise

Enter the simple program described above into the MiniIDE text editor and assemble. Once your program is error free, you can view the results of your assembly by loading the file *lab_code.lst* into the MiniIDE text editor. This will show your original code together with the actual binary code that is to be loaded into the microprocessor. Take note of the first op-code in your program (the value that will be loaded into location $7000).

Load the S-record file into the microprocessor. Check the value of memory location $7000 to make sure it has the correct op-code.

You can now start execution by typing

>**go 7000**

Your program will now execute until it reaches the *swi* instruction, at which point it will return to the monitor and issue a new prompt. This happens almost instantaneously as it only takes your program a few microseconds to execute.

The contents of the registers will be displayed when the program finishes executing. Both accumulators (A and B) should contain $05. Examine memory location $4000 to make sure that it also contains $05.

4. Laboratory Assignment

4.1 Simple Arithmetic

The following program subtracts the contents of memory location $4004 from the sum of memory locations $4000 and $4002 and stores the result in memory location $4010.

```
        ORG   $6000
        ldaa  $4000
        adda  $4002
        suba  $4004
        staa  $4010
        swi
```

(a) Using the D-Bug12 monitor initialize the values of locations $4000, $4002 and $4004 to be $23, $3F and $18 respectively.
(b) Enter the program into the MiniIDE text editor. Assemble, load and run the program. What is the answer you expect to be stored in location $4010 ?
(c) Display the data stored in locations $4000, $4002, $4004 and $4010.
(d) What has changed and what has not? Explain.
(e) What data is stored in the registers? Explain
(f) Using the monitor, change the data in locations $4000, $4002 and $4004 to new values (your choice) and re-run the program. Is it necessary to re-assemble? To re-load? Why?
(g) Check the new results.

4.2 Multiplication

The following program multiplies the contents of memory locations $4000 and $4001 and stores the 16-bit result in location $4002~$4003. Note that two labels (N1 and N2) are used to identify the memory locations $4000 and $4001 respectively. ANS labels the two-byte location $4002~$4003. The *mul* instruction multiplies the 8-bit contents of accumulator A by the 8-bit contents of accumulator B and places the 16-bit result in accumulator D, overwriting the operands in A and B. This means that the MSByte of the result will be in A and the LSByte of the result will be in B.

```
        ORG   $4000
N1:     DC.B  7             ; note that this is decimal 7
N2:     DC.B  12            ; note that this is decimal 12
ANS:    DS.W  1             ; reserve 2 bytes to hold result

        ORG   $7000
        ldaa  N1
        ldab  N2
        mul                 ; D = A x B
        std   ANS
        swi
```

(a) Enter the program into the MiniIDE text editor. Assemble, and load the program.
(b) Check the initial values in locations $4000 and $4001
(c) Run the program
(d) Check the result. Is it what you expected?
(e) Using the monitor change locations $4000 and $4001 to new values (your choice) each greater than 16 (decimal)
(f) Re-run the program and check the result. Is it what you expect?

4.3 Multiply-Add

Modify the multiply program (4.2) as follows:

(a) Use a DC.W assembler directive to add a third 16-bit (2-byte) operand N3 initialized to $1234 (hexadecimal).

(b) Change the program so that after performing the multiply, you add the 16-bit operand N3 to the product before storing it in ANS. In other words, you will be performing ANS = (N1*N2) + N3. Note that this will require a 16-bit addition.

Assemble, load and run your modified program. Check the results. Are they what you expect?