University of Windsor

ELECTRICAL AND COMPUTER ENGINEERING
ELEC3270: MICROPROCESSORS

ASSIGNMENT 6: PRINTING TEXT AND STACK INSPECTION & MANIPULATION

OBJECTIVE: To print text, analyze the stack, and to manipulate the stack to print text easily.

SECTIONS:

A. Printing Text to the LCD Efficiently (20%)
B. Passing Variable/Data to a Subroutine through the Stack (40%)
C. Printing Text from the Return Address and Changing the Return Address (40%)

A. PRINTING TEXT TO THE LCD EFFICIENTLY

The 64HC11 can send and receive information using both serial and parallel methods. When using a microcontroller development board, a PC is typically connected to it using an RS232 connection, a far earlier predecessor of USB. Most PCs today don't have RS232 connections anymore, but we can still use connect to devices supporting this by using a USB-to-serial adapter. Since we don't have the ability to use the development board in the physical labs, we will delay the discussion on how this all works until a later chapter. In the mean time we will use a different method to transfer information from the MCU to us by means of a simple text based interface which is something very similar to what you used in Assignment 3. In THRSim11 we have access to a virtual Liquid Crystal Display (LCD). You can see this by going to the "View", "I/O Box" menu. A new window will appear showing the state of ports B, C, and D as well as a small 20x4 LCD. We will send ASCII information to the display by writing to address $1040 in the MCU memory. To clear the LCD we simply write a 0 to it, and to move to a new line, we write a 13 (which is ASCII for carriage-return, a legacy term which told line printers to move the printing head back to the beginning of the line), anything else we write will be interpreted as ASCII text and shown on the LCD. When the text on the line goes beyond 20 columns, it will wrap back to the next line. When the text is on the last line, it will wrap back up to the top of the line (sorry, it doesn't support scrolling). A subroutine has been supplied which you can call (using a JSR) with the value you wish to send to the LCD in ACCA; it is called "SendChar". Although we don't need a subroutine, we could just write ACCA to $1040, using a subroutine will allow us to migrate to the RS232 interface later by changing the contents of the subroutine instead of our code which calls it.

We can load ACCA with a value and call "SendChar", then load ACCA with another value and call "SendChar", and so on, but this results in large code whose size depends on how much information we want to transfer. For this section of the assignment you will code subroutine which will be passed the location in memory of where **your name** is stored in ASCII via the INDX register. It will read this location using index X addressing to load in each letter and, if it isn't zero, then call the "SendChar" subroutine to display it. If it is zero, it will leave the subroutine. Please use your "official name" which can be found on the UWinsite-Student under "Profile", "Personal Details"; it should be listed as first, then last name. You can code your name using the "FCC" (Form table of Constant Characters) as well as "FCB" (Form table of Constant Bytes) for non-ASCII characters (such as carriage- return and a zero as it will specify the end of the text). Once your name is displayed, the code will go into an endless loop. The base code, "**SID_6a.asm**", is provided as well as some hints. Use THRSim11 to edit and simulate.

B. PASSING VARIABLE/DATA TO A SUBROUTINE THROUGH THE STACK

In the previous section you were asked to write code which would print out your name. You did this by passing the memory address of what to print via the MPU INDX register. However it is quite possible that we need to supply more information than registers we have available. An alternative, and more scalable approach, is to use the stack to pass the information. We will first review the assembled code at the top of the next page to understand how the stack operates. The first line loads INDX with $10, and then the TXS instruction subtracts 1

```
D000 CE 00 10 ldx  #$0010
D003 35       txs
D004 86 55    ldaa #$55
D006 C6 AA    ldab #$AA
D008 36       psha
D009 37       pshb
D00A CE 12 34 ldx  #$1234
D00D 3C       pshx
D00E BD D0 13 jsr  $D013
D011 20 F1    bra  $D004
D013 39       rts
```

from it and moves it into the stack pointer (SP); the SP is now $F which is the top of the stack. The stack grows backwards so when we "push" the first value onto it, it will be stored at $F in memory, and then SP will be decremented to $E. So until the PC gets to $D013, $55 (will be stored at $F), $AA (at $E), $34 (at $D), $12 (at $C), $11 (at $B), and $D0 (at $A) will be pushed on the stack. We can see why $55 is before $AA as we can see the individual instructions. However, when we push a 16-bit value on the stack, the LSB gets pushed on first. The same applies to subroutines. When a JSR (jump to subroutine) or BSR (branch to subroutine) instruction is encountered, the MPU pushes the NEXT PC on the stack and then changes the PC based on the operand of the instruction. So we can see that $D011 is pushed on the stack and then the PC is updated to $D013. When the MPU eventually runs the RTS (return from subroutine) instruction, it will pull off the last 16-bit value and store it into the PC, so the next instruction will be at $D011 and the SP will be at $C. We have to be careful with the SP to ensure it is in RAM. In this example, the SP is initially set very low to illustrate a potential problem. We can see that $D011 jumps back to $D004 which repeats pushing the same information on the stack. At some point the SP will go effectively negative (reset back to $FFFF or lower) due to the unsaturated subtraction, and since the memory from $D000-$FFFF is read only, the stack operations will not successfully write to this memory. When the MPU eventually encounters a PULL or an RTS, it will be reading the values from ROM instead of what was intended to be stored, and we can expect the system to fail from this point on. The 68HC11 doesn't have any protection mechanisms against this type of stack overflow, but it can trap illegal instructions or reset the system if the program doesn't reset a special timer in a given amount of time. We will discuss this in later lectures.

For this section of the assignment, you will rewrite your code from section A to use the last 16-bit entry on the stack (after the return PC in RAM, or before the return PC in the stack) to specify the location of where to read the text from. In the above example, the location in memory you could use to start reading and sending would be at $1234 assuming $D013 was our printing function. To summarize, the code on the right shows how we would pass a single value (a 16-bit address in this case) to a subroutine, but you can easily pass more pieces of information. Once the subroutine returns, it is important to pull off those values so that the stack doesn't overflow as more subroutines are called later on. Some MPUs have a series of instructions that can easily add to the SP. For example, if we had to pass 20 16-bit variables to a subroutine, we could push those 20 values, call the subroutine, and then pull them all off the stack. Or we could simply add 20 x 2 to the SP once the subroutine returns; far faster than pulling them all off the stack. This is what is done with languages such as "C" when they are compiled to assembly code. Download the base code "**SID_6b.asm**", and modify it appropriately. Remember that the SP points to the next available entry in the stack. The instruction TSX (transfer SP to INDX) actually moves SP+1 to INDX so that INDX will be pointing to the current top most entry in the stack, not the next empty entry. After this instruction, you can use any INDX addressing to indirectly access entries in the stack.

```
LDX   #message
PSHX
JSR   PrintText
PULX
```

C. Printing Text from the Return Address and Changing the Return Address

To maximize the challenge of this assignment, we can further extend the work we've done in stack interpretation to include some manipulation as well. For this section you will modify your printing subroutine "PrintText" so that it prints the text at the memory location immediately following the BSR or JSR to your subroutine (the first 16-bit value on the stack once you enter your subroutine, the return PC); see code example on right. It will print the text at this location until a zero is found. It will then modify the return PC so that the MPU returns to the point after the zero terminator. In practice this can save a lot of code space, but the parameters to the subroutine are always constant if the code is in ROM so it can limit its usefulness. Examine the base code "**SID_6c.asm**" as it is written to simply show you the location of where you should start reading from. A guideline is provided to offer some hints.

```
JSR   PrintText
FCC   'HELLO'
FCB   13,0
(next opcode)
```

ZIP all your ASM files into "SID_6.zip" and upload it to Blackboard.