ME 333 Homework 5

Marshall Johnson

February 11, 2022

Chapter 5 Exercises

3) Answer the following questions.

a. Which combinations of data types and arithmetic functions result in a jump to a subroutine? From your disassembly file, copy the C statement and the assembly commands it expands to (including the jump) for one example.

Data types and arithmetic functions that result in a jump to a subroutine:

- long long int
  - division
- float
  - addition
  - subtraction
  - multiplication
  - division
- long double
  - addition
  - subtraction
  - multiplication
  - division

Example (long double/addition):

```
    d3 = d1+d2;
9d003098:  8fc60040  lw  a2,64(s8)
9d00309c:  8fc70044  lw  a3,68(s8)
9d0030a0:  8fc40038  lw  a0,56(s8)
9d0030a4:  8fc5003c  lw  a1,60(s8)
9d0030a8:  0f4009c0  jal  9d002700 <__adddf3>
9d0030ac:  00000000  nop
9d0030b0:  afc20060  sw  v0,96(s8)
9d0030b4:  afc30064  sw  v1,100(s8)
```

b. **For those statements that do not result in a jump to a subroutine, which combination(s) of data types and arithmetic functions result in the fewest assembly commands? From your disassembly, copy the C statement and its assembly commands for one of these examples. Is the smallest data type, char, involved in it? If not, what is the purpose of extra assembly command(s) for the char data type vs. the int data type?**

c. int

   - addition
   - subtraction
   - multiplication

   Example(int/addition):

```
        i3 = i1+i2;
9d002f3c:  8fc30014   lw  v1,20(s8)
9d002f40:  8fc20018   lw  v0,24(s8)
9d002f44:  00621021   addu  v0,v1,v0
9d002f48:  afc2004c   sw  v0,76(s8)
```

   The extra assembly command (andi) for the char data type is used to turn the 8 bit char into the correct number of bits based on the CPU by performing an AND operation to add empty bits (in addition to the 8 bits that are desired).

d. **Fill in the following table. Each cell should have two numbers: the number of assembly commands for the specified operation and data type, and the ratio of this number (greater than or equal to 1.0) to the smallest number of assembly commands in the table.**

	char	int	long long	float	long double
+	1.25 (5)	1.0 (4)	2.75 (11)	1.25 (5) J	2.0 (8) J
-	1.25 (5)	1.0 (4)	2.75 (11)	1.25 (5) J	2.0 (8) J
*	1.25 (5)	1.0 (4)	4.5 (18)	1.25 (5) J	2.0 (8) J
/	1.75 (7)	1.75 (7)	2.0 (8) J	1.25 (5) J	2.0 (8) J

e. **From the disassembly, find out the name of any math subroutine that has been added to your assembly code. Now create a map file of the program. Where are the math subroutines installed in virtual memory? Approximately how much program memory is used by each of the subroutines? You can use evidence from the disassembly file and/or the map file.**

```
Microchip PIC32 Memory-Usage Report

kseg0 Program-Memory Usage
section                address   length [bytes]      (dec)
-------             ----------   ------------------------
.text.dp32mul       0x9d001e00           0x4b8       1208
.text               0x9d0022b8           0x440       1088
.text.dp32subadd    0x9d0026f8           0x430       1072
.text.dp32mul       0x9d002b28           0x32c        812
.text               0x9d002e54           0x5a8       1448
.text.fpsubadd      0x9d0033fc           0x278        632
.text.fp32div       0x9d003674           0x230        560
.text.fp32mul       0x9d0038a4           0x1bc        444
```

4) **How many commands does each use?**

a. u3 = u1 & u2: 4 commands

b. u3 = u1 | u2: 4 commands

c. u3 = u1 << u2: 3 commands

d. u3 = u1 >> u2: 3 commands

Chapter 6 Exercises

1) **Interrupts can be used to implement a fixed frequency control loop (e.g., 1 kHz). Another method for executing code at a fixed frequency is polling: you can keep checking the core timer, and when some number of ticks has passed, execute the control routine. Polling can also be used to check for changes on input pins and other events. Give pros and cons (if any) of using interrupts vs. polling.**

Interrupts allow for instant notifcation when data is ready using a hardware mechanism. This prevents unecessary checking that occurs with polling. On the other hand, with polling, data timing is predictable, whereas an interrupt can occur at any time.

4) a. **What happens if an IRQ is generated for an ISR at priority level 4, subpriority level 2 while the CPU is in normal execution (not executing an ISR)?**

   The ISR is executed immediately.

   b. **What happens if that IRQ is generated while the CPU is executing a priority level 2, subpriority level 3 ISR?**

   The CPU jumps to the higher priority ISR (from the previous question) and executes before returning to finish prcoessing the lower priority ISR.

   c. **What happens if that IRQ is generated while the CPU is executing a priority level 4, subpriority level 0 ISR?**

   Since they are the same priority level, the current ISR is finished before jumping to the priority 4, subpriority 2 interrupt from the first question.

   d. **What happens if that IRQ is generated while the CPU is executing a priority level 6, subpriority level 0 ISR?**

   The CPU finishes processing the priority level 6 ISR before jumping to and executing the ISR from the first question (priority 4.2).

5) **An interrupt asks the CPU to stop what it's doing, attend to something else, and then return to what it was doing. When the CPU is asked to stop what it's doing, it needs to remember "context" of what it was working on, i.e., the values currently stored in the CPU registers.**

   a. **Assuming no shadow register set, what is the first thing the CPU must do before executing the ISR and the last thing it must do upon completing the ISR?**

   Before executing the ISR, the CPU must save the contents of the CPU registers to RAM. After completing the ISR, must copy that data (the context) back from RAM to its registers. This restores the CPU state prior to executing the ISR, allowing it to continue from where it left off.

   b. **How does using the shadow register set change the situation?**

   The shadow register set (SRS) is an extra full set of registers. Rather than saving and restoring, the CPU can switch to this extra register set when processing the ISR. After finishing the ISR, it can switch back to the original register set without the need to restore, as it will be in the same state it was before switching to the SRS.

8) **For the problems below, use only the SFRs IECx, IFSx, IPCy, and INTCON, and their CLR, SET, and INV registers (do not use other registers, nor the bit fields as in IFS0bits.INT0IF). Give valid C bit manipulation commands to perform the operations without changing any uninvolved bits. Also indicate, in English, what you are trying to do, in case you have the right idea but wrong C statements. Do not use any constants defined in Microchip XC32 files; just use numbers.**

   a. **Enable the Timer2 interrupt, set its flag status to 0, and set its vector's priority and subpriority to 5 and 2, respectively.**

```
// Set IEC0 bit 8 to 1 (enable interrupt)
IEC0SET = 0x100;

// Clear IFS0 bit 8 to 0 (set flag status)
IFS0CLR = 0x100;

// Set IPC2 bits 2 and 4 to 1 (vector
    priority 5)
IPC2SET = 0x14;

// Set IPC2 bits 0–1 to 1 (subpriority 2)
IPC2SET = 0x3;
```

   b. **Enable the Real-Time Clock and Calendar interrupt, set its flag status to 0, and set its vector's priority and subpriority to 6 and 1, respectively.**

```
// Set IEC1 bit 15 to 1 (enable interrupt)
IEC1SET = 0x8000;

// Clear IFS1 bit 15 to 0 (set flag status)
IFS1CLR = 0x8000;

// Set IPC8 bits 27 and 28 to 1 (vector
    priority 6)
IPC8SET = 0x18;

// Set IPC8 bit 24 to 1 (subpriority 1)
IPC8SET = 0x1000000;
```

c. **Enable the UART4 receiver interrupt, set its flag status to 0, and set its vector's priority and subpriority to 7 and 3, respectively.**

```
// Set IEC2 bit 4 to 1 (enable interrupt)
IEC2SET = 0x10;

// Clear IFS2 bit 4 to 0 (set flag status)
IFS2CLR = 0x10;

// Set IPC12 bits 10-12 to 1 (vector
    priority 7)
IPC12SET = 0x1c00;

// Set IPC8 bits 8 and 9 to 1 (subpriority
    3)
IPC12SET = 0x300;
```

d. **Enable the INT2 external input interrupt, set its flag status to 0, set its vector's priority and subpriority to 3 and 2, and configure it to trigger on a rising edge.**

```
// Set IEC0 bit 11 to 1 (enable interrupt)
IEC0SET = 0x800;

// Clear IFS0 bit 11 to 0 (set flag status)
IFS0CLR = 0x800;

// Set IPC2 bits 26 and 27 to 1 (vector
    priority 3)
IPC12SET = 0xc000000;

// Set IPC8 bit 25 to 1 (subpriority 2)
IPC12SET = 0x2000000;
```

9) **Edit Code Sample 6.3 so that each line correctly uses the "bits" forms of the SFRs. In other words, the left-hand sides of the statements should use a form similar to that used in step 5, except using INTCONbits, IPC0bits, and IEC0bits.**

The following lines of code were extracted from Code Sample 6.3 and modified to use the "bits" forms of the SFRs. For the complete version of modified code, please see sample_6-3.c.

```
// clear the interrupt flag
IFS0bits.INT0IF = 0;
// clear the interrupt flag
IFS0bits.T2IF = 0;
// step 3: INT0 and INT1 trigger on rising edge
INTCONbits.INT0EP = 1;
INTCONbits.INT1EP = 1;
// step 4: clear 5 priority and subp bits for INT0
IPC0bits.INT0IP = 0;
IPC0bits.INT0IS = 0;
// step 4: set INT0 to priority 6 subpriority 0
IPC0bits.INT0IP = 6;
IPC0bits.INT0IS = 0;
// step 4: clear 5 priority and subp bits for INT1
IPC1bits.INT1IP = 0;
IPC1bits.INT1IS = 0;
// step 4: set INT1 to priority 6 subpriority 0
IPC1bits.INT1IP = 6;
IPC1bits.INT1IS = 0;
// step 6: enable INT0 and INT1 interrupts
IEC0bits.INT0IE = 1;
IEC0bits.INT1IE = 1;
```

16) **Modify Code Sample 6.2 so the USER button is debounced. How can you change the ISR so the LEDs do not flash if the falling edge comes at the beginning of a very brief, spurious down pulse? Verify that your solution works.**

Delay for 10 ms (duration of real button press), then check value of USER button. If pressed (false), execute code to turn LEDs on and off. Otherwise, just clear the interrupt flag.

See debounce.c for code.

17) **Using your solution for debouncing the USER button (Exercise 16), write a stopwatch program using an ISR based on INT2.**

See timer.c.