HW 5
Lance Go

5.3)

(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.

All of the operations that involve floats or long doubles require "jal". In the long long int operations, only divide needs a "jal" instruction. All other operations do not require "jal."

(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? (Hint: the assembly command ANDI takes the bitwise AND of the second argument with the third argument, a constant, and stores the result in the first argument. Or you may wish to look up a MIPS32 assembly instruction reference.)

For the two data types that are not involved in a jump, char and int, int had the fewer instructions.

```
// char add: c3 = c1+c2;
9d002ee4:       93c30010    lbu     v1,16(s8)
9d002ee8:       93c20011    lbu     v0,17(s8)
9d002eec:       00621021    addu    v0,v1,v0
9d002ef0:304200ff   andi    v0,v0,0xff
9d002ef4:a3c20048   sb      v0,72(s8)
// int add: 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 instruction in the char vs the int is and andi instruction. This exists to essentially truncate any overflow that could occur from adding two chars. This is not necessary in ints since the size of the general registers is the same size as an int.

(c) 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. For example, if addition of two ints takes four assembly commands, and

this is the fewest in the table, then the entry in that cell would be 1.0 (4). This has been filled in below, but you should change it if you get a different result. If a statement results in a jump to a subroutine, write J in that cell.

	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)	2.75(19)	1.25(5)J	2.0(8)J
/	1.75(7)	1.75(7)	2(8)J	1.25(5)J	3.75(15)J

(d) 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. (Hint: You can search backward from the end of your map file for the name of any math subroutines.)

Math Subroutine	Address	Length (bytes)
Dp32mul	0x9d001e00	1208
Dp32subadd	0x9d0026f8	1072
Dp32mul (different version in a different mem location?)	0x9d002b28	812
fpsubadd	0x9d003444	632
Fp32div	0x9d0036bc	560
Fp32mul	0x9d0038ec	444

5.4) How many commands does each use? For unsigned integers, bit-shifting left and right make for computationally efficient multiplies and divides, respectively, by powers of 2.

Operation	Number of Instructions
&	4
|	4
<<	3
>>	10

6.1) Give pros and cons (if any) of using interrupts vs. polling.

The main problem with polling is that it effectively puts the microcontroller on pause while it is waiting for in an input as it polls. If efficiency is an important to the application of the microcontroller, polling is not applicable. With an interrupt, other actions can be executed by the microcontroller as long as the interrupt is not going. One con of interrupts is that having a large amount of interrupts means there can be lots of interrupts queued and you only have control over the priority of the interrupts. Polling can be much simpler if you are only waiting on one input.

6.4) (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 CPU will execute the ISR.

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

The CPU will finish executing the current level 4 ISR and then move on to the level 2 ISR.

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

The CPU will finish executing the current level 4 ISR and then move on to the other level 4 ISR.

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

The CPU will execute the level 6 ISR and then move back to executing the level 4 ISR.

6.5) (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?

The CPU will use “context save and restore,” which is where the current state of all of the internal registers in the CPU will first be saved to RAM. Then the Interrupt will execute and will be able to change any of the internal registers. After the CPU is done executing the ISR, it will restore the registers by loading the saved states from RAM back into the registers.

(b) How does using the shadow register set change the situation?

The SRS is another set of registers in the CPU that are used for interrupts. When an interrupt triggers, the current set of registers does not need to be changed. Instead, the CPU switches over the SRS. When the interrupt is done, the CPU can switch back to its normal set of registers without having to do any save/load from RAM.

6.8) (a) Enable the Timer2 interrupt, set its flag status to 0, and set its vector’s priority and subpriority to 5 and 2, respectively.

```
IPC2SET = 0b10110;
IFS0SET = 0 << 8;
IEC0SET = 1 << 8;
```

(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.

IPC8SET = 0b11001 << 24;
IFS1SET = 0 << 15;
IEC1SET = 1 << 15;

(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.

IPC12SET = 0b11111 << 8;
IFS2SET = 0 << 5;
IEC2SET = 1 << 5;

(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.

IPC2SET = 0b01110 << 24;
IFS0SET = 0 << 11;
IEC0SET = 1 << 11;

6.9) Edit Code Sample 6.3 so that each line correctly uses the "bits" forms of the SFRs.

```
  INTCONbits.INT0EP = 1;            // step 3: INT0 and INT1 trigger on rising edge
  INTCONbits.INT1EP = 1;
  IPC0bits.INT0IP = 6;              // step 4: set INT0/INT1 to priority 6 subpriority
0
  IPC0bits.INT0IS = 0;
  IPC1bits.INT1IP = 6;
  IPC1bits.INT1IS = 0;

  IFS0bits.INT0IF = 0;              // step 5: clear INT0 flag status
  IFS0bits.INT1IF = 0;              // step 5: clear INT1 flag status

  IEC0bits.INT0IE = 1;
  IEC0bits.INT1IE = 1;
```

6.16) See included code. Named "INT_ext_int.c"

6.17) See included code. Named "main.c"