Chapter 5 (use no optimization for all exercises): Exercises 3, 4.

Chapter 6: Exercises 1, 4, 5, 8 (8.c Our PIC32 does not have UART4, either use UART1 or UART2 or refer to the 795 UART4), 9, 16,17. Make a demo video for 6.17

3

To write time-efficient code, it is important to understand that some mathematical operations are faster than others. We will look at the disassembly of code that performs simple arithmetic operations on different data types. Create a program with the following local variables in main:

```
char c1=5, c2=6, c3;
int i1=5, i2=6, i3;
long long int j1=5, j2=6, j3;
float f1=1.01, f2=2.02, f3;
long double d1=1.01, d2=2.02, d3;
```

Now write code that performs add, subtract, multiply, and divide for each of the five data types, i.e., for chars:

```
c3 = c1+c2;
c3 = c1-c2;
c3 = c1*c2;
c3 = c1/c2;
```

Build the program with no optimization and look at the disassembly. For each of the statements, you will notice that some of the assembly code involves simply loading the variables from RAM into CPU registers and storing the result (also in a register) back to RAM. Also, while some of the statements are completed by a few assembly commands in sequence, others result in a jump to a software subroutine to complete the calculation. (These subroutines are provided with our C installation and included in the linking process.) 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.

- long long int division
- float add
- float subtract
- float multiply
- float divide
- long double add
- long double subtract

- long double multiply
- long double divide

Example from long long division:

```
j3 = j1/j2;
9d00867c:   8fc60028    lw  a2,40(s8)
9d008680:   8fc7002c    lw  a3,44(s8)
9d008684:   8fc40020    lw  a0,32(s8)
9d008688:   8fc50024    lw  a1,36(s8)
9d00868c:   0f401e52    jal 9d007948 <__divdi3>
9d008690:   00000000    nop
9d008694:   afc20030    sw  v0,48(s8)
9d008698:   afc30034    sw  v1,52(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? (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.)

add subtract multiply with int have the smallest amount of instruction.

```
i3 = i1+i2;
9d008570:   8fc30014    lw  v1,20(s8)
9d008574:   8fc20018    lw  v0,24(s8)
9d008578:   00621021    addu    v0,v1,v0
9d00857c:   afc2001c    sw  v0,28(s8)
```

Char is not the shortest despite being the smallest

Assembly code from char for comparison.

```
c3 = c1+c2;
9d008508:   93c30010    lbu v1,16(s8)
9d00850c:   93c20011    lbu v0,17(s8)
9d008510:   00621021    addu    v0,v1,v0
9d008514:   304200ff    andi    v0,v0,0xff
9d008518:   a3c20012    sb  v0,18(s8)
```

Compare to the int operation, other then all the instruction changed from `word` to `byte`, there is an extra instruction of `andi`. Which is adding a constant to register `v0` with overflow trap. Which is the current register for `c3`. This is

basically clearing some of the bits to help store back into memory (since char is smaller then native memory type. Writing a whole registor affects things)

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.

Actual number of instruction

Instruction number Ratio (actual number of instruction)

	char	int	long long	float	long double
+	1.25(5)	1.0(4)	2.75(11)	j	j
-	1.25(5)	1.0(4)	2.75(11)	j	j
	1.25(5)	1.0(4)	4.5(18)	j	j
/	1.75(7)	1.75(7)	j	j	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.)

All reference of sub routine

```
__divdi3
__addsf3
__subsf3
__mulsf3
__divsf3
__adddf3
__subdf3
__muldf3
__divdf3
```

name	location	size	note
___divdi3	0x000000009d007948	0x444 , 1092 bytes	
___addsf3	0x000000009d0087d0	0x277 , 631 bytes	// ___addsf3 is 8 byte later then ___subsf3
___subsf3	0x000000009d0087c8	0x278 , 632 bytes	
___mulsf3	0x000000009d008c70	0x1bc , 444 bytes	
___divsf3	0x000000009d008a40	0x230 , 560 bytes	
___subdf3	0x000000009d007d8c	0x430 , 1072 bytes	
___adddf3	0x000000009d007d94	0x428 , 1064 bytes	// ___adddf3 is 8 byte later then
___muldf3	0x000000009d0081bc	0x32c , 812	
___divsf3	0x000000009d008a40	0x230 , 560	

Total size is 6867 bytes

4

Let us look at the assembly code for bit manipulation. Create a program with the following local variables:

```
unsigned int u1=33, u2=17, u3;
```

and look at the assembly commands for the following statements:

```
u3 = u1 & u2;
u3 = u1 | u2;
u3 = u2 << 4;
u3 = u1 >> 3;
// bitwise AND
// bitwise OR
// shift left 4 spaces, or multiply by 2^4 = 16
// shift right 3 spaces, or divide by 2^3 = 8
```

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	instruction number
u3 = u1 & u2;	4
u3 = u1 | u2;	4
u3 = u2 « 4;	3
u3 = u1 » 3;	4

Chapter 6

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.

- interrupt:

The benefit is low cpu usage. Whenever the control loop code is not running, cpu if free to work on any other stuff.

The drawback is more complicated configuration and context switching for the interrupt. Usually the control loop is not inside the ISP, so need some code to link the (hardware) ISP to trigger a soft-ISP to not block other interrupt.

- polling

The benefit is super simple implementation. Just a big while loop keep checking the cpu time.

Drawback is worse accuracy (depending on what else is in the loop) and hogging the CPU. With busy waiting, either cpu cannot do anything else, or user have to write his own context switching, and is always less accurate on response compare to interrupt.

4

The priority level for each of the 64 vectors is represented by five bits: taking values 0 to 7, or 0b000 to 0b111; an interrupt with priority of 0 is effectively disabled

(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 jump to this ISR and service the interrupt.

(b)

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

Since the new IRQ have higher priority level. The current ISR is paused, and new ISR is processed.

(c)

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

New task have same priority level as current, but new even have higher subpriority. Thus cpu will jump to new ISR

(d)

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

The current task have higher priority. Thus the interrupt is put into queue, and will be executed after current ISR is finished.

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?

Push the current context (registers) to stack, service ISR, Finally pop the stack to restore the context (registers).

(b)

How does using the shadow register set change the situation?

The shadow registers are a full set of registers that can be used instead of normal registers. When ISR happens, the shadow register is used instead of the normal register. This means the stack push and pop steps are skipped, thus improving the performance.

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.

Source	IRQ#	Vector#	flag	enable	priority	sub-priority	persistent
T2 – Timer2	9	8	IFS0<9>	IEC0<9>	IPC2<4:2>	IPC2<1:0>	No

```
IPC2CLR = 0b0001 1111 // In case of dirty bits.
IPC2SET = 5 <<2  // priority =5, which is  IPC2bits.T2IP = 5
IPC2SET = 2 ; // sub priority 2 which is IPC2bits.T2IS = 2
IFS0CLR = 1 << 9 ; // clear interrupt flag IFS0bits.T2IF =0
IEC0SET = 1 << 9 ; // Enable interrupt which is IEC0bits.T2IE =1
```

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.

RTCC – Real-Time Clock and Calendar * 30 * 25 * IFS0<30> * IEC0<30> * IPC6<12:10> * IPC6<9:8> * No

```
IPC6CLR = 0x1F <<8; // In case of dirty bits
IPC6SET = 6 << 10 ; // priority to 6 which is IPC6bits.RTCCIP = 6
IPC6SET = 1 << 8  ; // sub priority to 1 which is IPC6bits.RTCCIS = 1
IFS0CLR = 1<<30   ; // clear interrupt flag which is IFS0bits.RTCCIF = 0
IEC0SET = 1<<30   ; // enable interrupt which is IEC0bits.RTCCIE = 1
```

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. (8.c Our PIC32 does not have UART4, either use UART1 or UART2 or refer to the 795 UART4),

Using UART2

U2RX – UART2 Receiver * 54 * 37 * IFS1<22> * IEC1<22> * IPC9<12:10> * IPC9<9:8> * Yes

```
// No need to clear anything, 7 and 3 are all ones
IPC9SET = 7 << 10 ; // set priority to 7
IPC9SET = 3<<8 ; // set sub priority to 3
IFS1CLR = 1<<22 ; // clear interrupt flag
IEC1SET = 1<<22 ; // enable interrupt
```

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.

INT2 – External Interrupt 2 * 13 * 11 * IFS0<13> * IEC0<13> * IPC2<28:26> * IPC2<25:24> * No

bit 2 INT2EP: External Interrupt 2 Edge Polarity Control bit 1 = Rising edge 0 = Falling edge

```
INTCONSET = 1<<2 ; // set bit 2 for INT2 trigger on rising edge
IPC2CLR = 0x1f <<24 ; // Clear the priority bits
IPC2SET = 3 << 26 ; // set priority to 3
IPC2SET = 2 << 24 ; // set sub priority to 2
IFS0CLR = 1 <<13 ; //clear the flag
IEC0CLR = 1<<13 ; // enable interrupt
```

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.

Using the datasheet `PIC32MX5XX/6XX Family Data Sheet` for the problem. As the original code is intended for 7xx family. Which the bit location will be different from 1xx family.

See `INT_timing_bitsver.c` for all the changes.

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. (Hint: Any real button press should last much more than 10 ms, while the mechanical bouncing period of any decent switch should be much less than 10 ms. See also Chapter B.2.1 for a hardware solution to debouncing.)

see `INT_ext_debounce_int.c` for detail.

For debouncing, we can let ISR remember the timing it's being called each time. If the ISR is called within 10ms of last call, we simply ignore it. This way we can ignore any super close consecutive bounces.

```
// Global variable for recording.
unsigned int last_isr_time = 0;
const unsigned int count_10ms = 2400;

void __ISR(_EXTERNAL_0_VECTOR, IPL2SOFT) Ext0ISR(void) {
  if ((_CP0_GET_COUNT() - last_isr_time)< count_10ms){
  // Basically skip this round of handing if the interrupt happend
  // within 10ms of last recorded case.
  // We still update this. So the switch much to have a true 10ms
```

```
  // quiet period
    last_isr_time = _CP0_GET_COUNT();
    return ;
  }
  // Normal case, also record the timing.
  last_isr_time = _CP0_GET_COUNT();
  // Do rest of the ISR stuff
}
```

17

Make a demo video for 6.17 Using your solution for debouncing the USER button (Exercise 16), write a stopwatch program using an ISR based on INT2.

Connect a wire from the USER button pin to the INT2 pin so you can use the USER button as your timing button. Using the NU32 library, your program should send the following message to the user's screen: Press the USER button to start the timer. When the USER button has been pressed, it should send the following message: Press the USER button again to stop the timer. When the user presses the button again, it should send a message such as 12.505 seconds elapsed. The ISR should either

(1) start the core timer at 0 counts or
(2) read the current timer count, depending on whether the program is in the "waiting to begin timing" state or the "timing state."

Use priority level 6 and the shadow register set. Verify that the timing is accurate. The stopwatch only has to be accurate for periods of less than the core timer's rollover time. You could also try using polling in your main function to write out the current elapsed time (when the program is in the "timing state") to the user's screen every second so the user can see the running time.

See `stopwatch.c` for actual code. The pic we use having problem setting priority 6 on INT2. Thus using priority 2 instead.