CprE 381 – Computer Organization and Assembly Level Programming

HW-05

[Note from Joe: This assignment is meant to bridge the gap as we end the focus on MIPS assembly and begin looking into architectural and performance issues. Note that this is a relatively lengthy assignment – you have two weeks to complete it and should start early.]

Reading: Patterson & Hennessy, Sections 3.1-3.3, A.9, and B.5-B.7

1) Application Benchmarking

(a) The following code fragment processes two arrays and produces an important value in register `$v0`. Assume that each array consists of 2500 words indexed 0 through 2499, that the base addresses of the arrays are stored in `$a0` and `$a1`, respectively, and their sizes (2500) are stored in `$a2` and `$a3`, respectively. Add comments to the code and describe in one sentence what this code does. Specifically, what will be returned in `$v0`?

```
           sll $a2, $a2, 2
           sll $a3, $a3, 2
           add $v0, $zero, $zero
           add $t0, $zero, $zero
Outer:     add $t4, $a0, $t0
           lw $t4, 0($t4)
           add $t1, $zero, $zero
Inner:     add $t3, $a1, $t1
           lw $t3, 0($t3)
           bne $t3, $t4, skip
           addi $v0, $v0, 1
Skip:      addi $t1, $t1, 4
           bne $t1, $a3, inner
           addi $t0, $t0, 4
           bne $t0, $a2, outer
```

(b) Assume that the code from part (a) is run on a machine with a 2 GHz clock that requires the following number of cycles for each instruction:

Instruction	Cycles
`add, addi, sll`	1
`lw, bne`	2

In the worst case, how many seconds will it take to execute this code?

2) MIPS SIMD Programming

(a) Write a MIPS procedure that performs the 1D sum of absolute differences (SAD) of two byte arrays:

```
int sad(char *array1, char *array2, int len) {

   char sum = 0;

   for (int i=0; i<len; i++) {

       sum += sat8(abs(array1[i]-array2[i]));

       // The result of abs(…) becomes max unsigned byte

       // if it were to exceed the capacity of a byte

   }

   return sum;

}
```

The 2D version of this problem is an important and widely-used computational kernel for applications such as video compression (e.g., mp4 motion estimation between frames). You should begin the procedure with a label and end it with `jr $ra`. You can assume the base address of the arrays are in `$a0` and `$a1` and the size of the first (and guaranteed shorter) array is in `$a2`. The result will go in the lowest byte of `$v0`. If you were to overflow a byte, you should simply return the maximum unsigned value you could store in a byte (this is called saturating addition). For example, if the two arrays were {0, 1, 2, 3} and {0, 0, 0, 0}, the result would be 6. Submit as `prob2a_<NetID>.s`.

(b) Write a MIPS program that calls the procedure from part (a) three times with three unique input sets and prints the corresponding output. Submit as `prob2b_<NetID>.s`.

(c) Modify your MIPS code from part (a) to use the quad byte MIPS DSP instructions presented in class. Please verify that your test cases still work. Note that the goal is to reduce the number of dynamically executed instructions, but not necessarily the number of static instructions. Submit as `prob2c_<NetID>.s`. [You can view the quick reference document as well as more complete documentation on the CprE 381 Canvas "Resources" page. We've also included a modified version of MARS that includes the following instructions from MIPS DSP: `absq_s.qb`, `addu.qb`, `addu_s.qb`, `raddu.w.qb`, `repl.qb`, `replv.qb`, and `subu.qb`]

(d) How many instructions (i.e., dynamic instructions) were executed in your programs? Show your calculations. MARS has a tool that can count instructions, which I recommend you use to verify your hand calculations.

3) Computer Arithmetic

(a) P&H(B.24) <§B.5> The ALU in this section supports set on less than (`slt`) using just the sign bit of the adder. Let's try a set on less than operation using the values -7_{ten} and 6_{ten}. To make it simpler to follow the example, let's limit the binary representations to 4 bits: 1001_{two} and 0110_{two}.

$$1001_{two} - 0110_{two} = 1001_{two} + 1010_{two} = 0011_{two}$$

This result would suggest that -7 > 6, which is clearly wrong. Hence, we must factor in overflow in the decision. Modify the 1-bit ALU in Figure B.5.10 on page B-33 to handle `slt` correctly. You can make changes directly to this figure to save time.

(b) P&H(B.11) <§4.2, §B.2, §B.3> Assume that X consists of 3 bits, x2 x1 x0. Write four logic functions that are true if and only if

- X contains only one 0
- X contains an even number of 0s
- X when interpreted as an unsigned binary number is less than 4
- X when interpreted as a signed (two's complement) number is negative

(c) P&H(B.13) <§4.2, §B.2, §B.3> Assume that X consists of 3 bits, x2 x1 x0, and Y consists of 3 bits, y2 y1 y0. Write logic functions that are true if and only if

- X < Y, where X and Y are thought of as unsigned binary numbers
- X < Y, where X and Y are thought of as signed (two's complements) numbers
- X = Y

(d) P&H(B.39) <§B.2, §B.8, §B.10> Construct a 3-bit counter using three D flip-flops and a selection of gates. The inputs should consist of a signal that resets the counter to 0, called *reset*, and a signal to increment the counter, called *inc*. The outputs should be the value of the counter. When the counter has value 7 and is incremented, it should wrap around and become 0.