CprE 381 Homework 5

[Note: This homework gives you some more practice with procedure calls and MIPS programming. As a happy coincidence, it will help provide a class-wide test program that I will share with you. Then the homework begins to cover processor design choices.]

1. Processor Implementation Details P&H(4.2) <§4.1>. The basic single-cycle MIPS implementation in Figure 4.1.2 (COD Figure 4.2) can only implement some instructions. New instructions can be added to an existing Instruction Set Architecture (ISA), but the decision whether or not to do that depends, among other things, on the cost and complexity the proposed addition introduces into the processor datapath and control. The first three problems in this exercise refer to the new instruction:

   Instruction: **`JIC`** rt, imm
   Interpretation: PC = Reg[rt] + sign_extend(imm)

   a. Which existing blocks (if any) can be used for this instruction?
   PC, Instruction Memory, Register File (2 read ports and 1 write port), ALU Source B MUX, ALU.

   b. Which new functional blocks (if any) do we need for this instruction?
   ALU input A mux, JIC mux.

   c. What new signals do we need (if any) from the control unit to support this instruction?
   Two control signals will be needed, one for ALUSrcA, and the other being JIC (for the JIC mux).

2. Processor Cycle Time Determination
   Assume the following latencies for the logic blocks in Figure 4.4.5 (COD Figure 4.17) from the textbook.

I-Mem	Adder	MUX	ALU	Reg Read	D-Mem	Sign-Extend	Shift-Left-2	Control	ALU Control	AND gate
225ps	85ps	15ps	100 ps	110ps	340ps	15ps	10ps	70ps	15ps	10ps

   a. Identify and quantify (i.e., give the path through the blocks and the time for that path) the worst-case path for each of the following: an arithmetic R-format instruction, a **`lw`** instruction, and a conditional branch instruction.

      R-format:
      Critical path: I-Mem □ Reg Read □ MUX (ALUSrc) □ ALU □ MUX (MemReg)

Latency: 225ps + 110ps + 15ps + 100ps + 15ps = 465ps

`lw`:
Critical path: I-Mem □ Reg Read □ MUX (ALUSrc) □
ALU □ D-Mem □ MUX (MemReg) Latency: 225ps
+110ps + 15ps + 100ps + 340ps + 15ps = 805ps

Conditional branch:
Critical path: I-Mem □ Reg Read □ MUX (ALUSrc) □ ALU □ AND □ MUX (PCSrc)
Latency: 225ps + 110ps + 15ps + 100ps + 10ps + 15ps = 475ps

b. Rank the following design approaches in terms of which improve the cycle time the most. You **must** justify your ranking.
   i. Creating word addressable IMEM to eliminate branch / jump address shifting HW
   ii. Implementing quicker memory to have quicker DMEM access times
   iii. Designing a lower-latency control unit

The lw instruction causes the critical path by a wide margin (over 250ps), so only by improving the speed of components on **`lw`**'s critical path will help the total cycle time.

1. ii The DMEM lies on the **`lw`** critical path and is the biggest source of latency, meaning that a reduction in delay will provide the greatest improvement to cycle time.
2. iii While the control unit does output signals that affect dataflow in the **lw** critical path, it operates in parallel to the dataflow itself, and therefore will provide little if any improvement.
3. i While this would improve the latency on the jump / branch paths, our critical path lies with **lw**. Therefore, no improvement would be made with this change.

3. Performance Analysis

You are in charge of selecting processors for computing simple Natural Language Processing (NLP) tasks. You are considering two processors, A and B (the only ones you and your roommates can afford) that have the following CPIs:

Instruction Type	**Cycles per Instruction**	
	Processor A	**Processor B**
Arithmetic, Logical, Shifts, Stores	3	3
Jumps	3	2
Conditional Branch	3	2

Loads	3	5

The following applications are the primary ones that you will need to run on your system:

```
# Application 1:

# This is an implementation of a single stop word index identification #
in a string.

# $a0 contains &string (string is an asciiz array of size string_size)

# $a1 contains &stop_word (stop_word is an asciiz array of size sw_size)
# $a2 contains string_size variable

# $a3 contains sw_size variable



    addiu    $t0, $0, 0

    j outer_cond

outer_loop:

    addiu    $t1, $0, 0

    jal inner_cond

outer_loop_cont:

    subu     $t2, $t1, $a3

    ori      $at, $0, 1

    sltu     $t2, $t2, $at

    addiu $t0, $t0, 1

    beq      $t2, $0, outer_cond

    lui      $at, 0x1001

    ori      $a0, $at, 0x19

    addiu $v0, $0, 4

    syscall

    lui      $at, 0

    ori      $at, $at, 1
    subu     $t0, $t0, $at

    addu     $a0, $0, $t0

    addiu $v0, $0, 1

    syscall

    j exit

outer_cond:

    subu     $t2, $a2, $a3

    slt      $10, $8, $10

    beq      $t2, 1,

    outer_loop

    j exit

inner_loop:

    addu     $t2, $t0, $t1

    addu     $t2, $a0, $t2

    lb $t3, 0($t2)

    addu     $t2, $a1, $t1

    lb $t4, 0($t2)

    subu     $t2, $t3, $t4

    sltu     $t2, $0, $t2

    addiu $t1, $t1, 1

    beq      $t2, 1, outer_loop_cont

inner_cond:

    slt      $t2, $t1, $a3

    beq      $t2, 1, inner_loop

    jr $ra

    exit:
```

# Application 2:

# This is an implementation of simple string tokenizer. # $a1 contains string_size variable

$a2 contains &string (string is an asciiz array of size string_size) # $a3 contains &delimiter (delimiter is a byte array of size 1)

```
outer_cond:

   addiu $t0, $t0, 1

   slt    $t4, $t0, $a1

   beq    $t4, 1, outer_loop

   j exit

outer_loop:

   addu  $t4, $t0, $a2

   lb     $t4, 0($t4)

   lb     $t6, 0($a3)

   subu  $t5, $t4, $t6

   ori    $at, $0, 1

   sltu   $t5, $t5, $at

   beq    $t5, 1, set_ending_index

   addi   $at, $0, 0

   subu  $t5, $t4, $at

   ori    $at, $0, 1

   sltu   $t5, $t5, $at

   beq    $t5, 1, set_ending_index

   j outer_cond

set_ending_index:
   addiu $t3, $t0, 1

   addu  $t1, $0, $t2

   j inner_cond

inner_cond:

   slt      $t4, $t1, $t3

   beq    $t4, 1, inner_loop_print

   addiu $a0, $0, '\n'

   addiu $v0, $0, 0xB

   syscall

   j set_starting_index

set_starting_index:

   addu  $t2, $0, $t3

   j outer_cond

inner_loop_print:

   addu  $t4, $t1, $a2

   lb       $t4, 0($t4)

   addu  $a0, $0, $t4

   addiu $v0, $0, 0xB

   syscall

   addiu $t1, $t1, 1

j inner_cond

exit:
```

a. Consider the two applications above. Calculate the average CPI for each application on each processor (two applications cross two processors means you should have 4 different CPI values). Assume `string_size` is 20 in both applications, and `sw_size` is 3 (i.e., the stop word is “the”) for application 1.

Application 1:
Here an example string of size 20 with stop word “the” of size 3 is chosen: “CPRE381 is the best!” and M is defined as **sw_size**. The loops are designed to break based on the identification of the starting index of the stop word within a chosen string. This is where the 11 is generated (as it is the location of the ‘t’ within “the” in the chosen string) – meaning the loops will not always run in their entirety! Additionally, syscalls are ignored in the calculations but may be mapped to jumps/branches.

```
          A/L/S/S                                    }
           Jump
            addiu       $t0, $0, 0
                                               1
            j outer_cond
          outer_loop:
A/L/S/S     addiu    $t1, $0, 0         } N+1
   Jump     jal inner_cond
          outer_loop_cont:
A/L/S/S     subu $t2, $t1, $a3
A/L/S/S     ori       $at, $0, 1                    } N+1
A/L/S/S     sltu
A/L/S/S
 Branch
A/L/S/S     $t2, $t2, $at addiu
A/L/S/S     $t0, $t0, 1
A/L/S/S     beq$t2, $0, outer_cond lui
               $at, 0x1001
            ori    $a0, $at, 0x19
            addiu $v0, $0, 4
            syscall                           } 1
A/L/S/S     lui   $at, 0
A/L/S/S     ori $at, $at, 1 subu
A/L/S/S        $t0, $t0, $at
A/L/S/S
A/L/S/S     addu  $a0, $0, $t0
            addiu $v0, $0, 1
   Jump     syscall
            j exit
          outer_cond:
A/L/S/S     subu    $t2, $a2,
A/L/S/S     $a3 slt                         } N+1
 Branch
   Jump
            $10, $8, $10
            beq    $t2, 1, outer_loop
            j exit     } 0/1
          inner_loop:
A/L/S/S     addu   $t2, $t0, $t1
A/L/S/S     addu  $t2, $a0, $t2
  Loads     lb     $t3, 0($t2)
A/L/S/S     addu   $t2, $a1, $t1                   } N+M
  Loads     lb     $t4, 0($t2)
A/L/S/S     subu   $t2, $t3, $t4
A/L/S/S     sltu  $t2, $0, $t2
A/L/S/S     addiu $t1, $t1, 1
 Branch     beq $t2, 1, outer_loop_cont
          inner_cond:
A/L/S/S     slt    $t2, $t1, $a3            } N+M+1
 Branch     beq    $t2, 1, inner_loop
   Jump     jr    $ra } 1
          exit:
```

Instruction Type	# Instructions		Frequency	Processor A		Processor B	
				CPI_i	CPI_i*Freq_i	CPI_i	CPI_i*Freq_i
Arithmetic, Logical, Shifts, Stores	9 + 7(11+1) + 6(11+M) + 1(11+M+1) =	192	0.66666667	3	2.0	3	2.0
Jumps	3 + 1(11+1) =	15	0.05208333	3	0.15625	2	0.10416666
Conditional Branch	2(11+1) + 1(11+M) + 1(11+M+1) =	53	0.18402778	3	0.55208333	2	0.36805555
Loads	2(11+M) =	28	0.09722222	3	0.2916666	5	0.4861111

Application 2:

Here an example string of size 20 with delimiter ‘ ‘ (a space) is chosen: “CPRE381 is the best” and M is defined as the number of occurrences of the delimiter AND the null byte at the end of the string (M = 4). Note, the size of the string must include that null byte for this application. N is defined as the string size. Additionally, syscalls are ignored in the calculations but may be mapped to jumps/branches.

```
A/L/S/S   outer_cond:
A/L/S/S     addiu $t0, $t0, 1                      } N
 Branch     slt    $t4, $t0,
   Jump     $a1
            beq    $t4, 1, outer_loop
            j exit     } 1
          outer_loop:
A/L/S/S     addu   $t4, $t0, $a2                                  } N-1
  Loads     lb    $t4, 0($t4)
  Loads     lb  $t6, 0($a3) subu
A/L/S/S        $t5, $t4, $t6 ori
A/L/S/S        $at, $0, 1 sltu
A/L/S/S        $t5, $t5, $at
 Branch     beq    $t5, 1, set_ending_index                       } N-M
A/L/S/S     addi  $at, $0, 0
A/L/S/S     subu   $t5, $t4, $at
A/L/S/S     ori   $at, $0, 1
A/L/S/S     sltu   $t5, $t5,
 Branch     $at
   Jump     beq    $t5, 1, set_ending_index
            j outer_cond    } N-M-1
          set_ending_index:
A/L/S/S     addiu $t3, $t0, 1        } M
A/L/S/S     addu   $t1, $0, $t2
   Jump     j inner_cond
          inner_cond:
A/L/S/S     slt   $t4, $t1, $t3                          } M+N
 Branch     beq   $t4, 1, inner_loop_print
A/L/S/S     addiu $a0, $0, '\n'        } M
A/L/S/S     addiu $v0, $0, 0xB
            syscall
   Jump     j set_starting_index
          set_starting_index:
A/L/S/S     addu  $t2, $0, $t3         } M
   Jump     j outer_cond
          inner_loop_print:
A/L/S/S     addu  $t4, $t1, $a2        } N
  Loads     lb
A/L/S/S
A/L/S/S     $t4, 0($t4) addu
            $a0, $0, $t4 addiu
A/L/S/S     $v0, $0, 0xB
   Jump     syscall
            addiu $t1, $t1, 1
            j inner_cond
          exit:
```

Instruction Type	# Instructions		Frequency	Processor A		Processor B	
				CPI_i	CPI_i*Freq_i	CPI_i	CPI_i*Freq_i
Arithmetic, Logical, Shifts, Stores	6(N) + 4(N-1) + 4(N-M) + 5(M) + 1(M+N) =	304	0.62167689	3	1.86503	3	1.86503
Jumps	1 + 1(N-M-1) + 3(M) + 1(N) =	48	0.09815951	3	0.294478	2	0.196319
Conditional Branch	1(N) + 1(N-1) + 1(N-M) + 1(N+M) + =	79	0.16155419	3	0.484662	2	0.323108
Loads	2(N-1) + 1(N) =	58	0.11860941	3	0.355828	5	0.593047
Total		489	1	Average CPI	3	Average CPI	2.977505

b. Which processor has better performance? *[Careful answering this part…it is a tricksy professor question.]* Provide the quantitative evidence of “better performance” and include a description of how you evaluated the two applications together. What would the relative frequencies have to be between the two processors in order for their performance to be identical (i.e., calculate the “breakeven frequency”)?

Remember that Execution Time = # insts * CPI * cycle time. So, assuming both processors have the same cycle time, Processor B has slightly better performance for BOTH applications since the Average CPI is lower for both applications (3 vs 2.958333 or 2.977505).

In order for the processors to have equal performance on application 1, you would set the Execution Time equations equal:

ET_ProcA = # insts_App1 * CPI_App1_ProcA * cycle time_ProcA= # insts_App1 * CPI_App1_ProcB * cycle time_ProcB = ET_ProcB

Freq_B/Freq_A = CPI_App1_ProcB/CPI_App1_ProcA = 0.986111

Likewise for the processors to have equal performance on application 2: Freq_B/Freq_A = CPI_App2_ProcB/CPI_App2_ProcA = 0.992501

Note that to actually consider what the overall performance of the entire workload on each processor is, you would want to consider the number of expected executions for each application times. If application 1 was 75% of all executions, the relative frequency would be 0.75 * (288 / (288 + 489)) * 0.986111 + (1 - 0.75 * (489 / (288 + 489))) * 0.992501 = 0.430287. Overall, these applications have fairly similar behavior.