CSE30321 Computer Architecture I Fall 2009

Homework 03: Programmable Processors (100 Points)

Assigned: September 15 Due: September 22

1. **(10 Points)**

   Problems 1.13.4 - 1.13.6 in Patterson and Hennessy (p. 69-70)

   The table below shows data for benchmarks run on the AMD Barcelona processor.

   Table 1: Benchmark Data

	Name	Execution time (seconds)	CPI	Clock Rate
a.	sjeng	820	0.96	3 GHz
b.	omnetpp	580	2.94	3 GHz

   1.13.4 If the execution time is reduced by an additional 10% without affecting the CPI and with a clock rate of 4 GHz, determine the number of instructions.

   1.13.5 Determine the clock rate required to give a further 10% reduction in CPU time while maintaining the number of instructions and CPI unchanged.

   1.13.6 Determine the clock rate if the CPI is reduced by 15% and the CPU time by 20% while the number of instructions is unchanged.

2. **(20 Points)**

Problems 1.15.1 - 1.15.6 in Patterson and Hennessy (p. 71-72)

A common pitfall in performance benchmarking is expecting to improve the overall performance of a computer by improving only one aspect of the computer. This might be true, but not always. Consider a computer running programs with CPU times shown in the following table:

Table 2: Benchmark Data

	FP inst.	INT inst.	L/S inst.	Branch inst.	Total time
a.	35 s	85 s	50 s	30 s	200 s
b.	50 s	80 s	50 s	30 s	210 s

1.15.1 By how much is the total time reduced if the time for FP operations is reduced by 20%?

1.15.2 By how much is the time for INT operations reduced if the total time is reduced by 20%?

1.15.3 Can the total time be reduced by 20% by reducing only the time for branch instructions?

The following table shows the instruction type breakdown per processor of a given application executed in a different number of processors.

Table 3: Benchmark Data

	# Processors	FP inst.	INT inst.	L/S inst.	Branch inst.	CPI (FP)	CPI (INT)	CPI (L/S)	CPI (B)
a.	1	560×10^6	2000×10^6	1280×10^6	256×10^6	1	1	4	2
a.	8	80×10^6	240×10^6	160×10^6	32×10^6	1	1	4	2

Assume that each processor has a 2 GHz clock rate.

1.15.4 By how much must we improve the CPI of FP instructions if we want the program to run two times faster?

1.15.5 By how much must we improve the CPI of L/S instructions if we want the program to run two times faster?

1.15.6 By how much is the execution time of the program improved if the CPI of the INT and FP instructions is reduced by 40% and the CPI of L/S and branch is reduced by 30%?

3. **(35 Points)**

This problem relates to the six instruction processor and how best to optimize code given its architecture. This is important because it relates to how compilers implement your code, and additionally how coding choices can impact performance.

Part A:

The problem references the following code sequence.

```
for(i=1; i<5; i++) {
  if ((i < 3) || (i > 3)) {
    x = x + 3;
  }
  x = x + 1;
}
data = x;
```

The 6-instruction processor code for this C-code might look something like that shown below. We assume that this processor can process both positive and negative numbers. What is the average CPI assuming this code is executed from start to finish? (Note, you can answer this question with the help of the state machine shown in Fig. 1).

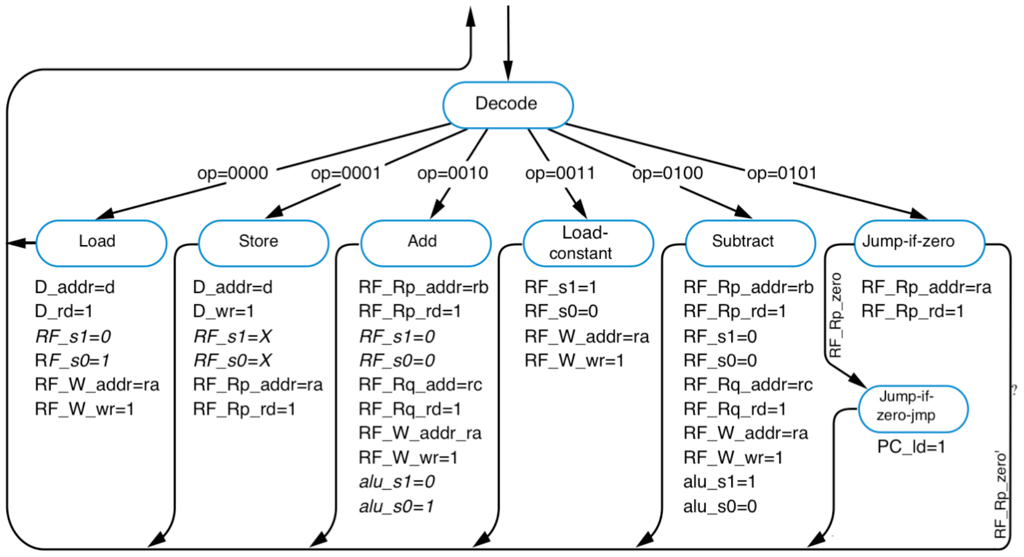

Figure 1: 6 Instruction Processor FSM (starting point for Question 3).

```
        MOV R14, #1        (x)
        MOV R1,  #1        (load constant 1)
        MOV R3,  #3        (load constant 3)
        MOV R5,  #5        (load constant 5)
        MOV R15, #1        (i)
start:  SUB R6, R15, R3
        JUMPZ R6, else
        ADD R14, R3, R14
 else:  ADD R14, R14, R1
        ADD R15, R15, R1
        SUB R6, R15, R5
        JUMPZ R6, done
        JUMPZ R0, start     (R0 always == 0, therefore loop back)
 done:  MOV 7, R14          (d(7) = R14)
```

Part B:
Now, assume that you have 3 new instructions at your disposal – add immediate (like we talked about in class), sub immediate (just like add immediate but this time we subtract instead of add), and jump-equal (where a jump is taken if two register values are equal. (see RTL below – also in terms of CCs, this new instruction mimics the JMPZ instruction.).

- Addi Rx, Ry, #: $R_x \leftarrow R_y +$ constant
- Subi Rx, Ry, #: $R_x \leftarrow R_y -$ constant
- JumpE Rx, Ry, target: If Rx == Ry, goto target; else, goto next instruction

Using these new instructions, we might rewrite the assembly for the above C-code as follows:

```
        MOV R14, #1         (x)
        MOV R3,  #3         (load constant 3)
        MOV R5,  #5         (load constant 5)
        MOV R15, #1         (i)
start:  JUMPE R3, R15, else
        ADD R14, R3, R14
 else:  ADDi R14, R14, 1
        ADD R15, R15, 1
        JUMPE R15, R5, done
        JUMPZ R0, start     (R0 always == 0, therefore loop back)
 done:  MOV 7, R14          (d(7) = R14)
```

What is the new CPI? Does this sequence perform better than the first?

Extra Credit:
Note that in the solution above, load constant instructions are still used instead of SUBi and ADDi instructions. Is the above solution more or less efficient as a result? Why?

Part C:
For both part B, what if each instruction took one extra clock cycle, but we could reduce the clock cycle time for every instruction by 40%? Would implementing this change be a good idea?

4. **(15 Points)**

Oftentimes a trade off must be made between a longer sequence of simple, quick instructions and a shorter sequence of more complex instructions when optimizing for performance or for code size. Refer to the following code sequence for the following question:

```
   MOV R7,  #7
   MOV R2,  #1
   MOV R9,  #0
   MOV R10, #0
X: ADD R9, R9, R7
   SUB R7, R7, R2
   JMPZ R7, Y
   JMPZ R10, X
Y:
```

Part A:
Find the average CPI and instruction count for the above assembly code.

Part B:
Now, assume a JUMP if NOT zero instruction exists (basically the opposite of JUMPZ). Can this instruction be used to make the above code more efficient? Quantify your answer.

5. **(20 Points)**

   Problem 2.2 in Patterson and Hennessy (p. 180-181)

   The following problems deal with translating from C to MIPS. Assume that the variables g, h, i, and j are given and could be considered 32-bit integers as declared in a C program.

   - f = f + f + i;
   - f = g + (j + 2);

   2.2.1 For the C statements above, what is the corresponding MIPS assembly code? Use a minimal number of MIPS assembly instructions.

   2.2.2 For the C statements above, how many MIPS assembly instructions are needed to perform the C statement?

   2.2.3 If the variables f, g, h, and i have values 1, 2, 3, and 4, respectively, what is the end value of f?

   The following problems deal with translating from MIPS to C. For the following exercise, assume that the variables g, h, i, and j are given and could be considered 32-bit integers as declared in a C program.

   - add f, f, h
   - sub f, $0, f; addi f, f, 1

   2.2.4 For the MIPS statements above, what is a corresponding C statement?

   2.2.5 If the variables f, g, h, and i have values 1, 2, 3, and 4, respectively, what is the end value of f?

6. **(Extra Credit)**

Problems 2.4.1-2.4.3 in Patterson and Hennessy (p. 182)

The following problems deal with translating from C to MIPS. Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7 respectively.

- f = g + h + B[4];
- f = g i A[B[4]];

2.4.1 For the C statements above, what is the corresponding MIPS assembly code?

2.4.2 For the C statements above, how many MIPS assembly instructions are needed for perform the C statement?

2.4.3 For the C statements above, how many different registers are needed to carry out the C statement?