CSE 30321 – Computer Architecture I – Fall 2010
Homework 02 – Architectural Performance Metrics – 100 points
Assigned: September 7, 2010 – **Due:** September 14, 2010

Problem 1: (20 points)

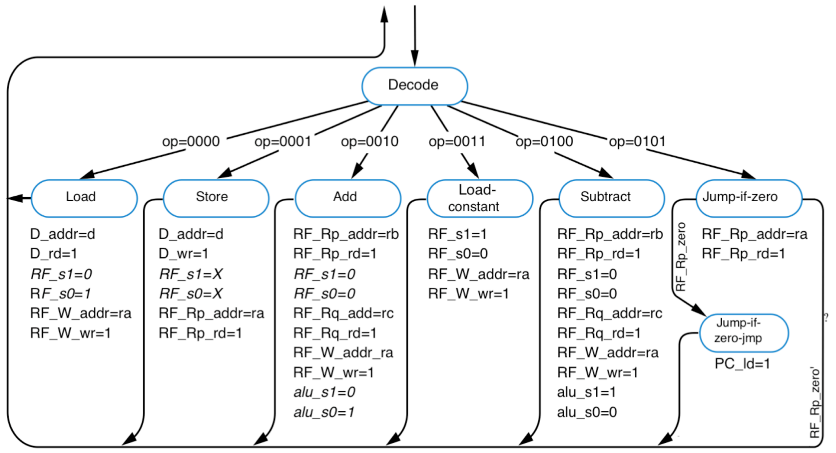
The scope of this 1st problem is more in-line with Lectures 02 and 03 than Lecture 04. It is included in this problem set to re-enforce and review concepts that will be revisited throughout the semester – i.e. register transfer language, when the contents of a register or memory address change, etc.

Parts A and B both refer to the C-code and 6-instruction processor equivalent assembly shown below:

C-Code	6-instruction processor assembly

`if (a == 7) {`
`  b = b + 1;`
`}`
`else {`
`  b = b – 1;`
`}`
	# Assume a maps to R10, b maps to R11, R0 = 0
	0: MOV R7, #7
	1: MOV R1, #1
	2: SUB R2, R7, R10
	3: JUMPZ R2, 3
	4: SUB R11, R11, R1
	5: JUMPZ R0, 2
	6: ADD R11, R11, R1
	7:

For your reference, the finite state diagram for the 6-instruciton processor is also shown below:



Part A: (15 points)

For all of the *executed instructions*, write out the basic register/memory transfer operations that occur during each clock cycle. Also, for each clock cycle, provide the contents of the registers/memory locations whose contents have changed. Assume R10 =7 and R11 = 12.

An example of how to get started is included below:

Cycle	Events and Register File (RF) /Memory Contents
0000	IR = I[0] ■ PC ← PC + 1
0001	IR = 0x3707 ■ PC = 1
0002	RF[1] = 7; (*initiate* register transer)
…	

Part B: (5 points)

During clock cycle 12, what state is the processor in? (i.e. "Fetch", "Decode", "Add", etc.)

Problem 2: (25 points)
Assume that a computer has 4 different classes of instructions:
1. Floating Point (i.e. for non-integer operations)
2. Integer (i.e. for general, integer addition and subtraction)
3. Load / Store (i.e. to fetch data from memory and write data to memory)
4. Branch (i.e. jump instructions that take you to different parts of a program)

Some instructions require more work than others and may require a different number of clock cycles (or percentage of the total program time). To make an analogy to the 6-instruction processor, you could say that:
1. ADD/SUB represents one class of instructions (each of which takes 3 CCs to execute)
2. MOV (constant) and MOV (load/store) represents another class (and all instructions in this class also require 3 CCs to finish)
3. JUMPZ represents a third class – and may take 3 or 4 CCs to complete depending on the state of the referenced register.

Part A: (12 points)
Assume that a given program takes 173 s to finish. As a computer architect, you have been asked to make this program run faster – and can make 1 of 3 design decisions:
1. Improve the processor hardware to make your floating-point unit 3X faster.
2. Improve the processor hardware to make your integer unit 1.2X faster.
3. Change your processor hardware *and* compiler to support a new instruction that will immediately allow data loaded from memory to be used in an arithmetic operation. With this new instruction, the time spent executing integer instructions can be reduced by 5% and the time spent executing load/store instructions can be reduced by 3%.

You may assume that:
- 7% of the program time is spent executing floating point instructions
- 45% of the program time is spent executing integer instructions
- 32% of the program time is spent executing load/store instructions
- 16% of the program time is spent executing branch instructions

Which option – if any – is best?

Part B: (3 points)
What insight does your answer to Part A provide about what type of enhancement (if any) may be best?

Part C: (10 points)
Note: Except for the fact that we assume the same 4 classes of instructions – floating point, integer, load/store, and branch – this part of the question is unrelated to Part A and Part B.

Assume that a 2.33 GHz microprocessor is used to execute a program where the mix of instructions is as shown in the table below. The finite state diagram for this processor is also shown below.

Table: Percentage of each instruction class for program in question.

Instruction Class	Floating Point	Integer	Load / Store	Branch
% of program	2%	49%	27%	22%

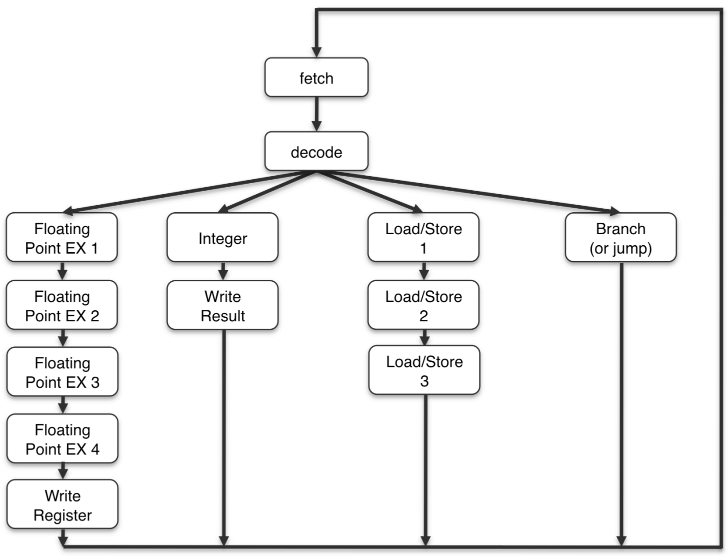


Figure: Finite state diagram for processor in question.

As before, as a computer architect, you have been asked to make this program run faster – and can make 1 of 3 design decisions:

1. Reduce the number of clock cycles required for *any* load/store instruction from 5 to 4 – but at the expense of a 2.15 GHz clock rate for *every* instruction.
2. Eliminate all floating-point instructions with no change to the clock rate. The new instruction mix is now: 51% Integer, 27% Load/Store, 22% Branch.
3. Improve the clock rate to 2.5 GHz for *every* instruction – but at the expense of more CCs for *some* instructions. Floating Point, Integer, Load/Store, and Branch instructions will now take 7, 5, 5, and 3 CCs respectively.

Which design option – if any – is best?

<u>Problem</u> <u>3</u>: (35 points)
The premise behind this problem is a compilation technique called loop unrolling – which can improve performance by reducing the overhead of instructions that manage the loop itself (albeit at the expense of code density). An example of a C-based for loop and the equivalent "rolled" and "unrolled" assembly is shown below. The assembly versions are based on 6-instruction processor code. However, I assume the existence of a MIPS-like load and store. (In other words, no self-modifying code is used and it is assumed that the data in a register could be used as an address to memory.)[1]

C-Code	"Rolled" 6-instruction assembly	"Unrolled" 6-instruction assembly

`N = 50,000`
`for (i=0; i<N; i++)`
`{`
`  z(i) = z(i) + a;`
`}`
	MOV R1, #1
	MOV R2, #0
	MOV R3, #N
	MOV R4, #a
	Loop: Load R5, M(i)
	ADD R5, R5, R4
	Store M(i), R5
	ADD R2, R2, R1
	SUB R6, R3, R2
	JUMPZ R6, Out
	JUMPZ R0, Loop
	Out:
		MOV R1, #4
		MOV R2, #0
		MOV R3, #N
		<u>MOV R4, #a</u>
		Loop: Load R5, M(i)
		ADD R5, R5, R4
		<u>Store M(i), R5</u>
		Load R5, M(i+1)
		ADD R5, R5, R4
		<u>Store M(i+1), R5</u>
		Load R5, M(i+2)
		ADD R5, R5, R4
		<u>Store M(i+2), R5</u>
		Load R5, M(i+3)
		ADD R5, R5, R4
		<u>Store M(i+3), R5</u>
		ADD R2, R2, R1
		SUB R6, R3, R2
		JUMPZ R6, Out
		JUMPZ R0, Loop
		Out:

<u>Part</u> <u>A</u>: (5 points)
By looking at the rolled and unrolled code given above, in 40 words or less, explain how loop unrolling can improve execution time.

<u>Part</u> <u>B</u>: (15 points)
Write an expression for the number of clock cycles required to execute the for loop as a function of the number of copies of the main body. The expression should contain the variable *x* – which can represent the number of copies of the main body (i.e. the number of times the loop is unrolled), and the variable N. Note that the maximum number of times that a loop can be unrolled is the number of iterations required – N in the C-code above.

To do this, you will need to know the number of clock cycles required for each instruction – and can assume the clock cycle times associated with the 6-instruction processor discussed in Lectures 02 and 03. Additionally, you can assume that the new Load and Store instructions also take just 3 CCs.

In your expression, you *do not* need to account for cases where the number of times a loop is unrolled is not a multiple of N.

After you have written your expression, create a graph where N is 50,000.

[1] This simplification is made to more easily explain the implementation and benefits of loop unrolling. Also the focus of *this* problem is *not* on how to implement a for loop. Finally, you need not be concerned with issues of overflow, etc.

Part C: (10 points)
From your graph in Part B, about how many times would you unroll this loop? There is no one "right number". Credit for this question will depend on how well your answer is justified.

Hint: Plot the total number of clock cycles against *instruction count* instead of the number of times that the loop is unrolled. To do this, write another expression for instruction count as a function of N and x.

Part D: (5 points)
Assume that for Parts A-C, the clock rate of the processor being studied was 2 GHz. We have the design option of making the clock rate 2.15 GHz – but now each instruction will take 1 CC longer. Will an unrolled loop benefit from this design change?

Problem 4: (10 points)
Given the instruction mixes, CPI, clock rate, and instruction counts shown below, calculate the MIPS rating (millions of instructions per second – not the new ISA we are discussing in Lecture 05) for each machine below. Assume the instruction mixes on Machine 1 and Machine 2 perform the exact same task. Which machine do you think is better? Why?

	Floating Point	Integer	Load / Store	Branch	Clock Rate	Instruction Count
Machine 1 Percentage	7%	40%	33%	20%	2.33 GHz	1,524,412,932
Machine 1 CPI	7	4	5	3		
Machine 2 Percentage	5%	55%	17%	23%	2 GHz	1,482,829,427
Machine 2 CPI	8	5	5	3		

Problem 5: (10 points)
You are currently running a program on a dual (2) core microprocessor. To improve performance, you are contemplating buying a new 8-core microprocessor.

You would like your overall execution time to improve by 3X. How much of your original code must you parallelize?