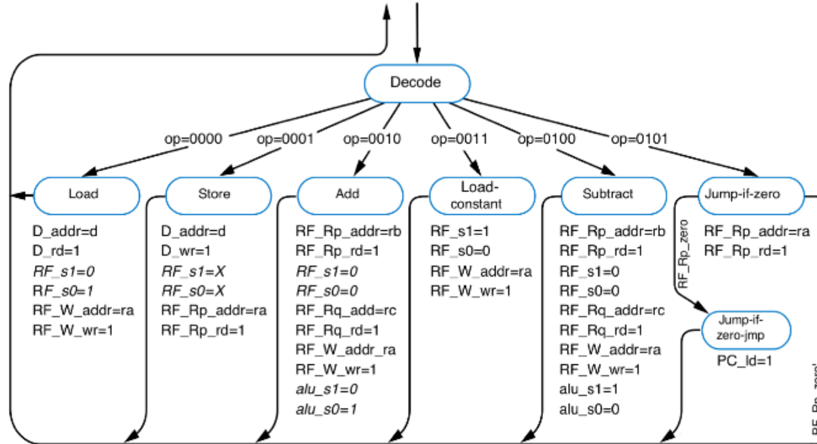
# <u>CSE 30321</u> – <u>Computer Architecture I</u> – <u>Fall 2011</u>
## Homework 02 – Architectural Performance Metrics – 70 points
**Assigned:** September 6, 2011 – **Due:** September 13, 2011

## <u>Problem 1</u>: (15 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 6-instruction processor equivalent assembly shown below:

| 6-instruction processor assembly |
|---|
| # Assume R0 = 0 |
| 0: MOV R6, #7 |
| 1: MOV R1, #1 |
| 2: SUB R2, R6, R1 |
| 3: ADD R2, R2, R2 |
| 4: JUMPZ R2, E |
| 5: SUB R11, R11, R1 |
| 6: JUMPZ R0, E |
| 7: ADD R1, R11 R1 |
| E: |

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




<u>Part A</u>: (9 points)
For instructions 1, 2, and 3, 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.

An example of how to get started (for instruction 0) is included below:

| Cycle | Events and Register File (RF) /Memory Contents |
|---|---|
| 0000 | IR = I[0] ■ PC ← PC + 1 |
| 0001 | IR = 0x3607 (in hex) ■ PC = 1 |
| 0002 | Determine/calculate value to go into R6 (value will be 7) |

<u>Part B</u>: (6 points)
*During* clock cycle 15, what state is the processor in? (i.e. "Fetch", "Decode", etc.)

**<u>Problem 2</u>: (20 points)**
**Setup:**
In Problem #5 of Homework #1, we considered a subset of the ARM ISA. One of the ideas discussed in this problem was that of "condition codes." More specifically, when certain instructions were executed, bit-sized flags would be set depending upon the result that the instruction produced. For example, consider the CMP (or compare) instruction:

- Assume that R1 = 10 and R2 = 9.
- Status flags are set based on the value produced by R1 – R2.
- Thus, after CMP R1, R2 is executed:
    - If GT condition is tested, the result will be TRUE, as R1 is > R2
    - If GE condition is tested, the result will be TRUE, as R1 is ≥ R2
    - If LT condition is tested, the result will be FALSE, as R1 is not < R2
    - If LE condition is tested, the result will be FALSE, as R1 is not ≤ R2
    - If EQ condition is tested, the result will be FALSE, as R1 ≠ R2
    - If Mi condition is tested, the result will be FALSE, as the result is positive / not negative
        - (Mi stands for "minus")
    - If Pl condition is tested, the result will be TRUE, as the result is positive
        - (Pl stands for "plus" and it is TRUE if a checked value is positive OR zero)

Interestingly, when a normal ADD or SUB instruction is executed, condition codes are NOT set. For example, if the instruction SUB R3, R4, R4 were executed, R3 would obviously contain the number 0. However, the existing state of the EQ flag would not change.

However, another type of ADD or SUB instruction can also set condition codes in addition to performing an add or subtract operation (as normal). For example, if the subtract instruction SUB R3, R4, R4 is replaced with the instruction SUBS R3, R4, R4, the following would be done:

- R3 would be set to 0
- The EQ flag would we be set to TRUE

Even more interestingly, add and subtract instructions exist that are conditionally executed based on the value of a condition code. For example, given the instruction SUBGT R3, R4, R4, the contents of R3 would only be changed if the GT flag were TRUE (or '1') – otherwise the value of R3 would remain the same (i.e. for the case where GT is FALSE or '0').

<u>Part A</u>: (15 points)
In Table 2.1, I have provided two translations of the C-code shown below into ARM assembly. (One translation uses instructions like SUBGT, and the other does not.)

- Assume that EVERY instruction –branches, jumps, conditional adds, etc. – take 3 CCs.
- **Is the new code faster than the old code? If so, calculate a speedup and explain – very specifically – where the speedup comes from.**

<u>C-code</u>:

```
for(i=0; i<1000; i++) {
      y(i) = y(i) + A*x(i);
      if y(i) > 0
            a = a + 1;
      else
            a = a – 1;
}
```

Table 2.1

| Program without conditional ALU instructions | | | | Program with conditional ALU instructions | | |
|---|---|---|---|---|---|---|
| | MOV R1, R0 | i maps to R1, initialize i to 0 | | | MOV R1, R0 | Same |
| loop | CMP R1, #1000 | compare R1 to the #1000, set condition flags | | loop | CMP R1, #1000 | Same |
| | BGE end | if i ≥ 1000, exit loop | | | BGE end | Same |
| | LDR R4, [R1] | load y(i) | | | LDR R4, [R1] | Same |
| | LDR R5, [R1+1000] | load x(i) | | | LDR R5, [R1+1000] | Same |
| | MULT R5, R5, R6 | (assume R6 = A) A*x(i); | | | MULT R5, R5, R6 | Same |
| | ADD R4, R5, R4 | y(i) + A*x(i) | | | ADDS R4, R5, R4 | Same; set code |
| | ADDi R1, R1, #1 | update counter | | | ADDi R1, R1, #1 | Same |
| | CMP R4, R0 | compare y(i) with Q | | | SUBiMi R10, R10, #1 | Set R10 only if Mi check is true |
| | BPl plus | goto “plus” if y(i) > Q (i.e. R4 > R0) | | | ADDiPl R10, R10, #1 | Set R10 only if Pl check is true |
| | SUBi R10, R10, #1 | otherwise, subtract 1 from a (assume R10 = a) | | | B loop | |
| | B loop | start loop again | | end | | |
| plus | ADDi R10, R10, #1 | add 1 to a, if condition met (assume R10 = a) | | | | |
| | B loop | start loop again | | | | |
| end | | | | | | |

Part B: (5 points)
Now, assume that condition flags are analyzed in the 2$^{nd}$ clock cycle of instruction execution as opposed to the 3$^{rd}$? Thus, an instruction like AddiPl only takes 2 CCs instead of 3 CCs. How does this change your answer to Part A?

**Problem 3: (15 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 2 hours 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 5X faster.
2. Improve the processor hardware to make your integer unit 1.5X 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 20% and the time spent executing load/store instructions can be reduced by 15%.

You may assume that:

- 10% of the program time is spent executing floating point instructions
- 35% of the program time is spent executing integer instructions
- 30% of the program time is spent executing load/store instructions
- 25% 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?

## Problem 4: (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 Table 4.1. The finite state diagram for this processor is also shown below.

**Table 4.1: Percentage of each instruction class for program in question.**

| **Instruction Class** | Floating Point | Integer | Load / Store | Branch |
|---|---|---|---|---|
| **% of program** | 10% | 50% | 25% | 15% |

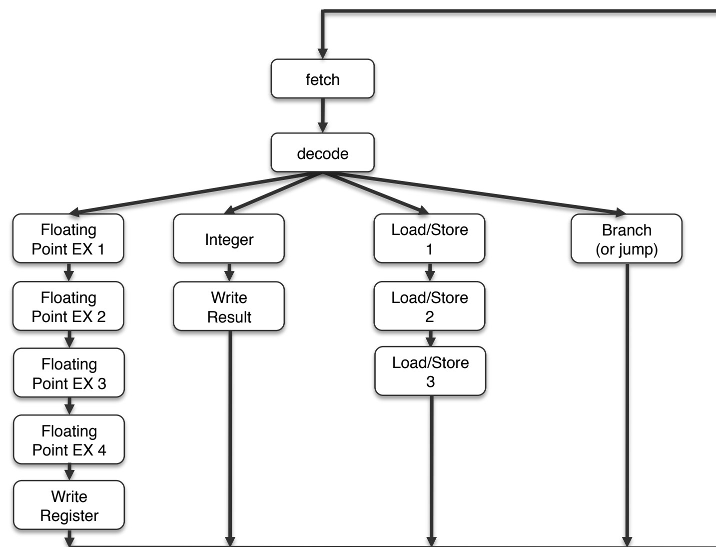



**Figure: Finite state diagram for processor in question.**

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* integer instruction from 4 to 3 – 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: 60% Integer, 25% Load/Store, 15% Branch.
3. Improve the clock rate to 2.5 GHz for *every* instruction – but at the expense of a 4-cycle branch instruction.

Which design option – if any – is best?

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

You would like your overall execution time to improve by 6X. How much of your original code must you parallelize?