<u>CSE 30321 – Computer Architecture I – Fall 2009</u>
Lecture 12– Review

October 1, 2009

<u>Beginning of Lecture 12</u>:

- Tie together everything we've done so far.
- Approaching (a) material transition and (b) midterm exam and its important to revisit the question "WHY?"
    1. In middle of midterms, projects, etc. and may fail to look up
    2. Sometimes you can sit through an entire class and not realize why the university makes you take it!

<u>Lectures 1-4</u>:

Lectures 1-4 dealt with the Vahid processor

1. Designed to give you simplified snapshot of how a HLL/language-like statement is eventually executed/done on chip
2. Also provides a framework for you to apply course principles in manageable labs

Let's talk about #1 though…

- We used example A = B + C;
    - Assume that A, B, and C are variables that are stored in memory
    - Using Vahid processor code, you would need the following assembly instructions to execute A = B+C;
        - MOV R1, B   # load B into Register #1
        - MOV R2, C   # load C into Register #2
        - ADD R3, R1, R2   # add R1 and R2 together and store the result in R3
        - MOV A, R3   # store the data in Register #3 into memory location A

Might also think about "Add R3, R1, R2" as:

< opcode > < destination > < source 1 > < source 2 >

or:

< 0010 > < 0011> <0001> <0010>

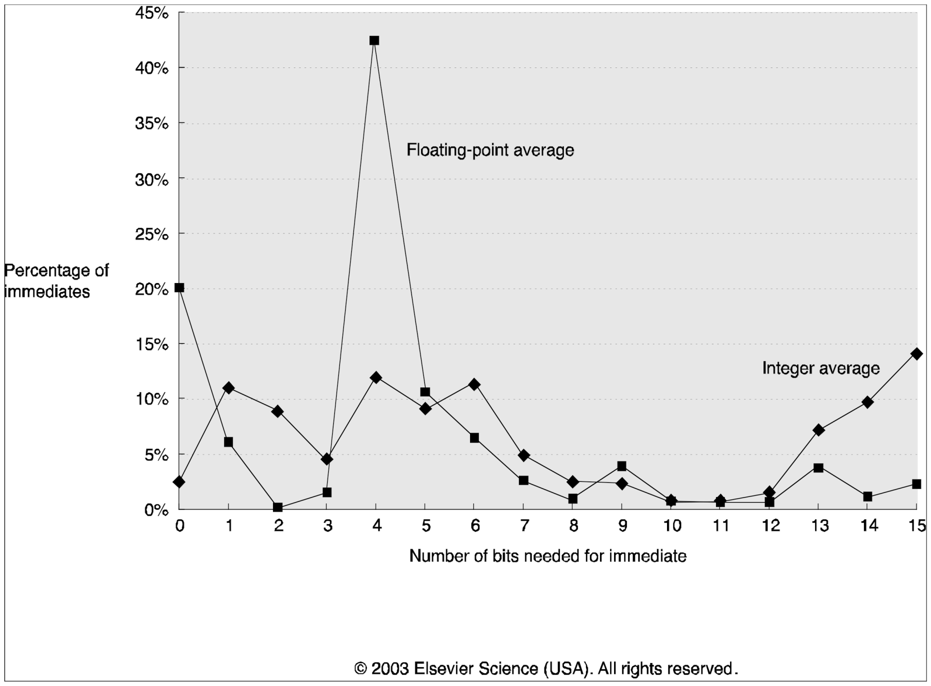
- Add R3, R2, R1 is easy for a human to interpret
- Sequence of 1s and 0s easier for computer to interpret
- Both tell us something about the HW that our HLL code is going to execute on…
    - E.g. number of registers, whether or not you can add immediate value, etc.
    - Note: programmer can say what register stuff goes into, but cannot directly manipulate the IR, PC, a specific multiplexor, etc. (these are not "programmer visible")

When you compile HLL code, you produce a set of instructions… which is really a sequence of 1s/0s

- Take away:
    - Your compiler has to know something about the HW that is available and also something about HLL syntax

In computer architecture, we often think about *adding some HW to make HLL construct more efficient*.

- In your lab, in class, and in the HW, we did examples of this
- Revisit i = i + 1;
    - We can profile compiled, HLL code to see how big of a "constant value" is frequently used – see example on next page:



- For Integer Instructions, ~82% of instructions need less than 15 bits!
  - If you think about this in context of Vahid processor, 82% of the time you could use one instruction instead of 2.
    - If all instructions take the same time, performance is improved.

Now, think about how you would "add a constant" with Vahid code:
- Have to have some way to get constant into memory in the first place!
- Then, we would use up one of our 16 registers to hold that constant

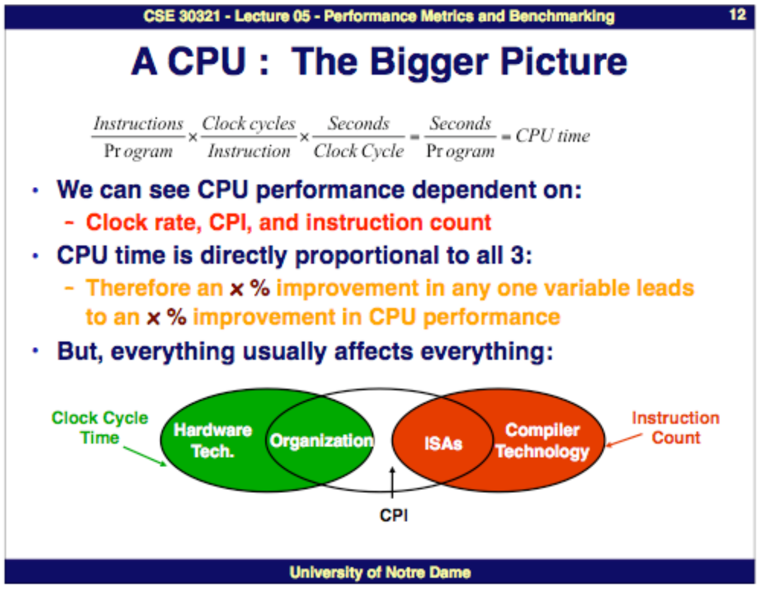
Clearly, we could (a) more efficiently use registers and (b) eliminate the overhead of loading from memory if we could encode an immediate value into the instruction itself.
- Of course, we have to have the HW to support this
- In Labs / HWs, you had to do this several times
  - Most often, you could just think about what new instruction has to do, and then modify the datapath HW accordingly.
    - E.g. If we encode a constant in an instruction, we need a path from the IR (where the instruction is stored) to the ALU, need between multiple inputs to the ALU, if the ALU adds 16-bit #s need to append 0s to the constant, plus still do other stuff like PC ← PC + 1
      - As a programmer, you interface to this with addi instruction generated by compiler

When deciding what HLL language construct to support, a good rule of thumb is to make the common case fast and efficient; probably not worth HW for rare case.

Lectures 5-6:
The above discussion implies design decisions are motivated by performance!
- Spent Lectures 5 & 6 explaining how to quantify if a (HW) design change or decision was good
  - (Or how good it was)
  - (Useful for determining if new instruction really will make your HLL code faster)

Two really important ideas:

- "Making the common case fast" is another way of expressing Amdahl's Law
- CPU time is the best performance metric
    - CPU time takes into account things like instruction count, CPI, CC time, etc.
    - Recall…



CSE 30321 - Lecture 05 - Performance Metrics and Benchmarking 12

A CPU : The Bigger Picture

$$\frac{Instructions}{Program} \times \frac{Clock\ cycles}{Instruction} \times \frac{Seconds}{Clock\ Cycle} = \frac{Seconds}{Program} = CPU\ time$$

- We can see CPU performance dependent on:
    - Clock rate, CPI, and instruction count
- CPU time is directly proportional to all 3:
    - Therefore an **x** % improvement in any one variable leads to an **x** % improvement in CPU performance
- But, everything usually affects everything:

University of Notre Dame

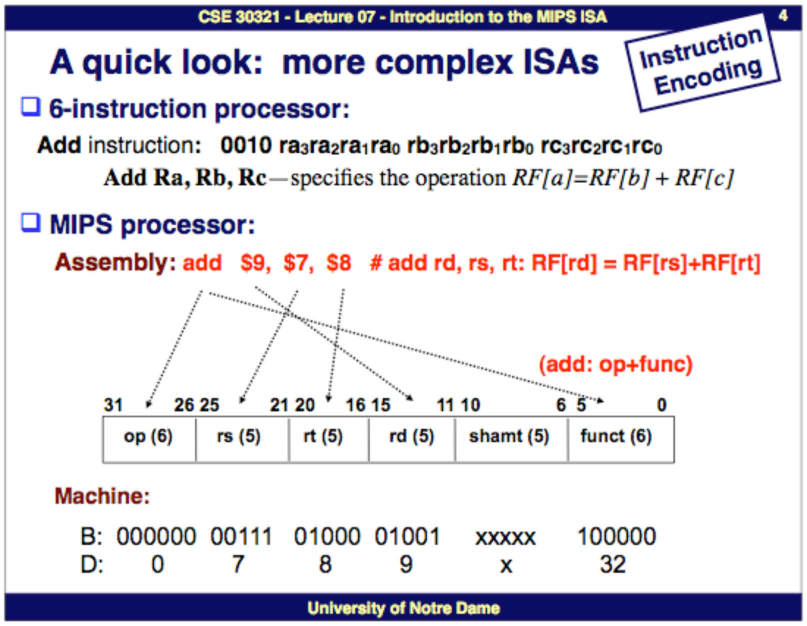
- As example, consider the following tradeoff:
    - You want to do i = i + 1
    - One processor executes the Vahid ISA, another executes the MIPS ISA
    - Both have clock rates of 1 GHz
    - The Vahid processor:
        - Needs to use a load constant instruction
        - Needs to use an add instruction
        - Each instruction takes 3 CCs (see FSM)
    - The MIPS processor
        - Can use an addi instruction
        - It takes 4 CCs
    - The Vahid time is:
        - 2 instructions X 3 CCs / instruction X Clock_Period
    - The MIPS time is:
        - 1 instruction x 4 CCs / instruction X Clock_Period
            - Actually, 1 more CC/instruction, but faster
    - MIPS is 1.5 X faster

<u>Lectures 7-8</u>:

- How a HLL statement executed on chip essentially captured in Lectures 1-4
- However, Vahid ISA not nearly sophisticated enough to support modern HLLs
    - For example, can only address 256 memory locations
    - Modern machine has gigabytes of memory!
- MIPS is a good example of a modern ISA

Still, despite disparity in sophistication, principles of how information is processed is quite similar:

- see 6-instruction vs. MIPS slide



CSE 30321 - Lecture 07 - Introduction to the MIPS ISA 4

A quick look: more complex ISAs

Instruction Encoding

- 6-instruction processor:

  Add instruction: **0010 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

  Add Ra, Rb, Rc—specifies the operation *RF[a]=RF[b] + RF[c]*

- MIPS processor:

  Assembly: `add $9, $7, $8 # add rd, rs, rt: RF[rd] = RF[rs]+RF[rt]`

  (add: op+func)

31 26	25 21	20 16	15 11	10 6	5 0
op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)

Machine:

```
B:  000000  00111  01000  01001   xxxxx    100000
D:     0      7      8      9       x        32
```

University of Notre Dame

(Function code allows for more than 2^6 (or 64) instructions)

Besides the add instruction shown above, we also explained how MIPS did things like conditional branches, loaded data from memory, stored data in memory, etc.

<u>Lectures</u> <u>9</u>-<u>10</u>:

- Used MIPS instructions to do simple things – like A = B + C; – but also more complex things (like recursive factorial)
- Also talked a lot about support for MIPS-based procedure calls

Why focus on things like procedure calls and register conventions?

- Actually, helps to make compiled assembly code much more compact
- Also makes it easier for your code to (for example) leverage system calls
  - Example: What if you call someone else's code (like a printf() function)
    - If no callee convention, every time printf() is called, your compiled code would have to include assembly languages to save registers
    - With callee convention, these instructions only need to appear once

<u>Lectures</u> <u>11</u>-<u>12</u>:

Now, we'll derive a datapath to execute the MIPS instructions we've talked about so far

- Before:
  - simple datapath was given and augmented to support new instructions
- Now:
  - Derive datapath to support MIPS instruction types

Efficiency is important!

- You'll see one example in your HW
- I'll do another here.

Example:

- Think about hardware to increment the program counter (PC)
  - What we did on Tuesday when we started to develop the MIPS datapath is to include a 2nd adder to just increment the PC
- Let's consider what might happen if this were *not* there…
  - As a result, each instruction takes 1 CC more

You can assume the following:

Instruction Type	Frequency	CCs with PC adder	CCs without PC adder
Immediate	20%	4 CCs	5 CCs
Load	15%	5 CCs	6 CCs
Store	8%	4 CCs	5 CCs
Branch/Jump	33%	3 CCs	4 CCs
Register ALU	24%	4 CCs	5 CCs

CPU time (old) =

$$I \times (.20(4) + .08(4) + .15(5) + .33(3) + .24(4)) \times T = 3.82 \times I \times T$$

CPU time (new) =

$$I \times (.20(5) + .08(5) + .15(6) + .33(4) + .24(5)) \times T = 4.82 \times I \times T$$

What if $I = 10^{13}$ and $T = 1 \times 10^{-9}$?

$4.82 \times 10^{13} \times 10^{-9} - 4.82 \times 10^{13} \times 10^{-9} = 10000$ seconds (or 2.78 hours longer!)