PS1 Review, Lab 2 Overview

2/11/2016

Section 4

Colin Schmidt

Agenda

- Quiz 1 Prep
- Problem Set 1 Review
  - ISAs
  - Microcode
  - Pipelines
  - Branch Prediction
  - CISC v RISC
  - Iron Law
- Lab 2 Overview

Quiz 1 Prep

- Next Wednesday (2/18)
- Completely closed books (no cheat sheets, only pencils)
- Similar to the problem set

Quiz 1 Prep

- Possible Topics
  - ISA design
  - Microcoding
  - Pipelining
    - Bypassing/interlocking
    - Precise exceptions
    - Control hazards
    - Branch speculation (BTB,BHT,return stacks)
  - Iron Law
- How to study?
  - Fully understand PS1
  - Go over past tests
  - Read book!
  - Go over lecture slides
  - Argue with a friend

PS1: ISA Question

- It depends
  - Different machines are good at different things and can be used in different circumstances
  - Can construct programs that make most machines look good
    - Benchmarks are important
  - For C code by a modern compiler almost always RISC
- Optimizing code by hand is important
  - Understand peak efficiency of your machine

Microcode

- For microcode problems, key is to get the pseudocode right
  - Control signals follow readily from pseudocode
- Sanity checks:
  - Only one device may drive the bus
  - The bus probably should be driven every cycle
  - Don’t read from a register whose write-enable was a don’t-care

Microcode

- don’t cares
  - If you won’t read A/B/MA registers again, their write-enables should be don’t-cares
  - If enMem is off, Mem Wr is a don’t-care
  - If enReg is off, Reg Wr is a don’t-care
  - If you *will* read rd,rs1,rs2 in the future, keep ldIR == 0

ADDm

- M[rd] <~~ M[rs1] + M[rs2]
  - MA <- R[rs1]
  - A <- Mem
  - MA <- R[rs2]
  - B <- Mem
  - MA <- R[rd]
  - Mem <- ALU (A+B);
  - uBR=J to Fetch
- Note efficiency: 10 cycles vs. 30 for ld,ld,add,st

Strcpy

- STRCPY
  - MA <- Rs; A <- Rs
  - B <- Mem
  - MA <- Rd
  - Mem <- B
  - If (B == 0) uBr to FETCH0
  - Rs <- A + 4
  - A <- Rd
  - Rd <- A+4, J to STRCPY

```
strcpy:
lw x2, 0(x3)
sw x2, 0(x4)
beq x2, x0, exit
addi x3, x3, 4
addi x4, x4, 4
jal x0, strcpy
exit:
```

Pipeline

- Think about hazards when pipeline questions come up
- Understand why 5 stage is well-balanced
  - What modifications affect that?
- Precise exception are important interface to programmer
- Tradeoffs
  - stalling vs bypassing vs speculating

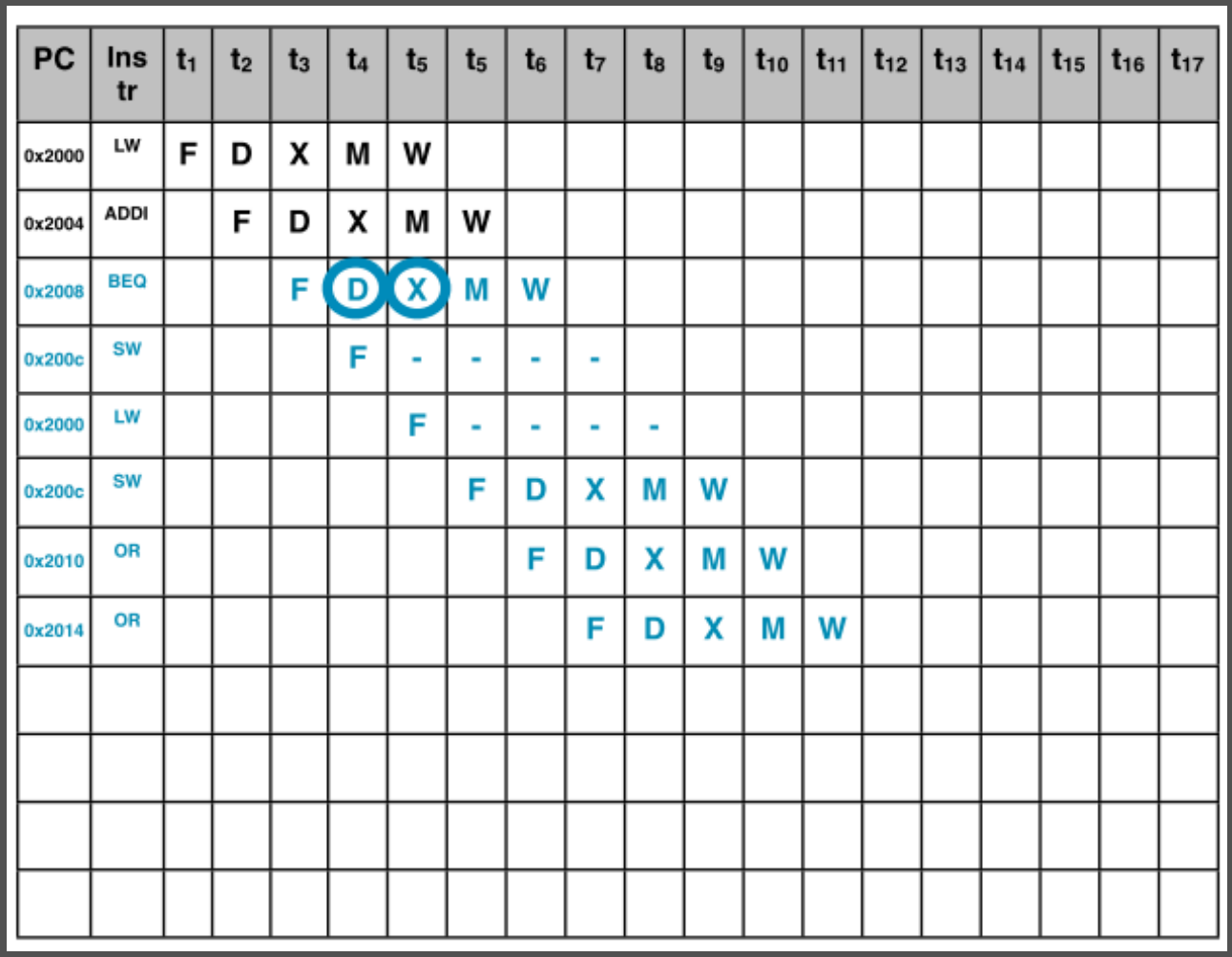
Branch Prediction

- Where does control flow resolve
- Pipeline diagram for control flow
- BHT predicts taken/not taken
  - Resolves 1 cycle earlier than no BHT on correct
- BTB predicts target
  - Resolves 1 cycle earlier than BHT
- Think about how these interact with different pipeline depths

Pipeline Diagram

- 0x2000: LW x7, 0(x6)
- 0x2004: ADDI x2, x2, 1
- 0x2008: BEQ x2, x3, 0x2000
- 0x200c: SW x7,0(x6)
- 0x2010: ORI x5, x5, 4
- 0x2014: ORI x7, x7, 5

Assume: BTB has 0x2008 valid with target=0x2000
x2 = 0 and x3 = 2



PC	Instr	t_1	t_2	t_3	t_4	t_5	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}
0x2000	LW	F	D	X	M	W													
0x2004	ADDI		F	D	X	M	W												
0x2008	BEQ			F	D	X	M	W											
0x200c	SW				F	-	-	-	-										
0x2000	LW					F	-	-	-	-									
0x200c	SW						F	D	X	M	W								
0x2010	OR							F	D	X	M	W							
0x2014	OR								F	D	X	M	W						

Problem 4.D **Adding a BTB**

BTB mispredicts the exit, and it takes two cycles for branch logic in Exe to catch the mistake.

The first circle is drawn to show when the BTB had a hit and predicted “taken”. The second circle in t3 shows when the branch comparison catches a mispredict and kills two cycles.

CISC vs RISC

- Bad/No compiler -> CISC
- Good Compiler -> RISC
- Hardware limitations affect choice
  - Fast logic, slow memory -> CISC
- High performance implementations -> RISC
  - Hard to deeply pipeline complexities
  - Dynamically scheduling around CISC is hard
  - CISCs are now internally RISC

Iron Law

- Think about whole system
  - Compiler/Programmer, Pipeline, Hazards, Memory, Critical Paths, etc.
  - Although sometimes no effect on parts
- Any questions on what affects
  - Instruction/Program
  - Cycles/Instrucion
  - Time/Cycle

		Instructions / Program	Cycles / Instruction	Seconds / Cycle	Overall Performance
a)	Adding a branch delay slot	Increase: Nops must be inserted when the branch delay slot cannot be usefully filled.	Decrease: Some control hazards are eliminated; also additional NOPs execute quickly because they have no data hazards.	No effect: doesn't change pipeline Decrease: branch_kill signal is no longer needed	Ambiguous: Depends on the program and how often the delay slot can be filled with useful work
b)	Adding a complex instruction	Decrease: if the added instruction can replace a sequence of instructions. No effect: if it is unusable.	Increase: if implementing the instruction means adding or re-using stages. No effect: if the number of cycles is kept constant but it just lengthens the logic in one stage.	Increase: since more logic and thus longer critical path. No effect: if it is implemented by more or re-used stages but each stage gets no longer.	Ambiguous: if the program can take advantage of the new instruction, it can mitigate the costs of implementing it. This is a hard decision for an ISA designer to make!
c)	Reduce number of registers in the ISA	Increase: values Will more frequently be spilled to the stack, increasing number of loads and stores	Increase: more loads followed by dependent instructions, will cause stalls, and likely be difficult to schedule around	Decrease: fewer registers means shorter register file access time	Ambiguous: if the program uses few registers and thus spills rarely to memory, the faster reg. access times may win out. Also, your instructions may be able to be shorter, improving amongst other things code density and I$ hit-rates.

d)	Improving memory access speed	No effect: since instructions make no assumption about memory speed.	Decrease: if access to Memory is pipelined (>1 cycle) since it will now take less cycles. No effect: if memory access is done in a single cycle.	Decrease: if memory access is on the critical path or memory was 1 cycle. No effect: if memory is pipelined and just takes less cycles.	Improve: improving memory access time, at least by these Iron Law metrics, will increase erformance of the whole system (unless you chose "no effect" for everything). Of course, there could be other secondary costs of improving mem. access speeds, like having to use smaller caches, but I'm getting carried away here.
e)	Adding 16-bit versions of the most common instructions in MIPS (normally 32-bits in length) to the ISA (i.e., make MIPS a variable length ISA)	No effect: because you are replacing 32b instructions with equivalent 16b versions, it saves on code space, but it leaves the Inst/ Program count unchanged	No effect: you are simple executing equivalent 16b versions of regular 32b instructions. Both appear identical to the pipeline. decrease: since code size has shrunk, I$ hits will increase and thus less cycles will be spent fetching instructions	Increase: decode may increase this since the instruction format is more complex (and you have to deal with figuring out where the instruction boundaries are) No effect: if this fits within the cycle time, since this makes no change to the pipeline and only increases the decode stage (or perhaps adds another stage to the front-end).	Ambiguous: the main advantage is smaller code size, whichcan improve I$ hit rates and save on fetch energy (get more instructions per fetch). This can improve performance (or at least energy), however the more complex decode could also counteract these gains.
f)	For a given CISC ISA, changing the implementation of the micro-architecture from a microcoded engine to a RISC pipeline (with a CISC-to-RISC decoder on the front-end)	No effect: because the ISA is not changing, the binary does not change, and thus there is no change to Inst/ Program.	Decrease: Microcoded machines take several clock cycles to execute an instruction, while the RISC pipeline should have a CPI near 1 (thanks to pipelining).	No effect: the amount of work done in one pipeline stage and one microcode cycle are about the same. Increase: the RISC pipeline introduces longer control paths and adds bypasses, which are likely to be on the critical path.	Increase: it should be far easier to pipeline RISC uops once the CISC instructions Have been decoded/translated, leading to a higher performance machine (see modern x86 machines).

Lab 2

- Released this morning
- Can use git now (to manage bug fixes I make)
  - Optional (cp still works just fine)
- Due 3 weeks from yesterday (3/2)

Lab 2 Content

- Caches!
- Use cache simulator and a set of benchmarks to understand
  - How well caches work and
  - How parameters affect performance
  - Size/access time tradeoffs
  - AMAT

Directed Portion

- How to collect stats?
- Determine working-set size
- Find best L1 Instruction Cache
- Find best L1 Data Cache
- Find best L1 Data Cache with an L2

Open Ended

- Design best L1s+L2 given an limited area budget
- Design a victim cache
  - Place to put recently evicted lines
- Design a prefetcher
- Design a replacement policy

Time to Complete

- If you don't listen to me you will complain at the end
- Simulations take a long time
- Asking you to do a lot!!!
- Don't manually run them all!
- There is already a script to do it
  - Just need to modify it

But what about my internet?

- Several ways to keep tasks running after you logout (or get dropped)
- Most popular – screen, tmux
- I use screen so I'll demo that

```
$ssh cs152-ta@derby.cs.berkeley.edu
$screen -S lab2
$./explore.py
$^Ad #Ctrl-A+d Disconnect
$screen -r lab2
$...explore.py still running...
```

Questions