

14. Pipelining: Control Hazards, Performance Evaluation

EECS 370 – Introduction to Computer Organization – Winter 2009

Prof. Todd Austin & Prof. Marios Papaefthymiou

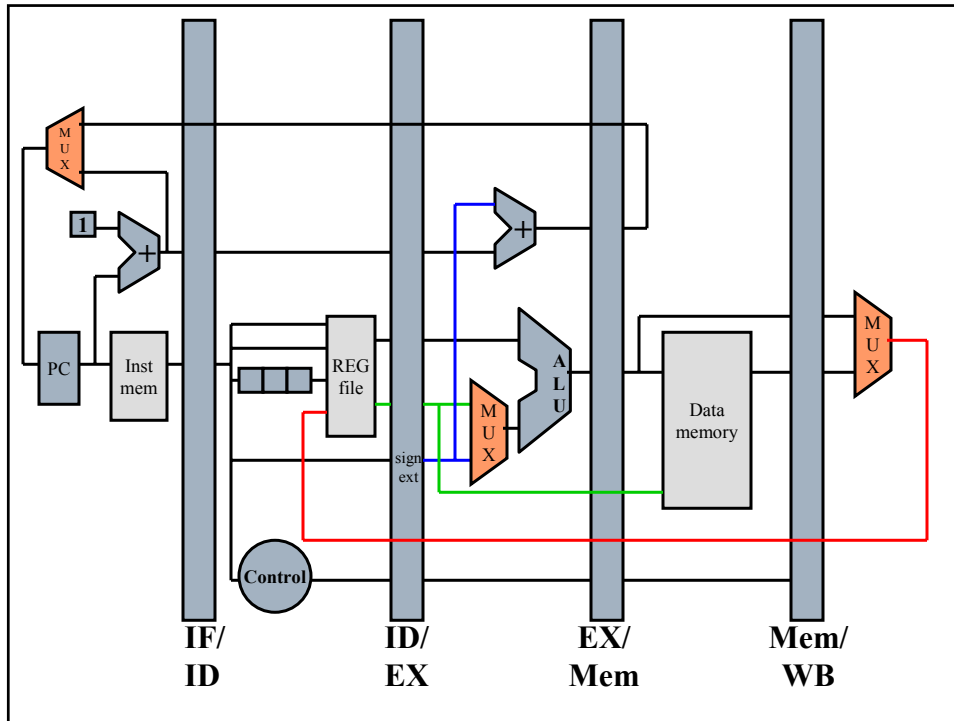
EECS Department
University of Michigan, Ann Arbor

© Austin & Papaefthymiou, 2009

The material in this presentation cannot be
copied in any form without our written permission

Announcements

- ❑ Exam regrade requests due this Thu, Mar 12
- ❑ Project 3 due Fri, Mar 27
 - Time to get started if you haven't done so already
 - Simulation of pipelined LC2K9
 - Test cases are required for the project
 - Good test cases critical for success
 - Very hard to debug without a thorough test suite
 - Get started on your test cases right away
 - After today's class, you have everything you need to know

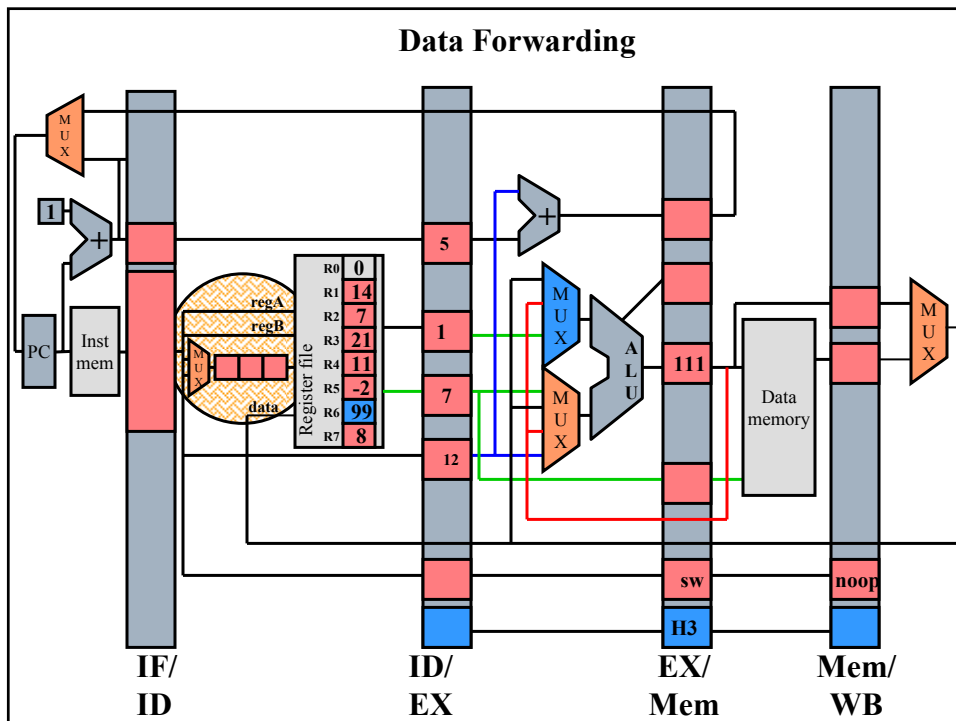


What can go wrong?

- ❑ **Data hazards:** Since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read the wrong value if it is about to be written.
- ❑ **Control hazards:** A branch instruction may change the PC, but not until stage 4. What do we fetch before that?
- ❑ **Exceptions:** How do you handle exceptions in a pipelined processor with 5 instructions in flight?

Approaches to handling data hazards

- ❑ Avoid
 - Make sure there are no hazards in the code.
- ❑ Detect and Stall
 - If hazards exist, stall the processor until they go away.
- ❑ Detect and Forward
 - If hazards exist, fix up the pipeline to get the correct value (if possible).



Example Time Graph

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nand 3 4 5		IF	ID	EX	ME	WB							
add 6 3 7			IF	ID	EX	ME	WB						
lw 3 6 10				IF	ID	EX	ME	WB					
sw 6 2 12					IF	no op	ID	EX	ME	WB			

Data Forwarding – Lecture vs Project 3

- ❑ Some questions you may have
 - What is the WBEND pipeline register in the project for?
 - Why are 3 noops required to avoid hazards in the project?
 - But only 2 noops in class?
- ❑ Answer
 - The “magic” register file
 - Lecture register file assumes internal forwarding – In a single cycle, values written to a register are immediately reflected to any reads that occur in the same cycle
 - Project register file does not do internal forwarding
 - Most modern processors have internal forwarding as it’s cheaper than having an additional pipeline register

Today: Control hazards

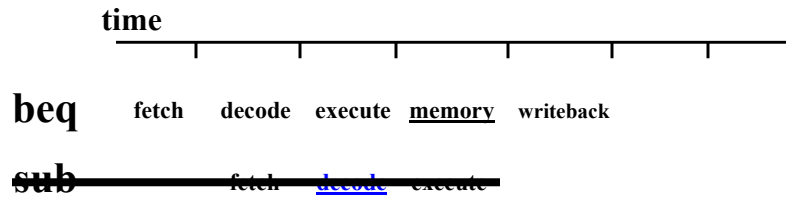
- ❑ How can the pipeline handle branch and jump instructions?

Pipeline function for BEQ

- ❑ Fetch: read instruction from memory
- ❑ Decode: read source operands from registers
- ❑ Execute: calculate target address and test for equality
- ❑ Memory: [Send target to PC](#) if test is equal
- ❑ Writeback: Nothing left to do

Control Hazards

beq 1 1 10
sub 3 4 5



Approaches to handling control hazards

- ❑ Avoid
 - Make sure there are no hazards in the code
- ❑ Detect and Stall
 - Delay fetch until branch resolved.
- ❑ Speculate and Squash-if-Wrong
 - Go ahead and fetch more instructions in case it is correct, but stop them if they shouldn't have been executed

Handling control hazards I: Avoid all hazards

- ❑ Don't have branch instructions!
 - Maybe a little impractical ☺

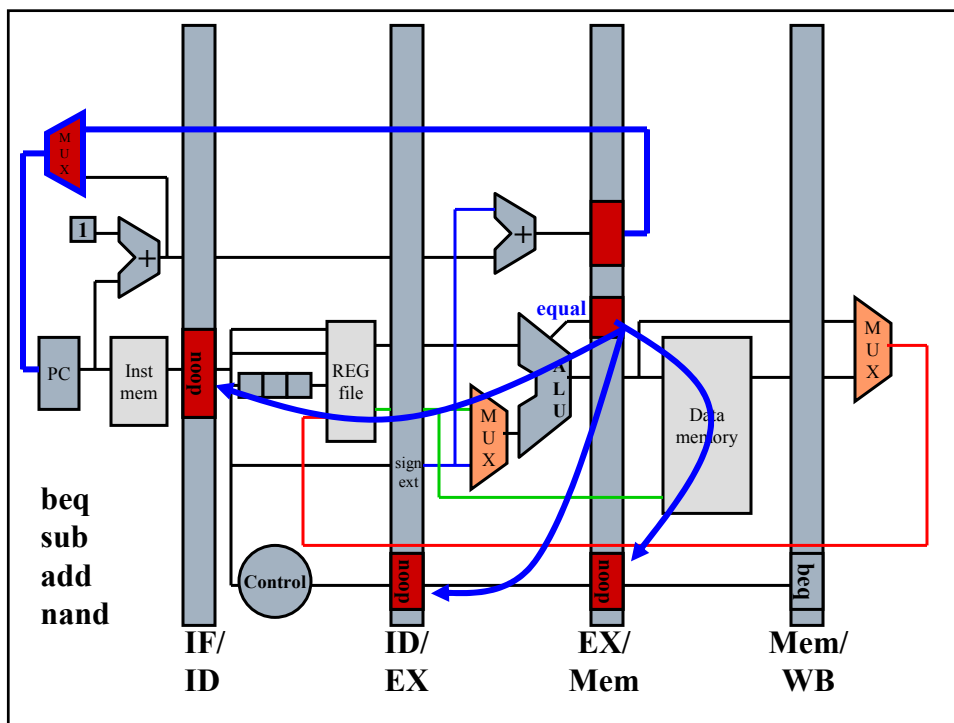
- ❑ Delay taking branch:
 - `dbeq r1 r2 offset`
 - Instructions at `PC+1`, `PC+2`, ..., `PC + <# delay slots>` will execute before deciding whether to fetch from `PC+1+offset`
 - If no useful instructions can be placed after `dbeq`, noops must be inserted.

Problems with this solution

- ❑ Old programs (legacy code) may not run correctly on new implementations
 - Longer pipelines need more instructions/noops after delayed `beq`
- ❑ Programs get larger as noops are included
 - Especially a problem for machines that try to execute more than one instruction every cycle
 - Intel EPIC: Often 25% - 40% of instructions are noops
- ❑ Program execution is slower
 - **CPI** equals 1, but some instructions are noops

Handling data hazards III: Speculate and squash

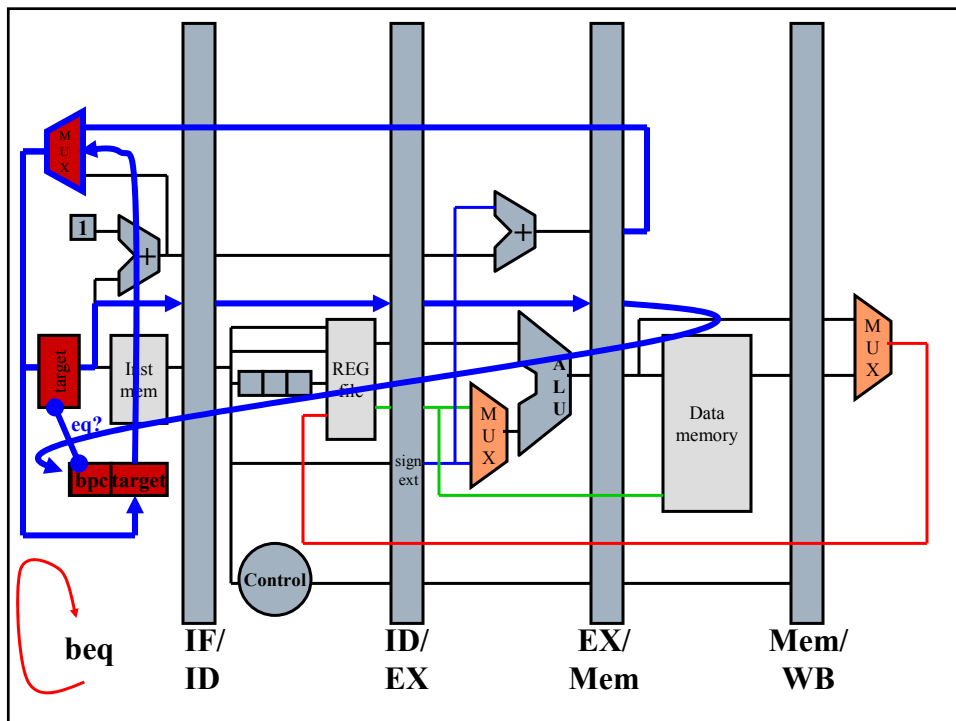
- ❑ Speculate: assume not equal
 - Keep fetching from PC+1 until we know that the branch is really taken
- ❑ Squash: stop bad instructions if taken
 - Send a noop to:
 - Decode, Execute and Memory
 - Send target address to PC



Problems with fetching PC+1

- ❑ CPI increases every time a branch is taken!
 - About 1/2 of the time
- ❑ Is that necessary?

No!, but how can you fetch from the target before you even know the previous instruction is a branch – much less whether it is taken???



Branch prediction

- ❑ Predict not taken: ~50% accurate
- ❑ Predict backward taken: ~65% accurate
- ❑ Predict same as last time: ~80% accurate

- ❑ Pentium: ~85% accurate
- ❑ Pentium Pro: ~92% accurate
- ❑ Best paper designs: ~96% accurate

Performance Issues

Execution time (Time/Program) =
of instr (I/P) × CPI (C/I) × cycle time (T/C)

Multi-cycle decreases cycle time, but increases CPI

Pipelining decreases CPI
Down to 1.0 if no stalls (hazards that are fixed by stalling)

Calculating performance with no stalls

```
add 1 2 3
nand 1 4 5
add 4 6 7
```

How many cycles does this code take to execute?

What value is written to the ALU result field of the Mem/WB pipeline register at the end of cycle 5 (the first add is written to the IF/ID pipeline register at cycle time 1)?

Calculating performance with data hazards (detect and stall)

```
add 1 2 3
nand 3 4 5
add 3 5 6
```

How many data hazards are there in this code?

How many stall cycles if we use detect and stall to handle the hazards?

Calculating performance with data hazards (detect and forward)

```
add 1 2 3
nand 3 4 5
add 3 5 6
lw 3 6 7
add 6 6 1
```

Where do the values for the nand instruction come from?

How many stall cycles on the LC2K pipelined datapath with data forwarding from lecture?

Calculating performance with control hazards (speculate and squash)

- ❑ How many cycles are saved if you perform speculate and squash for the following code (assume that branches are predicted to be not taken)?
- ❑ Effects on data hazards given a correct, or incorrect branch prediction.

```
add 1 2 3
beq 1 5 1
nand 6 4 1
add 3 4 5
```