Lecture 6: Pipelining Contd.

Kunle Olukotun
Gates 302
kunle@ogun.stanford.edu

http://www-leland.stanford.edu/class/ee282h/

Unfortunately, Our pipeline is broken!

- Branches and jumps don't work: the PC gets updated 3 cycles too late
- An instruction using the result of closely preceeding instruction(s) won't get the right data.
- The general solution to these and other problems: the pipeline must stall occasionally. Stalls occur when:
  - There are unexpected delays (eg: slow memory due to cache miss)
  - A pipeline section must wait for previous instructions. There are three kinds of these hazards:
    - Data hazards: wait for operands
    - Control hazards: wait for the right instruction after a branch
    - Structural hazards: wait for a hardware resource which is in use for another instruction.

Structural Hazards

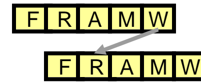
- When two instructions need to use the same hardware resource in the same cycle
  - » resource is not duplicated
    - – register file write ports
  - » resource is not fully pipelined, i.e. takes more than one cycle
    - – division, floating point
- Fix #1: Stall
  - » Low cost, but increases average CPI
  - » Best used for rare events
  - » Examples:
    - – MIPS R2000 multicycle multiply
    - – SPARC V1 single memory port for instructions and data
- Fix #2: Duplicate the resource
  - » Increases cost, but preserves CPI
  - » Best used for cheap resources and/or frequent events

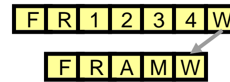
Structural Hazards, continued

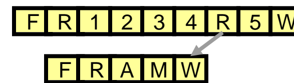
- » Example resource duplication:
  - – Separate instruction and data memory
  - – Separate ALU and PC adders
  - – Register files with multiple ports
- Fix #3: Pipeline an expensive resource
  - » Moderate cost compared to duplication, expensive compared to stalling
  - » Best used for high-performance or specialty machines
    - – Fully pipelined floating point units for scientific applications
- How to avoid structural hazards altogether: Design the ISA so that each resource needed by an instruction
  - » is used once
  - » is always used in the same pipeline stage
  - » takes one cycle

Types of Data Hazards

- RAW (read after write)
  - » only hazard for 'fixed' pipelines
  - » later instruction must *read* after earlier instruction *writes*
- WAW (write after write)
  - » variable-length pipeline
  - » later instruction must *write* after earlier instruction *writes*
- WAR (write after read)
  - » pipelines with late read
  - » later instruction must *write* after earlier instruction *reads*





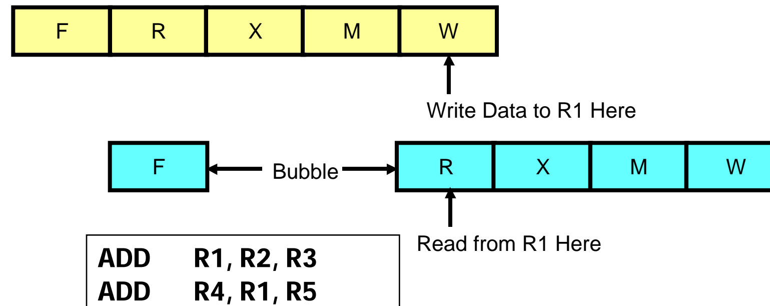


Example RAW pipeline hazard

Instruction	1	2	3	4	5	6	7	8	9
	IF	ID	EX	MEM	WB				
		IF	ID	EX	MEM	WB			
			IF	ID	EX	MEM	WB		
				IF	ID	EX	MEM	WB	
					IF	ID	EX	MEM	WB

Stall for RAW hazards

- Relatively cheap: just needs some extra compare and control logic
  - Detect in the ID stage by comparing the registers to be read with the register to be written for the instruction currently in the EX, MEM, or WB stages.
  - Stall if a match is found
- Increases average CPI
- Would happen much too frequently



Stall type #1: Freeze the whole pipeline

	1	2	3	4	*5*	**6**	7	8	9	10	11
i	IF	ID	EX	MEM	*WB*	**WB**					
i+1		IF	ID	EX	*MEM*	**MEM**	WB				
i+2			IF	ID	*EX*	**EX**	MEM	WB			
i+3				IF	*ID*	**ID**	EX	MEM	WB		
i+4					*IF*	**IF**	ID	EX	MEM	WB	
I+5							IF	ID	EX	MEM	WB
I+6								IF	ID	EX	MEM

- Freeze all pipe stages for one or more cycles, and supress write-back
- Needs only one global stall signal which supresses latching in all pipeline registers
- Sometimes called a “fixed pipe” or “frozen pipe” stall

- Works for cache misses
- WIll not work to remove pipeline hazards

Stall type #2:
Delay completion of an instruction

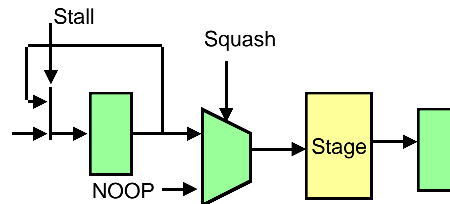
	1	2	3	4	5	6	7	8	9	10	11
i	IF	ID	EX	MEM	WB						
i+1		IF	ID	EX	MEM	WB					
i+2			IF	ID	**stall**	EX	MEM	WB			
i+3				IF	**stall**	ID	EX	MEM	WB		
i+4					**stall**	IF	ID	EX	MEM	WB	
I+5							IF	ID	EX	MEM	WB
I+6								IF	ID	EX	MEM

"Bubble" in: EX MEM WB

- Instruction progress stops for one cycle
- Earlier instructions continue towards completion
- Prior instructions must suspend and make no more progress
- An "elastic pipe" stall.
- Good when the need for stalling is only detected after decode, like for pipeline hazards

Implementing Stalls

- Detect the stall condition
- Freeze stalled instructions in place
  - » recycle pipeline registers
- Insert NOOP in pipeline causes bubble



Bypass (Forwarding)

- If data is available elsewhere in the pipeline, there is no need to stall
- Detect condition
- Bypass (or forward) data directly to the consuming pipeline stage
- Bypass eliminates stalls for *single-cycle* operations
  - » reduces longest stall to N-1 cycles for N-cycle operations

Fowarding data

Instruction	1	2	3	4	5	6	7	8	9
add r1, r2, r3	IF	ID	EX	MEM	WB				
sub r5,r1,r3		IF	ID	EX	MEM	WB			
or r6,r5,r1			IF	ID	EX	MEM	WB		
and r7,r8,r1				IF	ID	EX	MEM	WB	
xor r8, r1, r9					IF	ID	EX	MEM	WB

-- The third forwarding operation might not be necessary if we can make a clever read-after-write register file

Example fowarding decisions

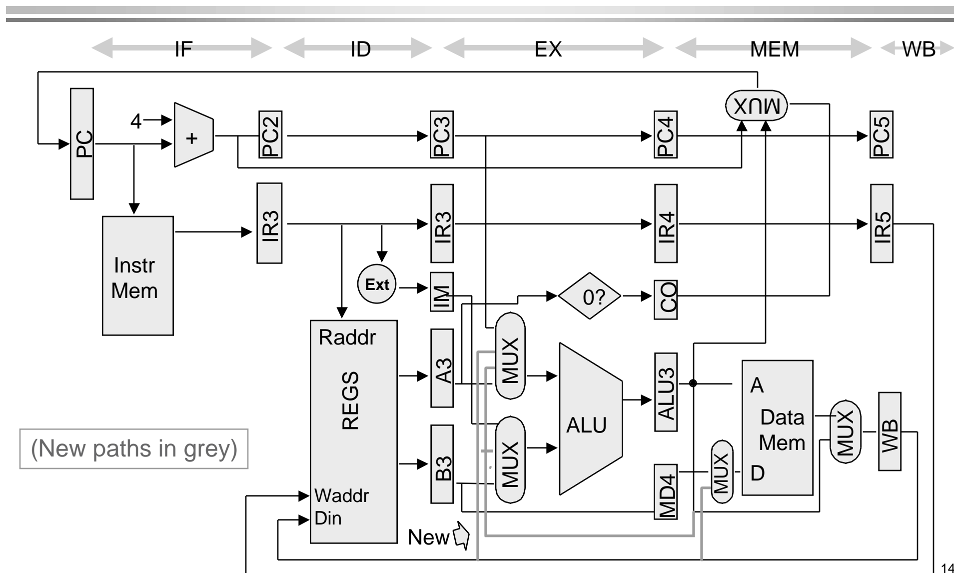
- If EX has just finished an operation for which ID wants to read the value from either operand, we must forward:

```
if  IR4.WillWriteReg
    and IR4.WriteRegNum == IR3.RS1RegNum
then ALUmuxA = SelectALU3

if  IR4.WillWriteReg
    and IR4.WriteRegNum == IR3.RS2RegNum
then ALUmuxB = SelectALU3
```

- Need one comparison and multiplexer control for each fowarding path
- Be careful: if you can forward from more than one instruction, choose the one closest in the pipeline

The pipelined diagram with fowarding paths



Other Data Hazards

- WAR (Write-after-Read)
  - Can happen if the instruction pipeline has early writes and/or late reads; something like:

DIV (R1), ...	suppose that it doesn't read destination indirect until after the divide
ADD ..., (R1)+	Incremented value of R1 is written before DIV has read the value of R1

  - Can't happen in DLX because all reads are early (ID) and all write are late (WB)
- WAW (Write-after-Write)
  - Can happen when a fast operation follows a slow one; like

LW R1, 0(R2)	IF	ID	EX	MEM1	MEM2	WB
ADD R, R2, R3		IF	ID	EX	WB	

  - Can't happen in regular DLX (integer) because there is only one WB stage and instructions use it in order

One data hazard left

Instruction	1	2	3	4	5	6	7	8	9
	IF	ID	EX	MEM	WB				
		IF	ID	EX	MEM	WB			
			IF	ID	EX	MEM	WB		
				IF	ID	EX	MEM	WB	
					IF	ID	EX	MEM	WB

- Loaded data is not available until the end of MEM, which is too late for the following instruction

- Fowarding can't help , so we must stall
  -- or just "decree" that you can't write code like this. Such a decree is called a "delayed load" and was used in the original MIPS 2000.

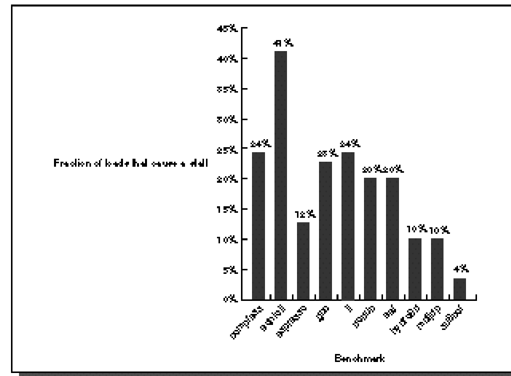
Stalling to interlock after a load

Instruction	1	2	3	4	5	6	7	8	9	
	IF	ID	EX	MEM	WB					
		IF	ID	EX	MEM	WB				
			IF	ID	EX	MEM	WB			
				IF	ID	EX	MEM	WB		
					IF	ID	EX	MEM	WB	
						IF	ID	EX	MEM	WB

The software fix: Instruction scheduling to avoid stalls

- Since we can't avoid a stall following a load, avoid the stall by rearranging code ("pipeline scheduling"), if possible:
  - » Replace
    ```
    sub   r4, r5, r7
    lw    r1, 50(r2)
    add   r3, r1, r4
    ```
  - » with
    ```
    lw    r1, 50(r2)
    sub   r4, r5, r7
    add   r3, r1, r4
    ```
- This can improve a simple RISC machine performance!
- But it's limited:
  - » Usually limited to basic blocks between branches, 5-7 instructions
  - » Difficult to do interchanges to variables referenced indirectly (pointer, array, or parameter) due to the risk of aliases.

The effect of load scheduling



Fix branches and jumps

- Easy fix: stall after decoding a branch until it finishes MEM and knows where to go:

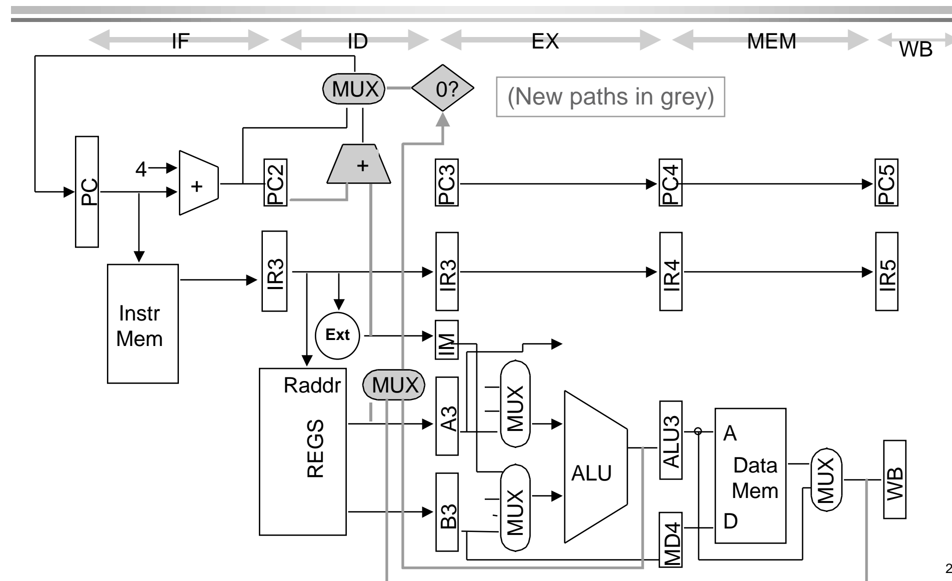
	1	2	3	4	5	6	7	8	9	10	11
Branch i	IF	ID	EX	MEM	WB						
i+1		IF	***<u>Stall</u>***	***<u>Stall</u>***	***<u>IF</u>***	ID	EX	MEM	WB		
i+2						IF	ID	EX	MEM	WB	
i+3							IF	ID	EX	MEM	WB
i+4								IF	ID	EX	MEM

- This 3-cycle branch delay works, but since branches occur every 5-7 instructions, it kills performance. What to do?
  - Determine the branch condition earlier than EX
  - Compute the target address earlier than MEM

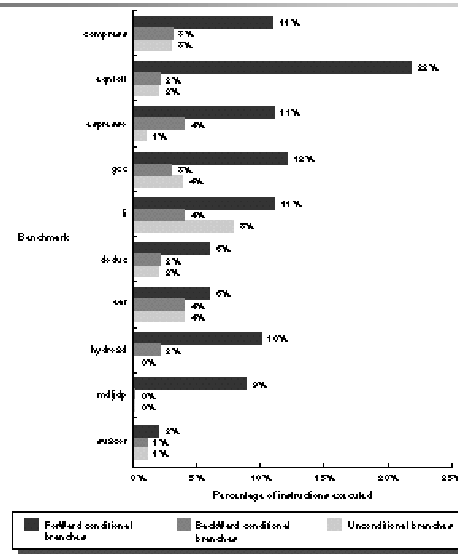
Characteristics of DLX branches and jumps

- The branch condition
  - » Only has EQ/NE comparison to zero
  - » Fast and cheap; no need for a full ALU
  - » Use a 32-bit NOR gate instead
- The branch target
  - » Always PC-relative
  - » Needs only 16-bit adder (and carry propagation)
- The jump target
  - » Always PC-relative
  - » Needs a 26-bit adder (and carry propagation)
  - » In the real MIPS R2000 CPU
    - target = {PC[31..28],offset,'b00}
    - Required no adder at all!
- All can be moved to the ID stage, at the cost of additional hardware (and maybe increased cycle time)
- Still requires one stall cycle

Pipeline additions for early branches (in ID)



Control flow
Instruction Statistics



Integer
- 13% FB, 3%BB, 4% UB
- 70% of branches and jumps change control flow

Floating point
- 7% FB, 2%BB, 1% UB
- 73% of branches and jumps change control flow

All
- 85% of backward branches branch
- 60% of foward branches branch

Reducing branch penalties 1

- Predict the branch won't be taken
  - » Easy to do:
    - If it isn't, continue
    - If it is, change the following instruction into a NOP and thus take a 1-cycle penalty
  - » Help a little, but bets the wrong way for loops

	1	2	3	4	5	6	7	8	9	10	11
Branch i	IF	ID	EX	MEM	WB						
i+1											
i+2											
i+3											
i+4											

Reducing branch penalties 2

- Predict the branch will be taken
  - Only useful if the target address is known before the branch condition -- not true for DLX
  - Always has some delay for fetching the branch target

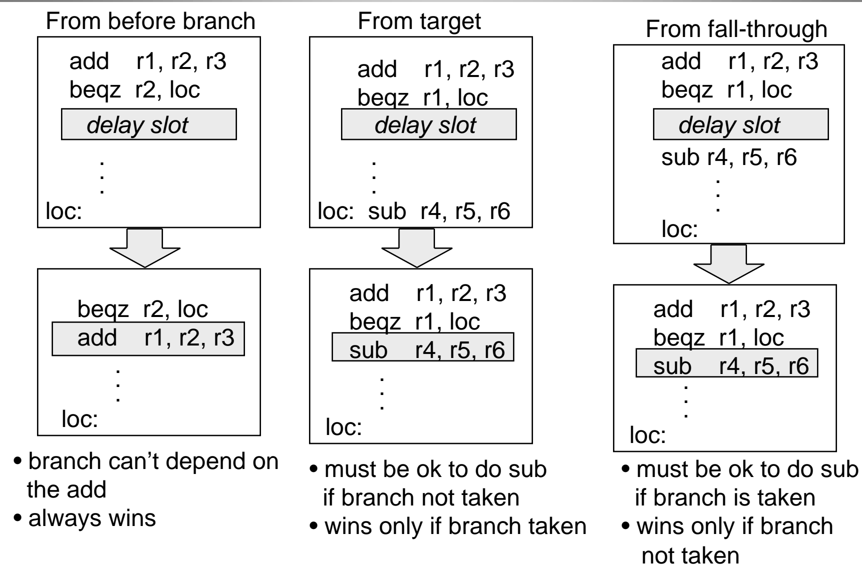
	1	2	3	4	5	6	7	8	9	10	11
Branch i	IF	ID	EX	MEM	WB						
i+1											
i+2											
i+3											
i+4											

Reducing branch penalties 3

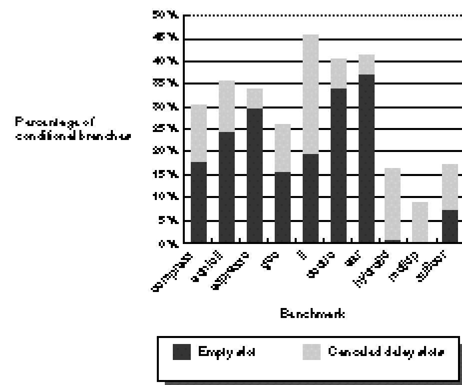
- Change the ISA: delay the effect of the branch
  - Always execute the instruction(s) after the branch or jump
  - Depend on the compiler to find something useful to do in the *branch delay slot(s).*
  - An ugly dependence of ISA on implementation -- may change.

	1	2	3	4	5	6	7	8	9	10	11
Branch i	IF	ID	EX	MEM	WB						
i+1											
i+2											
i+3											
i+4											

Filling the branch delay slot



How useful are canceling branches



Integer
• 35% slots wasted

Floating point
• 25% slots wasted

Performance of Branch Schemes?

Effective CPI = 1 + %branches * avg branch penalty

For integer DLX: 20% of instructions are branches or jumps
70% of them go to the target

Strategy	Branch-taken penalty	Branch-not-taken penalty	Effective CPI
Stall	3	3	1.6
Branch in ID	1	1	1.20
Predict taken	1	1	1.20
Predict not taken	1	0	1.14
Delay slot	0.5	0.5	1.10
Cancel branch	0.3	0.3	1.06

Pipeline Example

- Consider the following pipeline which implements the DLX-like ISA. The only variation on the DLX ISA is the support of full register compares in branch instructions.

	1	2	3	4	5	6	7	8	9	10	11
i	IF	ID	EX1	EX2/ Mem1	Mem2	WB					
i + 1		IF	ID	EX1	EX2/ Mem1	Mem2	WB				
i + 2			IF	ID	EX1	EX2/ Mem1	Mem2	WB			
i + 3				IF	ID	EX1	EX2/ Mem1	Mem2	WB		
i + 4					IF	ID	EX1	EX2/ Mem1	Mem2	WB	
i + 5						IF	ID	EX1	EX2/ Mem1	Mem2	WB

The Pipeline Stages

Stage	Function
IF	instruction fetch
ID	instruction decode register fetch
EX1	address generation (data and PC-target)
EX2/MEM1	ALU operation branch condition resolution first cycle of memory access
MEM2	second cycle of memory access
WB	register file writeback

Assumptions

- Writes to the register file occur in the first half of the clock cycle while reads from the register file occur in the second half cycle.
- All bypass paths have been implemented to minimize pipeline stalls due to data hazards.
- The pipeline implements hardware interlocks.

Questions

- How many register file ports does the processor need to minimize structural hazards?
- Indicate all forwarding required to minimize stalls in the given pipeline. Also, specify the minimum number of comparators needed to implement the forwarding.
- What is the worst case delay due to RAW data hazards?
- What is the branch delay of this pipeline?

Instruction Dependencies

- The frequencies in these tables are presented as percentages of all instructions executed.

Type	Instruction Sequence	Frequency
1	ALU op Rx, -, - ALU op -, -, Rx or -, Rx, -	10%
2	ALU op Rx, -, - Store Rx, -(-)	5%
3	ALU op Rx, -, - Load/Store -, -(Rx)	5%
4	ALU op Rx, -, - JumpRegister Rx	1%
5	ALU op Rx, -, - Branch Rx, -, # or -, Rx, #	2%
6	Load Rx, -(-) ALU op -, -, Rx or -, Rx, -	15%
7	Load Rx, -(-) Load/Store -, -(Rx)	3%
8	Load Rx, -(-) Branch Rx, -, # or -, Rx, #	2%
9	Load Rx, -(-) JumpRegister Rx	1%

More Questions

- List the instruction sequences from the above table that cause data stalls in the pipeline. Indicate the corresponding number of stall cycles.
- Compute the CPI for the pipeline due to data hazards only. Ignore instruction sequences that are not listed in the above table.
- If the frequency of conditional branches is 10%, of which 65% are taken and the frequency of unconditional branches is 6%, compute the overall CPI assuming a TAKEN branch prediction scheme.