Basic Pipelining

Control Hazards (Branches & Exceptions)

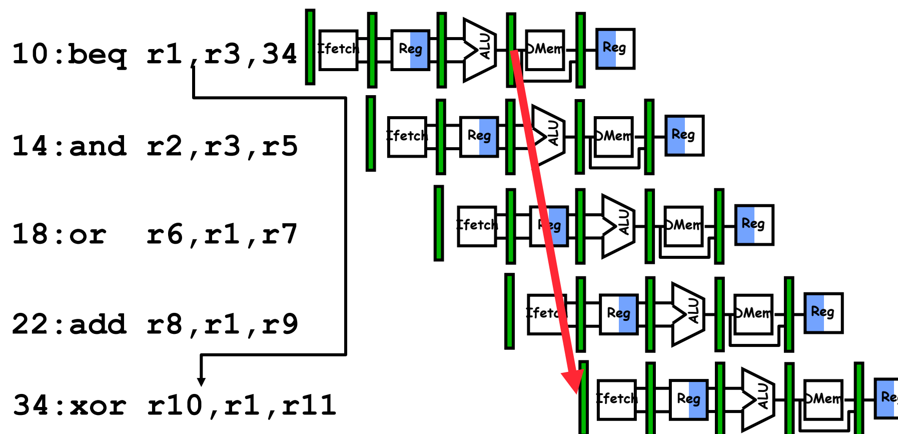
Control hazards

- **Question: When do we find out that the PC needs to be modified?**
  - **Answer: In pipeline stage ID of a branch instruction**
  - **So, if a branch is not-taken (i.e., if the PC is not modified), need a one-cycle delay**
- **Question: When is a taken branch's address known?**
  - **ALU used to compute, so EX stage**
  - **Need two (or three) cycle delay**

Example

- If branch in 30% of instructions, then instead of executing 1 instruction per cycle,
  - have 70% of instructions executing in 1 cycle and 30% of instructions executing in 2 cycles
- An average of .7 + .6 = 1.3 cycles per instruction
  - Worse by 30%

Control Hazard on Branches Three Stage Stall



What do you do with the 3 instructions in between?

How do you do it?

Where is the "commit"?

Pipeline Depth and Issue Width

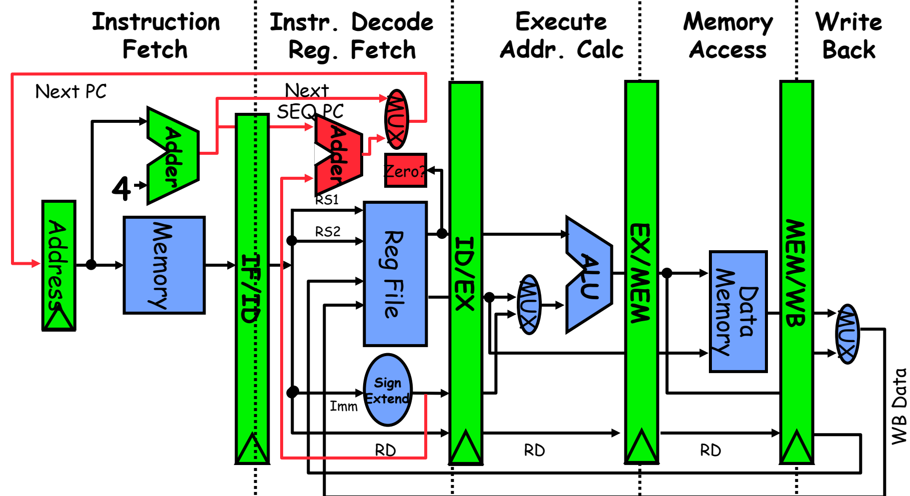
- **Intel Processors over Time**

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Cores	Power
i486	1989	25 MHz	5	1	1	5W
Pentium	1993	66 MHz	5	2	1	10W
Pentium Pro	1997	200 MHz	10	3	1	29W
P4 Willamette	2001	2000 MHz	22	3	1	75W
P4 Prescott	2004	3600 MHz	31	3	1	103W
Core 2 Conroe	2006	2930 MHz	14	4	2	75W
Core 2 Yorkfield	2008	2930 MHz	16	4	4	95W
Core i7 Gulftown	2010	3460 MHz	16	4	6	130W

Branch Stall Impact

- **If CPI = 1, 30% branch,**
  Stall 3 cycles => new CPI = 1.9!
- **Two part solution:**
  - **Determine branch taken or not taken sooner, AND**
  - **Compute taken branch address earlier**
- **MIPS branch tests if register = 0 or ≠ 0**
- **MIPS Solution:**
  - **Move Zero test to ID/RF stage**
  - **Adder to calculate new PC in ID/RF stage**
  - **1 clock cycle penalty for branch versus 3**

Pipelined MIPS Datapath



- Interplay of instruction set design and cycle time.

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken
- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken
- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
  - » MIPS still incurs 1 cycle branch penalty
  - » Other machines: branch target known before outcome

Four Branch Hazard Alternatives

#4: Delayed Branch

- Define branch to take place **AFTER** a following instruction

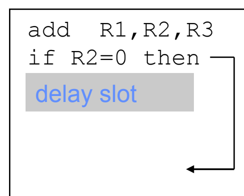
```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

Branch delay of length *n*

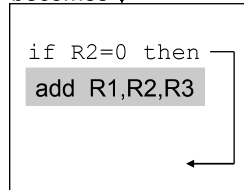
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Scheduling Branch Delay Slots

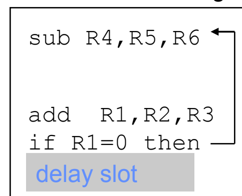
A. From before branch



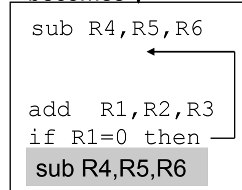
becomes



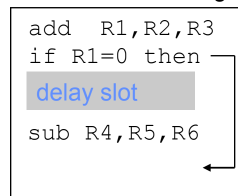
B. From branch target



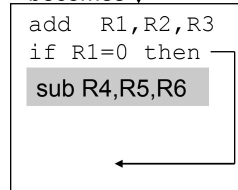
becomes



C. From fall through



becomes



- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Scheduling Branch Delay Slots

- If taken from before branch
  - Branch must not depend on rescheduled instruction
  - Always improves performance
- If taken from branch target
  - Must be OK to execute rescheduled instructions if branch not taken, and may need to duplicate instructions
  - Performance improved when branch taken
- If taken from fall through
  - Must be OK to execute instructions if branch taken
  - Improves performance when branch not taken

Delayed Branch

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside:
  - As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
- Result:
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Assume: 4% unconditional branch
6% conditional branch-not taken
10% conditional branch-taken

Scheduling Scheme	Branch Penalty	CPI	Speedup vs. Unpipelined	Speedup vs. Stall
Stall Pipeline	3	1.60	3.1	1.0
PredictTaken	1	1.20	4.2	1.33
Predict Not Taken	1	1.14	4.4	1.40
Delayed Branch	0.5	1.10	4.5	1.45

Pipelining Summary

- Pipelining can speed instruction execution (throughput)
- But need to deal with structural hazards, data hazards, and control hazards
- Next
  - How to handle exceptions?
  - How to handle long instructions, such as floating point arithmetic?

The Problem

- Question: What makes pipelining hard to implement?
- Answer: Surprises
- Technical names for surprises
  - Exceptions
  - Faults
  - Interrupts

Some Examples Of Exceptions

- Request for I/O
- Arithmetic troubles: overflow or underflow
- Cache miss: data not in (on-chip) cache memory
- Page fault: data not in (physical) memory
- Illegal address, giving a memory protection violation
- Hardware failure

Classifying Exceptions

- Synchronous: repeatable every time
  - Example: DIV R2, R2, R0

  Asynchronous: caused by external events like hardware failure and devices external to processor and memory
- User requested: user task asks for it (example: breakpoint)

  Coerced: cannot be predicted by user
- User maskable: can be disabled by user task
  - Example: arithmetic exception

  Nonmaskable: cannot be turned off
  - Example: hardware failure

Classifying Exceptions (cont.)

- Within instruction: prevents instruction from completing
  Between instructions: no instruction prevented
- Terminating: stops the task
  Resuming: task can continue
- Machines that handle exceptions, save the state, and then restart correctly are said to be **restartable**

Categorizing Exceptions

Exception type	Synch. vs. asynch.	User request vs. coerce	User maskable vs. not	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynch	Coerced	Not	Between	Resume
Invoke OS	Synch	User req.	Not	Between	Resume
Tracing instructions	Synch	User req.	Maskable	Between	Resume
Breakpoint	Synch	User req.	Maskable	Between	Resume
Integer overflow	Synch	Coerced	Maskable	Within	Resume
Floating pt. overflow/ underflow	Synch	Coerced	Maskable	Within	Resume

Categorizing Exceptions (cont.)

Exception type	Synch. vs. asynch.	User request vs. coerce	User maskable vs. not	Within vs. between instructions	Resume vs. terminate
Page fault	Synch	Coerced	Not	Within	Resume
Misaligned memory access	Synch	Coerced	Maskable	Within	Resume
Mem. prot. violation	Synch	Coerced	Not	Within	Resume
Undefined instruction	Synch	Coerced	Not	Within	Terminate
Hardware malfunction	Asynch	Coerced	Not	Within	Terminate
Power failure	Asynch	Coerced	Not	Within	Terminate

The Most Difficult Exceptions...

- ... are those that occur within EX or MEM stages and need to be handled in a restartable way
- Why difficult? Handling one includes:
  - The next IF gets a "trap instruction"
  - Until the trap is taken, turn off all "writes" for the faulting instruction and those that follow it
  - What does the trap do?
    - » The trap transfers control to the exception handling routine in the operating system, which saves the PC of the faulting instruction and handles the fault
  - The task is then resumed, using the saved PC and the MIPS instruction RFE or something like it
- Note: May need to save several PCs if delayed branches are involved

Exceptions (cont.)

- Ideally, pipeline can be interrupted so that instructions before the fault complete. Then want to restart execution just after the faulting instruction - precise exception handling
- This is the right way to do it, but sometimes architects/manufacturers take shortcuts

When Do MIPS Exceptions Occur?

- IF
  - Page fault on instruction fetch
  - Misaligned memory access
  - Memory protection violation
- ID
  - Undefined or illegal opcode
- EX
  - Arithmetic exception
- MEM
  - Page fault on data fetch/store
  - Misaligned memory access
  - Memory protection violation
- WB: None!

Examples Of Exception Handling

LD	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

- Handle the MEM fault first, then restart

LD	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

- IF fault occurs first, even though LD will fault later
- But for precise exceptions, must handle LD fault first

How Is This Done?

- Answer: Don't handle exceptions until the WB stage
  - Each instruction has an associated status vector that keeps track of faults
  - Any bit set in the status vector turns off register writes and memory writes
  - In WB stage, the status vector is checked and any fault is handled
  - So, since instructions reach WB in proper order, faults for earlier instructions are handled before faults for later instructions
    - » Unfortunately, will need to violate this later (for instructions that don't reach WB in proper order)

Commitment

- When an instruction is guaranteed to complete, it is committed
- Life is easier if no instruction changes the permanent machine state before it is committed
- In MIPS, commitment occurs at the end of the MEM stage - that's why register update occurs in the stage after that
- Some machines muddy the state before commitment, and the exception handler must do its best to restore the state that existed before the instruction started