Readings in Pipelining

H+P
• Appendix A (except for A.8)
• This will be mostly review for those who took ECE 152

Recent Research Papers
• “The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays”, Hrishikesh et al., ISCA 2002.
• “Power: A First Class Design Constraint”, Mudge, IEEE Computer, April 2001. (not directly related to pipelining)

Basic Pipelining
• basic := single, in-order issue
  • single issue := one instruction at a time (per stage)
  • in-order issue := instructions (start to) execute in order
  • next unit: multiple issue
  • unit after that: out-of-order issue
• pipelining principles
  • tradeoff: clock rate vs. IPC
  • hazards: structural, data, control
• vanilla pipeline: single-cycle operations
  • structural hazards, RAW hazards, control hazards
• dealing with multi-cycle operations
  • more structural hazards, WAW hazards, precise state

Piipelining

observe: instruction processing consists of $N$ sequential stages

idea: overlap different instructions at different stages

<table>
<thead>
<tr>
<th>non-pipelined</th>
<th>pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst0.1</td>
<td>inst0.1</td>
</tr>
<tr>
<td>inst0.2</td>
<td>inst0.2</td>
</tr>
<tr>
<td>inst0.3</td>
<td>inst0.3</td>
</tr>
</tbody>
</table>

• increase resource utilization: fewer stages sitting idle
• increase completion rate (throughput): up to 1 in 1/$N$ time
• almost every processor built since 1970 is pipelined
  • first pipelined processor: IBM Stretch [1962]

Without Pipelining

• 5 parts of instruction execution
  • fetch (F, IF): fetch instruction from I$
  • decode (D, ID): decode instruction, read input registers
  • execute (X, EX): ALU, load/store address, branch outcome
  • memory access (M, MEM): load/store to DS/DTLB
  • writeback (W, WB): write results (from ALU or Id) back to register file
Simple 5-Stage Pipeline

- 5 stages (pipeline depth is 5)
  - fetch (F, IF): fetch instruction from I$
  - decode (D, ID): decode instruction, read input registers
  - execute (X, EX): ALU, load/store address, branch outcome
  - memory access (M, MEM): load/store to D$/DTLB
  - writeback (W, WB): write results (from ALU or Id) back to register file
- stages divided by pipeline registers/latches

Pipeline Registers (Latches)
- contain info for controlling flow of instructions through pipe
  - PC: PC
  - F/D: PC, undecoded instruction
  - D/X: PC, opcode, regfile[rs1], regfile[rs2], immed, rd
  - X/M: opcode (why?), regfile[rs1], ALUOUT, rd
  - M/W: ALUOUT, MEMOUT, rd

Pipeline Diagram

<table>
<thead>
<tr>
<th>inst0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst3</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compared to non-pipelined case:
- Better throughput: an instruction finishes every cycle
- Same latency per instruction: each still takes 5 cycles

Principles of Pipelining

let: instruction execution require \( N \) stages, each takes \( t_n \) time

- un-pipelined processor
  - single-instruction latency \( T = \Sigma t_n \)
  - throughput = \( 1/T = 1/\Sigma t_n \)
  - \( M \)-instruction latency = \( M*T \) (\( M \gg 1 \))

- now: \( N \)-stage pipeline
  - single-instruction latency \( T = \Sigma t_n \) (same as unpipelined)
  - throughput = \( 1/ max(t_n) < N/T \) \( (max(t_n) \) is the bottleneck)
    - if all \( t_n \) are equal (i.e., \( max(t_n) = T/N \), then throughput = \( N/T \)
  - \( M \)-instruction latency (\( M \gg 1 \)) = \( M * max(t_n) \leq M*T/N \)
    - speedup \( \leq N \)
    - can we choose \( N \) to get arbitrary speedup?
### Wrong (part I): Pipeline Overhead

**V :=** overhead delay per pipe stage
- cause #1: latch overhead
- pipeline registers take time
- cause #2: clock/data skew

so, for an N-stage pipeline with overheads
- single-instruction latency \( T = \sum (V + t_n) = NV + \sum t_n \)
- throughput = \( 1/(\max(t_n) + V) \leq N/T \) (and \( \leq 1/V \))
- \( M \)-instruction latency = \( M^*(\max(t_n) + V) \leq M^*V + M^*T/N \)
- speedup = \( T/(V+\max(t_n)) \leq N \)

Overhead limits throughput, speedup & useful pipeline depth

### Clock Rate vs. IPC Example

- **G:** gate-delays to process an instruction
- **V:** gate-delays of overhead per stage
- **S:** average cycle stall per instruction per pipe stage
  - overly simplistic model for stalls
- compute optimal \( N \) (depth) given \( G, V, S \) [Smith+Pleszkun]
  - \( IPC = 1/(1 + S^*N) \)
  - clock rate (in gate-delays) = \( 1/(gate\,delays/stage) = 1/(G/N + O) \)
  - \( e.g.,\ G = 80, S = 0.16, V = 1 \)

<table>
<thead>
<tr>
<th>( N )</th>
<th>IPC := 1/(1+0.16*N)</th>
<th>clock := 1/(80/N+1)</th>
<th>execution rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.38</td>
<td>0.11</td>
<td>0.042</td>
</tr>
<tr>
<td>20</td>
<td>0.24</td>
<td>0.20</td>
<td>0.048</td>
</tr>
<tr>
<td>30</td>
<td>0.17</td>
<td>0.27</td>
<td>0.046</td>
</tr>
</tbody>
</table>

### Wrong (part II): Hazards

**hazards:** conditions that lead to incorrect behavior if not fixed
- **structural:** two instructions use same h/w in same cycle
- **data:** two instructions use same data (register/memory)
- **control:** one instruction affects which instruction is next

- hazards ⇒ stalls (sometimes)
  - stall: instruction stays in same stage for more than one cycle
- what if average stall per instruction = \( S \) stages?
  - latency' \( \Rightarrow \) \( T(N+S)/N = ((N+S)/N)^*latency > latency \)
  - throughput' \( \Rightarrow \) \( N^2/T(N+S) = (N/(N+S))^*throughput < throughput \)
  - \( M \)-latency' \( \Rightarrow \) \( M^*T(N+S)/N^2 = ((N+S)/N)^*M_{latency} > M_{latency} \)
  - speedup' \( \Rightarrow \) \( N^2/(N+S) = (N/(N+S))^*speedup < speedup \)

### Pipelining: Clock Rate vs. IPC

deeper pipeline (more stages, larger \( N \))
- increases clock rate
  - decreases IPC (longer stalls for hazards - will see later)
- ultimate metric is **execution rate** = clock rate*IPC
  - (clock cycle / unit real time) * (instructions / clock cycle)
  - number of instructions is fixed, for purposes of this discussion
- how does pipeline overhead factor in?

to think about this, **parameterize the clock cycle**
- basic time unit is the gate-delay (time to go through a gate)
  - \( e.g.,\ 80\ gate\,-delays\,to\,process\,(fetch,\,decode,...)\,an\,instruction\)
  - let’s look at an example ...
Pipeline Depth Upshot

trend is for *deeper pipelines* (more stages)

- why? faster clock (higher frequency)
  - clock period = \( f(\text{transistor latency, gate delays per pipe stage}) \)
  - superpipelining: add more stages to reduce gate-delays/pipe-stage
    - but increased frequency may not mean increased performance...
    - who cares? we can sell frequency!
- e.g., Intel IA-32 pipelines
  - 486: 5 stages (50+ gate-delays per clock period)
  - Pentium: 7 stages
  - Pentium II/III: 12 stages
  - Pentium 4: 22 stages (10 gate-delays per clock)
- Gotcha! 800MHz Pentium III performs better than 1GHz Pentium 4

Managing the Pipeline

to resolve hazards, need fine pipe-stage control

- play with pipeline registers to control pipe flow
- trick #1: *the stall (or the bubble)*
  - effect: stops *SOME* instructions in current pipe-stages
  - use: make younger instructions wait for older ones to complete
  - implementation: de-assert write-enable signals to pipeline registers
- trick #2: *the flush*
  - effect: clears instructions out of current pipe-stages
  - use: undoes speculative work that was incorrect (see later)
  - implementation: assert clear signals on pipeline registers
- stalls & flushes must be propagated upstream (why?)
  - upstream: towards fetch (downstream = towards writeback)

Structural Hazards

two different instructions need same h/w resource in same cycle

- e.g., loads/stores use the same cache port as fetch
  - assume unified L1 cache (for this example)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst2</td>
<td>F</td>
<td>D</td>
<td>x</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst3</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fixing Structural Hazards

- fix structural hazard by stalling \( (s^* = \text{structural stall}) \)
  - low cost, simple
  - decreases IPC
  - used rarely

- Q: which one to stall, inst4 or load?
  - always safe to stall younger instruction (why?)...
  - ...but may not be the best thing to do performance-wise (why?)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst3</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst4</td>
<td>s*</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Avoiding Structural Hazards

• option #1: replicate the contended resource
  + good performance
  – increased area, slower (interconnect delay)?
  • use for cheap, divisible, or highly-contended resources (e.g., I$/D$)
• option #2: pipeline the contended resource
  + good performance, low area
  – sometimes complex (e.g., RAM)
  • useful for multicycle resources
• option #3: design ISA/pipeline to reduce structural hazards
  • key 1: each instruction uses a given resource at most once
  • key 2: each instruction uses a given resource in same pipeline stage
  • key 3: each instruction uses a given resource for one cycle
  • this is why we force ALU operations to go thru MEM stage

Data Hazards

two different instructions use the same storage location

• we must preserve the illusion of sequential execution

add R1, R2, R3  sub R2, R4, R1  or R1, R6, R3
read-after-write (RAW)

sub R2, R4, R1  add R1, R2, R3  or R1, R6, R3
write-after-read (WAR)

or R1, R6, R3  sub R2, R4, R1  or R1, R6, R3
write-after-write (WAW)

true dependence (real)

anti-dependence (artificial)

output dependence (artificial)

Q: What about read-after-read dependences? (RAR)

RAW

read-after-write (RAW) = true dependence (dataflow)

add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3

• problem: sub reads R1 before add has written it
  • Pipelining enables this overlapping to occur
  • But this violates sequential execution semantics!
  • Recall: user just sees ISA and expects sequential execution

RAW: Detect and Stall

detect RAW and stall instruction at ID before it reads registers

• mechanics? disable PC, F/D write
• RAW detection? compare register names
  • notation: rs1(D) := source register #1 of instruction in D stage
  • compare rs1(D) and rs2(D) with rd(D/X), rd(X/M), rd(M/W)
  • stall (disable PC + F/D, clear D/X) on any match
• RAW detection? register busy-bits
  • set for rd(D/X) when instruction passes ID
  • clear for rd(M/W)
  • stall if rs1(D) or rs2(D) are “busy”

+ low cost, simple
  – low performance (many stalls)
Two Stall Timings

- depend on how ID and WB stages share the register file
  - each gets register file for half a cycle
  - 1st half ID reads, 2nd half WB writes \(\Rightarrow\) 3 cycle stall

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{add R1,R2,R3}</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{sub R2,R4,R1}</td>
<td>F</td>
<td>d*</td>
<td>d*</td>
<td>d*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>\text{load R5,R6,R7}</td>
<td>F</td>
<td>p*</td>
<td>p*</td>
<td>p*</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
</tr>
</tbody>
</table>

- 1st half WB writes, 2nd half ID reads \(\Rightarrow\) 2 cycle stall

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{add R1,R2,R3}</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{sub R2,R4,R1}</td>
<td>F</td>
<td>d*</td>
<td>d*</td>
<td>d*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

Reducing RAW Stalls: Bypassing

- why wait until WB stage? data available at end of EX/MEM stage
- bypass (aka "forward") data directly to input of EX
- very effective at reducing/avoiding stalls
  - in practice, a large fraction of input operands are bypassed (why?)
    - complex
  - does not relieve you from having to perform WB

Implementing Bypassing

- first, detect bypass opportunity
  - tag compares in D/X latch
    - similar to but separate from stall logic in F/D latch
  - then, control bypass MUX
    - if \(rs2(X) == rd(X/M)\) then ALUOUT(M)
    - else if \(rs2(X) == rd(M/W)\) then ALUOUT(W)
Pipeline Diagrams with Bypassing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add R1, R5, R3</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sub R2, R4, R1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>load R1, 24(R5)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>add R3, R6, R7</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sub R2, R4, R1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* even with full bypassing, not all RAW stalls can be avoided
* example: load to ALU in consecutive cycles

**example 1**

```
123456789 1 0 1 1
add R1, R5, R3
sub R2, R4, R1
```

**example 2**

```
123456789 1 0 1 1
load R1, 24(R5)
add R3, R6, R7
sub R2, R4, R1
```

**example 3**

```
123456789 1 0 1 1
load R1, 24(R5)
add R3, R6, R7
sub R2, R4, R1
```

WAR: Write After Read

write-after-read (WAR) = artificial (name) dependence

```
add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3
```

* problem: *add* could use wrong value for R2
* can’t happen in vanilla pipeline (reads in ID, writes in WB)
  * can happen if: early writes (e.g., auto-increment) + late reads
  * can happen if: out-of-order reads (e.g., out-of-order execution)
* artificial: using different output register for *sub* would solve
  * The dependence is on the name R2, but not on actual dataflow

WAW: Write After Write

write-after-write (WAW) = artificial (name) dependence

```
add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3
```

* problem: reordering could leave wrong value in R1
  * later instruction that reads R1 would get wrong value
* can’t happen in vanilla pipeline (register writes are in order)
  * another reason for making ALU ops go through MEM stage
  * can happen: multi-cycle operations (e.g., FP, cache misses)
* artificial: using different output register for *or* would solve
  * Also a dependence on a name: R1
RAR: Read After Read

read-after-read (RAR)

- add R1, R2, R3
- sub R2, R4, R
- or R1, R6, R3

• no problem: R3 is correct even with reordering

Memory Data Hazards

have seen register hazards, can also have memory hazards

 RAW | WAR | WAW
--- | --- | ---
store R1,0(SP) | load R4,0(SP) | store R1,0(SP)
load R4,0(SP) | store R1,0(SP) | store R4,0(SP)

```
1 2 3 4 5 6 7 8 9
```

- in simple pipeline, memory hazards are easy
  - in-order
  - one at a time
  - read & write in same stage

- in general, though, more difficult than register hazards

Hazards vs. Dependences

dependence: fixed property of instruction stream (i.e., program)
hazard: property of program and processor organization

- implies potential for executing things in wrong order
  - potential only exists if instructions can be simultaneously “in-flight”
  - property of dynamic distance between instrs vs. pipeline depth

For example, can have RAW dependence with or without hazard
  - depends on pipeline

Control Hazards

when an instruction affects which instruction executes next

- naive solution: stall until outcome is available (end of EX)
  + simple
    - low performance (2 cycles here, longer in general)
  - e.g. 15% branches * 2 cycle stall ⇒ 30% CPI increase!

```
```
- store R4,0(R5)
- bne R2,R3,loop
- sub R1,R6,R3

- naive solution: stall until outcome is available (end of EX)
  + simple
    - low performance (2 cycles here, longer in general)
  - e.g. 15% branches * 2 cycle stall ⇒ 30% CPI increase!
Control Hazards: “Fast” Branches

**fast branches**: can be evaluated in ID (rather than EX)
- reduce stall from 2 cycles to 1

```
sw R4, 0(R5)  |  F  D  X  M  W
bne R2, R3, loop | F  D  X  M  W
??            | c* F  D  X  M  W
```
- requires more hardware
  - dedicated ID adder for (PC + immediate) targets
- requires simple branch instructions
  - no time to compare two registers (would need full ALU)
  - comparisons with 0 are fast (beqz, bnez)

Control Hazards: Delayed Branches

delayed branch: execute next instruction whether taken or not
- instruction after branch said to be in “delay slot”
- old microcode trick stolen by RISC (MIPS)

```
sw R4, 0(R5)  |  F  D  X  M  W
bne R2, R3, loop | F  D  X  M  W
??            | c* F  D  X  M  W
```

Control Hazards: Speculative Execution

**idea**: doing anything is better than waiting around doing nothing
- speculative execution
  - guess branch target ⇒ start executing at guessed position
  - execute branch ⇒ verify (check) guess
  - minimize penalty if guess is right (to zero?)
    - wrong guess could be worse than not guessing
- branch prediction: guessing the branch
  - one of the “important” problems in computer architecture
  - very heavily researched area in last 15 years
  - static: prediction by compiler
  - dynamic: prediction by hardware
  - hybrid: compiler hints to hardware predictor

What To Put In Delay Slot?

- instruction from before branch
  - when? if branch and instruction are independent
  - helps? always
- instruction from target (taken) path
  - when? if safe to execute, but may have to duplicate code
  - helps? on taken branch, but may increase code size
- instruction from fall-through (not-taken) path
  - when? if safe to execute
  - helps? on not-taken branch
- upshot: short-sighted ISA feature
  - not a big win for today’s machines (why? consider pipeline depth)
  - complicates interrupt handling (later)
The Speculation Game

speculation: engagement in risky business transactions on the chance of quick or considerable profit

• speculative execution (control speculation)
  • execute before all parameters known with certainty

+ correct speculation
  • avoid stall/get result early, performance improves

– incorrect speculation (mis-speculation)
  • must abort/squash incorrect instructions
  • must undo incorrect changes (recover pre-speculation state)

• the speculation game: profit > penalty
  • profit = speculation accuracy * correct-speculation gain
  • penalty = (1 – speculation accuracy) * mis-speculation penalty

Static (Compiler) Branch Prediction

Some static prediction options

• predict always not-taken
  • very simple, since we already know the target (PC+4)
  • most branches (~65%) are taken (why?)

• predict always taken
  • better performance
  • more difficult, must know target before branch is decoded

• predict backward taken
  • most backward branches are taken

• predict specific opcodes taken
  • use profiles to predict on per-static branch basis
  • pretty good

Speculative Execution Scenarios

Correct speculation

<table>
<thead>
<tr>
<th>inst8/B</th>
<th>cycle1: fetch branch, predict next (inst8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F D X M W</td>
<td>c2, c3: fetch inst8, inst9</td>
</tr>
<tr>
<td>F D X</td>
<td>c3: execute/verify branch ⇒ correct</td>
</tr>
<tr>
<td>F D</td>
<td>nothing needs to be fixed or changed</td>
</tr>
</tbody>
</table>

Incorrect speculation: mis-speculation

<table>
<thead>
<tr>
<th>inst8/B</th>
<th>c1: fetch branch, predict next (inst1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F D X M W</td>
<td>c2, c3: fetch inst1, inst2</td>
</tr>
<tr>
<td>F D</td>
<td>c3: execute/verify branch ⇒ wrong</td>
</tr>
<tr>
<td>F D</td>
<td>c3: squash (abort) inst1, inst2 (flush F/D)</td>
</tr>
<tr>
<td>F D</td>
<td>c4: fetch inst8</td>
</tr>
</tbody>
</table>

Comparison of Some Static Schemes

\[
\text{CPI penalty} = \%_{\text{branch}} \times [\%_{T} \times \text{penalty}_{T} + \%_{NT} \times \text{penalty}_{NT}]
\]

<table>
<thead>
<tr>
<th>scheme</th>
<th>\text{penalty}_{T}</th>
<th>\text{penalty}_{NT}</th>
<th>CPI penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>stall</td>
<td>2</td>
<td>2</td>
<td>0.28</td>
</tr>
<tr>
<td>fast branch</td>
<td>1</td>
<td>1</td>
<td>0.14</td>
</tr>
<tr>
<td>delayed branch</td>
<td>1.5</td>
<td>1.5</td>
<td>0.21</td>
</tr>
<tr>
<td>not-taken</td>
<td>2</td>
<td>0</td>
<td>0.18</td>
</tr>
<tr>
<td>taken</td>
<td>0</td>
<td>2</td>
<td>0.10</td>
</tr>
</tbody>
</table>
Dynamic Branch Prediction

hardware (BP) guesses whether and where a branch will go

- start with branch PC (0x64) and produce
  - direction (Taken)
  - direction + target PC (0x74)
  - direction + target PC + target instruction (add r3, r2, r1)

Branch History Table (BHT)

branch PC ⇒ prediction (T, NT)
- need decoder/adder to compute target if taken
  - branch history table (BHT)
    - read prediction with least significant bits (LSBs) of branch PC
    - change bit on misprediction
      + simple
    - multiple PCs may map to same bit (aliasing)
  - major improvements
    - two-bit counters [Smith]
    - correlating/two-level predictors [Patt]
    - hybrid predictors [McFarling]

Improvement: Two-bit Counters

example: 4-iteration inner loop branch

<table>
<thead>
<tr>
<th>state/prediction</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch outcome</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
</tr>
<tr>
<td>mis-prediction?</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

- problem: two mis-predictions per loop
- solution: 2-bit saturating counter to implement hysteresis
  - 4 states: strong/weak not-taken (N/n), strong/weak taken (T/t)
  - transitions: N ⇔ n ⇔ t ⇔ T

<table>
<thead>
<tr>
<th>state/prediction</th>
<th>n</th>
<th>t</th>
<th>T</th>
<th>t</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>t</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch outcome</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
</tr>
<tr>
<td>mis-prediction?</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

+ only one mis-prediction per iteration

Improvement: Correlating Predictors

different branches may be correlated
- outcome of branch depends on outcome of other branches
  - makes intuitive sense (programs are written this way)
  - e.g., if the first two conditions are true, then third is false
    - if (aa == 2) aa = 0;
    - if (bb == 2) bb = 0;
    - if (aa != bb) { ... }

revelation: prediction = f(branch PC, recent branch outcomes)
- revolution: BP accuracies increased dramatically
- lots of research in designing that function for best BP
Correlating (Two-Level) Predictors

- branch history shift register (BHR) holds recent outcomes
  - combination of PC and BHR accesses BHT
  - basically, multiple predictions per branch, choose based on history

  **Design Space**
  - Number of BHRs
    - Multiple BHRs ("local", Intel)
    - 1 global BHR ("global", everyone else)
  - PC/BHR overlap
    - Full, partial, none (concatenated?)
  - Popular design: Gshare [McFarling]
    - 1 global BHR, full overlap, f = XOR

Correlating Predictor Example

- Example with alternating T,N (1-bit BHT, no correlation)

<table>
<thead>
<tr>
<th>State/Prediction</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Outcome</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>Mis-Prediction?</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

  - Add 1 1-bit BHR, concatenate with PC
    - Effectively, two predictors per PC
    - Top (BHR=N) bottom (BHR=T) active entry

<table>
<thead>
<tr>
<th>State/Prediction</th>
<th>N</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Outcome</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>Mis-Prediction?</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Hybrid/Competitive/Tournament Predictors

**Observation:** different schemes work better for different branches

**Idea:** multiple predictors, choose on per static-branch basis

**Mechanics**

- Two (or more) predictors
- Chooser
  - If chosen predictor is wrong...
  - ...and other is right...
  - Flip chooser
- Popular design: Gselect [McFarling]
  - Gshare + 2-bit saturating counter

Branch Target Buffer (BTB)

- Branch PC \(\rightarrow\) target PC
  - Target PC available at end of IF stage
    + No bubble for correct predictions
  - Branch target buffer (BTB)
    - Index: branch PC
    - Data: target PC (+ T/NT?)
    - Tags: branch PC (why are tags needed here and not in BHT?)
      - Many more bits per entry than BHT
    - Considerations: combine with I-cache? Store not-taken branches?
  - Branch target cache (BTC)
    - Data: target PC + target instruction(s)
    - Enables "branch folding" optimization (branch removed from pipe)
**Jump Prediction**

exploit behavior of different kinds of jumps to improve prediction

- function returns
  - use hardware return address stack (RAS)
  - call pushes return address on top of RAS
  - for return, predict address at top of RAS and pop
    - trouble: must manage speculatively
- indirect jumps (switches, virtual functions)
  - more than one taken target per jump
  - path-based BTB [Driesen+Holzle]

**Branch Issues**

issue1: how do we know at IF which instructions are branches?

- BTB: don’t need to “know”
  - check every instruction: BTB entry ⇒ instruction is a branch

issue2: BHR (RAS) depend on branch (call) history

- when are these updated?
  - at WB is too late (if another branch is in-flight)
  - at IF (after prediction)
  - must be able to recover BHR (RAS) on mis-speculation (nasty)

**Adding Multi-Cycle Operations**

RISC tenet #1: “single-cycle operations”

- why was this such a big deal?
- fact: not all operations complete in 1 cycle
  - FP add, int/FP multiply: 2–4 cycles, int/FP divide: 20–50 cycles
  - data cache misses: 10–150 cycles!
- slow clock cycle down to slowest operation?
  - can’t without incurring huge performance loss
- solution: extend pipeline - add pipeline stages to EX

**Extended Pipeline**

- separate integer/FP, pipe register files
- loads/stores in integer pipeline only (why?)
- additional, parallel functional units
  - $E_+$: FP adder (2 cycles, pipelined)
  - $E^+$: FP/integer multiplier (4 cycles, pipelined)
  - $E^/$: FP/integer divider (20 cycles, not pipelined)
Multi-Cycle Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>divf</td>
<td>f0, f1, f2</td>
<td>F</td>
<td>D</td>
<td>E/</td>
<td>E/</td>
<td>E/</td>
<td>E/</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulf</td>
<td>f0, f3, f4</td>
<td>F</td>
<td>D</td>
<td>E*</td>
<td>E*</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addf</td>
<td>f5, f6, f7</td>
<td>F</td>
<td>D</td>
<td>E+</td>
<td>E+</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subf</td>
<td>f8, f6, f7</td>
<td>F</td>
<td>D</td>
<td>*</td>
<td>E+</td>
<td>E+</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulf</td>
<td>f9, f8, f7</td>
<td>F</td>
<td>D</td>
<td>*</td>
<td>*</td>
<td>E*</td>
<td>E*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- write-after-write (WAW) hazards
- register write port structural hazards
- functional unit structural hazards
- elongated read-after-write (RAW) hazards

Another Multi-Cycle Example

example: SAXPY (math kernel)


<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldf</td>
<td>f2, 0(r1)</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulf</td>
<td>f6, f0, f2</td>
<td>F</td>
<td>D</td>
<td>d*</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addf</td>
<td>f8, f6, f7</td>
<td>F</td>
<td>D</td>
<td>d*</td>
<td>d*</td>
<td>E+</td>
<td>E+</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stf</td>
<td>f8, 0(r3)</td>
<td>F</td>
<td>p*</td>
<td>p*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>r1, r1, #4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>r2, r2, #4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>r3, r3, #4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Write Port Structural Hazards

where are these resolved?
- multiple writeback ports?
  - not a good idea (why not?)
- in ID?
  - reserve writeback slot in ID (writeback reservation bits)
  - simple, keeps stall logic localized to ID stage
  - won’t work for cache misses (why not?)
- in MEM?
  - works for cache misses, better utilization
  - two stall controls (F/D and M/W) must be synchronized
- in general: cache misses are hard
  - don’t know in ID whether they will happen early enough (in ID)

WAW Hazards

how are these dealt with?
- stall younger instruction writeback?
  + intuitive, simpler
  - lower performance (cascading writeback structural hazards)
- abort (don’t do) older instruction writeback?
  + no performance loss
  - but what if intermediate instruction causes an interrupt (next)
Dealing With Interrupts

Interrupts (aka faults, exceptions, traps)
- e.g., arithmetic overflow, divide by zero, protection violation
- e.g., I/O device request, OS call, page fault

Classifying interrupts
- terminal (fatal) vs. restartable (control returned to program)
- synchronous (internal) vs. asynchronous (external)
- user vs. coerced
- maskable (ignorable) vs. non-maskable
- between instructions vs. within instruction

Precise Interrupts

“unobserved system can exist in any intermediate state, upon observation system collapses to well-defined state”
- 2nd postulate of quantum mechanics

- system $\Rightarrow$ processor, observation $\Rightarrow$ interrupt

What is the “well-defined” state?
- von Neumann: “sequential, instruction atomic execution”
- precise state at interrupt
  - all instructions older than interrupt are complete
  - all instructions younger than interrupt haven’t started
- implies interrupts are taken in program order
- necessary for VM (why?), “highly recommended” by IEEE

Interrupt Example: Data Page Fault

<table>
<thead>
<tr>
<th>inst0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst3</td>
<td>F</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst4</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trap0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- squash (effects of) younger instructions
- inject fake TRAP instruction into IF
- from here, like a SYSCALL

More Interrupts

- interrupts can occur at different stages
  - IF, MEM: page fault, misaligned data, protection violation
  - ID: illegal/privileged instruction
  - EX: arithmetic exception
- too complicated to draw what goes on here
  - cycle2: instruction page fault, flush inst1, inject TRAP
  - c4: data page fault, flush inst0, inst1, TRAP
    - can get into an infinite loop here (with help of OS page placement)
Posted Interrupts

posted interrupts
• set interrupt bit when condition is raised
• check interrupt bit (potentially “take” interrupt) in WB
  + interrupts are taken in order
  – longer latency, more complex

<table>
<thead>
<tr>
<th>inst0</th>
<th>F D X M W</th>
<th>data page fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 5 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 8 9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inst1</th>
<th>F D X M W</th>
<th>instruction page fault</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W</td>
<td></td>
</tr>
</tbody>
</table>

what happens now?
• c2: set inst1 bit
• c4: set inst0 bit
• c5: take inst0 interrupt

Interrupts and Multi-Cycle Operations

multi-cycle operations + precise state = trouble
• #1: how to undo early writes?
  • e.g., must make it seem as if mulf hasn’t executed
  • undo writes: future file, history file -> ugly!
• #2: how to take interrupts in-order if WB is not in-order?
  • force in-order WB
  – slow

<table>
<thead>
<tr>
<th>1 2 3 4 5 6 7 8 9 10 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>divf f0,f1,f2</td>
</tr>
<tr>
<td>mulf f3,f4,f5</td>
</tr>
<tr>
<td>addf f6,f7,f8</td>
</tr>
</tbody>
</table>

div by 0 (posted)
<table>
<thead>
<tr>
<th>F D E* E* E* E* E* W</th>
</tr>
</thead>
<tbody>
<tr>
<td>F D E* E* W</td>
</tr>
<tr>
<td>F D E+ E+ s* W</td>
</tr>
</tbody>
</table>

Interrupts Are Nasty

• odd bits of state must be precise (e.g., CC)
• delayed branches
  • what if instruction in delay slot takes an interrupt?
• modes with early-writes (e.g., auto-increment)
  • must undo write (e.g., future-file, history-file)
• some machines had precise interrupts only in integer pipe
  • sufficient for implementing VM
  • e.g., VAX/Alpha

Lucky for us, there’s a nice, clean way to handle precise state
• We’ll see how this is done in a couple of lectures ...

Summary

• principles of pipelining
  • pipeline depth: clock rate vs. number of stalls (CPI)
• hazards
  • structural
  • data (RAW, WAR, WAW)
  • control
• multi-cycle operations
  • structural hazards, WAW hazards
• interrupts
  • precise state
next up: dynamic ILP (chapter 3)