This Unit: Pipelining

- Basic Pipelining
  > Single, in-order issue
  > Clock rate vs. IPC
- Data Hazards
  > Hardware: stalling and bypassing
  > Software: pipeline scheduling
- Control Hazards
  > Branch prediction
- Precise state

Quick Review

- Basic datapath: fetch, decode, execute
- Single-cycle control: hardwired
  > Low CPI (1)
  > Long clock period (to accommodate slowest instruction)
- Multi-cycle control: micro-programmed
  > Short clock period
  > High CPI
- Can we have both low CPI and short clock period?
  > Not if datapath executes only one instruction at a time
  > No good way to make a single instruction go faster

Pipelining

- Important performance technique
  > Improves instruction throughput rather instruction latency
- Begin with multi-cycle design
  > When instruction advances from stage 1 to 2
  > Allow next instruction to enter stage 1
  > Form of parallelism: " insn-stage parallelism"
  > Individual instruction takes the same number of stages
  > But instructions enter and leave at a much faster rate
- Automotive assembly line analogy

5 Stage Pipelined Datapath

- Temporary values (PC, IR, A, B, O, D) re-latched every stage
  > Why? 5 insns may be in pipeline at once, they share a single PC?
  > Notice, PC not latched after ALU stage (why not?)

Pipeline Terminology

- Five stage: Fetch, Decode, Execute, Memory, Writeback
  > Nothing magical about the number 5 (Pentium 4 has 22 stages)
- Latches (pipeline registers) named by stages they separate
  > PC, F/D, D/X, X/M, M/W
Pipeline Control

- One single-cycle controller, but pipeline the control signals

Abstract Pipeline

- This is an integer pipeline
  - Execution stages are X,M,W
- Usually also one or more floating-point (FP) pipelines
  - Separate FP register file
  - One "pipeline" per functional unit: E+, E*, E/
    - "Pipeline": functional unit need not be pipelined (e.g., E/)
    - Execution stages are E+, E', W (no M)

Floating Point Pipelines

Pipeline Diagram

- Pipeline diagram
  - Cycles across, insns down
  - Convention: X means ld r4,0(r5) finishes execute stage and writes into X/M latch at end of cycle 4
- Reverse stream analogy
  - "Downstream": earlier stages, younger insns
  - "Upstream": later stages, older insns
  - Reverse? instruction stream fixed, pipeline flows over it
    - Architects see instruction stream as fixed by program/compiler

Pipeline Performance Calculation

- Back of the envelope calculation
  - Branch: 20%, load: 20%, store: 10%, other: 50%
- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performance = 50ns/insn
- Pipelined
  - Clock period = 12ns
  - CPI = 4 (each insn takes 5 cycles, but 1 completes each cycle)
  - Performance = 12ns/insn

Principles of Pipelining

- Let: insn execution require N stages, each takes t_i time
  - Single-cycle execution
    - L_i (1-insn latency) = t_i
    - T (throughput) = 1/L
    - L_M (M-insn latency, where M>>1) = M*L
  - Now: N-stage pipeline
    - t_i (latency)
    - T (throughput) = 1/\sum_{i=1}^{N} t_i
    - L_M (M-insn latency, where M>>1) = M*L
  - Shorten (speedup) = [M*L_i / (\sum_{i=1}^{N} t_i)] = \leq N
    - Q: for arbitrarily high speedup, use arbitrarily high N?
No, Part I: Pipeline Overhead

- Let $O$ be extra delay per pipeline stage
  - Latch overhead: pipeline latches take time
  - Clock/data skew

- Now: $N$-stage pipeline with overhead
  - Assume $\max(t_{\text{delay}}) = L_1/N$
  - $L_1 + P + O = L_1 + N*O$
  - $T + P + O = 1/(L_1/N + O) = 1/(1/T + O) \leq T, \leq T/O$

- $O$ limits throughput and speedup → useful $N$

No, Part II: Hazards

- Dependence: relationship that serializes two insns
  - Data: two insns use the same value or storage location
  - Control: one instruction affects whether another executes at all
  - Maybe: two insns may have a dependence

- Hazard: dependence causes potential incorrect execution
  - Possibility of using or corrupting data or execution flow
  - Structural: two insns want to use same structure, one must wait
  - Often fixed with stalls: insn stays in same stage for multiple cycles

- Let $H$ be average number of hazard stall cycles per instruction
  - $L_1 + P + H = L_1 + P$ (no hazards for one instruction)
  - $T + P + H = \frac{N}{N+H} * N/L_1 = \frac{N}{N+H} * T + P$
  - $L_1 + P + H = M * \frac{L_1}{N} + M * O = M * L_1 + M * O$
  - $S + P + H = \frac{N}{N+H} * S + P$

- $H$ also limit throughput, speedup → useful $N$
  - $N \uparrow \rightarrow H \uparrow$ (more insns "in flight" → more dependences become hazards)

Clock Rate vs. IPC

- Deeper pipeline (bigger $N$)
  - frequency
  - IPC

- Ultimate metric is $IPC \cdot frequency$
  - But Intel got people to buy frequency, not $IPC \cdot frequency$

- Trend has been for deeper pipelines
  - Intel example:
    - 486: 5 stages (50+ gate delays / clock)
    - Pentium: 7 stages
    - Pentium 1/II/III: 12 stages
    - Pentium 4: 22 stages (10 gate delays / clock)
    - 800 MHz Pentium III was faster than 1 GHz Pentium 4

Optimizing Pipeline Depth

- Parameterize clock cycle in terms of gate delays
  - $G$ gate delays to process (fetch, decode, execute) a single insn
  - $O$ gate delays overhead per stage
  - $X$ average stall per instruction per stage
    - Simplistic: real $X$ function much, much more complex

- Compute optimal $N$ (pipeline stages) given $G, O, X$
  - $IPC = \frac{1}{(1 + X \cdot N)}$
  - $f = \frac{1}{(G \cdot N + O)}$
  - Example: $G = 80$, $O = 1, X = 0.16$, $IPC = \frac{1}{(1 + 0.16 \cdot 80)} = 0.059$, $f = \frac{1}{(80 \cdot 80 + 1)} = 0.004$

- Optimizes performance! What about power?

Managing a Pipeline

- Proper flow requires two pipeline operations
  - Mess with latch write-enable and clear signals to achieve

- Operation I: stall
  - Effect: stops some insns in their current stages
  - Use: make younger insns wait for older ones to complete
  - Implementation: de-assert write-enable

- Operation II: flush
  - Effect: removes insns from current stages
  - Use: see later
  - Implementation: assert clear signals

- Both stall and flush must be propagated to younger insns
Structural Hazards

- Structural hazard: resource needed twice in one cycle
  - Example: shared I/D$
  - What should we do in this case, and in general?

Avoiding Structural Hazards (PRS)

- Pipeline the contended resource
  - No IPC degradation, low area, power overheads
  - Sometimes tricky to implement (e.g., for RAMs)
  - For multi-cycle resources (e.g., multiplier)

- Replicate the contended resource
  - No IPC degradation
  - Increased area, power, latency (interconnect delay?)
  - For cheap, divisible, or highly contended resources (e.g., I/D$

- Schedule pipeline to reduce structural hazards (RISC)
  - Design ISA so instructions use registers at most once
  - Eliminate same insn hazards
  - Always in same pipe stage (hazards between two of same insn)
  - Reason why integer operations forced to go through M stage
  - Always for one cycle

Data Hazards

- Real insn sequences pass values via registers/memory
  - Three kinds of data dependences (where’s the fourth?)

- Dependence is property of the program and ISA
- Data hazards: function of data dependences and pipeline
  - Potential for executing dependent insns in wrong order
  - Require both insns to be in pipeline (“in flight”) simultaneously

Dependences and Loops

- Data dependences in loops
  - Intra-loop: within same iteration
  - Inter-loop: across iterations
  - Example: DARPHY (Double precision $A \times X + Y$)

```
for (i=0;i<100;i++)
  Z[i]=A*X[i]+Y[i];
0: ld f2,X(r1)
1: mul f2,f0,f4
2: ld f6,Y(r1)
3: add f8,f6,f8
4: st f8,Z(r1)
5: add r1,r3,r5
6: cmp r1,r3,r6
7: beq r2,Loop
```

Fixing Structural Hazards

- Can fix structural hazards by stalling
  - $s^*$ = structural stall
  - Q: which one to stall: ld or add?
    - Always safe to stall younger instruction (here add)
    - Fetch stall logic: $(X,M,op == 1d || X,M,op == 4t)$
    - But not always the best thing to do performance wise (?)
    - Low cost, simple
      - Decreases IPC
      - Upshot: better to avoid by design than to fix

RAW

- Need-after-write (RAW)
  - Problem: swap would mean add uses wrong value for $r1$
  - True: value flows through this dependency
    - Using different output register for add doesn’t help

```
add r2,r3,r1
sub r1,r3,r5
or r6,r3,r1
```

```
add r2,r3,r1
sub r1,r3,r5
or r6,r3,r1
```
Pipeline Diagrams with Bypassing

- If bypass exists, "from" stages execute in same cycle
  - Example: full bypass, use MX bypass
    
    | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
    |---|---|---|---|---|---|---|---|---|---|
    | d2 | r3 | d2 | r4 | d2 | r4 | d2 | r4 | d2 | r4 |
    | D  | X  | M  | W  | D  | X  | M  | W  | D  | X  |
  
  - Example: full bypass, use WX bypass
    
    | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
    |---|---|---|---|---|---|---|---|---|
    | d2 | r3 | d2 | r4 | d2 | r4 | d2 | r4 | d2 | r4 |
    | D  | X  | M  | W  | D  | X  | M  | W  | D  | X  |
  
  - Example: WM bypass
    
    | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
    |---|---|---|---|---|---|---|---|---|
    | F  | D  | X  | M  | F  | D  | X  | M  | F  | D  |

Two Stall Timings (without bypassing)

- Depend on how D and W stages share regfile
  - Each gets regfile for half a cycle
    - 1st half D reads, 2nd half W writes 3 cycle stall
    - \( d^* \) = data stall, \( p^* \) = propagated 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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r2, r3</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
</tr>
<tr>
<td>sub r1, r4</td>
<td>F</td>
<td>d*</td>
<td>d*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
<td>X</td>
</tr>
</tbody>
</table>

- 1st half W writes, 2nd half D reads 2 cycle stall
- How does the stall logic change here?

<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r2, r3</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
</tr>
<tr>
<td>sub r1, r4</td>
<td>F</td>
<td>d*</td>
<td>d*</td>
<td>B</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
<td>X</td>
</tr>
<tr>
<td>add r5, r6</td>
<td>p*</td>
<td>p*</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
<td>X</td>
</tr>
</tbody>
</table>

Bypass Logic

- Bypass logic: similar but separate from stall logic
  - Stall logic controls latches, bypass logic controls mux inputs
  - Complement one another: can't bypass
  - ALU input mux bypass logic:
    - \((D/X.rs2) \& (W/M.rd==D/X.rs2)\) \( \rightarrow \) 2 // check first
    - \((D/X.rs2) \& (M/W.rd==D/X.rs2)\) \( \rightarrow \) 1 // check second
    - \((D/X.rs2)\) \( \rightarrow \) 0 // check last

Load-Use Stalls

- Even with full bypass, stall logic is unavoidable
  - **Load-use stall**
    - Load value not ready at beginning of M \( \rightarrow \) can't use MX bypass
    - Use WX bypass

<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [r3+4]</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
</tr>
<tr>
<td>sub r1, r4</td>
<td>F</td>
<td>d*</td>
<td>d*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

- Aside: with WX bypassing, stall logic can be in D or X

<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [r3+4]</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
</tr>
<tr>
<td>sub r1, r4</td>
<td>F</td>
<td>d*</td>
<td>d*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>D</td>
<td>X</td>
</tr>
</tbody>
</table>

- Aside II: how does stall/bypass logic handle cache misses?
Compiler Scheduling

- Compiler can schedule (move) insns to reduce stalls
  - Basic pipeline scheduling: eliminate back-to-back load-use pairs
  - Example code sequence: `a = b + c; d = f - e;
  - MIPS notation: `ld r2,4(sp)` is `ld [sp+4]
  - What are some limitations/requirements for this approach?

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2,4(sp)</td>
<td>ld r2,4(sp)</td>
</tr>
<tr>
<td>ld r3,8(sp)</td>
<td>ld r3,8(sp)</td>
</tr>
<tr>
<td>add r3,r2,r1 //stall</td>
<td>ld r4,16(sp)</td>
</tr>
<tr>
<td>st r1,0(sp)</td>
<td>add r3,r2,r1 //no stall</td>
</tr>
<tr>
<td>ld r5,16(sp)</td>
<td>ld r6,20(sp)</td>
</tr>
<tr>
<td>ld r6,20(sp)</td>
<td>sub r5,r6,r4 //no stall</td>
</tr>
<tr>
<td>sub r2,r1,r1 //stall</td>
<td>sub r2,r1,r1</td>
</tr>
<tr>
<td>st r1,12(sp)</td>
<td>st r1,12(sp)</td>
</tr>
</tbody>
</table>

Before: max 4 values live
After: max 4 values live → 3 registers not enough → WAW violations

Compiler Scheduling Requires

- Enough registers
  - To hold additional "live" values
  - Example code contains 7 different values (including sp)
  - Before: max 3 values live at any time → 3 registers enough
  - After: max 4 values live → 3 registers not enough → WAW violations

<table>
<thead>
<tr>
<th>Original</th>
<th>Wrong?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2,4(sp)</td>
<td>ld r2,4(sp)</td>
</tr>
<tr>
<td>ld r3,8(sp)</td>
<td>ld r3,8(sp)</td>
</tr>
<tr>
<td>add r1,r2,r1 //stall</td>
<td>ld r4,16(sp)</td>
</tr>
<tr>
<td>st r1,0(sp)</td>
<td>add r1,r2,r1 //WAW</td>
</tr>
<tr>
<td>ld r2,16(sp)</td>
<td>ld r1,20(sp)</td>
</tr>
<tr>
<td>ld r1,20(sp)</td>
<td>st r1,0(sp) //WAW</td>
</tr>
<tr>
<td>sub r2,r1,r1 //stall</td>
<td>sub r2,r1,r1</td>
</tr>
<tr>
<td>st r1,12(sp)</td>
<td>st r1,12(sp)</td>
</tr>
</tbody>
</table>

Before: max 3 values live at any time
After: max 4 values live → 3 registers not enough → WAW violations

WAW Hazards

- Write-after-write (WAW)
  - `add r2,r3,r1`  `sub r1,r4,r2`  `or r6,r3,r1`
  - Compiler effects
    - Scheduling problem: reordering would leave wrong value in `r1`
    - Later instruction reading `r1` would get wrong value
    - Artificial: no value flows through dependence
  - Pipeline effects
    - Doesn’t affect in-order pipeline with single-cycle operations
    - One reason for making ALU operations go through M stage
    - Can happen with multi-cycle operations (e.g., FP or cache misses)

Handling WAW Hazards

<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>divf r0,r1→r2</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>E</td>
<td>F</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subf r0→r1(r2)</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>
<td></td>
</tr>
<tr>
<td>addf r0,r1→r2</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>
</tbody>
</table>

- What to do?
  - Option I: stall younger instruction (addf) at writeback
    - Intuitive, simple
    - Lower performance, cascading W structural hazards
  - Option II: cancel older instruction (divf) writeback
    - No performance loss
    - What if divf or subf cause an exception (e.g., /0, page fault)?
### Handling Interrupts

- **How are interrupts/exceptions handled in a pipeline?**
  - **Interrupt:** external, e.g., timer, I/O device requests
  - **Exception:** internal, e.g., /0, page fault, illegal instruction
  - We care about restartable interrupts (e.g. `stf` page fault)

- **Von Neumann says**
  - "Instruction execution should appear sequential and atomic"
  - Iron X should complete before instruction X+1 should begin
  - Doesn't physically have to be this way (e.g., pipeline)
  - But be ready to restore to this state at a moments notice
  - Called **precise state** or **precise interrupts**

- So far, have seen/dealt with register dependences
- Dependencies also exist through memory
- What about two simultaneous in-flight interrupts
  - Example: `add` page fault; `divf`/0
    - Interrupts must be handled in program order (as if fired)
    - Handler for `stf` must see program as if `divf` hasn't started
  - Must defer interrupts until writeback and force in-order writeback
  - In general: interrupts are really nasty
    - Some processors (Alpha) only implement precise integer interrupts
    - Easier because fewer WAR scenarios
    - Most floating-point interrupts are non-restartable anyway
      - `divf`/0 → rescale computation to prevent underflow
    - Typically doesn't restart computation at excepting instruction

### Memory Data Hazards

- So far, have seen/dealt with register dependences
  - Dependencies also exist through memory
  - But in an in-order pipeline like ours, they do not become hazards
  - Memory read and write happen at the same stage
  - Register read happens three stages earlier than register write
  - In general: memory dependences more difficult than register dependences

### Control Hazards

- **Control hazards**
  - Must fetch post branch insns before branch outcome is known
  - Default: assume "not-taken" (at fetch, can't tell it's a branch)
  - Control hazards indicated with c* (or not at all)
  - Taken branch penalty is 2 cycles

### WAR Hazards

- **Write-after-read (WAR)**
  - `add s3, r3, r1` (or `r6, r3, r1`)
  - Compiler effects
    - Scheduling problem: reordering would mean `add` uses wrong value for `r2`
    - Artificial: solve using different output register name for `add`
  - Pipeline effects
    - Can't happen in simple in-order pipeline
    - Can happen with out-of-order execution

### WAR Hazards

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
<th>Stage 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>div f0, f1</code></td>
<td>F D E E</td>
<td>E E E E</td>
<td>E E E E</td>
<td>E E E E</td>
</tr>
<tr>
<td><code>stf f2</code></td>
<td>F D d d d X M W</td>
<td>F D E+ E+ W</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add f0, f1</code></td>
<td>F D E E</td>
<td>E E E E</td>
<td>E E E E</td>
<td>E E E E</td>
</tr>
</tbody>
</table>

- In this situation
  - Make it appear as if `divf` finished and `stf, addf` hasn't started
  - Allow `divf` to writeback
  - **Flush** `stf` and `addf` (so that's what a flush is for)
    - But `addf` has already written back
    - Keep an "undo" register file? Complicated
    - Force in-order writebacks? Slow
    - Other solutions? Later
  - **Invoke exception handler**
  - **Restart** `stf`

- Back of the envelope calculation
  - **Branch:** 20%, other: 80%, 75% of branches are taken
  - `CPI_{branch} = 1`
  - `CPI_{branch} = 1 + 0.20*0.75*2 = 1.3`
  - Branches cause 30% slowdown
### ISA Branch Techniques

- **Fast branch**: resolves at D, not X
  - Test must be comparison to zero or equality, no time for ALU (RISC...)
  - New taken branch penalty is 1
  - Additional comparison insns (e.g., `cmp` or `slt`) for complex tests
  - Must bypass into decode now, too

- **Delayed branch**: branch that takes effect one insn later
  - Schedule insns that are independent of branch into "branch delay slot"
  - Preferably from before branch (always helps then)
  - But from after branch OK too
  - As long as no undoable effects (e.g., a store)
  - Upshot: short-sighted feature (MIPS regrets it)
    - Not a big win in today's pipelines
    - Complicates interrupt handling

### Big Idea: Speculation

- **Speculation**
  - "Engagement in risky transactions on the chance of profit"

- **Speculative execution**
  - Execute before all parameters known with certainty

- **Correct speculation**
  - Avoid stall, improve performance

- **Incorrect speculation (mis-speculation)**
  - Must abort/flush/squash incorrect instructions
  - Must undo incorrect changes (recover pre-speculation state)

  The "game": $[\%_{correct} \times \text{gain}] > [(1-\%_{correct}) \times \text{penalty}]$
Dynamic Branch Prediction

- **BP part I: Target predictor**
  - Applies to all control transfers
  - Supplies target PC, tells if insn is a branch prior to decode
  - Easy

- **BP part II: Direction predictor**
  - Applies to conditional branches only
  - Predicts taken/not-taken
  - Harder

Branch Target Buffer

- **Branch target buffer (BTB)**
  - A small cache: address = PC, data = target-PC
  - Hit? This is a control insn and it’s going to target-PC (if “taken”)
  - Miss? Not a control insn, or one I have never seen before
  - Partial data/tags: full tag not necessary, target-PC is just a guess
  - Aliasing: tag match, but not actual match (OK for BTB)
  - Pentium4 BTB: 2K entries, 4-way set-associative

Why Does a BTB Work?

- Because control insn targets are stable
  - Direct means constant target, indirect means in register
  - Direct conditional branches? Check
  - Direct calls? Check
  - Direct unconditional jumps? Check

  - Indirect conditional branches? Not that useful—not widely supported
  - Indirect calls? Two idioms
    - Dynamically linked functions (DLLs)? Check
    - Dynamically dispatched (virtual) functions? Pretty much check
  - Indirect unconditional jumps? Two idioms
    - Switches? Not really, but these are rare
    - Returns? No, but...

Return Address Stack (RAS)

- Return addresses are easy to predict without a BTB
  - Hardware return address stack (RAS) tracks call sequence
  - Calls push PC+4 onto RAS
  - Prediction for returns is RAS[TOS]
  - Q: How can you tell if an insn is a return before decoding it?
    - Add tags to make RAS a cache
    - (Better) attach pre-decode bits to I$

Branch Direction Prediction

- **Direction predictor (DIRP)**
  - Map conditional-branch PC to taken/not-taken (T/N) decision
  - Seemingly innocuous, but quite difficult to do well
  - Individual conditional branches often unbiased or weakly biased
    - 90%+ one way or the other considered “biased”

Branch History Table (BHT)

- **Branch history table (BHT):** simplest direction predictor
  - PC indexes table of bits (0 = N, 1 = T), no tags
  - Essentially: branch will go same way it went last time
  - Problem: consider inner loop branch below (* = mis-prediction)

  ```
  for (i=0;i<100;i++)
  for (j=0;j<3;j++)
    // whatever;
  ```

  - Two “built-in” mis-predictions per inner loop iteration
  - Branch predictor “changes its mind too quickly”
Two-Bit Saturating Counters (2bc)

- Two-bit saturating counters (2bc) [Smith]
  - Replace each single-bit prediction
  - Force DIRP to mis-predict twice before "changing its mind"
- Hydra Predictor
- Correlated Predictor

Outcome

<table>
<thead>
<tr>
<th>State/prediction</th>
<th>N^2</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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
</tr>
</tbody>
</table>

- Fixes this pathology (which is not contrived, by the way)

Correlated Predictor

- Correlated (two-level) predictor [Patil]
  - Exploits observation that branch outcomes are correlated
  - Maintains separate prediction per (PC, BHR)
    - Branch history register (BHR): recent branch outcomes
    - Simple working example: assume program has one branch
    - BHT: one 1-bit DIRP entry
    - BHT->2BHR: 4 1-bit DIRP entries
- Two-Bit Saturating Counters (2bc)
- Hydra Predictor
- Correlated Predictor

Outcome

<table>
<thead>
<tr>
<th>State/prediction</th>
<th>BHR=NN</th>
<th>N^2</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></td>
<td>BHR=NT</td>
<td>N^*</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>BHR=TN</td>
<td>N</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>BHR=TN</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>BHR=TT</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

- We didn't make anything better, what's the problem?

Hybrid Predictor

- Hybrid (tournament) predictor [McFarling]
  - Attacks correlated predictor BHT utilization problem
  - Idea: combine two predictors
    - Simple BHT predicts history independent branches
    - Correlated predictor predicts only branches that need history
    - Chooser assigns branches to one predictor or the other
    - Branches start in simple BHT, move mis-prediction threshold
    - Correlated predictor can be made smaller, handles fewer branches
    - 90-95% accuracy

Research: Perceptron Predictor

- Perceptron predictor [Jimenez]
  - Attacks BHR size problem using machine learning approach
  - BHT replaced by table of function coefficients \( f_i \) (signed)
  - Predict taken if \( \sum \text{BHR} \cdot f_i > \text{threshold} \)
  - Table size \( \#\text{PC} \cdot \#\text{BHR} \cdot \#f_i \) (can use long BHR: \~ 60 bits)
  - Equivalent correlated predictor would be \( \#\text{PC} \cdot 2^\#\text{BHR} \)
  - How does it learn? Update \( f_i \) when branch is taken
    - BHRs
    - "don't care" F, bits stay near 0, important F, bits saturate
  - Hybrid BHT/perceptron accuracy: 95-98%
Branch Prediction Performance

- Same parameters
  - Branch: 20%, load: 20%, store: 10%, other: 50%
  - 75% of branches are taken
- Dynamic branch prediction
  - Branches predicted with 95% accuracy
  - CPI = 1 + 0.20*0.05*2 = 1.02

Pipeline Performance Summary

- Base CPI is 1, but hazards increase it
- Nothing magical about a 5 stage pipeline
  - PentiumIII has 22 stage pipeline
- Increasing pipeline depth
  - Increases clock frequency (that's why companies do it)
    - But decreases IPC
  - Branch mis-prediction penalty becomes longer
    - More stages between fetch and whenever branch computes
  - Non-bypassed data hazard stalls become longer
    - More stages between register read and write
    - At some point, CPI losses offset clock gains, question is when?

Dynamic Pipeline Power

- Remember control-speculation game
  - 2 cycles * %correct - 0 cycles * (1- %correct)
  - No penalty -> mis-speculation no worse than stalling
  - This is a performance-only view.
  - From a power standpoint, mis-speculation is worse than stalling
- Power control-speculation game
  - 0nJ * %correct - XnJ * (1-%correct)
  - No benefit -> correct speculation no better than stalling
    - Not exactly, increased execution time increases static power
    - How to balance the two?

Research: Speculation Gating

- Speculation gating [Manne+]
  - Extend branch predictor to give prediction + confidence
  - Speculate on high-confidence (mis-prediction unlikely) branches
  - Stall (save energy) on low-confidence branches
- Confidence estimation
  - What kind of hardware circuit estimates confidence?
  - Hard in absolute sense, but easy relative to given threshold
  - Counter-scheme similar to %miss threshold for cache resizing
  - Example: assume 90% accuracy is high confidence
    - PC-indexed table of confidence-estimation counters
    - Correct prediction? table[PC]+=1 ; table[PC]–=9;
    - Prediction for PC is confident if table[PC] > 0;

Research: Razor

- Razor [Uht, Ernst+]
  - Identify pipeline stages with narrow signal margins (e.g., X)
  - Add "Razor" X/M latch: relatches X/M input signals after safe delay
  - Compare X/M latch with "safe" razor X/M latch, different?
    - Flush D,X & M
    - Restart M using X/M razor latch, restart F using D/X latch
    - Pipeline will not "break" -> reduce Vdd until flush rate too high
    - Alternatively: "over-clock" until flush rate too high

Summary

- Principles of pipelining
  - Effects of overhead and hazards
  - Pipeline diagrams
  - Data hazards
  - Stalling and bypassing
  - Control hazards
  - Branch prediction
  - Power techniques
    - Dynamic power: speculation gating
    - Static and dynamic power: razor latches
Research: Runahead Execution

- In-order writebacks essentially imply stalls on D$ misses
  - Can save power... or use idle time for performance
- **Runahead execution** [Dundas+]
  - Shadow regfile kept in sync with main regfile (write to both)
  - D$ miss: continue executing using shadow regfile (disable stores)
  - D$ miss returns: flush pipe and restart with stalled PC
  + Acts like a smart prefetch engine
  + Performs better as cache $t_{miss}$ grows (relative to clock period)