Pipelining: Branch Hazards

(“Which way did he go, George, which way did he go?”)

Control Dependence

• Just as an instruction will be dependent on other instructions to provide its operands (dependence), it will also be dependent on other instructions to determine whether it gets executed or not (dependence or dependence).
• Control dependences are particularly critical with branches.

add $5, $3, $2
sub $6, $5, $2
beq $6, $7, somewhere
and $9, $6, $1
...

somewhere: or $10, $5, $2
add $12, $11, $9
...

Branch Hazards

• Branch dependences can result in branch hazards (when they are too close to be handled correctly in the pipeline).
Branch Hazards

Dealing With Branch Hazards

- Hardware
  - stall until you know which direction
  - reduce hazard through earlier computation of branch direction
  - guess which direction
    - assume not taken (easiest)
    - more educated guess based on history (requires that you know it is a branch before it is even decoded!)
- Hardware/Software
  - nops, or instructions that get executed either way (delayed branch).

Stalling for Branch Hazards

- Seems wasteful, particularly when the branch isn’t taken.
- Makes all branches cost 4 cycles.
Assume Branch Not Taken

- works pretty well when you’re right

<table>
<thead>
<tr>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

beq $4$, $0$, there

and $12$, $2$, $5$

or ...

add ...

sw ...

Assume Branch Not Taken

- same performance as stalling when you’re wrong

<table>
<thead>
<tr>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

beq $4$, $0$, there

and $12$, $2$, $5$

or ...

add ...

there: sub $12$, $4$, $2$

Assume Branch Not Taken

- Performance depends on percentage of time you guess right.
- Flushing an instruction means to prevent it from changing any permanent state (registers, memory, PC).
  - sounds a lot like a bubble...
  - But notice that we need to be able to insert those bubbles later in the pipeline

Reducing the Branch Delay

- can easily get to 2-cycle stall
Stalling for Branch Hazards

beq $4, $0, there
and $12, $2, $5
or ...
add ...
sw ...

Reducing the Branch Delay

• Harder, but possible, to get to 1-cycle stall

The Pipeline with flushing for taken branches

• Notice the IF/ID flush line added.
Eliminating the Branch Stall

- There’s no rule that says we have to see the effect of the branch immediately. Why not wait an extra instruction before branching?
- The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches.
- The instruction after a conditional branch is always executed in those machines, regardless of whether the branch is taken or not!

Filling the branch delay slot

add $5, $3, $7
sub $6, $1, $4
and $7, $8, $2
beq $6, $7, there
nop /* branch delay slot */
add $9, $1, $2
sub $2, $9, $5
...
there:
mult $2, $10, $11

Filling the branch delay slot

- The branch delay slot is only useful if you can find something to put there.
- If you can’t find anything, you must put a nop to insure correctness.
Branch Delay Slots

- This works great for this implementation of the architecture, but becomes a permanent part of the ISA.
- What about the MIPS R10000, which has a 5-cycle branch penalty, and executes 4 instructions per cycle???

Branch Prediction

- Always assuming the branch is not taken is a crude form of.
- What about loops that are 95% of the time?
  - we would like the option of assuming not taken for some branches, and taken for others, depending on ???
Two different 2-bit schemes

Branch History Table
- has limited size
- 2 bits by N (e.g. 4K)
- uses low bits of branch address to choose entry
- what happens when table too small?
- what about even/odd branch?

Branch Prediction
- Latest branch predictors significantly more sophisticated, using more advanced correlating techniques, larger structures, and soon possibly using AI techniques.
- Presupposes what two pieces of information are available at fetch time?
  -
  - Branch Target Buffer supplies this information.

Pipeline performance

```assembly
    loop:    lw $15, 1000($2)
            add $16, $15, $12
            lw $18, 1004($2)
            add $19, $18, $12
            beq $19, $0, loop:
```
Control Hazards -- Key Points

- Control (or branch) hazards arise because we must fetch the next instruction before we know if we are branching or where we are branching.
- Control hazards are detected in hardware.
- We can reduce the impact of control hazards through:
  - early detection of branch address and condition
  - branch prediction
  - branch delay slots

Advanced Pipelining

Pipelining and Exceptions

- Exceptions represent another form of control dependence.
- Therefore, they create a potential branch hazard
- Exceptions must be recognized early enough in the pipeline that subsequent instructions can be flushed before they change any permanent state.
- As long as we do that, everything else works the same as before.
- Exception-handling that always correctly identifies the offending instruction is called precise interrupts.

Pipelining in Today’s Most Advanced Processors

- Not fundamentally different than the techniques we discussed
- Deeper pipelines
- Pipelining is combined with
  - superscalar execution
  - out-of-order execution
  - VLIW (very-long-instruction-word)
**Superscalar Execution**

- To execute four instructions in the same cycle, we must find four independent instructions.
- If the four instructions fetched are guaranteed by the compiler to be independent, this is a *VLIW* machine.
- If the four instructions fetched are only executed together if hardware confirms that they are independent, this is an *in-order superscalar* processor.
- If the hardware actively finds four (not necessarily consecutive) instructions that are independent, this is an *out-of-order superscalar* processor.
- What do you think are the tradeoffs?

---

**Superscalar Scheduling**

- Assume in-order, 2-issue, ld-store followed by integer
  - `lw $6, 36($2)`
  - `add $5, $6, $4`
  - `lw $7, 1000($5)`
  - `sub $9, $12, $5`
- Assume 4-issue, any combination (VLIW?)
  - `lw $6, 36($2)`
  - `add $5, $6, $4`
  - `lw $7, 1000($5)`
  - `sub $9, $12, $5`
  - `sw $5, 200($6)`
  - `add $3, $9, $9`
  - `and $11, $7, $6`
- When does each instruction begin execution?
Superscalar vs. superpipelined

(multiple instructions in the same stage, same CR as scalar)

(more total stages, faster clock rate)

Dynamic Scheduling or Out-of-Order Scheduling

- Issues (begins execution of) an instruction as soon as all of its dependences are satisfied, even if prior instructions are stalled.

  lw $6, 36($2)
  add $5, $6, $4
  lw $7, 1000($5)
  sub $9, $12, $8
  sw $5, 200($6)
  add $3, $9, $9
  and $11, $5, $6

Reservation Stations

- are a mechanism to allow dynamic scheduling

PowerPC 604, Pentium Pro (II, III)
Pentium 4

- Deep pipeline
- Dynamically Scheduled (out-of-order scheduling)
- Trace Cache
- Simultaneous Multithreading (HyperThreading)

**Basic Pentium® III Processor Misprediction Pipeline**

<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>Fetch</td>
<td>Fetch</td>
<td>Decode</td>
<td>Decode</td>
<td>Decode</td>
<td>Rename</td>
<td>ROB Rd</td>
<td>Rd/Sch</td>
<td>Dispatch</td>
<td>Exec</td>
</tr>
</tbody>
</table>

**Basic Pentium® 4 Processor Misprediction Pipeline**

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| TC Not IF | TC Fetch | Dec | Sch | Dec | Sch | Dec | Sch | Dec | Sch | Dec | Sch | Dec | Sch | Dec | Sch | Dec | Sch | Dec | Sch |

Modern "Performance" Pipelines

- Pentium II, III – 3-wide (μop) superscalar, out-of-order, 14 integer pipeline stages
- Pentium 4 – 3-wide (μop) superscalar, out-of-order, simultaneous multithreading, 20+ pipe stages (Prescott: 31!)
- Intel Core Duo – 3-wide (μop) out-of-order, 14 integer pipe stages... (*2)
- AMD Opteron (K8), 3-wide (μop) out-of-order, 12 int / 17 FP pipe stages
- Alpha 21264 – 4-wide ss, out-of-order, 7 pipe stages
- Intel Itanium 2 – 3-operation VLIW, 2-instruction issue, in-order, 8-stage
- IBM PowerPC G5 – 4+1-wide ss, out-of-order, 23 pipe stages
- MIPS R12000 – 4-wide ss, out-of-order, 6 integer pipe stages
- Sun UltraSPARC IV+ – 4-wide ss, in-order, 11 int. pipe stages... (*2)

CSE 141, S'06

Jeff Brown

Pipelining -- Key Points

- ET = Number of instructions * CPI * cycle time
- *Data hazards* and *branch hazards* prevent CPI from reaching 1.0, but *forwarding* and *branch prediction* get it pretty close.
- Data hazards and branch hazards need to be detected by hardware.
- Pipeline control uses combinational logic. All data and control signals move together through the pipeline.
- Pipelining attempts to get CPI close to 1. To improve performance we must reduce CT (superpipelining) or CPI below one (superscalar, VLIW).

CSE 141, S'06

Jeff Brown