Outline & Announcements

° Introduction to Hazards
° Forwarding
° 4 cycle Load Delay
° 1 cycle Branch Delay
° What makes pipelining hard

Pipelining – dealing with hazards

° Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  • structural hazards: HW cannot support this combination of instructions
  • data hazards: instruction depends on result of prior instruction still in the pipeline
  • control hazards: pipelining of branches & other instructions that change the PC

° Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline

Single Memory is a Structural Hazard

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Instr 1

Load

Mem

Reg

Instr 2

Mem

Reg

Instr 3

Mem

Reg

Instr 4

Mem

Reg
Option 1: Stall to resolve Memory Structural Hazard

Time (clock cycles)

Load Instr 1 Instr 2 Instr 3 (stall) Instr 4

Data Hazard on r1:

- add r1, r2, r3
- sub r4, r1, r3
- and r6, r1, r7
- or r8, r1, r9
- xor r10, r1, r11

Option 2: Duplicate to Resolve Structural Hazard

- Separate Instruction Cache (Im) & Data Cache (Dm)

Time (clock cycles)

Load Instr 1 Instr 2 Instr 3 Instr 4

Data Hazard on r1:

- Dependencies backwards in time are hazards
**Option 1: HW Stalls to Resolve Data Hazard**

- **add r1, r2, r3**
- **sub r4, r1, r3**
- **and r6, r1, r7**
- **or r8, r1, r9**
- **xor r10, r1, r11**

**Option 2: SW inserts independent instructions**

- **add r1, r2, r3**
- **nop**
- **nop**
- **sub r4, r1, r3**
- **and r6, r1, r7**

But recall how the control logic works:

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later

**Option 1: HW stalls pipeline**

- HW doesn’t change PC => keeps fetching same instruction & sets control signals to benign values (0)

**Option 2: SW inserts independent instructions**

- Worst case inserts NOP instructions
Option 3 Insight: Data is available!

- Pipeline registers already contain needed data

**Key enabler:** Reg file written at beginning of cycle, read at end

```
add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
```

HW Change for “Forwarding” (Bypassing):

- Increase multiplexers to add paths from pipeline registers

```
Forwarding reduces Data Hazard to 1 cycle:
```

Load delays

```
Clock

Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 | Cycle 8

I0: Load  | Fetch | Reg/Dec | Exec | Mem | Wr

Plus 1 | Fetch | Reg/Dec | Exec | Mem | Wr

Plus 2 | Fetch | Reg/Dec | Exec | Mem | Wr

Plus 3 | Fetch | Reg/Dec | Exec | Mem | Wr

Plus 4 | Fetch | Reg/Dec | Exec | Mem | Wr
```

- Although Load is fetched during Cycle 1:
  - Data loaded from memory in cycle 4
  - The data is NOT written into the Reg File until Cycle 5
  - We cannot read this value from the Reg File until Cycle 5
  - 2-instruction delay before the load take effect
**Option 1: HW Stalls to Resolve Data Hazard**

- Check for hazard & stalls

```
lw r1, 0(r2)
stall
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
```

```
IF     ID/RF     EX     MEM     WB
Inst. Order
```

```

*Software Scheduling to Avoid Load Hazards*

Try producing fast code for
\[ a = b + c; \]
\[ d = e - f; \]
assuming \( a, b, c, d, e, \) and \( f \)
in memory.
Slow code:
```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

**Option 2: SW inserts independent instructions**

- Worst case inserts NOP instructions

```
lw r1, 0(r2)
nop
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
```

```
IF     ID/RF     EX     MEM     WB
Inst. Order
```

```

*Software Scheduling to Avoid Load Hazards*

Try producing fast code for
\[ a = b + c; \]
\[ d = e - f; \]
assuming \( a, b, c, d, e, \) and \( f \)
in memory.
Slow code:
```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

```

Try producing fast code for
\[ a = b + c; \]
\[ d = e - f; \]
assuming \( a, b, c, d, e, \) and \( f \)
in memory.
Slow code:
```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

```

Try producing fast code for
\[ a = b + c; \]
\[ d = e - f; \]
assuming \( a, b, c, d, e, \) and \( f \)
in memory.
Slow code:
```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

Try producing fast code for
\[ a = b + c; \]
\[ d = e - f; \]
assuming \( a, b, c, d, e, \) and \( f \)
in memory.
Slow code:
Software Scheduling to Avoid Load Hazards

Try producing fast code for
\[ a = b + c; \]
\[ d = e - f; \]
assuming a, b, c, d, e, and f in memory.

Slow code:
- LW Rb,b
- LW Rc,c
- ADD Ra,Rb,Rc
- SW a,Ra
- LW Re,e
- LW Rf,f
- SUB Rd,Re,Rf
- SW d,Rd

\[ \text{Fast code:} \]
- LW Rb,b
- LW Rc,c
- ADD Ra,Rb,Rc
- SW a,Ra
- LW Re,e
- LW Rf,f
- SUB Rd,Re,Rf
- SW d,Rd

Compiler Avoiding Load Stalls:

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Scheduled</th>
<th>Unscheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>54%</td>
<td>31%</td>
</tr>
<tr>
<td>spice</td>
<td>42%</td>
<td>14%</td>
</tr>
<tr>
<td>tex</td>
<td>65%</td>
<td>25%</td>
</tr>
</tbody>
</table>

% loads stalling pipeline

Branch delay

Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 | Cycle 8 | Cycle 9 | Cycle 10 | Cycle 11
|---------|---------|---------|---------|---------|---------|---------|---------
| Clk     | Ifetch  | Reg/Dec | Exec    | Mem     | Wr      | Ifetch  | Reg/Dec | Exec    | Mem     | Wr      | Ifetch  | Reg/Dec | Exec    | Mem     | Wr      |
| 12: Beq (target is 1000) |
| 16: R-type |
| 20: R-type |
| 24: R-type |
| 1000: Target of Br |

- Although Beq is fetched during Cycle 4:
  - Target address is NOT written into the PC until the end of Cycle 7
  - Branch’s target is NOT fetched until Cycle 8
  - 3-instruction delay before the branch take effect

Branch Stall Impact

° If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
° 2 part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
° MIPS branch tests = 0 or != 0
° Solution Option 1:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch vs. 3
Option 1: move HW forward to reduce branch delay

Option 2: Define Branch as Delayed

° Add instructions after branch that need to execute independent of the branch outcome
  - Worst case, SW inserts NOP into branch delay

° Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch
  - From fall through: only valuable when don’t branch

° Compiler effectiveness for single branch delay slot:
  - Profiling: about 50% of slots usefully filled

Example

° Add r1, r2, r3
° Beq r2, r4, target
° Next

"Branch not depending on add, so swap"

Target: x

Branch prediction

° Aggressive pipelined processors:
  - Place branch resolution as early as possible in pipeline
  - Beyond that, use branch prediction and speculation

° Simple branch prediction:
  - Assume branch not taken, fetch from fall-through
  - If branch is taken, flush pipeline

° More complex techniques are often used:
  - Predict taken or not taken based on learning of past behavior of a branch
    - Keep counters indexed by PC on a “branch predictor table”
  - Predict target address before it is calculated
    - Branch target table, also indexed by PC
Branch prediction

° Speculative execution:
  • Trust, but verify
  • Assume branch prediction is correct, have mechanisms to detect otherwise and flush pipeline before any damage to architectural state is done (i.e. registers or memory get corrupted)

° Example: use the PC to look up a branch predictor table and a branch target table
  • If there is a matching entry for the PC, chances are it is a branch, and chances are the direction (taken/not taken) and target match the prediction
  • Go ahead and set the next PC to be the predicted one
  • Later on in the pipeline, once the branch is resolved (is it a branch? Condition satisfied? What is the target?), either let the instructions that follow it commit, or discard them

Summary – 5-stage pipeline revisited

° Pipeline registers
  • Data and control signals propagate every cycle

° Hazard detection logic and forwarding for data hazards
  • 1 cycle load delay slot, R-type has zero delay

° Move branch resolution to ID stage to reduce delay to 1 cycle
5-stage pipeline revisited

ID/EX.MemRead==1 and
((ID/EX.Rt==IF/ID.Rs) or
(ID/EX.Rt==IF/ID.Rt))

Clear control
Signals for EX, M, WB

Disable writing
PC, IF/ID
register

Forwarding
muxes

Register
Comparison
logic

Hazard
detection

5-stage pipeline revisited

Forwarding
muxes

5-stage pipeline revisited

5-stage pipeline revisited

Examples of other hazards

° “Read-after-write” (RAW)
  • Load followed by ALU instruction using same register
  • Register read must occur after load writes it

° “Write-after-write” (WAW)
  • div.d $f0,$f2,$f4
  • add.d $f0,$f6,$f8
  • add.d’s write must occur after div.d’s

° “Write-after-read” (WAR)
  • div.d $f0,$f2,$f4
  • add.d $f2,$f4,$f6
  • add.d’s write must occur after div.d’s read
When is pipelining hard?

° Interrupts: 5 instructions executing in 5 stage pipeline
  • How to stop the pipeline?
  • Restart?
  • Who caused the interrupt?

<table>
<thead>
<tr>
<th>Stage</th>
<th>Problem interrupts occurring</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory</td>
</tr>
<tr>
<td></td>
<td>access; memory-protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic interrupt</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access;</td>
</tr>
<tr>
<td></td>
<td>memory-protection violation</td>
</tr>
</tbody>
</table>

When is pipelining hard?

° Complex Addressing Modes and Instructions
  • Address modes: Autincrement causes register change during instruction execution
    • Now worry about write hazards since write no longer last stage
    - Write After Read (WAR): Write occurs before independent read
    - Write After Write (WAW): Writes occur in wrong order, leaving wrong result in registers
    - (Previous data hazard called RAW, for Read After Write)
  • Memory-memory Move instructions
    • Multiple page faults

When is pipelining hard?

° Floating Point: long execution time
  • Also, may pipeline FP execution unit so that can initiate new instructions without waiting for full latency

<table>
<thead>
<tr>
<th>FP Instruction</th>
<th>Latency (MIPS R4000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, Subtract</td>
<td>4</td>
</tr>
<tr>
<td>Multiply</td>
<td>8</td>
</tr>
<tr>
<td>Divide</td>
<td>36</td>
</tr>
<tr>
<td>Square root</td>
<td>112</td>
</tr>
<tr>
<td>Negate</td>
<td>2</td>
</tr>
<tr>
<td>Absolute value</td>
<td>2</td>
</tr>
<tr>
<td>FP compare</td>
<td>3</td>
</tr>
</tbody>
</table>

° Divide, Square Root take -10X to -30X longer than Add
  • Exceptions?
  • Adds WAR and WAW hazards since pipelines are no longer same length

First Generation RISC Pipelines ("Scalar")

° All instructions follow same pipeline order ("static schedule").
° Register write in last stage
  – Avoid WAW hazards
° All register reads performed in first stage after issue.
  – Avoid WAR hazards
° Memory access in stage 4
  – Avoid all memory hazards
° Control hazards resolved by delayed branch (with fast path)
° RAW hazards resolved by bypass, except on load results which are resolved by delayed load.

Substantial pipelining with very little cost or complexity.
Machine organization is (slightly) exposed!
Examples

° Alpha 21064 (92):
  • up to two instructions per cycle
  • One floating-point, one integer (in-order)
  • 7 stages (int), 10 stages (FP)

° MIPS R3000 (88)
  • One (integer) instruction per cycle
  • 5 stages (int)

° Sparc Micro (91)
  • 5 stages

Examples

° Alpha 21264 (98)
  • up to 4 instructions per cycle
  • 7 stages (int), 10 stages (FP)

° MIPS R10000 (96)
  • 4 instruction per cycle
  • 5 stages (int), 10 stages (FP)

° Sparc Ultra II (96)
  • 9 stages (int, FP)
  • 4 instructions issued per cycle

Today’s RISC Pipelines (“Superscalar”)

° Instructions can be issued out of order in pipeline (“dynamic schedule”)
  – Must handle WAW, WAR hazards in addition to RAW
  – Tomasulo, Scoreboarding techniques
° Multiple instructions issued in a single cycle
  • Instructions are “queued up” for execution in a reorder buffer
  • CPIeffective < 1!
° Control hazards resolved (speculatively) by predicting branches
° Single-cycle memory access in best case (cache hit)
  Tens-hundreds if need to go to main memory
° Aggressive pipelining with rapidly increasing cost/complexity.
° Diminishing returns as more resources are added

NetBurst

° Successor to Pentium Pro
  • 3 uops per cycle, out-of-order
° Key differences
  • Deeper pipeline for fast clocks: 20 stages
  • Seven integer execution units vs. 5
  • Can overlap instructions from two programs in the pipeline
    - “Hyper-threading”; simultaneous multi-threading
    - To software, looks as if it has 2 processors
Review: Summary of Pipelining Basics

- Speed Up proportional to pipeline depth; if ideal CPI is 1, then:
  \[
  \text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}} \cdot \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}
  \]

- Hazards limit performance on computers:
  - structural: need more HW resources
  - data: need forwarding, compiler scheduling
  - control: early evaluation & PC, delayed branch, prediction

- Increasing length of pipe increases impact of hazards since pipelining helps instruction bandwidth, not latency

- Compilers key to reducing cost of data and control hazards
  - load delay slots
  - branch delay slots

- Exceptions, Instruction Set, FP makes pipelining harder