Readings

H+P

• Appendix A, Chapter 2.3
• This will be partly review for those who took ECE 152

Recent Research Paper

• “The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays”, Hrishikesh et al., ISCA 2002.
Getting More Performance

Execution of each instruction involves 5 activities
  • Fetch, Decode, Execute, Memory Access, Writeback

How can we improve performance?
  • Latency/instruction? Tough to do
  • Instruction throughput? More opportunities here!

Key to improving throughput: parallelism

What kinds of parallelism can we exploit?
Instruction Level Parallelism (ILP)

ILP is a property of the software (not the hardware)

- how much parallelism exists among instructions?
- varies greatly across programs

many possible ways to exploit ILP

- pipelining: overlap processing of instructions
- superscalar: multiple instructions at a time
- out-of-order execution: dynamic (hardware) scheduling
- compiler scheduling of code: static (software) scheduling
ILP Example

```plaintext
add r1, r2, r3    # r1 = r2 + r3
sub r4, r1, r2
mul r5, r1, r4
xor r6, r2, r2
and r7, r6, r1
add r8, r3, r3
```

On a “perfectly parallel” machine, how many cycles would this code snippet take?

- Assume that all operations take 1 cycle
What is Limit of ILP?

H&P Chapter 3 focuses on this issue

Two important performance limiters
  – Limited ILP - why?
  – Inability to exploit all available ILP - why?

What kinds of software have more/less ILP?

We’ll now talk about one way to exploit ILP: pipelining
Basic Pipelined Processor

- basic = single, in-order issue
  - single issue = one instruction at a time (per stage)
  - in-order issue = instructions (start to) execute in order
  - next units: multiple issue, 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

- pipelining meets the x86 ISA
Pipelining

observe: instruction processing consists of $N$ sequential stages

idea: overlap different instructions at different stages

| non-pipelined | inst0.1 | inst0.2 | inst0.3 | inst1.1 | inst1.2 | inst1.3 |
| pipelined     | inst0.1 | inst0.2 | inst0.3 | inst1.1 | inst1.2 | inst1.3 |

+ 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 D$/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 ld) 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], immem, rd
  - X/M: opcode, regfile[rs1], ALUOUT, rd
  - M/W: ALUOUT, MEMOUT, rd
## Pipeline Diagram

Compared to non-pipelined case:

- Better throughput: an instruction finishes every cycle
- Same latency per instruction: each still takes 5 cycles

<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>
</tr>
</thead>
<tbody>
<tr>
<td>inst0</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>inst1</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>inst2</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>inst3</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2009 by Sorin, Roth, Hill, Wood, Sohi, Smith, Vijaykumar, Lipasti
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 = \( 1/\max(t_n) \leq N/T \) (\( \max(t_n) \) 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 \( T = \Sigma (V + t_n) = N*V + \Sigma t_n \)
- 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
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 \Rightarrow 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) * \text{latency} > \text{latency}
  - throughput’ \Rightarrow N^2/T(N+S) = (N/(N+S)) * \text{throughput} < \text{throughput}
  - M\_\text{latency’} \Rightarrow M*T(N+S)/N^2 = ((N+S)/N) * M\_\text{latency} > M\_\text{latency}
  - speedup’ \Rightarrow N^2/(N+S) = (N/(N+S)) * \text{speedup} < \text{speedup}
Pipelining: Clock Rate vs. IPC

deep 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 ...
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]
  - CPI = \(1 + S\times N\) --> IPC = \(1 / (1 + S\times N)\)
  - clock rate (in gate-delays) = \(1 / (\text{gate delays/stage}) = 1 / (G/N + V)\)
  - 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>
Pipeline Depth Upshot

trend is (was?) 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), later 31 stages
  - 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)

```
   | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 |
---|----|----|----|----|----|----|----|----|----|----|----|----|----|
load| F  | D  | X  | M  | W  |    |    |    |    |    |    |    |    |
inst2| F  | D  | X  | M  | W  |    |    |    |    |    |    |    |    |
inst3| F  | D  | X  | M  | W  |    |    |    |    |    |    |    |    |
inst3| F  | D  | X  | M  | W  |    |    |    |    |    |    |    |    |
```
Fixing Structural Hazards

• fix structural hazard by stalling ($s^*$ = 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>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>$s^*$</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</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

\[
\begin{align*}
\text{add } & R1, R2, R3 & \text{add } & R1, R2, R3 & \text{add } & R1, R2, R3 \\
\text{sub } & R2, R4, R1 & \text{sub } & R2, R4, R1 & \text{sub } & R2, R4, R1 \\
\text{or } & R1, R6, R3 & \text{or } & R1, R6, R3 & \text{or } & R1, R6, R3 \\
\end{align*}
\]

- read-after-write (RAW)
- write-after-read (WAR)
- write-after-write (WAW)

true dependence (real)
anti-dependence (artificial)
output dependence (artificial)

Q: What about read-after-read dependences? (RAR)
read-after-write (RAW) = true dependence (dataflow)

• 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 writing of PC and F/D latch

• Option#1 for 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

• Option#2 for 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

− poor performance (many stalls)
Two Stall Timings

depends 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 ⇒ 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>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>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>load R5,R6,R7</td>
<td>p*</td>
<td>p*</td>
<td>p*</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td></td>
</tr>
</tbody>
</table>

- 1st half WB writes, 2nd half ID reads ⇒ 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>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>sub R2,R4,R1</td>
<td>F</td>
<td>d*</td>
<td>d*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
</tbody>
</table>
### Stall Signal Example (2nd Timing)

<table>
<thead>
<tr>
<th>PC</th>
<th>F/D</th>
<th>D/X</th>
<th>X/M</th>
<th>M/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>↑write disable</td>
<td>↑write disable</td>
<td>↑clear</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Load r6, 0(r4)**
- **Add r4, r2, r1**
- **Load r2, 0(r3)**
- **Add r5, r5, #4**
- **Call func**
- **Load r6, 0(r4)**
- **Add r4, r2, r1**
- **Load r6, 0(r4)**
- **Add r4, r2, r1**
- **Load r6, 0(r4)**
- **Add r4, r2, r1**

**C1:** rs1(D) == rd(D/X) ⇒ stall

**C2:** rs1(D) == rd(X/M) ⇒ stall

**C3:** rs1(D) == rd(M/W) ⇒ go
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 use ALUOUT(M)
  - else if rs2(X) == rd(M/W), then use 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>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>
</tr>
<tr>
<td><strong>sub R2, R4, R1</strong></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>
</tr>
</tbody>
</table>

**example 1**

<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>load R1, 24(R5)</strong></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>
</tr>
<tr>
<td><strong>add R3, R6, R7</strong></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>
</tr>
<tr>
<td><strong>sub R2, R4, R1</strong></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>
</tr>
</tbody>
</table>

**example 2**

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

<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>load R1, 24(R5)</strong></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>
</tr>
<tr>
<td><strong>sub R2, R4, R1</strong></td>
<td>F</td>
<td>D</td>
<td>d*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**example 3**
Pipeline Scheduling

compiler schedules (moves) instructions to reduce stall

• eliminate back-to-back load-ALU scenarios
• example code sequence  \( a = b + c; \ d = e - f \)

before

```assembly
load R2, b
load R3, c
add R1, R2, R3 // stall
store R1, a
load R5, e
load R6, f
sub R4, R5, R6 // stall
store R4, d
```

after

```assembly
load R2, b
load R3, c
load R5, e
load R6, f
add R1, R2, R3 // no stall
load R1, a
sub R4, R5, R6 // no stall
store R4, d
```

© 2009 by Sorin, Roth, Hill, Wood, Sohi, Smith, Vijaykumar, Lipasti