Final QuickFire Review

Fall 2012
CS 465
Huzefa Rangwala, PhD

Chapters 1 - 6
Disclaimer:
These slides are not a substitute for reading the book.

To do well: Go over the detailed readings, solve exercises from HW, Quizzes, Classroom Exercises and back of the book*. 
Detailed Readings

- Look on piazza or class web-site
How to Design a Processor

- Analyze instruction set $\Rightarrow$ datapath requirements
  - Meaning of each instruction given by register transfers
  - Datapath must include storage element for ISA registers (possibly more)
  - Datapath must support each register transfer
- Select set of datapath components and establish clocking methodology
- Assemble datapath meeting the requirements
- Analyze implementation of each instruction to determine setting of control points that effects the register transfer
- Assemble the control logic
Datapath for Instruction Fetch

- PC
- Read address
- Instruction memory
- Instruction
- Add
- 4
Datapath for R-format Instructions

Instruction

Read register 1
Read register 2
Write register
Write data

Read data 1
Read data 2

4 ALU operation

ALU

ALU result

Zero

RegWrite
Datapath for LW & SW (Fix this)

- $R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[imm16]]$ Example: lw $rt, rs, imm16$
- Mem[ $R[rs] + \text{SignExt}[imm16] \leftarrow R[rt] \]$ Example: sw $rt, rs, imm16$
Branch Instructions

Just re-routes wires

PC + 4 from instruction datapath

Add Sum

Shift left 2

Branch target

4
ALU operation

ALU Zero

To branch control logic

Instruction

Read register 1
Read register 2
Write register
Write data

Registers

Read data 1
Read data 2

RegWrite

Sign-extend

16 32

Sign-bit wire replicated
Full Datapath
Branch-on-Equal Instruction
Implementing Jumps

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode
Delays in Single Cycle Datapath

What are the delays for lw, sw, R-Type, beq, j instructions?
## Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>Instruction Fetch</th>
<th>Register Access</th>
<th>ALU</th>
<th>Register/Memory Access</th>
<th>Register Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Type</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>R</td>
<td>6</td>
</tr>
<tr>
<td>Load</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td>X</td>
</tr>
<tr>
<td>Store</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipeline Performance

**Single-cycle \( T_c = 800 \text{ps} \)**

Program execution order (in instructions)

```
lw $1, 100(0)
lw $2, 200(0)
lw $3, 300(0)
```

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

Time

```
200 400 600 800 1000 1200 1400 1600 1800
```

800 ps

**Pipelined \( T_c = 200 \text{ps} \)**

Program execution order (in instructions)

```
lw $1, 100(0)
lw $2, 200(0)
lw $3, 300(0)
```

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

Time

```
200 400 600 800 1000 1200 1400
```

200 ps

800 ps

200 ps
Hazards

- Situations that prevent starting the next instruction in the next cycle
  - _______hazards
    - A required resource is busy
  - ______hazard
    - Need to wait for previous instruction to complete its data read/write
  - ______hazard
    - Deciding on control action depends on previous instruction
Structural Hazard: One Memory

- Hazards can always be resolved by waiting
Data Hazard Example

- Dependences backward in time are hazards

\[
\begin{align*}
\text{add} & \quad r_1, r_2, r_3 \\
\text{sub} & \quad r_4, r_1, r_3 \\
\text{and} & \quad r_6, r_1, r_7 \\
\text{or} & \quad r_8, r_1, r_9 \\
\text{xor} & \quad r_{10}, r_1, r_{11}
\end{align*}
\]

- Compilers can help, but it gets messy and difficult
Data Hazard Solution

- Solution: “forward” result from one stage to another
Data Hazard Even with Forwarding

- Can you resolve this?

lw  r1,0(r2)
sub  r4,r1,r3
Data Hazard Even with Forwarding

- Must delay/stall instruction dependent on loads
- Sometimes the instruction sequence can be reordered to avoid pipeline stalls
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch

- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage
**Stall on Branch**

- Wait until branch outcome determined before fetching next instruction

---

Program execution order (in instructions)

- add $4, $5, $6
- beq $1, $2, 40
- or $7, $8, $9
MIPS with Predict Not Taken

**Prediction correct**

Program execution order (in instructions)

- add $4, $5, $6
- beq $1, $2, 40
- lw $3, 300($0)

**Prediction incorrect**

Program execution order (in instructions)

- add $4, $5, $6
- beq $1, $2, 40
- lw $3, 300($0)

- or $7, $8, $9
- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Instruction fetch
- Reg
- ALU
- Data access
- Reg
MIPS Pipelined Datapath

Right-to-left flow leads to hazards
Pipeline registers

Need registers between stages (Why ?)
Multi-Cycle Pipeline Diagram

Form showing resource usage

Program execution order (in instructions)

Lw $10, 20($1)

Sub $11, $2, $3

Add $12, $3, $4

Lw $13, 24($1)

Add $14, $5, $6
Multi-Cycle Pipeline Diagram

Traditional form

Program execution order (in instructions)

lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages $\Rightarrow$ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
      - But dependencies reduce this in practice
MIPS with Static Dual Issue

Two-issue packets
- One ALU/branch instruction
- One load/store instruction
- 64-bit aligned
  - ALU/branch, then load/store
  - Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF</td>
</tr>
</tbody>
</table>
Loop Unrolling Example

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s1, $s1, -16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>lw $t3, 4($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addu $t2, $t2, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>5</td>
</tr>
<tr>
<td>addu $t3, $t4, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>6</td>
</tr>
<tr>
<td>nop</td>
<td>sw $t2, 8($s1)</td>
<td>7</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td>8</td>
</tr>
</tbody>
</table>

IPC = 14/8 = 1.75

- Closer to 2, but at cost of registers and code size
Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU
Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we’d like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well
Concluding Remarks for Processors

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall
QUESTIONS
What is the difference between a conflict miss and a compulsory miss? How would you reduce each type?
What are two different strategies for dealing with cache writes? What is an advantage and disadvantage of each type?
• Why might we want more than one level of a cache?
Suppose a processor has a CPI of 3.0 given a perfect cache. If there are 1.4 memory accesses per instruction, a miss penalty of 20 cycles, and a miss rate of 5%, what is the effective CPI with the real cache?
What are two advantages of using virtual memory?
• What is a TLB? Why do we need it?

• Design a cache controller.
How does pipelining improve performance?
What is multiple issue? Is this the same as VLIW?
Why is branch prediction so important? Does multiple issue increase or decrease the need for such prediction?
• Explain the stored-program concept.
For a data cache with a 92% hit rate and a 2-cycle hit latency, calculate the average memory access latency (AMAT). Assume that the latency to memory and a cache miss penalty together is 124 clock cycles.

\[
AMAT = 2 + 0.08 \times 124 \\
= 11.92 \text{ cycles}
\]
(5 pts) A compiler designer is trying to decide between two code segments for a particular machine. The hardware designers have provided the following data below about the CPI for each class, and the instruction counts being considered for each code sequence.

<table>
<thead>
<tr>
<th>Class</th>
<th>CPI for this instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code sequence</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

How many cycles are required for each code sequence?
- Code sequence #1:
- Code sequence #2:

Which is faster and how by how much?

What is the CPI for each code sequence?
- CPI for code sequence #1:
- CPI for code sequence #2: