CS 61C:
Great Ideas in Computer Architecture
Control and Pipelining
Instructor:
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp12

## Agenda
- Control Implementation
- Administrivia
- Vtune Demo (Rimas Avizenis)
- Pipelined Execution
- Pipelined Datapath
- (Pipeline Hazards)
- Summary

## Review: Single-Cycle Processor
- Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements; re-examine for pipelining
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits

## Given Datapath: RTL → Control
- Processor
- Input
- Output
- Memory

## Levels of Representation/Interpretation
- High Level Language Program (e.g., C)
- Assembly Language Program (e.g., MIPS)
- Machine Language Program (MIPS)

- Anything can be represented as a number, i.e., data or instructions

## You Are Here!
- Smart Phone
- Warehouse
- Computer
- Software
- Harness Scalability & Achieve High Performance
- Memory (Cache)
- Logic Gates
- Core
- Main Memory
- Instruction Unit
- Functional Unit
- Today’s Lecture

## Review:
- Single-Cycle Processor
- Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements: re-examine for pipelining
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits

- Levels of Representation/Interpretation
- High Level Language Program (e.g., C)
- Assembly Language Program (e.g., MIPS)
- Machine Language Program (MIPS)

- Anything can be represented as a number, i.e., data or instructions

- Agenda
- Control Implementation
- Administrivia
- Vtune Demo (Rimas Avizenis)
- Pipelined Execution
- Pipelined Datapath
- (Pipeline Hazards)
- Summary

- Given Datapath: RTL → Control
- Processor
- Input
- Output
- Memory
Summary of the Control Signals (1/2)

<table>
<thead>
<tr>
<th>Bus</th>
<th>Register Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>rtype = r[add] + r[add]; PC → PC + 4</td>
</tr>
<tr>
<td>sub</td>
<td>r[add] = r[add]; PC → PC + 4</td>
</tr>
<tr>
<td>ori</td>
<td>r[add] = r[add] + sign_ext(Imm16); PC → PC + 4</td>
</tr>
<tr>
<td>lw</td>
<td>r[add] = MEM[rs] + sign_ext(Imm16); PC → PC + 4</td>
</tr>
<tr>
<td>sw</td>
<td>r[add] = MEM[rs] + zero_ext(Imm16); PC → PC + 4</td>
</tr>
<tr>
<td>jump</td>
<td>ALUctr = or</td>
</tr>
<tr>
<td>beq</td>
<td>ALUctr = or</td>
</tr>
<tr>
<td>ori = ~op</td>
<td></td>
</tr>
<tr>
<td>rtype = ~op</td>
<td></td>
</tr>
</tbody>
</table>

Where:
- \( ALUctr[1] = \text{or} \)
- \( ALUctr[0] = \text{add} + \text{beq} \) (assume \( ALUctr = 00 \) ADD, 01: SUB, 10: OR)

ExtOp = lw + sw

Jump = jump

nPCsel = beq

MemWrite = sw

RegWrite = add + sub + ori + lw

MemtoReg = lw

ALUSrc = ori + lw + sw

RegDst = add + sub

Boolean Expressions for Controller

RegOut = add + sub

Add = ori + lw + sw

MemWrite = lw

RegWrite = add + sub + ori + lw

Jump = jump

ExtOp = lw + sw

ALUOut[0] = sub + beq (assume \( ALUctr = 00 \) ADD, 01: SUB, 10: OR)

ALUOut[1] = or

If beq if \( \{r[rs] = r[rt]\} \) then PC = PC + 4 else PC + 4

nPC_sel = beq, ALUctr = "SUB"

Summary of the Control Signals (2/2)

See Appendix A

Controller Implementation

<table>
<thead>
<tr>
<th>opcode</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td></td>
</tr>
<tr>
<td>ori</td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
</tr>
<tr>
<td>jump</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
</tr>
</tbody>
</table>

Summary of the Control Signals (2/2)

See Appendix A

We Don’t Care :-)

R-type

I-type

Add

Sub

Lw

Sw

Beq

OR Logic

AND Logic

Controller Implementation

Logisim

AND Control in Logisim

OR Control Logic in Logisim
Single Cycle Performance

- Assume time for actions are
  - 100ps for register read or write; 200ps for other events
- Clock rate is?

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>600ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td>700ps</td>
<td></td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>100ps</td>
<td>600ps</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td></td>
<td>500ps</td>
<td></td>
</tr>
</tbody>
</table>

- What can we do to improve clock rate?
- Will this improve performance as well?
  Want increased clock rate to mean faster programs

Getting To Know Your Prof

- Decided when got PhD to try to maximize happiness vs. wealth
  - Family
  - A job you enjoy
  - Friends
  - Helping others
  - Playing?

Getting to Know Your Prof

- Play soccer on Sunday mornings for >30 years
- After college, can forget it’s important to find time to play (If want to be happy)

Pipeline Analogy: Doing Laundry

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - “Folder” takes 30 minutes
  - “Stasher” takes 30 minutes to put clothes into drawers
A simpler core is a faster core.

Reduction in the number and complexity of instructions in the ISA simplifies pipelined implementation.

Common RISC strategies:
- Fixed instruction length, generally a single word (MIPS = 32b).
- Simplifies process of fetching instructions from memory.
- Simplified addressing modes (MIPS just register + offset).
- Fewer and simpler instructions in the instruction set.
- Simplifies process of executing instructions.
- Simplified memory access: only load and store instructions access memory.
- Let the compiler do it: Use a good compiler to break complex high-level language statements into a number of simple assembly language statements.
Steps in Executing MIPS
1) **IF**: Instruction Fetch, Increment PC
2) **ID**: Instruction Decode, Read Registers
3) **EX**: Execution
   Mem-ref: Calculate Address
   Arith-log: Perform Operation
4) **Mem**: Load: Read Data from Memory
   Store: Write Data to Memory
5) **WB**: Write Data Back to Register

Redrawn Single-Cycle Datapath

Pipelined Datapath

• Add registers between stages
  – Hold information produced in previous cycle
• 5 stage pipeline; clock rate potential 5X faster

More Detailed Pipeline

ID for Load, Store, ...

IF for Load, Store, ...
Every instruction must take the same number of steps, also called pipeline stages, so some will go idle sometimes.

**Pipelined Execution Representation**

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
</tr>
<tr>
<td>IF</td>
</tr>
<tr>
<td>IF</td>
</tr>
<tr>
<td>IF</td>
</tr>
</tbody>
</table>

**Graphical Pipeline Diagrams**

- Use the datapath figure below to represent pipeline stages.
Graphical Pipeline Representation
(In Reg, right half highlight read, left half write)
Time (clock cycles)

Pipeline Performance
• Assume time for stages is
  – 100ps for register read or write
  – 200ps for other stages
• What is pipelined clock rate?
  – Compare pipelined datapath with single-cycle datapath

Pipeline Speedup
• If all stages are balanced
  – i.e., all take the same time
  – Time between instructions\_\text{pipelined} = \frac{\text{Time between instructions}_{\text{non-pipelined}}}{\text{Number of stages}}
• If not balanced, speedup is less
• Speedup due to increased throughput
  – Latency (time for each instruction) does not decrease

Instruction Level Parallelism (ILP)
• Another parallelism form to go with Request Level Parallelism and Data Level Parallelism
  – RLP – e.g., Warehouse Scale Computing
  – DLP – e.g., SIMD, Map-Reduce
• ILP – e.g., Pipelined Instruction Execution
  – 5 stage pipeline \Rightarrow 5 instructions executing simultaneously, one at each pipeline stage

Agenda
• Control Implementation
• Administrivia
• Vtune Demo (Rimas Avizenis)
• Pipelined Execution
• Pipelined Datapath
• (Pipeline Hazards)
• Summary
Hazards

Situations that prevent starting the next logical instruction in the next clock cycle

1. Structural hazards
   - Required resource is busy (e.g., stasher is studying)
2. Data hazard
   - Need to wait for previous instruction to complete its data read/write (e.g., pair of socks in different loads)
3. Control hazard
   - Deciding on control action depends on previous instruction (e.g., how much detergent based on how clean prior load turns out)

1. Structural Hazard #1: Single Memory

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr 1</td>
</tr>
<tr>
<td>Load</td>
</tr>
<tr>
<td>I$</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Read same memory twice in one clock cycle

1. Structural Hazard #2: Registers (1/2)

Can we read and write to registers simultaneously?

1. Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) RegFile access is VERY fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle
  2) Build RegFile with independent read and write ports
     - Result: can perform Read and Write during same clock cycle

2. Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add $s0, $t0, $t1
  - sub $t2, $s0, $t3
Forwarding (aka Bypassing)

• Use result when it is computed
  – Don’t wait for it to be stored in a register
  – Requires extra connections in the datapath

Corrected Datapath for Forwarding?

Forwarding Paths

Load-Use Data Hazard

• Can’t always avoid stalls by forwarding
  – If value not computed when needed
  – Can’t forward backward in time!

Stall/Bubble in the Pipeline

Pipelining and ISA Design

• MIPS Instruction Set designed for pipelining
• All instructions are 32-bits
  – Easier to fetch and decode in one cycle
  – x86: 1- to 17-byte instructions
    (x86 HW actually translates to internal RISC instructions!)
• Few and regular instruction formats, 2 source register fields always in same place
  – Can decode and read registers in one step
• Memory operands only in Loads and Stores
  – Can calculate address 3rd stage, access memory 4th stage
• Alignment of memory operands
  – Memory access takes only one cycle
Why Isn’t the Destination Register Always in the Same Field in MIPS ISA?

• Need to have 2 part immediate if 2 sources and 1 destination always in same place

3/22/12

Spring 2012 - Lecture #20

3. 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
• BEQ, BNE in MIPS pipeline
• Simple solution Option 1: **Stall** on every branch until have new PC value
  – Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)

3. Control Hazard: Branching

• Optimization #1:
  – Insert special branch comparator in Stage 2
  – As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  – Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  – Side Note: means that branches are idle in Stages 3, 4 and 5

Corrected Datapath for BEQ/BNE?

One Clock Cycle Stall

Branch comparator moved to Decode stage.
3. Control Hazards

- Option 2: Predict outcome of a branch, fix up if guess wrong
  - Must cancel all instructions in pipeline that depended on guess that was wrong
- Simplest hardware if we predict that all branches are NOT taken
  - Why?

3. Control Hazard: Branching

- Option #3: Redefine branches
  - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (the branch-delay slot)
  - Delayed Branch means we always execute inst after branch
  - This optimization is used with MIPS

3. Control Hazard: Branching

- Notes on Branch-Delay Slot
  - Worst-Case Scenario: put a no-op in the branch-delay slot
  - Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn’t affect the logic of program
    - Re-ordering instructions is common way to speed up programs
    - Compiler usually finds such an instruction 50% of time
    - Jumps also have a delay slot ...

Example: Nondelayed vs. Delayed Branch

Nondelayed Branch
- or $8, $9, $10
- add $1, $2, $3
- sub $4, $5, $6
- beq $1, $4, Exit
- xor $10, $1, $11

Delayed Branch
- add $1, $2, $3
- sub $4, $5, $6
- beq $1, $4, Exit
- or $8, $9, $10
- xor $10, $1, $11

Delayed Branch/Jump and MIPS ISA?

- Why does JAL put PC+8 in register 31?

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F;

```
lw $t1, 0($t0)       lw $t1, 0($t0)
lw $t2, 4($t0)       lw $t2, 4($t0)
sw $t3, 12($t0)      add $t3, $t2, $t1       sw $t3, 12($t0)
sw $t4, 16($t0)      add $t4, $t1, $t3      sw $t5, 16($t0)
```

<table>
<thead>
<tr>
<th>Stall</th>
<th>11 cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exit: | Exit:
I. Thanks to pipelining, I have reduced the time it took me to wash my one shirt.

II. Longer pipelines are always a win (since less work per stage & a faster clock).

A) (orange) I is True and II is True
B) (green) I is False and II is True
C) (pink) I is True and II is False
D) (yellow) I is False and II is False

And in Conclusion, ...

The BIG Picture

- Pipelining improves performance by increasing instruction throughput: exploits ILP
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

In Conclusion, …