CS 61C:  
Great Ideas in Computer Architecture  
Control and Pipelining  

Instructor:  
Randy H. Katz  
http://inst.eecs.Berkeley.edu/~cs61c/fa13  

You Are Here!  

- **Parallel Requests**  
  Assigned to computer  
  e.g., Search “Katz”  

- **Parallel Threads**  
  Assigned to core  
  e.g., Lookup, Ads  

- **Parallel Instructions**  
  >1 instruction @ one time  
  e.g., 5 pipelined instructions  

- **Parallel Data**  
  >1 data item @ one time  
  e.g., Add of 4 pairs of words  

- **Hardware descriptions**  
  All gates @ one time  

- **Programming Languages**
Levels of Representation/Interpretation

- High Level Language Program (e.g., C)
  - Compiler
- Assembly Language Program (e.g., MIPS)
  - Assembler
- Machine Language Program (MIPS)

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 => 5 instructions executing simultaneously, one at each pipeline stage

```plaintext
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

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

```
0000 1001 1100 0110 1010 1111 0101 1000 1010 1111 0101 1000 0000 1001 1010 1111 0110 0110 1111 0101 1000 0000 1001 0101 1000 0000 1001 1100 0110 1010 1111 0101 1000 0000 1001
```

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)
Agenda

- Pipelined Execution
- Pipelined Datapath
- Structural and Data Hazards
- Control Hazards
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 affects the register transfer.
  5. Assemble the control logic
     • Formulate Logic Equations
     • Design Circuits

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
Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!
Pipelining Lessons (1/2)

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages (4 in this case)
- Time to fill pipeline and time to drain it reduces speedup: 8 hours/3.5 hours or 2.3X v. potential 4X in this example

Pipelining Lessons (2/2)

- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
Agenda

- Pipelined Execution
- Pipelined Datapath
- Structural and Data Hazards
- Control Hazards

Review: RISC Design Principles

- “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)
    Simplifies process of fetching operands from memory
  - 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
Review: Single Cycle Datapath

- Data Memory \( R[rs] + \text{SignExt}[\text{imm16}] \) = \( R[rt] \)

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

1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Write Back

Pipelined Datapath

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

Registers named for adjacent stages, e.g., IF/ID

IF for Load, Store, ...

Highlight combinational logic components used + right half of state logic on read, left half on write
ID for Load, Store, ...

EX for Load
MEM for Load

WB for Load

Has Bug that was in 1st edition of textbook!
Corrected Datapath for Load

Pipelined Execution Representation

<table>
<thead>
<tr>
<th>Time</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Every instruction must take the same number of steps, also called pipeline stages, so some will go idle sometimes
Graphical Pipeline Diagrams

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back

- Use datapath figure below to represent pipeline

Graphical Pipeline Representation
(In Reg, right half highlight read, left half write)

Time (clock cycles)

Load, Add, Store, Sub, Or
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

<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>200ps</td>
<td>100ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td></td>
<td>100ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

Pipeline Performance

Single-cycle ($T_c = 800ps$)

Pipelined ($T_c = 200ps$)
Pipeline Speedup

• If all stages are balanced
  — i.e., all take the same time
  — \( \text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}} \)

• If not balanced, speedup is less
• Speedup due to increased throughput
  — Latency (time for each instruction) does not decrease

Agenda

• Pipelined Execution
• Pipelined Datapath
• Structural and Data Hazards
• Control Hazards
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 Hazards

• Conflict for use of a resource
• In MIPS pipeline with a single memory
  – Load/Store requires memory access for data
  – Instruction fetch would have to stall for that cycle
    • Causes a pipeline “bubble”
• Hence, pipelined datapaths require separate instruction/data memories
  – In reality, provide separate L1 instruction cache and L1 data cache
1. Structural Hazard #1: Single Memory

Time (clock cycles)

Instr Order
Load Instr 1 Instr 2 Instr 3 Instr 4

Read same memory twice in same clock cycle

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

Time (clock cycles)

Instr Order
sw Instr 1 Instr 2 Instr 3 Instr 4

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

```plaintext
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?
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!
Chapter 4 — The Processor — 43

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 3\textsuperscript{rd} stage, access memory 4\textsuperscript{th} 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. 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.
Agenda

- Pipelined Execution
- Pipelined Datapath
- Structural and Data Hazards
- Control Hazards

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 $t1, 0($t0)
lw $t2, 4($t0)   lw $t2, 4($t0)   lw $t2, 4($t0)
add $t3, $t1, $t2 add $t3, $t1, $t2 add $t3, $t1, $t2
sw $t3, 12($t0)  sw $t3, 12($t0)  sw $t3, 12($t0)
lw $t4, 8($t0)   lw $t4, 8($t0)   lw $t4, 8($t0)
add $t5, $t1, $t4 add $t5, $t1, $t4 add $t5, $t1, $t4
sw $t5, 16($t0)  sw $t5, 16($t0)  sw $t5, 16($t0)
```

Peer Instruction

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, ...

- Pipelining improves performance by increasing instruction throughput: exploits ILP
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
  - Key enabler is placing registers between pipeline stages
- 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