Pipelining concepts, datapath and hazards
**Boolean Exprs for Controller**

<table>
<thead>
<tr>
<th>Inst Memory</th>
<th>Adr</th>
<th>Instruction&lt;31:0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Op 0-5 are really Instruction bits 26-31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Func 0-5 are really Instruction bits 0-5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>rtype</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>~op_5 • ~op_4 • ~op_3 • ~op_2 • ~op_1 • ~op_0,</td>
<td>~op_5 • ~op_4 • op_3 • op_2 • ~op_1 • op_0</td>
<td>op_5 • ~op_4 • ~op_3 • ~op_2 • op_1 • op_0</td>
<td>op_5 • ~op_4 • op_3 • ~op_2 • op_1 • op_0</td>
<td>~op_5 • ~op_4 • ~op_3 • op_2 • ~op_1 • ~op_0</td>
<td>~op_5 • ~op_4 • ~op_3 • ~op_2 • ~op_1 • ~op_0</td>
</tr>
</tbody>
</table>

add = rtype • func_5 • ~func_4 • ~func_3 • ~func_2 • ~func_1 • ~func_0
sub = rtype • func_5 • ~func_4 • ~func_3 • ~func_2 • func_1 • ~func_0

How do we implement this in gates?

Dr Dan Garcia
Boolean Exprs for Controller

RegDst = add + sub
ALUSrc = ori + lw + sw
MemtoReg = lw
RegWrite = add + sub + ori + lw
MemWrite = sw
nPCsel = beq
Jump = jump
ExtOp = lw + sw
ALUctr[0] = sub + beq
ALUctr[1] = ori

(assume ALUctr is 00 ADD, 01 SUB, 10 OR)

How do we implement this in gates?
Controller Implementation

“AND” logic:
- opcode
- func
- add
- sub
- ori
- lw
- sw
- beq
- jump

“OR” logic:
- RegDst
- ALUSrc
- MemtoReg
- RegWrite
- MemWrite
- nPCsel
- Jump
- ExtOp
- ALUctr[0]
- ALUctr[1]
Call home, we’ve made HW/SW contact!

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

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

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

```
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)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```
Review: Single-cycle Processor

• Five steps to design a processor:
  1. Analyze instruction set \(\rightarrow\) datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  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
Agenda

- Pipelining Performance
- Structural Hazards
- Data Hazards
  - Forwarding
  - Load Delay Slot
- Control Hazards
Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining
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>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</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
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>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</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
Gotta Do 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 takes 8 hours for 4 loads
Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!
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.

Time to “fill” pipeline and time to “drain” it reduces speedup: 2.3X v. 4X in this example.
• 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
Steps in Executing MIPS

1) **IFetch**: Instruction Fetch, Increment PC
2) **Dcd**: Instruction Decode, Read Registers
3) **Exec**:
   - 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
Single Cycle Datapath

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

- Need registers between stages
  - To hold information produced in previous cycle
More Detailed Pipeline
IF for Load, Store, ...
ID for Load, Store, ...
EX for Load
MEM for Load
WB for Load – Oops!

Wrong register number
Corrected Datapath for Load
Pipelined Execution Representation

- Every instruction must take the same number of steps, so some stages will idle
  - e.g. MEM stage for any arithmetic instruction
Graphical Pipeline Diagrams

• Use datapath figure below to represent pipeline:

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

Dr Dan Garcia
Graphical Pipeline Representation

- RegFile: left half is write, right half is read

Time (clock cycles)

Instr Order

Load
Add
Store
Sub
Or

Dr Dan Garcia
Pipelining Performance (1/3)

• Use $T_c$ (“time between completion of instructions”) to measure speedup
  - $T_{c,\text{pipelined}} \geq \frac{T_{c,\text{single-cycle}}}{\text{Number of stages}}$
  - Equality only achieved if stages are balanced (i.e. take the same amount of time)

• If not balanced, speedup is reduced

• Speedup due to increased throughput
  - *Latency* for each instruction does not decrease
Pipelining Performance (2/3)

• Assume time for stages is
  – 100ps for register read or write
  – 200ps for other stages

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

• What is pipelined clock rate?
  – Compare pipelined datapath with single-cycle datapath
Pipelining Performance (3/3)

**Single-cycle**

\( T_c = 800 \text{ ps} \)

- lw $1, 100(00)$
- lw $2, 200(00)$
- lw $3, 300(00)$

**Pipelined**

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

- lw $1, 100(00)$
- lw $2, 200(00)$
- lw $3, 300(00)$
Pipelining Hazards

A hazard is a situation that prevents starting the next instruction in the next clock cycle.

1) **Structural hazard**
   - A required resource is busy (e.g. needed in multiple stages)

2) **Data hazard**
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data read/write

3) **Control hazard**
   - Flow of execution depends on previous instruction
Agenda

- Pipelining Performance
- **Structural Hazards**
- Data Hazards
  - Forwarding
  - Load Delay Slot
- Control Hazards
1. Structural Hazards

• Conflict for use of a resource
• 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
  – Separate L1 I$ and L1 D$ take care of this
Structural Hazard #1: Single Memory

Trying to read the same memory twice in the same clock cycle.
Structural Hazard #2: Registers (1/2)

Can we read and write to registers simultaneously?
Structural Hazard #2: Registers (2/2)

• Two different solutions have been used:
  1) Split RegFile access in two: Write during 1\textsuperscript{st} half and Read during 2\textsuperscript{nd} half of each clock cycle
    • Possible because RegFile access is VERY fast (takes less than half the time of ALU stage)
  2) Build RegFile with independent read and write ports

• Conclusion: Read and Write to registers during same clock cycle is okay
Agenda

• Pipelining Performance
• Structural Hazards
  • Data Hazards
    – Forwarding
    – Load Delay Slot
• Control Hazards
2. Data Hazards (1/2)

- Consider the following sequence of instructions:

  - `add $t0, $t1, $t2`
  - `sub $t4, $t0, $t3`
  - `and $t5, $t0, $t6`
  - `or $t7, $t0, $t8`
  - `xor $t9, $t0, $t10`
2. Data Hazards (2/2)

- Data-flow *backwards* in time are hazards

**Time (clock cycles)**

- add $t0,$t1,$t2
- sub $t4,$t0,$t3
- and $t5,$t0,$t6
- or $t7,$t0,$t8
- xor $t9,$t0,$t10

Dr Dan Garcia
Data Hazard Solution: Forwarding

- Forward result as soon as it is available
  - OK that it’s not stored in RegFile yet

```
add $t0,$t1,$t2
sub $t4,$t0,$t3
and $t5,$t0,$t6
or  $t7,$t0,$t8
xor $t9,$t0,$t10
```
Datapath for Forwarding (1/2)

• What changes need to be made here?
Datapath for Forwarding (2/2)

- Handled by *forwarding unit*
Data Hazard: Loads (1/4)

- **Recall:** Dataflow backwards in time are hazards

\[
\text{lw } $t0,0($t1) \\
\text{sub } $t3,$t0,$t2
\]

- Can’t solve all cases with forwarding
  - Must *stall* instruction dependent on load, then forward (more hardware)
Data Hazard: Loads (2/4)

- **Hardware** stalls pipeline
  - Called “hardware interlock”

\[ \text{lw } \$t0, \ 0(\$t1) \]

\[ \text{sub } \$t3, \$t0, \$t2 \]

\[ \text{and } \$t5, \$t0, \$t4 \]

\[ \text{or } \$t7, \$t0, \$t6 \]

Schematically, this is what we want, but in reality stalls done “horizontally”

How to stall just part of pipeline?
Data Hazard: Loads (3/4)

- Stall is equivalent to \texttt{nop}

\texttt{lw} $\texttt{t0}, 0(\texttt{t1})$

\texttt{nop}

\texttt{sub} $\texttt{t3, t0, t2}$

and $\texttt{t5, t0, t4}$

or $\texttt{t7, t0, t6}$
Data Hazard: Loads (4/4)

• Slot after a load is called a load delay slot
  – If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle
  – Letting the hardware stall the instruction in the delay slot is equivalent to putting a \texttt{nop} in the slot (except the latter uses more code space)

• **Idea:** Let the compiler put an unrelated instruction in that slot $\rightarrow$ no stall!
Code Scheduling to Avoid Stalls

• Reorder code to avoid use of load result in the next instruction!

• MIPS code for \( D = A + B; \ E = A + C \):

  # Method 1:
  lw $t1, 0($t0)
  lw $t2, 4($t0)
  add $t3, $t1, $t2
  sw $t3, 12($t0)
  lw $t4, 8($t0)
  add $t5, $t1, $t4
  sw $t5, 16($t0)

  Stall!
  Stall!
  13 cycles

  # Method 2:
  lw $t1, 0($t0)
  lw $t2, 4($t0)
  lw $t4, 8($t0)
  add $t3, $t1, $t2
  sw $t3, 12($t0)
  add $t5, $t1, $t4
  sw $t5, 16($t0)

  11 cycles
Agenda

• More Pipelining
• Structural Hazards
• Data Hazards
  – Forwarding
  – Load Delay Slot
• Control Hazards
3. Control Hazards

• **Branch** \((\text{beq, bne})\) **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

• **Simple Solution:** Stall on *every* branch until we have the new PC value
  – How long must we stall?
Branch Stall

- When is comparison result available?

Time (clock cycles)

Instr Order
Instr 4
Instr 3
Instr 2
Instr 1
beq

TWO bubbles required per branch!
Summary

• Hazards reduce effectiveness of pipelining
  – Cause stalls/bubbles
• Structural Hazards
  – Conflict in use of datapath component
• Data Hazards
  – Need to wait for result of a previous instruction
• Control Hazards
  – Address of next instruction uncertain/unknown
Question: For each code sequences below, choose one of the statements below:

1:
   lw  $t0,0($t0)
   add $t1,$t0,$t0

2:
   add $t1,$t0,$t0
   addi $t2,$t0,5
   addi $t4,$t1,5

3:
   addi $t1,$t0,1
   addi $t2,$t0,5
   addi $t4,$t1,5
   addi $t5,$t1,5

A) No stalls as is
B) No stalls with forwarding
C) Must stall
Code Sequence 1

I$  lw  add  instr  instr  instr

Time (clock cycles)

Must stall

Dr Dan Garcia
**Question:** For each code sequences below, choose one of the statements below:

1:
   lw $t0,0($t0)
   add $t1,$t0,$t0

2:
   add $t1,$t0,$t0
   addi $t2,$t0,5
   addi $t4,$t1,5

3:
   addi $t1,$t0,1
   addi $t2,$t0,2
   addi $t3,$t0,2
   addi $t4,$t0,4
   addi $t5,$t1,5

A) **No stalls as is**

B) **No stalls with forwarding**

C) **Must stall**
Code Sequence 2

Time (clock cycles)

Instr Order

add

addi

addi

instr

instr

forwarding

no forwarding

No stalls with forwarding

Dr Dan Garcia
Question: For each code sequences below, choose one of the statements below:

1:
   lw  $t0,0($t0)
   add $t1,$t0,$t0

2:
   add $t1,$t0,$t0
   addi $t2,$t0,5
   addi $t4,$t1,5

3:
   addi $t1,$t0,1
   addi $t2,$t0,5
   addi $t4,$t1,5
   addi $t5,$t1,5

A) No stalls as is
B) No stalls with forwarding
C) Must stall
Code Sequence 3

Time (clock cycles)

Instr Order

addi
addi
addi
addi
addi

No stalls as is

Dr Dan Garcia