Announcements

• Homework #3 due today

• Homework #4 out tonight
  - Due next Thurs.
Project Overview

A. MIPS150 pipeline structure
B. Serial Interface
C. Memories, project memories and FPGAs
D. Video subsystem
E. Project specification and grading standard

MIPS 5-stage Pipeline Review

- **Fetch (IF):** Use PC register as address to instruction memory (IMEM) and retrieve next instruction.
- **Decode (ID):** Generate control signals, retrieve register values from regfile.
- **Execute (EX):** Use ALU to compute result, memory address, or compare registers.
- **Memory (DM):** Read or write data memory (DMEM).
- **Writeback (WB):** Send result back to regfile.
Control Hazard Example

MIPS 5-stage Pipeline

\[
\begin{array}{cccccc}
\text{cycle} & \text{IF} & \text{ID} & \text{EX} & \text{DM} & \text{WB} \\
\text{beq} $1, $2, L1 & \text{IF} & \text{ID} & \text{EX} & \text{DM} & \text{WB} \\
\text{add} $5, $3, $4 & \text{IF} & \text{ID} & \text{EX} & \text{DM} & \text{WB} \\
\text{L1: sub} $5, $3, $4 & \text{IF} & \text{ID} & \text{EX} & \text{DM} & \text{WB} \\
\end{array}
\]

Note: Still one remaining cycle of branch delay. "Architected branch delay slot" on MIPS allows compiler to deal with the delay. Other processors without architected branch-delay slot use branch predictors or pipeline stalling.

Modify the Datapath to Support Early Branch
MIPS 5-stage Pipeline

Data Hazard Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>DM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $5, $3, $4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $7, $6, $5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reg 5 value updated here but needed here!

New value is actually known here. Send it directly from the output register of the ALU to its input (and also down the pipeline to the register file).

Logic must be added to detect when such a hazard exists and to forward correct value to ALU. No alternative except to stall pipeline (thus hurting performance).

Modify the Datapath to Support Forwarding
MIPS 5-stage Pipeline
Load Hazard Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>DM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $5, offset($4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $7, $6, $5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>DM</td>
<td>WB</td>
</tr>
<tr>
<td>add $10, $9, $8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory value known here
value needed here!

"Architected load delay slot" on MIPS allows compiler to deal with the delay. Note, regfile still needs to be bypassed.

No other alternative except for stalling.

MIPS 5-stage Pipeline
Load Hazard Example (2)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>DM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $5, offset($4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $7, $6, $5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>DM</td>
<td>WB</td>
</tr>
<tr>
<td>add $10, $9, $8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What's the smartest thing for the compiler to do in this situation?
Processor Pipelining

Deeper pipeline example.

<table>
<thead>
<tr>
<th>IF1</th>
<th>IF2</th>
<th>ID</th>
<th>X1</th>
<th>X2</th>
<th>M1</th>
<th>M2</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF1</td>
<td>IF2</td>
<td>ID</td>
<td>X1</td>
<td>X2</td>
<td>M1</td>
<td>M2</td>
<td>WB</td>
</tr>
</tbody>
</table>

Deeper pipelines => less logic per stage => high clock rate.

But

Deeper pipelines => more hazards => more cost and/or higher CPI.

Cycles per instruction might go up because of unresolvable hazards.

Remember, Performance $\propto \frac{\text{Frequency}_{\text{clk}}}{\text{CPI}}$

How about shorter pipelines ... Less cost, less performance

MIPS150 Pipeline

The blocks in the datapath with the greatest delay are: IMEM, ALU, and DMEM. Allocate one pipeline stage to each:

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>M</th>
</tr>
</thead>
</table>

Use PC register as address to IMEM and retrieve next instruction. Instruction gets stored in a pipeline register, also called “instruction register”, in this case.

Use ALU to compute result, memory address, or compare registers for branch.

Access data memory or I/O device for load or store. Allow for setup time for register file write.

Most details you will need to work out for yourself. Some details to follow ... In particular, let’s look at hazards.
MIPS 3-stage Pipeline

Control Hazard Example

```
beq $1, $2, L1  I X M
```

```
delay slot
add $5, $3, $4  I X M
```

```
L1: sub $7, $6, $5  I X M
```

Architected branch delay slot allows us to delay branch target capture to here.

```
branch address ready here
```

```
but needed here!
```

Therefore no extra logic is required.

MIPS 3-stage Pipeline

Load Hazard

```
lw  $5, offset($4)  I X M
```

```
add $7, $6, $5  I X M
```

```
add $10, $9, $8  I X M
```

"Architected load delay slot" on MIPS allows compiler to deal with the delay. No regfile bypassing needed here assuming regfile "write before read".

```
lw  $5, offset($4)  I X M
```

```
add $10, $9, $8  I X M
```

```
add $7, $6, $5  I X M
```

"value needed here!

Memory value known here. It is written into the regfile on this edge.

### MIPS 3-stage Pipeline

#### Data Hazard

<table>
<thead>
<tr>
<th>add $5, $3, $4</th>
<th>I</th>
<th>X</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $7, $6, $5</td>
<td>I</td>
<td>X</td>
<td>M</td>
</tr>
</tbody>
</table>

- reg 5 value needed here!
- reg 5 value updated here

#### Ways to fix:

1. Stall the pipeline behind first add to wait for result to appear in register file. NOT ALLOWED this semester.
2. Selectively forward ALU result back to input of ALU.

- Need to add mux at input to ALU, add control logic to sense when to activate. A bit complex to design. Check book for details.

---

### Project CPU Pipelining Summary

#### 3-stage pipeline

- 3-stage pipeline: instruction fetch, execute access data memory

- **Pipeline rules:**
  - Writes/reads to/from DMem use leading edge of “M”
  - Writes to RegFile use trailing edge of “M”
  - Instruction Decode and Register File access is up to you.

- **1 Load Delay Slot, 1 Branch Delay Slot**
  - No Stalling may be used to accommodate pipeline hazards (in final version).

- **Other:**
  - Target frequency to be announced later [50-100MHz]
  - Minimize cost
  - Posedge clocking only
Final Project: Spring 2011

- Executes most commonly used MIPS instructions.
- Pipelined (high performance) implementation.
- Serial console interface for shell interaction, debugging.
- Video interface for display with 2-D vector graphics acceleration.
- Supported by a C language compiler.
More generally, how does software interface to I/O devices?

CPU

UART

UART-CPU Adapter

FPGA

Software communicates with UART using "UART-CPU Adapter".

UART: Universal Asynchronous Receiver and Transmitter converts to/from serial format with start/stop bits.

Oscilloscope trace of ASCII "K" transmission.
MIPS Uses Memory Mapped I/O

- Certain addresses are not regular memory
- Instead, they correspond to registers in I/O devices

Processor Checks Status Before Acting

- Path to device generally has 2 registers:
  - Control Register, says it’s OK to read/write (I/O ready) [think of a flagman on a road]
  - Data Register, holds data for transfer
- Processor reads from Control Register in loop, waiting for device to set Ready bit in Control reg \([0 \rightarrow 1]\) to say its OK
- Processor then loads from [input] or writes to [output] data register
MIPS150 Serial Line Interface

- Serial-Line Interface is a memory-mapped device.
- Modeled after SPIM terminal/keyboard interface.
  - Read from keyboard (receiver); 2 device regs
  - Writes to terminal (transmitter); 2 device regs

### Receiver Control
0xffff0000  Unused (00...00)

### Receiver Data
0xffff0004  Unused (00...00)  Received Byte

### Transmitter Control
0xffff0008  Unused (00...00)

### Transmitter Data
0xffff000c  Unused  Transmitted Byte

Serial I/O

- Control register rightmost bit [0]: Ready
  - Receiver: Ready==1 means character in Data Register not yet been read; 1 \(\rightarrow\) 0 when data is read from Data Reg
  - Transmitter: Ready==1 means transmitter is ready to accept a new character; 0 \(\rightarrow\) Transmitter still busy writing last char
  - I.E. bit not used in our implementation

- Data register rightmost byte has data
  - Receiver: last char from serial port; rest = 0
  - Transmitter: when write rightmost byte, writes goes to serial port
“Polling” MIPS code

- Input: Read from keyboard into $v0

```
lui $t0, 0xffff  #ffff0000
Waitloop1:  
  lw  $t1, 0($t0)  #control
  andi $t1,$t1,0x1
  beq  $t1,$zero, Waitloop1
  lw  $v0, 4($t0)  #data
```

- Output: Write to display from $a0

```
lui $t0, 0xffff  #ffff0000
Waitloop2:  
  lw  $t1, 8($t0)  #control
  andi $t1,$t1,0x1
  beq  $t1,$zero, Waitloop2
  sw  $a0, 12($t0)  #data
```