Agenda

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion
Datapath vs Control

- **Datapath**: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals

- **Controller**: State machine to orchestrate operation on the data path
  - Based on desired function and signals
Approaching an ISA

- **Instruction Set Architecture**
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing

- **Meaning of each instruction is described by RTL on architected registers and memory**

- **Given technology constraints assemble adequate datapath**
  - Architected storage mapped to actual storage
  - Function units to do all the required operations
  - Possible additional storage (e.g., MAR, MBR, …)
  - Interconnect to move information among regs and FUs

- **Map each instruction to sequence of RTLs**

- **Collate sequences into symbolic controller state transition diagram (STD)**

- **Lower symbolic STD to control points**

- **Implement controller**
A "Typical" RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  - base + displacement
    - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Basics of a RISC Instruction Set

- RISC architectures are characterized by the following features that dramatically simplifies the implementation:
  1. All ALU operations apply only on data in registers
  2. Memory is affected only by load and store operations
  3. Instructions follow very few formats and typically are of the same size

- All MIPS instructions are 32 bits, following one of three formats:

  **R-type**
  
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

  **I-type**
  
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>

  **J-type**
  
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
</tr>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

* Slide is courtesy of Dave Patterson
MIPS Instruction format

1. Register-format instructions:

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **op**: Basic operation of the instruction, traditionally called opcode
- **rs**: The first register source operand
- **rt**: The second register source operand
- **rd**: The register destination operand, it gets the result of the operation
- **shmat**: Shift amount
- **funct**: This field selects the specific variant of the operation of the op field

MIPS assembly language includes two conditional branching instructions using PC-relative addressing:

- `beq register1, register2, L1`  
  # go to L1 if (register1) = (register2)

- `bne register1, register2, L1`  
  # go to L1 if (register1) ≠ (register2)

Examples:

- `add $t2, $t1, $t1`  
  # Temp reg $t2 = 2 $t1

- `sub $t1, $s3, $s4`  
  # Temp reg $t1 = $s3 - $s4

- `and $t1, $t2, $t3`  
  # Temp reg $t1 = $t2 . $t

- `bne $s3, $s4, Else`  
  # if $s3 ≠ $s4 jump to Else
### MIPS Instruction format

#### Immediate-type instructions:

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- The 16-bit address means a load word instruction can load a word within a region of ±2\(^{15}\) bytes of the address in the base register.

- Examples:
  
  - `lw $t0, 32($s3)`
  
  - `sw $t1, 128($s3)`

- MIPS handle 16-bit constant efficiently by including the constant value in the address field of an I-type instruction (Immediate-type):
  
  - `addi $sp, $sp, 4` #\(\texttt{sp} = \texttt{sp} + 4\)

- For large constants that need more than 16 bits, a load upper-immediate (*lui*) instruction is used to concatenate the second part.

#### `lui $t0, 255`:

| 001111 | 00000 | 01000 | 0000 0000 1111 1111 |

Contents of \$t0 after execution:

| 0000 0000 1111 1111 | 0000 0000 0000 0000 |
Addressing in Branches & Jumps

- I-type instructions leaves only 16 bits for address reference limiting the size of the jump
- MIPS branch instructions use the address as an increment to the PC allowing the program to be as large as $2^{32}$ (called *PC-relative addressing*)
- Since the program counter gets incremented prior to instruction execution, the branch address is actually relative to $(PC + 4)$
- MIPS also supports an J-type instruction format for large jump instructions

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- The 26-bit address in a J-type instruct. is concatenated to upper 8 bits of PC

<table>
<thead>
<tr>
<th>Loop:</th>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t1, $s3, $s3</td>
<td>80000</td>
<td>0 19 19 9 0 32</td>
</tr>
<tr>
<td>add $t1, $t1, $t1</td>
<td>80004</td>
<td>0 9 9 9 0 32</td>
</tr>
<tr>
<td>add $t1, $t1, $s6</td>
<td>80008</td>
<td>0 9 22 9 0 32</td>
</tr>
<tr>
<td>lw $t0, 0($t1)</td>
<td>80012</td>
<td>35 9 8 0</td>
</tr>
<tr>
<td>bne $t0, $s5, Exit</td>
<td>80016</td>
<td>5 8 21 8</td>
</tr>
<tr>
<td>add $s3, $s3, $s4</td>
<td>80020</td>
<td>0 19 20 19 0 32</td>
</tr>
<tr>
<td>j Loop</td>
<td>80024</td>
<td>2 80000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exit:</th>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>80028</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>80012</td>
<td>35 9 8 0</td>
<td></td>
</tr>
</tbody>
</table>
5 Steps of MIPS Datapath

Instruction Fetch
- Next PC

Instr. Decode
- Reg. Fetch
- Memory Access

Execute
- Addr. Calc

Write Back
- Next SEQ PC

IR $\leftarrow$ mem[PC];
PC $\leftarrow$ PC + 4

Reg[IR$_{rd}$] $\leftarrow$ Reg[IR$_{rs}$] op$_{IRop}$ Reg[IR$_{rt}$]
5 Steps of MIPS Datapath

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Next PC</td>
<td>Next SEQ PC</td>
<td>Next SEQ PC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **IF/ID**
  - Address Memory
  - IR <= mem[PC];
  - PC <= PC + 4

- **ID/EX**
  - Reg File
  - A <= Reg[IR<sub>rs</sub>];
  - B <= Reg[IR<sub>rt</sub>]
  - rslt <= A op<sub>IRop</sub> B
  - WB <= rslt
  - Reg[IR<sub>rd</sub>] <= WB

- **EX/MEM**
  - Data Memory
  - Zero?

- **MEM/WB**
  - WB Data

**Adder**
- 4

**Mux**
- RS1
- RS2

**Sign Extender**
- Imm
Inst. Set Processor Controller

IR \leftarrow \text{mem}[PC];
PC \leftarrow PC + 4

A \leftarrow \text{Reg}[IR_{rs}];
B \leftarrow \text{Reg}[IR_{rt}]

Ifetch

If bop(A, b)
PC \leftarrow PC + IR_{im}

br

PC \leftarrow IR_{jaddr}

jmp

r \leftarrow A \text{ op}_{IR_{op}} B

rr

r \leftarrow A \text{ op}_{IR_{op}} IR_{im}

ri

r \leftarrow A + IR_{im}

st

WB \leftarrow r

LD

WB \leftarrow \text{Mem}[r]

JR

JR

WB \leftarrow r

LD

WB \leftarrow r

LD

Reg[IR_{rd}] \leftarrow WB

LD

Reg[IR_{rd}] \leftarrow WB

LD

Reg[IR_{rd}] \leftarrow WB
A Simple Implementation of MIPS
Multi-Cycle Implementation of MIPS

1. Instruction fetch cycle (IF)
   \[ IR \leftarrow \text{Mem}[PC]; \quad \text{NPC} \leftarrow PC + 4 \]

2. Instruction decode/register fetch cycle (ID)
   \[ A \leftarrow \text{Regs}[IR_{6..10}]; \quad B \leftarrow \text{Regs}[IR_{11..15}]; \quad \text{Imm} \leftarrow ((IR_{16})^{16} \# \# IR_{16..31}) \]

3. Execution/effective address cycle (EX)
   Memory ref: \[ \text{ALUOutput} \leftarrow A + \text{Imm}; \]
   Reg-Reg ALU: \[ \text{ALUOutput} \leftarrow A \func B; \]
   Reg-Imm ALU: \[ \text{ALUOutput} \leftarrow A \op \text{Imm}; \]
   Branch: \[ \text{ALUOutput} \leftarrow \text{NPC} + \text{Imm}; \quad \text{Cond} \leftarrow (A \op 0) \]

4. Memory access/branch completion cycle (MEM)
   Memory ref: \[ \text{LMD} \leftarrow \text{Mem}[\text{ALUOutput}] \quad \text{or} \quad \text{Mem}(\text{ALUOutput}) \leftarrow B; \]
   Branch: \[ \text{if (cond) PC} \leftarrow \text{ALUOutput}; \]

5. Write-back cycle (WB)
   Reg-Reg ALU: \[ \text{Regs}[IR_{16..20}] \leftarrow \text{ALUOutput}; \]
   Reg-Imm ALU: \[ \text{Regs}[IR_{11..15}] \leftarrow \text{ALUOutput}; \]
   Load: \[ \text{Regs}[IR_{11..15}] \leftarrow \text{LMD}; \]
Multi-cycle Instruction Execution
The load instruction is the longest.

All instructions follow at most the following five steps:

- **Ifetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory and update PC
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Calculate the memory address
- **Mem**: Read the data from the Data Memory
- **WB**: Write the data back to the register file
Instruction Pipelining

- Start handling of next instruction while the current instruction is in progress
- Pipelining is feasible when different devices are used at different stages of instruction execution

**Program Flow**

Time between instructions $_{pipelined} = \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of pipe stages}}$

Pipelining improves performance by increasing instruction throughput
Single Cycle, Multiple Cycle, vs. Pipeline

**Single Cycle Implementation:**

- Cycle 1: Load
- Cycle 2: Store
- Cycle 3: Waste

**Multiple Cycle Implementation:**

- Cycle 1: Load
- Cycle 2: Ifetch
- Cycle 3: Reg
- Cycle 4: Exec
- Cycle 5: Mem
- Cycle 6: Wr
- Cycle 7: Store
- Cycle 8: Ifetch
- Cycle 9: Reg
- Cycle 10: Exec
- Cycle 11: Mem
- Cycle 12: Ifetch

**Pipeline Implementation:**

- Load: Ifetch
- Reg
- Exec
- Mem
- Wr
- Store: Ifetch
- Reg
- Exec
- Mem
- Wr
- R-type: Ifetch
- Reg
- Exec
- Mem
- Wr

*Slide is courtesy of David Patterson*
Example of Instruction Pipelining

Program execution order (in instructions)

lw $1, 100(0)

lw $2, 200(0)

lw $3, 300(0)

Time

2 4 6 8 10 12 14 16 18

Time between first & fourth instructions is $3 \times 8 = 24$ ns

Ideal and upper bound for speedup is number of stages in the pipeline

Program execution order (in instructions)

lw $1, 100(0)$

lw $2, 200(0)$

lw $3, 300(0)$

Time

2 4 6 8 10 12 14

Time between first & fourth instructions is $3 \times 2 = 6$ ns
Pipeline Performance

- Pipeline increases the instruction throughput but does not reduce the execution time of the individual instruction.

- Execution time of the individual instruction in pipeline can be slower due:
  - Additional pipeline control compared to none pipeline execution
  - Imbalance among the different pipeline stages

- Suppose we execute 100 instructions:
  - **Single Cycle Machine**
    - \(45 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = 4500 \text{ ns}\)
  - **Multi-cycle Machine**
    - \(10 \text{ ns/cycle} \times 4.2 \text{ CPI (due to inst mix)} \times 100 \text{ inst} = 4200 \text{ ns}\)
  - **Ideal 5 stages pipelined machine**
    - \(10 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = 1040 \text{ ns}\)

- Due to fill and drain effects of a pipeline ideal performance can be achieved only for long (> > 2*pipeline_depth) instruction streams

**Example:** a sequence of 1000 load instructions would take 5000 cycles on a multi-cycle machine while taking 1004 on a pipeline machine

\[\Rightarrow \text{speedup} = \frac{5000}{1004} \approx 5\]
• Data stationary control
  - local decode for each instruction phase / pipeline stage
Pipelining is not quite that easy!

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
One Memory Port/Structural Hazards

Time (clock cycles)

Cycle 1: Load, Ifetch, Reg, ALU
Cycle 2: Instr 1, Ifetch, Reg, ALU
Cycle 3: Instr 1, ALU, DMem, Reg
Cycle 4: Instr 2, Ifetch, Reg, ALU
Cycle 5: Instr 2, ALU, DMem, Reg
Cycle 6: Instr 3, Ifetch, Reg, ALU
Cycle 7: Instr 3, ALU, DMem, Reg

Instr 4: Instr 4, Ifetch, Reg, ALU

CEN591 Fall 2011
One Memory Port/Structural Hazards

Time (clock cycles)

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7

Load Ifetch Reg ALU DMem Reg
Instr 1 Ifetch Reg ALU DMem Reg
Instr 2 Ifetch Reg ALU DMem Reg
Stall Bubble Bubble Bubble Bubble Bubble
Instr 3 Ifetch Reg ALU DMem Reg

How do you “bubble” the pipe?
Speed Up Equation for Pipelining

\[ \text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst} \]

\[ \text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}} \]

For simple RISC pipeline, CPI = 1:

\[ \text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}} \]
Example: Dual-port vs. Single-port

- Machine A: Dual ported memory (“Harvard Architecture”)
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

\[
\text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times \left(\frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}}\right) = \text{Pipeline Depth}
\]

\[
\text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1 + 0.4 \times 1} \times \left(\frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{unpipe}} / 1.05}\right) = \left(\frac{\text{Pipeline Depth}}{1.4}\right) \times 1.05 = 0.75 \times \text{Pipeline Depth}
\]

\[
\frac{\text{SpeedUp}_A}{\text{SpeedUp}_B} = \frac{\text{Pipeline Depth}}{0.75 \times \text{Pipeline Depth}} = 1.33
\]

- Machine A is 1.33 times faster
Summary

- One must be careful in interpreting the reliability (performance) figures quoted by vendors.
- RISC ISAs are designed for pipelining in mind.
- Pipeline performance is dependent upon many factors such as how balanced the pipeline stages are and the average number of stalls.
- Next class: Hazards and techniques to deal with them.