### Schedule

<table>
<thead>
<tr>
<th>Lecture #</th>
<th>Date</th>
<th>Day</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sep. 25</td>
<td>Thursday</td>
<td>Introduction, Ch. 1</td>
</tr>
<tr>
<td>2</td>
<td>Sep. 30</td>
<td>Tuesday</td>
<td>Performance, Ch. 2</td>
</tr>
<tr>
<td>3</td>
<td>Oct. 2</td>
<td>Thursday</td>
<td>ISA, Ch. 3</td>
</tr>
<tr>
<td>4</td>
<td>Oct. 7</td>
<td>Tuesday</td>
<td>Arithmetic, Ch. 4</td>
</tr>
<tr>
<td>5</td>
<td>Oct. 9</td>
<td>Thursday</td>
<td>Arithmetic, Ch. 4, Continued</td>
</tr>
<tr>
<td>6</td>
<td>Oct. 14</td>
<td>Tuesday</td>
<td>Single cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>7</td>
<td>Oct. 18</td>
<td>Thursday</td>
<td>Single-cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>8</td>
<td>Oct. 21</td>
<td>Tuesday</td>
<td>Multi-cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>9</td>
<td>Oct. 23</td>
<td>Thursday</td>
<td>Multi-cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>10</td>
<td>Oct. 28</td>
<td>Tuesday</td>
<td>Classes cancelled due to wildfires</td>
</tr>
<tr>
<td>11</td>
<td>Oct. 30</td>
<td>Thursday</td>
<td>Exceptions and Review for Midterm</td>
</tr>
<tr>
<td>12</td>
<td>Nov. 4</td>
<td>Tuesday</td>
<td>Mid-term Exam</td>
</tr>
<tr>
<td>13</td>
<td>Nov. 6</td>
<td>Thursday</td>
<td>Pipelining, Ch. 6</td>
</tr>
<tr>
<td>No Class</td>
<td>Nov. 11</td>
<td>Tuesday</td>
<td>Veteran's Day Holiday</td>
</tr>
<tr>
<td>14</td>
<td>Nov. 13</td>
<td>Thursday</td>
<td>Data and control hazards, Ch. 6</td>
</tr>
<tr>
<td>15</td>
<td>Nov. 18</td>
<td>Tuesday</td>
<td>Data and control hazards, Ch. 6</td>
</tr>
<tr>
<td>16</td>
<td>Nov. 20</td>
<td>Thursday</td>
<td>Data and control hazards, Ch. 6</td>
</tr>
<tr>
<td>17</td>
<td>Nov. 25</td>
<td>Tuesday</td>
<td>Advanced pipelining issues, Ch. 6</td>
</tr>
<tr>
<td>No Class</td>
<td>Nov. 27</td>
<td>Thursday</td>
<td>Thanksgiving Holiday</td>
</tr>
<tr>
<td>18</td>
<td>Dec. 2</td>
<td>Tuesday</td>
<td>Memory &amp; cache design, Ch. 7</td>
</tr>
<tr>
<td>19</td>
<td>Dec. 4</td>
<td>Thursday</td>
<td>Memory &amp; cache design, Ch. 7</td>
</tr>
<tr>
<td>Dec. 9</td>
<td>Tuesday</td>
<td>Final Exam</td>
<td></td>
</tr>
</tbody>
</table>

### What are Exceptions?

- Non-sequential control flow in a processor due to:
  - Explicit branch and jump instructions
  - Exceptions
- Branches:
  - Synchronous and deterministic
  - Conditional branches are generally data dependent
- Exceptions:
  - Typically asynchronous and non-deterministic
  - Generally more difficult to handle than branches
  - Detection and handling of exceptions is generally in timing critical path
- Priority and interrupt mask
- Non-maskable interrupt
Exceptions and Interrupts

- The terminology for all microprocessors is not consistent.
- MIPS Terminology:
  - *exceptions*: any unexpected change in control flow
  - *interrupts*: any externally-caused exception

So then, what is:
- arithmetic overflow
- divide by zero
- I/O device signals completion to CPU
- user program invokes the OS
- memory parity error
- illegal instruction
- timer signal

Handling exceptions: Two Methods

- Common step: PC saved in EPC (exception program counter)
- Method 1 (MIPS way)
  - A status *cause register*, and a single exception handler may be used to record the exception and transfer control
- Method 2 (Vectored Interrupt)
  - Control is transferred to a different location for each possible type of interrupt/exception
Simplification

- Restrict our design to handle two types of exceptions
  - Arithmetic overflow
  - Undefined instruction
    - Instruction that is not known to the processor
- On an exception, we need to
  - Save the PC
    - in Exception PC register (EPC)
  - Record the nature of the exception/interrupt
    - in cause register
  - Transfer control to a pre-defined instruction memory location
    - Generally access to this area of memory is restricted to OS only
    - OS takes appropriate action

Implementation of exceptions in MIPS

- For our MIPS-subset architecture, we will add two registers:
  - EPC Register: a 32-bit register to hold the user’s PC
  - Cause Register: A register to record the cause of the exception
    - we’ll assume undefined inst = 0, overflow = 1
- We will also add three control signals:
  - EPCWrite (will need to be able to subtract 4 from PC)
  - CauseWrite
  - IntCause
- We will extend PCSource multiplexor to be able to register the interrupt handler address into the PC
Exception Datapath

Supporting exceptions in our FSM

Instruction Fetch, state 0
Start
MemRead
ALUSelA = 0
IorD = 0
IRWrite
ALUSelB = 01
ALUOp = 00
PCWrite
PCSource = 00

Instruction Decode/ Register Fetch, state 1
ALUSelA = 0
ALUSelB = 11
ALUOp = 00

OprCode = LW or SW
OprCode = R-type
OprCode = BRD
OprCode = JMP

Memory Inst FSM
R-type Inst FSM
Branch Inst FSM
Jump Inst FSM

to state 10
Supporting Overflow exception in our FSM

from state 1

R-type instructions

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

state 6

state 7

RegDst = 1
RegWrite
MemtoReg = 0

overflow

To state 11

Normal completion of R-type instruction

To state 0

State to Support Exceptions

IntCause=1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource=11

IntCause=0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource=11

To state 0 (fetch)
Midterm Review

- Closed Book
- No Notes
- Through Chapter 5
The five classic components of computers

- **Computer**
- **Memory**
- **Datapath**
- **Control**
- **Output**
- **Input**

**Performance**

- **Performance**
  - Execution Time = (Instruction Count) * CPI * (Cycle Time)
  - Clock rate is in cycles per
    - MHz (Millions of cycles per second)
    - GHz (Billions of cycles per second)
  - Cycle time = 1/(Clock Rate)
  - speedup = (exe time without change / exe time with change)

  \[
  \frac{\text{Relative Performance}}{\text{Performance}_Y} = \frac{\text{Execution Time}_Y}{\text{Execution Time}_X} = n
  \]

- **Amdahl’s Law**

  \[
  \text{Execution time after improvement} = \frac{\text{Execution Time Affected}}{\text{Amount of Improvement}} + \text{Execution Time Unaffected}
  \]
RISC Processor:

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI((i))</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>23%</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>45%</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>14%</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>18%</td>
</tr>
</tbody>
</table>

\[ \text{Average CPI} = \frac{2.2}{1.6} = 1.38x \]

- What is average CPI?
- What percentage of time is spent in each instruction class?
- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?
  Speedup = \( \frac{2.2}{2.0} = 1.1x \)
  How does this compare with using branch prediction to shave a cycle off the branch time?
  Speedup = \( \frac{2.2}{1.95} = 1.13x \), worst case none

ISA

- Instruction Length
  - variable
  - fixed
  - hybrid
  - MIPS has fixed 32 bit length
- Basic ISA Types
  - load-store
  - reg-mem
  - stack
  - accumulator
Overview of MIPS ISA

- 3-operand, load-store architecture
- 32 general-purpose registers (integer, floating point)
  - R0 always equals 0.
- 2 special-purpose integer registers, HI and LO, because multiply and divide produce more than 32 bits.
- Registers are 32-bits wide (word)
- Register, immediate, base+displacement, PC-relative and pseudo-direct addressing modes
- Fixed 32-bit instructions
- 3 instruction formats

MIPS Addressing Modes

1. Immediate addressing
   - op  in  it  Immediate

2. Register addressing
   - op  in  it  (Register)

3. Base addressing
   - op  in  it  Address  Memory

4. PC-relative addressing
   - op  in  it  Address  Memory

5. Pseudo-direct addressing
   - op  in  it  Address  Memory
The MIPS Instruction Formats

- All MIPS instructions are 32 bits long.

### R-type

<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>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

### I-type

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

### J-type

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

---

To summarize:

<table>
<thead>
<tr>
<th>MIPS operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers, $a0-$a3, $v0-$v1, $gp</td>
<td>Fast locations for data; MIPS data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>2nd memory word, Memory[0], ..., Memory[4]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

### MIPS assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>$s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands, data in registers.</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>$s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from memory to register.</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>$s1, $s2, 25</td>
<td>if ($s2 = $s1) go to PC + 4 + 100</td>
<td>Jump to target address.</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>$s1, $s2, 25</td>
<td>if ($s2 != $s1) go to PC + 4 + 100</td>
<td>Jump to target address.</td>
</tr>
</tbody>
</table>

---

Pramod Argade
UCSD CSE 141, Fall 2003
Arithmetic

- Decimal, binary and hex representation
- Two’s Complement
  - 2’s complement representation of negative numbers
    - Take the bitwise inverse and add 1
- Ripple carry adder
  - Worst case delay for a N-bit adder: 2N-gate delay
- Carry Lookahead adder
  - Generate Carry at Bit i: \( g_i = A_i \land B_i \)
  - Propagate Carry via Bit i: \( p_i = A_i \lor B_i \)
  - 2 gate delay to calculate the carry in bits
    - \( C_{in1} = g_0 \lor (p_0 \land C_{in0}) \)
    - \( C_{in2} = g_1 \lor (p_1 \land g_0) \lor (p_1 \land p_0 \land C_{in0}) \)
    - \( C_{in3} = g_2 \lor (p_2 \land g_1) \lor (p_2 \land p_1 \land g_0) \lor (p_2 \land p_1 \land p_0 \land C_{in0}) \)
  - Worst case 5 gate delays
- Overflow flag: \( C_{in_{MSB}} \land C_{in_{1MSB}} \)

Math

- Ripple Carry Adder
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for a N-bit adder: 2N-gate delay
- Carry Lookahead adder
  - Generate Carry at Bit i: \( g_i = A_i \land B_i \)
  - Propagate Carry via Bit i: \( p_i = A_i \lor B_i \)
  - 2 gate delay to calculate the carry in bits
    - \( C_{in1} = g_0 \lor (p_0 \land C_{in0}) \)
    - \( C_{in2} = g_1 \lor (p_1 \land g_0) \lor (p_1 \land p_0 \land C_{in0}) \)
    - \( C_{in3} = g_2 \lor (p_2 \land g_1) \lor (p_2 \land p_1 \land g_0) \lor (p_2 \land p_1 \land p_0 \land C_{in0}) \)
Booth’s algorithm: Signed multiplication

<table>
<thead>
<tr>
<th>Current Bit</th>
<th>Bit to the Right</th>
<th>Explanation</th>
<th>Example</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Begins run of 1s</td>
<td>0001111000</td>
<td>sub</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of run of 1s</td>
<td>0001111000</td>
<td>none</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of run of 1s</td>
<td>001111000</td>
<td>add</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of run of 0s</td>
<td>0001111000</td>
<td>none</td>
</tr>
</tbody>
</table>

Originally for Speed (when shift was faster than add)

• Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one
• Potential speed up recognizing that string of 0’s and 1’s requires no operation!

Booth’s algorithm

• Example \(-2 \times -6 = 1110 \times 1010:\)

\[
\begin{array}{c}
1110 \\
\times \\
10100
\end{array}
\]

+ 0000 zero
- (1110x) sub
+ (1110xx) add
- (1110xxx) sub

Above sums are:

0000 zero
000010x sub
111110xx add
00010xxx sub
00001100 Result
IEEE Floating Point Standard

- Example:
  - decimal: -0.75 = -3/4 = -3/2^2
  - binary: -0.11 = -1.1 x 2^-1
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: 10111111010000000000000000000000

Single-cycle Datapath and Control

Diagram showing the datapath and control flow of a single-cycle processor, including memory read, instruction fetch, and ALU operations.
Multi-cycle CPU

Complete FSM