Last Time…

- An ISA can have multiple implementations
- (Briefly) CISC vs. RISC
  - CISC: microcoded (macro instructions & microinstructions)
  - RISC: combinational logic
- MIPS microarchitecture
  - Good RISC example
  - Fixed format instructions
  - Load/store architecture with single address mode
  - Simple branch
- Will continue on this today…

MIPS Instruction Formats

**Register-Register**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>16</th>
<th>20</th>
<th>21</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td>Func</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register-Immediate**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Branch (Same format as Reg-Imm)**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Op</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Jump / Call**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>target</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reg-Reg Instructions

**Register-Register**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>16</th>
<th>20</th>
<th>21</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td>Func</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Op (e.g., 0) encodes that this is a reg-reg instruction
- Func encodes the datapath operations (add, sub, etc.)
- Rd ← (Rs1) Func (Rs2)
- ADD R1,R2,R3
  - Add
  - Reg[R1] ← Reg[R2] + Reg[R3]

Jump / Call

**Register-Register**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>16</th>
<th>20</th>
<th>21</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td>Func</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Op (e.g., 0) encodes that this is a jump/call instruction
- Func encodes the jump address shift
- Rd ← (Rs1) Op (Imm)
- J name
  - Jump
  - PC_{L,R} ← name
- JAL name
  - Jump and link
  - Reg[R31] ← PC + 4; PC_{L,R} ← name

Reg-Imm Instructions

**Register-Immediate**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Rd ← (Rs1) Op (Imm)
- ADDI R1,R2,#3
  - Add immediate
  - Reg[R1] ← Reg[R2] + 3
- LD R1,30(R2)
  - Load word
  - Reg[R1] ← Mem[30 + Reg[R2]]
- BEQZ R4, name
  - Branch equal zero
  - If (Reg[R4] == 0) PC ← name
Implementing MIPS:
Single-cycle per instruction datapath & control logic

Datapath: Reg-Reg ALU Instructions

Datapath: Reg-Imm ALU Instructions

Conflicts in Merging Datapath

Datapath for Memory Instructions

Datapath for ALU Instructions

Should program and data memory be separate?

- Harvard style: separate (Aiken and Mark 1 influence)
  - read-only program memory
  - read/write data memory

- Note:
  Somehow there must be a way to load the program memory

- Princeton style: the same (von Neumann's influence)
  - single read/write memory for program and data

- Note:
  A Load or Store instruction requires accessing the memory more than once during its execution

Should program and data memory be separate?
Load/Store Instructions: Harvard Datapath

Load/Store Instructions:
- **Harvard Datapath**
- \( WBSrc \) to ALU / Mem
- \( rs \) is the base register
- \( rt \) is the destination of a Load or the source for a Store
- \( 6 \) \( 5 \) \( 5 \) \( 16 \) addressing mode
- \( 31 \) \( 26 \) \( 25 \) \( 21 \) \( 20 \) \( 16 \) \( 15 \) \( 0 \) \( (rs) + \) displacement

MIPS Control Instructions

Conditional (on GPR) PC-relative branch
- \( 6 \) \( 5 \) \( 5 \) \( 16 \) \( BEQZ, BNEZ \)

Unconditional register-indirect jumps
- \( 6 \) \( 5 \) \( 5 \) \( 15 \) \( JR, JALR \)

Unconditional absolute jumps
- \( 6 \) \( 26 \) \( 15 \) \( J, JAL \)
- PC-relative branches add \( offset \times 4 \) to \( PC+4 \) to calculate the target address (offset is in words): ±128 KB range
- Absolute jumps append target \( \times 4 \) to \( PC<31:28> \) to calculate the target address: 256 MB range
- Jump-&-link stores \( PC+4 \) into the link register (R31)

CSE 490/590 Administrivia

- Please purchase a BASYS2 board (100K) as soon as possible.
  - Projects should be done individually.
- Quiz 1
  - Fri, 2/4
  - Closed book, in-class
  - 10%
- Class cancelled on Fri, 4/15
  - Will update the schedule

Register-Indirect Jumps (JR)

Register-Indirect Jump-&-Link (JALR)
Single-Cycle Hardwired Control: Harvard architecture

We will assume
• clock period is sufficiently long for all of the following steps to be “completed”:

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. data fetch if required
5. register write-back setup time

⇒ \( t_c > t_{\text{IFetch}} + t_{\text{RFetch}} + t_{\text{ALU}} + t_{\text{DMem}} + t_{\text{RWB}} \)

• At the rising edge of the following clock, the PC, the register file and the memory are updated.

An Ideal Pipeline

- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

These conditions generally hold for industrial assembly lines.
But can an instruction pipeline satisfy the last condition?

Pipelined MIPS

To pipeline MIPS:

• First build MIPS without pipelining with CPI=1

• Next, add pipeline registers to reduce cycle time while maintaining CPI=1
Clock period can be reduced by dividing the execution of an instruction into multiple cycles

\[ t_C > \max\{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\} \]  

However, CPI will increase unless instructions are pipelined

“Iron Law” of Processor Performance

- Instructions per program depends on source code, compiler technology, and ISA
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CPI</th>
<th>cycle time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcoded</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>Single-cycle unpipelined</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>Pipelined</td>
<td>1</td>
<td>short</td>
</tr>
</tbody>
</table>

A 5-stage pipeline will be the focus of our detailed design

- some commercial designs have over 30 pipeline stages to do an integer add!

Acknowledgements

- These slides heavily contain material developed and copyright by:
  - Krste Asanovic (MIT/UCB)
  - David Patterson (UCB)
- And also by:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252