Hardwired, Non-pipelined ISA Implementation

Hyun Ryong (Ryan) Lee
Computer Science & Artificial Intelligence Lab
M.I.T.
Administerivia

- TA: Ryan Lee
- Contact: 6823-tas@lists.csail.mit.edu
  or hrlee@csail.mit.edu
- Please use Piazza extensively for questions!
- Office Hours: Wed. 4-5:30pm, 32-G725 or by appointment
- Tutorials every week (optional)
  - First two will cover background materials
  - Cover lab tools (Intel Pin, Murphi)
  - Go over problem sets, quiz prep
  - Two sessions will be reserved for Quizzes. Do not miss them!
Last Lecture...

• Computer Architecture as designing under constraints
  – Spans much of the computing stack nowadays

• Looked at some early computers and their instruction sets (e.g., ENIAC)
  – Instruction set tightly coupled to the technology
  – Need for compatibility, well-defined interfaces
    -> Instruction Set Architecture (ISA)
    -> Will see these in detail next lecture!

• Today: single-cycle implementation of a processor
Processor Performance

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- 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>
Hardware Elements

- Combinational circuits
  - Mux, Demux, Decoder, ALU, ...

- Synchronous state elements
  - Flipflop, Register, Register file, SRAM, DRAM

Edge-triggered: Data is sampled at the rising edge
No timing issues when reading and writing the same register (writes happen at the end of the cycle)
Register File Implementation

- Register files with a large number of ports are difficult to design
  - Area scales with ports
  - Almost all Alpha instructions have exactly 2 register source operands
  - Intel’s Itanium GPR File has 128 registers with 8 read ports and 4 write ports!!
A Simple Memory Model

- Reads and writes are always completed in one cycle
  - A Read can be done any time (i.e., combinational)
  - If enabled, a Write is performed at the rising clock edge
    
    \textit{(the write address and data must be stable at the clock edge)}

\textit{Later in the course we will present a more realistic model of memory}
Implementing MIPS:
Single-cycle per instruction datapath & control logic
The MIPS ISA

Processor State
32 32-bit GPRs, R0 always contains a 0
32 single precision FPRs, may also be viewed as
  16 double precision FPRs
FP status register, used for FP compares & exceptions
PC, the program counter
Some other special registers

Data types
8-bit byte, 16-bit half word
32-bit word for integers
32-bit word for single precision floating point
64-bit word for double precision floating point

Load/Store style instruction set
Data addressing modes: immediate & indexed
Branch addressing modes: PC relative & register indirect
Byte-addressable memory, big-endian mode

All instructions are 32 bits
Execution of an instruction involves:

1. Instruction fetch
2. Decode
3. Register fetch
4. ALU operation
5. Memory operation (optional)
6. Write back

And computing the address of the next instruction (next PC)
Datapath: Reg-Reg ALU Instructions

RegWrite Timing?

rd ← (rs) func (rt)

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>func</td>
</tr>
</tbody>
</table>

February 18, 2021
Datapath: Reg-Imm ALU Instructions

```
<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
</tbody>
</table>
```

rt ← (rs) op immediate
Conflicts in Merging Datapath

Introduce muxes

rd ← (rs) func (rt)
rt ← (rs) op immediate
Datapath for ALU Instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

rd ← (rs) func (rt)
rt ← (rs) op immediate
Datapath for Memory 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:
  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:
  Executing a Load or Store instruction requires accessing the memory more than once
Load/Store Instructions

Harvard Datapath

rs is the base register
rt is the destination of a Load or the source for a Store

addressing mode
(rs) + displacement
# MIPS Control Instructions

## Conditional (on GPR) PC-relative branch

![Diagram](https://example.com/diagram.png)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ, BNEZ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Unconditional register-indirect jumps

![Diagram](https://example.com/diagram.png)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs</th>
</tr>
</thead>
<tbody>
<tr>
<td>JR, JALR</td>
<td></td>
</tr>
</tbody>
</table>

## Unconditional absolute jumps

![Diagram](https://example.com/diagram.png)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>J, JAL</td>
<td></td>
</tr>
</tbody>
</table>

### Target PC

<table>
<thead>
<tr>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ</td>
</tr>
<tr>
<td>BNEZ</td>
</tr>
<tr>
<td>JR, JALR</td>
</tr>
<tr>
<td>J, JAL</td>
</tr>
</tbody>
</table>

- Jump-&-link stores PC+4 into the link register (R31)
- Control transfers are not delayed

*we will worry about the branch delay slot later*
Conditional Branches (BEQZ, BNEZ)
Register-Indirect Jumps (JR)
Register-Indirect Jump-&-Link (JALR)
Absolute Jumps (J, JAL)
Harvard-Style Datapath for MIPS
Hardwired Control is pure Combinational Logic

op code → combinational logic
zero? → combinational logic

ExtSel
BSrc
OpSel
MemWrite
WBSrc
RegDst
RegWrite
PCSrc
ALU Control & Immediate Extension

Inst<5:0> (Func)
Inst<31:26> (Opcode)

ALUop

OpSel
( Func, Op, +, 0? )

ExtSel
( sExt_{16}, uExt_{16}, High_{16} )

Decode Map
# Hardwired Control Table

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ExtSel</th>
<th>BSrc</th>
<th>OpSel</th>
<th>MemW</th>
<th>RegW</th>
<th>WBSrc</th>
<th>RegDst</th>
<th>PCSrc</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUiu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQZ(_{z=0})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQZ(_{z=1})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JALR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BSrc = Reg / Imm
RegDst = rt / rd / R31
WBSrc = ALU / Mem / PC
PCSsrc = pc+4 / br / rind / jabs
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

\[ t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB} \]

- At the rising edge of the following clock, the PC, the register file and the memory are updated
Princeton challenge

• What problem arises if instructions and data reside in the same memory?
Two-State Controller:
Princeton Architecture

**fetch phase**
- AddrSrc=PC
- IRen=on
- PCen=off
- Wen=off

**execute phase**
- AddrSrc=ALU
- IRen=off
- PCen=on
- Wen=on

A flipflop can be used to remember the phase
Hardwired Controller:
Princeton Architecture

IR → old combinational logic (Harvard)

IR → ExtSel, BSrc, OpSel, WBSrc, RegDest, PCsrc1, PCsrc2

IR → MemWrite, RegWrite

IR → New combinational logic

S → 1-bit Toggle FF

I-fetch / Execute

PCen, IRe, AddrSrc
Clock Rate vs CPI

\[ t_{C-Princeton} > \max \{ t_M, t_{RF} + t_{ALU} + t_M + t_{WB} \} \]
\[ t_{C-Princeton} > t_{RF} + t_{ALU} + t_M + t_{WB} \]
\[ t_{C-Harvard} > t_M + t_{RF} + t_{ALU} + t_M + t_{WB} \]

Suppose \( t_M >> t_{RF} + t_{ALU} + t_{WB} \)

\[ t_{C-Princeton} = 0.5 \times t_{C-Harvard} \]

\[ CPI_{Princeton} = 2 \]
\[ CPI_{Harvard} = 1 \]

No difference in performance!

Is it possible to design a controller for the Princeton architecture with CPI < 2?

\[ CPI = \text{Clock cycles Per Instruction} \]

Stay tuned!