Hardwired, Non-pipelined ISA Implementation

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

• 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!

February 18, 2021
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

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

- 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, ...

  ![Mux Diagram]
  ![Demux Diagram]
  ![Decoder Diagram]
  ![ALU Diagram]

  \[ \text{OpSelect} \quad \text{- Add, Sub, ...} \]
  \[ \text{- And, Or, Xor, Not, ...} \]
  \[ \text{- GT, LT, EQ, Zero, ...} \]

  \[ \text{Result} \quad \text{Comp?} \]

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

  ![Flipflop Diagram]
  ![Clock Diagram]

  \[ \text{Edge-triggered: Data is sampled at the rising edge} \]
Register Files

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$^2$
  - 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
    \[(the \ write \ address \ and \ data \ must \ be \ stable \ at \ the \ clock \ edge)\]

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
Instruction Execution

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)
Datapath: Reg-Imm ALU Instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>2120 16 15 0</td>
</tr>
</tbody>
</table>

rt ← (rs) op immediate
Conflicts in Merging Datapath

Introduce muxes

<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>opcode</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>func</td>
</tr>
</tbody>
</table>
| rs    | rs| rt| immediate

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

The diagram illustrates the datapath for ALU instructions, showing the flow of data and control signals. The instruction fetches data from memory, with the address and instruction fields. The Opcode, which is 6 bits long, is used to select the operation. The register file provides the source operands (rs and rt) and the destination register (rd). The immediate field is either a register or an immediate value, depending on the operation. The ALU performs the specified operation, and the result is written back to the register file. The control signals include RegWrite, which enables writing to the register file, and ALUControl, which controls the ALU operation. The diagram also includes the ExtSel, OpSel, and BSrcReg/Imm signals for specifying the operation and data sources.
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
### MIPS Control Instructions

#### Conditional (on GPR) PC-relative branch

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>offset</th>
</tr>
</thead>
</table>

- BEQZ, BNEZ

#### Unconditional register-indirect jumps

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
</tr>
</thead>
</table>

- JR, JALR

#### Unconditional absolute jumps

<table>
<thead>
<tr>
<th>opcode</th>
<th>target</th>
</tr>
</thead>
</table>

- J, JAL

<table>
<thead>
<tr>
<th>Target PC</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ</td>
<td>PC+4+offset*4</td>
</tr>
<tr>
<td>BNEZ</td>
<td>PC+4+offset*4</td>
</tr>
<tr>
<td>JR, JALR</td>
<td>(rs)</td>
</tr>
<tr>
<td>J, JAL</td>
<td>PC[31:28]</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)

[Diagram of a computer's pipeline with labels such as PCSrc, RegWrite, MemWrite, WBSrc, Addr, Inst, Memory, Add, ALU, ALU Control, PC, Inst, Memory, Data, we, rs1, rs2, ws, wd, rd1, rd2, GPRs, we, Addr, data, wdata, clk, zero?]

February 18, 2021
Register-Indirect Jumps (JR)
Register-Indirect Jump-&-Link (JALR)
Absolute Jumps (J, JAL)
Harvard-Style Datapath for MIPS
Hardwired Control is pure Combinational Logic

- ExtSel
- BSrc
- OpSel
- MemWrite
- WBSrc
- RegDst
- RegWrite
- PCSrc

op code

zero?
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

+ 0?
# 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>Reg</td>
<td>Func</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>ALUi</td>
<td>sExt16</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rt</td>
<td>pc+4</td>
</tr>
<tr>
<td>ALUiu</td>
<td>uExt16</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rt</td>
<td>pc+4</td>
</tr>
<tr>
<td>LW</td>
<td>sExt16</td>
<td>Imm</td>
<td>+</td>
<td>no</td>
<td>yes</td>
<td>Mem</td>
<td>rt</td>
<td>pc+4</td>
</tr>
<tr>
<td>SW</td>
<td>sExt16</td>
<td>Imm</td>
<td>+</td>
<td>yes</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>BEQZ&lt;sub&gt;z=0&lt;/sub&gt;</td>
<td>sExt16</td>
<td>*</td>
<td>0?</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>br</td>
</tr>
<tr>
<td>BEQZ&lt;sub&gt;z=1&lt;/sub&gt;</td>
<td>sExt16</td>
<td>*</td>
<td>0?</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>J</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>jabs</td>
</tr>
<tr>
<td>JAL</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>R31</td>
<td>jabs</td>
</tr>
<tr>
<td>JR</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>rind</td>
</tr>
<tr>
<td>JALR</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>R31</td>
<td>rind</td>
</tr>
</tbody>
</table>

BSrc = Reg / Imm  
WBSrc = ALU / Mem / PC  
RegDst = rt / rd / R31  
PCSrc = 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

$$\Rightarrow t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB}$$

- 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?

At least the instruction fetch and a Load (or Store) cannot be executed in the same cycle

Structural hazard
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**
  - **op code**
  - **zero?**

- **old combinational logic (Harvard)**
  - ExtSel, BSrc, OpSel, WBSrc, RegDest, PCsrc1, PCsrc2
  - MemWrite
  - RegWrite

- **new combinational logic**
  - Wen
  - PCen
  - IREN
  - AddrSrc

---

*1-bit Toggle FF*

*I-fetch / Execute*
Clock Rate vs CPI

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

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

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

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

*No difference in performance!*

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

\( CPI = \) Clock cycles Per Instruction

*Stay tuned!*