CIS 429/529

Single-cycle ISA Implementation I

Prof. Michel A. Kinsy
Instruction Set Architecture (ISA)

- ISA is the hardware/software interface
  - Defines data types
  - Defines set of programmer visible state
  - Defines instruction semantics (operations, sequencing)
  - Defines instruction format (bit encoding)
  - Examples: MIPS, Alpha, x86, IBM 360, VAX, ARM, JVM
Many possible implementations of one ISA

- 360 implementations: model 30 (c. 1964), z900 (c. 2001)
- x86 implementations: 8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4, Core i7, AMD Athlon, AMD Opteron, Transmeta Crusoe, SoftPC
- MIPS implementations: R2000, R4000, R10000, ...
- JVM: HotSpot, PicoJava, ARM Jazelle, ...
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

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

• Combinational circuits
  ‣ Mux, Demux, Decoder, ALU, ...

  \[
  \begin{align*}
  &\text{Mux, Demux, Decoder, ALU, ...} \\
  &\text{Mux} \\
  &\text{Demux} \\
  &\text{Decoder}
  \end{align*}
\]

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

  \[
  \begin{align*}
  &\text{Flipflop, Register, Register file, SRAM, DRAM} \\
  &\text{ff} \\
  &\text{Clk}
  \end{align*}
\]

  ‣ Edge-triggered: Data is sampled at the rising edge
Register Files

• No timing issues in reading a selected register
Register File Implementation

- Register files with a large number of ports are difficult to design
  - 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)
  ‣ a Write is performed at the rising clock edge
  • if it is enabled then the write address and data must be stable at the clock edge
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
The MIPS ISA

• 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
The MIPS ISA

• 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
7. Computation of the address of the next instruction
Datapath: Reg-Reg ALU Instructions

```
6        5       5          6
0        rs       rt       0       func       rd

!(rs) func (rt)

31        26  25      21 20     16 15       11             5             0

0x4       Add

Inst. Memory

addr

inst

0x4

Add

reg

inst<25:21>

inst<20:16>

inst<15:11>

inst<5:0>

OpCode

PC

clk

Add

RegWrite

clk

rd1

rs1

rs2

we

GPRs

ws

wd

rd2

ALU

z

ALU Control

rd \leftarrow (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>0x4 Add</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25 21 20 16 15 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
rt ← (rs) op immediate
```

![Datapath diagram](image)
Conflicts in Merging Datapath

- **OpCode**: 0x4 Add
- **rs**: 0
- **rt**: 5
- **rd**: 5
- **func**: 6

- **Inst. Memory**
- **PC**: clk
- **Addr**: inst

- **Add**
- **RegWrite**
- **clk**: we
- **rs1**: inst<25:21>
- **rs2**: inst<20:16>
- **Imm Ext**: inst<15:11>
- **ExtSel**: inst<15:0>
- **Inst<31:26>**
- **Inst<25:21>**
- **Inst<20:16>**
- **Inst<15:11>**

- **GPRs**
- **rd1**: we
- **rd2**: ws
- **wd**: rd

- **ALU**
- **z**: we
- **ALU Control**

**Rules**:
- \( rd \leftarrow (rs) \text{ func } (rt) \)
- \( rt \leftarrow (rs) \text{ op immediate} \)
Conflicts in Merging Datapath

0x4 Add

OpCode

inst<25:21>

inst<20:16>

inst<15:11>

inst<15:0>

inst<31:26>

inst<5:0>

RegWrite

clk

rs

rt

rd

func

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 I influence)
    • read-only program memory
    • read/write data memory
  • Princeton style: the same (von Neumann’s influence)
    ‣ single read/write memory for program and data
      • Executing a Load or Store instruction requires accessing the memory more than once
Harvard Architecture

Add

RegWrite

MemWrite

WBSrc

ALU / Mem

clk

addr

inst

Inst. Memory

rd1

rs1

rs2

Imm Ext

ALU

OpCode

RegDst

ExtSel

OpSel

BSrc

clk

disp

"base"

V we

rd2

GPRs

z

addr

wdata

rdata

Data Memory

we

addr
MIPS Control Instructions

- Conditional (on GPR) PC-relative branch
  
<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></td>
<td>offset</td>
</tr>
</tbody>
</table>
  
  BEQZ, BNEZ

- Unconditional register-indirect jumps
  
<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></td>
<td>offset</td>
</tr>
</tbody>
</table>
  
  JR, JALR

- Unconditional absolute jumps
  
<table>
<thead>
<tr>
<th>6</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>target</td>
</tr>
</tbody>
</table>
  
  J, JAL

- PC-relative branches add offset×4 to PC+4 to calculate the target address (offset is in words): ±128 KB range
MIPS Control Instructions

- Absolute jumps append targetx4 to PC<31:28> to calculate the target address: 256 MB range
- jump-&-link stores PC+4 into the link register (R31)
- All Control Transfers are delayed by 1 instruction
- We will worry about the branch delay slot in later lectures
Conditional Branches (BEQZ, BNEZ)
Register-Indirect Jumps (JR)
Register-Indirect Jump-&-Link (JALR)
Absolute Jumps (J, JAL)