Instruction Set Architecture (ISA) versus Implementation

• ISA is the hardware/software interface
  – Defines set of programmer visible state
  – Defines instruction format (bit encoding) and instruction semantics
  – Examples: *MIPS*, *x86*, *IBM 360*, *JVM*

• Many possible implementations of one ISA
  – 360 implementations: model 30 (c. 1964), z990 (c. 2004)
  – MIPS implementations: *R2000*, *R4000*, *R10000*, ...
  – JVM: *HotSpot*, *PicoJava*, *ARM Jazelle*, ...
**Styles of ISA**

- Accumulator
- Stack
- GPR
- CISC
- RISC
- VLIW
- Vector

- Boundaries are fuzzy, and hybrids are common
  - E.g., 8086/87 is hybrid accumulator-GPR-stack ISA
  - Many ISAs have added vector extensions

**Styles of Implementation**

- Microcoded
- Unpipelined single cycle
- Hardwired in-order pipeline
- Out-of-order pipeline with speculative execution and register renaming
- Software interpreter
- Binary translator
- Just-in-Time compiler
Last Time in Lecture 2

- Stack machines popular to simplify High-Level Language (HLL) implementation
  - Algol-68 & Burroughs B5000, Forth machines, Occam & Transputers, Java VMs & Java Interpreters

- General-purpose register machines provide greater efficiency with better compiler technology (or assembly coding)
  - Compilers can explicitly manage fastest level of memory hierarchy (registers)

- Microcoding was a straightforward methodical way to implement machines with low gate count

A Bus-based Datapath for MIPS

Microinstruction: register to register transfer (17 control signals)
MIPS Microcontroller: first attempt

Opcode zero?  Busy (memory)

latching the inputs may cause a one-cycle delay

ROM size?
= \(2^{(\text{opcode}+\text{status}+s)}\) words

Word size?
= control+s bits

Control store has to be fast \(\Rightarrow\) expensive

- Reduce the ROM height (= address bits)
  - reduce inputs by extra external logic
    each input bit doubles the size of the control store
  - reduce states by grouping opcodes
    find common sequences of actions
  - condense input status bits
    combine all exceptions into one, i.e., exception/no-exception

- Reduce the ROM width
  - restrict the next-state encoding
    Next, Dispatch on opcode, Wait for memory, ...
  - encode control signals (vertical microcode)
**MIPS Controller V2**

- **µJumpType** = `next`, `spin`, `fetch`, `dispatch`, `feqz`, `fnez`

- **Jump Logic**
  
  \[
  \mu\text{PCSrc} = \begin{cases} 
  \text{next} & \rightarrow \mu\text{PC} + 1 \\
  \text{spin} & \rightarrow \text{if (busy) then } \mu\text{PC} \text{ else } \mu\text{PC} + 1 \\
  \text{fetch} & \rightarrow \text{absolute} \\
  \text{dispatch} & \rightarrow \text{op-group} \\
  \text{feqz} & \rightarrow \text{if (zero) then absolute else } \mu\text{PC} + 1 \\
  \text{fnez} & \rightarrow \text{if (zero) then } \mu\text{PC} + 1 \text{ else absolute} 
  \end{cases}
  \]
### Instruction Fetch & ALU: \textit{MIPS-Controller-2}

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch(_0)</td>
<td>MA $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch(_1)</td>
<td>IR $\leftarrow$ Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch(_2)</td>
<td>A $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch(_3)</td>
<td>PC $\leftarrow$ A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU(_0)</td>
<td>B $\leftarrow$ Reg[rt]</td>
<td>next</td>
</tr>
<tr>
<td>ALU(_1)</td>
<td>Reg[rd]$\leftarrow$func(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>ALU(_i_0)</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU(_i_1)</td>
<td>B $\leftarrow$ sExt(_{16})(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>ALU(_i_2)</td>
<td>Reg[rd]$\leftarrow$ Op(A,B)</td>
<td>fetch</td>
</tr>
</tbody>
</table>

### Load & Store: \textit{MIPS-Controller-2}

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW(_0)</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>LW(_1)</td>
<td>B $\leftarrow$ sExt(_{16})(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>LW(_2)</td>
<td>MA $\leftarrow$ A+B</td>
<td>next</td>
</tr>
<tr>
<td>LW(_3)</td>
<td>Reg[rt]$\leftarrow$ Memory</td>
<td>spin</td>
</tr>
<tr>
<td>LW(_4)</td>
<td></td>
<td>fetch</td>
</tr>
<tr>
<td>SW(_0)</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>SW(_1)</td>
<td>B $\leftarrow$ sExt(_{16})(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>SW(_2)</td>
<td>MA $\leftarrow$ A+B</td>
<td>next</td>
</tr>
<tr>
<td>SW(_3)</td>
<td>Memory $\leftarrow$ Reg[rt]</td>
<td>spin</td>
</tr>
<tr>
<td>SW(_4)</td>
<td></td>
<td>fetch</td>
</tr>
</tbody>
</table>
### Branches: MIPS-Controller-2

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ(_0)</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ(_1)</td>
<td></td>
<td>fnez</td>
</tr>
<tr>
<td>BEQZ(_2)</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ(_3)</td>
<td>B ← sExt(_{16})(Imm&lt;&lt;2)</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ(_4)</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
<tr>
<td>BNEZ(_0)</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ(_1)</td>
<td></td>
<td>feqz</td>
</tr>
<tr>
<td>BNEZ(_2)</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ(_3)</td>
<td>B ← sExt(_{16})(Imm&lt;&lt;2)</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ(_4)</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
</tbody>
</table>

### Jumps: MIPS-Controller-2

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>J(_0)</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>J(_1)</td>
<td>B ← IR</td>
<td>next</td>
</tr>
<tr>
<td>J(_2)</td>
<td>PC ← JumpTarg(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>JR(_0)</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>JR(_1)</td>
<td>PC ← A</td>
<td>fetch</td>
</tr>
<tr>
<td>JAL(_0)</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>JAL(_1)</td>
<td>Reg[31] ← A</td>
<td>next</td>
</tr>
<tr>
<td>JAL(_2)</td>
<td>B ← IR</td>
<td>next</td>
</tr>
<tr>
<td>JAL(_3)</td>
<td>PC ← JumpTarg(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>JALR(_0)</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>JALR(_1)</td>
<td>B ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>JALR(_2)</td>
<td>Reg[31] ← A</td>
<td>next</td>
</tr>
<tr>
<td>JALR(_3)</td>
<td>PC ← B</td>
<td>fetch</td>
</tr>
</tbody>
</table>
Implementing Complex Instructions

Mem-Mem ALU Instructions:
*MIPS-Controller-2*

<table>
<thead>
<tr>
<th>Mem-Mem ALU op</th>
<th>M[(rd)] ← M[(rs)] op M[(rt)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUMM₀</td>
<td>MA ← Reg[rs]</td>
</tr>
<tr>
<td>ALUMM₁</td>
<td>A ← Memory</td>
</tr>
<tr>
<td>ALUMM₂</td>
<td>MA ← Reg[rt]</td>
</tr>
<tr>
<td>ALUMM₃</td>
<td>B ← Memory</td>
</tr>
<tr>
<td>ALUMM₄</td>
<td>MA ← Reg[rd]</td>
</tr>
<tr>
<td>ALUMM₅</td>
<td>Memory ← func(A,B)</td>
</tr>
<tr>
<td>ALUMM₆</td>
<td></td>
</tr>
</tbody>
</table>

Complex instructions usually do not require datapath modifications in a microprogrammed implementation
-- only extra space for the control program

Implementing these instructions using a hardwired controller is difficult without datapath modifications
Performance Issues

Microprogrammed control  
⇒ multiple cycles per instruction

Cycle time
\[ t_C > \max(t_{\text{reg-reg}}, t_{\text{ALU}}, t_{\mu\text{ROM}}) \]

Suppose \( 10 \times t_{\mu\text{ROM}} < t_{\text{RAM}} \)

*Good performance, relative to a single-cycle hardwired implementation, can be achieved even with a CPI of 10*

Horizonal vs Vertical \(\mu\)Code

- Horizontal \(\mu\)code has wider \(\mu\)instructions
  - Multiple parallel operations per \(\mu\)instruction
  - Fewer steps per macroinstruction
  - Sparser encoding ⇒ more bits

- Vertical \(\mu\)code has narrower \(\mu\)instructions
  - Typically a single datapath operation per \(\mu\)instruction
    - separate \(\mu\)instruction for branches
  - More steps to per macroinstruction
  - More compact ⇒ less bits

- Nanocoding
  - Tries to combine best of horizontal and vertical \(\mu\)code
Nanocoding

Exploits recurring control signal patterns in µcode, e.g.,

ALU₀ A ← Reg[rs]
...
ALUᵢ₀ A ← Reg[rs]
...

- MC68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

Microprogramming in IBM 360

<table>
<thead>
<tr>
<th></th>
<th>M30</th>
<th>M40</th>
<th>M50</th>
<th>M65</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datapath width (bits)</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>µinst width (bits)</td>
<td>50</td>
<td>52</td>
<td>85</td>
<td>87</td>
</tr>
<tr>
<td>µcode size (K µinsts)</td>
<td>4</td>
<td>4</td>
<td>2.75</td>
<td>2.75</td>
</tr>
<tr>
<td>µstore technology</td>
<td>CCROS</td>
<td>TCROS &lt;br&gt;BCROS</td>
<td>BCROS</td>
<td></td>
</tr>
<tr>
<td>µstore cycle (ns)</td>
<td>750</td>
<td>625</td>
<td>500</td>
<td>200</td>
</tr>
<tr>
<td>memory cycle (ns)</td>
<td>1500</td>
<td>2500</td>
<td>2000</td>
<td>750</td>
</tr>
<tr>
<td>Rental fee ($K/month)</td>
<td>4</td>
<td>7</td>
<td>15</td>
<td>35</td>
</tr>
</tbody>
</table>

Only the fastest models (75 and 95) were hardwired
Microcode Emulation

• IBM initially miscalculated the importance of software compatibility with earlier models when introducing the 360 series
• Honeywell stole some IBM 1401 customers by offering translation software (“Liberator”) for Honeywell H200 series machine
• IBM retaliated with optional additional microcode for 360 series that could emulate IBM 1401 ISA, later extended for IBM 7000 series
  – one popular program on 1401 was a 650 simulator, so some customers ran many 650 programs on emulated 1401s
    – (650 simulated on 1401 emulated on 360)

Microprogramming thrived in the Seventies

• Significantly faster ROMs than magnetic core memory or DRAMs were available
• For complex instruction sets (CISC), datapath and controller were cheaper and simpler
• New instructions, e.g., floating point, could be supported without datapath modifications
• Fixing bugs in the controller was easier
• ISA compatibility across various models could be achieved easily and cheaply

Except for the cheapest and fastest machines, all computers were microprogrammed
Writable Control Store (WCS)

- Implement control store in RAM not ROM
  - MOS SRAM memories now became almost as fast as control store
    (core memories/DRAMs were 2-10x slower)
  - Bug-free microprograms difficult to write

- User-WCS provided as option on several minicomputers
  - Allowed users to change microcode for each processor

- User-WCS failed
  - Little or no programming tools support
  - Difficult to fit software into small space
  - Microcode control tailored to original ISA, less useful for others
  - Large WCS part of processor state - expensive context switches
  - Protection difficult if user can change microcode
  - Virtual memory required restartable microcode

Microprogramming: early Eighties

- Evolution bred more complex micro-machines
  - Ever more complex CISC ISAs led to need for subroutine and call
    stacks in \( \mu \text{code} \)
  - Need for fixing bugs in control programs was in conflict with read-only
    nature of \( \mu \text{ROM} \)
  - --> WCS (B1700, QMachine, Intel i432, ...)

- With the advent of VLSI technology assumptions about
  ROM & RAM speed became invalid

- Better compilers made complex instructions less important

- Use of numerous micro-architectural innovations, e.g.,
  pipelining, caches and buffers, made multiple-cycle
  execution of reg-reg instructions unattractive
Microprogramming in Modern Usage

- *Microprogramming is far from extinct*

- Played a crucial role in micros of the Eighties
  
  *DEC uVAX, Motorola 68K series, Intel 386 and 486*

- Microcode pays an assisting role in most modern micros (*AMD Athlon, Intel Core 2 Duo, IBM PowerPC*)
  
  - Most instructions are executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke the microcode engine

- *Patchable* microcode common for post-fabrication bug fixes, e.g. Intel Pentiums load μcode patches at bootup

---

From CISC to RISC

- Use fast RAM to build fast instruction cache of user-visible instructions, not fixed hardware microroutines
  
  - Can change contents of fast instruction memory to fit what application needs right now

- Use simple ISA to enable hardwired pipelined implementation
  
  - Most compiled code only used a few of the available CISC instructions
  - Simpler encoding allowed pipelined implementations

- Further benefit with integration
  
  - In early ‘80s, could finally fit 32-bit datapath + small caches on a single chip
  - No chip crossings in common case allows faster operation
Nanocoding

Exploits recurring control signal patterns in μcode, e.g.,

\[ \text{ALU}_0 \ A \leftarrow \text{Reg}[rs] \]

... \[ \text{ALU}_i \ A \leftarrow \text{Reg}[rs] \]

- MC68000 had 17-bit μcode containing either 10-bit μjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

CDC 6600 \textit{Seymour Cray, 1964}

- A fast pipelined machine with 60-bit words
- Ten functional units
  - Floating Point: adder, multiplier, divider
  - Integer: adder, multiplier
- Hardwired control (no microcoding)
- Dynamic scheduling of instructions using a scoreboard
- Ten Peripheral Processors for Input/Output
  - a fast time-shared 12-bit integer ALU
- Very fast clock, 10MHz
- Novel freon-based technology for cooling
CDC 6600: Datapath

- Separate instructions to manipulate three types of reg.
  - 8 60-bit data registers (X)
  - 8 18-bit address registers (A)
  - 8 18-bit index registers (B)

- All arithmetic and logic instructions are reg-to-reg
  \[
  \text{opcode}\ i\ j\ k\quad \text{R}i \leftarrow \text{(R}j\) op \ (Rk)\]

- Only Load and Store instructions refer to memory!
  \[
  \text{opcode}\ i\ j\ disp\quad \text{R}i \leftarrow \text{M}[(\text{R}j) + \text{disp}]\]

- Touching address registers 1 to 5 initiates a load
- 6 to 7 initiates a store
  - very useful for vector operations
CDC6600: Vector Addition

B0 ← - n
loop: JZE B0, exit
A0 ← B0 + a0  \( \text{load } X0 \)
A1 ← B0 + b0  \( \text{load } X1 \)
X6 ← X0 + X1
A6 ← B0 + c0  \( \text{store } X6 \)
B0 ← B0 + 1
jump loop

Ai = address register
Bi = index register
Xi = data register

CDC6600 ISA designed to simplify high-performance implementation

- Use of three-address, register-register ALU instructions simplifies pipelined implementation
  - No implicit dependencies between inputs and outputs
- Decoupling setting of address register (Ar) from retrieving value from data register (Xr) simplifies providing multiple outstanding memory accesses
  - Software can schedule load of address register before use of value
  - Can interleave independent instructions inbetween
- CDC6600 has multiple parallel but unpipelined functional units
  - E.g., 2 separate multipliers
- Follow-on machine CDC7600 used pipelined functional units
  - Foreshadows later RISC designs
“Iron Law” of 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>

CS152 Administrivia

- Check web site for new calendar, quiz dates should not change
  - Feb 17 and Mar 17 lecture in 320 Soda
  - All other lectures in 306 Soda (here)
- PS1 and Lab 1 available now or tomorrow
  - PS 1 / Lab 1 due Tuesday February 10
- Section tomorrow (Friday 1/30) 12-1pm 258 Dwinelle
  - Covers lab 1 details
- Quiz 1 on Thursday Feb 12
**Hardware Elements**

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

![Combinational circuits diagram]

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

![Synchronous state elements diagram]

*Edge-triggered: Data is sampled at the rising edge*

---

**Register Files**

- Reads are combinational

![Register file diagram]
Register File Implementation

- Register files with a large number of ports are difficult to design
  - Almost all MIPS 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
  \[ \Rightarrow \text{the write address and data must be stable at the clock edge} \]

Later in the course we will present a more realistic model of memory
Implementing MIPS:

Single-cycle per instruction
datapath & control logic
(Should be review of CS61C)

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 and register fetch
3. ALU operation
4. memory operation (optional)
5. write back

and the computation of the address of the next instruction

Datapath: Reg-Reg ALU Instructions

RegWrite Timing?

rd ← (rs) func (rt)
Datapath: Reg-Imm ALU Instructions

Conflicts in Merging Datapath

Introduce muxes
**Datapath for ALU Instructions**

1. Opcode, rs, rt, immediate, and rd fields.
2. Func field indicating the operation.
3. ALU and control signals for the operation.

**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:
  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
Load/Store Instructions: *Harvard Datapath*

- **OpCode**
- **RegDst**
- **ExtSel**
- **ALU Control**
- **Addressing Mode**
- 
  \[
  \text{addressing mode} = (\text{rs}) + \text{displacement}
  \]
- \( \text{rs} \) is the base register
- \( \text{rt} \) is the destination of a Load or the source for a Store

MIPS Control Instructions

- **Conditional (on GPR) PC-relative branch**
  - \( \text{BEQZ, BNEZ} \)

- **Unconditional register-indirect jumps**
  - \( \text{JR, JALR} \)

- **Unconditional absolute jumps**
  - \( \text{J, JAL} \)

- PC-relative branches add offset×4 to PC+4 to calculate the target address (offset is in words): ±128 KB range
- Absolute jumps append target×4 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 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
### ALU Control & Immediate Extension

- **Inst<31:26> (Opcode)**
- **Inst<5:0> (Func)**
- **ALUop**
- **ExtSel** (\(s_{Ext_{16}}, u_{Ext_{16}}, \text{High}_{16}\))
- **OpSel** (Func, Op, +, 0?)

**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>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>(s_{Ext_{16}})</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>(u_{Ext_{16}})</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>(s_{Ext_{16}})</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>(s_{Ext_{16}})</td>
<td>Imm</td>
<td>+</td>
<td>yes</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>BEQZ(_{z=0})</td>
<td>(s_{Ext_{16}})</td>
<td>*</td>
<td>0?</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>br</td>
</tr>
<tr>
<td>BEQZ(_{z=1})</td>
<td>(s_{Ext_{16}})</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
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

\[ t_C > t_{IF} + t_{RF} + t_{ALU} + t_{DM} + t_{RWB} \]

- 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

Pipelined Datapath

Clock period can be reduced by dividing the execution of an instruction into multiple cycles

\[ t_C > \max \{ t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW} \} \quad (= t_{DM} \text{ probably}) \]

However, CPI will increase unless instructions are pipelined
How to divide the datapath into stages

Suppose memory is significantly slower than other stages. In particular, suppose

\[ t_{IM} = 10 \text{ units} \]
\[ t_{DM} = 10 \text{ units} \]
\[ t_{ALU} = 5 \text{ units} \]
\[ t_{RF} = 1 \text{ unit} \]
\[ t_{RW} = 1 \text{ unit} \]

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance

Alternative Pipelining

Write-back stage takes much less time than other stages. Suppose we combined it with the memory phase.

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

⇒ *increase the critical path by 10%*
Summary

- Microcoding became less attractive as gap between RAM and ROM speeds reduced
- Complex instruction sets difficult to pipeline, so difficult to increase performance as gate count grew
- Iron Law explains architecture design space
  - Trade instruction/program, cycles/instruction, and time/cycle
- Load-Store RISC ISAs designed for efficient pipelined implementations
  - Very similar to vertical microcode
  - Inspired by earlier Cray machines
- MIPS ISA will be used in class and problems, SPARC in lab (two very similar ISAs)

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252