CS 152 Computer Architecture and Engineering

Lecture 2 - Simple Machine Implementations

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

1/24/2013
CS152, Spring 2013

Last Time in Lecture 1

- Computer Architecture >> ISAs and RTL
  - CS152 is about interaction of hardware and software, and design of appropriate abstraction layers
- Technology and Applications shape Computer Architecture
  - History provides lessons for the future
- First 130 years of CompArch, from Babbage to IBM 360
  - Move from calculators (no conditionals) to fully programmable machines
  - Rapid change started in WWII (mid-1940s), move from electro-mechanical to pure electronic processors
- Cost of software development becomes a large constraint on architecture (need compatibility)
- IBM 360 introduces notion of “family of machines” running same ISA but very different implementations
  - Six different machines released on same day (April 7, 1964)
  - “Future-proofing” for subsequent generations of machine

1/24/2013
CS152, Spring 2013
IBM 360: Initial Implementations

<table>
<thead>
<tr>
<th></th>
<th>Model 30</th>
<th>. . .</th>
<th>Model 70</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>8K - 64 KB</td>
<td></td>
<td>256K - 512 KB</td>
</tr>
<tr>
<td>Datapath</td>
<td>8-bit</td>
<td></td>
<td>64-bit</td>
</tr>
<tr>
<td>Circuit Delay</td>
<td>30 nsec/level</td>
<td></td>
<td>5 nsec/level</td>
</tr>
<tr>
<td>Local Store</td>
<td>Main Store</td>
<td></td>
<td>Transistor Registers</td>
</tr>
<tr>
<td>Control Store</td>
<td>Read only 1µsec</td>
<td></td>
<td>Conventional circuits</td>
</tr>
</tbody>
</table>

IBM 360 instruction set architecture (ISA) completely hid the underlying technological differences between various models.

Milestone: The first true ISA designed as portable hardware-software interface!

IBM 360 Survives Today: z12 Mainframe Processor

6 Cores @ 5.5 GHz

Special-purpose coprocessors on each core

48MB of Level-3 cache on chip

32nm SOI Technology
2.75 billion transistors
23.7mm x 25.2mm
15 layers of metal
7.68 miles of wiring!
10,000 power pins (!)
1,071 I/O pins

[From IBM HotChips24 presentation, August 28, 2012]
Instruction Set Architecture (ISA)

- The contract between software and hardware
- Typically described by giving all the programmer-visible state (registers + memory) plus the semantics of the instructions that operate on that state
- IBM 360 was first line of machines to separate ISA from implementation (aka. microarchitecture)
- Many implementations possible for a given ISA
  - E.g., the Soviets build code-compatible clones of the IBM360, as did Amdahl after he left IBM.
  - E.g.2., today you can buy AMD or Intel processors that run the x86-64 ISA.
  - E.g.3: many cellphones use the ARM ISA with implementations from many different companies including TI, Qualcomm, Samsung, Marvell, etc.

ISA to Microarchitecture Mapping

- ISA often designed with particular microarchitectural style in mind, e.g.,
  - Accumulator ⇒ hardwired, unpipelined
  - CISC ⇒ microcoded
  - RISC ⇒ hardwired, pipelined
  - VLIW ⇒ fixed-latency in-order parallel pipelines
  - JVM ⇒ software interpretation
- But can be implemented with any microarchitectural style
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Simics: Software-interpreted SPARC RISC machine
  - ARM Jazelle: A hardware JVM processor
  - This lecture: a microcoded RISC-V machine
Today, Microprogramming

- To show how to build very small processors with complex ISAs
- To help you understand where CISC* machines came from
- Because still used in common machines (IBM360, x86, PowerPC)
- As a gentle introduction into machine structures
- To help understand how technology drove the move to RISC*

* “CISC”/“RISC” names much newer than style of machines they refer to.

---

Microarchitecture: Implementation of an ISA

Structure: How components are connected.  
Behavior: How data moves between components

Controller

---

Status lines

Control Points

Data path
Microcontrol Unit  *Maurice Wilkes, 1954*

First used in EDSAC-2, completed 1958

Embed the control logic state table in a memory array

**Microcoded Microarchitecture**

- **μcontroller (ROM)**: holds fixed microcode instructions
- **Datapath**: contains control lines to ALU, MUXs, Registers
- **Memory (RAM)**: holds user program written in macrocode instructions (e.g., x86, RISC-V, etc.)
RISC-V ISA

- New RISC design from UC Berkeley
- Realistic & complete ISA, but open & small
- Not over-architected for a certain implementation style
- Both 32-bit and 64-bit address space variants
  - RV32 and RV64
- Designed for multiprocessing
- Efficient instruction encoding
- Easy to subset/extend for education/research
- Techreport with RISC-V spec available on class website

- We’ll be using 32-bit RISC-V this semester in lectures and labs, very similar to MIPS you saw in CS61C

RV32 Processor State

Program counter (pc)
32x32-bit integer registers (x0-x31)
  - x0 always contains a 0
32 floating-point (FP) registers (f0-f31)
  - each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)
FP status register (fsr), used for FP rounding mode & exception reporting
### RISC-V Instruction Encoding

- Can support variable-length instructions.
- Base instruction set (RV32) always has fixed 32-bit instructions lowest two bits $= 11_2$
- All branches and jumps have targets at 16-bit granularity (even in base ISA where all instructions are fixed 32 bits)

---

### RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>Destination Reg.</th>
<th>Reg. Source 1</th>
<th>Reg. Source 2</th>
<th>Additional opcode bits/immediate</th>
<th>7-bit opcode field (but low 2 bits $= 11_2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>rs1</td>
<td>rs2</td>
<td>funct10</td>
<td>opcode</td>
</tr>
<tr>
<td>rd</td>
<td>rs1</td>
<td>rs2</td>
<td>funct5</td>
<td>opcode</td>
</tr>
<tr>
<td>rd</td>
<td>LUI</td>
<td>immediate[19:0]</td>
<td></td>
<td>opcode</td>
</tr>
<tr>
<td>rd</td>
<td>jump offset 24:0</td>
<td></td>
<td></td>
<td>opcode</td>
</tr>
</tbody>
</table>

1/24/2013 CS152, Spring 2013
R-Type/I-Type/R4-Type Formats

Reg-Reg ALU operations

Reg-Imm ALU operations

12-bit signed immediate

Load instructions, (rs1 + immediate) addressing

Only used for floating-point fused multiply-add

Reg. Source 3

B-Type

12-bit signed immediate split across two fields

Branches, compare two registers, PC+(immediate<<1) target

(Branches do not have delay slot)

Store instructions, (rs1 + immediate) addressing, rs2 data
L-Type

- Writes 20-bit immediate to top of destination register.
- Used to build large immediates.
- 12-bit immediates are signed, so have to account for sign when building 32-bit immediates in 2-instruction sequence (LUI high-20b, ADDI low-12b)

J-Type

- “J” Unconditional jump, PC+offset target
- “JAL” Jump and link, also writes PC+4 to x1
- Offset scaled by 1-bit left shift – can jump to 16-bit instruction boundary (Same for branches)
Data Formats and Memory Addresses

Data formats:
- 8-b Bytes, 16-b Half words, 32-b words and 64-b double words

Some issues
- **Byte addressing**
  - Little Endian
  - Most Significant Byte
    - 3 2 1 0
  - Big Endian
    - 0 1 2 3

- **Word alignment**

Suppose the memory is organized in 32-bit words.
Can a word address begin only at 0, 4, 8, ... ?

A Bus-based Datapath for RISC-V

Microinstruction: register to register transfer (17 control signals)

MA ← PC means RegSel = PC; enReg = yes; IdMA = yes
B ← Reg[rs2] means RegSel = rs2; enReg = yes; IdB = yes
Assumption: Memory operates independently and is slow as compared to Reg-to-Reg transfers (multiple CPU clock cycles per access)

Instruction Execution

Execution of a RISC-V instruction involves:

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back to register file (optional)
   + the computation of the next instruction address
Microprogram Fragments

\[
\begin{align*}
\text{instr fetch:} & \quad MA, A \leftarrow PC \\
& \quad PC \leftarrow A + 4 \\
& \quad IR \leftarrow \text{Memory} \\
& \quad \text{dispatch on Opcode} \\
& \quad \{ \text{can be treated as a macro} \}
\end{align*}
\]

\[
\begin{align*}
\text{ALU:} & \quad A \leftarrow \text{Reg}[rs1] \\
& \quad B \leftarrow \text{Reg}[rs2] \\
& \quad \text{Reg}[rd] \leftarrow \text{func}(A, B) \\
& \quad \text{do instruction fetch}
\end{align*}
\]

\[
\begin{align*}
\text{ALUi:} & \quad A \leftarrow \text{Reg}[rs1] \\
& \quad B \leftarrow \text{Imm} \\
& \quad \text{sign extension} \\
& \quad \text{Reg}[rd] \leftarrow \text{Opcode}(A, B) \\
& \quad \text{do instruction fetch}
\end{align*}
\]

Microprogram Fragments (cont.)

\[
\begin{align*}
\text{LW:} & \quad A \leftarrow \text{Reg}[rs1] \\
& \quad B \leftarrow \text{Imm} \\
& \quad MA \leftarrow A + B \\
& \quad \text{Reg}[rd] \leftarrow \text{Memory} \\
& \quad \text{do instruction fetch}
\end{align*}
\]

\[
\begin{align*}
\text{J:} & \quad A \leftarrow A - 4 \quad \text{Get original PC back in A} \\
& \quad B \leftarrow \text{IR} \\
& \quad PC \leftarrow \text{JumpTarg}(A, B) \\
& \quad \text{do instruction fetch} \\
& \quad \text{JumpTarg}(A, B) = (A + (B[31:7] \ll 1))
\end{align*}
\]

\[
\begin{align*}
\text{beq:} & \quad A \leftarrow \text{Reg}[rs1] \\
& \quad B \leftarrow \text{Reg}[rs2] \\
& \quad \text{if } A==B \text{ then go to bz-taken} \\
& \quad \text{do instruction fetch}
\end{align*}
\]

\[
\begin{align*}
\text{bz-taken:} & \quad A \leftarrow \text{PC} \\
& \quad A \leftarrow A - 4 \quad \text{Get original PC back in A} \\
& \quad B \leftarrow \text{BImm} \ll 1 \quad \text{BImm} = \text{IR}[31:27,16:10] \\
& \quad \text{PC} \leftarrow A + B \\
& \quad \text{do instruction fetch}
\end{align*}
\]
RISC-V Microcontroller: first attempt
pure ROM implementation

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

Word size?
= control+s bits

Microprogram in the ROM worksheet

<table>
<thead>
<tr>
<th>State</th>
<th>Op</th>
<th>zero?</th>
<th>busy</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch_0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>MA,A ← PC</td>
<td>fetch_1</td>
</tr>
<tr>
<td>fetch_1</td>
<td>*</td>
<td>*</td>
<td>yes</td>
<td>.....</td>
<td>fetch_1</td>
</tr>
<tr>
<td>fetch_1</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>IR ← Memory</td>
<td>fetch_2</td>
</tr>
<tr>
<td>fetch_2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fetch_2</th>
<th>ALU</th>
<th>*</th>
<th>*</th>
<th>PC ← A + 4</th>
<th>ALU_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU_0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← Reg[rs1]</td>
<td>ALU_1</td>
</tr>
<tr>
<td>ALU_1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>B ← Reg[rs2]</td>
<td>ALU_2</td>
</tr>
<tr>
<td>ALU_2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Reg[rd] ← func(A,B)</td>
<td>fetch_0</td>
</tr>
</tbody>
</table>

1/24/2013 CS152, Spring 2013
### Microprogram in the ROM

<table>
<thead>
<tr>
<th>State</th>
<th>Op</th>
<th>zero?</th>
<th>busy</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch₀</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>MA, A ← PC</td>
<td>fetch₁</td>
</tr>
<tr>
<td>fetch₁</td>
<td>*</td>
<td>*</td>
<td>yes</td>
<td>....</td>
<td>fetch₁</td>
</tr>
<tr>
<td>fetch₁</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>IR ← Memory</td>
<td>fetch₂</td>
</tr>
<tr>
<td>fetch₂ ALU</td>
<td>*</td>
<td>*</td>
<td></td>
<td>PC ← A + 4</td>
<td>ALU₀</td>
</tr>
<tr>
<td>fetch₂ ALU i</td>
<td>*</td>
<td>*</td>
<td></td>
<td>PC ← A + 4</td>
<td>ALU₀</td>
</tr>
<tr>
<td>fetch₂ LW</td>
<td>*</td>
<td>*</td>
<td></td>
<td>PC ← A + 4</td>
<td>LW₀</td>
</tr>
<tr>
<td>fetch₂ SW</td>
<td>*</td>
<td>*</td>
<td></td>
<td>PC ← A + 4</td>
<td>SW₀</td>
</tr>
<tr>
<td>fetch₂ J</td>
<td>*</td>
<td>*</td>
<td></td>
<td>PC ← A + 4</td>
<td>J₀</td>
</tr>
<tr>
<td>fetch₂ JAL</td>
<td>*</td>
<td>*</td>
<td></td>
<td>PC ← A + 4</td>
<td>JAL₀</td>
</tr>
<tr>
<td>fetch₂ JR</td>
<td>*</td>
<td>*</td>
<td></td>
<td>PC ← A + 4</td>
<td>JR₀</td>
</tr>
<tr>
<td>fetch₂ JAL R</td>
<td>*</td>
<td>*</td>
<td></td>
<td>PC ← A + 4</td>
<td>JALR₀</td>
</tr>
<tr>
<td>fetch₂ beq</td>
<td>*</td>
<td>*</td>
<td></td>
<td>PC ← A + 4</td>
<td>beq₀</td>
</tr>
<tr>
<td>... ALU₀</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← Reg[rs1]</td>
<td>ALU₁</td>
</tr>
<tr>
<td>ALU₁</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>B ← Reg[rs2]</td>
<td>ALU₂</td>
</tr>
<tr>
<td>ALU₂</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Reg[rd] ← func(A, B)</td>
<td>fetch₀</td>
</tr>
</tbody>
</table>

### Microprogram in the ROM Cont.

<table>
<thead>
<tr>
<th>State</th>
<th>Op</th>
<th>zero?</th>
<th>busy</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU₀</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← Reg[rs1]</td>
<td>ALU₁</td>
</tr>
<tr>
<td>ALU₁</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>B ← Imm</td>
<td>ALU₂</td>
</tr>
<tr>
<td>ALU₂</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Reg[rd] ← Op(A, B)</td>
<td>fetch₀</td>
</tr>
<tr>
<td>... J₀</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← A - 4</td>
<td>J₁</td>
</tr>
<tr>
<td>J₁</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>B ← IR</td>
<td>J₂</td>
</tr>
<tr>
<td>J₂</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>PC ← JumpTarg(A, B)</td>
<td>fetch₀</td>
</tr>
<tr>
<td>... beq₀</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← Reg[rs1]</td>
<td>beq₁</td>
</tr>
<tr>
<td>beq₁</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>B ← Reg[rs2]</td>
<td>beq₂</td>
</tr>
<tr>
<td>beq₂</td>
<td>*</td>
<td>yes</td>
<td>*</td>
<td>A ← PC</td>
<td>beq₃</td>
</tr>
<tr>
<td>beq₃</td>
<td>*</td>
<td>no</td>
<td>*</td>
<td>....</td>
<td>fetch₀</td>
</tr>
<tr>
<td>beq₄</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← A - 4</td>
<td>beq₄</td>
</tr>
<tr>
<td>beq₅</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>B ← Blimm</td>
<td>beq₅</td>
</tr>
<tr>
<td>beq₆</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>PC ← A+B</td>
<td>fetch₀</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**CS152 Administrivia**

- Each lab worth 7% of grade
  - 2% for directed portion
  - 5% for open-ended portion
- Everyone gets two automatic late lab extensions, up to one class after original deadline
  - Zero grade after that unless serious documented circumstances
- First discussion section tomorrow 10:30AM, 9 Evans
- Sign up for Piazza! This will be main forum for class announcements and discussion

---

**Size of Control Store**

\[ \text{size} = 2^{(w+s)} \times (c + s) \]

**RISC-V:**
- \( w = 5+2 \)
- \( c = 17 \)
- \( s = ? \)

- No. of steps per opcode = 4 to 6 + fetch-sequence
- No. of states = \((4 \text{ steps per op-group } \times \text{ op-groups}) + \text{ common sequences} \)
- = 4 \times 8 + 10 states = 42 states \( \Rightarrow \) \( s = 6 \)

Control ROM = \( 2^{(s+6)} \times 23 \text{ bits} = 24 \text{ Kbytes} \)
Reducing Control Store Size

Control store has to be fast ⇒ 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)*

RISC-V Controller V2

Opcode \(\rightarrow\) *ext*  \(\rightarrow\) absolute  \(\rightarrow\) *op-group*  \(\rightarrow\) \(\mu\)PC  \(\rightarrow\) \(\mu\)PC+1

\(\mu\)JumpType = *next* | *spin* | *fetch* | *dispatch* | *true* | *false*

Control ROM

Control Signals (17)  \(\rightarrow\) jump logic

\(next-state encoding reduces ROM width\)  \(\rightarrow\) \(\mu\)PC (state)  \(\rightarrow\) \(\mu\)PCSrc

\(input encoding reduces ROM height\)  \(\rightarrow\) \(\mu\)PCSrc

\(\mu\)PCSrc \(\rightarrow\) jump logic

\(\mu\)PCSrc \(\rightarrow\) zero  \(\rightarrow\) busy

1/24/2013  
CS152, Spring 2013
Jump Logic

\[ \muPCs = \text{Case} \quad \muJumpTypes \]

- next \[\Rightarrow \muPC + 1\]
- spin \[\Rightarrow \text{if (busy) then } \muPC \text{ else } \muPC + 1\]
- fetch \[\Rightarrow \text{absolute}\]
- dispatch \[\Rightarrow \text{op-group}\]
- ftrue \[\Rightarrow \text{if (zero) then } \text{absolute else } \muPC + 1\]
- ffalse \[\Rightarrow \text{if (zero) then } \muPC + 1 \text{ else } \text{absolute}\]

Instruction Fetch & ALU: \textit{RISC-V-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, A [\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>PC [\leftarrow] A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU_0</td>
<td>A [\leftarrow] Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>ALU_1</td>
<td>B [\leftarrow] Reg[rs2]</td>
<td>next</td>
</tr>
<tr>
<td>ALU_2</td>
<td>Reg[rd] [\leftarrow] func(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>ALU_i_0</td>
<td>A [\leftarrow] Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>ALU_i_1</td>
<td>B [\leftarrow] 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: RISC-V-Controller-2

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW&lt;sub&gt;0&lt;/sub&gt;</td>
<td>A ← Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>LW&lt;sub&gt;1&lt;/sub&gt;</td>
<td>B ← Imm</td>
<td>next</td>
</tr>
<tr>
<td>LW&lt;sub&gt;2&lt;/sub&gt;</td>
<td>MA ← A+B</td>
<td>next</td>
</tr>
<tr>
<td>LW&lt;sub&gt;3&lt;/sub&gt;</td>
<td>Reg[rd] ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>LW&lt;sub&gt;4&lt;/sub&gt;</td>
<td></td>
<td>fetch</td>
</tr>
<tr>
<td>SW&lt;sub&gt;0&lt;/sub&gt;</td>
<td>A ← Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>SW&lt;sub&gt;1&lt;/sub&gt;</td>
<td>B ← BImm</td>
<td>next</td>
</tr>
<tr>
<td>SW&lt;sub&gt;2&lt;/sub&gt;</td>
<td>MA ← A+B</td>
<td>next</td>
</tr>
<tr>
<td>SW&lt;sub&gt;3&lt;/sub&gt;</td>
<td>Memory ← Reg[rs2]</td>
<td>spin</td>
</tr>
<tr>
<td>SW&lt;sub&gt;4&lt;/sub&gt;</td>
<td></td>
<td>fetch</td>
</tr>
</tbody>
</table>

## Branches: RISC-V-Controller-2

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq&lt;sub&gt;0&lt;/sub&gt;</td>
<td>A ← Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>beq&lt;sub&gt;1&lt;/sub&gt;</td>
<td>B ← Reg[rs2]</td>
<td>next</td>
</tr>
<tr>
<td>beq&lt;sub&gt;2&lt;/sub&gt;</td>
<td>A ← PC</td>
<td>ffalse</td>
</tr>
<tr>
<td>beq&lt;sub&gt;3&lt;/sub&gt;</td>
<td>A ← A- 4</td>
<td>next</td>
</tr>
<tr>
<td>beq&lt;sub&gt;3&lt;/sub&gt;</td>
<td>B ← BImm&lt;&lt;1</td>
<td>next</td>
</tr>
<tr>
<td>beq&lt;sub&gt;4&lt;/sub&gt;</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
</tbody>
</table>
Jumps: RISC-V-Controller-2

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>J₀</td>
<td>A ← A-4</td>
<td>next</td>
</tr>
<tr>
<td>J₁</td>
<td>B ← IR</td>
<td>next</td>
</tr>
<tr>
<td>J₂</td>
<td>PC ← JumpTarg(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>JR₀</td>
<td>A ← Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>JR₁</td>
<td>PC ← A</td>
<td>fetch</td>
</tr>
<tr>
<td>JAL₀</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>JAL₁</td>
<td>Reg[1] ← A</td>
<td>next</td>
</tr>
<tr>
<td>JAL₂</td>
<td>A ← A-4</td>
<td>next</td>
</tr>
<tr>
<td>JAL₃</td>
<td>B ← IR</td>
<td>next</td>
</tr>
<tr>
<td>JAL₄</td>
<td>PC ← JumpTarg(A,B)</td>
<td>fetch</td>
</tr>
</tbody>
</table>

VAX 11-780 Microcode

1/24/2013 CS152, Spring 2013
Implementing Complex Instructions

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

Mem-Mem ALU Instructions:

Mem-Mem ALU op  M([rd]) ← M([rs1]) op M([rs2])

ALUMM_0  MA ← Reg[rs1]  next
ALUMM_1  A ← Memory  spin
ALUMM_2  MA ← Reg[rs2]  next
ALUMM_3  B ← Memory  spin
ALUMM_4  MA ← Reg[rd]  next
ALUMM_5  Memory ← func(A,B)  spin
ALUMM_6  fetch
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*

Horizontal vs Vertical μCode

- Horizontal μcode has wider μinstructions
  - Multiple parallel operations per μinstruction
  - Fewer microcode steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical μcode has narrower μinstructions
  - Typically a single datapath operation per μinstruction
  - Separate μinstruction for branches
  - More microcode steps per macroinstruction
  - More compact ⇒ less bits
- Nanocoding
  - Tries to combine best of horizontal and vertical μcode
Nanocoding

Exploits recurring control signal patterns in \( \mu \text{code} \), e.g.,

\[
\begin{align*}
\text{ALU}_0 & \quad A \leftarrow \text{Reg[rs1]} \\
\cdots & \quad \quad \cdots \\
\text{ALU}_i & \quad A \leftarrow \text{Reg[rs1]}
\end{align*}
\]

- MC68000 had 17-bit \( \mu \text{code} \) containing either 10-bit \( \mu \text{jump} \) 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>( \mu \text{inst} ) width (bits)</td>
<td>50</td>
<td>52</td>
<td>85</td>
<td>87</td>
</tr>
<tr>
<td>( \mu \text{code} ) size (K ( \mu \text{insts} ))</td>
<td>4</td>
<td>4</td>
<td>2.75</td>
<td>2.75</td>
</tr>
<tr>
<td>( \mu \text{store} ) technology</td>
<td>CCROS</td>
<td>TCROS</td>
<td>BCROS</td>
<td>BCROS</td>
</tr>
<tr>
<td>( \mu \text{store} ) 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
IBM Card Capacitor Read-Only Storage

Punched Card with metal film

Fixed sensing plates

[ IBM Journal, January 1961]

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 DRAMs were available
- For complex instruction sets, 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 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 is far from extinct

- Played a crucial role in micros of the Eighties
  - DEC uVAX, Motorola 68K series, Intel 286/386
- Plays an assisting role in most modern micros
  - e.g., AMD Bulldozer, Intel Ivy Bridge, Intel Atom, IBM PowerPC, ...
  - Most instructions executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke microcode
- Patchable microcode common for post-fabrication bug fixes, e.g. Intel processors load μcode patches at bootup

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