ECE 4750 Computer Architecture
Topic 1: Microcoding

Christopher Batten
School of Electrical and Computer Engineering
Cornell University

http://www.csl.cornell.edu/courses/ece4750
slide revision: 2013-09-01-10-42
Agenda

Instruction Set Architecture
IBM 360 Instruction Set
MIPS Instruction Set
ISA to Microarchitecture Mapping

Microcoded MIPS Processor
Microcoded MIPS Microarchitecture #1
Microcoded MIPS Microarchitecture #2

Microcoding Discussion and Trends
Instruction Set Architecture

- Contract between software & hardware
- Typically specified as all of the programmer-visible state (registers & memory) plus the semantics of instructions that operate on this state
- IBM 360 was first line of machines to separate ISA from microarchitecture and implementation

... the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

— Amdahl, Blaauw, Brooks, 1964
Compatibility Problem at IBM

- By early 1960’s, IBM had several incompatible lines of computers!
  - Defense : 701
  - Scientific : 704, 709, 7090, 7094
  - Business : 702, 705, 7080
  - Mid-Sized Business : 1400
  - Decimal Architectures : 7070, 7072, 7074

- Each system had its own:
  - Instruction set
  - I/O system and secondary storage (tapes, drums, disks)
  - Assemblers, compilers, libraries, etc
  - Market niche
IBM 360: A General-Purpose Register Machine

- **Processor State**
  - 16 general-purpose 32-bit registers
  - 4 floating point 64-bit registers
  - Program counter (PC), condition codes, control flags

- **Data Format**
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words
  - IBM 360 is why bytes are 8-bits long today!

- **Instruction Format**
  - Variable length from 2–6 bytes long

- **Memory Addressing**
  - 32-bit machine with only 24-bit addressing
IBM 360: Initial Implementations

<table>
<thead>
<tr>
<th></th>
<th>Model 30</th>
<th>Model 70</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>8–64 KB</td>
<td>256–512 KB</td>
</tr>
<tr>
<td>Datapath</td>
<td>8-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>Circuit Delay</td>
<td>30 ns/level</td>
<td>5 ns/level</td>
</tr>
<tr>
<td>Local Store</td>
<td>Main store</td>
<td>Transistor registers</td>
</tr>
<tr>
<td>Control Store</td>
<td>Read only 1μs</td>
<td>Conventional circuits</td>
</tr>
</tbody>
</table>

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

Significant Milestone: The first true ISA designed as a portable hardware-software interface.
IBM 360: 45 years later ...

The zSeries Z196 Microprocessor

- 5.2 GHz in IBM 45 nm SOI
- 1.4B transistors in 512 mm²
- Four cores per chip
- Aggressive out-of-order execution
- Four-level cache hierarchy with embedded DRAM L3/L4
- Available in September 2010
- Can still run IBM 360 code!

— Hot Chips 2010
MIPS32 Instruction Set

- **Processor State**
  - 32 general-purpose 32-bit registers
  - 32 SP (16 DP) floating-point regs
  - Program counter, FP status reg

- **Data Format**
  - 8-bit bytes, 16-bit half-words, 32-bit words
  - 32-bit single-precision, 64-bit double-precision floating-point

- **Instruction Format**
  - Fixed-length 32-bit instructions

- **Memory Addressing**
  - Data addressing with immediate and indexed
  - Branch addressing with PC relative and register indirect
  - Byte addressable memory as big or little endian
MIPS Data Formats & Memory Addressing

Data formats:

- Bytes, Half words, words and double words

Some issues

- **Byte addressing**
  - Big Endian vs. Little Endian

- **Word alignment**
  - Suppose the memory is organized in 32-bit words.
  - Can a word address begin only at 0, 4, 8, ...?
## MIPS Instruction Formats

### ALU

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>rs</td>
</tr>
<tr>
<td>26</td>
<td>rt</td>
</tr>
<tr>
<td>21</td>
<td>rd</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>func</td>
</tr>
</tbody>
</table>

R[rd] ← R[rs] func R[rt]

### ALUI

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>opcode</td>
</tr>
<tr>
<td>26</td>
<td>rs</td>
</tr>
<tr>
<td>21</td>
<td>rt</td>
</tr>
<tr>
<td>16</td>
<td>immediate</td>
</tr>
</tbody>
</table>

R[rt] ← R[rs] op immediate

### LD/ST

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>opcode</td>
</tr>
<tr>
<td>26</td>
<td>rs</td>
</tr>
<tr>
<td>21</td>
<td>rt</td>
</tr>
<tr>
<td>16</td>
<td>offset</td>
</tr>
</tbody>
</table>

ST: M[ R[rs] + sext(offset) ] ← R[rt]
LD: R[rt] ← M[ R[rs] + sext(offset) ]

### BEQZ

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>opcode</td>
</tr>
<tr>
<td>26</td>
<td>rs</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>offset</td>
</tr>
</tbody>
</table>

if ( R[rs] == 0 )
PC ← PC+4 + offset*4

### JR/JALR

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>opcode</td>
</tr>
<tr>
<td>26</td>
<td>rs</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>

PC ← R[rs]
JALR also does R[31] ← PC+8

### J/JAL

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>opcode</td>
</tr>
<tr>
<td>26</td>
<td>target</td>
</tr>
</tbody>
</table>

PC ← j targ( PC, target )
JAL also does R[31] ← PC+8
More Information on MIPS32

- Harris & Harris, “Digital Design & Computer Architecture,” 2nd ed, Ch. 6, App. B
- Sweetman, “See MIPS Run Linux,” 2nd ed
- The MIPS32/64 Instruction Set manual (http://www.mips.com)
ISA to Microarchitecture Mapping

- **Microarchitectural Structure:** How components are statically connected together
- **Microarchitectural Function:** How data is dynamically processed or stored in components

**Diagram:**
- **Control Unit**
  - Status Signals
  - Control Signals
  - Datapath
- **Logic**
- **State**
Many Microarchitectures are Possible

- ISA map to many similar yet distinct microarchitectures & impls:
  - 360 → IBM 360 Model 30, IBM Z196
  - x86 → Intel 80486, AMD Bobcat, VIA Nano
  - ARM → Cellphone implementations from TI, Qualcomm, Samsung
  - PARC → Labs: pipelined with stalling, pipelined with bypassing

- Architectural design patterns capture common classes of instruction sets and a suitable microarchitectural template:
  - CISC → microcoded
  - RISC → hardwired, pipelined
  - VLIW → fixed-latency in-order parallel pipelines
  - JVM → software interpretation

- ISAs can always be implemented with alternative microarchitectures:
  - Intel Nehalem: Hardwired pipelined CISC proc, w/ some μcode
  - ARM Jazelle: A hardwired JVM processor
  - This lecture: A microcoded RISC (MIPS) processor
Agenda

Instruction Set Architecture
- IBM 360 Instruction Set
- MIPS Instruction Set
- ISA to Microarchitecture Mapping

Microcoded MIPS Processor
- Microcoded MIPS Microarchitecture #1
- Microcoded MIPS Microarchitecture #2

Microcoding Discussion and Trends
Why Look Microprogramming at Now?

- To show how to build very small processors with complex ISAs
- Because it is still used in most common machines
- As a gentle introduction to machine structures
- To help you understand where CISC machines came from
- To help understand how technology drove the move to RISC
Microcoding: Embed ctrl logic state in memory array

— Maurice Wilkes, 1954
Bus-Based Datapath for MIPS

Microinstruction: register to register transfer (17 control signals)

MA ← PC  means  RegSel = PC; enReg=yes;  ldMA = yes
B ← Reg[rt] means  RegSel = rt; enReg=yes;  ldB  = yes
Assumption: Memory operates independently and is slow compared to register-to-register transfers (multiple CPU clock cycles per access)
 Instruction Execution Steps

1. Instruction fetch
2. Decode and register fetch
3. ALU operation
4. Memory operation if required
5. Register write-back if required

— Computation of the next instruction to fetch
Microprogram Fragments

- **Instruction Fetch**
  - MA ← PC
  - IR ← Memory
  - A ← PC
  - PC ← A + 4
  - dispatch on opcode

- **ALU**
  - A ← R[rs]
  - B ← R[rt]
  - R[rd] ← func(A,B)
  - do instruction fetch

- **ALUI**
  - A ← R[rs]
  - B ← Imm (sign extend?)
  - R[rt] ← opcode(A,B)
  - do instruction fetch

- **LW**
  - A ← R[rs]
  - B ← Imm (sign extend?)
  - MA ← A + B
  - R[rt] ← Memory
  - do instruction fetch
Microprogram Fragments

- **J**
  - A ← PC
  - B ← IR
  - PC ← { A[31:28], B[25:0], 00 }
  - do instruction fetch

- **BEQZ**
  - A ← R[rs]
  - if zero?(A) then goto br-taken
  - do instruction fetch

- **br-taken**
  - A ← PC
  - B ← Imm << 2 (sign extend?)
  - PC ← A + B
  - do instruction fetch
MIPS Controller: Microarchitecture #1

- Opcode
- Busy (memory)
- µPC (state)
- µProgram ROM
- Control Signals (17)

- ROM size? = $2^{(\text{opcode}+\text{status}+s)}$ words
- Word size? = control+s bits

- How big is “s”?
## MIPS Microcode: Microarchitecture #1

<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 ← 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</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← PC</td>
<td>fetch₃</td>
</tr>
<tr>
<td>fetch₃</td>
<td>ALU</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>ALU₀</td>
</tr>
<tr>
<td>ALU₀</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← Reg[rs]</td>
<td>ALU₁</td>
</tr>
<tr>
<td>ALU₁</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>B ← Reg[rt]</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>
### MIPS Microcode: Microarchitecture #1

<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 ← 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</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← PC</td>
<td>fetch₃</td>
</tr>
<tr>
<td>fetch₃</td>
<td>ALU</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>ALU₀</td>
</tr>
<tr>
<td>fetch₃</td>
<td>ALUi</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>ALUi₀</td>
</tr>
<tr>
<td>fetch₃</td>
<td>LW</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>LW₀</td>
</tr>
<tr>
<td>fetch₃</td>
<td>SW</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>SW₀</td>
</tr>
<tr>
<td>fetch₃</td>
<td>J</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>J₀</td>
</tr>
<tr>
<td>fetch₃</td>
<td>JAL</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>JAL₀</td>
</tr>
<tr>
<td>fetch₃</td>
<td>JR</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>JR₀</td>
</tr>
<tr>
<td>fetch₃</td>
<td>JALR</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>JALR₀</td>
</tr>
<tr>
<td>fetch₃</td>
<td>beqz</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>beqz₀</td>
</tr>
</tbody>
</table>

...  

ALU₀ * * * A ← Reg[rs] ALU₁  
ALU₁ * * * B ← Reg[rt] ALU₂  
ALU₂ * * * Reg[rd] ← func(A,B) fetch₀
## MIPS Microcode: Microarchitecture #1

<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>ALUi₀</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← Reg[rs]</td>
<td>ALUi₁</td>
</tr>
<tr>
<td>ALUi₁</td>
<td>sExt</td>
<td>*</td>
<td>*</td>
<td>B ← sExt₁₆(Imm)</td>
<td>ALUi₂</td>
</tr>
<tr>
<td>ALUi₁</td>
<td>uExt</td>
<td>*</td>
<td>*</td>
<td>B ← uExt₁₆(Imm)</td>
<td>ALUi₂</td>
</tr>
<tr>
<td>ALUi₂</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 ← PC</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>beqz₀</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beqz₁</td>
<td>*</td>
<td>yes</td>
<td>*</td>
<td>A ← Reg[rs]</td>
<td>beqz₁</td>
</tr>
<tr>
<td>beqz₁</td>
<td>*</td>
<td>no</td>
<td>*</td>
<td>A ← PC</td>
<td>beqz₂</td>
</tr>
<tr>
<td>beqz₂</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beqz₃</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Size of Control Store

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

\textit{MIPS:} \quad w = 6+2 \quad c = 17 \quad s = ?

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

Control ROM = \(2^{(8+6)} \times 23 \text{ bits} \approx 48 \text{ Kbytes}\)
Reducing Control Store Size

Control store has to be fast which makes it expensive

- Reduce the ROM height (i.e., number of address bits)
  - Reduce status inputs with extra external logic
  - Reduce states by grouping opcodes

- Reduce the ROM width (i.e., number of bits per μinstruction)
  - Restrict the next state encoding
  - Encode control signals (called vertical microcoding)
MIPS Controller: Microarchitecture #2

µJumpType = next | spin | fetch | dispatch | feqz | fnez

Control Signals (17)

input encoding reduces ROM height

next-state encoding reduces ROM width

µPC Src input encoding reduces ROM height

µPC +1

µPC (state)

address

data

jump logic

zero

busy

µJumpType = next | spin
| fetch | dispatch
| feqz | fnez

 opcode

 encode

µPC

µPC+1

Control ROM

next-state encoding reduces ROM width

ECE 4750 T01: Microcoding 28 / 45
Jump Logic

\( \mu\text{PCSrc} = \text{Case } \mu\text{JumpTypes} \)

- next \( \rightarrow \mu\text{PC} + 1 \)
- spin \( \rightarrow \) if (busy) then \( \mu\text{PC} \) else \( \mu\text{PC} + 1 \)
- fetch \( \rightarrow \) absolute
- dispatch \( \rightarrow \) op-group
- feqz \( \rightarrow \) if (zero) then absolute else \( \mu\text{PC} + 1 \)
- fnez \( \rightarrow \) if (zero) then \( \mu\text{PC} + 1 \) else absolute
### MIPS Microcode: Microarchitecture #2

<table>
<thead>
<tr>
<th>State</th>
<th>Control point</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fetch_0</code></td>
<td>MA ← PC</td>
<td>next</td>
</tr>
<tr>
<td><code>fetch_1</code></td>
<td>IR ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td><code>fetch_2</code></td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td><code>fetch_3</code></td>
<td>PC ← A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>ALU_0</code></td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td><code>ALU_1</code></td>
<td>B ← Reg[rt]</td>
<td>next</td>
</tr>
<tr>
<td><code>ALU_2</code></td>
<td>Reg[rd] ← func(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td><code>ALUi_0</code></td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td><code>ALUi_1</code></td>
<td>B ← sExt&lt;sub&gt;16&lt;/sub&gt;(Imm)</td>
<td>next</td>
</tr>
<tr>
<td><code>ALUi_2</code></td>
<td>Reg[rd] ← Op(A,B)</td>
<td>fetch</td>
</tr>
</tbody>
</table>
### MIPS Microcode: Microarchitecture #2

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>LW₁</td>
<td>B ← sExt₁₆(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>LW₂</td>
<td>MA ← A+B</td>
<td>next</td>
</tr>
<tr>
<td>LW₃</td>
<td>Reg[rt] ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>LW₄</td>
<td></td>
<td>fetch</td>
</tr>
<tr>
<td>SW₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>SW₁</td>
<td>B ← sExt₁₆(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>SW₂</td>
<td>MA ← A+B</td>
<td>next</td>
</tr>
<tr>
<td>SW₃</td>
<td>Memory ← Reg[rt]</td>
<td>spin</td>
</tr>
<tr>
<td>SW₄</td>
<td></td>
<td>fetch</td>
</tr>
</tbody>
</table>
### MIPS Microcode: Microarchitecture #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>
## MIPS Microcode: Microarchitecture #2

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>J₀</td>
<td>A ← PC</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) fetch</td>
<td></td>
</tr>
<tr>
<td>JR₀</td>
<td>A ← Reg[rs]</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[31] ← A</td>
<td>next</td>
</tr>
<tr>
<td>JAL₂</td>
<td>B ← IR</td>
<td>next</td>
</tr>
<tr>
<td>JAL₃</td>
<td>PC ← JumpTarg(A,B) fetch</td>
<td></td>
</tr>
<tr>
<td>JALR₀</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>JALR₁</td>
<td>B ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>JALR₂</td>
<td>Reg[31] ← A</td>
<td>next</td>
</tr>
<tr>
<td>JALR₃</td>
<td>PC ← B</td>
<td>fetch</td>
</tr>
</tbody>
</table>
Agenda

Instruction Set Architecture
IBM 360 Instruction Set
MIPS Instruction Set
ISA to Microarchitecture Mapping

Microcoded MIPS Processor
Microcoded MIPS Microarchitecture #1
Microcoded MIPS Microarchitecture #2

Microcoding Discussion and Trends
Implementing Complex Instructions

- **Opcode**
  - ldA
  - ldB
  - ldMA
- **ExtSel**
  - A
  - B
- **RegWrt**
  - enReg
  - enMem
- **RegSel**
  - MA
  - addr
- **ALU**
  - data
- **Bus**
  - 32
- **Memory**
  - data
  - addr
- **MemWrt**
  - enMem
- **OpSel**
  - 2
- **ldIR**
- **ldMA**
- **32-bit Reg**
- **ALU control**
- **enImm**
  - 2
- **Ir**
  - Ext
- **Imm Ext**
- **zero?**
  - 3
- **rd**
  - M[(rs)] op (rt)
- **rt**
  - (rs) op (rt)
- **rs**
  - M[(rd)] ← M[(rs)] op M[(rt)]
- **Reg-Memory-src ALU op**
- **Reg-Memory-dst ALU op**
- **Reg-Mem-Mem ALU op**

**Equations:**

- rd ← M[(rs)] op (rt)
- M[(rd)] ← (rs) op (rt)
- M[(rd)] ← M[(rs)] op M[(rt)]
Mem-Mem ALU Instructions with μArch #2

\[ M[R[rd]] \leftarrow M[R[rs]] \text{ op } M[R[rt]] \]

- \text{ALUMM}_0 \quad MA \leftarrow R[rs] \quad \text{next}
- \text{ALUMM}_1 \quad A \leftarrow \text{Memory} \quad \text{spin}
- \text{ALUMM}_2 \quad MA \leftarrow R[rt] \quad \text{next}
- \text{ALUMM}_3 \quad B \leftarrow \text{Memory} \quad \text{spin}
- \text{ALUMM}_4 \quad MA \leftarrow R[rd] \quad \text{next}
- \text{ALUMM}_5 \quad \text{Memory} \leftarrow \text{func}(A,B) \quad \text{spin}
- \text{ALUMM}_6

- Complex instructions usually do not require datapath modifications in a microprogrammed μarch, only extra space for the control program.
- Implementing these instructions using a hardwired controller is difficult without datapath modifications.
“Iron Law” of Processor Performance

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

- Instructions / program depends on source code, compiler, ISA
- Cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

- Microprogrammed Control
  - Multiple cycles per instruction
  - Cycle time = \( \max(t_{\text{regrd}}, t_{\text{regwr}}, t_{\text{ALU}}, t_{\text{urom}}) \)
  - Good performance relative to hardwired implementation is possible if \( t_{\text{urom}} \) is small and \( t_{\text{urom}} \ll t_{\text{ram}} \)
Horizontal vs Vertical μCode

- Horizontal μcode has wider μinstructions
  - Multiple parallel operations per μinstruction
  - Fewer microcode steps per macroinstruction
  - Sparser encoding = more bits, but less decoding logic

- Vertical μcode has narrower μinstructions
  - Typically a single datapath operation per μinstruction
  - More microcode steps per macroinstruction
  - More compact = less bits, but more decoding logic
IBM 360 Microprogramming

<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 (1K 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</td>
<td>BCROS</td>
<td>BCROS</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 (75,95) were hardwired
IBM 360/M30 Card Capacitor Read-Only Storage

— IBM Technology Journal, Jan 1961
IBM 360/M30 Microprogram: Reg-Reg Logic Or

**Instruction Fetch**

- Fetch first byte of operands

**OR**

- Writeback Result

---

ECE 4750 T01: Microcoding
IBM 360/M30 Microprogram: Reg-Reg Binary Add
Microcoding Historical Trends

Microcoding thrived in the 1970’s
- ROMs significantly faster than DRAMs
- For complex instruction sets, microcode was cheaper and simpler
- New instructions supported without modifying datapath
- Fixing bugs in controller is easier
- ISA compatibility across models relatively straight-forward

Microcoding started to fall out of favor in the 1980’s
- VLSI changed technology assumptions for ROM vs DRAM
- Better compilers made complex instructions less important
- Numerous microarchitectural innovations (pipelining, caches) made multi-cycle reg-reg instructions less attractive

Looking ahead to RISC over next few lectures
- Build fast instruction caches (i.e., user-visible vertical microinstructions)
- Use simple ISA to enable fast hardwired pipelined implementation
Microprogramming Today

- Microprogramming is far from extinct

- Played a crucial role in microprocessors of the 1980s (DEC VAX, Motorola 68K series, Intel 386/486)

- Microprogramming plays assisting role in many modern processors (AMD Phenom, Intel Nehalem, Intel Atom, IBM Z196)
  - 761 Z196 instructions executed with hardwired control
  - 219 Z196 “complex” instructions always executed with microcode
  - 24 Z196 instructions conditionally executed with microcode

- Patchable microcode common for post-fabrication bug fixes (Intel processors load µcode patches at bootup)
Acknowledgements

Some of these slides contain material developed and copyrighted by:

Arvind (MIT), Krste Asanović (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 courses CS152 and CS252