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
Last Time in Lecture 1

• Computer Science at crossroads from sequential to parallel computing

• Computer Architecture >> ISAs and RTL
  – CS152 is about interaction of hardware and software, and design of appropriate abstraction layers

• Comp. Arch. shaped by technology and applications
  – History provides lessons for the future

• Cost of software development a large constraint on architecture
  – Compatibility a key solution to software cost

• 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
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., today you can buy AMD or Intel processors that run the x86-64 ISA.
  – E.g.2: many cellphones use the ARM ISA with implementations from many different companies including TI, Qualcomm, Samsung, Marvell, etc.
  – E.g.3., the Soviets build code-compatible clones of the IBM360, as did Amdhal after he left IBM.
Microprogramming

• Today, a brief look at microprogrammed machines
  – To show how to build very small processors with complex ISAs
  – To help you understand where CISC* machines came from
  – Because it is still used in the most common machines (x86, PowerPC, IBM360)
  – As a gentle introduction into machine structures
  – To help understand how technology drove the move to RISC*

* CISC/RISC names came much later than the style of machines they refer to.
ISA to Microarchitecture Mapping

• ISA often designed with particular microarchitectural style in mind, e.g.,
  – CISC  ⇒  microcoded
  – RISC  ⇒  hardwired, pipelined
  – VLIW  ⇒  fixed-latency in-order parallel pipelines
  – JVM   ⇒  software interpretation
• But can be implemented with any microarchitectural style
  – Intel Nehalem: hardwired pipelined CISC (x86) machine (with some microcode support)
  – Simics: Software-interpreted SPARC RISC machine
  – Intel could implement a dynamically scheduled out-of-order VLIW Itanium (IA-64) processor
  – ARM Jazelle: A hardware JVM processor
  – This lecture: a microcoded RISC (MIPS) machine
Microarchitecture: *Implementation of an ISA*

**Structure:** How components are connected.

*Static*

**Behavior:** How data moves between components

*Dynamic*
Microcontrol Unit  *Maurice Wilkes, 1954*

First used in EDSAC-2, completed 1958

Embed the control logic state table in a memory array

- \( \mu \) address
- Matrix A
- Matrix B
- Decoder
- Memory
- Control lines to ALU, MUXs, Registers

<table>
<thead>
<tr>
<th>op</th>
<th>conditional code</th>
<th>flip-flop</th>
</tr>
</thead>
</table>

January 21, 2010

CS152 Spring 2010
Microcoded Microarchitecture

- **μcontroller (ROM)**
  - busy?
  - zero?
  - opcode
  - holds fixed microcode instructions

- **Datapath**
  - Data
  - Addr

- **Memory (RAM)**
  - holds user program written in macrocode instructions (e.g., MIPS, x86, etc.)
  - enMem
  - MemWrt
The MIPS32 ISA

• Processor State
  32 32-bit GPRs, R0 always contains a 0
  16 double-precision/32 single-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
MIPS Instruction Formats

**ALU**

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>func</td>
</tr>
</tbody>
</table>

`rd ← (rs) func (rt)`

**ALUi**

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
</tr>
</tbody>
</table>

`rt ← (rs) op immediate`

**Mem**

<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>rt</td>
<td>displacement</td>
</tr>
</tbody>
</table>

`M[(rs) + displacement]`

**Mem (BEQZ, BNEZ)**

<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>rt</td>
<td>offset</td>
</tr>
</tbody>
</table>

**Mem (JR, JALR)**

<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>rt</td>
<td>offset</td>
</tr>
</tbody>
</table>

**Mem (J, JAL)**

<table>
<thead>
<tr>
<th>6</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>offset</td>
</tr>
</tbody>
</table>
Data Formats and Memory Addresses

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, .... ?

![Byte Addresses Diagram](image)
A Bus-based Datapath for MIPS

Microinstruction: register to register transfer (17 control signals)

MA $\leftarrow$ PC means RegSel = PC; enReg=yes; ldMA = yes
B $\leftarrow$ Reg[rt] means RegSel = rt; enReg=yes; ldB = yes

January 21, 2010 CS152 Spring 2010
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 MIPS 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

instr fetch:
MA ← PC
A ← PC
IR ← Memory
PC ← A + 4
dispatch on OPcode

can be treated as a macro

ALU:
A ← Reg[rs]
B ← Reg[rt]
Reg[rd] ← func(A,B)
do instruction fetch

ALUi:
A ← Reg[rs]
B ← Imm
Reg[rt] ← Opcode(A,B)
sign extension ...
do instruction fetch
Microprogram Fragments (cont.)

LW:
A ← Reg[rs]
B ← Imm
MA ← A + B
Reg[rt] ← Memory

do instruction fetch

J:
A ← PC
B ← IR
PC ← JumpTarg(A,B)

do instruction fetch

beqz:
A ← Reg[rs]
If zero?(A) then go to bz-taken

do instruction fetch

bz-taken:
A ← PC
B ← Imm << 2
PC ← A + B

do instruction fetch

JumpTarg(A,B) = {A[31:28],B[25:0],00}
MIPS Microcontroller: *first attempt*

**Operations:**
- Opcode
- Busy (memory)

**Control Signals (17):**
- How big is “s”?
- Next state

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

**µPC (state):**
- addr
- data
## 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>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>*</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>?</td>
</tr>
<tr>
<td>fetch₃</td>
<td>ALU</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>ALU₀</td>
</tr>
</tbody>
</table>

| ALU₀ | * | * | * | A ← Reg[rs] | ALU₁ |
| ALU₁ | * | * | * | B ← Reg[rt] | ALU₂ |
| ALU₂ | * | * | * | Reg[rd] ← func(A,B) | fetch₀ |
### 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 ← 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>
<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>
### 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>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>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beqz₀</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← Reg[rs]</td>
<td>beqz₁</td>
</tr>
<tr>
<td>beqz₁</td>
<td>*</td>
<td>yes</td>
<td>*</td>
<td>A ← PC</td>
<td>beqz₂</td>
</tr>
<tr>
<td>beqz₁</td>
<td>*</td>
<td>no</td>
<td>*</td>
<td>…</td>
<td>fetch₀</td>
</tr>
<tr>
<td>beqz₂</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>B ← sExt₁₆(Imm)</td>
<td>beqz₃</td>
</tr>
<tr>
<td>beqz₃</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>PC ← A+B</td>
<td>fetch₀</td>
</tr>
</tbody>
</table>

*JumpTarg(A,B) = \{A[31:28],B[25:0],00\}*
Size of Control Store

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

MIPS:
- \( w = 6 + 2 \)
- \( c = 17 \)
- \( s = ? \)

no. of steps per opcode = 4 to 6 + fetch-sequence
no. of states \( \approx (4 \text{ steps per op-group }) \times \text{op-groups} \)
+ common sequences
\( = 4 \times 8 + 10 \) states = 42 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 \textit{fast} \Rightarrow \textit{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, ...
CS152 Administrivia

- Lab 1 coming out on Tuesday, together with PS1
- Lab 1 overview in Section, next Thursday, 2pm, 320 Soda
- Lab 1 and PS 1 due **start of class** Thursday Feb. 11
  - No extensions for Problem set. Zero credit afterwards.
  - Problem sets graded on 0,1,2 scale
  - Up to two free lab extensions per student, up till next class (Tuesday). Zero credit afterwards.
- Solutions to PS 1 released at **end of same class**
- Section reviewing PS 1, same Thursday at 2pm
- First Quiz, in class, Tue Feb 16, 9:30-11AM
  - Closed book, no calculators, no computers, no cellphones
- PS 2 and Lab 2 handed out day of Quiz 1
Collaboration Policy

• Can collaborate to understand problem sets, but must turn in own solution. Some problems repeated from earlier years - do not copy solutions. (Quiz problems will not be repeated…)

• Each student must complete directed portion of the lab by themselves. OK to collaborate to understand how to run labs
  – Class news group info on web site.
  – Lab reports must be readable English summaries. Zero credit for handing in output log files from experiments.

• Can work in group of up to 3 students for open-ended portion of each lab
  – OK to be in different group for each lab -just make sure to label participants’ names clearly on each turned-in lab section
MIPS Controller V2

\[ \mu \text{JumpType} = \text{next} | \text{spin} | \text{fetch} | \text{dispatch} | \text{feqz} | \text{fnez} \]

Input encoding reduces ROM height

Next-state encoding reduces ROM width
Jump Logic

\[ \mu\text{PCSrc} = \text{Case} \]
\[ \mu\text{JumpTypes} \]

- next \[ \Rightarrow \] \[ \mu\text{PC}+1 \]
- spin \[ \Rightarrow \] \[ \text{if (busy) then } \mu\text{PC} \text{ else } \mu\text{PC}+1 \]
- fetch \[ \Rightarrow \] \[ \text{absolute} \]
- dispatch \[ \Rightarrow \] \[ \text{op-group} \]
- feqz \[ \Rightarrow \] \[ \text{if (zero) then absolute else } \mu\text{PC}+1 \]
- fnez \[ \Rightarrow \] \[ \text{if (zero) then } \mu\text{PC}+1 \text{ else absolute} \]
### 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>...</td>
<td></td>
</tr>
<tr>
<td>ALU_0</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU_1</td>
<td>B $\leftarrow$ Reg[rt]</td>
<td>next</td>
</tr>
<tr>
<td>ALU_2</td>
<td>Reg[rd] $\leftarrow$ func(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>ALUi_0</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALUi_1</td>
<td>B $\leftarrow$ sExt(_{16})(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>ALUi_2</td>
<td>Reg[rd] $\leftarrow$ Op(A,B)</td>
<td>fetch</td>
</tr>
</tbody>
</table>
# Load & Store: *MIPS-Controller-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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<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>
## Branches: MIPS-Controller-2

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ₁</td>
<td></td>
<td>fnez</td>
</tr>
<tr>
<td>BEQZ₂</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ₃</td>
<td>B ← sExt₁₆(Imm&lt;&lt;2)</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ₄</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
<tr>
<td>BNEZ₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ₁</td>
<td></td>
<td>feqz</td>
</tr>
<tr>
<td>BNEZ₂</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ₃</td>
<td>B ← sExt₁₆(Imm&lt;&lt;2)</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ₄</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 \leftarrow PC )</td>
<td>next</td>
</tr>
<tr>
<td>( J_1 )</td>
<td>( B \leftarrow IR )</td>
<td>next</td>
</tr>
<tr>
<td>( J_2 )</td>
<td>( PC \leftarrow \text{JumpTarg}(A, B) ) fetch</td>
<td></td>
</tr>
<tr>
<td>( JR_0 )</td>
<td>( A \leftarrow \text{Reg[rs]} )</td>
<td>next</td>
</tr>
<tr>
<td>( JR_1 )</td>
<td>( PC \leftarrow A )</td>
<td>fetch</td>
</tr>
<tr>
<td>( JAL_0 )</td>
<td>( A \leftarrow PC )</td>
<td>next</td>
</tr>
<tr>
<td>( JAL_1 )</td>
<td>( \text{Reg[31]} \leftarrow A )</td>
<td>next</td>
</tr>
<tr>
<td>( JAL_2 )</td>
<td>( B \leftarrow IR )</td>
<td>next</td>
</tr>
<tr>
<td>( JAL_3 )</td>
<td>( PC \leftarrow \text{JumpTarg}(A, B) ) fetch</td>
<td></td>
</tr>
<tr>
<td>( JALR_0 )</td>
<td>( A \leftarrow PC )</td>
<td>next</td>
</tr>
<tr>
<td>( JALR_1 )</td>
<td>( B \leftarrow \text{Reg[rs]} )</td>
<td>next</td>
</tr>
<tr>
<td>( JALR_2 )</td>
<td>( \text{Reg[31]} \leftarrow A )</td>
<td>next</td>
</tr>
<tr>
<td>( JALR_3 )</td>
<td>( PC \leftarrow B )</td>
<td>fetch</td>
</tr>
</tbody>
</table>
VAX 11-780 Microcode

January 21, 2010

CS152 Spring 2010

Page 31
Implementing Complex Instructions

rd ← M[(rs)] op (rt)
M[(rd)] ← (rs) op (rt)
M[(rd)] ← M[(rs)] op M[(rt)]

Reg-Memory-src ALU op
Reg-Memory-dst ALU op
Mem-Mem ALU op
### Mem-Mem ALU Instructions:

**MIPS-Controller-2**

<table>
<thead>
<tr>
<th>Mem-Mem ALU op</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ALUMM_0$</td>
<td>$MA \leftarrow \text{Reg}[rs]$</td>
</tr>
<tr>
<td>$ALUMM_1$</td>
<td>$A \leftarrow \text{Memory}$</td>
</tr>
<tr>
<td>$ALUMM_2$</td>
<td>$MA \leftarrow \text{Reg}[rt]$</td>
</tr>
<tr>
<td>$ALUMM_3$</td>
<td>$B \leftarrow \text{Memory}$</td>
</tr>
<tr>
<td>$ALUMM_4$</td>
<td>$MA \leftarrow \text{Reg}[rd]$</td>
</tr>
<tr>
<td>$ALUMM_5$</td>
<td>Memory $\leftarrow \text{func}(A,B)$</td>
</tr>
<tr>
<td>$ALUMM_6$</td>
<td>Memory $\leftarrow \text{func}(A,B)$</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*
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 µcode, e.g.,

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

- 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</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 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: early Eighties

• Evolution bred more complex micro-machines
  – Complex instruction sets led to need for subroutine and call stacks in µcode
  – Need for fixing bugs in control programs was in conflict with read-only nature of µROM
  – → WCS (B1700, QMachine, Intel i432, …)

• With the advent of VLSI technology assumptions about ROM & RAM speed became invalid → more complexity

• 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

• Looking ahead to RISC next time
  – Use chip area to build fast instruction cache of user-visible vertical microinstructions - use software subroutine not hardware microroutines
  – Use simple ISA to enable hardwired pipelined implementation
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 Phenom, Intel Nehalem, Intel Atom, 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 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