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
  – Within same generation of machines
  – “Future-proofing” for subsequent generations of machine
Burrough’s B5000 Stack Architecture:
An ALGOL Machine, Robert Barton, 1960

- Machine implementation can be completely hidden if the programmer is provided only a high-level language interface.

- Stack machine organization because stacks are convenient for:
  1. expression evaluation;
  2. subroutine calls, recursion, nested interrupts;
  3. accessing variables in block-structured languages.

- B6700, a later model, had many more innovative features
  - tagged data
  - virtual memory
  - multiple processors and memories

A Stack Machine

A Stack machine has a stack as a part of the processor state

typical operations: push, pop, +, *, ...

Instructions like + implicitly specify the top 2 elements of the stack as operands.
Evaluation of Expressions

\[
\frac{(a + b \times c)}{(a + d \times c - e)}
\]

Reverse Polish

\[
a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /
\]

Evaluation Stack

```
push a
push b
push c
multiply
```

Evaluation of Expressions

\[
\frac{(a + b \times c)}{(a + d \times c - e)}
\]

Reverse Polish

\[
a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /
\]

Evaluation Stack

```
add
```

```
```
Hardware organization of the stack

• Stack is part of the processor state
  ⇒ stack must be bounded and small
  ⇒ number of Registers, not the size of main memory

• Conceptually stack is unbounded
  ⇒ a part of the stack is included in the processor state; the rest is kept in the main memory

Stack Operations and Implicit Memory References

• Suppose the top 2 elements of the stack are kept in registers and the rest is kept in the memory.

  Each push operation ⇒ 1 memory reference
  pop operation ⇒ 1 memory reference

  No Good!

• Better performance can be got if the top N elements are kept in registers and memory references are made only when register stack overflows or underflows.

  Issue - when to Load/Unload registers?
Stack Size and Memory References

\[
\begin{align*}
\text{program} & \quad \text{stack (size = 2)} & \quad \text{memory refs} \\
push a & \quad R0 & \quad a \\
push b & \quad R0 R1 & \quad b \\
push c & \quad R0 R1 R2 & \quad c, \ ss(a) \\
* & \quad R0 R1 & \quad sf(a) \\
+ & \quad R0 & \quad \\
push a & \quad R0 R1 & \quad a \\
push d & \quad R0 R1 R2 & \quad d, \ ss(a+b*c) \\
push c & \quad R0 R1 R2 R3 & \quad c, \ ss(a) \\
* & \quad R0 R1 R2 & \quad sf(a) \\
+ & \quad R0 R1 & \quad sf(a+b*c) \\
push e & \quad R0 R1 R2 & \quad e, \ ss(a+b*c) \\
- & \quad R0 R1 & \quad sf(a+b*c) \\
/ & \quad R0 & \quad \\
\end{align*}
\]

4 stores, 4 fetches (implicit)

Stack Size and Expression Evaluation

\[
\begin{align*}
\text{program} & \quad \text{stack (size = 4)} \\
push a & \quad R0 \\
push b & \quad R0 R1 \\
push c & \quad R0 R1 R2 \\
* & \quad R0 R1 \\
+ & \quad R0 \\
push a & \quad R0 R1 \\
push d & \quad R0 R1 R2 \\
push c & \quad R0 R1 R2 R3 \\
* & \quad R0 R1 R2 \\
+ & \quad R0 R1 \\
push e & \quad R0 R1 R2 \\
- & \quad R0 R1 \\
/ & \quad R0 \\
\end{align*}
\]

\[a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /\]

a and c are "loaded" twice
⇒ not the best use of registers!
Register Usage in a GPR Machine

\[
\frac{a + b \times c}{a + d \times c - e}
\]

More control over register usage since registers can be named explicitly

Load R0 a
Load R1 c
Load R2 b
Mul R2 R1
Add R2 R0
Load R3 d
Mul R3 R1
Add R3 R0
Load R0 e
Sub R3 R0
Div R2 R3

Reload R2
Reload R3
Reload R0

but instructions may be longer!

Stack Machines: Essential features

- In addition to push, pop, +, etc., the instruction set must provide the capability to
  - refer to any element in the data area
  - jump to any instruction in the code area
  - move any element in the stack frame to the top

machinery to carry out +, -, etc.

\[
\text{push a} \\
\text{push b} \\
\text{push c} \ast \\
\text{push e} / \\
\text{code}
\]

\[
\text{SP} \\
\text{DP} \\
\text{PC} \\
\text{stack} \\
\text{data}
\]
Stack versus GPR Organization
*Amdahl, Blaauw and Brooks, 1964*

1. The performance advantage of push down stack organization is derived from the presence of fast registers and not the way they are used.

2. “Surfacing” of data in stack which are “profitable” is approximately 50% because of constants and common subexpressions.

3. Advantage of instruction density because of implicit addresses is equaled if short addresses to specify registers are allowed.


5. Recursive subroutine advantage can be realized only with the help of an independent stack for addressing.

6. Fitting variable-length fields into fixed-width word is awkward.

Stack Machines *(Mostly) Died by 1980*

1. Stack programs are not smaller if short (Register) addresses are permitted.

2. Modern compilers can manage fast register space better than the stack discipline.

   *GPR’s and caches are better than stack and displays*

   *Early language-directed architectures often did not take into account the role of compilers!*

   *B5000, B6700, HP 3000, ICL 2900, Symbolics 3600*

   *Some would claim that an echo of this mistake is visible in the SPARC architecture register windows - more later...*
Stacks post-1980

- Inmos Transputers (1985-2000)
  - Designed to support many parallel processes in Occam language
  - Fixed-height stack design simplified implementation
  - Stack trashed on context swap (fast context switches)
  - Inmos T800 was world’s fastest microprocessor in late 80’s
- Forth machines
  - Direct support for Forth execution in small embedded real-time environments
  - Several manufacturers (Rockwell, Patriot Scientific)
- Java Virtual Machine
  - Designed for software emulation, not direct hardware execution
  - Sun PicoJava implementation + others
- Intel x87 floating-point unit
  - Severely broken stack model for FP arithmetic
  - Deprecated in Pentium-4, replaced with SSE2 FP registers

Microprogramming

- 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
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 pipelines
  - JVM ⇒ software interpretation
- But can be implemented with any microarchitectural style
  - Core 2 Duo: hardwired pipelined CISC (x86) machine (with some microcode support)
  - This lecture: a microcoded RISC (MIPS) machine
  - Intel could implement a dynamically scheduled out-of-order VLIW (IA-64) processor
  - ARM Jazelle: A hardware JVM processor
  - Simics: Software-interpreted SPARC RISC machine

Microarchitecture: Implementation of an ISA

- **Structure:** How components are connected. 
- **Behavior:** How data moves between components
  - Static
  - Dynamic
Microcontrol Unit *Maurice Wilkes, 1954*

*Embed the control logic state table in a memory array*

![Diagram of control logic state table]

Microcoded Microarchitecture

![Diagram of microcoded microarchitecture]
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

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>rd ← (rs) func (rt)</td>
</tr>
<tr>
<td>ALUi</td>
<td>rt ← (rs) op immediate</td>
</tr>
<tr>
<td>Mem</td>
<td>M[(rs) + displacement]</td>
</tr>
<tr>
<td></td>
<td>BEQZ, BNEZ</td>
</tr>
<tr>
<td></td>
<td>JR, JALR</td>
</tr>
<tr>
<td></td>
<td>J, JAL</td>
</tr>
</tbody>
</table>
A Bus-based Datapath for MIPS

Microinstruction: register to register transfer (17 control signals)

Memory Module

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: $\text{MA} \leftarrow \text{PC}$
$\text{A} \leftarrow \text{PC}$
$\text{IR} \leftarrow \text{Memory}$
$\text{PC} \leftarrow \text{A} + 4$
dispatch on OPcode

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

\text{can be treated as a macro}
Microprogram Fragments (cont.)

LW:
\[ A \leftarrow \text{Reg}[rs] \]
\[ B \leftarrow \text{Imm} \]
\[ MA \leftarrow A + B \]
\[ \text{Reg}[rt] \leftarrow \text{Memory} \]
\[ \text{do instruction fetch} \]

J:
\[ A \leftarrow \text{PC} \]
\[ B \leftarrow \text{IR} \]
\[ \text{PC} \leftarrow \text{JumpTarg}(A,B) \]
\[ \text{do instruction fetch} \]

beqz:
\[ A \leftarrow \text{Reg}[rs] \]
\[ \text{If zero?}(A) \text{ then go to bz-taken} \]
\[ \text{do instruction fetch} \]

bz-taken:
\[ A \leftarrow \text{PC} \]
\[ B \leftarrow \text{Imm} \ll 2 \]
\[ \text{PC} \leftarrow A + B \]
\[ \text{do instructionfetch} \]

JumpTarg(A,B) = \{A[31:28],B[25:0],00\}

MIPS Microcontroller: first attempt

 Opcode
     zero?  6
Busy (memory)

latching the inputs may cause a one-cycle delay

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

Word size?
= \(\text{control}+s\) bits

\(\mu\)PC (state)

\(\mu\)Program ROM

addr

data

Control Signals (17)

next state

How big is “s”? 

= s

1/27/2009
CS152-Spring’09
Microprogram in the ROM

### 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>ALU</td>
<td>*</td>
<td>*</td>
<td>PC ← A + 4</td>
<td>?</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

<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>ALU₁₀</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>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td>...</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>ALUi₂</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>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 = (4 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 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)

CS152 Administrivia

- Room change/decision:
  - Either Soda 306 (here) for all but two lectures, back in 320 for those
  - Or 182 Dwinelle for all lectures
  - Vote ?

- Class schedule almost up to date (but might need to change quiz depending on vote above)
- Lab 1 coming out on Thursday, together with PS1
Problem Sets

- Designed to help you learn material and practice for quiz
- Must hand in day before section that reviews problem set, which will be shortly before quiz
- Solutions handed out in review section
- Quiz assumes you have worked through problem set.
- Problem sets graded on 0,1,2 scale

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.
- 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

Jump Logic

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

- 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
### Instruction Fetch & ALU: *MIPS-Controller-2*

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch₀</td>
<td>MA ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch₁</td>
<td>IR ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch₂</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch₃</td>
<td>PC ← A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU₁</td>
<td>B ← Reg[rt]</td>
<td>next</td>
</tr>
<tr>
<td>ALU₂</td>
<td>Reg[rd] ← func(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>ALUᵢ₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALUᵢ₁</td>
<td>B ← sExt₁₆(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>ALUᵢ₂</td>
<td>Reg[rd] ← 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>
<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>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)</td>
<td>fetch</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)</td>
<td>fetch</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>
Implementing Complex Instructions

VAX 11-780 Microcode
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) spin</td>
</tr>
<tr>
<td>ALUMM₆</td>
<td>fetch</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 steps per macroinstruction
  - Sparser encoding ⇔ more bits

- Vertical µcode has narrower µinstructions
  - Typically a single datapath operation per µinstruction
    - separate µinstruction for branches
  - More steps to 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.,

\[ ALU_0 \ A \leftarrow \text{Reg}[rs] \]
\[ \ldots \]
\[ ALU_i \ A \leftarrow \text{Reg}[rs] \]
\[ \ldots \]

- 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

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 \( \mu \)code
  - Need for fixing bugs in control programs was in conflict with read-only nature of \( \mu \)ROM
  - \( \rightarrow \) WCS (B1700, QMachine, Intel i432, …)

- With the advent of VLSI technology assumptions about ROM & RAM speed became invalid \( \rightarrow \) 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 Opteron, Intel Core 2 Duo, 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 \( \mu \)code 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