Microprogramming

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.

Based on the material prepared by
Arvind and Krste Asanovic
ISA to Microarchitecture Mapping

• An ISA often designed for a particular microarchitectural style, e.g.,
  - CISC ⇒ microcoded
  - RISC ⇒ hardwired, pipelined
  - VLIW ⇒ fixed latency in-order pipelines
  - JVM ⇒ software interpretation

• But an ISA can be implemented in any microarchitectural style
  - Pentium-4: hardwired pipelined CISC (x86) machine (with some microcode support)
  - This lecture: a microcoded RISC (MIPS) machine
  - Intel will probably eventually have a dynamically scheduled out-of-order VLIW (IA-64) processor
  - PicoJava: A hardware JVM processor
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

op conditional code flip-flop

\[ \mu \text{ address} \]

Next state

Decoder

Matrix A Matrix B

Control lines *to* ALU, MUXs, Registers

September 21, 2005
Microcoded Microarchitecture

- **μcontroller (ROM)**: holds fixed microcode instructions
- **Datapath**: contains processes like zero?, busy?, opcode, etc.
- **Memory (RAM)**: holds user program written in macrocode instructions (e.g., MIPS, x86, etc.)
- **Controller**:
  - Microcode instructions
  - User program

September 21, 2005
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

See H&P p129-137 & Appendix C (online) for full description

All instructions are 32 bits

September 21, 2005
MIPS Instruction Formats

**ALU**

- 0 rs rt rd 0 func
- \( \text{rd} \leftarrow (\text{rs}) \text{ func} (\text{rt}) \)

**ALUi**

- | opcode | rs | rt | immediate |
- \( \text{rt} \leftarrow (\text{rs}) \text{ op} \text{ immediate} \)

**Mem**

- | opcode | rs | rt | displacement |
- \( \text{M}[(\text{rs}) + \text{displacement}] \)

**opcode**

- | 6 | 5 | 5 | 16 |
- | opcode | rs | rt | offset |

**BEQZ, BNEZ**

- | 6 | 5 | 5 | 16 |
- | opcode | rs | offset |

**JR, JALR**

- | 6 | 5 | 5 | 16 |
- | opcode | rs | offset |

**J, JAL**

- | 6 | 26 |
- | opcode | offset |

September 21, 2005
A Bus-based Datapath for MIPS

Microinstruction: register to register transfer (17 control signals)

MA ← PC means RegSel = PC; enReg=yes; IdMA = yes
B ← Reg[rt] means RegSel = rt; enReg=yes; IdB = yes
Assumption: Memory operates asynchronously and is slow as compared to Reg-to-Reg transfers.
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:**
- \( MA \leftarrow PC \)
- \( A \leftarrow PC \)
- \( IR \leftarrow Memory \)
- \( PC \leftarrow A + 4 \)

*dispatch on OPcode*

*can be treated as a macro*

**ALU:**
- \( A \leftarrow \text{Reg[rs]} \)
- \( B \leftarrow \text{Reg[rt]} \)
- \( \text{Reg[rd]} \leftarrow \text{func}(A,B) \)

*do instruction fetch*

**ALUi:**
- \( A \leftarrow \text{Reg[rs]} \)
- \( B \leftarrow \text{Imm} \)

*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

- Opcode zero?
- Busy (memory)
- Latching the inputs may cause a one-cycle delay

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

Word size?
= control+s bits

How big is “s”? 

\[ \mu \text{PC (state)} \]

\[ \mu \text{Program ROM} \]

data

\[ \text{addr} \]

Control Signals (17)

Sept. 21, 2005
### 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></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></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></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></td>
</tr>
</tbody>
</table>

September 21, 2005
## 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>ALUi₁₀</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) fetch₀</td>
<td></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>

\[\text{JumpTarg}(A,B) = \{A[31:28], B[25:0], 00\}\]
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 \( \approx (4 \text{ steps per op-group } ) \times \text{op-groups} + \text{common sequences} \)
- \( = 4 \times 8 + 10 \text{ states} = 42 \text{ states} \Rightarrow s = 6 \)

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

September 21, 2005
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)
**MIPS Controller V2**

- **Opcode** → `ext`
  - **op-group**
  - **μPC** → **μPC + 1**
  - **μPC (state)**
  - **μPCSsrc**

- **Control ROM**
  - **address**
  - **data**
  - **Control Signals (17)**

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

- **Jump Logic**

---

**September 21, 2005**
Jump Logic

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

\[ \begin{align*}
\text{next} & \quad \Rightarrow \quad \mu \text{PC} + 1 \\
\text{spin} & \quad \Rightarrow \quad \text{if (busy) then } \mu \text{PC else } \mu \text{PC} + 1 \\
\text{fetch} & \quad \Rightarrow \quad \text{absolute} \\
\text{dispatch} & \quad \Rightarrow \quad \text{op-group} \\
\text{feqz} & \quad \Rightarrow \quad \text{if (zero) then } \text{absolute else } \mu \text{PC} + 1 \\
\text{fnez} & \quad \Rightarrow \quad \text{if (zero) then } \mu \text{PC} + 1 \text{ else } \text{absolute}
\end{align*} \]
# 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: \textit{MIPS-Controller-2}

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ_0</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ_1</td>
<td></td>
<td>fnez</td>
</tr>
<tr>
<td>BEQZ_2</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ_3</td>
<td>B ← sExt_{16}(Imm&lt;&lt;2)</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ_4</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
<tr>
<td>BNEZ_0</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ_1</td>
<td></td>
<td>feqz</td>
</tr>
<tr>
<td>BNEZ_2</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ_3</td>
<td>B ← sExt_{16}(Imm&lt;&lt;2)</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ_4</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
</tbody>
</table>
# Jumps: MIPS-Controller-2

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>
Five-minute break to stretch your legs
Implementing Complex Instructions

OpSel | ldIR | ldA | ldB | zero? |
------|------|-----|-----|-------|
       | OpSel | ldA | ldB |

RegSel

32(PC)
31(Reg)
rd rt
rs
RegSel

Memory

MemWrt
enMem

 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

September 21, 2005
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>ALUMM(_0)</td>
<td>MA ← Reg[rs]</td>
</tr>
<tr>
<td>ALUMM(_1)</td>
<td>A ← Memory</td>
</tr>
<tr>
<td>ALUMM(_2)</td>
<td>MA ← Reg[rt]</td>
</tr>
<tr>
<td>ALUMM(_3)</td>
<td>B ← Memory</td>
</tr>
<tr>
<td>ALUMM(_4)</td>
<td>MA ← Reg[rd]</td>
</tr>
<tr>
<td>ALUMM(_5)</td>
<td>Memory ← func(A,B)</td>
</tr>
<tr>
<td>ALUMM(_6)</td>
<td></td>
</tr>
</tbody>
</table>

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

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

September 21, 2005
Performance Issues

Microprogrammed control
⇒ multiple cycles per instruction

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

Given complex control, \( t_{\text{ALU}} \) & \( t_{\text{RAM}} \) can be broken into multiple cycles. However, \( t_{\mu\text{ROM}} \) cannot be broken down. Hence
\[ t_C > \max(t_{\text{reg-reg}}, t_{\mu\text{ROM}}) \]

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

Good performance, relative to the 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

September 21, 2005
Nanocoding

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

\[
\begin{align*}
\text{ALU}_0 & \text{ } A \leftarrow \text{Reg[rs]} \\
\vdots & \\
\text{ALU}_i & \text{ } A \leftarrow \text{Reg[rs]} \\
\vdots &
\end{align*}
\]

- 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
Some more history ...

- IBM 360
- Microcoding through the seventies
- Microcoding now
### 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 minsts)</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 with SRAM 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 process

• 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: *late seventies*

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

- **Micromachines became more complicated**
  - Micromachines were pipelined to overcome slower ROM
  - Complex instruction sets led to the 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
    \[ \Rightarrow WCS \ (B1700, QMachine, Intel432, ...) \]

- Introduction of caches and buffers, especially for instructions, made multiple-cycle execution of reg-reg instructions unattractive
Modern Usage

- *Microprogramming is far from extinct*

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

- Microcode pays an assisting role in most modern CISC micros (*AMD Athlon, Intel Pentium-4 …*):
  - Most instructions are executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke the microcode engine

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