Microprogramming

Krste Asanovic
Laboratory for Computer Science
Massachusetts Institute of Technology
Instruction Set Architecture (ISA) versus Implementation

- ISA is the hardware/software interface
  - Defines set of programmer visible state
  - Defines instruction format (bit encoding) and instruction semantics
  - Examples: DLX, x86, IBM 360, JVM

- Many possible implementations of one ISA
  - 360 implementations: model 30 (c. 1964), z900 (c. 2001)
  - x86 implementations: 8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4 (c. 2000), AMD Athlon, Transmeta Crusoe, SoftPC
  - DLX implementations: microcoded, pipelined, superscalar
  - JVM: HotSpot, PicoJava, ARM Jazelle, ...
ISA to Microarchitecture Mapping

- ISA often designed for particular microarchitectural style, e.g.,
  - CISC ISAs designed for microcoded implementation
  - RISC ISAs designed for hardwired pipelined implementation
  - VLIW ISAs designed for fixed latency in-order pipelines
  - JVM ISA designed for software interpreter

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

*Microcode* instructions fixed in ROM inside microcontroller

Memory (RAM) holds user program written using *macrocode* instructions (e.g., DLX, x86, etc.)
A Bus-based Datapath for DLX

Microinstruction: register to register transfer (17 control signals)

MA $\leftarrow$ PC means RegSel = PC; enReg=yes; IdMA= yes
B $\leftarrow$ Reg[rf2] means RegSel = rf2; enReg=yes; IdB = yes
Instruction Execution

Execution of a DLX instruction involves

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back to register file (optional)

and the computation of the address of the next instruction
### Microprogram Fragments

**instr fetch:**
- \( MA \leftarrow PC \)
- \( IR \leftarrow \text{Memory} \)
- \( A \leftarrow PC \)
- \( PC \leftarrow A + 4 \)

*can be treated as a macro*

**dispatch on OPcode**

**ALU:**
- \( A \leftarrow \text{Reg}[rf1] \)
- \( B \leftarrow \text{Reg}[rf2] \)
- \( \text{Reg}[rf3] \leftarrow \text{func}(A,B) \)
- *do instruction fetch*

**ALUi:**
- \( A \leftarrow \text{Reg}[rf1] \)
- \( B \leftarrow \text{Imm} \quad \text{sign extention ...} \)
- \( \text{Reg}[rf2] \leftarrow \text{Opcode}(A,B) \)
- *do instruction fetch*
Microprogram Fragments (cont.)

LW:  
A ← Reg[rf1]  
B ← Imm  
MA ← A + B  
Reg[rf2] ← Memory  
do instruction fetch

J:  
A ← PC  
B ← Imm  
PC ← A + B  
do instruction fetch

beqz:  
A ← Reg[rf1]  
If zero?(A) then go to bz-taken  
do instruction fetch

bz-taken:  
A ← PC  
B ← Imm  
PC ← A + B  
do instruction fetch
DLX Microcontroller: first attempt

- Opcode zero?
- Busy (memory)

latching the inputs may cause a one-cycle delay

μPC (state)

ROM Size?
How big is “s”?

μProgram ROM

2^{(opcode+status+s)} words

word = (control+s) bits

addr
data

next state

17 Control Signals
### Microprogram in the ROM

**worksheet**

<table>
<thead>
<tr>
<th>State</th>
<th>Op</th>
<th>zero?</th>
<th>busy</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch(_0)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>MA $\leftarrow$ PC</td>
<td>fetch(_1)</td>
</tr>
<tr>
<td>fetch(_1)</td>
<td>*</td>
<td>*</td>
<td>yes</td>
<td>IR $\leftarrow$ Memory</td>
<td>fetch(_1)</td>
</tr>
<tr>
<td>fetch(_1)</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>A $\leftarrow$ PC</td>
<td>fetch(_2)</td>
</tr>
<tr>
<td>fetch(_2)</td>
<td>*</td>
<td>*</td>
<td></td>
<td>PC $\leftarrow$ A + 4</td>
<td>?</td>
</tr>
<tr>
<td>fetch(_3)</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| ALU\(_0\) | * | *     |     | A $\leftarrow$ Reg[rf1]  | ALU\(_1\)  |
| ALU\(_1\) | * | *     |     | B $\leftarrow$ Reg[rf2]  | ALU\(_2\)  |
| ALU\(_2\) | * | *     |     | Reg[rf3] $\leftarrow$ func(A,B) | fetch\(_0\) |
## 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>ALU₀</td>
<td>*</td>
<td></td>
<td>PC ← A + 4</td>
<td>ALU₀</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[rf1]</td>
<td>ALU₁</td>
</tr>
<tr>
<td>ALU₁</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>B ← Reg[rf2]</td>
<td>ALU₂</td>
</tr>
<tr>
<td>ALU₂</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Reg[rf3] ← 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[rf1]</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[rf3] ← 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 ← sExt₂₆(Imm)</td>
<td>J₂</td>
</tr>
<tr>
<td>J₂</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>
<tr>
<td>beqz₀</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>A ← Reg[rf1]</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>
Size of Control Store

status & opcode

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

\(\mu PC\)

addr

Control ROM

data

Control signals

\(\mu PC\)

next \(\mu PC\)

\(w\)

\(c\)

DLX

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

no. of steps per opcode = 4 to 6 + fetch-sequence

no. of states \(\approx (4\) steps per op-group \( ) \times \) op-groups + common sequences

= 4 \times 8 + 10 states = 42 states

\(\Rightarrow s = 6\)

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

Control store has to be fast \( \Rightarrow \) expensive

- Reduce the ROM height (= address bits)
  \( \Rightarrow \) *reduce inputs by extra external logic*
  each input bit doubles the size of the control store
  \( \Rightarrow \) *reduce states by grouping opcodes*
  find common sequences of actions
  \( \Rightarrow \) *condense input status bits*
  combine all exceptions into one, i.e., exception/no-exception

- Reduce the ROM width
  \( \Rightarrow \) *restrict the next-state encoding*
  Next, Dispatch on opcode, Wait for memory, ...
  \( \Rightarrow \) *encode control signals (vertical microcode)*
DLX Controller V2

Opcode → ext

absolute

op-group

µPC

µPC+1

+1

µPCSrc

Reduced ROM height by encoding inputs

address

Control ROM

data

µJumpType

17 Control Signals

Reduce ROM width by encoding next-state

JumpType

(next, spin, fetch,
dispatch, feqz, fnez)
Jump Logic

\[ \mu_{PC}^{Src} = \text{Case } \mu_{JumpTypes} \]

- **next** \(\Rightarrow\) \(\mu_{PC}+1\)
- **spin** \(\Rightarrow\) if (busy) then \(\mu_{PC}\) else \(\mu_{PC}+1\)
- **fetch** \(\Rightarrow\) absolute
- **dispatch** \(\Rightarrow\) op-group
- **feqz** \(\Rightarrow\) if (zero) then absolute else \(\mu_{PC}+1\)
- **fnez** \(\Rightarrow\) if (zero) then \(\mu_{PC}+1\) else absolute
### Instruction Fetch & ALU: DLX-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[rf1]</td>
<td>next</td>
</tr>
<tr>
<td>ALU₁</td>
<td>B ← Reg[rf2]</td>
<td>next</td>
</tr>
<tr>
<td>ALU₂</td>
<td>Reg[rf3] ← func(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>ALUᵢ₀</td>
<td>A ← Reg[rf1]</td>
<td>next</td>
</tr>
<tr>
<td>ALUᵢ₁</td>
<td>B ← sExt₁₆(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>ALUᵢ₂</td>
<td>Reg[rf3] ← Op(A,B)</td>
<td>fetch</td>
</tr>
</tbody>
</table>
## Load & Store: DLX-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[rf1]</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[rf2] ← 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[rf1]</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[rf2]</td>
<td>spin</td>
</tr>
<tr>
<td>SW₄</td>
<td></td>
<td>fetch</td>
</tr>
</tbody>
</table>
## Branches: DLX-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[rf1]</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)</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[rf1]</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)</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ₄</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
</tbody>
</table>
### Jumps: DLX-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 ← sExt_{26}(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>J₂</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
<tr>
<td>JR₀</td>
<td>PC ← Reg[rf1]</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 ← sExt_{26}(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>JAL₃</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
<tr>
<td>JALR₀</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>JALR₁</td>
<td>Reg[31] ← A</td>
<td>next</td>
</tr>
<tr>
<td>JALR₂</td>
<td>PC ← Reg[rf1]</td>
<td>fetch</td>
</tr>
</tbody>
</table>
# Microprogramming in IBM 360

<table>
<thead>
<tr>
<th></th>
<th>M30</th>
<th>M40</th>
<th>M50</th>
<th>M65</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datapath width (bits)</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>(\mu)inst width (bits)</td>
<td>50</td>
<td>52</td>
<td>85</td>
<td>87</td>
</tr>
<tr>
<td>(\mu)code size (K (\mu)insts)</td>
<td>4</td>
<td>4</td>
<td>2.75</td>
<td>2.75</td>
</tr>
<tr>
<td>(\mu)store technology</td>
<td>CCROS</td>
<td>TCROS</td>
<td>BCROS</td>
<td>BCROS</td>
</tr>
<tr>
<td>(\mu)store cycle (ns)</td>
<td>750</td>
<td>625</td>
<td>500</td>
<td>200</td>
</tr>
<tr>
<td>memory cycle (ns)</td>
<td>1500</td>
<td>2500</td>
<td>2000</td>
<td>750</td>
</tr>
<tr>
<td>Rental fee ($K/month)</td>
<td>4</td>
<td>7</td>
<td>15</td>
<td>35</td>
</tr>
</tbody>
</table>

- Only fastest models (75 and 95) were hardwired
Horizontal vs Vertical \( \mu \)Code

- **Horizontal \( \mu \) code has longer \( \mu \) instructions**
  - Can specify multiple parallel operations per \( \mu \) instruction
  - Needs fewer steps to complete each macroinstruction
  - Sparser encoding \( \Rightarrow \) more bits

- **Vertical \( \mu \) code has more, narrower \( \mu \) instructions**
  - In limit, only single datapath operation per \( \mu \) instruction
  - \( \mu \) code branches require separate \( \mu \) instruction
  - More steps to complete each macroinstruction
  - More compact \( \Rightarrow \) less bits

- **Nanocoding**
  - Tries to combine best of horizontal and vertical \( \mu \) code
Nanocoding

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

\[
\begin{align*}
\text{ALU}_0 & \quad A \leftarrow \text{Reg}[rf1] \\
& \quad \text{...}
\end{align*}
\]

\[
\begin{align*}
\text{ALU}_i & \quad A \leftarrow \text{Reg}[rf1] \\
& \quad \text{...}
\end{align*}
\]

- MC68000 had 17-bit \( \mu \)code containing either 10-bit \( \mu \)jump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals
Implementing Complex Instructions

- Opcode
- zero?
- busy

IdIR → ALUop → IdA → IdB

Imm → Ext

ExSel → 32 GPRs + PC ...

addr → ALU

RegSel

RegWrt → M[rf3] ← M[(rf1)] op (rf2)

M[(rf3)] ← (rf1) op (rf2)

M[(rf3)] ← M[(rf1)] op M[(rf2)]

Reg-Memory-src ALU op

Reg-Memory-dst ALU op

Mem-Mem ALU op
Mem-Mem ALU Instructions: DLX-Controller-2

Mem-Mem ALU op  \[ M[(rf3)] \leftarrow M[(rf1)] \text{ op } M[(rf2)] \]

\begin{align*}
\text{ALUMM}_0 & : \quad \text{MA} \leftarrow \text{Reg}[rf1] \quad \text{next} \\
\text{ALUMM}_1 & : \quad A \leftarrow \text{Memory} \quad \text{spin} \\
\text{ALUMM}_2 & : \quad \text{MA} \leftarrow \text{Reg}[rf2] \quad \text{next} \\
\text{ALUMM}_3 & : \quad B \leftarrow \text{Memory} \quad \text{spin} \\
\text{ALUMM}_4 & : \quad \text{MA} \leftarrow \text{Reg}[rf3] \quad \text{next} \\
\text{ALUMM}_5 & : \quad \text{Memory} \leftarrow \text{func}(A,B) \quad \text{spin} \\
\text{ALUMM}_6 &
\end{align*}

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
Microcode Emulation

- IBM initially miscalculated importance of software compatibility when introducing 360 series
- Honeywell started effort to steal IBM 1401 customers by offering translation software (“Liberator”) for Honeywell H200 series machine
- IBM retaliates with optional additional microcode for 360 series that can 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->1401->360)
Microprogramming in the Seventies

Thrived because:

- 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 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 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
- Hard 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
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 \( 10 \cdot t_{\mu\text{ROM}} < t_{\text{RAM}} \)
good performance, relative to the single-cycle hardwired implementation, can be achieved even with a CPI of 10
VLSI & Microprogramming

*By late seventies*
- technology assumption about ROM & RAM speed became invalid

- micromachines became more complicated
  - to overcome slower ROM, micromachines were pipelined
  - 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
  \[ \Rightarrow \text{WCS (B1700, QMachine, Intel432, ...)} \]
- 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 is present in most modern CISC micros in an assisting role (e.g. 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