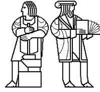
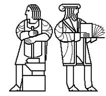


Microprogramming

Krste Asanovic
Laboratory for Computer Science
Massachusetts Institute of Technology

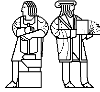


Instruction Set Architecture (ISA) versus Implementation

- **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

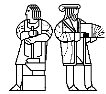
- ISA often designed for particular microarchitectural style, e.g.,
  - CISC ISAs designed for microcoded implementation
  - RISC ISAs designed for hardwired pipelined implementation
  - VLIW ISAs designed for fixed latency in-order pipelines
  - JVM ISA designed for software interpreter
- But ISA can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



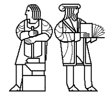
A Bus-based Datapath for DLX

Opcode
zero?
busy
ldIR
OpSel
ldA
ldB
32(PC)
31(Link)
rf3
rf2
rf1
ldMA
2
RegSel
3
MA
IR
rf3
rf2
rf1
A
B
addr
addr
32 GPRs
+ PC ...
32-bit Reg
data
Memory
data
ExtSel
2
Imm
Ext
ALU
control
ALU
RegWrt
enReg
MemWrt
enMem
enImm
enALU
Bus
32

Microinstruction: register to register transfer (17 control signals)

MA ← PC ***means*** **RegSel = PC; enReg=yes; ldMA= yes**

B ← Reg[rf2] ***means*** **RegSel = rf2; enReg=yes; ldB = 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*

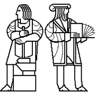
Microprogram Fragments

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

can be treated as a macro

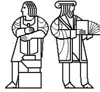
ALU: A ← Reg[rf1]
B ← Reg[rf2]
Reg[rf3] ← func(A,B)
do instruction fetch

ALUi: A ← Reg[rf1]
B ← Imm *sign extention ...*
Reg[rf2] ← Opcode(A,B)
do instruction fetch



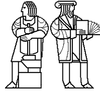
Microprogram Fragments *(cont.)*

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



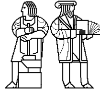
DLX Microcontroller: *first attempt*

ROM Size ?
How big is "s"?



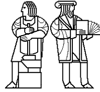
Microprogram in the ROM
worksheet

State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
$fetch_3$	*	*	*	PC ← A + 4	?
ALU_0	*	*	*	A ← Reg[rf1]	ALU_1
ALU_1	*	*	*	B ← Reg[rf2]	ALU_2
ALU_2	*	*	*	Reg[rf3]← func(A,B)	$fetch_0$



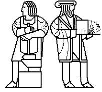
Microprogram in the ROM

State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
$fetch_3$	ALU	*	*	PC ← A + 4	ALU_0
$fetch_3$	ALUi	*	*	PC ← A + 4	$ALUi_0$
$fetch_3$	LW	*	*	PC ← A + 4	LW_0
$fetch_3$	SW	*	*	PC ← A + 4	SW_0
$fetch_3$	J	*	*	PC ← A + 4	J_0
$fetch_3$	JAL	*	*	PC ← A + 4	JAL_0
$fetch_3$	JR	*	*	PC ← A + 4	JR_0
$fetch_3$	JALR	*	*	PC ← A + 4	$JALR_0$
$fetch_3$	beqz	*	*	PC ← A + 4	$beqz_0$
...					
ALU_0	*	*	*	A ← Reg[rf1]	ALU_1
ALU_1	*	*	*	B ← Reg[rf2]	ALU_2
ALU_2	*	*	*	Reg[rf3]← func(A,B)	$fetch_0$

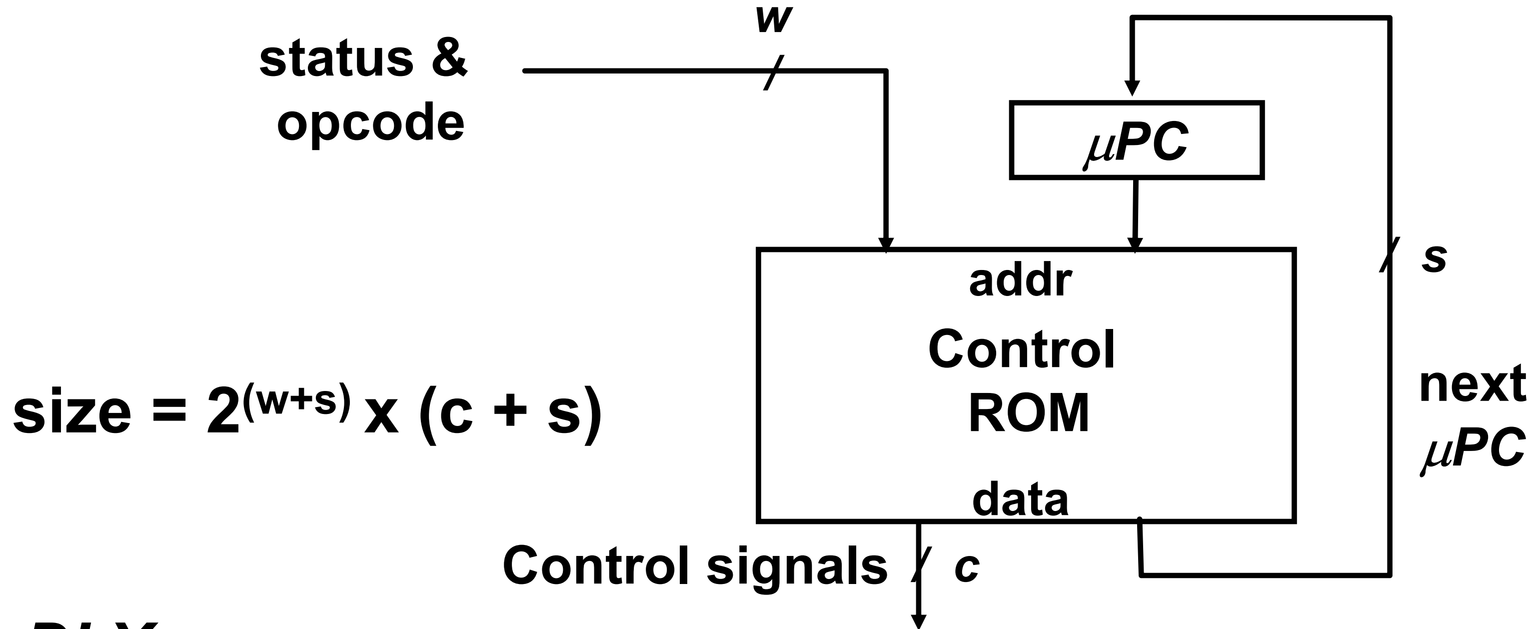


Microprogram in the ROM
Cont.

State	Op	zero?	busy	Control points	next-state
$ALUi_0$	*	*	*	$A \leftarrow Reg[rf1]$	$ALUi_1$
$ALUi_1$	sExt	*	*	$B \leftarrow sExt_{16}(Imm)$	$ALUi_2$
$ALUi_1$	uExt	*	*	$B \leftarrow uExt_{16}(Imm)$	$ALUi_2$
$ALUi_2$	*	*	*	$Reg[rf3] \leftarrow Op(A,B)$	$fetch_0$
...					
J_0	*	*	*	$A \leftarrow PC$	J_1
J_1	*	*	*	$B \leftarrow sExt_{26}(Imm)$	J_2
J_2	*	*	*	$PC \leftarrow A+B$	$fetch_0$
...					
$beqz_0$	*	*	*	$A \leftarrow Reg[rf1]$	$beqz_1$
$beqz_1$	*	yes	*	$A \leftarrow PC$	$beqz_2$
$beqz_1$	*	no	*	....	$fetch_0$
$beqz_2$	*	*	*	$B \leftarrow sExt_{16}(Imm)$	$beqz_3$
$beqz_3$	*	*	*	$PC \leftarrow A+B$	$fetch_0$
...					



Size of Control Store



size = $2^{(w+s)}$ x (c + s)

DLX

w = 6+2 c = 17 s = ?

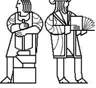
no. of steps per opcode = 4 to 6 + fetch-sequence

no. of states ≈ (4 steps per op-group) x op-groups + common sequences

= 4 x 8 + 10 states = 42 states

⇒ s = 6

Control ROM = $2^{(8+6)}$ x 23 bits ≈ 48 Kbytes

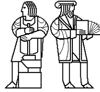


Reducing Size of Control Store

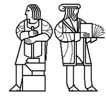
Control store has to be *fast* ⇒ *expensive*

- 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)*



DLX Controller V2



Jump Logic

μPCSrc = *Case* μJumpTypes

next	⇒	μPC+1
spin	⇒	if (busy) then μPC else μPC+1
fetch	⇒	absolute
dispatch	⇒	op-group
feqz	⇒	if (zero) then absolute else μPC+1
fnez	⇒	if (zero) then μPC+1 else absolute

Instruction Fetch & ALU:

DLX-Controller-2

State	Control points	next-state
$fetch_0$	MA ← PC	next
$fetch_1$	IR ← Memory	spin
$fetch_2$	A ← PC	next
$fetch_3$	PC ← A + 4	dispatch
...		
ALU_0	A ← Reg[rf1]	next
ALU_1	B ← Reg[rf2]	next
ALU_2	Reg[rf3]← func(A,B)	fetch
$ALUi_0$	A ← Reg[rf1]	next
$ALUi_1$	B ← $sExt_{16}$(Imm)	next
$ALUi_2$	Reg[rf3]← Op(A,B)	fetch

Load & Store: *DLX-Controller-2*

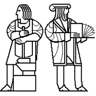
State	Control points	next-state
LW_0	$A \leftarrow Reg[rf1]$	next
LW_1	$B \leftarrow sExt_{16}(Imm)$	next
LW_2	$MA \leftarrow A+B$	next
LW_3	$Reg[rf2] \leftarrow Memory$	spin
LW_4		fetch
SW_0	$A \leftarrow Reg[rf1]$	next
SW_1	$B \leftarrow sExt_{16}(Imm)$	next
SW_2	$MA \leftarrow A+B$	next
SW_3	$Memory \leftarrow Reg[rf2]$	spin
SW_4		fetch

Branches: *DLX-Controller-2*

State	Control points	next-state
$BEQZ_0$	$A \leftarrow$ Reg[rf1]	next
$BEQZ_1$		fnez
$BEQZ_2$	$A \leftarrow PC$	next
$BEQZ_3$	$B \leftarrow sExt_{16}(Imm)$	next
$BEQZ_4$	$PC \leftarrow A+B$	fetch
$BNEZ_0$	$A \leftarrow$ Reg[rf1]	next
$BNEZ_1$		feqz
$BNEZ_2$	$A \leftarrow PC$	next
$BNEZ_3$	$B \leftarrow sExt_{16}(Imm)$	next
$BNEZ_4$	$PC \leftarrow A+B$	fetch

Jumps: *DLX-Controller-2*

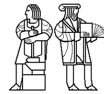
State	Control points	next-state
J_0	$A \leftarrow PC$	next
J_1	$B \leftarrow sExt_{26}(Imm)$	next
J_2	$PC \leftarrow A+B$	fetch
JR_0	$PC \leftarrow Reg[rf1]$	fetch
JAL_0	$A \leftarrow PC$	next
JAL_1	$Reg[31] \leftarrow A$	next
JAL_2	$B \leftarrow sExt_{26}(Imm)$	next
JAL_3	$PC \leftarrow A+B$	fetch
$JALR_0$	$A \leftarrow PC$	next
$JALR_1$	$Reg[31] \leftarrow A$	next
$JALR_2$	$PC \leftarrow Reg[rf1]$	fetch



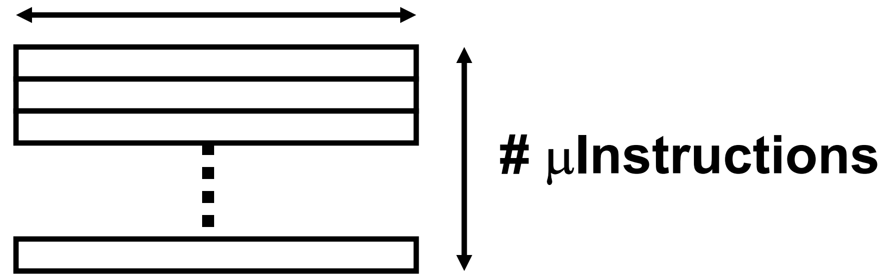
Microprogramming in IBM 360

	M30	M40	M50	M65
Datapath width (bits)	8	16	32	64
μinst width (bits)	50	52	85	87
μcode size (K μinsts)	4	4	2.75	2.75
μstore technology	CCROS	TCROS	BCROS	BCROS
μstore cycle (ns)	750	625	500	200
memory cycle (ns)	1500	2500	2000	750
Rental fee ($K/month)	4	7	15	35

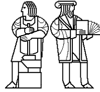
- Only fastest models (75 and 95) were hardwired



Horizontal vs Vertical μCode



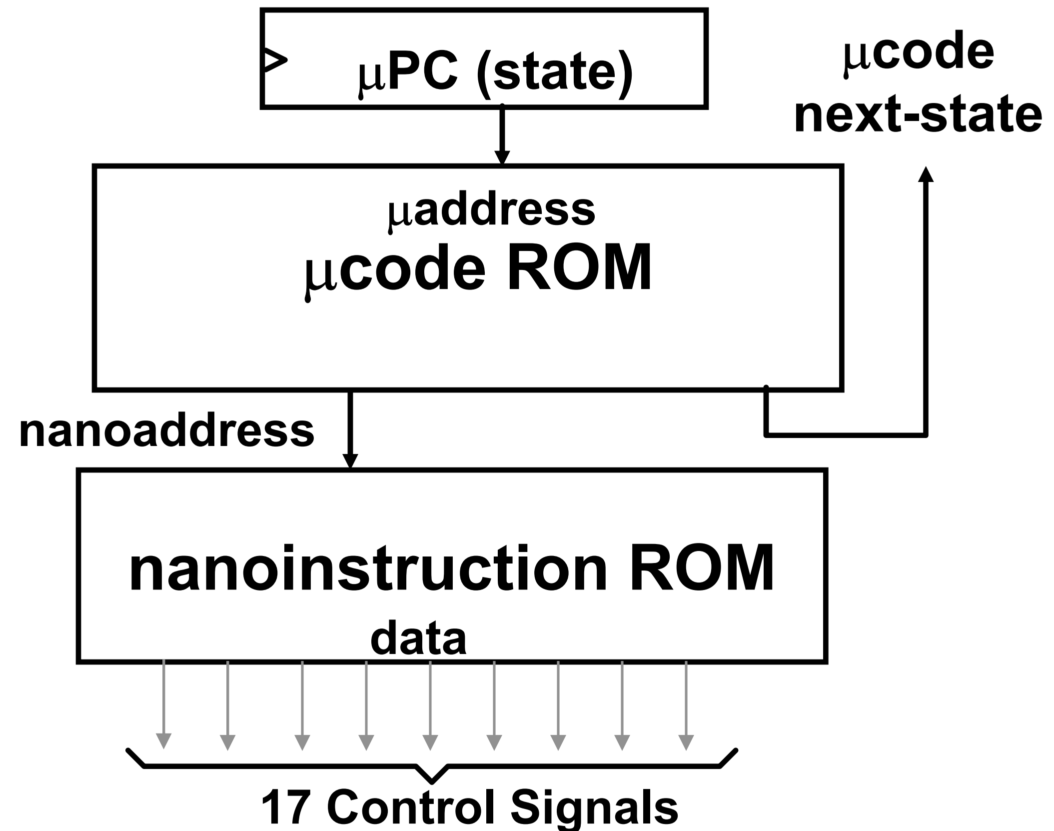
- Horizontal μcode has longer μinstructions
  - Can specify multiple parallel operations per μinstruction
  - Needs fewer steps to complete each macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical μcode has more, narrower μinstructions
  - In limit, only single datapath operation per μinstruction
  - μcode branches require separate μinstruction
  - More steps to complete each macroinstruction
  - More compact ⇒ less bits
- Nanocoding
  - Tries to combine best of horizontal and vertical μcode



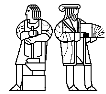
Nanocoding

Exploits recurring control signal patterns in μcode, e.g.,

ALU_0 A ← Reg[rf1]
...
$ALUi_0$ A ← Reg[rf1]
...



- **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**



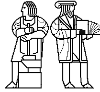
Implementing Complex Instructions

Opcode
zero?
busy
ldIR
ALUop
ldA
ldB
32(PC)
31(Link)
rf3
rf2
rf1
ldMA
RegSel
MA
IR
A
B
addr
32 GPRs
+ PC ...
32-bit Reg
data
addr
Memory
data
ExSel
Imm
Ext
ALU
6
RegWrt
enReg
MemWrt
enMem
enImm
enALU
Bus

rf3 ← M[(rf1)] op (rf2) — *Reg-Memory-src ALU op*

M[(rf3)] ← (rf1) op (rf2) — *Reg-Memory-dst ALU op*

M[(rf3)] ← M[(rf1)] op M[(rf2)] — *Mem-Mem ALU op*



Mem-Mem ALU Instructions:

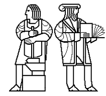
DLX-Controller-2

Mem-Mem ALU op M[(rf3)] ← M[(rf1)] op M[(rf2)]

$ALUMM_0$	MA ← Reg[rf1]	next
$ALUMM_1$	A ← Memory	spin
$ALUMM_2$	MA ← Reg[rf2]	next
$ALUMM_3$	B ← Memory	spin
$ALUMM_4$	MA ← Reg[rf3]	next
$ALUMM_5$	Memory ← func(A,B)	spin
$ALUMM_6$		fetch

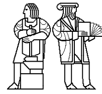
Complex instructions usually do not require datapath modifications in a microprogrammed implementation
-- only extra space for the control program

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)

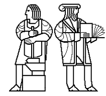


Microprogramming in the Seventies

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

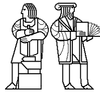
- MOS SRAM memories now almost as fast as control store (core memories/DRAMs were 10x slower)
- Bug-free microprograms difficult to write

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 <u>*restartable*</u> microcode



Performance Issues

Microprogrammed control

$\Rightarrow$ multiple cycles per instruction

Cycle time ?

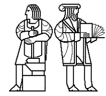
$t_C > \max(t_{reg\text{-}reg}, t_{ALU}, t_{\mu ROM}, t_{RAM})$

Given complex control, t_{ALU} & t_{RAM} can be broken into multiple cycles. However, $t_{\mu ROM}$ cannot be broken down. Hence

$t_C > \max(t_{reg\text{-}reg}, t_{\mu ROM})$

Suppose $10 * t_{\mu ROM} < t_{RAM}$

good performance, relative to the single-cycle hardwired implementation, can be achieved even with a CPI of 10

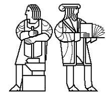


VLSI & Microprogramming

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 μcode.
  - need for fixing bugs in control programs was in conflict with read-only nature of μROM
    $\Rightarrow$ *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 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**