Microprogramming

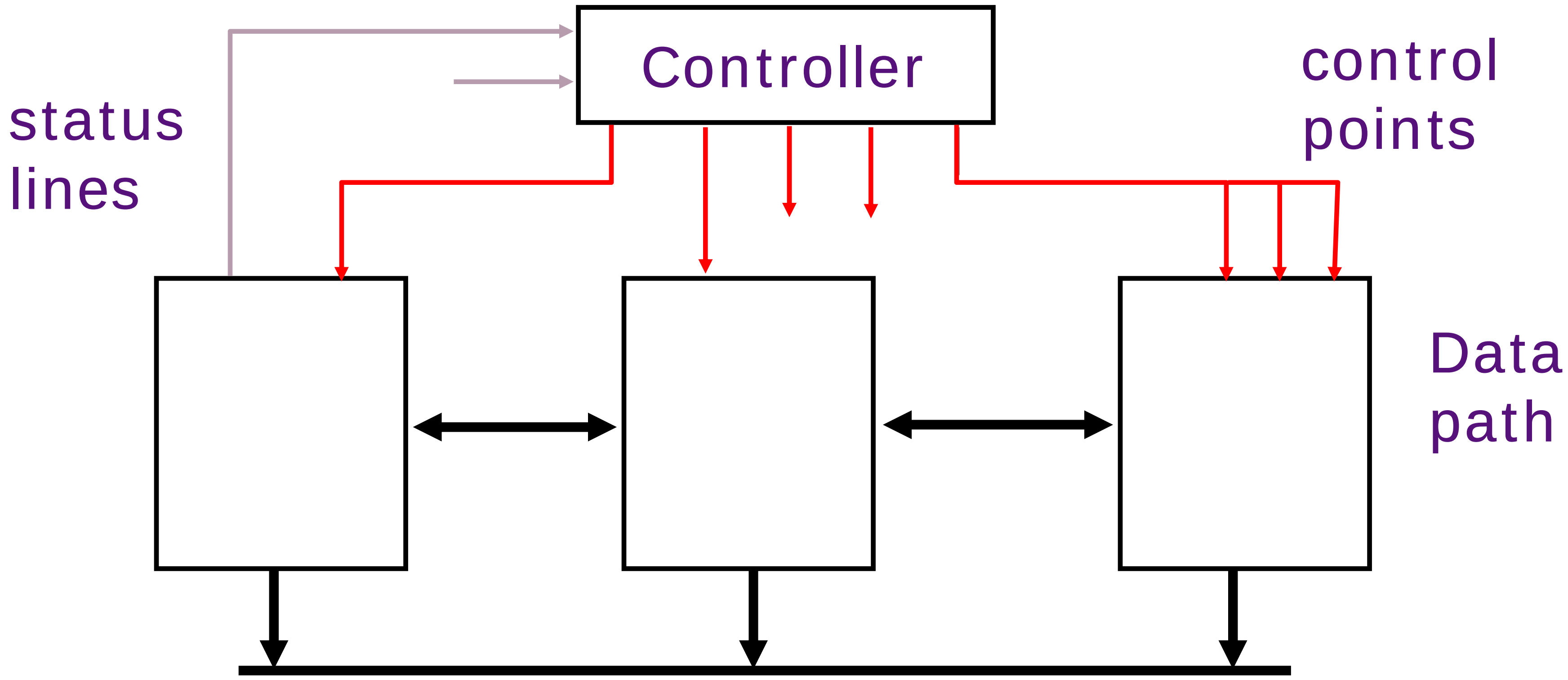
Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.

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 $\Rightarrow$ microcoded
  - RISC $\Rightarrow$ hardwired, pipelined
  - VLIW $\Rightarrow$ fixed latency in-order pipelines
  - JVM $\Rightarrow$ 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*

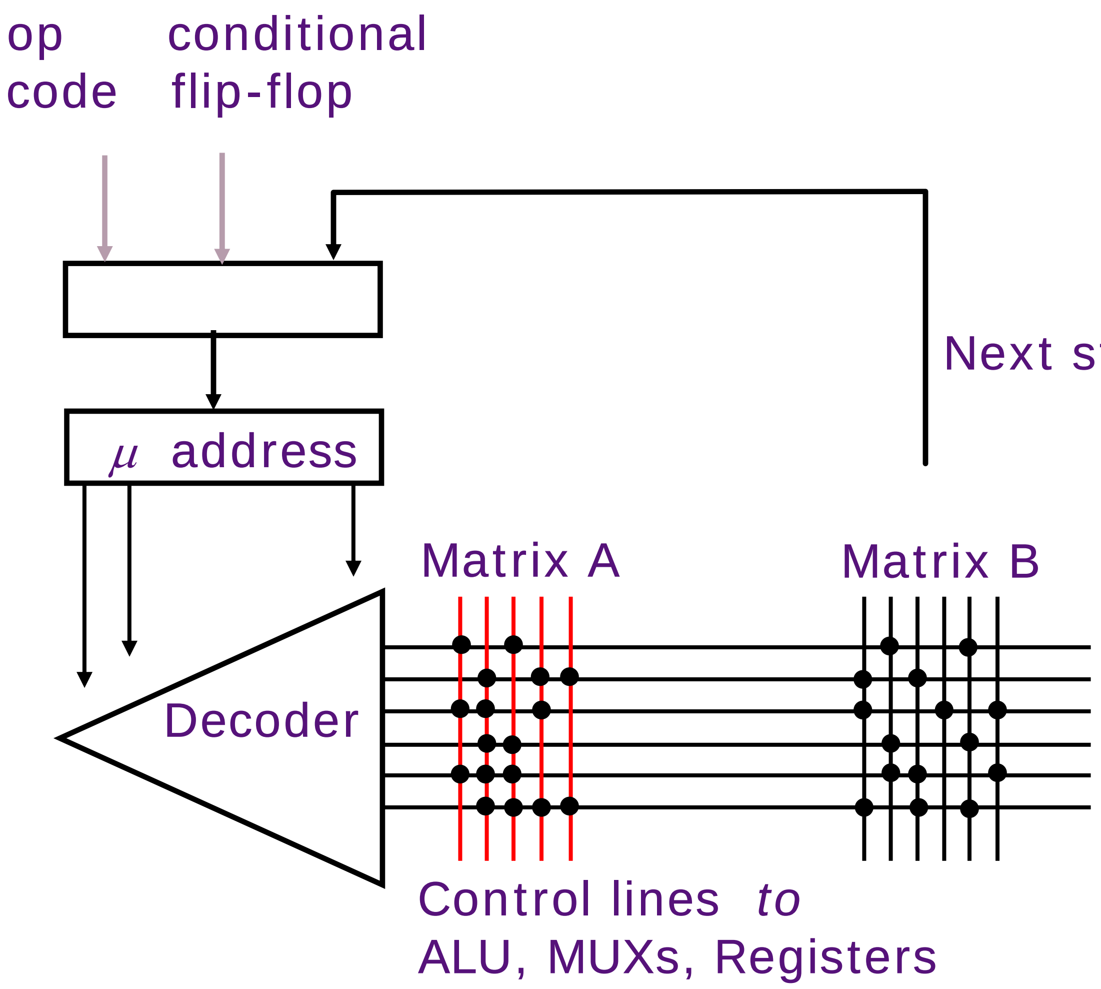


Structure: How components are connected.
Static

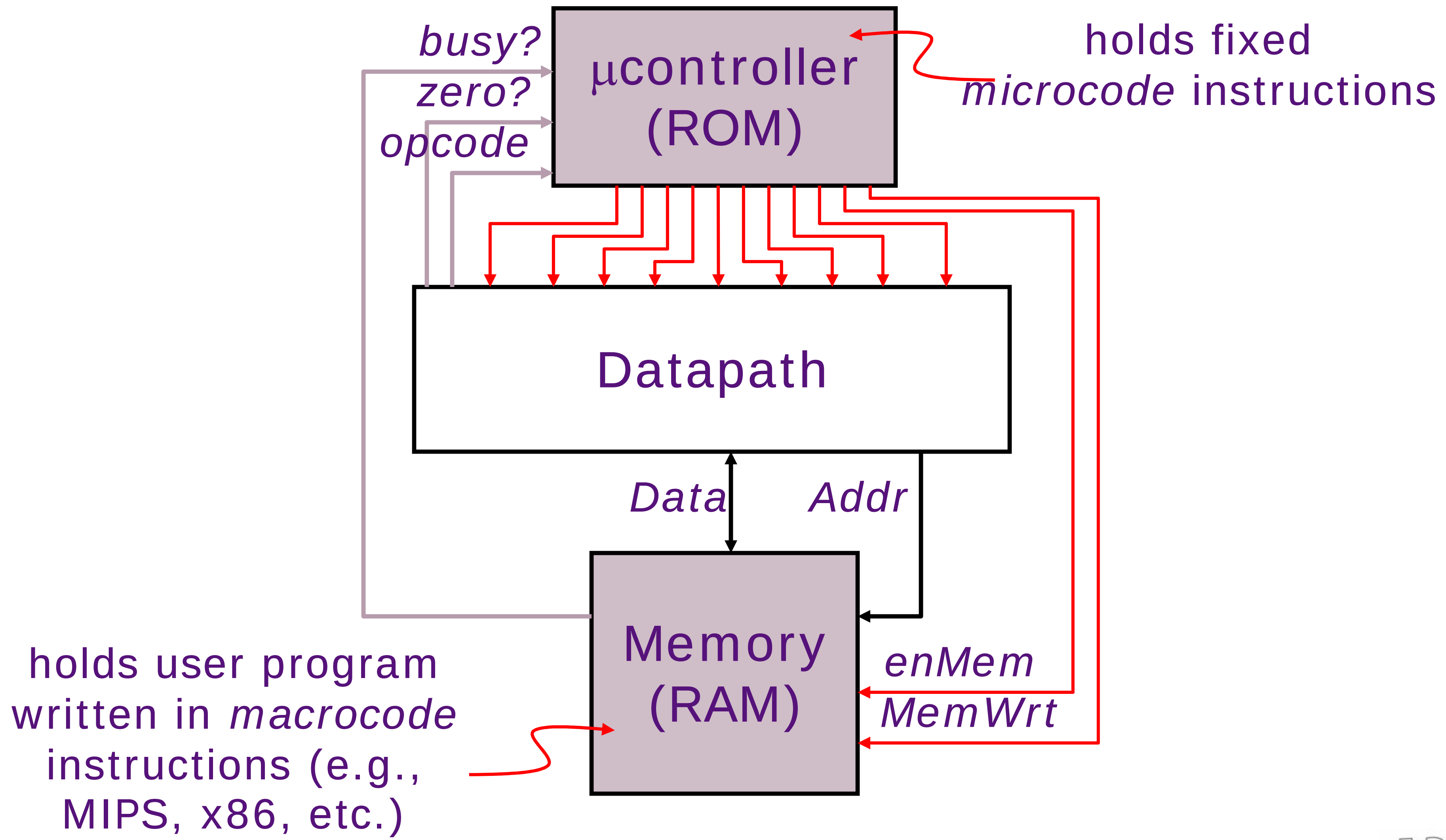
Behavior: How data moves between components
Dynamic

Microcontrol Unit *Maurice Wilkes, 1954*

Embed the control logic state table in a memory array



Microcoded Microarchitecture


busy?
zero?
opcode
μcontroller (ROM)
holds fixed *microcode* instructions
Datapath
Data
Addr
Memory (RAM)
enMem
MemWrt
holds user program written in *macrocode* instructions (e.g., MIPS, x86, etc.)

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

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

- 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

	6	5	5	5	5	6	
ALU	0	rs	rt	rd	0	func	rd ← (rs) func (rt)

	6	5	5	16	
ALUi	opcode	rs	rt	immediate	rt ← (rs) op immediate

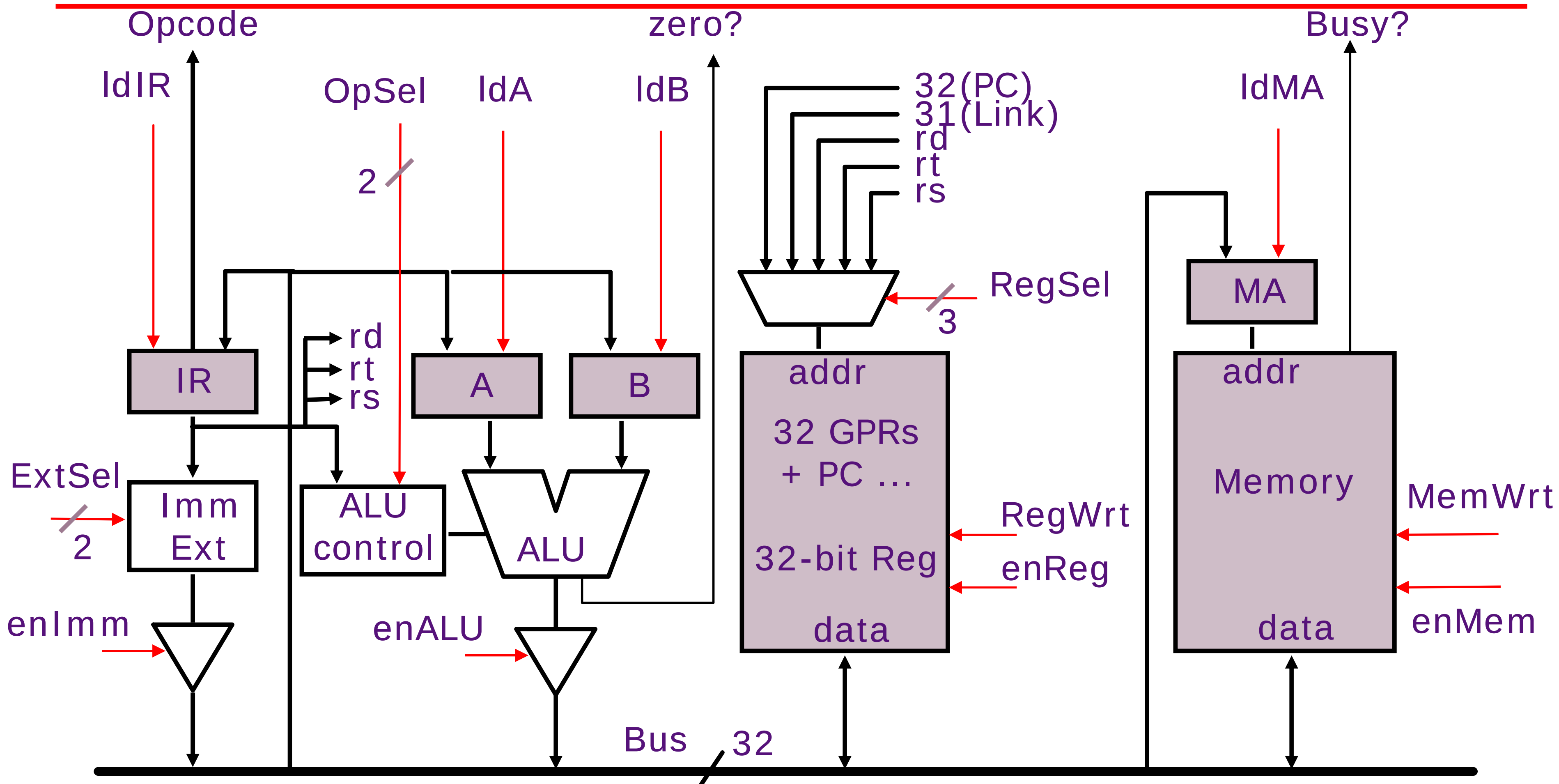
	6	5	5	16	
Mem	opcode	rs	rt	displacement	M[(rs) + displacement]

6	5	5	16	
opcode	rs		offset	BEQZ, BNEZ

6	5	5	16	
opcode	rs			JR, JALR

6	26	
opcode	offset	J, JAL

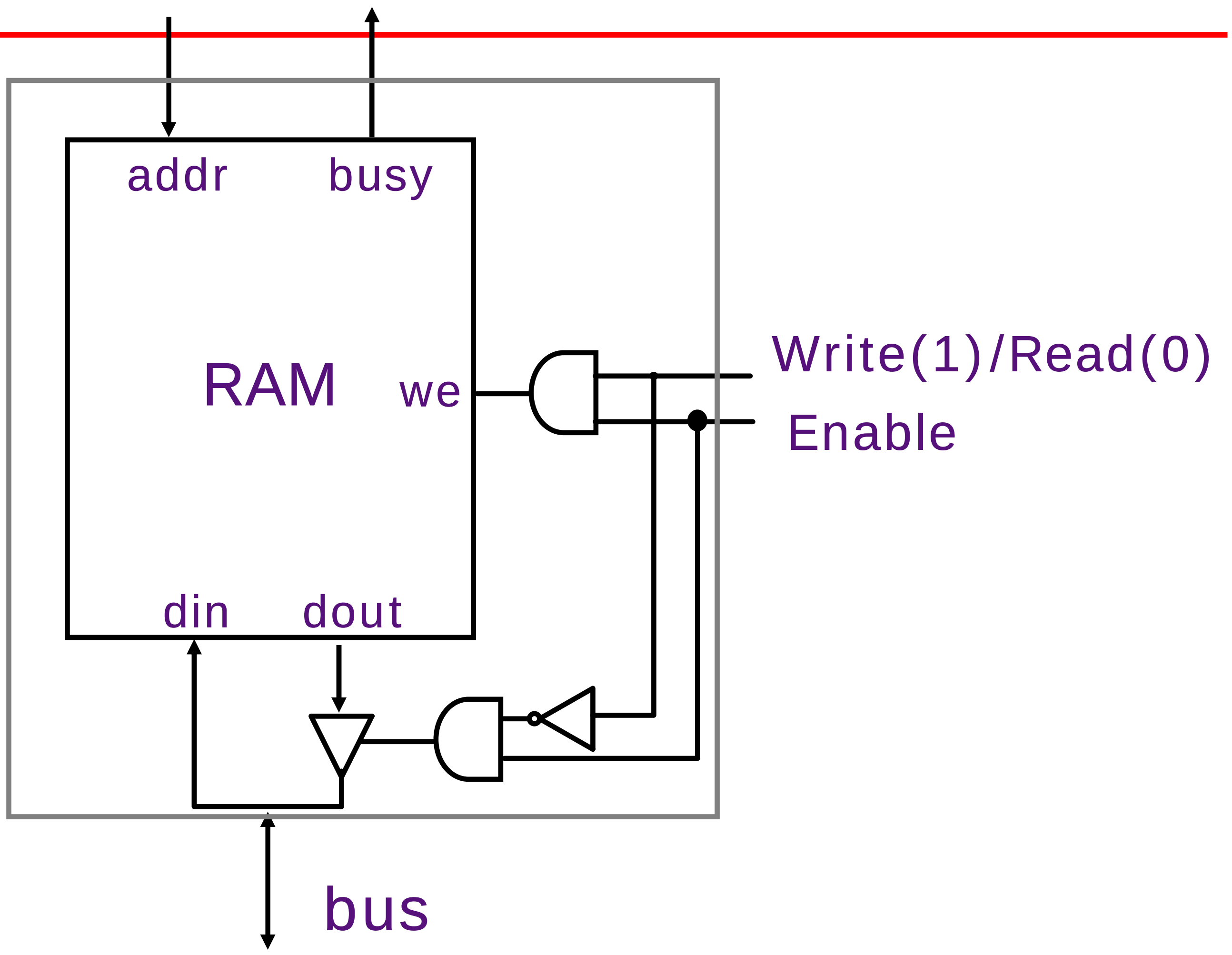
A Bus-based Datapath for MIPS



Microinstruction: register to register transfer (17 control signals)

MA	← PC	*means*	RegSel = PC;	enReg=yes;	ldMA= yes	
B	← Reg[rt]	*means*	RegSel = rt;	enReg=yes;	ldB	= yes

Memory Module



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 ← 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                  sign extension ...
                Reg[rt] ← Opcode(A,B)
                do instruction fetch
```

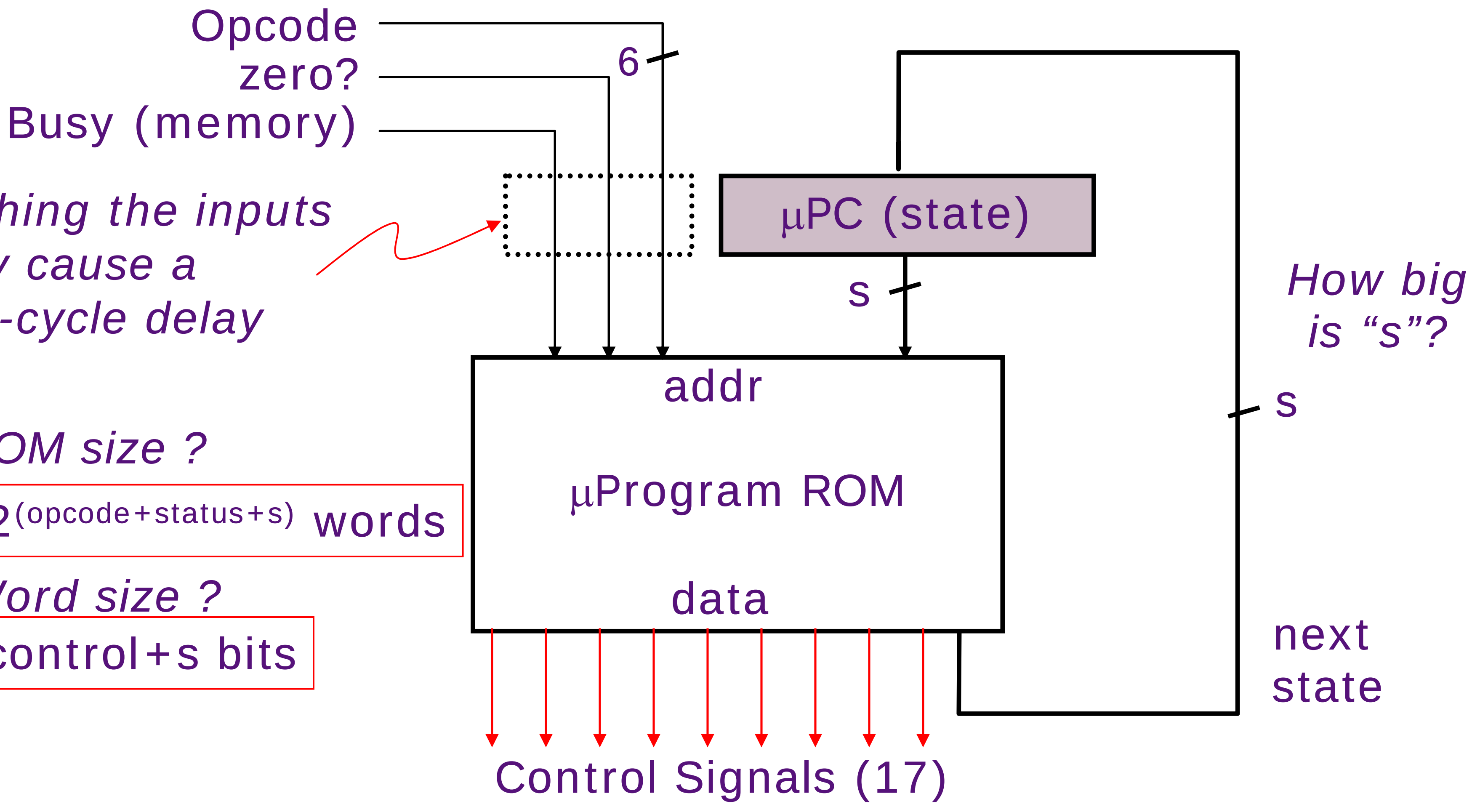
Microprogram Fragments *(cont.)*

LW:	$A \leftarrow$ Reg[rs]
	$B \leftarrow$ Imm
	$MA \leftarrow A + B$
	Reg[rt] $\leftarrow$ Memory
	do instruction fetch
J:	$A \leftarrow$ PC
	$B \leftarrow$ IR
	PC $\leftarrow$ JumpTarg(A,B)
	do instruction fetch
beqz:	$A \leftarrow$ Reg[rs]
	If zero?(A) *then go to* bz-taken
	do instruction fetch
bz-taken:	$A \leftarrow$ PC
	$B \leftarrow$ Imm << 2
	PC $\leftarrow A + B$
	do instruction fetch

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

MIPS Microcontroller: *first attempt*

latching the inputs may cause a one-cycle delay



ROM size ?

= $2^{(\text{opcode}+\text{status}+s)}$ words

Word size ?

= control+s bits

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~~	~~?~~
$fetch_3$	ALU	*	*	PC ← A + 4	ALU_0
ALU_0	*	*	*	A ← Reg[rs]	ALU_1
ALU_1	*	*	*	B ← Reg[rt]	ALU_2
ALU_2	*	*	*	Reg[rd] ← func(A,B)	$fetch_0$

Microprogram in the ROM

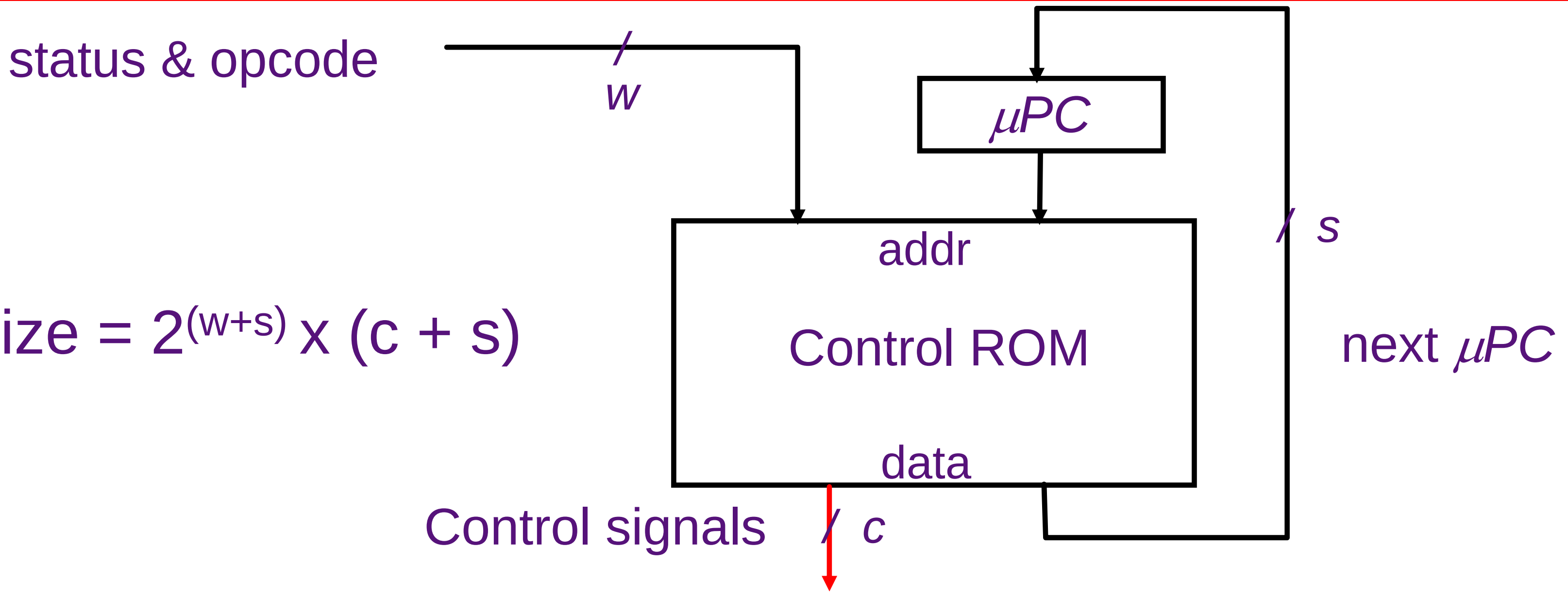
State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
$fetch_3$	ALU	*	*	PC ← A + 4	ALU_0
$fetch_3$	ALUi	*	*	PC ← A + 4	$ALUi_0$
$fetch_3$	LW	*	*	PC ← A + 4	LW_0
$fetch_3$	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[rs]	ALU_1
ALU_1	*	*	*	B ← Reg[rt]	ALU_2
ALU_2	*	*	*	Reg[rd] ← func(A,B)	$fetch_0$

Microprogram in the ROM *Cont.*

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

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

Size of Control Store



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) x op-groups + common sequences

= 4 x 8 + 10 states = 42 states ⇒ s = 6

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

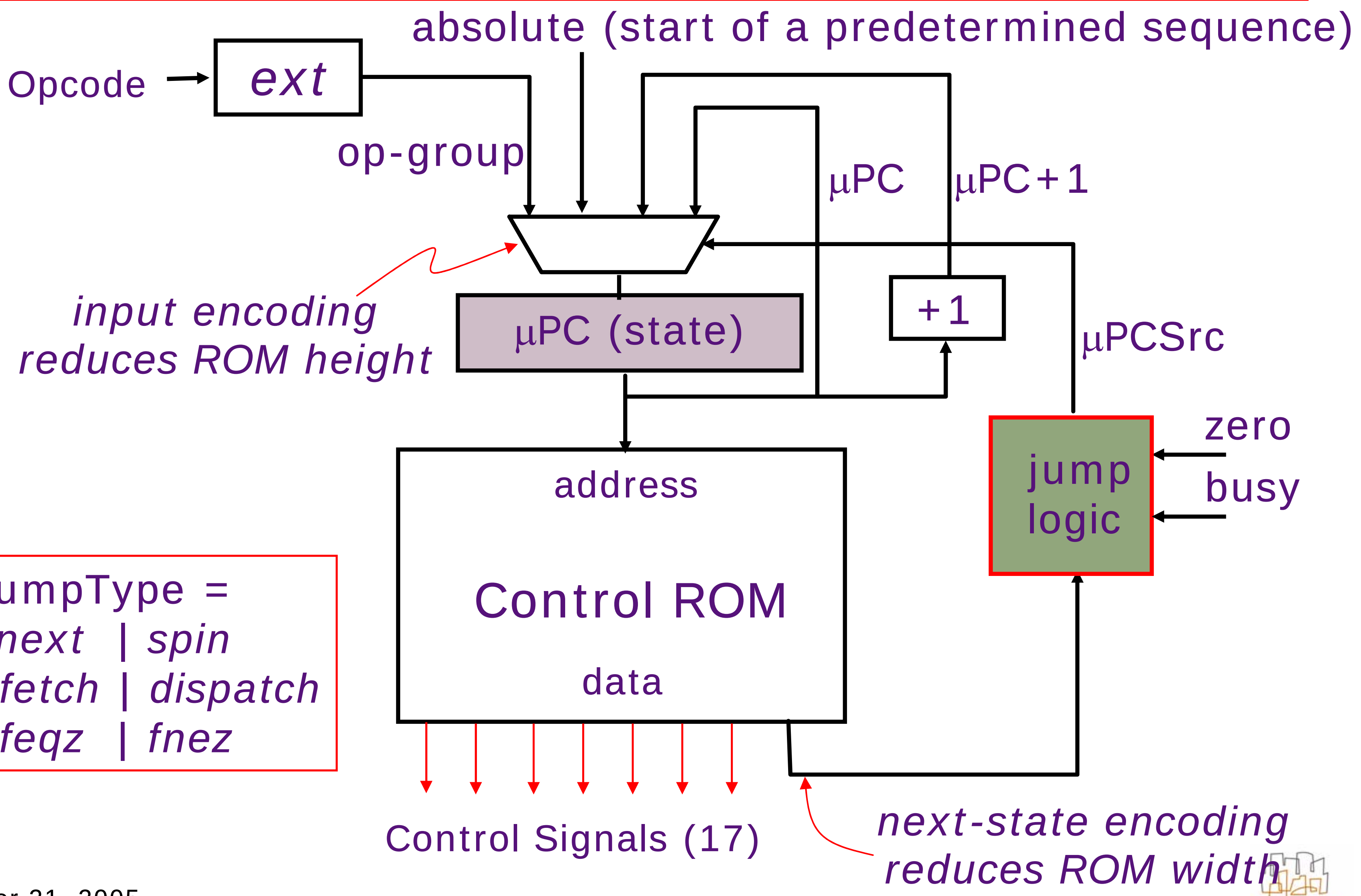
Reducing Control Store Size

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)*

MIPS Controller V2


absolute (start of a predetermined sequence)
Opcode
ext
op-group
μPC
μPC+1
input encoding reduces ROM height
μPC (state)
+1
μPCSrc
zero
busy
jump logic
address
Control ROM
data
Control Signals (17)
next-state encoding reduces ROM width

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

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: *MIPS-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[rs]	next
ALU_1	B ← Reg[rt]	next
ALU_2	Reg[rd]←func(A,B)	fetch
$ALUi_0$	A ← Reg[rs]	next
$ALUi_1$	B ← $sExt_{16}$(Imm)	next
$ALUi_2$	Reg[rd]← Op(A,B)	fetch

Load & Store: *MIPS-Controller-2*

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

Branches: *MIPS-Controller-2*

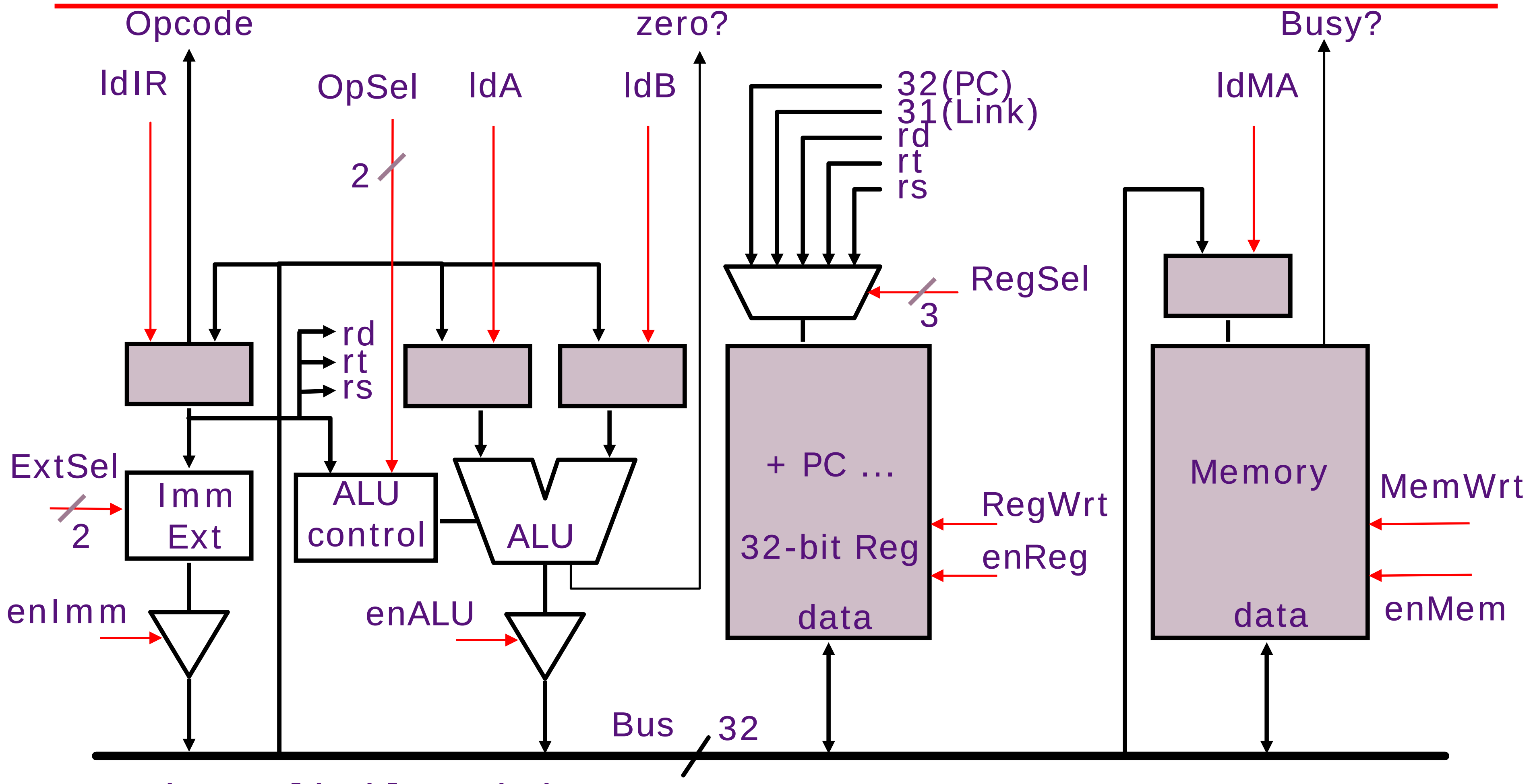
State	Control points	next-state
$BEQZ_0$	$A \leftarrow Reg[rs]$	next
$BEQZ_1$		fnez
$BEQZ_2$	$A \leftarrow PC$	next
$BEQZ_3$	$B \leftarrow sExt_{16}(Imm<<2)$	next
$BEQZ_4$	$PC \leftarrow A+B$	fetch
$BNEZ_0$	$A \leftarrow Reg[rs]$	next
$BNEZ_1$		feqz
$BNEZ_2$	$A \leftarrow PC$	next
$BNEZ_3$	$B \leftarrow sExt_{16}(Imm<<2)$	next
$BNEZ_4$	$PC \leftarrow A+B$	fetch

Jumps: *MIPS-Controller-2*

State	Control points	next-state
J_0	$A \leftarrow PC$	next
J_1	$B \leftarrow IR$	next
J_2	$PC \leftarrow JumpTarg(A,B)$	fetch
JR_0	$A \leftarrow Reg[rs]$	next
JR_1	$PC \leftarrow A$	fetch
JAL_0	$A \leftarrow PC$	next
JAL_1	$Reg[31] \leftarrow A$	next
JAL_2	$B \leftarrow IR$	next
JAL_3	$PC \leftarrow JumpTarg(A,B)$	fetch
$JALR_0$	$A \leftarrow PC$	next
$JALR_1$	$B \leftarrow Reg[rs]$	next
$JALR_2$	$Reg[31] \leftarrow A$	next
$JALR_3$	$PC \leftarrow B$	fetch

Five-minute break to stretch your legs

Implementing Complex Instructions



rd ← M[(rs)] op (rt) *Reg-Memory-src ALU op*

M[(rd)] ← (rs) op (rt) *Reg-Memory-dst ALU op*

M[(rd)] ← M[(rs)] op M[(rt)] *Mem-Mem ALU op*

Mem-Mem ALU Instructions:
MIPS-Controller-2

Mem-Mem ALU op $M[(rd)] \leftarrow M[(rs)]$ op $M[(rt)]$

$ALUMM_0$	$MA \leftarrow Reg[rs]$	next
$ALUMM_1$	$A \leftarrow Memory$	spin
$ALUMM_2$	$MA \leftarrow Reg[rt]$	next
$ALUMM_3$	$B \leftarrow Memory$	spin
$ALUMM_4$	$MA \leftarrow Reg[rd]$	next
$ALUMM_5$	$Memory \leftarrow func(A,B)$	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

Performance Issues

Microprogrammed control

$\Rightarrow$ multiple cycles per instruction

Cycle time ?

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

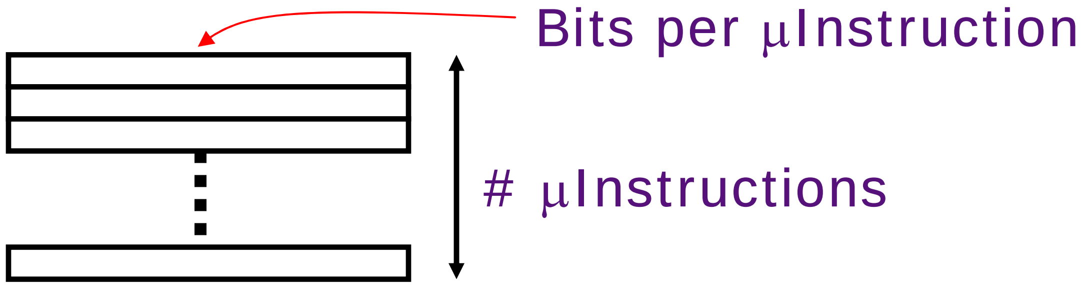
Given complex control, t_{ALU} & t_{RAM} can be broken into multiple cycles. However, $t_{\mu ROM}$ cannot be broken down. Hence

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

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

Horizontal vs Vertical µCode

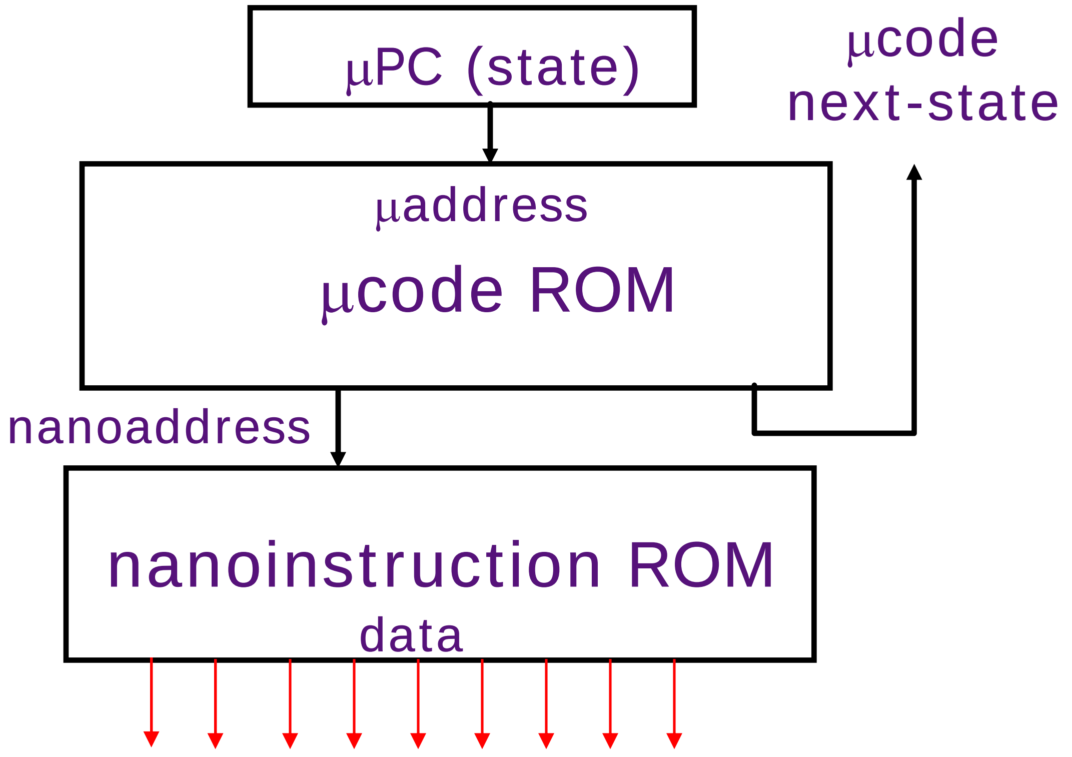


- Horizontal µcode has wider µinstructions
  - Multiple parallel operations per µinstruction
  - Fewer steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical µcode has narrower µinstructions
  - Typically a single datapath operation per µinstruction
    - separate µinstruction for branches
  - 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 $\leftarrow$ Reg[rs]
...
ALUi$_0$ A $\leftarrow$ Reg[rs]
...



- 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

Some more history …

- IBM 360

- Microcoding through the seventies

- Microcoding now

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 minsts)	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 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 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 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 !