

ECE 552 / CPS 550

Advanced Computer Architecture I

Lecture 3

Early Microarchitectures

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall12.html



ECE 552 Administria

11 September – Homework #1 Due

Assignment on web page. Teams of 2-3.

Submit soft copies to Sakai.

Use Piazza for questions.

11 September – Class Discussion

Roughly one reading per class. Do not wait until the day before!

1. Hill et al. "Classic machines: Technology, implementation, and economics"
2. Moore. "Cramming more components onto integrated circuits"
3. Radin. "The 801 minicomputer"
4. Patterson et al. "The case for the reduced instruction set computer"
5. Colwell et al. "Instruction sets and beyond: Computers, complexity, controversy"



Microarchitectures

Instruction Set Architecture (ISA)

- Defines the hardware-software interface
- Provides convenient functionality to higher levels (e.g., software)
- Provides efficient implementation to lower levels (e.g., hardware)

Microarchitecture

- Microarchitecture implements ISA
- Implementation abstracted from programmer

Early Microarchitectures

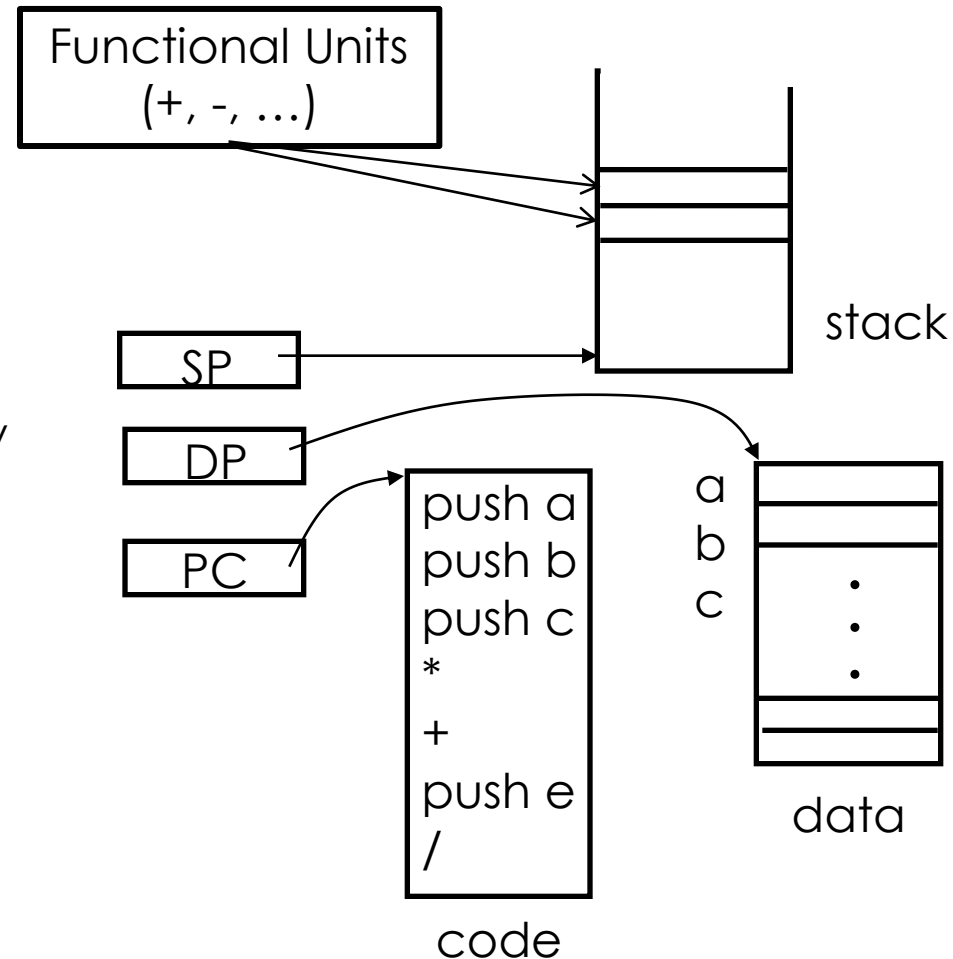
- Stack Machines (1960s)
- Microprogrammed Machines (1970s-1980s)
- Reduced Instruction Set Computing (1990s)



1: Stack Machine Overview

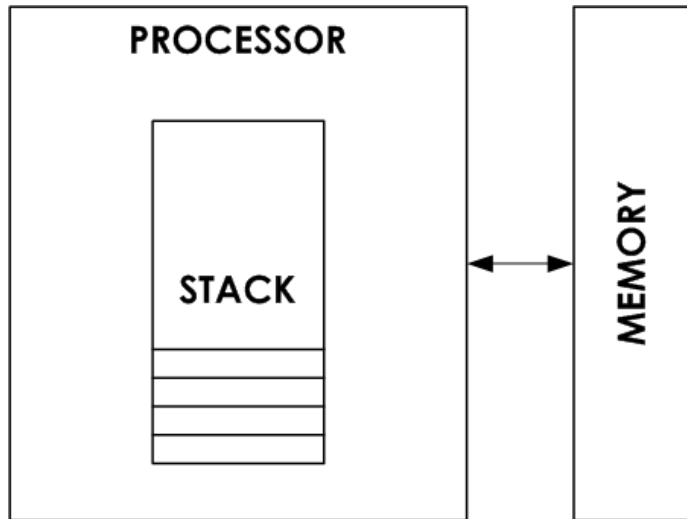
Essential Features

- Stack operations (e.g., push/pop)
- Computation (e.g., +, -, ...)
- Data pointer (DP) for accessing any element in data area
- Program counter (PC) for accessing any instruction in code area
- Stack pointer (SP) to accessing and moving any element in stack frame





Stack Machine Overview

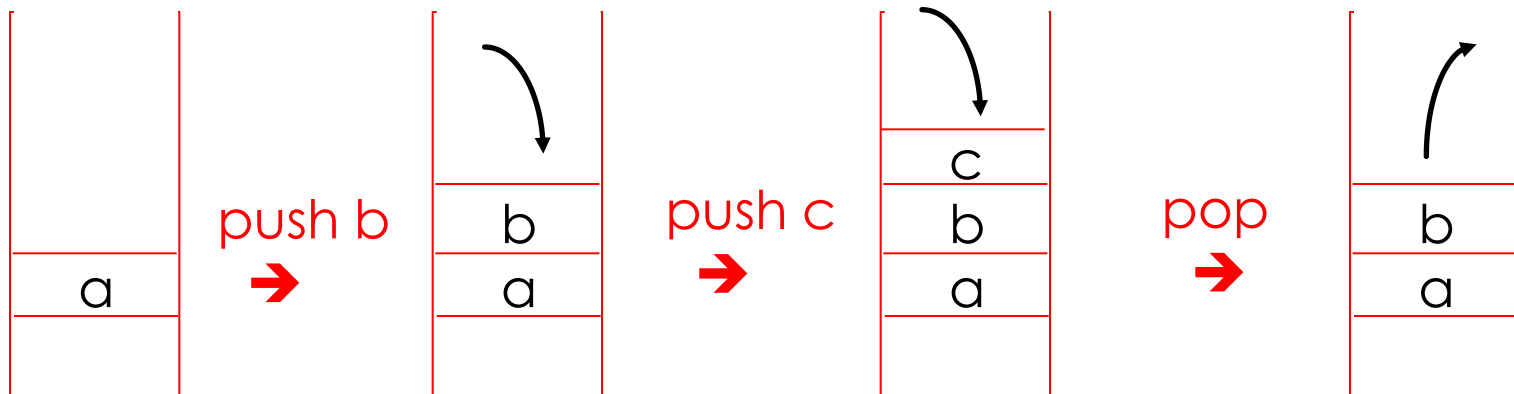


Processor state includes stack

typical operations:

push, pop, +, *, ...

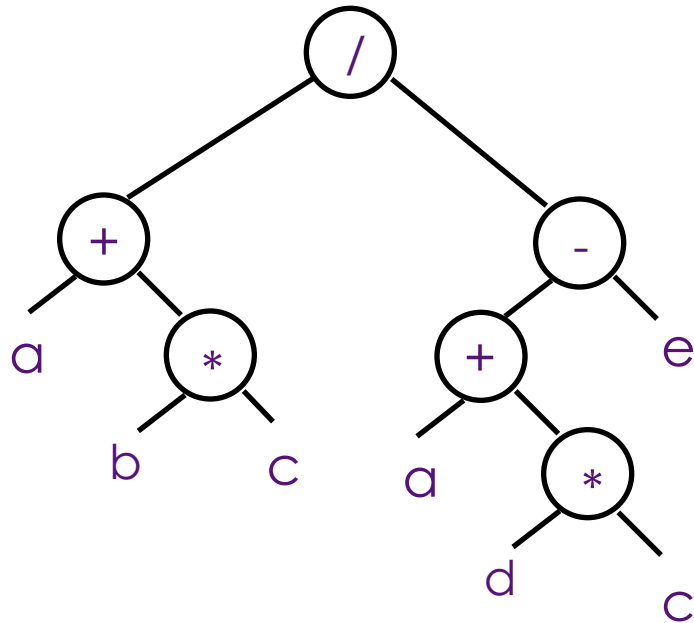
Instructions like + implicitly specify the top 2 elements of the stack as operands.





Expression Evaluation

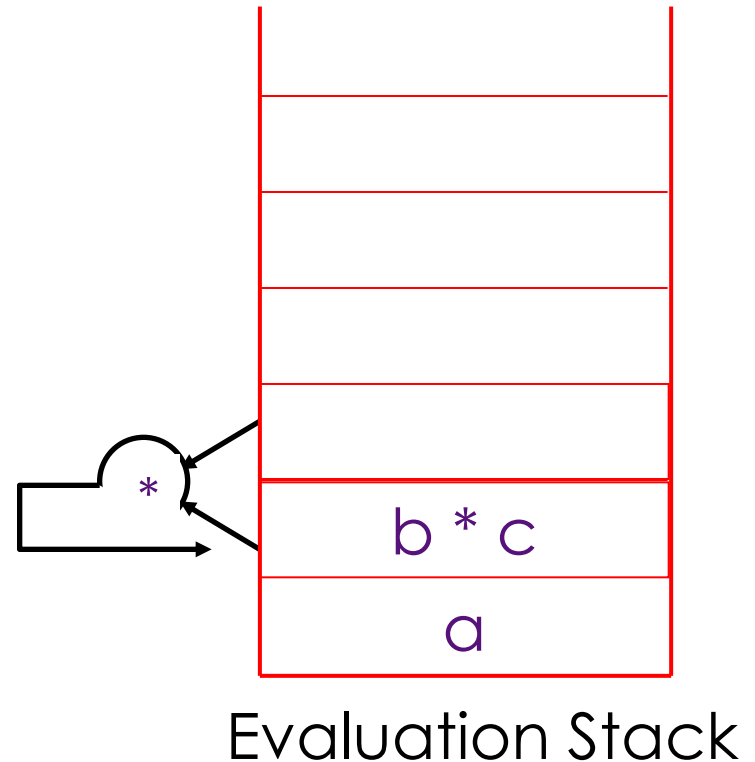
$$(a + b * c) / (a + d * c - e)$$



Reverse Polish

a b c * + a d c * + e - /

↑ ↑ ↑ ↑
push push push multiply





Stack Hardware Organization

Processor State

- Stack is part of processor state
- Stack must be bounded and small
- Max number stack elements ~ register file size, not memory size

Unbounded Stacks

- Part of stack in processor
- Remainder of stack in main memory

Stacks and Memory References

- Option 1: Top 2 stack elements in registers, remainder in memory
- Each push/pop requires memory reference; poor performance
- Option 2: Top N stack elements in registers, remainder in memory
- Overflows/underflows require memory reference; better performance
- Analogous to register spilling



Stack Size & Memory References

$(a+b*c)/(a+d*c-e) \rightarrow a\ b\ c\ * + a\ d\ c\ * + e\ - /$

	<i>program stack (size = 2)</i>	<i>memory refs</i>
push a	R0	a
push b	R0 R1	b
push c	R0 R1 R2	c, ss(a)
*	R0 R1	sf(a)
+	R0	
push a	R0 R1	a
push d	R0 R1 R2	d, ss(a+b*c)
push c	R0 R1 R2 R3	c, ss(a)
*	R0 R1 R2	sf(a)
+	R0 R1	sf(a+b*c)
push e	R0 R1 R2	e,ss(a+b*c)
-	R0 R1	sf(a+b*c)
/	R0	

4 stores, 4 fetches (implicit)



Stack Size & Evaluating Expressions

$(a+b*c)/(a+d*c-e) \rightarrow a\ b\ c\ *\ +\ a\ d\ c\ *\ +\ e\ -\ /\$

push a
push b
push c
*
+
push a
push d
push c
*
+
push e
-
/

program stack (size = 4)

R0
R0 R1
R0 R1 R2
R0 R1
R0
R0 R1
R0 R1 R2
R0 R1 R2 R3
R0 R1 R2
R0 R1
R0 R1 R2
R0 R1
R0

a and c are loaded twice, inefficient register use



vs General Purpose Register File

$$(a+b*c)/(a+d*c-e) \rightarrow a \ b \ c \ * \ + \ a \ d \ c \ * \ + \ e \ - \ /$$

Load	R0	a	
Load	R1	c	
Load	R2	b	
Mul	R2	R1	Reuse R2, Store result into R2
Add	R2	R0	Reuse R2, ...
Load	R3	d	
Mul	R3	R1	Reuse R3, Store result into R3
Add	R3	R0	Reuse R3, ...
Load	R0	e	Reuse R0, ...
Sub	R3	R0	Reuse R3, ...
Div	R2	R3	Reuse R2, ...

Efficient Register Usage

- Control register use with explicitly named registers
- Eliminate unnecessary loads and stores
- Fewer registers required but instructions are longer



Stack vs General Purpose Registers

- Amdahl, Blaauw, and Brooks. "Architecture of the IBM System/360." 1964

"In the final analysis, the stack organization would have been about break-even...the general-purpose objective weighed heavily in favor of the more flexible addressed register organization."

1. Stack machine's advantage is from fast registers, not how they're used
2. Surfacing instructions, which bring submerged data to active positions, is profitable 50% of the time due to repeated operands.
3. Code density for stacks, register files is comparable
4. Stack depth limited by number of fast registers; requires stack management
5. Stacks benefit recursive sub-routines but requires independently addressed stacks (SP management)
6. Fitting variable-length fields into fixed-width stack is awkward



Transition from Stack Machines

Stack machine's popularity faded in the 1980s

Code Density

- Stack programs are not necessary smaller
- Consider frequent stack surfacing instructions
- Consider short register addresses

Advent of Modern Compilers

- Stack machines require stack discipline to manage finite stack
- Register allocation improves register space management
- Early language-directed architectures did not account for compilers
- B5000, B6700, HP3000, ICL 2900, Symbolics 3600



2: Microprogrammed Machines

Why do we care?

- Provide background on CISC machines
- Illustrate small processors with complex instruction sets
- Used in most modern machines (x86, PowerPC, IBM360)
- Introduce machine structures
- Provide context for RISC machines

ISA favors particular microarchitecture style

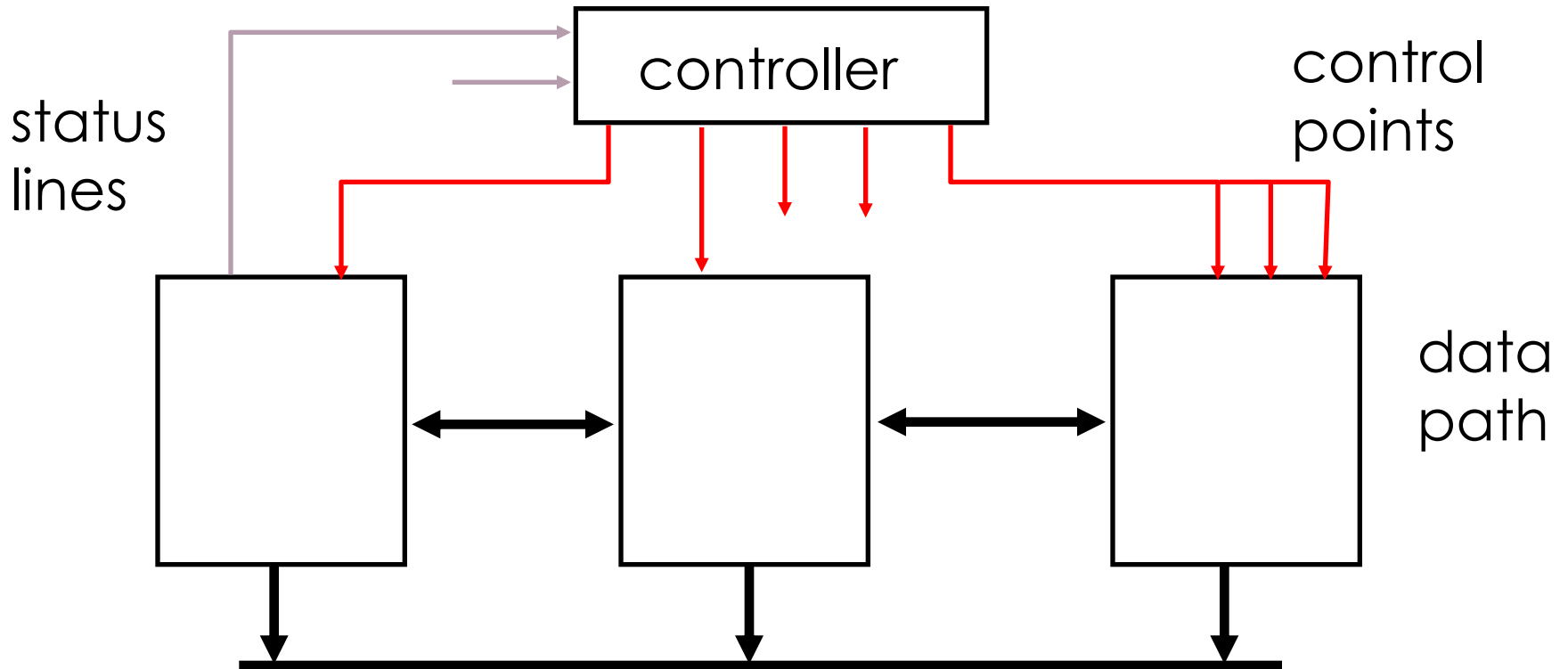
- CISC: microprogrammed
- RISC: hardwired, pipelined
- VLIW: fixed latency, in-order pipelines
- JVM: software interpretation

ISA can use any microarchitecture style

- Core2 Duo: hardwired, pipelined CISC (x86) w/ microcode support
- This Lecture: Microcoded RISC (MIPS) machine



Hardware Organization

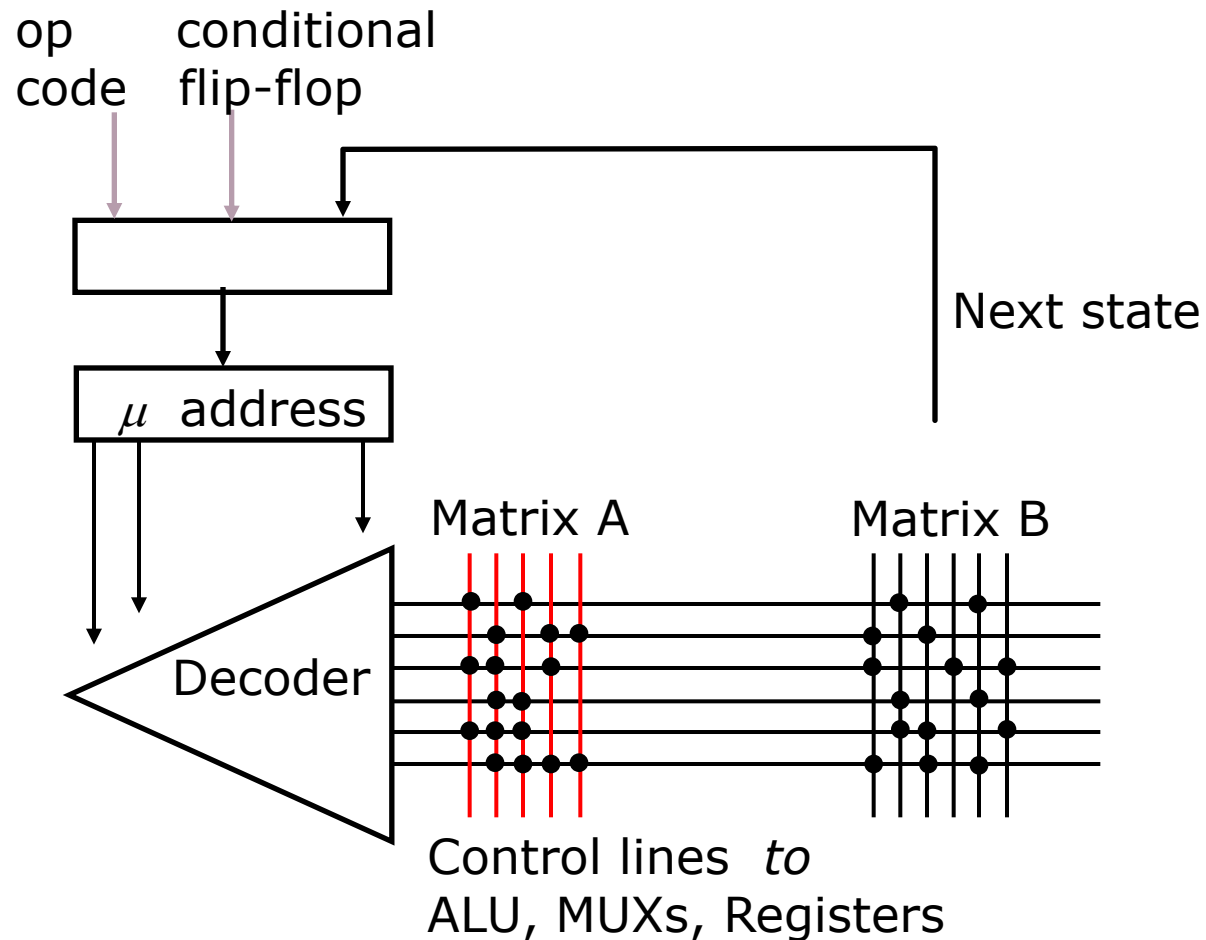


- Structure: How are components connected? Statically
- Behavior: How does data move between components? Dynamically



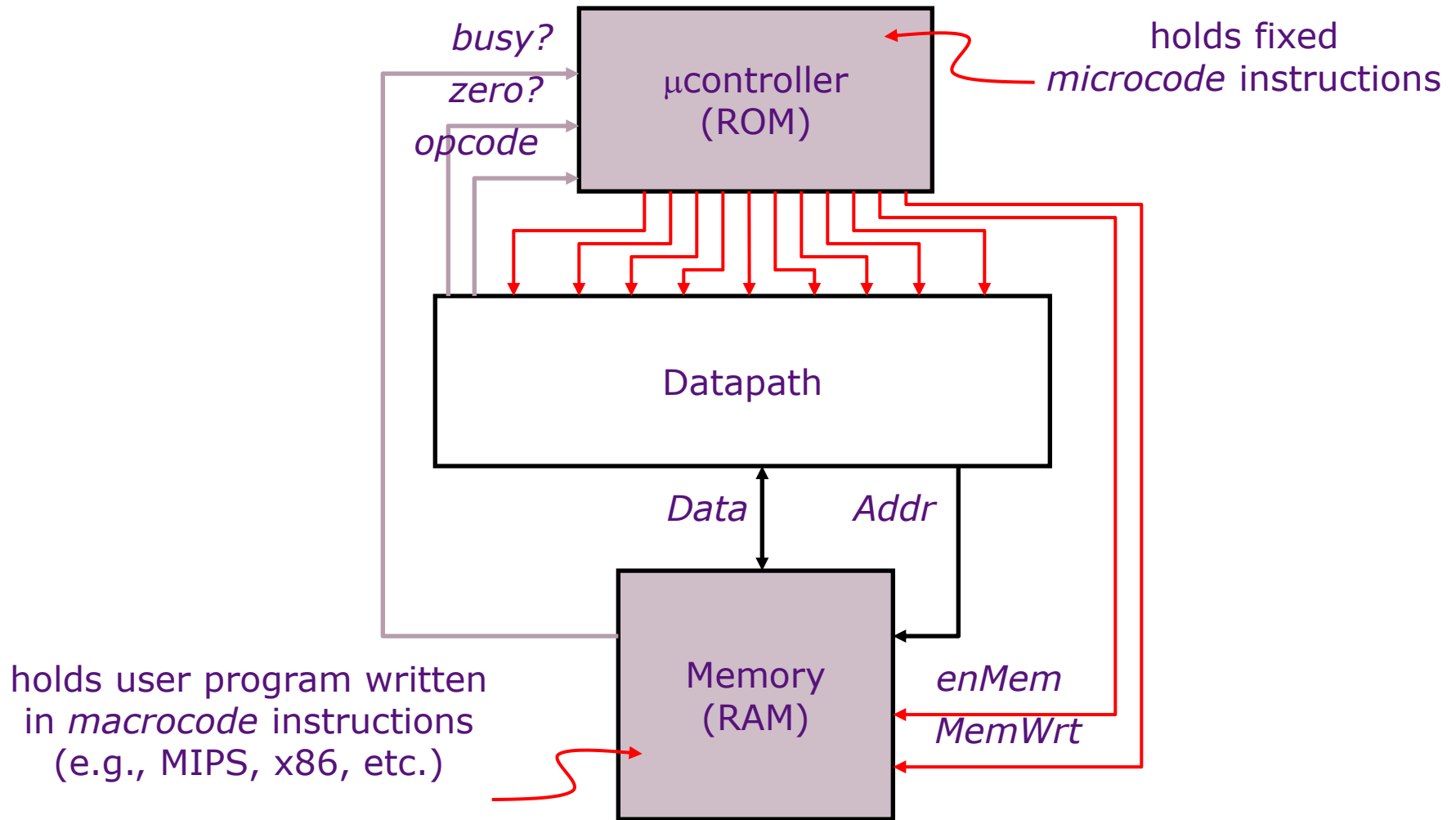
Microcontrol Unit

- Maurice Wilkes, 1954
- Embed control logic state table in a memory array





Microcoded Microarchitecture





MIPS32 ISA

- See Hennessy and Patterson, Appendix for full description

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
- 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

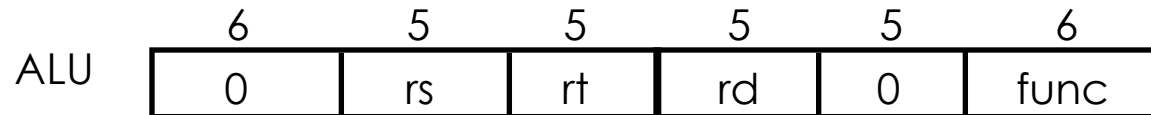
Load/Store style instruction set

- data addressing modes: immediate & indexed
- branch addressing modes: PC relative & register indirect
- byte addressable memory: big-endian mode

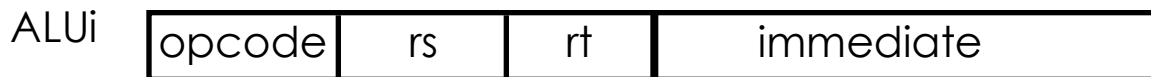
All instructions are 32 bits



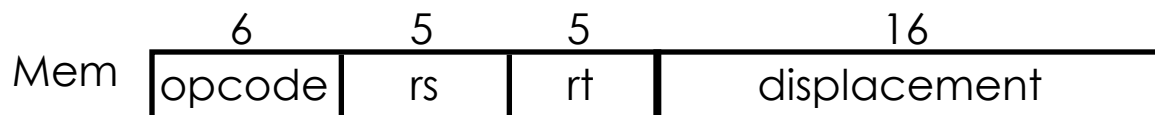
MIPS Instruction Formats



$(rd) \leftarrow (rs) \text{ func } (rt)$



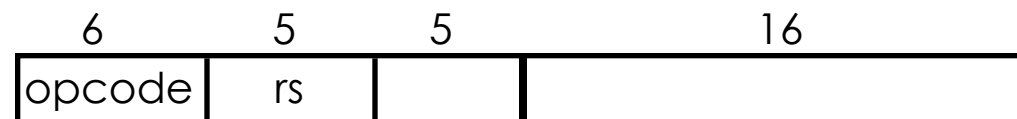
$(rt) \leftarrow (rs) \text{ op } \text{immediate}$



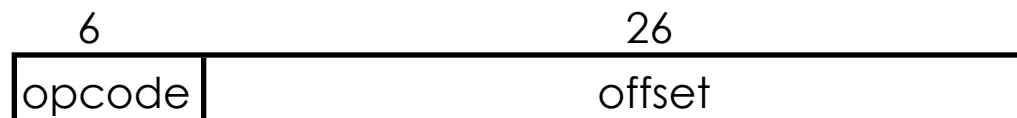
$M[(rs) + \text{displacement}]$



BEQZ, BNEZ



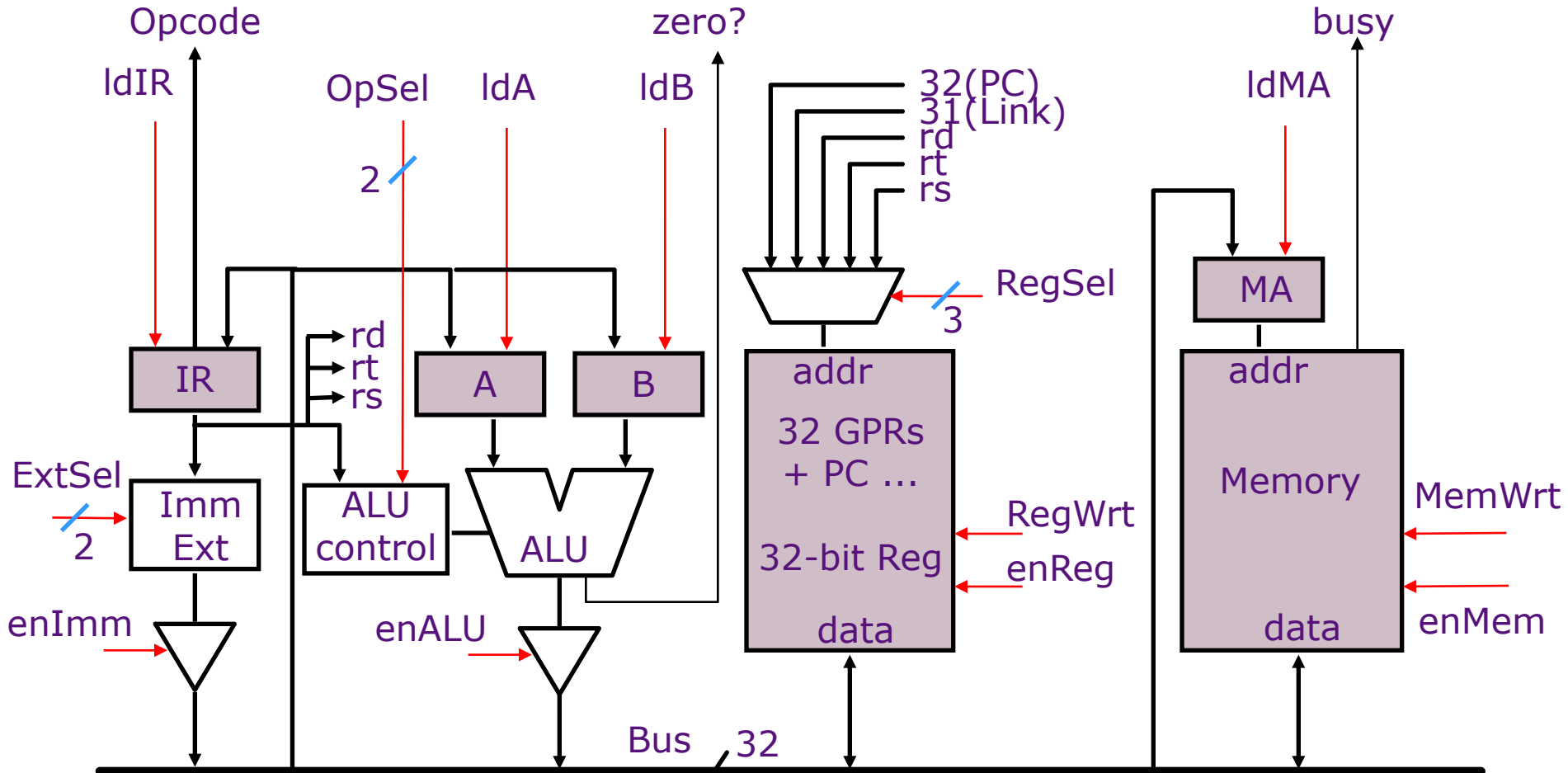
JR, JALR



J, JAL



A Bus-Based MIPS Datapath



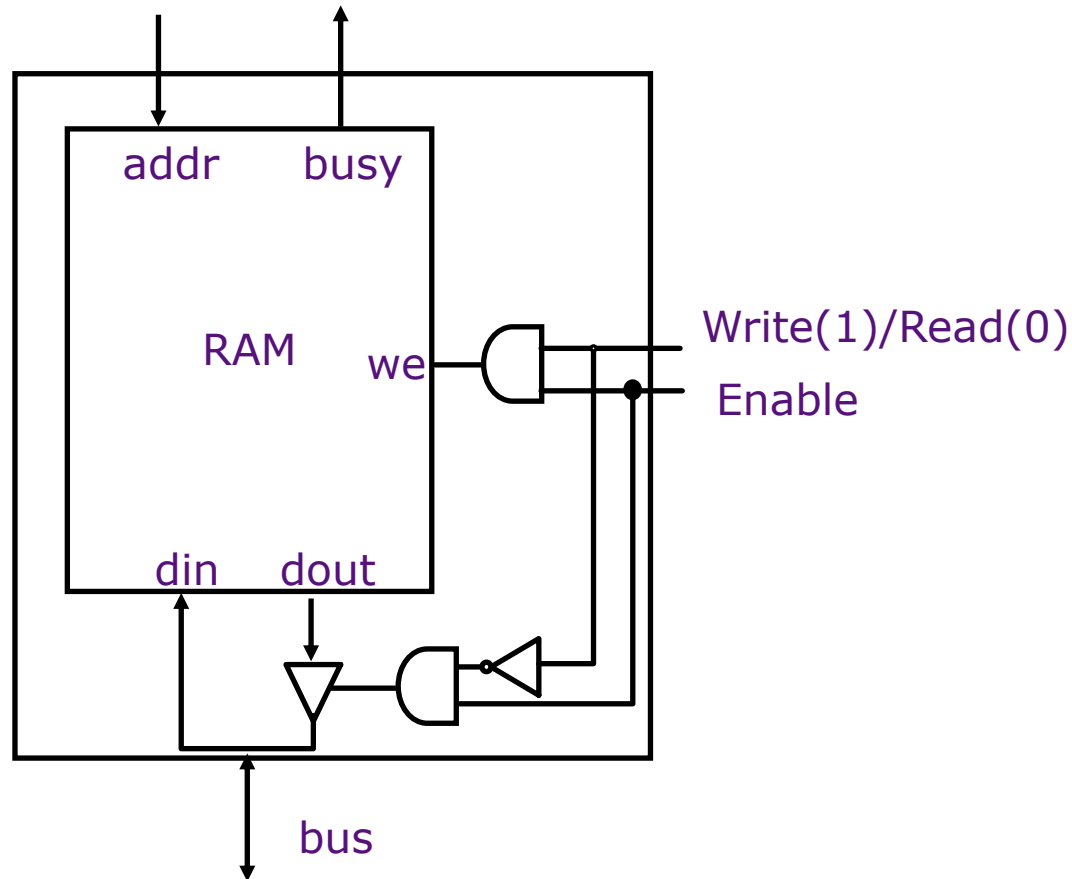
Microinstruction: register to register transfer (17 control signals)

$MA \leftarrow PC$ means $RegSel = PC$; $enReg = yes$; $IdMA = yes$

$B \leftarrow Reg[rt]$ means $RegSel = rt$; $enReg = yes$; $IdB = yes$



Memory Module



Assumption: Memory operates independently and is slow as compared to Reg-to-Reg transfers (multiple CPU clock cycles per access)



Instruction Execution

1. Instruction fetch
2. Decode and register fetch
3. ALU operation
4. Memory operation (optional)
5. Write back to register file (optional)
+ computation of the next instruction address



```

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
```

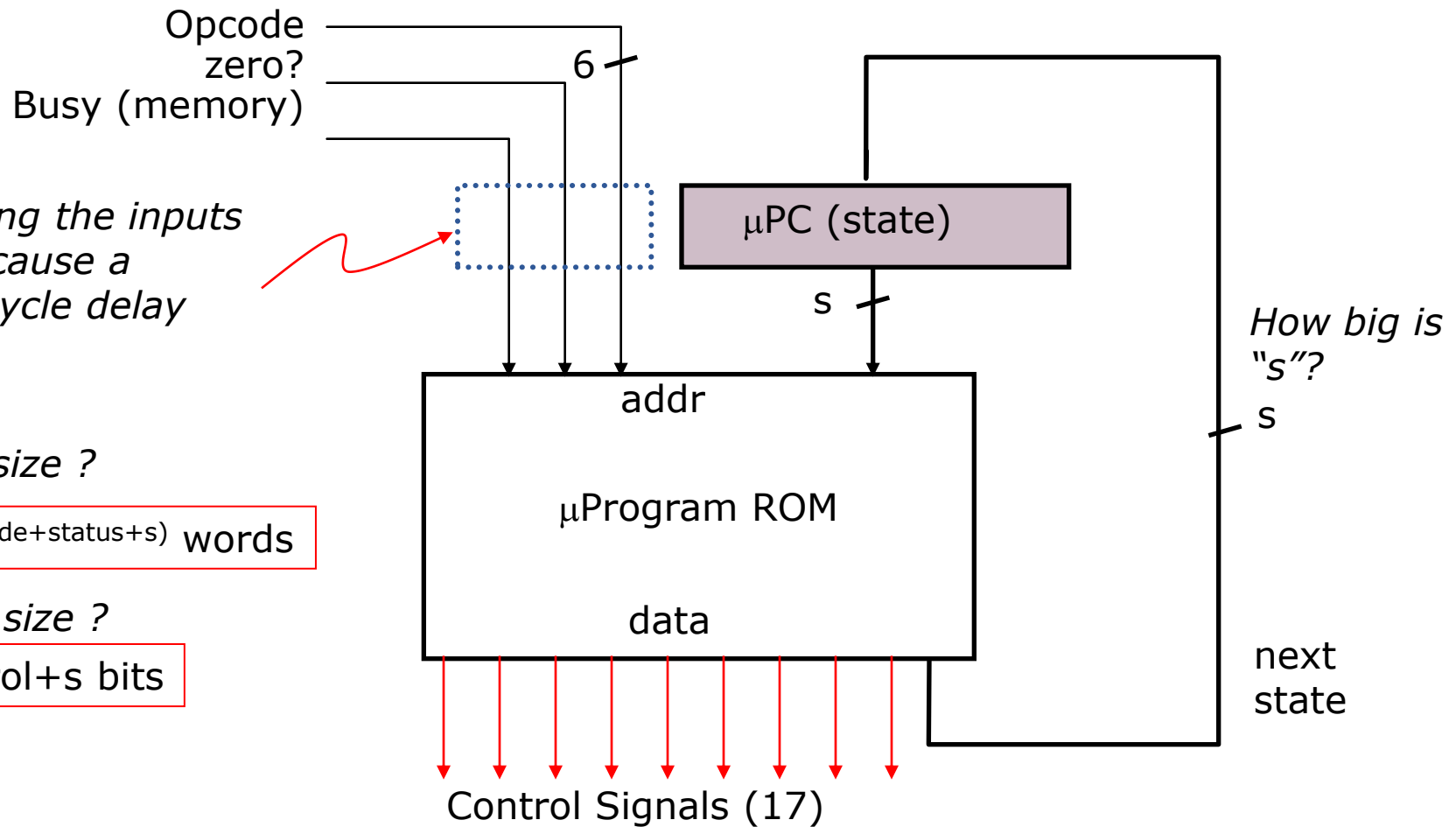


Microprogram Fragments

LW:	$A \leftarrow \text{Reg}[\text{rs}]$ $B \leftarrow \text{Imm}$ $\text{MA} \leftarrow A + B$ $\text{Reg}[\text{rt}] \leftarrow \text{Memory}$ do instruction fetch	# compute address # load from memory
J:	$A \leftarrow \text{PC}$ $B \leftarrow \text{IR}$ $\text{PC} \leftarrow \text{JumpTarg}(A,B)$ do instruction fetch	# $\text{JumpTarg}(A,B) = \{A[31:28], B[25:0], 00\}$
beqz:	$A \leftarrow \text{Reg}[\text{rs}]$ <i>If zero?(A) then go to bz-taken</i> do instruction fetch	
bz-taken:	$A \leftarrow \text{PC}$ $B \leftarrow \text{Imm} \ll 2$ $\text{PC} \leftarrow A + B$ do instruction fetch	



MIPS Microcontroller



ROM size ?

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

Word size ?

= control + s bits



Microprogram in the ROM

State	Op	zero?	busy	Control points	next-state
fetch ₀	*	*	*	MA $\leftarrow$ PC	fetch ₁
fetch ₁	*	*	yes		fetch ₁
fetch ₁	*	*	no	IR $\leftarrow$ Memory	fetch ₂
fetch ₂	*	*	*	A $\leftarrow$ PC	fetch ₃
fetch ₃	*	*	*	PC $\leftarrow$ A + 4	?
fetch ₃	ALU	*	*	PC $\leftarrow$ A + 4	ALU ₀
ALU ₀	*	*	*	A $\leftarrow$ Reg[rs]	ALU ₁
ALU ₁	*	*	*	B $\leftarrow$ Reg[rt]	ALU ₂
ALU ₂	*	*	*	Reg[rd] $\leftarrow$ func(A,B)	fetch ₀



Microprogram in the ROM

State	Op	zero?	busy	Control points	next-state
fetch ₀	*	*	*	MA $\leftarrow$ PC	fetch ₁
fetch ₁	*	*	yes		fetch ₁
fetch ₁	*	*	no	IR $\leftarrow$ Memory	fetch ₂
fetch ₂	*	*	*	A $\leftarrow$ PC	fetch ₃
fetch ₃	ALU	*	*	PC $\leftarrow$ A + 4	ALU ₀
fetch ₃	ALUi	*	*	PC $\leftarrow$ A + 4	ALUi ₀
fetch ₃	LW	*	*	PC $\leftarrow$ A + 4	LW ₀
fetch ₃	SW	*	*	PC $\leftarrow$ A + 4	SW ₀
fetch ₃	J	*	*	PC $\leftarrow$ A + 4	J ₀
fetch ₃	JAL	*	*	PC $\leftarrow$ A + 4	JAL ₀
fetch ₃	JR	*	*	PC $\leftarrow$ A + 4	JR ₀
fetch ₃	JALR	*	*	PC $\leftarrow$ A + 4	JALR ₀
fetch ₃	beqz	*	*	PC $\leftarrow$ A + 4	beqz ₀
...					
ALU ₀	*	*	*	A $\leftarrow$ Reg[rs]	ALU ₁
ALU ₁	*	*	*	B $\leftarrow$ Reg[rt]	ALU ₂
ALU ₂	*	*	*	Reg[rd] $\leftarrow$ func(A,B)	fetch ₀



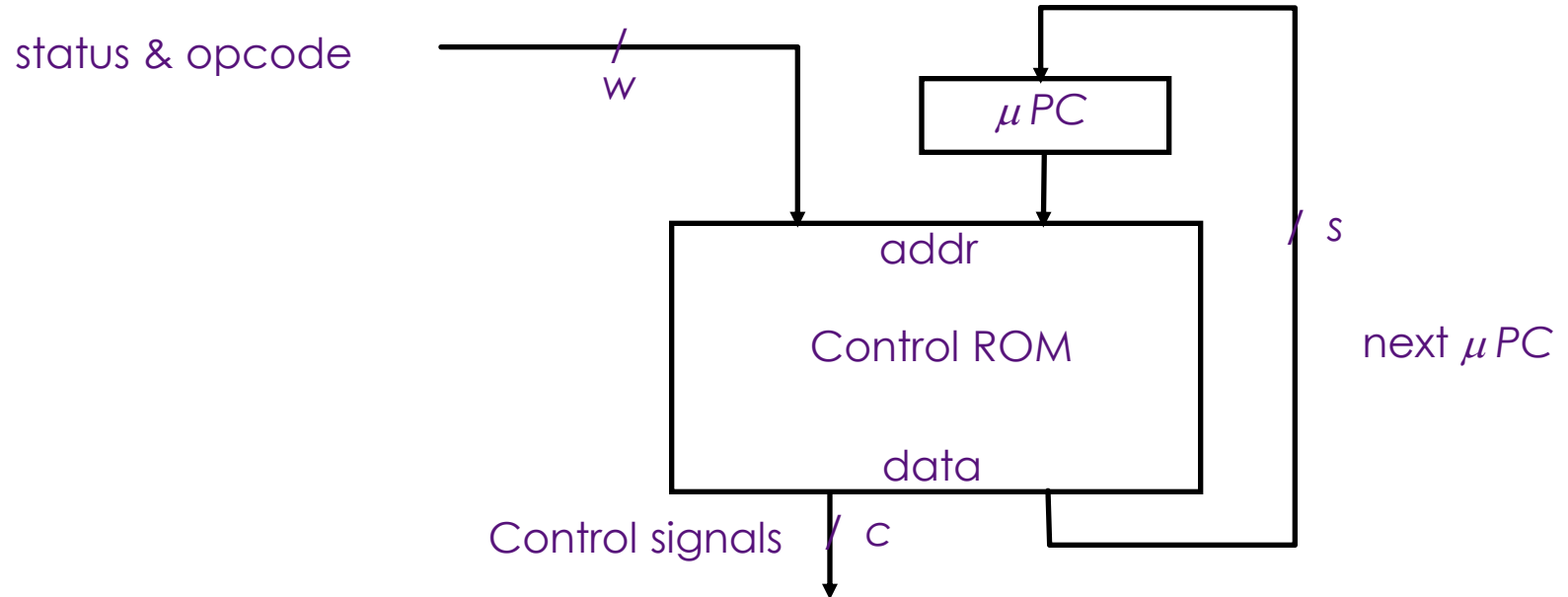
Microprogram in the ROM

State	Op	zero?	busy	Control points	next-state
ALUi ₀	*	*	*	$A \leftarrow \text{Reg}[\text{rs}]$	ALUi ₁
ALUi ₁	sExt	*	*	$B \leftarrow \text{sExt}_{16}(\text{Imm})$	ALUi ₂
ALUi ₁	uExt	*	*	$B \leftarrow \text{uExt}_{16}(\text{Imm})$	ALUi ₂
ALUi ₂	*	*	*	$\text{Reg}[\text{rd}] \leftarrow \text{Op}(A, B)$	fetch ₀
...					
J ₀	*	*	*	$A \leftarrow \text{PC}$	J ₁
J ₁	*	*	*	$B \leftarrow \text{IR}$	J ₂
J ₂	*	*	*	$\text{PC} \leftarrow \text{JumpTarg}(A, B)$	fetch ₀
...					
beqz ₀	*	*	*	$A \leftarrow \text{Reg}[\text{rs}]$	beqz ₁
beqz ₁	*	yes	*	$A \leftarrow \text{PC}$	beqz ₂
beqz ₁	*	no	*		fetch ₀
beqz ₂	*	*	*	$B \leftarrow \text{sExt}_{16}(\text{Imm})$	beqz ₃
beqz ₃	*	*	*	$\text{PC} \leftarrow A + B$	fetch ₀
...					

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



Size of Control Store



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

$$w = 6+2, c = 17, s = ?$$

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

$$\begin{aligned} \text{no. of states} &= (4 \text{ steps per op-group}) \times \text{op-groups} + \text{common sequences} \\ &= 4 \times 8 + 10 \text{ states} = 42 \text{ states} \rightarrow s = 6 \end{aligned}$$

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



Reducing Size of Control Store

Control store must be fast, which is expensive

Reduce ROM height (= address bits)

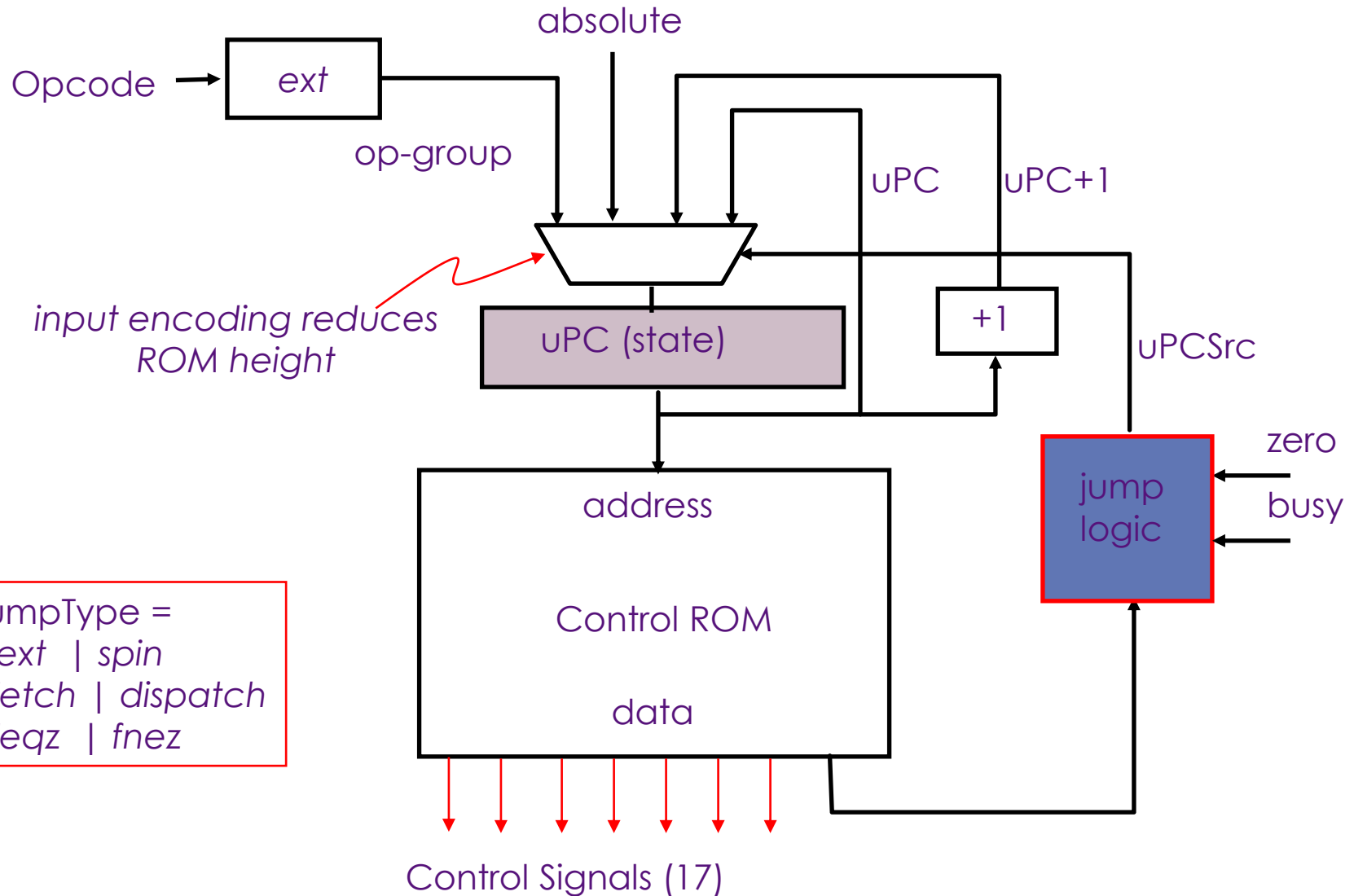
- reduce inputs by extra external logic
- each input bit doubles the size of the control store
- reduce states by grouping opcodes
- find common sequences of actions
- condense input status bits
- combine all exceptions into one, i.e., exception/no-exception

Reduce ROM width

- restrict the next-state encoding-
- next, dispatch on opcode, wait for memory, ...
- encode control signals (vertical microcode)



MIPS Microcontroller v2





Jump Logic

$\mu\text{PCSrc} = \text{Case } \mu\text{JumpTypes}$

next $\rightarrow \mu\text{PC}+1$

spin $\rightarrow$ if (busy) then μPC 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 and ALU

State	Control Points	Next State
fetch0	$MA \leftarrow PC$	next
fetch1	$IR \leftarrow \text{Memory}$	spin
fetch2	$A \leftarrow PC$	next
fetch3	$PC \leftarrow A+4$	dispatch
...		
ALU0	$A \leftarrow \text{Reg}[\text{rs}]$	next
ALU1	$B \leftarrow \text{Reg}[\text{rt}]$	next
ALU2	$\text{Reg}[\text{rd}] \leftarrow \text{func}(A,B)$	fetch
...		
ALUi0	$A \leftarrow \text{Reg}[\text{rs}]$	next
ALUi1	$B \leftarrow \text{sExt16}(\text{Imm})$	next
ALUi2	$\text{Reg}[\text{rd}] \leftarrow \text{Op}(A,B)$	fetch



Load and Store

State	Control Points	Next State
LW0	$A \leftarrow \text{Reg}[\text{rs}]$	next
LW1	$B \leftarrow \text{sExt16}(\text{Imm})$	next
LW2	$MA \leftarrow A+B$	next
LW3	$\text{Reg}[\text{rt}] \leftarrow \text{Memory}$	spin
LW4		fetch
...		
SW0	$A \leftarrow \text{Reg}[\text{rs}]$	next
SW1	$B \leftarrow \text{sExt16}(\text{Imm})$	next
SW2	$MA \leftarrow A+B$	next
SW3	$\text{Memory} \leftarrow \text{Reg}[\text{rt}]$	spin
SW4		fetch

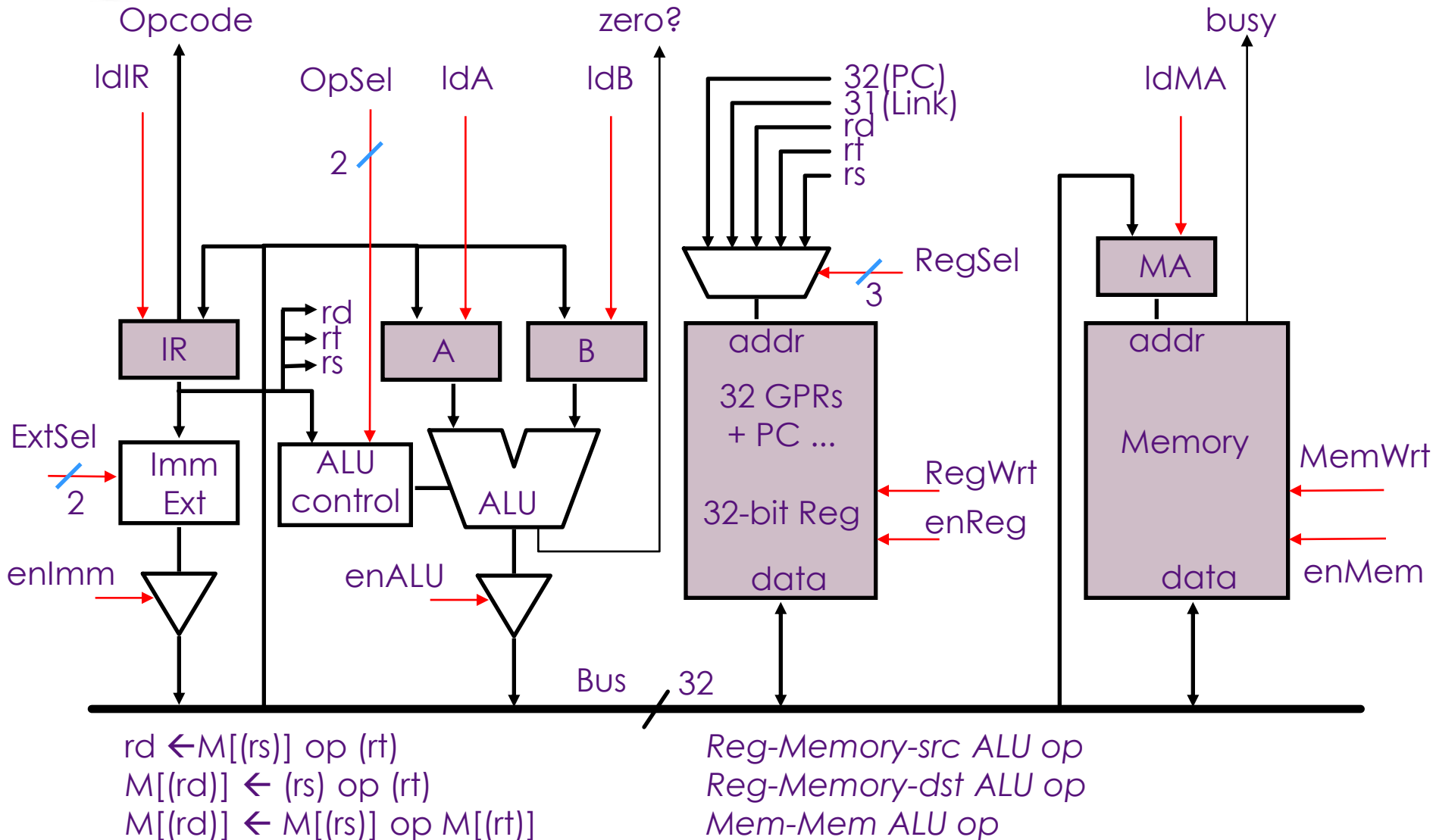


Jumps

State	Control Points	Next State
J0	$A \leftarrow PC$	next
J1	$B \leftarrow IR$	next
J2	$PC \leftarrow \text{JumpTarg}(A,B)$	fetch
JR0	$A \leftarrow \text{Reg}[rs]$	next
JR1	$PC \leftarrow A$	fetch
JAL0	$A \leftarrow PC$	next
JAL1	$\text{Reg}[31] \leftarrow A$	next
JAL2	$B \leftarrow IR$	next
JAL3	$PC \leftarrow \text{JumpTarg}(A,B)$	fetch
JALR0	$A \leftarrow PC$	next
JALR1	$B \leftarrow \text{Reg}[rs]$	next
JALR2	$\text{Reg}[31] \leftarrow A$	next
JALR3	$PC \leftarrow B$	fetch



Complex Instructions





Mem-Mem ALU Instruction

$$M[(rd)] \leftarrow M[(rs)] \text{ op } M[(rt)]$$

State	Control Points	Next State
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 instructions without a hardwired controller is difficult without datapath modifications



Performance

Microcode requires multiple cycles per instruction

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

Microcode requires multiple cycles per instruction

- Suppose $10 * t_{\mu\text{ROM}} < t_{\text{RAM}}$
- Compare against single-cycle hardwired implementation
- Good performance can be achieved even with CPI of 10

Microprogramming Advantages (1970's)

- ROMs significantly faster than DRAMs
- For complex instruction sets, datapath and controller were simpler
- Adding instructions (e.g., floating-point) without datapath changes
- Fixing bugs in the controller is easy
- ISA compatibility across models is easy



Decline of Microprogramming

Increasingly complex microcode

- Complex instruction sets led to subroutine, call stacks in microcode
- Fixing bugs difficult with read-only nature of ROMs

Evolving Technology

- VLSI changed assumptions of ROMs vs RAMs, which became faster
- Microarchitectural innovations (pipelining, caches, buffers) make multi-cycle register-to-register execution unattractive

Evolving Software

- Better compilers made complex instructions less important
- Most complex instructions are rarely used

Transition to RISC

- Build fast instruction cache
- Use software subroutines, not hardware subroutines
- Use simple ISA to enable hardwired implementations



Modern Microprogramming

Microprogramming is far from extinct

- Played crucial role in microprocessors of 1980's (e.g., Intel 386, 486)
- Plays assisting role in modern microprocessors

Important role in early microprocessors (1980s)

- Example: Intel 386, 486

Assisting role in modern microprocessors

- Example: AMD Athlon, Intel Core 2 Duo, IBM Power PC
- Most instructions executed directly (hardwired control)
- Infrequently-used, complicated instructions invoke microcode engine

Assisting role in modern microprocessors

- Patchable microcode common for post-fabrication bug fixes
- Example: Intel Pentiums load microcode patches at startup



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)
- Alvin Lebeck (Duke)
- David Patterson (UCB)
- Daniel Sorin (Duke)