ECE 252 / CPS 220
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_ece252fall11.html

ECE 252 Administria

8 September – Homework #1 Due

Assignment on web page. Teams of 2-3.

Submit hard copy in class. Email code to TA's

13 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)

Stack Machine Overview

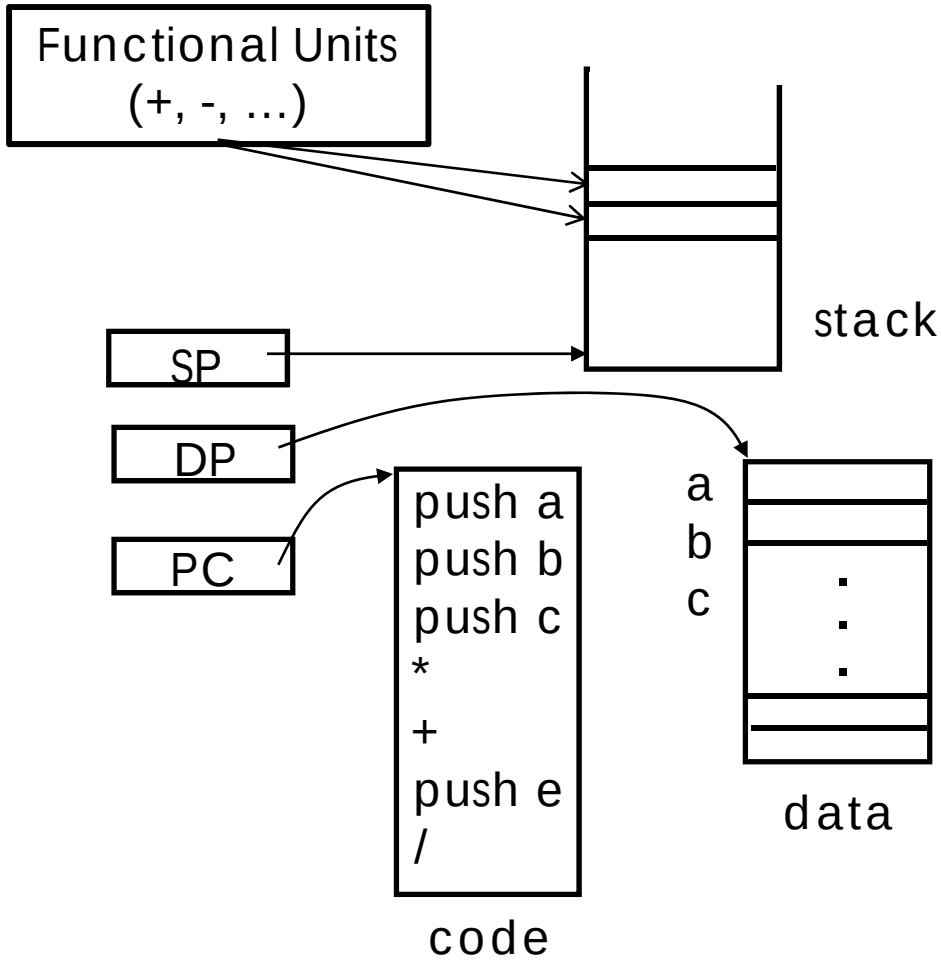
Essential Features

Stack operations (e.g., push/pop)
Computation (e.g., +, -, ...)

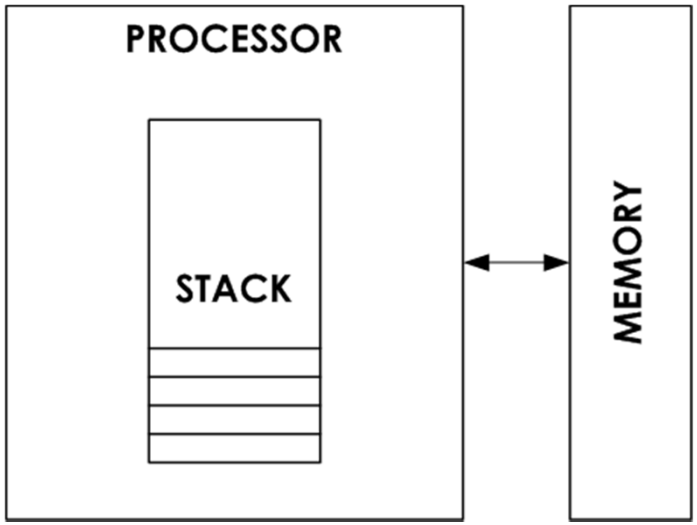
Data pointer (DP) for accessing any element in data area

Program counter (PC) for accessing any instruction in code area

Stack pointer (SP) to accessing and moving any element in stack frame



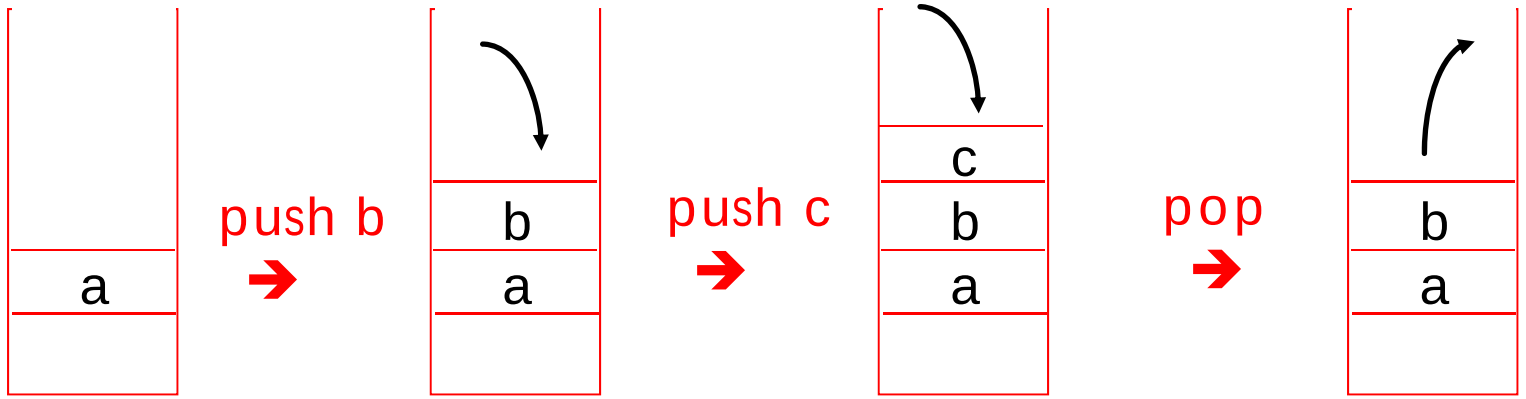
Stack Machine Overview



Processor state includes state

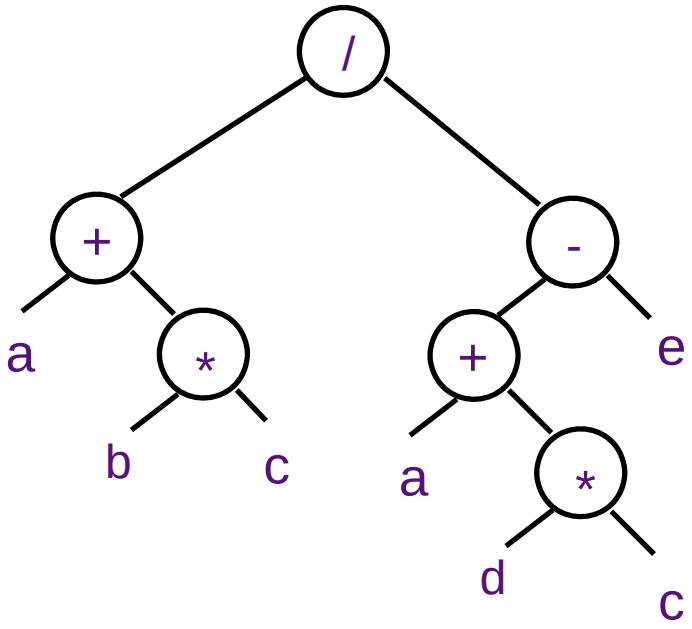
typical operations:
push, pop, +, *, ...

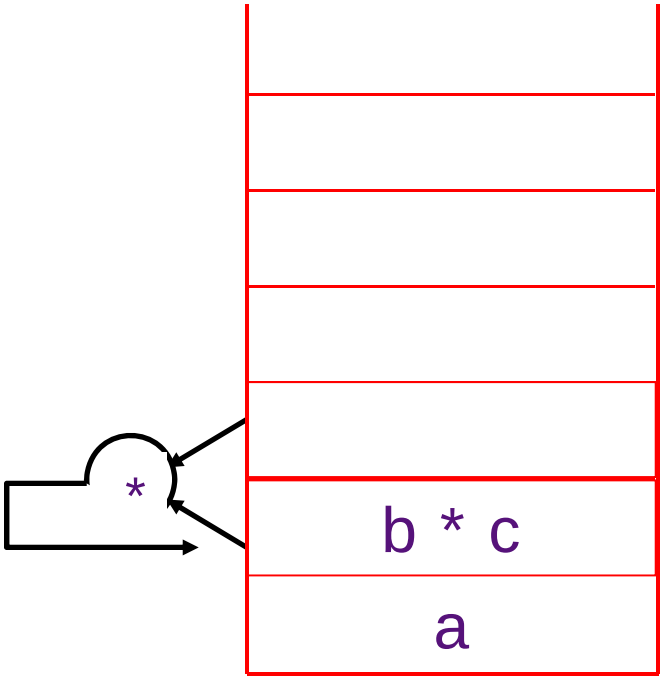
Instructions like + implicitly specify the top 2 elements of the stack as operands.



Expression Evaluation

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





Evaluation Stack

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

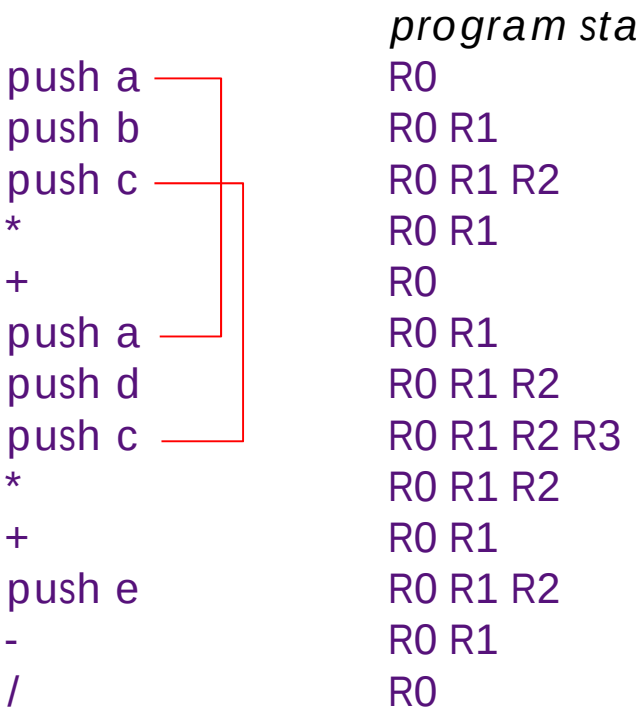
(a+b*c)/(a+d*c-e) ➔ a b c * + a d c * + e - /

	program stack (size = 2)	*memory refs*
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) ➔ a b c * + a d c * + e - /



a and c are loaded twice, inefficient register use

vs General Purpose Register File

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

```
Load    R0    a
Load    R1    c
Load    R2    b
Mul     R2    R1    Reuse R2

Add     R2    R0
Load    R3    d
Mul     R3    R1    Reuse R3
Add     R3    R0

Load    R0    e     Reuse R0
Sub     R3    R0
Div     R2    R3
```

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 Register

- Amdahl, Blaauw, and Brooks. "Architecture of the IBM System/360." 1964
- Stacks: access fast registers with stack operations (e.g., push, pop)
- General purpose registers: access fast registers with explicit addresses (e.g., R1)

"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 and recursion 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

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

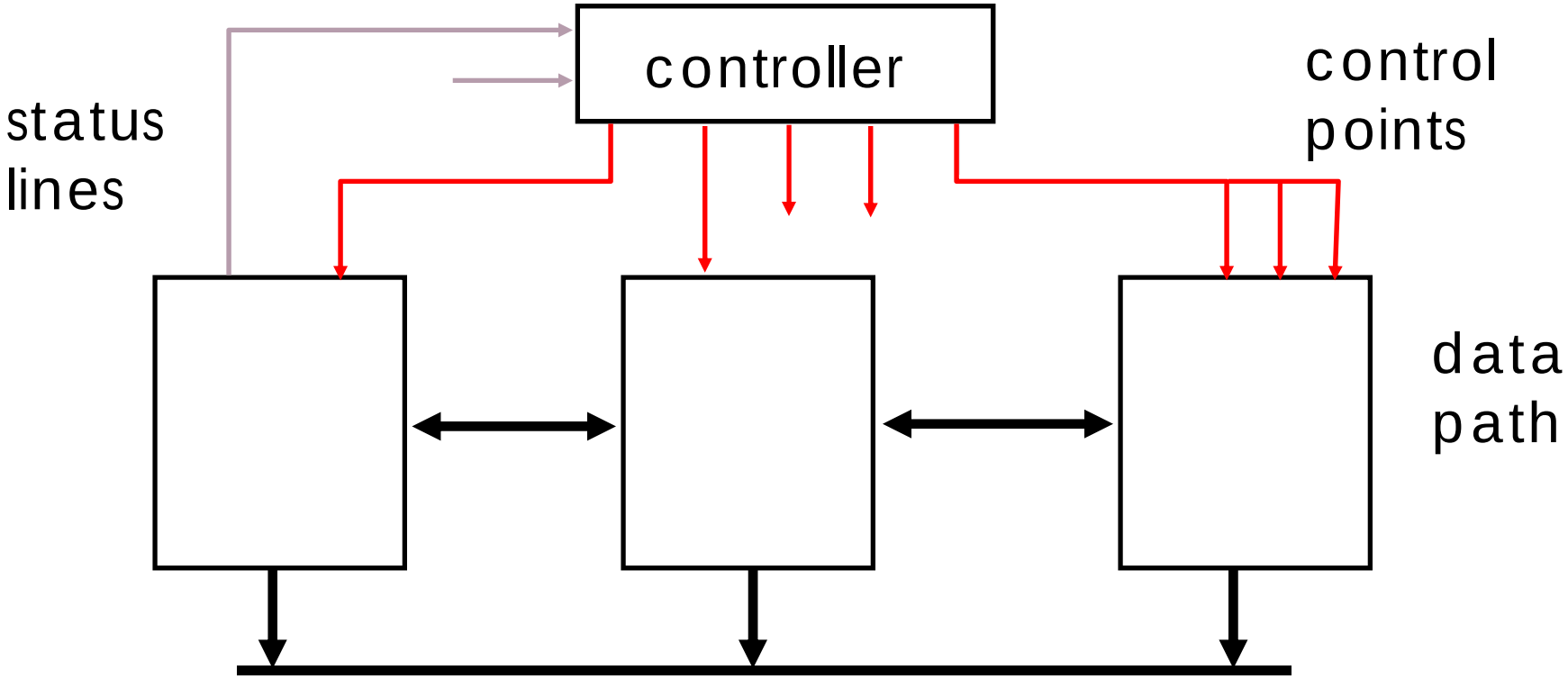
ISA favors particular microarchitecture style

- 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

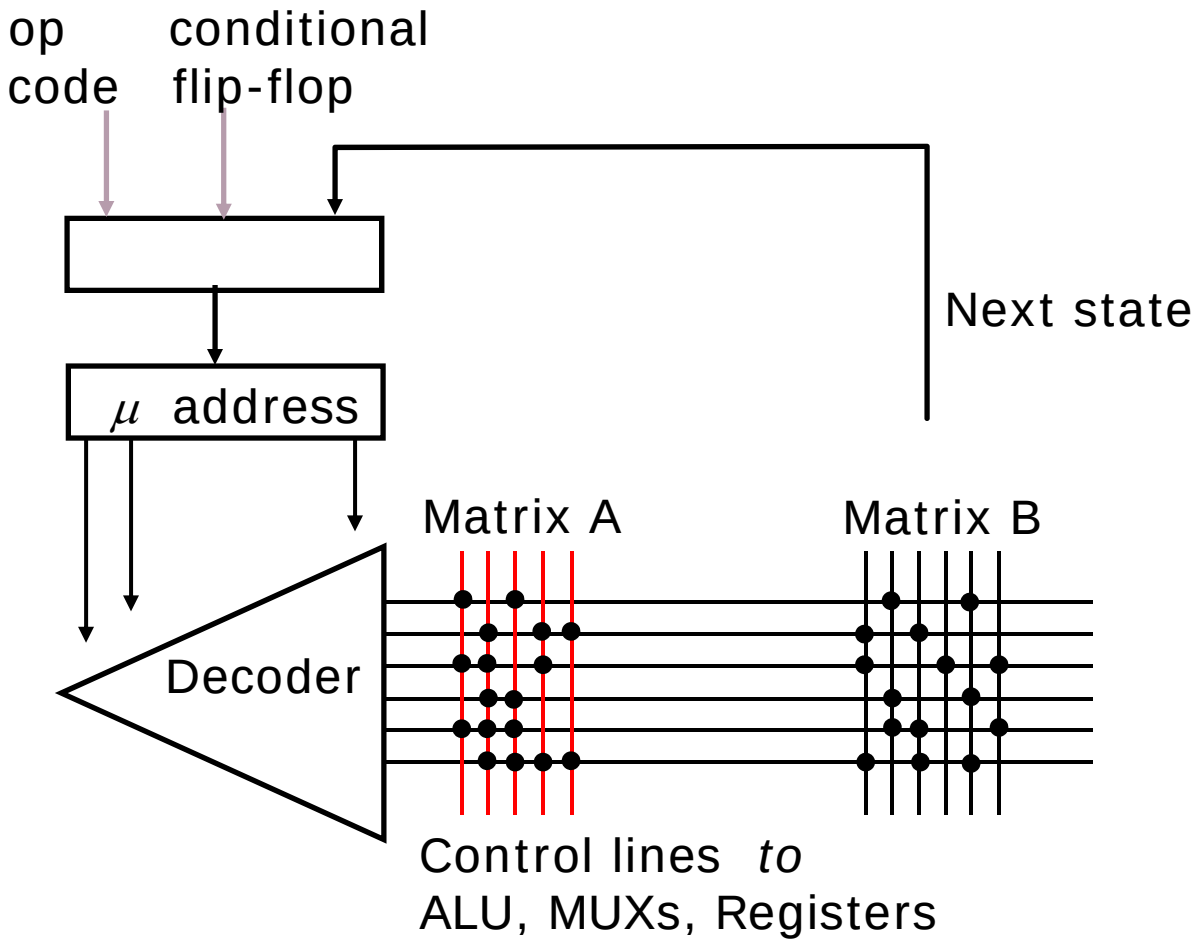
Hardware Organization



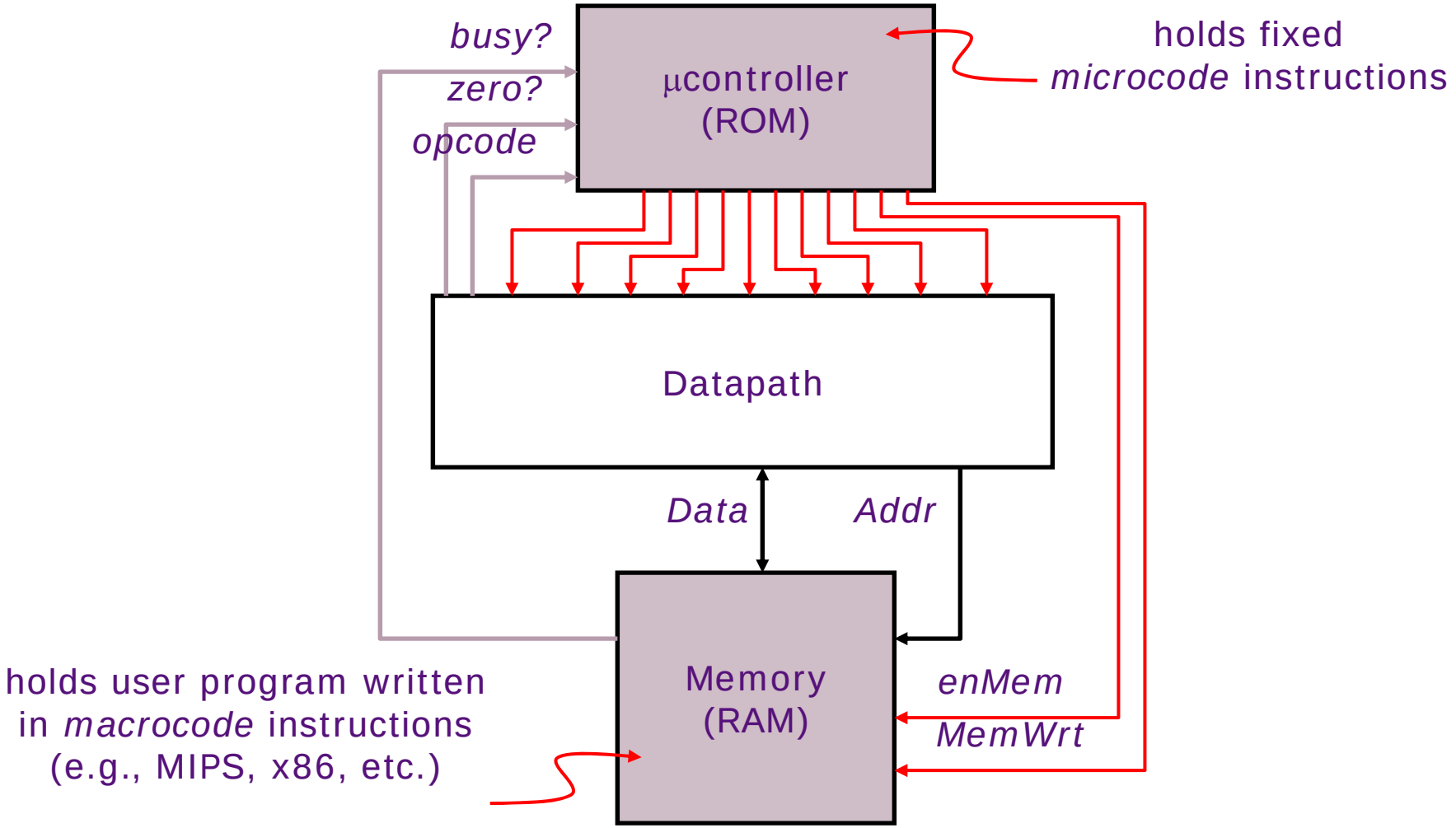
- Structure: How are components connected? Statically
- Behavior: How does data move between components? Dynamically

Microcontrol Unit

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



Microcoded Microarchitecture



MIPS32 ISA

- See Hennessy and Patterson, Appendix B 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

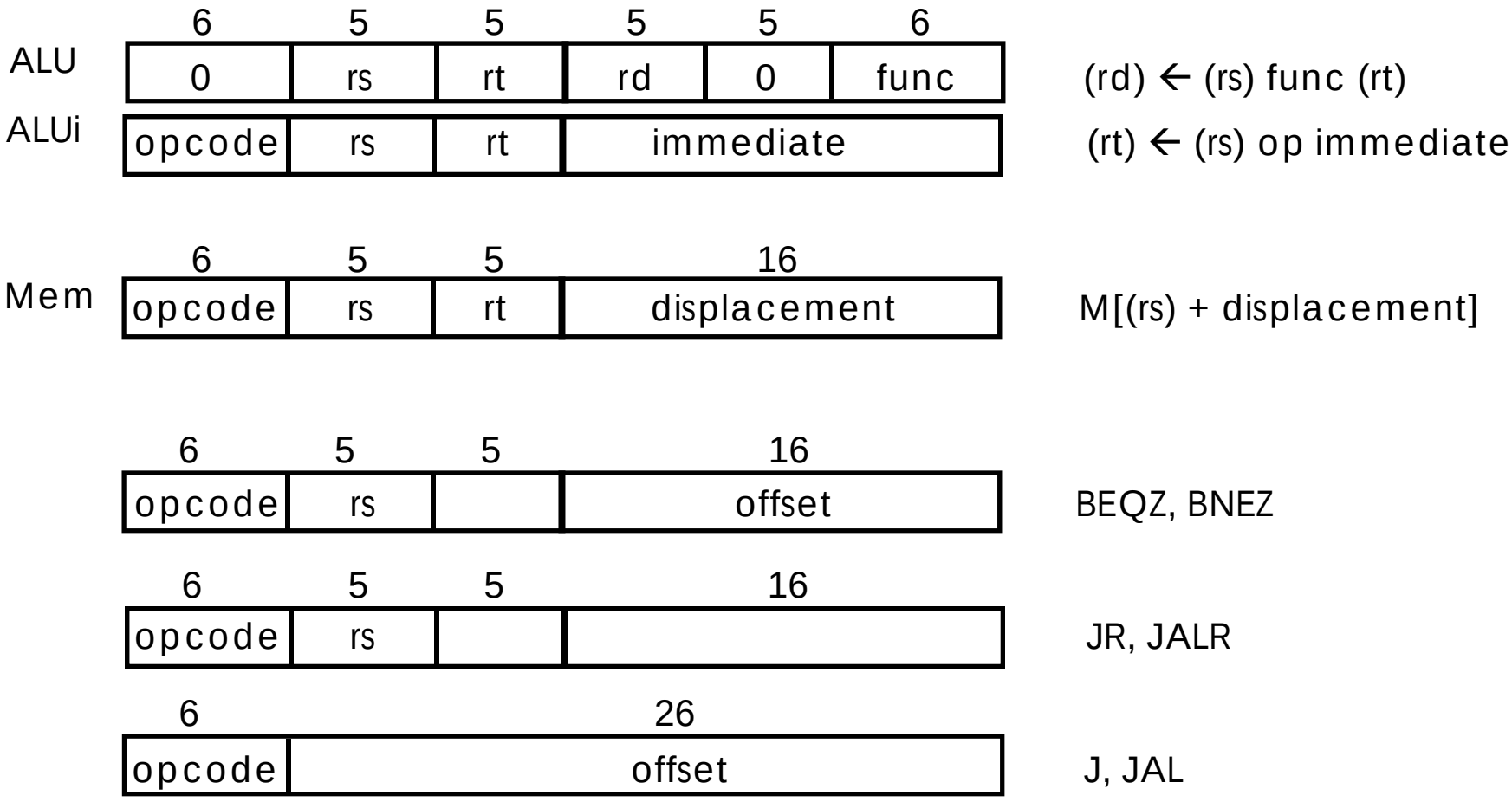
- 8-bit byte, 16-bit half word
- 32-bit word for integers
- 32-bit word for single precision floating point
- 64-bit word for double precision floating point

Load/Store style instruction set

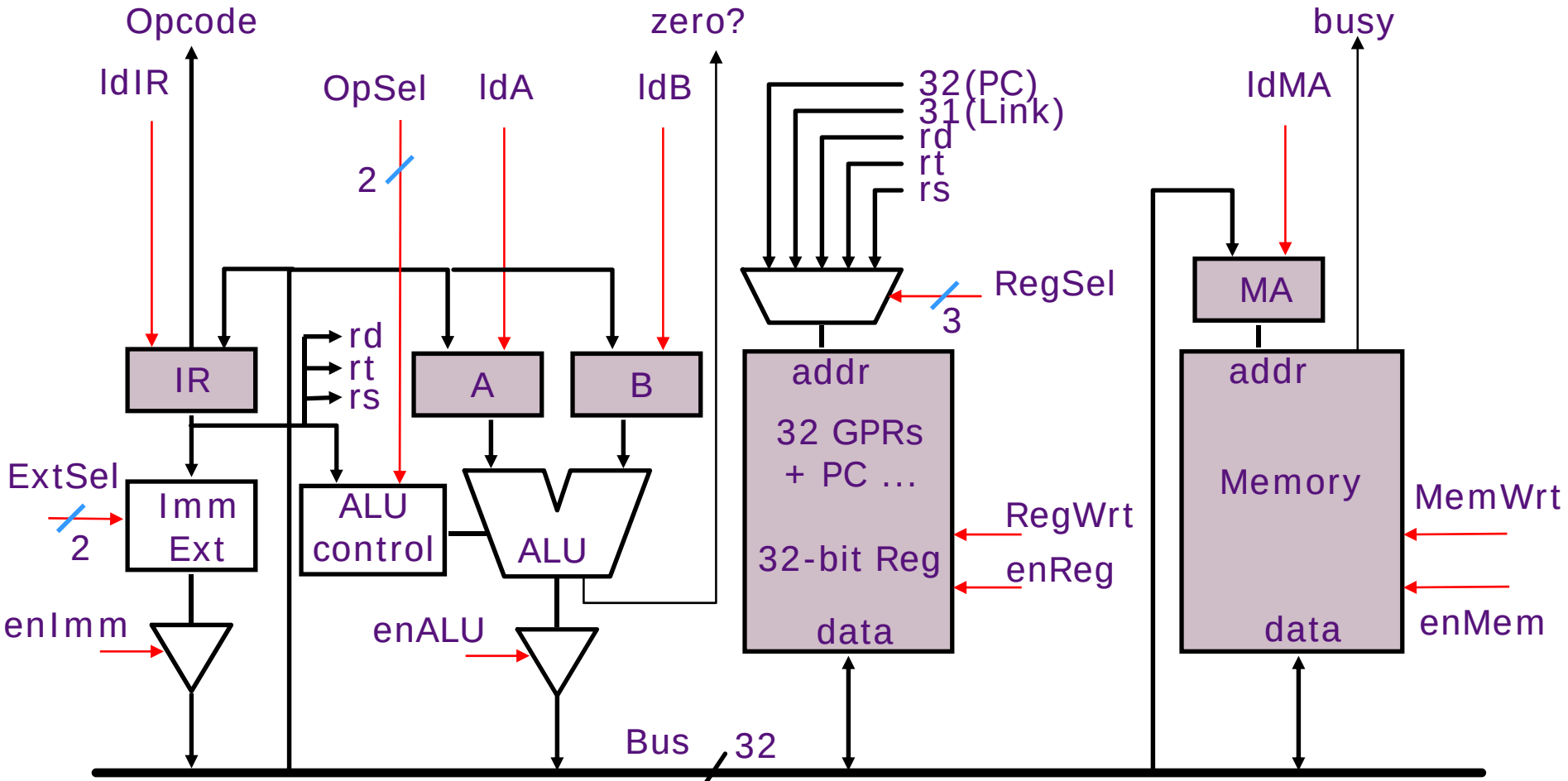
- data addressing modes: immediate & indexed
- branch addressing modes: PC relative & register indirect
- byte addressable memory: big-endian mode

All instructions are 32 bits

MIPS Instruction Formats



A Bus-Based MIPS Datapath

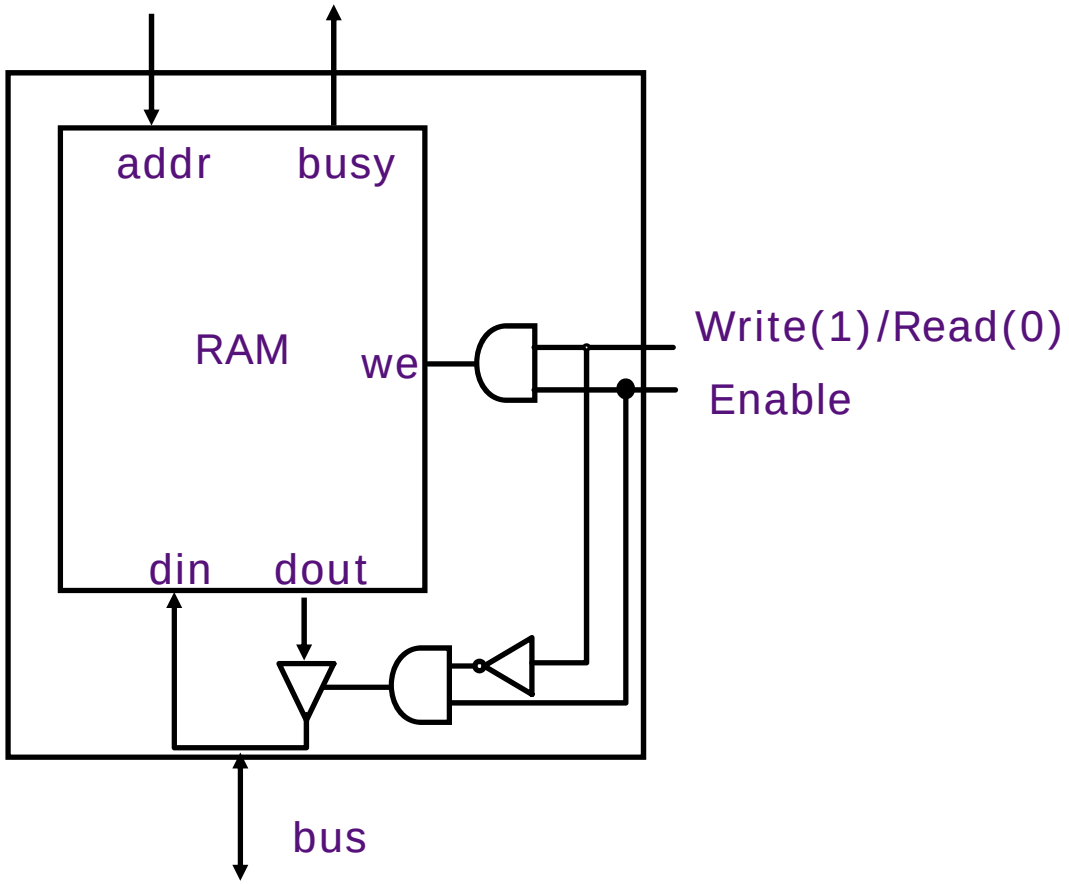


Microinstruction: register to register transfer (17 control signals)

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

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

Memory Module



Assumption: Memory operates 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

Microprogram Fragments

```
instr fetch:    MA ← PC                  # fetch current instr
                A ← PC                   # next PC calculation
                IR ← Memory
                PC ← A + 4
                dispatch on Opcode       # start microcode

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

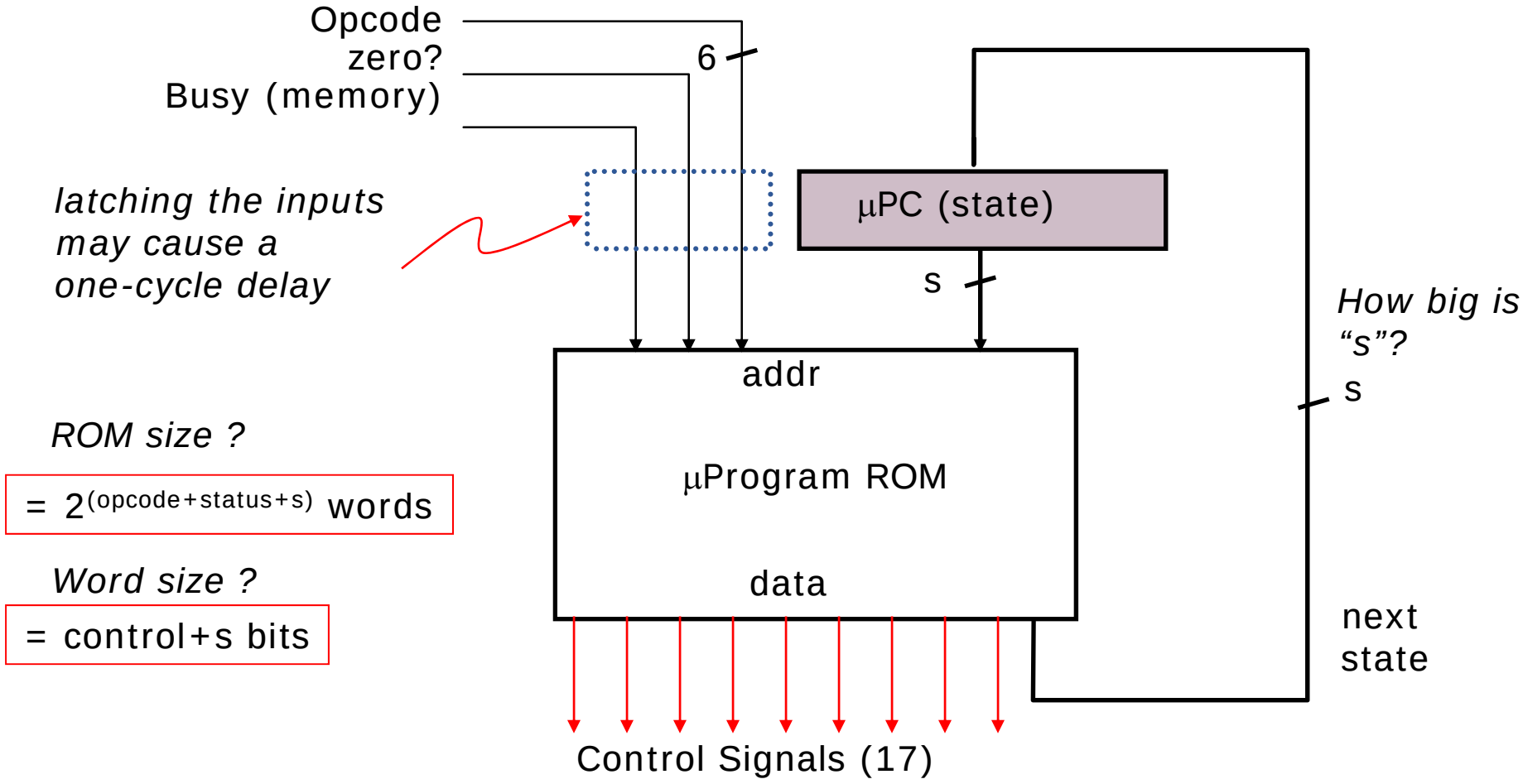
```
LW:         A ← Reg[rs]                 # compute address
            B ← Imm
            MA ← A + B
            Reg[rt] ← Memory            # load from memory
            do instruction fetch

J:          A ← PC
            B ← IR
            PC ← JumpTarg(A,B)          #JumpTarg(A,B) =
            do instruction fetch        {A[31:28],B[25:0],00}

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
```

MIPS Microcontroller



ROM size ?

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

Word size ?

= control+s bits

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	$fetch_3$
~~$fetch_3$~~	~~*~~	~~*~~	~~*~~	~~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

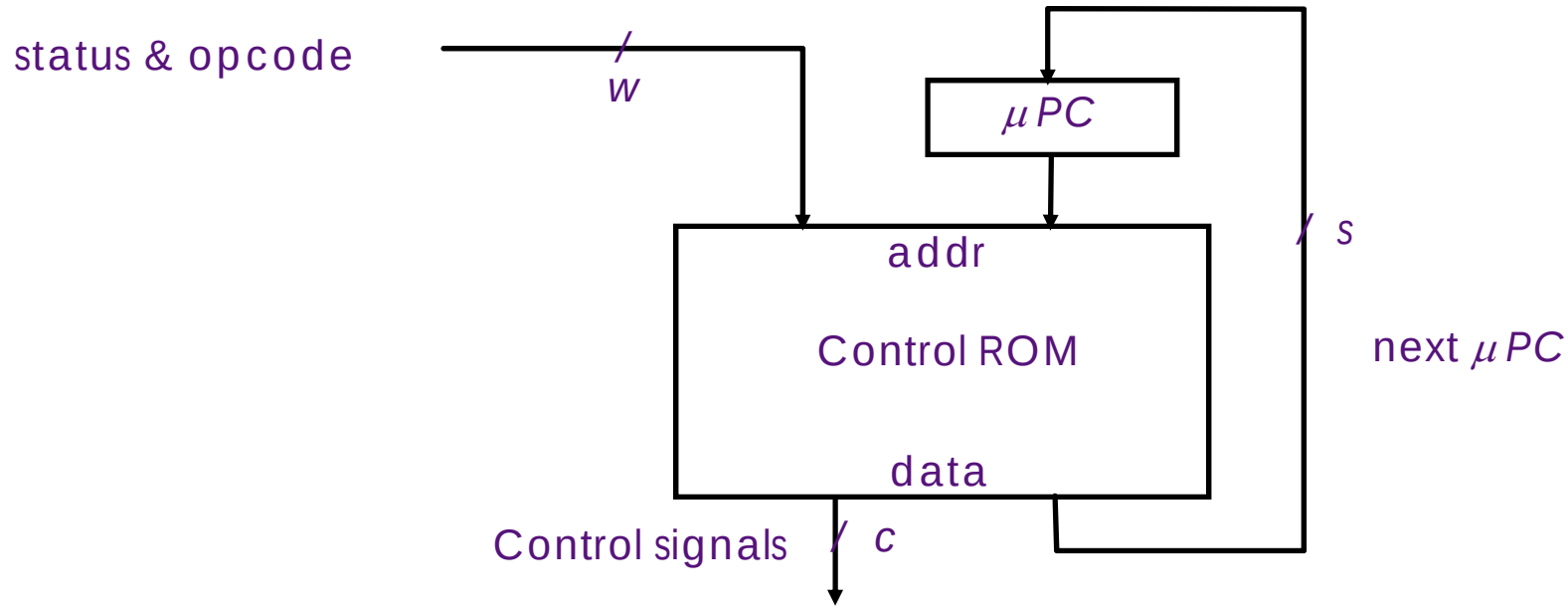
State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
$fetch_3$	ALU	*	*	PC ← A + 4	ALU_0
$fetch_3$	ALUi	*	*	PC ← A + 4	$ALUi_0$
$fetch_3$	LW	*	*	PC ← A + 4	LW_0
$fetch_3$	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

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

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

Size of Control Store



size = $2^{(w+s)}$ x (c + s)
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 must be fast, which is expensive

Reduce ROM height (= address bits)

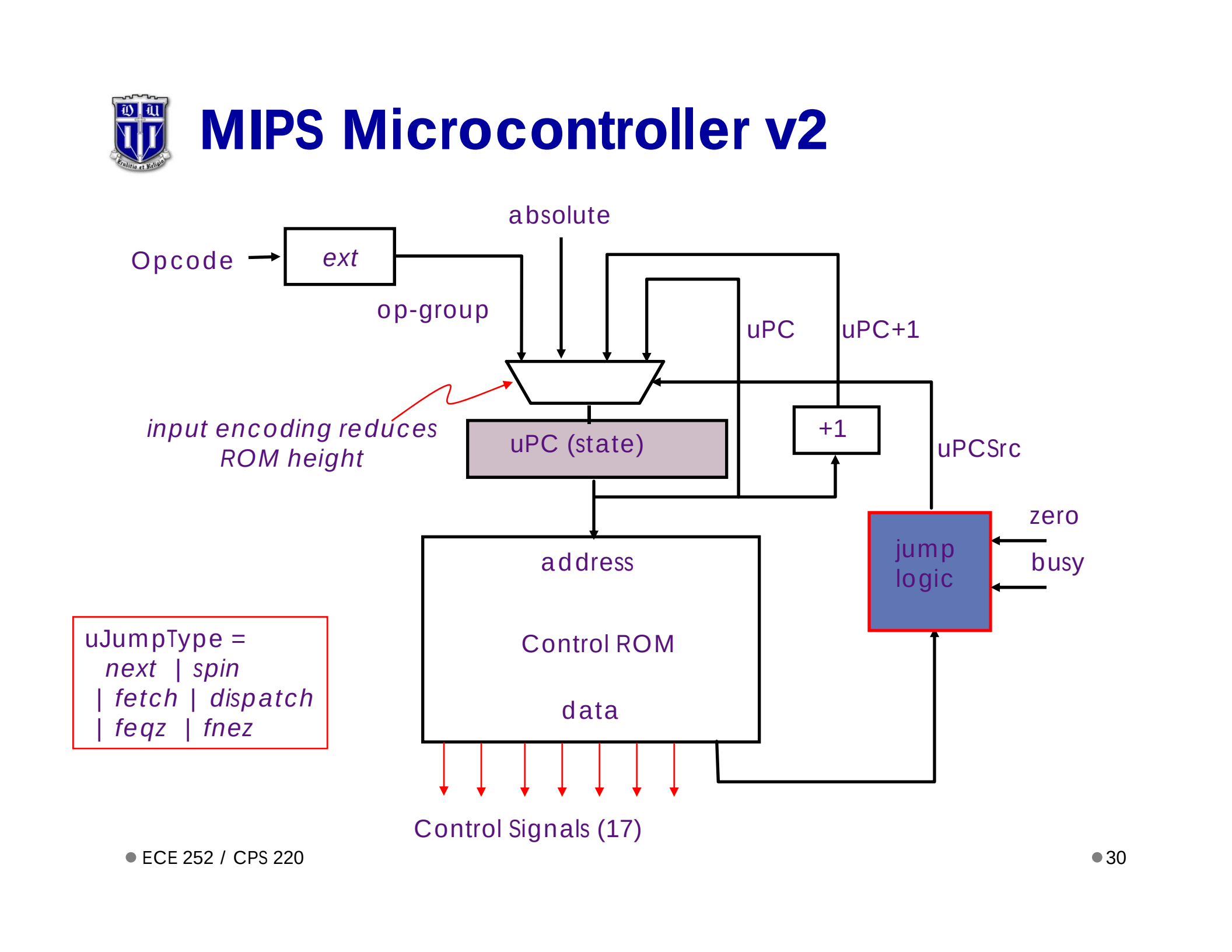
- reduce inputs by extra external logic
- each input bit doubles the size of the control store

- 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

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 and ALU

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

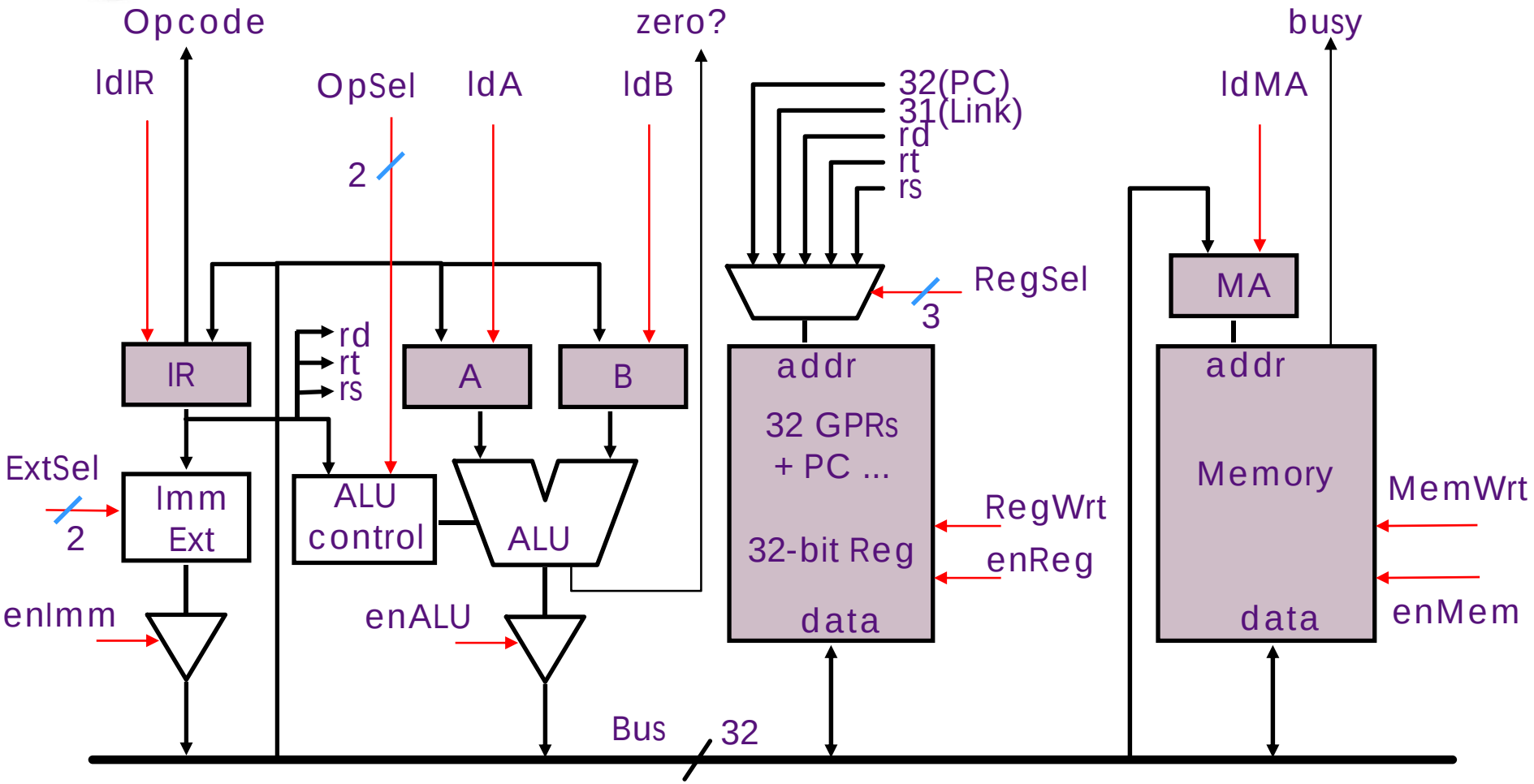
Load and Store

State	Control Points	Next State
LW0	A ← Reg[rs]	next
LW1	B ← sExt16(Imm)	next
LW2	MA ← A+B	next
LW3	Reg[rt] ← Memory	spin
LW4		fetch
…		
SW0	A ← Reg[rs]	next
SW1	B ← sExt16(Imm)	next
SW2	MA ← A+B	next
SW3	Memory ← Reg[rt]	spin
SW4		fetch

Jumps

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

Complex Instructions



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

Mem-Mem ALU Instruction

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

State	Control Points	Next State
$ALUMM_0$	MA ← Reg[rs]	next
$ALUMM_1$	A ← Memory	spin
$ALUMM_2$	MA ← Reg[rt]	next
$ALUMM_3$	B ← Memory	spin
$ALUMM_4$	MA ← Reg[rd]	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 instructions without a hardwired controller is difficult without datapath modifications

Performance

Microcode requires multiple cycles per instruction

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

Microcode requires multiple cycles per instruction

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

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)