CS 152 Computer Architecture and Engineering

Lecture 2 - Simple Machine Implementations

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

Last Time in Lecture 1

- Computer Science at crossroads from sequential to parallel computing
- Computer Architecture >> ISAs and RTL
  - CS152 is about interaction of hardware and software, and design of appropriate abstraction layers
- Comp. Arch. shaped by technology and applications
  - History provides lessons for the future
- Cost of software development a large constraint on architecture
  - Compatibility a key solution to software cost
- IBM 360 introduces notion of “family of machines” running same ISA but very different implementations
  - Six different machines released on same day (April 7, 1964)
  - “Future-proofing” for subsequent generations of machine

Instruction Set Architecture (ISA)

- The contract between software and hardware
- Typically described by giving all the programmer-visible state (registers + memory) plus the semantics of the instructions that operate on that state
- IBM 360 was first line of machines to separate ISA from implementation (aka. *microarchitecture*)
- Many implementations possible for a given ISA
  - E.g., today you can buy AMD or Intel processors that run the x86-64 ISA.
  - E.g.2: many cellphones use the ARM ISA with implementations from many different companies including TI, Qualcomm, Samsung, Marvell, etc.
  - E.g.3., the Soviets build code-compatible clones of the IBM360, as did Amdhal after he left IBM.

Microprogramming

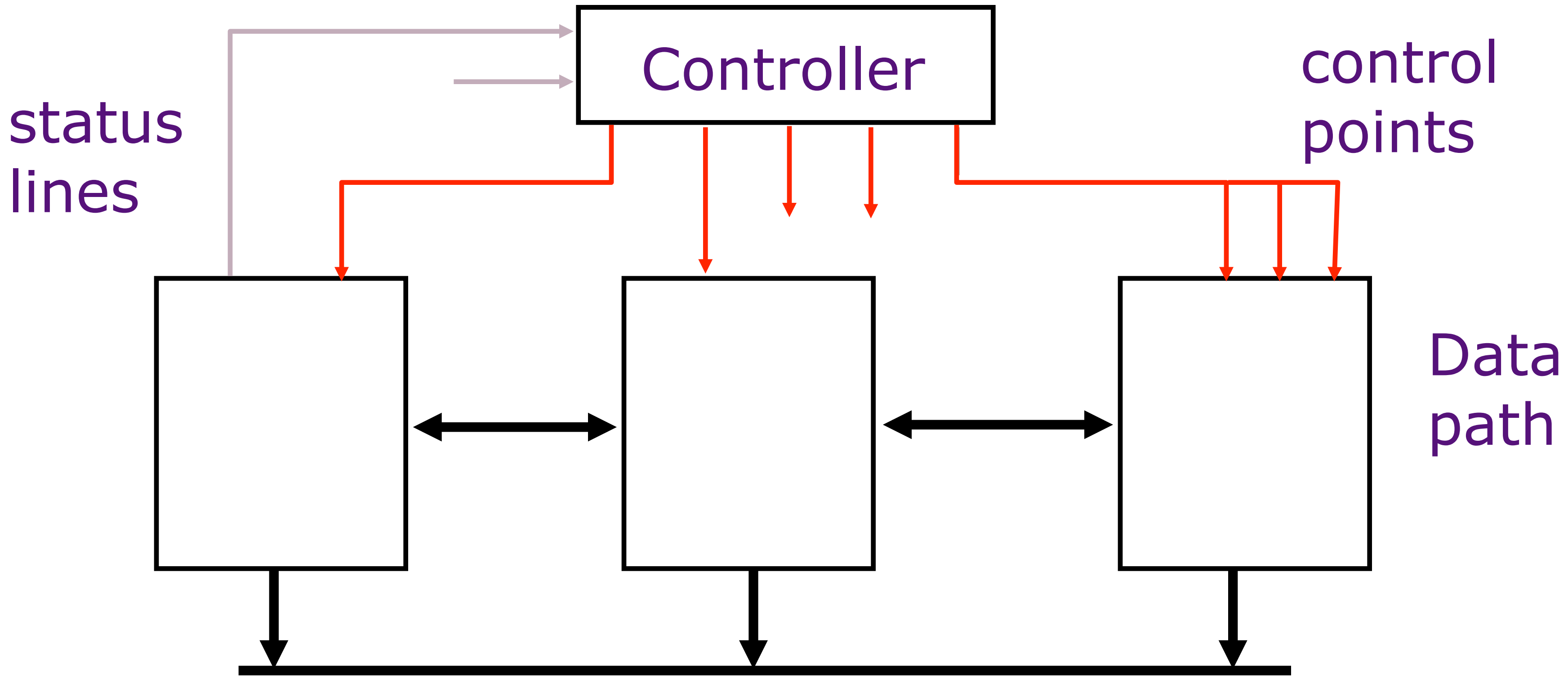
- Today, a brief look at microprogrammed machines
  - To show how to build very small processors with complex ISAs
  - To help you understand where CISC* machines came from
  - Because it is still used in the most common machines (x86, PowerPC, IBM360)
  - As a gentle introduction into machine structures
  - To help understand how technology drove the move to RISC*

* CISC/RISC names came much later than the style of machines they refer to.

ISA to Microarchitecture Mapping

- ISA often designed with particular microarchitectural style in mind, e.g.,
  - CISC $\Rightarrow$ microcoded
  - RISC $\Rightarrow$ hardwired, pipelined
  - VLIW $\Rightarrow$ fixed-latency in-order parallel pipelines
  - JVM $\Rightarrow$ software interpretation
- But can be implemented with any microarchitectural style
  - Intel Nehalem: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Simics: Software-interpreted SPARC RISC machine
  - Intel could implement a dynamically scheduled out-of-order VLIW Itanium (IA-64) processor
  - ARM Jazelle: A hardware JVM processor
  - This lecture: a microcoded RISC (MIPS) machine

Microarchitecture: *Implementation of an ISA*

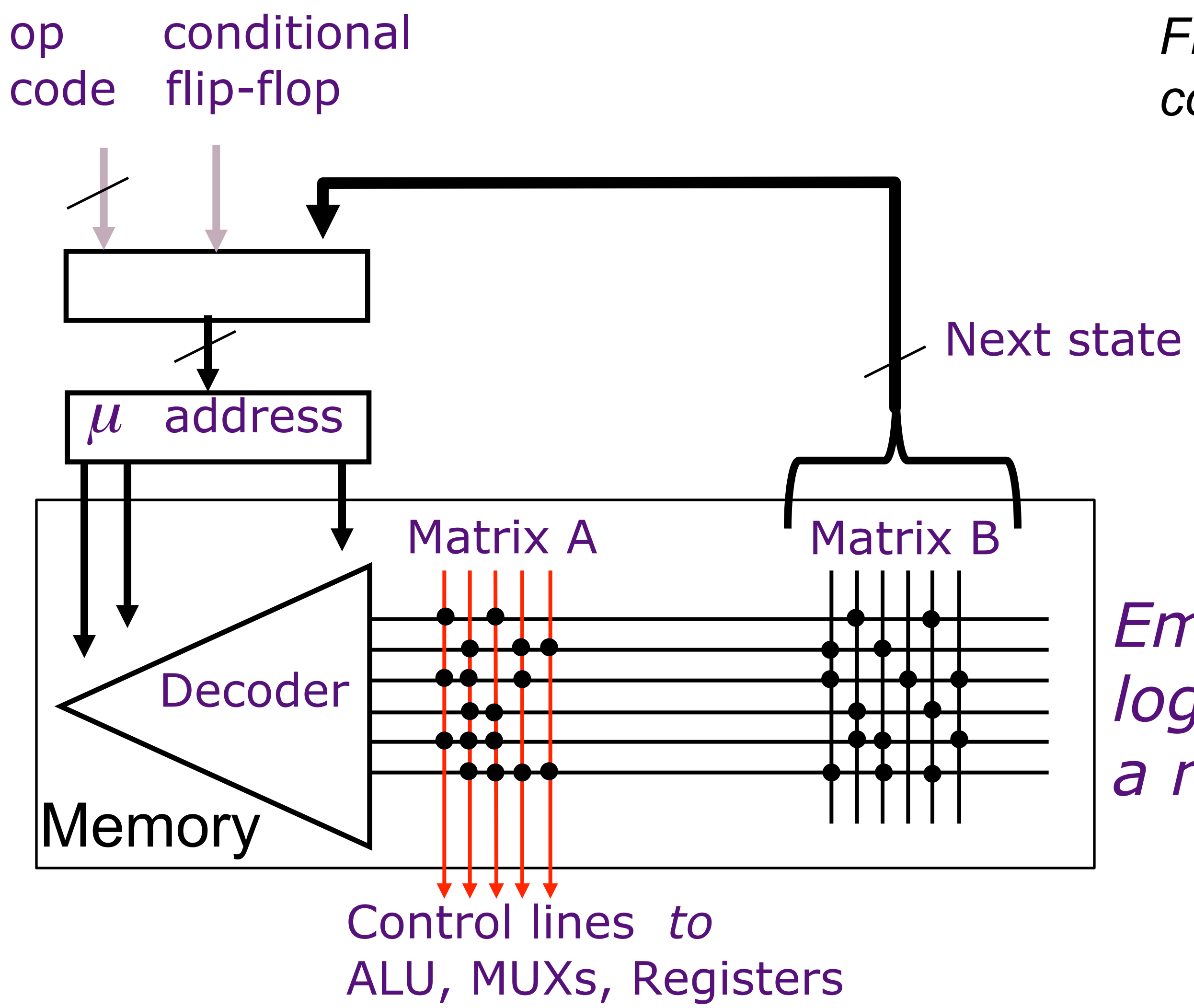


Structure: How components are connected.
Static

Behavior: How data moves between components
Dynamic

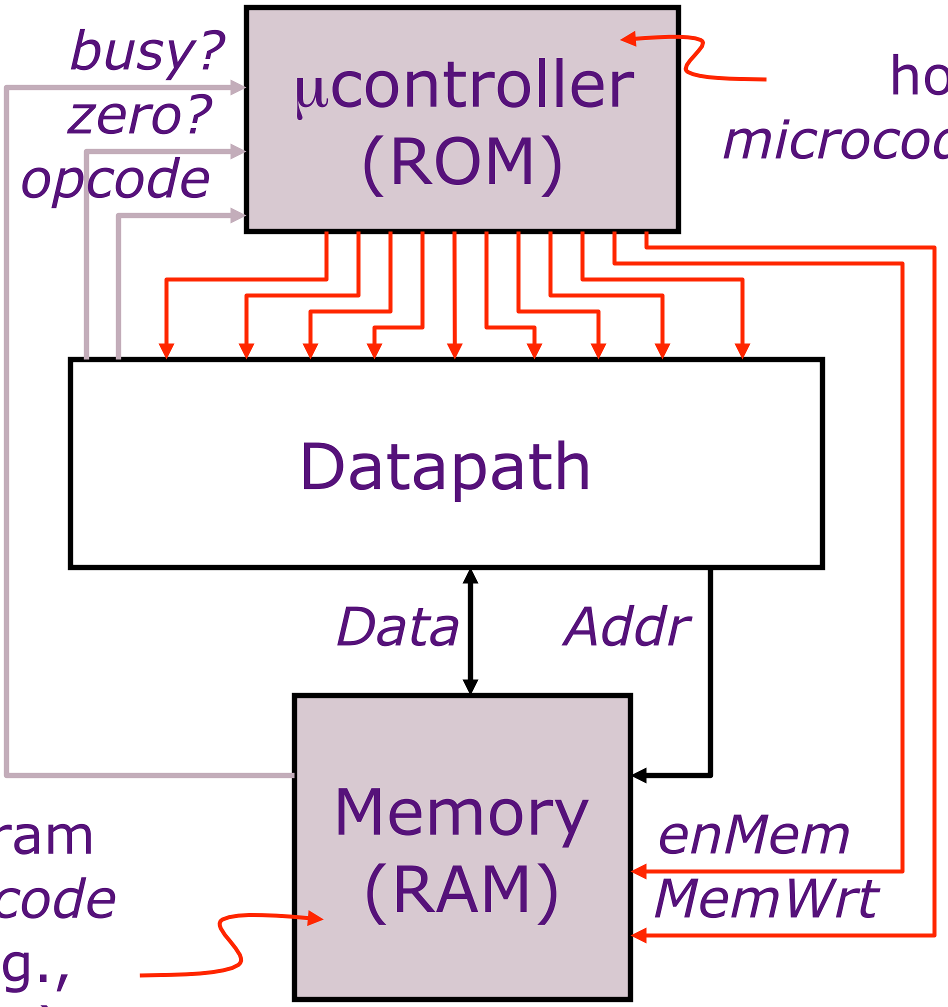
Microcontrol Unit *Maurice Wilkes, 1954*

First used in EDSAC-2, completed 1958



Embed the control logic state table in a memory array

Microcoded Microarchitecture



holds fixed *microcode* instructions

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 Appendix B 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

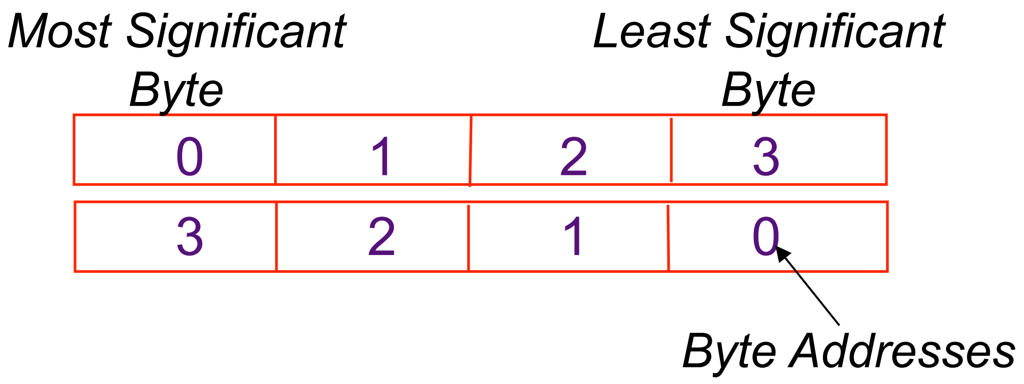
Data Formats and Memory Addresses

Data formats:

Bytes, Half words, words and double words

Some issues

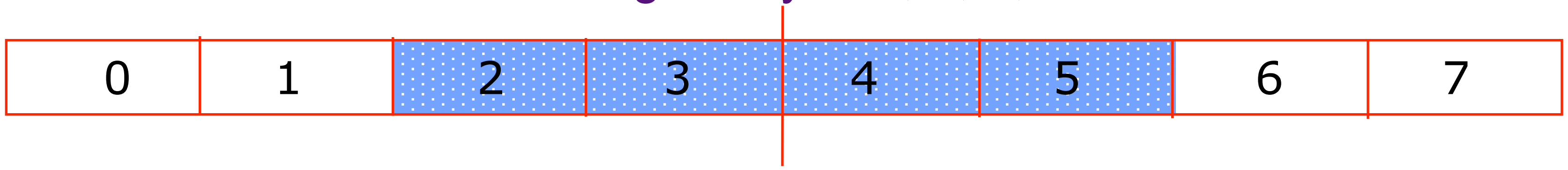
- *Byte addressing*



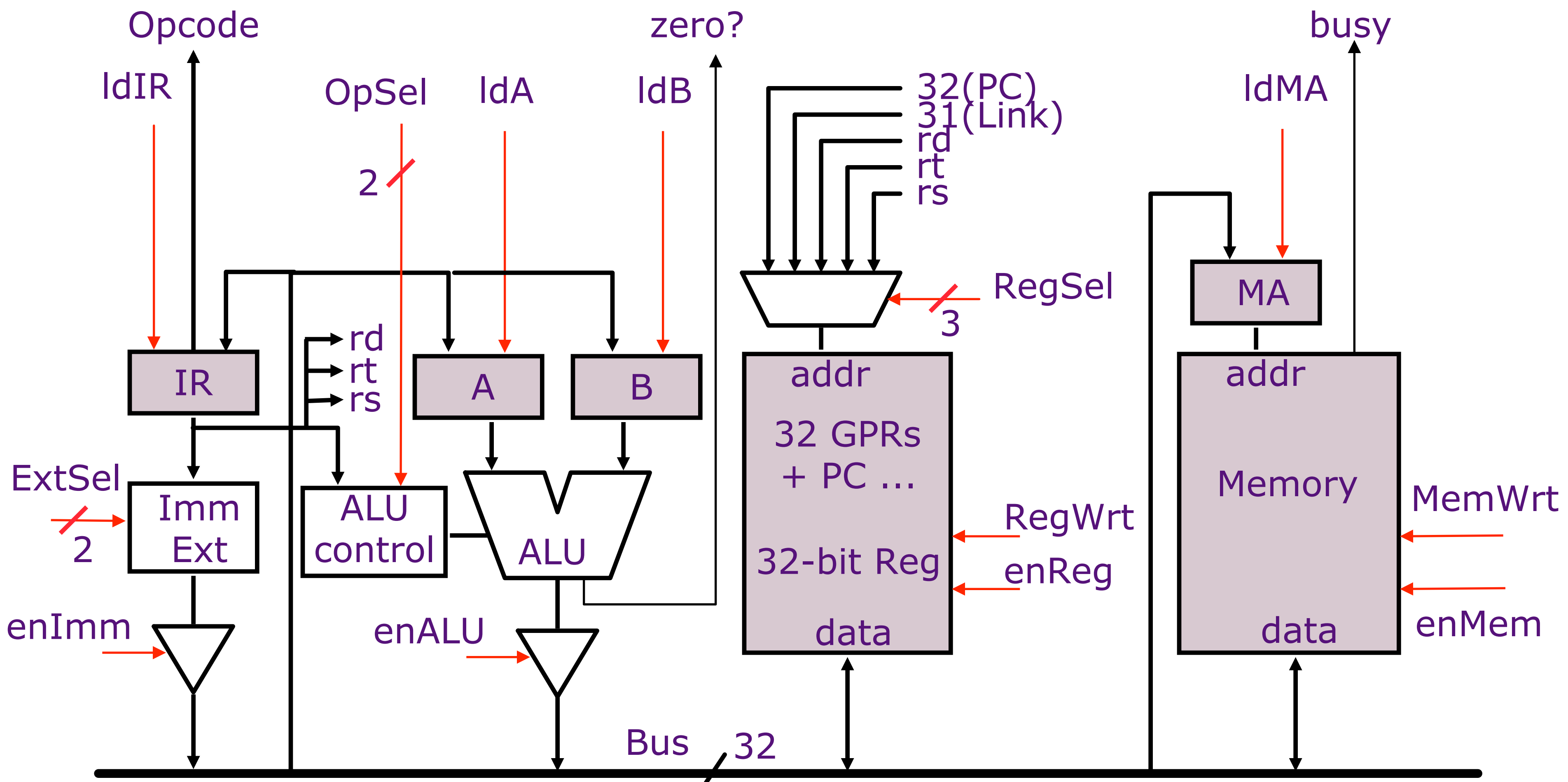
- *Word alignment*

Suppose the memory is organized in 32-bit words.

Can a word address begin only at 0, 4, 8, ... ?



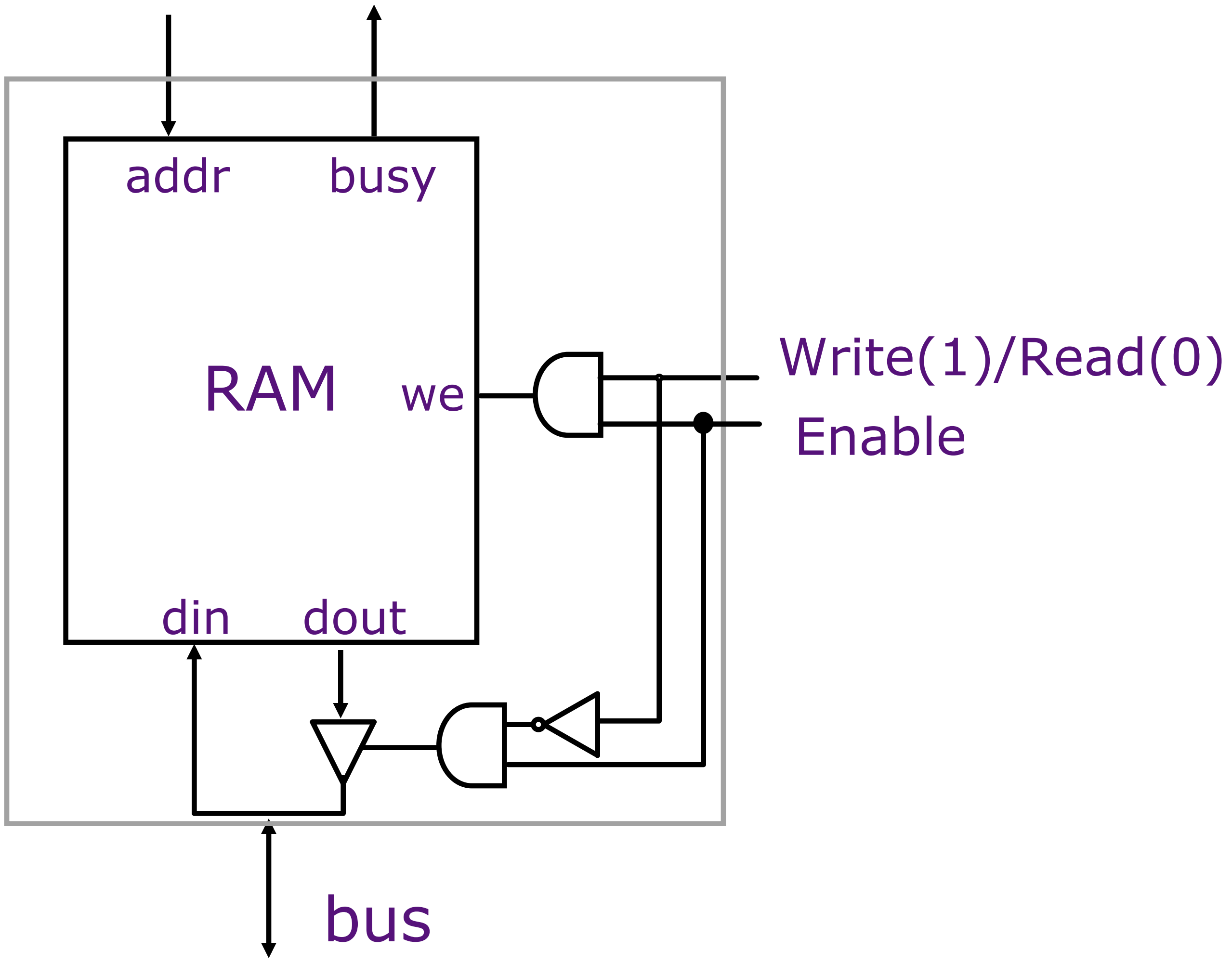
A Bus-based Datapath for MIPS



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

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 $\leftarrow$ PC
A $\leftarrow$ PC
IR $\leftarrow$ Memory
PC $\leftarrow$ A + 4
dispatch on OPcode

can be treated as a macro

ALU:
A $\leftarrow$ Reg[rs]
B $\leftarrow$ Reg[rt]
Reg[rd] $\leftarrow$ func(A,B)
do instruction fetch

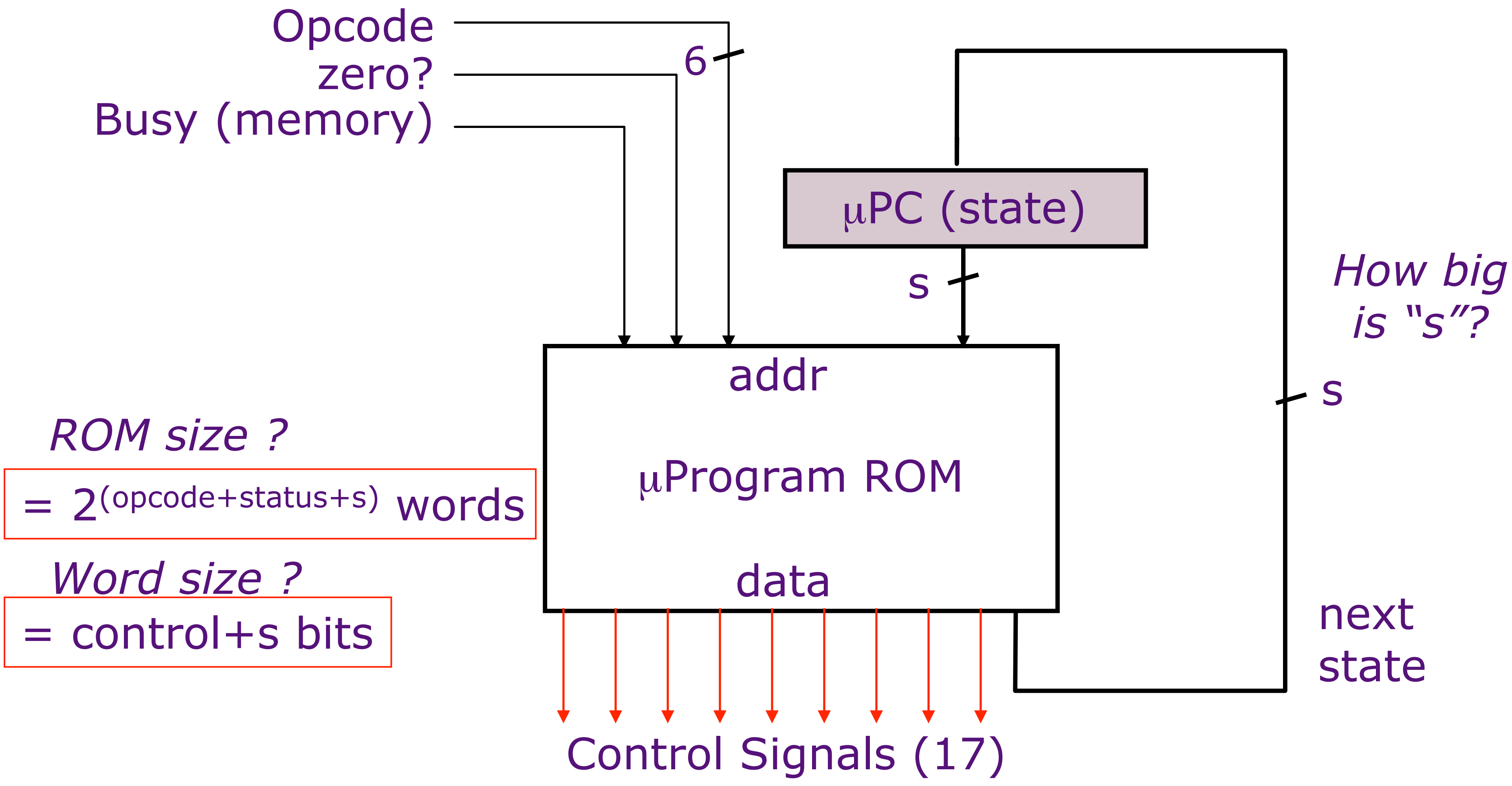
ALUi:
A $\leftarrow$ Reg[rs]
B $\leftarrow$ Imm *sign extension ...*
Reg[rt] $\leftarrow$ Opcode(A,B)
do instruction fetch

Microprogram Fragments *(cont.)*

```
LW:         A ← Reg[rs]
            B ← Imm
            MA ← A + B
            Reg[rt] ← Memory
            do instruction fetch

J:          A ← PC
            B ← IR
            PC ← JumpTarg(A,B)
            do instruction fetch

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
```

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

MIPS Microcontroller: *first attempt*



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 $\leftarrow$ 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

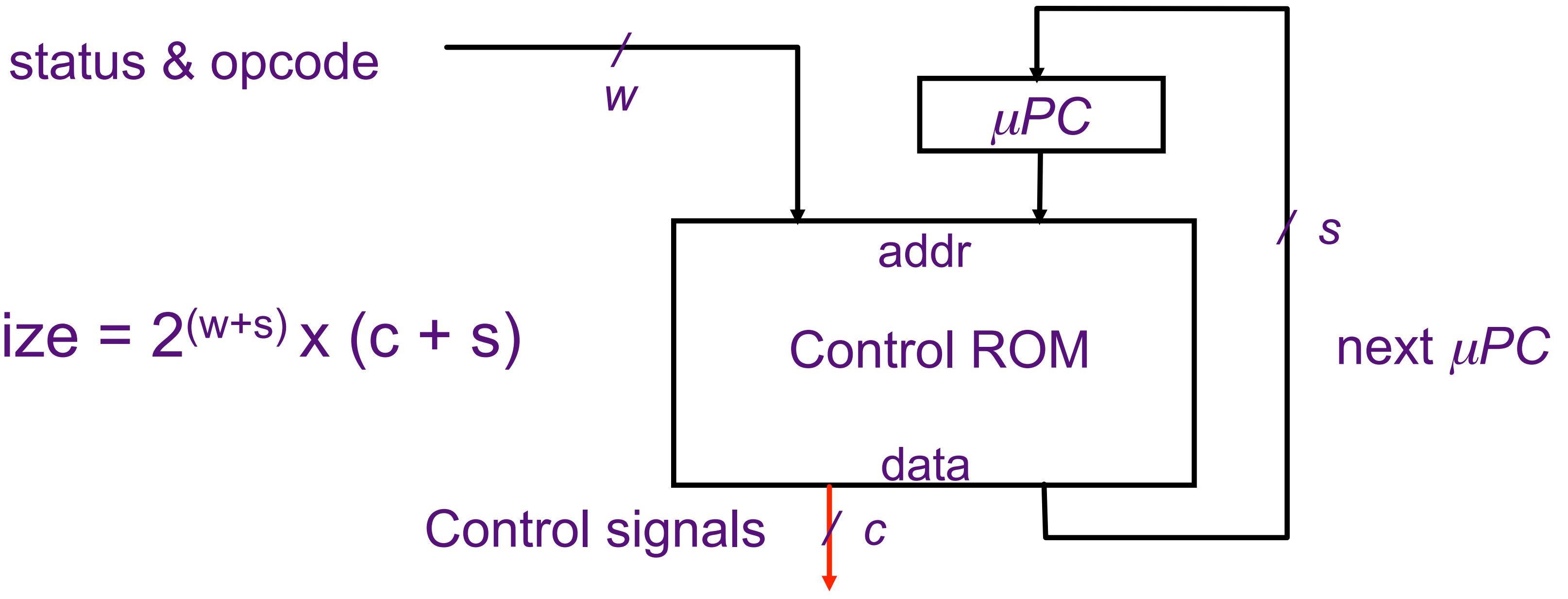
State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	$MA \leftarrow PC$	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	$IR \leftarrow Memory$	$fetch_2$
$fetch_2$	*	*	*	$A \leftarrow PC$	$fetch_3$
$fetch_3$	ALU	*	*	$PC \leftarrow A + 4$	ALU_0
$fetch_3$	ALUi	*	*	$PC \leftarrow A + 4$	$ALUi_0$
$fetch_3$	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 *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	*	....	$fetch_0$
$beqz_2$	*	*	*	$B \leftarrow sExt_{16}(Imm)$	$beqz_3$
$beqz_3$	*	*	*	$PC \leftarrow A+B$	$fetch_0$
...					

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* ⇒ *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)*

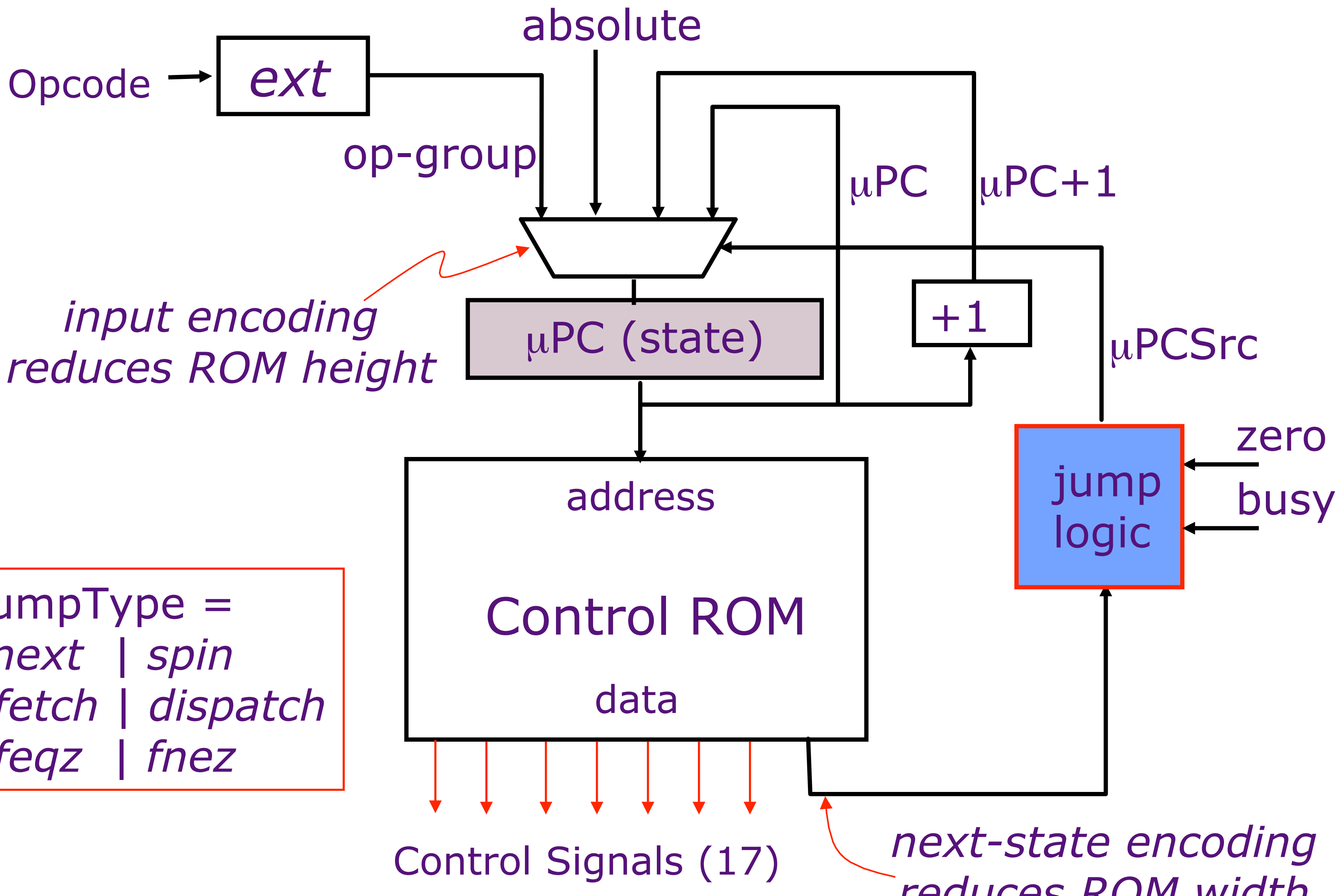
CS152 Administrivia

- Lab 1 coming out on Tuesday, together with PS1
- Lab 1 overview in Section, next Thursday, 2pm, 320 Soda
- Lab 1 and PS 1 due ***start of class*** Thursday Feb. 11
  - No extensions for Problem set. Zero credit afterwards.
  - Problem sets graded on 0,1,2 scale
  - Up to two free lab extensions per student, up till next class (Tuesday). Zero credit afterwards.
- Solutions to PS 1 released at ***end of same class***
- Section reviewing PS 1, same Thursday at 2pm
- First Quiz, in class, Tue Feb 16, 9:30-11AM
  - Closed book, no calculators, no computers, no cellphones
- PS 2 and Lab 2 handed out day of Quiz 1

Collaboration Policy

- Can collaborate to understand problem sets, but must turn in own solution. Some problems repeated from earlier years - do not copy solutions. (Quiz problems will not be repeated…)
- Each student must complete directed portion of the lab by themselves. OK to collaborate to understand how to run labs
  - Class news group info on web site.
  - Lab reports must be readable English summaries. Zero credit for handing in output log files from experiments.
- Can work in group of up to 3 students for open-ended portion of each lab
  - OK to be in different group for each lab -just make sure to label participants' names clearly on each turned-in lab section

MIPS Controller V2



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

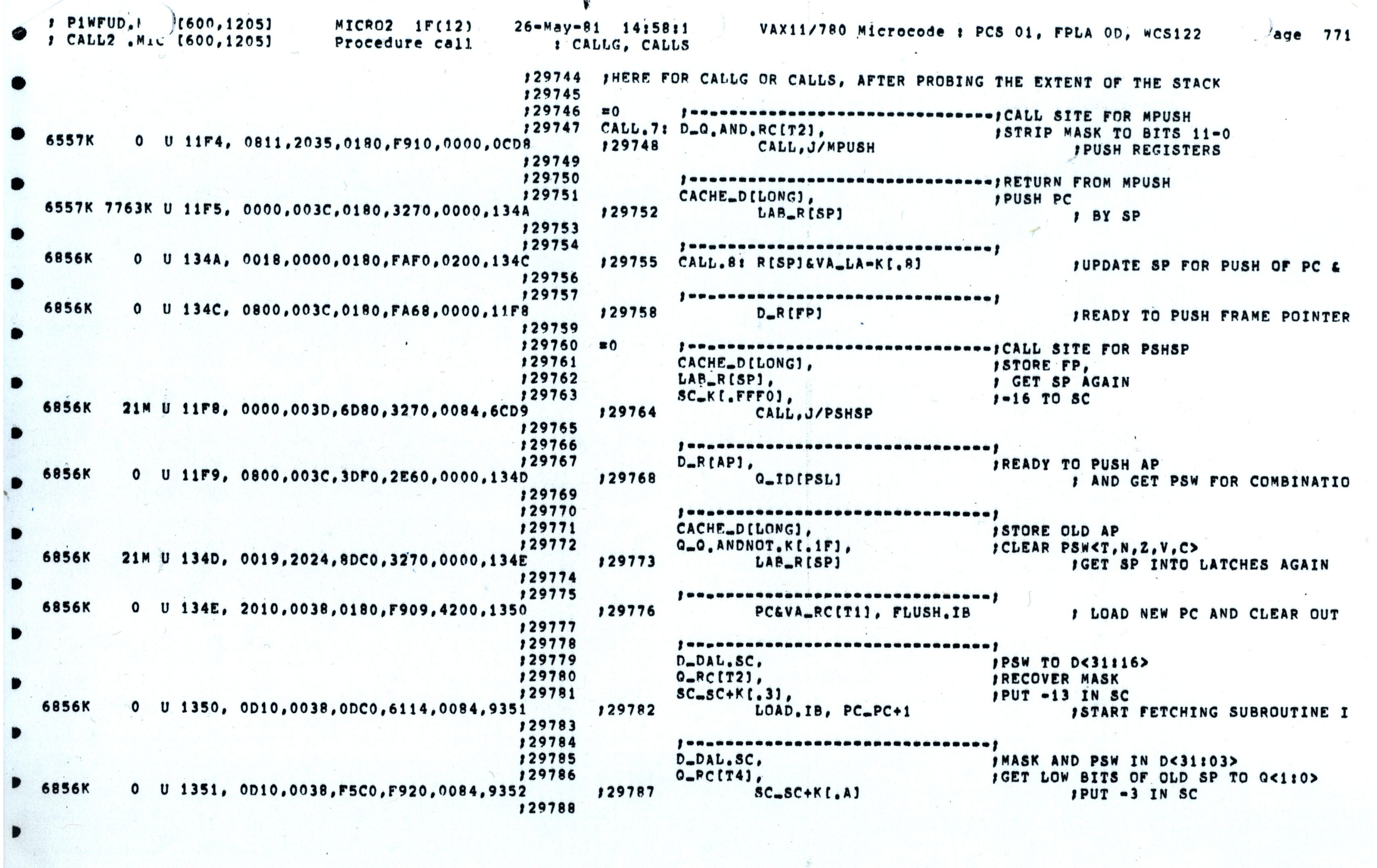
Branches: *MIPS-Controller-2*

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*

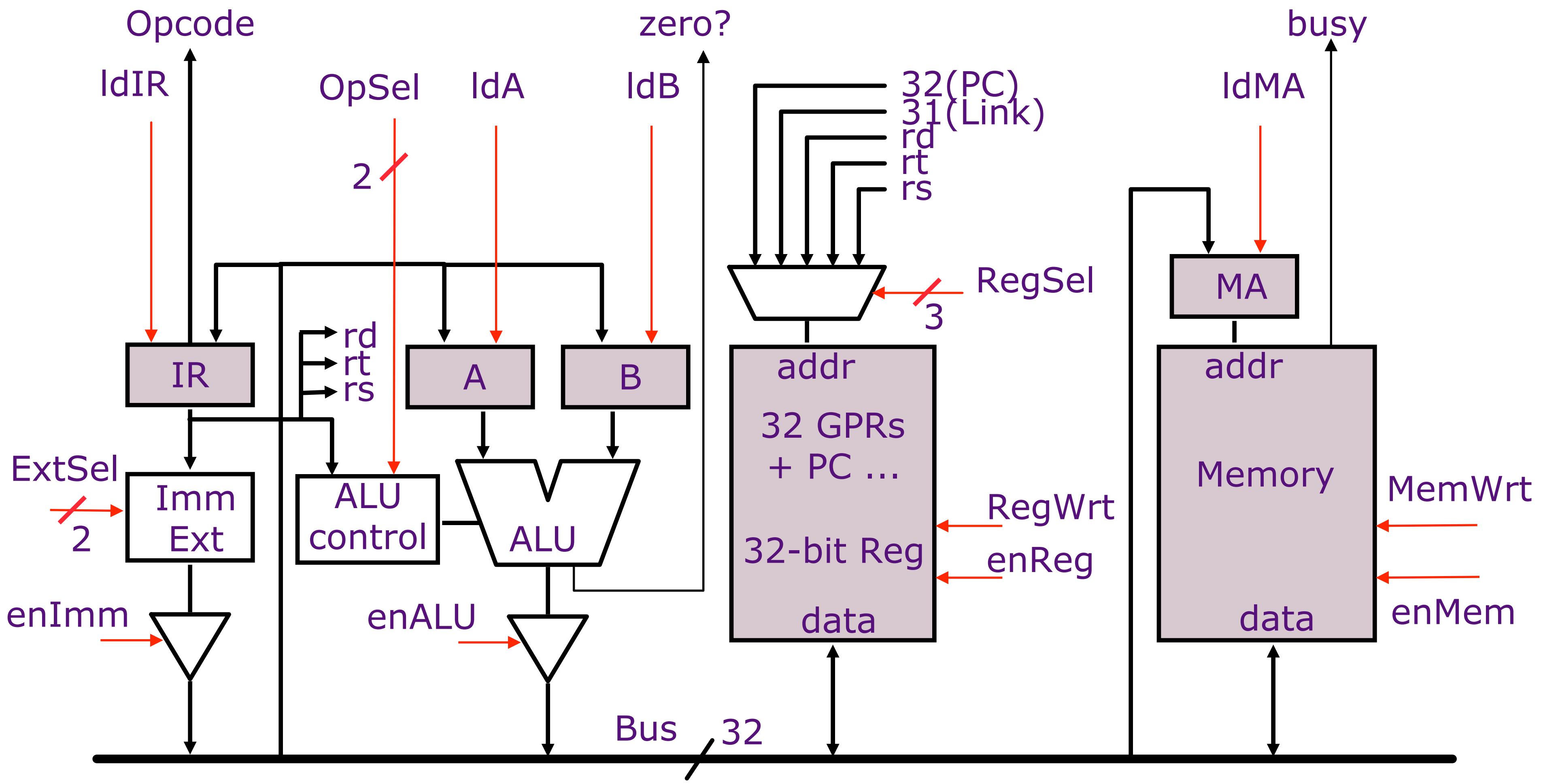
State	Control points	next-state
J_0	$A \leftarrow PC$	next
J_1	$B \leftarrow IR$	next
J_2	$PC \leftarrow \text{JumpTarg}(A,B)$	fetch
JR_0	$A \leftarrow \text{Reg}[rs]$	next
JR_1	$PC \leftarrow A$	fetch
JAL_0	$A \leftarrow PC$	next
JAL_1	$\text{Reg}[31] \leftarrow A$	next
JAL_2	$B \leftarrow IR$	next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

VAX 11-780 Microcode



```
; P1WFUD.[illegible] [600,1205]      MICRO2  1F(12)    26-May-81  14:58:1      VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122      age  771
; CALL2 .MIC [600,1205]      Procedure call          : CALLG, CALLS

                                                  ;29744  ;HERE FOR CALLG OR CALLS, AFTER PROBING THE EXTENT OF THE STACK
                                                  ;29745
                                                  ;29746  =0      ;--------------------------------;CALL SITE FOR MPUSH
                                                  ;29747  CALL.7: D_Q.AND.RC[T2],                  ;STRIP MASK TO BITS 11-0
6557K     0  U 11F4, 0811,2035,0180,F910,0000,0CD8  ;29748          CALL,J/MPUSH                            ;PUSH REGISTERS
                                                  ;29749
                                                  ;29750          ;--------------------------------;RETURN FROM MPUSH
                                                  ;29751          CACHE_D[LONG],                   ;PUSH PC
6557K 7763K U 11F5, 0000,003C,0180,3270,0000,134A  ;29752                  LAB_R[SP]                        ; BY SP
                                                  ;29753
                                                  ;29754          ;--------------------------------;
6856K     0  U 134A, 0018,0000,0180,FAF0,0200,134C  ;29755  CALL.8: R[SP]&VA_LA-K[.8]                       ;UPDATE SP FOR PUSH OF PC &
                                                  ;29756
                                                  ;29757          ;--------------------------------;
6856K     0  U 134C, 0800,003C,0180,FA68,0000,11F8  ;29758                  D_R[FP]                         ;READY TO PUSH FRAME POINTER
                                                  ;29759
                                                  ;29760  =0      ;--------------------------------;CALL SITE FOR PSHSP
                                                  ;29761          CACHE_D[LONG],                   ;STORE FP,
                                                  ;29762          LAB_R[SP],                       ; GET SP AGAIN
                                                  ;29763          SC_K[.FFF0],                     ;-16 TO SC
6856K   21M U 11F8, 0000,003D,6D80,3270,0084,6CD9  ;29764                  CALL,J/PSHSP
                                                  ;29765
                                                  ;29766          ;--------------------------------;
                                                  ;29767          D_R[AP],                         ;READY TO PUSH AP
6856K     0  U 11F9, 0800,003C,3DF0,2E60,0000,134D  ;29768                  Q_ID[PSL]                       ; AND GET PSW FOR COMBINATIO
                                                  ;29769
                                                  ;29770          ;--------------------------------;
                                                  ;29771          CACHE_D[LONG],                   ;STORE OLD AP
                                                  ;29772          Q_Q.ANDNOT.K[.1F],               ;CLEAR PSW<T,N,Z,V,C>
6856K   21M U 134D, 0019,2024,8DC0,3270,0000,134E  ;29773                  LAB_R[SP]                       ;GET SP INTO LATCHES AGAIN
                                                  ;29774
                                                  ;29775          ;--------------------------------;
6856K     0  U 134E, 2010,0038,0180,F909,4200,1350  ;29776                  PC&VA_RC[T1], FLUSH.IB          ; LOAD NEW PC AND CLEAR OUT
                                                  ;29777
                                                  ;29778          ;--------------------------------;
                                                  ;29779          D_DAL.SC,                        ;PSW TO D<31:16>
                                                  ;29780          Q_RC[T2],                        ;RECOVER MASK
                                                  ;29781          SC_SC+K[.3],                     ;PUT -13 IN SC
6856K     0  U 1350, 0D10,0038,0DC0,6114,0084,9351  ;29782                  LOAD.IB, PC_PC+1                ;START FETCHING SUBROUTINE I
                                                  ;29783
                                                  ;29784          ;--------------------------------;
                                                  ;29785          D_DAL.SC,                        ;MASK AND PSW IN D<31:03>
                                                  ;29786          Q_RC[T4],                        ;GET LOW BITS OF OLD SP TO Q<1:0>
6856K     0  U 1351, 0D10,0038,F5C0,F920,0084,9352  ;29787                  SC_SC+K[.A]                     ;PUT -3 IN SC
                                                  ;29788
```

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

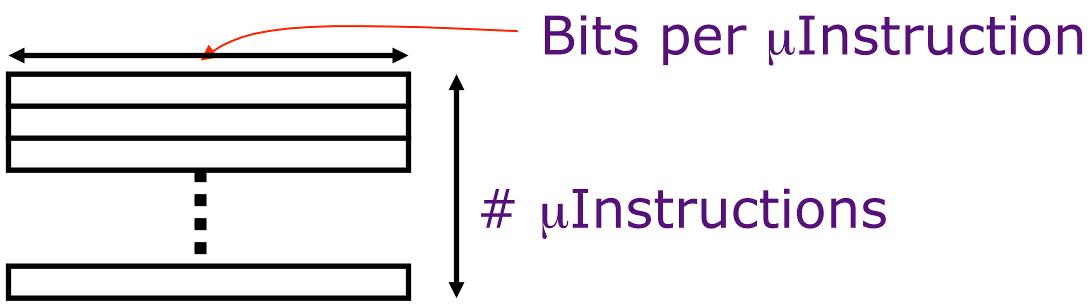
Cycle time ?

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

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

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

Horizontal vs Vertical μCode

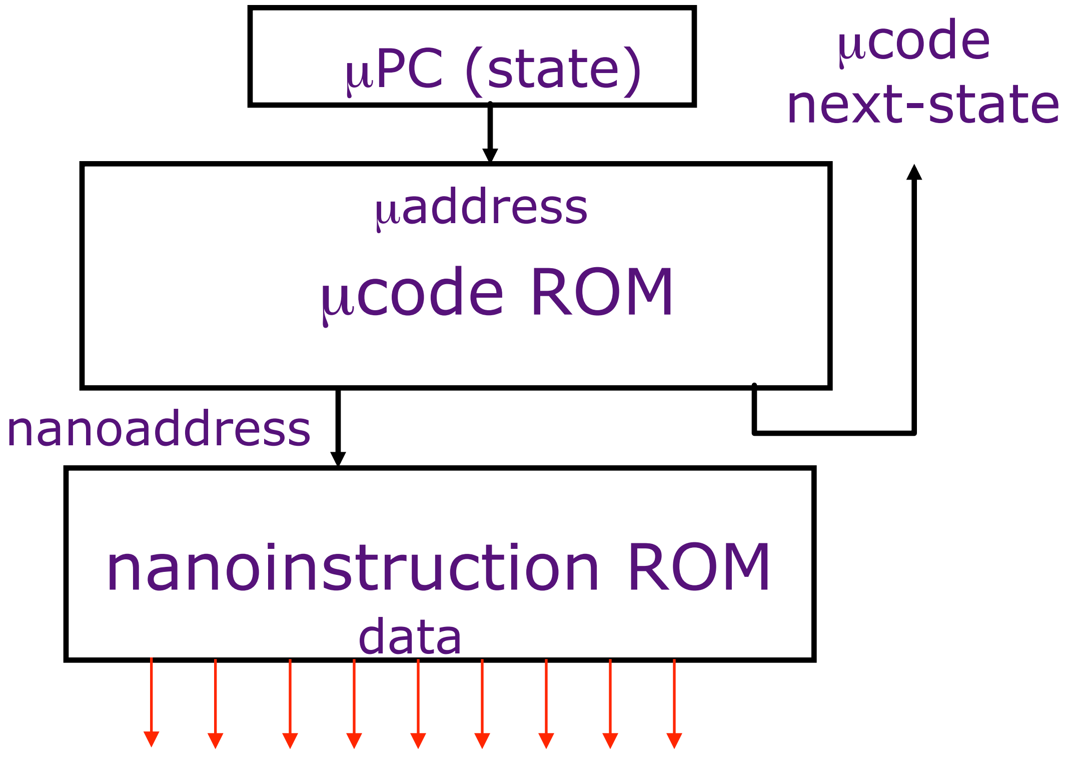


- Horizontal μcode has wider μinstructions
  - Multiple parallel operations per μinstruction
  - Fewer microcode steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical μcode has narrower μinstructions
  - Typically a single datapath operation per μinstruction
    - separate μinstruction for branches
  - More microcode steps 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 ← Reg[rs]
...
$ALUi_0$ A ← Reg[rs]
...



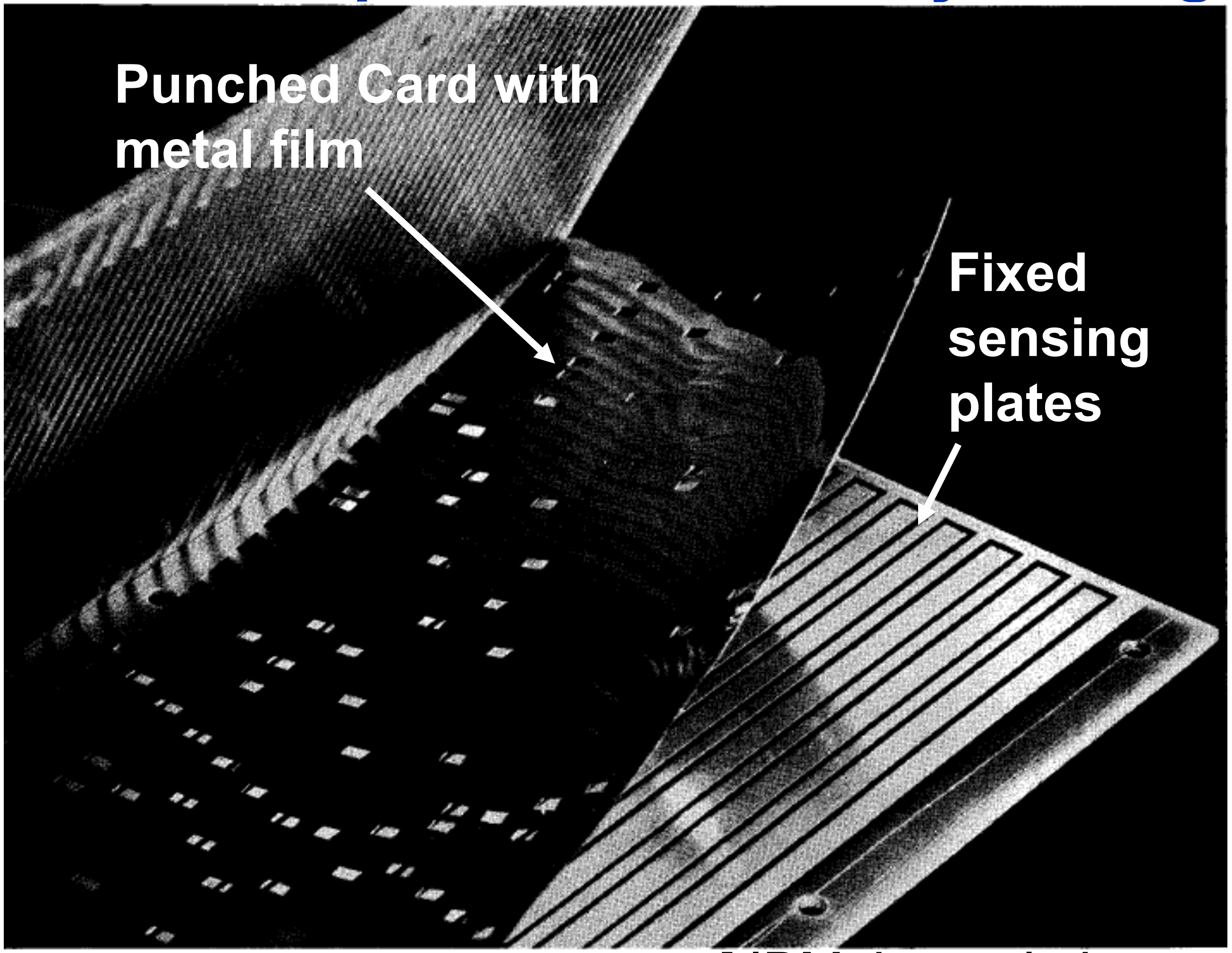
- MC68000 had 17-bit μcode containing either 10-bit μjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

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 the fastest models (75 and 95) were hardwired

IBM Card Capacitor Read-Only Storage



[IBM Journal, January 1961]

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 in RAM 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 processor
- 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: early Eighties

- Evolution bred more complex micro-machines
  - Complex instruction sets led to need for subroutine and call stacks in µcode
  - Need for fixing bugs in control programs was in conflict with read-only nature of µROM
  - ➔WCS (B1700, QMachine, Intel i432, …)
- With the advent of VLSI technology assumptions about ROM & RAM speed became invalid →more complexity
- Better compilers made complex instructions less important.
- Use of numerous micro-architectural innovations, e.g., pipelining, caches and buffers, made multiple-cycle execution of reg-reg instructions unattractive
- Looking ahead to RISC next time
  - Use chip area to build fast instruction cache of user-visible vertical microinstructions - use software subroutine not hardware microroutines
  - Use simple ISA to enable hardwired pipelined implementation

Modern Usage

- *Microprogramming is far from extinct*

- Played a crucial role in micros of the Eighties

  DEC uVAX, Motorola 68K series, Intel 386 and 486

- Microcode pays an assisting role in most modern micros *(AMD Phenom, Intel Nehalem, Intel Atom, IBM PowerPC)*
  - 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 processors load µcode 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)
  - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252