Architecture of Computer Systems

Lecture 2 - Simple Machine Implementations

Last Time in Lecture 1

- Computer Architecture >> ISAs and RTL
  - CS152 is about interaction of hardware and software, and design of appropriate abstraction layers
- Technology and Applications shape Computer Architecture
  - History provides lessons for the future
- First 130 years of CompArch, from Babbage to IBM 360
  - Move from calculators (no conditionals) to fully programmable machines
  - Rapid change started in WWII (mid-1940s), move from electro-mechanical to pure electronic processors
- Cost of software development becomes a large constraint on architecture (need compatibility)
- 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

IBM 360: Initial Implementations

	Model 30	. . .	*Model 70*
Memory	8K - 64 KB		256K - 512 KB
Datapath	8-bit		64-bit
Circuit Delay	30 nsec/level		5 nsec/level
Local Store	Main Store		Transistor Registers
Control Store	Read only 1μsec		Conventional circuits

IBM 360 instruction set architecture (ISA) completely hid the underlying technological differences between various models.

Milestone: The first true ISA designed as portable hardware-software interface!

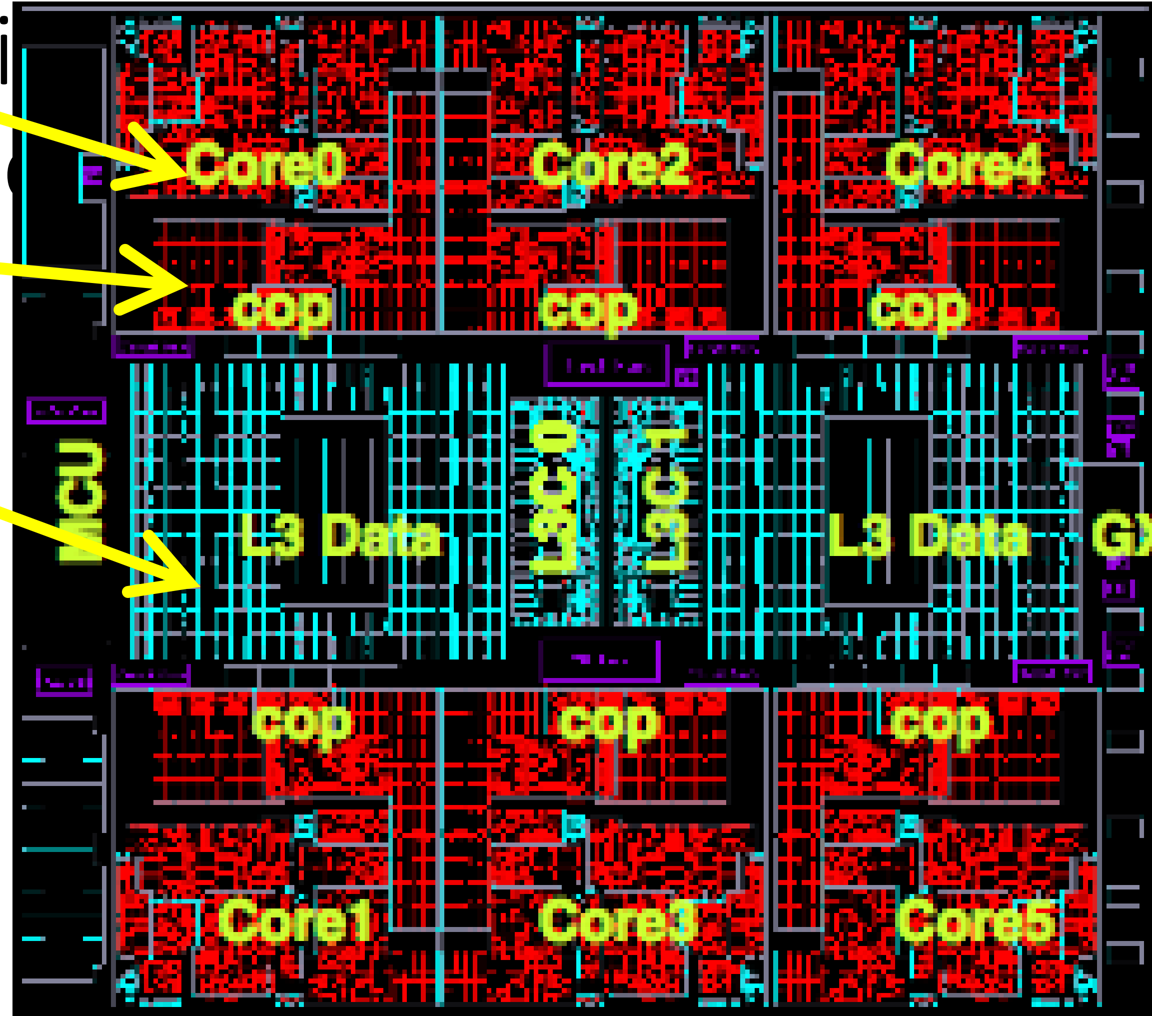
IBM 360 Survives Today: z12 Mainframe Processor

6 Cores @ 5.5 GHz

Special-purpose coprocessors on each core

48MB of Level-3 cache on chip

32nm SOI Technology
2.75 billion transistors
23.7mm x 25.2mm
15 layers of metal
7.68 miles of wiring!
10,000 power pins (!)
1,071 I/O pins



[From IBM HotChips24 presentation, August 28, 2012]

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., the Soviets build code-compatible clones of the IBM360, as did Amdahl after he left IBM.
  - E.g.2., today you can buy AMD or Intel processors that run the x86-64 ISA.
  - E.g.3: many cellphones use the ARM ISA with implementations from many different companies including TI, Qualcomm, Samsung, Marvell, etc.

ISA to Microarchitecture Mapping

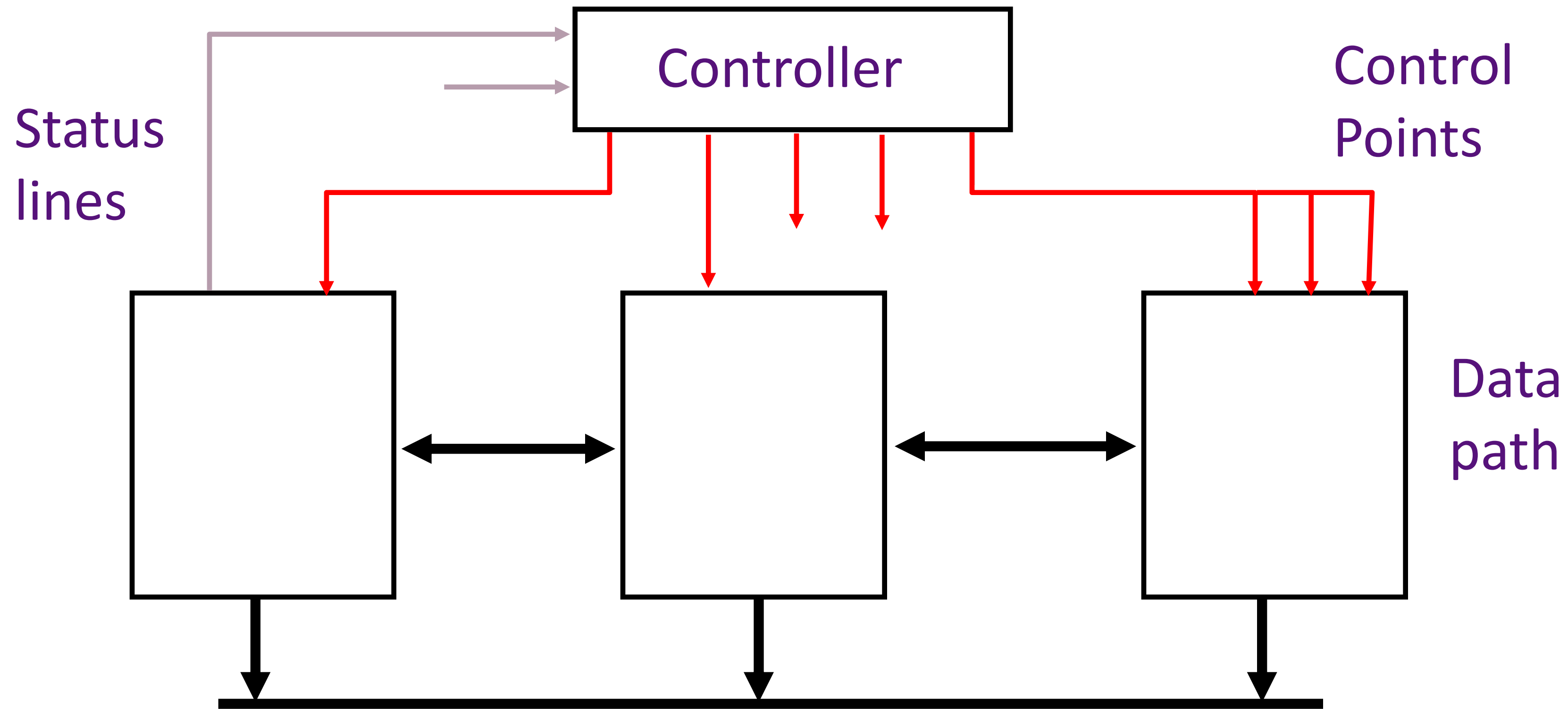
- ISA often designed with particular microarchitectural style in mind, e.g.,
  - Accumulator $\Rightarrow$ hardwired, unpipelined
  - CISC $\Rightarrow$ microcoded
  - RISC $\Rightarrow$ hardwired, pipelined
  - VLIW $\Rightarrow$ fixed-latency in-order parallel pipelines
  - JVM $\Rightarrow$ software interpretation
- But can be implemented with any microarchitectural style
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Simics: Software-interpreted SPARC RISC machine
  - ARM Jazelle: A hardware JVM processor
  - This lecture: a microcoded RISC-V machine

Today, Microprogramming

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

* “CISC”/”RISC” names much newer than style of machines they refer to.

Microarchitecture: *Implementation of an ISA*



Structure: How components are connected.

Static

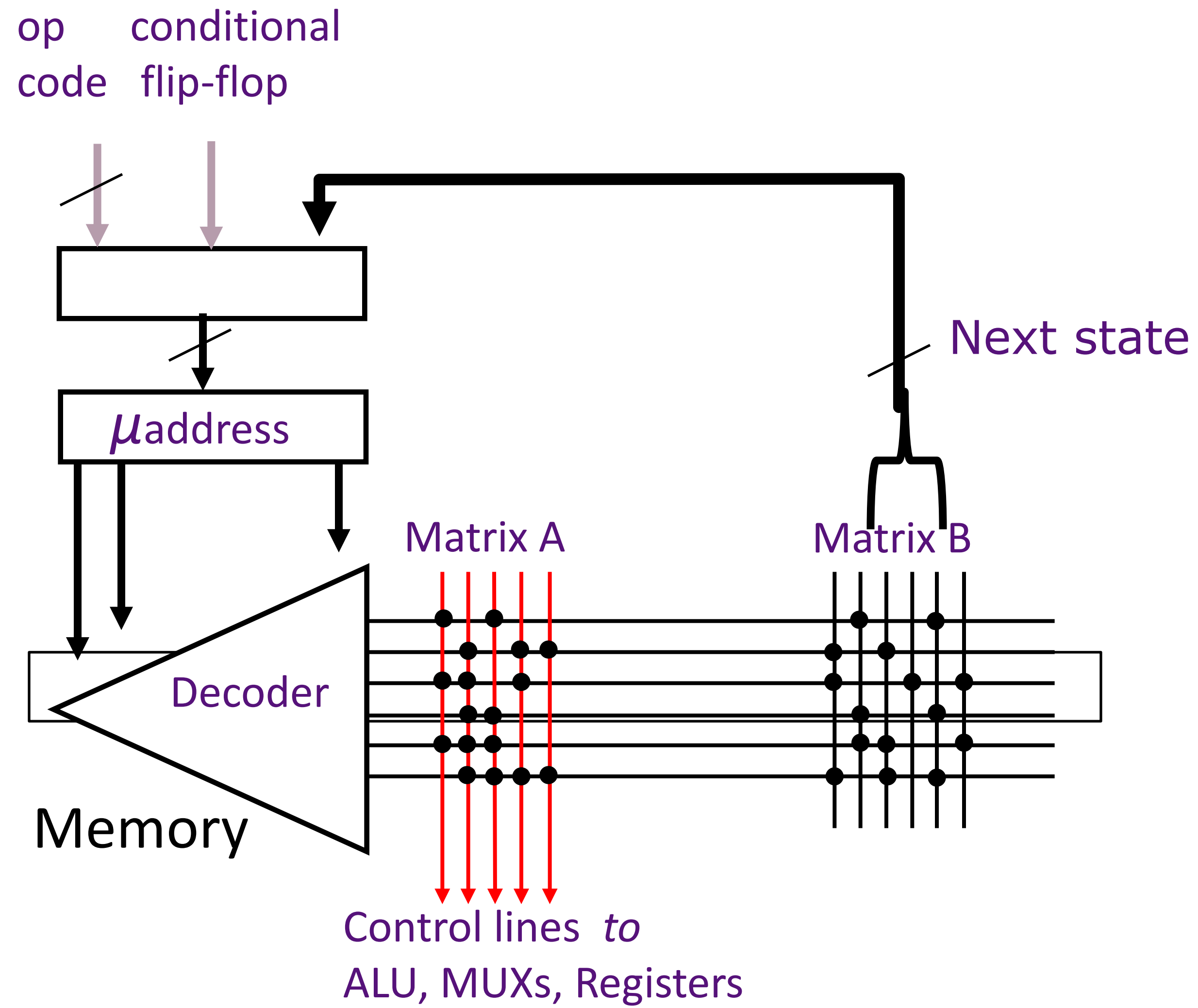
Behavior: How data moves between components

Dynamic

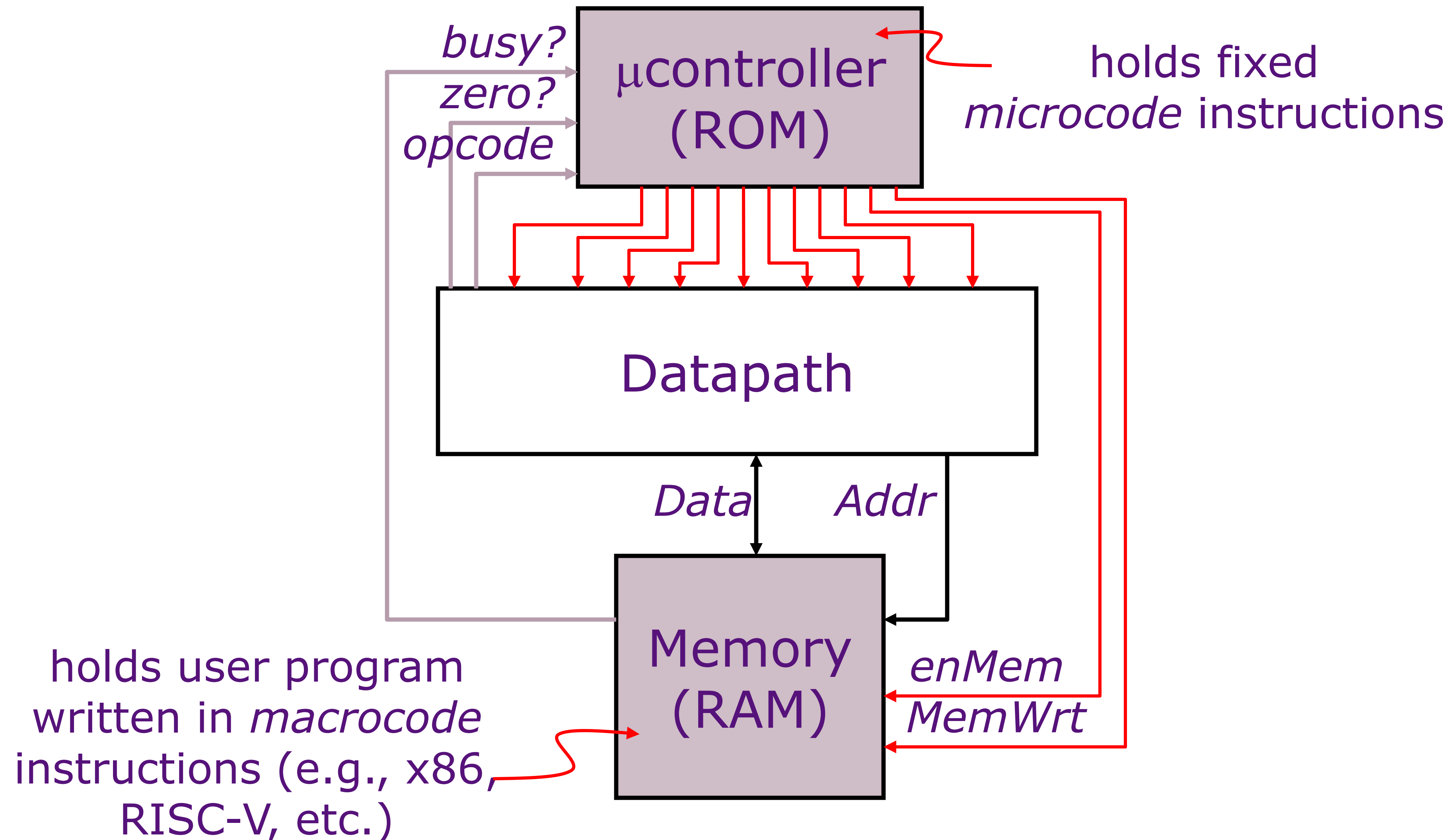
Microcontrol Unit *Maurice Wilkes, 1954*

First used in EDSAC-2, completed 1958

Embed the control logic state table in a memory array



Microcoded Microarchitecture



RISC-V ISA

- New RISC design from UC Berkeley
- Realistic & complete ISA, but open & small
- Not over-architected for a certain implementation style
- Both 32-bit and 64-bit address space variants
  - RV32 and RV64
- Designed for multiprocessing
- Efficient instruction encoding
- Easy to subset/extend for education/research
- Techreport with RISC-V spec available on class website

- We’ll be using 32-bit RISC-V this semester in lectures and labs, very similar to MIPS you saw in CS61C

RV32 Processor State

Program counter (**pc**)

32x32-bit integer registers (**x0-x31**)
• **x0** always contains a 0

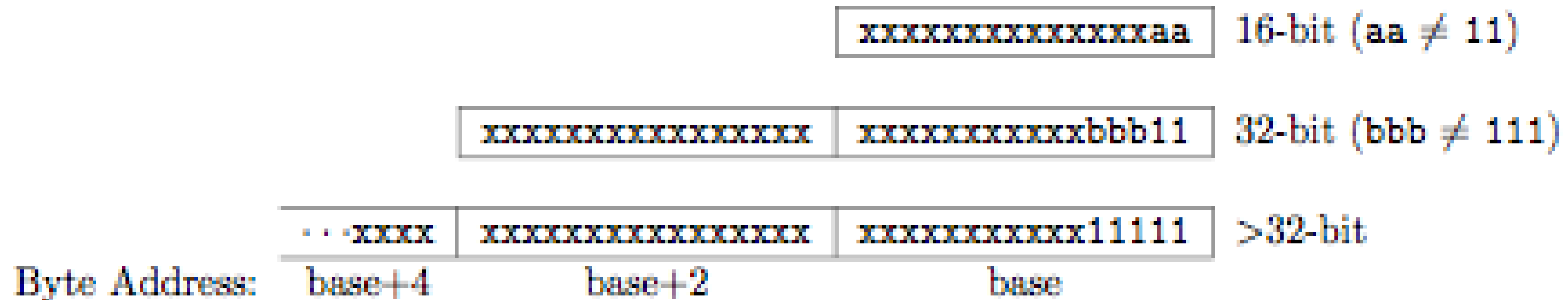
32 floating-point (FP) registers (**f0-f31**)
• each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)

FP status register (**fsr**), used for FP rounding mode & exception reporting

XPRLEN-1 … 0	63 … 0
x0 / zero	f0
x1 / ra	f1
x2	f2
x3	f3
x4	f4
x5	f5
x6	f6
x7	f7
x8	f8
x9	f9
x10	f10
x11	f11
x12	f12
x13	f13
x14	f14
x15	f15
x16	f16
x17	f17
x18	f18
x19	f19
x20	f20
x21	f21
x22	f22
x23	f23
x24	f24
x25	f25
x26	f26
x27	f27
x28	f28
x29	f29
x30	f30
x31	f31
XPRLEN	64

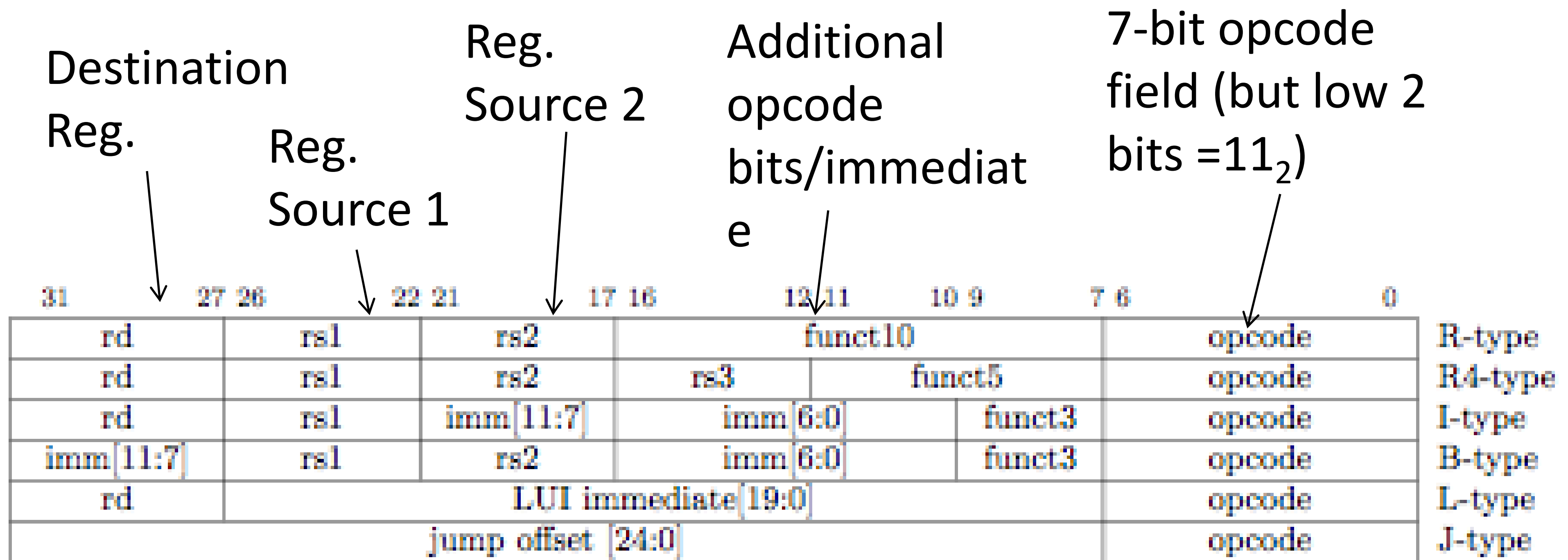
XPRLEN-1 … 0	31 … 0
pc	fsr
XPRLEN	32

RISC-V Instruction Encoding



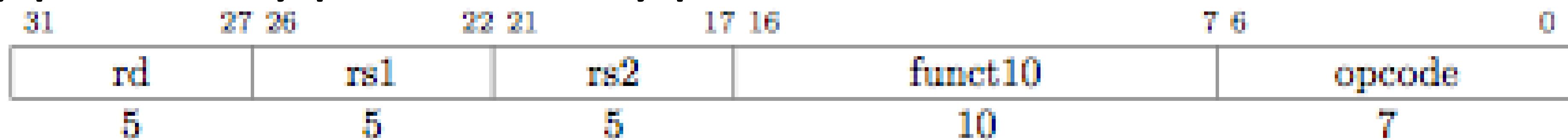
- Can support variable-length instructions.
- Base instruction set (RV32) always has fixed 32-bit instructions lowest two bits = 11_2
- All branches and jumps have targets at 16-bit granularity (even in base ISA where all instructions are fixed 32 bits)

RISC-V Instruction Formats

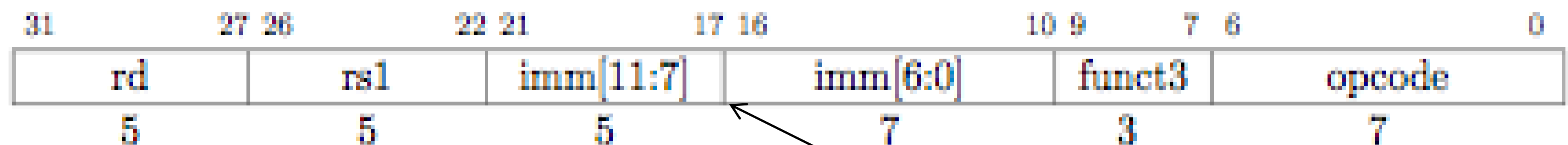


31 27	26 22	21 17	16 12	11 10	9 7	6 0	
rd	rs1	rs2	funct10			opcode	R-type
rd	rs1	rs2	rs3	funct5		opcode	R4-type
rd	rs1	imm[11:7]	imm[6:0]		funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]		funct3	opcode	B-type
rd	LUI immediate[19:0]					opcode	L-type
jump offset [24:0]						opcode	J-type

R-Type/I-Type/R4-Type Formats



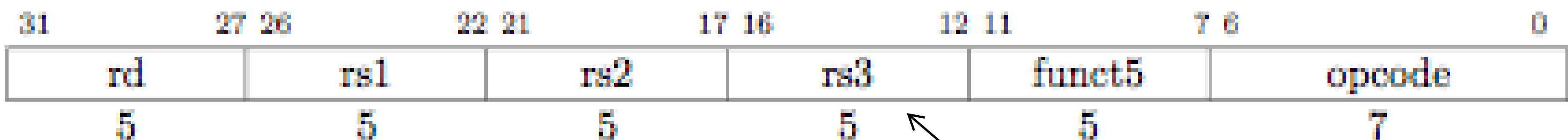
Reg-Reg ALU operations



Reg-Imm ALU operations

12-bit signed immediate

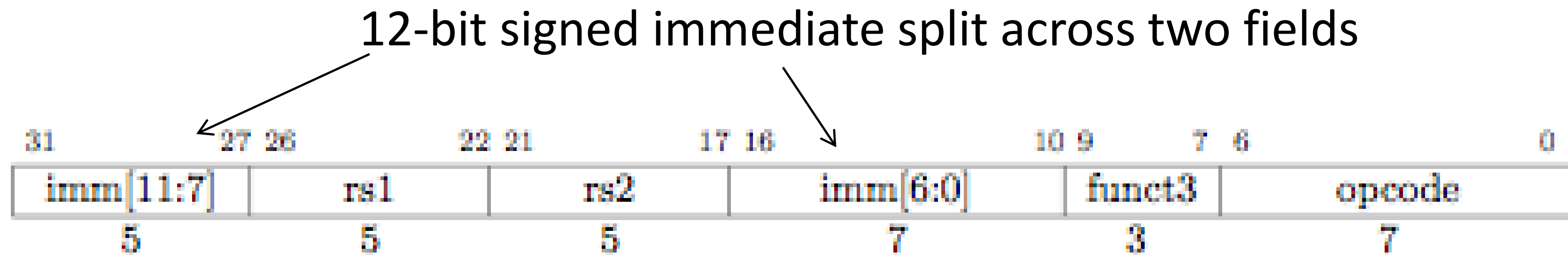
Load instructions, (rs1 + immediate) addressing



Only used for floating-point fused multiply-add

Reg. Source 3

B-Type



Branches, compare two registers, PC+(immediate<<1) target

(Branches do not have delay slot)

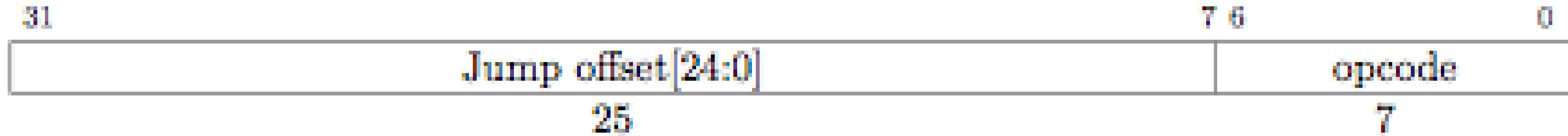
Store instructions, (rs1 + immediate) addressing, rs2 data

L-Type

31 27	26 7	6 0
rd	LUI immediate[19:0]	opcode
5	20	7

Writes 20-bit immediate to top of destination register.
Used to build large immediates.
12-bit immediates are signed, so have to account for sign when building 32-bit immediates in 2-instruction sequence (LUI high-20b, ADDI low-12b)

J-Type



"J" Unconditional jump, PC+offset target
"JAL" Jump and link, also writes PC+4 to **x1**

Offset scaled by 1-bit left shift – can jump to 16-bit instruction boundary (Same for branches)

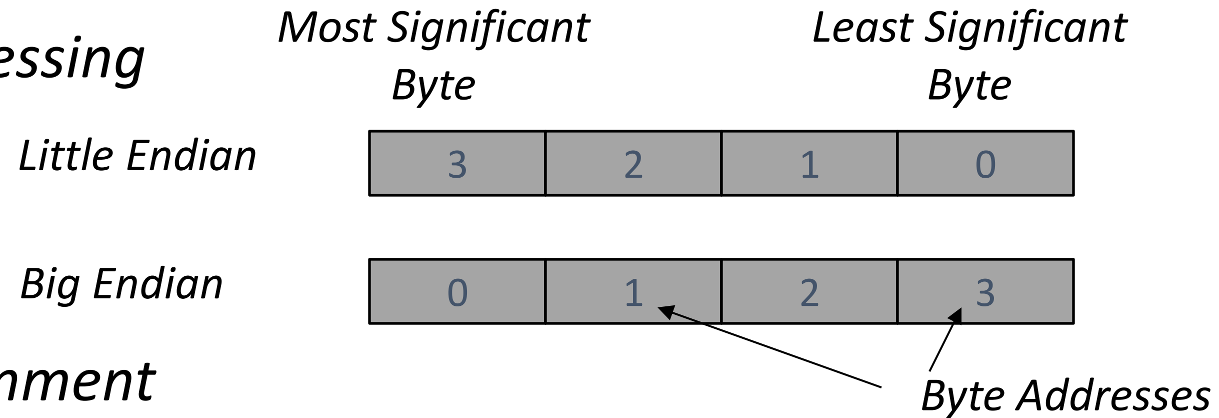
Data Formats and Memory Addresses

Data formats:

8-b Bytes, 16-b Half words, 32-b words and 64-b double words

Some issues

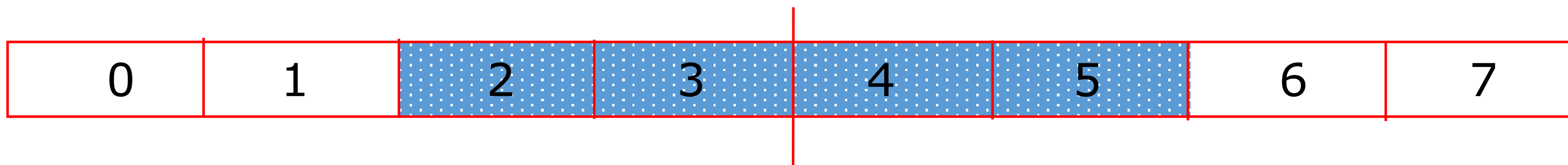
- *Byte addressing*



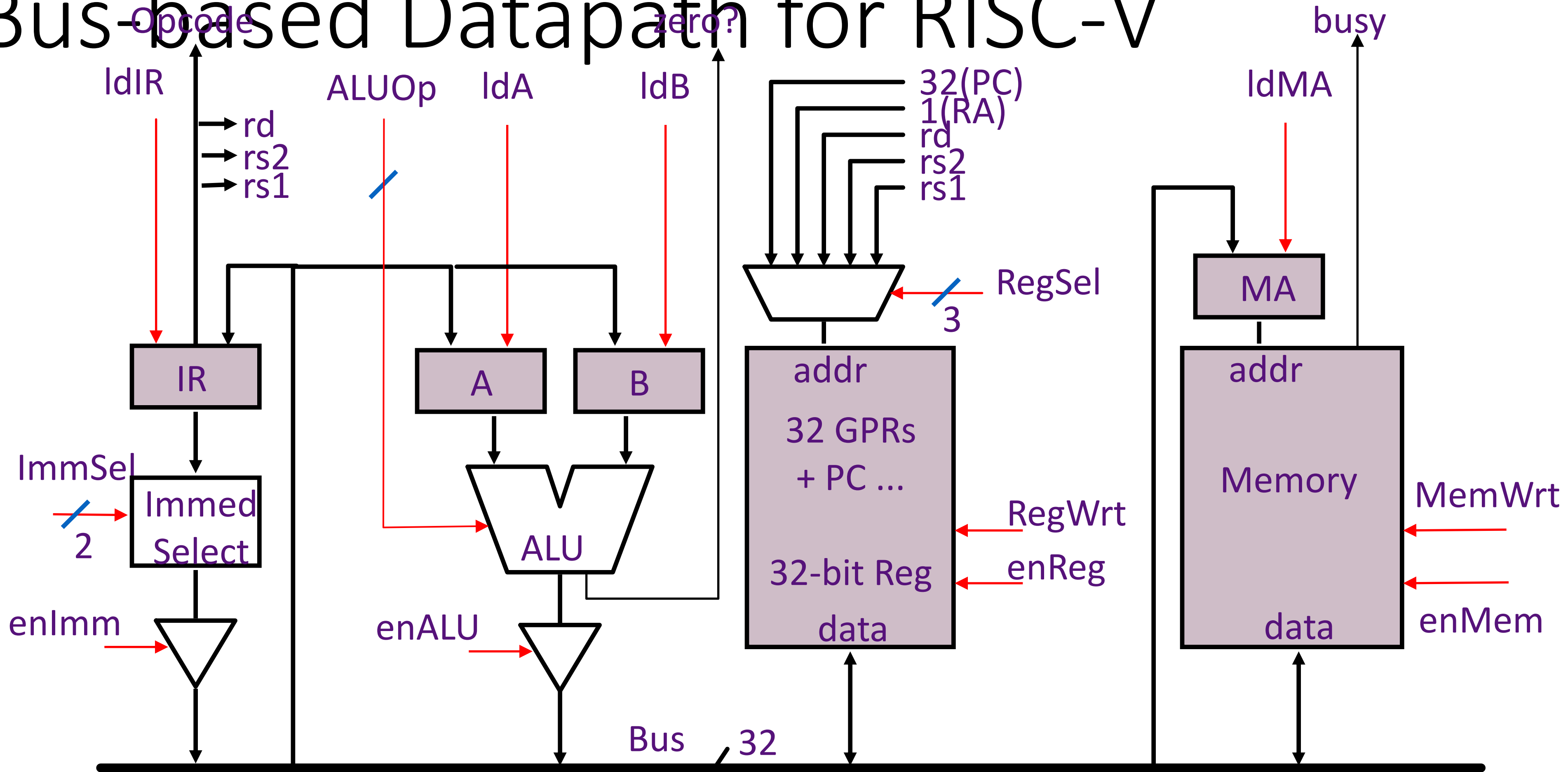
- *Word alignment*

Suppose the memory is organized in 32-bit words.

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



A Bus-based Datapath for RISC-V

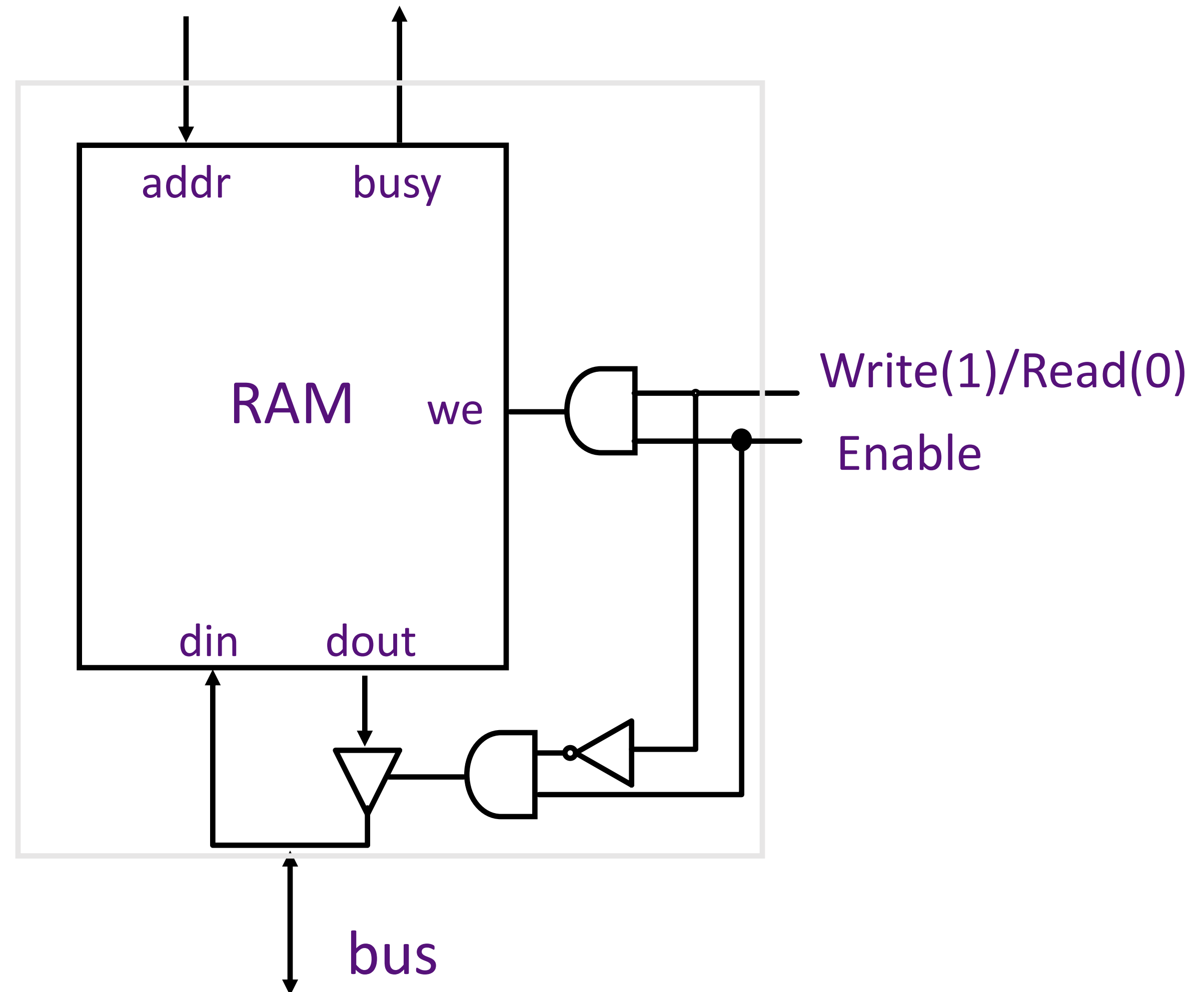


Microinstruction: register to register transfer (17 control signals)

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

B ← Reg[rs2] *means* RegSel = rs2; 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 RISC-V 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, A ← PC
PC ← A + 4
IR ← Memory
dispatch on Opcode

can be treated as a macro

ALU:

A ← Reg[rs1]
B ← Reg[rs2]
Reg[rd] ← func(A,B)
do instruction fetch

ALUi:

A ← Reg[rs1]
B ← Imm *sign extension*
Reg[rd] ← Opcode(A,B)
do instruction fetch

Microprogram Fragments *(cont.)*

LW: A ← Reg[rs1]
B ← Imm
MA ← A + B
Reg[rd] ← Memory
do instruction fetch

J: A ← A - 4 Get original PC back in A
B ← IR
PC ← JumpTarg(A,B)
do instruction fetch

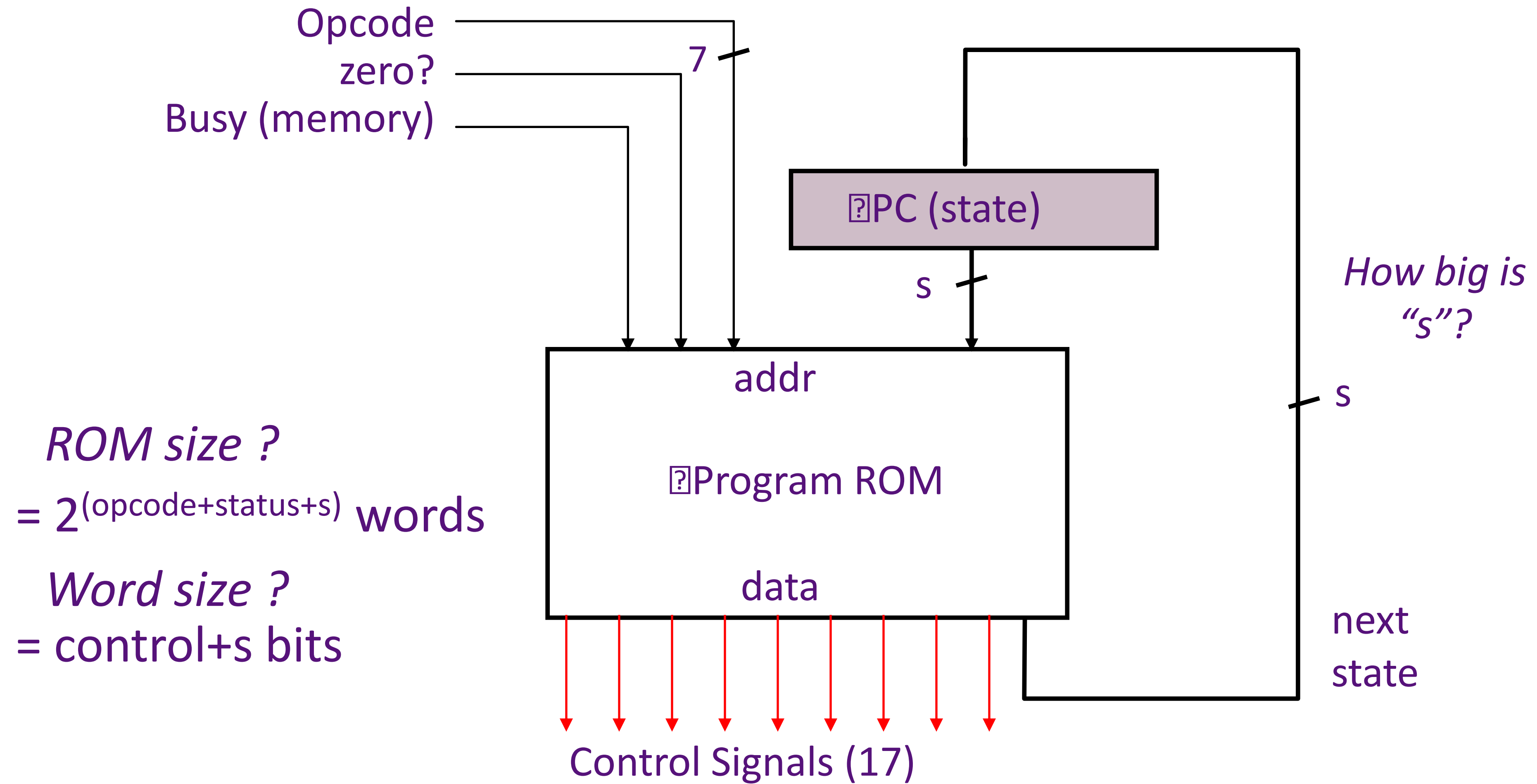
JumpTarg(A,B) =
{A + (B[31:7]<<1)}

beq: A ← Reg[rs1]
B ← Reg[rs2]
If A==B *then go to* bz-taken
do instruction fetch

bz-taken: A ← PC
A ← A - 4 Get original PC back in A
B ← BImm << 1 BImm = IR[31:27,16:10]
PC ← A + B
do instruction fetch

RISC-V Microcontroller: *first attempt*

pure ROM implementation



Microprogram in the ROM *worksheet*

State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA,A ⍰ PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ⍰ Memory	$fetch_2$
$fetch_2$	*	*	*	PC ⍰ A + 4	?
$fetch_2$	ALU	*	*	PC ⍰ A + 4	ALU_0
ALU_0	*	*	*	A ⍰ Reg[rs1]	ALU_1
ALU_1	*	*	*	B ⍰ Reg[rs2]	ALU_2
ALU_2	*	*	*	Reg[rd] ⍰ func(A,B)	$fetch_0$

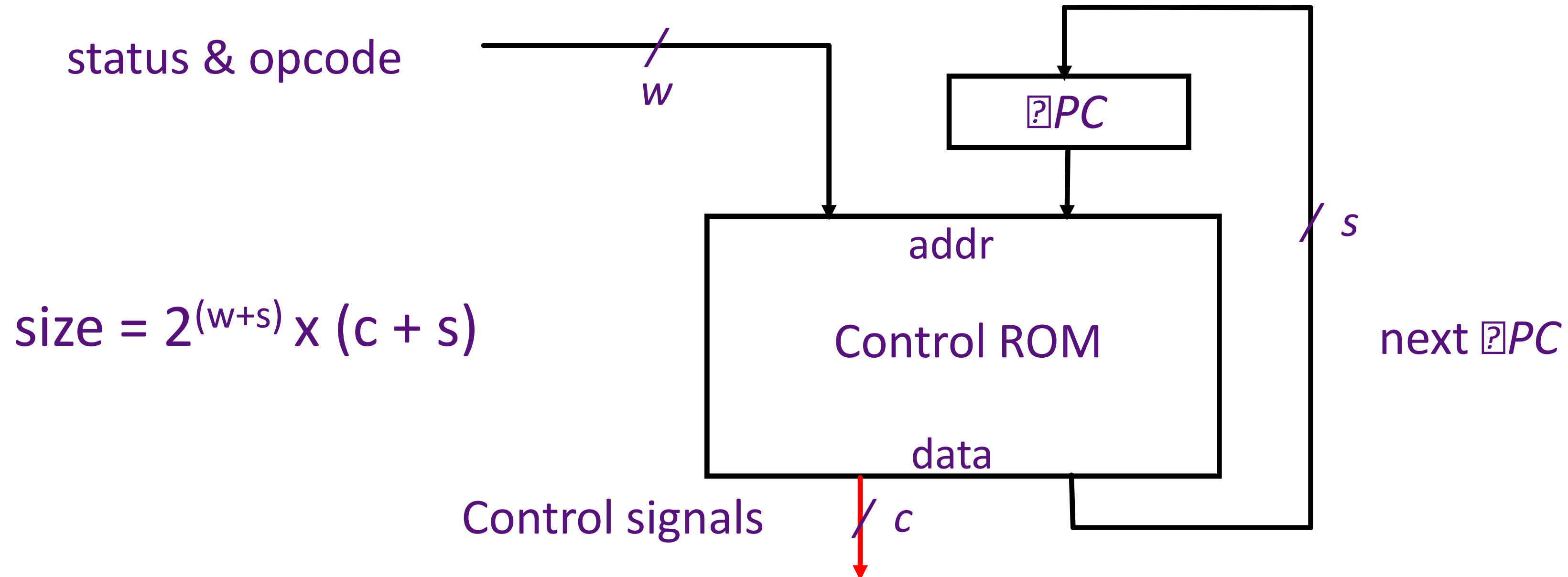
Microprogram in the ROM

State	Op	zero?	busy	Control points	next-state
fetch$_0$	*	*	*	MA,A ← PC	fetch$_1$
fetch$_1$	*	*	yes	....	fetch$_1$
fetch$_1$	*	*	no	IR ← Memory	fetch$_2$
fetch$_2$	ALU	*	*	PC ← A + 4	ALU$_0$
fetch$_2$	ALUi	*	*	PC ← A + 4	ALUi$_0$
fetch$_2$	LW	*	*	PC ← A + 4	LW$_0$
fetch$_2$	SW	*	*	PC ← A + 4	SW$_0$
fetch$_2$	J	*	*	PC ← A + 4	J$_0$
fetch$_2$	JAL	*	*	PC ← A + 4	JAL$_0$
fetch$_2$	JR	*	*	PC ← A + 4	JR$_0$
fetch$_2$	JALR	*	*	PC ← A + 4	JALR$_0$
fetch$_2$	beq	*	*	PC ← A + 4	beq$_0$
...					
ALU$_0$	*	*	*	A ← Reg[rs1]	ALU$_1$
ALU$_1$	*	*	*	B ← Reg[rs2]	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 ← Reg[rs1]	$ALUi_1$
$ALUi_1$	*	*	*	B ← Imm	$ALUi_2$
$ALUi_2$	*	*	*	Reg[rd]← Op(A,B)	$fetch_0$
...					
J_0	*	*	*	A ← A - 4	J_1
J_1	*	*	*	B ← IR	J_2
J_2	*	*	*	PC ← JumpTarg(A,B)	$fetch_0$
...					
beq_0	*	*	*	A ← Reg[rs1]	beq_1
beq_1	*	*	*	B ← Reg[rs2]	beq_2
beq_2	*	yes	*	A ← PC	beq_3
beq_2	*	no	*	....	$fetch_0$
beq_3	*	*	*	A ← A - 4	beq_4
beq_4	*	*	*	B ← BImm	beq_5
beq_5	*	*	*	PC ← A+B	$fetch_0$
...					

Size of Control Store



RISC-V: w = 5+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 = $2^{(5+6)}$ x 23 bits ≈ 24 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*

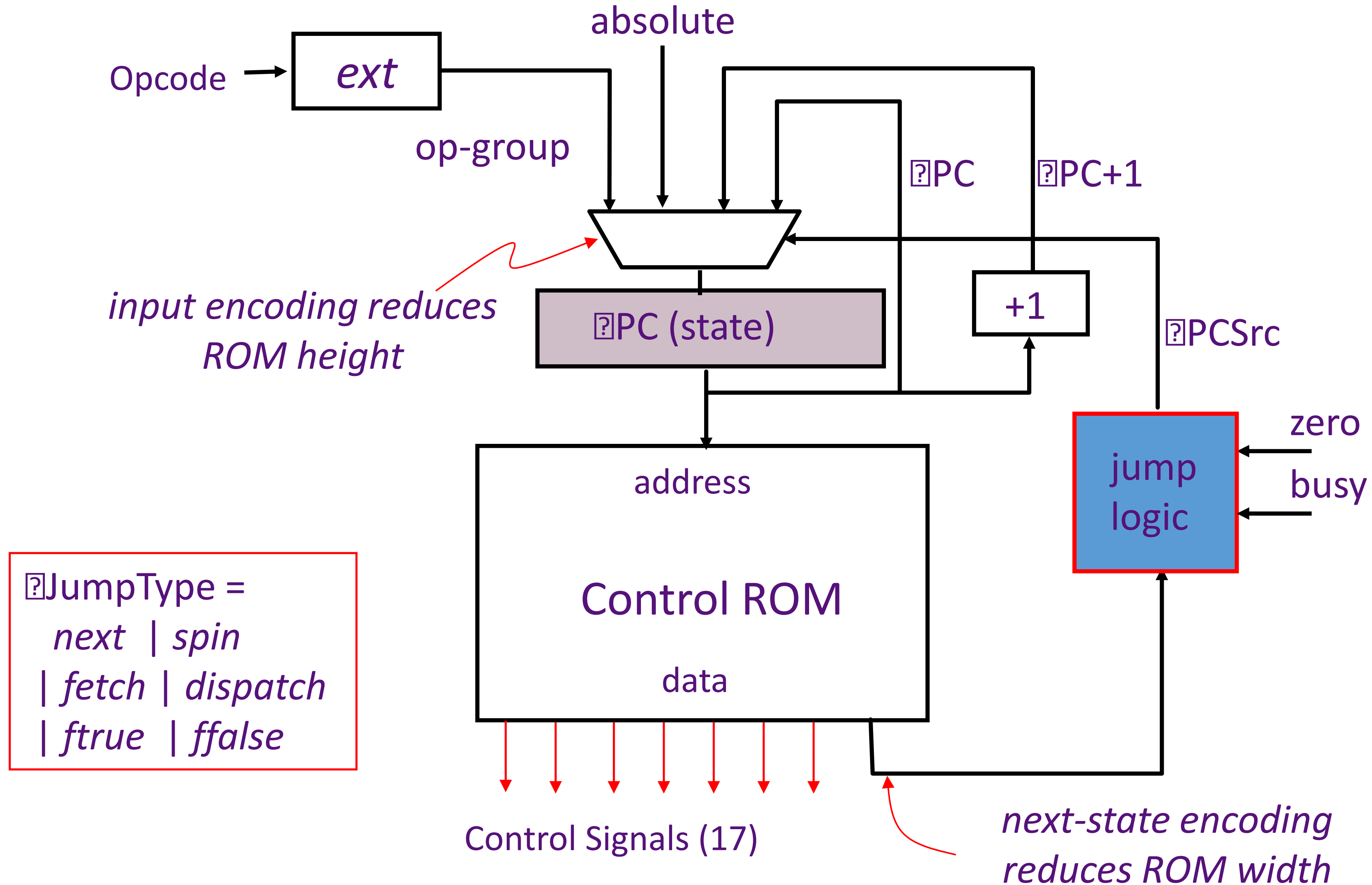
    find common sequences of actions
  - *condense input status bits*

    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)*

RISC-V Controller V2



Jump Logic

□PCSrc = *Case* □JumpTypes

next□ □PC+1

spin□ if (busy) then □PC else □PC+1

fetch □ absolute

dispatch □ op-group

ftrue □ if (zero) then absolute else □PC+1

ffalse □ if (zero) then □PC+1 else absolute

Instruction Fetch & ALU: *RISC-V Controller-2*

State	Control points	next-state
$fetch_0$	MA,A ← PC	next
$fetch_1$	IR ← Memory	spin
$fetch_2$	PC ← A + 4	dispatch
...		
ALU_0	A ← Reg[rs1]	next
ALU_1	B ← Reg[rs2]	next
ALU_2	Reg[rd]←func(A,B)	fetch
$ALUi_0$	A ← Reg[rs1]	next
$ALUi_1$	B ← Imm	next
$ALUi_2$	Reg[rd]← Op(A,B)	fetch

Load & Store: *RISC-V Controller-2*

State	Control points	next-state
LW_0	A ← Reg[rs1]	next
LW_1	B ← Imm	next
LW_2	MA ← A+B	next
LW_3	Reg[rd] ← Memory	spin
LW_4		fetch
SW_0	A ← Reg[rs1]	next
SW_1	B ← BImm	next
SW_2	MA ← A+B	next
SW_3	Memory ← Reg[rs2]	spin
SW_4		fetch

Branches: *RISC-V-Controller-2*

State	Control points	next-state
beq_0	A ← Reg[rs1]	next
beq_1	B ← Reg[rs2]	next
beq_2	A ← PC	ffalse
beq_3	A ← A- 4	next
beq_3	B ← BImm<<1	next
beq_4	PC ← A+B	fetch

Jumps: *RISC-V-Controller-2*

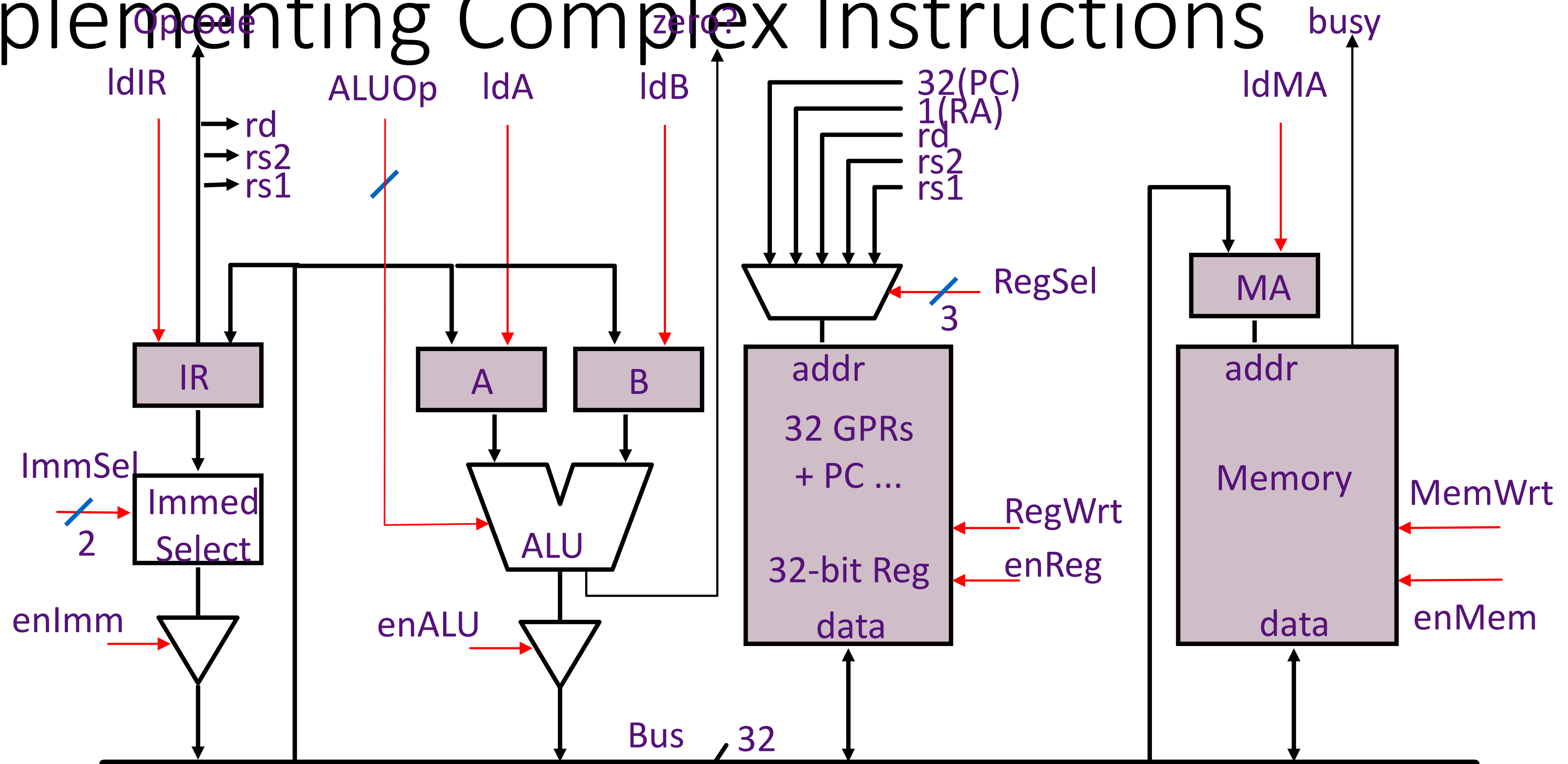
State	Control points	next-state
J_0	A ← A-4	next
J_1	B ← IR	next
J_2	PC ← JumpTarg(A,B)	fetch
JR_0	A ← Reg[rs1]	next
JR_1	PC ← A	fetch
JAL_0	A ← PC	next
JAL_1	Reg[1] ← A	next
JAL_2	A ← A-4	next
JAL_3	B ← IR	next
JAL_4	PC ← JumpTarg(A,B)	fetch

VA

```
; P1WFUD.! ][600,1205]      MICRO2  1F(12)    26-May-81  14:58:1      VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122      Page  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[(rs1)] op (rs2) *Reg-Memory-src ALU op*

M[(rd)] ← (rs1) op (rs2) *Reg-Memory-dst ALU op*

M[(rd)] ← M[(rs1)] op M[(rs2)] *Mem-Mem ALU op*

Mem-Mem ALU Instructions:

RISC-V-Controller-2

Mem-Mem ALU op M[(rd)] ← M[(rs1)] op M[(rs2)]

$ALUMM_0$	MA ← Reg[rs1]	next
$ALUMM_1$	A ← Memory	spin
$ALUMM_2$	MA ← Reg[rs2]	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 these instructions using a hardwired controller is difficult without datapath modifications

Performance Issues

Microprogrammed control

□ multiple cycles per instruction

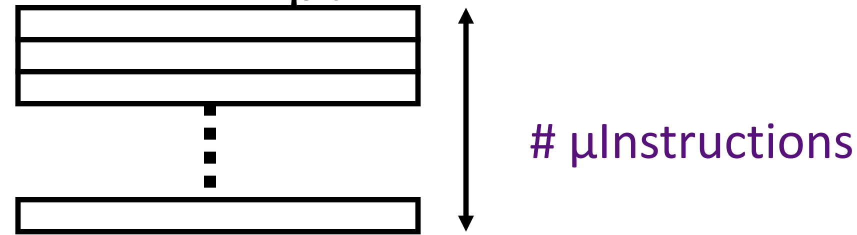
Cycle time ?

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

Suppose $10 * t_{□ROM} < t_{RAM}$

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

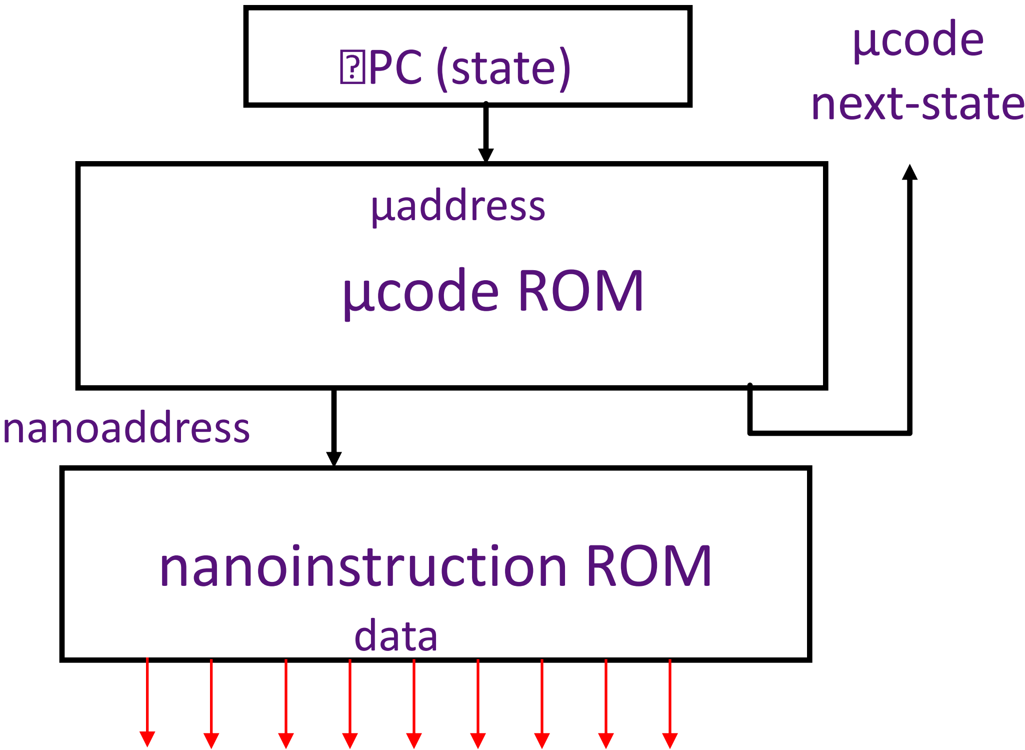
Exploits recurring control signal patterns in µcode, e.g.,

ALU_0 A ← Reg[rs1]

...

ALUi_0 A ← Reg[rs1]

...



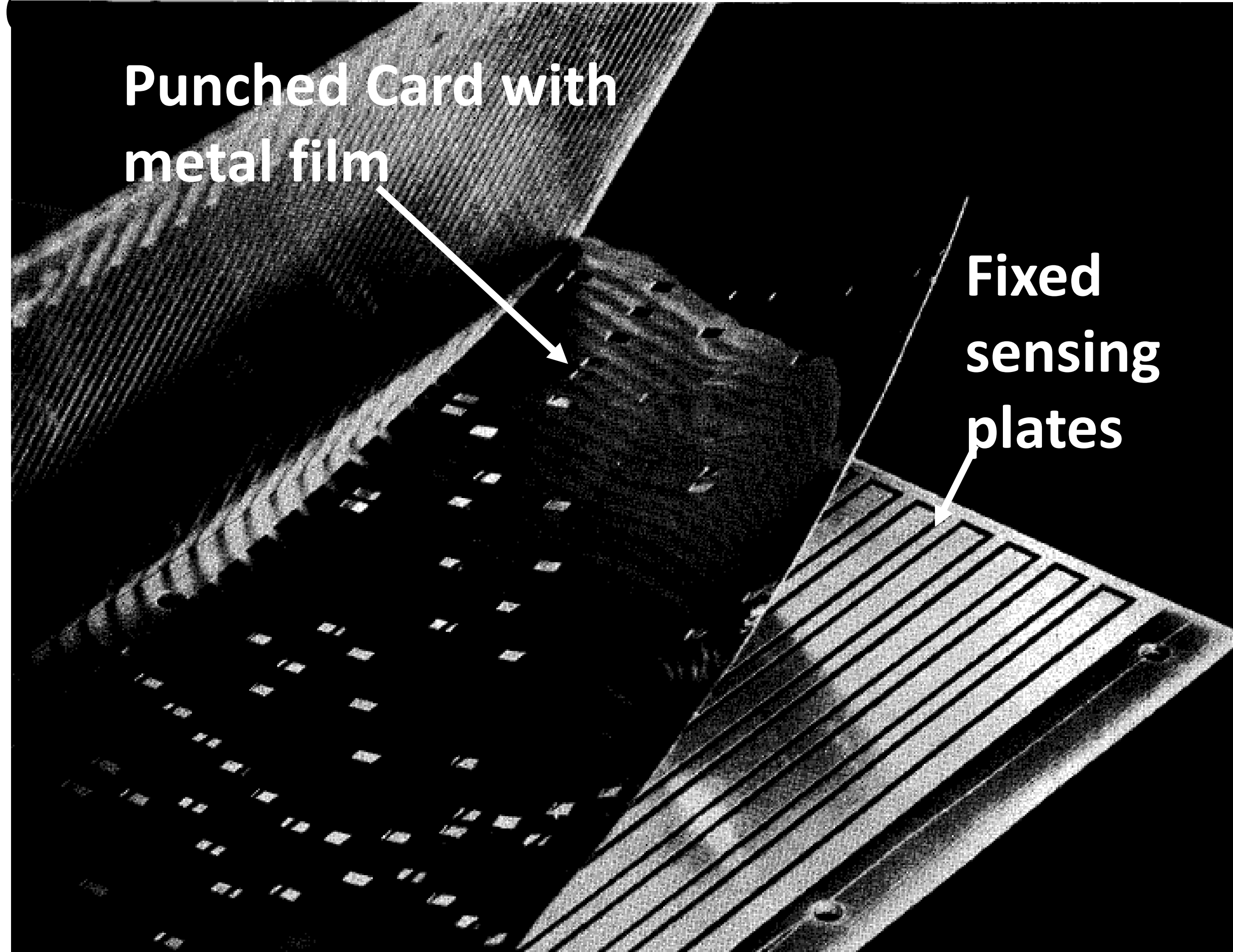
- MC68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

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 is far from extinct

- Played a crucial role in micros of the Eighties
  - DEC uVAX, Motorola 68K series, Intel 286/386
- Plays an assisting role in most modern micros
  - e.g., AMD Bulldozer, Intel Ivy Bridge, Intel Atom, IBM PowerPC, …
  - Most instructions executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke microcode
- 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