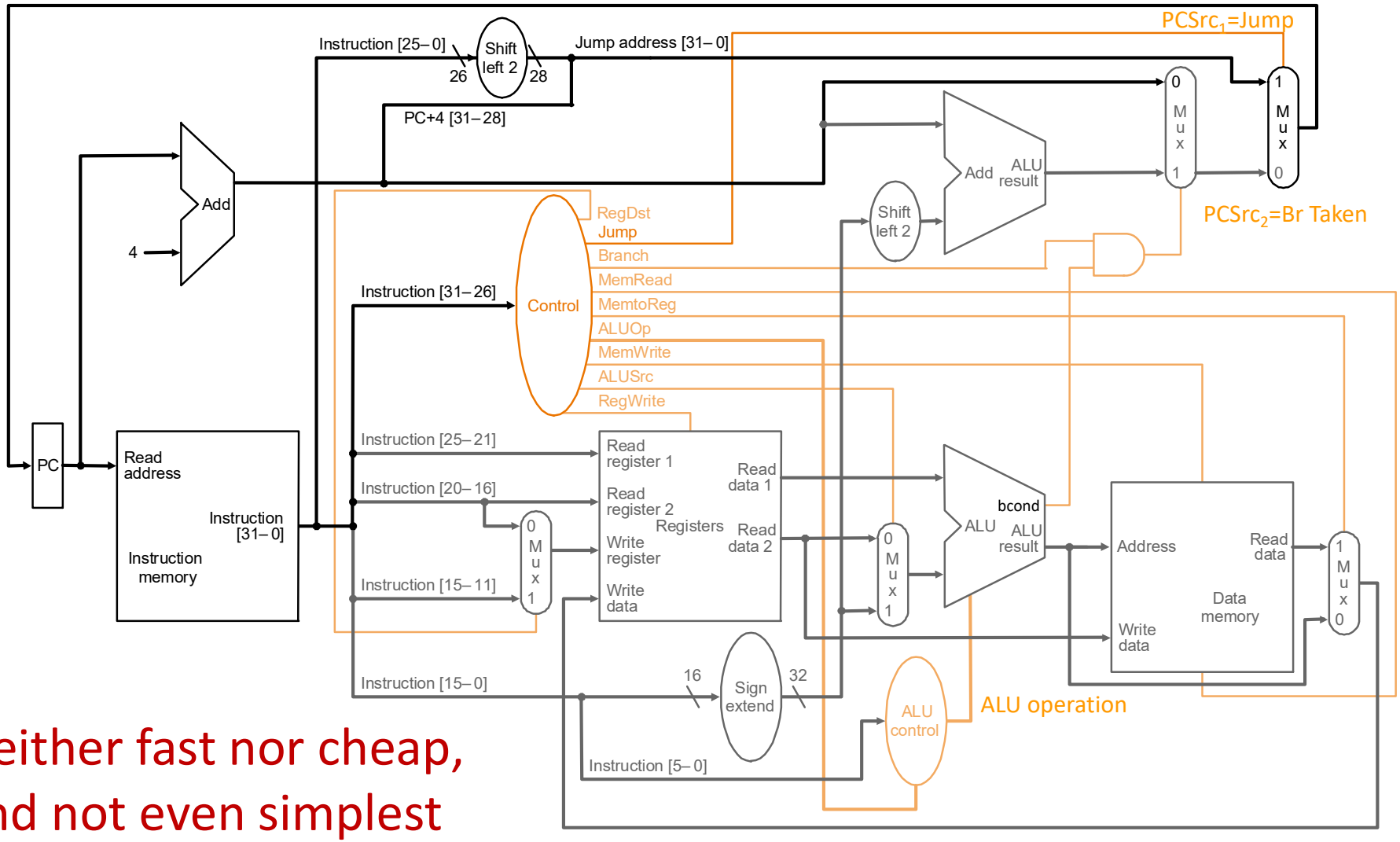
18-447 Lecture 6: Microprogrammed Multi-Cycle Implementation

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today
  - understand why VAX was possible and reasonable
- Notices
  - HW1, past due (see Handout #6: HW 1 solutions)
  - Lab 1, Part B, due this week
  - HW2, due Mon 2/21 (Handout #5: HW 2)
- Readings
  - P&H Appendix C
  - Start reading the rest of P&H Ch 4

"Single-Cycle" Datapath: Is it any good?



Neither fast nor cheap, and not even simplest

Go Fast(er)!!

Iron Law of Processor Performance

- time/program = (inst/program) (cyc/inst) (time/cyc)

1/IPC 1/MIPS 1/GHz

note workload dependence

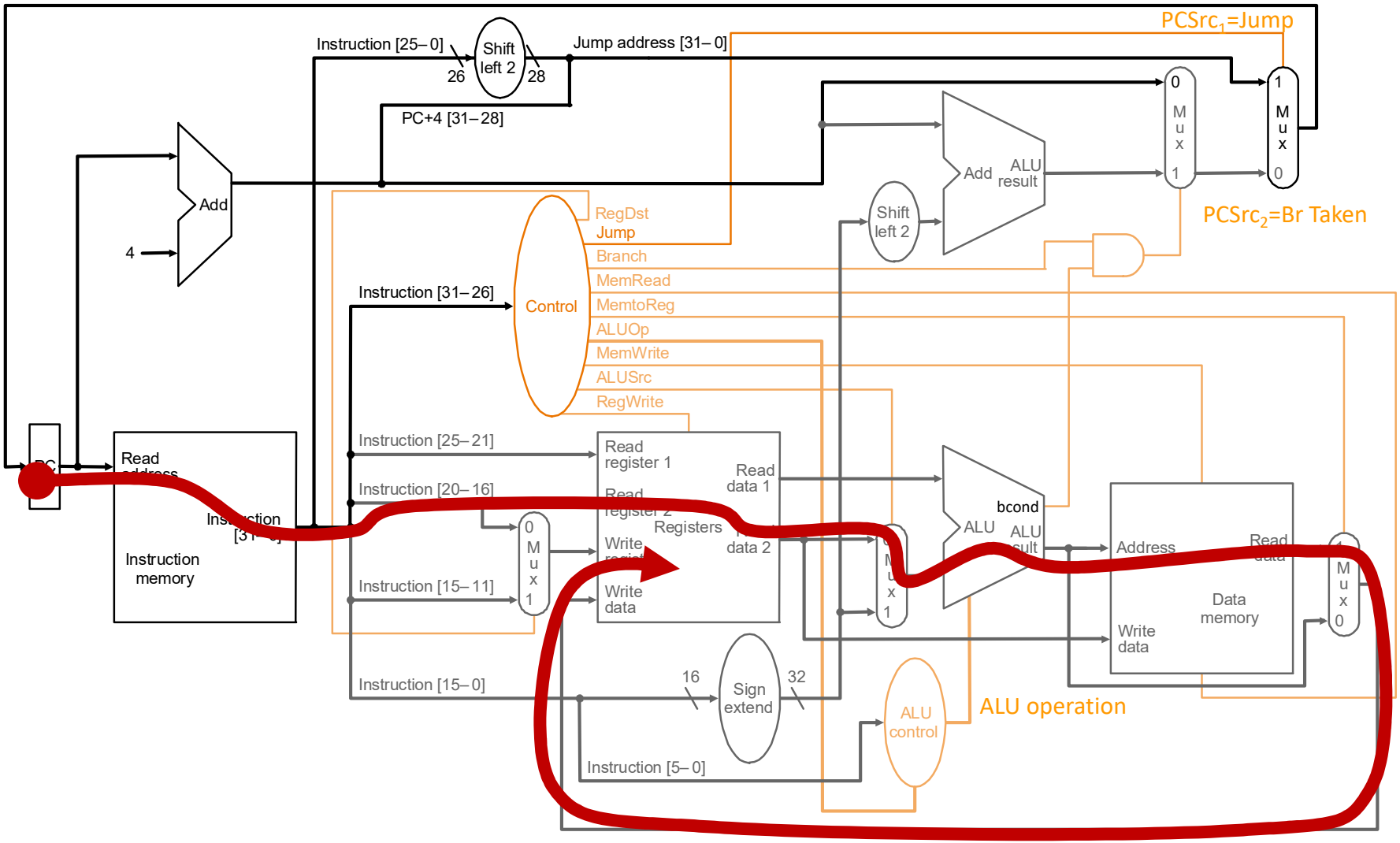
- Contributing factors
  - time/cyc: architecture and implementation
  - cyc/inst: architecture, implementation, instruction mix
  - inst/program: architecture, nature and quality of prgm

- **Note**: cyc/inst is a workload average

potentially large instantaneous variations due to instruction type and sequence

Recall

Worst-Case Critical Path



Single-Cycle Datapath Analysis

- Assume (numbers from P&H)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Bxx	200	50	100			350
JALR	200	50	100		~~50~~	350
JAL	200		100		~~50~~	300

Single-Cycle Implementations

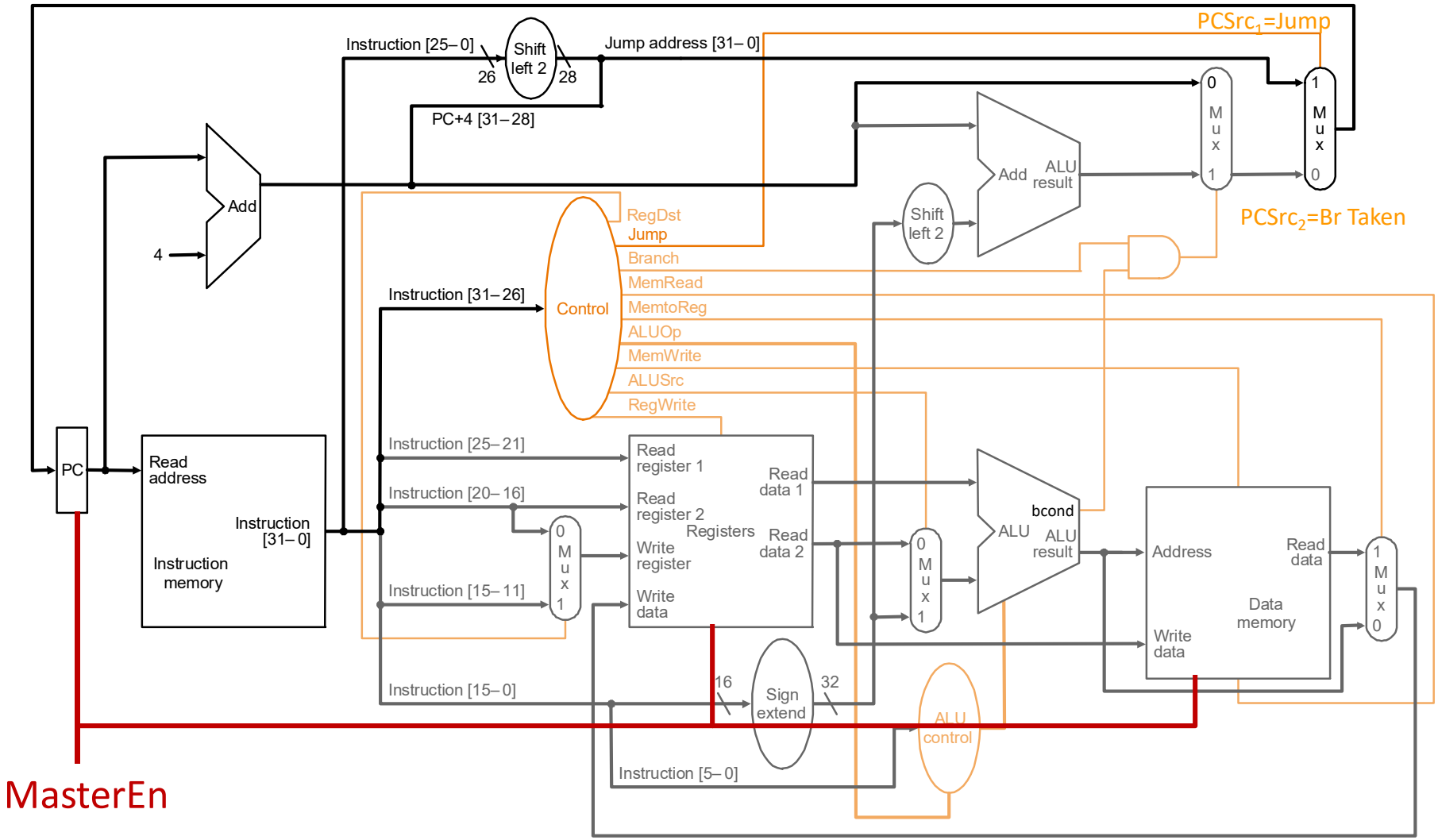
- Good match for the sequential and atomic semantics of ISAs
  - instantiate programmer-visible state one-for-one
  - map instructions to combinational next-state logic
- But, contrived and inefficient
  1. all instructions run as slow as slowest instruction
  2. must provide worst-case combinational resource in parallel as required by any one instruction
  3. what about CISC ISAs? polyf?

Not the fastest, cheapest or even the simplest way

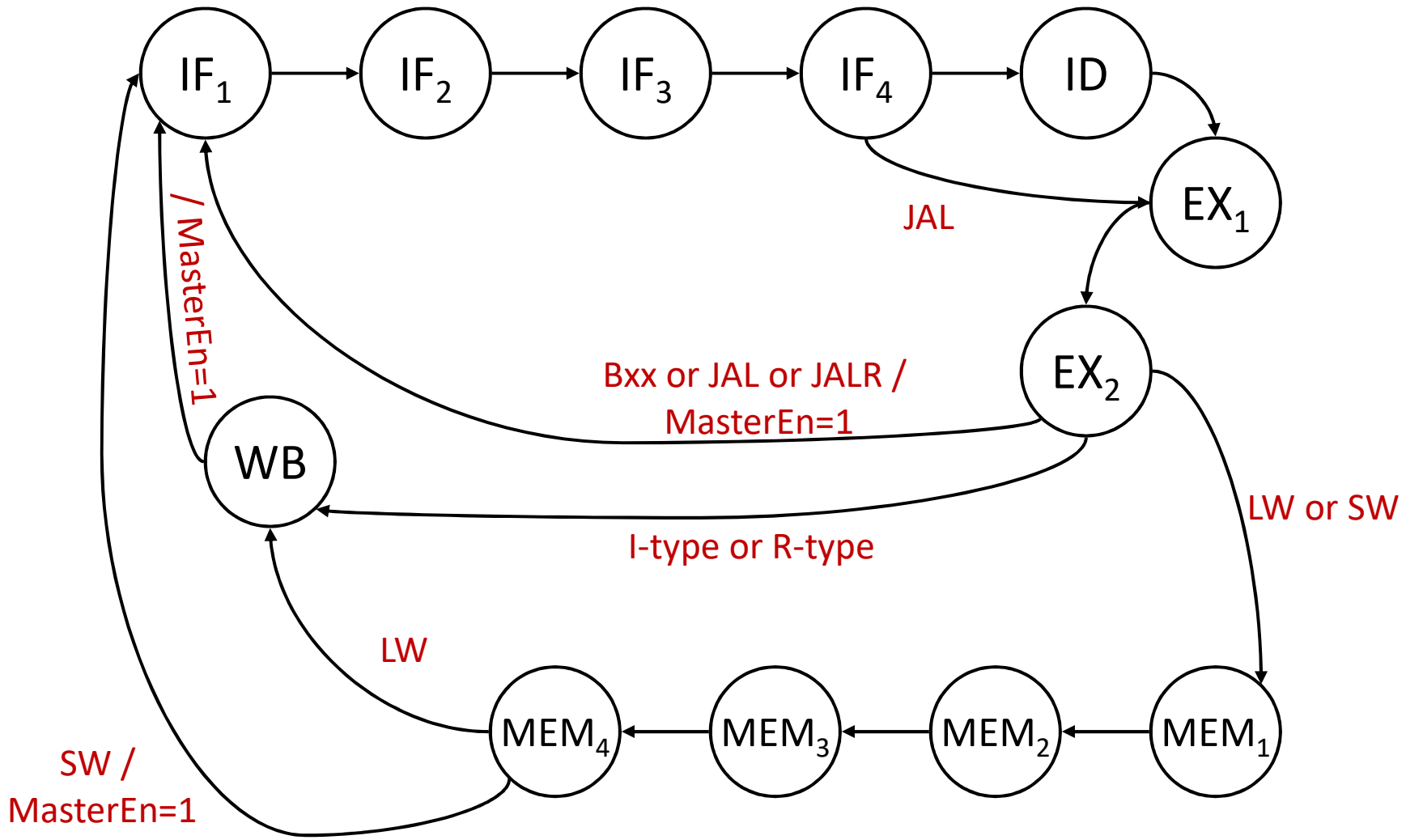
Multi-cycle Implementation: Ver 1.0

- Each instruction type take only as much time as needed
  - run a 50 psec clock
  - each instruction type take as many 50-psec clock cycles as needed
- Add "MasterEnable" signal so architectural state ignores clock edges until after enough time
  - an instruction's effect is still purely combinational from state to state
  - all other control signal unaffected

Multi-Cycle Datapath: Ver 1.0



Sequential Control: Ver 1.0



Performance Analysis

- Iron Law:

  time/program = (inst/program) (cyc/inst) (time/cyc)

- For same ISA, inst/program is the same; okay to compare

$$\text{MIPS} = \text{IPC} \times f_{\text{clk in MHz}}$$

frequency in MHz

million instructions per second

instructions per cycle

Performance Analysis

- Single-Cycle Implementation

  $1 \times 1{,}667\text{MHz} = 1667$ MIPS

- Multi-Cycle Implementation

  $\text{IPC}_{avg} \times 20{,}000$ MHz = 2178 MIPS

  what is $\text{IPC}_{average}$?

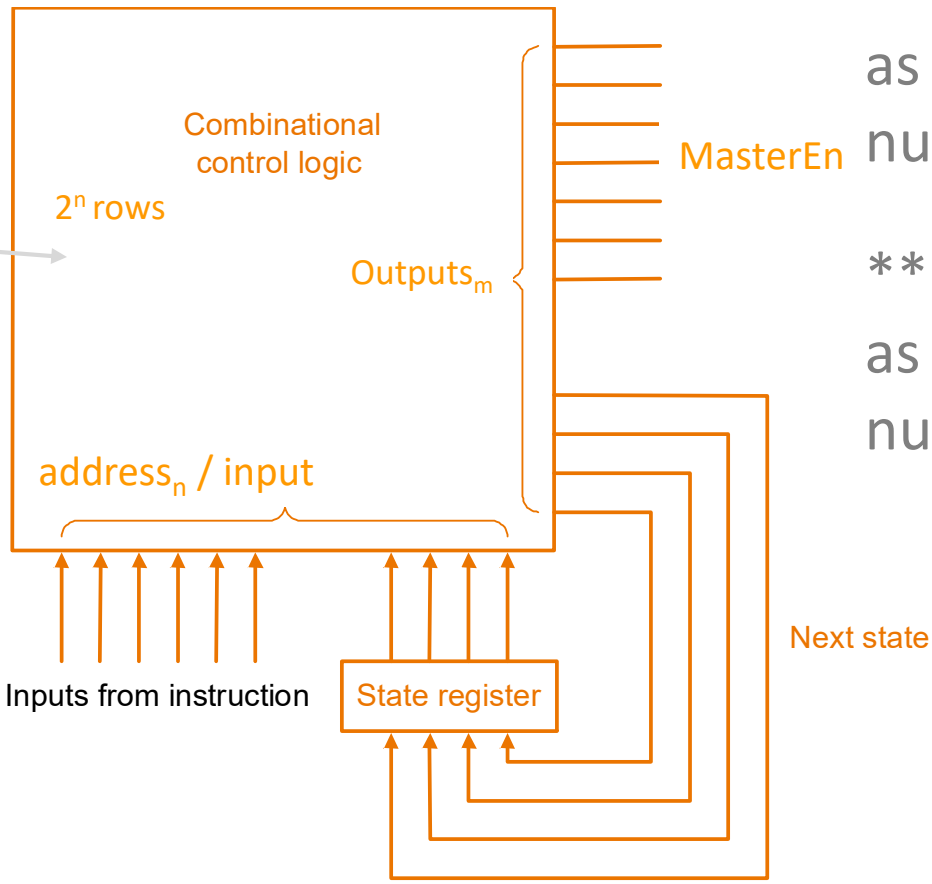
- Assume: 25% LW, 15% SW, 40% ALU, 13.3% Branch, 6.7% Jumps [Agerwala and Cocke, 1987]
  - weighted arithmetic mean of CPI $\Rightarrow$ 9.18
  - weighted harmonic mean of IPC $\Rightarrow$ 0.109
  - ~~weighted arithmetic mean of IPC $\Rightarrow$ 0.115~~

$\text{MIPS} = \text{IPC} \times f_{clk}$

Microsequencer: Ver 1.0

- ROM as a combinational logic lookup table

literally holds the truth table



** ROM size grows as $O(2^n)$ as the number of inputs

** ROM size grows as $O(m)$ as the number of outputs

Microcoding: Ver 0

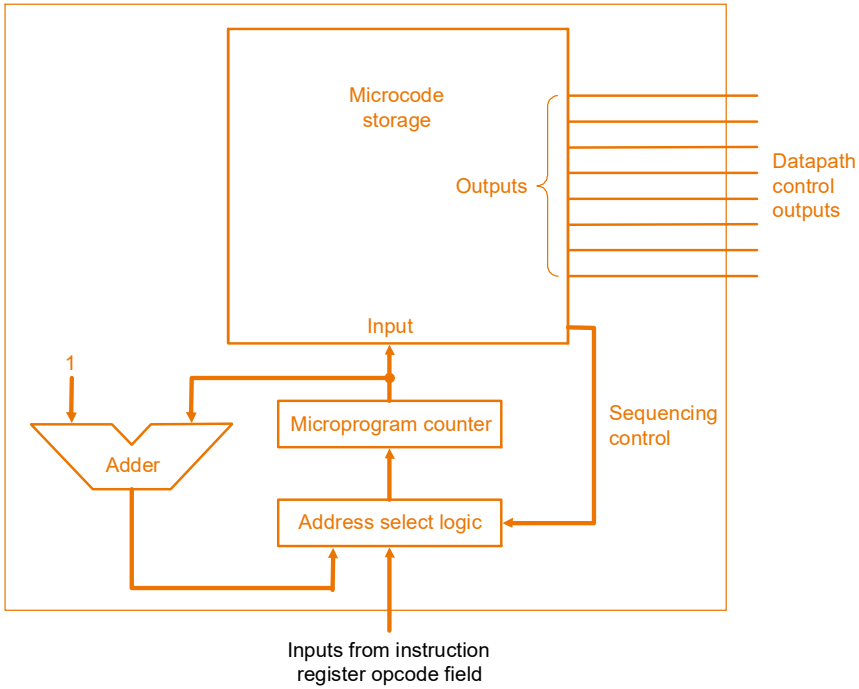
(note: this is only about counting clock ticks)

state label	cntrl flow	conditional targets					
		R/I-type	LW	SW	Bxx	JALR	JAL
IF_1	**next**	-	-	-	-	-	-
IF_2	**next**	-	-	-	-	-	-
IF_3	**next**	-	-	-	-	-	-
IF_4	**goto**	**ID**	**ID**	**ID**	**ID**	**ID**	**EX_1**
ID	**next**	-	-	-	-	-	-
EX_1	**next**	-	-	-	-	-	-
EX_2	**goto**	**WB**	**MEM_1**	**MEM_1**	**IF_1**	**IF_1**	**IF_1**
MEM_1	**next**	-	-	-	-	-	-
MEM_2	**next**	-	-	-	-	-	-
MEM_3	**next**	-	-	-	-	-	-
MEM_4	**goto**	-	**WB**	**IF_1**	-	-	-
WB	**goto**	**IF_1**	**IF_1**	-	-	-	-
CPI		**8**	**12**	**11**	**7**	**7**	**6**

A systematic approach to FSM sequencing/control

Microcontroller/Microsequencer

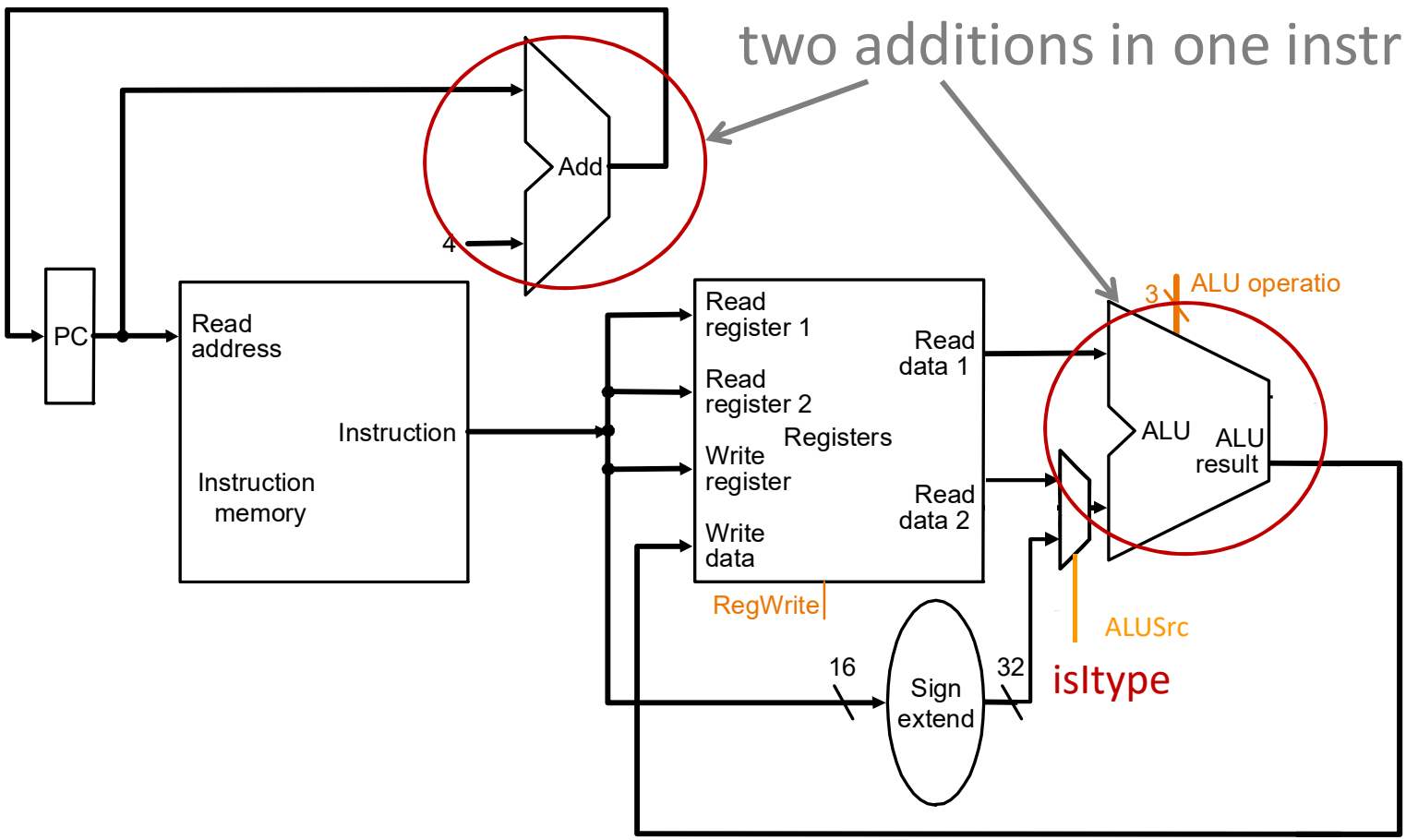
- A stripped-down "processor" for sequencing and control
  - control states are like μPC
  - μPC indexed into a μprogram ROM to select an μinstruction
  - μprogram state and well-formed control-flow support (branch, jump)
  - fields in the μinstruction maps to control signals
- Very elaborate μcontrollers have been built



Go Cheap!!
(And More Capable)

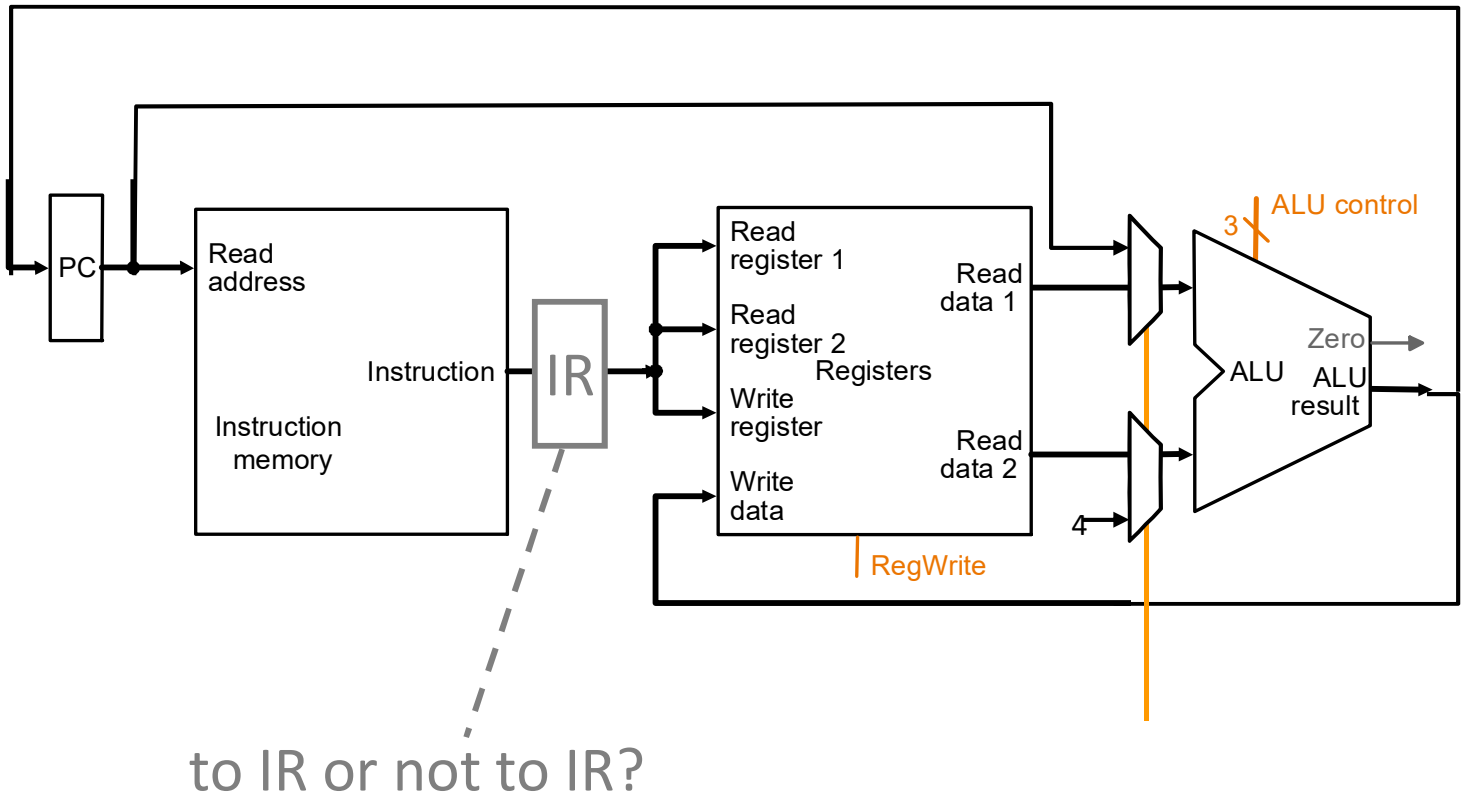
Reducing Datapath by Resource Reuse

How to reuse same adder for two additions in one instruction



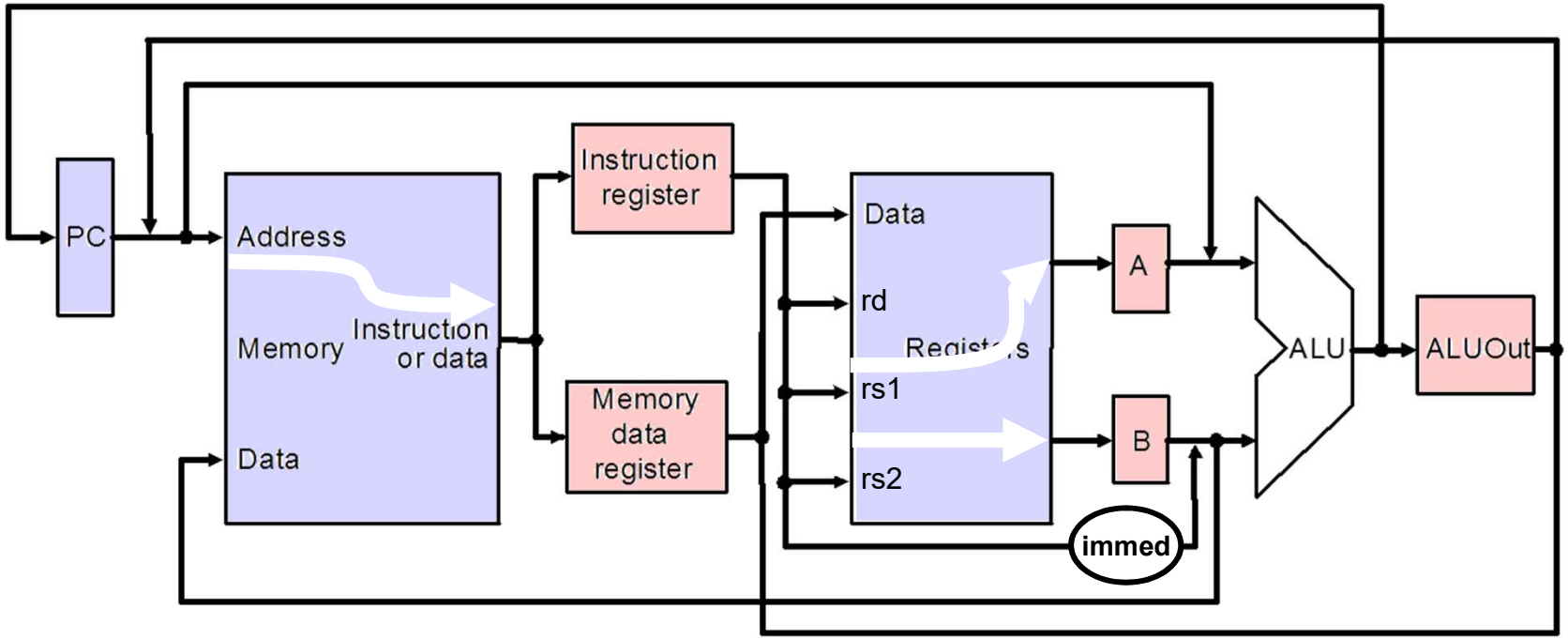
"Single-cycle" reused same adder for different instructions

Reducing Datapath by Sequential Reuse



to IR or not to IR?

Removing Redundancies



- Latch Enables: PC, IR, MDR, A, B, ALUOut, RegWr, MemWr
- Steering: ALUSrc1{RF,PC}, ALUSrc2{RF, immed},
  MAddrSrc{PC, ALUOut}, RFDataSrc{ALUOut, MDR}

Could also reduce down to a single register read-write port!

Synchronous Register Transfers

- Synchronous state with latch enables
  - PC, IR, RF, MEM, A, B, ALUOut, MDR
- One can enumerate all possible "register transfers"
- For example starting from PC
  - IR $\leftarrow$ MEM[PC]
  - MDR $\leftarrow$ MEM[PC]
  - PC $\leftarrow$ PC $\oplus$ 4
  - PC $\leftarrow$ PC $\oplus$ B
  - PC $\leftarrow$ PC $\oplus$ immediate(IR)
  - ALUOut $\leftarrow$ PC $\oplus$ 4
  - ALUOut $\leftarrow$ PC $\oplus$ immediate(IR)
  - ALUOut $\leftarrow$ PC $\oplus$ B

Not all feasible RTs are meaningful

Useful Register Transfers (by dest)

- PC ← PC + 4
- PC ← PC + $\text{immediate}_{\text{SB-type,U-type}}$(IR)
- PC ← A + $\text{immediate}_{\text{SB-type}}$(IR)
- IR ← MEM[PC]
- A ← RF[rs1(IR)]
- B ← RF[rs2(IR)]
- ALUOut ← A + B
- ALUOut ← A + $\text{immediate}_{\text{I-type,S-type}}$(IR)
- ALUOut ← PC + 4
- MDR ← MEM[ALUOut]
- MEM[ALUOut] ← B
- RF[rd(IR)] ← ALUOut,
- RF[rd(IR)] ← MDR

RT Sequencing: R-Type ALU

- **IF**

  IR ← MEM[PC] step 1
- **ID**

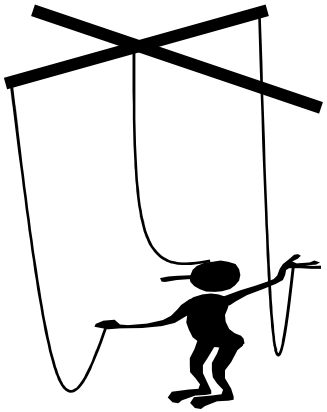
  A ← RF[rs1(IR)] step 2

  B ← RF[rs2(IR)] step 3
- **EX**

  ALUOut ← A + B step 4
- **MEM**
- **WB**

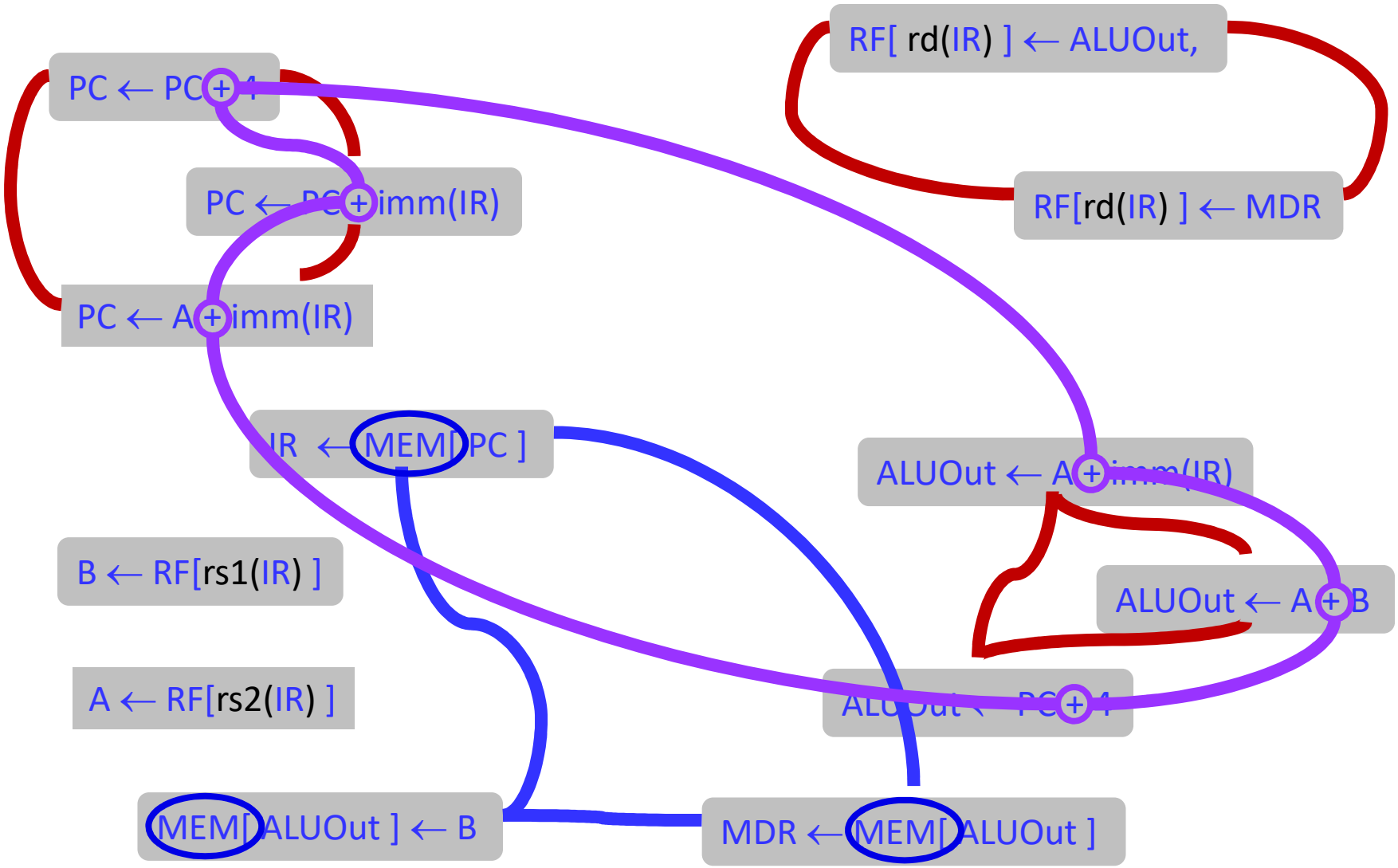
  RF[rd(IR)] ← ALUOut step 5

  PC ← PC+4 step 6



```
if MEM[PC] == ADD rd rs1 rs2
        GPR[rd] ← GPR[rs1] + GPR[rs2]
        PC ← PC + 4
```

RT Datapath Conflicts



Can utilize each resource only once per control step (cycle)

RT Sequencing: R-Type ALU

IR ← MEM[PC] step 1

A ← RF[rs1(IR)]
B ← RF[rs2(IR)] step 2

ALUOut ← A + B
step 3

RF[rd(IR)] ← ALUOut
PC ← PC+4 step 4

RT Sequencing: LW

- **IF**

  IR ← MEM[PC]
- **ID**

  A ← RF[rs1(IR)]

  B ← RF[rs2(IR)]
- **EX**

  ALUOut ← A + $imm_{I\text{-}type}$(IR)
- **MEM**

  MDR ← MEM[ALUOut]
- **WB**

  RF[rd(IR)] ← MDR

  PC ← PC+4

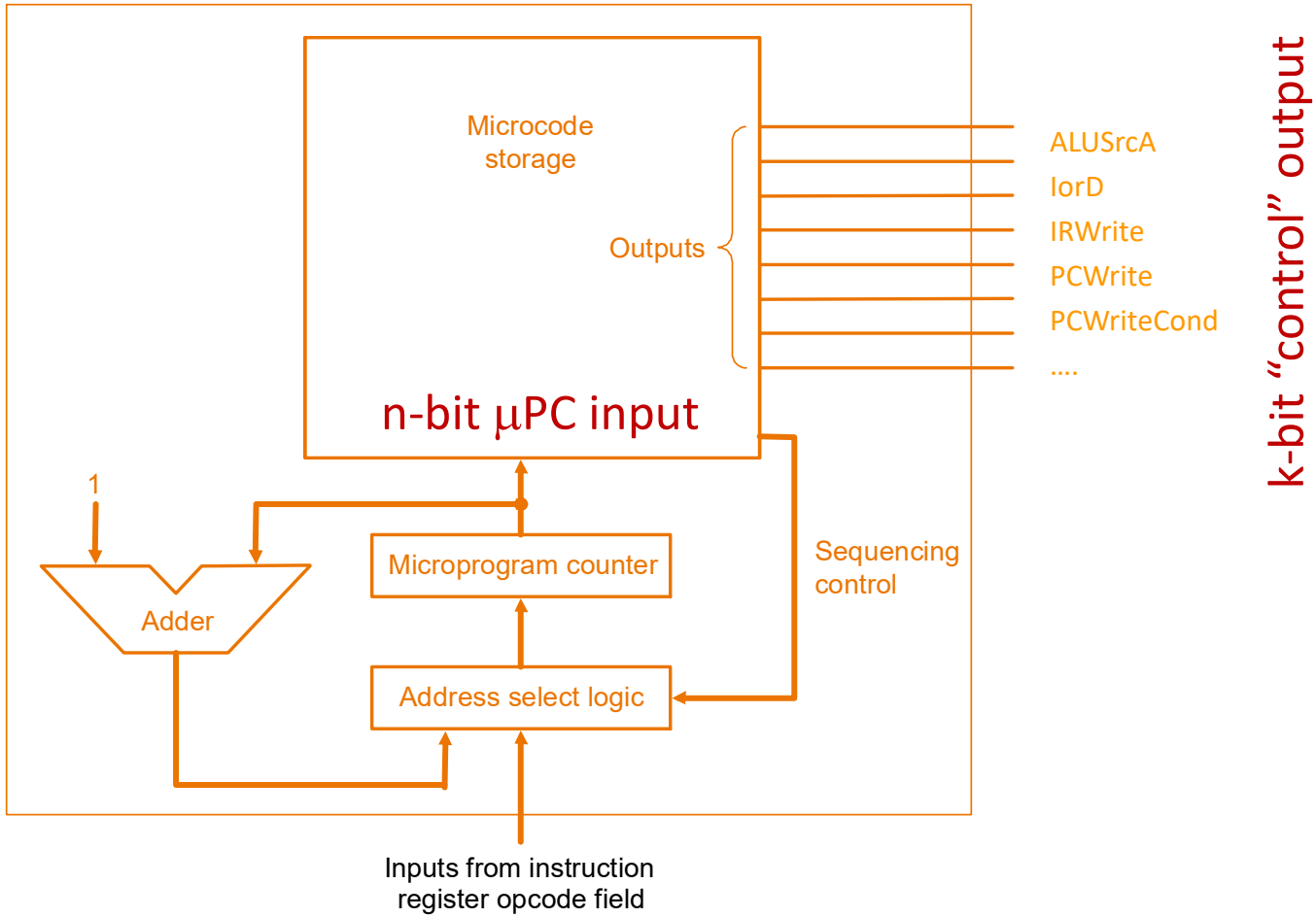
```
if MEM[PC]==LW rd offset(base)
    EA = sign-extend(offset) + GPR[base]
    GPR[rd] ← MEM[ EA ]
    PC ← PC + 4
```

Combined RT Sequencing

	R-Type	LW	SW	Branch	Jump
common steps	start: IR ← MEM[PC]				next
	A ← RF[rs1(IR)] B ← RF[rs2(IR)] ALUOut ← PC+imm(IR)				
opcode dependent steps	ALUOut ← A+B	ALUOut ← A+imm(IR)	case opcode ALUOut ← A+imm(IR)	PC ← PC + 4	PC ← PC+imm(IR) start
	next RF[rd(IR)] ← ALUOut PC ← PC+4 start	next MDR ← M[ALUOut]	next M[ALUOut] ← B PC ← PC+4 start	next cond?(A , B) start if-not-cond	
		next RF[rd(IR)] ← MDR PC ← PC+4 start		PC ← ALUOut start	

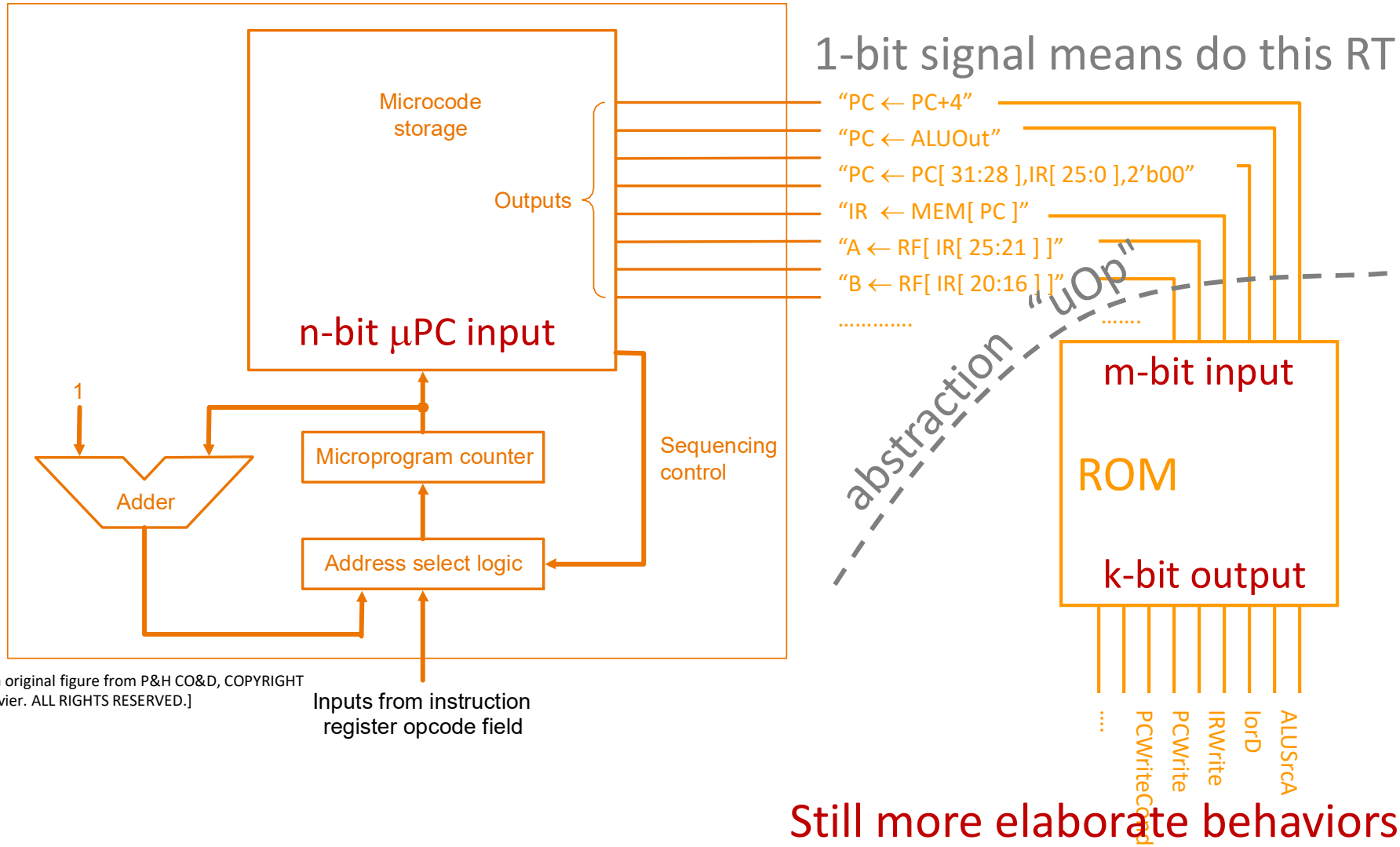
RTs in each state corresponds to some setting of the control signals

Horizontal Microcode



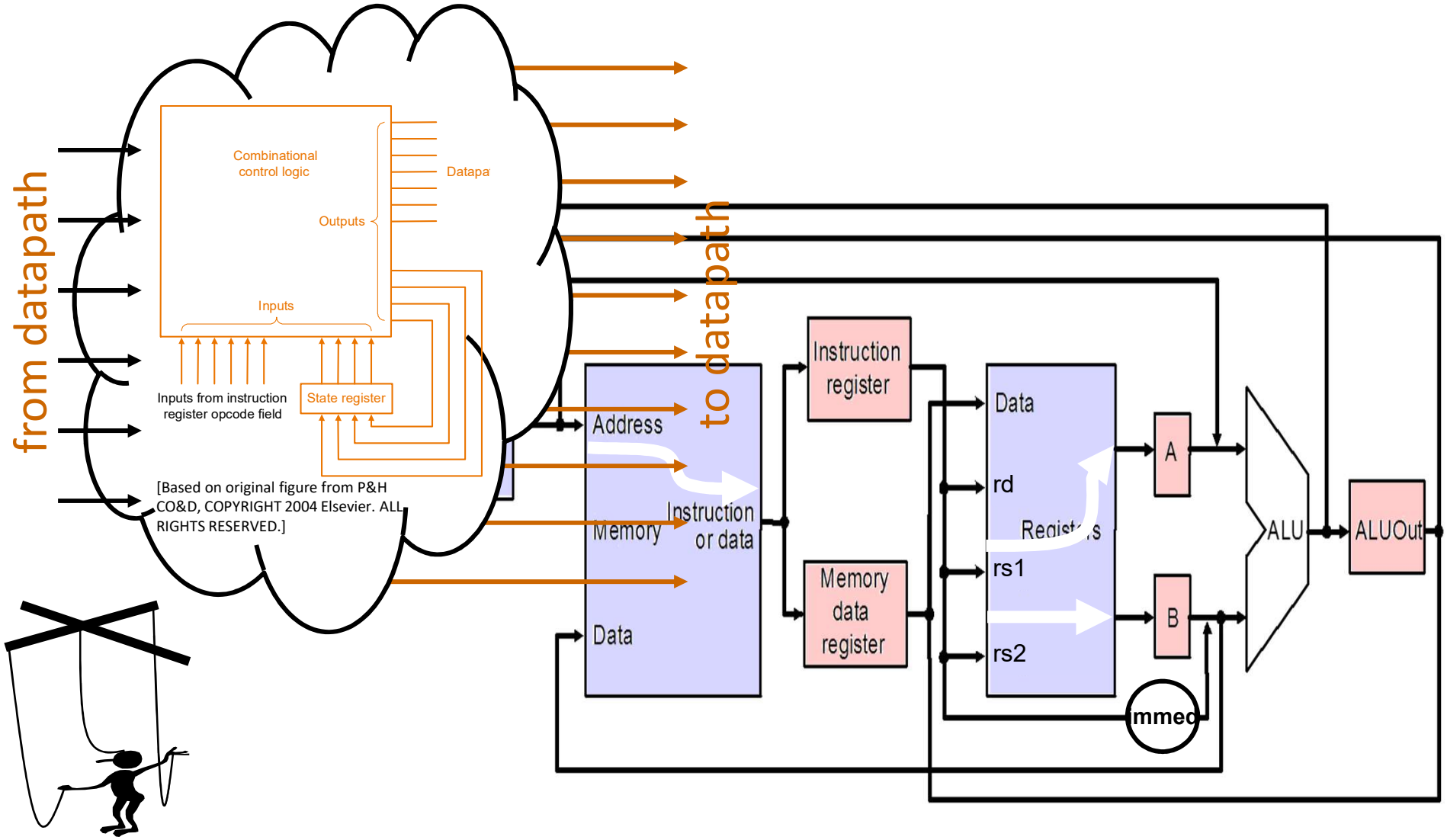
Control Store: $2^n \times$ k bit (not including sequencing)

Vertical Microcode



Still more elaborate behaviors can be sequenced as μsubroutines

μProgrammed Implementation



Microcoding for CISC

- Can we extend last slide
  - to support a new instruction?
  - to support a complex instruction, e.g. polyf?
- Yes, very simple datapath do very complicated things <u>easily but with a slowdown</u>
  - Turing complete

    With enough uOp's, can sequence arbitrary complex instructions and even whole programs
  - will need some μISA state (e.g. loop counters) for more elaborate μprograms
  - more elaborate μISA features also make life easier

Single-Bus Microarchitecture

[8086 Family User's Manual]

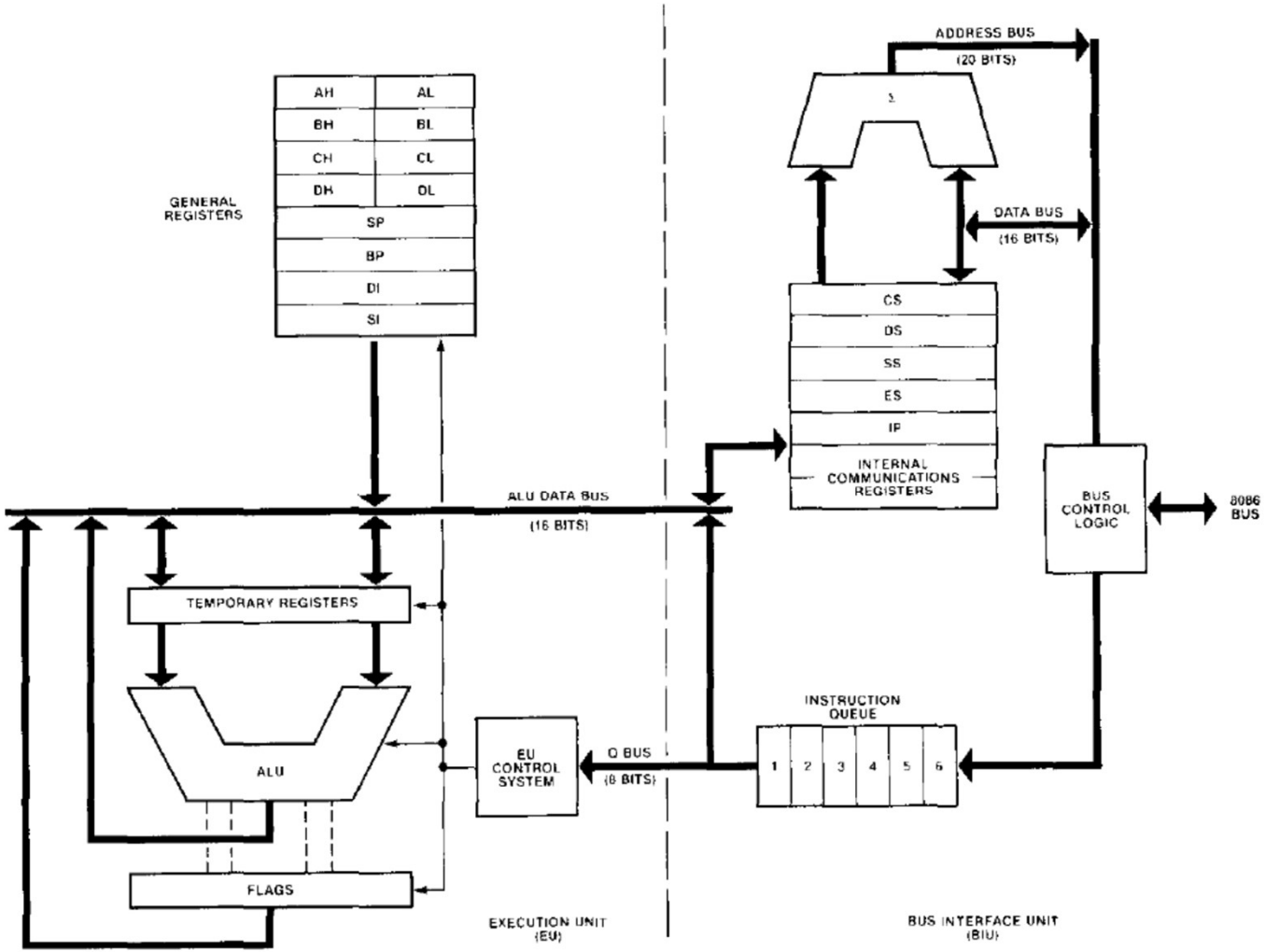


Figure 4-3. 8086 Elementary Block Diagram

You get a try on HW2

Evolution of ISAs

- Why were the earlier ISAs so simple? e.g., EDSAC
  - technology
  - precedence
- Why did it get so complicated later? e.g., VAX11
  - assembly programming
  - lack of memory size and performance
  - microprogrammed implementation
- Why did it become simple again? e.g., RISC
  - memory size and speed (cache!)
  - compilers
- Why is x86 still so popular?
  - technical merit vs. {SW base, psychology, deep pocket}
- Why has ARM thrived while other RISC ISAs vanished

CISC Complex Instruction Set Architecture

Reduced Instruction Set Architecture

Recall

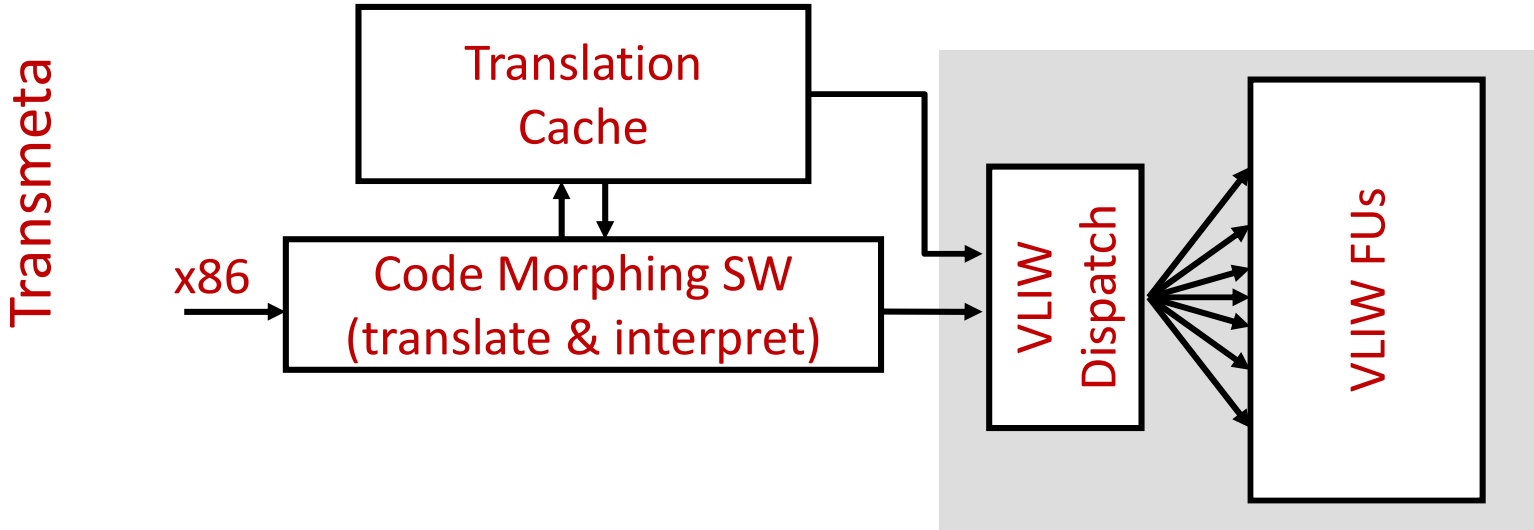
Why RISC-V now?

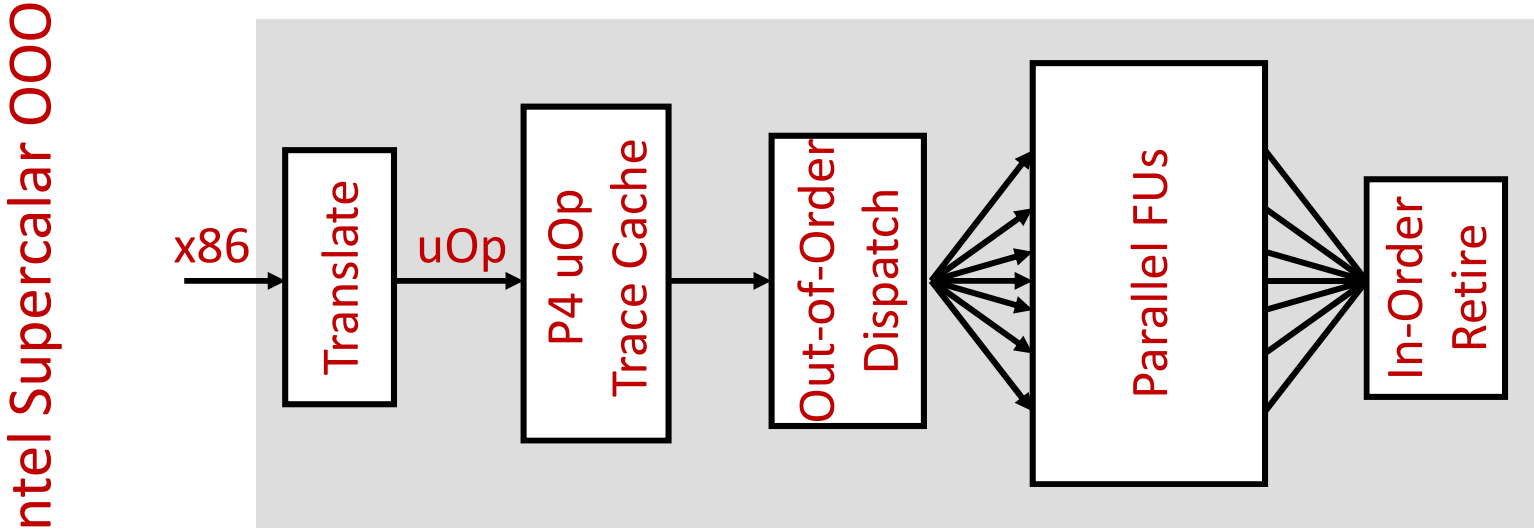
1980's CISC vs RISC Debate

- time/program = (inst/program) (cyc/inst) (time/cyc)
- *"Performance from architecture: comparing a RISC and a CISC with similar hardware organization", Bhandarkar&Clark, 1991*
  - time/cyc on par (MIPS R2000 vs VAX 8700)
  - RISC increases inst/program by ~2
  - CISC increases cyc/inst by ~6

RISC factor: 2.7 savings in cyc/program

End of RISC/CISC Debate

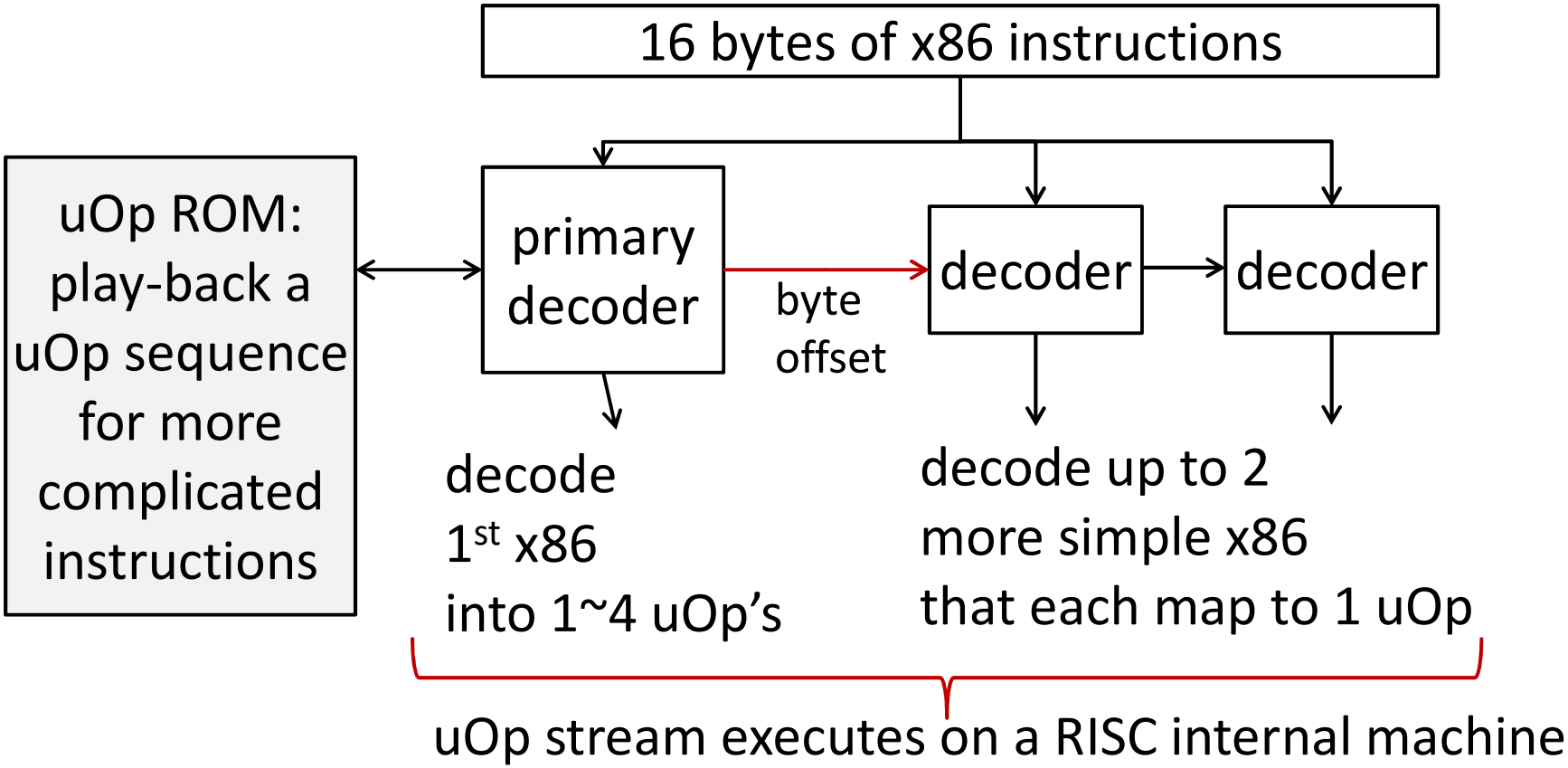




CISC won or RISC won?

High Performance CISC Today

- High-perf x86s translate CISC inst's to RISC uOp's
- Pentium-Pro decoding example:



Compilers helps by avoiding bad insts