CSED311 Computer Architecture – Lecture 6

Multi-Cycle CPU

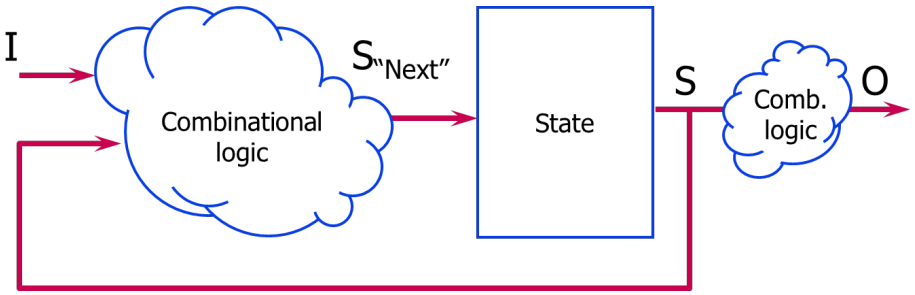
Eunhyeok Park

Department of Computer Science and Engineering
POSTECH

Disclaimer: Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch @ Carnegie Mellon University, University of Michigan, Purdue University, University of Pennsylvania, University of Wisconsin and POSTECH.

Single-Cycle Implementation

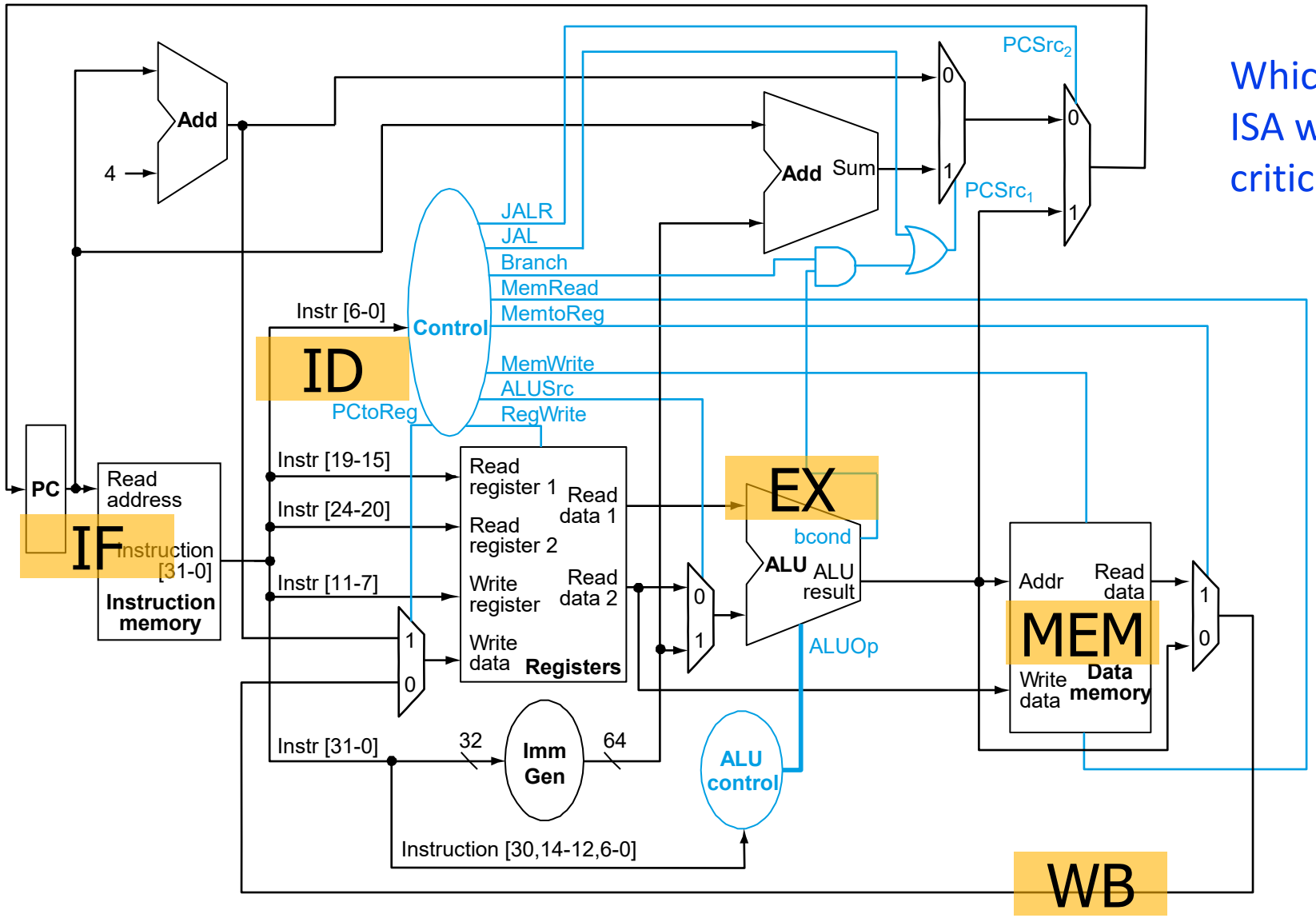
- Matches naturally with the sequential and atomic semantics inherent to most ISAs
  - Instantiate programmer-visible state one-for-one
  - Map instructions to a combinational next-state logic



- But, this is too inefficient
  - All instructions run as slow as the slowest instruction (why?)
  - Must provide the worst-case combinational resource in parallel as required by any instruction

- Not necessarily the simplest way to implement an ISA
  - Gets much worse for a CISC ISA (e.g., polyf?)

Single-Cycle Implementation



Which instruction from RISC-V ISA would have the longest critical path?

Single-Cycle Datapath Analysis

- Assume
  - Memory units (read or write): 200 ps (=0.2 nano-sec)
  - ALU (add op) : 100 ps
  - Register file (read or write): 50 ps
  - Other combinational logic: 0 ps

What's the shortest clock period possible for this implementation?

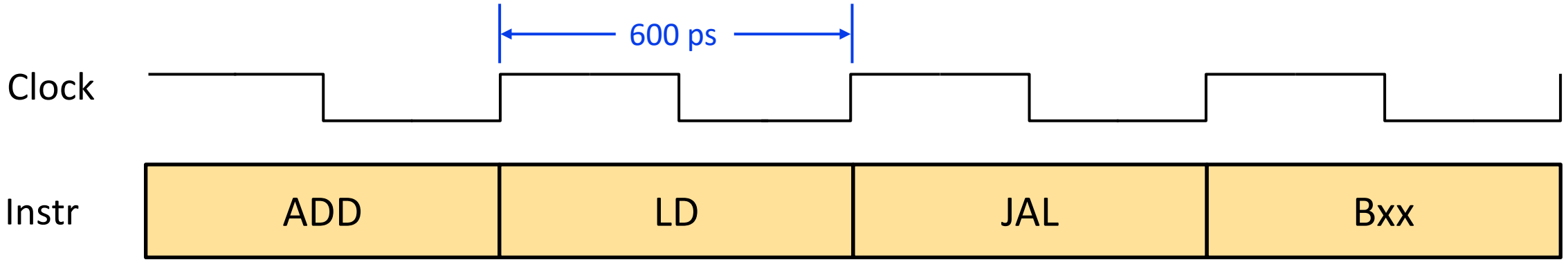
Steps	IF	ID	EX	MEM	WB	Delay
Resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type(Arith.)	200	50	100		50	400
LD	200	50	100	200	50	600
SD	200	50	100	200		550
Bxx	200	50	100			350
JAL	200		100		50	350
JALR	200	50	100		50	400

Multi-Cycle Implementation: Ver 1.0

- Let's make each instruction type take **only as much time as it needs**
- Idea
  - Run a 50ps clock
  - Let each instruction type take as many clock cycles as needed
  - Programmer-visible state (PVS) only updates at the end of an instruction's cycle-sequence
    - Controlled by "`PVSWriteEn`" signal (see next slide)
    - No changes in the architectural state when `PVSWriteEn==0`
  - An instruction's effect is still purely combinational from PVS (program visible state) to PVS

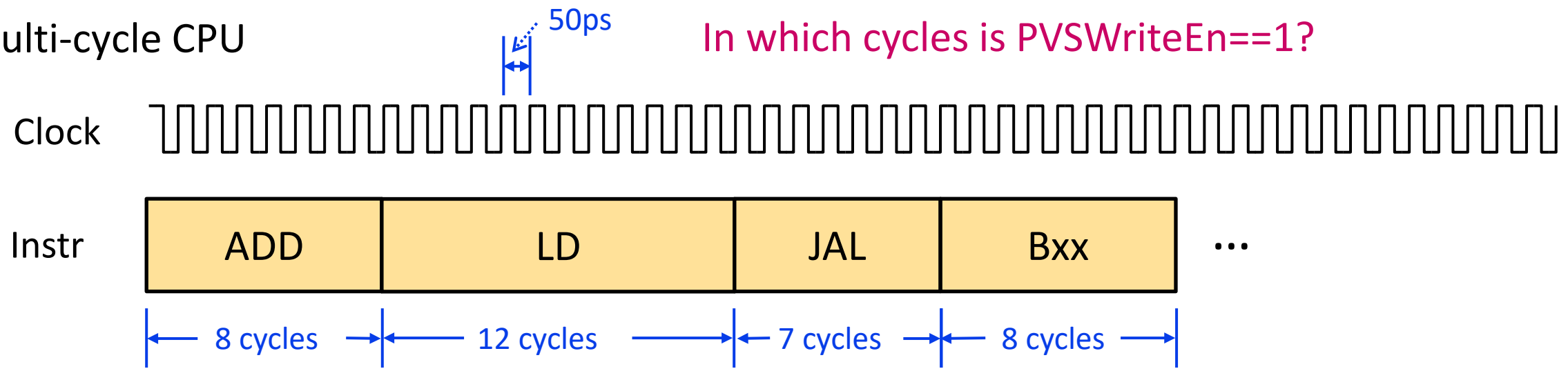
Single-Cycle vs. Multi-Cycle CPU

- Single-cycle CPU

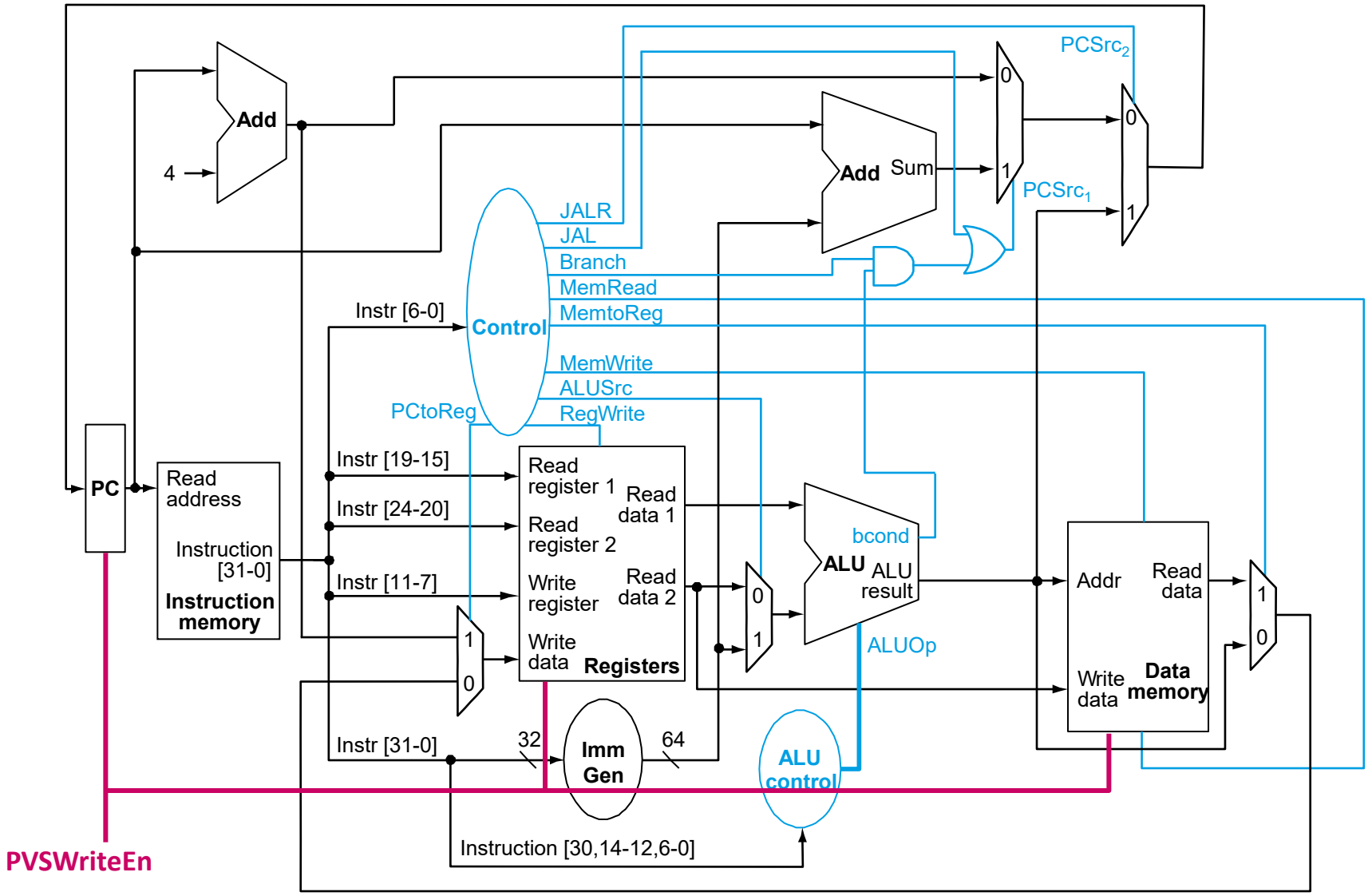


- Multi-cycle CPU

In which cycles is PVSWriteEn==1?



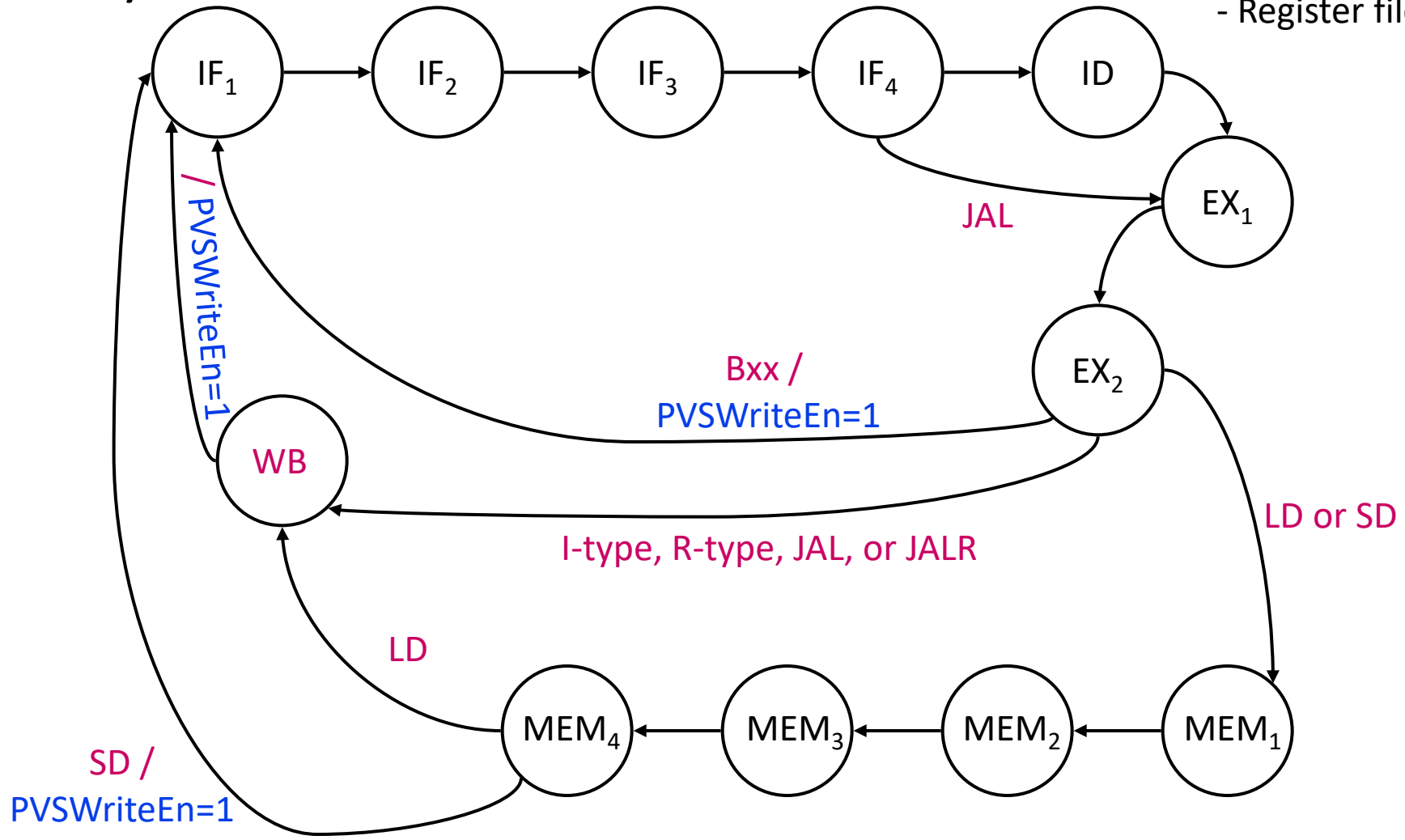
Multi-Cycle Datapath: Ver 1.0



Sequential Control: Ver 1.0

Assumptions (from slide 4):
- Memory (R/W): 200 ps
- ALU: 100 ps
- Register file (R/W): 50 ps

Mealy FSM



What about the rest of the control signals?

Performance Analysis

- Iron law:

$$T_{wall\text{-}clock} = (instrs/program) \times \underbrace{(cycles/instr)}_{CPI} \times \underbrace{(time/cycle)}_{T_{clk}}$$

- "# of instructions" is fixed for given ISA, compiler, and application

  → Then, we can simply compare

$$T_{avg\text{-}inst} = CPI \times T_{clk}$$

  or

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

Million instructions per second

Instructions per cycle

Frequency in MHz

Performance Analysis

- Our assumption: clock period is 600ps for single-cycle CPU, 50ps for multi-cycle CPU (1 clock == 12 μclock)

- Single-cycle implementation (MIPS = IPC × f_{clk})

  → 1 × 1,667MHz = 1667 MIPS

- Multi-cycle implementation

  → **IPC_{avg}** × 20,000 MHz – but what is average IPC?

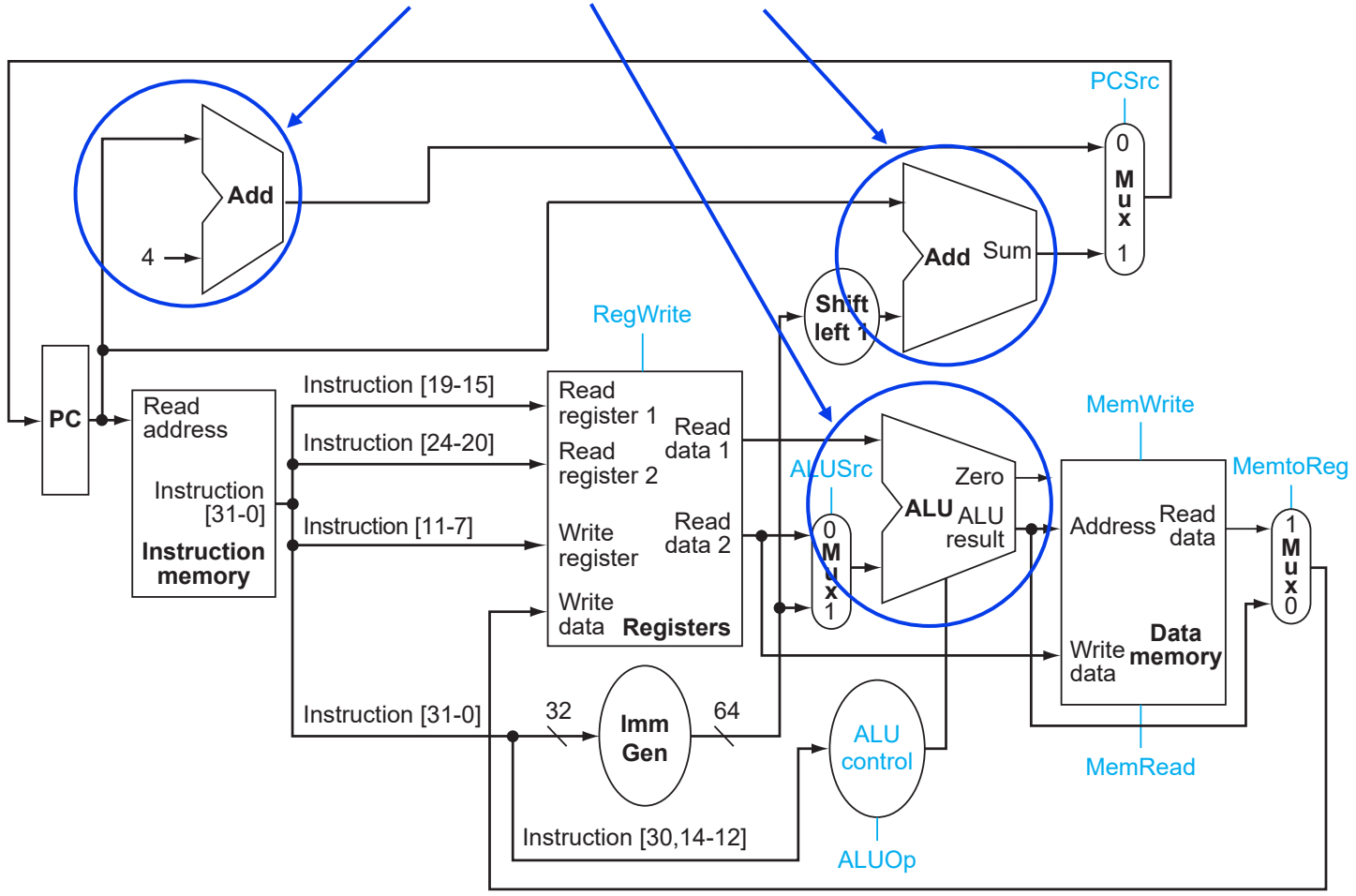
  – Assume: 25% LD, 10% SD, 45% ALU, 15% Bxx, 3% JAL and 2% JALR

    - Weighted arithmetic mean (WAM) of CPI
      - 0.25×12+0.1×11+0.45×8+0.15×7+0.03×7+0.02×8 = 9.12
    - **Weighted harmonic mean (WHM) of IPC** = 1/CPI ≈ 0.1096

  → **IPC_{avg}** × 20,000 MHz = 2193 MIPS → 2193/1667 = 31.6% speedup!!

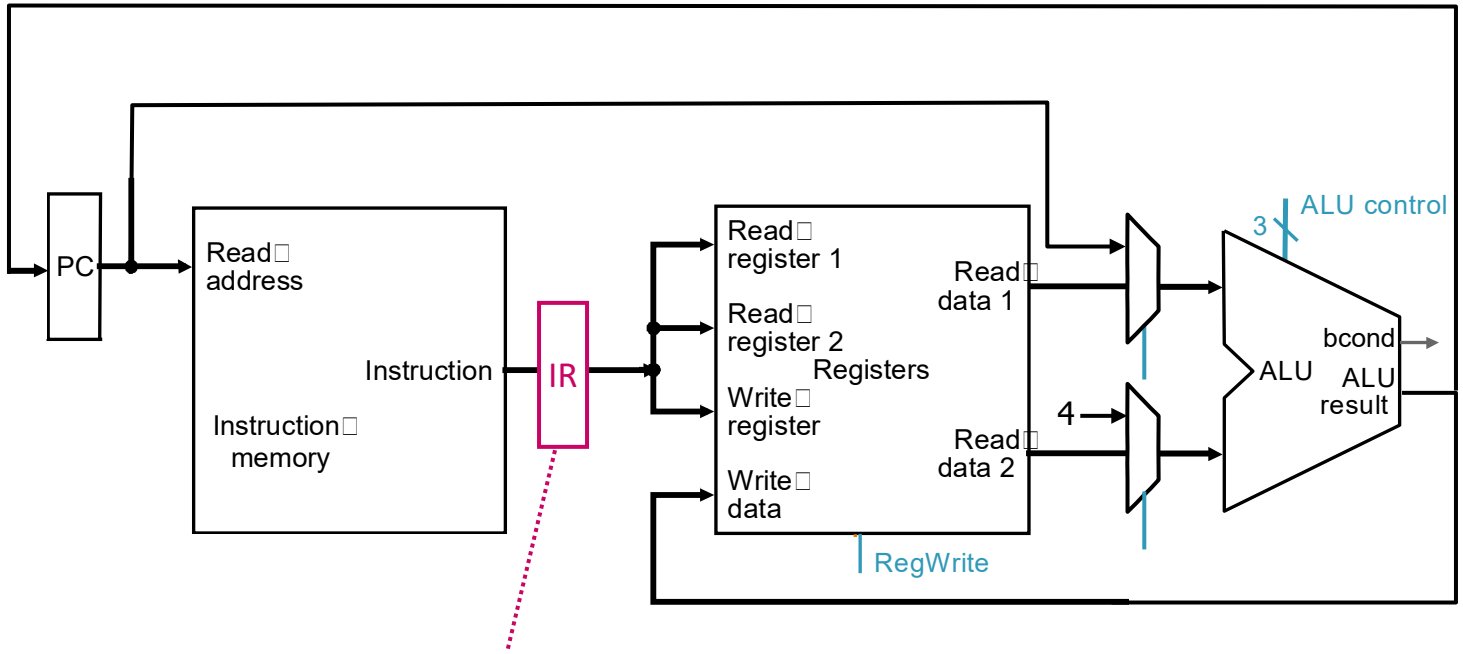
Reducing Datapath by Resource Reuse

How to reuse the same adder for all additions in the same instruction?



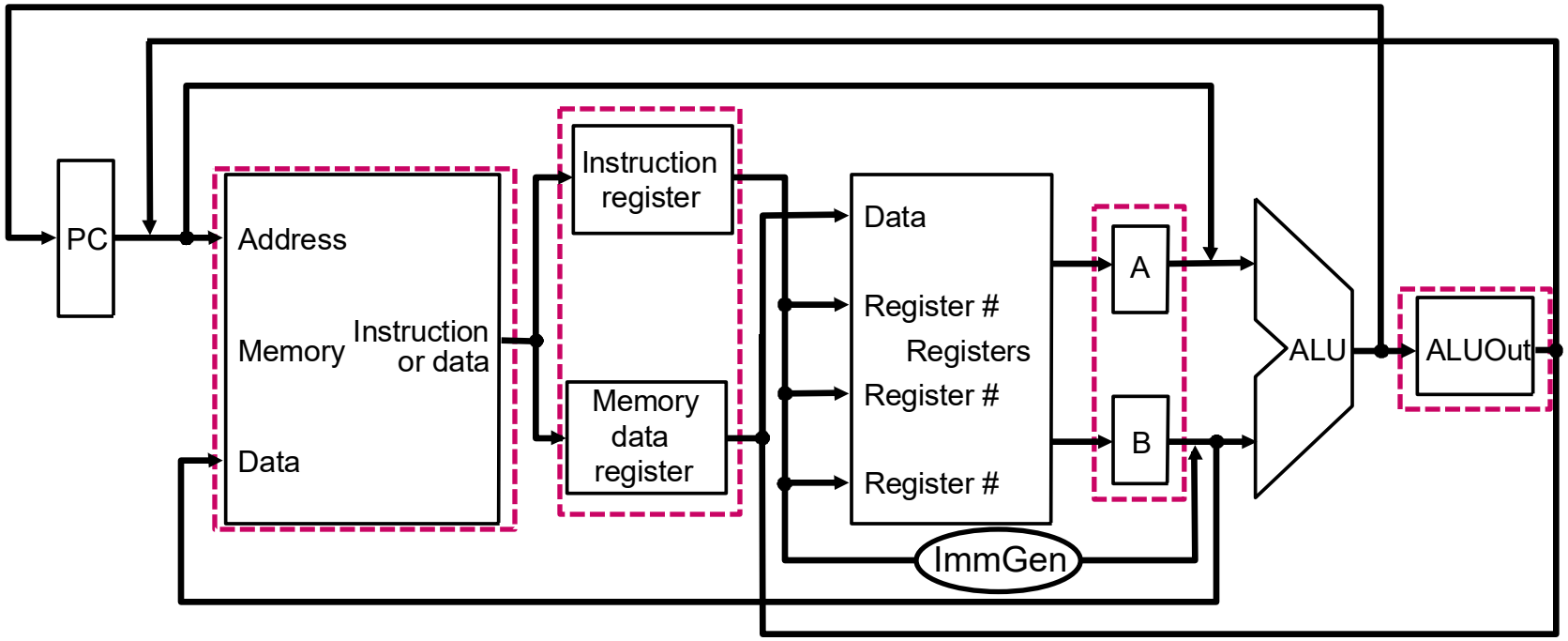
Reducing Datapath (ALU) by Sequential Reuse

But, must create timing difference to avoid ALU resource conflict
(i.e., use the ALU in different cycles)



Instruction Register (IR)
--> Keep the instruction after reading it once from memory

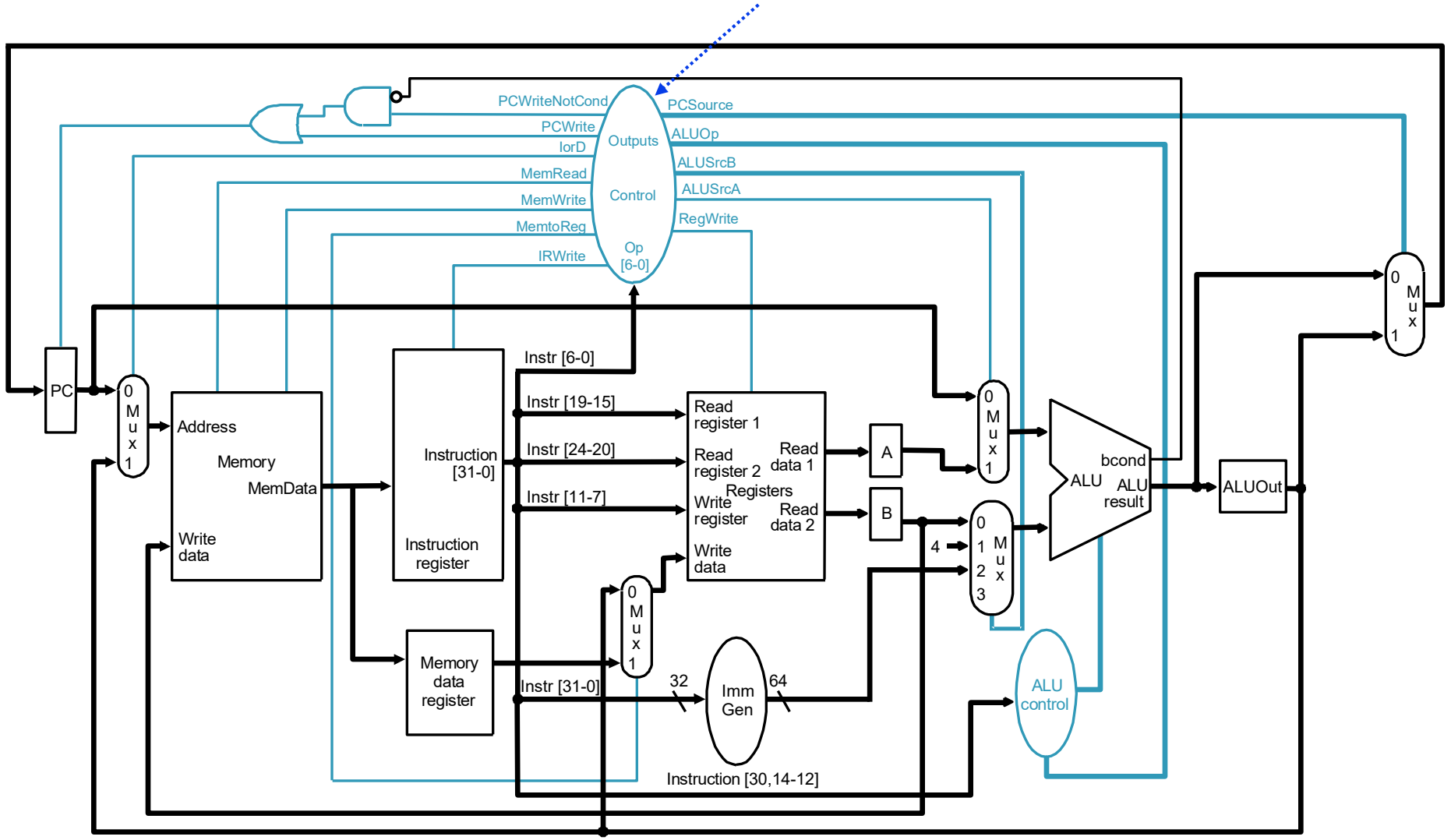
Removing Redundancies



We can also use single memory by creating **timing differences** with proper controls
(It is also possible to even reduce to a single register read-write port)

Control Points

Needs to give different control signal values in different cycles while executing an instruction.
This can be implemented using a microsequencer (later slide).



[Figure Based on figures from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

New Sequential Control Signals (Single-bit)

	When De-asserted	When Asserted
ALUSrcA	1st ALU input from **PC**	1st ALU input from 1st **RF** read port (latched in **A**)
IorD	**PC** supplies instruction address	**ALUOut** supplies data address
IRWrite	**IR** latching disabled	**IR** latching enabled
PCSource	next **PC** from ALU	next **PC** from **ALUOut**
PCWrite	no effect	**PC** latching enabled unconditionally
PCWriteNotCond	no effect	**PC** latching enabled only if branch condition is NOT satisfied

When both PCWrite and PCWriteNotCond are de-asserted, **PC** latching is disabled

New Sequential Control Signals (Multi-bit)

Signal		Effect
ALUSrcB[1:0]	00	2nd ALU input from 2nd **RF** read port (latched in **B**)
	01	2nd ALU input is 4 (for PC increment)
	10	2nd ALU input is output from the immediate value generation logic
	11	(unused)

Old Control Signals (similar to single-cycle CPU)

	When De-asserted	When Asserted
RegWrite	**RF** write disabled	**RF** write enabled
MemRead	Memory read disabled	Memory read port return load value
MemWrite	Memory write disabled	Memory write enabled
MemtoReg	Steer ALU result (latched in **ALUOut**) to **RF** write port	Steer memory load result (latched in **MDR**) to **RF** write port

Synchronous Register Transfers

- Synchronous states with latch enabled by control
  - **PC**, **IR**, **A**, **B**, **ALUOut**, **MDR**, **MEM**, **RF**
- Now we can enumerate all possible combinational "**Register Transfers**" in the datapath! (whether they will be used or not)
- For example, starting from **PC**
  - **PC ← PC + 4**
  - **PC ← PC + immediate(IR)**
  - **PC ← A + immediate(IR)**
  - **PC ← ALUOut**
  - **IR ← MEM[PC]**
  - **A ← RF[rs1(IR)]**
  - **B ← RF[rs2(IR)]**
  - **ALUOut ← A + B**
  - **ALUOut ← A + immediate(IR)**
  - **ALUOut ← PC + 4**
  - **ALUOut ← PC + B**
  - **MDR ← MEM[ALUOut]**
  - **MEM[ALUOut] ← B**
  - **RF[rd(IR)] ← ALUOut**
  - **RF[rd(IR)] ← MDR**
  - **…**

RT Sequencing: R-Type ALU

- IF
  - **IR ← MEM[PC]**
- ID
  - **A ← RF[rs1(IR)]**
  - **B ← RF[rs2(IR)]**
- EX
  - **ALUOut ← A + B**
- MEM
- WB
  - **RF[rd(IR)] ← ALUOut**
  - **PC ← PC+4**

◆ Semantics

- GPR[rd] ← GPR[rs1] + GPR[rs2]
- PC ← PC + 4

RT Datapath Conflicts

PC ← PC+4

PC ← PC+imm(IR)

PC ← A+imm(IR)

IR ← MEM[PC]

A ← RF[rs2(IR)]

B ← RF[rs2(IR)]

MEM[ALUOut] ← B

RF[rd(IR)] ← ALUOut

RF[rd(IR)] ← MDR

ALUOut ← A + imm(IR)

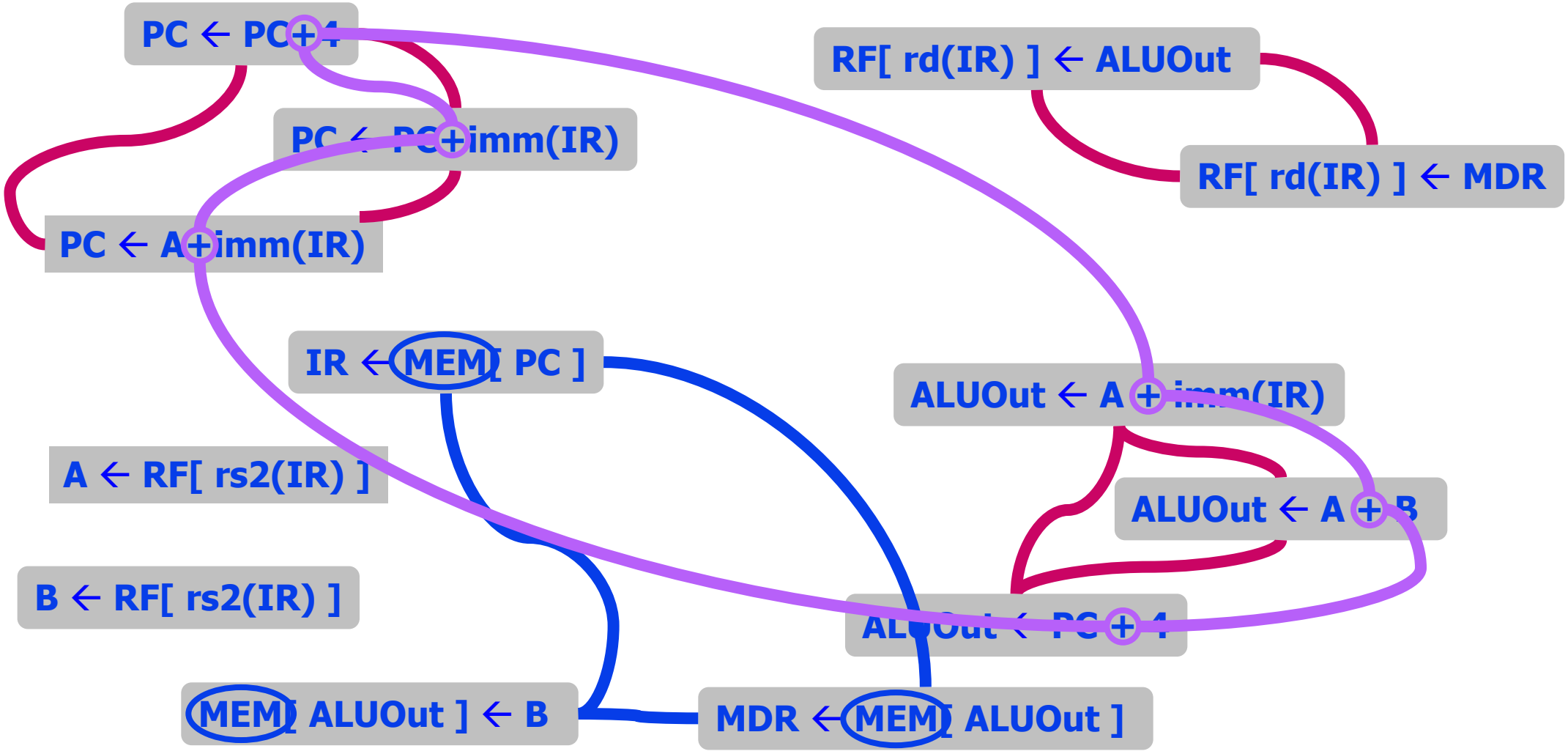
ALUOut ← A + B

ALUOut ← PC + 4

MDR ← MEM[ALUOut]

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

RT Datapath Conflicts



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

RT Sequencing: R-Type ALU

- IF
  - IR ← MEM[PC]

step 1

- ID
  - A ← RF[rs1(IR)]
  - B ← RF[rs2(IR)]

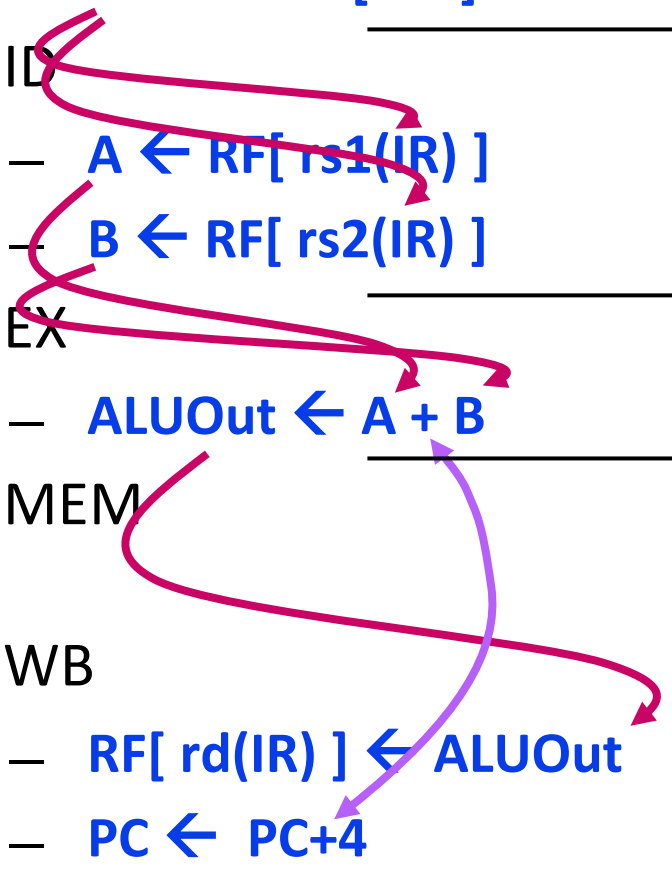
step 2

- EX
  - ALUOut ← A + B

step 3

- MEM
- WB
  - RF[rd(IR)] ← ALUOut
  - PC ← PC+4

step 4

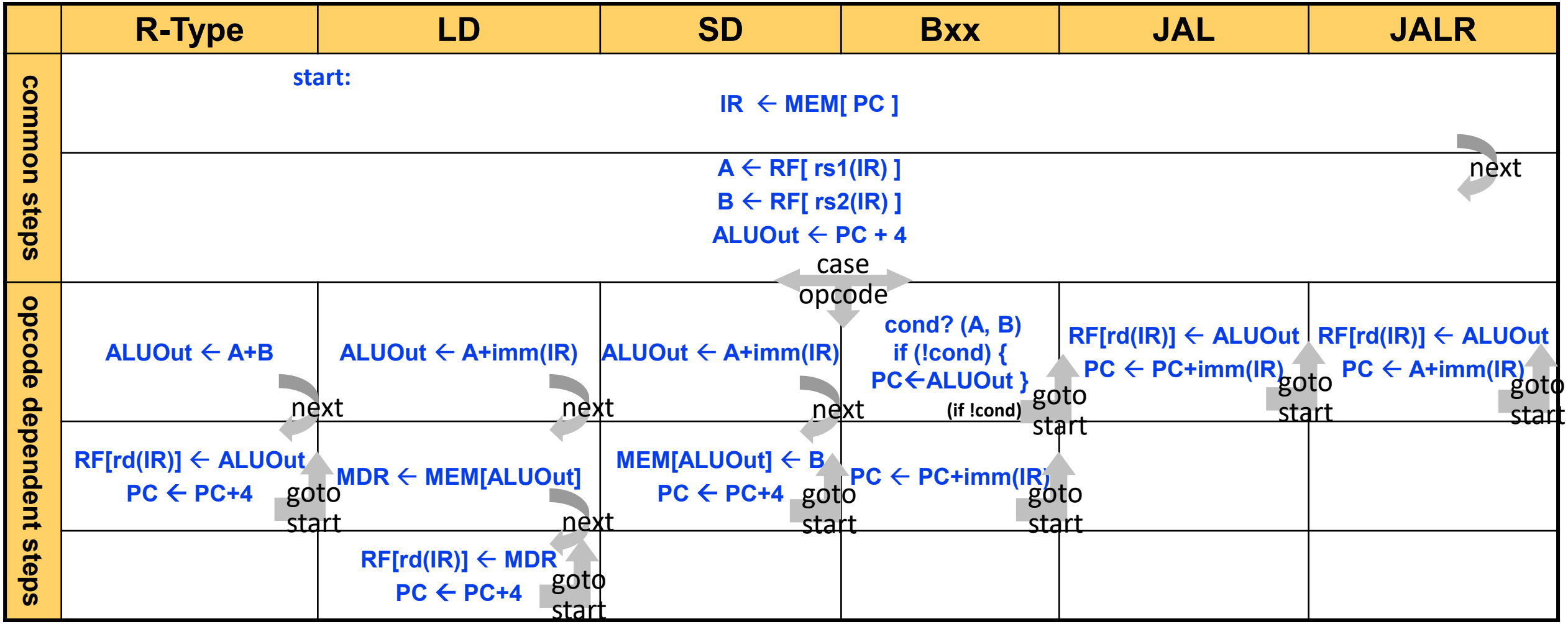


RT Sequencing: LD

- IF
  - **IR ← MEM[PC]**
- ID
  - **A ← RF[rs1(IR)]**
- EX
  - **ALUOut ← A + $imm_{I\text{-}type}$ (IR)**
- MEM
  - **MDR ← MEM[ALUOut]**
- WB
  - **RF[rd(IR)] ← MDR**
  - **PC ← PC+4**

- Semantics:
  - $byte\_addr_{32}$ = sign-extend($offset_{12}$) + GPR[base]
  - GPR[rd] ← MEM_{32} [$byte\_addr_{32}$]
  - PC ← PC + 4

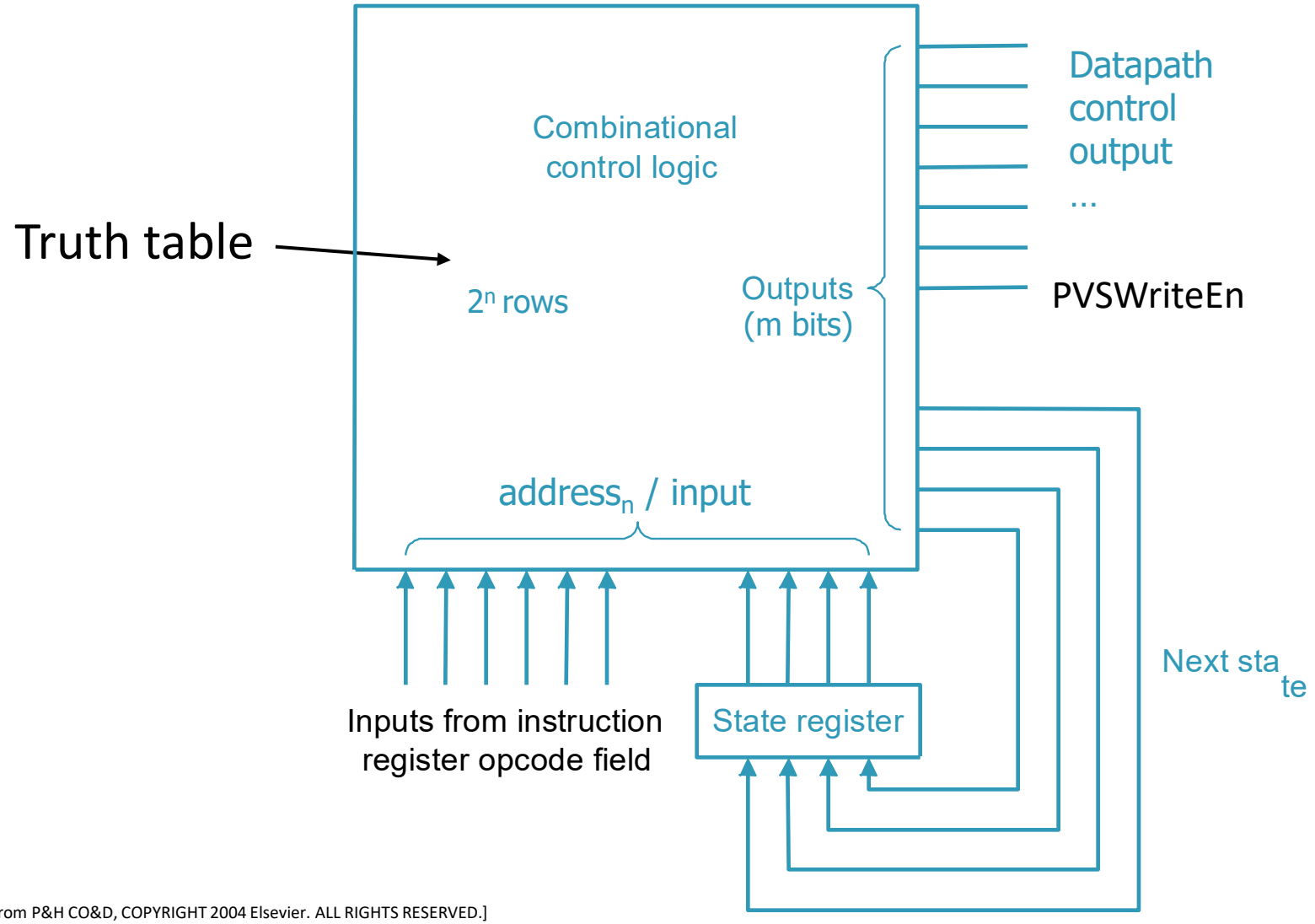
Combined RT Sequencing



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

MicroSequencer: Ver 1.0

- ROM as a combinational logic lookup table to make FSM



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

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

➔ This includes a lot of Invalid inputs! (e.g., “MEMx” state for all instructions other than load/store, unused opcode values)

Microcoding: Ver 1.0

~2^(opcode field's bit width)

~2^(state register's bit width)

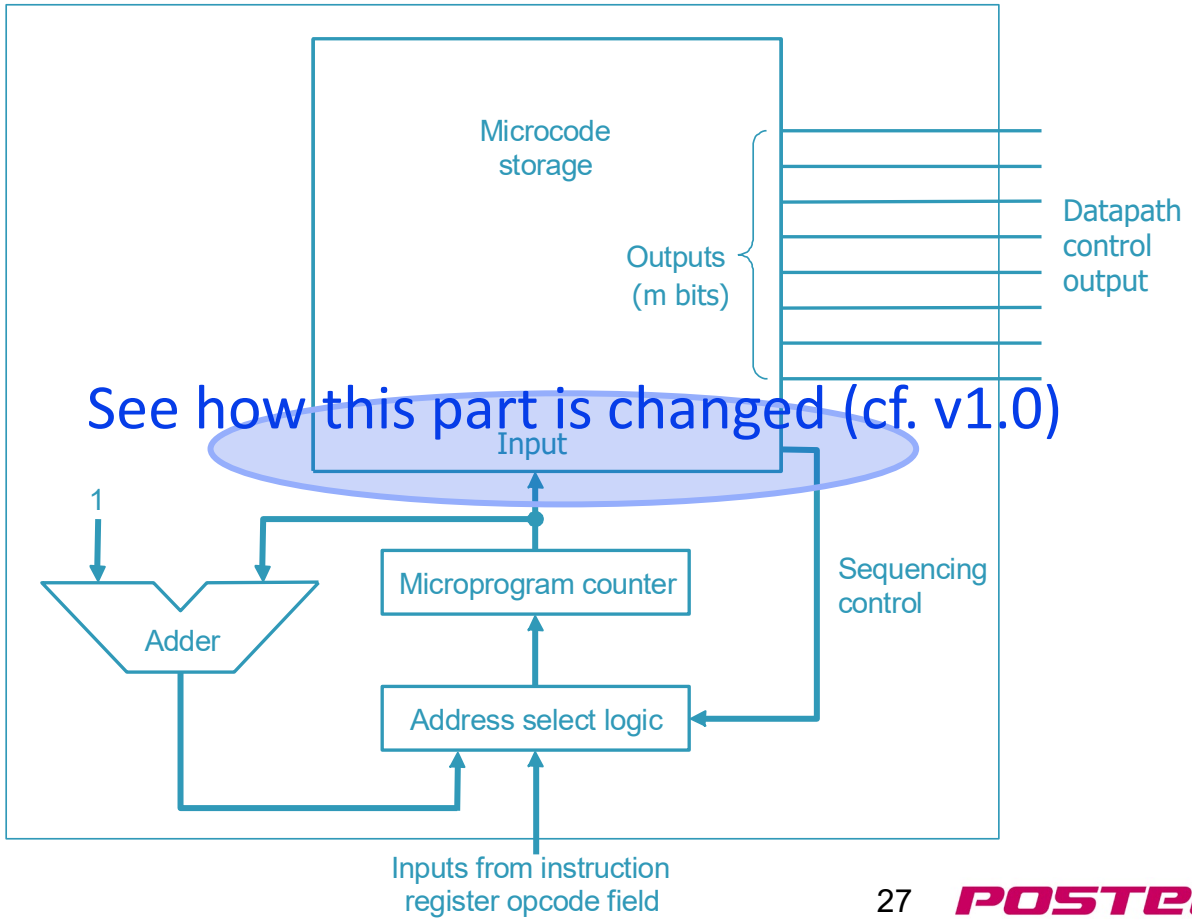
State label	Control flow	Conditional targets					
		R/I-type	LD	SD	Bxx	JALR	JAL
IF_1	next	-	-	-	-	-	-
IF_2	next	-	-	-	-	-	-
IF_3	next	-	-	-	-	-	-
IF_4	go to	ID	ID	ID	ID	ID	EX_1
ID	next	-	-	-	-	-	
EX_1	next	-	-	-	-	-	-
EX_2	go to	WB	MEM_1	MEM_1	IF_1	WB	WB
MEM_1	next		-	-			
MEM_2	next		-	-			
MEM_3	next		-	-			
MEM_4	go to		WB	IF_1			
WB	go to	IF_1	IF_1			IF_1	IF_1
CPI		8	12	11	7	8	7

Each cell corresponds to a row in the ROM. Note the redundant or unused cells.

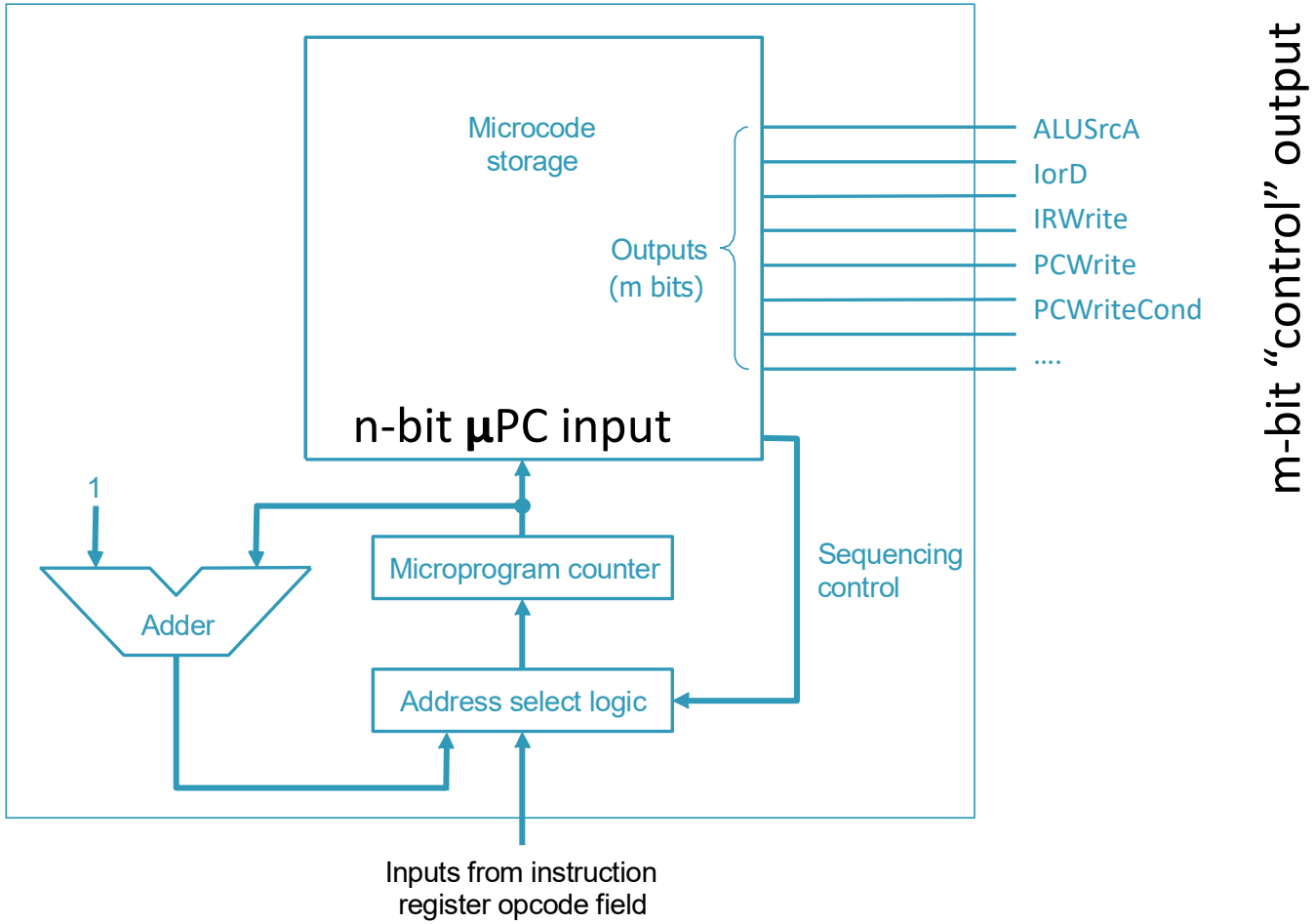
Micro-code controller – A Better Way

- A small processor for sequencing and control purpose
  - Control states are like μPC (mostly just incremented by one)
  - μPC indexed into a μprogram ROM to select a μinstruction
  - Fields in the μinstruction maps to control signals
  - Well-formed control-flow architecture

- Large instruction set for CISC machines can be built efficiently this way!
  - Using μISAs for CISC machines inspired RISC ISA

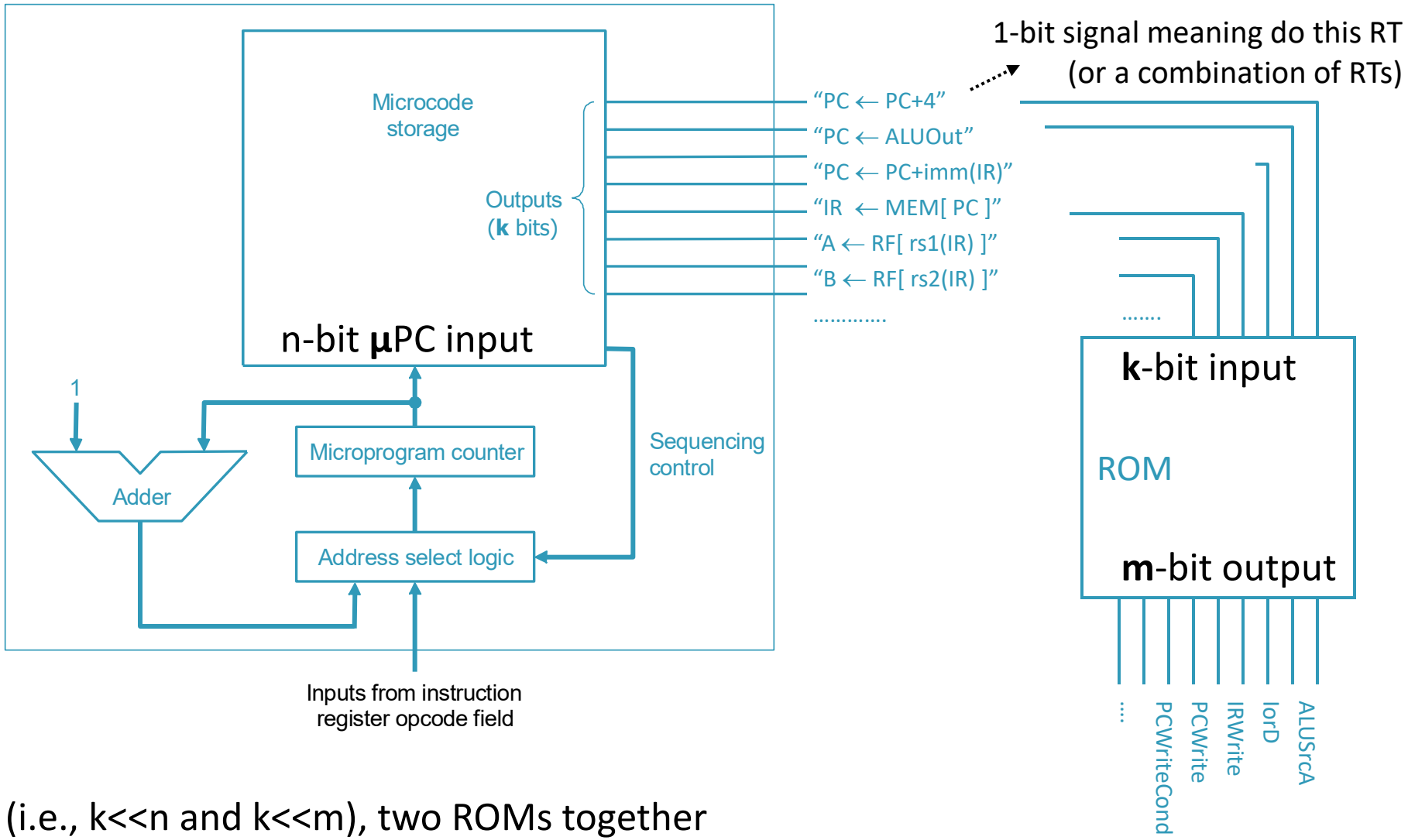


Horizontal Microcode



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

Vertical Microcode



If done right (i.e., k<<n and k<<m), two ROMs together
($2^n \times k + 2^k \times m$ bit) should be smaller than horizontal microcode ROM ($2^n \times m$ bit)

Microcoding for CISC

- Can we extend the μcontroller and datapath?
  - To support a new instruction I haven't thought of yet
  - To support a complex instruction, e.g., polyf
- Yes, and probably more
  - If I can sequence an arbitrary RISC instruction, then I can sequence an arbitrary "RISC program" as a μprogram sequence
  - Will need some μISA state (e.g., loop counters) for more elaborate μprograms
  - More elaborate μISA features also make life easier
- μcoding allows very simple datapath to do very powerful computation
  - A datapath as simple as a Turing machine is universal
  - μcode enables a minimal datapath to emulate any ISA you like (possibly with a very large slow down)

Nanocode and Millicode

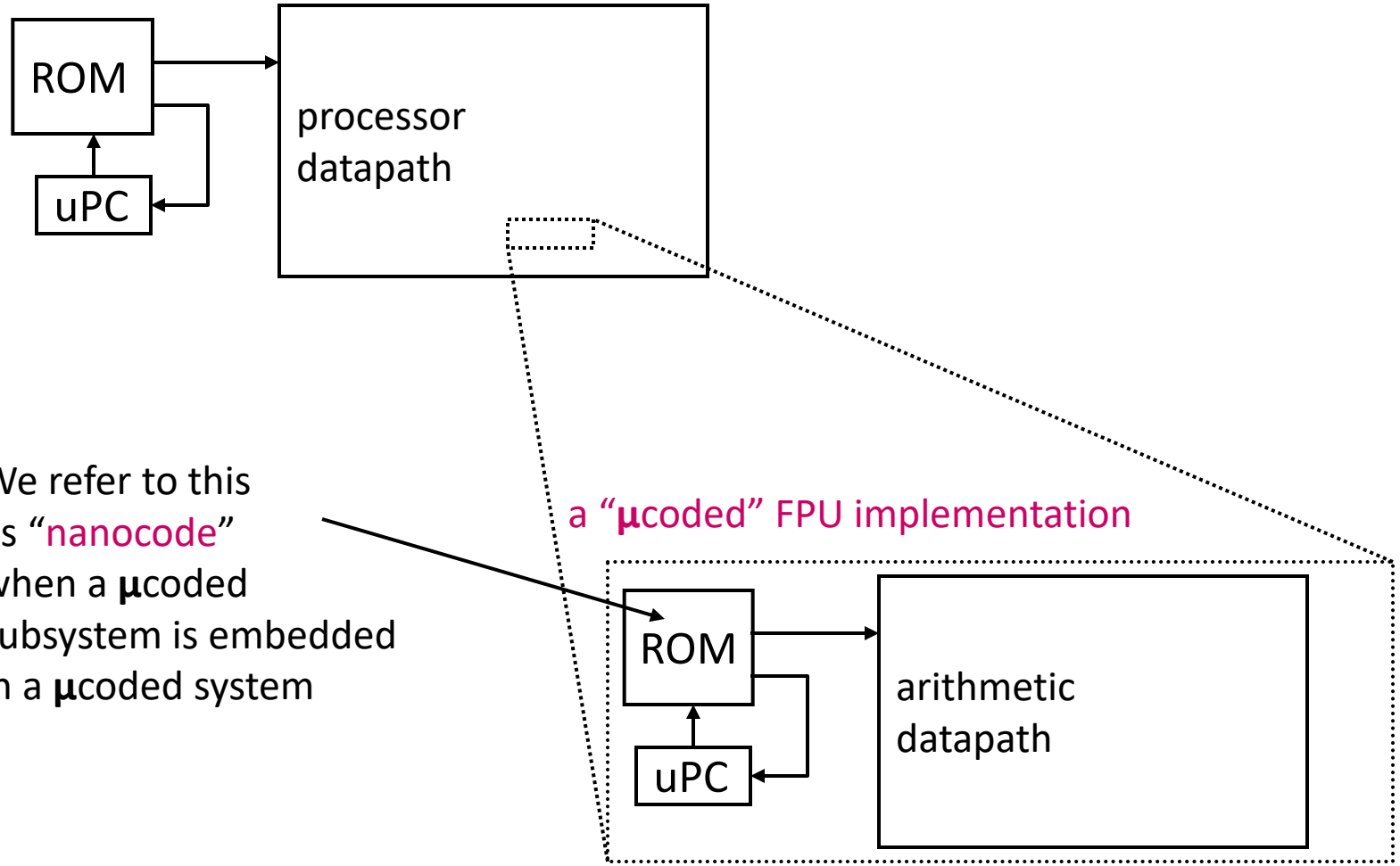
- Millicode
  - A level **above** μcode
  - ISA-level subroutines hardcoded into a ROM that can be called by the μcontroller to handle really complicated operations

    e.g., To add polyf to MIPS ISA, one may code up polyf as a software routine that is called by the μcontroller when the polyf **opcode** is decoded → "Instruction emulation"
- Nanocode
  - Another level **below** μcode
  - μprogrammed control for sub-systems (e.g., a complicated floating-point module) that acts as a slave in a μcontrolled datapath
  - e.g., The polyf sequence may be generated by a separate nanocontroller in the FPU → more-fine grained coding
- In both cases, we don't complicate the μcontroller for polyf support

The power of abstraction!!

Nanocode Example

a "μcoded" processor implementation



Now you know how to design a microcode-based multi-cycle CPU!

Questions?

Announcements

- Reading: P&H (RISC-V ed.) Ch 4.5-4.6