Den simpleste maskine

Slides af Finn Schiermer Andersen og

Slides fra COD – maltrakteret af Finn Schiermer Andersen

Hvad er et godt instruktions-sæt?

Registre vs lager – basic facts:

Registre har lille adresserum administreret af compileren
Lager har stort adresserum administreret af software

Tilgang til registre har kendt, fast, kort latenstid
Tilgang til lageret har variabel og længere latenstid

Tilgang til registre lykkes altid
Tilgang til lager kan fejle

Registre

En compiler kan gøre god brug af et begrænset antal registre, 16-32.
- løber hurtigt tør hvis der er mindre end 16
- løber sjældent tør hvis der er over 32

En compiler kan bedst generere kode med simple operationer med to input og et resultat, hvor registre bruges til både input og resultat

Indkodning af 3 registre kræver 15 bit (hvis der er 32 registre)

Konstanter

Konstanter bruges ofte som det ene input til en operation
Konstanter er ofte små

En compiler kan gøre god brug af operationer med et register input,
et konstant input og et register output.

Tilgang til lageret

Tilgang til lageret kræver en adresse
En addresse i lageret er et tal der skal beregnes (en pointer)

Disse beregninger kan udtrykkes særskilt via instruktioner
der følger før omtalte 2-input-1-output format

De kan også specificeres som en del af en anden instruktion,
f.eks. ADD 8(x7),24(x9,x10) – det designvalg medfører altid
et instruktionssæt hvor instruktionerne har variabel længde.

Hvad er mon ulempen ved variabel længde instruktioner?

Load/Store arkitektur

En load/store arkitektur har dedikerede instruktioner til at læse fra og skrive til lageret.

Disse instruktioner kan indeholde en simpel addresse-beregning, typisk register+konstant eller register+register

Alle instruktioner har samme længde og hver angiver max to input-registre og et output-register

RISC-V er en load/store arkitektur

Mikroarkitektur

En load/store maskine tillader en simpel mikroarkitektur.

Mikroarkitekturen matcher compileren

Datavejen er fokuseret på læsning af to registre,
Beregning af et resultat, skrivning til et register.

De grumme detaljer følger om lidt!

Lidt historie

Første Load/Store arkitektur: CDC6600 (Seymor Cray, 1964)

Men frem til midt-80'erne byggede man maskiner med mere og mere komplicerede instruktioner. Kulminerende i en instruktion som kunne evaluere et polynomium.

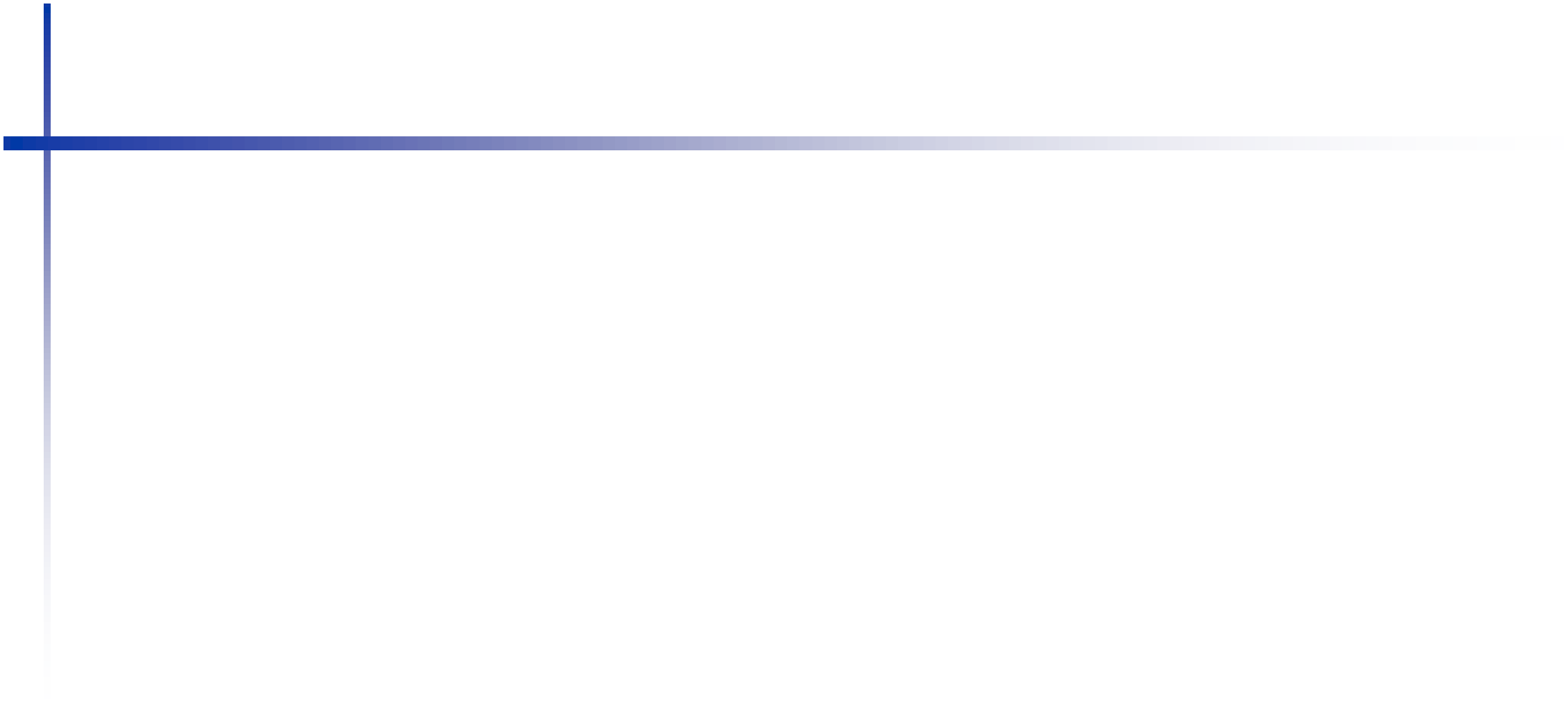
IBM "genopdagede" load/store og hvor godt det passede til compileren i sidste halvdel af halvfjerdserne. Glemte det igen. Genopdagede det sidst I 80'erne.

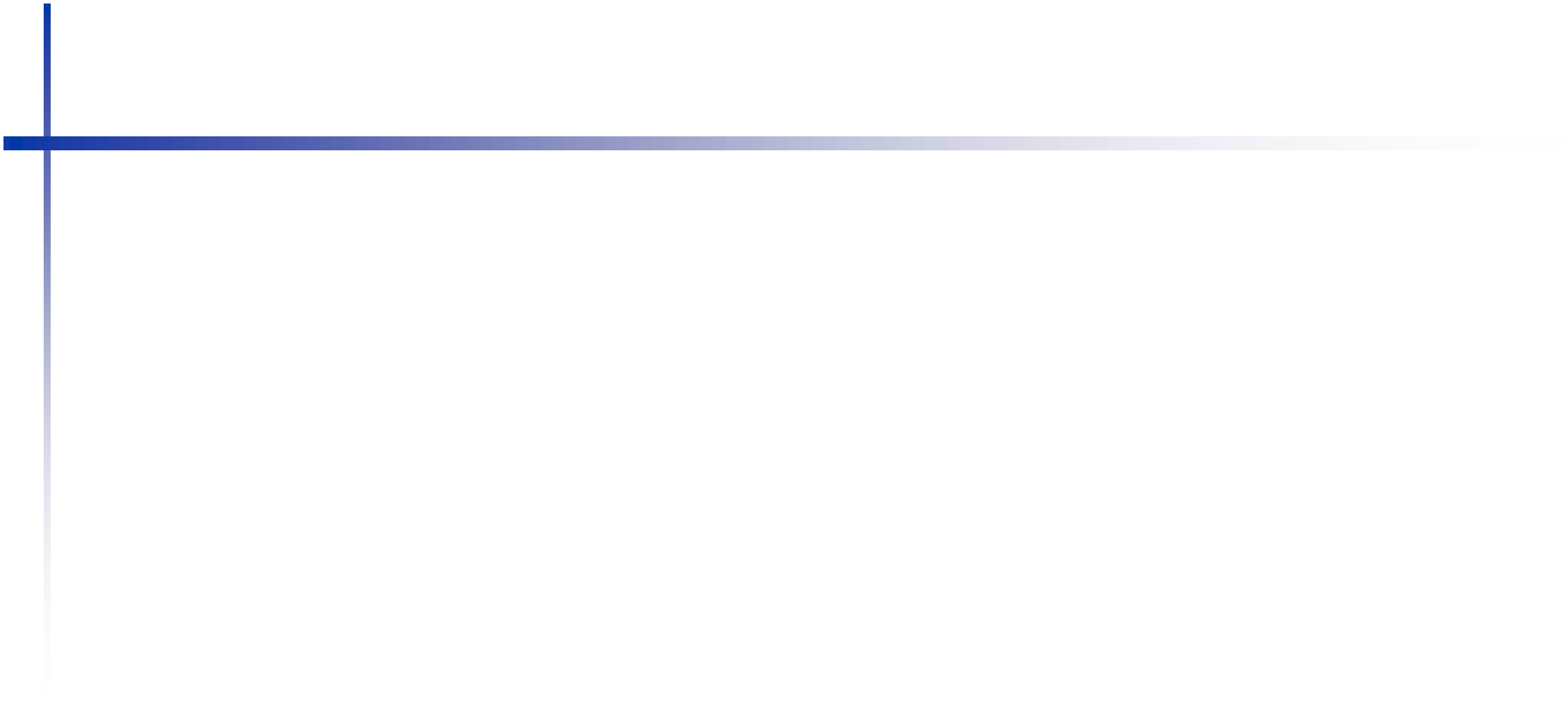
I første halvdel af 80'erne blev tre load/store arkitekturer udviklet cirka samtidigt: MIPS (Stanford), SPARC (Berkeley) og ARM (Acorn, UK). Forfatterne til COD medvirkede til de oprindelige MIPS og SPARC design.

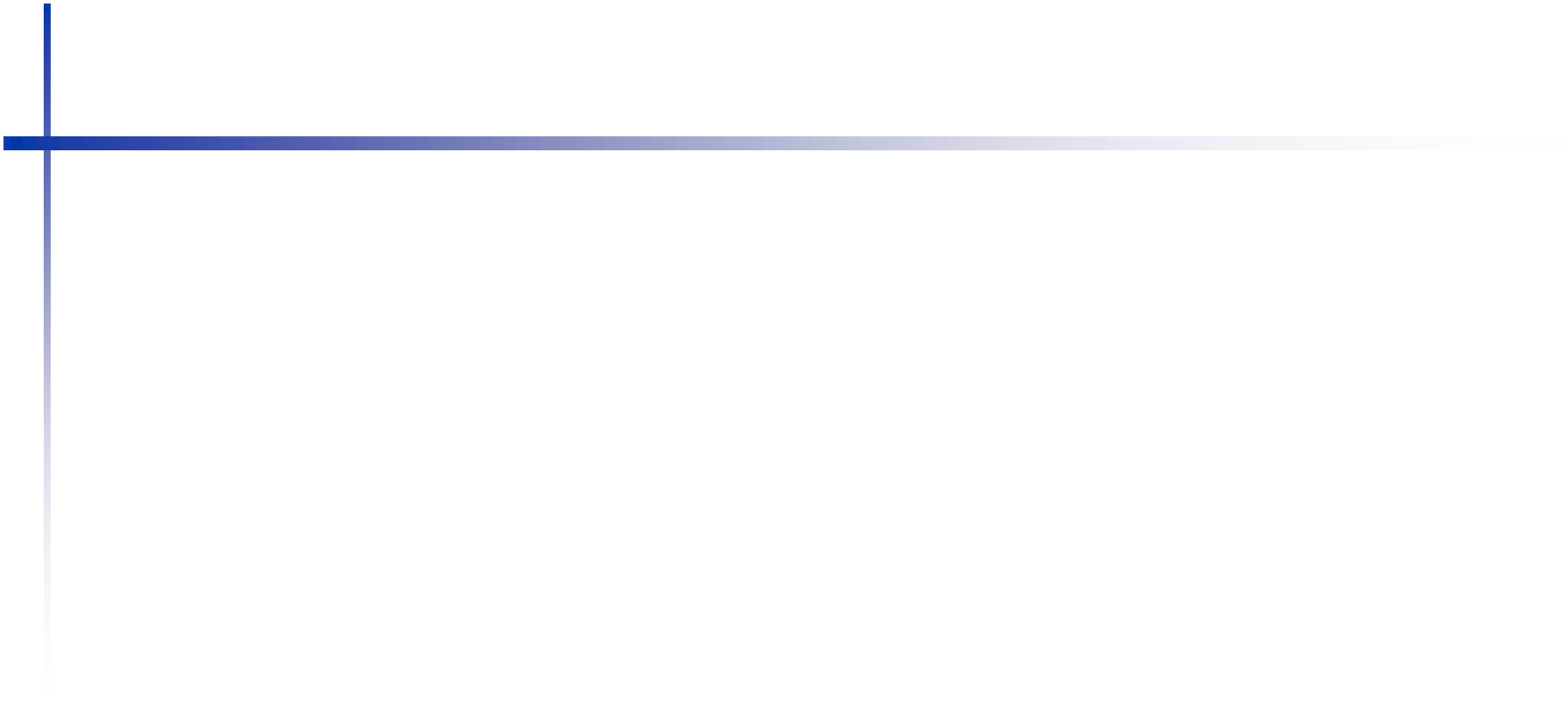
RISC-V er tættest beslægtet med MIPS. Nyere ARM arkitekturer er også meget tæt beslægtet med MIPS.

Logic Design Basics

- Information encoded in binary
- Low voltage = 0, High voltage = 1
- One wire per bit
- Multi-bit data encoded on multi-wire buses
- Combinational element
- Operate on data
- Output is a function of input
- State (sequential) elements
- Store information

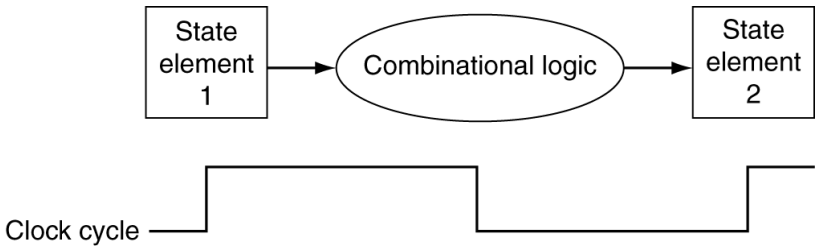


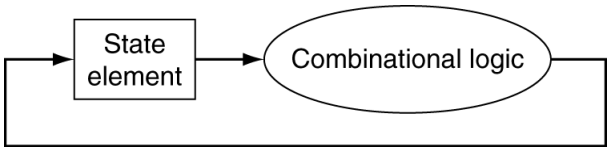




Clocking Methodology

- Combinational logic transforms data during clock cycles
- Between clock edges
- Input from state elements, output to state element
- Longest delay determines clock period





Chapter 4

The Processor

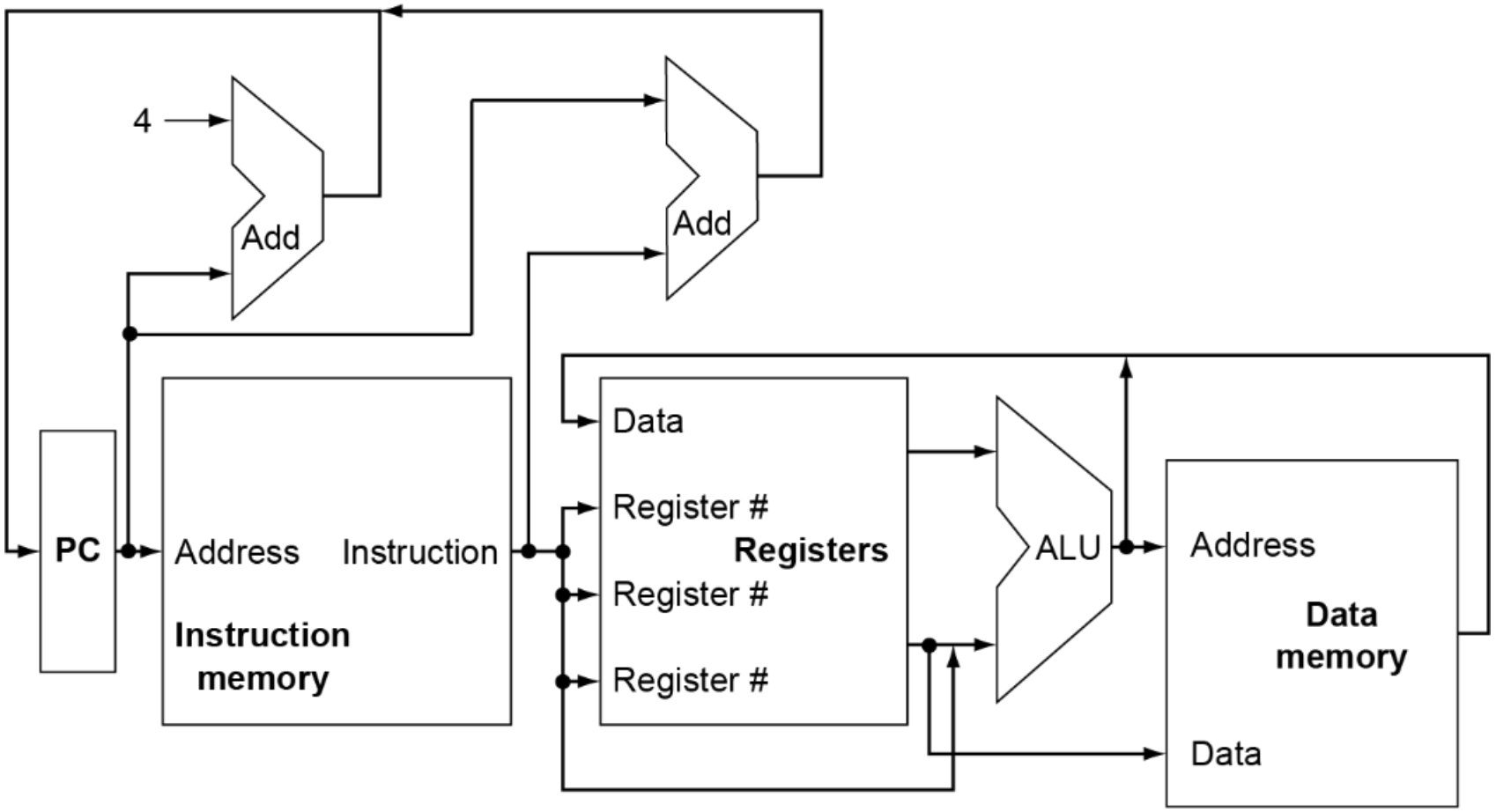
Introduction

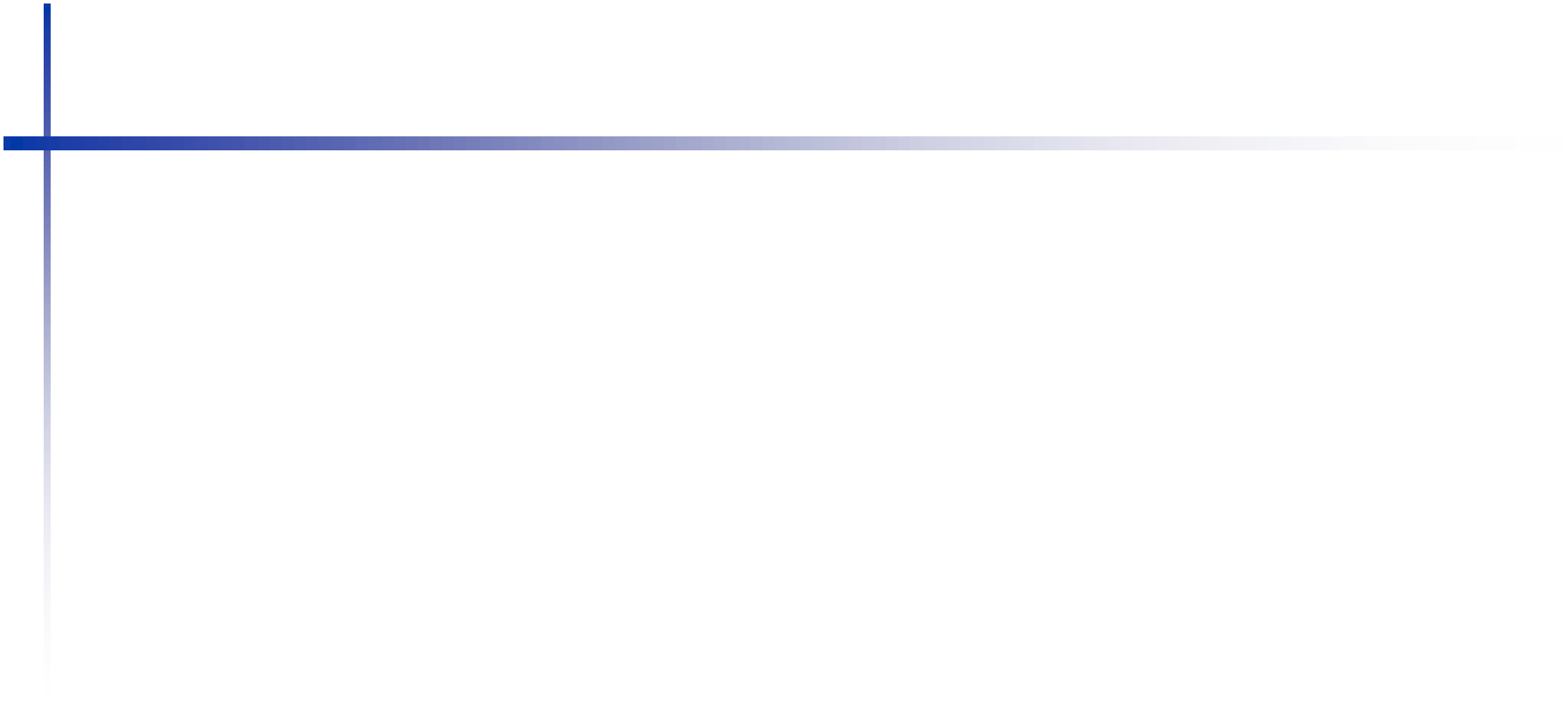
- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine a simple RISC-V implementation
  - A single-cycle design
- Simple subset, shows most aspects
  - Memory reference: `ld`, `sd`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`
  - Control transfer: `beq`

Instruction Execution

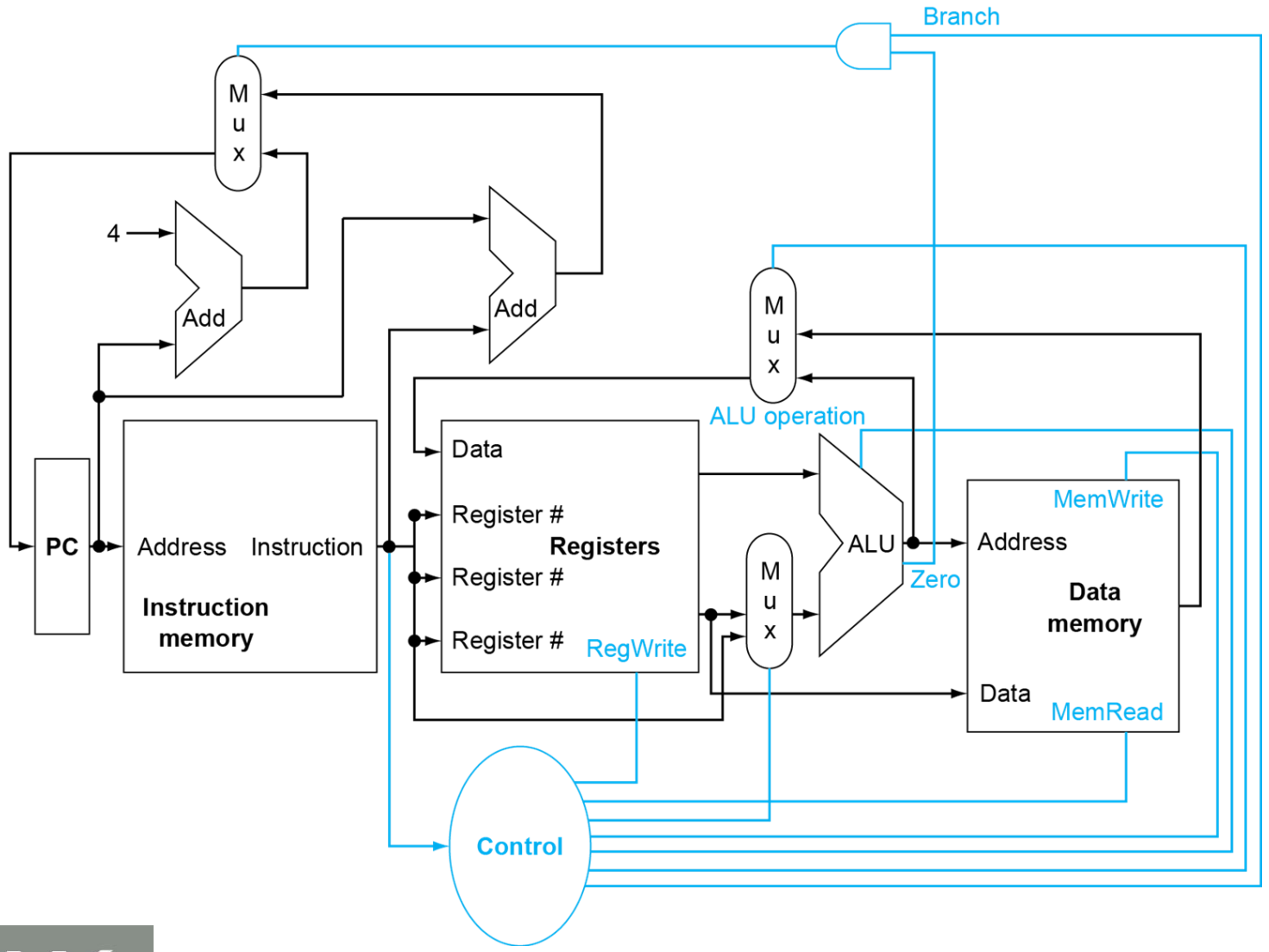
- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
- Use ALU to calculate
- Arithmetic result
- Memory address for load/store
- Branch comparison
- Access data memory for load/store
- PC ← target address or PC + 4

CPU Overview





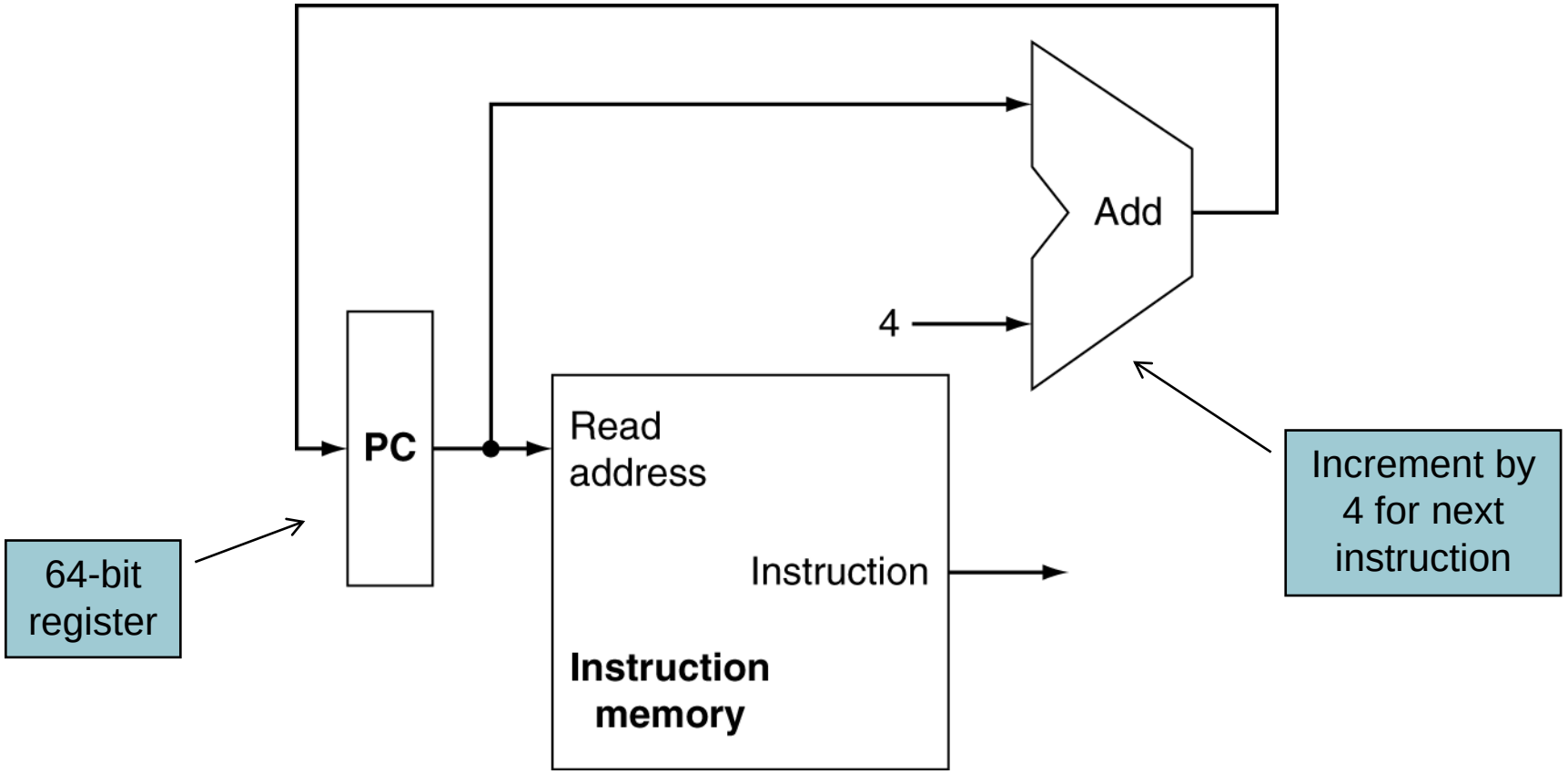
Control


Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Building a Datapath

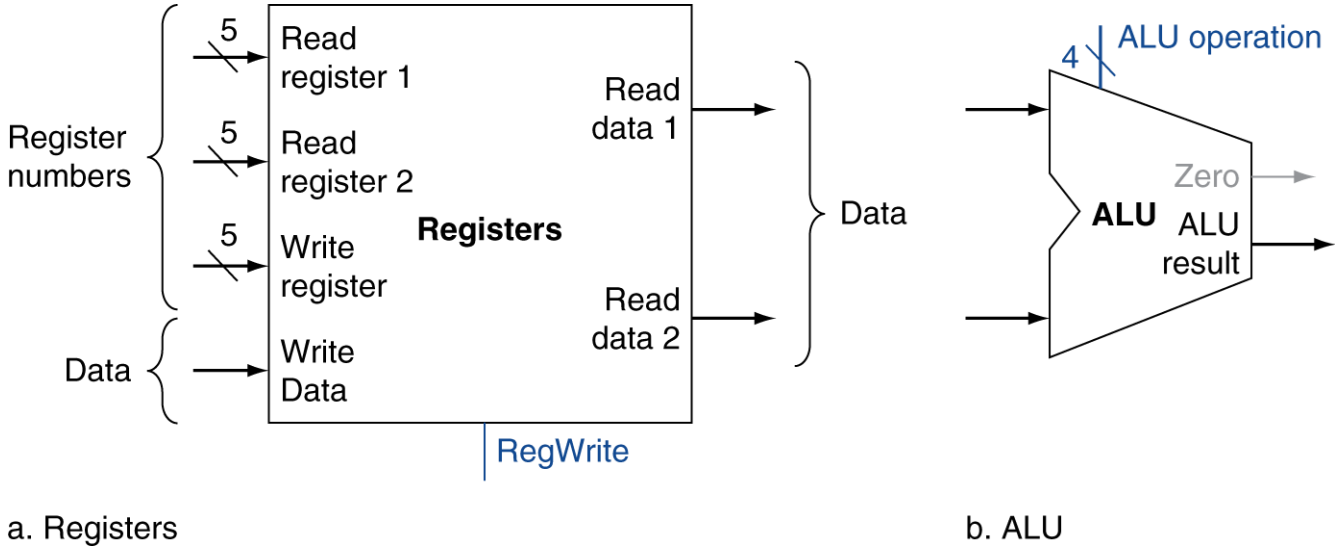
- Datapath
- Elements that process data and addresses in the CPU
- Registers, ALUs, mux's, memories, …
- We will build a RISC-V datapath incrementally
- Refining the overview design

Instruction Fetch



R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

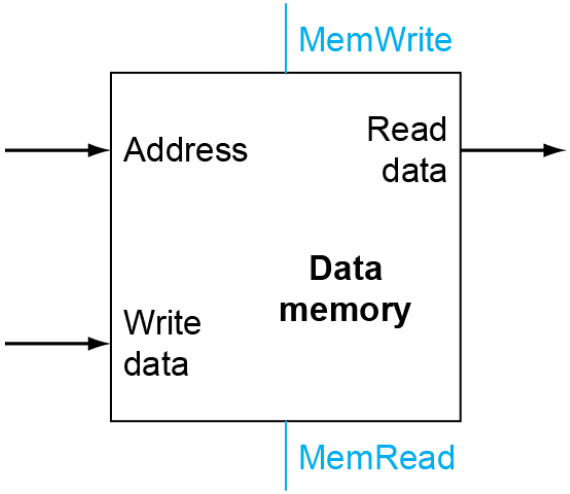


a. Registers

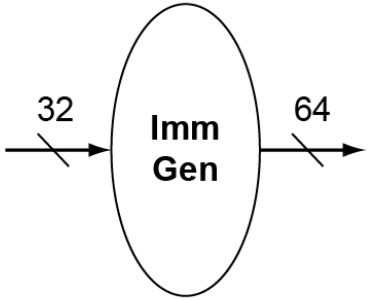
b. ALU

Load/Store Instructions

- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit

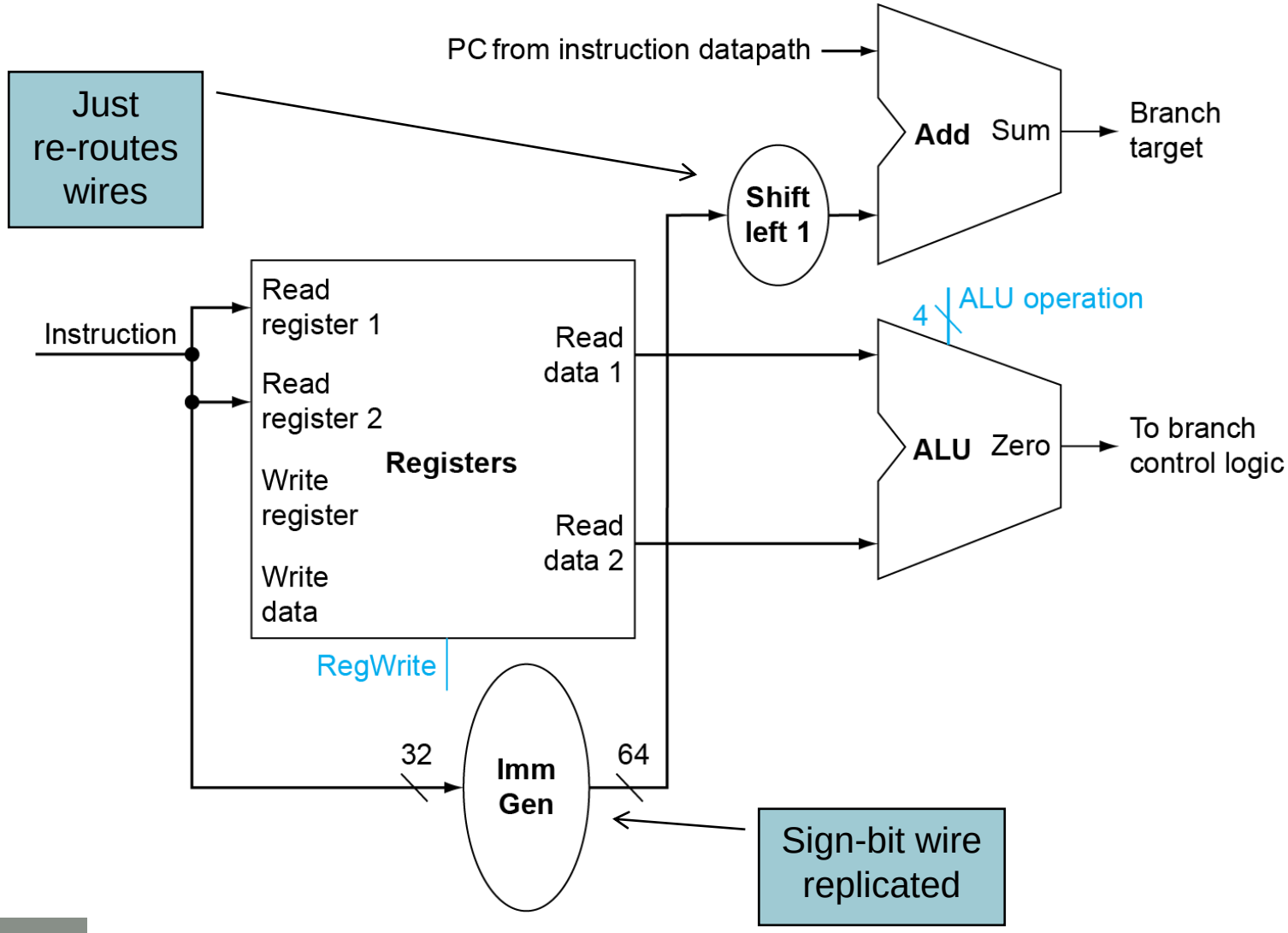


b. Immediate generation unit

Branch Instructions

- Read register operands
- Compare operands
- Use ALU, subtract and check Zero output
- Calculate target address
- Sign-extend displacement
- Shift left 1 place (halfword displacement)
- Add to PC value

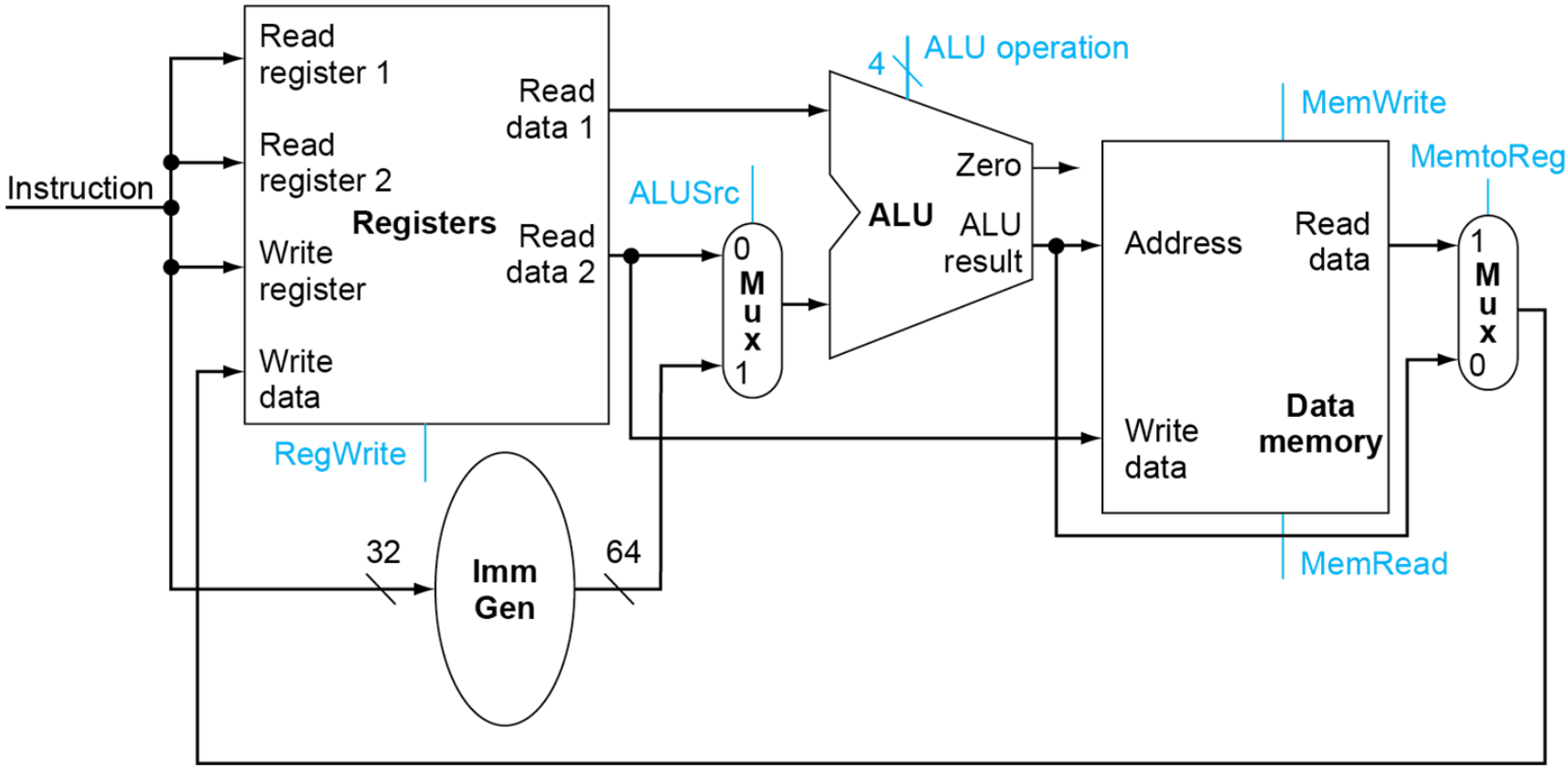
Branch Instructions



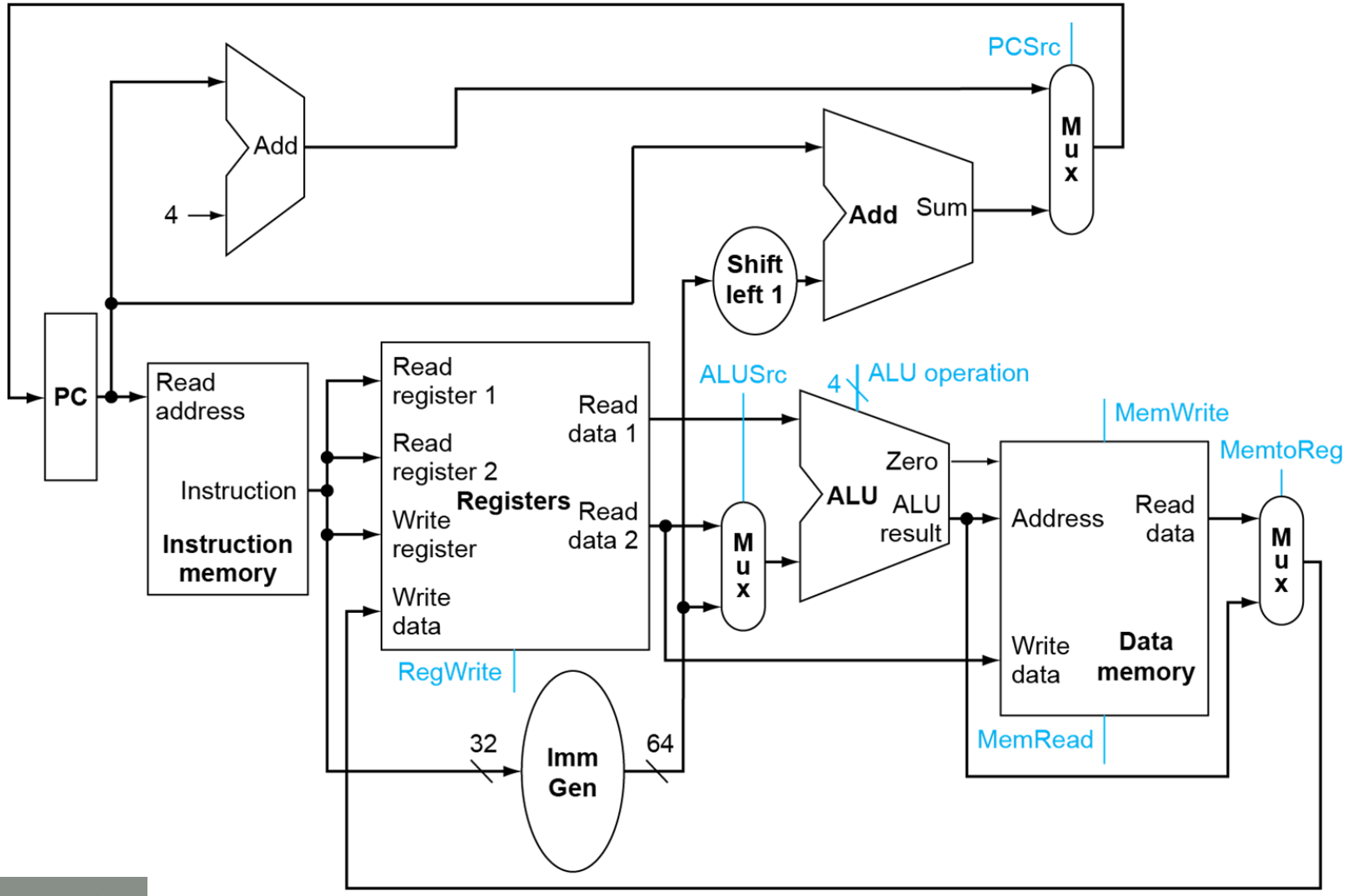
Composing the Elements

- First-cut data path does an instruction in one clock cycle
- Each datapath element can only do one function at a time
- Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



ALU Control

- ALU used for
- Load/Store: F = add
- Branch: F = subtract
- R-type: F depends on opcode

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

ALU Control

- Assume 2-bit ALUOp derived from opcode
- Combinational logic derives ALU control

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
ld	00	load register	XXXXXXXXXXX	add	0010
sd	00	store register	XXXXXXXXXXX	add	0010
beq	01	branch on equal	XXXXXXXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

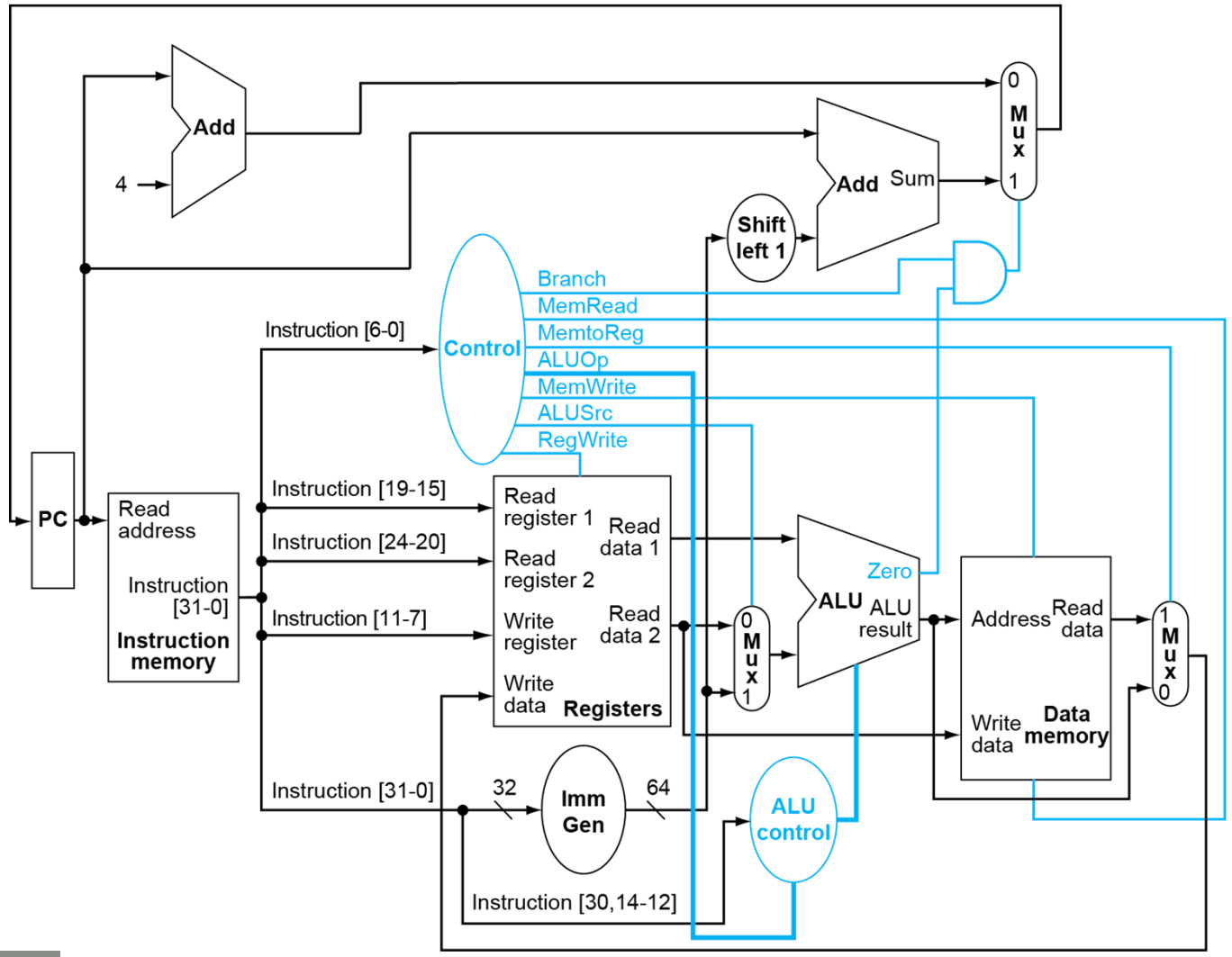
The Main Control Unit

- Control signals derived from instruction

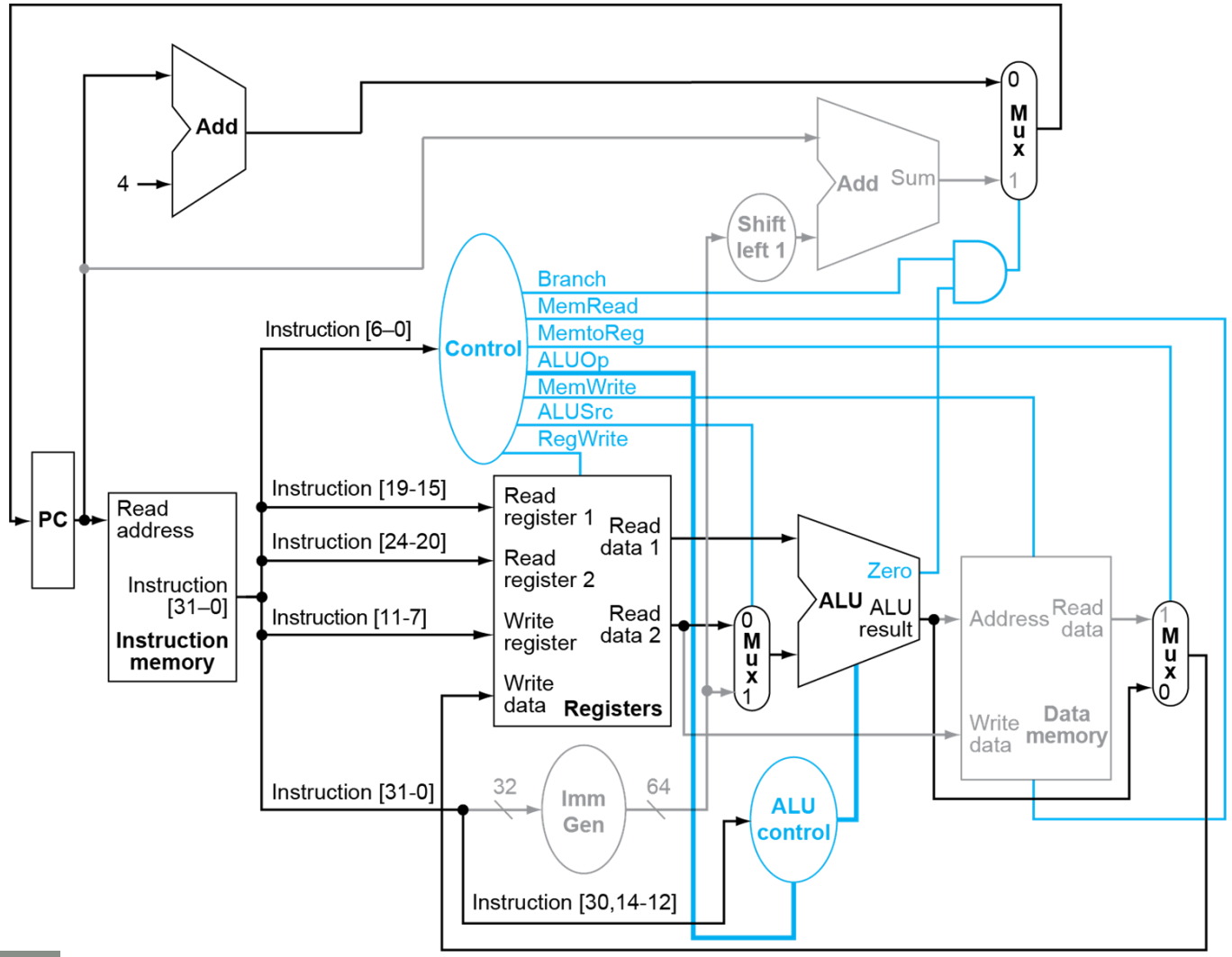
Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

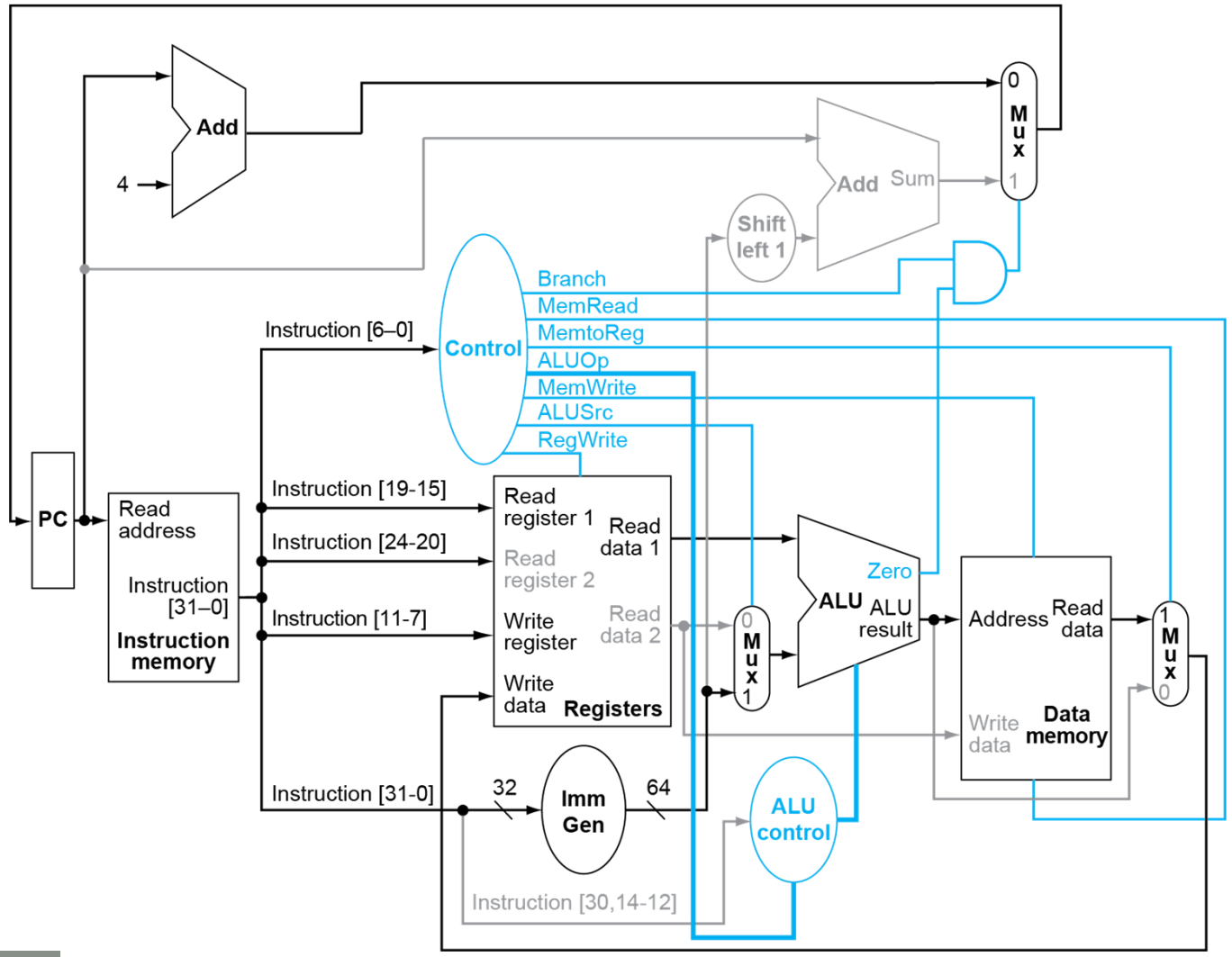
Datapath With Control



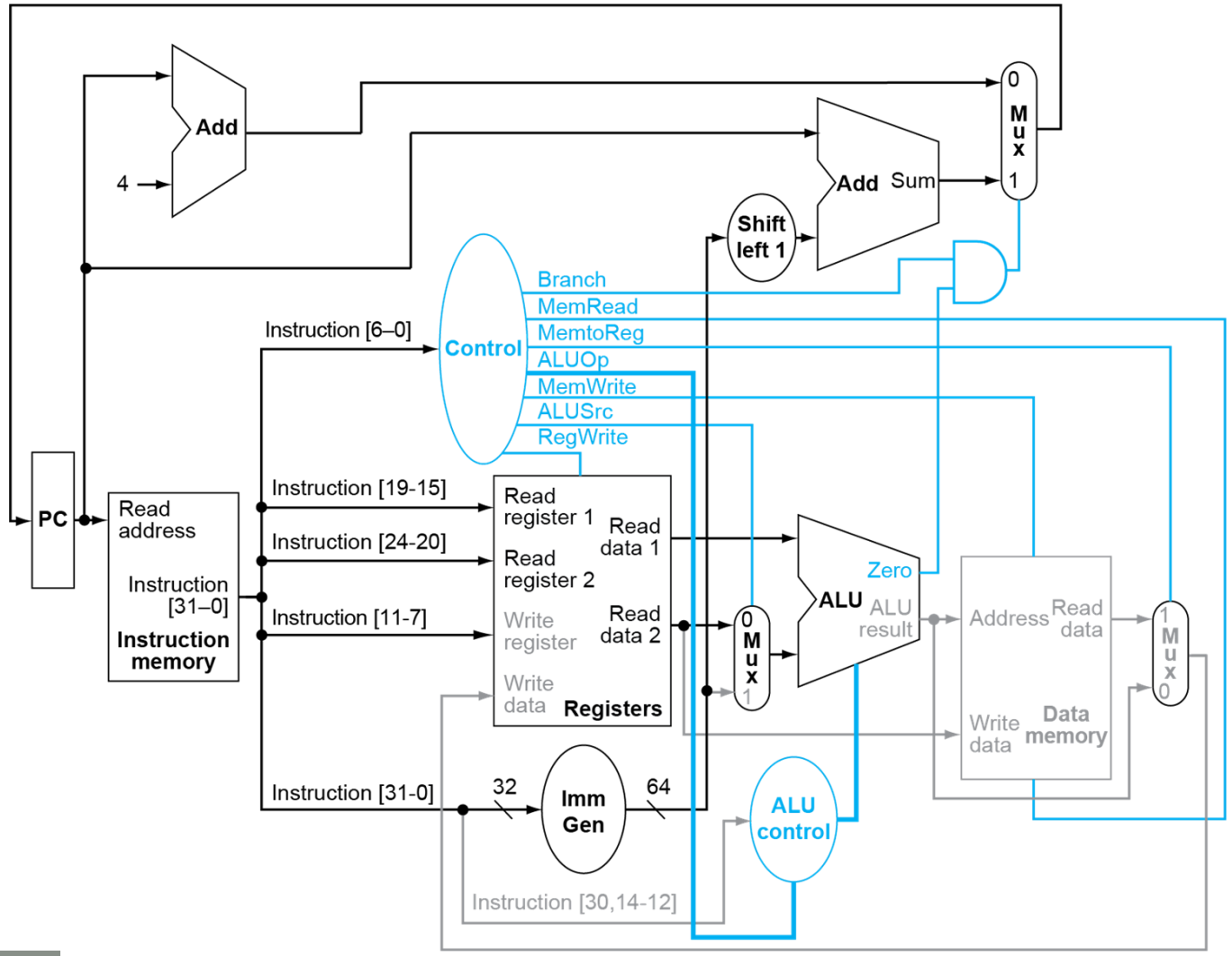
R-Type Instruction



Load Instruction



Branch-on-Equal Instruction



Performance Issues

- Longest delay determines clock period
- Critical path: load instruction
- Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
- Making the common case fast
- We will improve performance next week