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

Det afspejles i instruktions-sættet

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.

Instruktions-indkodningen balancerer
- plads til konstanter,
- plads til at angive registre og
- plads til at angive hvilken operation der er tale om

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 det 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 opdagede load/store og hvor godt det passede til compileren i sidste halvdel af halvfjerdserne (IBM 801). Glemte det igen. Genopdagede det sidst I 80'erne (PowerPC).

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 tættest beslægtet med MIPS (tættere end den ældre ARM !!).

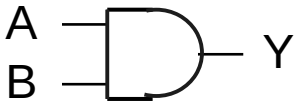
Minimal intro til digitallogisk design!

Logic Design Basics

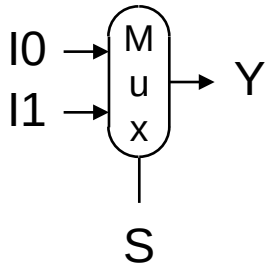
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

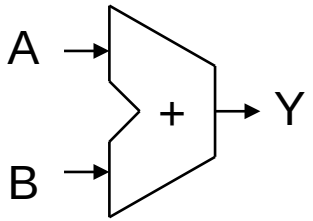
- AND-gate
  - Y = A & B



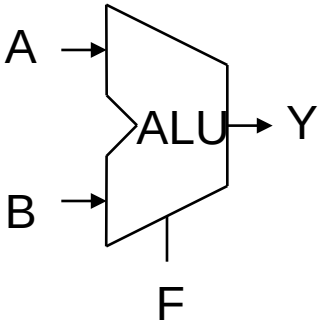
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

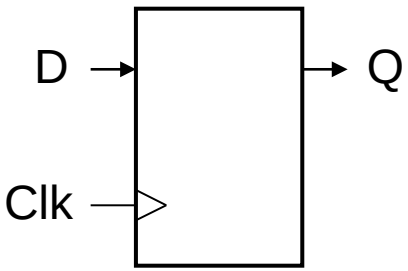


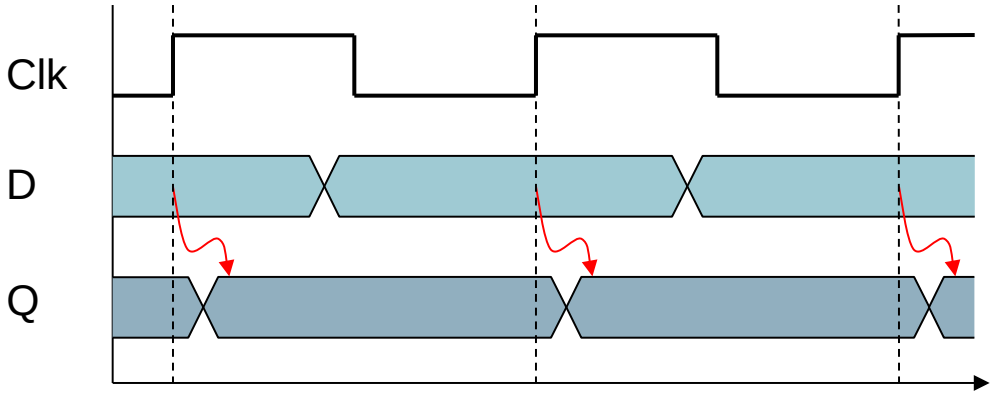
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

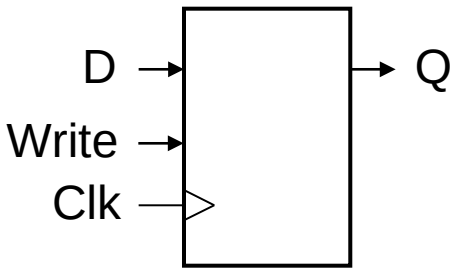
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

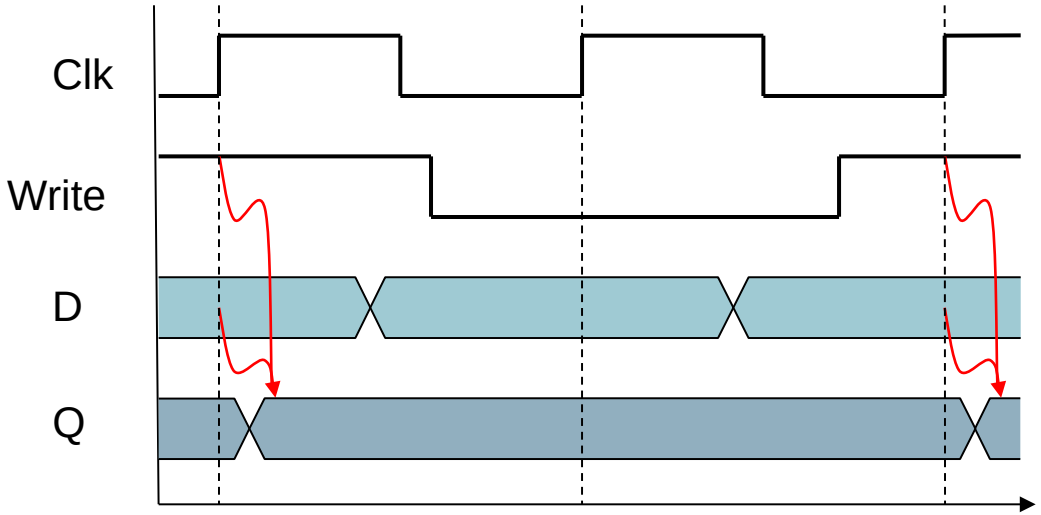




Sequential Elements

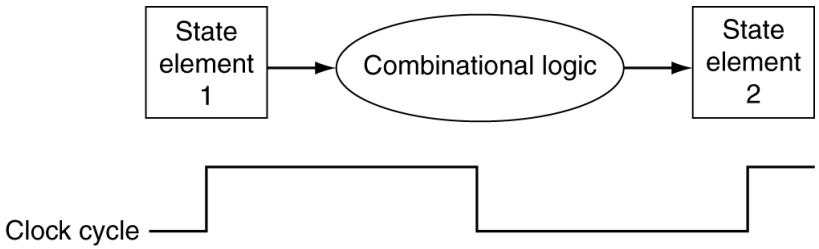
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when update is conditional

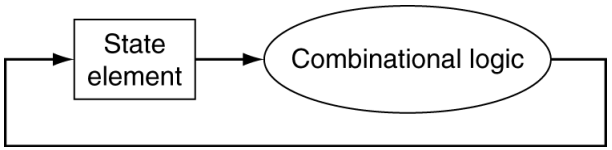




Clocking Methodology

- 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

Complete with faulty figures left over from the 64-bit MIPS edition – sorry, couldn't be bothered to fix them…

Immediate constants are 32 bit in a 32-bit processor even if some figures says otherwise :-(

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 set – selected instructions for simple datapath

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI	
imm[31:12]				rd	0010111	AUIPC	
imm[20|10:1|11|19:12]				rd	1101111	JAL	
imm[11:0]		rs1	000	rd	1100111	JALR	
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ	This one
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE	
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT	
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE	
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU	
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU	
imm[11:0]		rs1	000	rd	0000011	LB	
imm[11:0]		rs1	001	rd	0000011	LH	
imm[11:0]		rs1	010	rd	0000011	LW	This one
imm[11:0]		rs1	100	rd	0000011	LBU	
imm[11:0]		rs1	101	rd	0000011	LHU	
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB	
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH	
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW	This one
imm[11:0]		rs1	000	rd	0010011	ADDI	
imm[11:0]		rs1	010	rd	0010011	SLTI	
imm[11:0]		rs1	011	rd	0010011	SLTIU	
imm[11:0]		rs1	100	rd	0010011	XORI	
imm[11:0]		rs1	110	rd	0010011	ORI	
imm[11:0]		rs1	111	rd	0010011	ANDI	
0000000	shamt	rs1	001	rd	0010011	SLLI	
0000000	shamt	rs1	101	rd	0010011	SRLI	
0100000	shamt	rs1	101	rd	0010011	SRAI	
0000000	rs2	rs1	000	rd	0110011	ADD	This one
0100000	rs2	rs1	000	rd	0110011	SUB	This one
0000000	rs2	rs1	001	rd	0110011	SLL	
0000000	rs2	rs1	010	rd	0110011	SLT	
0000000	rs2	rs1	011	rd	0110011	SLTU	
0000000	rs2	rs1	100	rd	0110011	XOR	
0000000	rs2	rs1	101	rd	0110011	SRL	
0100000	rs2	rs1	101	rd	0110011	SRA	
0000000	rs2	rs1	110	rd	0110011	OR	This one
0000000	rs2	rs1	111	rd	0110011	AND	This one
fm | pred | succ		rs1	000	rd	0001111	FENCE	
000000000000		00000	000	00000	1110011	ECALL	
000000000001		00000	000	00000	1110011	EBREAK	

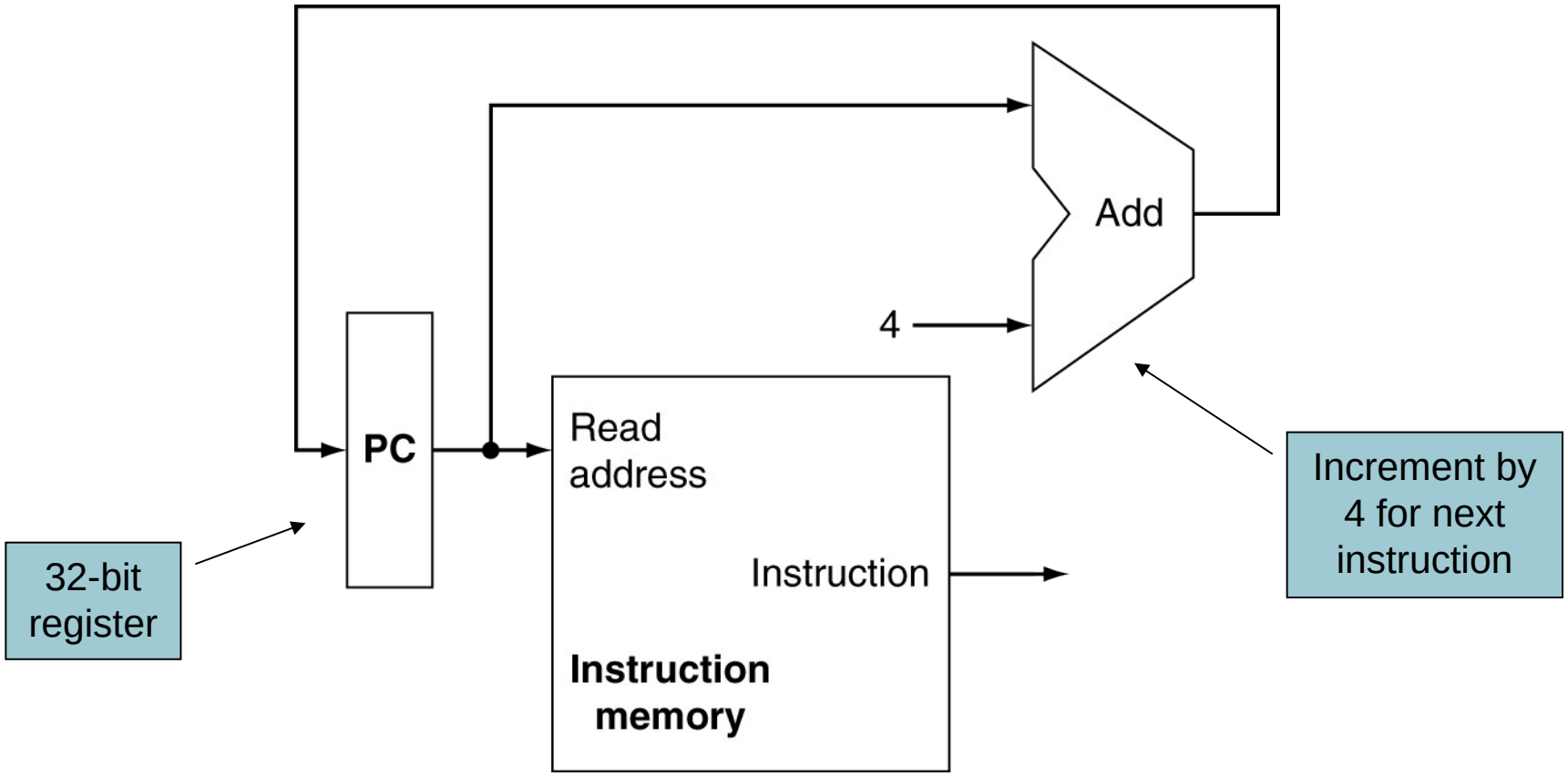
Instruction Execution

- 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

Building a Datapath

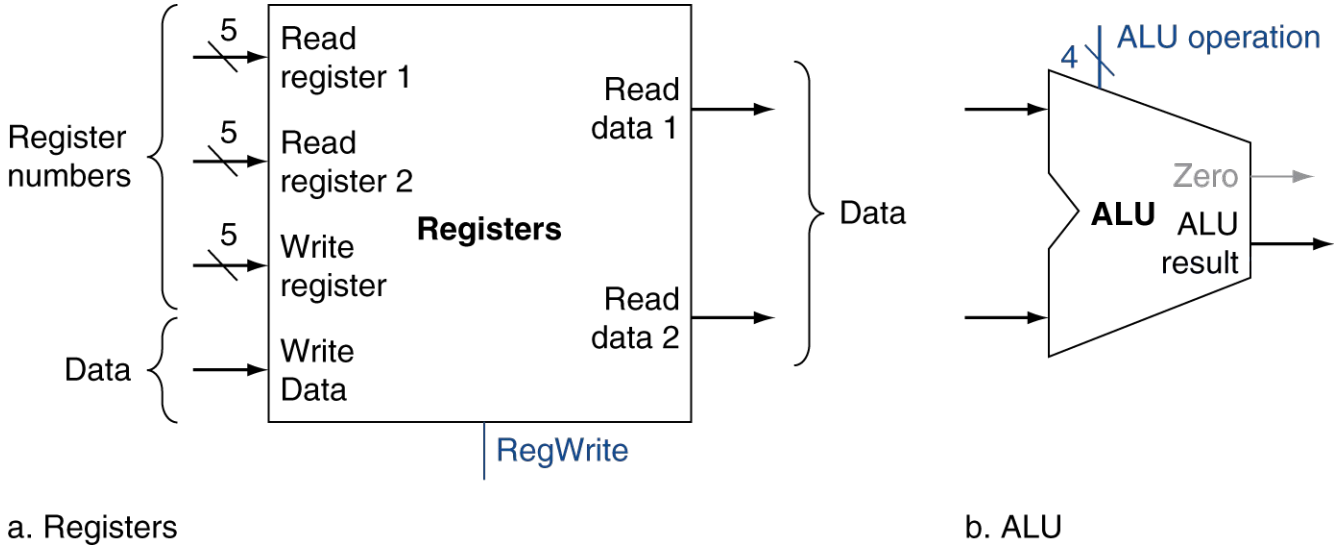
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a RISC-V datapath incrementally

Instruction Fetch



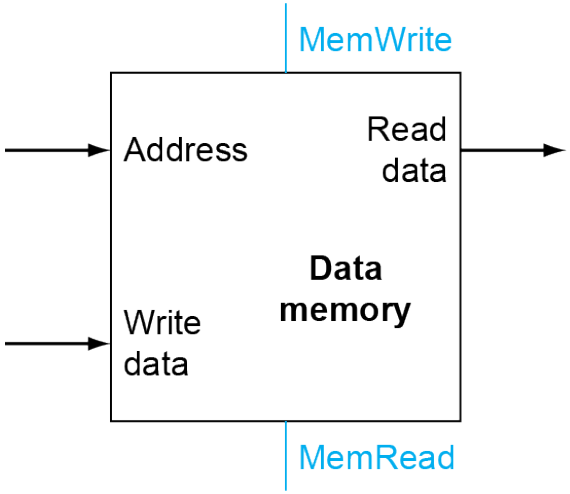
R-Format Instructions

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

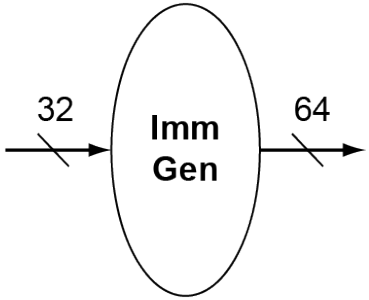


Load/Store Instructions

- 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

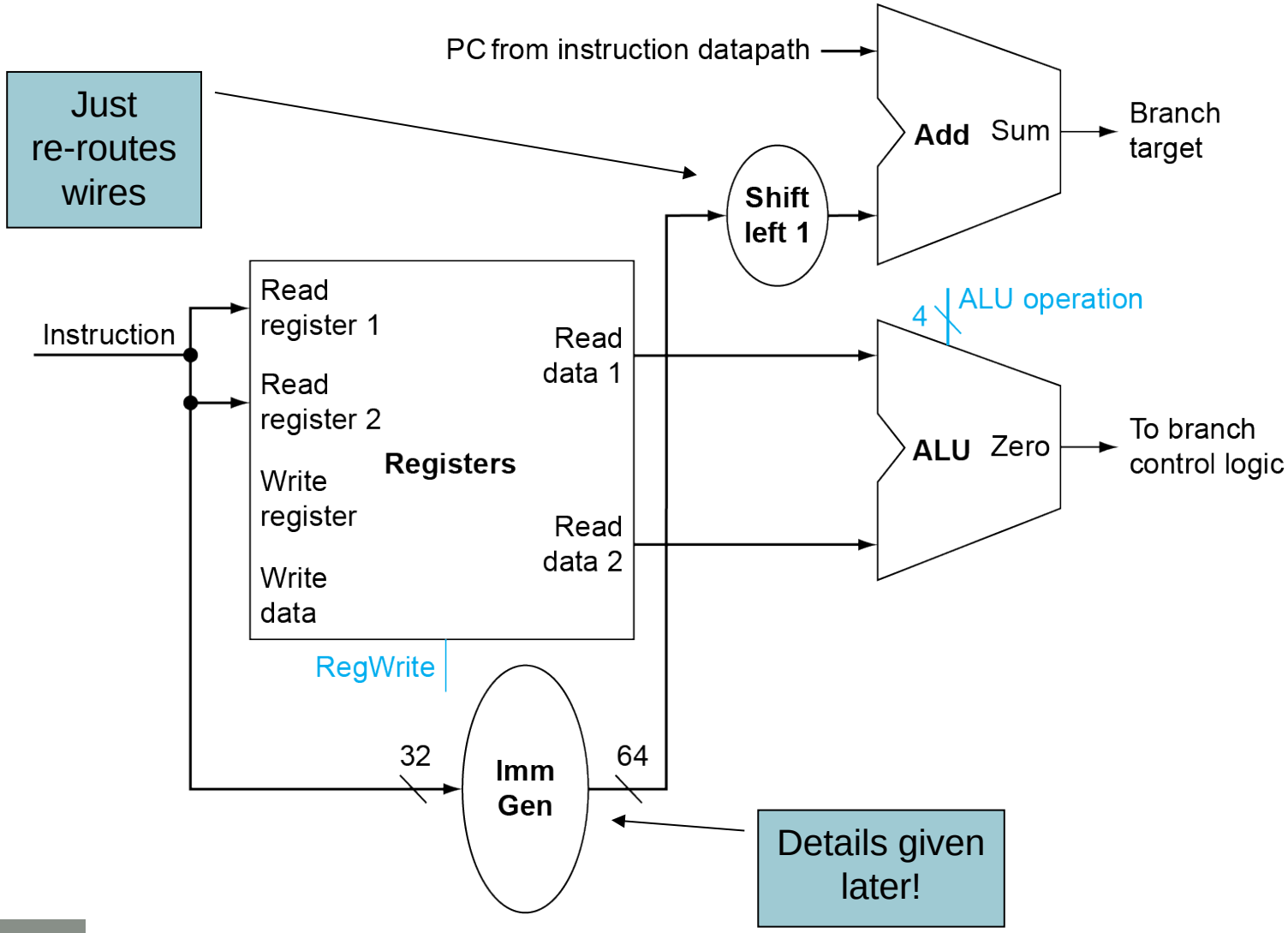


b. Immediate generation unit

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

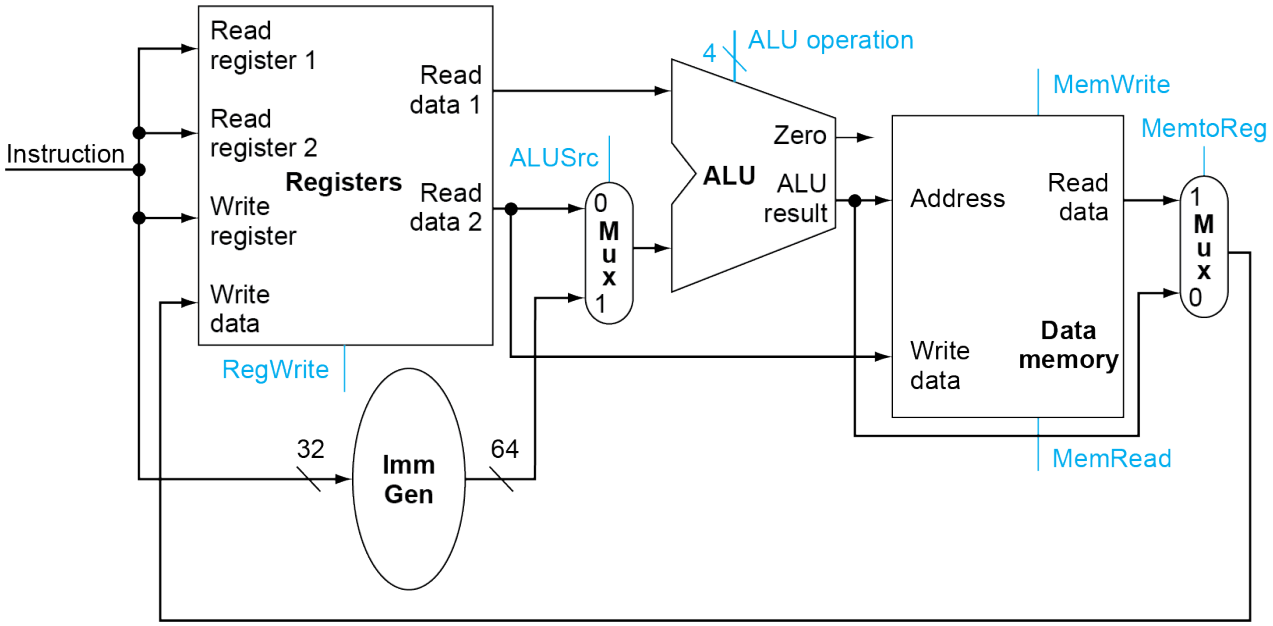
Branch Instructions



Composing the Elements

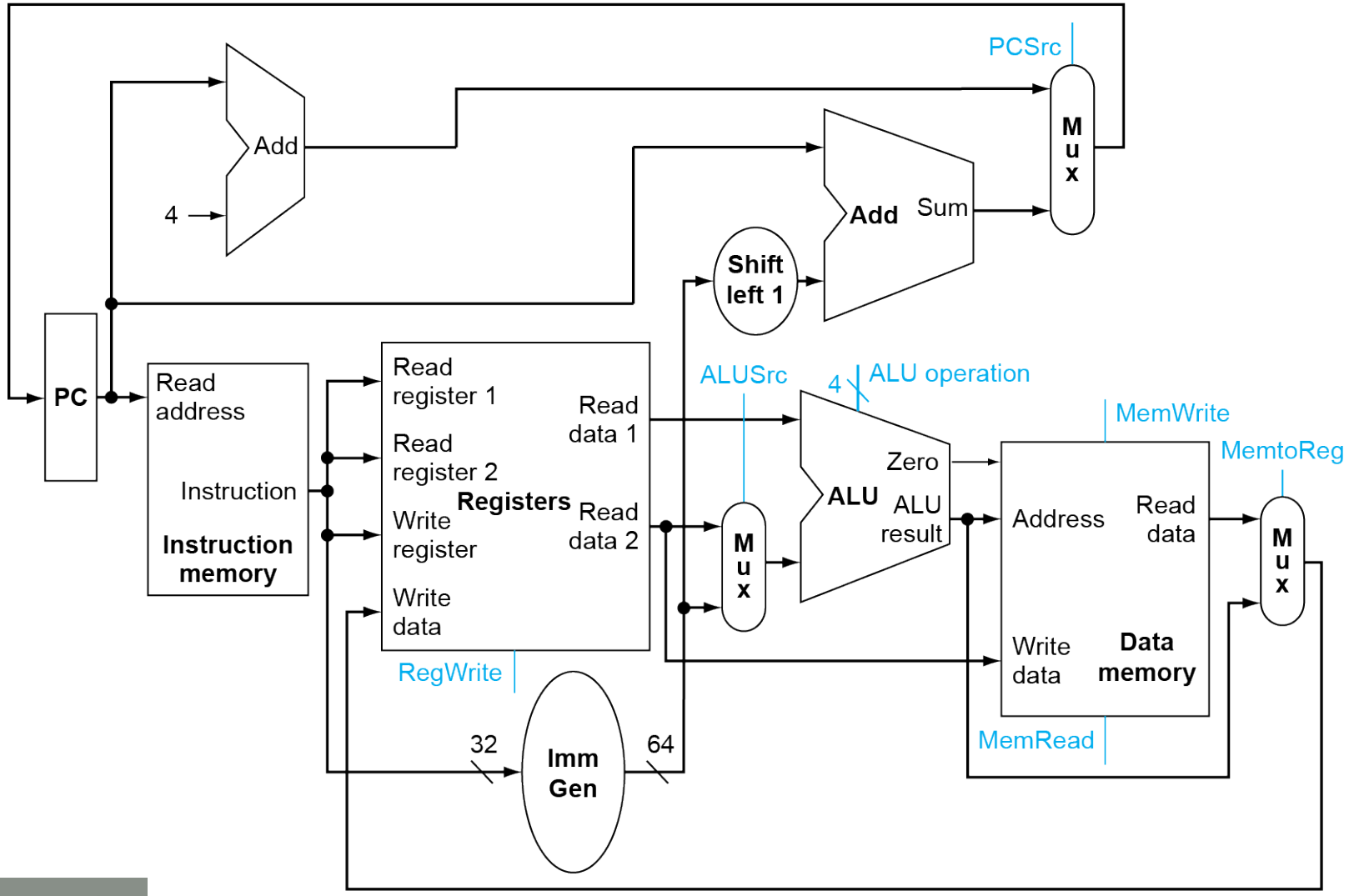
- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

Encoding vs Datapath

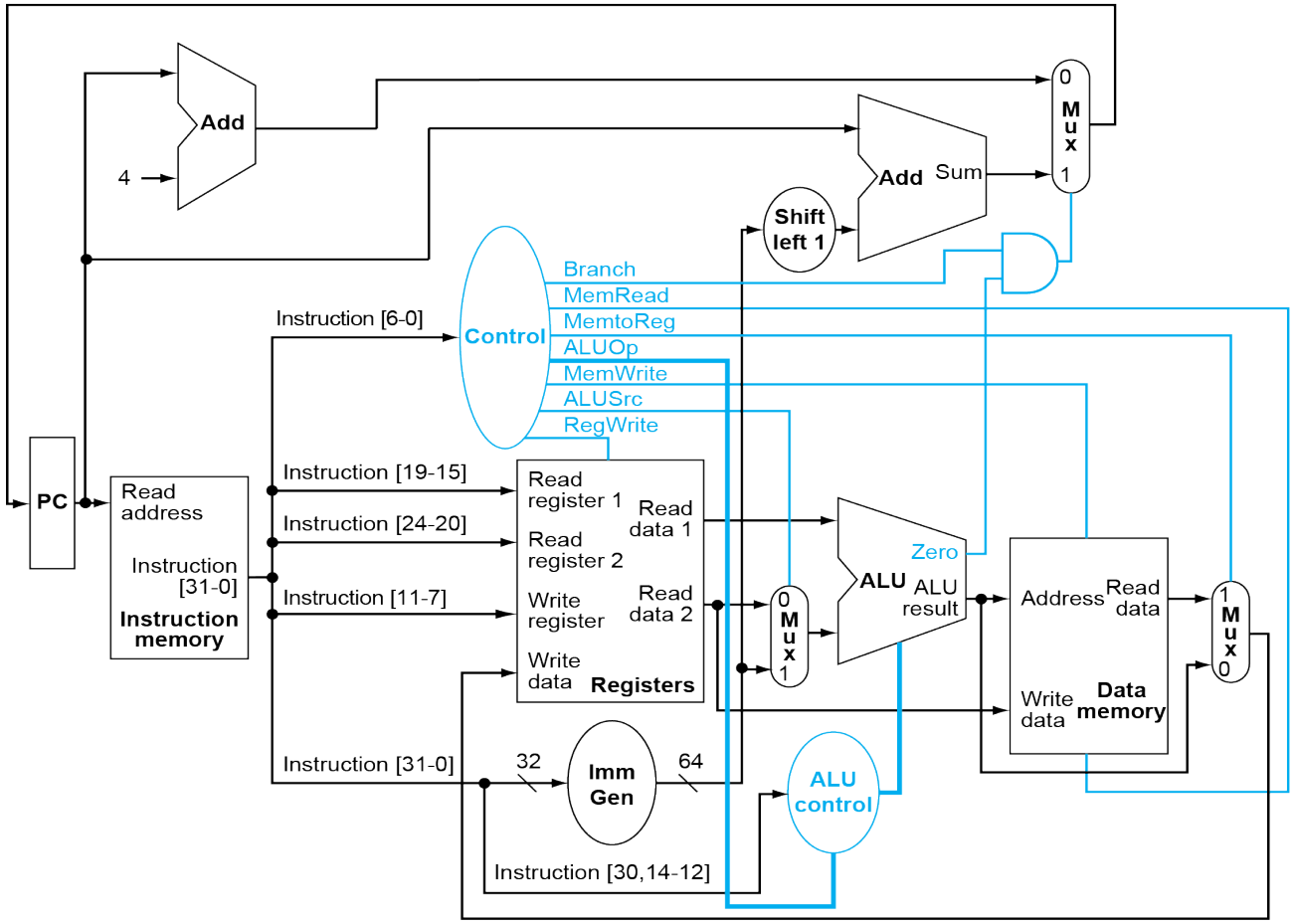


31–25	24–21	20	19–15	14–12	11–8	7	6–0	
funct7	rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]			rs1	funct3	rd		opcode	I-type
imm[11:5]	rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]					rd		opcode	U-type
imm[20]	imm[10:1]	imm[11]	imm[19:12]		rd		opcode	J-type

Full Datapath



Datapath With Control



Control is in 2 stages, (Primary) Control and ALU-control
Primary Control works from the opcode (bit 6-0) only

The Main Control Unit

- Control signals derived from instruction

Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Bug: “sd” should be “sw”, “ld” should be “lw”
(64 bit vs 32 bit again)

ALU Control

- ALU Op given from primary control:
  - ALUOp=00 Load/Store: Func = add
  - ALUOp=01 Branch: Func = subtract
  - ALUOp=10 R-type: Func depends on other fields

Control inputs to the ALU block – to be generated

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	funct7+3 fields	ALU function	ALU control
ld	00	load register	XXXXXXX XXX	add	0010
sd	00	store register	XXXXXXX XXX	add	0010
beq	01	branch on equal	XXXXXXX XXX	subtract	0110
R-type	10	add	0000000 000	add	0010
		subtract	0100000 000	subtract	0110
		AND	0000000 111	AND	0000
		OR	0000000 110	OR	0001

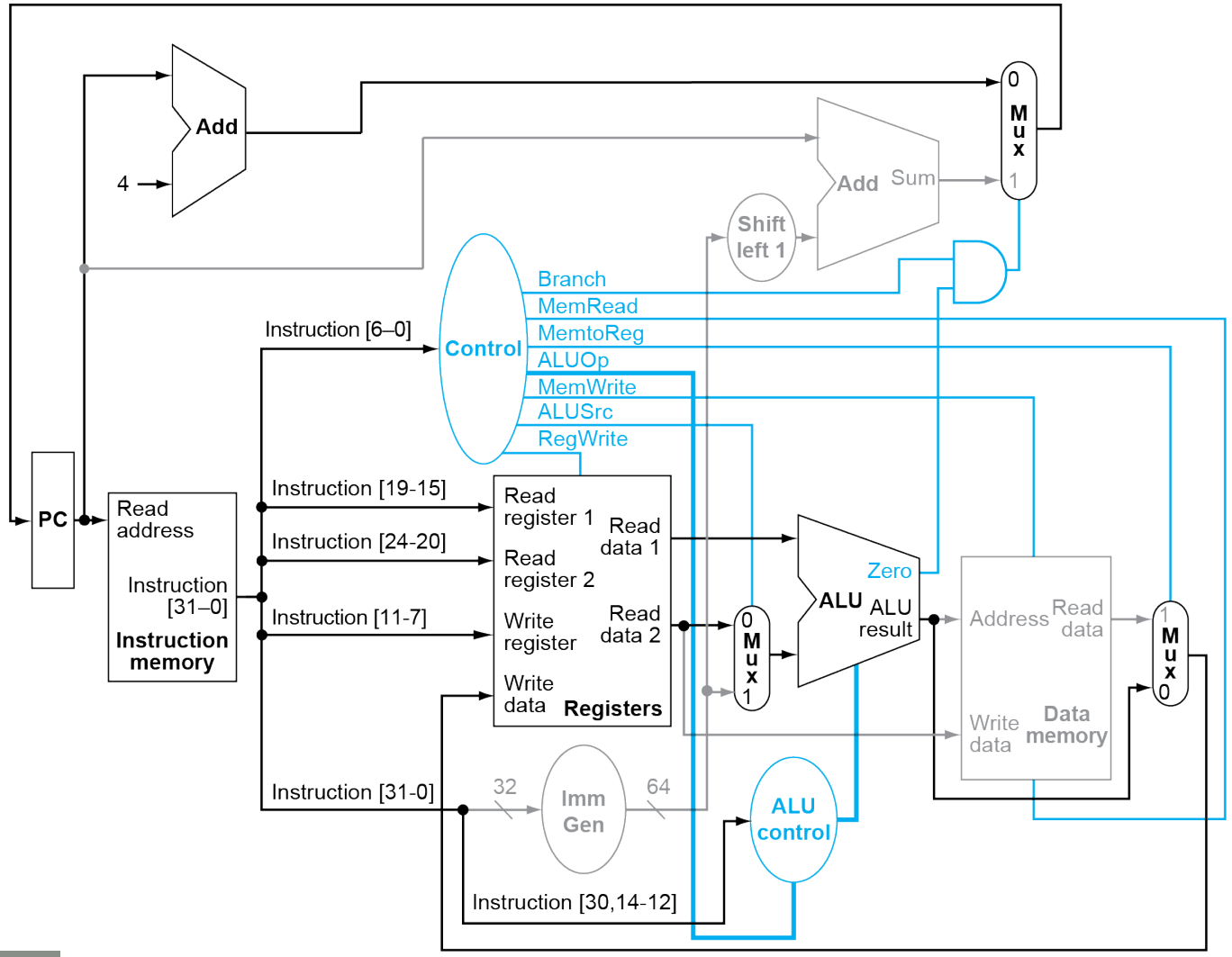
ALU Control

- Control signals derived from instruction and ALUOp field

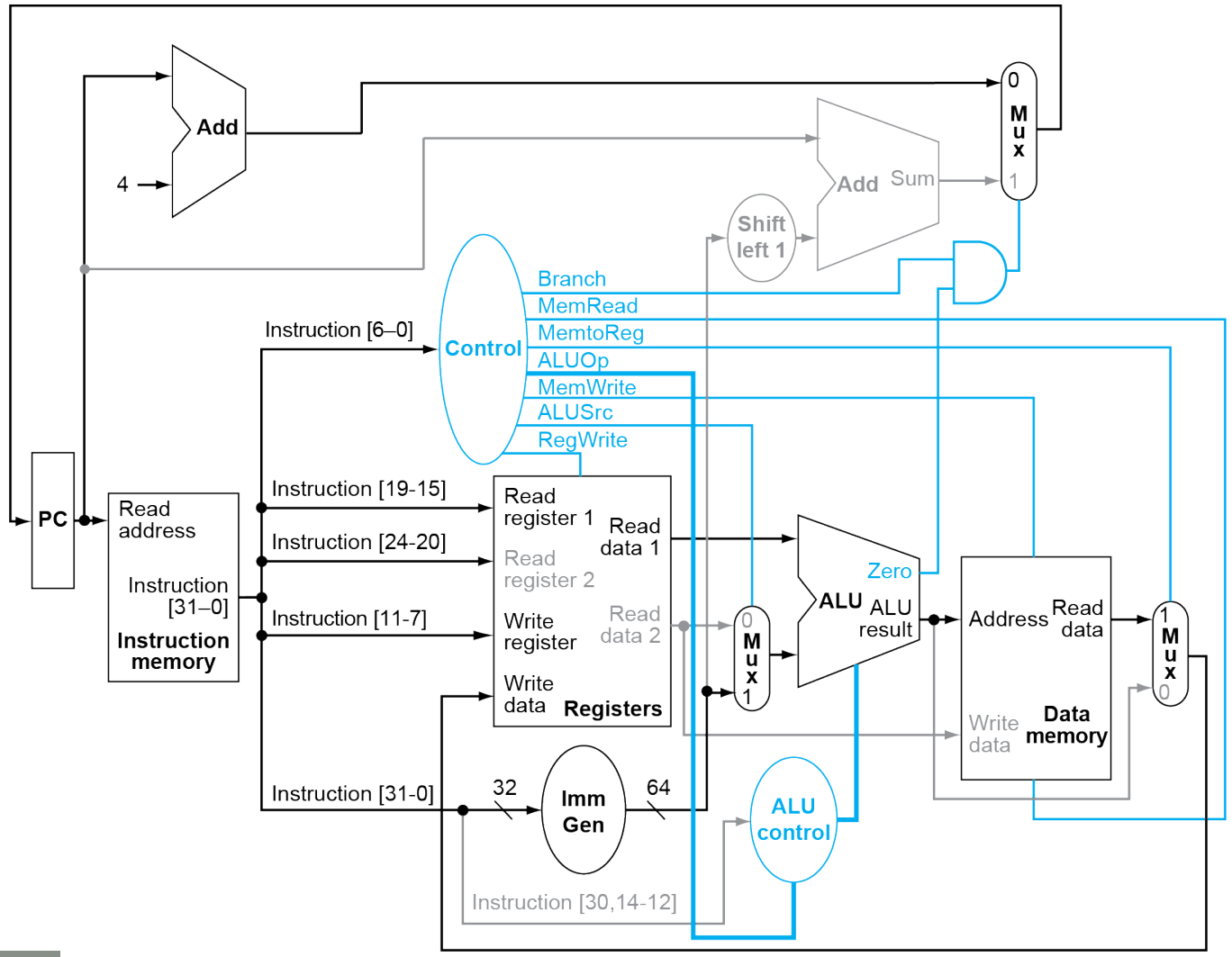
Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

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

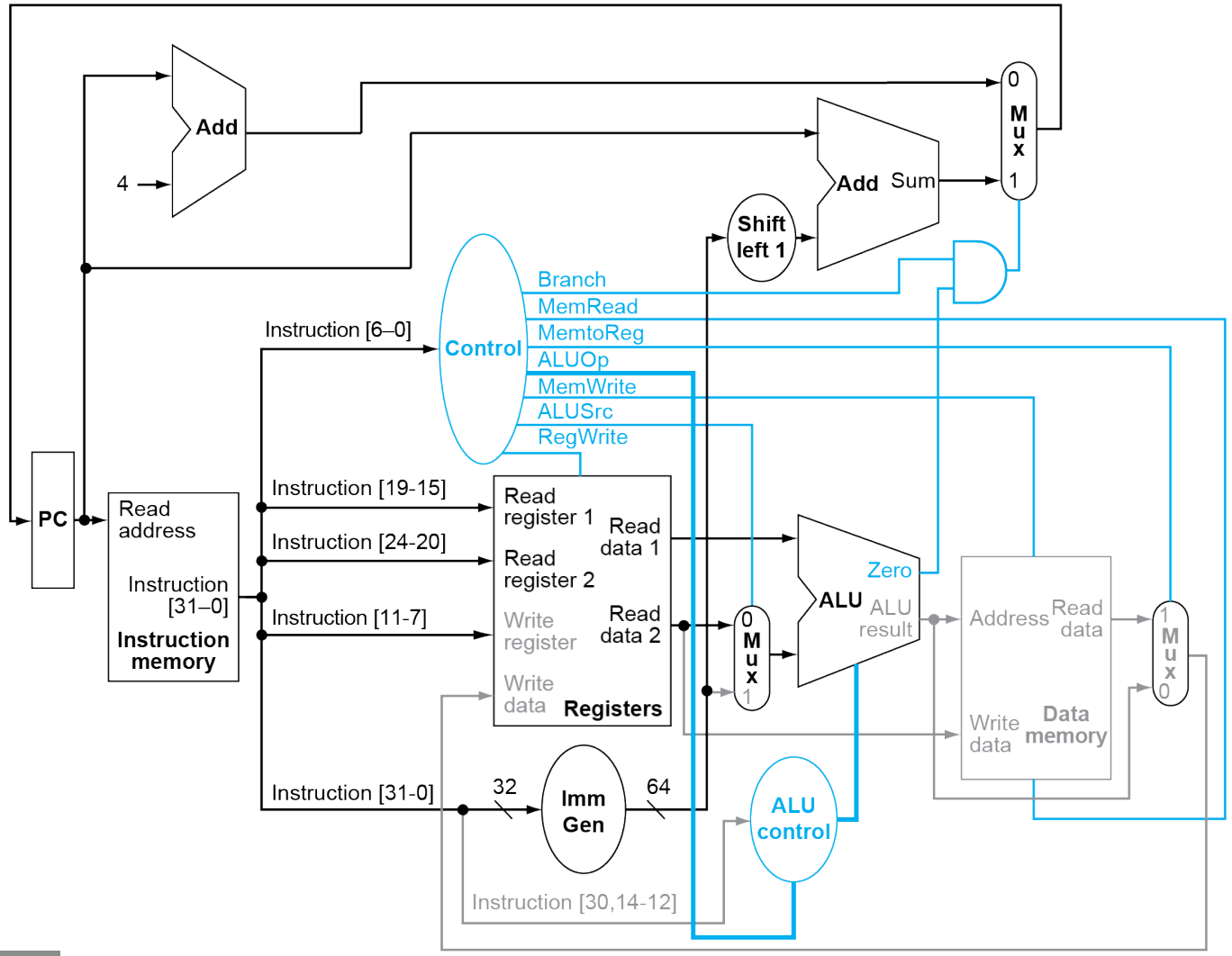
R-Type Instruction Exec



Load Instruction Exec



Branch-on-Equal Instr. Exec



Immediate Generator

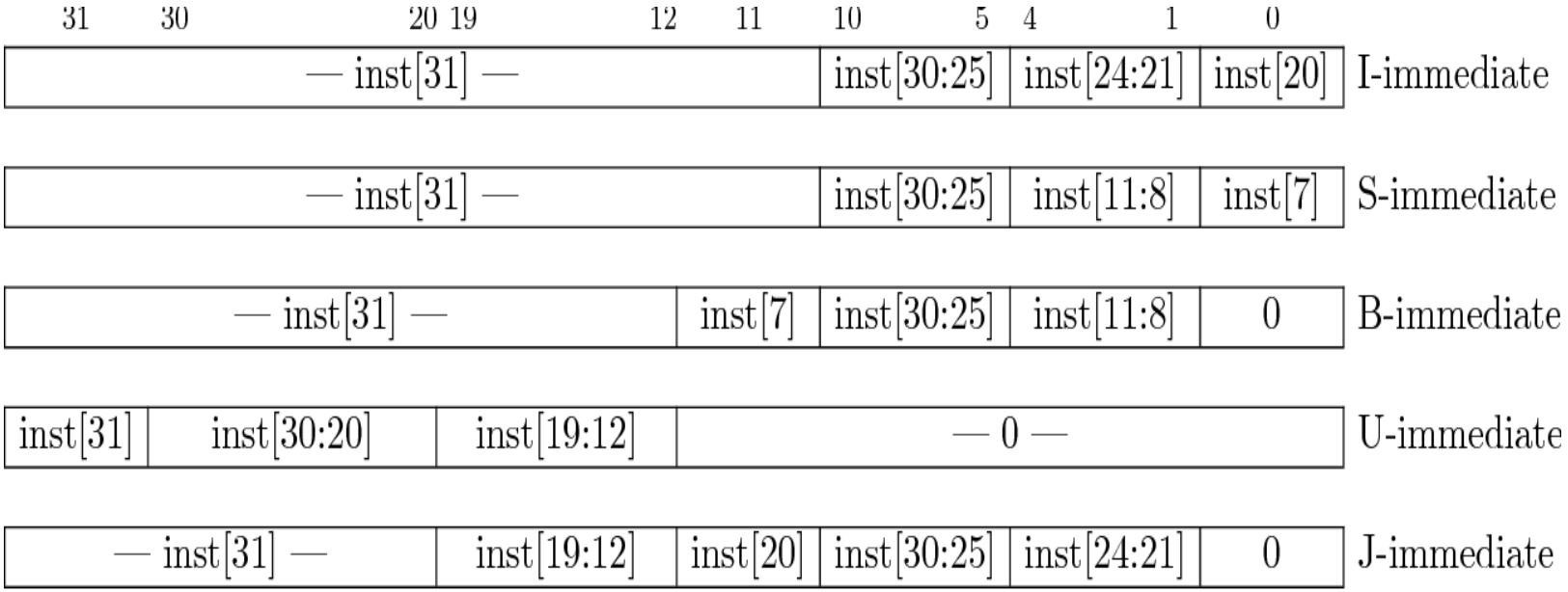


Fig 2.4, Risc-v specification – use this to understand how immediate values are decoded, not the figures from COD

The Primary Control can determine instruction format from opcode and use it to control the immediate generator. This is not shown in COD. Instead, one is left to assume another decoder exist inside the immediate generator.

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