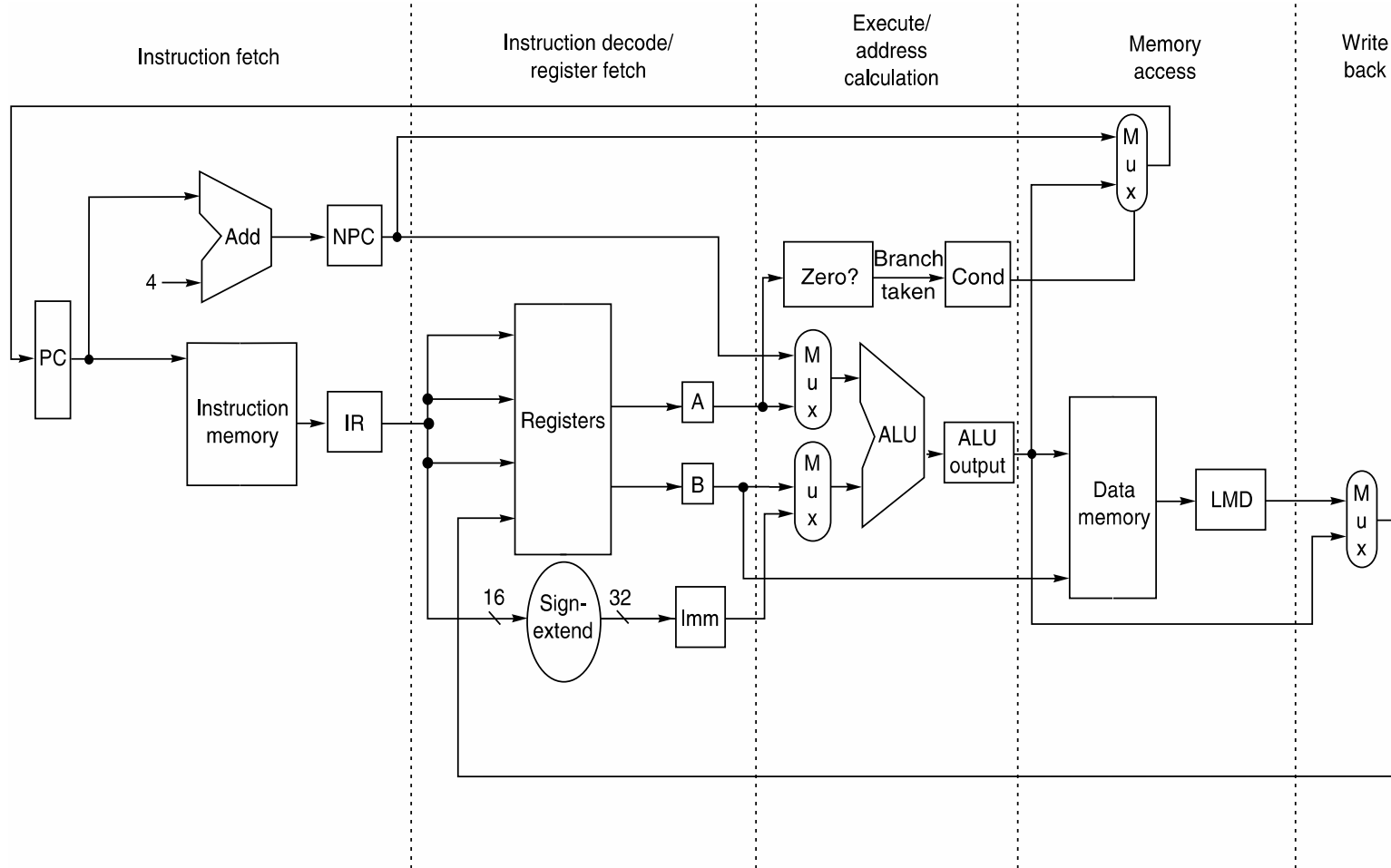




RISC-V Design

A non-pipelined RISC architecture

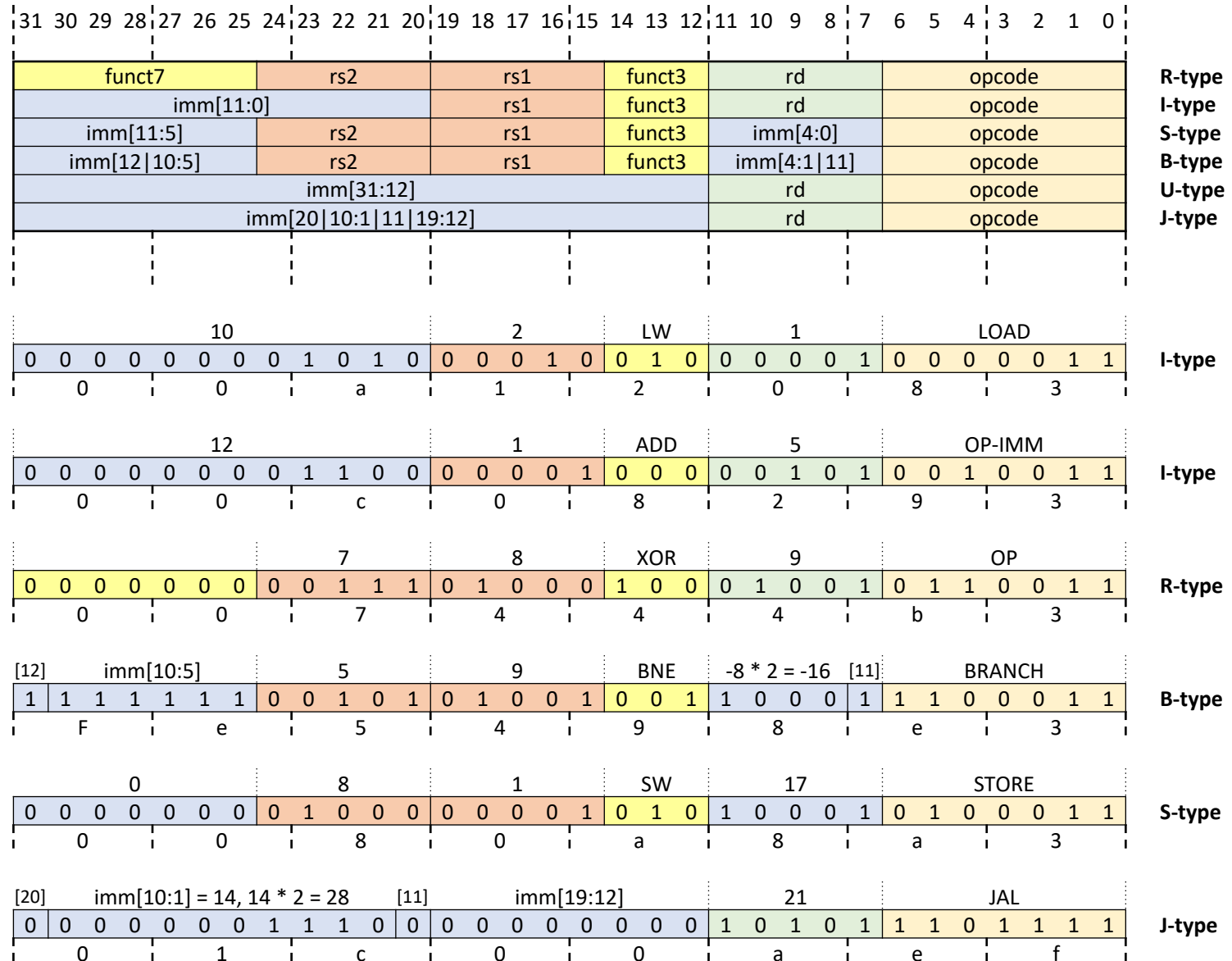


Opcodes

- ▶ Defines the “opcode” for various classes of operations
- ▶ For 32-bit instruction words, the 2 LSBs are always 11

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111
00	LOAD	LOAD-FP	<i>custom-0</i>	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	<i>custom-1</i>	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	<i>reserved</i>	<i>custom-2/rv128</i>	48b
11	BRANCH	JALR	<i>reserved</i>	JAL	SYSTEM	<i>reserved</i>	<i>custom-3/rv128</i>	>= 80b

LUI	imm[31:12]				rd	0110111	U-type
AUIPC	imm[31:12]				rd	0010111	U-type
JAL	imm[20 10:1 11 19:12]				rd	1101111	J-type
JALR	imm[11:0]	rs1	000		rd	1100111	I-type
BEQ	imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	B-type
BNE	imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	B-type
BLT	imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	B-type
BGE	imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	B-type
BLTU	imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	B-type
BGEU	imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	B-type
LB	imm[11:0]		rs1	000	rd	0000011	I-type
LH	imm[11:0]		rs1	001	rd	0000011	I-type
LW	imm[11:0]		rs1	010	rd	0000011	I-type
LBU	imm[11:0]		rs1	100	rd	0000011	I-type
LHU	imm[11:0]		rs1	101	rd	0000011	I-type
SB	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S-type
SH	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S-type
SW	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S-type
ADDI	imm[11:0]		rs1	000	rd	0010011	I-type
SLTI	imm[11:0]		rs1	010	rd	0010011	I-type
SLTIU	imm[11:0]		rs1	011	rd	0010011	I-type
XORI	imm[11:0]		rs1	100	rd	0010011	I-type
ORI	imm[11:0]		rs1	110	rd	0010011	I-type
ANDI	imm[11:0]		rs1	111	rd	0010011	I-type
SLLI	0000000	shamt	rs1	001	rd	0010011	R-type
SRLI	0000000	shamt	rs1	101	rd	0010011	R-type
SRAI	0100000	shamt	rs1	101	rd	0010011	R-type
ADD	0000000	rs2	rs1	000	rd	0110011	R-type
SUB	0100000	rs2	rs1	000	rd	0110011	R-type
SLL	0000000	rs2	rs1	001	rd	0110011	R-type
SLT	0000000	rs2	rs1	010	rd	0110011	R-type
SLTU	0000000	rs2	rs1	011	rd	0110011	R-type
XOR	0000000	rs2	rs1	100	rd	0110011	R-type
SRL	0000000	rs2	rs1	101	rd	0110011	R-type
SRA	0100000	rs2	rs1	101	rd	0110011	R-type
OR	0000000	rs2	rs1	110	rd	0110011	R-type
AND	0000000	rs2	rs1	111	rd	0110011	R-type



RISC-V online resources

▶ Online RISC-V assembler

- ▶ <https://riscvasm.lucasteske.dev/#>
- ▶ Make sure you select only RV32I instructions!

▶ Online RISC-V C compiler

- ▶ <https://godbolt.org/>
- ▶ Select the C language
- ▶ Select a RISC-V compiler (per esempio gcc) for the 32-bit architecture
- ▶ Unfortunately we can't constrain the compiler to generate code for the base ISA only (it might include instructions that you have not implemented!)

RISC-V online compiler

The screenshot displays the Compiler Explorer interface at godbolt.org. The left pane shows the C source code for a function named `iaxpy`. The right pane shows the resulting RISC-V assembly code, with the compiler option `-Os` selected. Annotations highlight the source code, the compiler, the compiled code, and the optimization options.

C language

```
1 /* Type your code here, or load an example. */
2 void iaxpy( int data[ ] ) {
3     for ( int i = 0; i < 1000; i++ ) {
4         data[ i ] = data[ i ] + i;
5     }
6 }
7
```

RISC-V gcc compiler. Could use clang as alternative

Compiler options. -Os optimizes for size (you get fewer instructions)

```
1 iaxpy:
2     li    a5,0
3     li    a3,1000
4 .L2:
5     lw    a4,0(a0)
6     add   a4,a4,a5
7     sw    a4,0(a0)
8     addi  a5,a5,1
9     addi  a0,a0,4
10    bne   a5,a3,.L2
11    ret
```

Source code

Compiled code

RISC-V online assembler

Copy code here

Hit "BUILD"

Get the hex dump to be copied in
instruction memory initialization

Also get the hex to code
correspondence

The screenshot shows the 'RISC-V Online Assembler' web application. The interface includes a text area for assembly code, a 'BUILD' button, and three output sections: Console, Hex Dump, and Objdump. Annotations with colored lines point to specific parts of the interface:

- A blue line points from the 'Copy code here' box to the assembly code text area.
- A red line points from the 'Hit "BUILD"' box to the 'BUILD' button.
- A blue line points from the 'Get the hex dump to be copied in instruction memory initialization' box to the Hex Dump section.
- A green line points from the 'Also get the hex to code correspondence' box to the Objdump section.

Assembly Code:

```
1 iaxpy:
2     li    a5,0
3     li    a3,1000
4 .L2:
5     lw     a4,0(a0)
6     add    a4,a4,a5
7     sw     a4,0(a0)
8     addi   a5,a5,1
9     addi   a0,a0,4
10    bne    a5,a3,.L2
11    ret
12
```

Hex Dump:

```
00000793
3e800693
00052703
00f70733
00e52023
00178793
00450513
fed796e3
00008067
```

Objdump:

```
file.elf:      file format elf64-littleriscv

Disassembly of section .text:

0000000000000000 :
0: 00000793          li    a5,0
4: 3e800693          li    a3,1000

0000000000000008 <.L2>:
8: 00052703          lw     a4,0(a0)
c: 00f70733          add    a4,a4,a5
10: 00e52023          sw     a4,0(a0)
14: 00178793          addi   a5,a5,1
```

Example

```
void iaxpy( int data[ ] ) {
    for ( int i = 0; i < 1000; i++ ) {
        data[ i ] = data[ i ] + 1;
    }
}
```

Program																	
li a5,0	0x00000793	0				0				0				0			
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
li a3,1000	0x3e800693	1000				0				0				0			
loop:		0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0
lw a4,0(a0)	0x00052703	0				10				0				0			
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
add a4,a4,a5	0x00f70733	0				15				14				14			
		0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0
sw a4,0(a0)	0x00e52023	0				14				10				0			
		0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0
addi a5,a5,1	0x00178793	1				15				15				15			
		0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
addi a0,a0,4	0x00450513	4				10				10				10			
		0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
bne a5,a3,loop	0xfed796e3	negative				13				15				-20			
		1	1	1	1	1	1	1	0	0	1	0	1	0	1	1	1

Notes:

- Registers **a0**, **a1**, **a2**, etc. correspond to **x10**, **x11**, **x12**, etc. (it is a standard naming convention)
- **a0** is the pointer to the array in memory (initially zero?)
- **a5** contains the current loop index (variable *i* in the C code)
- **a3** is the end of loop condition (1000)
- **a4** is a temporary that holds the value loaded from memory (*data[i]*), then it gets added to **a5** (the index) and is stored back to memory (*data[i]*)
- Branch checks if the index (**a5**) has not reached the end (**a3**)



Example architecture

Instruction Fetch

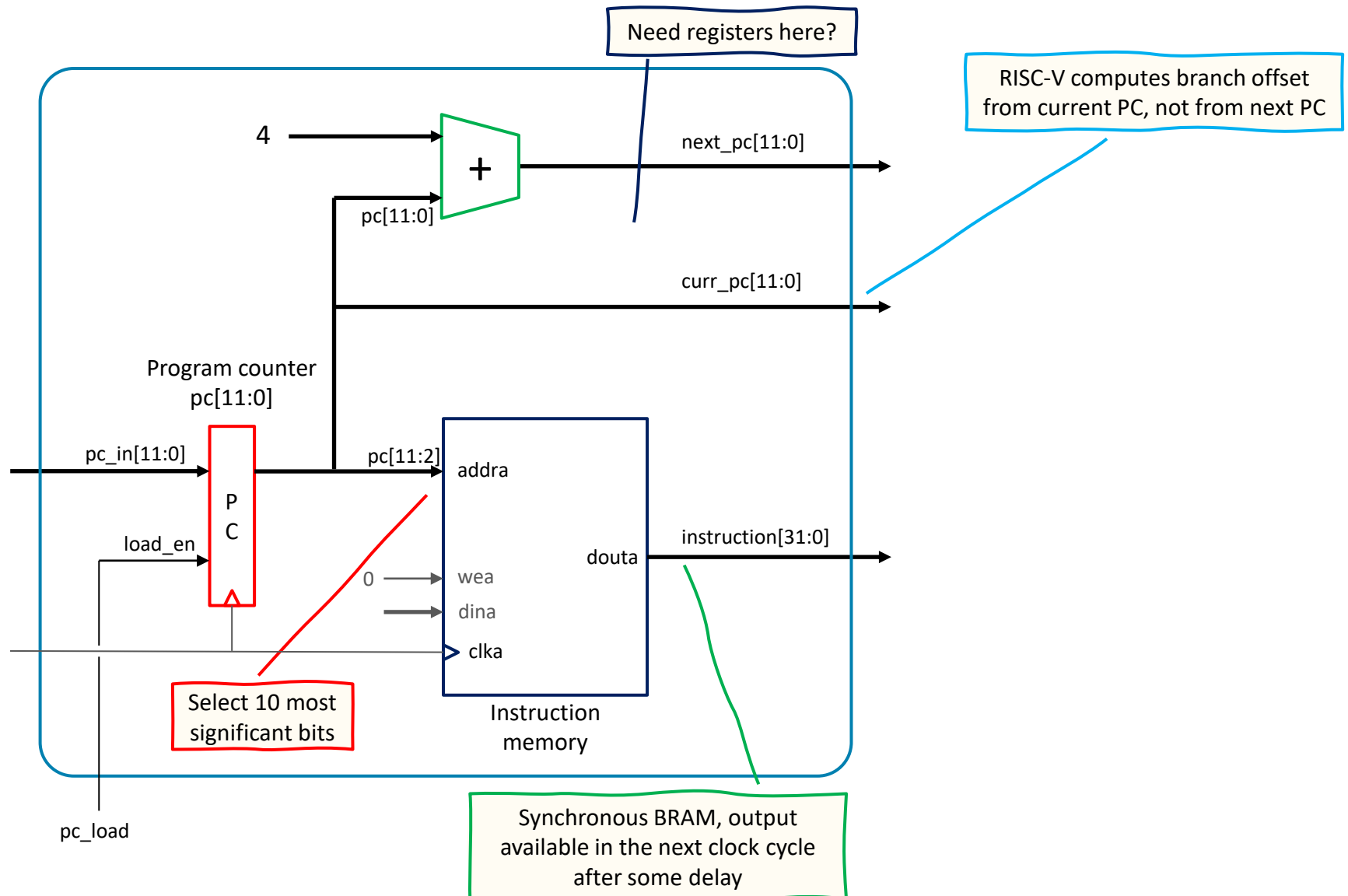
▶ **Instruction Memory**

- ▶ Defined as 1024 words of 32 bits each, total of 4 kB
 - ▶ Address is 10 bits
- ▶ Using a BRAM, it has an output register
 - ▶ Instruction code available only in the clock cycle that follows the new value of the program counter
- ▶ Could be potentially be placed outside, to be more general

▶ **Program Counter (PC)**

- ▶ Must address 4 kB, hence it is 12 bits if addressing bytes
 - ▶ The two least significant bits will always be 0, removed when connecting to instruction memory
 - ▶ PC arithmetic in the ALU done at byte level
- ▶ The stage computes the address of the next instruction, by adding 4 for the PC
- ▶ Load enable (pc_load) activated by control state machine

Instruction fetch



Instruction Decode

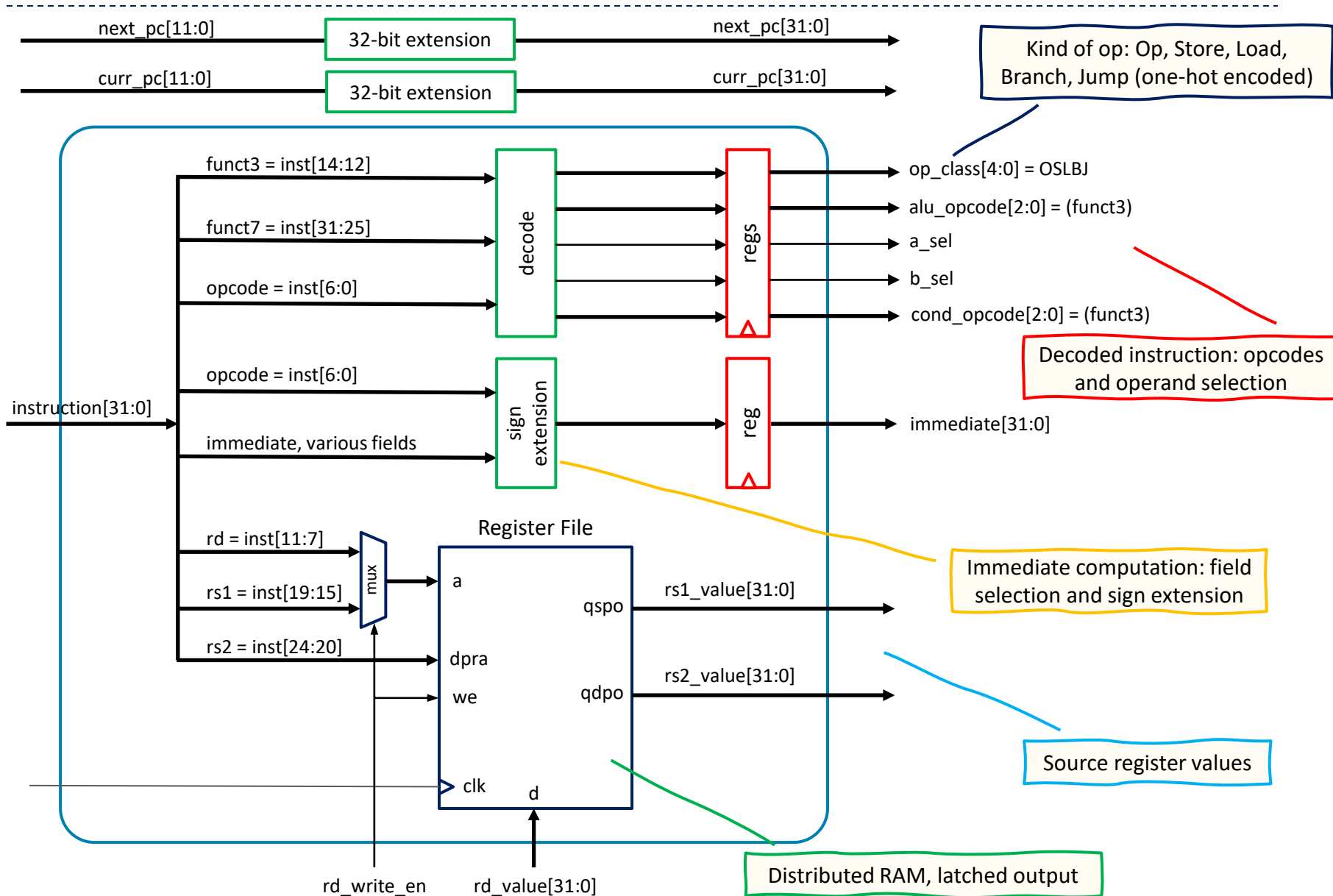
▶ Register file

- ▶ Defined as 32 words of 32 bits each, two ports, distributed memory
 - ▶ First port used to read and write, at different times
 - ▶ Address (source or destination) chosen depending on instruction execution phase
 - ▶ write_en generated by control state machine
 - ▶ Latched output, since we would need to add it anyway

▶ Decoding logic and sign extension

- ▶ Checks instruction class and defines operand selection signals
- ▶ Defines operation signal for ALU and comparator
- ▶ Selects immediate fields and reconstructs the sign extended immediate according to the instruction class
- ▶ Outputs are latched

Instruction decode



Instruction Execute

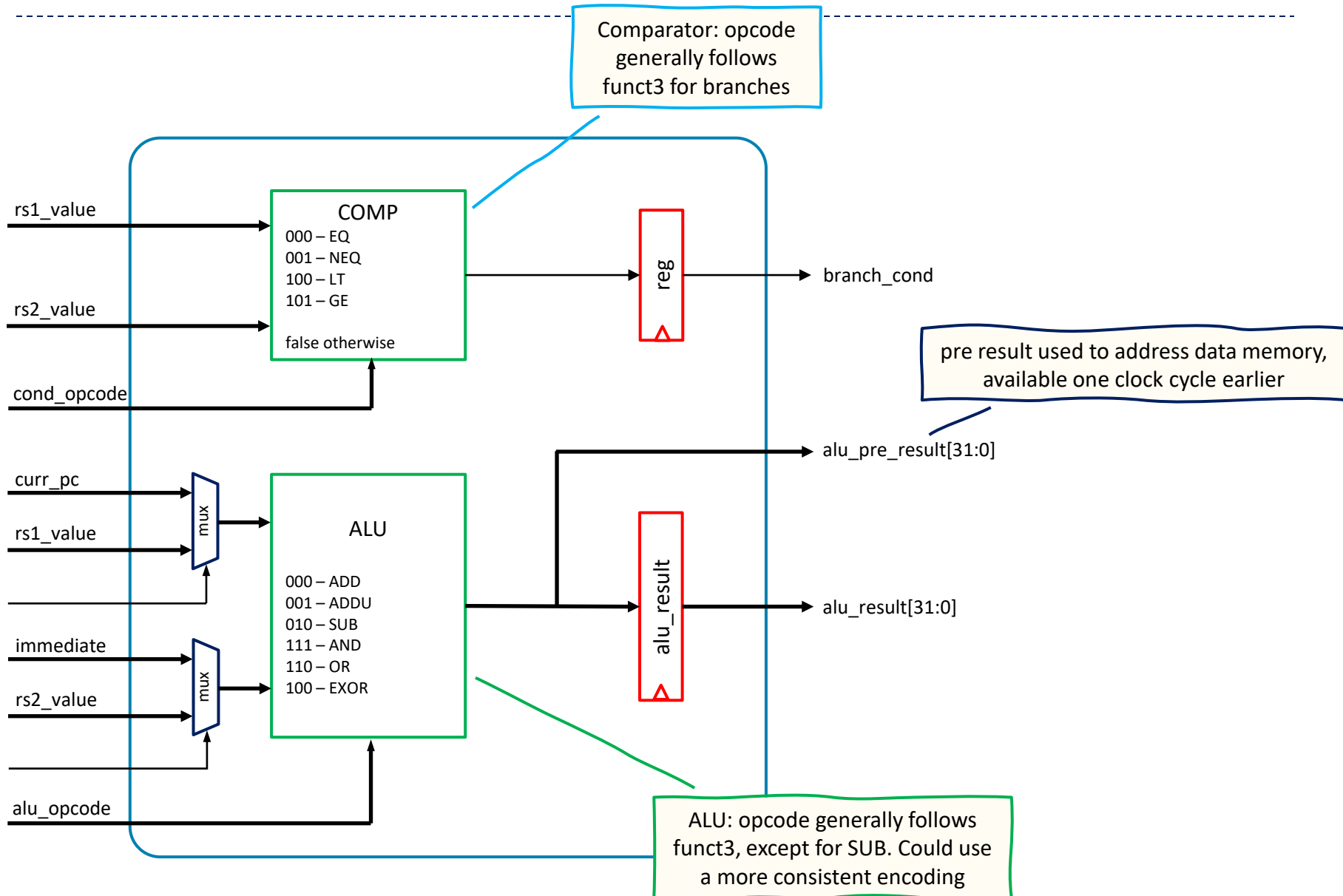
▶ **ALU**

- ▶ Implements a subset of arithmetic and logic operations
- ▶ Muxes at inputs select the operands
- ▶ Opcode generally follows the RISC-V standard convention
- ▶ Both latched and unlatched outputs available

▶ **Comparator**

- ▶ Used for branch operations, compares register values according to opcode
- ▶ Output determines if branch is to be taken or not

Instruction execute



Memory access and write back

▶ **Data Memory**

- ▶ Implemented as 4096 words of 32 bits
 - ▶ Address is 12 bits, take only the required bits from effective address (coming from the ALU)
 - ▶ write_en generated by control state machine
- ▶ **Memory introduces one clock cycle delay**
 - ▶ Take the ALU output from previous clock cycle, to get memory output during the memory phase
 - ▶ Write value (rs2) already available since end of decode phase

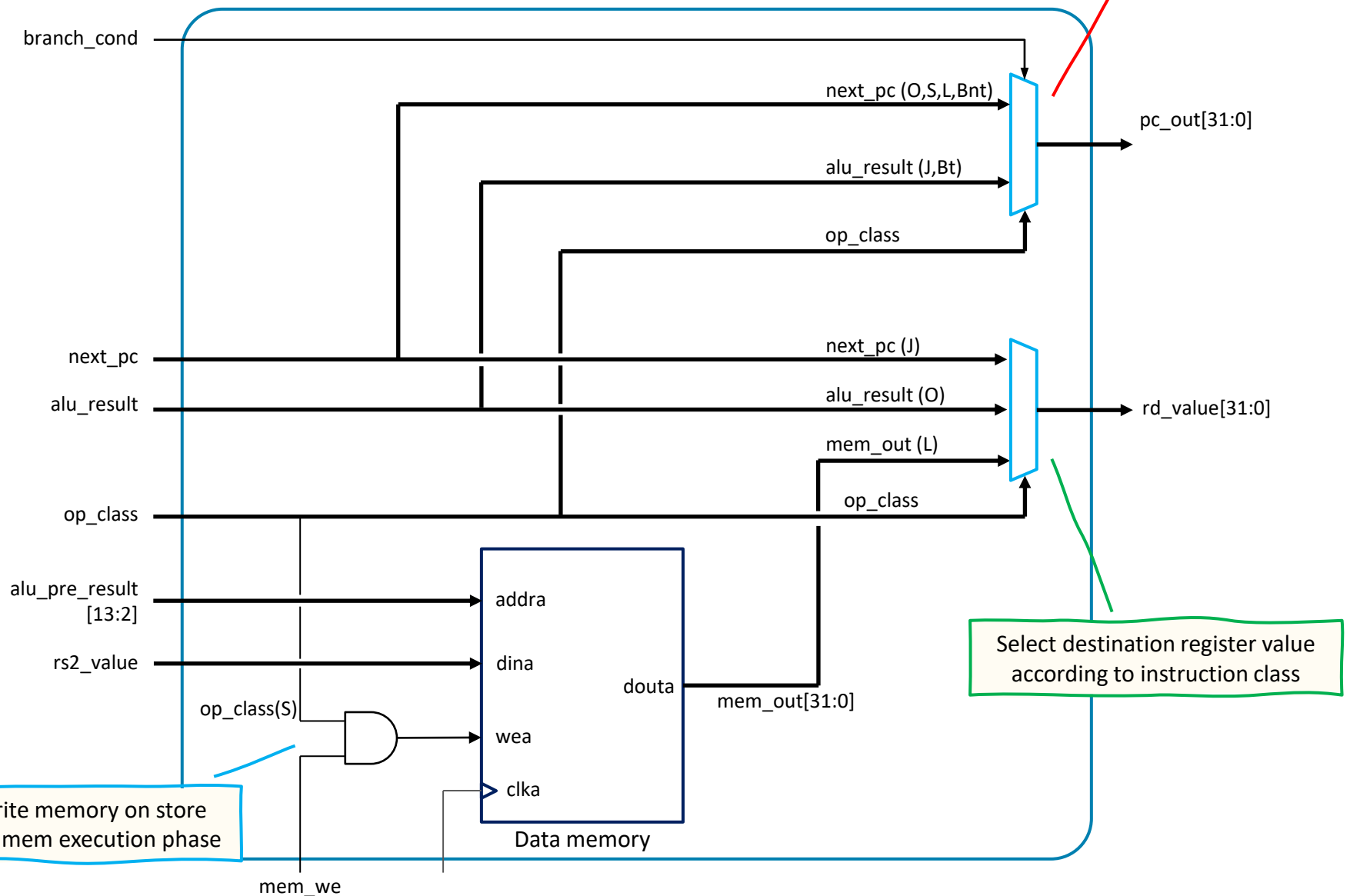
▶ **Destination register value selection**

- ▶ Could be the next_pc (for JAL), the ALU result (for ALU operations), or the memory output (for loads)

▶ **Value of program counter**

- ▶ Next instruction at next_pc (for ALU operations, loads, stores, and not taken branches)
- ▶ Next instruction at alu_result (for Jumps and taken branches)

Memory and write back



Control state machine

- ▶ **Schedules certain write operations**
 - ▶ Updating the program counter
 - ▶ Writing into the register file
 - ▶ Writing into memory
- ▶ **Uses op_class to decide**
 - ▶ Plus keeps track of the phase of execution

Phases

