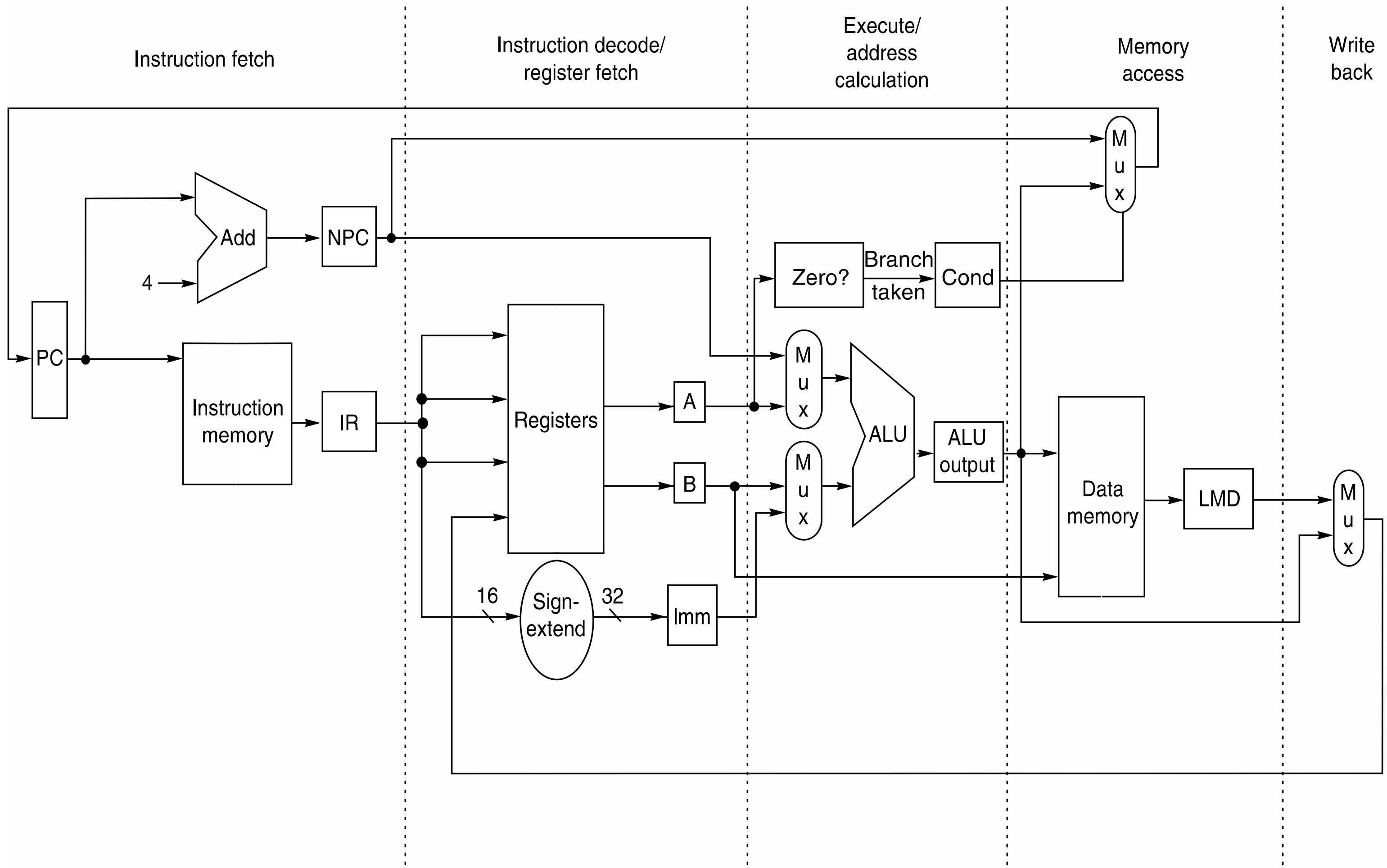
RISC-V CPU design guidelines

A non-pipelined RISC architecture



RISC-V encoding

- **6 encoding types**
  - R-type: for two source register operations
  - I-type: one source register and one immediate value
  - S-type: for store operations
  - B-type: for branch operations
  - U-type: for specific LUI instruction
  - J-type: for jump operations

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<tr><td>funct7</td><td>rs2</td><td>rs1</td><td>funct3</td><td>rd</td><td>opcode</td><td>R-type</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>funct3</td><td>rd</td><td>opcode</td><td>I-type</td></tr>
<tr><td>imm[11:5]</td><td>rs2</td><td>rs1</td><td>funct3</td><td>imm[4:0]</td><td>opcode</td><td>S-type</td></tr>
<tr><td>imm[12|10:5]</td><td>rs2</td><td>rs1</td><td>funct3</td><td>imm[4:1|11]</td><td>opcode</td><td>B-type</td></tr>
<tr><td colspan="4">imm[31:12]</td><td>rd</td><td>opcode</td><td>U-type</td></tr>
<tr><td colspan="4">imm[20|10:1|11|19:12]</td><td>rd</td><td>opcode</td><td>J-type</td></tr>
</table>

Opcodes

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

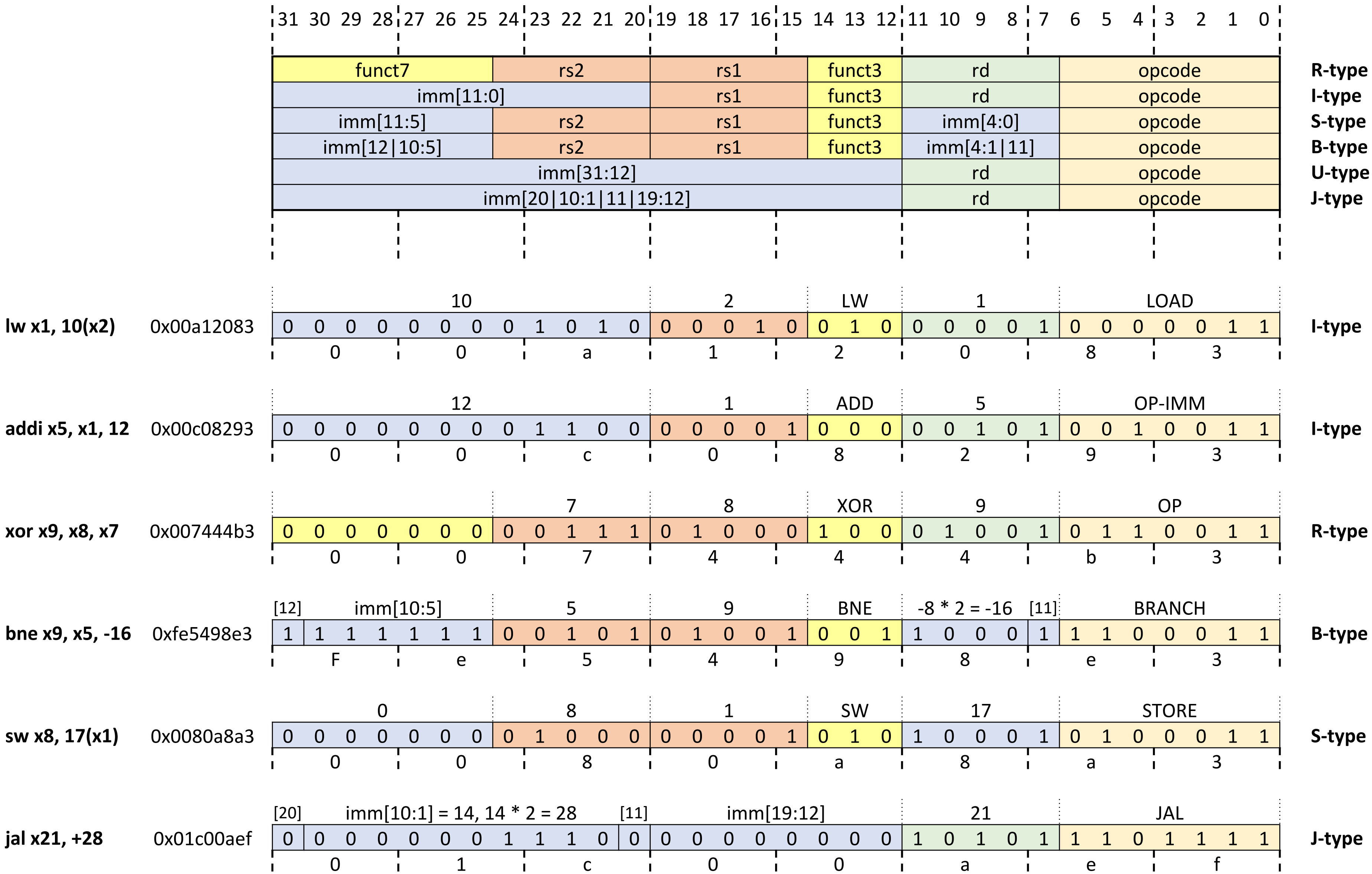
inst[4:2] / **inst[6:5]**	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM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

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 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|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	**B-type**
imm[31:12]				rd	opcode	**U-type**
imm[20|10:1|11|19:12]				rd	opcode	**J-type**

JAL	imm[20|10:1|11|19:12]				rd	1101111	**J-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**
LW	imm[11:0]		rs1	010	rd	0000011	**I-type**
SW	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	**S-type**
ADDI	imm[11:0]		rs1	000	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**
ADD	0000000	rs2	rs1	000	rd	0110011	**R-type**
SUB	0100000	rs2	rs1	000	rd	0110011	**R-type**
XOR	0000000	rs2	rs1	100	rd	0110011	**R-type**
OR	0000000	rs2	rs1	110	rd	0110011	**R-type**
AND	0000000	rs2	rs1	111	rd	0110011	**R-type**



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 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|10:5] | rs2 | rs1 | funct3 | imm[4:1|11] | opcode
B-type
imm[31:12] | rd | opcode
U-type
imm[20|10:1|11|19:12] | rd | opcode
J-type
lw x1, 10(x2)
0x00a12083
10 | 2 | LW | 1 | LOAD
0 0 0 0 0 0 0 0 1 0 1 0 | 0 0 0 1 0 | 0 1 0 | 0 0 0 0 1 | 0 0 0 0 0 1 1
0 0 a 1 2 0 8 3
I-type
addi x5, x1, 12
0x00c08293
12 | 1 | ADD | 5 | OP-IMM
0 0 0 0 0 0 0 0 1 1 0 0 | 0 0 0 0 1 | 0 0 0 | 0 0 1 0 1 | 0 0 1 0 0 1 1
0 0 c 0 8 2 9 3
I-type
xor x9, x8, x7
0x007444b3
7 | 8 | XOR | 9 | OP
0 0 0 0 0 0 0 | 0 0 1 1 1 | 0 1 0 0 0 | 1 0 0 | 0 1 0 0 1 | 0 1 1 0 0 1 1
0 0 7 4 4 4 b 3
R-type
bne x9, x5, -16
0xfe5498e3
[12] | imm[10:5] | 5 | 9 | BNE | -8 * 2 = -16 | [11] | BRANCH
1 | 1 1 1 1 1 1 | 0 0 1 0 1 | 0 1 0 0 1 | 0 0 1 | 1 0 0 0 | 1 | 1 1 0 0 0 1 1
F e 5 4 9 8 e 3
B-type
sw x8, 17(x1)
0x0080a8a3
0 | 8 | 1 | SW | 17 | STORE
0 0 0 0 0 0 0 | 0 1 0 0 0 | 0 0 0 0 1 | 0 1 0 | 1 0 0 0 1 | 0 1 0 0 0 1 1
0 0 8 0 a 8 a 3
S-type
jal x21, +28
0x01c00aef
[20] | imm[10:1] = 14, 14 * 2 = 28 | [11] | imm[19:12] | 21 | JAL
0 | 0 0 0 0 0 0 0 1 1 1 0 | 0 | 0 0 0 0 0 0 0 0 | 1 0 1 0 1 | 1 1 0 1 1 1 1
0 1 c 0 0 a e f
J-type

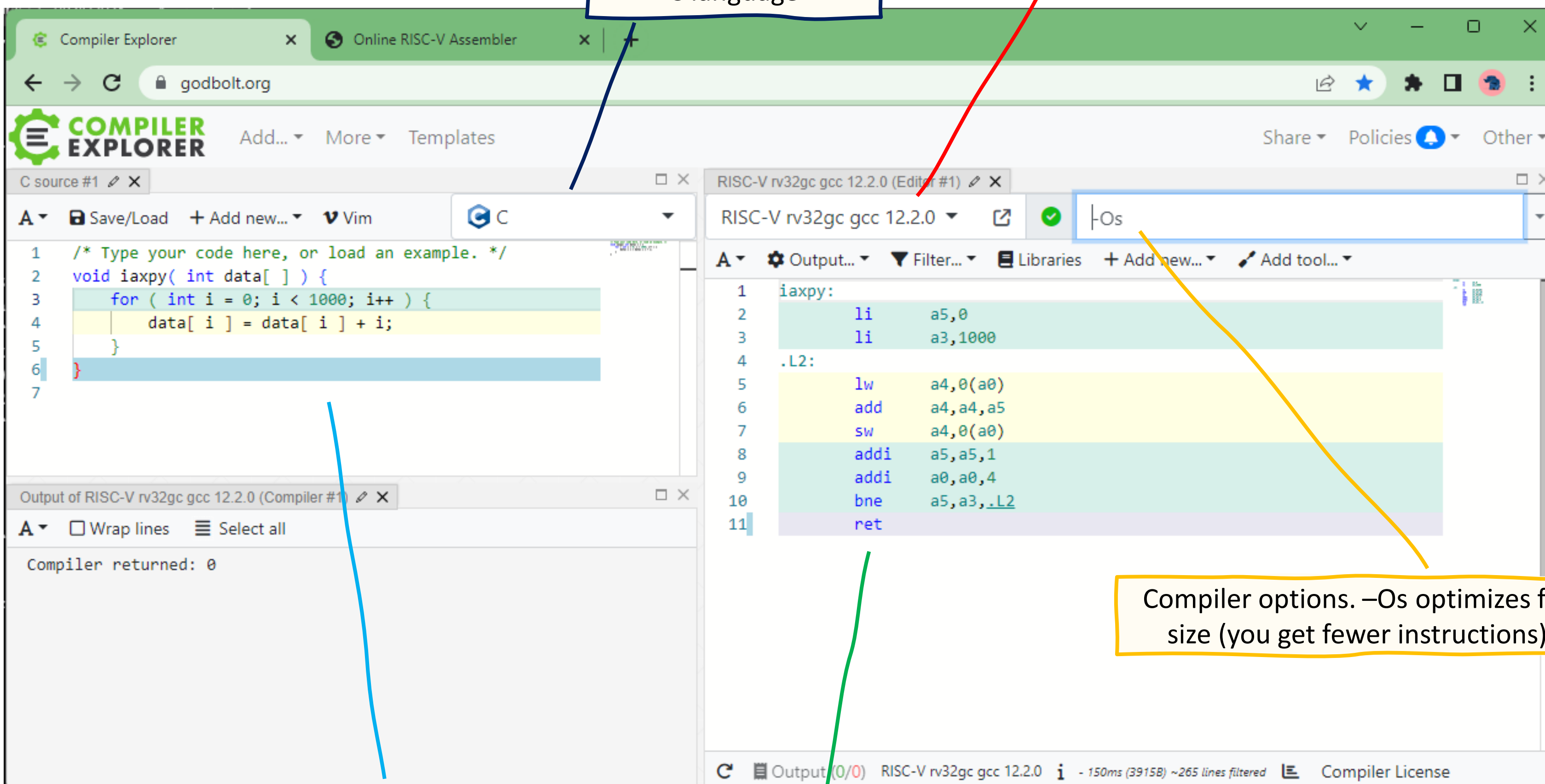
RISC-V online resources

- **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

RISC-V gcc compiler. Could use clang as alternative

C language



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

Source code

Compiled code

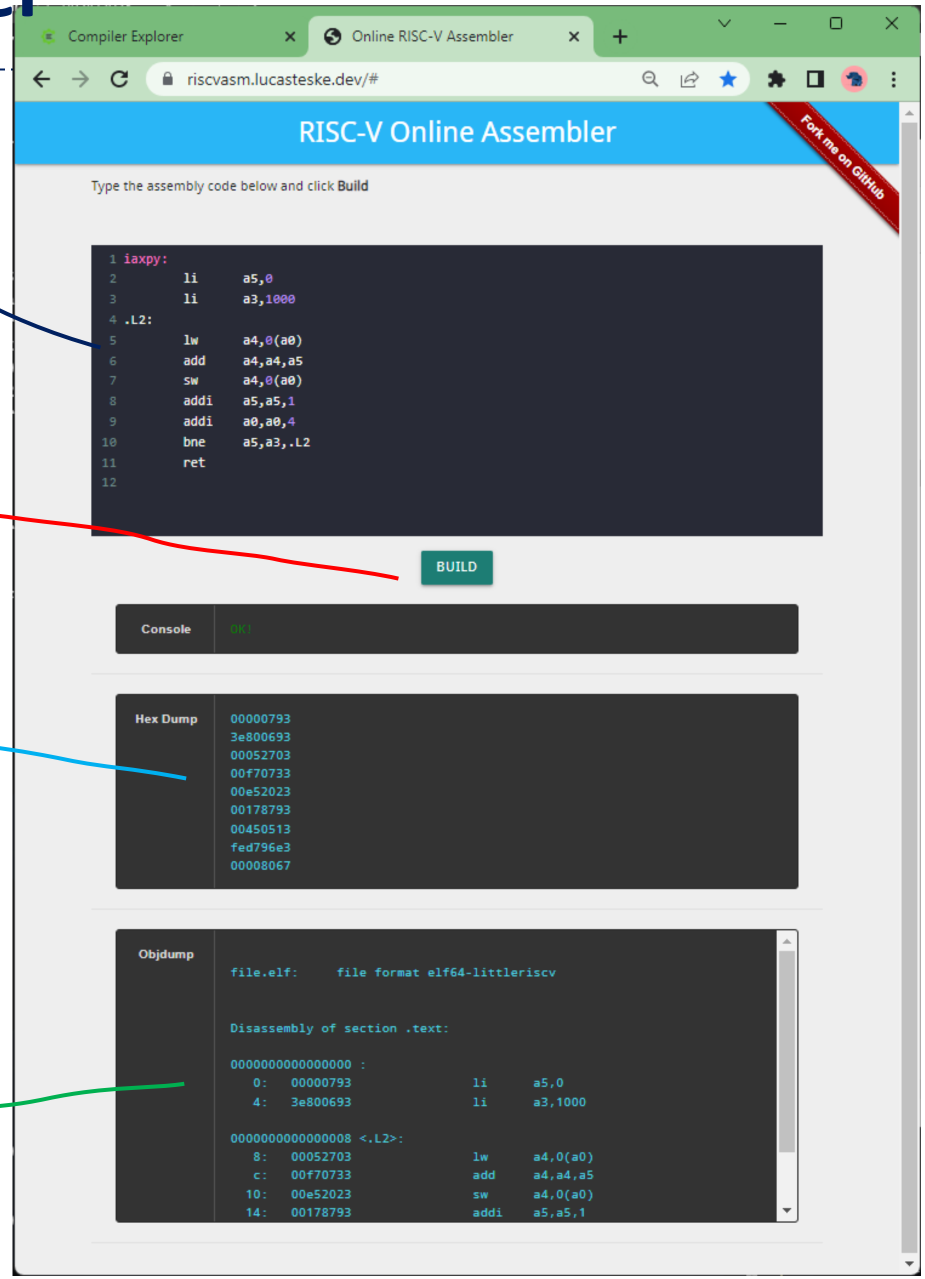
RISC-V online assembler

Copy code here

Hit "BUILD"

Get the hex dump to be copied in instruction memory initialization

Also get the hex to code correspondence



Example

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

Program

		0		0	ADDI	15	OP-IMM
li a5,0	0x00000793	0 0 0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0	0 1 1 1 1	0 0 1 0 0 1 1
		1000		0	ADDI	13	OP-IMM
li a3,1000	0x3e800693	0 0 1 1 1 1 1	0 1 0 0 0	0 0 0 0 0	0 0 0	0 1 1 0 1	0 0 1 0 0 1 1
loop:		0		10	LW	14	LOAD
lw a4,0(a0)	0x00052703	0 0 0 0 0 0 0	0 0 0 0 0	0 1 0 1 0	0 1 0	0 1 1 1 0	0 0 0 0 0 1 1
		0	15	14	ADD	14	OP
add a4,a4,a5	0x00f70733	0 0 0 0 0 0 0	0 1 1 1 1	0 1 1 1 0	0 0 0	0 1 1 1 0	0 1 1 0 0 1 1
		0	14	10	SW	0	STORE
sw a4,0(a0)	0x00e52023	0 0 0 0 0 0 0	0 1 1 1 0	0 1 0 1 0	0 1 0	0 0 0 0 0	0 1 0 0 0 1 1
		1		15	ADDI	15	OP-IMM
addi a5,a5,1	0x00178793	0 0 0 0 0 0 0	0 0 0 0 1	0 1 1 1 1	0 0 0	0 1 1 1 1	0 0 1 0 0 1 1
		4		10	ADDI	10	OP-IMM
addi a0,a0,4	0x00450513	0 0 0 0 0 0 0	0 0 1 0 0	0 1 0 1 0	0 0 0	0 1 0 1 0	0 0 1 0 0 1 1
		negative	13	15	BNE	-20	BRANCH
bne a5,a3,loop	0xfed796e3	1 1 1 1 1 1 1	0 1 1 0 1	0 1 1 1 1	0 0 1	0 1 1 0 1	1 1 0 0 0 1 1

Notes:

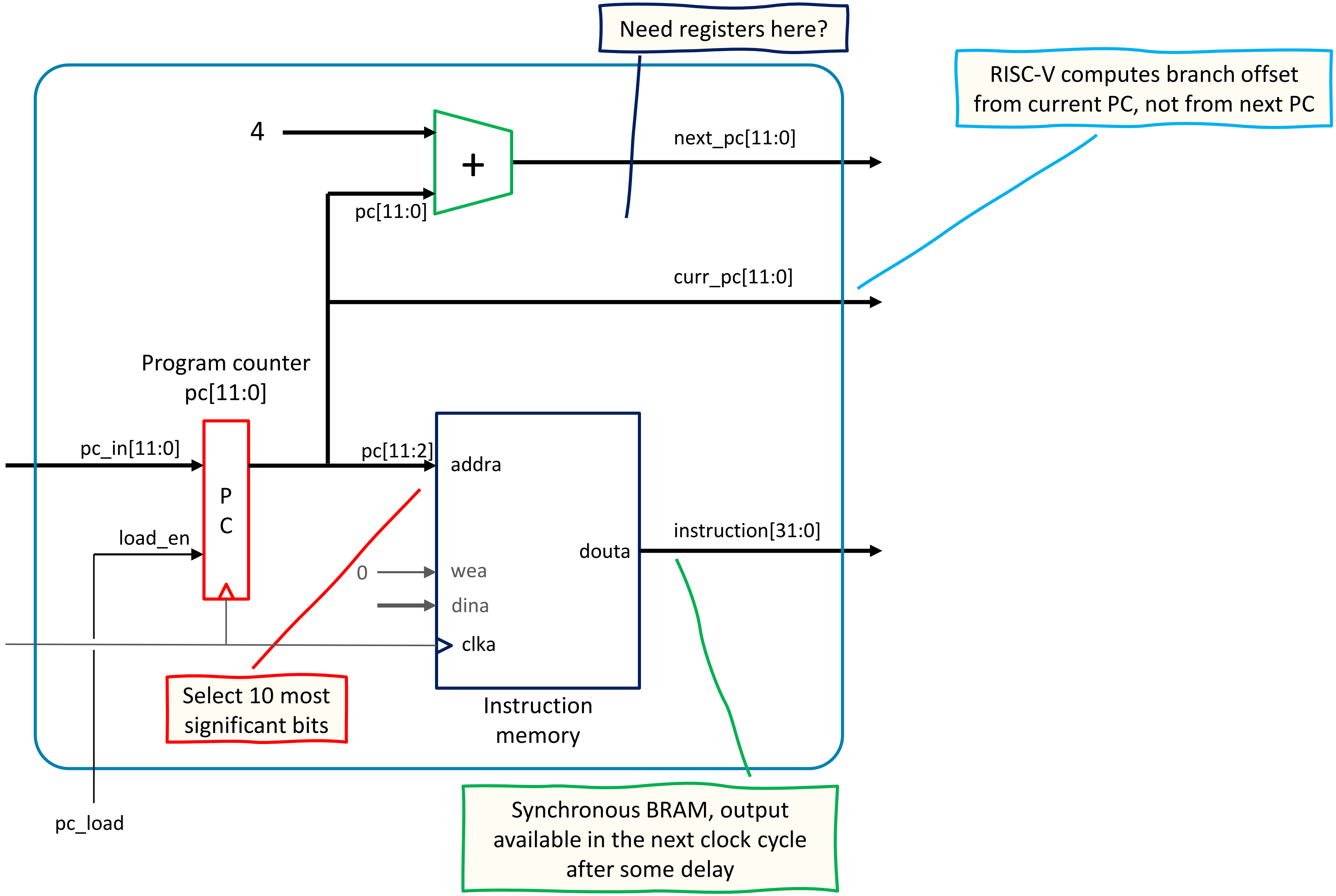
- Registers **a0**, **a1**, **a2**, etc. correspond to **x10**, **x11**, **x12**, etc. (it is a standard naming convention)
- **a0** is the pointer to the array in memory (initially zero?)
- **a5** contains the current loop index (variable i in the C code)
- **a3** is the end of loop condition (1000)
- **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

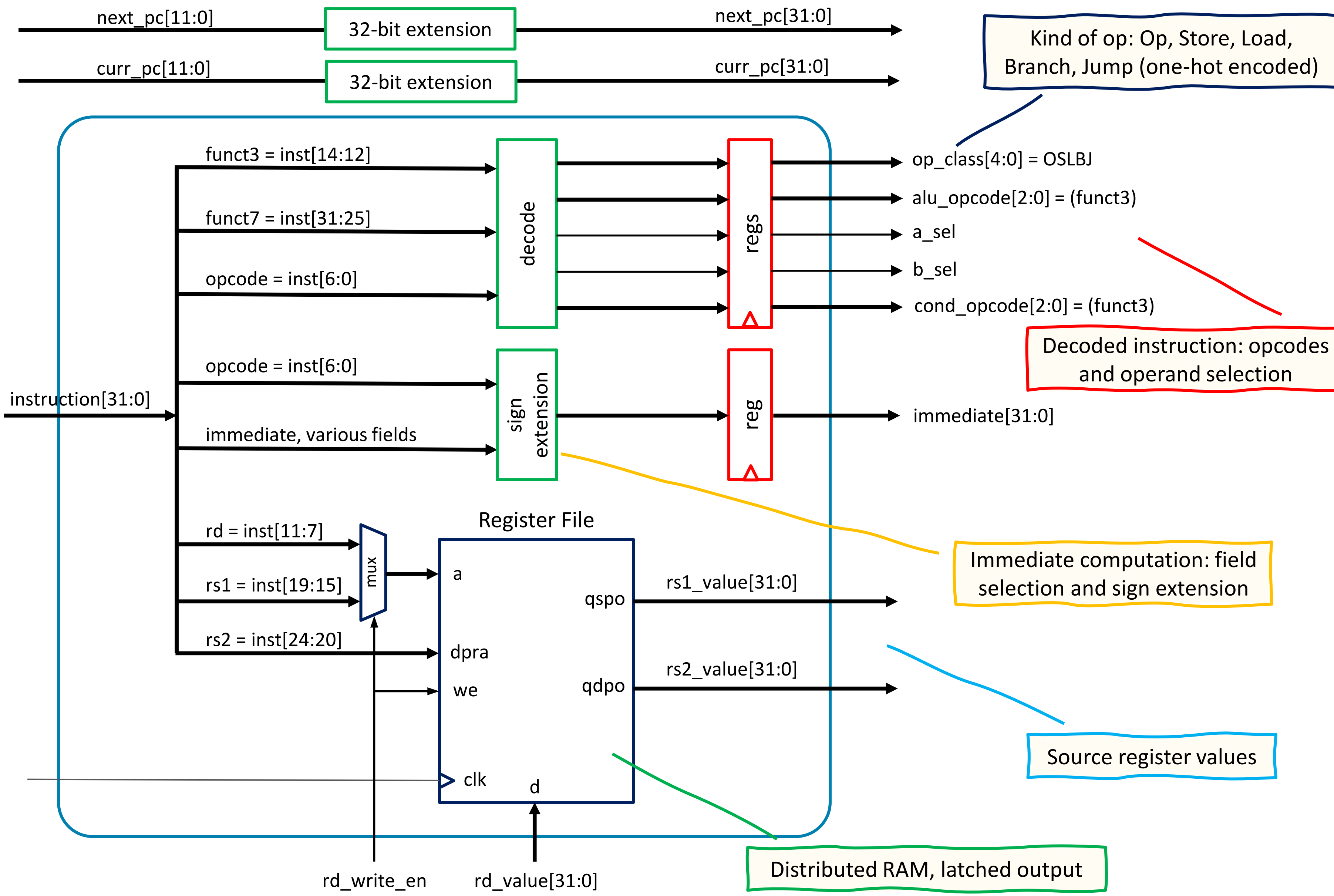
Instruction fetch



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

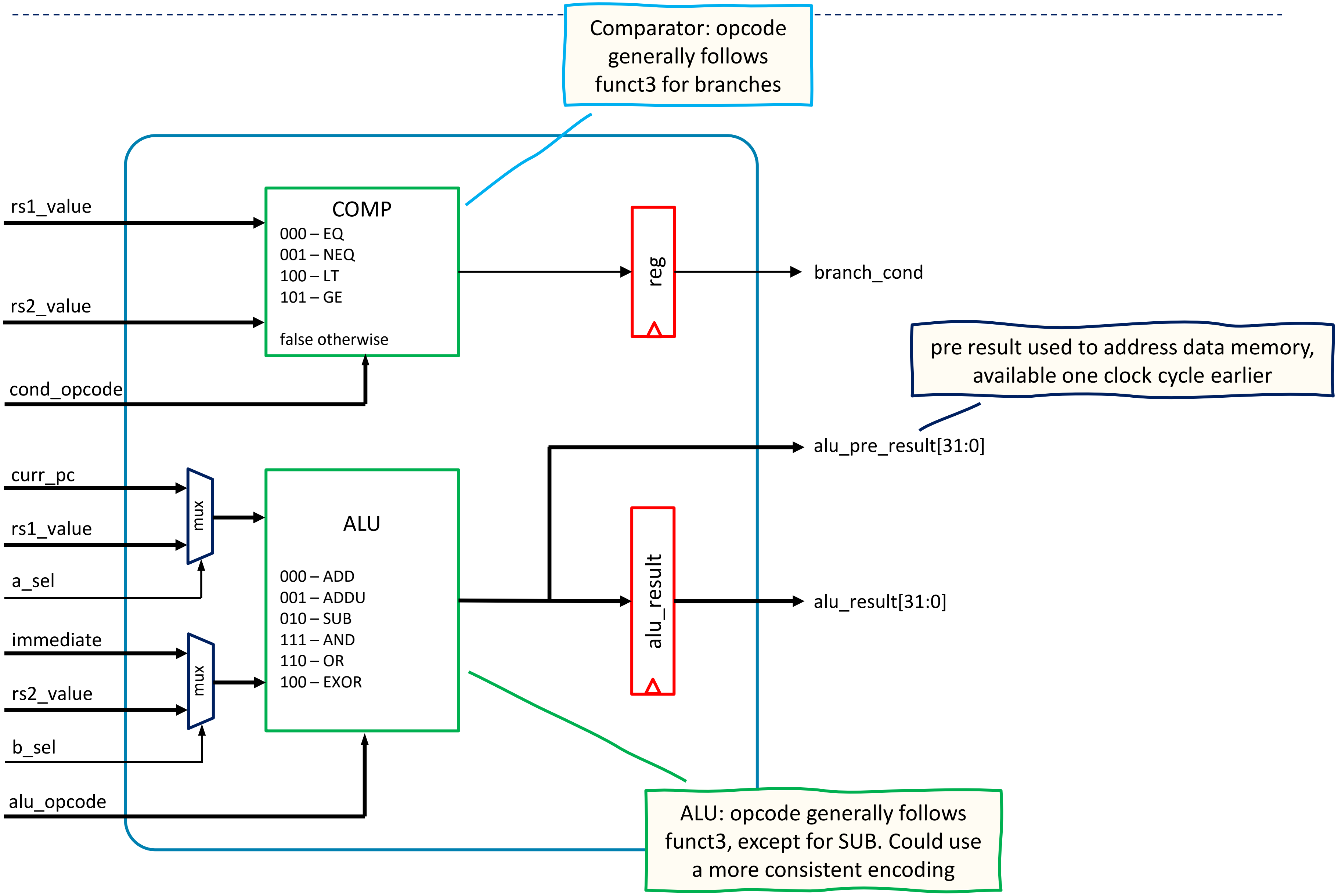
Instruction decode



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

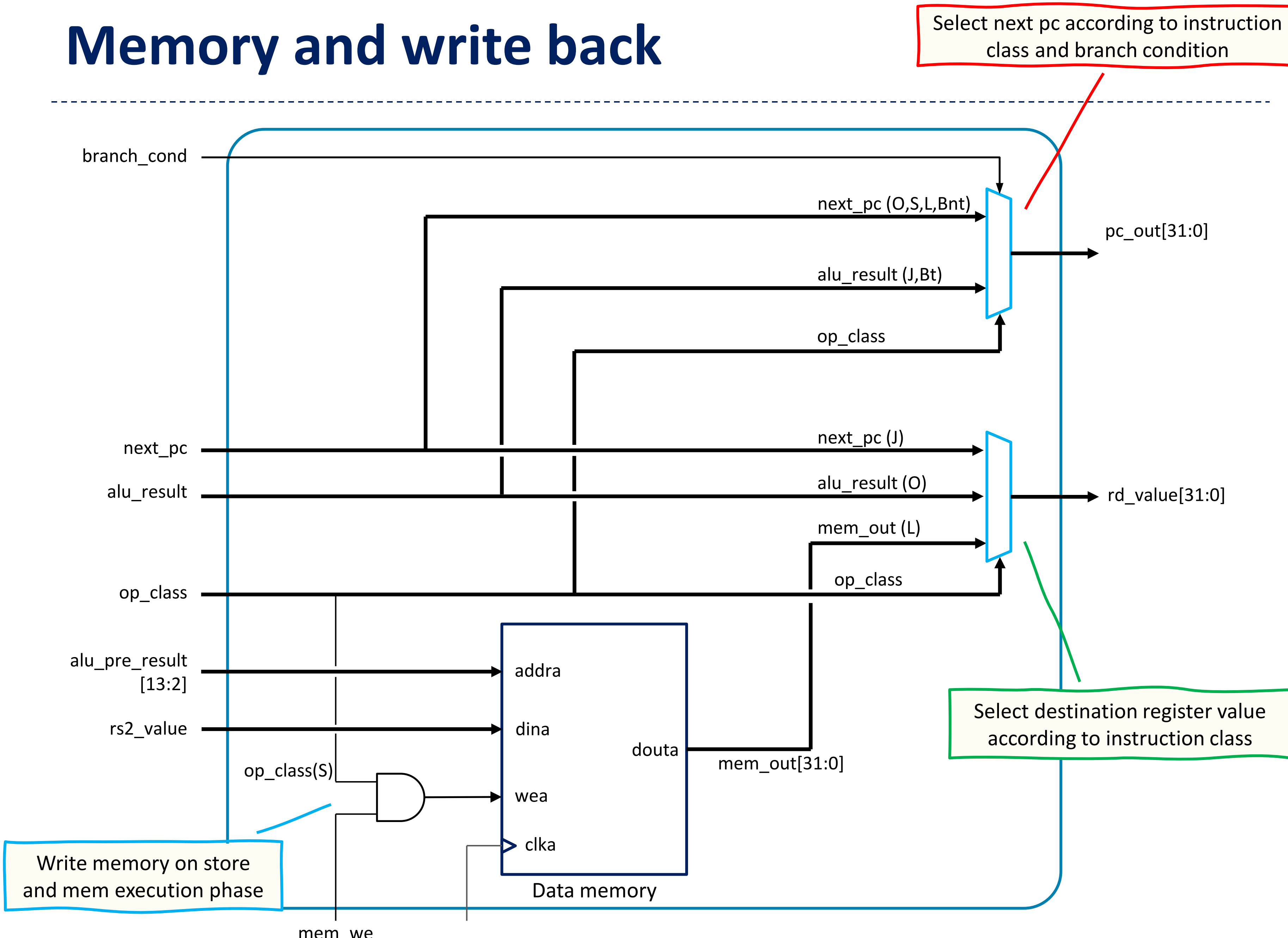
Instruction execute



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)

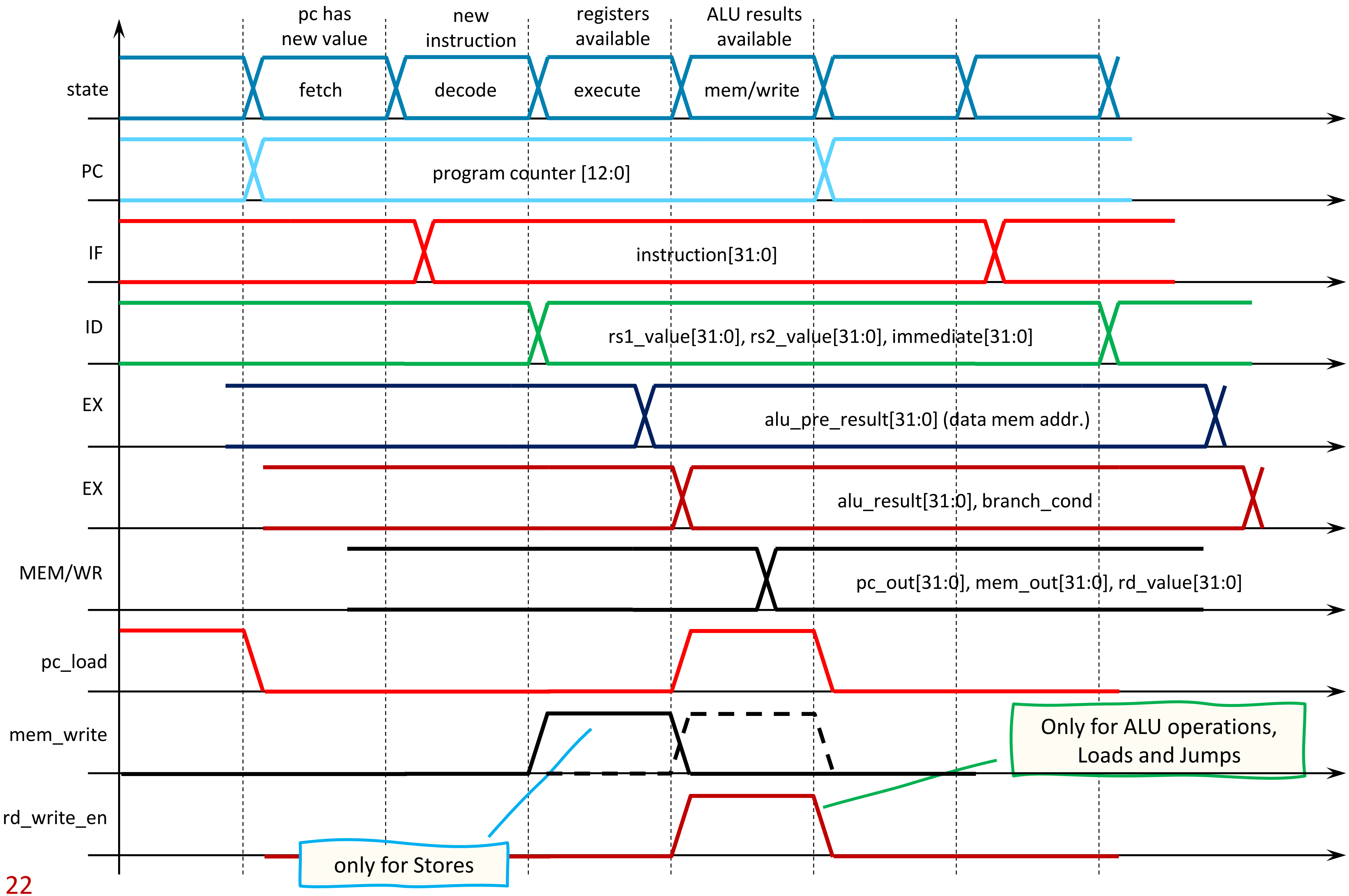
Memory and write back



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



pc has new value
new instruction
registers available
ALU results available
state
fetch
decode
execute
mem/write
PC
program counter [12:0]
IF
instruction[31:0]
ID
rs1_value[31:0], rs2_value[31:0], immediate[31:0]
EX
alu_pre_result[31:0] (data mem addr.)
EX
alu_result[31:0], branch_cond
MEM/WR
pc_out[31:0], mem_out[31:0], rd_value[31:0]
pc_load
mem_write
Only for ALU operations, Loads and Jumps
rd_write_en
only for Stores