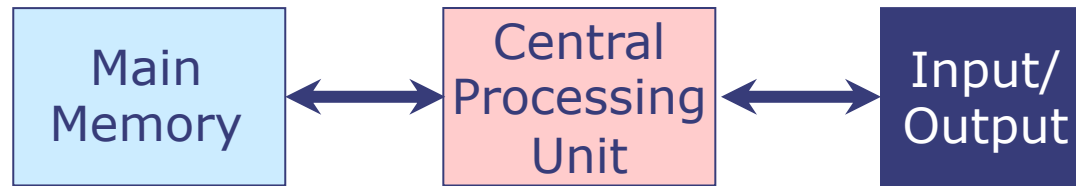


Implementing RISC-V Processor in Hardware



The von Neumann Model

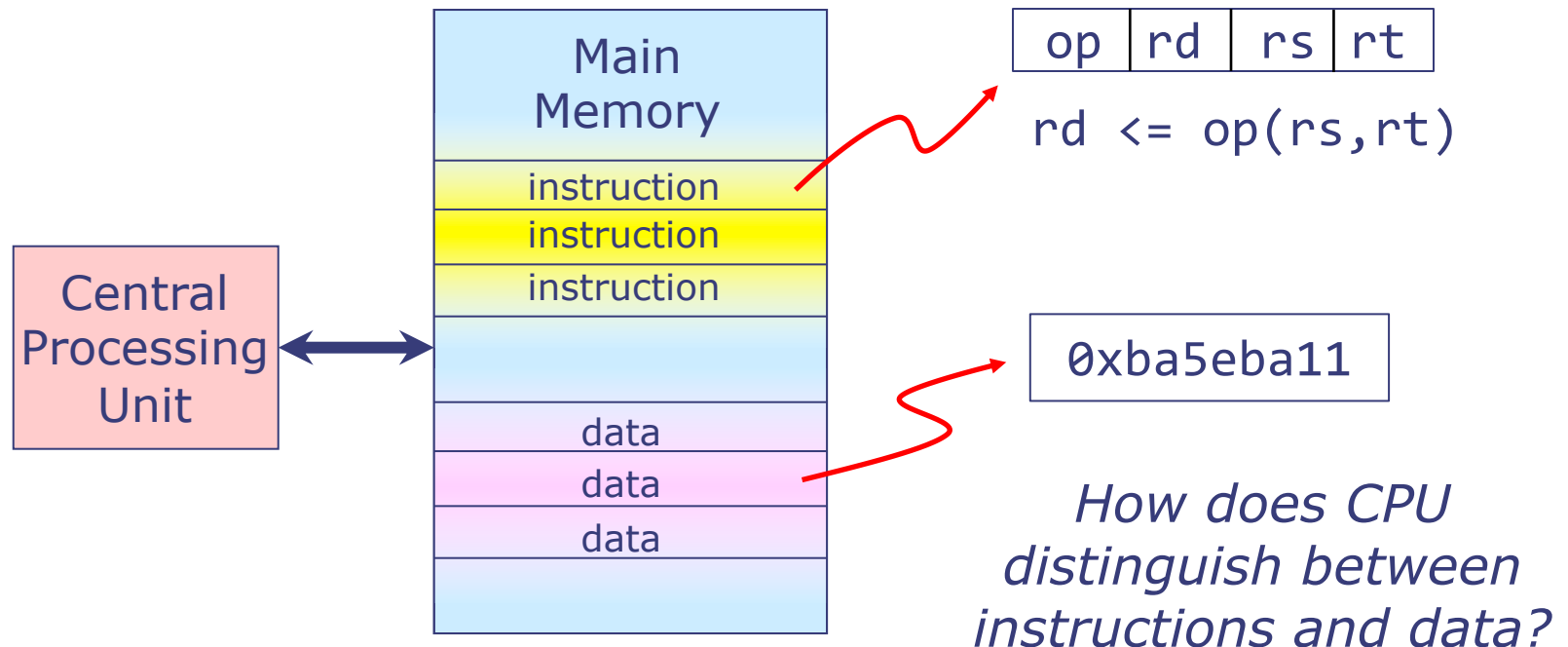
- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:



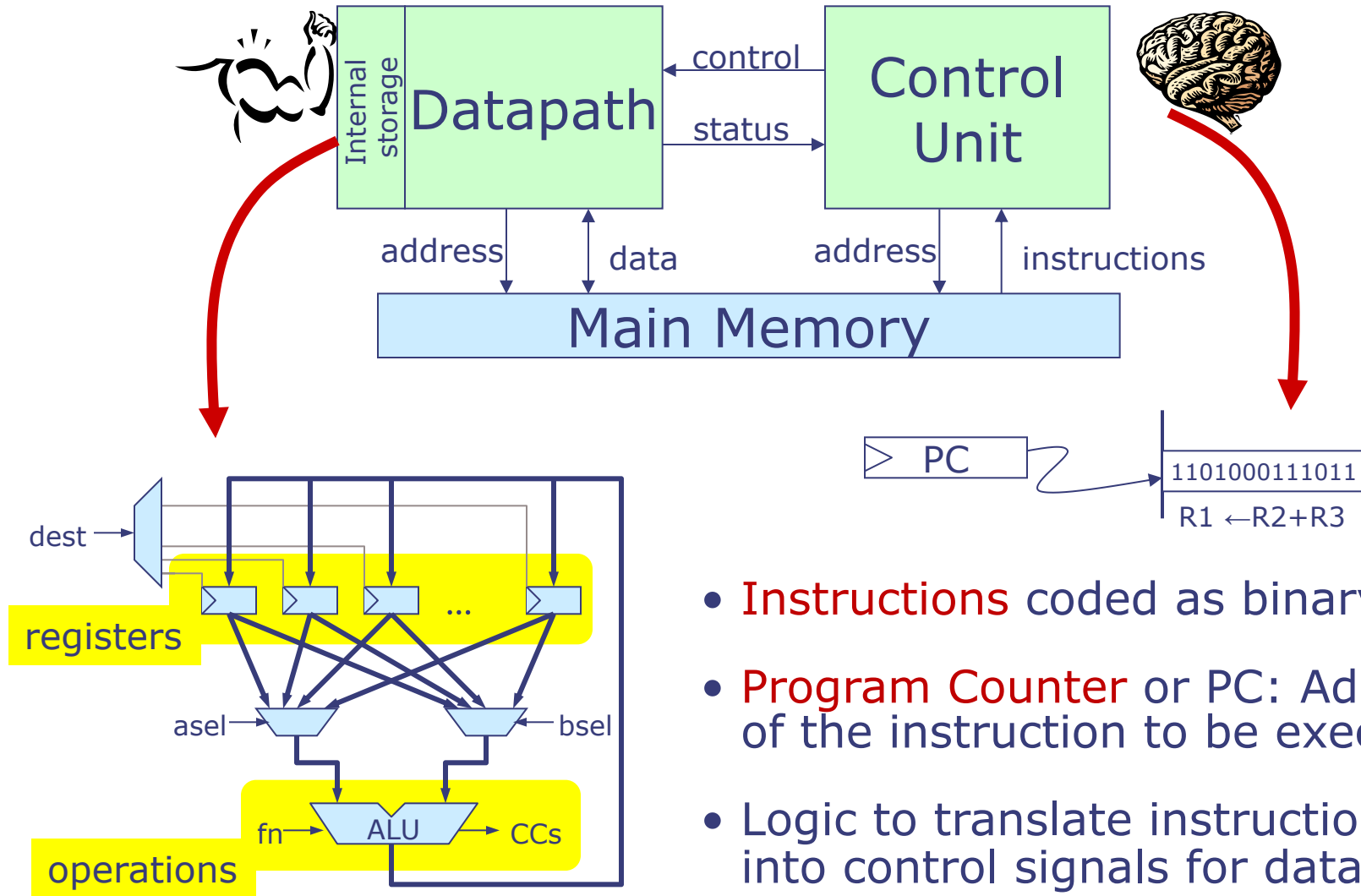
- **Main memory** holds programs and their data
- **Central processing unit** accesses and processes memory values
- **Input/output devices** to communicate with the outside world

Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program



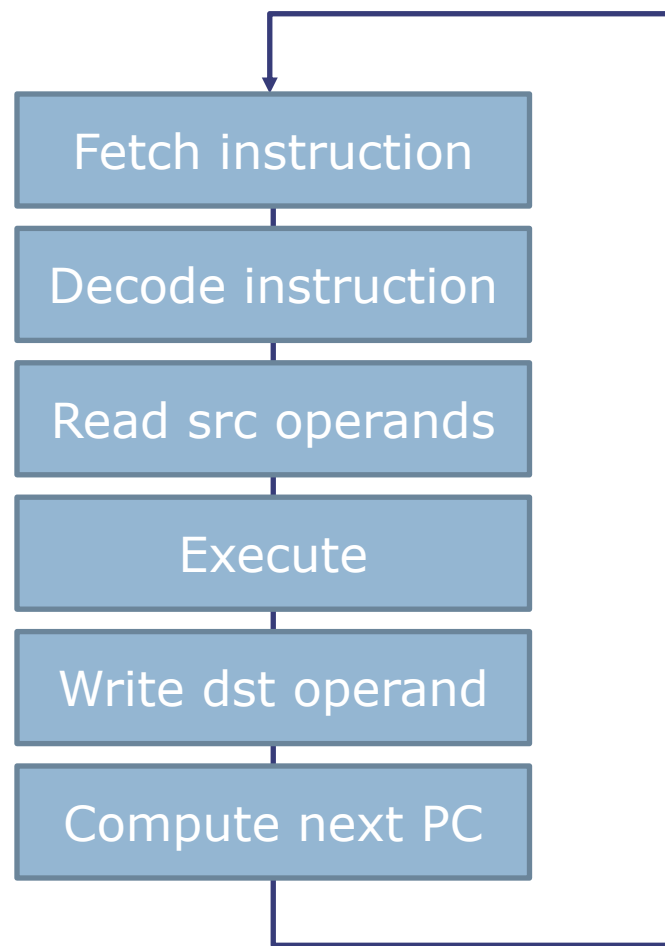
Anatomy of a von Neumann Computer



- **Instructions** coded as binary data
- **Program Counter** or PC: Address of the instruction to be executed
- Logic to translate instructions into control signals for datapath

Instructions

- Instructions are the fundamental unit of work
- Each instruction specifies:
 - An operation or **opcode** to be performed
 - Source **operands** and **destination** for the result
- In a von Neumann machine, instructions are executed sequentially
 - CPU logically implements this loop:
 - By default, the next PC is current PC + size of current instruction unless the instruction says otherwise



Approach: Incremental Featurism

We'll implement datapaths for each instruction class individually, and merge them (using MUXes, etc)

Steps:

1. ALU instructions
2. Load & store instructions
3. Branch & jump instructions

Component Repertoire:



Registers

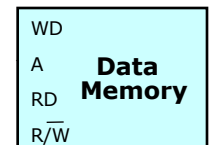
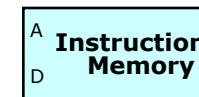
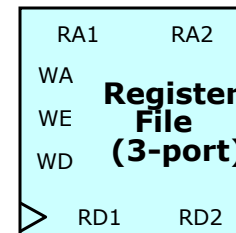


Muxes



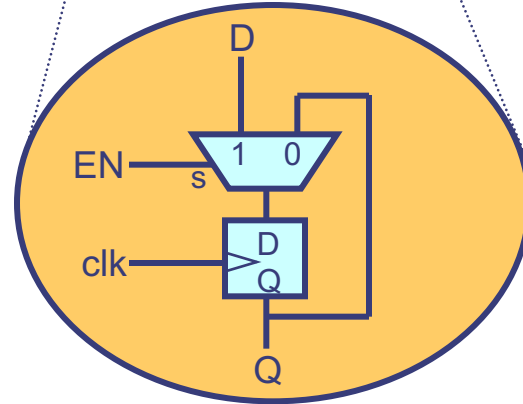
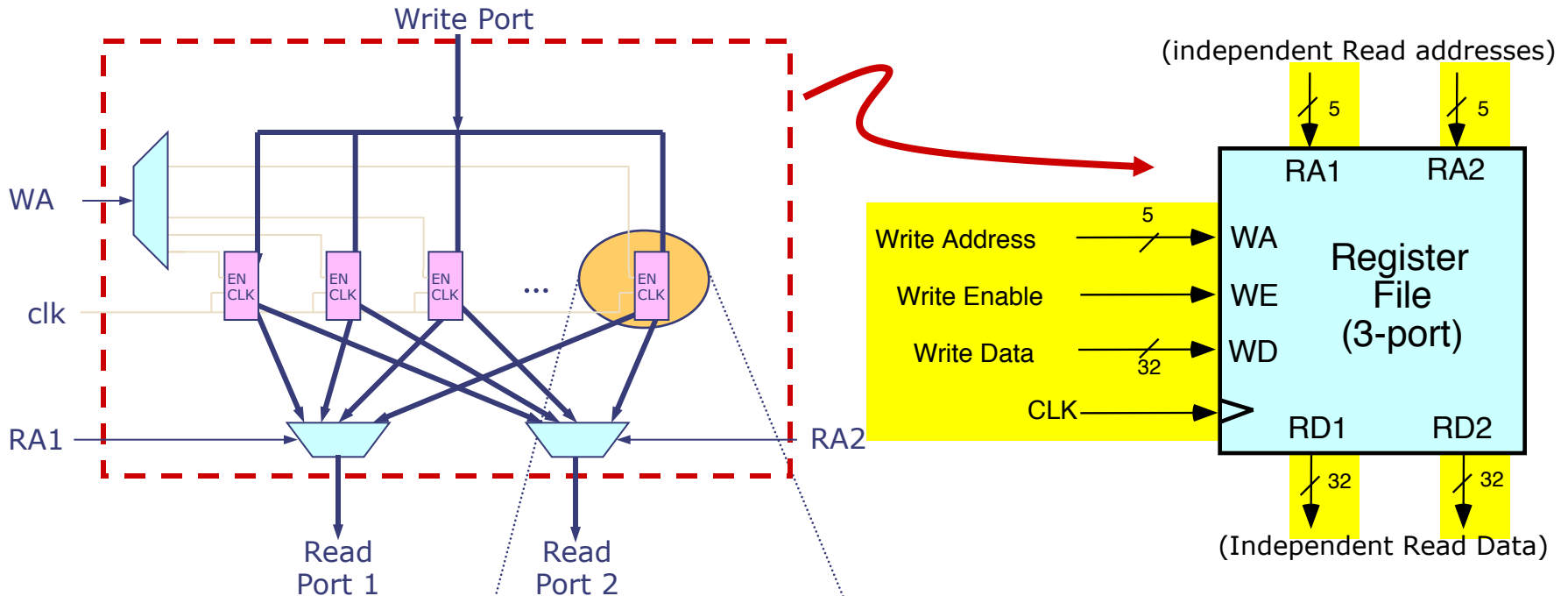
"Black box" ALU

Register File



Memories

Multi-Ported Register File



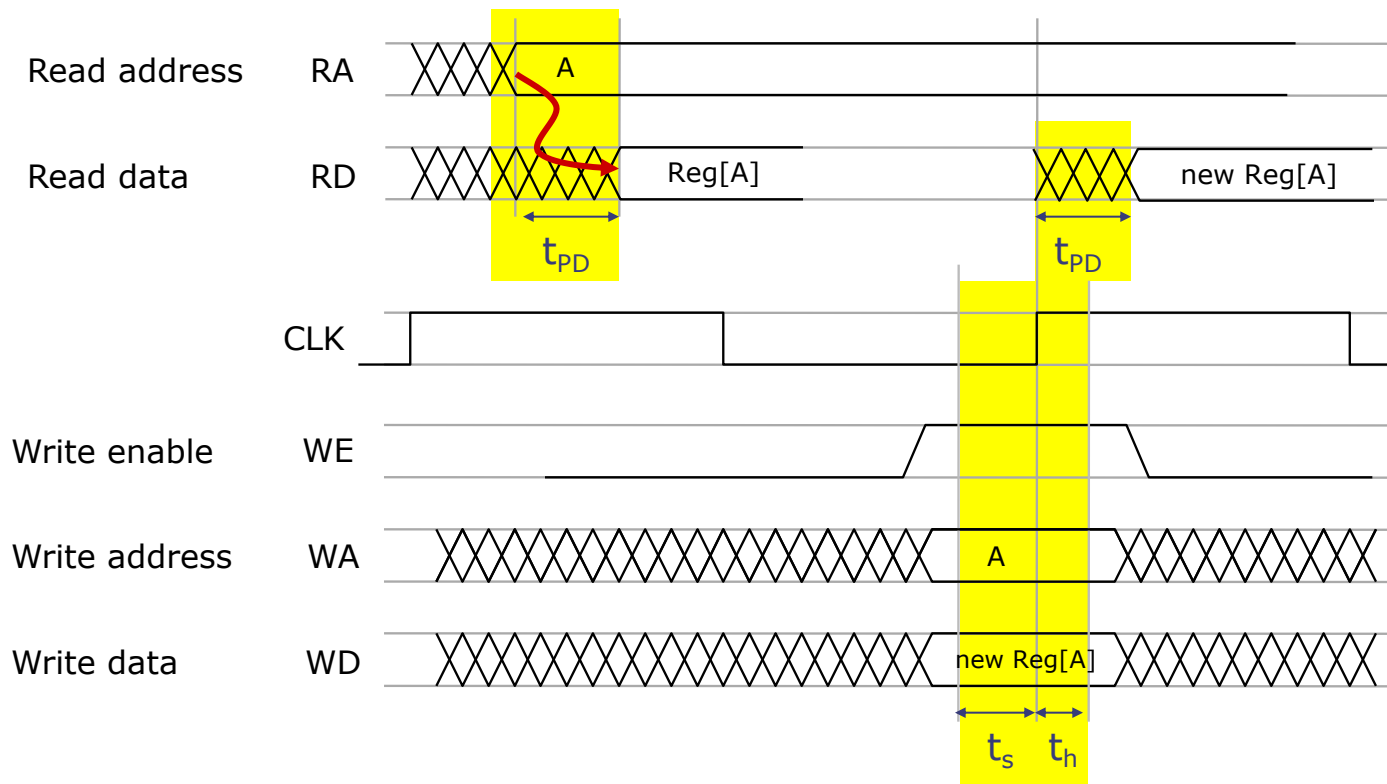
Load-enabled register

2 combinational READ ports*,
1 clocked WRITE port

*internal logic ensures Reg[0] reads as 0

Register File Timing

2 combinational READ ports, 1 clocked WRITE port



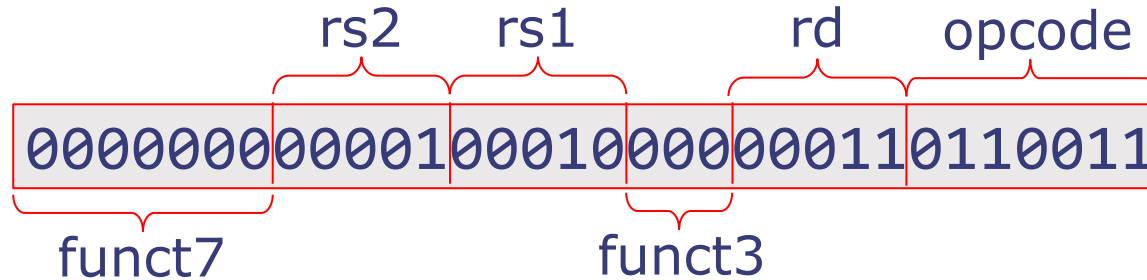
What if $WA=RA1$?

RD1 reads "old" value of Reg[RA1]
until next clock edge!

Memory Timing

- For now (lab 6), we will assume that our memories behave just like our register file in terms of timing.
 - Loads are combinational – data is returned in the same clock cycle as load request.
 - Stores are clocked
 - In lab 6, you will see the memory module referred to as a **magic memory** since its not really realistic.
- Next week we will learn about various different memory models and their tradeoffs.
- In the design project, we will use realistic memories for our processor.

ALU Instructions

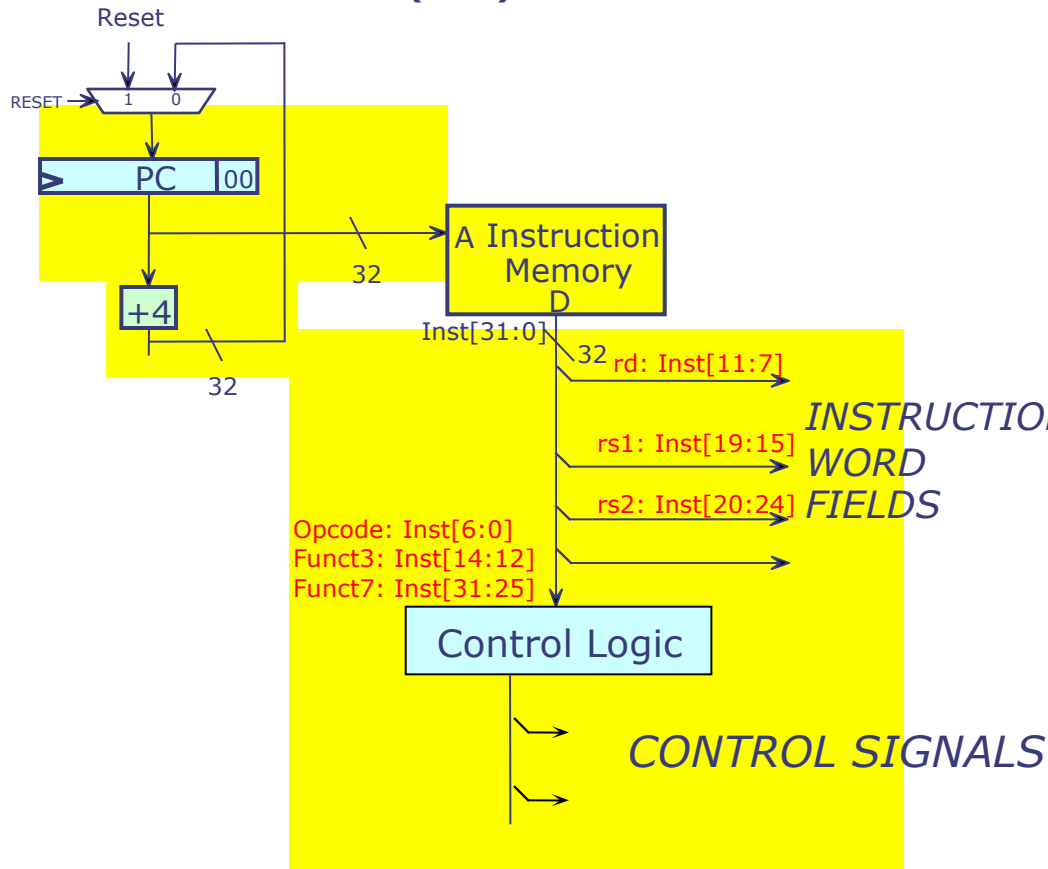


- What RISC-V instruction is represented by these 32 bits?
- Reference manual specifies the fields as follows:
 - opcode = 0110011 => opCode Op, R-type encoding
 - funct3 = 000 => ADD
 - funct7 = 0000000
 - rd = 00011 => x3
 - rs1 = 00010 => x2
 - rs2 = 00001 => x1

ADD x3, x2, x1

Instruction Fetch/Decode

Use a counter to FETCH the next instruction: PROGRAM COUNTER (PC)



- Use PC as memory address
- Add 4 to PC, load new value at end of cycle
- Fetch instruction from memory
- Decode instruction:
 - Use some instruction fields directly (register numbers, immediate values)
 - Use opcode, funct3, and funct7 bits to generate control signals

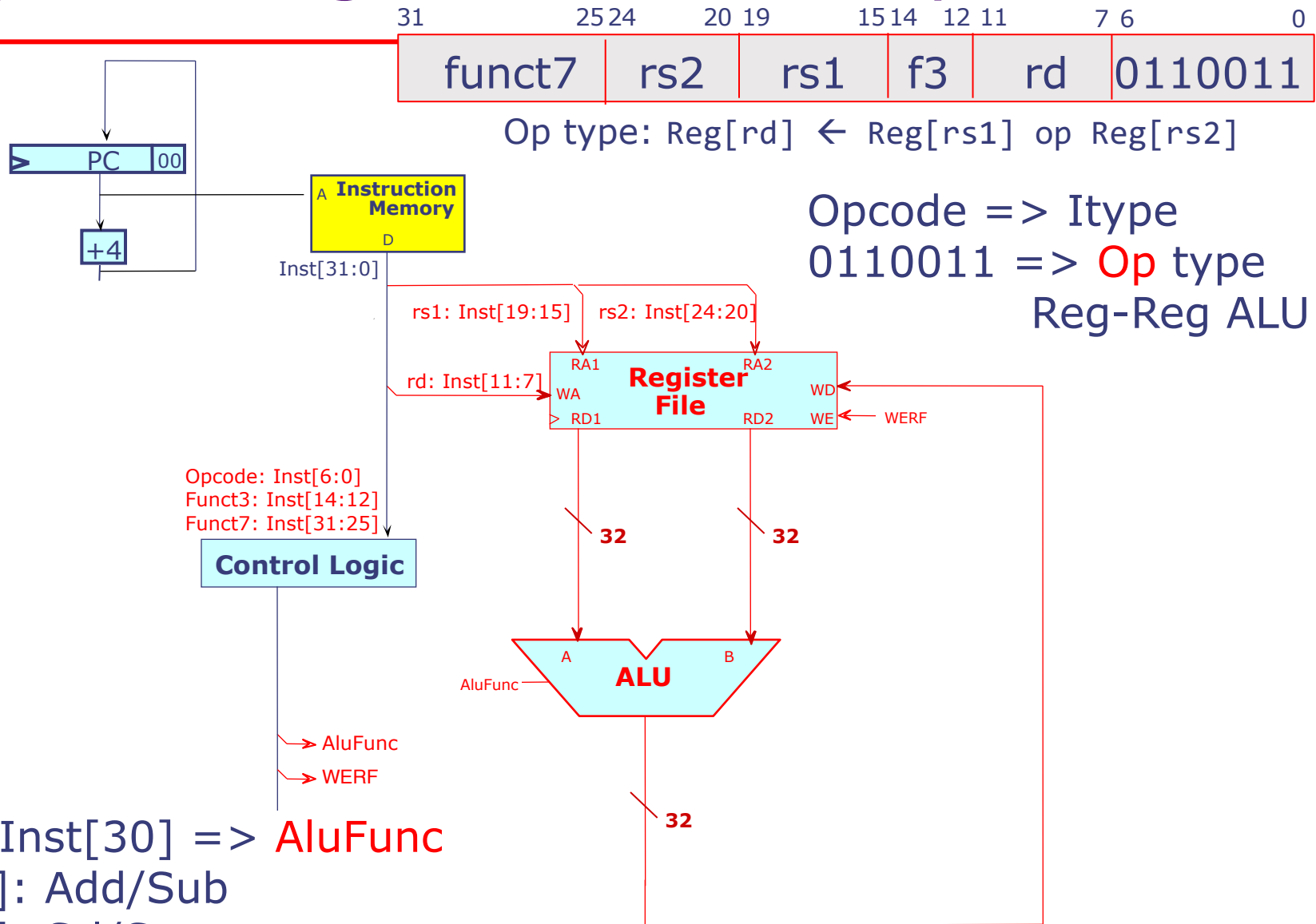
ALU Instructions

Differ only in the ALU op to be performed

Instruction	Description	Execution
ADD rd, rs1, rs2	Add	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] + \text{reg}[\text{rs2}]$
SUB rd, rs1, rs2	Sub	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] - \text{reg}[\text{rs2}]$
SLL rd, rs1, rs2	Shift Left Logical	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \ll \text{reg}[\text{rs2}]$
SLT rd, rs1, rs2	Set if < (Signed)	$\text{reg}[\text{rd}] \leftarrow (\text{reg}[\text{rs1}] <_s \text{reg}[\text{rs2}]) ? 1 : 0$
SLTU rd, rs1, rs2	Set if < (Unsigned)	$\text{reg}[\text{rd}] \leftarrow (\text{reg}[\text{rs1}] <_u \text{reg}[\text{rs2}]) ? 1 : 0$
XOR rd, rs1, rs2	Xor	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \wedge \text{reg}[\text{rs2}]$
SRL rd, rs1, rs2	Shift Right Logical	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \gg_u \text{reg}[\text{rs2}]$
SRA rd, rs1, rs2	Shift Right Arithmetic	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \gg_s \text{reg}[\text{rs2}]$
OR rd, rs1, rs2	Or	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \text{reg}[\text{rs2}]$
AND rd, rs1, rs2	And	$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \& \text{reg}[\text{rs2}]$

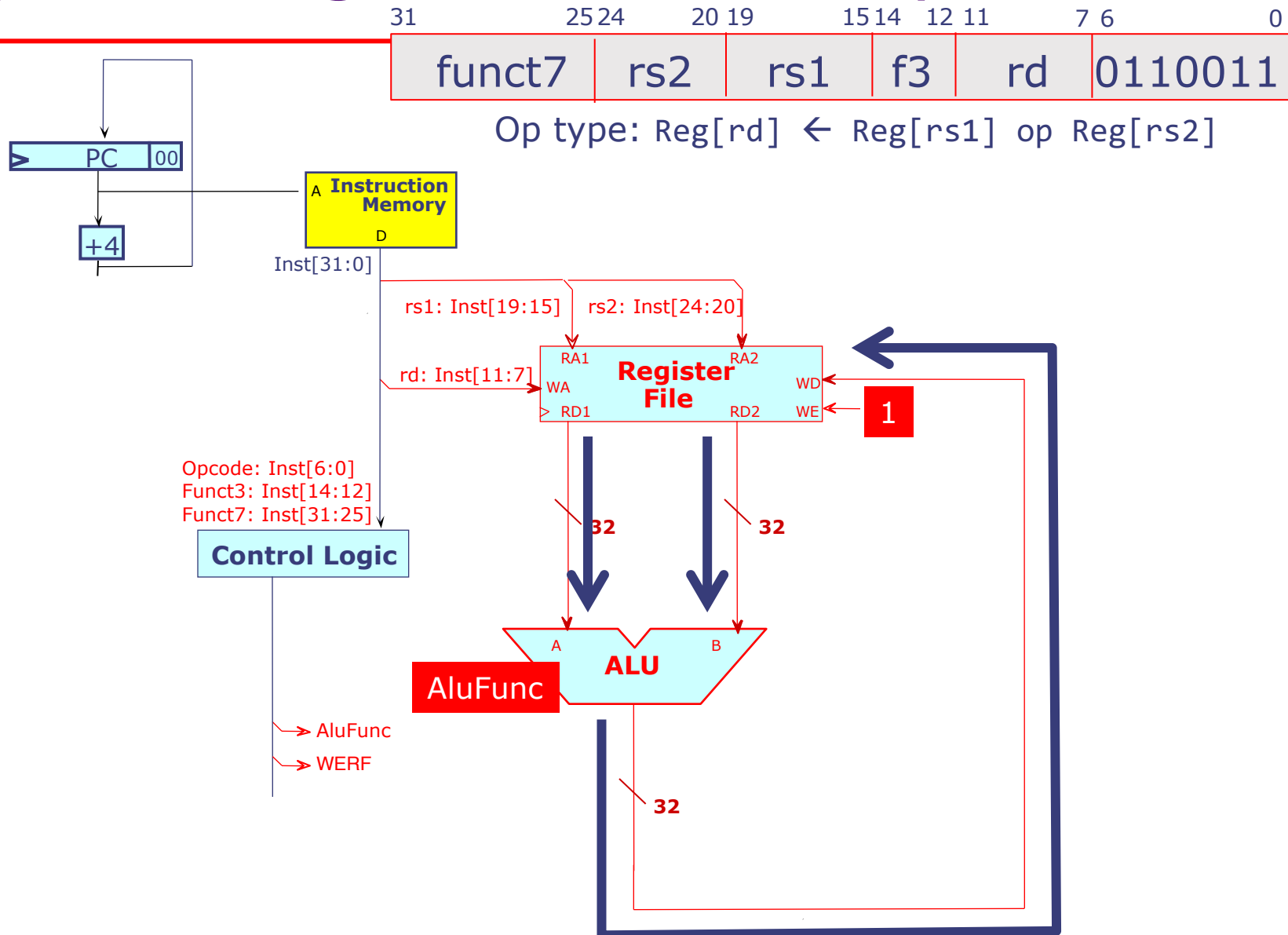
These instructions are grouped in a category called OP with fields (AluFunc, rd, rs1, rs2)

Register-Register ALU Datapath



funct3, Inst[30] => **AluFunc**
 Inst[30]: Add/Sub
 Inst[30]: Srl/Sra

Register-Register ALU Datapath



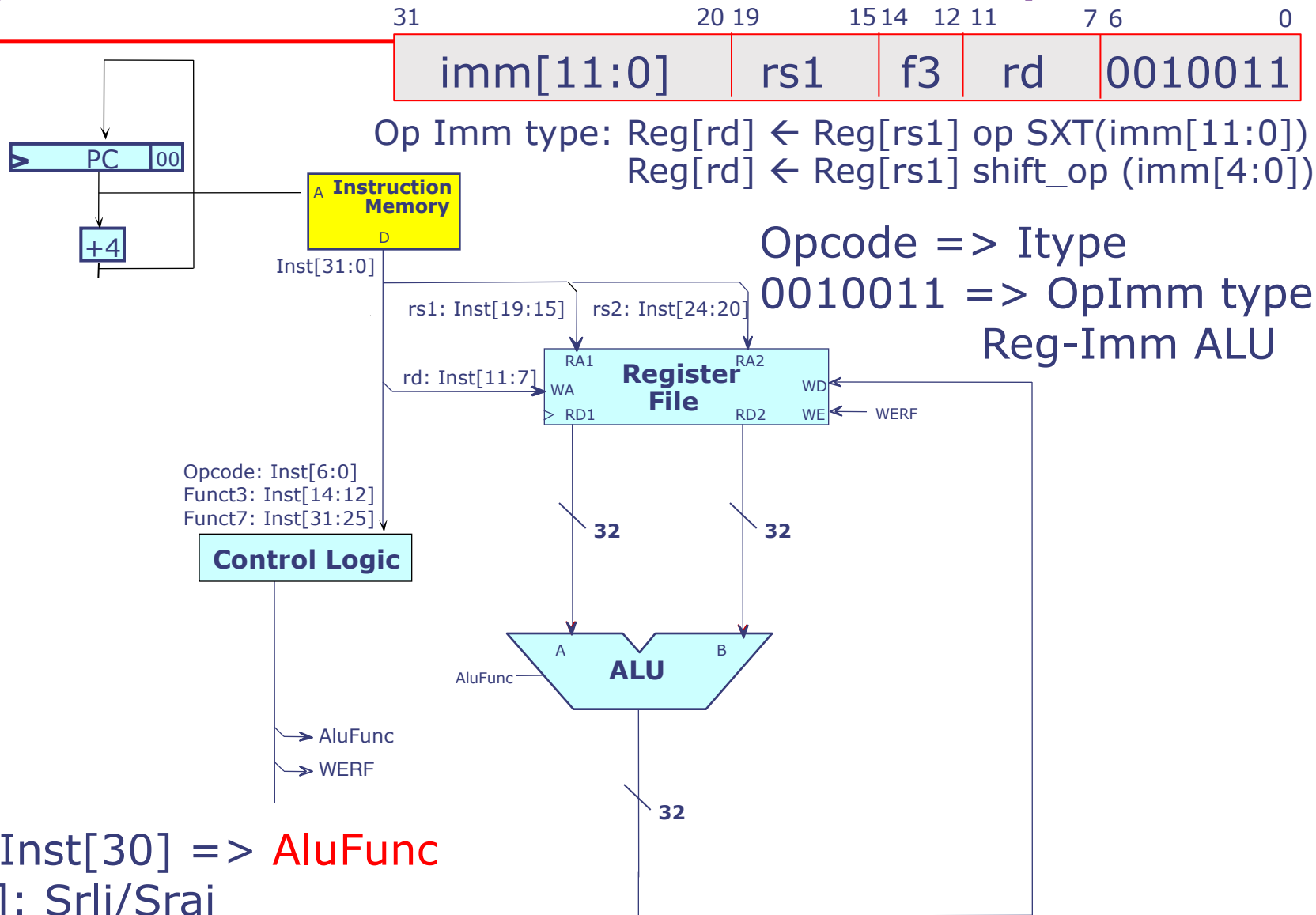
ALU Instructions

with one Immediate operand

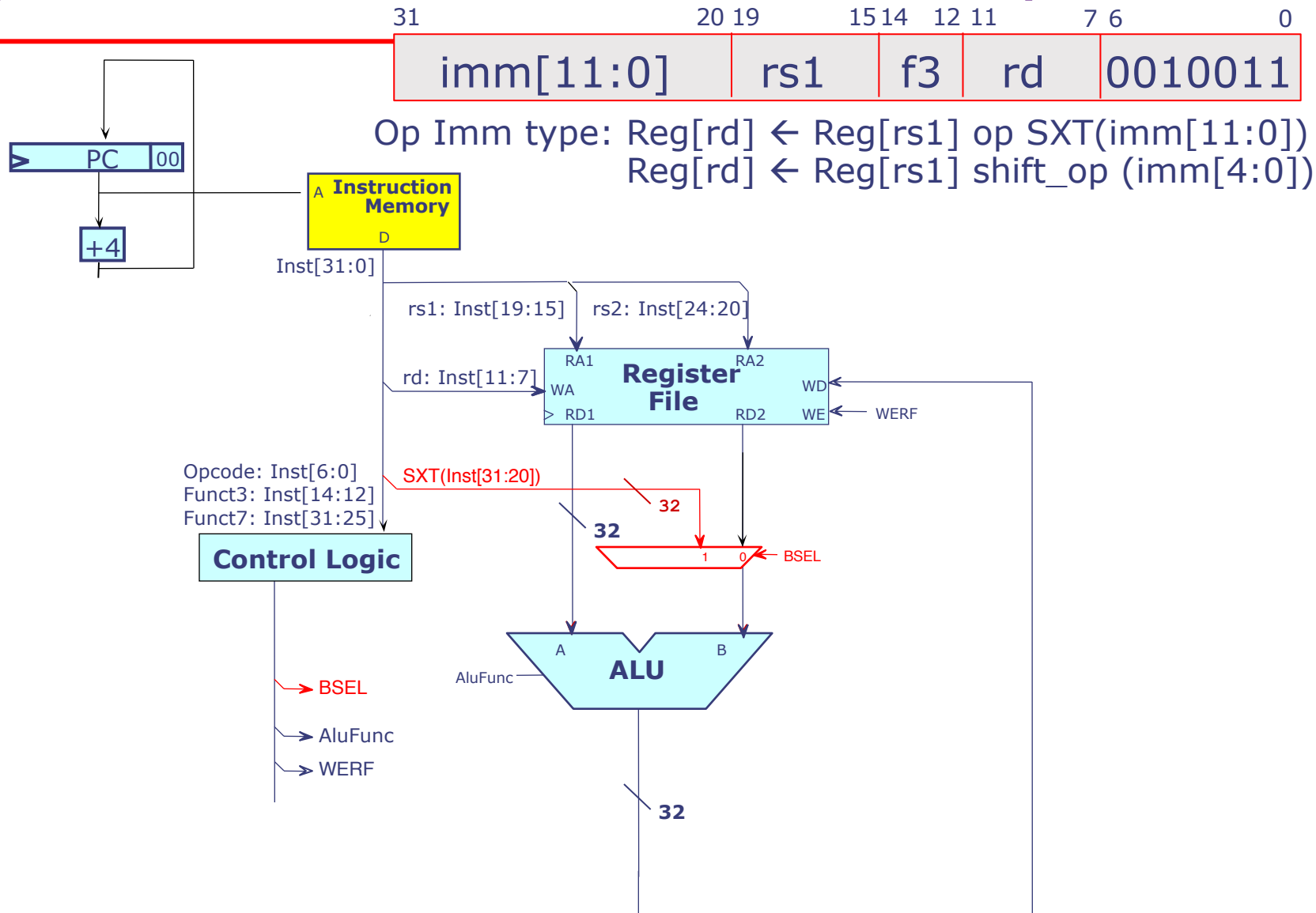
Instruction	Description	Execution
ADDI rd, rs1, immI	Add Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] + \text{immI}$
SLTI rd, rs1, immI	Set if < Immediate (Signed)	$\text{reg}[\text{rd}] \leq (\text{reg}[\text{rs1}] <_s \text{immI}) ? 1 : 0$
SLTIU rd, rs1, immI	Set if < Immediate (Unsigned)	$\text{reg}[\text{rd}] \leq (\text{reg}[\text{rs1}] <_u \text{immI}) ? 1 : 0$
XORI rd, rs1, immI	Xor Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \wedge \text{immI}$
ORI rd, rs1, immI	Or Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \mid \text{immI}$
ANDI rd, rs1, immI	And Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \& \text{immI}$
SLLI rd, rs1, immI	Shift Left Logical Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \ll \text{immI}$
SRLI rd, rs1, immI	Shift Right Logical Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \gg_u \text{immI}$
SRAI rd, rs1, immI	Shift Right Arithmetic Immediate	$\text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \gg_s \text{immI}$

These instructions are grouped in a category called OPIMM with fields (AluFunc, rd, rs1, immI)

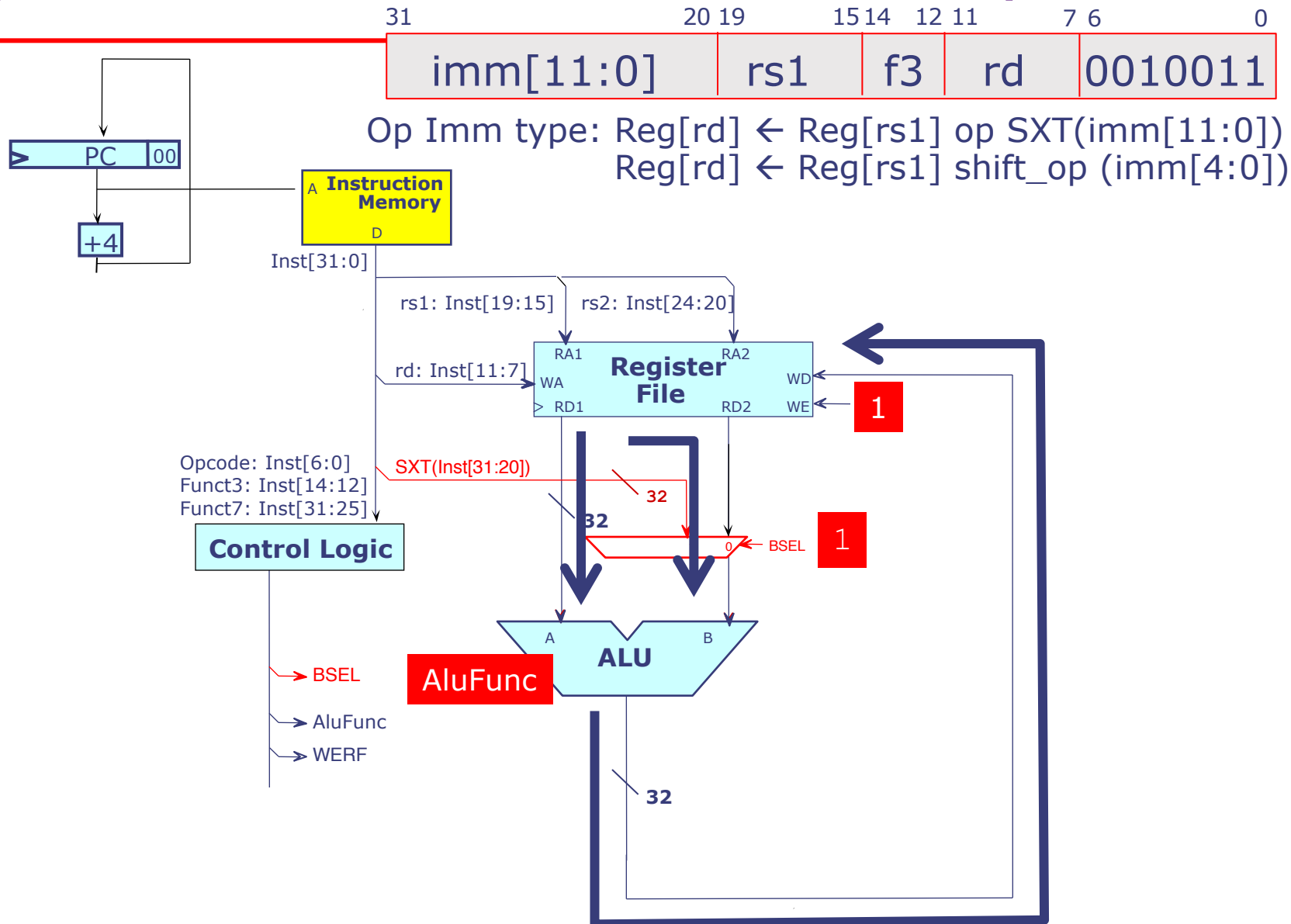
Register-Immediate ALU Datapath



Register-Immediate ALU Datapath



Register-Immediate ALU Datapath

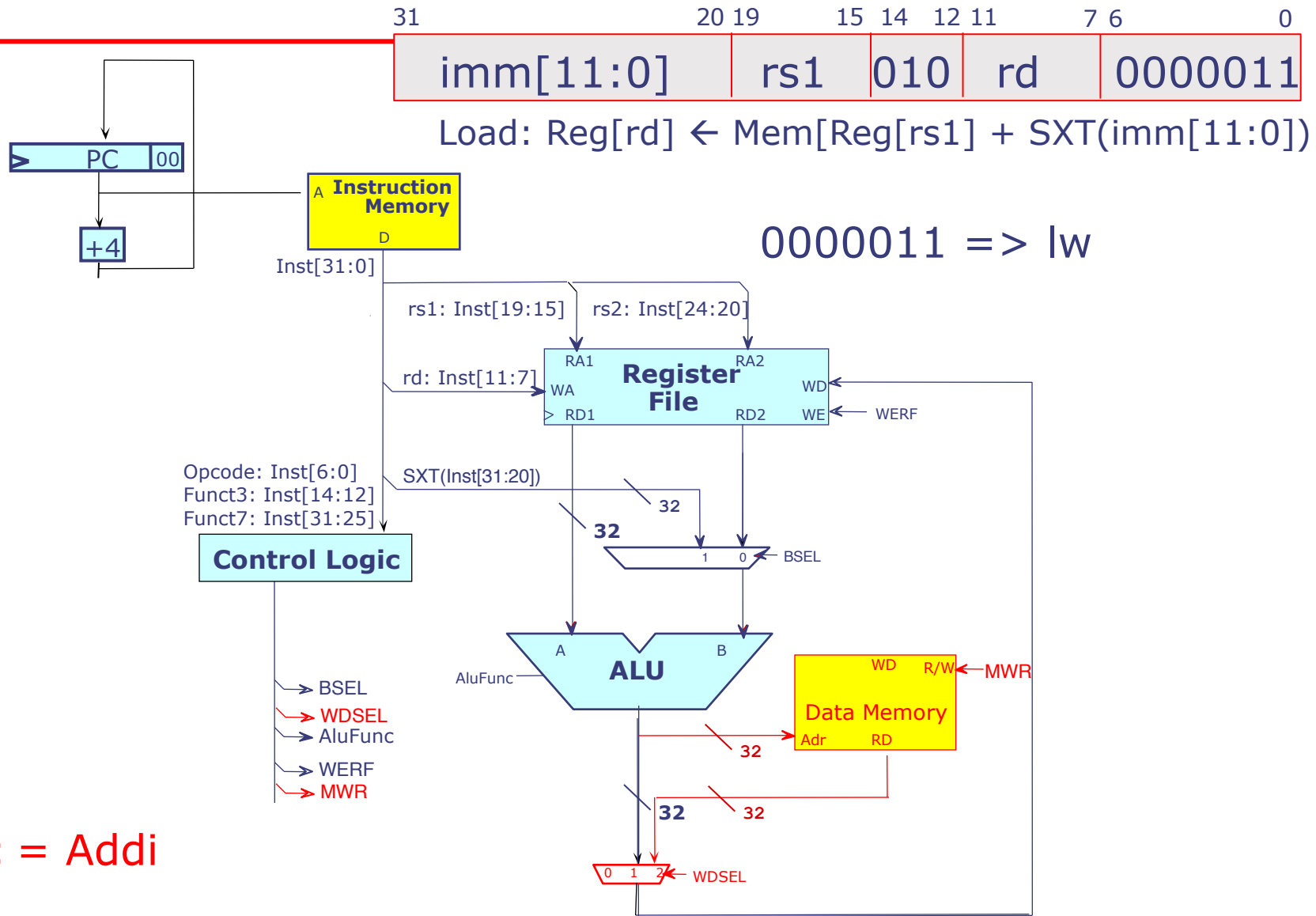


Load and Store Instructions

Instruction	Description	Execution
LW rd, immI(rs1)	Load Word	$\text{reg}[\text{rd}] \leftarrow \text{mem}[\text{reg}[\text{rs1}] + \text{immI}]$
SW rs2, immS(rs1)	Store Word	$\text{mem}[\text{reg}[\text{rs1}] + \text{immS}] \leftarrow \text{reg}[\text{rs2}]$

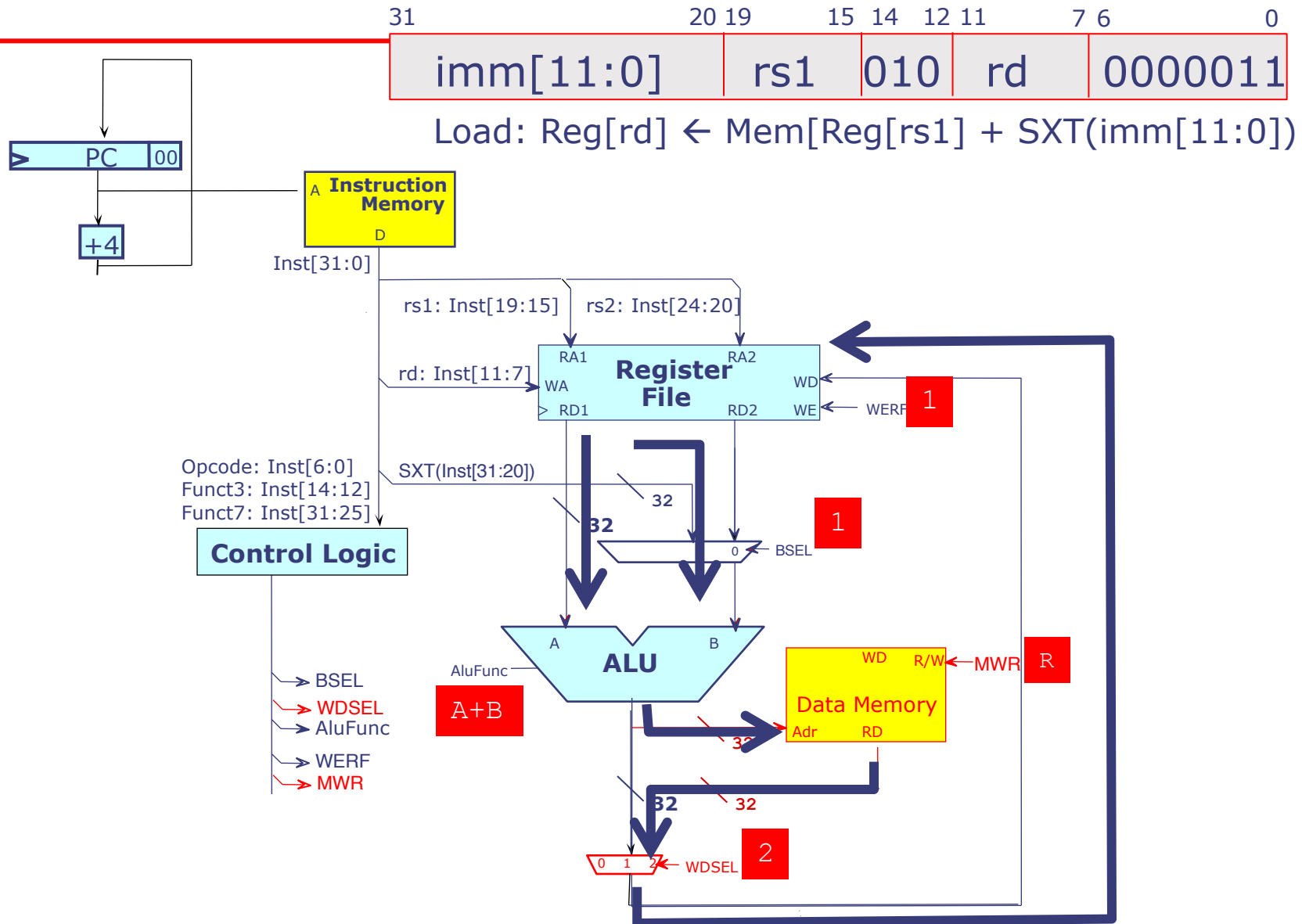
LW and SW need to access memory for execution and thus, are required to compute an effective memory address

Load Instruction

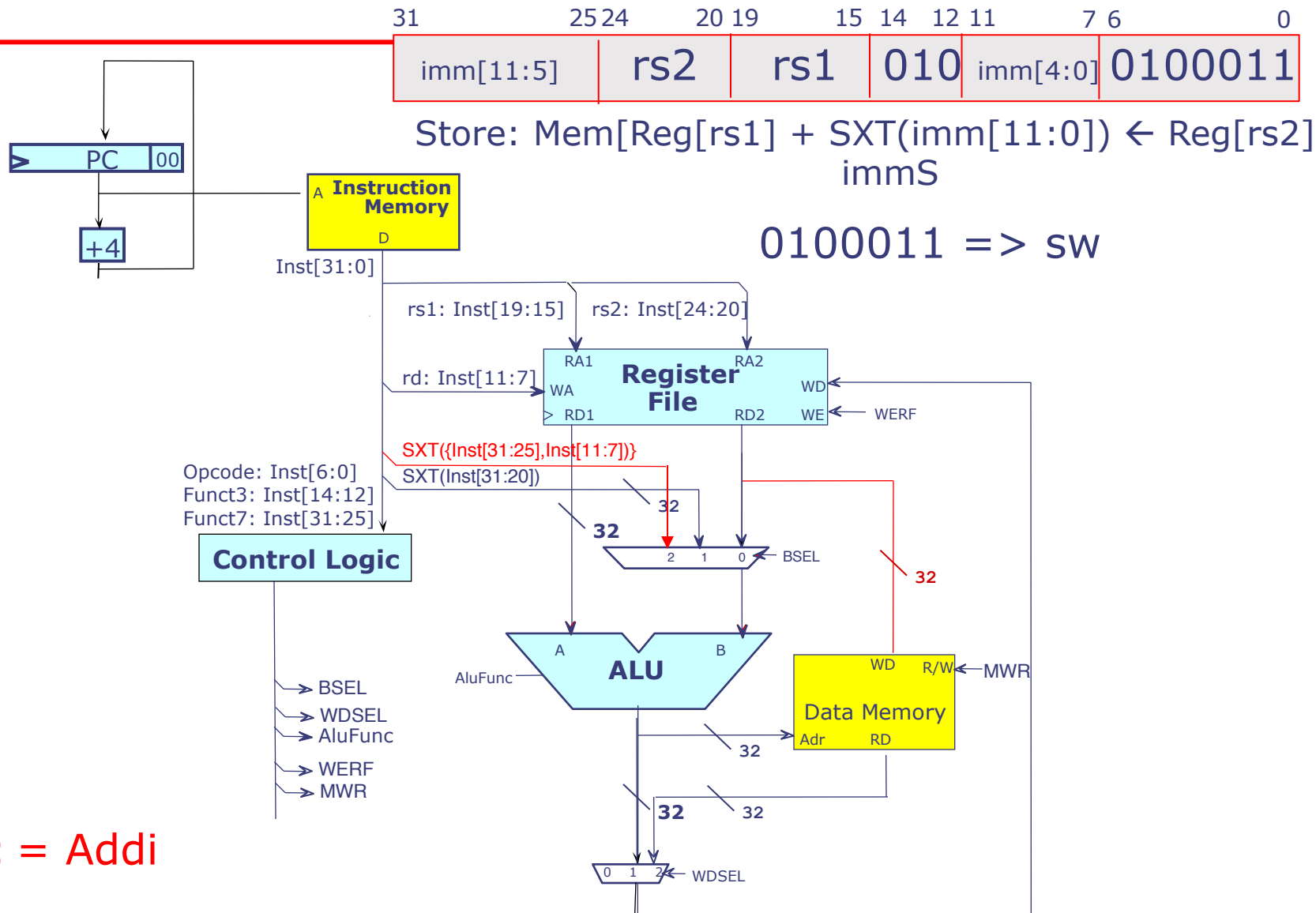


AluFunc = Addi

Load Instruction

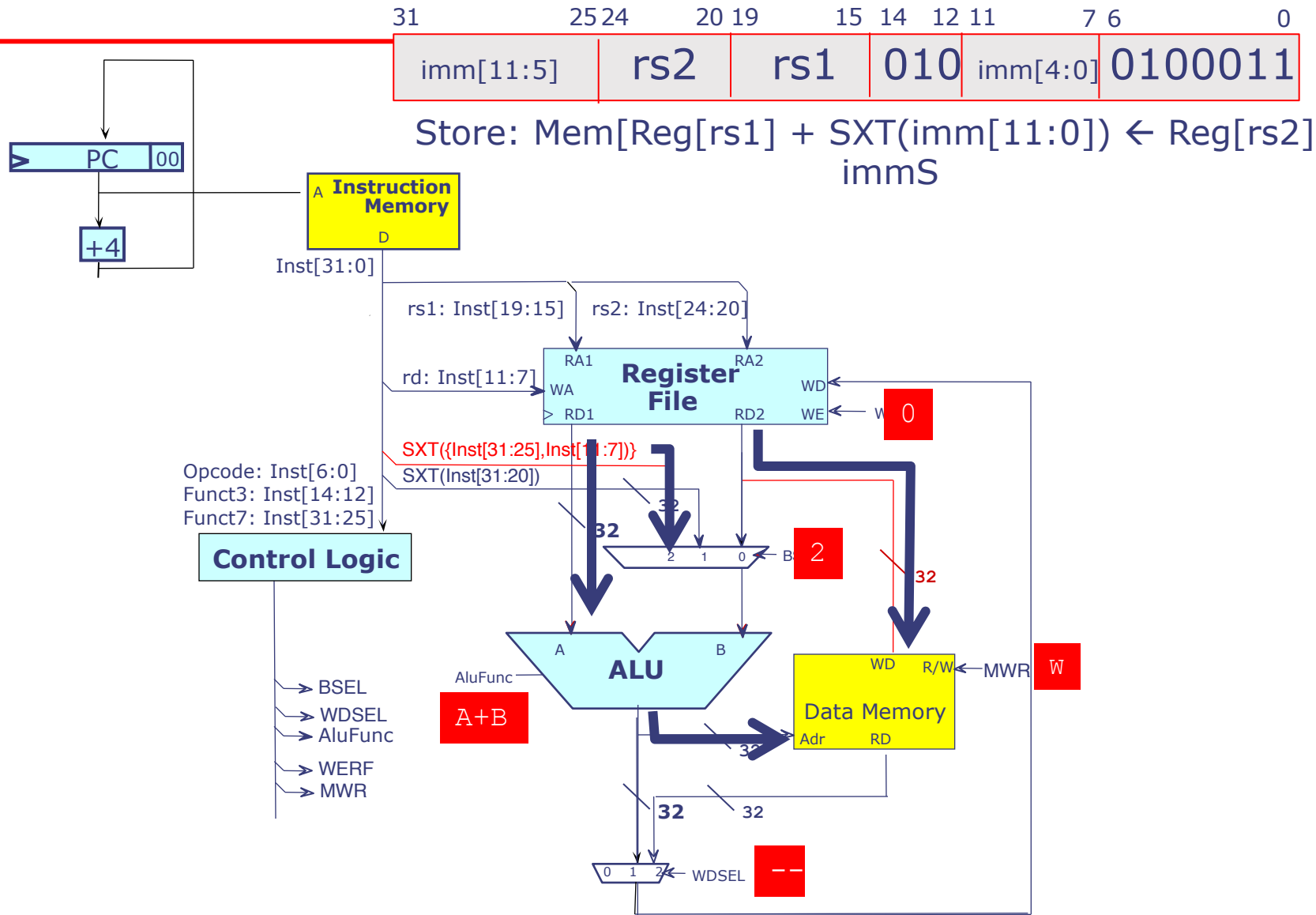


Store Instruction



AluFunc = Addi

Store Instruction



Branch Instructions

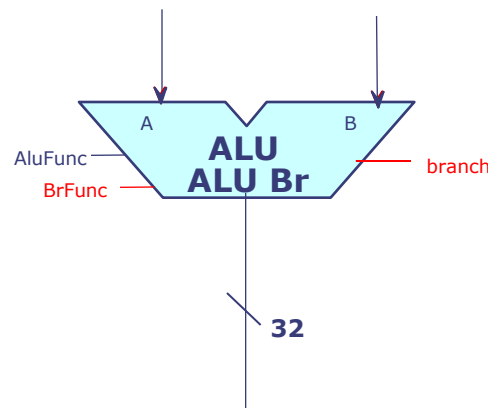
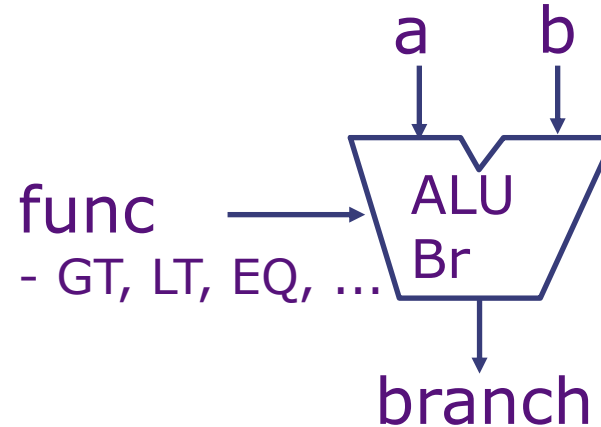
differ only in the aluBr operation they perform

Instruction	Description	Execution
BEQ rs1, rs2, immB	Branch =	$pc \leq (\text{reg}[\text{rs1}] == \text{reg}[\text{rs2}]) ? pc + \text{immB} : pc + 4$
BNE rs1, rs2, immB	Branch !=	$pc \leq (\text{reg}[\text{rs1}] != \text{reg}[\text{rs2}]) ? pc + \text{immB} : pc + 4$
BLT rs1, rs2, immB	Branch < (Signed)	$pc \leq (\text{reg}[\text{rs1}] <_s \text{reg}[\text{rs2}]) ? pc + \text{immB} : pc + 4$
BGE rs1, rs2, immB	Branch \geq (Signed)	$pc \leq (\text{reg}[\text{rs1}] \geq_s \text{reg}[\text{rs2}]) ? pc + \text{immB} : pc + 4$
BLTU rs1, rs2, immB	Branch < (Unsigned)	$pc \leq (\text{reg}[\text{rs1}] <_u \text{reg}[\text{rs2}]) ? pc + \text{immB} : pc + 4$
BGEU rs1, rs2, immB	Branch \geq (Unsigned)	$pc \leq (\text{reg}[\text{rs1}] \geq_u \text{reg}[\text{rs2}]) ? pc + \text{immB} : pc + 4$

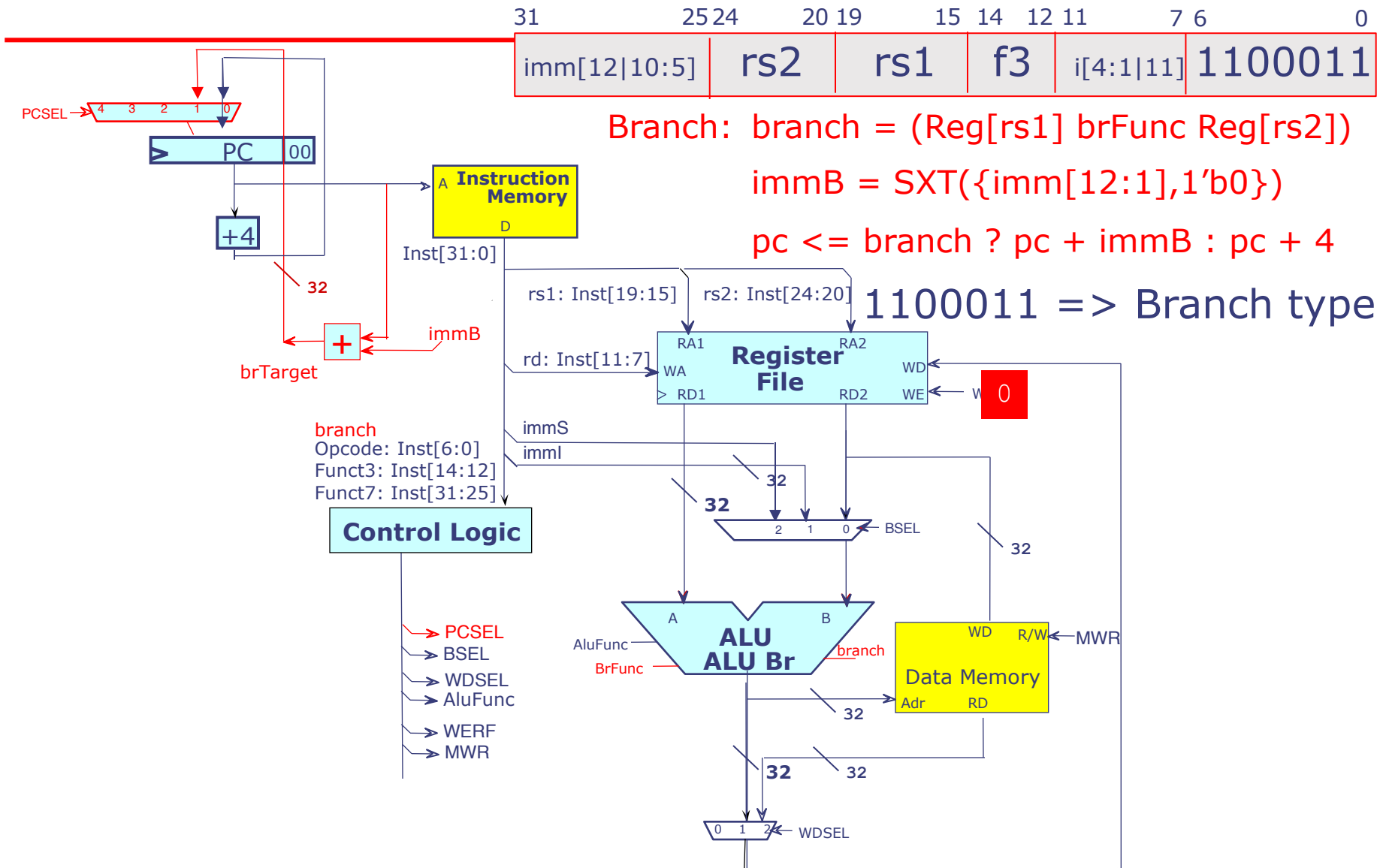
These instructions are grouped in a category called **BRANCH** with fields (brFunc, rs1, rs2, immB)

ALU for Branch Comparisons

Like ALU but
returns a Bool



Branch Instructions

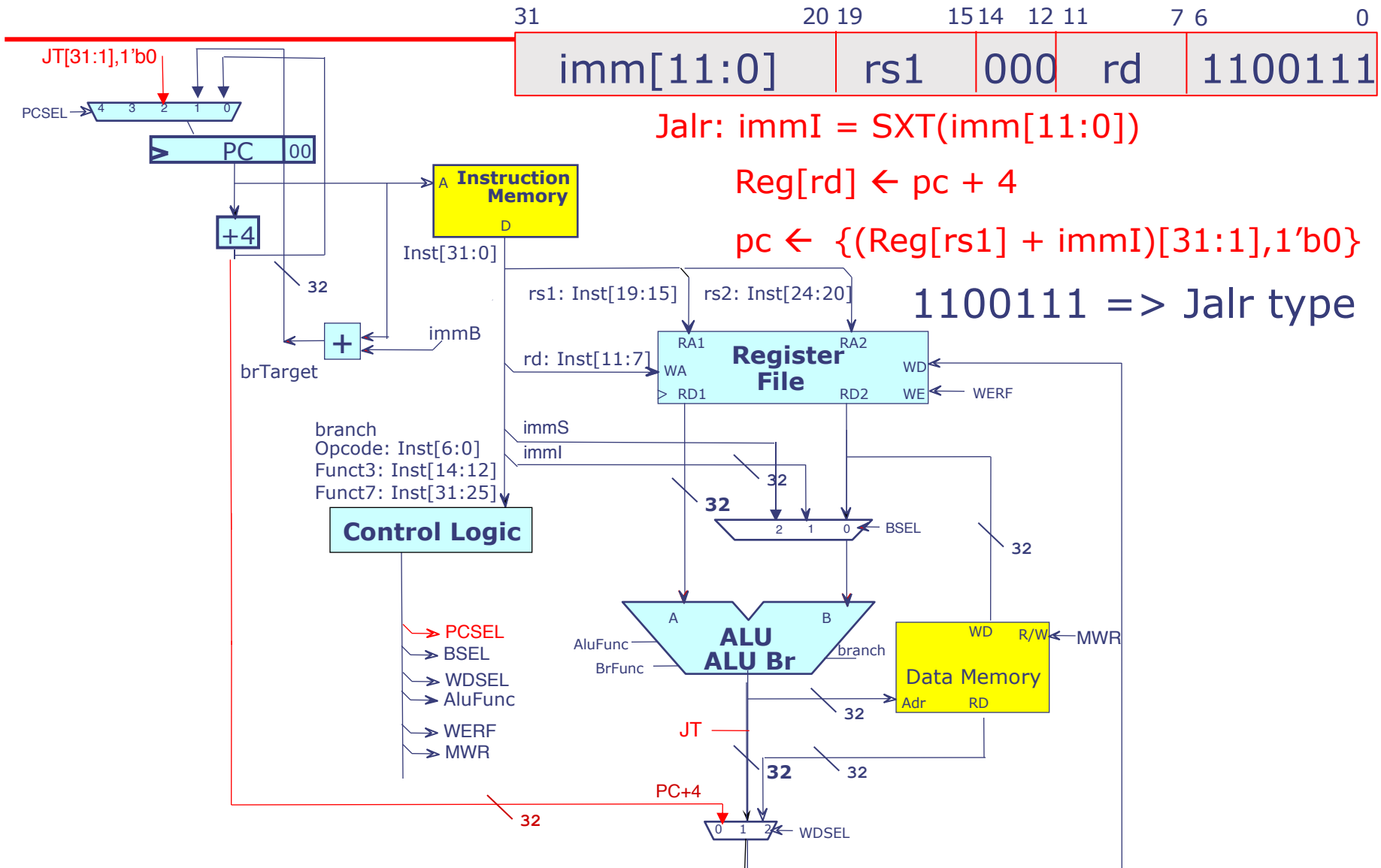


Remaining Instructions

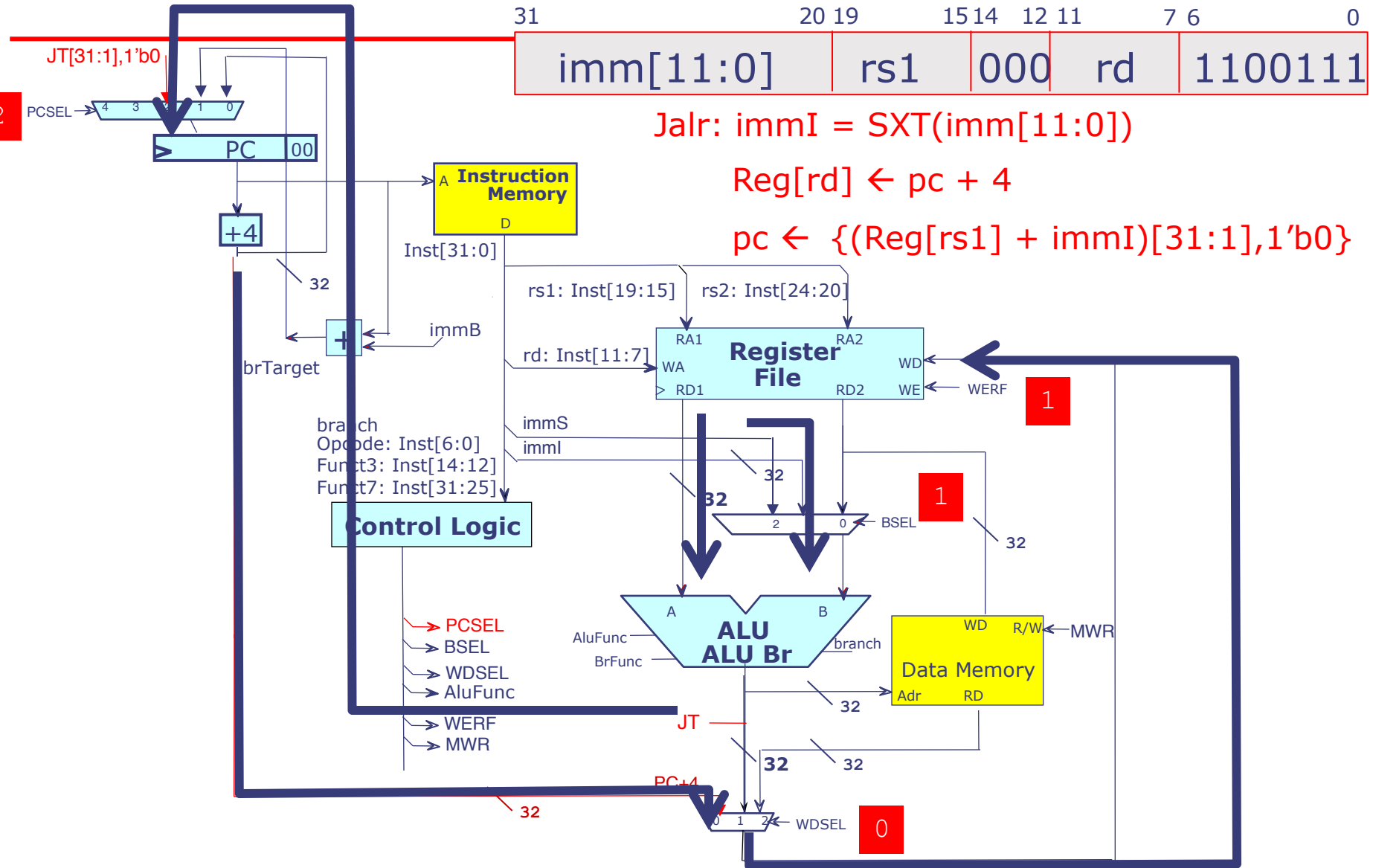
Instruction	Description	Execution
JAL rd, immJ	Jump and Link	$\text{reg}[\text{rd}] \leq \text{pc} + 4$ $\text{pc} \leq \text{pc} + \text{immJ}$
JALR rd, immI(rs1)	Jump and Link Register	$\text{reg}[\text{rd}] \leq \text{pc} + 4$ $\text{pc} \leq \{(\text{reg}[\text{rs1}] + \text{immI})[31:1], 1'b0\}$
LUI rd, immU	Load Upper Immediate	$\text{reg}[\text{rd}] \leq \text{immU}$

Each of these instructions is in a category by itself and needs to extract different fields from the instruction. The jal and jalr instructions update both the pc and reg[rd].

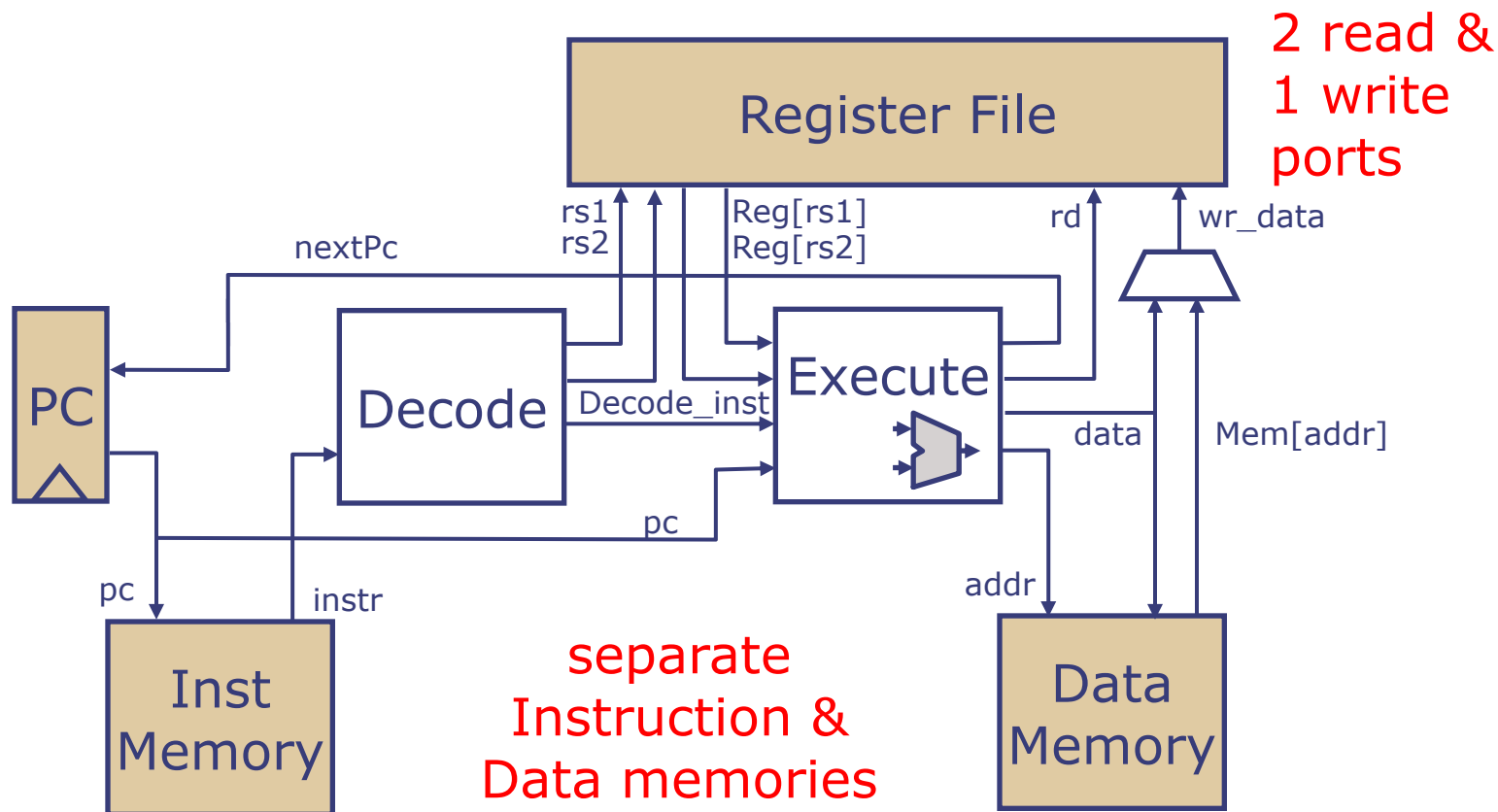
Jalr Instruction



Jalr Instruction



Single-Cycle RISC-V Processor



Instruction decoder

- We need a function to extract the instruction type and the various fields for each type from a 32-bit instruction
- Fields we have identified so far are:
 - Instruction type: OP, OPIMM, BRANCH, JAL, JALR, LUI, LOAD, STORE, Unsupported
 - Function for alu: aluFunc
 - Function for brAlu: brFunc
 - Register fields: rd, rs1, rs2
 - Immediate constants: immI(12), immB(12), immJ(20), immU(20), immS(12) *but* each is used as a 32-bit value with proper sign extension

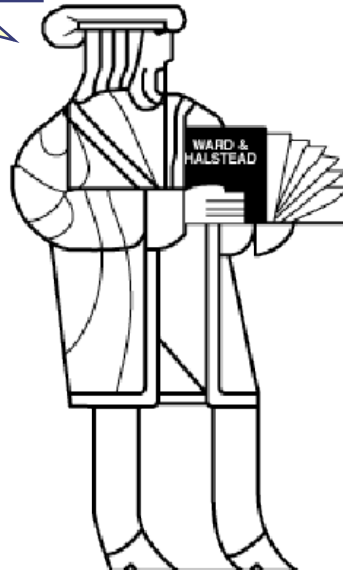
Notice that no instruction has all the fields

Execute Function

- Inputs:
 - Values read from register file
 - Decoded instruction fields
 - PC
- Logic:
 - ALU
 - BrALU
 - NextPC generation
- Outputs:
 - Destination register
 - Data to write to register file or memory
 - Address for load and store operations
 - NextPC

RISC-V Inside

*Is **that** all
there is to
building a
processor???*



*No.
You've gotta print
up all those little
"RISC-V Inside"
stickers.*

