Building a Single-Cycle RISC-V Processor

The von Neumann Model

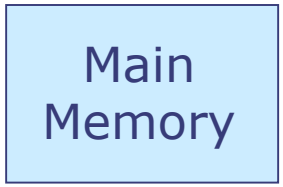
- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)

The von Neumann Model

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

The von Neumann Model

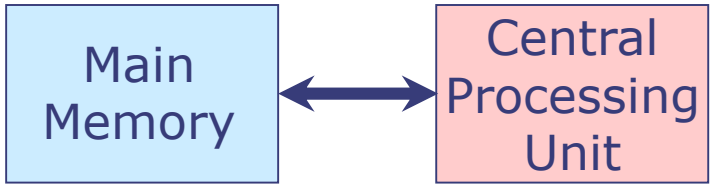
- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:



  - Main memory holds programs and their data

The von Neumann Model

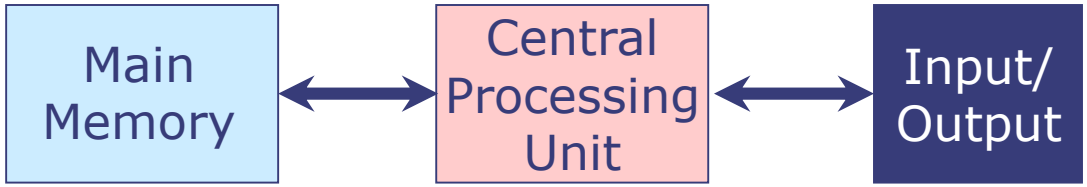
- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:



  - Main memory holds programs and their data
  - Central processing unit accesses and processes memory values

The von Neumann Model

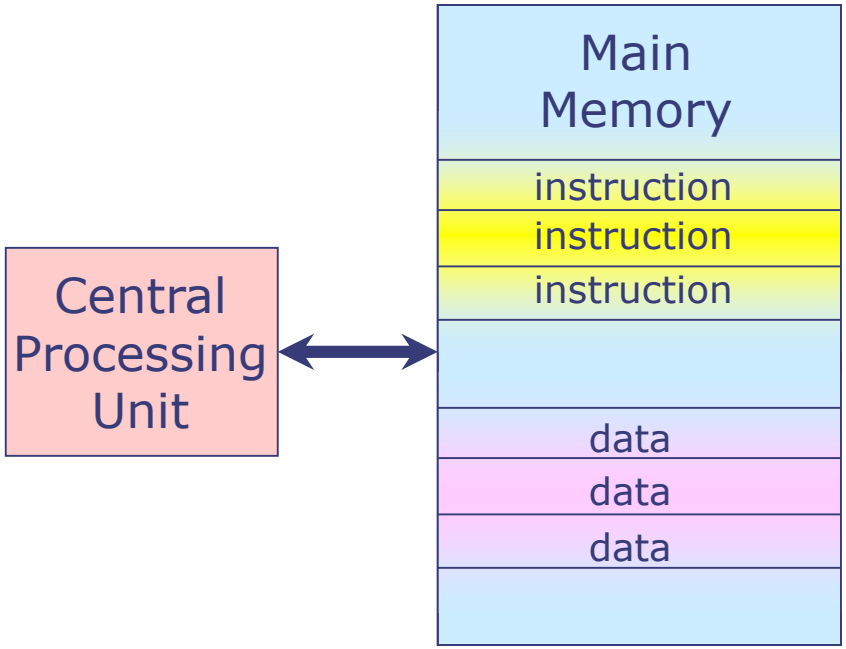
- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:



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

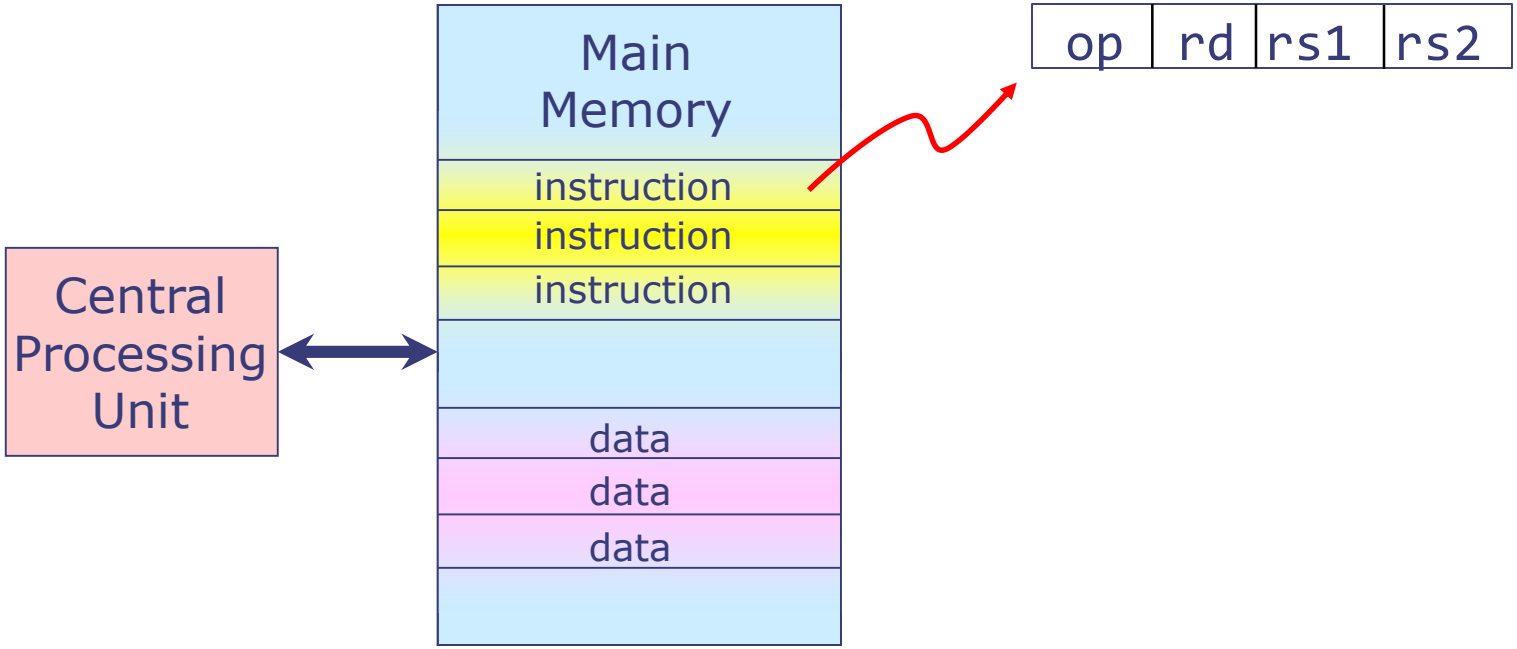
Key Idea: Stored-Program Computer

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



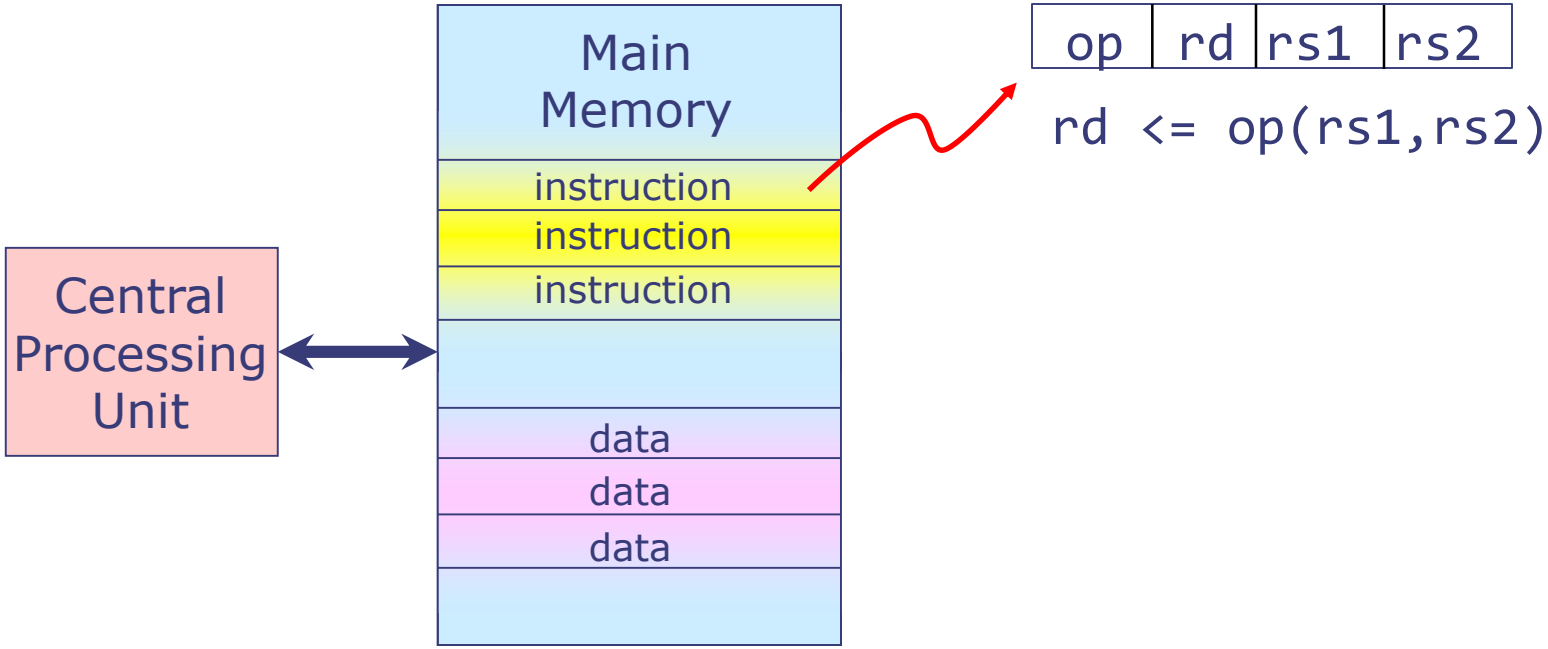
Key Idea: Stored-Program Computer

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



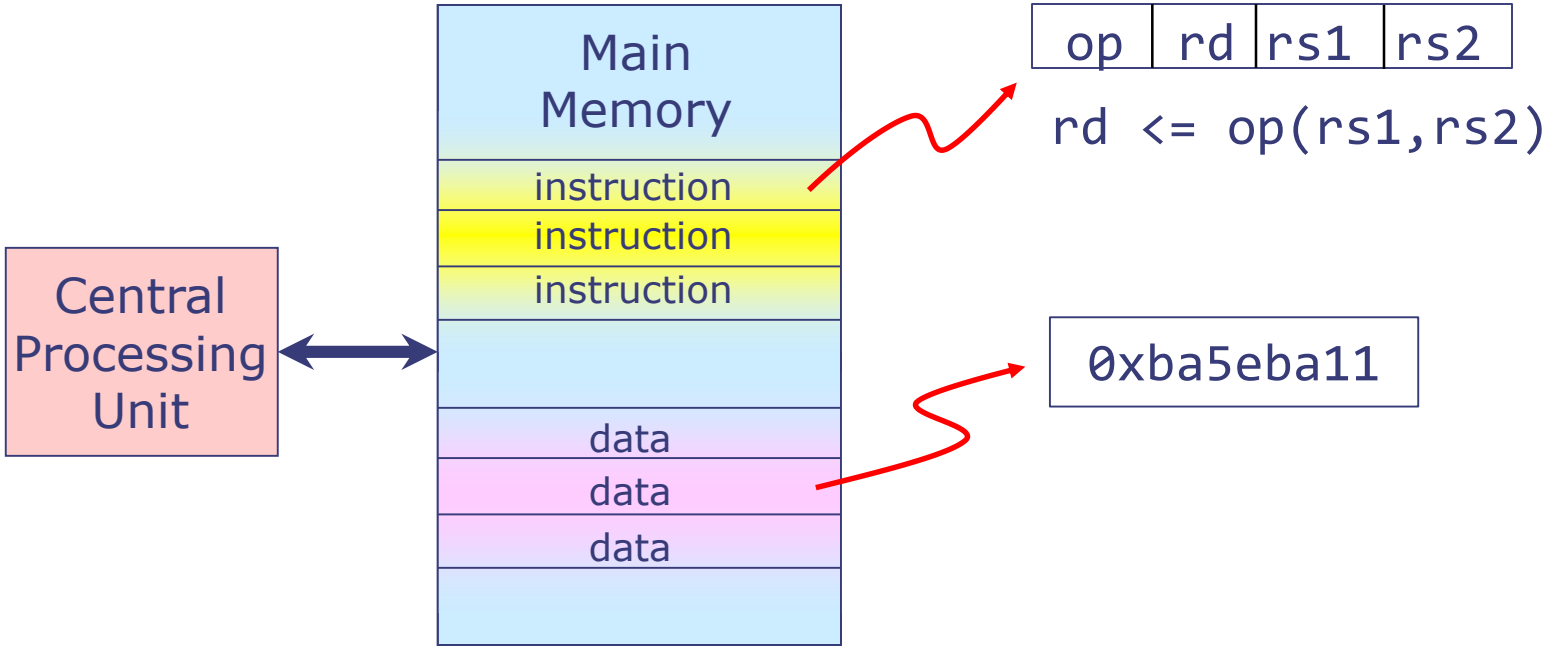
Key Idea: Stored-Program Computer

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



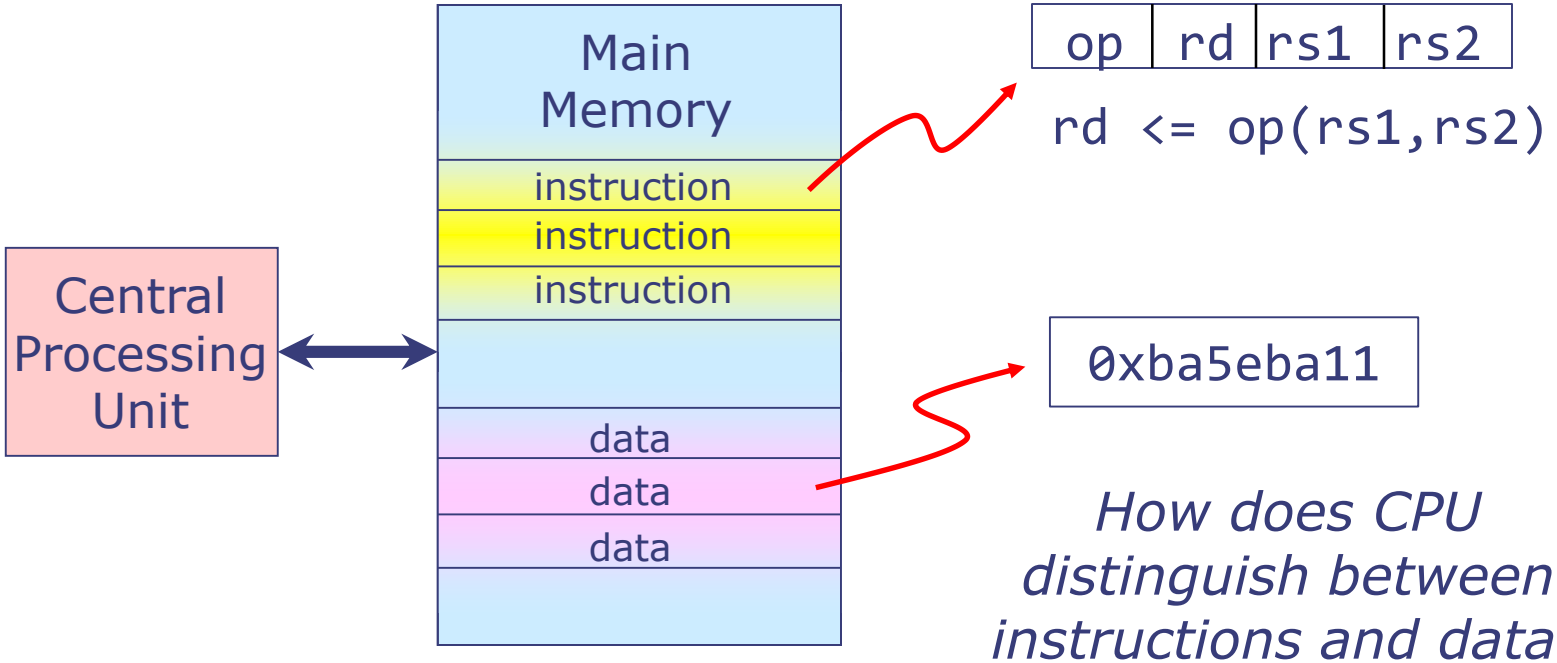
Key Idea: Stored-Program Computer

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

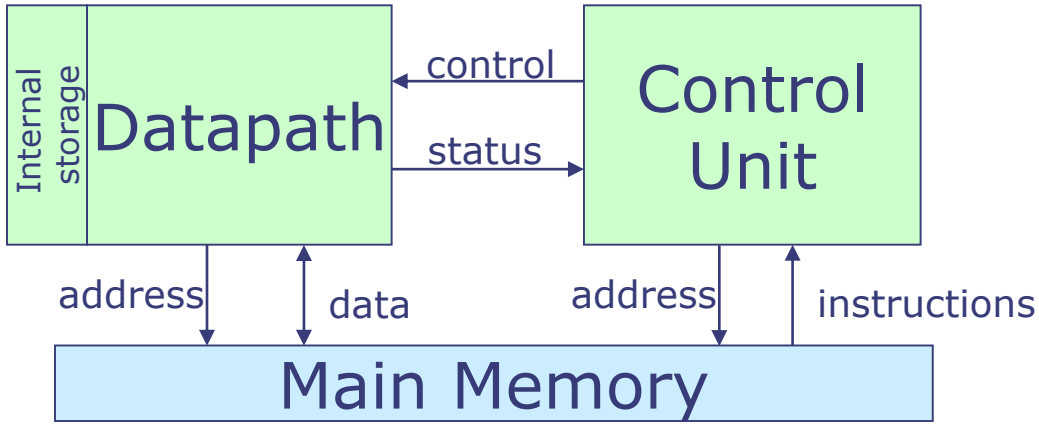


Key Idea: Stored-Program Computer

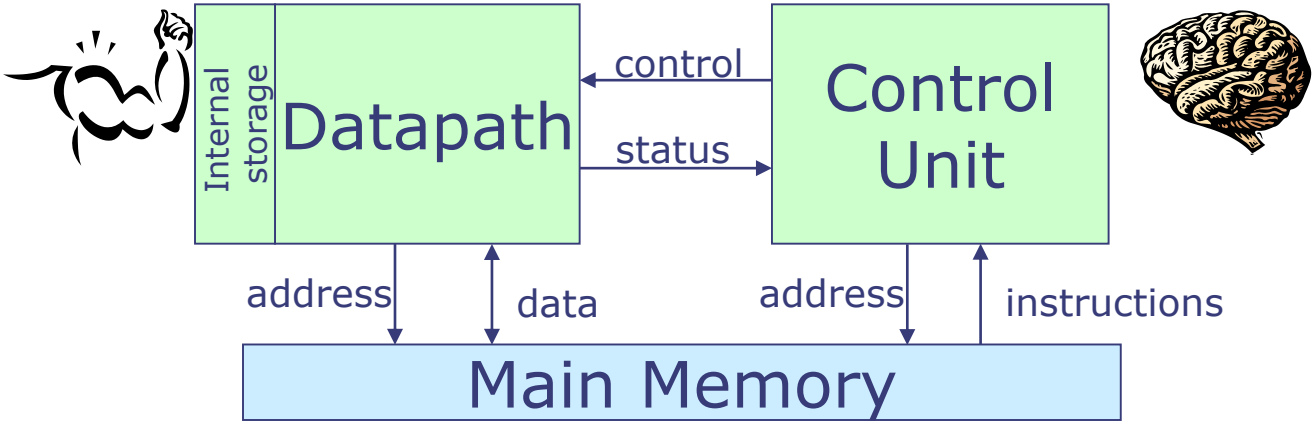
- Express program as a sequence of coded instructions
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program



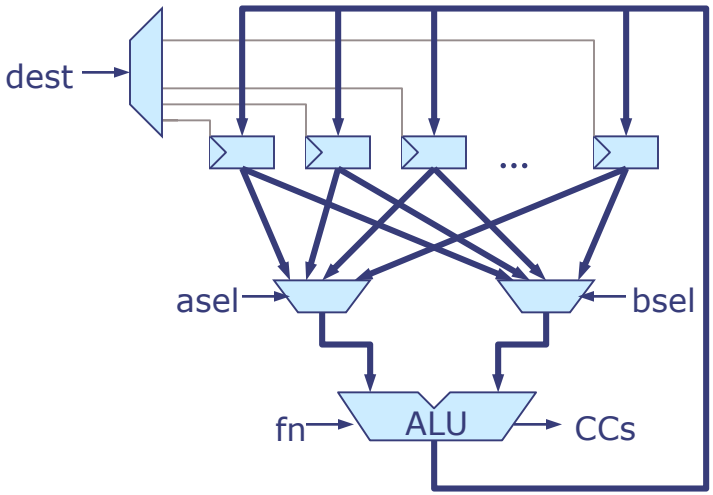
Anatomy of a von Neumann Computer



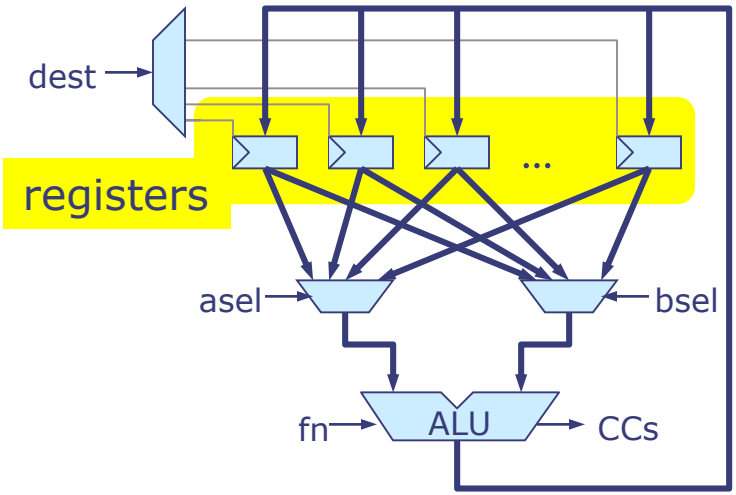
Anatomy of a von Neumann Computer



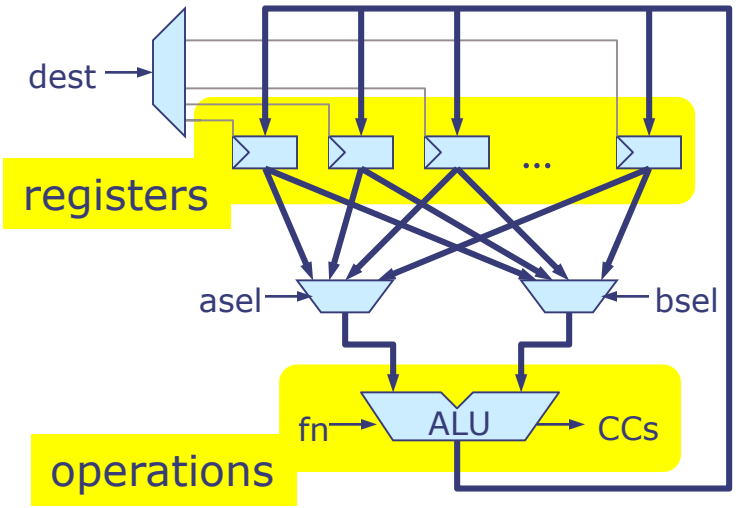
Anatomy of a von Neumann Computer



Anatomy of a von Neumann Computer



Anatomy of a von Neumann Computer



Anatomy of a von Neumann Computer

Internal storage

Datapath

control

status

Control Unit

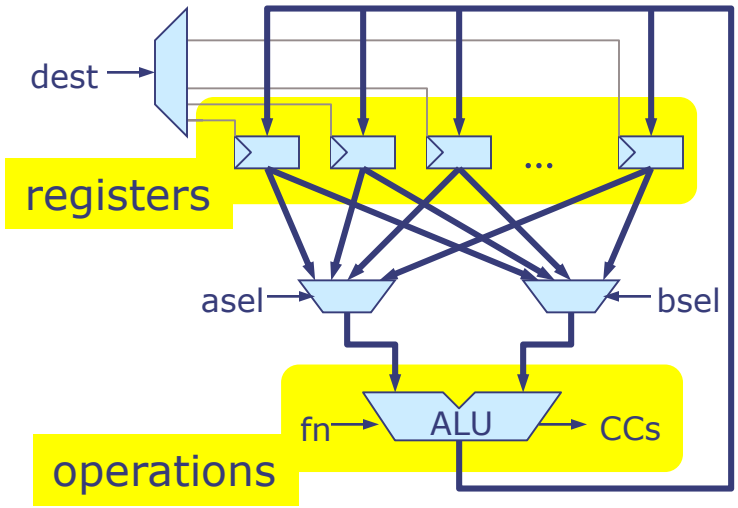
address

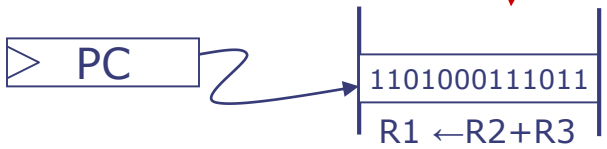
data

address

instructions

Main Memory





- 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

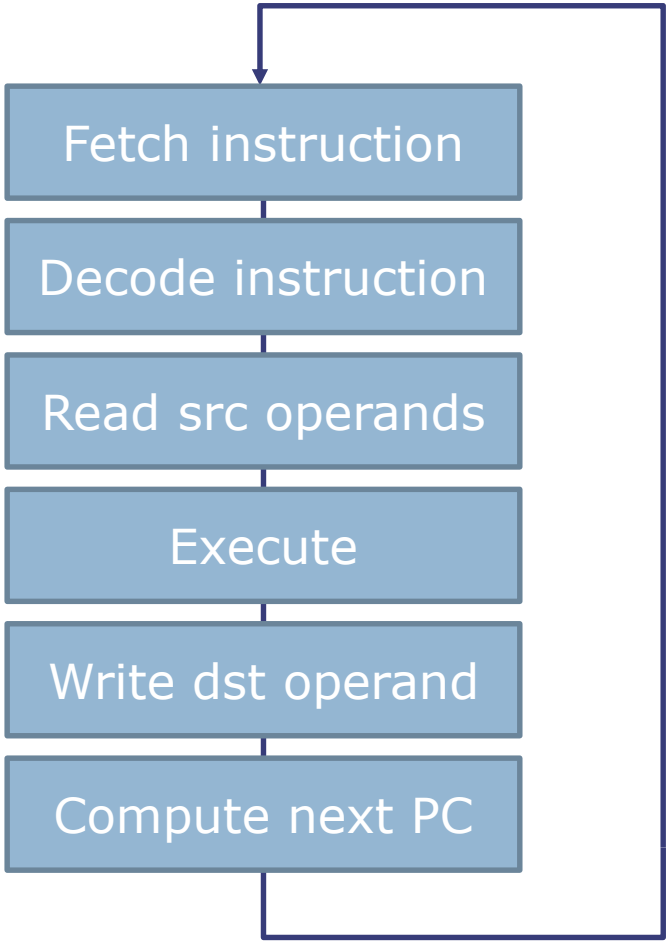
- Instructions are the fundamental unit of work

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

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



Processor Performance

- "Iron Law" of performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructio ns}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instructio n}} \cdot \frac{\text{Time}}{\text{Cycle}} \qquad \text{Perf} = \frac{1}{\text{Time}}$$

Processor Performance

- "Iron Law" of performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructio ns}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instructio n}} \cdot \frac{\text{Time}}{\text{Cycle}} \qquad \text{Perf} = \frac{1}{\text{Time}}$$

- Options to reduce execution time:
  - Executed instructions ↓ (work/instruction ↑)

Processor Performance

- "Iron Law" of performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructio ns}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instructio n}} \cdot \frac{\text{Time}}{\text{Cycle}} \qquad \text{Perf} = \frac{1}{\text{Time}}$$

- Options to reduce execution time:
  - Executed instructions ↓ (work/instruction ↑)
  - Cycles per instruction (CPI) ↓

Processor Performance

- "Iron Law" of performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructio ns}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instructio n}} \cdot \frac{\text{Time}}{\text{Cycle}} \qquad \text{Perf} = \frac{1}{\text{Time}}$$

- Options to reduce execution time:
  - Executed instructions ↓ (work/instruction ↑)
  - Cycles per instruction (CPI) ↓
  - Cycle time ↓ (frequency ↑)

Processor Performance

- "Iron Law" of performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructio ns}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instructio n}} \cdot \frac{\text{Time}}{\text{Cycle}} \qquad \text{Perf} = \frac{1}{\text{Time}}$$

- Options to reduce execution time:
  - Executed instructions ↓ (work/instruction ↑)
  - Cycles per instruction (CPI) ↓
  - Cycle time ↓ (frequency ↑)

- Today: Simple single-cycle processor, executes one instruction from start to end each clock cycle

Processor Performance

- "Iron Law" of performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructio ns}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instructio n}} \cdot \frac{\text{Time}}{\text{Cycle}} \qquad \text{Perf} = \frac{1}{\text{Time}}$$

- Options to reduce execution time:
  - Executed instructions ↓ (work/instruction ↑)
  - Cycles per instruction (CPI) ↓
  - Cycle time ↓ (frequency ↑)

- Today: Simple single-cycle processor, executes one instruction from start to end each clock cycle
  - CPI = 1, but low frequency
  - Later: Pipelining to increase frequency

Approach: Incremental Featurism

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

Approach: Incremental Featurism

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

Steps:

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

Approach: Incremental Featurism

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

Steps:

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

Component Repertoire:

Registers

Approach: Incremental Featurism

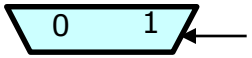
We'll implement datapaths for each instruction class individually, and merge them (using MUXes)

Steps:

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

Component Repertoire:

Registers



Muxes

Approach: Incremental Featurism

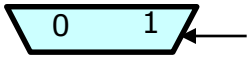
We'll implement datapaths for each instruction class individually, and merge them (using MUXes)

Steps:

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

Component Repertoire:

Registers



Muxes

"Black box" ALU

Approach: Incremental Featurism

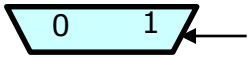
We'll implement datapaths for each instruction class individually, and merge them (using MUXes)

Steps:

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

Component Repertoire:

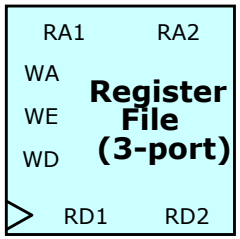
Registers



Muxes

"Black box" ALU

Register File



Approach: Incremental Featurism

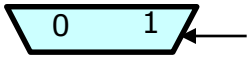
We'll implement datapaths for each instruction class individually, and merge them (using MUXes)

Steps:

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

Component Repertoire:

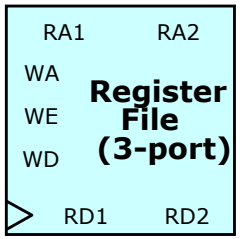
Registers

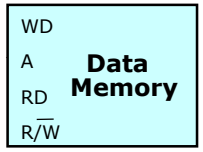


Muxes

"Black box" ALU

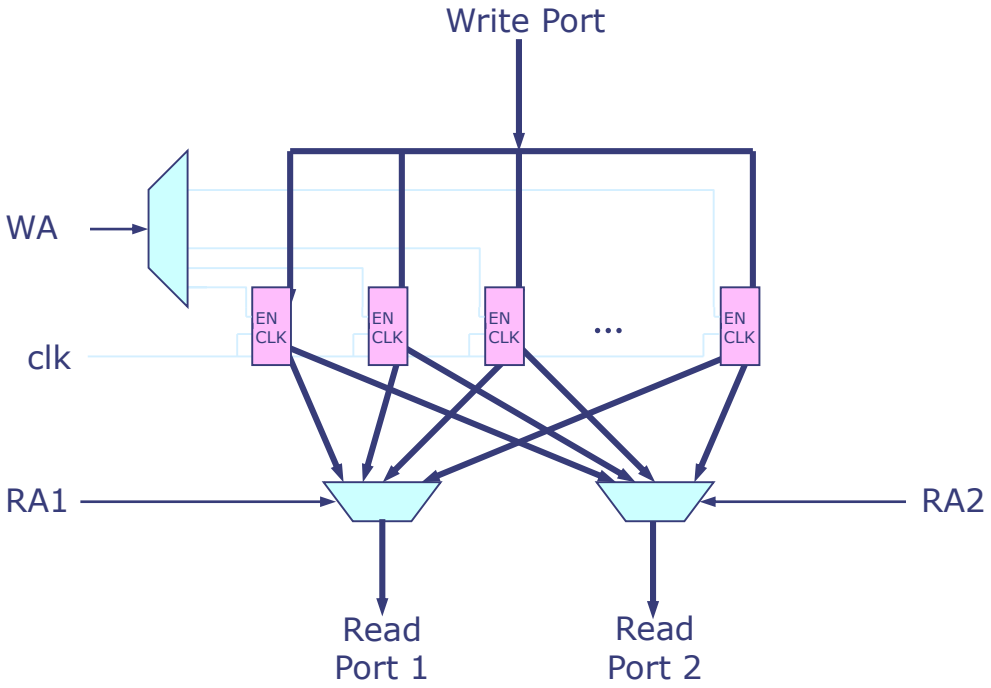
Register File



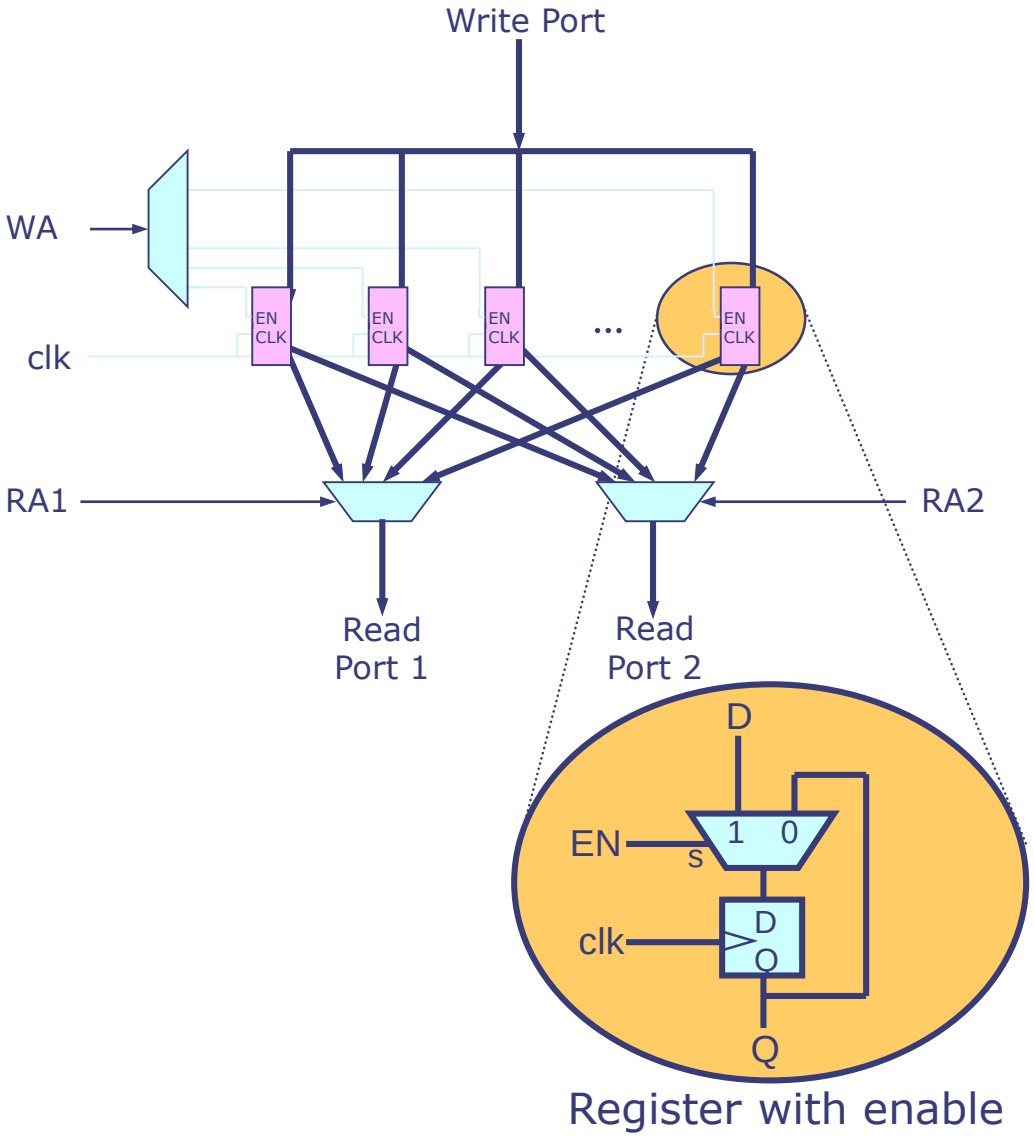


Memories

Multi-Ported Register File

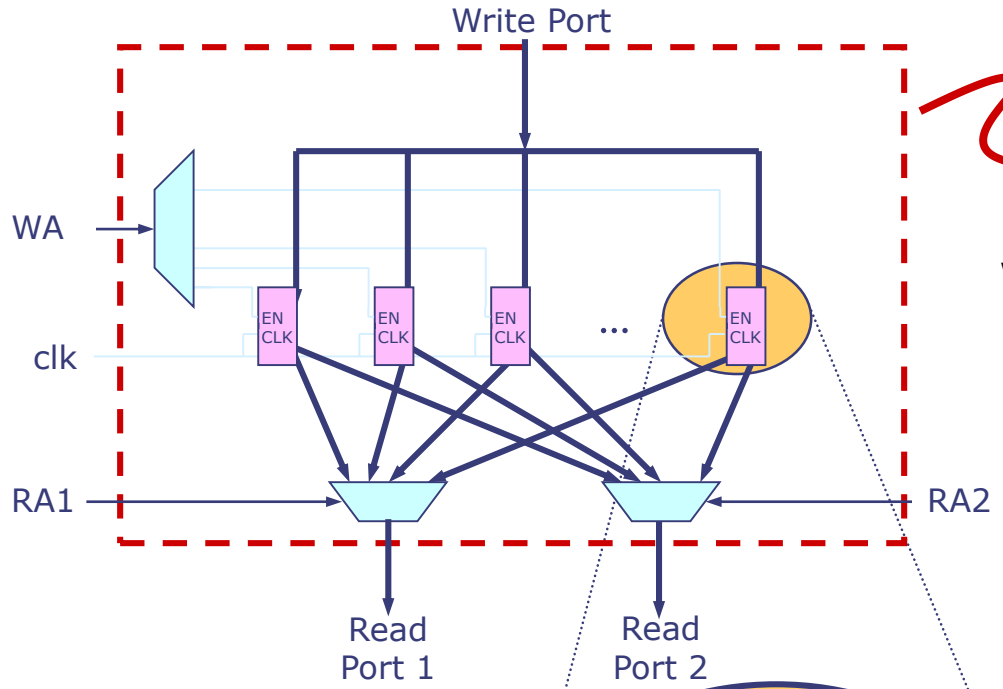


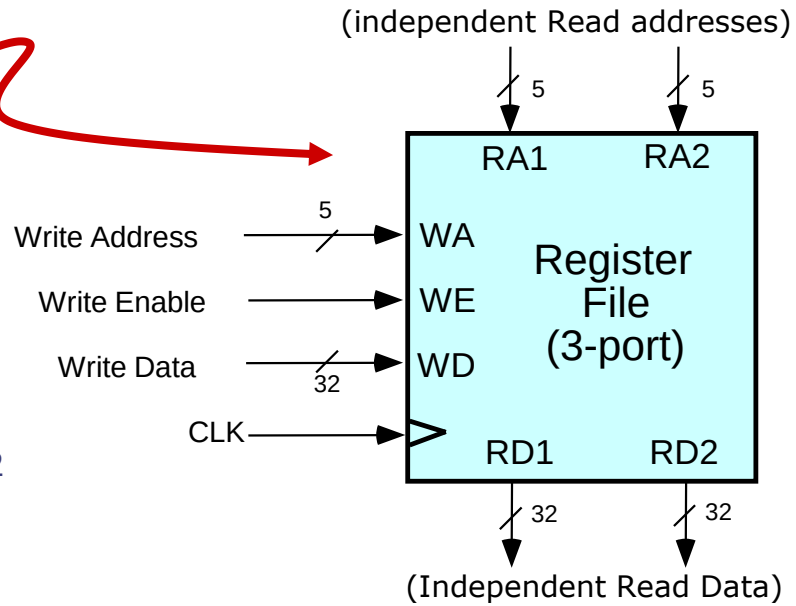
Multi-Ported Register File

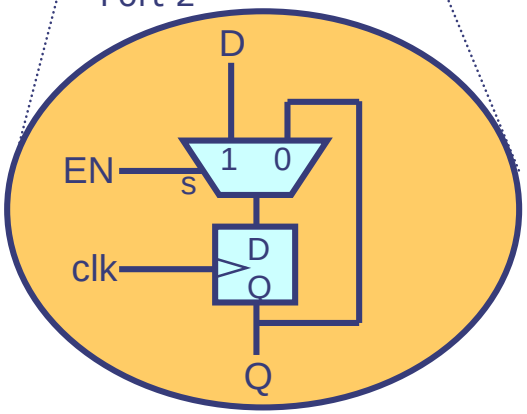


Register with enable

Multi-Ported Register File





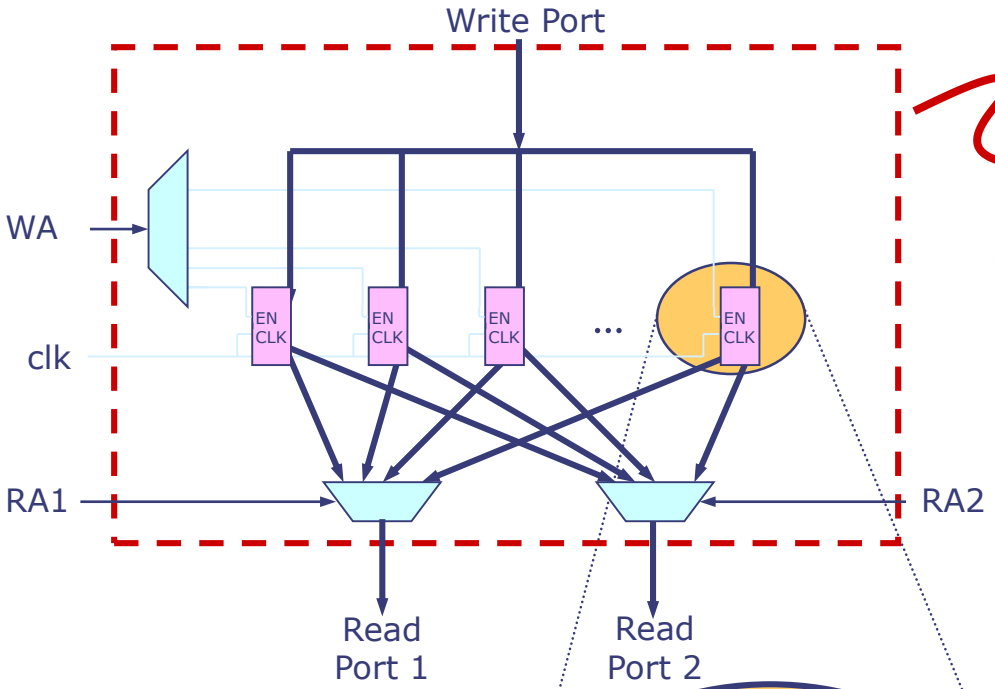


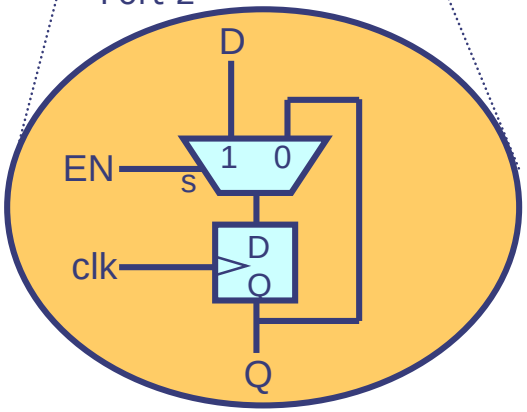
Register with enable

2 combinational READ ports*,
1 clocked WRITE port

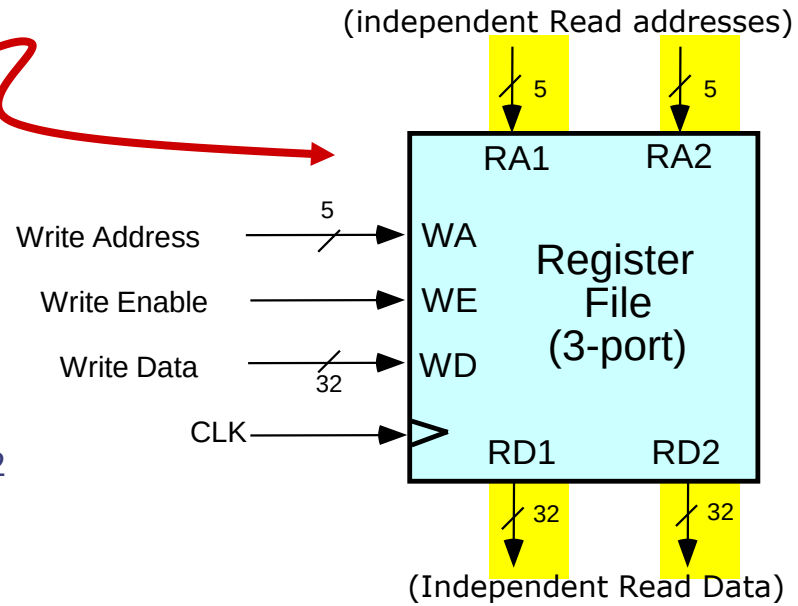
*internal logic ensures Reg[0] reads as 0

Multi-Ported Register File





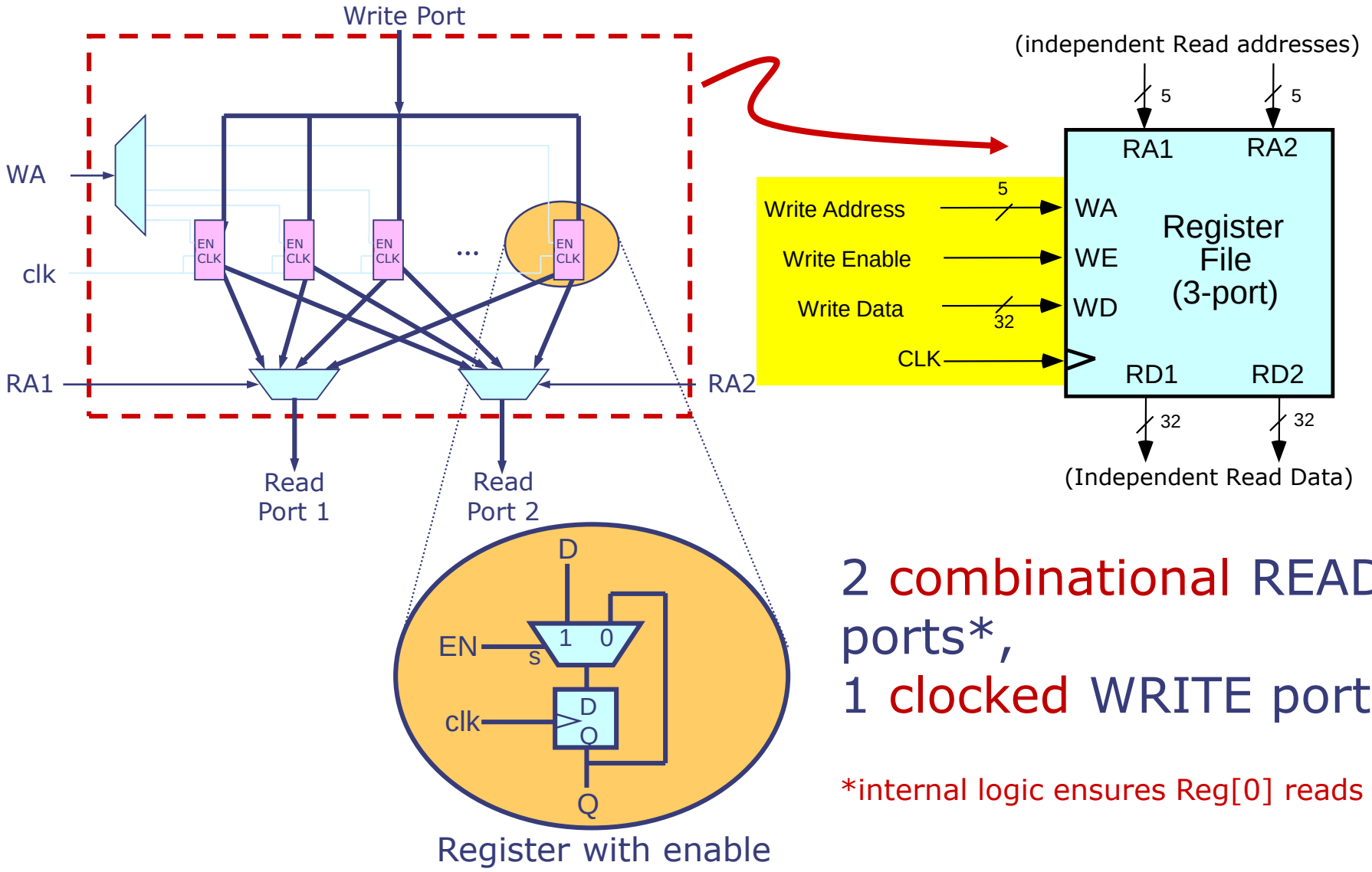
Register with enable



2 combinational READ ports*,
1 clocked WRITE port

*internal logic ensures Reg[0] reads as 0

Multi-Ported Register File

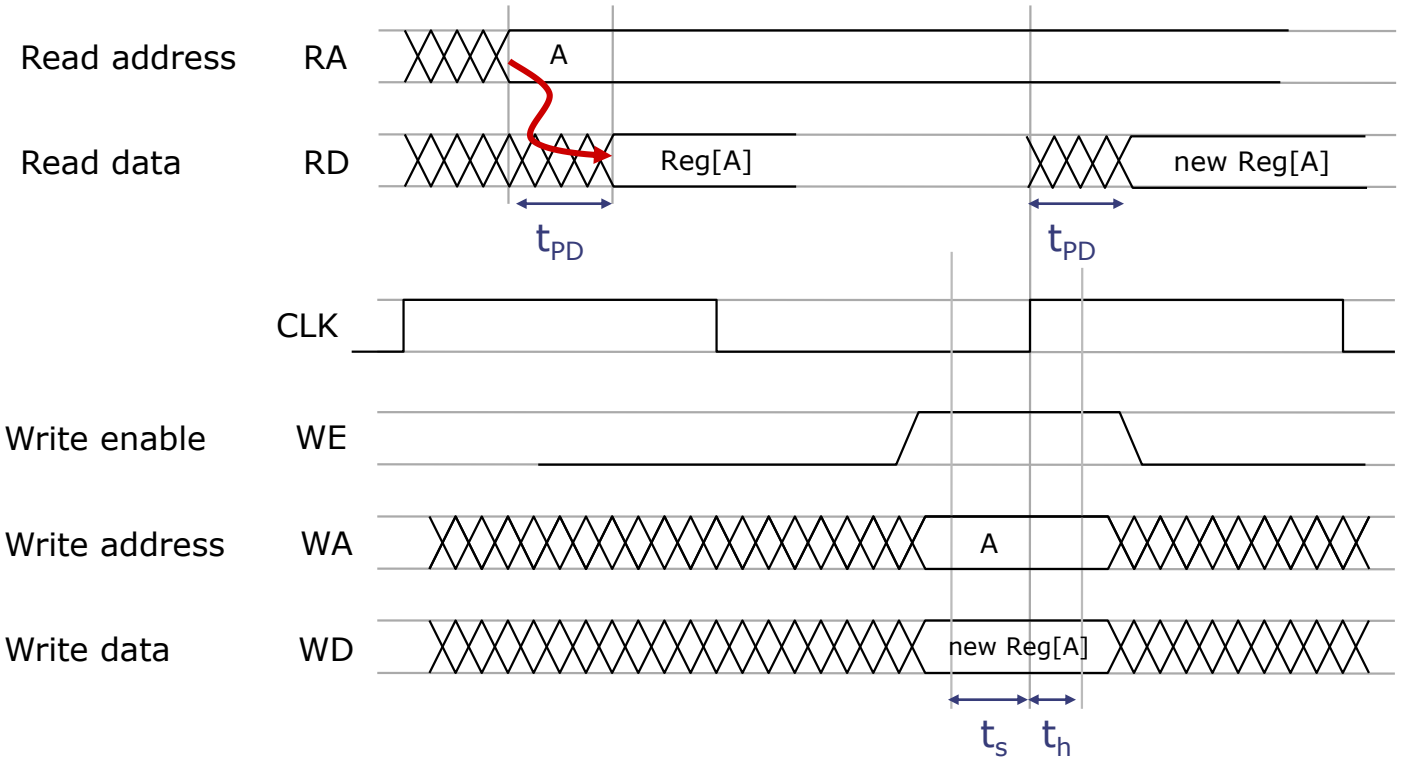


2 combinational READ ports*,
1 clocked WRITE port

*internal logic ensures Reg[0] reads as 0

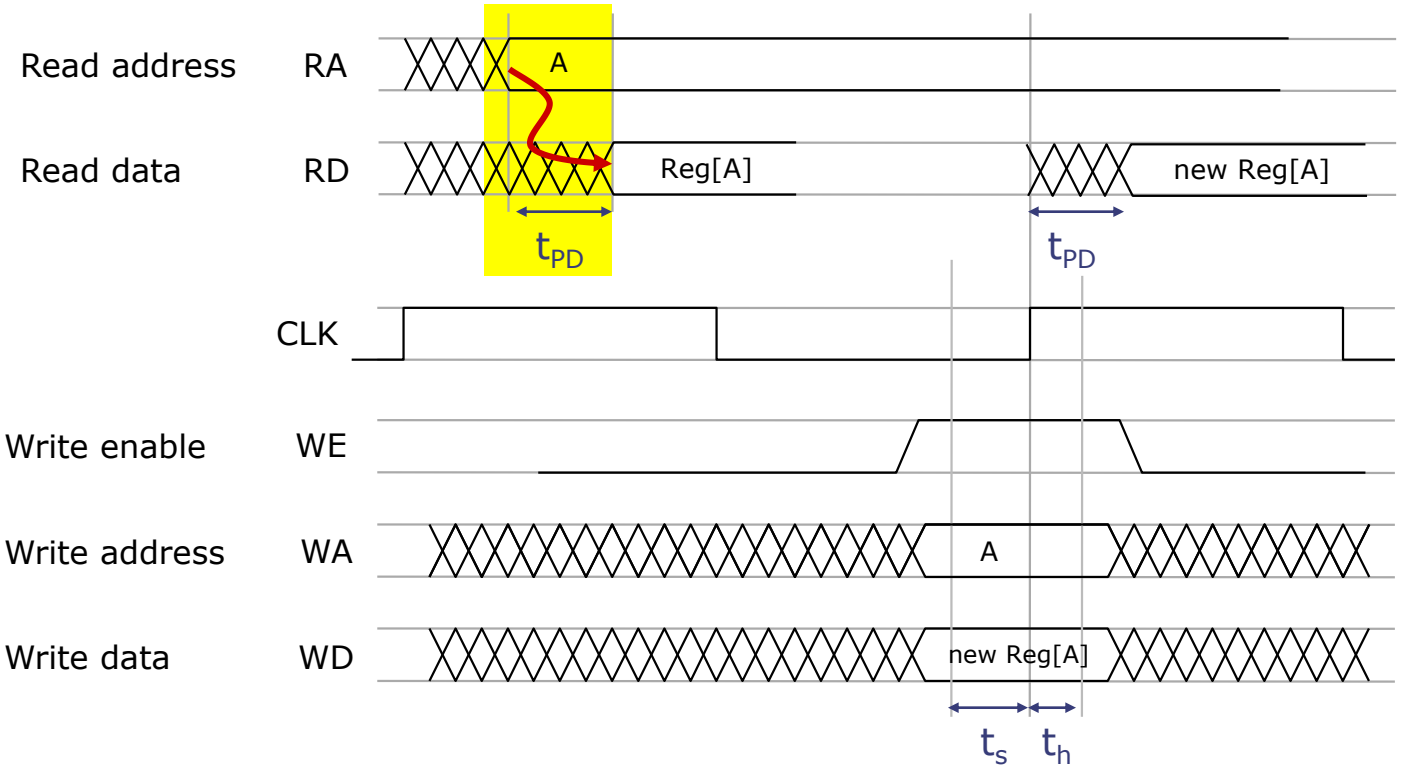
Register File Timing

2 combinational READ ports, 1 clocked WRITE port



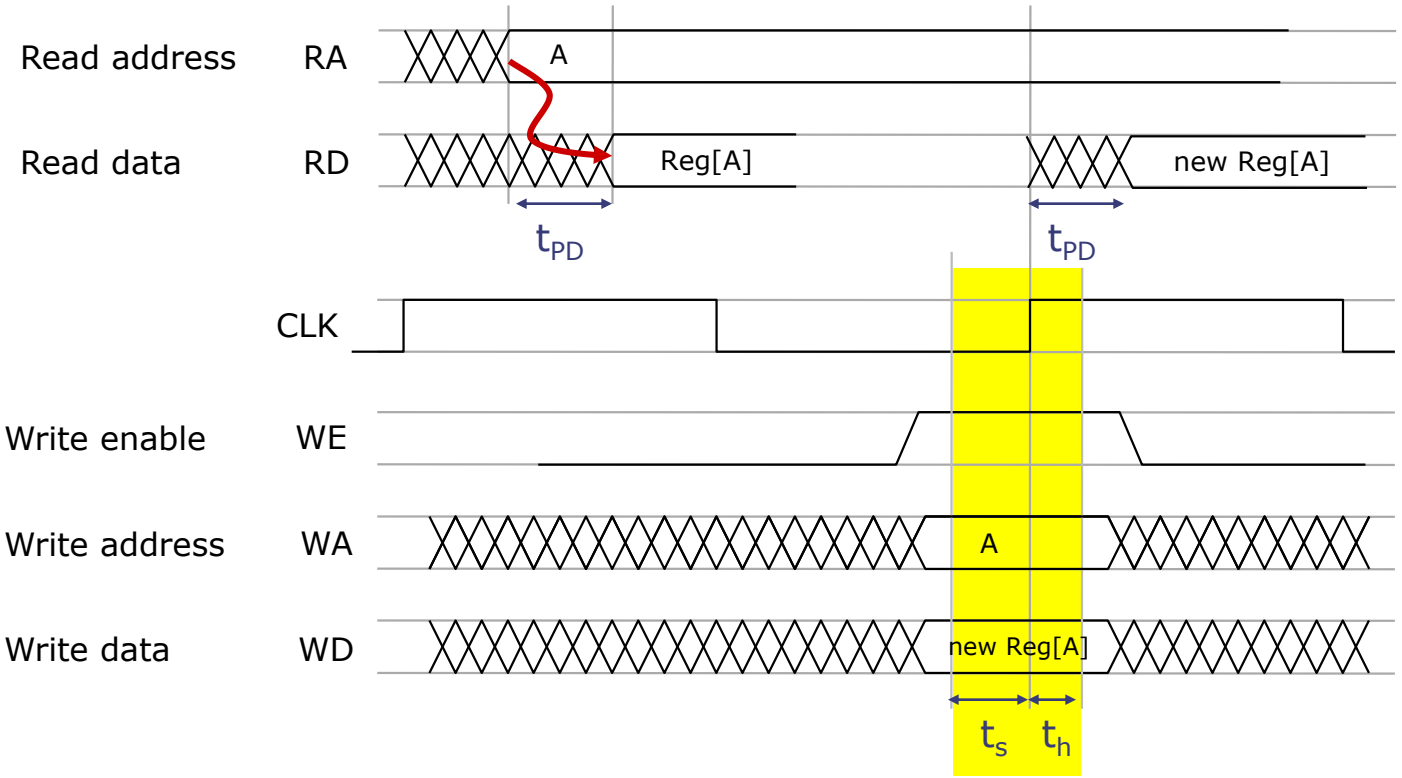
Register File Timing

2 combinational READ ports, 1 clocked WRITE port



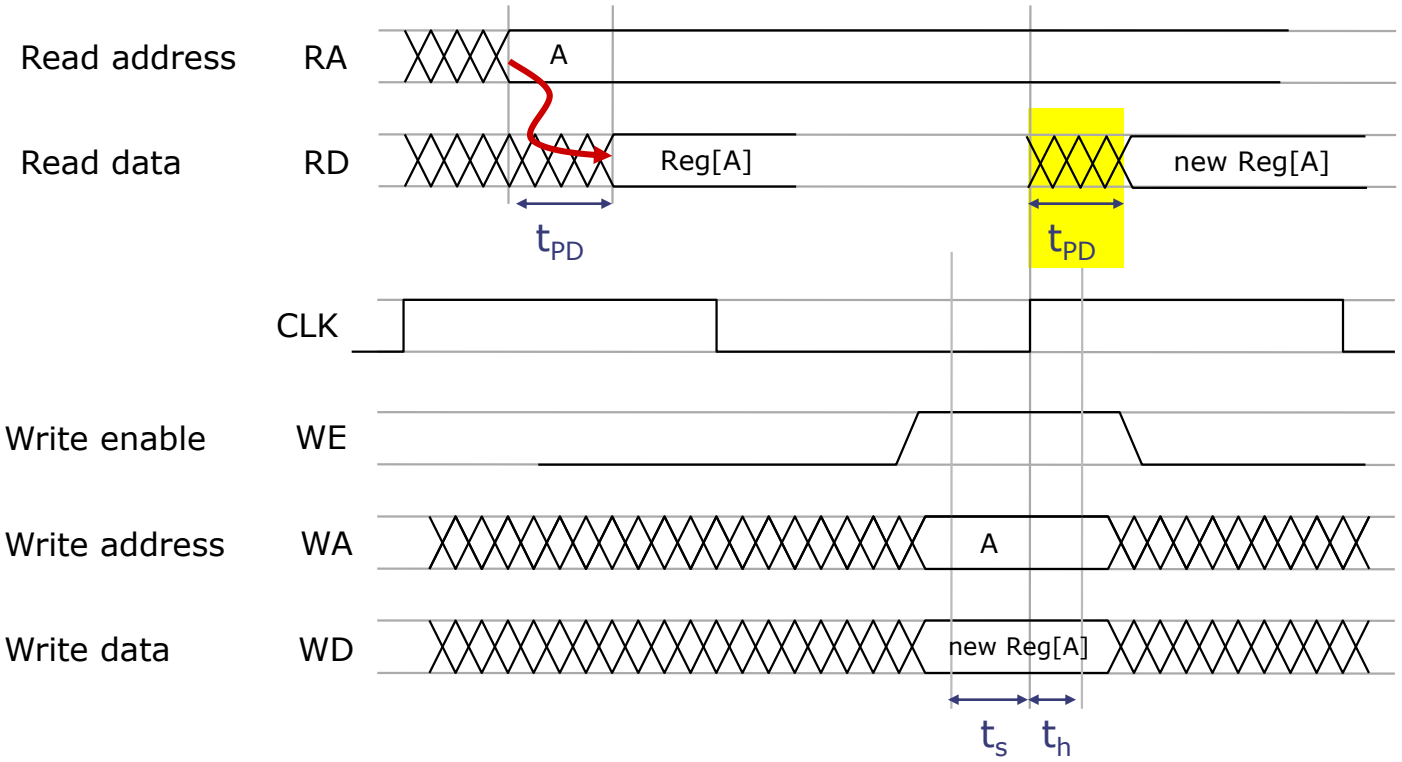
Register File Timing

2 combinational READ ports, 1 clocked WRITE port



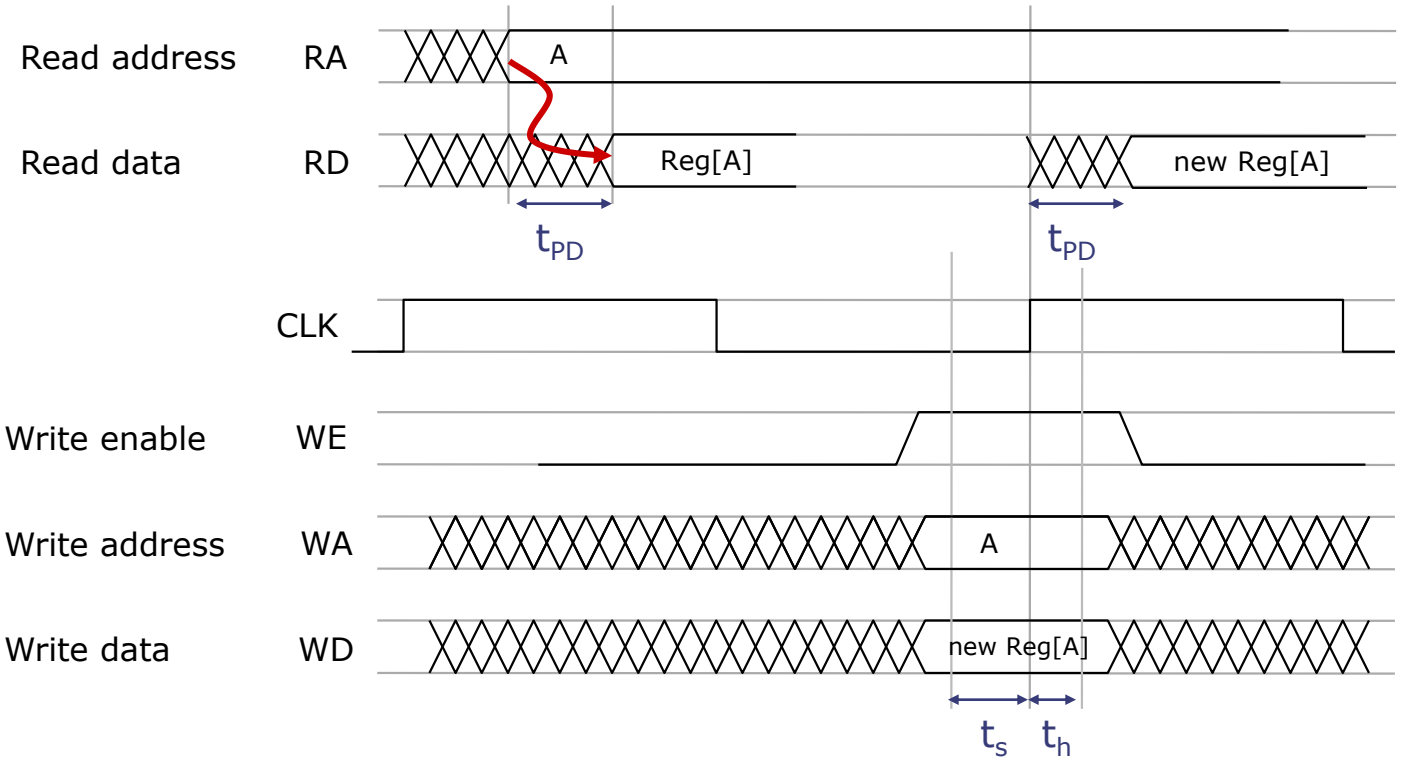
Register File Timing

2 combinational READ ports, 1 clocked WRITE port



Register File Timing

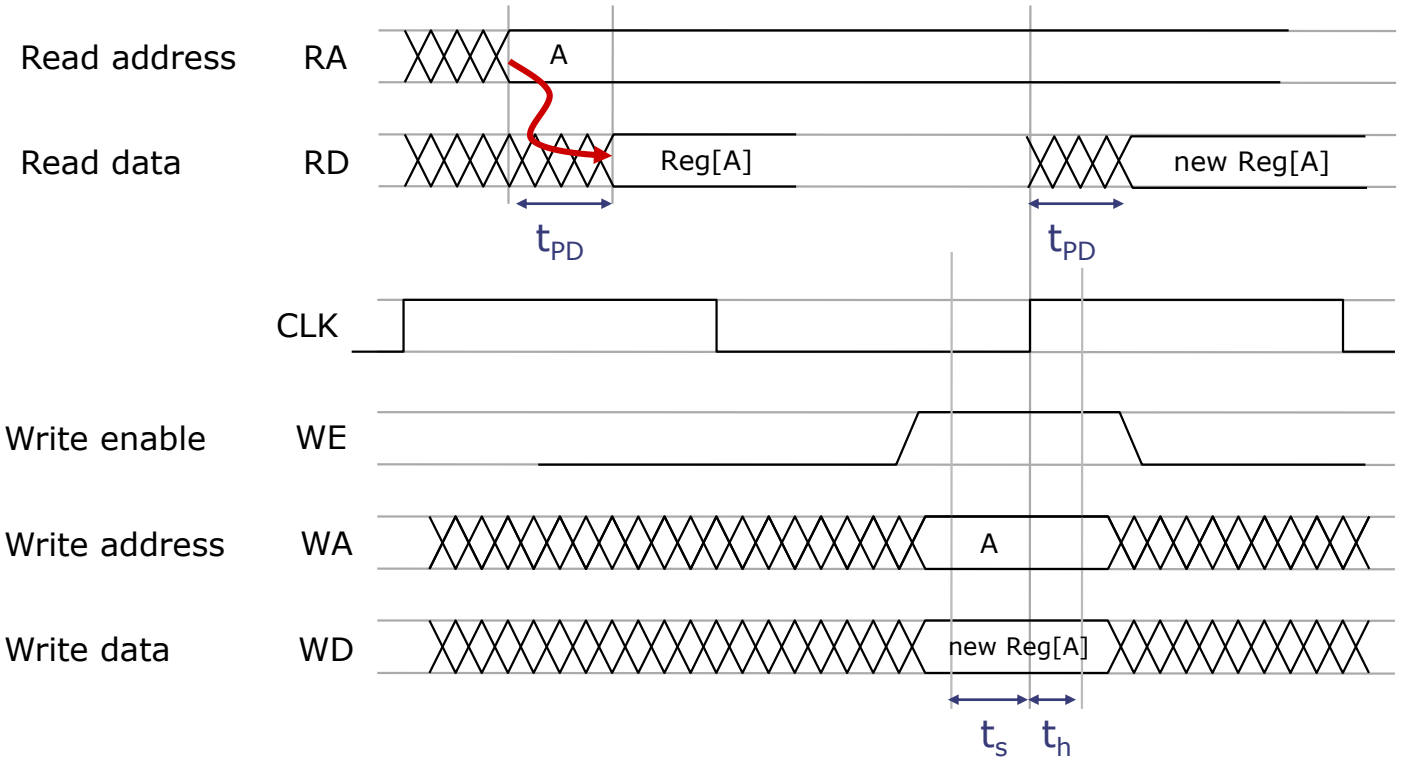
2 combinational READ ports, 1 clocked WRITE port



What if WA=RA1?

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

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

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

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 it's not realistic

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 it's not realistic

- Next week we will learn about the implementation of these memories

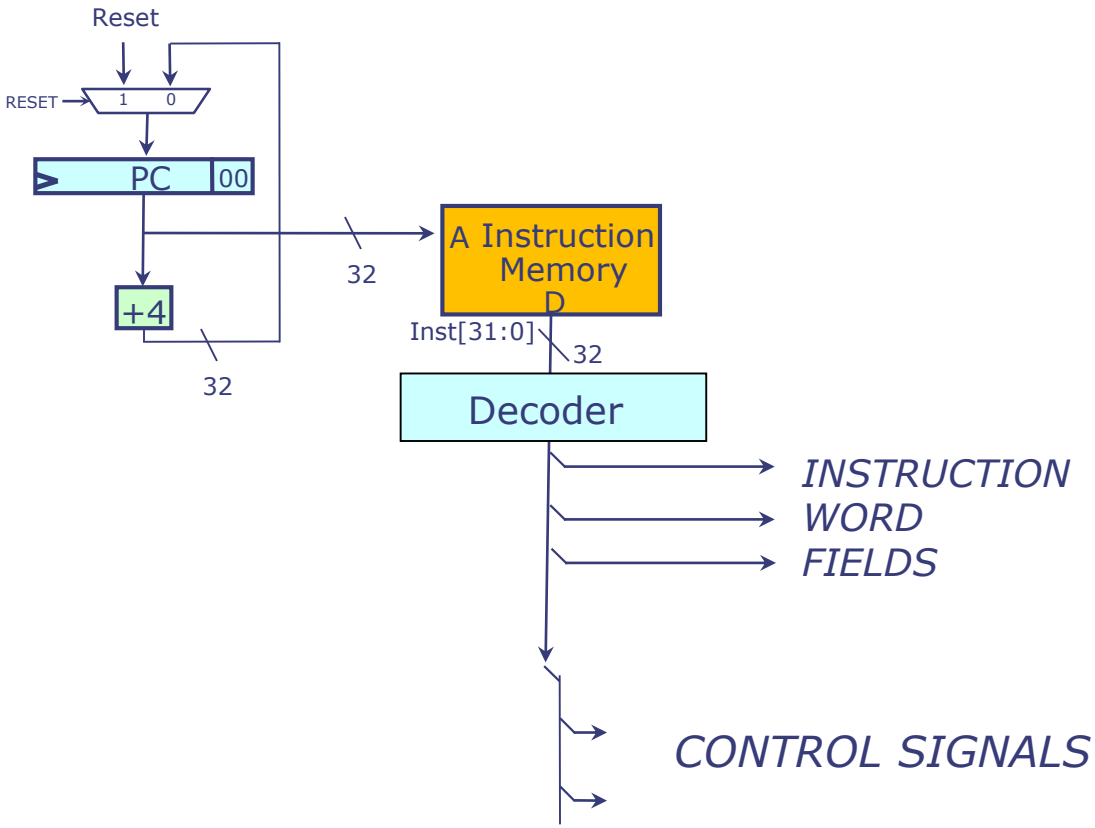
Memory Timing

- 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 it's not realistic

- Next week we will learn about the implementation of these memories
- In lab 7 and the design project, we will use realistic memories for our processor

Instruction Fetch/Decode

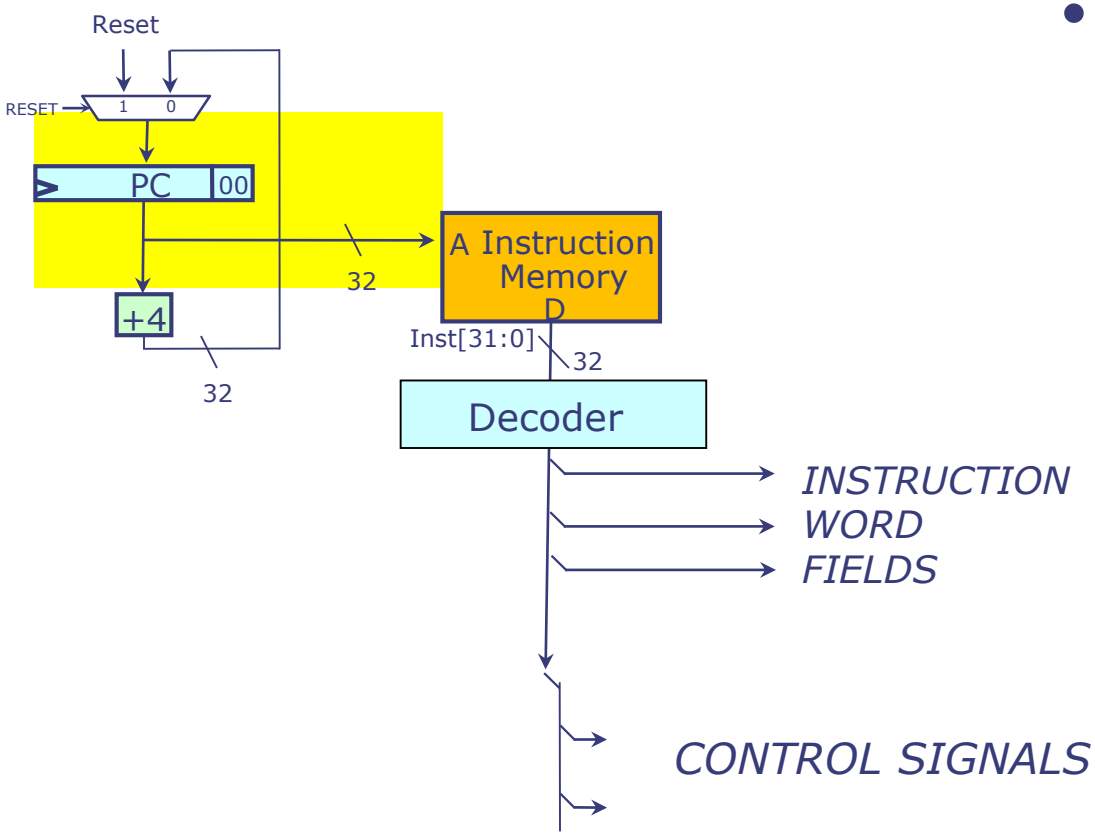
Use Program Counter (PC) to fetch the next instruction:



Instruction Fetch/Decode

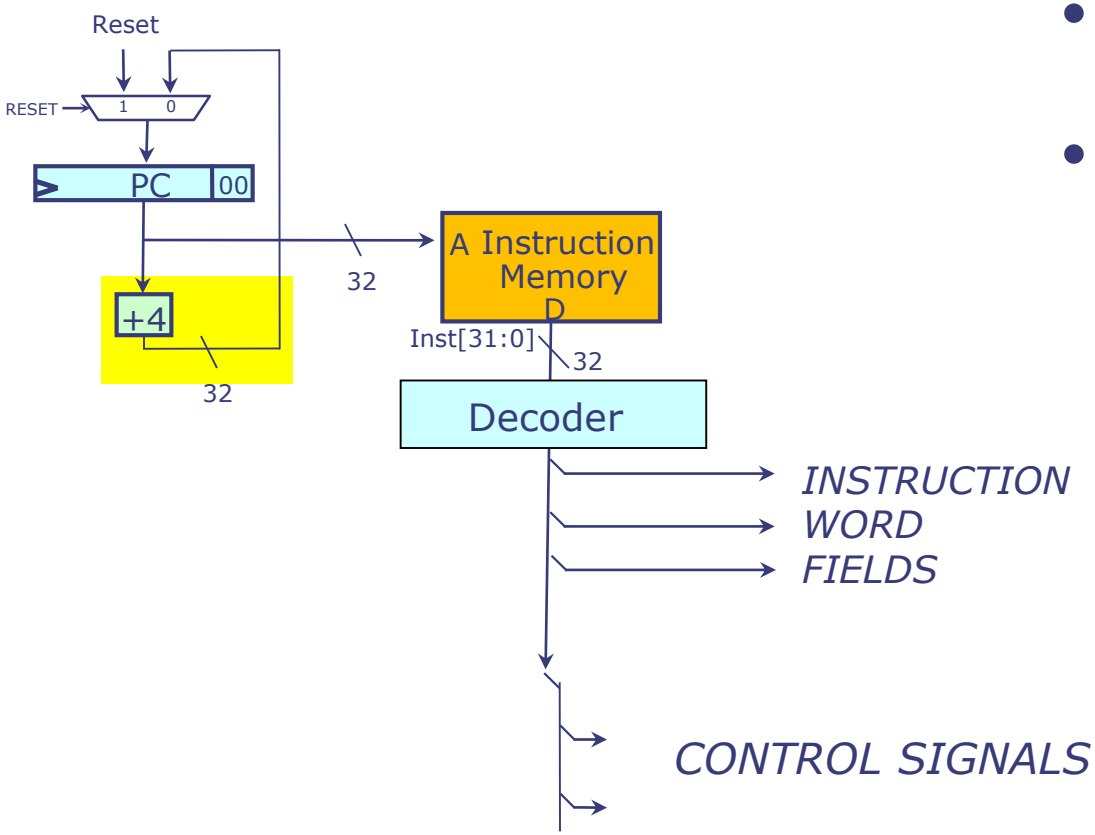
Use Program Counter (PC) to fetch the next instruction:

- Use PC as memory address



Instruction Fetch/Decode

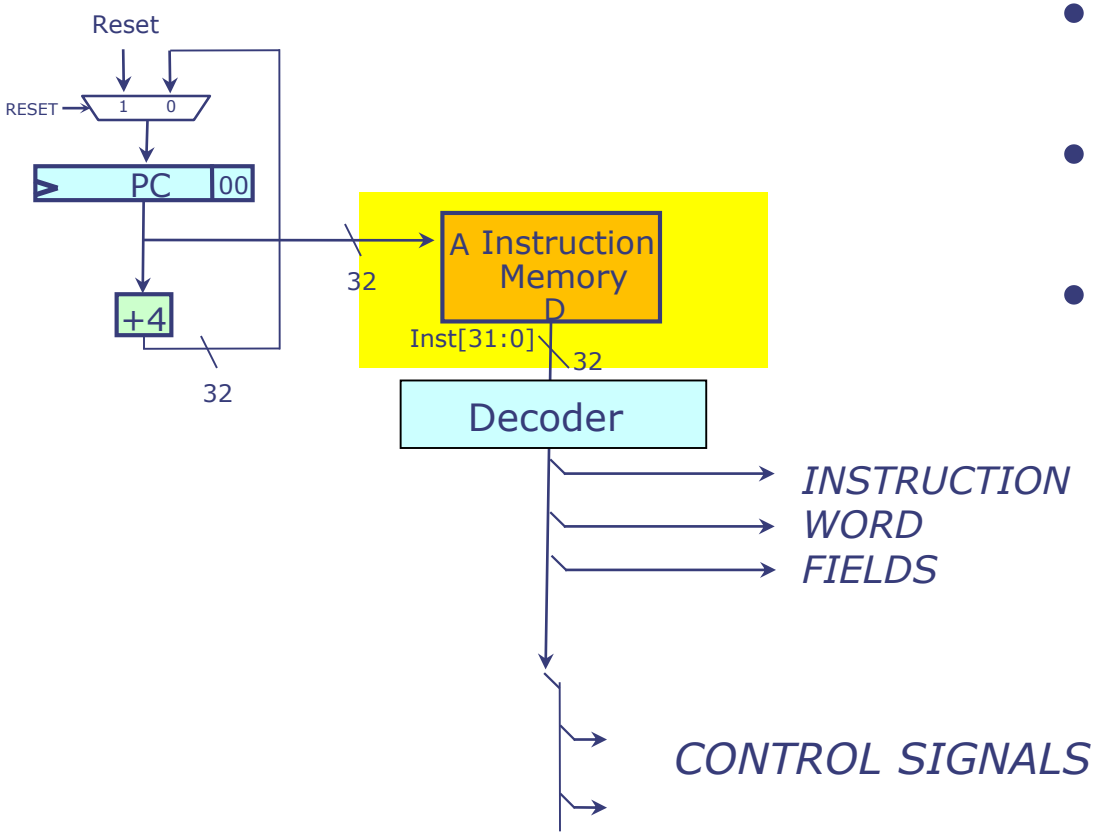
Use Program Counter (PC) to fetch the next instruction:



- Use PC as memory address
- Add 4 to PC, load new value at end of cycle

Instruction Fetch/Decode

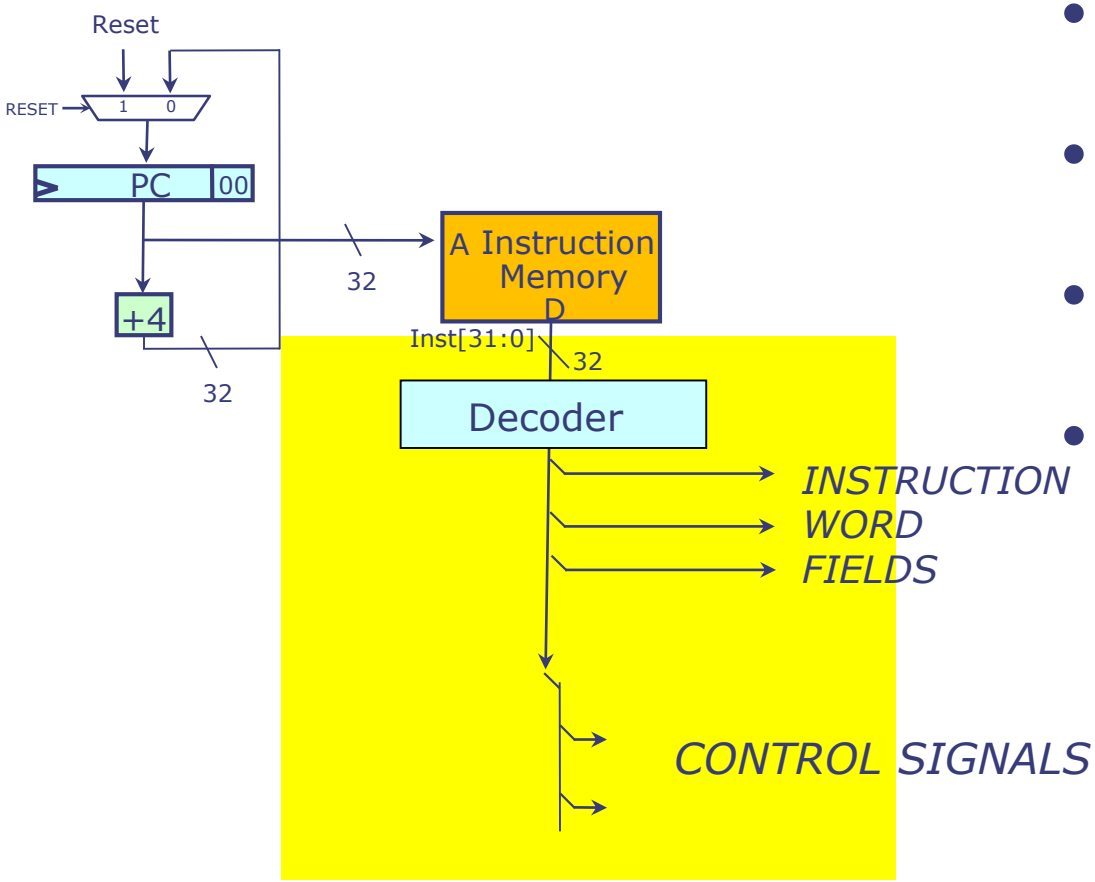
Use Program Counter (PC) to fetch the next instruction:



- Use PC as memory address
- Add 4 to PC, load new value at end of cycle
- Fetch instruction from memory

Instruction Fetch/Decode

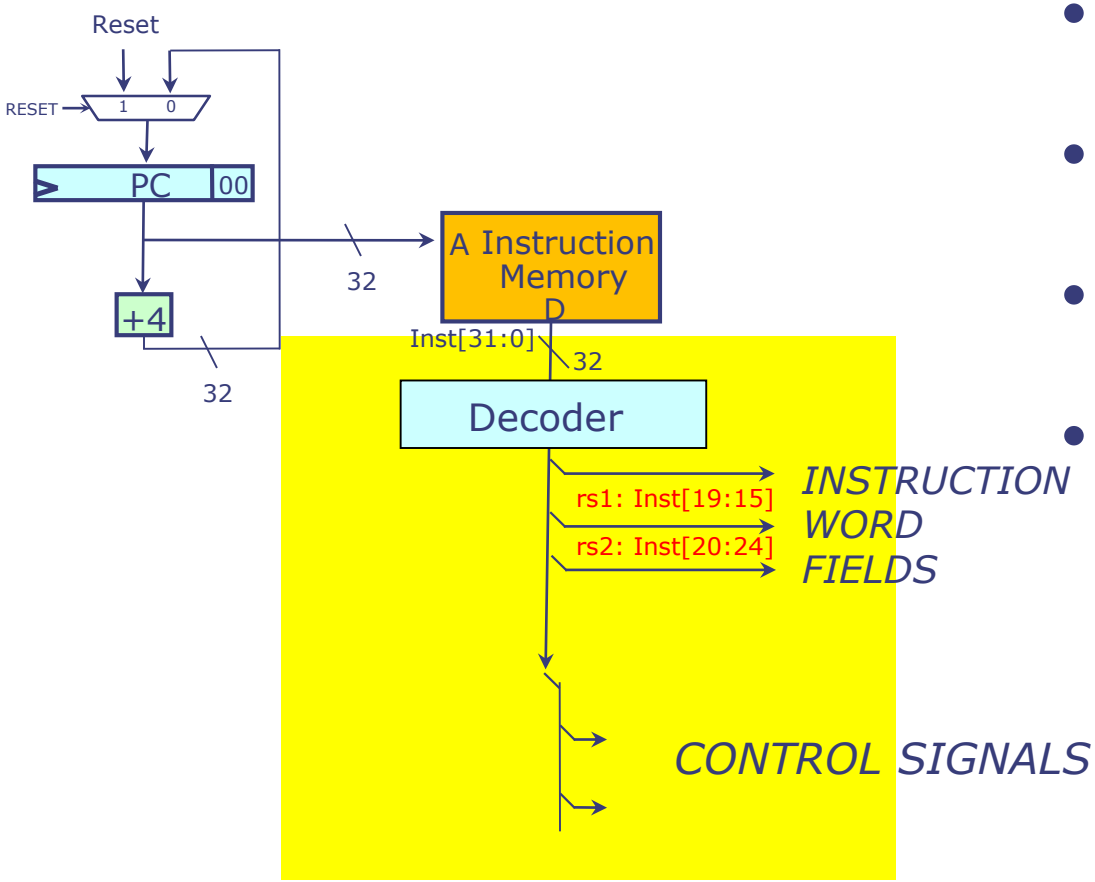
Use Program Counter (PC) to fetch the next instruction:



- Use PC as memory address
- Add 4 to PC, load new value at end of cycle
- Fetch instruction from memory
- Decode instruction:

Instruction Fetch/Decode

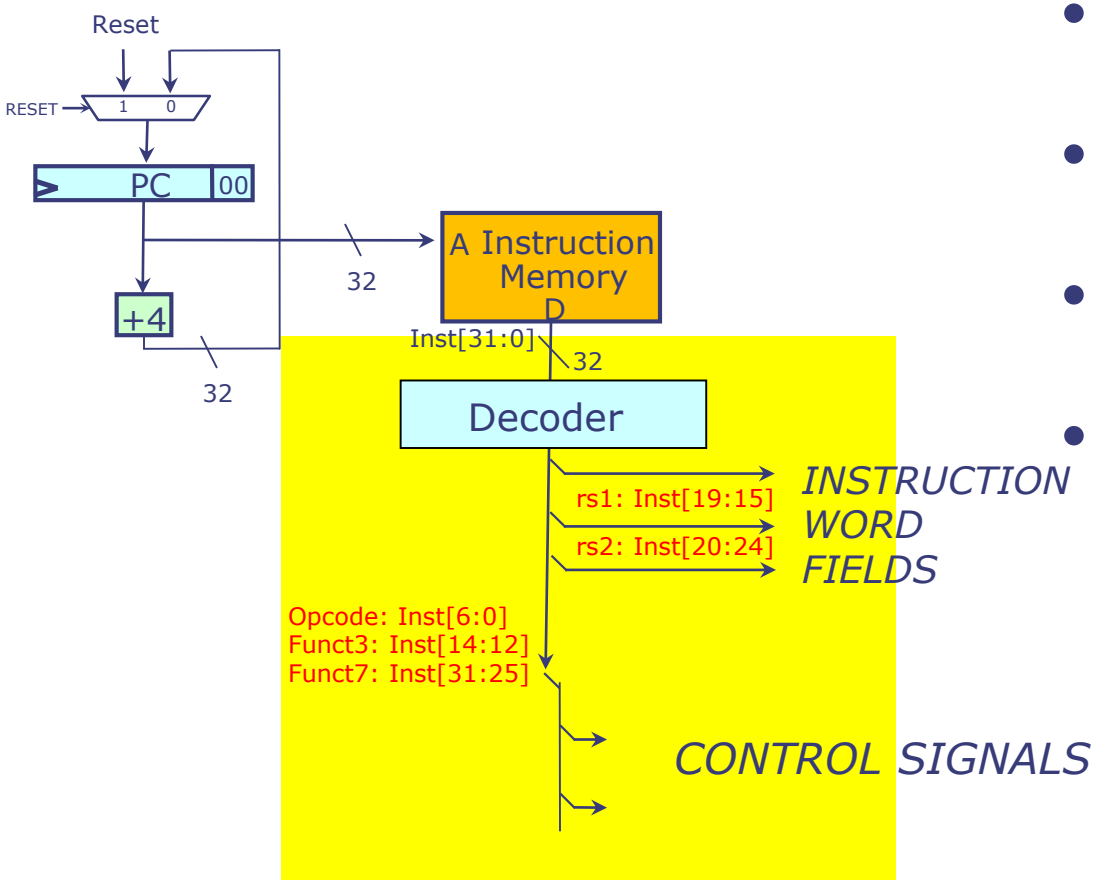
Use Program Counter (PC) to fetch the next instruction:



- 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 indexes, immediate values)

Instruction Fetch/Decode

Use Program Counter (PC) to fetch the next instruction:



- 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 indexes, immediate values)
  - Use opcode, funct3, and funct7 bits to generate control signals

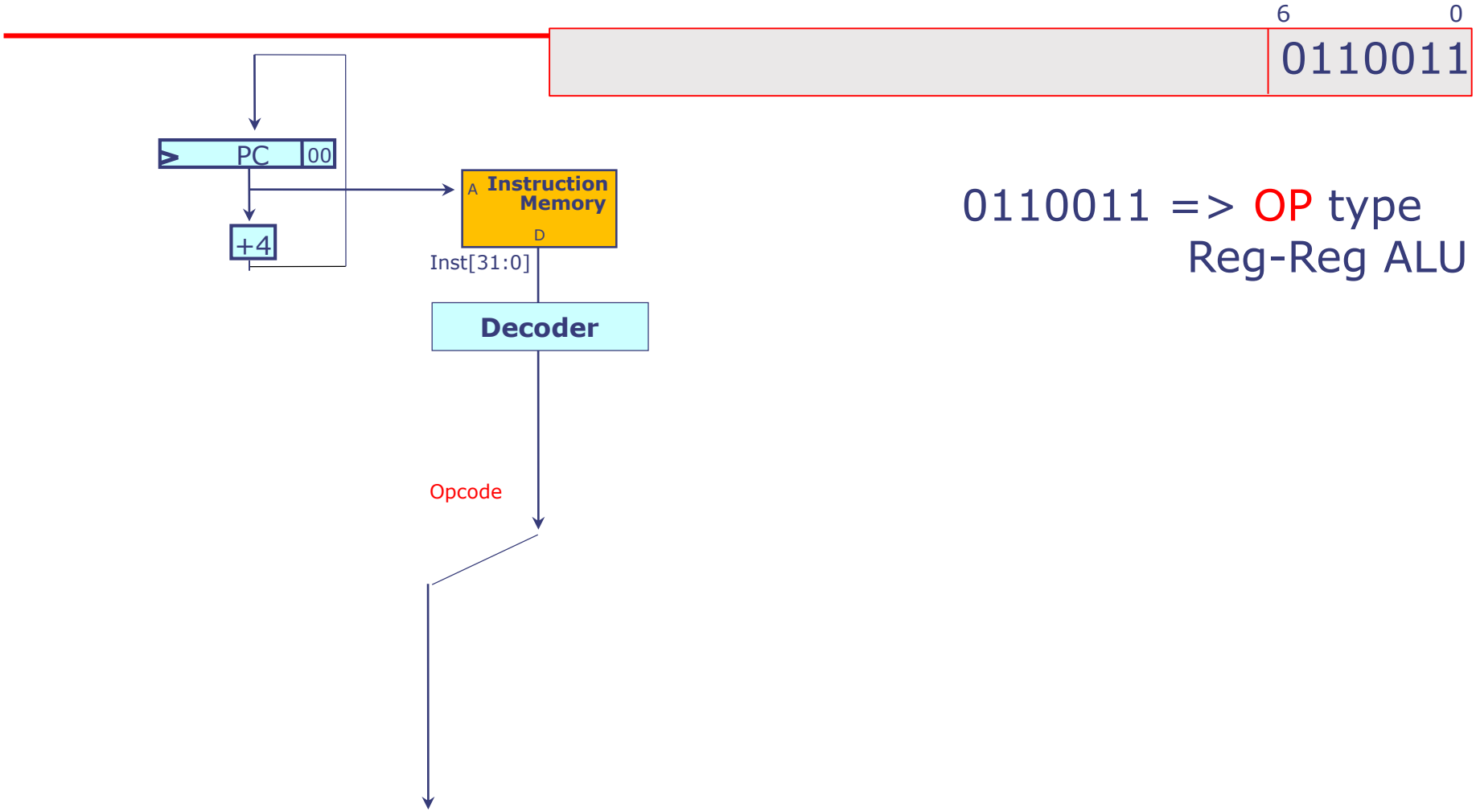
ALU Instructions

Differ only in the ALU op to be performed

Instruction	Description	Execution
ADD rd, rs1, rs2	Add	reg[rd] <= reg[rs1] + reg[rs2]
SUB rd, rs1, rs2	Sub	reg[rd] <= reg[rs1] – reg[rs2]
SLL rd, rs1, rs2	Shift Left Logical	reg[rd] <= reg[rs1] << reg[rs2]
SLT rd, rs1, rs2	Set if < (Signed)	reg[rd] <= (reg[rs1] $<_s$ reg[rs2]) ? 1 : 0
SLTU rd, rs1, rs2	Set if < (Unsigned)	reg[rd] <= (reg[rs1] $<_u$ reg[rs2]) ? 1 : 0
XOR rd, rs1, rs2	Xor	reg[rd] <= reg[rs1] ^ reg[rs2]
SRL rd, rs1, rs2	Shift Right Logical	reg[rd] <= reg[rs1] $>>_u$ reg[rs2]
SRA rd, rs1, rs2	Shift Right Arithmetic	reg[rd] <= reg[rs1] $>>_s$ reg[rs2]
OR rd, rs1, rs2	Or	reg[rd] <= reg[rs1] | reg[rs2]
AND rd, rs1, rs2	And	reg[rd] <= reg[rs1] & reg[rs2]

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

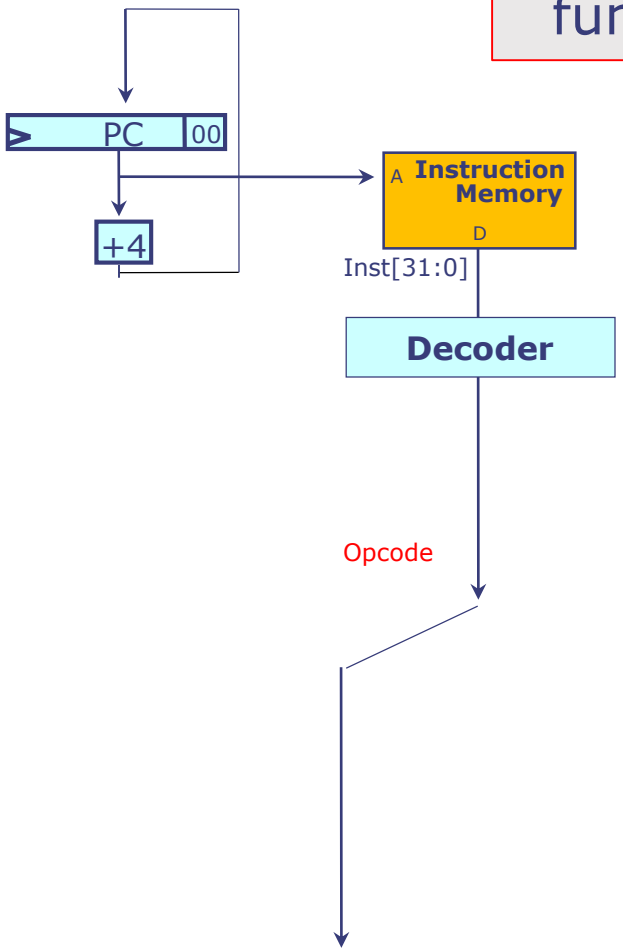
Register-Register ALU Datapath



0110011 => OP type
Reg-Reg ALU

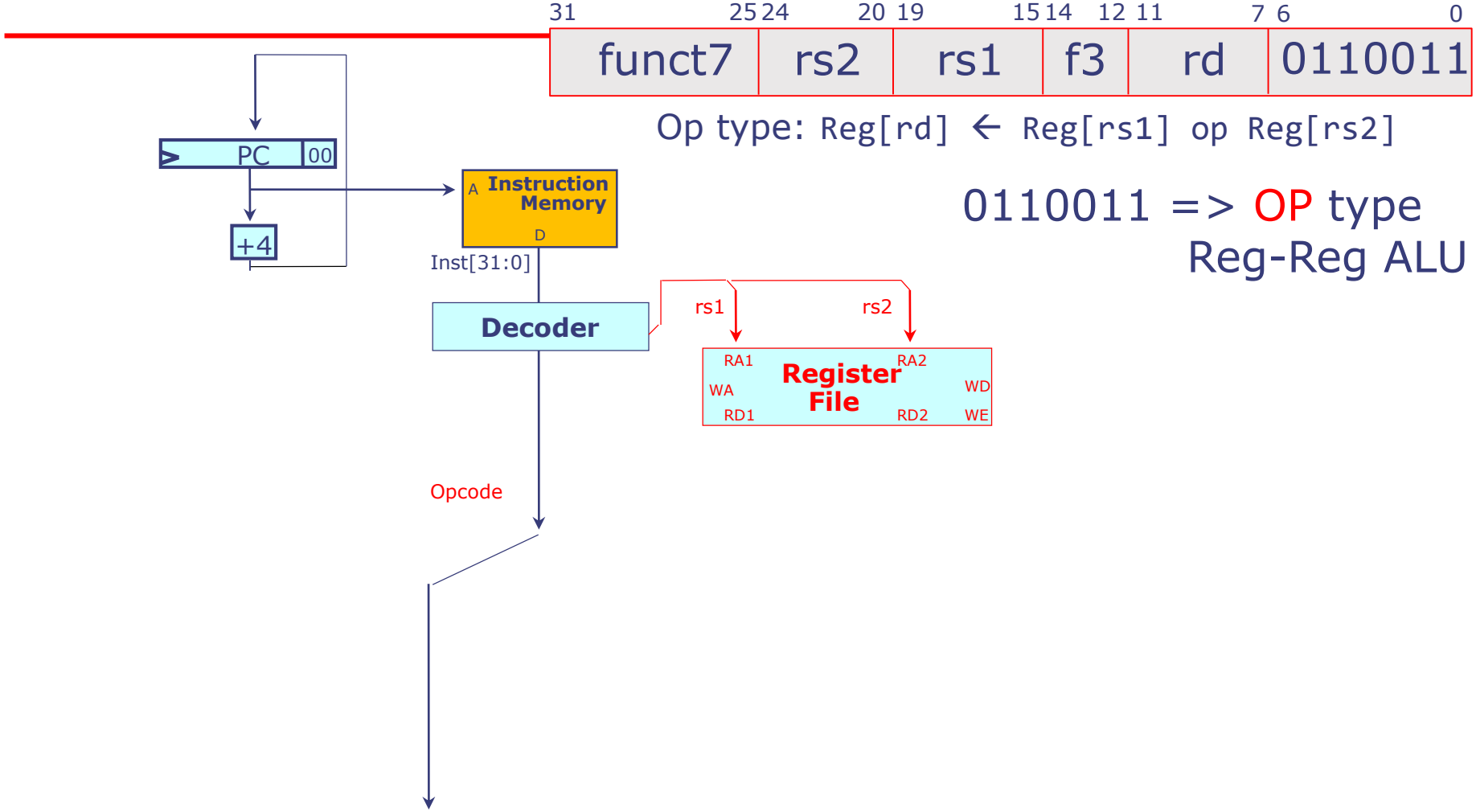
Register-Register ALU Datapath

Op type: `Reg[rd] ← Reg[rs1] op Reg[rs2]`

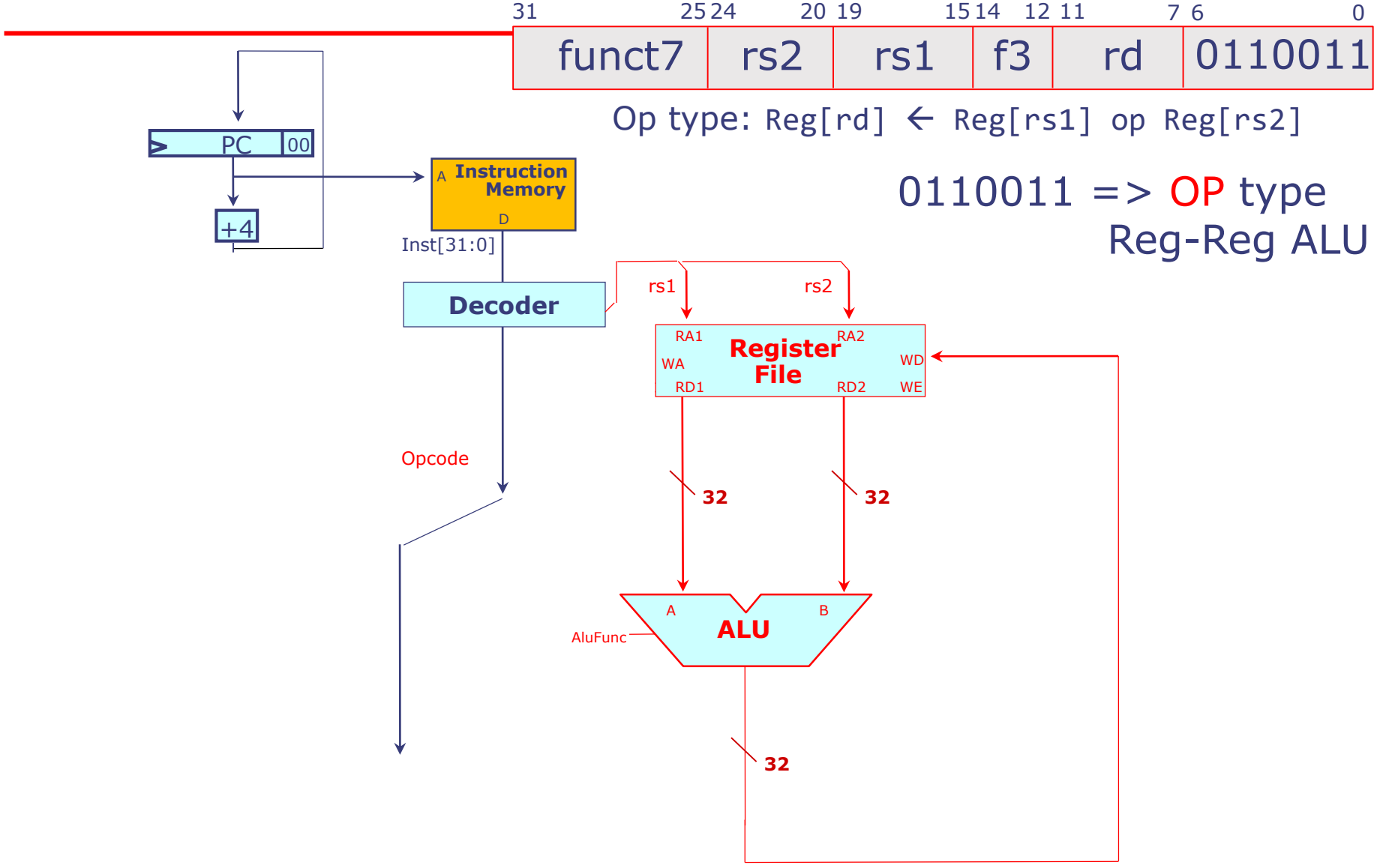


0110011 => OP type
Reg-Reg ALU

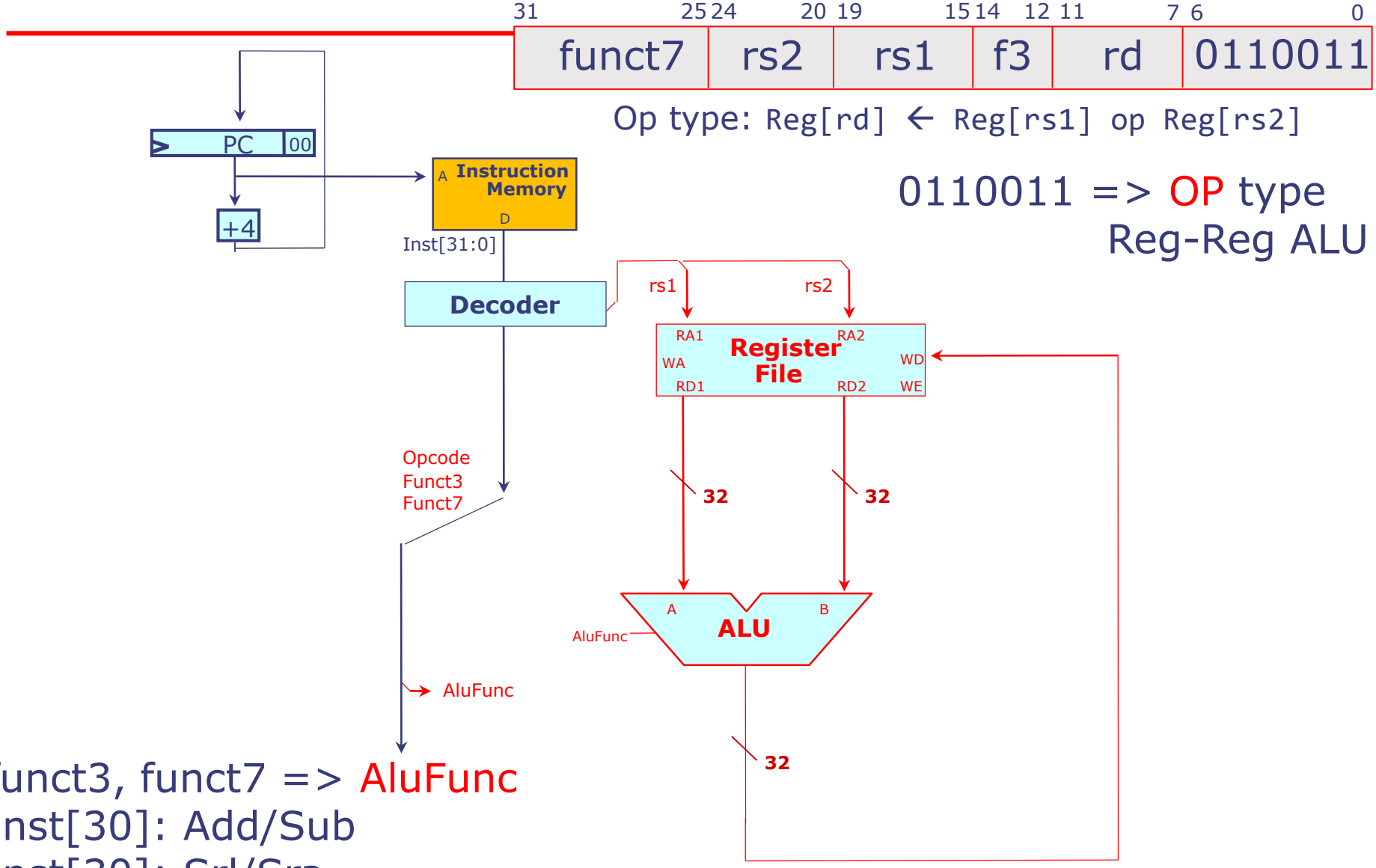
Register-Register ALU Datapath



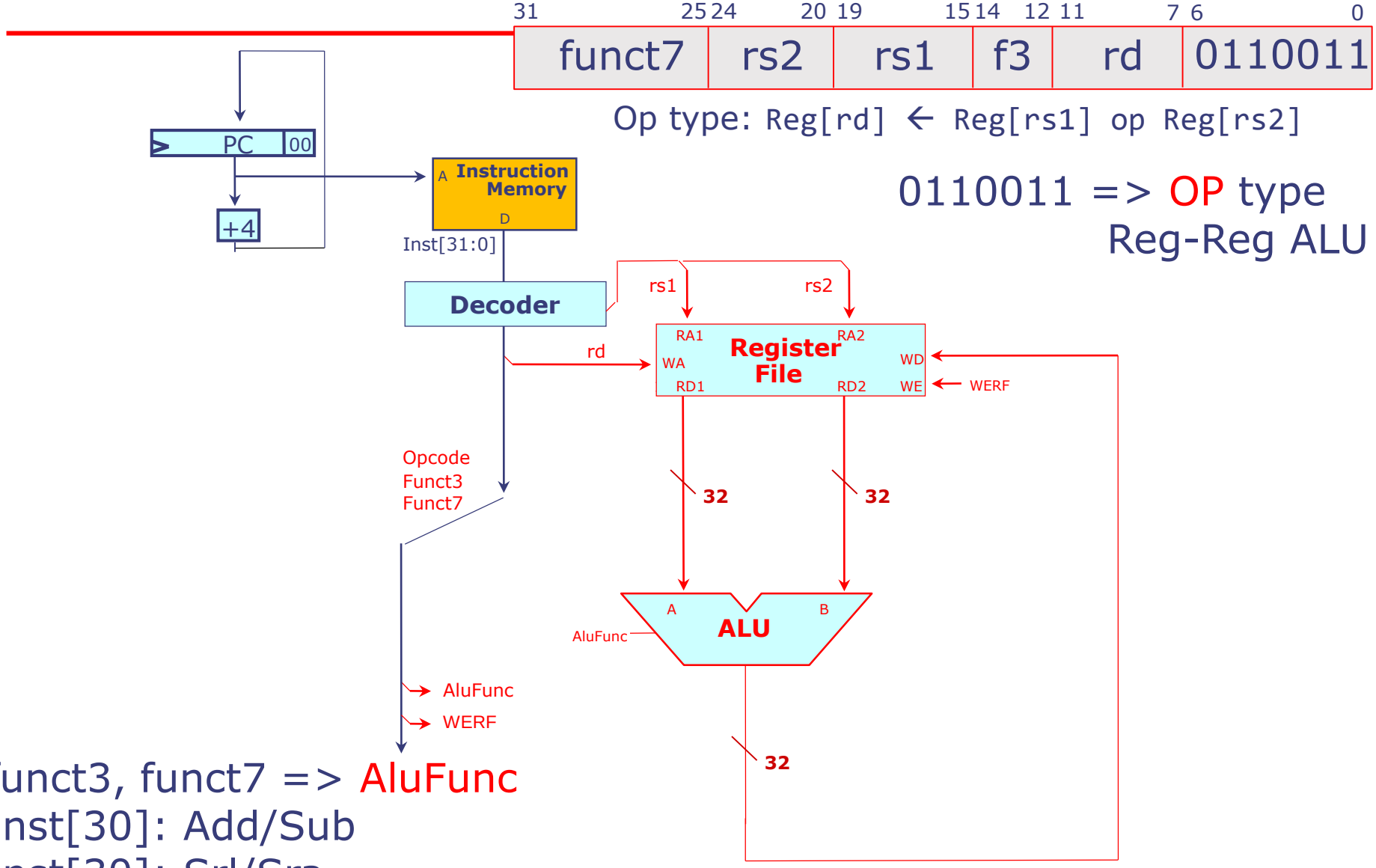
Register-Register ALU Datapath



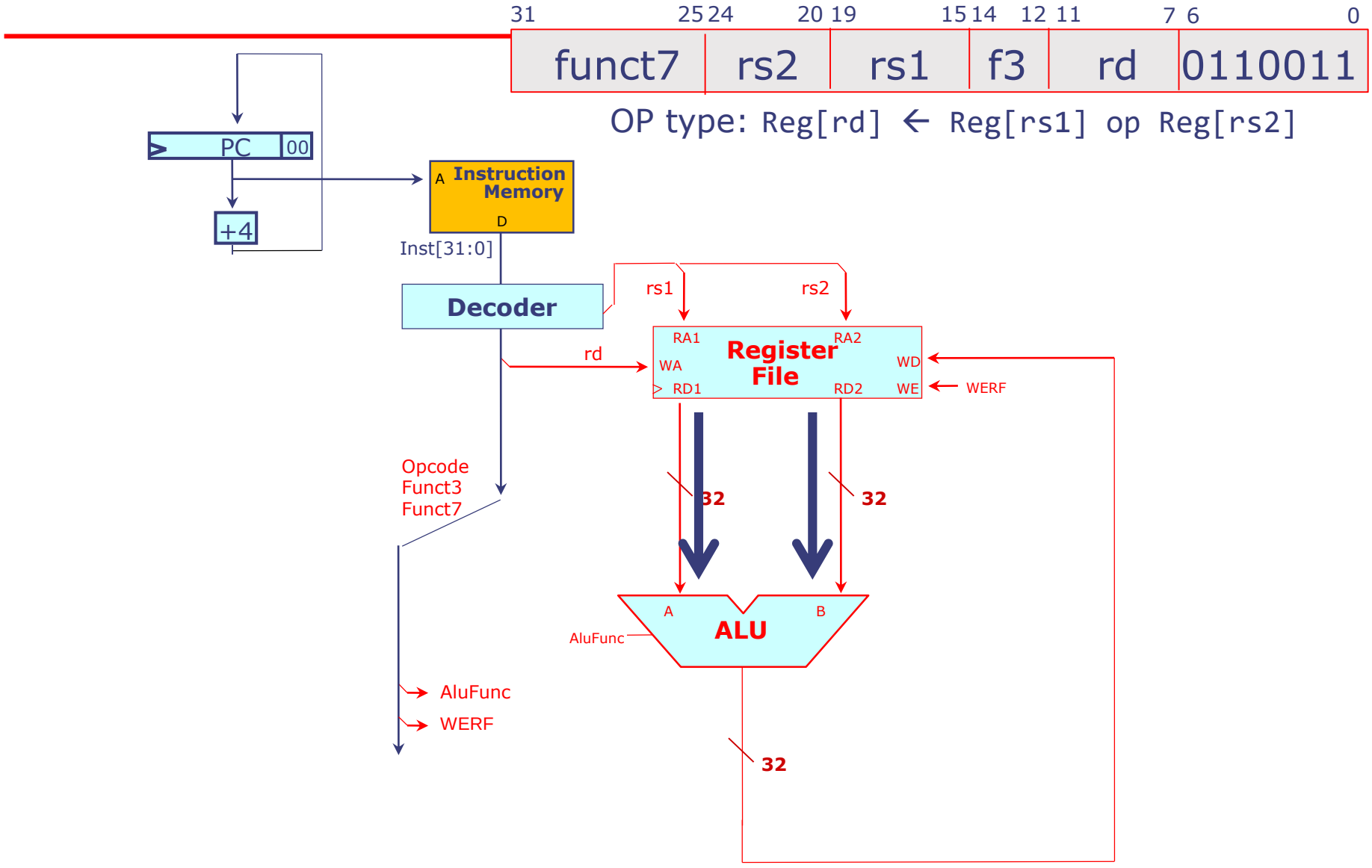
Register-Register ALU Datapath



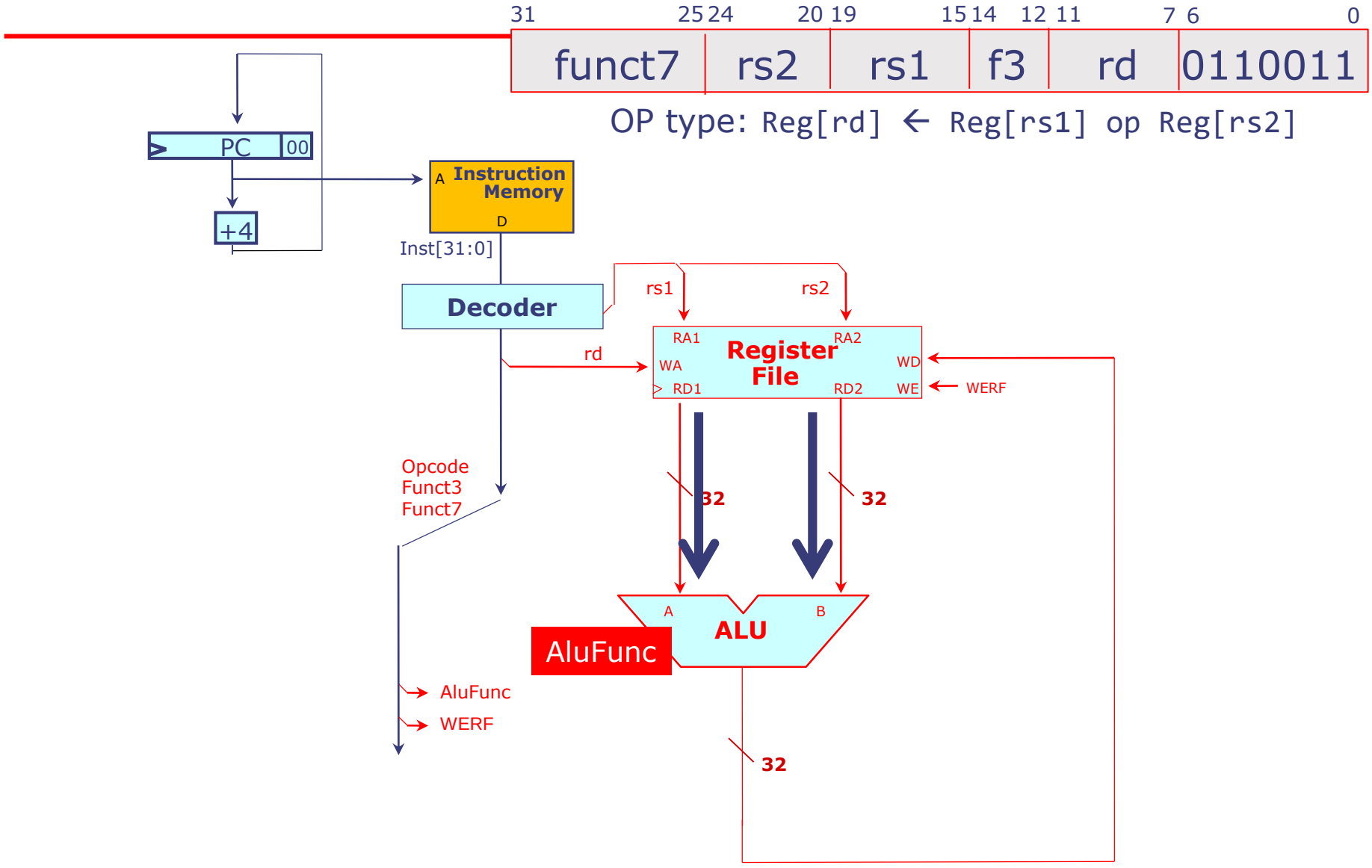
Register-Register ALU Datapath



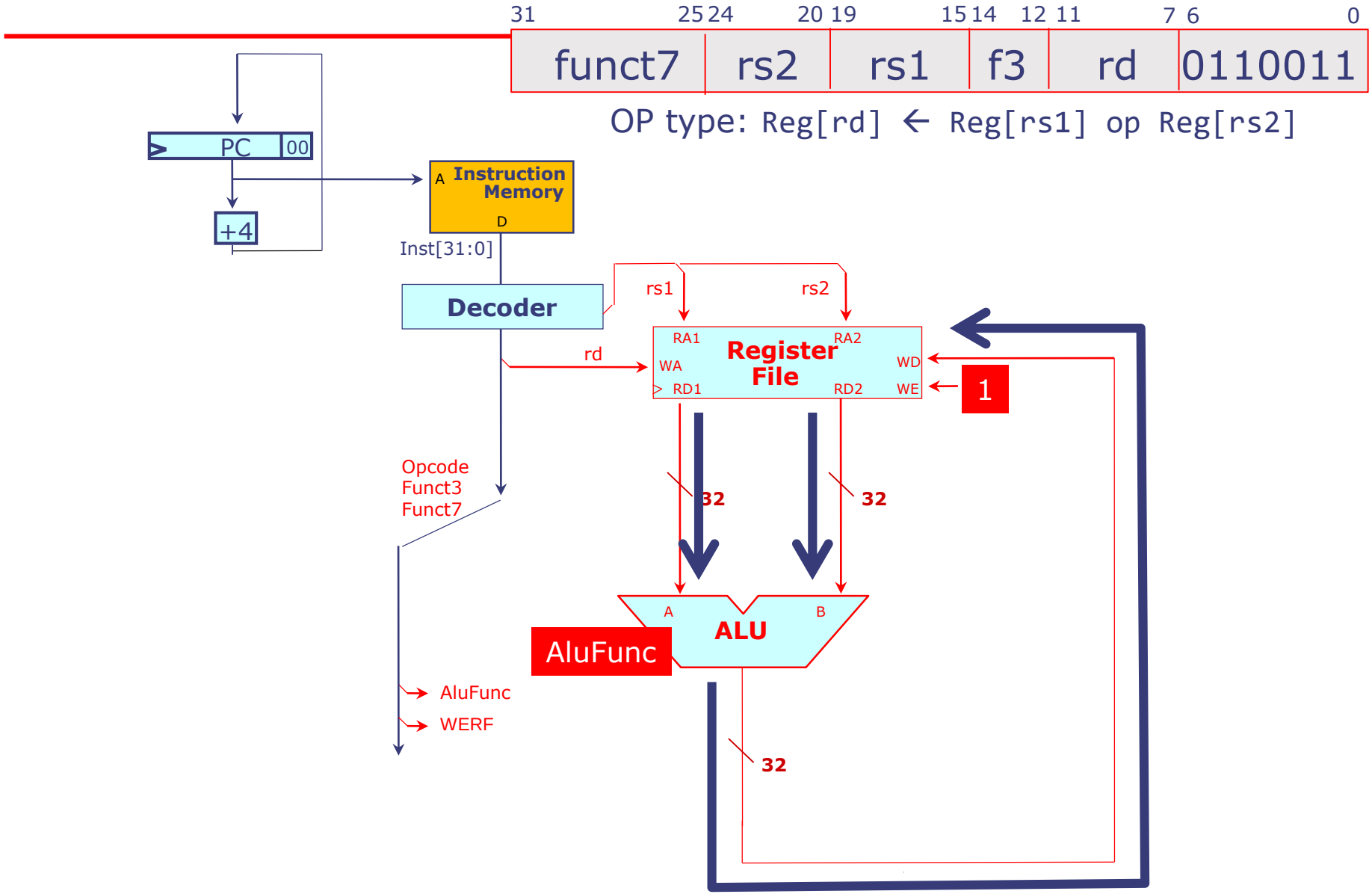
Register-Register ALU Datapath



Register-Register ALU Datapath



Register-Register ALU Datapath



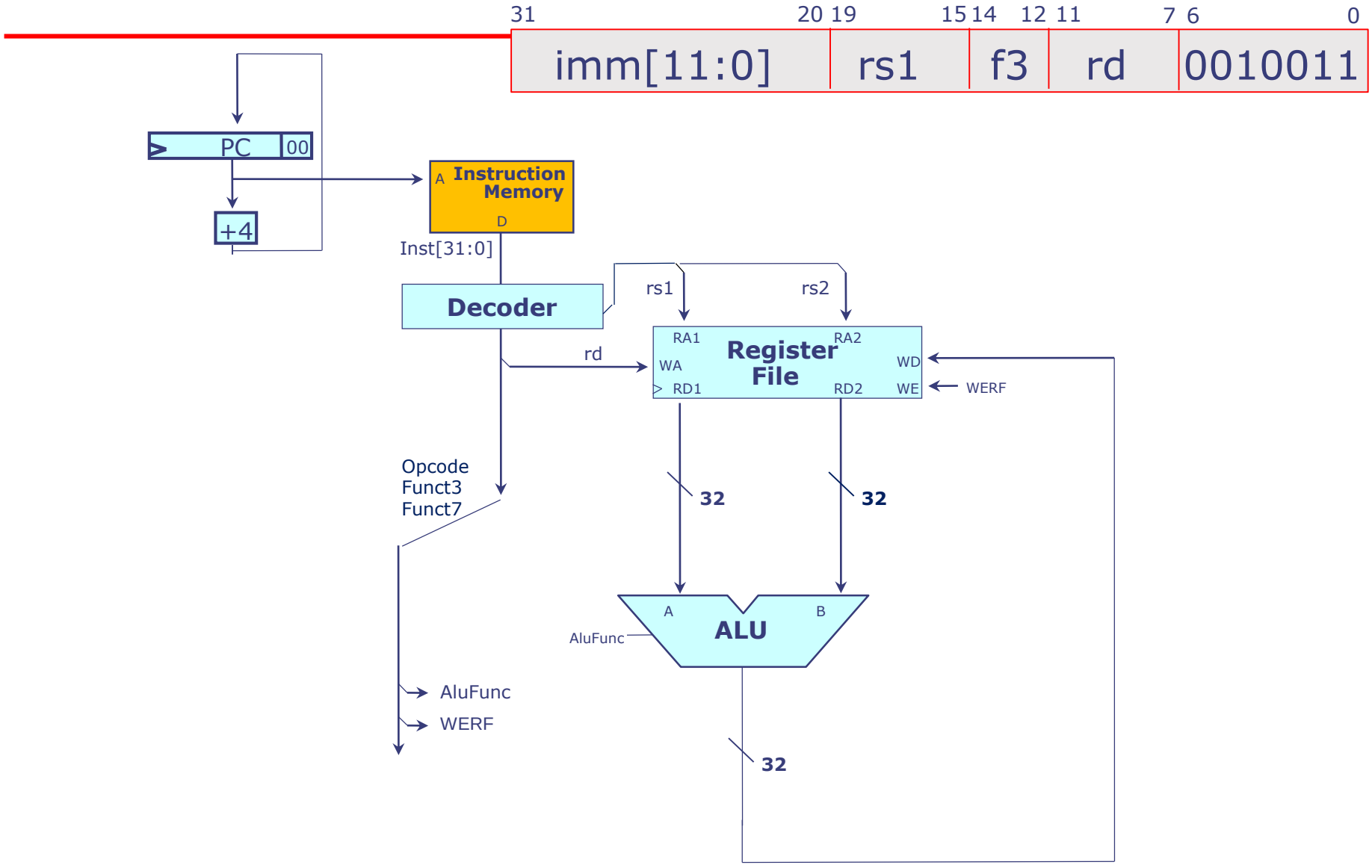
ALU Instructions

with one Immediate operand

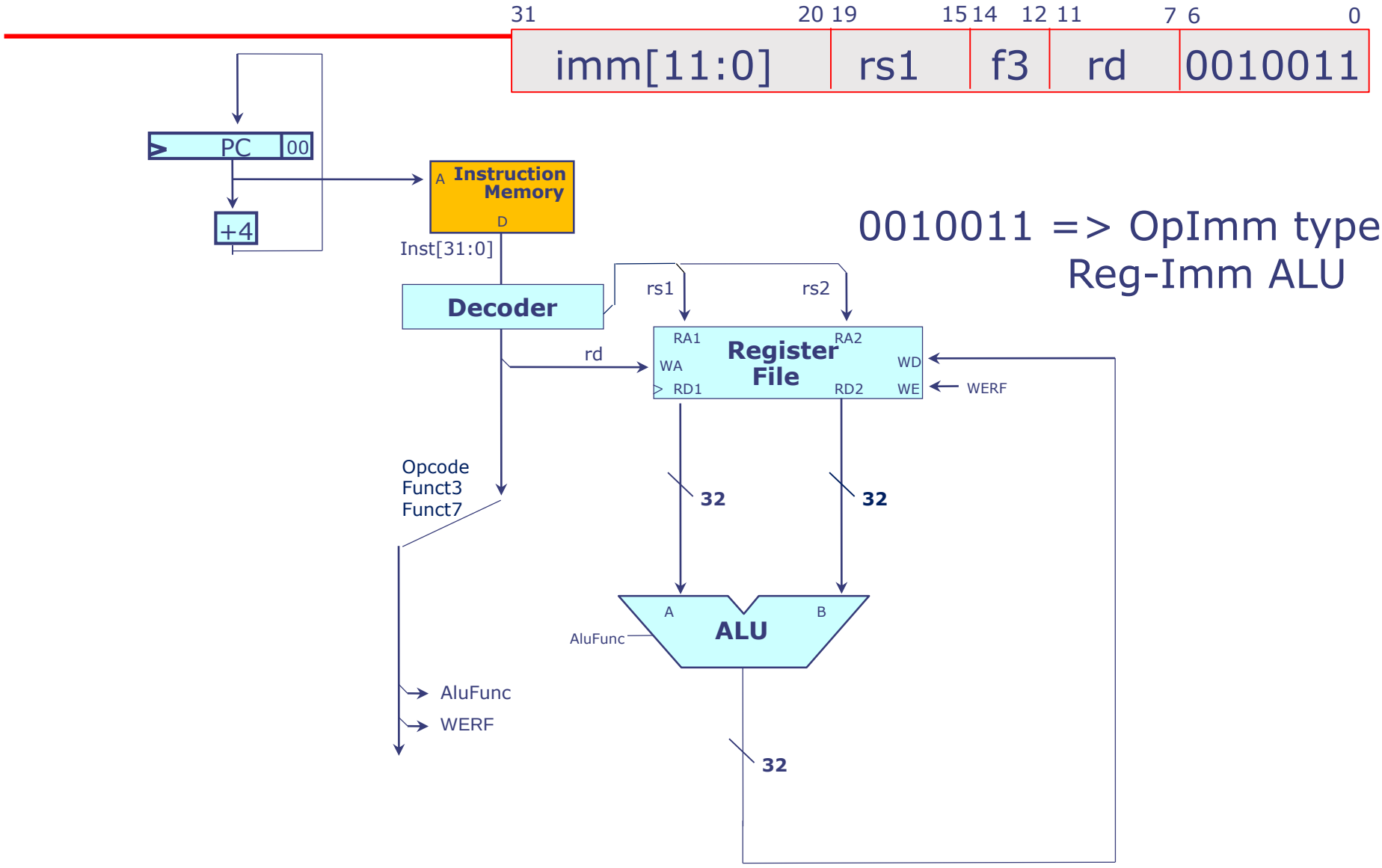
Instruction	Description	Execution
ADDI rd, rs1, const	Add Immediate	reg[rd] <= reg[rs1] + const
SLTI rd, rs1, const	Set if < Immediate (Signed)	reg[rd] <= (reg[rs1] $<_s$ const) ? 1 : 0
SLTIU rd, rs1, const	Sef it < Immediate (Unsigned)	reg[rd] <= (reg[rs1] $<_u$ const) ? 1 : 0
XORI rd, rs1, const	Xor Immediate	reg[rd] <= reg[rs1] ^ const
ORI rd, rs1, const	Or Immediate	reg[rd] <= reg[rs1] | const
ANDI rd, rs1, const	And Immediate	reg[rd] <= reg[rs1] & const
SLLI rd, rs1, shamt	Shift Left Logical Immediate	reg[rd] <= reg[rs1] << shamt
SRLI rd, rs1, shamt	Shift Right Logical Immediate	reg[rd] <= reg[rs1] $>>_u$ shamt
SRAI rd, rs1, shamt	Shift Right Arithmetic Immediate	reg[rd] <= reg[rs1] $>>_s$ shamt

These instructions are grouped in a category called OPIMM with fields (AluFunc, rd, rs1, immI32)

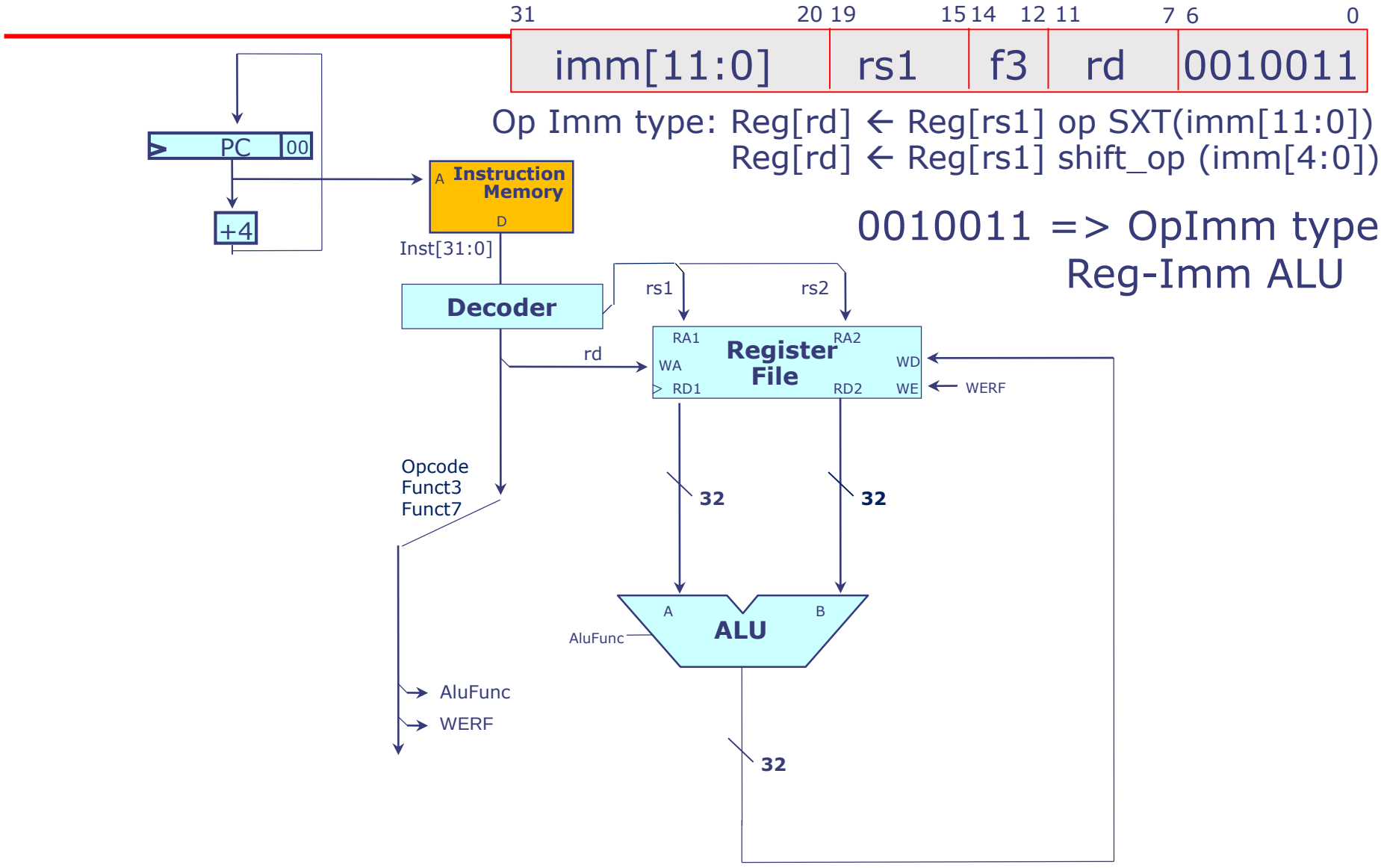
Register-Immediate ALU Datapath



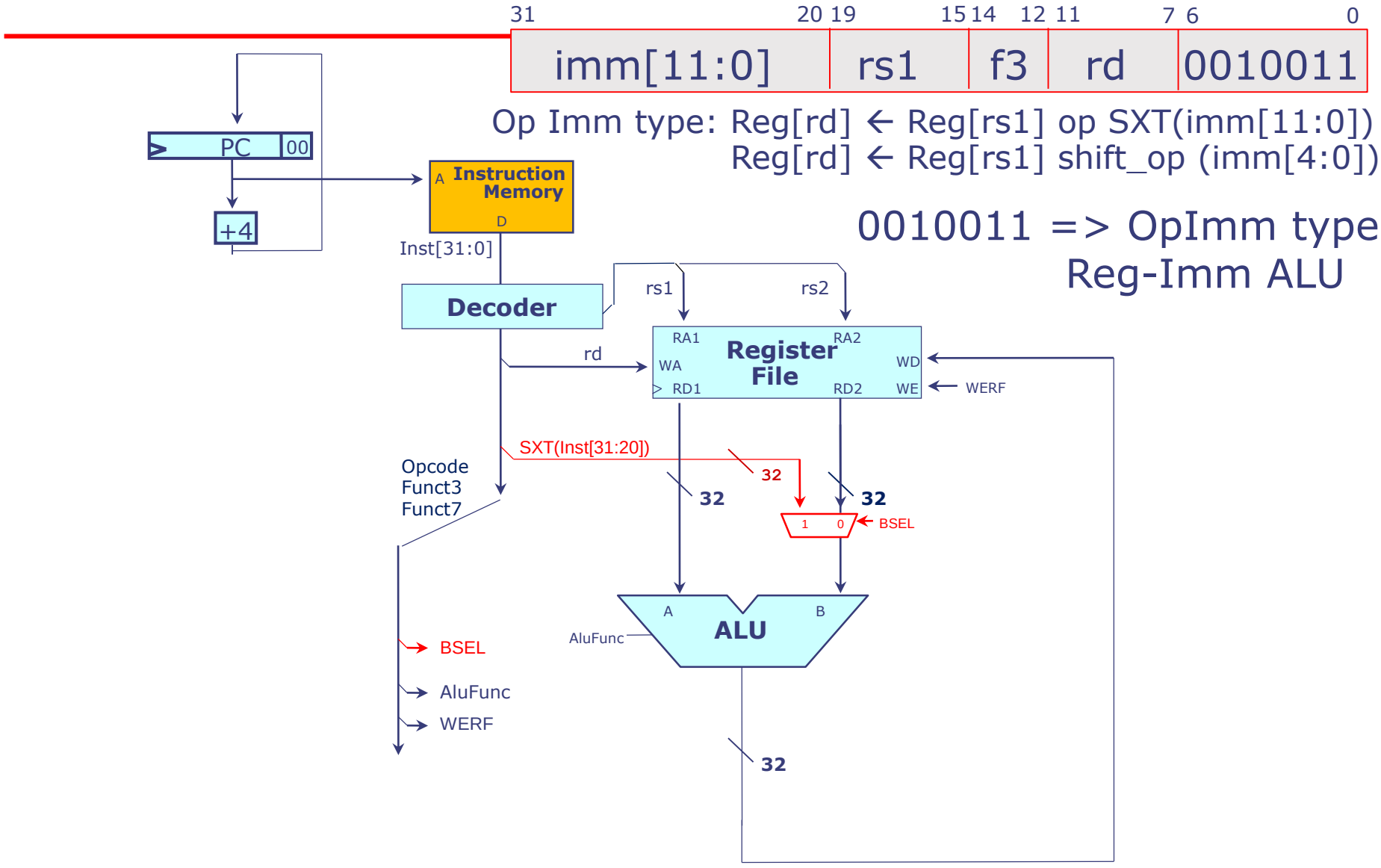
Register-Immediate ALU Datapath



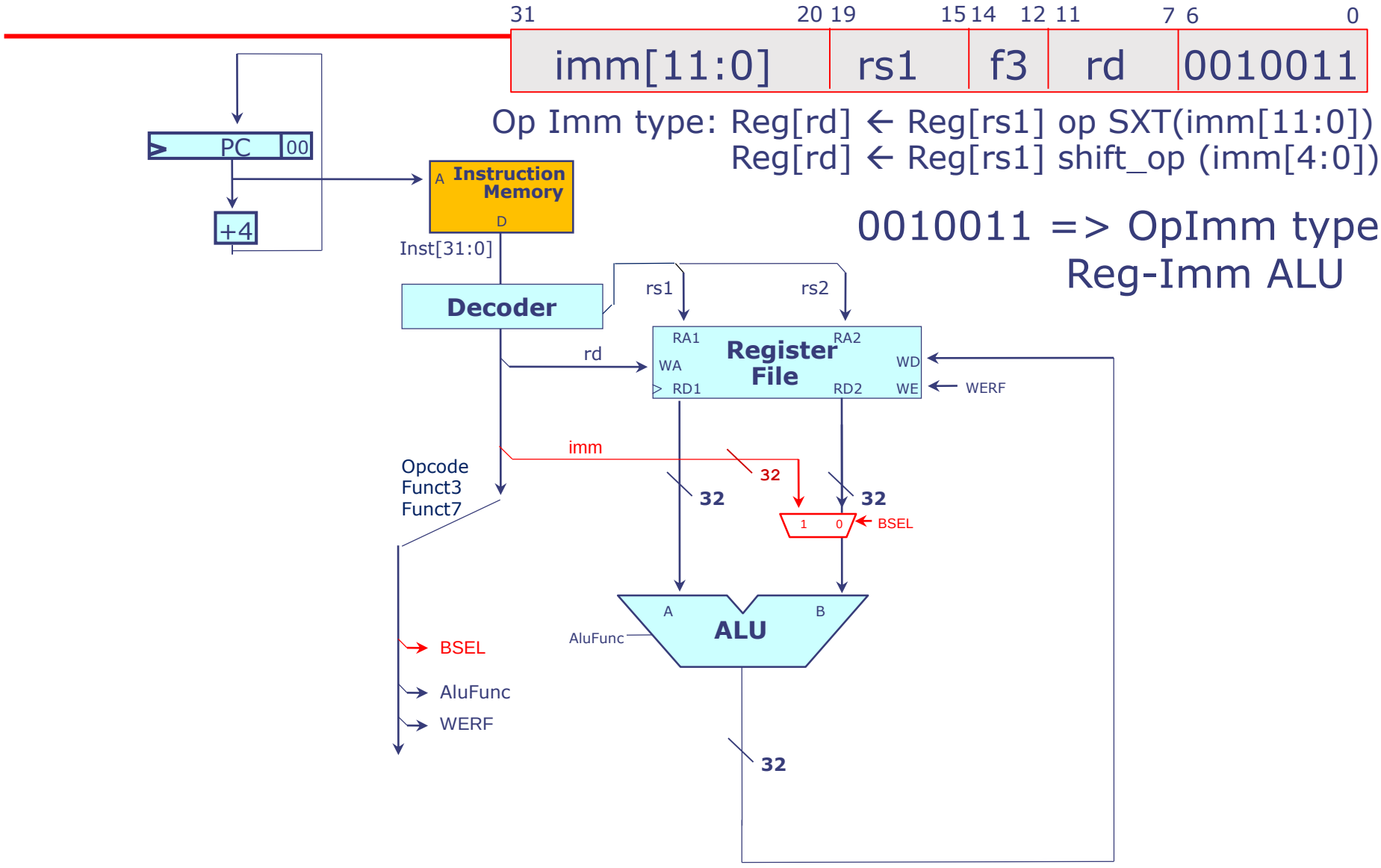
Register-Immediate ALU Datapath



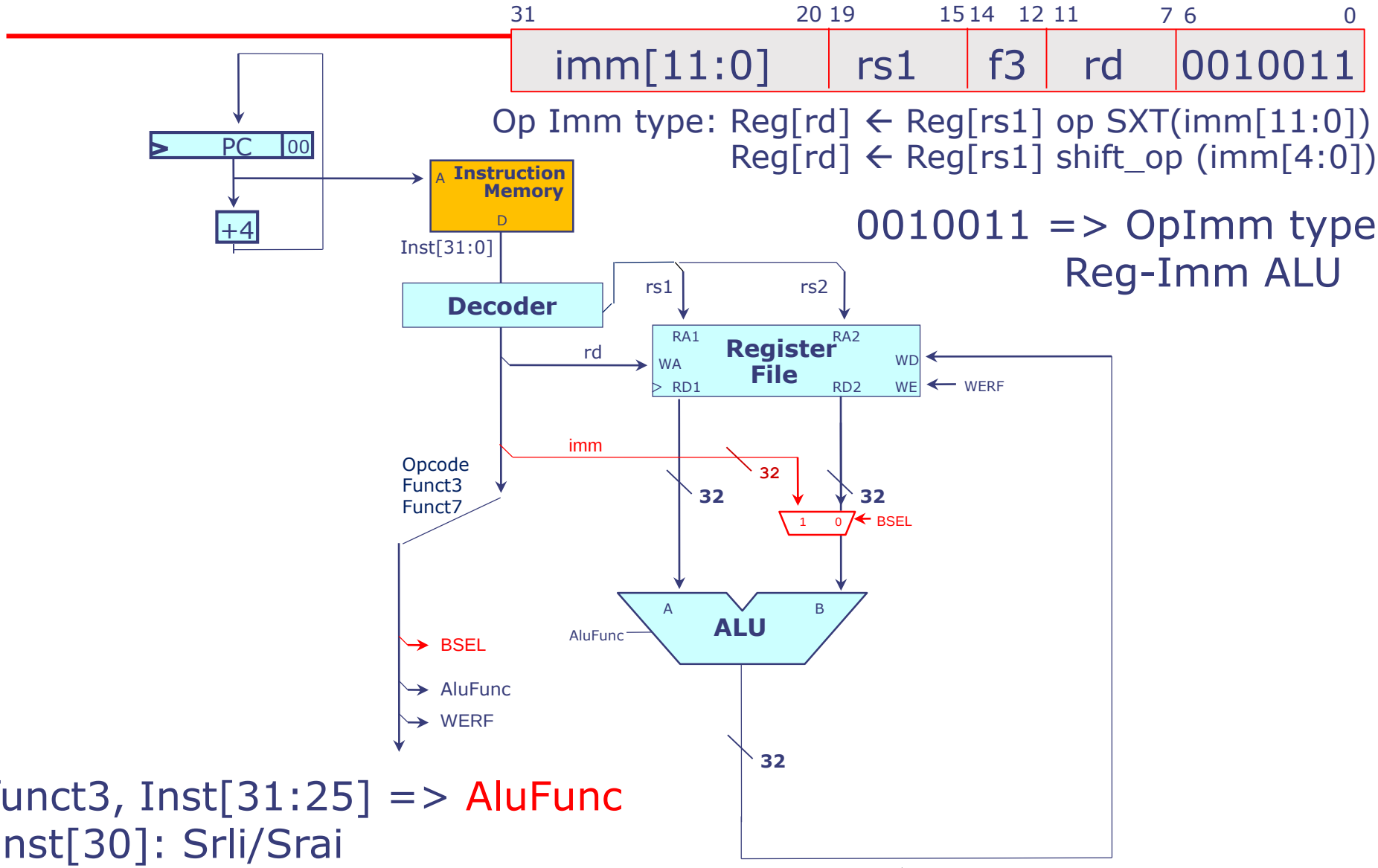
Register-Immediate ALU Datapath



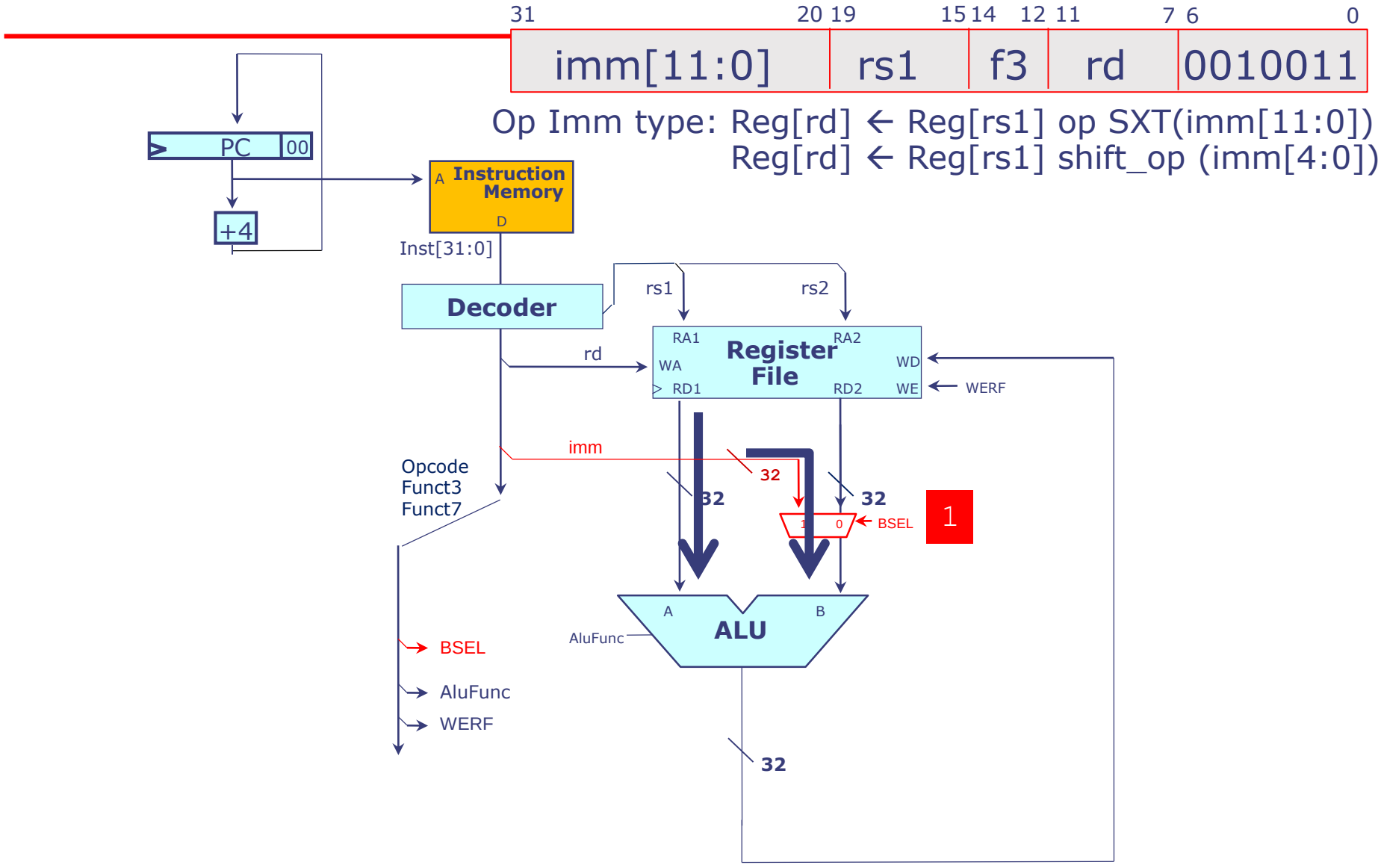
Register-Immediate ALU Datapath



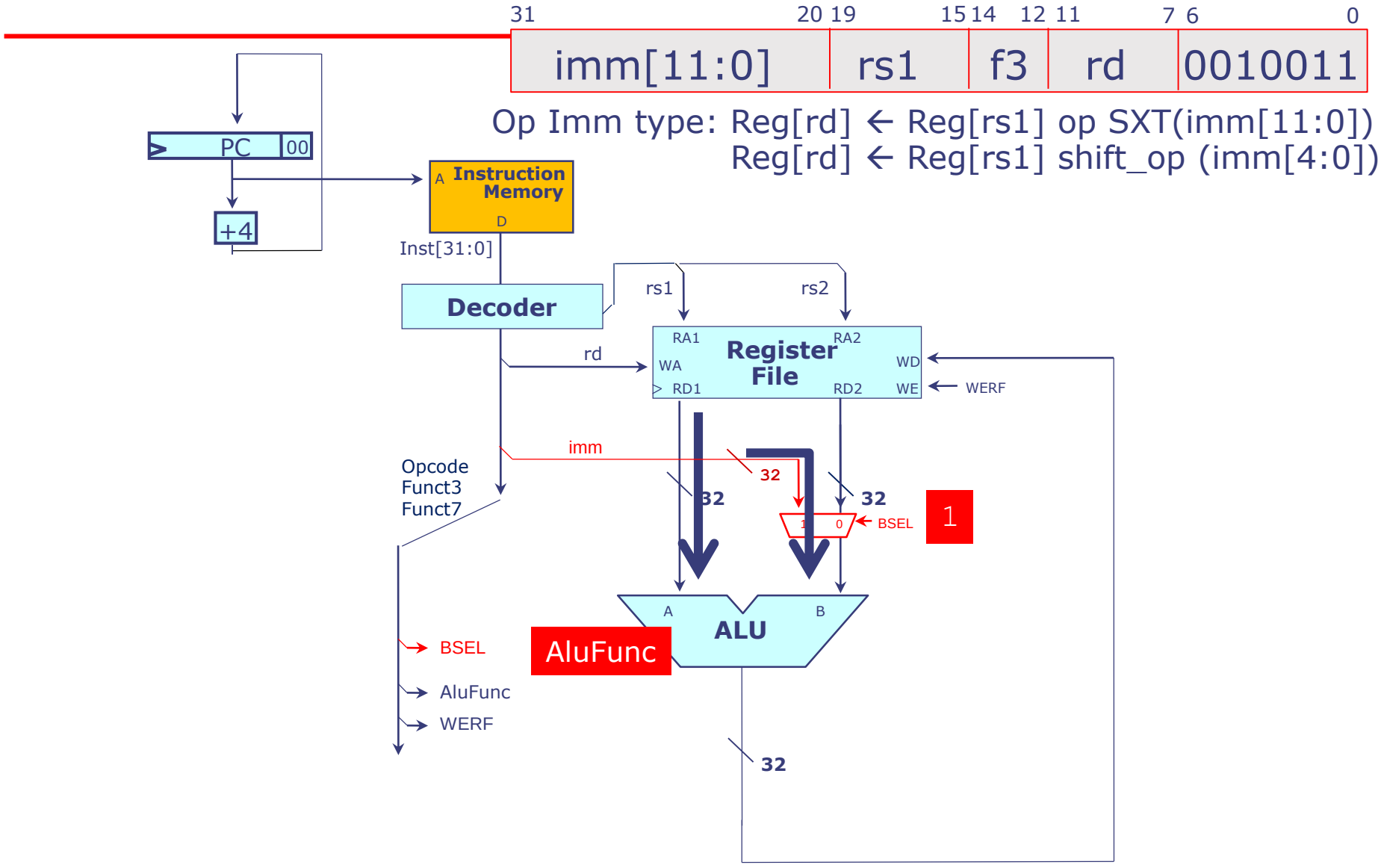
Register-Immediate ALU Datapath



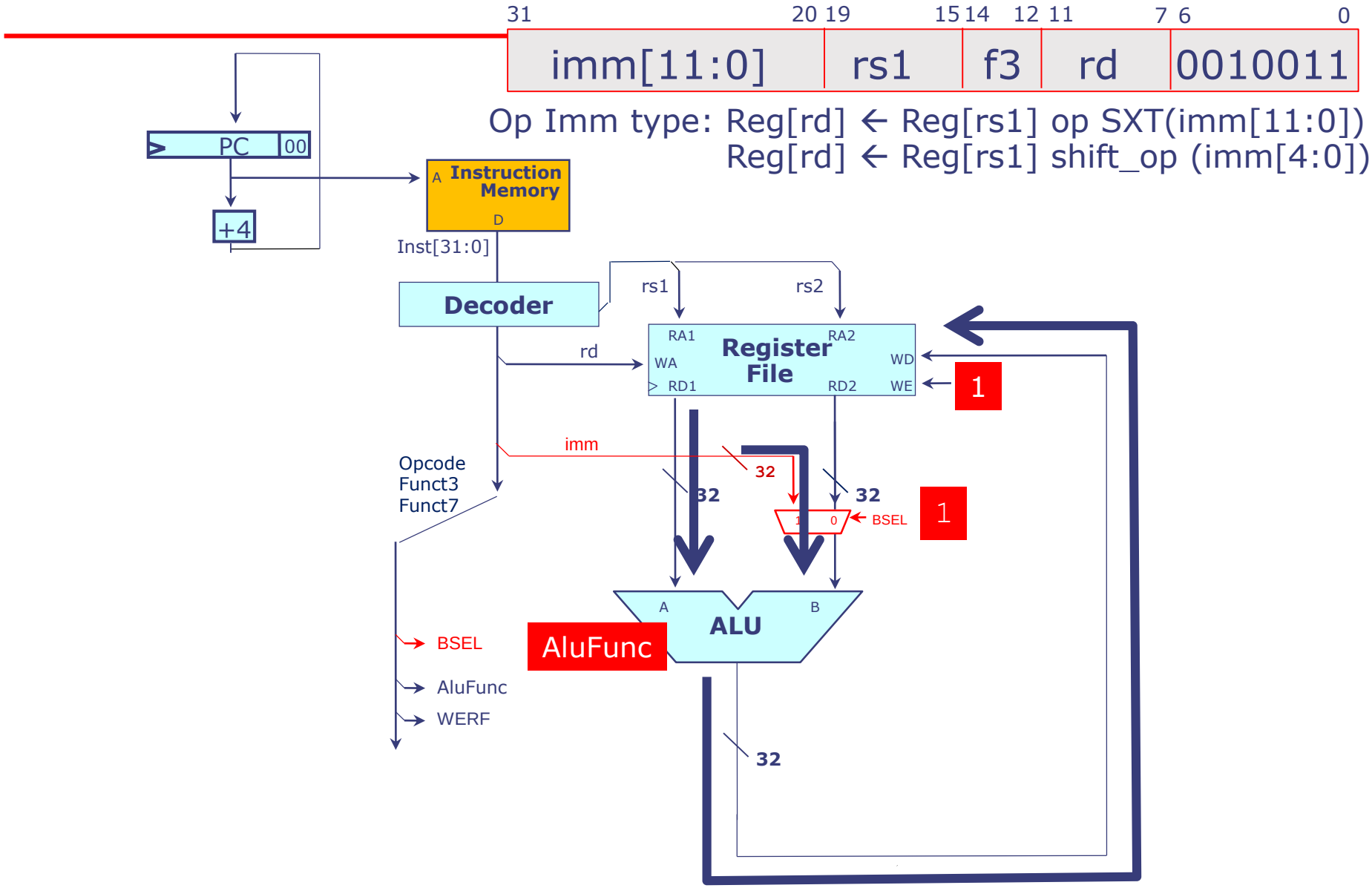
Register-Immediate ALU Datapath



Register-Immediate ALU Datapath



Register-Immediate ALU Datapath

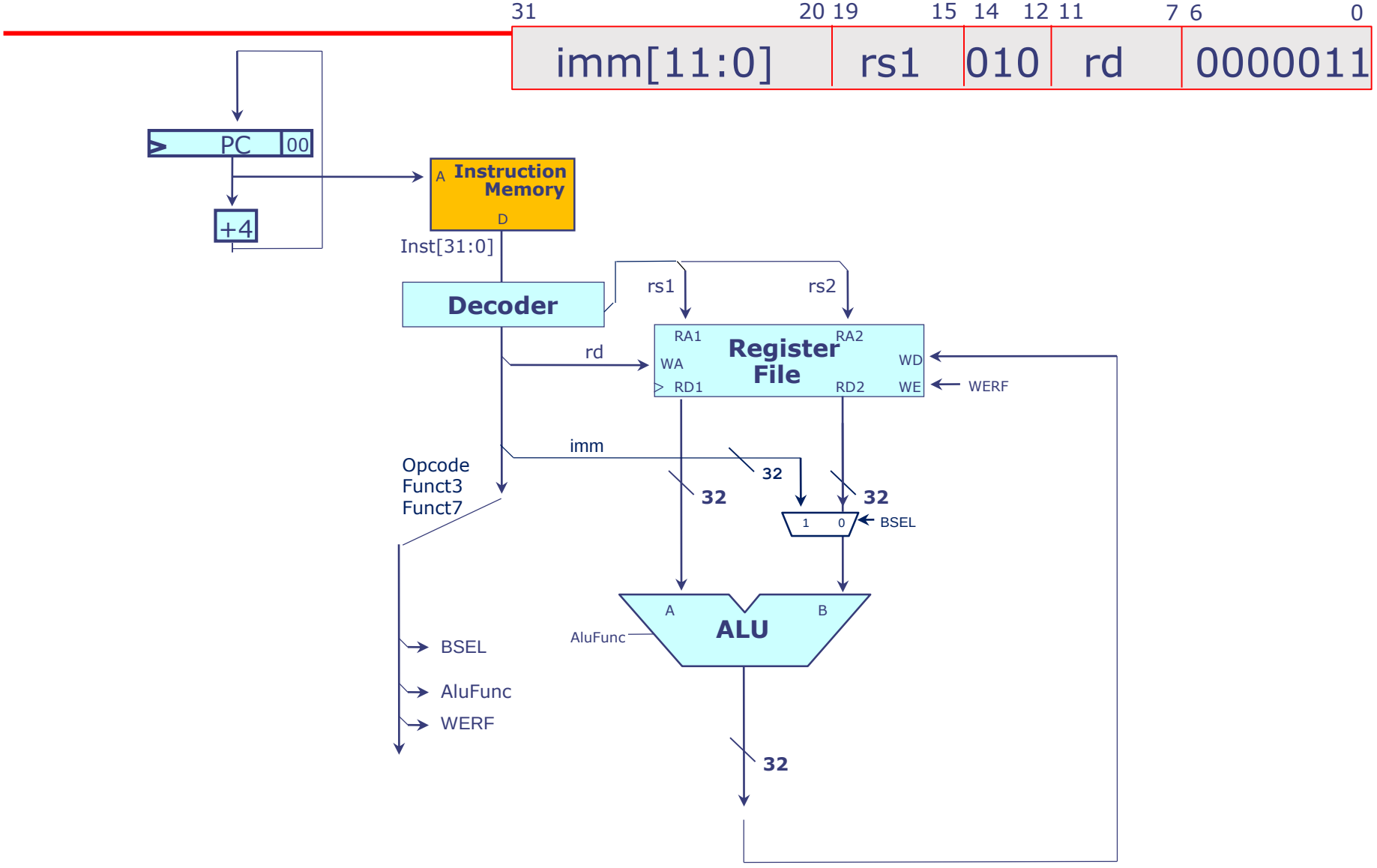


Load and Store Instructions

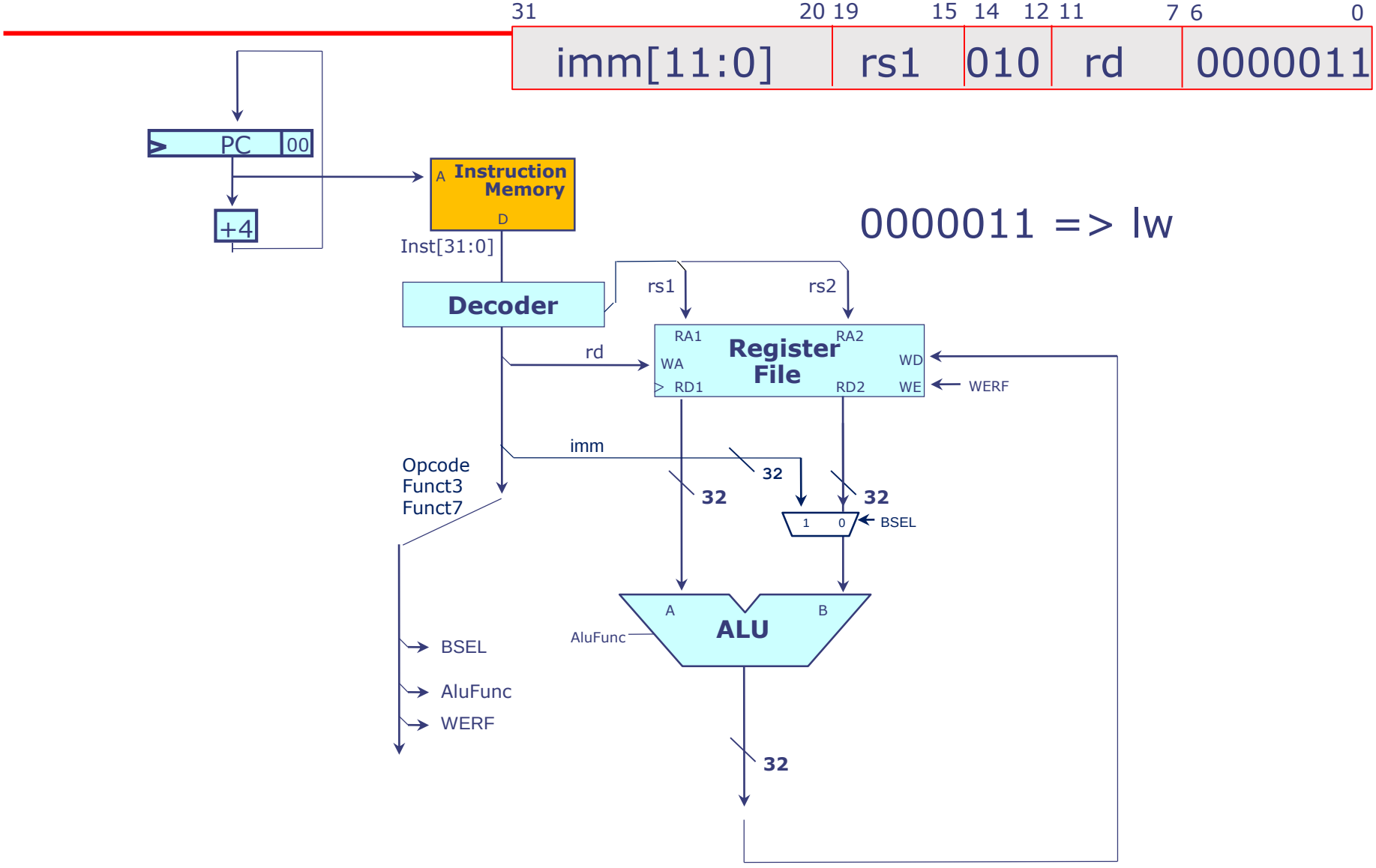
Instruction	Description	Execution
LW rd, offset(rs1)	Load Word	reg[rd] <= mem[reg[rs1] + offset]
SW rs2, offset(rs1)	Store Word	mem[reg[rs1] + offset] <= reg[rs2]

LW and SW need to access memory for execution,
so they need to compute an effective memory address

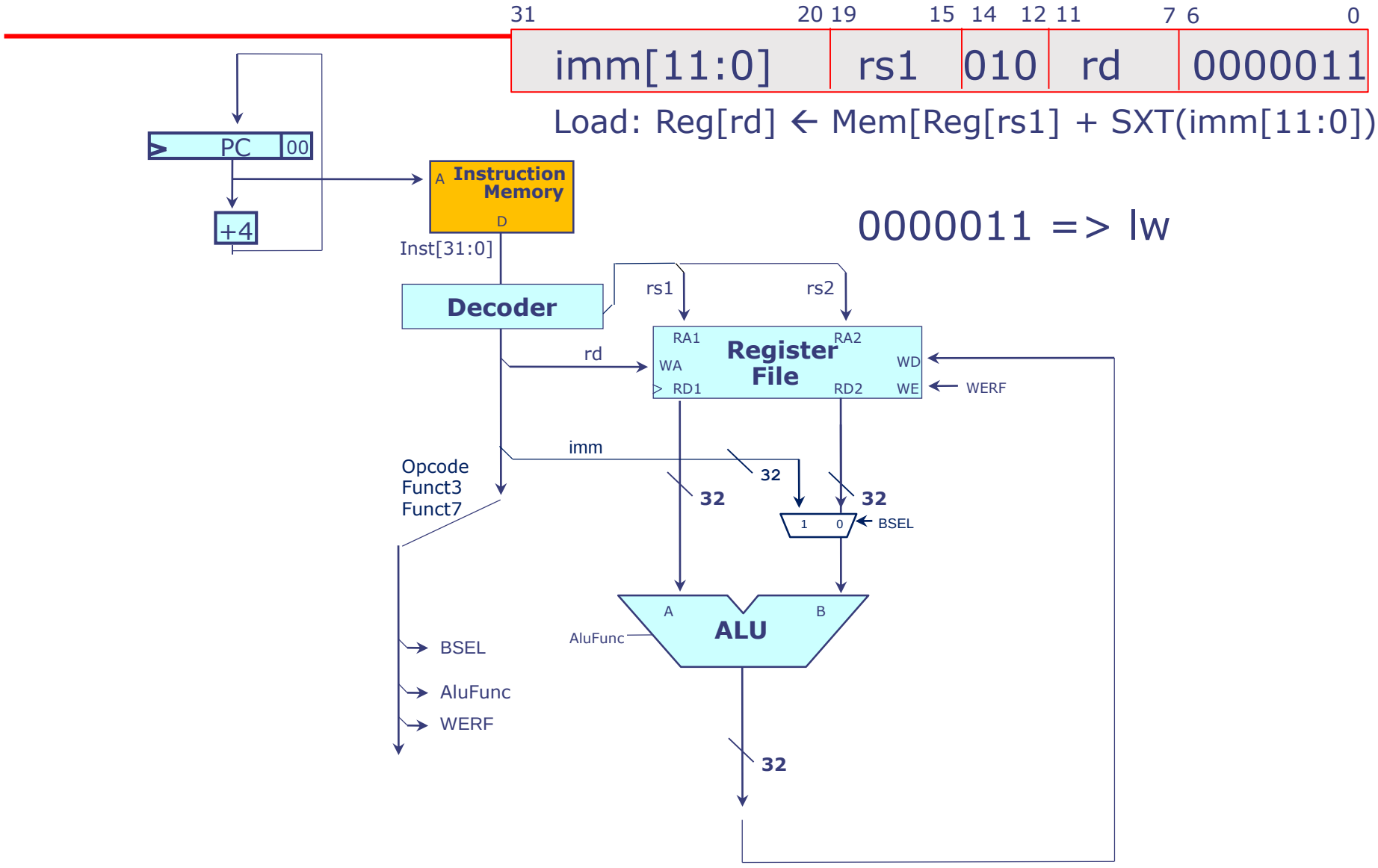
Load Instruction



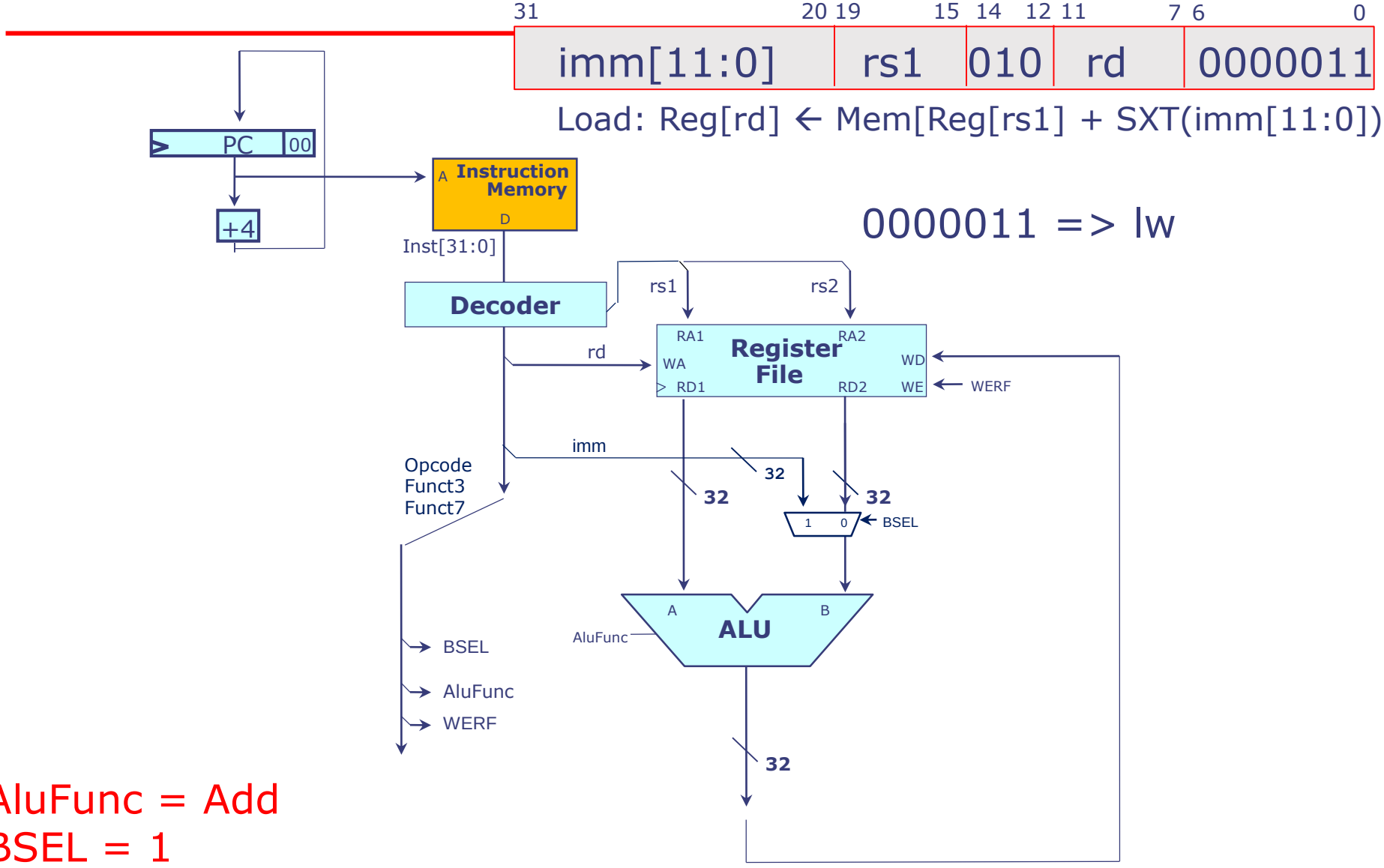
Load Instruction



Load Instruction

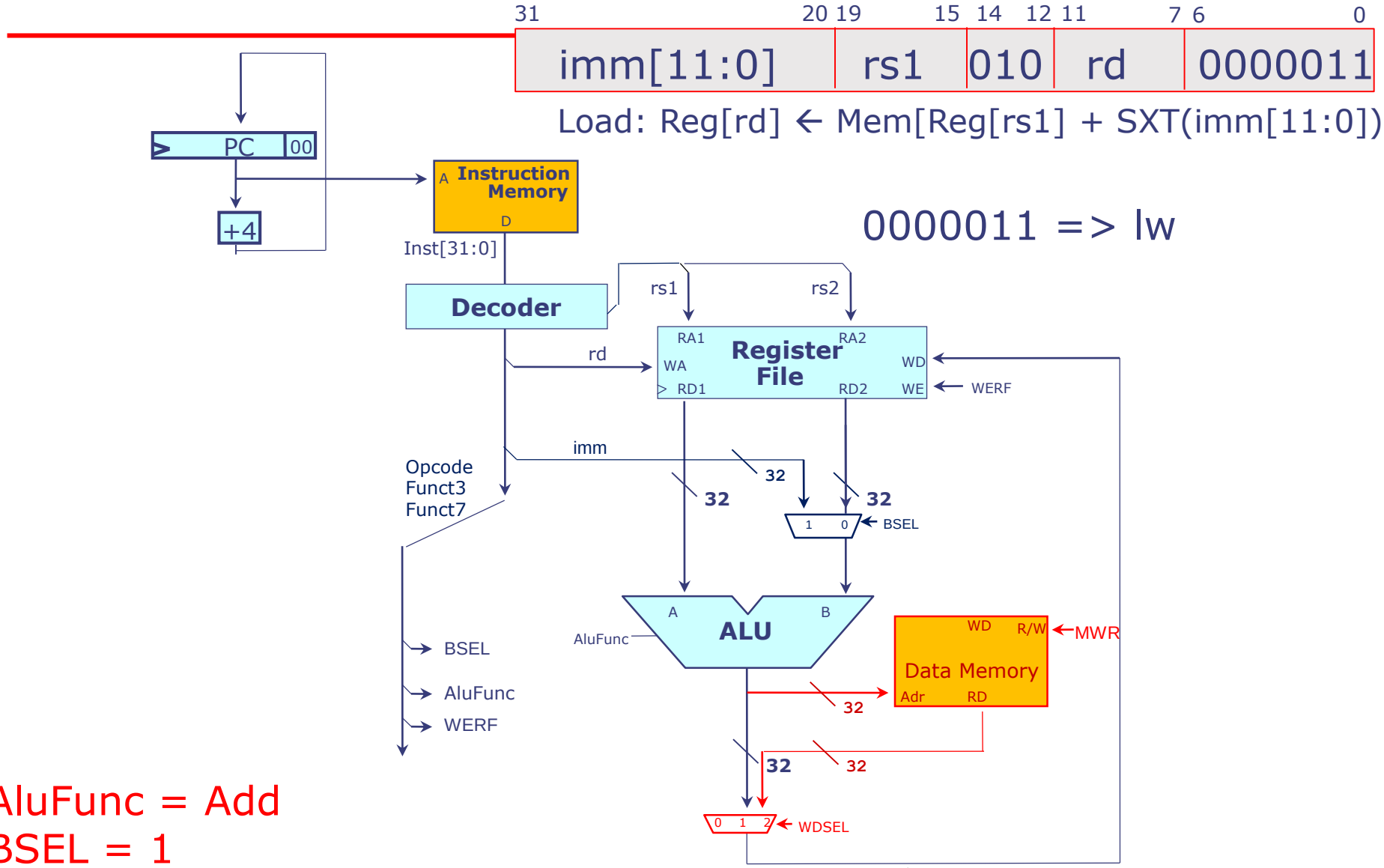


Load Instruction



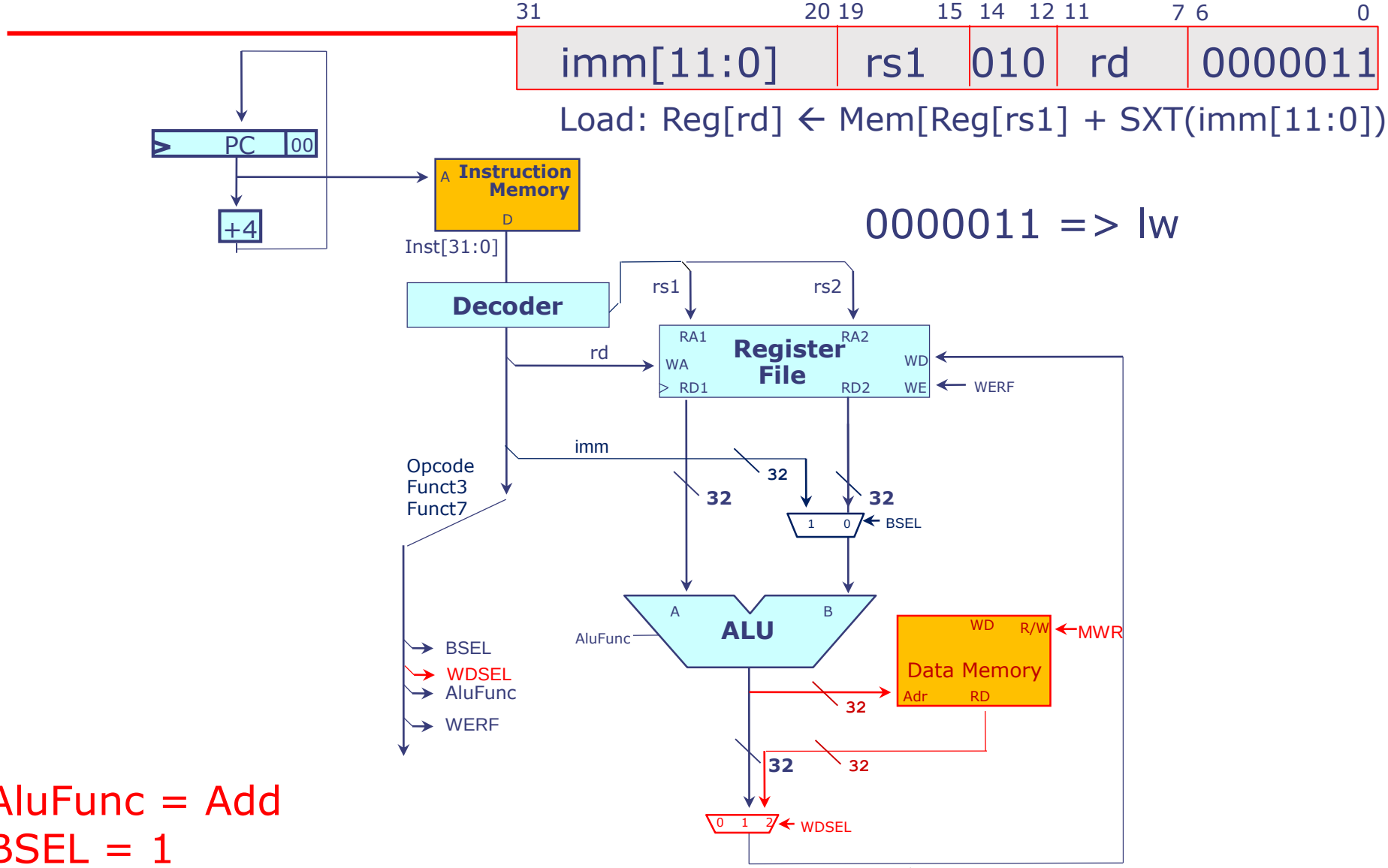
AluFunc = Add
BSEL = 1

Load Instruction



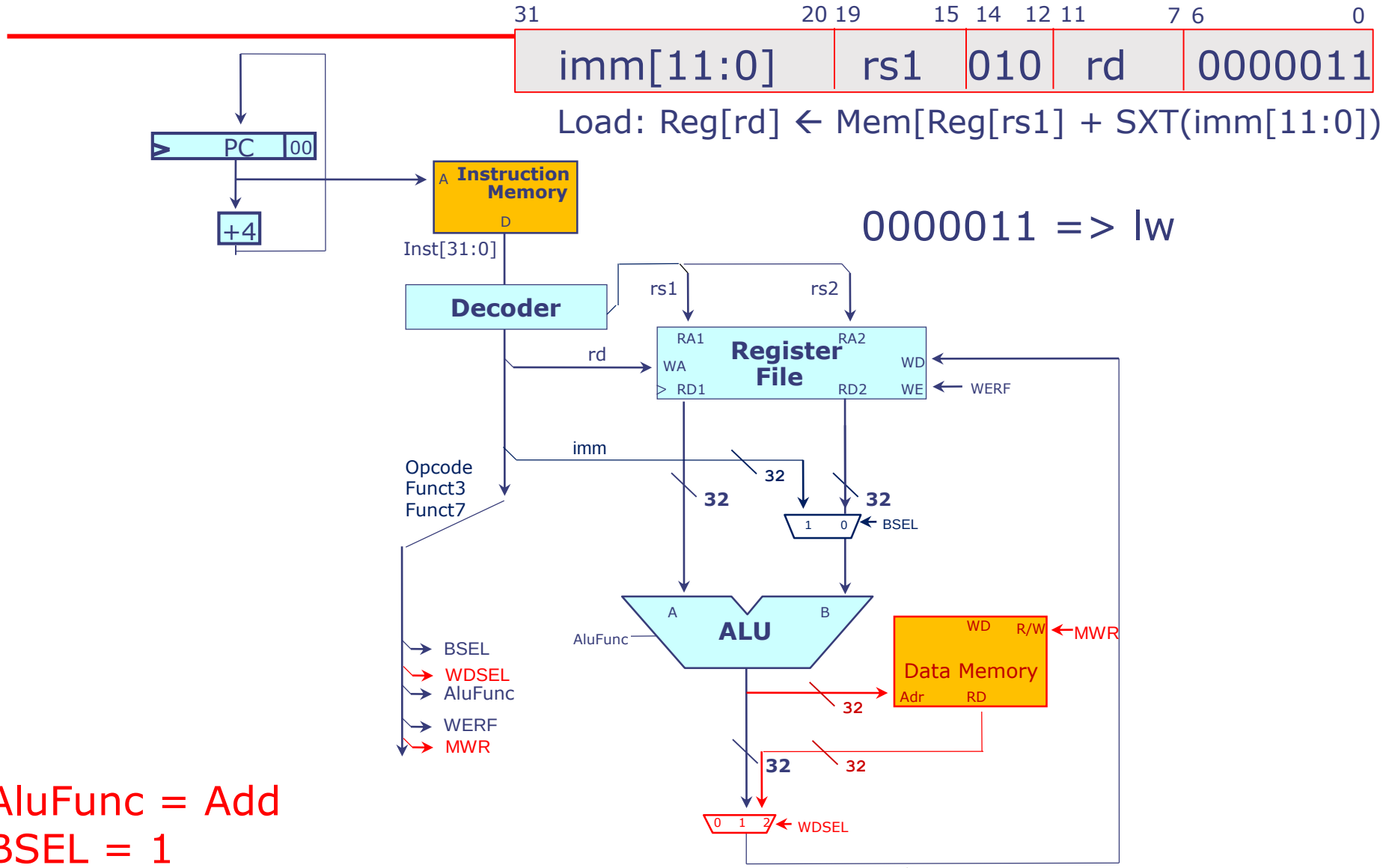
AluFunc = Add
BSEL = 1

Load Instruction



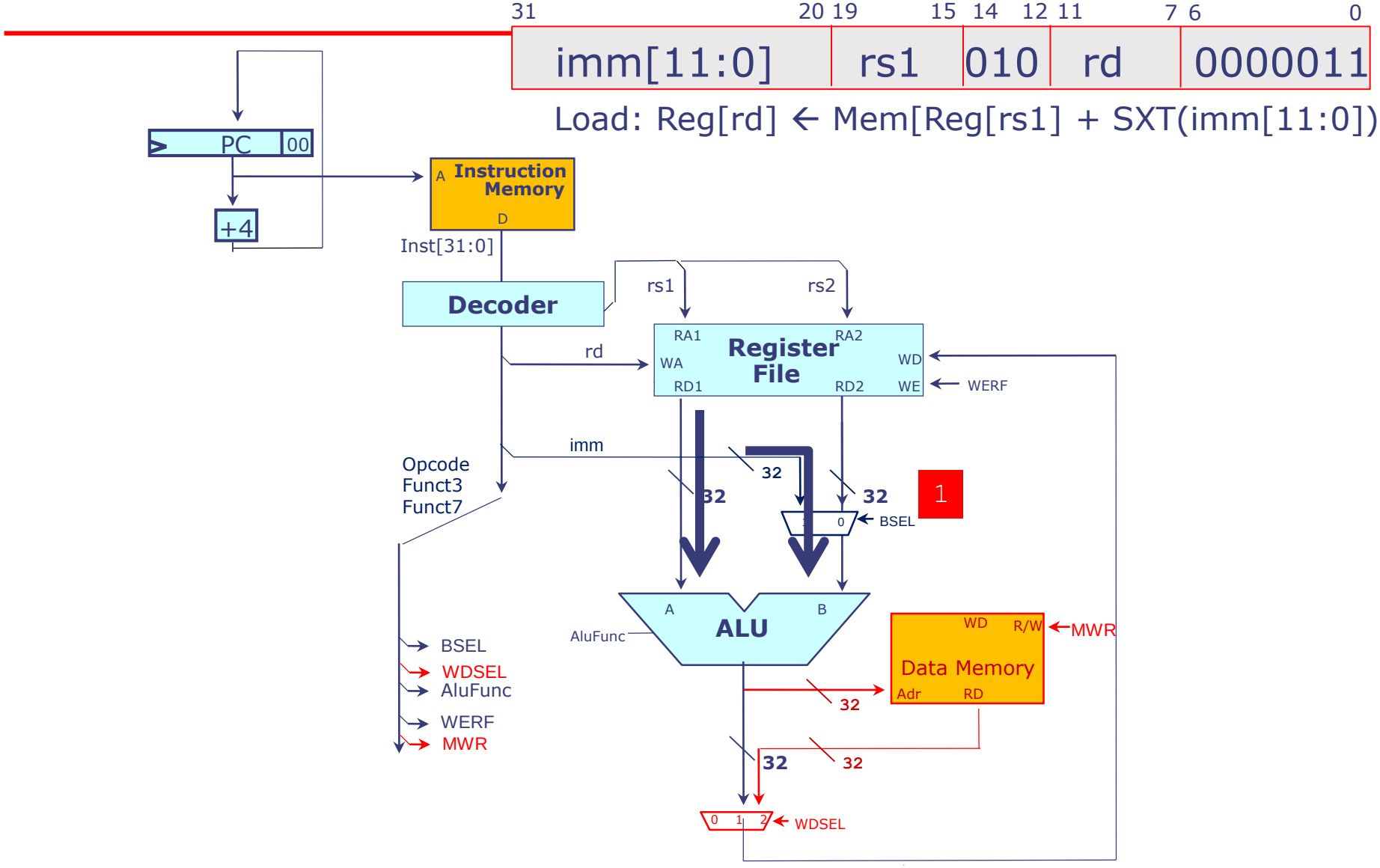
AluFunc = Add
BSEL = 1

Load Instruction

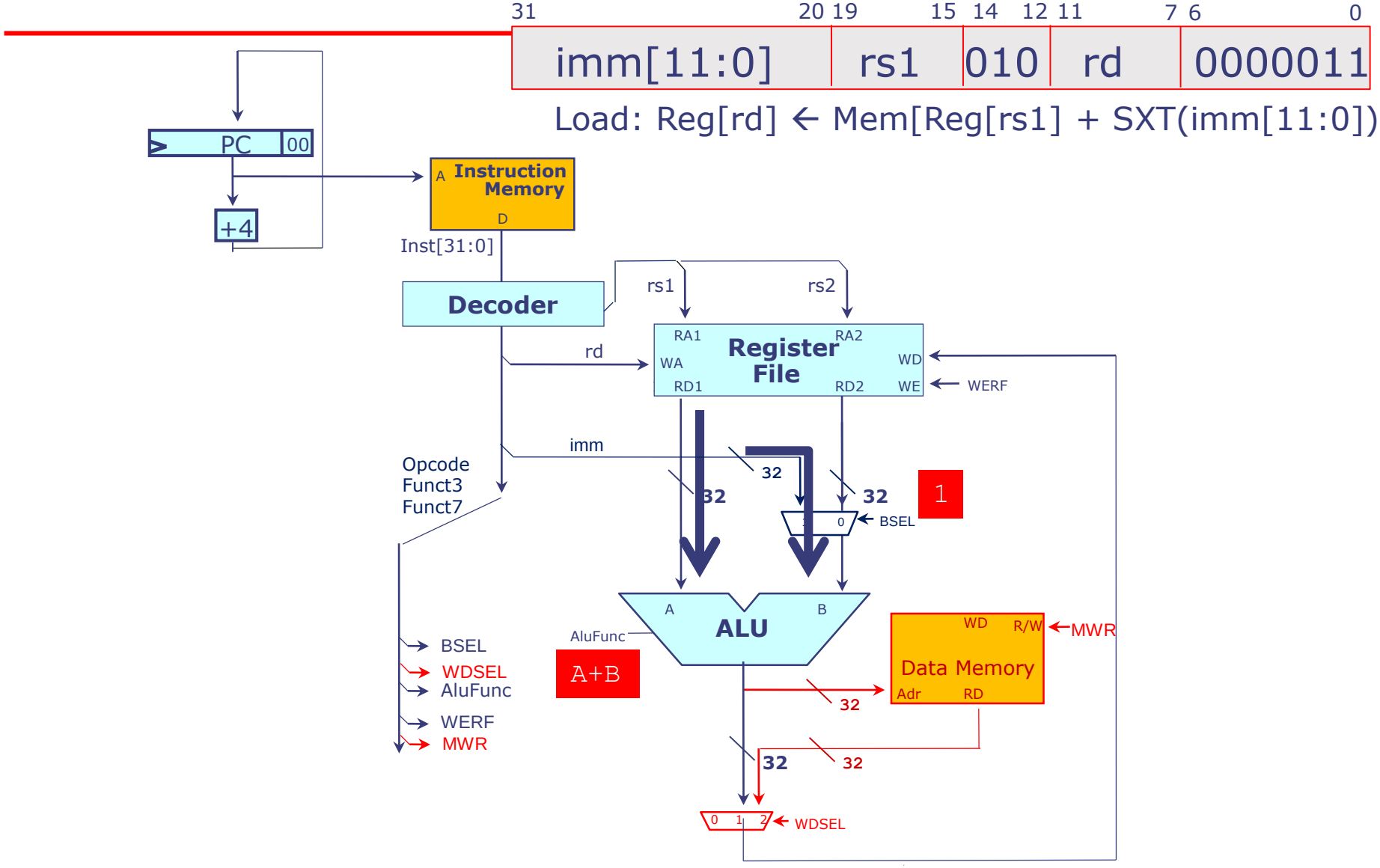


AluFunc = Add
BSEL = 1

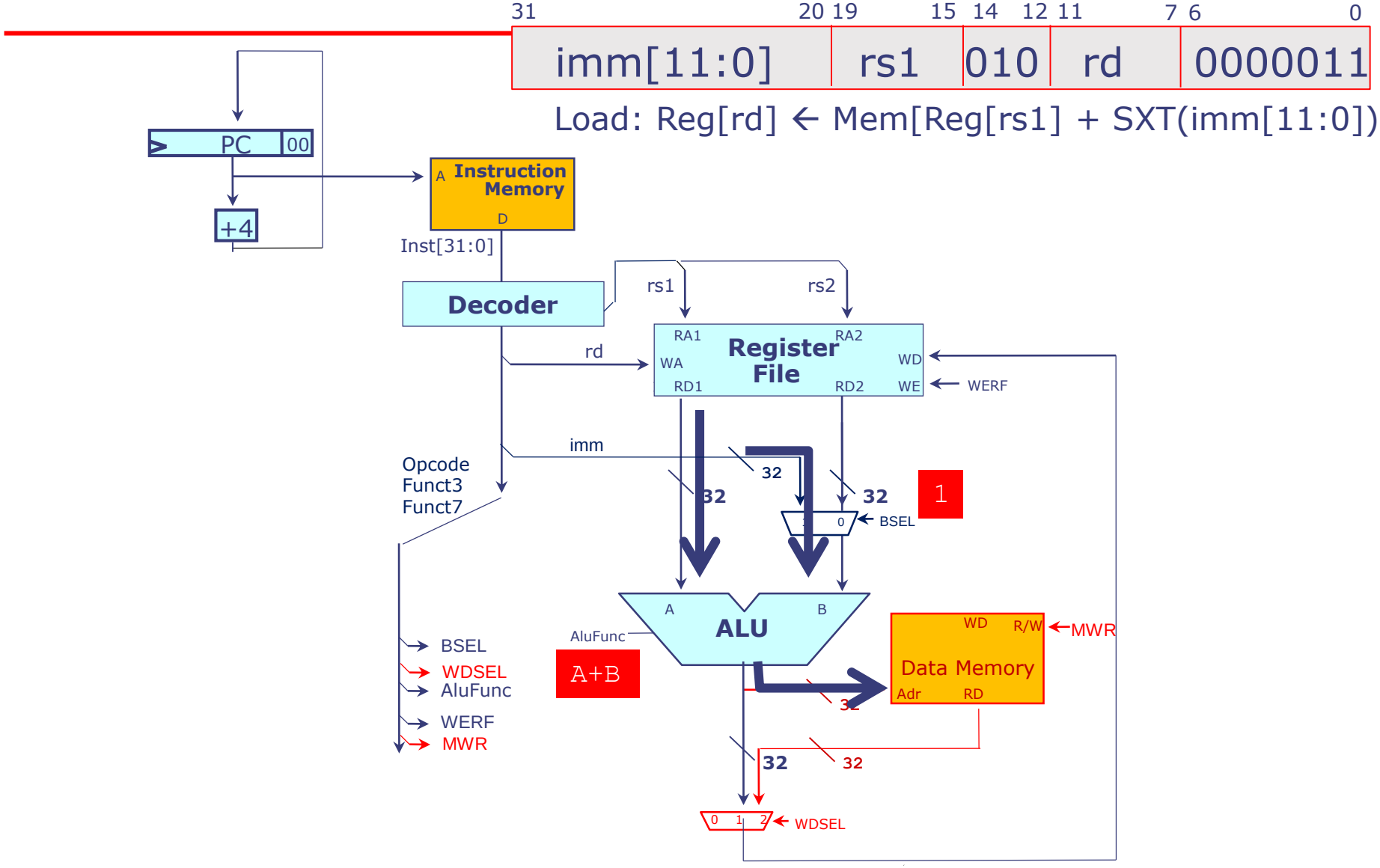
Load Instruction



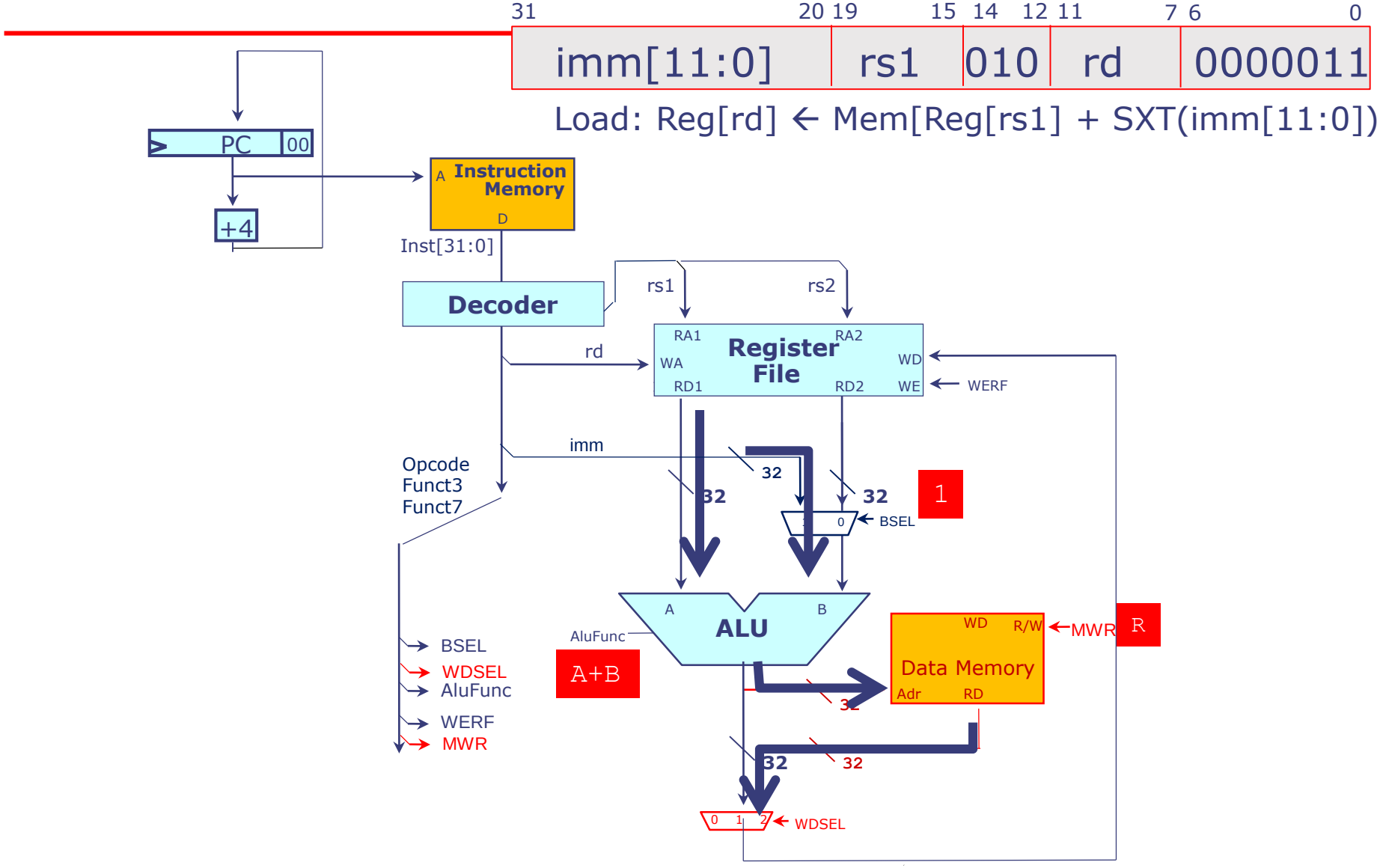
Load Instruction



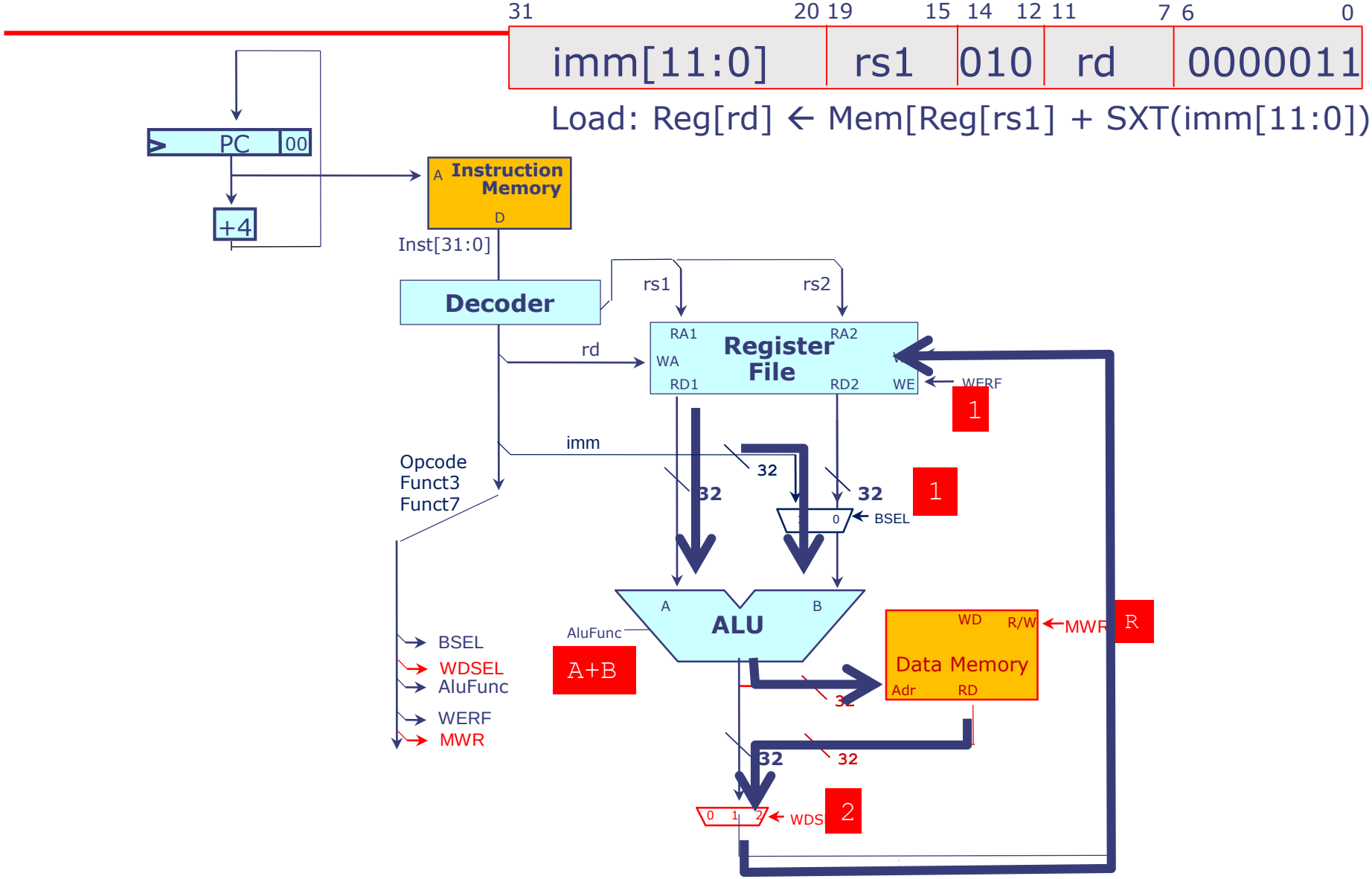
Load Instruction



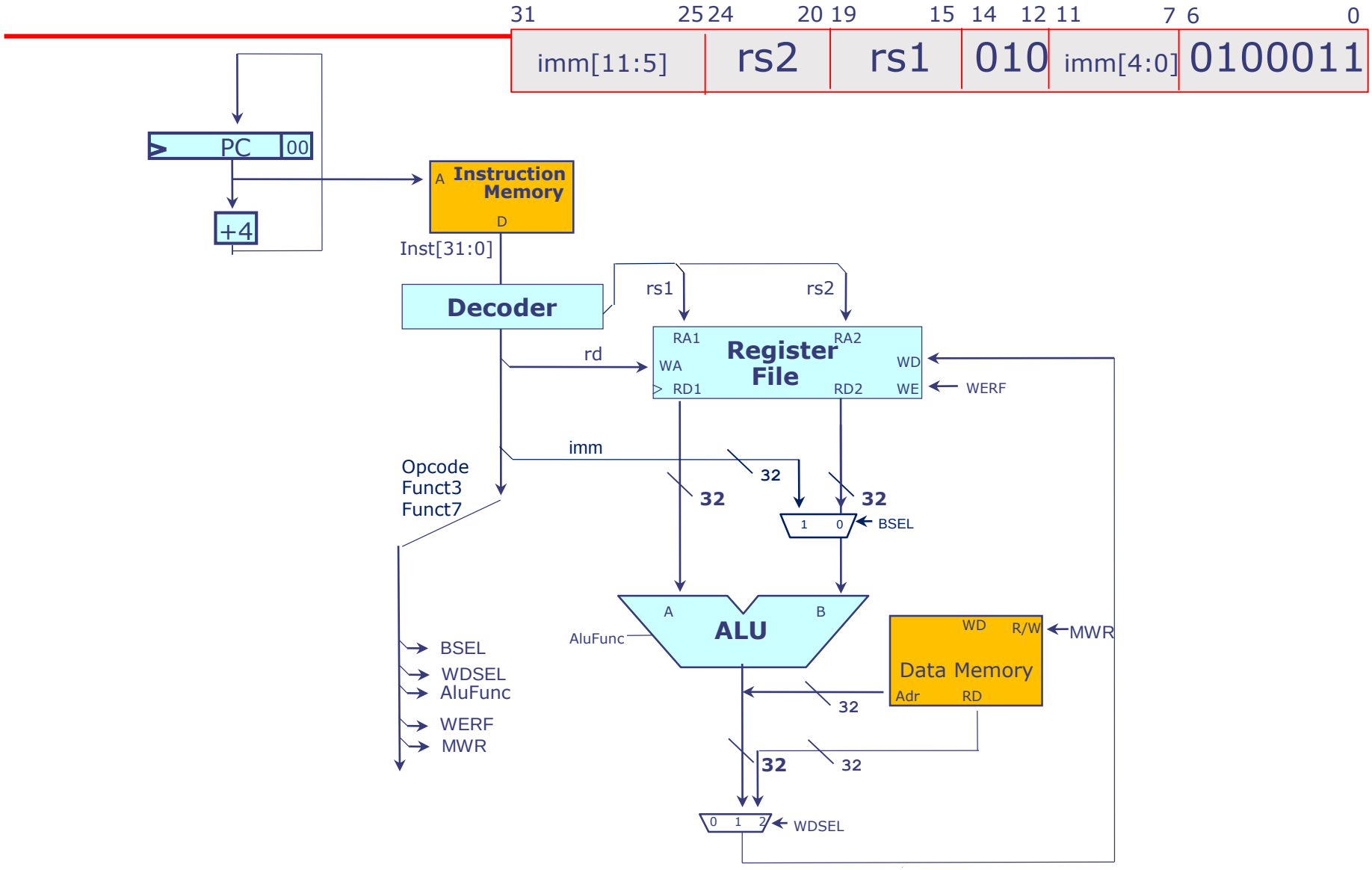
Load Instruction



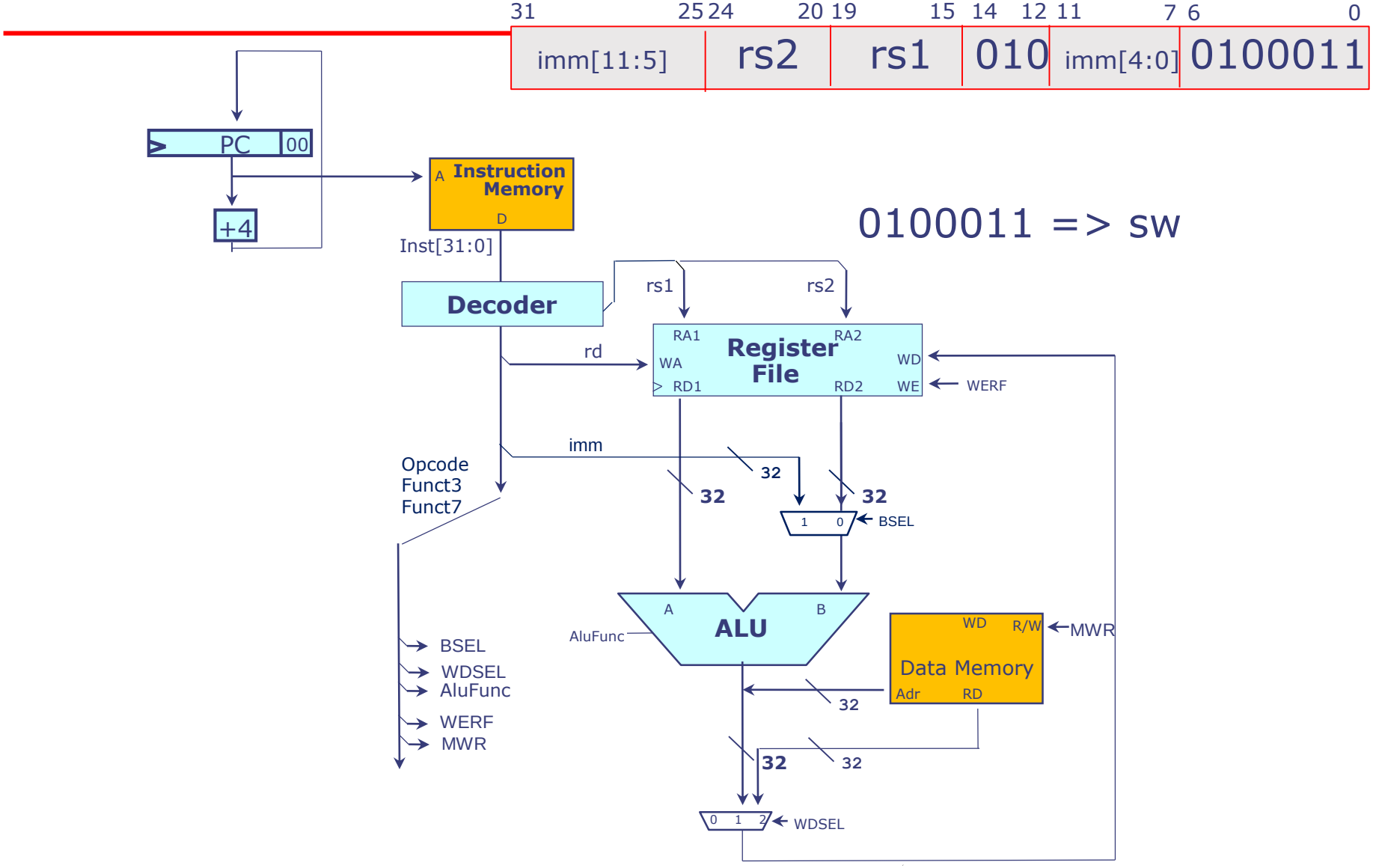
Load Instruction



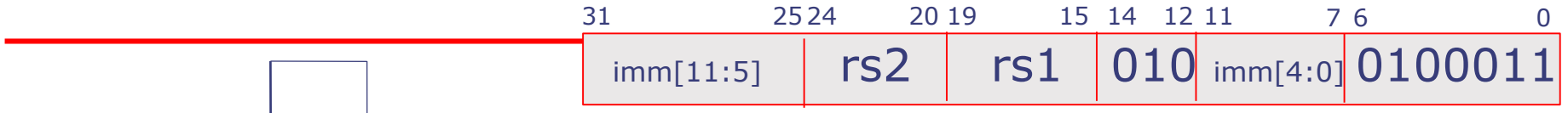
Store Instruction



Store Instruction

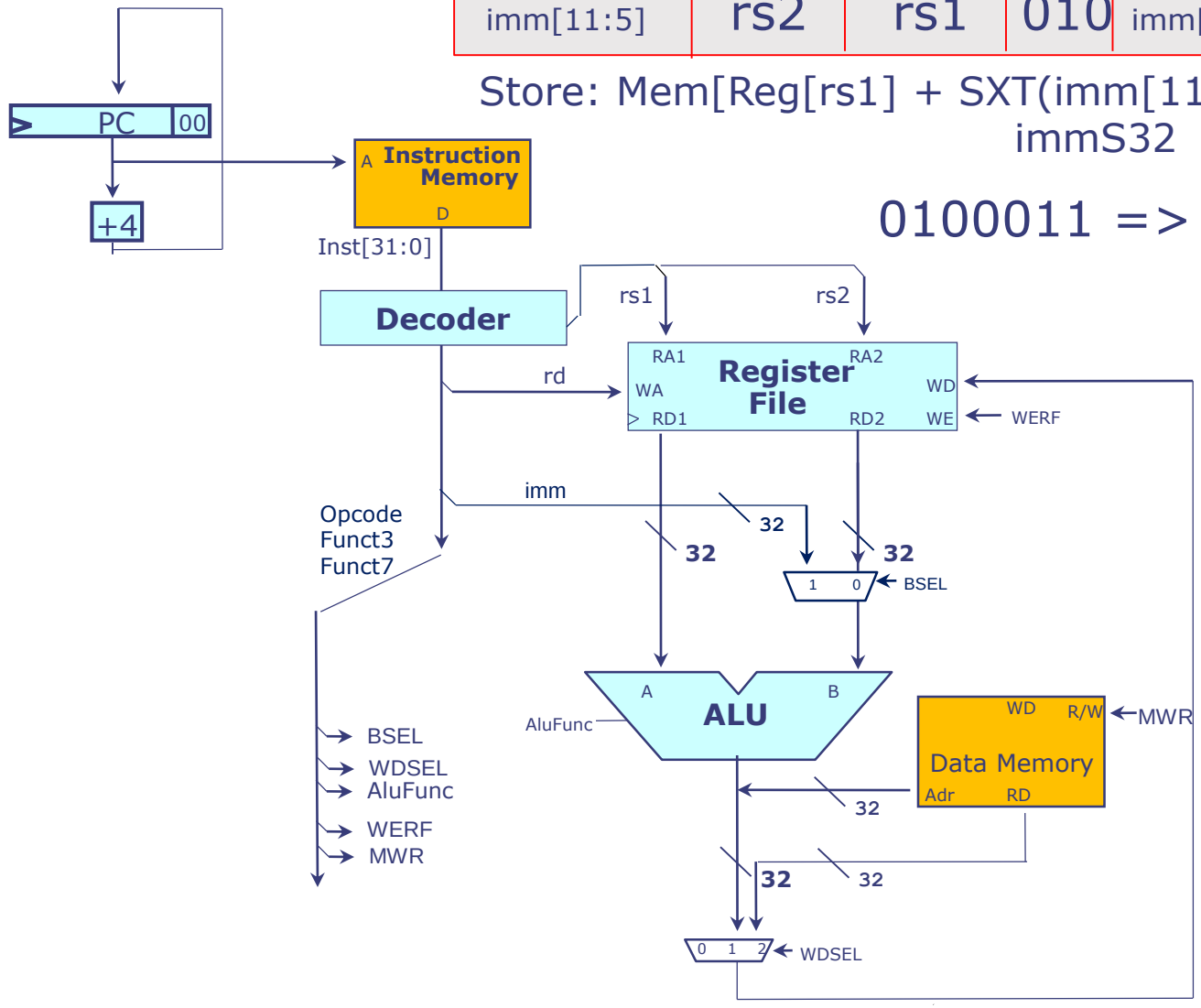


Store Instruction

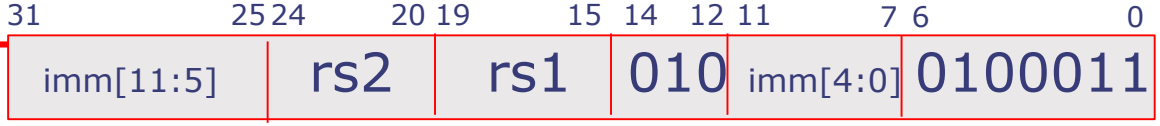


Store: Mem[Reg[rs1] + SXT(imm[11:0]) ← Reg[rs2]
immS32

0100011 => sw

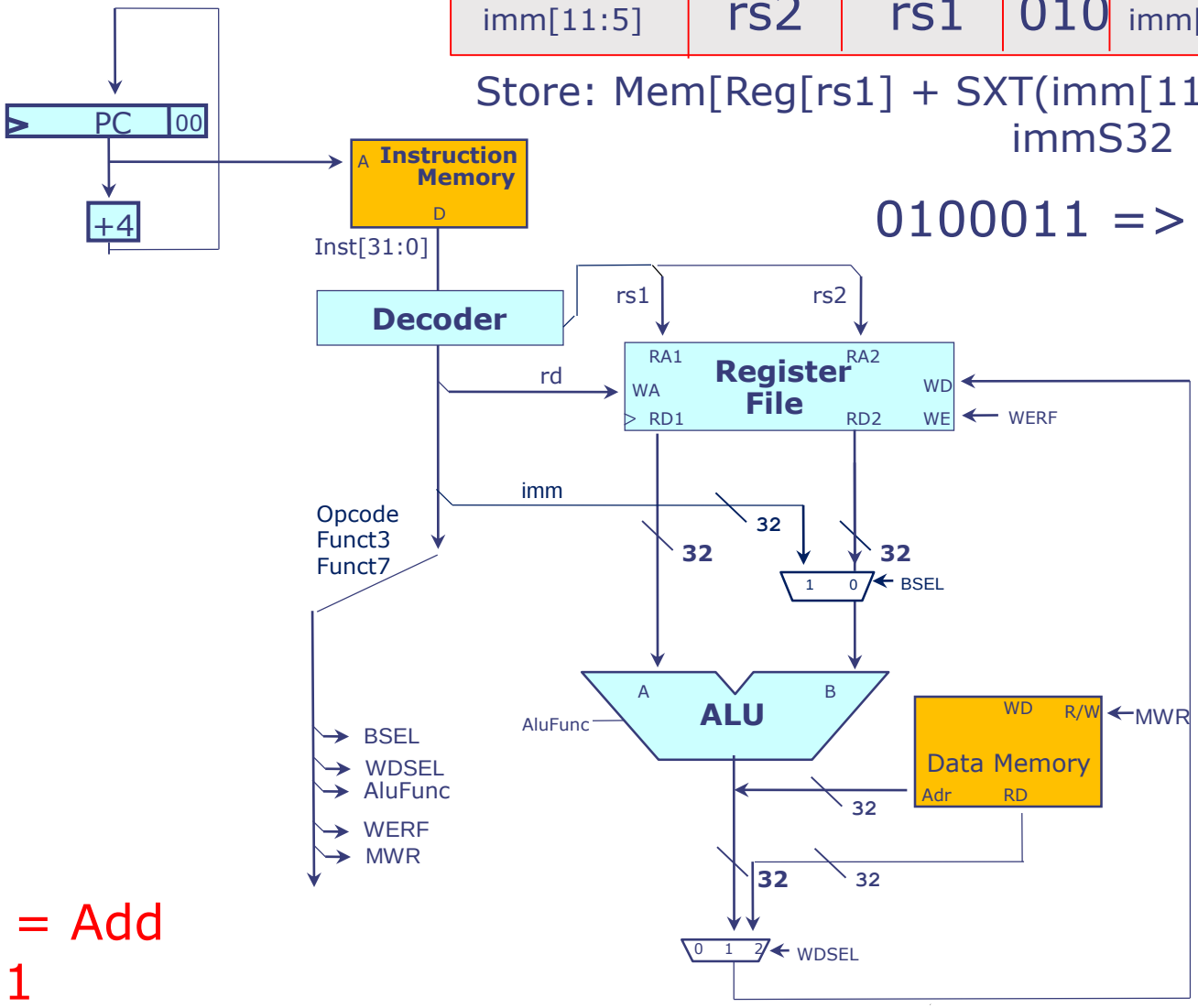


Store Instruction



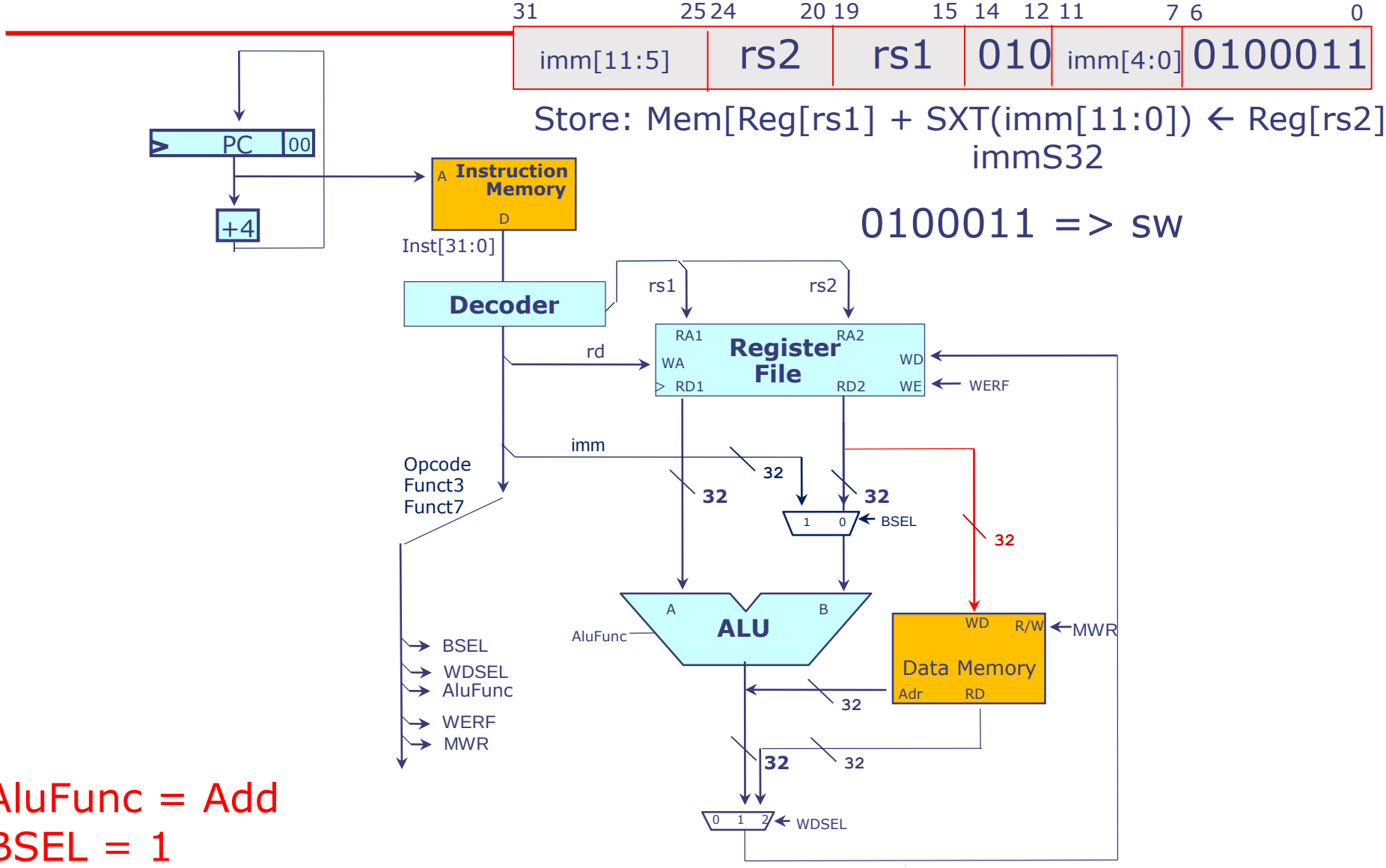
Store: Mem[Reg[rs1] + SXT(imm[11:0]) ← Reg[rs2]
immS32

0100011 => sw



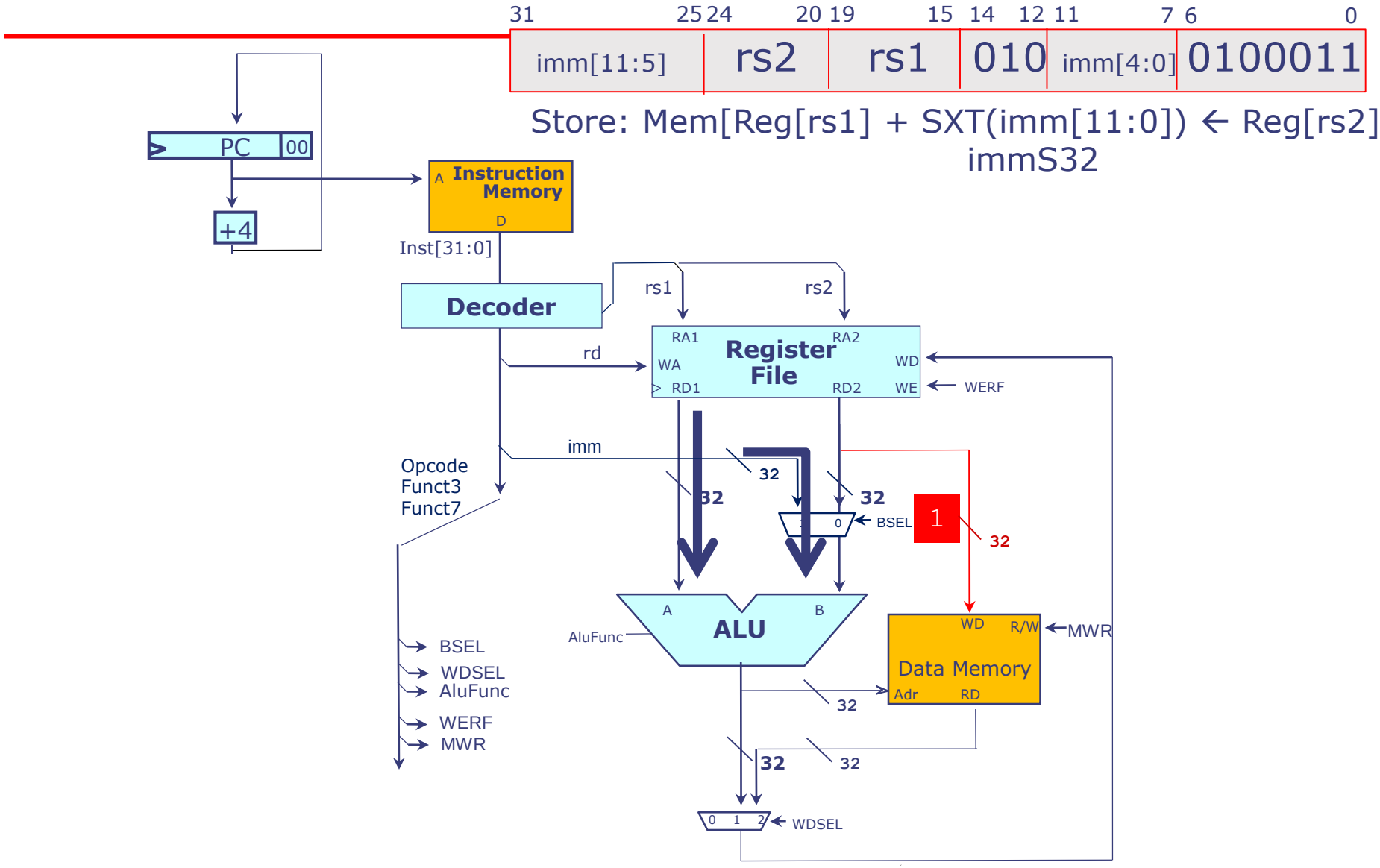
AluFunc = Add
BSEL = 1

Store Instruction

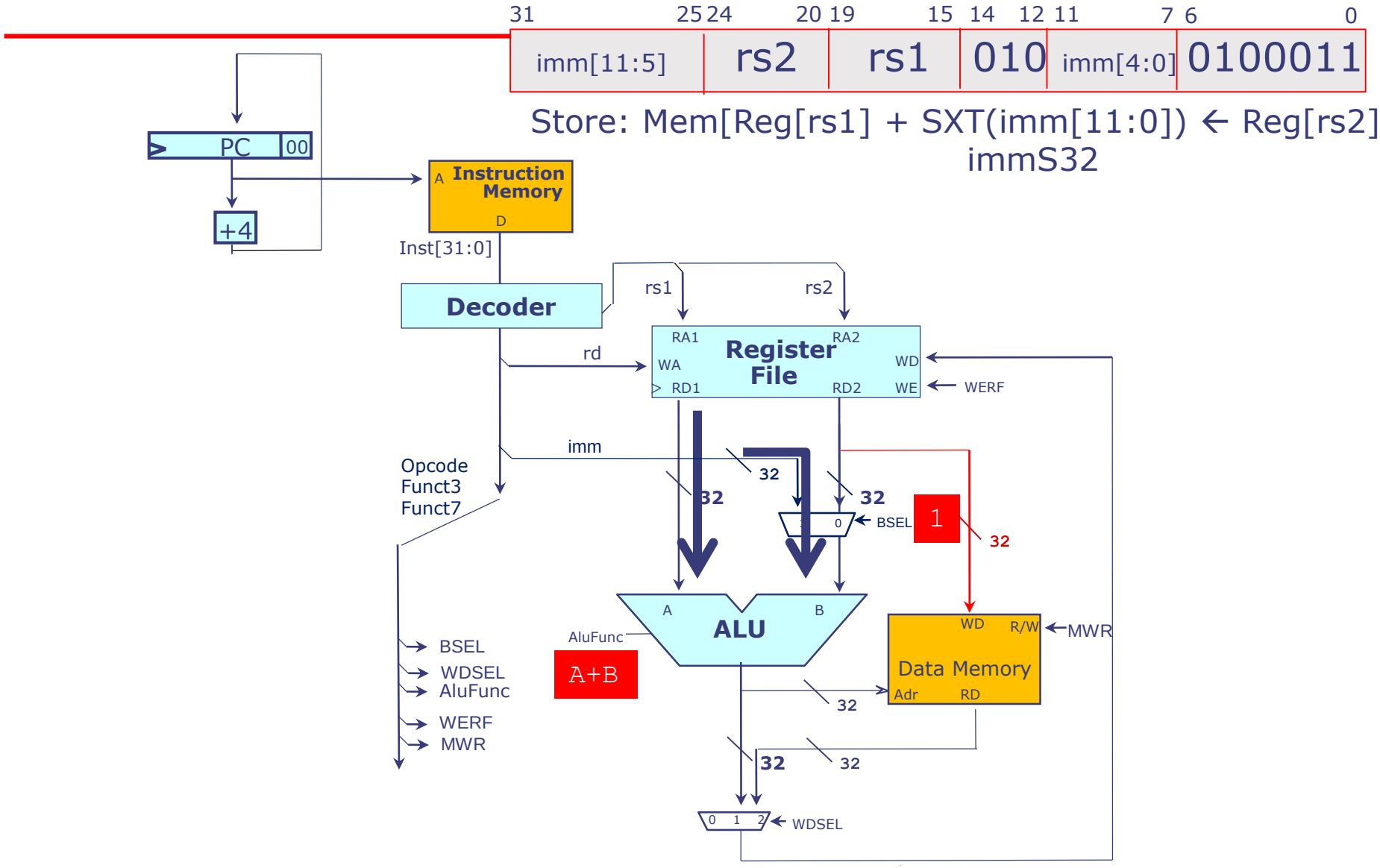


AluFunc = Add
BSEL = 1

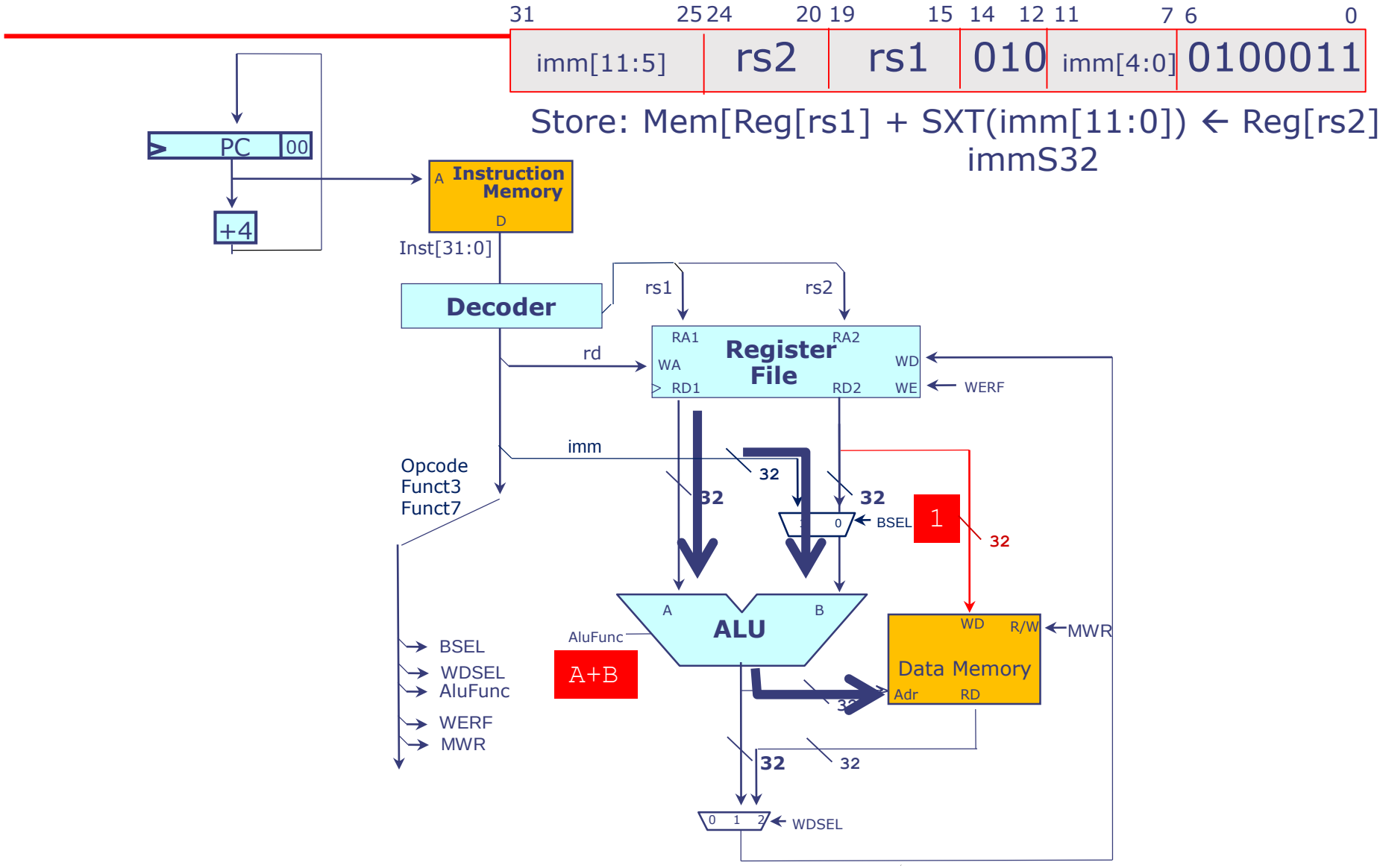
Store Instruction



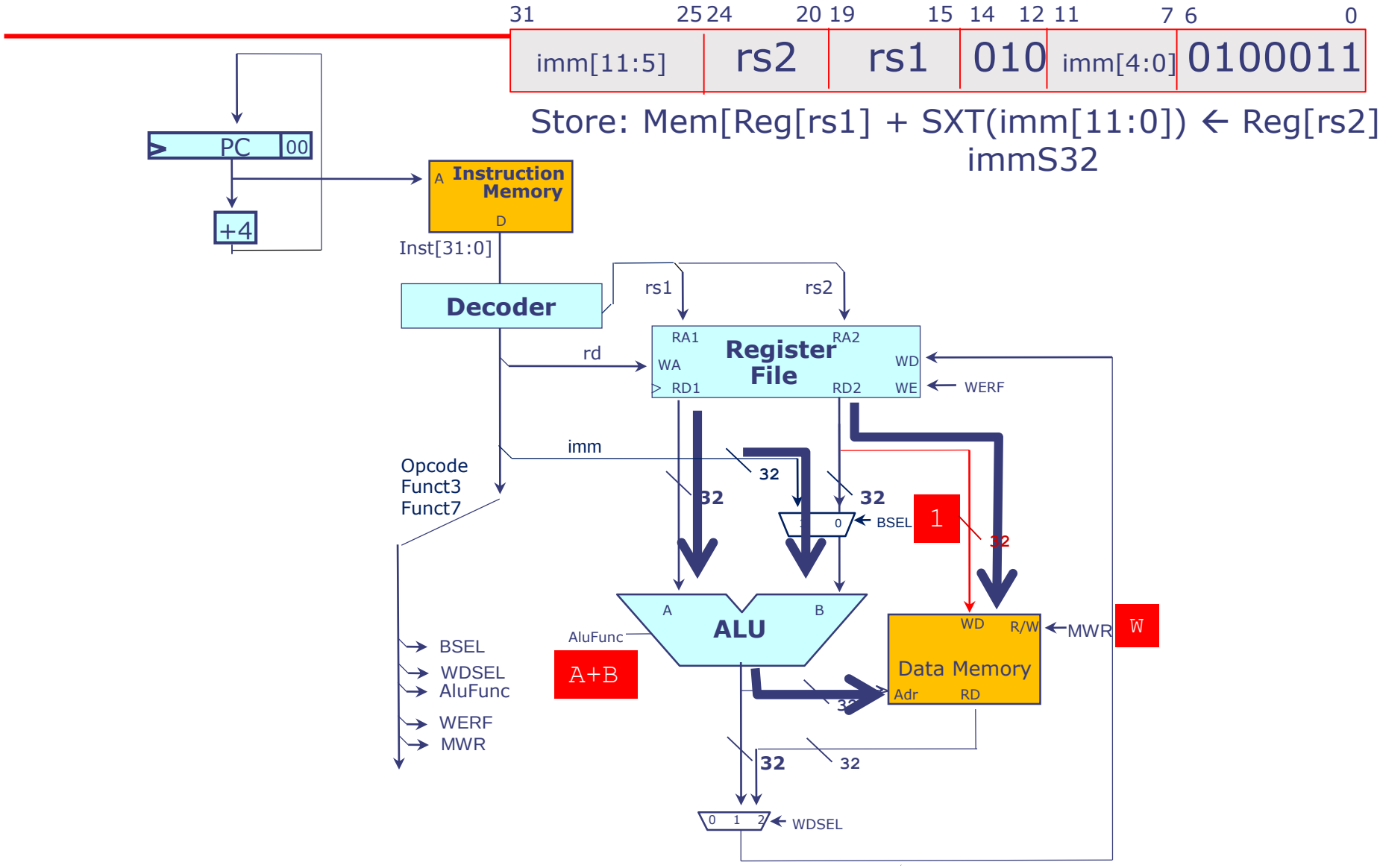
Store Instruction



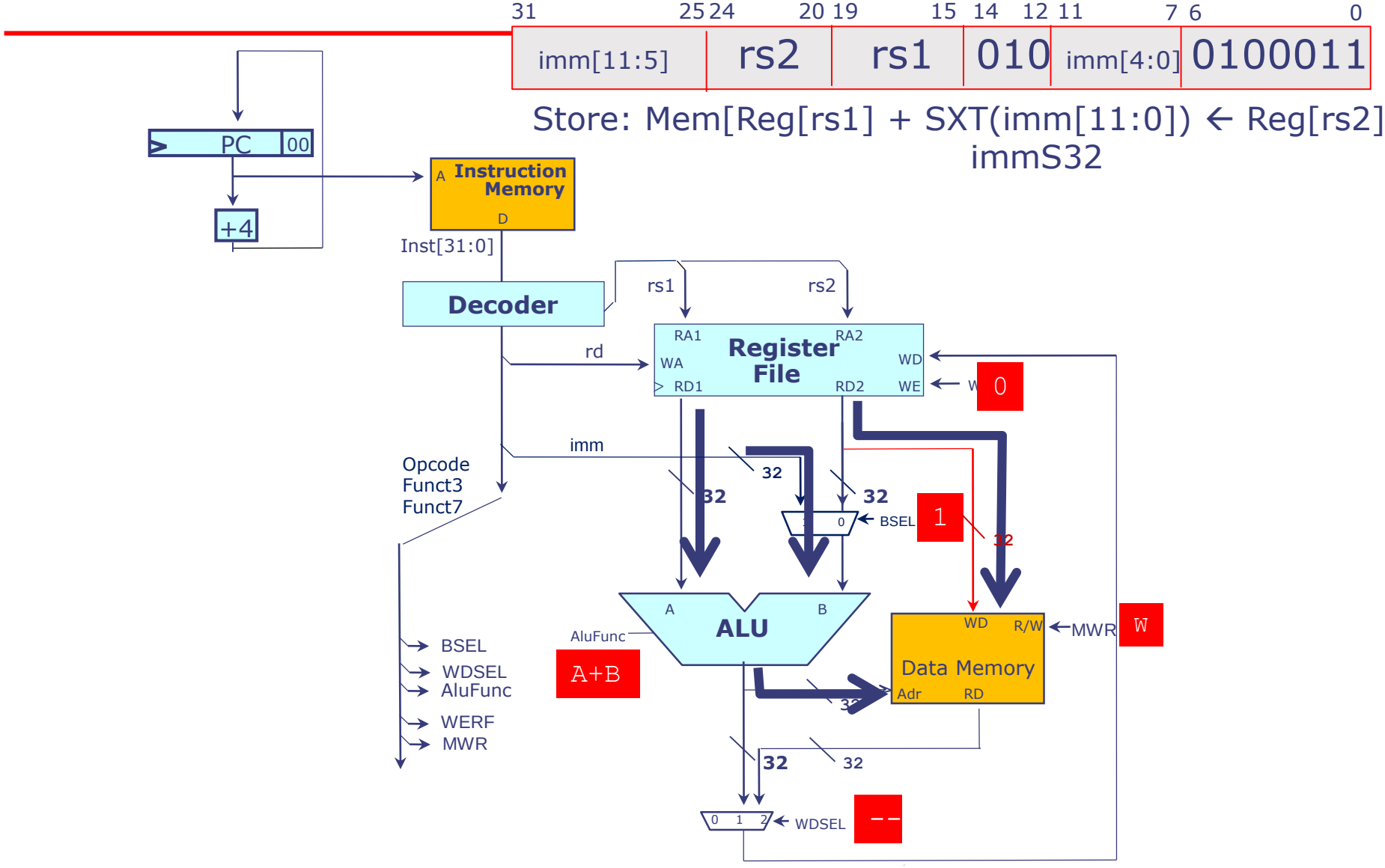
Store Instruction



Store Instruction



Store Instruction



Branch Instructions

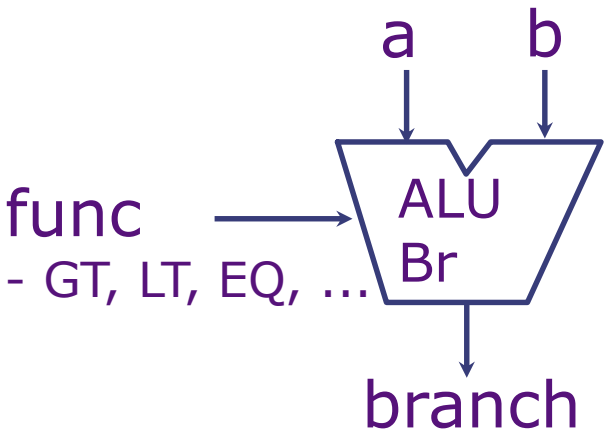
differ only in the aluBr operation they perform

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

These instructions are grouped in a category called BRANCH with fields (brFunc, rs1, rs2, immB32)

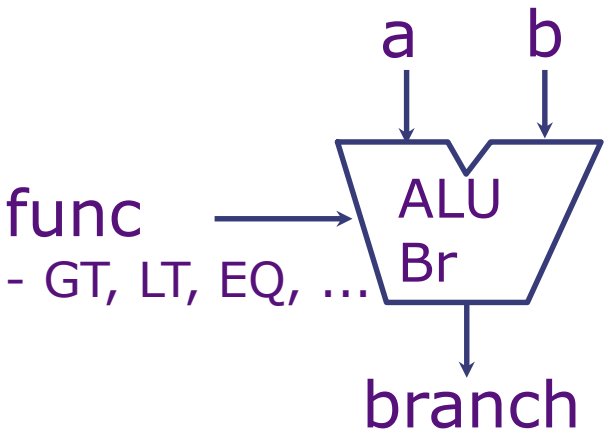
ALU for Branch Comparisons

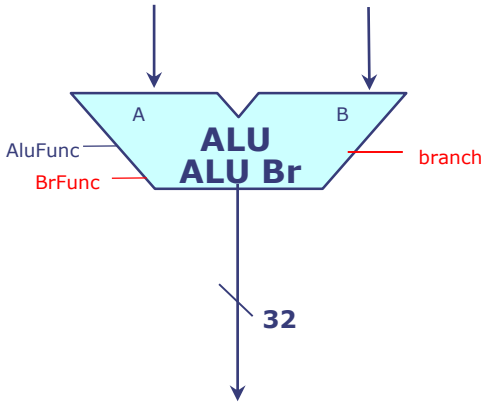
Like ALU, but
returns a Bool



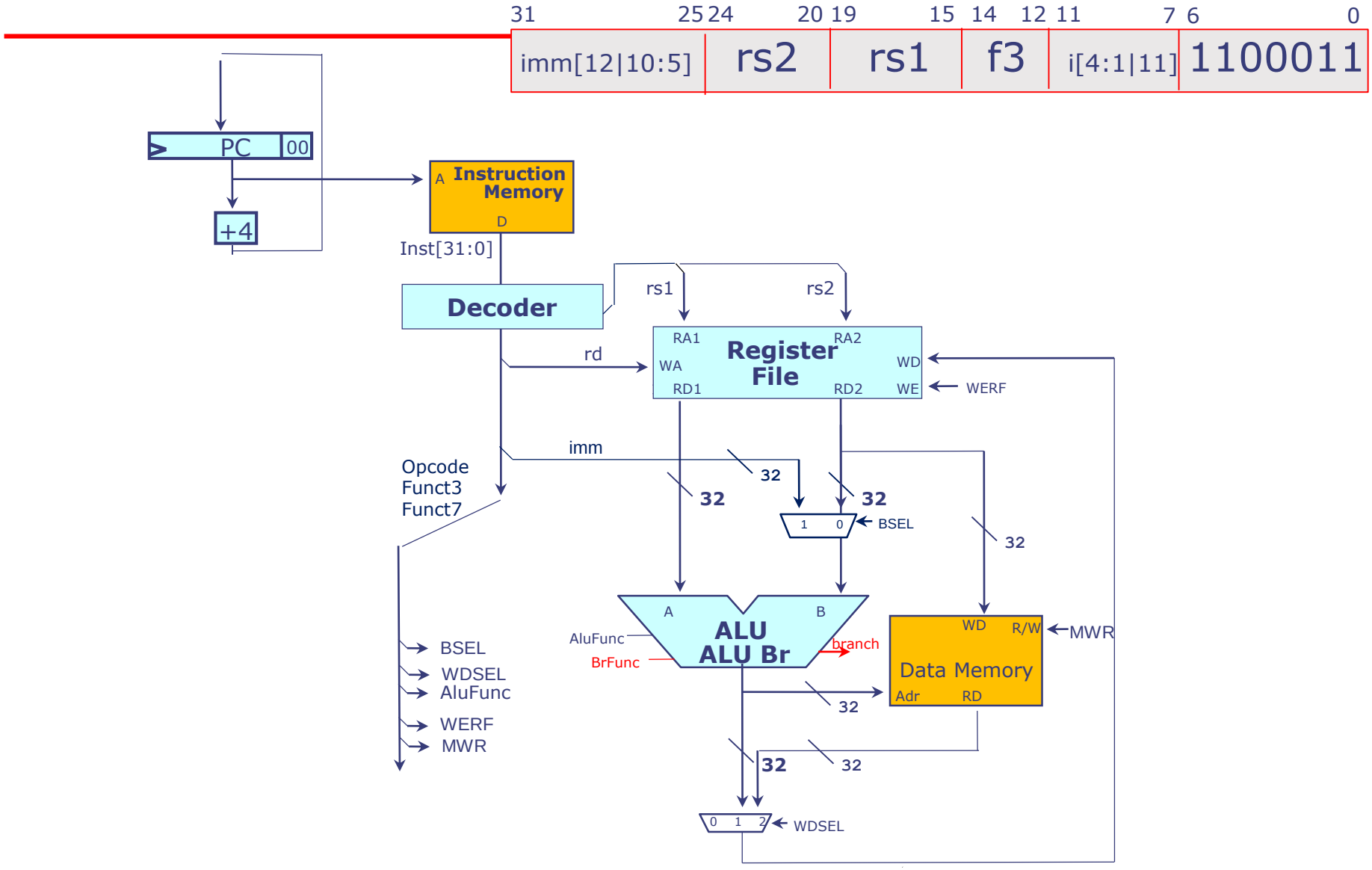
ALU for Branch Comparisons

Like ALU, but returns a Bool

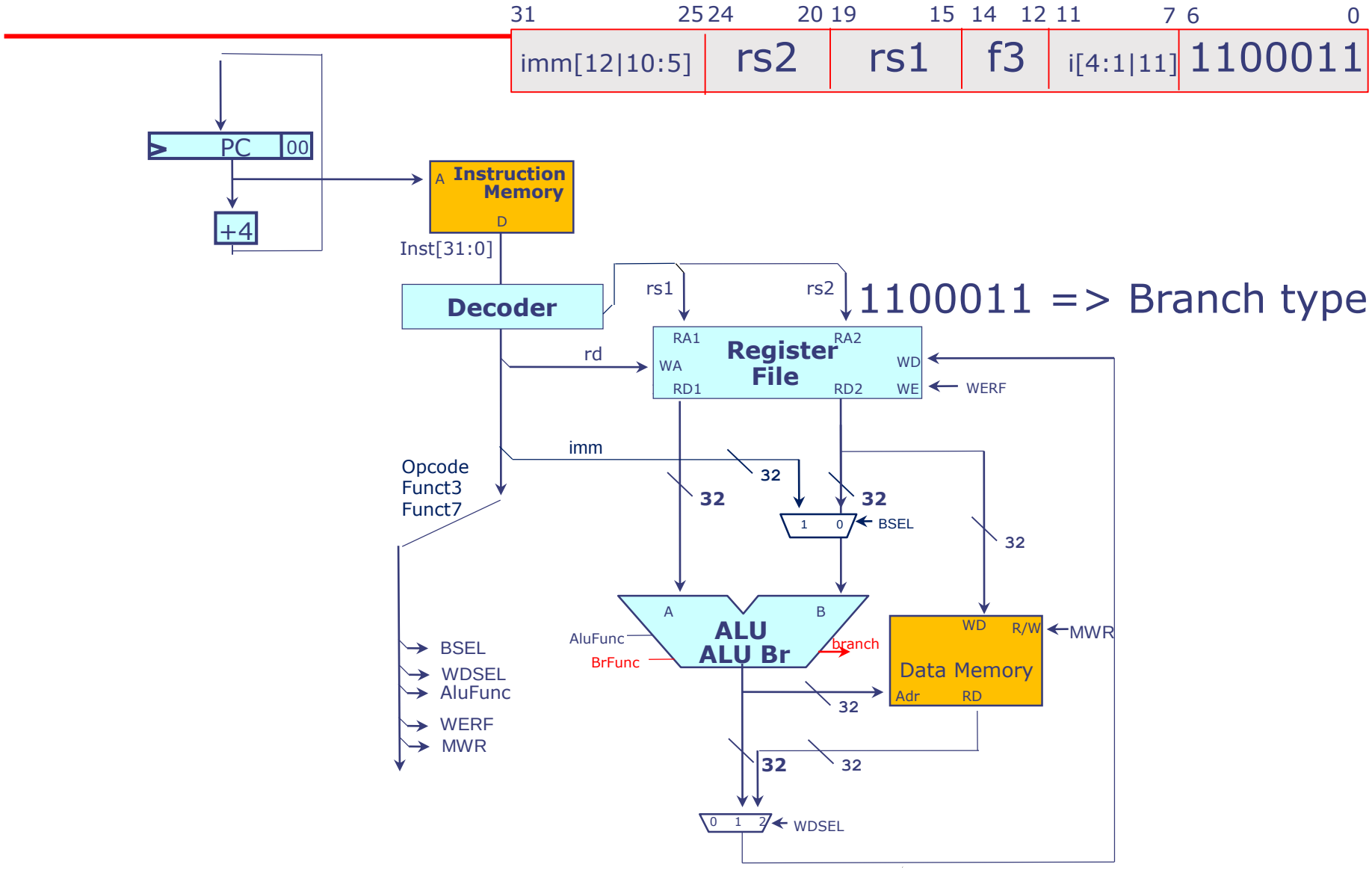




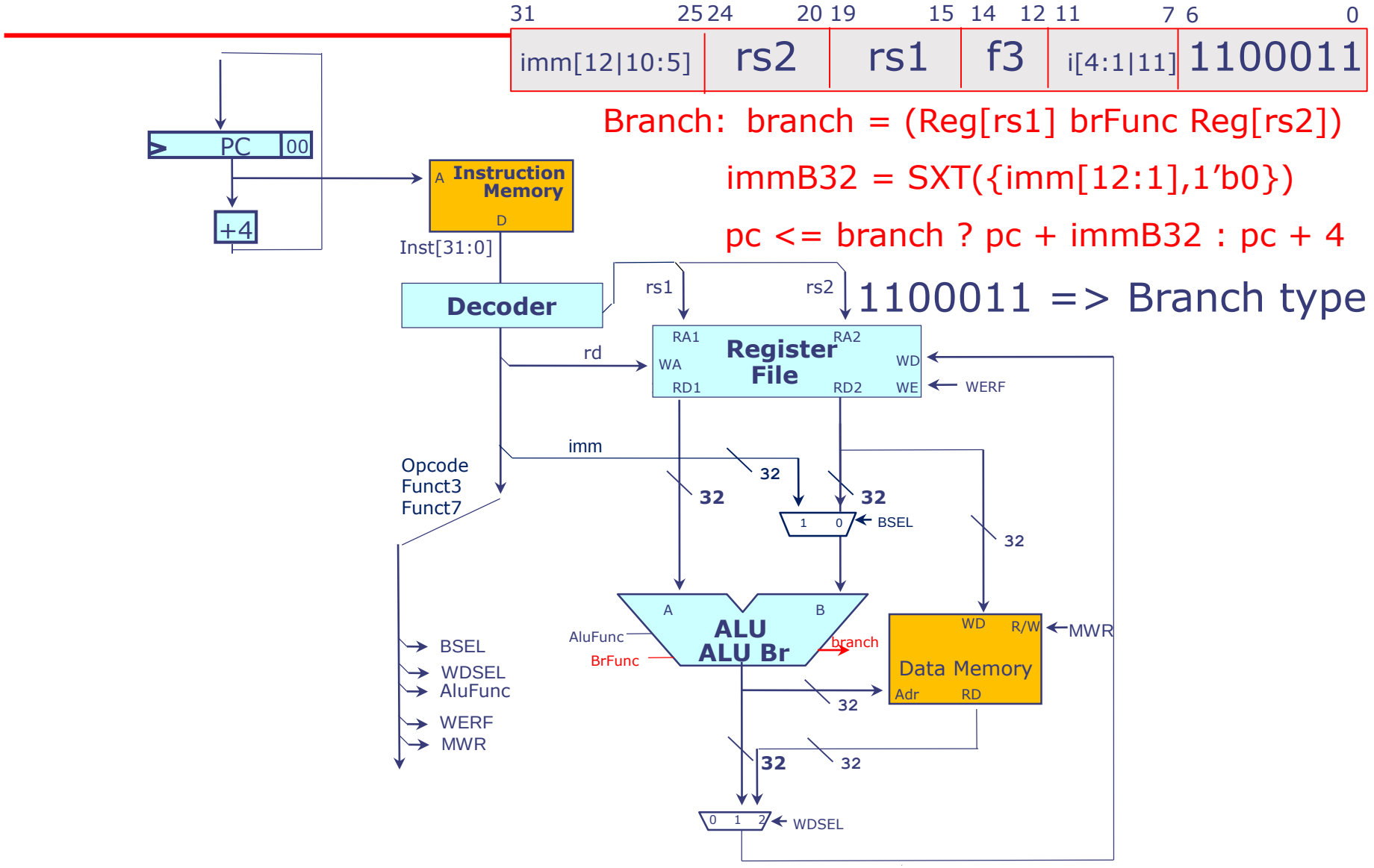
Branch Instructions



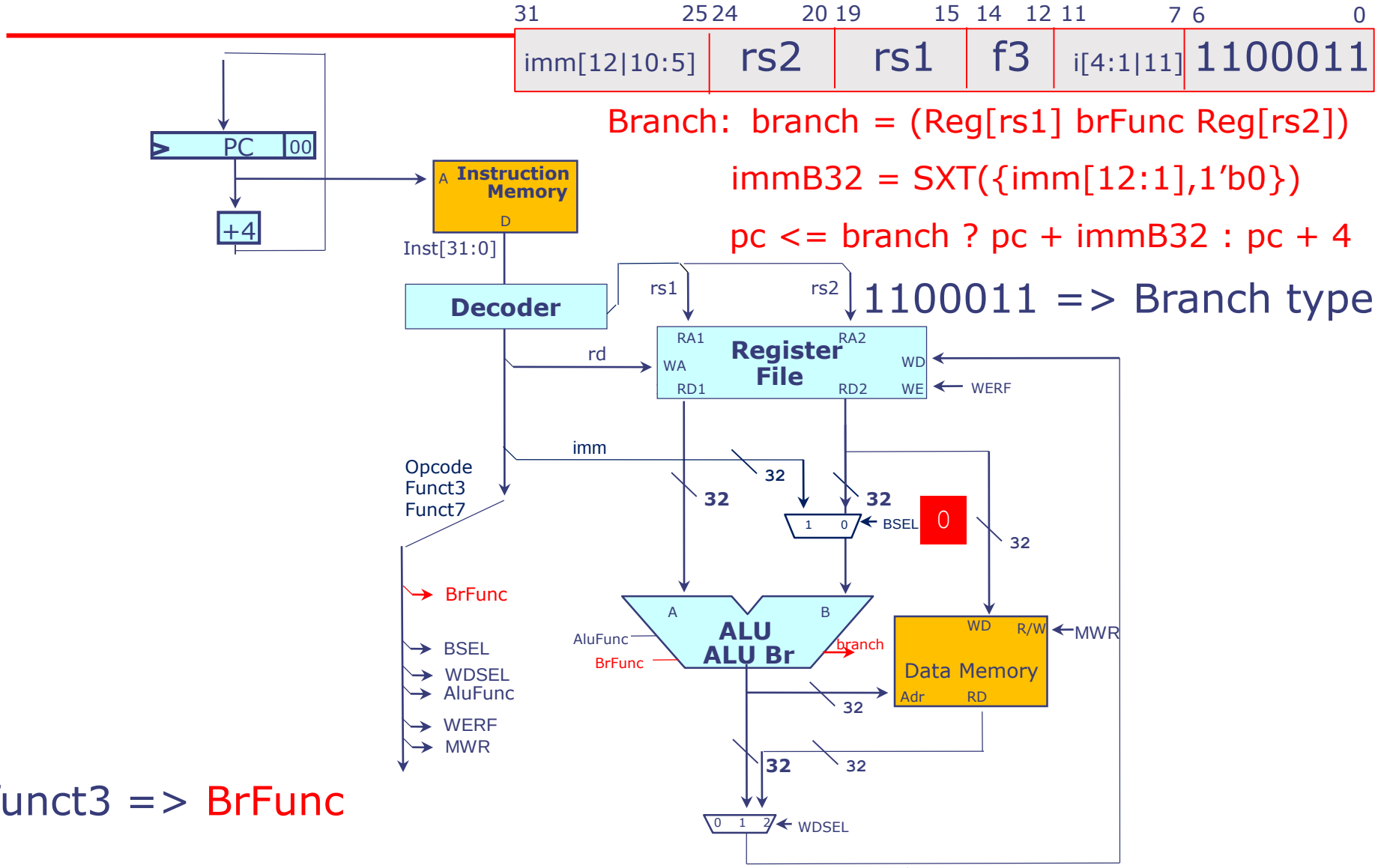
Branch Instructions



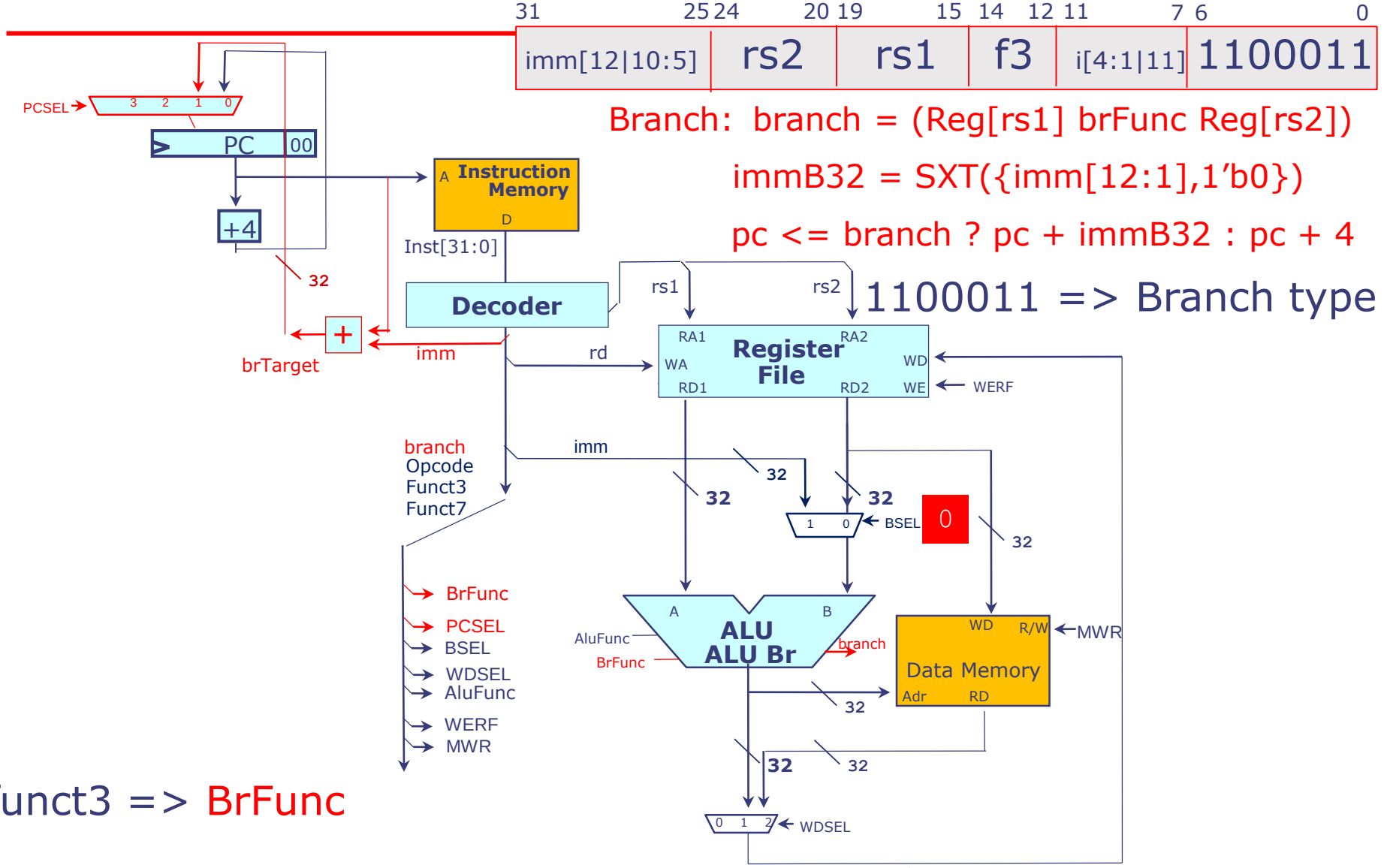
Branch Instructions



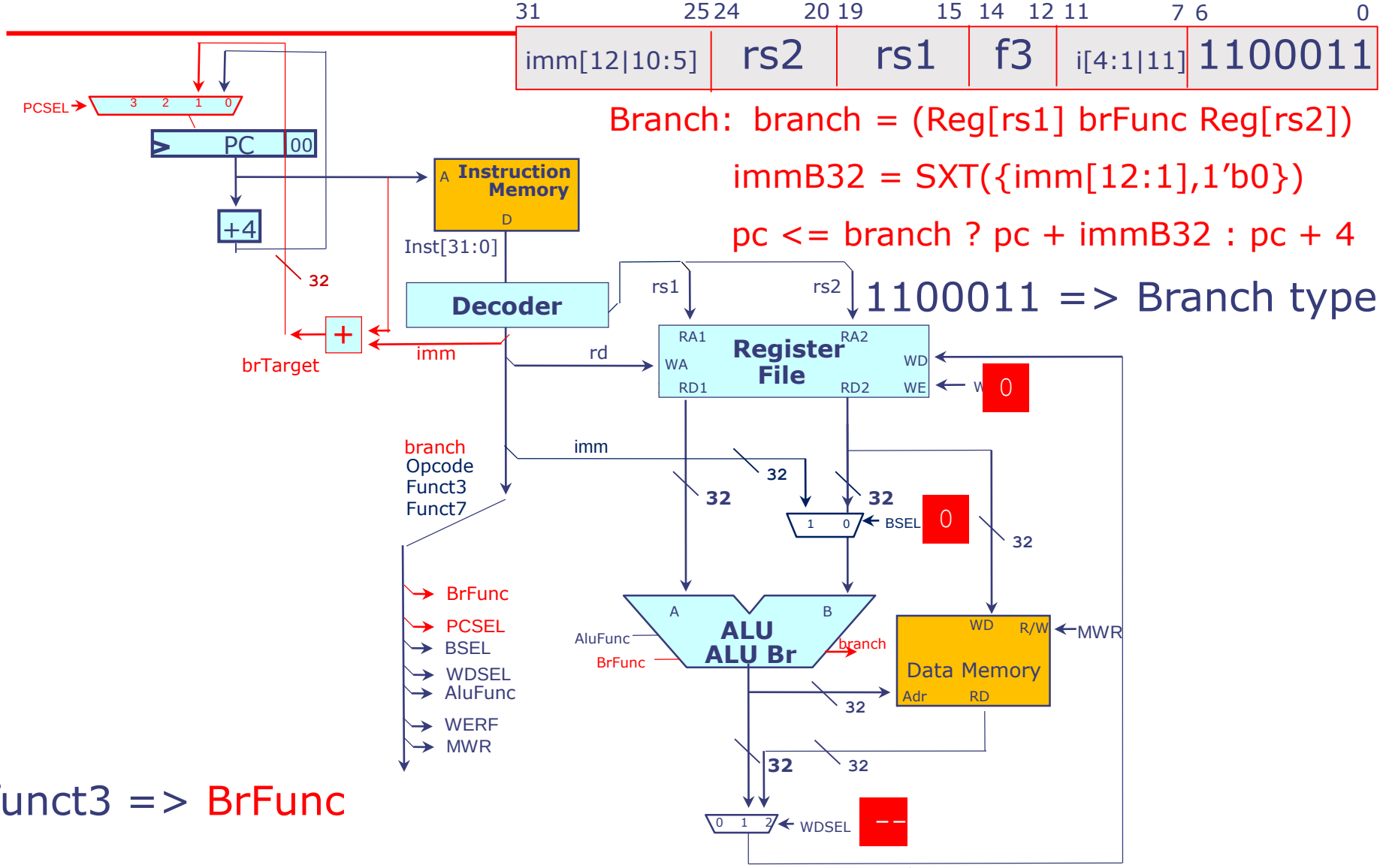
Branch Instructions



Branch Instructions



Branch Instructions



Remaining Instructions

Instruction	Description	Execution
JAL rd, label	Jump and Link	reg[rd] <= pc + 4 pc <= label
JALR rd, offset(rs1)	Jump and Link Register	reg[rd] <= pc + 4 pc <= {(reg[rs1] + offset)[31:1], 1'b0}
LUI rd, luiConstant	Load Upper Immediate	reg[rd] <= luiConstant << 12

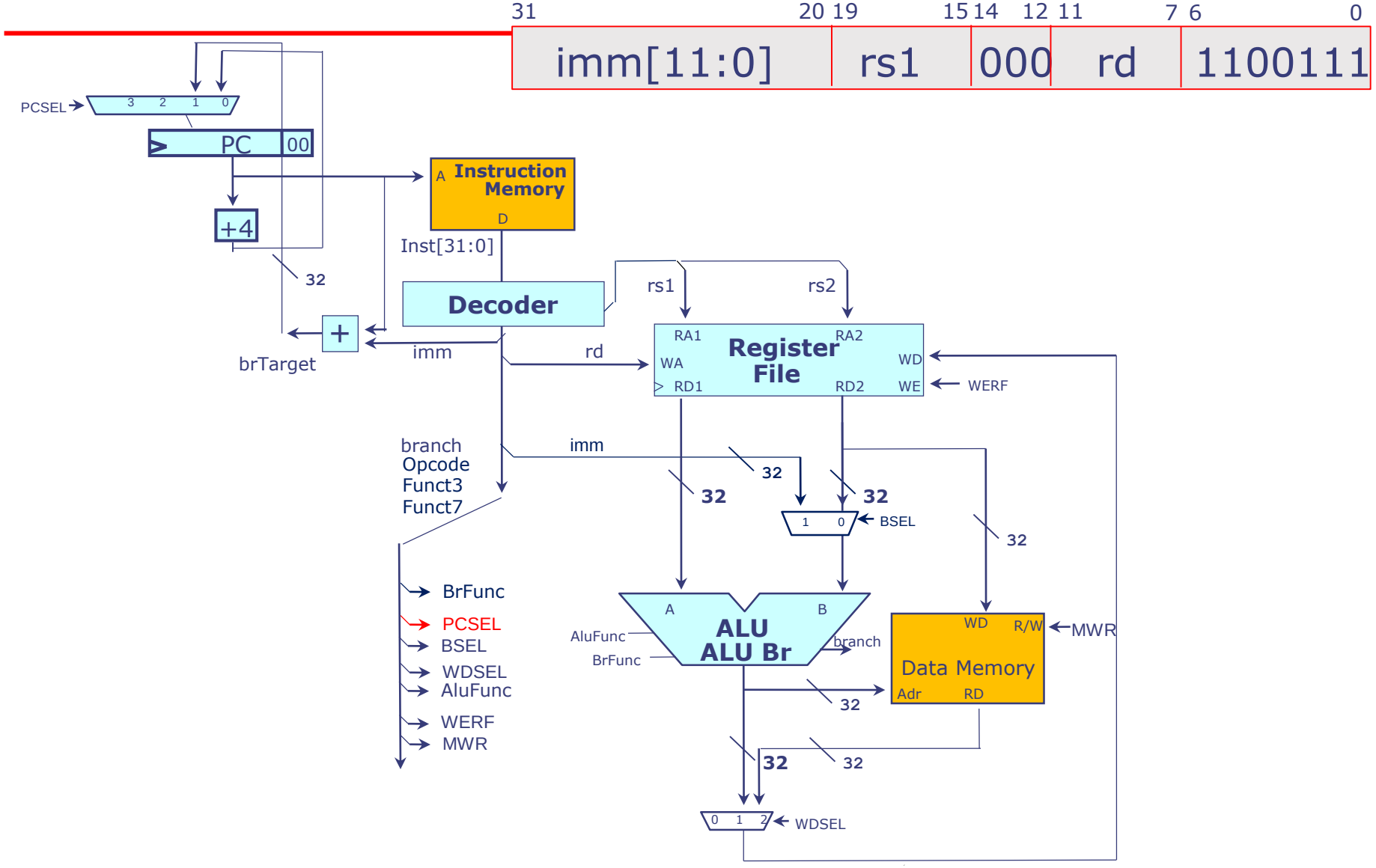
Each of these instructions is in a category by itself and needs to extract different fields from the instruction.

Remaining Instructions

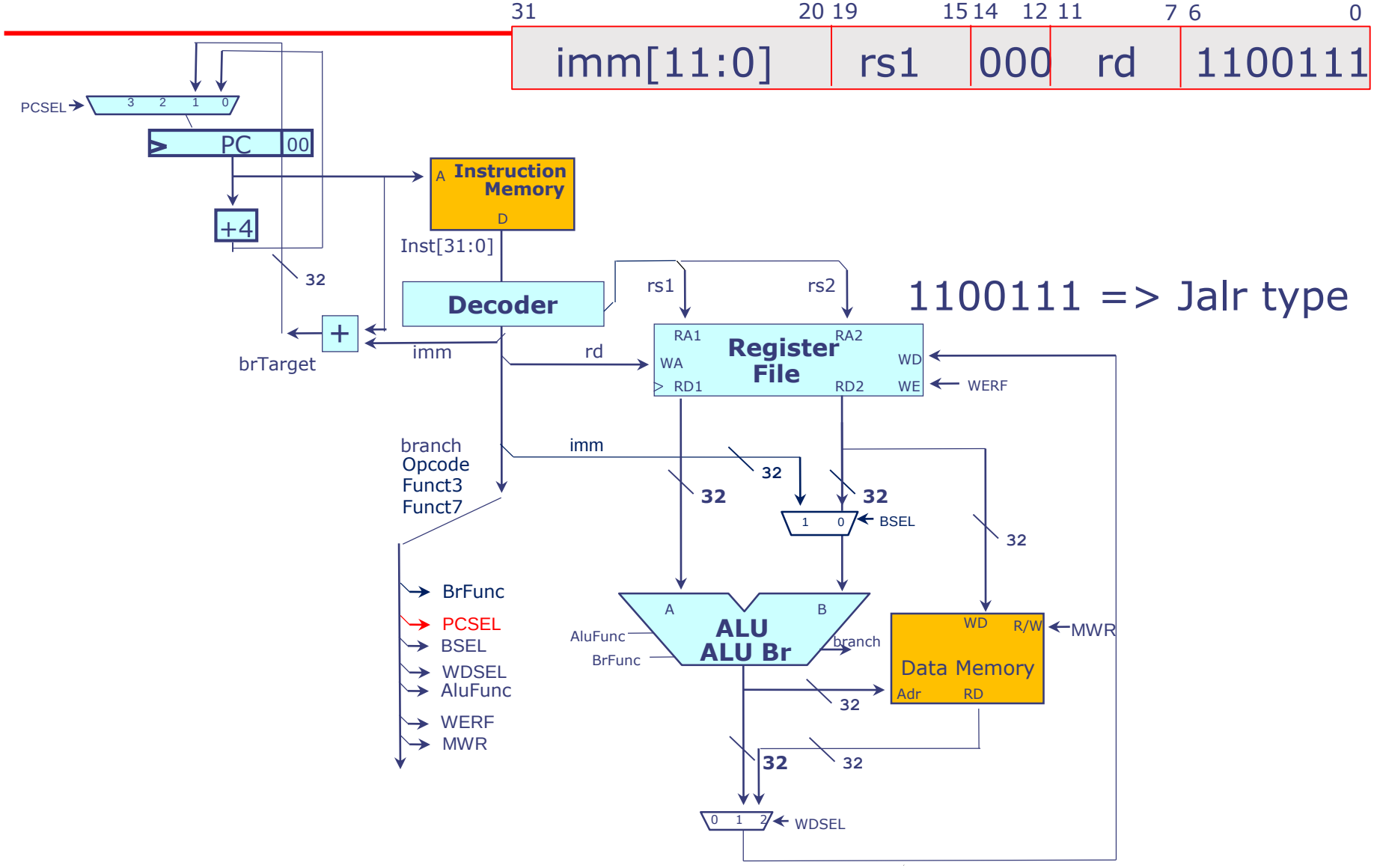
Instruction	Description	Execution
JAL rd, label	Jump and Link	reg[rd] <= pc + 4 pc <= label
JALR rd, offset(rs1)	Jump and Link Register	reg[rd] <= pc + 4 pc <= {(reg[rs1] + offset)[31:1], 1'b0}
LUI rd, luiConstant	Load Upper Immediate	reg[rd] <= luiConstant << 12

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

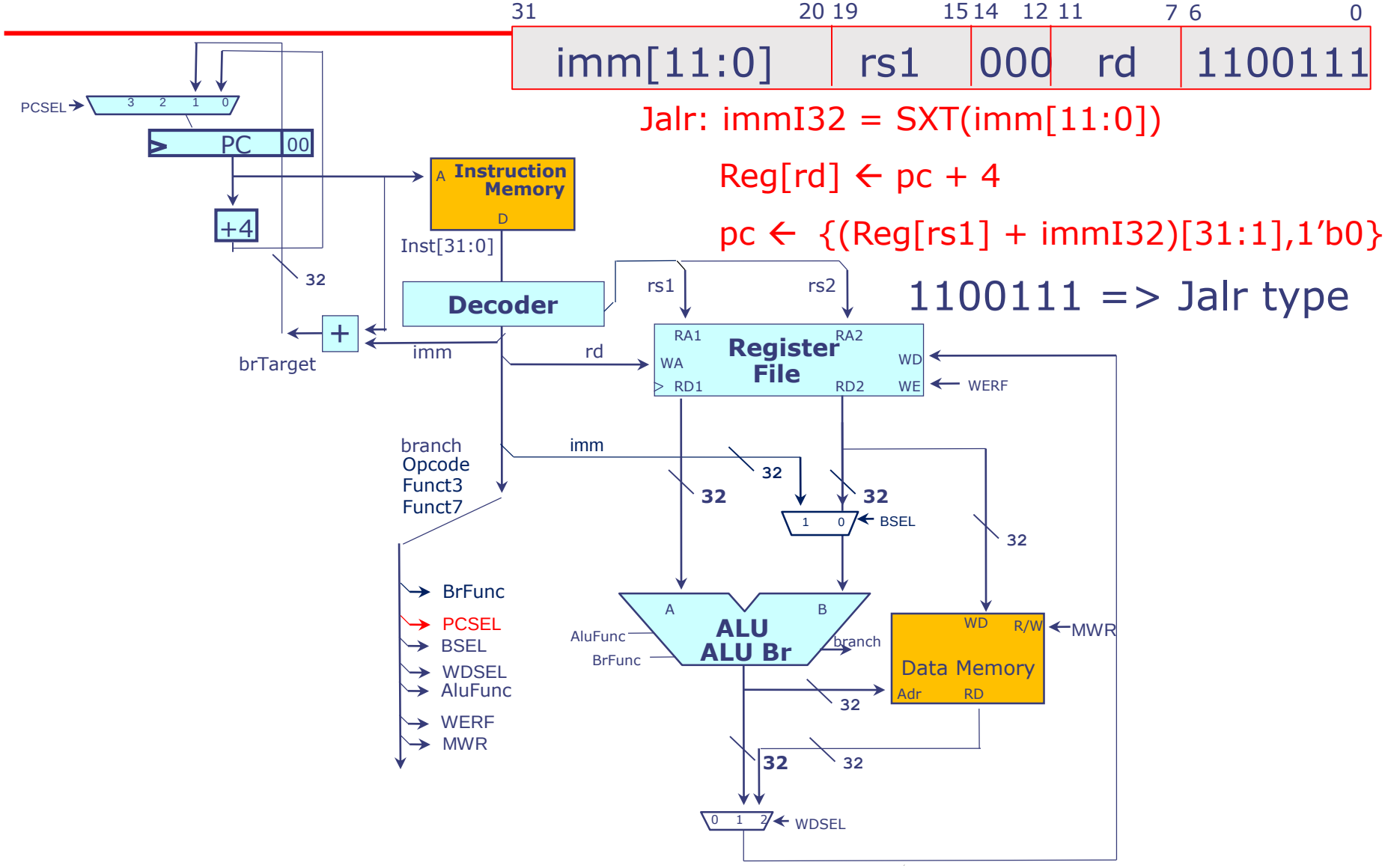
Jalr Instruction



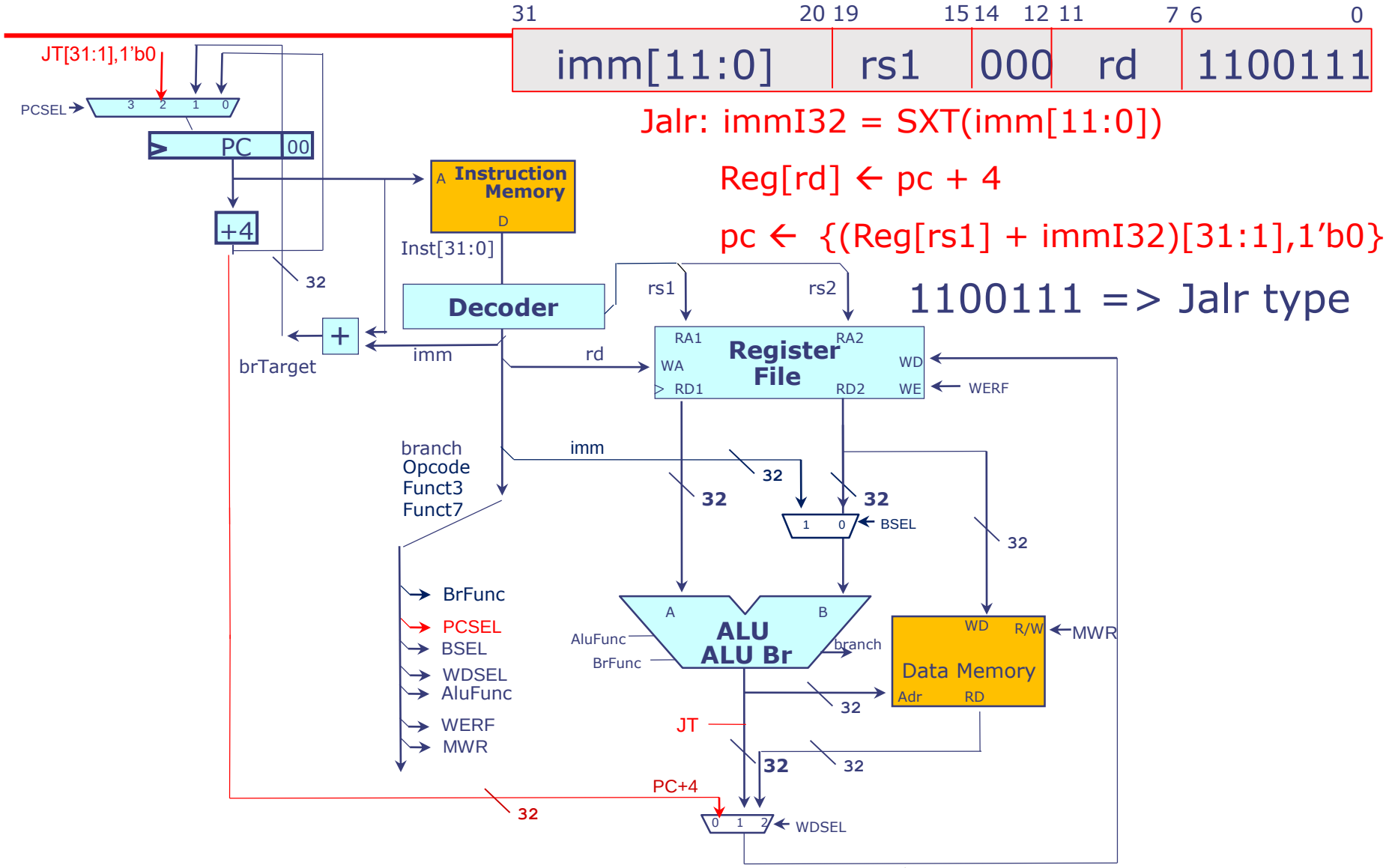
Jalr Instruction



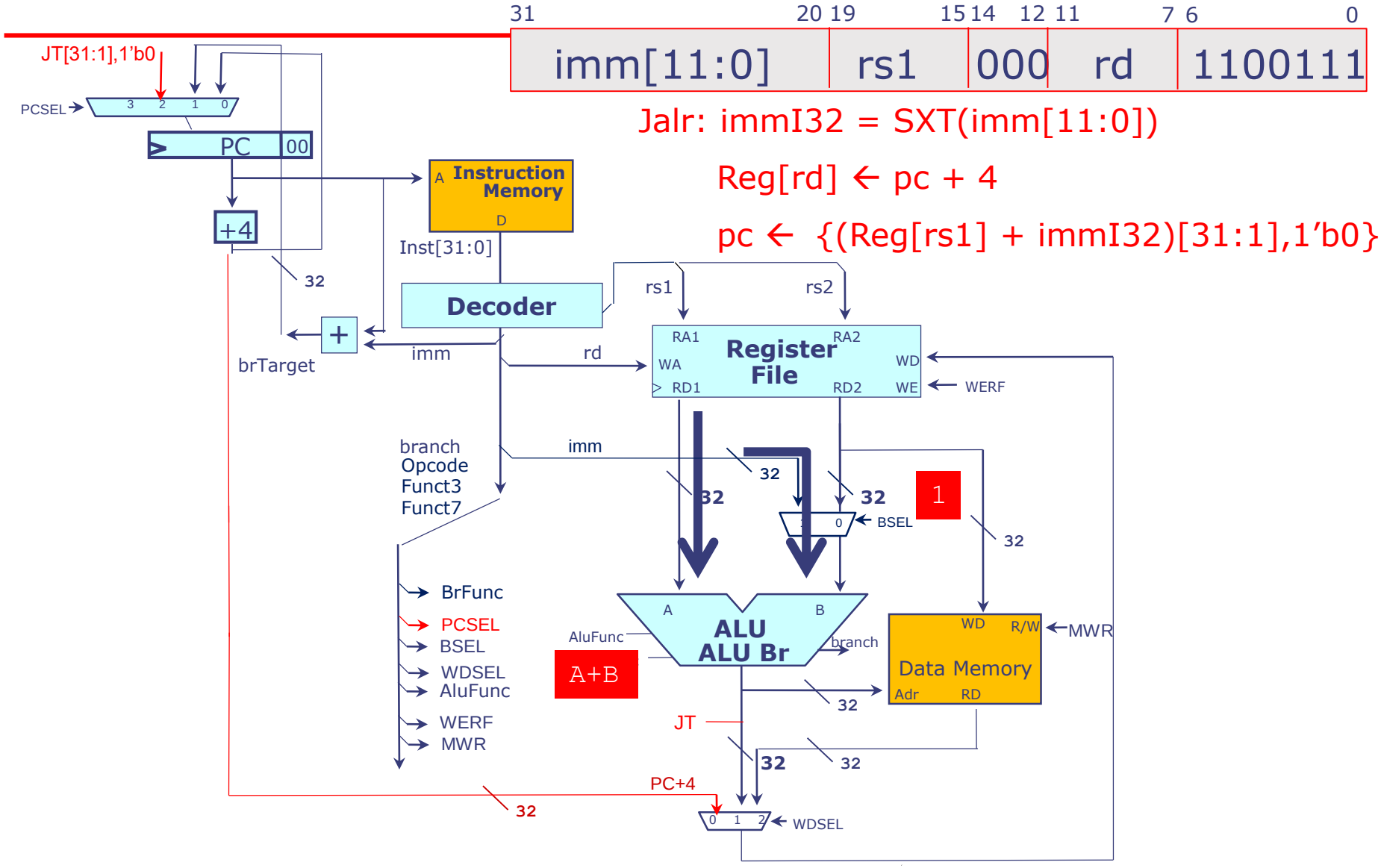
Jalr Instruction



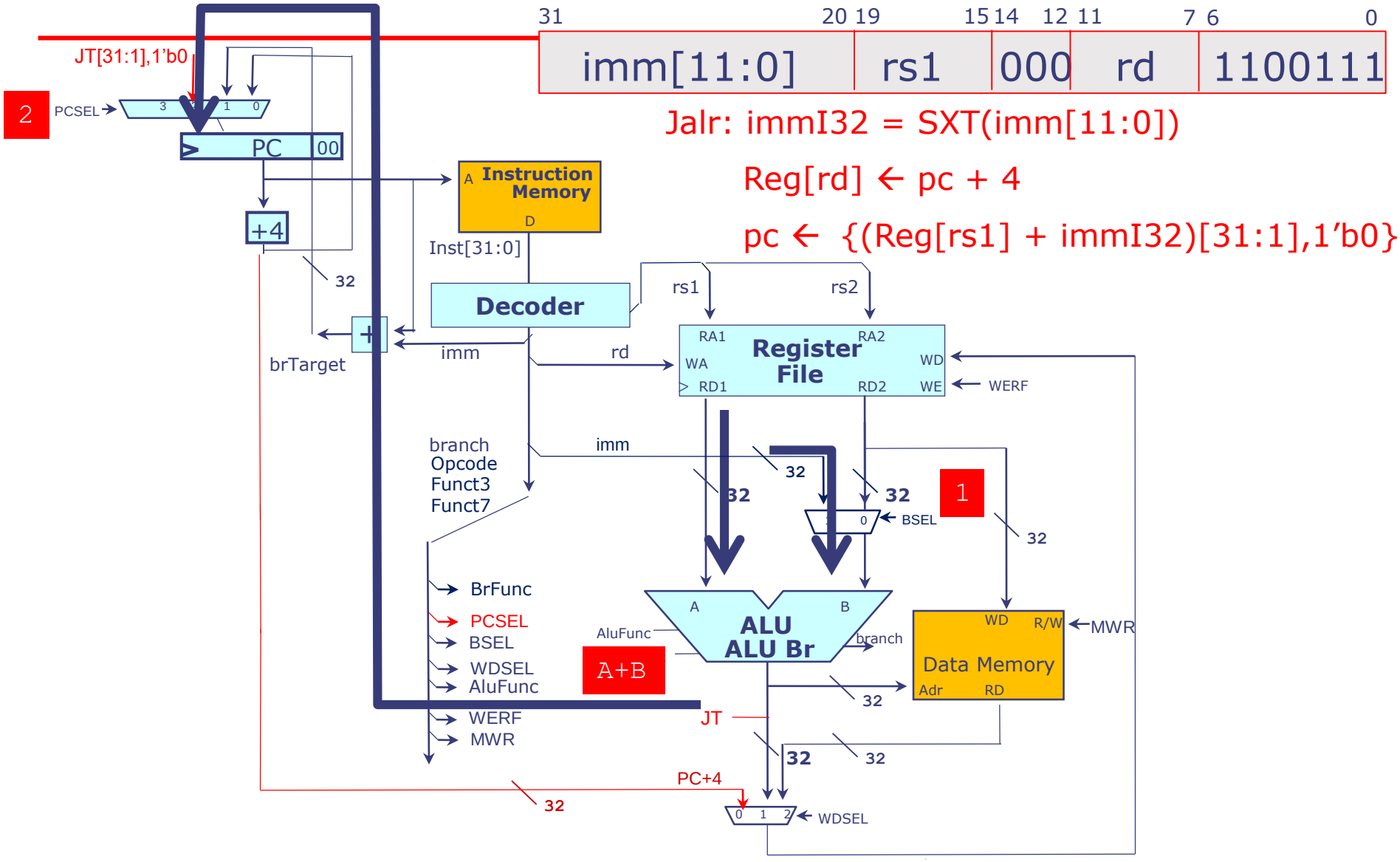
Jalr Instruction



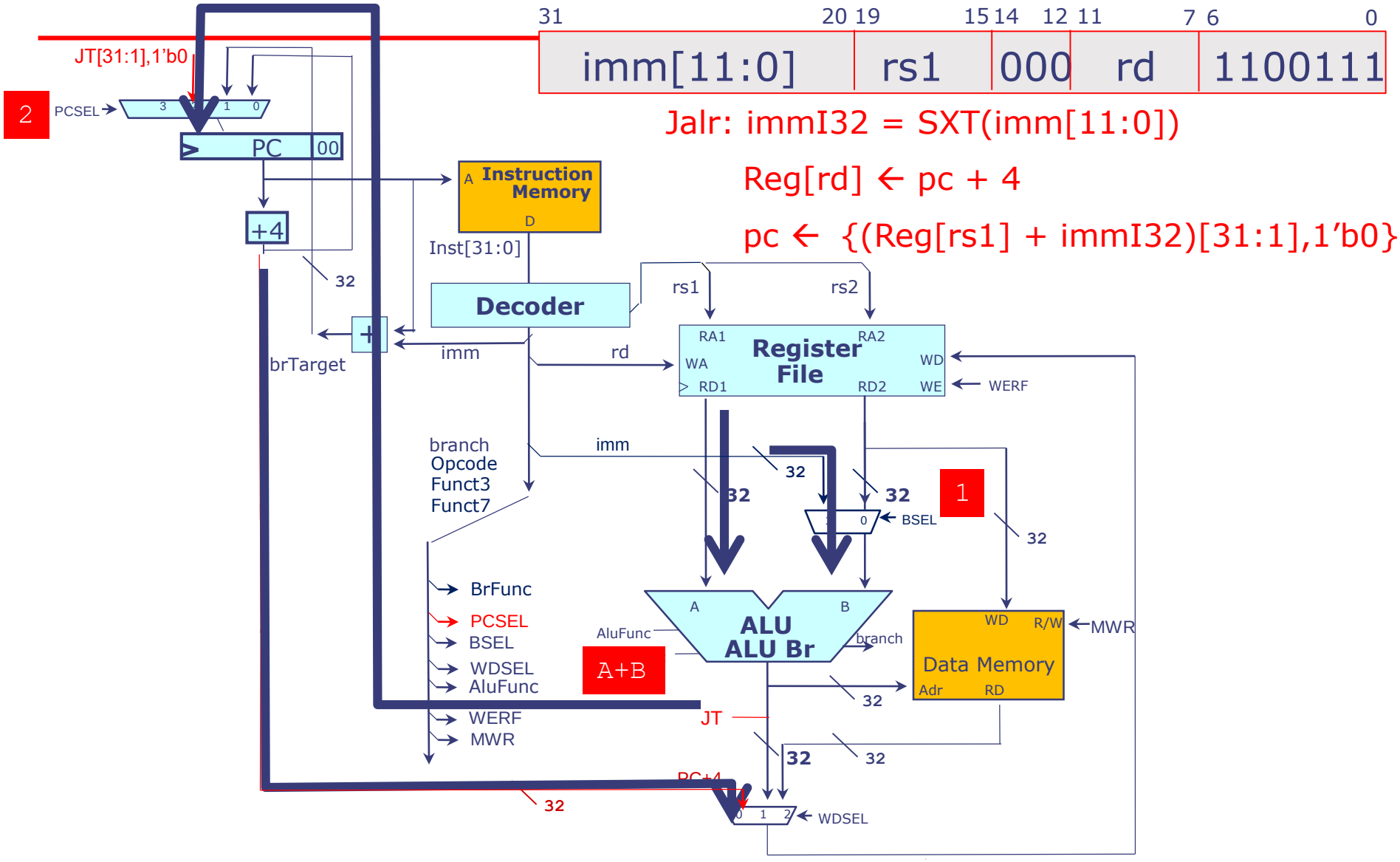
Jalr Instruction



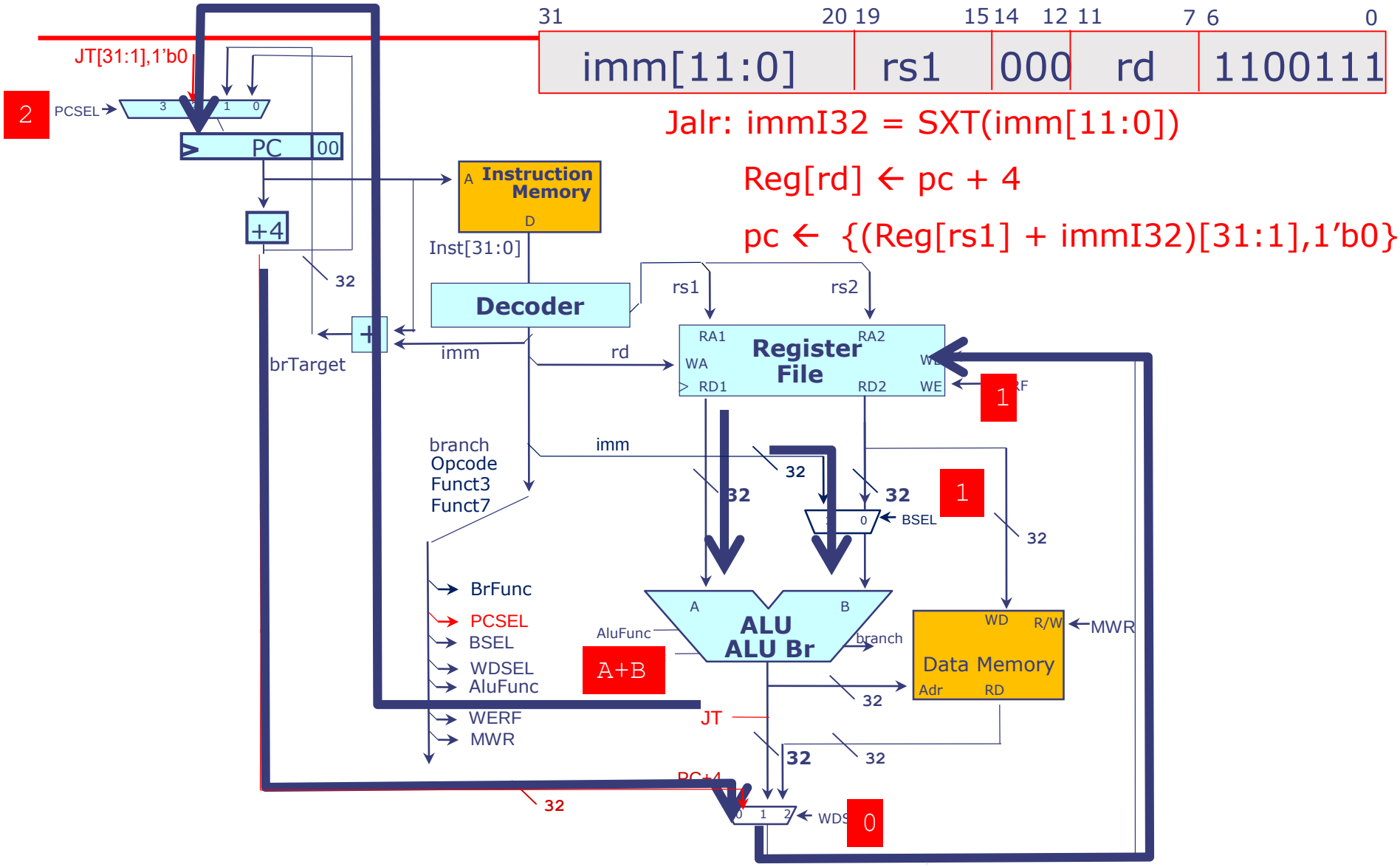
Jalr Instruction



Jalr Instruction

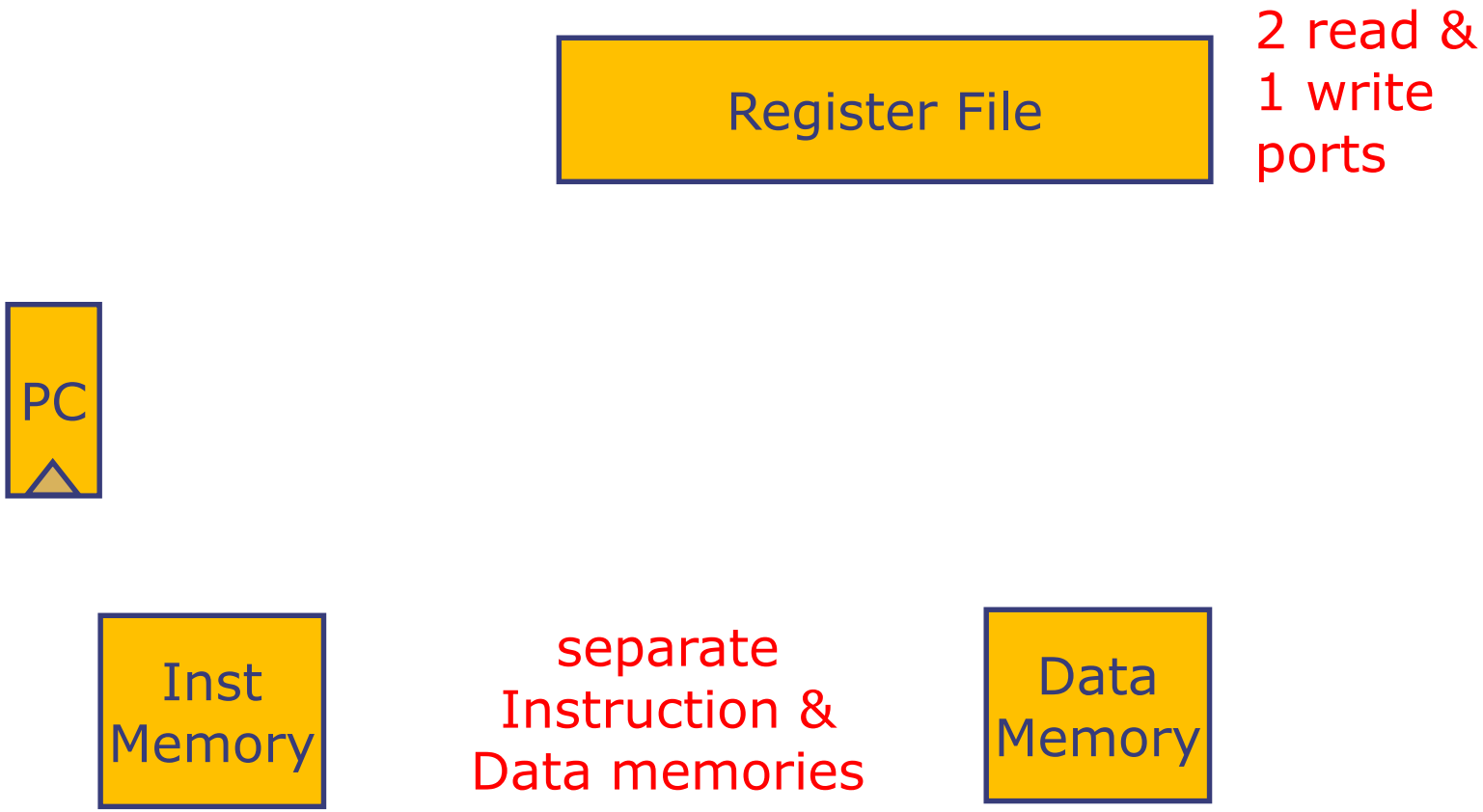


Jalr Instruction

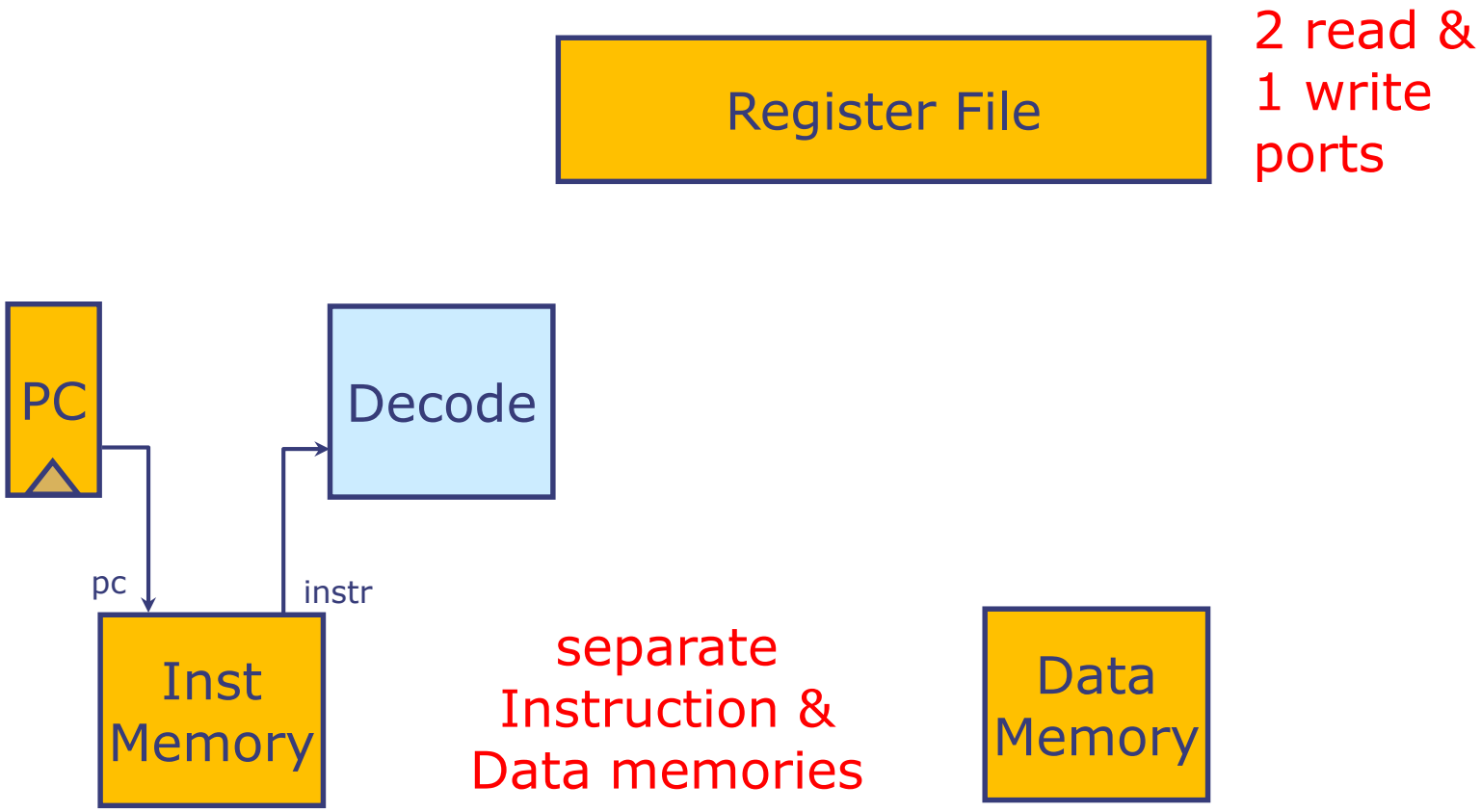


Single-Cycle RISC-V Processor

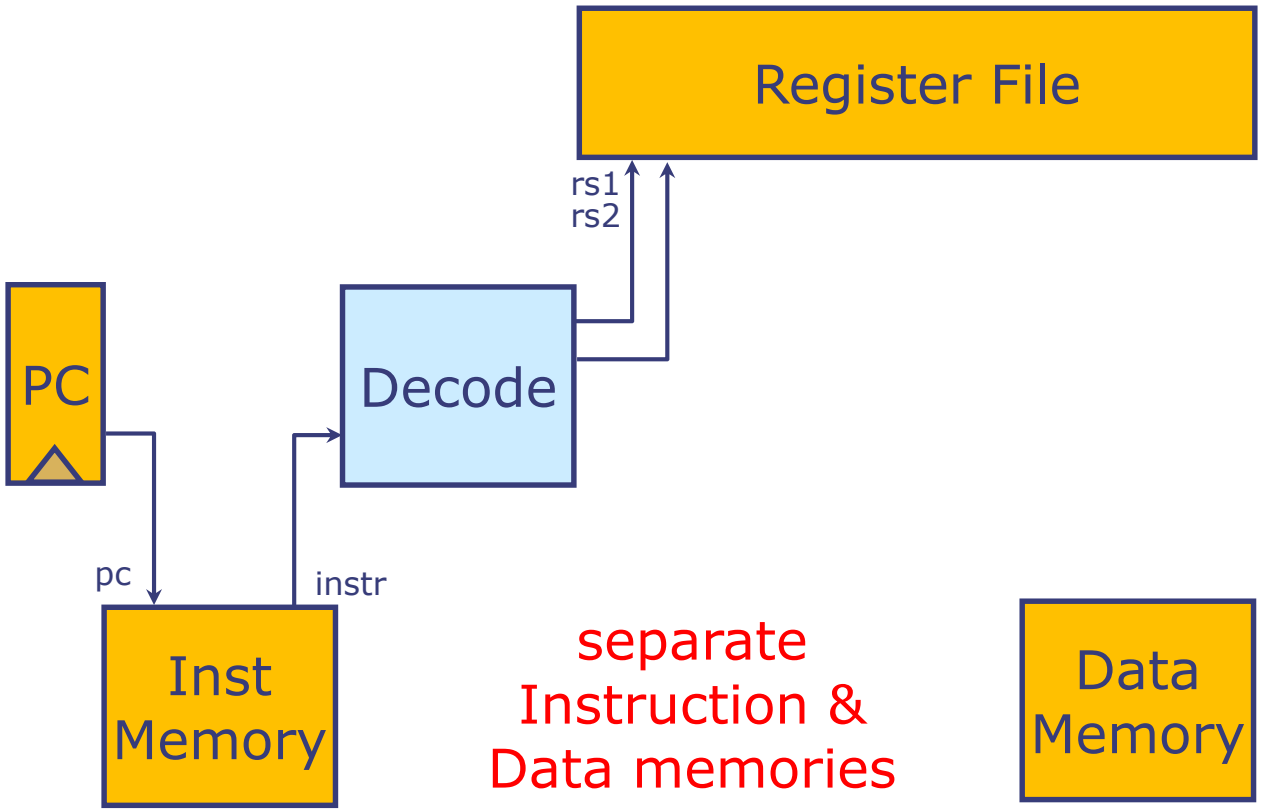
Single-Cycle RISC-V Processor



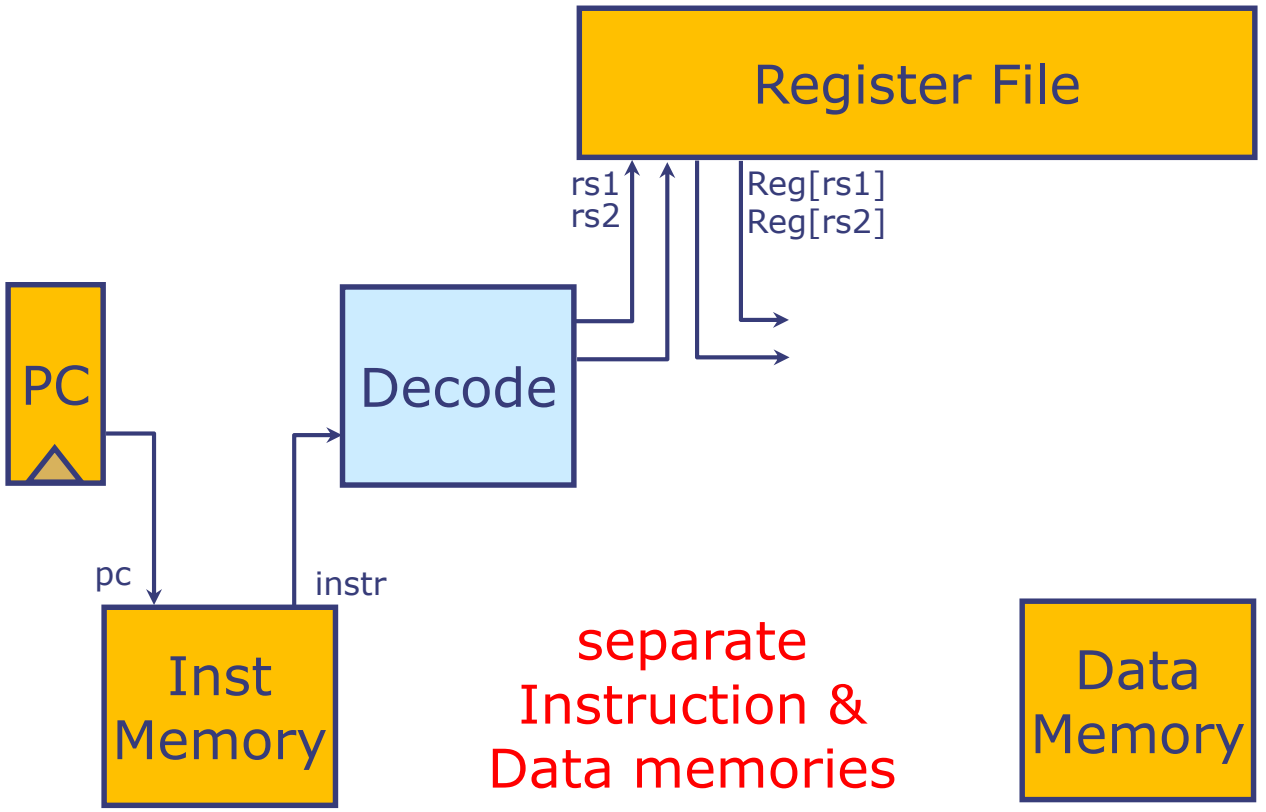
Single-Cycle RISC-V Processor



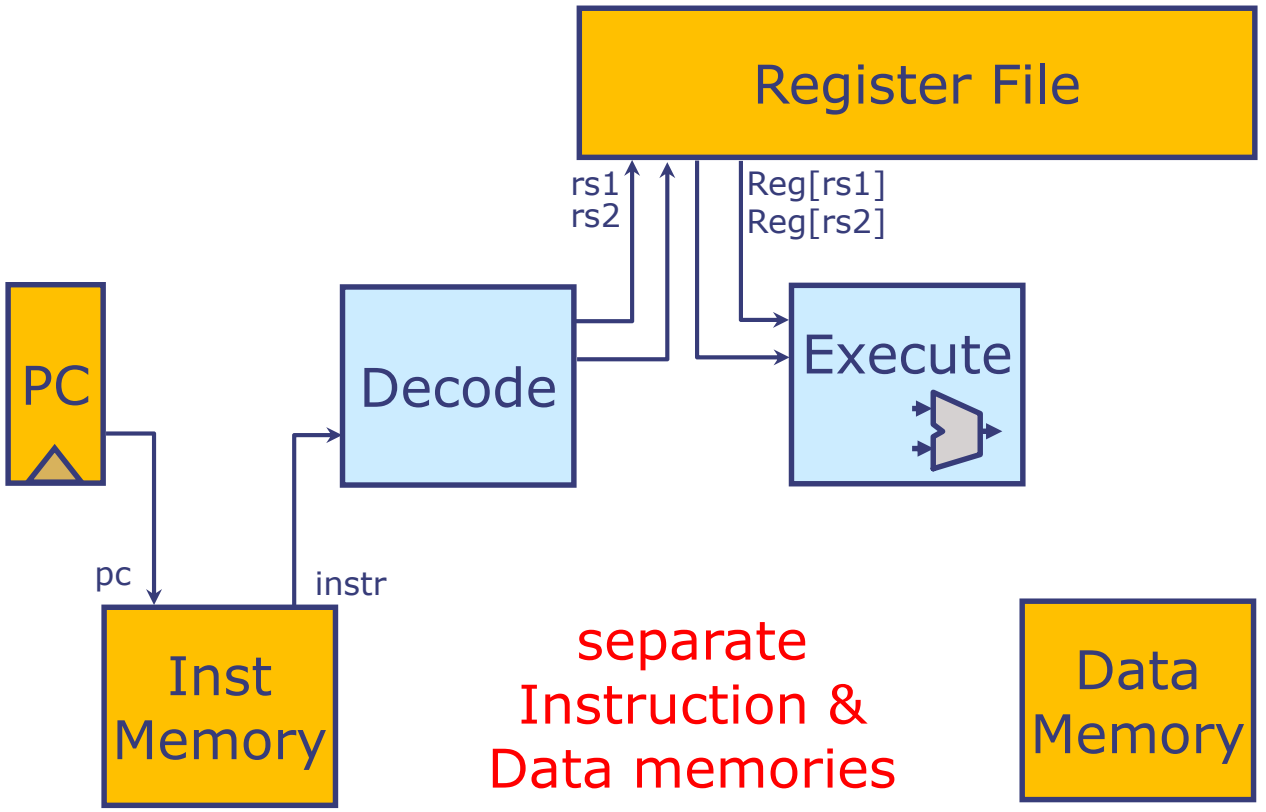
Single-Cycle RISC-V Processor



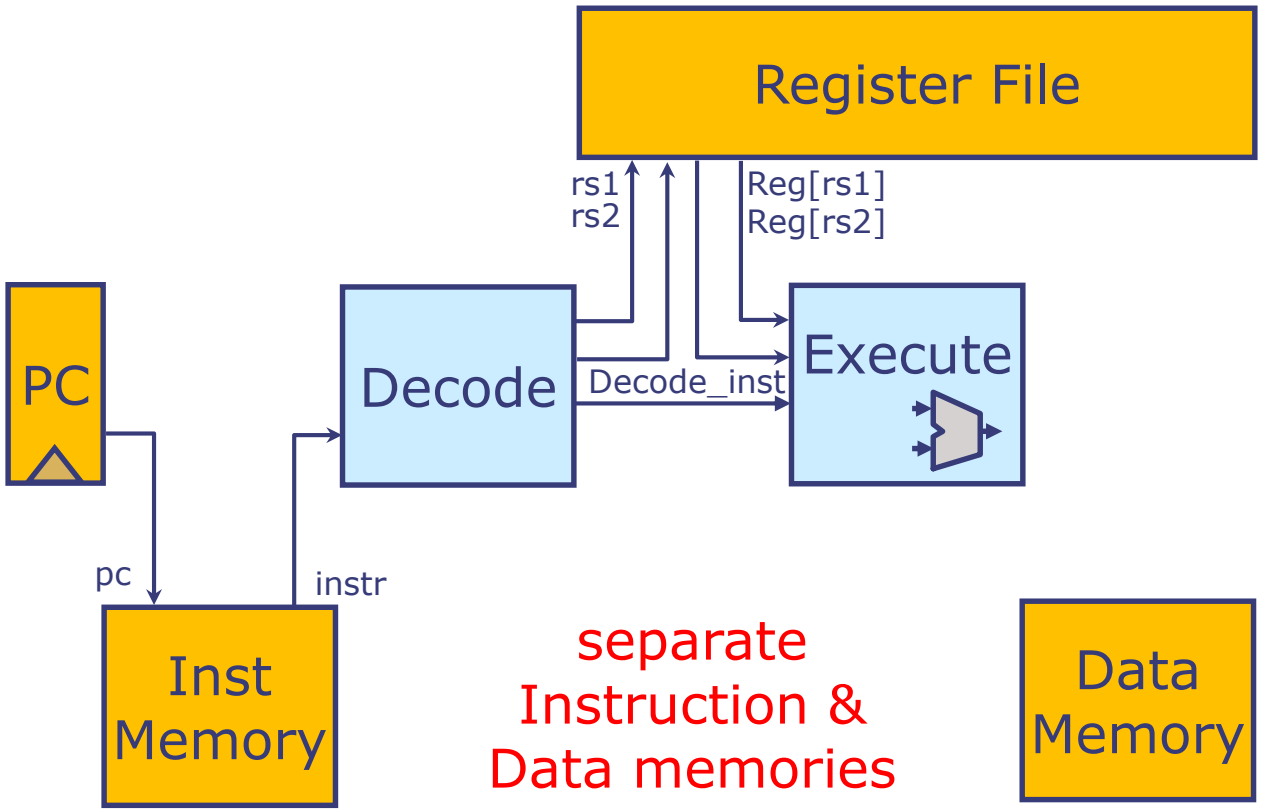
Single-Cycle RISC-V Processor



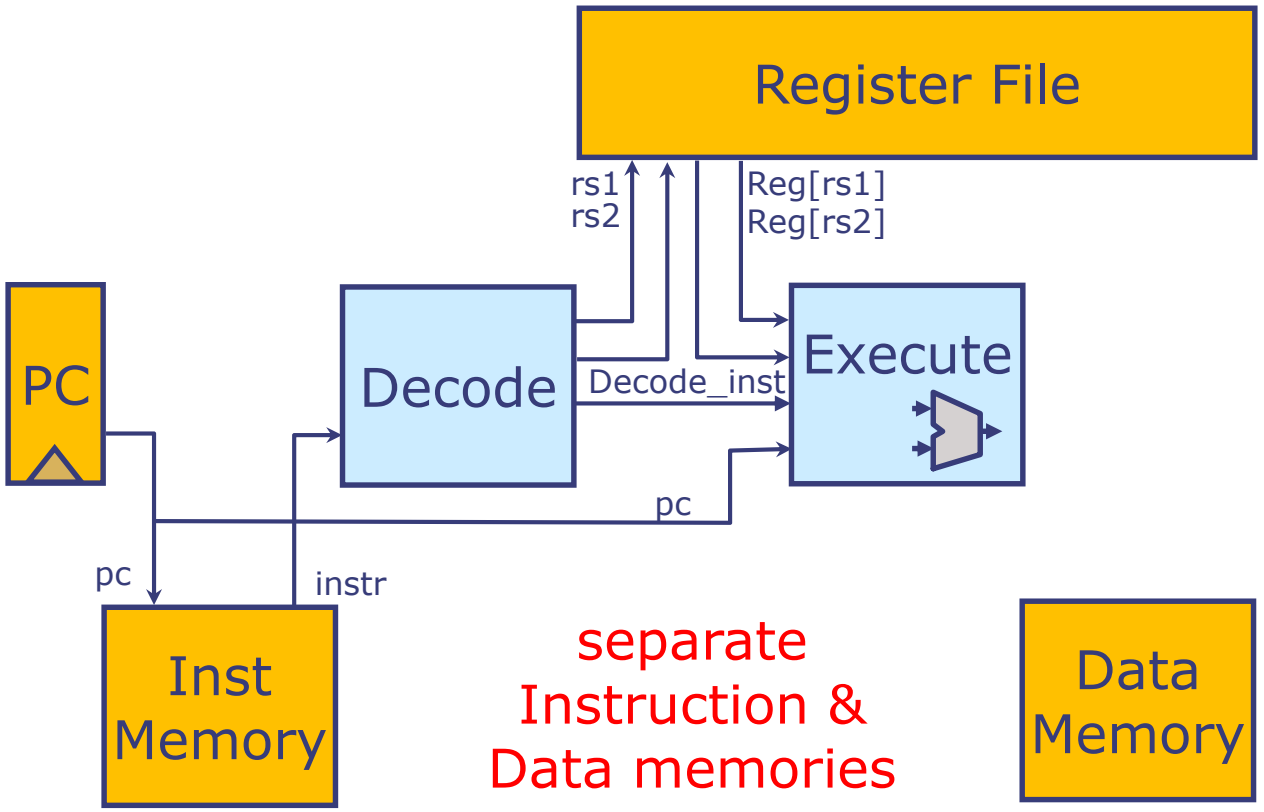
Single-Cycle RISC-V Processor



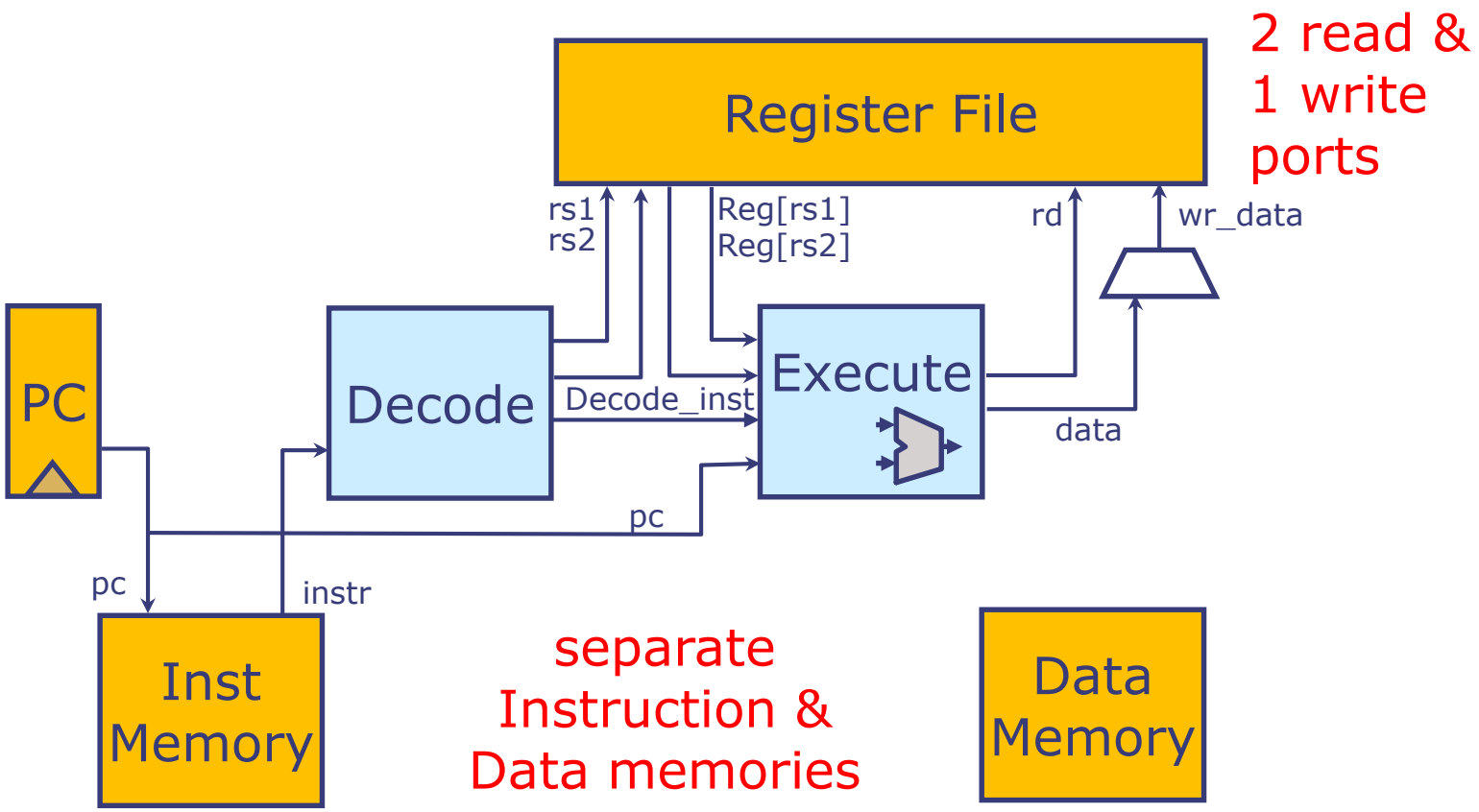
Single-Cycle RISC-V Processor



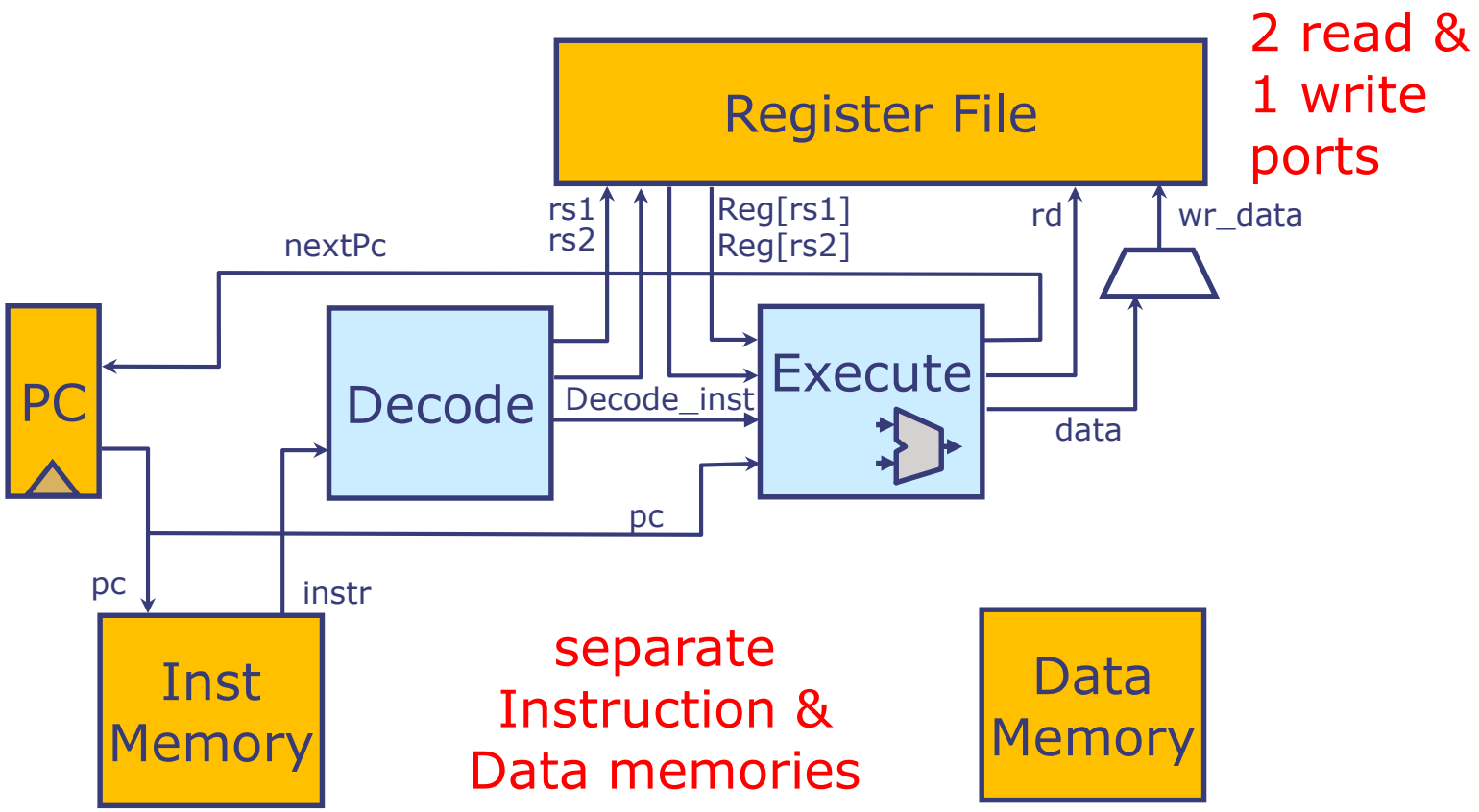
Single-Cycle RISC-V Processor



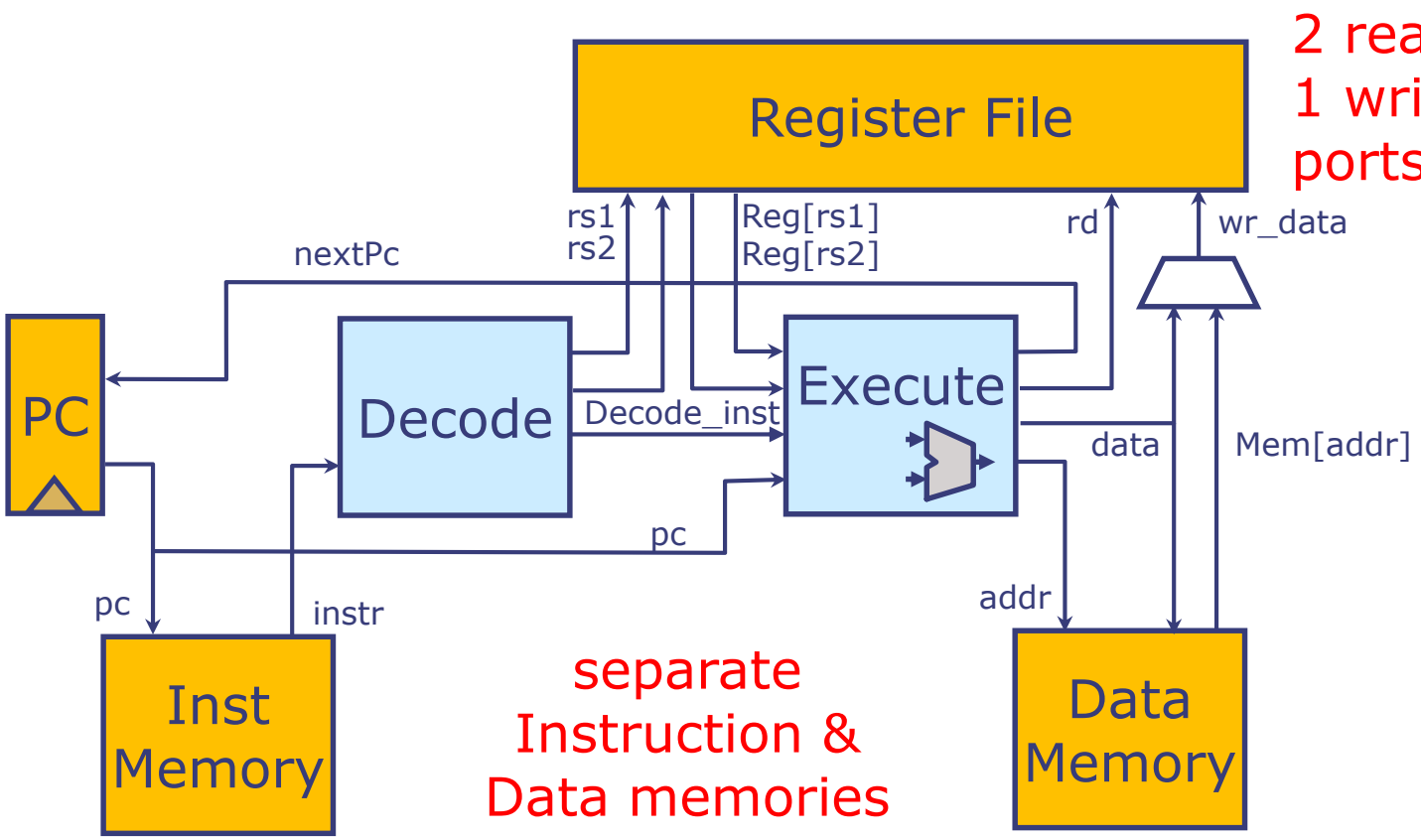
Single-Cycle RISC-V Processor



Single-Cycle RISC-V Processor



Single-Cycle RISC-V Processor



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

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