Module 5:

Single-cycle processor design

Introduction to computers II

José Manuel Mendías Cuadros
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Outline

- ✓ Introduction.
- ✓ Reduced architecture RISC-V.
- ✓ Data path design.
- ✓ Controller design.

- ✓ Technology.

These slides are based on:

- S.L. Harris and D. Harris. *Digital Design and Computer Architecture. RISC-V Edition.*
- D.A. Patterson and J.L. Hennessy. *Computer Organization and Design. RISC-V Edition.*

Introduction

- The most reliable method to know the performance of a computer is to measure the time that a program takes to execute.
  - The faster the program execution, the higher the performance.

- Given an architecture, the execution time of a program mainly depends on:
  - Number of instructions of the program.
  - Number of cycles that each instruction needs.
  - Cycle time (clock frequency).

depends on the programmer

depends on the HW design

- Usually, the number of cycles and the cycle time are inverse magnitudes:
  - Single-cycle processor: 1 cycle per instruction, with long cycle time.
  - Multicycle processor: several cycles per instruction, with short cycle time.

Introduction

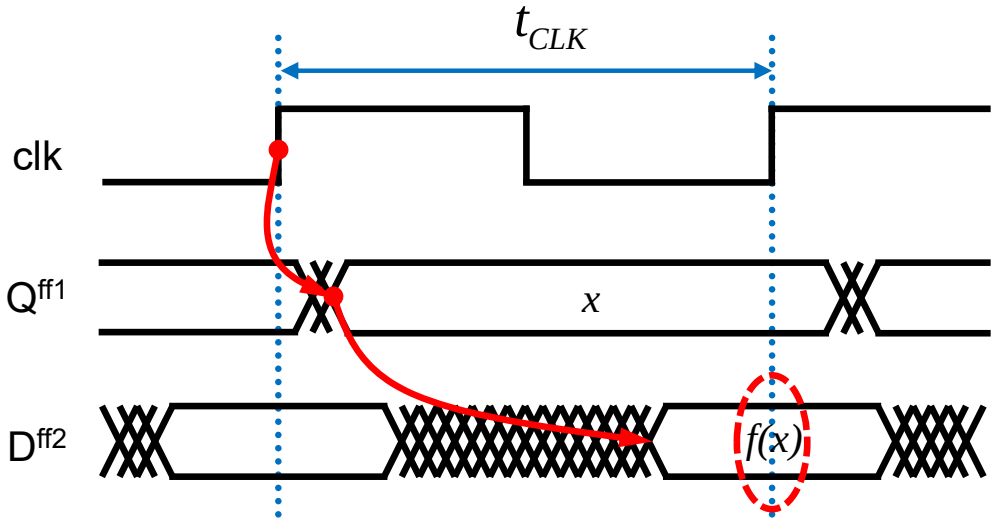
- In order to design a processor, algorithm design techniques are used, where the specification of the circuit is its architecture.

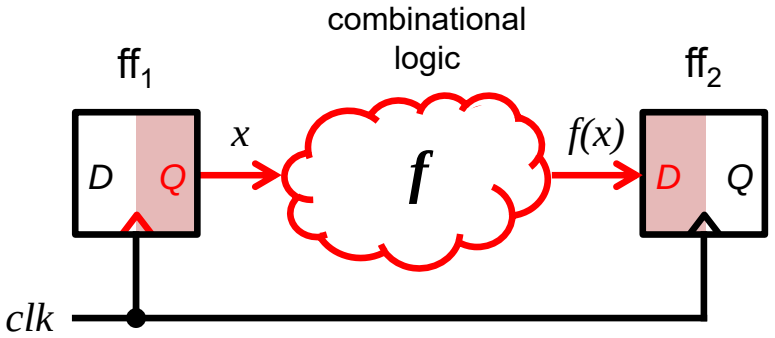
- The processor is formed by 2 elements:
  - Data path: performs operations and stores results.
    - At least, it has to include as many storage elements as defined in the architecture (visible to programmers).
    - It has to include the functional elements that are needed to perform all the operations of the instruction set.
  - Controller: sequences the transfers between registers that are defined for each instruction in the set.

- Depending on the chosen design strategy, we will have:
  - Single-cycle processors: all transfers between registers that happen in an instruction are performed in a single clock cycle.
  - Multicycle processors: all transfers between registers that happen in an instruction are distributed among several consecutive clock cycles.

Introduction

- Processors are designed according to the global clock (rising) edge synchronous timing model:
  - The clock arrives at all the flip-flops of the system, and all them change their state simultaneously at the (rising) clock edge.
  - The new values propagate through the combinational networks until they get at the flip-flop inputs.
  - This process repeats in each clock cycle.
  - The clock cycle time must be long enough so that all combinational systems can reach their steady state.





Reduced architecture RISC-V

Instruction set (i)

- A microarchitecture will be designed, able to execute a subset of the RISCV32 instruction set (which operates with 32-bit data only).

- Memory access instructions

`lw` ***rd, imm$_{12b}$(rs1)***	rd ← Mem[rs1 + sExt(imm)]	I-type
`sw` ***rs2, imm$_{12b}$(rs1)***	Mem[rs1 + sExt(imm$_{12b}$)] ← rs2	S-type

- Arithmetic-logic with both operands in registers

`add` ***rd, rs1, rs2***	rd ← rs1 + rs2	R-type
`sub` ***rd, rs1, rs2***	rd ← rs1 – rs2	R-type
`and` ***rd, rs1, rs2***	rd ← rs1 & rs2	R-type
`or` ***rd, rs1, rs2***	rd ← rs1 | rs2	R-type
`slt` ***rd, rs1, rs2***	rd ← *if* (rs1 <$_S$ rs2) *then* (1) *else* (0)	R-type

Reduced architecture RISC-V

Instruction set (ii)

- Arithmetic-logic with one immediate operand

`addi` ***rd, rs1,*** imm_{12b}	rd ← rs1 + sExt(imm),	I-type
`andi` ***rd, rs1,*** imm_{12b}	rd ← rs1 & sExt(imm)	I-type
`ori` ***rd, rs1,*** imm_{12b}	rd ← rs1 | sExt(imm)	I-type
`slti` ***rd, rs1,*** imm_{12b}	rd ← *if* (rs1 $<_S$ sExt(imm)) *then* (1) *else* (0)	I-type

- Conditional branch instructions

`beq` ***rs1, rs2,*** imm_{13b}	PC ← *if* (rs1 = rs2) *then* (PC + sExt($imm_{12:1}$ << 1)) *else* (PC+4)	B-type

- Branch to function instruction

`jal` **rd,** imm_{21b}	PC ← PC + sExt($imm_{20:1}$ << 1), rd ← PC+4	J-type

Reduced architecture RISC-V

Instruction format

- The instruction formats and field encoding will be the same as in the complete architecture.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	op	*R-type*
$imm_{11:0}$		rs1	funct3	rd	op	*I-type*
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	*S-type*
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	*B-type*
$imm_{20,10:1,11,19:12}$				rd	op	*J-type*

Reduced architecture RISC-V

Instruction format: field encoding

Instruction	Type	funct7 bits 31:25	funct3 bits 14:12	op bits 6:0
`lw`	I	–	010	0000011
`sw`	S	–	010	0100011
`add`	R	0000000	000	0110011
`sub`	R	0100000	000	
`slt`	R	0000000	010	
`or`	R	0000000	110	
`and`	R	0000000	111	
`addi`	I	–	000	0010011
`slti`	I	–	010	
`ori`	I	–	110	
`andi`	I	–	111	
`beq`	B	–	000	1100011
`jal`	J	–	–	1101111

Data path design

Storage elements (i)

- The storage elements are those that are visible to programmers: Memory, PC and 32 general purpose registers.

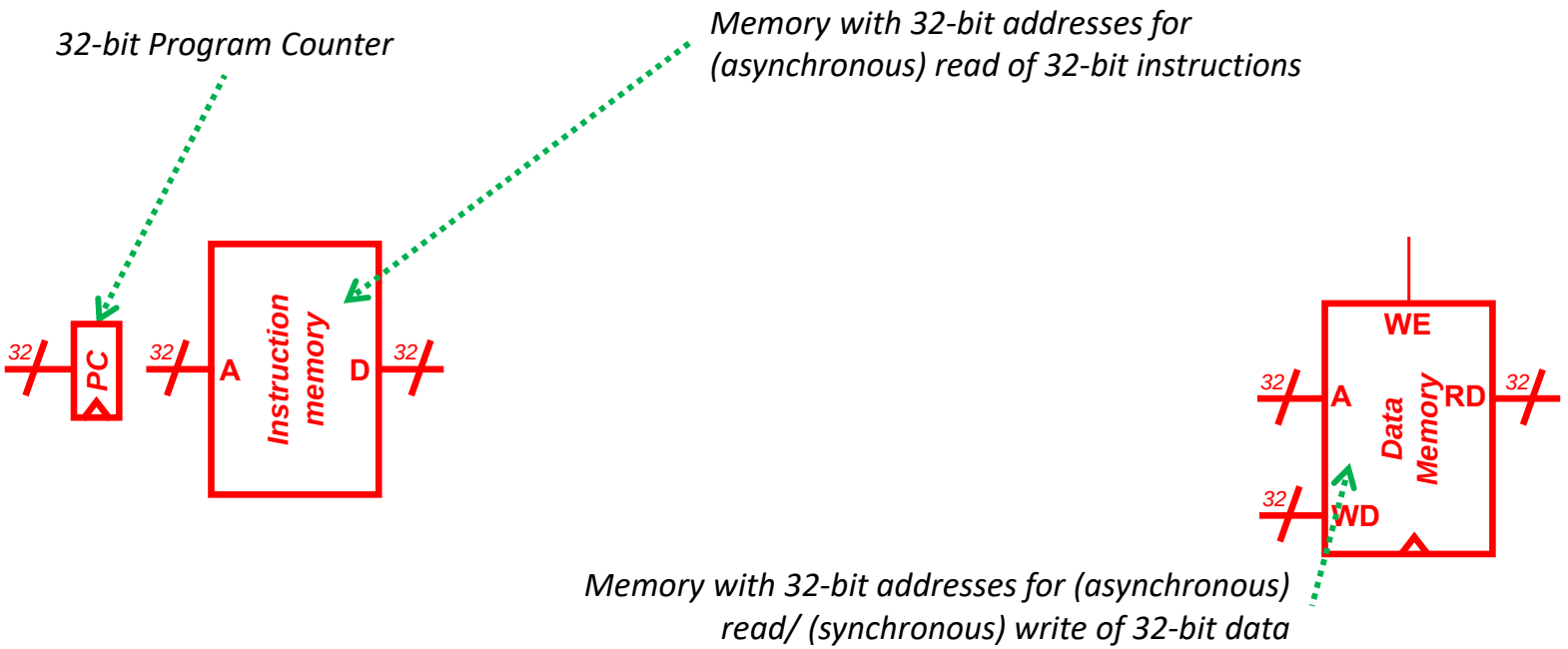
- The Memory will have an idealized behavior:
  - Integrated in the processor.
  - Byte-addressable, but able to accept/offer 4 bytes per access.
  - Access time lower than the processor cycle time.
  - Split in two, because instructions must be read from memory in the same clock cycle in which data are read/written.
    - Instruction memory: it will behave as a combinational ROM.
    - Data memory: it will behave as a large Register File, with double data port for separate input and output.

- The Program Counter will be an array of D flip-flops:
  - Since instructions are executed in a single cycle, the PC must be updated in all the cycles and does not require a signal to control its load.

Data path design

Storage elements (ii)

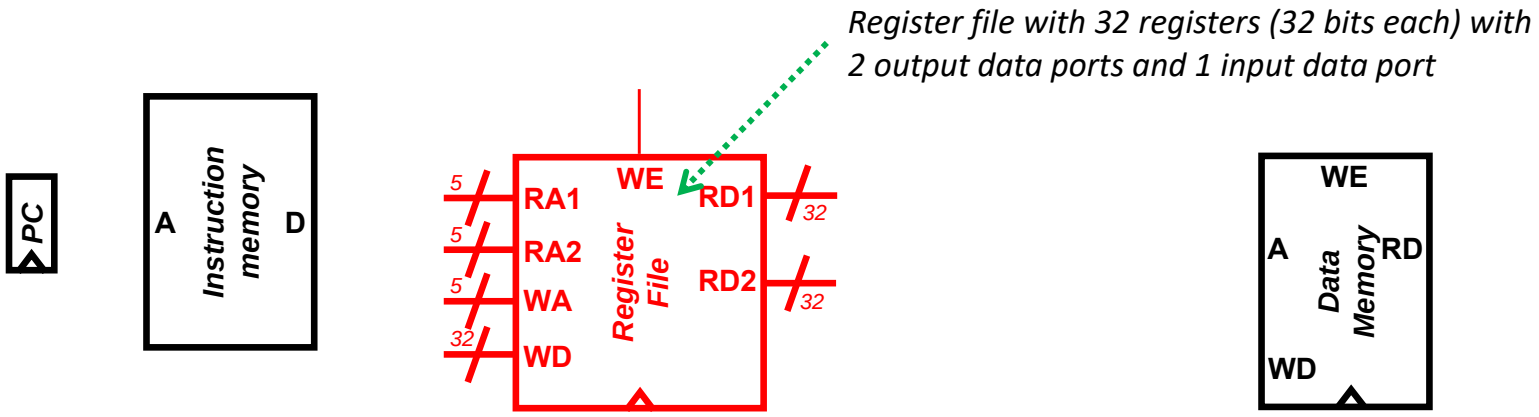
- The storage elements are those that are visible to programmers: Memory, PC and 32 general purpose registers.



Data path design

Storage elements (iii)

- The storage elements are those that are visible to programmers: Memory, PC and 32 general purpose registers.



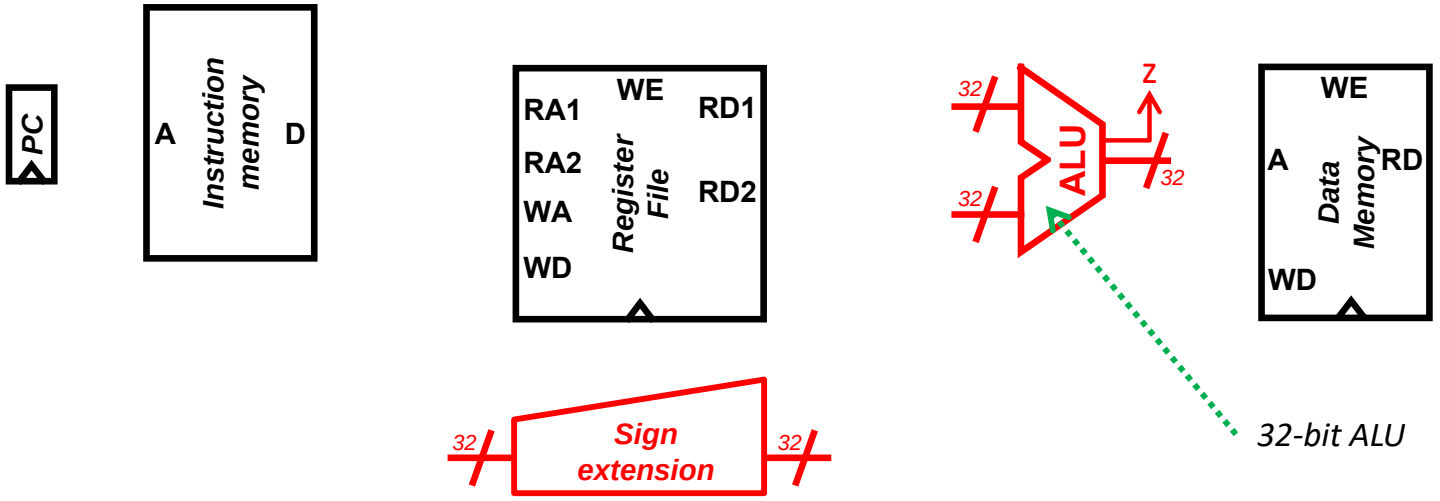
- The 32 registers are organized in a 3-port Register File:
  - In a single-cycle processor, R-type instructions read 2 registers and write 1 register from the Register File in the same clock cycle.

```
add rd, rs1, rs2
```
RF[rd] ← RF[rs1] + RF[rs2], PC ← PC+4

Data path design

Functional elements (i)

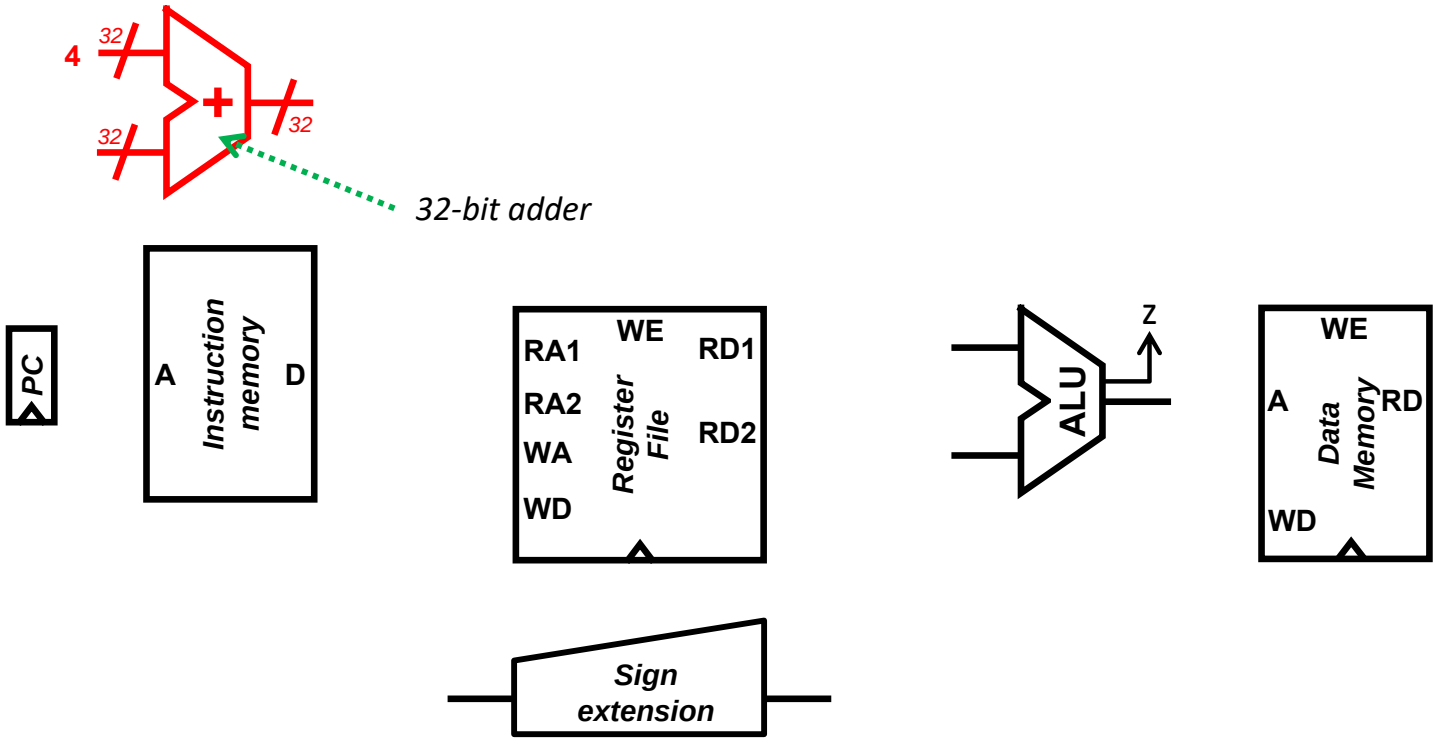
- It will have an ALU and a Sign Extension module, both combinational:
  - The ALU will perform all the arithmetic-logic operations of the ISA.
    - With a Z flag, to perform the equality condition of **beq** instructions by means of a subtraction.
  - The Sign Extension module will build the 32-bit immediate operand based on the imm fields (with a smaller width) of the instructions.



Data path design

Functional elements (ii)

- It will have an additional adder to increment the PC
  - In a single-cycle processor, the PC is updated with the address of the following instruction in the same clock cycle in which the ALU works.

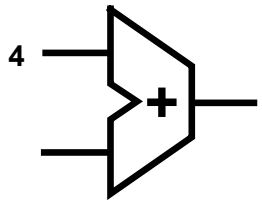


- It always adds +4 because the Memory is byte-addressable, and all the instructions take 32 bits (4 bytes).

Data path design

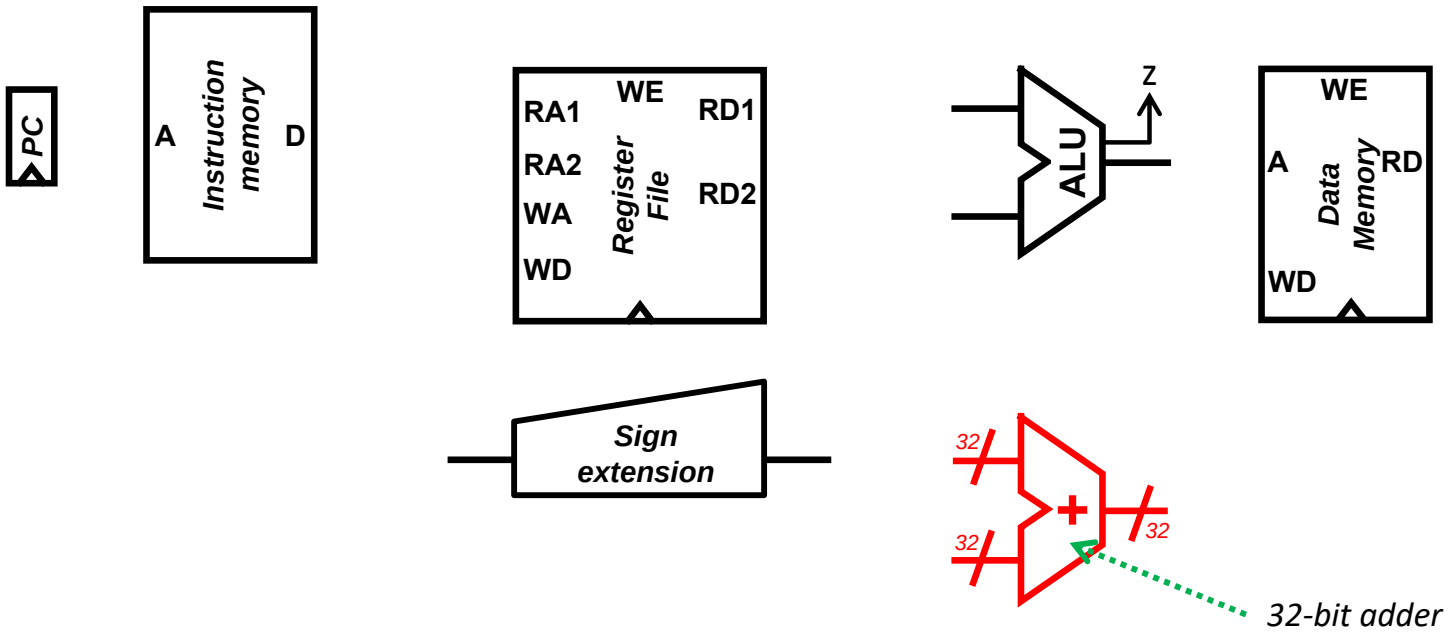
Functional elements (iii)

- It will have an additional adder to calculate branch addresses
  - In a single-cycle processor, **beq** instructions operate in the ALU, calculate PC+4 and calculate the branch address in the same clock cycle.



`beq rs1,rs2,imm`

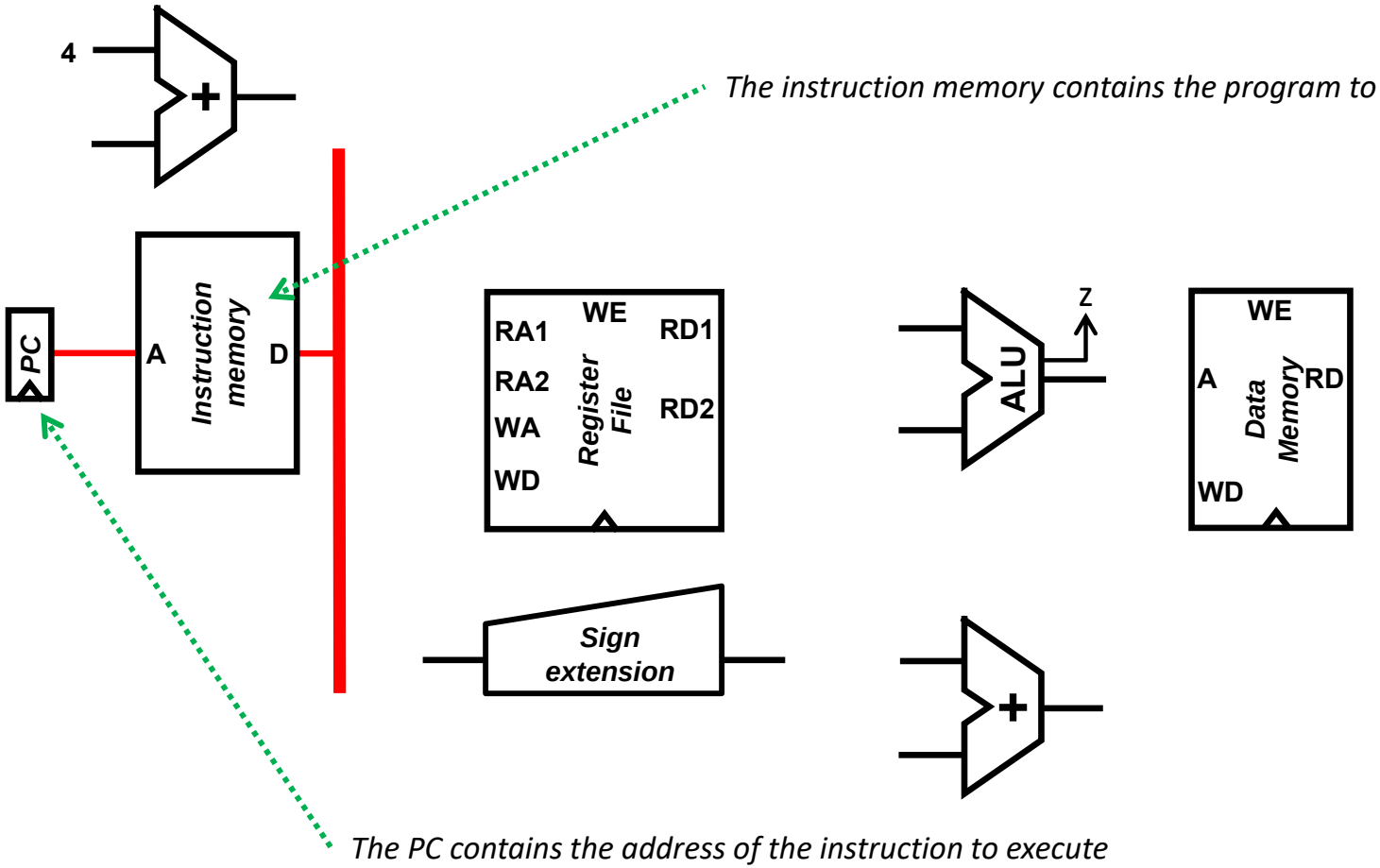
PC ← *if* (RF[rs1] = RF[rs2])
then (PC + sExt(imm)) *else* (PC+4)



Data path design

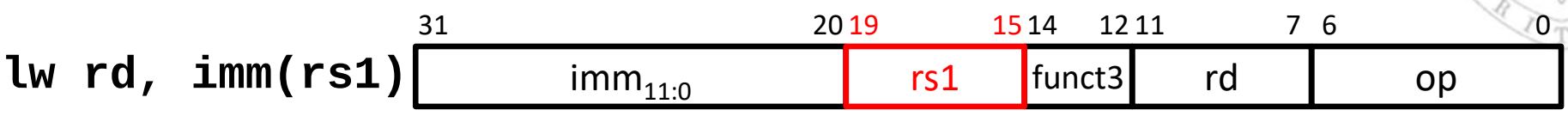
Instruction fetch

- The execution of an instruction starts by reading it from memory (fetch).
  - The PC and the Instruction Memory are connected to read the instruction to execute (the one whose address is stored in the PC).

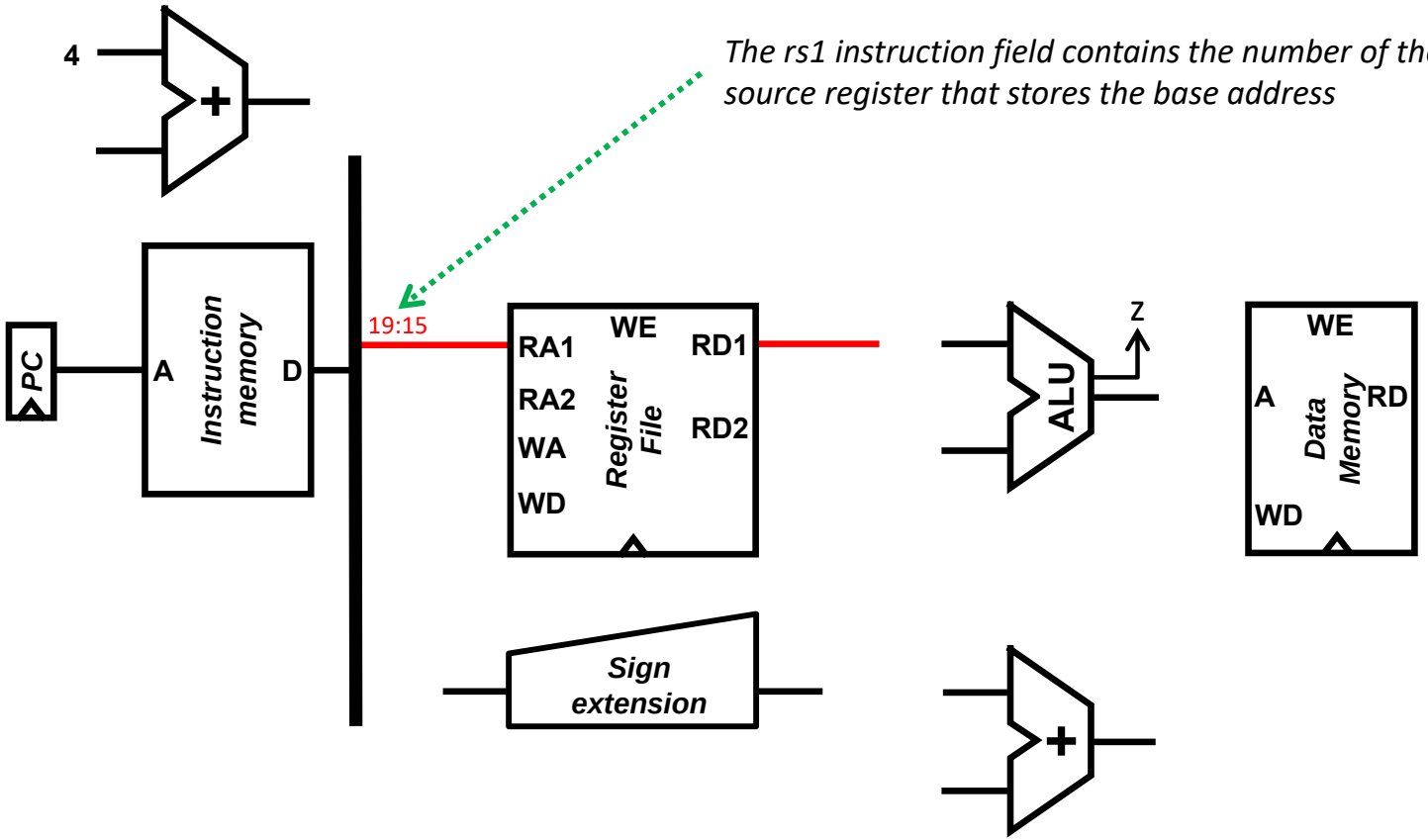


Data path design

`lw` instruction: reading the base register



RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4

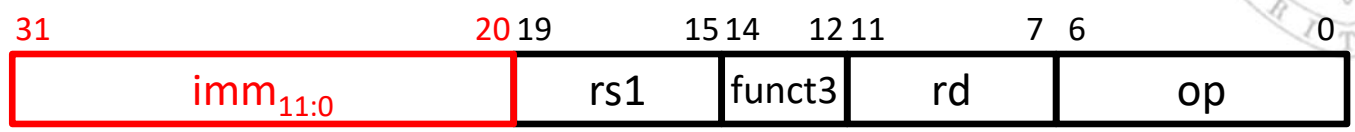


- The Instruction Memory and the Register File are connected to read the base address contained in register rs1.

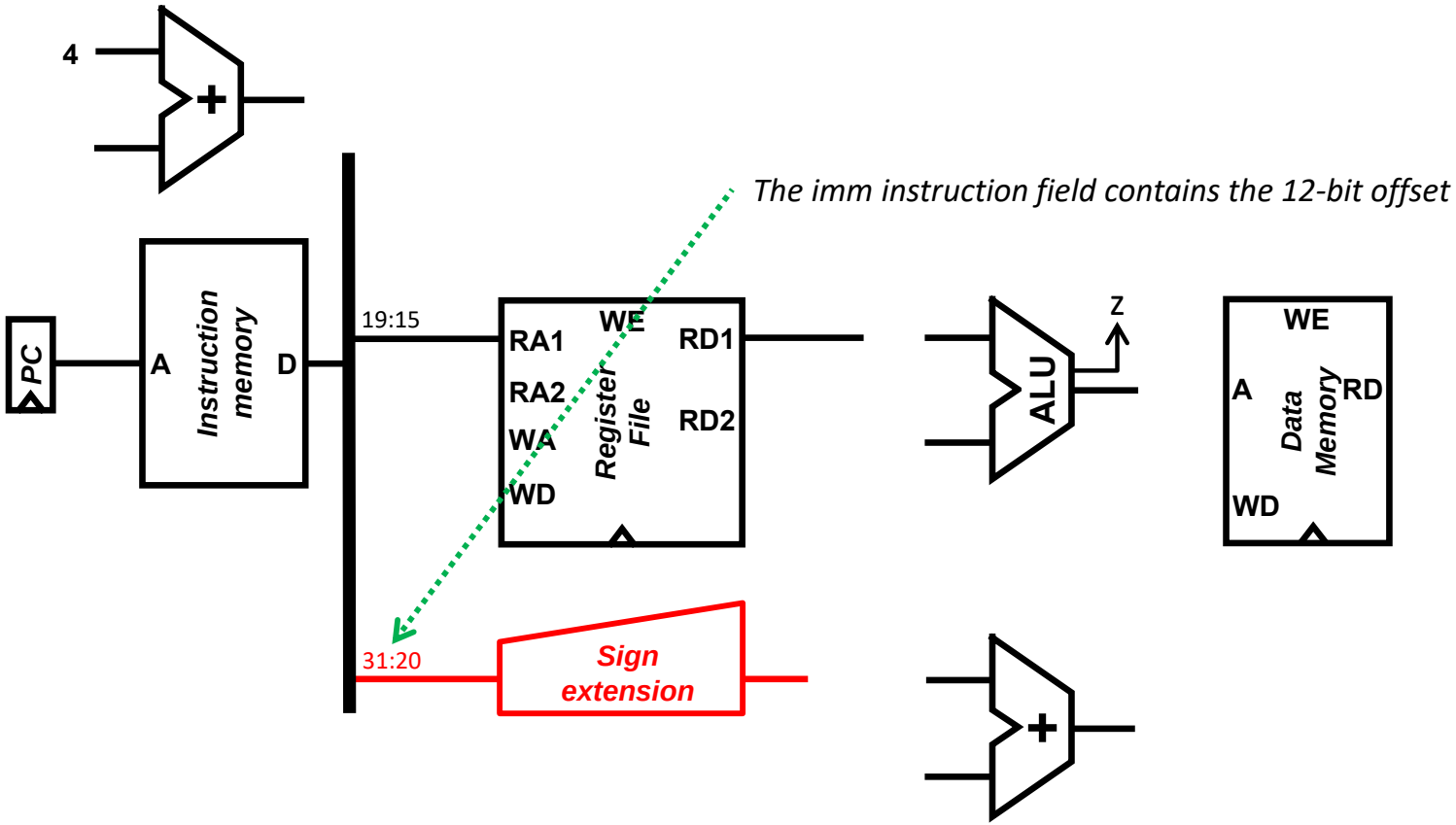
Data path design

`lw` instruction: calculating the offset

`lw rd, imm(rs1)`



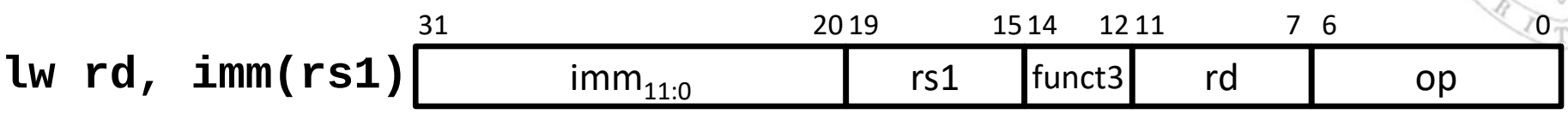
RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4



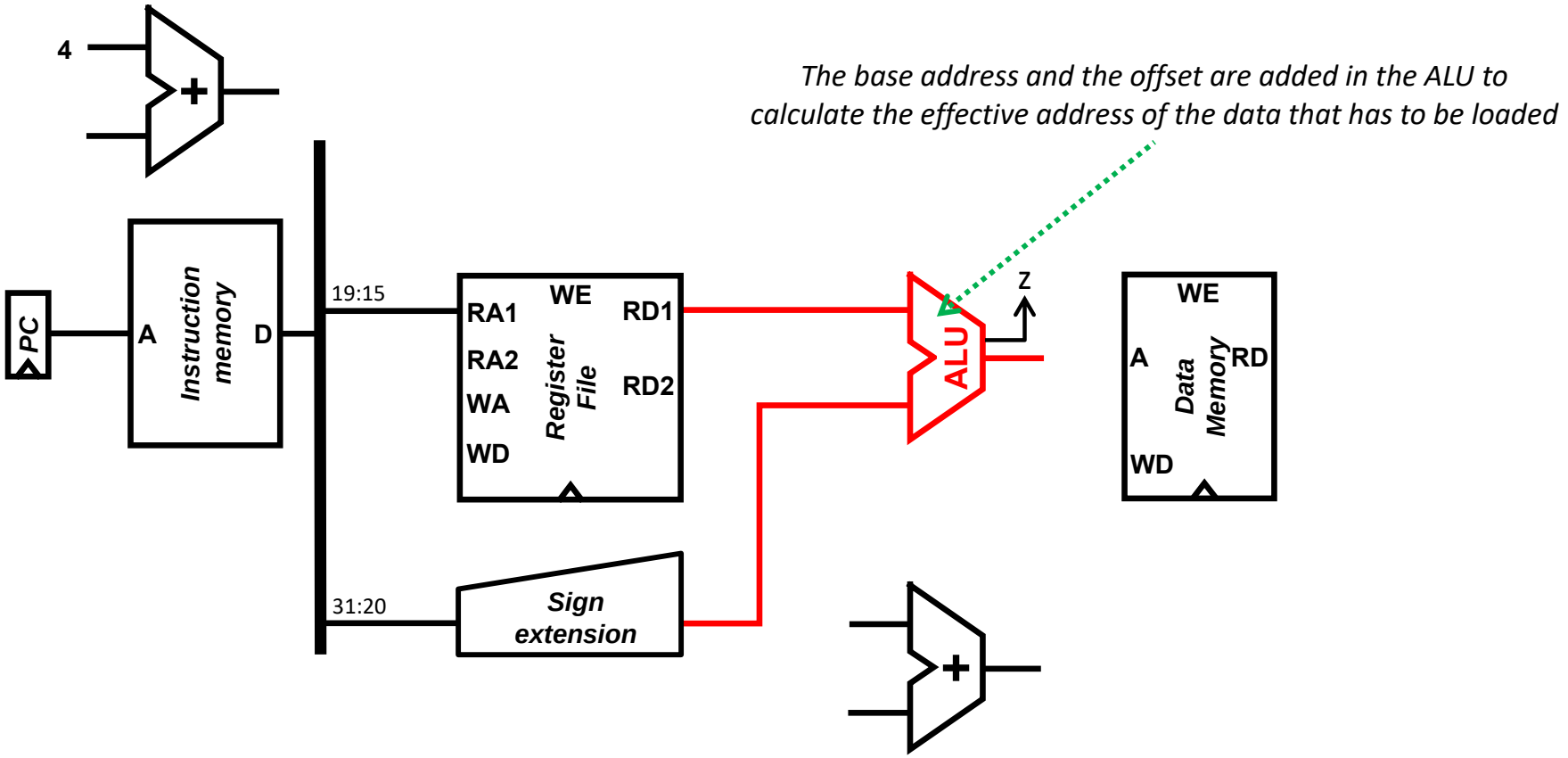
- The Instruction Memory and the Sign Extension module are connected to extend the 12-bit instruction offset to 32 bits.

Data path design

`lw` instruction: calculating the effective address



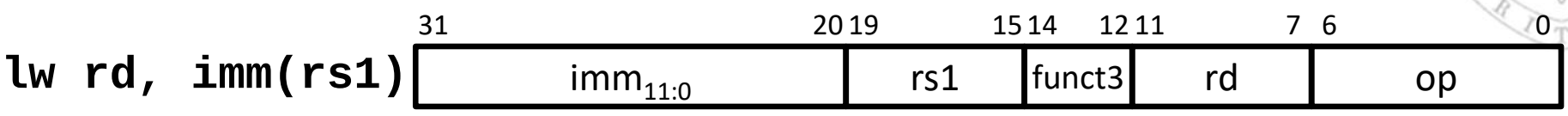
RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4



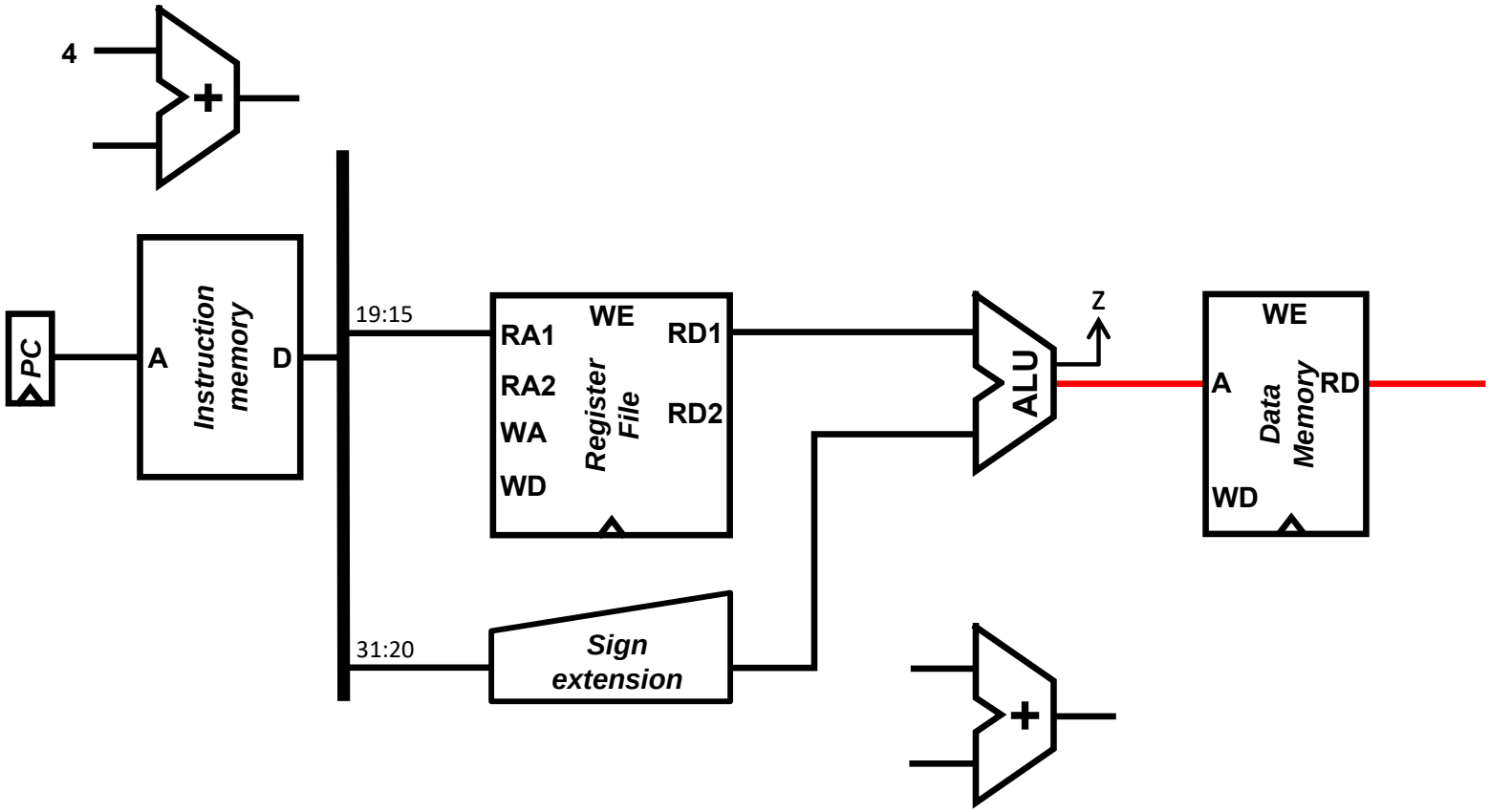
- The Register File and the ALU are connected to calculate the effective address

Data path design

lw instruction: reading the operand



RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4

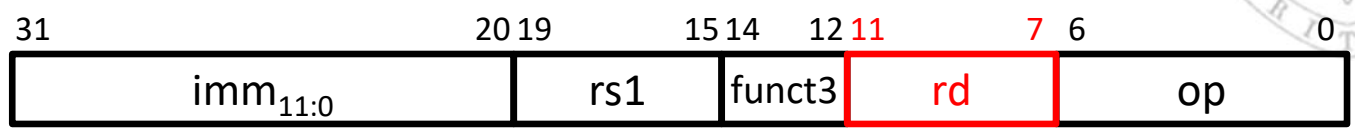


- The ALU and the Data Memory are connected to read the data

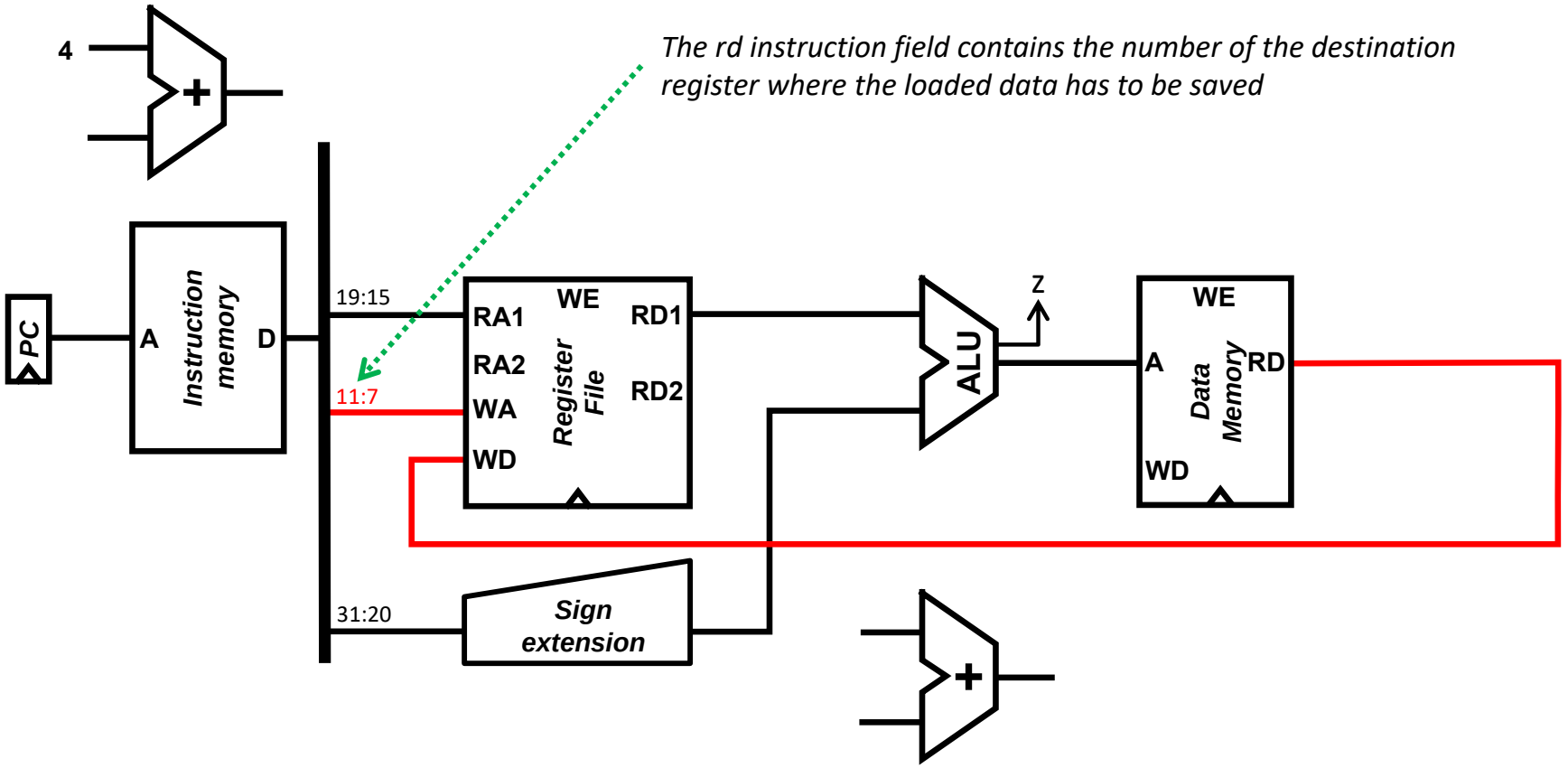
Data path design

`lw` instruction: loading the operand

`lw rd, imm(rs1)`



RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4

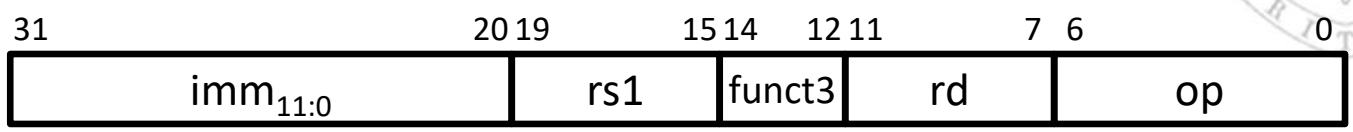


- The Data Memory and the Register File are connected to load the data into the rd register

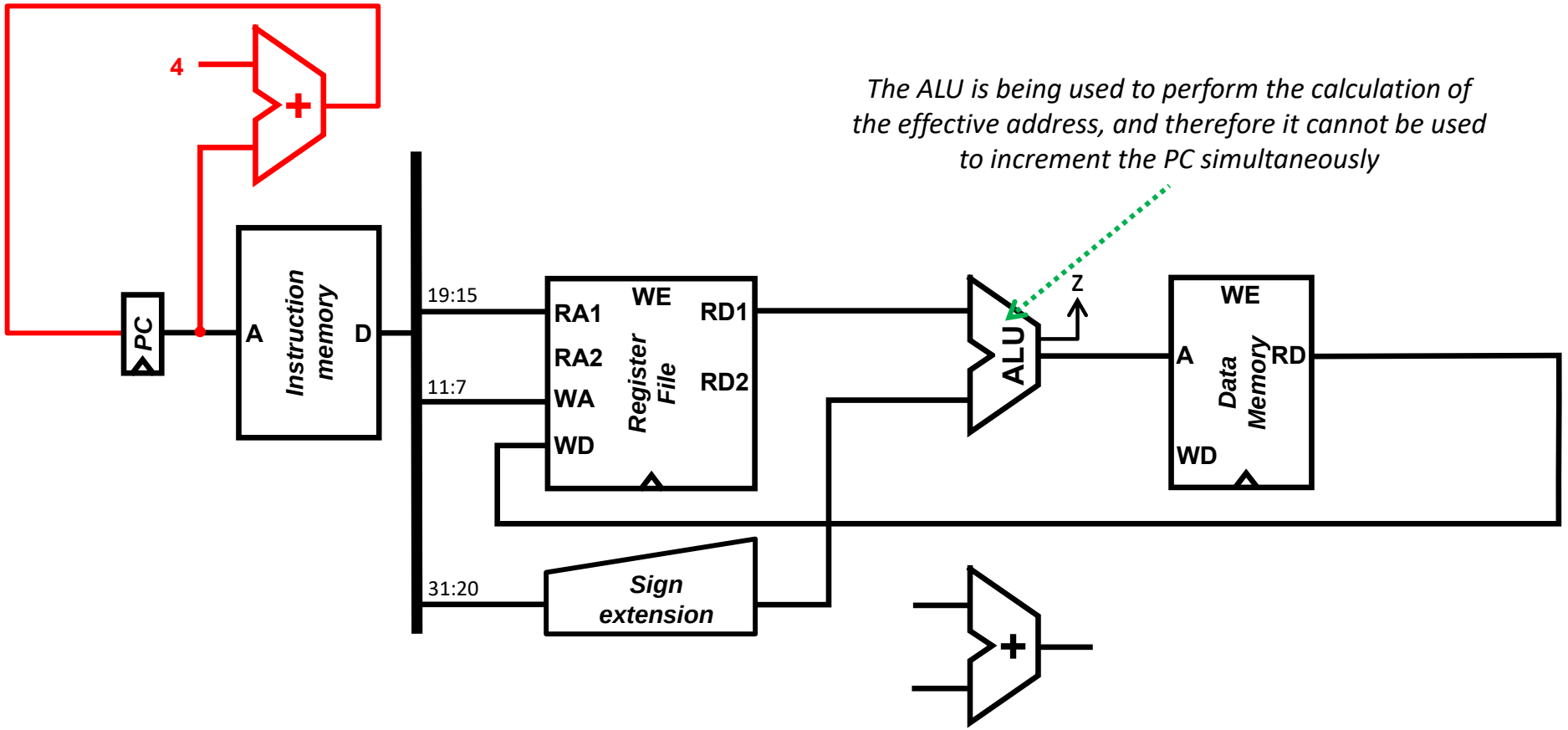
Data path design

`lw` instruction: incrementing the PC

`lw rd, imm(rs1)`



RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4

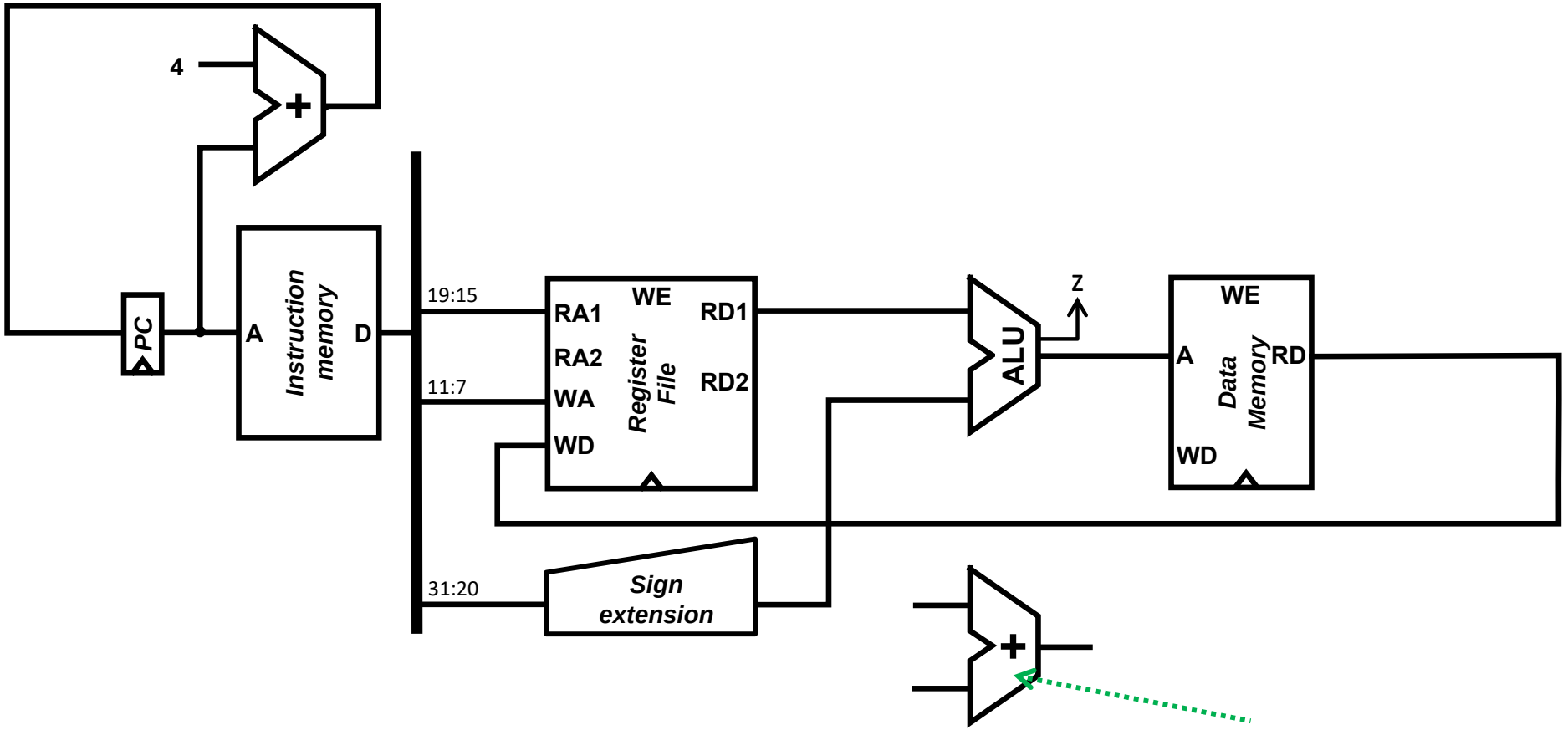


- The PC and the Incrementer are connected to update the PC with the address of the following instruction

Data path design

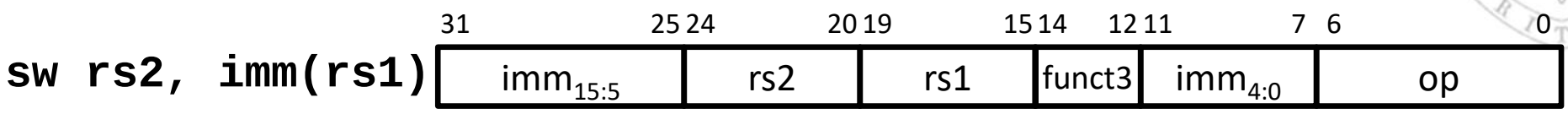
Data path for `lw` instructions

- This data path can execute any sequence of `lw` instructions. It will be expanded in order to execute others.



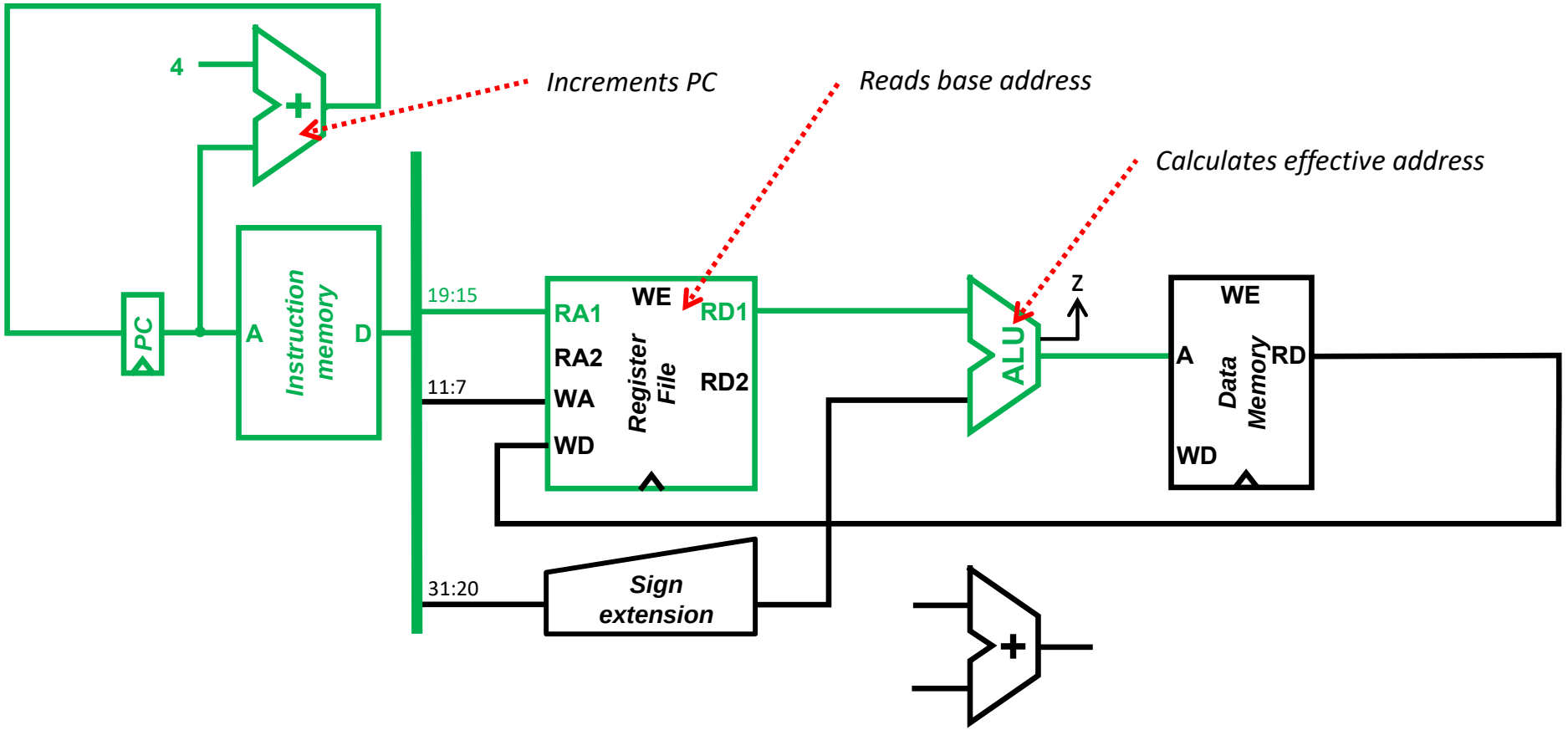
Data path design

Data path for **sw** instructions



sw rs2, imm(rs1)

Mem[RF[rs1] + sExt(imm)] ← RF[rs2], PC ← PC+4

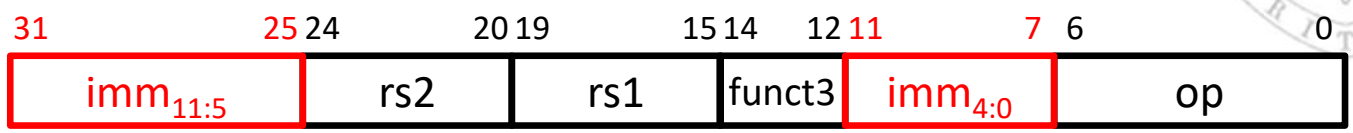


- Part of this data path can be reused to execute **sw** instructions

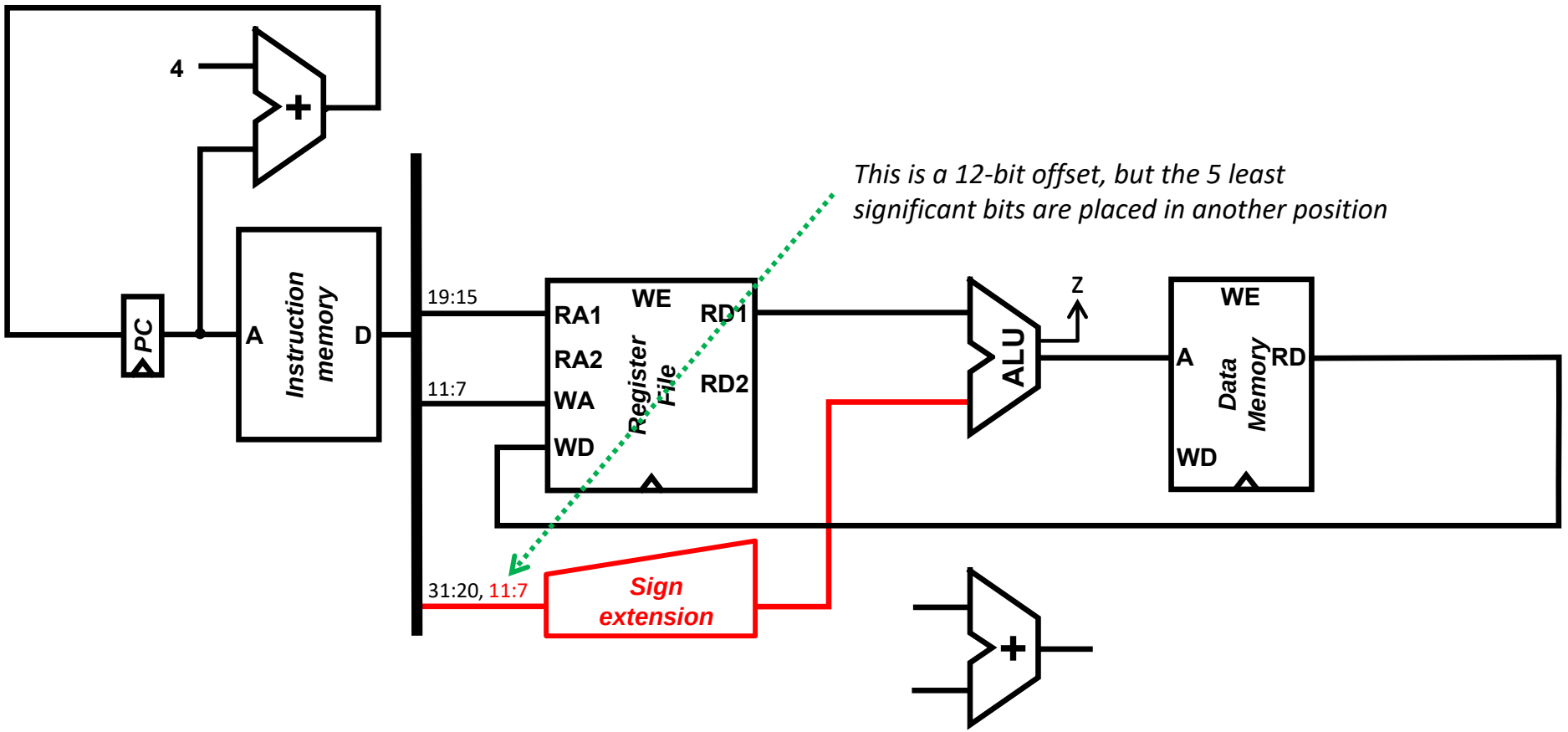
Data path design

`sw` instruction: calculating the offset

`sw rs2, imm(rs1)`



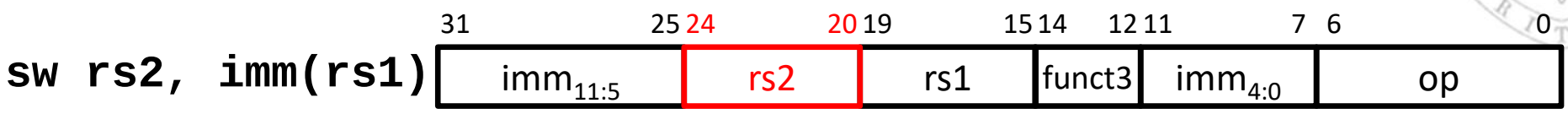
Mem[RF[rs1] + sExt(imm)] ← RF[rs2], PC ← PC+4



- The connection between the Instruction Memory and the Sign Extension module is expanded because the offset is placed in another position within the instruction.

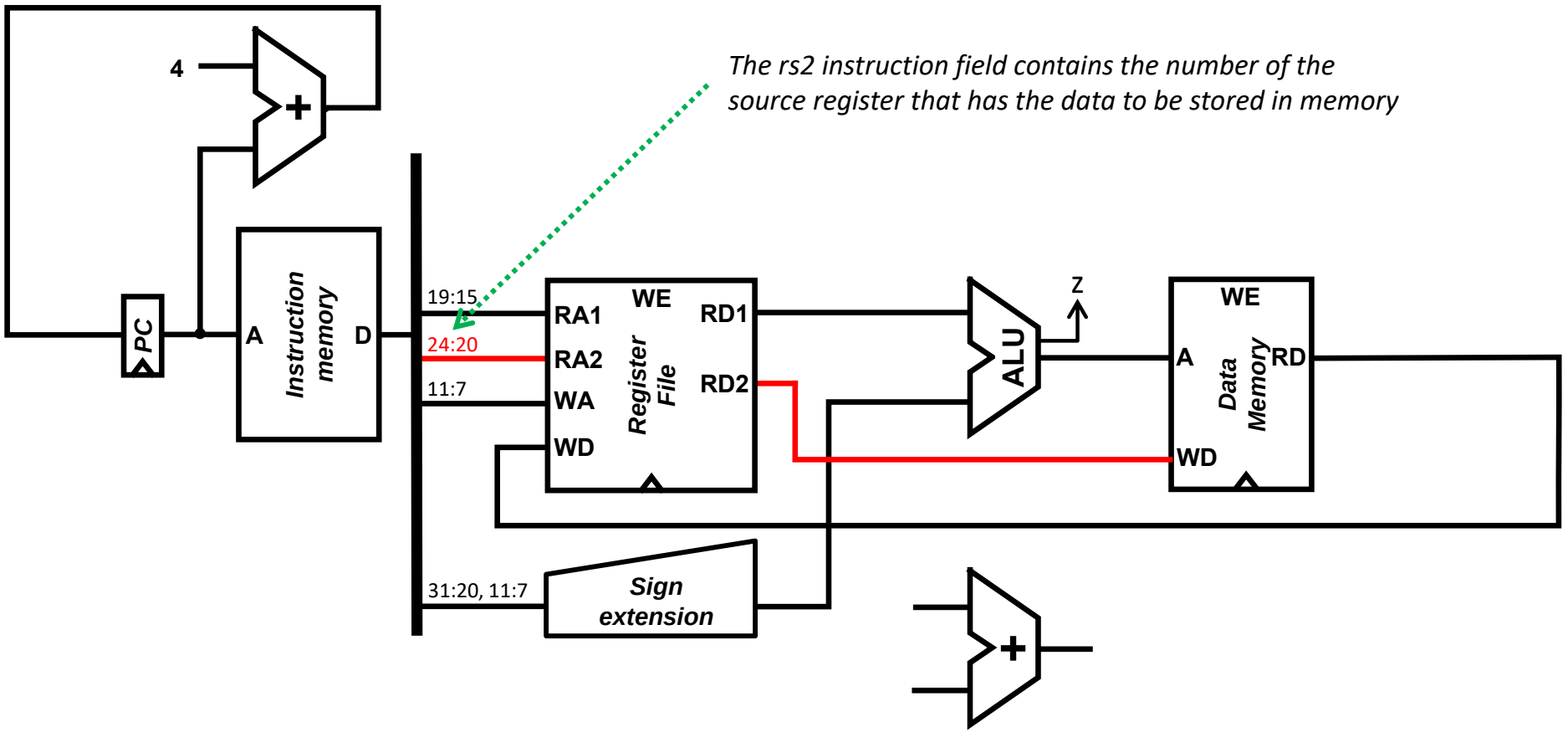
Data path design

`sw` instruction: storing the data



`sw rs2, imm(rs1)`

Mem[RF[rs1] + sExt(imm)] ← RF[rs2], PC ← PC+4

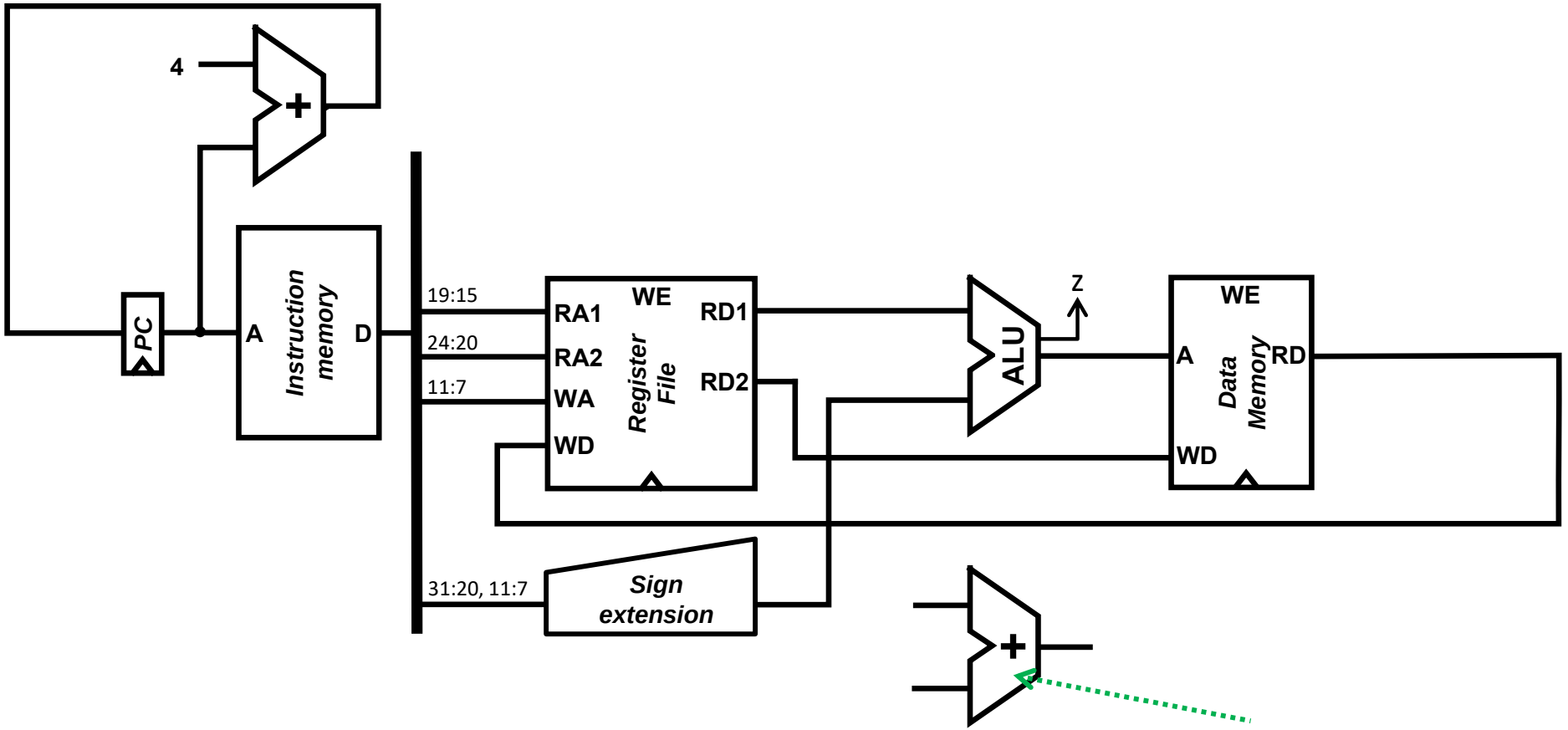


- The Register File and the Data Memory are connected to store the data contained in register rs2 into memory

Data path design

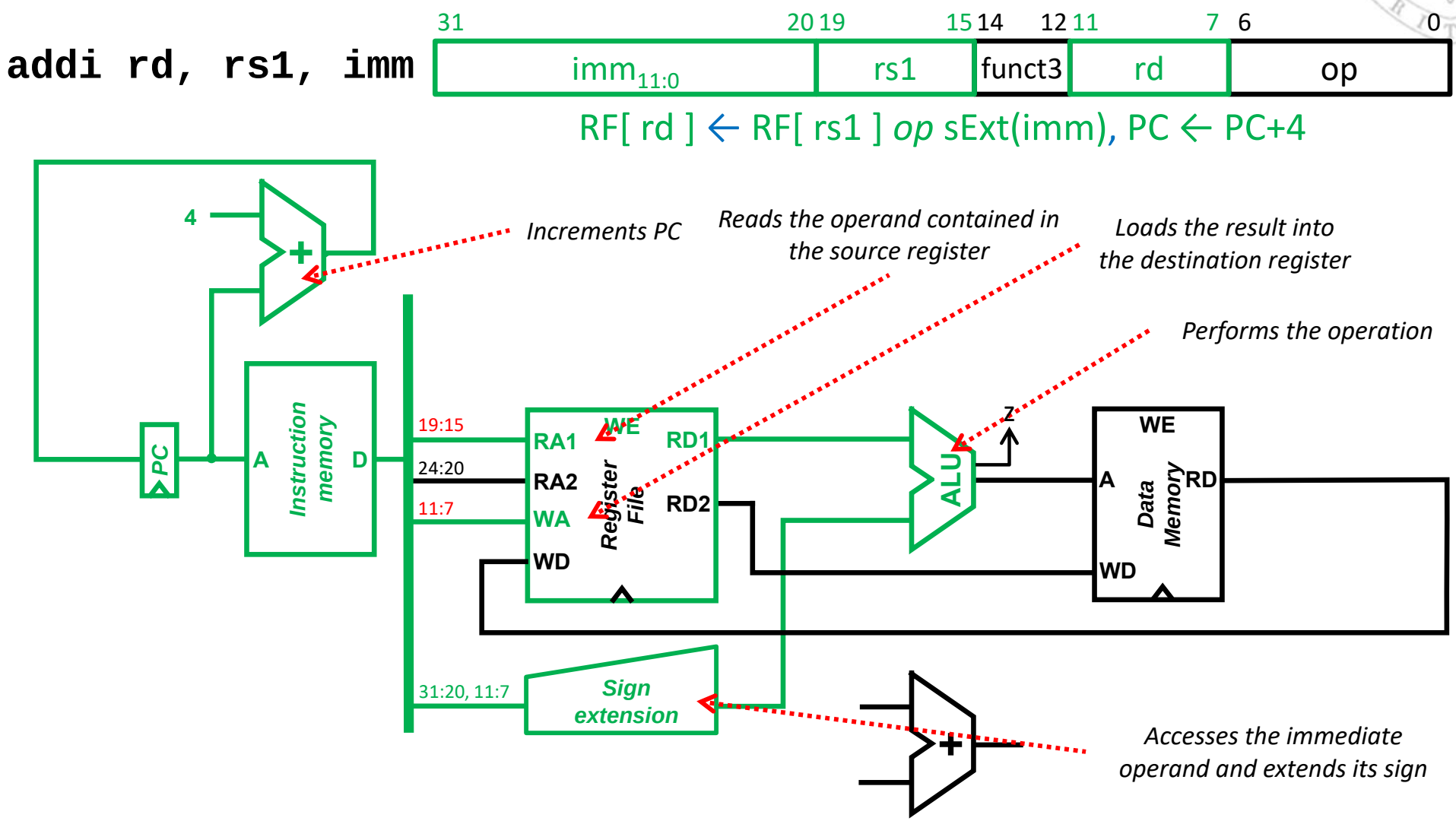
Data path for `lw/sw` instructions

- This data path can execute any sequence of **`lw`** and/or **`sw`** instructions. It will be further expanded.



Data path design

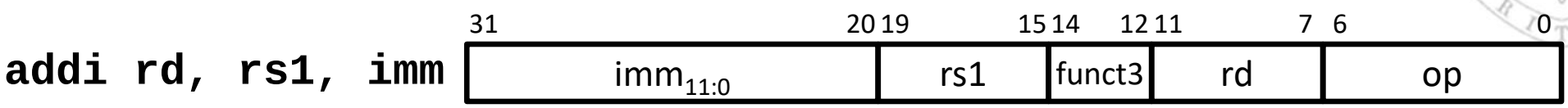
Data path for **addi**-like instructions



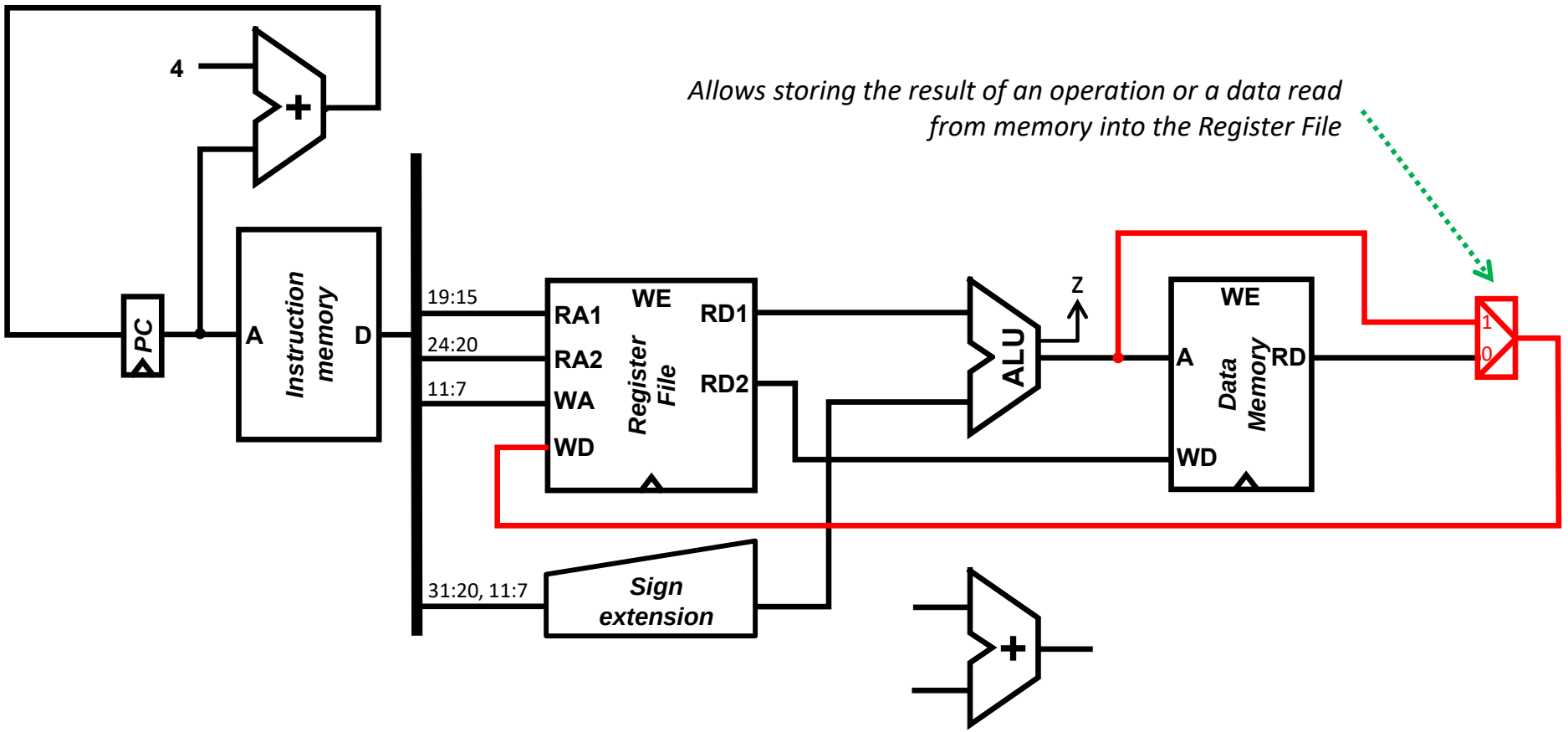
- Part of this data path can be reused to execute **addi/andi/ori/slti** instructions

Data path design

addi-like instructions: storing the result



RF[rd] ← RF[rs1] *op* sExt(imm), PC ← PC+4

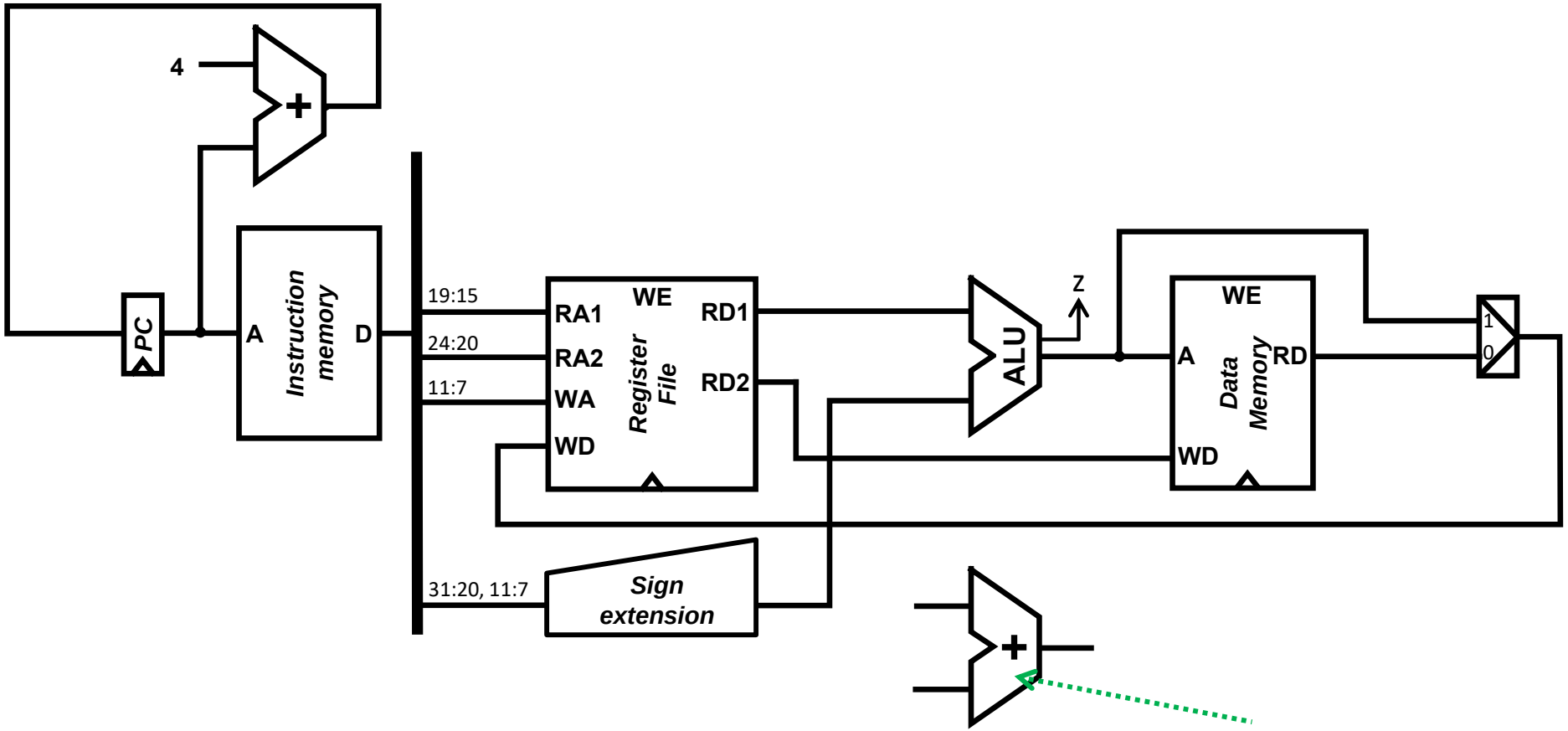


- A multiplexer is added in order to store the ALU operation result into register rd

Data path design

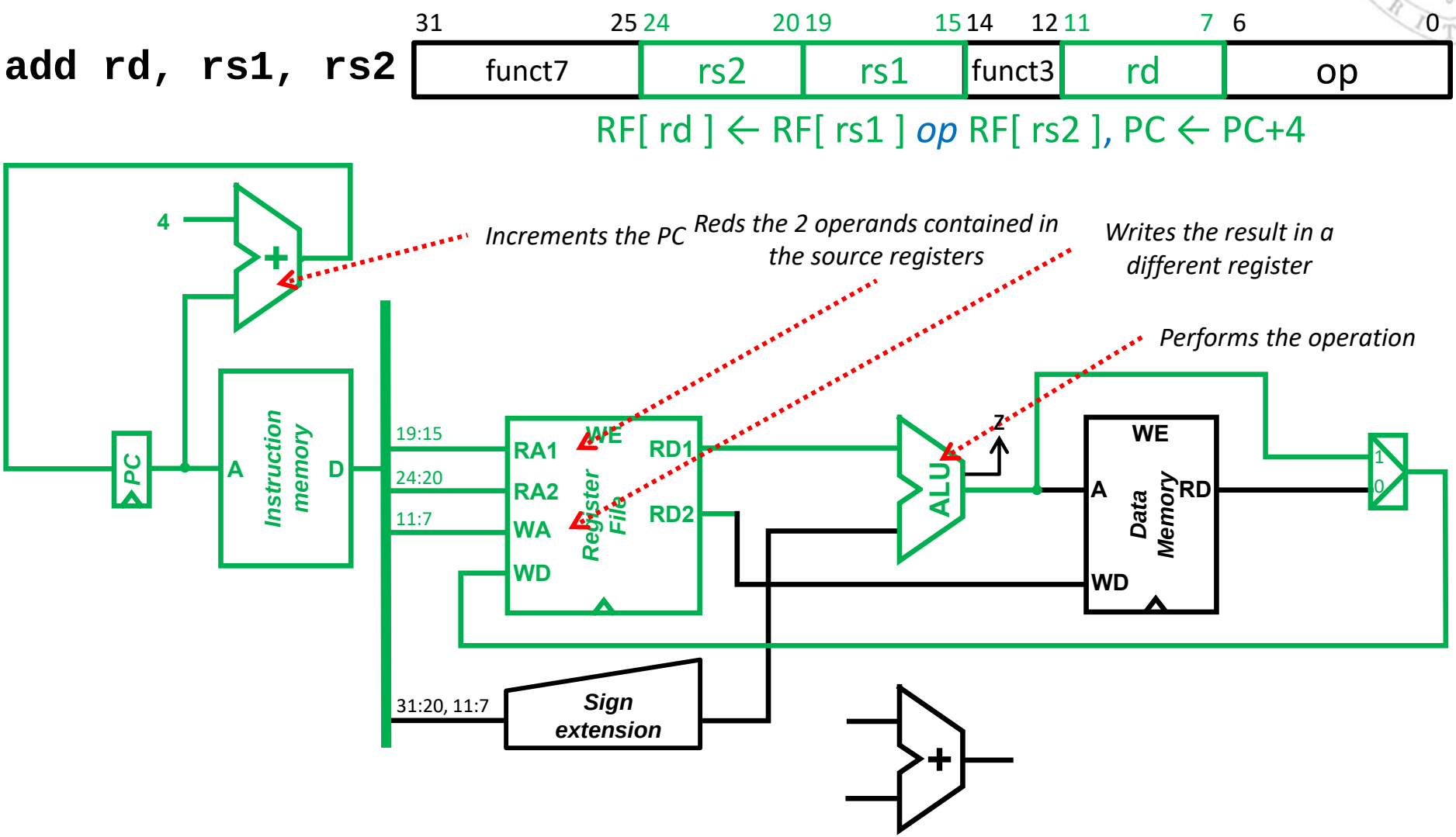
Data path for `lw/sw/addi` instructions

- This data path can execute any sequence of **`lw`**, **`sw`**, and/or arithmetic-logic with immediate operand instructions.



Data path design

Data path for **add**-like instructions

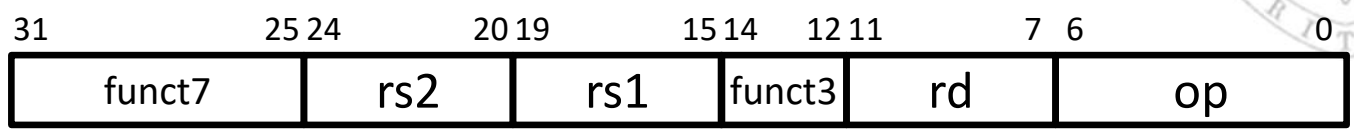


- Part of this data path can be reused to execute **`add/sub/and/or/slt`** instructions

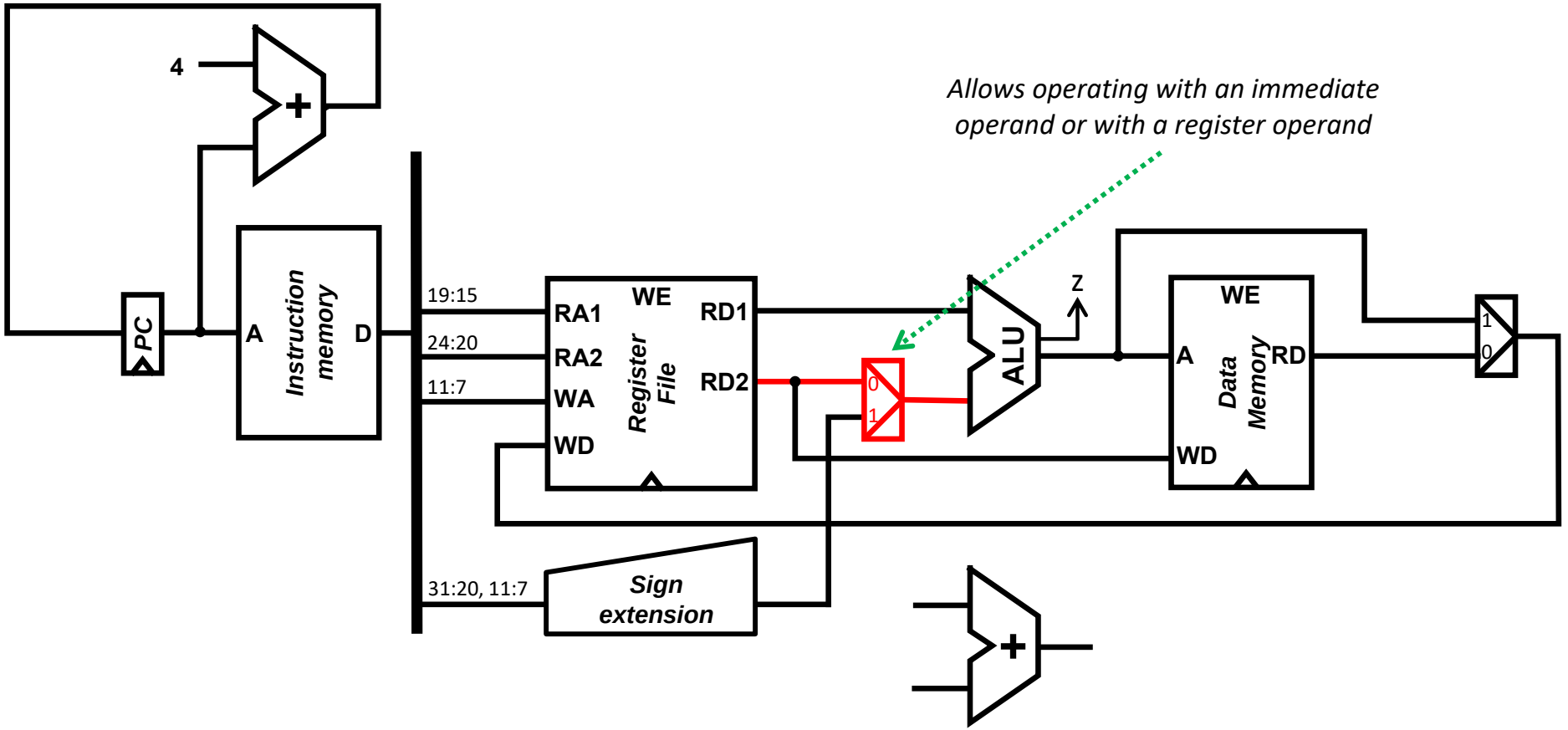
Data path design

`addi`-like instructions: calculating the operation

`add rd, rs1, rs2`



RF[rd] ← RF[rs1] *op* RF[rs2], PC ← PC+4

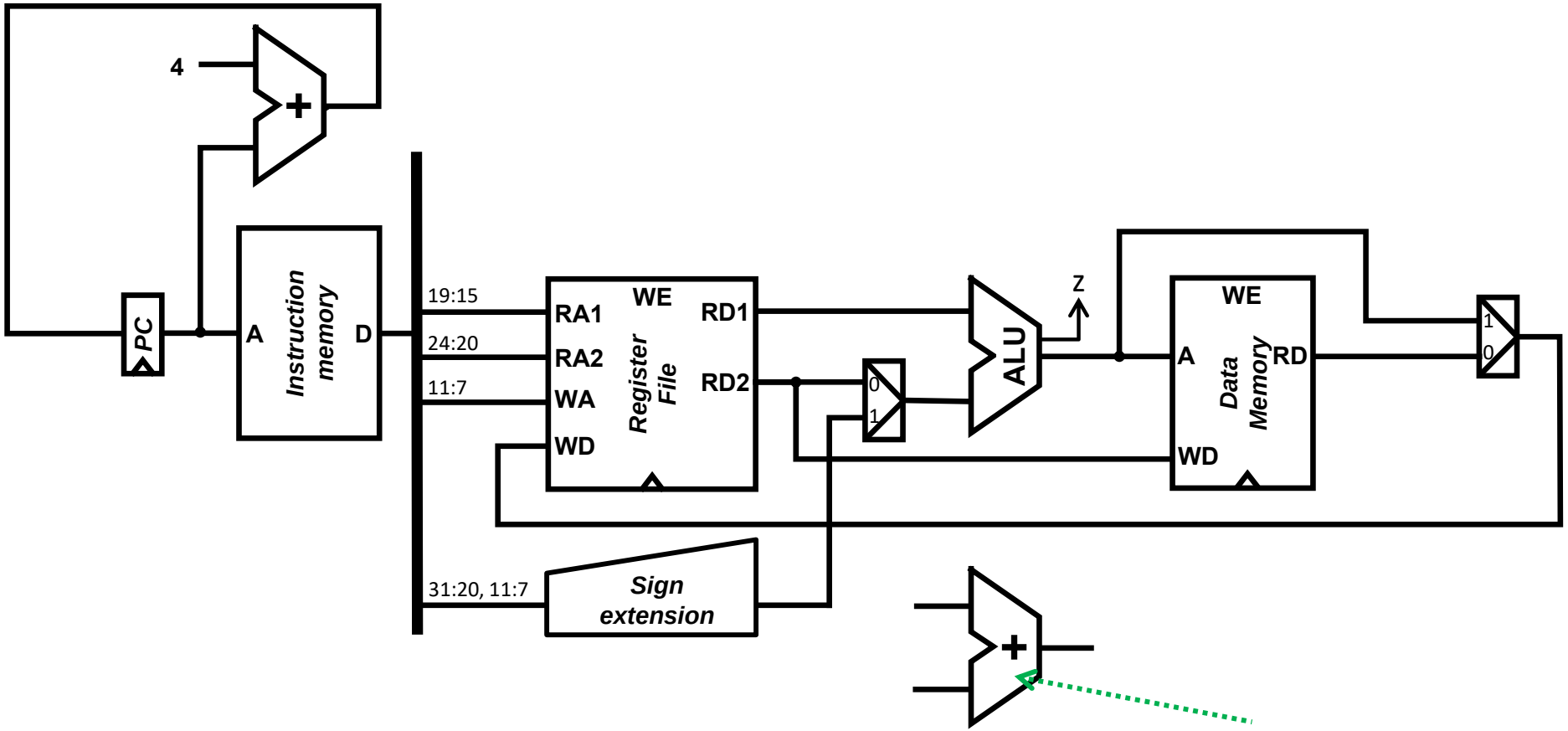


- A multiplexer is added in order to reuse the ALU to perform arithmetic-logic operations with 2 registers

Data path design

Data path for **lw/sw/addi/add** instructions

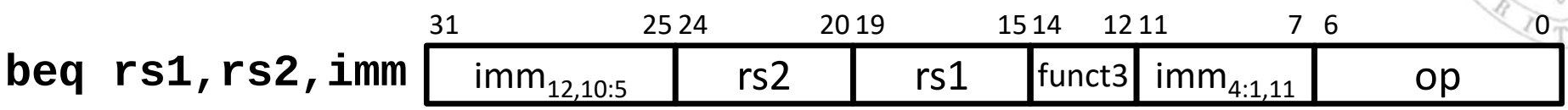
- This data path can execute any sequence of **lw**, **sw**, and/or arithmetic-logic instructions.



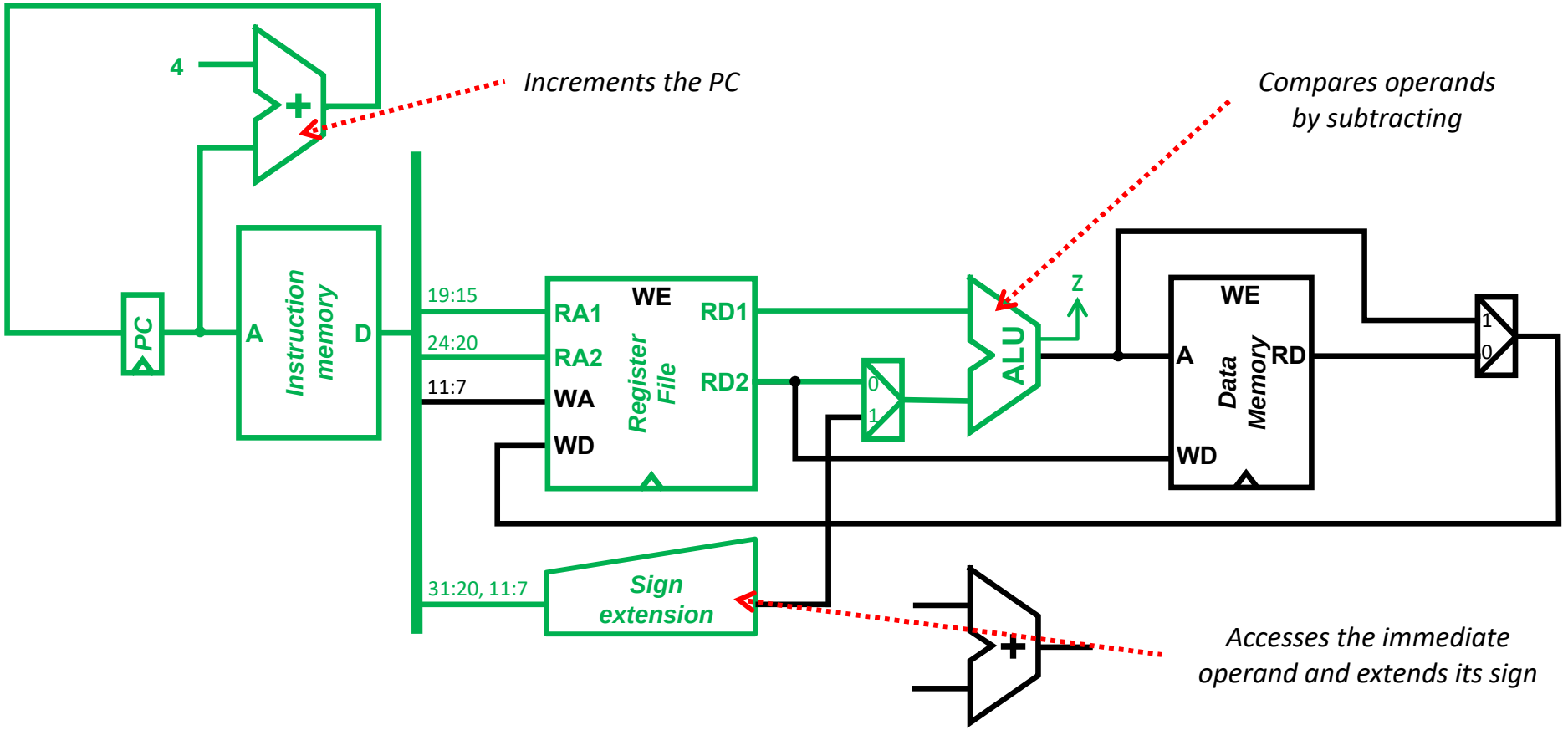
*This adder is only used to calculate branch addresses in **jal/beq** instructions*

Data path design

Data path for **beq** instructions



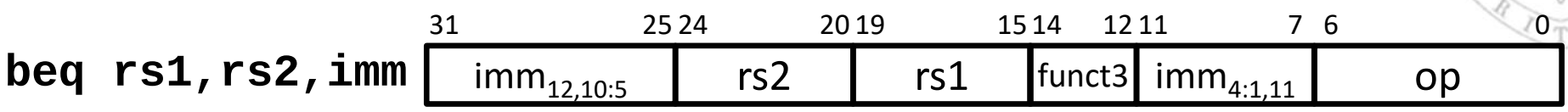
PC ← *if* (RF[rs1] = RF[rs2]) *then* (PC + sExt(imm)) *else* (PC+4)



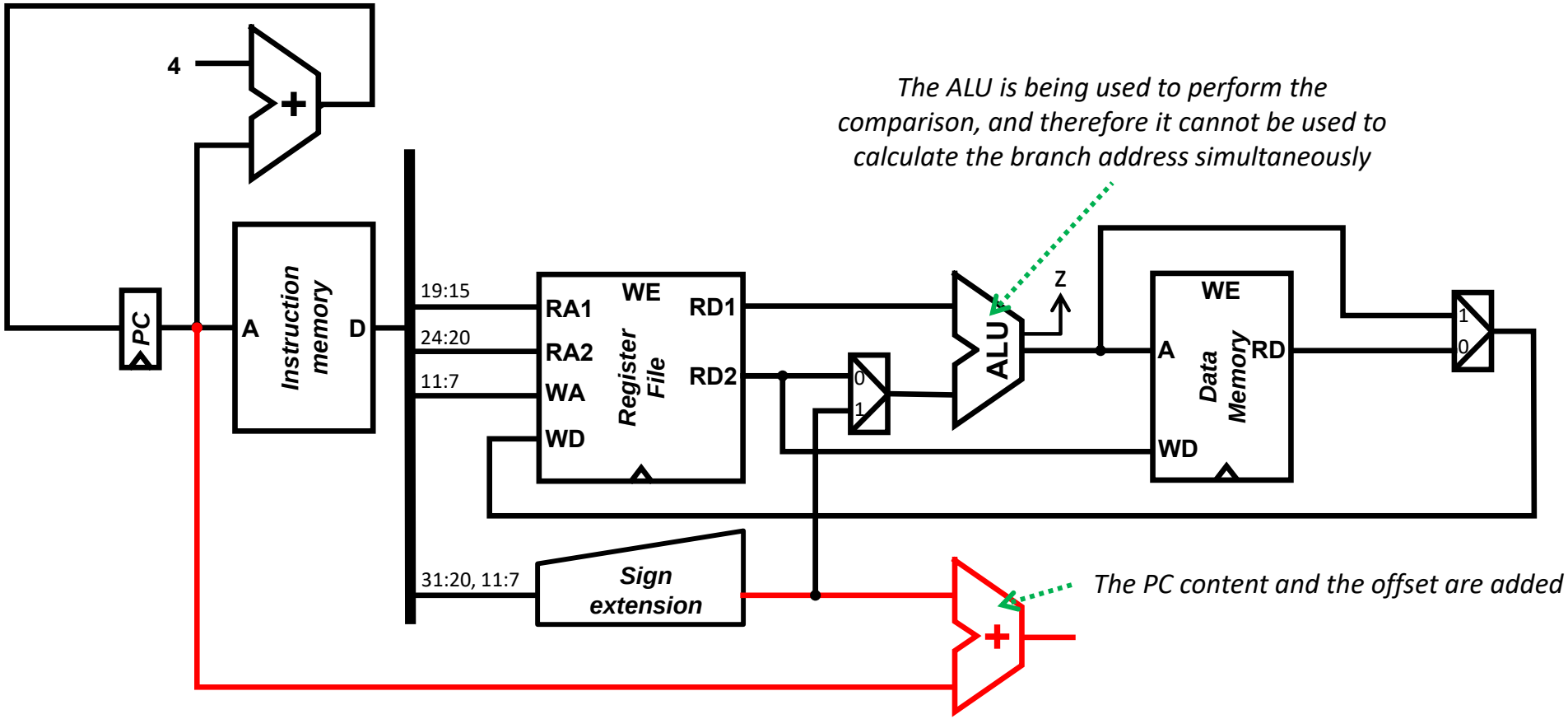
- Part of this data path can be reused to execute **beq** instructions

Data path design

beq instructions: calculating the branch address



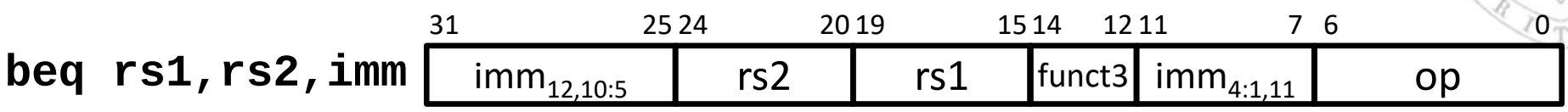
PC ← *if* (RF[rs1] = RF[rs2]) *then* (PC + sExt(imm)) *else* (PC+4)



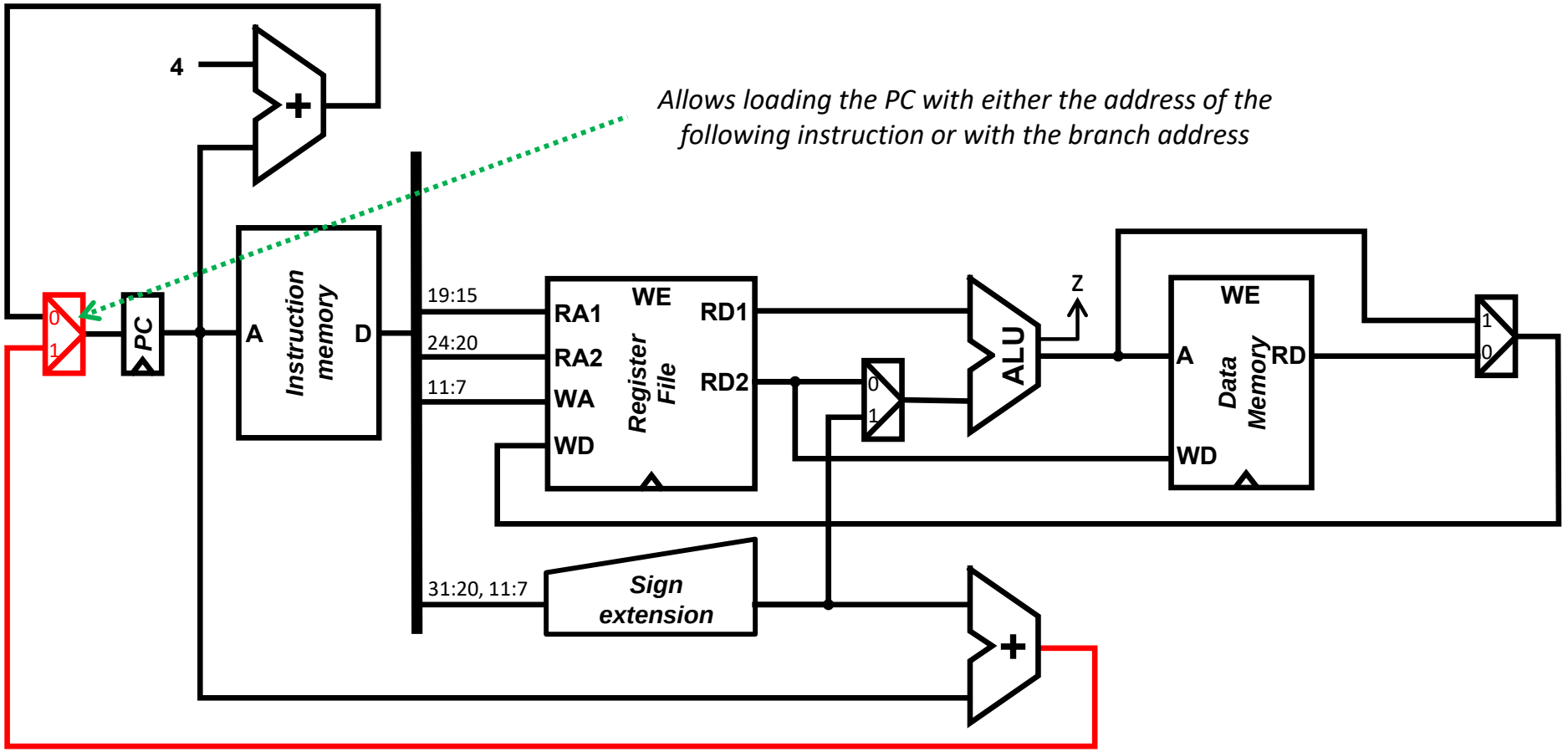
- The PC, the Sign Extension module and the Adder are connected to calculate the branch address.

Data path design

beq instructions: updating the PC



PC ← *if* (RF[rs1] = RF[rs2]) *then* (PC + sExt(imm)) *else* (PC+4)

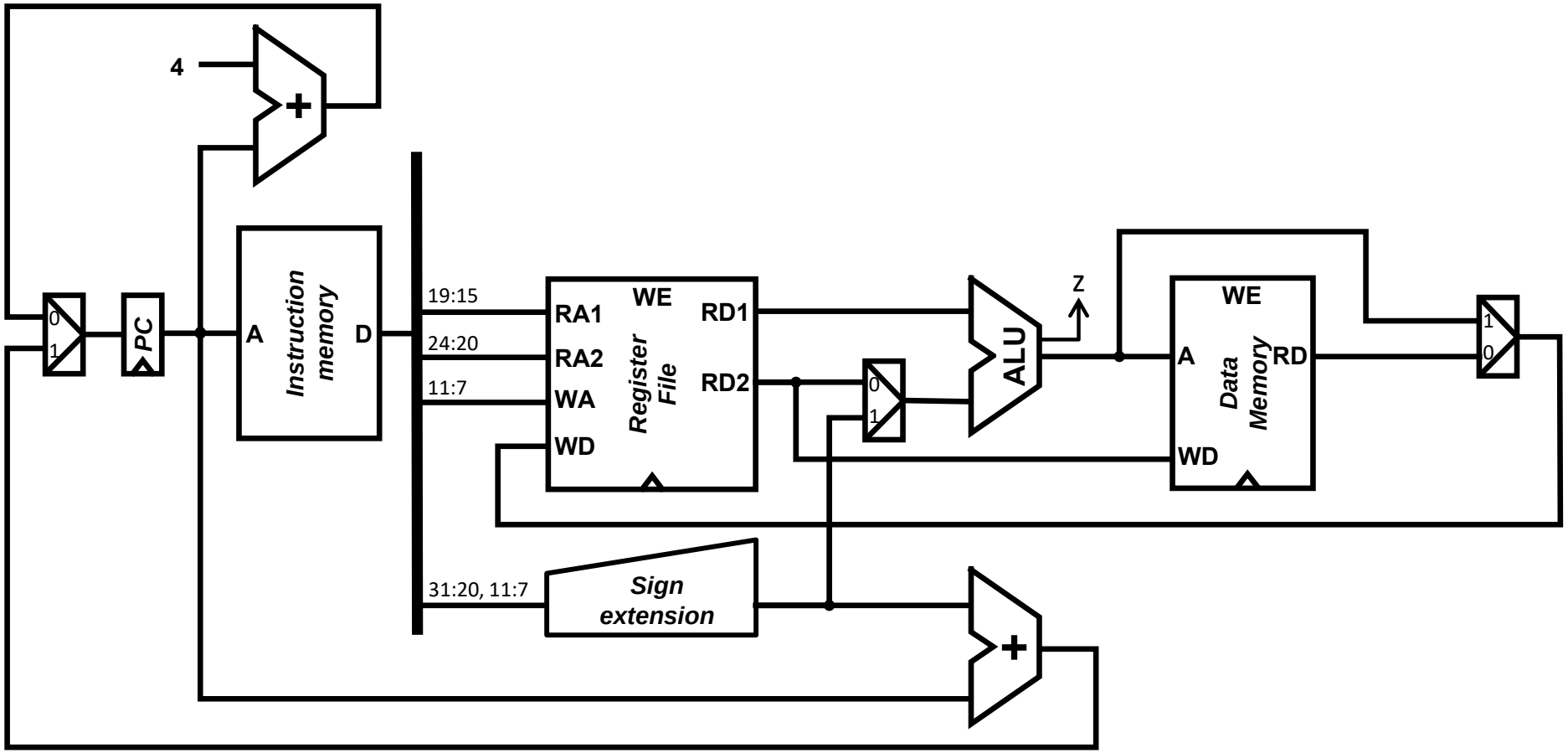


- A multiplexer is added to be able to load the branch address into the PC

Data path design

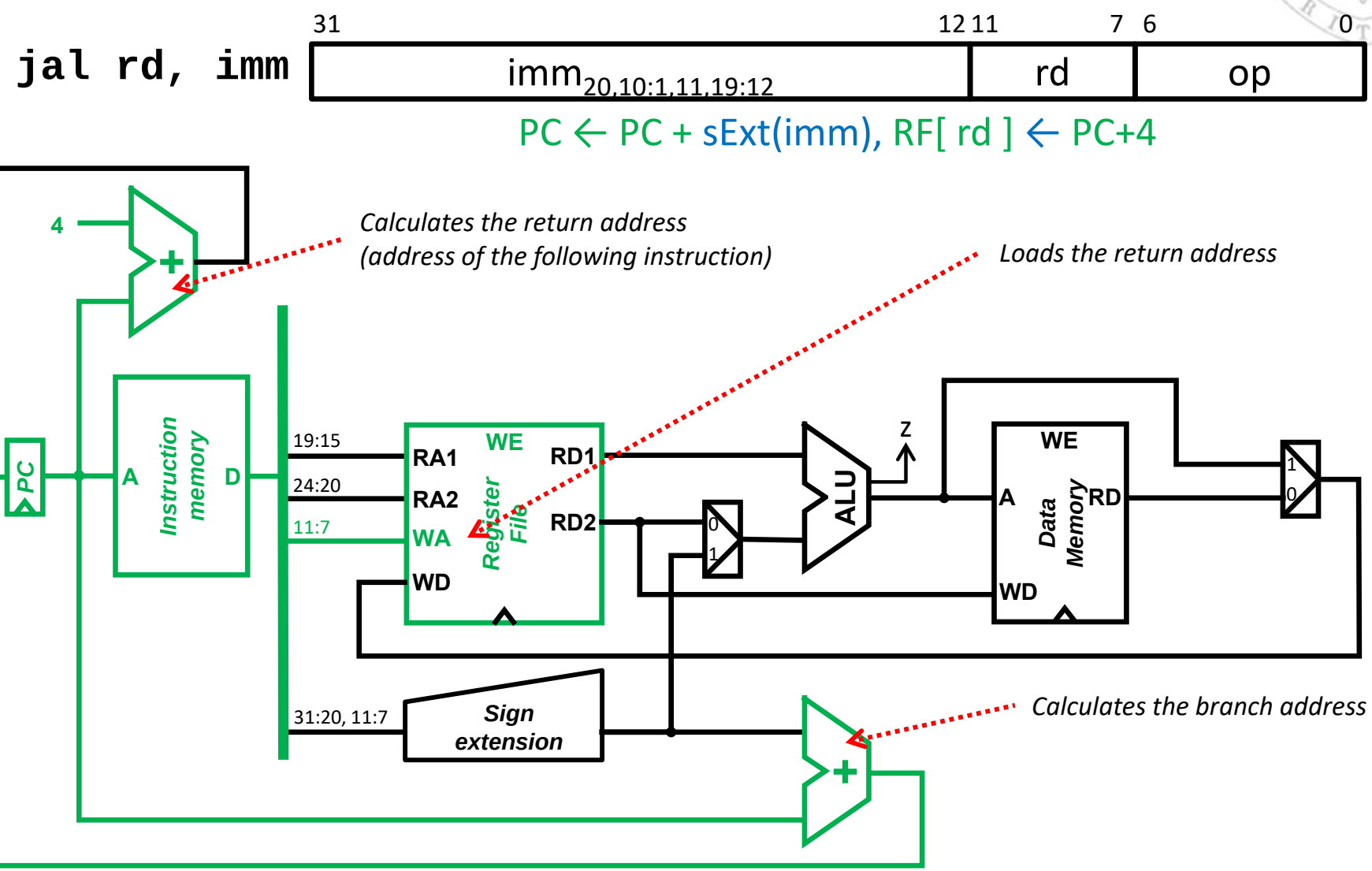
Data path for `lw/sw/addi/add/beq` instructions

- This data path can execute any sequence of **`lw`**, **`sw`**, arithmetic-logic and/or conditional branch instructions.



Data path design

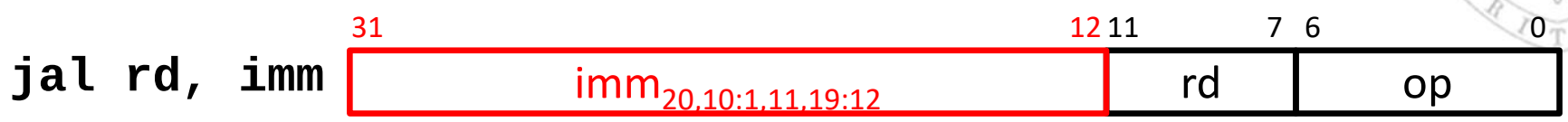
Data path for **jal** instructions



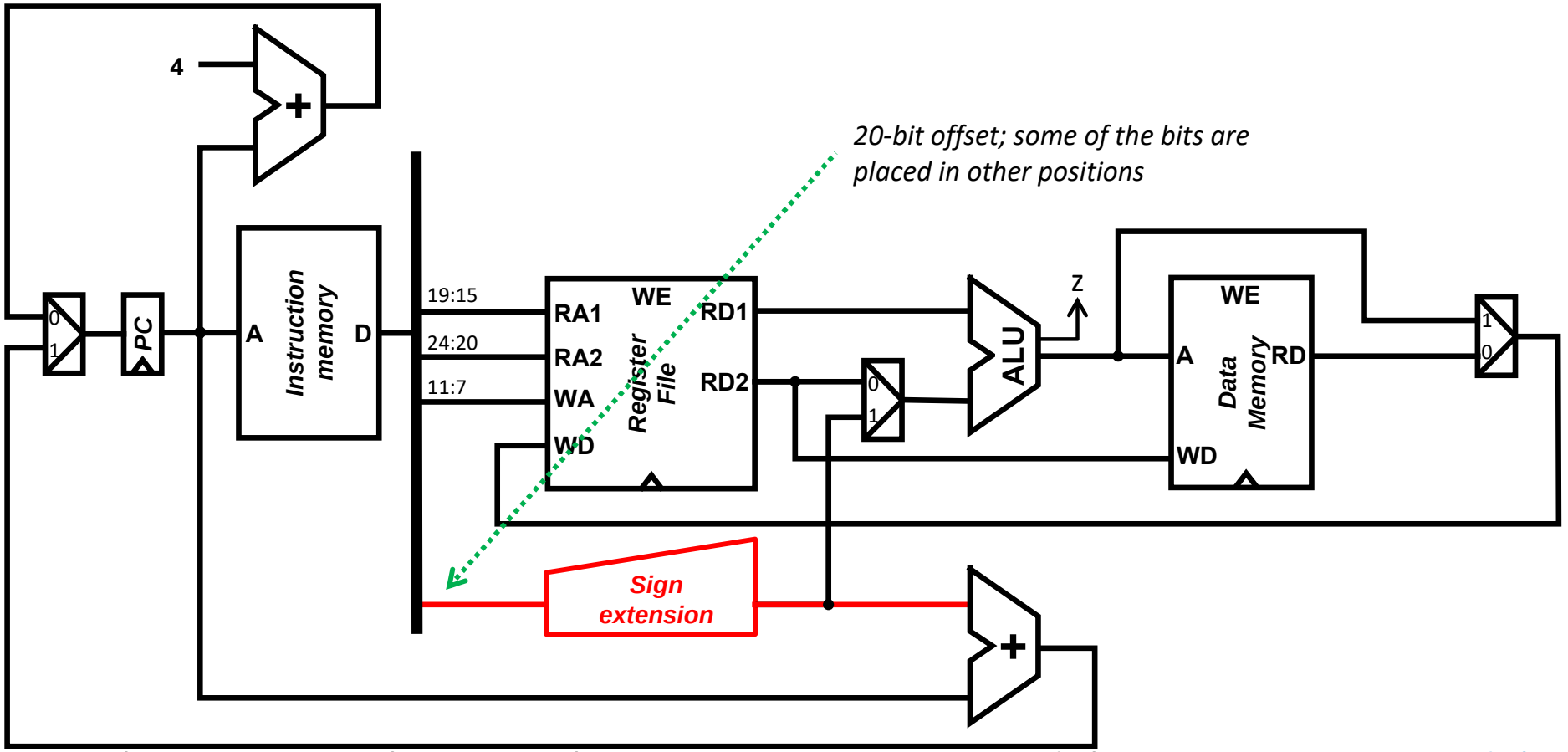
- Part of this data path can be reused to execute **jal** instructions

Data path design

`jal` instructions: calculating the branch address



PC ← PC + sExt(imm), RF[rd] ← PC+4

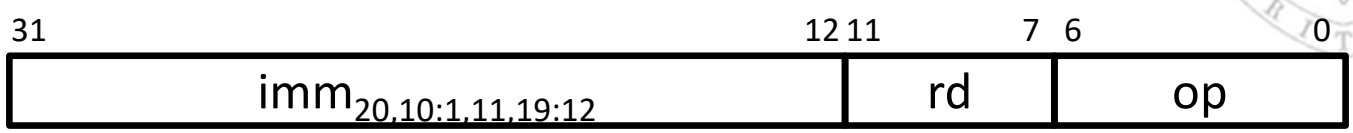


- The connection between the Instruction Memory and the Sign Extension module is extended.

Data path design

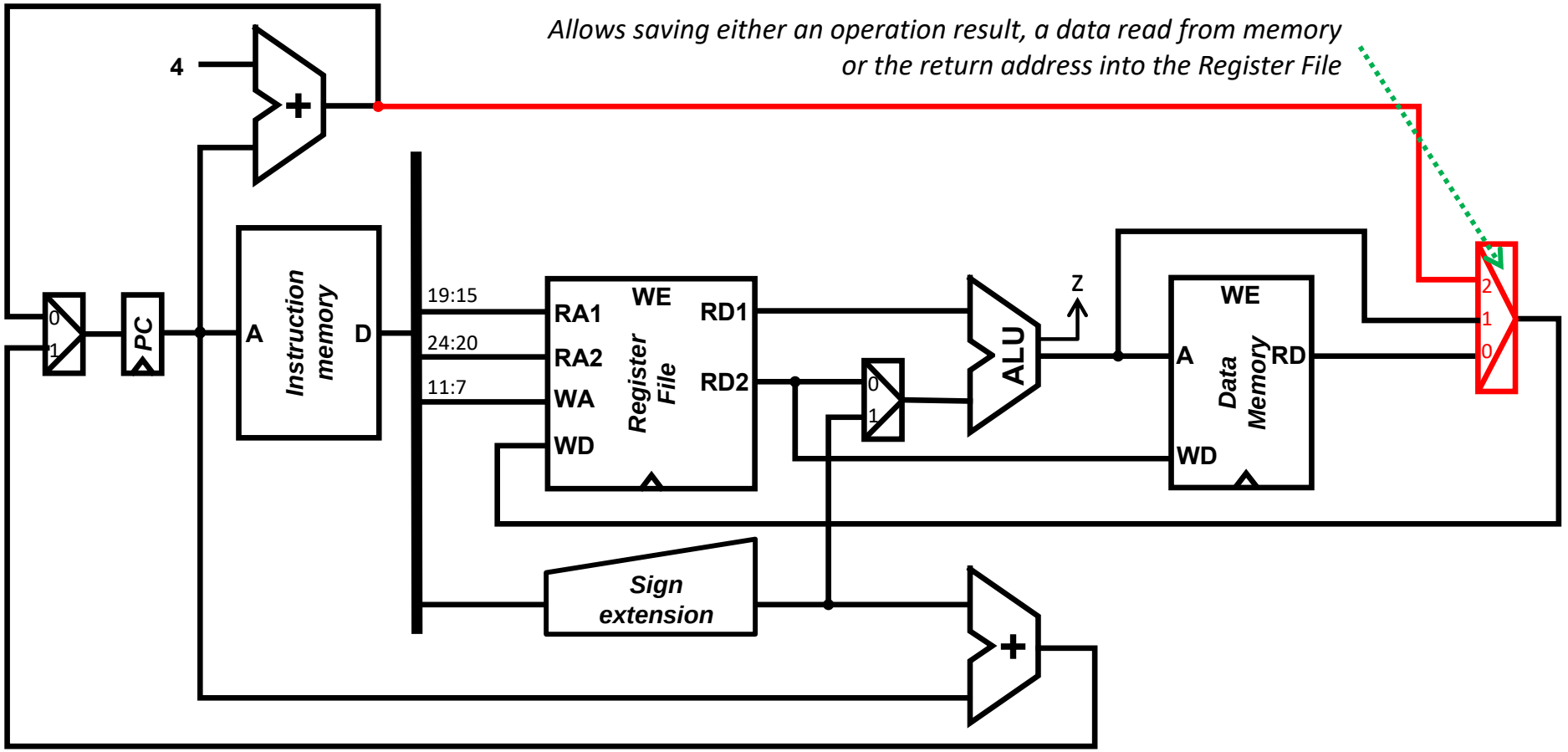
`jal` instructions: saving the return address

`jal rd, imm`



PC ← PC + sExt(imm), RF[rd] ← PC+4

Allows saving either an operation result, a data read from memory or the return address into the Register File

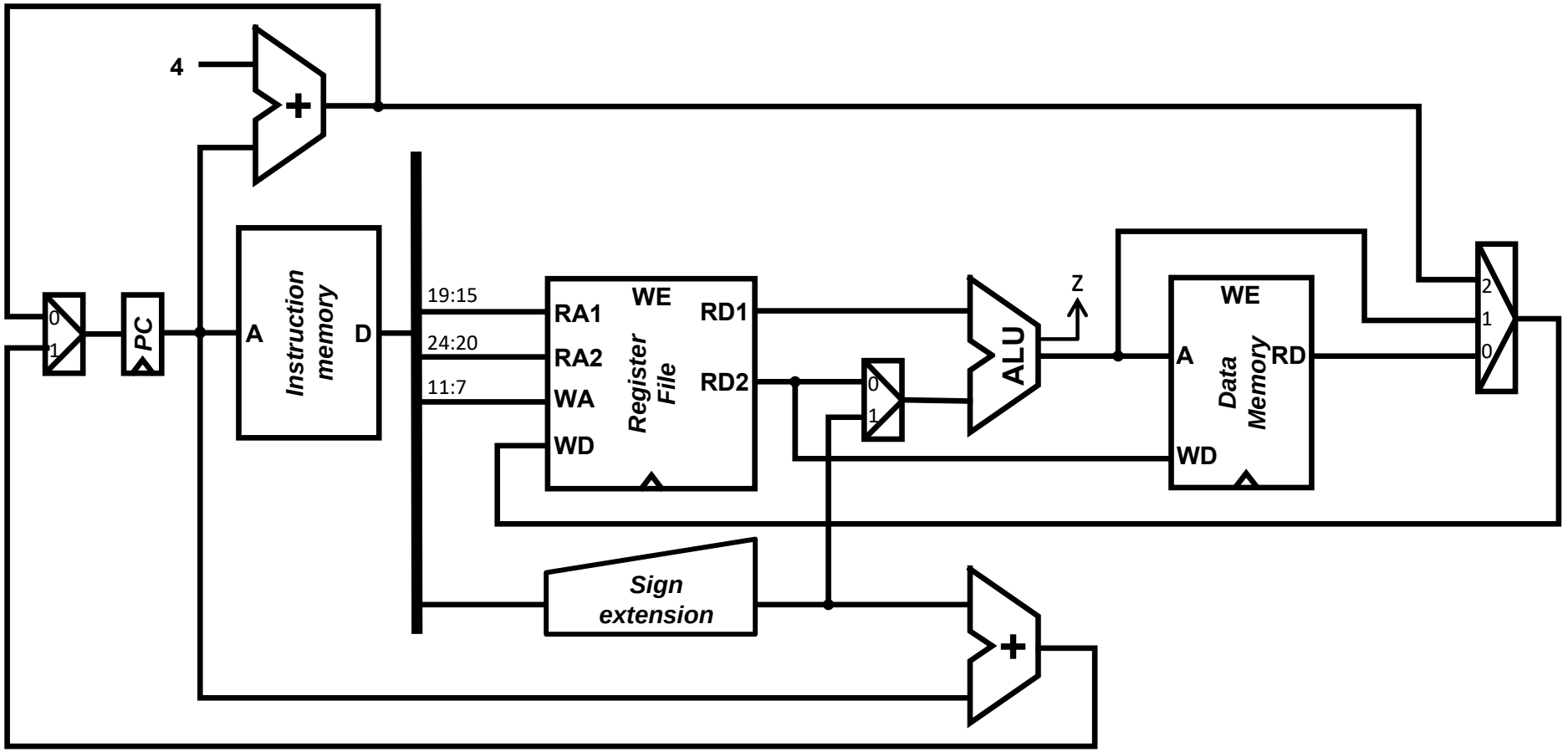


- The multiplexer is extended to be able to save the return address into rd.

Data path design

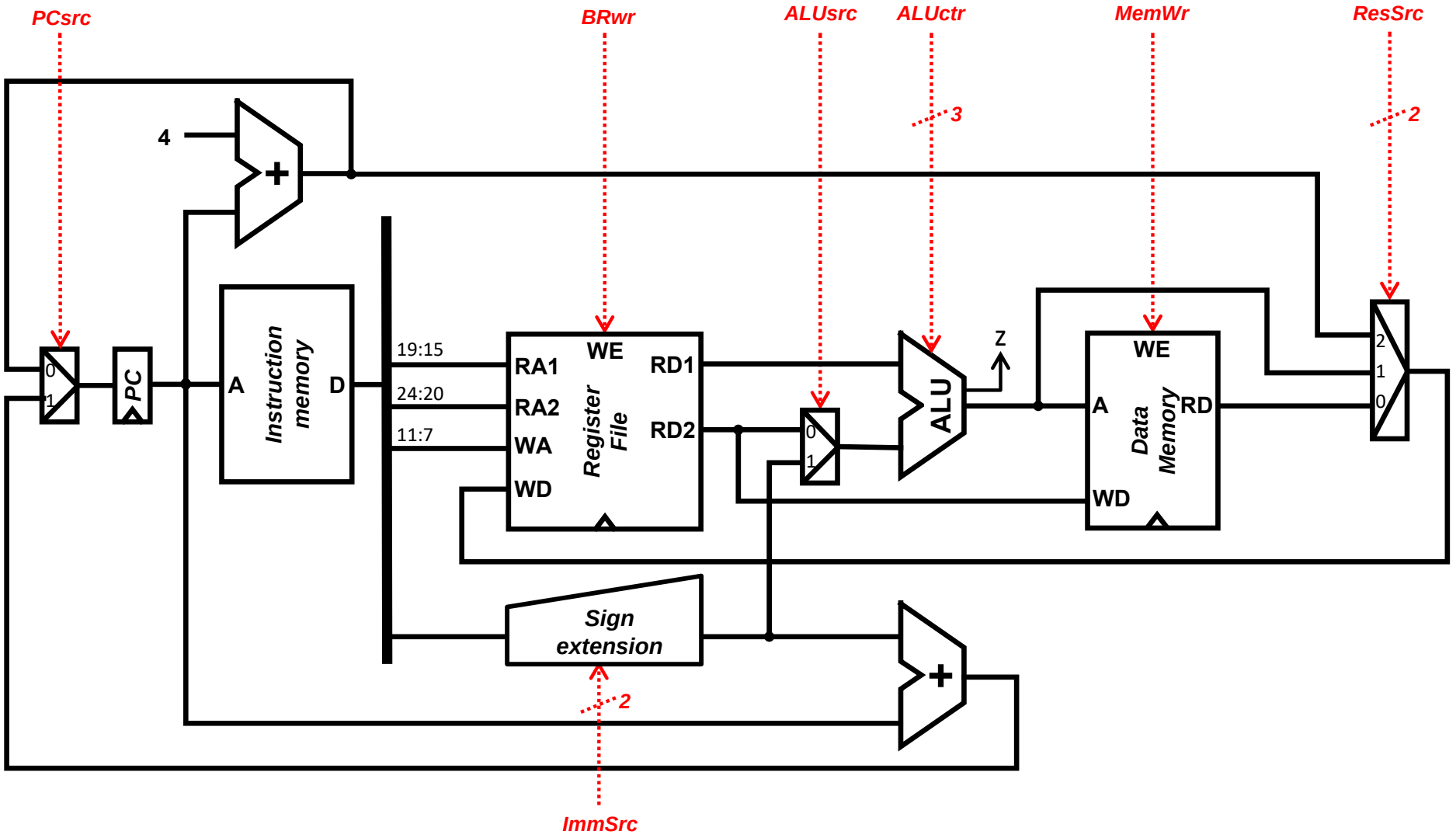
Reduced RISC-V data path

- This data path can execute any sequence of instructions of the RISC-V reduced ISA.



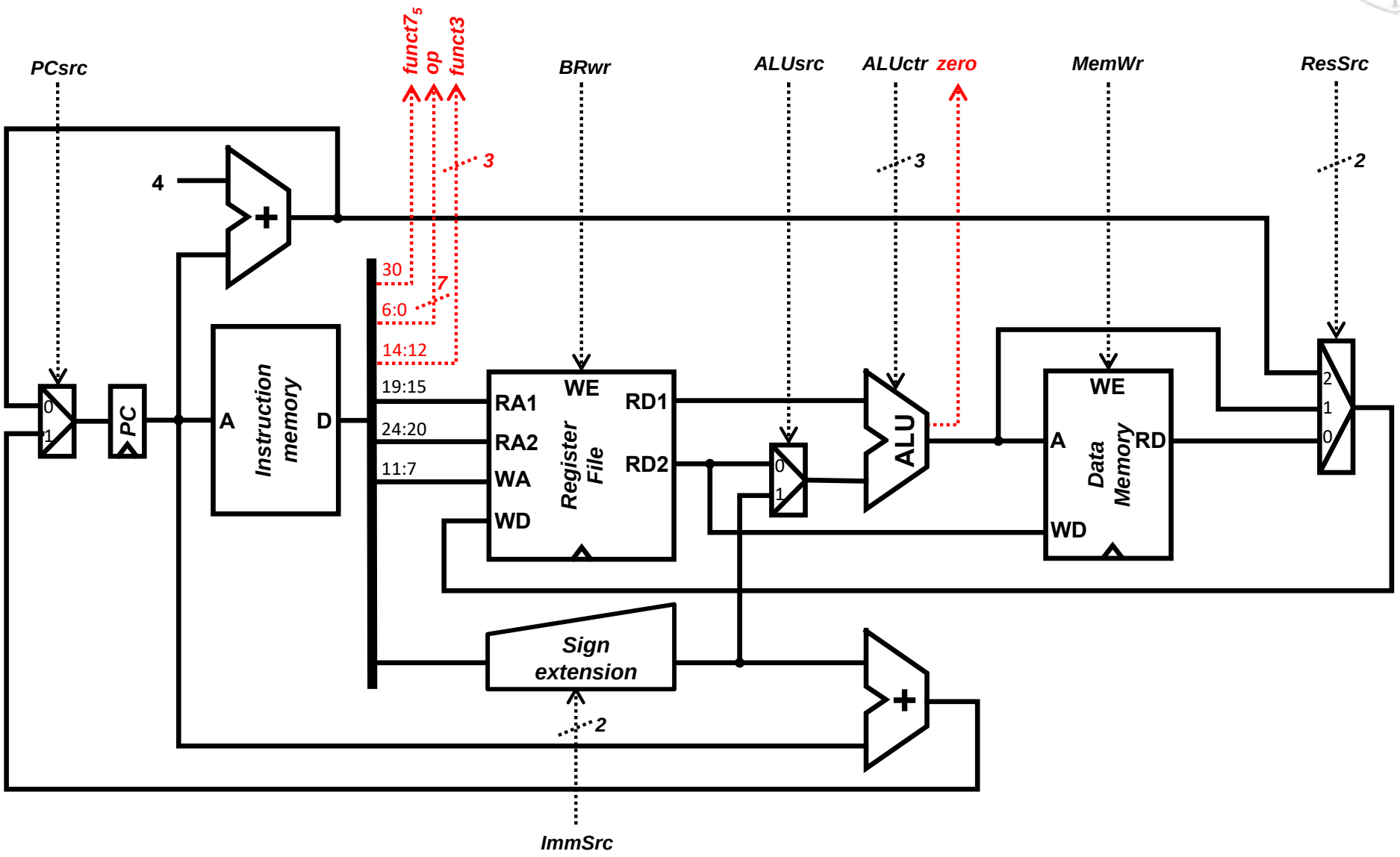
Data path design

Control signals



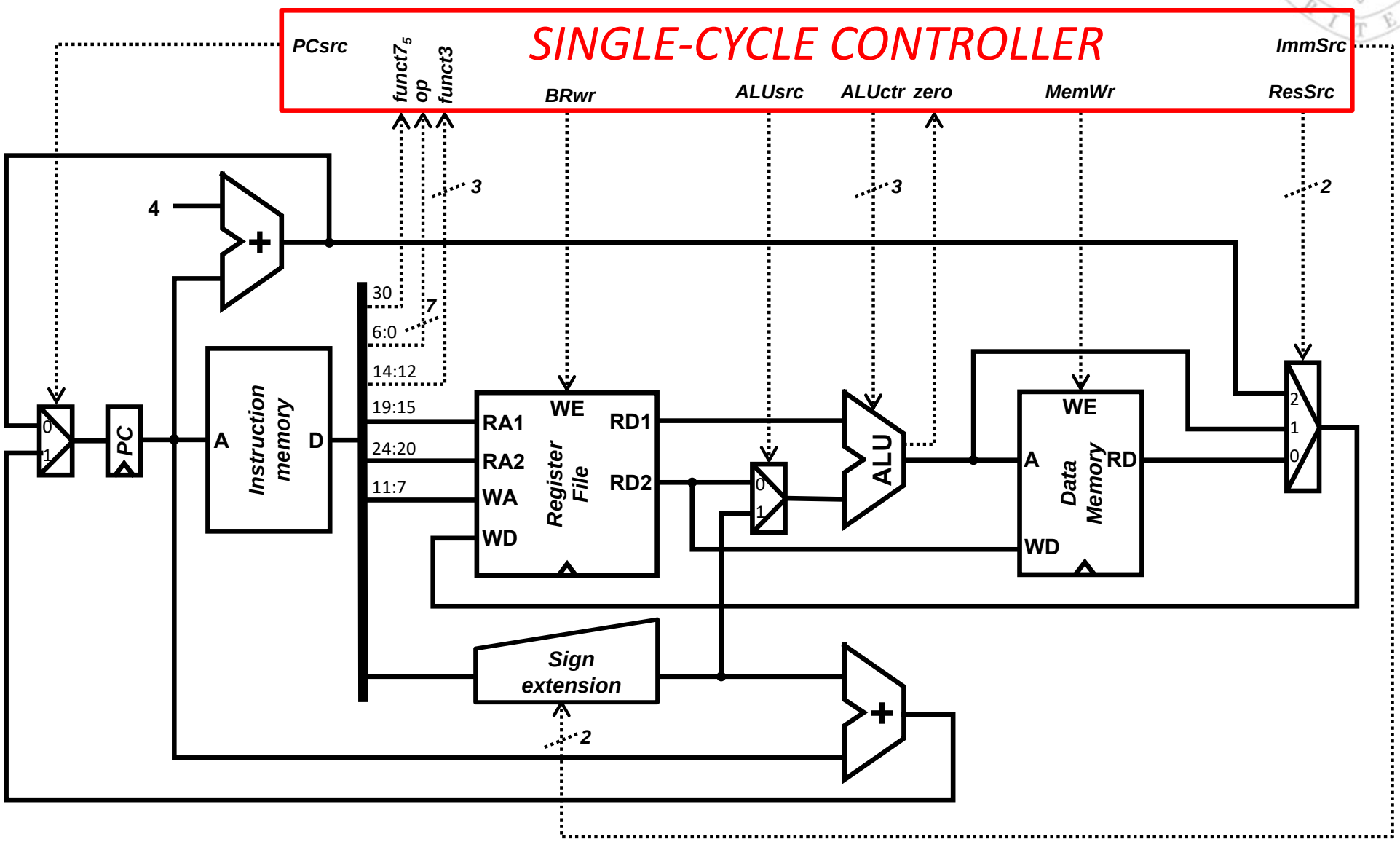
Data path design

Status signals



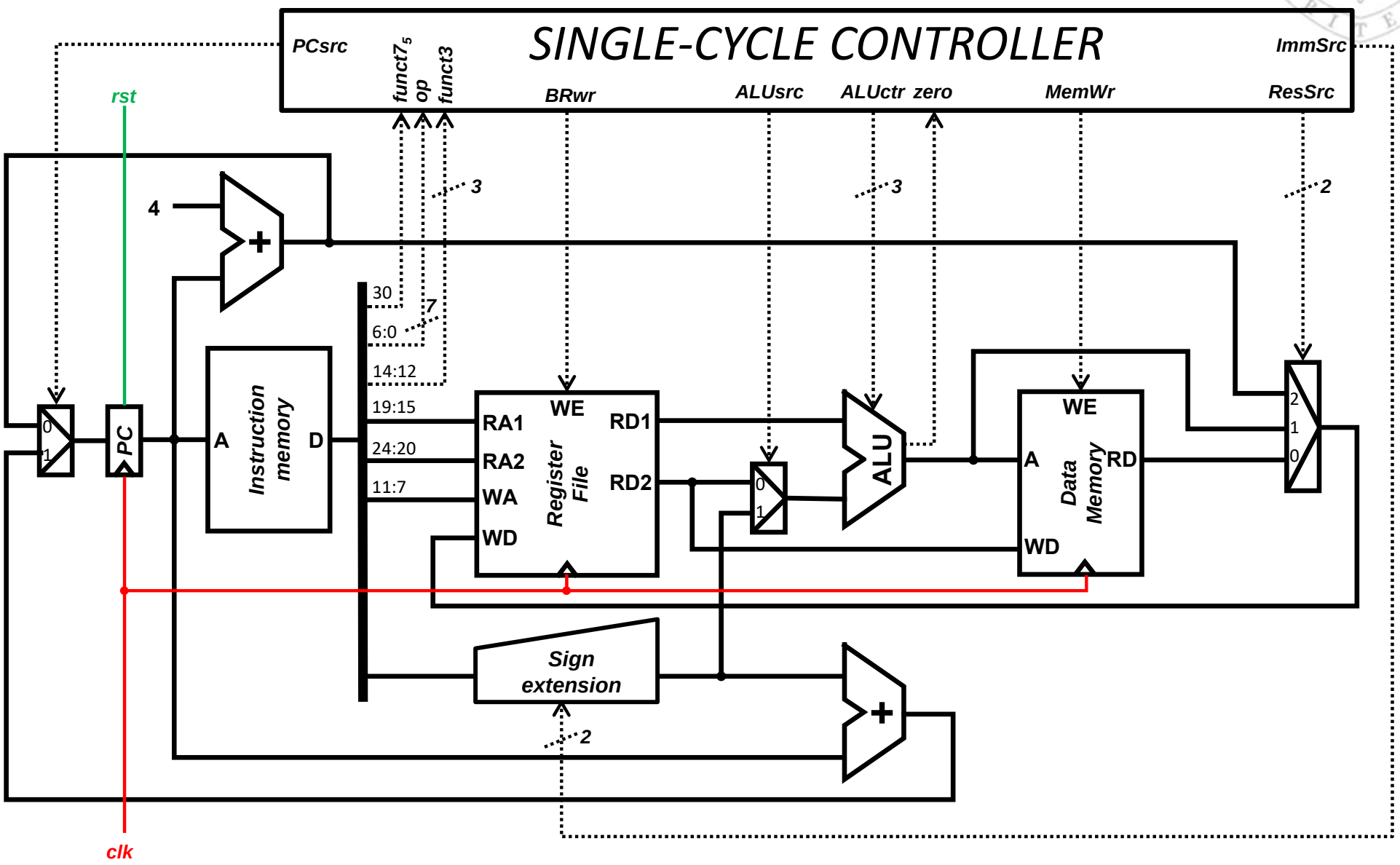
Data path design

Connection with the controller



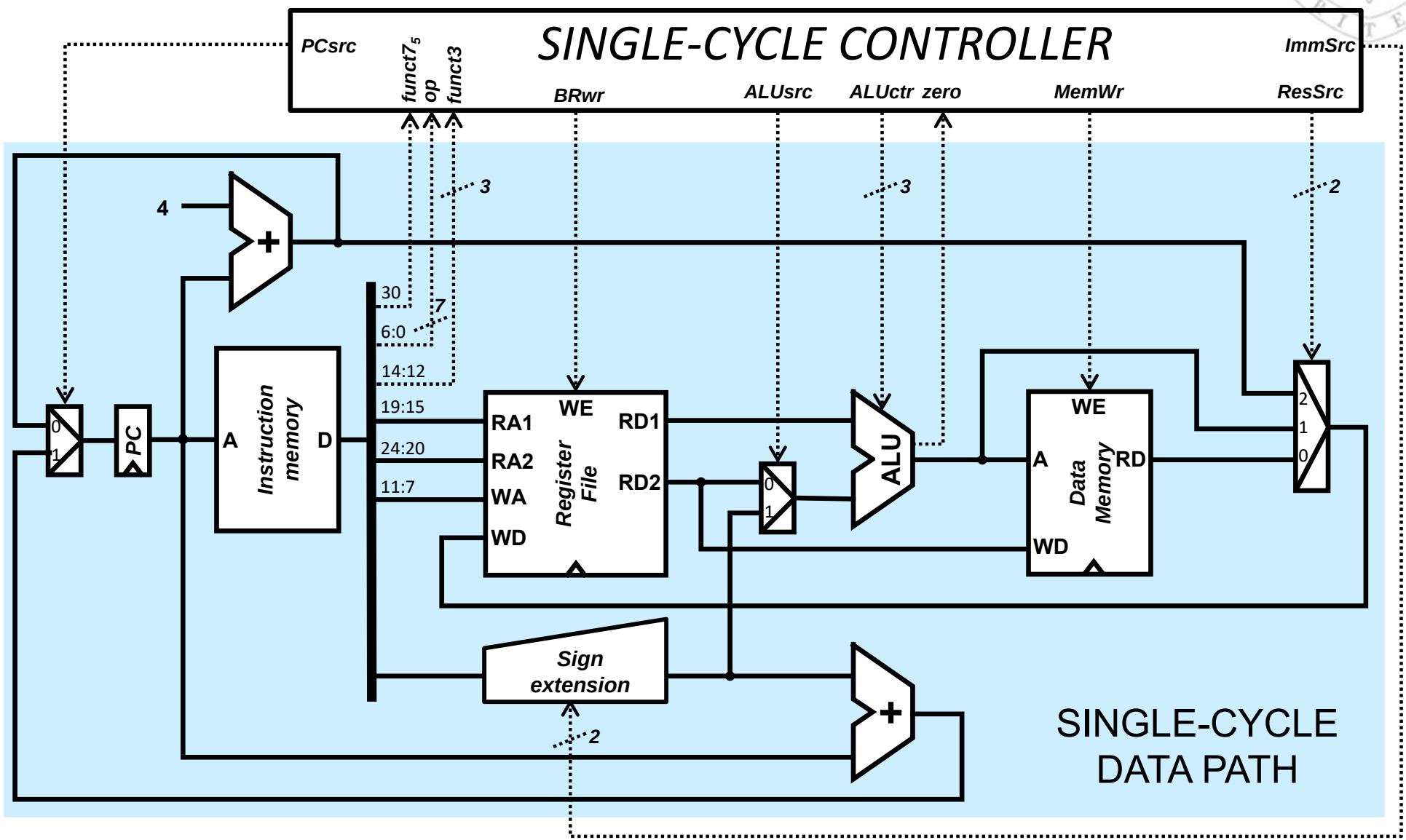
Data path design

Connection with the clock and reset



Data path design

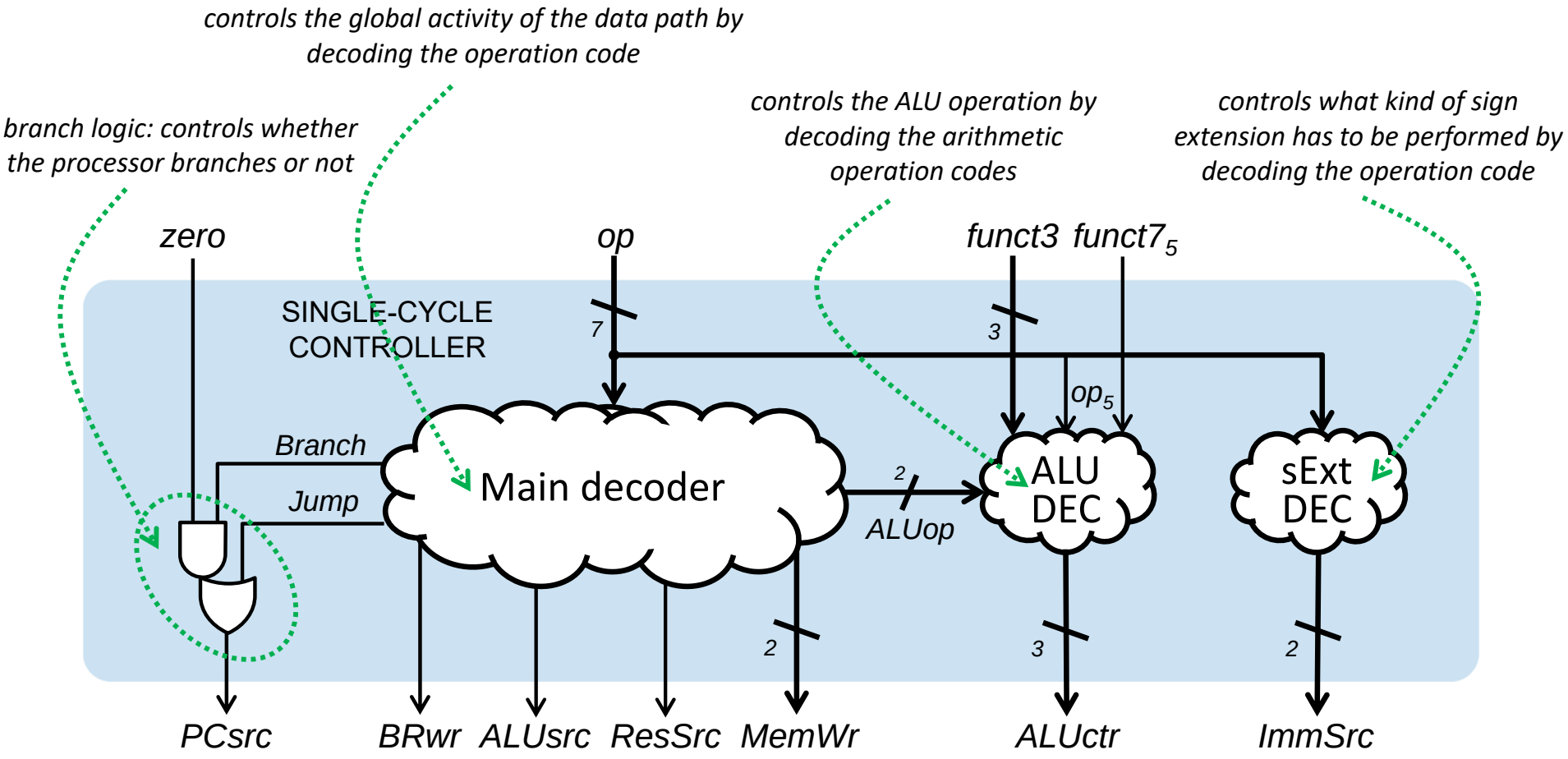
Full system



Controller design

Controller structure (i)

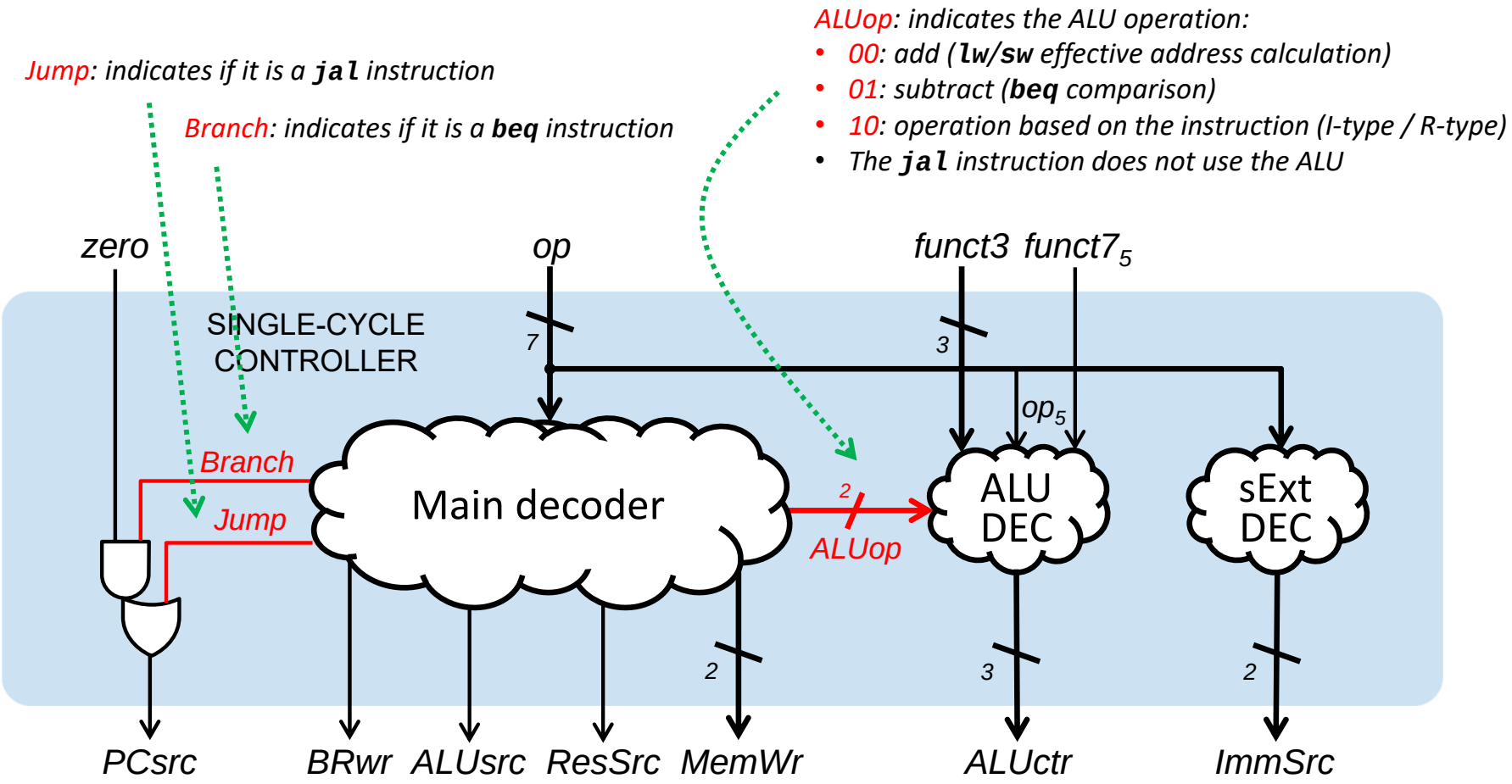
- In the single-cycle processor, the controller is a combinational circuit:
  - It is composed of 4 subcircuits to facilitate its design:



Controller design

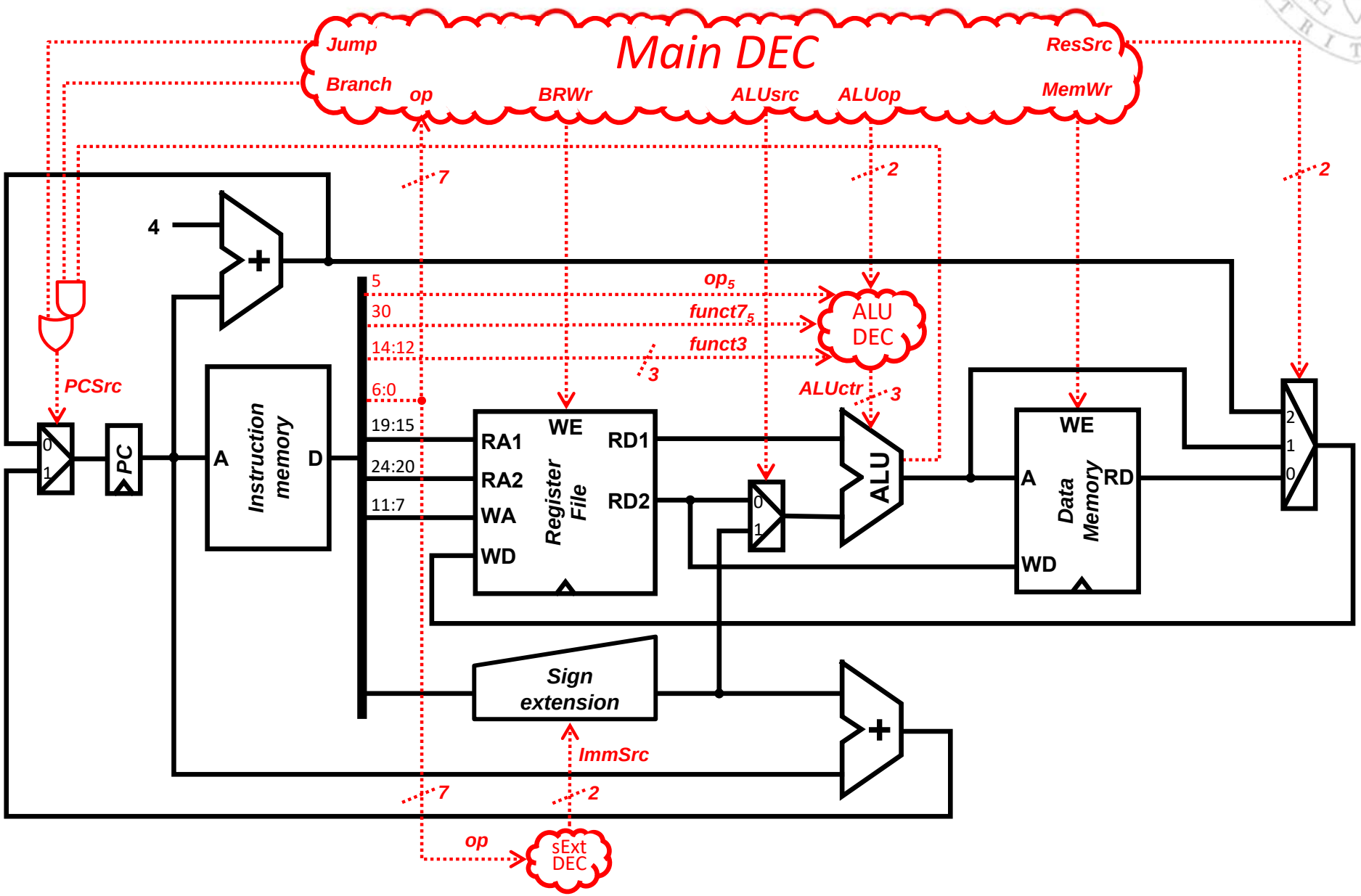
Controller structure (ii)

- In the single-cycle processor, the controller is a combinational circuit:
  - It is composed of 4 subcircuits to facilitate its design:



Controller design

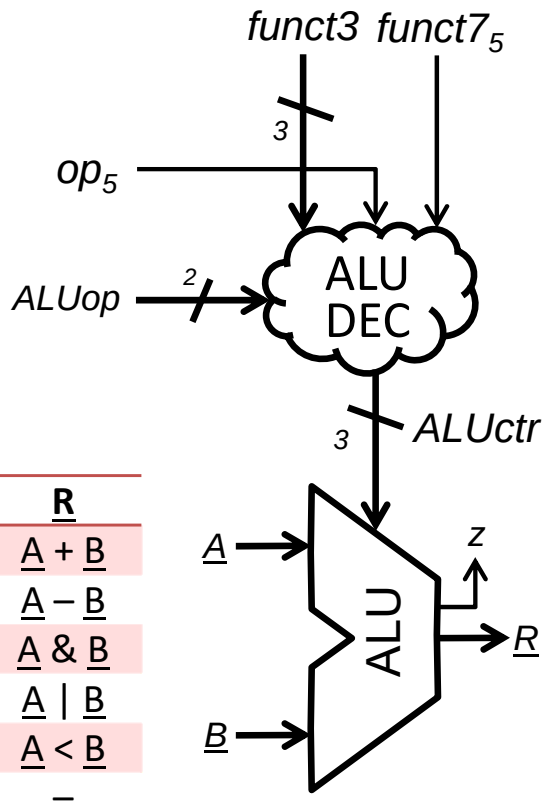
Controller structure (iii)



Controller design

Controller design: ALU local DEC

- This circuit indicates what type of operation is performed by the ALU.
  - Adapts the instruction operation encoding to the ALU operation encoding



ALUctr	R
000	A + B
001	A – B
010	A & B
011	A | B
101	A < B
others	–

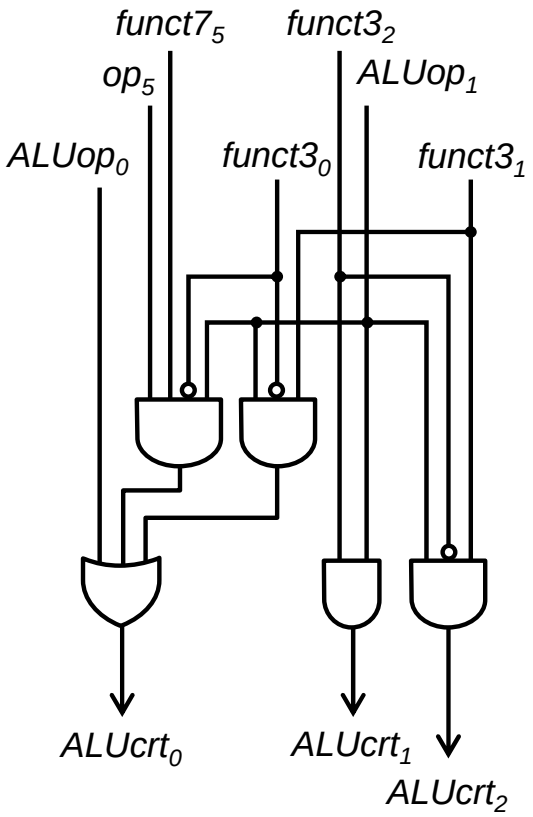
Truth table

ALUop	op_5	$funct7_5$	funct3	ALUctr
00 (add)	X	X	XXX	000 (A + B)
01 (subtract)	X	X	XXX	001 (A – B)
10 (operate)	0	X	000 (addi)	000 (A + B)
10 (operate)	1	0	000 (add)	000 (A + B)
10 (operate)	1	1	000 (sub)	001 (A – B)
10 (operate)	X	X	010 (slt/slti)	101 (A < B)
10 (operate)	X	X	110 (or/ori)	011 (A | B)
10 (operate)	X	X	111 (and/andi)	010 (A & B)

Controller design

Controller design: ALU local DEC

- This circuit indicates what type of operation is performed by the ALU.
  - Adapts the instruction operation encoding to the ALU operation encoding



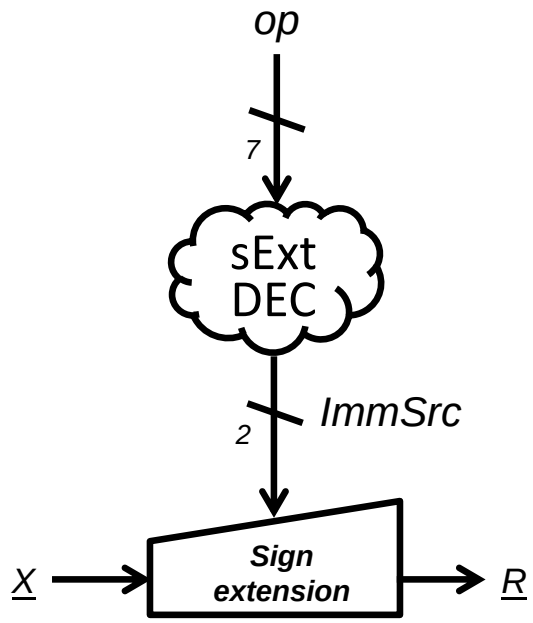
Truth table

ALUop	op$_5$	funct7$_5$	funct3	ALUctr
00 (add)	X	X	XXX	000 (A + B)
01 (subtract)	X	X	XXX	001 (A – B)
10 (operate)	0	X	000 (addi)	000 (A + B)
10 (operate)	1	0	000 (add)	000 (A + B)
10 (operate)	1	1	000 (sub)	001 (A – B)
10 (operate)	X	X	010 (slt/slti)	101 (A < B)
10 (operate)	X	X	110 (or/ori)	011 (A | B)
10 (operate)	X	X	111 (and/andi)	010 (A & B)

Controller design

Controller design: Sign Extension local DEC

- This circuit indicates what type of extension is performed by the Sign Extension module.



ImmSrc	R
00	sExt(X) I-type
01	sExt(X) S-type
10	sExt(X) B-type
11	sExt(X) J-type

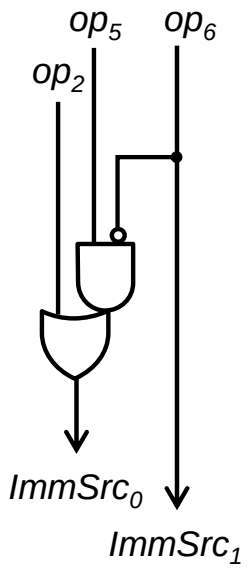
Truth table

Op	ImmSrc
0000011 (lw)	00 (I-type)
0100011 (sw)	01 (S-type)
0010011 (I-type)	00 (I-type)
0110011 (R-type)	–
1100011 (beq)	10 (B-type)
1101111 (jal)	11 (I-type)

Controller design

Controller design: Sign Extension local DEC

- This circuit indicates what type of extension is performed by the Sign Extension module.



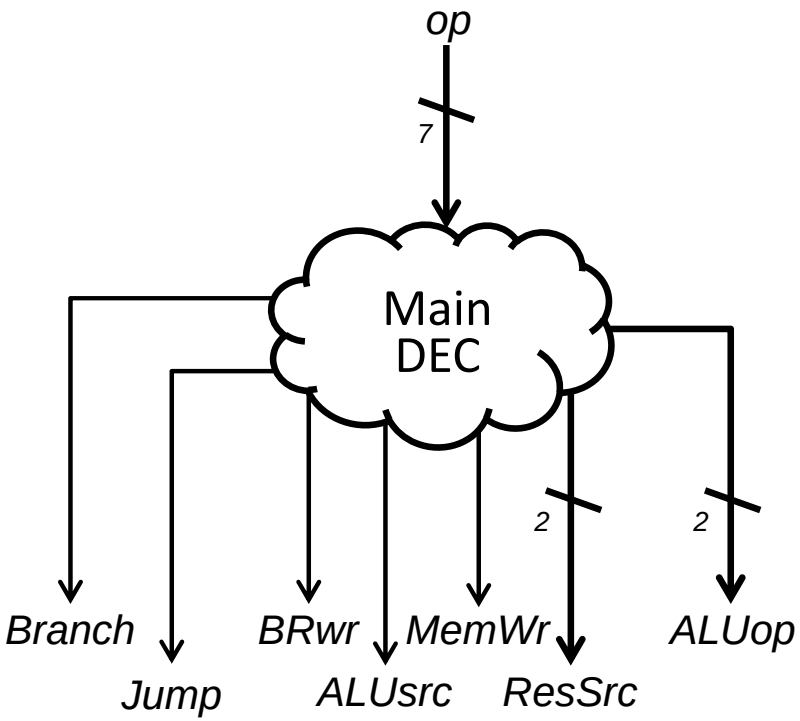
Truth table

Op	ImmSrc
0000011 (`lw`)	00 (I-type)
0100011 (`sw`)	01 (S-type)
0010011 (I-type)	00 (I-type)
0110011 (R-type)	–
1100011 (`beq`)	10 (B-type)
1101111 (`jal`)	11 (I-type)

Controller design

Controller design: main DEC

- This subcircuit rules the general behavior of the processor.



Truth table

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0000011 (lw)							
0100011 (sw)							
0010011 (I-type)							
0110011 (R-type)							
1100011 (beq)							
1101111 (jal)							

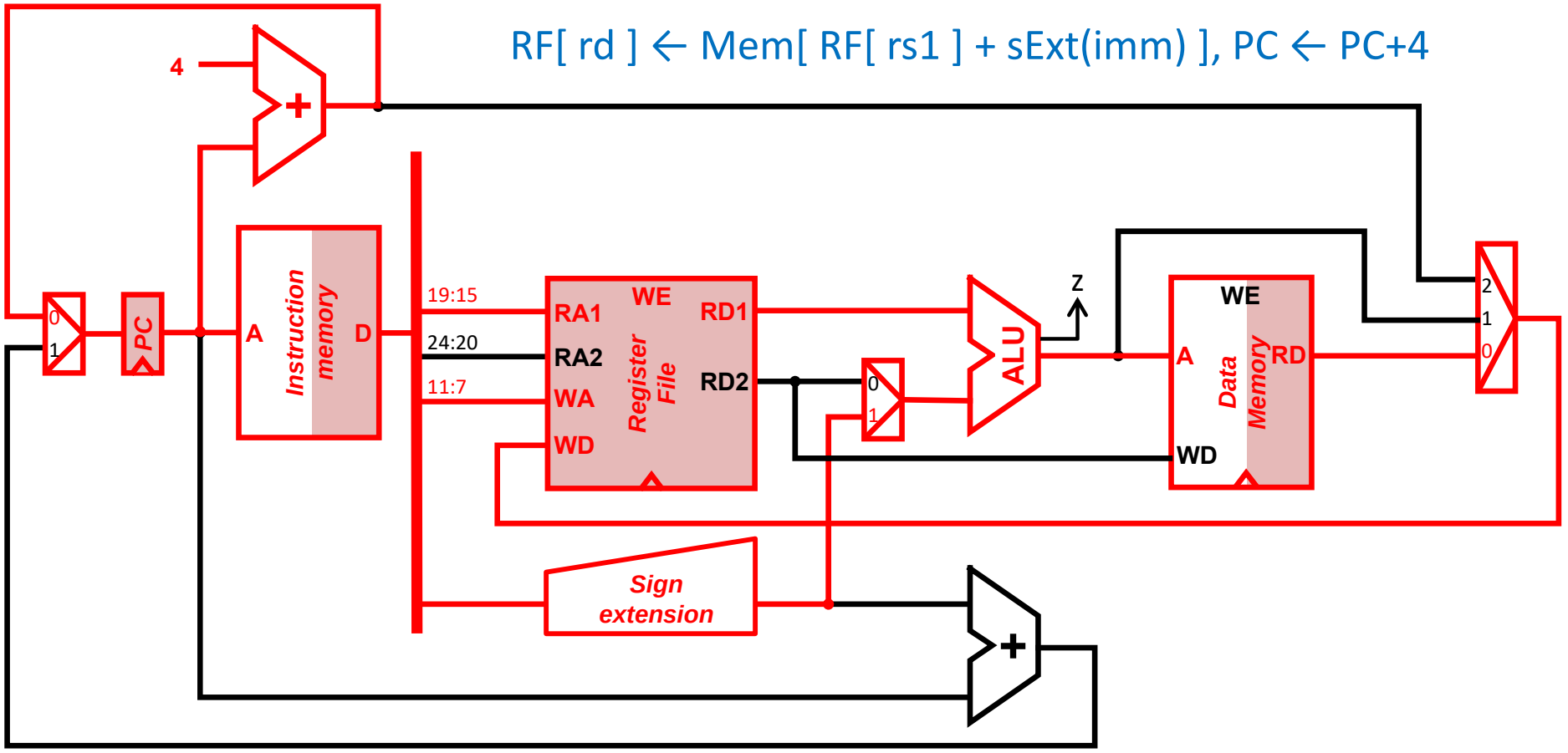
Controller design

Controller design: main DEC

```
lw rd, imm(rs1)
```

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0000011 (lw)							

RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4



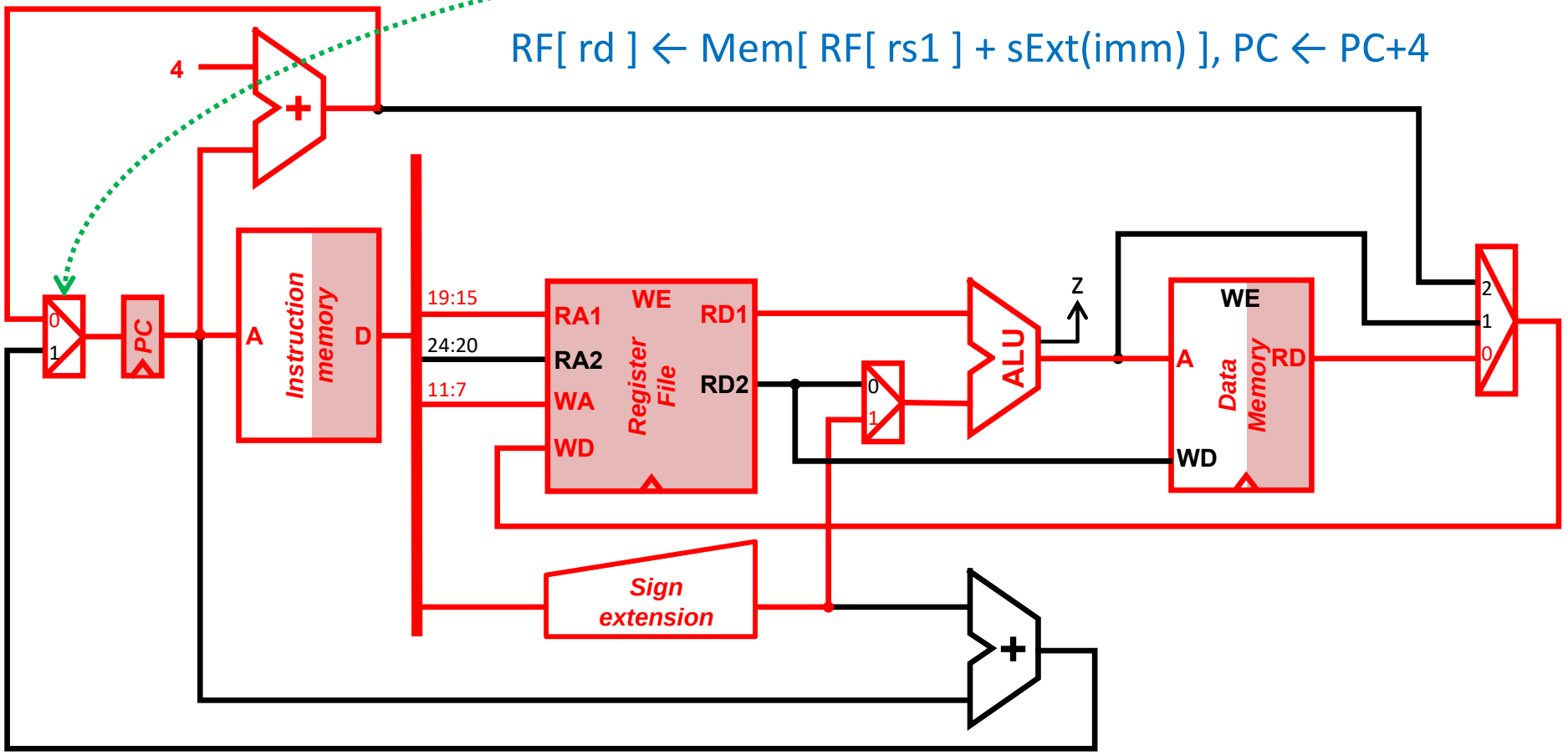
Controller design

Controller design: main DEC

```
lw rd, imm(rs1)
```

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0000011 (lw)	0	0					

RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4



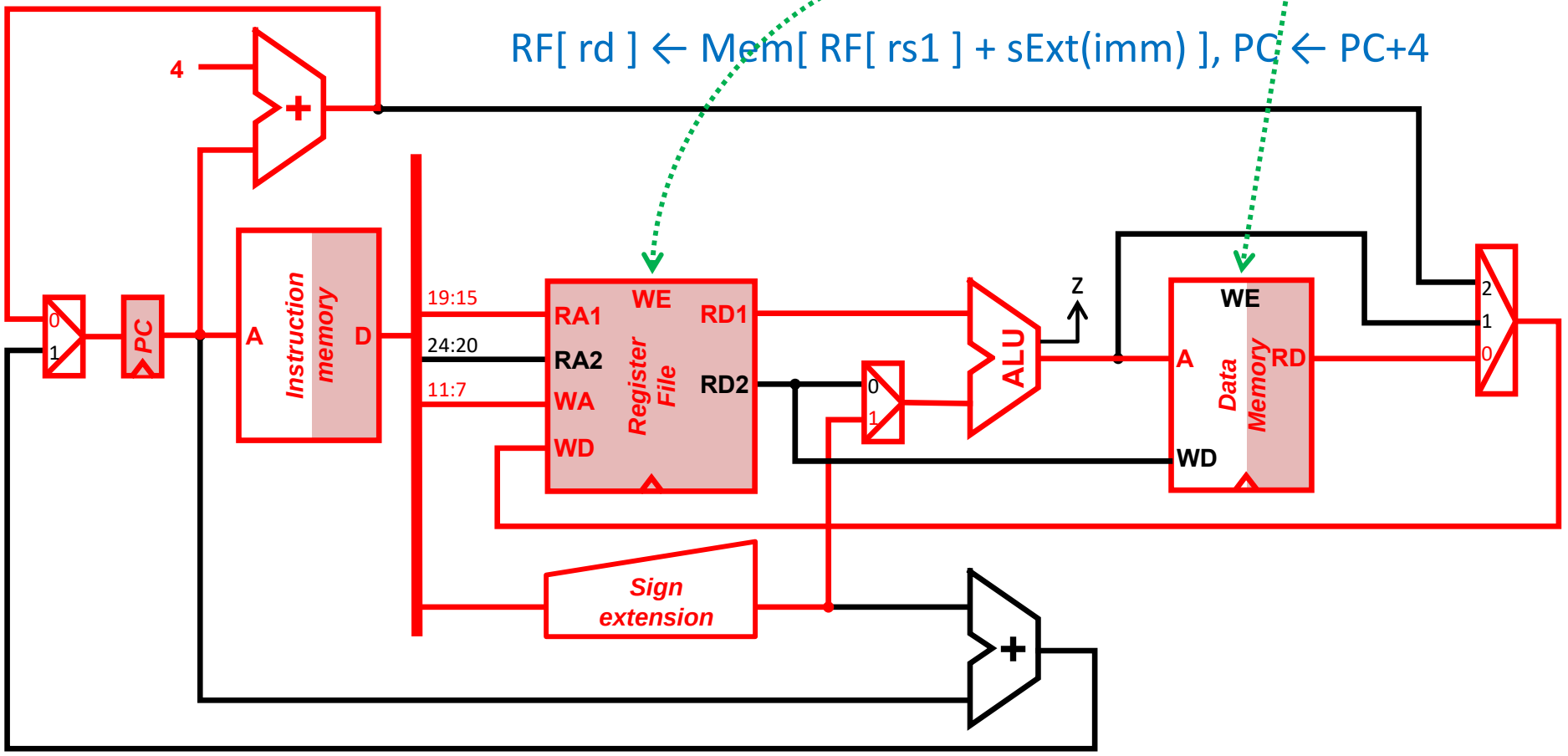
Controller design

Controller design: main DEC

```
lw rd, imm(rs1)
```

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0000011 (lw)	0	0	1			0	

RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4



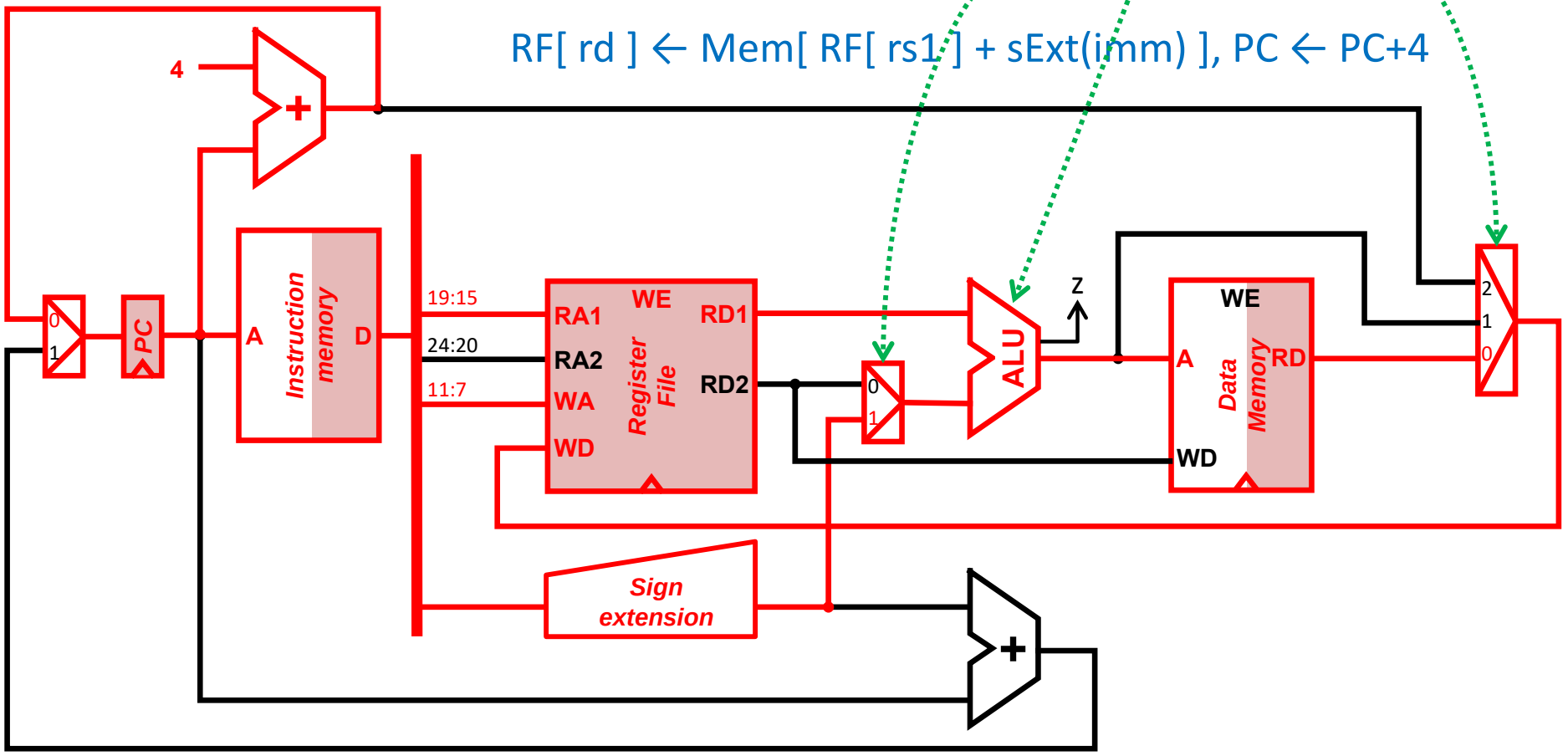
Controller design

Controller design: main DEC

```
lw rd, imm(rs1)
```

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0000011 (lw)	0	0	1	1	00 (add)	0	00

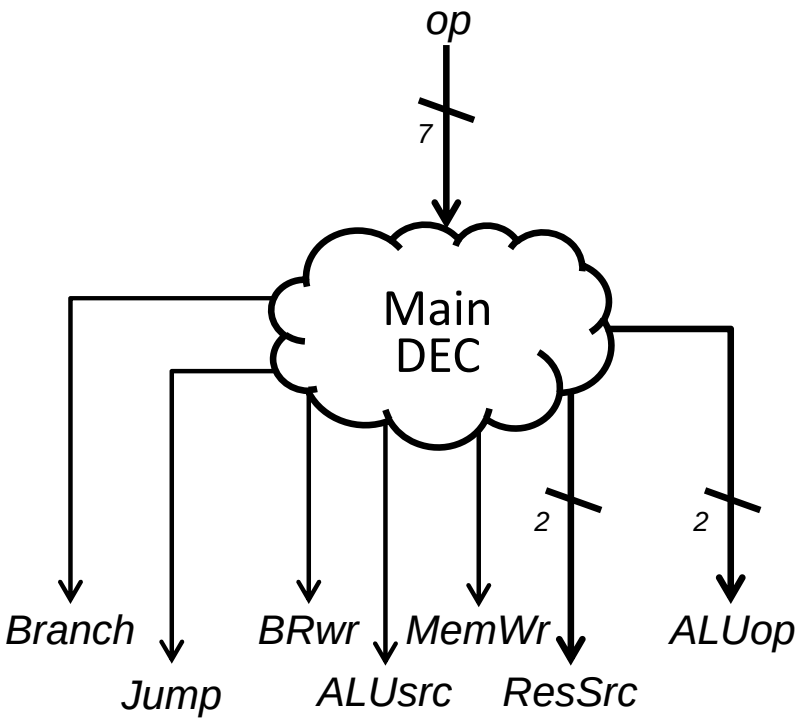
RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4



Controller design

Controller design: main DEC

- This subcircuit rules the general behavior of the processor.



Truth table

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0000011 (lw)	0	0	1	1	00(add)	0	00
0100011 (sw)							
0010011 (I-type)							
0110011 (R-type)							
1100011 (beq)							
1101111 (jal)							

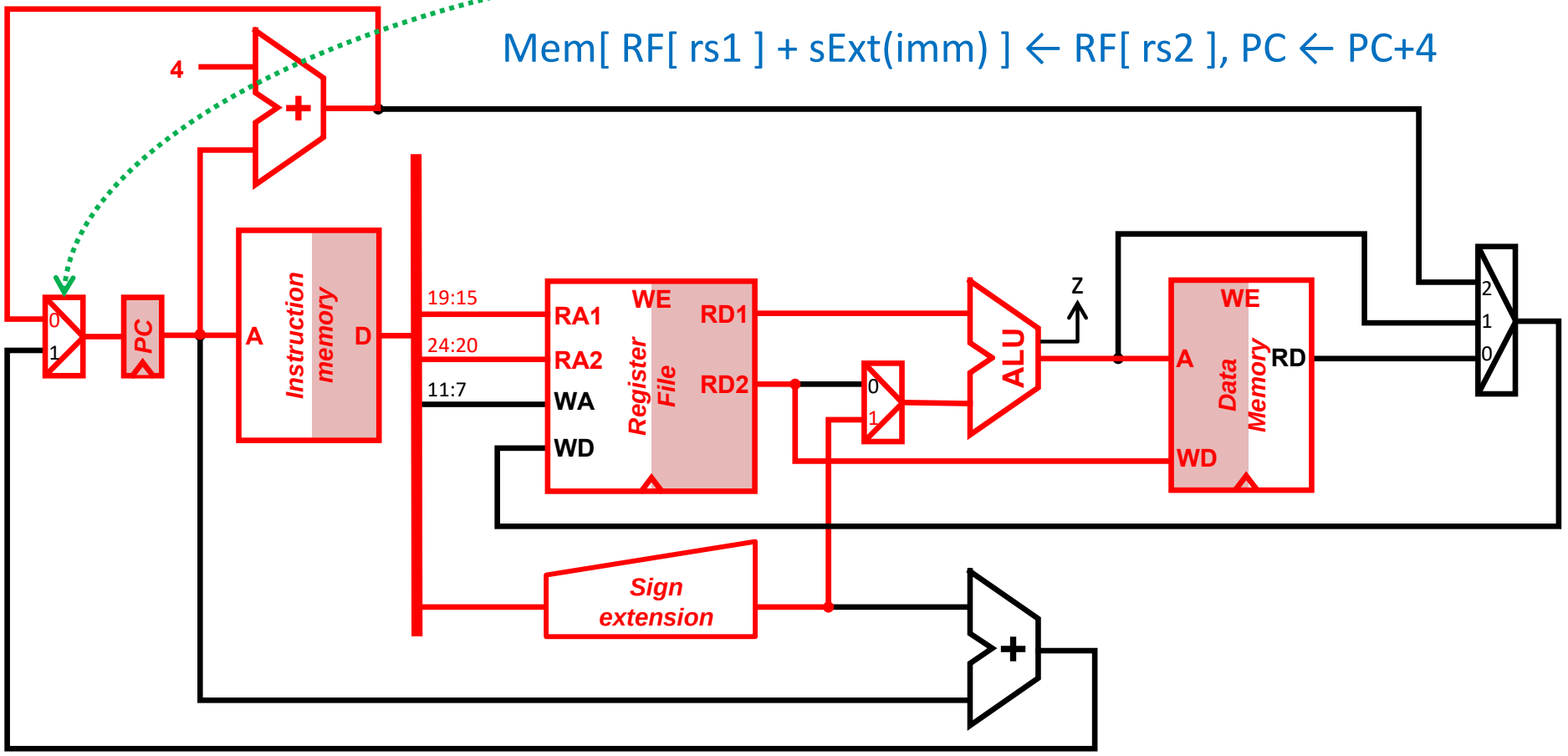
Controller design

Controller design: main DEC

```
sw rs2, imm(rs1)
```

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0100011 (sw)	0	0					

Mem[RF[rs1] + sExt(imm)] ← RF[rs2], PC ← PC+4



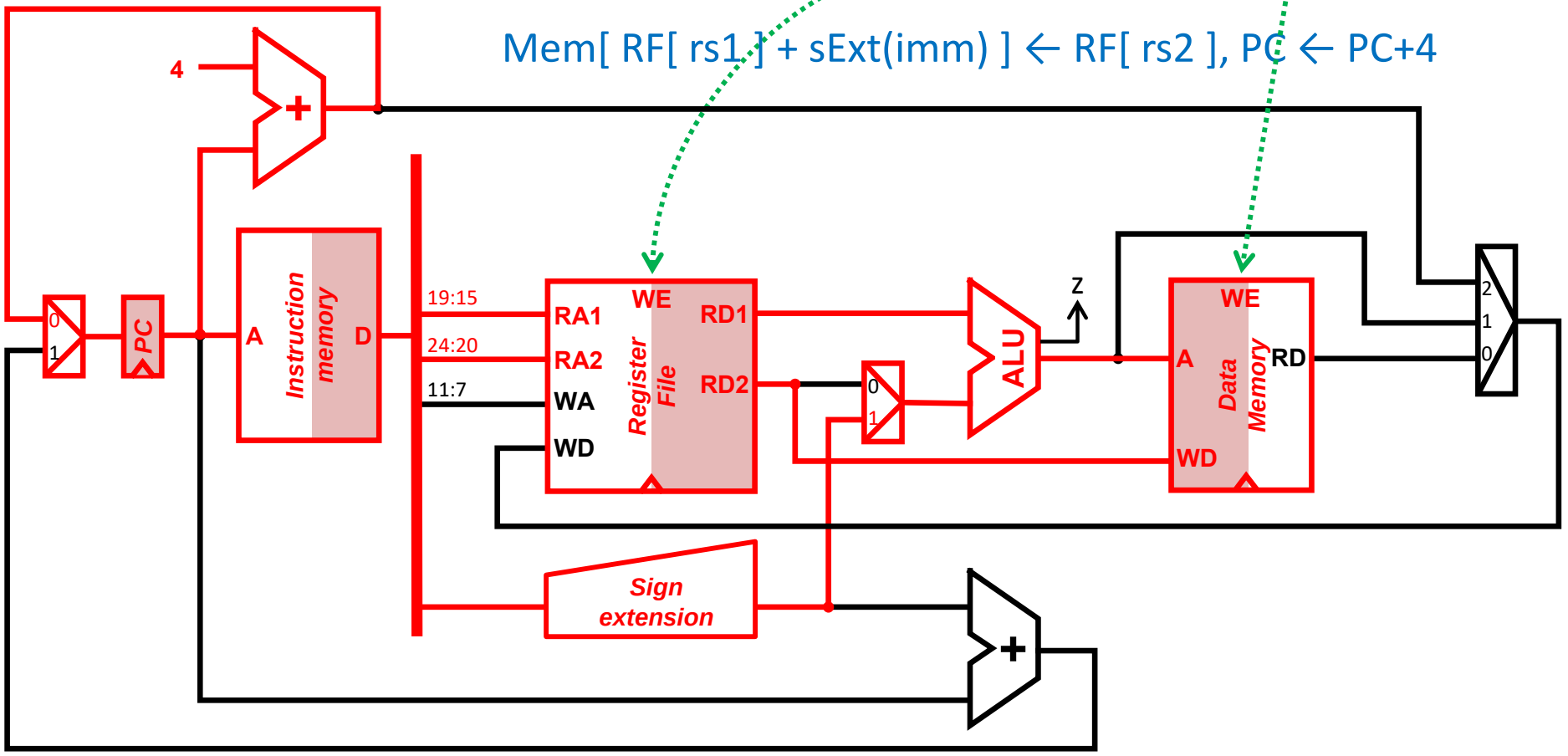
Controller design

Controller design: main DEC

```
sw rs2, imm(rs1)
```

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0100011 (sw)	0	0	0			1	

Mem[RF[rs1] + sExt(imm)] ← RF[rs2], PC ← PC+4



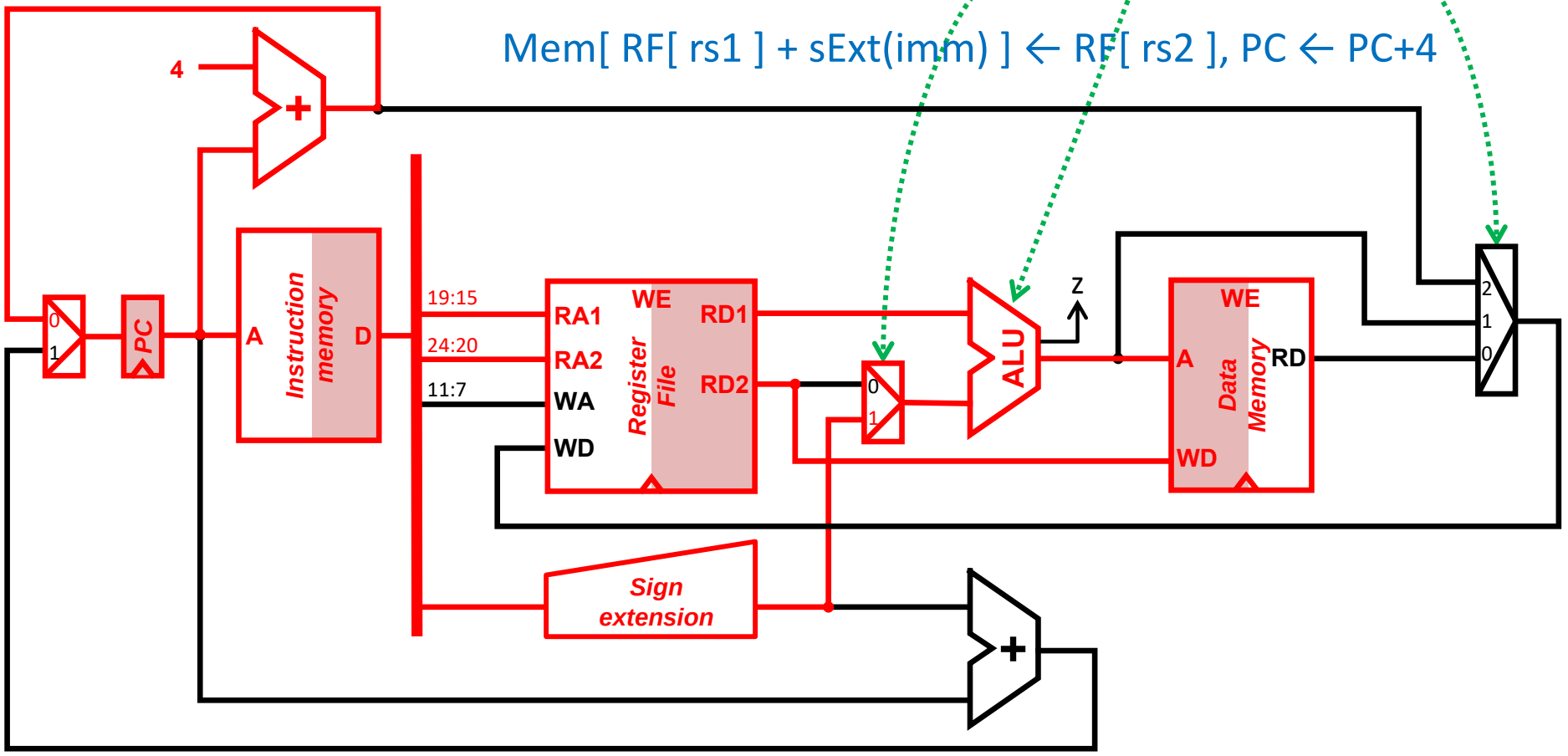
Controller design

Controller design: main DEC

```
sw rs2, imm(rs1)
```

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0100011 (sw)	0	0	0	1	00 (add)	1	–

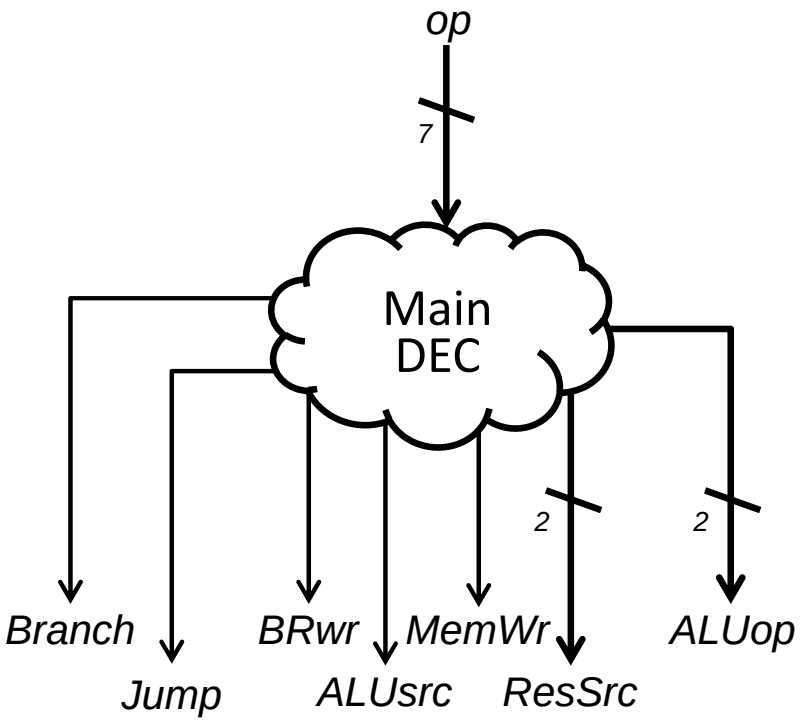
Mem[RF[rs1] + sExt(imm)] ← RF[rs2], PC ← PC+4



Controller design

Controller design: main DEC

- This subcircuit rules the general behavior of the processor.



Truth table

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0000011 (lw)	0	0	1	1	00(add)	0	00
0100011 (sw)	0	0	0	1	00(add)	1	–
0010011 (I-type)							
0110011 (R-type)							
1100011 (beq)							
1101111 (jal)							

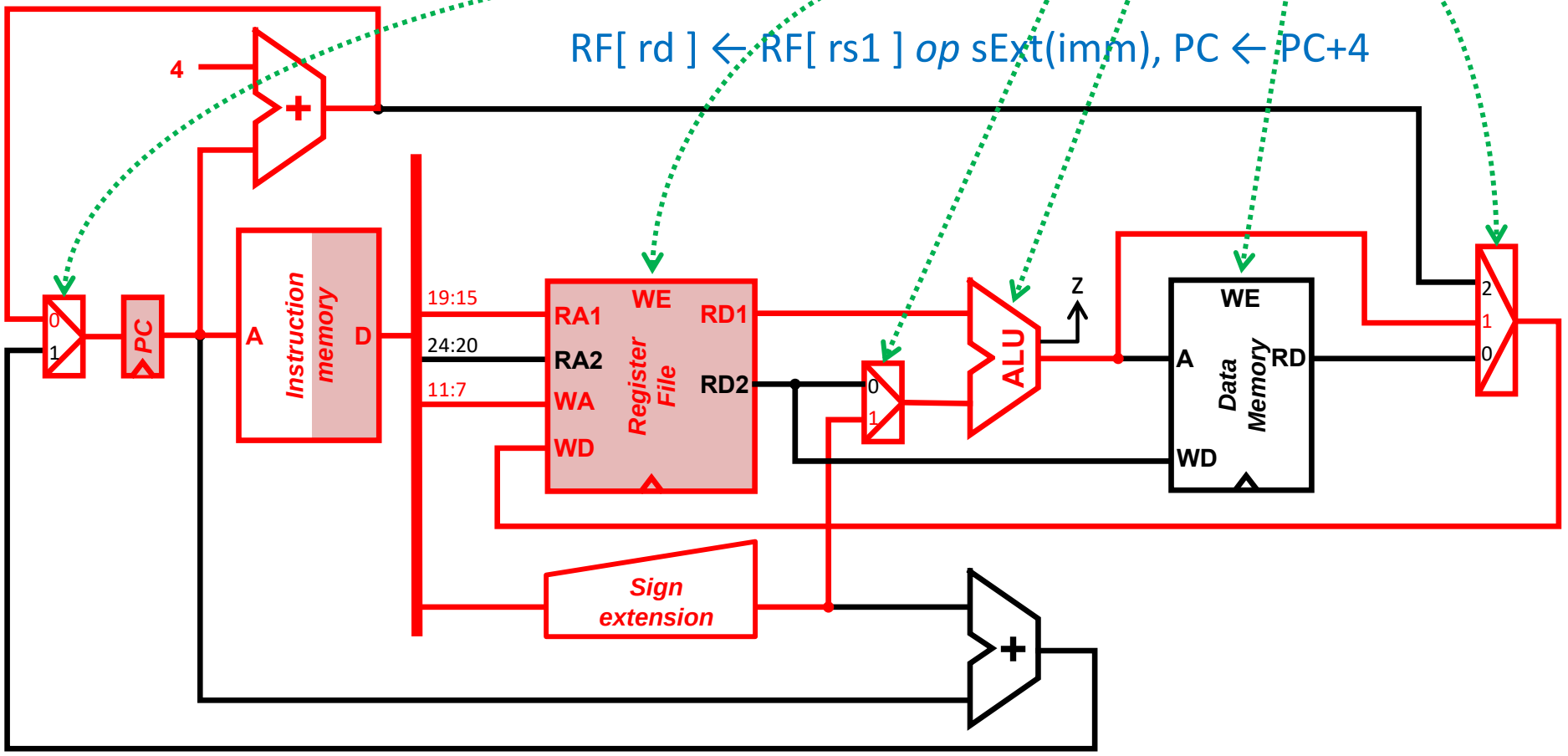
Controller design

Controller design: main DEC

```
addi rd, rs1, imm
```

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0010011 (I-type)	0	0	1	1	10 (operate)	0	01

RF[rd] ← RF[rs1] *op* sExt(imm), PC ← PC+4



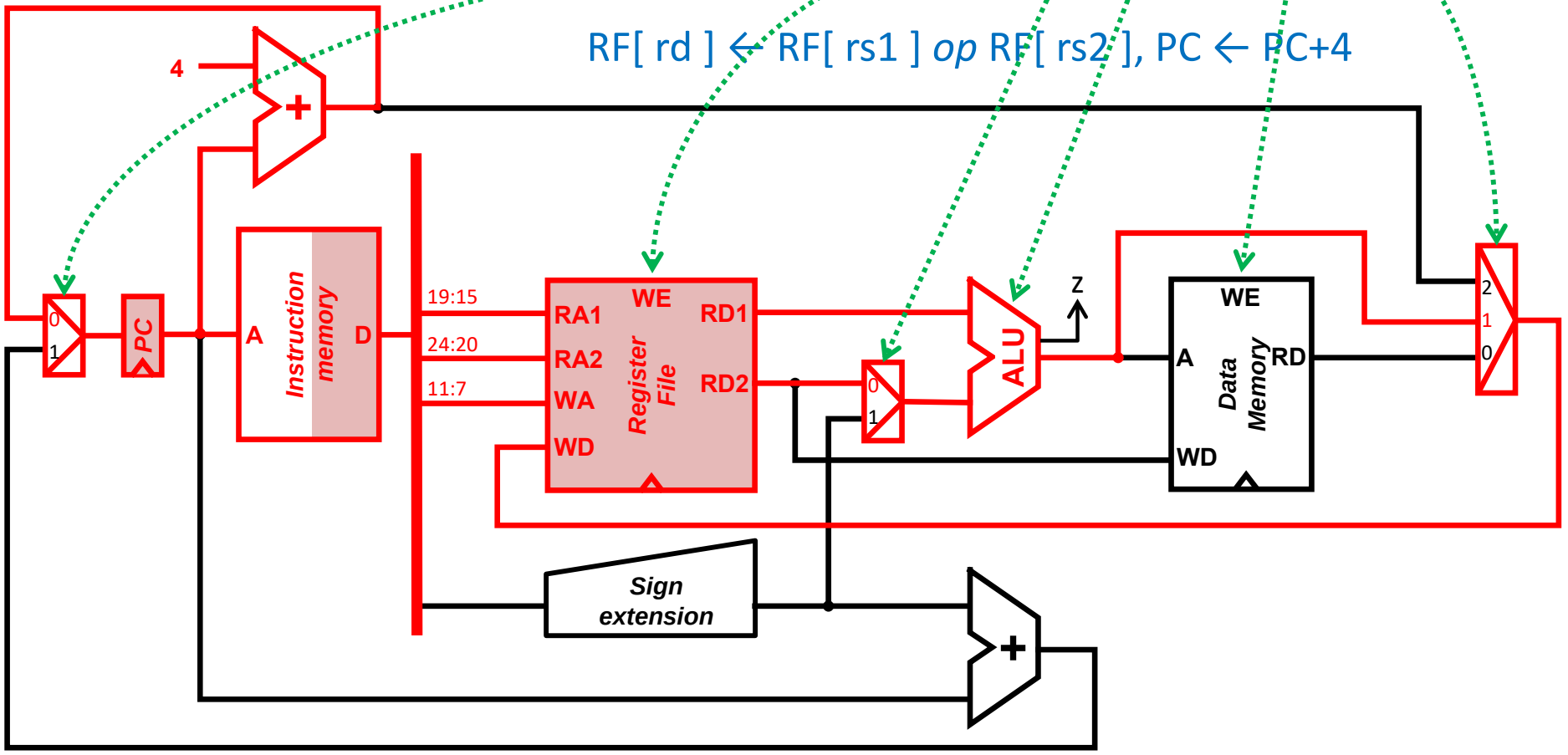
Controller design

Controller design: main DEC

```
add rd, rs1, rs2
```

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0110011 (R-type)	0	0	1	0	10 (operate)	0	01

RF[rd] ← RF[rs1] *op* RF[rs2], PC ← PC+4



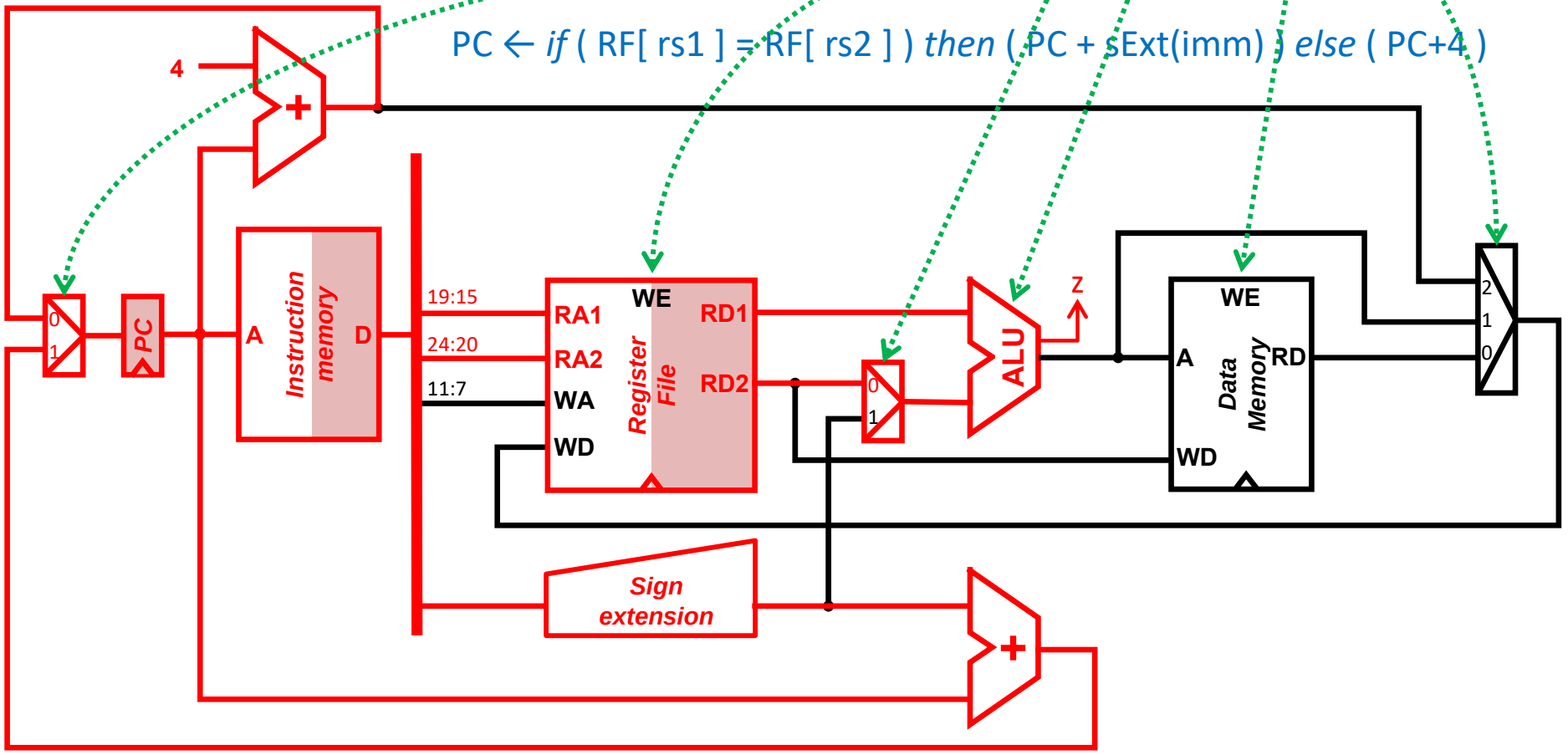
Controller design

Controller design: main DEC

```
beq rs1,rs2,imm
```

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
1100011 (beq)	1	0	0	0	01 (subtract)	0	–

PC ← *if* (RF[rs1] = RF[rs2]) *then* (PC + sExt(imm)) *else* (PC+4)



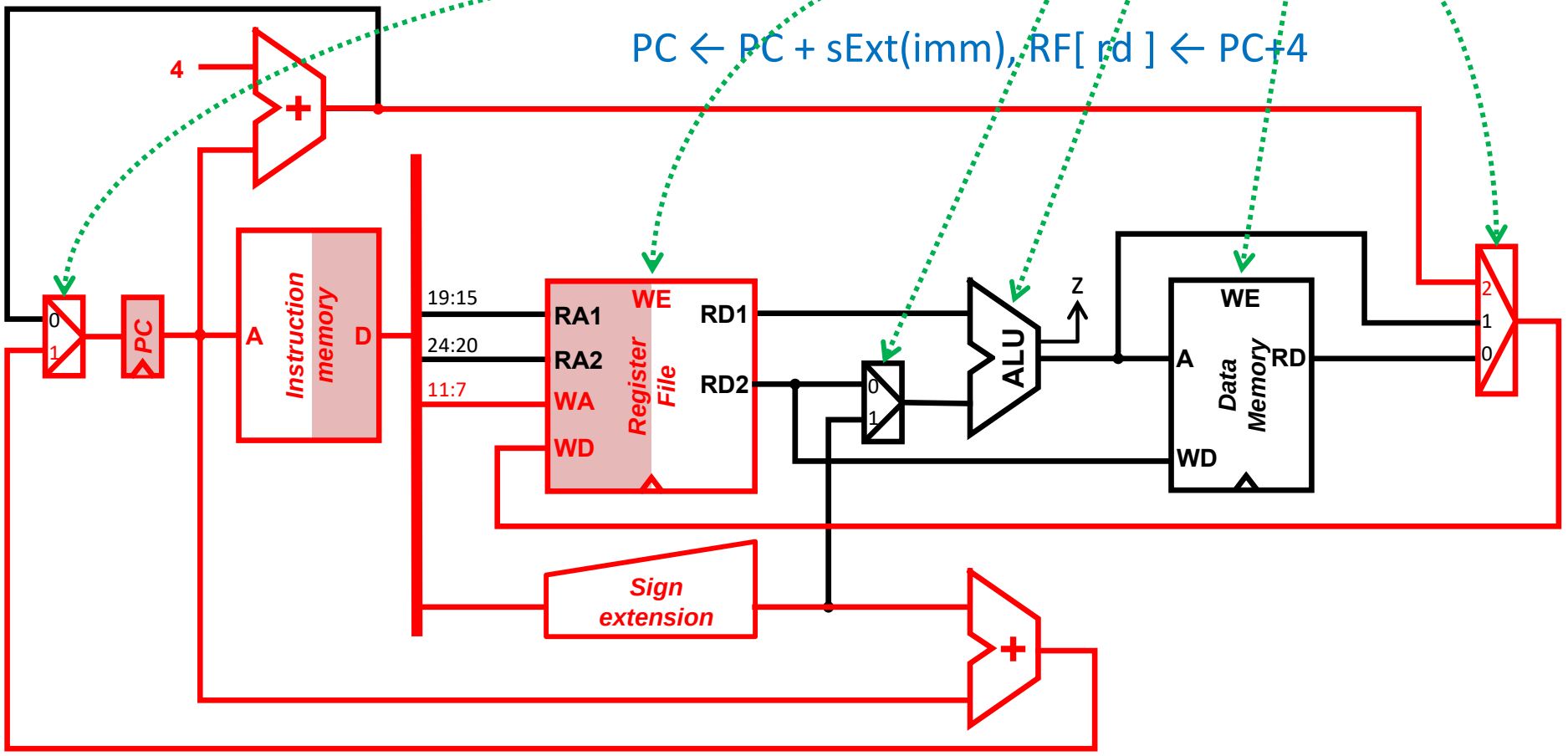
Controller design

Controller design: main DEC

```
jal rd, imm
```

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
1101111 (jal)	0	1	1	–	–	0	10

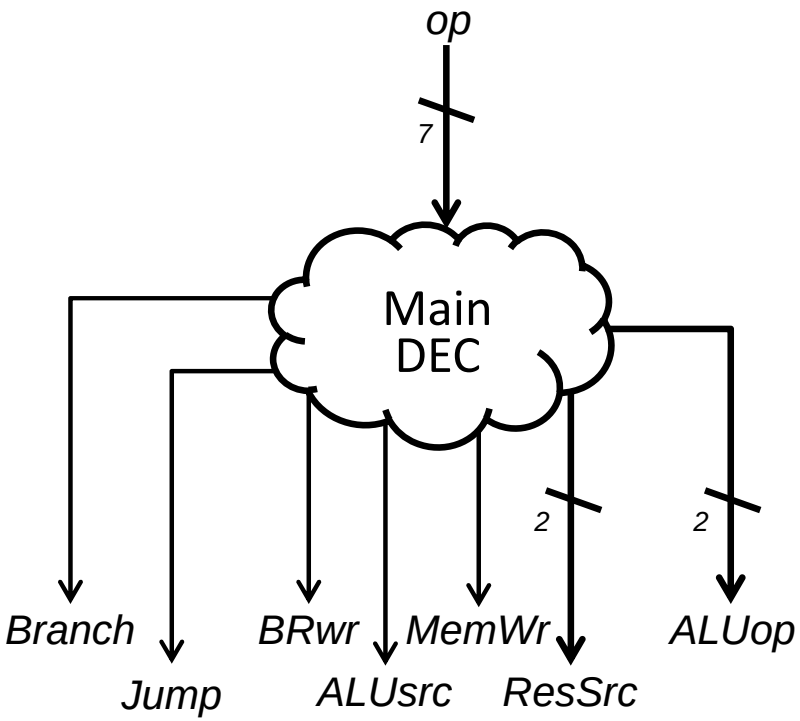
PC ← PC + sExt(imm), RF[rd] ← PC+4



Controller design

Controller design: main DEC

- This subcircuit rules the general behavior of the processor.



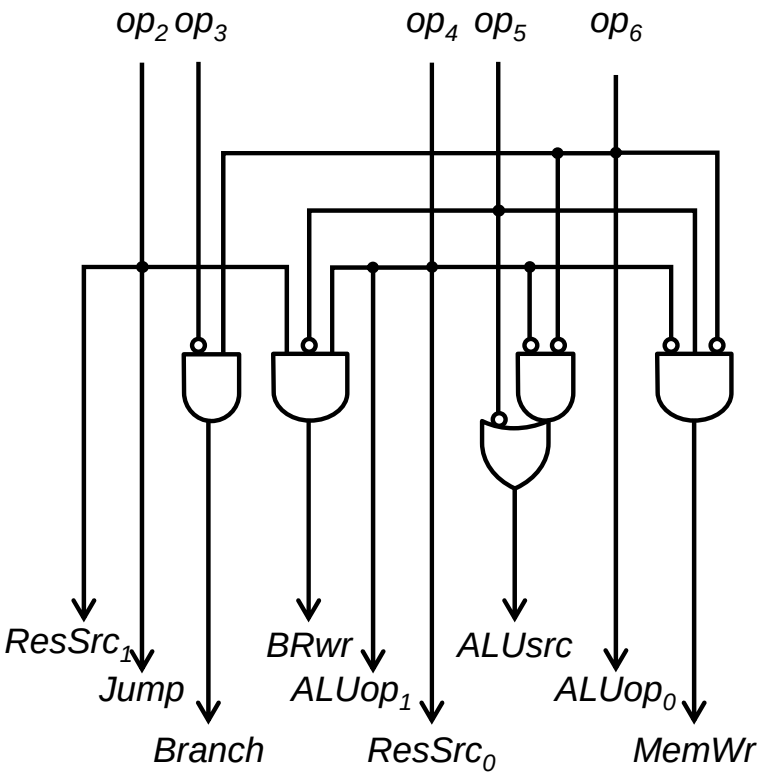
Truth table

op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0000011 (lw)	0	0	1	1	00 (add)	0	00
0100011 (sw)	0	0	0	1	00 (add)	1	–
0010011 (I-type)	0	0	1	1	10 (operate)	0	01
0110011 (R-type)	0	0	1	0	10 (operate)	0	01
1100011 (beq)	1	0	0	0	01 (subtract)	0	–
1101111 (jal)	0	1	1	–	–	0	10

Controller design

Controller design: main DEC

- This subcircuit rules the general behavior of the processor.

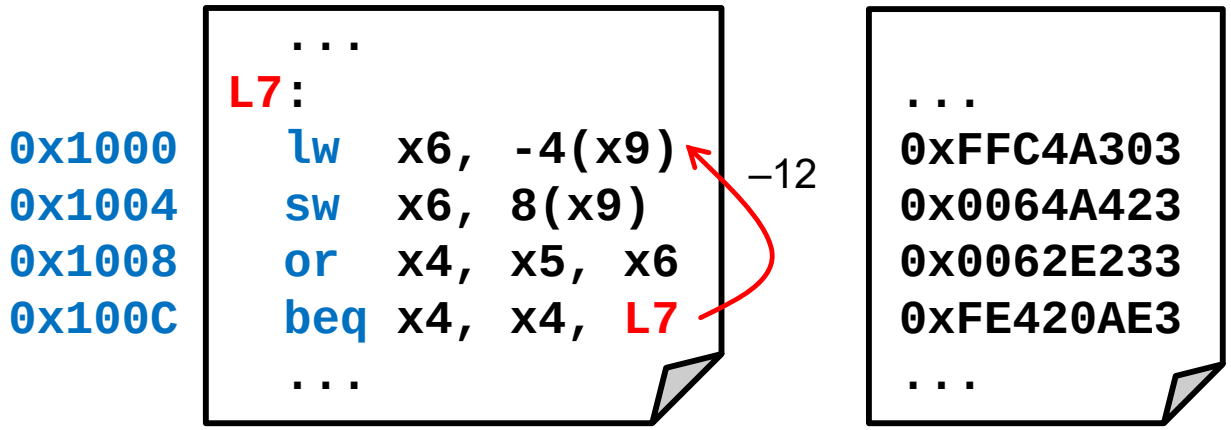


Truth table

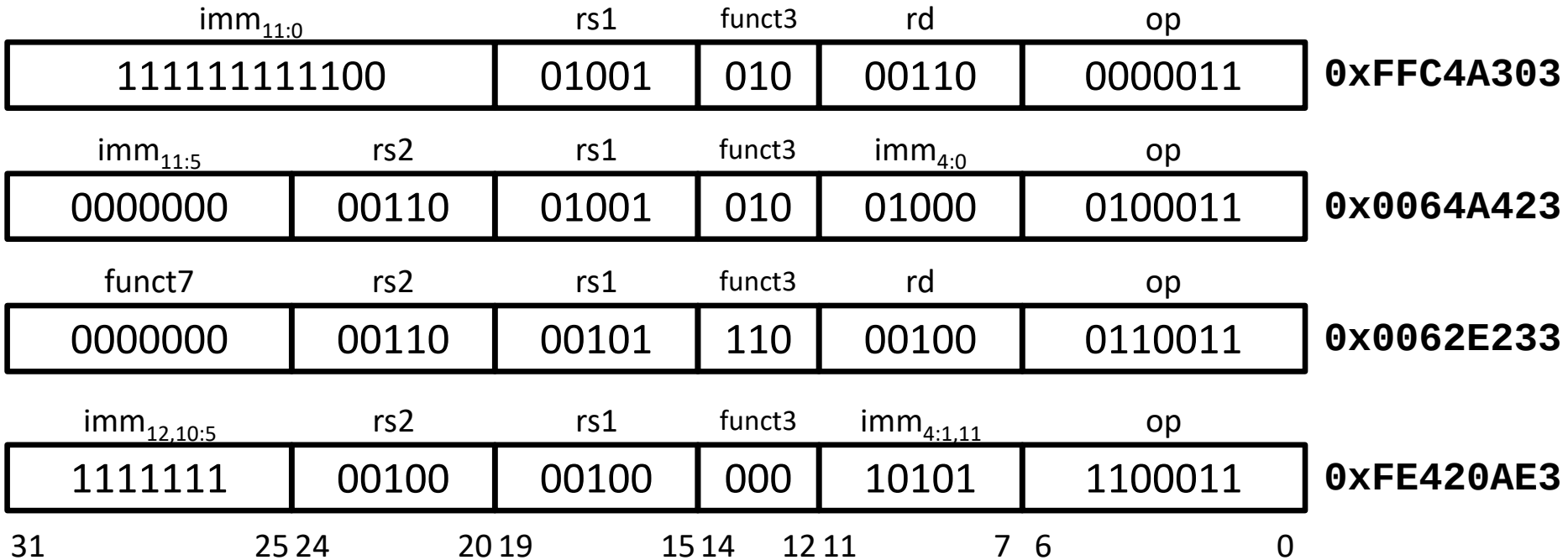
op	Branch	Jump	BRwr	ALUsrc	ALUop	MemWr	ResSrc
0000011 (lw)	0	0	1	1	00 (add)	0	00
0100011 (sw)	0	0	0	1	00 (add)	1	–
0010011 (I-type)	0	0	1	1	10 (operate)	0	01
0110011 (R-type)	0	0	1	0	10 (operate)	0	01
1100011 (beq)	1	0	0	0	01 (subtract)	0	–
1101111 (jal)	0	1	1	–	–	0	10

Single-cycle processor

Simulation (i)

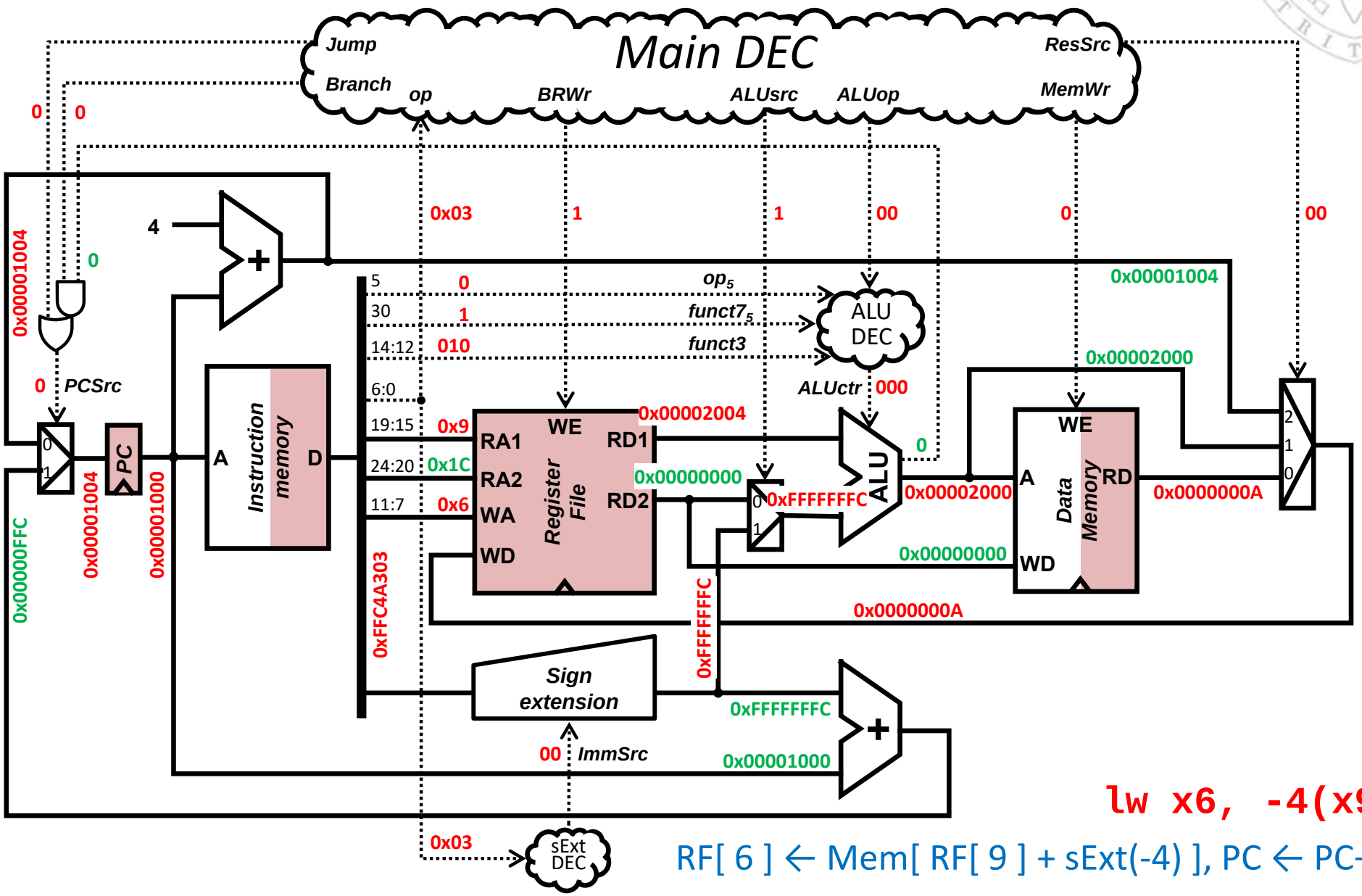


x5	6
x9	0x2004
Mem[0x2000]	10
PC	0x1000



Single-cycle processor

Simulation: 1st cycle

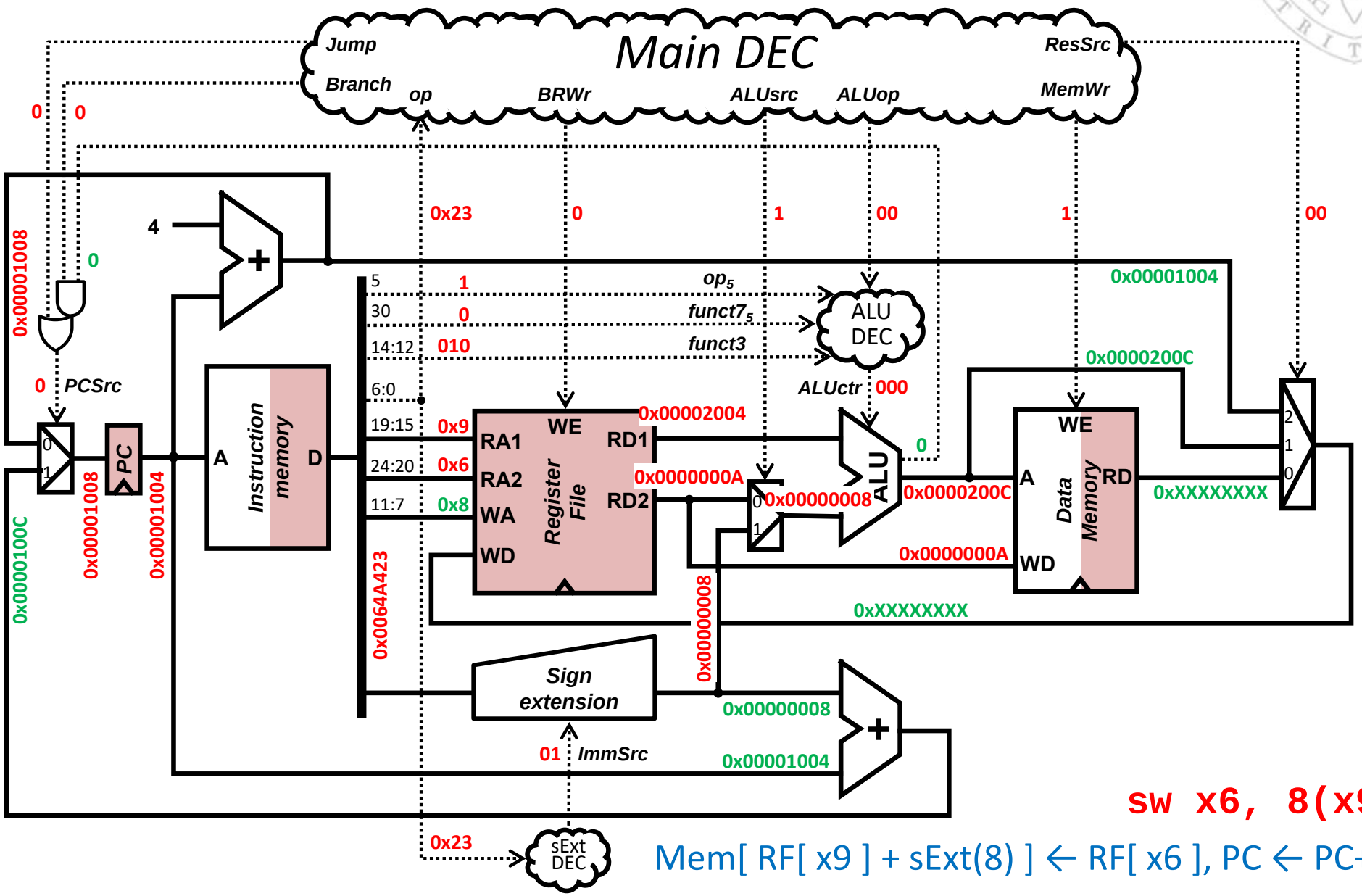


`lw x6, -4(x9)`

RF[6] ← Mem[RF[9] + sExt(-4)], PC ← PC+4

Single-cycle processor

Simulation: 2nd cycle

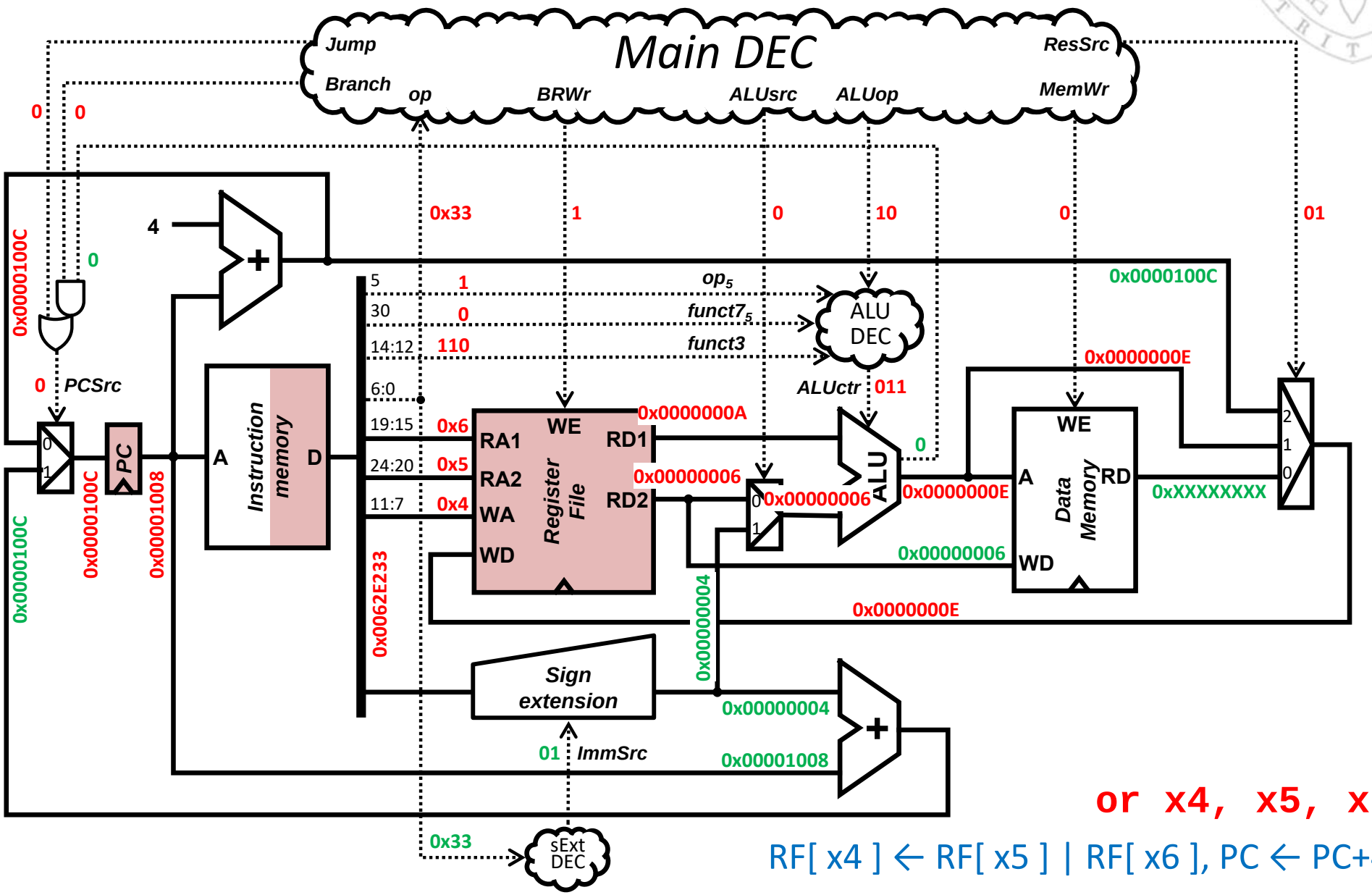


`sw x6, 8(x9)`

Mem[RF[x9] + sExt(8)] ← RF[x6], PC ← PC+4

Single-cycle processor

Simulation: 3rd cycle

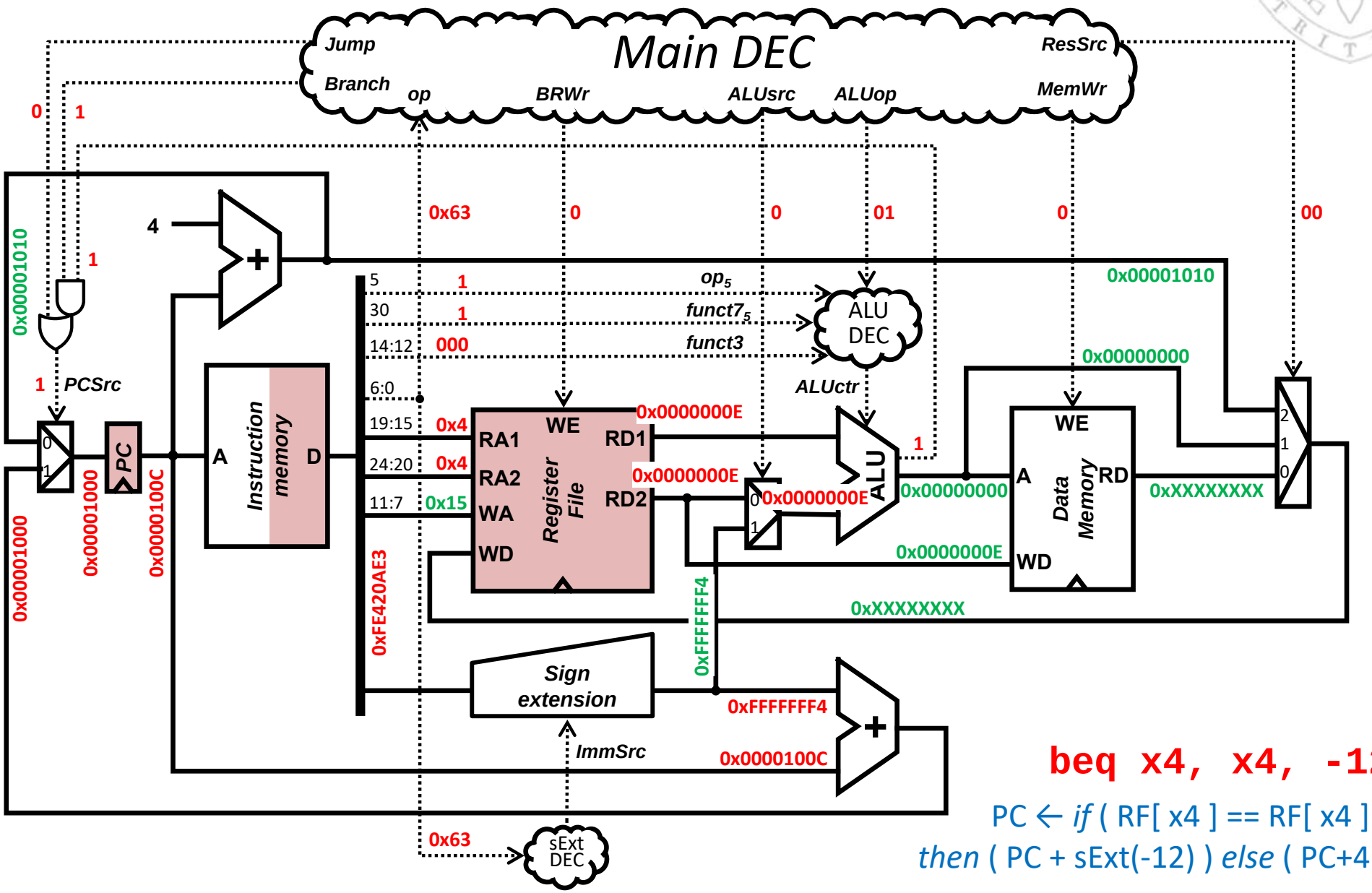


or x4, x5, x6

RF[x4] ← RF[x5] | RF[x6], PC ← PC+4

Single-cycle processor

Simulation: 4th cycle



beq x4, x4, -12

PC ← *if* (RF[x4] == RF[x4])
then (PC + sExt(-12)) *else* (PC+4)

Single-cycle processor

Cost and cycle time (90 nm CMOS)

$$area = 59181\ \mu m^2$$

$$t_{clk} = 27.6\ ns$$

$$f_{clk} = \frac{1}{t_{clk}} = \frac{1}{27.6 \cdot 10^{-9}\text{s}} = 36.2\ \text{MHz}$$

register transfer	instr.	critical path
PC ← PC+4	various	9,692 ps
RF[rd] ← Mem[RF[rs1] + sExt(imm)]	`lw`	27,616 ps
Mem[RF[rs1] + sExt(imm)] ← RF[rs2]	`sw`	26,661 ps
RF[rd] ← RF[rs1] *op* sExt(imm)	I-type	19,116 ps
RF[rd] ← RF[rs1] *op* RF[rs2]	R-type	18,928 ps
PC ← *if* (RF[rs1] = RF[rs2]) *then* (PC + sExt(imm)) *else* (PC+4)	`beq`	18,547 ps
RF[rd] ← PC+4	`jal`	10,073 ps
PC ← PC + sExt(imm)		17,033 ps
	max.	27,616 ps

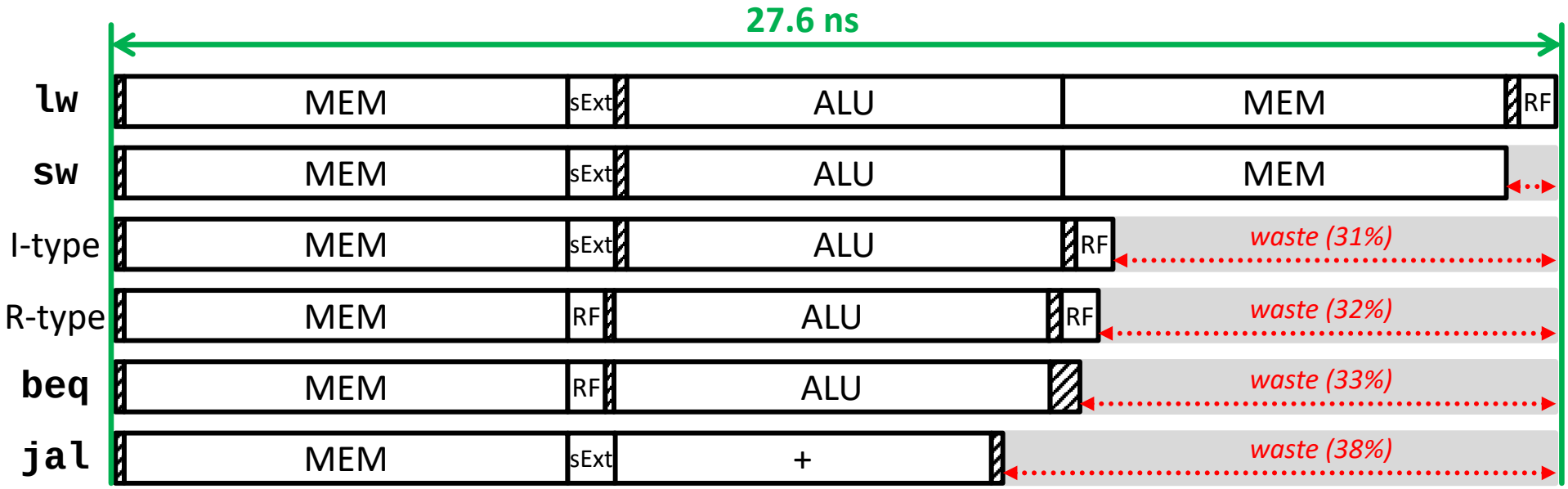
```
   ...
L7:
   lw   x6, -4(x9)   ····> 1 cycle
   sw   x6, 8(x9)    ····> 1 cycle
   or   x4, x5, x6   ····> 1 cycle
   beq  x4, x4, L7   ····> + 1 cycle
   ...                     4 cycles
```

$$t_{exec} = 4 \times 27.6\ ns = 110.4\ ns$$

Single-cycle processor

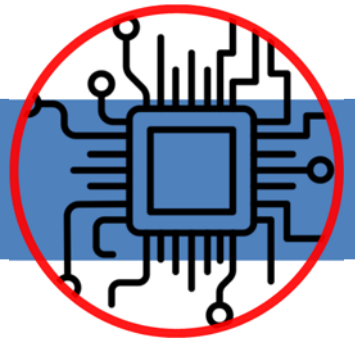
Conclusions

- The single-cycle implementation has some problems:
  - The cycle time is determined by the slowest instruction
    - All the instructions take the same time to execute regardless of its complexity: time is wasted in the execution of the faster instructions.
    - In real ISAs, there are some very long instructions: slow memories, complex arithmetic operations, complex addressing modes...
  - It is not possible to reuse hardware:
    - It requires one ALU and 2 adders, separate instruction and data memories...



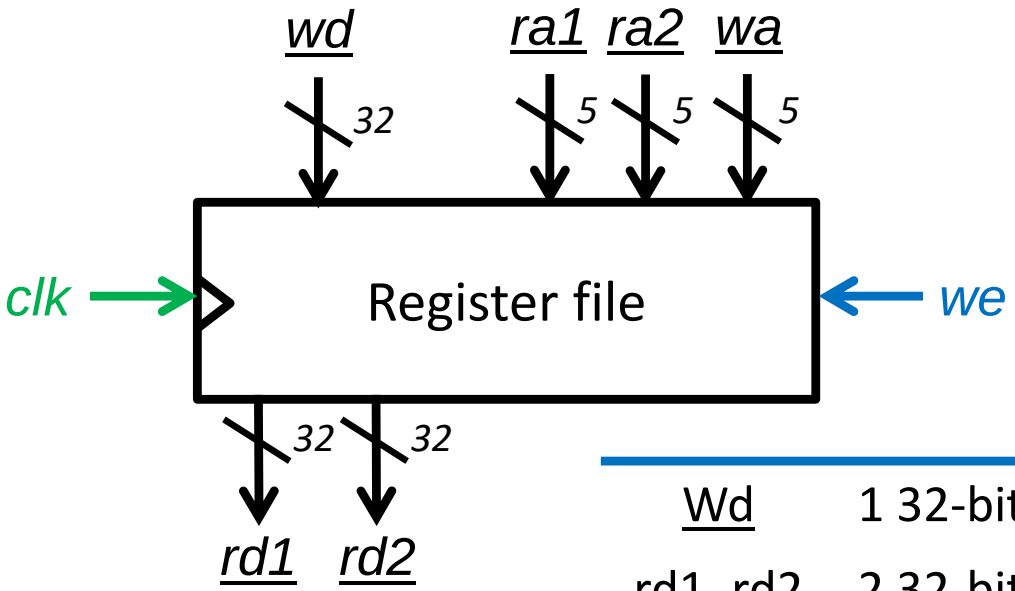
- Register File design.
- Memory design.
- ALU design.
- Sign Extension module design.
- Cost calculation.
- Cycle time calculation.

Technology



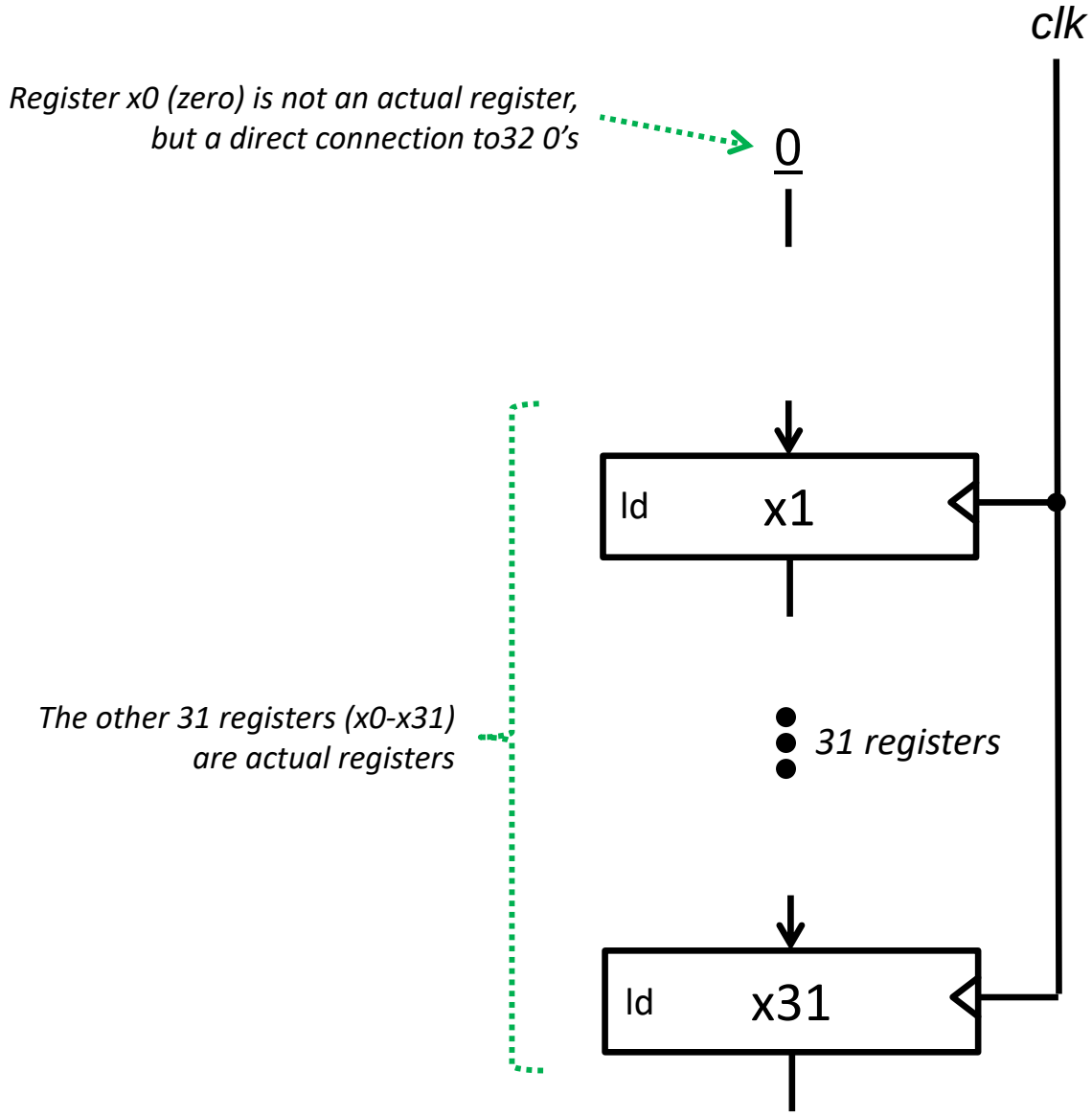
Register File design

- The Register File contains 32 data registers, each one with 32 bits.

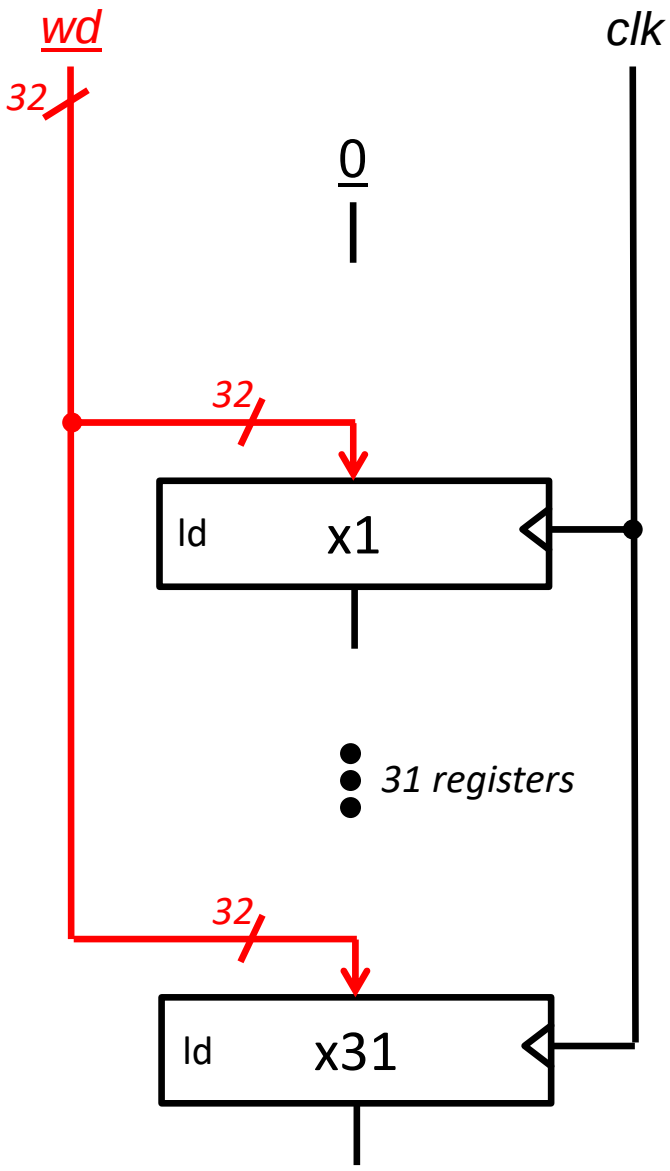


Wd	1 32-bit data input
rd1, rd2	2 32-bit data outputs
Wa	1 5-bit write address input
ra1, ra2	2 5-bit read address inputs
We	1 write enable input
Clk	1 clock input

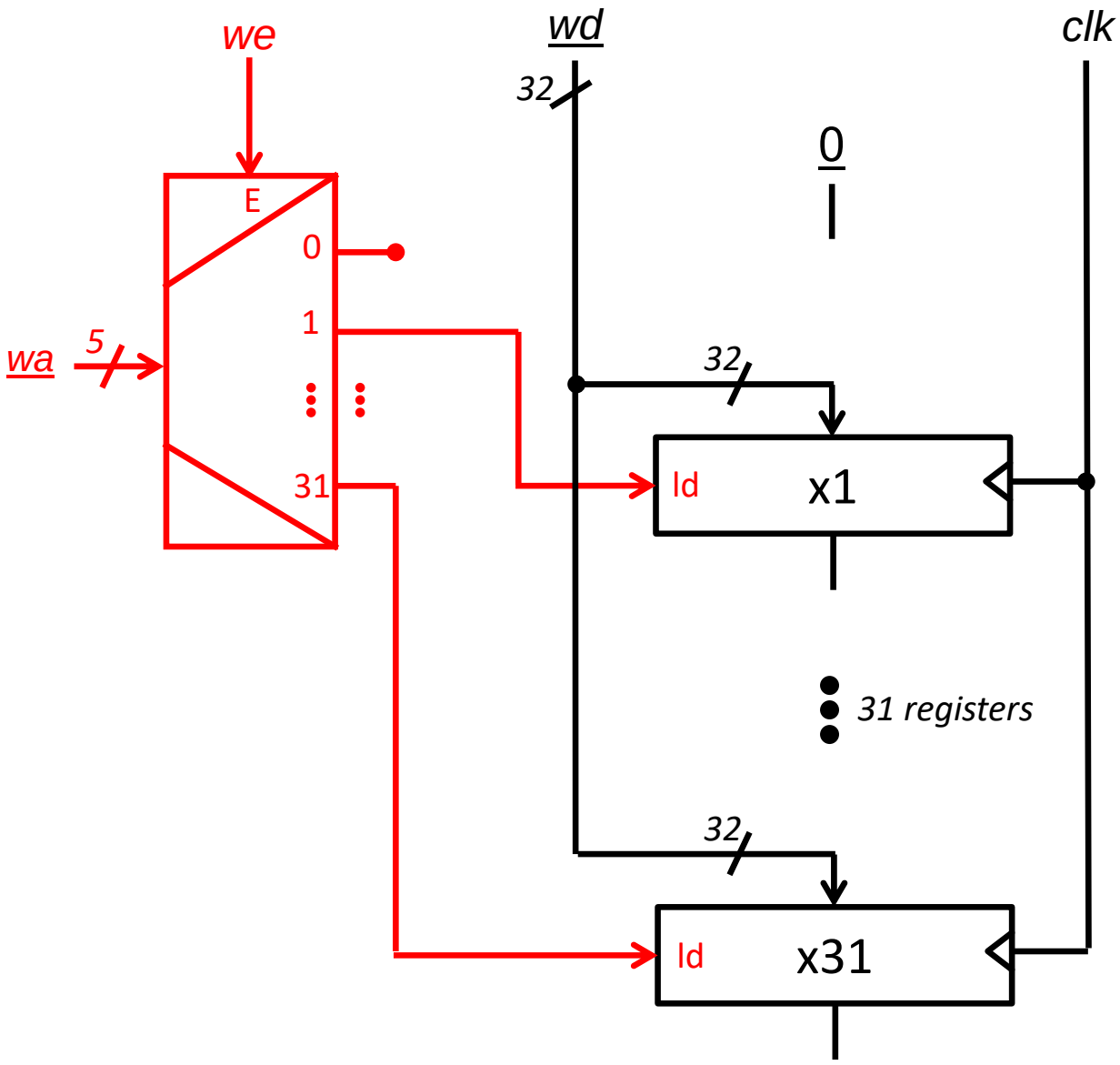
Register File design



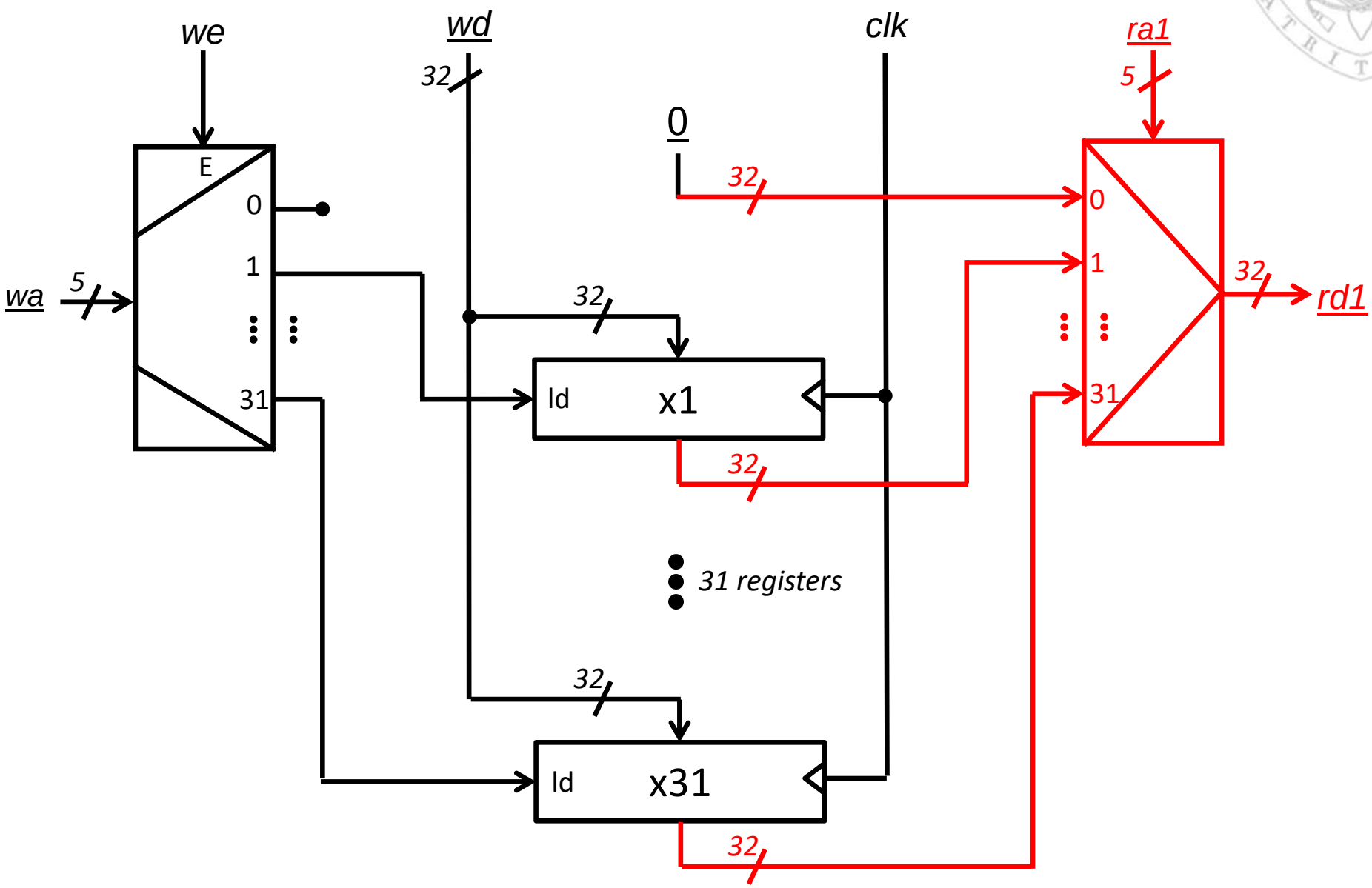
Register File design



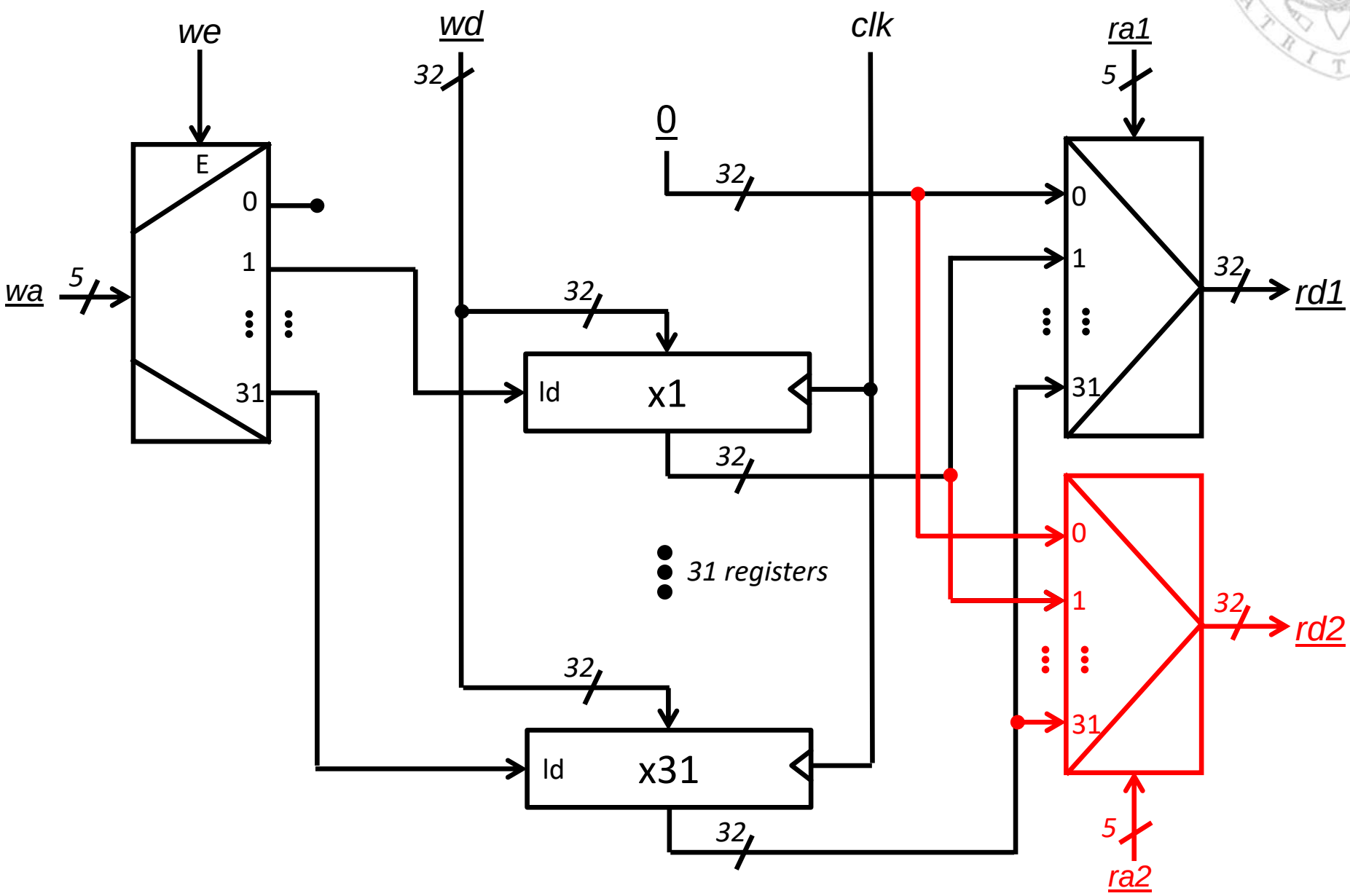
Register File design



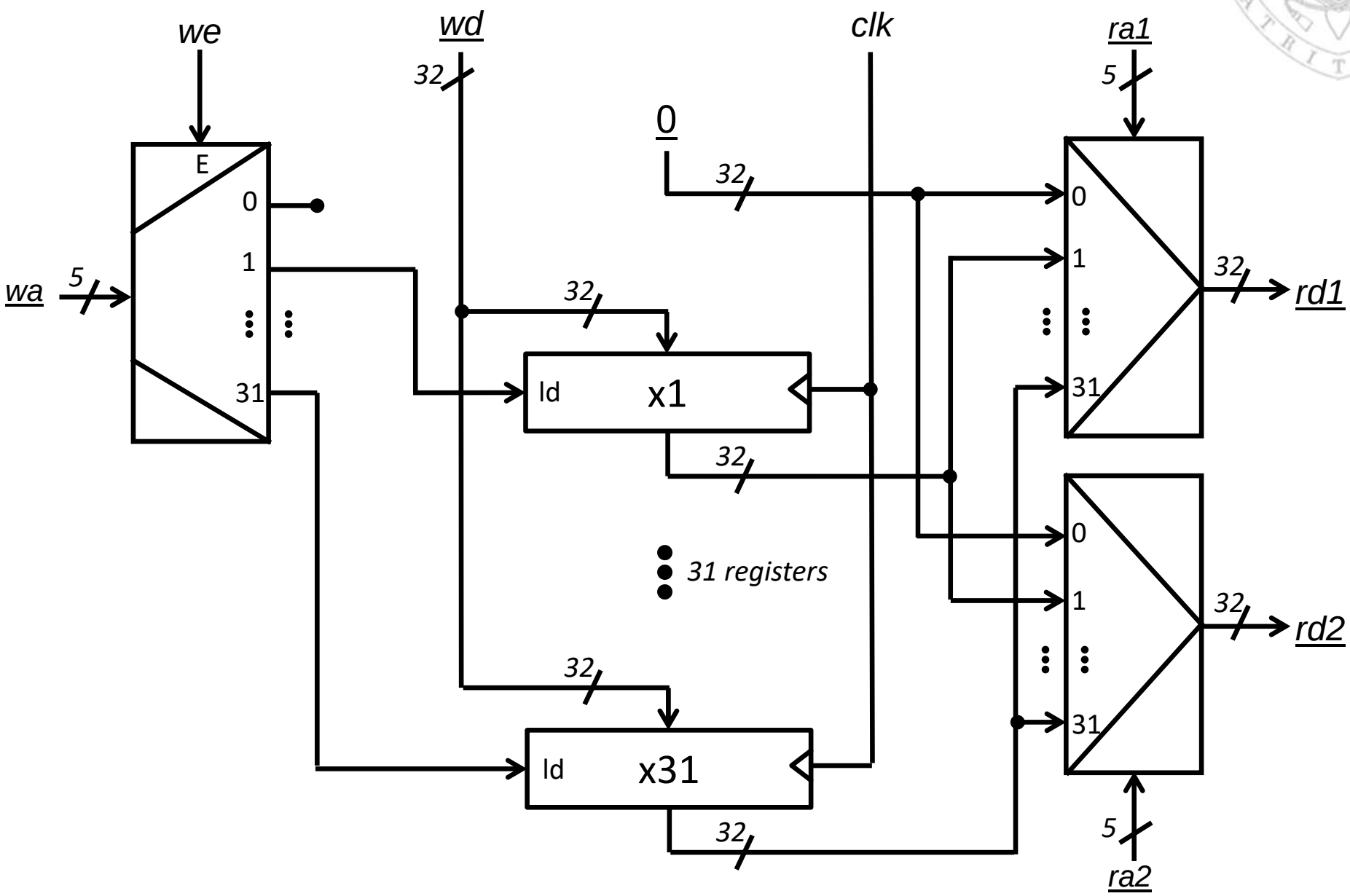
Register File design



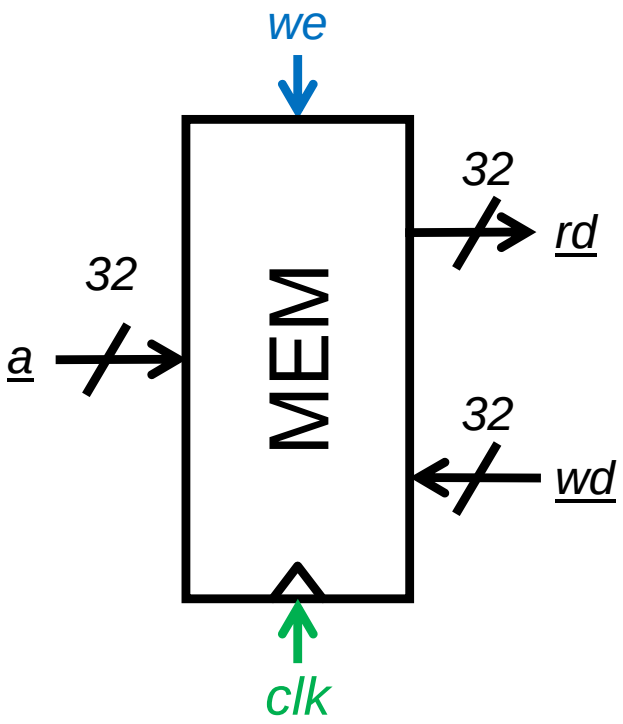
Register File design



Register File design



Memory design

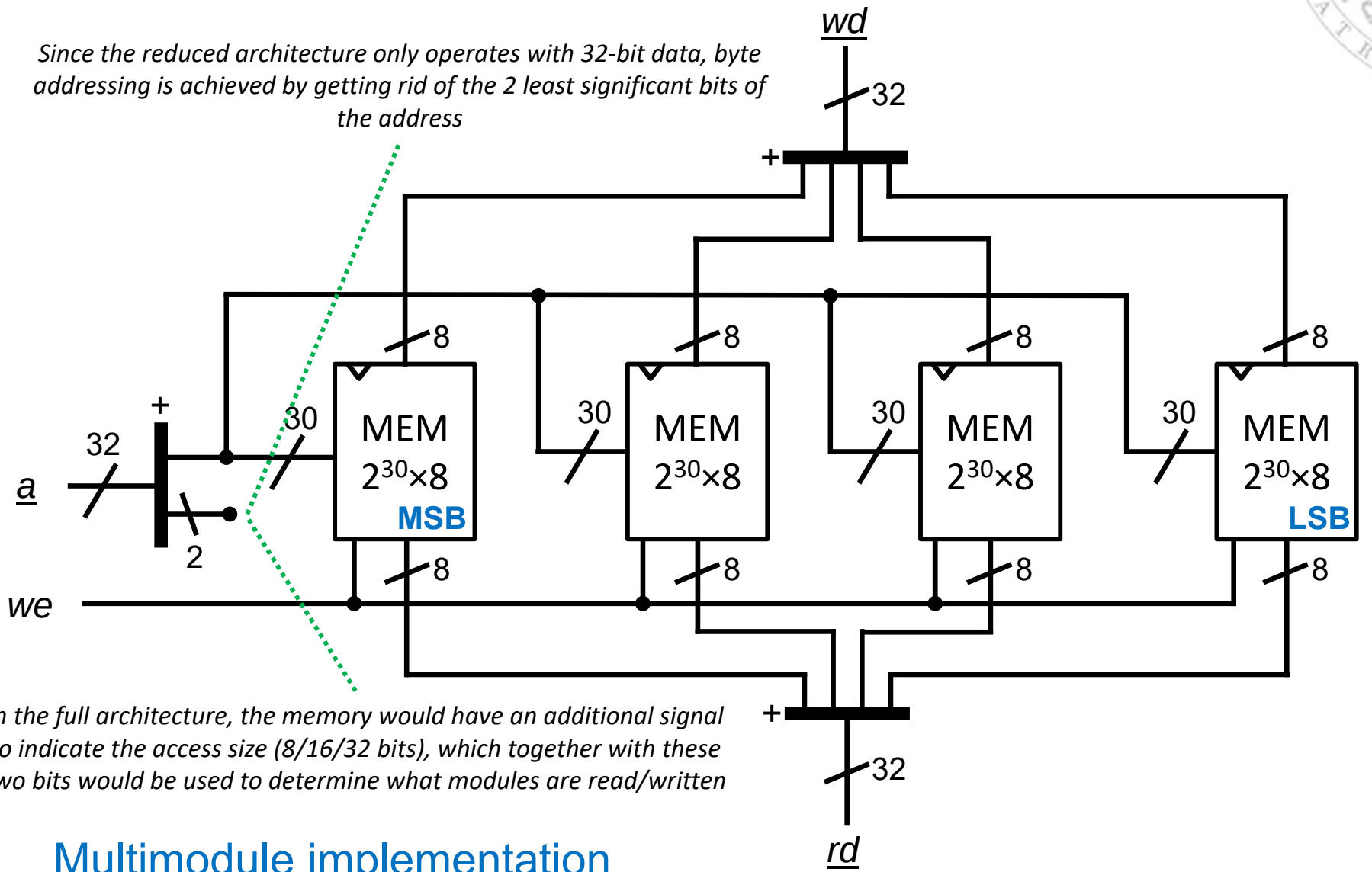


2^{32}×8 RAM = 2^{30}×32
(2^{30} words with 32 bits)

wd	1 32-bit data input
rd	1 32-bit data output
a	1 32-bit address input
we	1 write enable input
clk	1 clock input

Memory with double data port, byte addressable and with little-endian organization

Memory design



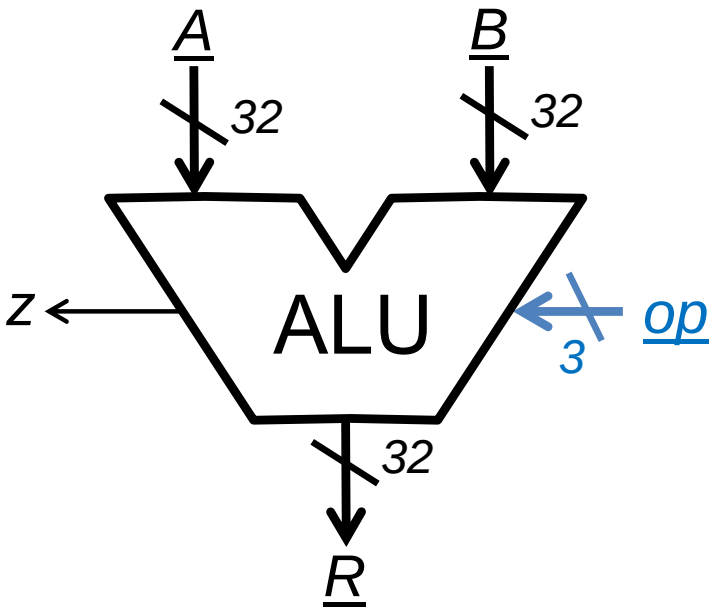
Since the reduced architecture only operates with 32-bit data, byte addressing is achieved by getting rid of the 2 least significant bits of the address

In the full architecture, the memory would have an additional signal to indicate the access size (8/16/32 bits), which together with these two bits would be used to determine what modules are read/written

Multimodule implementation

2^{30}×32 MEM (4 GiB) byte addressable using 4 2^{30}×8 MEM (1 GiB)

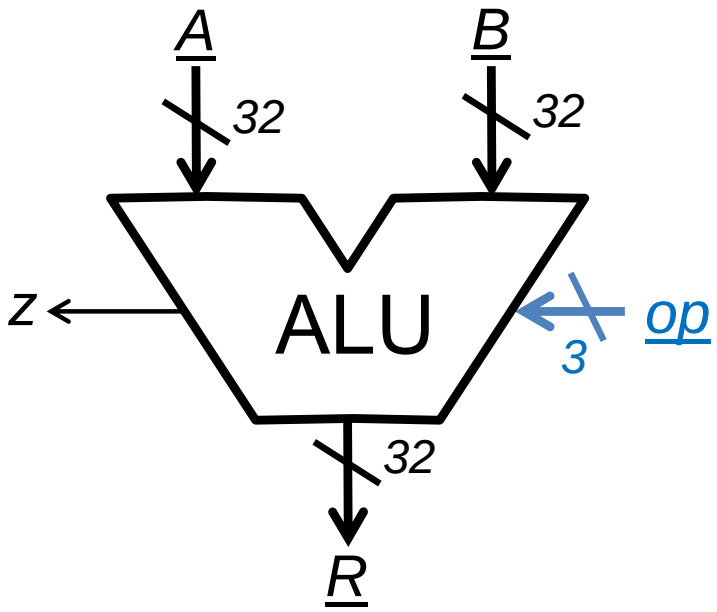
ALU design



A, B	2 32-bit data inputs
op	1 operation selector input
R	1 32-bit data output
z	1 zero flag

- The ALU is a combination module that performs:
  - The memory effective address calculation in **`lw`**/**`sw`** instructions.
  - All arithmetic-logic operations in I-type / R-type instructions.
  - The operand comparison in **`beq`** instructions.
  - In the multicycle data path, it will also be used to increment the PC and perform the branch address calculation in **`beq`**/**`jal`** instructions.

ALU design



A, B	2 32-bit data inputs
op	1 operation selector input
R	1 32-bit data output
z	1 zero flag

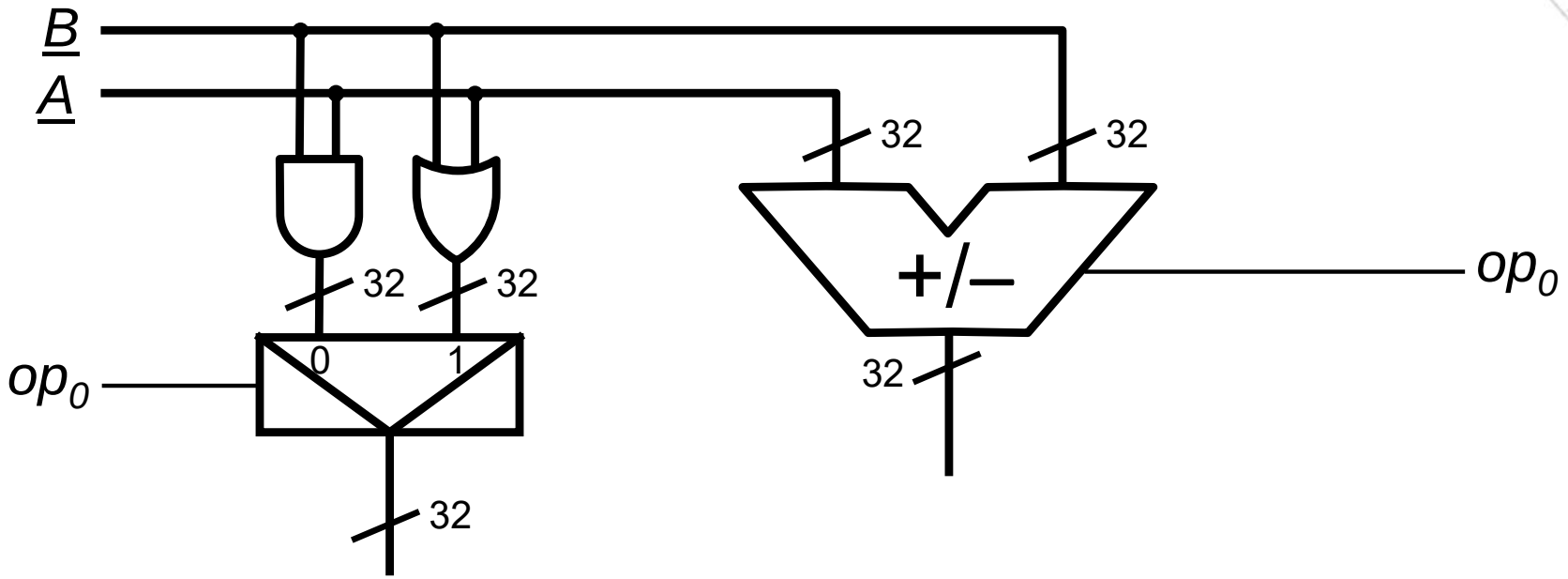
arithmetic operations

op_2	op_1	op_0	R
0	0	0	A + B
0	0	1	A – B
1	0	0	–
1	0	1	*if* (A<B) *then* 1 *else* 0

logical operations

op_2	op_1	op_0	R
0	1	0	A & B
0	1	1	A | B
1	1	0	–
1	1	1	–

ALU design



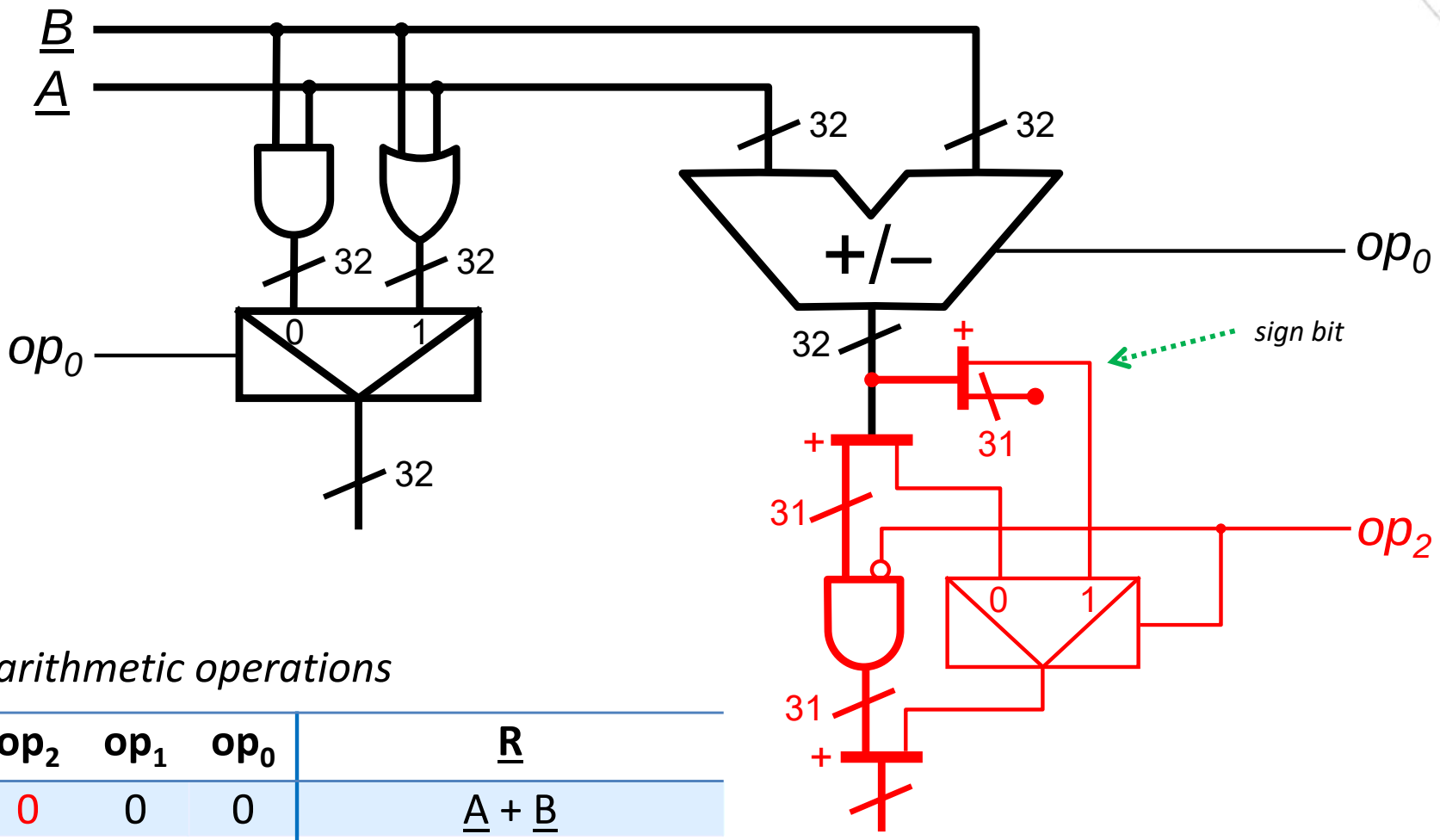
arithmetic operations

op_2	op_1	op_0	R
0	0	0	A + B
0	0	1	A – B
1	0	0	–
1	0	1	*if* (A<B) *then* 1 *else* 0

logical operations

op_2	op_1	op_0	R
0	1	0	A & B
0	1	1	A | B
1	1	0	–
1	1	1	–

ALU design



arithmetic operations

op_2	op_1	op_0	R
0	0	0	A + B
0	0	1	A – B
1	0	0	–
1	0	1	*if* (A<B) *then* 1 *else* 0

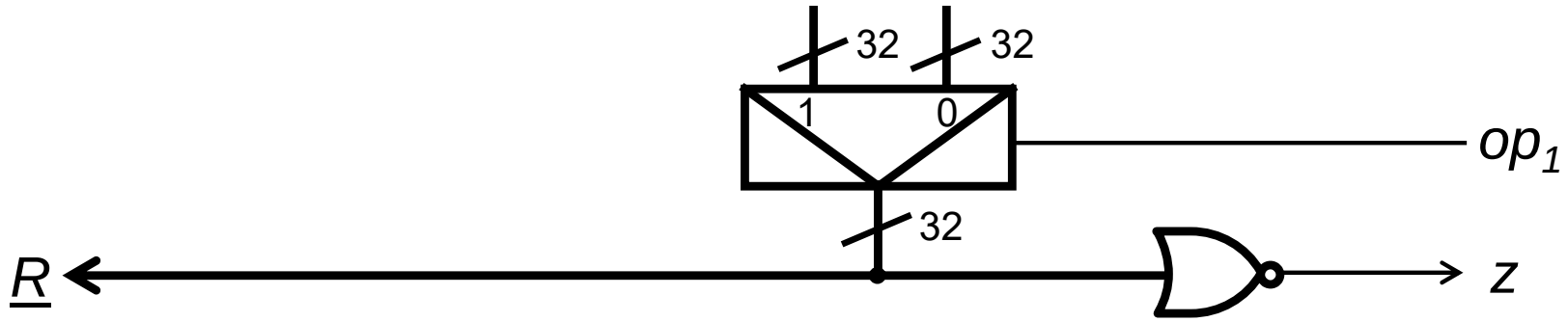
ALU design

arithmetic operations

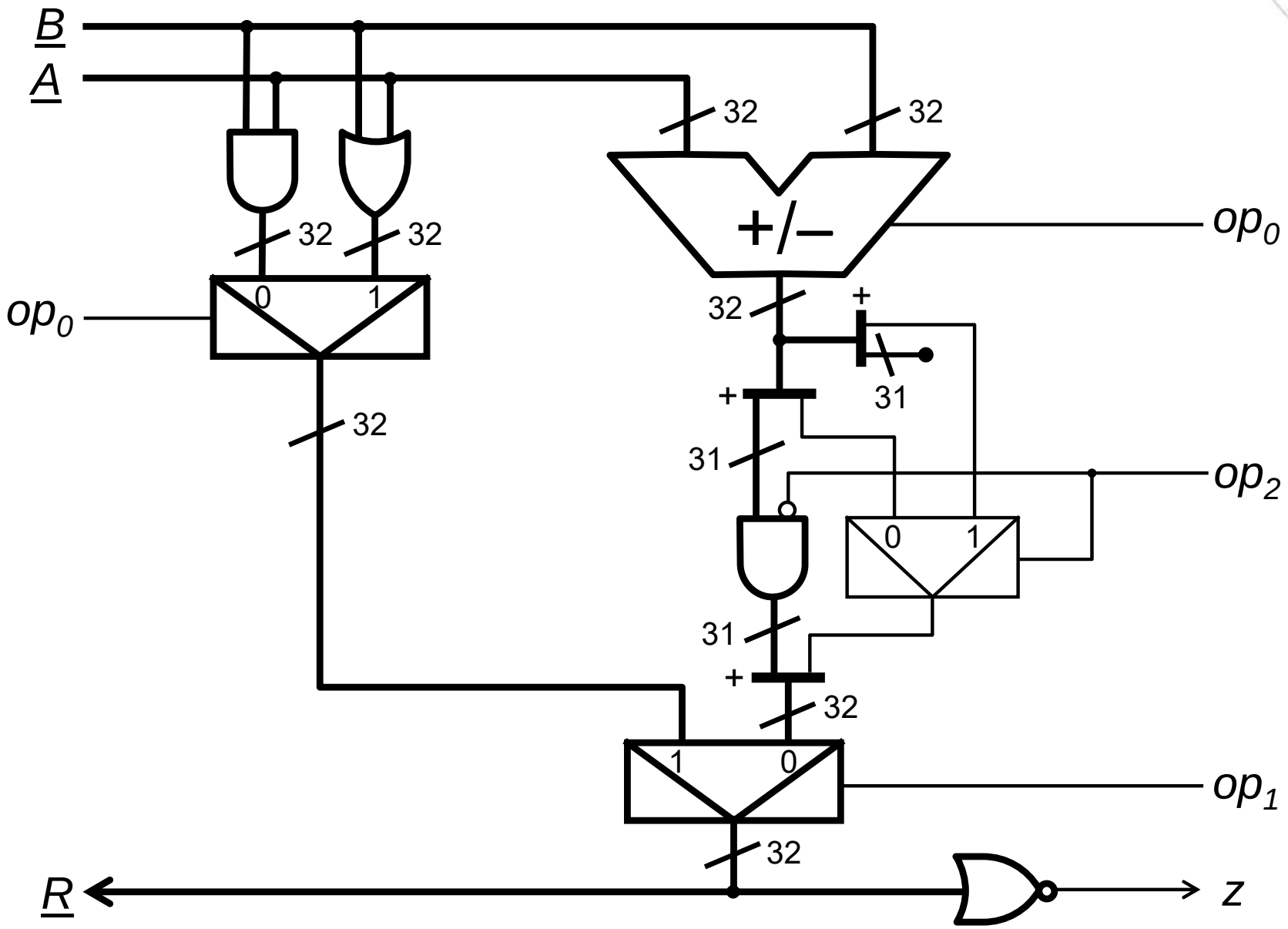
op_2	op_1	op_0	R
0	0	0	A + B
0	0	1	A – B
1	0	0	–
1	0	1	*if* (A<B) *then* 1 *else* 0

logical operations

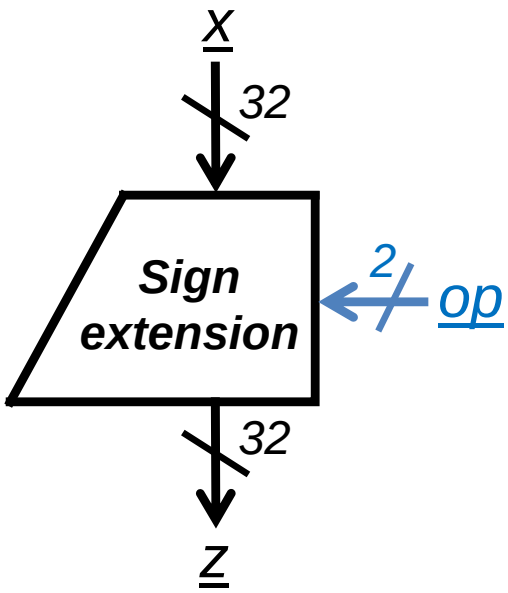
op_2	op_1	op_0	R
0	1	0	A & B
0	1	1	A | B
1	1	0	–
1	1	1	–



ALU design



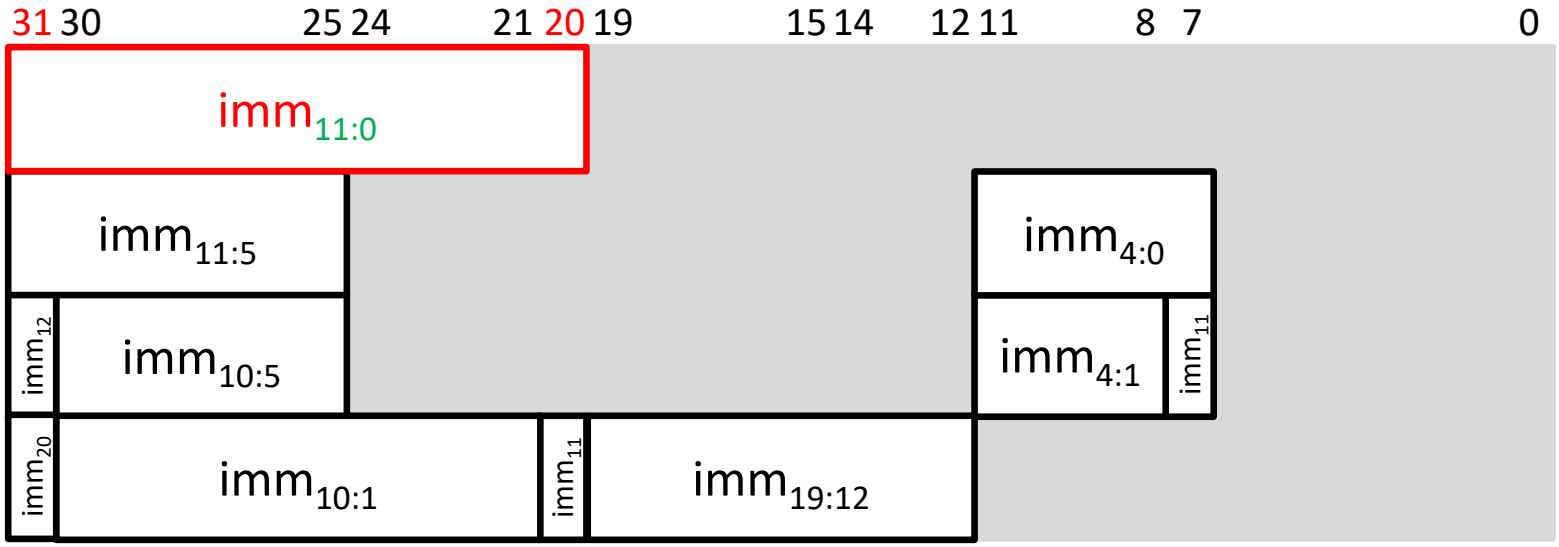
Sign Extension module design



x	1 32-bit data input (instruction)
op	1 operation selector input
z	1 32-bit data output (immediate operand)

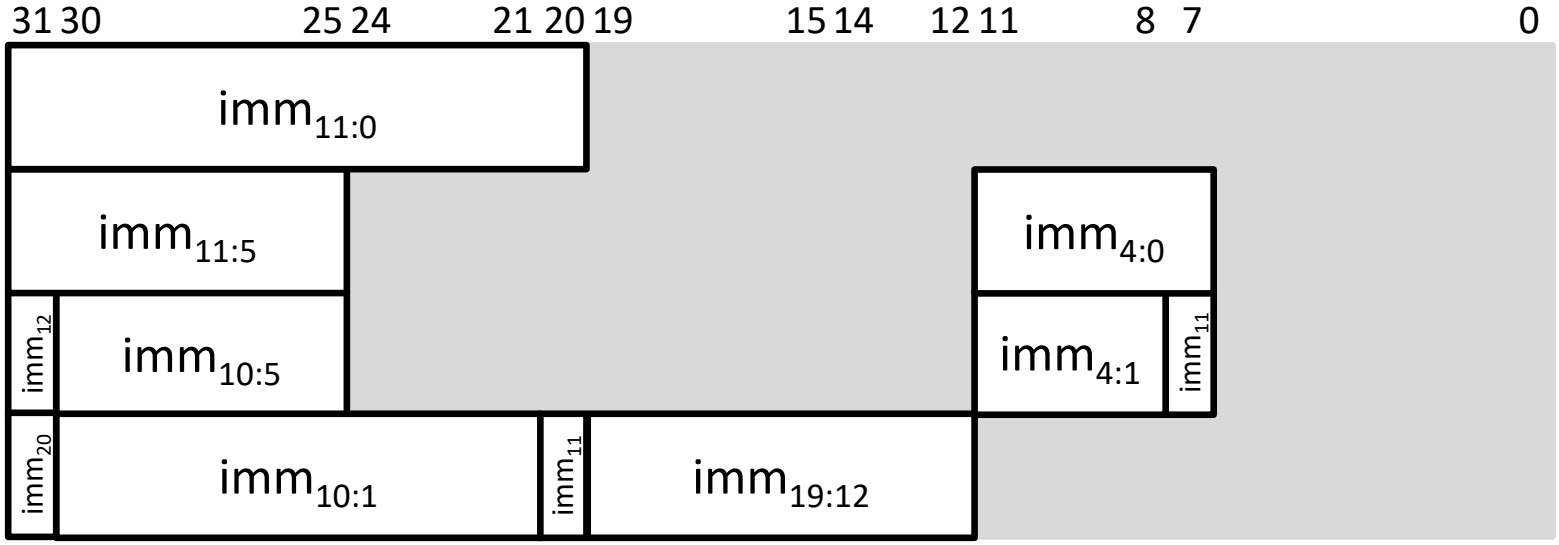
- The Sign Extension module is a combinational module that builds the 32-bit immediate operand based on the information contained in the imm fields of the instruction:
  - For each instruction type, the imm field has a different size and position within the instruction.
  - Therefore, apart from extending the sign, it must reorder the bits and fill with 0 in the case of branch addresses.

Sign Extension module design



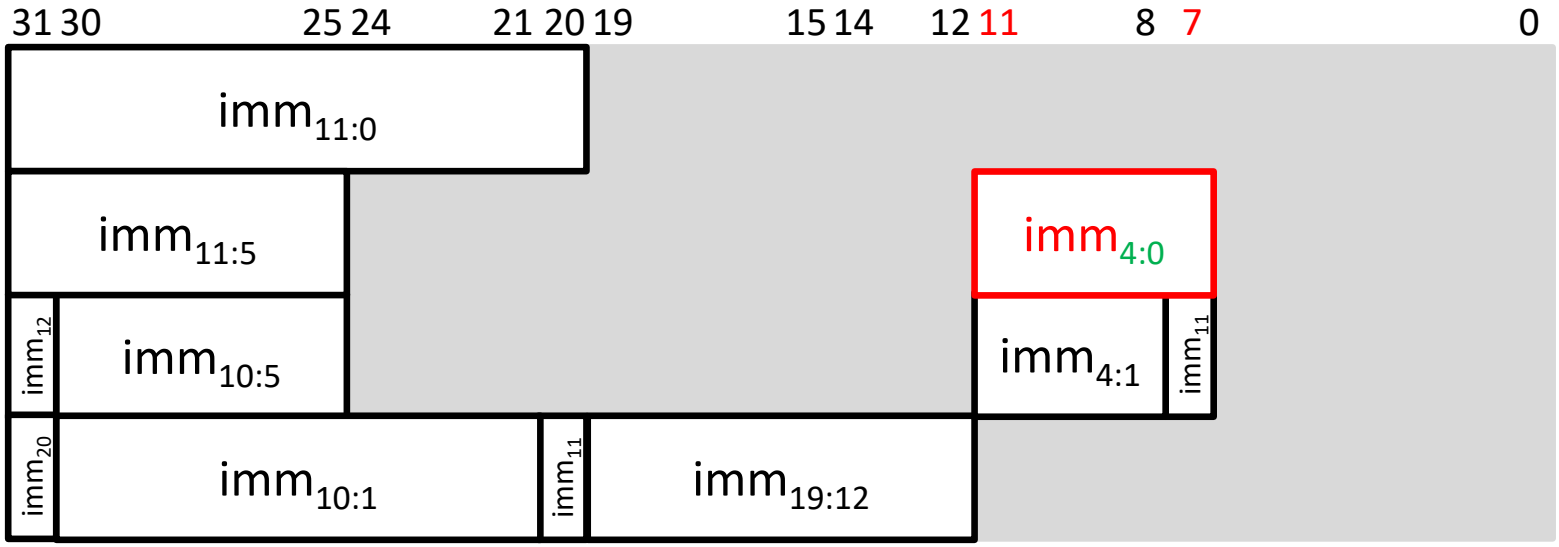
op	z31	z30	z29	z28	z27	z26	z25	z24	z23	z22	z21	z20	z19	z18	z17	z16	z15	z14	z13	z12	z11	z10	z9	z8	z7	z6	z5	z4	z3	z2	z1	z0
00																					x31	x30	x29	x28	x27	x26	x25	x24	x23	x22	x21	x20

Sign Extension module design



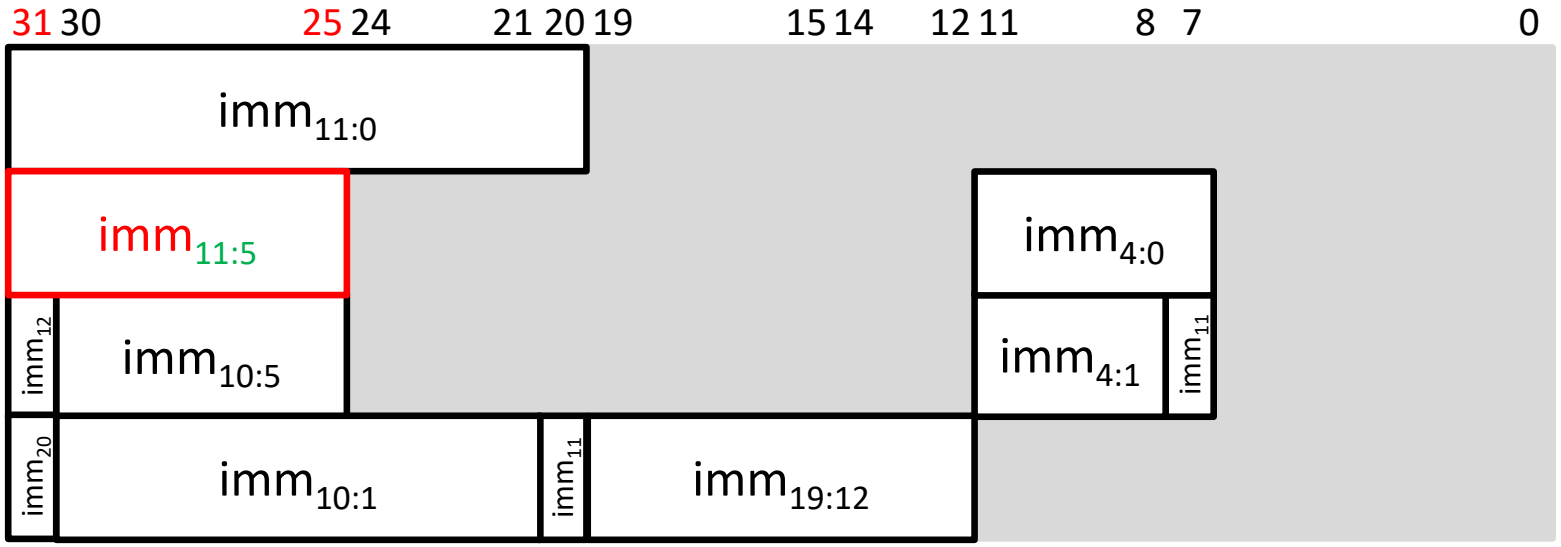
op	z_{31}	z_{30}	z_{29}	z_{28}	z_{27}	z_{26}	z_{25}	z_{24}	z_{23}	z_{22}	z_{21}	z_{20}	z_{19}	z_{18}	z_{17}	z_{16}	z_{15}	z_{14}	z_{13}	z_{12}	z_{11}	z_{10}	z_9	z_8	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
00	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	x_{20}

Sign Extension module design



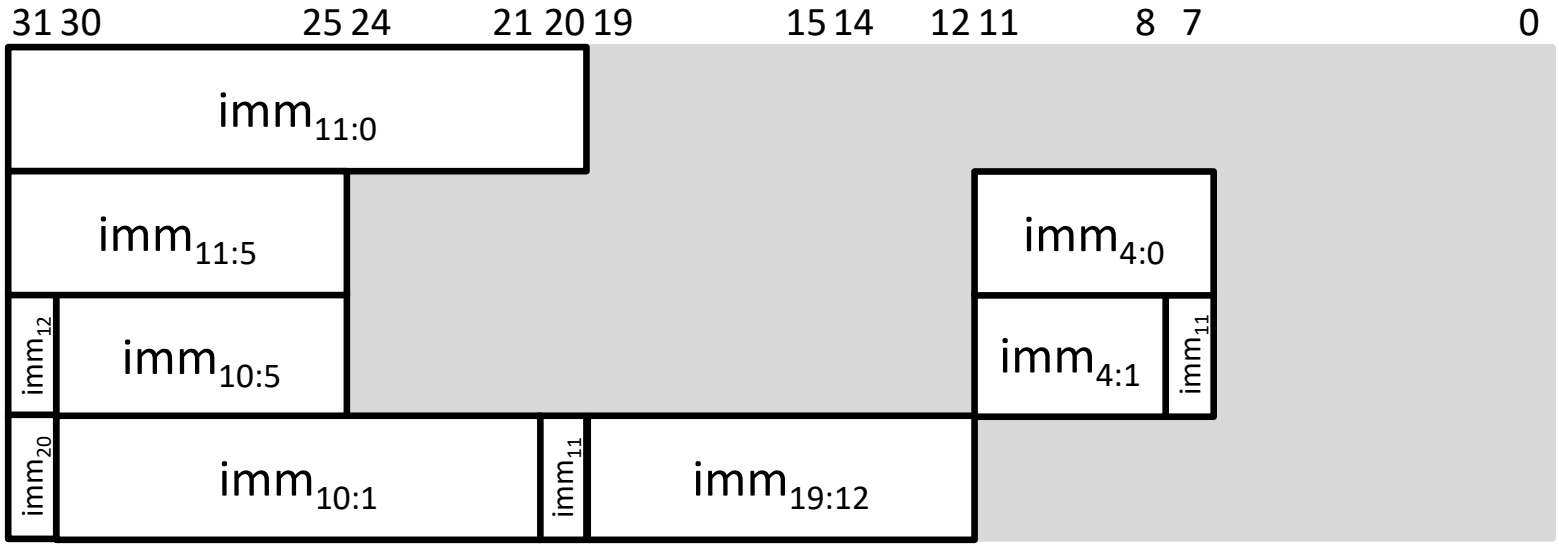
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{11} x_{10} x_9 x_8 x_7

Sign Extension module design



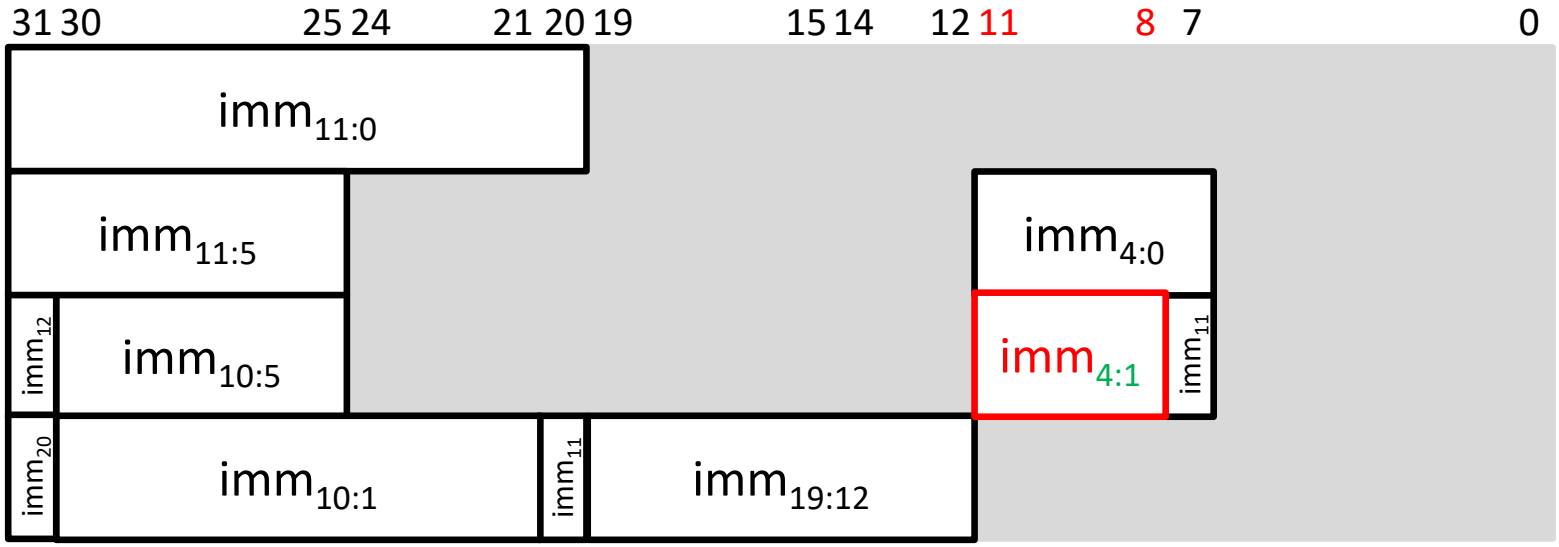
op	z_{31}	z_{30}	z_{29}	z_{28}	z_{27}	z_{26}	z_{25}	z_{24}	z_{23}	z_{22}	z_{21}	z_{20}	z_{19}	z_{18}	z_{17}	z_{16}	z_{15}	z_{14}	z_{13}	z_{12}	z_{11}	z_{10}	z_9	z_8	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
00	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	x_{20}
01																					x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{11}	x_{10}	x_9	x_8	x_7

Sign Extension module design



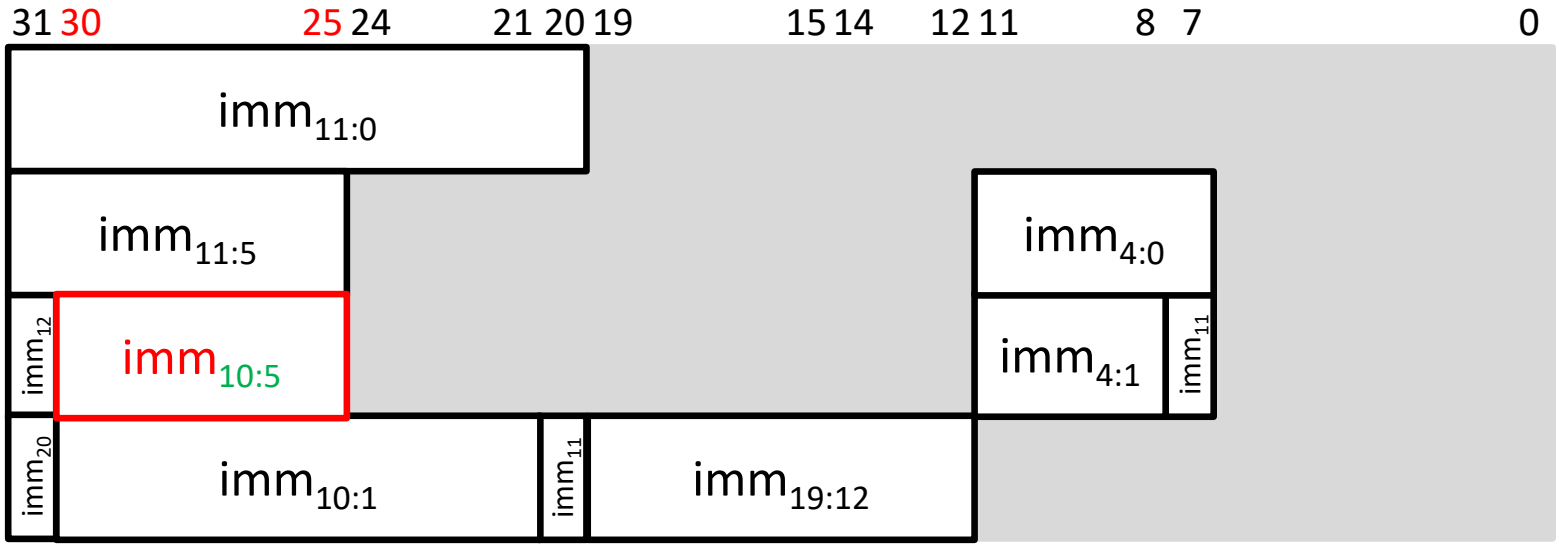
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7

Sign Extension module design



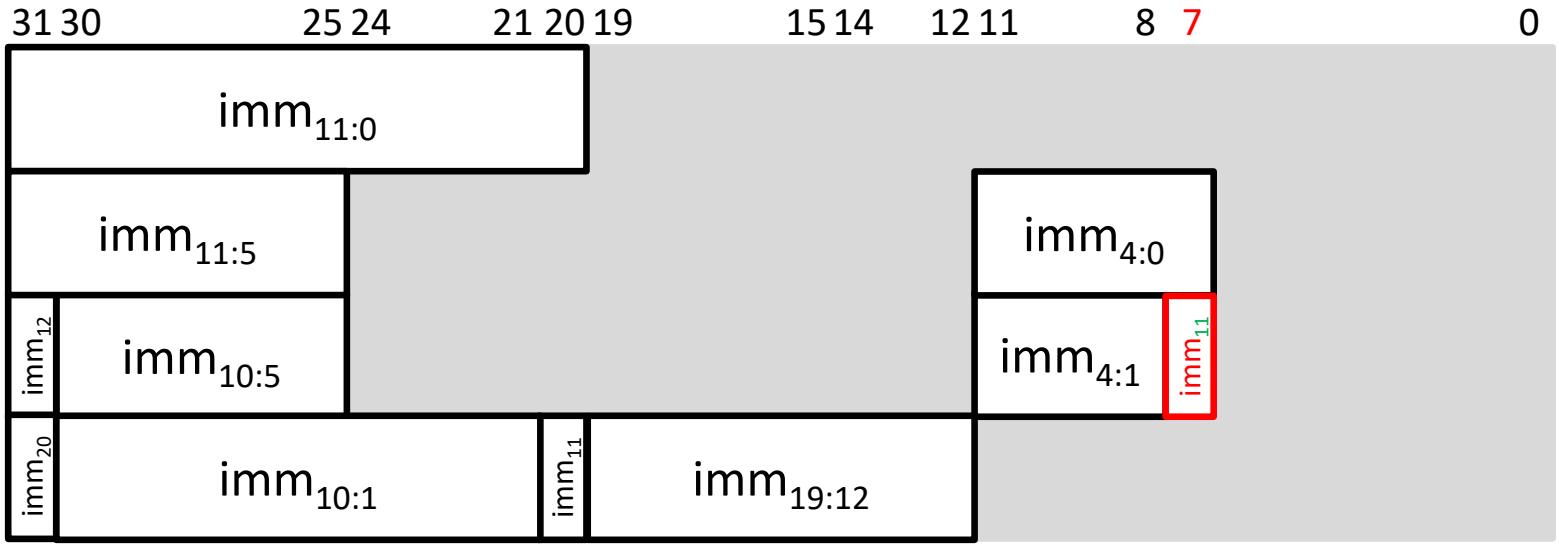
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7
10	x_{11} x_{10} x_9 x_8

Sign Extension module design



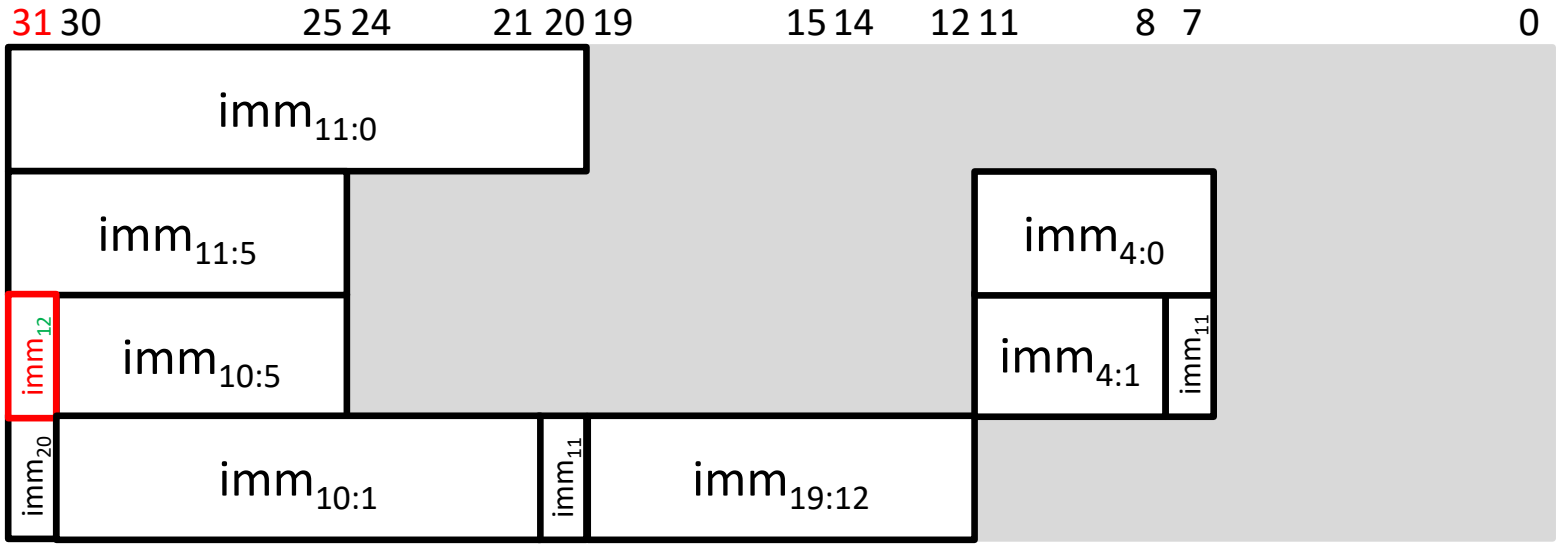
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7
10	x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8

Sign Extension module design



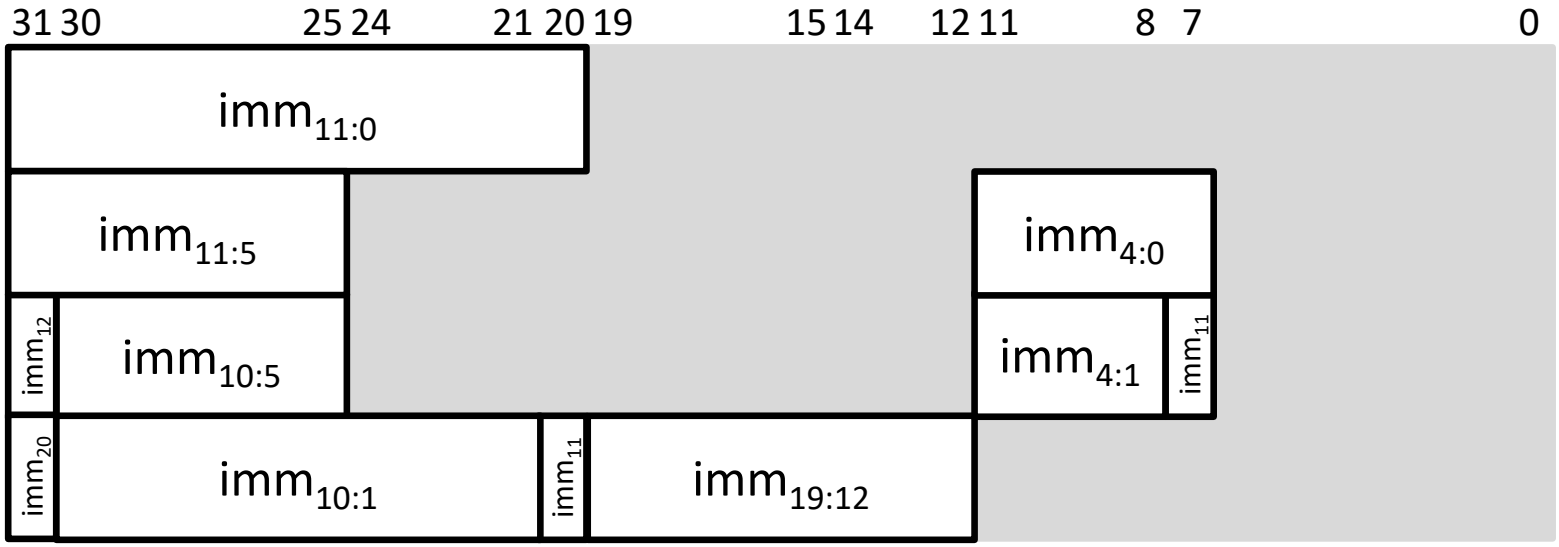
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7
10	x_7 x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8

Sign Extension module design



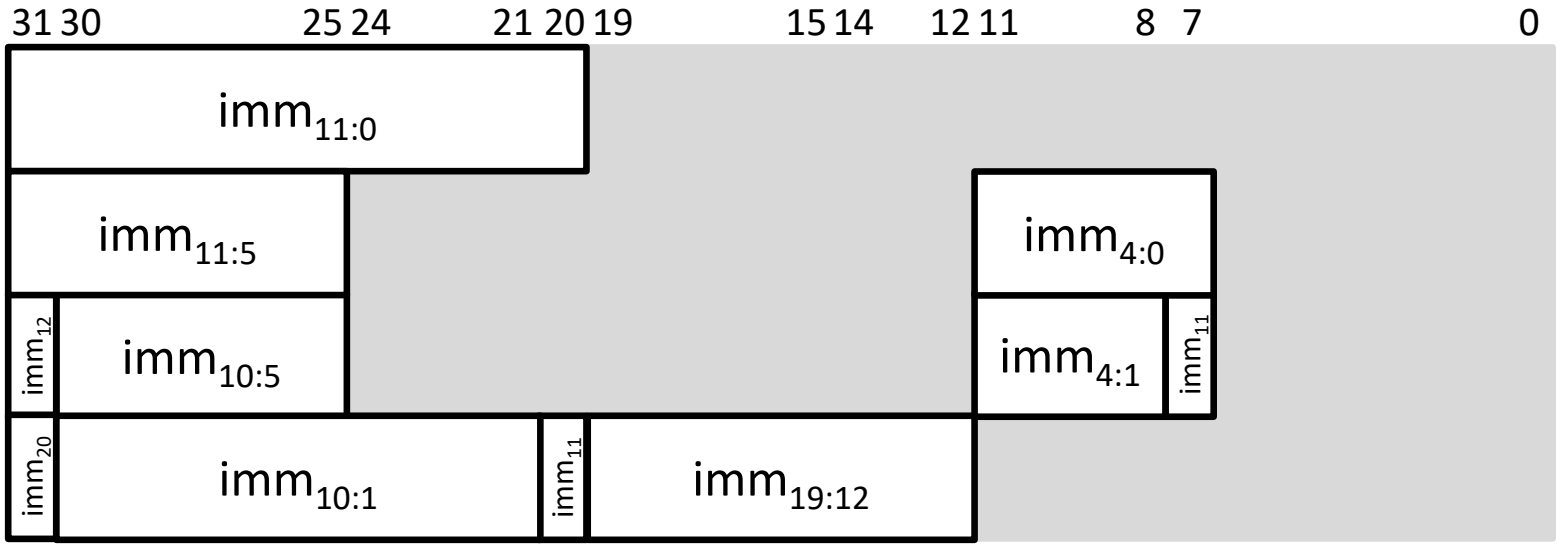
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7
10	x_{31} x_7 x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8

Sign Extension module design



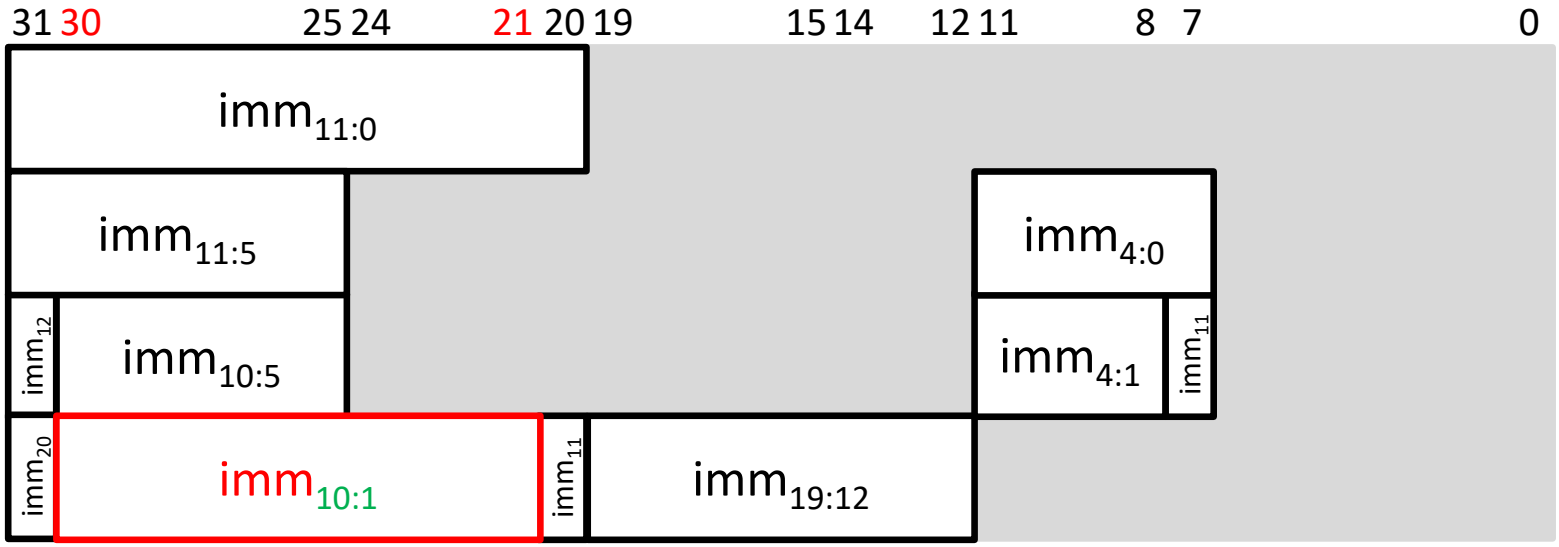
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7
10	x_{31} x_7 x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 0

Sign Extension module design



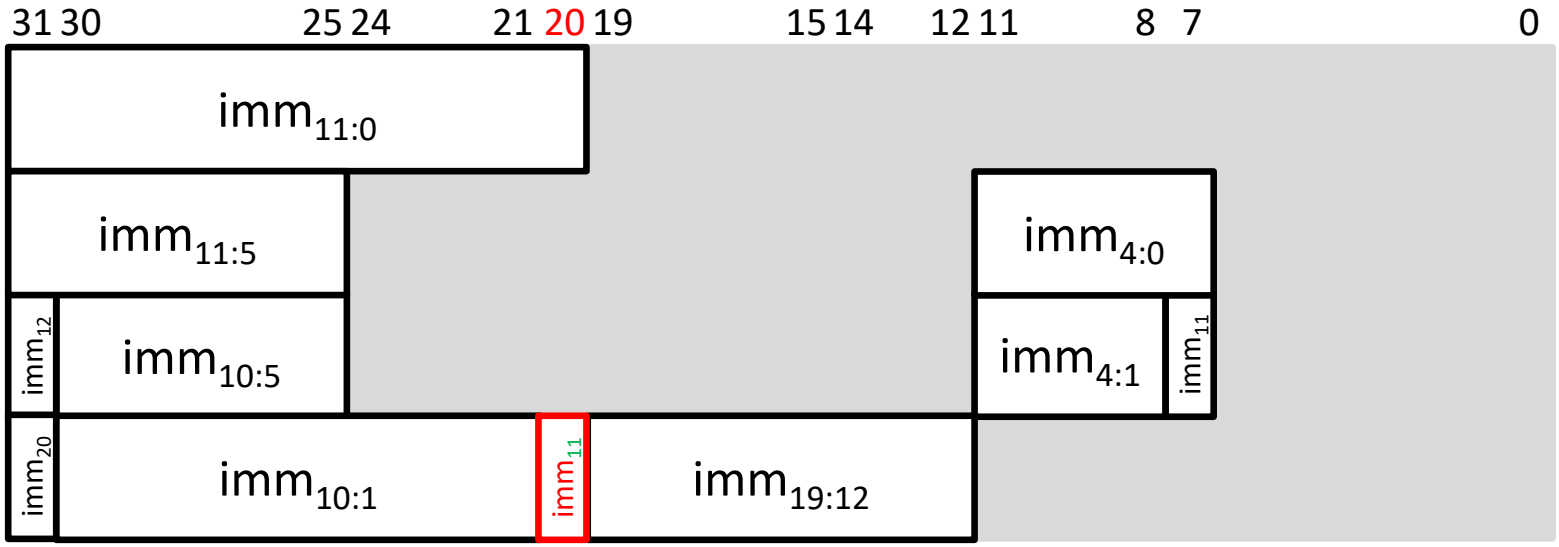
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7
10	x_{31} x_7 x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 0

Sign Extension module design



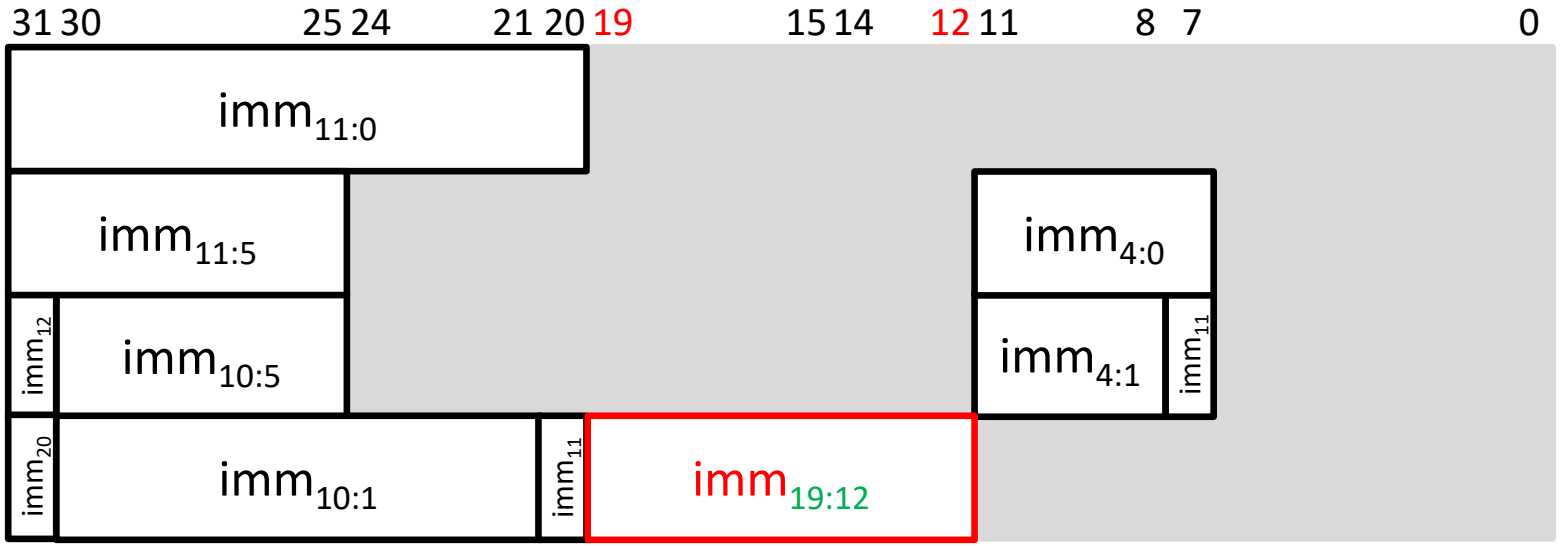
op	z_{31}	z_{30}	z_{29}	z_{28}	z_{27}	z_{26}	z_{25}	z_{24}	z_{23}	z_{22}	z_{21}	z_{20}	z_{19}	z_{18}	z_{17}	z_{16}	z_{15}	z_{14}	z_{13}	z_{12}	z_{11}	z_{10}	z_9	z_8	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
00	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	x_{20}
01	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{11}	x_{10}	x_9	x_8	x_7
10	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_7	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{11}	x_{10}	x_9	x_8	0
11																						x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	

Sign Extension module design



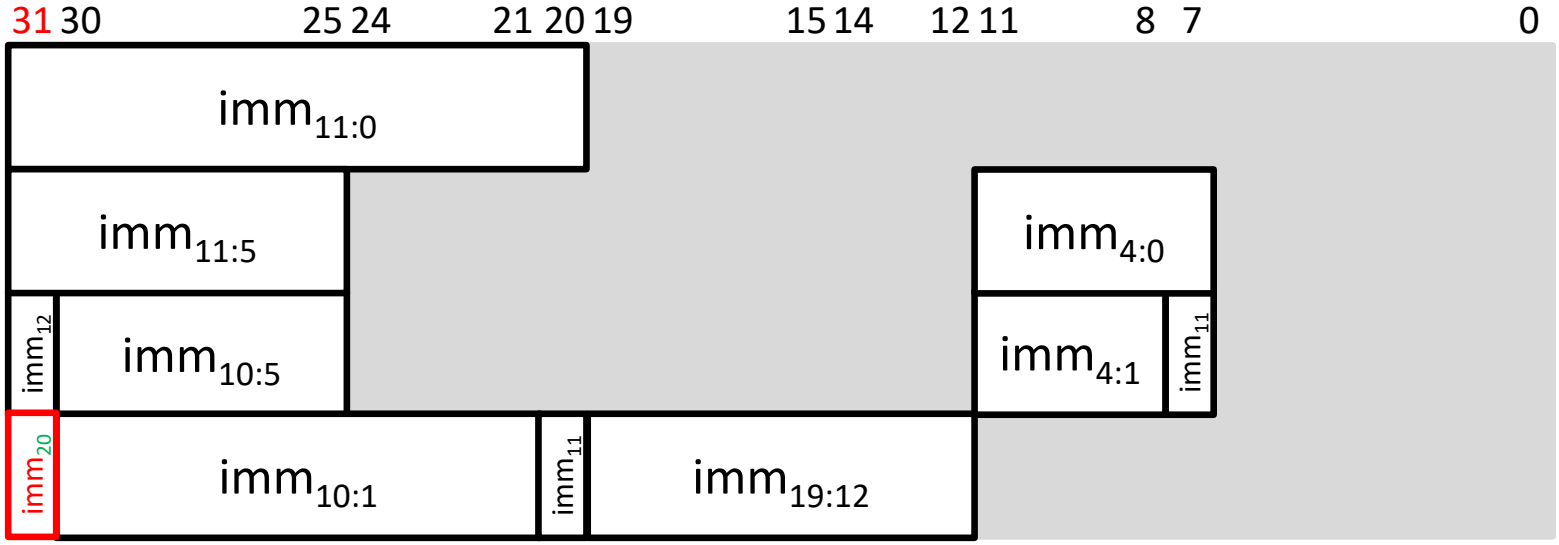
op	z_{31}	z_{30}	z_{29}	z_{28}	z_{27}	z_{26}	z_{25}	z_{24}	z_{23}	z_{22}	z_{21}	z_{20}	z_{19}	z_{18}	z_{17}	z_{16}	z_{15}	z_{14}	z_{13}	z_{12}	z_{11}	z_{10}	z_9	z_8	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
00	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	x_{20}
01	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{11}	x_{10}	x_9	x_8	x_7
10	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_7	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{11}	x_{10}	x_9	x_8	0
11																					x_{20}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	

Sign Extension module design



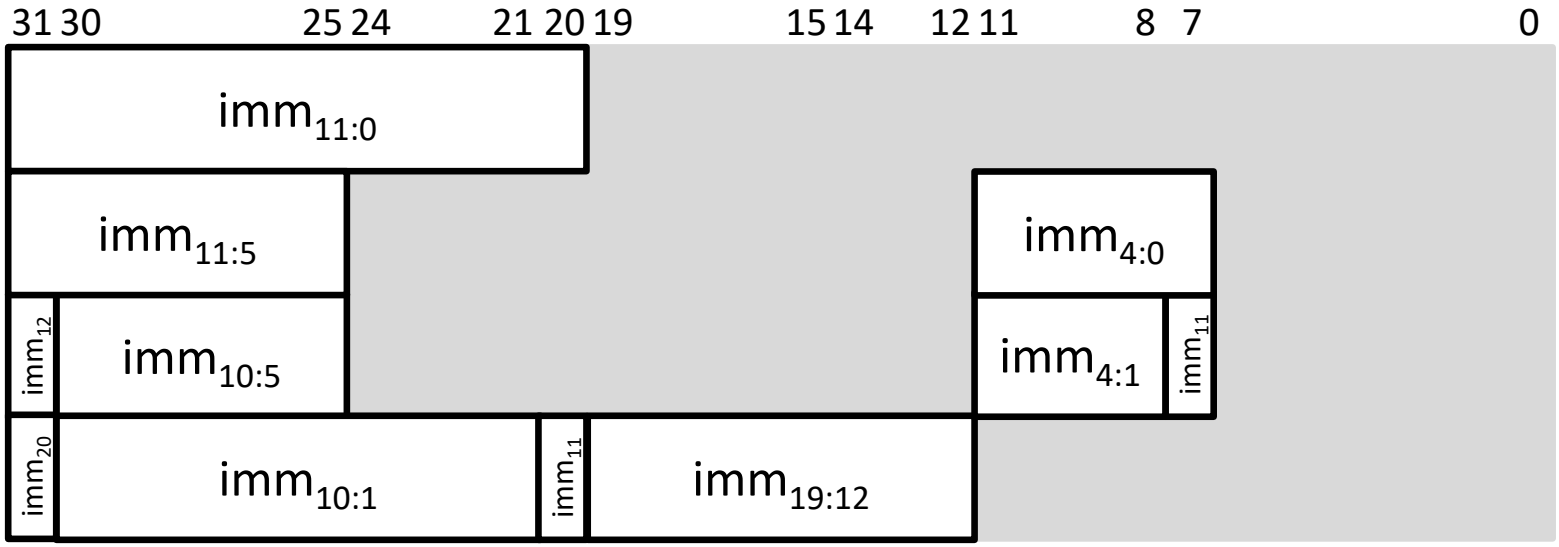
op	z_{31}	z_{30}	z_{29}	z_{28}	z_{27}	z_{26}	z_{25}	z_{24}	z_{23}	z_{22}	z_{21}	z_{20}	z_{19}	z_{18}	z_{17}	z_{16}	z_{15}	z_{14}	z_{13}	z_{12}	z_{11}	z_{10}	z_9	z_8	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
00	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	x_{20}
01	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{11}	x_{10}	x_9	x_8	x_7
10	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_7	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{11}	x_{10}	x_9	x_8	0
11													x_{19}	x_{18}	x_{17}	x_{16}	x_{15}	x_{14}	x_{13}	x_{12}	x_{20}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	

Sign Extension module design



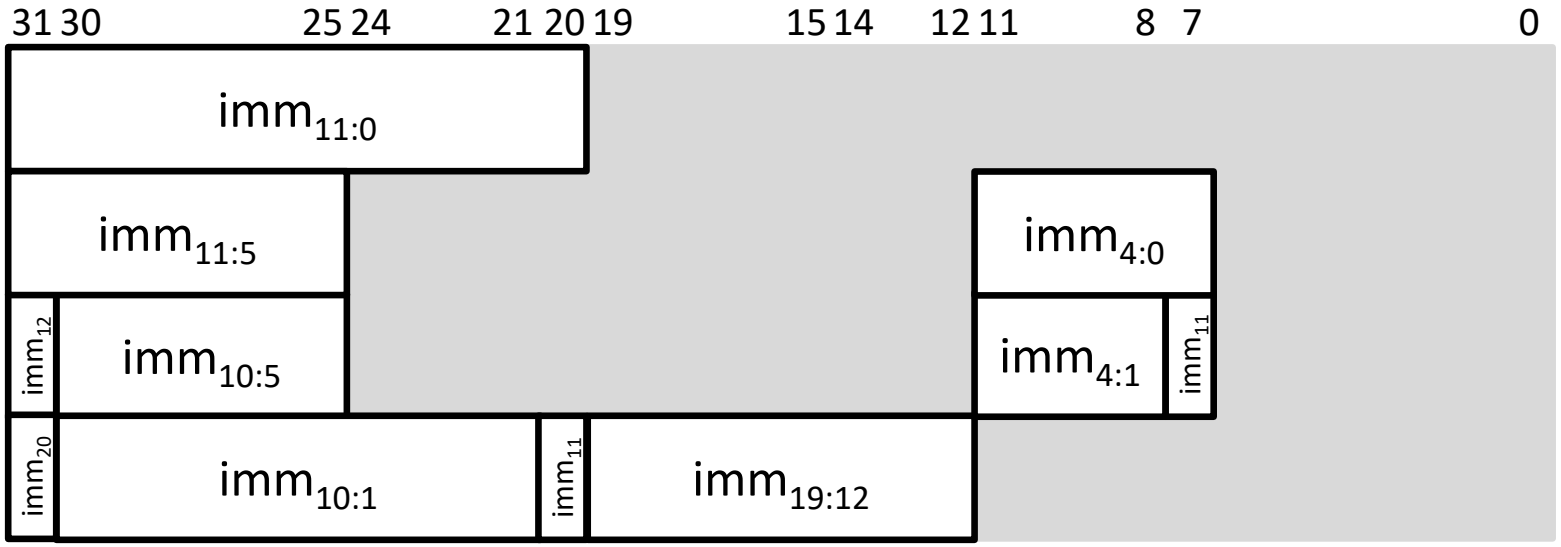
op	z_{31}	z_{30}	z_{29}	z_{28}	z_{27}	z_{26}	z_{25}	z_{24}	z_{23}	z_{22}	z_{21}	z_{20}	z_{19}	z_{18}	z_{17}	z_{16}	z_{15}	z_{14}	z_{13}	z_{12}	z_{11}	z_{10}	z_9	z_8	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
00	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	x_{20}
01	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{11}	x_{10}	x_9	x_8	x_7
10	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_7	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{11}	x_{10}	x_9	x_8	0
11												x_{31}	x_{19}	x_{18}	x_{17}	x_{16}	x_{15}	x_{14}	x_{13}	x_{12}	x_{20}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	

Sign Extension module design



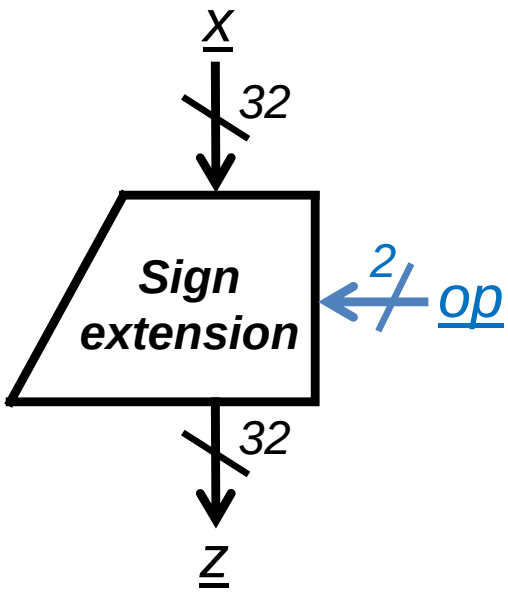
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7
10	x_{31} x_7 x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 0
11	x_{31} x_{19} x_{18} x_{17} x_{16} x_{15} x_{14} x_{13} x_{12} x_{20} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} 0

Sign Extension module design



op	z_{31}	z_{30}	z_{29}	z_{28}	z_{27}	z_{26}	z_{25}	z_{24}	z_{23}	z_{22}	z_{21}	z_{20}	z_{19}	z_{18}	z_{17}	z_{16}	z_{15}	z_{14}	z_{13}	z_{12}	z_{11}	z_{10}	z_9	z_8	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
00	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	x_{20}
01	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{11}	x_{10}	x_9	x_8	x_7
10	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_7	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{11}	x_{10}	x_9	x_8	0
11	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{31}	x_{19}	x_{18}	x_{17}	x_{16}	x_{15}	x_{14}	x_{13}	x_{12}	x_{20}	x_{30}	x_{29}	x_{28}	x_{27}	x_{26}	x_{25}	x_{24}	x_{23}	x_{22}	x_{21}	0

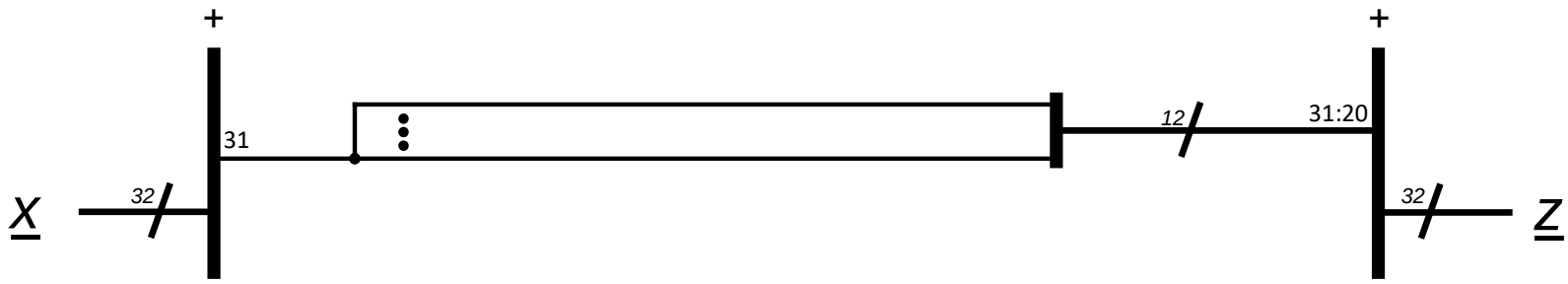
Sign Extension module design



x	1 32-bit data input (instruction)
op	1 operation selector input
z	1 32-bit data output (immediate operand)

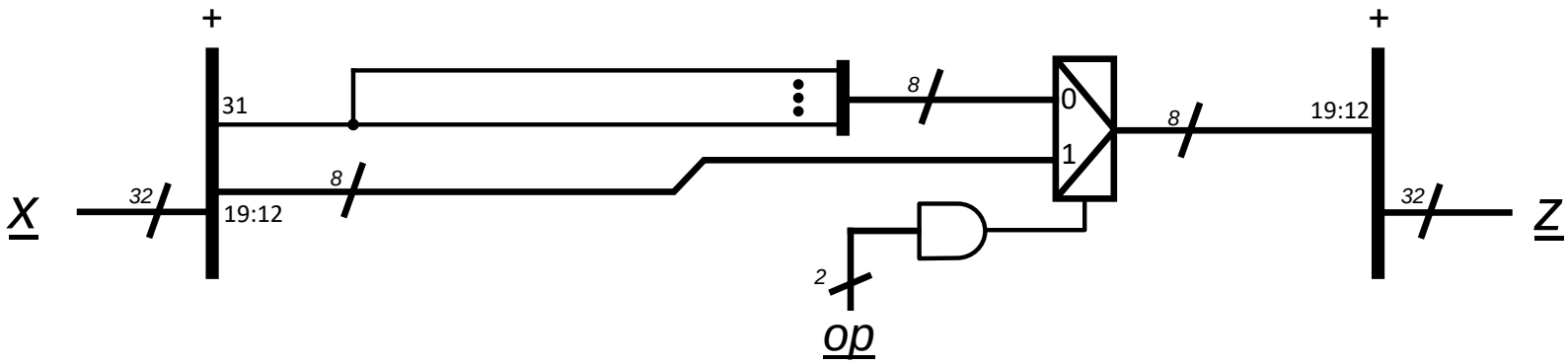
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7
10	x_{31} x_7 x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 0
11	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{19} x_{18} x_{17} x_{16} x_{15} x_{14} x_{13} x_{12} x_{20} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} 0

Sign Extension module design



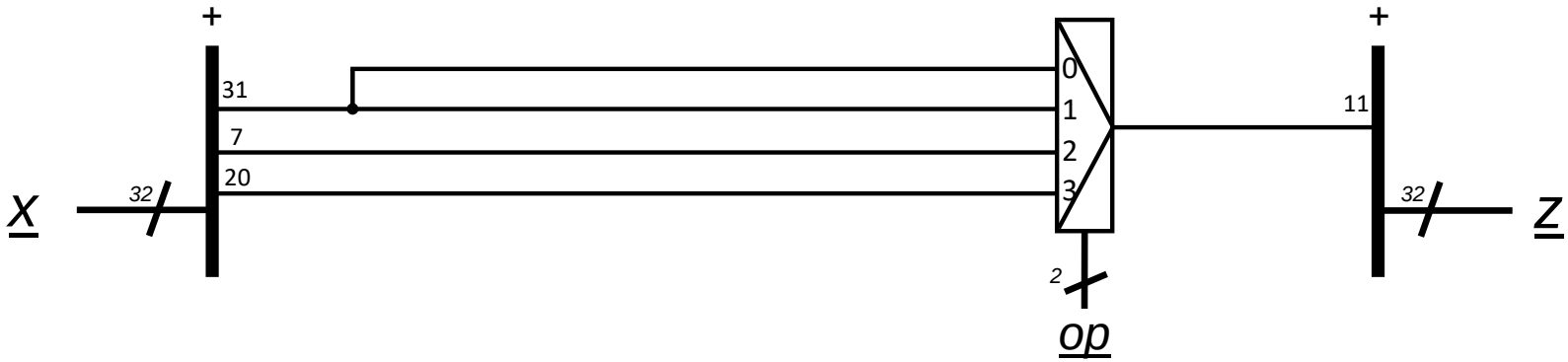
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20}	z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7
10	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_7 x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 0
11	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{19} x_{18} x_{17} x_{16} x_{15} x_{14} x_{13} x_{12} x_{20} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} 0

Sign Extension module design



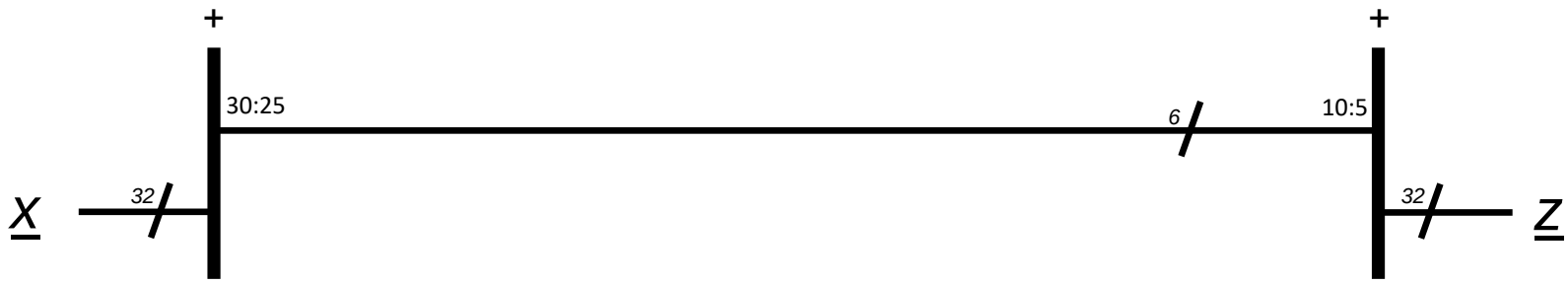
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20}	z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12}	z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7
10	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_7 x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 0
11	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{19} x_{18} x_{17} x_{16} x_{15} x_{14} x_{13} x_{12}	x_{20} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} 0

Sign Extension module design



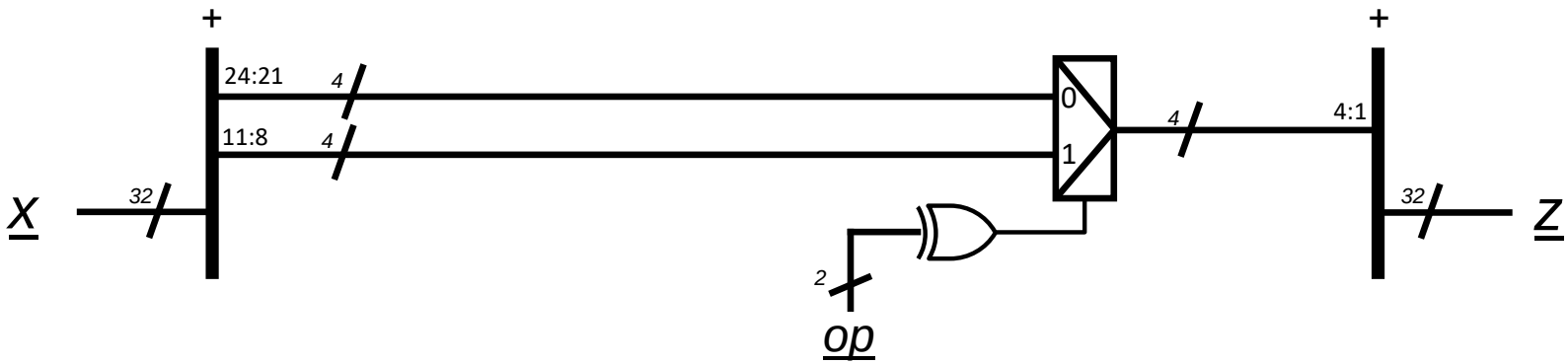
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12}	z_{11}	z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{31}	x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_{31}	x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 x_7
10	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31}	x_7	x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8 0
11	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{19} x_{18} x_{17} x_{16} x_{15} x_{14} x_{13} x_{12}	x_{20}	x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21} 0

Sign Extension module design



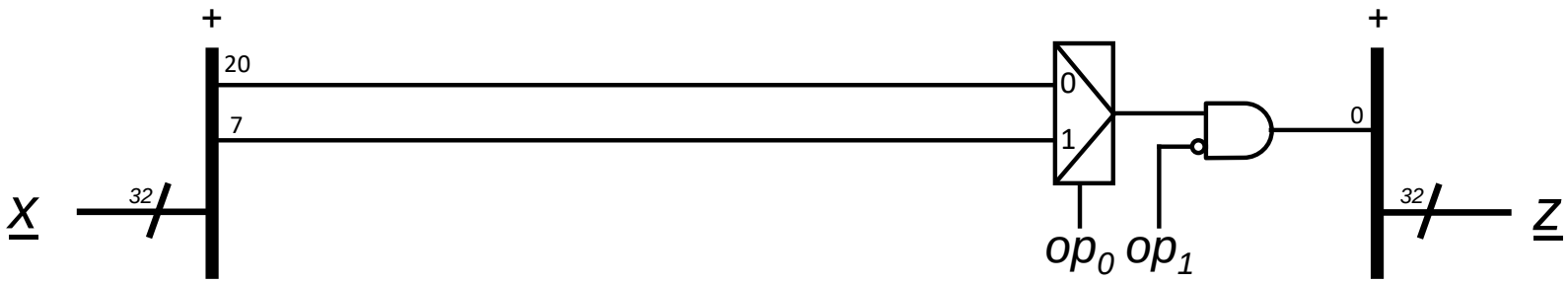
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11}	z_{10} z_9 z_8 z_7 z_6 z_5	z_4 z_3 z_2 z_1 z_0
00	x_{31} x_{31}	x_{30} x_{29} x_{28} x_{27} x_{26} x_{25}	x_{24} x_{23} x_{22} x_{21} x_{20}
01	x_{31} x_{31}	x_{30} x_{29} x_{28} x_{27} x_{26} x_{25}	x_{11} x_{10} x_9 x_8 x_7
10	x_{31} x_7	x_{30} x_{29} x_{28} x_{27} x_{26} x_{25}	x_{11} x_{10} x_9 x_8 0
11	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{19} x_{18} x_{17} x_{16} x_{15} x_{14} x_{13} x_{12} x_{20}	x_{30} x_{29} x_{28} x_{27} x_{26} x_{25}	x_{24} x_{23} x_{22} x_{21} 0

Sign Extension module design



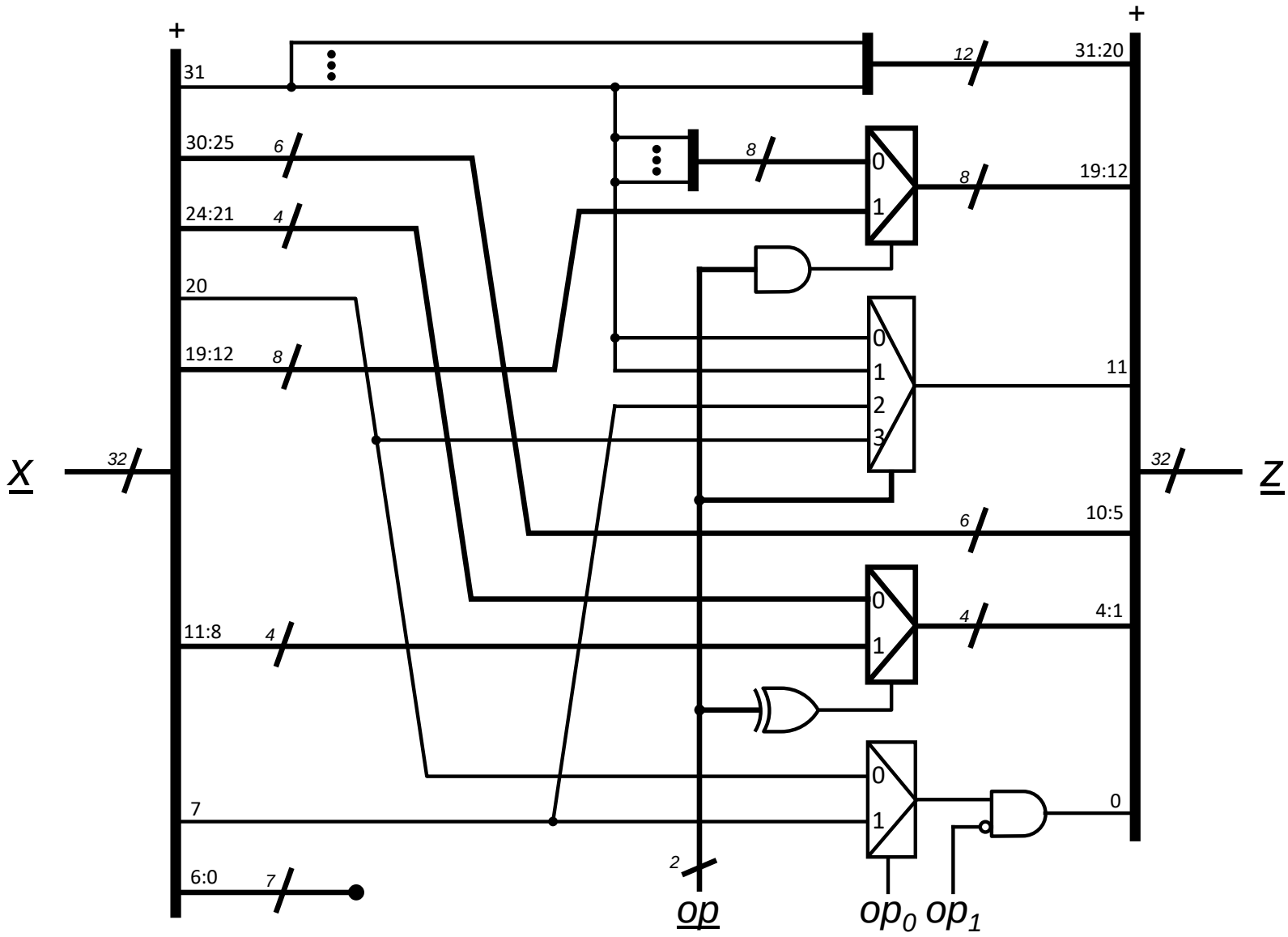
op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5	z_4 z_3 z_2 z_1	z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25}	x_{24} x_{23} x_{22} x_{21}	x_{20}
01	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25}	x_{11} x_{10} x_9 x_8	x_7
10	x_{31} x_7 x_{30} x_{29} x_{28} x_{27} x_{26} x_{25}	x_{11} x_{10} x_9 x_8	0
11	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{19} x_{18} x_{17} x_{16} x_{15} x_{14} x_{13} x_{12} x_{20} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25}	x_{24} x_{23} x_{22} x_{21}	0

Sign Extension module design



op	z_{31} z_{30} z_{29} z_{28} z_{27} z_{26} z_{25} z_{24} z_{23} z_{22} z_{21} z_{20} z_{19} z_{18} z_{17} z_{16} z_{15} z_{14} z_{13} z_{12} z_{11} z_{10} z_9 z_8 z_7 z_6 z_5 z_4 z_3 z_2 z_1	z_0
00	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21}	x_{20}
01	x_{31} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8	x_7
10	x_{31} x_7 x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{11} x_{10} x_9 x_8	0
11	x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{31} x_{19} x_{18} x_{17} x_{16} x_{15} x_{14} x_{13} x_{12} x_{20} x_{30} x_{29} x_{28} x_{27} x_{26} x_{25} x_{24} x_{23} x_{22} x_{21}	0

Sign Extension module design



Cost and cycle time calculation

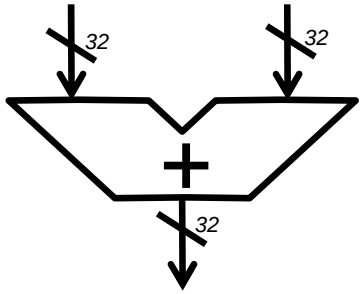
- The processor cost is the addition of the costs of each of the components that form it.
  - The cost of each component is calculated by adding the cost of its cells.

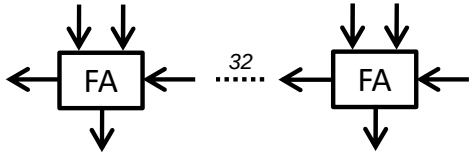
- The processor cycle time is the maximum critical path of the register transfers performed by the processor.
  - The critical path of a register transfer is the data path with the largest delay among all the paths involved in that transfer.
  - In the single-cycle processor, one instruction involving 1 or 2 register transfers is executed per cycle: the PC update and the specific of each instruction.

- The same cell library (90nm CMOS) used in FC-1 will be utilized for all the calculations.

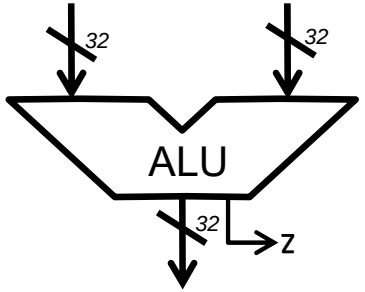
Cost and cycle time calculation

90 nm CMOS

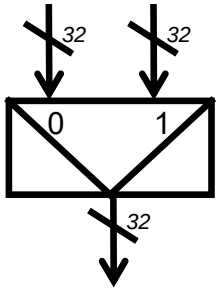


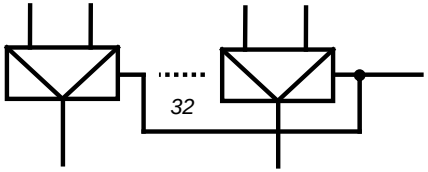


area: 32×29.49 = **944 μm²**
delay: 32×226 = **7,232 ps**

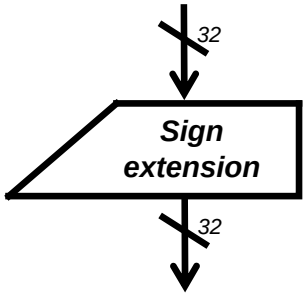


area: **3,052 μm²**
delay: **8,360 ps**

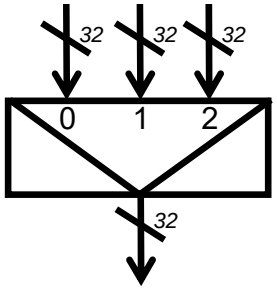


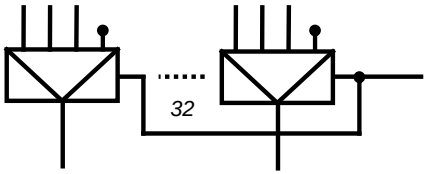


area: 32×11.05 = **354 μm²**
delay: **223 ps**

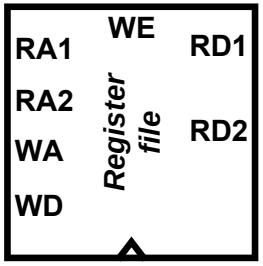


area: **202 μm²**
delay: **460 ps**





area: 32×23.04 = **737 μm²**
delay: **250 ps**



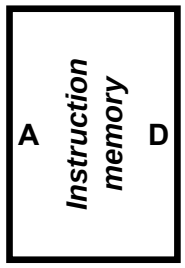
area: **51,405 μm²**
read delay: **723 ps**
write setup: **705 ps**
(due to the address DEC)

90 nm CMOS cell library, source: Synopsys (SAED EDK 90 nm)

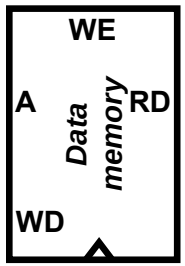
Cost and cycle time calculation

90 nm CMOS

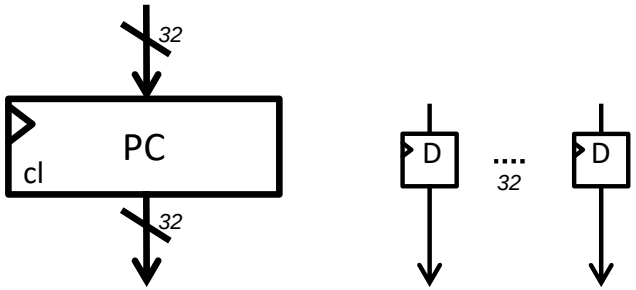
Idealized behavior: delay comparable to the one of the ALU
(so that it can be read in one clock cycle)



area: –
access time: 8,500 ps



area: –
access time: 8,500 ps



area: 32×32.26 = **1,032 μm²**
CLK→Q delay: 1×167 = **167 ps**
setup: **0 ps**

area: **56 μm²**
delay: **490 ps**

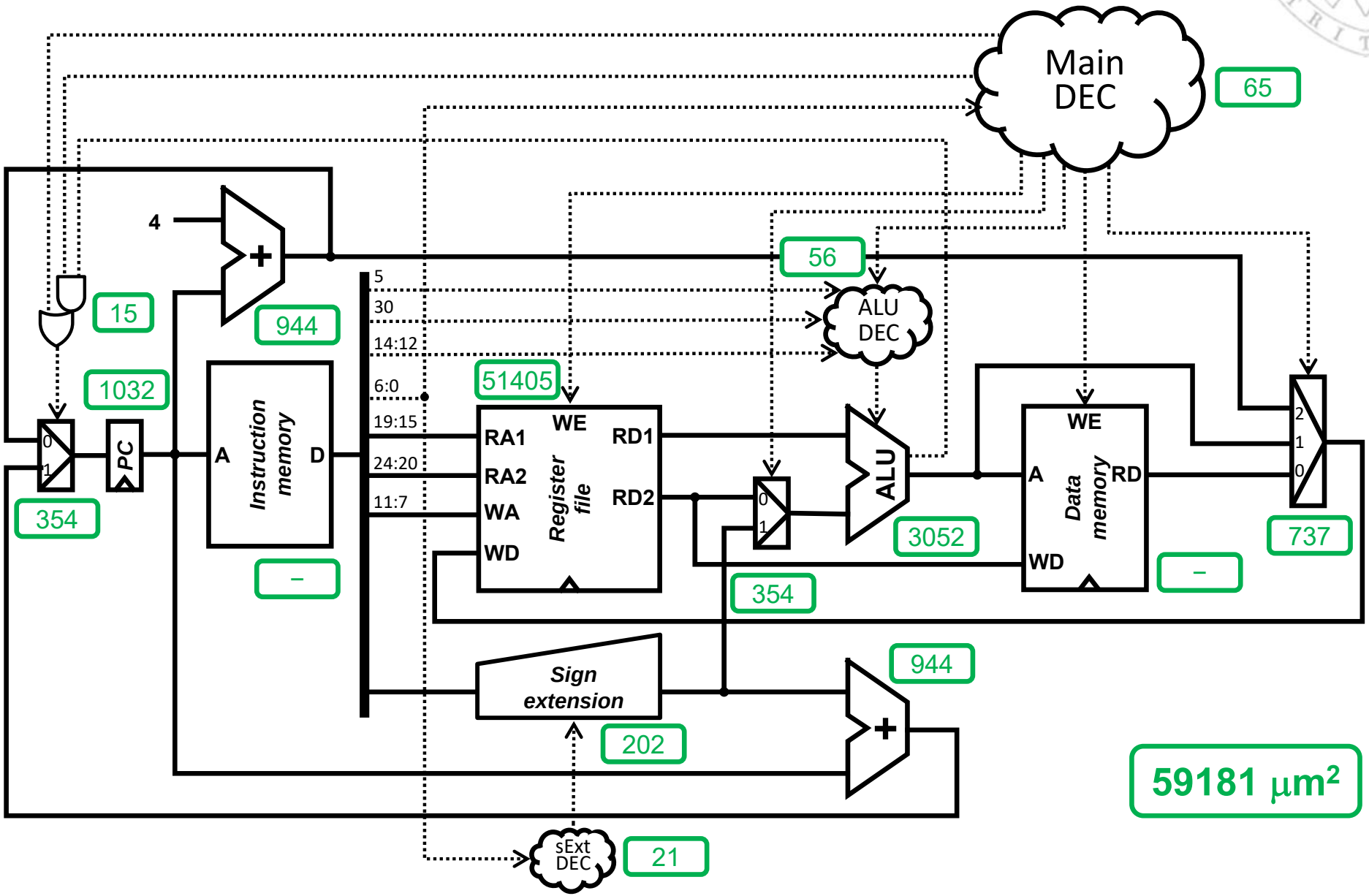
area: **65 μm²**
delay: **451 ps**

area: **21 μm²**
delay: **451 ps**

area: **15 μm²**
delay: **351 ps**

90 nm CMOS cell library, source: Synopsys (SAED EDK 90 nm)

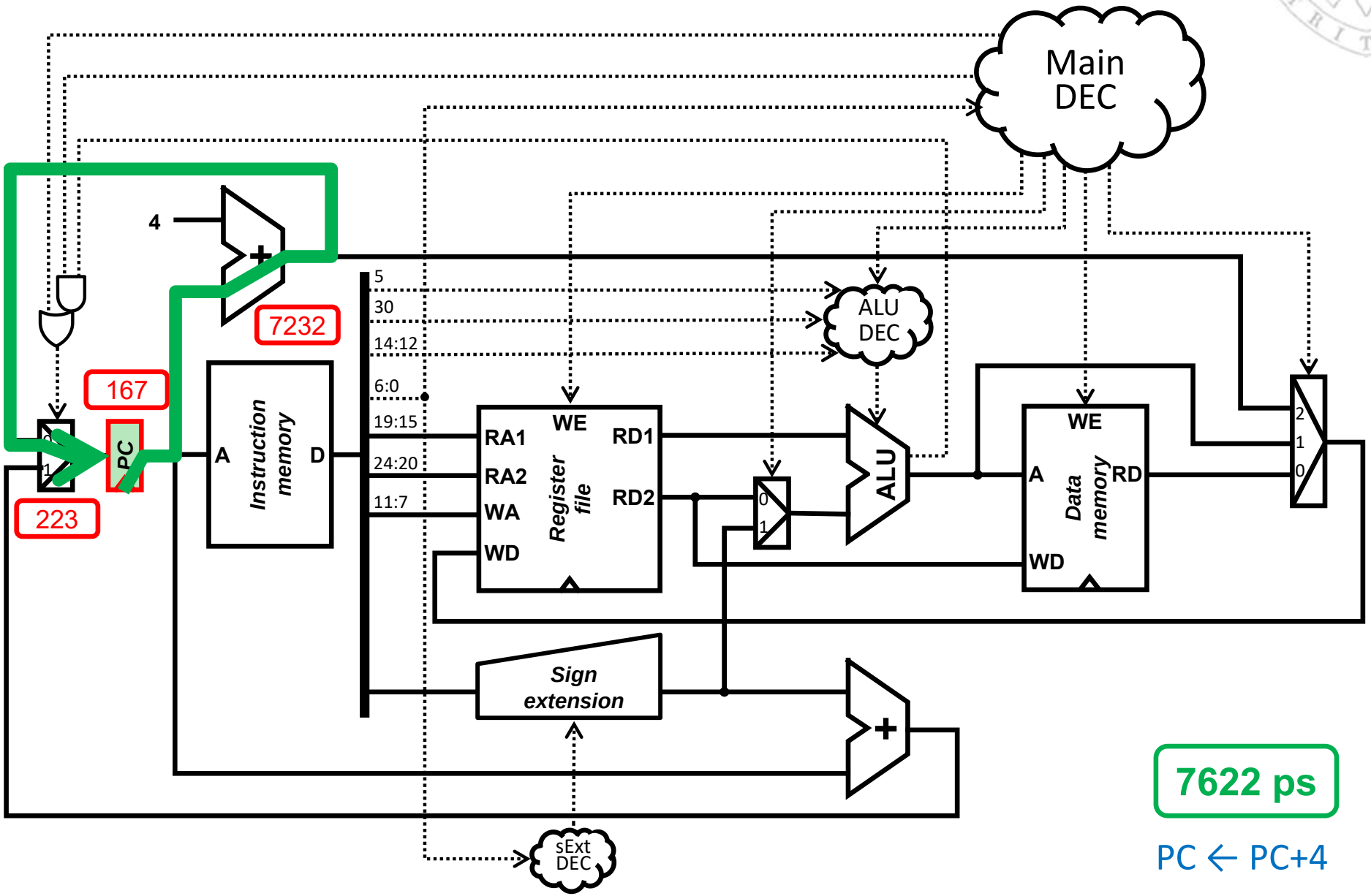
Cost calculation



59181 μm²

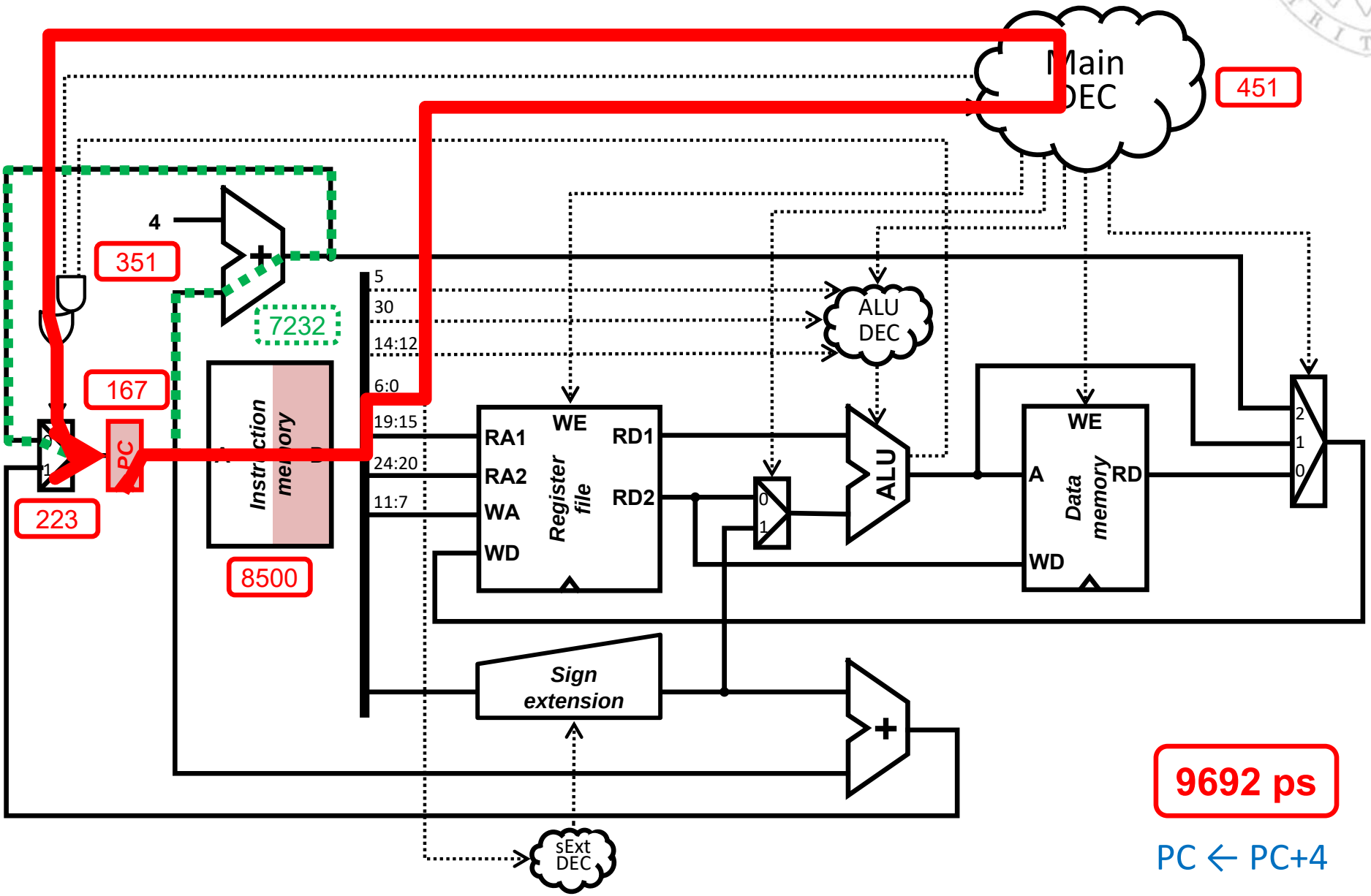
Cycle time calculation

PC increment



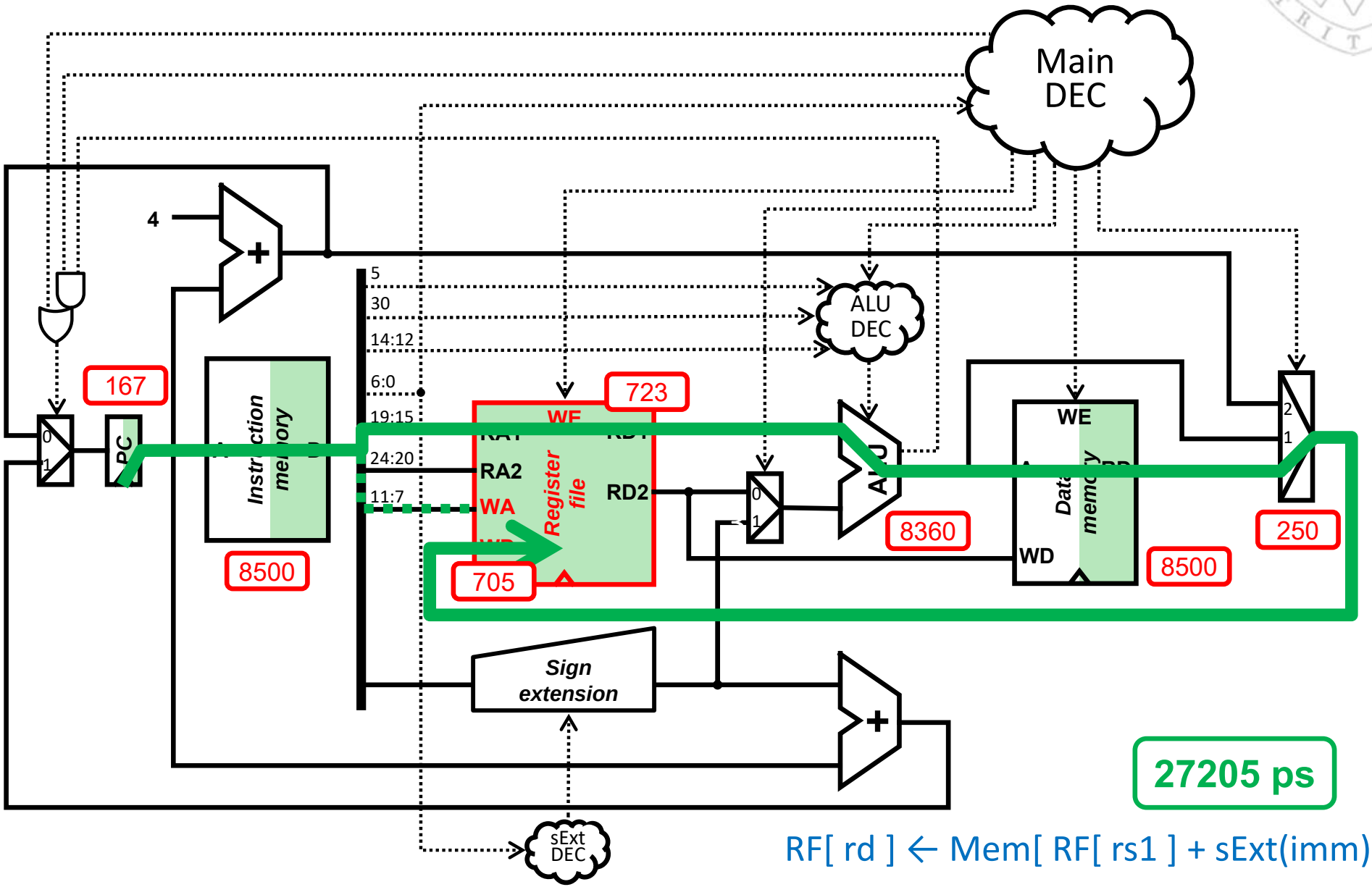
Cycle time calculation

PC increment: critical path



Cycle time calculation

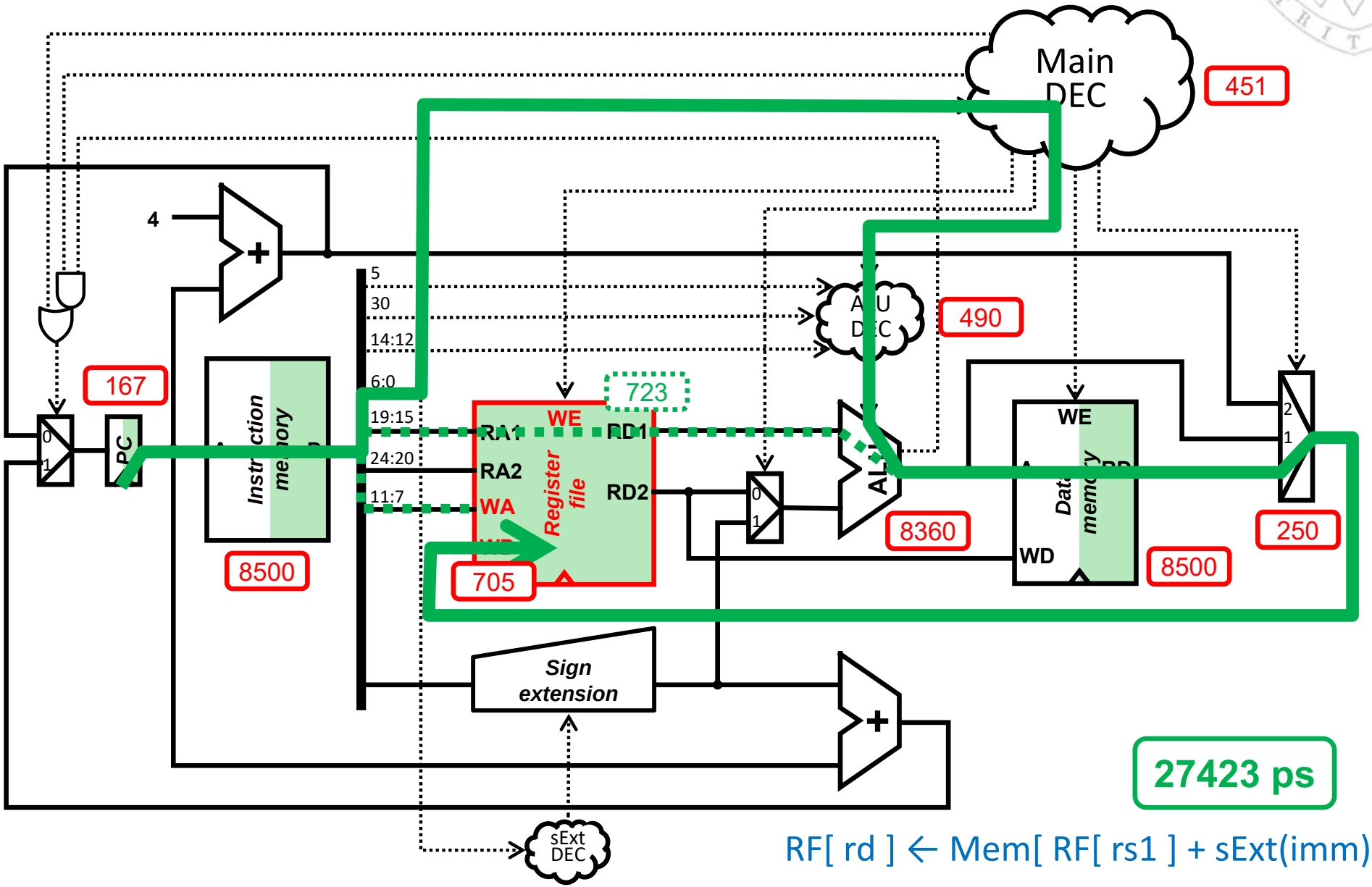
lw instruction (i)



RF[rd] ← Mem[RF[rs1] + sExt(imm)]

Cycle time calculation

lw instruction (ii)

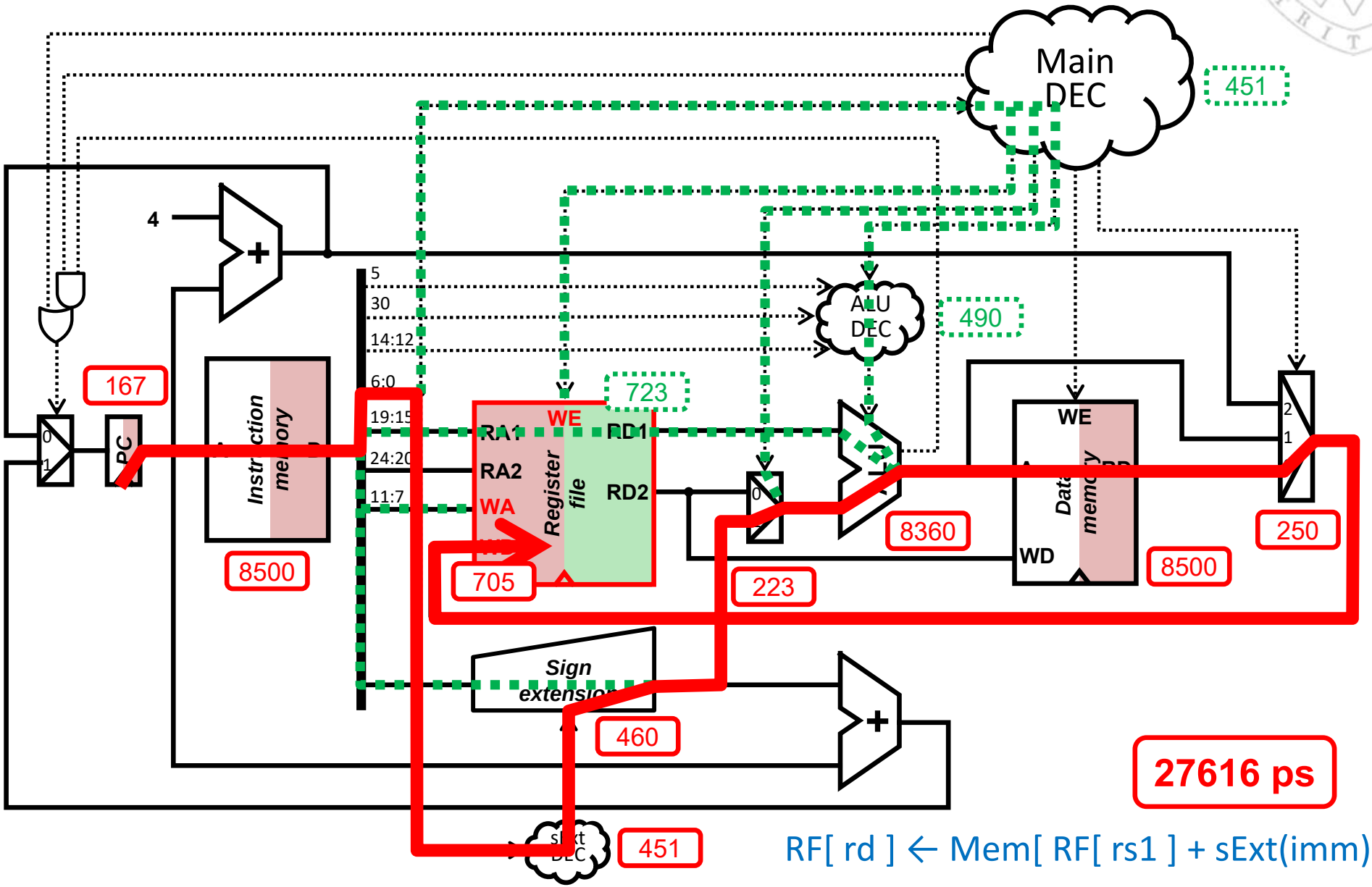


27423 ps

RF[rd] ← Mem[RF[rs1] + sExt(imm)]

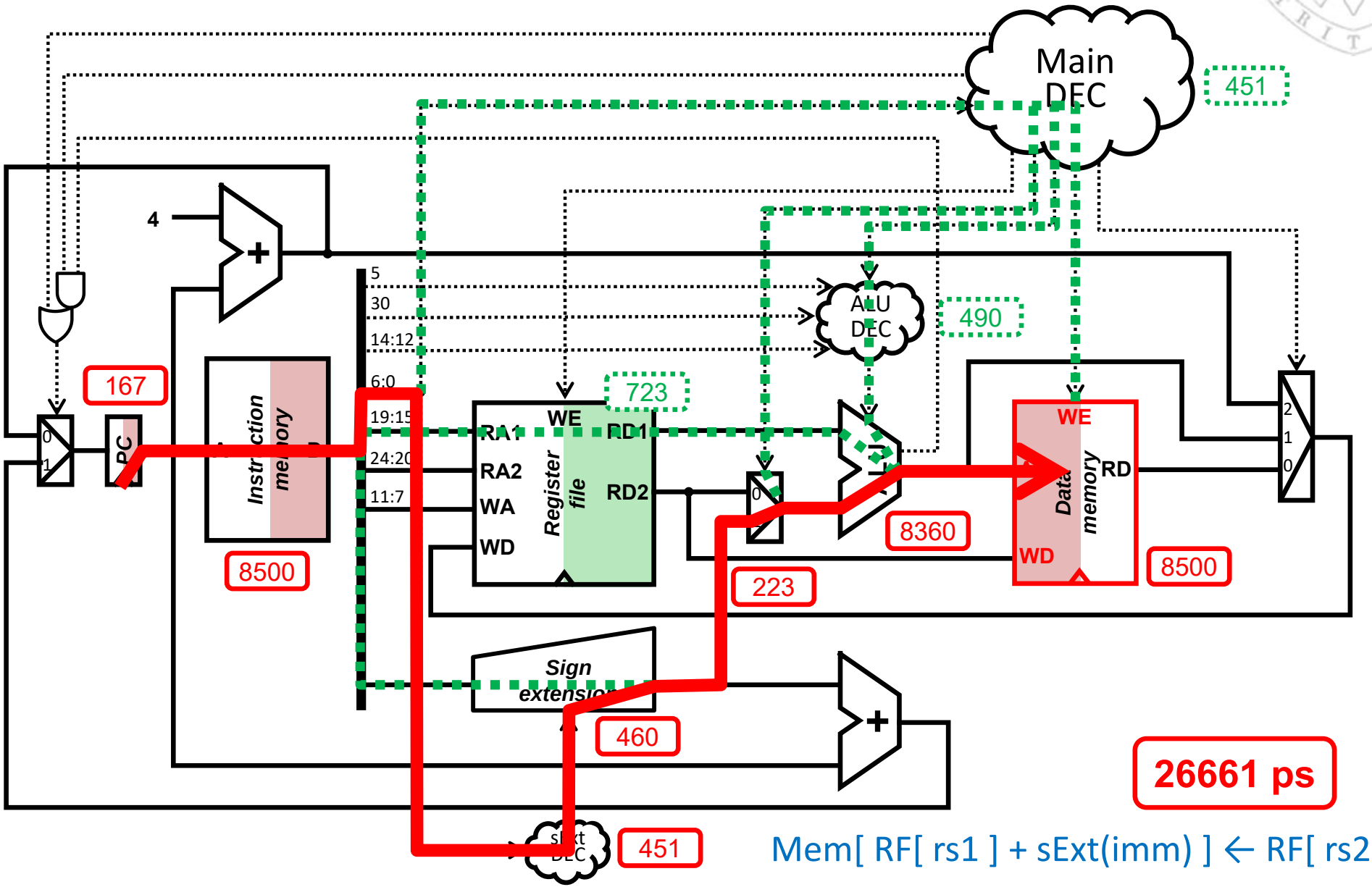
Cycle time calculation

lw instruction: critical path



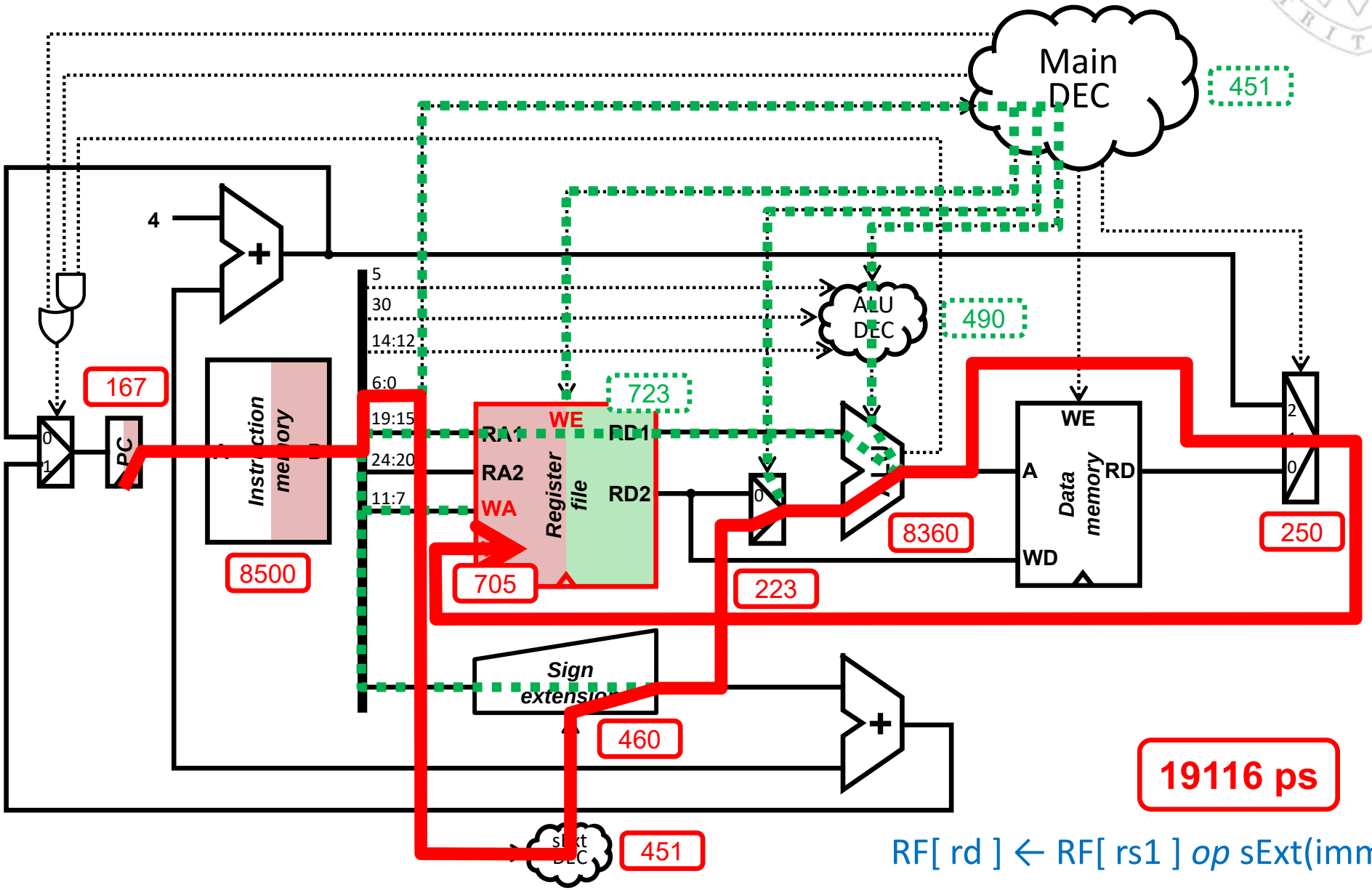
Cycle time calculation

sw instruction: critical path



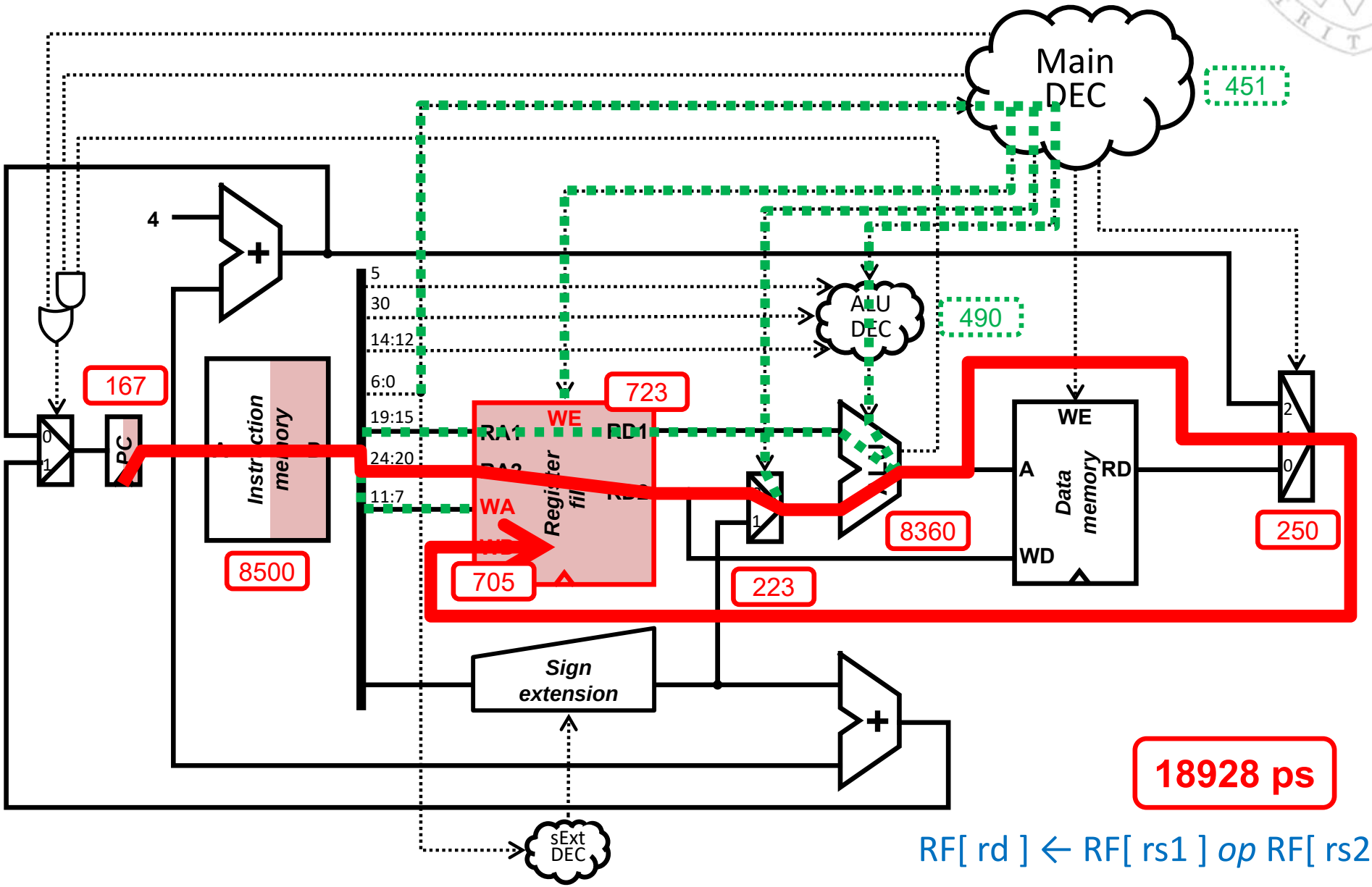
Cycle time calculation

`addi`-like instructions: critical path



Cycle time calculation

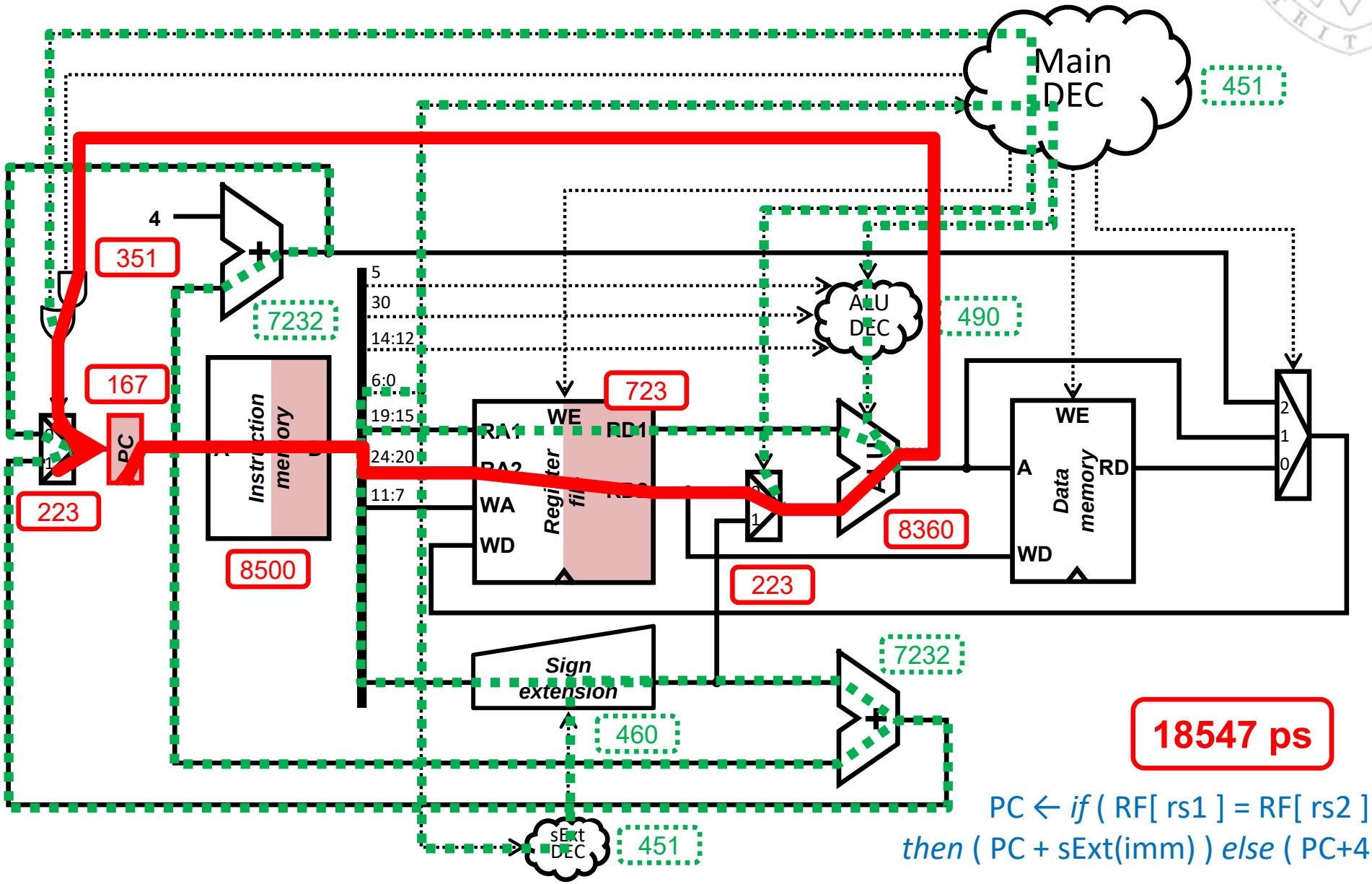
add-like instructions: critical path



RF[rd] ← RF[rs1] *op* RF[rs2]

Cycle time calculation

beq instruction: critical path

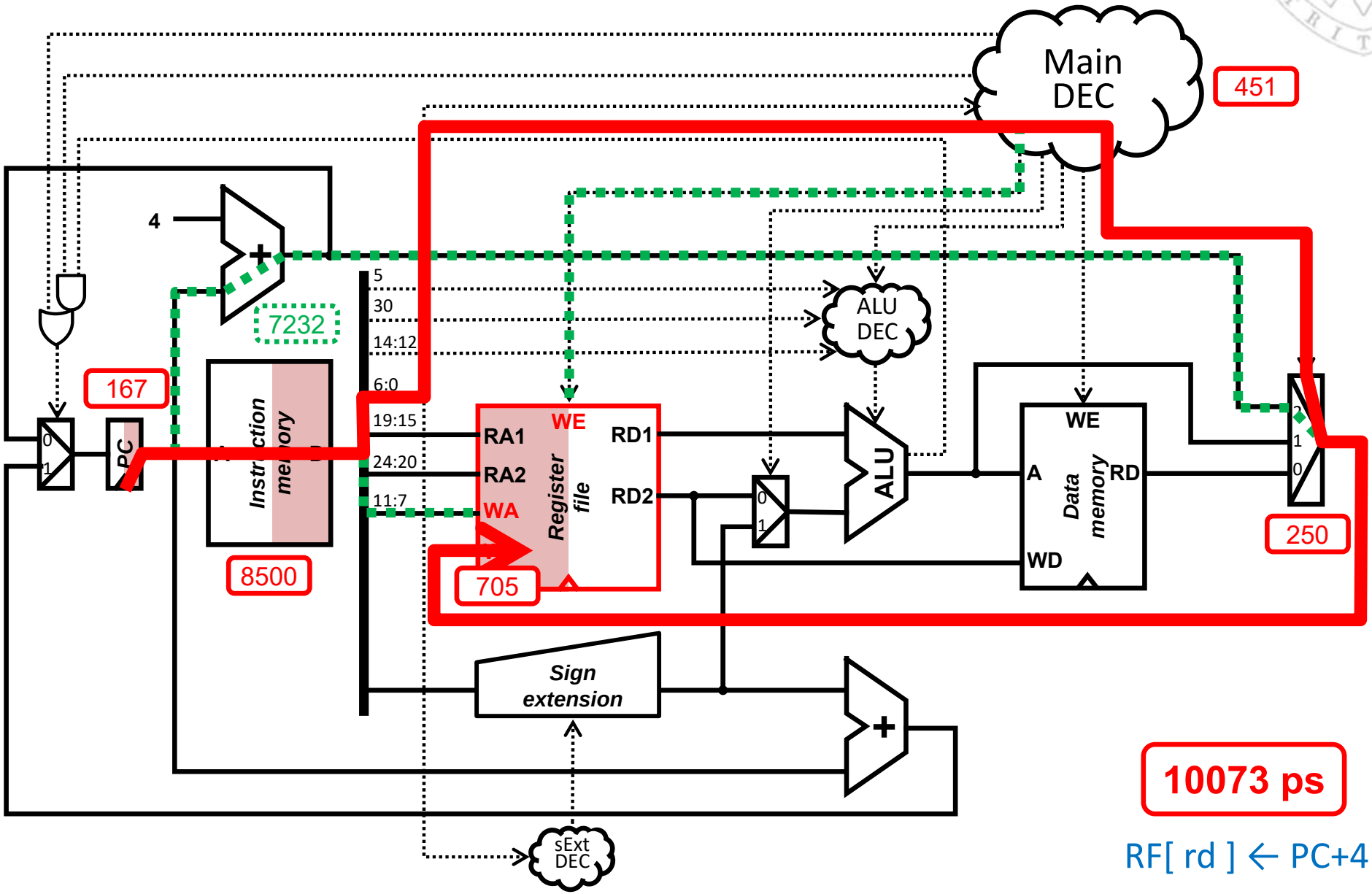


18547 ps

PC ← *if* (RF[rs1] = RF[rs2])
then (PC + sExt(imm)) *else* (PC+4)

Cycle time calculation

`jal` instruction (return address store): critical path

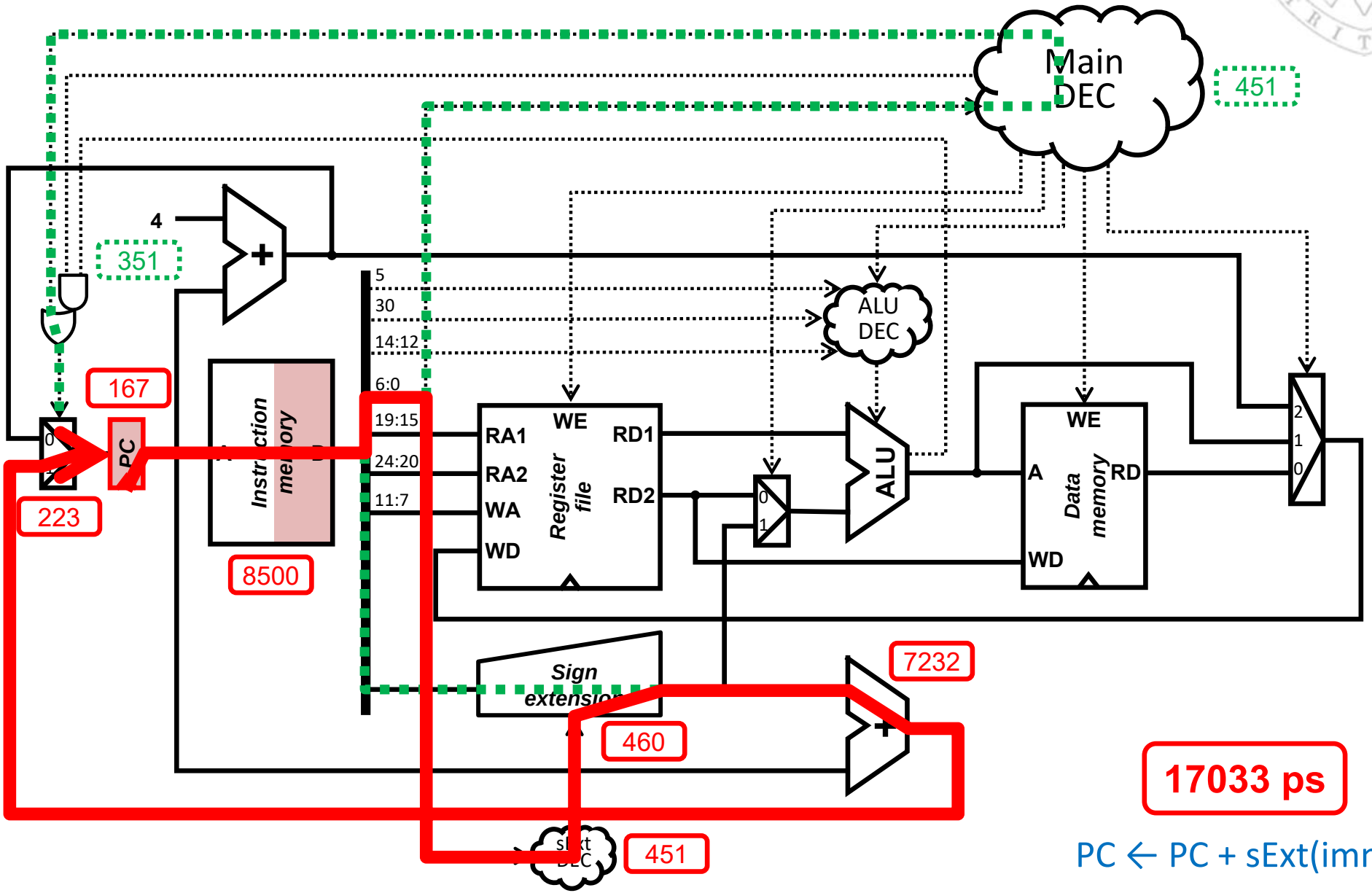


10073 ps

RF[rd] ← PC+4

Cycle time calculation

`jal` instruction (PC update): critical path



17033 ps

PC ← PC + sExt(imm)

About *Creative Commons*

- CC license (Creative Commons)

  - This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms:

Attribution:
Credit must be given to the creator.

Non commercial:
Only noncommercial uses of the work are permitted.

Share alike:
Adaptations must be shared under the same terms.

More information: https://creativecommons.org/licenses/by-nc-sa/4.0/