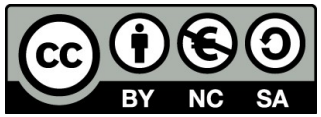
Module 6:

Multicycle processor design

Introduction to computers II

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



Outline

- ✓ Introduction.
- ✓ Data path design.
- ✓ Controller design.
- ✓ Comparison: single-cycle vs. multicycle.
- ✓ Performance metrics.

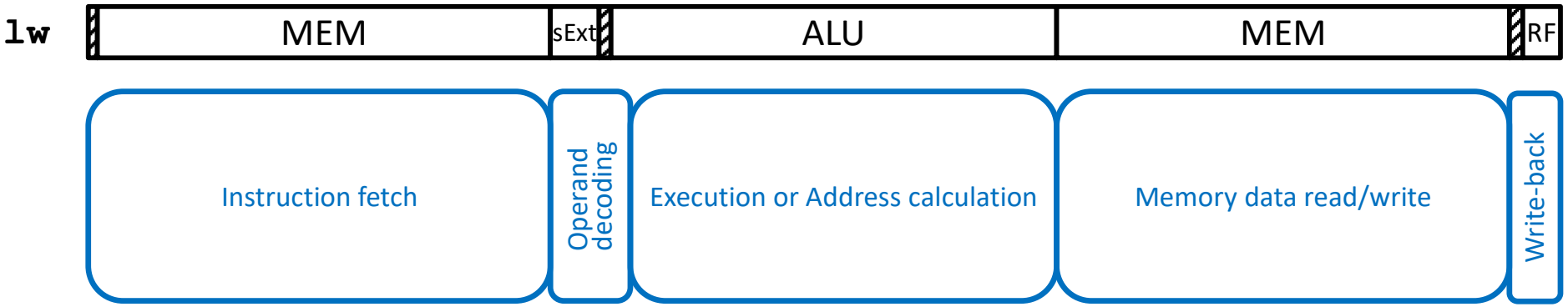
- ✓ Technology.

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

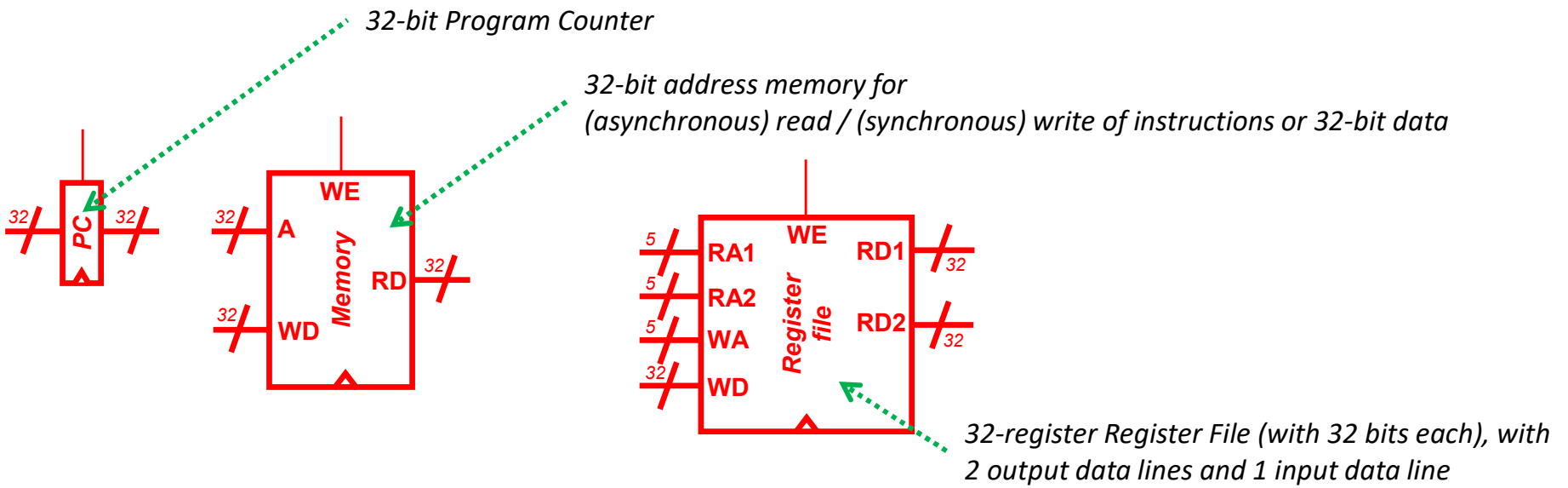
- The multicycle processor solves the problem of the single-cycle processor by splitting the instruction execution into several stages:
  - The cycle time is determined by the slowest stage.
    - It will be much lower than the single-cycle processor cycle time.
    - Each instruction will need different time to execute, since each requires a different number of clock cycles.
    - The execution time of an instruction will be proportional to its complexity.
  - Hardware can be reused, as long as it is used in different cycles.
    - It requires only one ALU and one memory for instructions and data.



Data path design

Architectural storage elements

- The storage elements visible to programmers are the same as in the single-cycle processor.

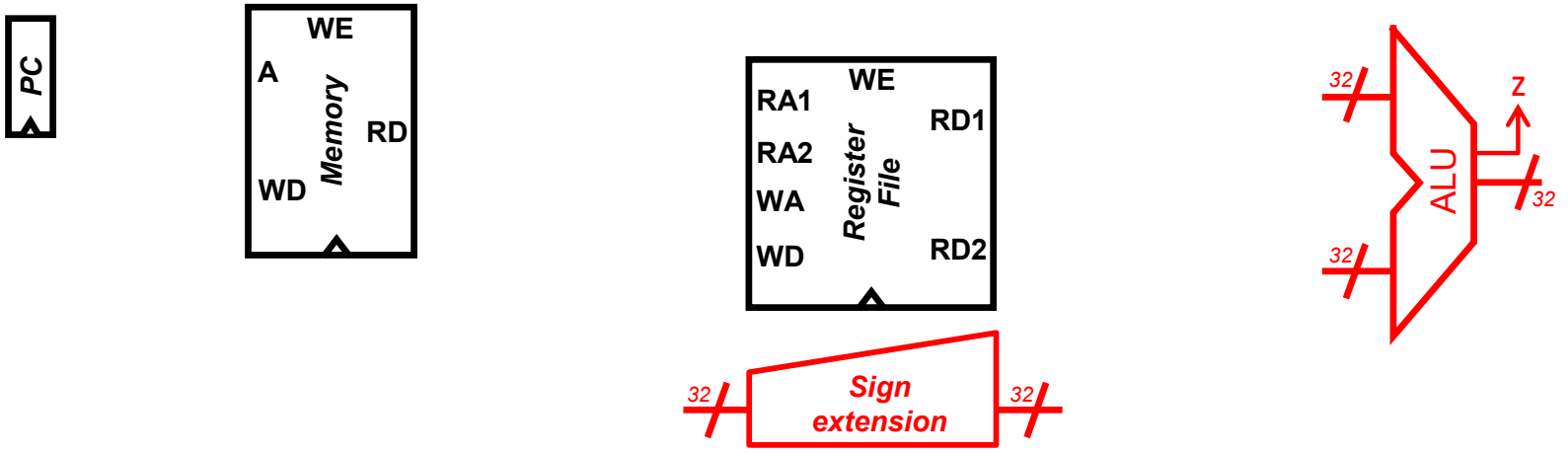


- However, the multicycle processor has only one unified memory.
  - Instructions are read from memory in a clock cycle different from the cycle in which data are read/written, and therefore there is no point in having 2.
- Besides, the PC will be a conventional register with a load signal.
  - In the multicycle processor, the PC is not updated in all the clock cycles.

Data path design

Functional elements

- It also has one ALU and one Sign Extension module, both combinational, identical to the ones in the single-cycle implementation.

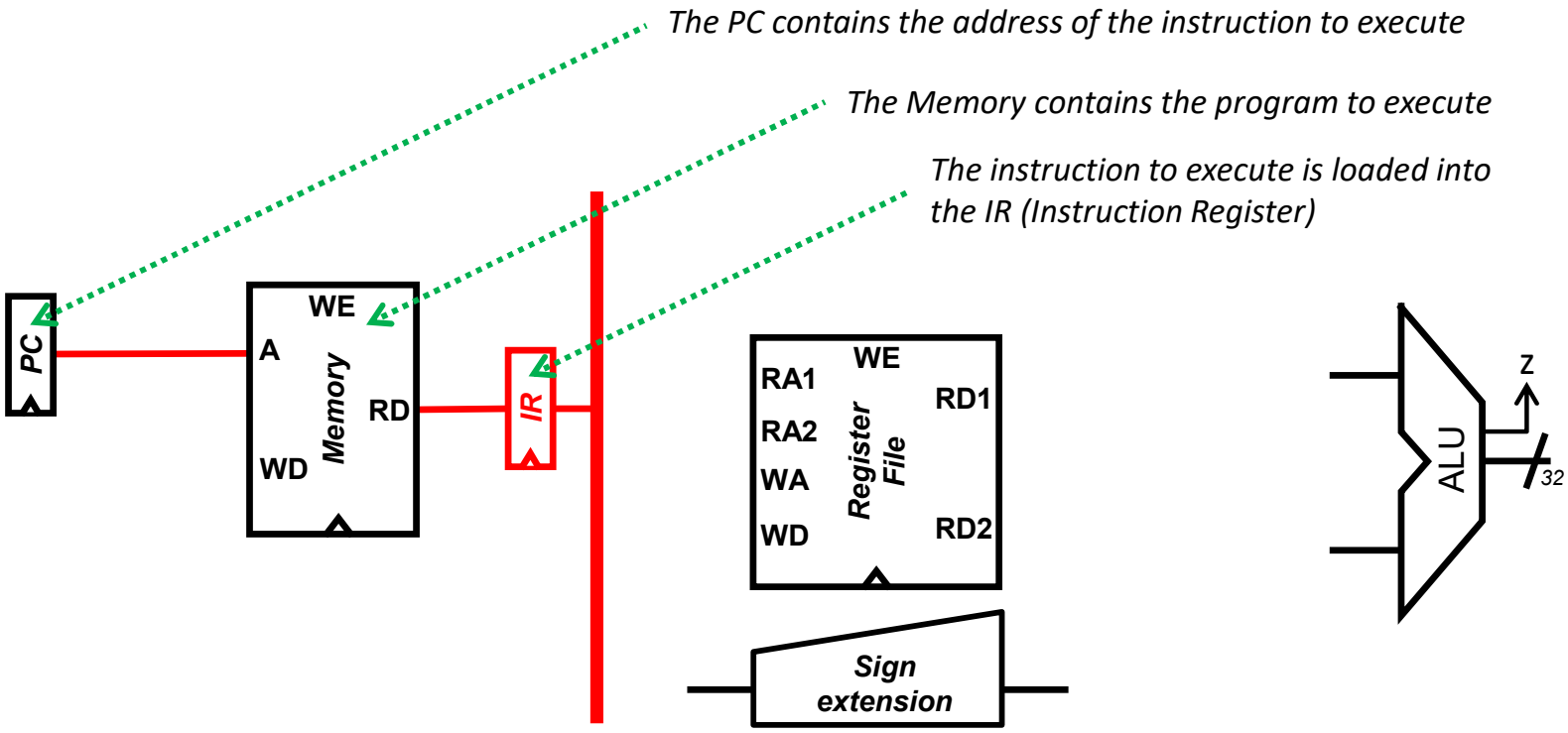


- In the multicycle processor, the ALU will be reused to perform all the required arithmetic operations.
  - There are no additional adders to operate with addresses.

Data path design

Instruction fetch

- The execution of any instruction starts with the instruction fetch.

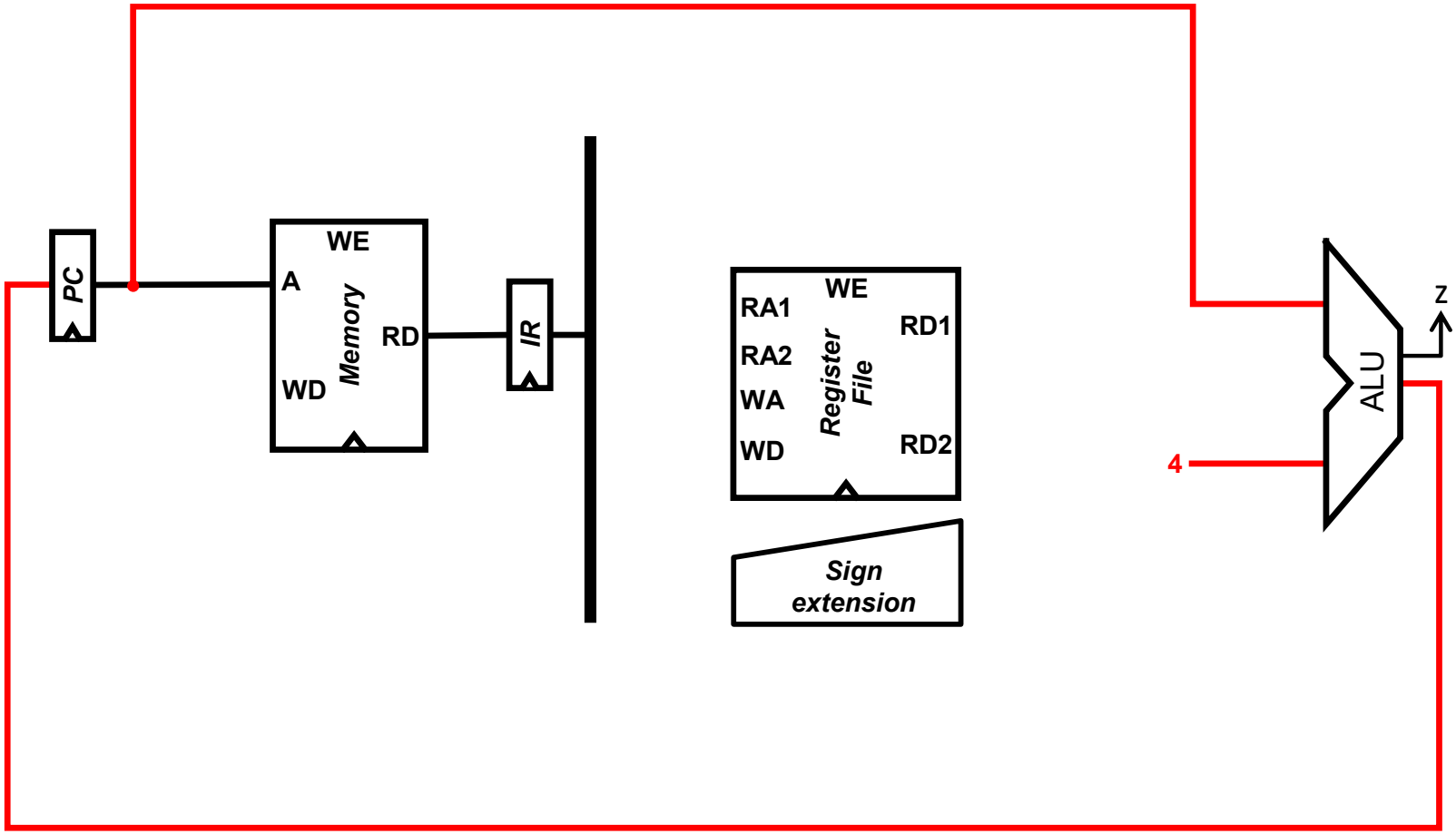


- The instruction fetched from Memory is loaded into the IR auxiliary register, which will store it during all the cycles of the execution.

Data path design

PC increment

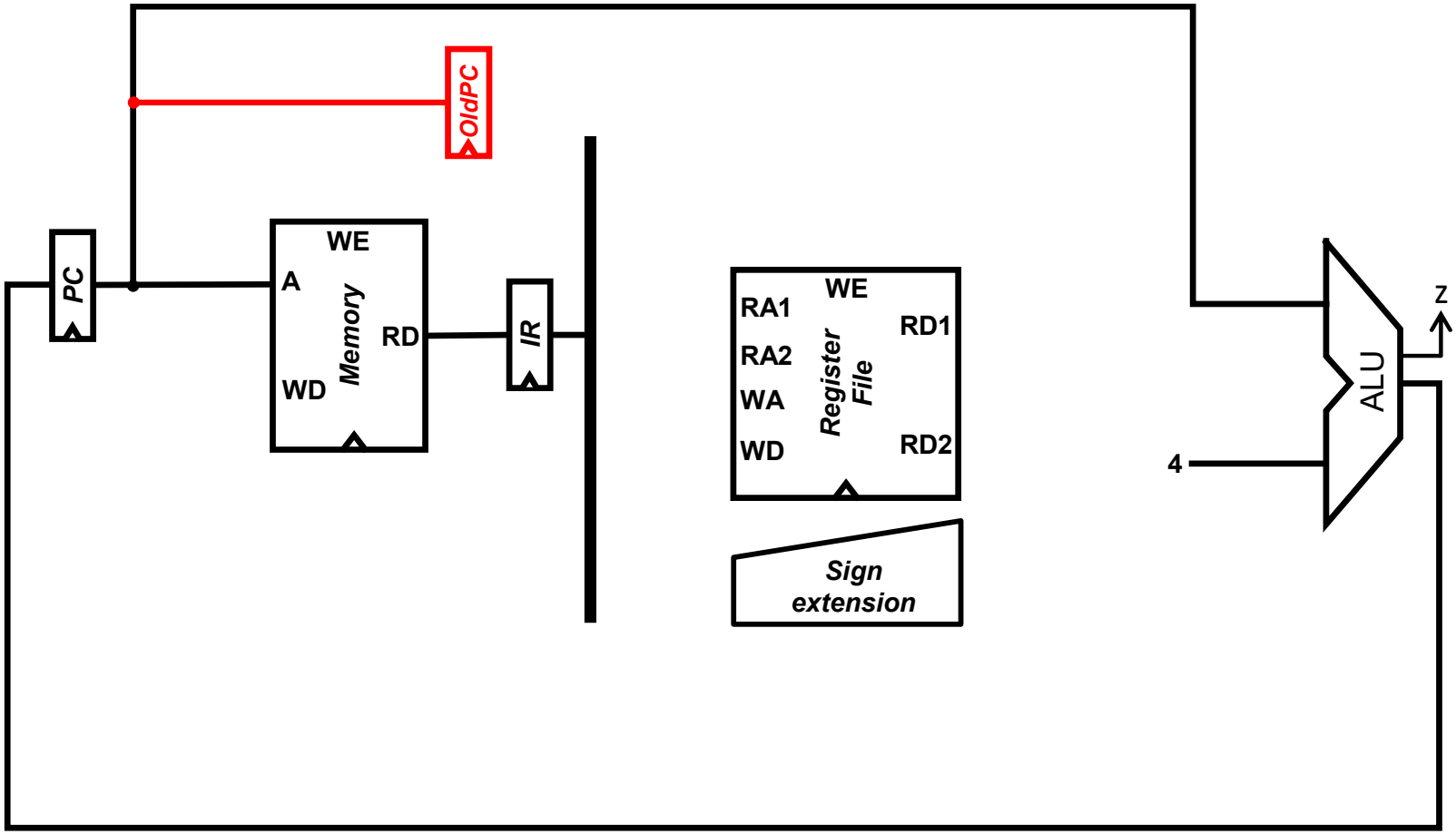
- At the same time as the instruction is loaded into IR, the PC is updated with the address of the following instruction, by adding +4 in the ALU



Data path design

Saving the current instruction address

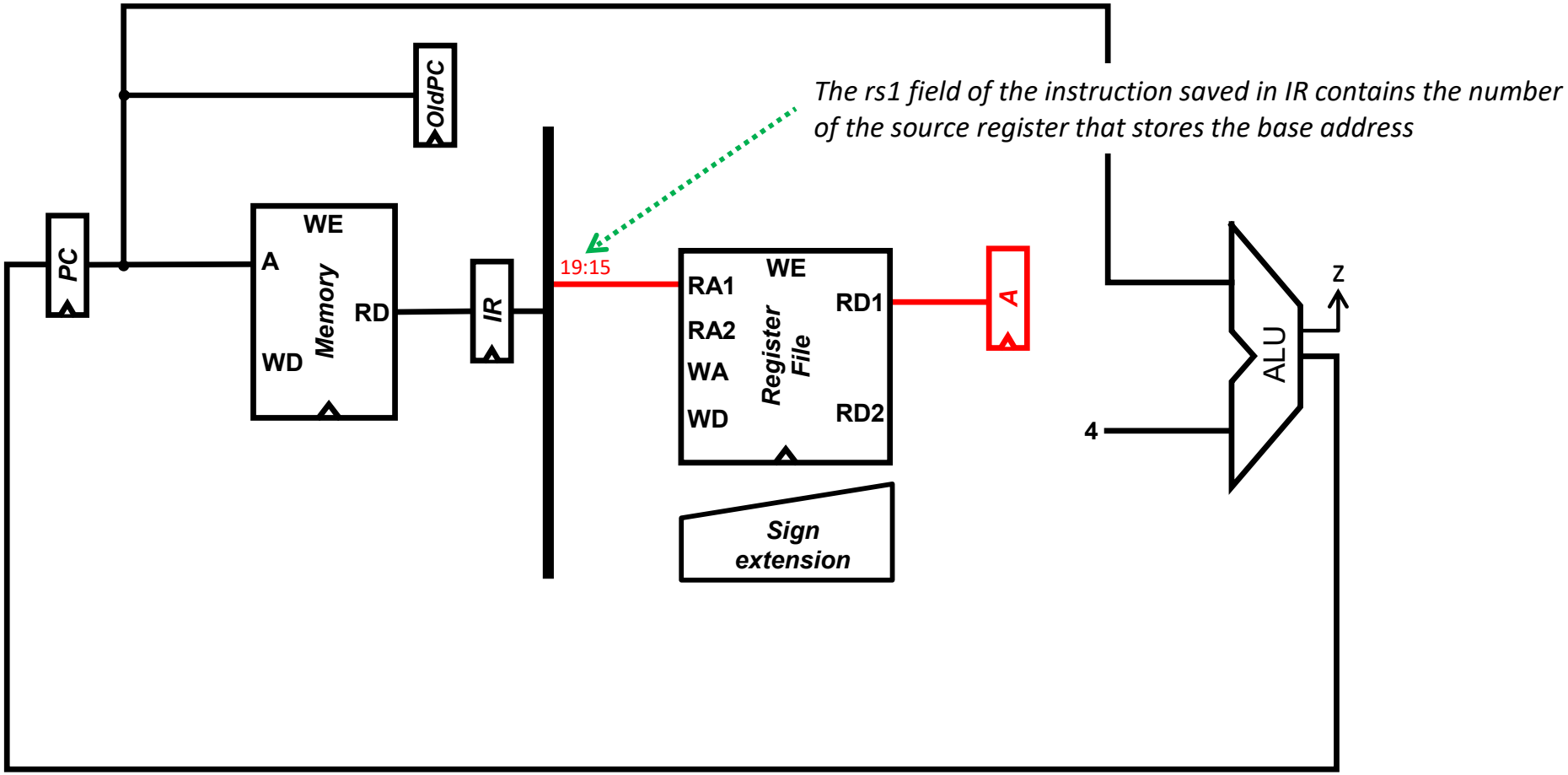
- The PC without the increment is stored in the oldPC auxiliary register, to be used in the PC-relative branch address calculation (**`beq`** or **`jal`** instructions).



Data path design

`lw` instruction: reading the base register

- The A auxiliary register is added to store the base address contained in the Register File rs1 register.

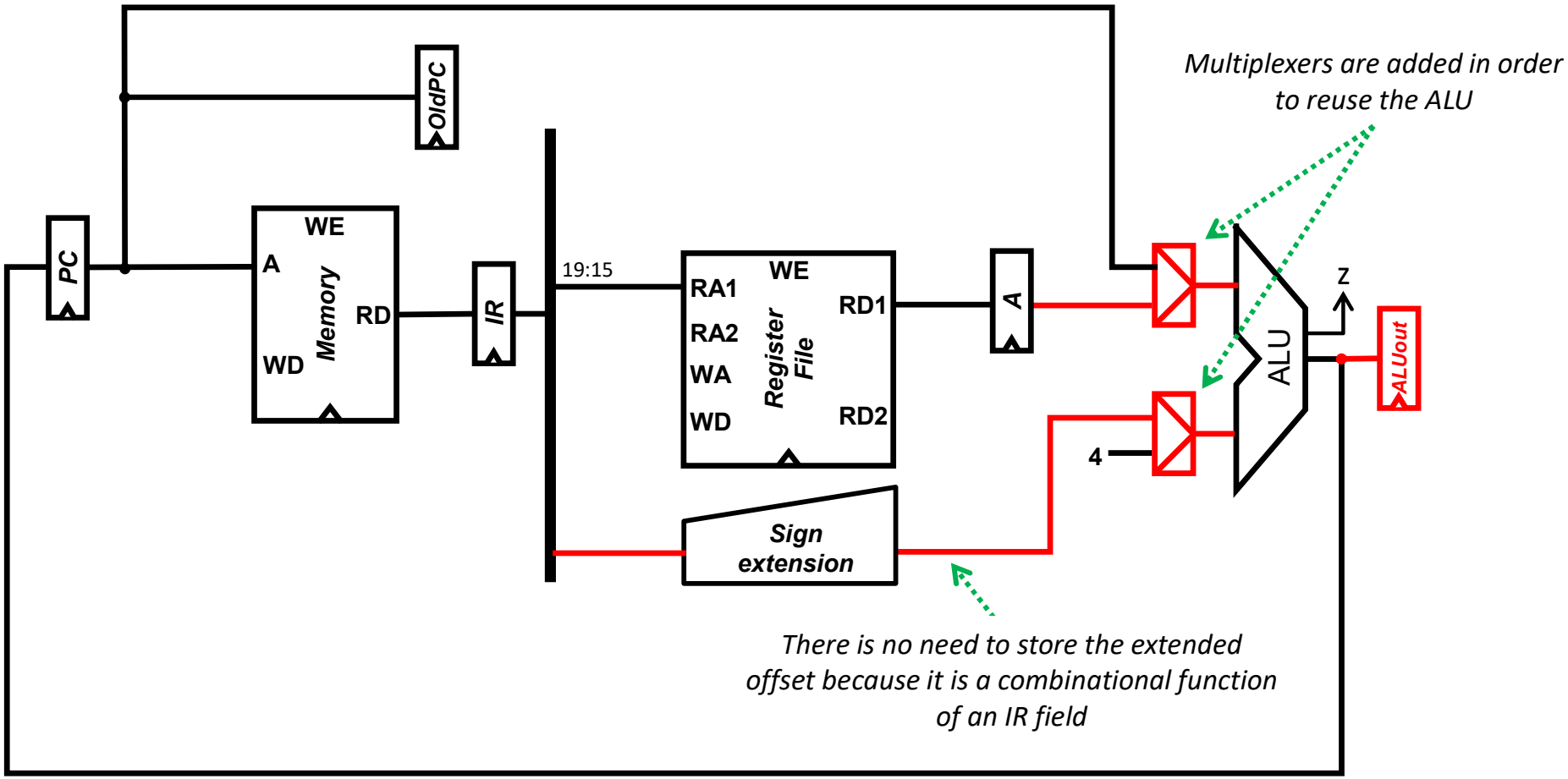


RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4

Data path design

`lw` instruction: calculating the effective address

- The offset sign is extended, it is added in the ALU with the address base stored in A and it is loaded in the ALUout auxiliary register.

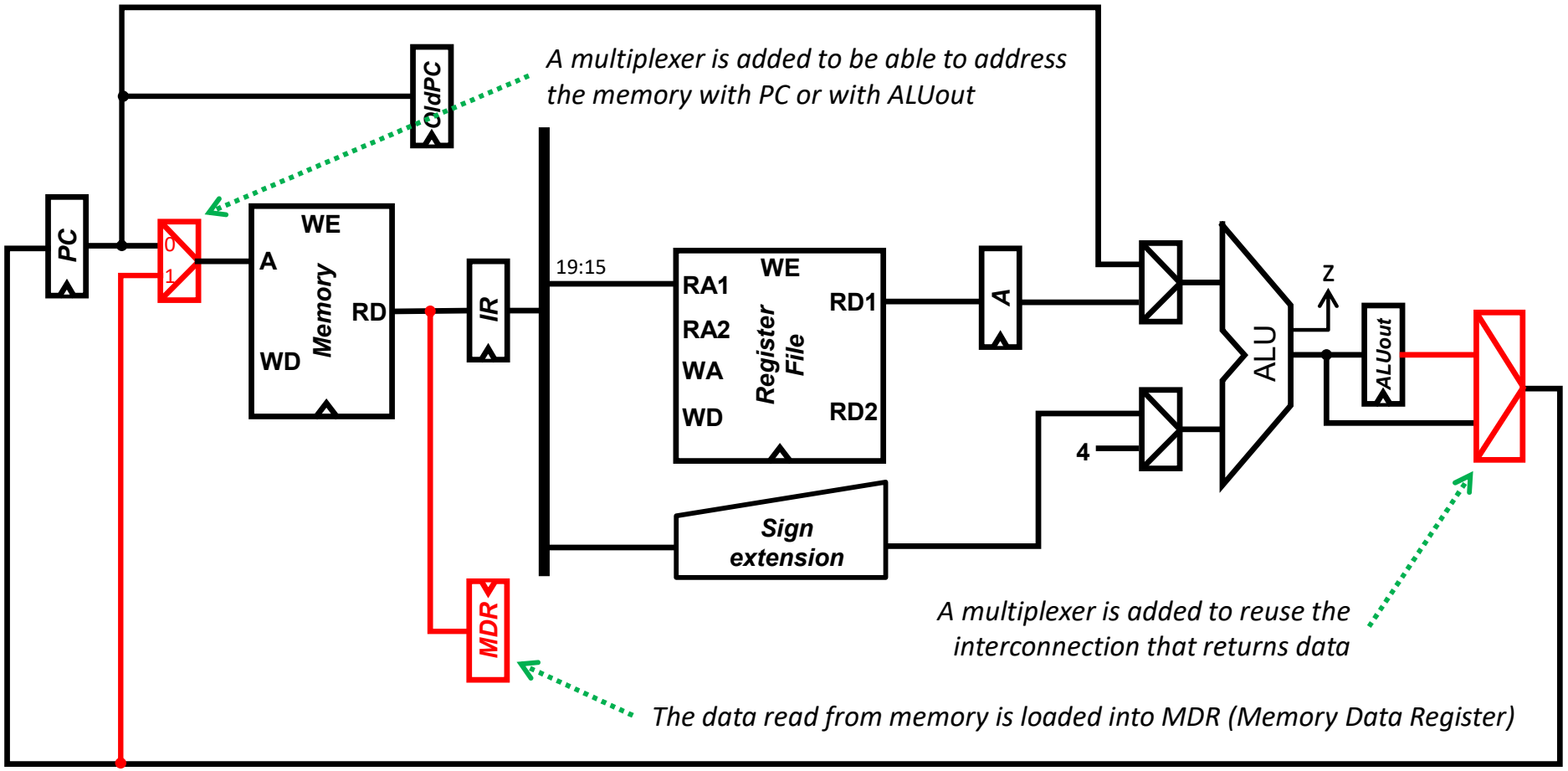


RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4

Data path design

`lw` instruction: reading the operand

- The memory is addressed with the effective address stored in ALUout and the read data is saved in the MDR auxiliary register.

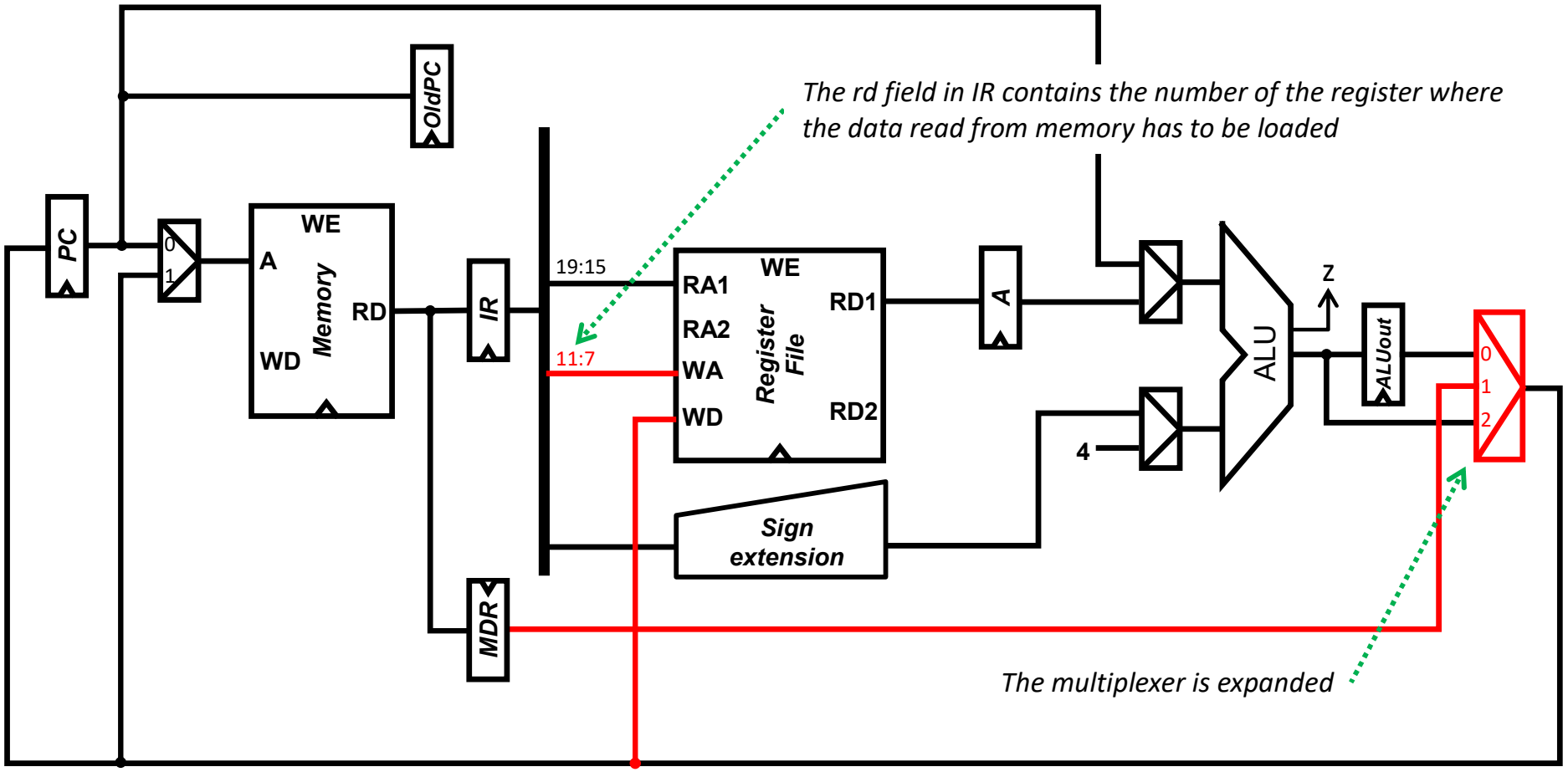


RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4

Data path design

`lw` instruction: saving the operand

- The data stored in MDR (previously read from Memory) is saved in the Register File.

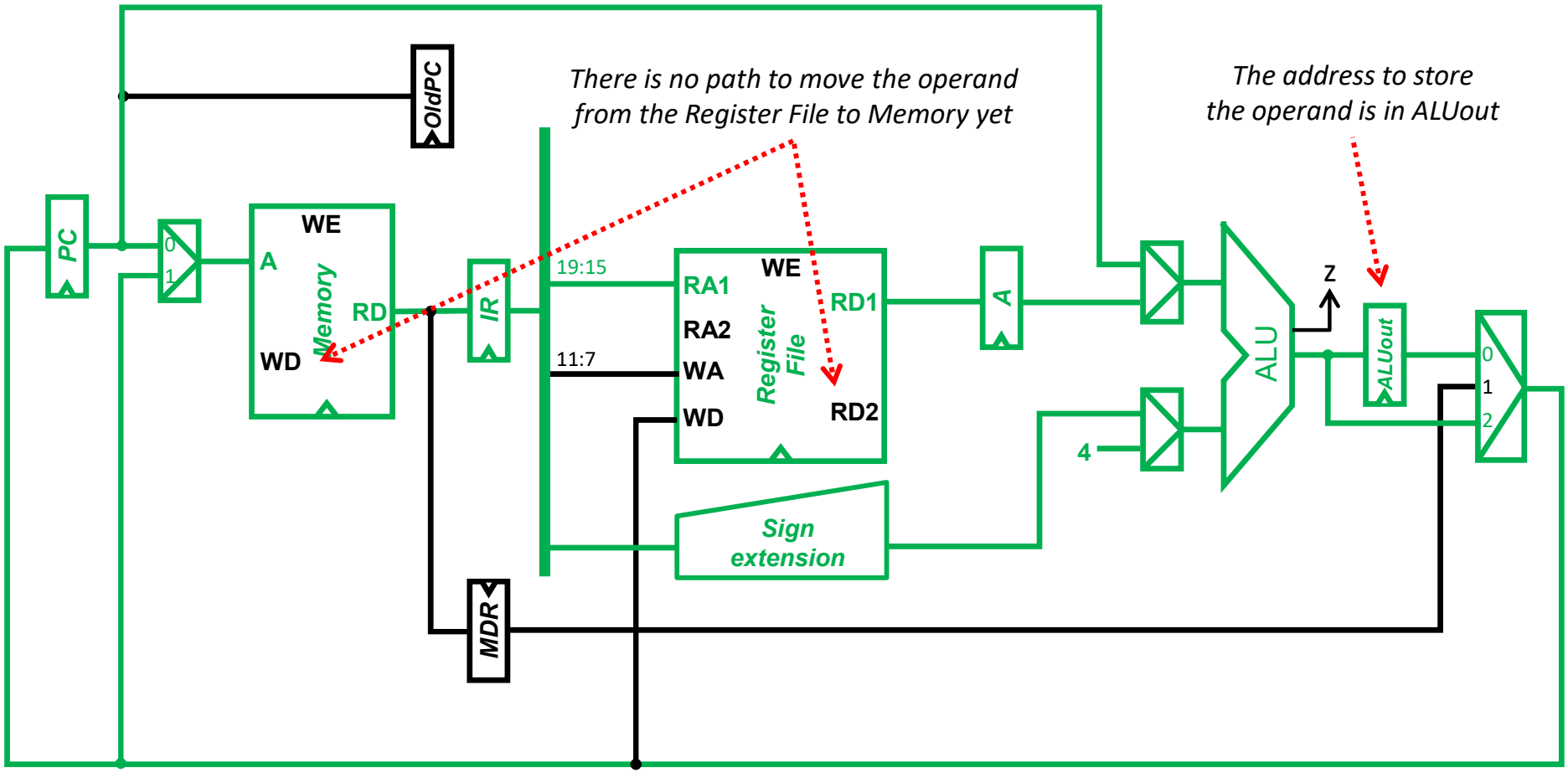


RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4

Data path design

Data path for `sw` instructions

- This data path can be reused to perform all the actions of the `sw` instruction except for writing the operand in memory.

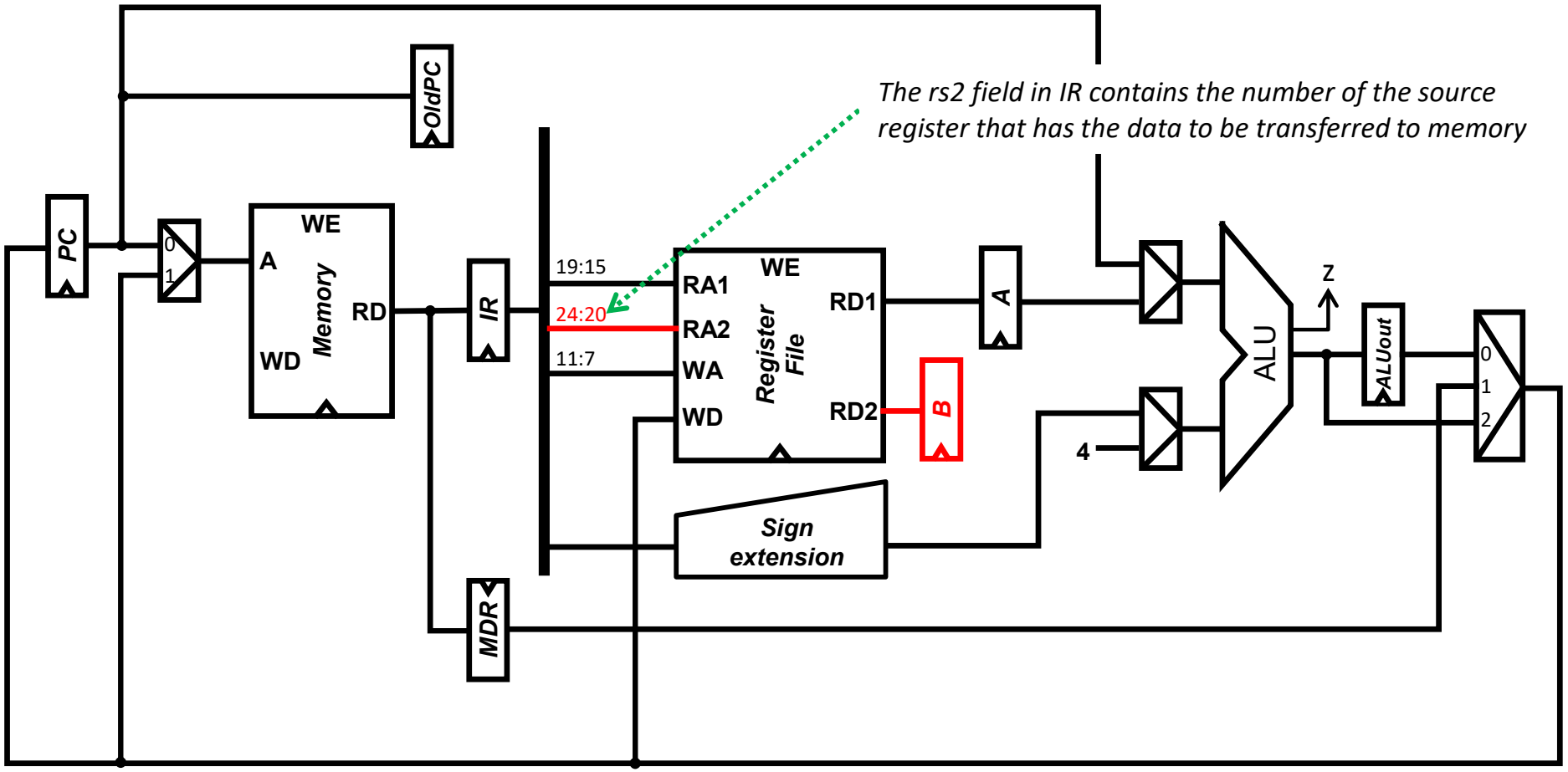


Mem[RF[rs1] + sExt(imm)] ← RF[rs2], PC ← PC+4

Data path design

`sw` instruction: reading the operand

- The B auxiliary register is added to store the data contained in the Register File rs2 register.

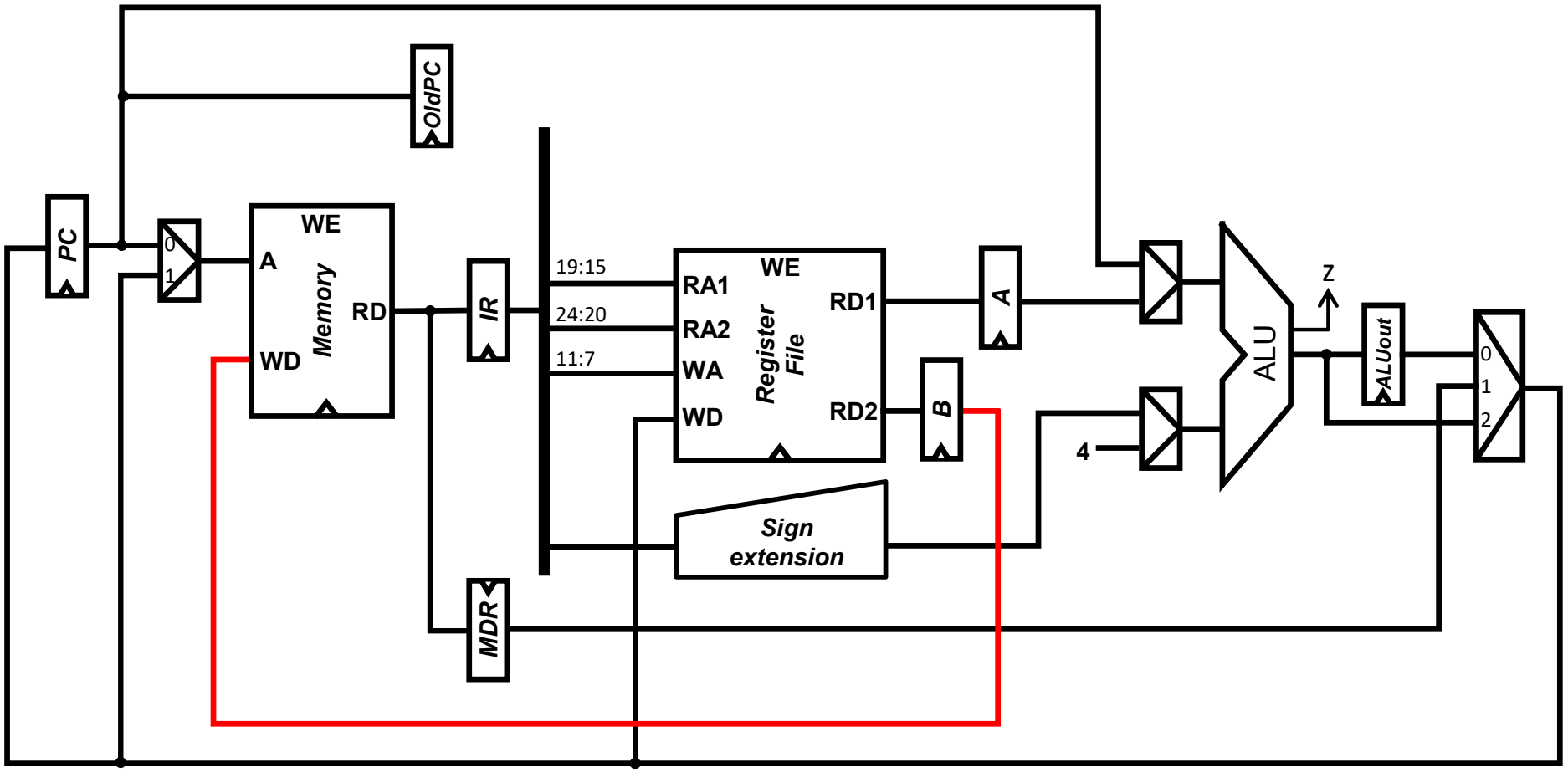


Mem[RF[rs1] + sExt(imm)] ← RF[rs2], PC ← PC+4

Data path design

`sw` instruction: storing the operand

- The data stored in B (previously read from the Register File) is stored in the Memory.

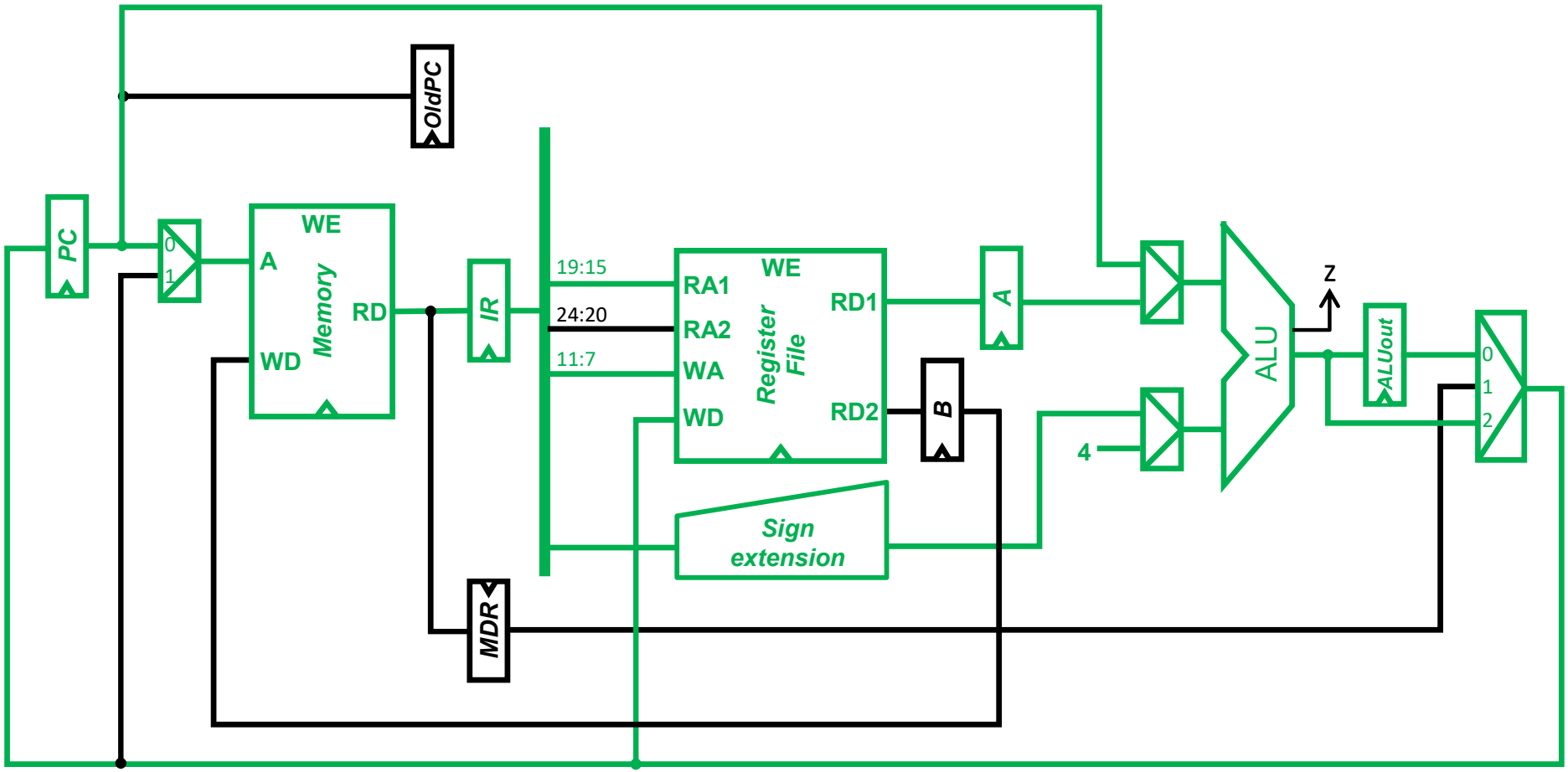


Mem[RF[rs1] + sExt(imm)] ← RF[rs2], PC ← PC+4

Data path design

Data path for **`addi`**-like instructions

- This data path can be reused to perform all the actions of the arithmetic-logic instructions (I-type)

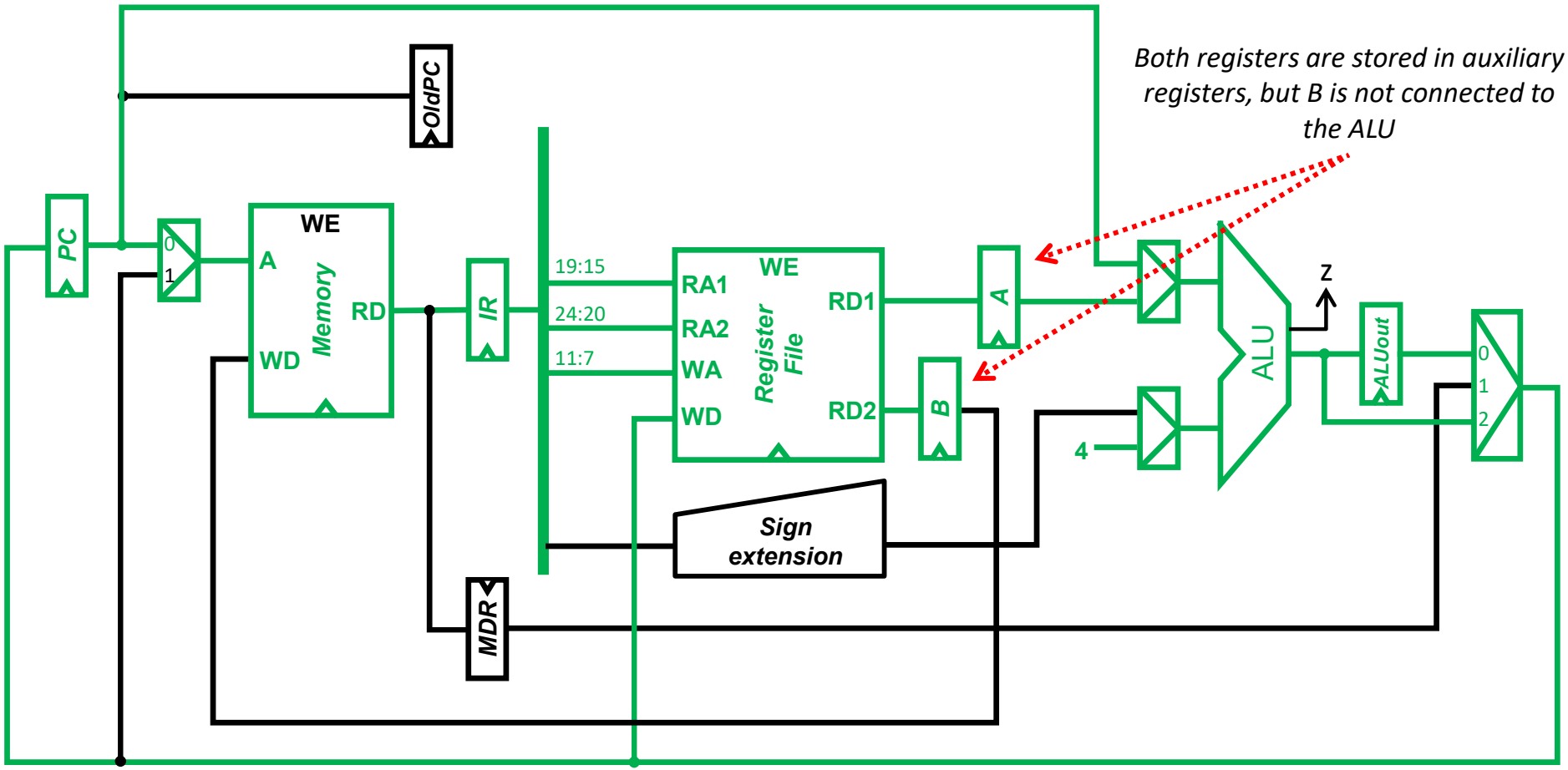


RF[rd] ← RF[rs1] *op* sExt(imm), PC ← PC+4

Data path design

Data path for **add**-like instructions

- This data path can be reused to perform all the actions of the arithmetic-logic instructions (R-type) except for the operation itself.

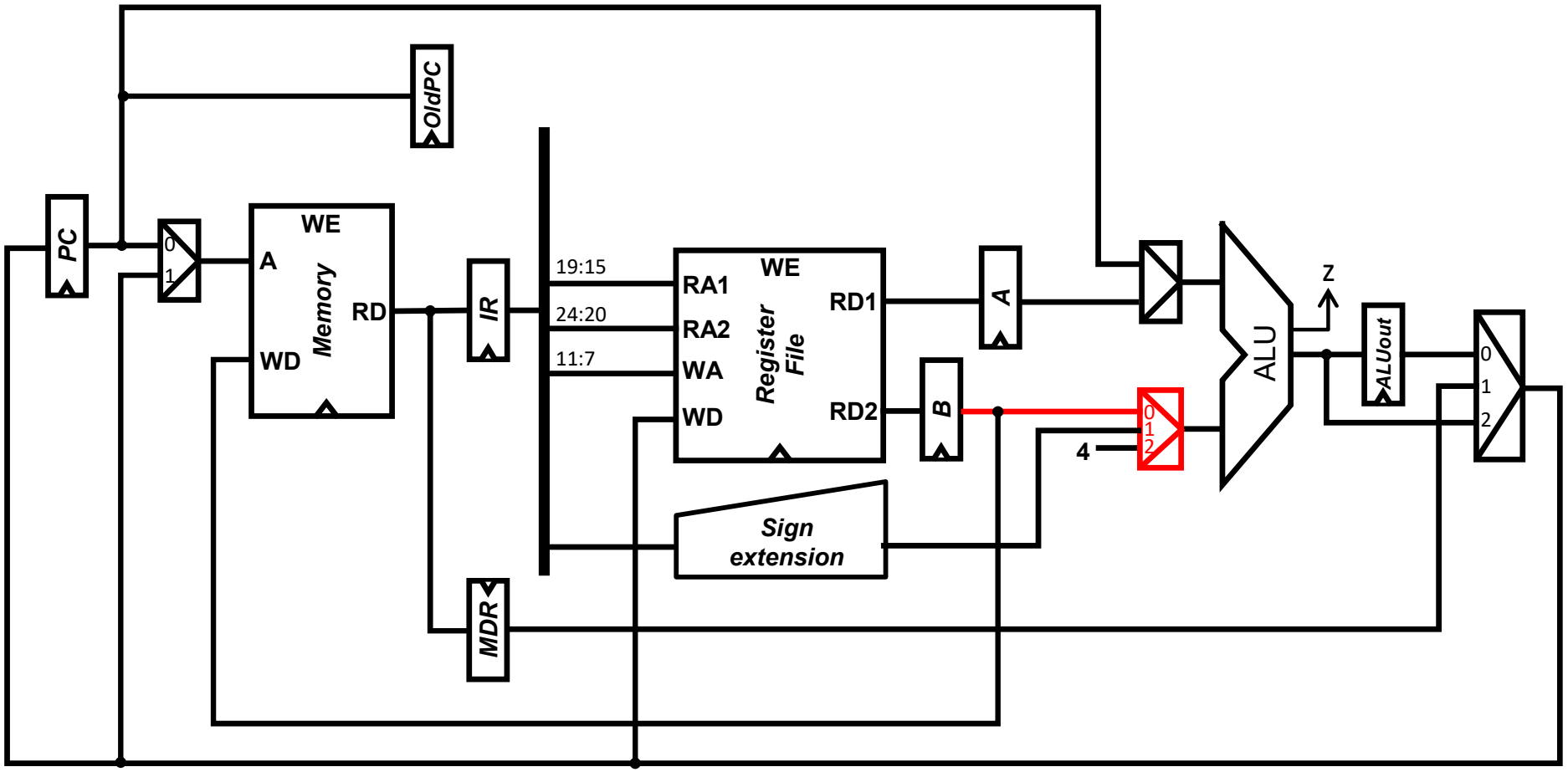


RF[rd] ← RF[rs1] *op* RF[rs2], PC ← PC+4

Data path design

add-like instructions: calculating the operation

- The multiplexer is expanded to reuse the ALU, so that it can perform the arithmetic-logic operations with both operands in the Register File.

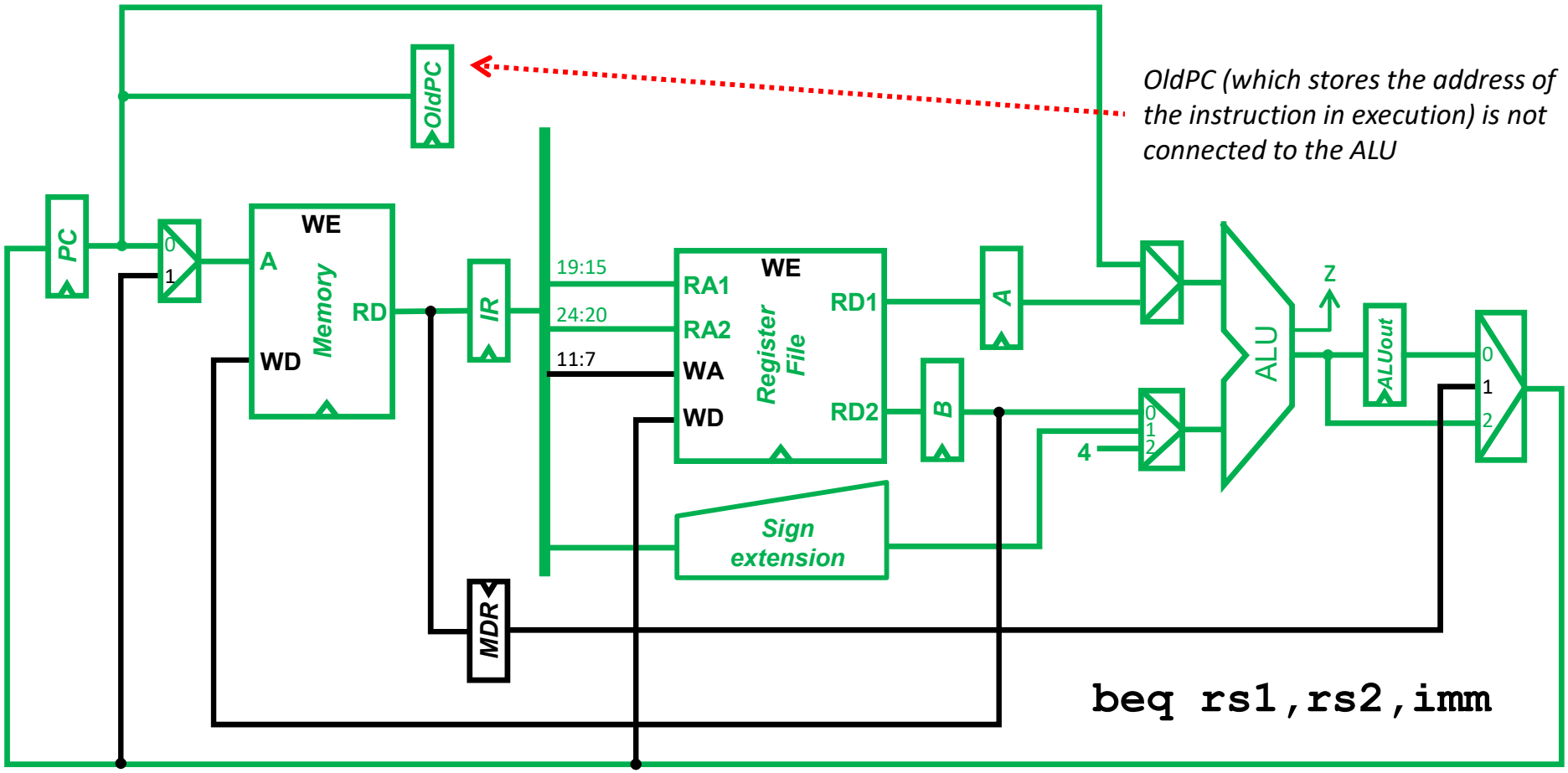


RF[rd] ← RF[rs1] *op* RF[rs2], PC ← PC+4

Data path design

Data path for **beq** instructions

- This data path can be reused to perform all the actions of the **beq** instruction except for the branch address calculation.

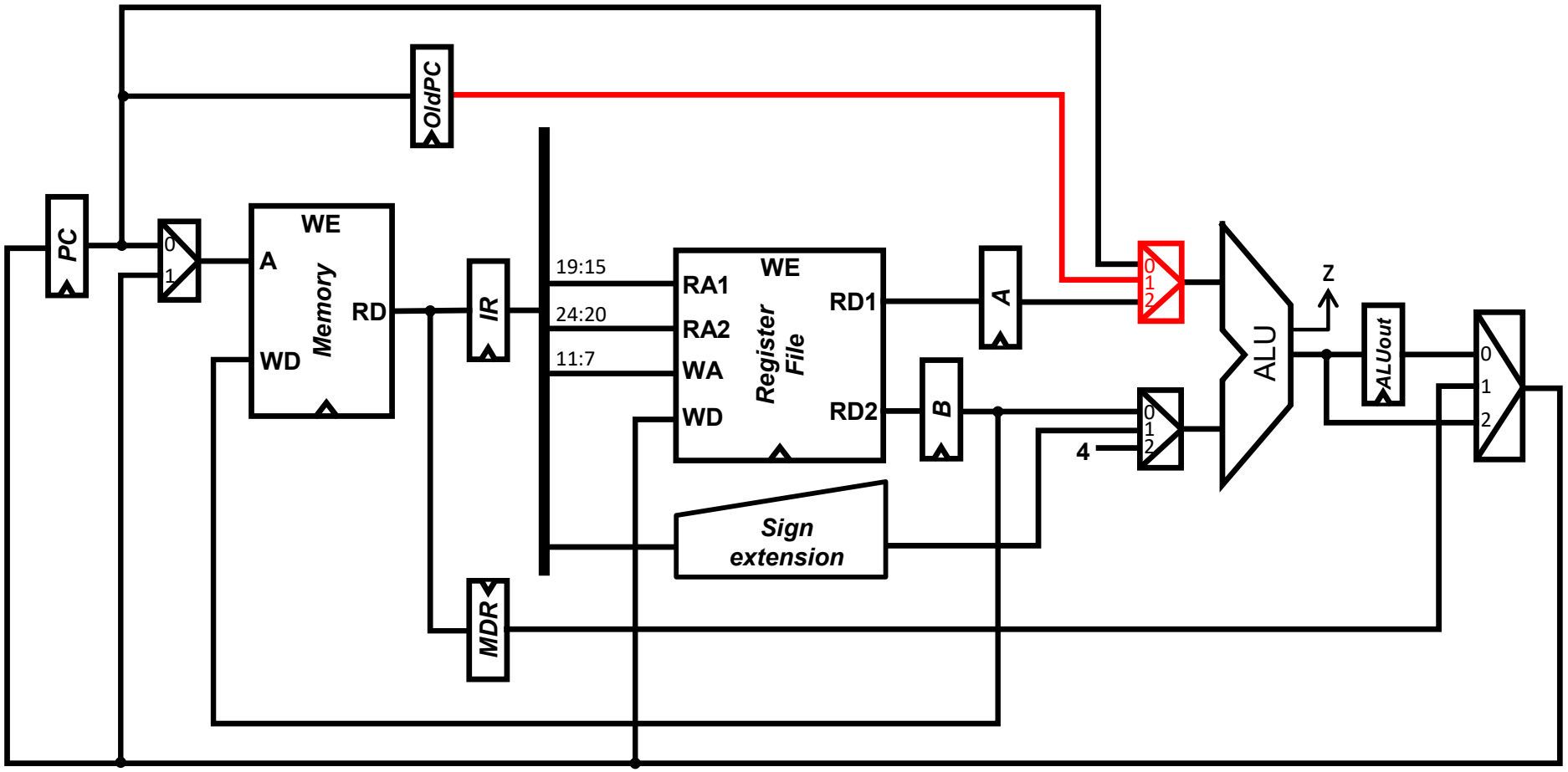


PC ← *if* (RF[rs1] = RF[rs2]) *then* (PC + sExt(imm)) *else* (PC+4)

Data path design

`beq` instruction: calculating the branch address

- The multiplexer is expanded to reuse the ALU, so that it can perform branch address calculation.

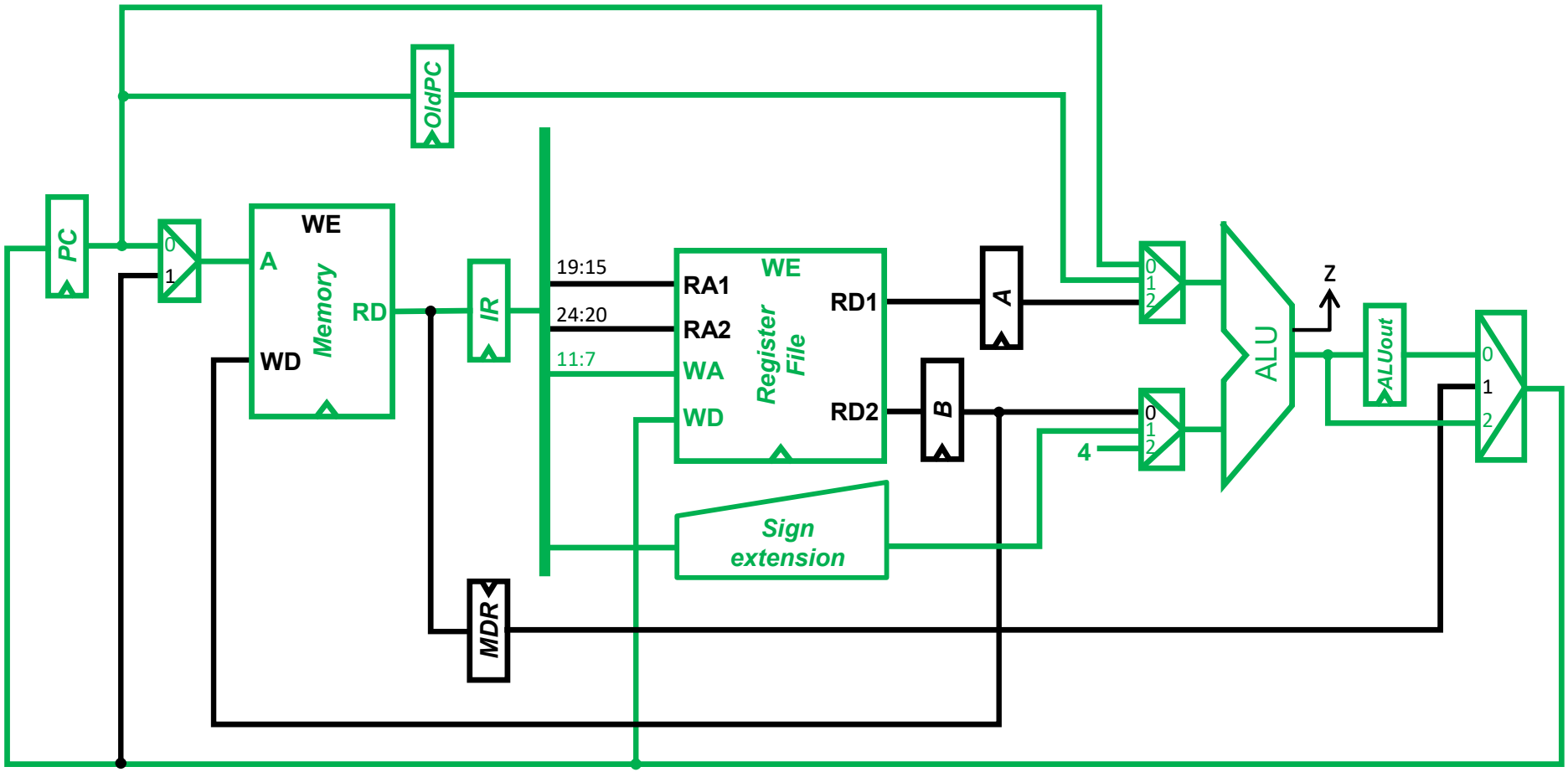


PC ← *if* (RF[rs1] = RF[rs2]) *then* (PC + sExt(imm)) *else* (PC+4)

Data path design

Data path for `jal` instructions

- This data path can be reused to perform all the actions of the **`jal`** instruction

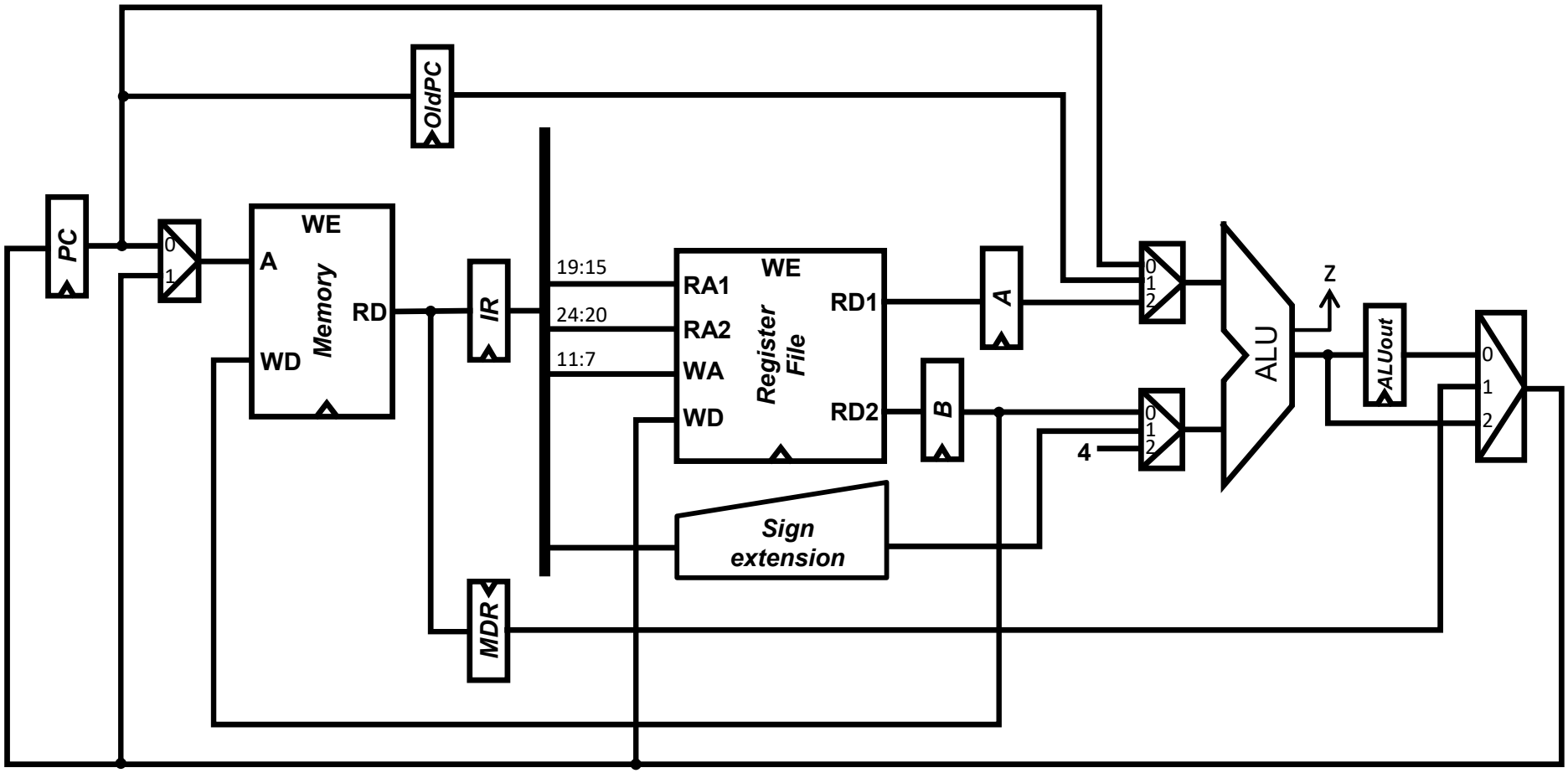


PC ← PC + sExt(imm), RF[rd] ← PC+4

Data path design

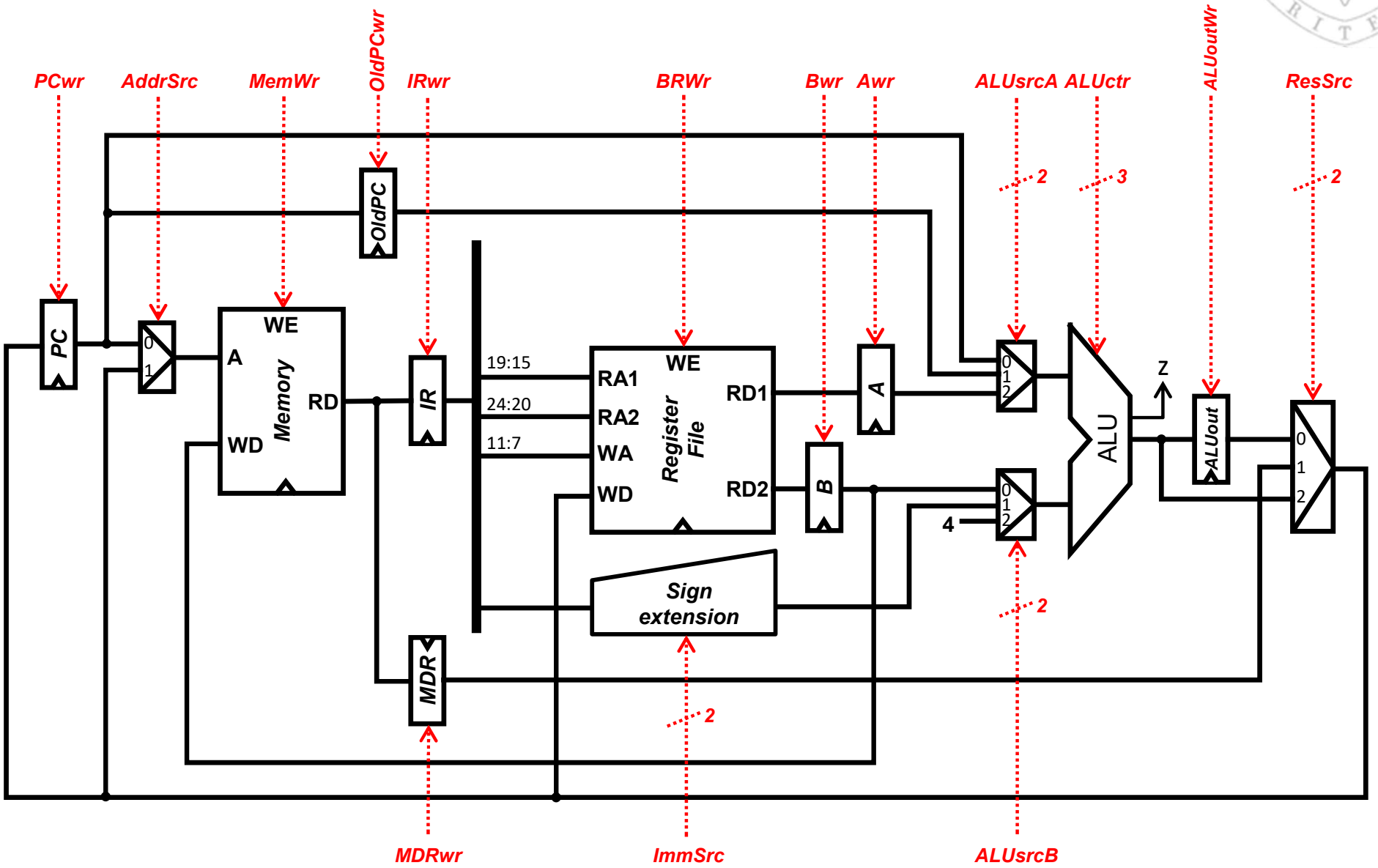
Reduced RISC-V data path

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



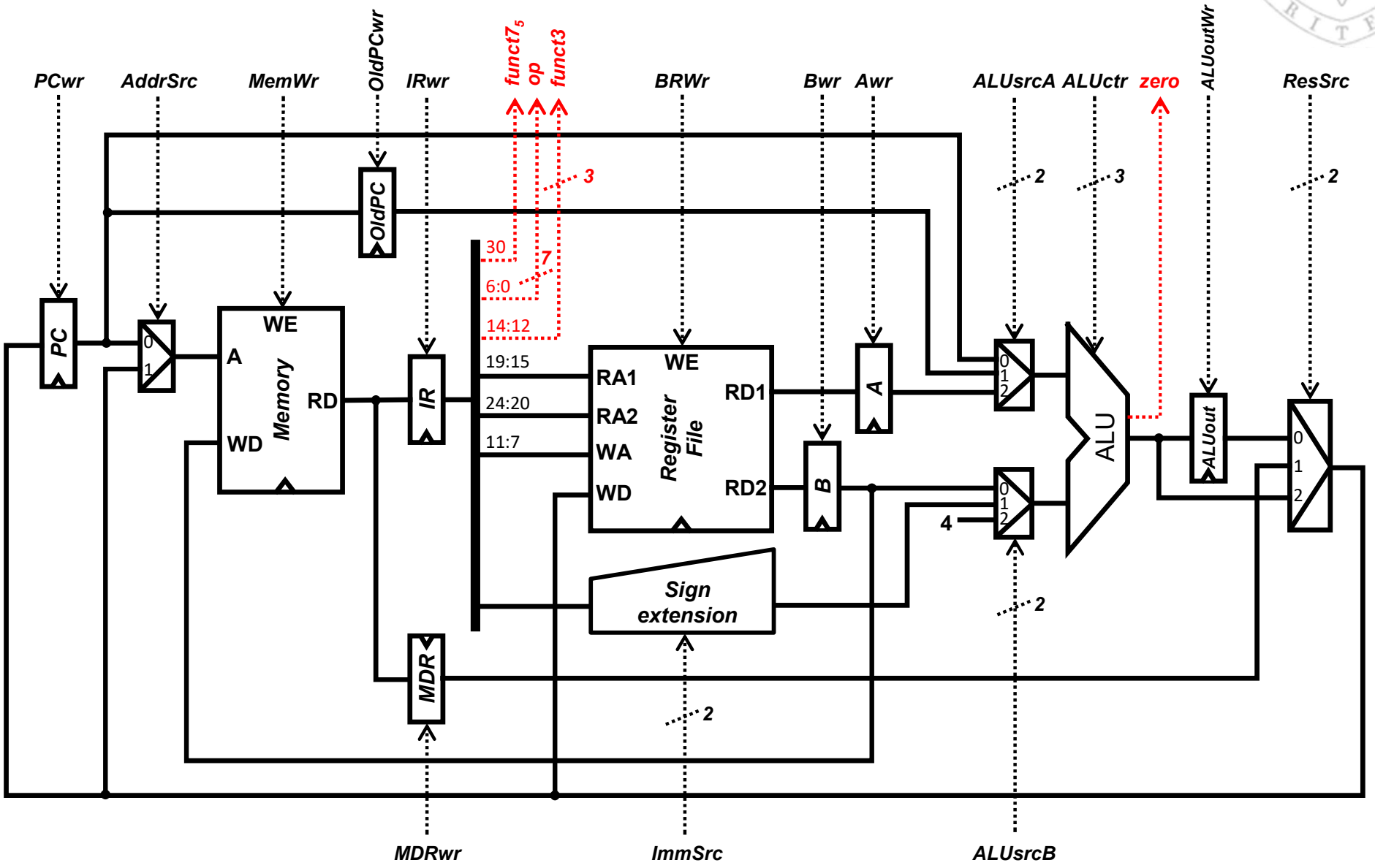
Data path design

Control signals



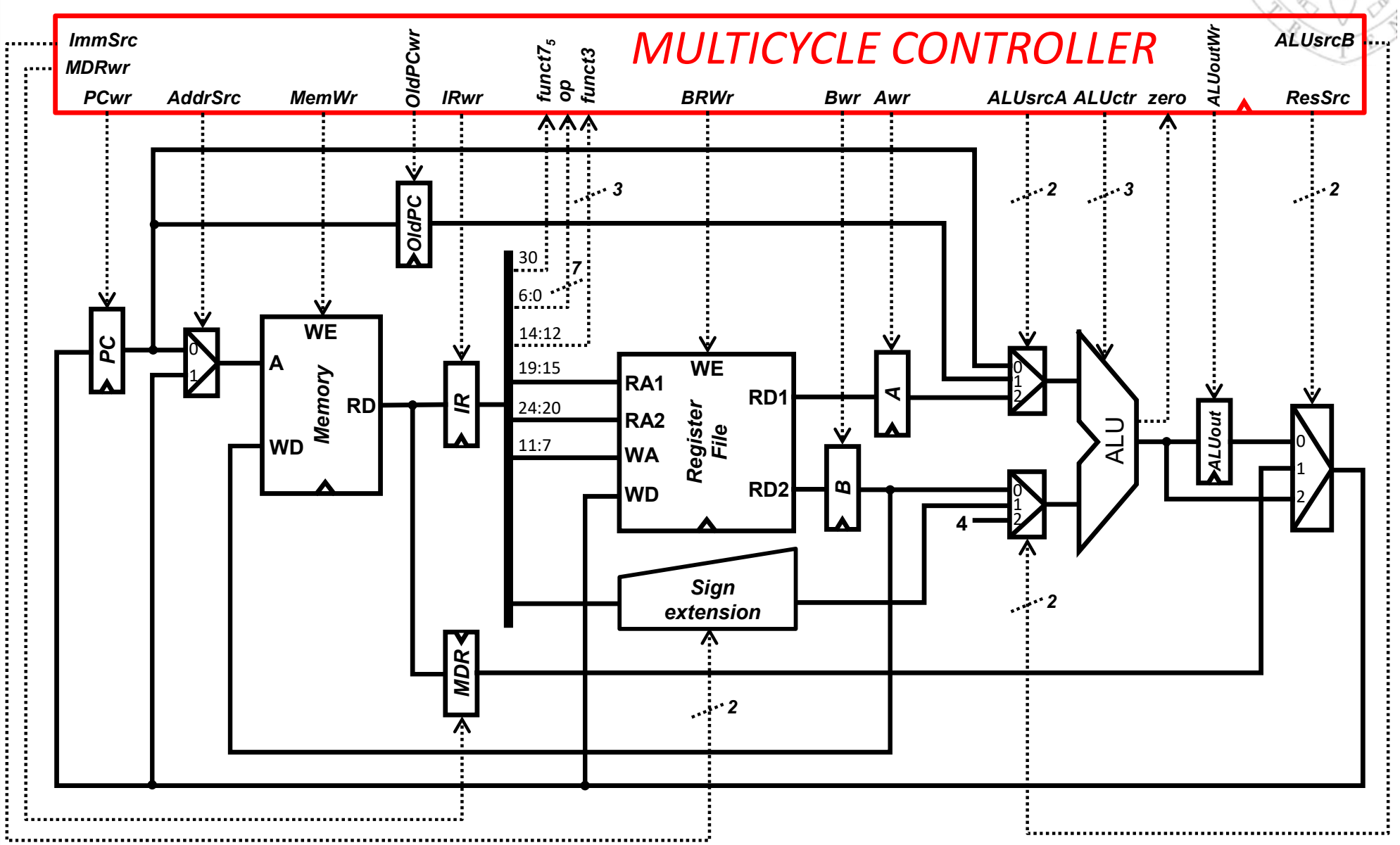
Data path design

Status signals



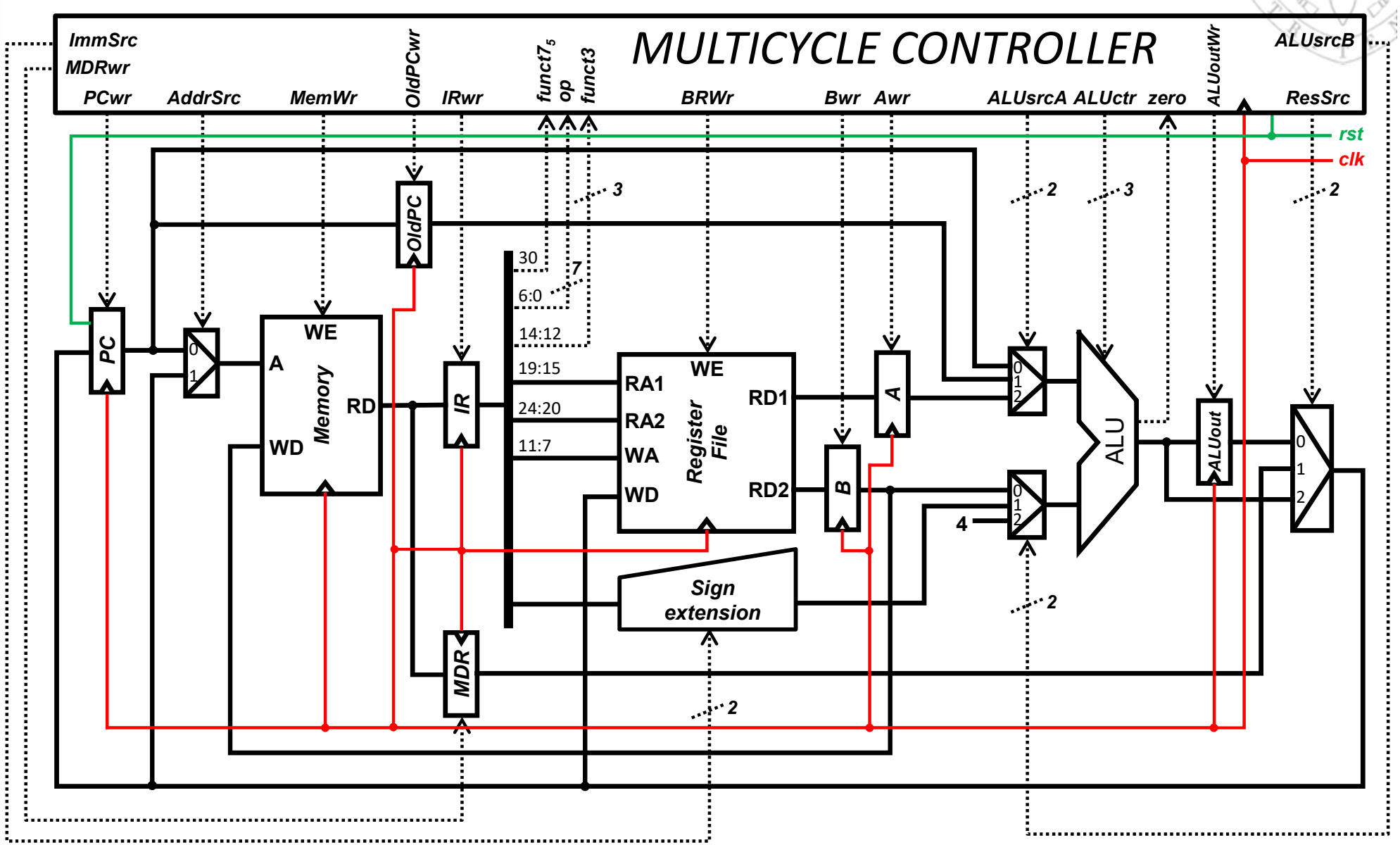
Data path design

Connection with the controller



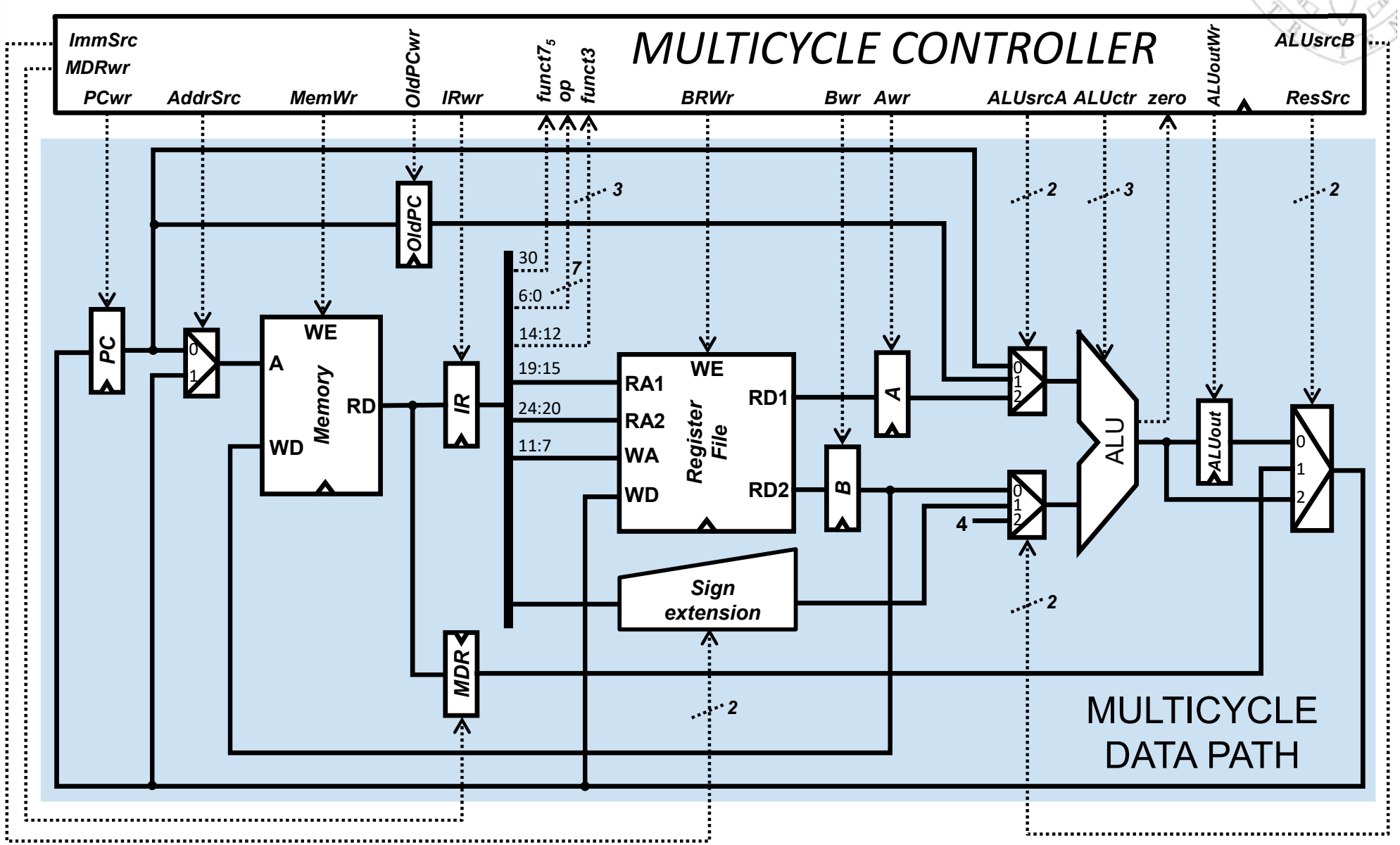
Data path design

Connection with the clock and reset



Data path design

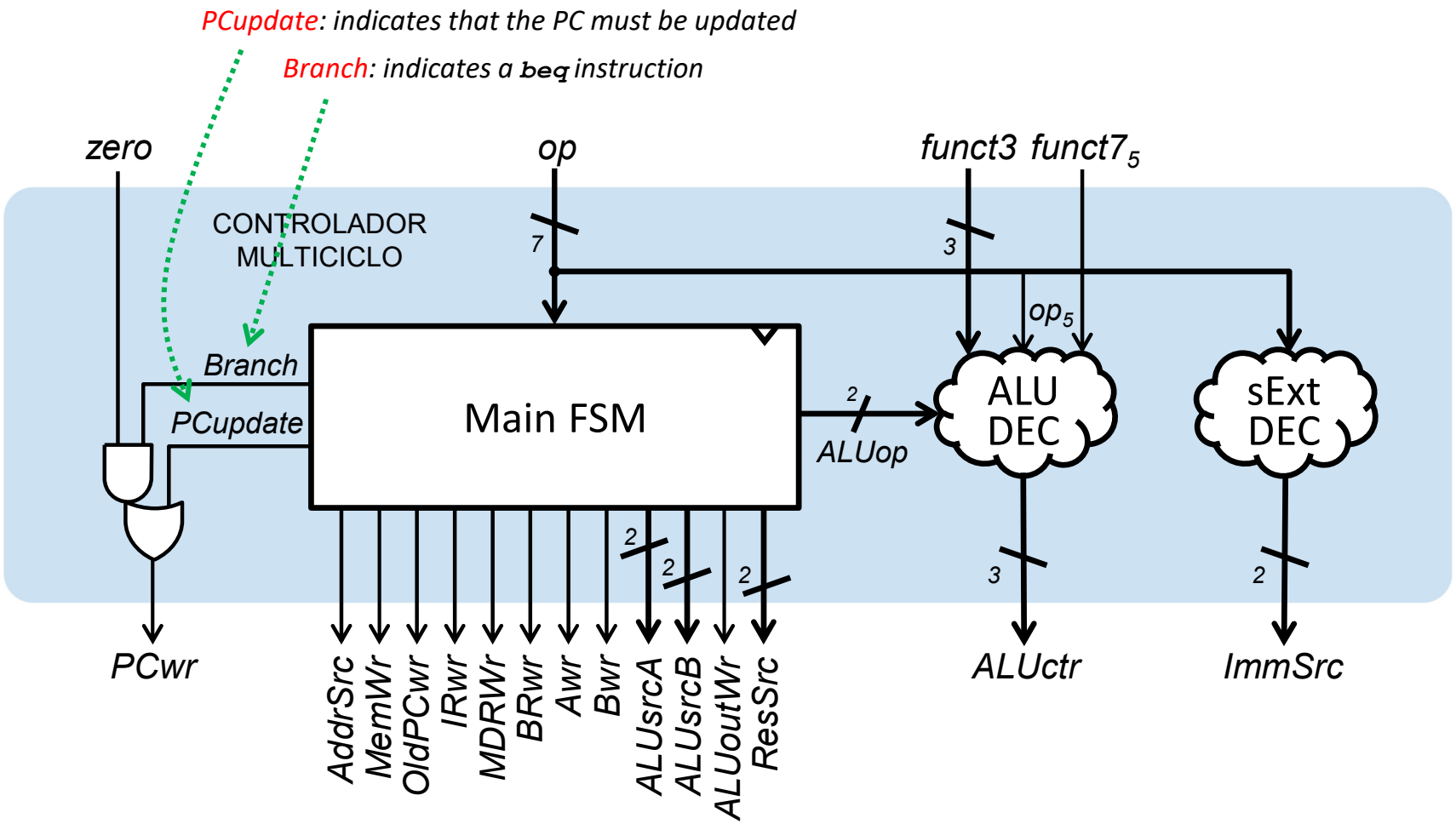
Full system



Controller design

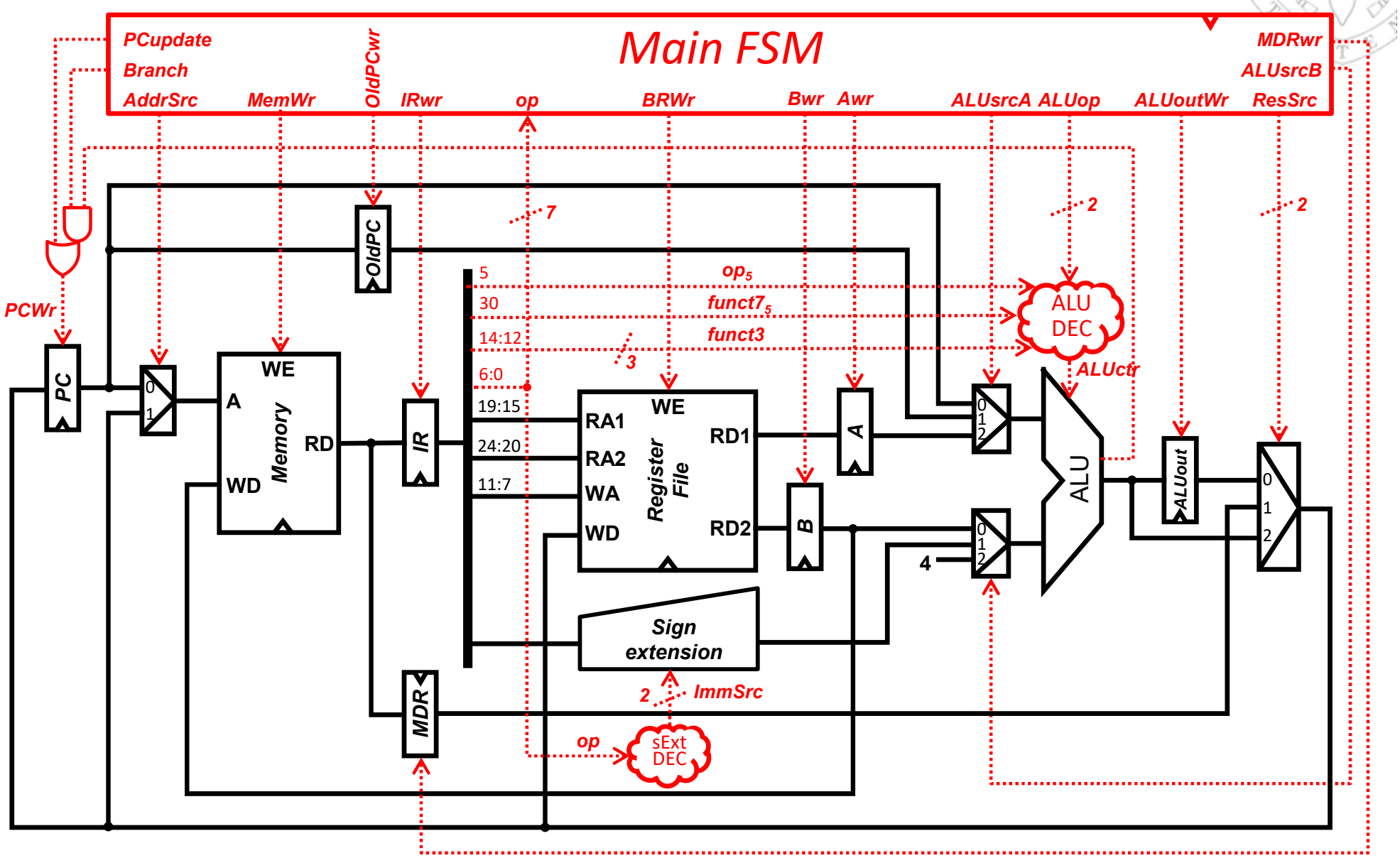
Controller structure (i)

- In the multicycle processor, the controller is a sequential circuit:
  - With a structure of 4 subcircuits, similar to the single-cycle controller



Controller design

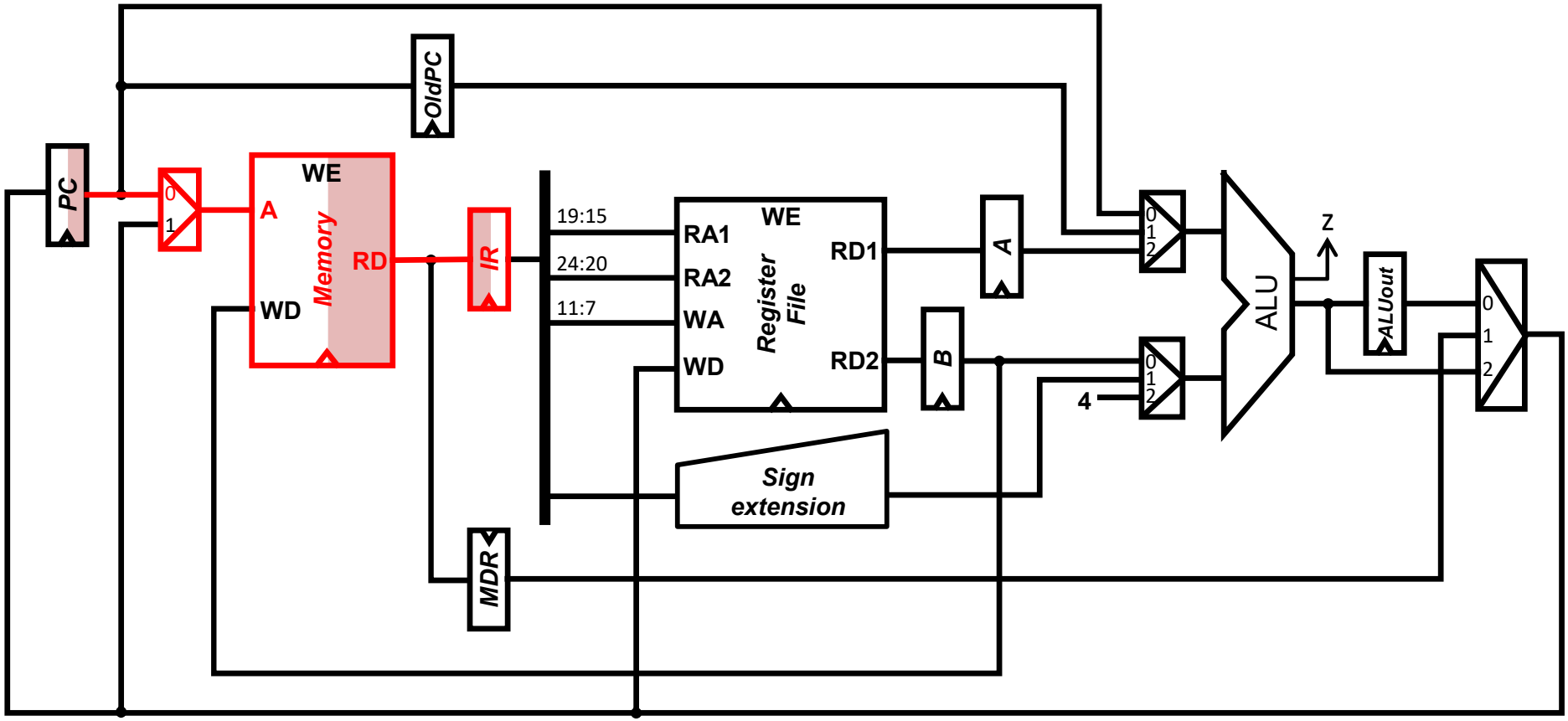
Controller structure (ii)



Controller design

`lw` instruction: register transfers

1. IR ← Mem[PC]

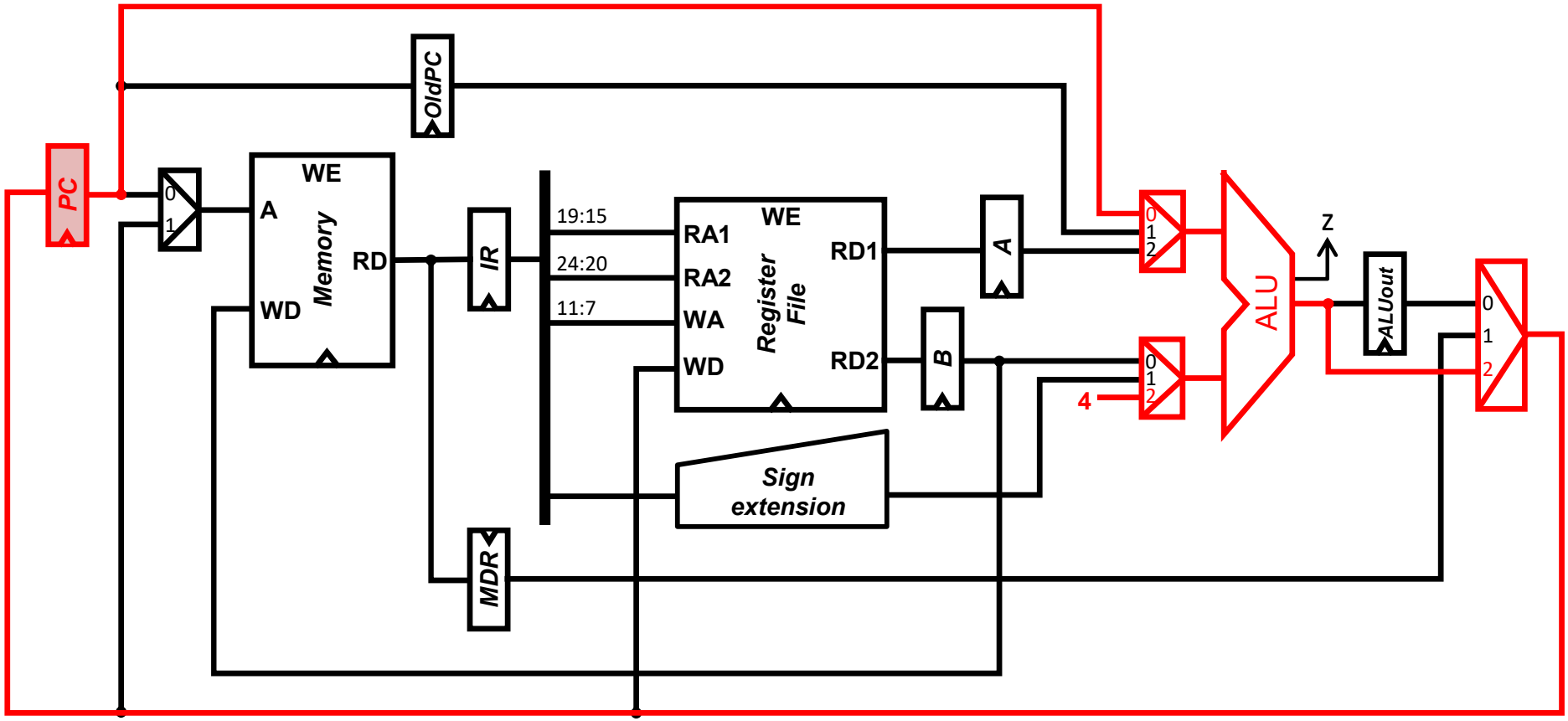


RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4

Controller design

`lw` instruction: register transfers

1. IR ← Mem[PC], PC ← PC + 4

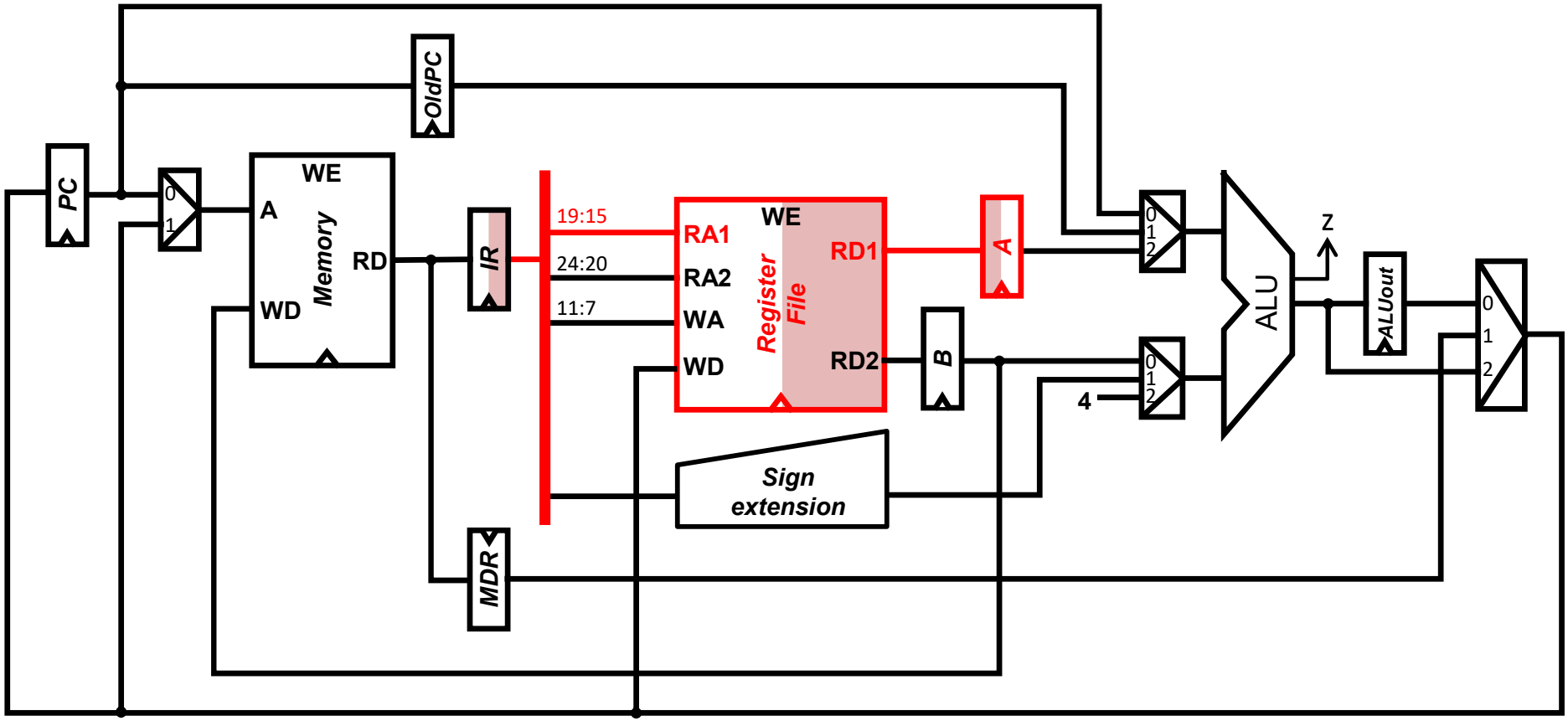


RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4

Controller design

`lw` instruction: register transfers

1. IR ← Mem[PC], PC ← PC + 4
2. A ← RF[rs1]

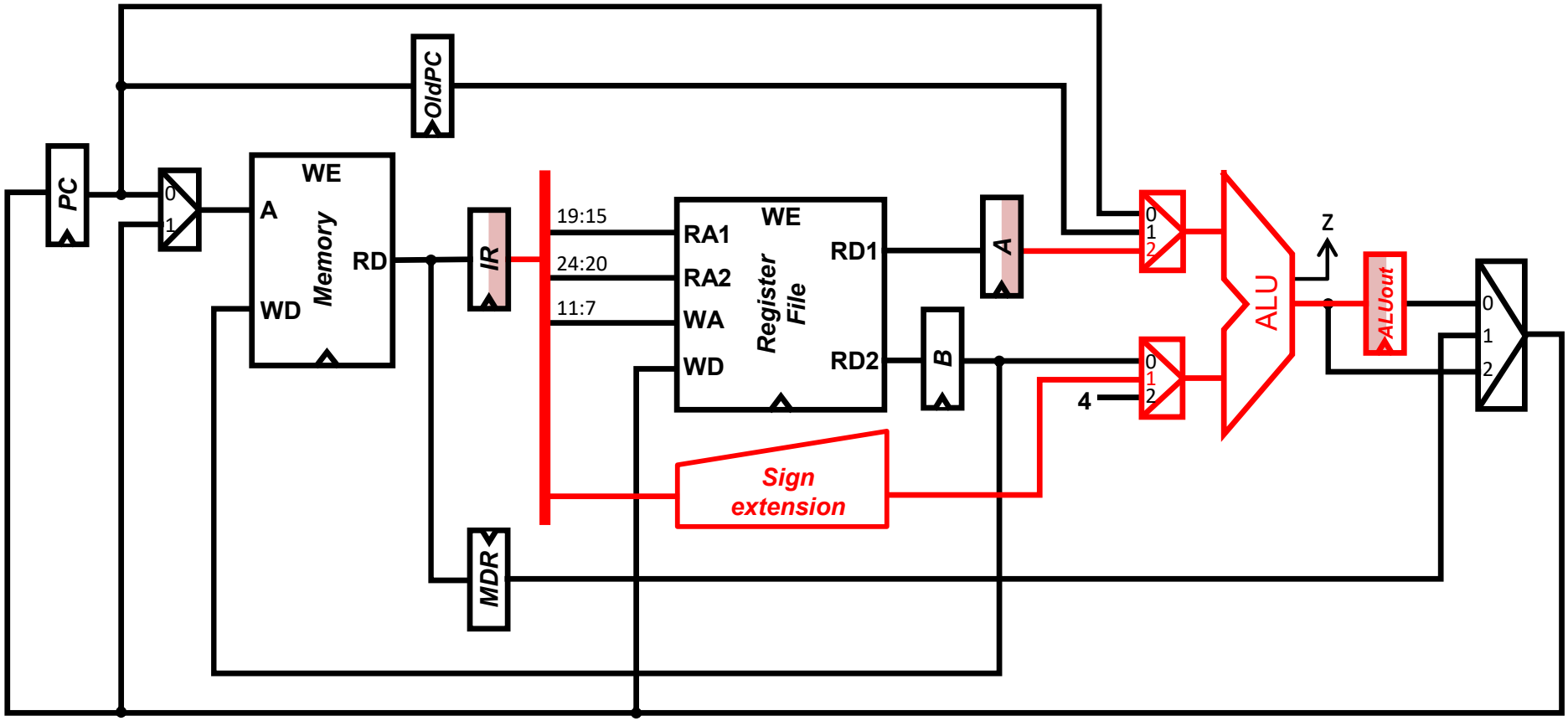


RF[rd] ← Mem[RF[rs1] + sExt(imm)], PC ← PC+4

Controller design

`lw` instruction: register transfers

1. IR ← Mem[PC], PC ← PC + 4
2. A ← RF[rs1]
3. ALUout ← A + sExt(imm)



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

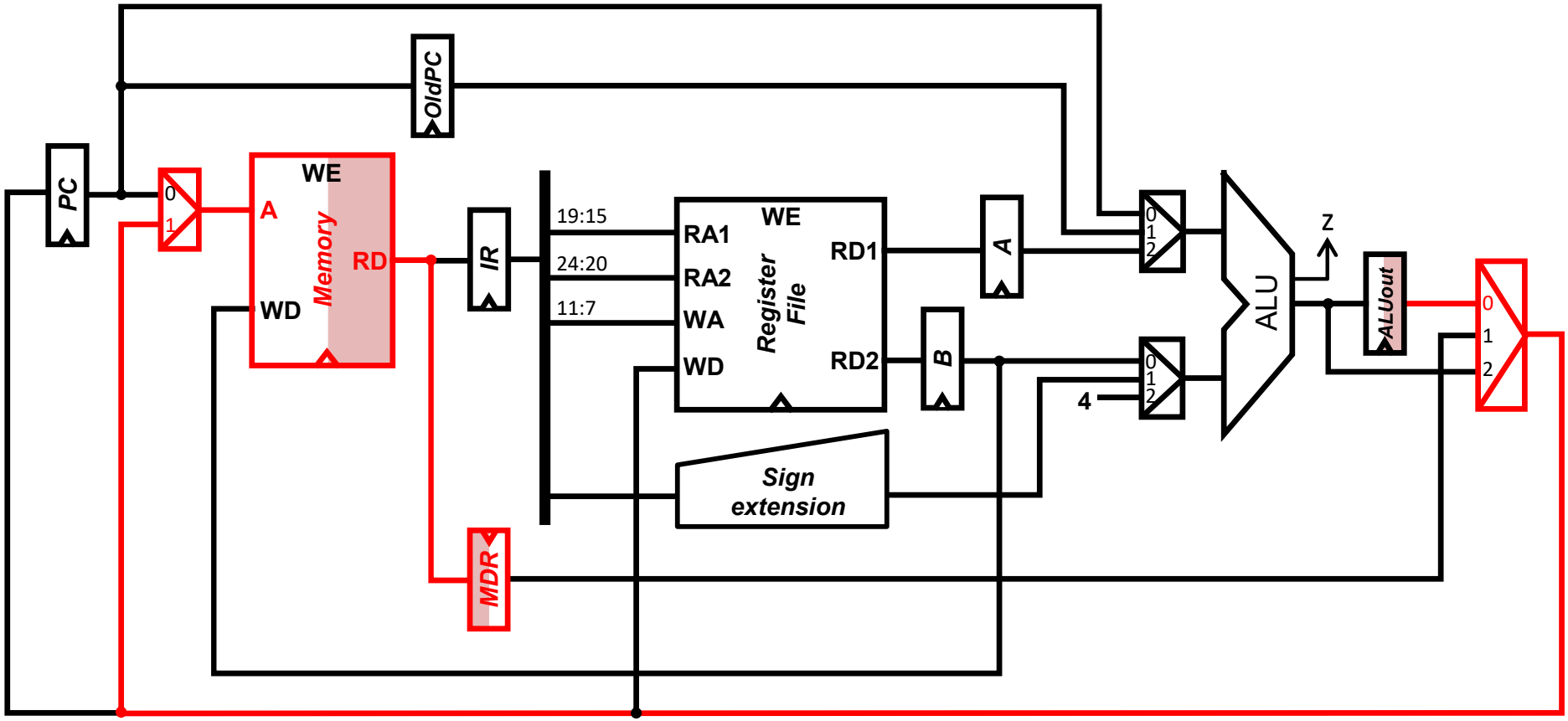
Controller design

`lw` instruction: register transfers

1. IR ← Mem[PC], PC ← PC + 4 **4.** MDR ← Mem[ALUout]

2. A ← RF[rs1]

3. ALUout ← A + sExt(imm)



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

Controller design

`lw` instruction: register transfers

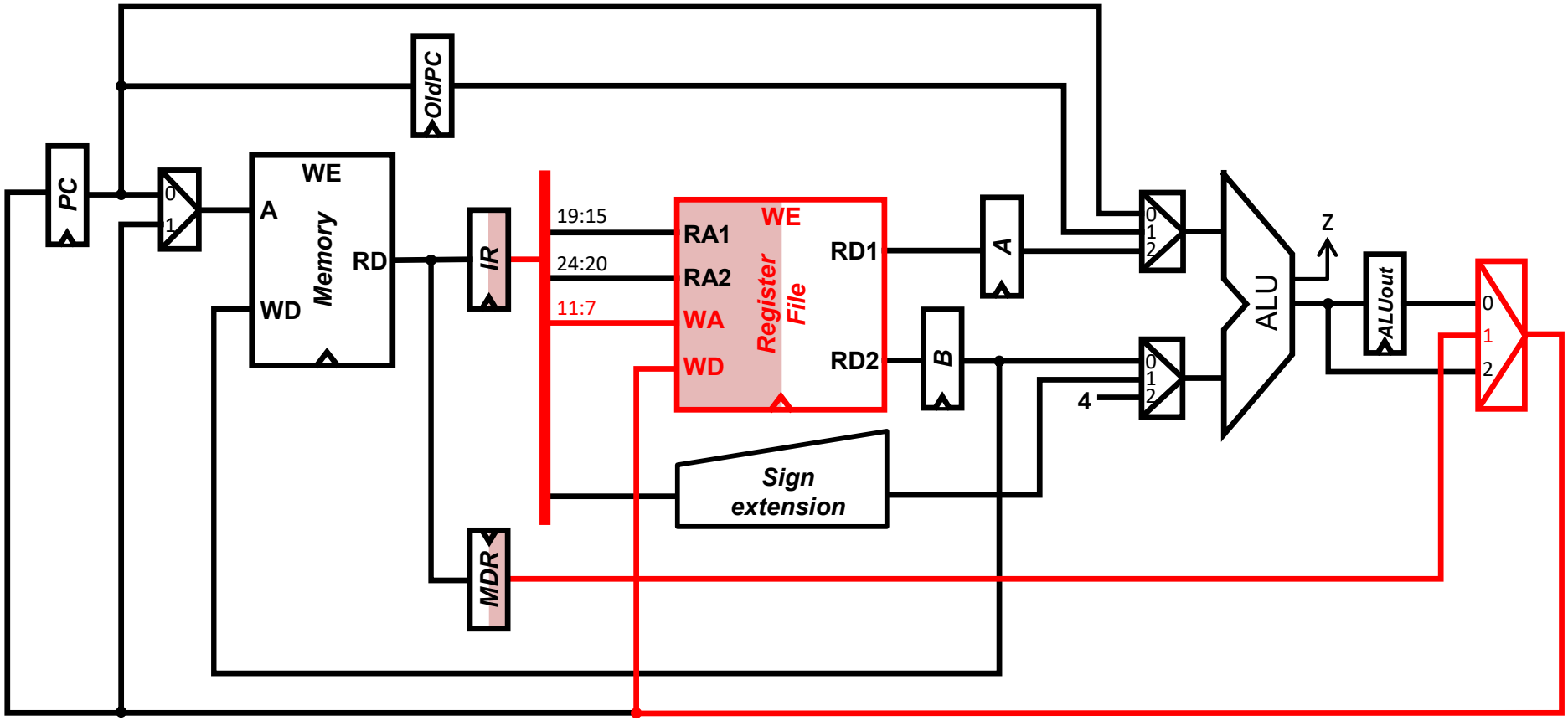
1. IR ← Mem[PC], PC ← PC + 4

2. A ← RF[rs1]

3. ALUout ← A + sExt(imm)

4. MDR ← Mem[ALUout]

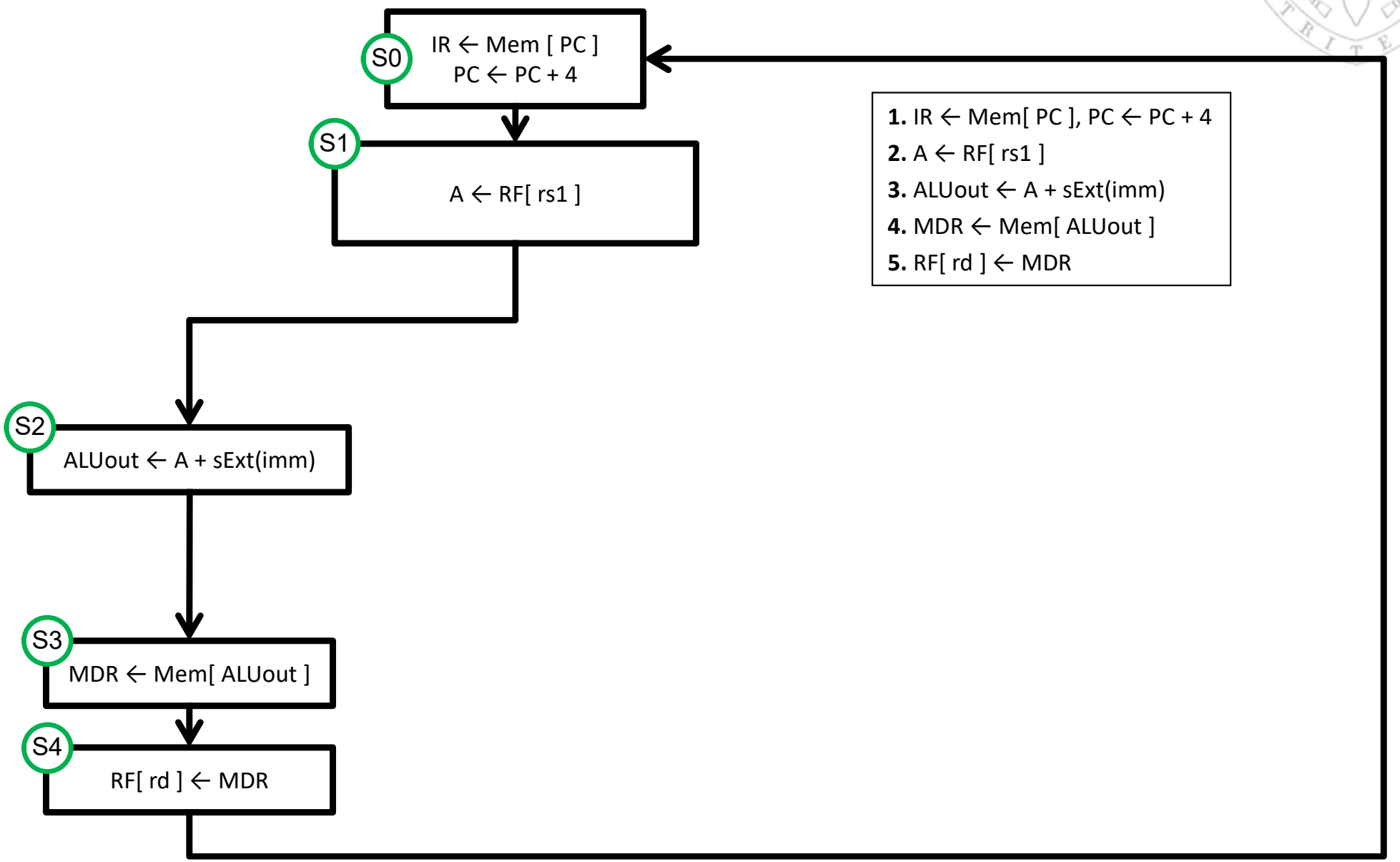
5. RF[rd] ← MDR



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

Controller design

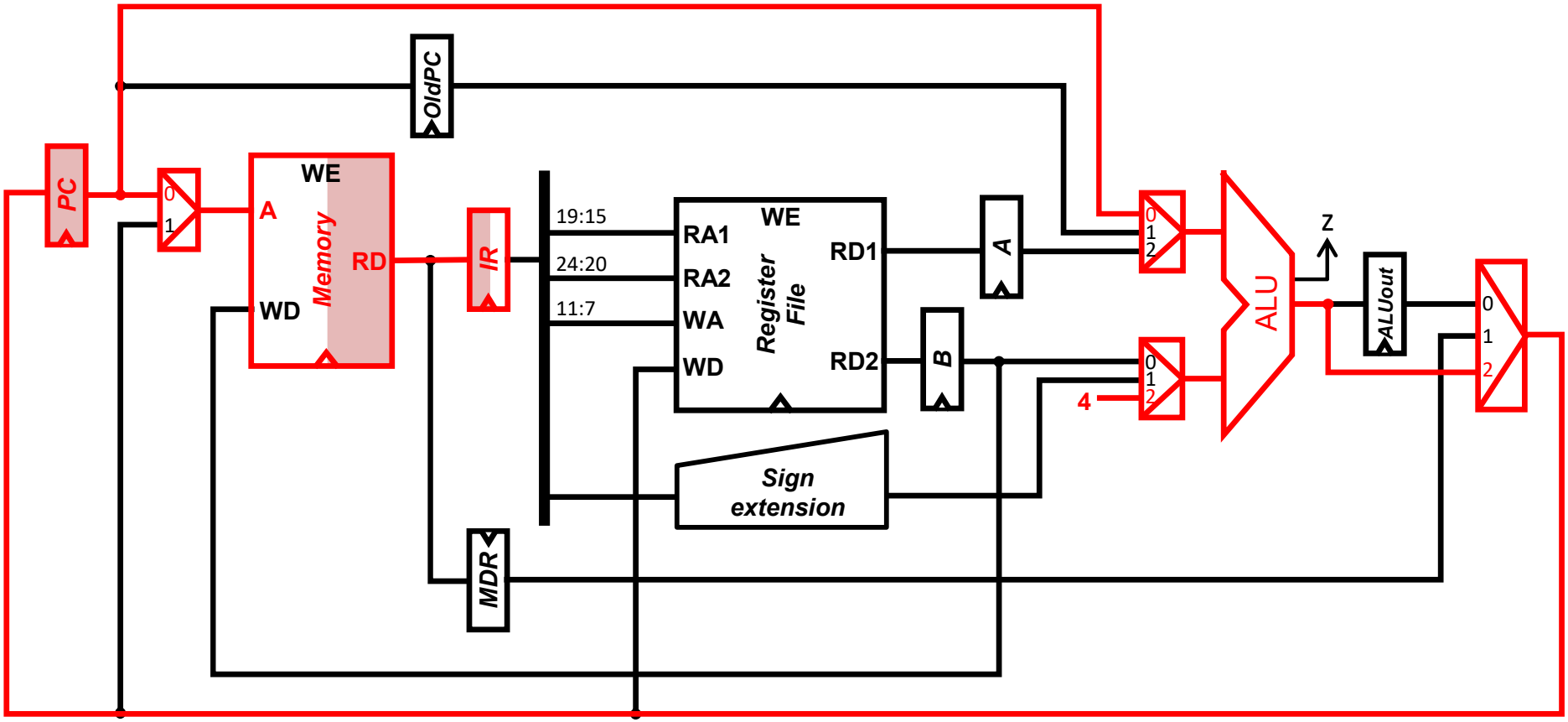
ASM diagram of the main FSM: `lw` instruction



Controller design

`sw` instruction: register transfers

1. IR ← Mem[PC], PC ← PC + 4



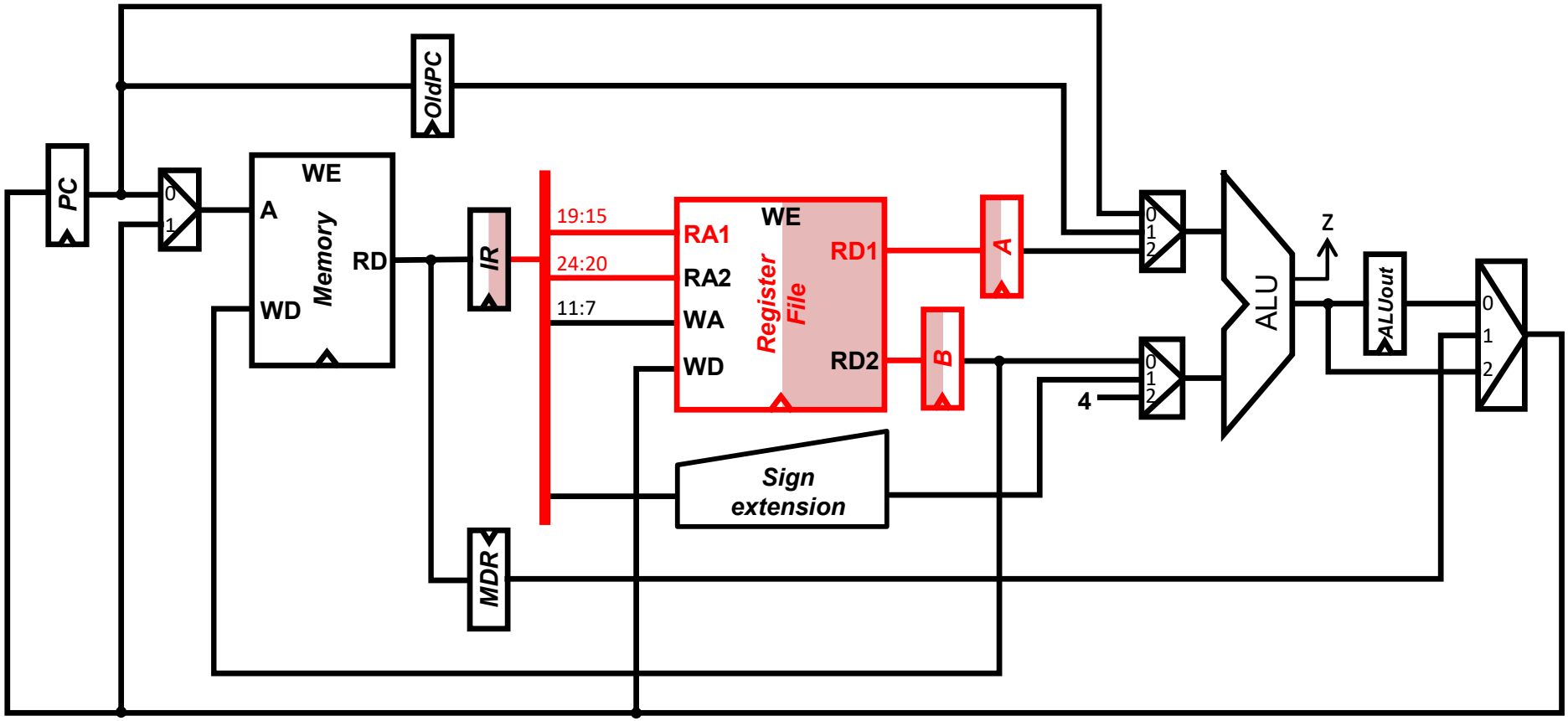
Mem[RF[rs1] + sExt(imm)] ← RF[rs2], PC ← PC+4

Controller design

`sw` instruction: register transfers

1. IR ← Mem[PC], PC ← PC + 4

2. A ← RF[rs1], B ← RF[rs2]



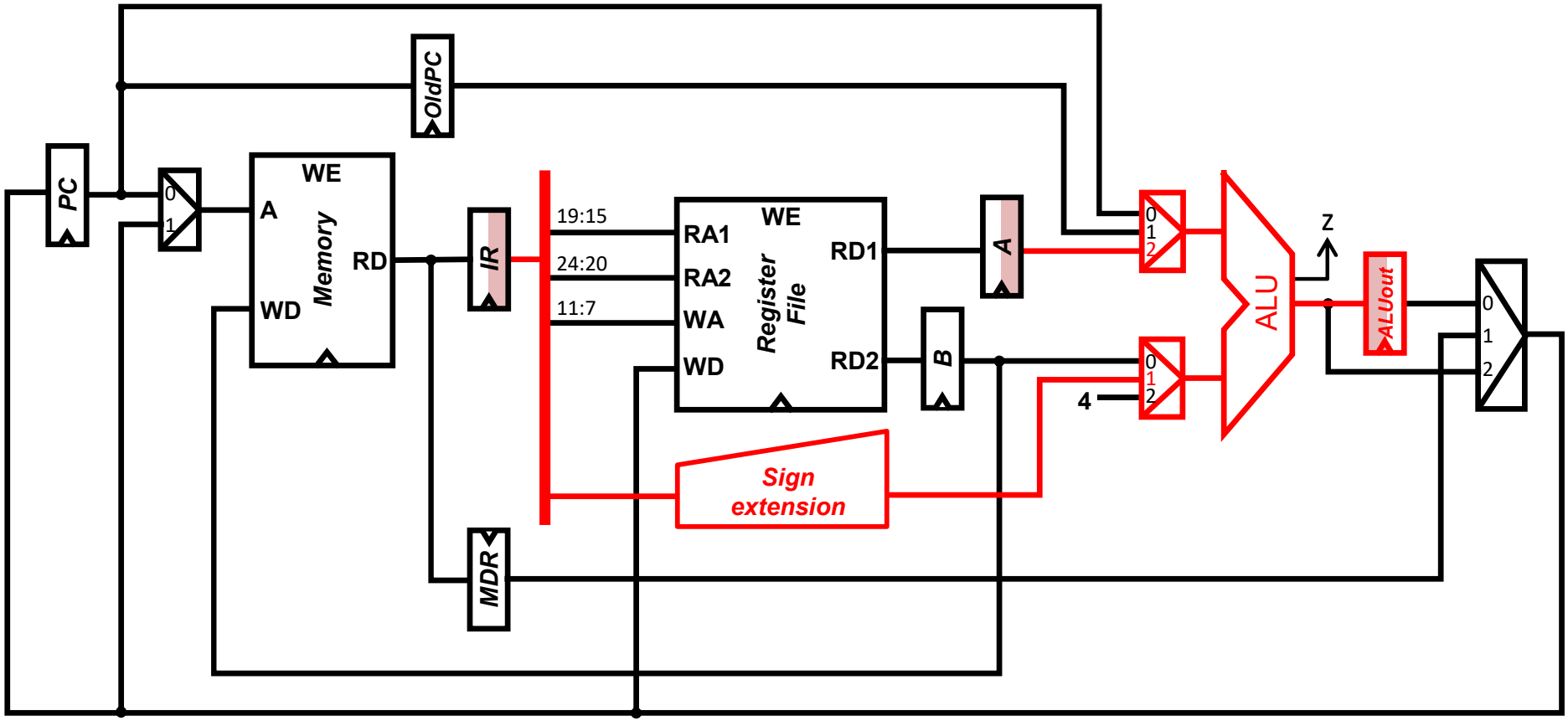
Mem[RF[rs1] + sExt(imm)] ← RF[rs2], PC ← PC+4

Controller design

`sw` instruction: register transfers

1. IR ← Mem[PC], PC ← PC + 4 **3.** ALUout ← A + sExt(imm)

2. A ← RF[rs1], B ← RF[rs2]



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

Controller design

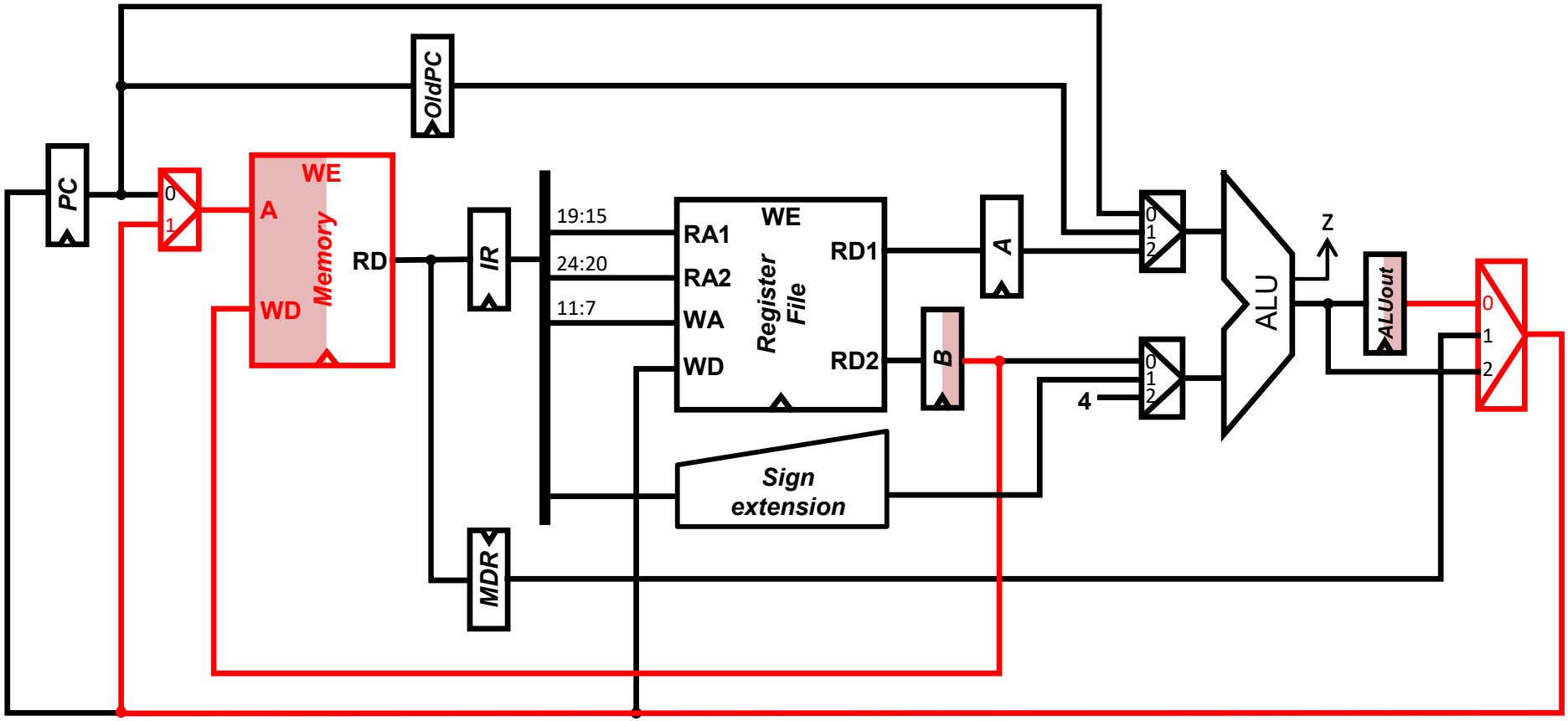
`sw` instruction: register transfers

1. IR ← Mem[PC], PC ← PC + 4

2. A ← RF[rs1], B ← RF[rs2]

3. ALUout ← A + sExt(imm)

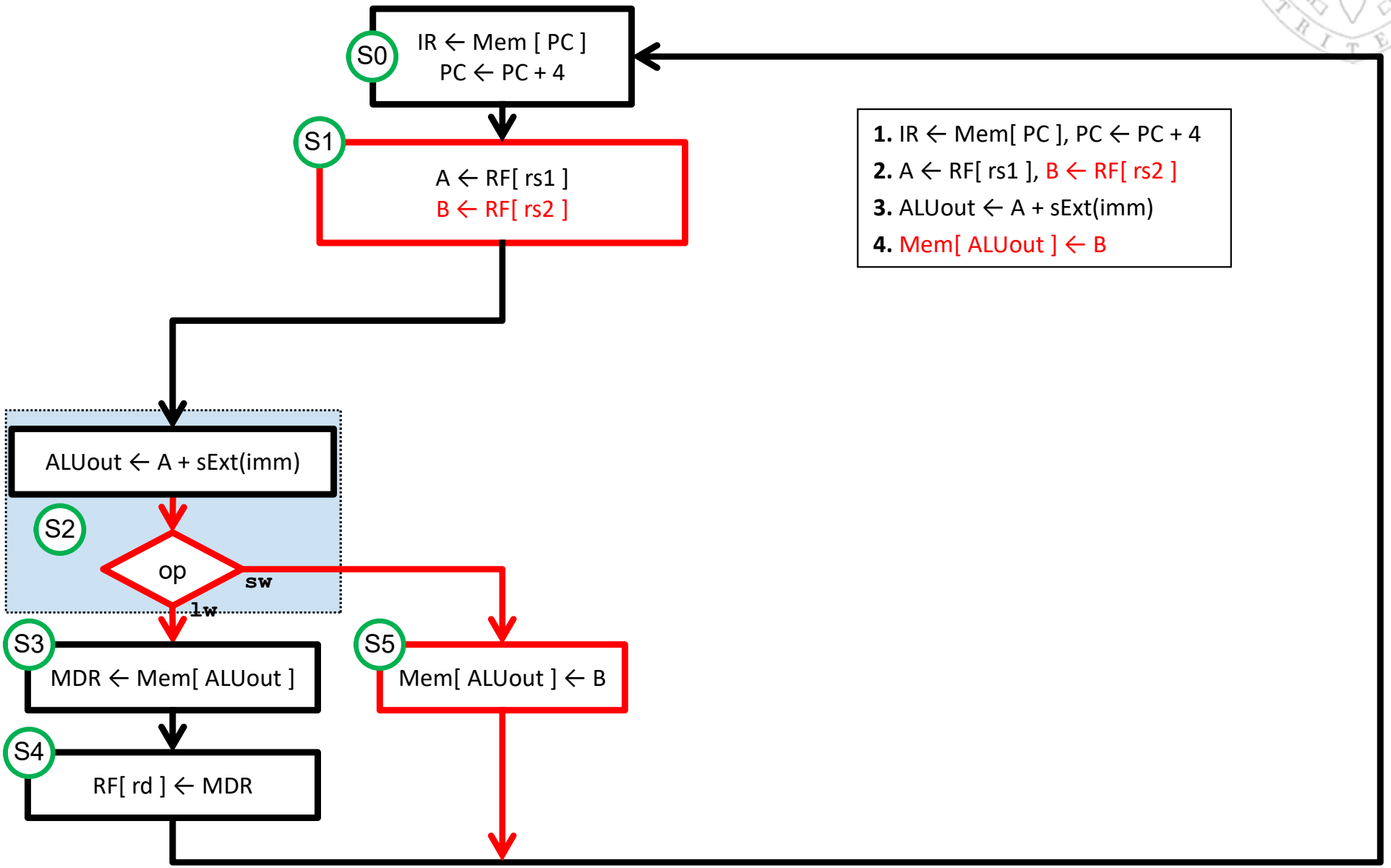
4. Mem[ALUout] ← B



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

Controller design

ASM diagram of the main FSM: **sw** instruction

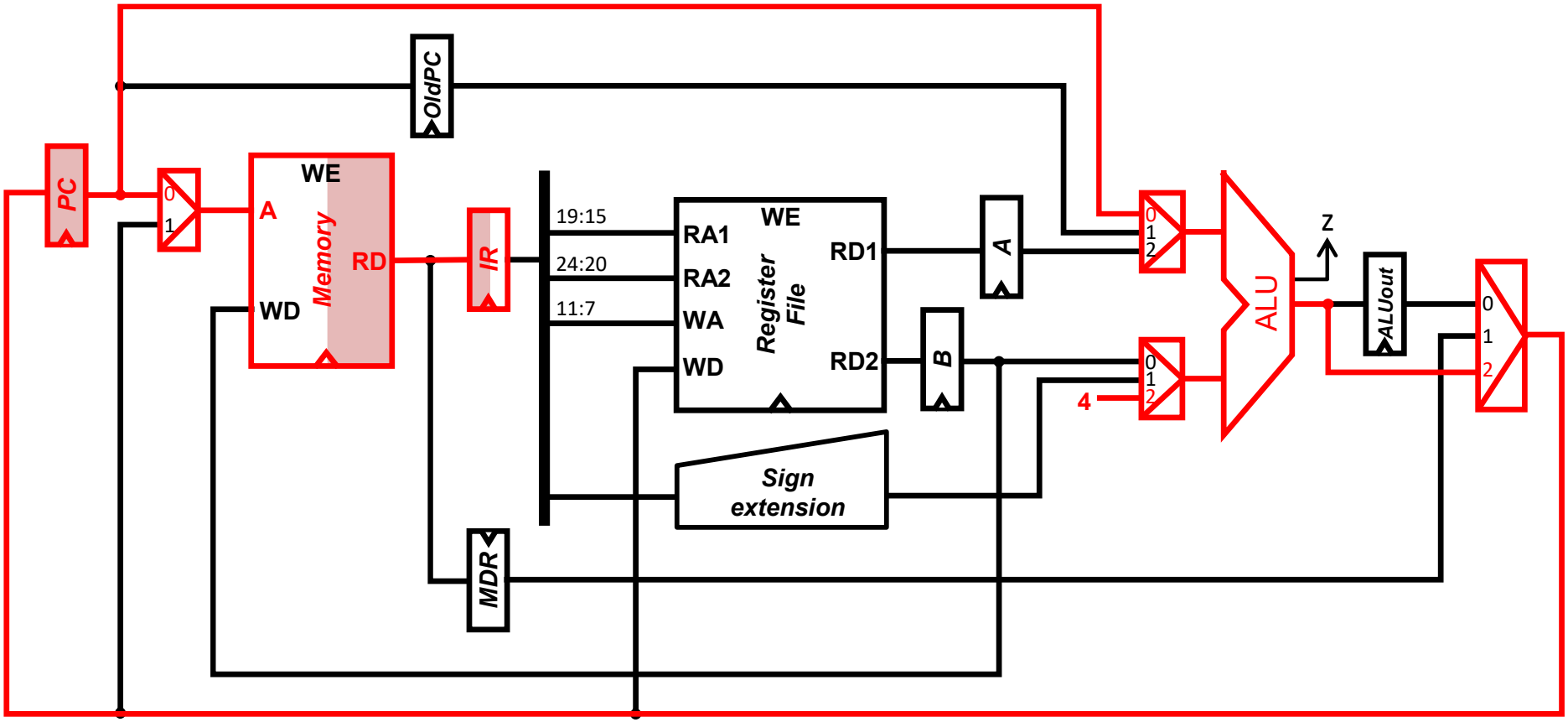


1. IR ← Mem[PC], PC ← PC + 4
2. A ← RF[rs1], B ← RF[rs2]
3. ALUout ← A + sExt(imm)
4. Mem[ALUout] ← B

Controller design

`addi`-like instructions: register transfers

1. IR ← Mem[PC], PC ← PC + 4

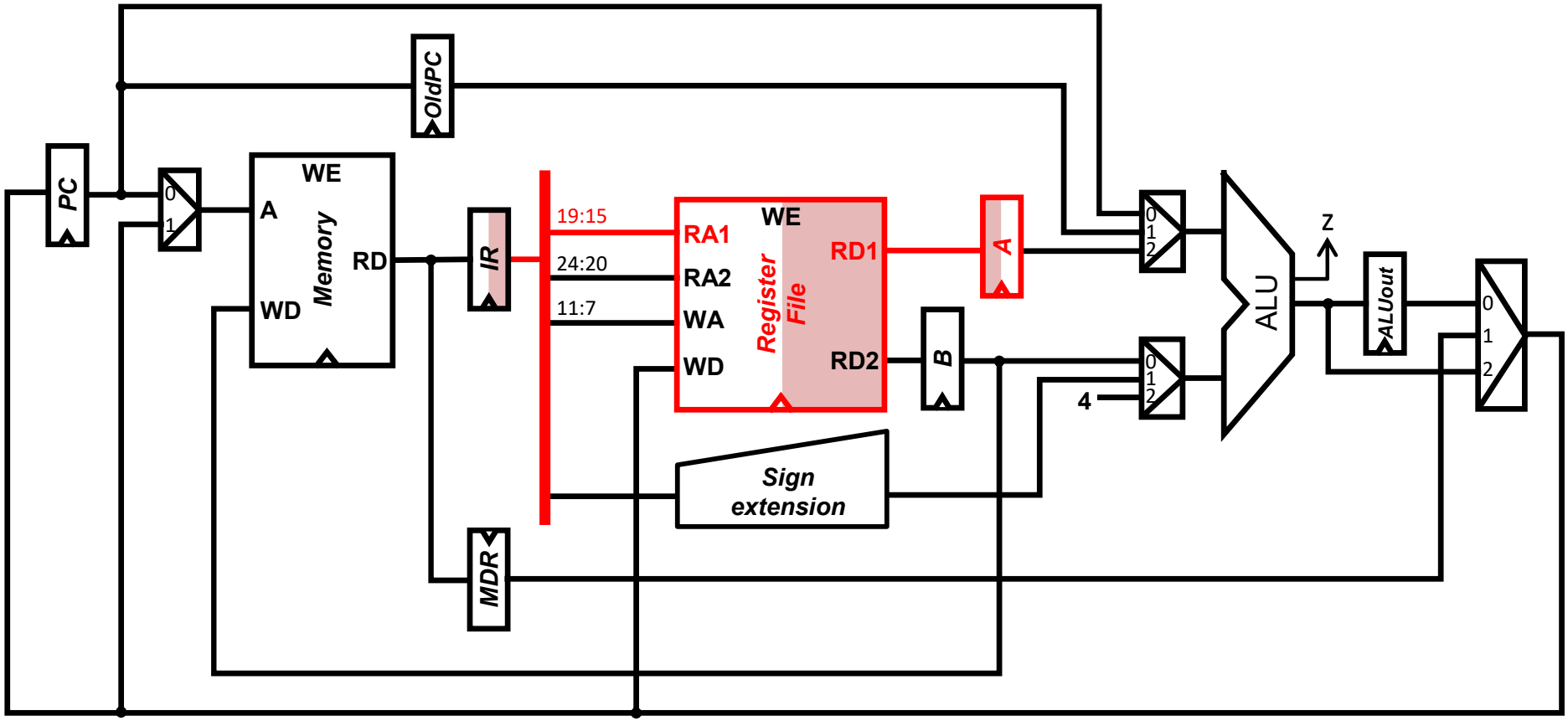


RF[rd] ← RF[rs1] *op* sExt(imm), PC ← PC+4

Controller design

`addi`-like instructions: register transfers

1. IR ← Mem[PC], PC ← PC + 4
2. A ← RF[rs1]



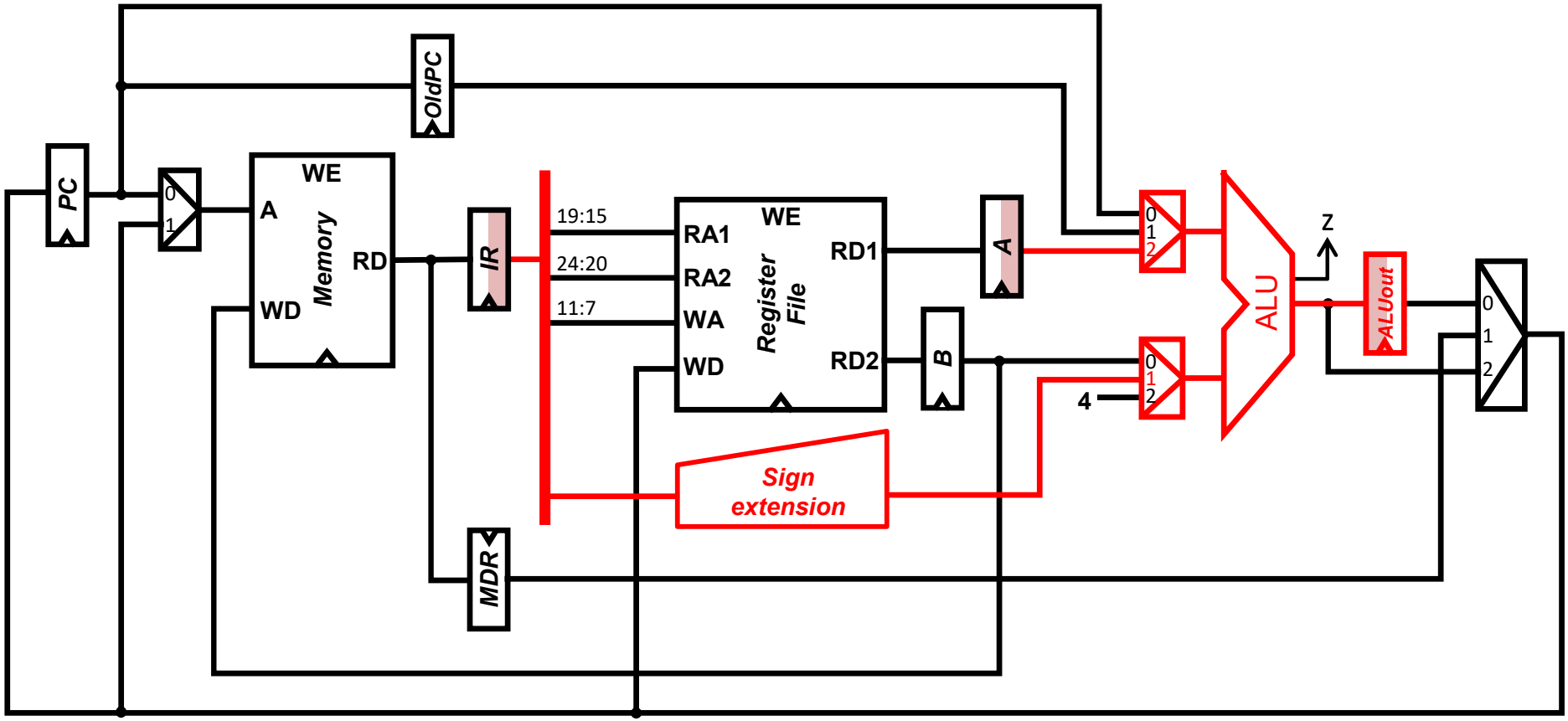
RF[rd] ← RF[rs1] *op* sExt(imm), PC ← PC+4

Controller design

`addi`-like instructions: register transfers

1. IR ← Mem[PC], PC ← PC + 4 **3.** ALUout ← A *op* sExt(imm)

2. A ← RF[rs1]



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

Controller design

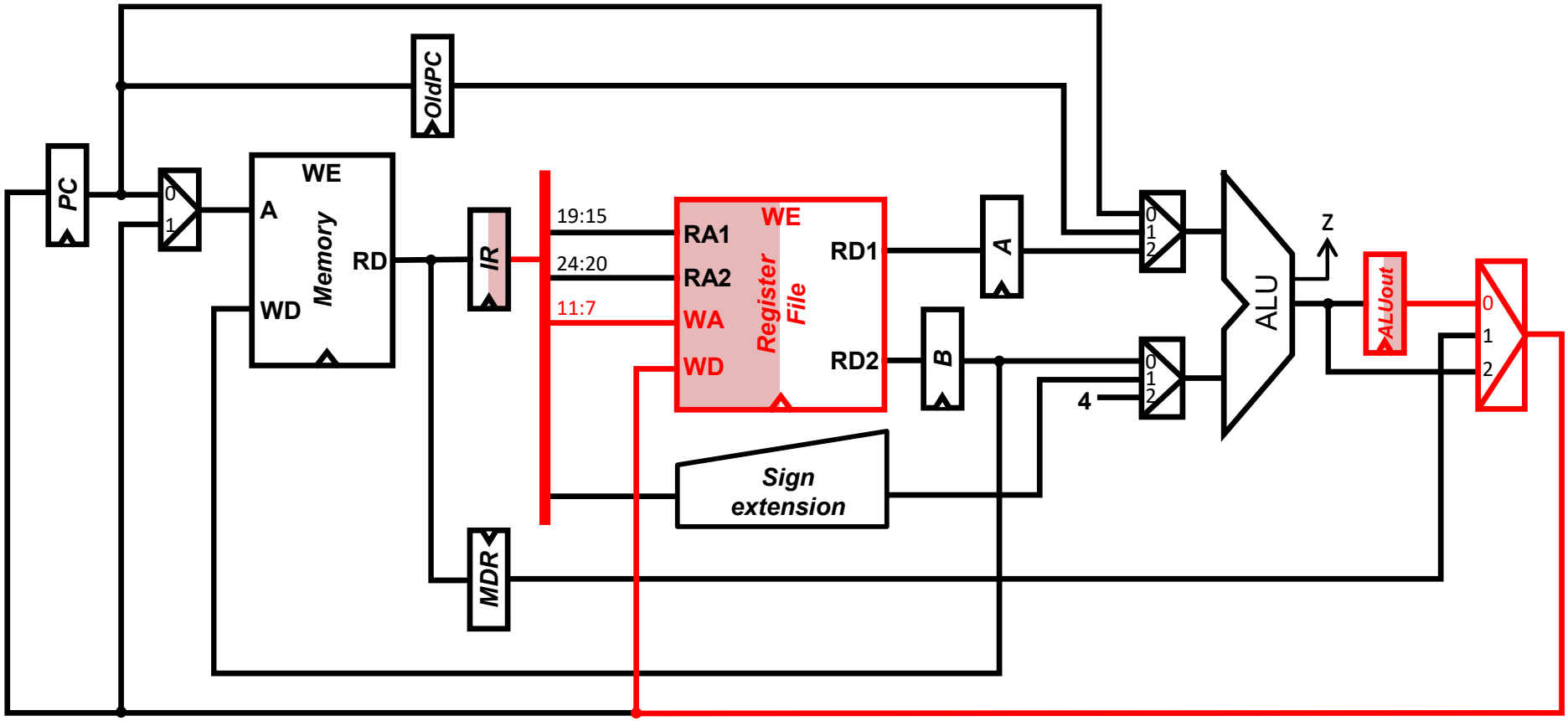
`addi`-like instructions: register transfers

1. IR ← Mem[PC], PC ← PC + 4

2. A ← RF[rs1]

3. ALUout ← A *op* sExt(imm)

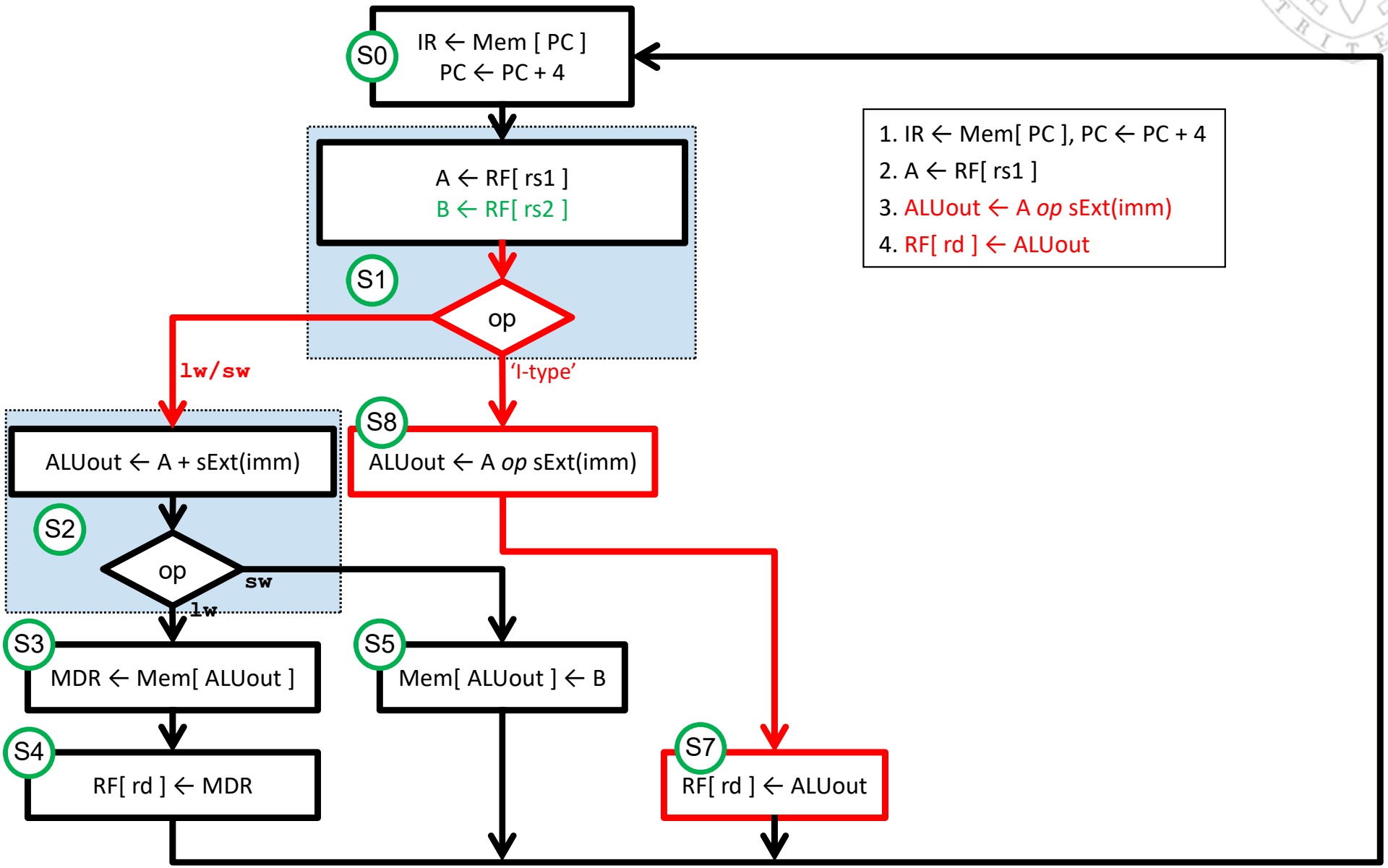
4. RF[rd] ← ALUout



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

Controller design

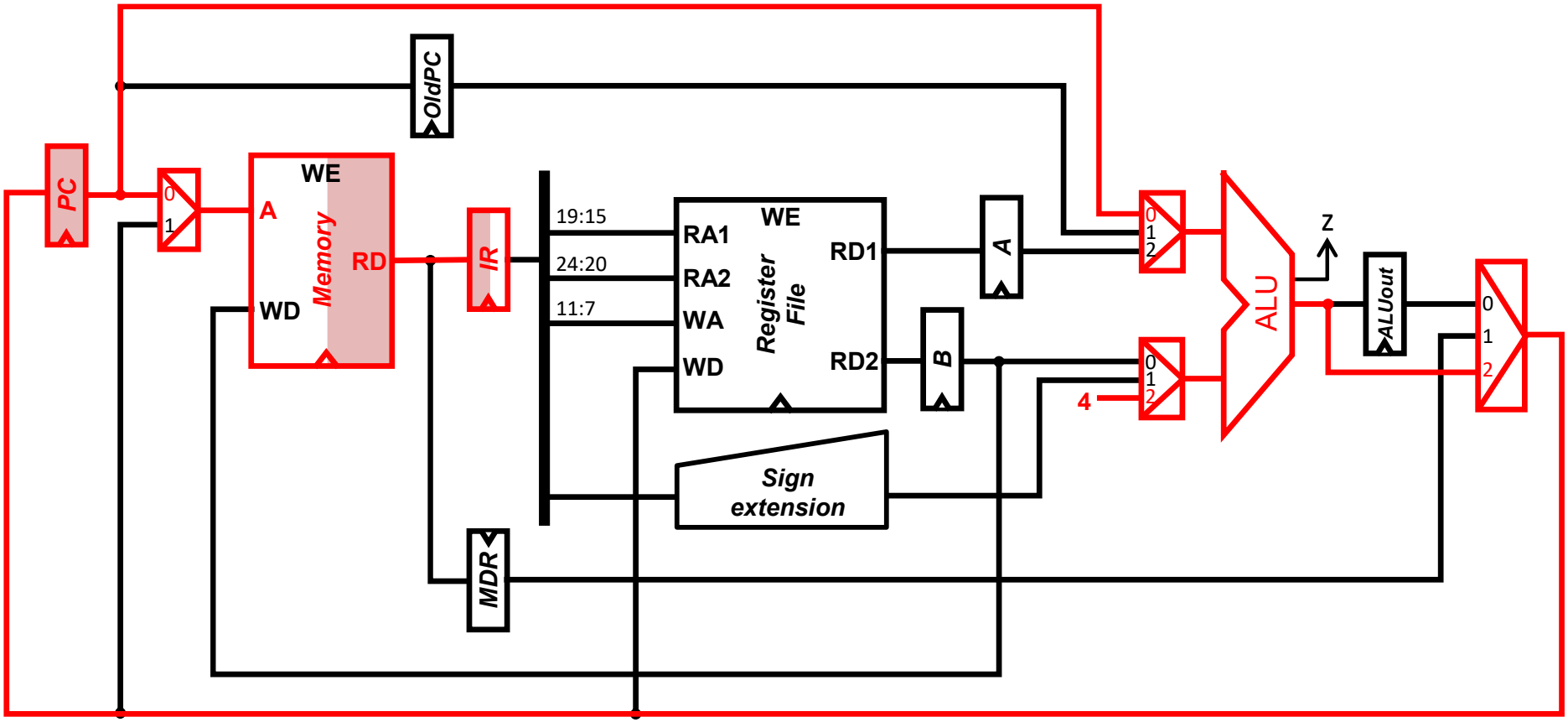
ASM diagram of the main FSM: **addi**-like instructions



Controller design

add-like instructions: register transfers

1. IR ← Mem[PC], PC ← PC + 4

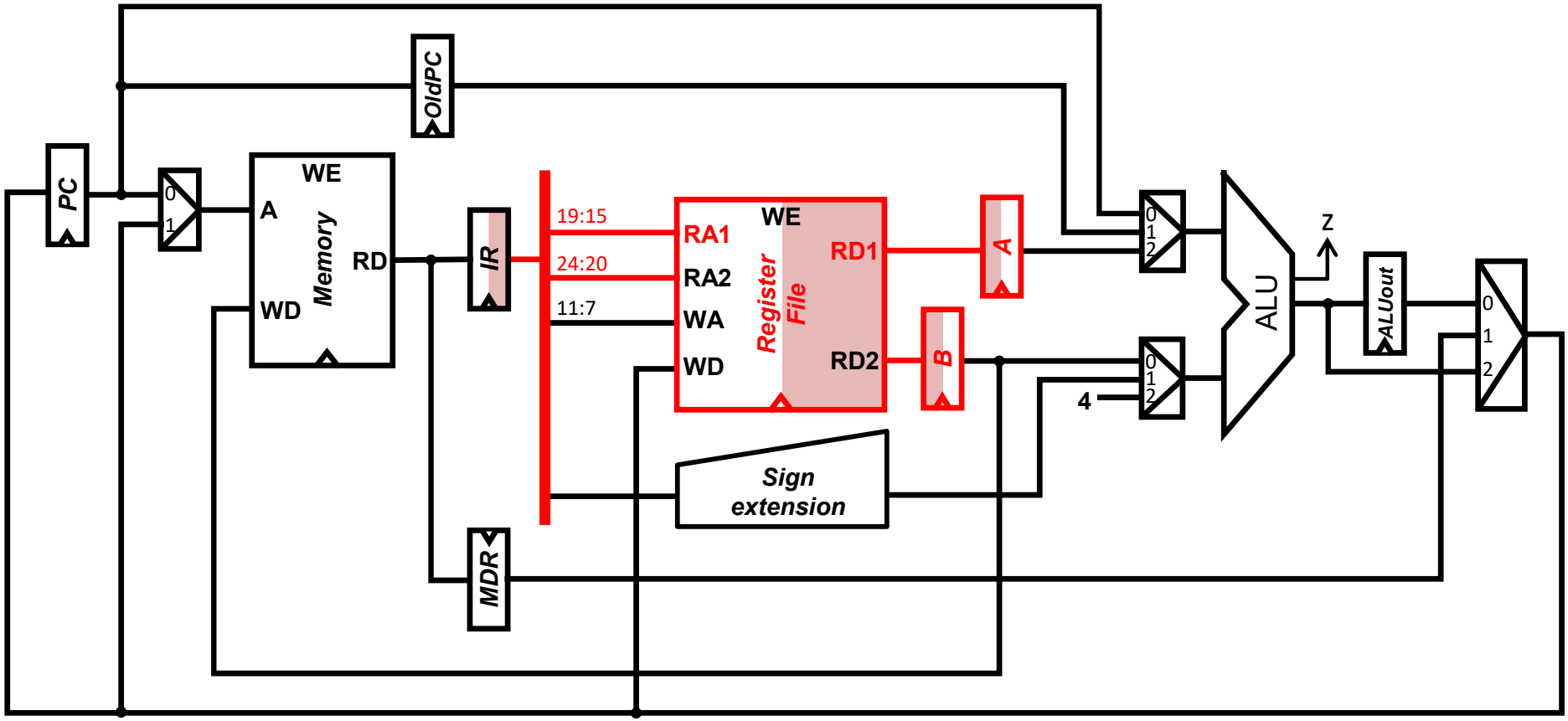


RF[rd] ← RF[rs1] *op* RF[rs2], PC ← PC+4

Controller design

add-like instructions: register transfers

1. IR ← Mem[PC], PC ← PC + 4
2. A ← RF[rs1], B ← RF[rs2]



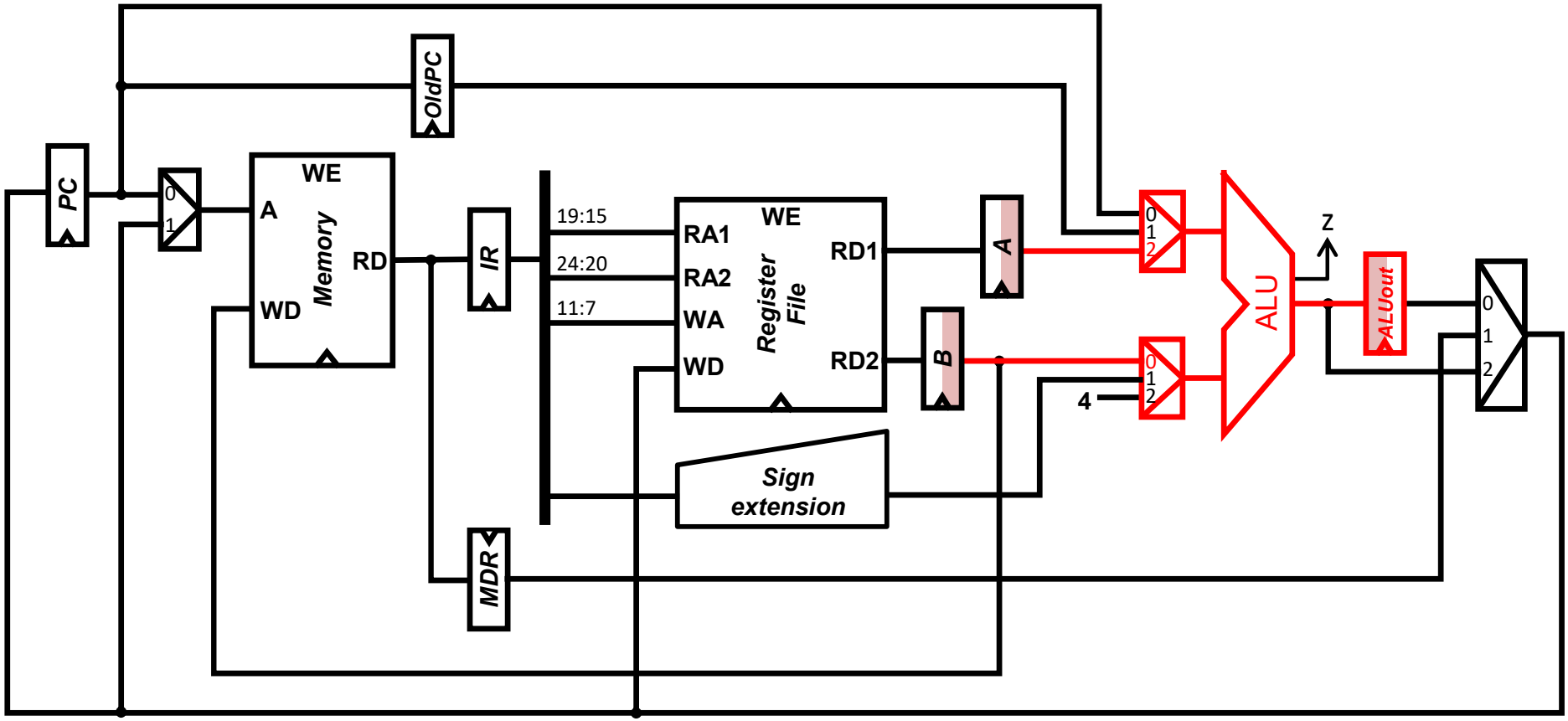
RF[rd] ← RF[rs1] *op* RF[rs2], PC ← PC+4

Controller design

add-like instructions: register transfers

1. IR ← Mem[PC], PC ← PC + 4 **3.** ALUout ← A *op* B

2. A ← RF[rs1], B ← RF[rs2]



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

Controller design

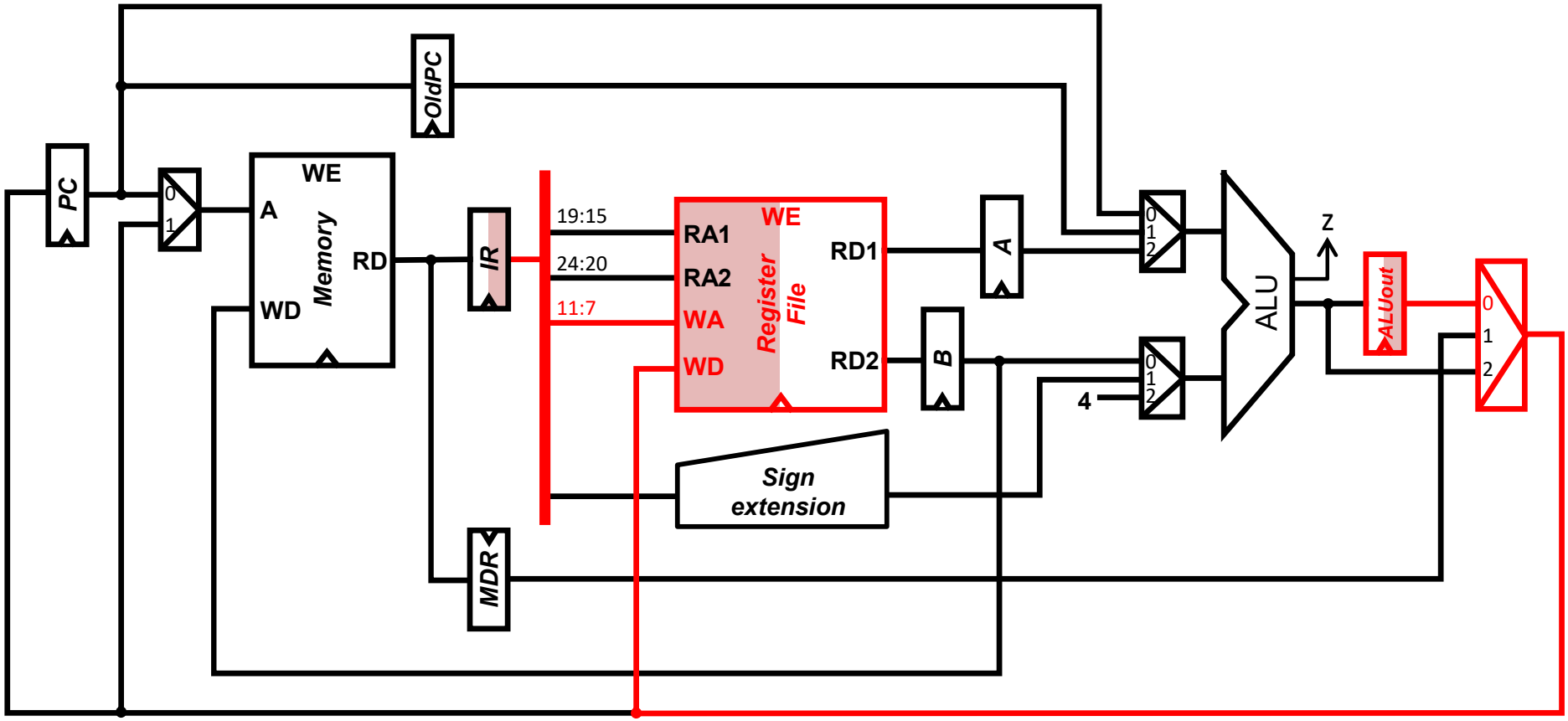
add-like instructions: register transfers

1. IR ← Mem[PC], PC ← PC + 4

2. A ← RF[rs1], B ← RF[rs2]

3. ALUout ← A *op* B

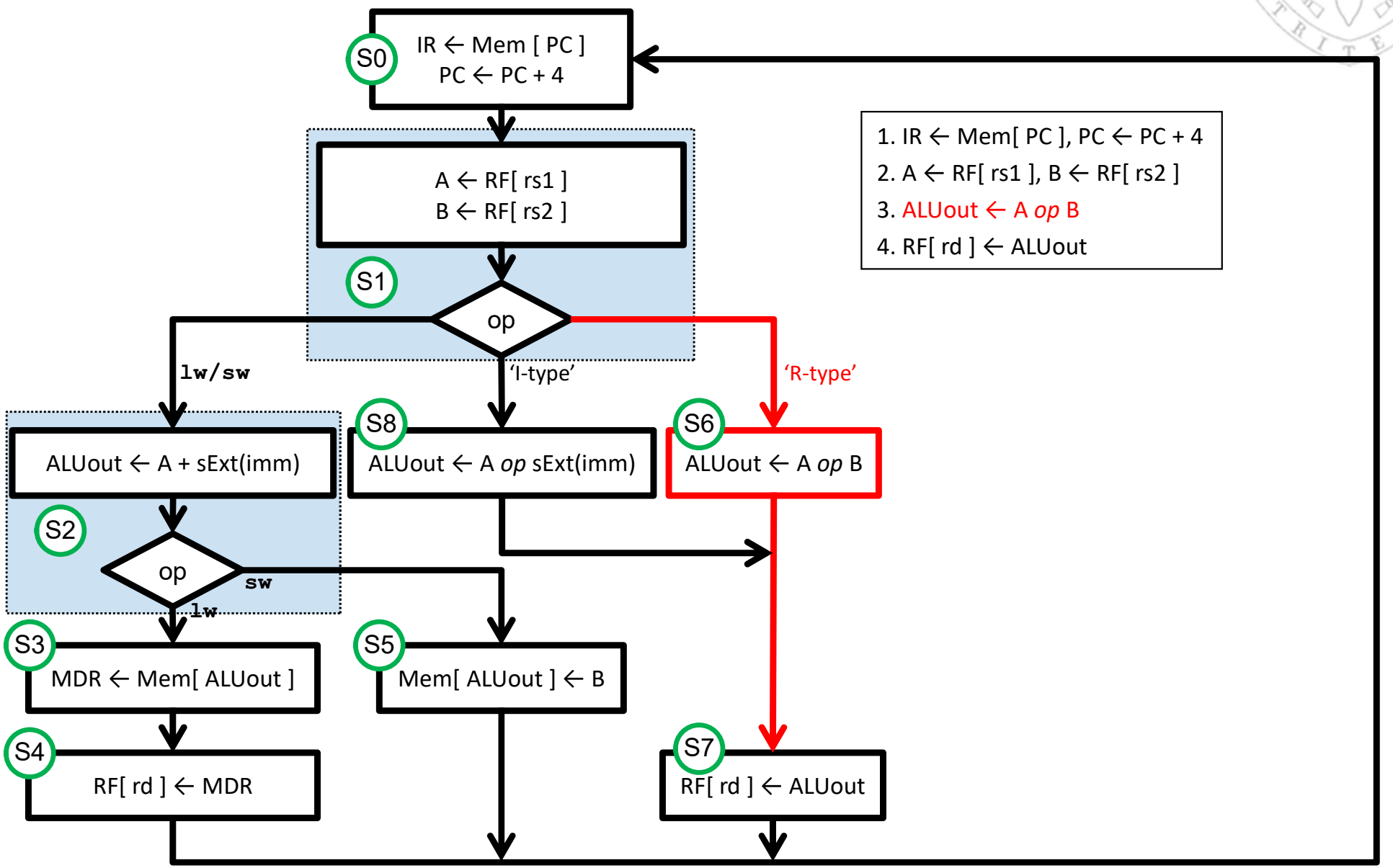
4. RF[rd] ← ALUout



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

Controller design

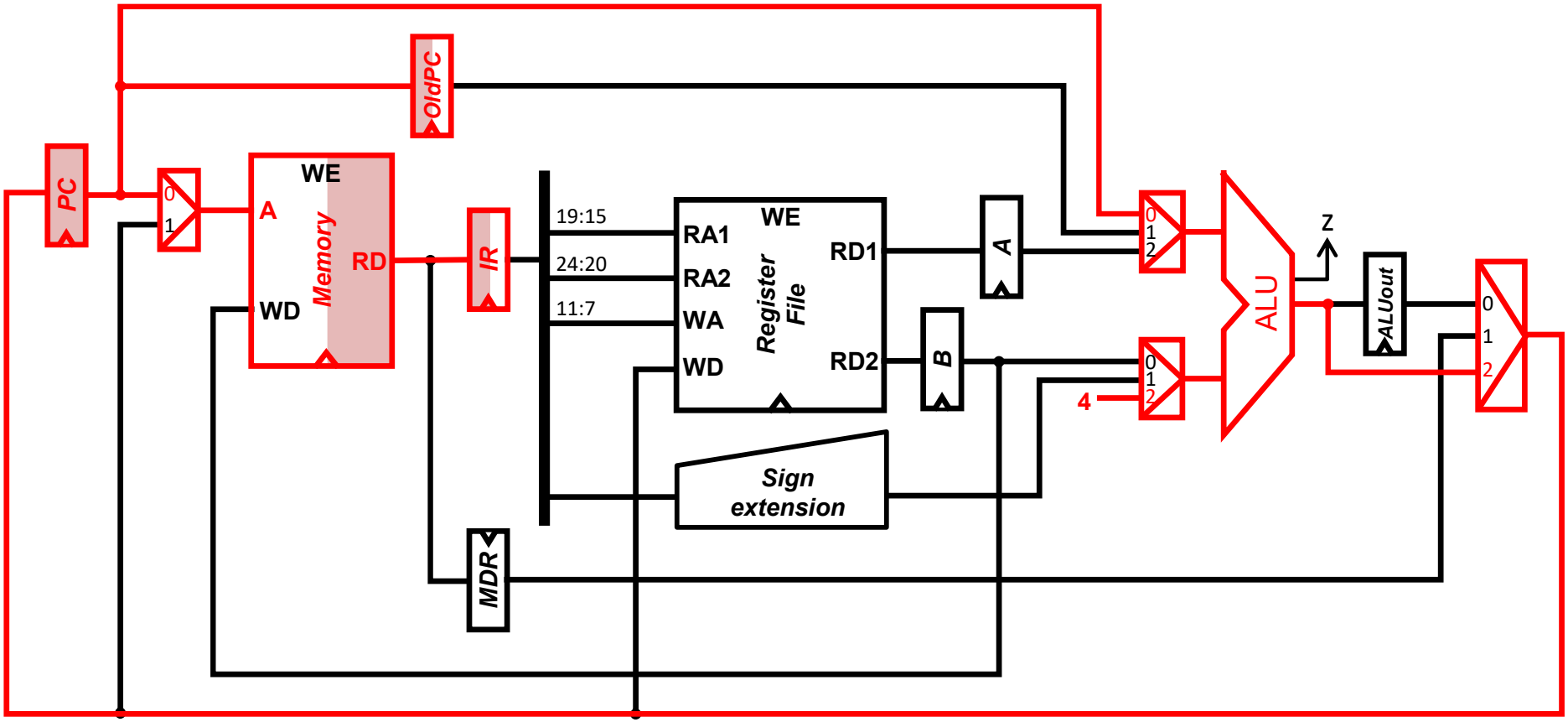
ASM diagram of the main FSM: **add**-like instructions



Controller design

beq instruction: register transfers

1. IR ← Mem[PC], PC ← PC + 4, OldPC ← PC

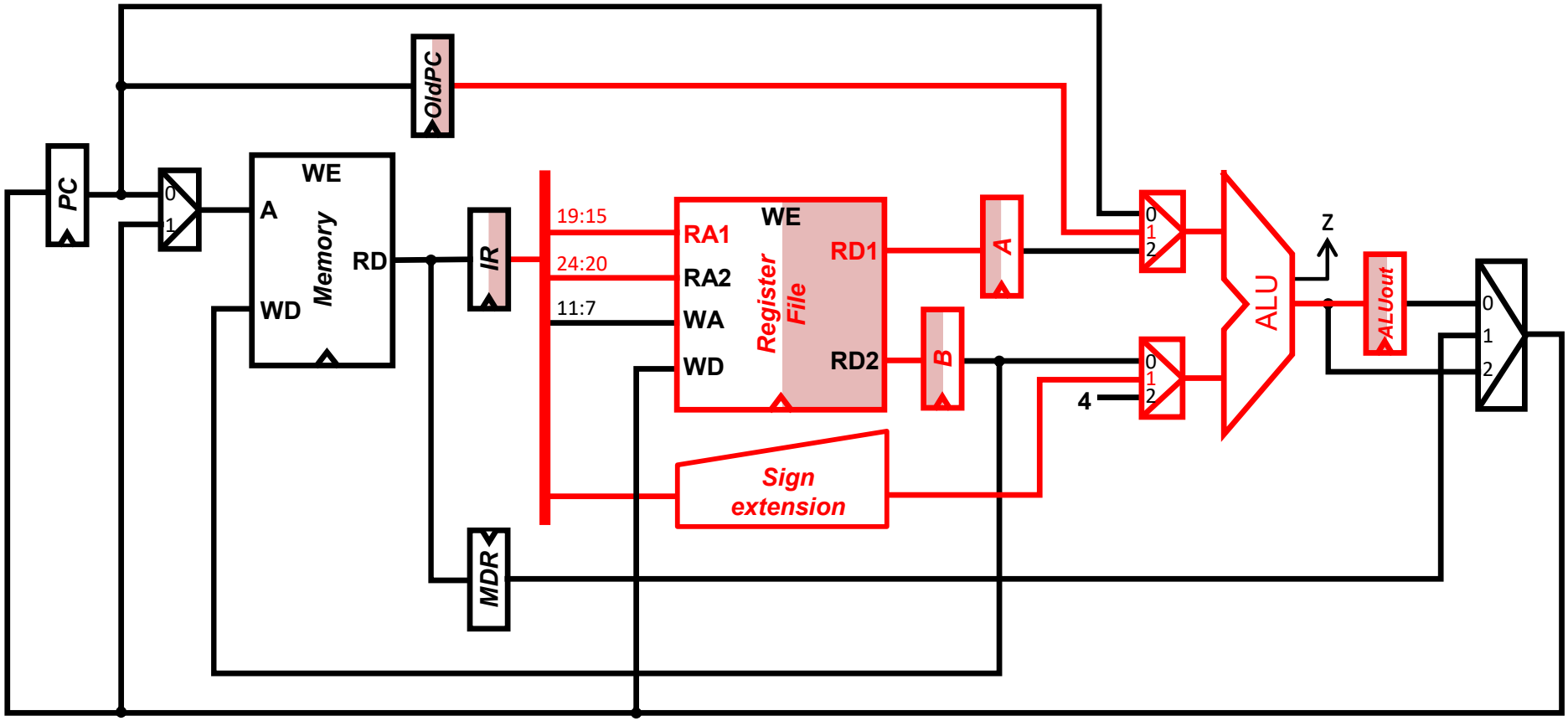


PC ← *if* (RF[rs1] = RF[rs2]) *then* (PC + sExt(imm)) *else* (PC+4)

Controller design

beq instruction: register transfers

1. IR ← Mem[PC], PC ← PC + 4, OldPC ← PC
2. A ← RF(rs1), B ← RF(rs2), ALUout ← oldPC + sExt(imm)

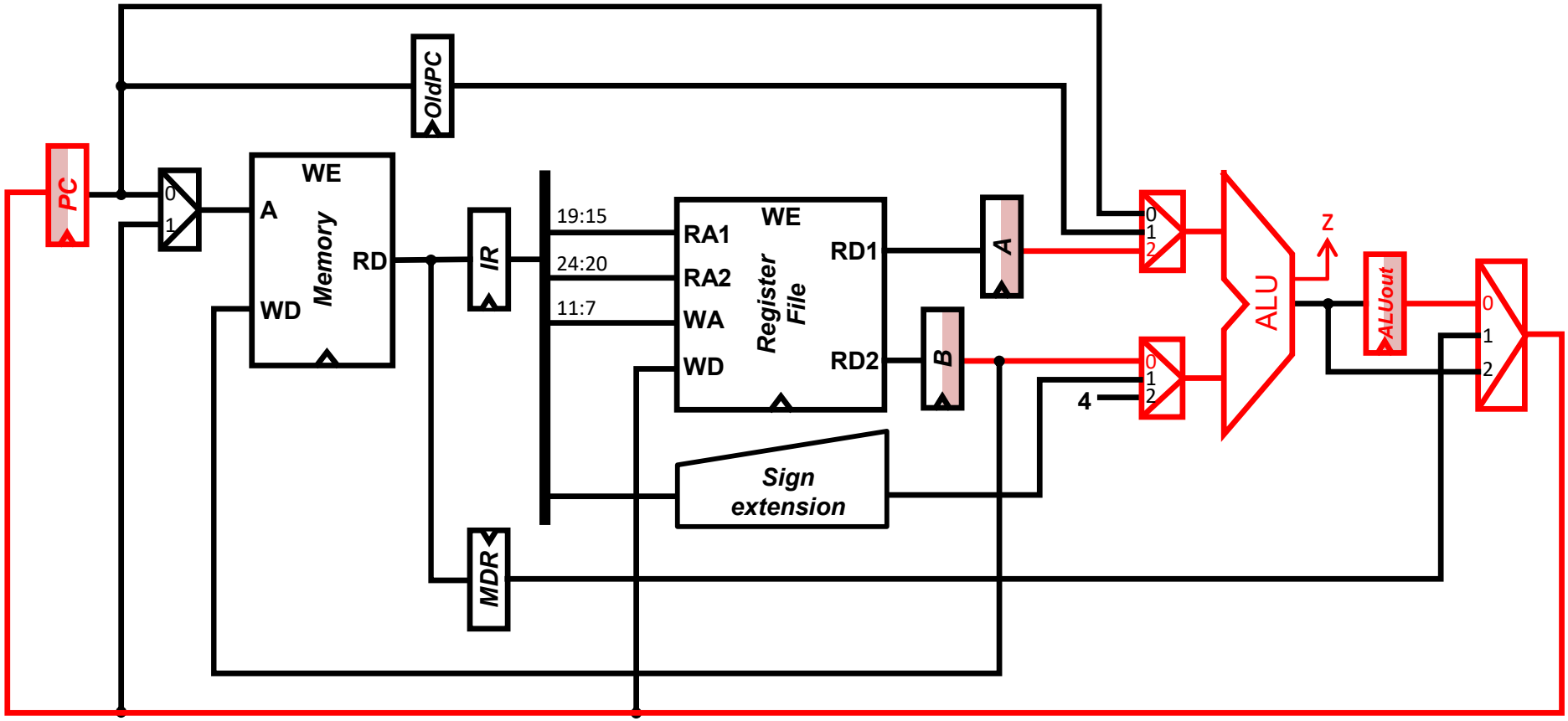


PC ← *if* (RF[rs1] = RF[rs2]) *then* (PC + sExt(imm)) *else* (PC+4)

Controller design

beq instruction: register transfers

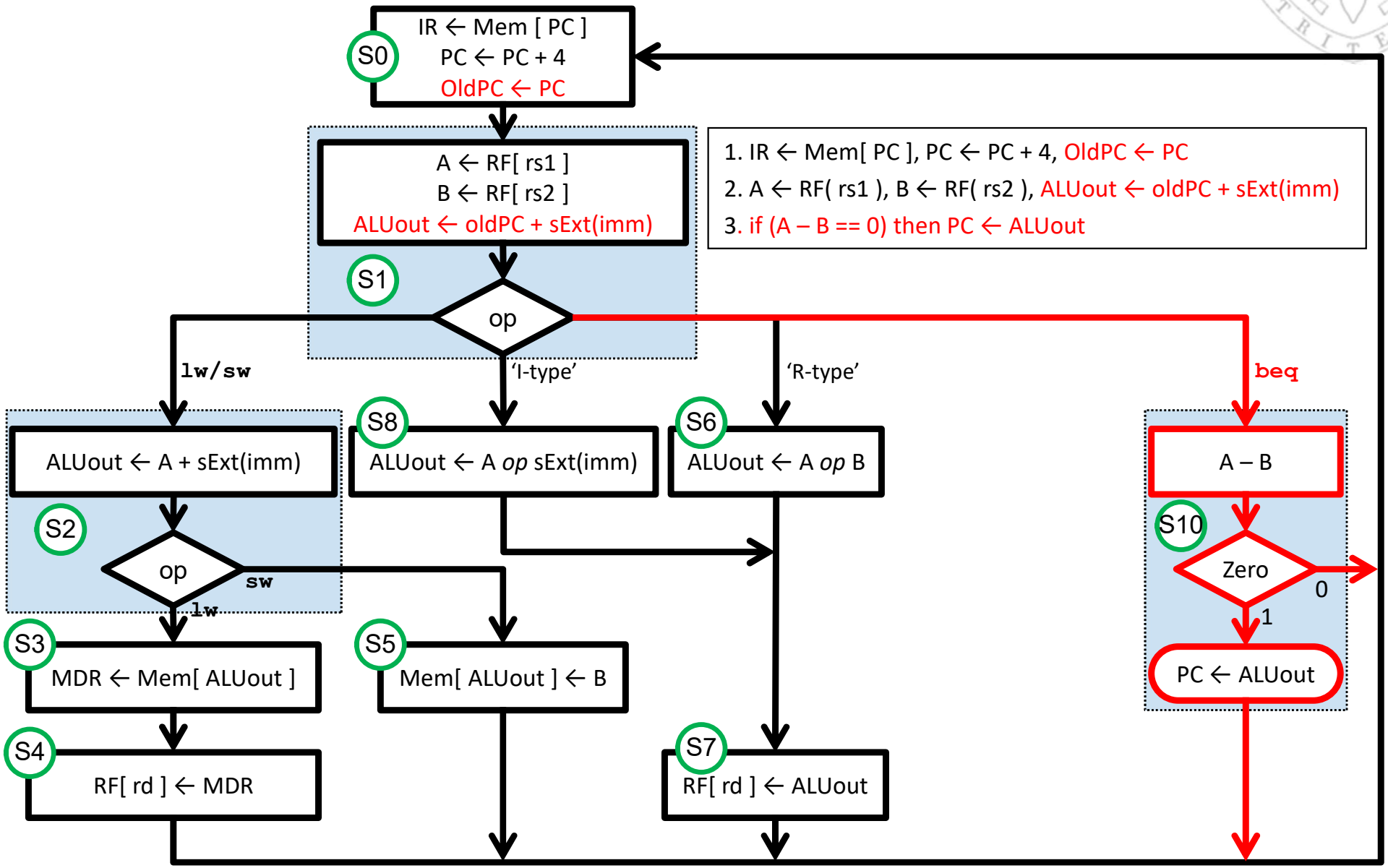
1. IR ← Mem[PC], PC ← PC + 4, OldPC ← PC
2. A ← RF(rs1), B ← RF(rs2), ALUout ← oldPC + sExt(imm)
3. if (A – B == 0) then PC ← ALUout



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

Controller design

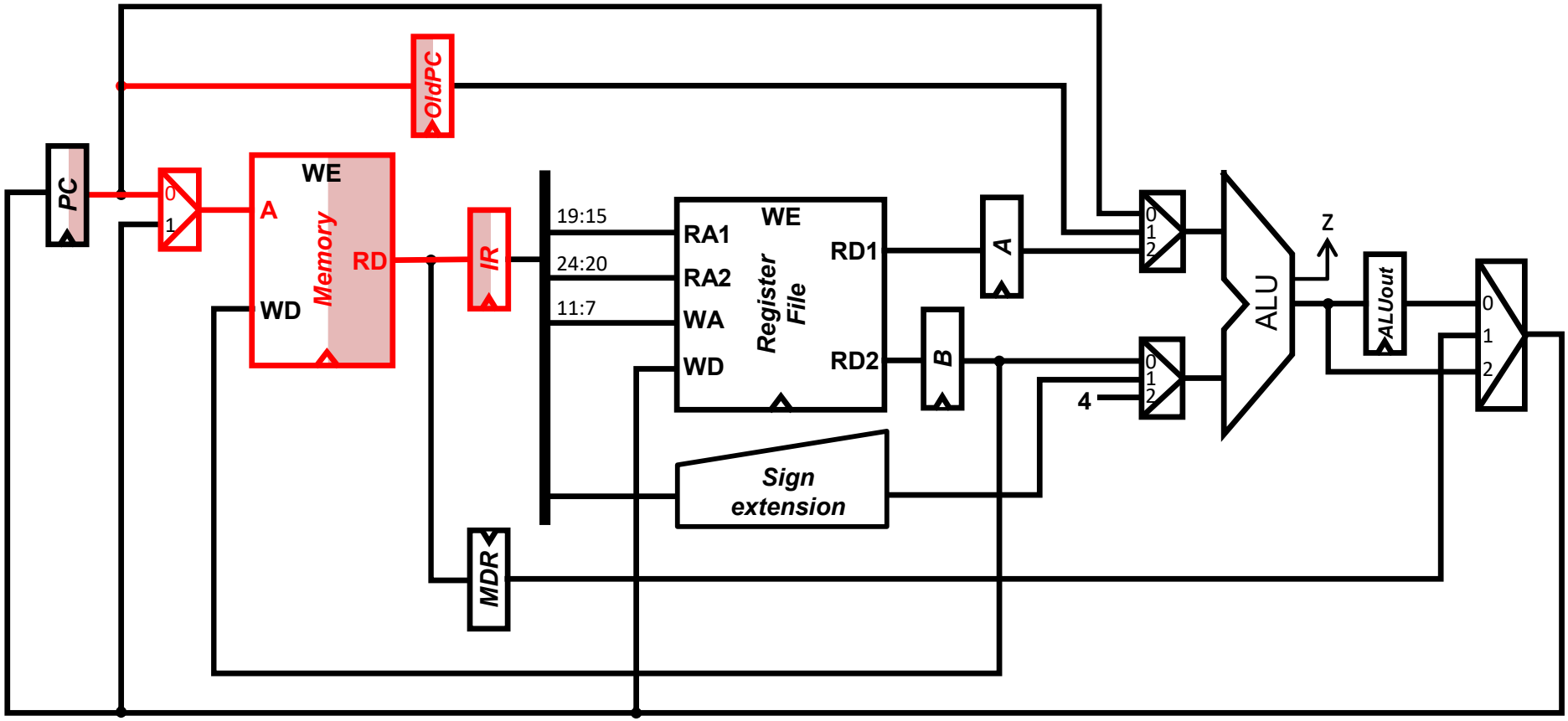
ASM diagram of the main FSM: `beq` instruction



Controller design

`jal` instruction: register transfers

1. IR ← Mem[PC], OldPC ← PC



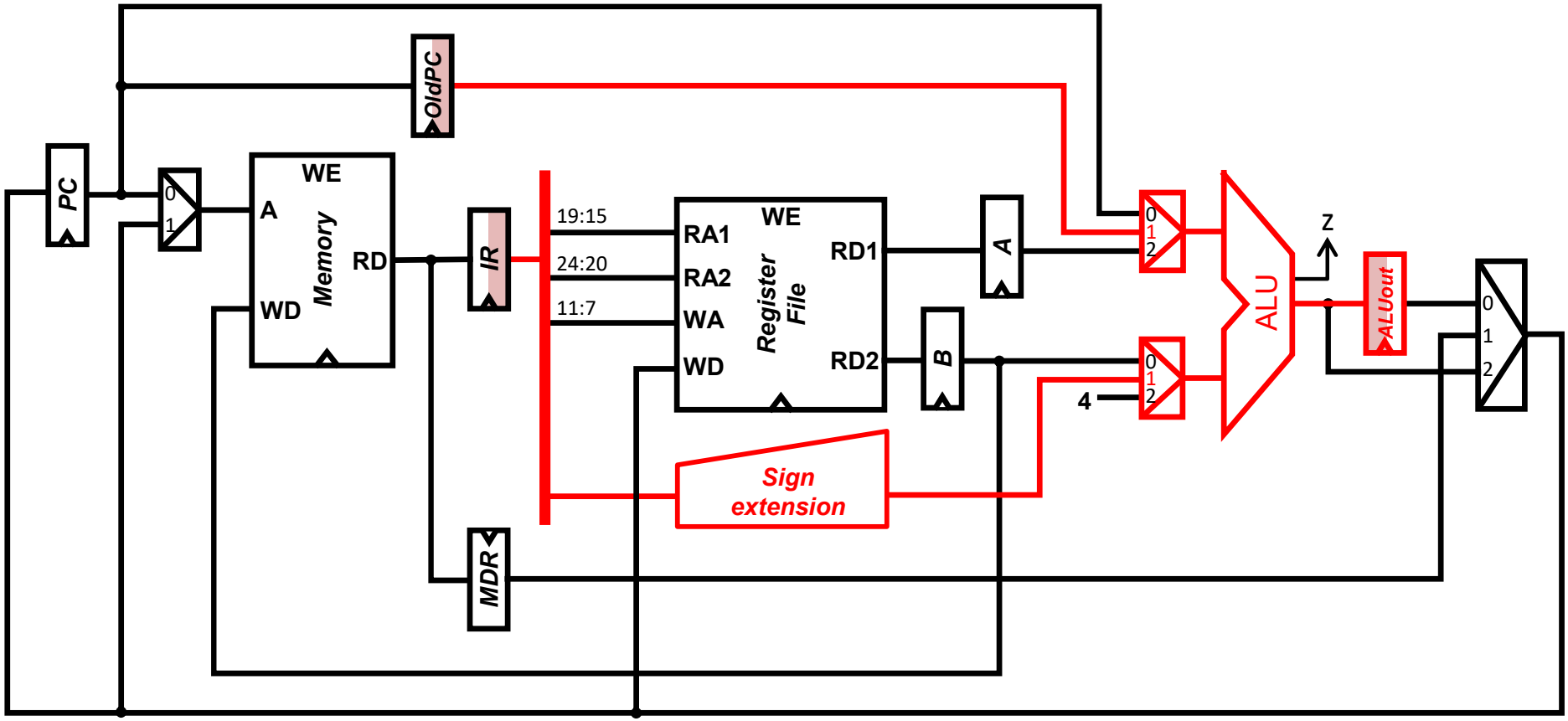
PC ← PC + sExt(imm), RF[rd] ← PC+4

Controller design

`jal` instruction: register transfers

1. IR ← Mem[PC], OldPC ← PC

2. ALUout ← oldPC + sExt(imm)



PC ← PC + sExt(imm), RF[rd] ← PC+4

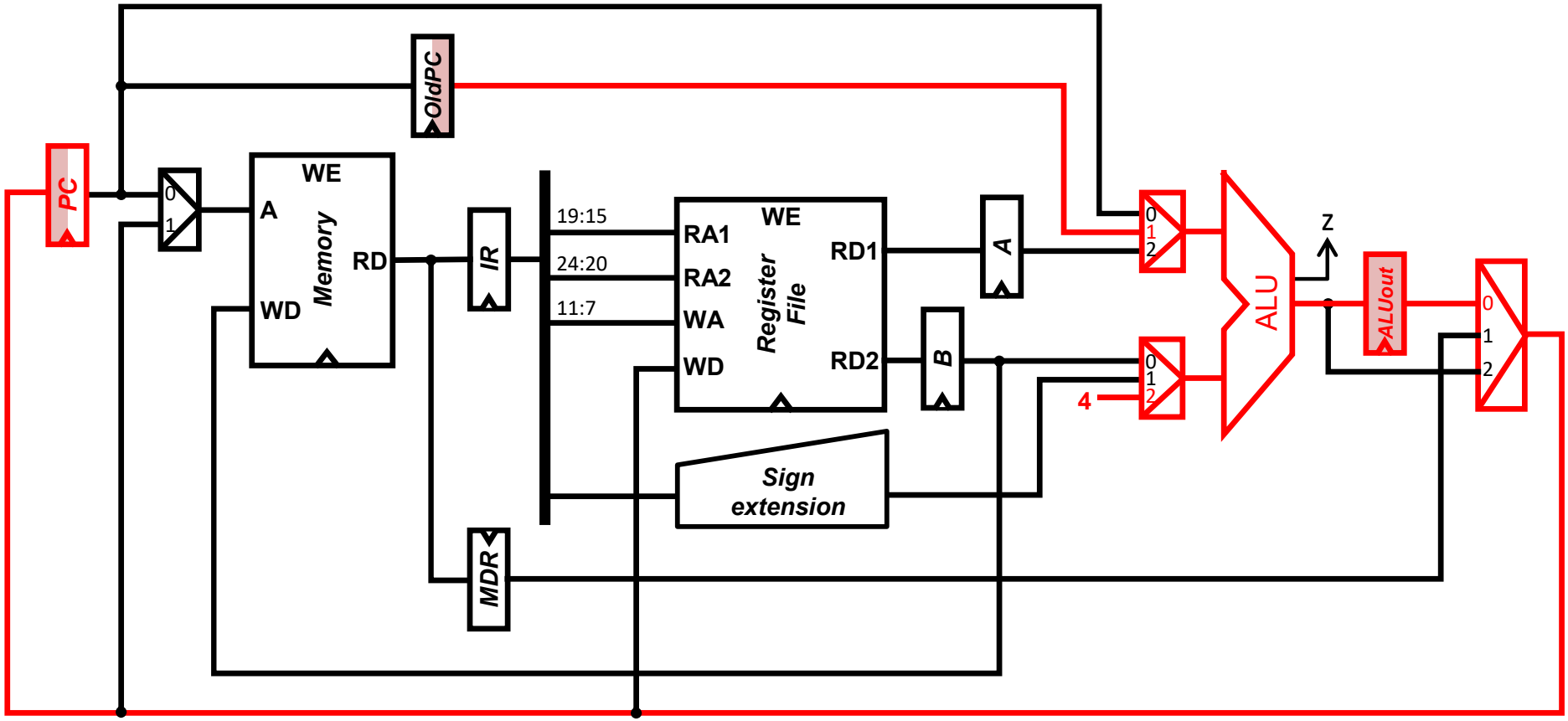
Controller design

`jal` instruction: register transfers

1. IR ← Mem[PC], OldPC ← PC

2. ALUout ← oldPC + sExt(imm)

3. PC ← ALUout, ALUout ← OldPC + 4



PC ← PC + sExt(imm), RF[rd] ← PC+4

Controller design

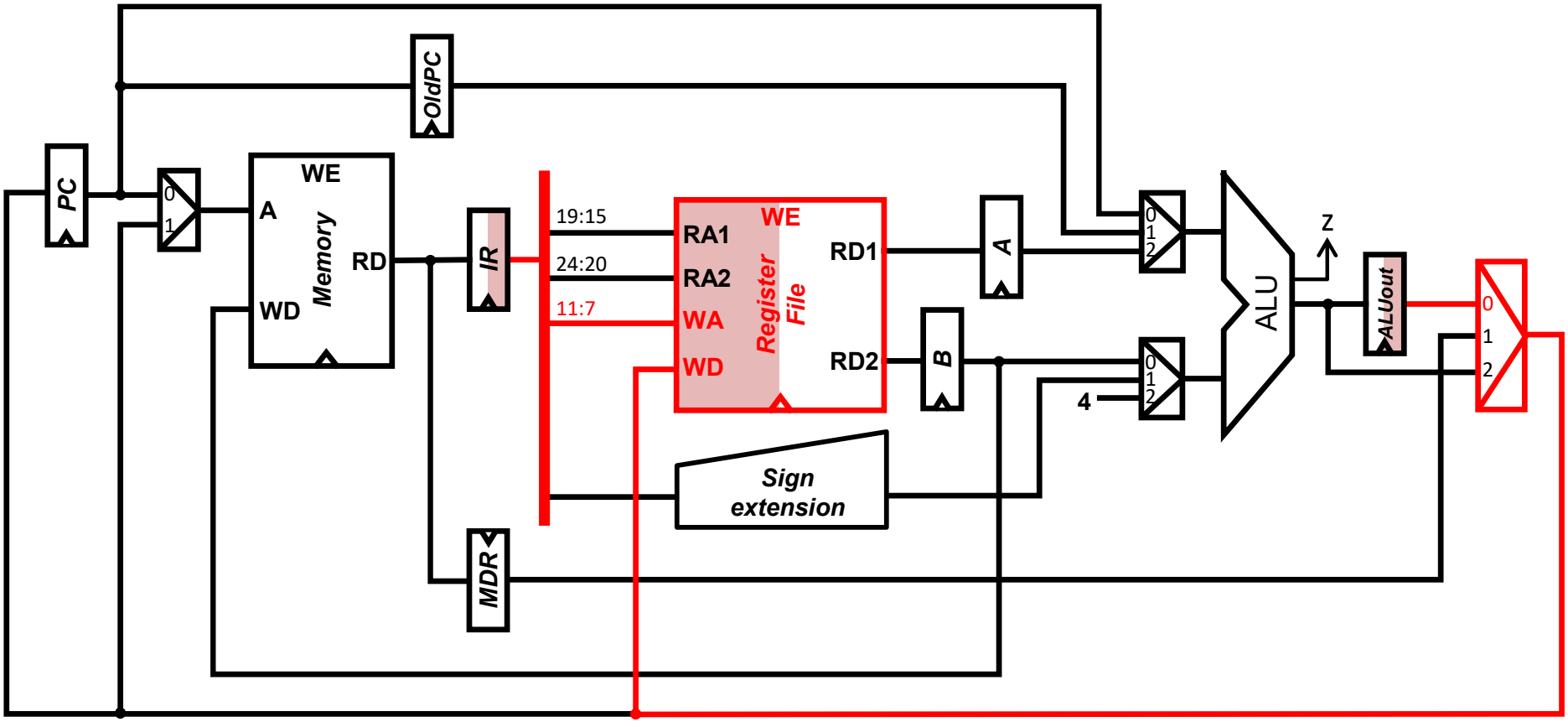
`jal` instruction: register transfers

1. IR ← Mem[PC], OldPC ← PC

2. ALUout ← oldPC + sExt(imm)

3. PC ← ALUout, ALUout ← OldPC + 4

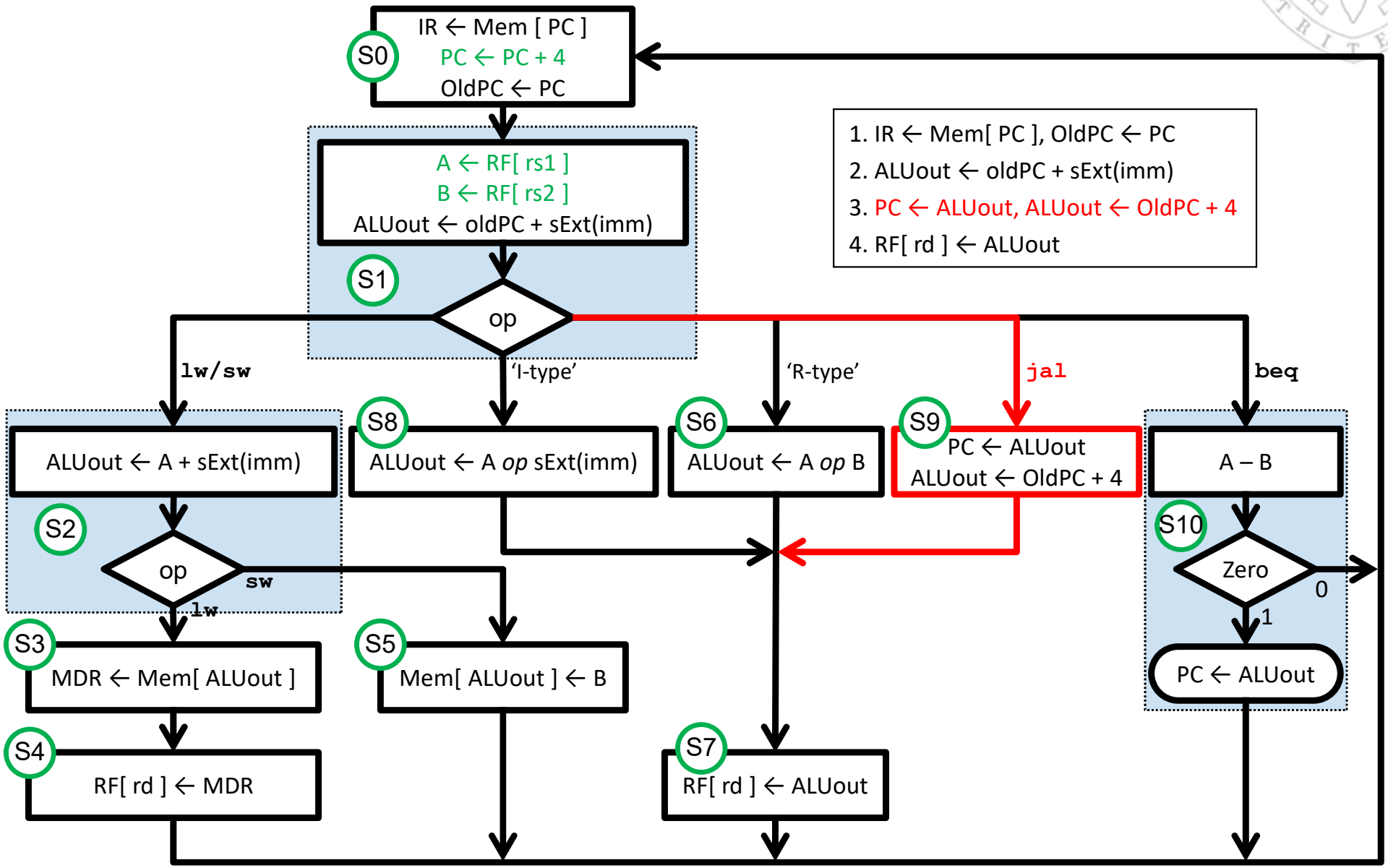
4. RF[rd] ← ALUout



PC ← PC + sExt(imm), RF[rd] ← PC+4

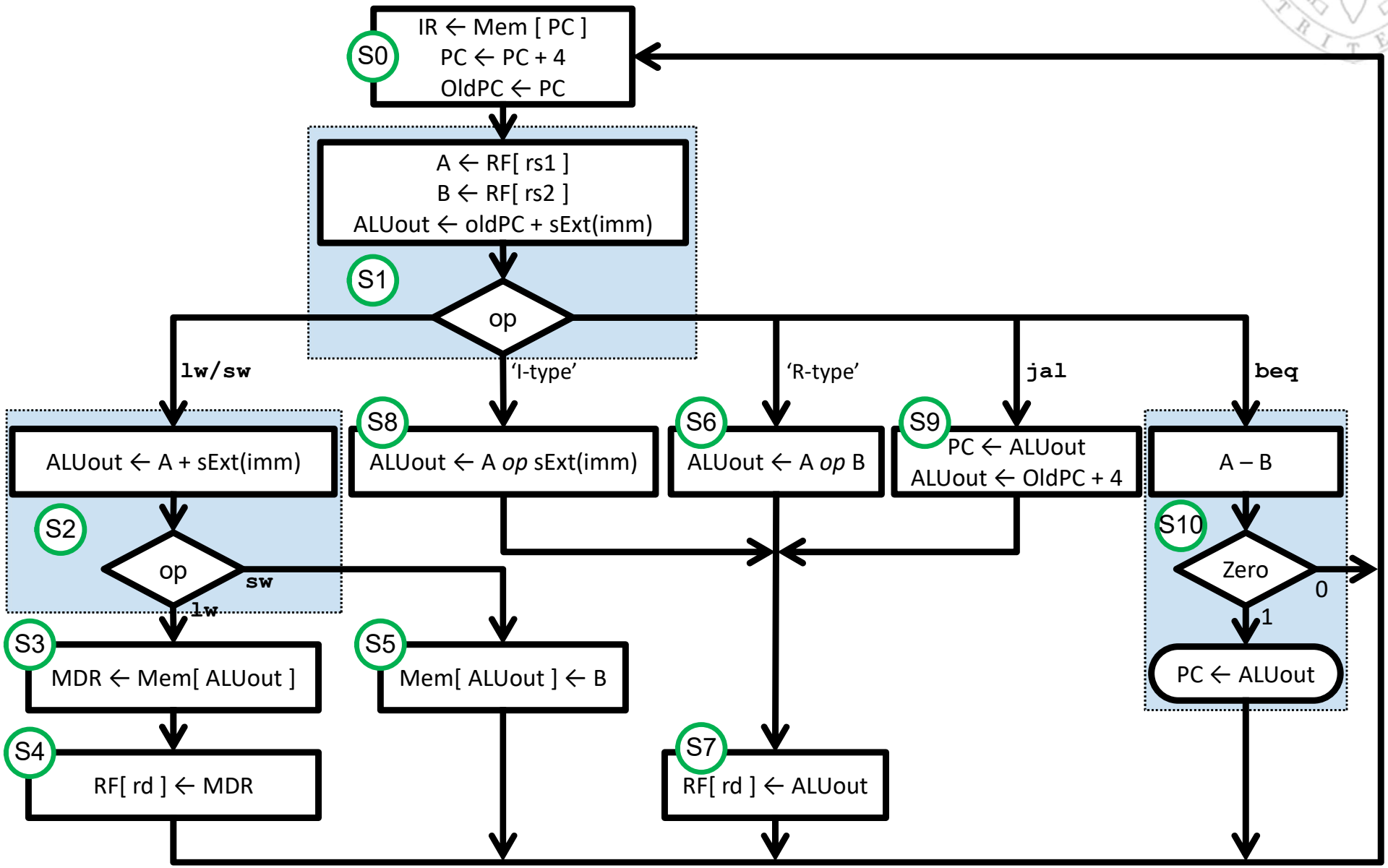
Controller design

ASM diagram of the main FSM: `jal` instruction



Controller design

Full ASM diagram of the main FSM



Controller design

Instruction cycle (i)

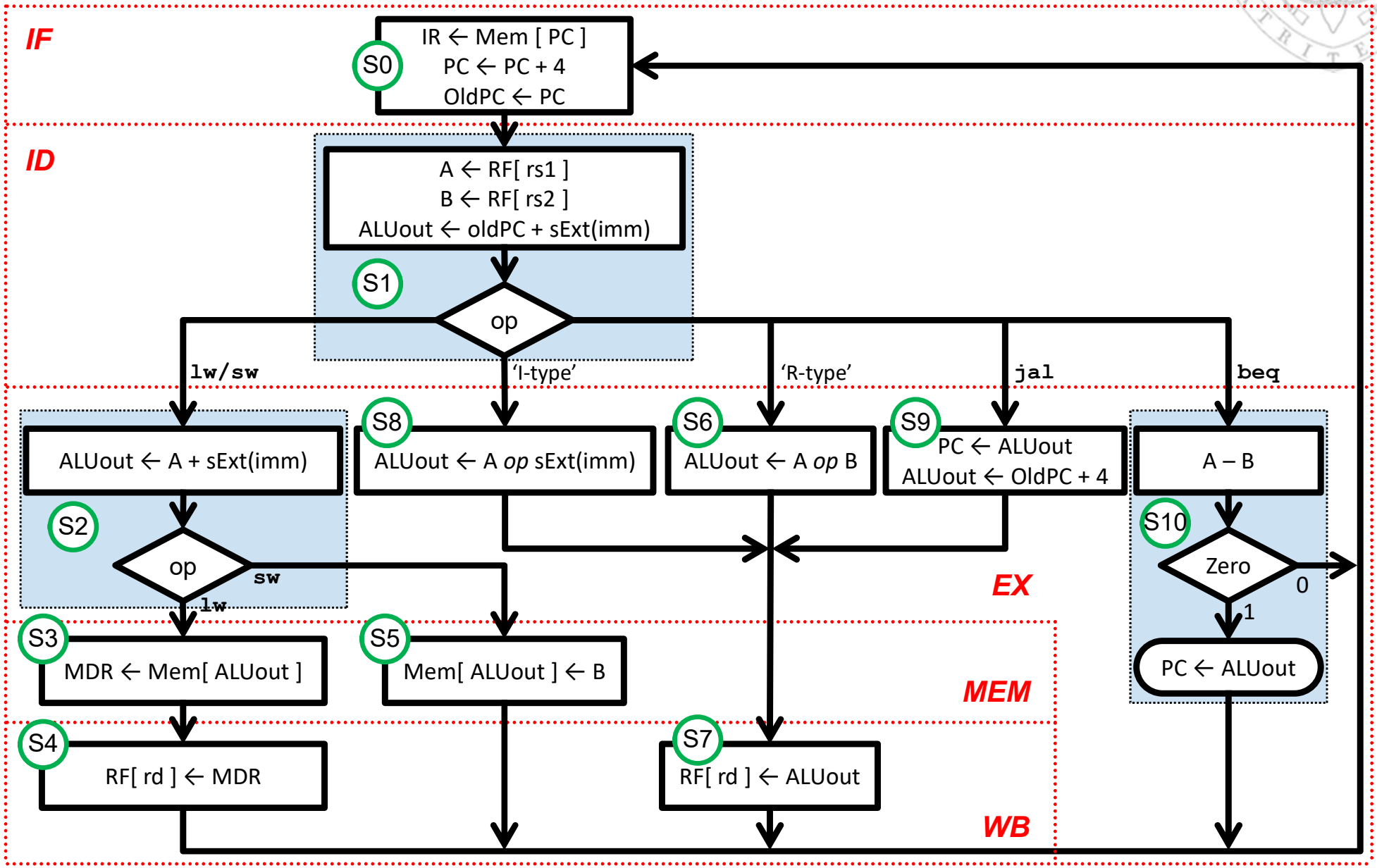
- Instructions are executed by repeating 5 stages cyclically:
  - Instruction Fetch — IF
  - Decode and operand reading — ID
  - Execution or address calculation — EX
  - Data Memory access — MEM
  - Write-Back — WB

- This is known as instruction cycle
  - In the multicycle processor, each stage is performed in 1 clock cycle.
  - Each instruction takes a different number of cycles, depending on the instruction cycle stages that each must perform:

Instruction	# of cycles
`lw`	5
`sw` / `jal` / arithmetic-logic	4
`beq`	3

Controller design

Instruction cycle (ii)



Controller design

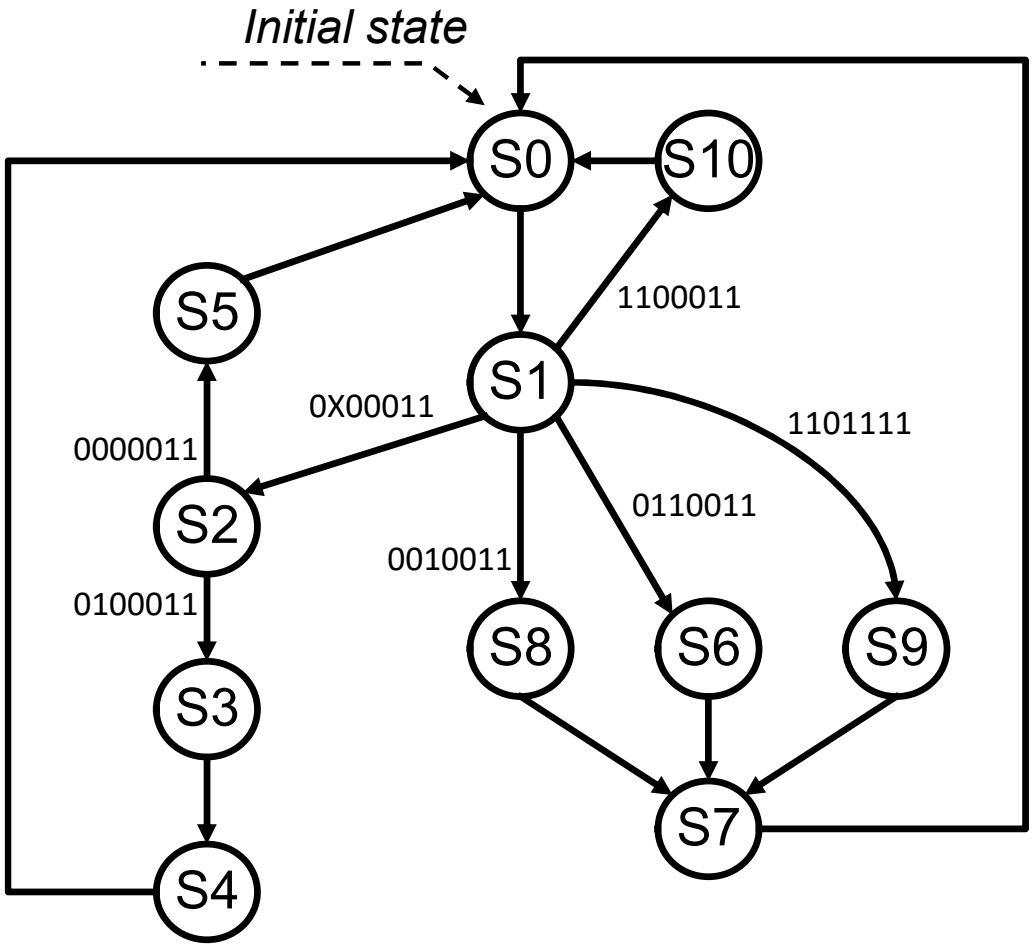
- ASM diagram = FSM state diagram.
  - Each ASM block is equivalent to an FSM state.
  - Transitions between blocks are equivalent to state transitions.
  - Each register transfer of an ASM block is translated into control signal values in the corresponding state.

- The register transfers performed in each state are translated by:
  - Setting (=1) the load signals of the destination registers.
  - Resetting (=0) the load signals of the rest of the registers.
  - Choosing the appropriate values for the selection signals of the MUX taking part in the transfers.
  - Assigning *don't care* to the selection signals of the rest of the MUX.

- Once the controller state diagram is designed, it is optimized in order to reduce the logic cost.

Controller design

Main FSM: state transition function

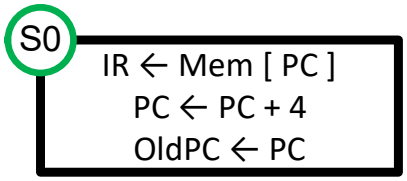


State transition function

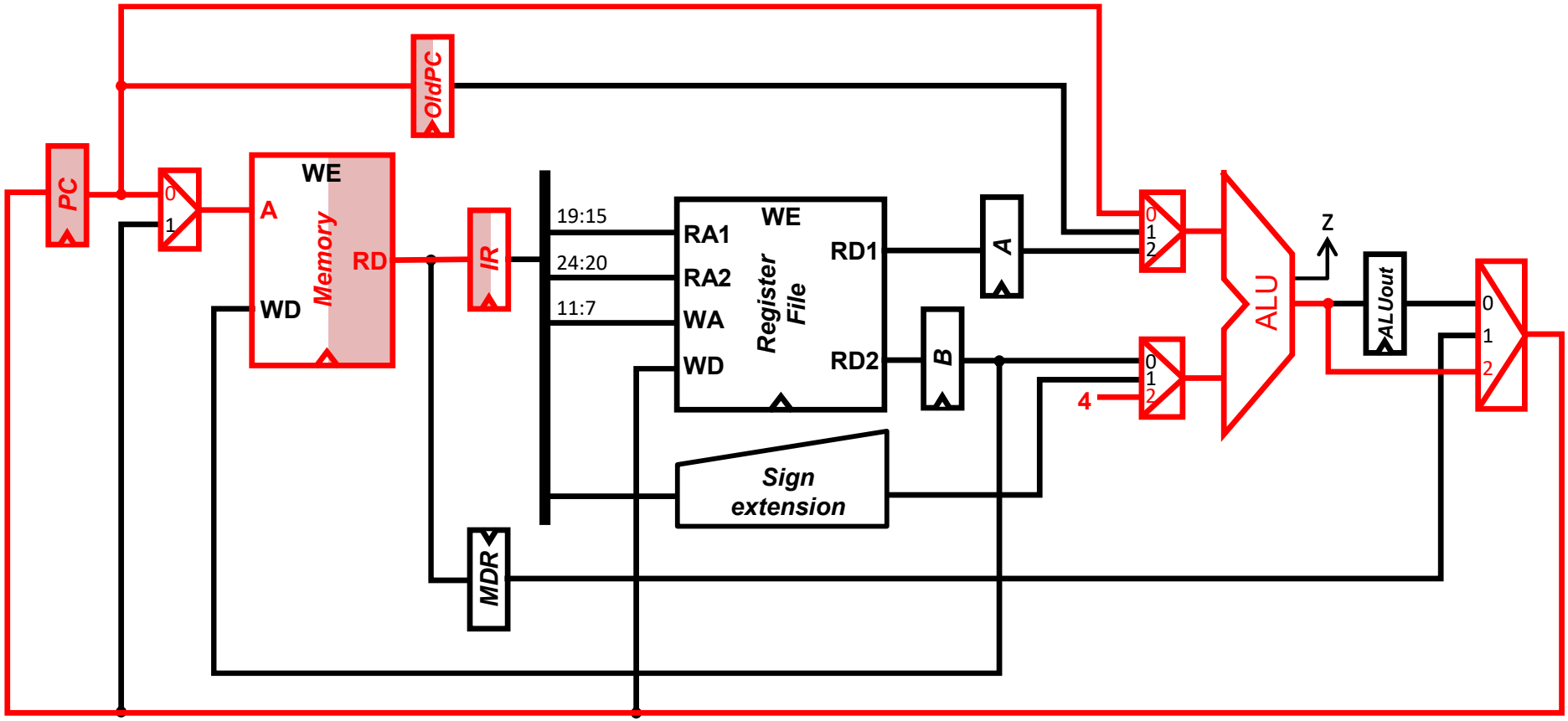
state	op	state'
S0	XXXXXXX	S1
S1	0X00011 (lw/sw)	S2
S1	0010011 (I-type)	S8
S1	0110011 (R-type)	S6
S1	1101111 (jal)	S9
S1	1100011 (beq)	S10
S2	0000011 (lw)	S3
S2	0100011 (sw)	S5
S3	XXXXXXX	S4
S4	XXXXXXX	S0
S5	XXXXXXX	S0
S6	XXXXXXX	S7
S7	XXXXXXX	S0
S8	XXXXXXX	S7
S9	XXXXXXX	S7
S10	XXXXXXX	S0

Controller design

Main FSM: output function

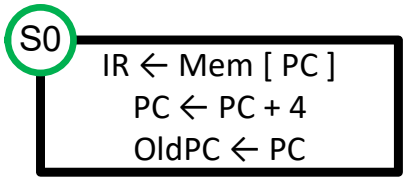


state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0															

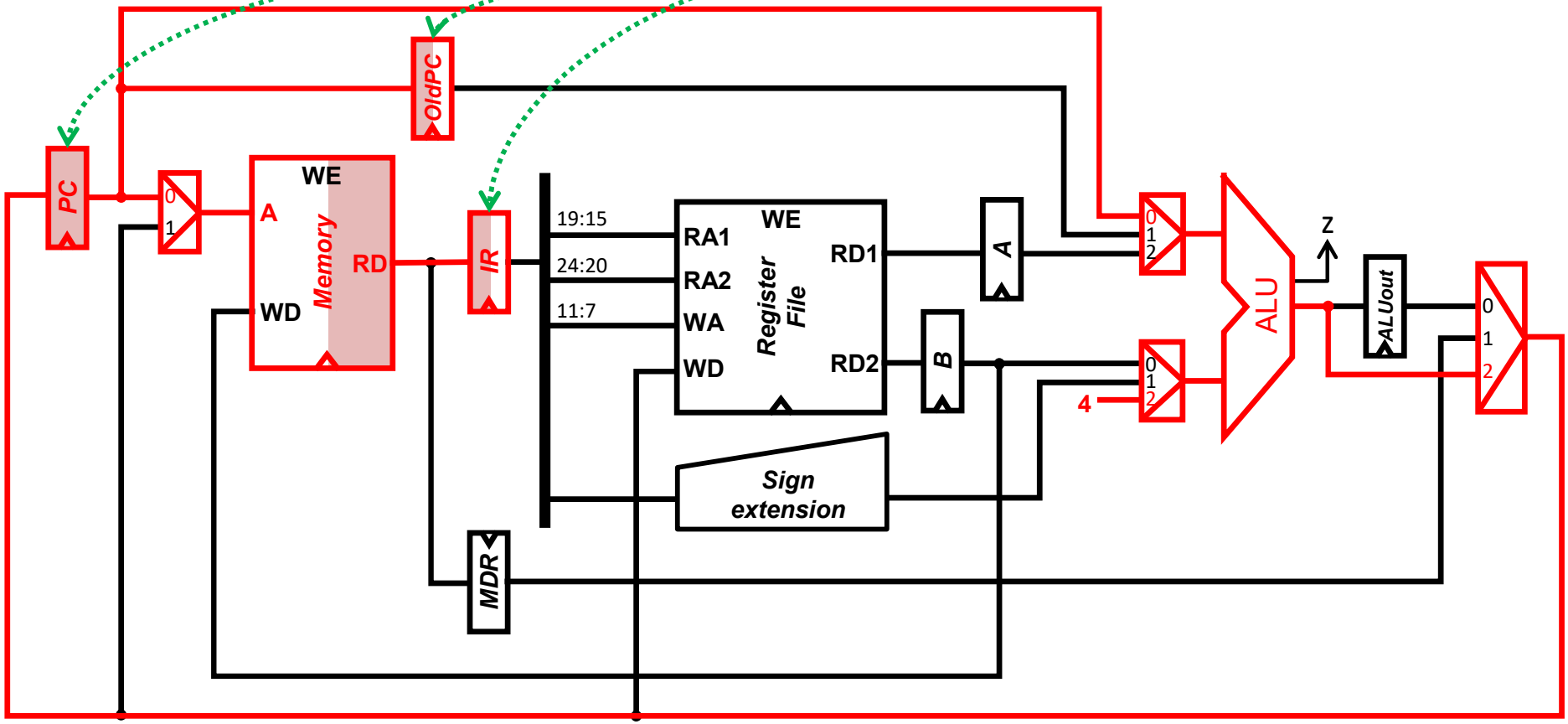


Controller design

Main FSM: output function

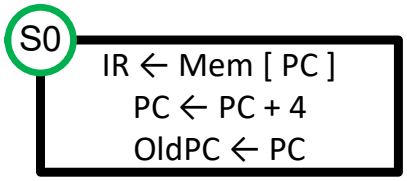


state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1			1	1									

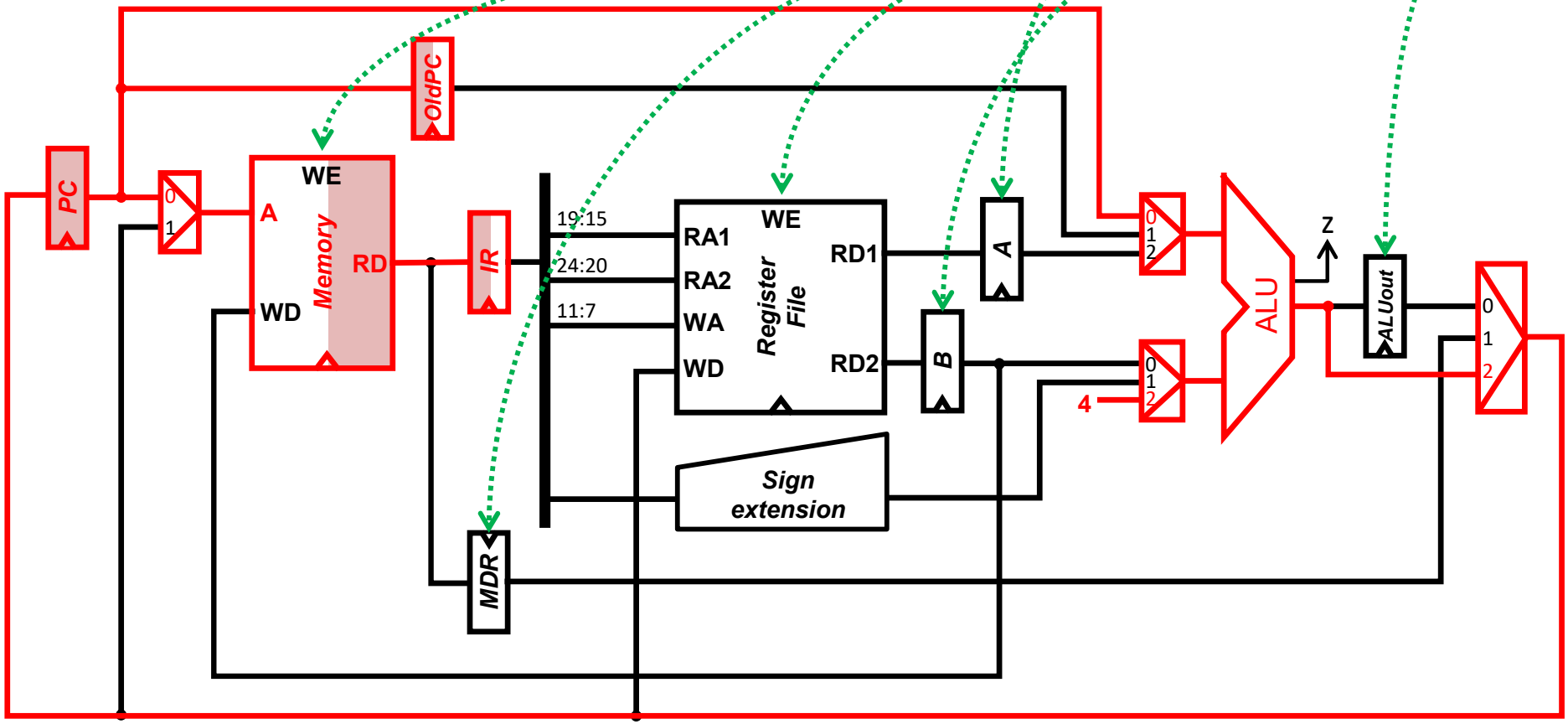


Controller design

Main FSM: output function

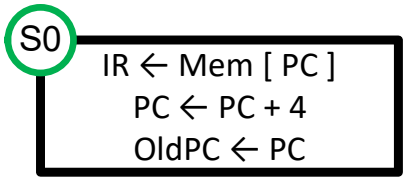


state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1		0	1	1	0	0	0	0				0	

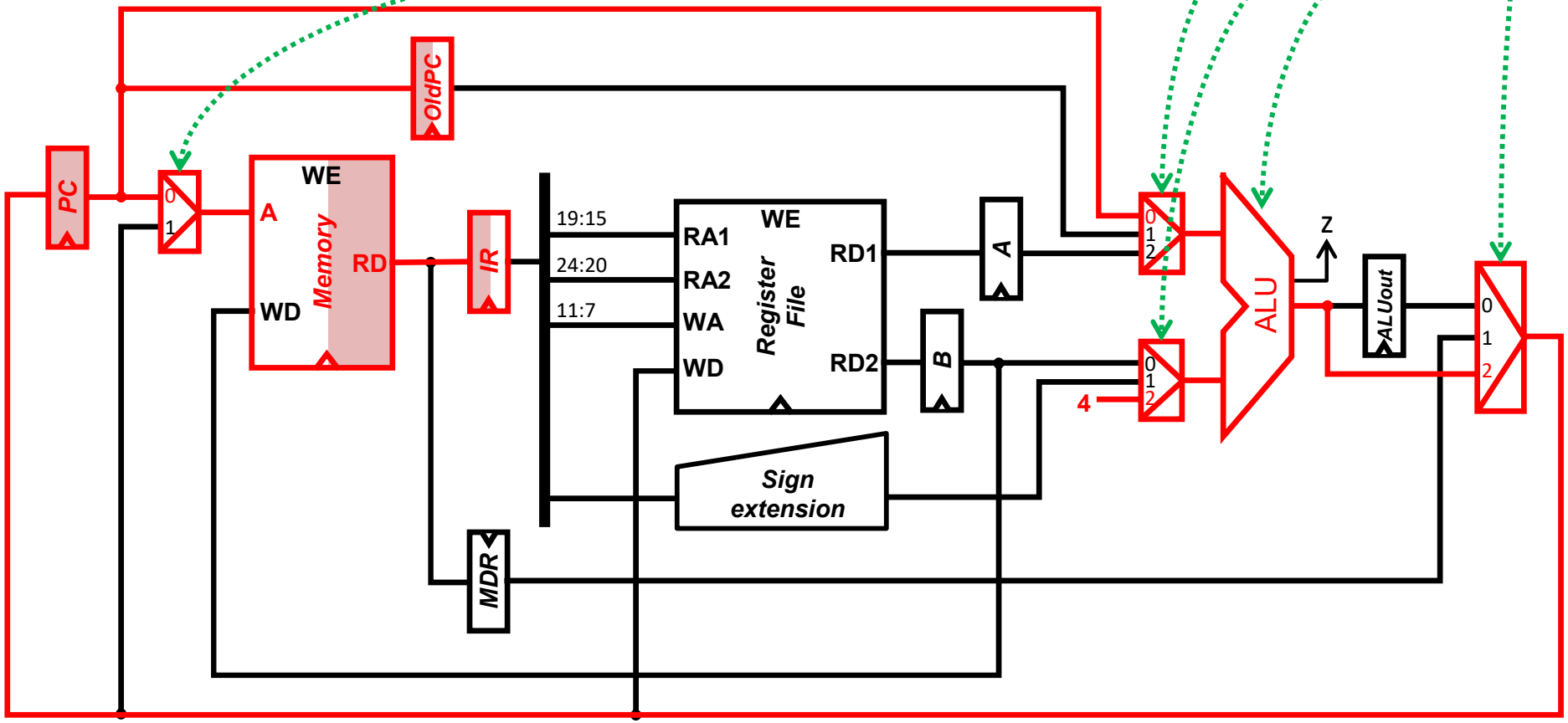


Controller design

Main FSM: output function



state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10

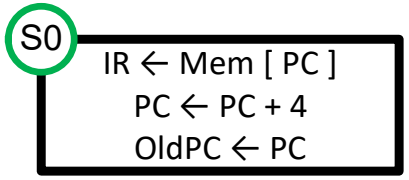


Controller design

Main FSM: output function

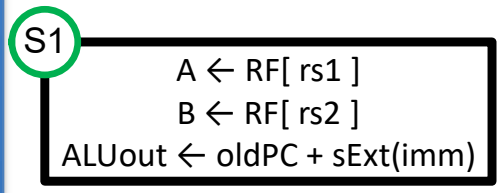
Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1															
S2															
S3															
S4															
S5															
S6															
S7															
S8															
S9															
S10															

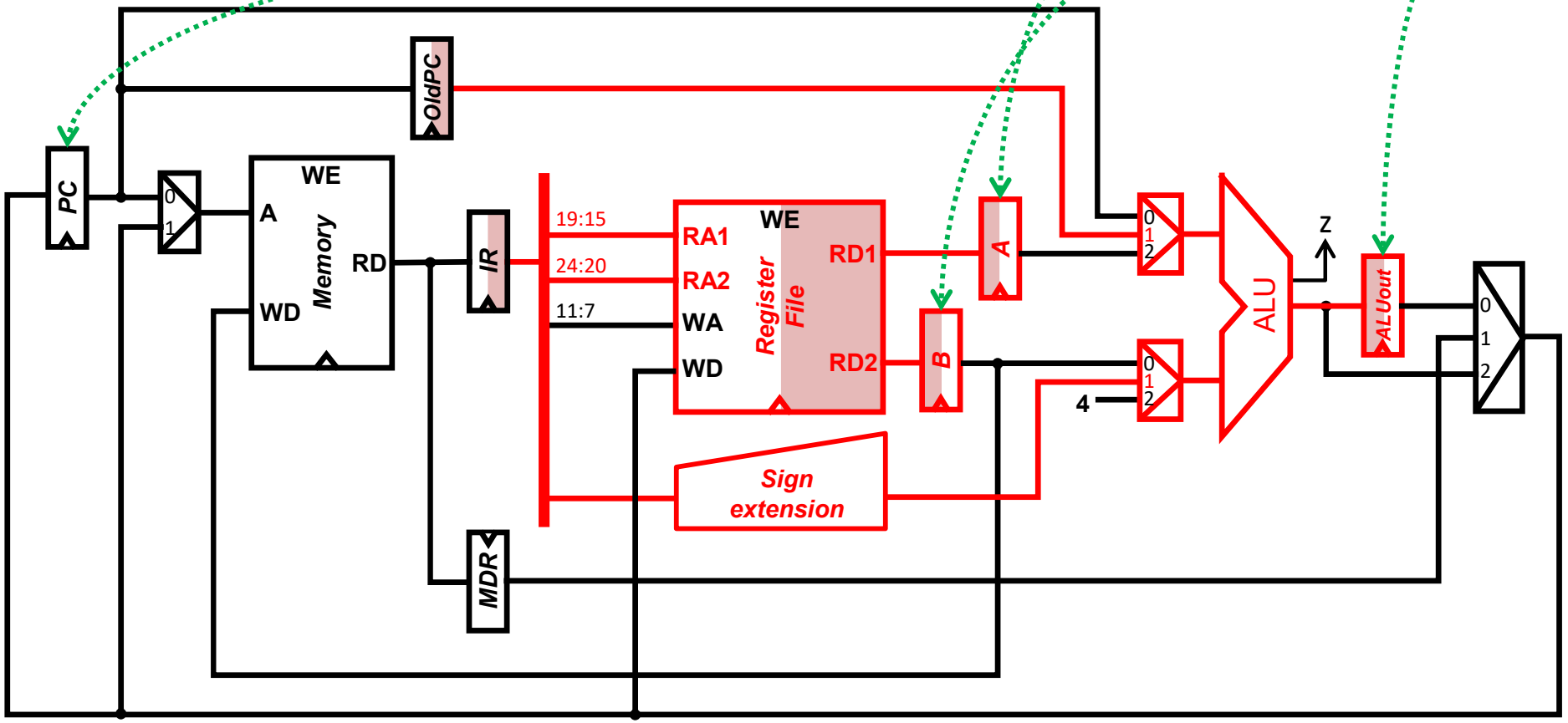


Controller design

Main FSM: output function

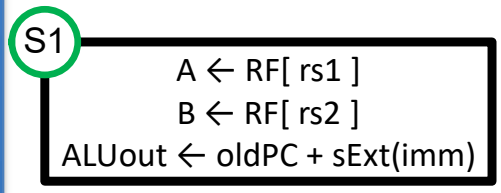


state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S1	0	0							1	1				1	

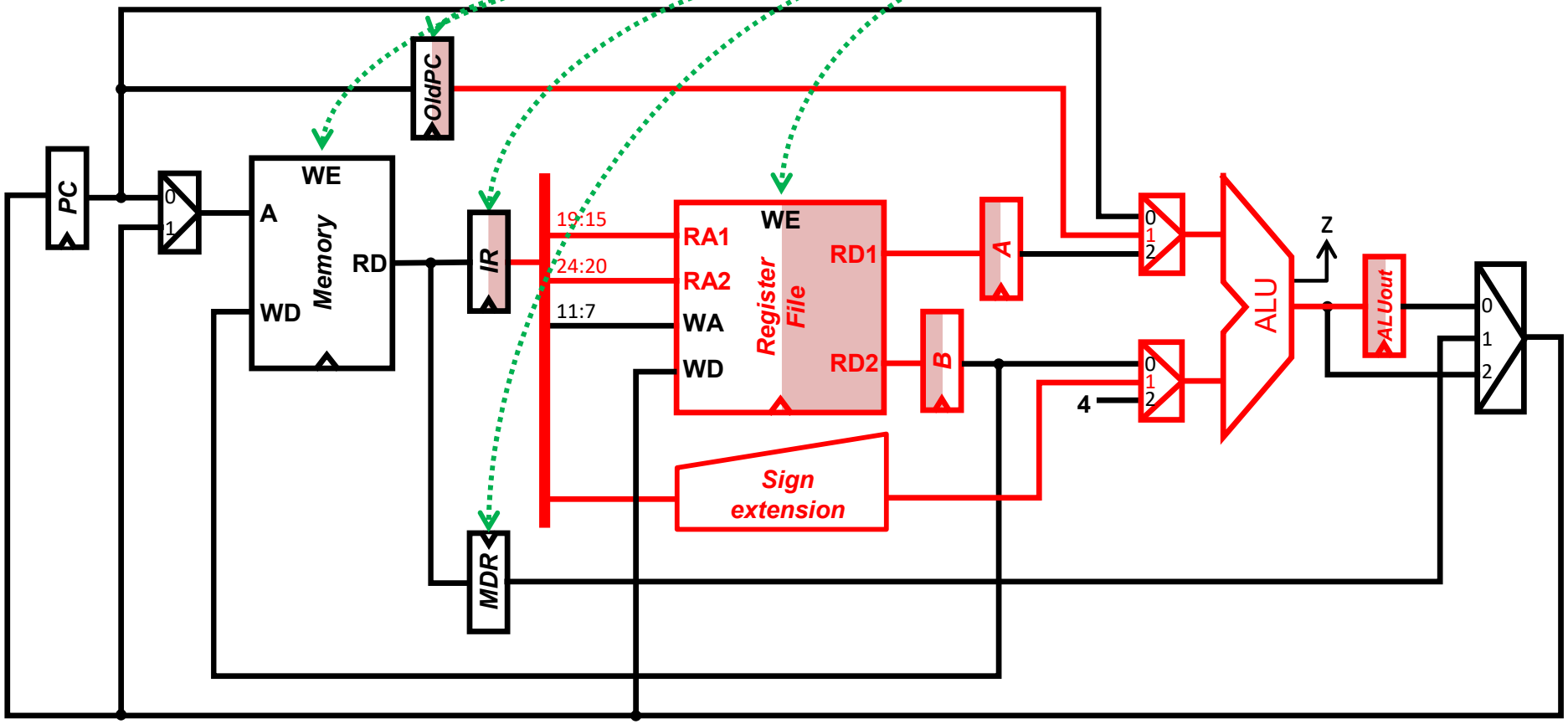


Controller design

Main FSM: output function

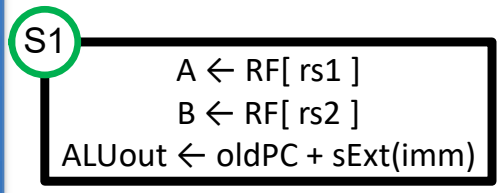


state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S1	0	0		0	0	0	0	0	1	1				1	

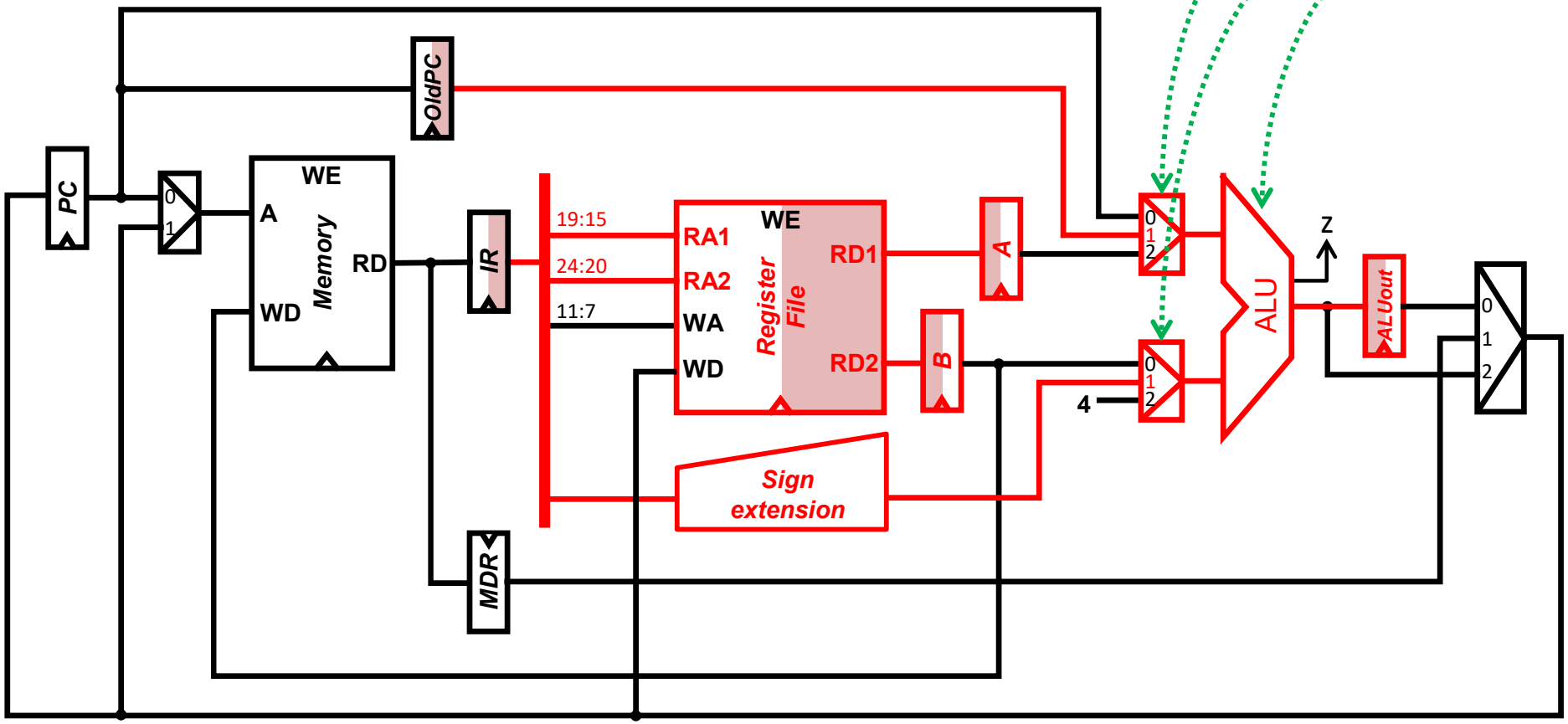


Controller design

Main FSM: output function

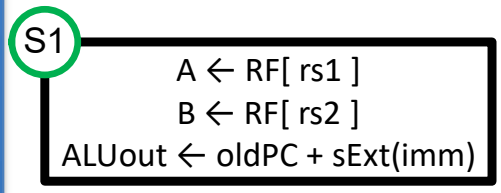


state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S1	0	0		0	0	0	0	0	1	1	01	01	00	1	

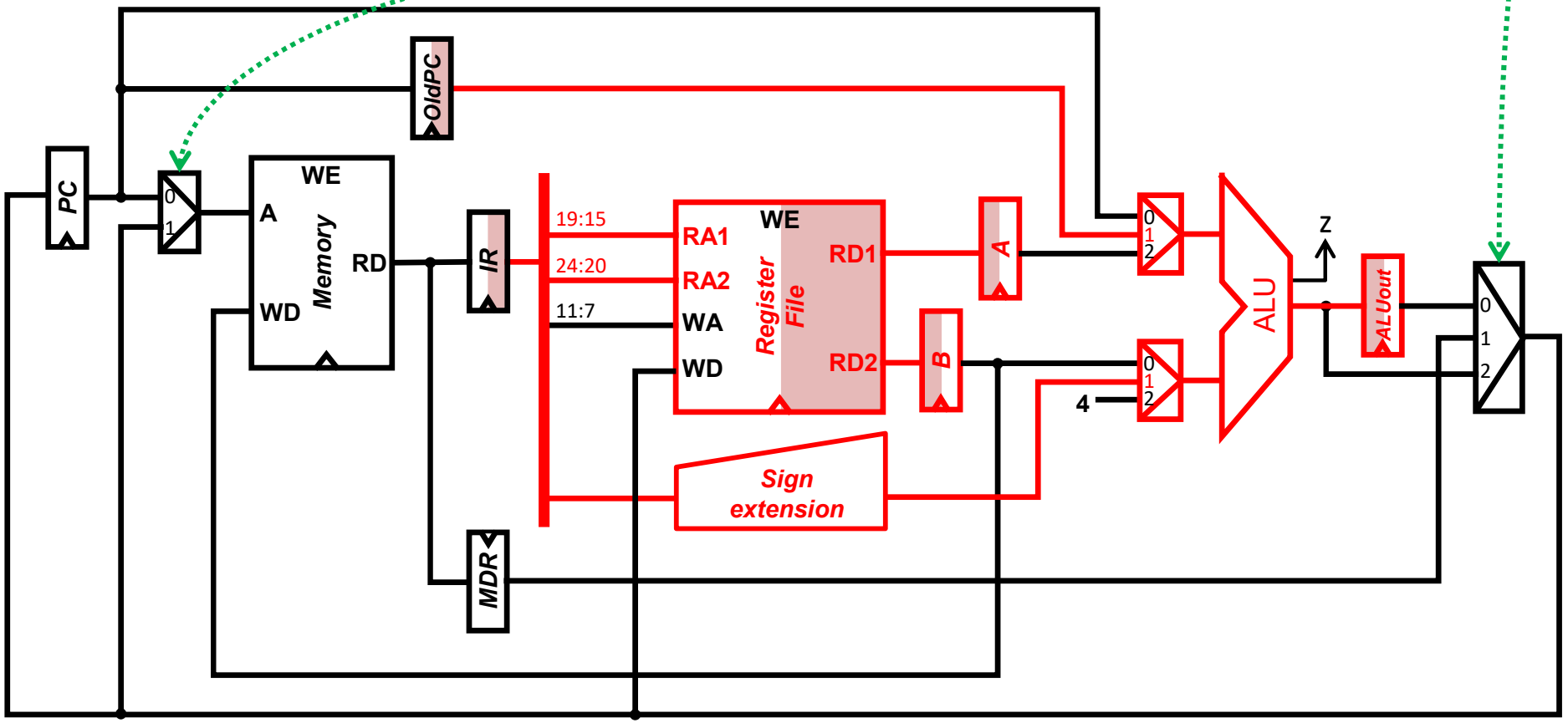


Controller design

Main FSM: output function



state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–

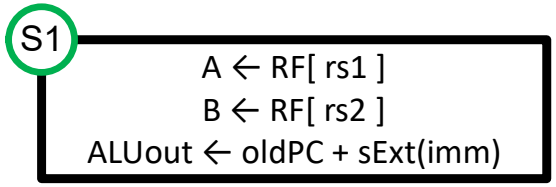


Controller design

Main FSM: output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2															
S3															
S4															
S5															
S6															
S7															
S8															
S9															
S10															

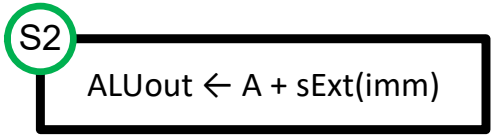


Controller design

Main FSM: output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	0	0	0	0	0	10	01	00	1	–
S3															
S4															
S5															
S6															
S7															
S8															
S9															
S10															



Controller design

Main FSM: output function

Output function

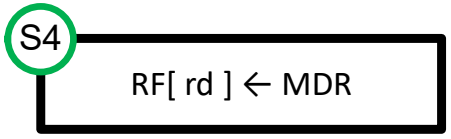
state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	0	0	0	0	0	10	01	00	1	–
S3	0	0	1	0	0	0	1	0	0	0	–	–	–	0	00
S4															
S5															
S6															
S7															
S8															
S9															
S10															

Controller design

Main FSM: output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	0	0	0	0	0	10	01	00	1	–
S3	0	0	1	0	0	0	1	0	0	0	–	–	–	0	00
S4	0	0	–	0	0	0	0	1	0	0	–	–	–	0	01
S5															
S6															
S7															
S8															
S9															
S10															

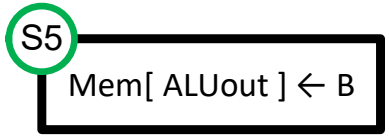


Controller design

Main FSM: output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	0	0	0	0	0	10	01	00	1	–
S3	0	0	1	0	0	0	1	0	0	0	–	–	–	0	00
S4	0	0	–	0	0	0	0	1	0	0	–	–	–	0	01
S5	0	0	1	1	0	0	0	0	0	0	–	–	–	0	00
S6															
S7															
S8															
S9															
S10															

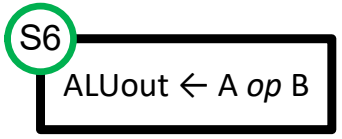


Controller design

Main FSM: output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	0	0	0	0	0	10	01	00	1	–
S3	0	0	1	0	0	0	1	0	0	0	–	–	–	0	00
S4	0	0	–	0	0	0	0	1	0	0	–	–	–	0	01
S5	0	0	1	1	0	0	0	0	0	0	–	–	–	0	00
S6	0	0	–	0	0	0	0	0	0	0	10	00	10	1	–
S7															
S8															
S9															
S10															

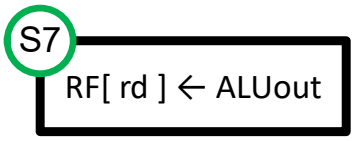


Controller design

Main FSM: output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	0	0	0	0	0	10	01	00	1	–
S3	0	0	1	0	0	0	1	0	0	0	–	–	–	0	00
S4	0	0	–	0	0	0	0	1	0	0	–	–	–	0	01
S5	0	0	1	1	0	0	0	0	0	0	–	–	–	0	00
S6	0	0	–	0	0	0	0	0	0	0	10	00	10	1	–
S7	0	0	–	0	0	0	0	1	0	0	–	–	–	0	00
S8															
S9															
S10															

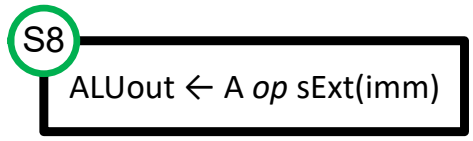


Controller design

Main FSM: output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	0	0	0	0	0	10	01	00	1	–
S3	0	0	1	0	0	0	1	0	0	0	–	–	–	0	00
S4	0	0	–	0	0	0	0	1	0	0	–	–	–	0	01
S5	0	0	1	1	0	0	0	0	0	0	–	–	–	0	00
S6	0	0	–	0	0	0	0	0	0	0	10	00	10	1	–
S7	0	0	–	0	0	0	0	1	0	0	–	–	–	0	00
S8	0	0	–	0	0	0	0	0	0	0	10	01	10	1	–
S9															
S10															

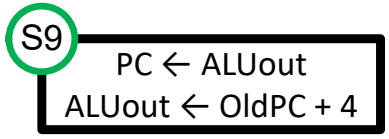


Controller design

Main FSM: output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	0	0	0	0	0	10	01	00	1	–
S3	0	0	1	0	0	0	1	0	0	0	–	–	–	0	00
S4	0	0	–	0	0	0	0	1	0	0	–	–	–	0	01
S5	0	0	1	1	0	0	0	0	0	0	–	–	–	0	00
S6	0	0	–	0	0	0	0	0	0	0	10	00	10	1	–
S7	0	0	–	0	0	0	0	1	0	0	–	–	–	0	00
S8	0	0	–	0	0	0	0	0	0	0	10	01	10	1	–
S9	0	1	–	0	0	0	0	0	0	0	01	10	00	1	00
S10															

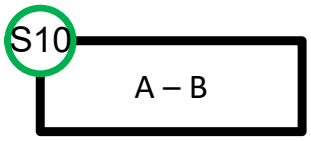


Controller design

Main FSM: output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	0	0	0	0	0	10	01	00	1	–
S3	0	0	1	0	0	0	1	0	0	0	–	–	–	0	00
S4	0	0	–	0	0	0	0	1	0	0	–	–	–	0	01
S5	0	0	1	1	0	0	0	0	0	0	–	–	–	0	00
S6	0	0	–	0	0	0	0	0	0	0	10	00	10	1	–
S7	0	0	–	0	0	0	0	1	0	0	–	–	–	0	00
S8	0	0	–	0	0	0	0	0	0	0	10	01	10	1	–
S9	0	1	–	0	0	0	0	0	0	0	01	10	00	1	00
S10	1	0	–	0	0	0	0	0	0	0	10	00	01	0	00



Controller design

Main FSM: output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	0	0	0	0	0	10	01	00	1	–
S3	0	0	1	0	0	0	1	0	0	0	–	–	–	0	00
S4	0	0	–	0	0	0	0	1	0	0	–	–	–	0	01
S5	0	0	1	1	0	0	0	0	0	0	–	–	–	0	00
S6	0	0	–	0	0	0	0	0	0	0	10	00	10	1	–
S7	0	0	–	0	0	0	0	1	0	0	–	–	–	0	00
S8	0	0	–	0	0	0	0	0	0	0	10	01	10	1	–
S9	0	1	–	0	0	0	0	0	0	0	01	10	00	1	00
S10	1	0	–	0	0	0	0	0	0	0	10	00	01	0	00

Controller design

1st optimization

identical

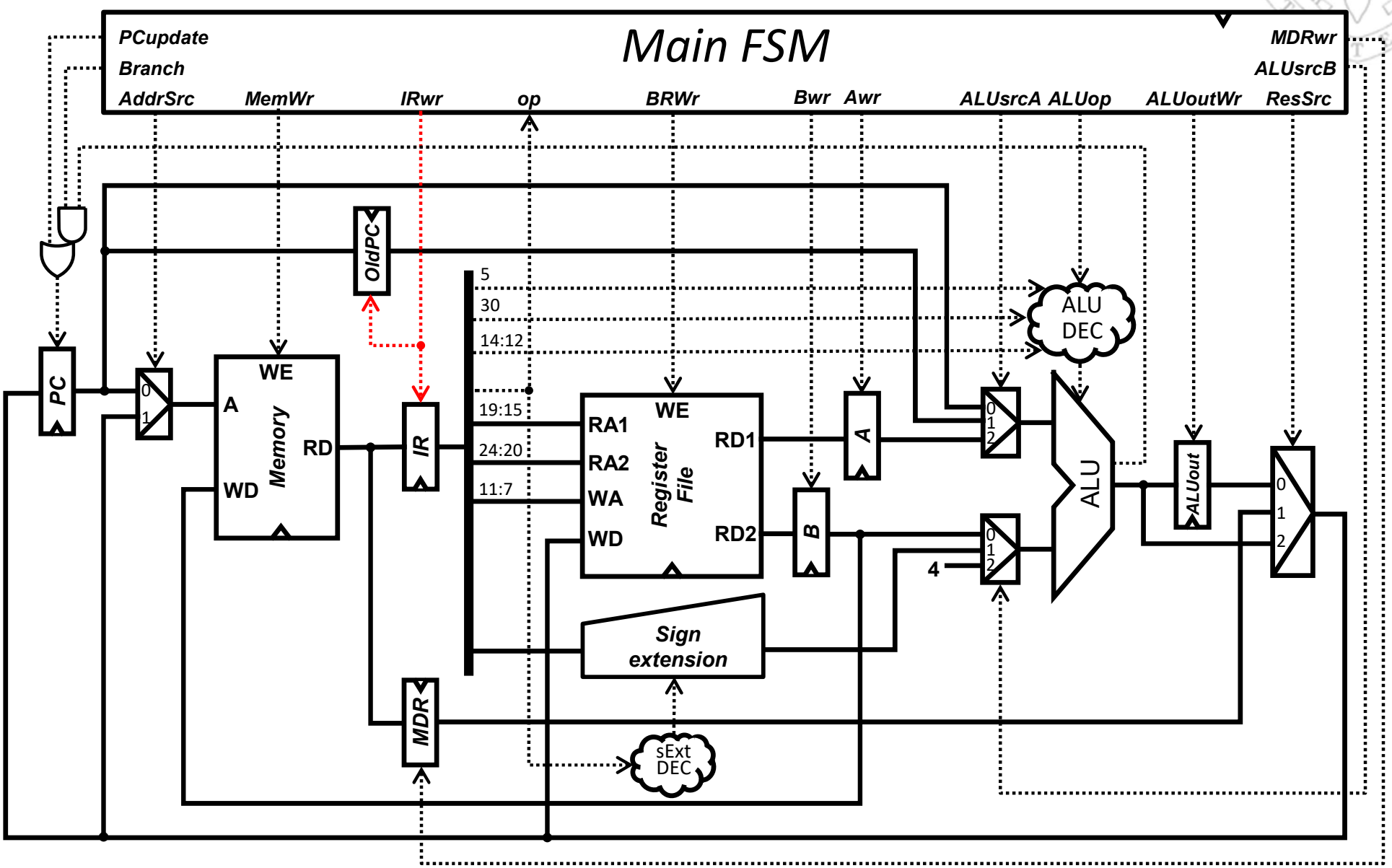
Output function

state	Branch	PCupdate	AddrSrc	MemWr	OldPCwr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	1	0	0	0	0	00	10	00	0	10
S1	0	0	–	0	0	0	0	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	0	0	0	0	0	10	01	00	1	–
S3	0	0	1	0	0	0	1	0	0	0	–	–	–	0	00
S4	0	0	–	0	0	0	0	1	0	0	–	–	–	0	01
S5	0	0	1	1	0	0	0	0	0	0	–	–	–	0	00
S6	0	0	–	0	0	0	0	0	0	0	10	00	10	1	–
S7	0	0	–	0	0	0	0	1	0	0	–	–	–	0	00
S8	0	0	–	0	0	0	0	0	0	0	10	01	10	1	–
S9	0	1	–	0	0	0	0	0	0	0	01	10	00	1	00
S10	1	0	–	0	0	0	0	0	0	0	10	00	01	0	00

- The OldPCwr and IRwr signals have the same truth table
  - Both control points can be controlled with just one of them
  - The other one can be removed from the controller

Controller design

1st optimization

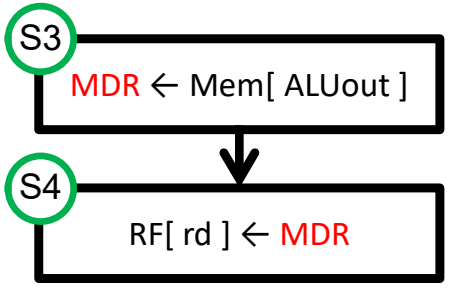


Controller design

2nd optimization

- Data life is the cycle interval between the load of a data into a register and its last use.

- Data stored in some of the auxiliary registers of the multicycle processor has a very short life:
  - The MDR auxiliary register is only used in the execution of **`lw`** instructions.
  - In state S3, MDR is loaded with a data read from the memory, and such data is consumed in S4 (the state after S3) and stored in the Register File.
  - Once the data is stored, it is not longer needed by that instruction.
  - Therefore, whatever is done with MDR is irrelevant in the rest of the states.



Controller design

2nd optimization

Output function

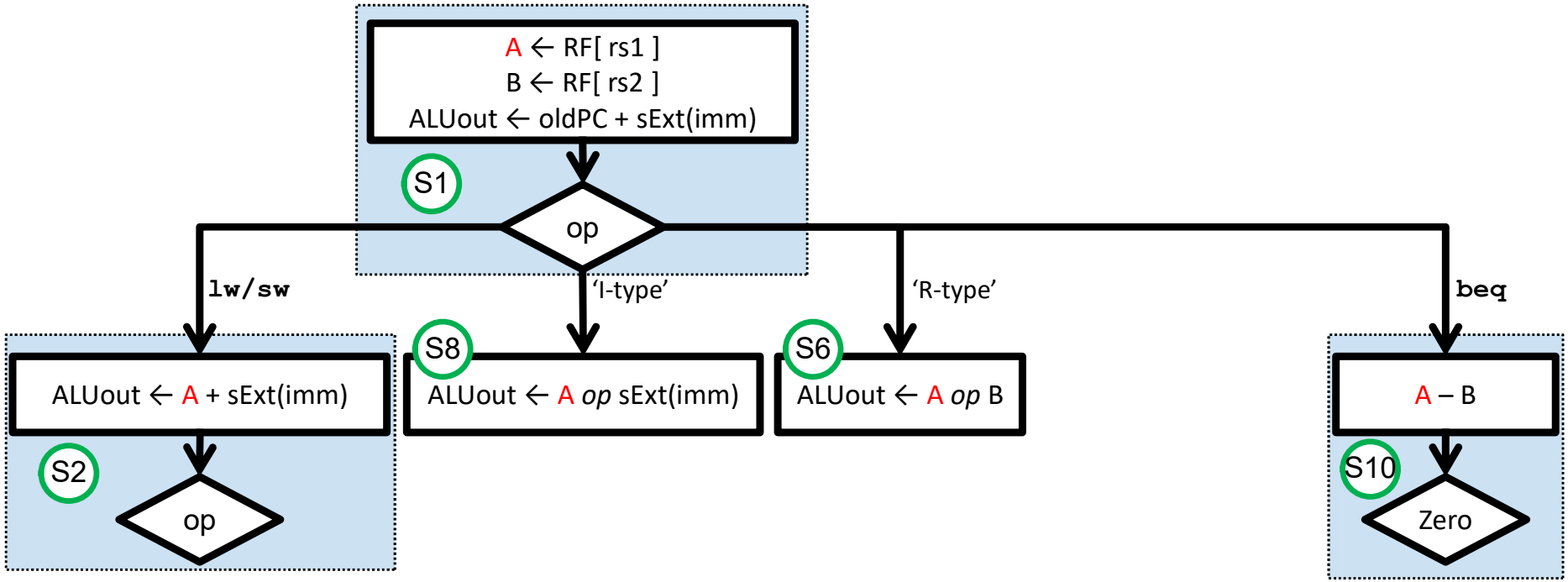
MDRwr	state	Branch	PCupdate	AddrSrc	MemWr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
0	S0	0	1	0	0	1	–	0	0	0	00	10	00	0	10
0	S1	0	0	–	0	0	–	0	1	1	01	01	00	1	–
0	S2	0	0	–	0	0	–	0	0	0	10	01	00	1	–
1	S3	0	0	1	0	0	1	0	0	0	–	–	–	0	00
0	S4	0	0	–	0	0	–	1	0	0	–	–	–	0	01
0	S5	0	0	1	1	0	–	0	0	0	–	–	–	0	00
0	S6	0	0	–	0	0	–	0	0	0	10	00	10	1	–
0	S7	0	0	–	0	0	–	1	0	0	–	–	–	0	00
0	S8	0	0	–	0	0	–	0	0	0	10	01	10	1	–
0	S9	0	1	–	0	0	–	0	0	0	01	10	00	1	00
0	S10	1	0	–	0	0	–	0	0	0	10	00	01	0	00

- Since MDR must be loaded in state S3, the MDRwr signal must be 1 in S3
  - Its value is irrelevant in the rest of the states

Controller design

2nd optimization

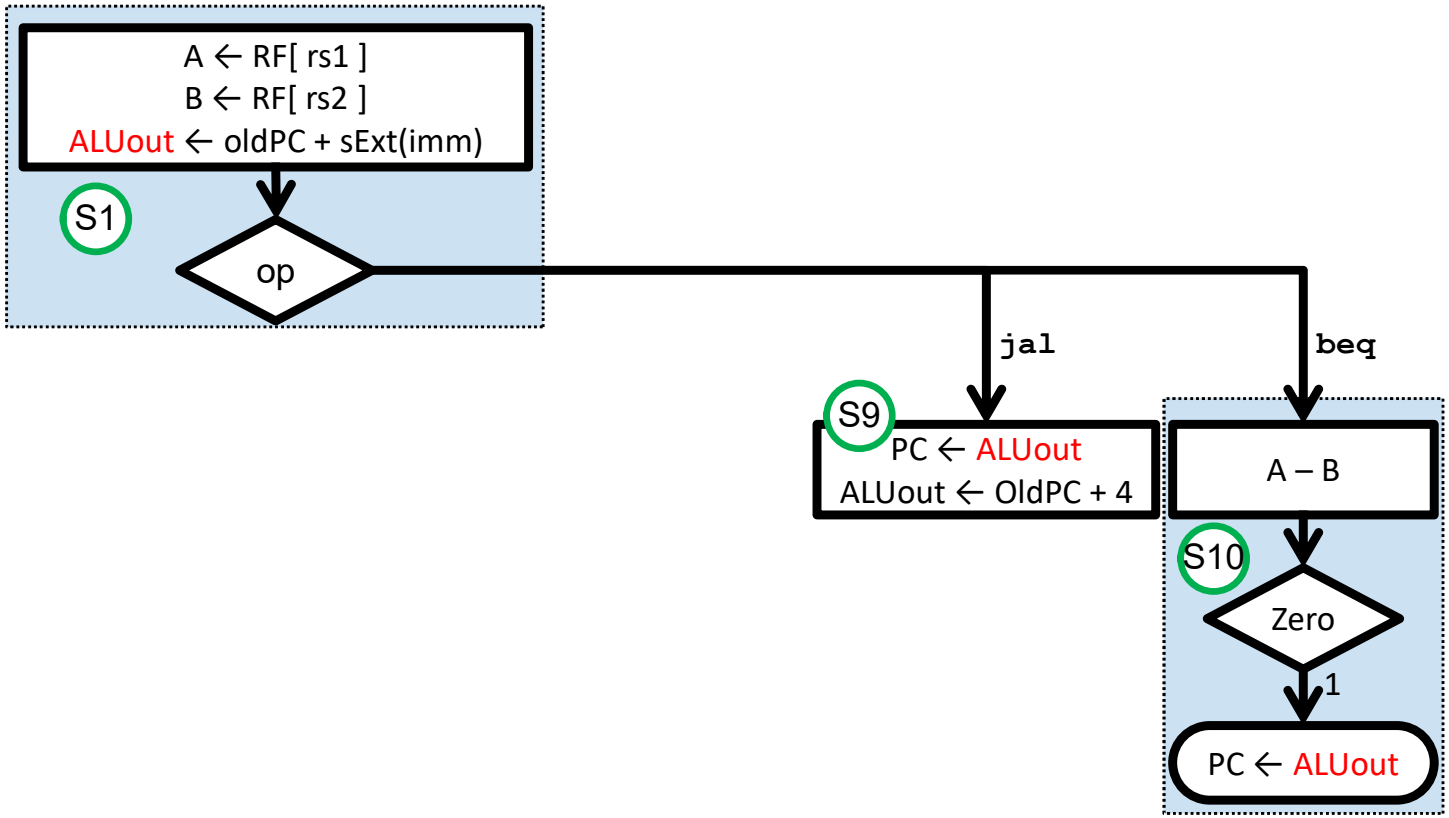
- A similar case happens with the data stored in the A auxiliary register:
  - A source operand is read from the Register File and loaded in A in S1, which is always consumed in the following state: S2, S8, S6 or S10 depending on the type of instruction.



Controller design

2nd optimization

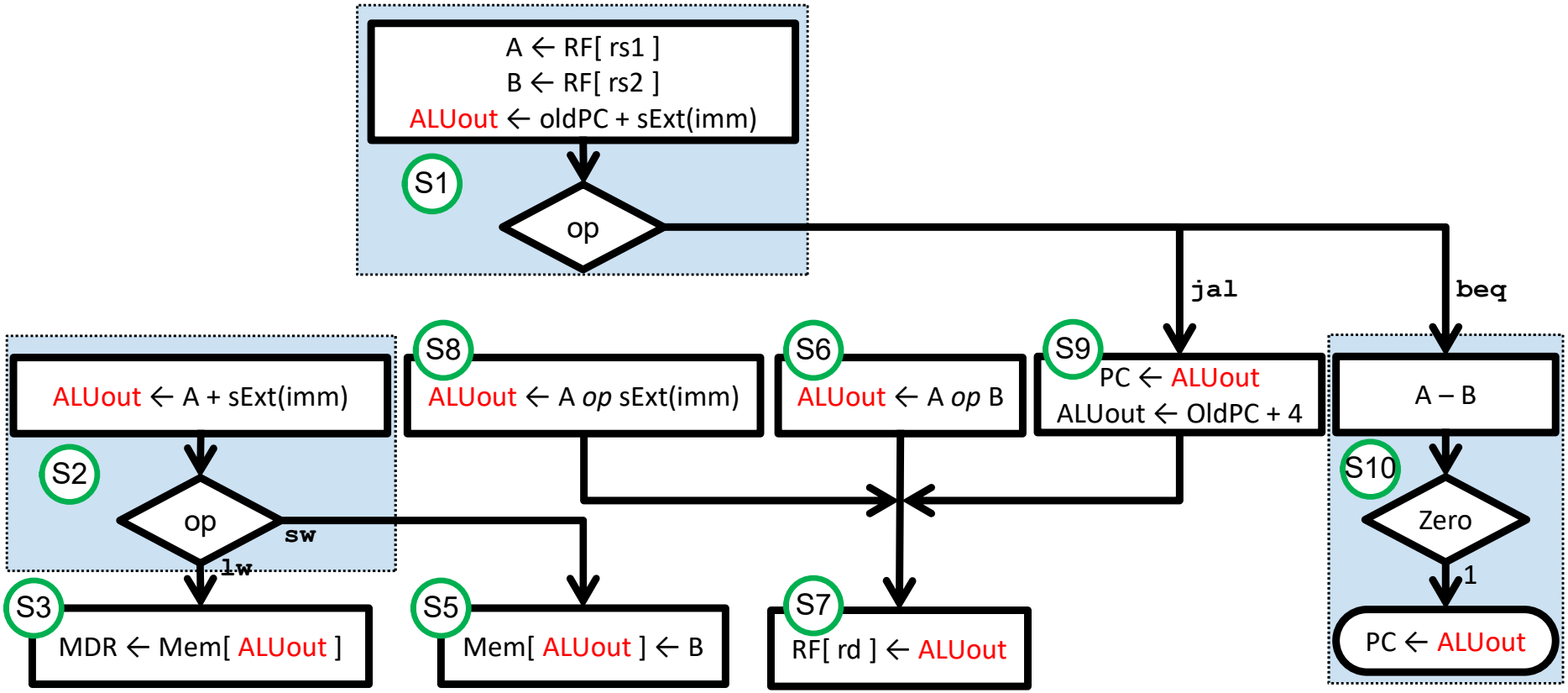
- Same happens with the data in the ALUout auxiliary register:
  - The branch address is loaded in S1 and consumed in S9 or S10.



Controller design

2nd optimization

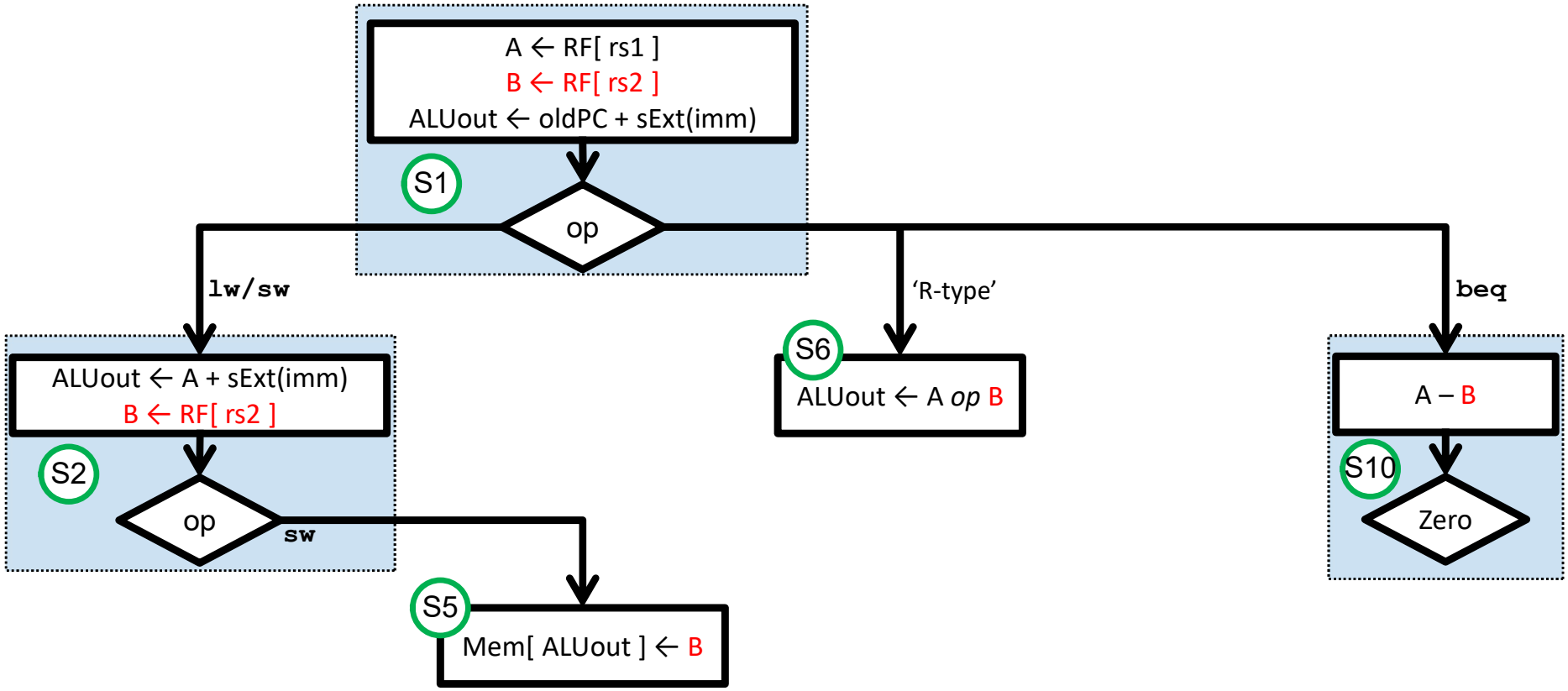
- Same happens with the data in the ALUout auxiliary register:
  - The branch address is loaded in S1 and consumed in S9 or S10.
  - The result of an ALU operation is loaded in S2 and consumed in S3 or S5. And the results loaded in S8, S6 and S9 are consumed in S7.



Controller design

2nd optimization

- The case of the B auxiliary register is slightly different:
  - An operand read from the Register File is loaded in B in S1, which is consumed in either the following cycle (S6 or S10) or 2 cycles later (S5).
  - But if B is loaded again in S2, the final behavior will not change, and this case would be similar to the previous ones.



Controller design

2nd optimization

Output function

MDRwr	Awr	Bwr	ALUoutWr
0	0	0	0
0	1	1	1
0	0	0	1
1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	1
0	0	0	0
0	0	0	1
0	0	0	1
0	0	0	0

state	Branch	PCupdate	AddrSrc	MemWr	IRwr	MDRwr	BRwr	Awr	Bwr	ALUsrcA	ALUsrcB	ALUop	ALUoutWr	ResSrc
S0	0	1	0	0	1	–	0	–	–	00	10	00	–	10
S1	0	0	–	0	0	–	0	1	1	01	01	00	1	–
S2	0	0	–	0	0	–	0	–	1	10	01	00	1	–
S3	0	0	1	0	0	1	0	–	–	–	–	–	–	00
S4	0	0	–	0	0	–	1	–	–	–	–	–	–	01
S5	0	0	1	1	0	–	0	–	–	–	–	–	–	00
S6	0	0	–	0	0	–	0	–	–	10	00	10	1	–
S7	0	0	–	0	0	–	1	–	–	–	–	–	–	00
S8	0	0	–	0	0	–	0	–	–	10	01	10	1	–
S9	0	1	–	0	0	–	0	–	–	01	10	00	1	00
S10	1	0	–	0	0	–	0	–	–	10	00	01	–	00

- All these control signals can have a value of 1 permanently and be removed from the controller
  - These auxiliary registers would store garbage in the non-relevant cycles.

Controller design

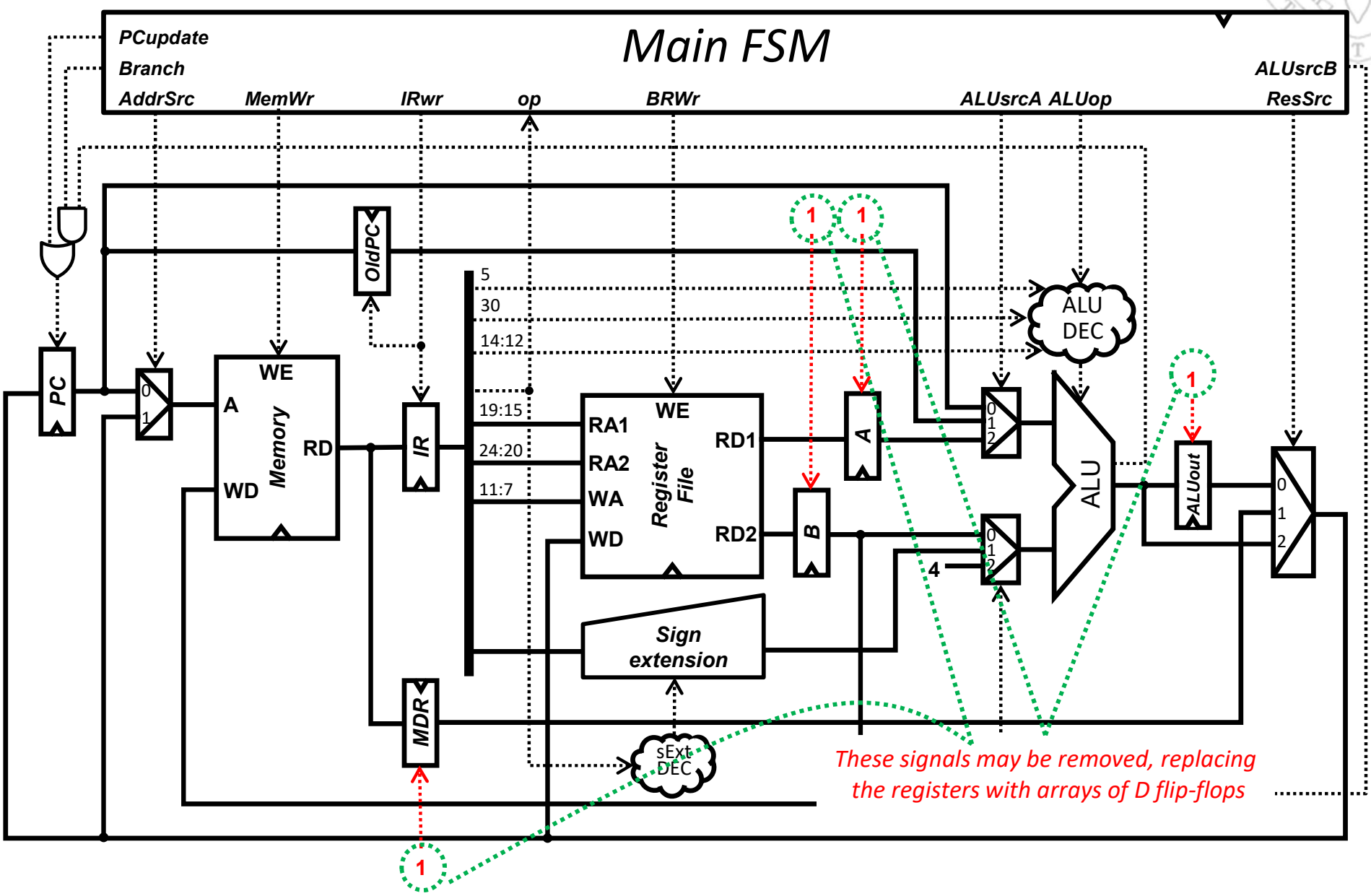
Main FSM: optimized output function

Output function

state	Branch	PCupdate	AddrSrc	MemWr	IRwr	BRwr	ALUsrcA	ALUsrcB	ALUop	ResSrc
S0	0	1	0	0	1	0	00	10	00	10
S1	0	0	–	0	0	0	01	01	00	–
S2	0	0	–	0	0	0	10	01	00	–
S3	0	0	1	0	0	0	–	–	–	00
S4	0	0	–	0	0	1	–	–	–	01
S5	0	0	1	1	0	0	–	–	–	00
S6	0	0	–	0	0	0	10	00	10	–
S7	0	0	–	0	0	1	–	–	–	00
S8	0	0	–	0	0	0	10	01	10	–
S9	0	1	–	0	0	0	01	10	00	00
S10	1	0	–	0	0	0	10	00	01	00

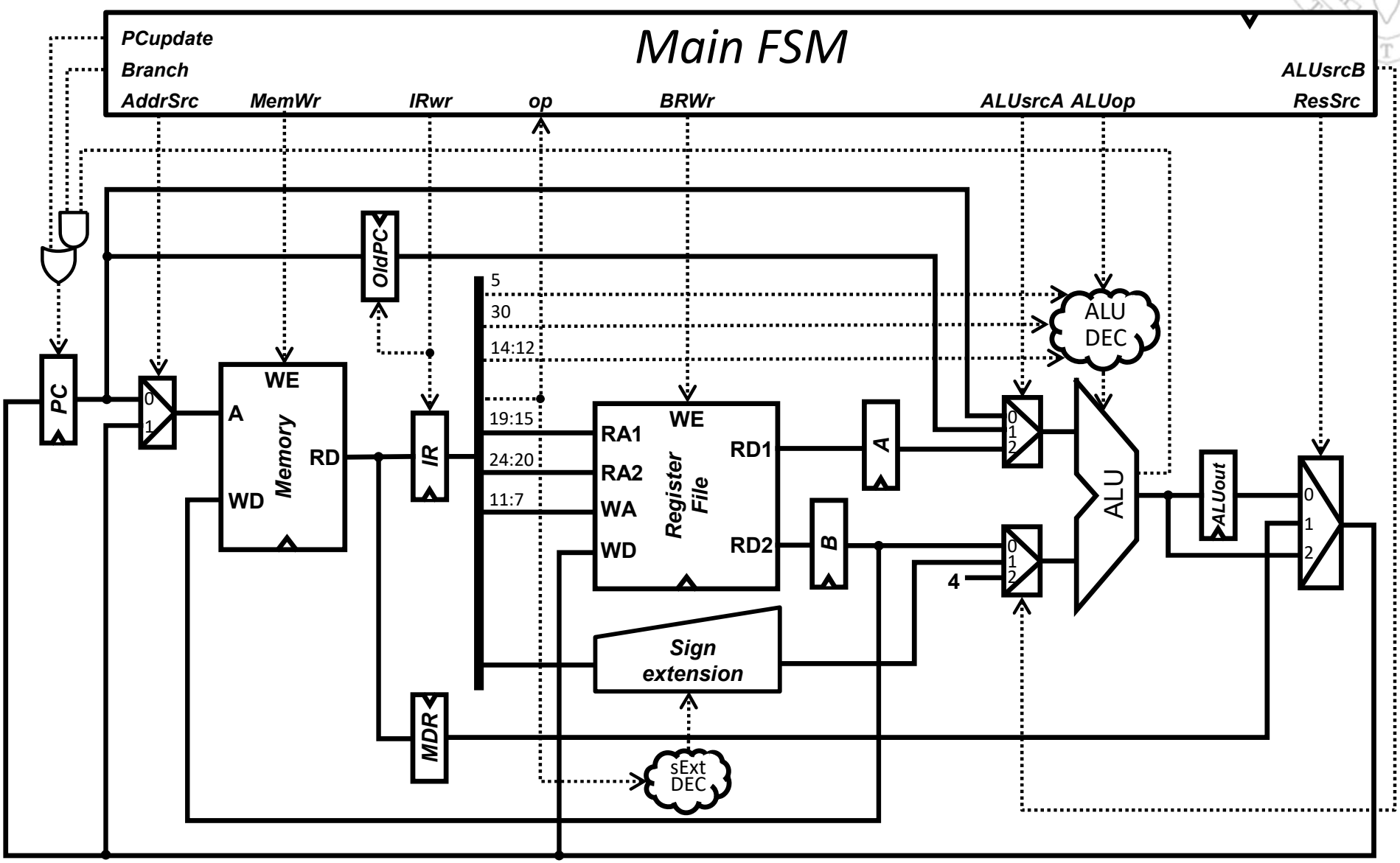
Controller design

Full system optimized structure



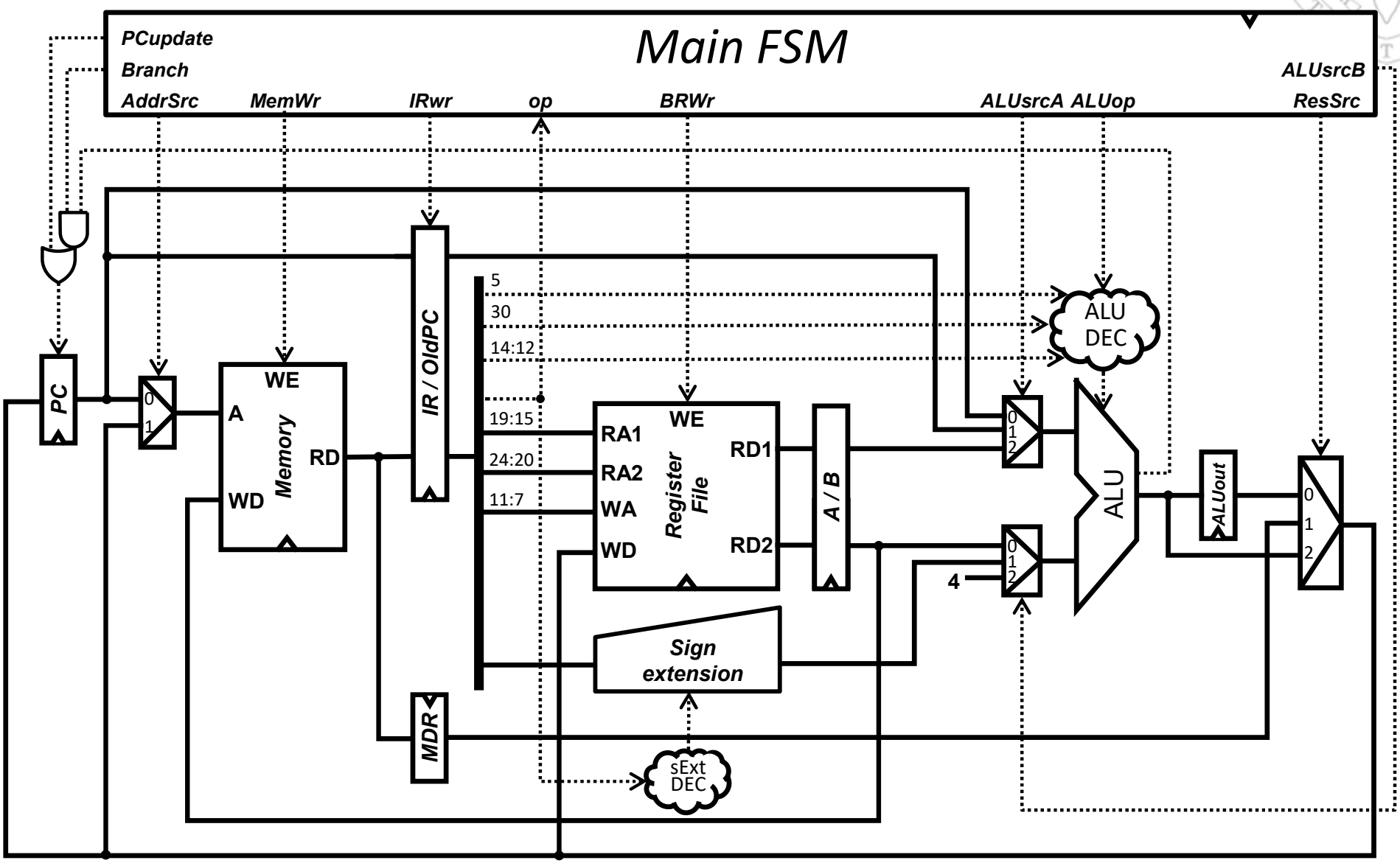
Controller design

Full system optimized structure



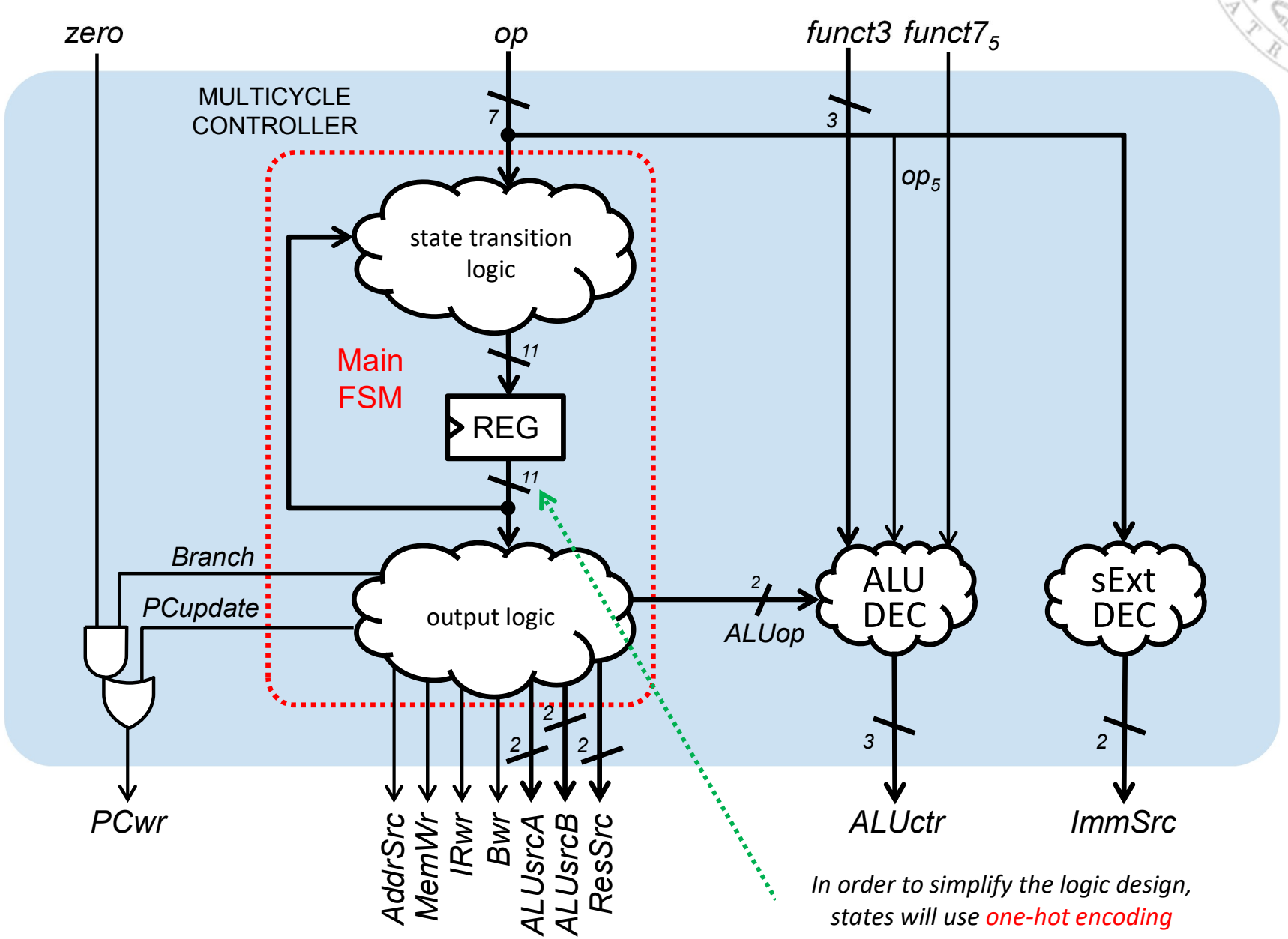
Controller design

Structure according to the terminology in Harris & Harris



Controller design

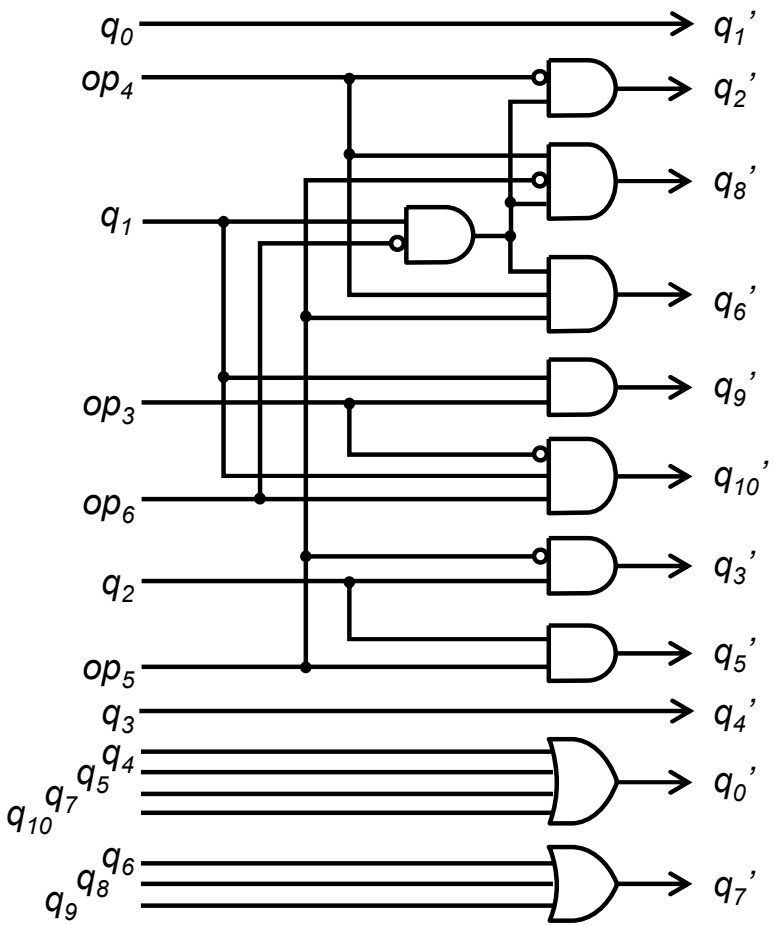
Main FSM design



In order to simplify the logic design, states will use one-hot encoding

Controller design

Main FSM design: transition function

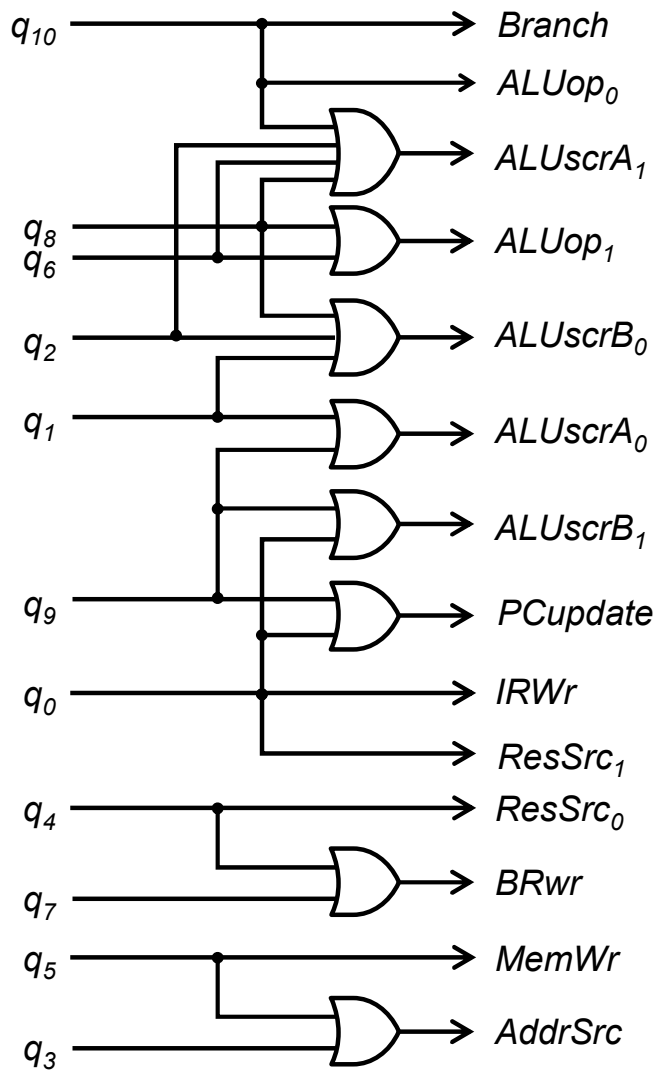


State transition function

$(q_{10}...q_0)$	op	$(q_{10}...q_0)'$
00000000001 (S0)	XXXXXXX	00000000010 (S1)
00000000010 (S1)	0X00011	00000000100 (S2)
00000000010 (S1)	0010011	00100000000 (S8)
00000000010 (S1)	0110011	00001000000 (S6)
00000000010 (S1)	1101111	01000000000 (S9)
00000000010 (S1)	1100011	10000000000 (S10)
00000000100 (S2)	0000011	00000001000 (S3)
00000000100 (S2)	0100011	00000100000 (S5)
00000001000 (S3)	XXXXXXX	00000010000 (S4)
00000010000 (S4)	XXXXXXX	00000000001 (S0)
00000100000 (S5)	XXXXXXX	00000000001 (S0)
00001000000 (S6)	XXXXXXX	00010000000 (S7)
00010000000 (S7)	XXXXXXX	00000000001 (S0)
00100000000 (S8)	XXXXXXX	00010000000 (S7)
01000000000 (S9)	XXXXXXX	00010000000 (S7)
10000000000 (S10)	XXXXXXX	00000000001 (S0)

Controller design

Main FSM design: output function



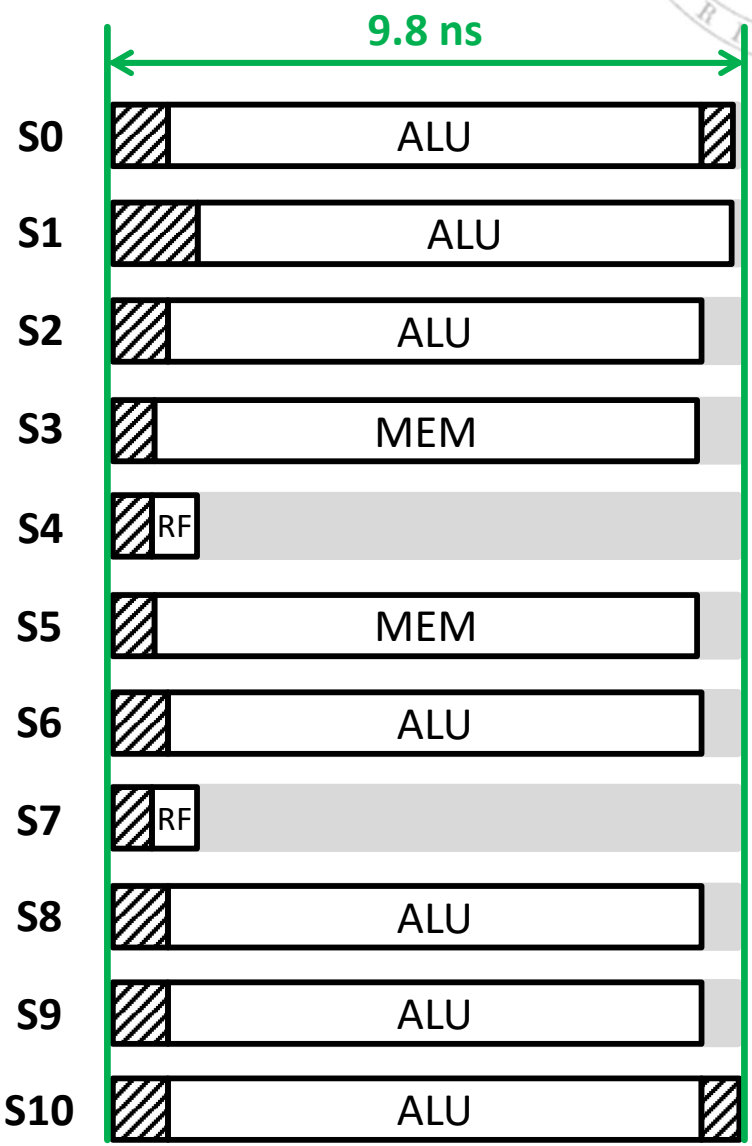
Output function

$(q_{10}...q_0)$	Branch	PCupdate	AddrSrc	MemWr	IRwr	BRwr	ALUsrcA	ALUsrcB	ALUop	ResSrc
00000000001 (S0)	0	1	0	0	1	0	00	10	00	10
00000000010 (S1)	0	0	–	0	0	0	01	01	00	–
00000000100 (S2)	0	0	–	0	0	0	10	01	00	–
00000001000 (S3)	0	0	1	0	0	0	–	–	–	00
00000010000 (S4)	0	0	–	0	0	1	–	–	–	01
00000100000 (S5)	0	0	1	1	0	0	–	–	–	00
00001000000 (S6)	0	0	–	0	0	0	10	00	10	–
00010000000 (S7)	0	0	–	0	0	1	–	–	–	00
00100000000 (S8)	0	0	–	0	0	0	10	01	10	–
01000000000 (S9)	0	1	–	0	0	0	01	10	00	00
10000000000 (S10)	1	0	–	0	0	0	10	00	01	00

Multicycle processor

Cost and cycle time (90 nm CMOS)

transfer	state	critical path
IR ← Mem [PC]	S0	9,312 ps
PC ← PC+4		9,689 ps
OldPC ← PC		589 ps
A ← RF[rs1]	S1	890 ps
B ← RF[rs2]		890 ps
ALUout ← oldPC + sExt(imm)		9,688 ps
ALUout ← A + sExt(imm)	S2	9,216 ps
MDR ← Mem[ALUout]	S3	9,140 ps
RF[rd] ← MDR	S4	1,321 ps
Mem[ALUout] ← B	S5	9,140 ps
ALUout ← A *op* B	S6	9,216 ps
RF[rd] ← ALUout	S7	1,321 ps
ALUout ← A *op* sExt(imm)	S8	9,216 ps
PC ← ALUout	S9	940 ps
ALUout ← OldPC + 4		9,216 ps
if (A – B = 0) *then* PC ← ALUout	S10	**9,790 ps**
	max.	**9,790 ps**



Multicycle processor

Cost and cycle time (90 nm CMOS)

transfer	state	critical path
IR ← Mem [PC]	S0	9,312 ps
PC ← PC+4		9,689 ps
OldPC ← PC		589 ps
A ← RF[rs1]	S1	890 ps
B ← RF[rs2]		890 ps
ALUout ← oldPC + sExt(imm)		9,688 ps
ALUout ← A + sExt(imm)	S2	9,216 ps
MDR ← Mem[ALUout]	S3	9,140 ps
RF[rd] ← MDR	S4	1,321 ps
Mem[ALUout] ← B	S5	9,140 ps
ALUout ← A *op* B	S6	9,216 ps
RF[rd] ← ALUout	S7	1,321 ps
ALUout ← A *op* sExt(imm)	S8	9,216 ps
PC ← ALUout	S9	940 ps
ALUout ← OldPC + 4		9,216 ps
if (A – B = 0) *then* PC ← ALUout	S10	**9,790 ps**
	max.	**9,790 ps**

$$area = 65626\ \mu m^2$$

$$t_{clk} = 9.8\ \text{ns}$$

$$f_{clk} = \frac{1}{t_{clk}} = \frac{1}{9.8 \cdot 10^{-9}\text{s}} = 102\ \text{MHz}$$

```
   ...
L7:
   lw   x6, -4(x9)   ·····→  5 cycles
   sw   x6, 8(x9)    ·····→  4 cycles
   or   x4, x5, x6   ·····→  4 cycles
   beq  x4, x4, L7   ·····→  +3 cycles
   ...
                             16 cycles
```

$$t_{exec} = 16 \times 9.8\ ns = 156.8\ ns$$

Comparison

Reduced RISC-V: single-cycle vs. multicycle (i)

- Single-cycle processor:
  - All instructions take one cycle to execute.
  - It has a long cycle time determined by the slowest instruction.
  - All the resources are dedicated to perform a single operation.
  - It requires two separate data and instruction memories.

- Multicycle processor:
  - All instructions take more than one cycle to execute.
    - Simple instructions take fewer cycles to execute than complex ones.
  - It has a short cycle time determined by the slowest micro-operation.
  - Resources may be used to perform different operations in different clock cycles.
  - There is a single memory shared by data and instructions.
  - It requires non-architectural auxiliary registers (i.e., invisible to programmers) in order to store partial results.

Comparison

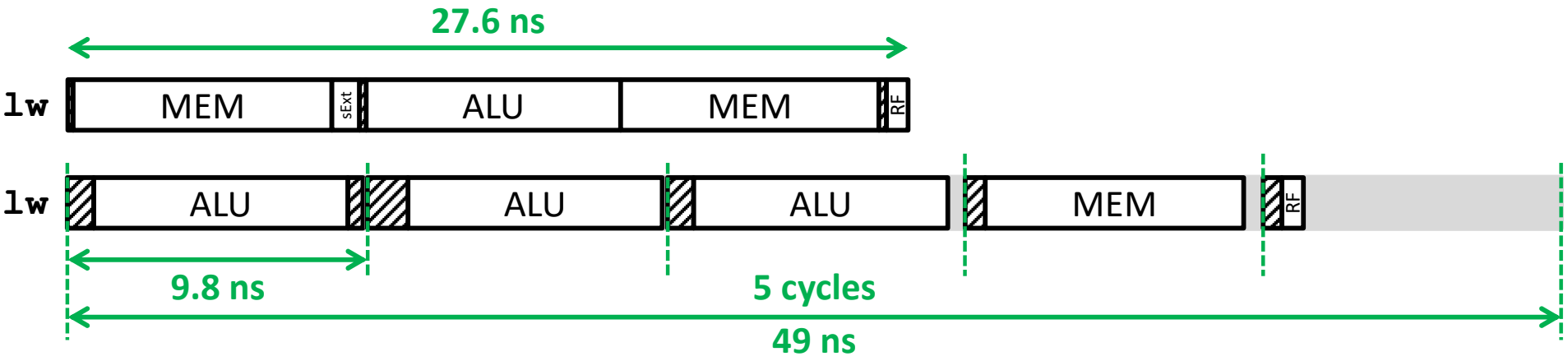
Reduced RISC-V: single-cycle vs. multicycle (ii)

- The multicycle processor is more expensive than the single-cycle

$$65{,}626\ \mu m^2 = cost_{multicycle} > cost_{single-cycle} = 59{,}181\ \mu m^2$$

- The multicycle processor has worse performance than the single-cycle:

$$156.8\ ns = t_{exec-multicycl} > t_{exec-single-cycl} = 110.4\ ns$$



- It looks like the multicycle processor is worse, however, there are no single-cycle processors in the market.
- The explanation of this paradox is based on the simplifications that we have made during the design process.

Comparison

Reduced RISC-V: single-cycle vs. multicycle (iii)

- The single-cycle processor is cheaper because:
  - The cost of the memory has not been considered.
    - This processor requires 2 memories and the multicycle version only 1.
  - Reusing the ALU implies adding multiplexers and registers with a higher cost than the removed adders.
    - Increasing the reuse would make the multicycle processor cheaper.
- The multicycle processor has worse performance because:
  - It has been assumed that the memory access time and the ALU computation time are less than the cycle time.
    - The external RAM memories and the complex ALU are slower.
  - The computational load of the states is not perfectly balanced.
    - There are cycles in which most of the HW is inactive.
- A single-cycle processor is only better in simplified architectures.
  - Complex ISAs require adding more functional elements that cannot be reused and which make the critical path longer.

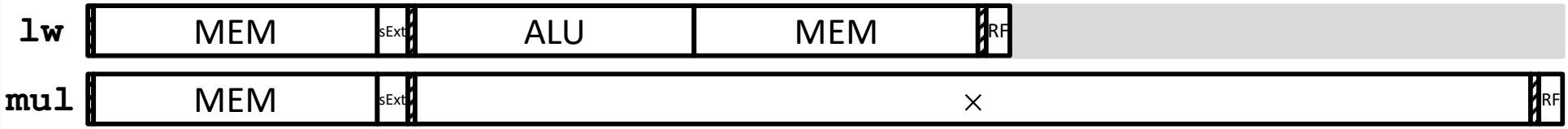
Comparison

Reduced RISC-V: single-cycle vs. multicycle (iv)

- For example, let us suppose that we extend the ISA to include a multiplication instruction:

`mul rd, rs1, rs2`	rd ← rs1 × rs2	R-type

- We add a 32-bit combinational multiplier to both data paths, with a computation time 4 times greater than the one of the ALU.
  - In the single-cycle processor, the cycle time would be determined by the slowest instruction, which in this case is the `mul` instruction.



$$t_{clk-single-cycle} = \mathbf{44\ ns}\quad \mathbf{(18,928\ ps + 3 \times 8,360\ ps)}$$
$$t_{exec-single-cycle} = 4 \times 44\ ns = \mathbf{176\ ns}$$

  - In the multicycle processor, the cycle time would remain the same at the expense of the `mul` instruction taking 7 cycles in its execution.
    - The number of cycles required by the other instructions would not change.

$$t_{clk-multicycle} = \mathbf{9.8\ ns}$$
$$t_{exec-multicycle} = 16 \times 9.8\ ns = \mathbf{156.8\ ns}$$

Performance metrics

Execution time of a program

- Performance metrics allow comparing the performance of different processors in an objective way.
  - From the user point of view, the total execution time of a program is the most reliable metric.
  - Since it depends on many factors, we will focus on the CPU time.
- The execution time of N instructions (of i types), which are part of a program running in a given CPU will be:

$$\boldsymbol{t_{exec}} = t_{clk} \cdot \sum_i c_i \cdot n_i \equiv \sum_i c_i \cdot n_i / f_{clk}$$

- where:
  - n_i = number of executed instructions of type i.
  - c_i = number of cycles that takes the execution of a type i instruction.
  - t_{clk} = cycle time (in seconds/cycle).
  - f_{clk} = clock frequency (in hertz = cycles/second).
  - $N \equiv \sum_i n_i$ = total number of executed instructions.
  - $\sum_i c_i \cdot n_i$ = total number of cycles that the program takes in its execution

Performance metrics

Execution time of a program

- To get a more compact expression of the execution time, the Cycles Per Instruction (CPI) metric is introduced.
  - The CPI is the weighted sum of the number of cycles that each kind of instruction takes to execute.
  - The CPI is specific to each program and each CPU.

$$\boldsymbol{CPI} = \frac{\sum_i c_i \cdot n_i}{\sum_i n_i}$$

- The execution time of N instructions of a certain program, running in a given CPU is:

$$\boldsymbol{t_{exec}} = N \cdot CPI \cdot t_{clk} \equiv \frac{N \cdot CPI}{f_{clk}}$$

- where:
  - $N \equiv \sum_i n_i$ = total number of executed instructions.
  - $N \cdot CPI$ = total number of cycles that the program takes to execute.
  - t_{clk} = cycle time (in seconds/cycle).
  - f_{clk} = clock frequency (in hertz = cycles/second).

Performance metrics

Execution time of a program

- This expression is interesting because it allows highlighting the contribution of each design element to the computer performance.

Programmer's skill
Compiler
Computer architecture

Computer organization
Microelectronic technology

CISC

R1 ← [addr(A) + 4 * R2] + 1

RISC

R3 ← 4 * R2
R3 ← addr (A) + R3
R5 ← [R4]
R1 ← R5 + 1

$$t_{exec} = N \cdot CPI \cdot t_{clk}$$

Computer architecture
Computer organization

Performance metrics

MIPS and MFLOPS

- There are other less reliable metrics that are commonly used, which indicate the number of instructions that a computer executes per second.
  - Instructions of an architecture family can have very different functionality and execution times respect to other families (RISC vs CISC), making this comparison useless.
- Millions of Instructions Per Second (MIPS):

$$MIPS = \frac{N}{10^6 \cdot t_{exec}} \equiv \frac{f_{clk}}{10^6 \cdot CPI}$$

- Millions of FLoating-point Operations Per Second (MFLOPS)
  - These are chosen because they are the instructions that need more time to execute.
  - This is not good to compare processors/programs with integer arithmetic.

Performance metrics

Reduced RISC-V: single-cycle vs. multicycle

- Consider a program that executes 10^8 instructions (100 million) so that:
  - 25% of the instructions are `lw`
  - 10% of the instructions are `sw`
  - 11% of the instructions are `beq`
  - 2% of the instructions are `jal`
  - 52% of the instructions are arithmetic-logic

- Single-cycle CPI: all the instructions take 1 cycle (CPI = 1).

$$\begin{aligned} \boldsymbol{CPI} &= \mathbf{1} \\ \boldsymbol{t_{exec}} &= 10^8 \cdot 1 \cdot 27.6\, ns = \mathbf{2.76\ s} \\ \boldsymbol{MIPS} &= 10^8/(10^6 \cdot 2.76\ s) = \mathbf{32.6\ Minst/s} \end{aligned}$$

- Multicycle CPI: all the instructions take more than 1 cycle (CPI > 1).
  - `lw`: 5 cycles, `sw`: 4 cycles, `beq`: 3 cycles, `jal`: 4 cycles, arithmetic-logic: 4 cycles

$$\begin{aligned} \boldsymbol{CPI} &= 0.25 \cdot 5 + 0.10 \cdot 4 + 0.11 \cdot 3 + 0.02 \cdot 4 + 0.52 \cdot 4 = \mathbf{4.14} \\ \boldsymbol{t_{exec}} &= 10^8 \cdot 4.14 \cdot 9.8\, ns = \mathbf{4.06\ s} \\ \boldsymbol{MIPS} &= 10^8/(10^6 \cdot 4.06\ s) = \mathbf{25\ Minst/s} \end{aligned}$$

Performance metrics

Speedup

- The comparison between absolute execution times of the same program in 2 different processors is not very visual.
  - It is more effective to indicate how much faster a processor is respect to the other.
- *Speedup* is the relation between the execution time of the same program in two processors, A and B
  - It measures how much faster processor A is respect to processor B, executing the same program.

$$Speedup = \frac{t_{exec}^{B}}{t_{exec}^{A}}$$

- Thus, we can say that the single-cycle processor executes the previous program 1.47 times faster than the multicycle processor because:

$$t_{exec}^{single-cycle} = 2.76\ s \qquad t_{exec}^{multicycle} = 4.06\ s$$

$$Speedup = \frac{t_{exec}^{multicyle}}{t_{exec}^{single-cycle}} = \frac{4.06}{2.76} = 1.47$$

Performance metrics

Benchmarking

- The result of any performance metric depends on the program that is executed
  - To make the comparison fair, the executed program should be the same.
  - But a single program may not be representative enough.

- A benchmark is a collection of programs that are used to compare different computers
  - Dhrystone, CoreMark: integer calculation synthetic programs.
  - Whetstone: floating-point calculation synthetic programs.
  - Linkpack: scientific calculation actual programs.
  - SPEC: integer and floating-point actual programs.

Performance metrics

TOP500

- The Top500 project is the ranking of the 500 supercomputers with the greatest performance worldwide.

#	Computer	Country	Manufacturer	# cores	Processor	Peak Perf. (Pflops*)	Power (kW)
1	Frontier	USA	HP	8,730,112	AMD Opt 3rd Gen EPYC 64C @ 2GHz	**1,685.65**	21,100
2	Fugaku	Japan	Fujitsu	7,630,848	A64FX 48C @ 2.2GHz	537.21	29,899
3	LUMI	Finland	HP	1,110,144	AMD Opt 3rd Gen EPYC 64C @ 2GHz	214.35	2,942
4	Summit	USA	IBM	2,414,592	IBM POWER9 22C @ 3.07GHz	200.79	10,096
5	Sierra	USA	IBM	1,572,480	IBM POWER9 22C @ 3.1GHz	125.71	7,438
...	...	...	...	...	...	...	...
82	MareNostrum	Spain	Lenovo	153,216	Xeon Platinum 8160 24C @ 2.1GHz	10.30	1,632

source: Top 500 (June 2022), https://www.top500.org/

(*) 1 Pflops = 10^9 Mflops

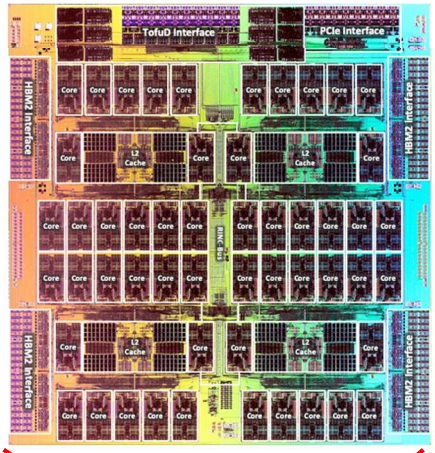
- My laptop.

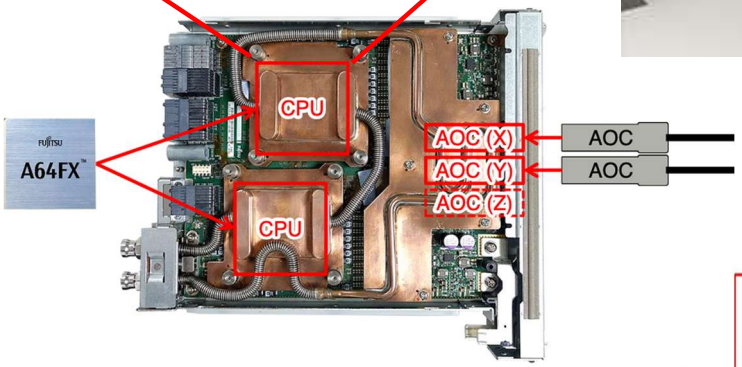
#	Computer	Country	Manufacturer	# cores	Processor	Peak Perf. (Mflops)	Power (kW)
–	Elite Dragonfly	USA	HP	4	Intel Core i7-8565U @ 1.80GHz	**13.77**	–

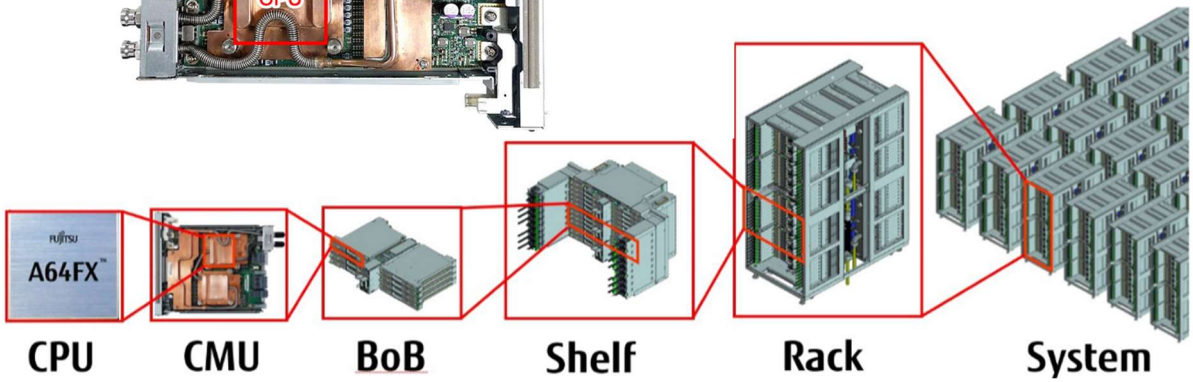
source: PassMark Software Pty Ltd , https://www.cpubenchmark.net

Performance metrics

TOP500: Fugaku





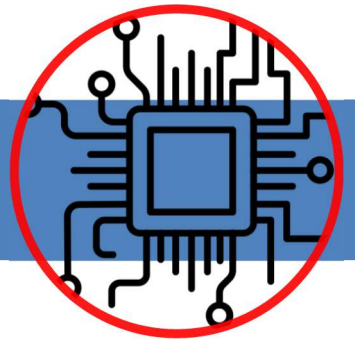


Unit	# nodes	Description
CPU	1	Node of 48+2 processors
CMU	2	CPU Memory Unit: 2 CPUs
BoB	16	Bunch of Blades: 8 CMUs
Shelf	48	3 BoBs
Rack	192/384	4/8 Shelfs
Fugaku	158976	396 + 36 Racks (8/4 Shelfs)

sources: Jack Dongarra, Report on the Fujitsu Fugaku System, Tech Report No. ICL-UT-20-06 (2020)
Riken Center for Computational Science, https://www.r-ccs.riken.jp/en/

- Cost calculation.
- Cycle time calculation.

Technology



Cost and cycle time calculation

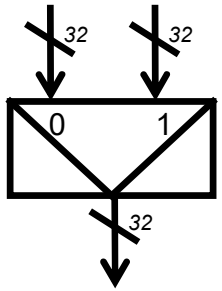
- The processor cost is the addition of the costs of each of the components that form it.
  - The cost of each component is calculated by adding the cost of its cells.

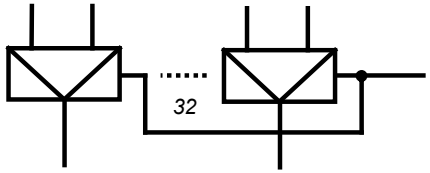
- The processor cycle time is the maximum critical path of the register transfers performed by the processor.
  - The critical path of a register transfer is the data path with the largest delay among all the paths involved in that transfer.
  - In the multicycle processor, between 1 and 3 register transfers are performed per cycle. The execution of one instruction implies performing between 7 and 9 register transfers.

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

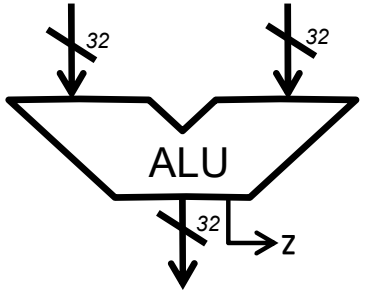
Cost and cycle time calculation

90 nm CMOS

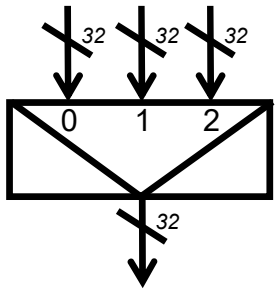


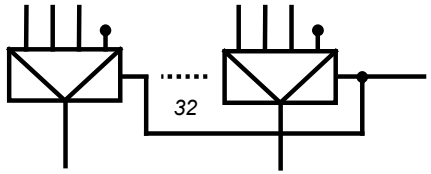


area: 32×11.05 = **354 μm²**
delay: **223 ps**

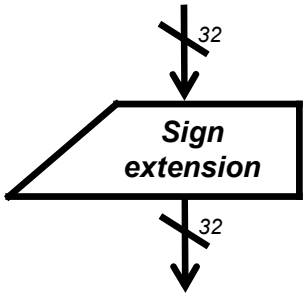


area: **3,052 μm²**
delay: **8,360 ps**





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

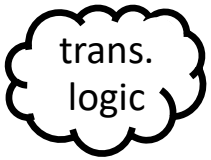


area: **202 μm²**
delay: **460 ps**

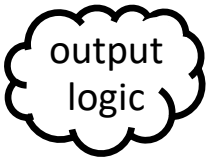
area: **56 μm²**
delay: **490 ps**

area: **15 μm²**
delay: **351 ps**

area: **21 μm²**
delay: **451 ps**



area: **107 μm²**
delay: **486 ps**

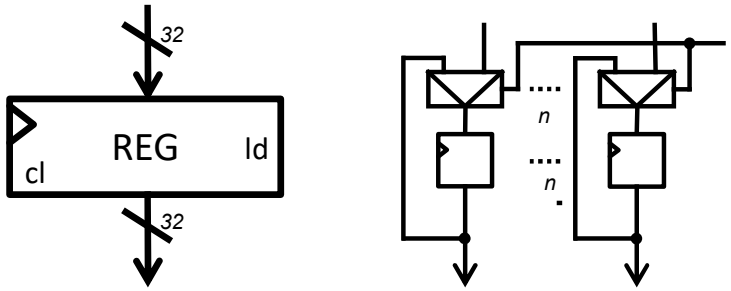


area: **64 μm²**
delay: **199 ps**

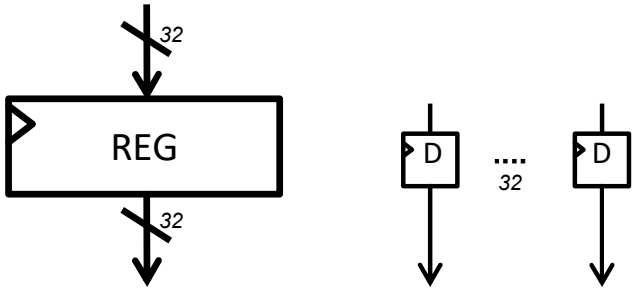
90 nm CMOS cell library, source: Synopsys (SAED EDK 90 nm)

Cost and cycle time calculation

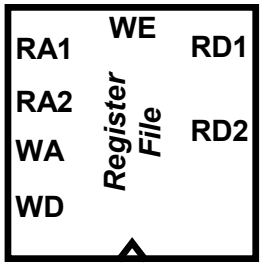
90 nm CMOS



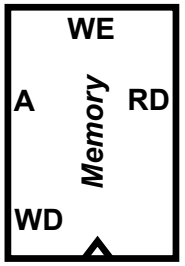
area: 32×11.05 + 32×32.26 = **1,386 μm²**
CLK→Q delay: **167 ps**
setup: 1×223 = **223 ps** (due to a load MUX)



area: 32×24.88 = **796 μm²**
CLK→Q delay: **167 ps**
setup: **0 ps**



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



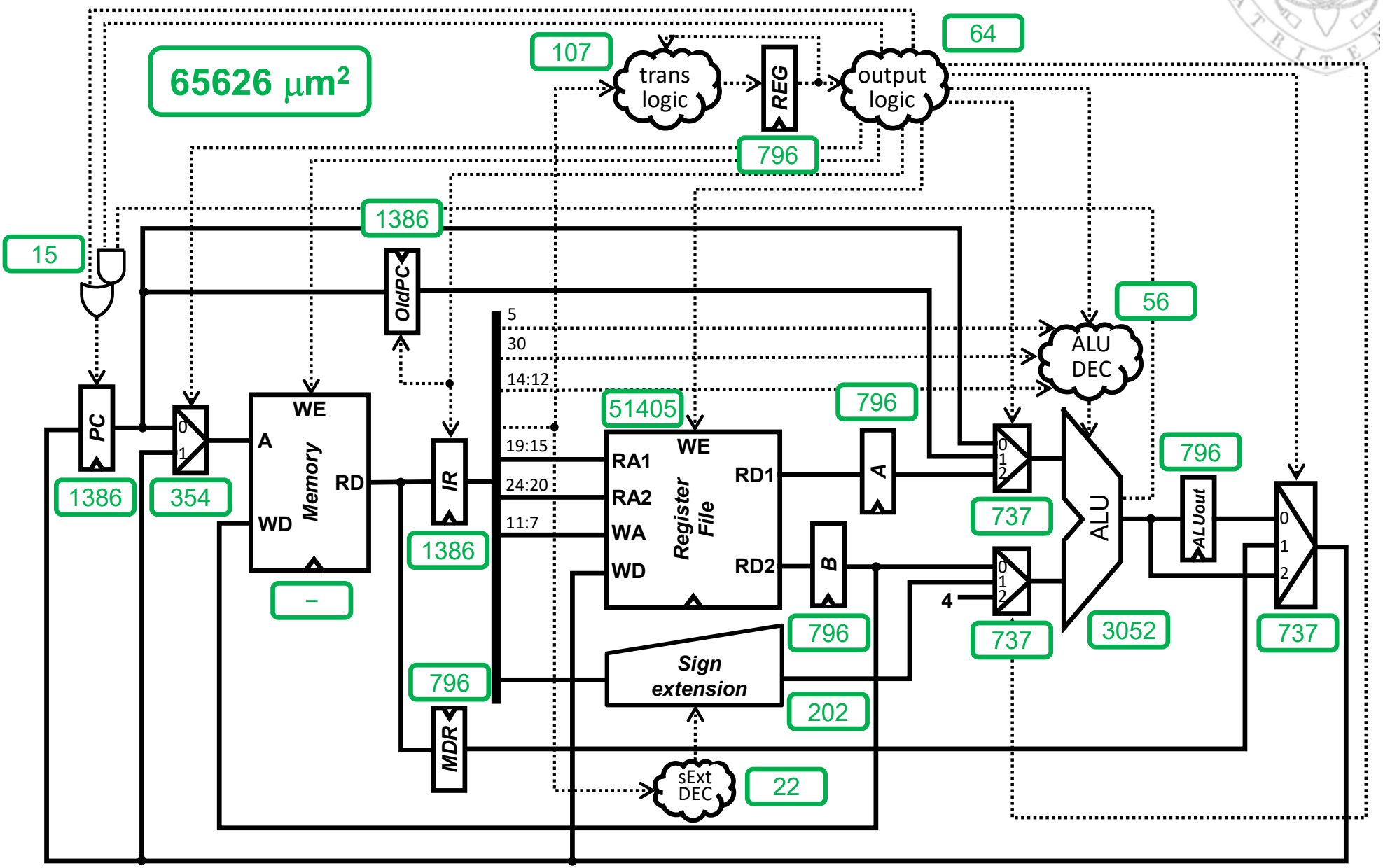
area: **–**
access time: **8,500 ps**

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

area: **525 μm²**
CLK→Q delay: **366 ps**
setup: **486 ps** (due to the trans. logic)

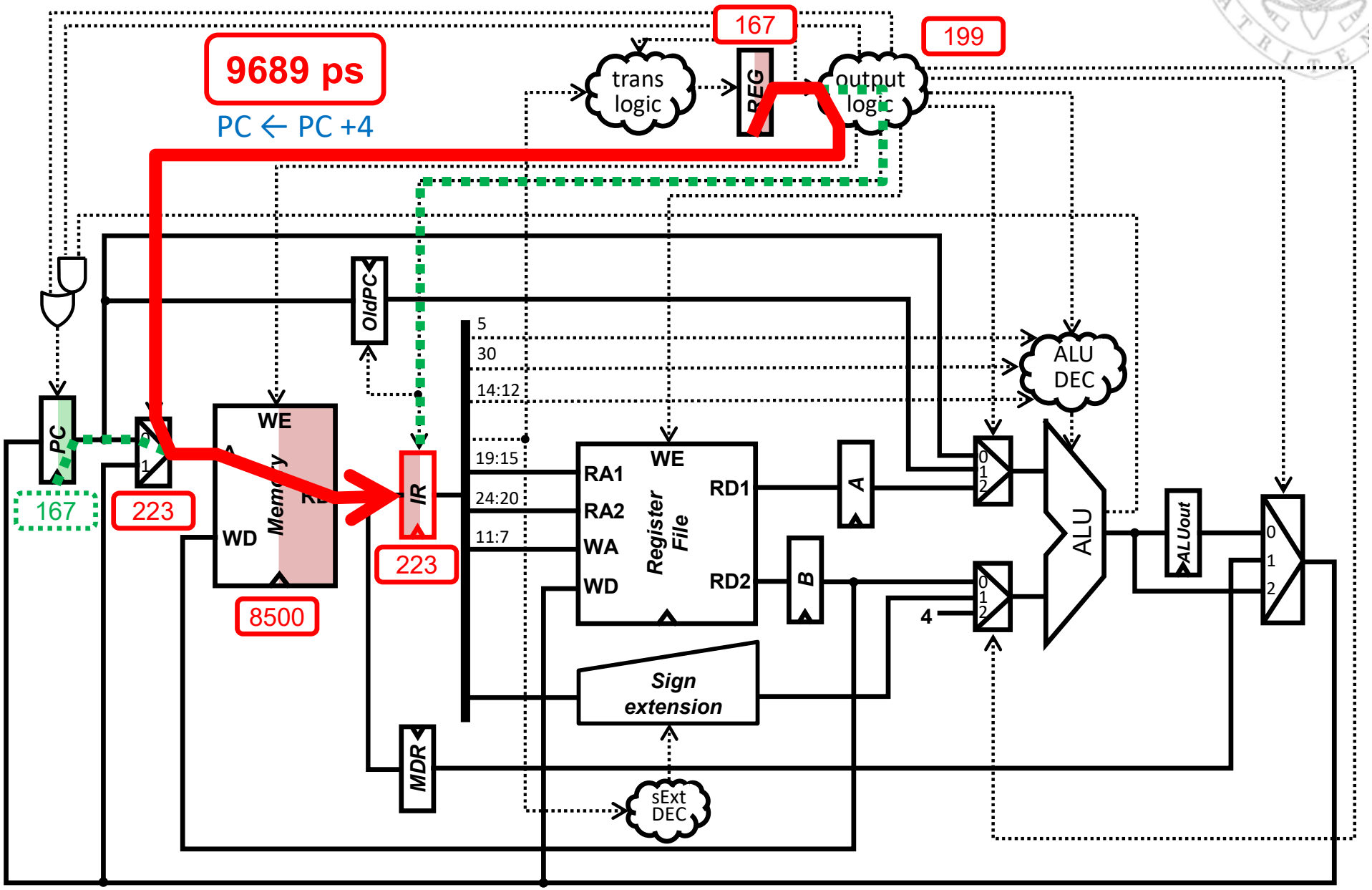
90 nm CMOS cell library, source: Synopsys (SAED EDK 90 nm)

Cost calculation



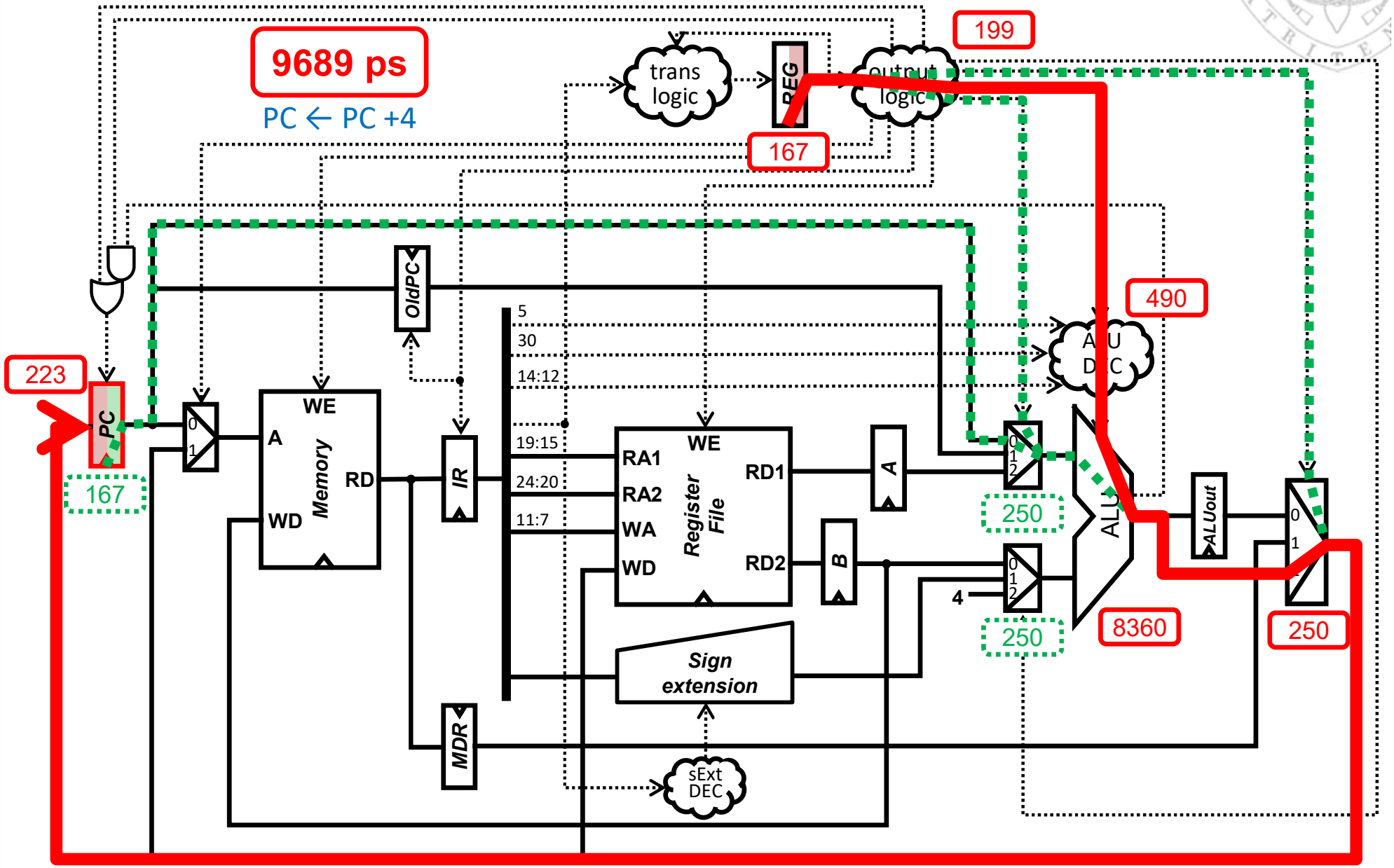
Cycle time calculation

State S0 (instruction fetch): critical path



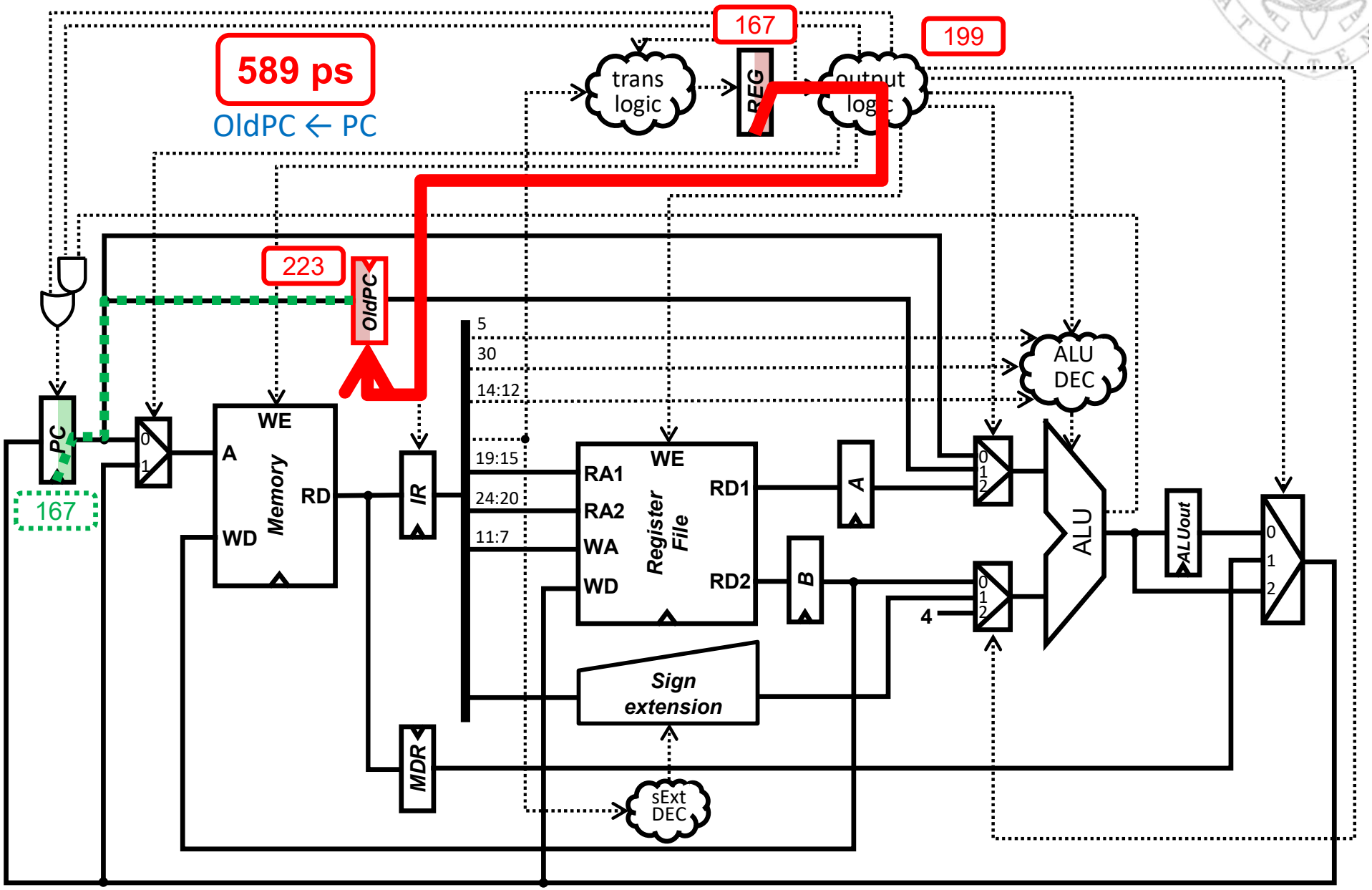
Cycle time calculation

State S0 (PC increment): critical path



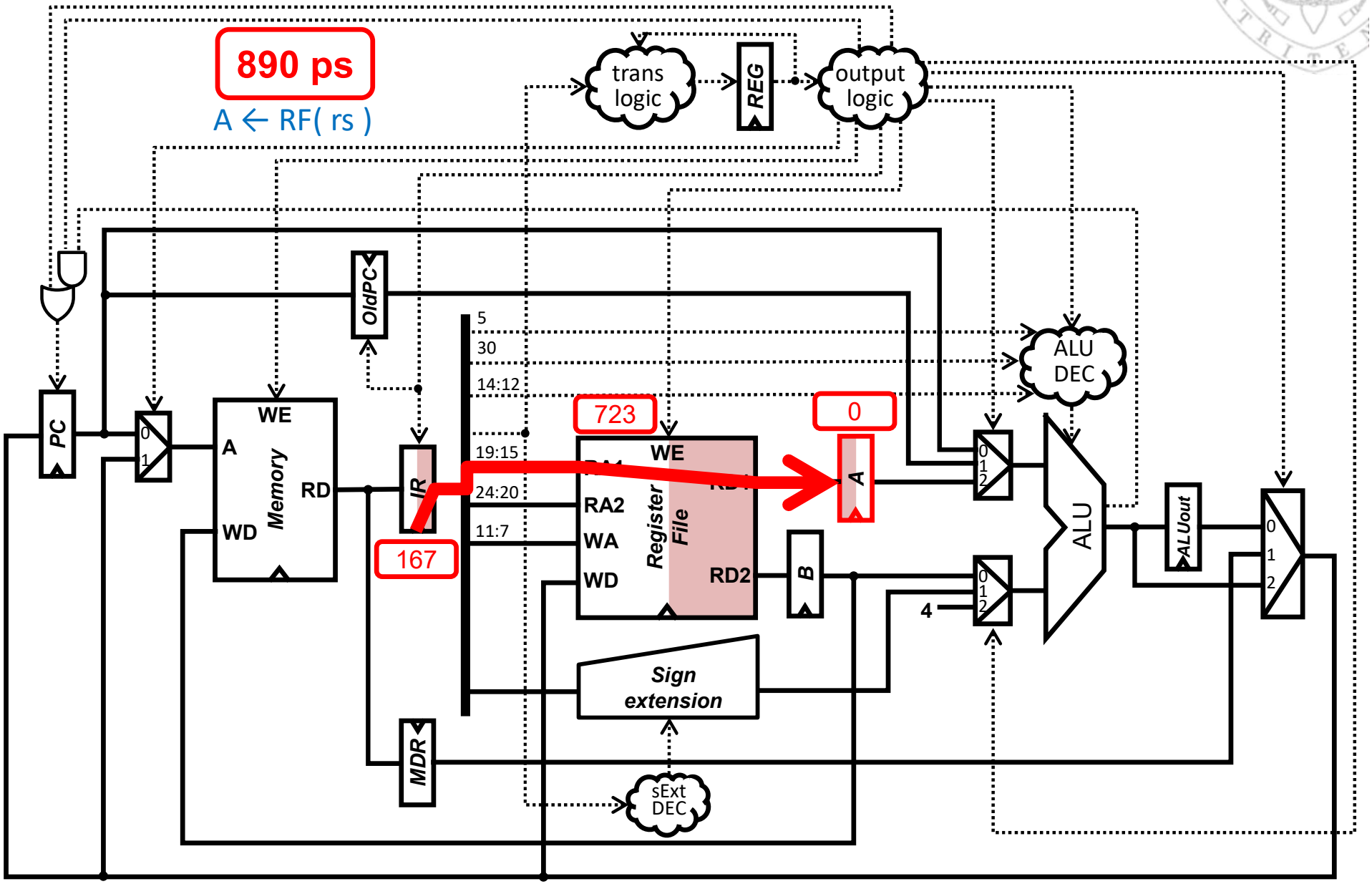
Cycle time calculation

State S0 (PC save): critical path



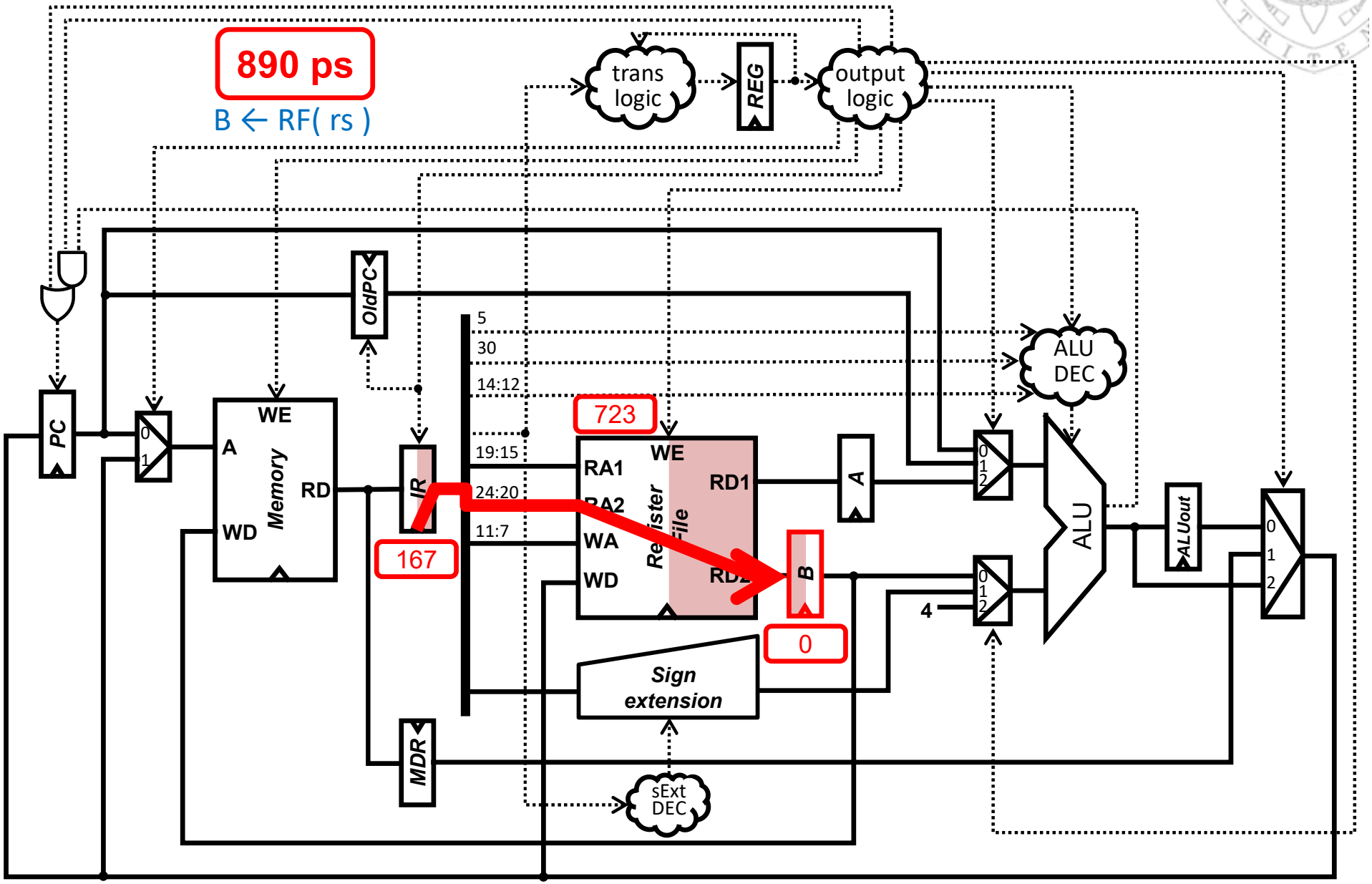
Cycle time calculation

State S1 (operand search): critical path



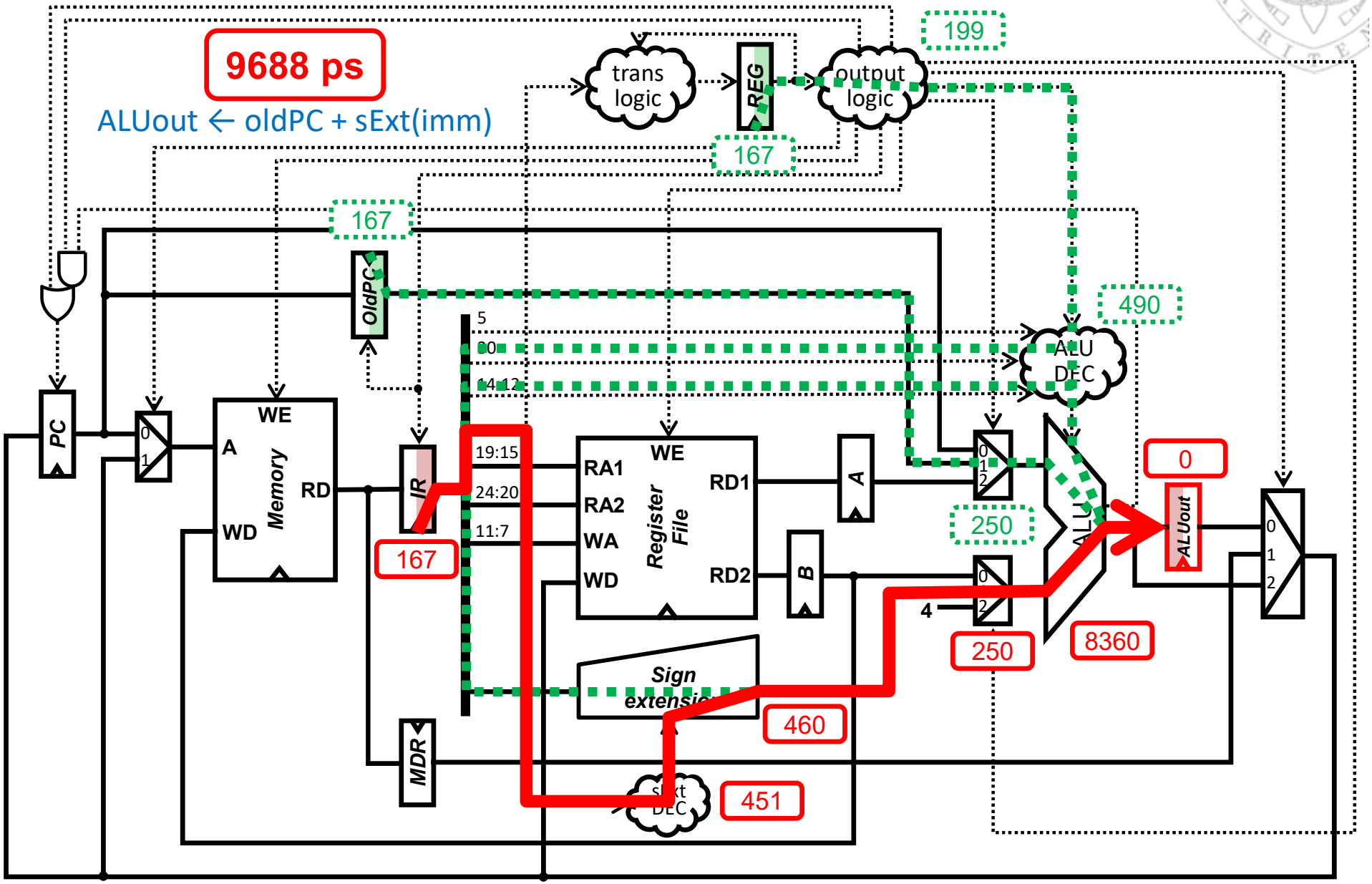
Cycle time calculation

State S1 (operand search): critical path



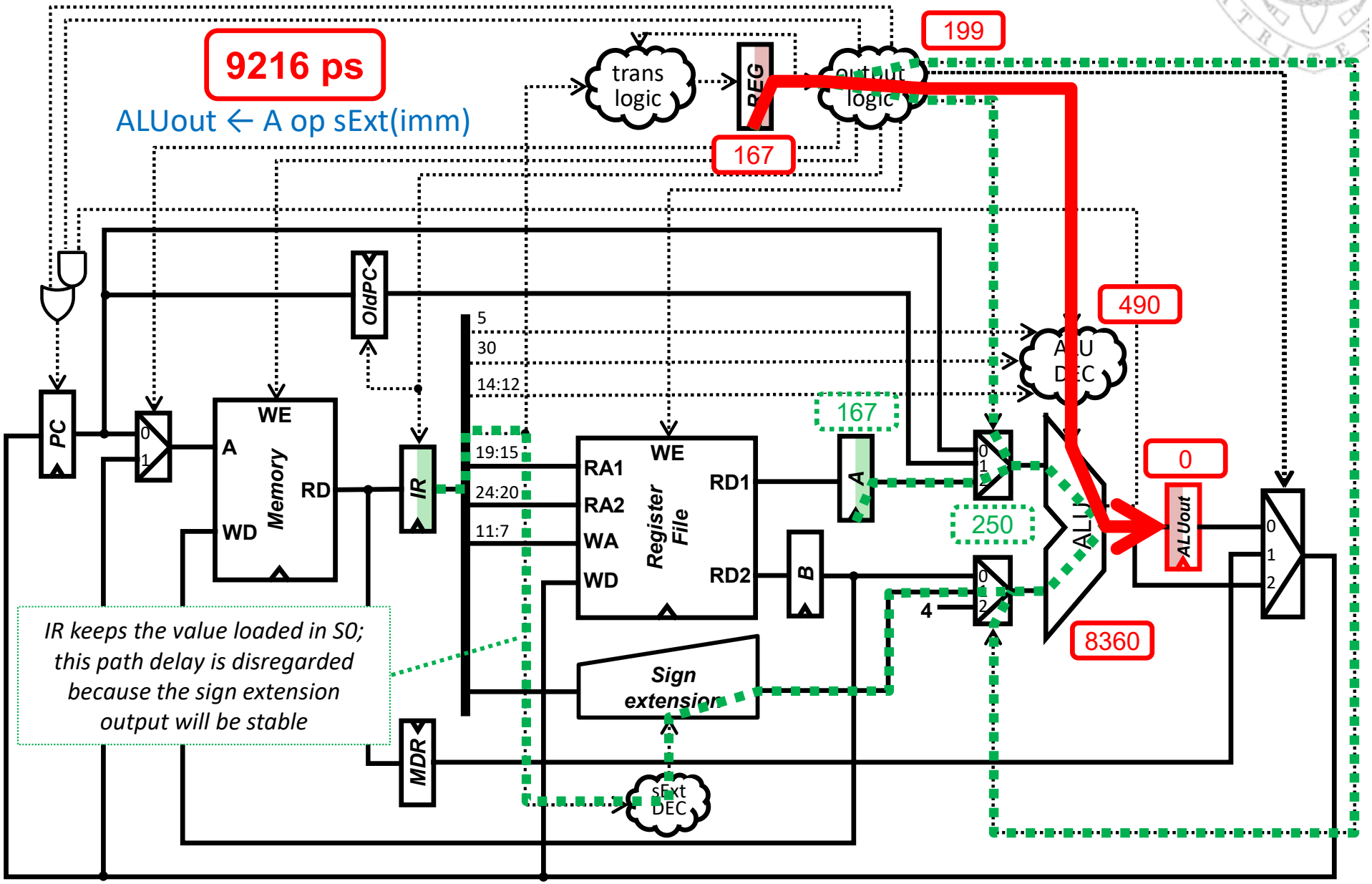
Cycle time calculation

State S1 (branch address calculation): critical path



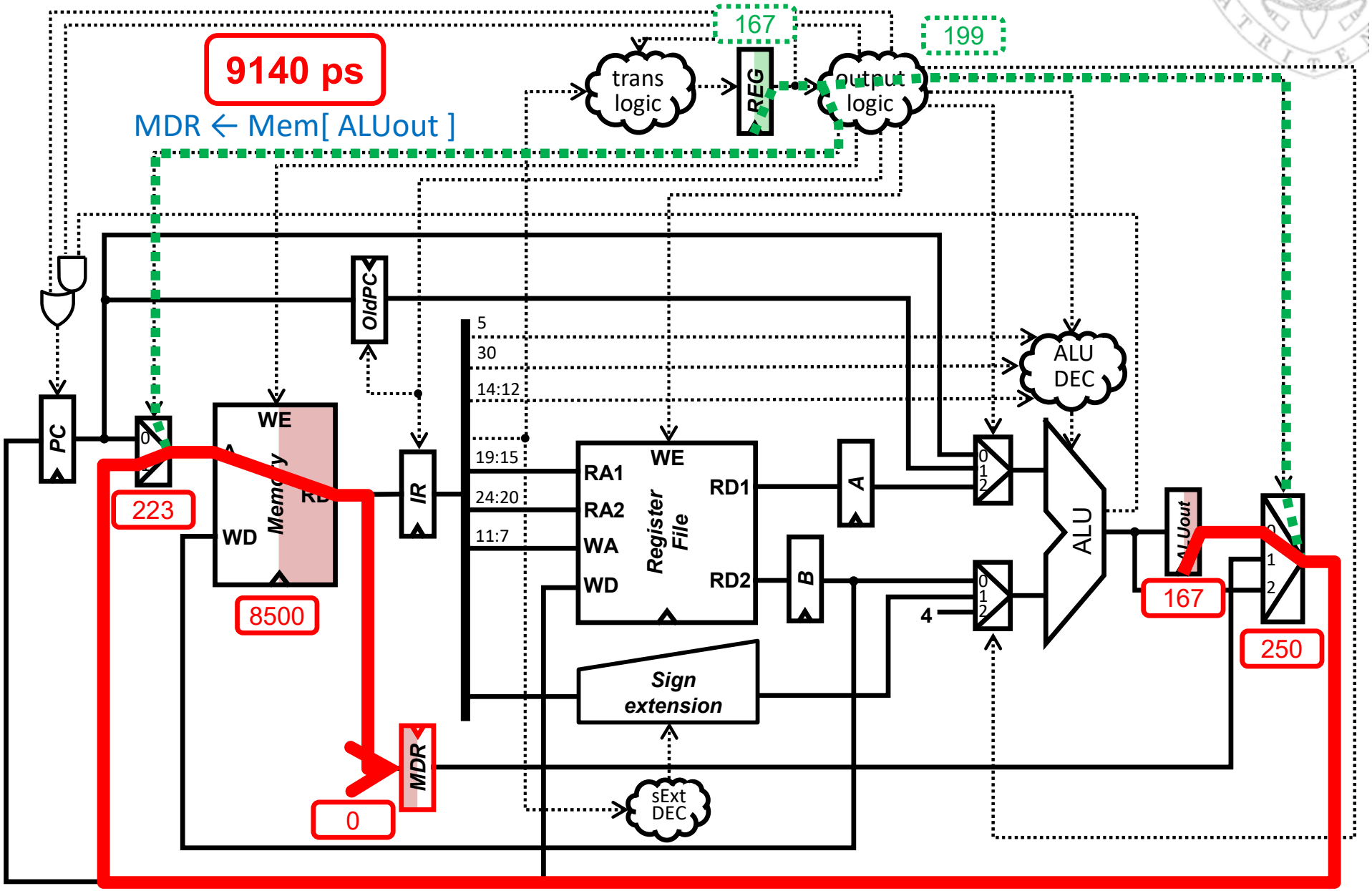
Cycle time calculation

State S2 / S6 / S8 (ALU calculation): critical path



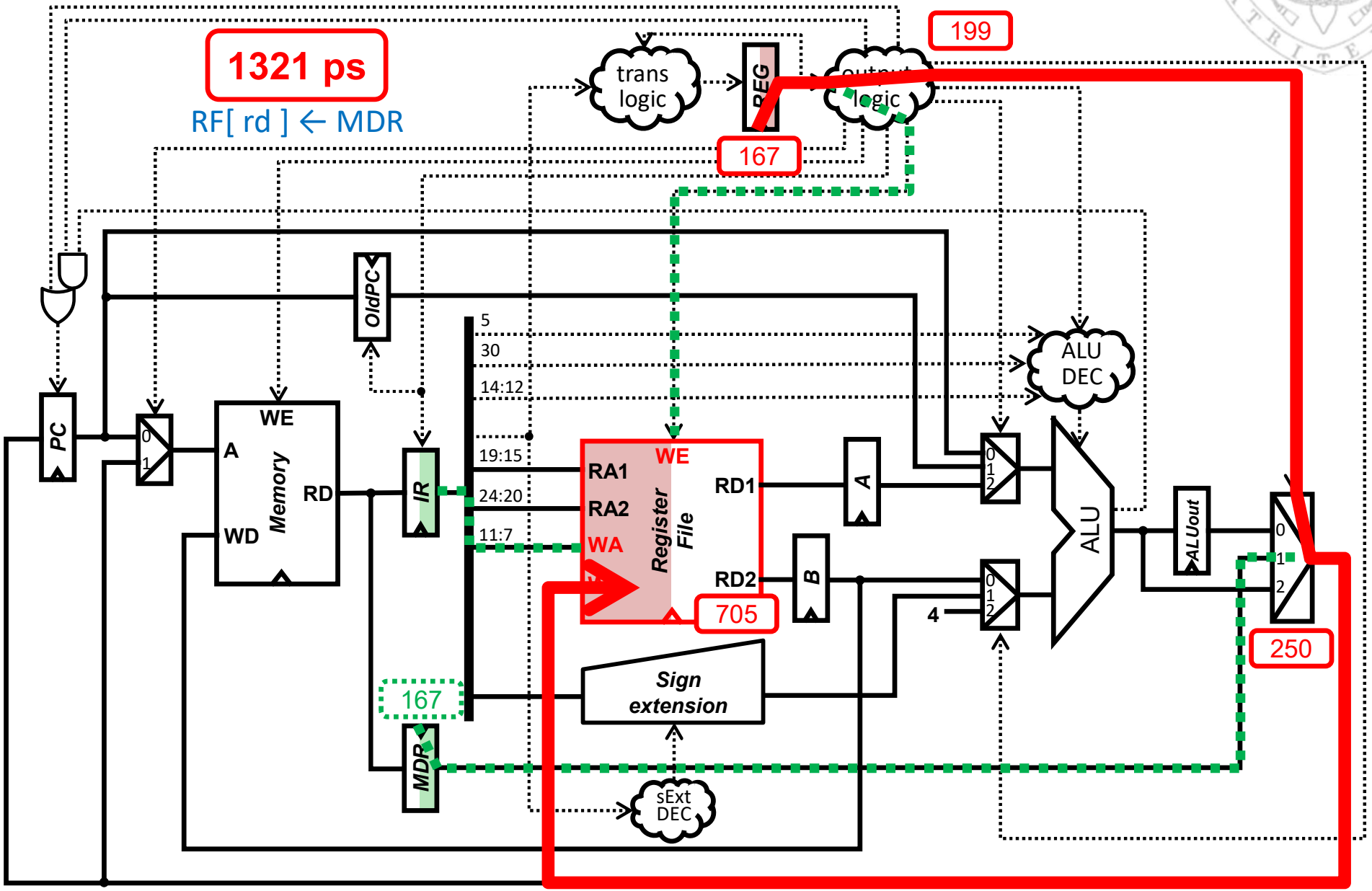
Cycle time calculation

State S3 (memory read): critical path



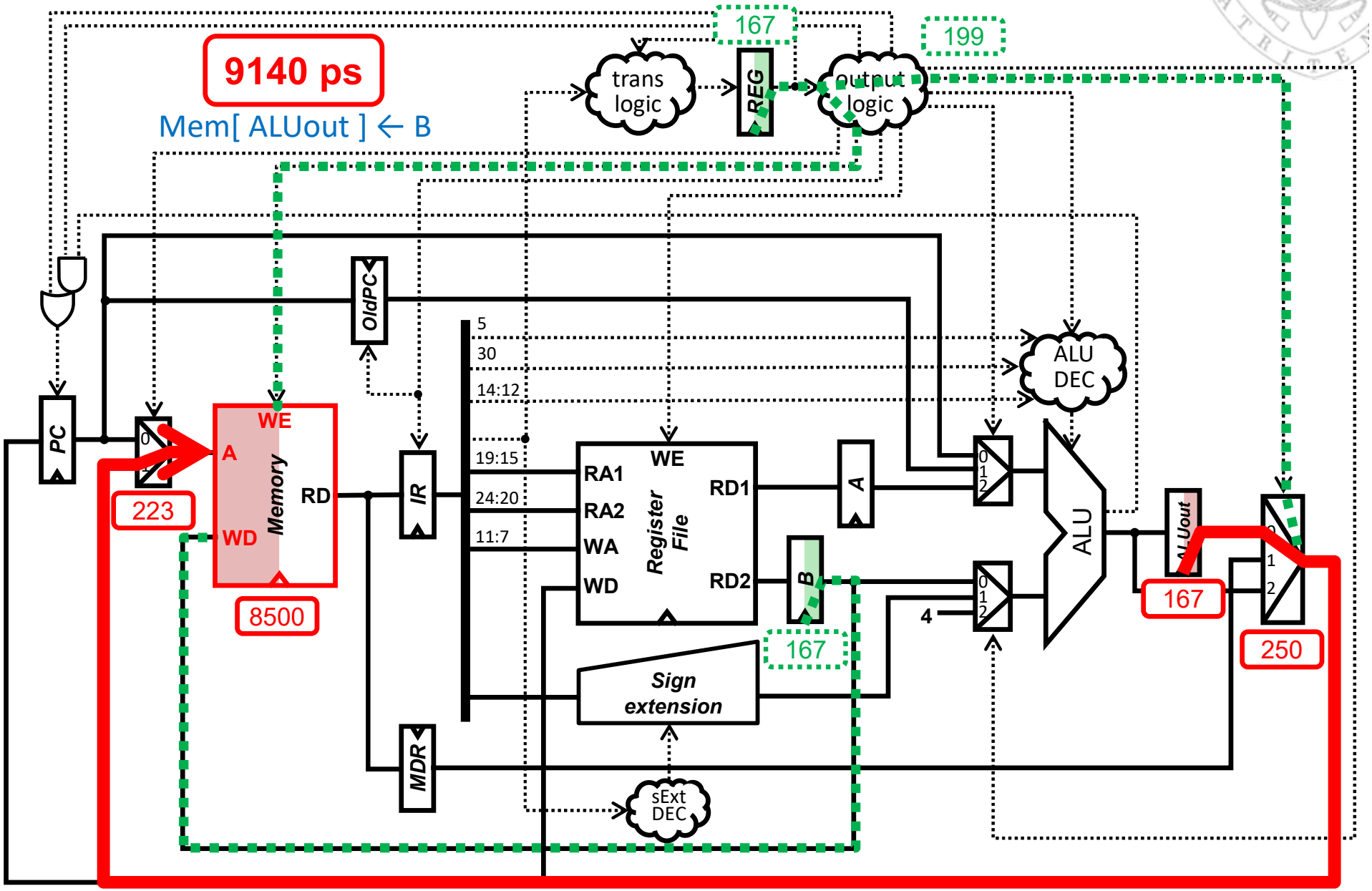
Cycle time calculation

State S4 (RF write): critical path



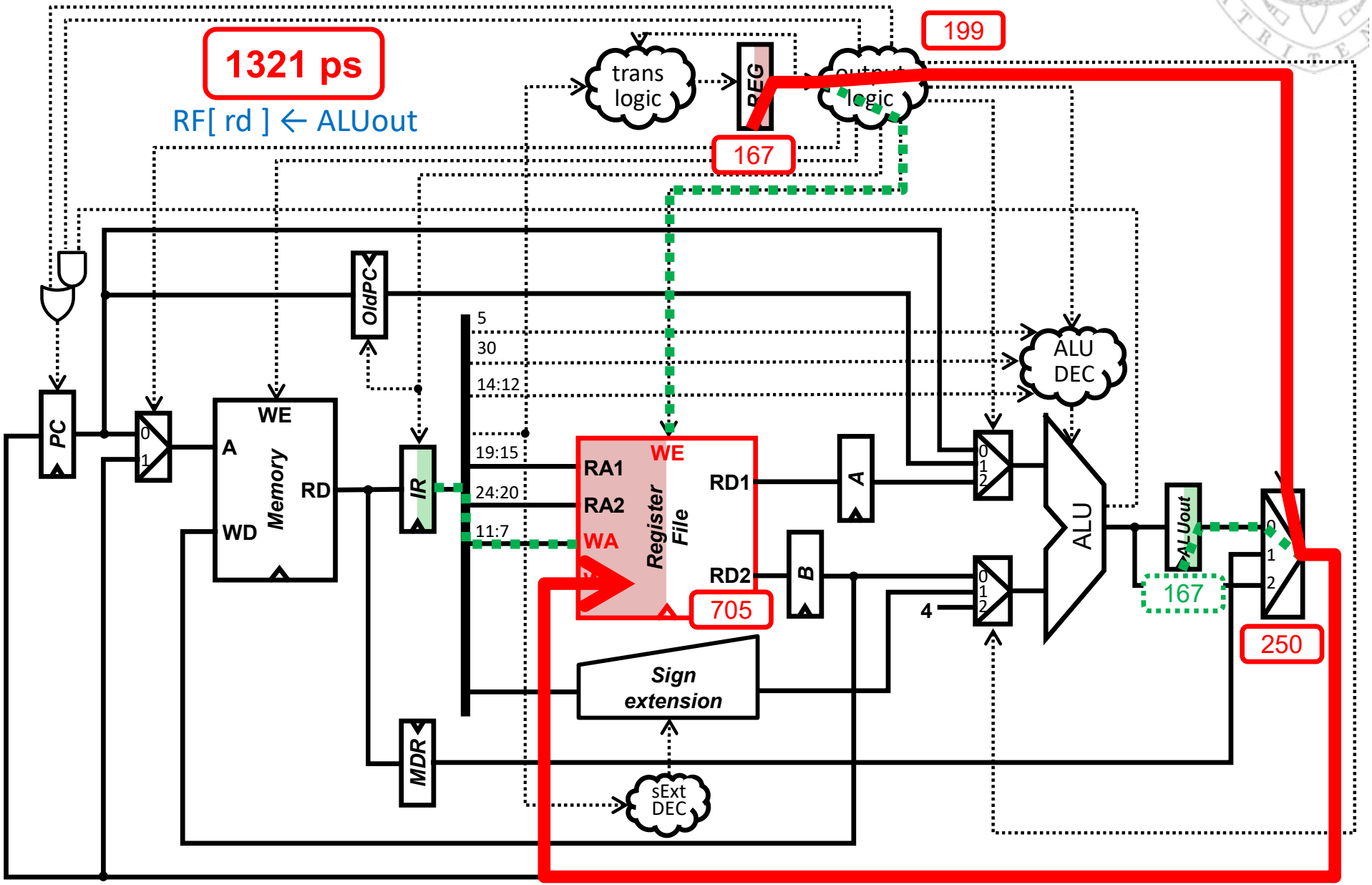
Cycle time calculation

State S5 (memory write): critical path



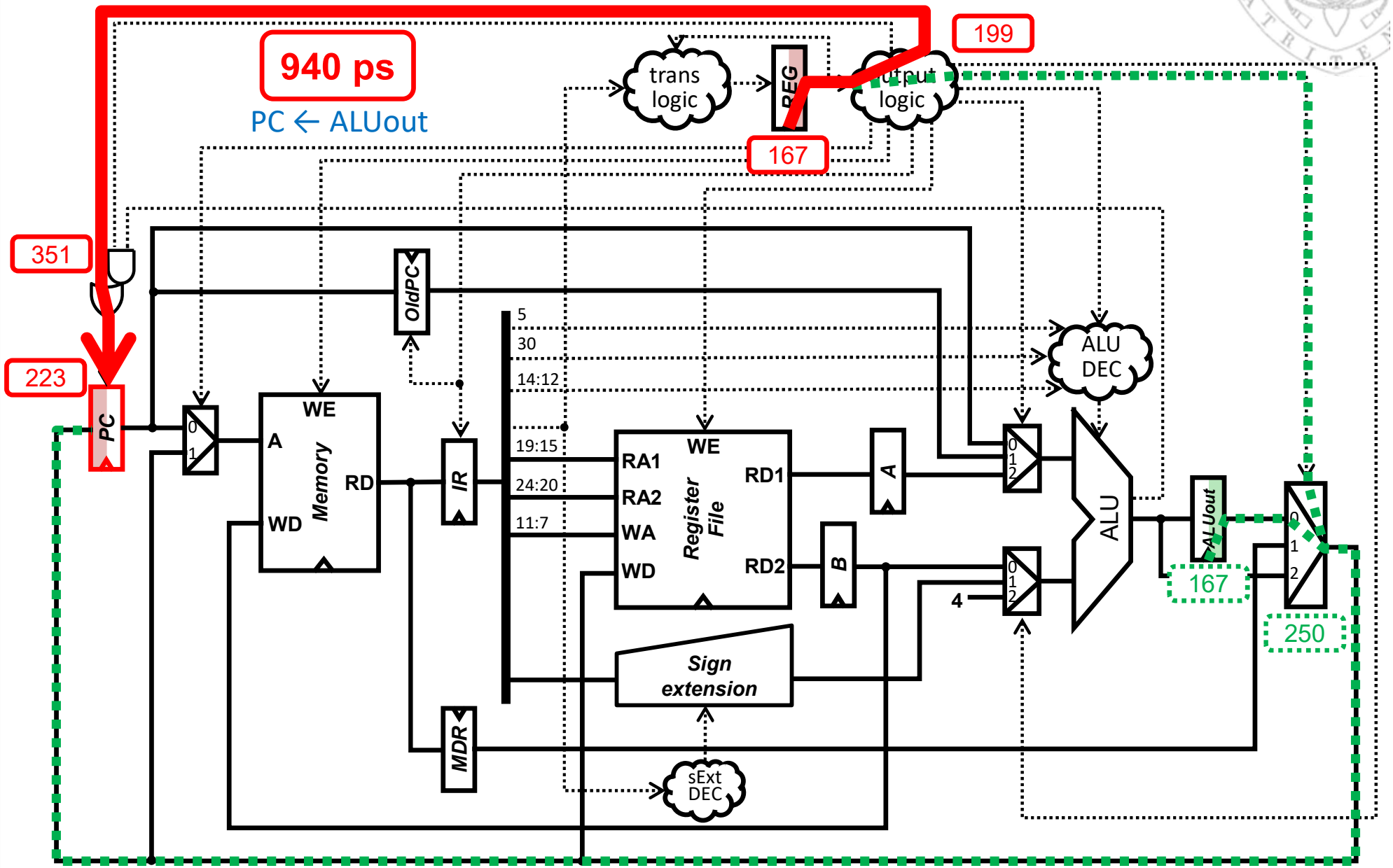
Cycle time calculation

State S7 (RF write): critical path



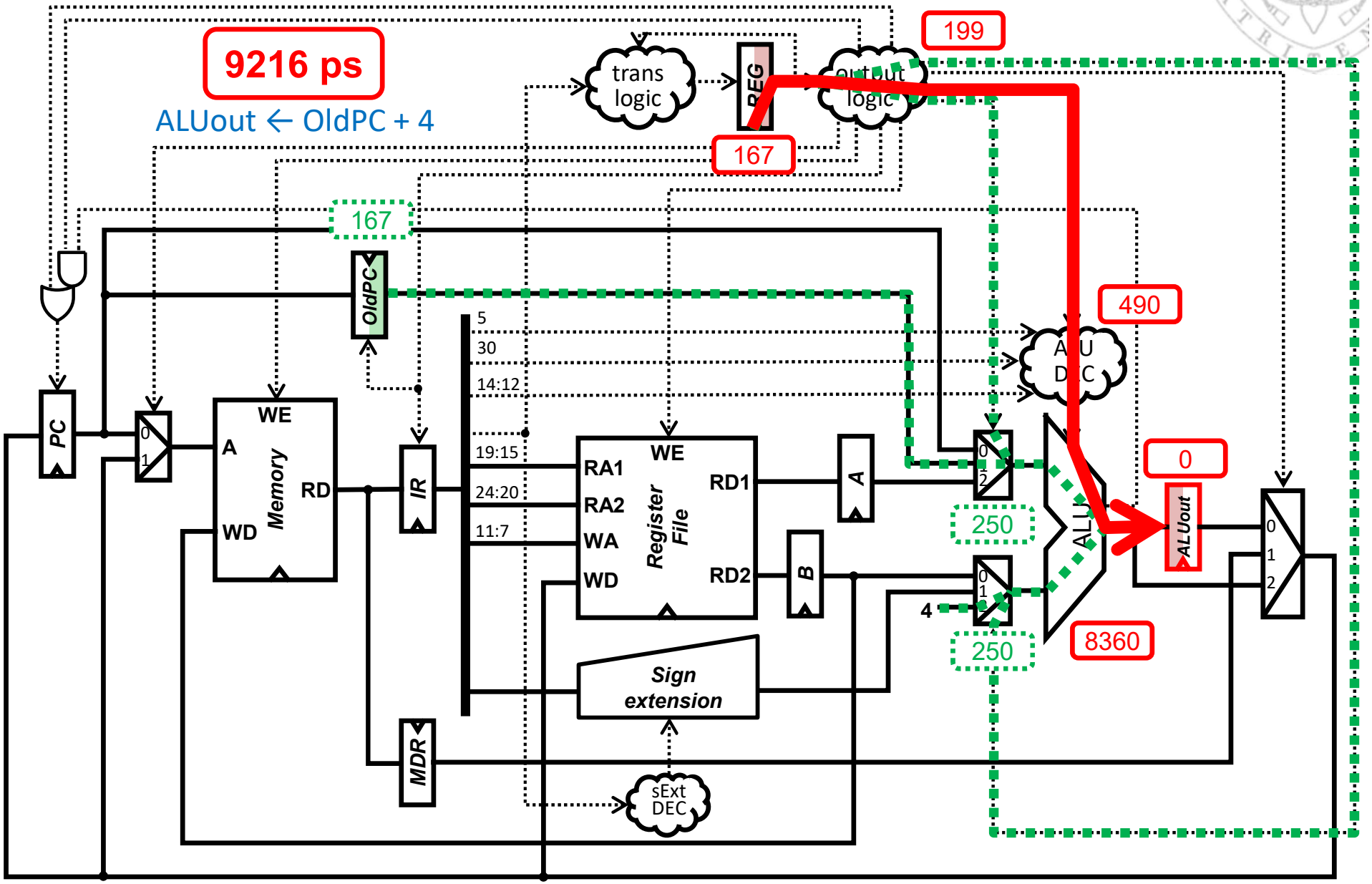
Cycle time calculation

State S9 (PC update): critical path



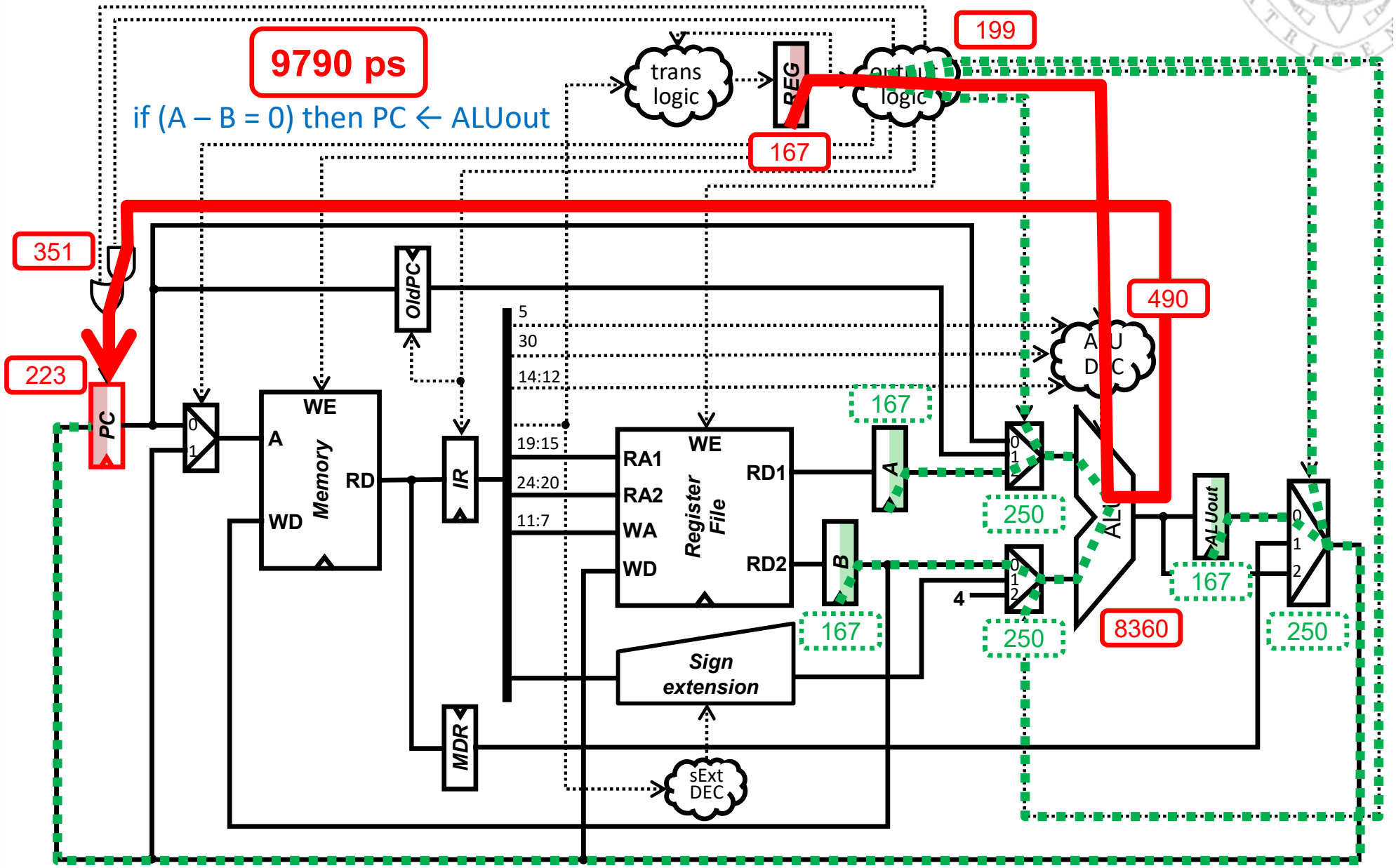
Cycle time calculation

State S9 (return address calculation): critical path



Cycle time calculation

State S10: critical path



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/