ELECTRONICS AND COMMUNICATION TECHNOLOGIES: ELECTRONICS SYSTEMS

LM Cyber Security – Fall 2024

Federico Baronti, Luca Crocetti

Dip. Ing. Informazione

Via G. Caruso, 16 – Stanza B-1-09

050 2217581 – federico.baronti@unipi.it

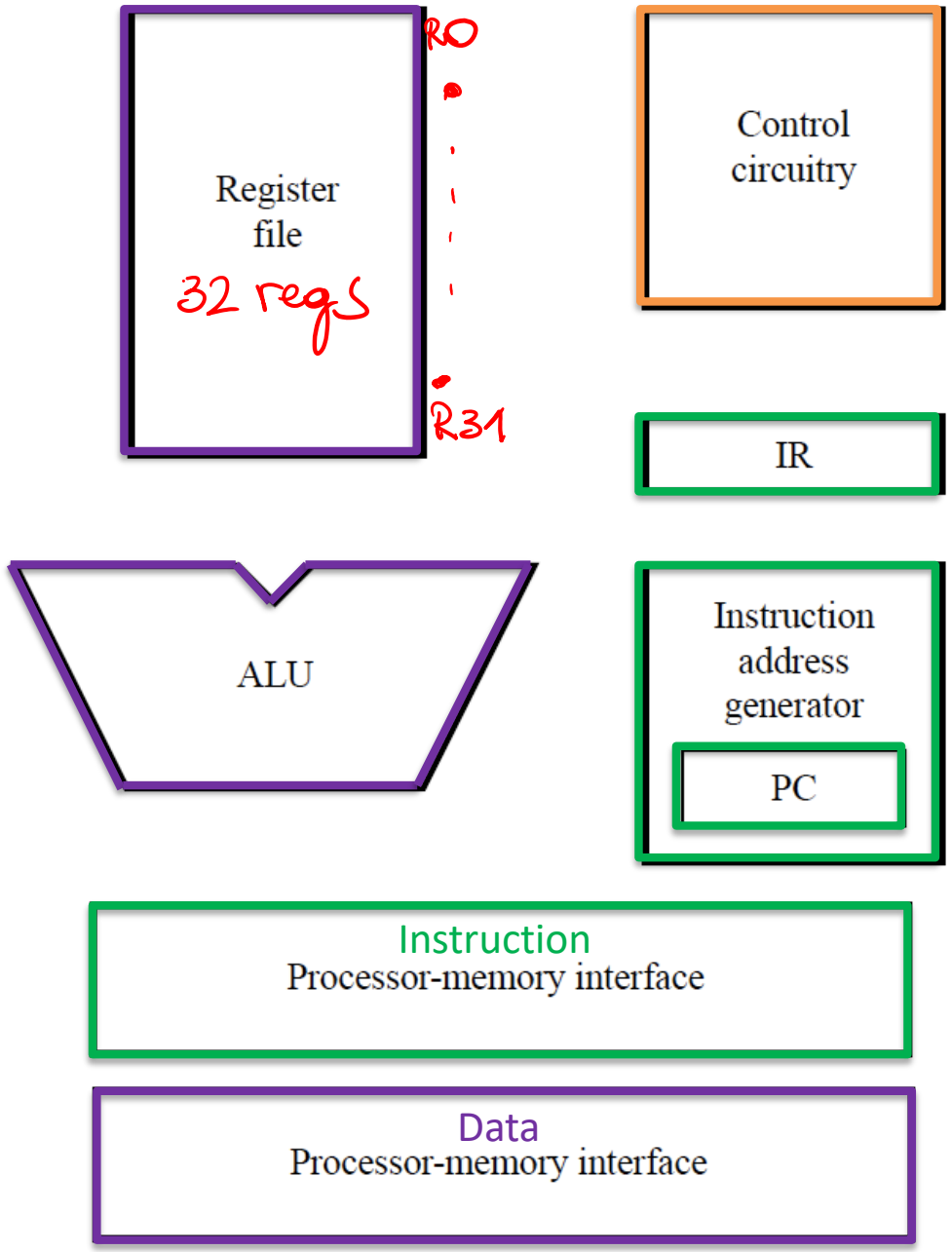
Office hours:

Friday 14-16. Please, contact me in advance before showing up. We can also arrange an appointment remotely on Microsoft Teams.

COMPUTER ORGANIZATION

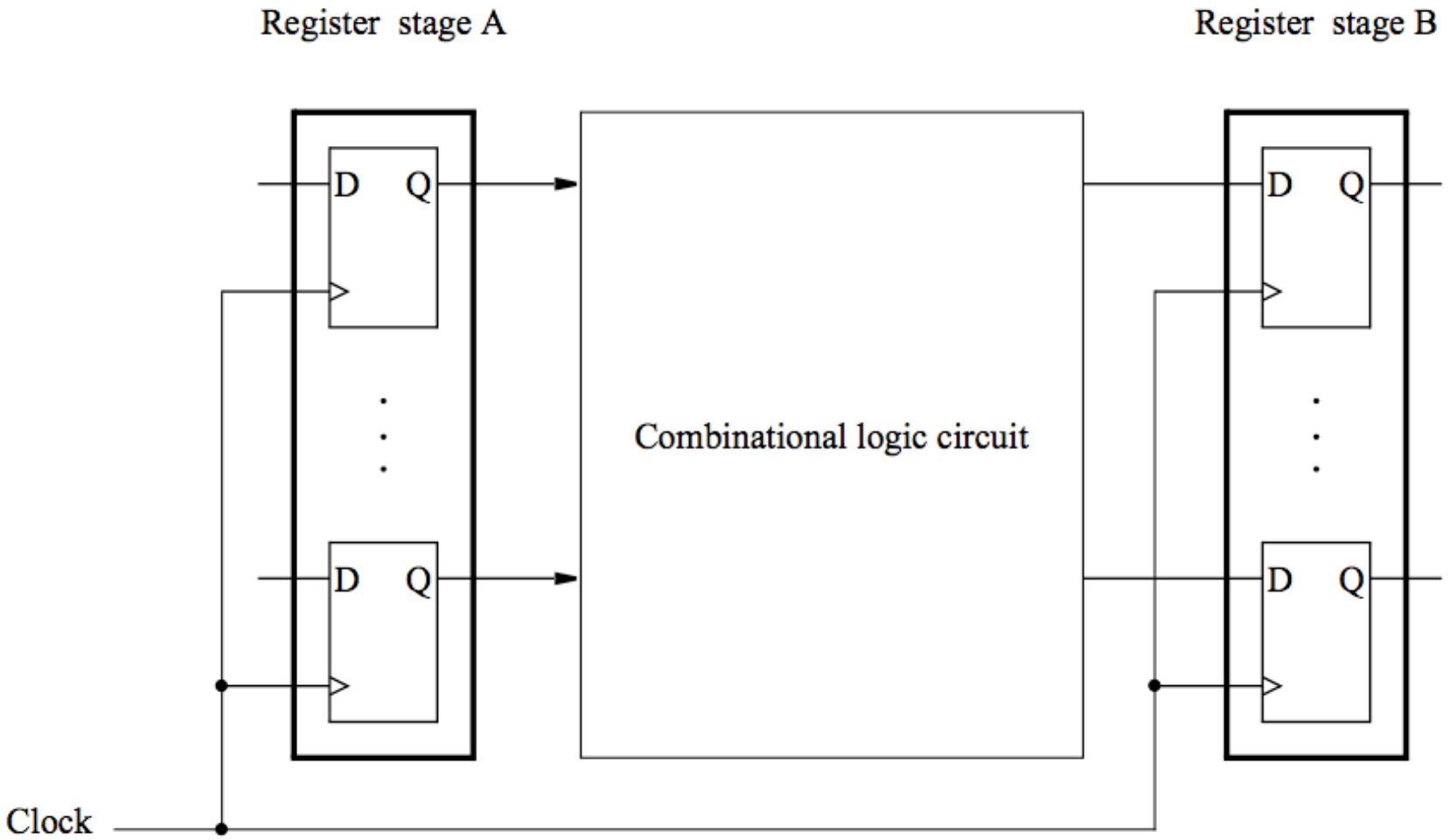
Processor's building blocks

- PC provides instruction address
- Instruction is fetched into IR
- Instruction address generator updates PC
- Instruction in IR is decoded and operands in registers are read
- ALU performs some computation
- For load/store instruction a data memory access is performed
- Control circuitry supervises instruction execution by generating the control signals needed to perform the required actions



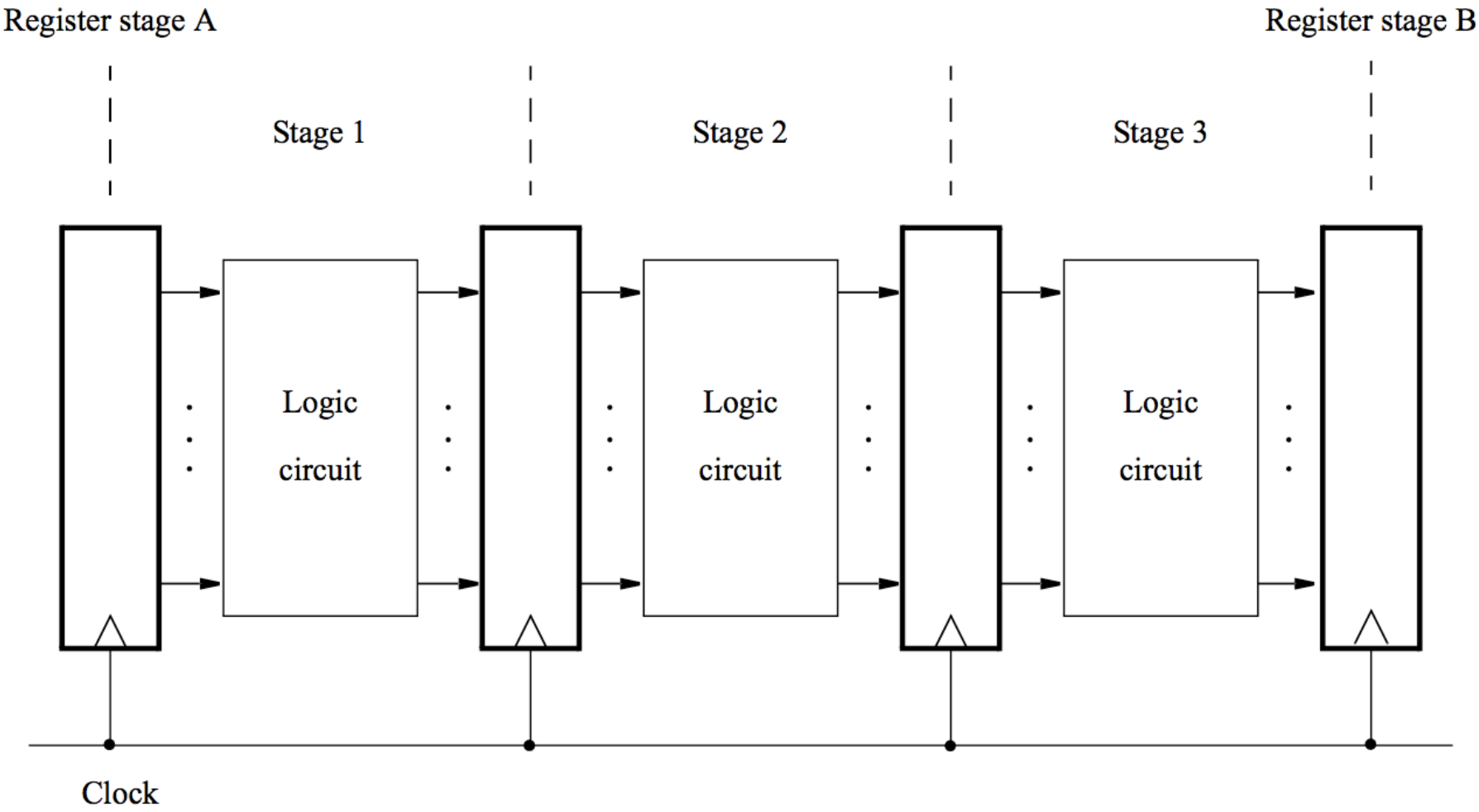
A digital processing system

- datapath



A multi-stage digital processing system

- datapath

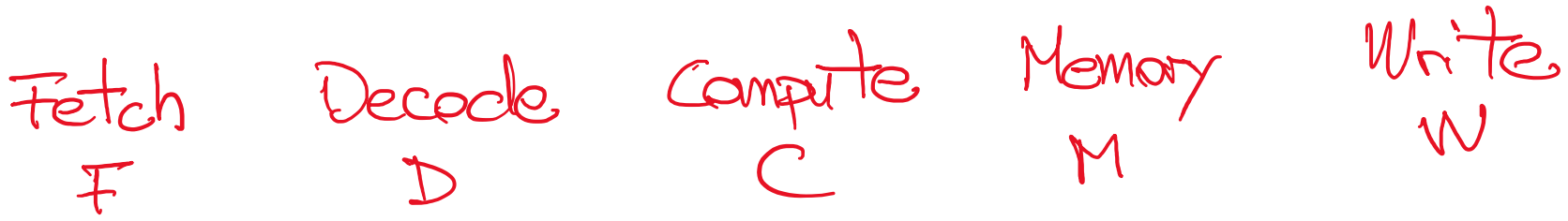


Why multi-stage?

- Processing moves from one stage to the next in each clock cycle
- Such a multi-stage system **is the basis** for pipelined operation
  - High-performance processors have a pipelined organization
  - Pipelining enables the execution of successive instructions to be overlapped
- We will get back to **pipeline** later. **Let's now focus on the basics of the multi-stage architecture of a RISC-style processor**

Instruction execution

- Pipelined organization is most effective if all instructions can be executed in the same number of steps.
- Each step is carried out in a separate hardware stage.
- Processor design will be illustrated using **five** hardware stages.
- **How can instruction execution be divided into five steps?**
  - Let's start from some representative RISC instructions



A memory access instruction:

signed extension to 32 bits

I-type

ldw R5, X(R7) R5 ← Mem32[X + R7]

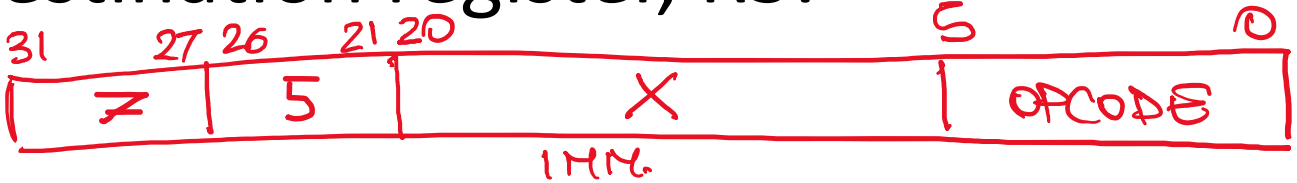
F 1. Fetch the instruction and increment the program counter.

D 2. Decode the instruction and read the contents of register R7 in the register file.

C 3. Compute the effective address = X + [R7].

M 4. Read the memory source operand at effective address.

W 5. Load the operand read from memory into the destination register, R5.



A computational instruction:

R-type

add R3, R4, R5 ; R3 ← R4+R5

F 1. Fetch the instruction and increment the program counter.

D 2. Decode the instruction and read registers R4 and R5.

C 3. Compute the sum [R4] + [R5].

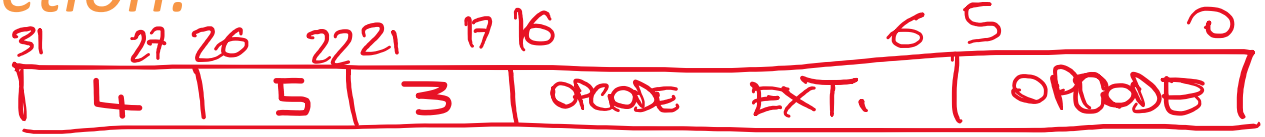
M 4. No action.

ALU

CN is a short circuit

W 5. Load the result into the destination register, R3.

- *Stage 4 (memory access) is not involved in this instruction.*



5-stage Architecture of a RISC Processor

F 1. Fetch an instruction and increment the program counter.

D 2. Decode the instruction and read registers from the register file.

C 3. Perform an ALU operation.

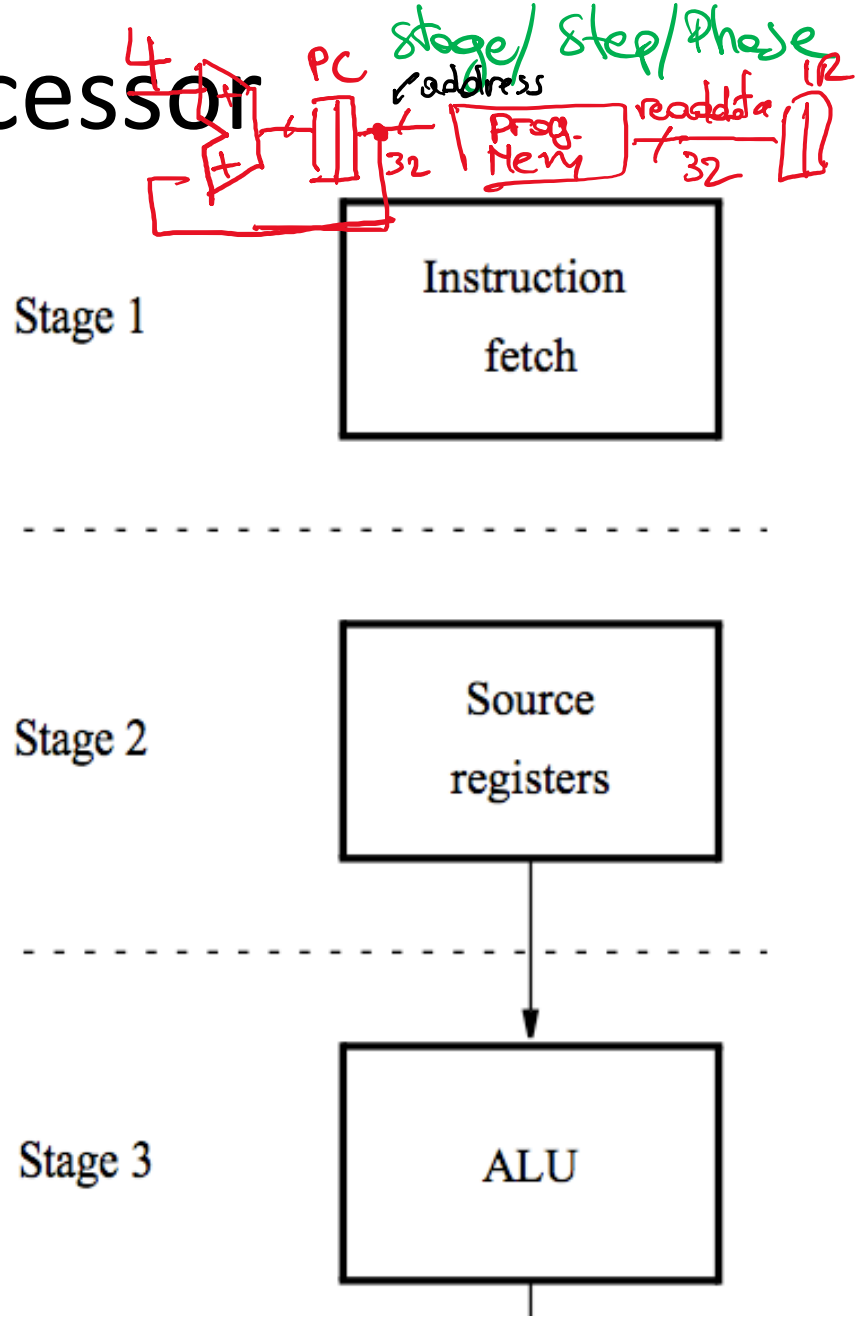
M 4. Read or write memory data if the instruction involves a memory operand.

W 5. Write the result into the destination register.

- This sequence determines the hardware stages needed.

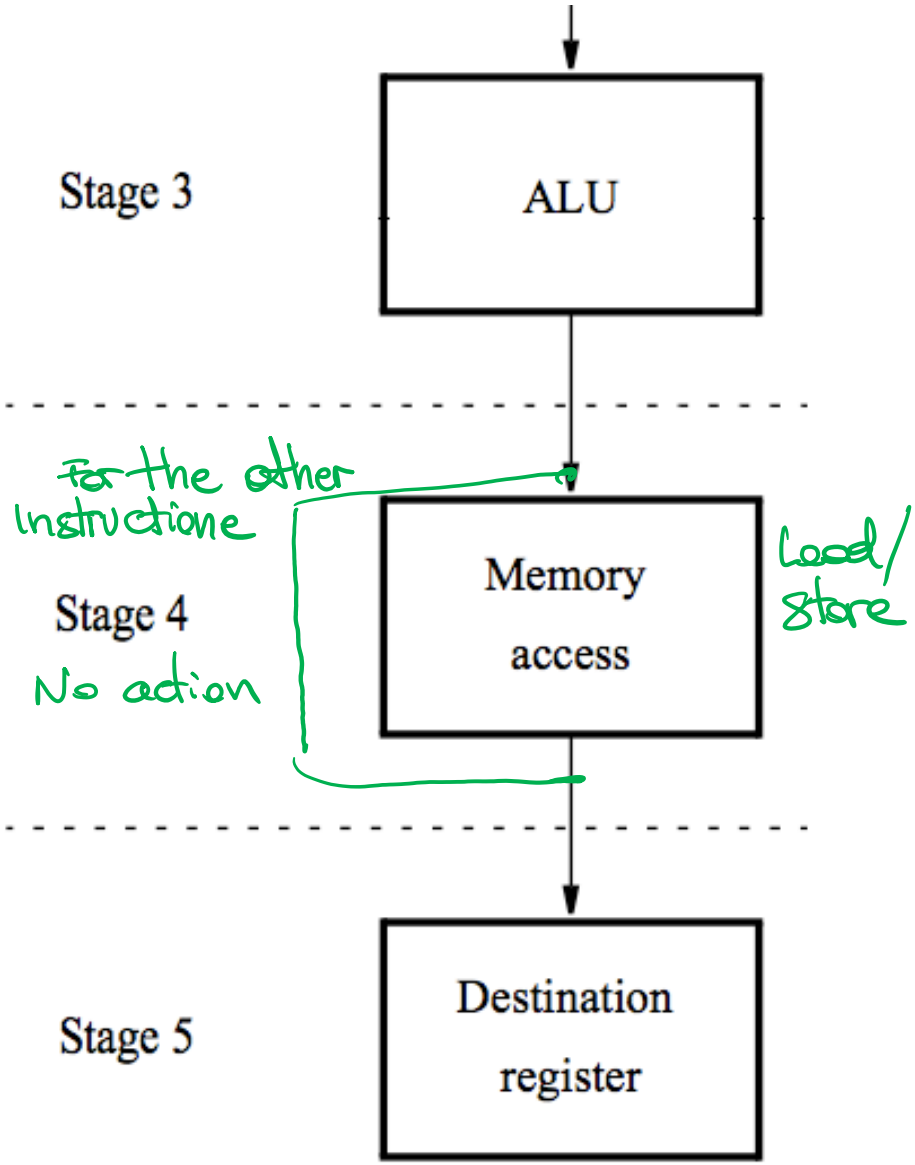
A 5-stage implementation of a RISC processor

- Instruction processing moves from stage to stage in every clock cycle, starting with **fetch**.
- The instruction is **decoded** and the source registers are read in stage 2.
- **Computation** takes place in the ALU in stage 3.



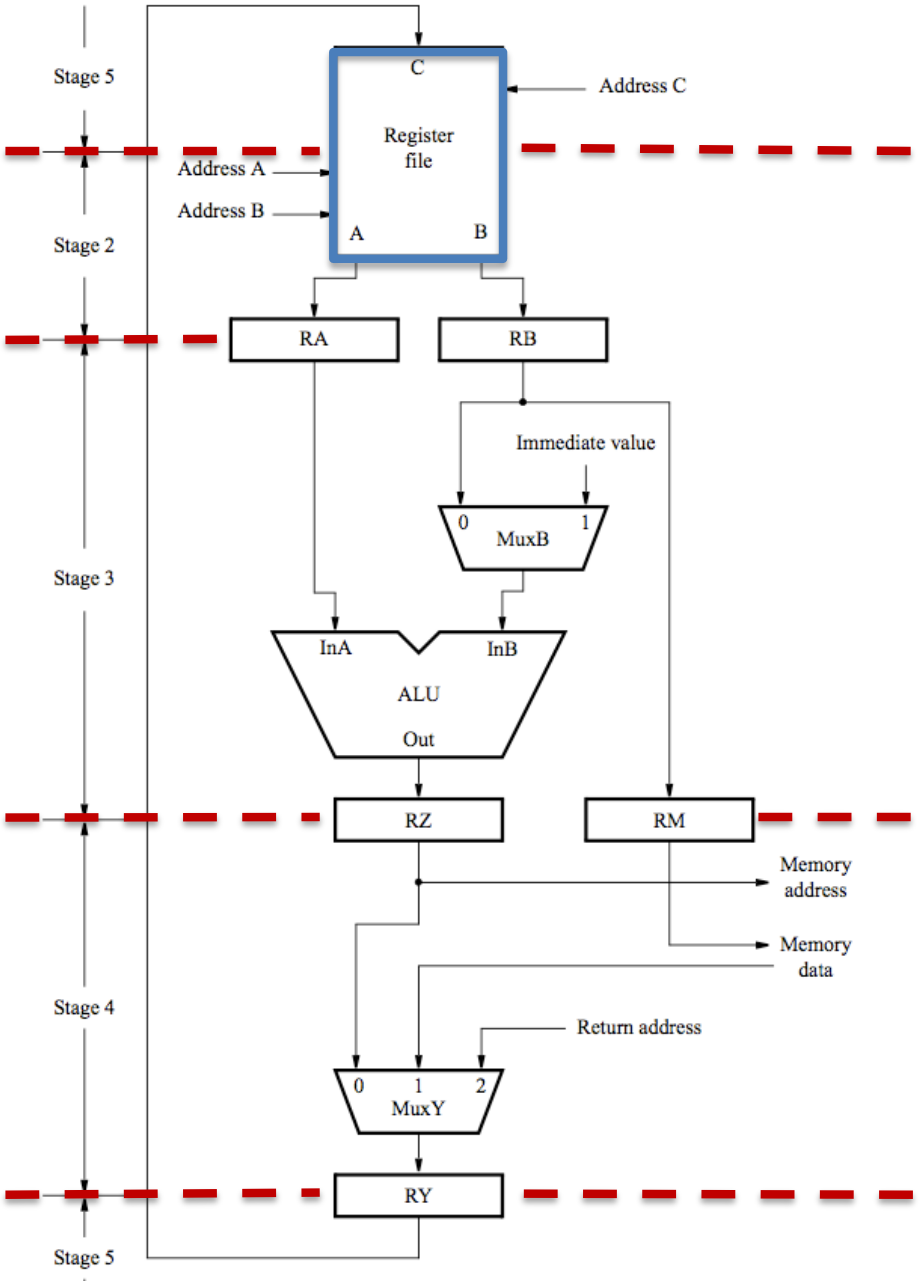
A 5-stage implementation of a RISC processor

- …
- If a **memory** operation is involved, it takes place in stage 4.
- The result (if any) of the instruction is **written** in the destination register in stage 5.



The datapath – Stages 2 to 5

- Register file,
  used in stages 2 and 5
  - (Inter-stage registers RA, RB, RZ, RM, RY needed to carry data from one stage to the next)

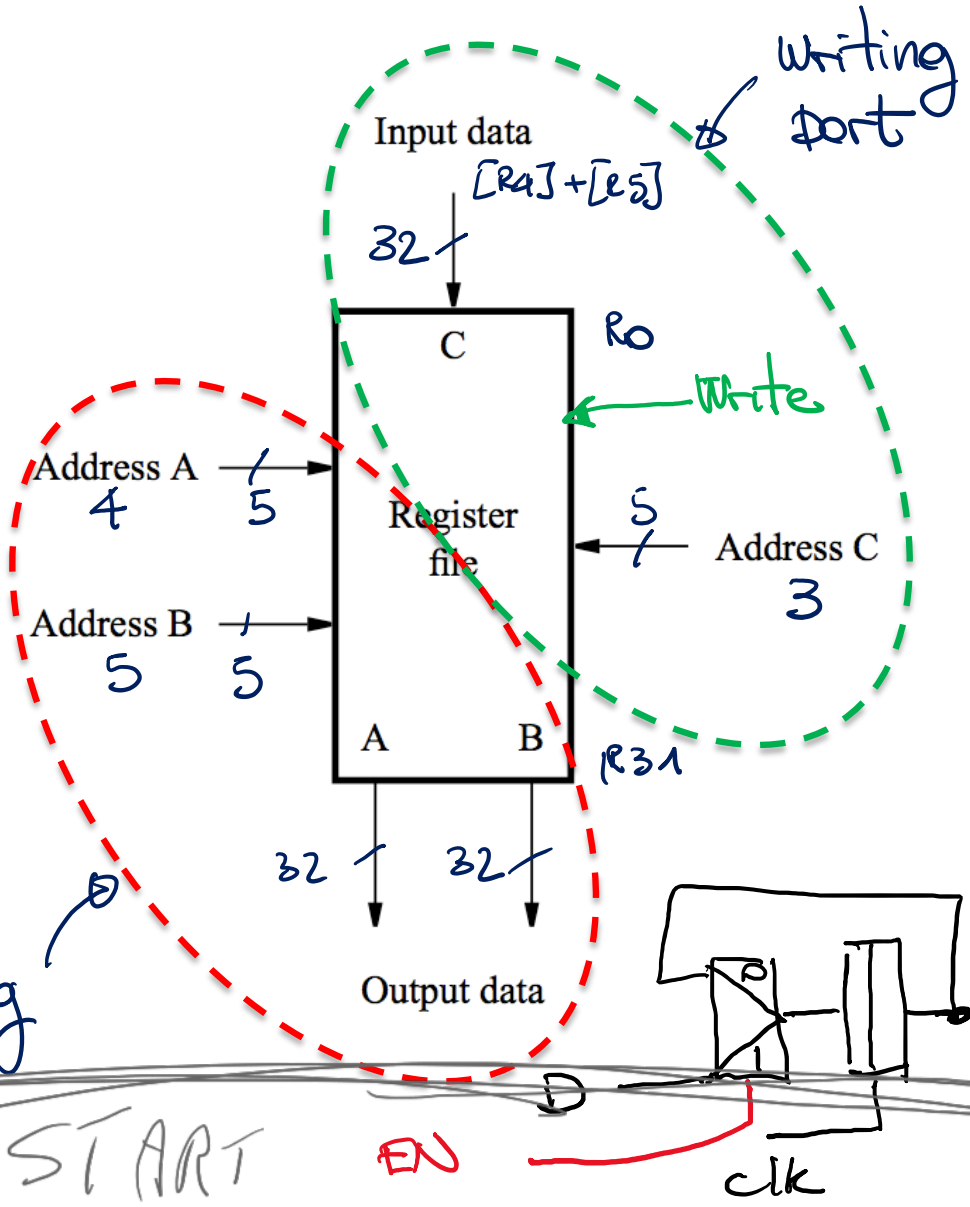


32 REGISTERS

Hardware components: Register file

is involved in 2 phases: DECODE and WRITE (may be)

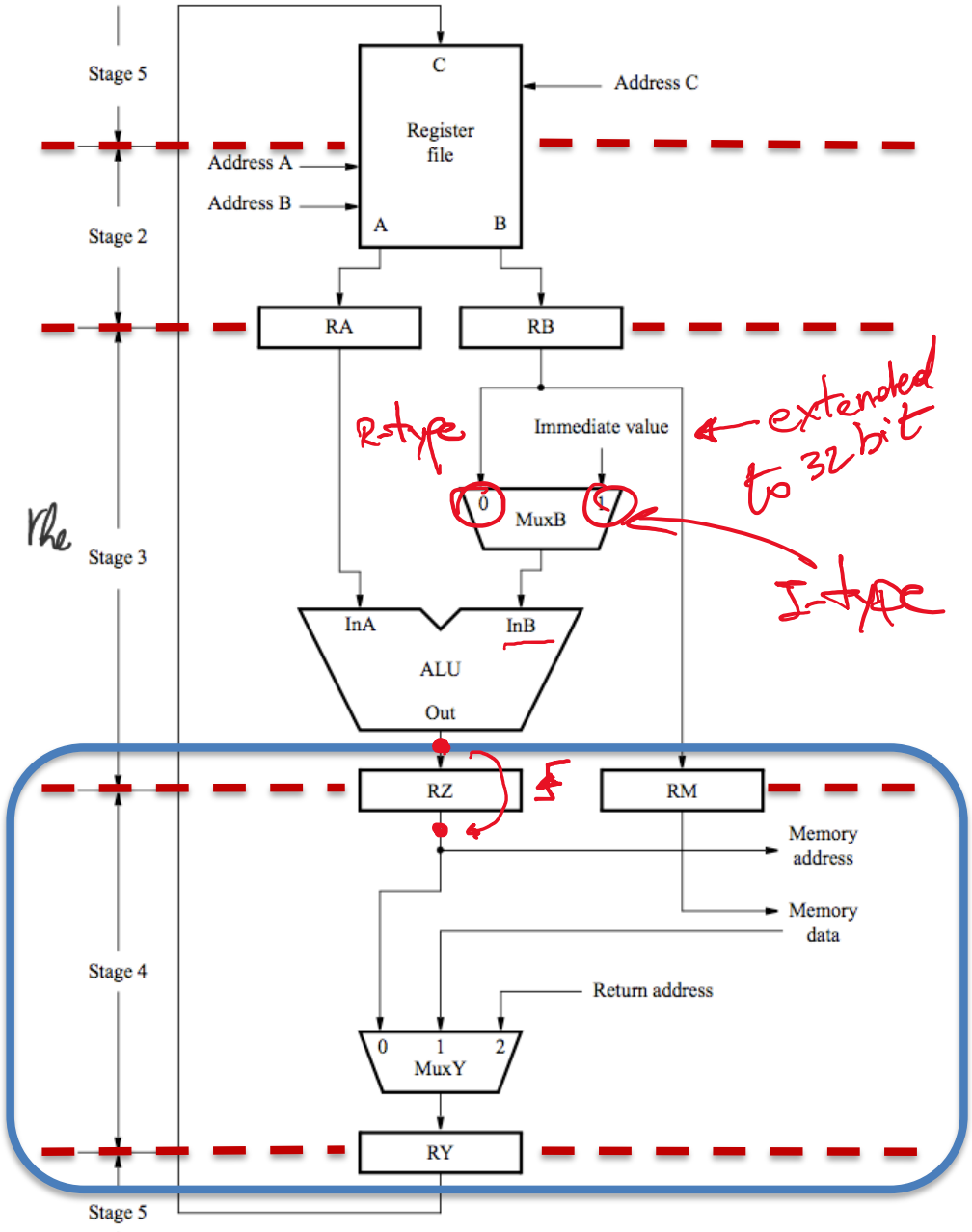
- A 2-port register file is needed to read the two source registers at the same time.
- It may be implemented using a 2-port memory.



Eg. ADD R3, R4, R5

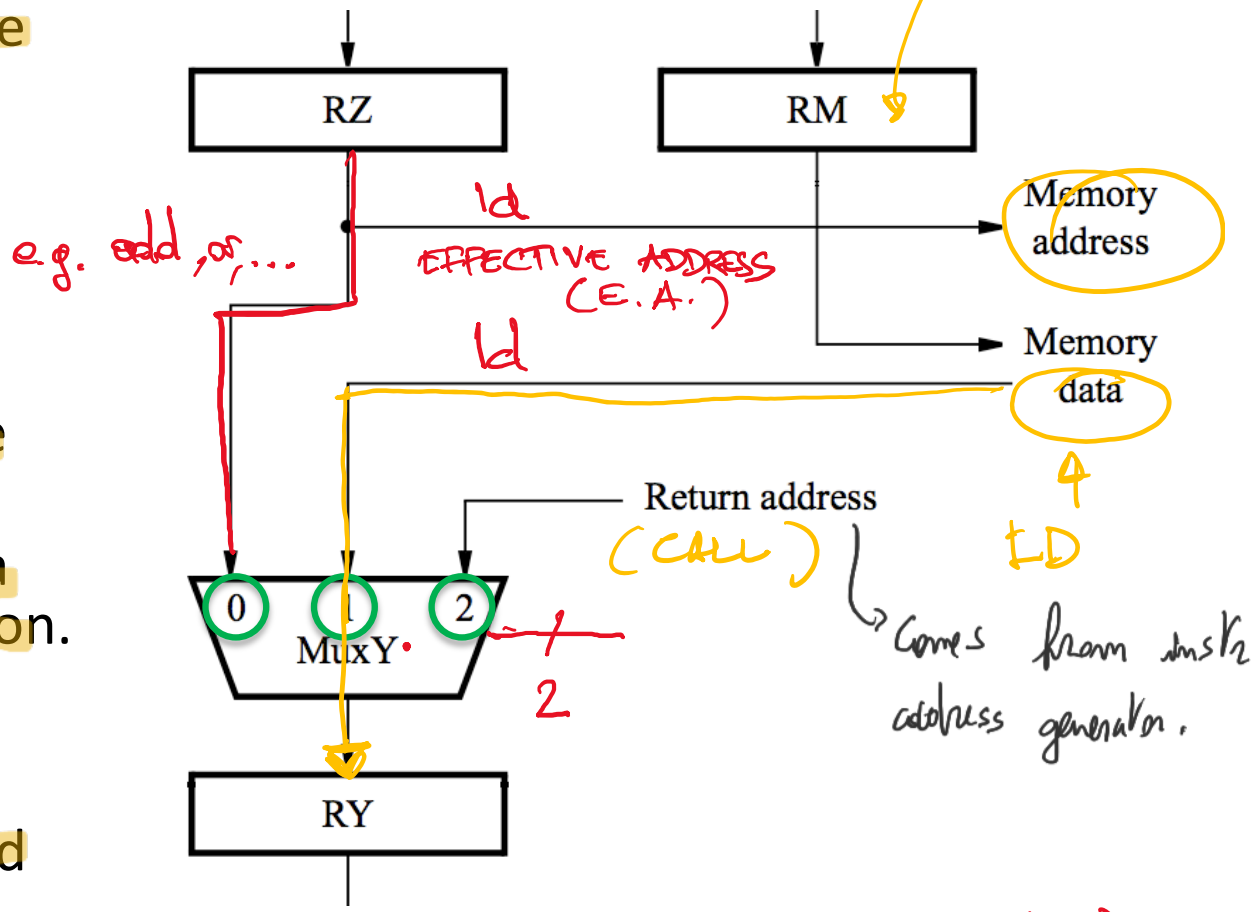
The datapath – Stages 2 to 5

- Register file,
  used in stages 2 and 5
  - (Inter-stage registers RA, RB, RZ, RM, RY needed to carry data from one stage to the next)
- ALU stage
- Memory stage
- Final stage to store result to the register file



Memory stage

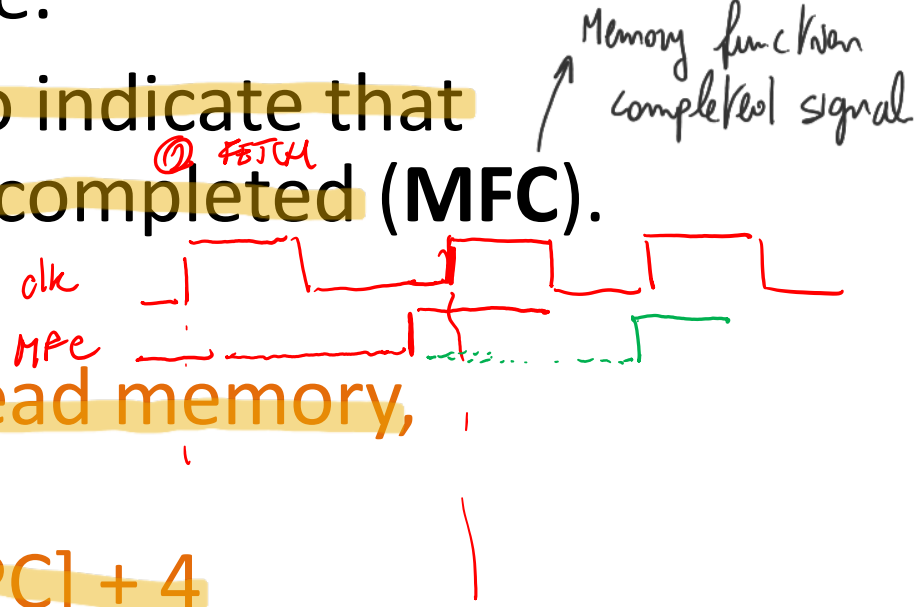
- **For a calculation instruction:**
  - MuxY selects [RZ] to be placed in RY.
- **For a memory instruction:**
  - RZ provides memory address, and MuxY selects read data to be placed in RY.
  - RM provides data for a memory write operation.
- **In subroutine calls or exception handling:**
  - Input 2 of MuxY is used to save the return address in the link register



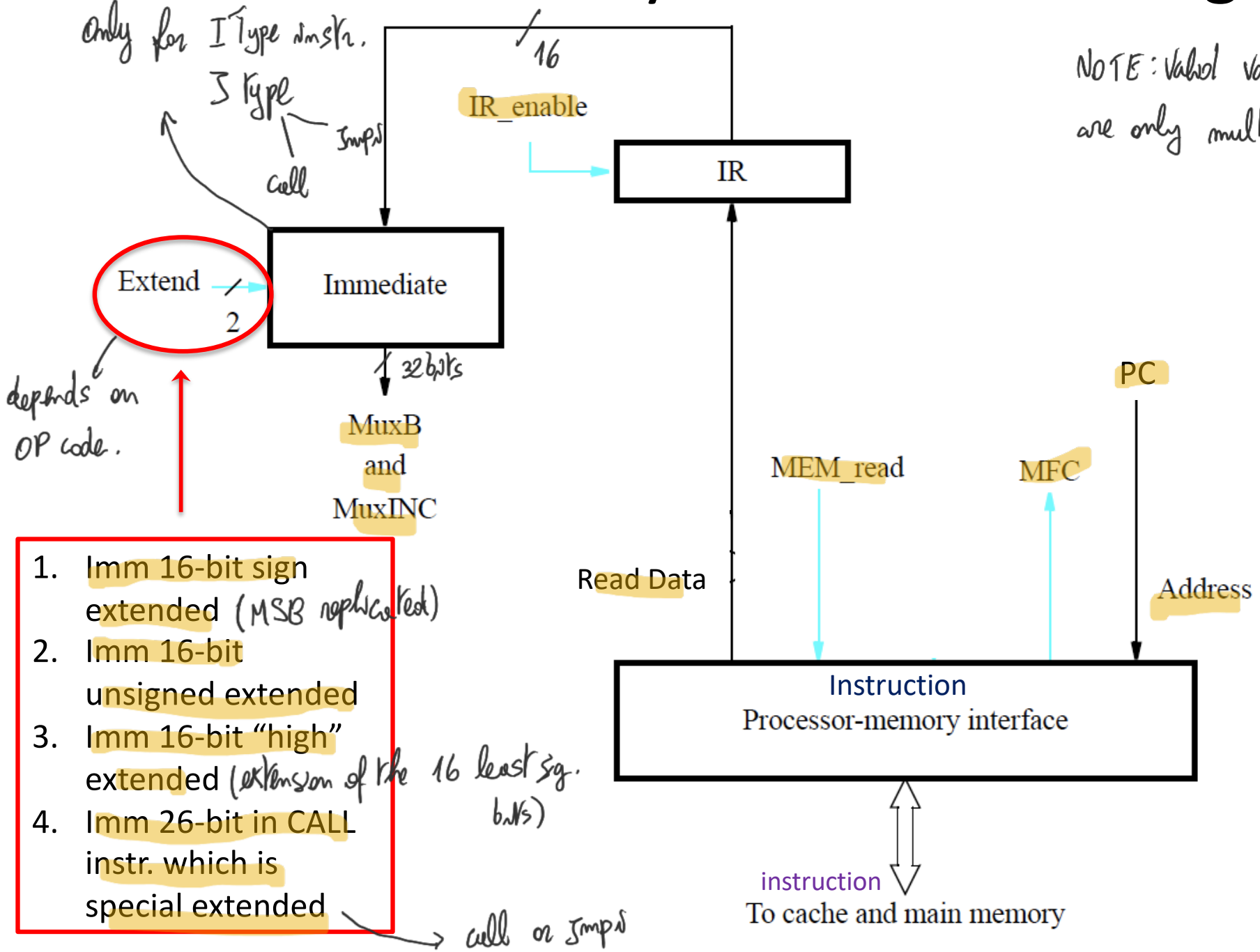
Memory access

- When data are found in the cache, access to memory can be completed in one clock cycle.
- Otherwise, read and write operations may require several clock cycles to load data from main memory into the cache.
- A control signal is needed to indicate that memory function has been completed (**MFC**).
  E.g., for step 1:

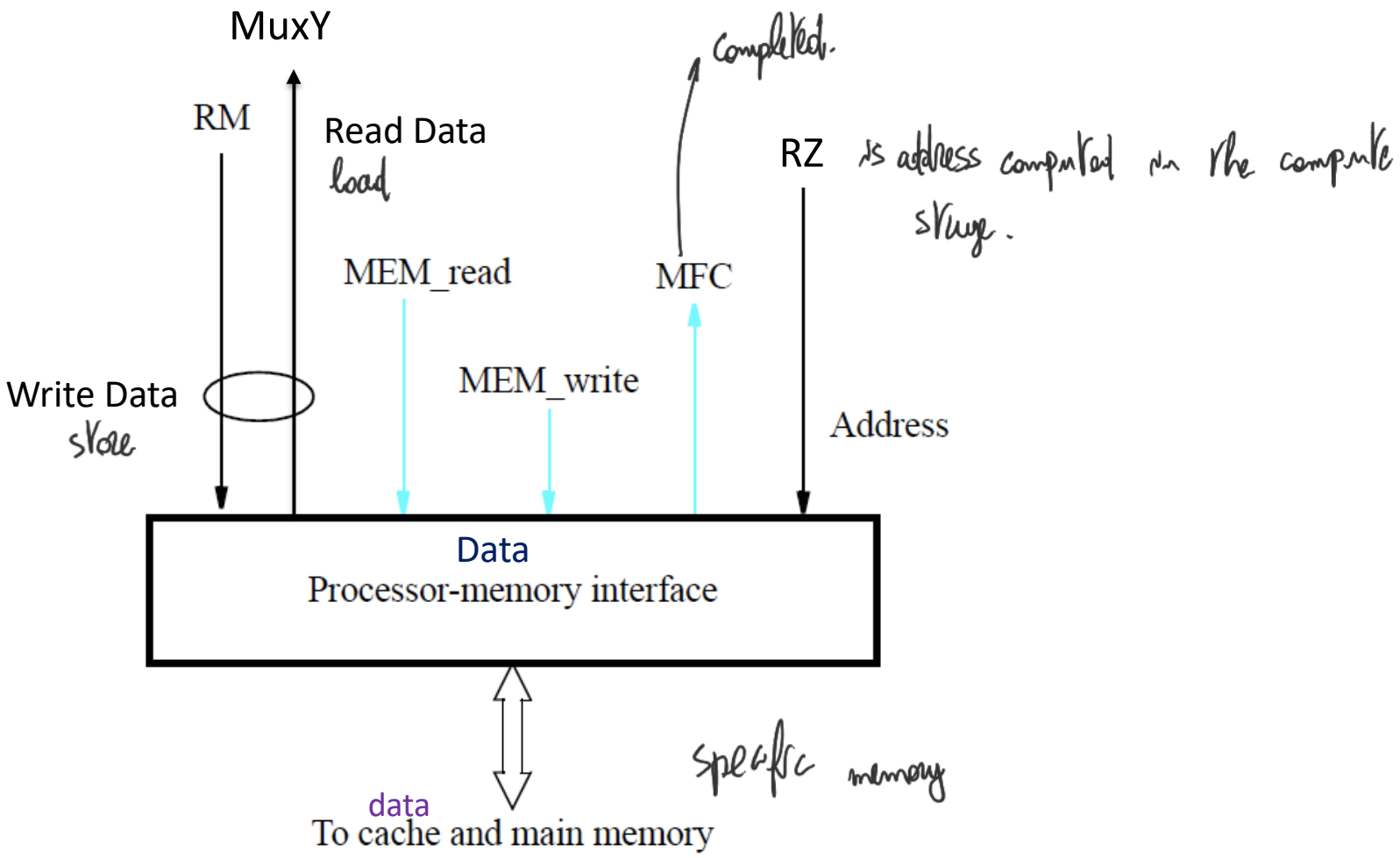
1. Memory address ← [PC], Read memory,
   Wait for MFC,
   IR ← Memory data, PC ← [PC] + 4



Instruction memory and IR control signals

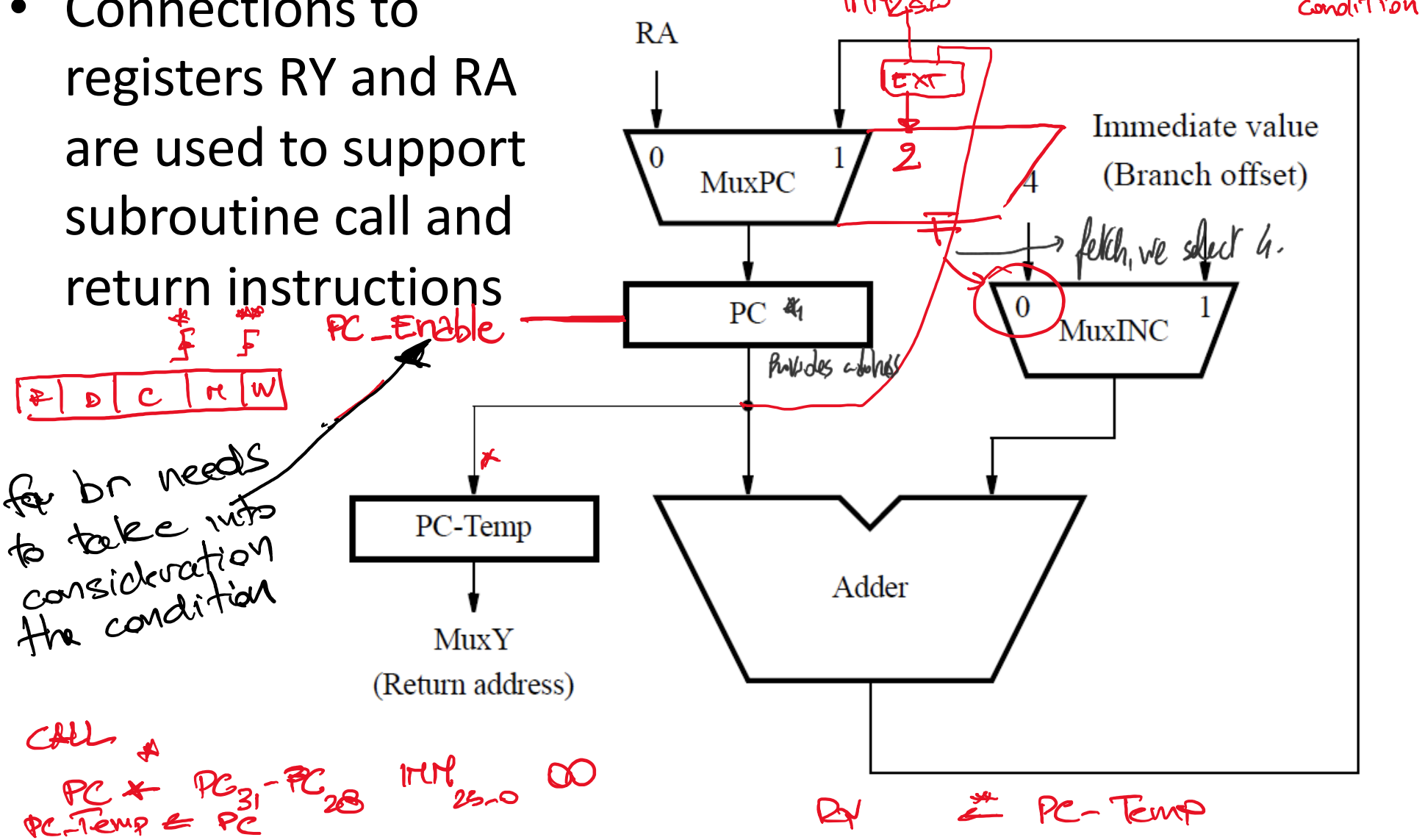


Data memory interface



Instruction address generator

- Connections to registers RY and RA are used to support subroutine call and return instructions



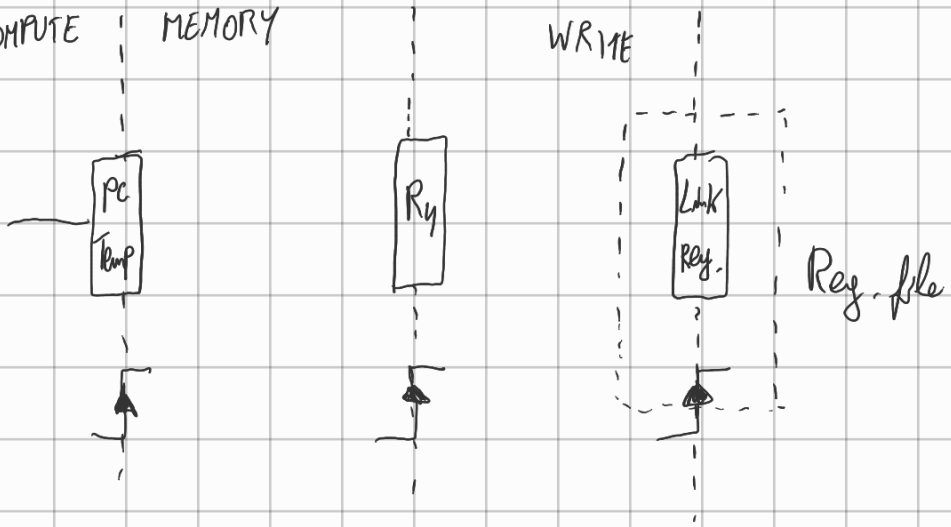
Activated on the fetch phase.
* When MFC is high, the PC is enabled to take the next value.

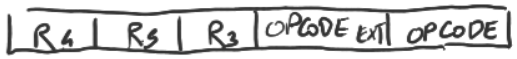
But during compute stage this piece of HW is activated by instructions that change the value of PC: call, Jmp, branch, return.

CALL: places the extension of the value in the PC.
CALLR: address of next instr. is the content of a register. Rm is read and placed in RA.
Same for the return. For return we take the value in the link reg. (r31) and put it in RA.
JMP similar to CALLR, JMI TO CALL.

Unconditional branch: we have immediate value sign extended.
If branch depends on condition, during compute ALU is evaluating. So if we need to do the Jump we enable the register. Otherwise not.

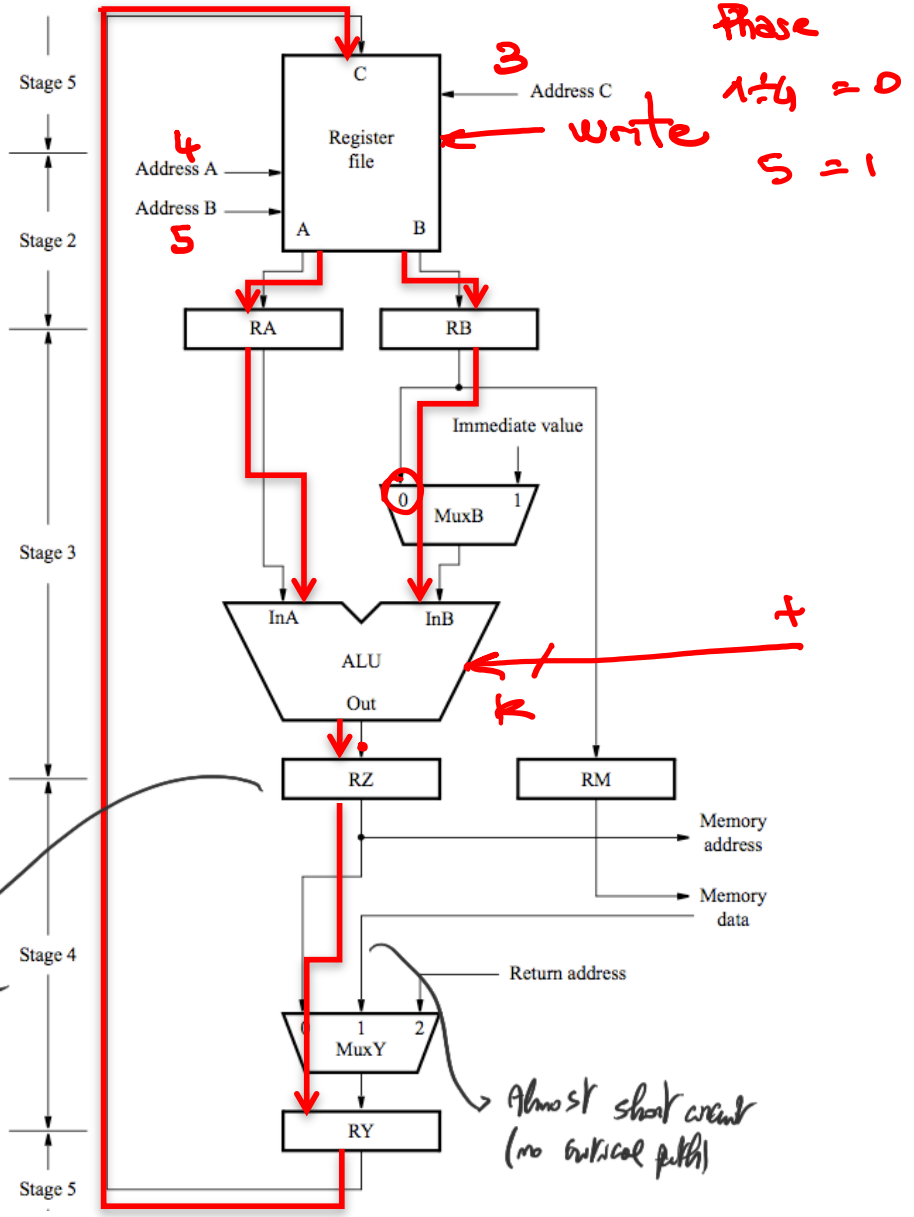
NOTE: Call, CallR, we need somewhere to store the return address. When the edge comes when we update PC, we also need to store old value.
Return address is stored in link register.
PC-Temp: between compute and memory. During memory, value through Mux Y goes into Ry, that contains the return address,





R-type Example: add R3, R4, R5

1. Memory address ←[PC], Read memory, Wait for MFC, IR←Memory data, PC ← [PC] + 4
2. Decode instruction, RA ← [R4], RB ← [R5]
3. RZ ← [RA] + [RB]
4. RY ← [RZ]
5. R3 ← [RY]



R0 = 0

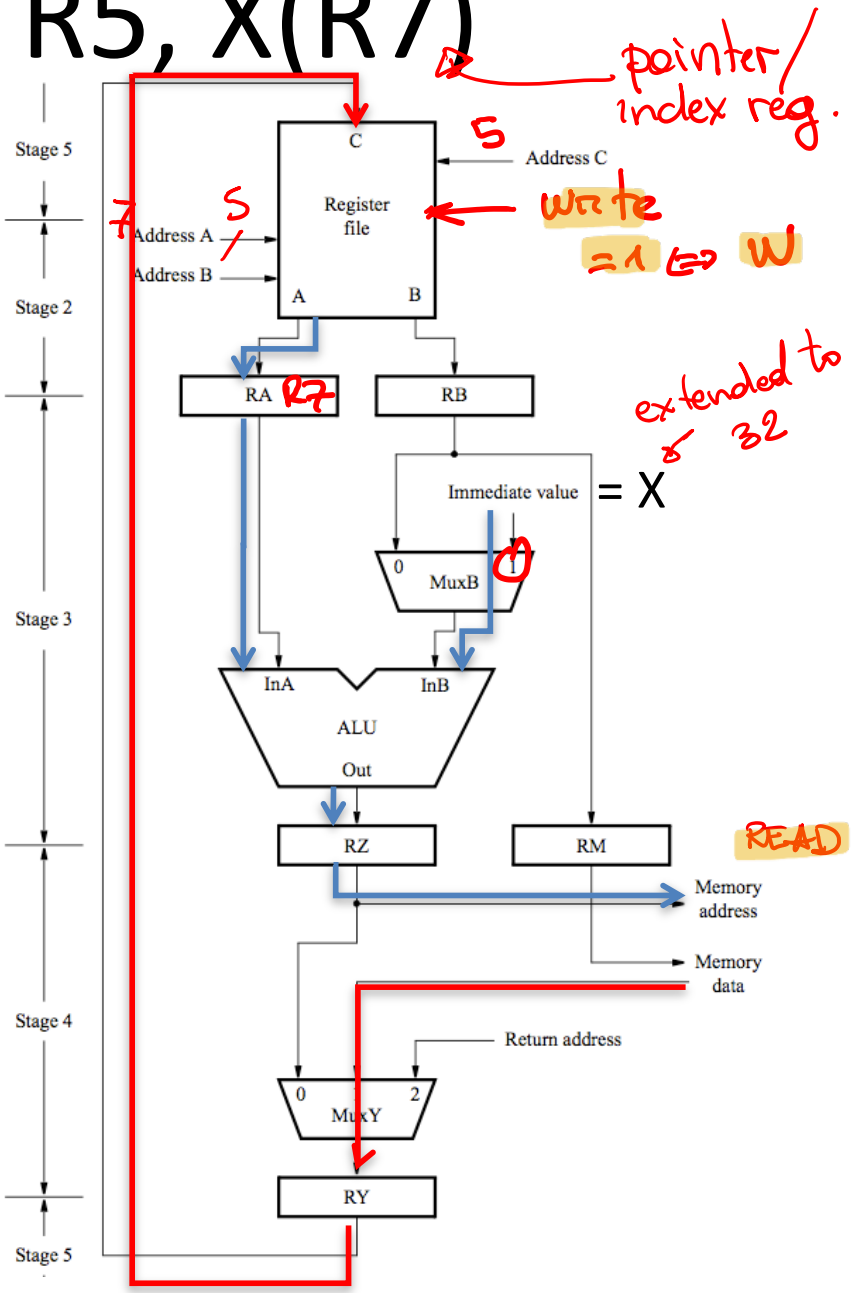
I-type

R7 | R5 | IMMEDIATE | OP CODE

Displacement (16 bits signed)

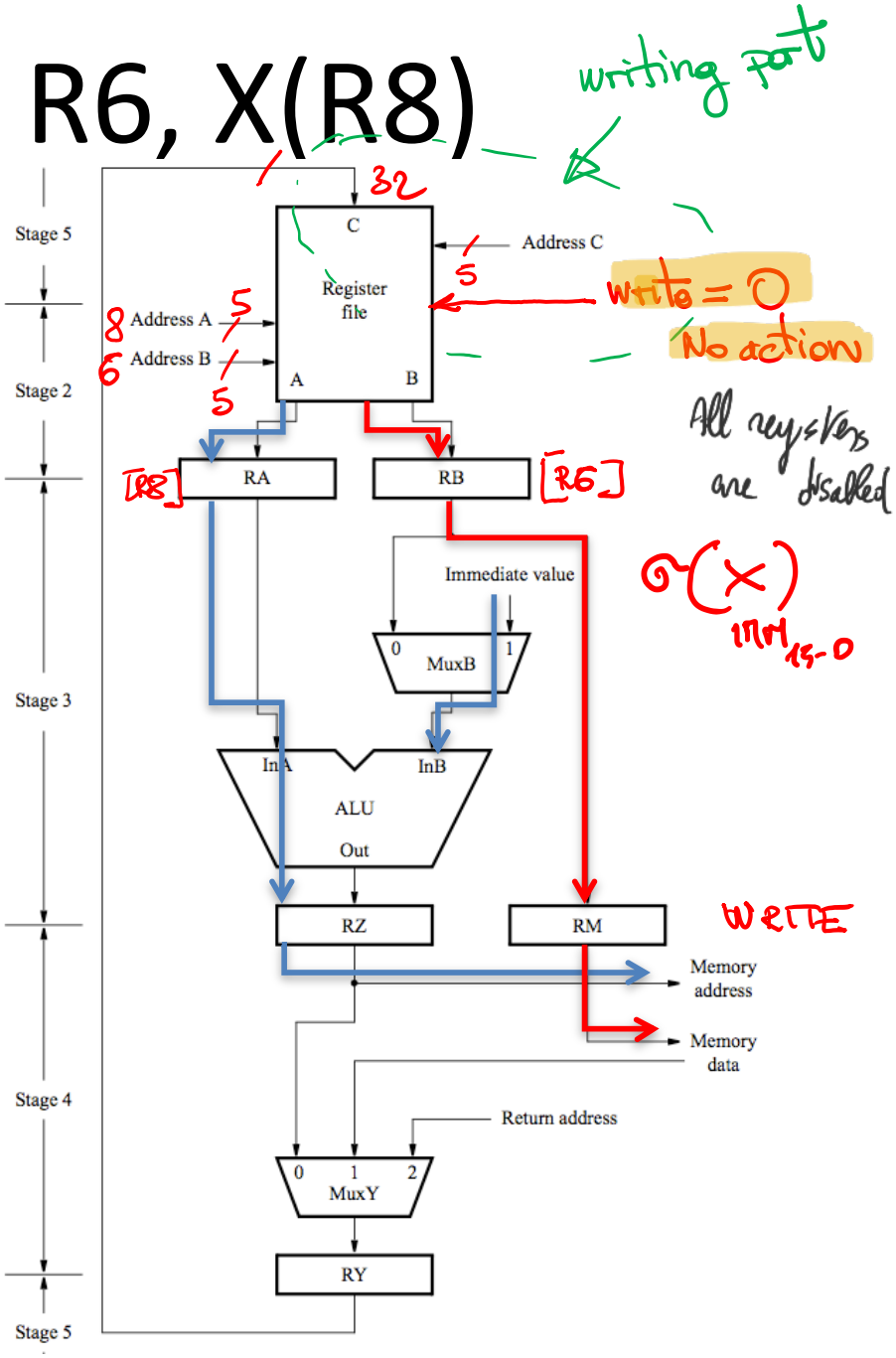
Example: ldw R5, X(R7)

1. Memory address ← [PC],
   Read memory,
   IR ← Memory data,
   PC ← [PC] + 4
2. Decode instruction,
   RA ← [R7]
3. RZ ← [RA] + Immediate value X
4. Memory address ←[RZ],
   Read memory, Wait for MFC, RY ← Memory data
5. R5 ← [RY]



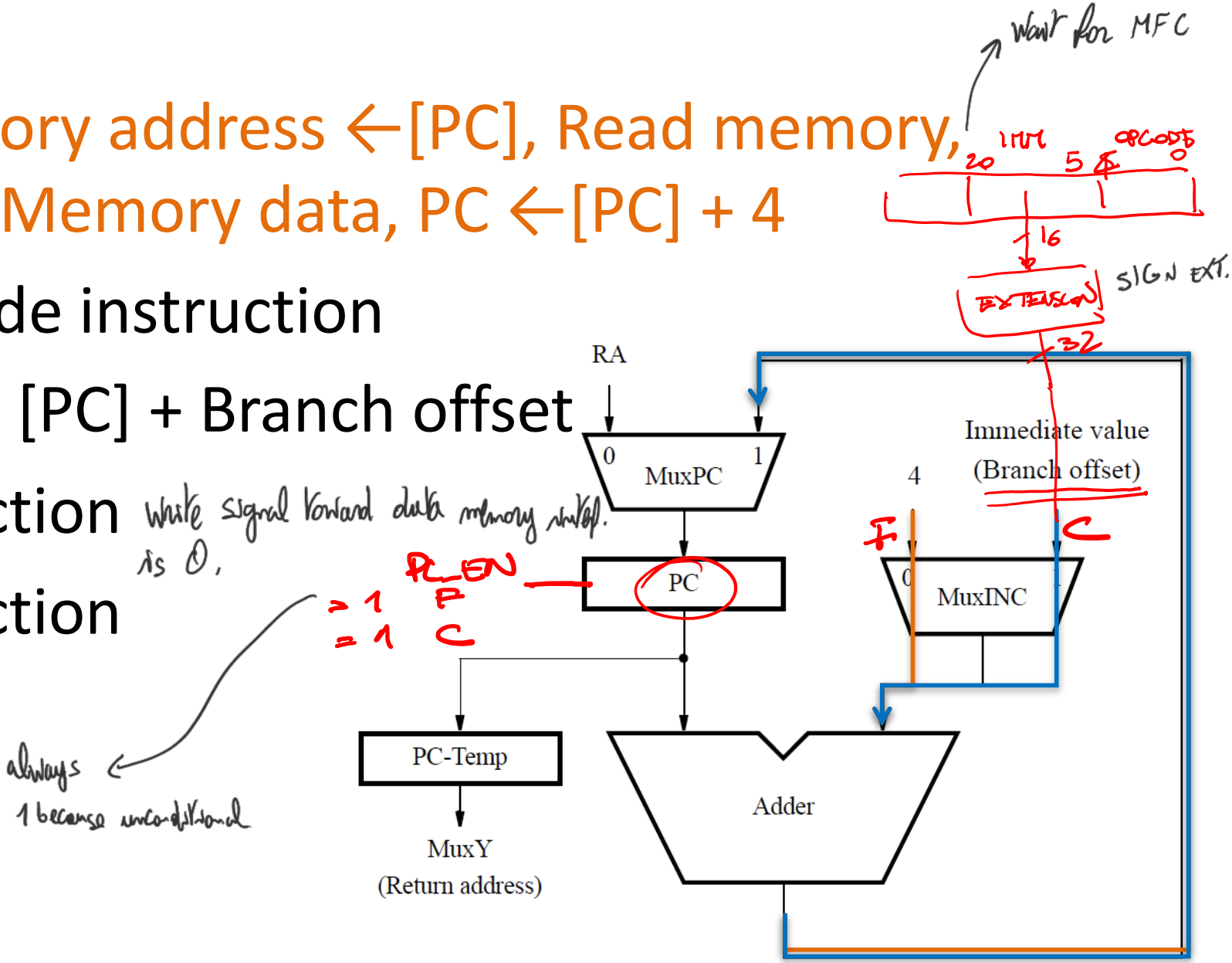
Example: stw R6, X(R8)

1. Memory address ← [PC],
   Read memory,
   IR ← Memory data,
   PC ← [PC] + 4
2. Decode instruction,
   RA ← [R8], RB ← [R6]
3. RZ ← [RA] + Immediate value X, RM ← [RB]
4. Memory address ←[RZ],
   Memory data ← [RM],
   Write memory, Wait for MFC
5. No action



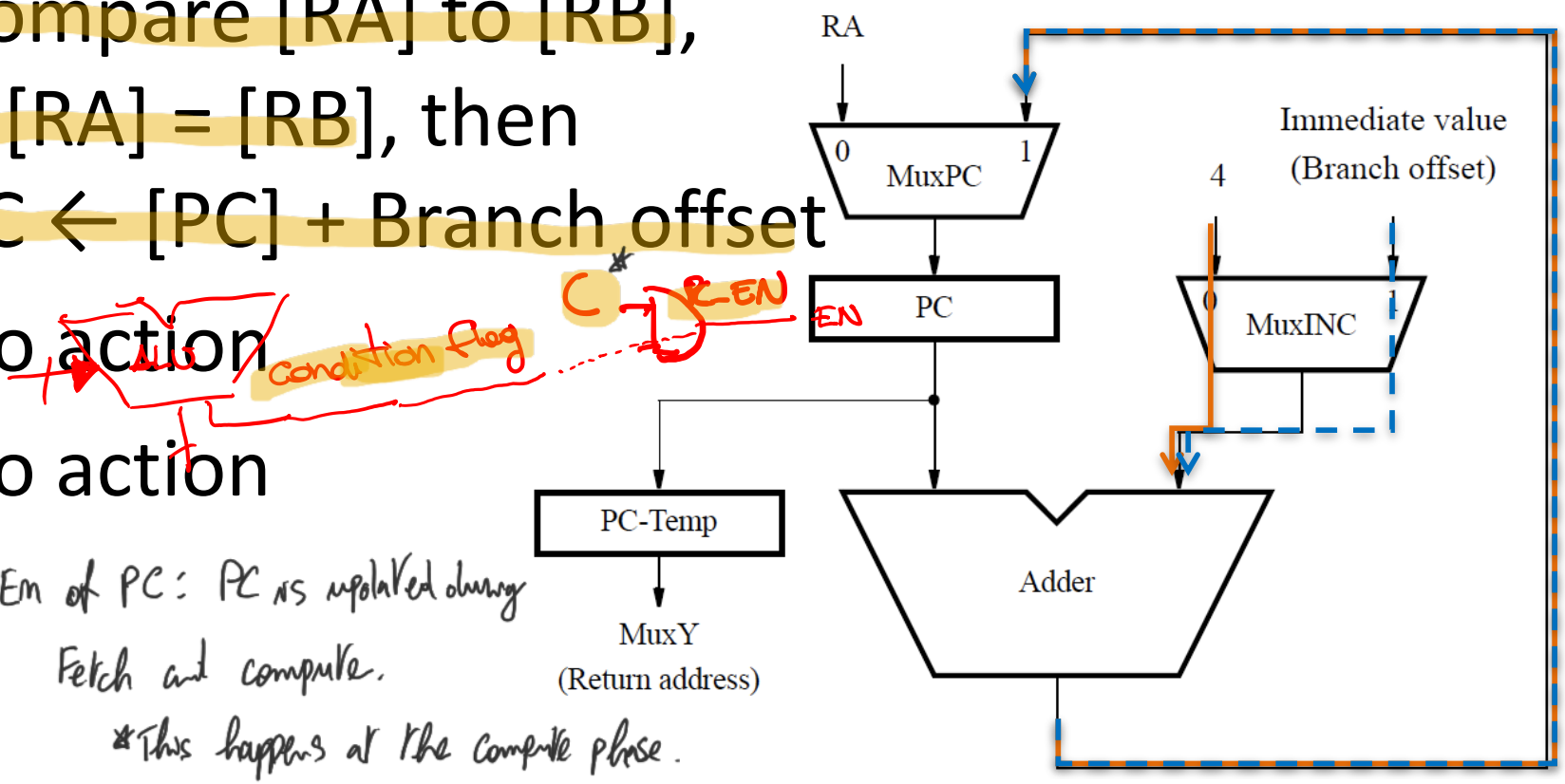
Unconditional branch

1. Memory address ←[PC], Read memory,
   IR ← Memory data, PC ←[PC] + 4
2. Decode instruction
3. PC ← [PC] + Branch offset
4. No action
5. No action



Conditional branch:
Branch_if_[R5]=[R6] LOOP

1. Memory address ← [PC], Read memory,
   IR ← Memory data, PC ←[PC] + 4
2. Decode instruction, RA ← [R5], RB ←[R6]
3. Compare [RA] to [RB],
   If [RA] = [RB], then
   PC ← [PC] + Branch offset
4. No action
5. No action



* En of PC: PC is updated always
Fetch and compute.
*This happens at the compute phase.

Subroutine call with indirection:

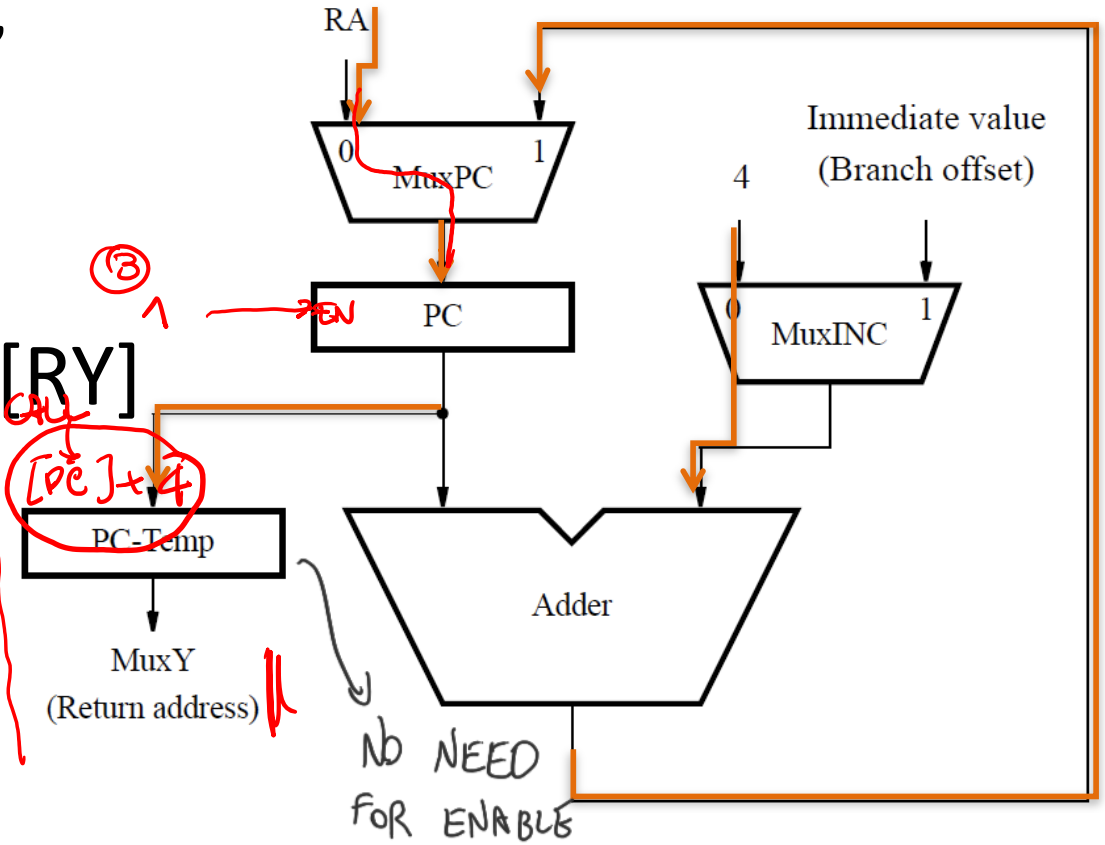
Value on PC changes after Read, greater than Read. So PC value can be sampled by PCTemp

callr

Call_register R9

CALL
RET

1. Memory address ← [PC], Read memory, wait for MFC
   IR ← Memory data, PC ←[PC] + 4
2. Decode instruction, RA ← [R9]
3. PC-Temp ← [PC],
   PC ← [RA]
4. RY ← [PC-Temp]
5. Register LINK ← [RY]

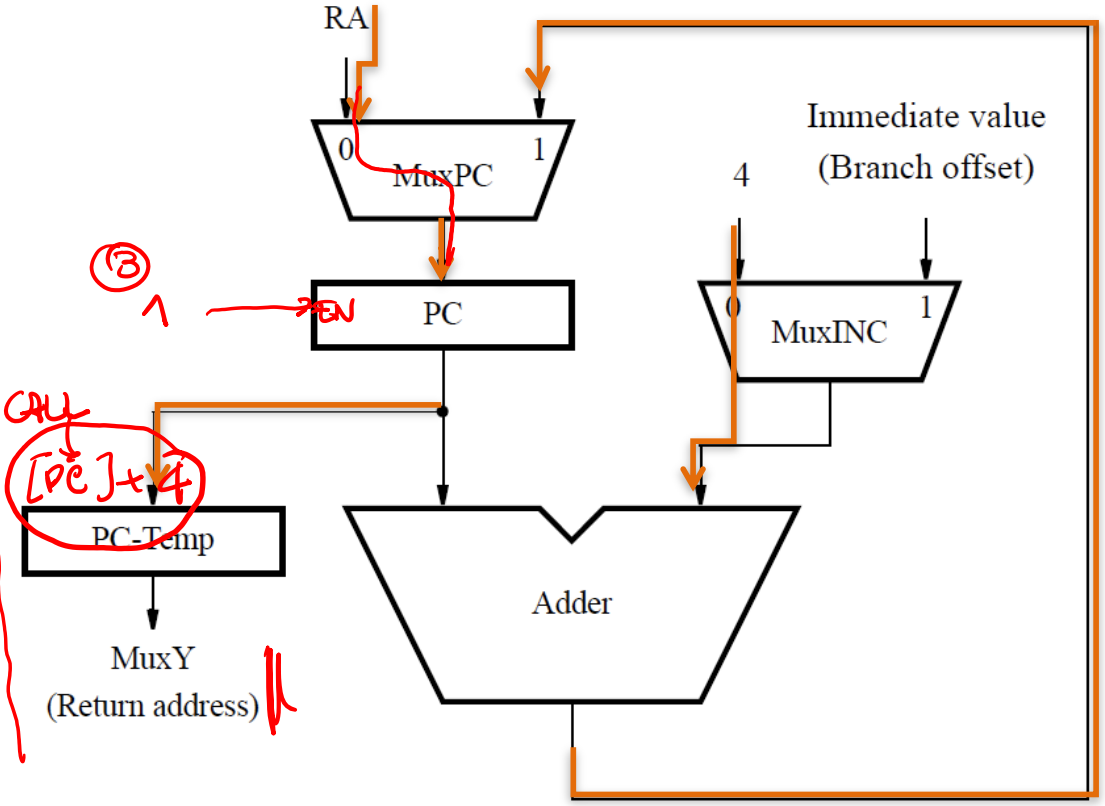


SUBROUTINE LINKAGE
The return address is stored in one of the general purpose regs

Subroutine RETurn

CALL
RET

1. Memory address ← [PC], Read memory,
   IR ← Memory data, PC ← [PC] + 4
2. Decode instruction, RA ← Register LINK
3. PC ← [RA]
4. No action
5. No action

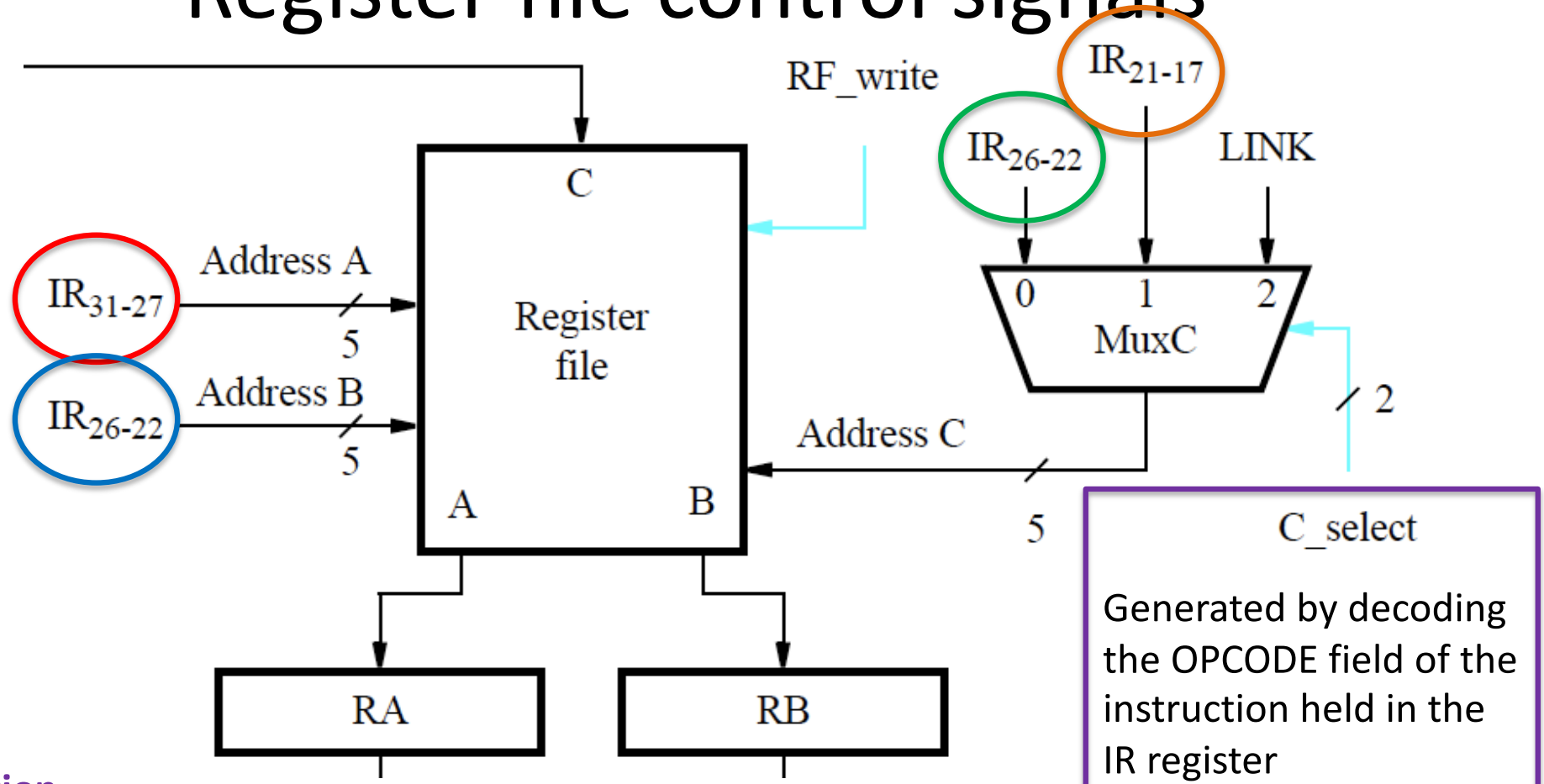


SUBROUTINE LINKAGE
The return address is stored in one of the general purpose regs

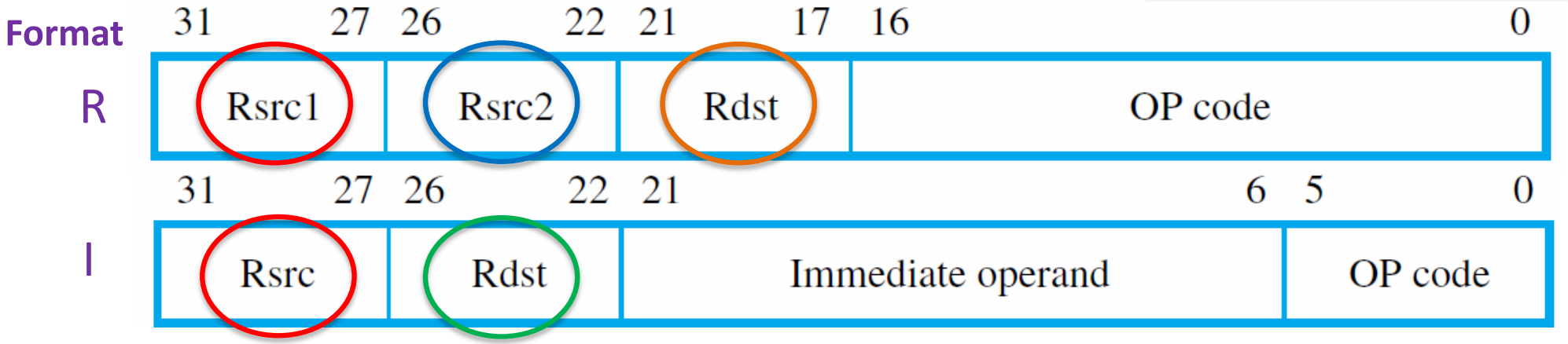
Control signals

- Select multiplexer inputs to route the flow of data
- Set the function performed by the ALU
- Determine when data are written into the PC, the IR, the register file, and the memory

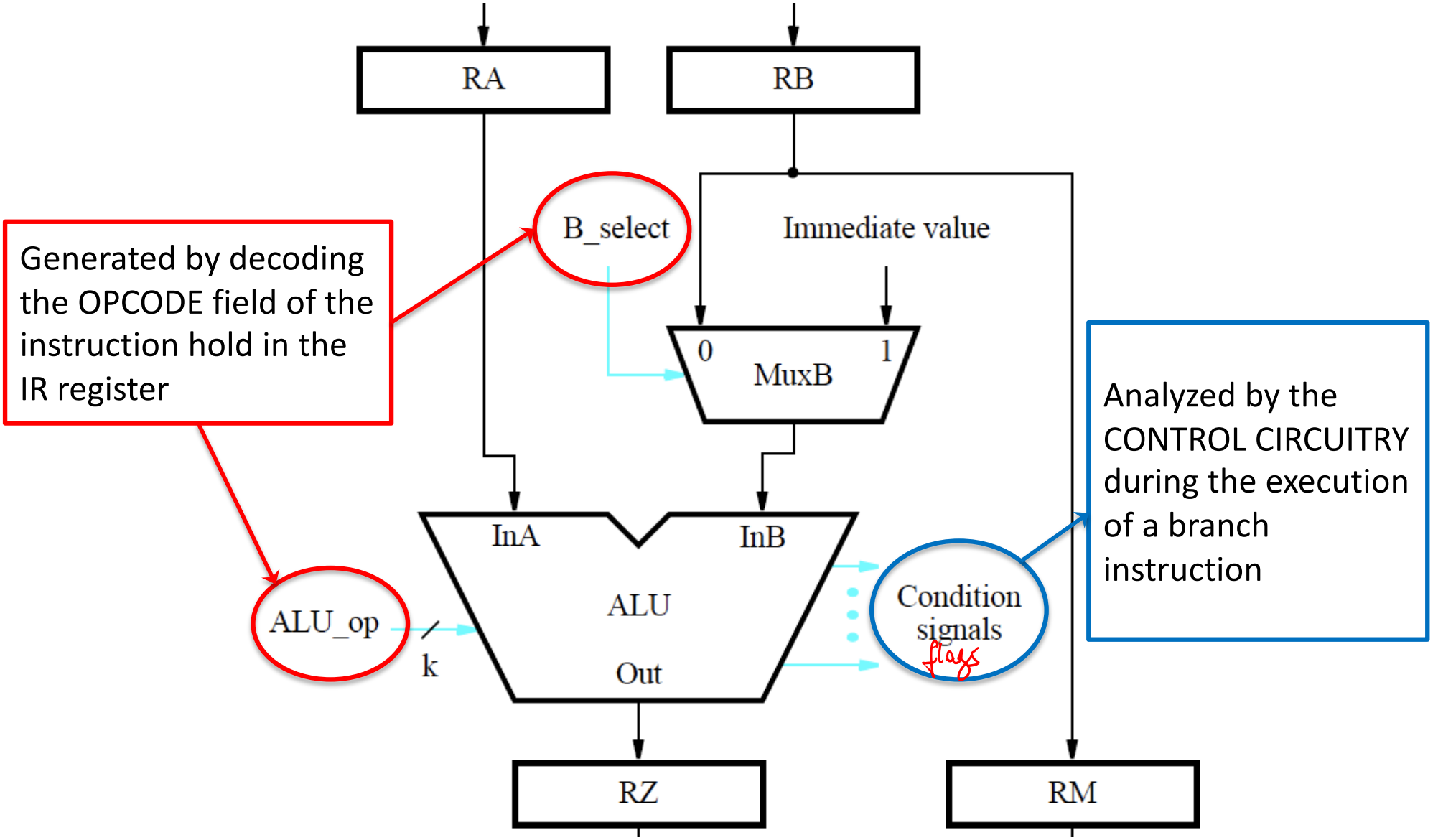
Register file control signals



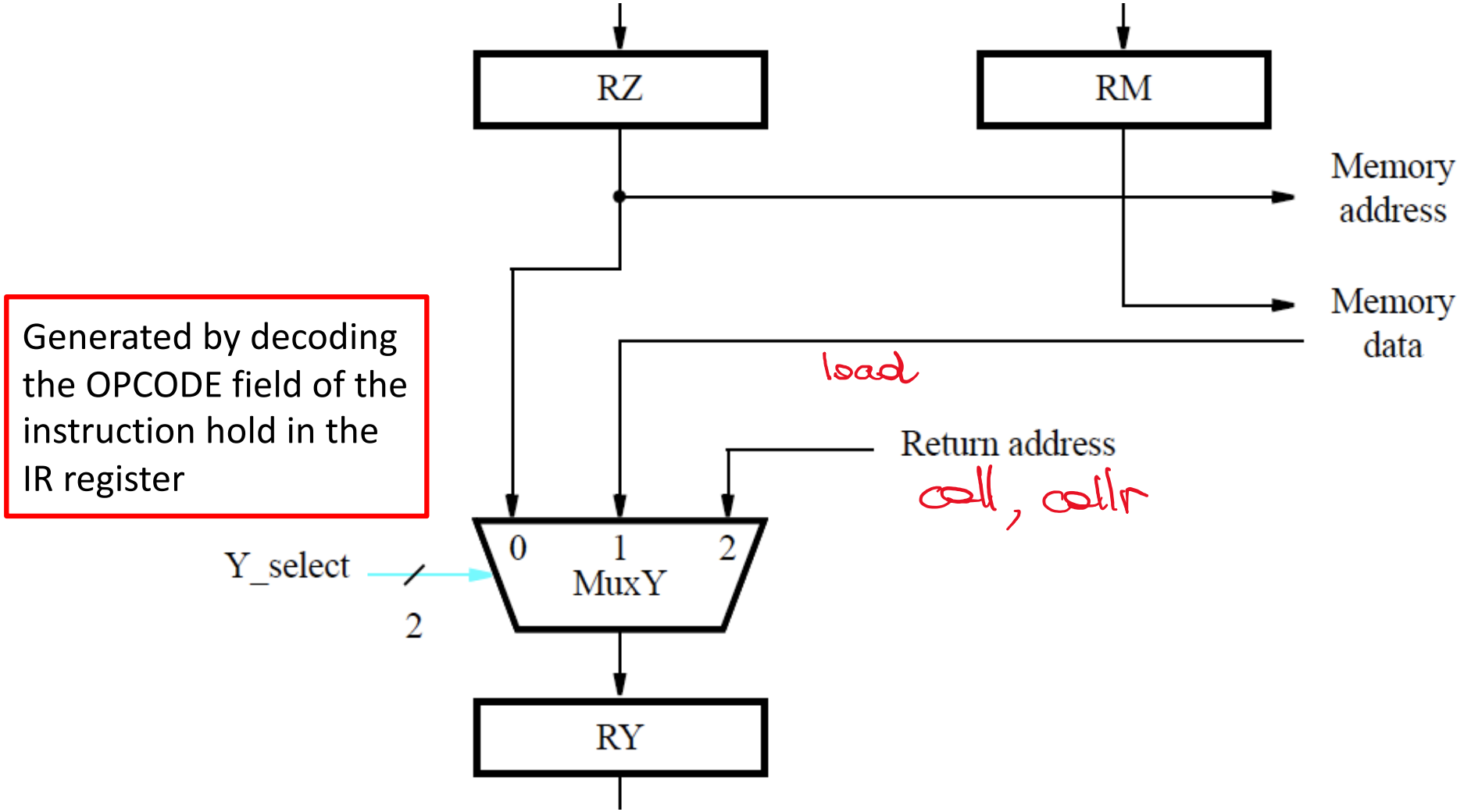
Instruction Format



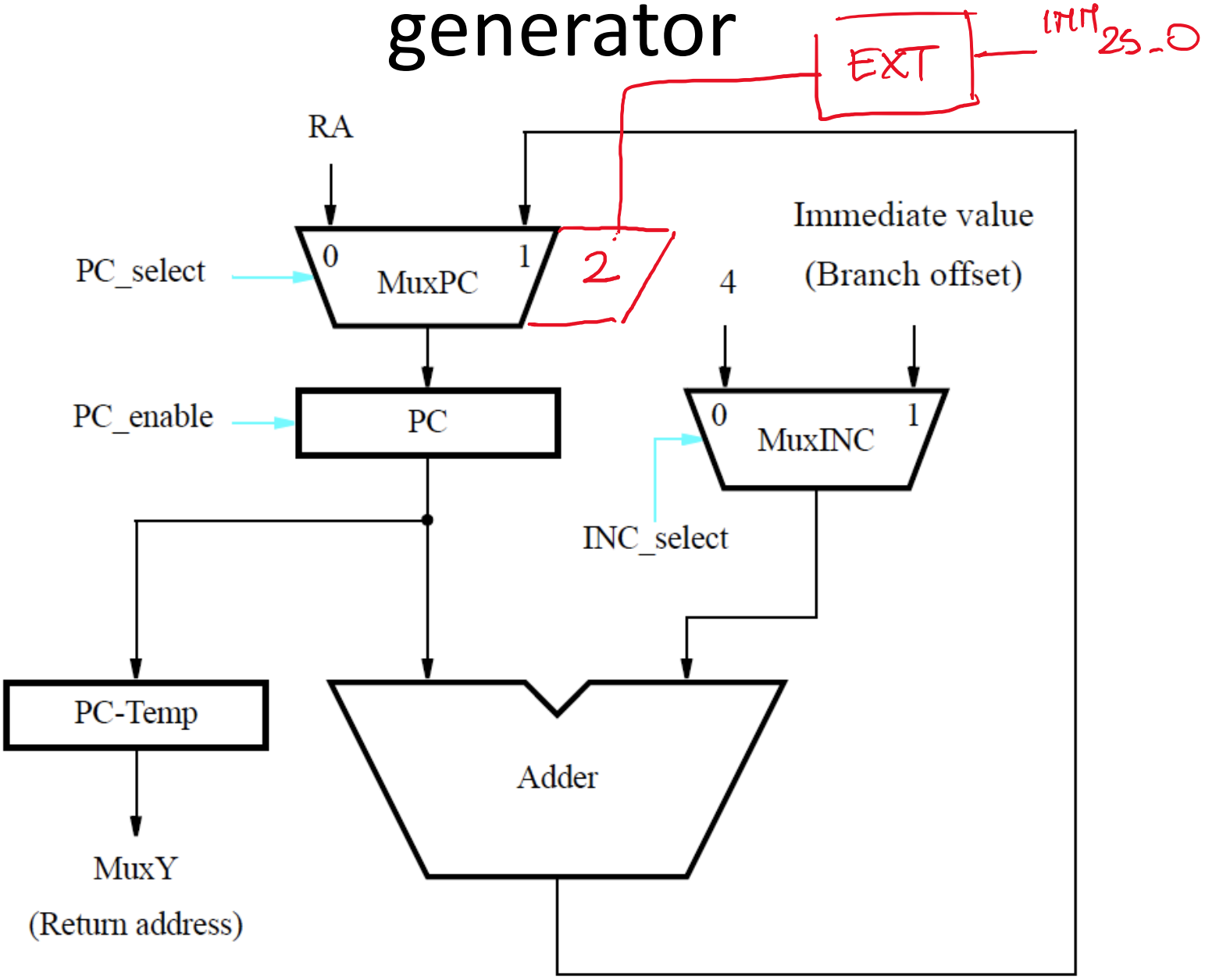
ALU control signals



Result selection



Control signals of instruction address generator



Control signal generation

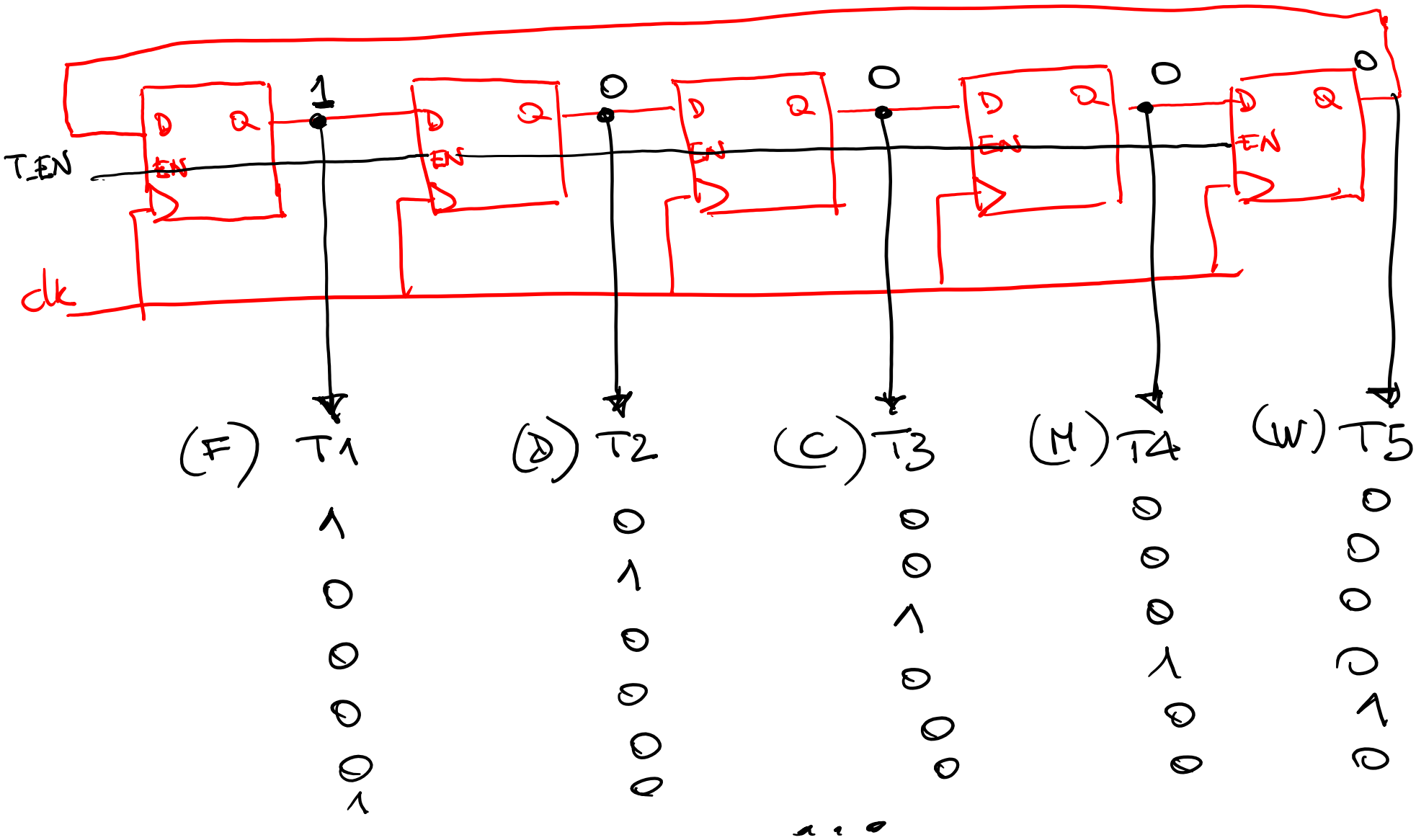
5-stage architecture: F D C M W

- Circuitry must be implemented to generate control signals so actions take place in correct sequence and at correct time.
- There are two basic approaches:
  hardwired control and microprogramming
- Hardwired control involves implementing circuitry that considers step (ring) counter, IR, ALU result, and external inputs.
- Step (Ring) counter keeps track of execution progress, one clock cycle for each of the five steps described (unless a memory access takes longer than one cycle).

STEP COUNTER

RING

10000 is reset configuration

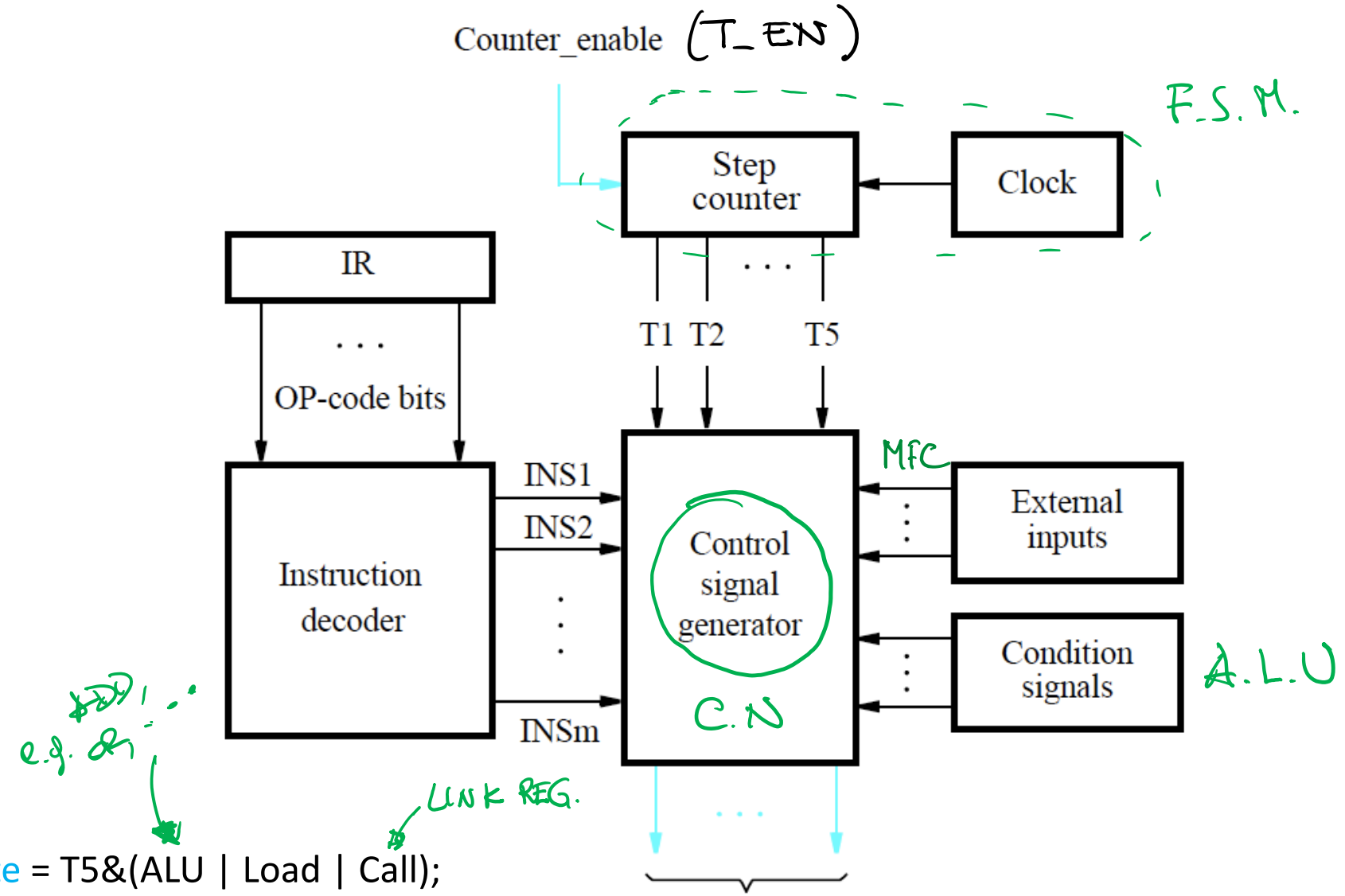


$$RF\_write = T5 \cdot (add, addi, or, \vdots, load, call, callr)$$

$$\overline{T\_EN}$$

$$PC\_enable = T1 \cdot MFC + T3 \cdot (call, callr, br, jmp, jmpi, brcond \cdot ret, cond)$$

Hardwired generation of control signals



E.g.

RF_write = T5&(ALU | Load | Call);

PC_enable = T1&MFC | T3&(BR | Ret | Call);

FETCH

COMPUTE

MFC is produced by the mem. intf. at end of the memory access

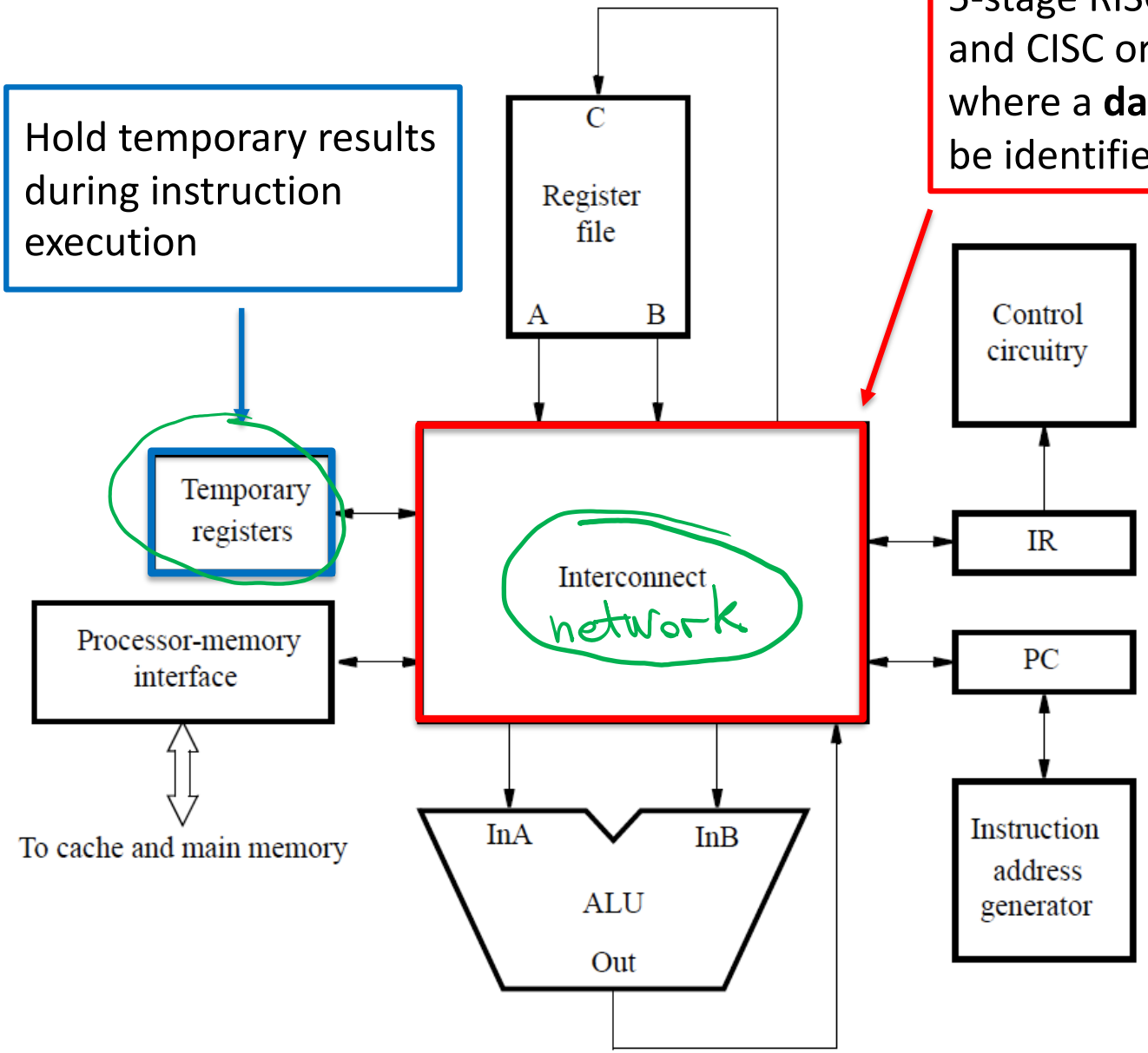
CISC processors

- CISC-style processors have more complex instructions.
- The full set of instructions cannot all be implemented in a fixed number of steps.
- Execution steps for different instructions do not all follow a prescribed sequence of actions.
- Hardware organization should therefore enable a flexible flow of data and actions to accommodate CISC.

Hardware organization for a CISC computer

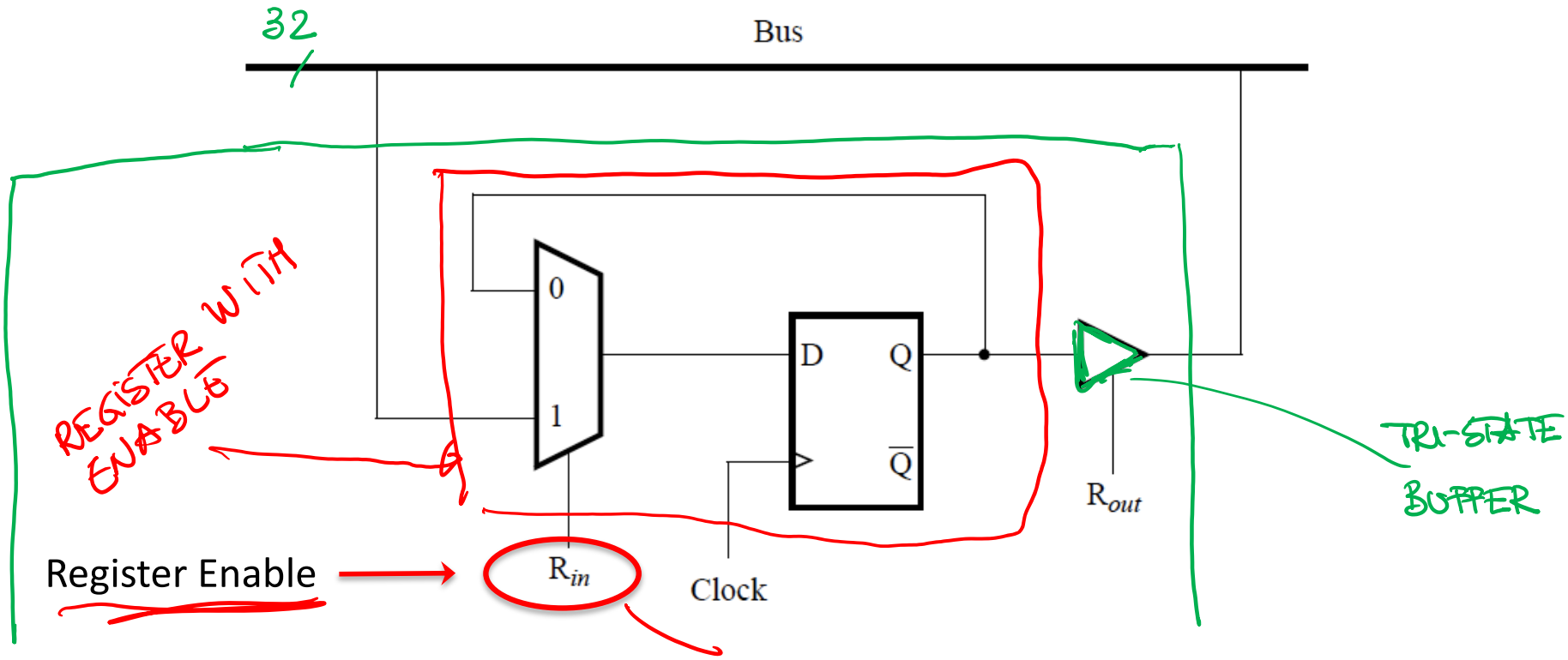
Main difference between 5-stage RISC organization and CISC organization, where a **datapath** cannot be identified easily

Hold temporary results during instruction execution

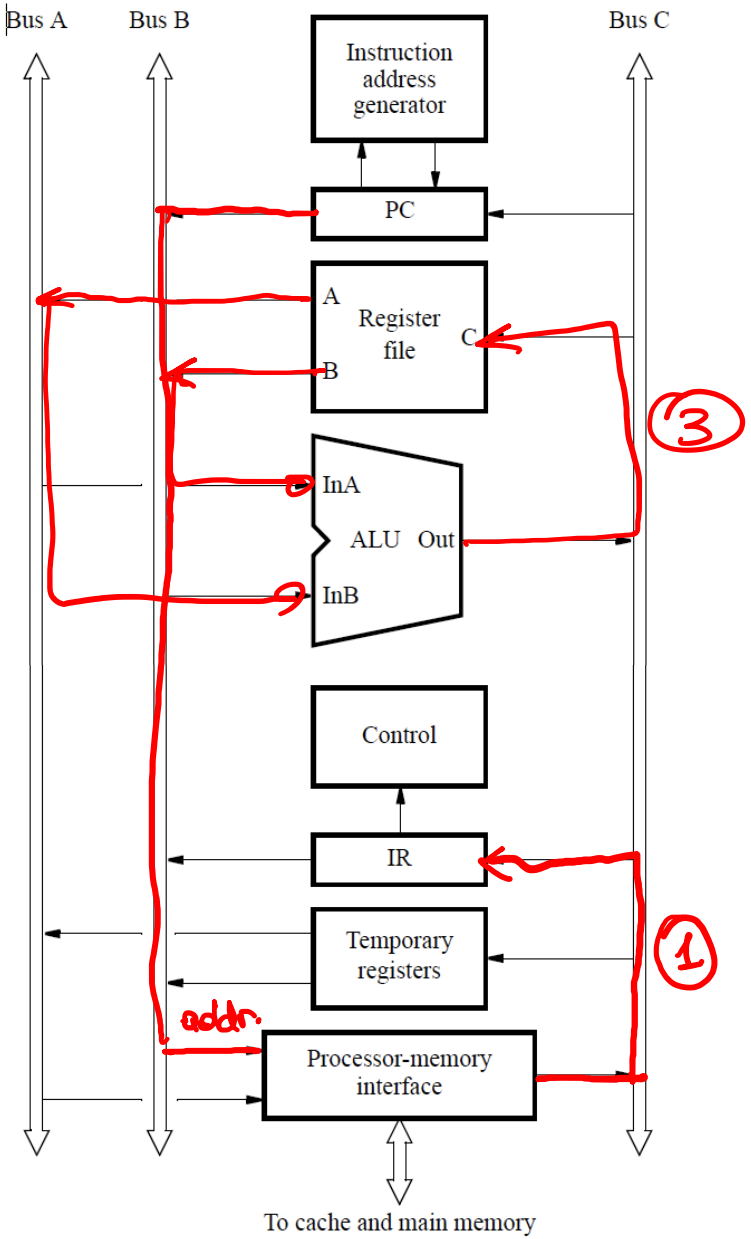


Bus

- An example of an interconnection network.
- When functional units are connected to a common bus, tri-state drivers are needed.



A 3-bus interconnection network

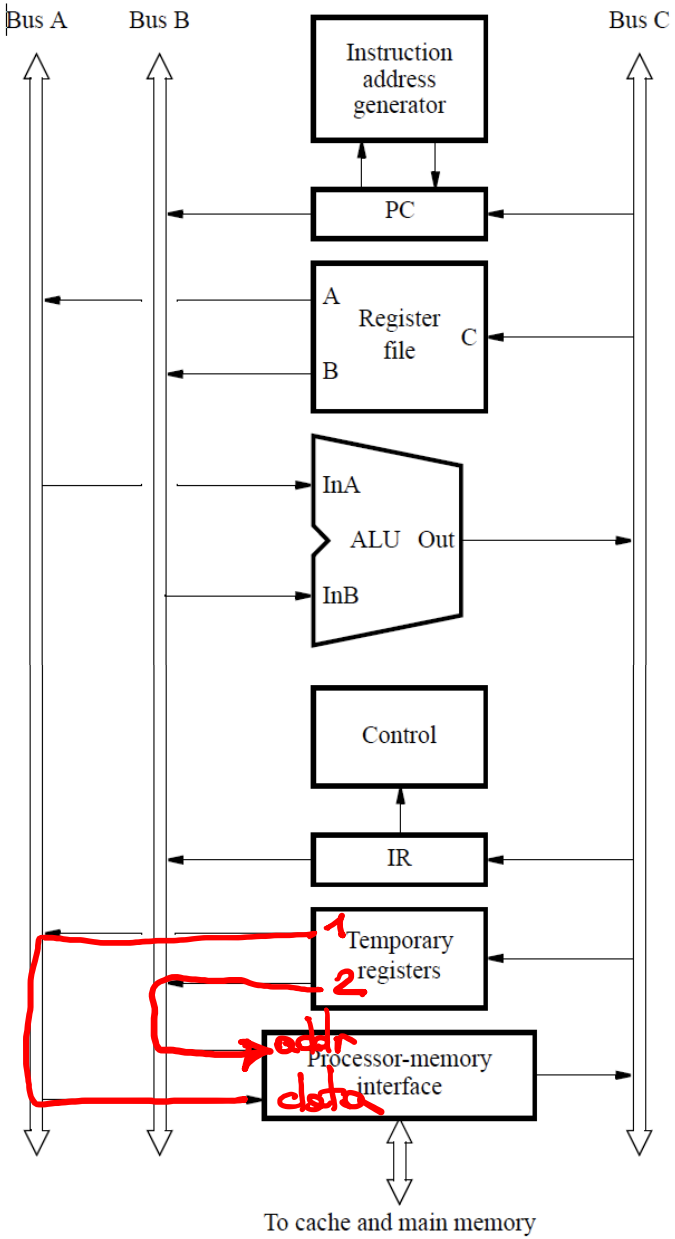


R5 ← R5 + R6

Example 1: Add R5, R6

1. Memory address ← [PC], Read memory, Wait for MFC, IR ← Memory data, PC ← [PC] + 4
2. Decode instruction
3. R5 ← [R5] + [R6]

A 3-bus interconnection network



Example 2*: And X(R7), R9

1. Memory address ← [PC], Read memory, Wait for MFC,
   IR ← Memory data,
   PC ← [PC] + 4
2. Decode instruction
3. Memory address ← [PC], Read memory, Wait for MFC,
   Temp1 ← Memory data,
   PC ← [PC] + 4
4. **Temp2** ← [**Temp1**] + [R7]
5. Memory address ← [**Temp2**], Read memory, Wait for MFC, **Temp1** ← Memory data
6. **Temp1** ←[**Temp1**] AND [R9]
7. Memory address ← [**Temp2**], Memory data ← [**Temp1**], Write memory, Wait for MFC

*X is stored as a second word of the instruction

References

- C. Hamacher, Z. Vranesic, S. Zaky, N. Manjikian "Computer Organization and Embedded Systems," *McGraw-Hill International Edition*
  - Chapter V: Basic Processing Unit