# 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

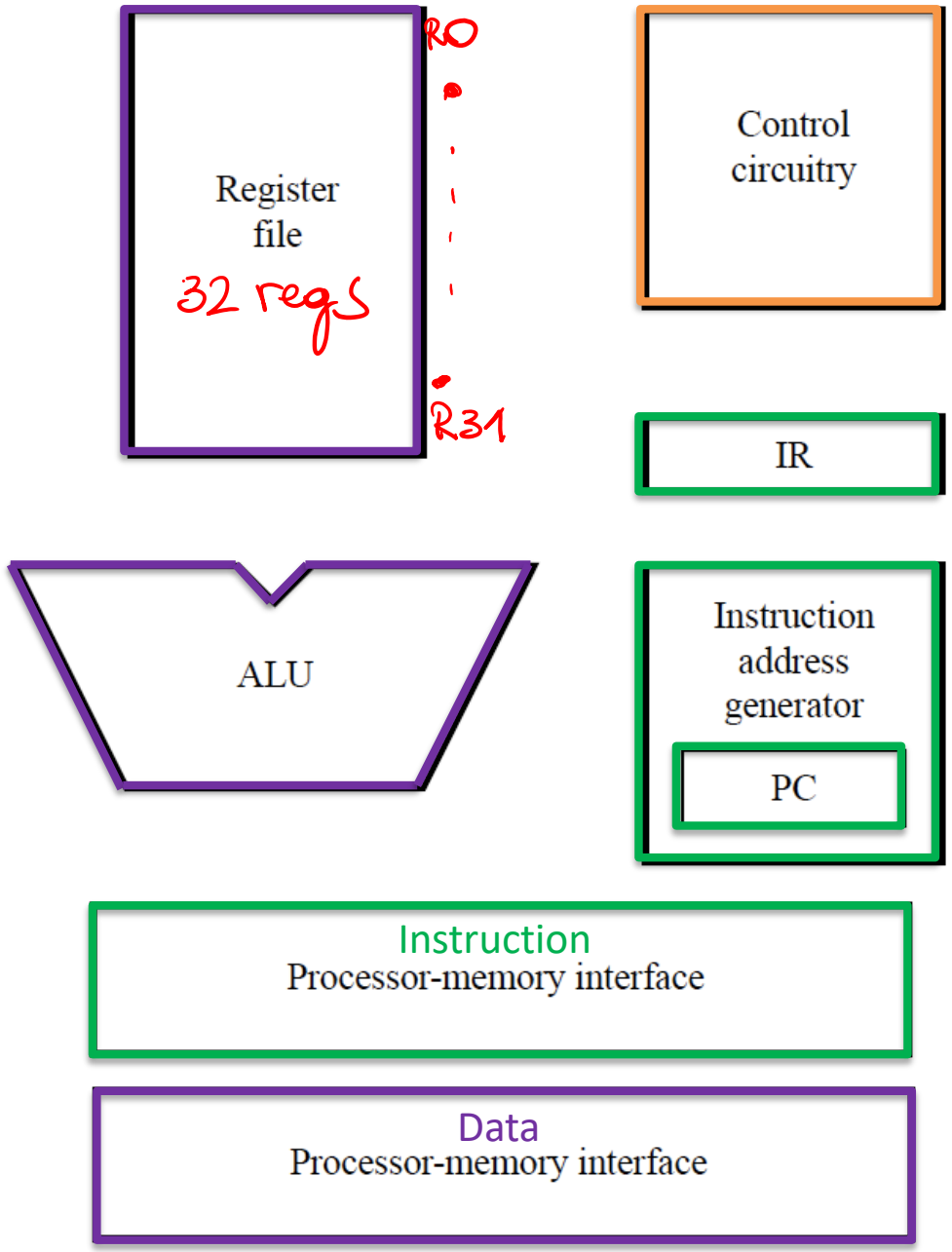
Office hours:

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

# COMPUTER ORGANIZATION

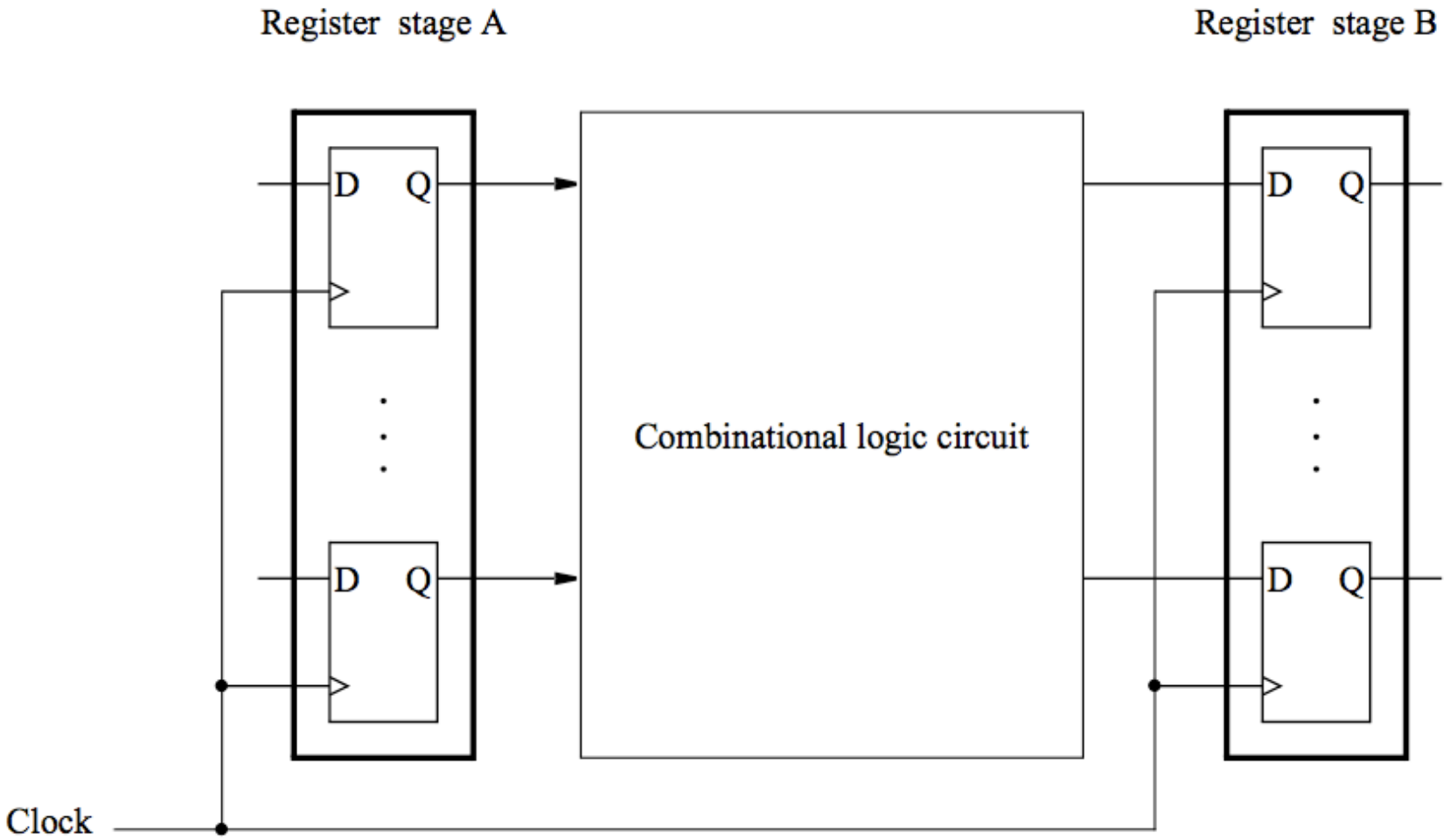
# Processor's building blocks

- 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



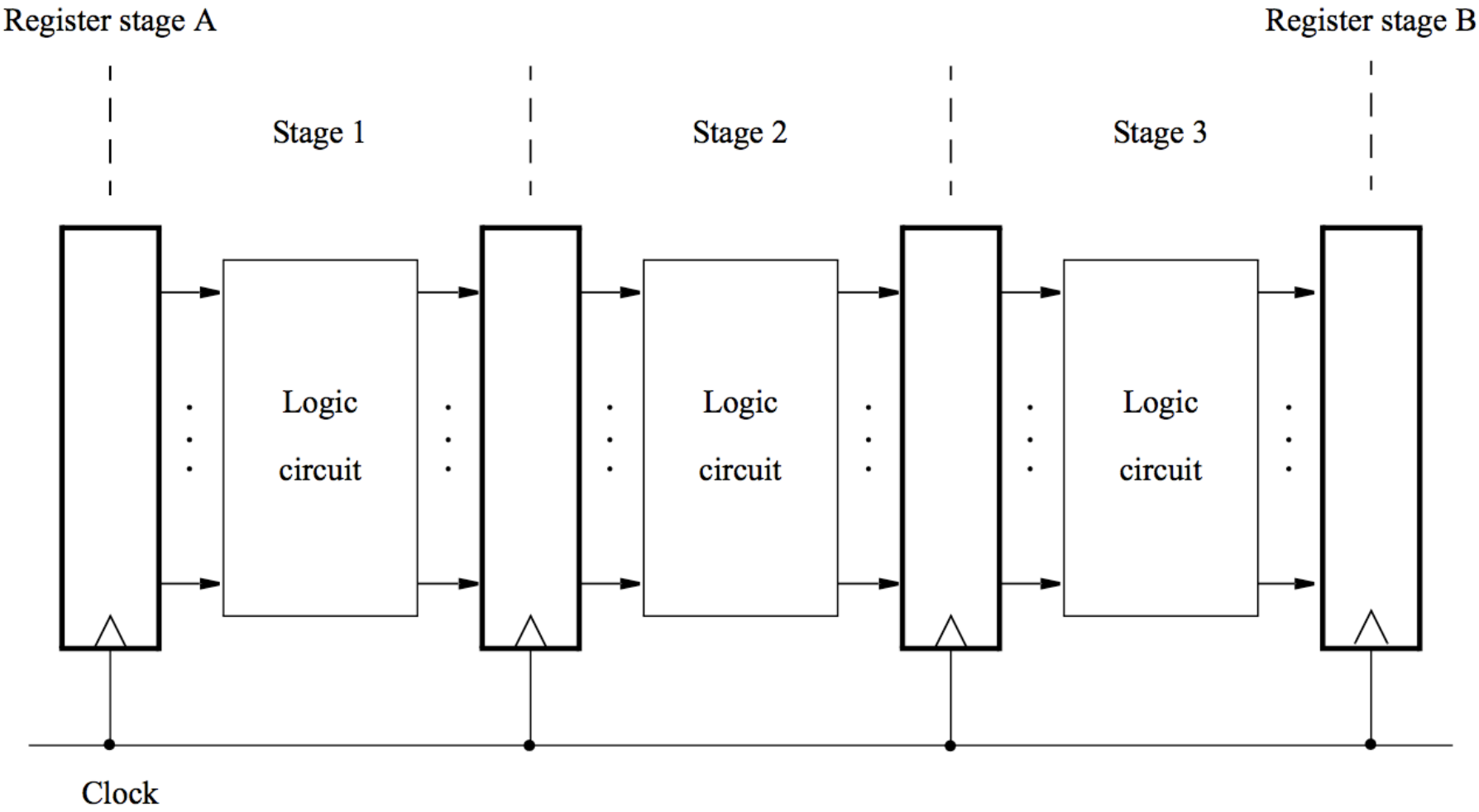
# A digital processing system

- datapath



# A multi-stage digital processing system

- datapath

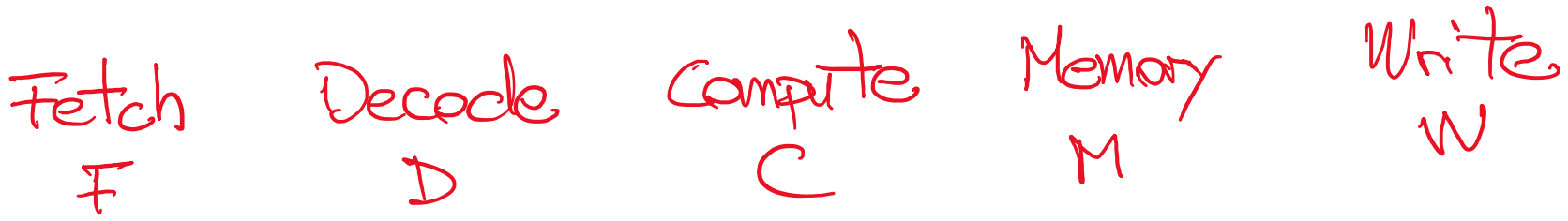


# Why multi-stage?

- 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]

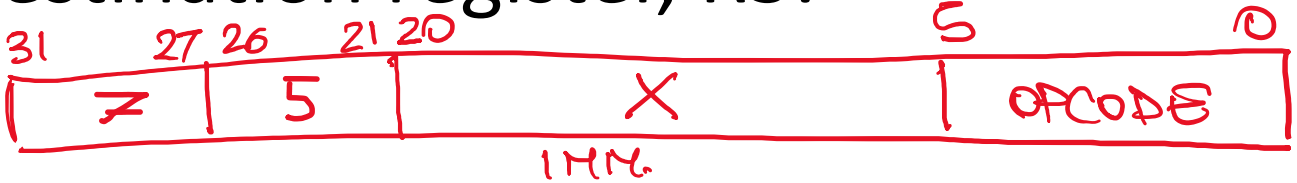
F 1. Fetch the instruction and increment the program counter.

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].

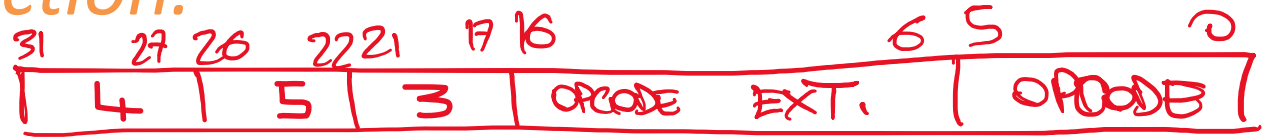
M 4. No action.


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.

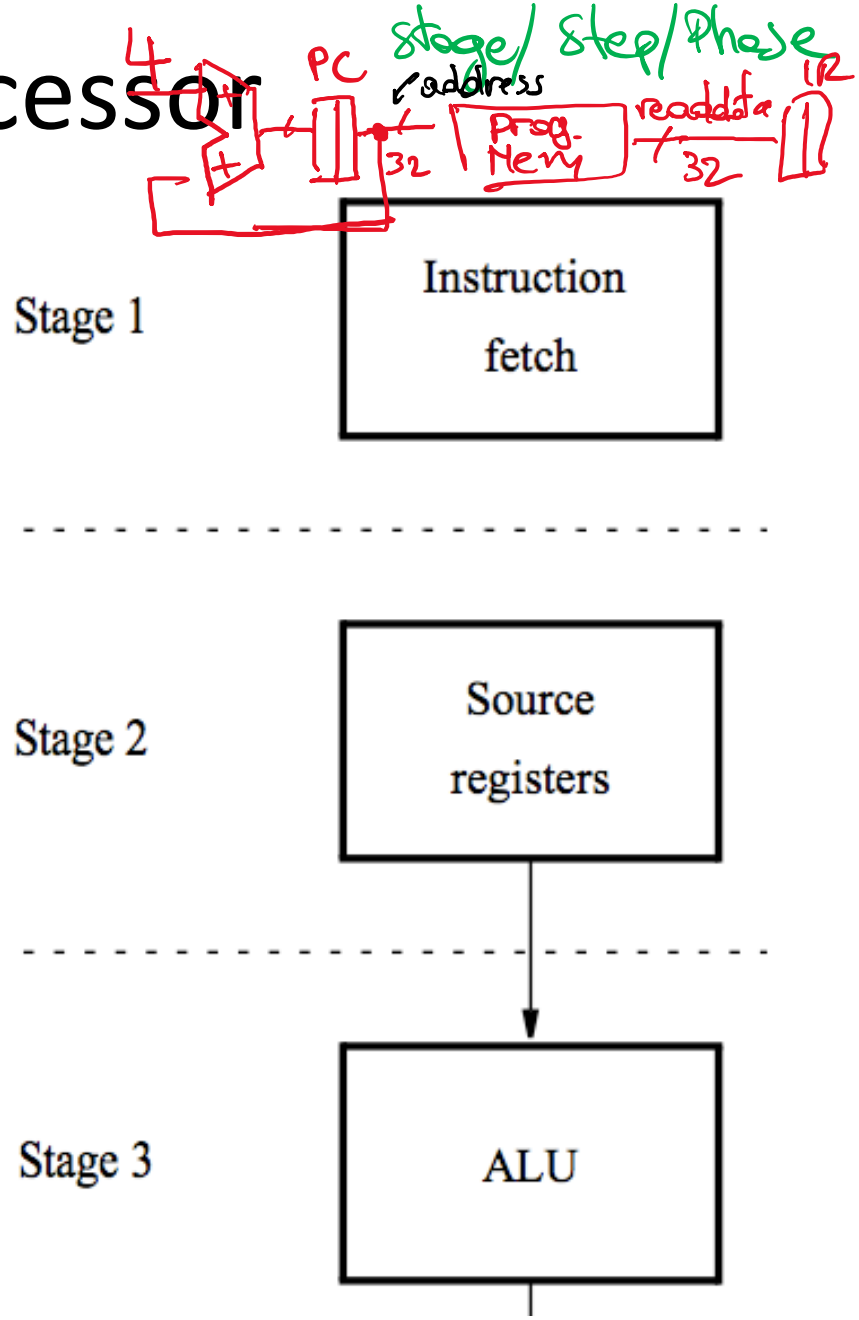
M 4. Read or write memory data if the instruction involves a memory operand.

W 5. Write the result into the destination register.

- This sequence determines the hardware stages needed.

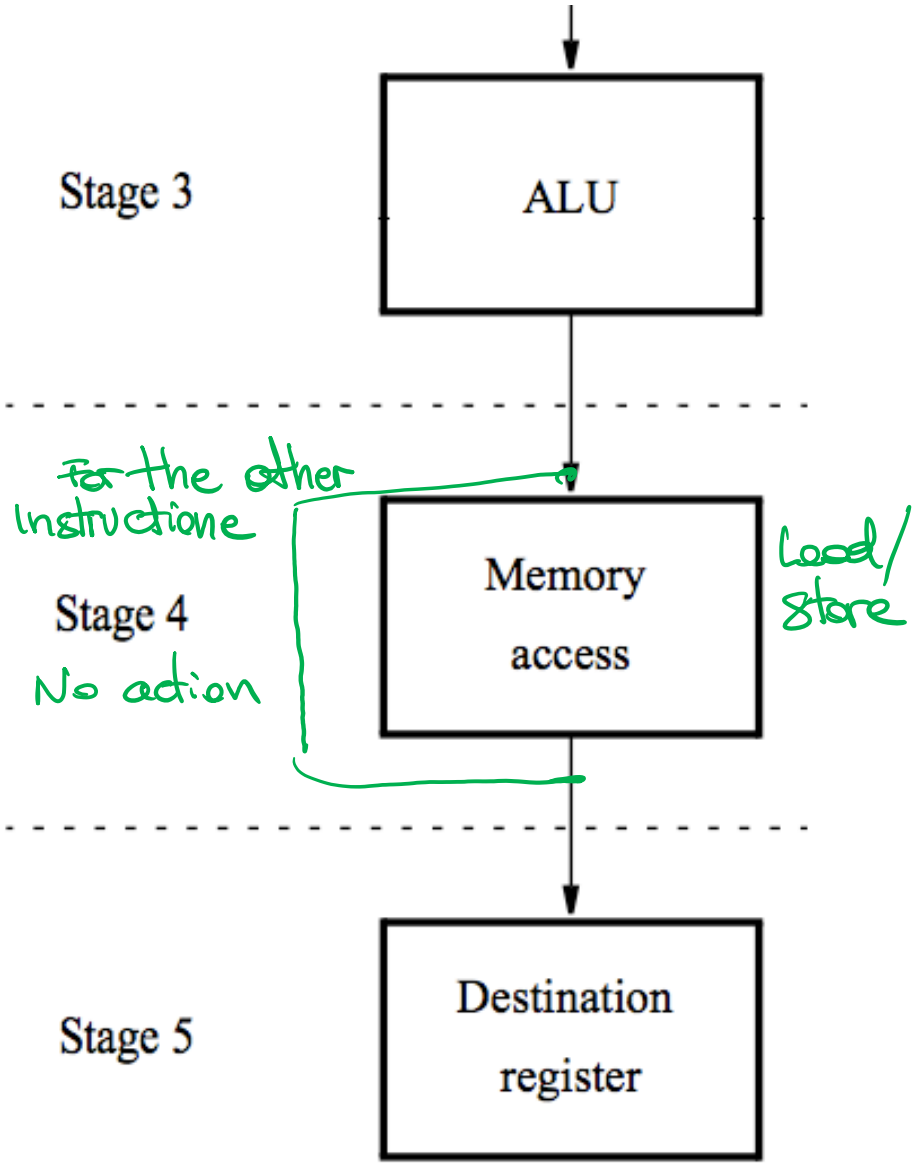
# A 5-stage implementation of a RISC processor

- 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.



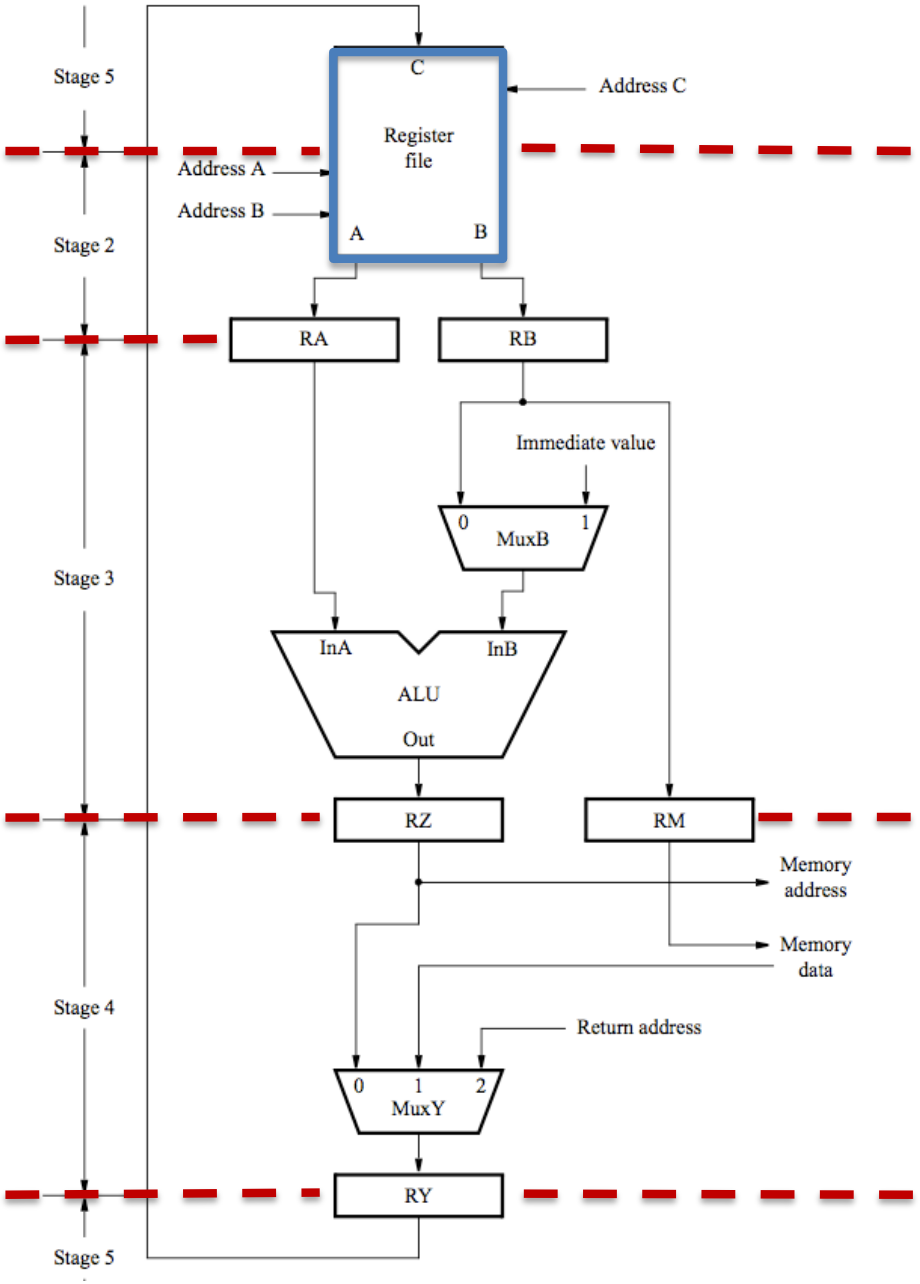
# A 5-stage implementation of a RISC processor

- …
- 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)



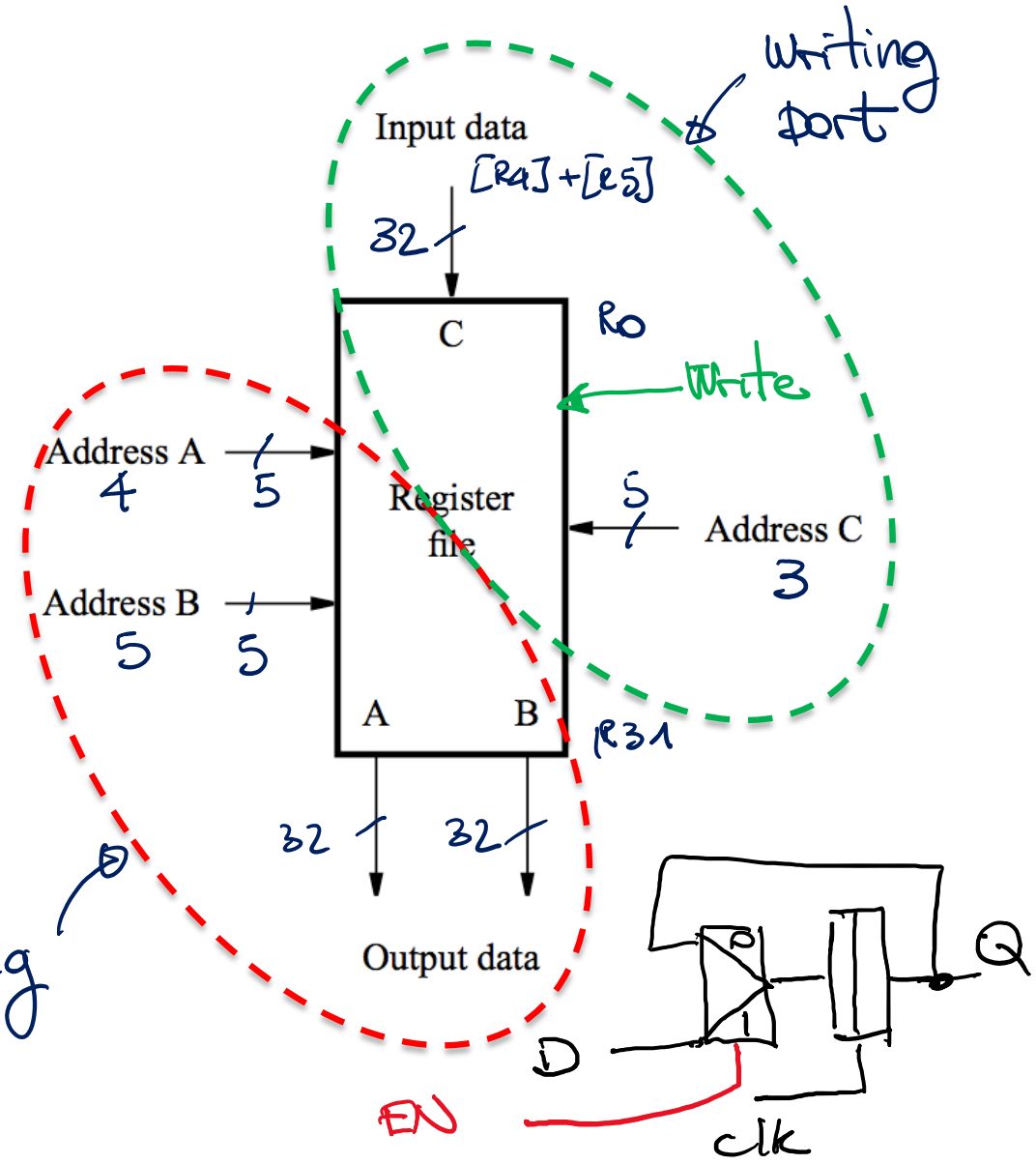
32 REGISTERS

# Hardware components: Register file

is involved in 2 phases: DECODE and WRITE

(may be)

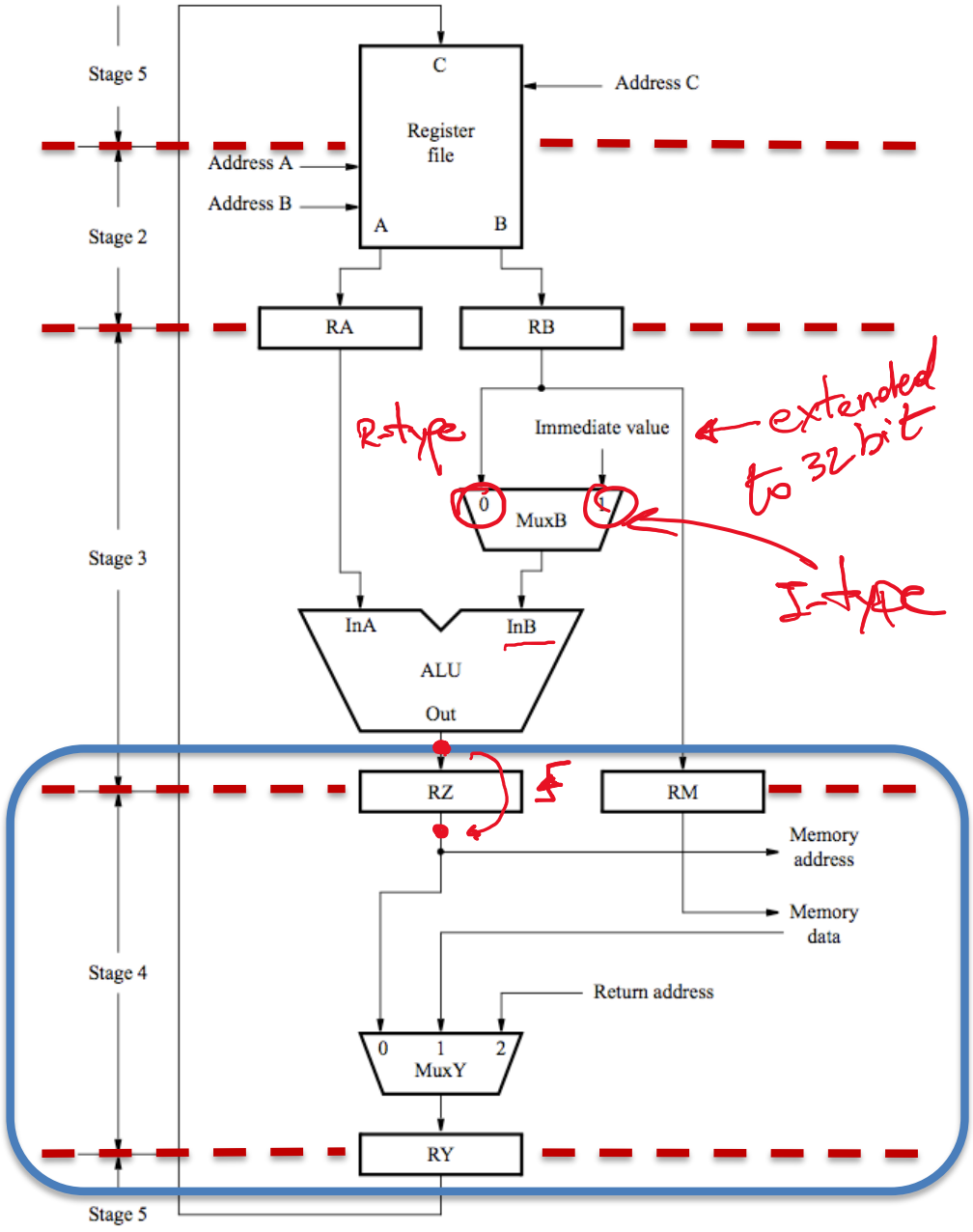
- A 2-port register file is needed to read the two source registers at the same time.
- It may be implemented using a 2-port memory.




Eg. ADD R3, R4, R5

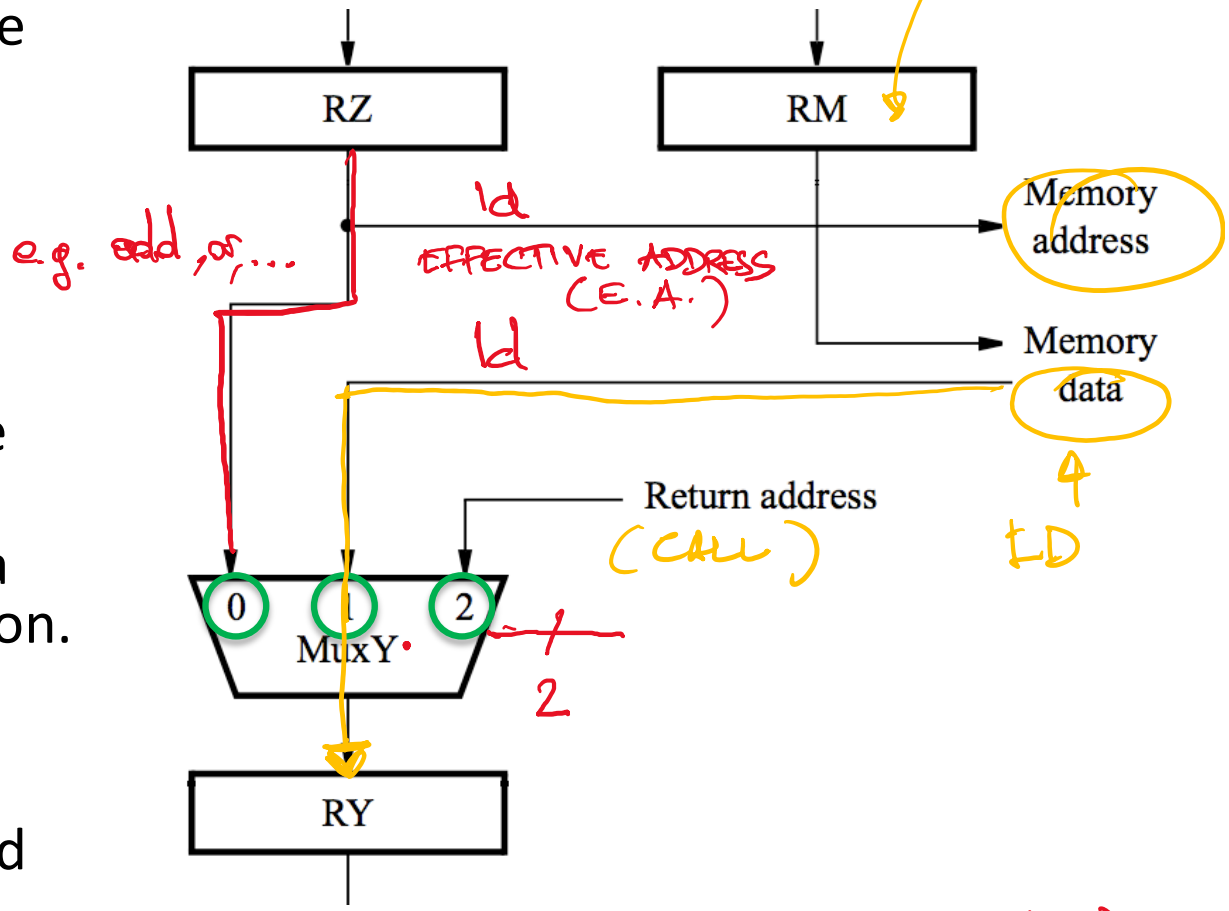
# The datapath – Stages 2 to 5

- 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

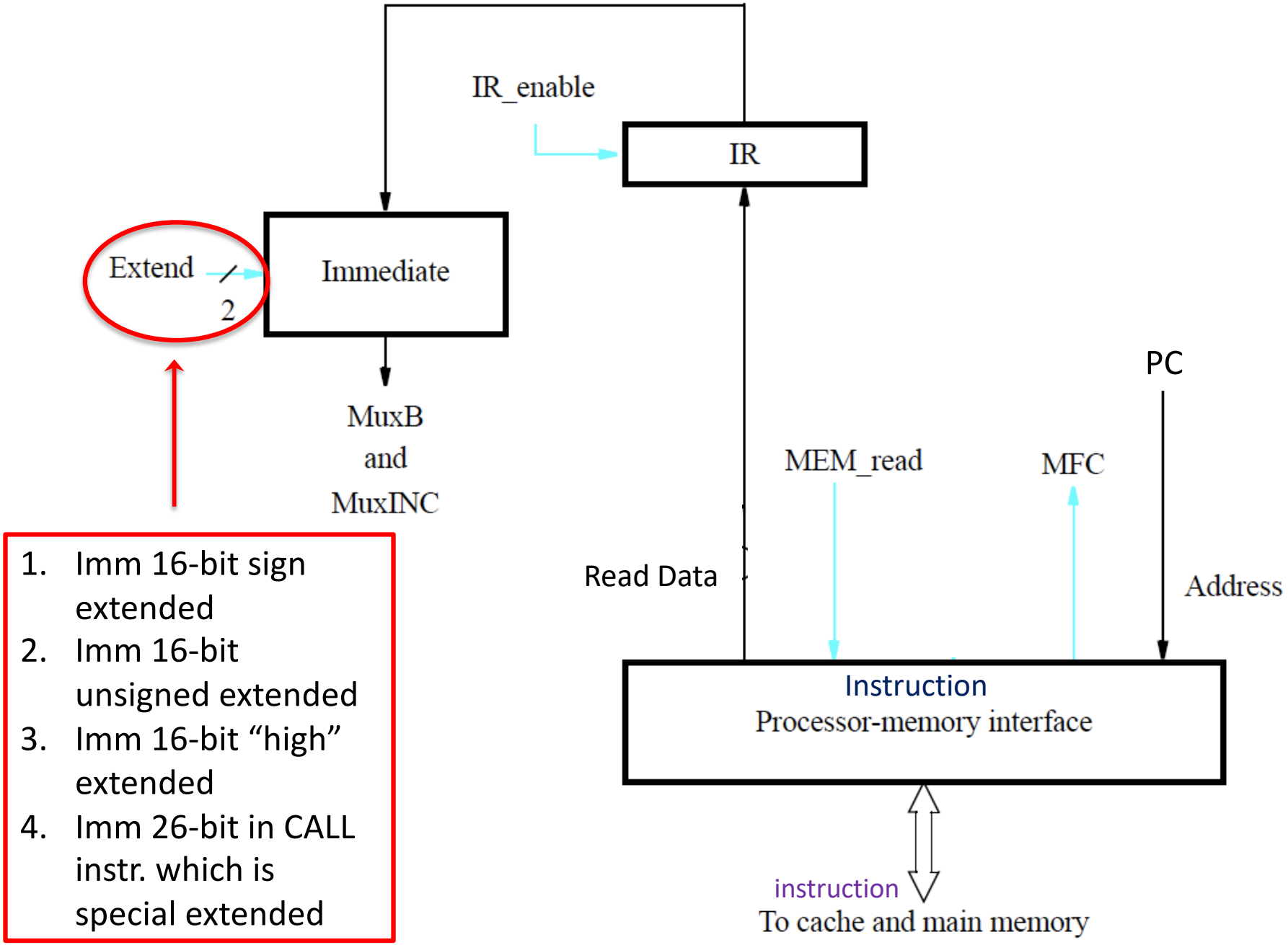


# Memory access

- 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:

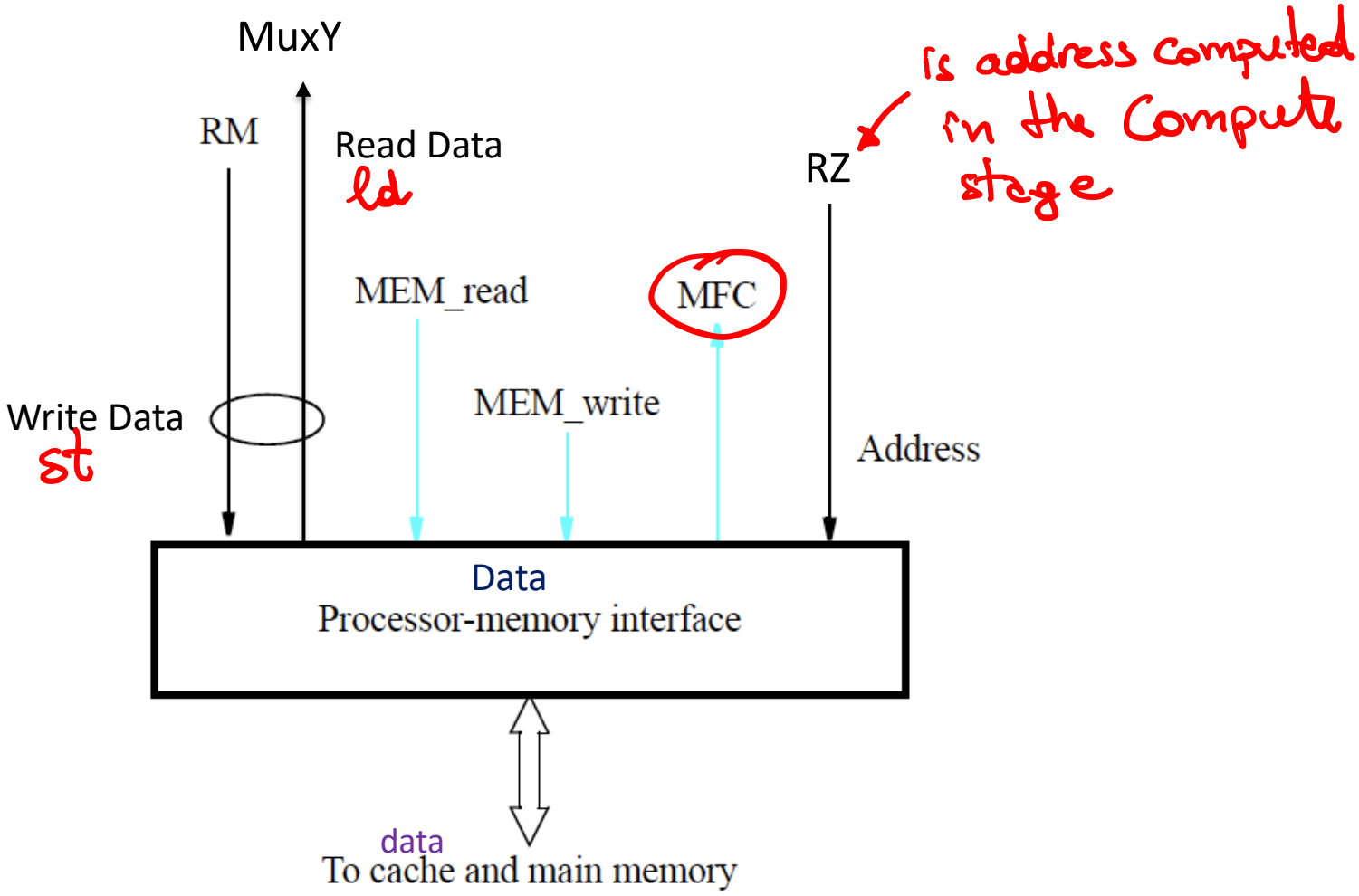

1. Memory address ← [PC], Read memory,
   Wait for MFC,
   IR ← Memory data, PC ← [PC] + 4

# Instruction memory and IR control signals



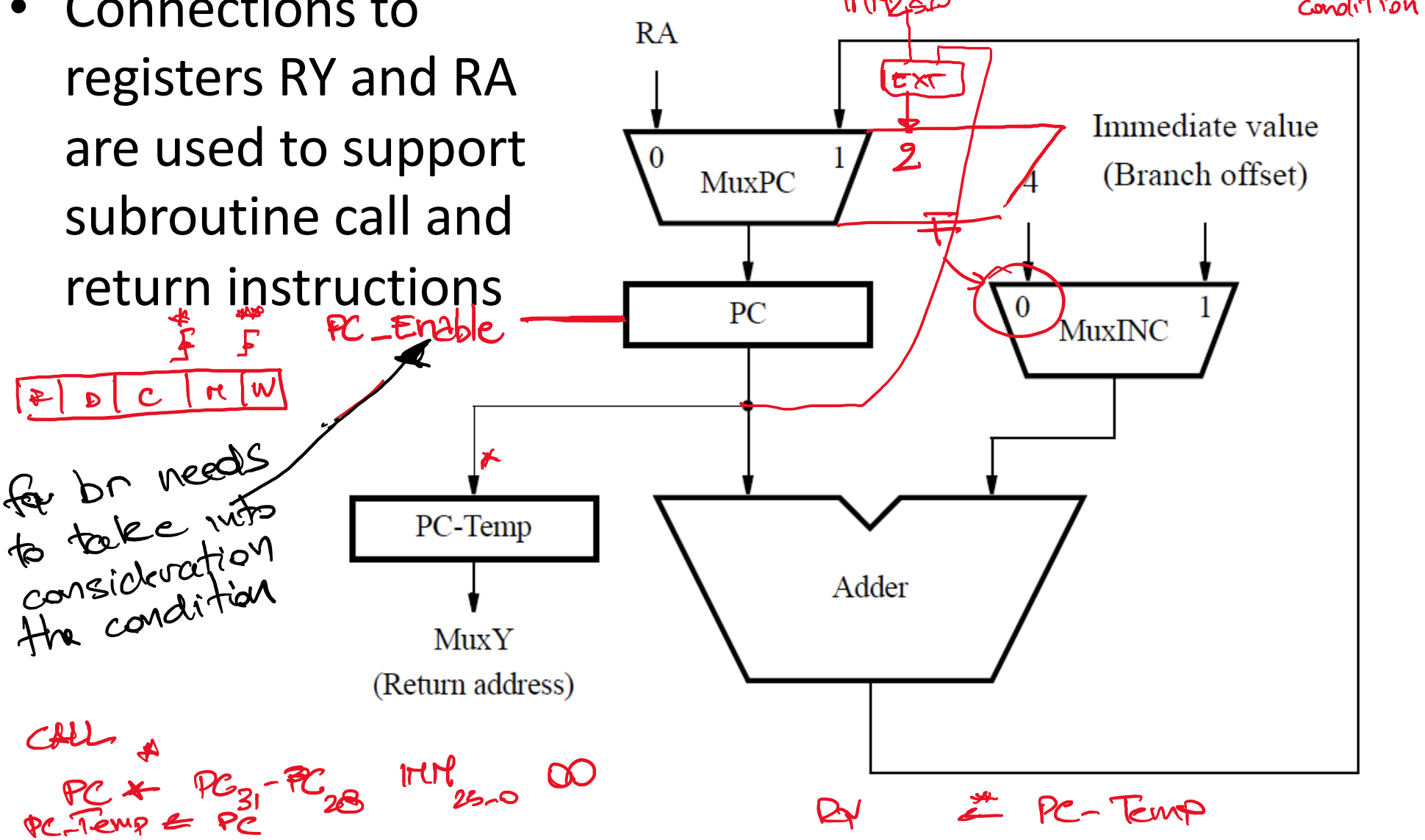

1. Imm 16-bit sign extended
2. Imm 16-bit unsigned extended
3. Imm 16-bit “high” extended
4. Imm 26-bit in CALL instr. which is special extended

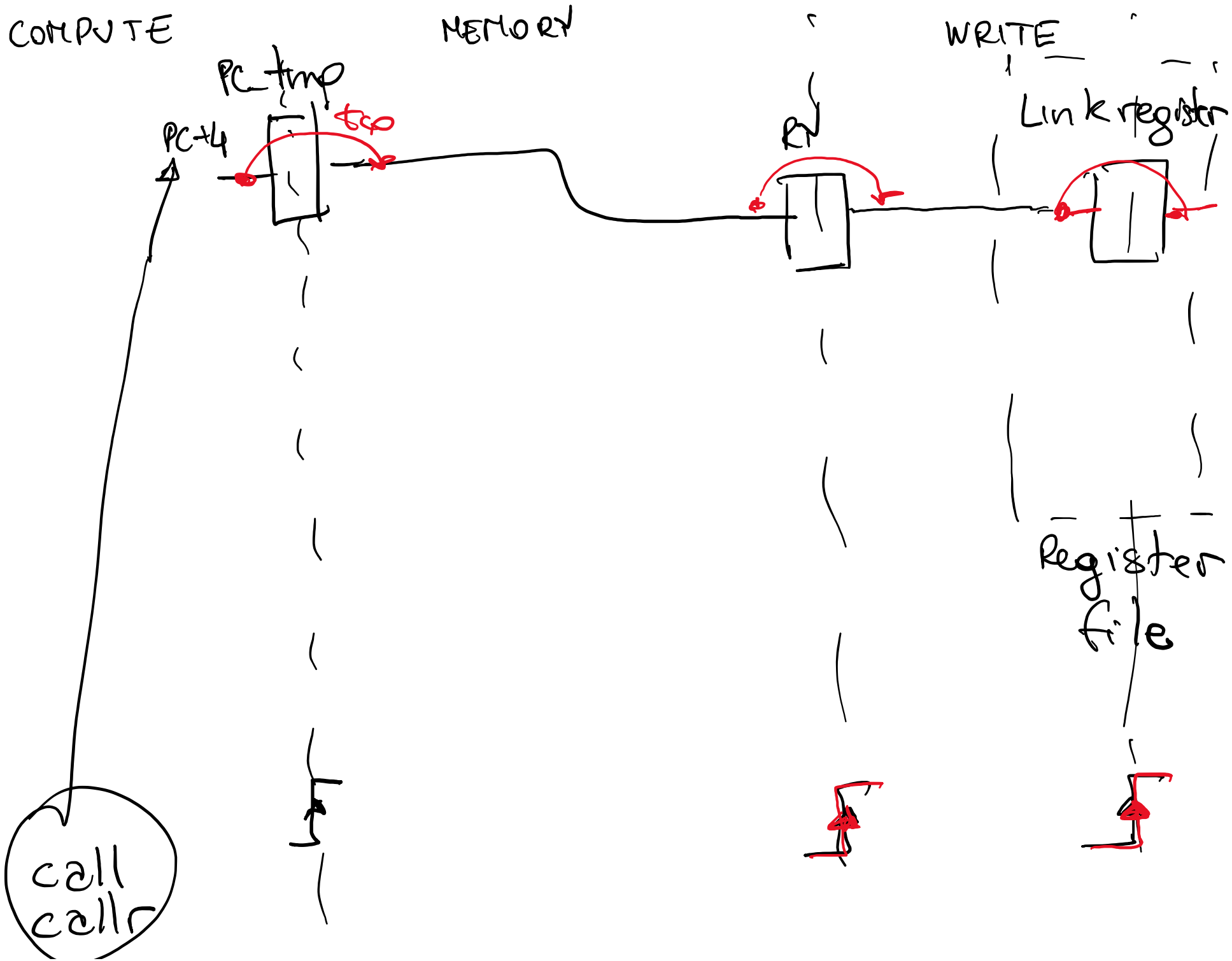
# Data memory interface



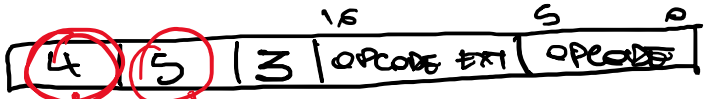
# Instruction address generator

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

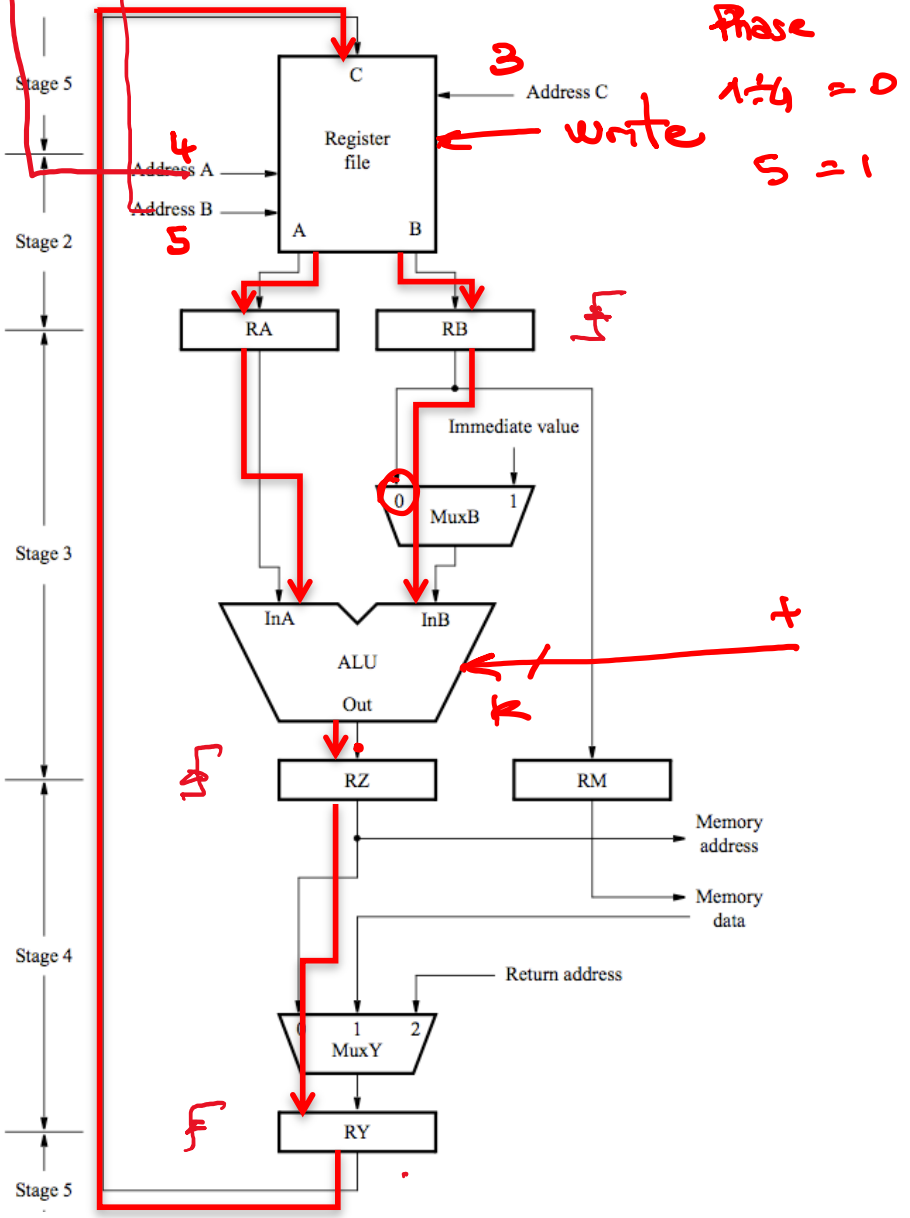



COMPUTE
MEMORY
WRITE
PC_tmp
tco
PC+4
RV
Link register
Register file
call
callr

# R-type Example: add R3, R4, R5

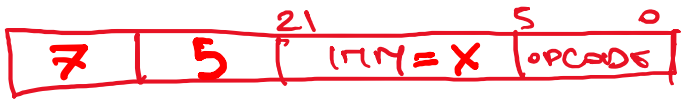



1. Memory address ←[PC], Read memory, Wait for MFC, IR←Memory data, PC ← [PC] + 4
2. Decode instruction, RA ← [R4], RB ← [R5]
3. RZ ← [RA] + [RB]
4. RY ← [RZ]
5. R3 ← [RY]



R0 = 0

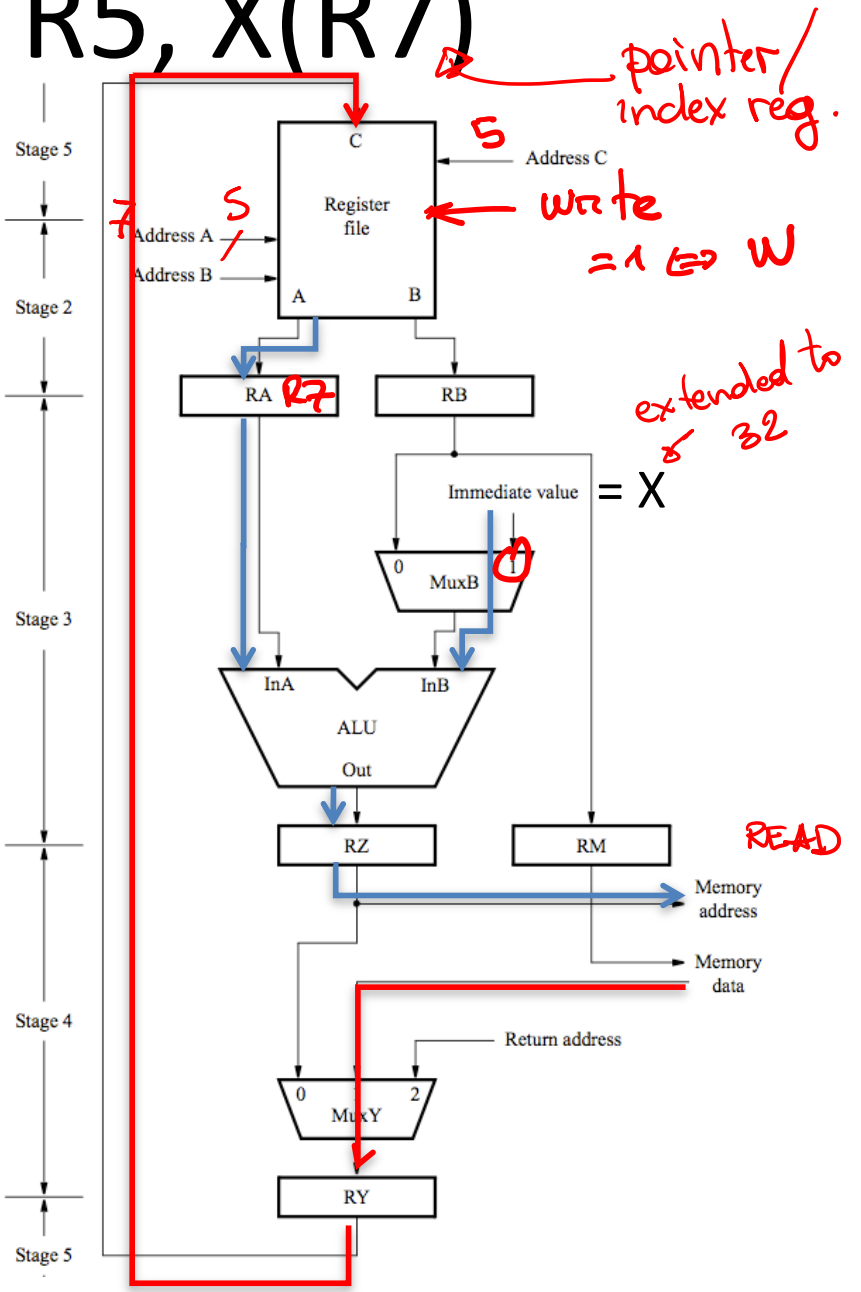
I-type




Displacement (16 bits signed)

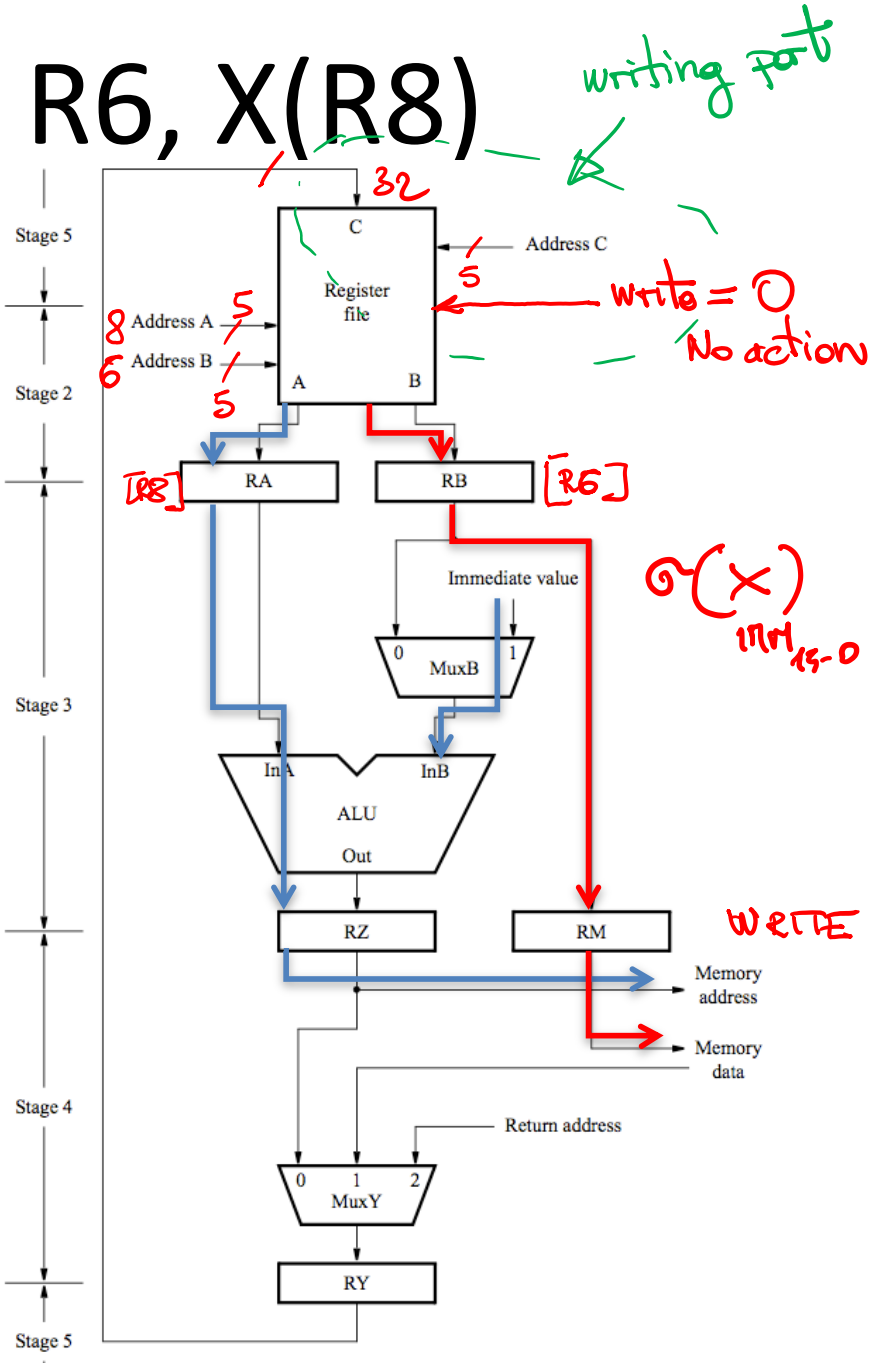
# Example: ldw R5, X(R7)

1. Memory address ← [PC], Read memory, Wait for MFC, 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]



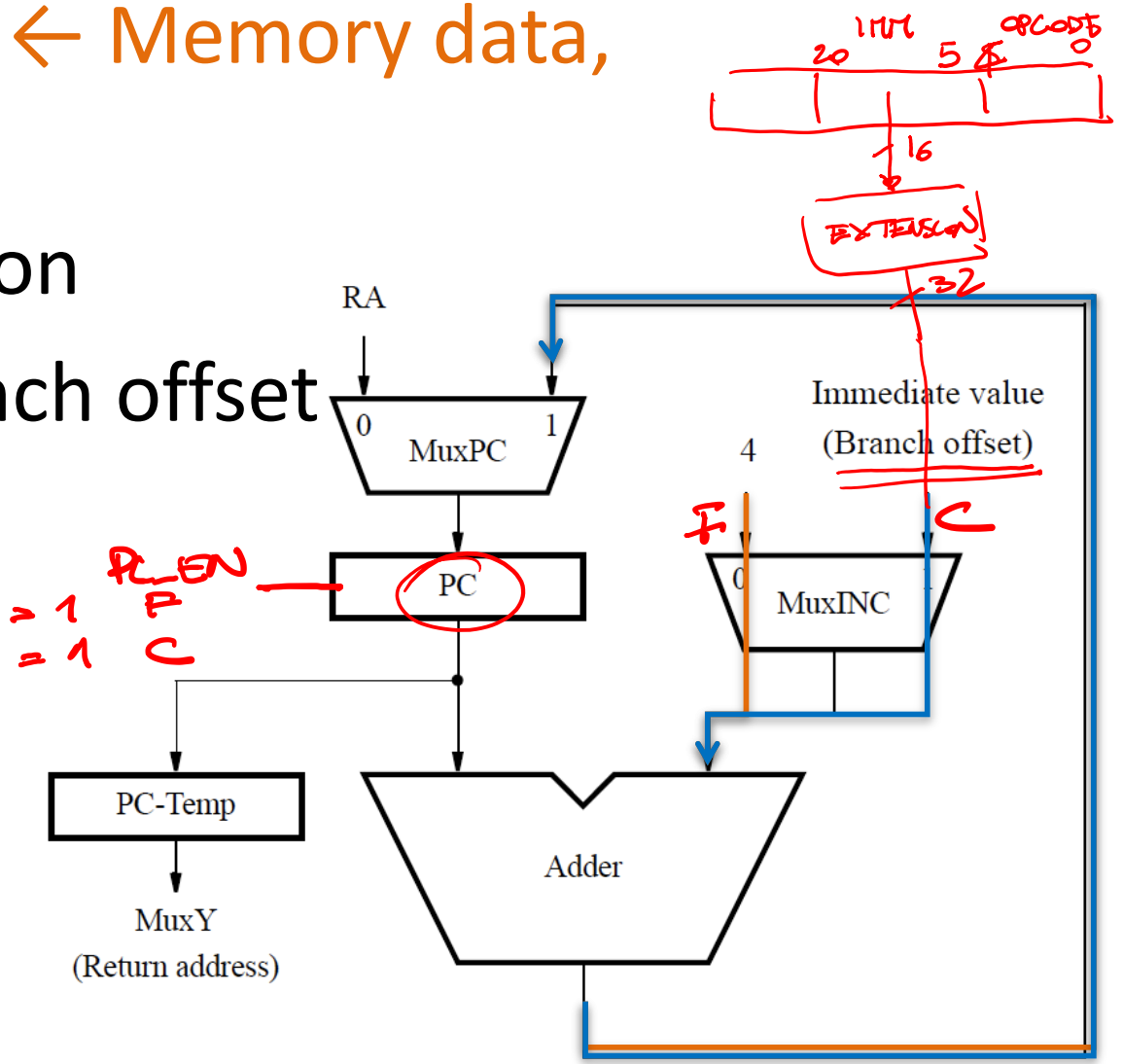
# Example: stw R6, X(R8)

1. Memory address ← [PC], Read memory, Wait for MFC, 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



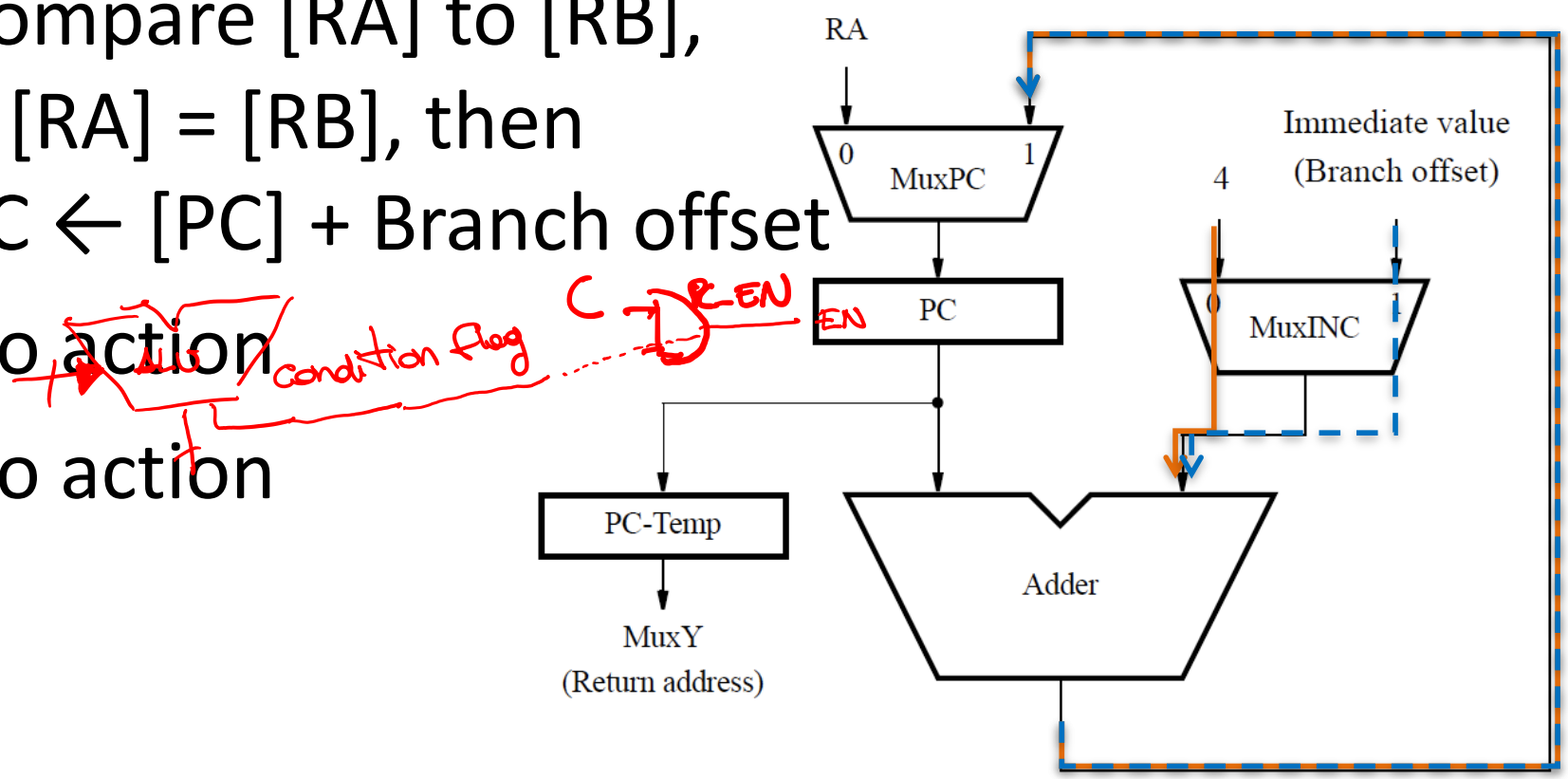
# Unconditional branch

1. Memory address ←[PC], Read memory, Wait for MFC, 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, Wait for MFC, 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

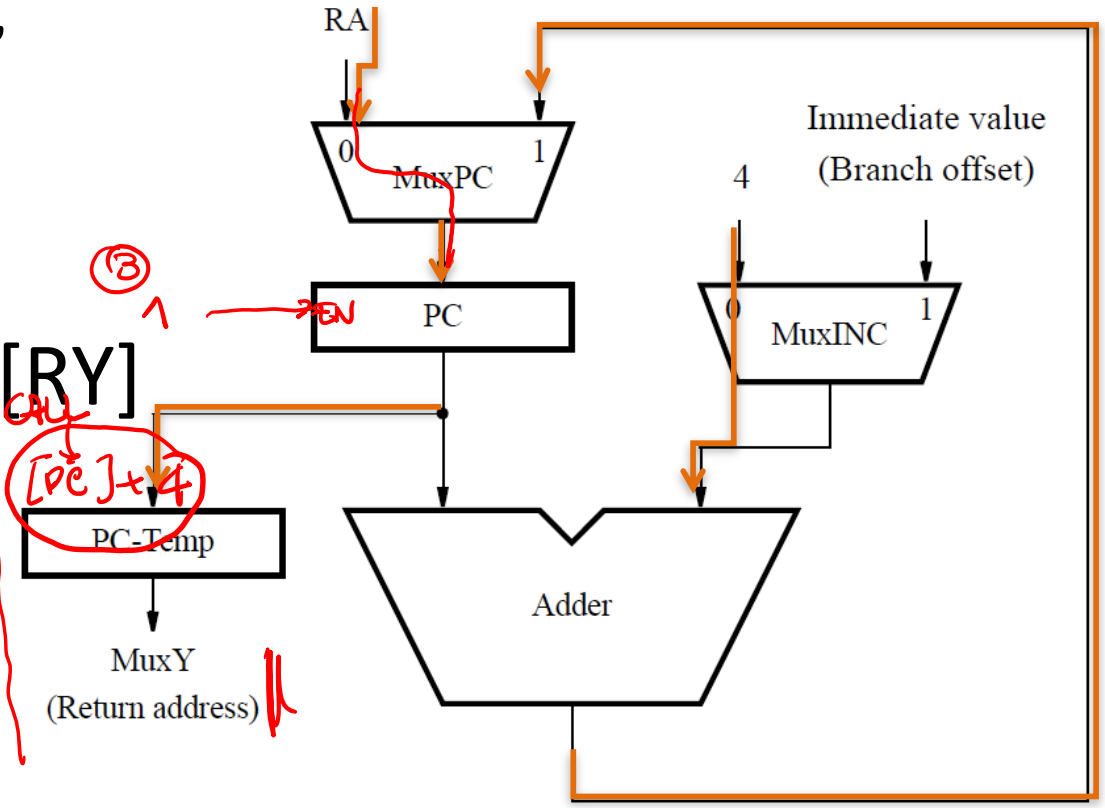


# Subroutine call with indirection:
# Call_register R9

callr

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]

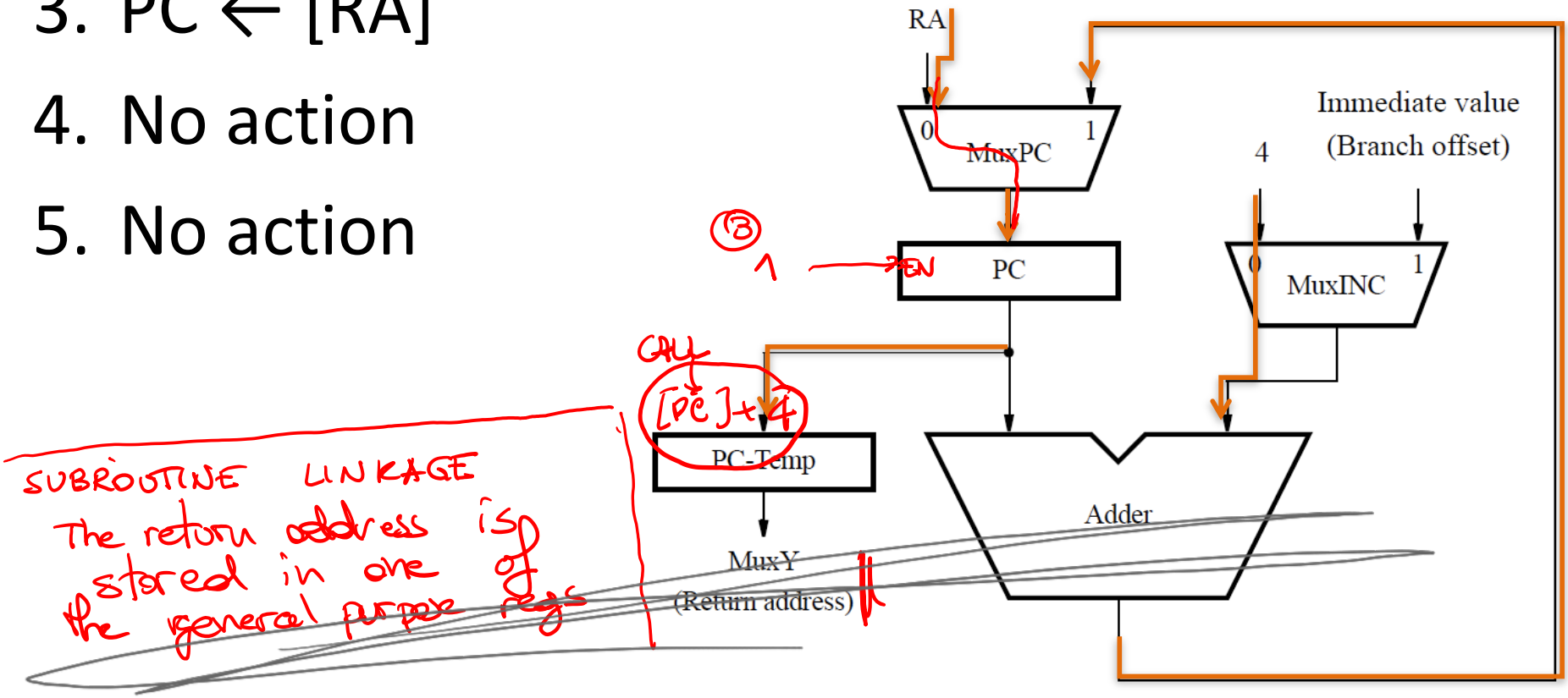



SUBROUTINE LINKAGE
The return address is stored in one of the general purpose regs

# Subroutine RETurn

CALL
RET

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



START

# Control signals

What are the various that need to be generated?

Assumption: we are executing only 1 instr. at a time (ex. IR won't change during execution)

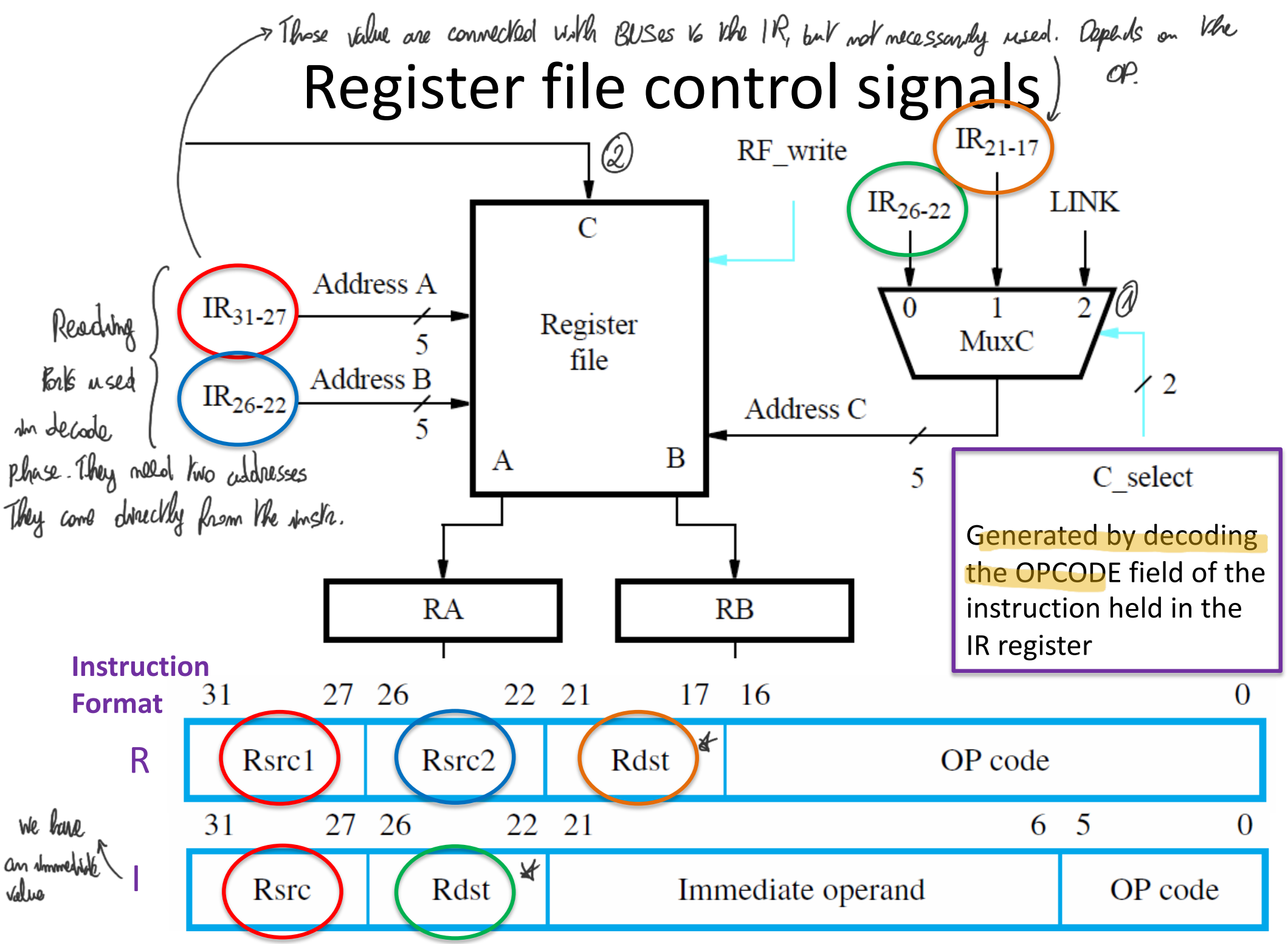
↓ This simplify the design

- Select multiplexer inputs to route the flow of data

Pipelining will introduce issues to be taken into account.

- Set the function performed by the ALU
- Generation of enable signals for the registers
- Determine when data are written into the PC, the IR, the register file, and the memory

# Register file control signals

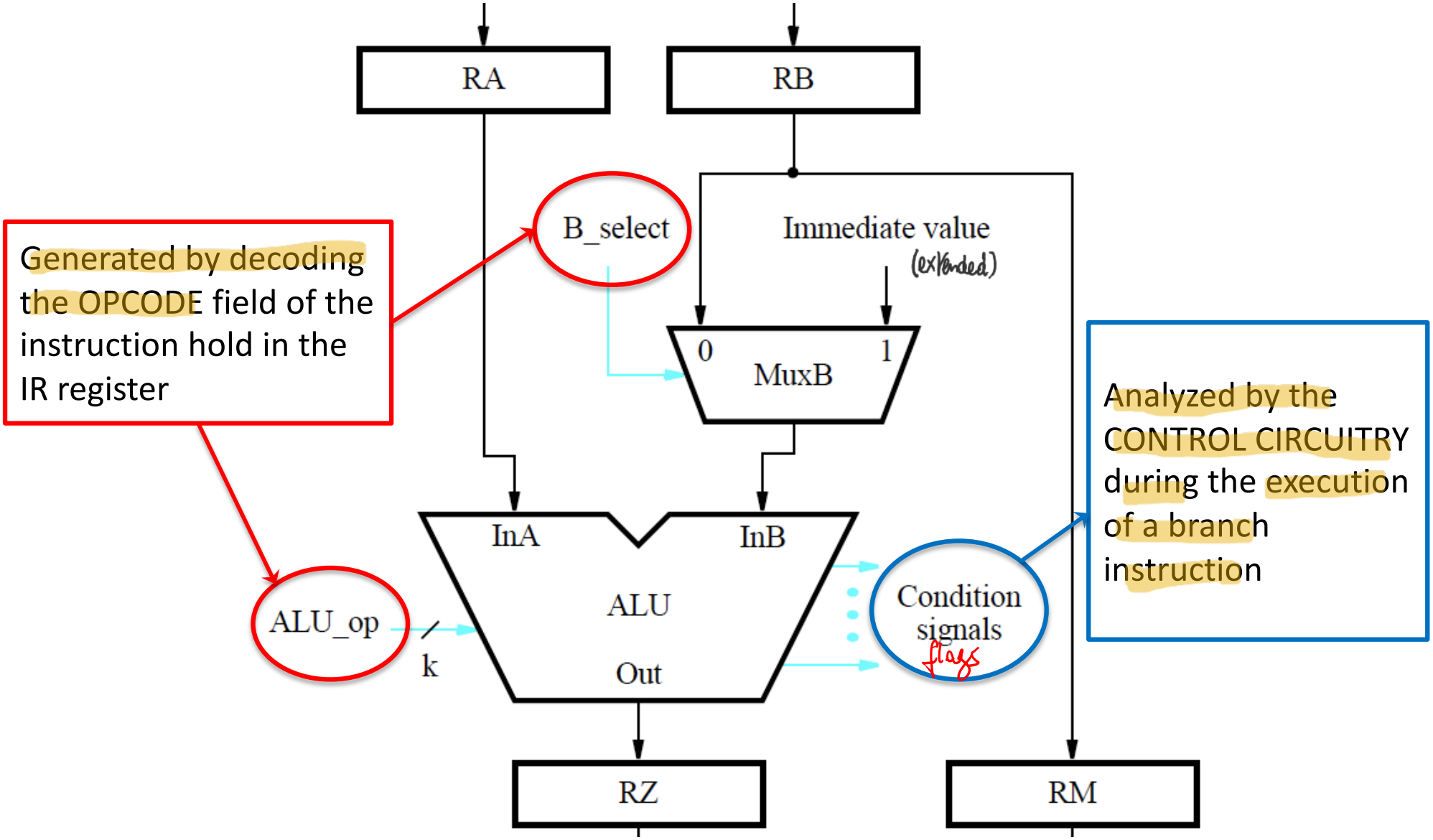


Under the assumption of no pipelining, * they can be retrieved anytime.
NOTE: ① is selected in case of a CALL. In ② we will have the return address.

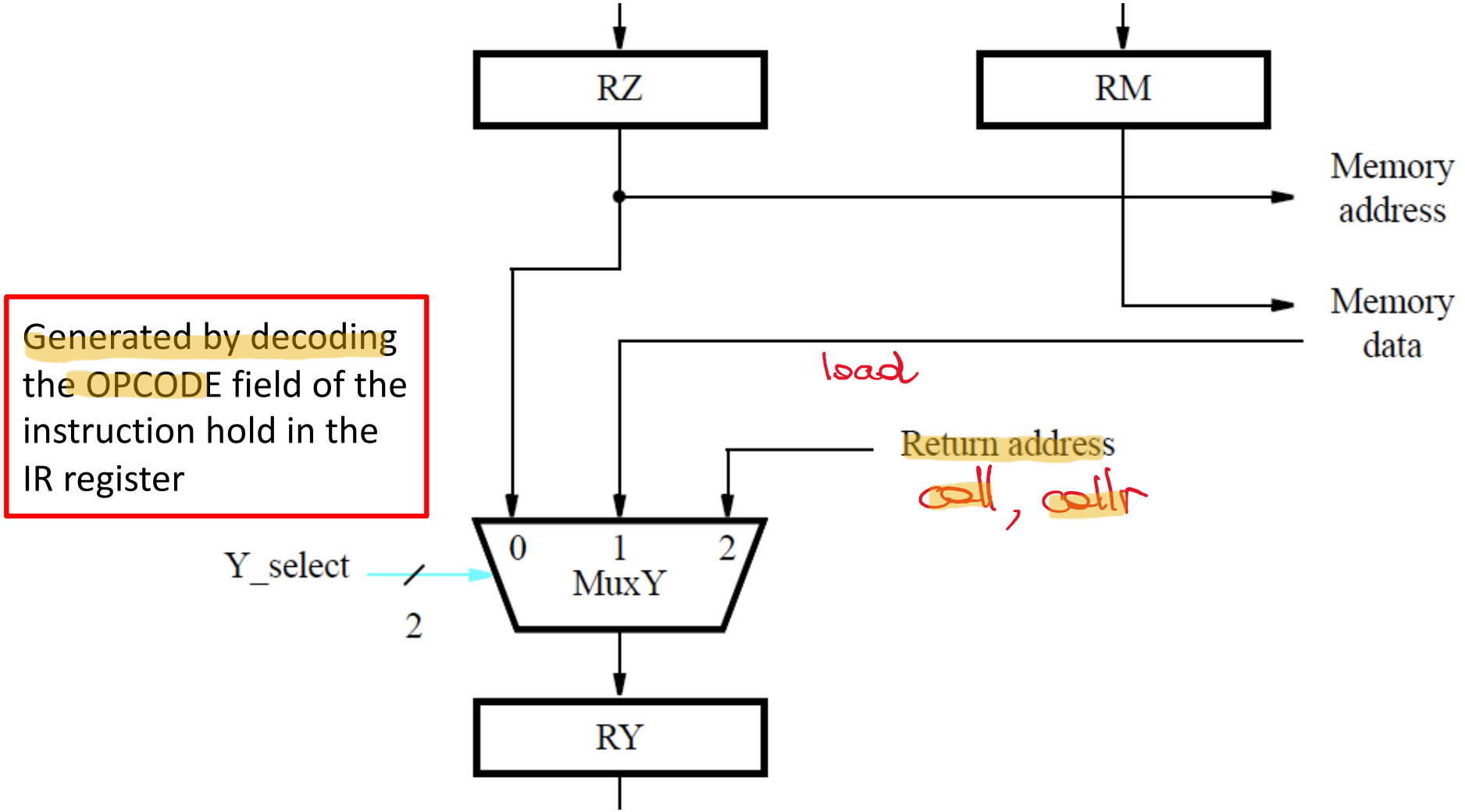
POSSIBLE OPTIMIZATION: look at implementation of the CALLR
CALLR is an R-Type instr. The info that comes in the LINK can be taken by the instruction. In general for a call/callr, the destination reg. is LINK reg. that is fixed at hardware.

COMPUTE PHASE

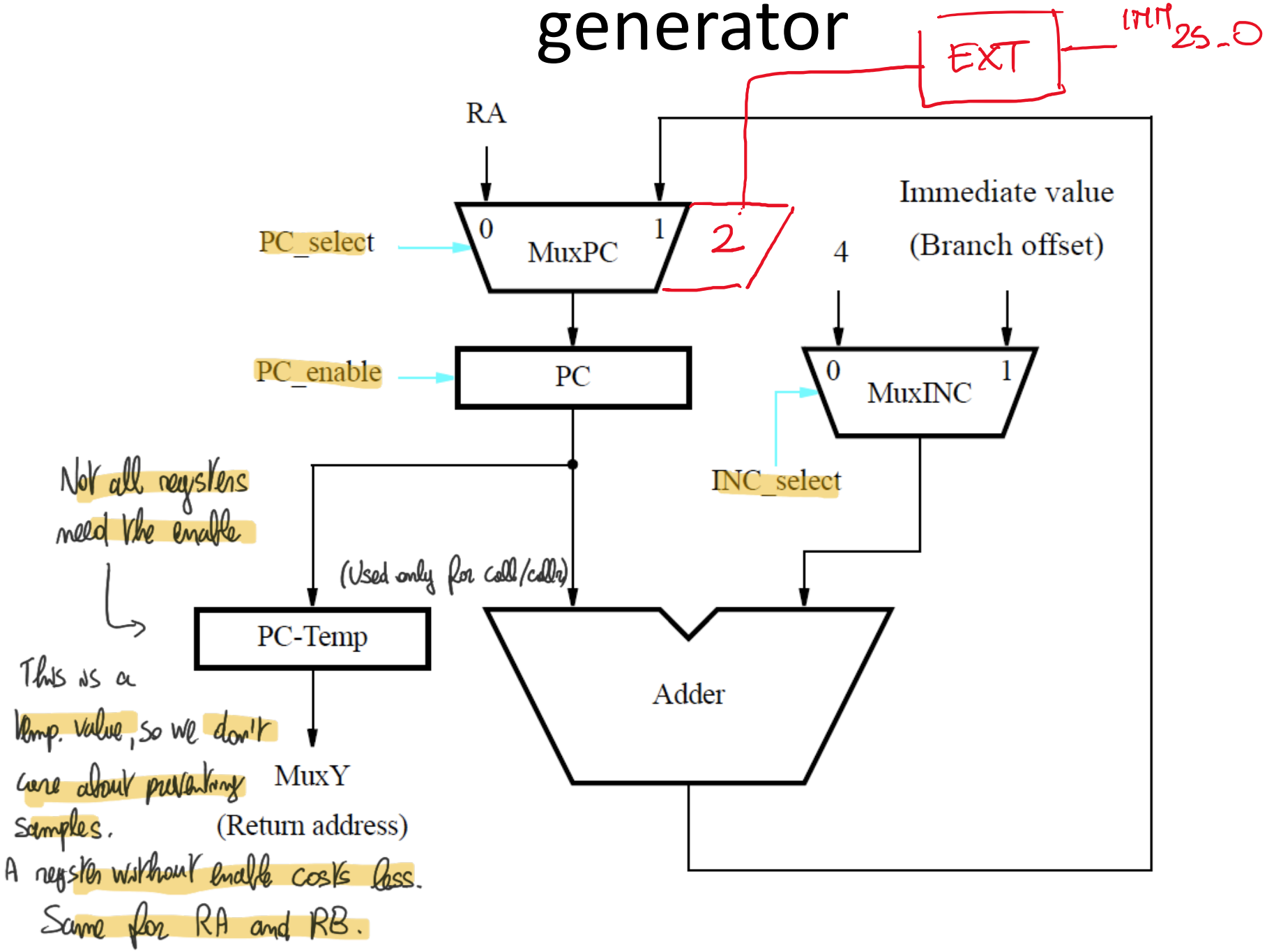
# ALU control signals



# 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

design by hand

description of a circuit using a language.

- Hardwired control involves implementing circuitry that considers step (ring) counter, IR, ALU result, and external inputs. You can just think the combinational network
- 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).

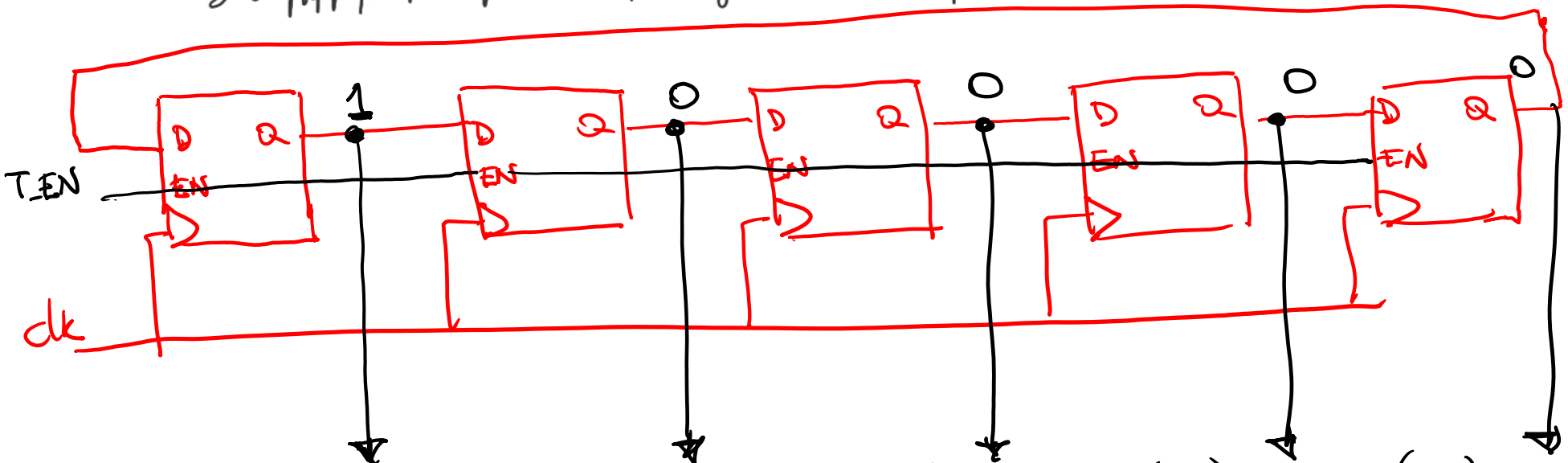
structure of a counter

# STEP COUNTER

RING

10000 is reset ① configuration

5 D flipflops that form a shift register with a feedback from last to first one.




| | (F) T1 | (D) T2 | (C) T3 | (M) T4 | (W) T5 |
|---|---|---|---|---|---|
| | 1 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 0 | 0 |
| | 0 | 0 | 0 | 1 | 0 |
| | 0 | 0 | 0 | 0 | 1 |
| | 1 | 0 | 0 | 0 | 0 |
| | | ... | | | |

① NOTE: When switch on this piece of hw, we want the first FF as 1 and all the others as 0.

Size of the ring depends on the number of stages.

Note: if every operation required one clock cycle we wouldn't need a mechanism to keep the counter fixed. But this is not always true. This is implemented with a T_EN signal. It's an enable for all flip flops. (load, store, interaction with the cache exc.)

[WRITE BACK VS WRITE THROUGH are used in cache]

JMP SLIDE 30: enable for register file

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

↳ Signal is one when we are in write and we are executing some instr.

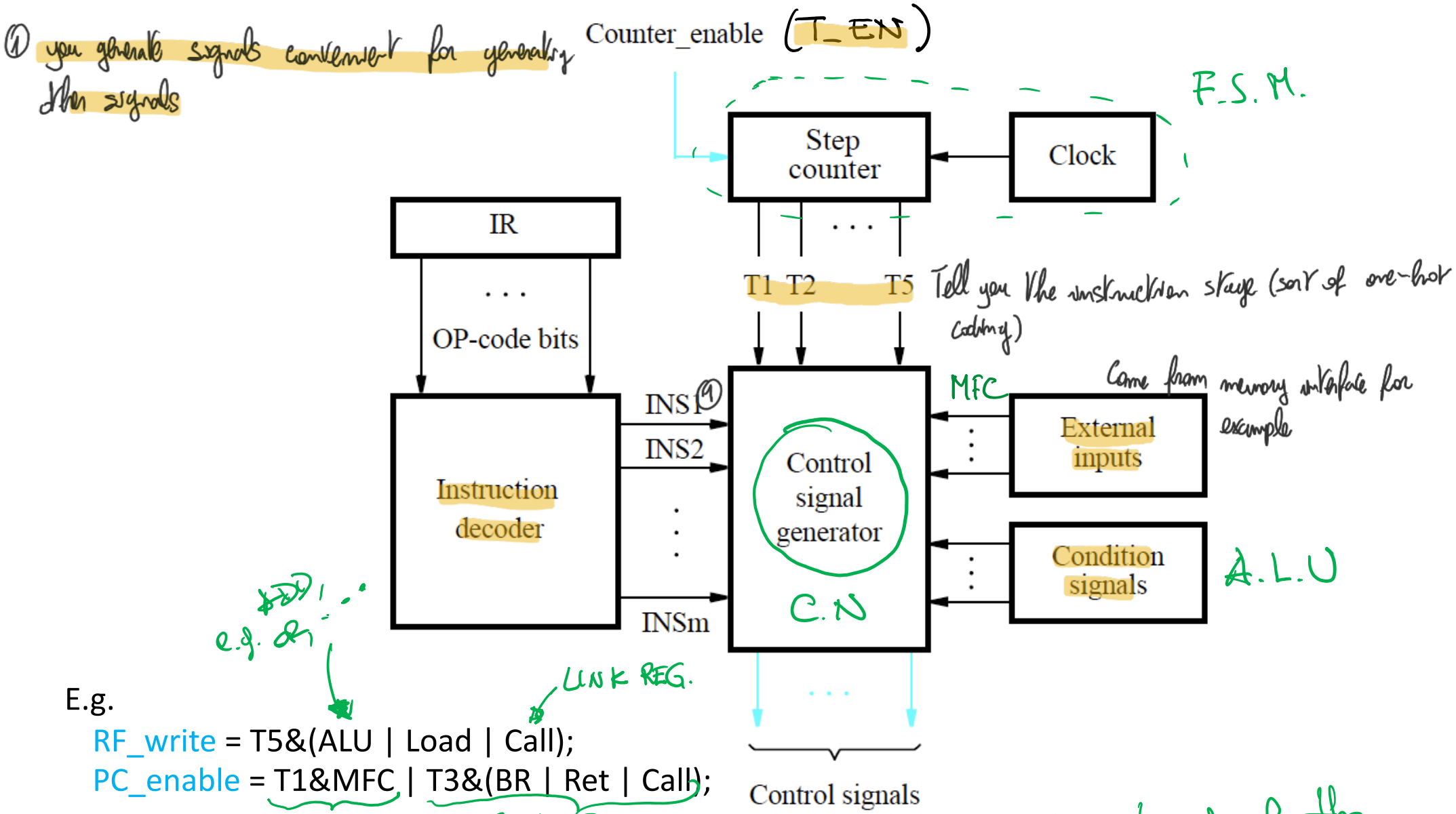
$\overline{T\_EN}$

PC enable is active in phase 1 or three

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

T1 → Fetch; MFC → Only if memory function completed is high; + → OR; T3 → Compute

# Hardwired generation of control signals




E.g.

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

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

# CISC processors

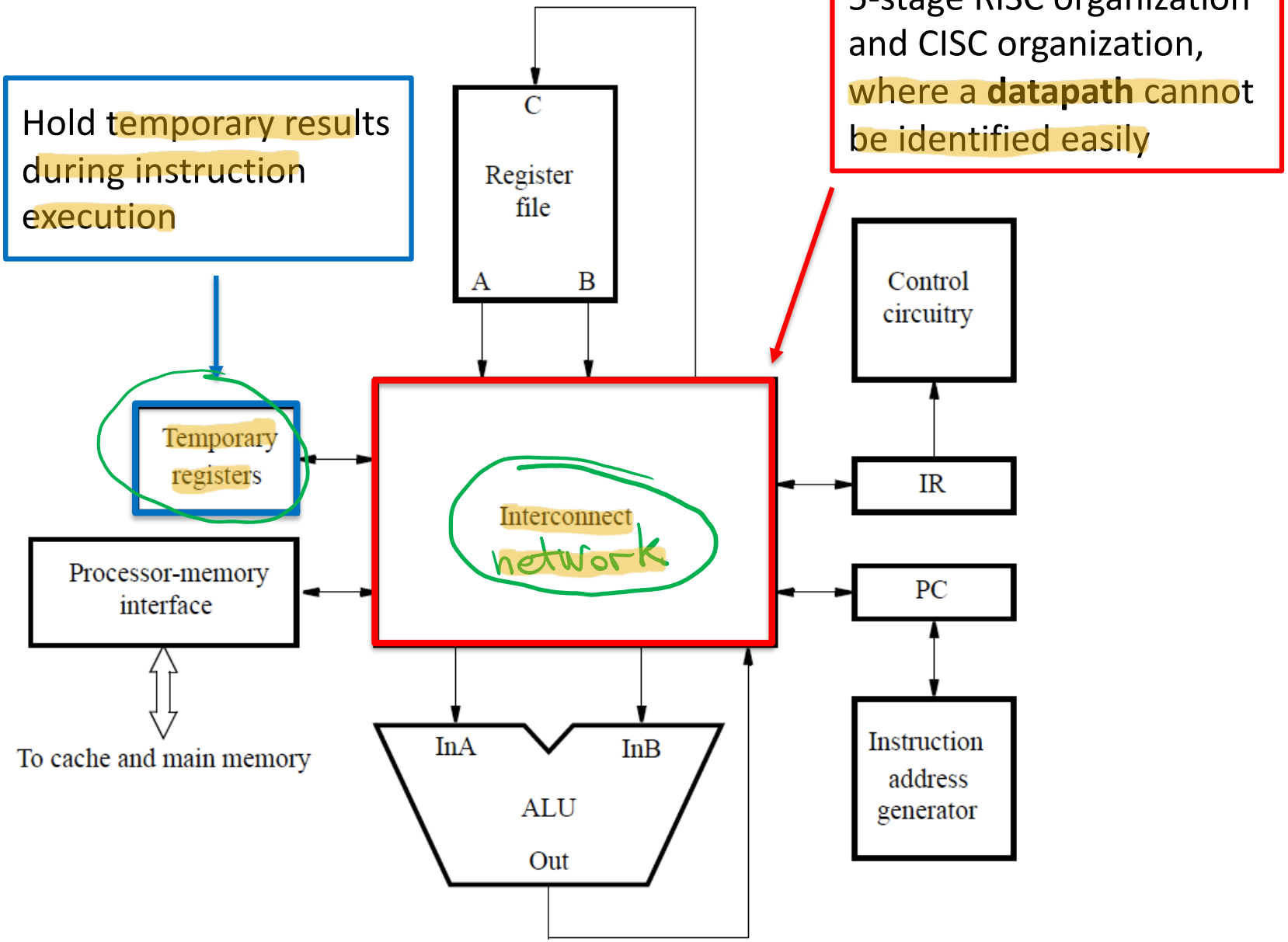
- CISC-style processors have more complex instructions. (longer than 1 word)
- The full set of instructions cannot all be implemented in a fixed number of steps.
- Execution steps for different instructions do not all follow a prescribed sequence of actions.
- Hardware organization should therefore enable a flexible flow of data and actions to accommodate CISC.

Recall that there's a clear flow of info in RISC. In a CISC the flow is more sophisticated and requires buses. Usually we have 3 buses in a CISC.

# Hardware organization for a CISC computer

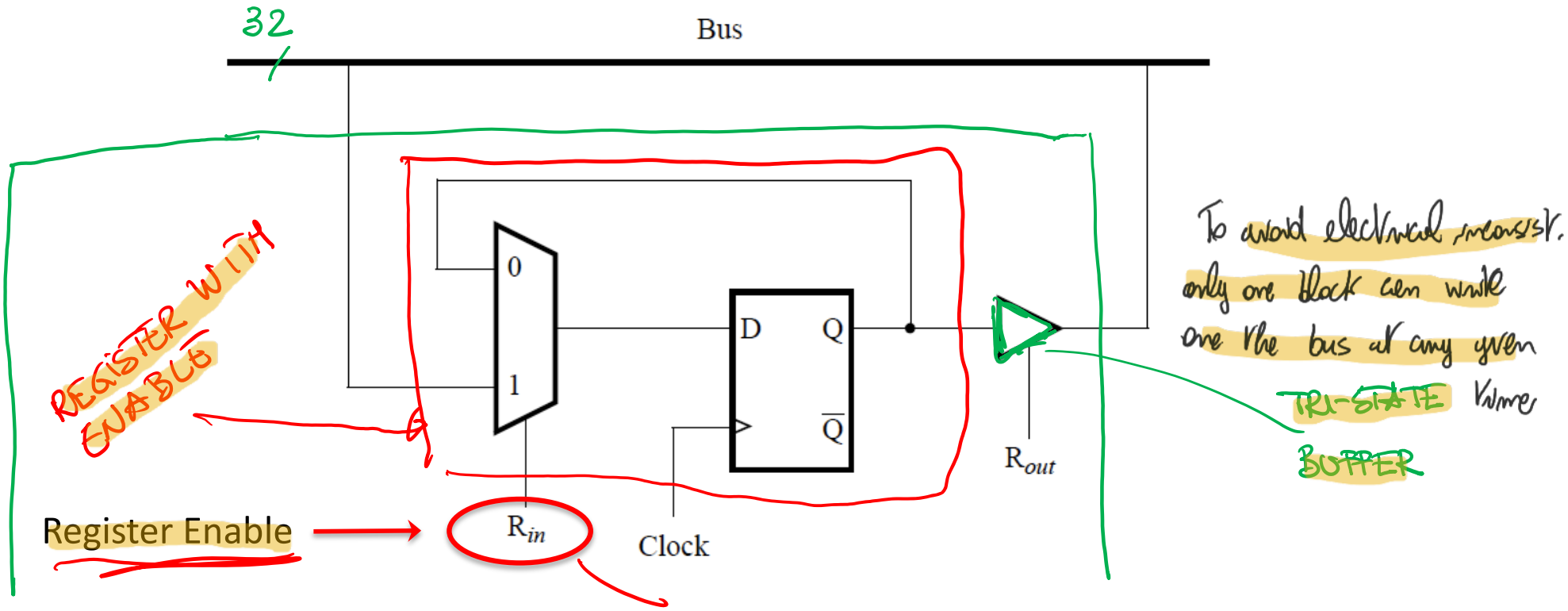
Hold temporary results during instruction execution

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

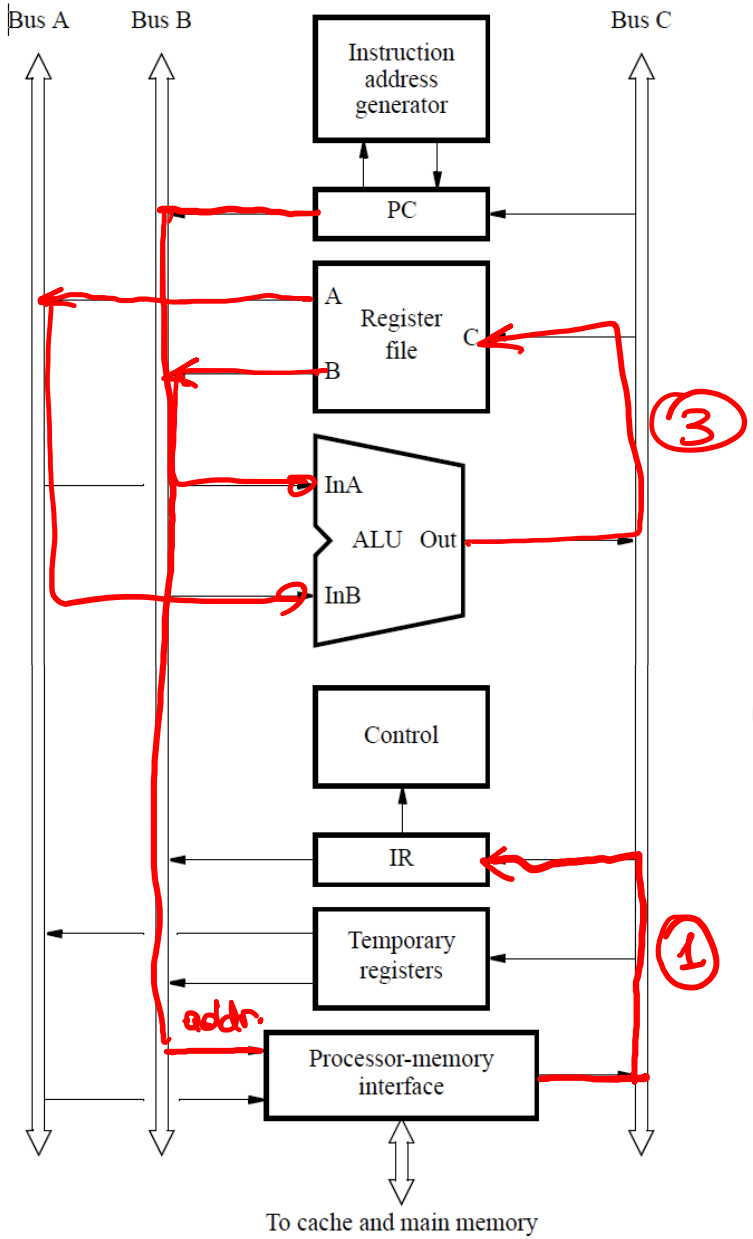


# Bus

- 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

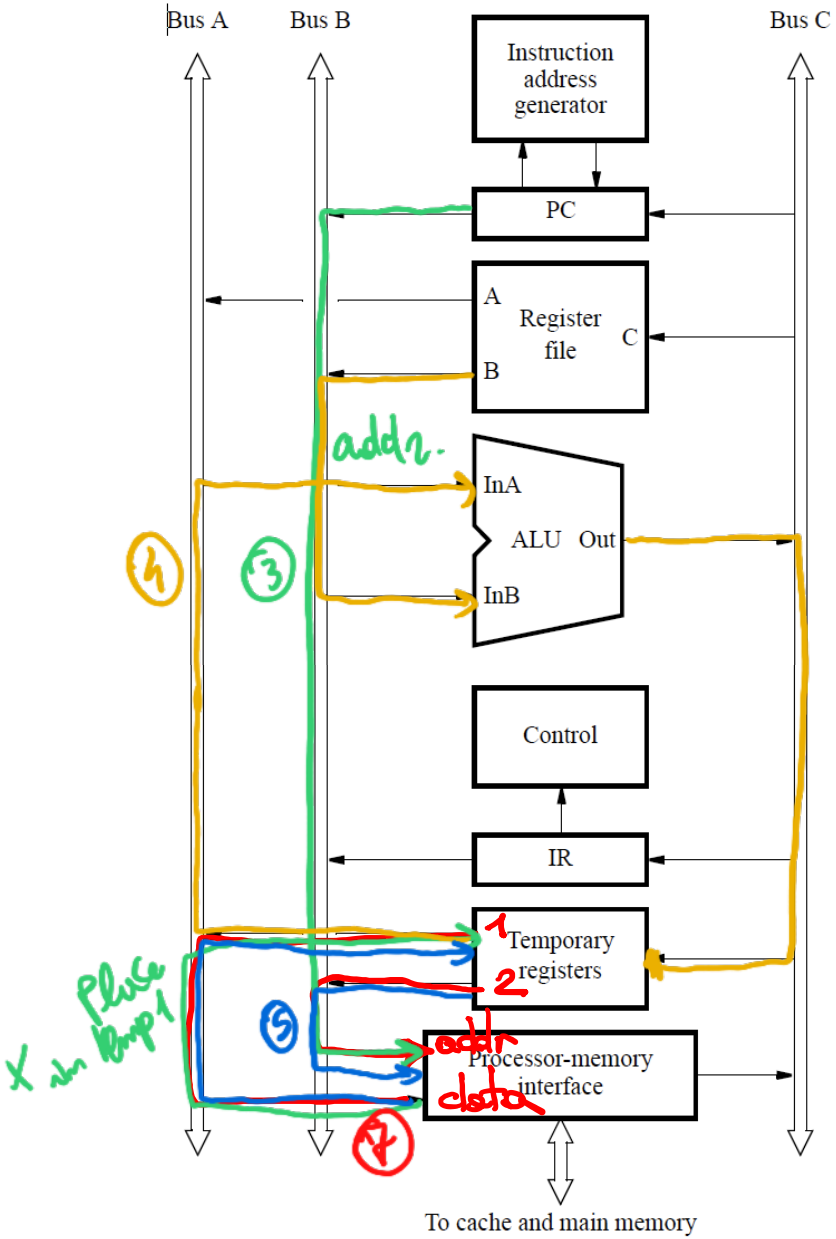
1. Memory address ← [PC], Read memory, Wait for MFC, IR ← Memory data, PC ← [PC] + 4
2. Decode instruction
3. R5 ← [R5] + [R6]

Prog. counter on bus B. What is read on memory goes on bus C in IR.

→ R5 and R6 need to be executed.

Just 3 clock cycles!

# A 3-bus interconnection network

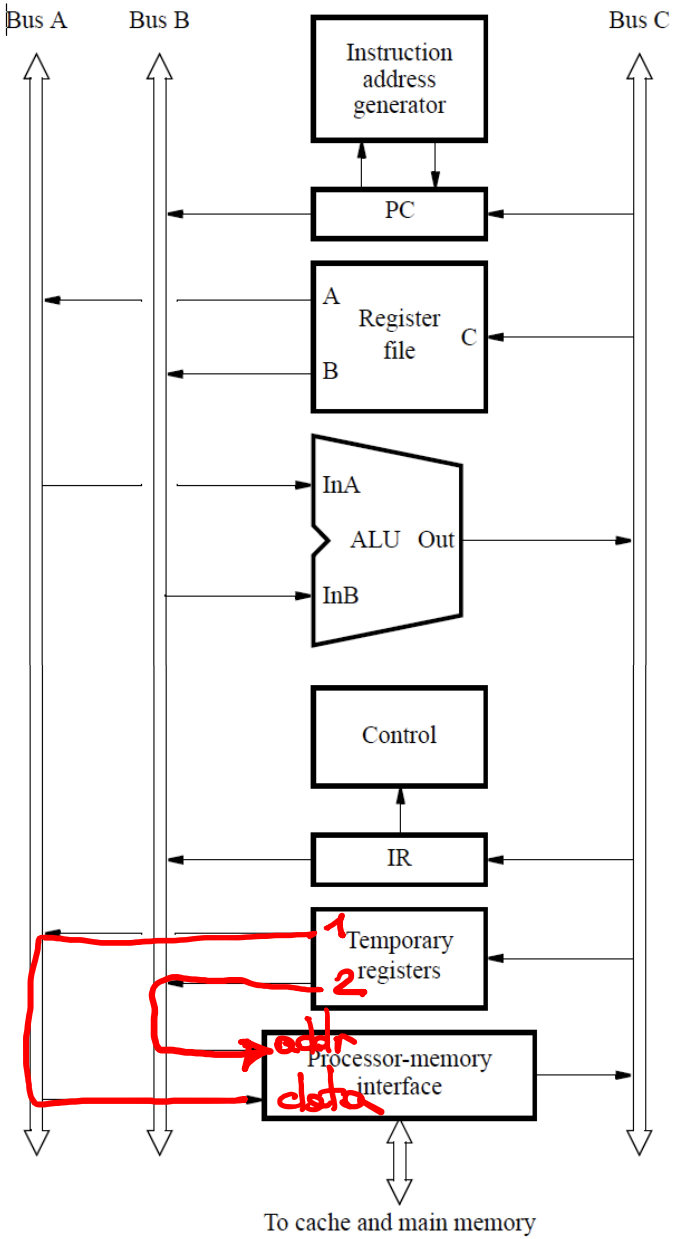



## Example 2*: And X(R7), R9

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

# 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