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:

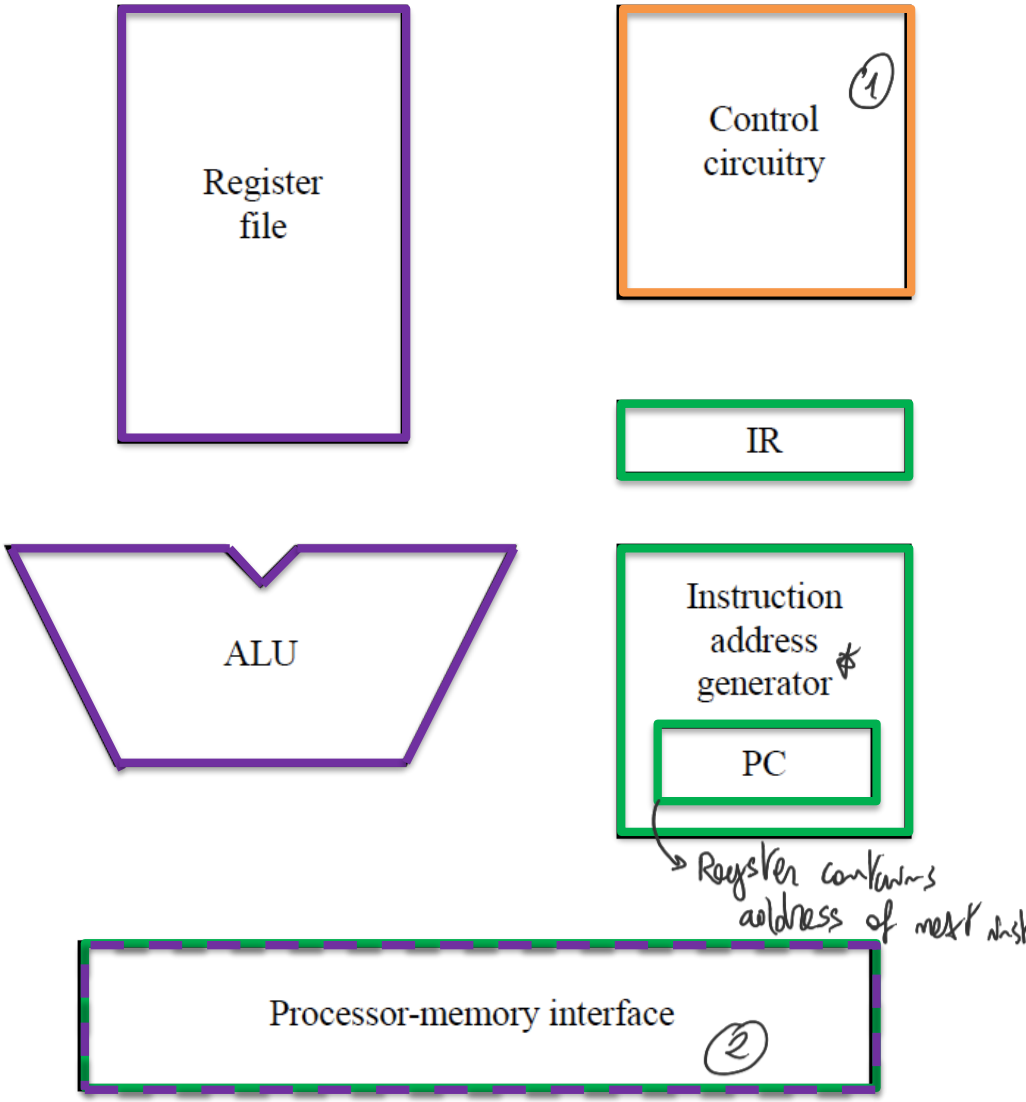
Friday 14-16. Please, contact me in advance before showing up. We can also arrange an appointment remotely on Microsoft Teams.

Possible architecture for implementation of a CISC arch.

COMPUTER ORGANIZATION

Processor's building blocks

- PC provides instruction address
- Instruction is fetched into IR
- Instruction address generator updates PC
- ALU performs some computation during execution
- Control circuitry interprets instruction and generates control signals to perform the actions needed.



* Hardware for determining the next instruction

During the fetch phase, the value of the memory is put in the instruction register after the program memory is read. After the access to memory, the address of the PC is incremented by 4. So we have to wait that.
This is called fetch phase. For the next phases the PC points to the next instr.

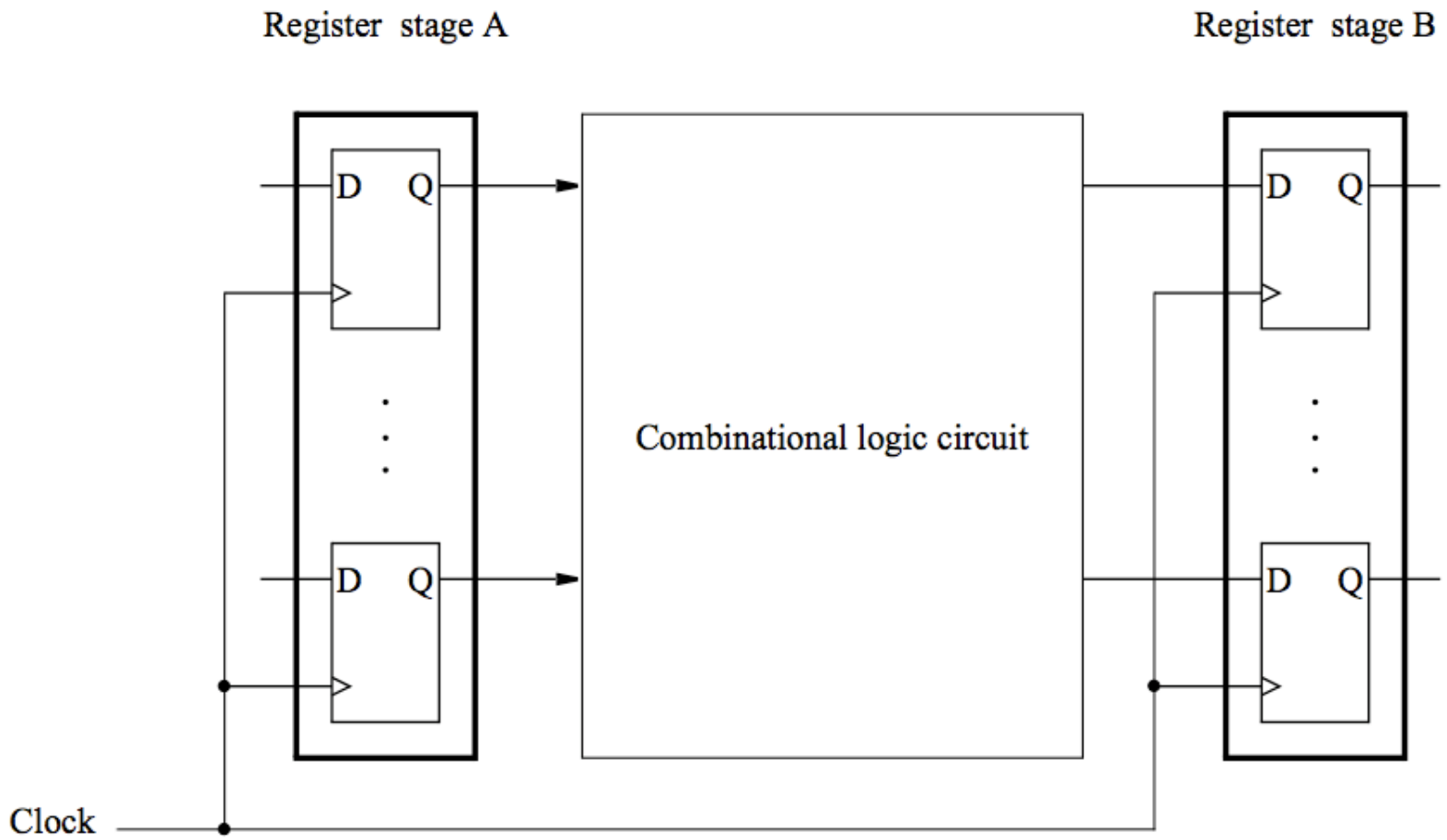
For some instr. (call, ret, jmp), the PC is further updated during the exec. of the instr., not only the fetch phase.

① Looks at OP code of the instruction to execute it. If needed, the ALU performs OPs.

② Two memory interfaces for Harvard architecture.

A digital processing system

- datapath

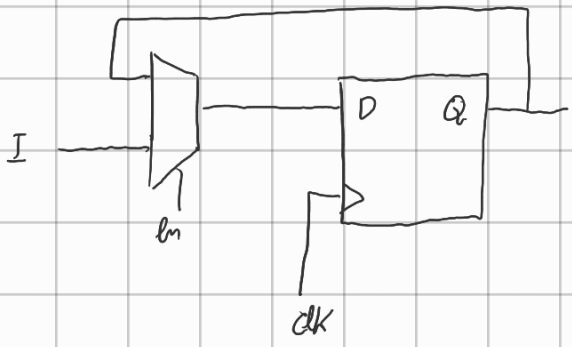


In a digital system you always find a structure like this: you have a register that provides data to a CN, which result is then caught by a catching registers.

Reg: collection of edge triggered D Flip Flop with the same clock signal.

3 times to describe performances of a register: t_s, t_h, t_{co}.

By construction we know that $t_{co} > t_h$.



If en = 0, the launch and the catch registers commute.

So, to sample correctly the input, d has to be stable t_h after the edge. So by construction $t_{co} > t_h$, to make the feedback possible. The hold constraint has to be verified at the same edge in which the out changes. The setup constraint on the other hand, verified at the next edge.

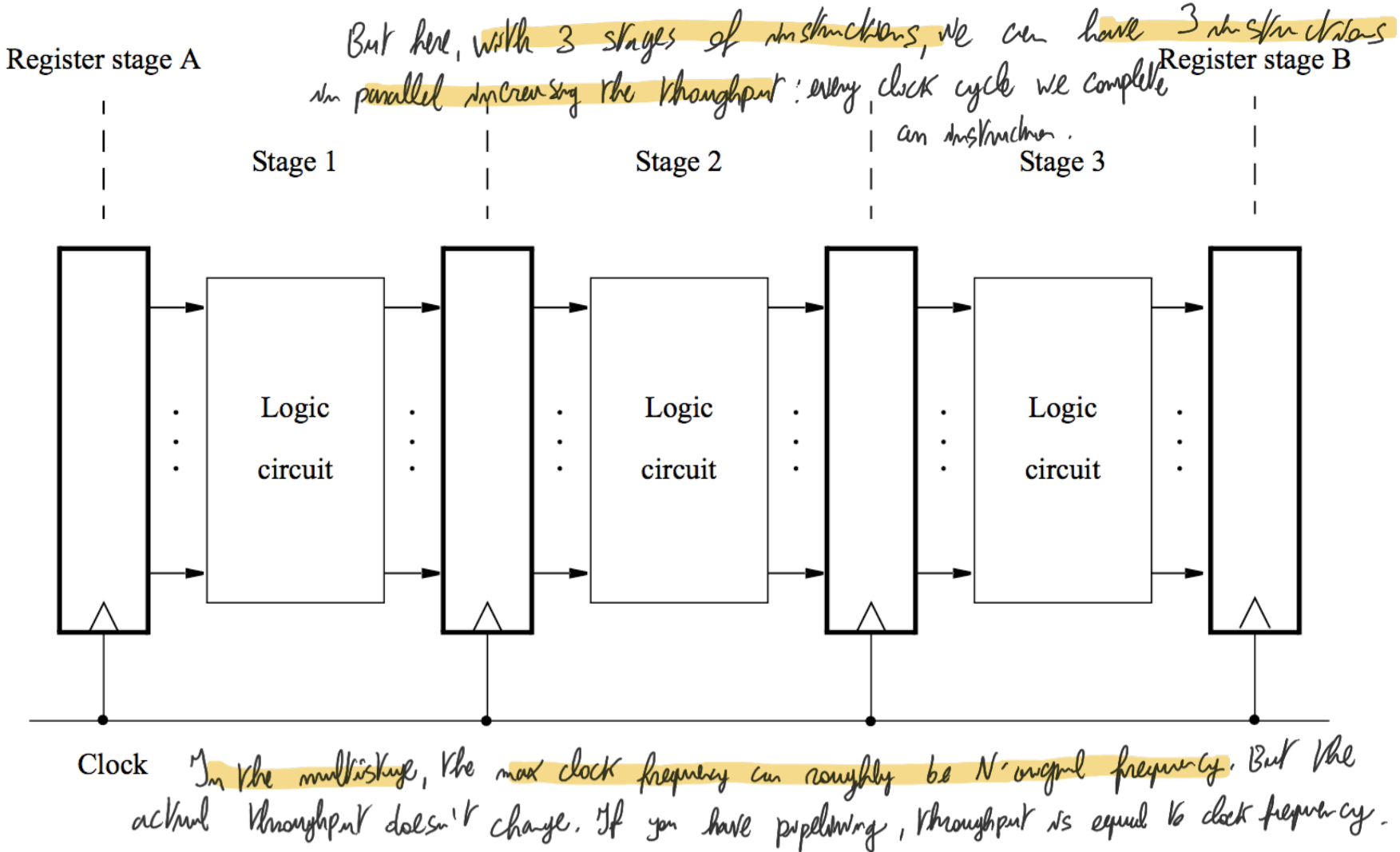
SNAP BACK: We want the catching register to correctly catch the out of the combinational network. Input has to be stable already before t_s.

$T_{clk} \geq t_{co} + t_{pd,max} + t_{su}$. This condition is to be satisfied for every datapath in the circuit. So we have to take the critical datapath, which is the one that has the max $(t_{co} + t_{pd,max} + t_{su})$.

In general $t_{pd} >> t_{co}$ and t_{su}, so it's the comb. network that defines the criticality. If we wanted to increase the clock frequency, we could split a complex data path in multiple paths by splitting the CN in a number of sub CN.

A multi-stage digital processing system

- datapath



Throughput: # of instr. completed each second.
Latency: number of clock cycles needed to complete execution of 1 instruction.

We will work on a 5 stage org.

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

5 STAGES:

FETCH DECODE COMPUTE MEMORY WRITE

Every instr. goes into every phase, even when this means no actions.
In this case the CN is a short circuit.

With those 5 stages we can execute all instructions.

A memory access instruction:

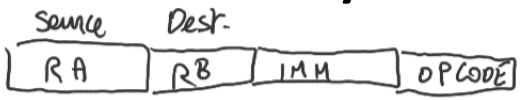
Signed ext. to 32bits

I-type

ldw R5, X(R7)

R5 ← Mem32[6(X) + R7]

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read the contents of register R7 in the register file.
3. Compute the effective address = X + [R7].
4. Read the memory source operand at effective address.
5. Load the operand read from memory into the destination register, R5.



NOTE: To block the value of a register we do not enable W. Or only to write we enable W

For R-Type, A,B are source and are put at most sig. bits, then dest. register.

A computational instruction:

add R3, R4, R5

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read registers R4 and R5.
3. Compute the sum [R4] + [R5].
4. No action.
5. Load the result into the destination register, R3.

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

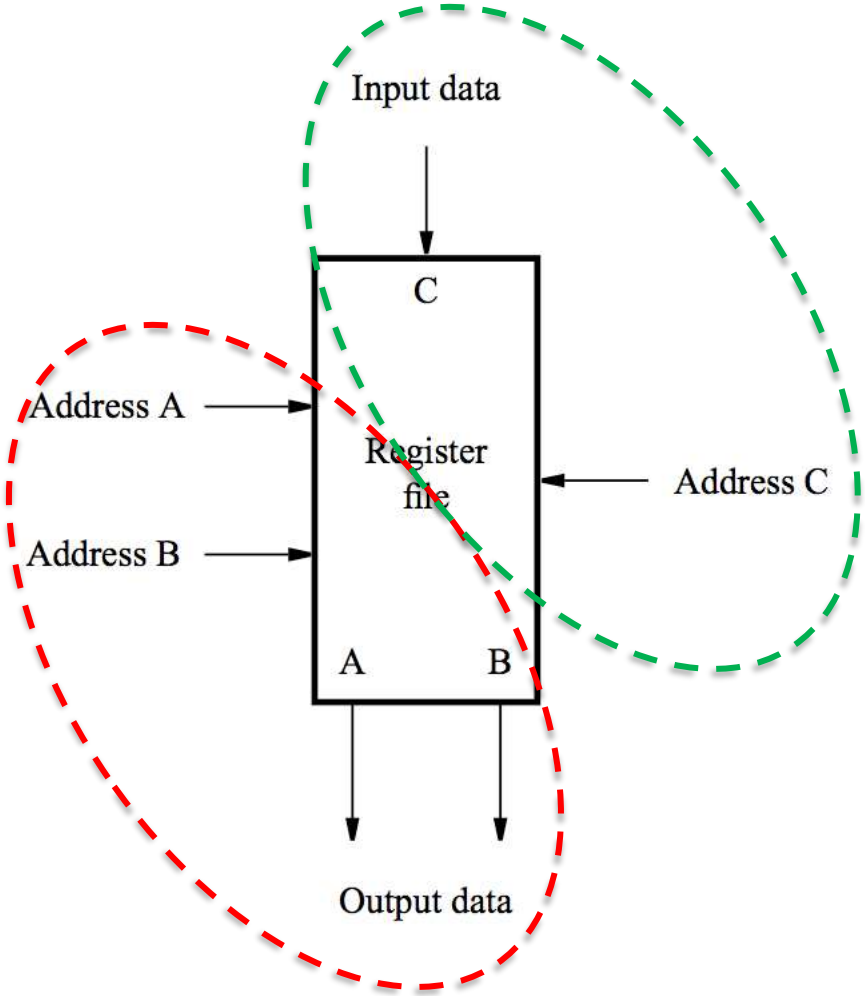
5-stage Architecture of a RISC Processor

1. Fetch an instruction and increment the program counter.
2. Decode the instruction and read registers from the register file.
3. Perform an ALU operation.
4. Read or write memory data if the instruction involves a memory operand.
5. Write the result into the destination register.

- This sequence determines the hardware stages needed.

Hardware components: Register file

- 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.



32 REGISTERS

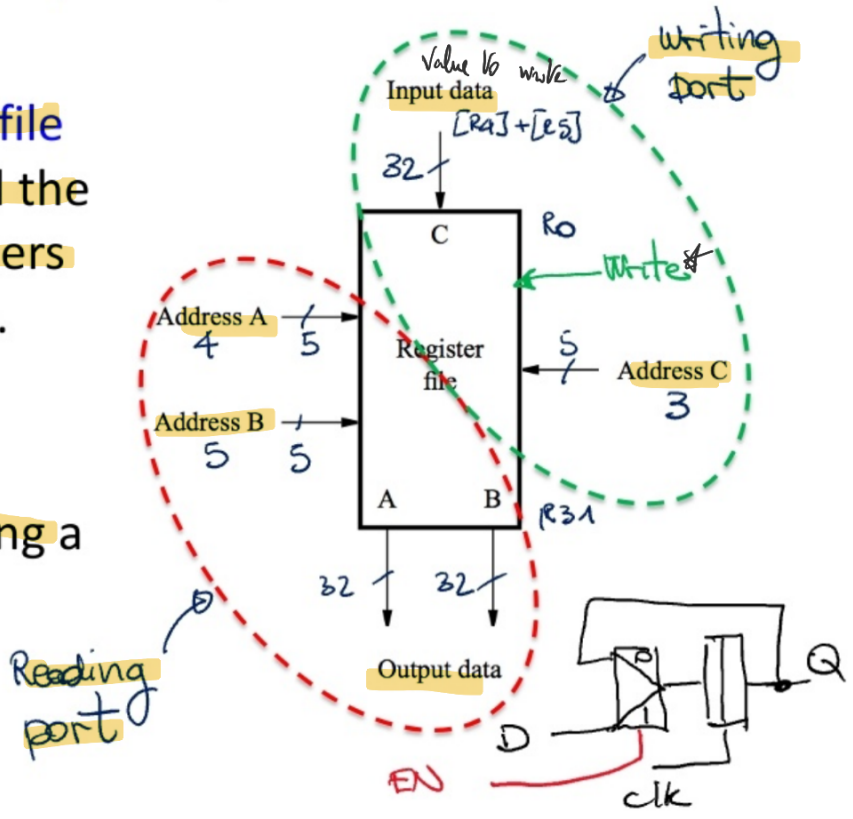
Hardware components: Register file

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



* Active only in the right phase and when instruction has output register. Generate a control circuitry. Write can be seen as enable for all the 32 reg. When write is 1, reg. specified in address C is updated.

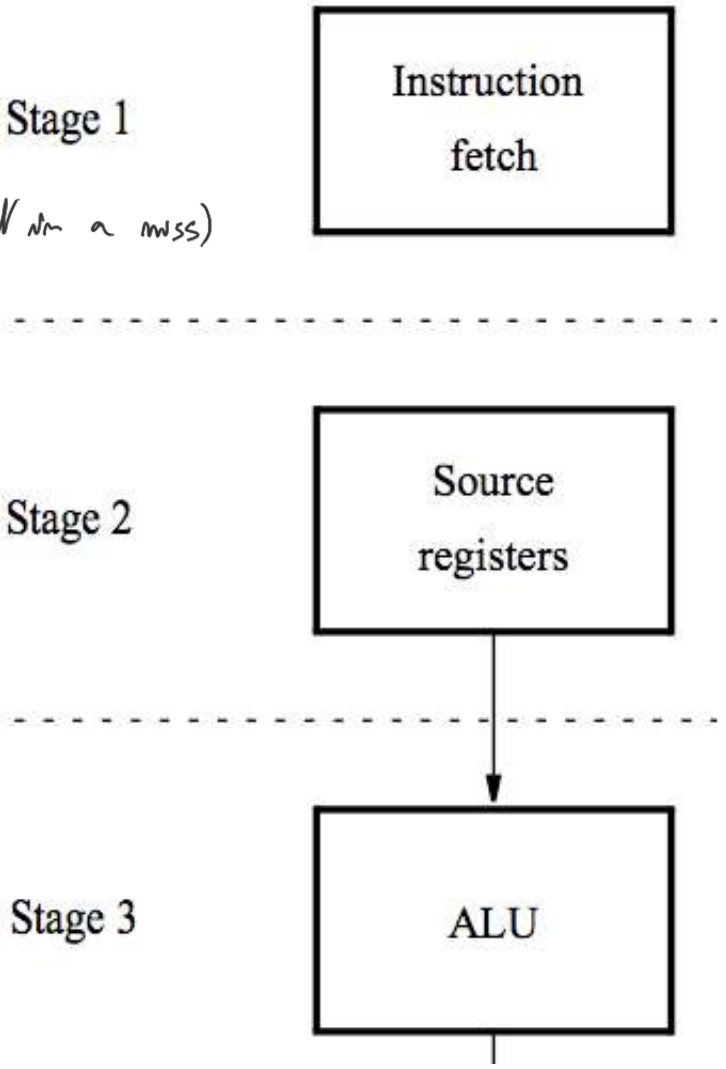
A 5-stage implementation of a RISC processor

- Instruction processing moves from stage to stage in every clock cycle, starting with **fetch**.

(cache might result in a miss)

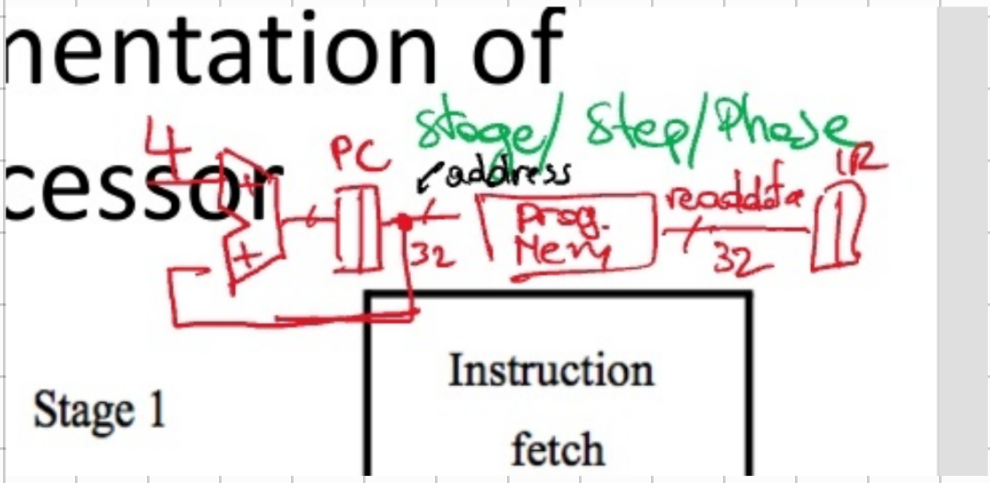
- The instruction is **decoded** and the source registers are read in stage 2.

- **Computation** takes place in the ALU in stage 3.



nentation of

cessor



Stage/ Step/Phase

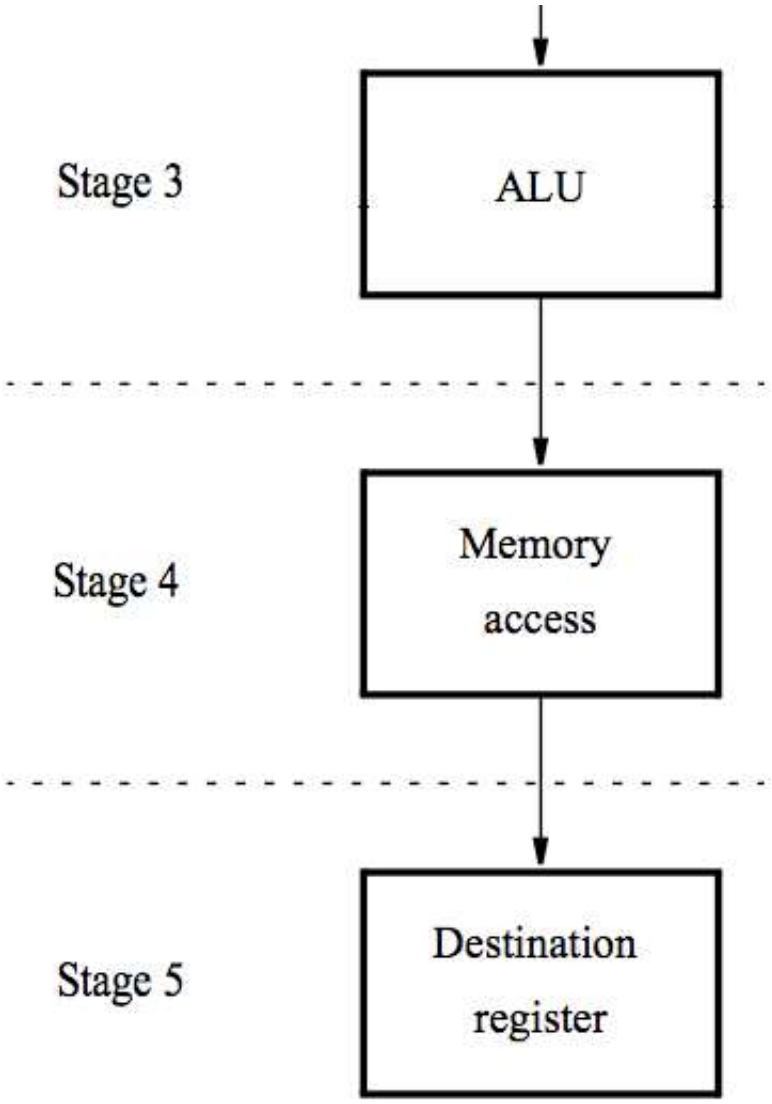
Program counter also has an enable signal.

push RS

```
subw  SP, 16
stw   RS, (SP)
stw   R4, 4(SP)
```

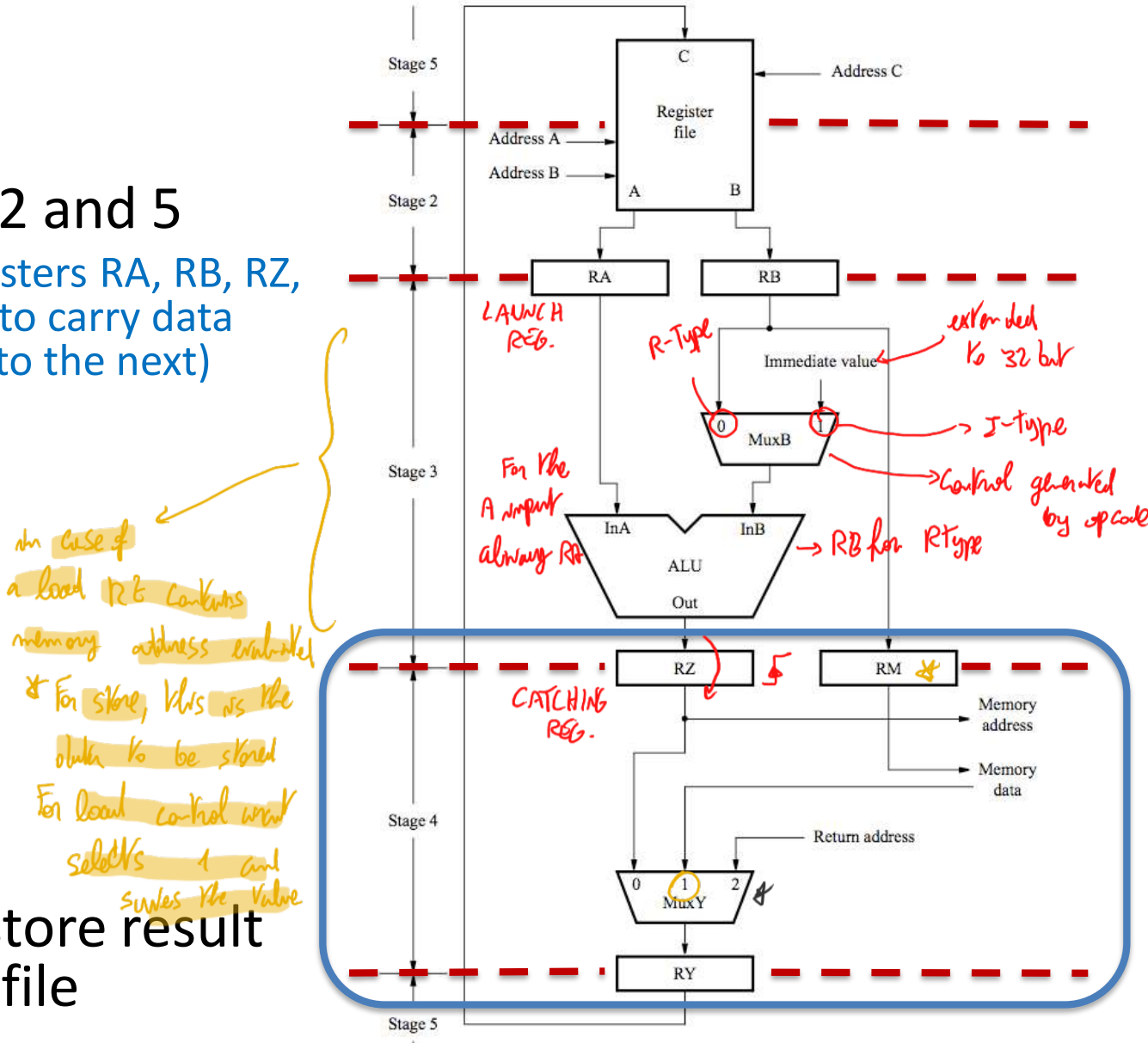
A 5-stage implementation of a RISC processor

- …
- 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)
- ALU stage
- Memory stage
- Final stage to store result to the register file



* In case of a call instruction, we need the return address to be stored.

Here there is no way to increment the pointer in a register. We have a simplified hardware losing some features.

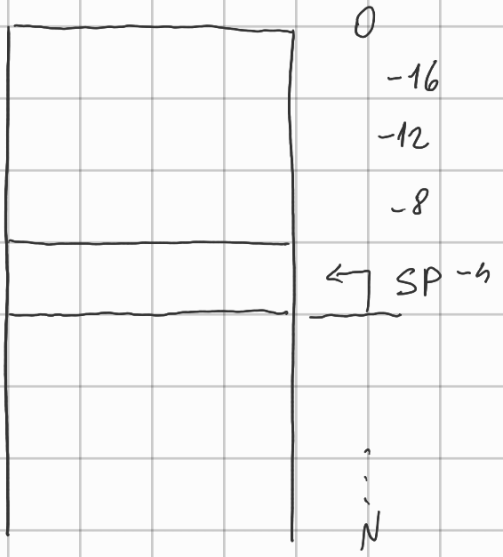
Push ish, as pseudo-instr: decrement by 4 the ptr. and store.

Pop: increment by 4 and then load.

This can be optimise: the stack pointer can be decremented by a big value, let's say 16. Then uses +4, +8, +12 to access those cells.

Remember, compiler have a lot of responsibility.

What have we lost here? In some instruction set, when we have something in the memory, there might be the possibility to increment or decrement value from memory.



Push Rx: subi SP, SP, 4
stw Rx, (SP)

Pop Rx: ldw Rx, (SP)
addi SP, SP, 4

So: if you do:

```
push R10
push R11
push R12
push R13
```

Would require 8 instr. but can be optimised.

What you do? You first reserve 4 words in the stack:

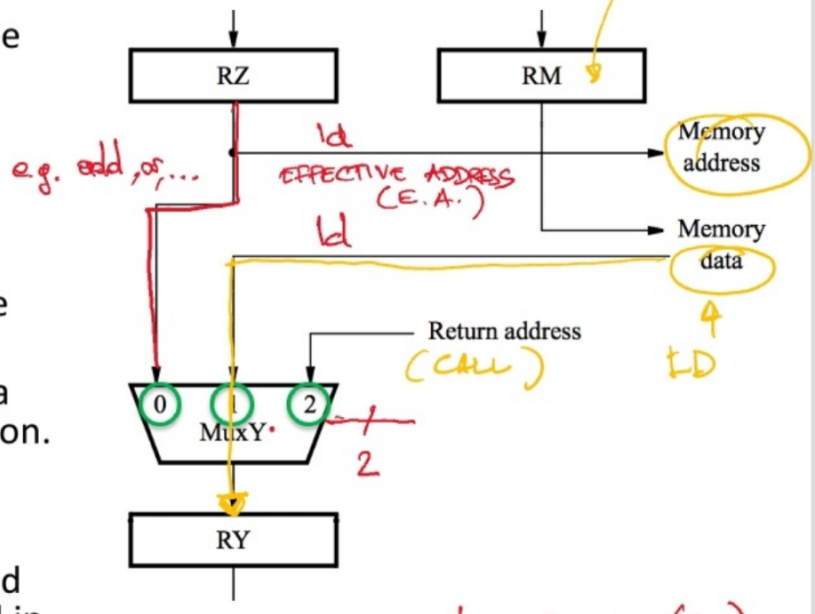
```
sub sp, sp, 16
str r13, (sp)
str r12, 4(sp)
str r11, 8(sp)
str r10, 12(sp)
```

In case of pop, better to use add at the end to modify the stack to not lose it.

First you reserve space in stack, then you save values. In pop, if you free space first you may lose variables.

Memory stage

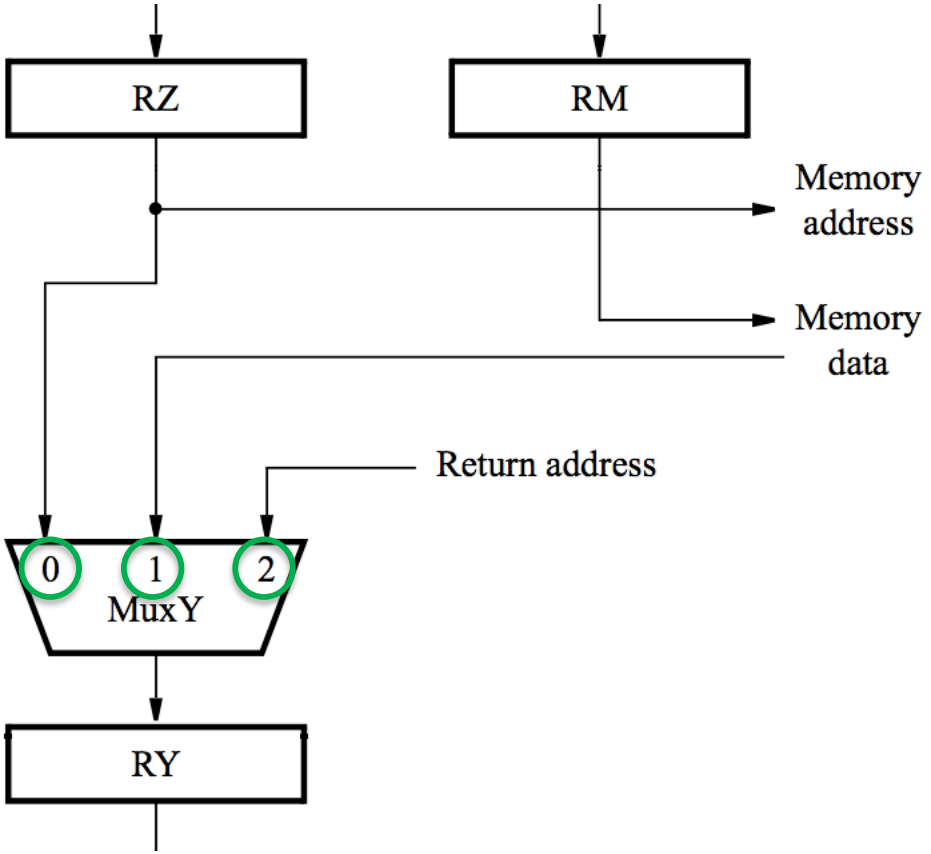
- **For a calculation instruction**:
  - MuxY selects [RZ] to be placed in RY.
- **For a memory instruction**:
  - RZ provides memory address, and MuxY selects read data to be placed in RY.
  - RM provides data for a memory write operation.
- **In subroutine calls or exception handling:**
  - Input 2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



STOP

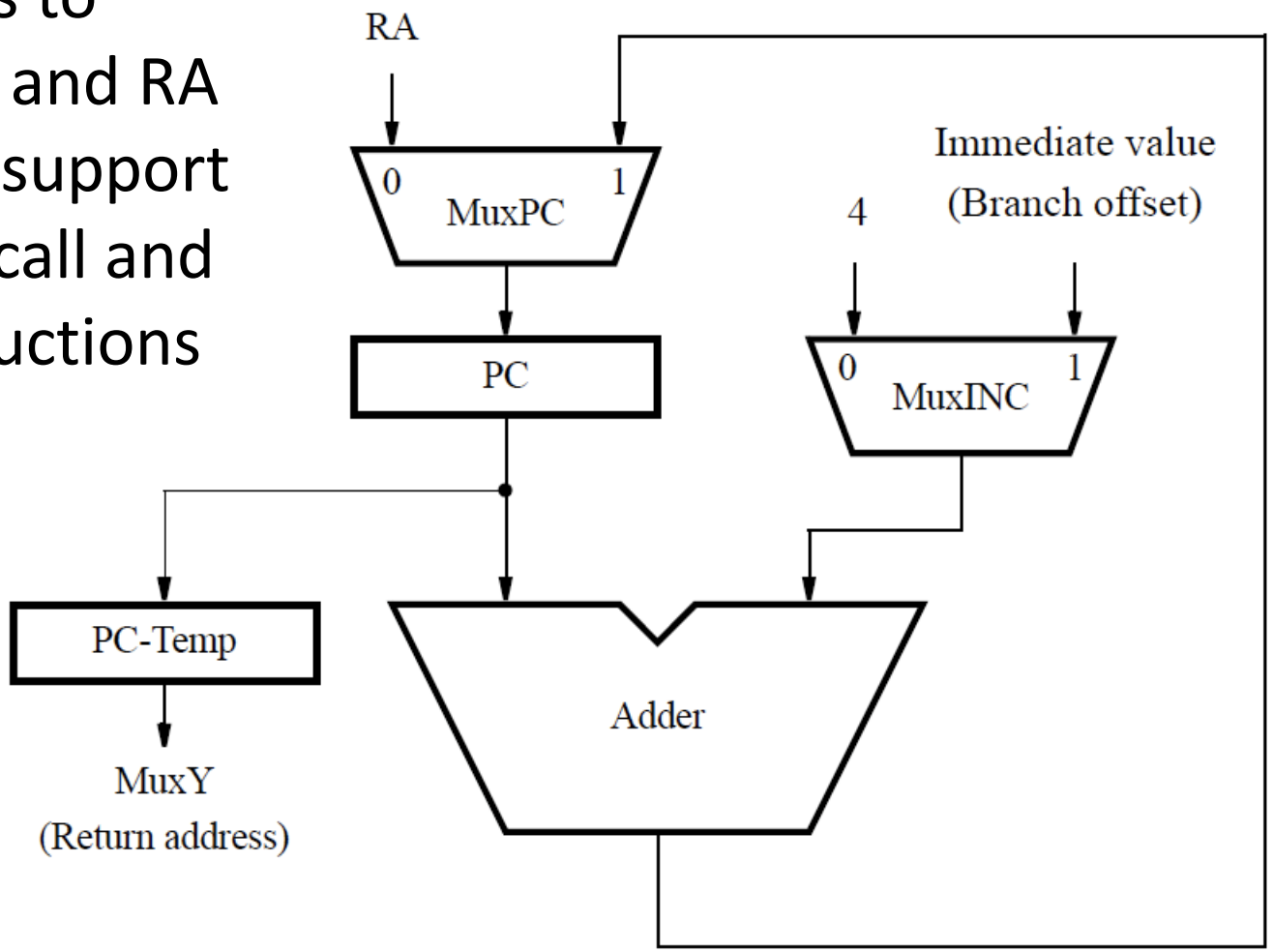
Memory stage

- **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 (return address stored in the register file)



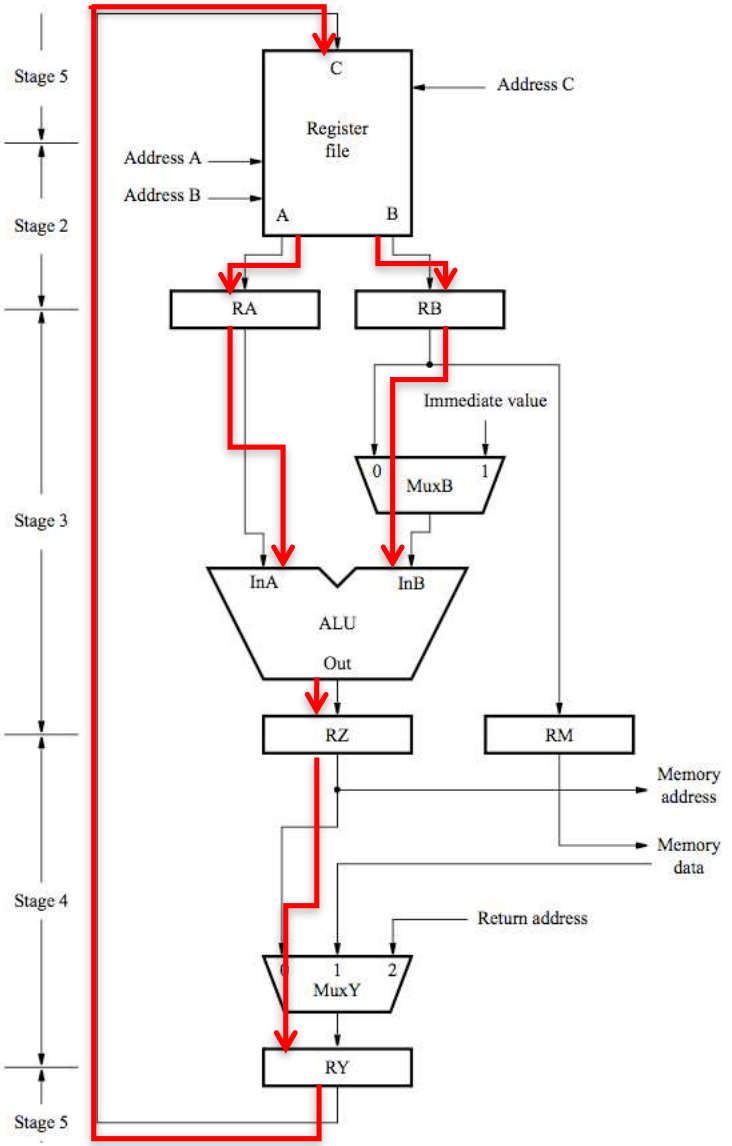
Instruction address generator

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



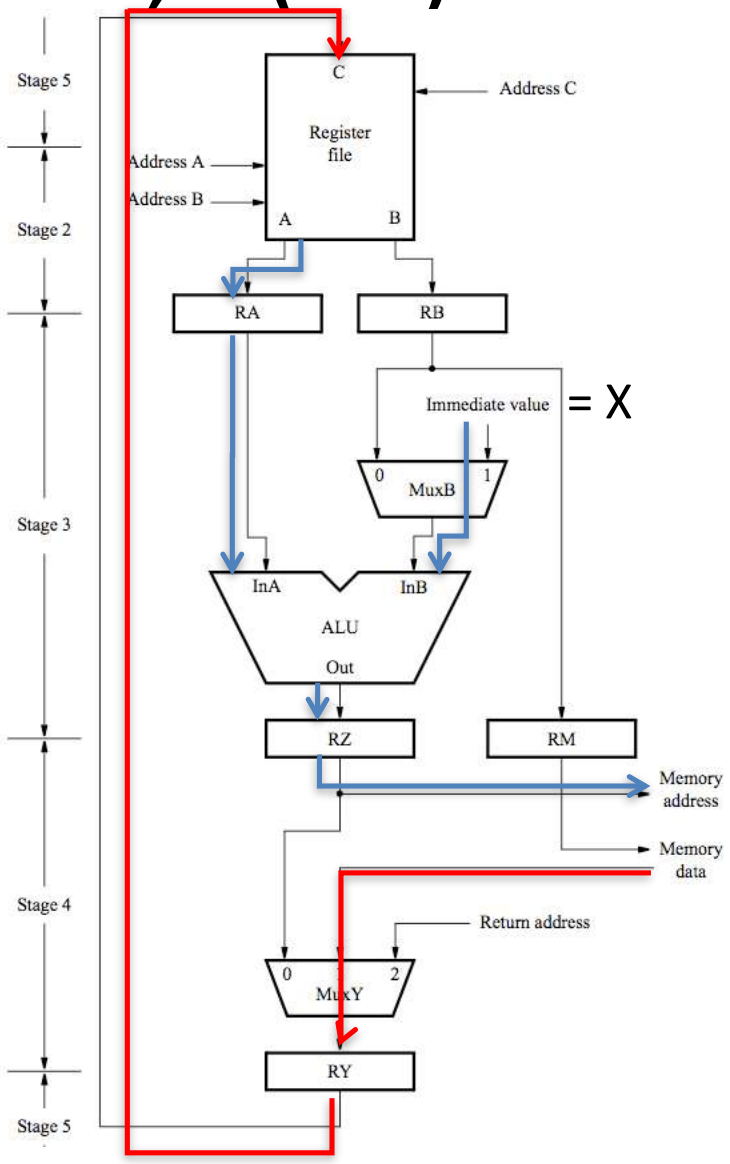
Example: add R3, R4, R5

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



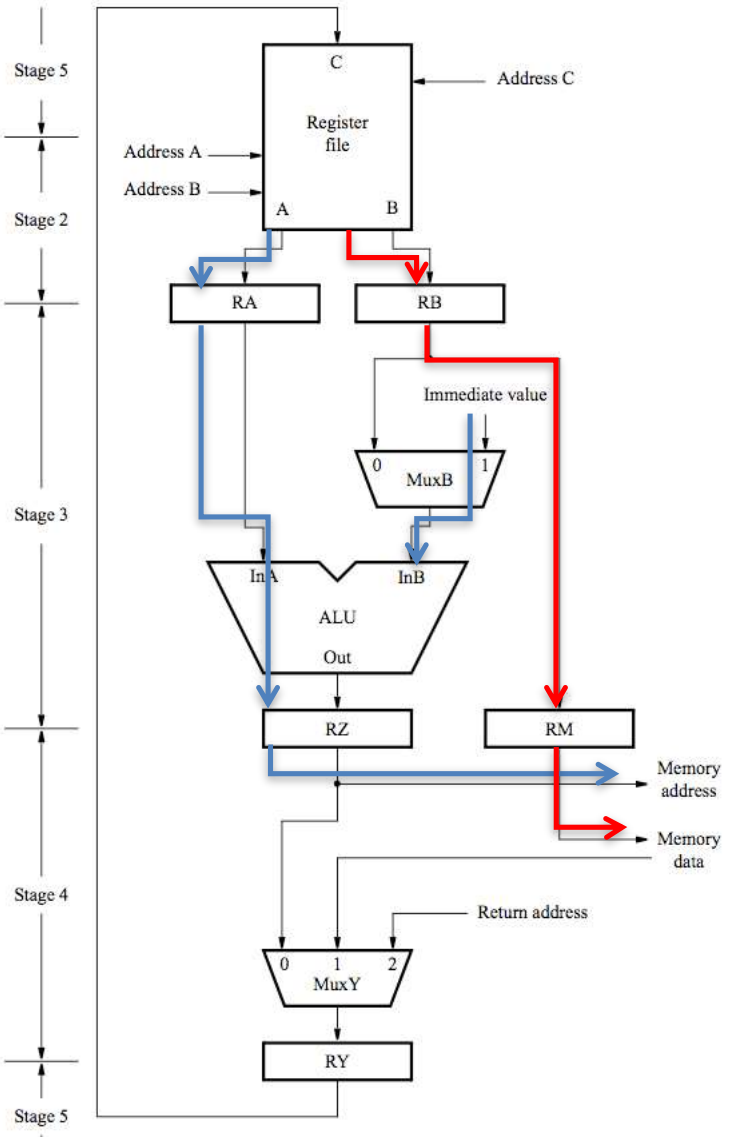
Example: ldw R5, X(R7)

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,
   RY ← Memory data
5. R5 ← [RY]



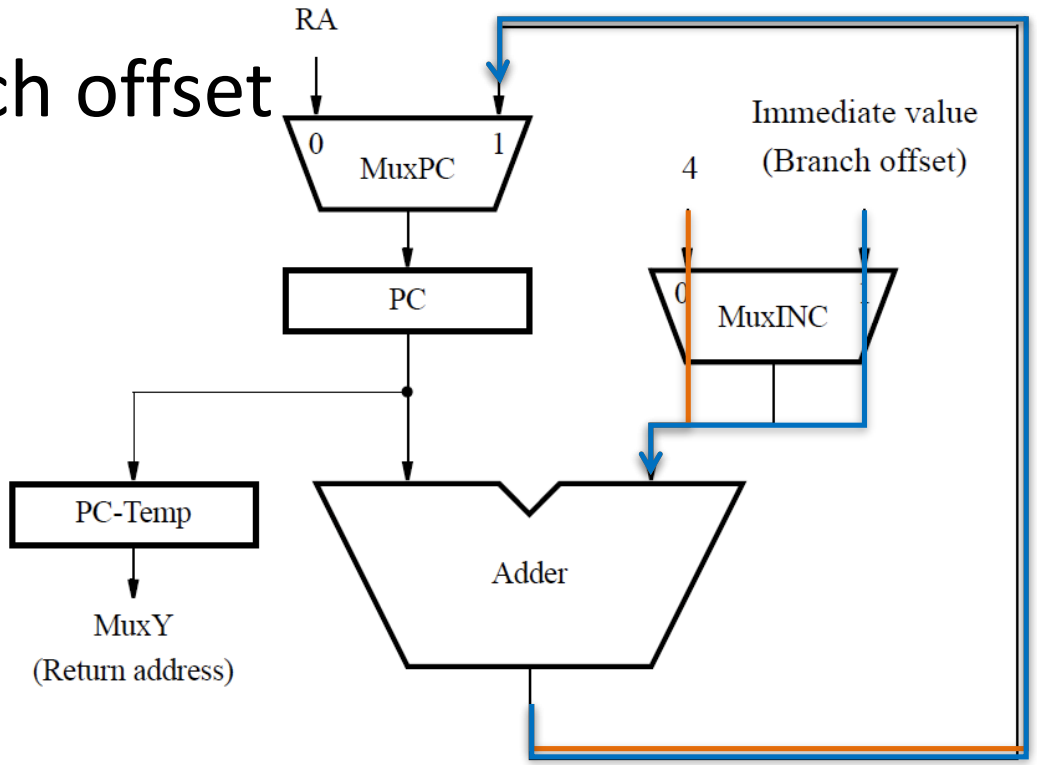
Example: stw R6, X(R8)

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
5. No action



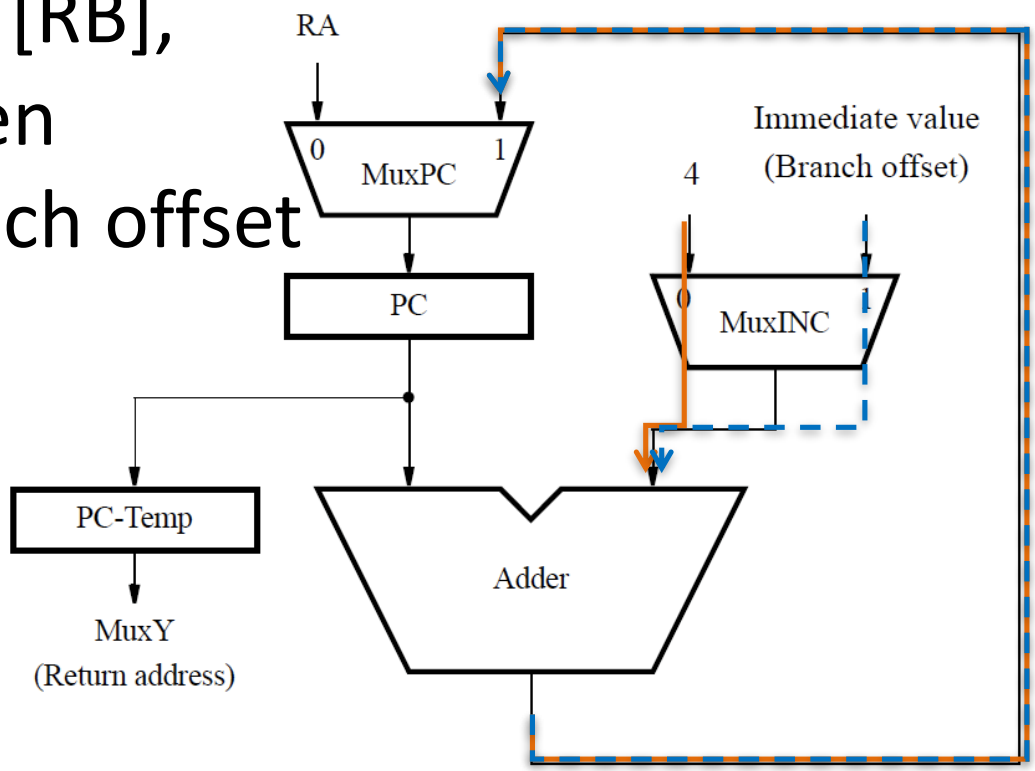
Unconditional branch

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



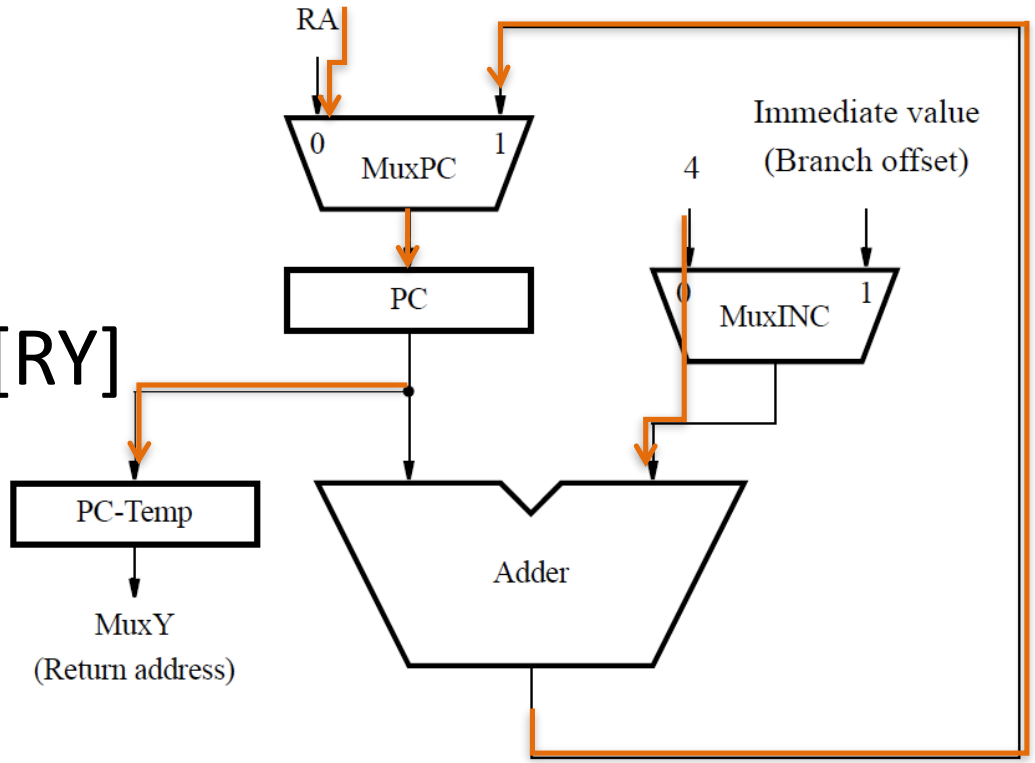
Conditional branch: Branch_if_[R5]=[R6] LOOP

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



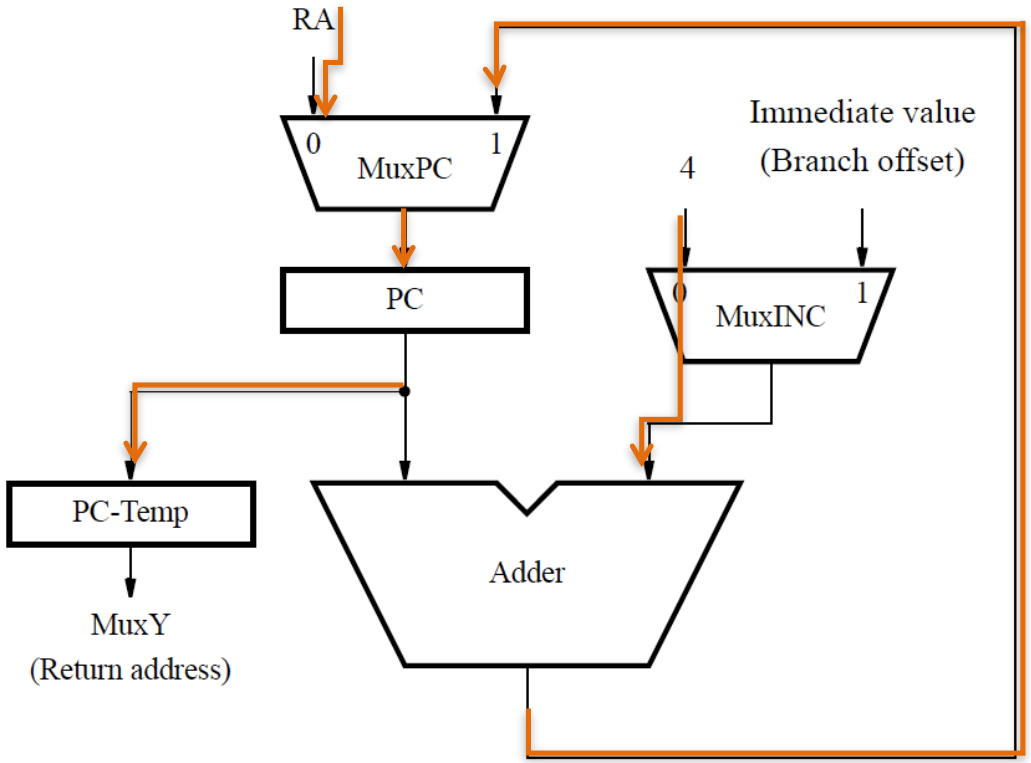
Subroutine call with indirection: Call_register R9

1. Memory address ← [PC], Read memory,
   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 RETurn

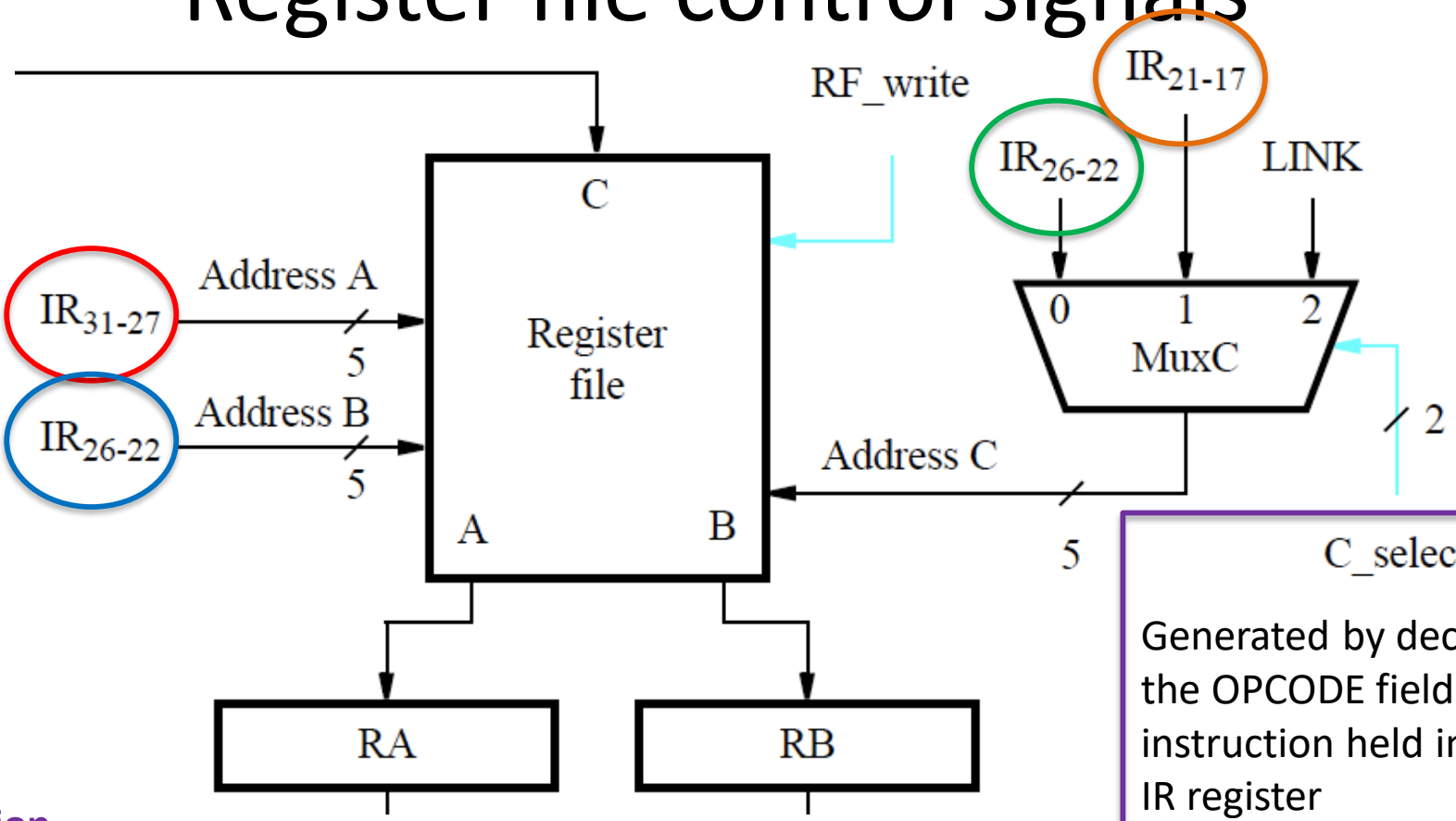
1. Memory address ← [PC], Read memory,
   IR ← Memory data, PC ←[PC] + 4
2. Decode instruction, RA ← Register LINK
3. PC ← [RA]
4. No action
5. No action



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

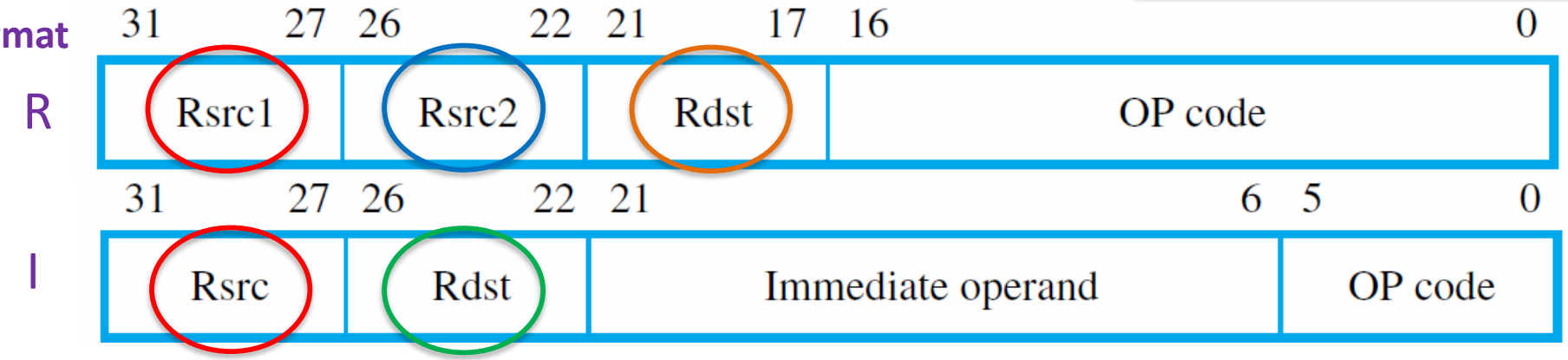
Register file control signals



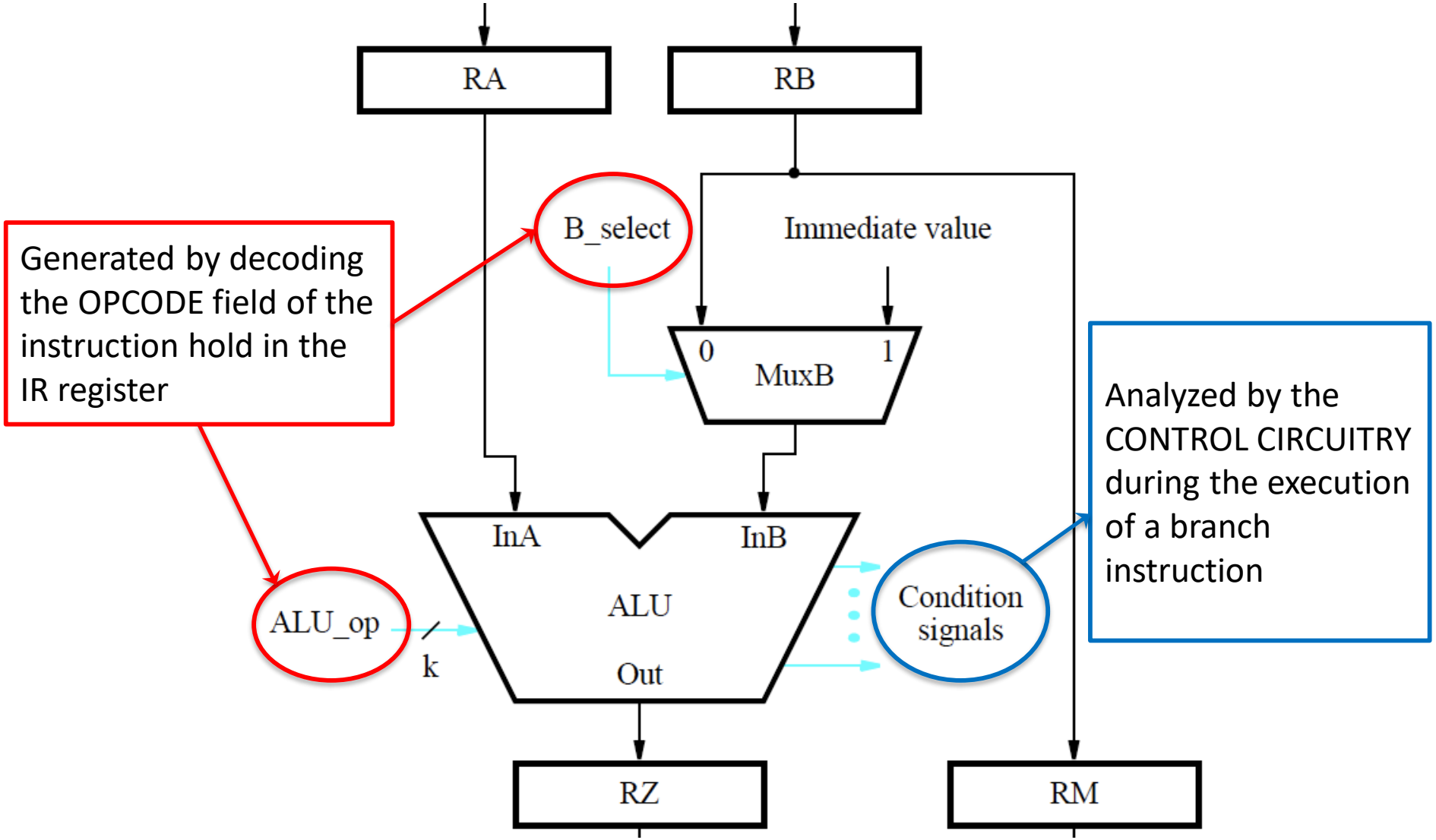
C_select

Generated by decoding the OPCODE field of the instruction held in the IR register

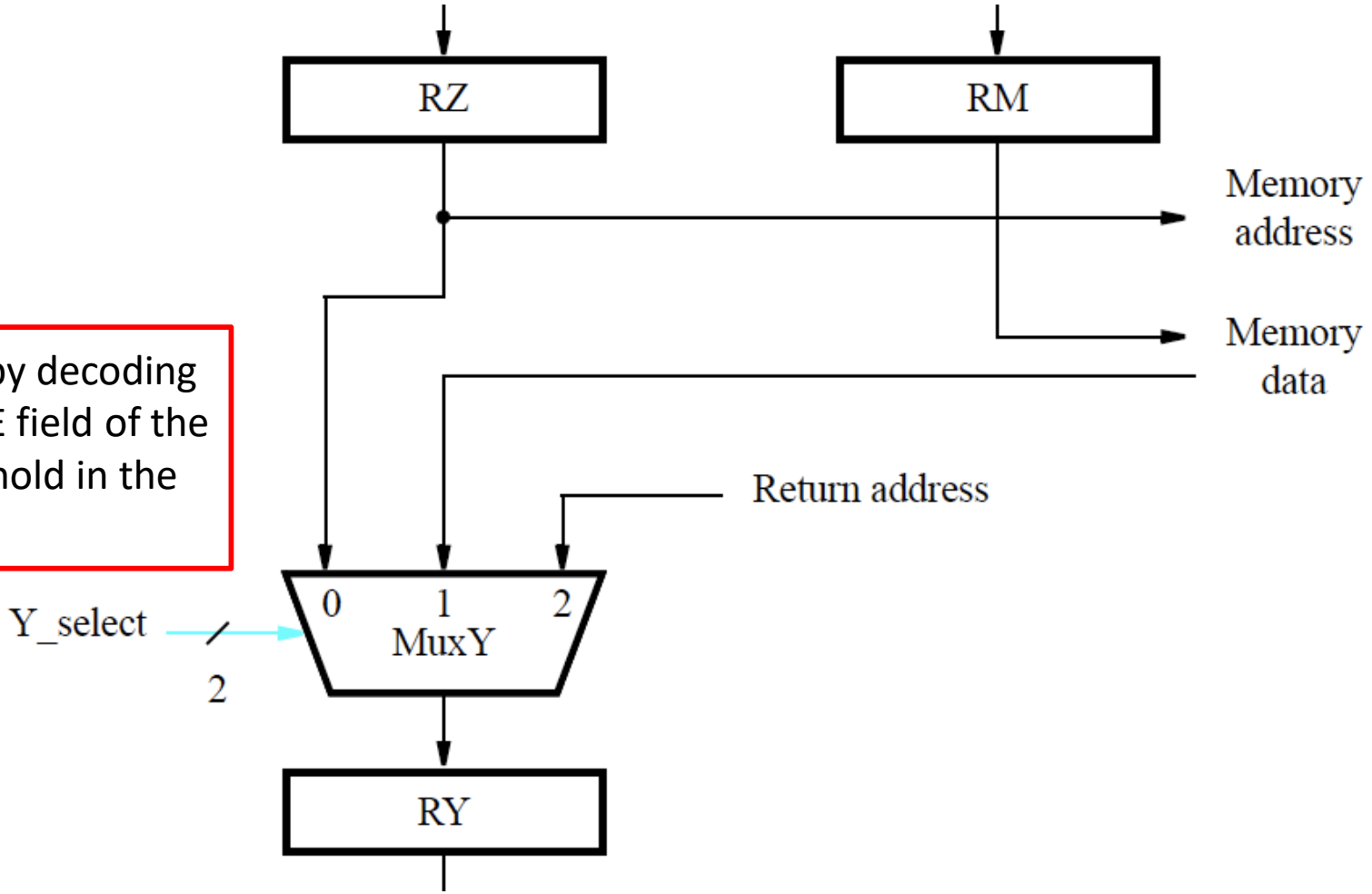
Instruction Format



ALU control signals



Result selection



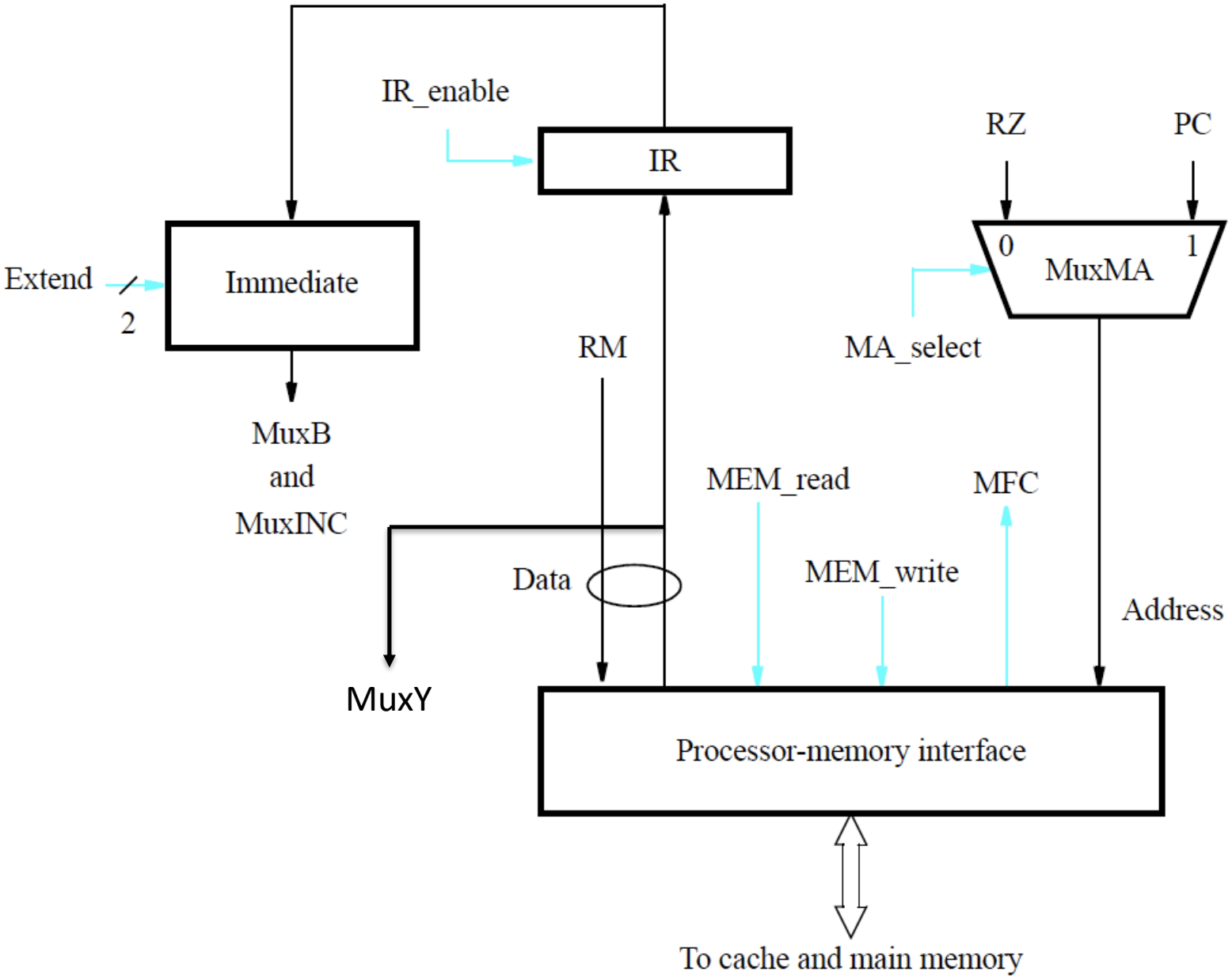
Generated by decoding the OPCODE field of the instruction hold in the IR register

Memory access

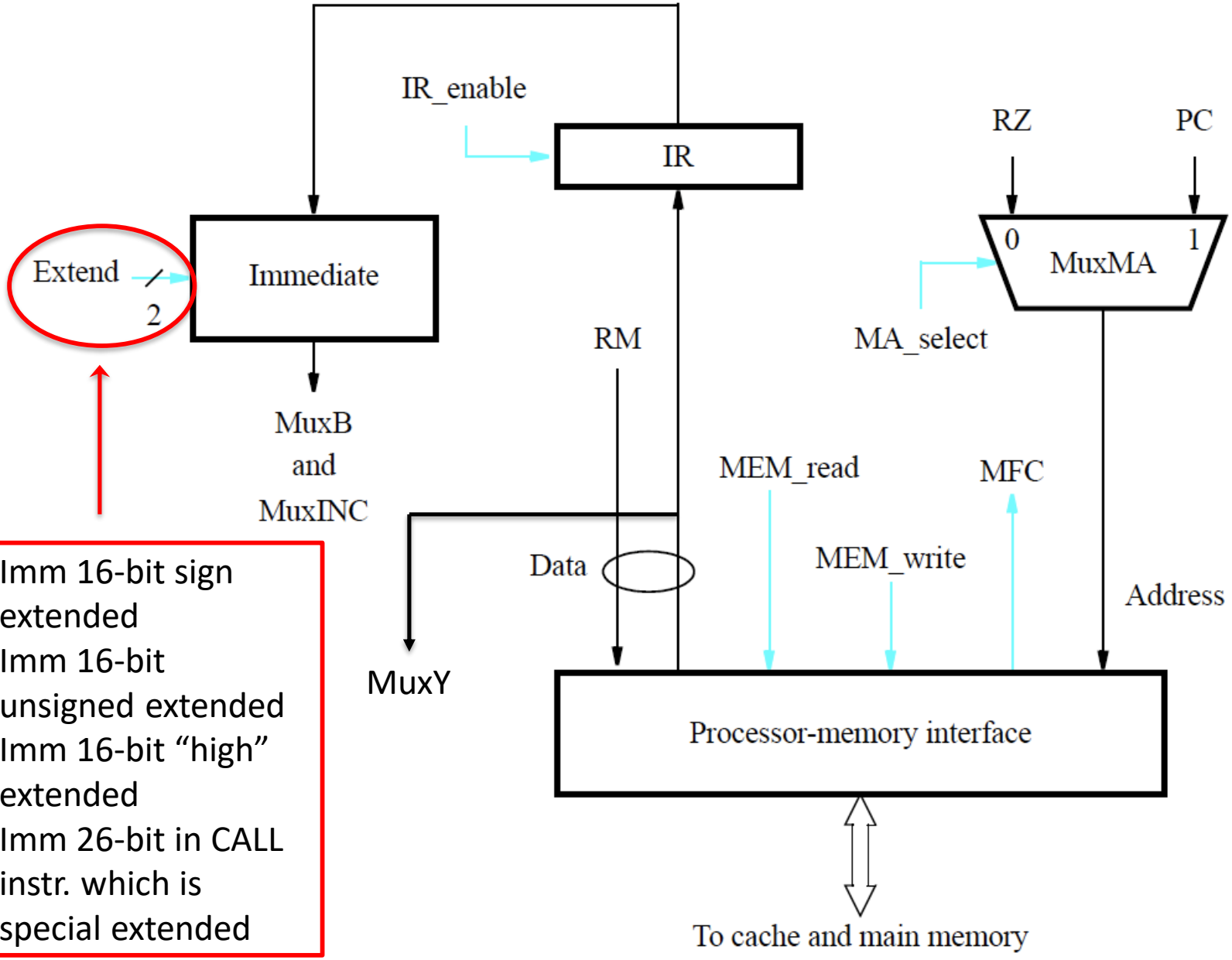
- When data are found in the cache, access to memory can be completed in one clock cycle.
- Otherwise, read and write operations may require several clock cycles to load data from main memory into the cache.
- A control signal is needed to indicate that memory function has been completed (**MFC**). E.g.,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

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

Memory and IR control signals

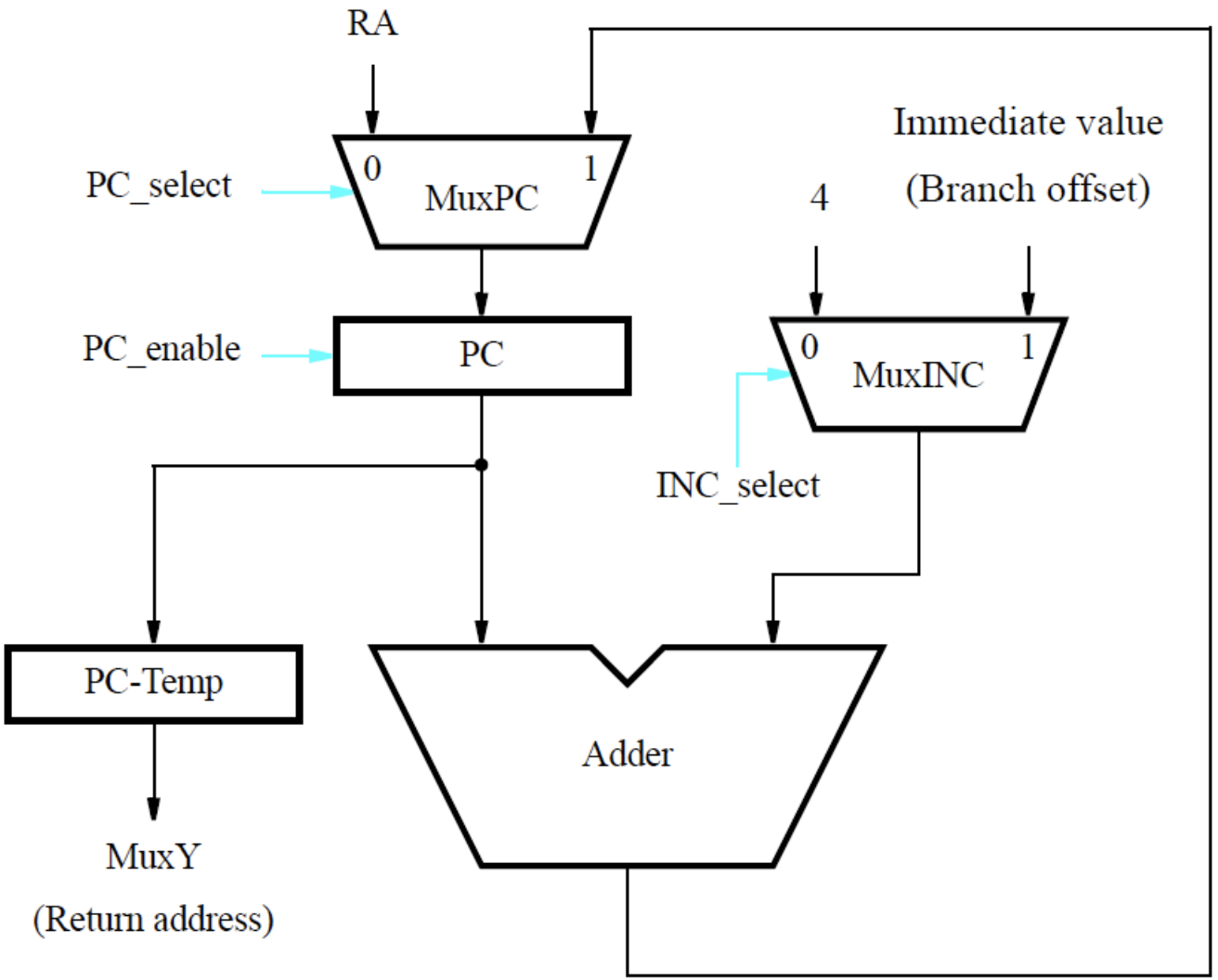


Memory and IR control signals



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

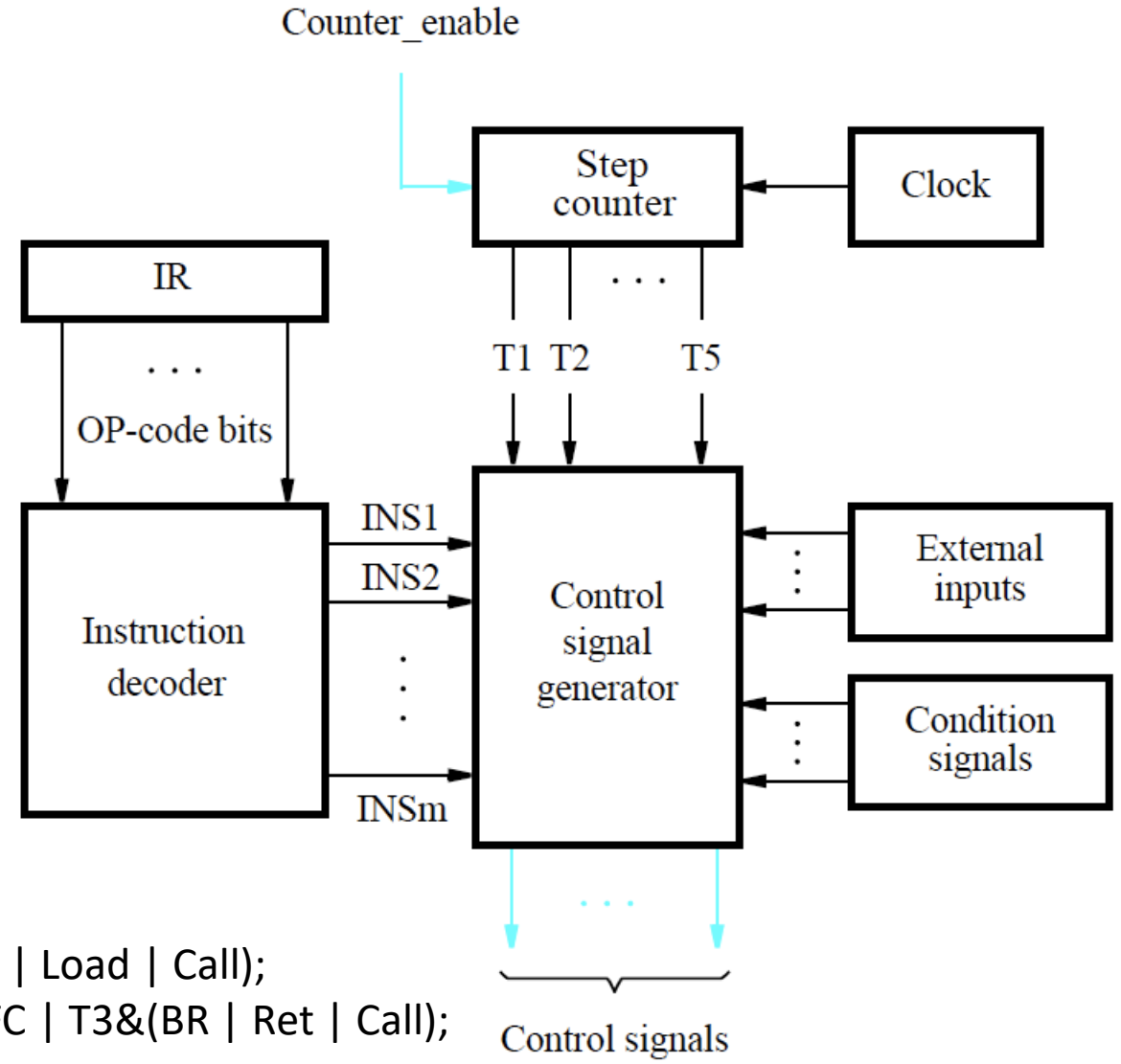
Control signals of instruction address generator



Control signal generation

- 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).

Hardwired generation of control signals



E.g.

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

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

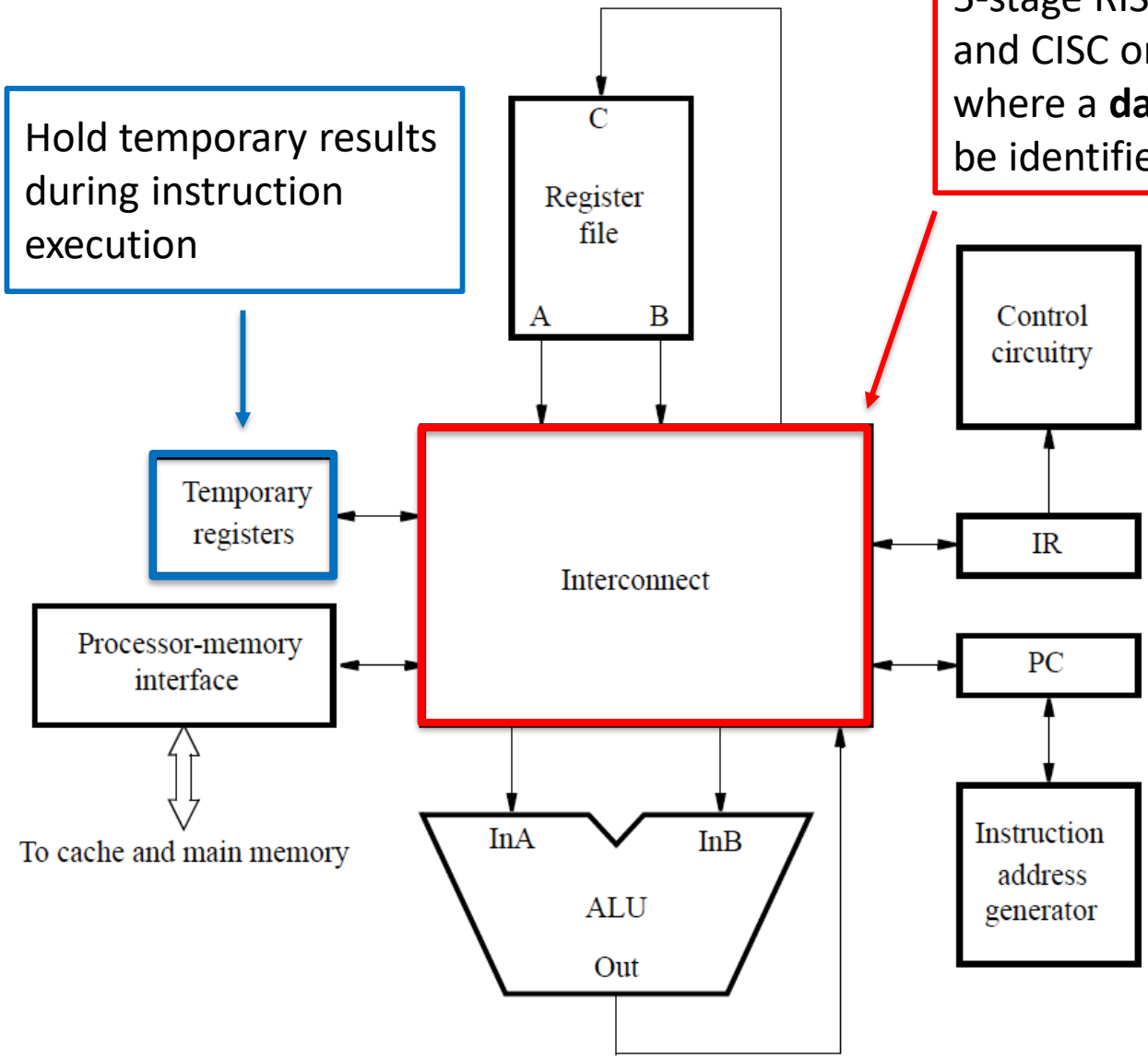
CISC processors

- CISC-style processors have more complex instructions.
- 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

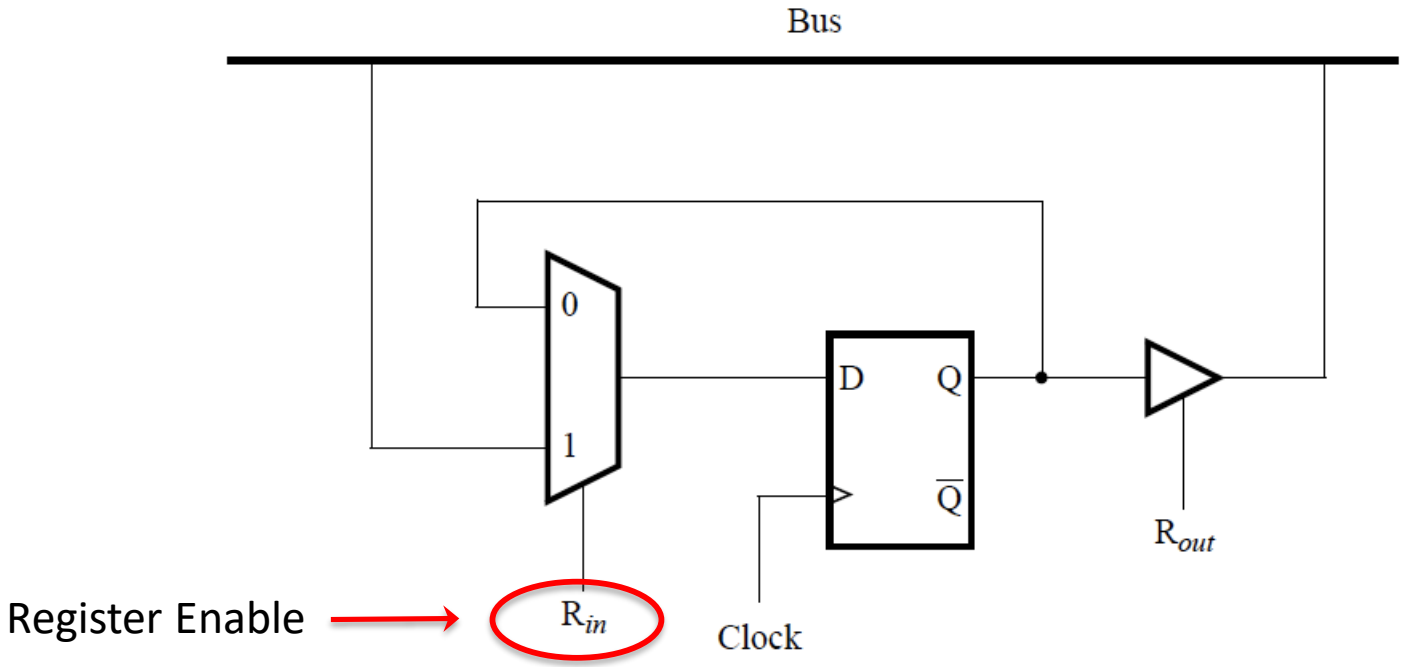
Main difference between 5-stage RISC organization and CISC organization, where a **datapath** cannot be identified easily

Hold temporary results during instruction execution

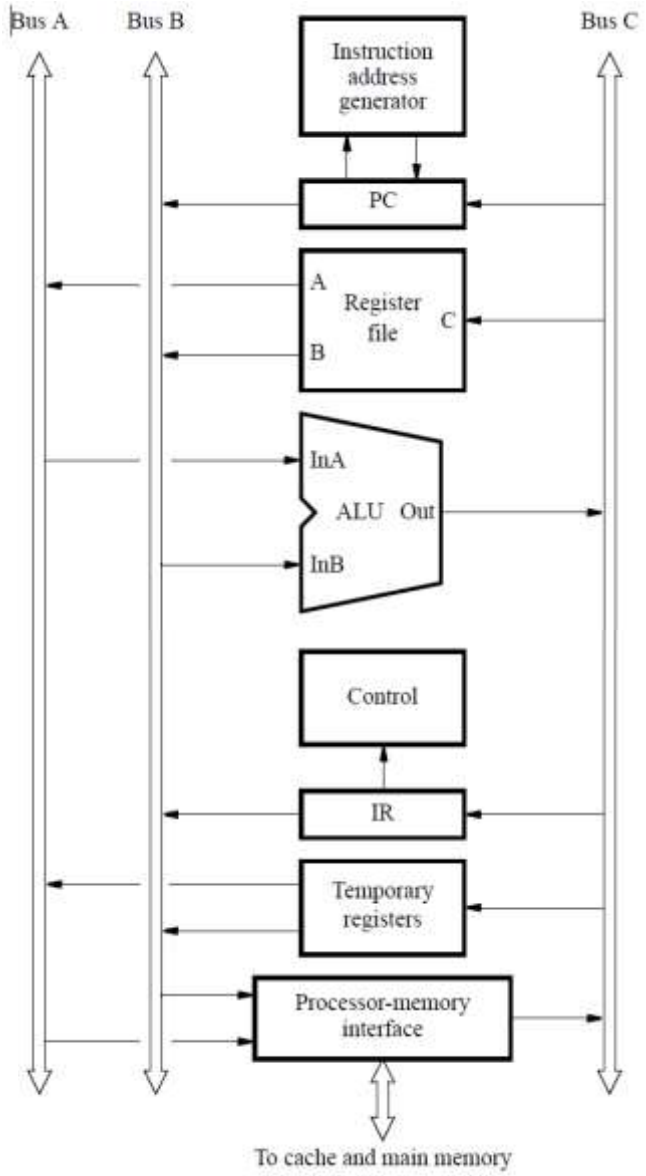


Bus

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



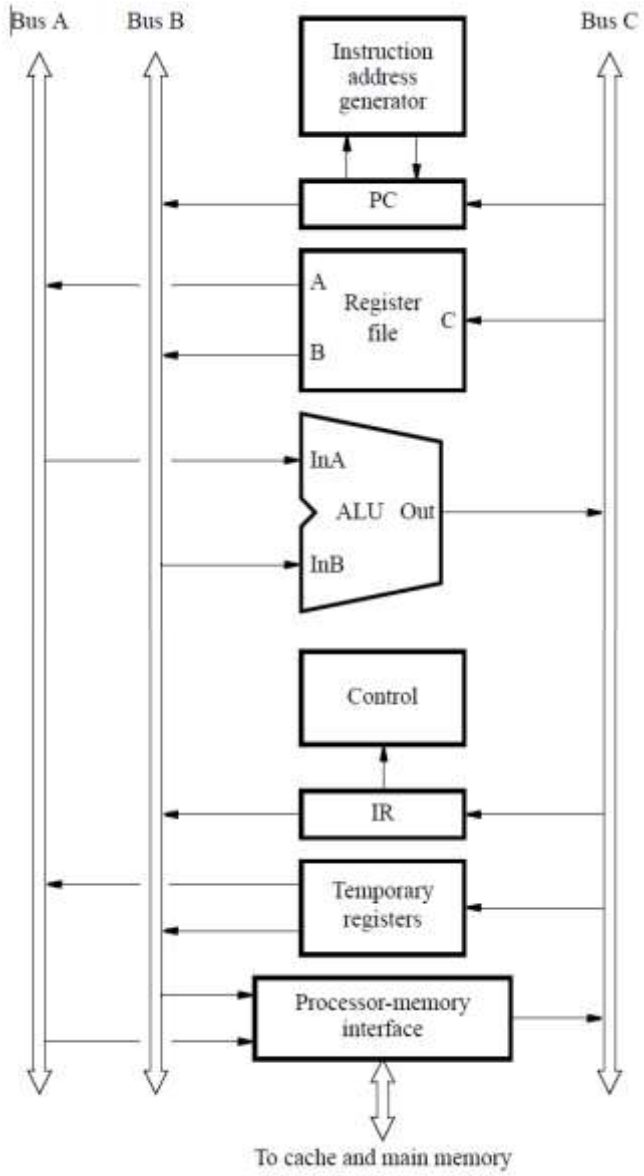
A 3-bus interconnection network



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