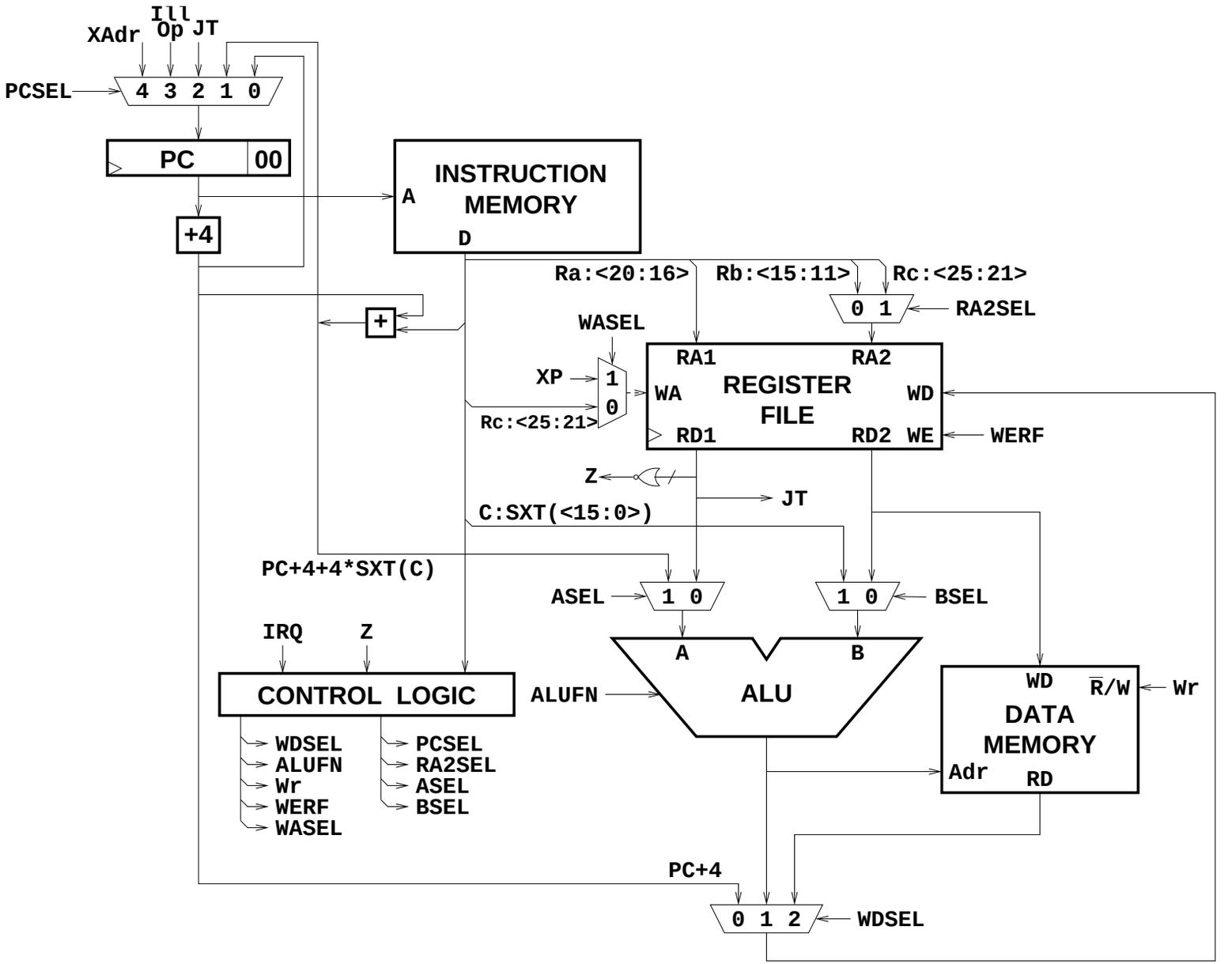
The "pipeline" technique and the β machine

Review : a "one cycle" implementation of β



The basic idea of a pipeline

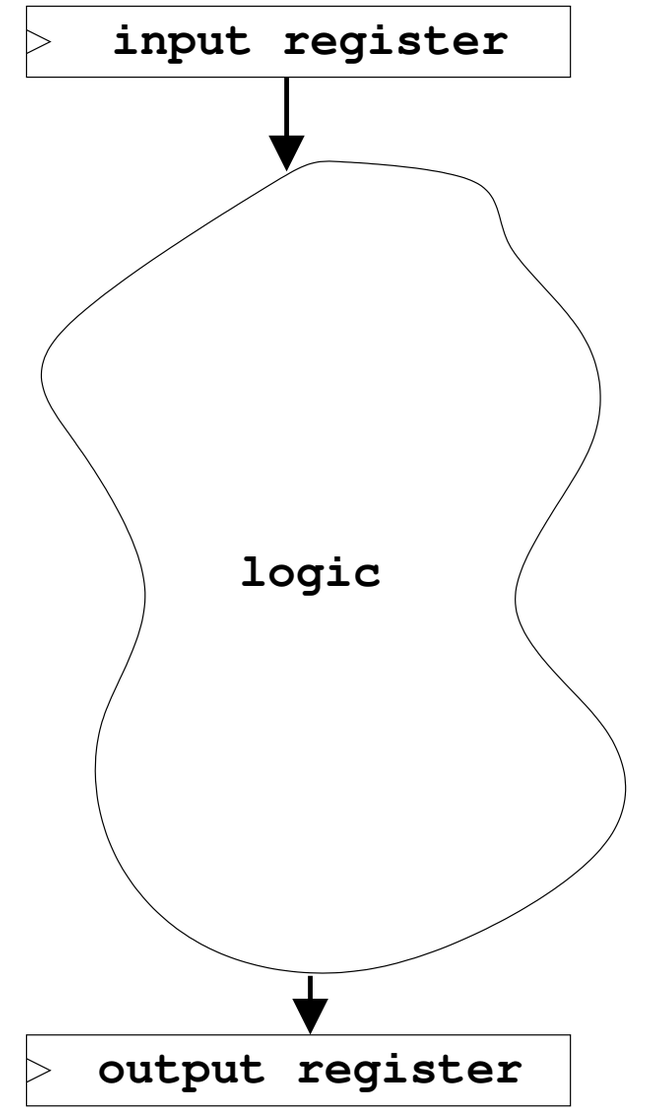
- A pipeline is useful to increase the rate at which a deep combinatorial circuit can process data.

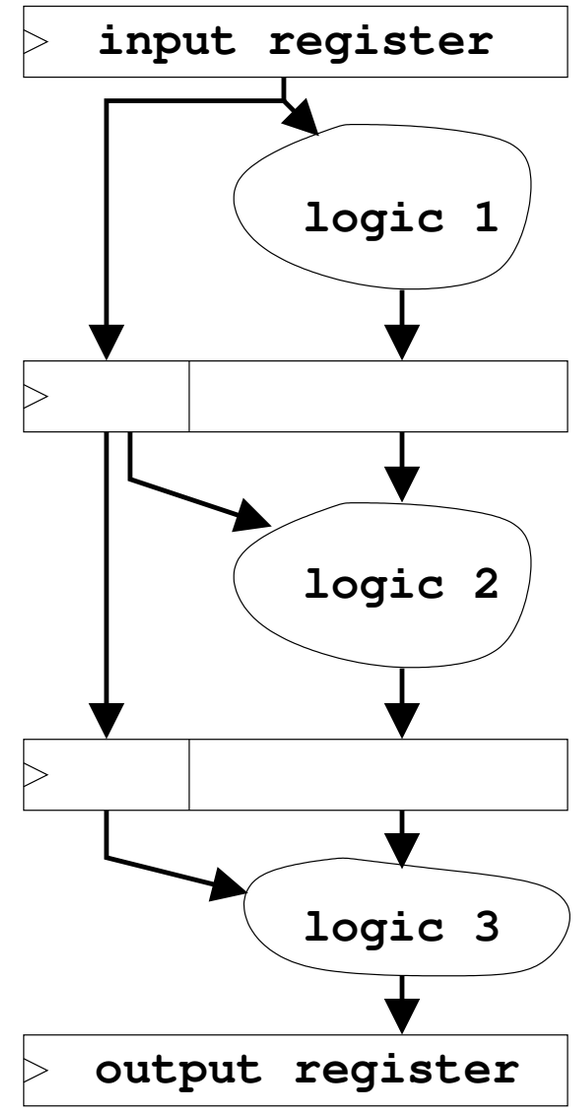
- Setting up a pipeline is done is done by splitting the circuit into layers (or *stages*) and separating these layers with registers.

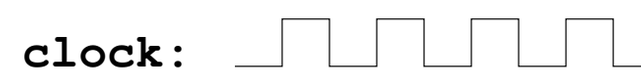
- The data is saved in the registers between successive layers. Several cycles are needed to process the data, but each cycle can be substantially shorter.

- The transit time of the data through the circuit is unchanged or slightly increased, but the rate at which data can be provided to the circuit is significantly increased.

The basic idea of a pipeline: illustration



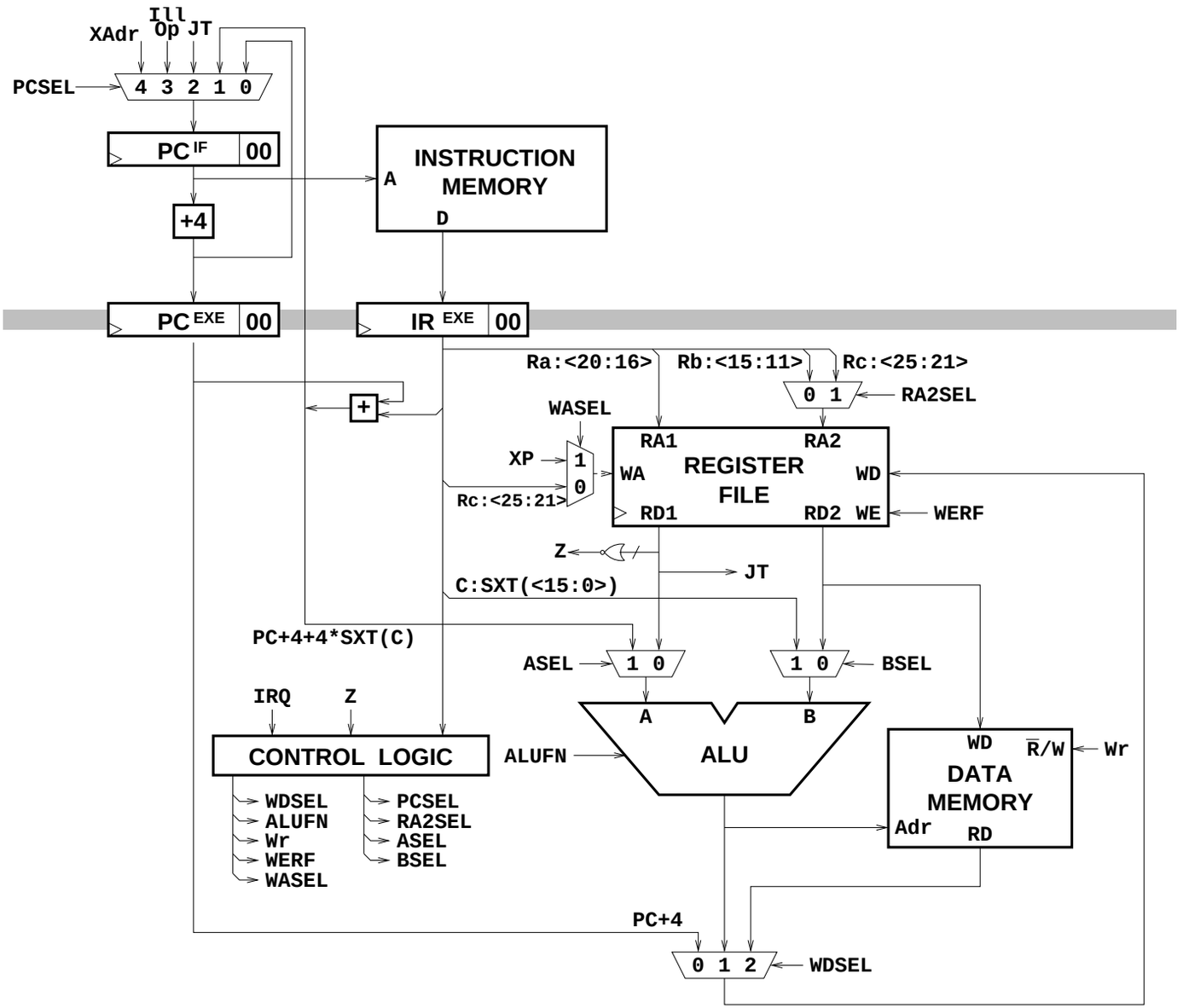




A pipeline for the β machine

- In the implementation of the β machine under consideration, the combinatorial logic is composed of four clearly identifiable *stages*.
  - Reading the instruction from memory: *Instruction Fetch* (IF);
  - Reading the arguments in the registers: *Register Fetch* (RF);
  - Computation by the ALU: ALU;
  - Reading data from memory (instruction `LD`): *Memory* (MEM);
  - and ends with the result being written back to the registers: *Write Back* (WB).
- A pipeline with four stages would thus be a natural choice. However, to simplify the presentation, we will start with a 2-stage pipeline: IF et EXE (*execute*).

A β machine with a 2-stage pipeline



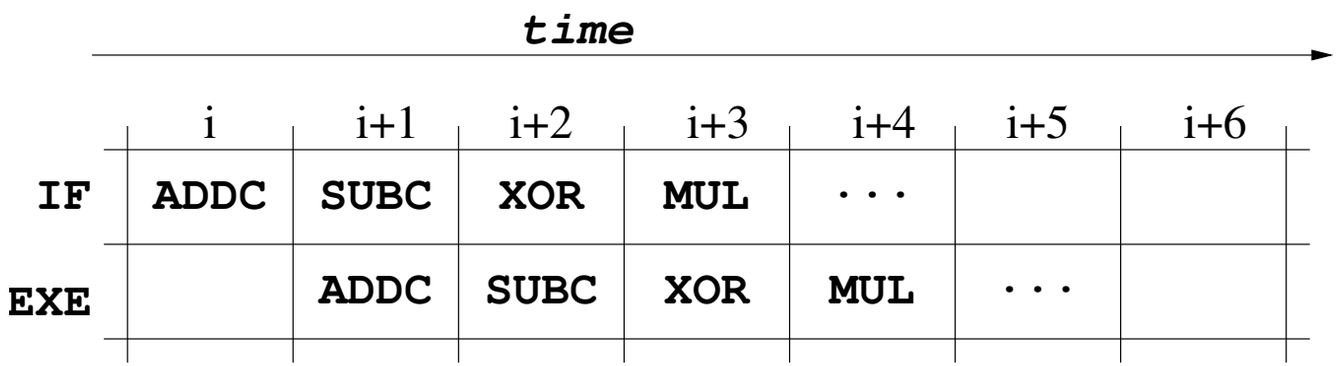
The operation of the 2-stage pipeline

At all times, two instructions are being executed: one at the IF stage, one at the EXE stage.

Consider, for example, the following program:

```
ADDC(r1, 1, r2)
SUBC(r1, 1, r3)
XOR(r1, r5, r1)
MUL(r2, r6, r0)
```

The state of the pipeline evolves as described below.



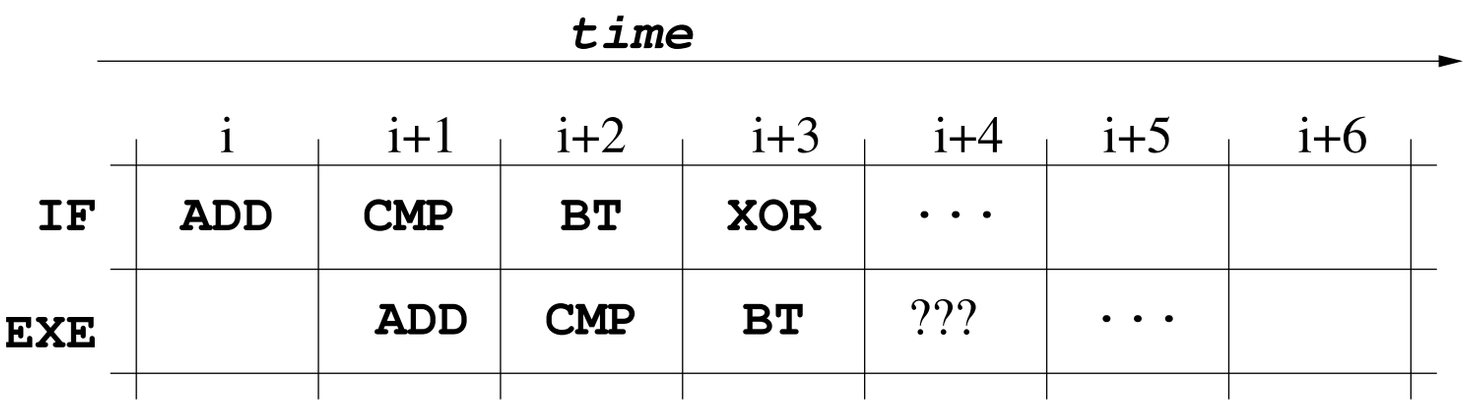
The operation of the 2-stage pipeline: The problem with jumps

When a jump or branch (`JMP or BXX`) instruction appears, a problem occurs since the next instruction has already been read.

Let's consider for example:

```
LOOP: ADD(r1, r3, r3)
      CMPLEC(r3, 100, r0)
      BT(r0, LOOP)
      XOR(r3, -1, r3)
```

If the branch has to be executed, the pipeline is in an incoherent state.



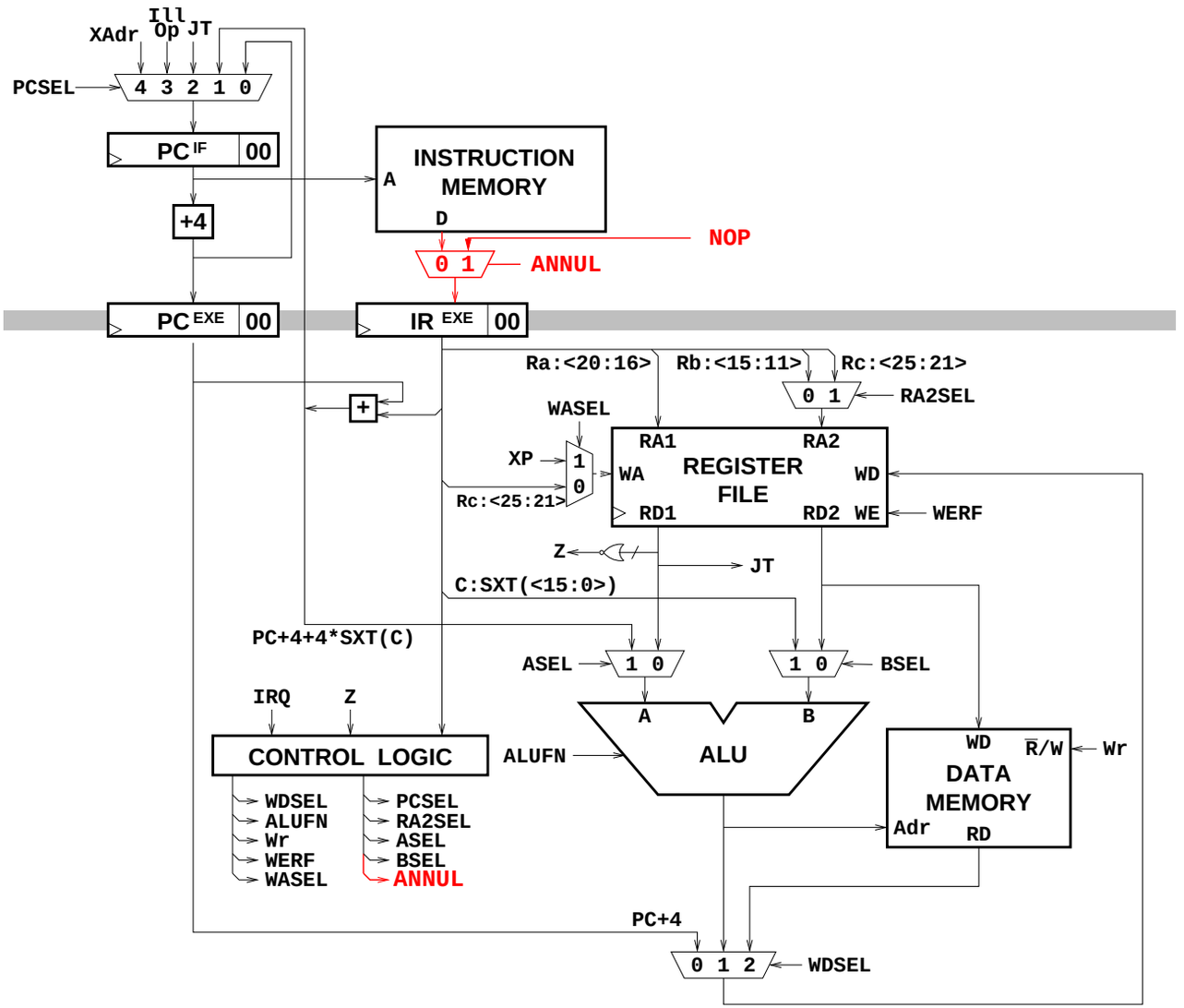
The operation of the 2-stage pipeline: Solutions to the branch/jump problem

The are two types of solutions for this problem.I

- **Hardware solutions**: the hardware "cancels" the instruction following jumps that are followed.

- **Software solutions** : a branch/jump is always followed by an instruction that does nothing (`NOP`), or by a non problematic useful instruction.

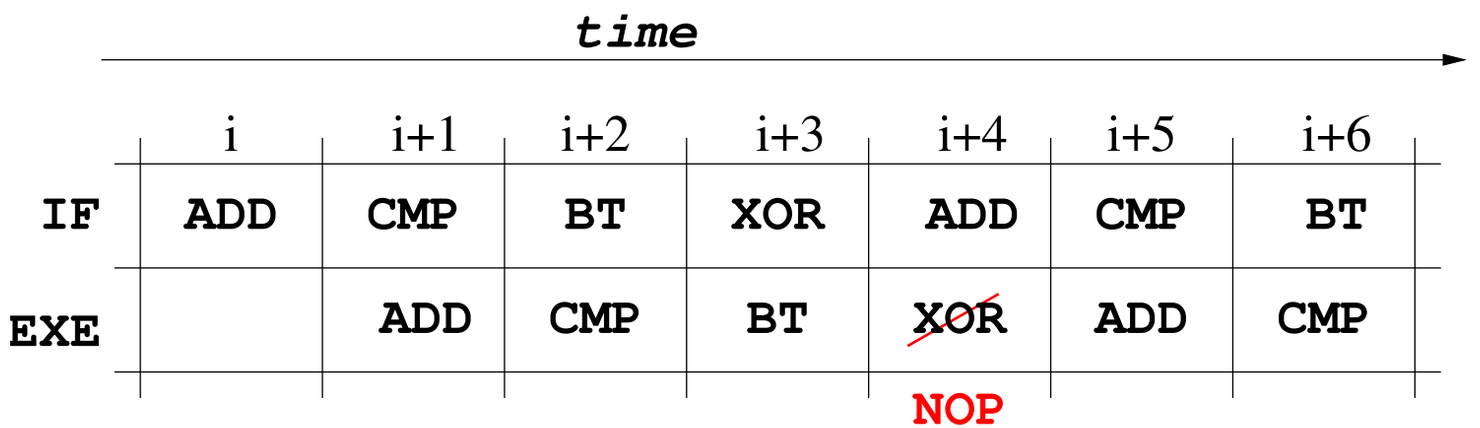
A β machine with a 2-stage pipeline
A hardware solution to the jump/branch problem



A β machine with a 2-stage pipeline
A hardware solution to the jump/branch problem – example

For the following program
```
LOOP: ADD(r1, r3, r3)
      CMPLEC(r3, 100, r0)
      BT(r0, LOOP)
      XOR(r3, -1, r3)
```

If the branch is executed, the pipeline behaves as follows.

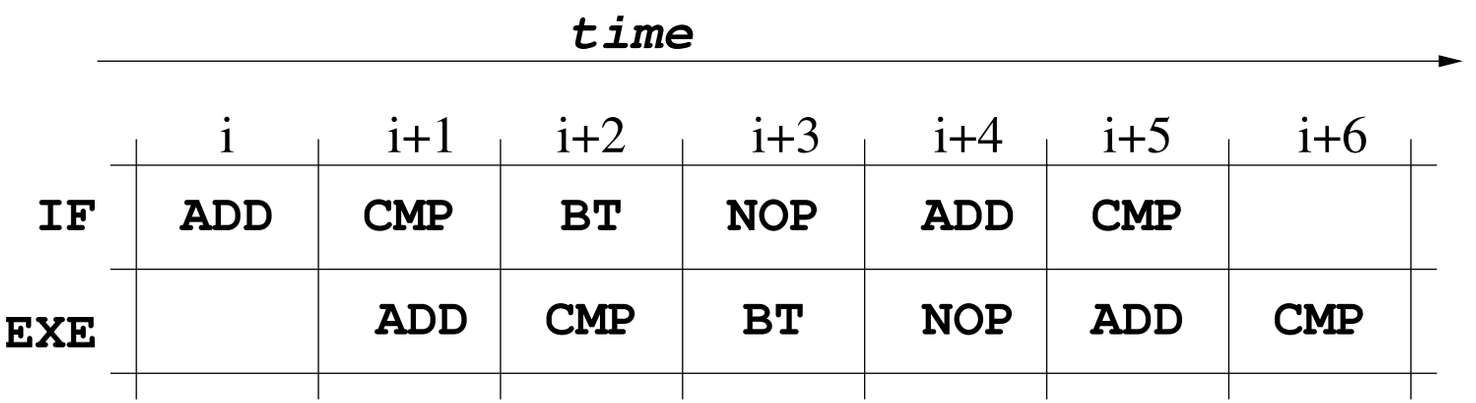


A β machine with a 2-stage pipeline
A first software solution – example

A `NOP` instruction is added added after the branch.

```
LOOP: ADD(r1, r3, r3)
      CMPLEC(r3, 100, r0)
      BT(r0, LOOP)
      NOP()
      XOR(r3, -1, r3)
```

The `NOP` instruction is always executed after the branch but has no effect.



A β machine with a 2-stage pipeline
A second software solution – example

The preceeding program is rewritten as follows.

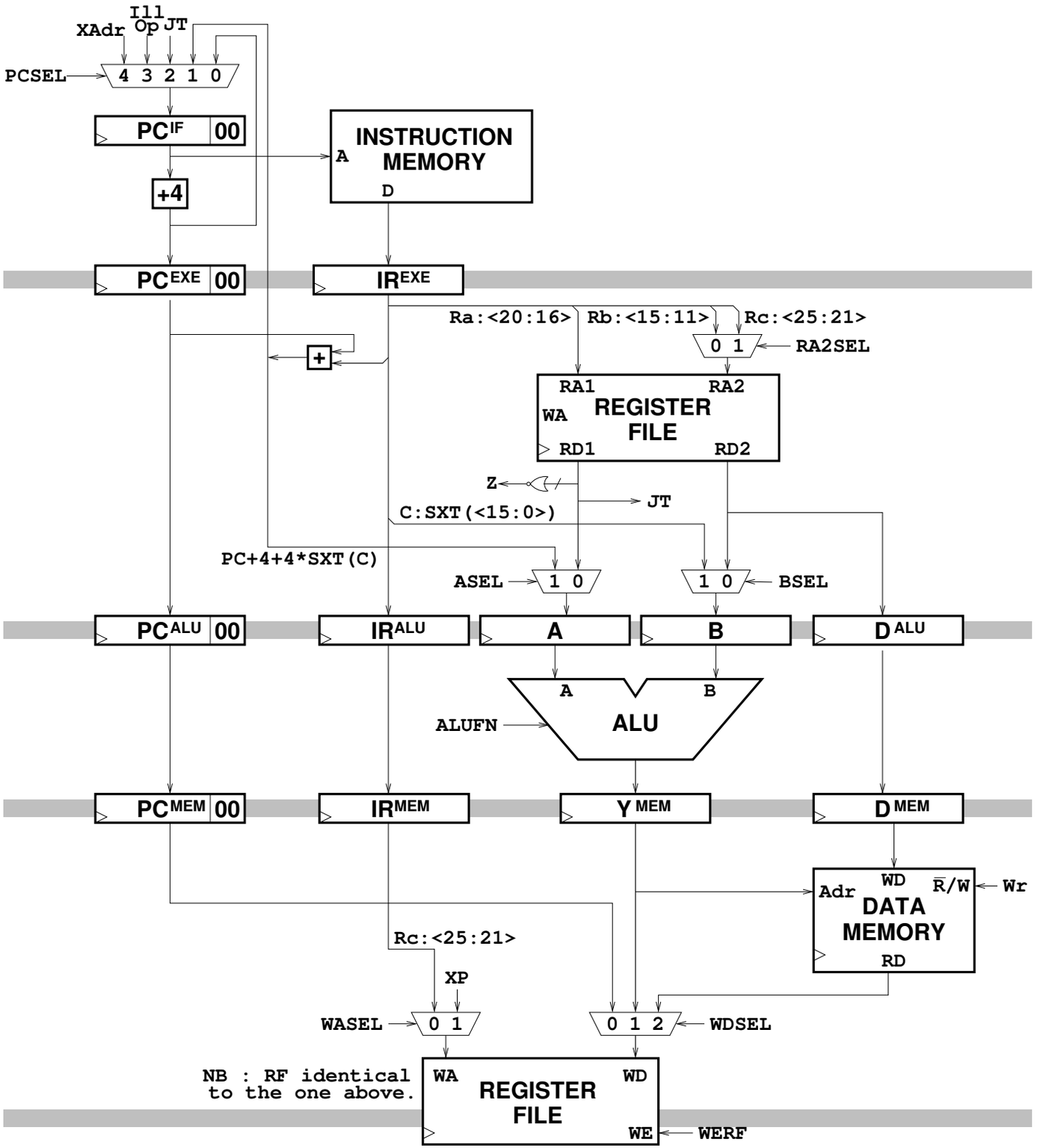
```
LOOP:  ADD(r1, r3, r3)
LOOPx: CMPLEC(r3, 100, r0)
       BT(r0, LOOPx)
       ADD(r1, r3, r3)
       SUB(r3, r1, r3)
       XOR(r3, -1, r3)
```

The instruction inserted after the `BT` is an instruction that must anyway be executed after the branch is taken (the majority of cases) and the effect of this instruction is compensated for if the branch is not followed.

Note that this program is not correct on a machine without a pipeline.

A β machine with a 4-stage pipeline

- The following figure shows an outline of an implementation of the β machine with a 4-stage pipeline.
- The stages are: IF, RF, ALU, and MEM/WB.
- The registers appear as a combinatorial circuit when read from, and as a clocked circuit when written to.
- The registers appearing at the bottom of the figure are identical to those appearing at the top.


Ill
XAdr
Op
JT
PCSEL
4 3 2 1 0
PC IF
00
+4
INSTRUCTION MEMORY
A
D
PC EXE
00
IR EXE
Ra:<20:16>
Rb:<15:11>
Rc:<25:21>
0 1
RA2SEL
+
RA1
RA2
WA
REGISTER FILE
RD1
RD2
Z
JT
C:SXT(<15:0>)
PC+4+4*SXT(C)
ASEL
1 0
1 0
BSEL
PC ALU
00
IR ALU
A
B
D ALU
A
B
ALUFN
ALU
PC MEM
00
IR MEM
Y MEM
D MEM
Adr
WD
R/W
Wr
DATA MEMORY
RD
Rc:<25:21>
XP
WASEL
0 1
0 1 2
WDSEL
NB : RF identical to the one above.
WA
WD
REGISTER FILE
WE
WERF

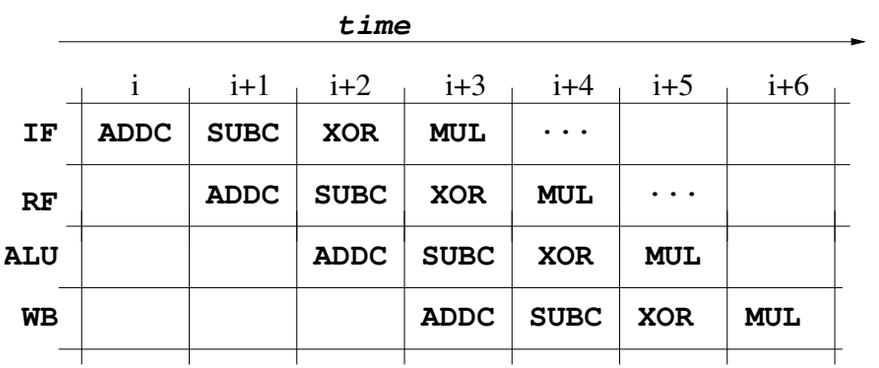
The operation of the 4-stage pipeline

At any time, there are four instructions being executed.

Consider for example the following program:

```
ADDC(r1, 1, r2)
SUBC(r1, 1, r3)
XOR(r1, r5, r1)
MUL(r2, r6, r0)
```

The state of the pipeline evolves as described below.

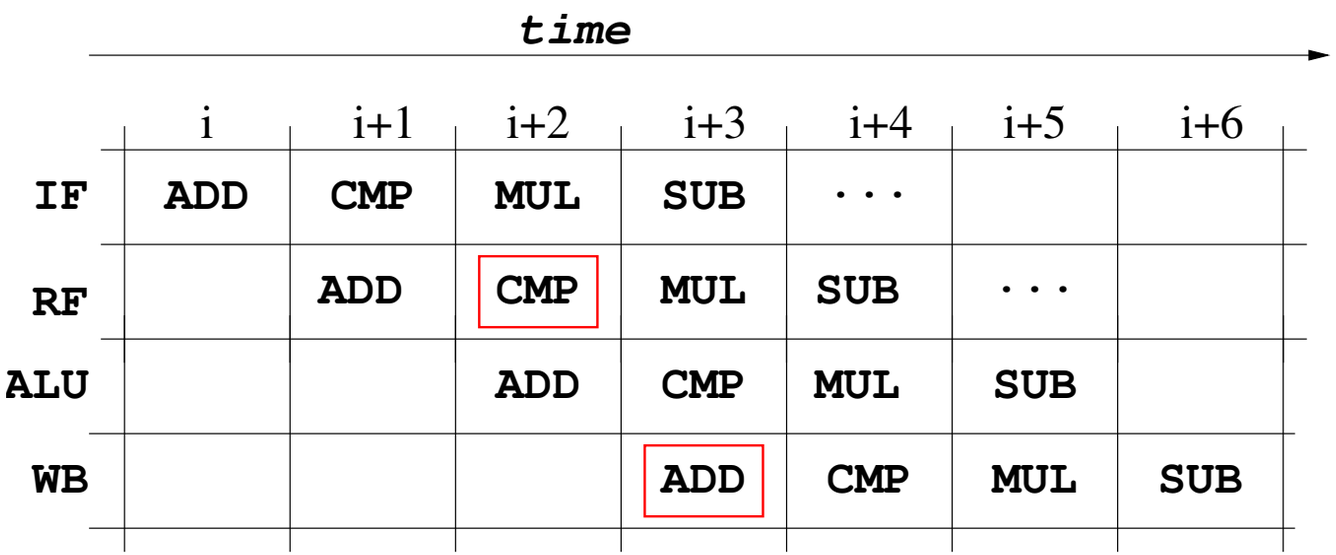


The operation of the 4-stage pipeline: the problem with data conflicts

Consider for example the following program:

```
ADD(r1, r2, r3)
CMPLEC(r3, 100, r0)
MULC(r1, 100, r4)
SUB(r1, r2, r5)
```

The instruction `CMPLEC` needs at time `i+2` the content of the register `r3` that will only be written at the end of cycle `i+3`.



The operation of the 4-stage pipeline: the problem with data conflicts – software solution

A first solution is to modify the program in order to eliminate data conflicts.

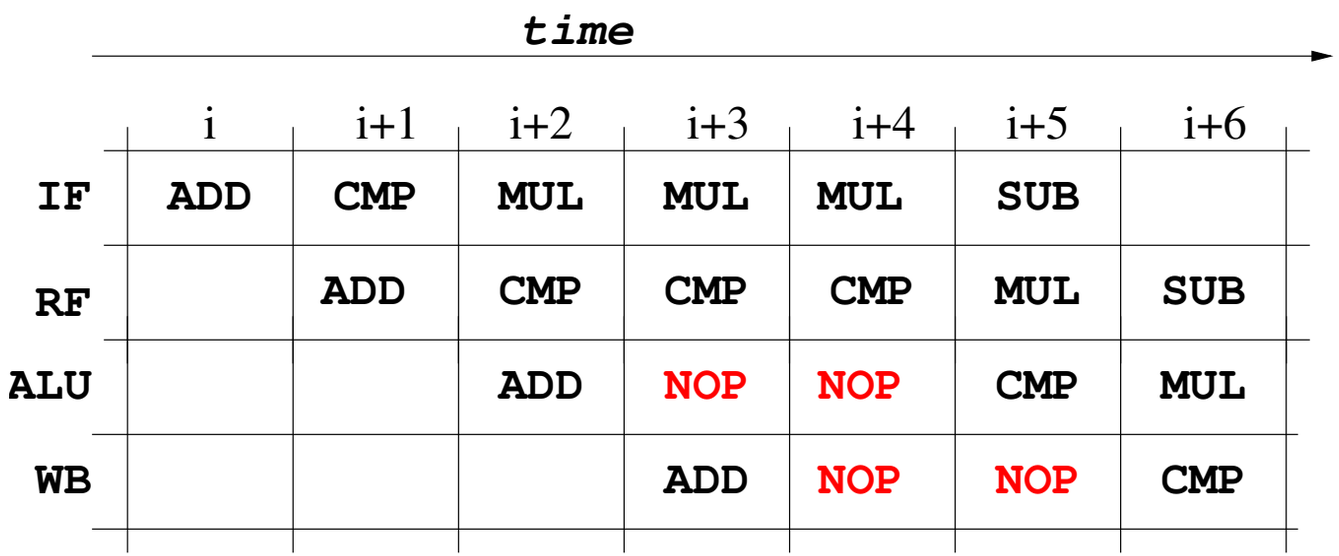
```
ADD(r1, r2, r3)
CMPLEC(r3, 100, r0)
MULC(r1, 100, r4)
SUB(r1, r2, r5)
```

becomes

```
ADD(r1, r2, r3)
MULC(r1, 100, r4)
SUB(r1, r2, r5)
CMPLEC(r3, 100, r0)
```

The operation of the 4-stage pipeline: the problem with data conflicts – hardware solution 1

A hardware solution is to block the pipeline (and insert `NOP` instructions) when there are data conflicts.

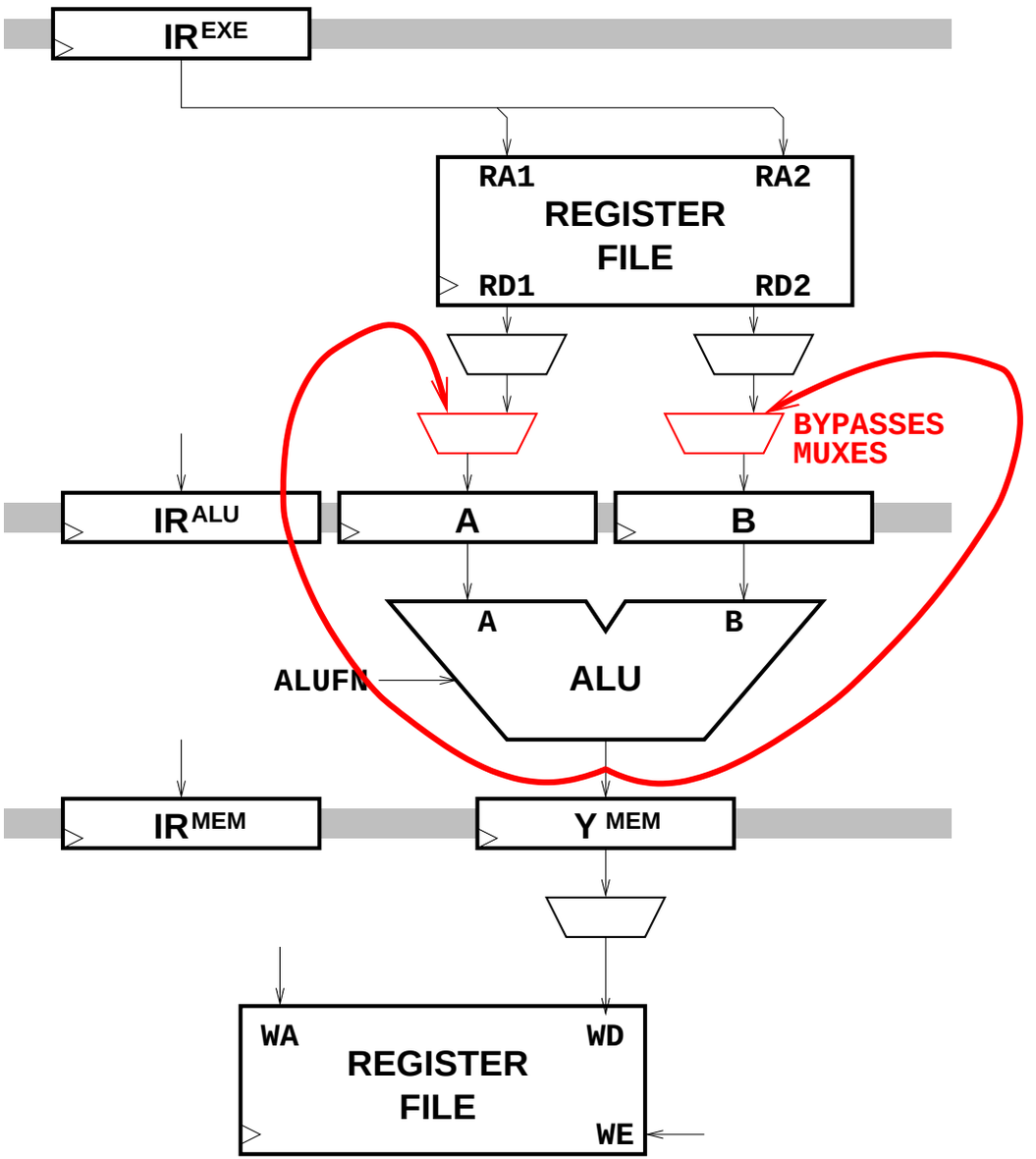


The operation of the 4-stage pipeline: the problem with data conflicts – hardware solution 2

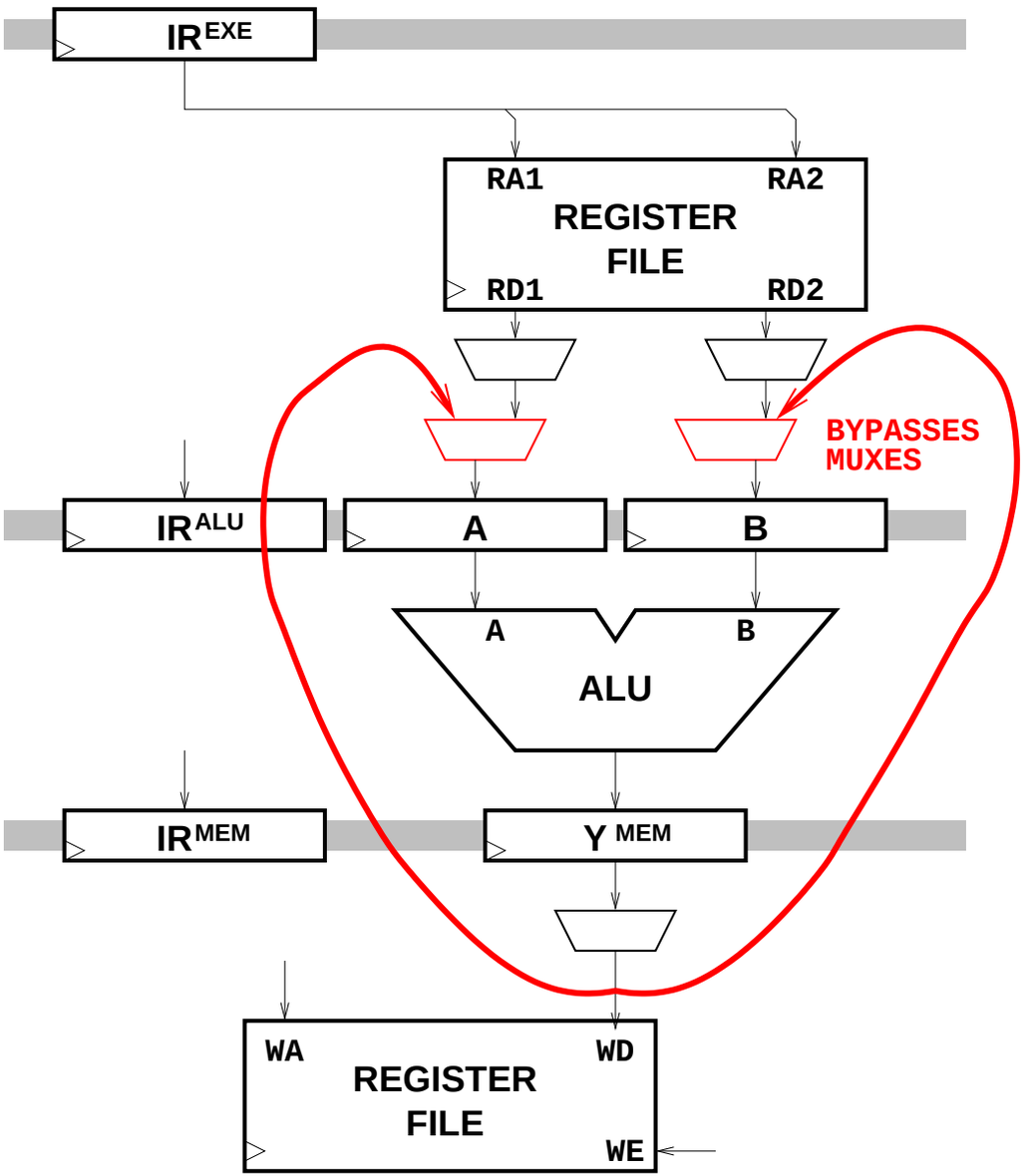
- A second hardware solution introduces *bypass* paths that make it possible to use directly the output of the ALU, instead of the output of the registers.

- It is also necessary to have access to the output of the WB stage, when the conflicting instructions are separated by another instruction. instruction.

Bypass paths 1



Bypass paths 2



Other problems to consider with respect to pipelines

- The conflict between a `LD` instruction and the instruction just following it cannot be solved by a bypass path.

- Reading from memory might be slow, which forces a longer cycle time. One solution is to add a pipeline stage and allow two cycles for memory reads.

- When executing a branch/jump, the correct `LP` must be saved, and similarly for `XP` upon exceptions. The requirement is to maintain coherence between the return address saved and what has already been executed.