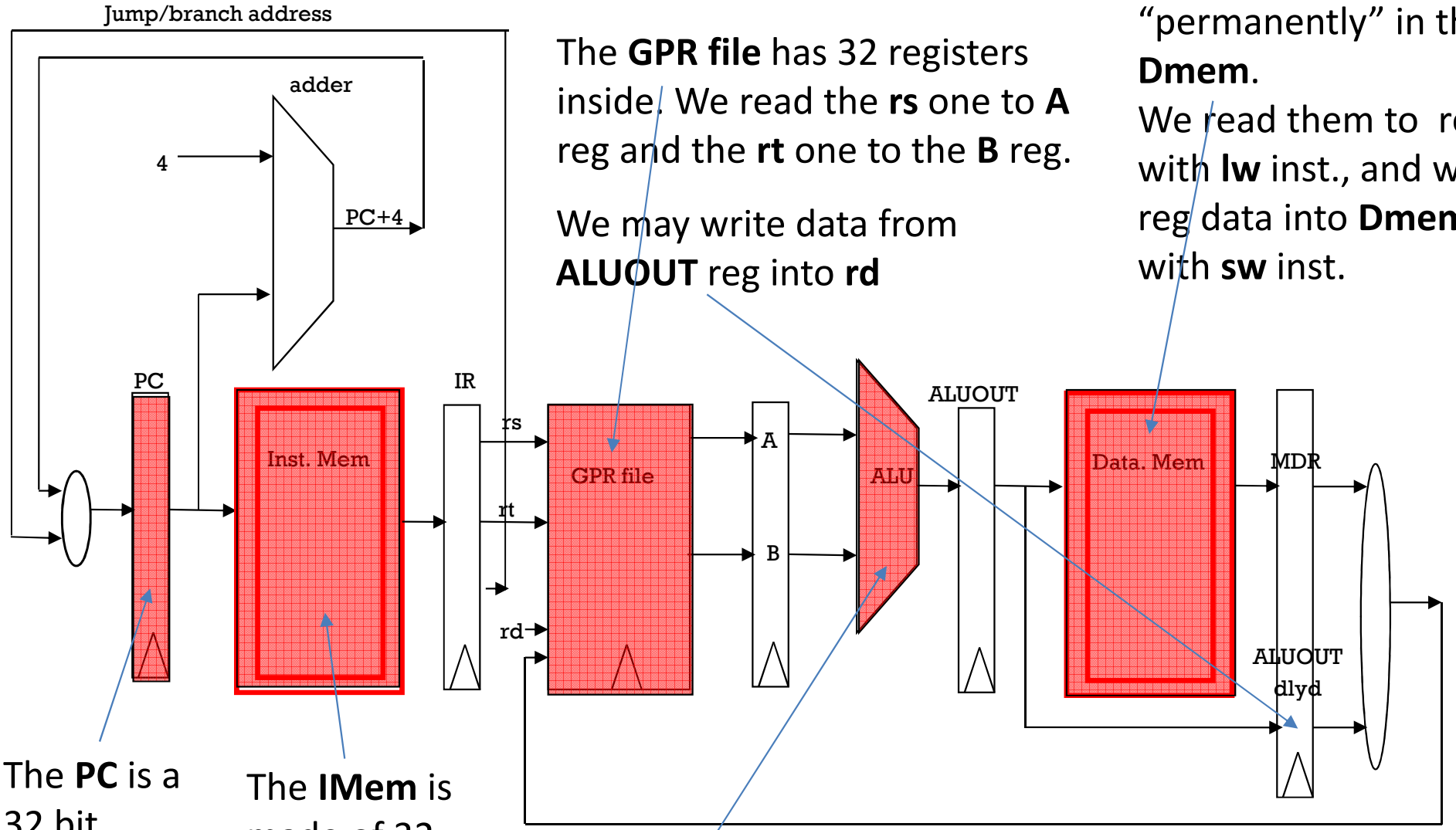
Pipelined MIPS

The pipelined MIPS



The **PC** is a 32 bit register

The **IMem** is made of 32 bit words. Addresses are 0,4,8,C,10

The **GPR file** has 32 registers inside. We read the **rs** one to **A** reg and the **rt** one to the **B** reg.

We may write data from **ALUOUT** reg into **rd**

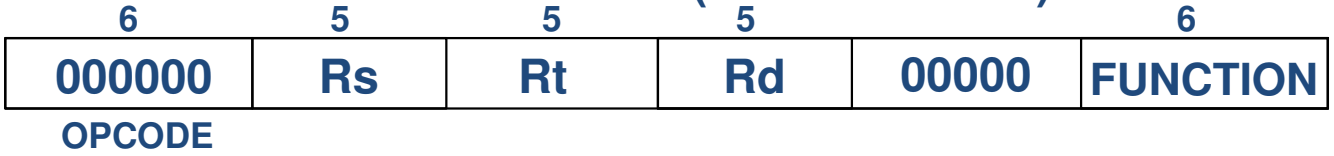
The **ALU** calculates **A** op **B**. Its result goes to the **ALUOUT** reg

Data (variables) reside "permanently" in the **Dmem**.
We read them to reg with **lw** inst., and write reg data into **Dmem** with **sw** inst.

MIPS instruction set

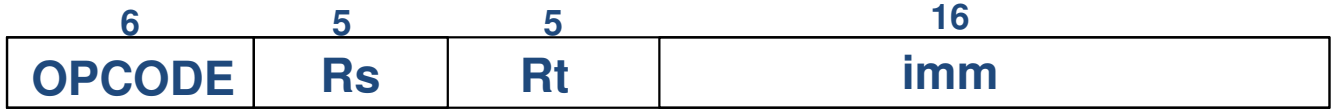
R-type

```
add  Rd, Rs, Rt      #  Rd=Rs+Rt
sub Rd, Rs, Rt       #  Rd=Rs-Rt
and Rd, Rs, Rt       #  Rd=Rs AND Rt
or Rd, Rs, Rt        #  Rd=Rs OR Rt
xor Rd, Rs, Rt       #  Rd=Rs XOR Rt
slt Rd, Rs, Rt       #  if Rs<Rt  Rd=1 else Rd=0
jr Rs                #  PC=Rs    (note that Rd=0)
```



I-type

```
addi  Rt, Rs, imm    #  Rt=Rs+ sext(imm)
lw  Rt, imm(Rs)      #  Rt=M[Rs + sext(imm)]
sw  Rt, imm(Rs)      #  M[Rs + sext(imm)]=Rt
beq  Rs, Rt, label   #  if Rs==Rt,  PC=PC+4+ sext(imm)*4
                     #  else        PC=PC+4
bne  Rs, Rt, label   #  same as beq with cond of Rs≠Rt
ori  Rt, Rs, imm     #  Rt=Rs OR imm     (no sext)
lui   Rt, imm        #  Rt= imm<<16   (no sext)
```



---

j-type

```
j    imm             #  PC= imm*4                  (no sext)
jal  imm             #  PC= imm*4,  $31=PC+4       (no sext)
```

MIPS instructions

R-type instructions: add, sub, slt ,and, or, xor (& jr)

sub rd, rs, rt **# rd = rs - rt**

means subtract the contents of **rt** from the contents of **rs** and store the result into **rd**

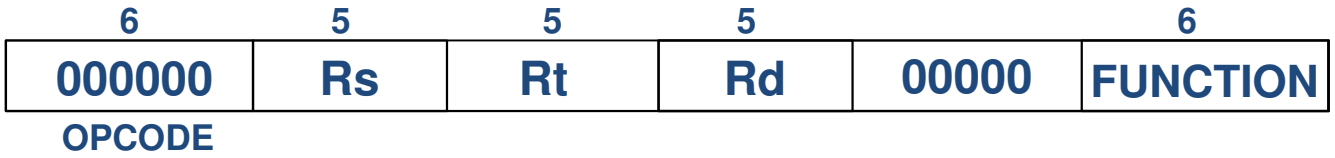
The R-type instructions we use are:

add rd, rs, rt **# rd = rs + rt**
sub rd, rs, rt **# rd = rs – rt**
and rd, rs, rt **# rd = rs AND rt**
or rd, rs, rt **# rd = rs OR rt**
xor rd, rs, rt **# rd = rs XOR rt**
slt rd, rs, rt **# rd =1 if rs < rt, 0 otherwise [=set less than]**

The last R-type instruction is:

jr rs **# PC = rs**

The R-type instruction binary representation is:



MIPS instructions

I-type instructions: addi, ori, beq, bne , lui, lw, sw

addi rt, rs, imm **# rt = rs + sext(imm)** **[imm is 16 bit 2's comp]**

means add imm to the contents of **rs** and put the result in **rt**

ori rt, rs, imm **# rt = rs OR imm** **[imm is 16 bits –no sext]**

means OR imm with the contents of **rs** and put the result in **rt** (16 MSBs will not change)

lui rt, imm **# rt = imm << 16** **[imm is 16 bits –no sext]**

means shift imm left by 16 and put the result in **rt** (16 LSBs will be '0'-s)

---

beq rt, rs, imm **# if rs == rt** **PC = PC + 4 + 4*sext(imm)** **[imm is 16 bit 2's comp]**

else PC = PC +4

means if the contents of **rs** equals the contents of **rt**, jump imm+1 instructions forward

bne - same, but checks if not equal

---

lw rt, (imm) rs **# rt = M[rs + sext(imm)]** **[imm is 16 bit 2's comp]**

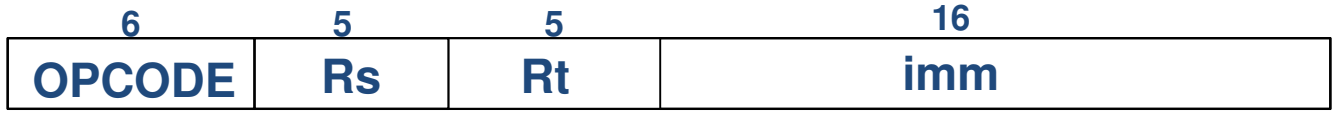
Calc. address by adding imm to the contents of **rs**, read from the D.Mem and copy into **rt**

sw rt, (imm) rs **# M[rs + sext(imm)]= rt** **[imm is 16 bit 2's comp]**

Calc. address by adding imm to the contents of **rs**, write **rt** into that address in the D.Mem

---

The I-type instruction binary representation is:



MIPS instructions

J-type instructions: j, jal

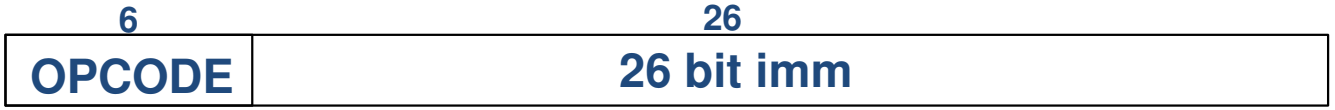
j imm26 **# PC = PC[31:28]|| 4*imm26 [imm26 = 26 bits]**

means jump to the word specified by the 26 bit imm in the instruction

jal imm26 **# PC = PC[31:28]|| 4*imm26 [imm26 = 26 bits]**
 # $31= PC+4

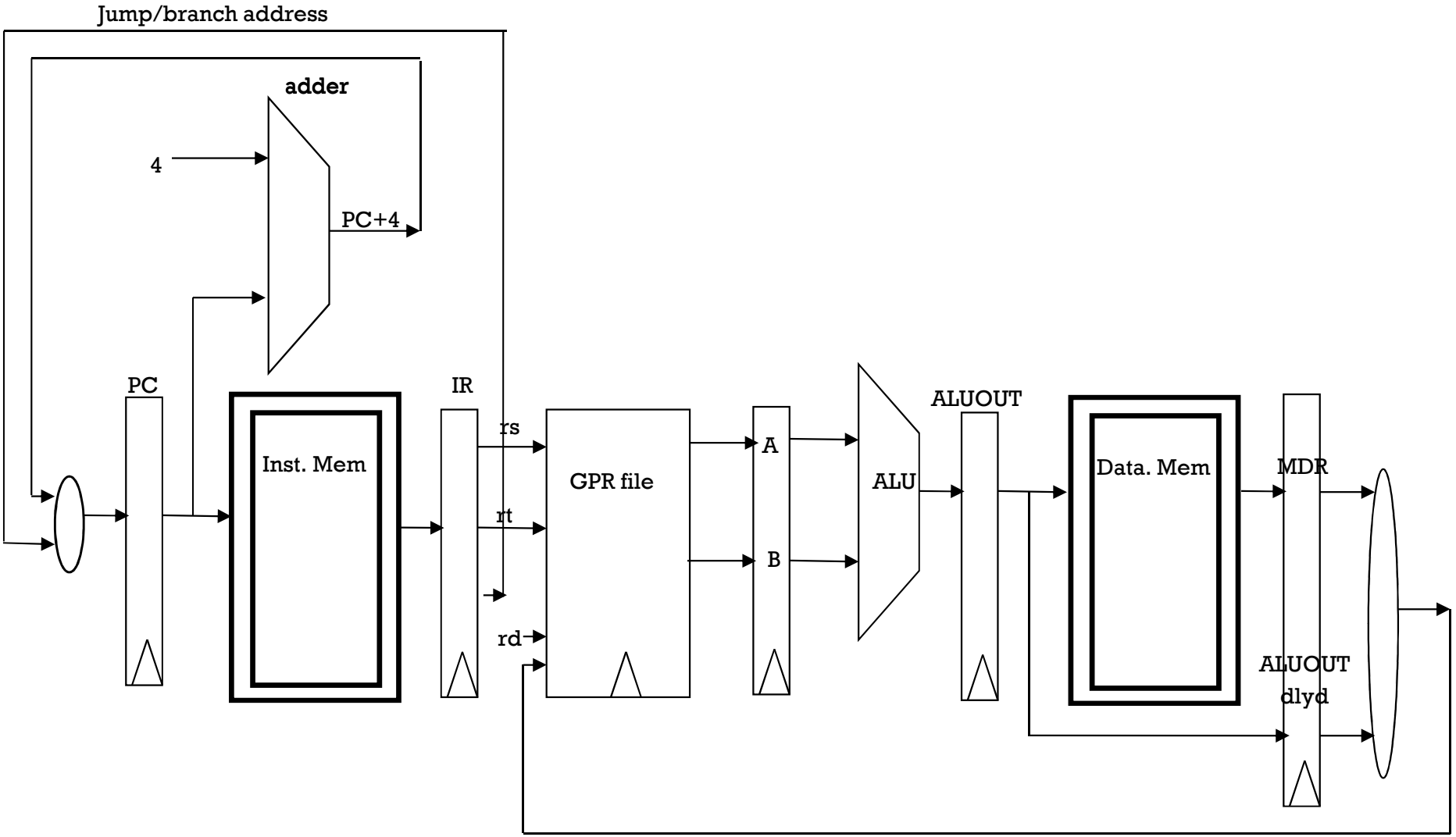
means jump to the word specified by the 26 bit imm in the instruction, and also keep the address of the next instructions in register $31

The J-type instruction binary representation is:

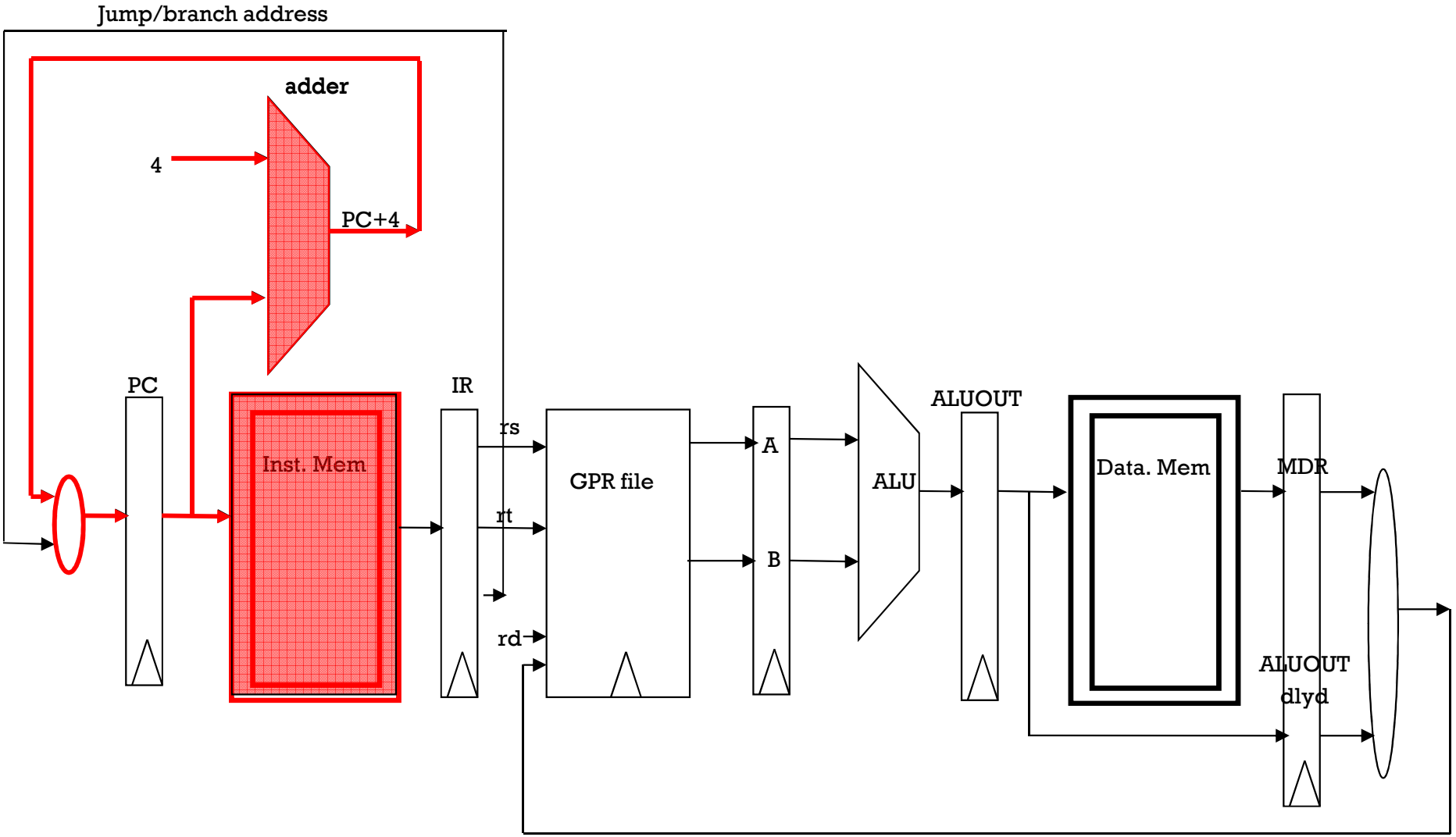


Rtype instructions

Performing Rtype inst.



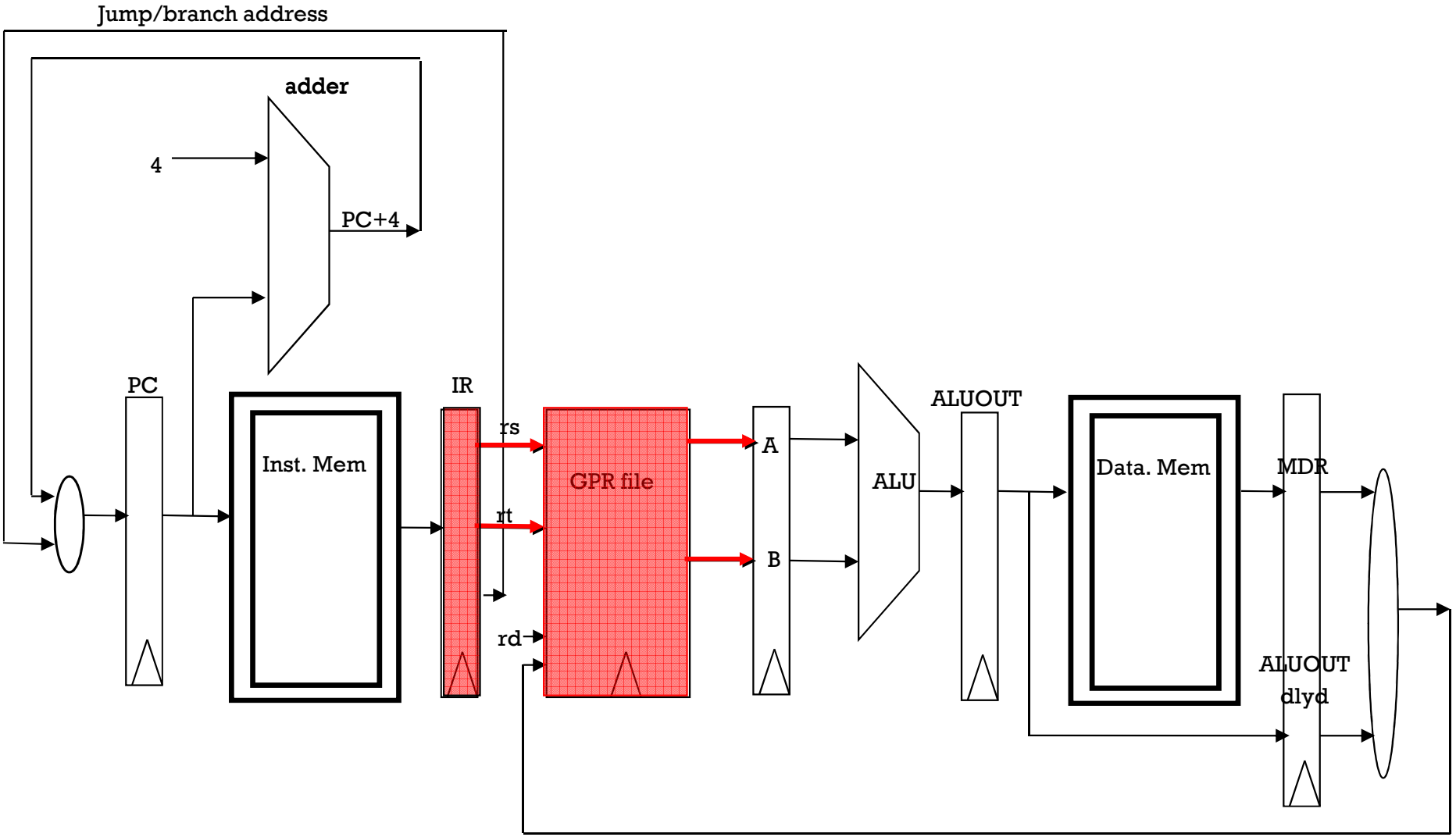
Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

Performing Rtype inst.

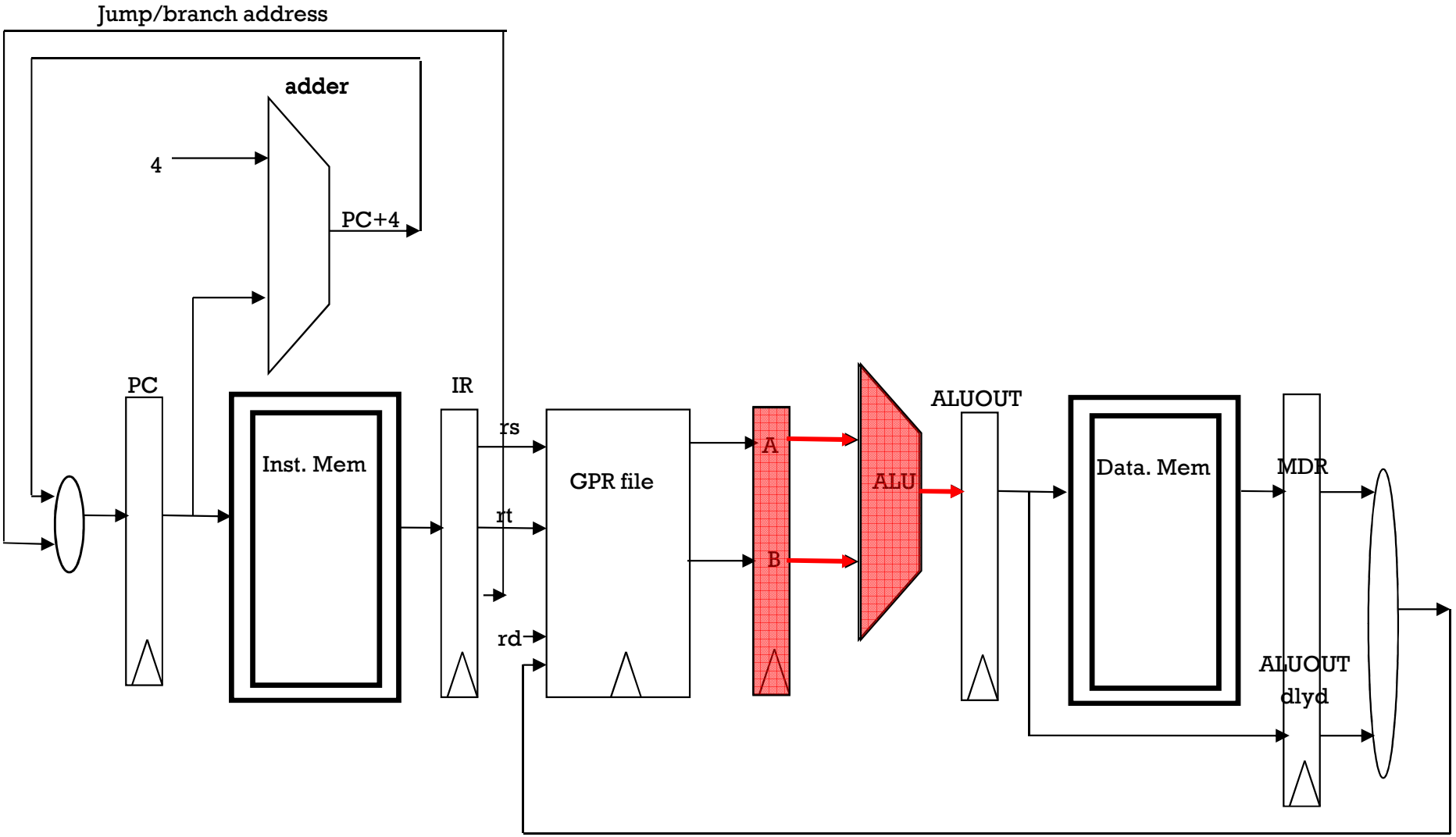


ID – Inst. Decode

A = rs , B = rt

(& decode
control signals)

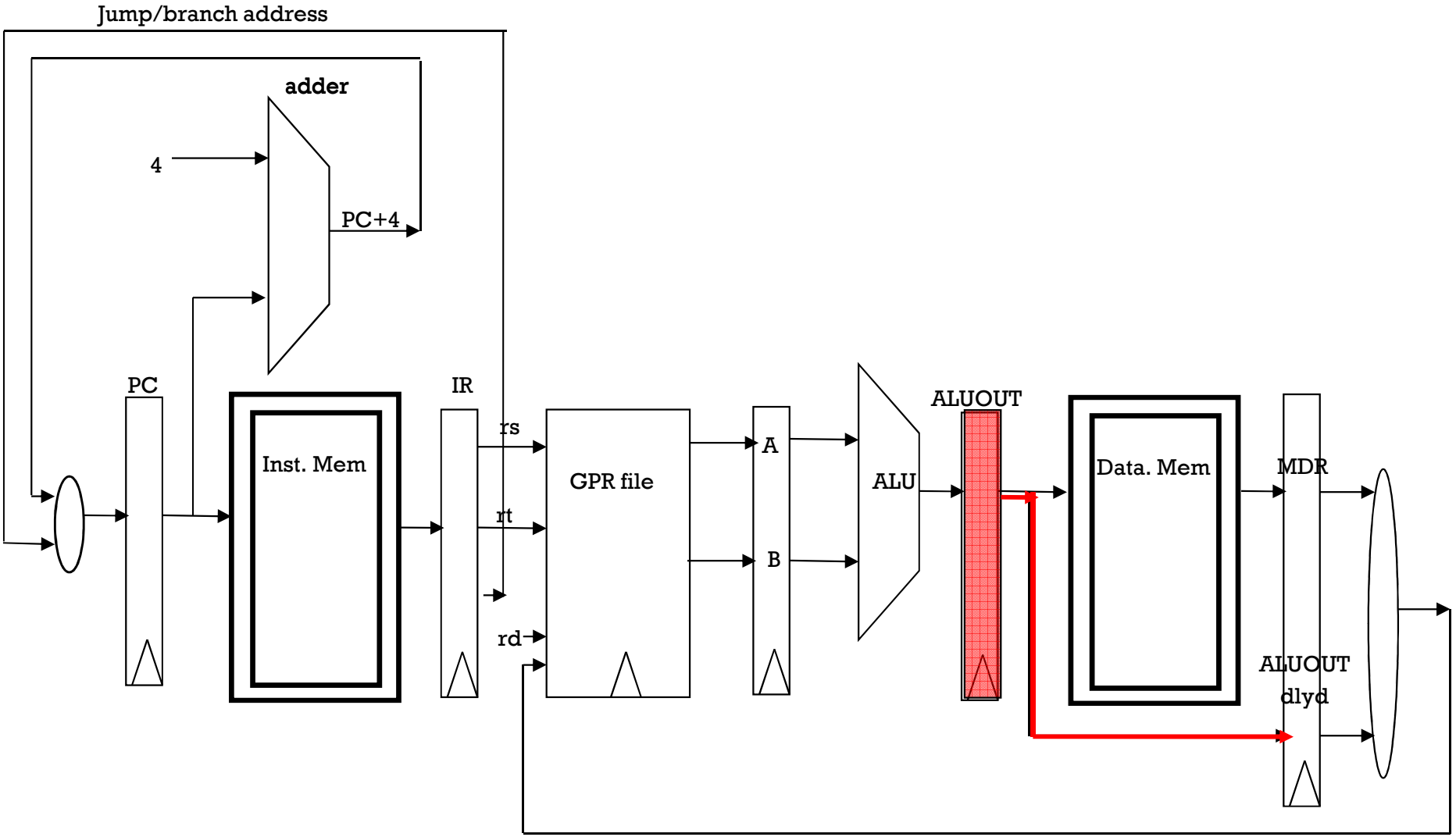
Performing Rtype inst.



EX– Execute

ALUOUT = A op B

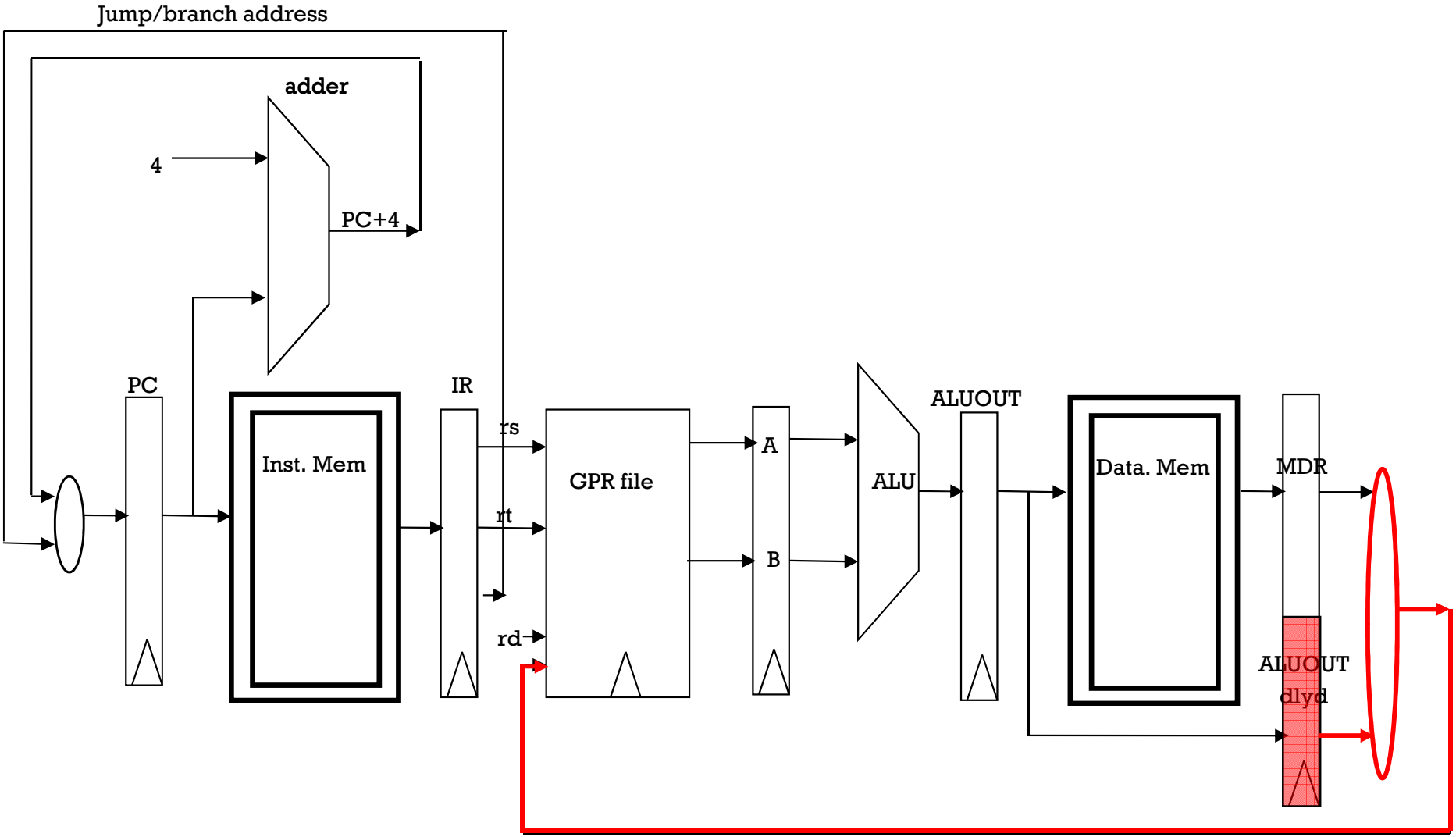
Performing Rtype inst.



MEM – Memory

In Rtype – wait 1 ck

Performing Rtype inst.

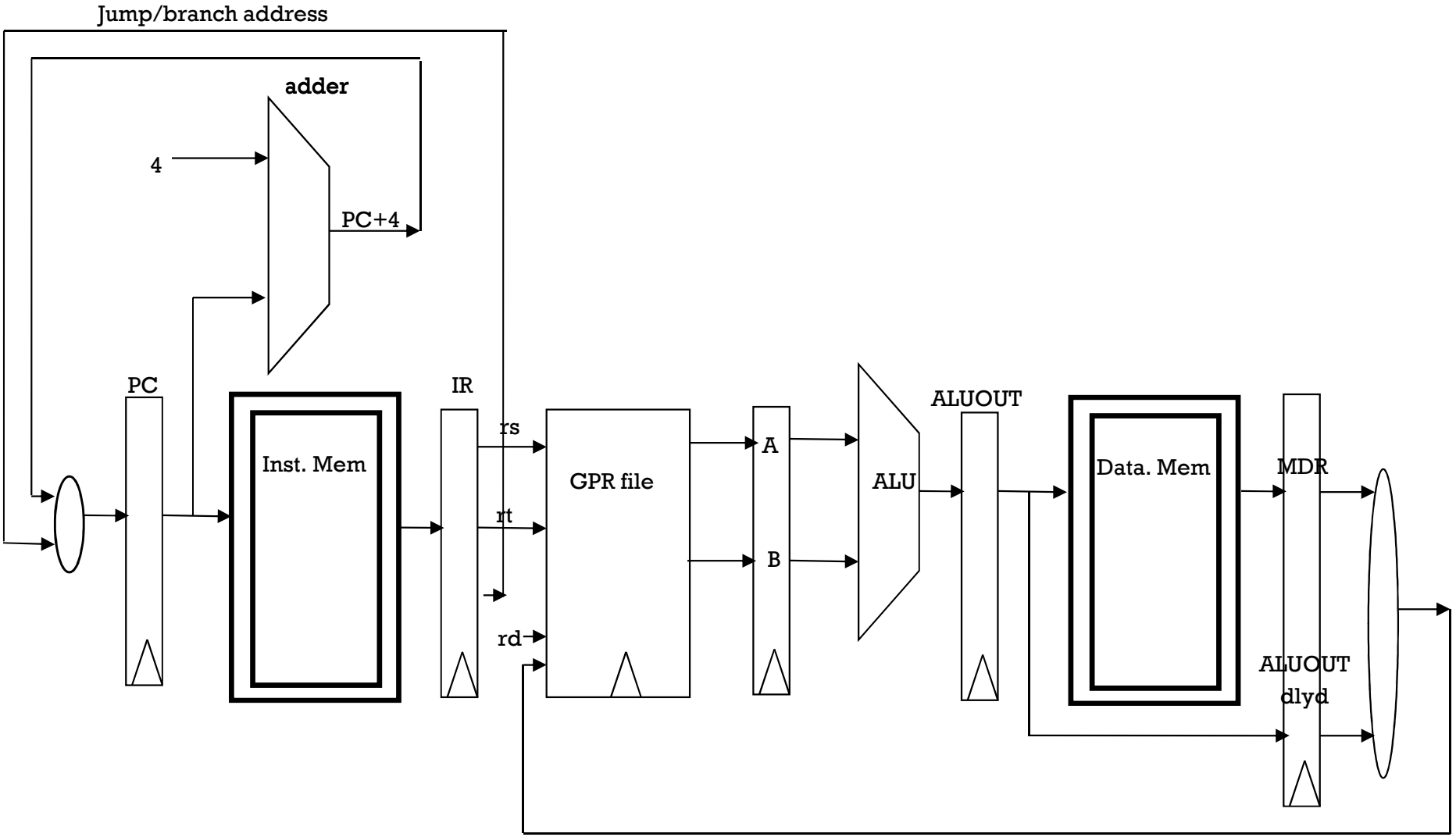


WB – write back

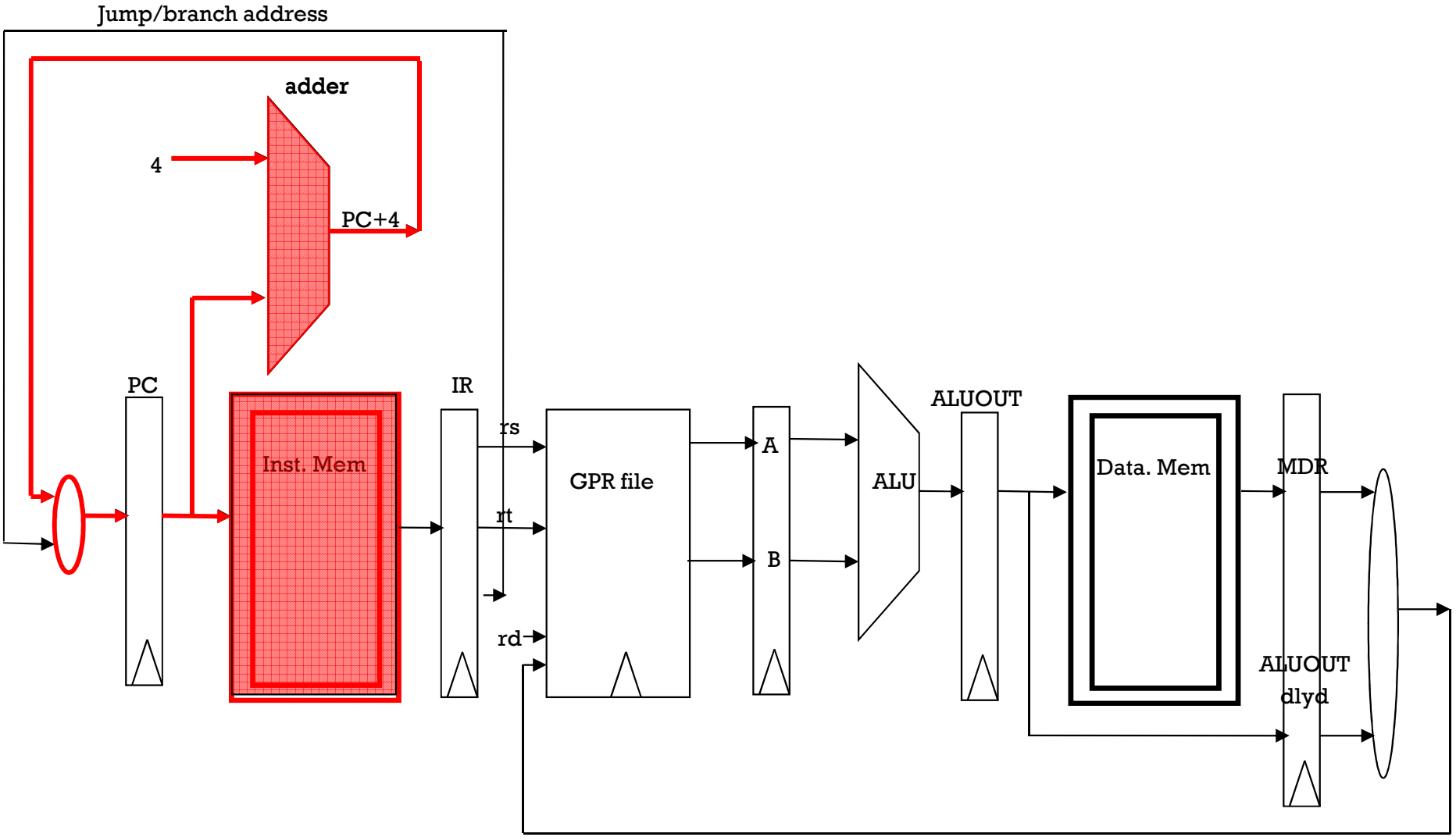
Rd = ALUOUT

lw instruction

Performing lw inst.



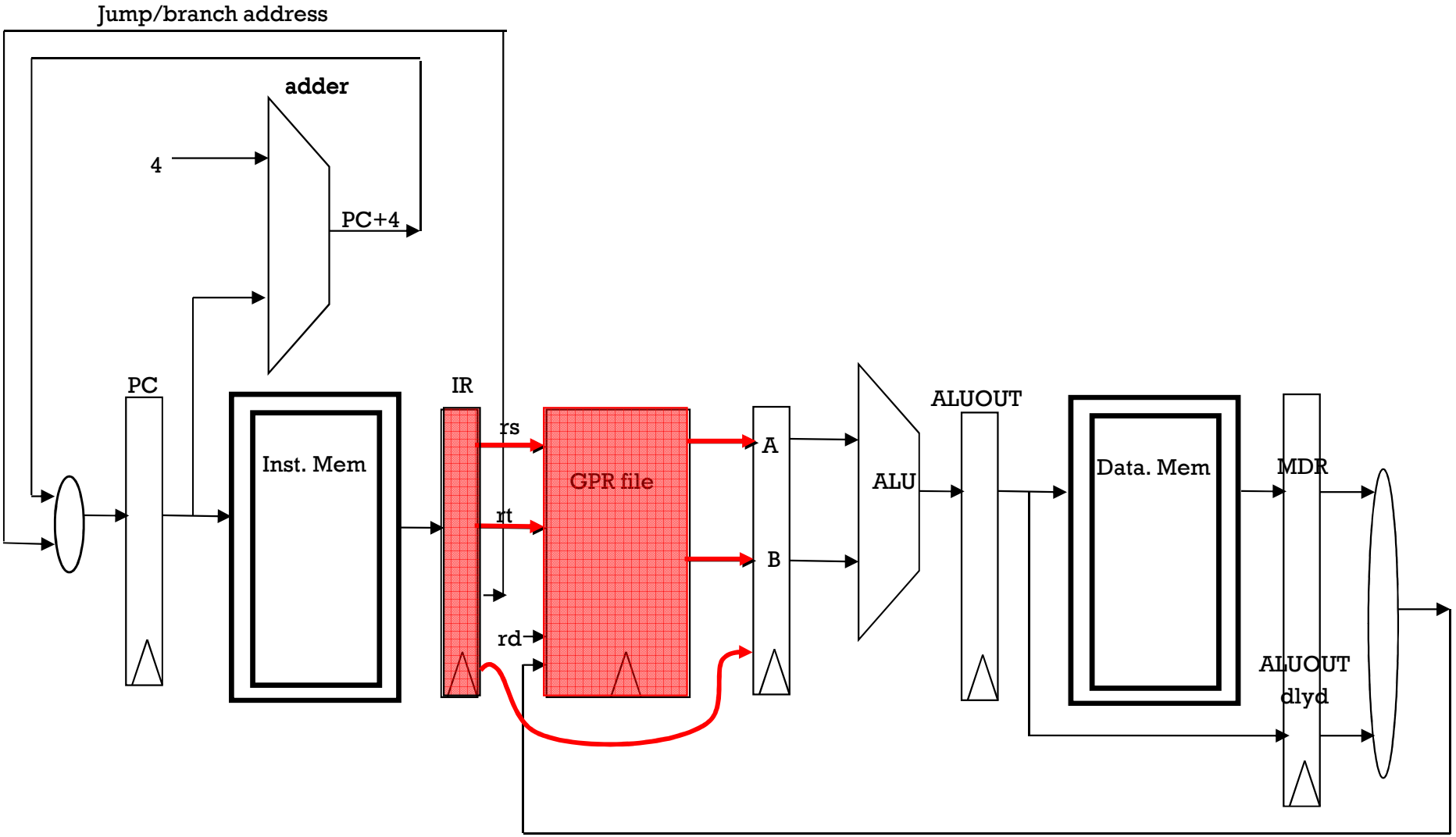
Performing lw inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

Performing lw inst.

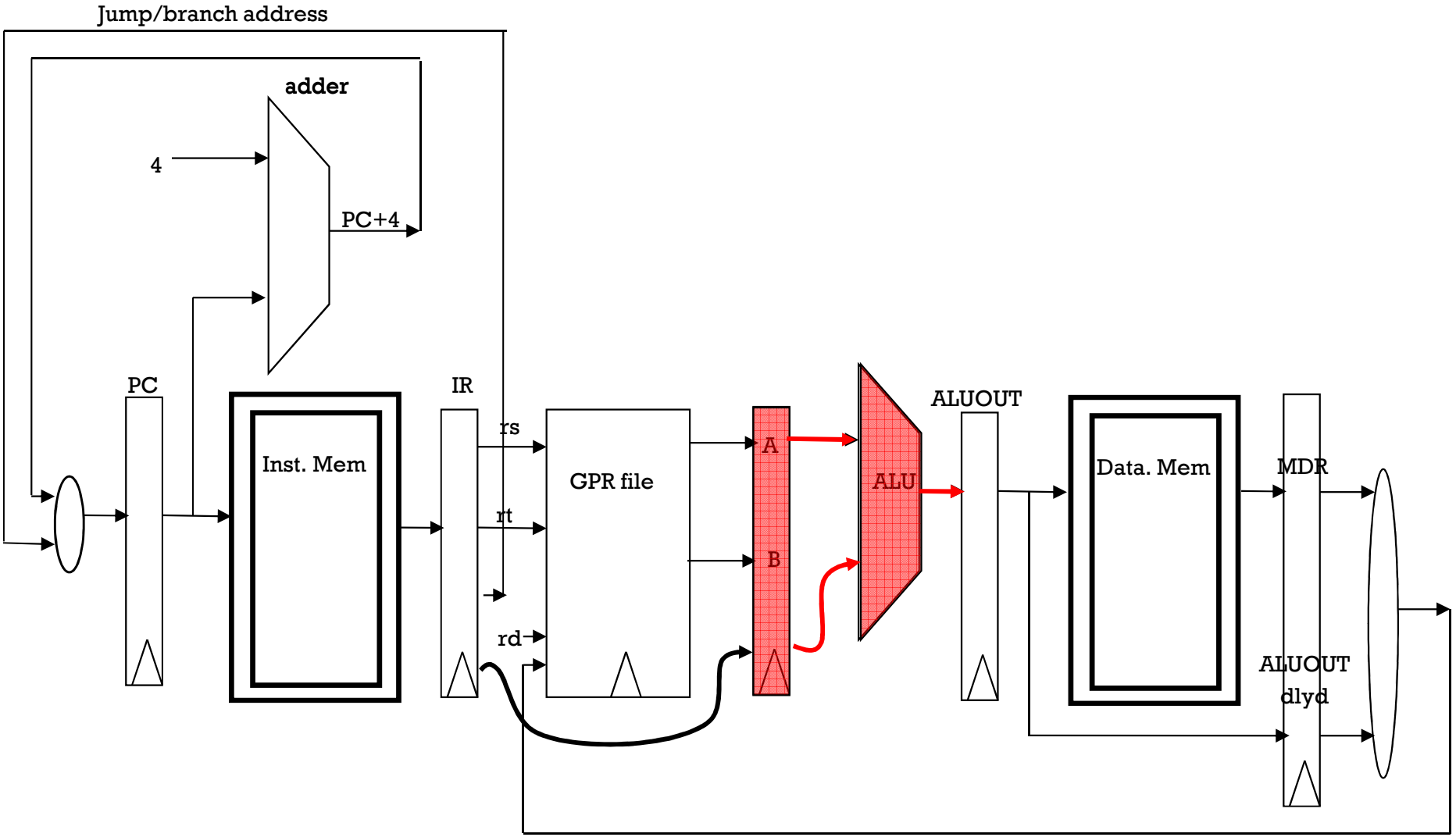


ID – Inst. Decode

A = rs , B = rt

(& decode
control signals)

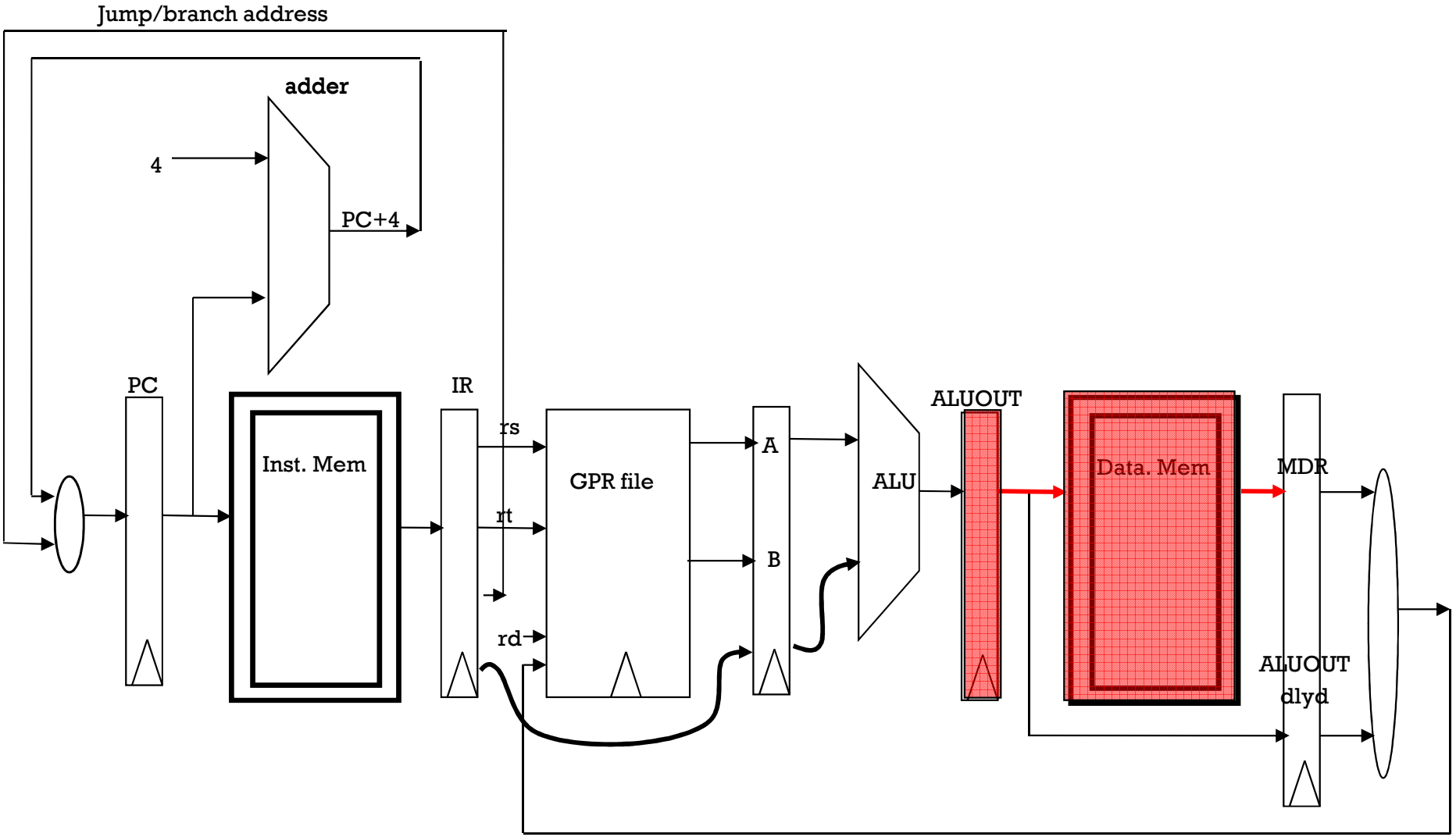
Performing lw inst.



EX– Execute

ALUOUT = A+sext(imm)

Performing lw inst.

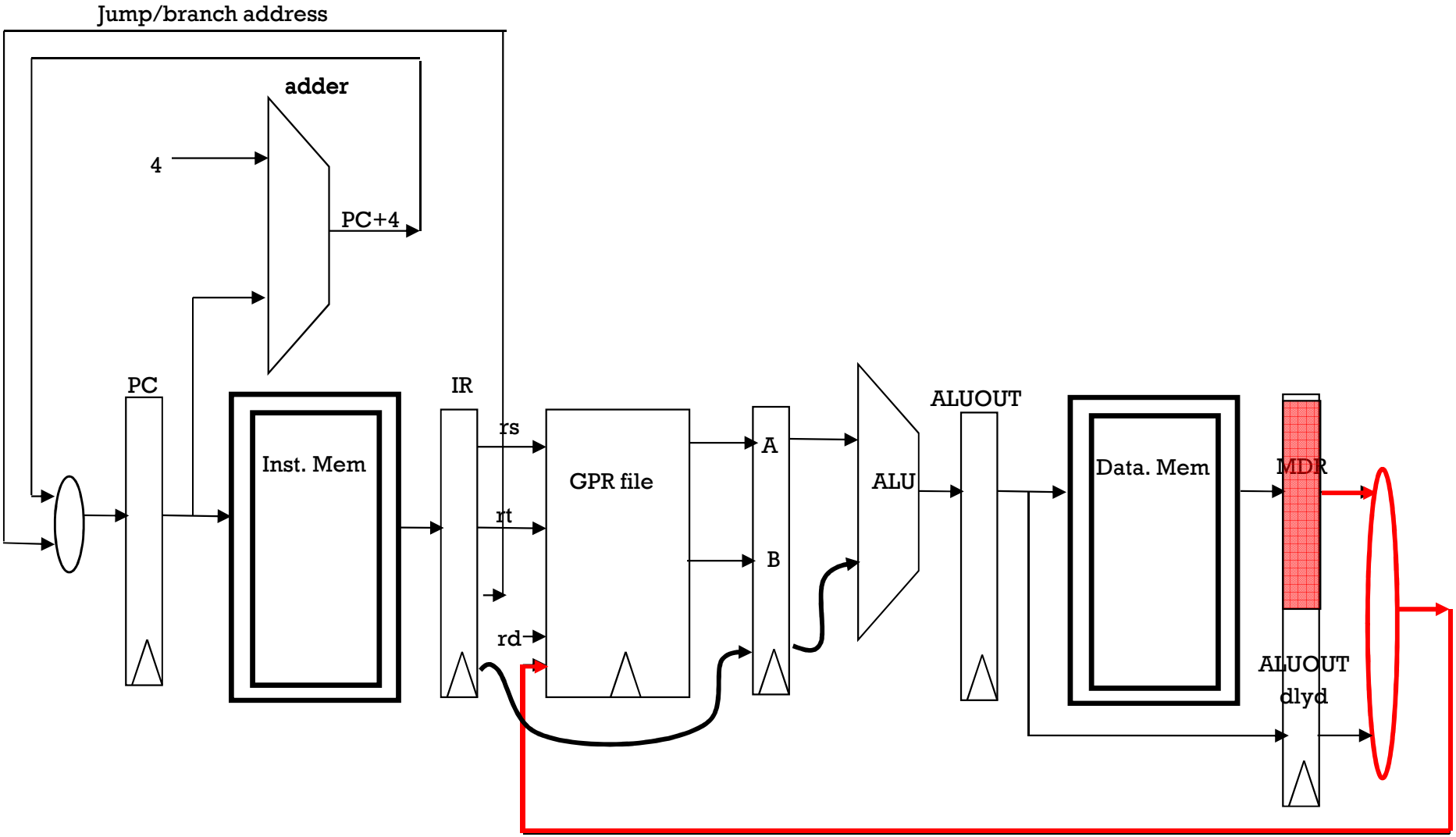


MEM – Memory

In lw:

MDR = M[ALUOUT]

Performing lw inst.

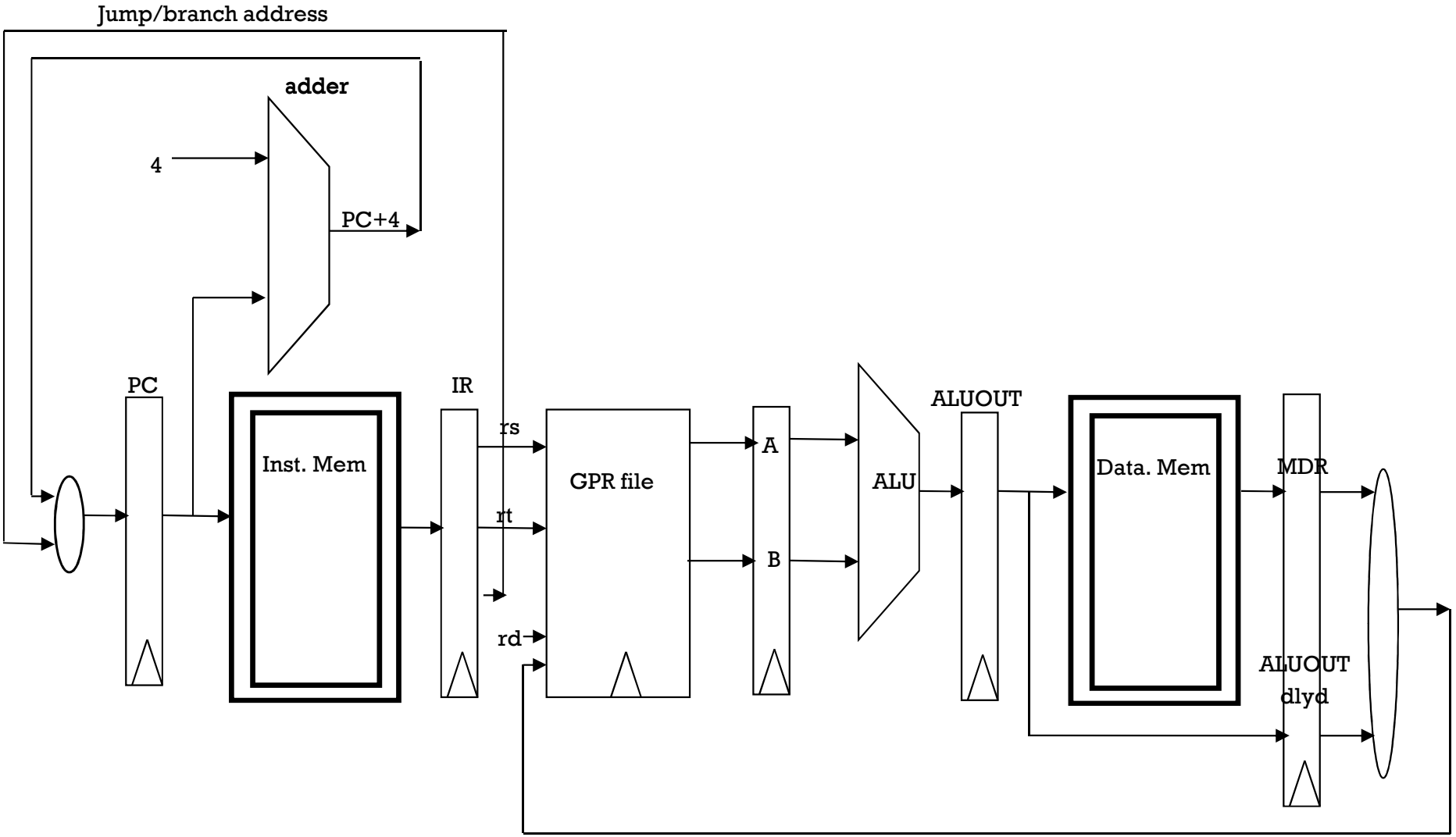


WB – write back

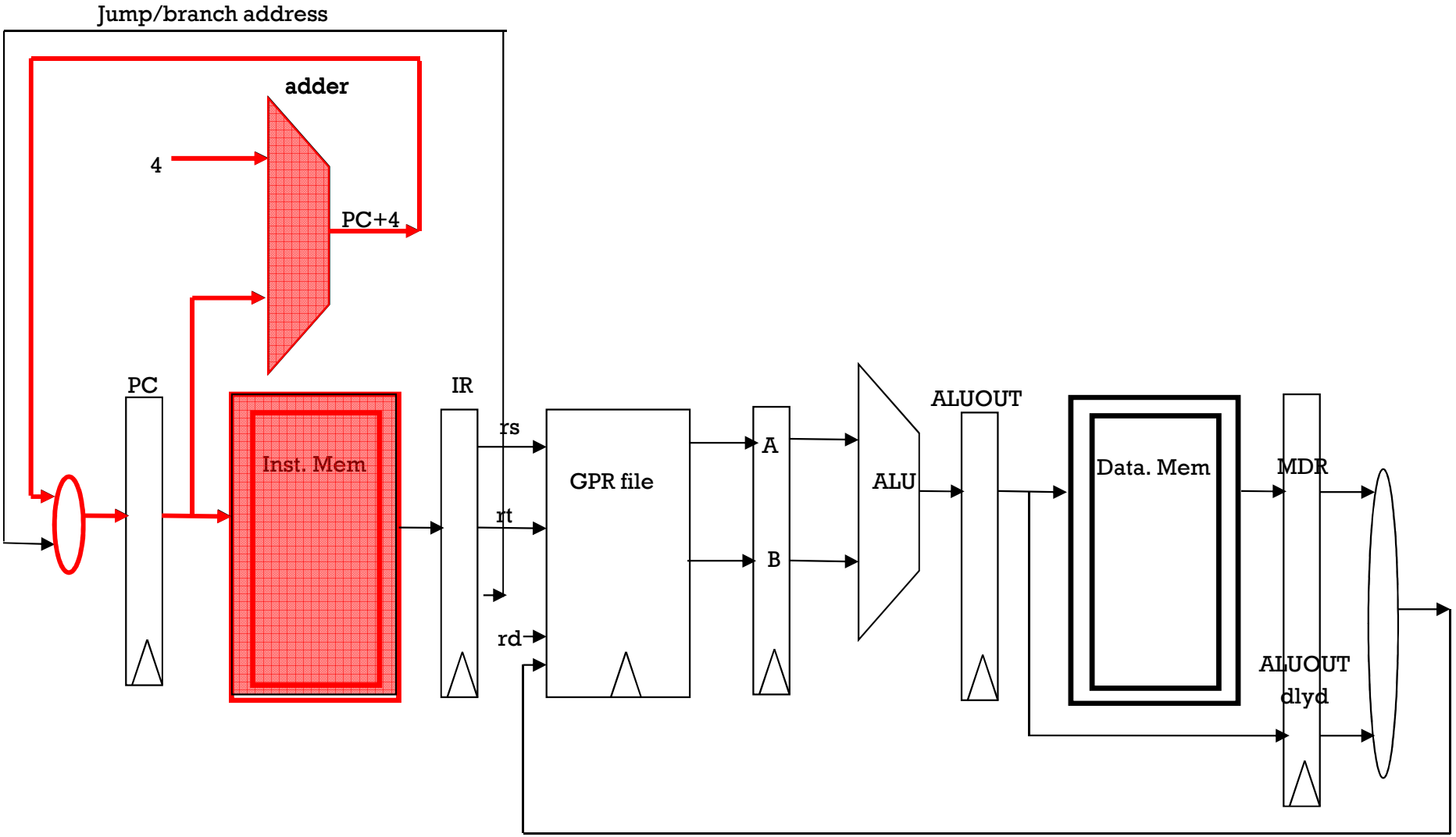
Rt = MDR

sw instruction

Performing sw inst.



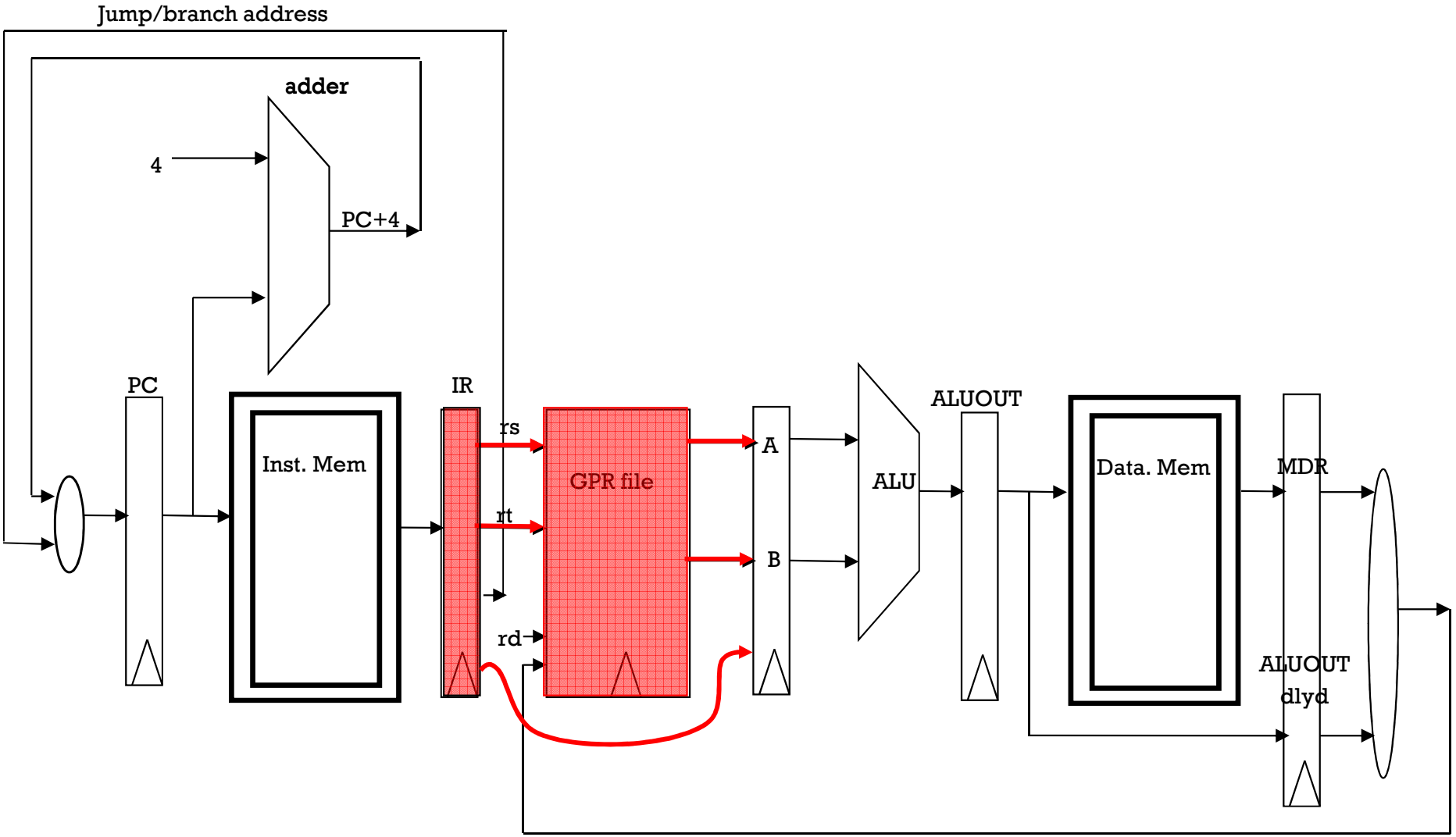
Performing sw inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

Performing sw inst.

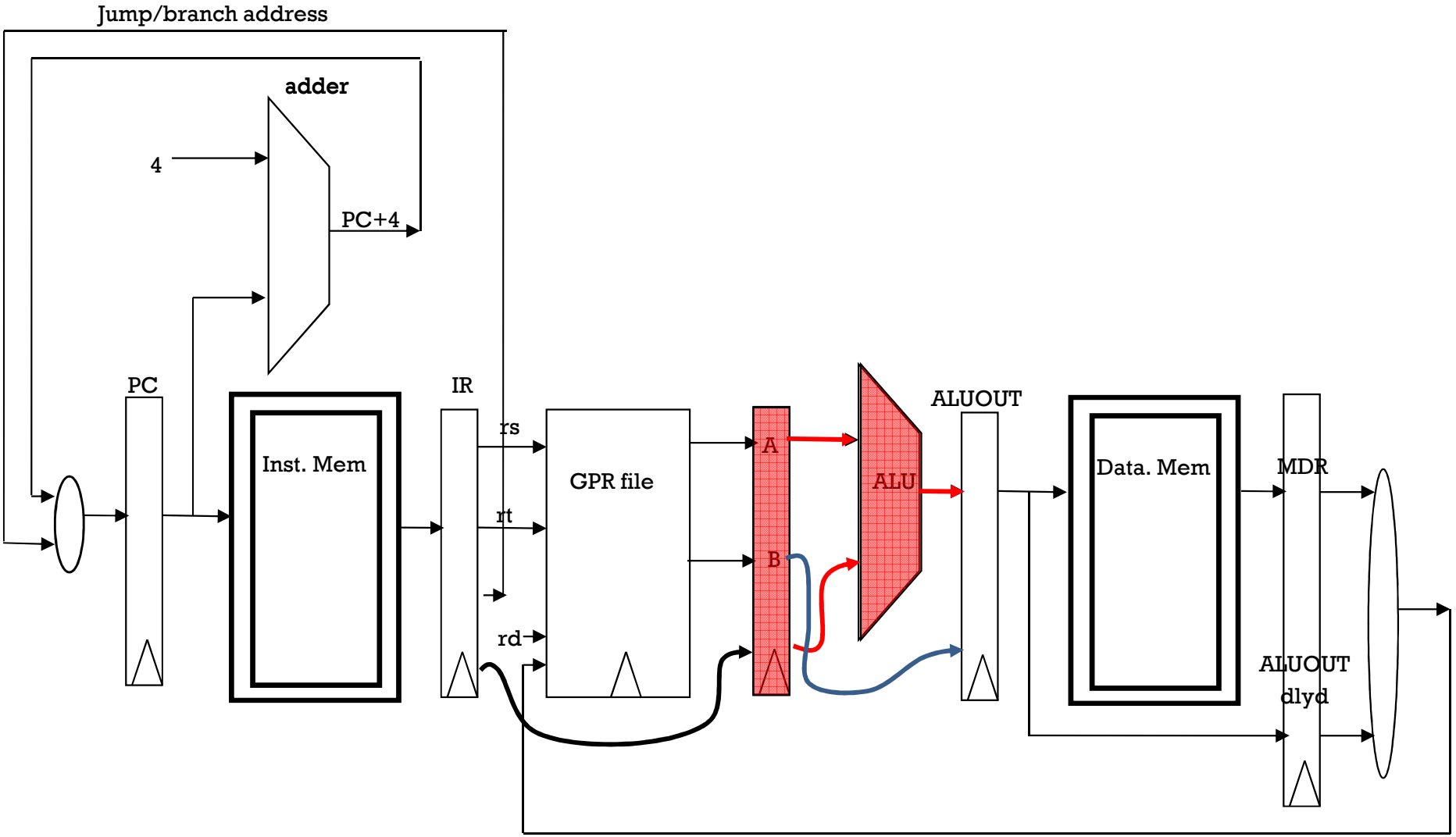


ID – Inst. Decode

A = rs , B = rt

(& decode
control signals)

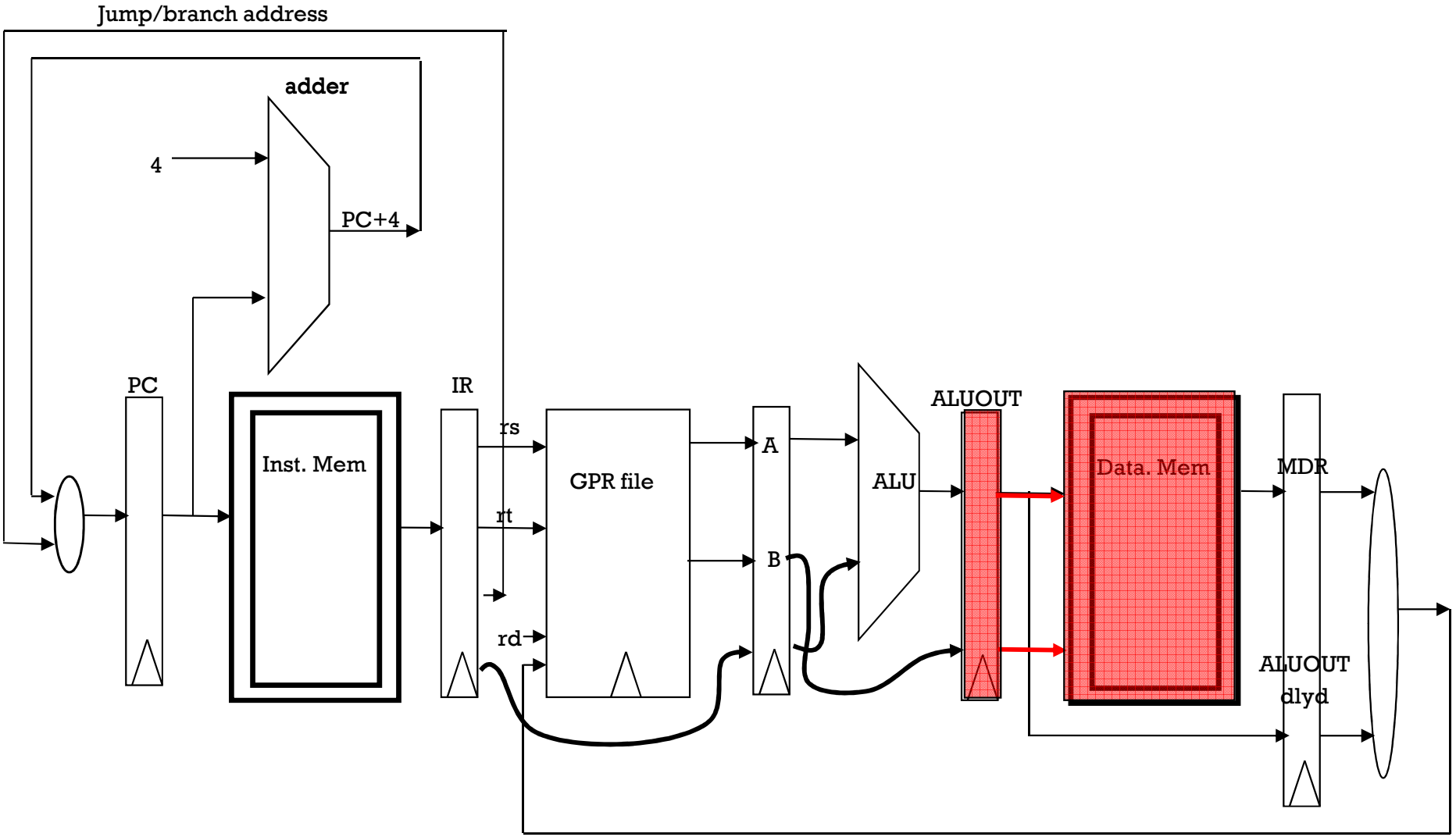
Performing sw inst.



EX– Execute

ALUOUT = A+sext(imm)

Performing sw inst.

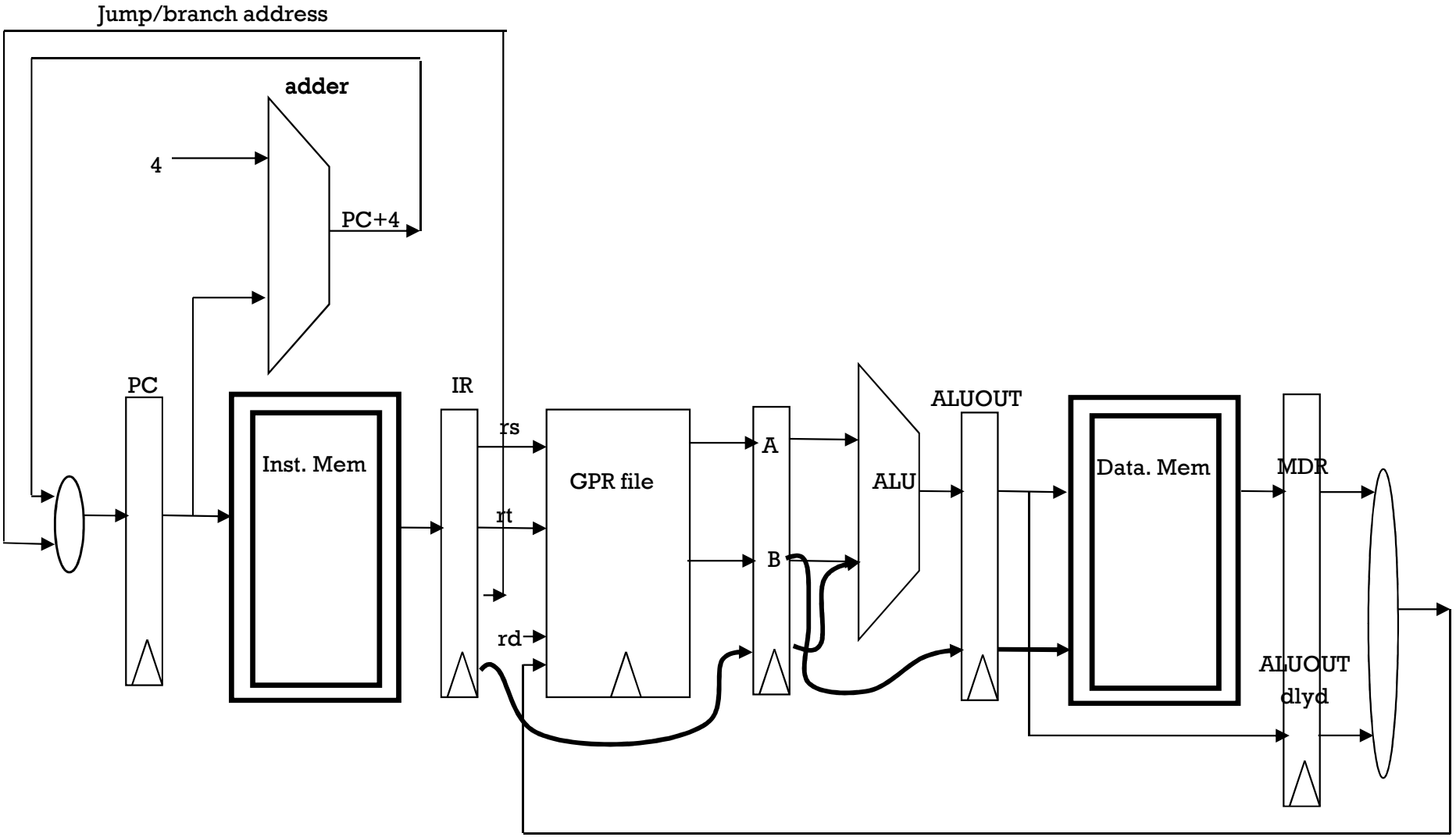


MEM – Memory

In sw:

M[ALUOUT] = B dlyd

Performing lw inst.

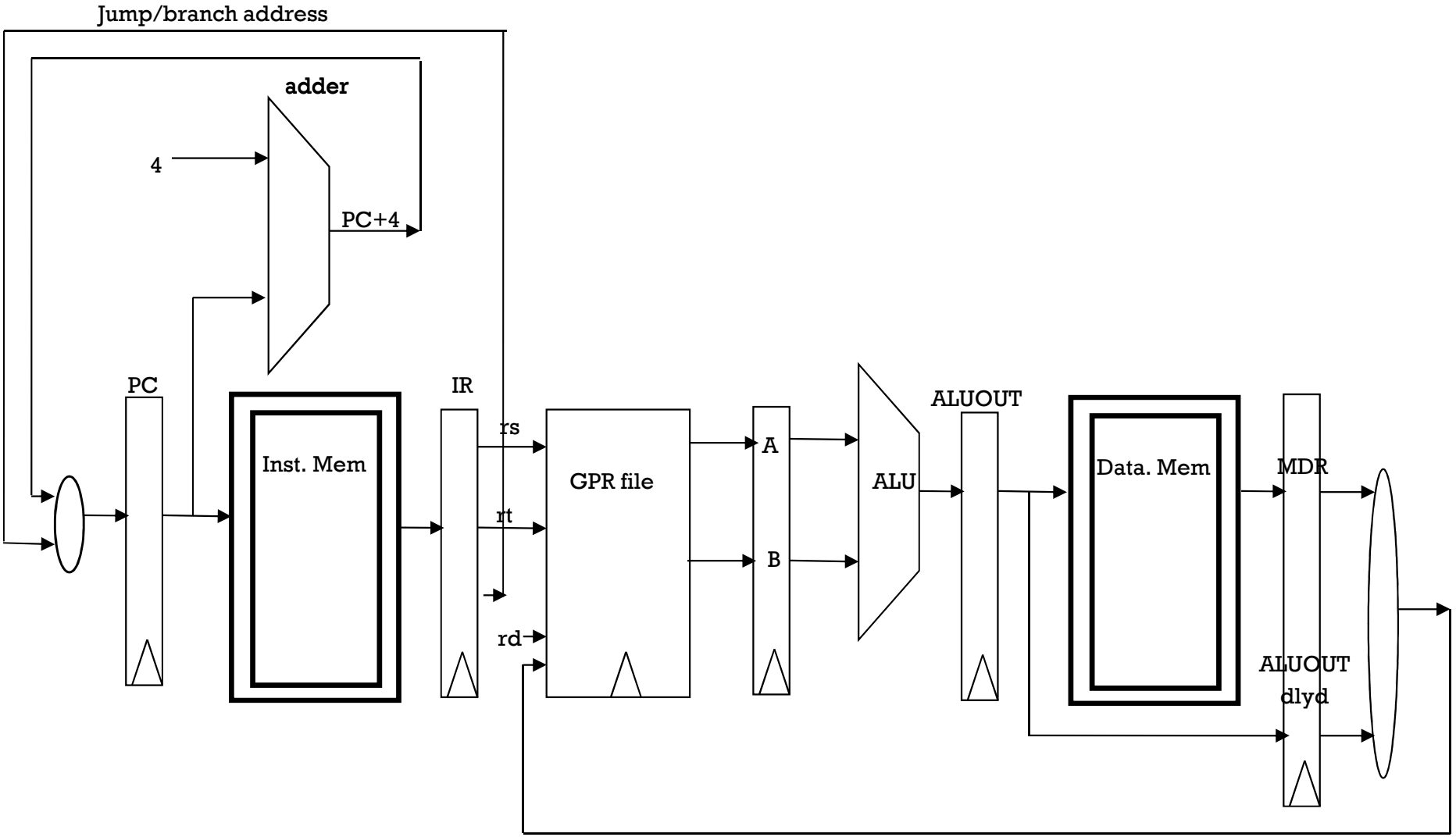


WB – write back

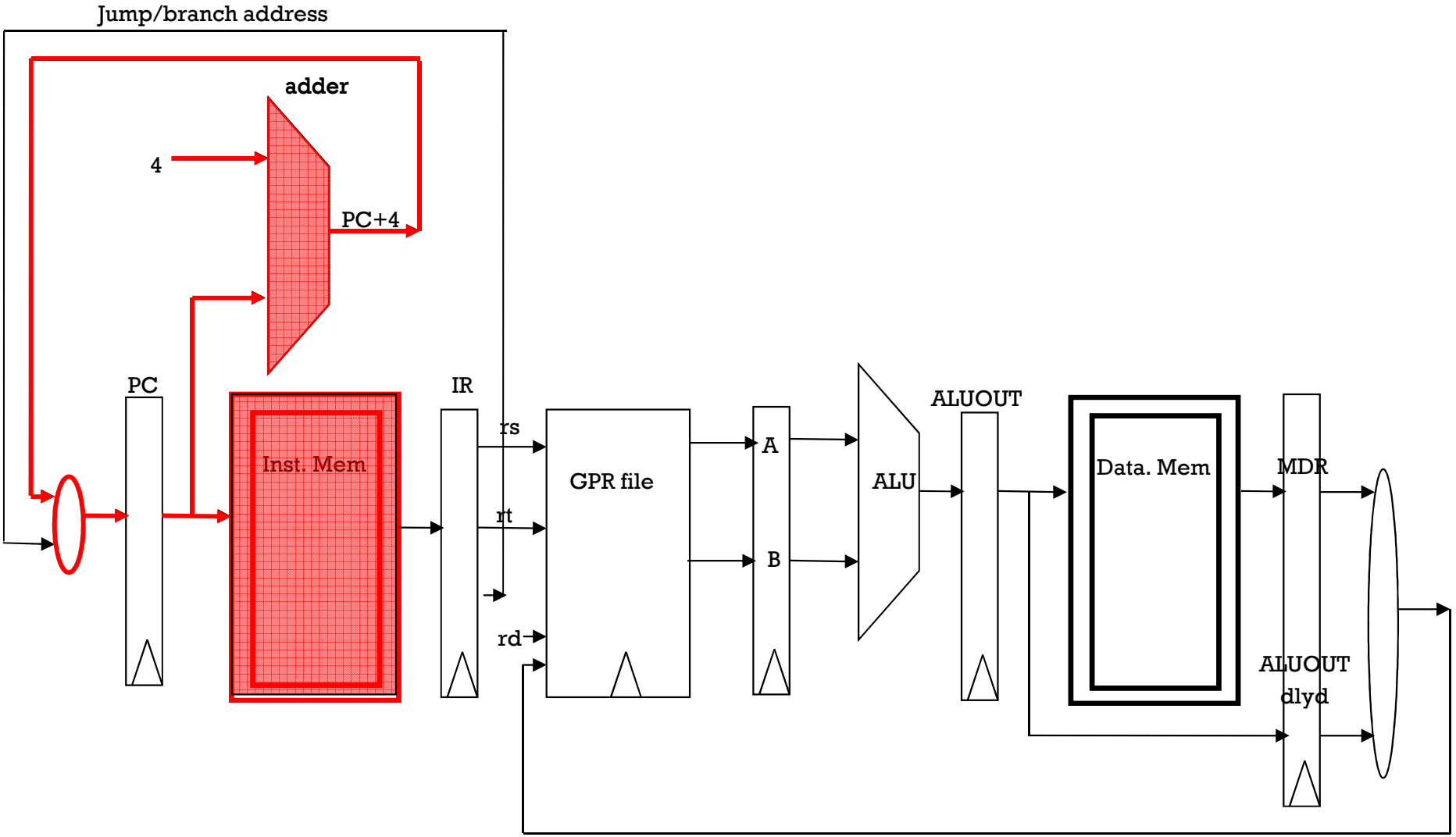
Do nothing

j or beq instructions

Performing jump or branch inst.



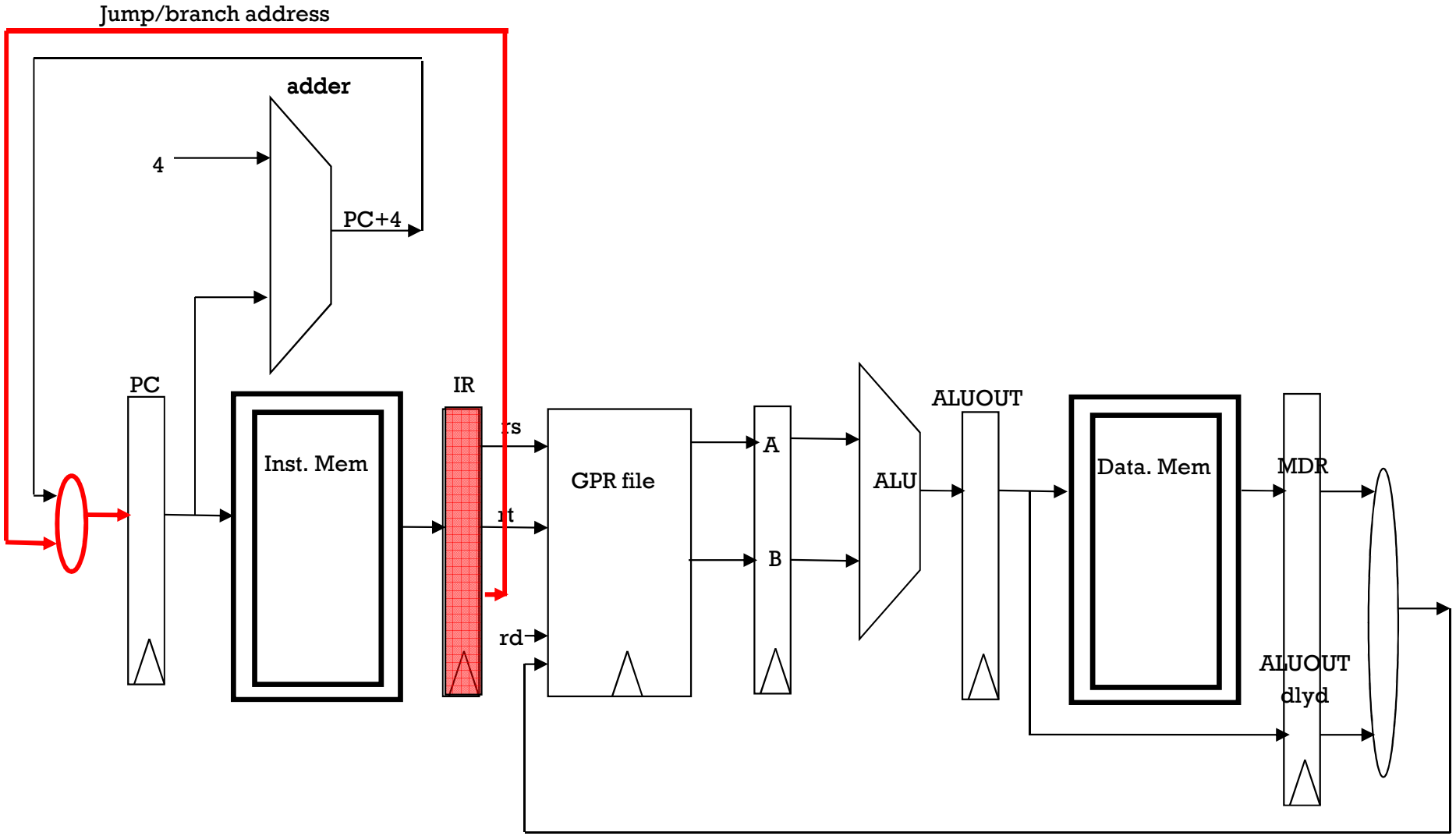
Performing jump or branch inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

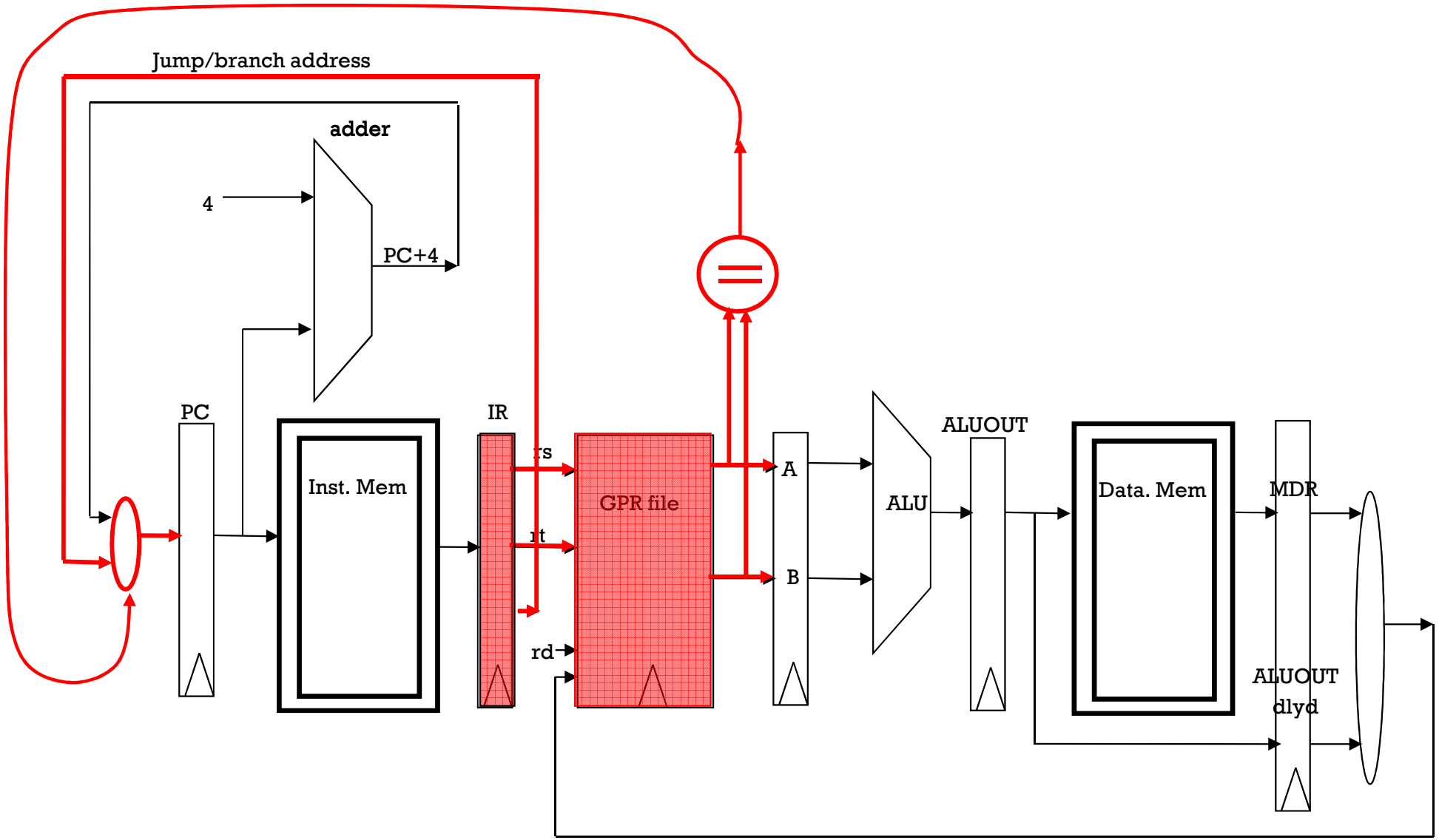
Performing jump or branch inst.



ID – Inst. Decode
In jump:

PC = jump adrs

Performing jump or branch inst.

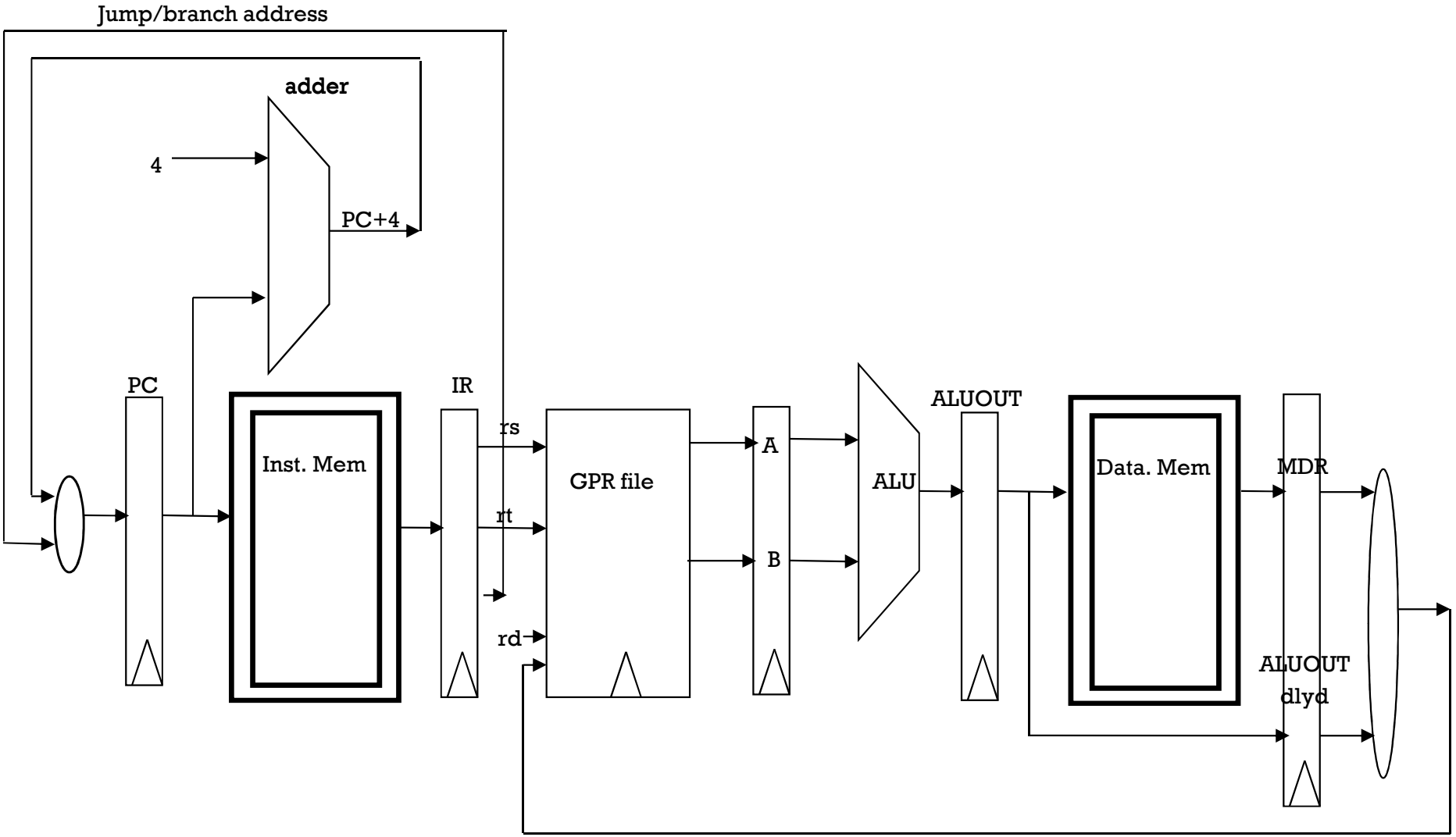


ID – Inst. Decode
In jump:

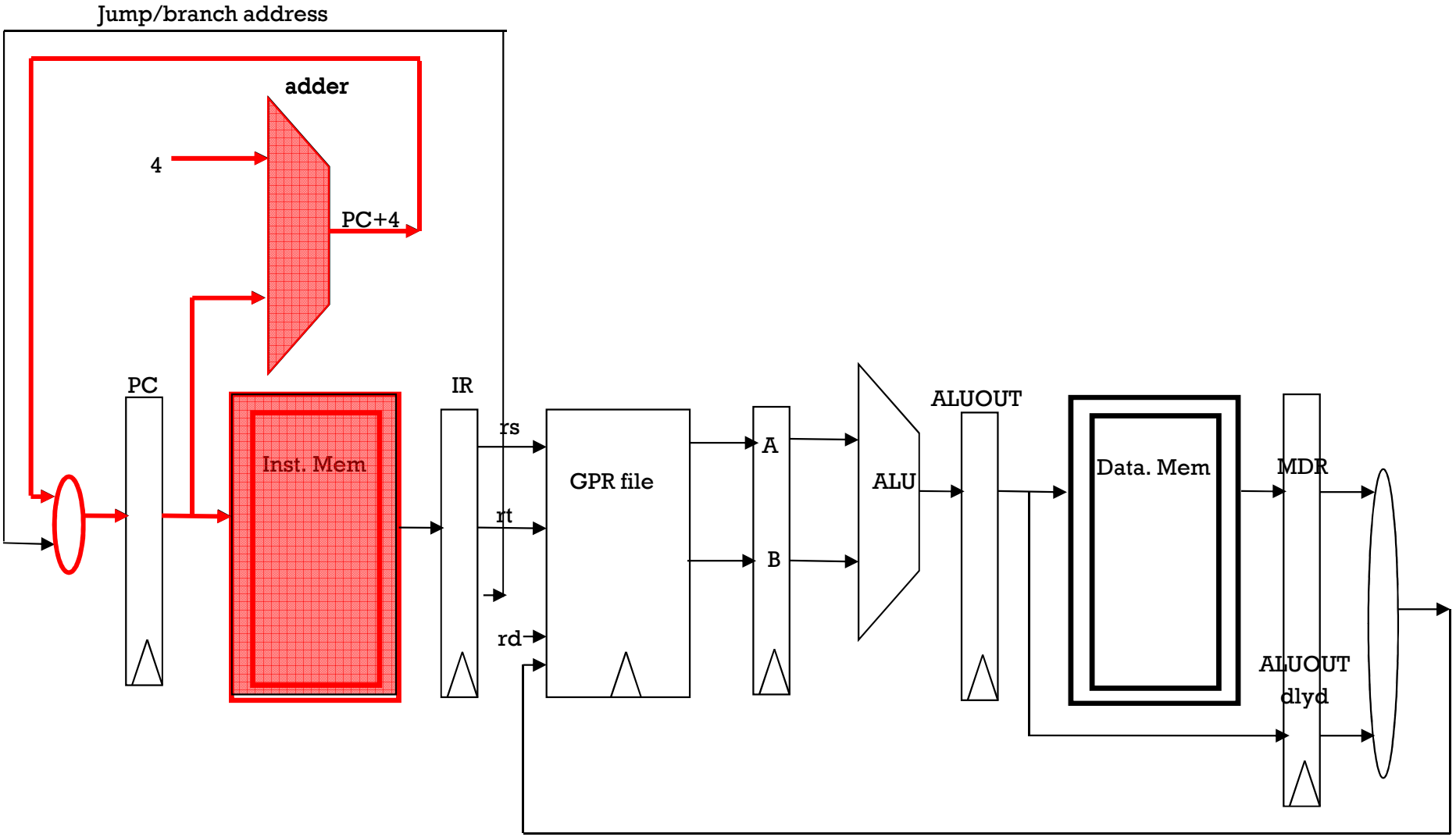
PC = branch adrs
if rs==rt

Pipelined operation

Performing Rtype inst.



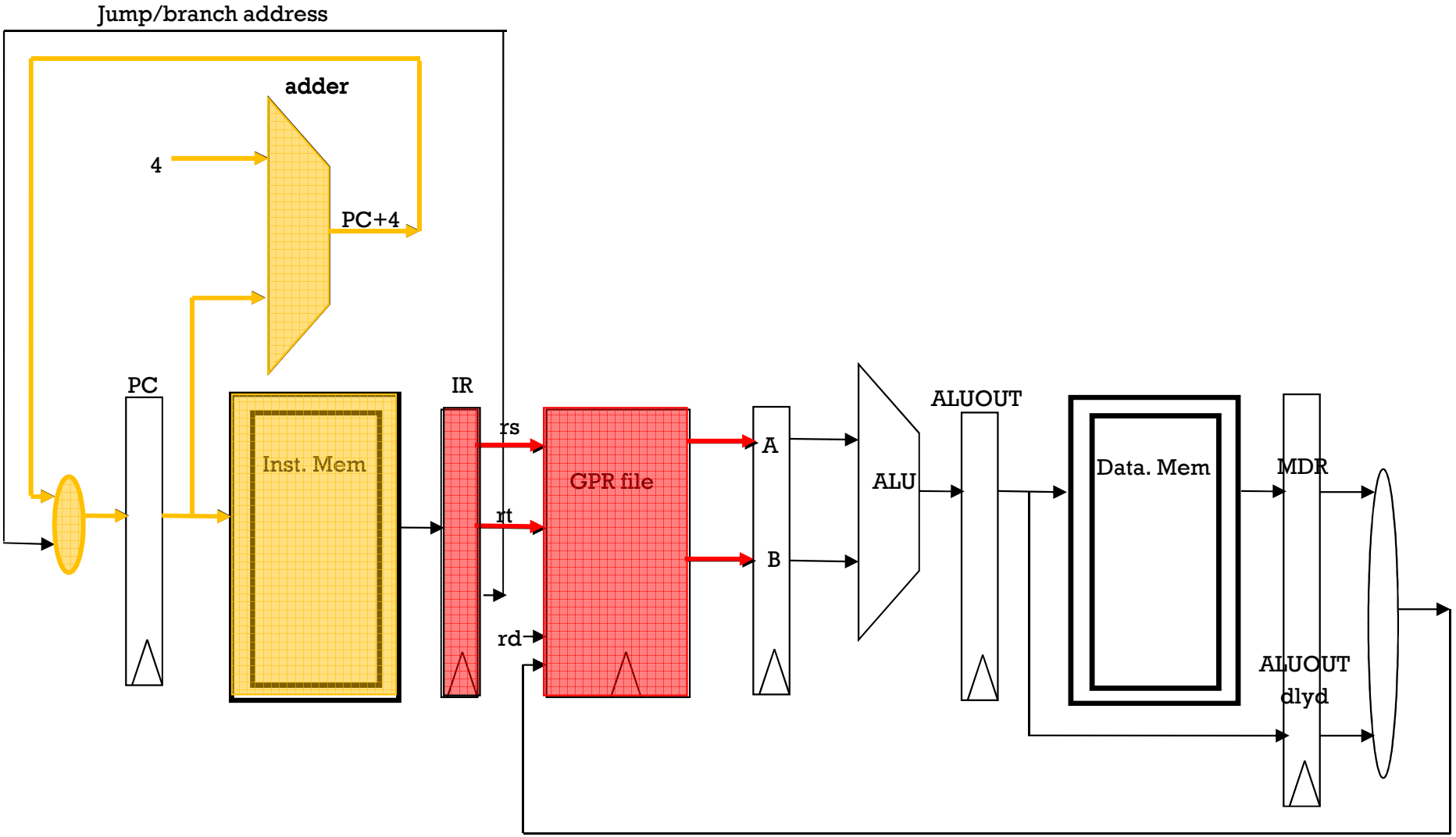
Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

Performing Rtype inst.



IF – Inst. Fetch

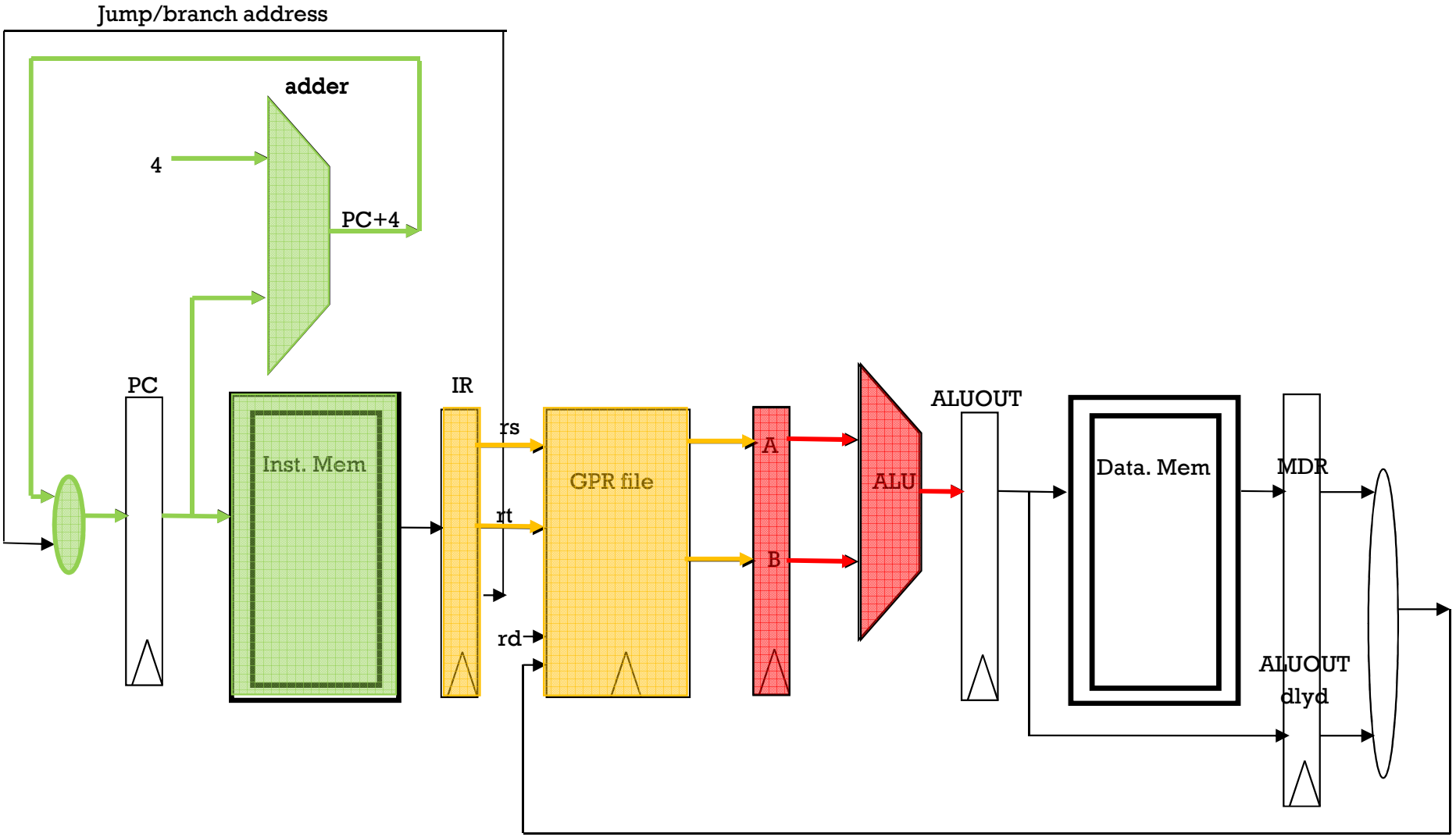
IR = Imem[PC]
PC = PC+4

ID – Inst. Decode

A = rs , B = rt

(& decode
control signals)

Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

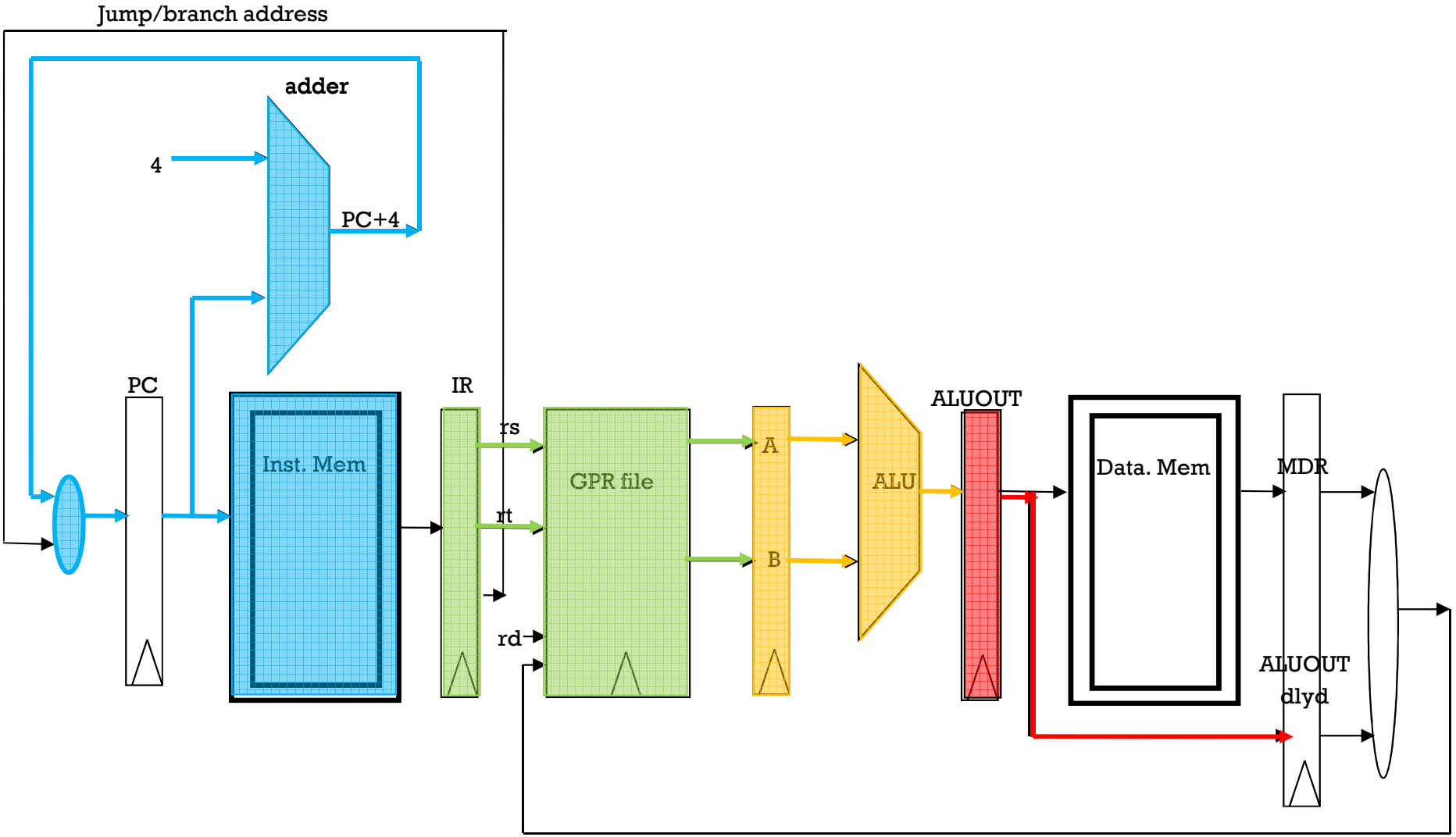
ID – Inst. Dec.
A = rs , B = rt

(& decode control signals)

EX– Execute

ALUOUT = A op B

Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

ID – Inst. Dec.
A = rs , B = rt

(& decode control signals)

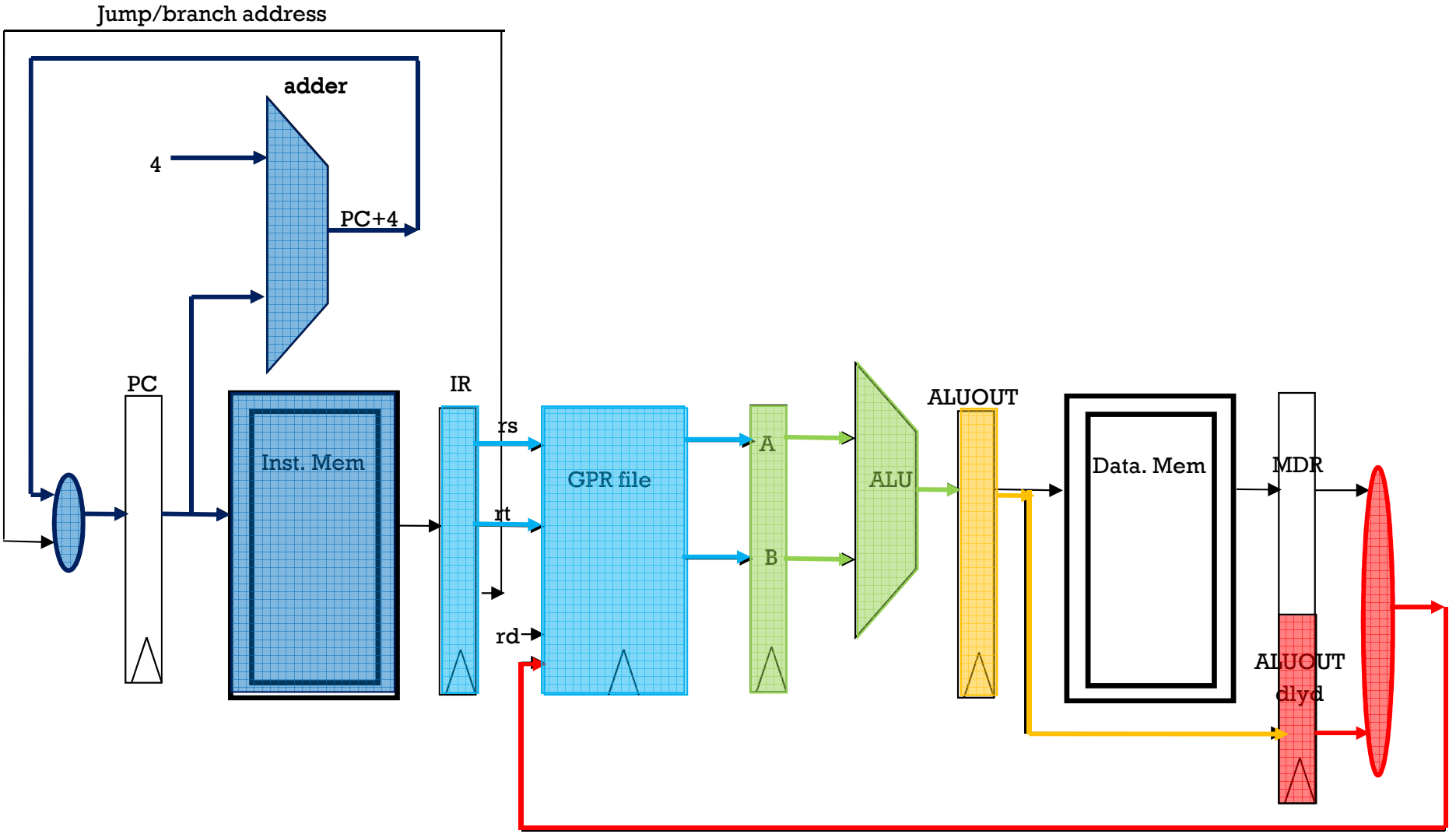
EX– Execute

ALUOUT = A op B

MEM

In Rtype – wait 1 ck

Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

ID – Inst. Dec.
A = rs , B = rt

(& decode control signals)

EX– Execute

ALUOUT = A op B

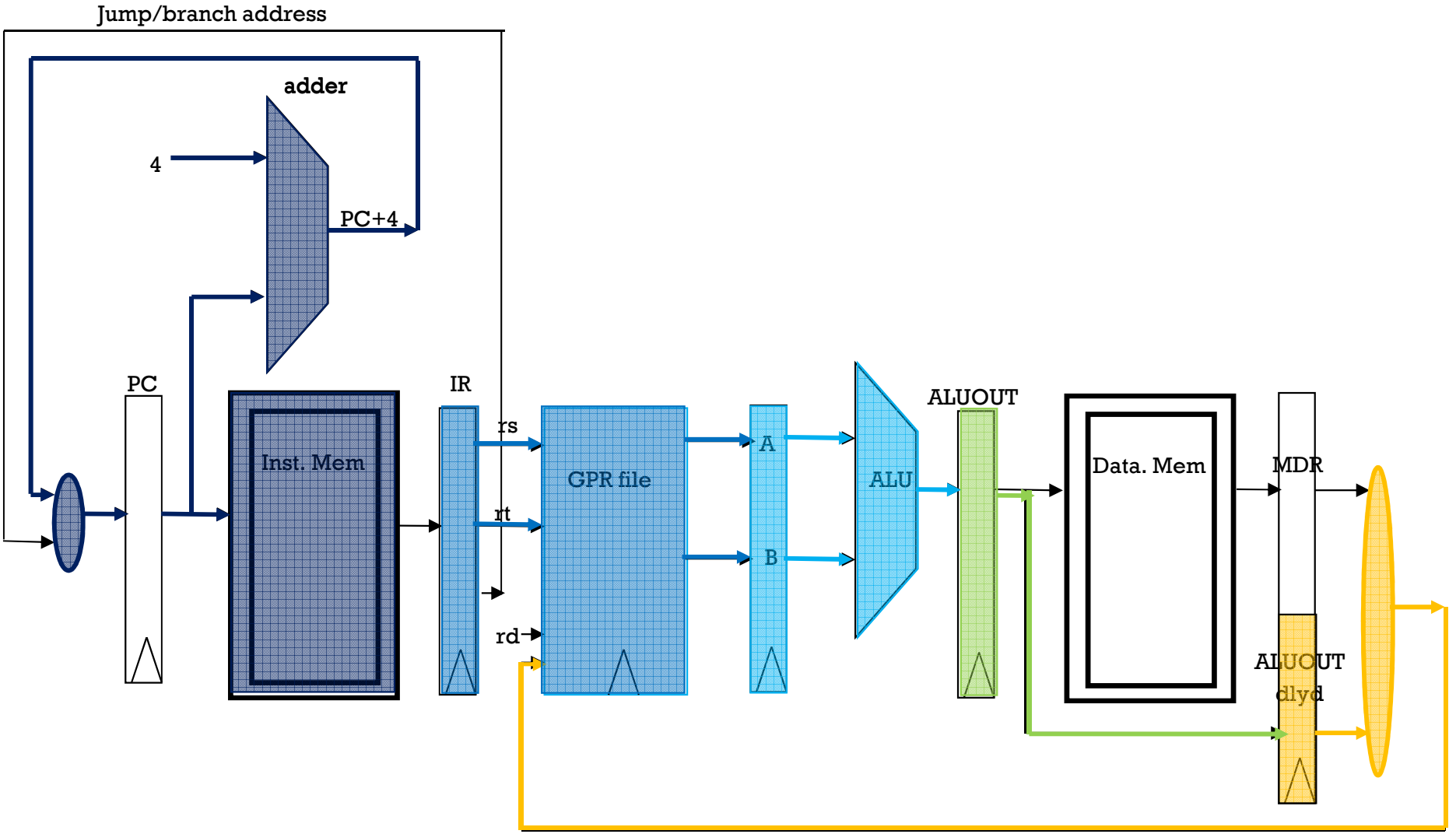
MEM

In Rtype – wait 1 ck

WB – write back

Rd = ALUOUT

Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

ID – Inst. Dec.
A = rs , B = rt

(& decode control signals)

EX– Execute

ALUOUT = A op B

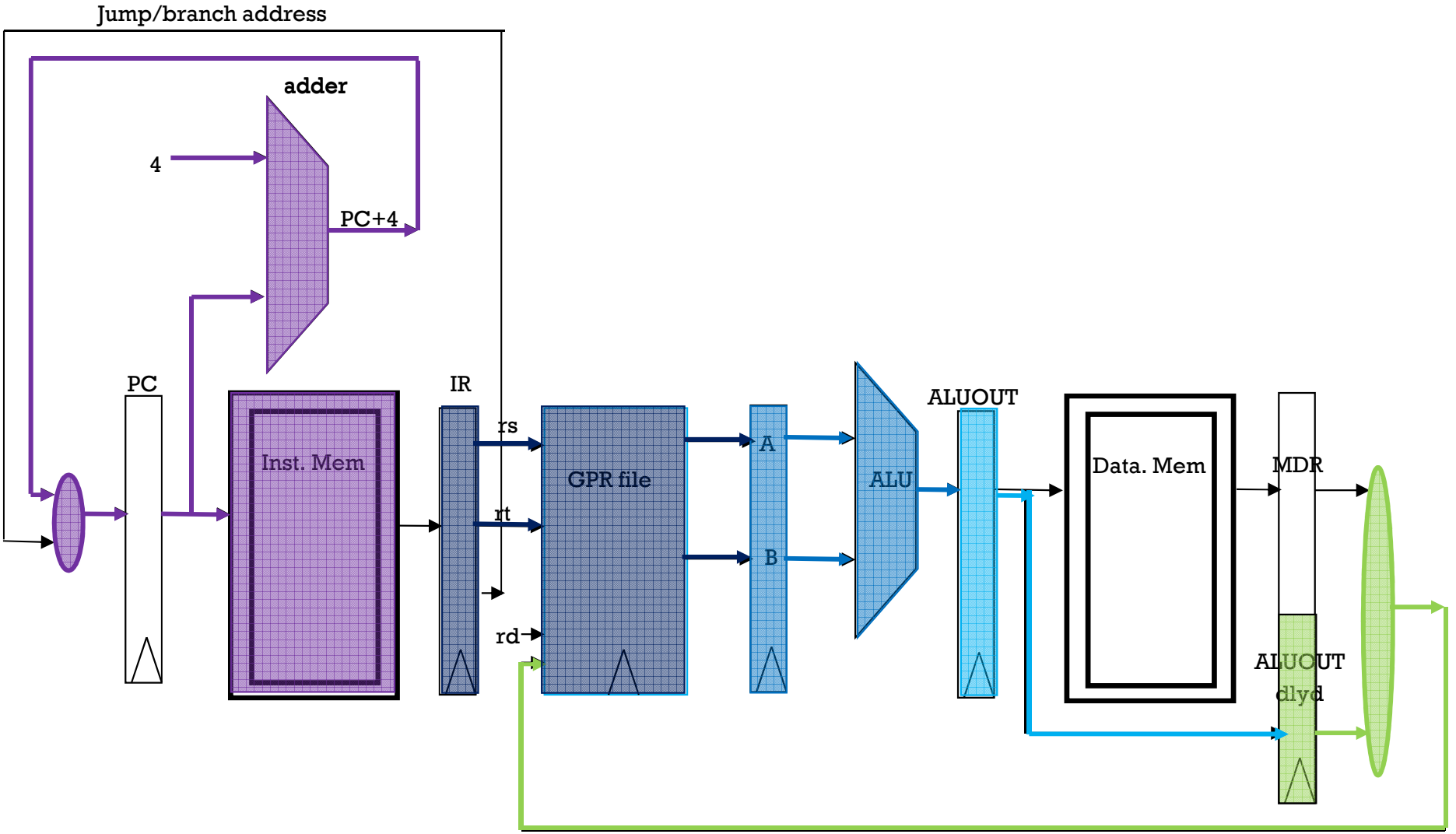
MEM

In Rtype – wait 1 ck

WB – write back

Rd = ALUOUT

Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

ID – Inst. Dec.
A = rs , B = rt

(& decode control signals)

EX– Execute

ALUOUT = A op B

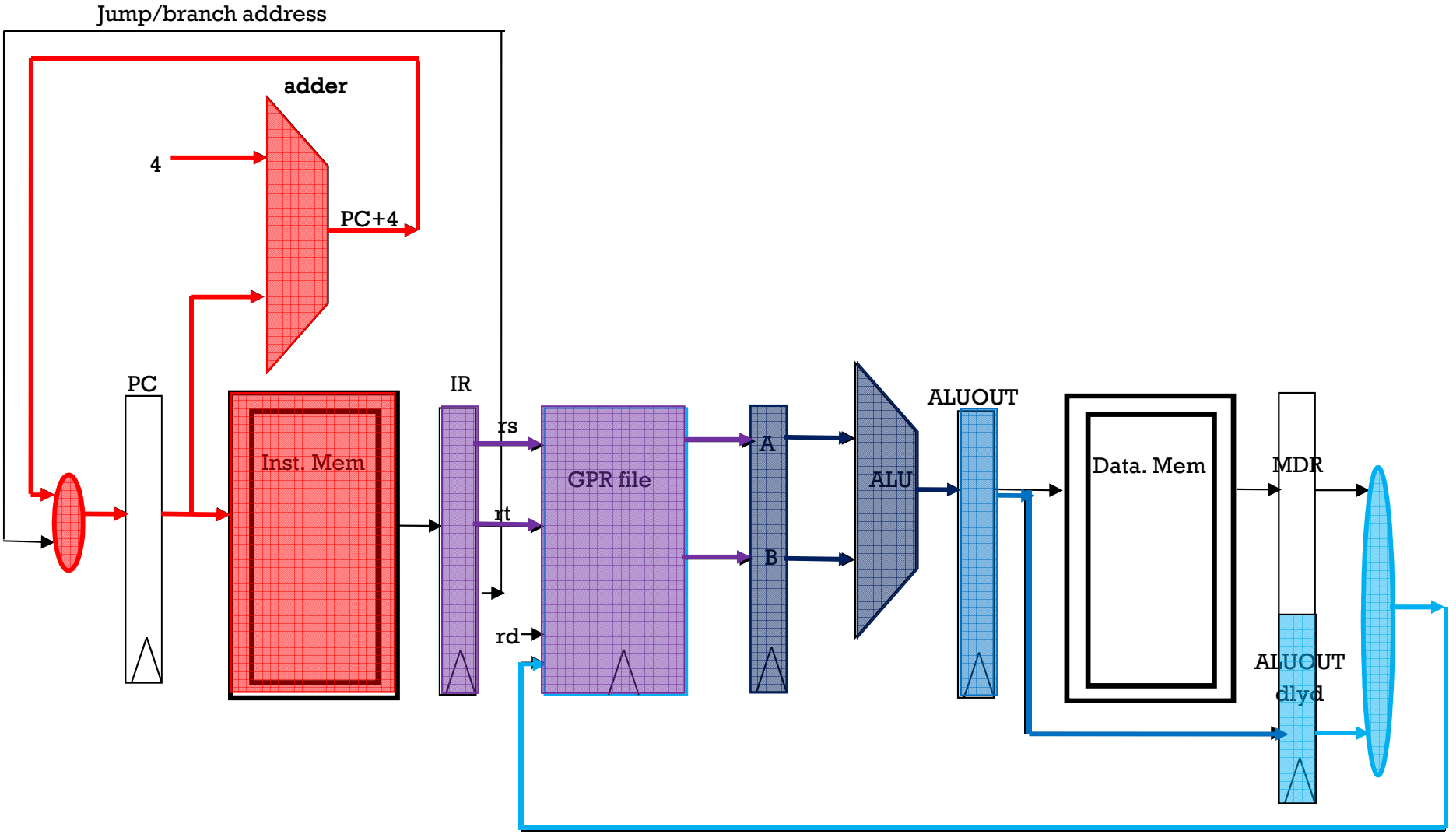
MEM

In Rtype – wait 1 ck

WB – write back

Rd = ALUOUT

Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

ID – Inst. Dec.
A = rs , B = rt

(& decode control signals)

EX– Execute

ALUOUT = A op B

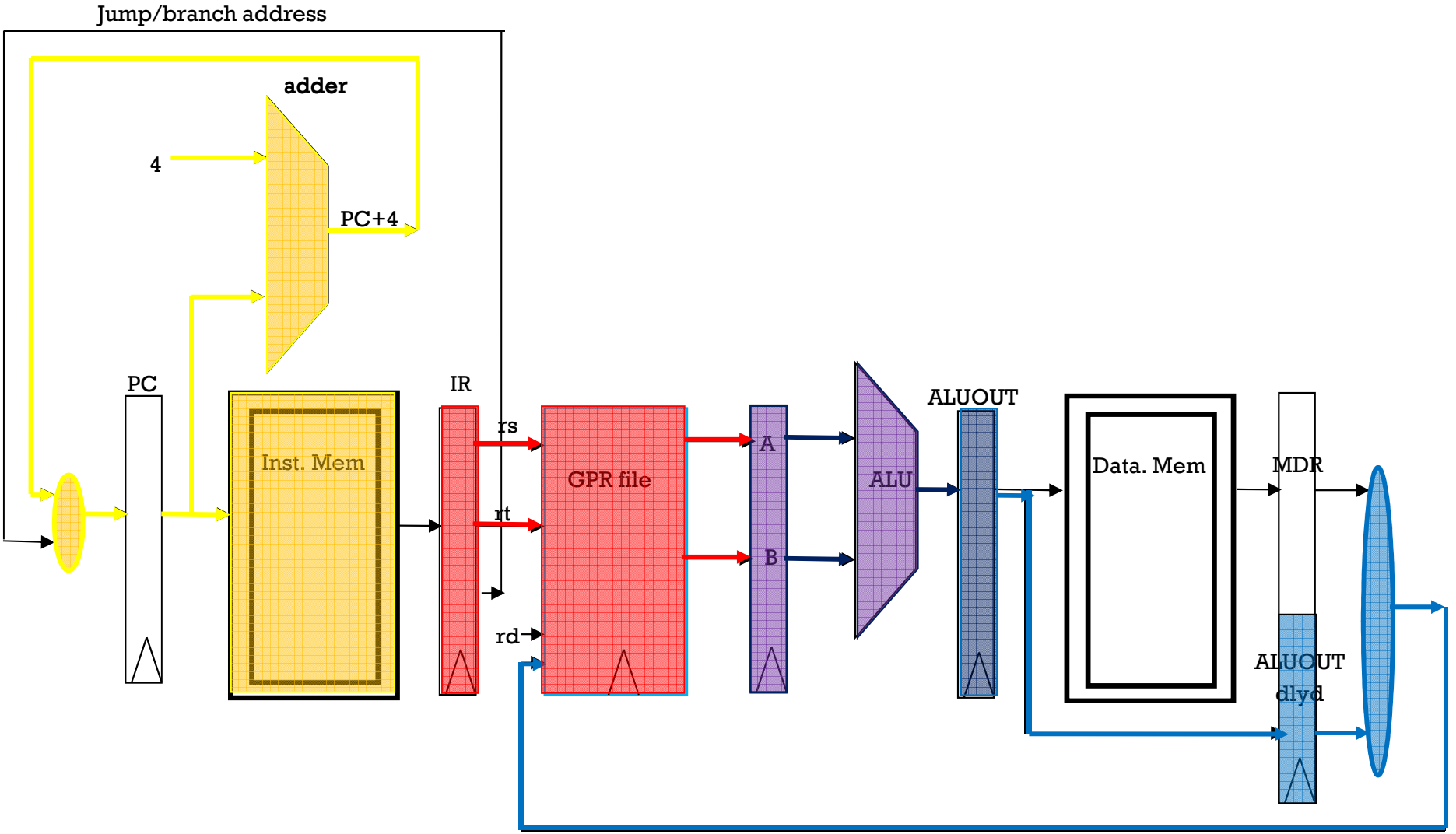
MEM

In Rtype – wait 1 ck

WB – write back

Rd = ALUOUT

Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

ID – Inst. Dec.
A = rs , B = rt

(& decode control signals)

EX– Execute

ALUOUT = A op B

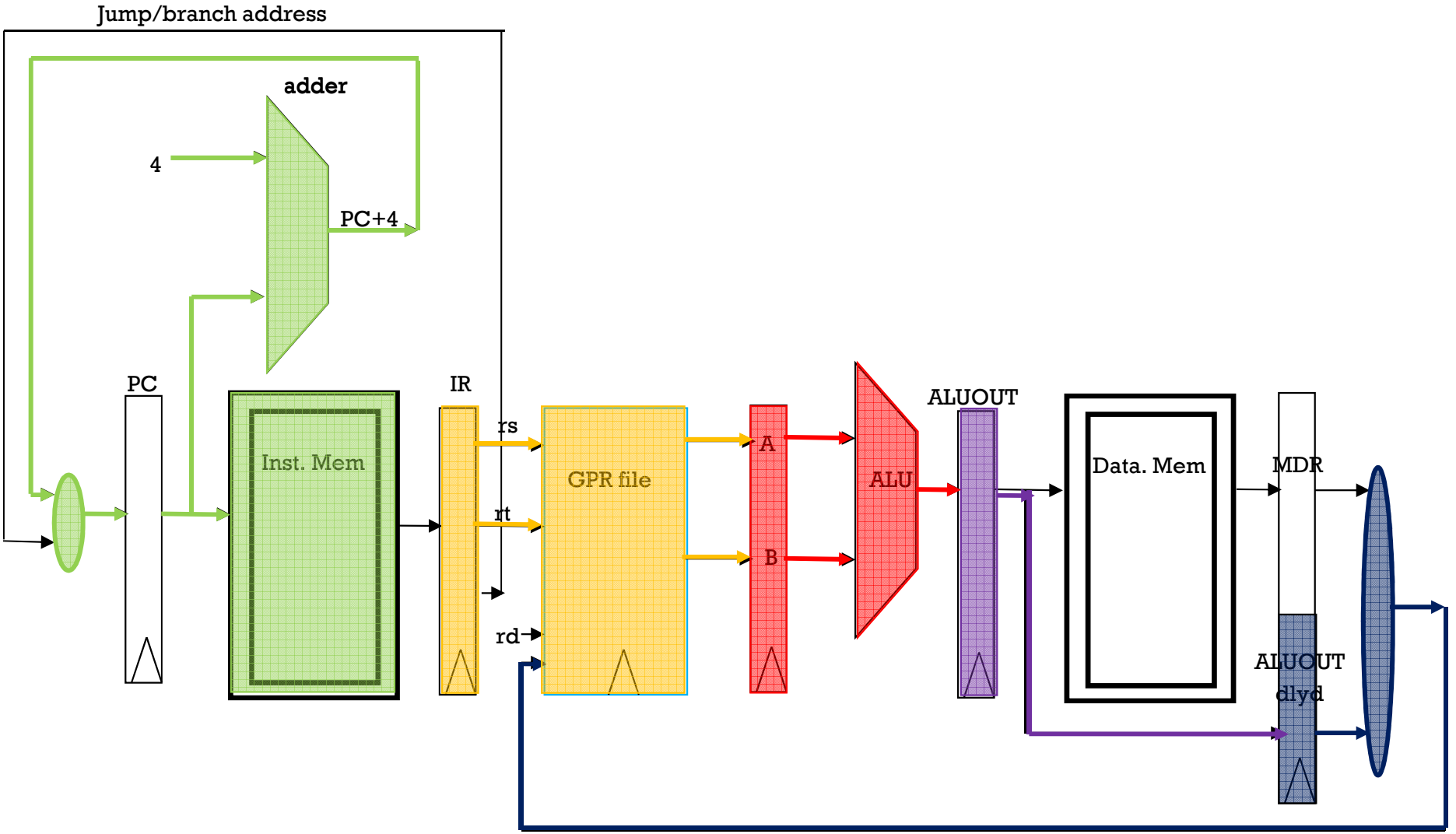
MEM

In Rtype – wait 1 ck

WB – write back

Rd = ALUOUT

Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

ID – Inst. Dec.
A = rs , B = rt

(& decode control signals)

EX– Execute

ALUOUT = A op B

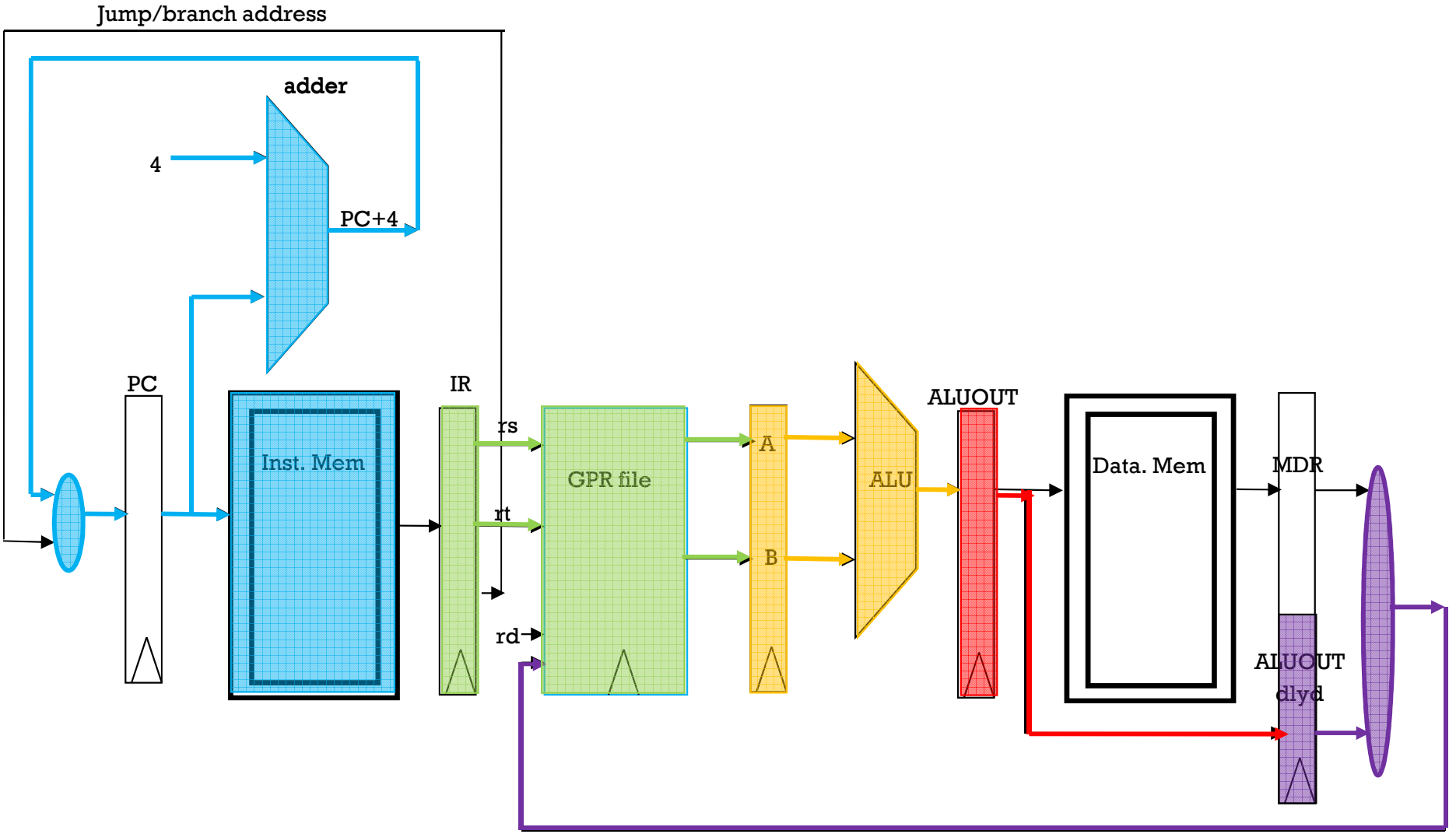
MEM

In Rtype – wait 1 ck

WB – write back

Rd = ALUOUT

Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

ID – Inst. Dec.
A = rs , B = rt

(& decode control signals)

EX– Execute

ALUOUT = A op B

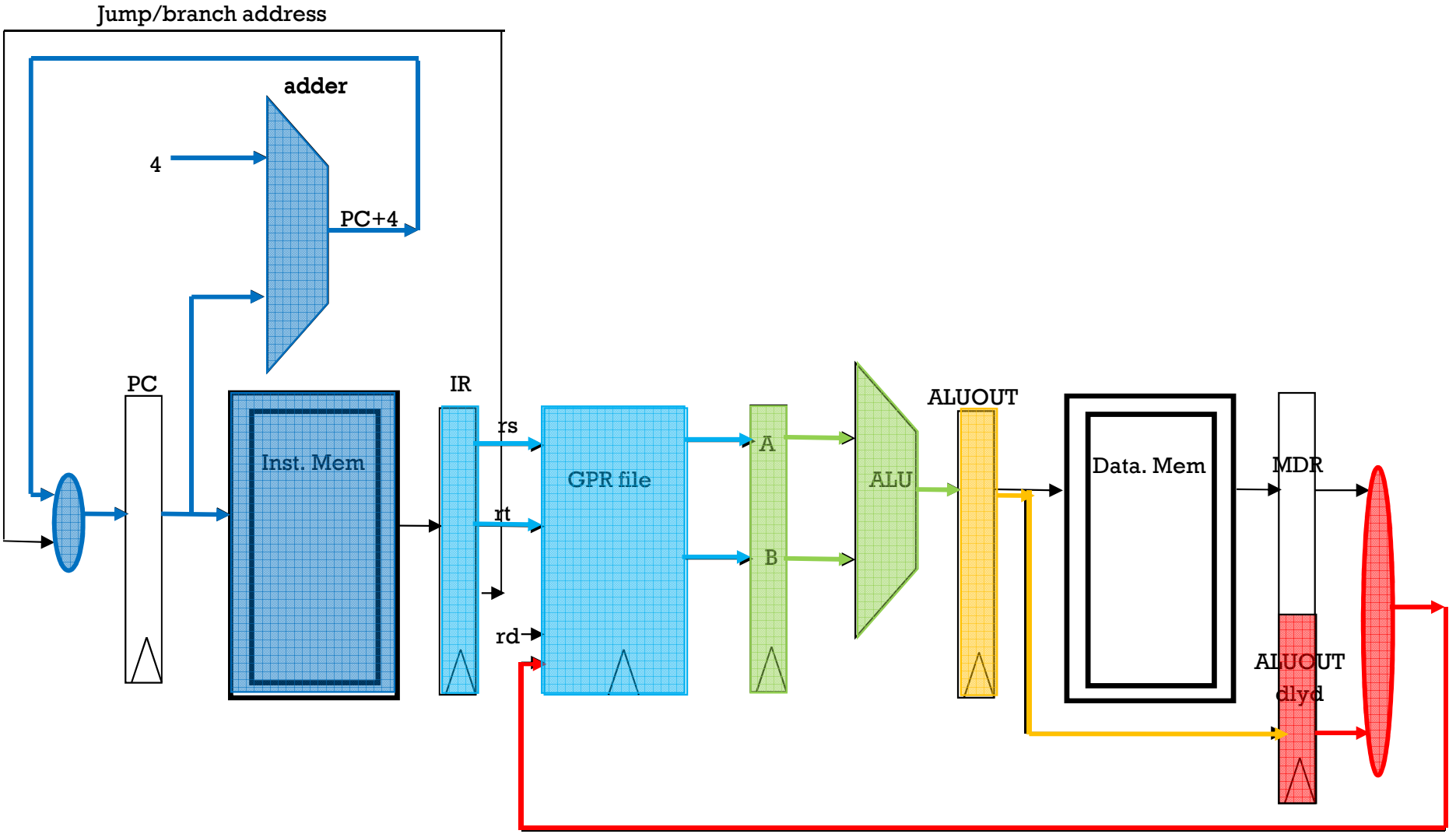
MEM

In Rtype – wait 1 ck

WB – write back

Rd = ALUOUT

Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

ID – Inst. Dec.
A = rs , B = rt

(& decode control signals)

EX– Execute

ALUOUT = A op B

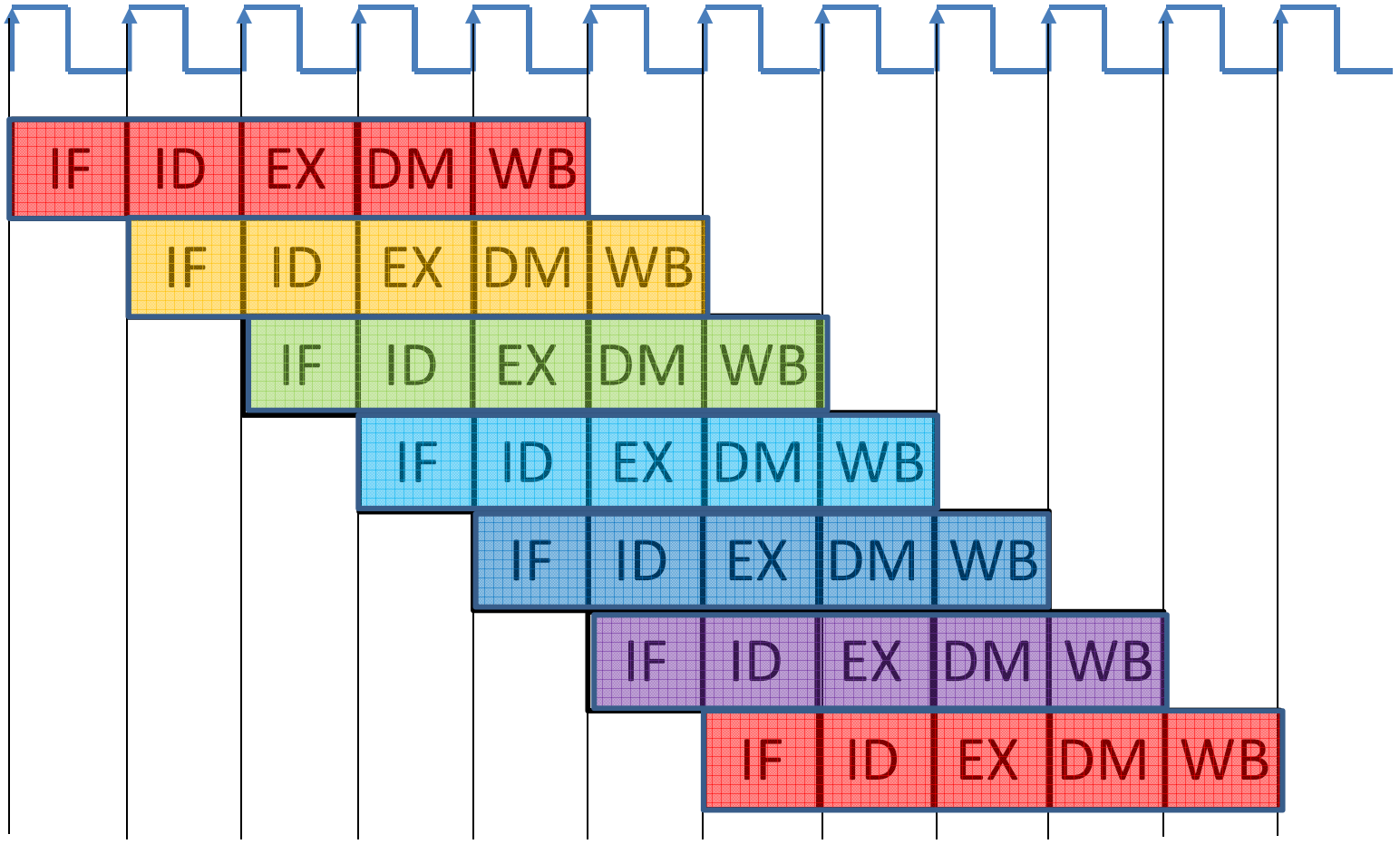
MEM

In Rtype – wait 1 ck

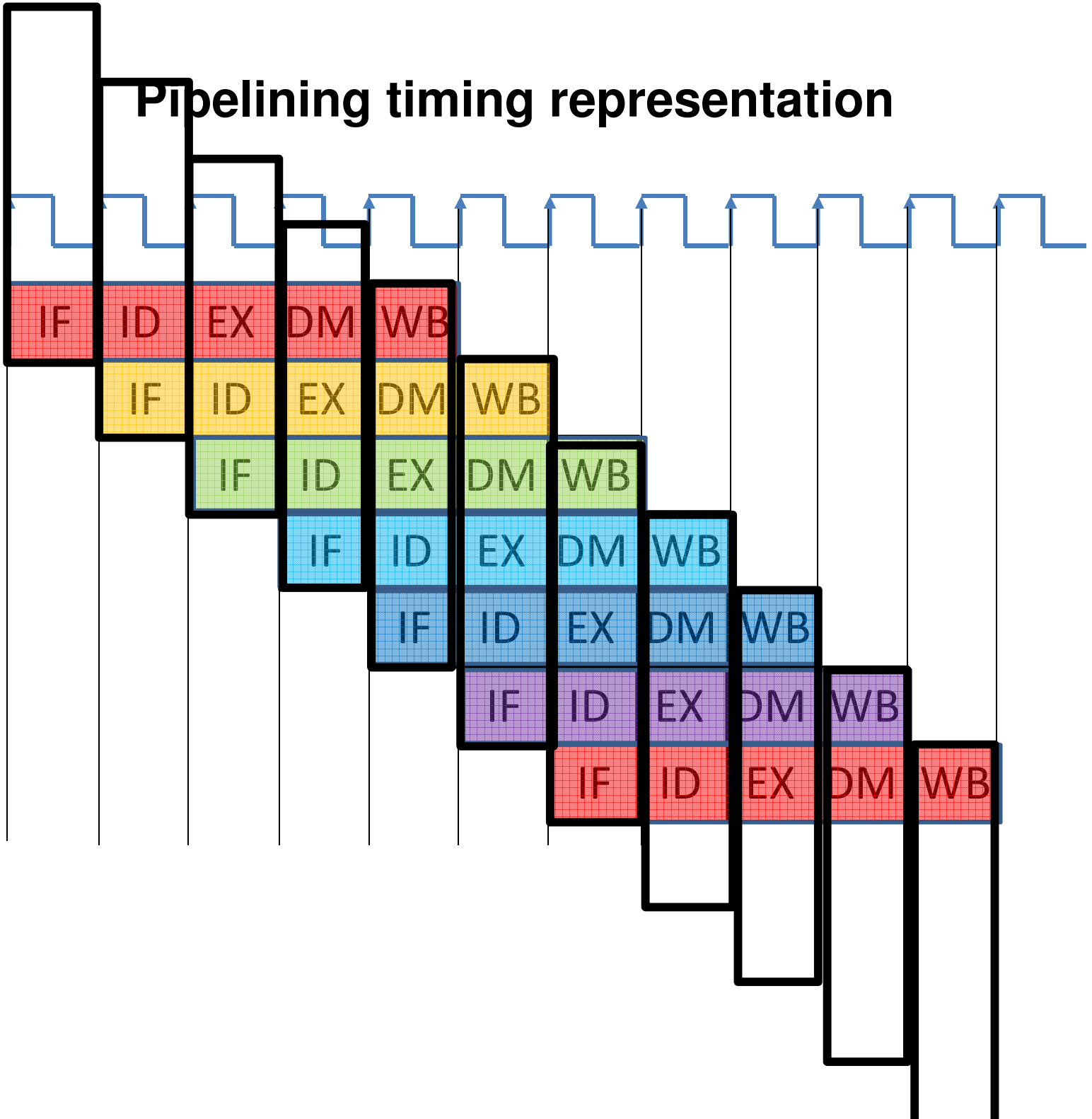
WB – write back

Rd = ALUOUT

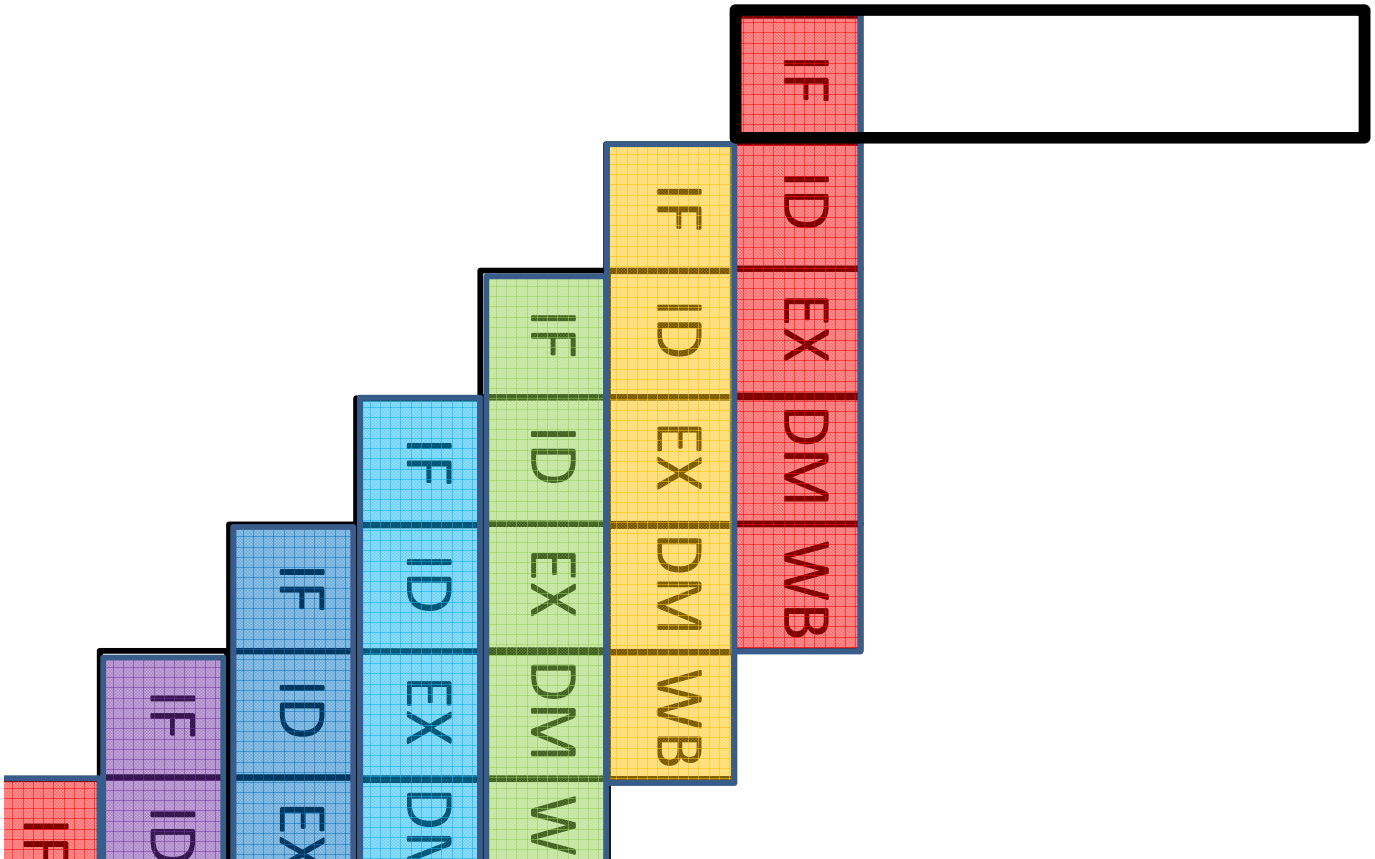
Pipelining timing representation



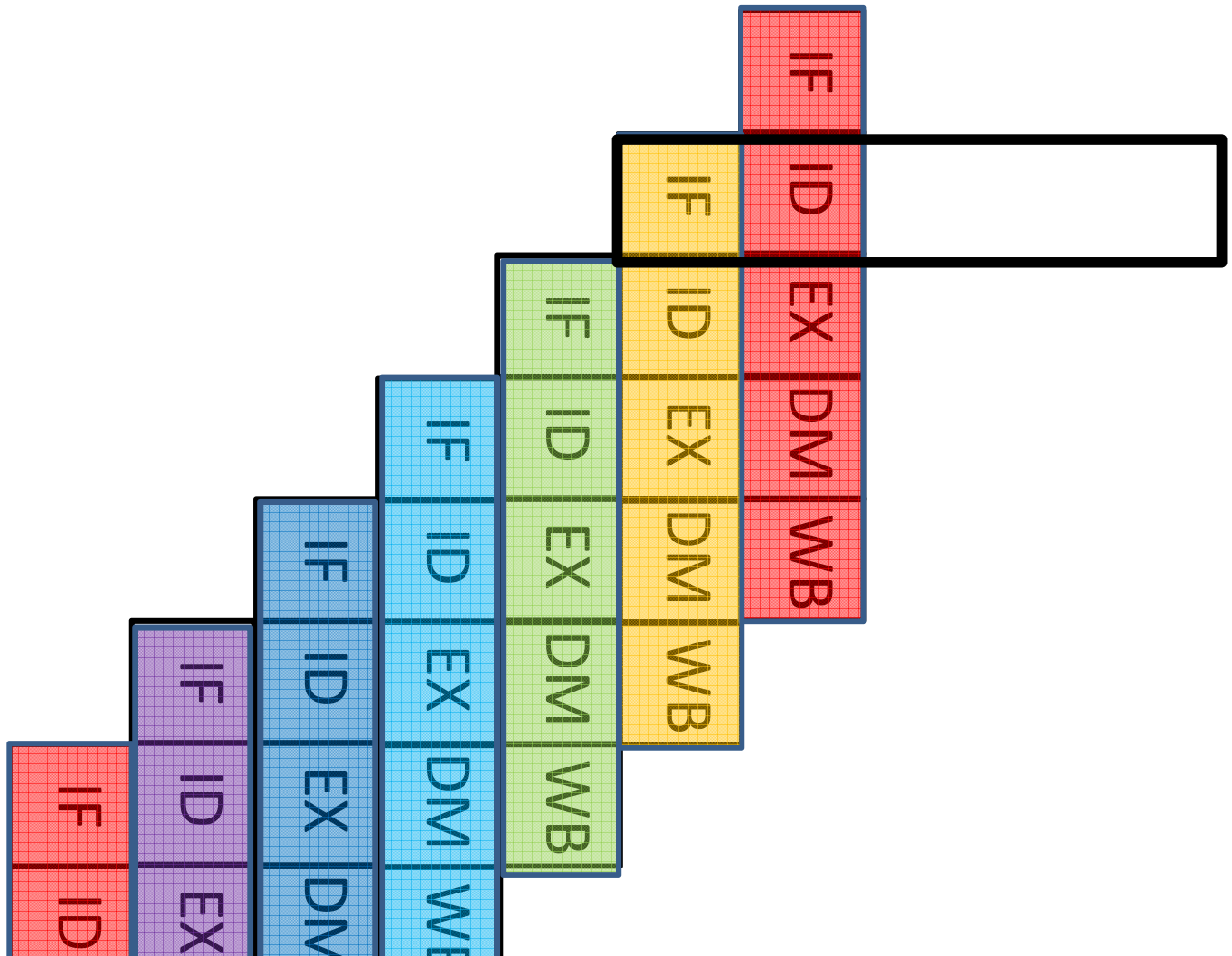
Pipelining timing representation



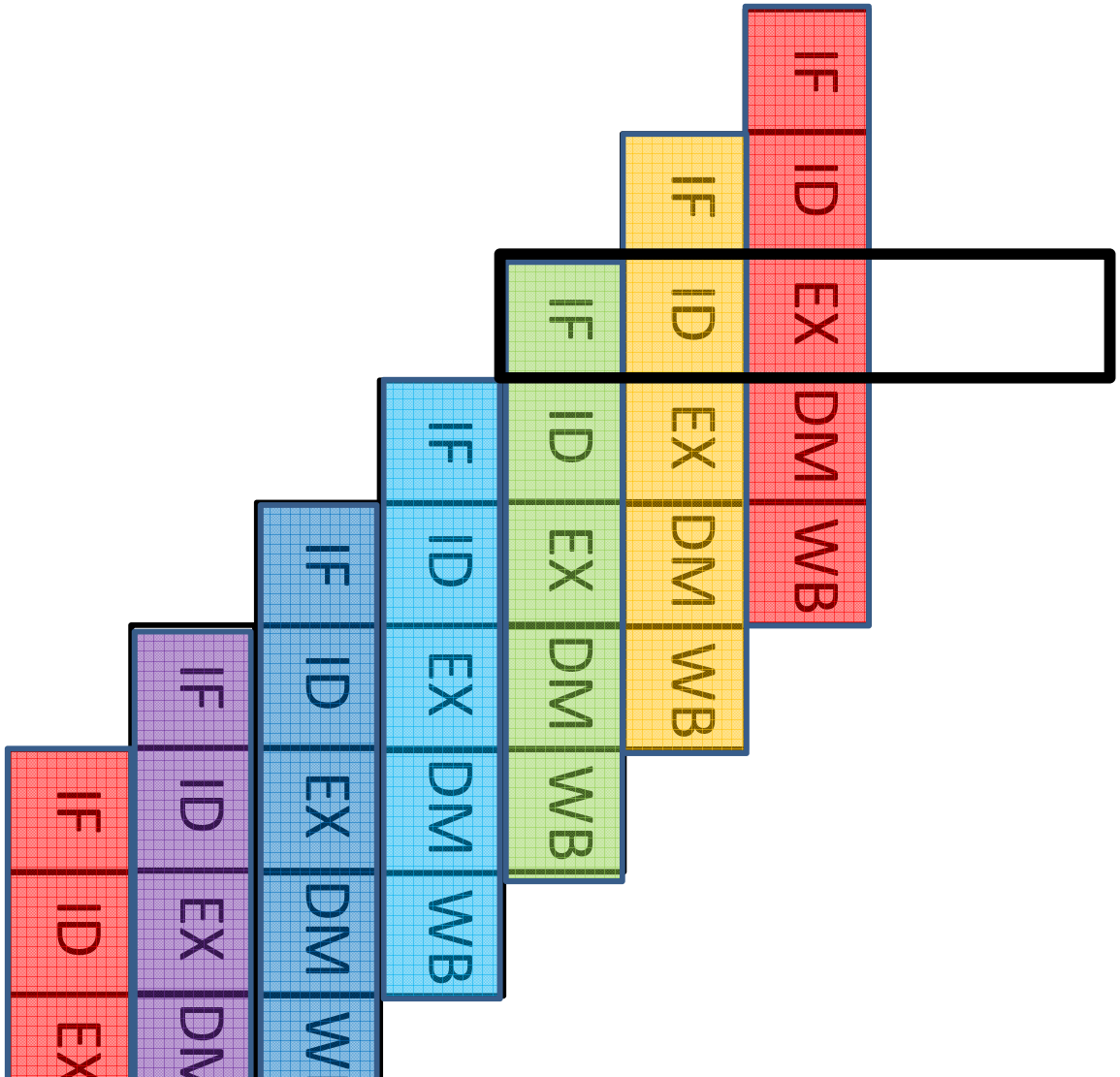
If we rotate that, we see the same picture we saw in the previous slides.



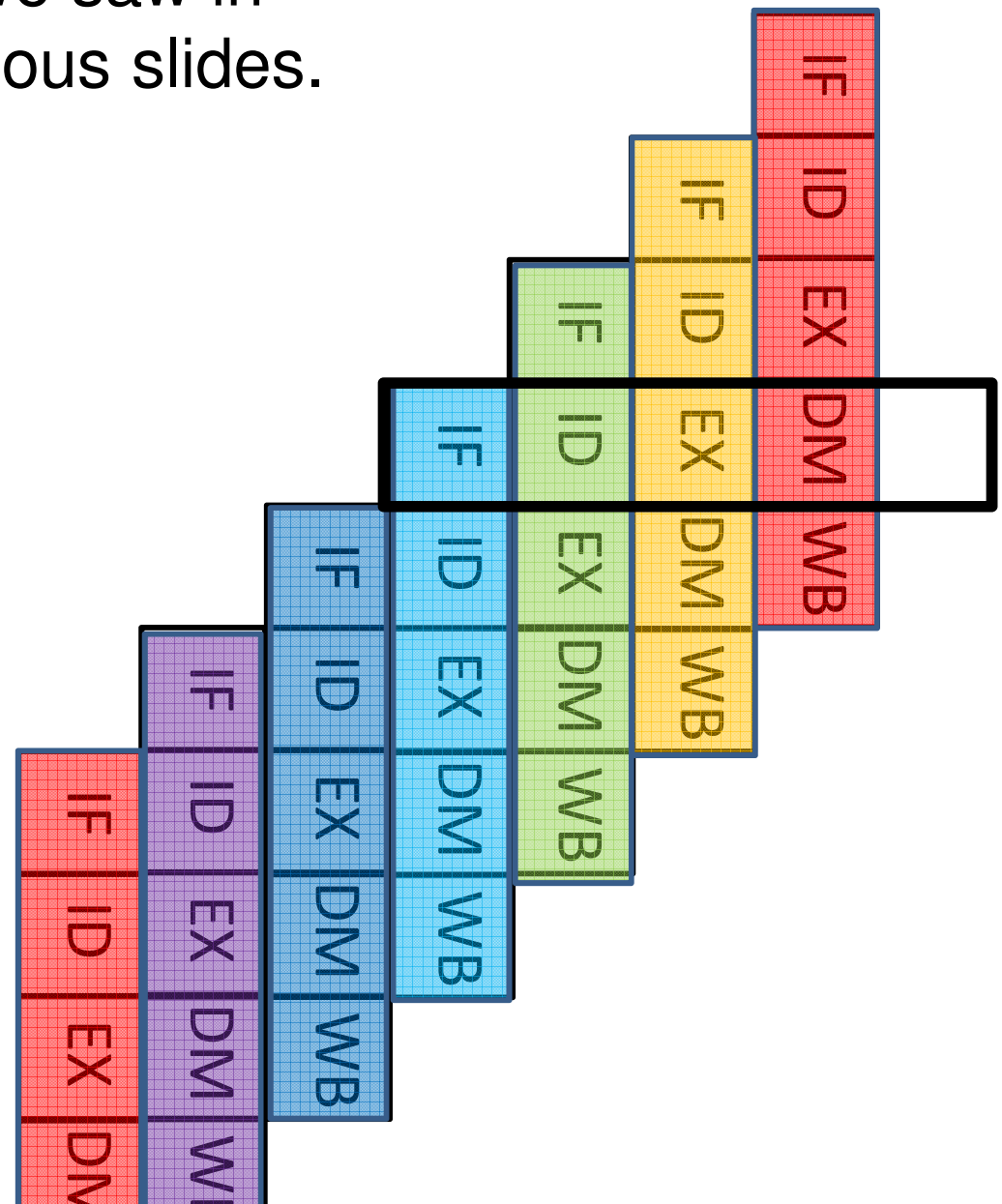
If we rotate that, we see the same picture we saw in the previous slides.



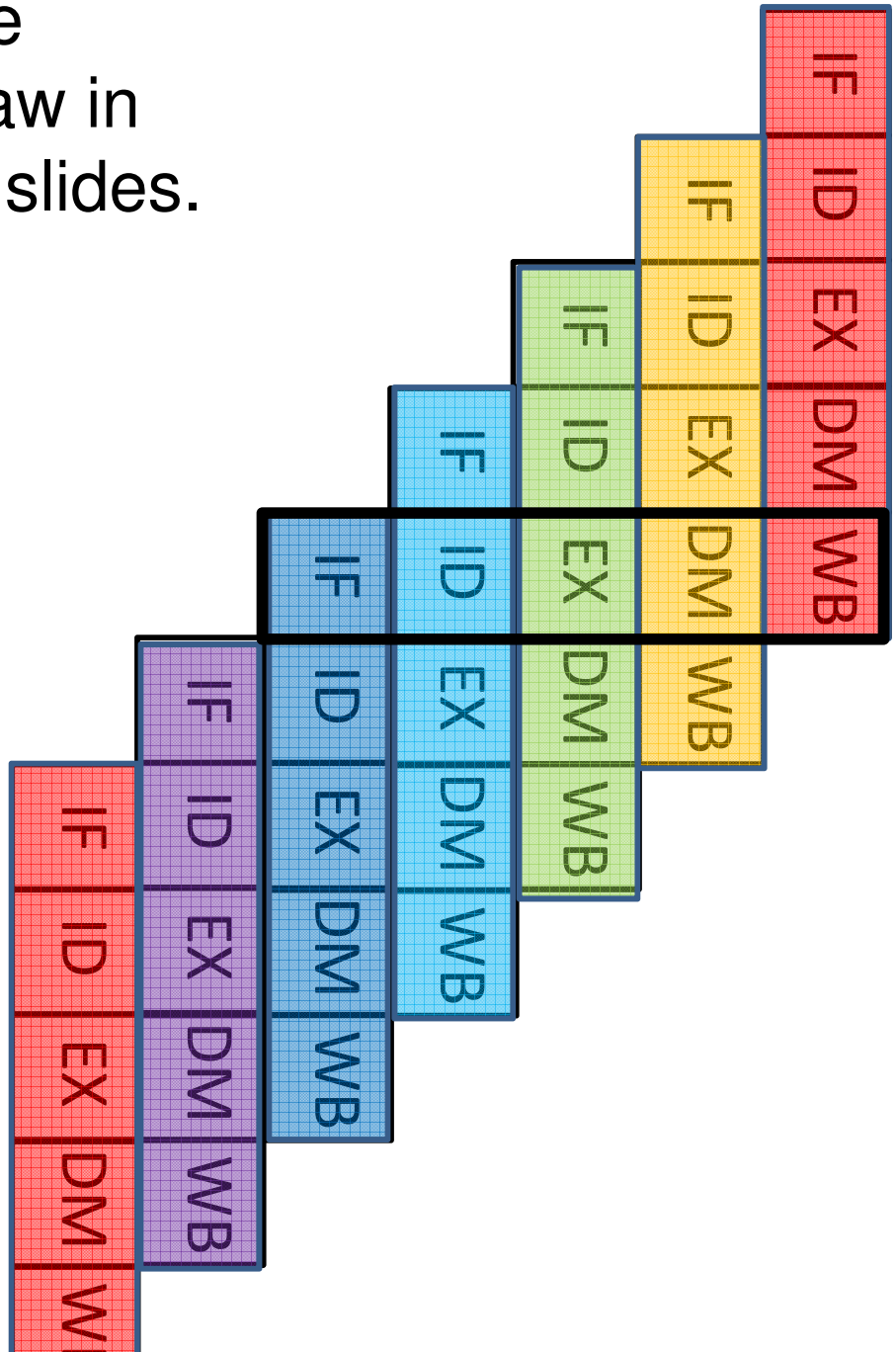
If we rotate that, we see the same picture we saw in the previous slides.



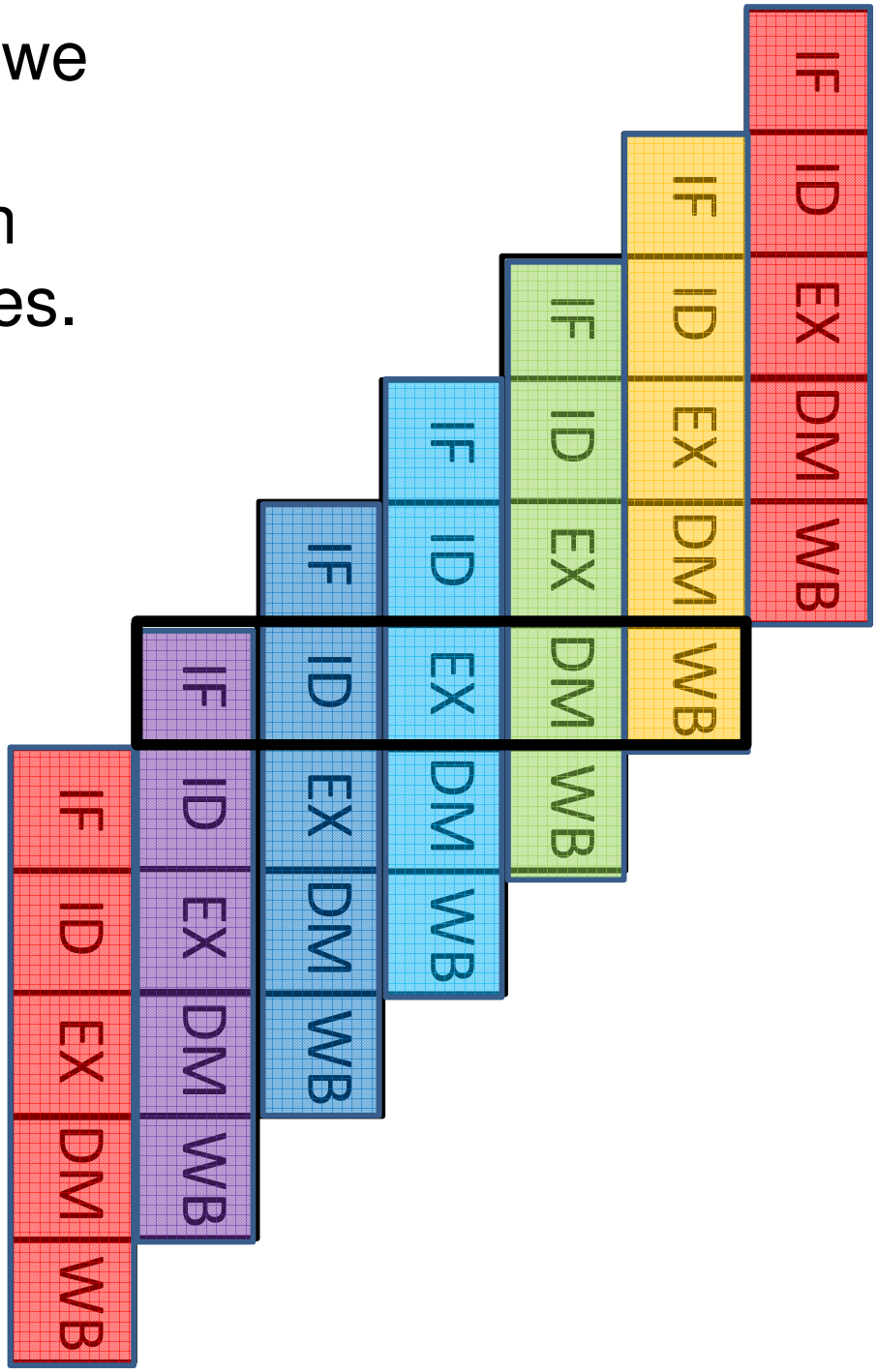
If we rotate that, we see the same picture we saw in the previous slides.



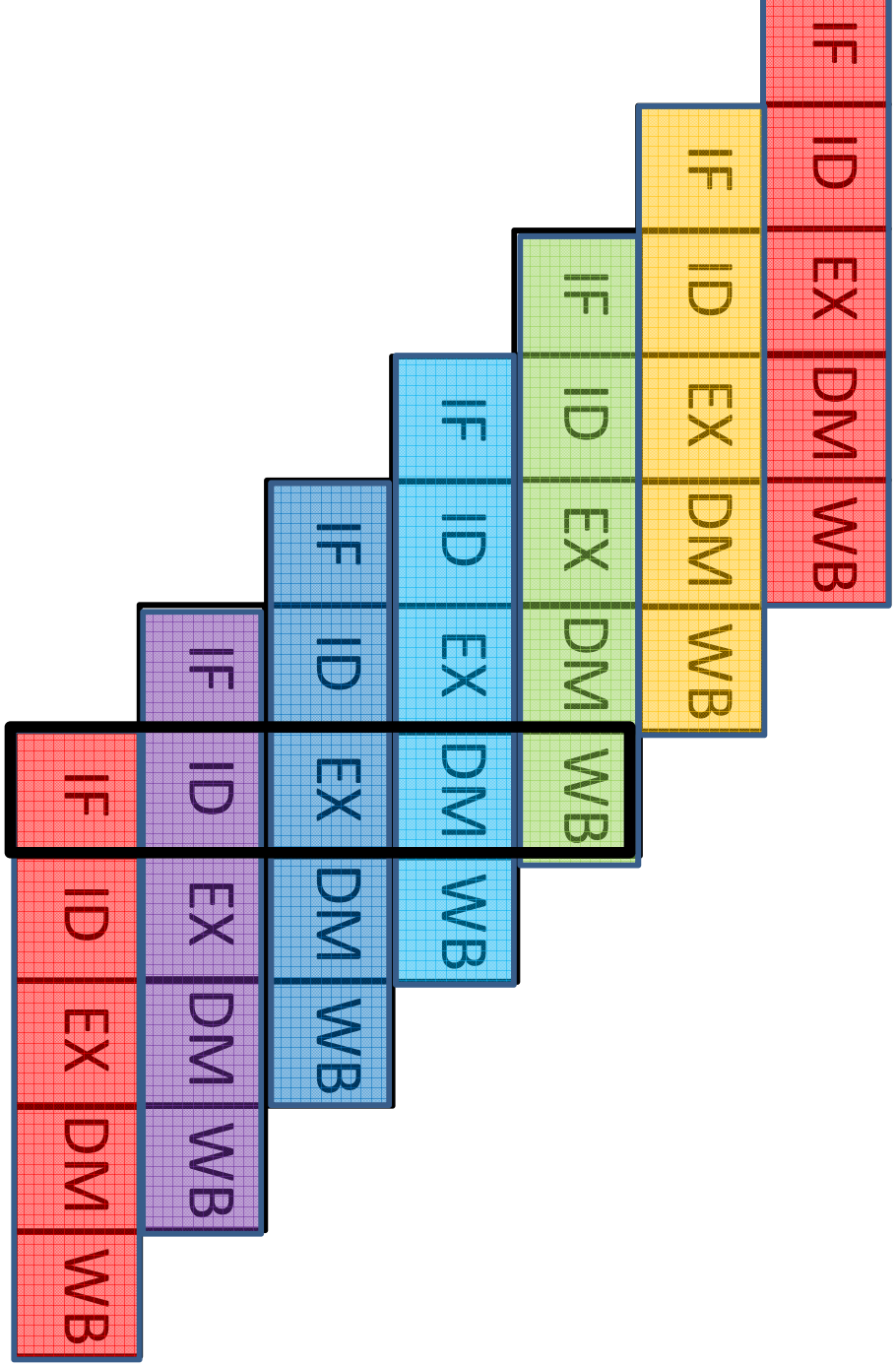
If we rotate that, we see the same picture we saw in the previous slides.



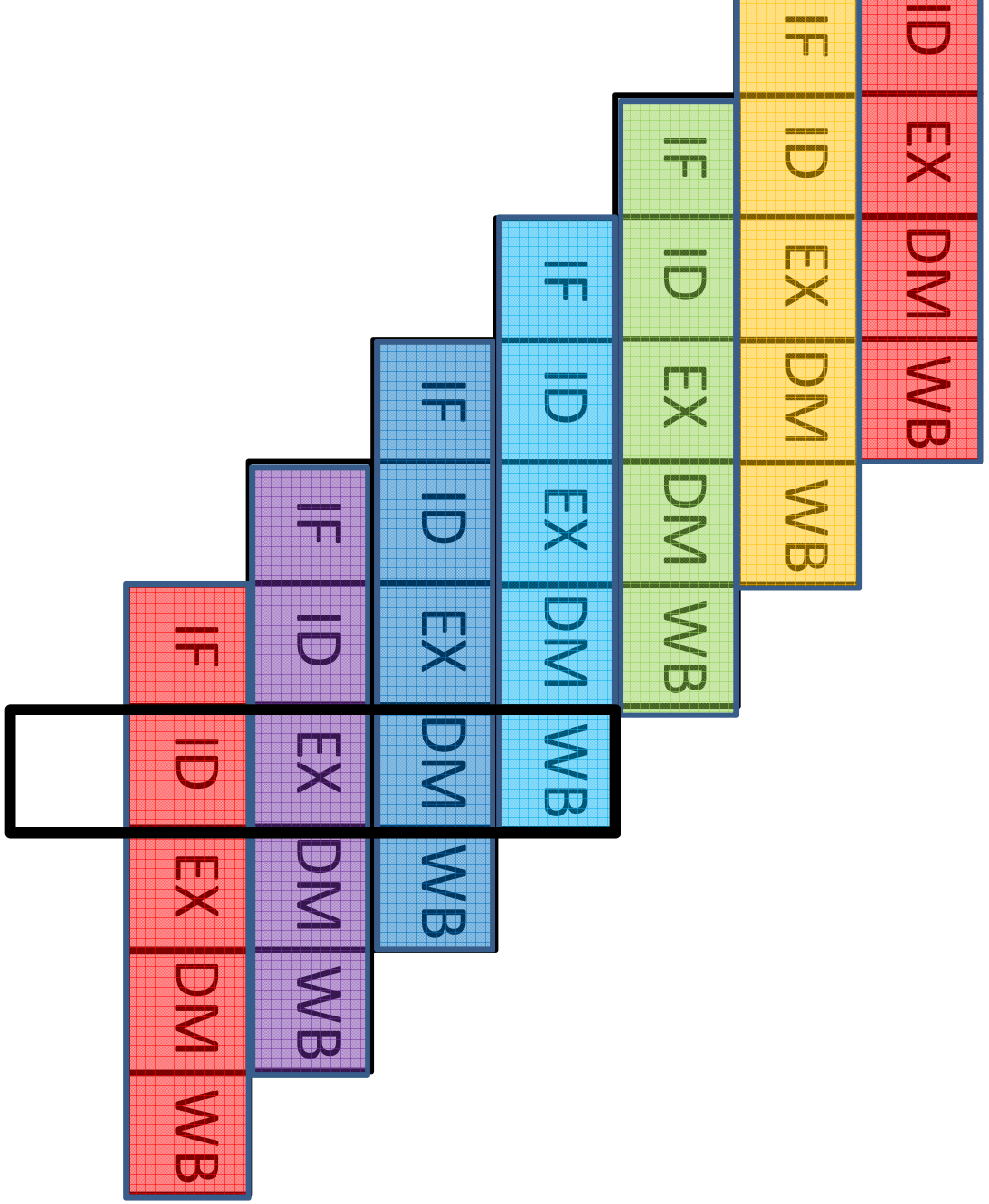
If we rotate that, we see the same picture we saw in the previous slides.



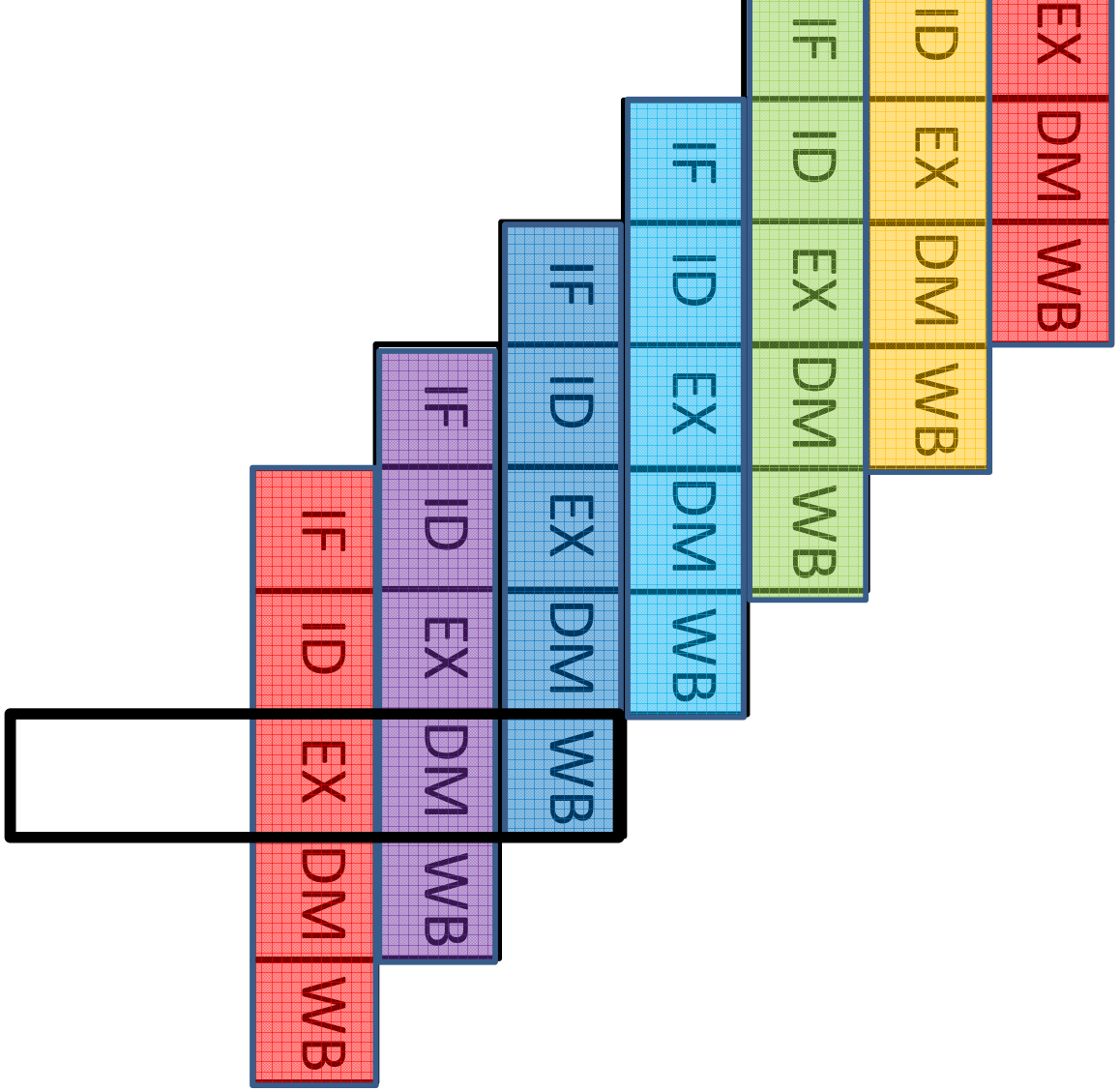
If we rotate that, we see the same picture we saw in the previous slides.



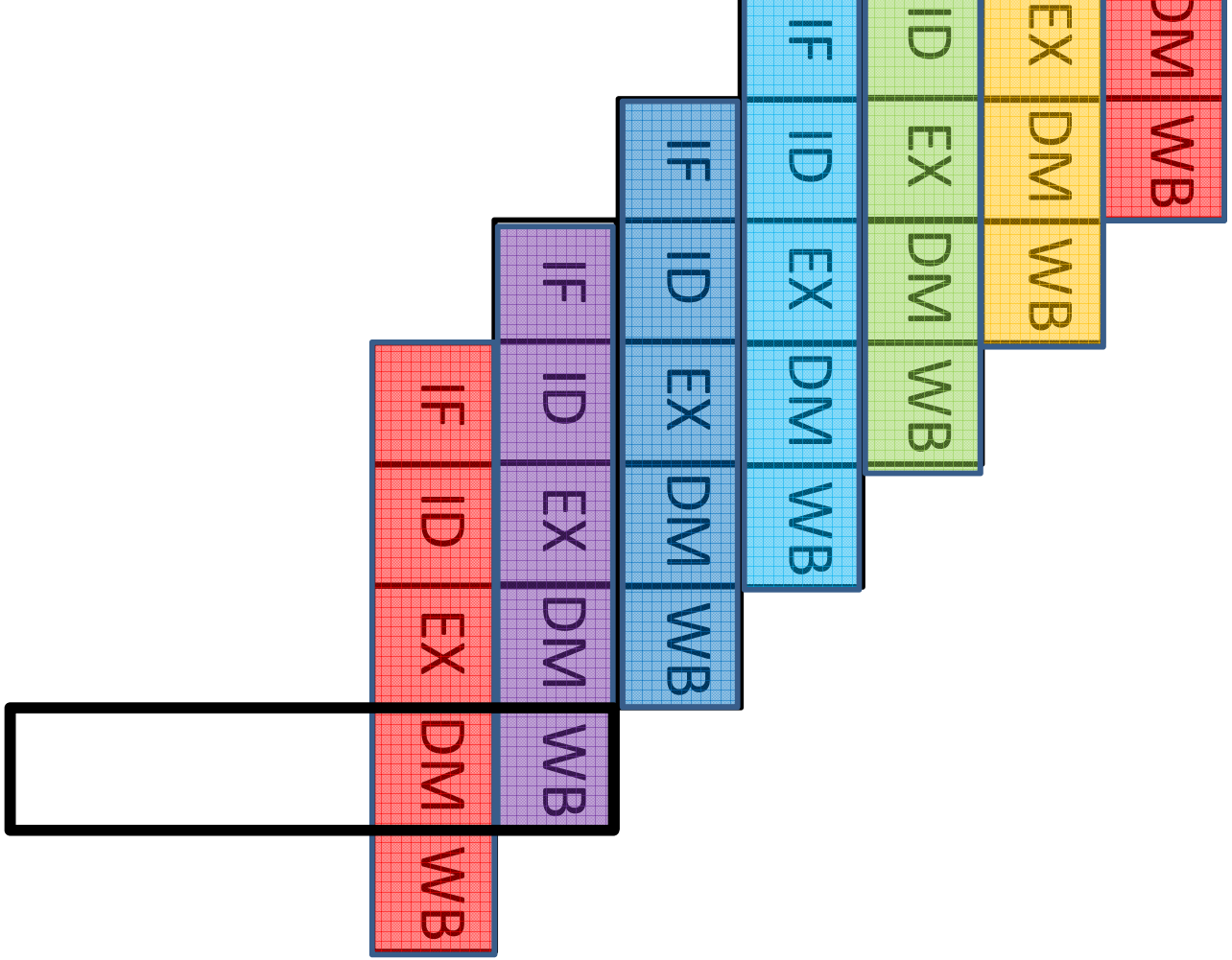
If we rotate that, we see the same picture we saw in the previous slides.



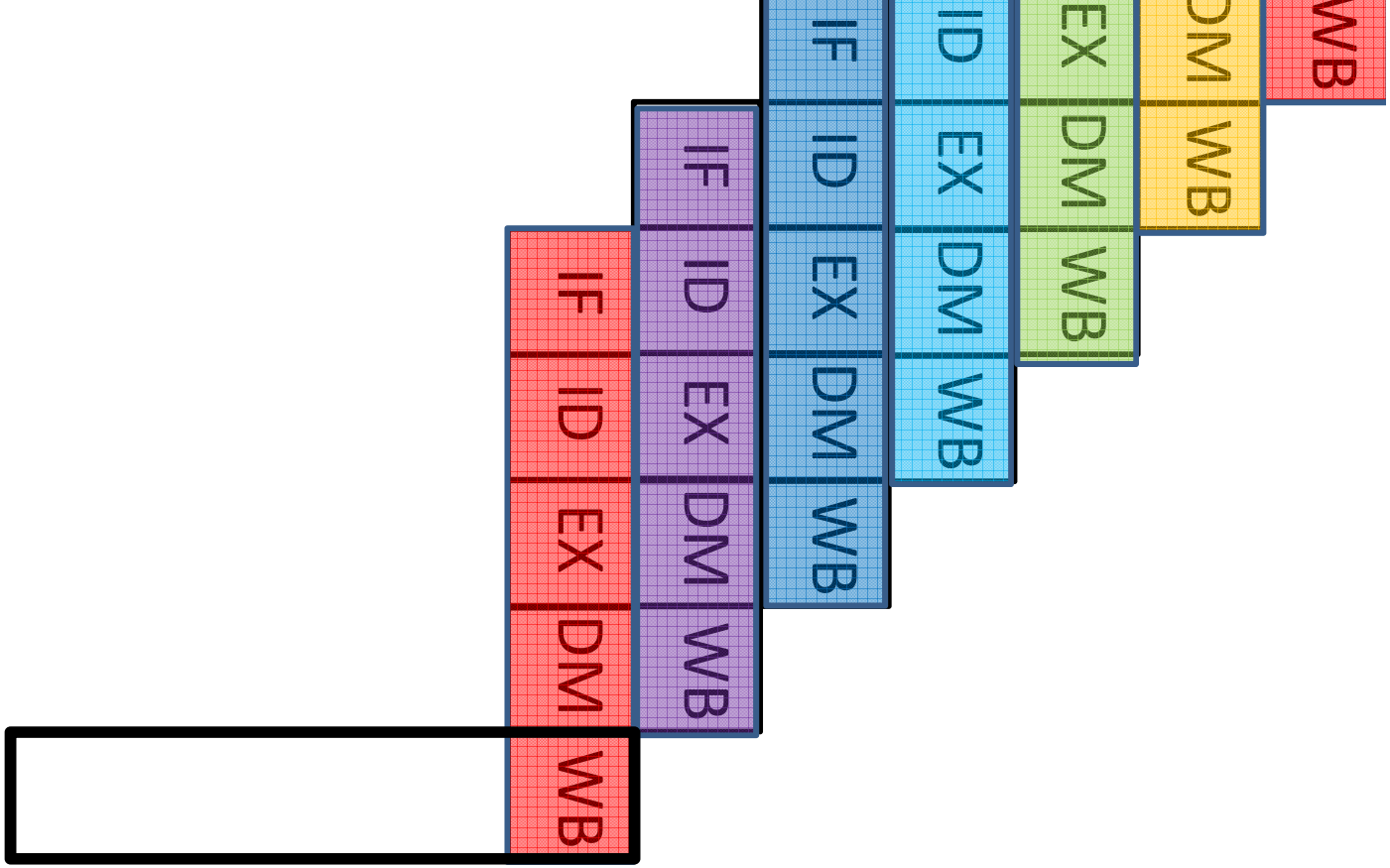
If we rotate that, we see the same picture we saw in the previous slides.



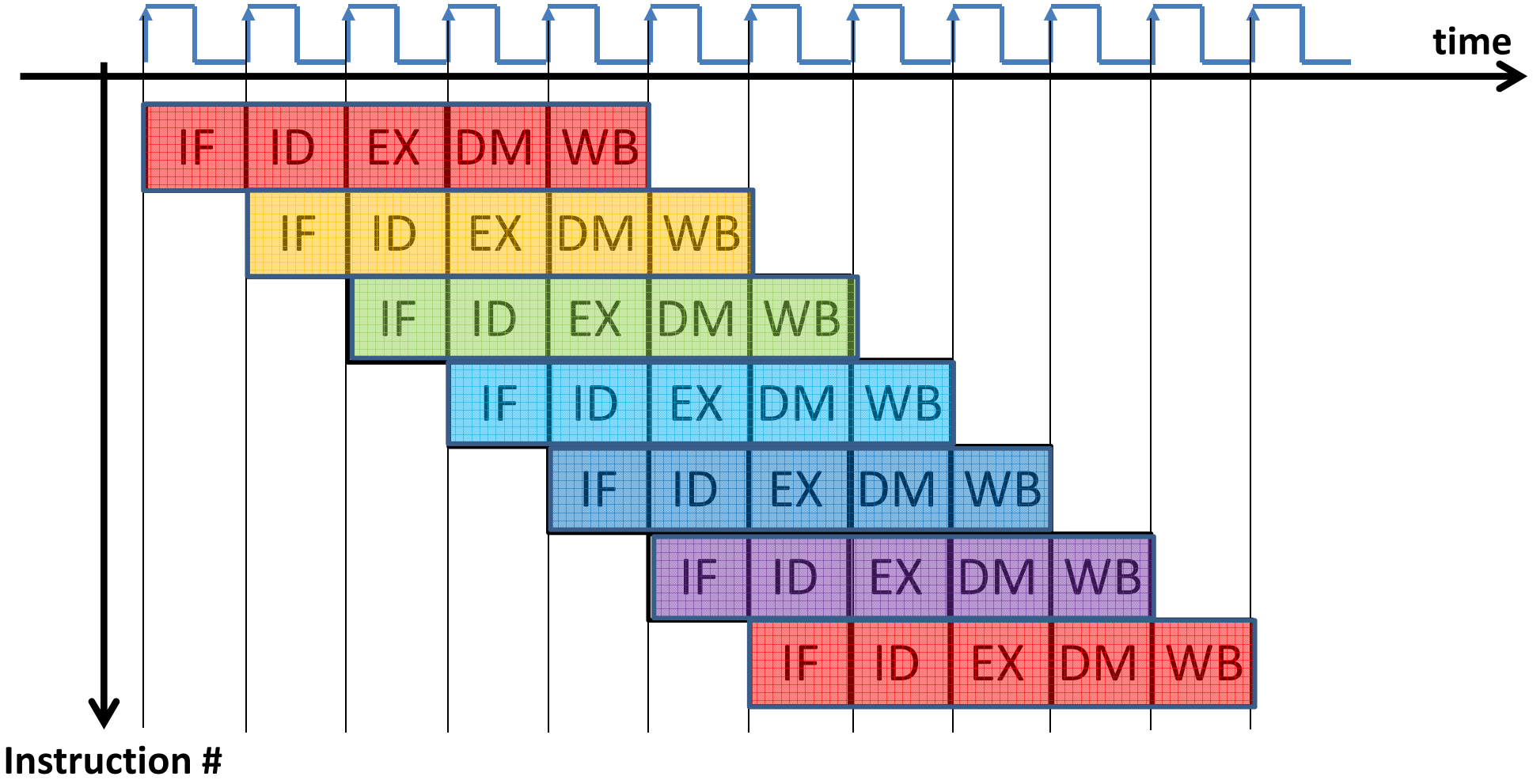
If we rotate that, we see the same picture we saw in the previous slides.



If we rotate that, we see the same picture we saw in the previous slides.



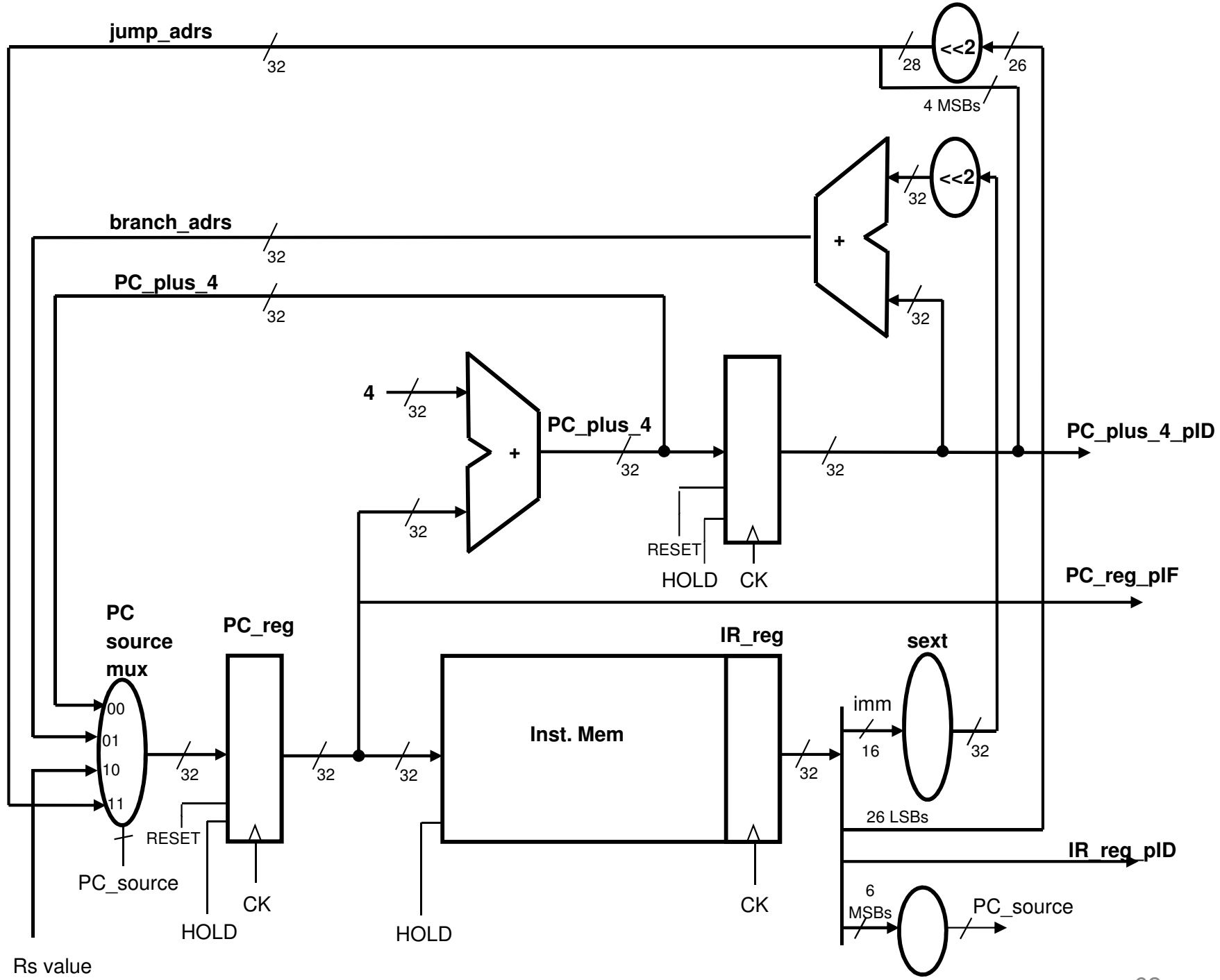
Pipelining timing representation



From now and on, we will use this scheme to describe what happens in the CPU.

HW2 – the Fetch Unit

The Fetch Unit

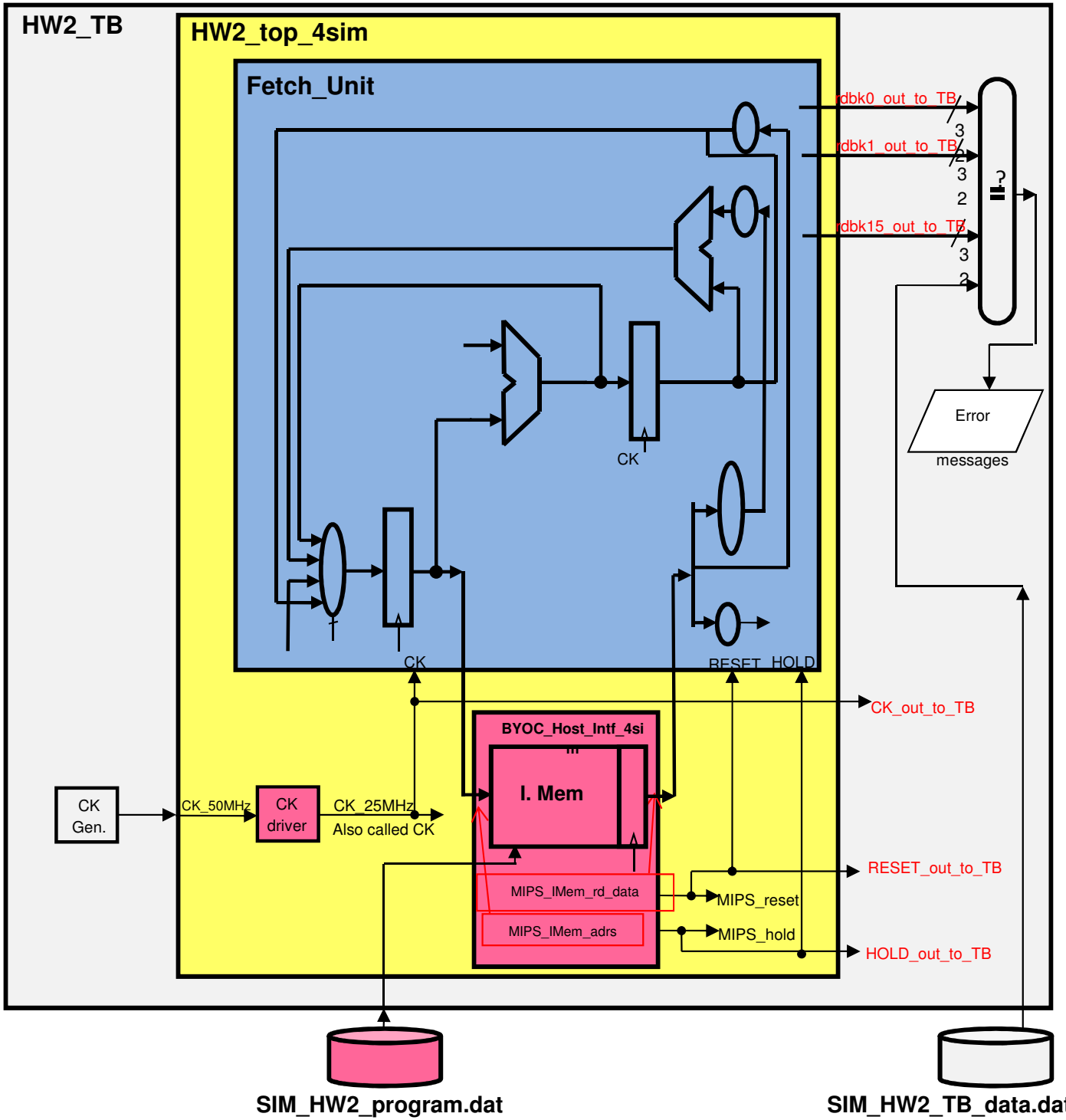


Names & definition of signals inside the Fetch Unit:

You must use these exact signal names in your design.

- **PC_reg** – a 32 bit register. **Reset** should force PC_reg= 0x400000
- **PC_plus_4** – a 32 bit signal that has the PC_reg value + 4.
- **PC_plus_4_pID** – a registered version of the PC_plus_4 to be used in the ID phase.
  This is why we added _pID at the end of that signal name.
- **branch_adrs** – a 32 bit signal which is made of PC_plus_4_pID + sext(imm)<<2.
  This is the address to be loaded into the PC when a successful branch is performed.
  Imm signal is made of the lower 16 bits of IR_reg (see IR_reg in #8 below).
- **jump_adrs** – a 32 bit signal made of PC_plus_4_pID[31:26] & IR[25:0] & b“00”, i.e.,
  the jump address in words multiplied by 4. This is the address to be loaded into the PC when a jump or a jal instruction is performed.
- **jr_adrs** – a 32 bit signal made of the Rs value in a JR instruction.
  Since we do not have a GPR file, we set the Rs value to x”00400004”.
  In the complete CPU this will be the address to be loaded into the PC when inst. is jr.
- **PC_source** – a 2 bit signal. When “00”, PC_reg is loaded with PC_plus_4. When “01” it is loaded with branch_adrs, when “10” with jr_adrs, when “11” with the jump_adrs.
- **IR_reg-** a 32 bit register that has the instruction we read from the IMem. This register is part of the IMem (The IMem is an already designed component we use in the Fetch Unit).
- **imm** – the 16 LSBs of IR_reg
- **sext_imm** – sign extension of imm to 32 bits
- **opcode** – the 6 MSBs of IR_reg. We sould determine the PC_source by the instruction opcode (j,jal-11, beq,bne-01,jr -10, any other instruction-00).
- **HOLD** – This signal is meant to freeze **all** registers when it is “1”. Very important !!

The Simulation project



The files we require to have in order to run the simulation are:

Group #1 – The design files

1. **HW2_top_4sim.vhd** – The pre-prepared top file connecting the Fetc_Unit, clock_driver & BYOC_host intf
2. **Fetch_Unit.vhd** - your design implementing figure 1

Group #2 – The infrastructure files [pre-prepared] The same for HW2, HW4, HW5,HW6.

3. **BYOC_clock_driver_4sim.vhd** – component dividing the CK from 50MHz to 25MHz
4. **BYOC_Host_Intf_4sim.vhd** – component having the IMem (& creates reset & hold)

Group #3 – The simulation files [different for each HW]

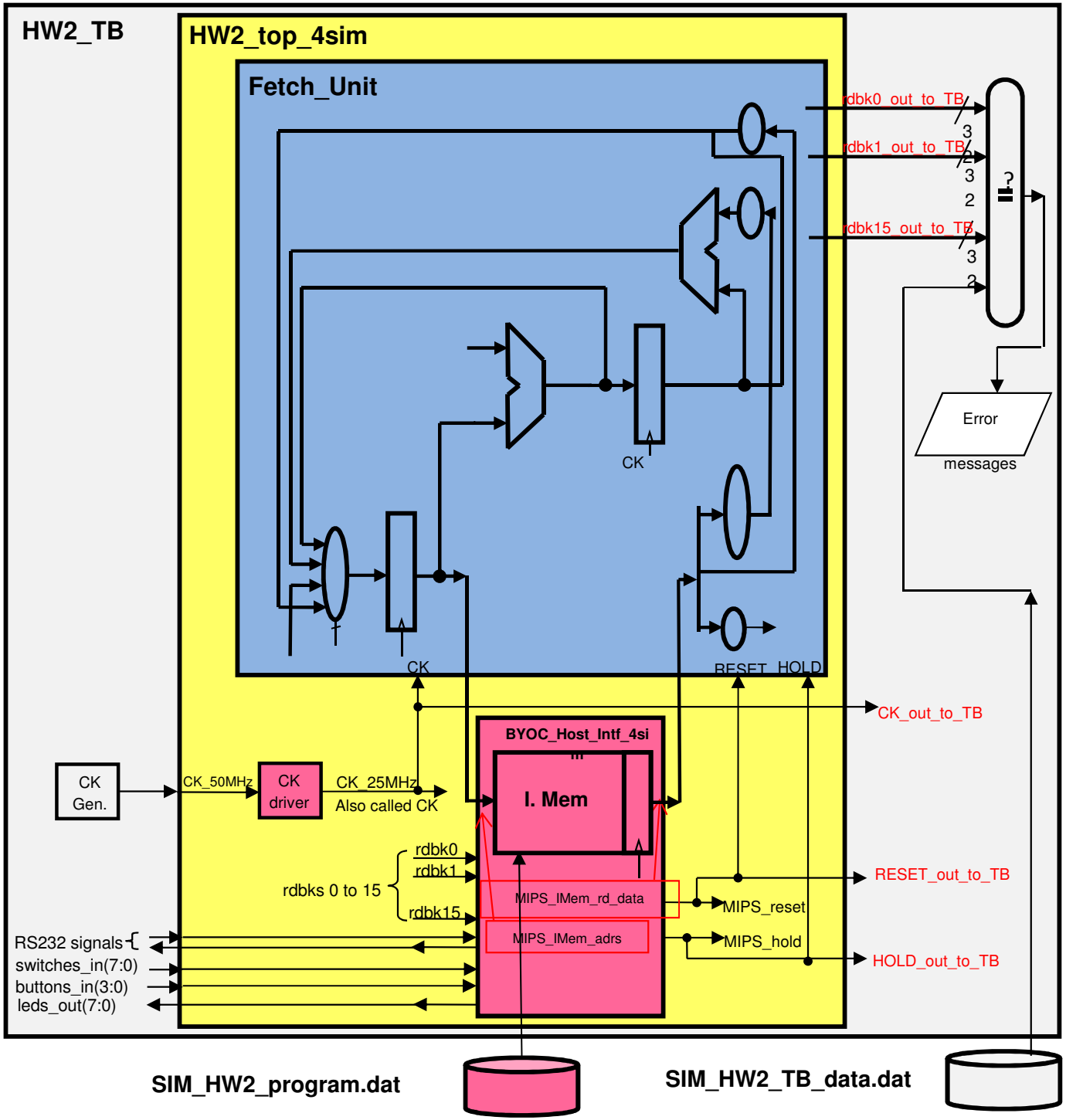
3. **SIM_HW2_TB.vhd** - The TB vhd file prepared in advance
4. **SIM_HW2_TB_data.dat** - The data file read by the TB during simulation for compari
5. **SIM_HW2_program.dat** - The program file for simulation.
6. **SIM_HW2_filenames.vhd** - The actual path information of the two dat files.

We prepared an “empty” **Fetch_Unit.empty** file in which we already did the following:

- Defined the I/O pins of the Fetch_Unit_4sim design
- Defined all necessary internal signal inside the Fetch_Unit_4sim design
- Connected other signals, e.g., signals to be outputted to the TB & rdbk signals

You have to rename it to Fetch_Unit.vhd and write the equations describing the logic circuitry of Figure 1!

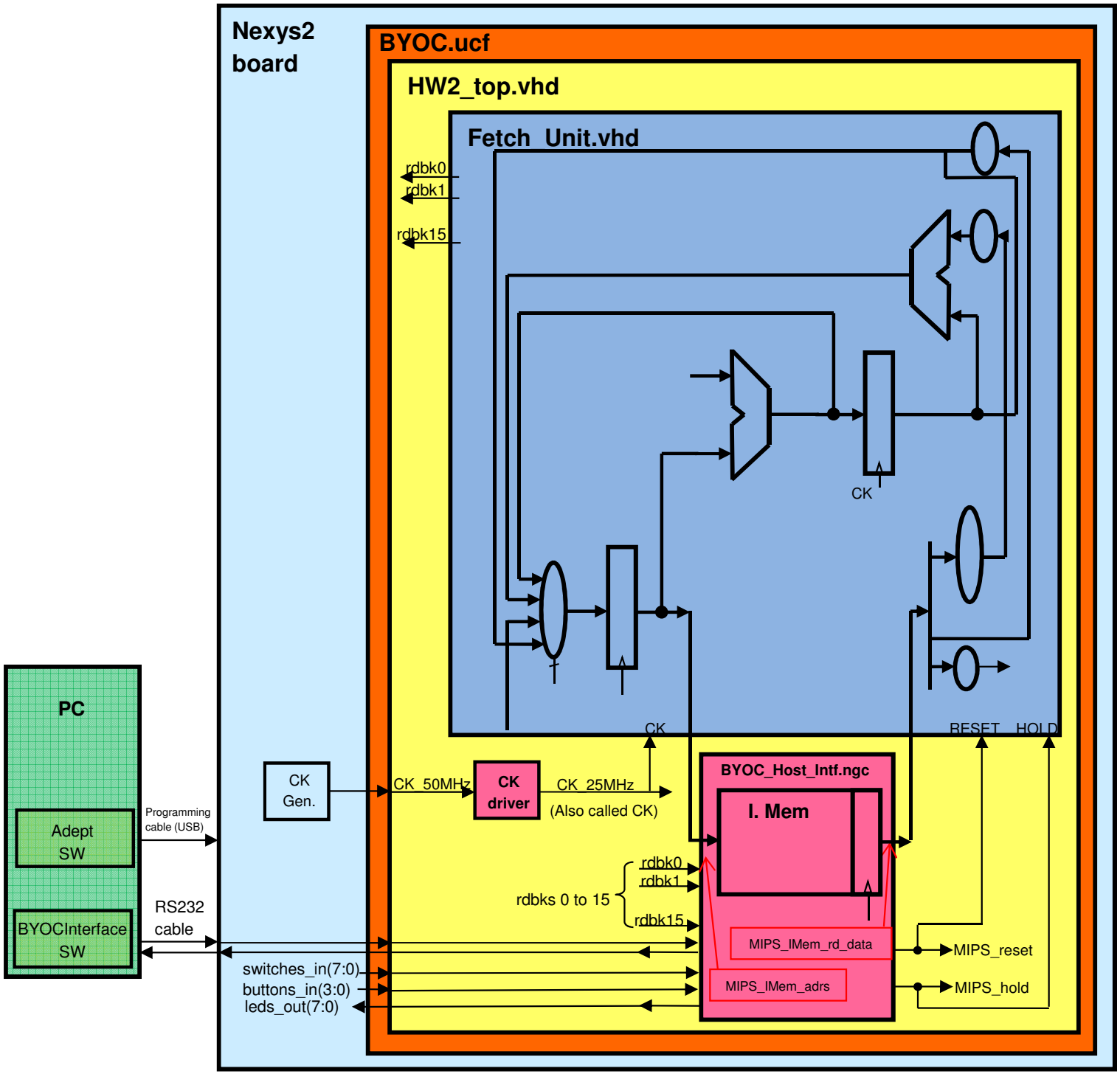
The Simulation project



Simulation report

- You should submit a zip file of your entire simulation & implementation project. Your zip file should have two directories **Simulation** & **Implementation**.
  In the **Simulation** one you'll have 3 sub-directories:
  Src_4sim – here you put all of the *.vhd sources and the *.dat file (used by the TB)
  Sim – here you should have the Fetch_unit_4sim project created by the simulator
  Docs – Here you put your simulation report. With your ID numbers (names-opt.)
- Run the simulation to 5500 nS.
- In the DOC file you need to attach with screen captures describing the simulation you made. All signals mentioned in section 1a above should be presented in the screen capture. Show at least the 1st 10 ck cycle following the end of the reset pulse and make the values of all signals readable.
- In that doc file you need to answer the questions appearing in the HW2 doc.

The Implementation project



So the files we require to have in order to run the implementation are:

Group #1 – The design files

1. **HW2_top.vhd** – The top file after renaming and removal of all signals outputted to the TB
1. **Fetch_Unit.vhd** - your design implementing figure 1

Group #2 – The infrastructure files

3. **BYOC_clock_driver.vhd** – A CK divider from 50MHz to 25MHz that has a **BUFG** driver inside
3. **BYOC_Host_Intf.ngc** – The pre-compiled component including the IMem and creating the reset & hold signals and the infrastructure interfacing to the PC which is required to run the implemented design.

Group #3 – The implementation files

5. **BYOC.ucf** - The file listing which signals are connected to which FPGA pins in the Nexys2 board.

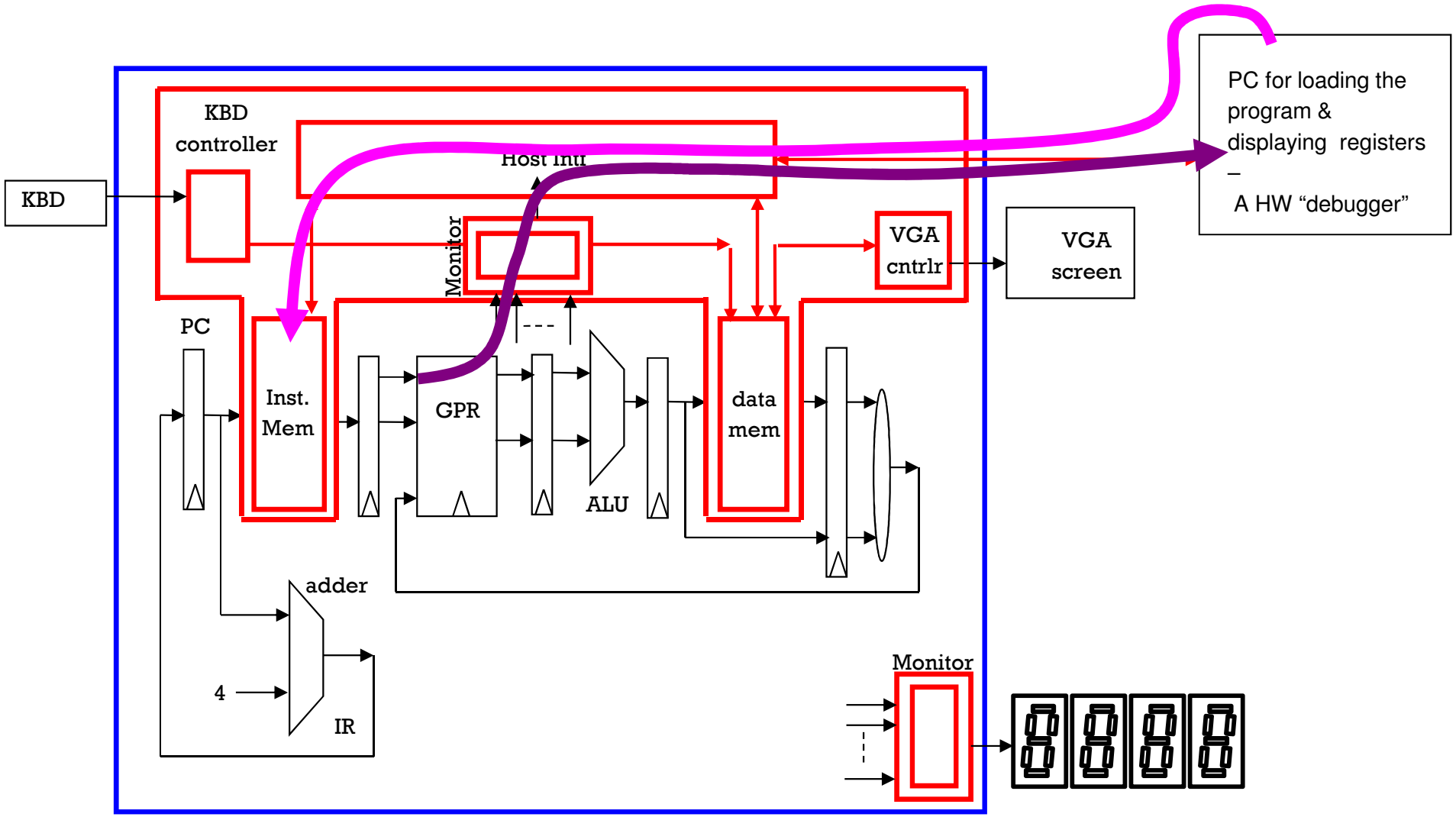
A reminder: We connected the rdbk signals as follows:

rdbk0 => PC_reg,
rdbk1 => PC_plus_4,
rdbk2 => branch_adrs,
rdbk3 => jr_adrs,
rdbk4 => jump_adrs,
rdbk5 => PC_plus_4_pID,
rdbk6 => IR_reg.
rdbk7 => PC_source (bits 1:0),
rdbk8 => RESET (bit 0),
rdbk9 => output of PC_mux (32 bit signal).,
rdbk10 - 15 => 0x00000000

- **We run ISE and get a bit file**
- **Load the design using the Adept SW**
- **Load the IMem via the BYOCIntf SW**
- **Then use it also to apply a single ck and check the rdbk signals**

- **Load the IMem via the BYOCIntf SW**

- **Then use it also to apply a single ck and check the rdbk signals**



Now it is your turn!

Thanks for listening!