MIPS ALU & GPR File

[GPR File - General Purpose Register File]

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)
```

6	5	5	5		6
000000	Rs	Rt	Rd	00000	FUNCTION
OPCODE					

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)
```

6	5	5	16
OPCODE	Rs	Rt	imm

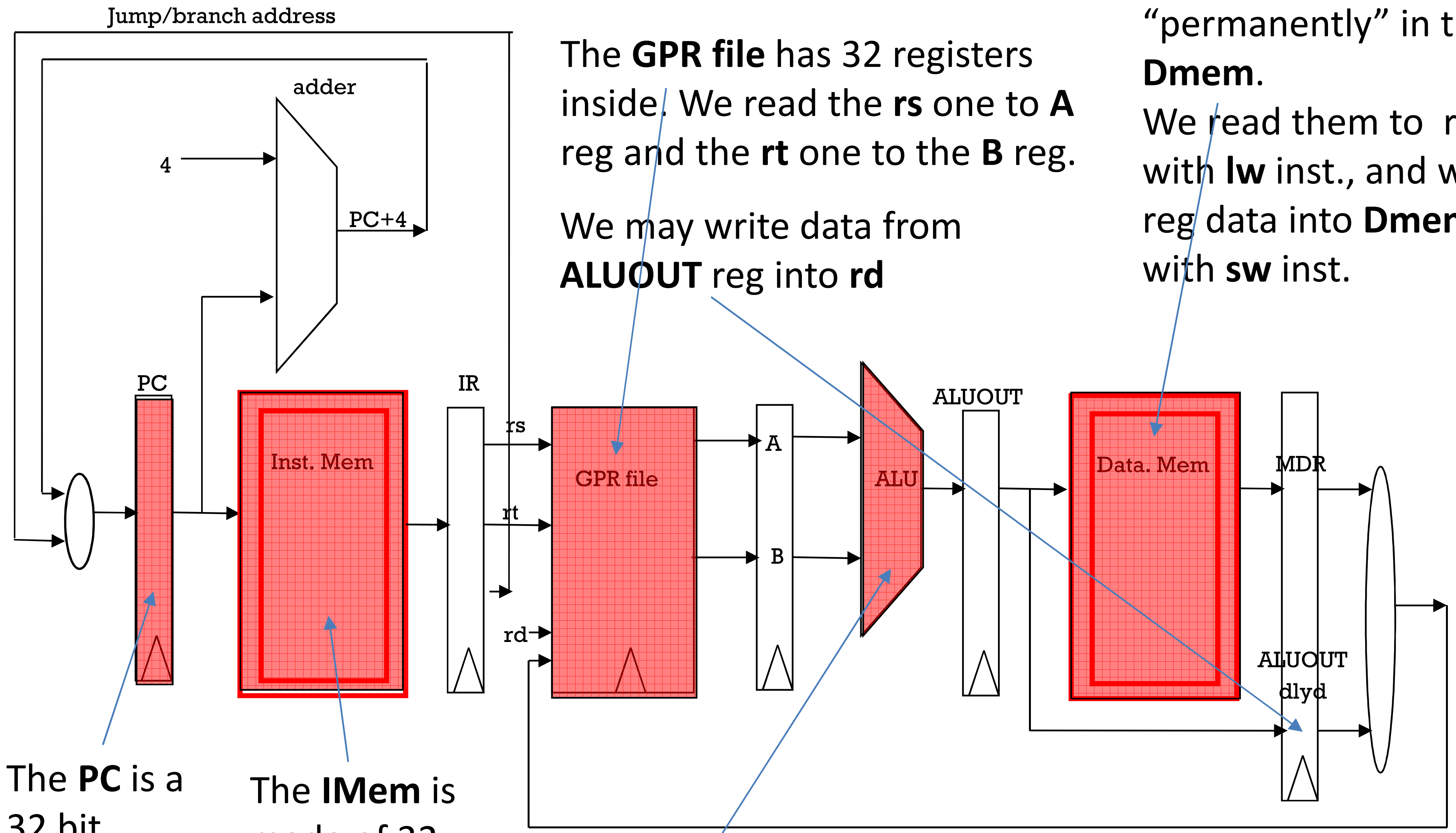
---

j-type

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

6	26
OPCODE	26 bit imm

The pipelined MIPS



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**

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

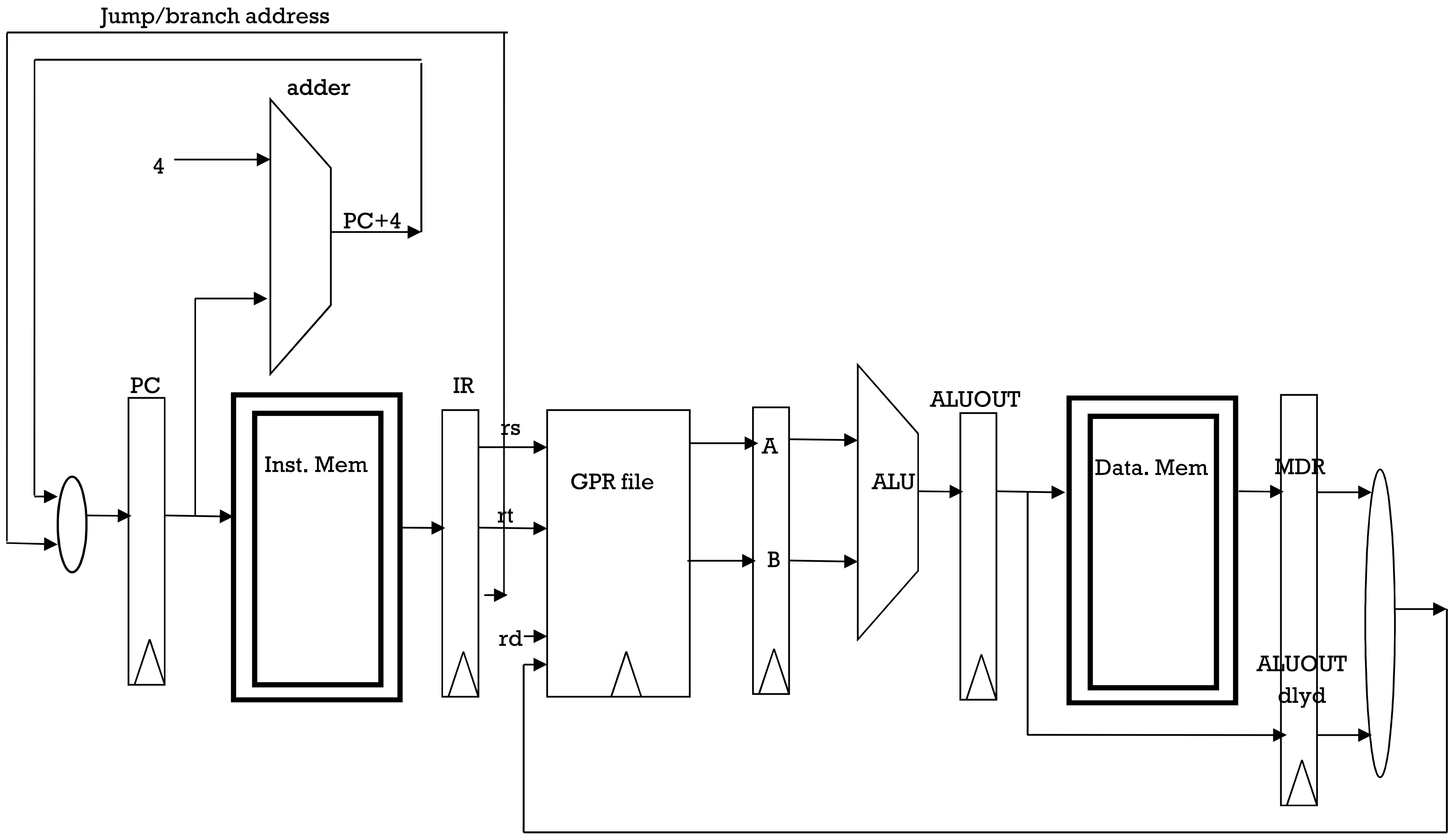
The **PC** is a 32 bit register

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

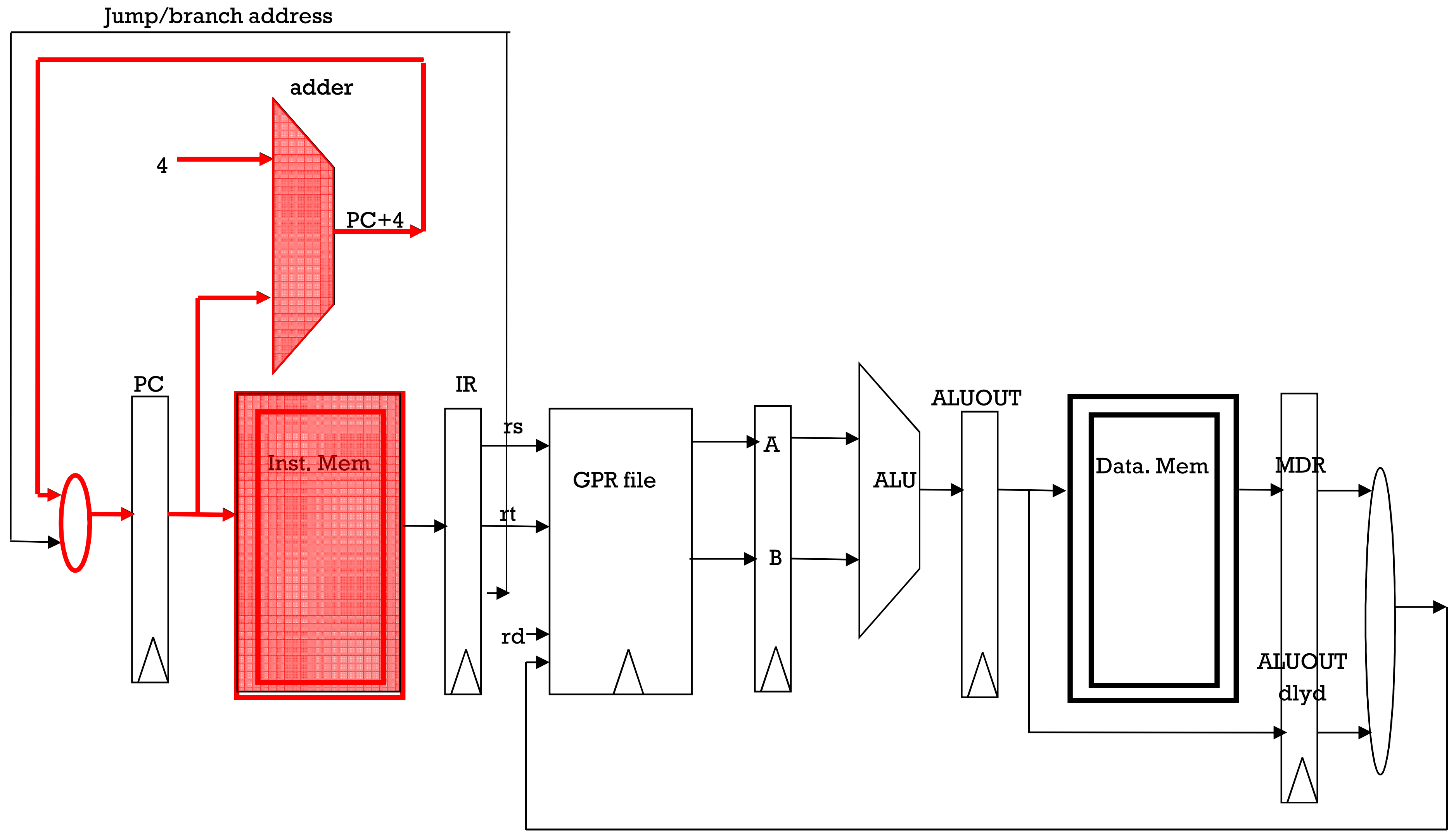
The **ALU** calculates **A** op **B**.
Its result goes to the **ALUOUT** reg

Rtype instructions

Performing Rtype inst.



Performing Rtype inst.

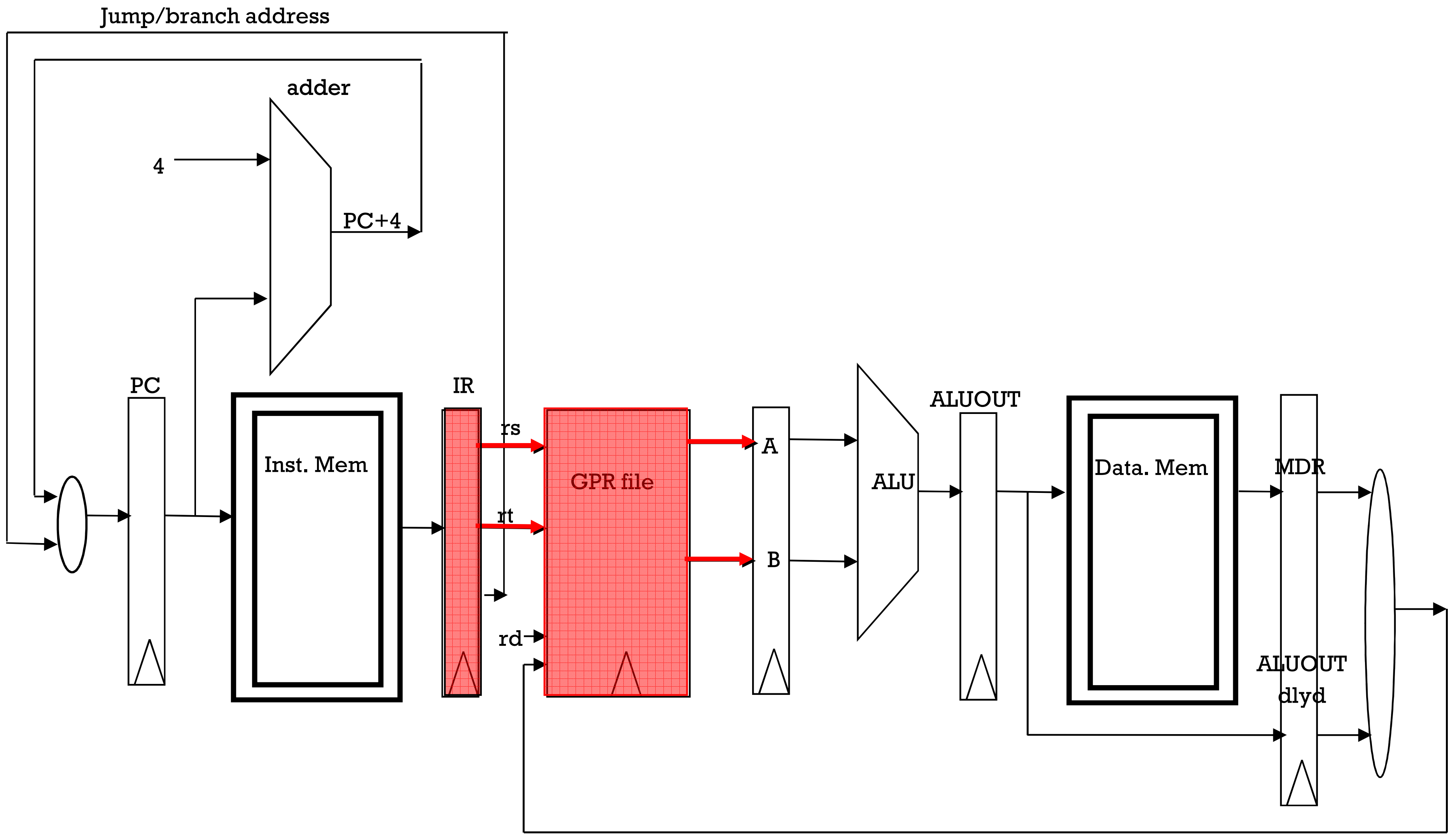


IF – Inst. Fetch

IR = Imem[PC]

PC = PC+4

Performing Rtype inst.

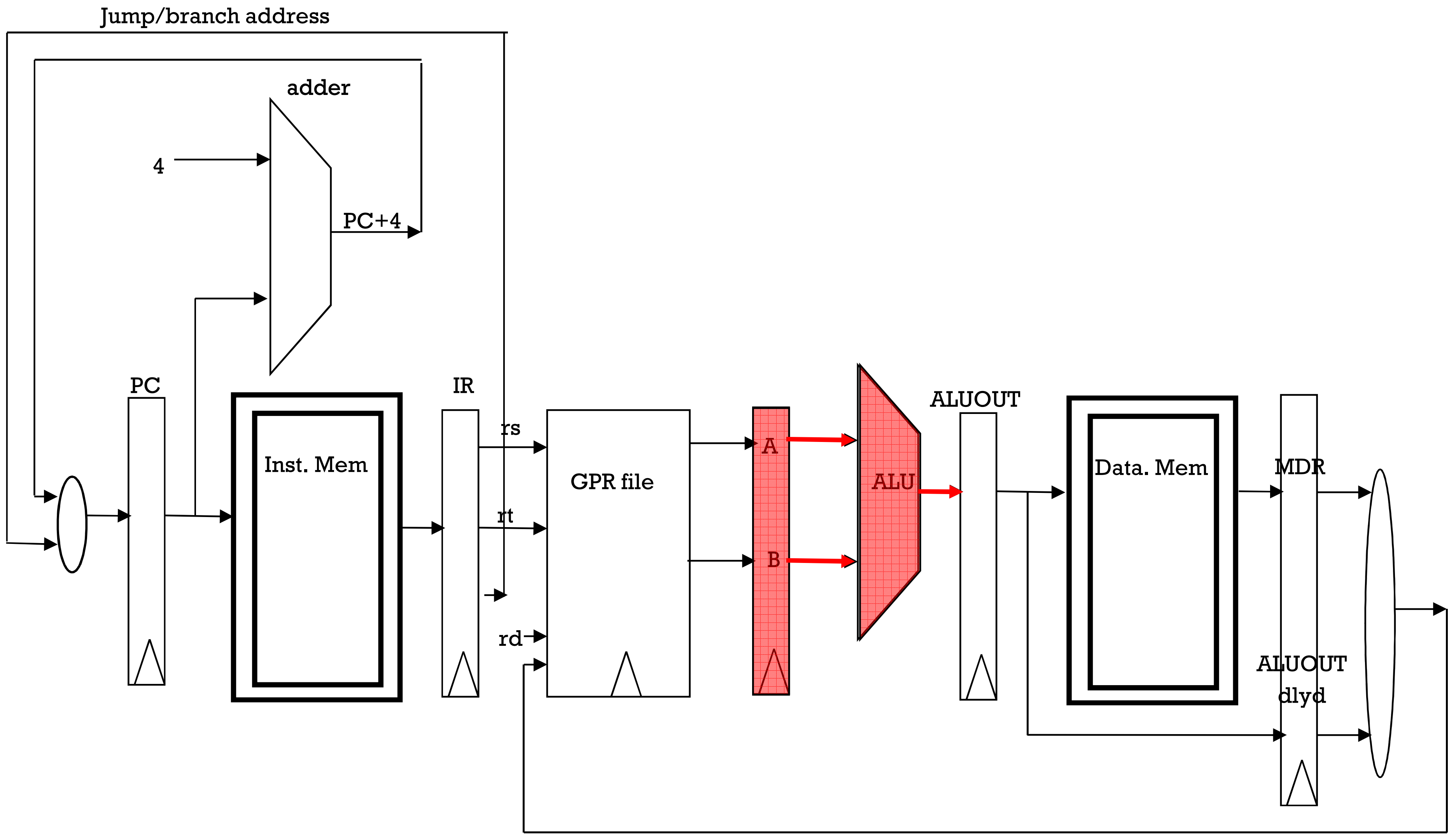


ID – Inst. Decode

A = rs , B = rt

(& decode
control signals)

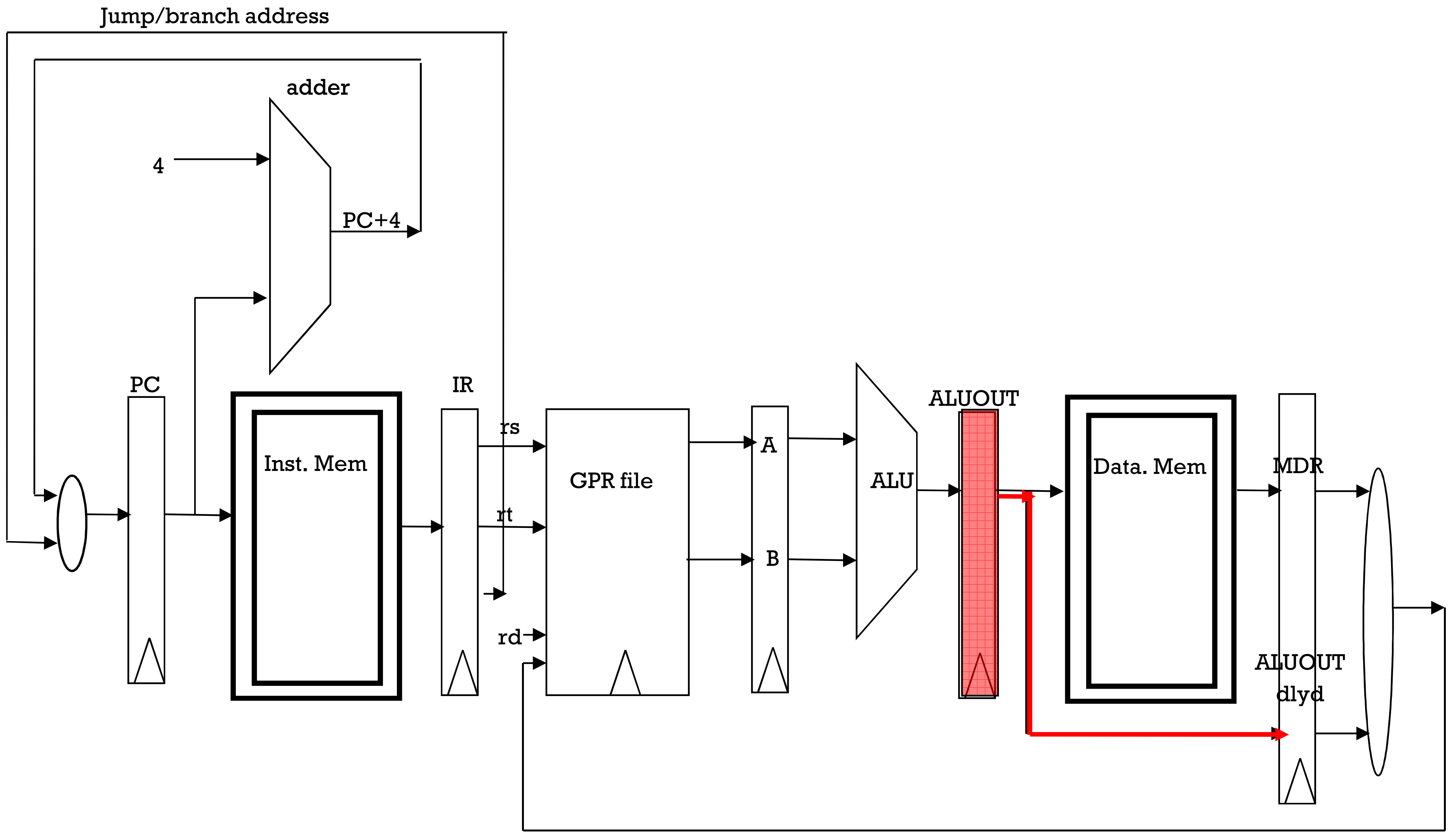
Performing Rtype inst.



EX– Execute

ALUOUT = A op B

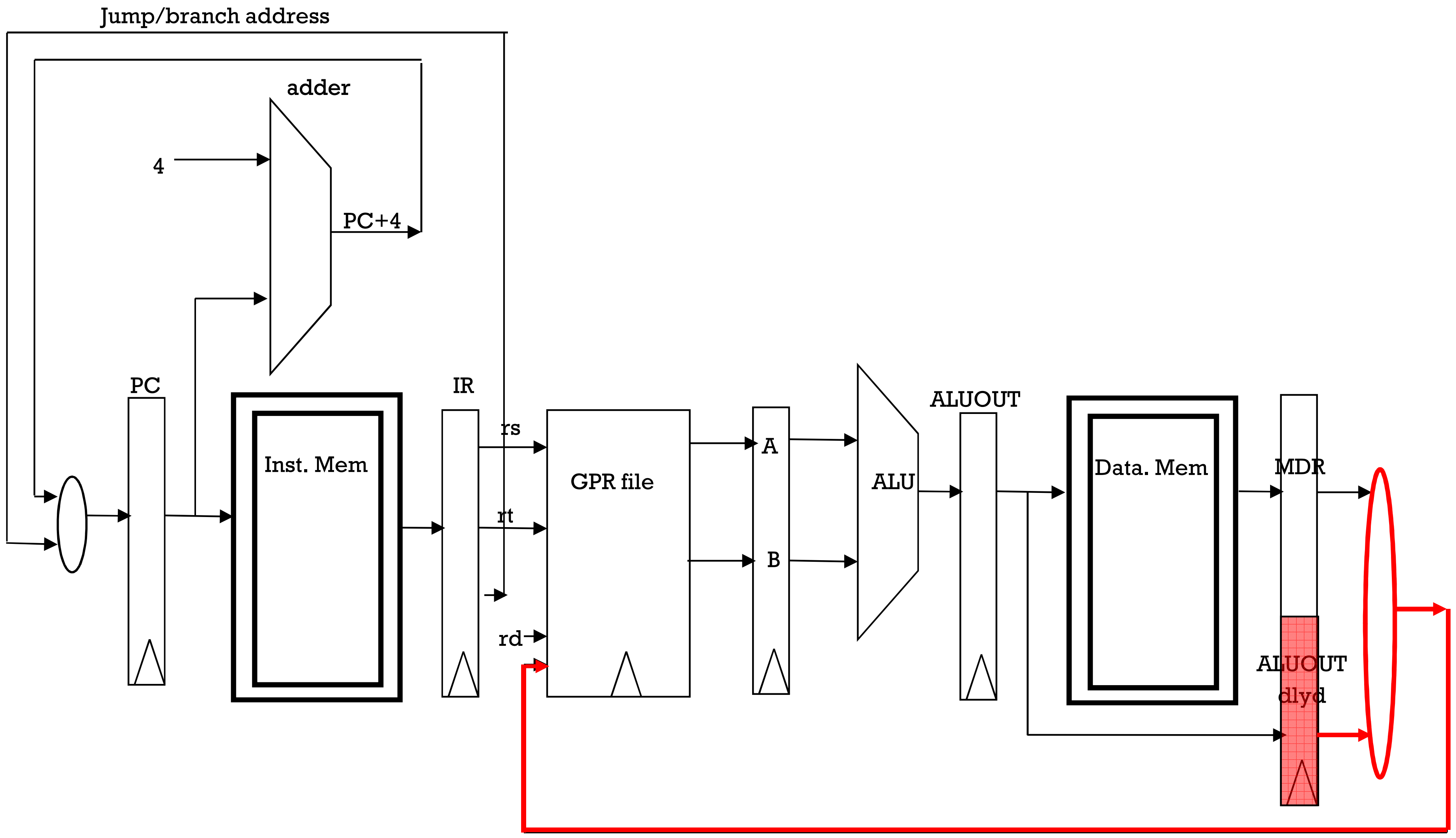
Performing Rtype inst.



MEM – Memory

In Rtype – wait 1 ck

Performing Rtype inst.

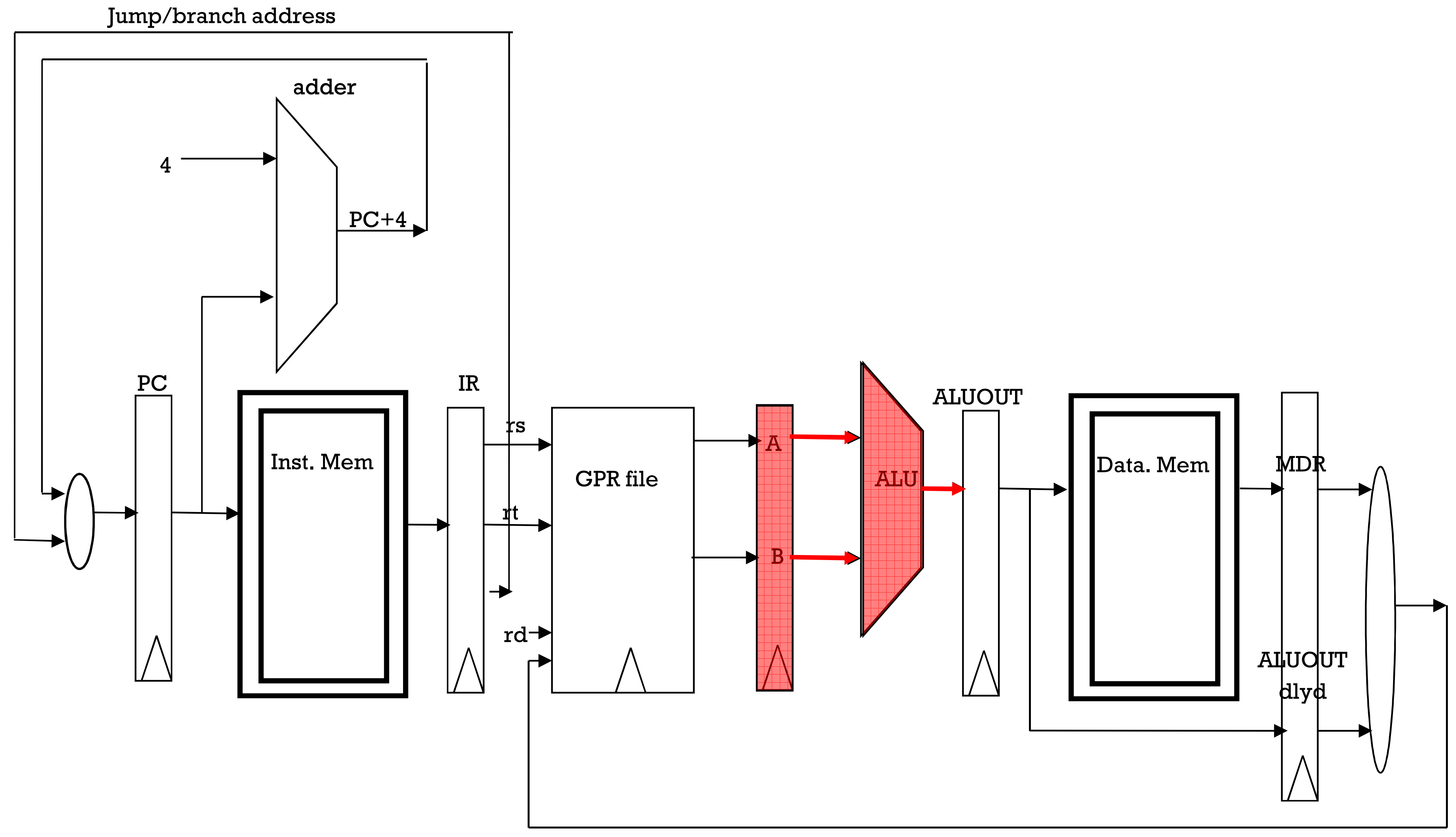


WB – write back

Rd = ALUOUT

The ALU

Performing Rtype inst.



EX– Execute

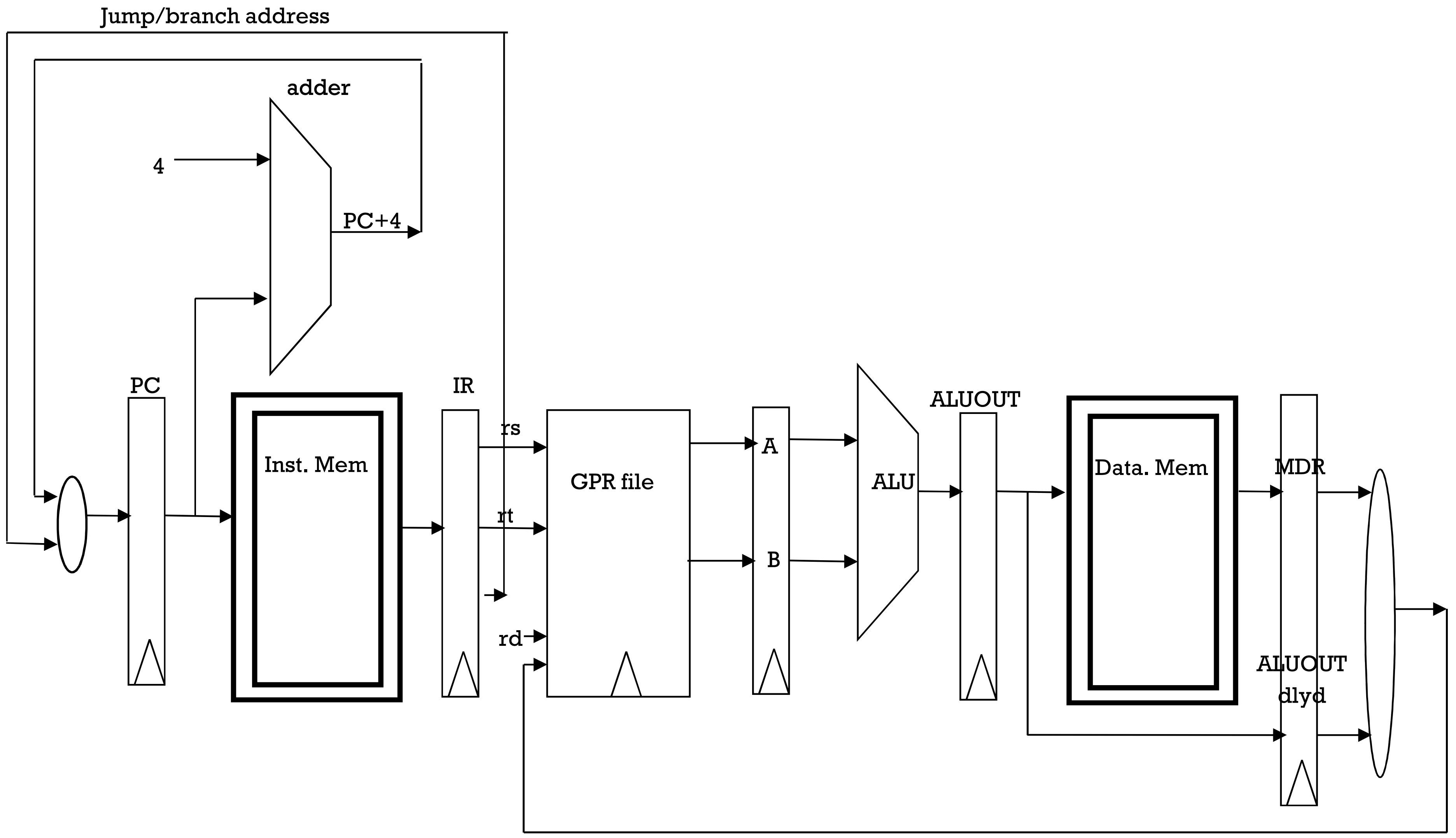
ALUOUT = A op B

MIPS ALU

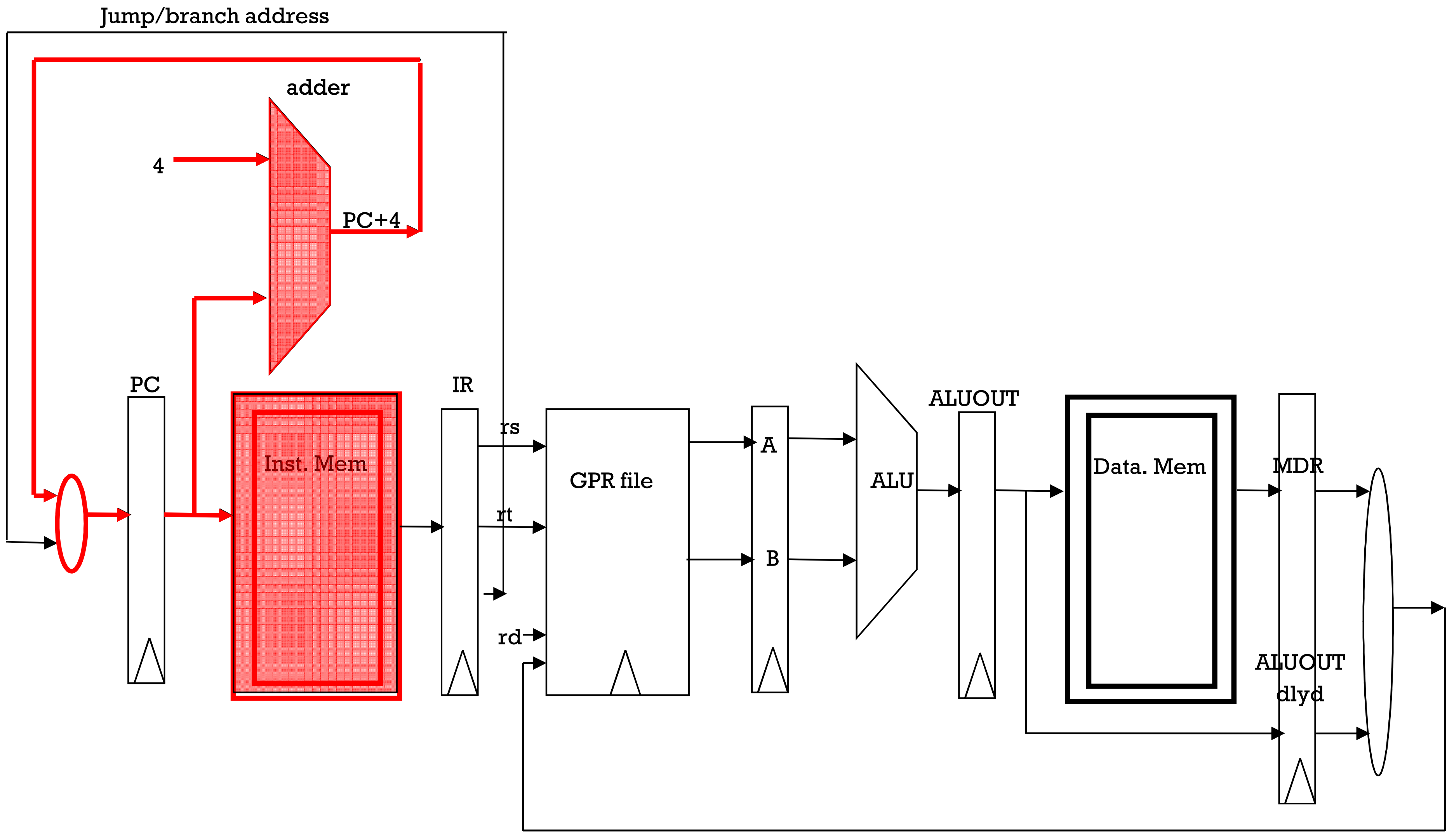
```
-- MIPS ALU
process(ALU_A_in, ALU_B_in, ALU_cmd, sign_of_sub)
begin
   case ALU_cmd is
      when b"000" =>   ALU_output <= ALU_A_in and ALU_B_in;-- AND
      when b"001" =>   ALU_output <= ALU_A_in or ALU_B_in; -- OR
      when b"010" =>   ALU_output <= ALU_A_in + ALU_B_in; -- ADD
      when b"011" =>   ALU_output <= ALU_A_in xor ALU_B_in; -- XOR
      when b"100" =>   ALU_output <= not(ALU_A_in and  ALU_B_in); -- ???
      when b"101" =>   ALU_output <= not(ALU_A_in or ALU_B_in); --???
      when b"110" =>   ALU_output <= ALU_A_in - ALU_B_in; -- SUB
      when others  =>   ALU_output <= x"0000000" & b"000" & sign_of_sub;-- SLT
   end case;
end process;
```

Pipelined operation

Performing Rtype inst.



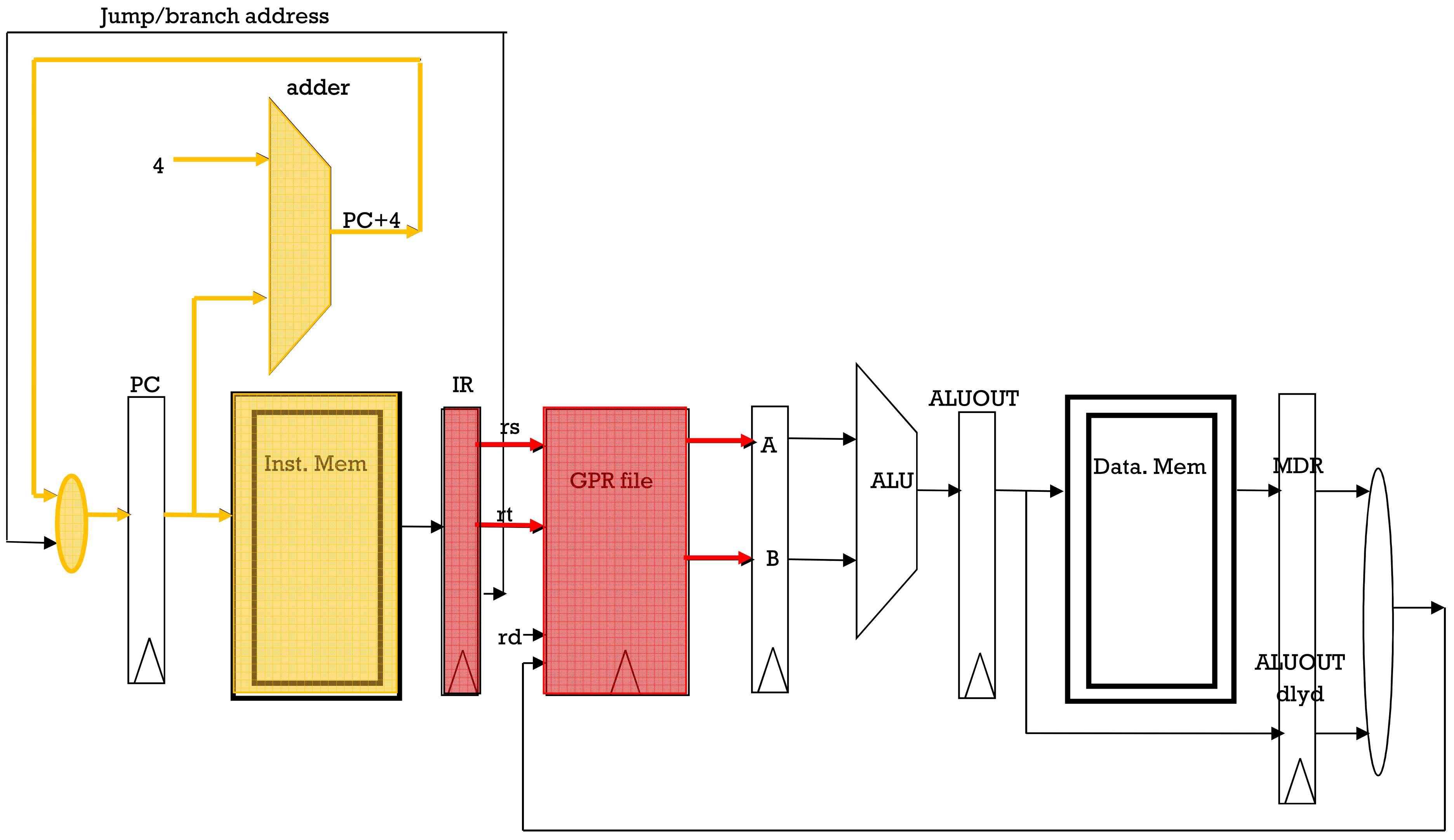
Performing Rtype inst.



IF – Inst. Fetch

IR = Imem[PC]
PC = PC+4

Performing Rtype inst.



IF – Inst. Fetch

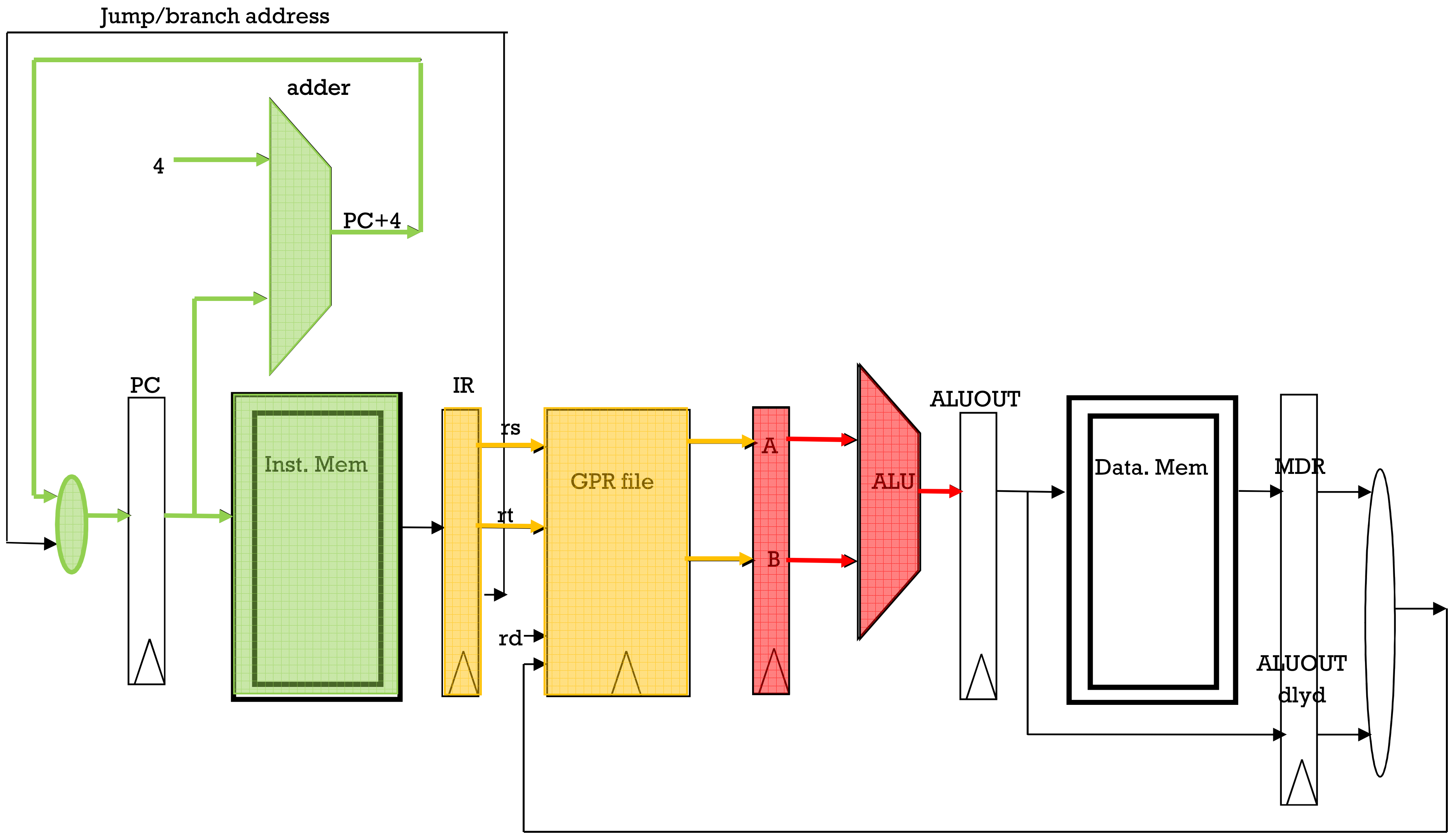
IR = Imem[PC]
PC = PC+4

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.

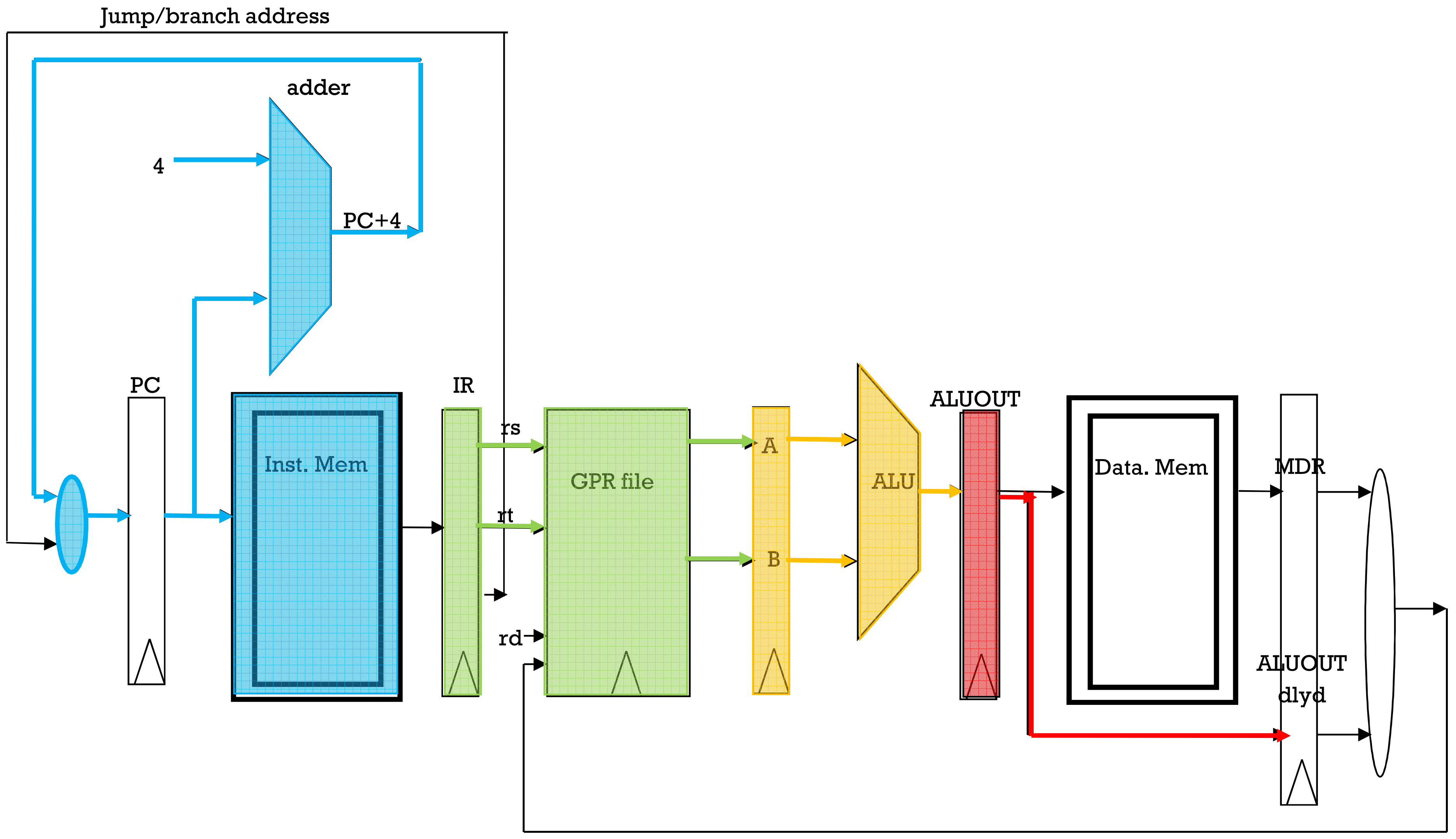
A = rs , B = rt

(& 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)

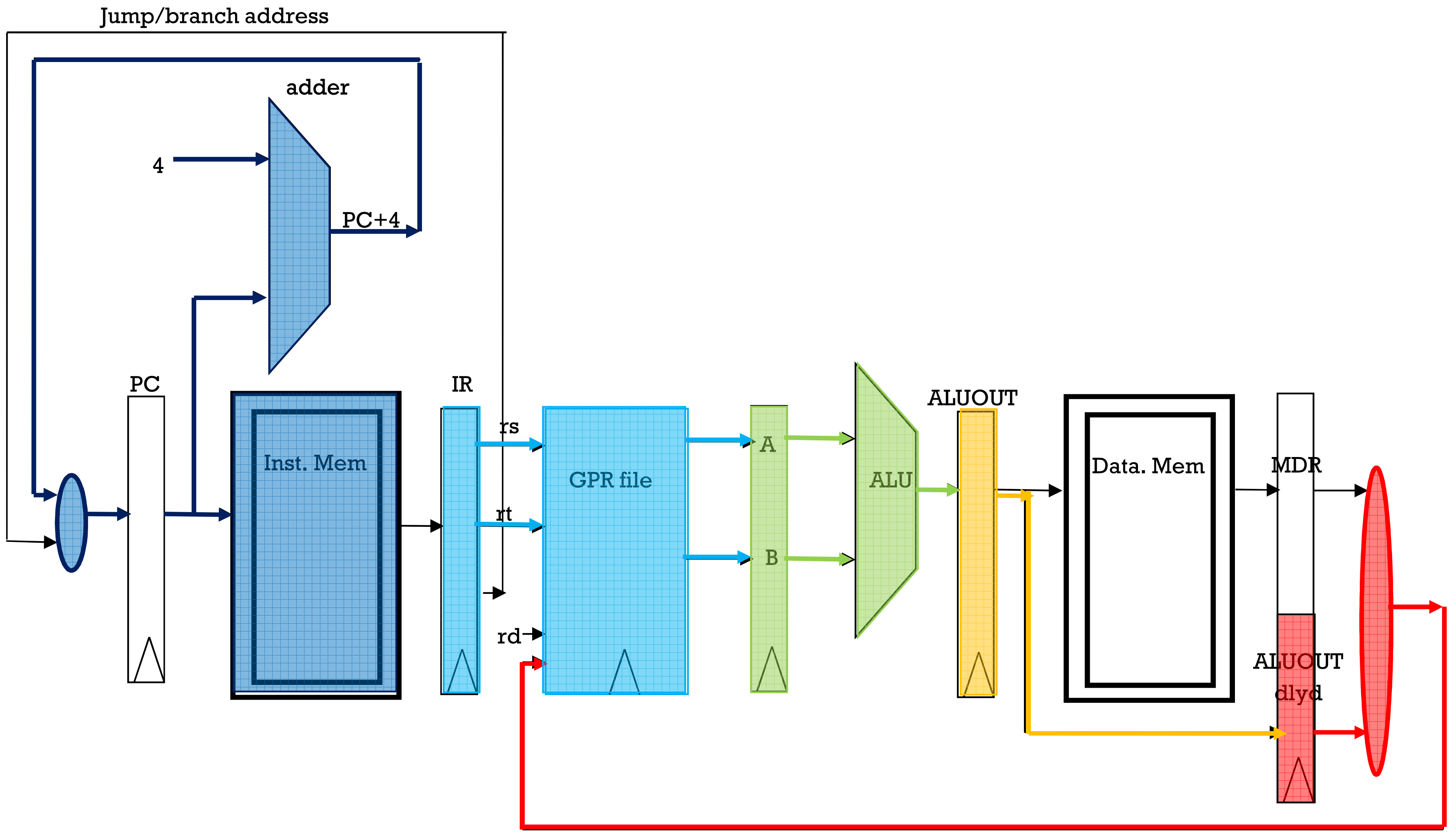
EX– Execute

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

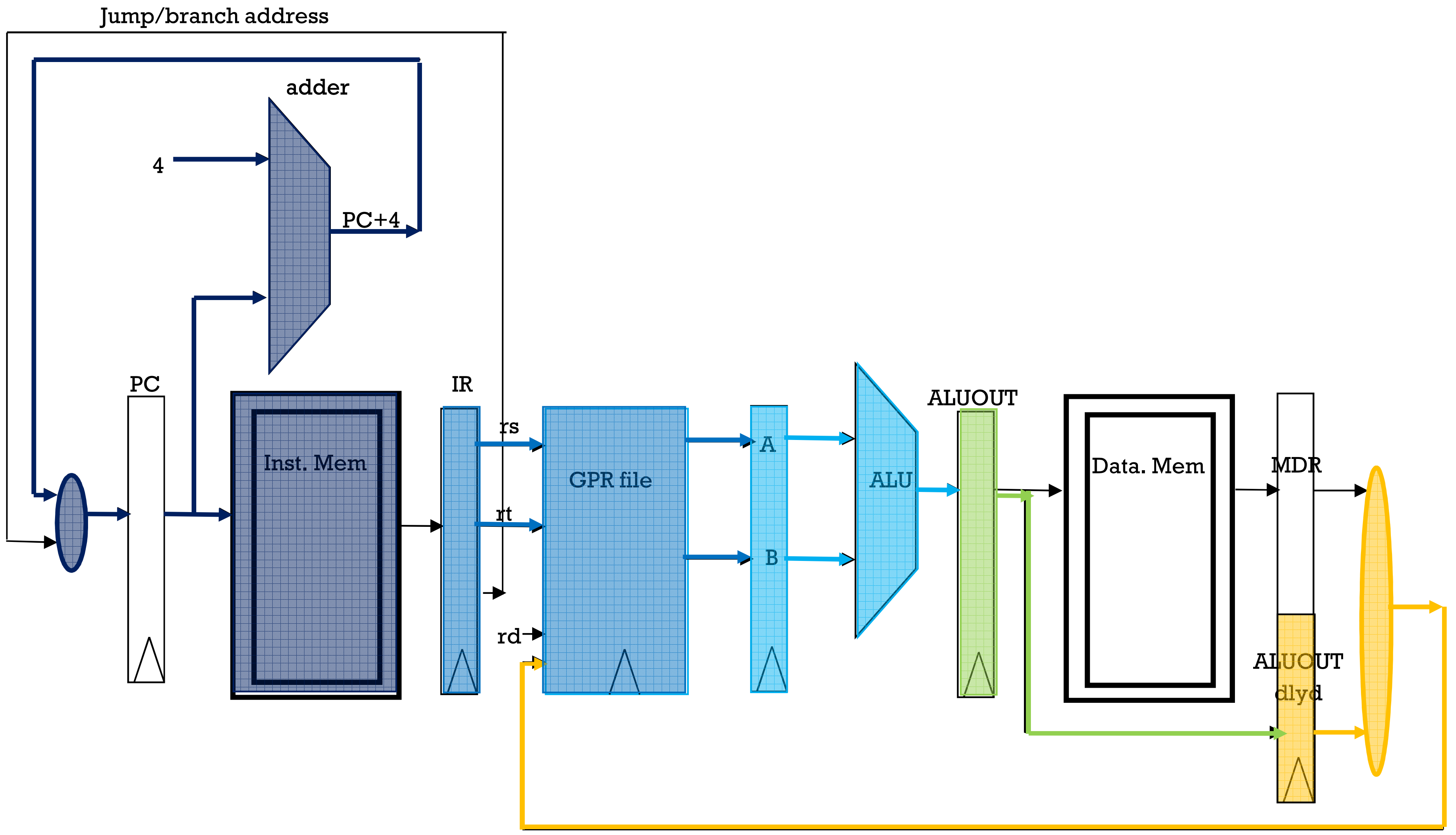
MEM

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

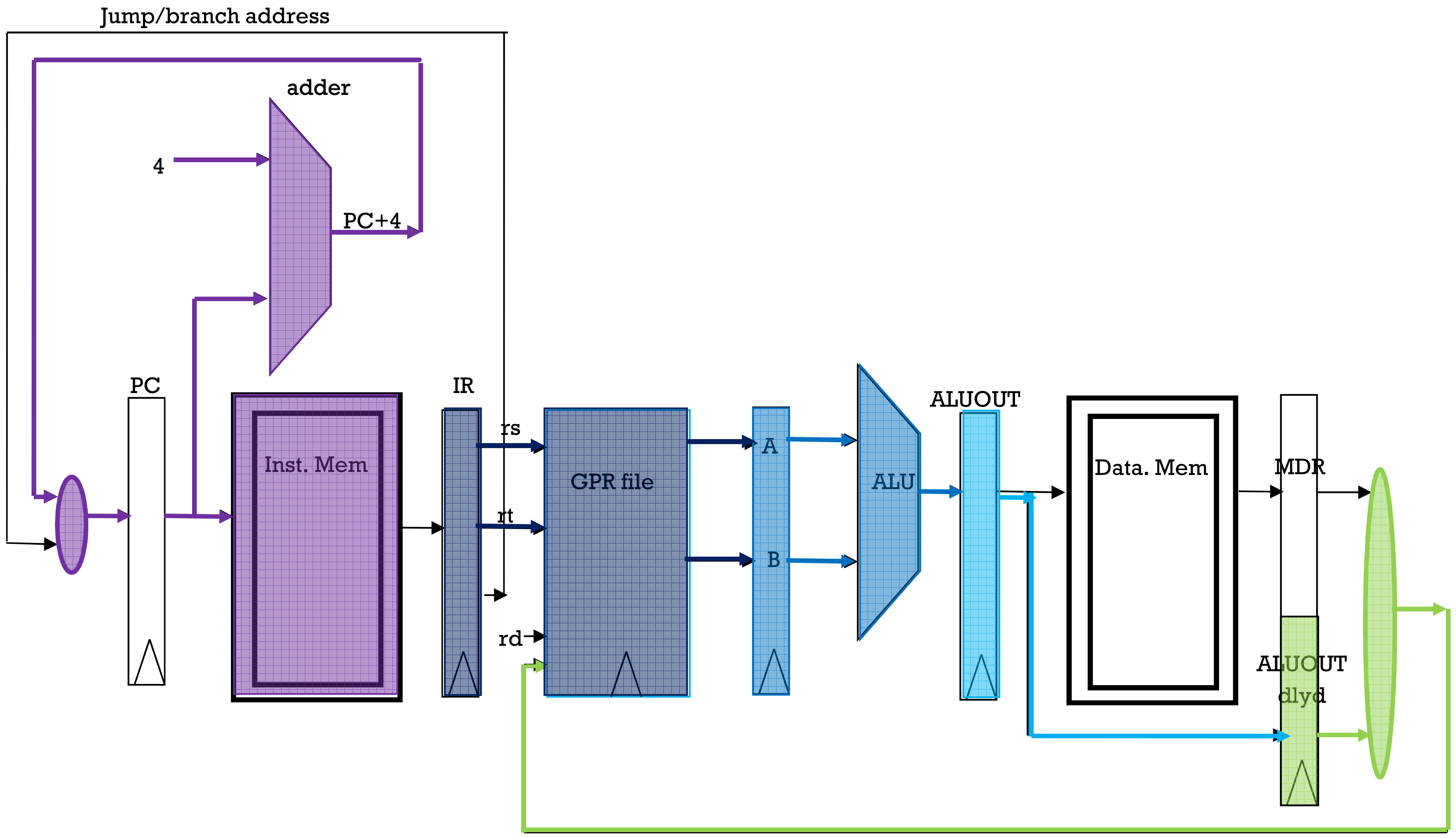
MEM

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

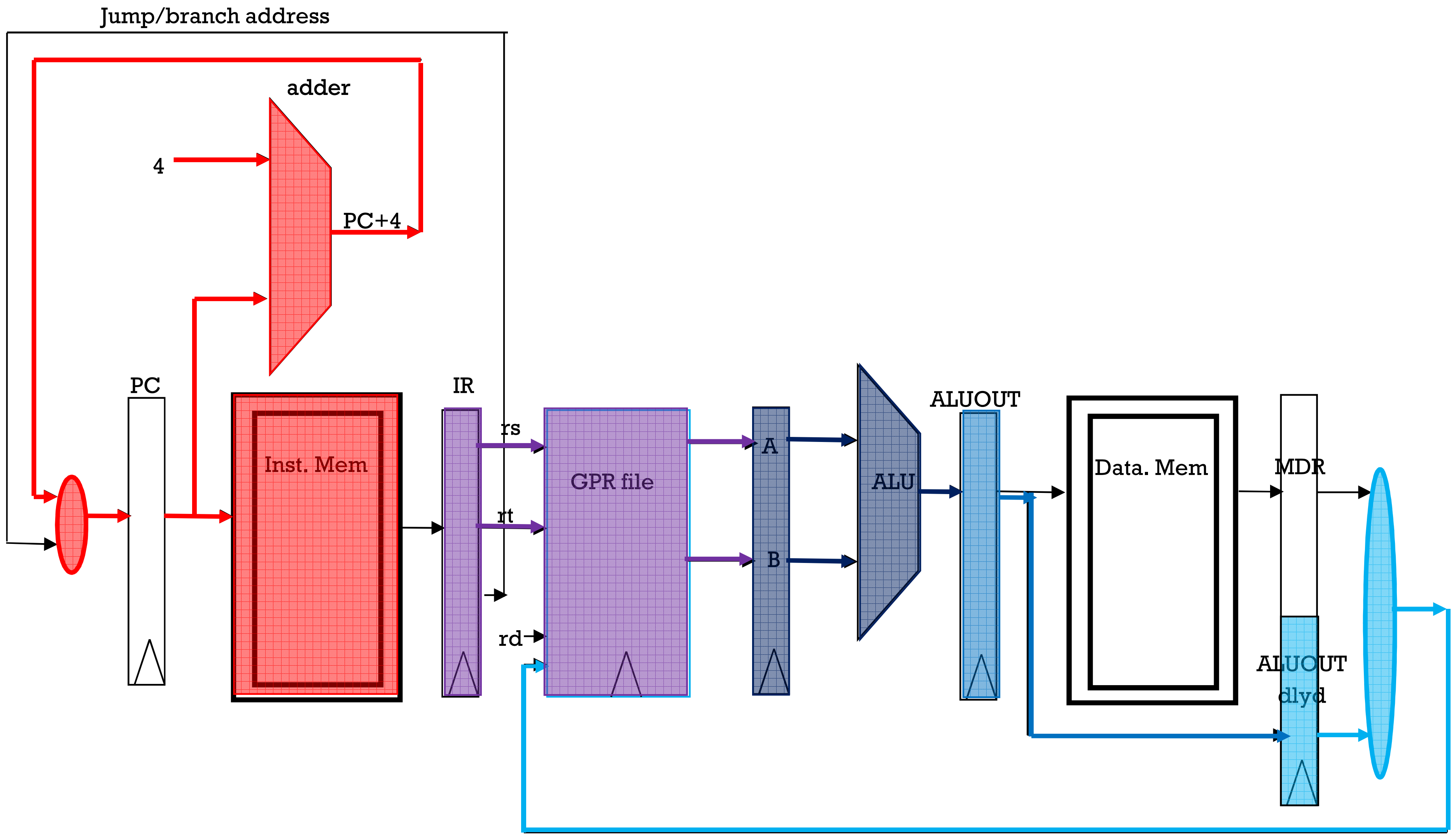
MEM

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

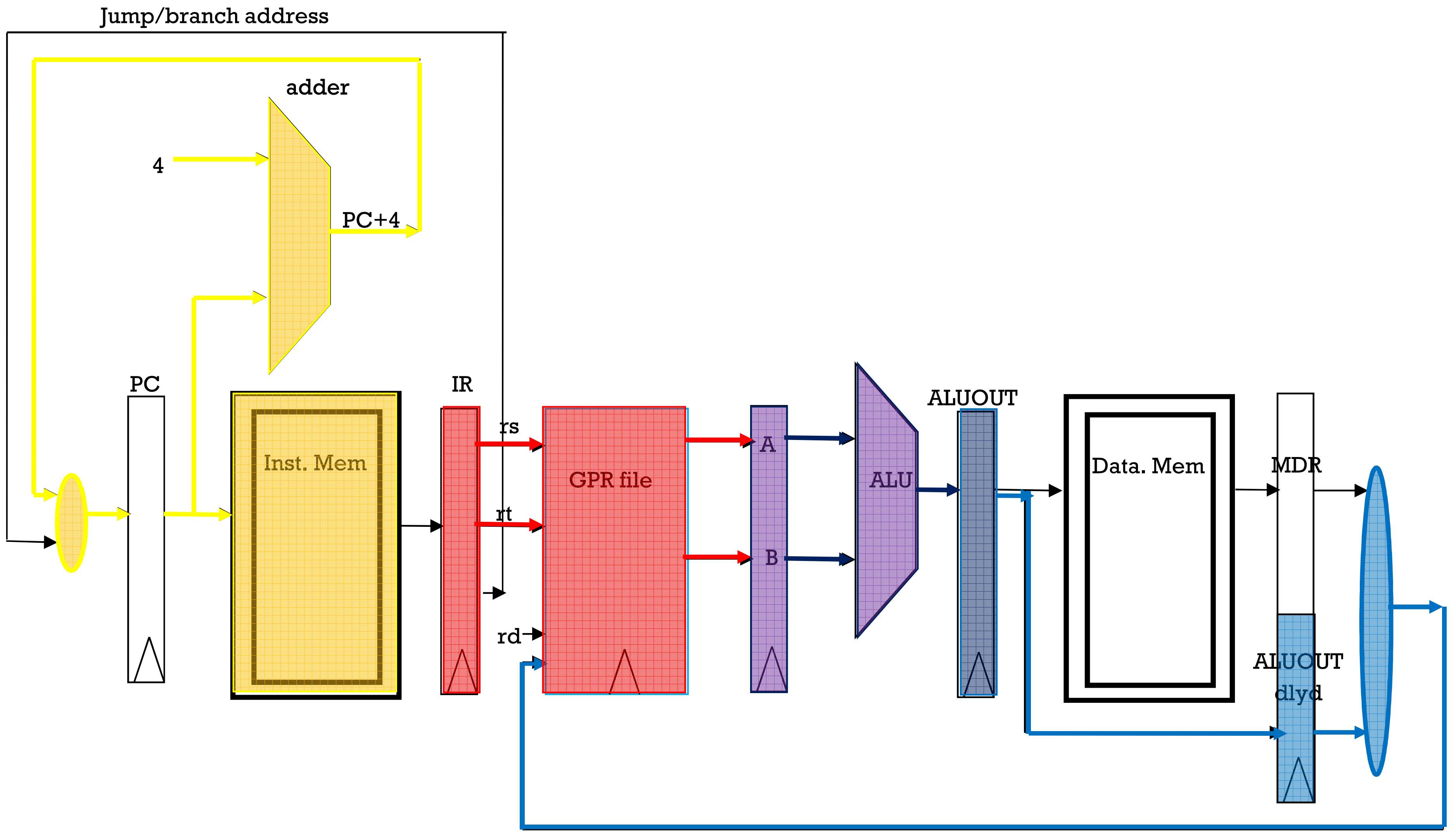
MEM

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

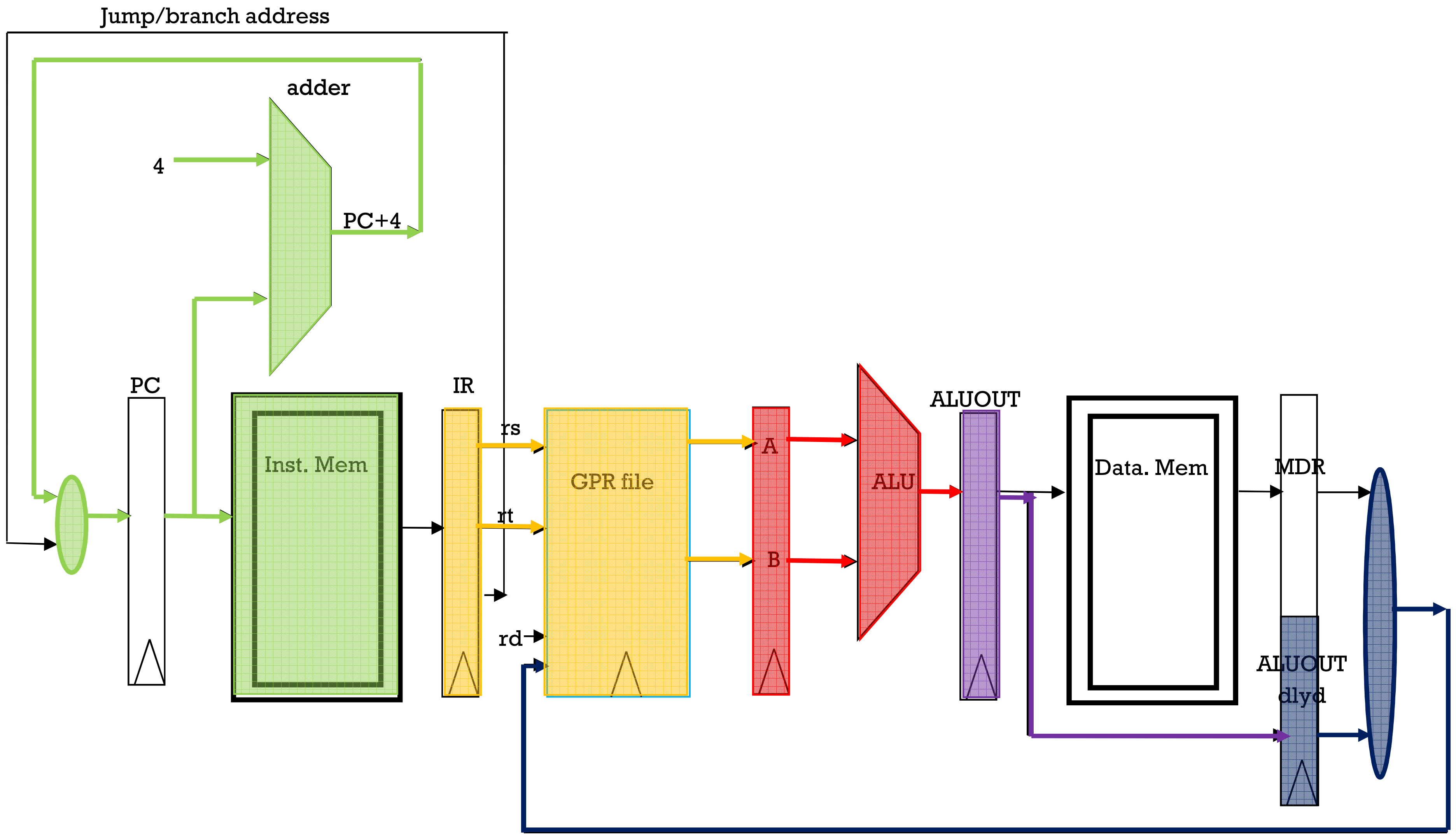
MEM

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

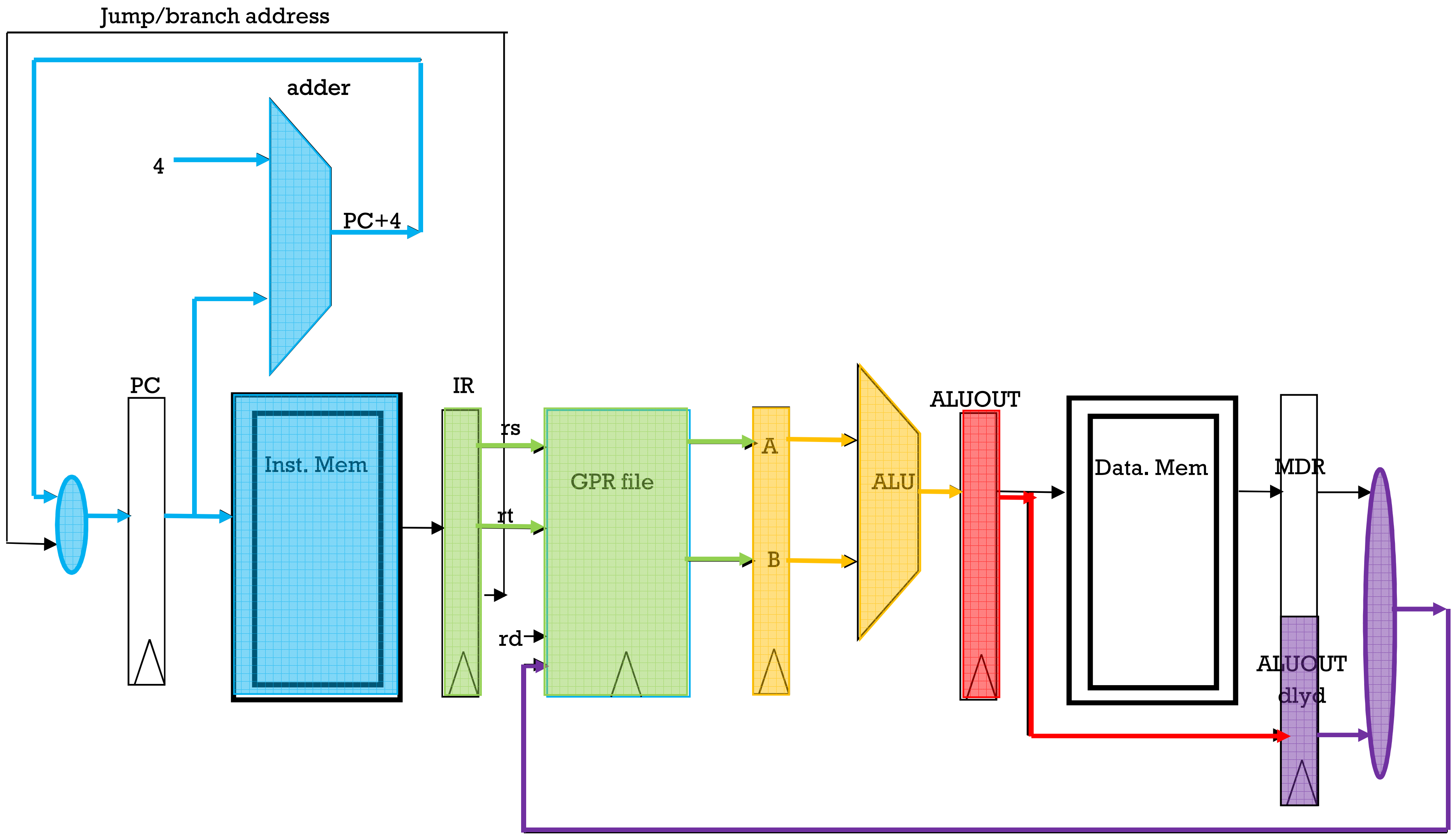
MEM

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

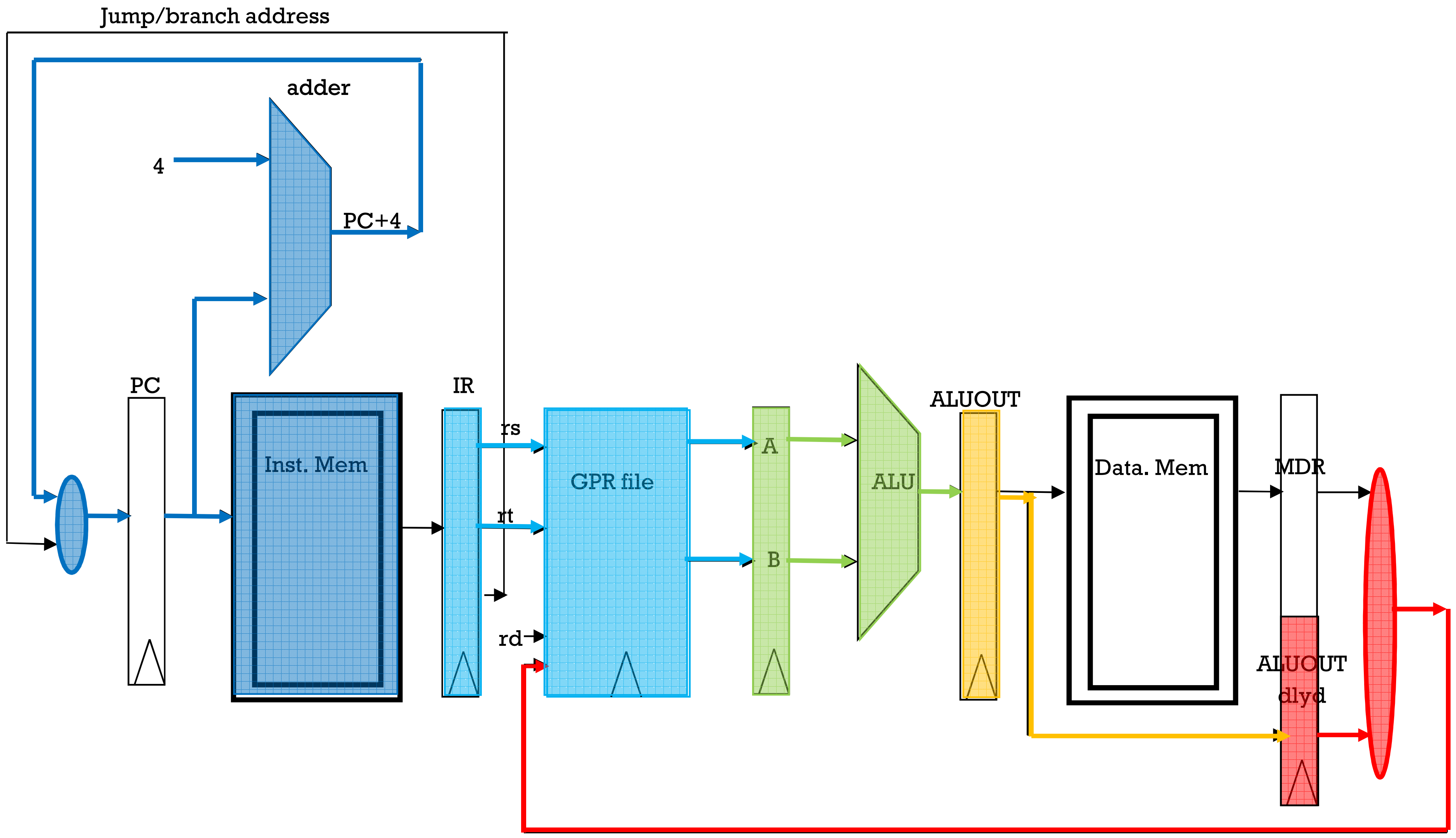
MEM

In Rtype – wait 1 ck

WB – write back

Rd = ALUOUT

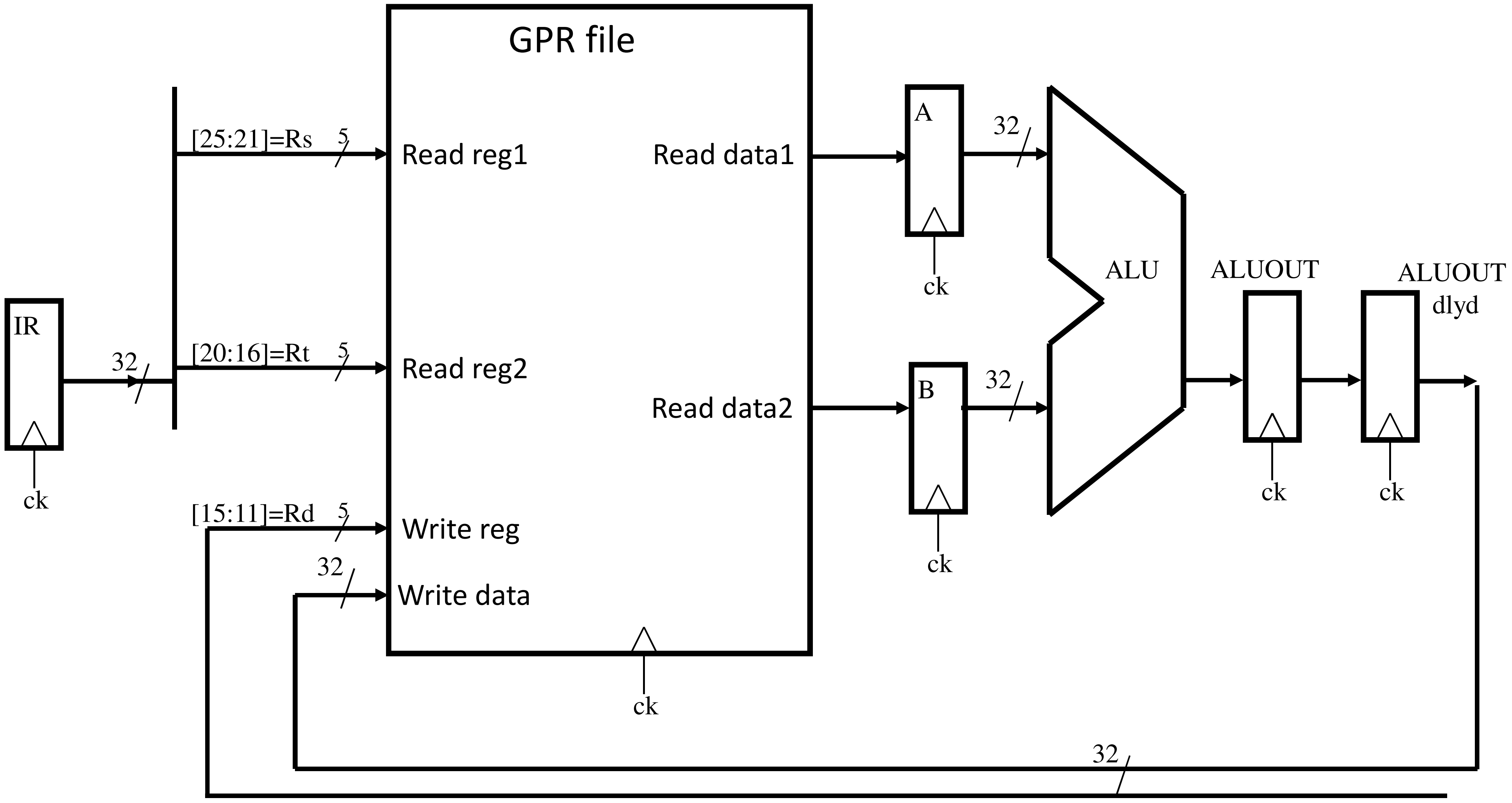
Performing Rtype inst.



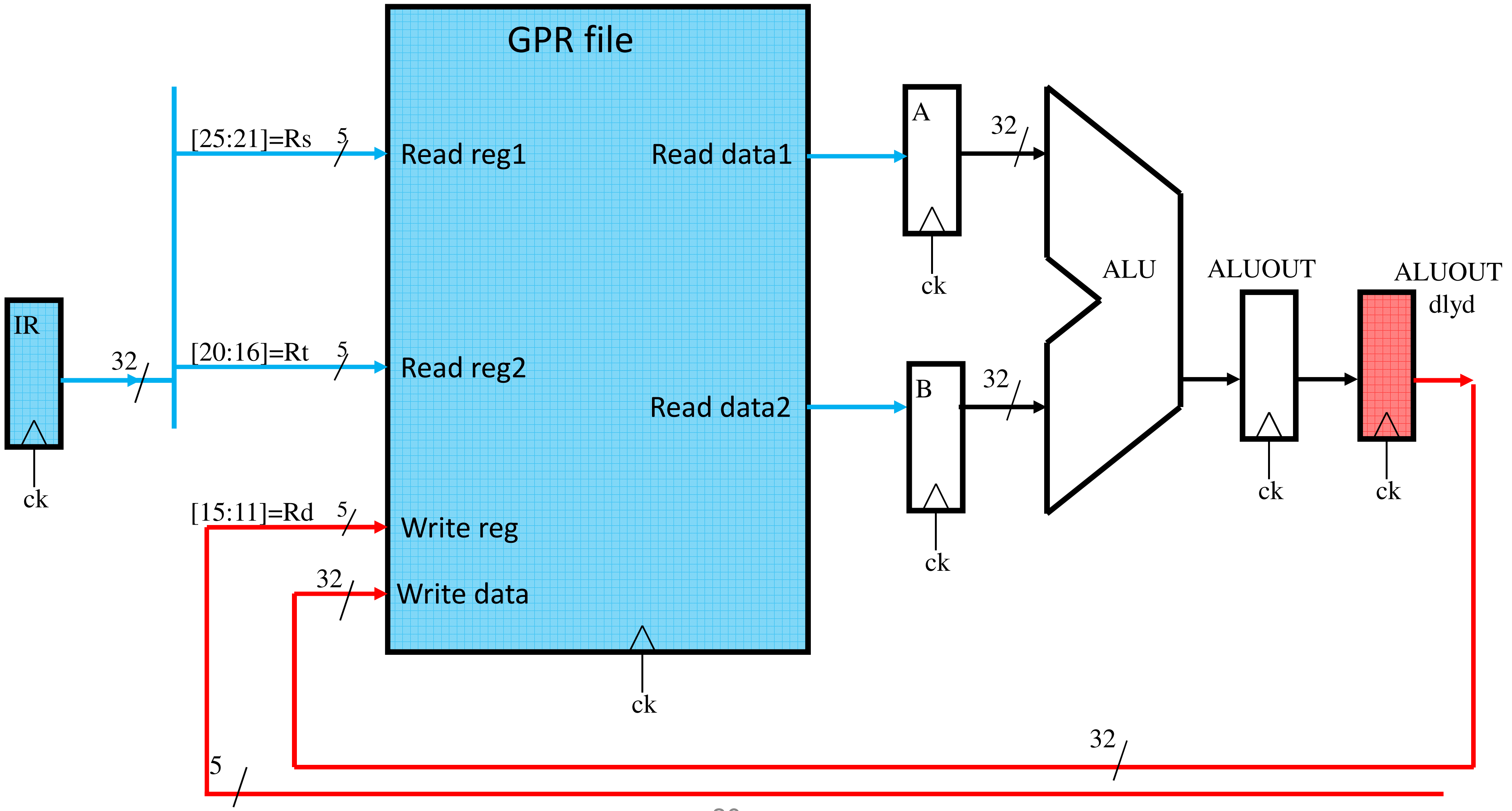
IF – Inst. Fetch
 	
IR = Imem[PC]
PC = PC+4
	ID – Inst. Dec.
	A = rs , B = rt
	
	(& decode control signals)
		EX– Execute
		
		ALUOUT = A op B
			MEM
			
			In Rtype – wait 1 ck
				WB – write back
				
				Rd = ALUOUT

The GPR File

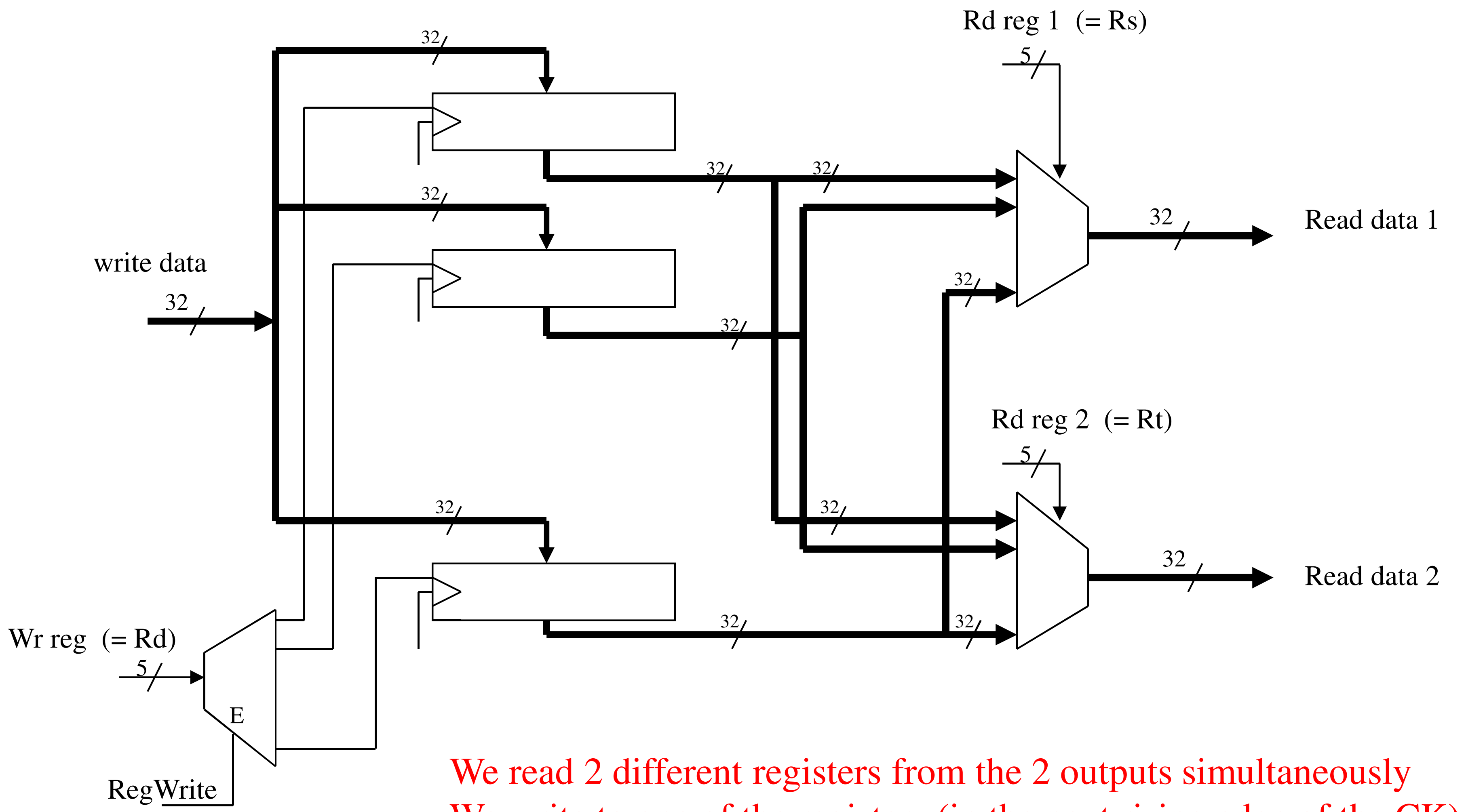
Functional diagram



Functional diagram

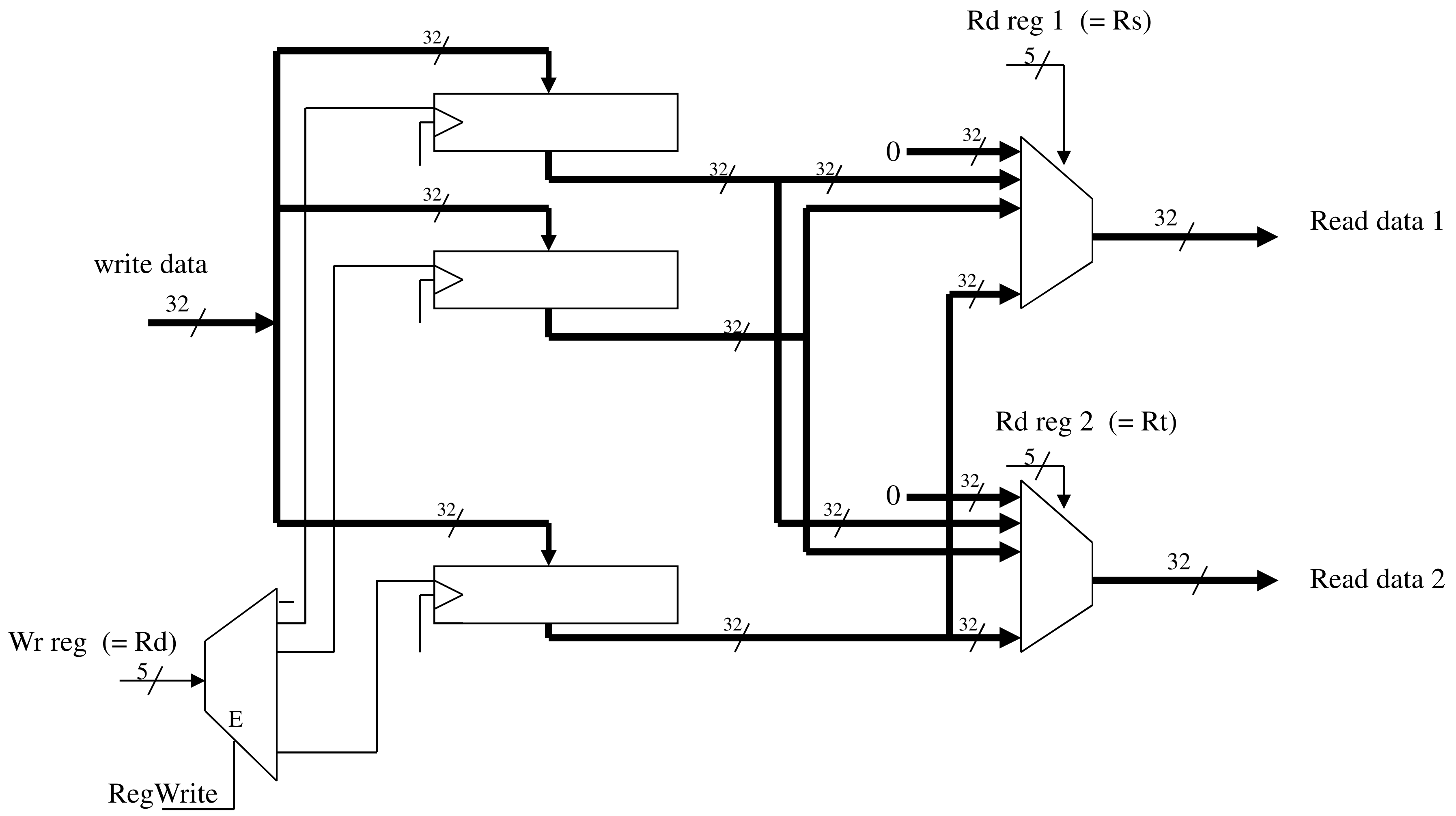


One possible structure of the Register File

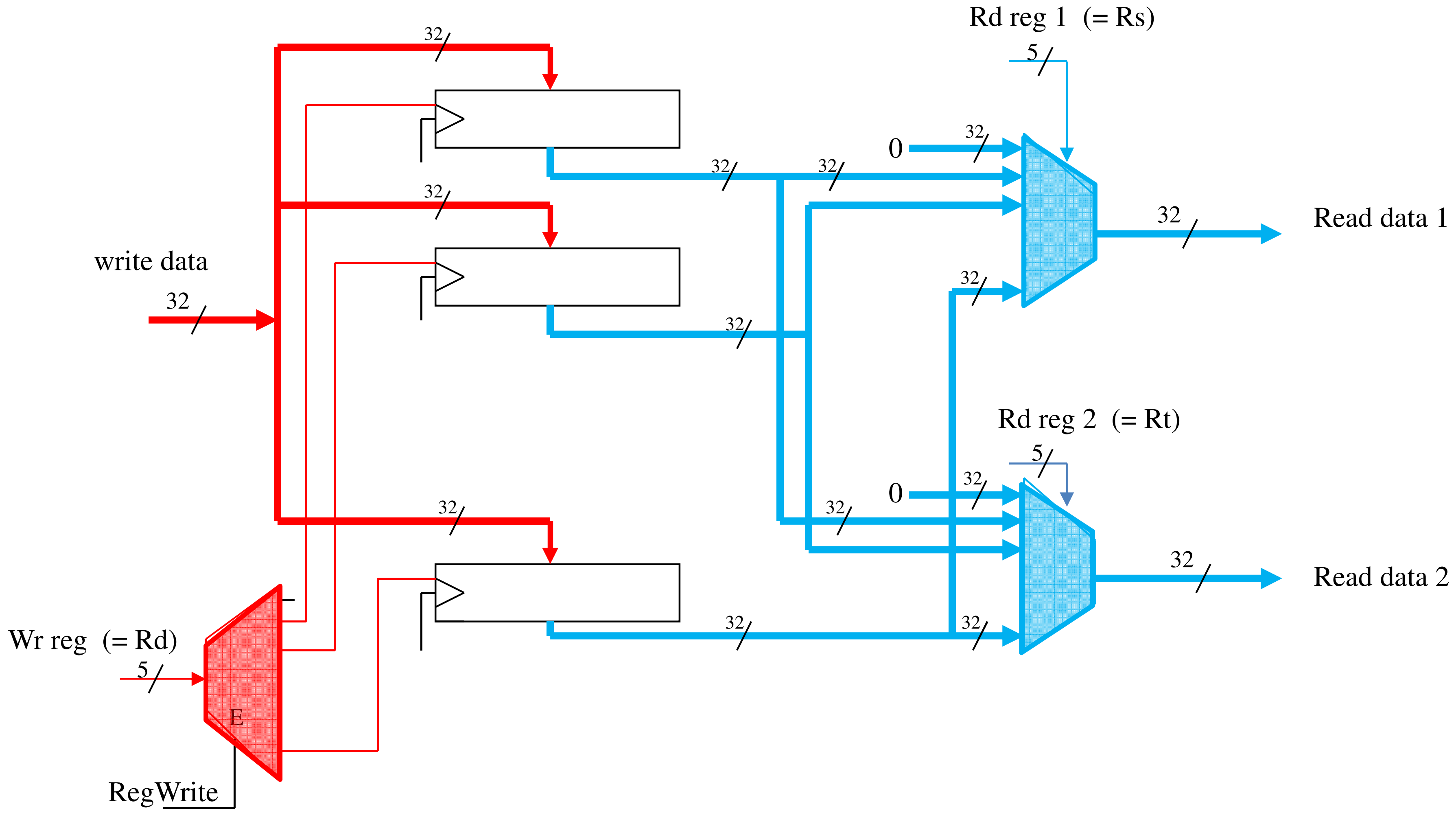


We read 2 different registers from the 2 outputs simultaneously
We write to one of the registers (in the next rising edge of the CK).

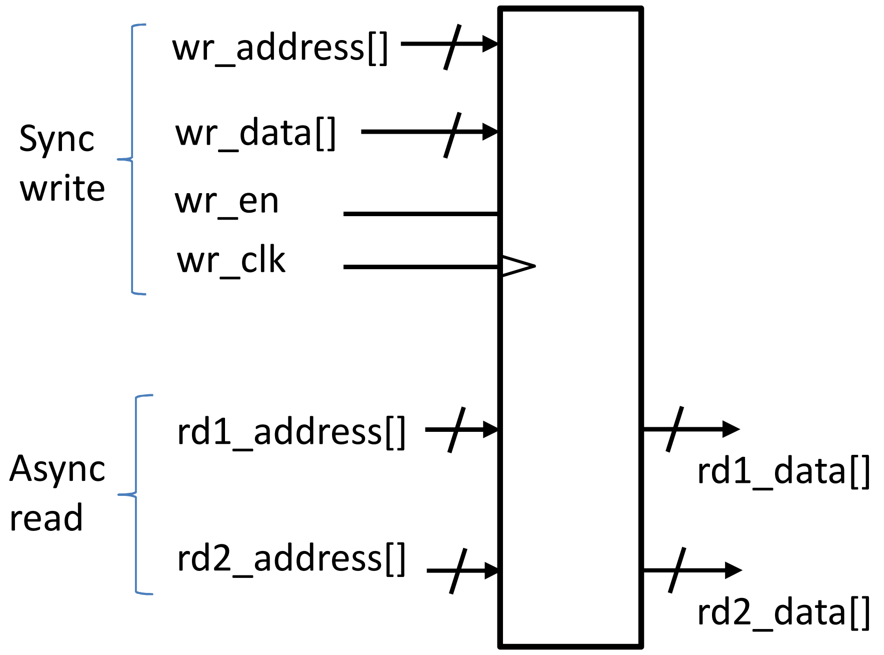
One possible structure of the Register File



One possible structure of the Register File



Dual port async read memory - I



Data will be written to this memory on the rising edge of wr_clk if wr_en is '1'
In that case the wr_data will be written into wr_address in the memory

Data can be read at all times (async read) from rd1_address. It appears at rd1_data output.

An additional read can be done at the same time (also async read) from rd2_address.
That data appears at the rd2_data output.

You get a single_port_memory.vhd file and a dual_port_memory.empty file and need to convert the single port memory to a dual port one.

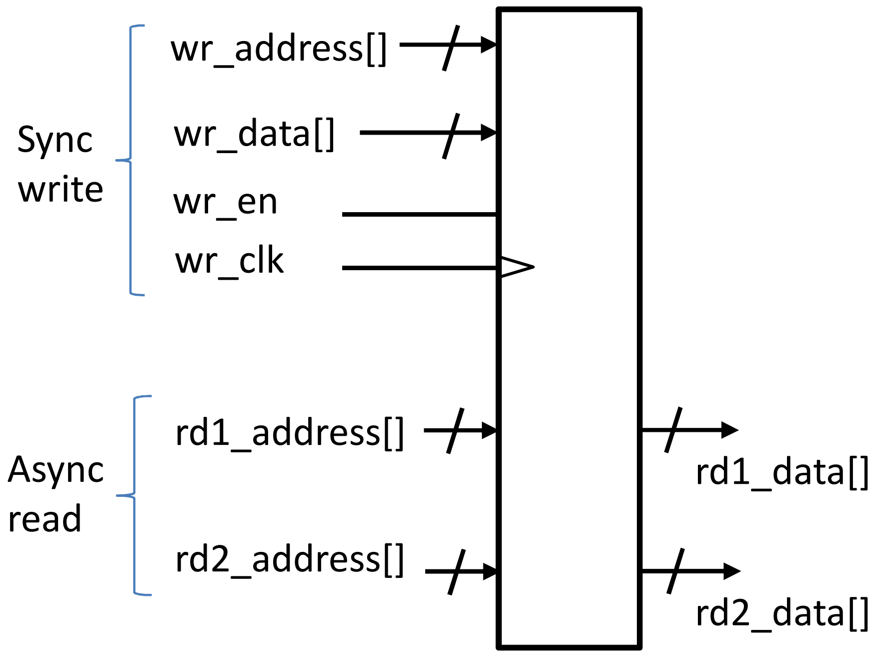
Dual port async read memory - II

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity dual_port_memory_no_CK_read is
GENERIC(
      width :  integer :=32;
      depth :  integer :=32  );

port (
     wr_address    :  in   INTEGER range depth-1 downto 0;
     wr_data       :  in   STD_LOGIC_VECTOR(width-1 downto 0);
     wr_clk        :  in   STD_LOGIC;
     wr_en         :  in   STD_LOGIC;
     --
     rd1_address  :  in   integer range depth-1 downto 0;
     rd1_data      :  out   std_logic_vector(width-1 downto 0);
     --
     rd2_address   :  in   integer range depth-1 downto 0;
     rd2_data       :  out  std_logic_vector(width-1 downto 0)   );

end entity dual_port_memory_no_CK_read;
```



In this device we would like to choose the data width (in bits) and the memory depth (in addresses) when we use this device.

For that we use the **GENERIC** statement.

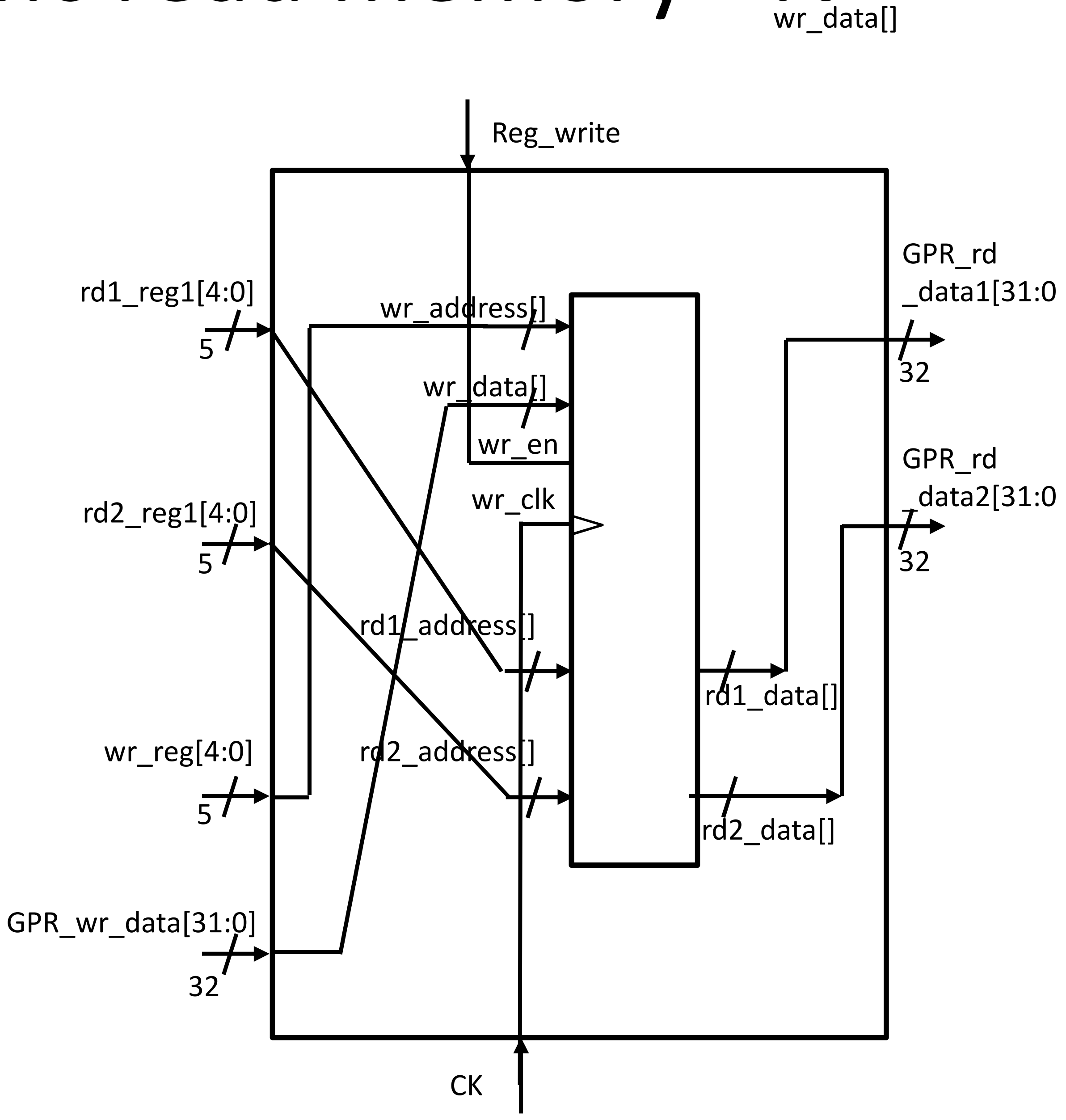
We also give default values

Dual port async read memory - IV

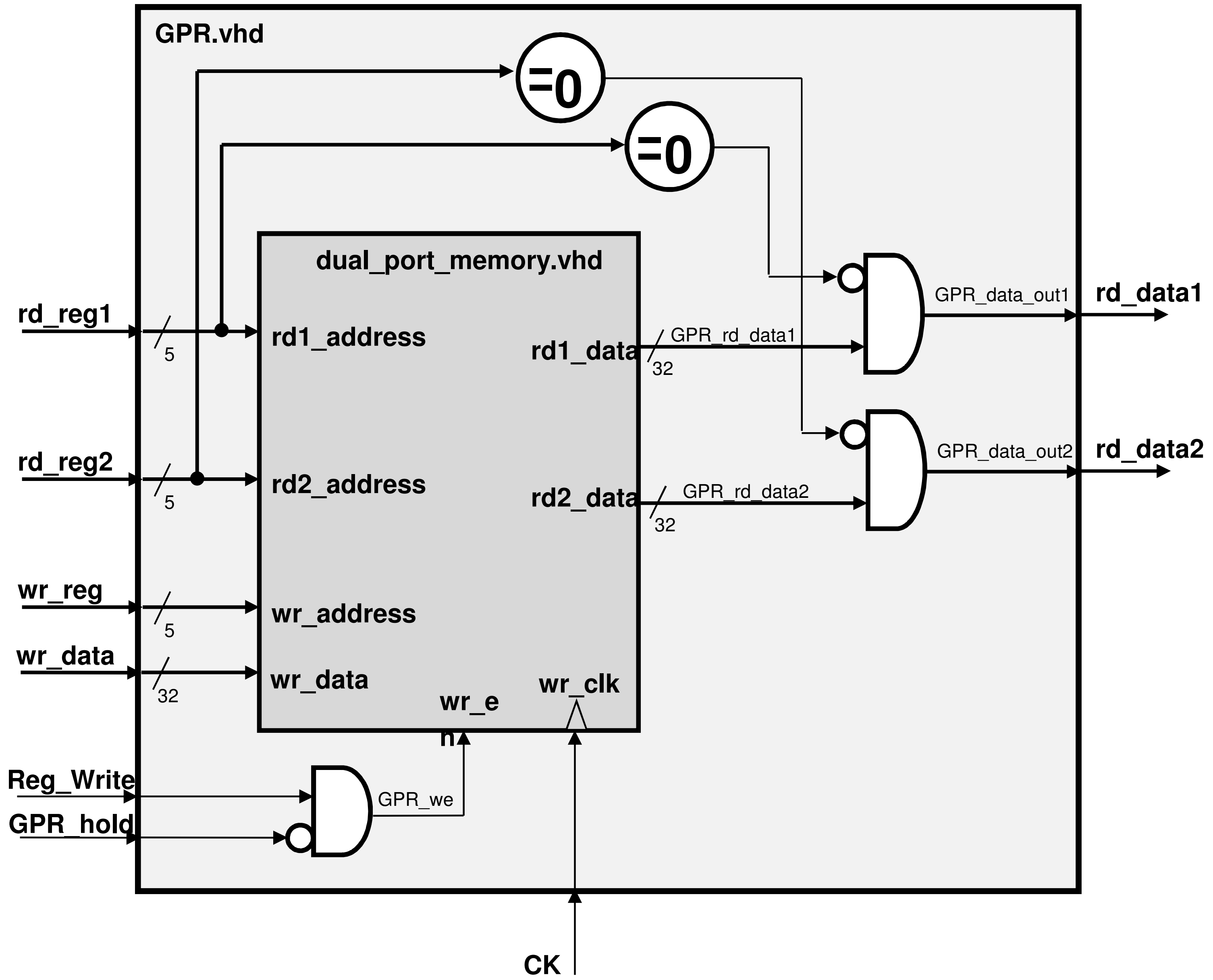
```
-- connecting the GPR  to the memory
GPR_file : dual_port_memory_no_CK_read
generic map (32, 32)
port map(
wr_address    =>     conv_integer(wr_reg),
wr_data       =>     GPR_wr_data,
wr_clk        =>     CK,
wr_en         =>     Reg_Write,
rd1_address   =>     conv_integer(rd_reg1),
rd1_data      =>     GPR_rd_data1,
rd2_address   =>     conv_integer(rd_reg2),
rd2_data      =>     GPR_rd_data2
);
```

The General Purpose Register file we build has 32 addresses of 32 bits each (width=32, depth=32).

Note that the addresses we connect to the memory are converted from STD_LOGIC_VECTORs to INTEGERs



You get GPR.empty file and need to complete the design including making sure That register #0 will always read the value 0.

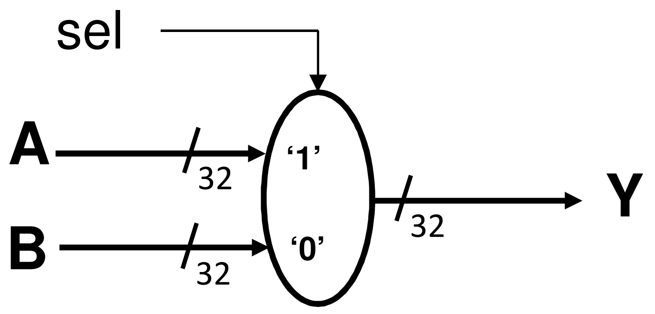


$Y = A \cdot sel + B \cdot \overline{sel}$ if A is ‘0’-s, it becomes $Y = B \cdot \overline{sel}$

The following VHDL code:

```
process (A, B, sel)
begin
    if sel='1' then
          Y <= A;
    else
          Y <= B;
    end if;
end process;
```

creates a 2→1 mux:

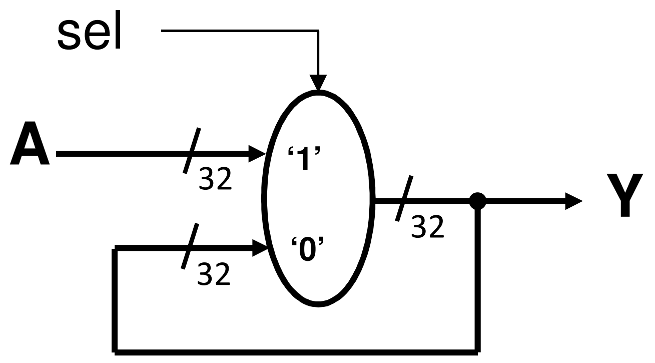


Y=A· sel + **B·** $\overline{\text{sel}}$

The following VHDL code:

```
process (A, B, sel)
begin
    if sel='1' then
          Y <= A;
    end if;
end process;
```

creates a latch:



Since in VHDL the output of a process stays unchanged in cases we do not specify in the "IF"

Now it is your turn!

Thanks for listening!