MIPS Assembly language summary

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Uncondional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

MIPS instructions

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   (Rd=0)
```

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

I-type

```
addi  Rt, Rs, imm     #  Rt=Rs+ imm
lw  Rt, imm(Rs)       #  Rt=M[Rs + imm]
sw  Rt, imm(Rs)       #  M[Rs + imm]=Rt
beq  Rs, Rt, label    #  if Rs==Rt,  PC=PC+4+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
lui   Rt, imm         #  Rt= imm<<16
```

6	5	5	16
OPCODE	Rs	Rt	imm

j-type

```
j     imm             #  PC= imm*4
jal    imm            #  PC= imm*4, $31=PC+4
```

6	26
OPCODE	26 bit imm

1) Description of the Fetch unit

Here we design the Fetch Unit of a pipelined MIPS CPU.

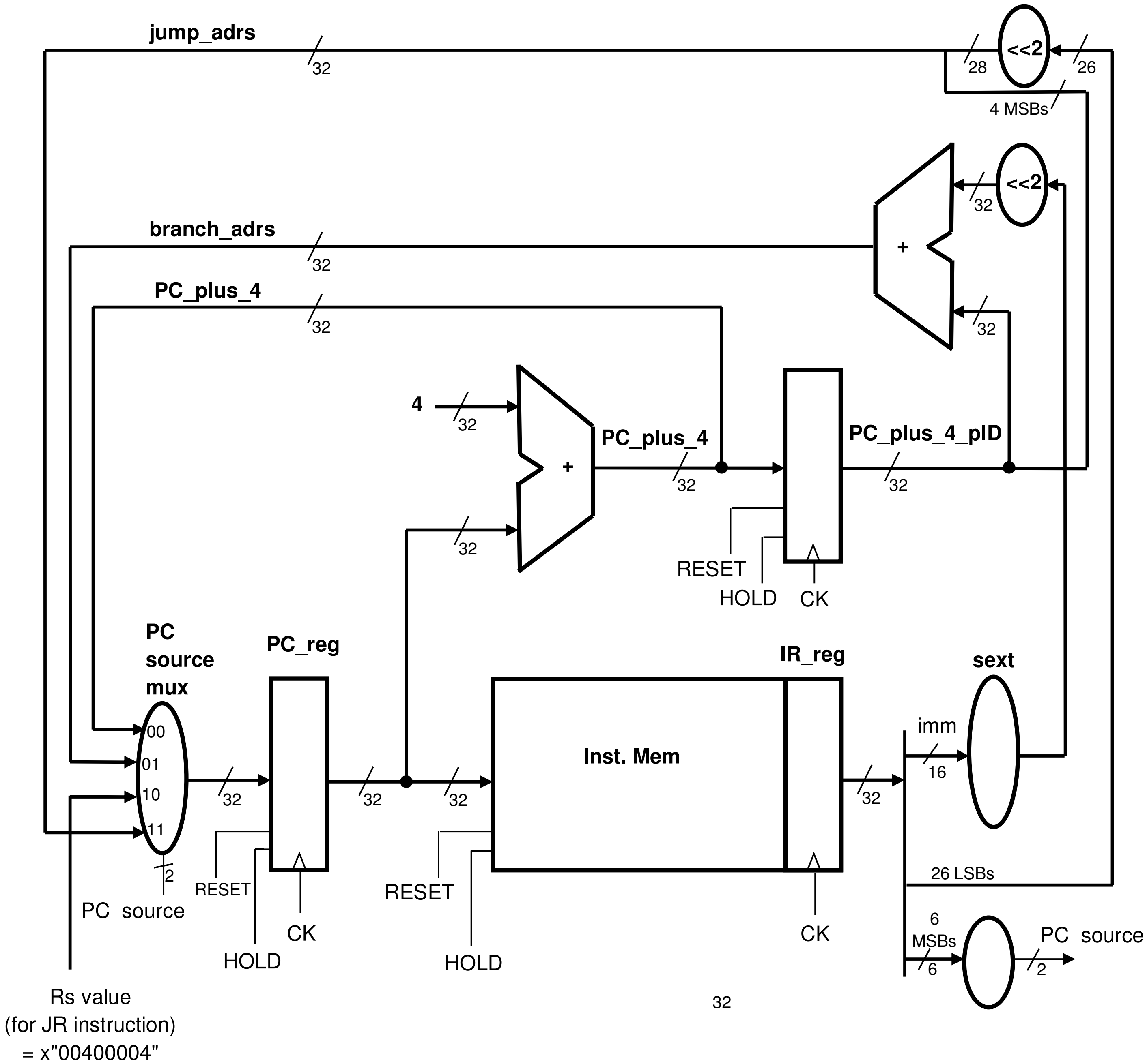


Fig. 1 – The Fetch Unit

The Fetch Unit is the part of the CPU that fetches the instruction from the Instruction Memory (IMem) into the Instruction Register (IR_reg). It also handles jumps and branches. The Fetch Unit's main components are the PC register (PC_reg) and IMem. The PC_reg is a 32 bit register that advances by 4 in every clock. Thus we should have also a 32 bit Adder that adds 4 to the current PC_reg value. In order to jump or branch, we need to input the jump address or branch address to the PC register. Thus, we have a multiplexer at the input of the PC register. This is depicted in Fig.1 above

a. Names & definition of signals inside the Fetch Unit

You must use these exact signal names in your design.

1. PC_reg – a 32 bit register. When RESET is '1', the PC_reg value becomes 0x400000. (All other registers and FFs will be cleared by RESET='1').
2. PC_plus_4 – a 32 bit signal that has the PC_reg value + 4.
3. 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.
4. 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).
5. jump_adrs – a 32 bit signal made of PC_plus_4_pID[31:28] & 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.
6. 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 a jr (jump register) instruction is performed.
7. 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 (Rs value for jr instruction) and when "11", PC_reg is loaded with the jump_adrs.
   The PC_source signal is created by a decoder looking at the opcode field of the instruction residing in the IR_reg.
8. 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).
9. imm – the 16 LSBs of IR_reg
10. sext_imm – sign extension of imm to 32 bits
11. opcode – the 6 MSBs of IR_reg. We sould determine the PC_source value according to the instruction opcode (j,jal-11, beq,bne-01,jr -10, any other instruction-00).
12. HOLD – This signal is meant to freeze all registers when it is "1". It will be used later for running the design in a single clock mode. At that mode this signal will be "1" all the time except for the clock cycles in which we want to perform a single clock "step". This means that all of the registers should have a HOLD input. The IMem itself and its output register (the IR) already support that signal.

1) <u>The Rtype only MIPS CPU and its main components</u>

We would like to design part of the MIPS CPU which is capable of running simple programs with Rtype instructions only. There are 3 main parts involved. These are the Fetch Unit from HW2, the GPR File and the MIPS ALU.

In this homework/lab exercise we will design the GPR File and the MIPS ALU. In the next exercise we will tie the GPR File, the MIPS ALU and the Fetch unit together to form an Rtype MIPS CPU.

Below we see a simplified drawing of the Rtype MIPS CPU.

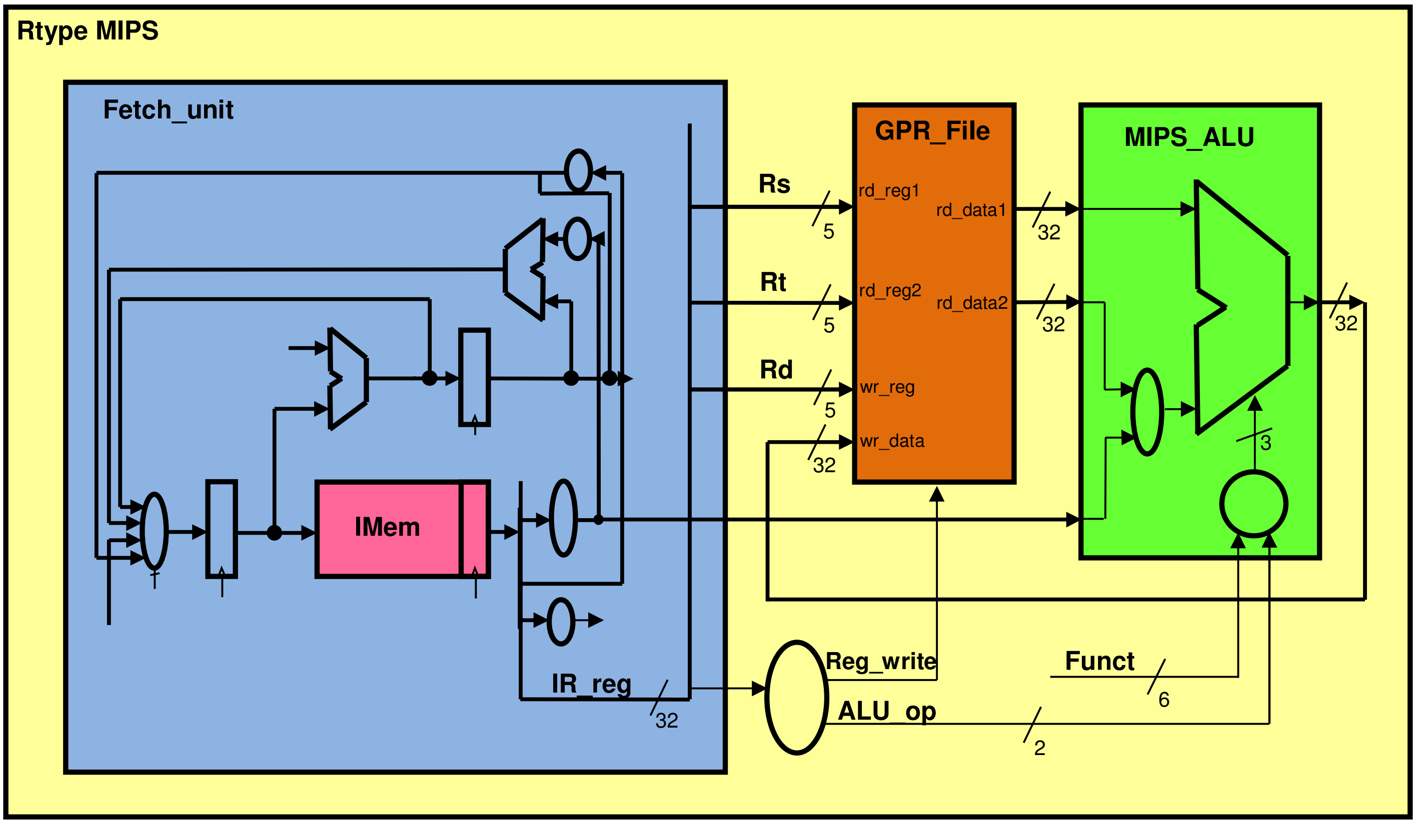


Fig. 1 – The Rtype only MIPS CPU – a simplified drawing

2) <u>GPR – design & simulation</u>

The inside of the GPR File is made of a dual port memory. That memory does not have a register at its output as we had in the IMem we used in the Fetch Unit of HW2. Only the writing process of the memory is triggered by the rising edge of the clock. When “wr_en”=’1’ and there is a rising edge in “wr_clk”, then the “wr_data” is written into the “wr_address” location of the memory. The reading from the dual port memory is a combinationl process.

Note that “wr_address” and “rd_address” are integers, and if your address signals is a STD_LOGIC_VECTOR signal you need to use the function conv_integer(your signal vector name) to convert the STD_LOGIC_VECTOR value to integer in order to set a value to the wr_address or rd_address.

We give you a vhd file called **single_port_memory.vhd** and you need to manipulate it to become a **dual_port_memory.vhd**. The skeleton of the **dual_port_memory.vhd** is given in the **dual_port_memory.empty** file so that you will use the signal names we decided on.

The outside to the GPR File is described in the skeleton file **GPR.empty**. In this file we implement the following:

2.1) Although the dual port memory we use has address 0 and so we can write data into that address and read data from that address, we will make sure that when we read from read_reg1=0, we will get rd_data1=x”00000000”.

2.2) Similarly, when reading from read_reg2=0, we will get rd_data2=x”00000000”.

2.3) We will add a GPR_hold input to the GPR file. When this input is ‘1’ there should not be a write operation at the rising edge of the clock even if the RegWrite signal is ‘1’.

So you need to prepare the files:

- **dual_port_memory.vhd** – that describes the dual port memory in which only the writing is synchronous (activated by the rising edge of the clock)
- **GPR.vhd** – that “wraps” the dual_port_memory component of 32 addresses of 32 bits each and performs what was requested in 2.1 and 2.2 above

To ease the design for you the GPR.vhd content is depicted in Fig. 2 below.

Now you can run a simulation and check your design with the additional three files of:

- **SIM_GPR_TB.vhd** - the TestBench file we prepared for you ahead of time
- **SIM_GPR_TB_data.dat** - the TestBench testing data file we prepared ahead of time
- **SIM_HW3_GPR_filenames.vhd** – In this file we specify the path of the data files used in simulation

With these 5 files you need to run the simulation and verify your design works fine. Note that you need to update the **SIM_HW3_GPR_filenames.vhd** with the actual path of the **SIM_GPR_TB_data.dat** file.

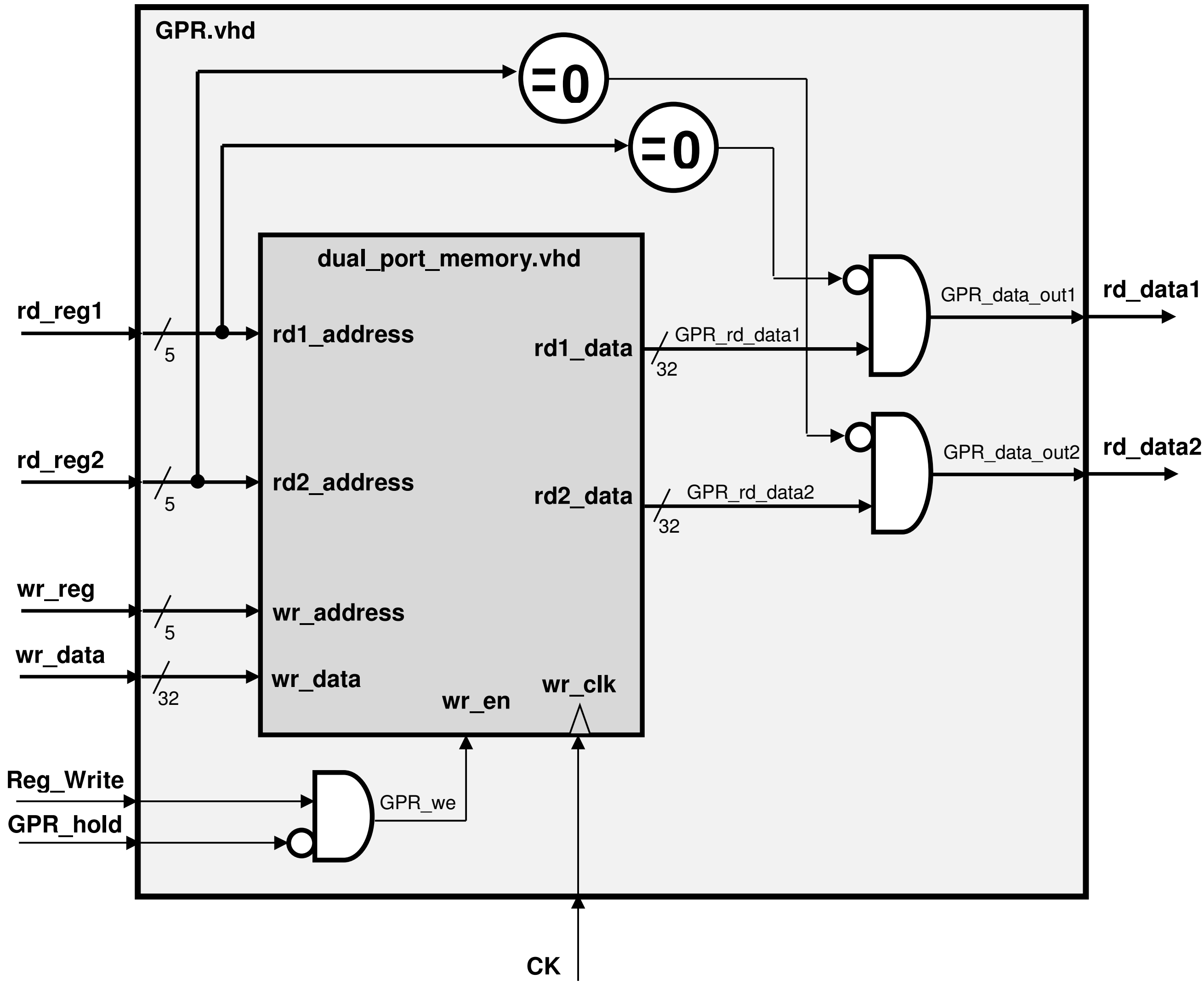


Fig. 2 – The inside of the GPR.vhd

You should submit a zip file of the entire GPR_File simulation project – see detailed instructions at section 4 of this document.

Also you need to attach a doc file with screen captures describing the simulation you made.
All i/o signals of GPR entity should be presented in the screen capture. Show the 2nd session of writing into the GPR File (clock cycles 46 to 55 = 920ns to 1100ns) and make sure that the values of all signals are readable. Explain what is seen in the rd_data1 and rd_data2 outputs of the GPR_File in these clock cycles.

3) MIPS ALU – design & simulation

The MIPS ALU is a combinational circuit. No FFs are involved. We "added" to the ALU also the ALU_src_B multiplexer and also the logic control that issues the ALU_cmd signal. The ALU_cmd is a 3 bit signal vector which determines what is the calculation done by the ALU.
If the ALU_cmd is "010", the ALU performs an addition. If ALU_cmd is "110", the ALU performs a subtraction. Here is the list of operation done by the ALU according to the ALU_cmd bits:

- ALU_cmd="000" => A and B
- ALU_cmd="001" => A or B
- ALU_cmd="010" => A + B
- ALU_cmd="011" => A xor B
- ALU_cmd="100" => A nand B - not used
- ALU_cmd="101" => A nor B - not used
- ALU_cmd="110" => A - B
- ALU_cmd="111" => SLT. 1 if A<B, 0 if not. A & B are considered 2's complement numbers

The logic that drives the ALU_cmd gets the 2 bit signal vector called ALUOP. When ALUOP="00", the ALU performs addition. When ALUOP="01", the ALU performs subtraction. When ALUOP="10", the ALU operation is determined by the 6 bit vector called Funct (function) that comes from the 6 LSBs of the IR reg. Here is the list of the Funct codes:

- Funct=" 100000" => ADD
- Funct=" 100010" => SUB
- Funct=" 100100" => AND
- Funct=" 100101" => OR
- Funct=" 100110" => XOR
- Funct=" 101010" => SLT

In all other cases we request to perform ADD.

The ALU_src_B mux selects what will be fed into the B input of the ALU. If ALUsrcB='0', we input the B_in data into the ALU B input. If ALUsrcB='1', we input the sext_imm data into the ALU B input.

We prepared a **MIPS_ALU.empty** file for your convenience.
You need to add all of the logic described above. When done, you should run a simulation using the **MIPS_ALU.vhd** file and the additional three files of:

- **SIM_MIPS_ALU_TB.vhd** - the TestBench file we prepared for you ahead of time
- **SIM_MIPS_ALU_TB_data.dat** - the TestBench Data file we prepared ahead of time
- **SIM_HW3_ALU_filenames.vhd** – In this file we specify the path of the data files used in simulation

With these 4 files you need to run the simulation and verify your design works fine. Note that you need to update the **SIM_HW3_ALU_filenames.vhd** with the actual path of the **SIM_MIPS_ALU_TB_data.dat** file.

You should submit a zip file of the entire MIPS_ALU simulation project– see detailed instructions at section 4 of this document. Also you need to attach a doc file with screen captures describing the entire simulation you made – till 1200 ns. All i/o signals of the MIPS_ALU entity should be presented in the screen capture.

4) HW3 report

You should submit a single zip file for the Simulation of both entities. It should have three directories/folders. The first is called **GPR_File**, the 2nd is called **MIPS_ALU**, the 3rd is called **Disassembly**.

In the **GPR_File** directory you will have the following 3 sub-directories:

- **GPR_File_Src** - with all of your simulation sources
- **GPR_File_Sim** - with the simulation project
- **GPR_File_Docs** - Add a doc file with screen capture of the simulation showing the waveforms of the TB signal and the Console window. All i/o signals of GPR entity should be presented in the screen capture. Show the 2nd session of writing into the GPR File (clock cycles 46 to 55 = 920ns to 1100ns) and make sure that the values of all signals are readable. Explain in detail what do we see in rd_data1 and rd_data2 in these 10 clock cycles. The first few lines in the report will have your ID numbers (names are optional).

In the **MIPS_ALU** directory you will have the following 3 sub-directories:

- **MIPS_ALU_Src** - with all of your simulation sources
- **MIPS_ALU_Sim** - with the simulation project
- **MIPS_ALU_Docs** - Add a doc file with screen capture of the simulation showing the waveforms of the TB signal and the Console window. The screen captures should have the entire simulation you made (from its start to its end – till 1200 ns), and all of the MIPS_ALU i/o signals. No need to see the values of the signals, just the total picture and the console with a "Test Pass" message. The first few lines in the report will have your ID numbers (names are optional).

In the **Disassembly** directory you should have a doc file in which you disassemble a MIPS binary code and some explanations (answer questions).
See the questions in the file 18.1_MIPS_binary_code_for_disassembly_v4.docx

Note that the binary MIPS code you need to disassemble is the program we will be using in HW4. You need this disassembled code to understand what is done in HW3. That binary program to be disassembled appears in a Word file called 18.1_MIPS_binary_code_for_disassembly_v4.docx and also in a text file called 18.2_MIPS_binary_code_for_disassembly_v4.txt.

Use this file and add your disassembled code. See the appendix at the end of the document for MIPS instructions coding. Also explain in detail what is done by this code. Also explain how this code tests the GPR_file and ALU parts of a MIPS CPU.

At the end of this assignment you will have the necessary building blocks for our next assignment, HW4 – the "Rtype" MIPS CPU.

Enjoy the assignment !!

5) <u>Appendix A – MIPS instructions coding</u>

a. Codes of the Opcode fields - IR(31 downto 26)

sw	=[101011]=43
lw	=[100011]=35
lui	=[001111]=15
ori	=[001101]=13
addi	=[001000]=8
beq	=[000100]=4
bne	=[000101]=5
j	=[000010]=2
jal	=[000011]=3
R-type	=[000000]=0

b. Function field codes for RType instructions – IR(5 downto 0)

add	=[100000]=32
sub	=[100010]=34
and	=[100100]=36
or	=[100101]=37
xor	=[100110]=38
slt	=[101010]=42
jr	=[001000]=8

Rs, Rt and Rd fields have a 5 bit binary number of the register (0-31)

1) The Rtype MIPS CPU and its main components

In HW3 we stated that we want to design part of the MIPS CPU which is capable of running simple programs with Rtype instructions only. There are 3 main parts involved. These are the Fetch Unit from HW2, the GPR File and the MIPS ALU. We built the last two components in HW3.

In this homework/lab exercise we are going to tie the GPR File, the MIPS ALU and the Fetch unit together to form an Rtype MIPS CPU.

Below we see a simplified drawing of the Rtype MIPS CPU we used in HW3.

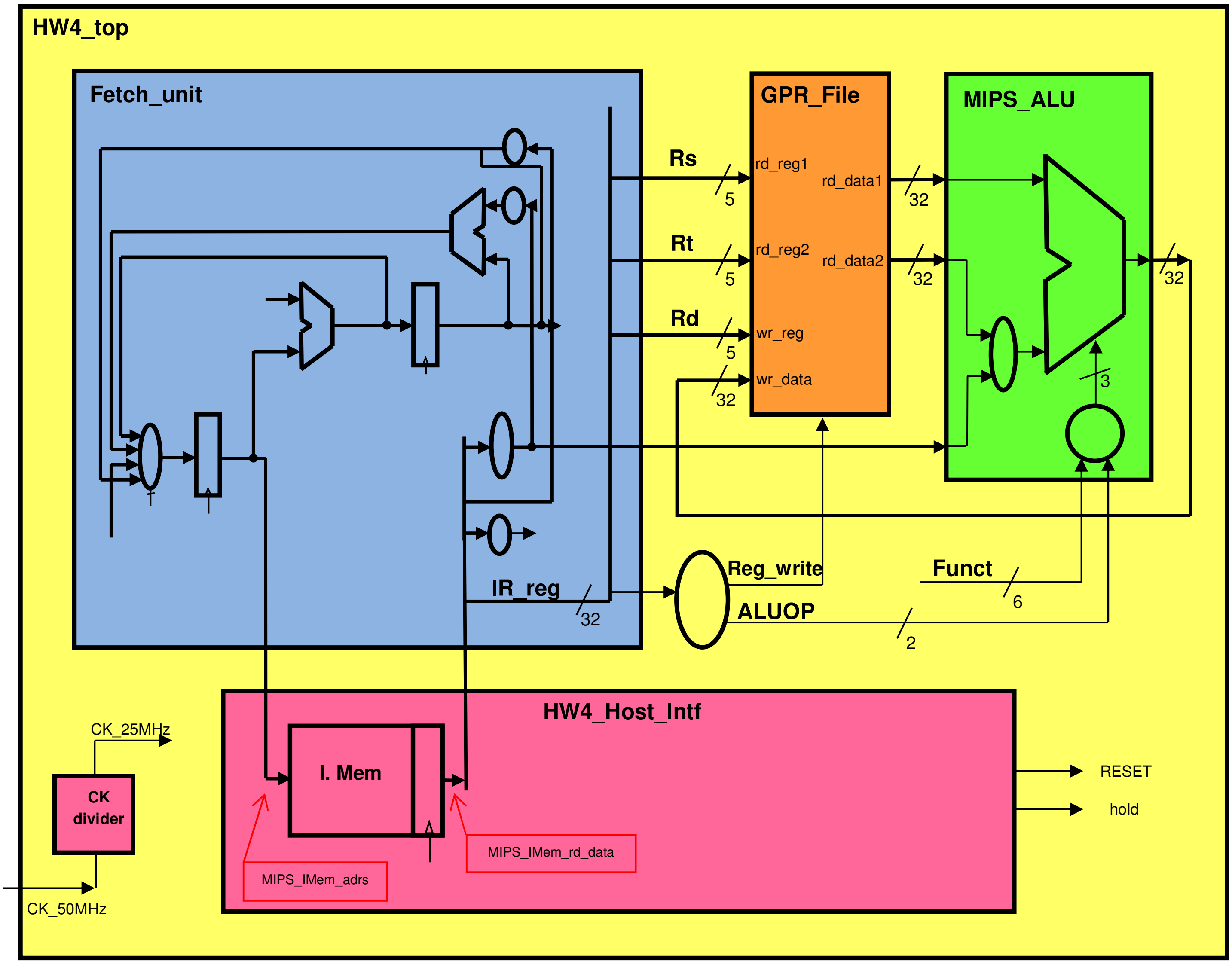


Fig. 1 – The Rtype only MIPS CPU – a simplified drawing

In HW3 we called this CPU the Rtype only MIPS. However, in the Fetch Unit we already have the ability to support jump and branch instructions. Supporting **beq** and **bne** instructions might require some minor additions. In order to make things more interesting, we will also support the **addi** instruction. Thus, this “Rtype” MIPS CPU will start running from address 400000h and preform **Rtype** instructions and also **j**, **beq**, **bne** and **addi** instructions.

Some changes in the Fetch Unit are necessary to “tailor” it into the Rtype MIPS CPU. Our design of the Rtype MIPS CPU resides in the **HW4_top.vhd**.

A more accurate description appears in Figure 2 below.

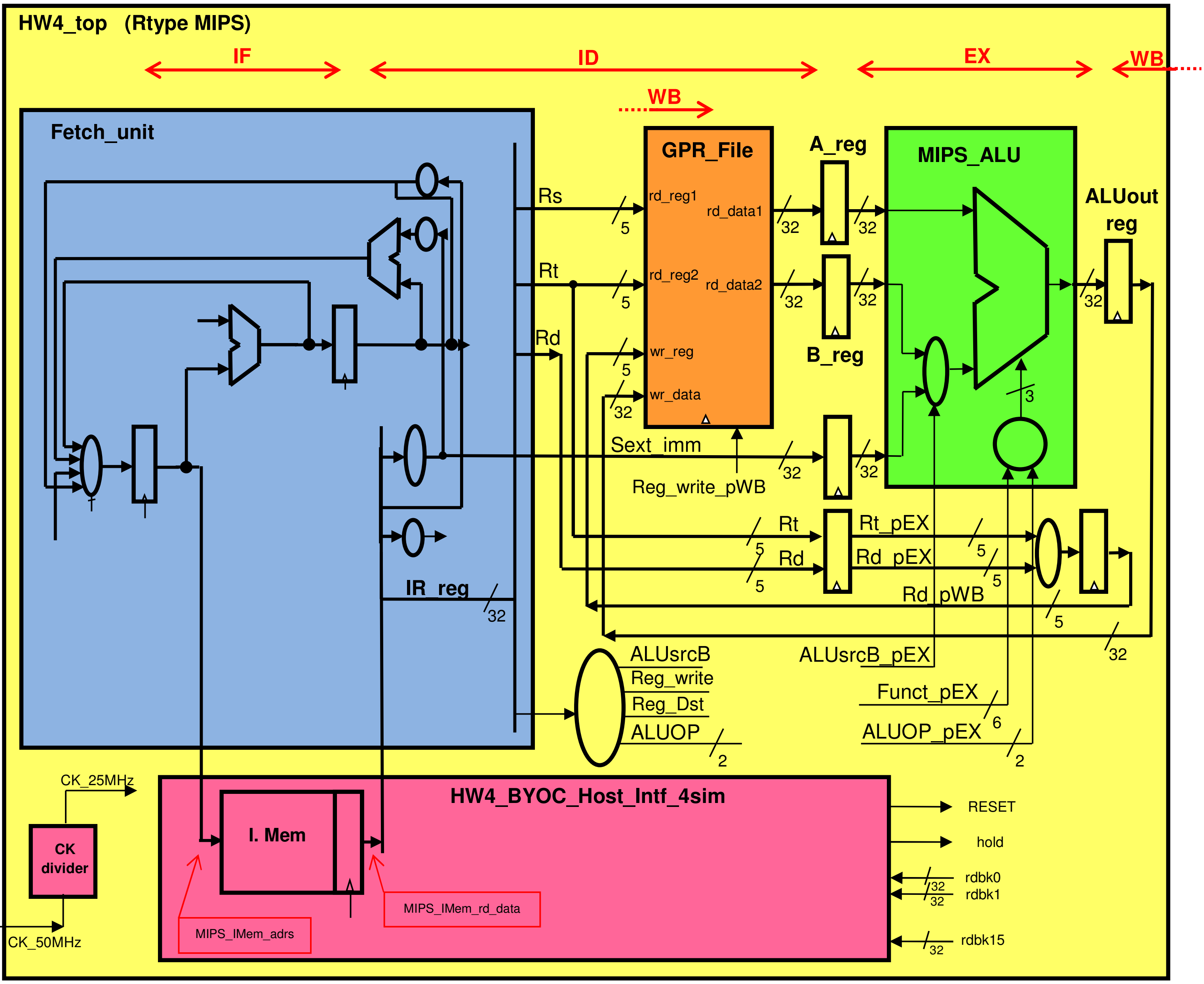


Fig. 2 – The Rtype MIPS or HW4_MIPS CPU

2) HW4 Rtype MIPS CPU – design & simulation

The HW4 Rtype MIPS CPU will have four phases.

- **IF** – Instruction Fetch, which is carried out inside the Fetch Unit producing the instruction in the IR_reg at the rising edge of the clock which ends the IF phase and starts the ID phase.
- **ID** – Instruction Decode, which is the stage in which we do the following:
  - Decode the instruction residing now at the IR_reg and decide what should be done.
    This means, we produce all control signals to be used by that instruction in all phases of this instruction – ID, ED and WB.
  - Read Rs into A_reg and Rt into B_reg

  The rising edge of the clock sampling data into the A_reg and B_reg ends the ID phase and starts the EX phase.
- **EX** – Execute, which is the phase in which the ALU calculates the result of A op B (in **Rtype** instructions) or A+sext_imm (in **addi** instructions). The result is sampled into the ALUout_reg at the rising edge of the clock which ends the EX phase and starts the WB phase.
  In this phase we also select Rs or Rd as the GPR file destination register to be written into in the Write Back phase.
- **WB** – Write Back, which is the final phase of the instruction. If this is an **Rtype** or **addi** instruction, then we write the ALUout_reg value into the GPR file. If this is a **j**, **beq** or **bne** instruction, we do nothing at that stage. The rising edge of the clock sampling data into the GPR File ends the WB phase and completes the instruction.

As explained above, the control signals are created by decoding the instruction residing in the IR_reg at the ID phase. If the control signal is supposed to influence at the EX phase, it must be delayed by 1 clock cycle. If that control signal is supposed to influence at the WB phase, it must be delayed by 2 clock cycles. You will have to handle these timing issues in order to make your design function properly.

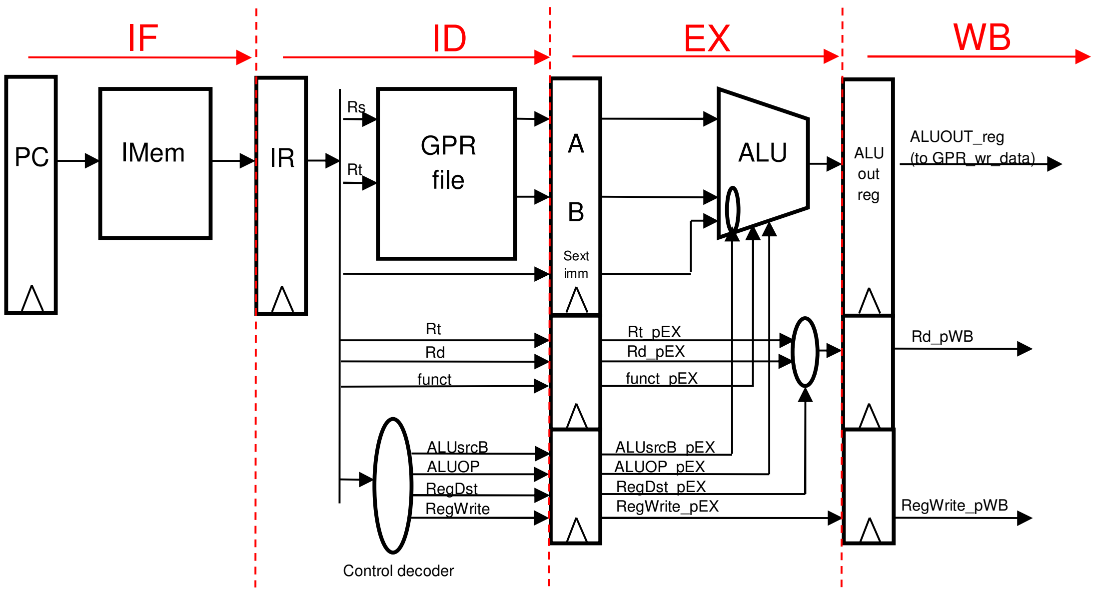


Fig. 2b – The Rtype MIPS control scheme

a. Modifications required in the Fetch Unit

We do the following changes in the Fetch_Unit entity so it will be possible to use it in the HW4_TOPdesign. See Figure 3 on the next page.

We remove all rdbk0-15 output signals from the Fetch Unit. We hope we won't need them since the Fetch Unit is already debugged and the changes we introduce are minor.

Instead we add output signals coming out of the Fetch_Unit that should be used by the rest of the CPU. These *output* signals are:

1. IR_reg_pID - This is a 32 bit signal of the IR_reg (the instruction bits). We added pID to that signal name to indicate it is the IR_reg value at the ID phase.
2. sext_imm_pID - Similarly, this is the 32 bit sext_imm signal we calculate at the ID phase. It is outputted from the Fetch Unit to be used later in the EX phase.
3. PC_reg_pIF - this is the 32 bit PC_reg we use during the IF phase for the Instruction Fetch, i.e., for reading from the IMem. It is outputted from the Fetch Unit to be used for verification purposes only (debugging).

These signals are used in the **HW4_top** entity. They also allow us testing the IR_reg and sext_imm (and the PC_reg) during simulation. Our Fetch_Unit stays the same for simulation & implementation – no changes are required when going from the simulation phase to the implementation phase. Note that for TB purposes we output the CK_out_to_TB, RESET_out_to_TB, HOLD_out_to_TB signals from the **HW4_top_4sim.vhd** which in HW4 is our top component. Therefore, when going from simulation to implementation, we will need to change the **HW4_top** and remove these signals.

Now we add an input signal to the updated Fetch_Unit.

1. We add the Rs_equals_Rt_pID signal that tells us whether to branch in beq (if it is '1') or not (if it is '0'). This signal should come from comparing the two data outputs of the GPR File which resides outside the Fetch_Unit. You should modify the PC_source signal decoder so that the **beq** and **bne** instructions are properly performed. Make sure that theh **addi** instruction is also supported.

The rest of the Fetch_Unit signals are left unchanged. See Fig. 3 below for the updated Fetch_Unit with the new signals in **RED**

When simulating our top file is **HW4_top_4sim.vhd**. In this entity we will use the **BYOC_Host_Intf_4sim.vhd** as our Host Interface circuit having the pre-loaded IMem. For implementation our top vhd file will be renamed to **HW4_top.vhd** and inside it, we will use the **BYOC_Host_Intf.ngc** file. The difference between the two Host_Intf versions is that in the sim version the Host Interface has the program already loaded inside (actually it is loaded at the beginning of the simulation). The implementation version includes the real Host_Intf mechanism allowing us to load a program from the PC, run the design in single clock mode and see the readback signals. The difference between the **HW4_top_4sim.vhd** and the **HW4_top.vhd** will be minor - removal of TB signals.

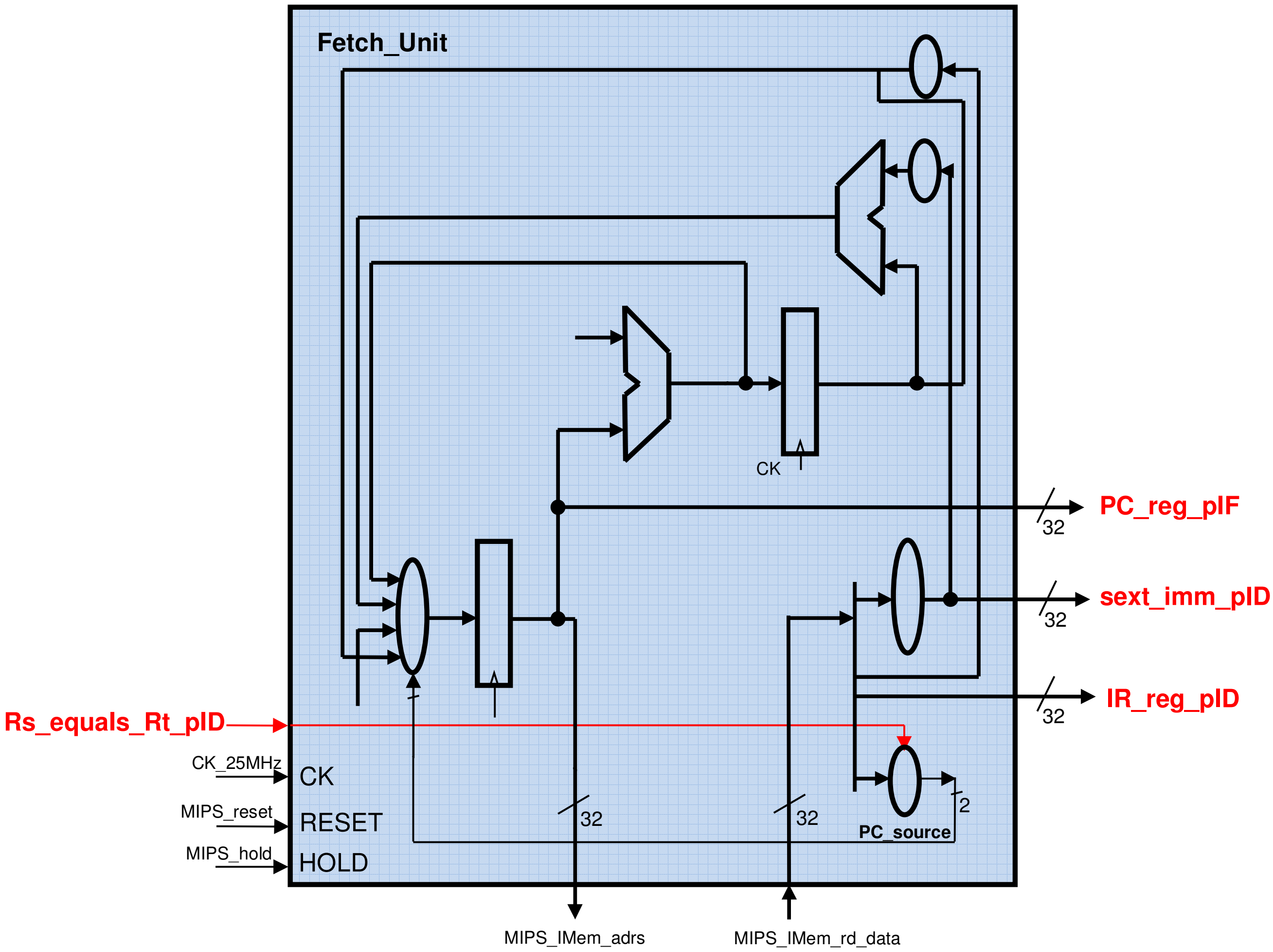


Fig. 3 – The updated Fetch_Unit (new signals – in red)

Note that in the **HW4_top_4sim.empty** file we already connected all of the components (Fetch_Unit, GPR, MIPA_ALU, BYOC_Host_Intf – those are the blue, orange, green and pink parts of Fig. 2). We also defined all of the HW4_top signals (see in section **b** below). Your job is therefore to rename it to **HW4_top_4sim.vhd** and build the missing "logic" in the **HW4_top_4sim.vhd** (which is the yellow part in Fug. 2). That "logic" is made of the registers, FFs and combinational logic forming the Rtype MIPS CPU.

1. The HW5 MIPS CPU and its main components

In this assignment we add lw and sw instructions to the Rtype MIPS CPU we designed in HW4. This means we have to add the Data Memory (DMem) to our design. Following this we will have an almost complete MIPS CPU capable of performing Rtype, addi, j, beq, bne, lw and sw instructions. In our next & final assignment we will complete the CPU by adding jal, jr, lui and ori instructions and add forwarding to enhance the CPU performance.

The DMen we add is located inside the BYOC_Host_Intf component that includes infrastructure allowing loading data into the IMem and DMem memories.

Below we see a simplified drawing of the HW5 MIPS CPU we are going to build in this assignment.

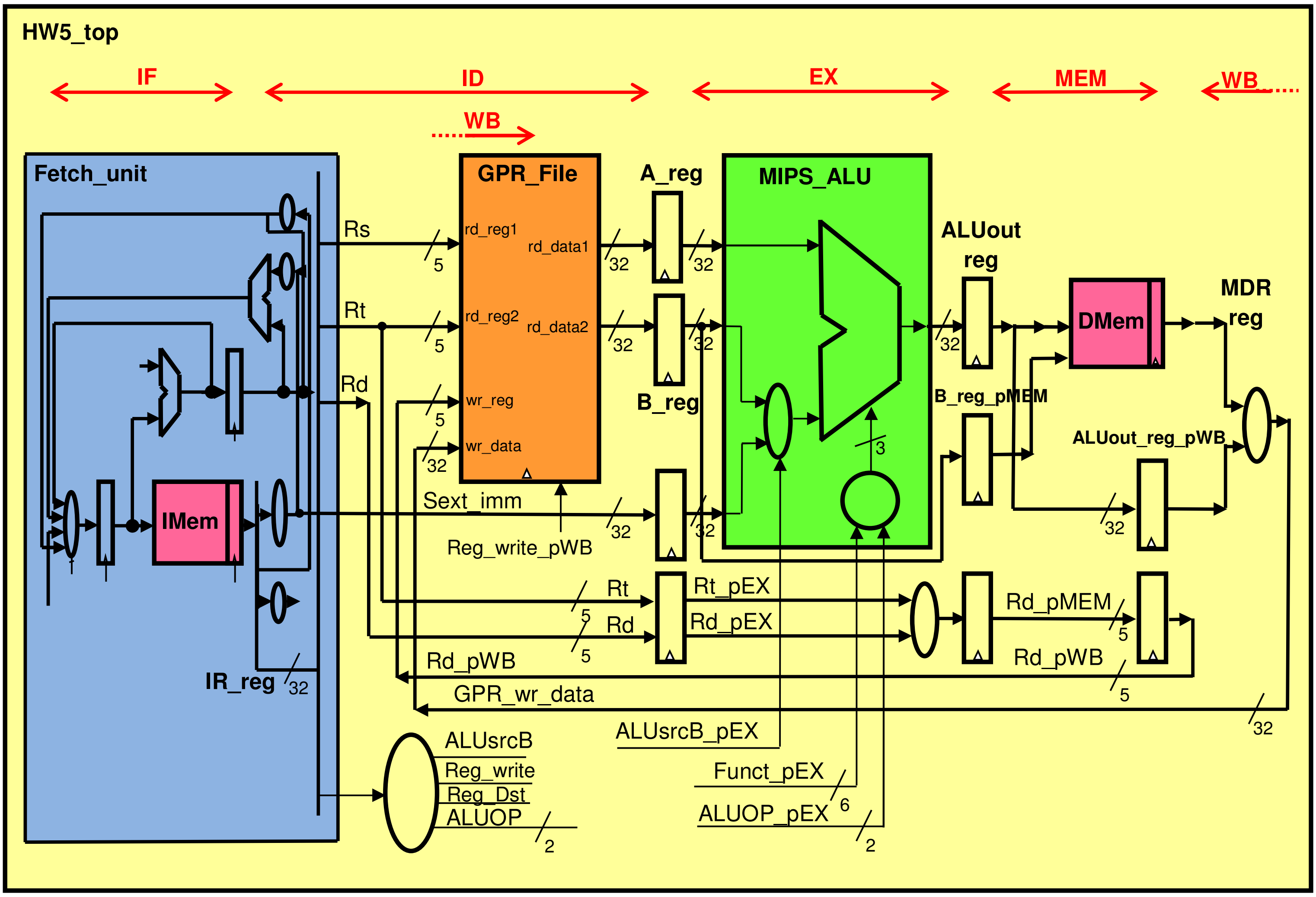


Fig. 1 – The HW5 MIPS CPU

A more accurate drawing includes the BYOC_Host_Intf part – as depicted in Fig.2 below:

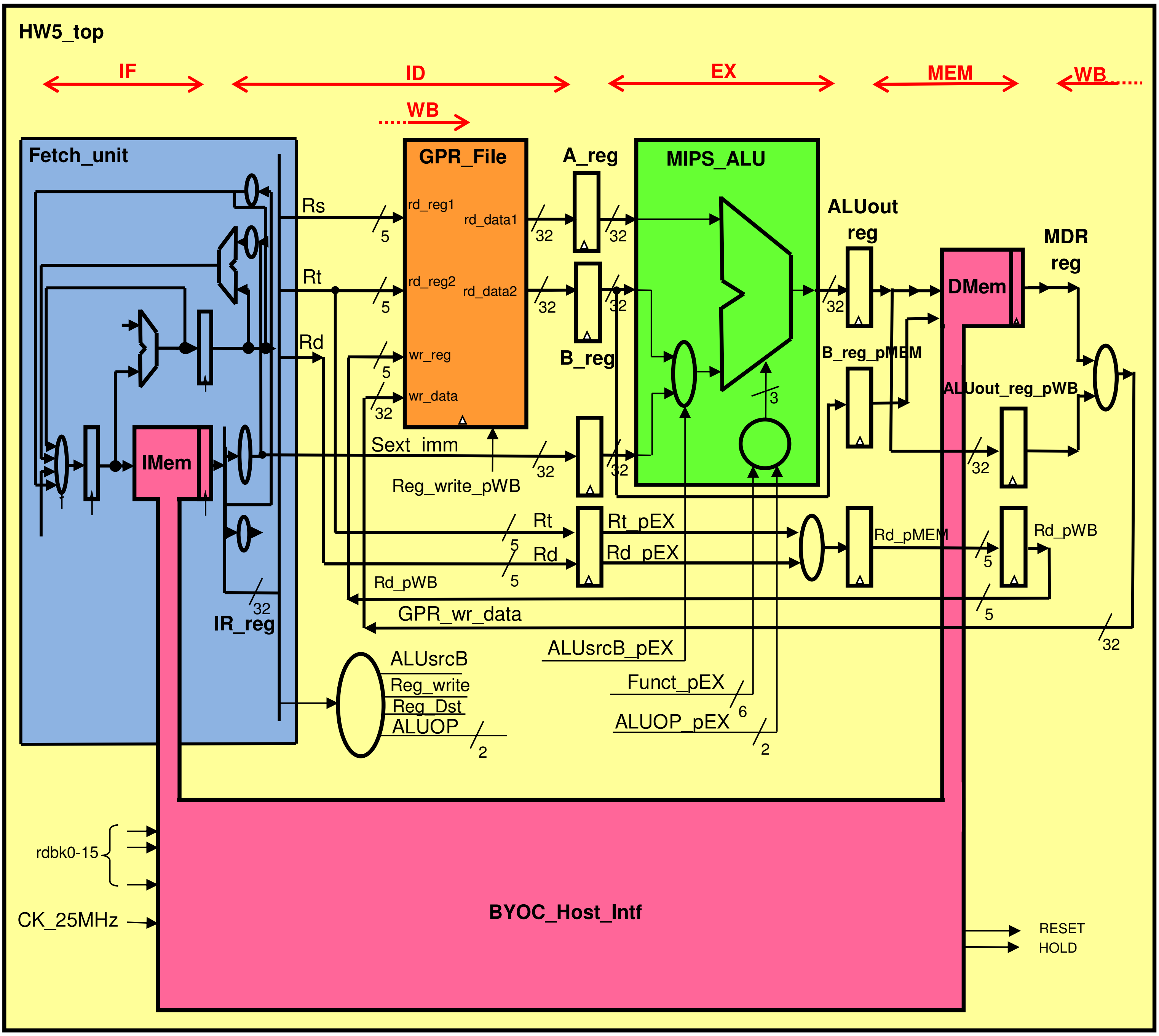


Fig. 2 – The HW5_MIPS CPU with the Host_Intf infrastructure

To your **HW5_top** design, the only required connections to the **BYOC_Host_Intf** are the RESET & HOLD signals, the IMem connections and the DMem connections and the rdbk0-15 coming from the signals we want to check during implementation. These and the rest of the **BYOC_Host_Intf** connections are already given in the **HW5_top_4sim.empty** file.

You should rename that file to **HW5_top_4sim.vhd** and add the necessary equations – all based on HW4 design). Below we describe the actual work required.

a. Connecting the DMem

The DMem signals are already connected in the **HW5_top_4sim.empty** file. There is no need to add any more DMem connections, but you need to understand these connections:

1. MIPS_DMem_adrs – a 32 bit address signal of DMem is connected to ALUout_reg signal.
2. MIPS_DMem_rd_data – the 32 bit data read from the DMem (we read from the address specified by MIPS_DMem_adrs). This is after a register. I.e., it is actually the MDR data. It is directly connected to the HW5_top signal called MDR_reg.
3. MIPS_DMem_wr_data – 32 bit data to be written into the DMem to the address specified by MIPS_DMem_adrs at the rising edge of the CK if MIPS_DMem_we is '1'. It is connected to (i.e., driven by) the B_reg_pMEM signal of HW5_top.
4. MIPS_DMem_we – a '1' means data will be written into the DMem at the rising edge of the CK. This is driven by the MemWrite_pMEM signal of HW5_top.

b. Names & definition of signals inside the HW5_top MIPS CPU

In your design, you should use the exact signal names as were used in the Rtype MIPS CPU of HW4 and **add** the following signals using the exact signal names shown below:

ID additional signals

5. MemWrite – '1' when this is a sw instruction and we write into the DMem, '0' otherwise.
6. MemToReg – '1' when we read from memory, i.e., in lw instruction.

EX phase signals

7. MemWrite_pEX – MemWrite delayed by 1 clock cycle.
8. MemToReg_pEX – MemToReg delayed by 1 clock cycle.

MEM phase signals.

9. B_reg_pMEM – a 32 bit register receiving the B_reg signal (i.e., B_reg delayed by 1 CK cycle). This register has the data to be written into the DMem in sw instruction.
10. Rd_pMEM – the output of RegDest mux selecting to which register the CPU writes in the WB phase.
11. MemWrite_pMEM - MemWrite_pEX delayed by 1 clock cycle.
12. MemToReg_pMEM – MemToReg_pEX delayed by 1 clock cycle.
13. RegWrite_pMEM – RegWrite_pEX delayed by 1 clock cycle.

WB phase signals

14. MDR_reg - a 32 bit register that has the data read from the memory. This is a rename of the DMem_rd_data signal coming out of the **BYOC_Host_Intf_4sim** component.
15. ALUout_reg_pWB - a 32 bit register that has the ALUour_reg data delayed by 1 CK cycle.
16. GPR_wr_data - a 32 bit signal that is the output of the MemToReg mux (selecting between MDR_reg and ALUout_reg_pWB).
17. Rd_pWB – Rd_pMEM delayed by 1 clock cycle.
18. MemToReg_pWB – MemToReg_pMEM delayed by 1 clock cycle
19. RegWrite_pWB – RegWrite_pMEM delayed by 1 clock cycle.

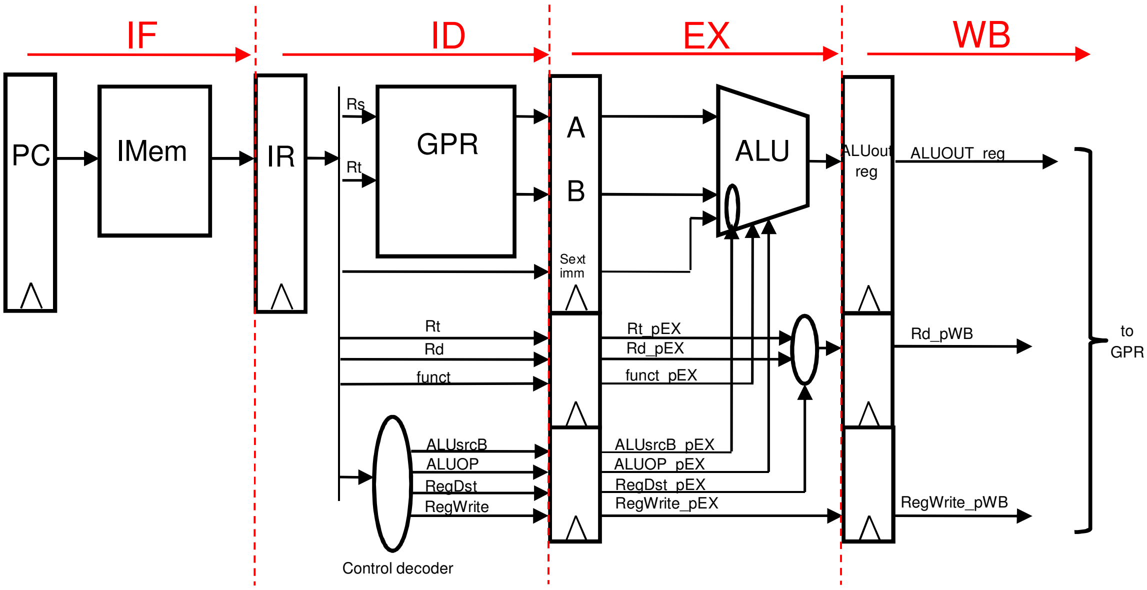


Fig. 1 – HW4 control scheme

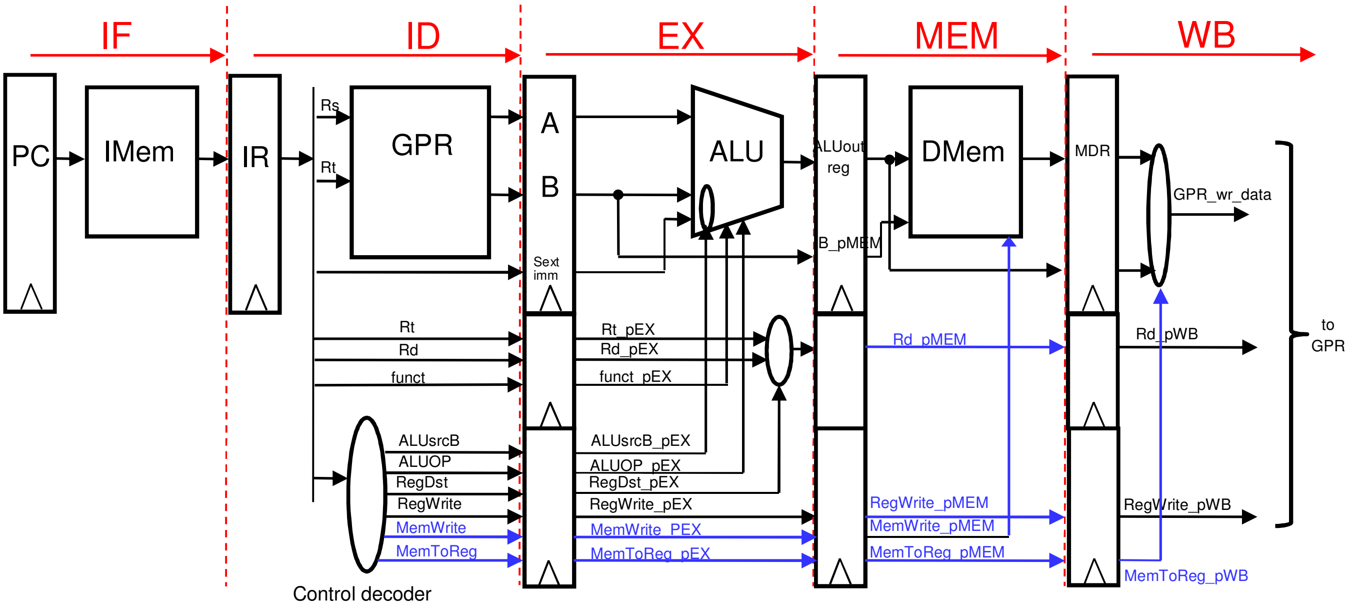


Fig. 2 – HW5 control scheme

(Additions to HW4 control signals - in blue)

1. The HW6 MIPS CPU

In this assignment we add jal, jr, lui, ori instructions to the MIPS CPU we designed in HW5. Thus, we will have a CPU supporting Rtype (add, sub, and, or, xor, slt), addi, lui, ori, beq, bne, lw ,sw, j, jal and jr instructions. Besides adding these instructions we would like to add a forwarding mechanism to enhance the CPU performance.

It is highly recommended to watch the lecture in: http://youtu.be/Yu6FFVhI4D4
and the first 11 minutes of: http://youtu.be/-fylybz8p_M

Below we remind you of the HW5 MIPS CPU we designed in HW5. It is almost the same as the HW6 MIPS of this assignment

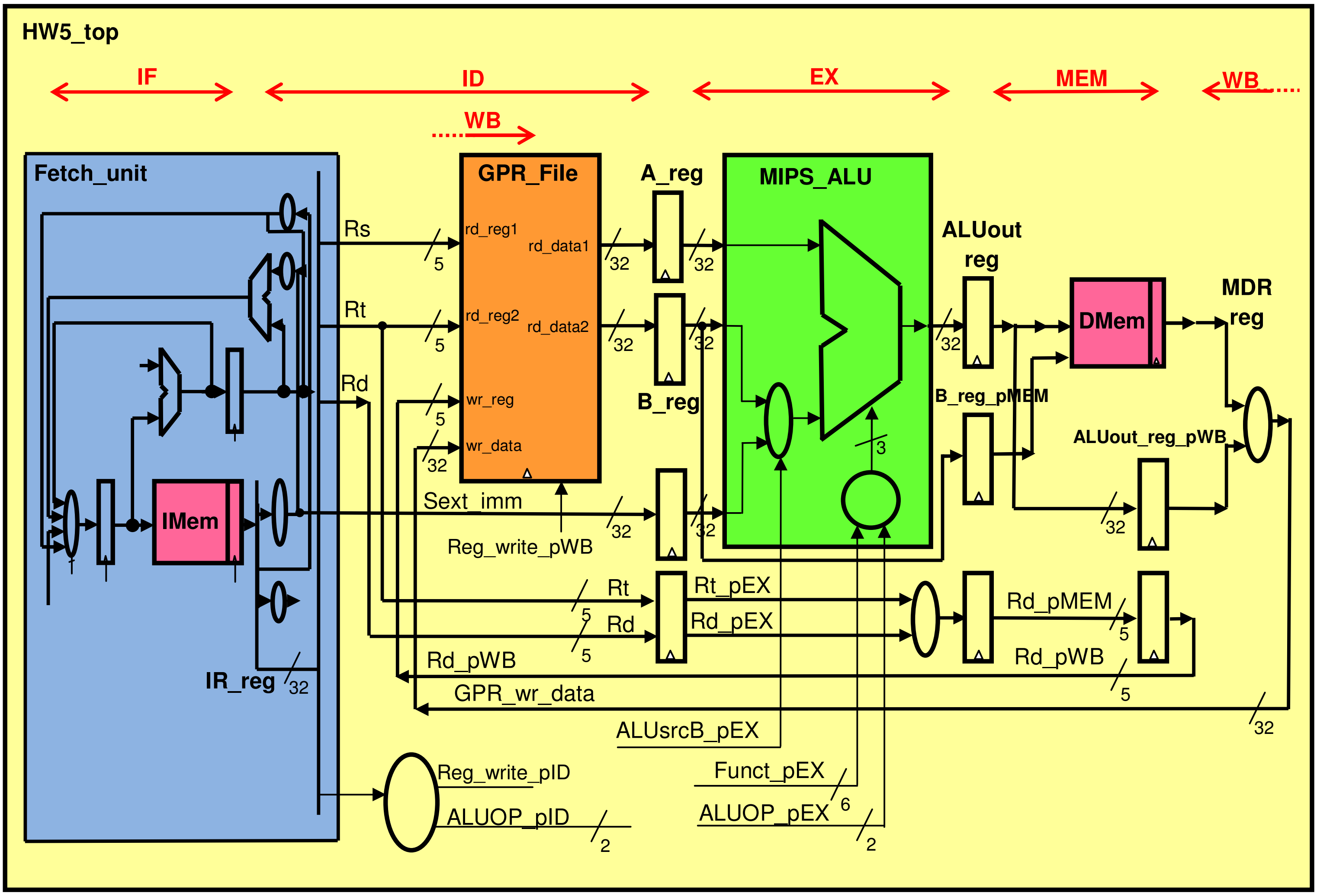


Fig. 1 – The HW5 MIPS CPU

a. HW6 outline

This assignment has 3 parts. The first is to add the new instructions and it is described in section **b** below. It is recommended to fill up the table in **Appendix A** before starting to add the new instructions. We will not implement that design, just run the simulation. After a successful simulation of this part you should add the forwarding mechanism. This is done in two parts, data forwarding (which is described in section **c**) and branch forwarding (described in section **d**). Thus this HW has 3 parts: i) Add the new instructions ii) Add data forwarding iii) Add branch forwarding.

b. PART I - Adding the new instructions

i. LUI – The simplest way to add the lui insruction is to change the sign extension circuit so that when we have a lui instruction, it shifts the imm left for 16 times. We should make sure that the rest of the circuit will behave in a similar manner to addi instruction. For example, the ALU will add it's a input value to the sext_imm_reg value that appears in its B input. Thus, we should make sure that the A_reg value is 0. This can be done in several ways. The simplest way is to make sure that the Assembler always translate lui instruction so that Rs=0. Another way is to force Rs to be 0 (b"00000") when we decode a lui instruction.

ii. ORI – This instruction is almost the same as addi one. There are two differences. The first is that in ori instruction we should prevent sign extension of the imm. This is easily done by an additional change in the sign extension circuit. The other difference is forcing the ALU to perform a OR operation instead of an ADD one. The simplest way to do that is to use the 4th combination of the ALUOP vector signal. While b"00" means ADD, b"01" means SUB and b"10" means use the FUNCTION field to determine the ALU operation, we will add the combination b"11" and will change the MIPS_ALU so that ALUOP="b11" will result with an OR operation.
Thus for supporting ORI, we should fix the sext_imm circuit, force ALUOP control signal to be b"11" and change the MIPS_ALU to support this combination.
[The expected behavior of the ALUOP signal is: "10" in Rtype instructions, "01" in beq & bne, "11" in ori, "00" in all other instructions]

iii. JR – Supporting this instruction is pretty easy. We should direct the Rs content value (GPR_rd_data1) back into the Fetch_Unit so that the jr_adrs signal inside the Fetch Unit will get the GPR_rd_data1 instead of the constant x"00400004" we had so far.
This means we need to add a input signal to the Fetch_Unit entity. This new 32 bit input signal is called jr_adrs_in.

iv. JAL – Supporting this instruction is a little more involved. The jal should behave exactly as the j instruction in the Fetch Unit so that when a j instruction or jal instruction appear in the IR_reg, the PC_source will be "11" and the PC_reg will get the "jump_adrs" signal at its input. This makes sure that we jump properly in both cases. In jal we should also write the PC_plus_4 of the instruction to $ra, i.e., to register $31 in the GPR File. How do we do that? We "propagate" the PC_plus_4 value till the WB phase and there, add it as an additional input to the MemToReg mux. We need to output the PC_plus_4_pID from the Fetch_Unit (this means a change in the i/o pins of the Fetch_Unit). This signal needs to "propagate" till it becomes be PC_plus_4_reg_pWB. We need to make sure we issue RegWrite='1' in jal and we should force "Rd" to be 31. Since the rule for RegDst mux is

that RegDst='1' only in Rtype instructions, it means that in jal instruction it is '0' and the RegDst mux choose Rd_pMEM to be Rt_pEX, it means that in jal instruction we should force Rt to be 31 (b"11111").
To summarize, we need support jal in the Fetch Unit the same as we do for j instruction, we need to output PC_plus_4_pID from the Fetch_Unit and delay it till the WB phase, we need to issue a RegWrite='1', we need to expand the MemToReg mux to write the PC_plus_4 in the WB phase of jal instruction and we need to force Rt to be 31 in jal instruction.

See more in section **e** below.

c. PART II - Data forwarding

In a pipelined implementation of a CPU we encounter an inherent latency problem. The result of an add instruction (we will use add instruction as an example, but the analysis is applicable also for all instructions writing back into the GPR File except lw and jal, i.e., Rtype, addi, lui and ori instructions) is available for a later instruction that uses it only after the WB phase of the add instruction is completed. The instruction using that result "reads" it from the GPR File in its' ID phase. Thus, we need to wait 3 time slots before "using" the add result in a new instruction. This is depicted in Fig. 2 below. The updated value of **$3** is written into the GPR File in the rising edge of the clock ending the WB phase of the "add **$3**,$5,$8" instruction (marked by the red line).
Thus, the ID phase of the "add $y,**$3**,$x" instruction which uses that value, can occur to the right of the red line. We see that the inherent 5 CKs latency of the pipelined implementation results with "wasted" time slots.

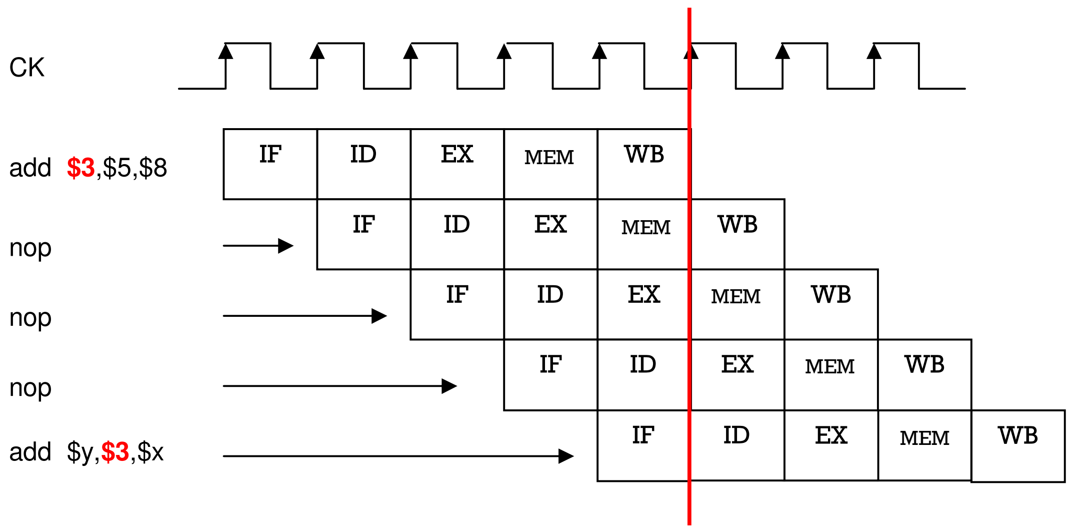


Fig. 2 – The pipelined MIPS latency

We can use these time slots for other instructions that do not write to $3 or $x (those who are used by the "add $y,**$3**,$x" instruction). A smart C compiler can therefore improve the situation. However, it is easy to overcome this problem and improve the situation dramatically by "Data Forwarding".

Data Forwarding means using the updated value to be written into the GPR File even before it is written into the GPR File. This is possible since that data already exists inside the pipeline – in most cases. We read data from the GPR File in the ID phase of an instruction in order to use it in the EX phase of the instruction. This means that the forwarding should occur in the EX phase or before it, in the ID phase of the instruction we want to forward the data to.

We have 3 cases of Data Forwarding.

1. Case I: Forward data from previous instruction in the EX phase of the current instruction if the Rs or Rt of the current instruction is written into by the previous instruction.

   I.e., if RegWrite_pMEM='1' and Rd_pMEM=Rs_pEX, we should use ALUout_reg value instead of A_reg value.
   Similarly, if RegWrite_pMEM='1' and Rd_pMEM=Rt_pEX, we should use ALUout_reg value instead of B_reg value.

   This is described by the arrow from the MEM phase of the 1st instruction (the top one) in Fig. 3, to the EX phase of the 2nd instruction.

2. Case II: Forward data from the instruction that was done 2 clocks ago in the EX phase of the current instruction if the Rs or Rt of the current instruction is written into by the instruction from 2 clocks ago.

   I.e., .if RegWrite_pWB='1' and Rd_pWB=Rs_pEX, we should use MemToReg mux output value instead of A_reg value.
   Similarly, if RegWrite_pWB='1' and Rd_pWB=Rt_pEX, we should use MemToReg mux output value instead of B_reg value.

   This is described by the arrow from the WB phase of the 1st instruction in Fig. 3, to the EX phase of the 3rd instruction.

3. Case III: Forward data from the instruction that was done 3 clocks ago. This is done in the ID phase of the current instruction (through a "transparent GPR") if the Rs or Rt of the current instruction is written into by the instruction from 3 clocks ago.
   This means that inside the GPR, if rd_reg1=wr_reg and Reg_Write='1', then we should bypass the GPR file and output the wr_data instead of the "regular" rd_data1. Similarly to rd_reg2 and rd_data2.

   This is described by the arrow from the MEM phase of the 1st instruction in Fig. 3, to the ID phase of the 4th instruction.

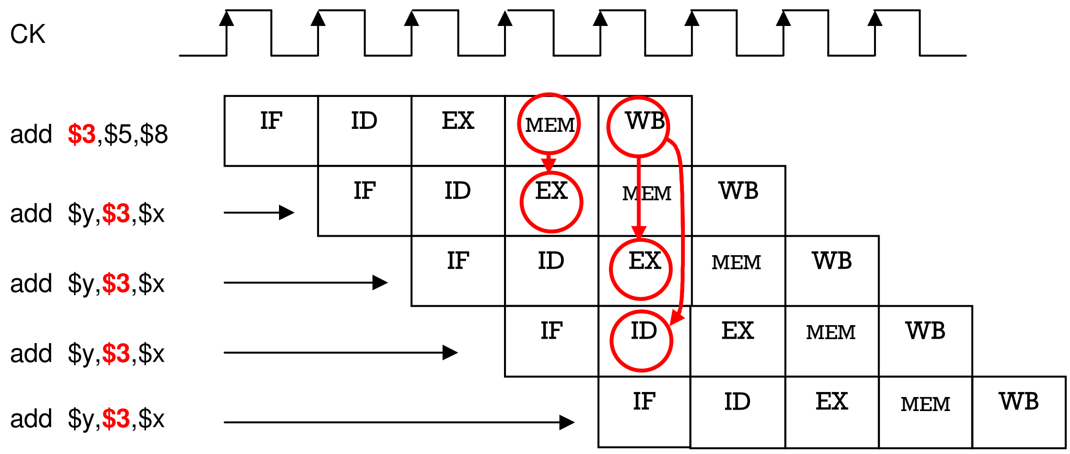


Fig. 3 – Data Forwarding timing diagram
(from the 1st instruction to future instructions)

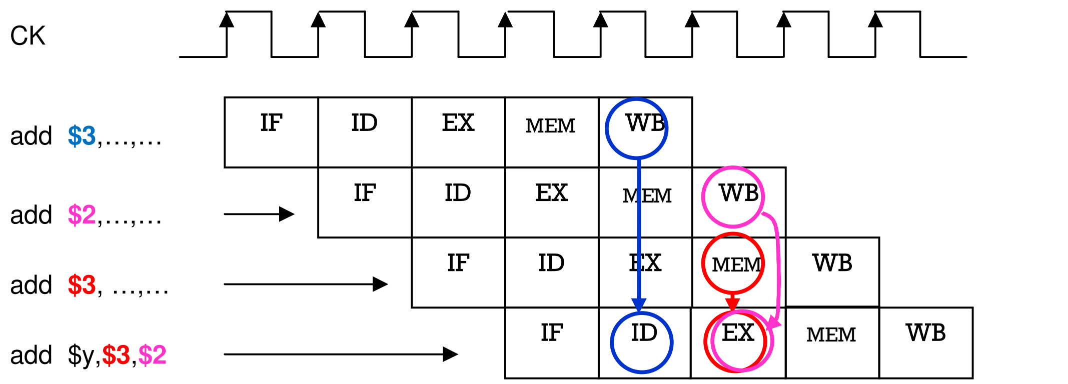


Fig. 3B – The 3 Data Forwarding options to an instruction
(to the 4th instruction from previous instructions)

Fig. 3B shows we see that the 1st instruction writes to register $3, the 2nd instruction writes to register $2 and the 3rd instruction writes to register $3. We see the 3 forwarding mechanisms working to supply updated data to the 4th instruction. In the ID phase of the 4th instruction we read the result of the 1st instruction via the “transparent GPR” mechanism supporting forwarding from 3 instructions ago. In the EX phase of the 4th instructions we see forwarding of Rs from the previous instruction (in red) and from 2 instructions ago (in magenta).

In Fig. 4 and Fig. 5 below we see the MIPS data path without and with Data Forwarding. The changes are drawn in red. The connections shown in the MIPS data path in Fig. 5 support forwarding from previous instruction (case I) and from instruction before the previous one (case II). The forwarding through “transparent” GPR File (case III) is not shown in Fig. 5. It is described in Fig. 6 further below with the changes inside the GPR File also drawn in red.

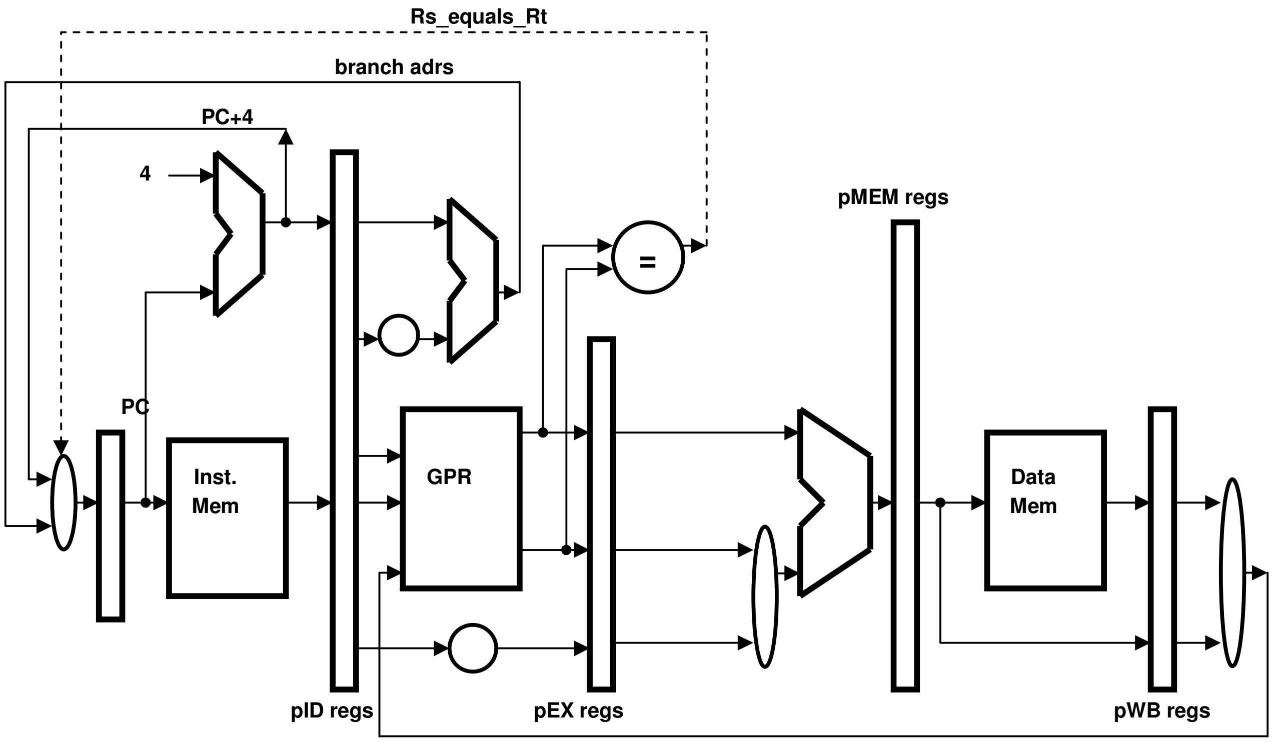


Fig. 4 – MIPS data path (part) with no forwarding

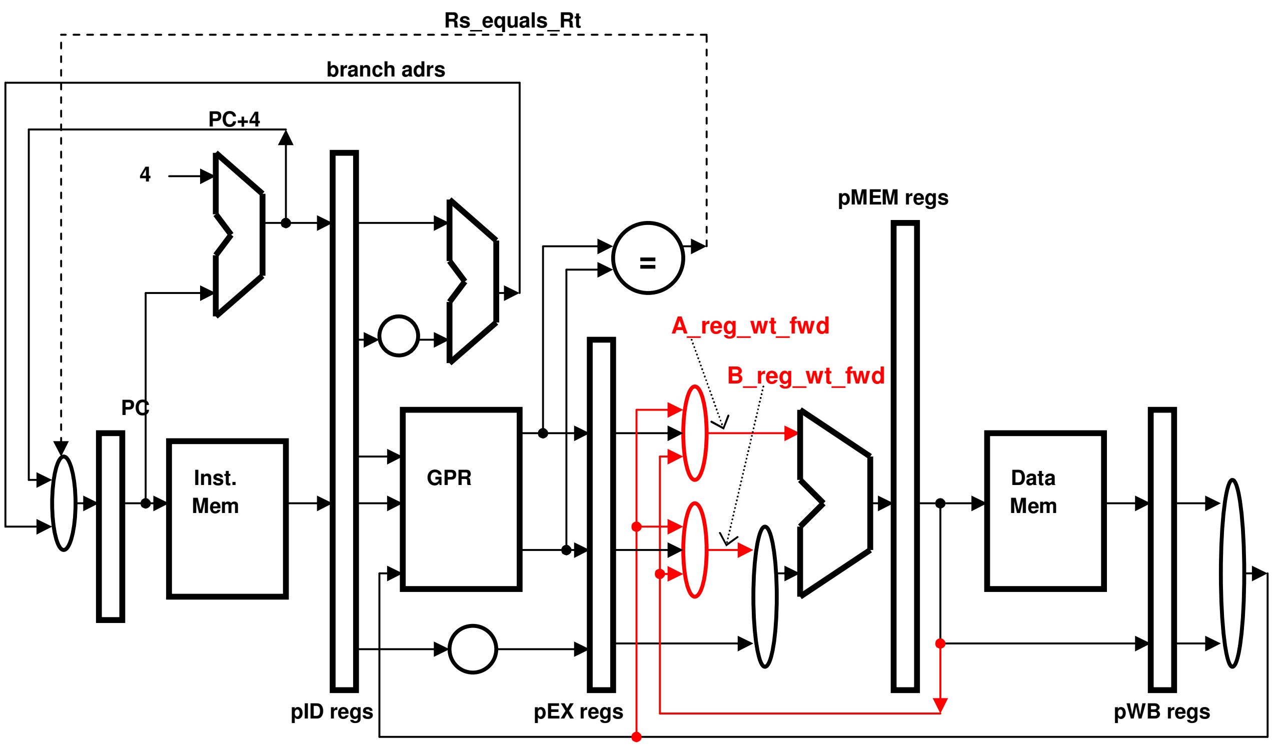


Fig. 5 – MIPS Data Path with Data Forwarding

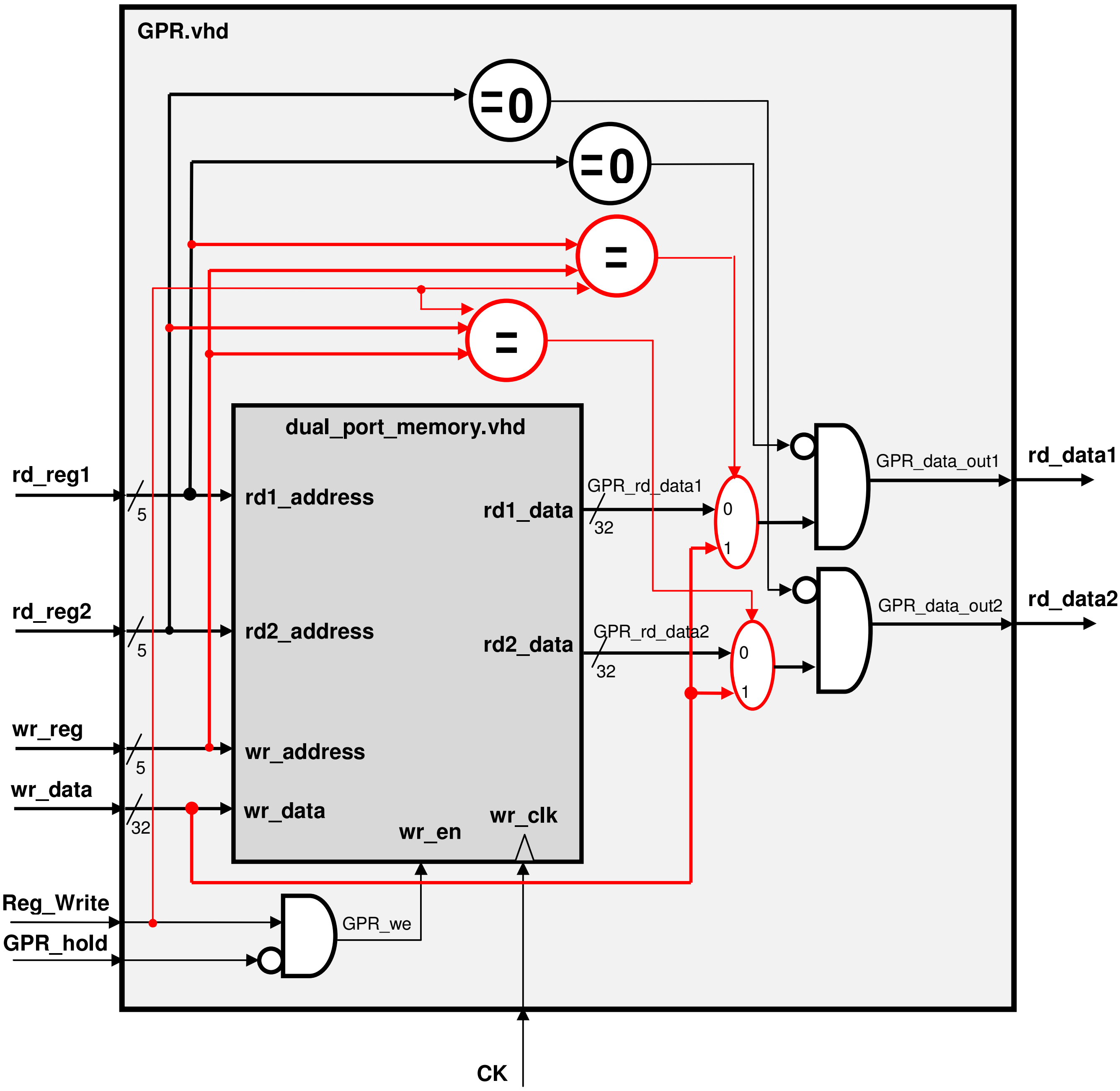


Fig. 6 – MIPS Data Path with Data Forwarding

The only two signals we added to the HW6 MIPS CPU to support Data Forwarding are A_reg_wt_fwd and B_reg_wt_fwd that are the outputs of two new muxes at the A and B inputs of the ALU. Actually we also need to keep the value of Rs till the EX phase so that we can use to check whether forwarding data to the A input of the ALU is required. Thus, we also added the RS_pEX register.

Important notes:
1) No forwarding should be done if we read from register $0 [see how it is handled in Fig. 6].
2) We need to make sure that we handle the situation properly also in cases where we have 2 or 3 previous instructions writing to the same register we are reading from in the current instruction.
3) We need to make sure that we use the correct data also in sw instruction.
4) This forwarding does not apply to lw instruction (if it is the previous instruction) since a lw instruction has valid write data only at the WB phase - after the MEM phase, while other instructions such as Rtype, addi, ori & lui that write to the GPR File have their valid data after the EX phase – from MEM phase and on.
5) Similarly, jal data path is different than the regular instructions and data is available for forwarding only at the WB of the jal instruction.

After adding the data forwarding muxes, we should use A_reg_wt_fwd instead of A_reg wherever the A_reg data was used and similarly, use B_reg_wt_fwd instead of B_reg wherever the B_reg data was used.

See more in section **e** below

d. PART III - Branch forwarding

In a similar manner, the pipeline inherent latency also creates problems when we perform a branch instruction. In order to decide whether to branch or not, we compared the data values read from both outputs of the GPR file. This means we have to wait until the data inside the GPR file is updated before we can branch. As in the data case, we would like to build a forwarding mechanism allowing us to compare the right values as soon they are available.

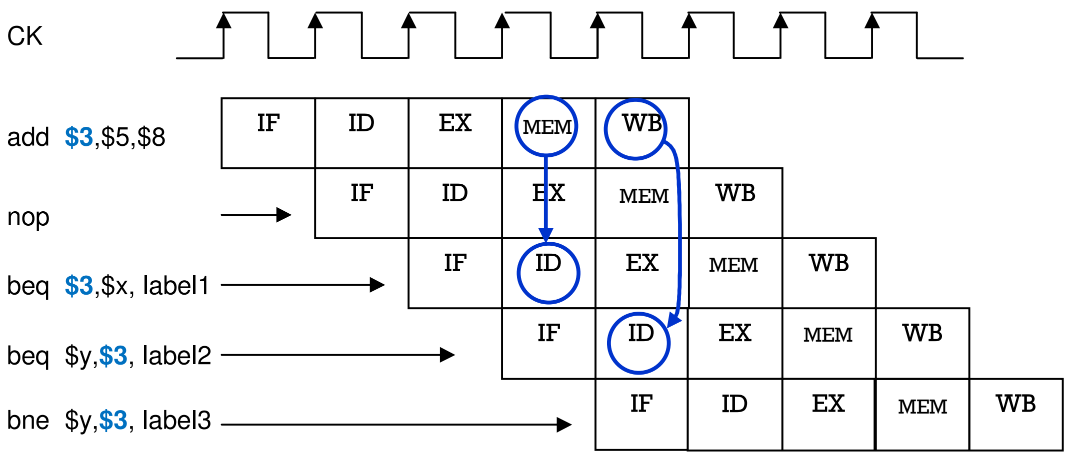


Fig. 7 – Branch Forwarding timing diagram

In Fig. 7 we see that an add instruction writes to register $3. If we want to compare $3 in our branch instruction, we must wait at least 1 time slot. This is so since the result of the add instruction is only available after the EX phase, i.e., from MEM phase and on. Since the branch comparison is done it its' ID phase, the branch instruction' ID phase cannot be performed before the MEM phase of the add instruction. This is not enough. We need a Branch Forwarding mechanism that will bring the updated MEM phase value to the Rs_equals_Rt comparator. Usually this comparator compares GPR_rd_data1 to GPR_rd_data2. Only when we compare a register that was written into (actually, will be written into) by the instruction before the previous (2 instructions ago) we need to forward the MEM phase data (which is the ALUOUT_reg data).
Note that we do not need to handle Branch Forwarding from earlier instructions since from 3 instructions ago, the "transparent GPR" of the data forwarding does that for us, and from 4 instructions ago there is no forwarding problem since the GPR File is updated on time.

You should add this mechanism as depicted in Fig. 8 below.

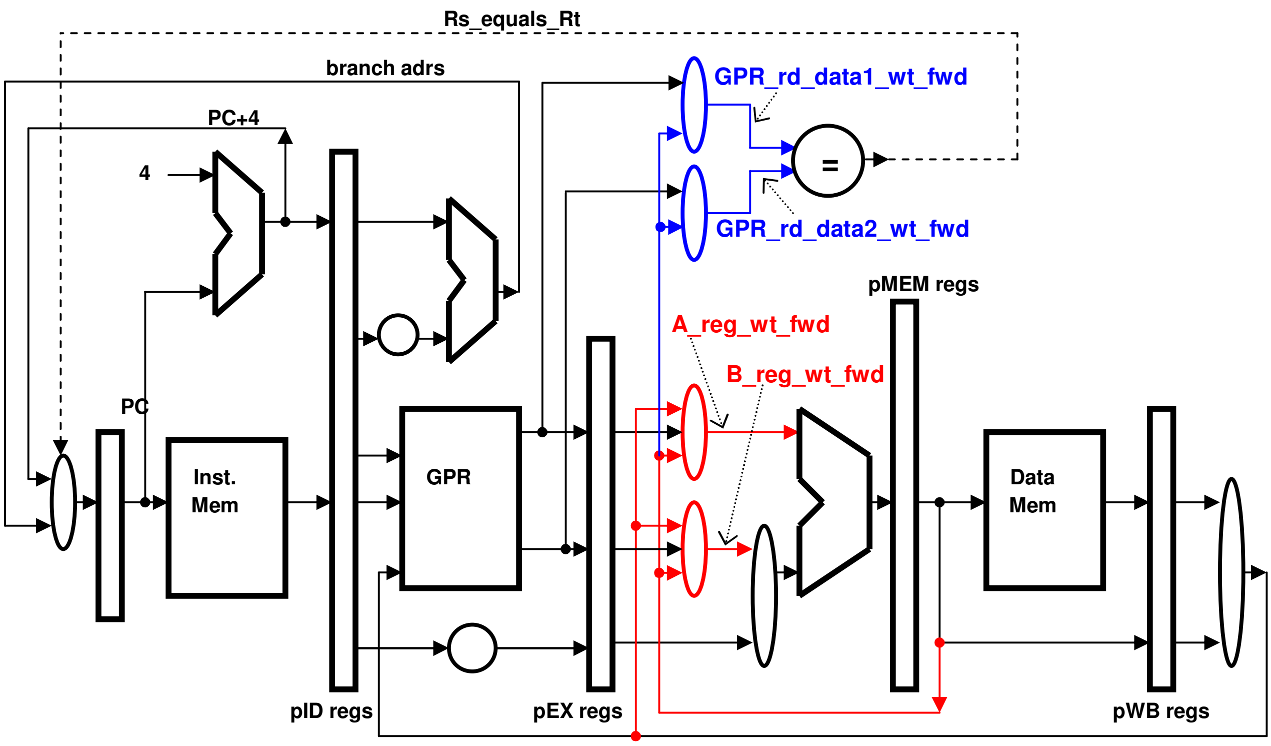


Fig. 8 – MIPS Data Path with Data and Branch Forwarding

The only two signals we added to the HW6 MIPS CPU to support Branch Forwarding are GPR_rd_data1_wt_fwd and GPR_rd_data2_wt_fwd that are the outputs of two new muxes at the inputs of the Rs_equals_Rt comparator.

Note that this mechanism should also be used by the jr instruction. In the jr instruction we have a similar latency issue. Instead of sending back the GPR_rd_data1 to the Fetch Unit, you need now to use the GPR_rd_data1_wt_fwd vector signal.

See more in section **e** below.

Appendix A – IMem program for simulation – 2nd part

Address	label	Inst.	Rd/Rt	Rs	Rt	Imm/label	# remark	Inst. code
4001A8	cont:	addi	$1	$0		1000h	# prep4 sw/lw test	20011000
4001AC		nop						00000000
4001B0		addi	$2	$0		5555h		20025555
4001B4		addi	$3	$0		AAAAh		2003AAAA
4001B8		add	$1	$1	$1		# 1 add once	00210820
4001BC		nop						00000000
4001C0		nop						00000000
4001C4		nop						00000000
4001C8		add	$1	$1	$1		# 2 add for the 2nd time	00210820
4001CC		nop						00000000
4001D0		nop						00000000
4001D4		nop						00000000
4001D8		add	$1	$1	$1		# 3	00210820
4001DC		nop						00000000
4001E0		nop						00000000
4001E4		nop						00000000
4001E8		add	$1	$1	$1		# 4	00210820
4001EC		nop						00000000
4001F0		nop						00000000
4001F4		nop						00000000
4001F8		add	$1	$1	$1		# 5	00210820
4001FC		nop						00000000
400200		nop						00000000
400204		nop						00000000
400208		add	$1	$1	$1		# 6	00210820
40020C		nop						00000000
400210		nop						00000000
400214		nop						00000000
400218		add	$1	$1	$1		# 7	00210820
40021C		nop						00000000
400220		nop						00000000
400224		nop						00000000
400228		add	$1	$1	$1		# 8	00210820
40022C		nop						00000000
400230		nop						00000000
400234		nop						00000000
400238		add	$1	$1	$1		# 9	00210820
40023C		nop						00000000
400240		nop						00000000

400244	nop						00000000
400248	add	$1	$1	$1		# 10	00210820
40024C	nop						00000000
400250	nop						00000000
400254	nop						00000000
400258	add	$1	$1	$1		# 11	00210820
40025C	nop						00000000
400260	nop						00000000
400264	nop						00000000
400268	add	$1	$1	$1		# 12	00210820
40026C	nop						00000000
400270	nop						00000000
400274	nop						00000000
400278	add	$1	$1	$1		# 13	00210820
40027C	nop						00000000
400280	nop						00000000
400284	nop						00000000
400288	add	$1	$1	$1		# 14	00210820
40028C	nop						00000000
400290	nop						00000000
400294	nop						00000000
400298	add	$1	$1	$1		# 15	00210820
40029C	nop						00000000
4002A0	nop						00000000
4002A4	nop						00000000
4002A8	add	$1	$1	$1		# 16 - the 16th addition	00210820
4002AC	nop						00000000
4002B0	nop						00000000
4002B4	nop						00000000
4002B8	sw	$2	$1		0	# now $1=???	AC220000
4002BC	sw	$3	$1		4		AC230004
4002C0	lw	$4	$1		0		8C240000
4002C4	lw	$5	$1		4		8C250004
4002C8	nop						00000000
4002CC	nop						00000000
4002D0	nop						00000000
4002D4	add	$5	$5	$4			00A42820
4002D8	nop						00000000
4002DC	nop						00000000
4002E0	nop						00000000
4002E4	addi	$5	$5		1		20A50001
4002E8	nop						00000000
4002EC	nop						00000000

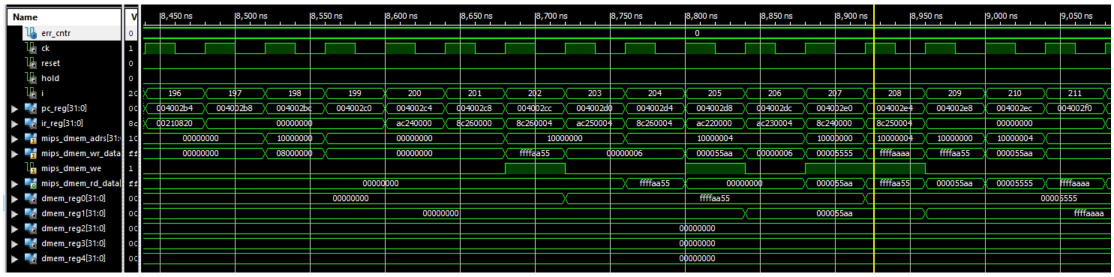
4002F0		nop					00000000
4002F4		bne	$5	$0	errlp		14A00007
4002F8		nop					00000000
4002FC		nop					00000000
400300		nop					00000000
400304	endlp:	j			endlp		081000C1
400308		nop					00000000
40030C		nop					00000000
400310		nop					00000000
400314	errlp:	j			errlp		081000C5
400318		nop					00000000
40031C		nop					00000000
400320		nop				end of errlop	00000000
400324	endlp2:	j			endlp2		081000C9
400328		nop					00000000
40032C		nop					00000000
400330		nop				end of program	00000000

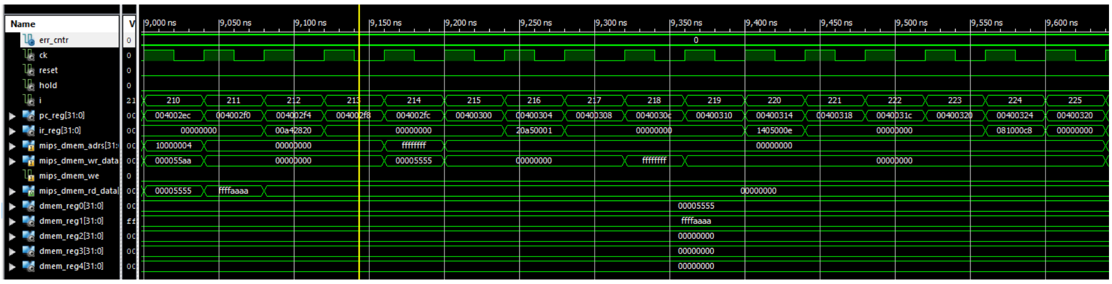
Appendix B – Rect4 - IMem program for implementation

Address in Hex	label	instruction	Rd/ Rt	Rs/ Rt	Rt	Imm/ label	remark	MIPS Hex ode
400000	main	addi	$1	$0		64		20010040
400004		addi	$2	$0		2000h		20022000
400008		addi	$4	$0		16		20040010
40000C		nop						00000000
400010		nop						00000000
400014	shft_lp	add	$2	$2	$2			00421020
400018		addi	$4	$4		-1		2084FFFF
40001C		nop						00000000
400020		nop						00000000
400024		nop						00000000
400028		bne	$4	$0		shft_lp		1480FFFA
40002C		nop						00000000
400030		addi	$2	$2		18h		20420018
400034		addi	$3	$0		-1		2003FFFF
400038		nop						00000000
40003C		nop						00000000
400040		nop						00000000
400044	drawlp	sw	$3	$2		0		AC430000
400048		addi	$1	$1		-1		2021FFFF
40004C		addi	$2	$2		52		20420034
400050		nop						00000000
400054		nop						00000000
400058		bne	$1	$0		drawlp		1420FFFA
40005C		nop						00000000
400060	end	j				end		08100018
400064		nop						00000000

Simulation report

3.1) The listed below signals should be presented in the screen capture you need to attach to your report. Show clock cycles 196-224 (following the end of the reset pulse, find i=196-224) and make the values of all signals readable. For this you will probably need to show clocks 196-210 and 210-224 separately. These are the signals that can help you in "testing" the DMem.





3.2) Explain in detail what happens, i.e., what do we see here. Note that it is essential to the success of your future design that you will verify that the design does what we wanted it to do in these CK cycles.

We store data using SW inside of a loop using branching to fill the memory

3.3) What is the latency of an R-type instruction? That is: How many nop-s should be inserted between two consecutive R-type instructions if the 2nd one uses the result of the 1st one?

There should be 3 nop-s between two consecutive R-type instructions. i.e., the latency of a R-type instruction is 3 CK cycles.

3.4) Explain the limitation of beq that tests a register that is calculated by Rtype instruction. As an example, translate the following C if statement:

for (i=0;i<10;i++) { … }

where i resides in register $3.

It takes 3 clocks to the data in rtype to be written in the memory so we need to make sure there is at least 3 clocks between the rtype instruction and the beq instruction

```
sw      $3 0
sw      $4 10
loop:
addi    $3 $3 1
no op
no op
beq     $3 $4 loop
```

3.5) Are there any other limitations due to the pipeline structure in the instructions we implemented (Rtype, addi, beq, bne, lw, sw, j)? How can we overcome these limitations (e.g., by adding nop-s)? Try to list all of the **SW** & **HW** based solutions you can think of.

We need to make sure that there is enough clock cycles between memory write/read commands so that there will be no conflict while reading/writing to a specific register

Implementation report

1) What is the value of register $2 after 122 cks?

   The address of the first line: 20000018.

2) What happens after 126 CKs?

   The beginning of printing a rectangle on the screen, i.e. the first line is printing on the screen.

3) What happens when you press the RUN button?

   A rectangle has been drawn on the screen which connected to the board.

4) Explain the **HW5_rect4** program (what is the job of every register used. What is done in each loop, etc.)

   The main aim of the program is to draw a rectangle to the screen. It do that by:

   Register $1 saves the number of lines of the rectangle that will be drawn.
   Register $2 represents the address in the screen of the pixel that we want to change.
   Register $3 is the value that allows you to paint specific pixels (each bit marked with 1 turns on the screen).
   Register $4 is the loop counter for register $2.

   label shft_lp: Increasing Register $2 to 20 million.
   label drawlp: write to screen the rectangle.
   label end: end program. An endless loop.

5) How long does it take [in seconds] to draw a 32x32 white square when we use the draw loop of the **HW5_rect4** program?

   It takes 7 clock cycles to draw a single line, so drawing 32 lines will take us 7x31 = 217 clock cycles which are 8,680 nanoseconds.

6) Can you shorten the loop? If you can, write the code and explain.

   The loop can be shortened by canceling the designated counter and using it in comparison to the final value expected from the register $2, i.e. 20003294 (52X 63).

the code:

```
drawlp:     sw $3, 0($2)              #    write to screen
            addi $2,$2,52             #    increment ptr ($2)
            nop
            nop
            bne $2,20003294, drawlp   #    if $2 not end addrs goto drawlp
```

7) Can you think of a faster way to draw the square in the same short loop? If you can, write the code and explain.

A faster way to perform the painting in the draw loop is to define another register that will point to the next row (after the initial value of register $2) and to increase by 104 instead of 52 of the two registers.

The code:

```
drawlp:sw $3, 0($2)     #    write to screen
sw $3, 0($5)
addi $1,$1,-2           #    decrement counter
addi $2,$2,104          #    increment ptr ($2)
addi $5,$5,104          #    increment ptr ($7)
nop
nop
bne $1,$0, drawlp       #    if cntr not 0 goto drawlp
```

A.1) Fill up the following table describing what happens in each CK cycle in all instructions. You should specify the specific operations that are required for the execution of the instruction.

We filled in the Rtype and j instructions – as examples. We also gave the list of required registers & signals to be mentioned in the table, in the ori instruction line.

phase Instruction	IF	ID	EX	MEM	WB
Rtype	IR=IMem[PC] PC= PC+4	A=GPR[Rs] B=GPR[Rt] <u>Active signals:</u> RegDst='1' RegWrite='1' ALUOP="10" MemToReg='0'	ALUOUT = A op B Rd is chosen: Rd_pMEM=Rd_pEX	ALUOUT_pWB= ALUOUT (ALUOUT is delayed 1ck)	GPR[Rd_pWB] = ALUOUT_pWB
addi	IR=IMem[PC] PC= PC+4	A=GPR[Rs] B=imm ALUsrcB <= '1' RegWrite <= '1'	ALUOUT = A + B	ALUOUT_pWB= ALUOUT	GPR[Rd_pWB] = ALUOUT_pWB
ori	IR=IMem[PC] PC= PC+4.	A=GPR[Rs] B=imm ALUsrcB <= '1' ALUOP <= b"11" RegWrite <= '1'	ALUOUT = A or B All regs that are relevant (ALUOUT, B_reg_pMEM, Rd_pMEM, sext_imm)	ALUOUT_pWB= ALUOUT	GPR[Rd_pWB] = ALUOUT_pWB
lui	IR=IMem[PC] PC= PC+4	A=imm B=GPR[Rt] ALUsrcB <= '1' RegWrite <= '1'	ALUOUT = imm<<16	ALUOUT_pWB= ALUOUT	GPR[Rd_pWB] = ALUOUT_pWB
beq	IR=IMem[PC] PC= PC+4	PC= branch_adrs ALUOP <= b"01"	nothing	nothing	nothing
bne	IR=IMem[PC] PC= PC+4	PC= branch_adrs ALUOP <= b"01"	nothing	nothing	nothing

lw	IR=IMem[PC] PC= PC+4	ALUsrcB <= '1' MemToReg <= '1' RegWrite <= '1'	Rt and Rd registes delayed By 1 ck	MDR= DMem[adrs]	GPR[Rd_pWB] = ALUOUT_pWB
sw	IR=IMem[PC] PC= PC+4	ALUsrcB <= '1' MemWrite <= '1'	Rt and Rd registes delayed By 1 ck	DMem[adrs]=B_reg_pMEM	nothing
j	IR=IMem[PC] PC=PC+4	PC= jump adrs	nothing	nothing	nothing
jal	IR=IMem[PC] PC= PC+4	PC= jump adrs RegWrite <= '1' JAL <= '1'	nothing	nothing	nothing
jr	IR=IMem[PC] PC= PC+4	PC= jr_adrs	nothing	nothing	nothing

Answer the following questions.

A.2) Describe the changes done in order to support the ORI instruction.

- ביטול מריחת סימן.

```
process (imm, opcode)
begin
    if opcode = 15 then -- lui
        sext_imm <= imm & x"0000"; --@@@HW6
    elsif opcode /= 13 then -- @@@HW6 not ori
        if imm(15) = '0' then
            sext_imm <= x"0000" & imm;
        elsif imm(15) = '1' then
            sext_imm <= x"FFFF" & imm;
        end if;
    else
        sext_imm <= x"0000" & imm;
    end if;
end process;
```

- regwrite:

```vhdl
if Opcode = 0 or Opcode = 8 or Opcode = 13 or Opcode = 15 or Opcode = 3 or Opcode = 35 then --rtype or addi or ori or lui or jal or lw
    RegWrite <= '1';
else
    RegWrite <= '0';
end if;
```

- aluop ו-aluscrB ובחירת מקור האיבר השני לALU:

```vhdl
process (Opcode, IR_reg)
begin
    if Opcode = 8 or Opcode = 13 or Opcode = 15  or Opcode = 35 or Opcode = 43 then --addi or ori or lui or lw or sw
        ALUsrcB <= '1';
    else
        ALUsrcB <= '0';
    end if;

    if Opcode = 0 then --rtype
        ALUOP <= b"10";
    elsif Opcode = 4 or Opcode = 5 then --beq and bne
        ALUOP <= b"01";
    elsif Opcode = 13 then -- ori
        ALUOP <= b"11";
    else
        ALUOP <= b"00";
    end if;
```

- שינויים בקובץ הalu לתמיכה בפקודה:

```vhdl
-- ALU
process(ALUOP, Funct, ORI)
begin
    if ALUOP = b"00" then
        ALU_cmd <= b"010"; -- ADD
    elsif ALUOP= b"01" then
            ALU_cmd <= b"110";--  SUB
    elsif ALUOP  = b"11" then -- @@@ ORi HW6
        ALU_cmd <= b"001";

    else
        if Funct = b"100000" then
            ALU_cmd <= b"010"; -- FUNCT=ADD
        elsif Funct = b"100010" then
            ALU_cmd <= b"110"; -- FUNCT=SUB
        elsif Funct = b"100100" then
            ALU_cmd <= b"000"; -- FUNCT=AND
        elsif Funct = b"100101" then
            ALU_cmd <= b"001"; -- FUNCT=OR
        elsif Funct = b"100110" then
            ALU_cmd <= b"011"; -- FUNCT=XOR
        elsif Funct = b"101010" then
            ALU_cmd <= b"111"; -- FUNCT=SLT
        else
            ALU_cmd <= b"010"; -- ADD
        end if;
    end if;
end process;
--
```

A.3) Describe the changes done in order to support the LUI instruction.

- שינוי ערך הsext imm

```vhdl
process (imm, opcode)
begin
    if opcode = 15 then -- lui
        sext_imm <= imm & x"0000"; --@@@HW6
```

- איפוס הrs.

```vhdl
if Opcode = 15 then --lui
    Rs <= b"00000";
else
    Rs <= IR_reg(25 downto 21);
end if;
```

- regwrite

```vhdl
if Opcode = 0 or Opcode = 8 or Opcode = 13 or Opcode = 15 or Opcode = 3 or Opcode = 35 then --rtype or addi or ori or lui or jal or lw
    RegWrite <= '1';
else
    RegWrite <= '0';
end if;
```

- alusrcB

```vhdl
process (Opcode, IR_reg)
begin
    if Opcode = 8 or Opcode = 13 or Opcode = 15  or Opcode = 35 or Opcode = 43 then --addi or ori or lui or lw or sw
        ALUsrcB <= '1';
    else
        ALUsrcB <= '0';
    end if;
```

A.4) Describe the changes done in order to support the JR instruction.

- יצרנו משתנה חדש ככניסה לfetch unit בשם jr_adrs_in .

```vhdl
-- JR address      (create the jr_adrs signal)
jr_adrs <= jr_adrs_in; --@@@HW6
```

- בtop משתנה זה מקבל את הערך שלו מgpr_read_data1.

```vhdl
----@@@HW6 add JR support    -- HW6 adding JR forwarding means a change here
jr_address  <= GPR_rd_data1_wt_fwd;
```

A.5) Describe the changes done in order to support the JAL instruction.

- ה pc register מקבל את הכתובת הרלוונטית לקפיצה.

```vhdl
process (RESET, CK, opcode)
begin
    if RESET='1' then
        PC_reg <= x"00400000";
    elsif CK'event and CK='1' and HOLD ='0' then
        if opcode = 2  or opcode = 3 then
            PC_reg <= jump_adrs;  --@@@HW6
        else
            PC_reg <= PC_mux_out;
        end if ;
    end if;
end process;
```

- העברנו החוצה מה fetch unit את pc +4 ו"שרשרנו" אותו ע"ב רגיסטרים לשלב הWB.
- יצירת "דגל" והדלקתו ו "אילוץ rt " לערך31 :

```vhdl
if Opcode = 3 then --jal
    JAL <= '1';
    Rt <= b"11111";
else
    JAL <= '0';
    Rt <= IR_reg(20 downto 16);
end if;
```

- regwrite

```vhdl
if Opcode = 0 or Opcode = 8 or Opcode = 13 or Opcode = 15 or Opcode = 3 or Opcode = 35 then --rtype or addi or ori or lui or jal or lw
    RegWrite <= '1';
else
    RegWrite <= '0';
end if;
```

- שינוי ה mux כך שיקבל את ה- 4 + pc המעודכן:

```vhdl
process (MemToReg_pWB, MDR_reg, ALUout_reg_pWB, PC_plus_4_pWB, JAL_pWB)
begin
    if JAL_pWB = '1' then
        GPR_wr_data <= PC_plus_4_pWB;
    elsif MemToReg_pWB = '1' then
        GPR_wr_data <= MDR_reg;
    else
        GPR_wr_data <= ALUout_reg_pWB;
    end if;
end process;
```

In your answers, besides stating the reasoning in detail, show the relevant VHDL code sections to better explain your answers.

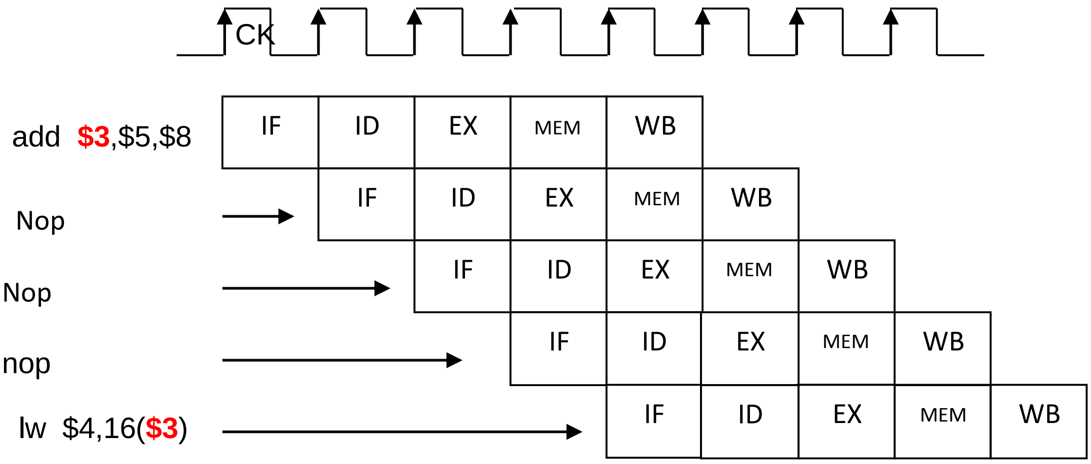
Answer the following questions.

B.1) What are the limitations due to the pipeline latency of the following combinations:

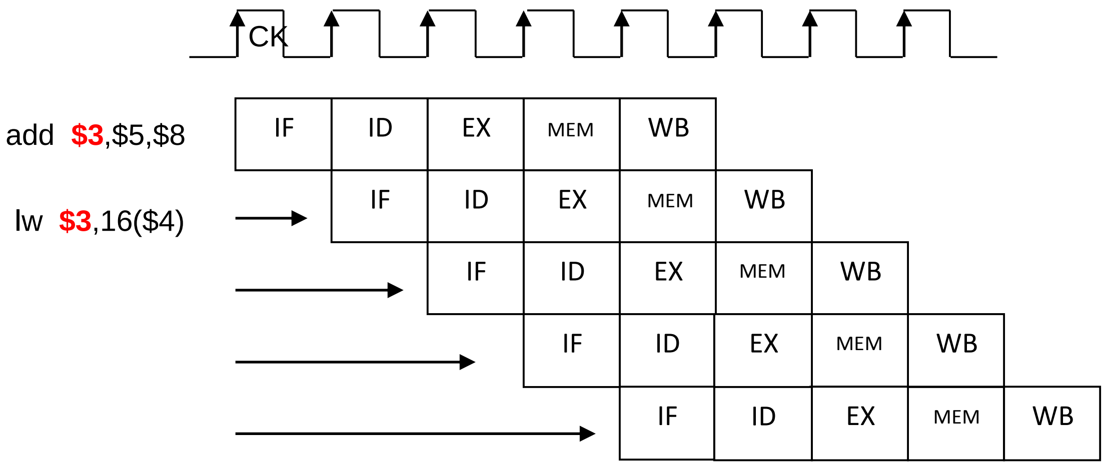
- lw after add where the add Rd is the lw Rs
- lw after add where the add Rd is the lw Rt
- add after lw where the lw Rt is the add Rt
- beq after lw where the lw Rt is the beq Rs

Use a similar figure to Fig.2 and Fig. 3 to demonstrate your answers. <u>Explain your answer</u>!

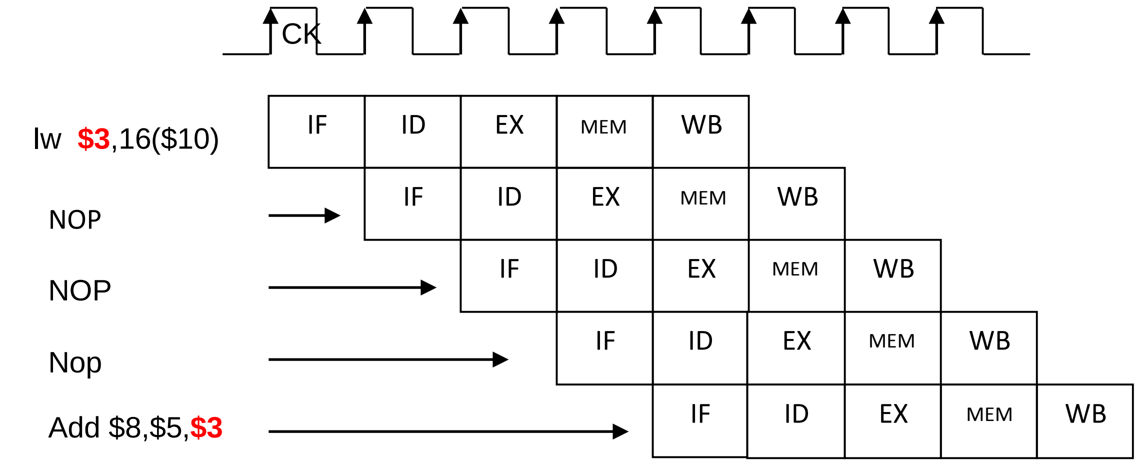
B.1.a - lw after add where the add Rd is the lw Rs



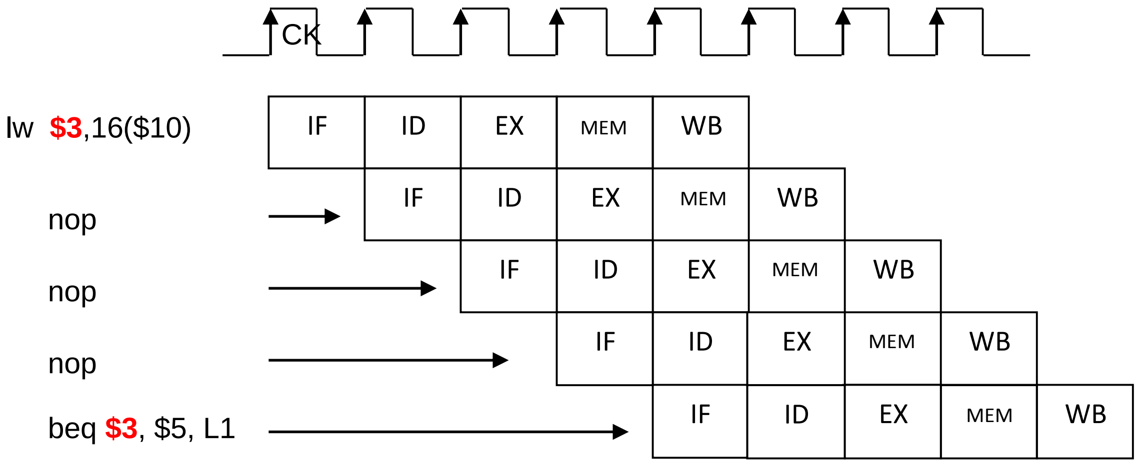
B.1.b - lw after add where the add Rd is the lw Rt



B.1.c - add after lw where the lw Rt is the add Rt

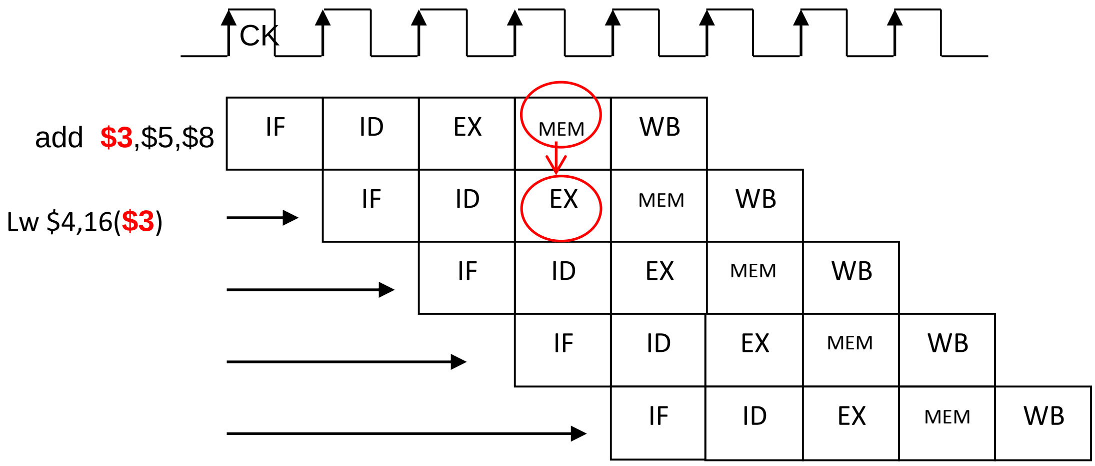


B.1.d - beq after lw where the lw Rt is the beq Rs



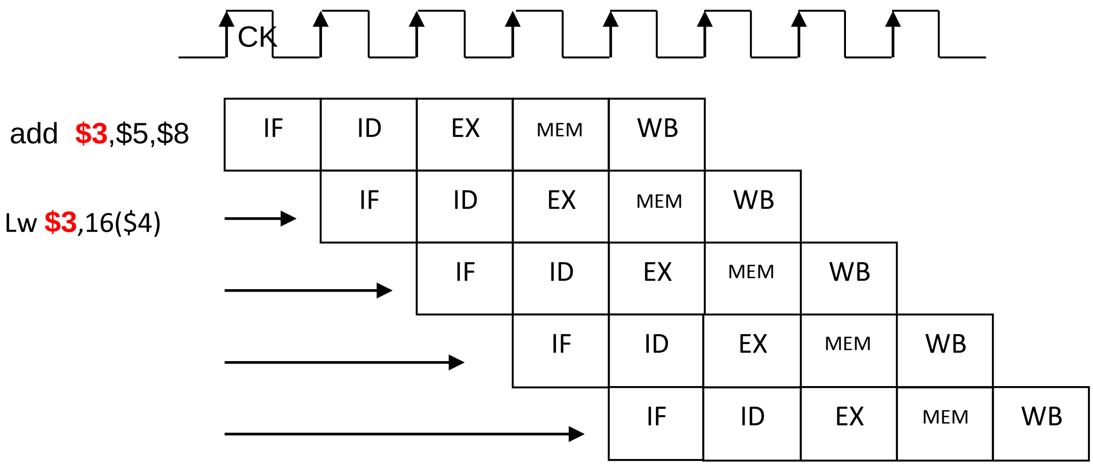
B.2) What are the limitations of all cases of B.1 after you add he Data Forwarding? .
Explain your answer!

B.2.a - lw after add where the add Rd is the lw Rs



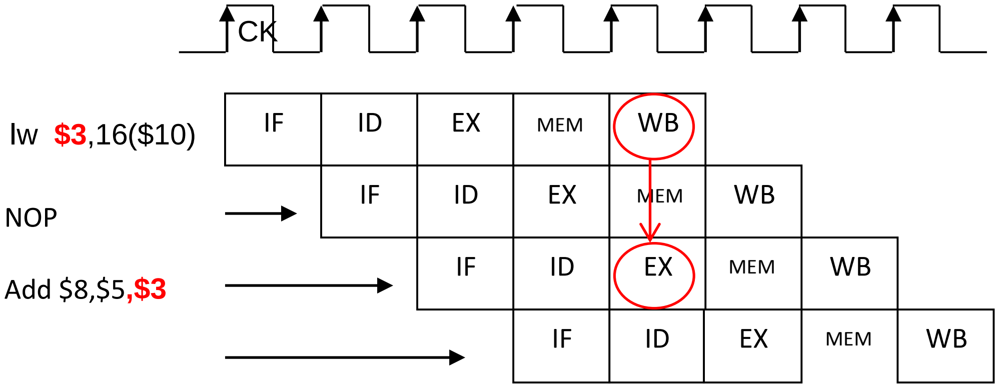
הסבר – add מקבל תוצאה בשלב הex ומבצע forward לשלב ה id של lw .

B.2.b - lw after add where the add Rd is the lw Rt



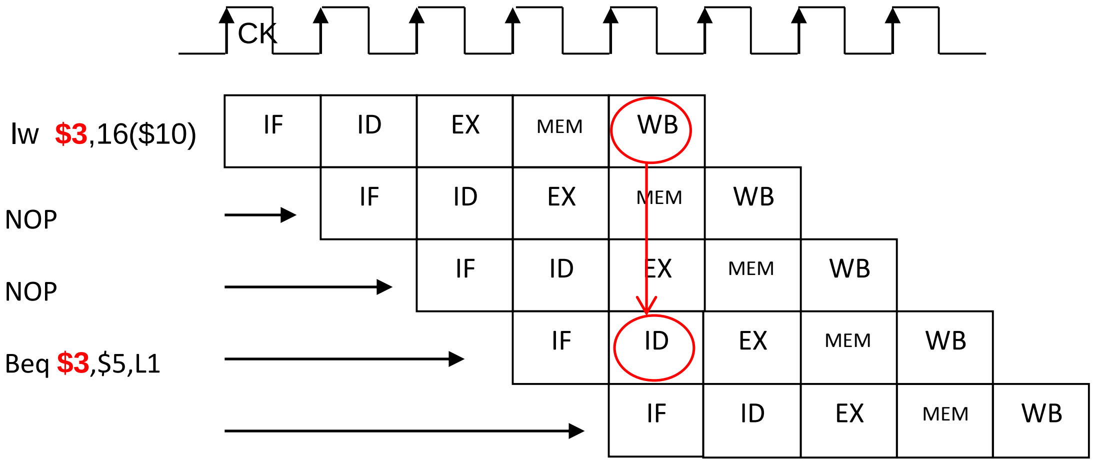
הסבר – לא נדרש שום דבר בדיוק כמו בסעיף B1.b

B.2.c - add after lw where the lw Rt is the add Rt



הסבר: lw מקבל תוצאה בשלב הMEM ויבצע forwarding לשלב ה id של add

B.2.d - beq after lw where the lw Rt is the beq Rs



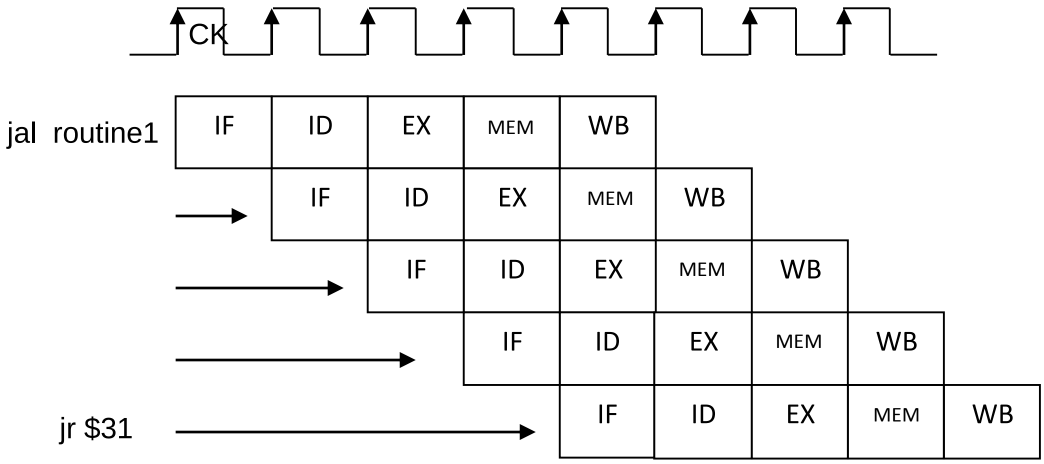
הסבר: lw מקבל תוצאה בשלב הmem וbeq צריך אותה בשלב הID לכן יבוצע forwarding לאחר NOP.

B.3) How many times do we perform the instruction following a jal instruction? Explain in detail. What are the implications? If this is a problem, what do you suggest in order to solve it?

ההוראה לא מתבצעת מכיון שבעת קבלת פקודה ה PCמקודם ב4 אוטומטית ולכן כשנשמר ברגיסטר $RA את כתובת החזרה היא תיהיה PC+8 כלומר הפקודה השניה אחרי JAL, הפתרון הוא קידום הPC+4 משלב ה IFלשלב ה WB

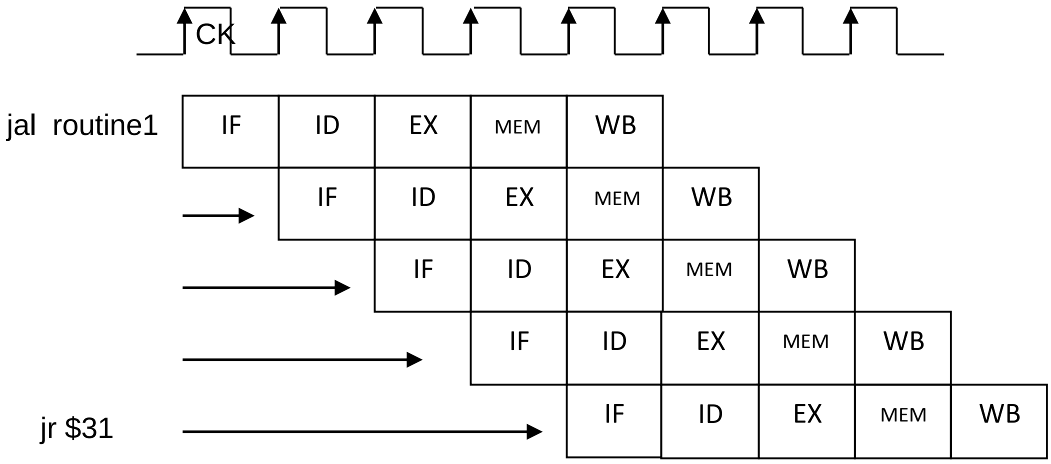
B.4) How soon after jal instruction can we issue a jr $31 instruction in order to return to the right location in the code? Give the answer before data forwarding is added and then after the data forwarding is added. . <u>Explain your answer</u>!

No data forwarding:



נדרש שה PC+4 ישמר ב GPRוזה יכול לקרות רק אחרי של ה WBשל פקודת הJAL

With data forwarding:



מכיוון ש DATA FORWARDING מעביר מידע לשלב הEX לכן אינו מקצר את מספר מחזורי השעון שיש להמתין עד ביצוע jr $31 אשר מתבצע בשלב הID

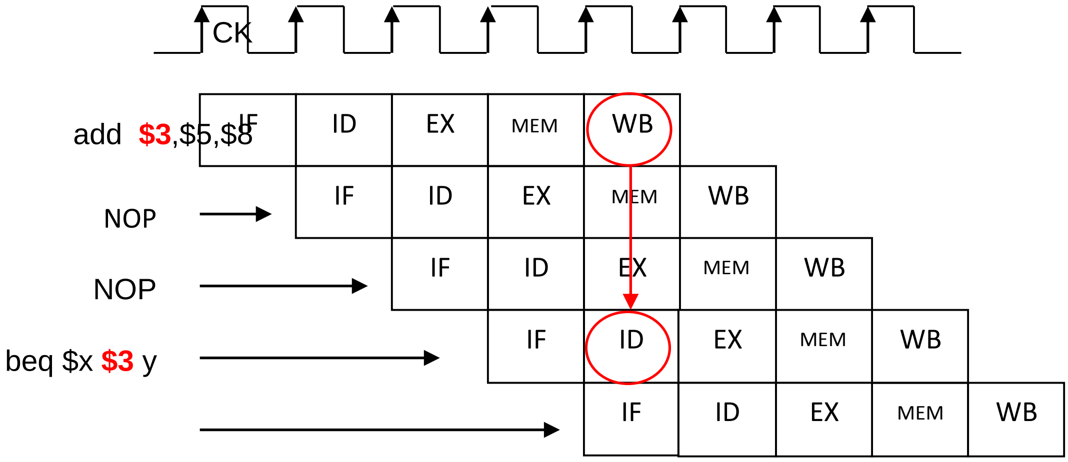
Answer the following questions.

C.1) What are the limitations due to the pipeline latency of the following combinations (assume Data Forwarding already exists):

- beq after add where the add Rd is the beq Rt
- beq after lw where the lw Rt is the beq Rs

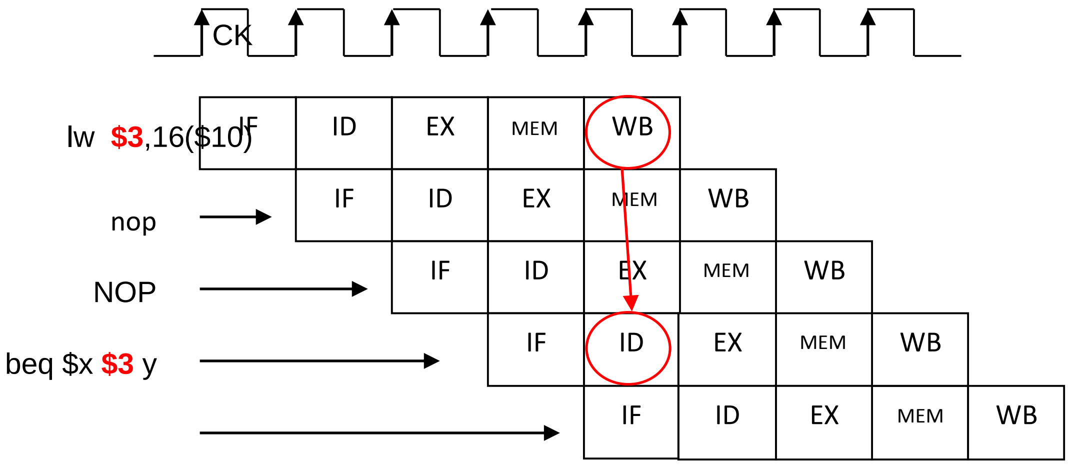
Use a similar figure to Fig.2 and Fig. 3 to demonstrate your answers. <u>Explain your answers</u>!

C.1.a - beq after add where the add Rd is the beq Rt



ללא branch forwarding ניתן לקבל מידע עדכני לשלב הID רק משלושה צעדים אחורה (transparent GPR)

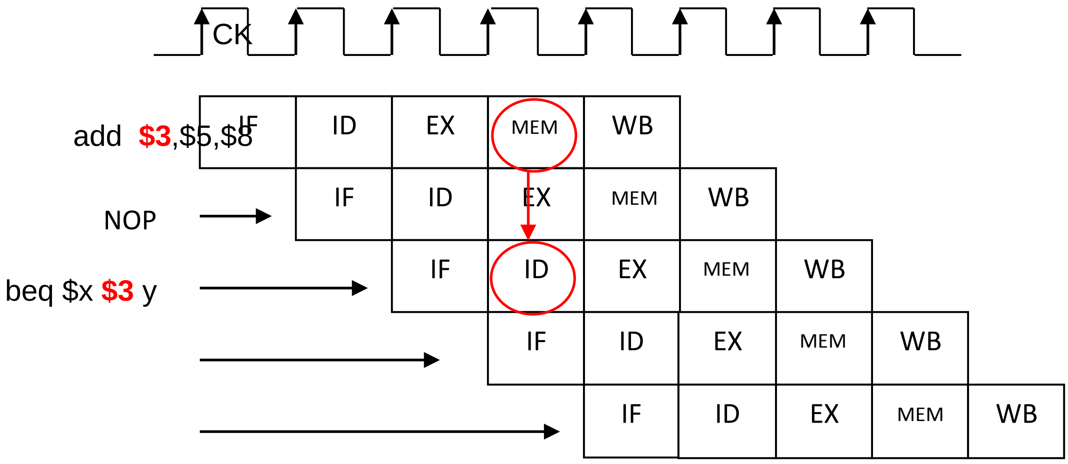
C.1.b - beq after lw where the lw Rt is the beq Rs



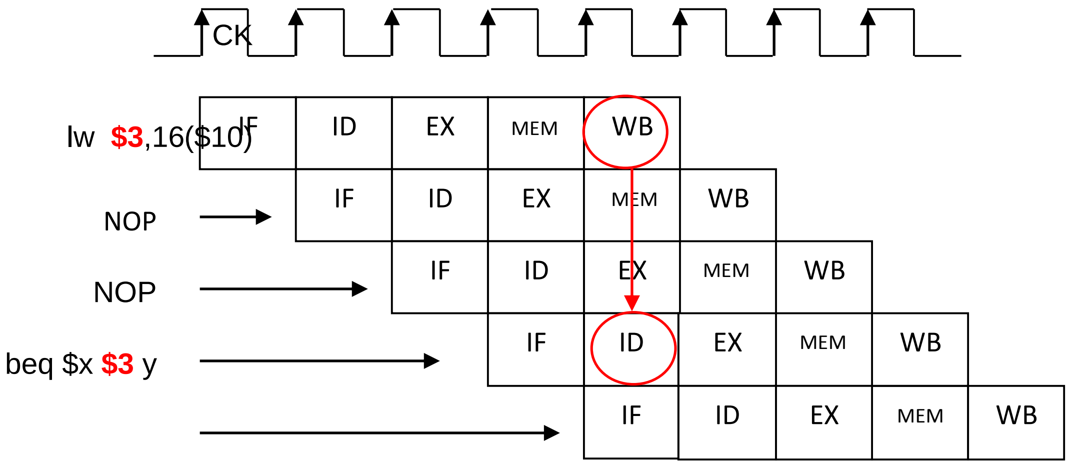
ללא branch forwarding ניתן לקבל מידע עדכני לשלב הID רק משלושה צעדים אחורה (transparent GPR)

C.2) What are the limitations of all cases of C.1 after you add the Branch Forwarding? . Explain your answers!

C.2.a - beq after add where the add Rd is the beq Rt



C.2.b - beq after lw where the lw Rt is the beq Rs



ניתן לקדם מידע ב LWרק לאחר שנשלף מהזכרון

C.3) Why can't we check the result of the previous instruction (time slot n-1) by a beq instruction following it (time slot n)?

מכיוון שבביצוע beq אנחנו מבצעים את ההשוואה בשלב הID ולכן יש לחכות NOP אחד לפחות כדי שהפקודה הקודמת תסיים את של הEX על מנת להשתמש במידע העדכני שלה

C.4) List all of the limitations for Assembly programmer you can think of that still exist after adding the Data & Branch Forwarding circuits. . <u>Explain your answer</u>!

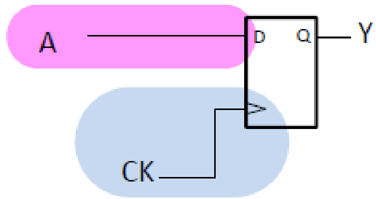
אחרי כל פקודת LW יש לחכות לפחות NOP אחד לפקודת R TYPE
כל BRANCH דורש לפחות NOP אחד לפניו

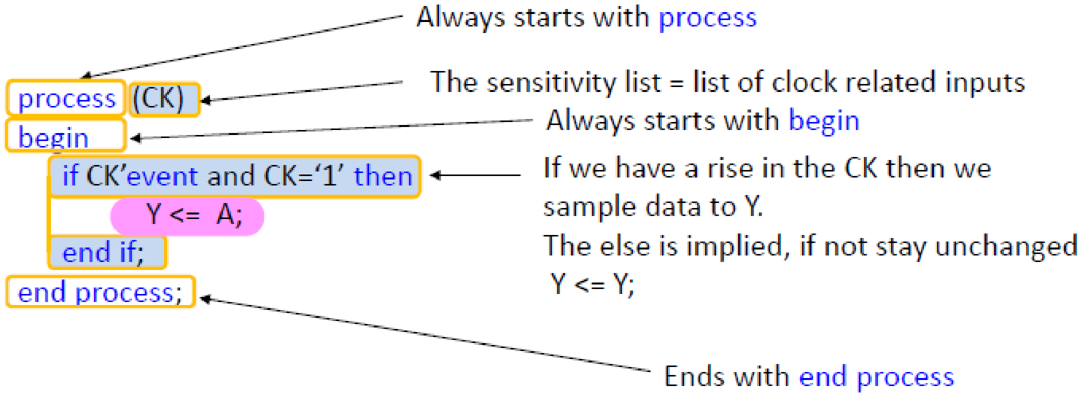
5C) What is the shortest loop code possible (not an infinite loop)? Any limitations? Explain in detail

Lw $8 0h
Lw $9 (loop length)
Nop (if first command in loop is r type
Loop:
Loop operations
Addi $8 $8 4
nop
Bne $8 $9 Loop

VHDL.

```
process (list of clock related inputs)
begin
        all commands are here
        if comes with end if; and elsif if needed
end process;
```





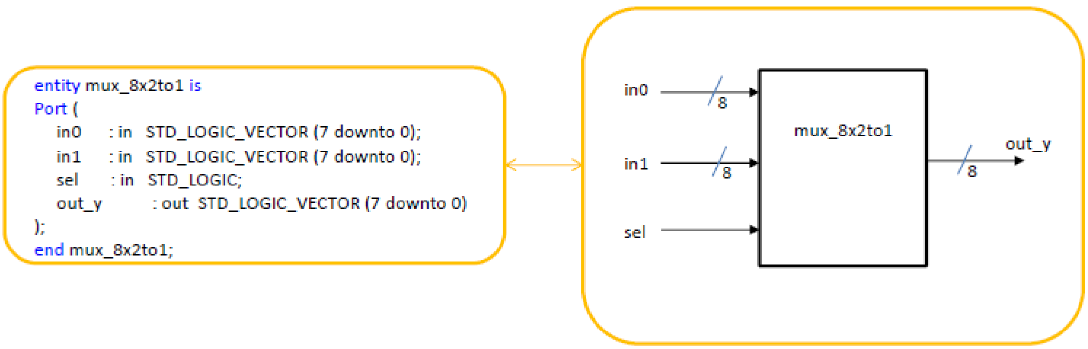
<u>סיגנלים – signals.</u>

- ביט בודד: `signal NAME: std_logic;`.
- מספר ביטים: `signal NAME: std_logic_vector (MSB down to 0)`.

<u>ישות – entity.</u>

ישות מתארת את היציאות והכניסות של מכשיר או רכיב שרוצים להגדיר.

```
entity NAME is
Port (list of ins and outs);
end Name;
```



MIPS' Assembler.

רגיסטרים.

שם	מספר	שימוש
$zero	0	ערך קבוע 0
$v0 - $v1	2-3	ערכים עבור תוצאות והערכת ביטוי
$a0 - $a3	4-7	ארגומנטים
$t3 - $t7	8-15	זמניים
$s0 - $s7	16-23	נשמרים
$t8 - $t9	24-25	עוד זמניים
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

פקודות מסוג R-Type.

מכילה 32 ביטים, כאשר:

31 26	25 21	20 16	15 11	10 6	5 0
Opp Code	R_s	R_t	R_d	00000	Function

פקודות מסוג I-type.

מכילה 32 ביטים, כאשר:

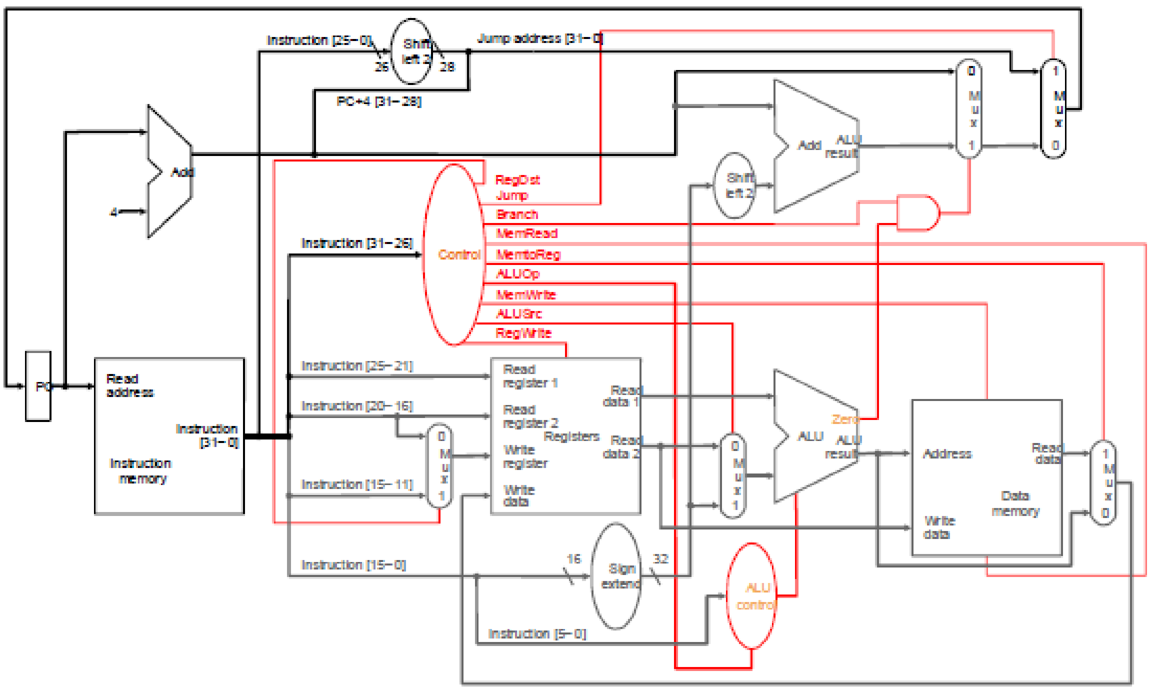
31 26	25 21	20 16	15 0
Opp Code	R_s	R_t	imm

פקודות מסוג J-type.

31 26	25 0
Opp Code	26 bit imm

שלבי ביצוע של המכונה.

1. Fetch – קריאת ההוראות מהזיכרון באמצעות ה-PC כפוינטר.
2. Decode – פיענוח ההוראות (החלטה מה לעשות בצעד הבא) וקריאת הרגיסטרים ההכרחיים (1 או 2).
3. Execute – חישוב התוצאה או כתובת הזיכרון באמצעות ה-ALU.
4. Memory – שימוש בתוצאת ה-ALU כדי לגשת לזיכרון אם צריך (קריאת נתונים, כתיבת נתונים).
5. Write back – אם צריך, כתיבת התוצאה לתוך הרגיסטר המתאים (עדכון רגיסטרים).



Codes of the Opcode fields – IR (31 downto 26):

Command	Binary Opcode	Decimal Value
sw	101011	43
lw	100011	35
lui	001111	15
ori	001101	13
addi	001000	8
beq	000100	4
bne	000101	5
j	000010	2
jal	000011	3
R-type	000000	0

Functions field codes for R-type instructions – IR (5 down 0):

Command	Binary Opcode	Decimal Value
add	100000	32
sub	100010	34
and	100100	36
or	100101	37
xor	100110	38
slt	101010	42
jr	001000	8

תקציר פקודות.

קטגוריה	פקודה	דוגמה	משמעות	הערות
אריתמטיות	add	`add $S1, $S2, $S3`	`$S1 = $S2 + $S3`	המידע ברגיסטרים
	sub	`sub $S1, $S2, $S3`	`$S1 = $S2 - $S3`	המידע ברגיסטרים
	addi	`addi $s1, $s2, 100`	`$S1 = $S2 + 100`	הוספת קבוע
העברת מידע	lw	`lw $s1, 100($s2)`	`$S1 = Memory[$S2 + 100]`	מילה מהזיכרון לרגיסטר
	sw	`sw $s1, 100($s2)`	`Memory[$S2 + 100] = $S1`	מילה מרגיסטר לזיכרון
	lb	`lb $s1, 100($s2)`	`$S1 = Memory[$S2 + 100]`	בייט מהזיכרון לרגיסטר
	sb	`sb $s1, 100($s2)`	`Memory[$S2 + 100] = $S1`	בייט מרגיסטר לזיכרון
	lui	`lui $s1, 100`	$\$S1 = 100 * 2^{16}$	טעינת קבוע לתוך 16 ביטים עליונים
	ori	`ori $12, $0, 0xBEEF`	`$12 = 0x0000BEEF`	טעינת קבוע לתוך 16 ביטים תחתונים
קפיצה מותנית	beq	`beq $s1, $s2, 25`	`if ($S1 == $S2) go to PC +4 + 100`	Branch on equal
	bne	`bne $s1, $s2, 25`	`if ($S1 != $S2) go to PC +4 + 100`	Branch on not equal
	slt	`slt $s1, $s2, $s3`	`if ($S2 < $S3) $S1 = 1;` `else $S1 = 0;`	Set on less than
	slti	`slti $s1, $s2, 100`	`if ($S2 < 100) $S1 = 1;` `else $S1 = 0;`	Set on less than immediate
קפיצה	j	`j 2500`	`go to 10000`	קפיצה לכתובת
	Jr	`jr $ra`	`go to $ra`	קפיצה לרגיסטר
	jal	`jal 2500`	`$ra = PC + 4 ; go to 10000`	קפיצה לכתובת וחזרה לכתובת שאחרי הפקודה המקורית

דגשים חשובים.

- GPR – general-purpose registers: כל 32 הרגיסטרים בעלי 32 ביטים של המעבד.
- בין כל שתי פקודות R-type צריך 2 nop כאשר משנים את אותו רגיסטר. אפשר לפתור את זה באמצעות data forwarding.
- Data forwarding – מעבירים מידע לשלב ה-EX של 1-3 צעדים אחורה. מתי צריך את זה? כאשר פונים לרגיסטר שכתבתנו אליו לפני 1-3 צעדים קודם.
- Branch forwarding – מעבירים מידע לשלב ה-ID.

```vhdl
----------------------------------------------------------------------------------
--
--
-- This module is the HW6_top entity for simulation     see --@@@HW6 for HW6 related changes
--
--
-- It supports Rtype instructions of: add, sub, and, or, xor, slt
-- It also supports addi, beq, bne, lw & sw instructions
-- It also supports lui, ori, jr & jal instructions
--
-- There are 5 phases in HW6 MIPS CPU: IF, ID, EX, MEM, WB
--
--
----------------------------------------------------------------------------------
library  IEEE;
use  IEEE.STD_LOGIC_1164.ALL;
use  IEEE.STD_LOGIC_UNSIGNED.ALL;
use  IEEE.STD_LOGIC_ARITH.ALL;

-- ****************************************************************************************************
-- ****************************************************************************************************

entity HW6_top is
Port  (
--- Infrastructure signals  [To be used by PC via RS232 or from Nexys2 board switches & pushbuttons and VGA signals to the screen]
-- Host intf signals
RS232_Rx        :   in   STD_LOGIC;
RS232_Tx        :   out  STD_LOGIC;
-- VGA signals
VGA_h_sync        :     out      STD_LOGIC;
VGA_v_sync        :     out      STD_LOGIC;
VGA_red0        :     out      STD_LOGIC;
VGA_red1        :     out      STD_LOGIC;
VGA_red2        :     out      STD_LOGIC;
VGA_grn0        :     out      STD_LOGIC;
VGA_grn1        :     out      STD_LOGIC;
VGA_grn2        :     out      STD_LOGIC;
VGA_blu1        :     out      STD_LOGIC;
VGA_blu2        :     out      STD_LOGIC;
--Flash Mem signals
MT_ce_n            :     out      STD_LOGIC; -- '0' when accessing MOBILE SDRAM mem
Flash_adrs         :     out        STD_LOGIC_VECTOR (23 downto 1); -- Flash read/write address
Flash_ce_n        :     out      STD_LOGIC; -- '0' when accessing Flash mem
Flash_we_n        :     out      STD_LOGIC; -- '0' when writing to Flash mem
Flash_oe_n        :     out      STD_LOGIC; -- '0' when reding from Flash mem
Flash_rp_n        :     out      STD_LOGIC; -- '0' when reseting Flash mem
Flash_sts        :     in      STD_LOGIC; -- '1' when Flash mem FSM is done
Flash_data        :    inout    STD_LOGIC_VECTOR (15 downto 0); -- Date read from Imem or Dmem to be written to Flash mem or data read from Flash mem to be written to
--KBD signals
PS2C            :     in      STD_LOGIC; -- PS2 keyboard clock
PS2D            :     in      STD_LOGIC; -- PS2 keyboard data
--general signals
leds_out        :    out        STD_LOGIC_VECTOR (7 downto 0);-- 7=Flash_stts, 6=MIPS_ck, 5-0=Host_intf version
CK_50MHz         :    in        STD_LOGIC;
buttons_in        :    in        STD_LOGIC_vector(3 downto 0) ;--  btn0 is single clock (manual clock), btn3 is manual reset
switches_in     :    in        STD_LOGIC_VECTOR (7 downto 0);-- 4-0 to select which part to be displayed on the 7Segnets LEDs
sevenseg_out    :    out        STD_LOGIC_VECTOR (6 downto 0);-- to the 7 seg LEDs
anodes_out        :    out        STD_LOGIC_VECTOR (3 downto 0)-- to the 7 seg LEDs

        );
end HW6_top;


architecture Behavioral of HW6_top is

-- ****************************************************************************************************
-- ****************************************************************************************************

-- constants
constant MIPS_data_width : INTEGER :=32; -- data width in bits
constant MIPS_adrs_width : INTEGER :=32; -- Full address width of MIPS CPU



--  Put here all the components used:  Clock_Driver, BYOC_Host_intf, your components
-- ================================================================================

-- ****************************************************************************************************
COMPONENT Clock_Driver is
port
 (
   CK_50MHz_IN        : in  std_logic;
   CK_25MHz_OUT        : out std_logic
  );
END COMPONENT;


-- ****************************************************************************************************
-- ****************************************************************************************************
COMPONENT BYOC_Host_intf is
Port    (
--========================The student's part================================
-- MIPS signals    [to be used by students]
MIPS_reset          :     out      STD_LOGIC; -- output to the Student's design
MIPS_hold          :     out      STD_LOGIC; -- output to the Student's design
-- MIPS IMem signals
MIPS_IMem_adrs       :     in      STD_LOGIC_VECTOR (31 downto 0);-- MIPS IMem read address
MIPS_IMem_rd_data :    out        STD_LOGIC_VECTOR (31 downto 0);-- read data (sync read - at the rising edge of MIPS_ck,  all the time)
-- MIPS DMem signals
MIPS_DMem_we      :        in        STD_LOGIC; -- '1' when the CPU writes to MIPD_DMem (MIPS_Dmem_wr_data is written to MIPS_DMem_adrs at the rising edge of MIPS_ck),
MIPS_DMem_adrs       :     in      STD_LOGIC_VECTOR (31 downto 0);-- MIPS DMem read/write address
MIPS_DMem_wr_data :        in        STD_LOGIC_VECTOR (31 downto 0);-- write data  (sync write - at the rising edge of MIPS_ck, if MIPS_DMem_we='1')
MIPS_DMem_rd_data :        out        STD_LOGIC_VECTOR (31 downto 0);-- read data (sync read - at the rising edge of MIPS_ck,  all the time)
--
```

```
--============================Other signals to be directed to i/o pins============================
--Flash Mem signals
Flash_adrs           :     out        STD_LOGIC_VECTOR (23 downto 1);-- Flash read/write address
Flash_ce_n          :     out      STD_LOGIC;-- '1' when accessing Flash mem
Flash_we_n          :     out      STD_LOGIC;-- '1' when writing to Flash mem
Flash_oe_n          :     out      STD_LOGIC;-- '1' when reding from Flash mem
Flash_rp_n          :     out      STD_LOGIC;-- '0' when reseting Flash mem
Flash_sts          :     in      STD_LOGIC;-- '1' when Flash mem FSM is done
Flash_rd_data      :        in        STD_LOGIC_VECTOR (15 downto 0);-- data read from Flash mem to be written to Imem or Dmem
Flash_wr_data      :        out        STD_LOGIC_VECTOR (15 downto 0);-- Date read from Imem or Dmem to be written to Flash
--
-- Infrastructure signals [To be used by PC via RS232 or from Nexys2 board switches & pushbuttons, and VGA signals to the screen],
-- Host intf signals
RS232_Rx        : in STD_LOGIC;
RS232_Tx        : out STD_LOGIC;
-- VGA signals
VGA_h_sync          :     out      STD_LOGIC;
VGA_v_sync          :     out      STD_LOGIC;
VGA_red0          :     out      STD_LOGIC;
VGA_red1          :     out      STD_LOGIC;
VGA_red2          :     out      STD_LOGIC;
VGA_grn0          :     out      STD_LOGIC;
VGA_grn1          :     out      STD_LOGIC;
VGA_grn2          :     out      STD_LOGIC;
VGA_blu1          :     out      STD_LOGIC;
VGA_blu2          :     out      STD_LOGIC;
--PS2 kbd signals
PS2_kbd_ck          :        in      STD_LOGIC;
PS2_kbd_data      :        in      STD_LOGIC;
--
--general signals
CK_25MHz         : in STD_LOGIC; -- main clock input to the Host interface. From this clock we create all other clock signals in the design
buttons_in        : in STD_LOGIC_vector(3 downto 0);--  btn0 is single clock (manual clock), btn3 is manual reset
switches_in     : in STD_LOGIC_VECTOR (7 downto 0);-- 4-0 to select which part to be displayed on the 7Segnets LEDs
sevenseg_out    : out STD_LOGIC_VECTOR (6 downto 0);-- to the 7 seg LEDs
anodes_out         : out STD_LOGIC_VECTOR (3 downto 0);-- to the 7 seg LEDs
leds_out         : out STD_LOGIC_VECTOR (7 downto 0);-- to 8 LEDs (leftmost = Flash status, next = MIPS_ck, 6 right ones = version number)
--
--========================= additional part for the student ================================
-- RDBK signals
rdbk0      :      in    STD_LOGIC_VECTOR (31 downto 0);
rdbk1      :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk2     :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk3      :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk4      :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk5     :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk6     :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk7     :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk8     :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk9     :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk10     :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk11     :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk12     :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk13     :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk14     :     in    STD_LOGIC_VECTOR (31 downto 0);
rdbk15     :     in    STD_LOGIC_VECTOR (31 downto 0)
 ) ;
END  COMPONENT;


-- ***************************************************************************************************
-- put your components declarations here

-- ***************************************************************************************************
COMPONENT Fetch_Unit  is
Port  (
-- general input signals
CK_25MHz         : in  STD_LOGIC;
RESET_in         : in  STD_LOGIC;
HOLD_in         : in  STD_LOGIC;
-- MIPS signals
IR_reg_pID             :    out        STD_LOGIC_VECTOR (31 downto 0);-- The IR_reg (instruction) to be used in ID
sext_imm_pID         :    out        STD_LOGIC_VECTOR (31 downto 0);-- The sext_imm to be used in ID
PC_reg_pIF             :    out        STD_LOGIC_VECTOR (31 downto 0);-- The PC_reg value in IF. To be read by TB in simulation and rdbk in implementation - for verifi
PC_plus_4_pID_out    : out    STD_LOGIC_VECTOR (31 downto 0);-- The PC_plus_4 value in ID  --@@@HW6 JAL support - this is the address to be written to $ra ($31) in the
Rs_equals_Rt_pID     : in      STD_LOGIC; -- '1' if value read from Rs equals the value read from Rt, '0' otherwise. Used in branch instructions.
jr_adrs_in             : in    STD_LOGIC_VECTOR (31 downto 0);-- @@@HW6 JR support -- the  value to be load into the PC in jr instruction   --@@@HW6 add JR support
--- IMem signals
MIPS_IMem_adrs         : out  STD_LOGIC_VECTOR (31 downto 0);
MIPS_IMem_rd_data    : in  STD_LOGIC_VECTOR (31 downto 0)
    );
END COMPONENT;


-- ***************************************************************************************************
COMPONENT  GPR  is
Port (
--  RST        :    in        STD_LOGIC;
CK            :     in      STD_LOGIC;
rd_reg1     :     in      STD_LOGIC_VECTOR (4 downto 0); -- Rs
rd_reg2     :     in      STD_LOGIC_VECTOR (4 downto 0); -- Rt
wr_reg         :     in      STD_LOGIC_VECTOR (4 downto 0); -- Rd (in R-Type instruction, Rt in LW)
rd_data1     :     out     STD_LOGIC_VECTOR (31 downto 0); -- Rs contents
rd_data2     :     out     STD_LOGIC_VECTOR (31 downto 0); -- Rt contents
wr_data     :     in         STD_LOGIC_VECTOR (31 downto 0); -- contents to be written into Rd (or Rt)
Reg_Write     :     in      STD_LOGIC; -- "0" means no register is written into
GPR_hold     :     in      STD_LOGIC  -- "1" means no register is written into
  );
end COMPONENT;


-- ***************************************************************************************************
COMPONENT MIPS_ALU is
Port  (
-- ALU operation control inputs
```

```vhdl
ALUOP        :   in    STD_LOGIC_VECTOR(1 downto 0); -- 00=add, 01=sub, 10=by Function
Funct        :   in    STD_LOGIC_VECTOR(5 downto 0); -- 32=ADD, 34=sub, 36=AND, 37=OR, 38=XOR, 42=SLT
-- data inputs & data control inputs
A_in        :   in    STD_LOGIC_VECTOR(31 downto 0);
B_in        :   in    STD_LOGIC_VECTOR(31 downto 0);
sext_imm    :   in    STD_LOGIC_VECTOR(31 downto 0);
ALUsrcB        :   in    STD_LOGIC;
-- data output
ALU_out        :   out   STD_LOGIC_VECTOR(31 downto 0)
    );
end COMPONENT;

-- ****************************************************************************************************
-- ****************************************************************************************************

-- signals connecting the components, inputs & external logic
-- ==============================================================
-- Reset and CK signals
signal  CK : STD_LOGIC :='0';
signal  RESET : STD_LOGIC :='0'; -- The main RESET signal combined from switches in & MIPS_reset
signal  HOLD : STD_LOGIC :='0'; -- The main RESET signal combined from switches in & MIPS_reset
signal  RESET_from_Host_Intf  :STD_LOGIC; -- is coming from the BYOC_Host_intf

-- Flash data bus signals (used to connect to the Flash_data "inout" pin)
signal  data_from_Flash    : STD_LOGIC_VECTOR (15 downto 0);
signal  data_to_Flash    : STD_LOGIC_VECTOR (15 downto 0);
-- Flasn control signals
signal    Flash_ce_n_line : STD_LOGIC;
signal    Flash_we_n_line : STD_LOGIC;
signal    Flash_oe_n_line : STD_LOGIC;

signal    Flash_rp_n_in_BYOC    :     STD_LOGIC; -- '0' when reseting Flash mem
signal    Flash_sts_in_BYOC    :     STD_LOGIC; -- '1' when Flash mem FSM is done

signal    leds_out_from_host_intf    : STD_LOGIC_VECTOR (7 downto 0); -- 7=Flash_stts, 6=MIPS_ck, 5-0=Host_intf version

--- ===========================================================================
-- Your design signals
--- ===========================================================================

--- ======================== MIPS signals =======================================
--- ============================================================================

-- =========================== IF phase ==============================================
-- ==================================================================================
-- almost all signals are inside the Fetch Unit

-- except IMem signals:
signal    IMem_adrs         : STD_LOGIC_VECTOR (31 downto 0);
signal  IMem_rd_data    : STD_LOGIC_VECTOR (31 downto 0);

-- and we have the PC_reg (PC_reg_pIF) coming out of the Fetch_Unit for rdbk to Host_Intf  & TB
signal  PC_reg    : STD_LOGIC_VECTOR (31 downto 0);

signal  PC_plus_4_pID    : STD_LOGIC_VECTOR (31 downto 0); -- @@@HW6 changes to support JAL instruction

--=========================== ID phase ==============================================
--==================================================================================
-- ID phase  (a register with valid value along the ID phase)
signal  IR_reg    : STD_LOGIC_VECTOR (31 downto 0) ;
-- IR reg signals   (valid in ID phase)
signal  Opcode    : STD_LOGIC_VECTOR (5 downto 0); -- IR[5:0]
signal  Rs : STD_LOGIC_VECTOR (4 downto 0); -- IR[25:21]
signal  Rt : STD_LOGIC_VECTOR (4 downto 0); -- IR[20:16]
signal  Rd : STD_LOGIC_VECTOR (4 downto 0); -- IR[15:11]
signal  Funct    : STD_LOGIC_VECTOR (5 downto 0);-- IR[5:0]

signal rt_tmp : STD_LOGIC_VECTOR(4 downto 0);

-- other signals active in ID phase
signal  sext_imm     : STD_LOGIC_VECTOR (31 downto 0);
signal  GPR_rd_data1 : STD_LOGIC_VECTOR (31 downto 0);
signal  GPR_rd_data2 : STD_LOGIC_VECTOR (31 downto 0);
signal  Rs_equals_Rt : STD_LOGIC; -- '1' if contents of Rs equals the contents of Rt, '0' if not.

--@@@HW6  - add JR support
signal jr_address    : STD_LOGIC_VECTOR (31 downto 0);--the Rs value (usually from GPR_rd_data1) to be loaded into the PC in jr instruction

--@@@HW6  - adding branch forwarding
signal  GPR_rd_data1_wt_fwd    : STD_LOGIC_VECTOR (31 downto 0);--@@@HW6 adding branch forwarding
signal  GPR_rd_data2_wt_fwd    : STD_LOGIC_VECTOR (31 downto 0);--@@@HW6 adding branch forwarding

-- MIPS control signals - created at the ID phase
--------------------------------------------------------------
-- Decoded signals for EX phase
signal  ALUsrcB  : STD_LOGIC;-- '0' selects A_reg, '1' selects sext sext_imm
signal  ALUOP    : STD_LOGIC_VECTOR  (1 downto 0);-- 00=add, 01=sub, 10=by Function   --@@@HW6 11=or to support ORI instruction
signal  RegDst   : STD_LOGIC;--'0' selects Rt, '1' selects Rd
-- Decoded signals for MEM phase
signal  MemWrite : STD_LOGIC;-- '1' for writing to the DMem
-- Decoded signals for WB phase
signal  RegWrite : STD_LOGIC;-- '1' for writing to the GPR file
signal  MemToReg : STD_LOGIC;-- '1' for writing MDR data to the GPR file, '0 for writing ALUout_reg_pWB data to the GPR file
```

```vhdl

signal    JAL         :  STD_LOGIC;-- '1' in JAL instruction -- @@@HW6  - adding JAL instruction



--========================== EX phase ============================================
--===============================================================================
--Registerd valid in EX phase
signal  A_reg            :  STD_LOGIC_VECTOR  (31 downto 0);
signal  B_reg            :  STD_LOGIC_VECTOR  (31 downto 0);
signal  sext_imm_reg     :  STD_LOGIC_VECTOR  (31 downto 0);
signal  Rt_pEX            :  STD_LOGIC_VECTOR  (4 downto 0) ;
signal  Rd_pEX            :  STD_LOGIC_VECTOR  (4 downto 0) ;
signal  ALU_output        :  STD_LOGIC_VECTOR  (31 downto 0);

signal  PC_plus_4_pEX   :  STD_LOGIC_VECTOR  (31 downto 0);     --@@@HW6  - adding JAL instruction

signal  A_reg_wt_fwd    :  STD_LOGIC_VECTOR  (31 downto 0);    --@@@HW6 - adding data forwarding
signal  B_reg_wt_fwd    :  STD_LOGIC_VECTOR  (31 downto 0);    --@@@HW6 - adding data forwarding
signal  Rs_pEX           :  STD_LOGIC_VECTOR  (4 downto 0);            --@@@HW6 - adding data forwarding



-- MIPS control signals - created at the ID phase - delayed to EX phase
-------------------------------------------------------------------------------------------
-- Decoded signals for EX phase
signal  ALUsrcB_pEX     :  STD_LOGIC;
signal  Funct_pEX        :  STD_LOGIC_VECTOR  (5 downto 0);--IR[5:0]
signal  ALUOP_pEX       :  STD_LOGIC_VECTOR  (1 downto 0);
signal  RegDst_pEX      :  STD_LOGIC;
signal  RegWrite_pEX    :  STD_LOGIC;
signal  MemWrite_pEX    :  STD_LOGIC;
signal  MemToReg_pEX    :  STD_LOGIC;

signal  JAL_pEX            :  STD_LOGIC;--@@@HW6 adding JAL instruction



--========================== MEM phase ===========================================
--==============================================================================
--Registerd valid in EX phase
signal  B_reg_pMEM       : STD_LOGIC_VECTOR  (31 downto 0);
signal  Rd_pMEM           : STD_LOGIC_VECTOR  (4 downto 0);
signal  ALUout_reg       : STD_LOGIC_VECTOR  (31 downto 0);

signal  PC_plus_4_pMEM : STD_LOGIC_VECTOR  (31 downto 0); --@@@HW6 - adding JAL instruction


-- MIPS control signals - created at the ID phase - delayed to EX phase
-------------------------------------------------------------------------------------------
-- Decoded signals for MEM phase
signal  RegWrite_pMEM  : STD_LOGIC;
signal  MemWrite_pMEM  : STD_LOGIC;
signal  MemToReg_pMEM  : STD_LOGIC;

signal  JAL_pMEM       : STD_LOGIC;--@@@HW6 adding JAL instruction




--========================== WB phase ===========================================
--===============================================================================
--Registers valid in WB phase
signal  MDR_reg    : STD_LOGIC_VECTOR (31 downto 0); -- renaming of the MIPS_DMem_rd_data signal
signal  ALUout_reg_pWB    : STD_LOGIC_VECTOR (31 downto 0);
signal  GPR_wr_data    : STD_LOGIC_VECTOR (31 downto 0);
signal  Rd_pWB : STD_LOGIC_VECTOR (4 downto 0);

signal  PC_plus_4_pWB : STD_LOGIC_VECTOR (31 downto 0); --@@@HW6 adding JAL instruction


-- signals valid in WB phase
-- MIPS control signals - created at the ID phase - delayed to WB phase
-----------------------------------------------------------------------------------------
-- Decoded signals for WB phase
signal  RegWrite_pWB : STD_LOGIC ;
signal  MemToReg_pWB : STD_LOGIC ;

signal  JAL_pWB         : STD_LOGIC;--@@@HW6 adding JAL instruction



--- ================= End of MIPS signals ======================================
--- ============================================================================



-- *****************************************************************************************
---  Host Intf signals

signal  rdbk3_vec   : STD_LOGIC_VECTOR(31 downto 0);
signal  rdbk4_vec   : STD_LOGIC_VECTOR(31 downto 0);
signal  rdbk5_vec   : STD_LOGIC_VECTOR(31 downto 0);
signal  rdbk12_vec  : STD_LOGIC_VECTOR(31 downto 0);




-- *****************************************************************************************





begin
```

```
-- ***********************************************************************************************
-- Component connections
-- ============================================================================
-- Connect all components used: Clock_Driver, BYOC_Host_intf,  your components ...
-- ============================================================================

-- Connecting the Clock_Driver
-- =======================================
clock_divider : Clock_Driver
port map
 (
   CK_50MHz_IN        =>      CK_50MHz, -- directly form the HW_MIPS i/o pin
   CK_25MHz_OUT        =>      CK        -- the CK signal to the entire HW4_MIPS design
  );

 -- Connecting the HW4_Host_intf
-- ========================================
hostintf :  BYOC_Host_intf
Port Map(
--======================= The student's part ===============================
-- MIPS signals    [to be used by students]
MIPS_reset            =>       RESET_from_host_intf, -- The Host_intf drives the RESET signal
MIPS_hold            =>       HOLD,          -- The Host_intf also drives the HOLD signal
-- MIPS IMem signals
MIPS_IMem_adrs        =>       IMem_adrs,      -- driven by the Fetch_Unit
MIPS_IMem_rd_data    =>       IMem_rd_data,  -- driven by the Host_intf and sent to the Fetch_Unit
-- MIPS DMem signals
MIPS_DMem_we        =>       MemWrite_pMEM, -- '1' if we want to write into DMem at the next rising edge of the MIPS_ck (for sw instruction)
MIPS_DMem_adrs        =>       ALUout_reg,    -- driven by the ALUout_reg = The address to DMem
MIPS_DMem_wr_data    =>       B_reg_pMEM,    -- The data to be written into DMem_adrs in sw instruction
MIPS_DMem_rd_data    =>       MDR_reg,        -- The data read from DMem_adrs in lw instruction. It is registered, i.e.= the MDR data
--
--========================= Other signals to be directed to i/o pins ============================
-- Flash Mem signals
Flash_adrs             =>       Flash_adrs,
Flash_ce_n            =>       Flash_ce_n_line,
Flash_we_n            =>       Flash_we_n_line,
Flash_oe_n            =>       Flash_oe_n_line,
Flash_rp_n            =>       Flash_rp_n_in_BYOC,
Flash_sts            =>       Flash_sts,
Flash_rd_data        =>       data_from_Flash,
Flash_wr_data        =>       data_to_Flash,
--
-- Infrastructure signals  [To be used by PC via RS232 or from Nexys2 board switches & pushbuttons, and VGA signals to the screen],
-- Host intf signals
RS232_Rx            =>       RS232_Rx,
RS232_Tx            =>       RS232_Tx,
-- VGA signals
VGA_h_sync            =>       VGA_h_sync,
VGA_v_sync            =>       VGA_v_sync,
VGA_red0            =>       VGA_red0,
VGA_red1            =>       VGA_red1,
VGA_red2            =>       VGA_red2,
VGA_grn0            =>       VGA_grn0,
VGA_grn1            =>       VGA_grn1,
VGA_grn2            =>       VGA_grn2,
VGA_blu1            =>       VGA_blu1,
VGA_blu2            =>       VGA_blu2,
--PS2 kbd signals
PS2_kbd_ck            =>       PS2C,
PS2_kbd_data        =>       PS2D,
--
--general signals
CK_25MHz            =>       CK, -- CK_25MHz from the Clock_Driver
buttons_in            =>       buttons_in,
switches_in            =>       switches_in,
sevenseg_out        =>       sevenseg_out,
anodes_out            =>       anodes_out,
leds_out            =>       leds_out_from_host_intf,
--
--================== additional part for student ===============================
-- RDBK signals
rdbk0              =>         PC_reg,
rdbk1              =>         IR_reg,
rdbk2              =>         sext_imm,
rdbk3              =>         rdbk3_vec,
rdbk4              =>         rdbk4_vec,
rdbk5             =>         rdbk5_vec,
rdbk6             =>         A_reg,
rdbk7             =>         B_reg,
rdbk8             =>         sext_imm_reg,
rdbk9             =>         ALU_output,
rdbk10             =>         ALUout_reg,
rdbk11             =>         B_reg_pMEM,
rdbk12             =>         rdbk12_vec,
rdbk13             =>         MDR_reg,
rdbk14             =>         ALUout_reg_pWB,
rdbk15             =>         GPR_wr_data
 ) ;


-- ***********************************************************************************************
-- Connecting the Fetch_Unit
-- =======================================
fetch_unit_imp : Fetch_Unit
Port map (
-- general input signals
CK_25MHz        =>       CK,
RESET_in         =>         RESET,
HOLD_in         =>         HOLD,
-- MIPS signals
IR_reg_pID        =>         IR_reg, -- connecting IR_reg_pID to the signal called IR_reg
sext_imm_pID    =>         sext_imm, -- same for the signal called sext_imm
```

```vhdl
PC_reg_pIF        =>         PC_reg,
PC_plus_4_pID_out =>     PC_plus_4_pID, --@@@HW6 for JR support
Rs_equals_Rt_pID  =>     Rs_equals_Rt,
jr_adrs_in        =>         jr_address, --@@@HW6 for JR support
--- IMem signals
MIPS_IMem_adrs      =>     IMem_adrs,
MIPS_IMem_rd_data =>     IMem_rd_data
  );



-- Connecting the GPR file
-- ========================================
GPR_file : GPR
Port map (
--RST        =>     not connected
CK            =>        CK,
rd_reg1     =>        Rs,
rd_reg2     =>        Rt,
wr_reg         =>        Rd_pWB,
rd_data1    =>        GPR_rd_data1,
rd_data2    =>        GPR_rd_data2,
wr_data     =>      GPR_wr_data,
Reg_Write     =>      RegWrite_pWB,
GPR_hold     =>      HOLD -- ,
  );


-- Connecting the MIPS_ALU
-- ========================================
ALU : MIPS_ALU
Port map (
-- ALU operation control inputs
ALUOP        =>        ALUOP_pEX,
Funct        =>        Funct_pEX,
-- data inputs & data control inputs
A_in        =>        A_reg_wt_fwd,   -- @@@HW6 should be A_reg_wt_fwd for adding data forwarding in EX phase
B_in        =>        B_reg_wt_fwd,   -- @@@HW6 should be B_reg_wt_fwd for adding data forwarding in EX phase
sext_imm    =>        sext_imm_reg,
ALUsrcB        =>        ALUsrcB_pEX,
-- data output
ALU_out        =>        ALU_output
  );



-- all signal equations


--  Signals to external components
-- =================================
-- disconnecting the Mobile SRAM
MT_ce_n <= '1' ; -- making sure that the SRAM is not active

-- connecting Flash_data bidir signal
data_from_Flash      <=  Flash_data;
Flash_data   <=  data_to_Flash    when (Flash_oe_n_line ='1' and Flash_ce_n_line='0') else (others => 'Z');

-- connecting other Flash signals
Flash_ce_n    <=  Flash_ce_n_line;
Flash_oe_n    <=  Flash_oe_n_line;
Flash_we_n    <=  Flash_we_n_line;

Flash_rp_n      <=  Flash_rp_n_in_BYOC and ( not switches_in(4) );
Flash_sts_in_BYOC <= Flash_sts;

--leds_out(7) <=  Flash_sts_in_BYOC ;
leds_out      <=  Flash_sts_in_BYOC & leds_out_from_host_intf(6 downto 0); -- 7=Flash_stts, 6=MIPS_ck, 5-0=Host_intf version


-- General signals
-- ========================
RESET  <=  switches_in(6) or RESET_from_Host_Intf;


-- Here is your part, i.e., your equations

-- ============================ IF phase processes =====================================
-- ==================================================================================
-- no such processes. They are all inside the Fetch Unit

-- ============================ ID phase processes =====================================
-- ==================================================================================
-- IR fields signals
Opcode <= IR_reg(31 downto 26);
--Rs     <= IR_reg(25 downto 21);
------------------------------------------------------------------------------------------------------------
--Rt     <= IR_reg(20 downto 16); --@@@HW6 a change is required here to support JAL
------------------------------------------------------------------------------------------------------------
Rd     <= IR_reg(15 downto 11);
Funct  <= IR_reg(5 downto 0);

--beq & bne & jr forwarding                --@@@HW6 adding branch & JR forwarding
--A mux of the Rs_equal_Rt comparator (beq/bne forwarding)
process (RegWrite_pMEM, Rd_pMEM, Rs, GPR_rd_data1, ALUout_reg)
begin
    if RegWrite_pMEM = '1' and Rd_pMEM = Rs and Rs /= b"00000" then
        GPR_rd_data1_wt_fwd <= ALUout_reg;
    else
        GPR_rd_data1_wt_fwd <= GPR_rd_data1;
    end if;
end process;

--B mux of the Rs_equal_Rt comparator (beq/bne forwarding)                --@@@HW6 adding branch & JR forwarding
process (RegWrite_pMEM, Rd_pMEM, Rt, GPR_rd_data2, ALUout_reg)
```

```vhdl
begin
    if RegWrite_pMEM = '1' and Rd_pMEM = Rt and Rt /= b"00000" then
        GPR_rd_data2_wt_fwd <= ALUout_reg;
    else
        GPR_rd_data2_wt_fwd <= GPR_rd_data2;
    end if;
end process;

--beq/bne comparator      --@@@HW6 adding branch forwarding means a change here
process (GPR_rd_data1_wt_fwd, GPR_rd_data2_wt_fwd)
begin
    if GPR_rd_data1_wt_fwd = GPR_rd_data2_wt_fwd then
        Rs_equals_Rt <= '1';
    else
        Rs_equals_Rt <= '0';
    end if;
end process;

----@@@HW6 add JR support    -- HW6 adding JR forwarding means a change here
jr_address  <= GPR_rd_data1_wt_fwd;





-- Control decoder  - calculates the signals in ID phase
-- creates the following signals according to the opcode:
--          ALUsrcB         '0' - selects B_reg, '1' - selects sext_imm_reg
--          ALUOP         b"00" - add, b"01" - sub, b"10" - the Function field determines the ALU operation, b"11" - or --@@@HW6 adding  ORI support
--          RegDst          '1' - "Rd"=Rd (write to Rd - in Rtype inst. only),  '0' - "Rd"=Rt (write to Rt - in all other instructions)
--          MemWrite      '1' - write to DMem
--          MemToReg      '0' - write ALUout_reg data (to "Rd"), '1' - write MDR_reg data (to "Rd")
--          RegWrite      '1' - write to GPR file (to "Rd")
--          JAL          '1' - wrhen we are in jal instruction --@@@HW6 adding  JAL support
process (Opcode, IR_reg)
begin
    if Opcode = 8 or Opcode = 13 or Opcode = 15  or Opcode = 35 or Opcode = 43 then --addi or ori or lui or lw or sw
        ALUsrcB <= '1';
    else
        ALUsrcB <= '0';
    end if;

    if Opcode = 0 then --rtype
        ALUOP <= b"10";
    elsif Opcode = 4 or Opcode = 5 then --beq and bne
        ALUOP <= b"01";
    elsif Opcode = 13 then -- ori
        ALUOP <= b"11";
    else
        ALUOP <= b"00";
    end if;


    if Opcode = 0 then --rtype
        RegDst <= '1';
    else
        RegDst <= '0';
    end if;

    if Opcode = 43 then --sw
        MemWrite <= '1';
    else
        MemWrite <= '0';
    end if;

    if Opcode = 35 then --lw
        MemToReg <= '1';
    else
        MemToReg <= '0';
    end if;

    if Opcode = 0 or Opcode = 8 or Opcode = 13 or Opcode = 15 or Opcode = 3 or Opcode = 35 then --rtype or addi or ori or lui or jal or lw
        RegWrite <= '1';
    else
        RegWrite <= '0';
    end if;

    if Opcode = 3 then --jal
        JAL <= '1';
        Rt <= b"11111";
    else
        JAL <= '0';
        Rt <= IR_reg(20 downto 16);
    end if;

    if Opcode = 15 then --lui
        Rs <= b"00000";
    else
        Rs <= IR_reg(25 downto 21);
    end if;

end process;


-- ============================ EX phase processes ======================================
-- ====================================================================================
-- A & B registers
process (RESET, CK)
begin
    if RESET='1' then
        A_reg <= x"00000000";
    elsif CK'event and CK='1' and HOLD ='0' then
        A_reg <= GPR_rd_data1;
    end if;
end process;
```

```vhdl

process (RESET, CK)
begin
    if RESET='1' then
        B_reg <= x"00000000";
    elsif CK'event and CK='1' and HOLD ='0' then
        B_reg <= GPR_rd_data2;
    end if;
end process;

-- with forwarding                                                      -- @@@HW6 adding data forwarding
-- src_A mux (forwarding)                                              -- @@@HW6 adding data forwarding in EX phase
process (RegWrite_pMEM, Rd_pMEM, Rs_pEX, RegWrite_pWB, Rd_pWB, JAL_pMEM, GPR_wr_data, ALUout_reg, A_reg)
begin
    if RegWrite_pMEM = '1' and Rd_pMEM = Rs_pEX and Rs_pEX /= b"00000" and JAL_pMEM = '0' then
        A_reg_wt_fwd <= ALUout_reg;
    elsif RegWrite_pWB = '1' and Rd_pWB = Rs_pEX and Rs_pEX /= b"00000" then
        A_reg_wt_fwd <= GPR_wr_data;
    else
        A_reg_wt_fwd <= A_reg;
    end if;
end process;

-- src B mux (forwarding part)                                           -- @@@HW6 adding data forwarding in EX phase
process (RegWrite_pMEM, Rd_pMEM, Rt_pEX, RegWrite_pWB, Rd_pWB, JAL_pMEM, GPR_wr_data, ALUout_reg, B_reg)
begin
    if RegWrite_pMEM = '1' and Rd_pMEM = Rt_pEX  and Rt_pEX /= b"00000" and JAL_pMEM = '0' then
        B_reg_wt_fwd <= ALUout_reg;
    elsif RegWrite_pWB = '1' and Rd_pWB = Rt_pEX and Rt_pEX /= b"00000" then
        B_reg_wt_fwd <= GPR_wr_data;
    else
        B_reg_wt_fwd <= B_reg;
    end if;
end process;


-- sext_imm register
process (RESET, CK)
begin
    if RESET='1' then
        sext_imm_reg <= x"00000000";
    elsif CK'event and CK='1' and HOLD ='0' then
        sext_imm_reg <= sext_imm;
    end if;
end process;

-- Rs register    --@@@HW6 added for data forwarding support
process (RESET, CK)
begin
    if RESET='1' then
        Rs_pEX <= b"00000";
    elsif CK'event and CK='1' and HOLD ='0' then
        Rs_pEX <= Rs;
    end if;
end process;

-- Rt register
process (RESET, CK)
begin
    if RESET='1' then
        Rt_pEX <= b"00000";
    elsif CK'event and CK='1' and HOLD ='0' then
        Rt_pEX <= Rt;
    end if;
end process;

-- Rd register
process (RESET, CK)
begin
    if RESET='1' then
        Rd_pEX <= b"00000";
    elsif CK'event and CK='1' and HOLD ='0' then
        Rd_pEX <= Rd;
    end if;
end process;

-- PC_plus_4_pEX  --@@@HW6 added to support JAL instruction
process (RESET, CK)
begin
    if RESET='1' then
        PC_plus_4_pEX <= x"00000000";
    elsif CK'event and CK='1' and HOLD ='0' then
        PC_plus_4_pEX <= PC_plus_4_pID;
    end if;
end process;

-- control signals regs   --@@@HW6  add JAL support here to
process (RESET, CK)
begin
    if RESET='1' then
        ALUsrcB_pEX <= '0';
        Funct_pEX <= b"000000";
        ALUOP_pEX <= b"00";
        RegDst_pEX <= '0';
        RegWrite_pEX <= '0';
        MemWrite_pEX <= '0';
        MemToReg_pEX <= '0';
        JAL_pEX <= '0';
    elsif CK'event and CK='1' and HOLD ='0' then
        ALUsrcB_pEX <= ALUsrcB;
        Funct_pEX <= Funct;
        ALUOP_pEX <= ALUOP;
        RegDst_pEX <= RegDst;
        RegWrite_pEX <= RegWrite;
        MemWrite_pEX <= MemWrite;
```

```vhdl
        MemToReg_pEX <= MemToReg;
        JAL_pEX <= JAL;
    end if;
end process;

-- RegWrite_pEX, MemToReg_pEX, MemWrite_pEX FFs



-- ============================ MEM phase processes =======================================
-- =======================================================================================
--  ALUOUT register
process (RESET, CK)
begin
    if RESET='1' then
        ALUout_reg <= x"00000000";
    elsif CK'event and CK='1' and HOLD ='0' then
        ALUout_reg <= ALU_output;
    end if;
end process;

-- B delayed reg    --@@@HW6 need a change for data forwarding support
process (RESET, CK) ---------------------------------------------------------------------------------------------------------------
begin
    if RESET='1' then
        B_reg_pMEM <= x"00000000";
    elsif CK'event and CK='1' and HOLD ='0' then
        if RegWrite_pMEM = '1' and Rd_pMEM = Rt_pEX  and Rt_pEX /= b"00000" and JAL_pMEM = '0' then
            B_reg_pMEM <= B_reg_wt_fwd;
        elsif RegWrite_pWB = '1' and Rd_pWB = Rt_pEX and Rt_pEX /= b"00000" then
            B_reg_pMEM <= B_reg_wt_fwd;
        else
            B_reg_pMEM <= B_reg;
        end if;
    end if;
end process;

-- RegDst mux and Rd_pMEM register
process (RESET, CK, RegDst_pEX, Rt_pEX, Rd_pEX)
begin
    if RESET='1' then
        Rd_pMEM <= b"00000";
    elsif CK'event and CK='1' and HOLD ='0' then
        if RegDst_pEX = '0' then
            Rd_pMEM <= Rt_pEX;
        else
            Rd_pMEM <= Rd_pEX;
        end if;
    end if;
end process;

-- PC_plus_4_pMEM reg --@@@HW6 added to support JAL instruction
process (RESET, CK)
begin
    if RESET='1' then
        PC_plus_4_pMEM <= x"00000000";
    elsif CK'event and CK='1' and HOLD ='0' then
        PC_plus_4_pMEM <= PC_plus_4_pEX;
    end if;
end process;

-- control signals FFs
--  RegWrite_pMEM, MemToReg_pMEM, MemWrite_pEX FFs  --@@@HW6  add JAL_pMEM to support JAL
process (RESET, CK)
begin
    if RESET='1' then
        RegWrite_pMEM <= '0';
        MemToReg_pMEM <= '0';
        MemWrite_pMEM <= '0';
        JAL_pMEM <= '0';
    elsif CK'event and CK='1' and HOLD ='0' then
        RegWrite_pMEM <= RegWrite_pEX;
        MemToReg_pMEM <= MemToReg_pEX;
        MemWrite_pMEM <= MemWrite_pEX;
        JAL_pMEM <= JAL_pEX;
    end if;
end process;

-- ============================ WB phase processes =======================================
-- =======================================================================================
-- MDR_reg - no need to define -- connected directly from BYOC_Host_intf - resides inside the DMem

--ALUout_pWB register
process (RESET, CK)
begin
    if RESET='1' then
        ALUout_reg_pWB <= x"00000000";
    elsif CK'event and CK='1' and HOLD ='0' then
        ALUout_reg_pWB <= ALUout_reg;
    end if;
end process;

-- MemToReg mux     --@@@HW6 requires changes to support JAL instruction
process (MemToReg_pWB, MDR_reg, ALUout_reg_pWB, PC_plus_4_pWB, JAL_pWB)
begin
    if JAL_pWB = '1' then
        GPR_wr_data <= PC_plus_4_pWB;
    elsif MemToReg_pWB = '1' then
        GPR_wr_data <= MDR_reg;
    else
        GPR_wr_data <= ALUout_reg_pWB;
    end if;
end process;

-- Rd_pWB register
```

```vhdl
process (RESET, CK)
begin
    if RESET='1' then
        Rd_pWB <= b"00000";
    elsif CK'event and CK='1' and HOLD ='0' then
        Rd_pWB <= Rd_pMEM;
    end if;
end process;

-- PC_plus_4_pWB  --@@@HW6 added to support JAL instruction
process (RESET, CK)
begin
    if RESET='1' then
        PC_plus_4_pWB <= x"00000000";
    elsif CK'event and CK='1' and HOLD ='0' then
        PC_plus_4_pWB <= PC_plus_4_pMEM;
    end if;
end process;

-- control signals FFs
--  RegWrite_pWB, MemToReg_pWB FFs   --@@@HW6 added JAL_pWB FF to support JAL instruction
process (RESET, CK)
begin
    if RESET='1' then
        RegWrite_pWB <= '0';
        MemToReg_pWB <= '0';
        JAL_pWB <= '0';
    elsif CK'event and CK='1' and HOLD ='0' then
        RegWrite_pWB <= RegWrite_pMEM;
        MemToReg_pWB <= MemToReg_pMEM;
        JAL_pWB <= JAL_pMEM;
    end if;
end process;

-- ***************************************************************************************************
--build special rdbk signals
rdbk3_vec    <=     b"000" & Rs  &  b"000" & Rt  &  b"000" & Rd  &  b"00" & Funct;
rdbk4_vec    <=     b"000" & RegWrite & b"0000"  &  b"00000000"  &  b"00000000"  &  b"0000" & b"000" & Rs_equals_Rt;
rdbk5_vec    <=  b"000" & ALUsrcB_pEX & b"0000"  & b"00000000" & b"0000"   &  b"00" & ALUOP_pEX & "00" & Funct_pEX;
rdbk12_vec   <=     MemWrite_pMem & b"00" & MemToReg_pMEM & b"000" &  RegWrite_pMEM & b"000" & Rd_pMEM & b"000" & MemToReg_pWB & b"000" & RegWrite_pWB & b"000" & Rd_pWI




-- ************************************************************************************************



end   Behavioral;

-- ***********************************************************************************************
-- ****************************************************************^******************************
```

```
--
--
-- This module is the Fetch Unit
--
--
--
--
--
--------------------------------------------------------------------------------
library  IEEE ;
use  IEEE.STD_LOGIC_1164.ALL;
use  IEEE.STD_LOGIC_ARITH.ALL;
use  IEEE.STD_LOGIC_UNSIGNED.ALL;

-- *********************************************************************************************
-- *********************************************************************************************

entity Fetch_Unit is
Port    (
--
CK_25MHz            : in STD_LOGIC;
RESET_in            : in STD_LOGIC;
HOLD_in            : in STD_LOGIC;
-- IMem signals
MIPS_IMem_adrs      : out STD_LOGIC_VECTOR (31 downto 0);
MIPS_IMem_rd_data : in STD_LOGIC_VECTOR (31 downto 0);
IR_reg_pID : out STD_LOGIC_VECTOR (31 downto 0);
sext_imm_pID : out STD_LOGIC_VECTOR (31 downto 0);
PC_reg_pIF : out STD_LOGIC_VECTOR (31 downto 0);
Rs_equals_Rt_pID : in STD_LOGIC;
jr_adrs_in            : in STD_LOGIC_VECTOR  (31 downto 0); --@@@HW6
PC_plus_4_pID_out    : out STD_LOGIC_VECTOR  (31 downto 0)--@@@HW6
        );
end Fetch_Unit;


architecture Behavioral of Fetch_Unit is

-- *********************************************************************************************
-- *********************************************************************************************


--- ========================  Host intf signals  ===================================
--==================================================================================
signal  RESET             : STD_LOGIC; -- is coming directly from the Fetch_Unit_Host_intf
signal  CK                 : STD_LOGIC; -- is coming directly from the Fetch_Unit_Host_intf
signal  HOLD             : STD_LOGIC; -- is coming directly from the Fetch_Unit_Host_intf
signal    IMem_adrs         : STD_LOGIC_VECTOR(31 downto 0);
signal  IMem_rd_data    : STD_LOGIC_VECTOR(31 downto 0);


-- ========================  MIPS signals  ========================================
-- ================================================================================

-- =========================== IF phase ==============================================
-- ==================================================================================
--- IR & related signals
signal  IR_reg            : STD_LOGIC_VECTOR  (31 downto 0) := x"00000000";
signal  imm             : STD_LOGIC_VECTOR  (15 downto 0);
signal  sext_imm         : STD_LOGIC_VECTOR  (31 downto 0);
signal  opcode             : STD_LOGIC_VECTOR  (5 downto 0);
signal  funct             : STD_LOGIC_VECTOR  (5 downto 0);

-- PC
signal  PC_reg            : STD_LOGIC_VECTOR  (31 downto 0) := x"00000000";

-- PC_mux
-- control signal of PC_mux
signal  PC_Source         : STD_LOGIC_VECTOR  (1 downto 0);-- 0=PC+4, 1=BRANCH, 2=JR, 3=JUMP
```

```
-- input signals to PC_mux
signal  PC_plus_4         : STD_LOGIC_VECTOR  (31 downto 0);
signal  jump_adrs         : STD_LOGIC_VECTOR  (31 downto 0);
signal  branch_adrs     : STD_LOGIC_VECTOR  (31 downto 0);
signal  jr_adrs         : STD_LOGIC_VECTOR  (31 downto 0);
-- output
signal  PC_mux_out        : STD_LOGIC_VECTOR  (31 downto 0);


signal  PC_plus_4_pID     : STD_LOGIC_VECTOR  (31 downto 0);



--================= End of MIPS signals ========================================
--==============================================================================


-- additional "complex" rdbk signals
signal  rdbk_vec1         : STD_LOGIC_VECTOR  (31 downto 0);
signal  rdbk_vec2         : STD_LOGIC_VECTOR  (31 downto 0);




-- ****************************************************************************************************


begin

-- Connecting the Fetch_Unit pins to inner signals
-- ===============================================================
-- MIPS signals    [to be used by students]
CK             <=        CK_25MHz;
RESET         <=        RESET_in;
HOLD         <=       HOLD_in;
MIPS_IMem_adrs      <=  IMem_adrs;
IMem_rd_data    <=    MIPS_IMem_rd_data;
-- RDBK signals   [to be used by students]

--
IR_reg_pID <= MIPS_IMem_rd_data;
sext_imm_pID <= sext_imm;
PC_reg_pIF <= PC_reg;



-- @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
-- your Fetch_Unit code starts here  <<<<@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
-- @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

-- ============================= IF phase processes =====================================
-- ======================================================================================
-- PC register
process (RESET, CK, opcode)
begin
    if RESET='1' then
        PC_reg <= x"00400000";
    elsif CK'event and CK='1' and HOLD ='0' then
        if opcode = 2  or opcode = 3 then
            PC_reg <= jump_adrs;  --@@@HW6
        else
            PC_reg <= PC_mux_out;
        end if ;
    end if;
end process;

IMem_adrs <= PC_reg; -- connect PC_reg to IMem_adrs

-- PC source mux
process (PC_Source, PC_plus_4, branch_adrs, jr_adrs, jump_adrs)
```

```vhdl
begin
    if PC_Source = b"00" then --PC_plus_4
        PC_mux_out <= PC_plus_4;
    elsif    PC_Source = b"01" then --branch_adrs
        PC_mux_out <= branch_adrs;
    elsif PC_Source = b"10" then --jr_instruction
        PC_mux_out <= jr_adrs;
    elsif PC_Source = b"11" then --jump_adrs
        PC_mux_out <= jump_adrs;
    end if;
end process;

-- PC Adder - incrementing PC by 4   (create the PC_plus_4 signal)
PC_plus_4 <= PC_reg + 4;


-- IR_reg   (rename of the IMem_rd_data signal)
IR_reg <= IMem_rd_data;

imm <= IR_reg(15 downto 0);


-- imm sign extension      (create the sext_imm signal)
process (imm, opcode)
begin
    if opcode = 15 then -- lui
        sext_imm <= imm & x"0000"; --@@@HW6
    elsif opcode /= 13 then -- @@@HW6 not ori
        if imm(15) = '0' then
            sext_imm <= x"0000" & imm;
        elsif imm(15) = '1' then
            sext_imm <= x"FFFF" & imm;
        end if;
    else
        sext_imm <= x"0000" & imm;
    end if;
end process;

-- BRANCH address  (create the branch_adrs signal)
branch_adrs <= (sext_imm(29 downto 0) & b"00") + PC_plus_4_pID;

-- JUMP address    (create the jump_adrs signal)
jump_adrs <= PC_plus_4_pID(31 downto 28) & IR_reg(25 downto 0) & b"00";

-- JR address      (create the jr_adrs signal)
jr_adrs <= jr_adrs_in; --@@@HW6
PC_plus_4_pID_out <= PC_plus_4_pID; --@@@HW6

-- PC_plus_4_pID register   (create the PC_plus_4_pID signal)
process (RESET, CK)
begin
    if RESET='1' then
        PC_plus_4_pID <= x"00000000";
    elsif CK'event and CK='1' and HOLD ='0' then
        PC_plus_4_pID <= PC_plus_4;
    end if;
end process;

-- instruction decoder
opcode <=  IR_reg (31 downto 26);
funct  <=  IR_reg (5 downto 0);


-- PC_source decoder  (create the PC_source signal)
process (opcode, funct, Rs_equals_Rt_pID)
begin
    if opcode = b"000010"  or opcode = b"000011" then --j or jal
        PC_source <= b"11"; --jump_adrs
    elsif    opcode = b"000100" and Rs_equals_Rt_pID = '1' then -- beq
        PC_source <= b"01"; --branch_adrs
```

```
    elsif    opcode = b"000101" and Rs_equals_Rt_pID = '0' then -- bne
        PC_source <= b"01"; --branch_adrs
    elsif opcode = b"000000" and funct = b"001000" then -- jr
        PC_source <= b"10"; --jr_instruction
    else --any other
        PC_source <= b"00"; --PC_plus_4
    end if;
end process;


-- @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
-- your Fetch_Unit code ends here    <<<<@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
-- @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@



-- rdbk signals
rdbk_vec1  <=  x"0000000" & b"00" & PC_source; -- add leading zeros to create 32 bit vec






end Behavioral;

-- *****************************************************************************************
-- *****************************************************************************************
```

```vhdl
-- This module is the MIPS General Purpose Register (GPR) file implementation for HW3
--
--
--
---------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity GPR is
Port(

CK              :     in      STD_LOGIC;
rd_reg1       :     in      STD_LOGIC_VECTOR (4 downto 0);-- Rs
rd_reg2       :     in      STD_LOGIC_VECTOR (4 downto 0);-- Rt
wr_reg          :     in      STD_LOGIC_VECTOR (4 downto 0);-- Rd (in R-Type instruction, Rt in LW)
rd_data1      :     out     STD_LOGIC_VECTOR (31 downto 0);-- Rs contents
rd_data2      :     out     STD_LOGIC_VECTOR (31 downto 0);-- Rt contents
wr_data       :     in        STD_LOGIC_VECTOR (31 downto 0);-- contents to be written into Rd (or Rt)
Reg_Write      :     in      STD_LOGIC;-- "0" means no register is written into
GPR_hold      :     in      STD_LOGIC-- "1" means no register is written into

    );
end GPR;


architecture Behavioral of GPR is

--signals used
signal  Equal             : STD_LOGIC;
signal  GPR_rd_data1      :     STD_LOGIC_VECTOR (31 downto 0);-- Rt contents
signal  GPR_rd_data2      :     STD_LOGIC_VECTOR (31 downto 0);-- Rt contents
signal  GPR_data_out1      :     STD_LOGIC_VECTOR (31 downto 0);-- Rt contents
signal  GPR_data_out2      :     STD_LOGIC_VECTOR (31 downto 0);-- Rt contents
signal  GPR_wr_data     :     STD_LOGIC_VECTOR (31 downto 0);-- Rt contents

signal  GPR_we             :     STD_LOGIC;-- the we signal to the memory. made of (Reg_Write and (not GPR_hold))


-- components used
COMPONENT 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 COMPONENT;



begin

GPR_wr_data <= wr_data;


-- produce rd_data1:
-- Here we ensure that reg 0 is always zero
process(rd_reg1, GPR_rd_data1, wr_reg, GPR_wr_data, Reg_Write)
begin
    if rd_reg1 = b"00000" then
```

```
        GPR_data_out1 <= x"00000000";
    elsif rd_reg1 = wr_reg and Reg_Write = '1' then
        GPR_data_out1 <= GPR_wr_data;
    else
        GPR_data_out1 <= GPR_rd_data1;
    end if;
end process;

rd_data1 <= GPR_data_out1;

--process (rd_reg1, wr_reg, Reg_Write) --@@@HW6
--begin
--    if rd_reg1 = wr_reg and Reg_Write = '1' then
--        rd_data1 <= wr_data;
--    else
--        rd_data1 <= GPR_data_out1;
--    end if ;
--end process;


-- produce rd_data2:
-- Here we ensure that reg 0 is always zero
process(rd_reg2, GPR_rd_data2, wr_reg, GPR_wr_data, Reg_Write)
begin
    if rd_reg2 = b"00000" then
        GPR_data_out2 <= x"00000000";
    elsif rd_reg2 = wr_reg and Reg_Write = '1' then
        GPR_data_out2 <= GPR_wr_data;
    else
        GPR_data_out2 <= GPR_rd_data2;
    end if;
end process;

rd_data2 <= GPR_data_out2;

--process (rd_reg2, wr_reg, Reg_Write) --@@@HW6
--begin
--    if rd_reg2 = wr_reg and Reg_Write = '1' then
--        rd_data2 <= wr_data;
--    else
--        rd_data2 <= GPR_data_out2;
--    end if ;
--end process;


GPR_we <= Reg_Write and (not GPR_hold);

-- connecting the GPR 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         =>    GPR_we,
rd1_address   =>    conv_integer(rd_reg1),
rd1_data      =>    GPR_rd_data1,
rd2_address   =>    conv_integer(rd_reg2),
rd2_data      =>    GPR_rd_data2
);


end Behavioral;
```

```vhdl
--
--
-- This module is the MIPS ALU for HW3
--
--
--
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- ***************************************************************************************************
-- ***************************************************************************************************

entity MIPS_ALU is
Port    (
-- ALU operation control inputs
ALUOP        : in STD_LOGIC_VECTOR(1 downto 0);-- 00=add, 01=sub, 10=by Function
Funct        : in STD_LOGIC_VECTOR(5 downto 0);-- 32=ADD, 34=sub, 36=AND, 37=OR, 38=XOR, 42=SLT
-- data inputs & data control inputs
A_in        : in STD_LOGIC_VECTOR(31 downto 0);
B_in        : in STD_LOGIC_VECTOR(31 downto 0);
sext_imm    : in STD_LOGIC_VECTOR(31 downto 0);
ALUsrcB        : in STD_LOGIC;
-- data output
ALU_out        : out STD_LOGIC_VECTOR(31 downto 0)
        );
end MIPS_ALU;


architecture Behavioral of MIPS_ALU is

-- ***************************************************************************************************
-- ***************************************************************************************************


-- inner signals
-- ====================================================
signal  ALU_cmd : STD_LOGIC_VECTOR  (2 downto 0);-- 000=AND, 001=OR, 010=ADD, 011=XOR, 110=sub, 111=slt, 100,101= not used for now
signal  ALU_A_in  : STD_LOGIC_VECTOR  (31 downto 0);
signal  ALU_B_in  : STD_LOGIC_VECTOR  (31 downto 0);
signal  ALU_output : STD_LOGIC_VECTOR  (31 downto 0);

signal  sub_rslt : STD_LOGIC_VECTOR  (32 downto 0);-- use this for creating the sign of sub in SLT instruction
signal  sign_of_sub : STD_LOGIC;

-- Decoded signals for ID phase
signal  LUI : STD_LOGIC;-- '1' when we decode a LUI instruction
signal  ORI : STD_LOGIC;-- '1' when we decode an ORI instruction
signal  JAL : STD_LOGIC;-- '1' when we decode a JAL instruction


begin

--ORI <= '0';


-- ALU
process(ALUOP, Funct, ORI)
begin
    if ALUOP = b"00" then
        ALU_cmd <= b"010"; -- ADD
    elsif ALUOP= b"01" then
            ALU_cmd <= b"110";--  SUB
    elsif ALUOP  = b"11" then -- @@@ ORi HW6
        ALU_cmd <= b"001";

    else
        if Funct = b"100000" then
            ALU_cmd <= b"010"; -- FUNCT=ADD
        elsif Funct = b"100010" then
            ALU_cmd <= b"110"; -- FUNCT=SUB
        elsif Funct = b"100100" then
            ALU_cmd <= b"000"; -- FUNCT=AND
        elsif Funct = b"100101" then
            ALU_cmd <= b"001"; -- FUNCT=OR
        elsif Funct = b"100110" then
            ALU_cmd <= b"011"; -- FUNCT=XOR
        elsif Funct = b"101010" then
```

```
            ALU_cmd <= b"111"; -- FUNCT=SLT
        else
            ALU_cmd <= b"010"; -- ADD
        end if;
    end if;
end process;
--


---- before forwarding
process(ALUsrcB, sext_imm, B_in)
begin
    if  ALUsrcB='0' then
        ALU_B_in <= B_in;
    else
        ALU_B_in <= sext_imm;
    end if;
end process;
ALU_A_in <= A_in;



-- if we consider both inputs as 2's comp numbers then
sub_rslt <= (ALU_A_in(31) & ALU_A_in) - (ALU_B_in(31) & ALU_B_in);
sign_of_sub <= sub_rslt(32);


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); -- NAND
            when b"101" =>   ALU_output <= not(ALU_A_in or ALU_B_in); -- NOR
            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;


ALU_out <= ALU_output;


end Behavioral;

-- ***************************************************************************************************
-- ***************************************************************************************************
```

```vhdl
--
-- dual_port_memory no CK for read for HW3
--
-- Created:
--          by - Danny Seidner, 31/8/2013
--
--

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;

--
ARCHITECTURE dual_port_memory OF dual_port_memory_no_CK_read IS
type Memory_Type is array ((depth-1) downto 0) of std_logic_vector((width-1) downto 0);
shared variable  Memory_array : Memory_Type := (others => (others =>'0')); -- reset initial value to be 0


BEGIN


Memory_wrdata: PROCESS (wr_clk)
begin
if wr_clk'event and wr_clk = '1' then
   if wr_en = '1' then
      Memory_array(wr_address) := wr_data;
   end if;
end if ;
end process Memory_wrdata;


Memory_rddata1 : PROCESS (rd1_address,wr_clk) -- need to add wr_clk, otherwise
                                              -- if we leave rd1_address constant,
                                              -- we won't see changes in rd data even
                                              -- we write new data (in simulation)
begin
   rd1_data <= Memory_array(rd1_address);
end process Memory_rddata1;



Memory_rddata2 : PROCESS (rd2_address,wr_clk) -- need to add wr_clk, see Memory_rddata1 above
begin
   rd2_data <= Memory_array(rd2_address);
end process Memory_rddata2;



END ARCHITECTURE dual_port_memory;
```