BYOC course

Assignment #6

MIPS2 CPU

1. The MIPS2 CPU

In this assignment we add jal, jr, lui, ori instructions to the MIPS1 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/Yu6FFVhl4D4
and the first 11 minutes of: http://youtu.be/-fylybz8p_M

Below we remind you of the MIPS1 CPU we designed in HW5. It is almost the same as the MIPS2 of this assignment

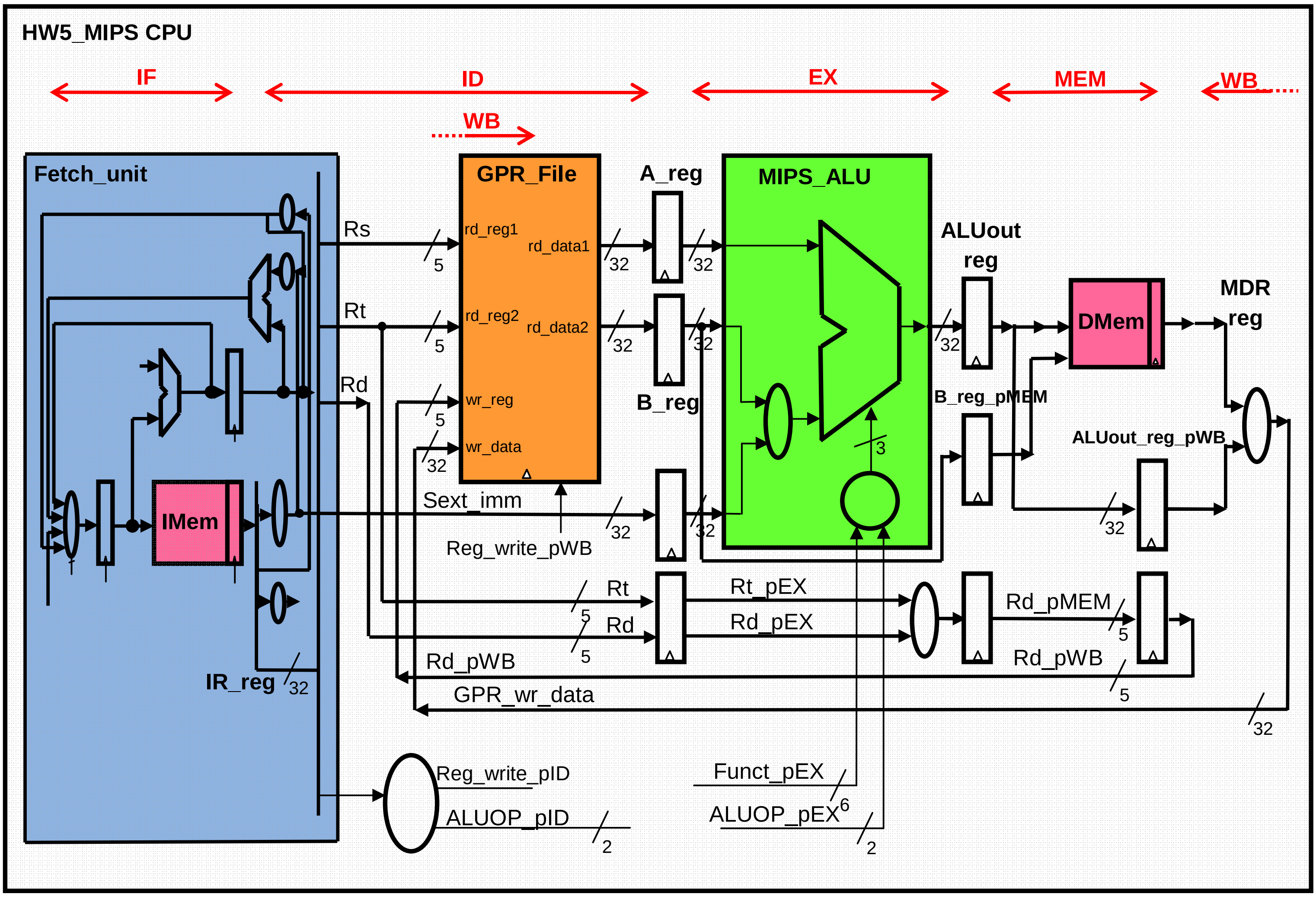

Fig. 1 – The MIPS1 CPU

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 4^{th} 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"00400000" 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 "10" and the IR_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 already have the PC_plus_4_reg_pID that 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 delay the PC_plus_4_reg_pID 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.

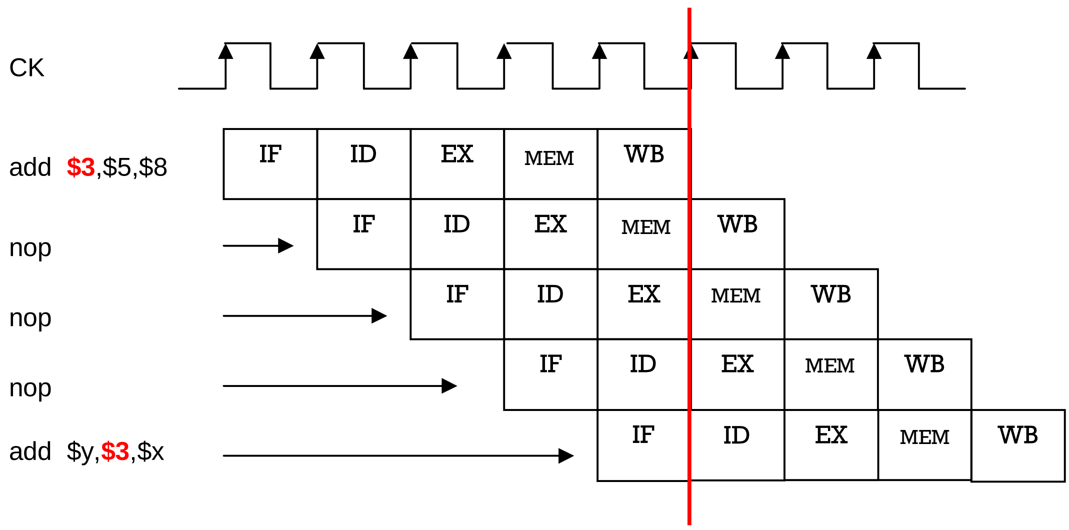

Fig. 2 – The pipelined MIPS latency

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_pMEM='1' and Rd_pMEM=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.

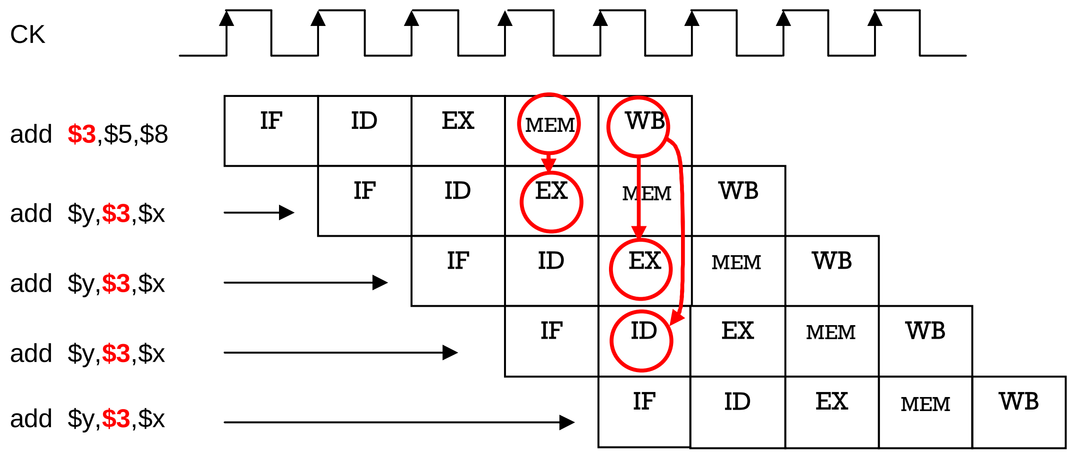

Fig. 3 – Data Forwarding timing diagram
(from the 1st instruction to future instructions)

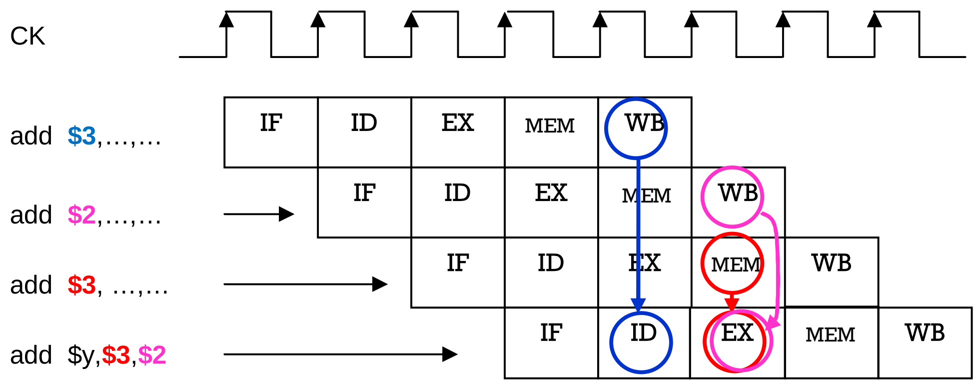

Fig. 3B – The 3 Data Forwarding options to an instruction
(to the 4th instruction from previous instructions)

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.

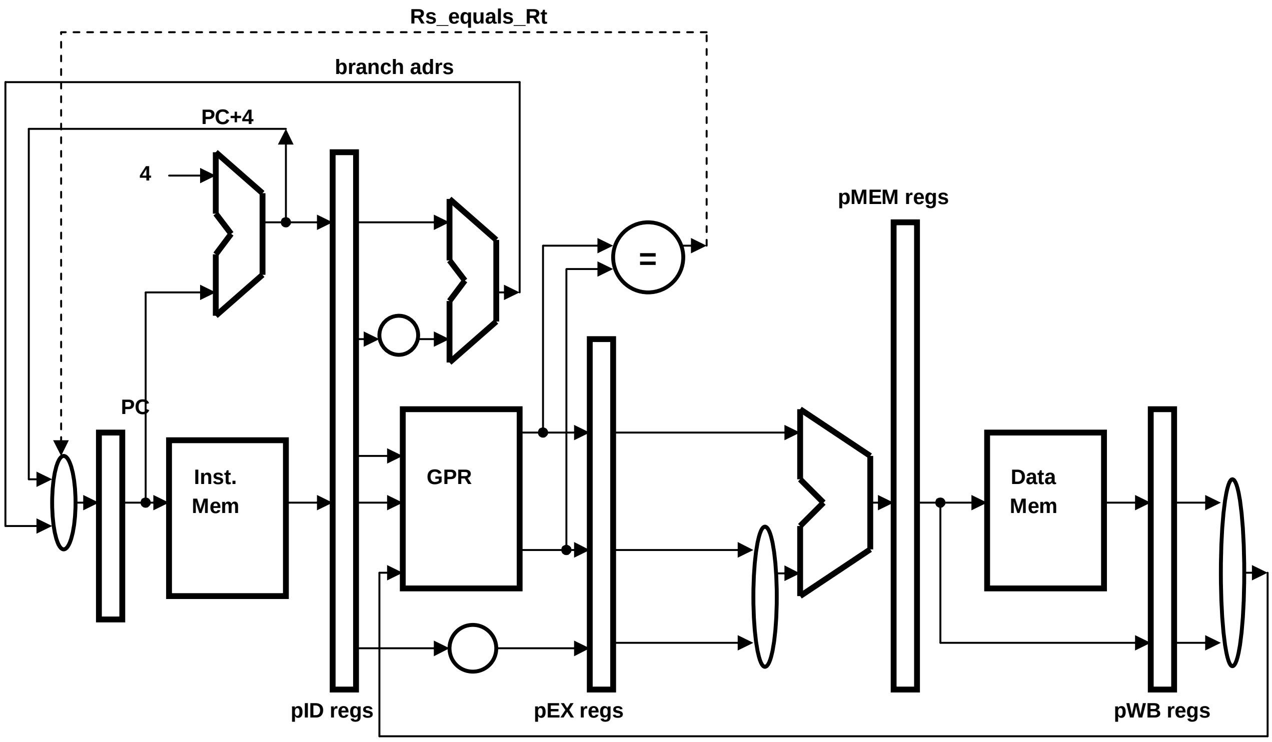

Fig. 4 – MIPS data path (part) with no forwarding

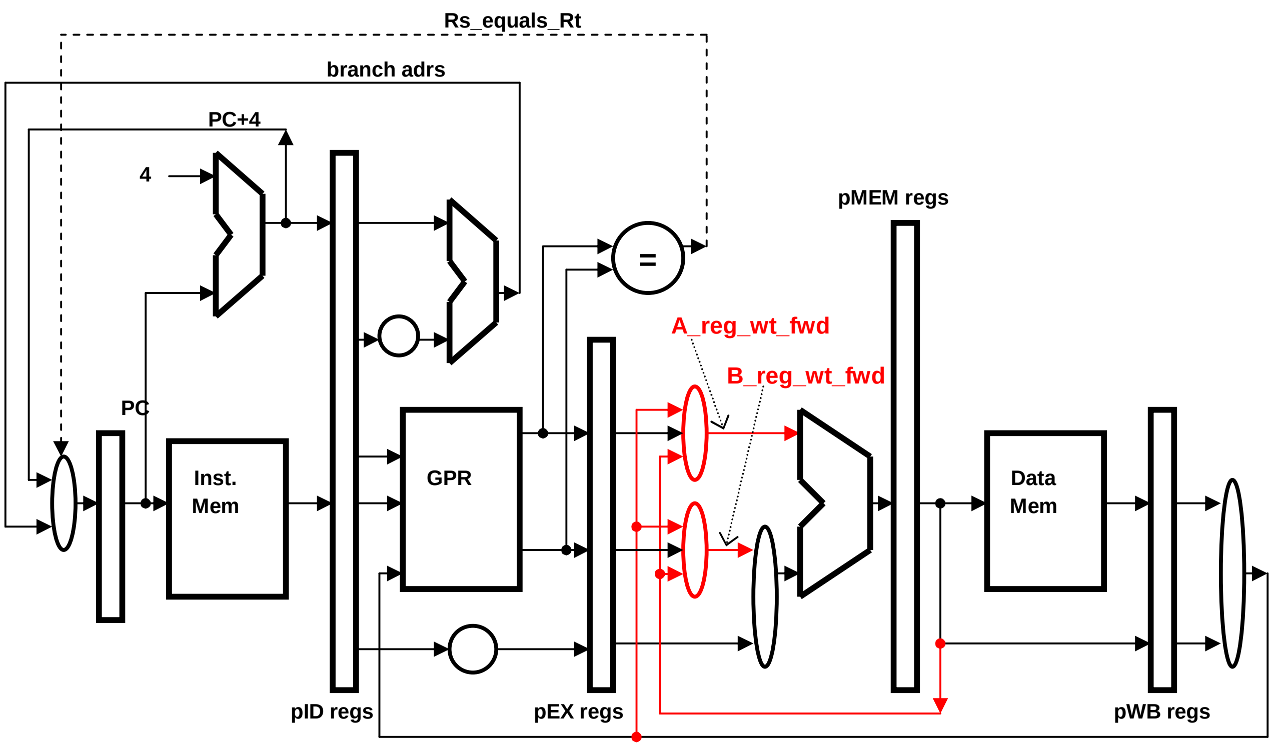

Fig. 5 – MIPS Data Path with Data Forwarding

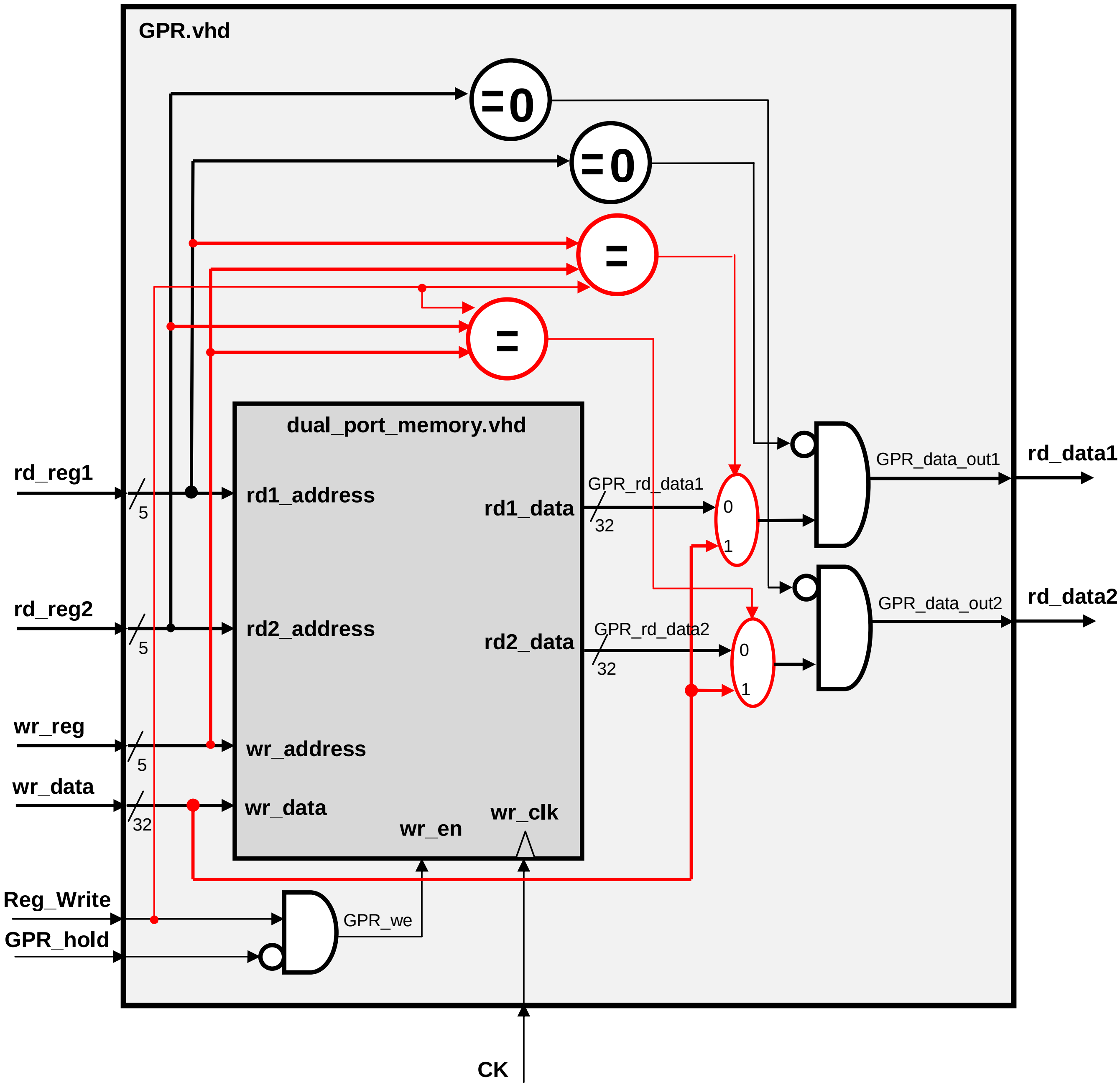

Fig. 6 – MIPS Data Path with Data Forwarding

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.

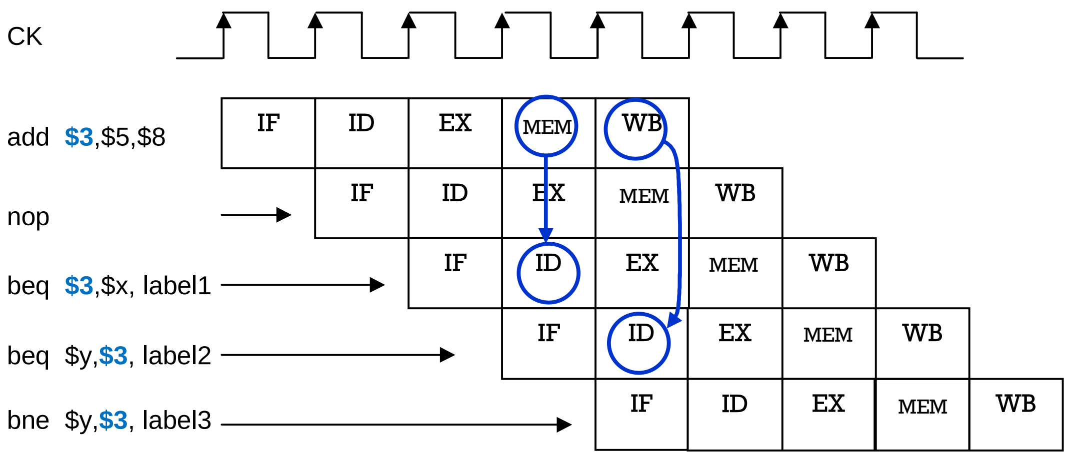

Fig. 7 – Branch Forwarding timing diagram

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 compare 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.

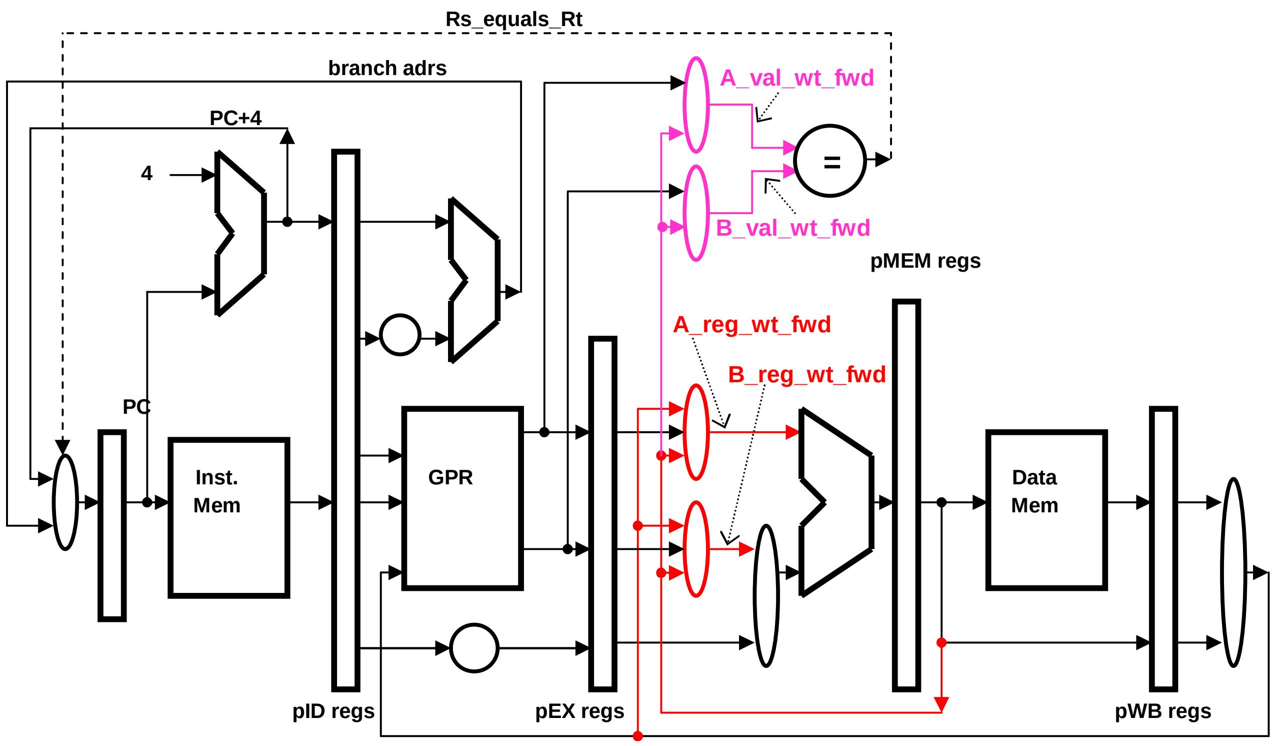

Fig. 8 – MIPS Data Path with Data and Branch Forwarding

The only two signals we added to the HW6 MIPS CPU to support Branch Forwarding are A_val_wt_fwd and B_val_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 A_val_wt_fwd vector signal.

See more in section **e** below.

e. PART II - Names & definition of signals inside the HW6_MIPS CPU

In your design, you should use the exact signal names as were used in the HW5 design and **add** the following signals using the exact signal names shown below. These are already defined in the **HW6_MIPS_4sim-empty.vhd** and the **HW6_MIPS-empty.vhd** files:

ID additional signals

1. jr_address – 32 bit vector signal that has the Rs data value (usually from GPR_rd_data1) to be loaded into the PC_reg in jr instruction
2. A_val_wt_fwd – 32 bit vector signal – the output of the Branch Forwarding mux of Rs value
3. B_val_wt_fwd – 32 bit vector signal – the output of the Branch Forwarding mux of Rt value
4. JAL – ‘1’ when we have jal instruction in the IR_reg (decoded from the opcode)

EX phase signals

5. PC_plus_4_pEX – PC_plus_4_pID (or PC_plus_4) delayed by 1 clock cycle.
6. A_reg_wt_fwd – 32 bit vector signal – the output of the Data Forwarding mux of Rs value
7. B_reg_wt_fwd – 32 bit vector signal – the output of the Data Forwarding mux of Rt value
8. Rs_pEX – Rs delayed by 1 clock cycle.
9. JAL_pEX – JAL delayed by 1 clock cycle.

MEM phase signals

10. PC_plus_4_pMEM – PC_plus_4_pEX delayed by 1 clock cycle
11. JAL_pMEM – JAL_pEX delayed by 1 clock cycle

WB phase signals

12. PC_plus_4_pWB – PC_plus_4_pMEM delayed by 1 clock cycle
13. JAL_pWB – JAL_pMEM delayed by 1 clock cycle

Add more signals according to your needs. Consult the teacher just to be on the safe side.

Additional input in the Fetch_Unit

1. Jr_adrs_in – 32 bit vector signal of the GPR_rd_data1 value to be load into the PC in jr instruction (should be changed to A_val_wt_fwd to support Branch Forwarding). To simplify things we connected it to the jr_address signal of the HW6_MIPS (1st on the list in this page).

f. Names and definitions of output signals from HW6_MIPS CPU to the TB

You need to define all output signals coming out of the **HW6_MIPS_4sim** entity to be tested by the HW6_TB. These signals are the same as we used in **HW6_MIPS_4sim**:

1) CK_out_to_TB - a signal identical to the MIPS_ck "internal" signal
2) RESET_out_to_TB - a signal identical to the MIPS_reset "internal" signal
3) HOLD_out_to_TB - a signal identical to the MIPS_hold "internal" signal
4) rdbk0_out_to_TB to rdbk15_out_to_TB – 16 vector signals, 32 bit each that will have the data we want to check – as detailed below.

In your **HW6_MIPS_4sim** design you should connect the rdbk signals as follows:

<u>ID signals:</u>
rdbk0 => PC_plus_4_pID (PC_plus_4 signal in HW6_MIPS)
rdbk1 => IR_pID (IR_reg signal in HW6_MIPS),
rdbk2 => sext_imm_pID (sext_imm signal in HW6_MIPS)
rdbk3 => Rs, Rt, Rd, Funct (Rs= bits 28:24, Rt= bits 20:16, Rd= bits 12:8, Funct= bits 5:0)
rdbk4 => RegWrite, Rs_eq_Rt, MemWrite
(Reg Write= bit 28, MemWrite= bit 24, Rs_eq_Rt=bit0)
<u>EX signals:</u>
rdbk5 => ALUsrcB_pEX, ALUOP, Funct_pEX
(ALUsrcB=bit 28, ALUOP= bits 9:8, Funct_pEX = bits5:0)
rdbk6 => A_reg,
rdbk7 => B_reg,
rdbk8 => sext_imm_reg,
rdbk9 => ALU_output,

<u>MEM signals:</u>
rdbk10 => ALUOUT_reg
rdbk11 => B_reg_pMEM

<u>MEM & WB control signals</u>
rdbk12 => MemWrite_pMEM (bit31), MemToReg_pMEM (bit28), RegWrite_pMEM (bit24), Rd_pMEM (bits 20:16),
MemToReg_pWB (bit12), RegWrite_pWB (bit8), Rd_pWB (bits 4:0)

<u>WB signals:</u>
rdbk13 => MDR_reg
rdbk14 => ALUOUT_reg_pW
rdbk15 => GPR_wr_data

By the way, These are the exact same signals we used in HW5 TB. As ususal, the TB will compare the expected data of these signals to the actual result of the simulation and will tell you where the errors are. It uses a file called **HW6_TB_data.dat** containing the expected results. Since you will use these signals in the implementation phase, you should also connect them to the Host Interface [all connections are already done for you in the **HW6_MIPS_4sim-empty.vhd** file].

g. Simulation of the 3 designs – describing the 3 projects

In all the 3 design you should run simulations. We will give you an “empty” **HW6_MIPS_4sim.vhd** file that has all of the additional new signals for the 3 parts already inside. The name and definitions of the new signals appear in section **f** below. The names and definition of the TB signal appear in section **e**. They are similar to those used in HW5. The required simulation reports are described in section **f**. Then we have the implementation instructions and the Implementation report instructions.

The files we will use to run the simulation are:

1) **HW6_MIPS_4sim.vhd** – This is your design of HW6. It uses your designs of the Fetch_Unit, GPR, MIPS_ALU and the pre-prepared HW6_Host_Intf_4sim. We give you an “empty” version that has all of the additional signals for the 3 parts already inside. You should take your **HW5_MIPS.vhd** design and copy the VHDL code that you wrote into the appropriate location in the Architecture part (after the general signals and before the components connections) of the **HW6_MIPS_4sim.vhd**. You should then add the necessary changes to form the HW6 MIPS as explained in sections **b**, **c** and **d**.
2) **Fetch_Unit.vhd** - The Fetch Unit you designed in HW2 after the modifications of HW4. **A change is required here in HW6!!**
3) **GPR.vhd** – your GPR File design you designed in HW3.
4) **dual_port_memory.vhd** – part of the GPR File you designed in HW3.
5) **MIPS_ALU.vhd** – your MIPS_ALU design prepared in HW3. **A change is required in HW6!!**
6) **HW6_Host_Intf_4sim.vhd** – The pre-prepared component including the IMem, DMem and “pre-loaded” program and data. This component also creates the reset & hold signals to the rest of the HW6_MIPS CPU.
7) **HW6_TB_for_students.vhd** - The TB vhd file prepared in advance.
8) **HW6_TB_data.dat** – this is a data file prepared in advance that has the expected TB values. It is read by the HW6_TB and used to compare the actual simulation results to the expected ones. You will have 3 versions of this file: **HW6_TB_data_no_fwdng.dat**, **HW6_TB_data_wt_data_fwdng.dat** and **HW6_TB_data_wt_data_&_branch_fwdng.dat**. Again, remember to modify the path of the **HW6_TB.dat** file inside the **HW6_TB_for_students.vhd** .

2) <u>HW6 Simulation report</u>

You should submit a single zip file for the Simulation and implementation phases. It should have four directories/folders. The first is called **Simulation1**, the 2nd is called **Simulation2,** the 3rd is called **Simulation 3**, the 4th is called **Implementation**. In the Simulation folders you will have 3 sub-folder of:

- **Src_4sim** – here you put all of the *.vhd sources for simulation
- **Sim** – here you should have the HW6_4sim project created by the simulator you used
- **Docs** – Here you put your simulation report. The first few lines in the report will have your ID numbers (names are optional). See the instructions below for the rest of the simulation report.

Simulation1 will have the “no forwarding” design of part I where you add the lui, ori, jr and jal instructions. You should answer the questions in **Appendix A** and insert these to your report of **Simulation1**.

Simulation2 will have the “data forwarding” design of part II where you add the data forwarding to the design of **Simulation1**. You should answer the questions in **Appendix B** and insert these to your report of **Simulation2**.

Simulation3 will have the “branch forwarding” design of part III where you add Branch Forwarding to the design of **Simulation2**. You should answer the questions in **Appendix C** and insert these to your report of **Simulation3**.

Later, in the Implementation phase you will add 2 sub-folders to the **Implementation** folder. These will be:

- **Src_4ISE** – here you put all of the *.vhd sources and the *.ucf file (and no TB file)
- **ISE** – here you should have the HW6_MIPS project created by the Xilinx ISE SW.

3) HW6 - MIPS2 CPU – implementation

After a successful simulation we want to implement the design on the Nexys2 board. The same change done in HW5 to go from the simulation version to the implementation version are required here too.

We can rename the **HW6_MIPS_4sim.vhd** to **HW6_MIPS.vhd** and then, remove all the signals that were outputted to the TB (see **1f** above). These are not required anymore. Then we should add all of the signals that are connected to the Nexys2 board functions. These are described in the **BYOC_HW5.doc** file. A much easier way is to use the pre-prepared **HW6_MIPS-empty.vhd** file which includes all of the Nexys2 signals, rename it to **HW6_MIPS.vhd**, and copy your VHDL code from your **HW6_MIPS_4sim.vhd** file into the appropriate location inside the **HW6_MIPS.vhd** file.

The files we will use to implement the design on the Nexys2 board are:

1) **HW6_MIPS.ucf -** The file listing which signal are connected to which FPGA pins in the Nexys2 board.
2) **HW6_MIPS.vhd** – This is your design of HW6. It uses the GPR, MIPS_ALU and the Fetch_Unit. This should be based on the 3rd simulation version that supports the new istructions (ori, lui, jr, jal) and data and branch forwarding.
3) **GPR.vhd** – your GPR File design you prepared in HW3 and modified during the HW6 simulation.
4) **dual_port_memory.vhd** – part of the GPR File design you prepared in HW3.
5) **MIPS_ALU.vhd** – your MIPS_ALU design you prepared in HW3 and modified during the HW6 simulation.
6) **Fetch_Unit.vhd** - The Fetch Unit you prepared in HW2 after the modifications of HW4, HW5 and HW6.
7) **BYOC_Host_Intf.ngc** – This prepared component includes the infrastructure interfacing to the PC allowing us to load programs into IMem and data into DMenm and read feedback signals in single clock mode. It also has the VGA controller and the KBD and Flash interfaces. We give that component in the form of a single netlist file, which is an already compiled version of the vhd files forming the BYOC_Hos_interface.

To allow an easy debugging you should connect the rdbk signals (eventually connected to the BYOC_Host_intf) as follows [same as in the simulation & implementation part and in HW5 – again, these are all connected for you in the **HW6_MIPS-empty.vhd** file]:

ID signals:
rdbk0 => PC_plus_4_pID
rdbk1 => IR_pID,
rdbk2 => sext_imm_pID
rdbk3 => Rs, Rt, Rd, Funct (Rs= bits 28:24, Rt= bits 20:16, Rd= bits 12:8, Funct= bits 5:0)
rdbk4 => RegWrite, Rs_eq_Rt, MemWrite
(Reg Write= bit 28, MemWrite= bit 24, Rs_eq_Rt=bit0)

EX signals:
rdbk5 => ALUsrcB_pEX, ALUOP, Funct_pEX
(ALUsrcB=bit 28, ALUOP= bits 9:8, Funct_pEX = bits5:0)
rdbk6 => A_reg,
rdbk7 => B_reg,
rdbk8 => sext_imm_reg,
rdbk9 => ALU_output,

MEM signals:
rdbk10 => ALUOUT_reg
rdbk11 => B_reg_pMEM

MEM & WB control signals
rdbk12 => MemWrite_pMEM (bit31), MemToReg_pMEM (bit28), RegWrite_pMEM (bit24), Rd_pMEM (bits 20:16),
MemToReg_pWB (bit12), RegWrite_pWB (bit8), Rd_pWB (bits 4:0)

WB signals:
rdbk13 => MDR_reg
rdbk14 => ALUOUT_reg_pW
rdbk15 => GPR_wr_data

So we'll run that the BYOCInterface SW and load the IMem. Then run the circuit in a single ck mode and check that the reading we see at the points we "hooked" to the rdbk signals are as what we expect.

The file we want to load into the IMem is called "Pong1_v32.txt". Following the loading, we can run in single ck mode and see the readback values on the PC screen after each clock. If you press the run button, you should get the Pong game running on the VGA screen.

If it works, GREAT!!!

You **B**uilt **Y**our **O**wn **C**omputer!!

4) Implemetation report

You should submit a single zip file that has the simulations and implementation projects in 4 folders as described in section 2 above. In the **Implementation** folder you will include your entire ISE implementation project in two sub-folder that will be as follows:

- **Src_4ISE** – here you put all of the *.vhd sources and the *.ucf file (and no TB file)
- **ISE** – here you should have the HW6_MIPS project created by the Xilinx ISE SW.

Including a copy of the bit file you created at the **Implementation** directory.

As part of completing this part of the course you will have to show me how you run the design on the Nexys2 board in the lab. And answer some questions.

Enjoy the assignment !!

At the end of this assignment you will have our final CPU and run a simple computer game on the CPU that you yourself built. That is great! I hope you enjoyed the course.