BYOC course

Assignment #2

Fetch unit

1) Description of the Fetch unit

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

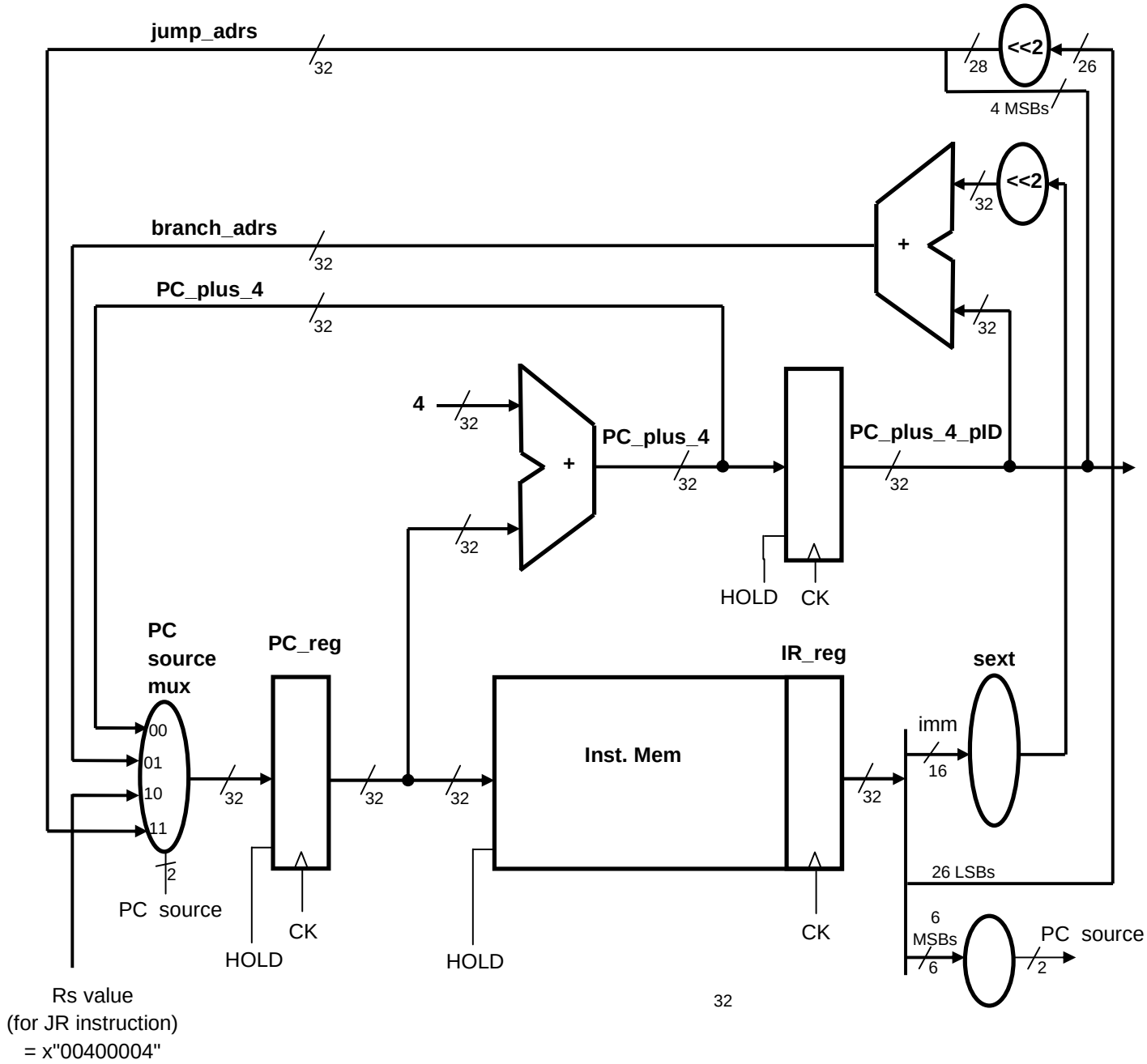

Fig. 1 – The Fetch Unit

The Fetch Unit's main components are the PC register (PC_reg) and Instruction Memory (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.
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[31:26] & IR[25:0] & b"00", i.e., the jump address in words multiplied by 4. This is the address to be loaded into the PC when a jump or a jal instruction is performed.
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 for running the design later 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 include that signal.

b. Description of the Fetch_Unit project

You need to define all signals in your design. Then, you should write the equations of the PC_reg, PC_plus_4 adder, branch_adrs, jump_adrs, jr_adrs, PC_source mux, sext_imm and the instruction decoder producing the PC_source signal.

This should be followed by simulation. For simulation you will use an IMem component prepared in advance (it is part of a component called the Fetch_Unit_Host_Intf_4sim) and the Fetch_Unit_TB.vhd file that is also prepared in advance.

After succeeding in the simulation phase, you will replace the IMem for simulation with another version, suitable for implementation (instead of the Fetch_Unit_Host_Intf_4sim you will use the Fetch_Unit_Host_Intf component. This component has inside a mechanism enabling loading of “program” into the IMem via a RS232 cable from a PC computer). Then we will create a BIT file and load and test the design on the Nexys2 board.

These stages will be explained in detail below.

2) Fetch unit project – first part – design & simulation

Figure 2 below describe the simulation project scheme:

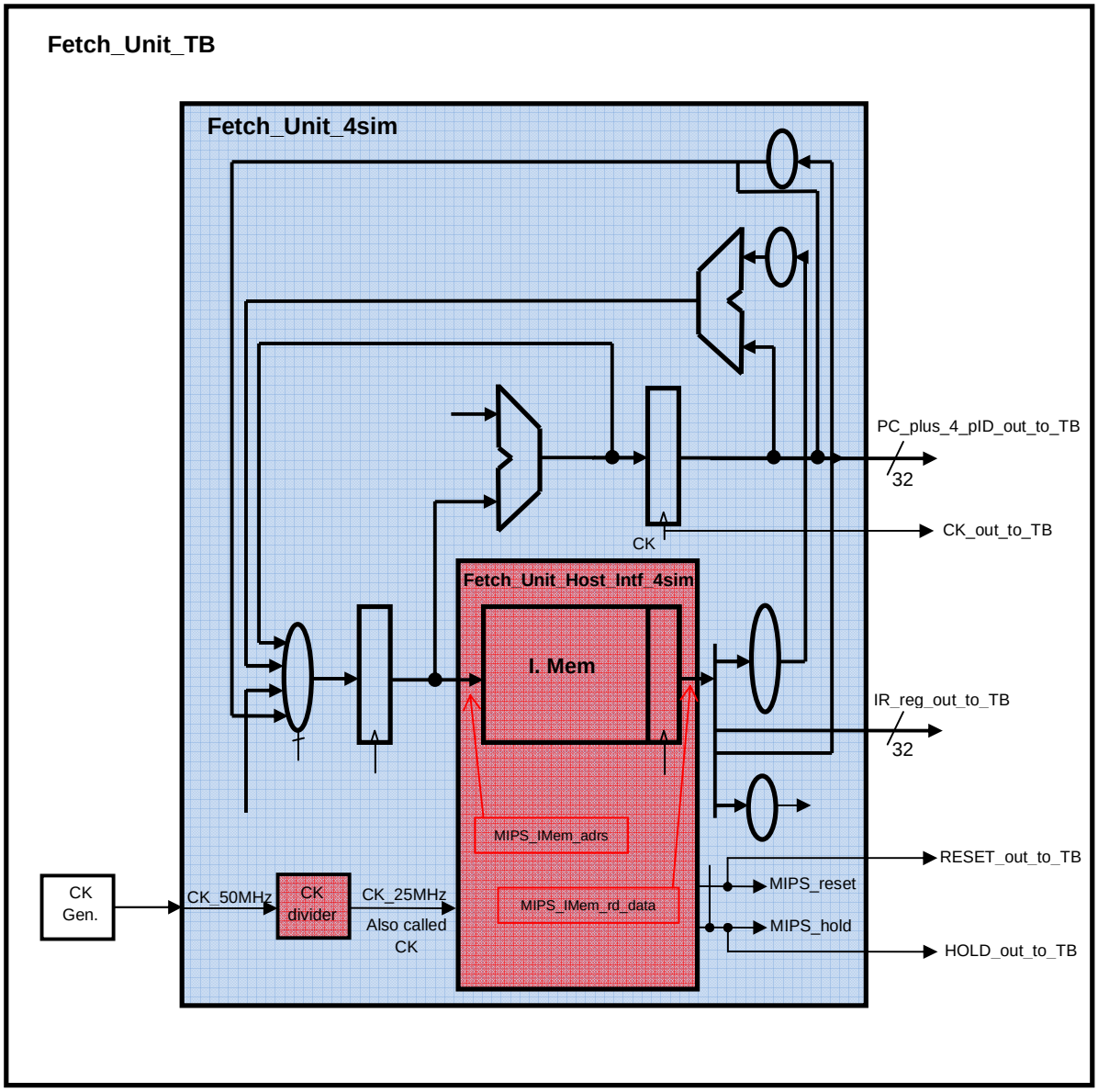

Fig. 2 – The simulation project – 1st version

The light blue part is the part you design. It is called the **Fetch_Unit.vhd**. Actually, you start with a simulation version, so it should be called the **Fetch_Unit_4sim.vhd**. Your design includes all signals, registers and logic we saw in Figure 1 except the IMem and IR itself and except the clock divider. These are implemented for you in advance in two components. The first is called **Clock_driver.vhd** and the second is a component called **Fetch_Unit_Host_Intf_4sim**.

The **Clock_driver.vhd** component has a T-FF that divides the 50 MHz CK input to 25 MHz output (We us CK as an abbreviation to CLOCK). It also has a special driver, **BUFG**, that can drive many FFs and registers. The entire design is driven by that CK signals. We will feed all other components with the output clock of 25 MHz. We wil also rename it to CK and use it for the Fetch unit design.

The ISIM simulator is familiar with the special clock driver we use inside the **Clock_driver.vhd** component. However, if you use the Modelsim simulator, you need to use another file that was prepared for simulation only called **Clock_driver_for_Modelsim.vhd** instead. The **BUFG** inside is removed.

The **Fetch_Unit_Host_Intf_4sim** component has already some "program data" inside it. The 32 address lines of the IMem are called MIPS_IMem_adrs and the 32 output data lines are called MIPS_IMem_rd_data. We will drive the MIPS_IMem_adrs from the PC_reg and will direct the MIPS_IMem_rd_data lines (rename them) to the IR_reg signal.

Besides having the IMem inside it, this component also outputs two signals that are required for the Fetch_Unit design. These are MIPS_reset and the MIPS_hold signals. You will make sure registers reset signal is connected to the MIPS_reset signal coming out of the **Fetch_Unit_Host_Intf_4sim**. The MIPS_hold signal will drive the HOLD signal we mentioned earlier. You will make sure that every FF and register in the Fetch Unit will be "frozen" when the HOLS signal is '1'.

The Test Bench (TB) design "wraps" your **Fetch_Unit_4sim** design and supply the required signals. In our case that is the CK_50MHz signal only (for now). The TB should also look at some output signals from the **Fetch_Unit_4sim** so it can test them and verify their correctness. When running simulation, you can see on the screen waveforms of all signals in the design, "internal" and "external". However, the TB needs to get the signals we want to check as outputs from the design under test. Thus we will output the following signals from the **Fetch_Unit_4sim** design:

1) RESET_out_to_TB - a signal identical to the MIPS_reset "internal" signal
2) CK_out_to_TB - a signal identical to the MIPS_ck "internal" signal
3) HOLD_out_to_TB - a signal identical to the MIPS_hold "internal" signal
4) IR_reg_out_to_TB - a signal identical to the IR_reg "internal" signal
5) PC_plus_4_pID_out_to_TB - a signal identical to the PC_plus_4_pID "internal" signal

During simulation, the TB we prepared reads a data file containing the expected signal values and compares them to the actual signal values coming out of your **Fetch_Unit_4sim** tested design. It reports errors to the simulation console screen. This will make it easier for you to debug your design.

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

1) **Fetch_Unit_4sim.vhd** - your design implementing figure 1
2) **Clock_driver.vhd** (or **Clock_driver_for_Modelsim.vhd** – if you use the Modelsim simulator)

3) **Fetch_Unit_Host_Intf_4sim.vhd** – The prepared components including the IMem and creating the reset & ck signals
4) **Fetch_Unit_TB.vhd** - The TB vhd file prepared in advance
5) **Fetch_Unit_TB_data.dat** - The data file read by the Fetch_Unit_TB.vhd during simulation.

We even prepare an "empty" **Fetch_Unit_4sim.vhd** file in which we already did the following:

- Defined the I/O pins of the Fetch_Unit_4sim design
- Defined the components used (Clock_driver and Fetch_Unit_Host_Intf_4sim)
- Defined all necessary internal signal inside the Fetch_Unit_4sim design
- Connected the Clock_driver component to the required internal signals
- Connected the Fetch_Unit_Host_Intf_4sim component to the required internal signals
- Connected other signals, e.g., signals to be outputted to the TB

You now have to write the equations describing the logic circuitry of Figure 1. For that we also listed the required functionality inside the "empty" file – a remarks.

Until now we did not tell you that the Fetch_Unit_Host_Intf_4sim component has the same i/o pins as the Fetch_Unit_Host_Intf component we will use in the next phase of the project – the real implementation phase. In that phase we use the real Fetch_Unit_Host_Intf component that has the IMem that can be loaded with any program via the PC. That component therefore includes a hidden mechanism that can communicate with the PC via RS232 interface. The hidden mechanism has additional features. It is connected to the 8 switches and 4 of the push-buttons existing on the Nexys2 board. This means that the Fetch_Unit_Host_Intf component has additional i/o pins which we did not mention yet. We built the Fetch_Unit_Host_Intf_4sim component with the same i/o pins as the Fetch_Unit_Host_Intf component. So we have these additional signals also in out simulation design. These signals are:

1) RS232_Rx – Input signal from the RS232 cable. We will not use that at all in the simulation.
2) RS232_TX – Output signal to the cable (we will connect it to '1' in the simulation design)
3) buttons_in(3 downto 0) - input signal from 4 buttons. Will not be used in the simulation.
4) switches _in(7 downto 0) - input signal from 8 switches. Will not be used in the simulation.
5) leds _out(7 downto 0) - output indication to the 8 on-board Leds. Will not be used in the simulation.
6) rdbk0(31 downto 0) – a 32 bit read back signal that will be used in real implementation to read from a point in the design and display it on the PC screen when working in a single_ck mode during debugging of the design.
   Actually there are 16 vectors like that denoted rdbk0-15.

The updated drawing describing the simulation scheme appears in Figure 3 below.

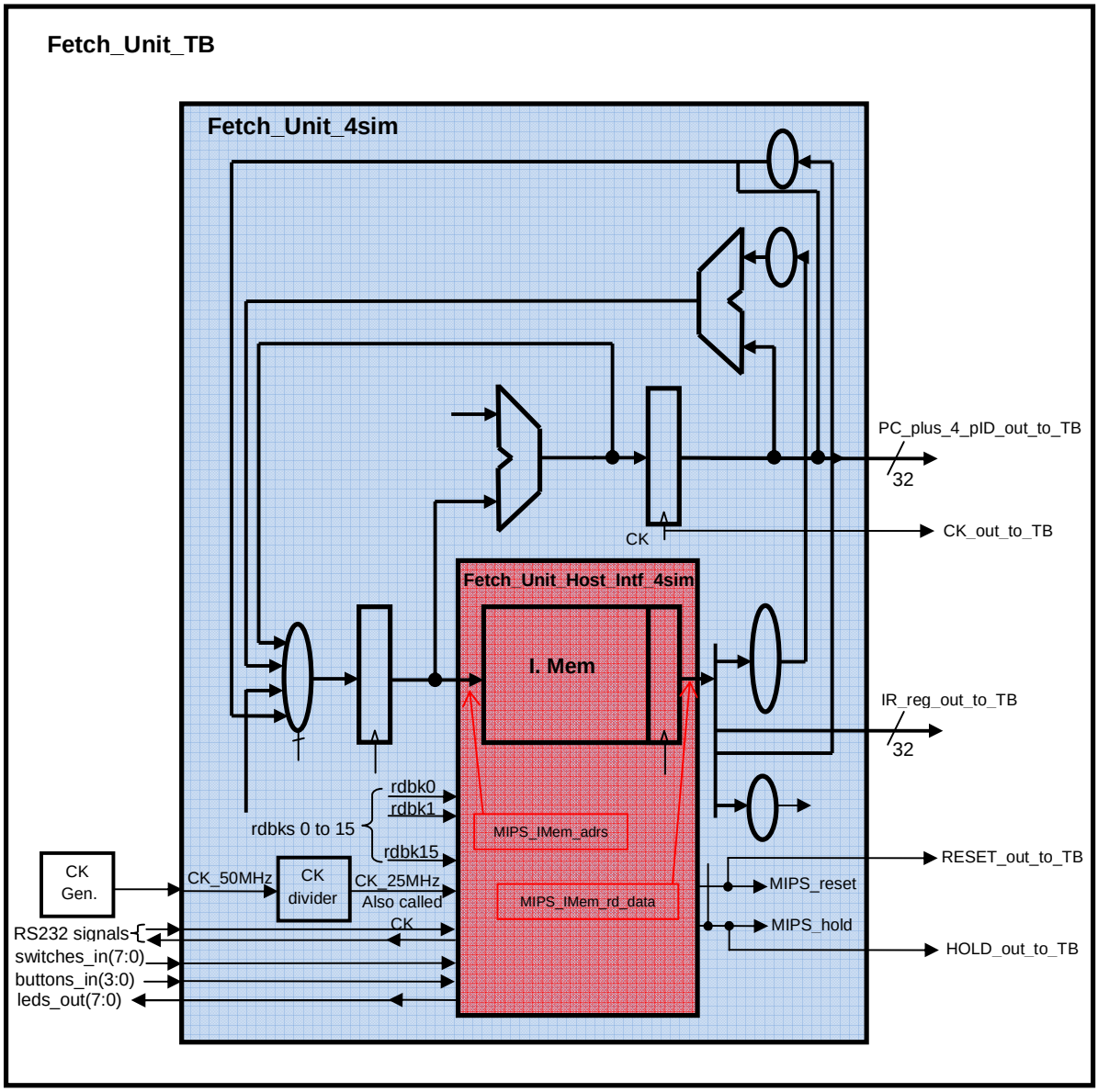

Fig. 3 – The simulation project – the full picture

3) Simulation report

You should submit a zip file of your entire simulation project.

Run the simulation to 3500 nS.

Also you need to attach a doc file with screen captures describing the simulation you made. All signals mentioned in section 1a above should be presented in the screen capture. Show at least the 1st 10 ck cycle following the end of the reset pulse and make the values of all signals readable.

In that doc file you need to answer the following questions:

3.1) Say we have a value V in location Adrs in the IMem. What will be the expected value of the Tested_PC_plus_4_pID signal in the TB when the Tested_IR_reg signal has the value V? Adrs? Adrs-4? Adrs+4? Other? Explain your reply.

3.2) In the IMem we have the value of a bne instruction (x"14000002") in address x”400014”. This is a bne instruction. Do we expect to actually branch? Who checks the condition?

3.3) Did we branch in the simulation? If so how come we also see the values of Tested_PC_plus_4_pID =x”40001C” and Tested_IR_reg =x”00006666” ? These are the value of the next instruction! If we branched, where is the branch seen?

4) Fetch unit project – 2nd part - implementation

After a successful simulation we want to implement the design on the Nexys2 borad. A few changes are required. First we will take our **Fetch_Unit_4sim.vhd** file and rename it to **Fetch_Unit.vhd** as well as removing the 5 signals we outputted for simulation. The Picture now is as described in Figure 4 below:

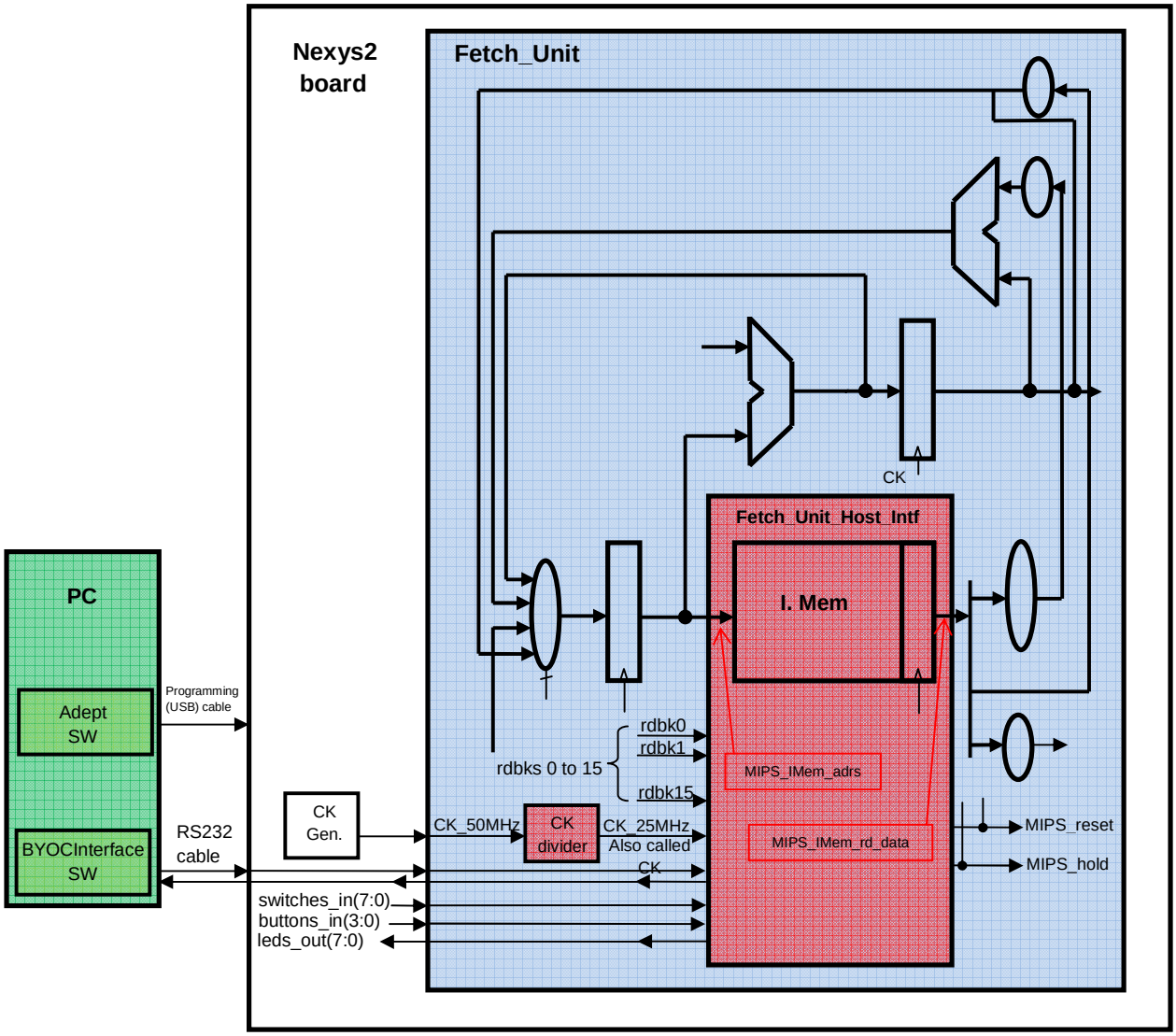

Fig. 4 – The Fetch Unit implementation scheme

We see that the signals we outputted towards the TB are removed. We see that the entire Fetch_Unit design (described in the **Fetch_Unit.vhd** file drawn in light blue) is somehow connected to the real world – i.e., to the Nexys2 board. It resides "inside" the board. Inside it we have the component **Fetch_Unit_Host_Intf.vhd** instead of the **Fetch_Unit_Host_Intf_4sim.vhd** we used in the simulation step.

So how do we "connect" the i/o pins of the **Fetch_Unit.vhd** to the Nexys2 board? We do that by listing the connections between i/o pin signals and the FPGA real i/o pins. This list is given in the **Fetch_Unit.ucf** file where UCF stands for User Constraints File.

The **Fetch_Unit_Host_Intf.vhd** is a file that hides a more complicated design called the **BYOC_Host_Intf**. That design includes the MIPS Instruction Memory (which we already aware of), the MIPS Data Memory, a circuit called a VGA controller that displays part of the DMem on a monitor screen – if we connect such a monitor to the Nexys2 board, a connection to a P/S2 keyboard that can be also connected to the Nexys2 board and more functions. In our current project we are not going to use all of these extra parts. We are only using the MIPS IMem, the reset and hold signals and the 16 rdbk signals which we can connect to any point inside the **Fetch_Unit.vhd** design. Thus, we hid all of these extra functions and "wrapped" the **BYOC_Host_Intf** with the **Fetch_Unit_Host_Intf.vhd** file. When implementing the design we need to add that **BYOC_Host_Intf** which is a component inside the **Fetch_Unit_Host_Intf.vhd** file. This component is described in the **BYOC_Host_Intf.ngc** file. That file does not have the equations but the netlist created as part of the compilation (like the difference between the source code and its compiled object or exe file in SW compilation). The list of files we need to have in order to "compile" our project is therefore:

1) **Fetch_Unit.ucf** - The file listing which sgnasl are connected to which FPGA pins in the Nexys2 board.
2) **Fetch_Unit.vhd** - your design implementing figure 1.
3) **Clock_driver.vhd** – The clock divider & driver
4) **Fetch_Unit_Host_Intf.vhd** – The prepared components including the IMem and the one creating the reset & ck signals.
5) **BYOC_Host_Intf.ngc** - The infrastructure interfacing to the PC which is part of the Fetch_Unit_Host_Intf

When we have these 5 files we can compile them on the Xilinx ISE SW and produce a **Fetch_Unit.bit** file which is the file we load into the FPGA in order to configure it to implement our design.

When we have the **Fetch_Unit.bit**, we can load it into the FPGA inside the Nexys2 board by connecting a USB cable from an external PC into the mini-USB input on the Nexys2 board and activating the Adept SW application that supports "loading" a design into the Nexys2 board. The Nexys2 board and the **Adept** SW as developed by a company called Digilent.

OK. We loaded the design. But what about loading the IMem with data? The design we created has an empty Instruction Memory. Furthermore, say we loaded the design with some program and run it. How do we know it really works? We need some means or tool to be hooked to the signals inside the FPGA (or to direct them to external pins in the FPGA so we can really connect to them) and "read" the values of the signal and display them somewhere (on the screen as waveforms or as data values). How do we do that?

In order to load the MIPS IMem with data, and in order to read data from desired points in the design we use the **BYOCInterface** SW. This SW can communicate with the BYOC_Host_Intf component via a RS232 cable connected from the PC to the Nexys2 board.

So we'll run that SW. 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.

We connected the rdbk signals as follows:

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

In order to "teach" the BYOCInterface SW the names of these signals we have a "**labels.txt**" file in which we write the name of the signals we hooked to in the same order (The name of the signal connected to rdbk0 is written in the first line of the **labels.txt** file. The name of the signal connected to rdbk1 in is the 2nd line, etc.).

The file we want to load into the IMem is called "**HW2_Fetch_Unit_IMem_load.txt**". The file itself includes all the information required in order to load it into the IMem and switch to a single ck mode. Following the loading, we can run in single ck mode and see the readback values on the PC screen after each clock.
We can also ask the BYOCInterface to compare the values on the screen to the expected values by selecting a comparison to a file called "**HW2_Fetch_Unit_TB_data_for_BYOCIntf_SW.dat**". If we do that we will see errors on the screen in red.

5) Implametation report

You should submit a zip file of your entire implementation project.
Including a copy of the bit file you created at the top 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 a complete Fetch Unit which is the basis for our next designs.