The BYOC course -VHDL implementation of a simplified MIPS CPU in a lab course

Danny Seidner

School of Computer Science
College of Management Academic Studies - COMAS
Rishon-LeZion
Israel

Outline

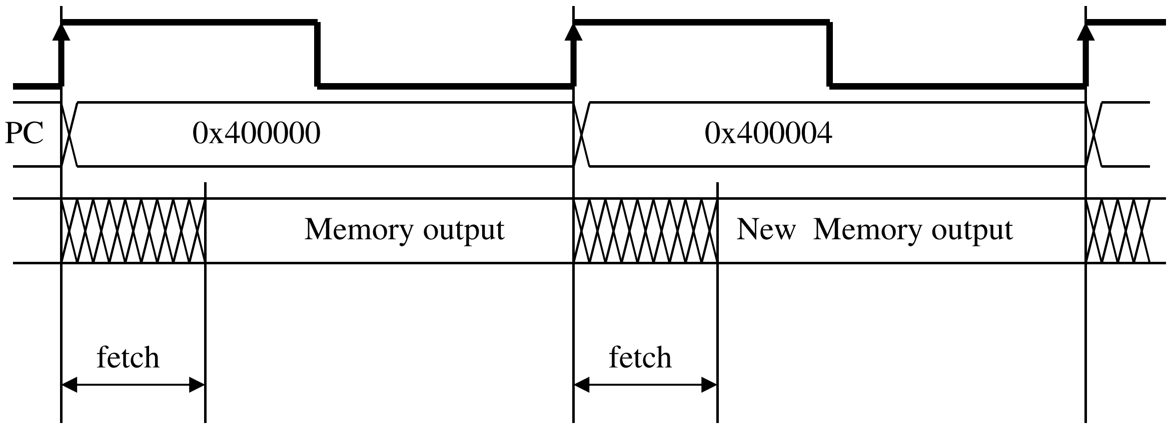
- Introduction
- How we teach Single Cycle implementation
- Use the same approach for teaching pipelined impl.
- FPGA design cycle
- BYOC course infrastructure
  - Support in Lab exercise & source files
  - Simulation infrastructure
  - Implementation infrastructure
- Summary

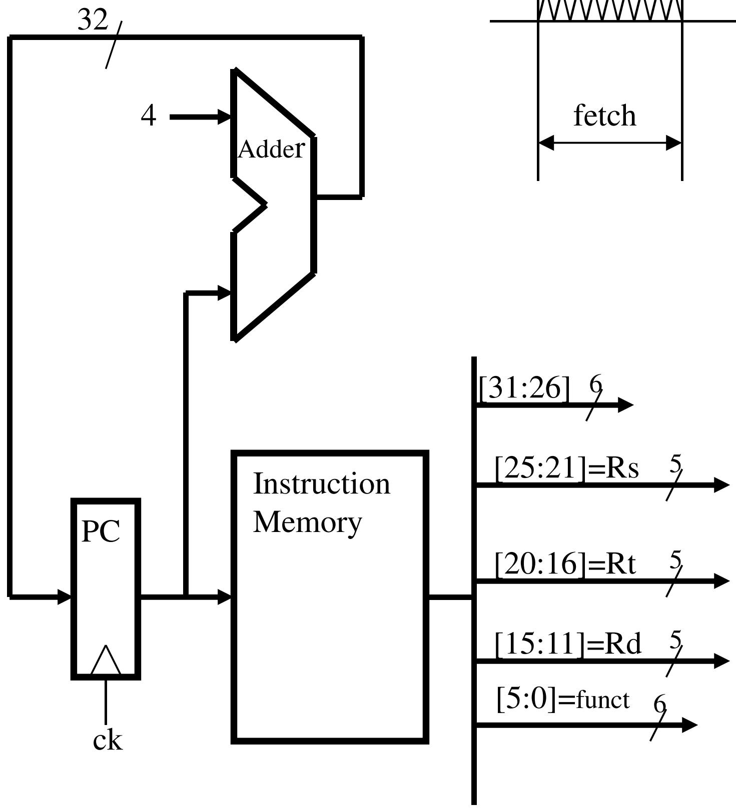
Introduction

- **Many universities & colleges base their Computer structure course on Patterson & Hennessy's "Computer Organization & Design – the Hardware/Software interface"**
- **Their approach is to build the CPU in steps:**
  - **Steps in building the Single Cycle implementation**
  - **Go from Single Cycle to Multi-Cycle and then to pipelined version**
- **We follow this approach for a lab course in which the students actually implement a simplified pipelined MIPS CPU**
- **This paper describes the course and the infrastructure we built allowing control on the effort required from the student**
- **Thus, we can adjust the course to different populations – from Computer Science programmers to Electrical Engineering students**

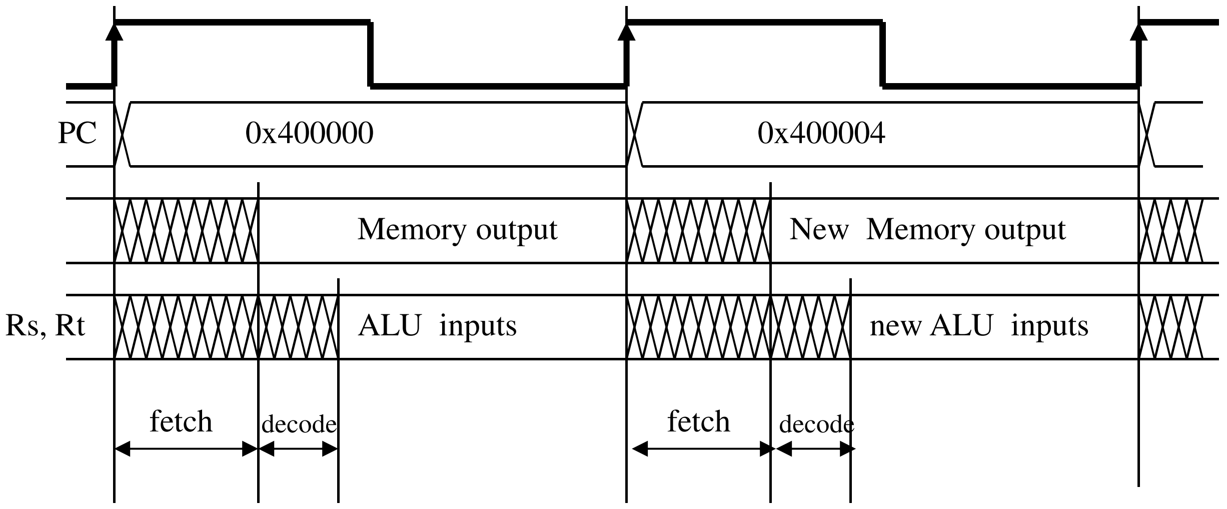
How we teach Single Cycle implementation:

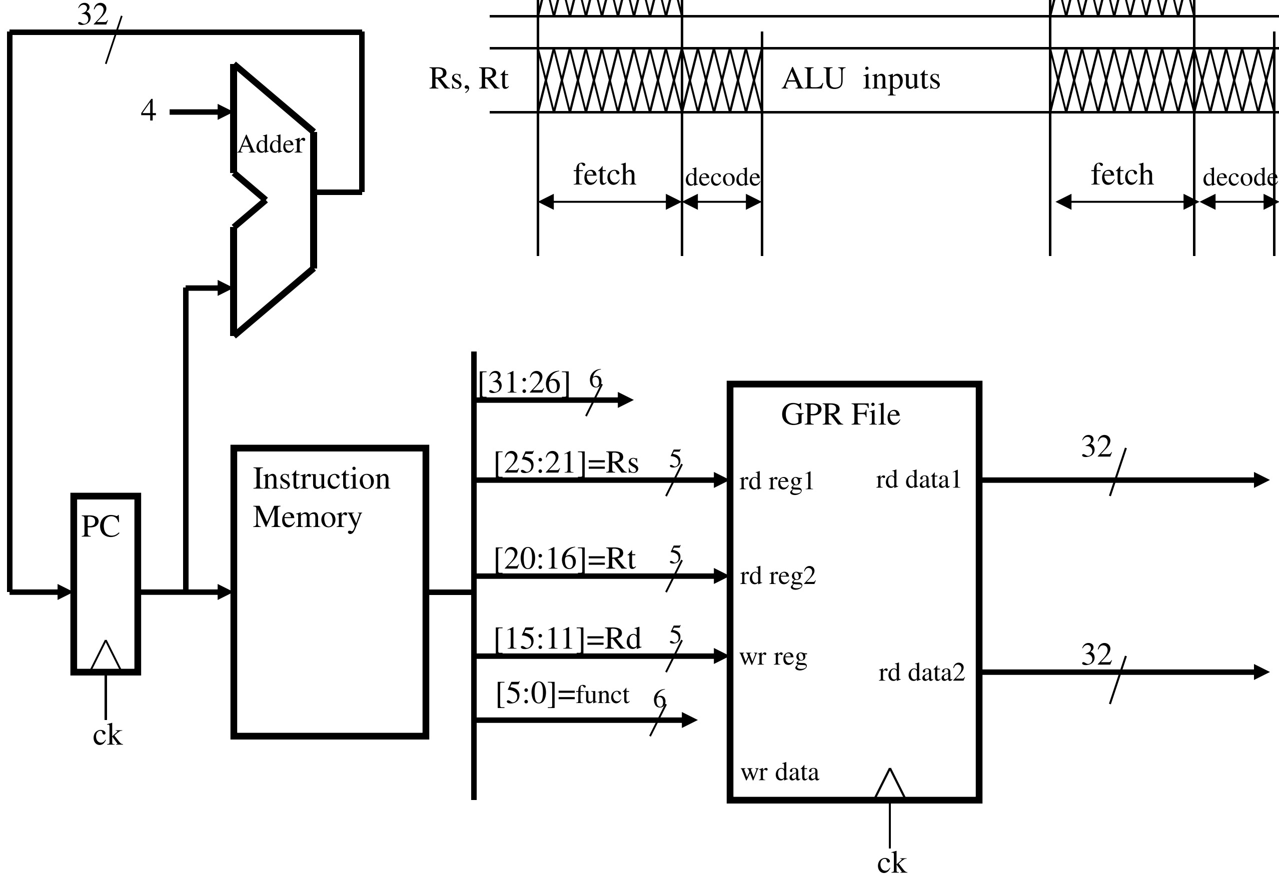
- **We start with the FETCH phase – of a simple R-Type CPU** **(R-Type inst. only)**
    - Reading the instruction from Inst. Mem.
- **Then describe the DECODE phase**
    - Describing The GPR File – that has all 32 General Purpose Regs
    - Showing how Rs & Rt data is read from the GPR File
- **Followed by the EXECUTE phase**
    - The ALU gets the Rs & Rt data and calculates result
- **And finally, the Write Back phase**
    - Where the calculated result is written back into Rd in the GPR file



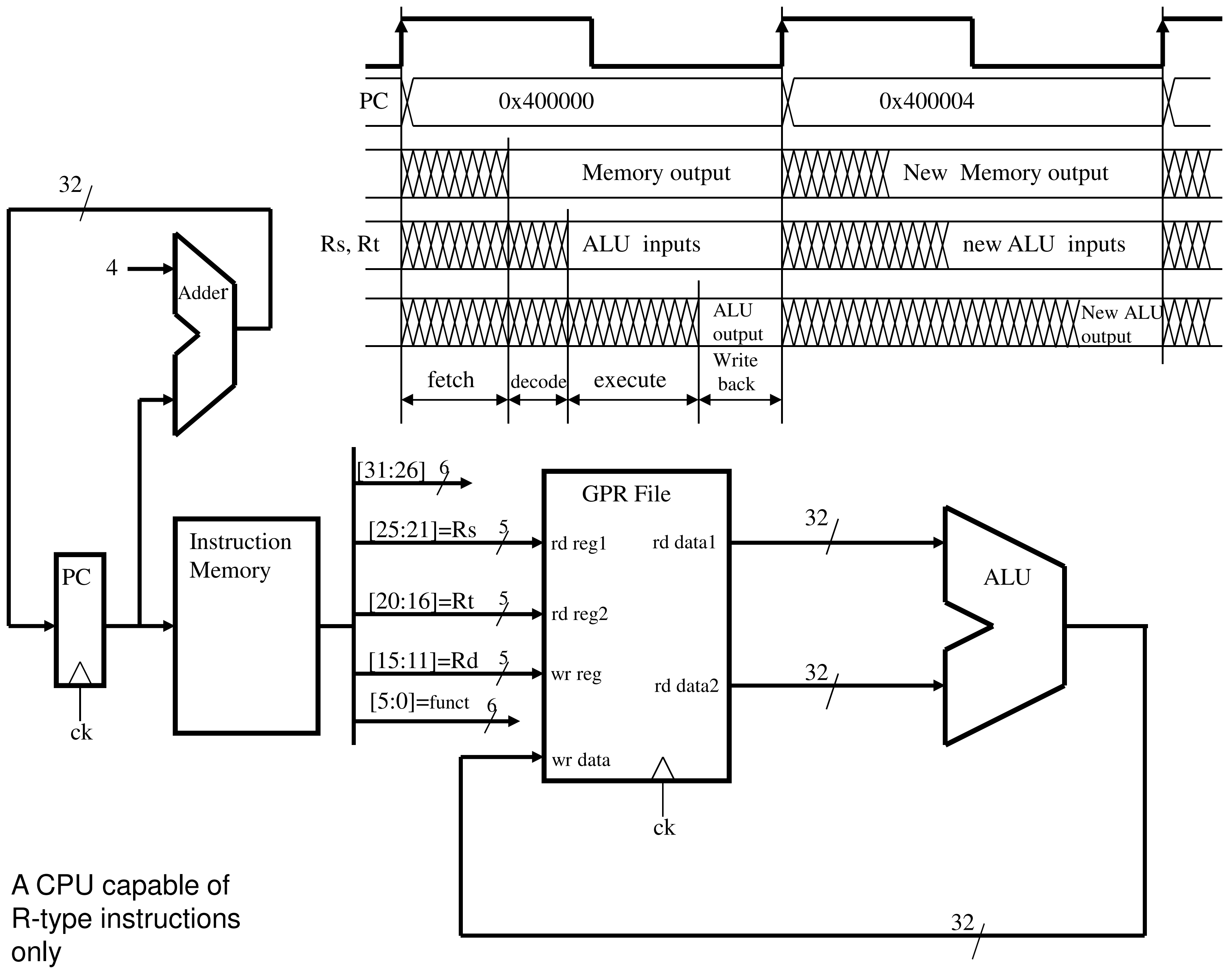


A CPU capable of
R-type instructions
only



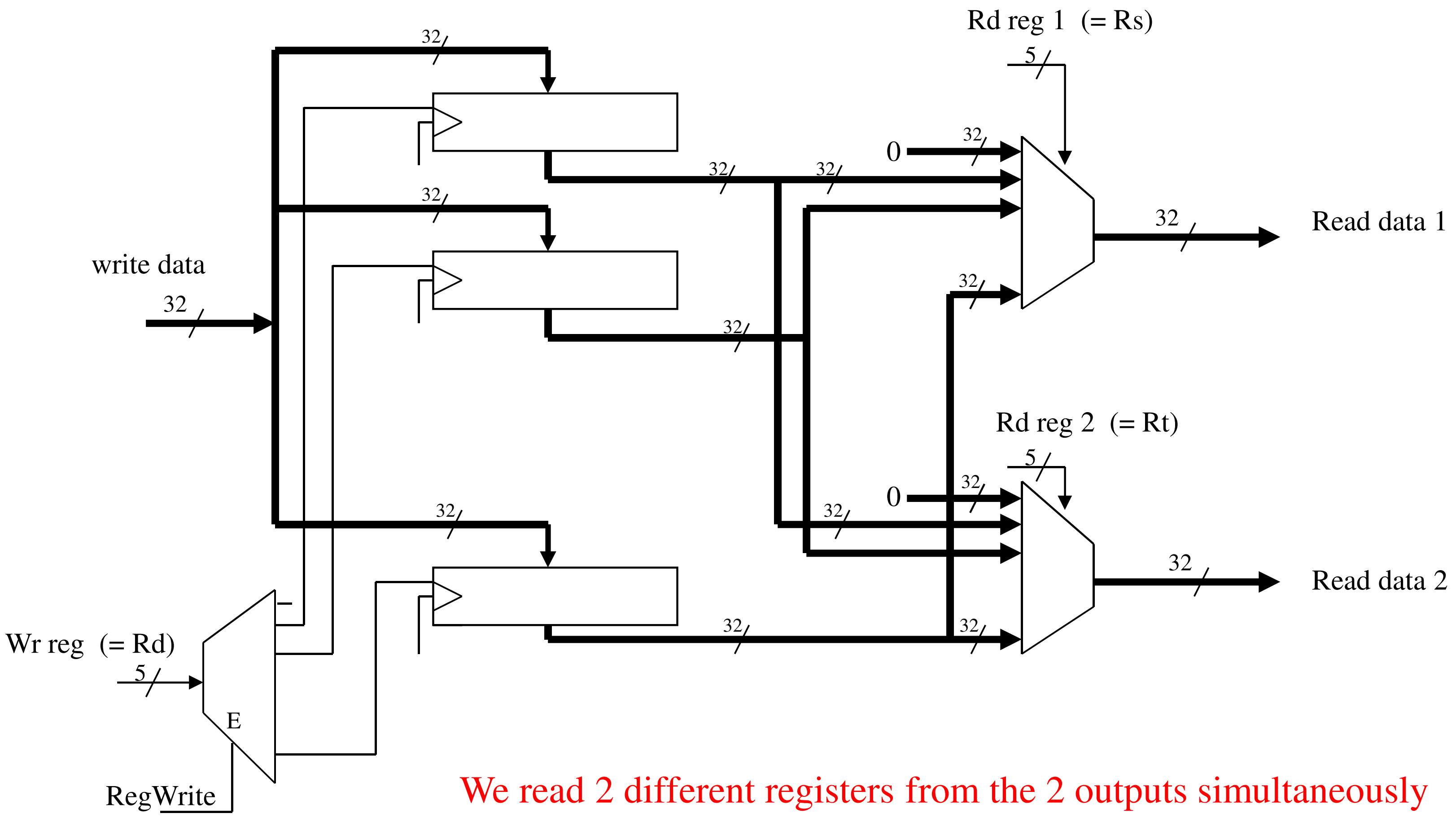


A CPU capable of
R-type instructions
only



A CPU capable of R-type instructions only

The internal structure of the GPR File

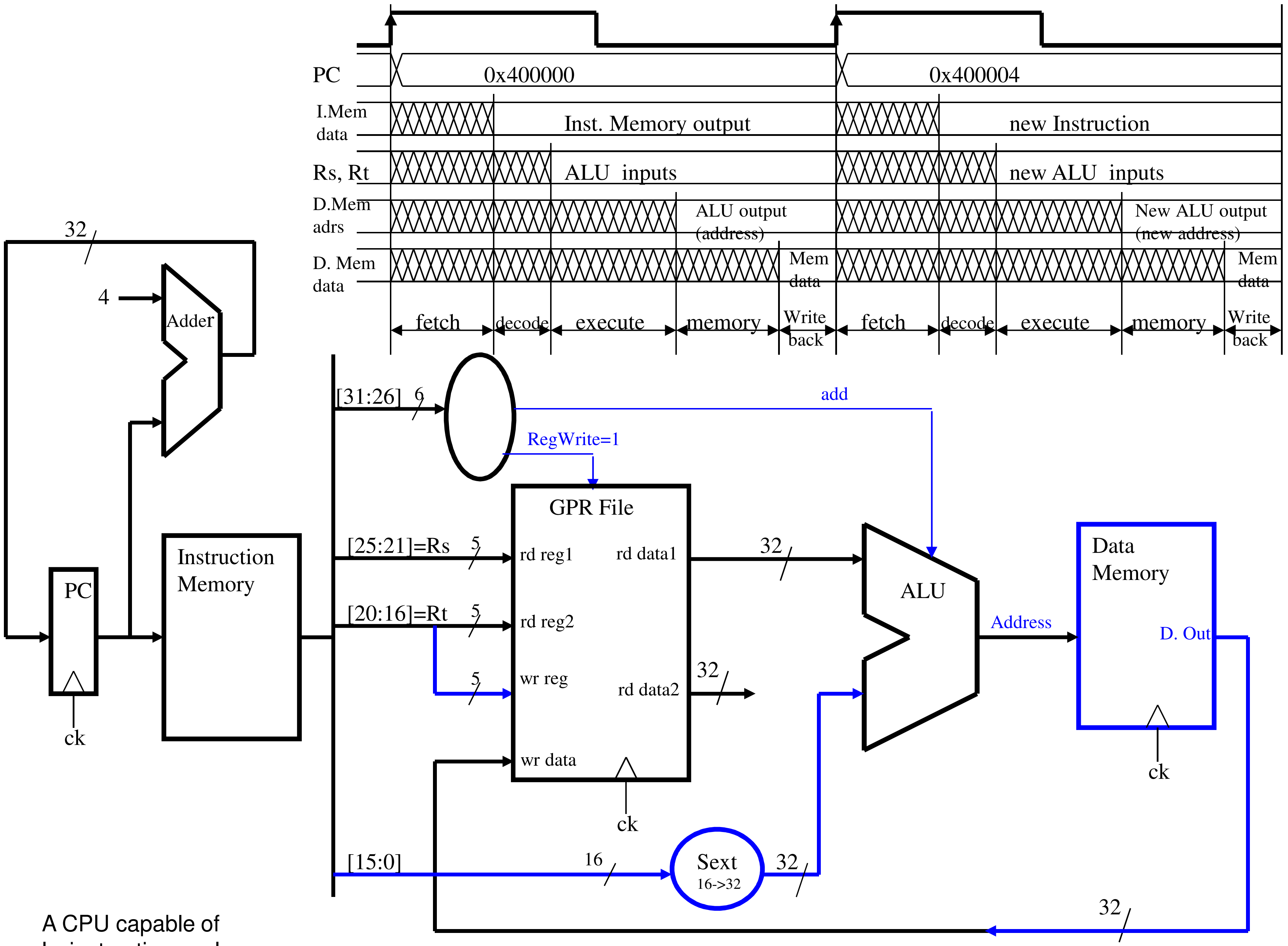


We read 2 different registers from the 2 outputs simultaneously
We write to one of the registers (in the next rising edge of the CK).

Register #0 does not really exist

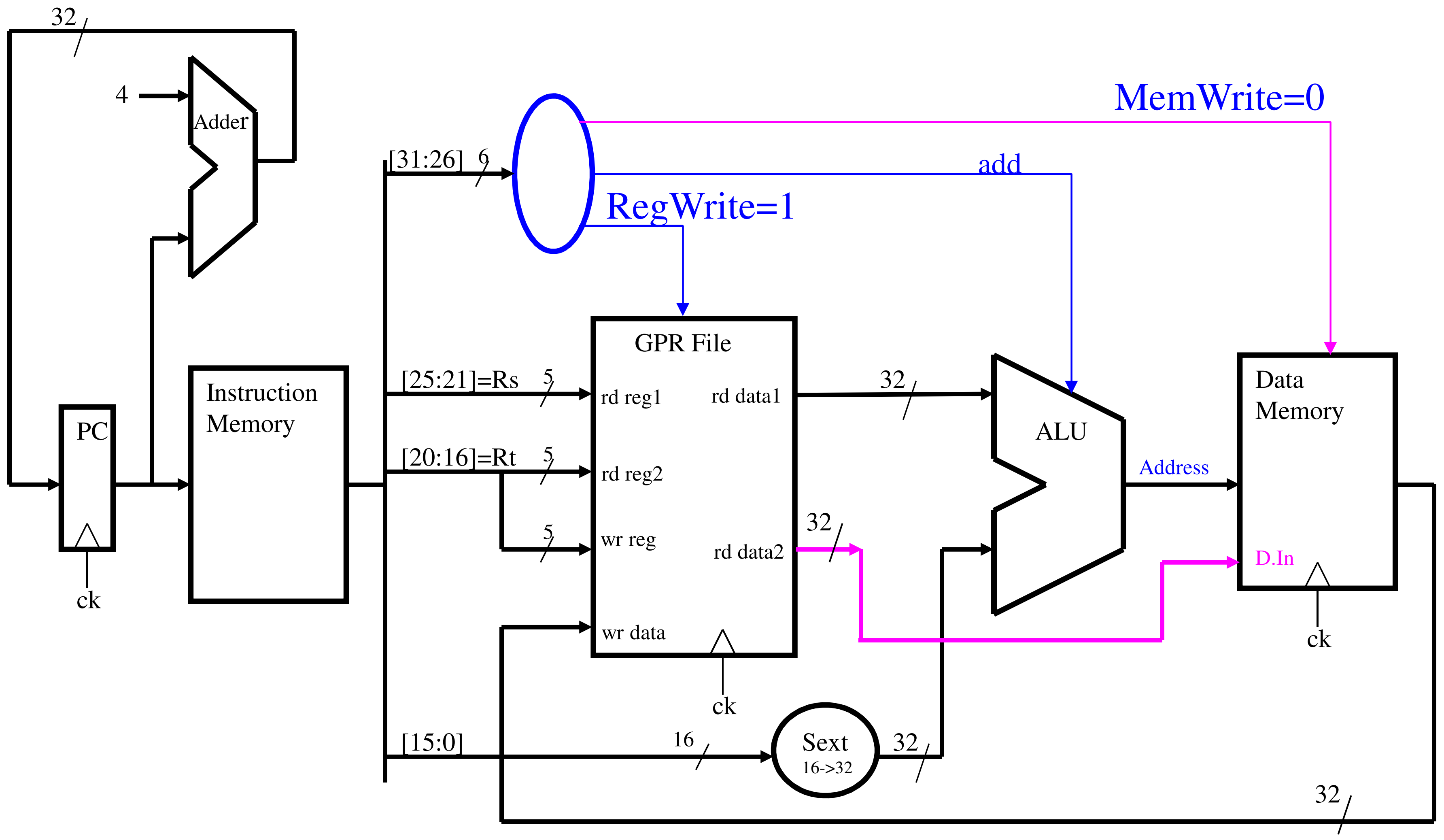
Cont. with the Single Cycle implementation:

- **Next we show a Single Cycle CPU capable of LW instructions only**
  - It has FETCH, DECODE, EXECUTE, MEMORY & WRITE BACK phases
  - It had a Data Memory as well
- **Then we add support of SW instruction**
  - Same Data Path - Just apply “1”-s to the right control signals
- **Next we combine the two CPUs – Rtype only with LW & SW only**
  - Combining the 2 Data Paths requires a few MUX-s
- **Finally we add other instructions**
  - The data path is changed to support BEQ and J instructions
  - Control decoder is then explained in detail

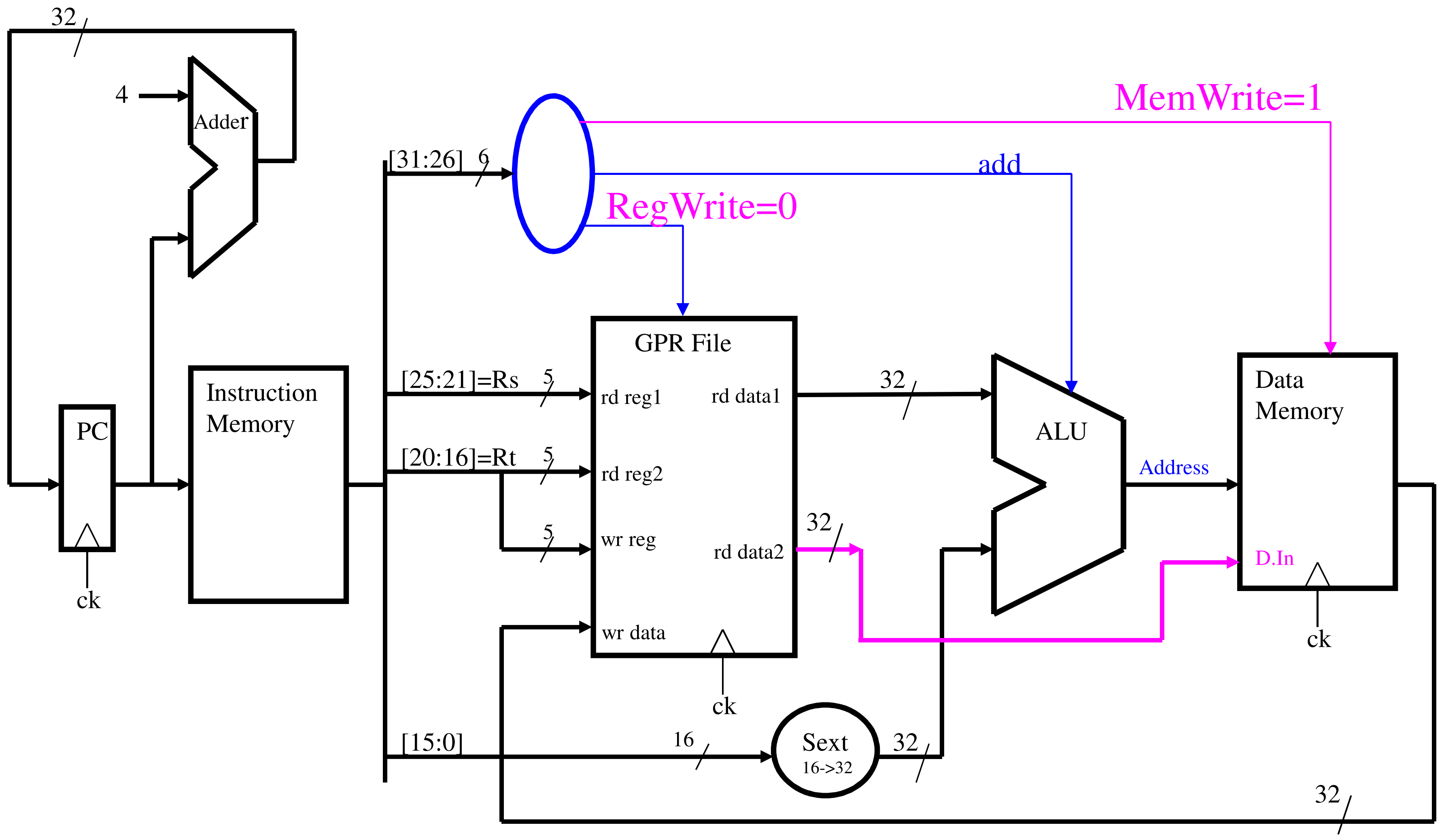


A CPU capable of
lw instructions only

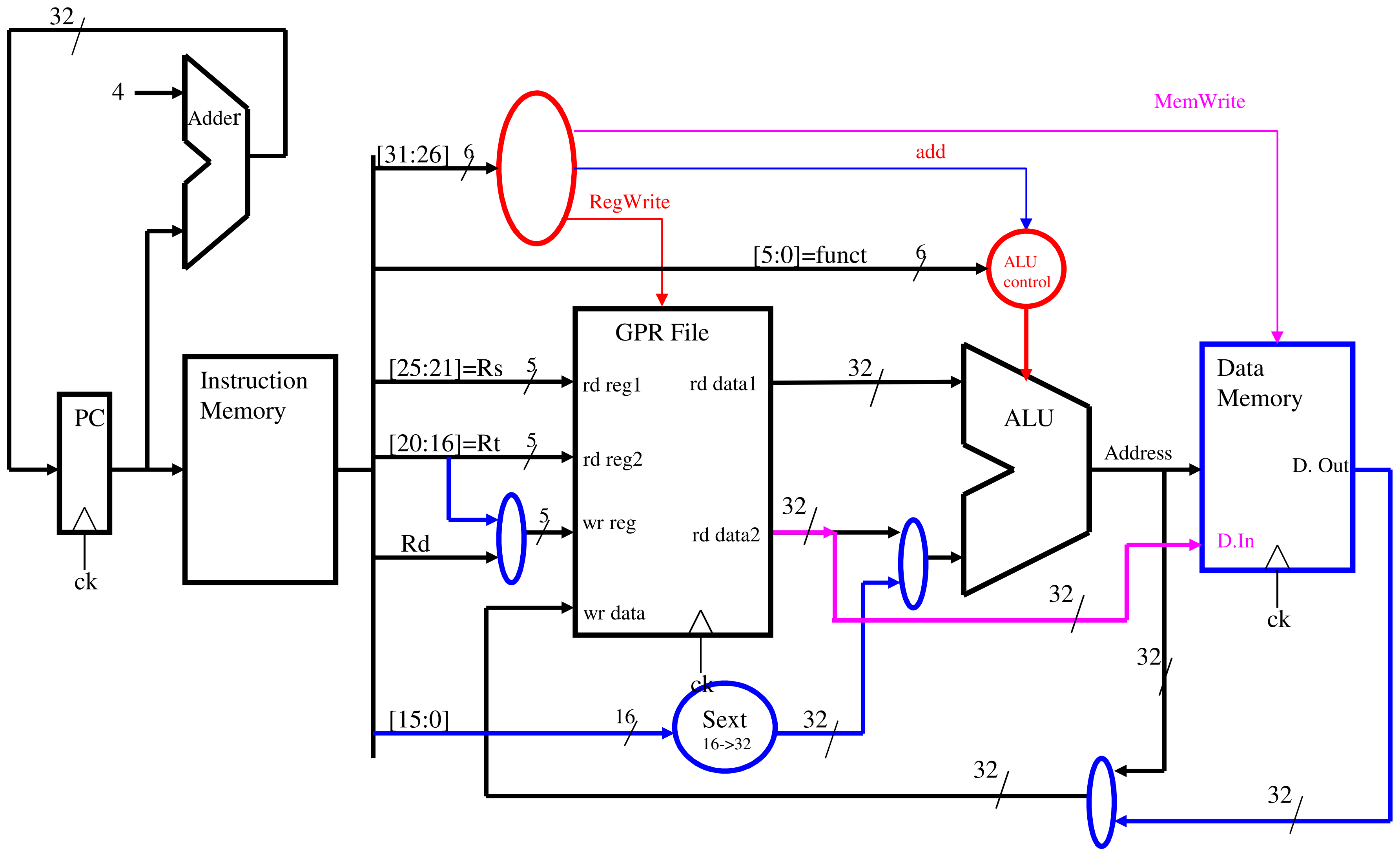
A CPU capable of lw & sw instructions only



A CPU capable of lw & sw instructions only



A CPU capable of R-type & lw/sw instructions



The same approach for VHDL pipelined MIPS

- **We start with the FETCH unit**
  - Reading the instruction
  - Ready for jump & branch instructions
- **We build the GPR File & ALU as components – for future phases**
  - Describing The GPR File – that has all 32 General Purpose Regs
  - Showing how Rs & Rt data is read from the GPR File
- **Combine the Fetch Unit, the GPR File and ALU into a "Rtype" CPU**
  - This CPU has 4 phases: FETCH, DECODE, EXECUTE, WRITE BACK
  - It supports Rtype instructions, but also branch & jumps which in pipelined MIPS are performed in the DECODE phase
  - We also support ADDI instruction – to allow testing of the CPU
- **Then, add more instructions in steps** (lw & sw, then lui, ori, jal, jr)

We need also to introduce FPGA & VHDL

- **Actually we need to start with FPGA design cycle and VHDL language**
  - Implementing a design involves:
    - Writing VHDL code – description of the H/W in VHDL
    - Simulating the design – check ALL(?) signals
    - Compiling into bit file – only after successful simulation
    - Loading into the circuit & testing the implementation
- **In the first 3-4 classes we teach all of the above**
  - Lectures 1&2 – Basics of VHDL & FPGA design process
  - Lectures 3&4 – Debugging a pre-prepared simple design which is required in order to learn the tools and the process
- **Then the implementation of the simplified pipelined MIPS begins**

BYOC course agenda – 6 projects

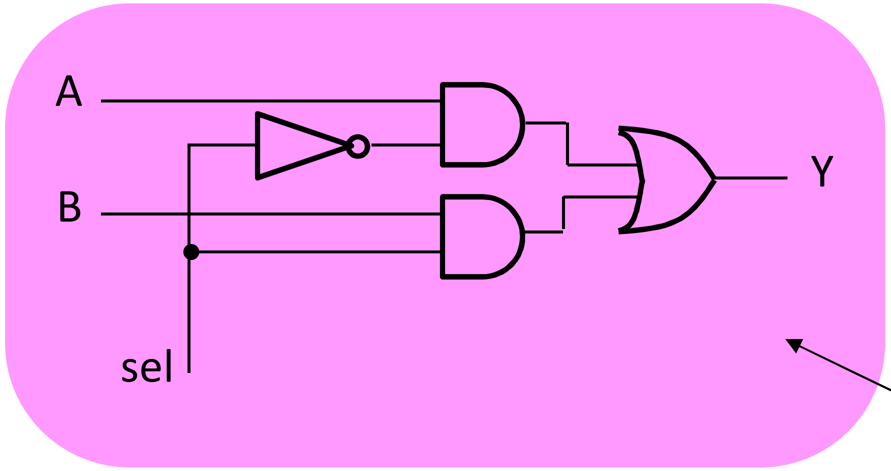
- VHDL & SW tools intro - (L1-L2)
- First design - learning the system & VHDL - **P1** (L2-L4)
- Fetch unit of our MIPS CPU - **P2** (L3-L5)
- The GPR file & the ALU - **P3** (L5-L6)
- R-type only CPU – combining P2 & P3 - **P4** (L6-L8)
- Adding the Data Memory and lw, sw inst. - **P5** (L8-L10)
- Adding jal, jr, lui, ori inst. & forwarding - **P6** (L10-L13) and running a simple Pong game – if successful design

A short intro to VHDL

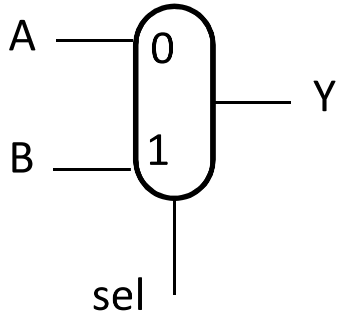
- VHDL is a HW description language
- In this language we write "equations" describing combinational or sequential "entities" or components & their connections
- We then convert it to a chip with a "silicon compiler" by implementing gates, FFs, memories etc., on a silicon layer

  or
- We configure a special chip called FPGA to behave according to the equations we wrote in VHDL
- This will be explained in the next few slides

A VHDL process example: 2→1 mux



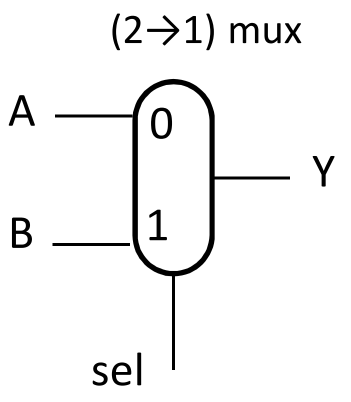
≡

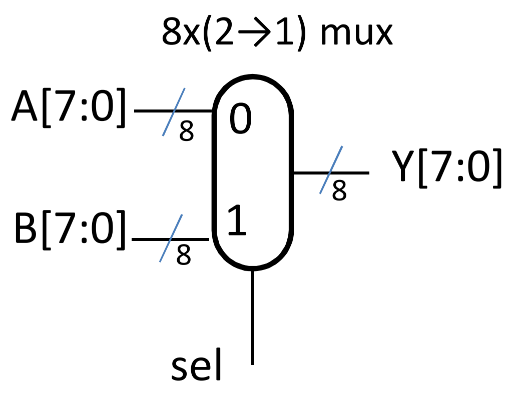


```
mux_2to1:  process  (A, B, sel)
begin
    if sel = ‘0' then
            Y <=  A;
    else
            Y <= B;
    end if;
end process;
```

This IF statement stands for that

2→1 mux vs. Nx(2→1)





In both cases we have the same **process** code

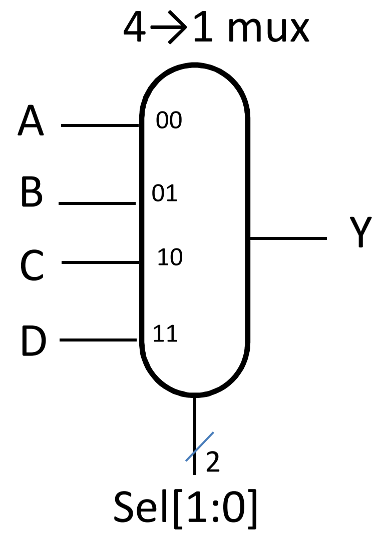
```
process  (A, B, sel)
begin
    if sel = ‘0' then
          Y <=  A;
    else
          Y <= B;
    end if;
end process;
```

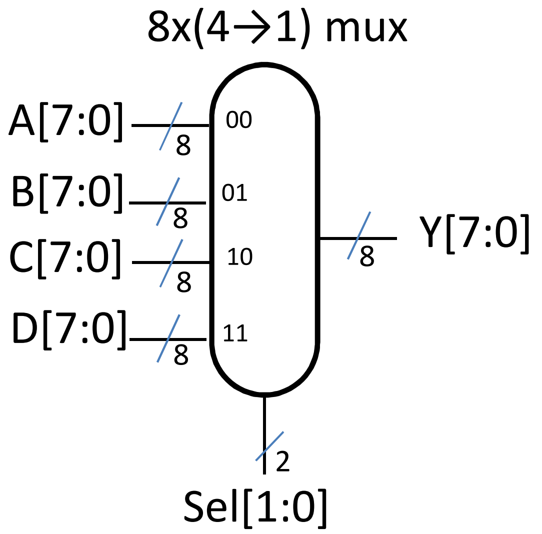
In the single wire case we define:
signal A : STD_LOGIC;

In the multi-wire case we define:
signal A : STD_LOGIC_VECTOR (7 downto 0);

4→1 mux

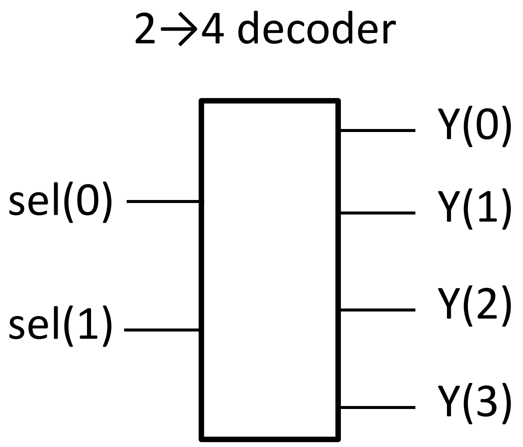
```
process  (A, B, C, D sel)
begin
     if sel = b”00” then
             Y <=  A;
     elsif sel = b”01” then
             Y <= B;
    elsif sel = b”10” then
             Y <= C;
    else
             Y <= D;
     end if;
end process;
```



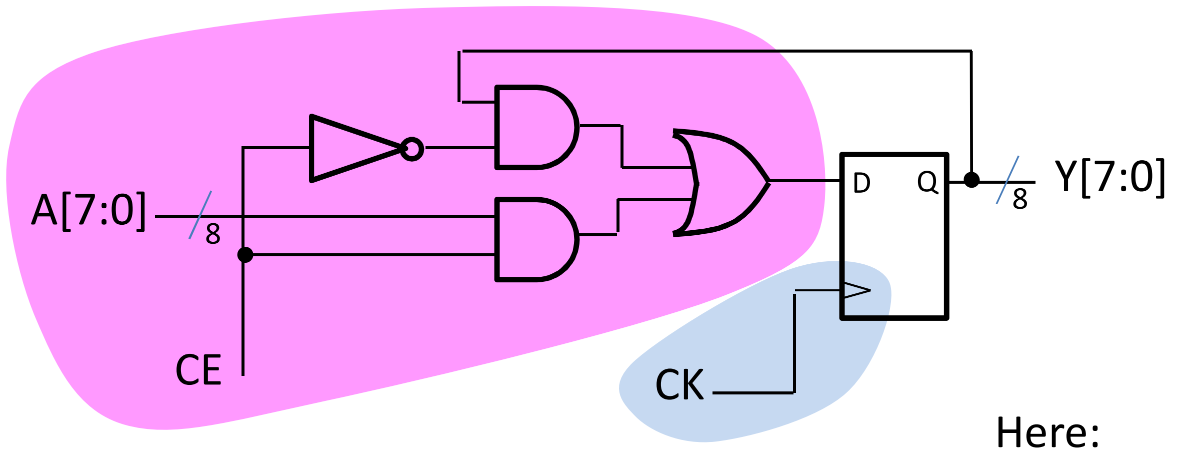


2→4 decoder

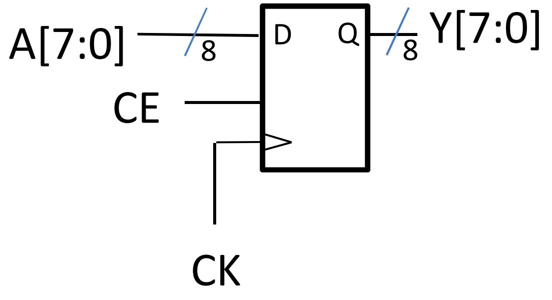
```
signal sel : STD_LOGIV_VECTOR (1 downto 0);
signal Y : STD_LOGIV_VECTOR (3 downto 0);


process  (sel)
begin
     if sel = b”00” then
             Y <=  b“0001”;
     elsif sel = b”01” then
             Y <= b”0010”;
    elsif sel = b”10” then
             Y <= b”0100”;
    else
             Y <= b”1000”;
     end if;
end process;
```



A sequential process example



≡



Here:

```
signal  A : std_logic_vector (7 downto 0);
signal  Y : std_logic_vector (7 downto 0);
signal  CE : std_logic;
signal  CK : std_logic;
```

```
process  (CK)
begin
   if CK'event and CK='1' then
      if  CE = '1' then
         Y <=  A;
      end if;
   end if;
end process;
```

If we have a rise in the CK then:

If CE='1' we sample data,
else we keep the data unchanged.

What do we do with VHDL?

- We describe our design in VHDL

  This is similar to writing a program

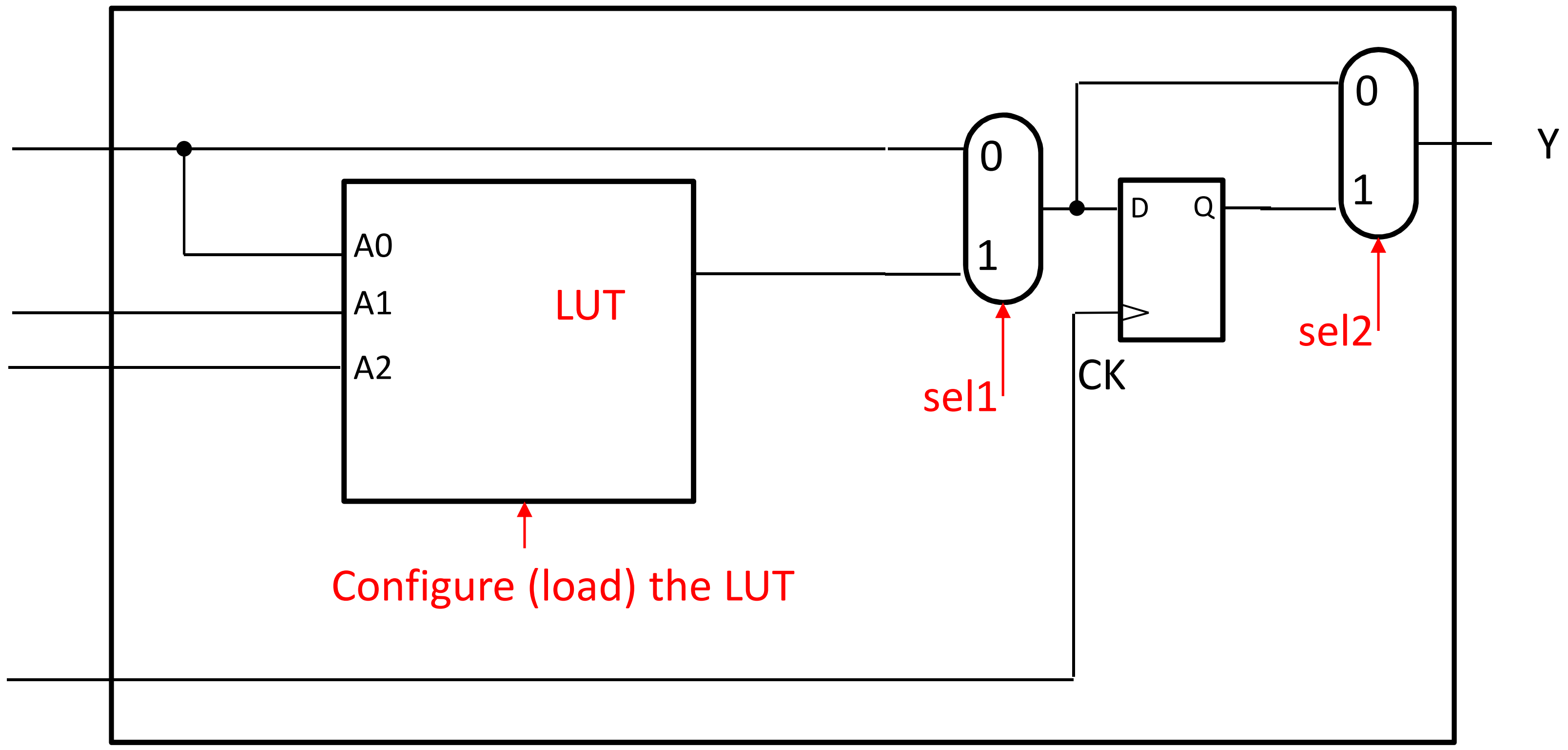
- While programs are compiled to machine language and then loaded into a computer and run, we here must implement our design on some kind of HW

- We “compile” our design and “load” it into a special HW device called FPGA

- FPGA device can implement any function we want
- How??

FPGA concept

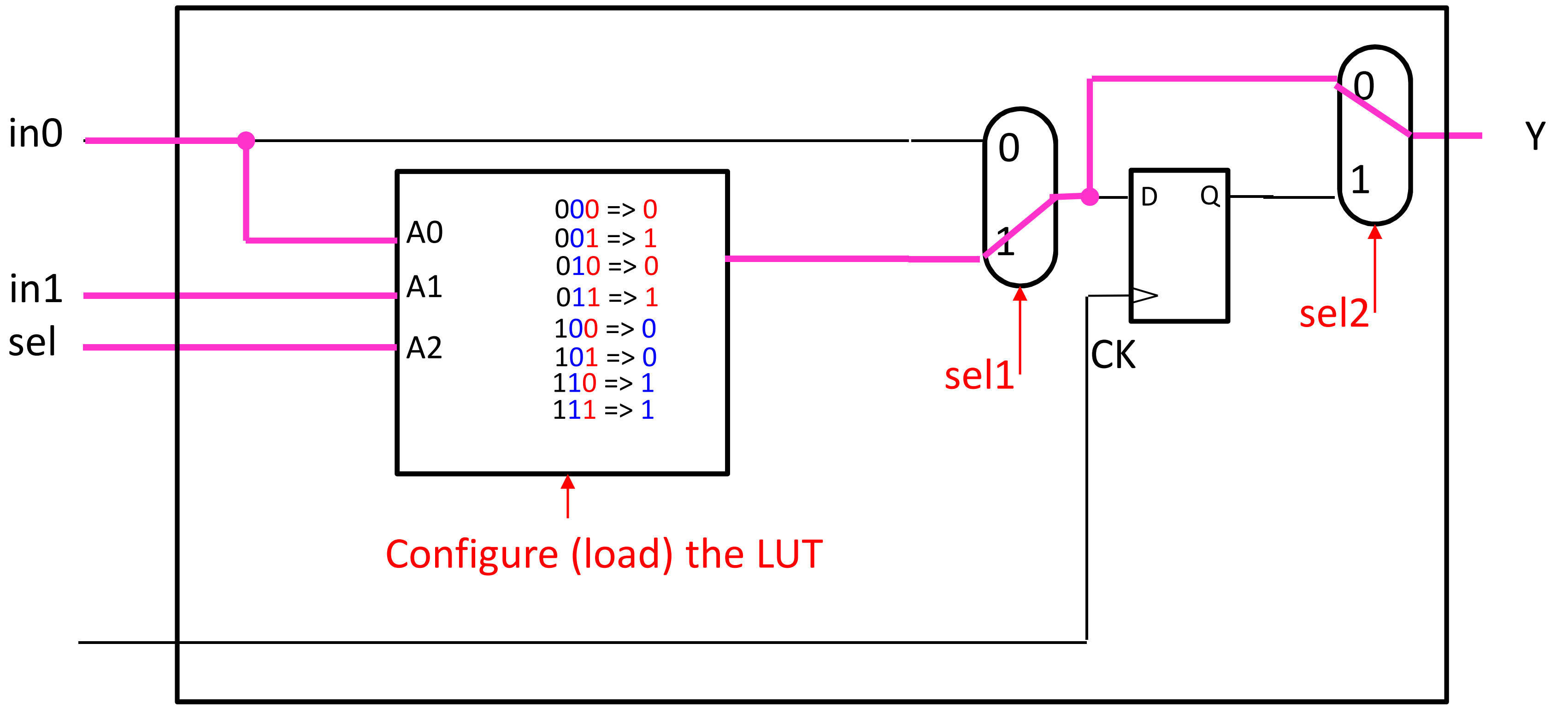
The Field Programmable Gate Array has an array of Logic Blocks



Let’s demonstrate implementation of a simple mux.
During “configuration phase”, we fill up the LUT and choose values for sel1 & sel2

FPGA concept

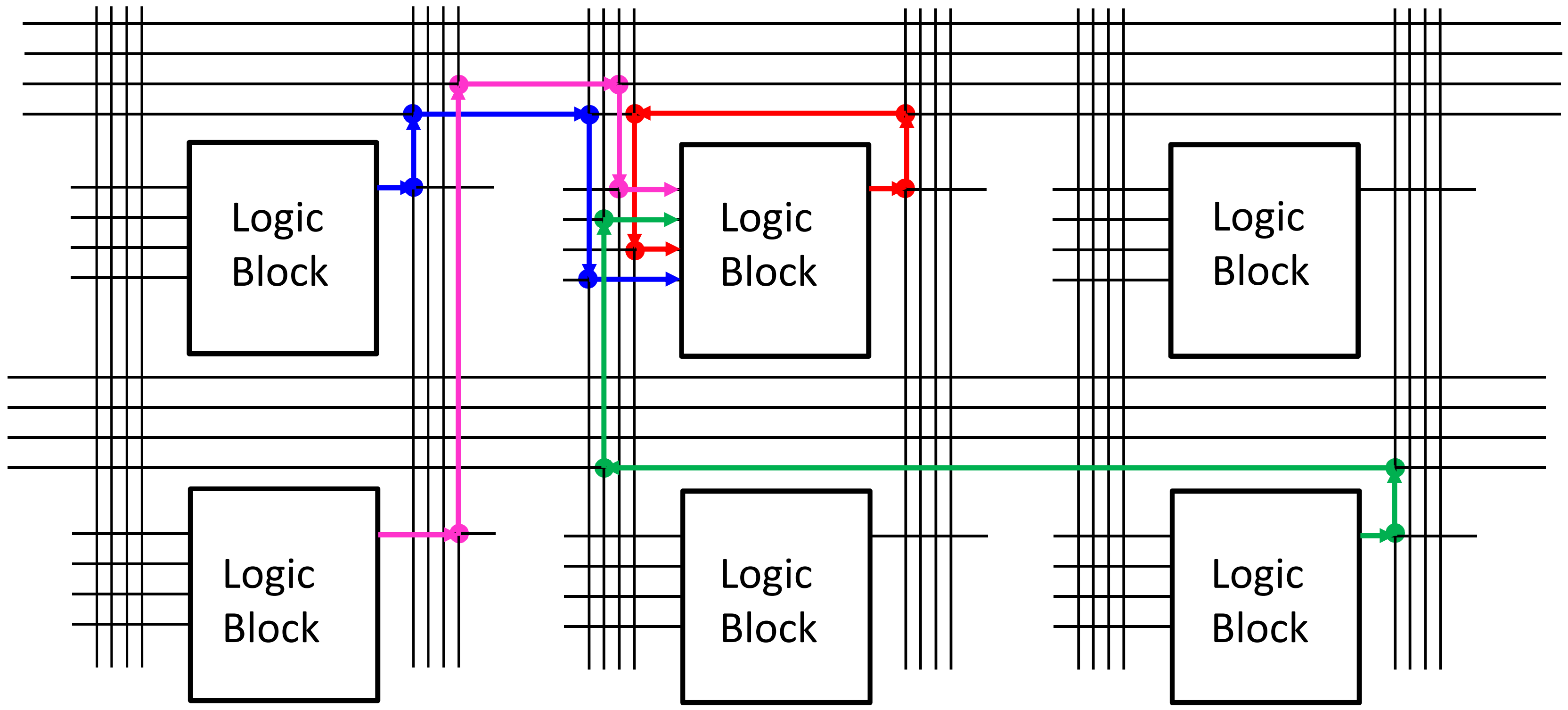
The Field Programmable Gate Array has an array of Logic Blocks



Let's demonstrate implementation of a simple mux.
During "configuration phase", we fill up the LUT and choose values for sel1 & sel2
We fill up the LUT with the truth-table representing the required function!
Here it is the mux truth table. When sel=0, we have Y=in0, when sel=1 we have Y=in1

FPGA concept

There is also a matrix of internal lines allowing connections to/from the Logic Blocks

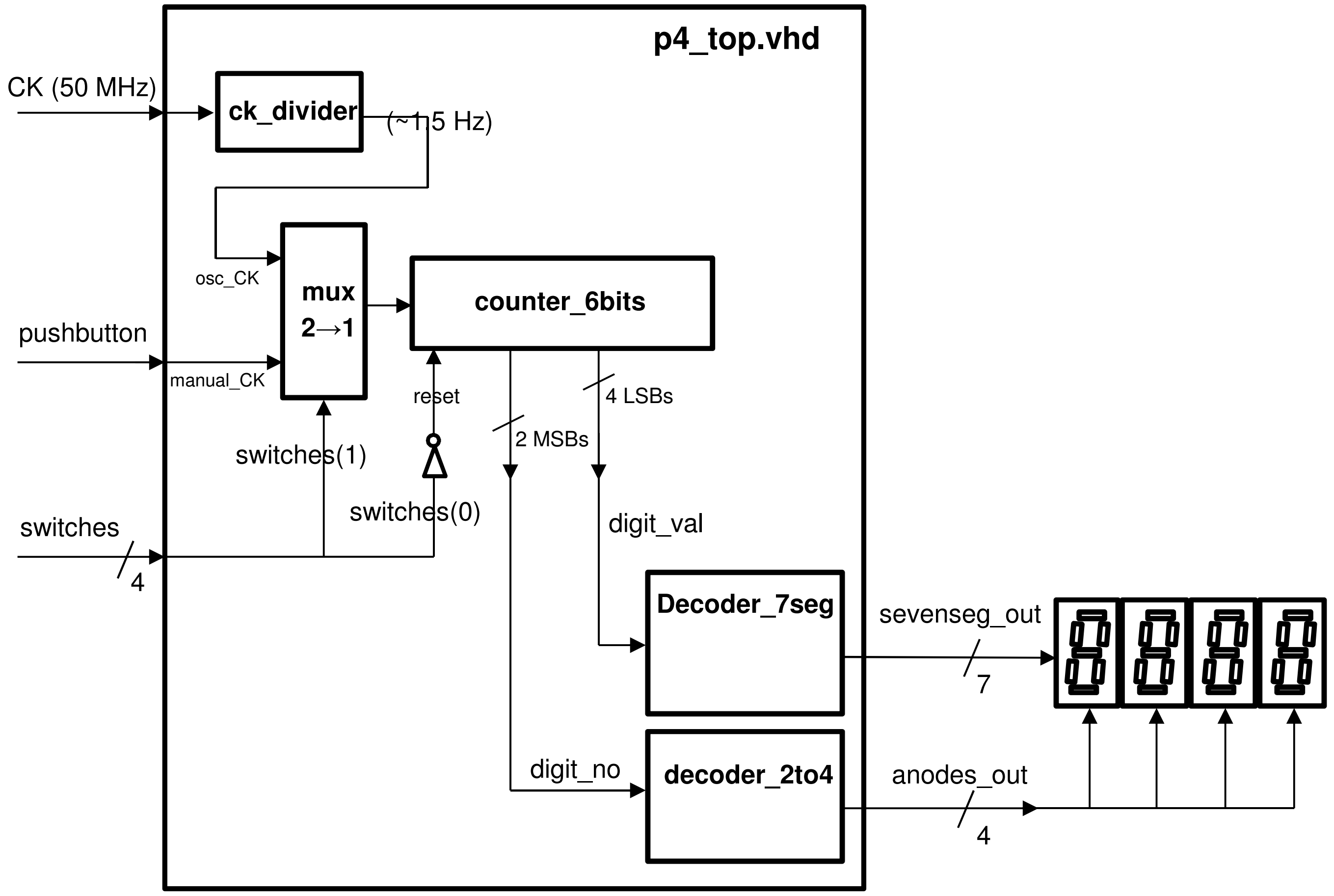


We can “connect” between Logic Blocks by connecting specific intersections

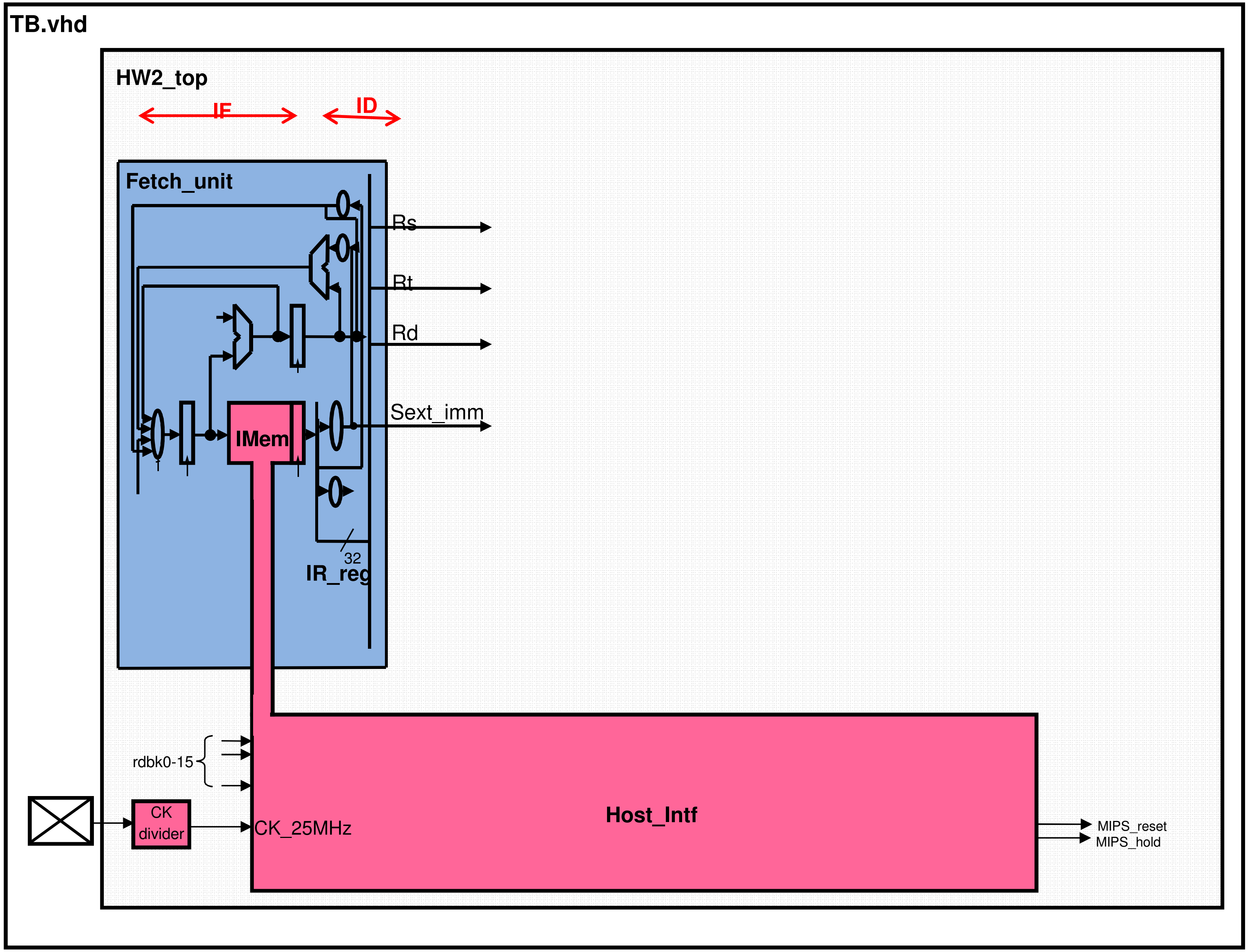
The connections are determined during configuration - 1 bit determines a connection

HW1 – The 1st design

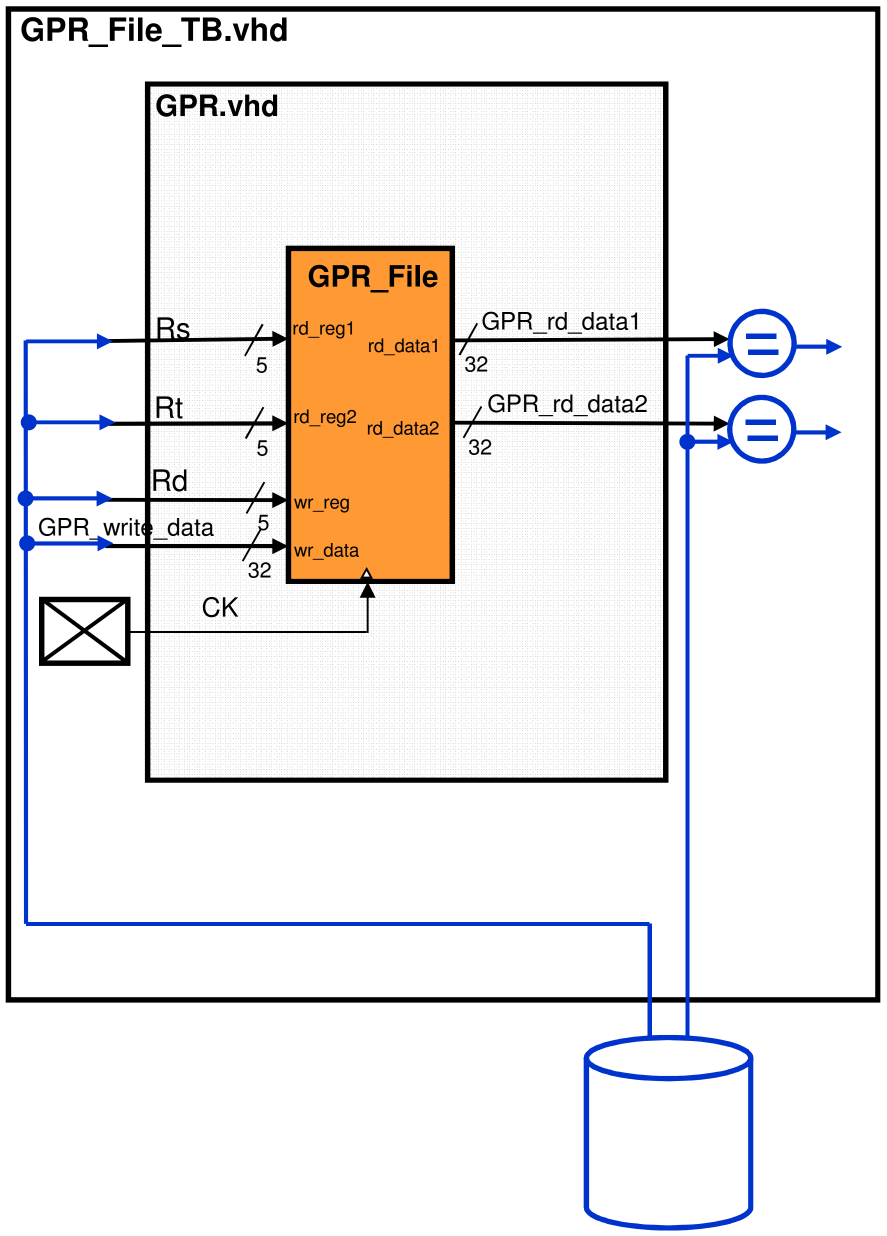
The design is a free-running 6 bit counter that is displayed on 4 LEDs
It has many errors and the student needs to simulate it, create a bit file & test it on the Nexys2 board – i.e., the complete FPGA design cycle

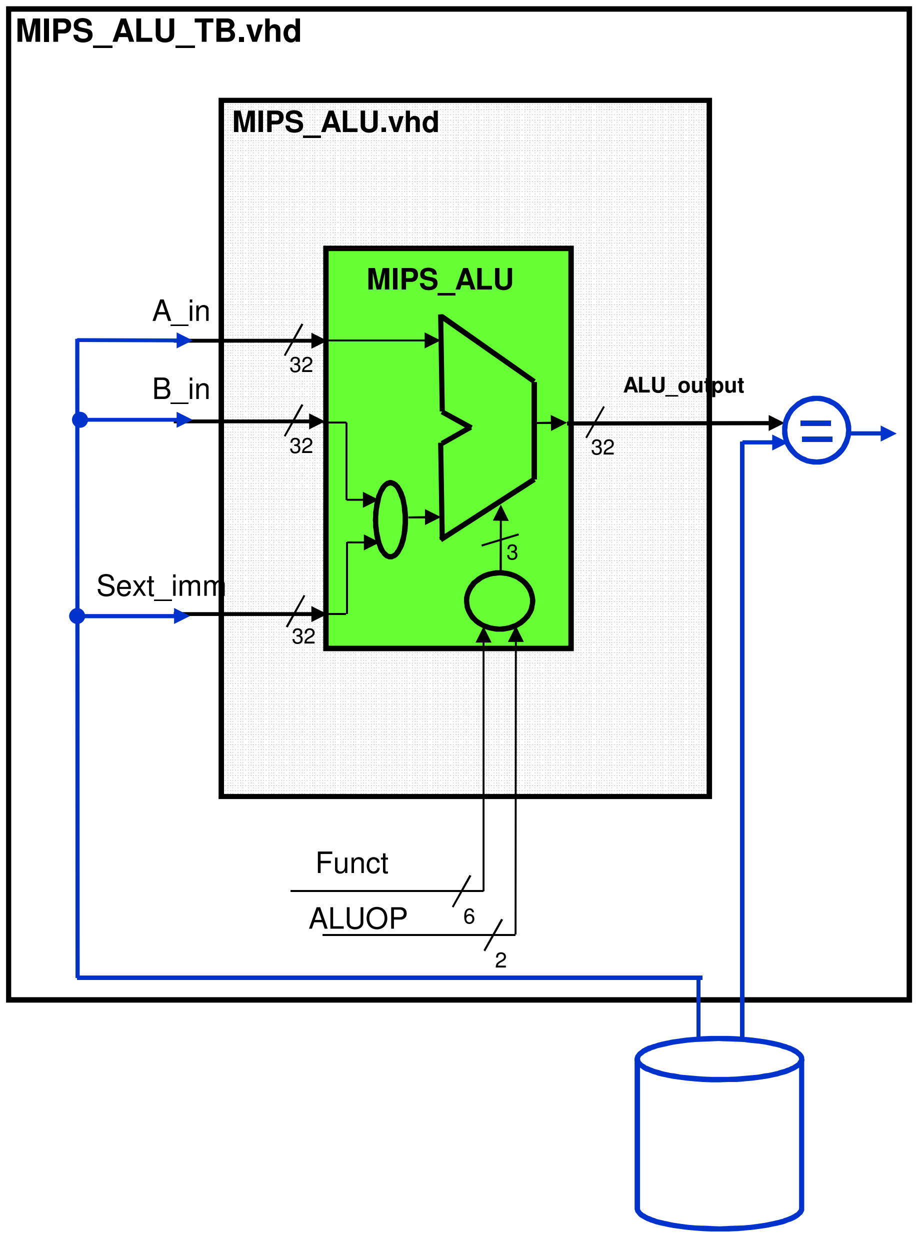


HW2 – The Fetch Unit

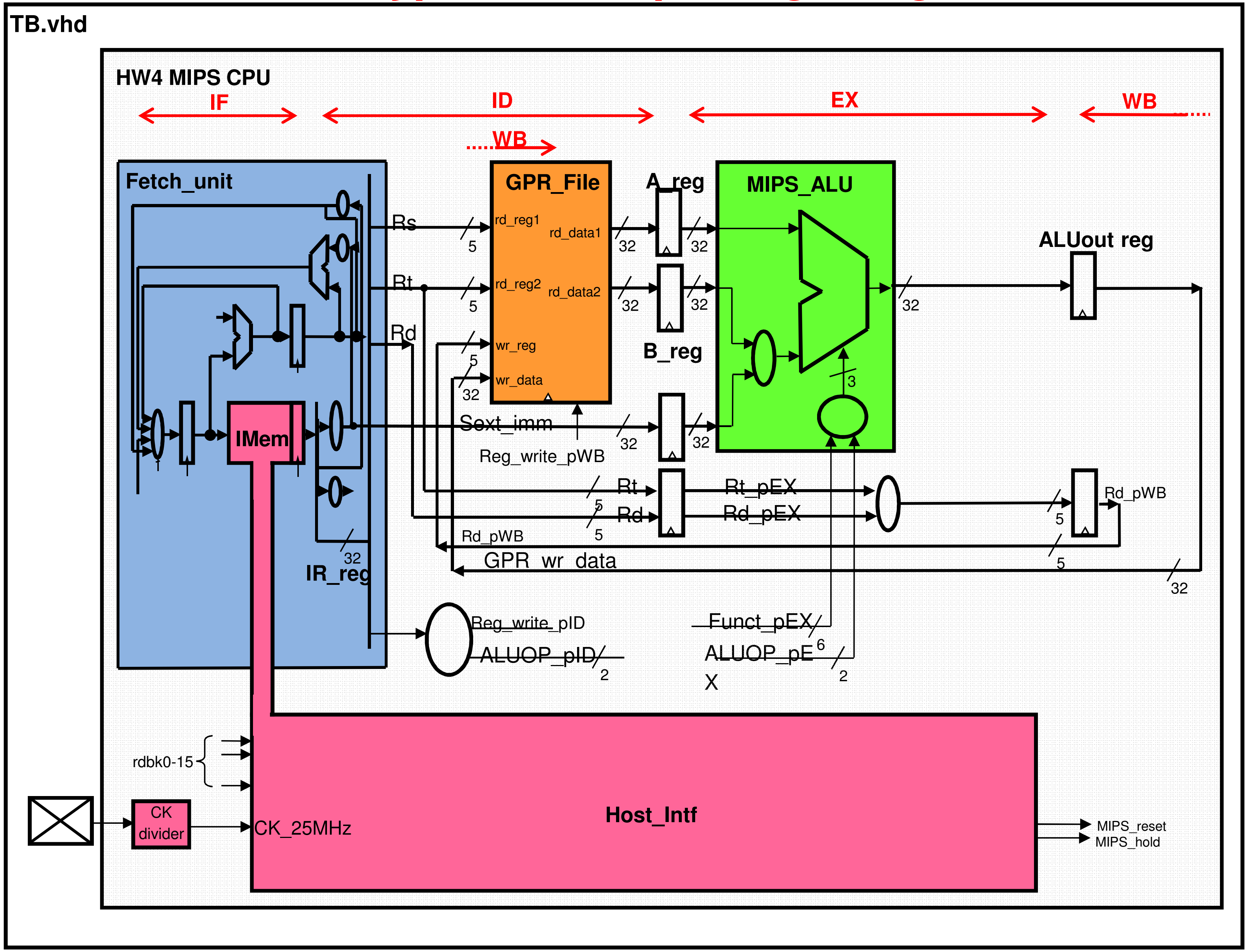


HW3 – GPR File & MIPS ALU – simulation only

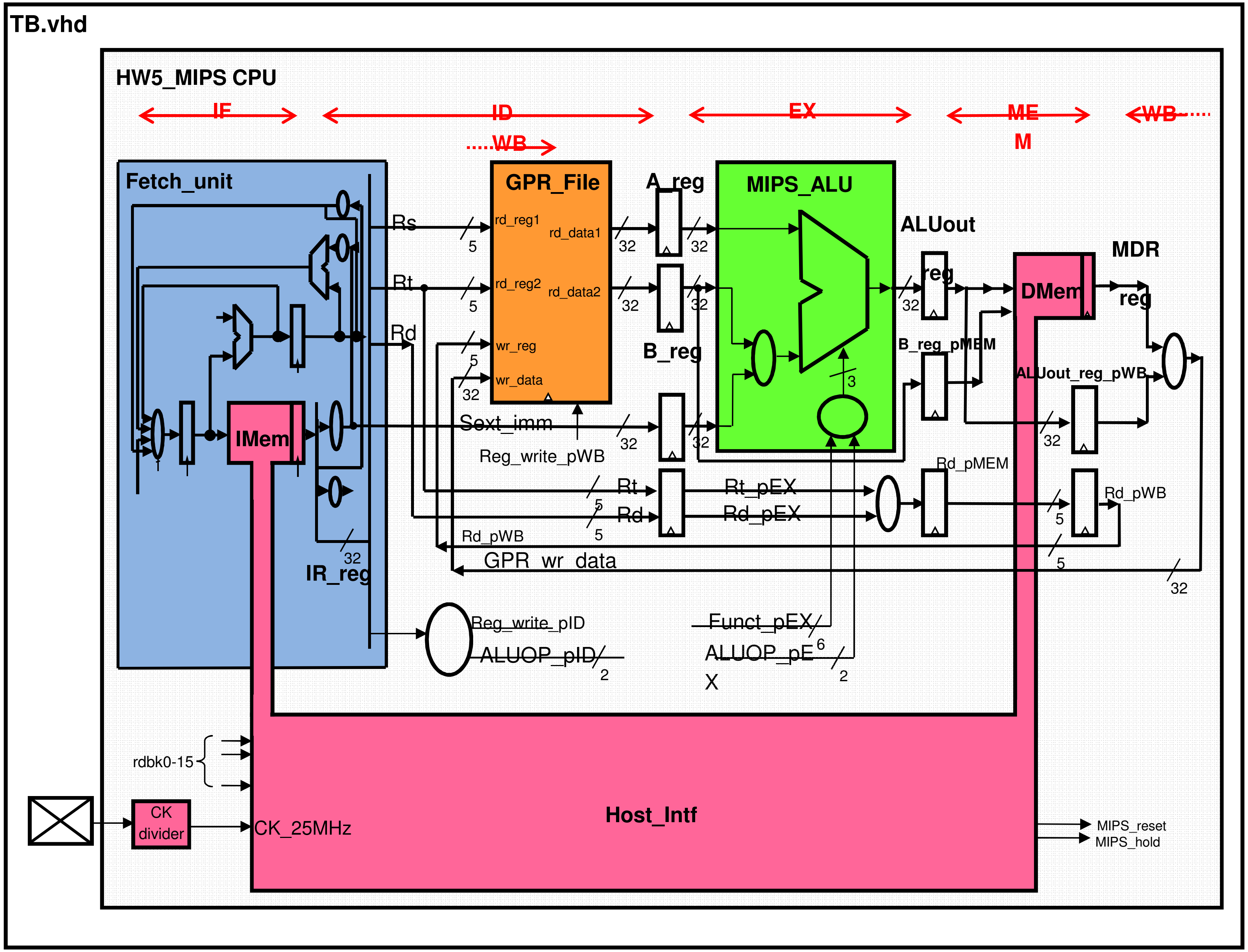




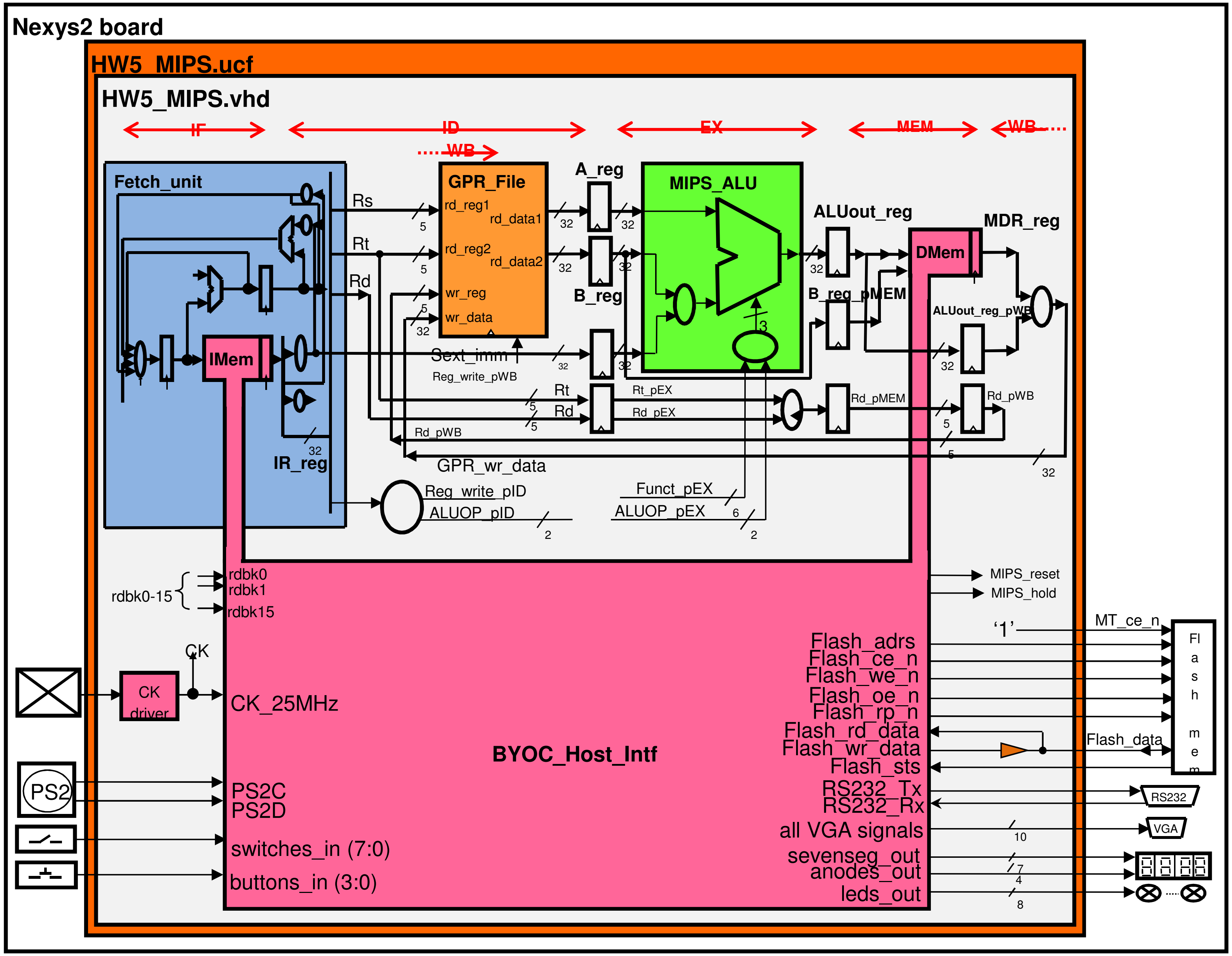
HW4 – "Rtype" CPU – putting it together (Rtype & addi, j, branch, no jr)



HW5 (adding Data Memory) – The simulation version



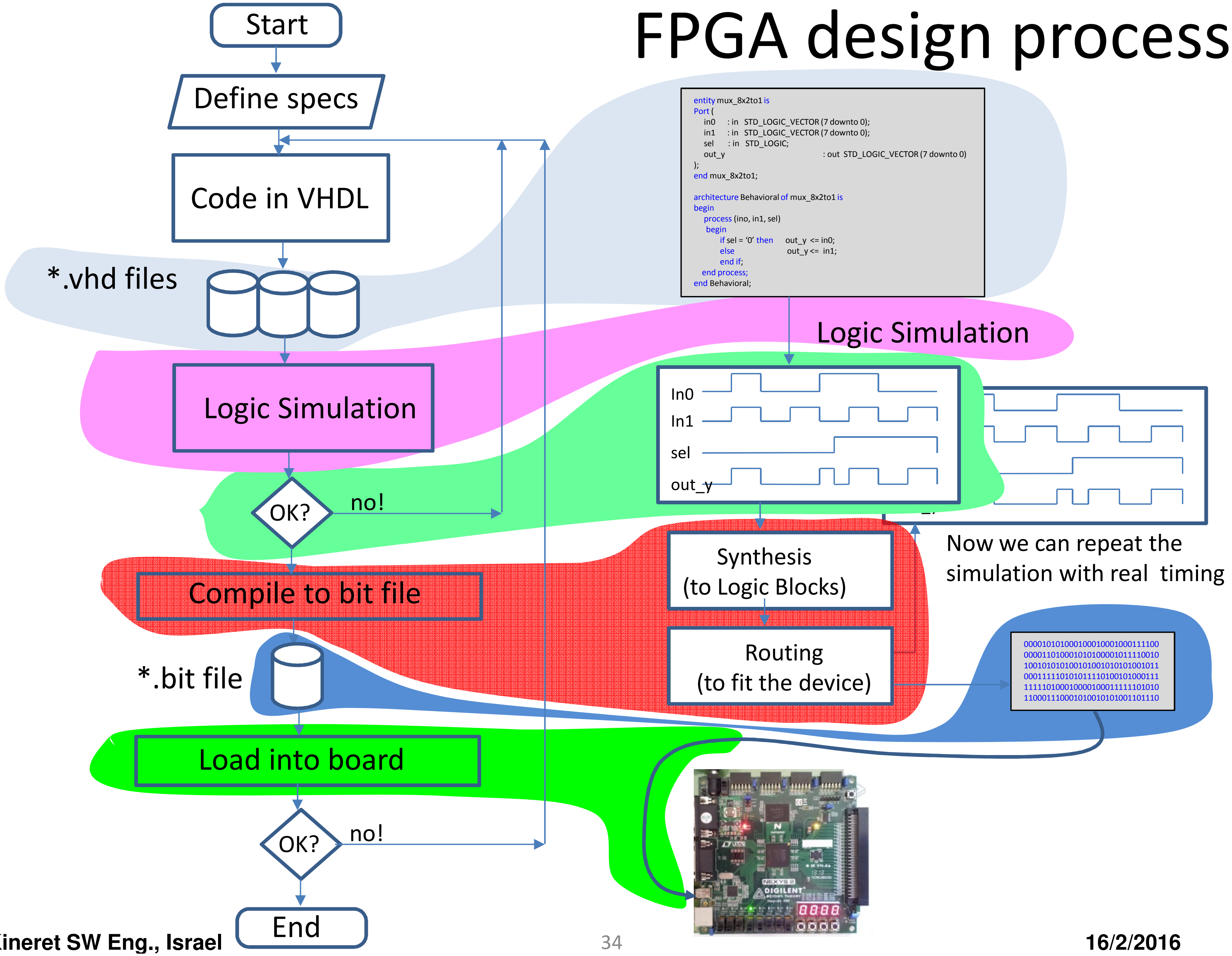
HW5 -The implementation version



Now is a good time to discuss the process

- **First we define the design we want**
- **Then we code it in VHDL**
  - It can be done in text file or in graphical mode
  - We use text files
- **Next step is simulation**
  - It means testing the design by SW simulation- same as unit test
  - MUST be done – otherwise success chances are slim to none
- **Only then we compile the design into a BIT file**
- **Now we can “load” it into the FPGA on the board and run it**
- **Debugging the circuit requires a way to check signal values**

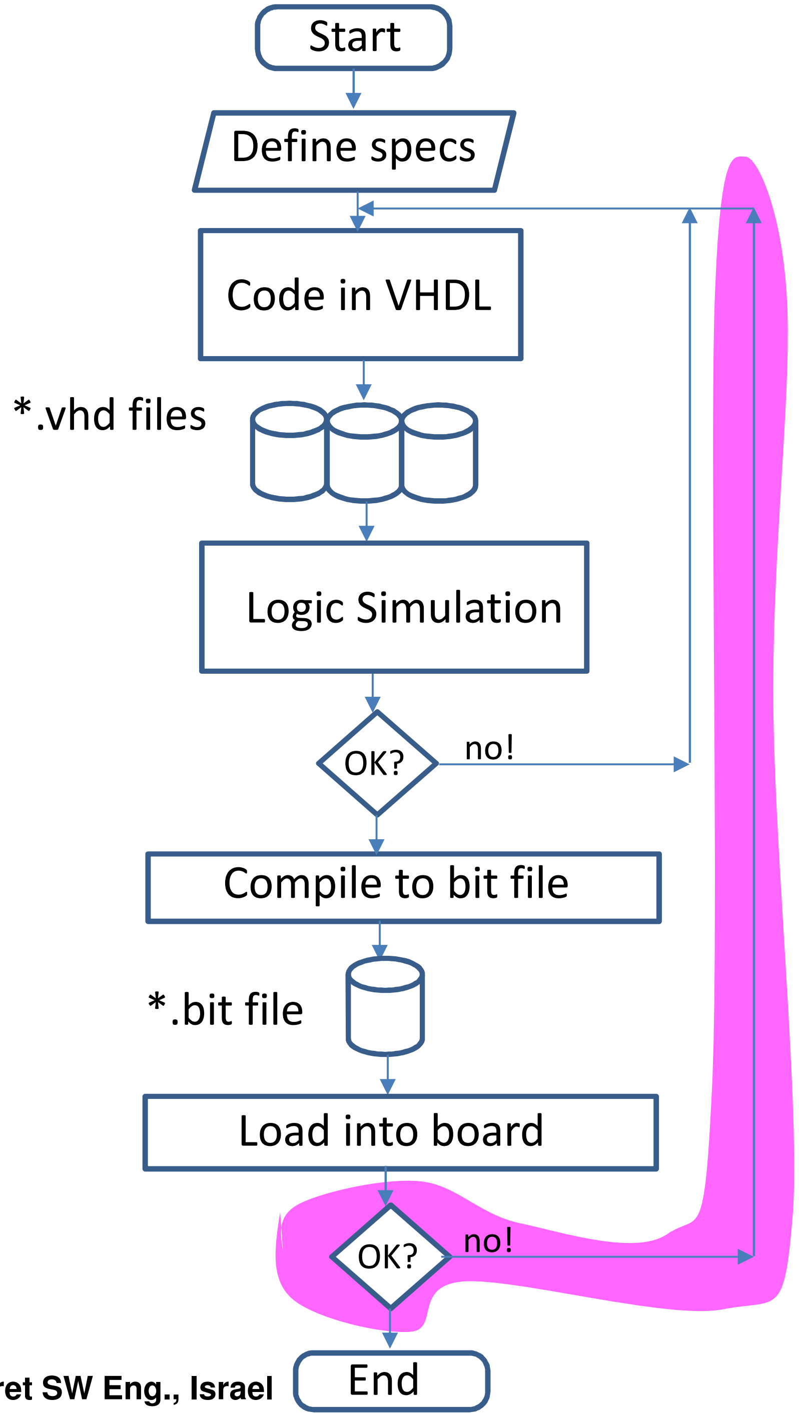
FPGA design process



```
entity mux_8x2to1 is
Port (
   in0     : in   STD_LOGIC_VECTOR (7 downto 0);
   in1     : in   STD_LOGIC_VECTOR (7 downto 0);
   sel     : in   STD_LOGIC;
   out_y                         : out  STD_LOGIC_VECTOR (7 downto 0)
);
end mux_8x2to1;

architecture Behavioral of mux_8x2to1 is
begin
   process (ino, in1, sel)
    begin
       if sel = ‘0’ then     out_y  <= in0;
       else                  out_y <=  in1;
       end if;
   end process;
end Behavioral;
```

```
00001010100010001000100111100
00001101000101010000101111001
0
10010101010010100101010100101
1
00011111010101111010010100011
1
11111010001000010001111110101
0
11000111000101001010100110111
0
```

FPGA design process



How do we know what to fix if it does not work??

Debugging in the board requires tools:

A Logic Analyzer is a measurement device allowing to see signals in the design

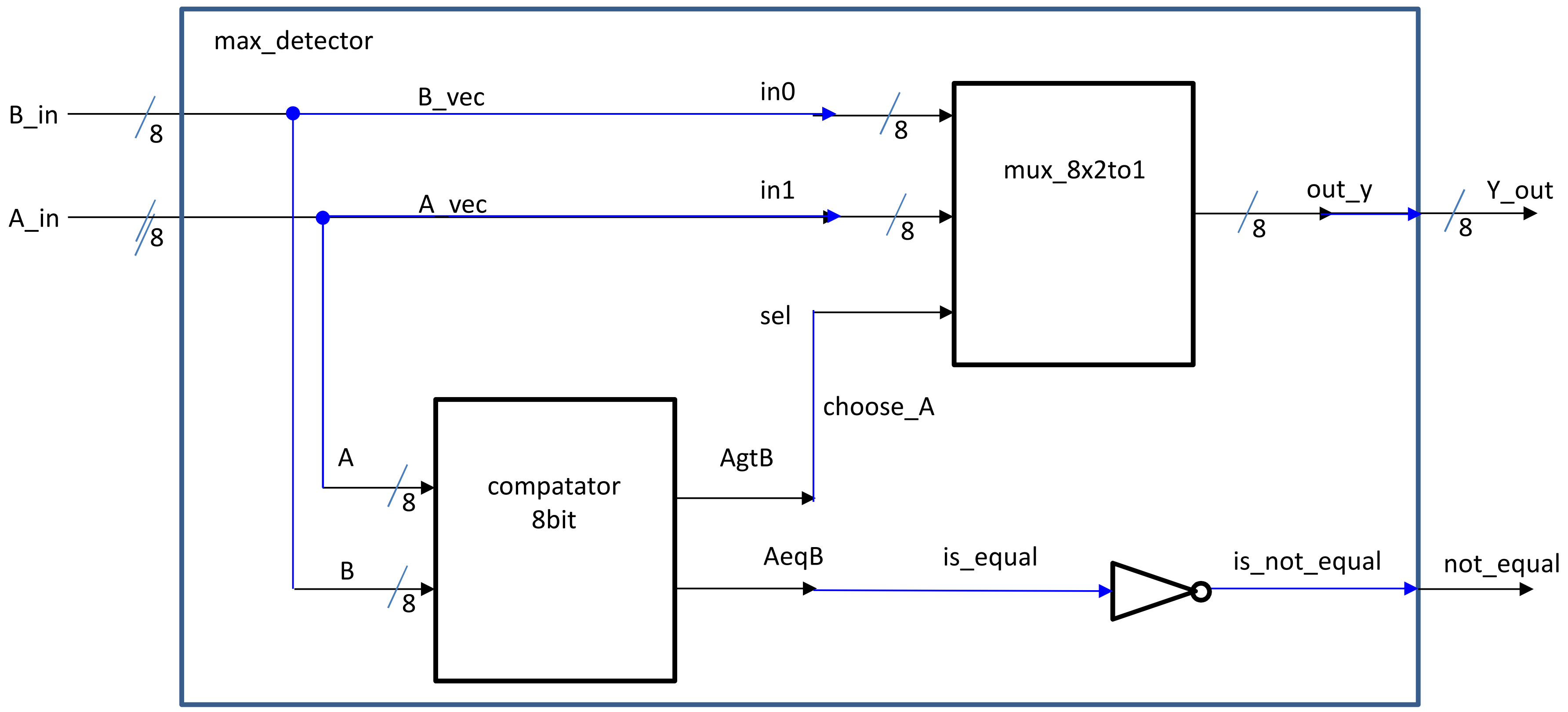
We could route the required signals to external pins and hook them to a Logic Analyzer

Instead we route the required signals to a RS232 port that can be read by a PC with a **BYOCInterface** SW

BYOC course infrastructure

- **Support while coding**
  - Start with a detailed explanation of the lab project including all signal names
  - Give a pre-prepared vhd file with i/o pins definitions
  - Add signal definitions – all of them or part of them
  - Give also the components used and components connections (port maps)
  - Add notes describing what should be written & where
  - Give some of the equations – as an example

An example of a simple design



We need to define the I/Os of a new entity, max detector
We need to specify the two components we use – mux_8x2to1 and compataror_8bit
We need to "connect" the blue wires directly or via signals
If there is other logic inside (additional processes), we also need to specify them

The vhd file of this example

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity max_detector  is
Port (
    A_in        : in   STD_LOGIC_VECTOR (7 downto 0);
    B_in         : in   STD_LOGIC_VECTOR (7 downto 0);
    Y_out       :out STD_LOGIC_VECTOR (7 downto 0);
    not_equal  :out : STD_LOGIC
);
end max_detector ;

architecture Behavioral of max_detector is

component   mux_8x2to1  is
Port (
    in0      : in   STD_LOGIC_VECTOR (7 downto 0);
    in1      : in   STD_LOGIC_VECTOR (7 downto 0);
    sel       : in   STD_LOGIC;
    out_y          : out  STD_LOGIC_VECTOR (7 downto 0)
);
end component ;

component  comarator_8bit  is
Port (
    A           : in   STD_LOGIC_VECTOR (7 downto 0);
    B           : in   STD_LOGIC_VECTOR (7 downto 0);
    AgtB      :out   STD_LOGIC;
    AeqB       : out  STD_LOGIC
);
end component ;
```

Here we define the entity max_detector that is, define the I/Os – the “pins”

In the architecture part we specify the inside of the max_detector entity

We start with defining components to be used inside the entity

The vhd file of this example (cont.)

```
-- signals
signal  is_equal : STD_LOGIC;
signal  is_not_equal : STD_LOGIC;
signal  choose_A: STD_LOGIC;
signal  A_vec: STD_LOGIC_VECTOR (7 downto 0);
signal  B_vec : STD_LOGIC_VECTOR (7 downto 0);

begin

-- Here we have the internal logic.
-- We may add instructions of what to write here
is_not_equal <= not ( is_equal );

-- connectinting the i/o pins to signals
A_vec   <=  A_in;
B_vec   <=  B_in;
not_equal <= is_not_equal ;

Comp1 :  comparator_8bit
port map (
A        =>    A_vec,
B        =>    B_vec,
AgtB   =>    choose_A,
AeqB   =>    is_equal  );

Comp2 : mux_8x2to1
port map (
in_0      =>    B_vec,  -- connect to an internal signal
in_1      =>    A_vec,
sel       =>    choose_A,
out_y    =>    Y_out ); -- direct to pin

end Behavioral;
```

we also define the inner signals

Processes inside this entity come here.

This is the design logic & usually it is the main part of the code

Here are all the “connections”

The entity’s i/o pins to signals
& components to signals

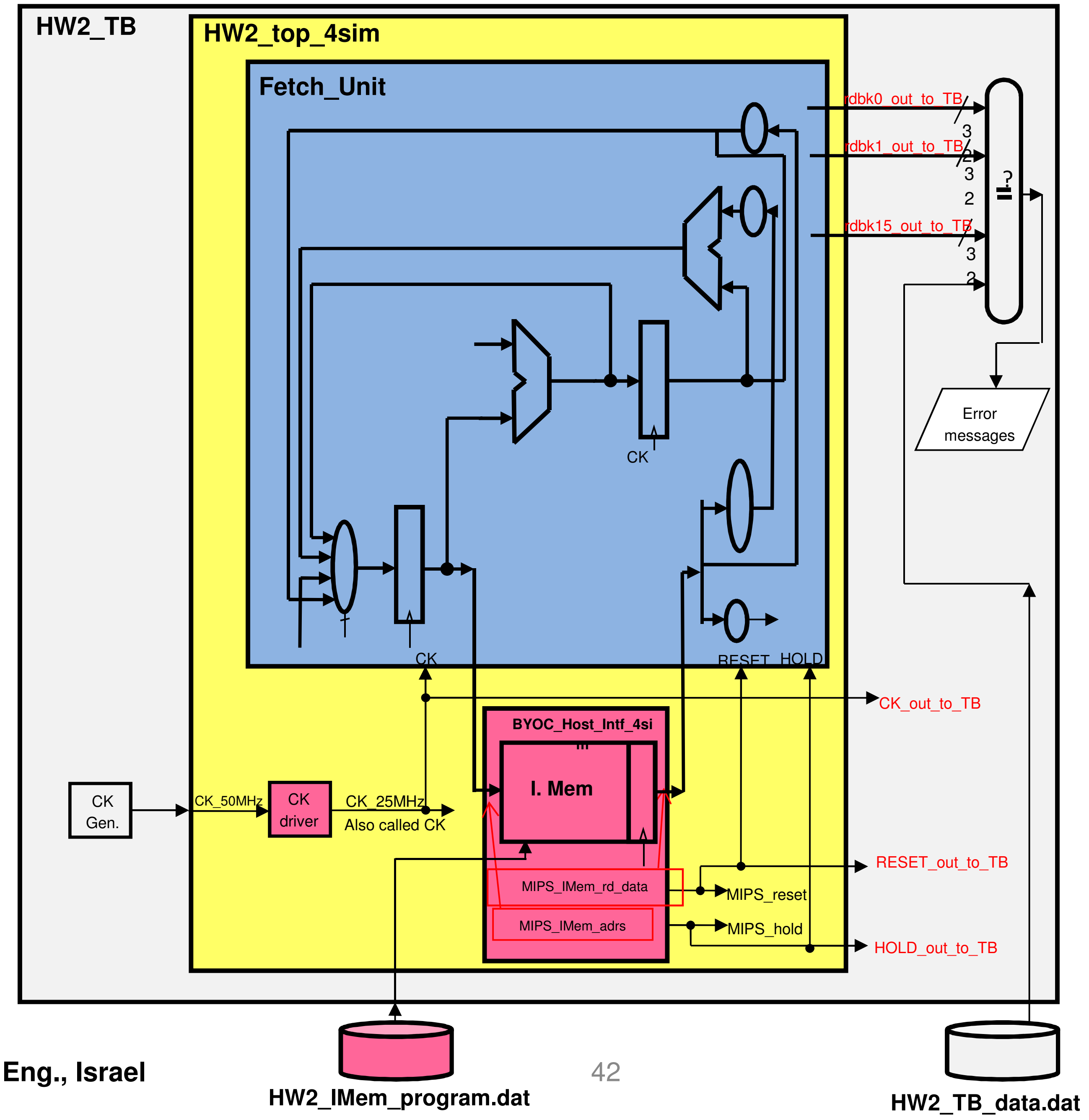
BYOC course infrastructure

- **Support in simulation**
    - Start with a detailed explanation of the lab project including all signal names, all rdbk signals and their connection to the TestBench
    - Prepare a TestBench that reads the rdbk signals, compares them to ones “recorded” from a correct design and reports errors to the simulator console
    - Prepare the appropriate MIPS assembly program that test the functionality of the design. You may deliberately omit part of the functionality – so that malfunctioning will be found later in the course
    - Give the students the program and the compare data for the parts of the design for which you want to ease the debugging
    - For other parts ask the students to look at the signal waveforms in the simulator and explain what they see
    - Have a complete TB version for teacher that checks everything

BYOC course infrastructure

- **Support in simulation**
  - The students get two pre-prepared components – a clock driver and the BYOC_Host_Intf
  - The clock driver is a simple divide by 2 circuit that in the implementation phase requires a special BUFG component which the students are not familiar with
  - The BYOC_Host_Intf has the Instruction Memory (IMem) and the Data Memory (Dmem) and some infrastructure capable of “loading” program into the IMem at the beginning of simulation
  - Actually we have two versions of this component. The BYOC_Host_Intf_4sim has the same interface (i/o pins) as the BYOC_Host_Intf component used for implementation. This makes it very easy to convert the simulation version of the design to an implementation version.

The Fetch Unit Simulation project



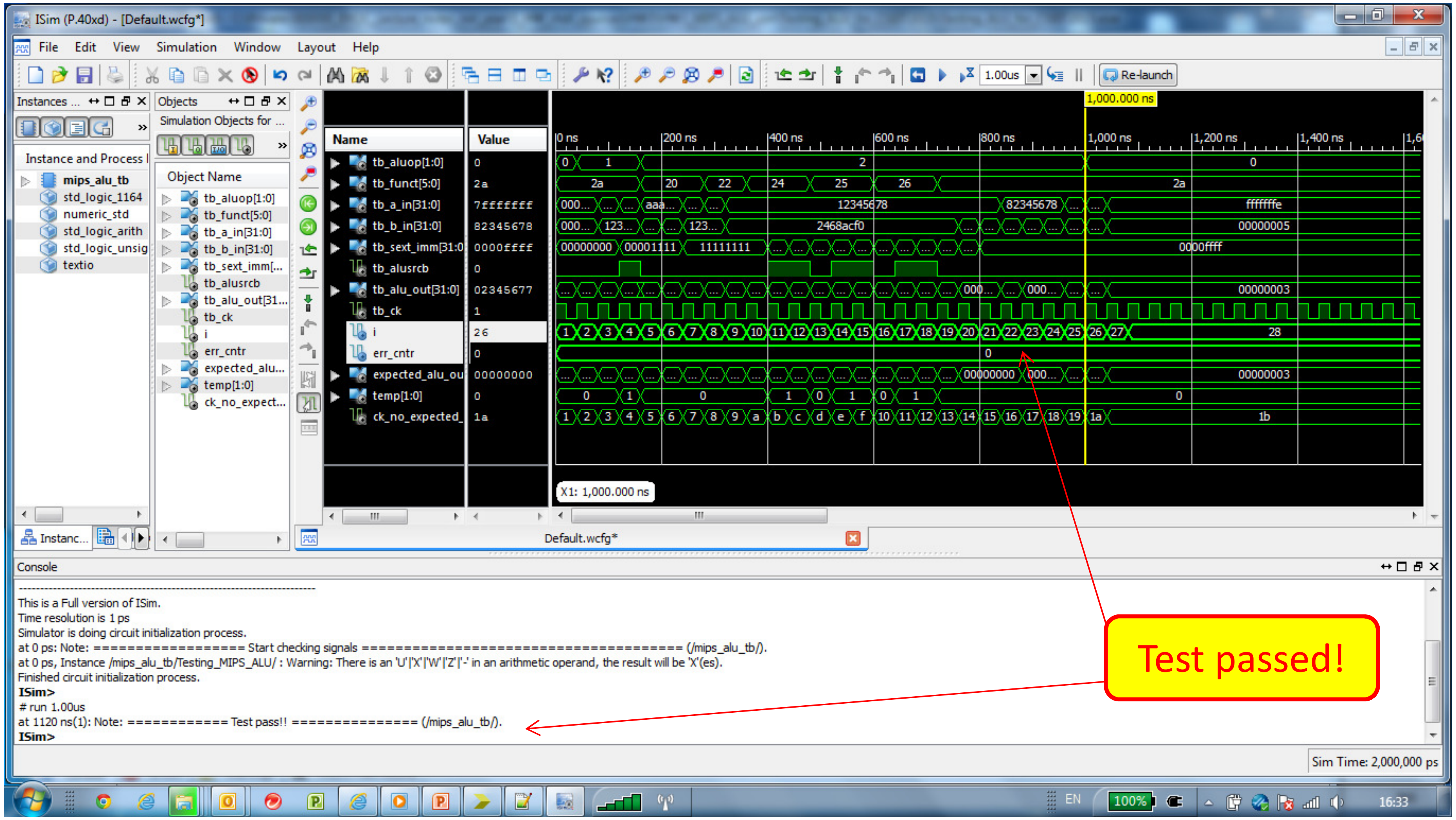
Kineret SW Eng., Israel

16/2/2016

Here is the ALU control correct code

```
-- ALU
    process(ALUOP, Funct, ORI)
    begin
        if ORI = '1' then
                ALU_cmd <= b"001"; -- FUNCT=OR
        elsif  ALUOP = b"00" then
                ALU_cmd <= b"010"; -- ADD
        elsif ALUOP= b"01" then
                        ALU_cmd <= b"110";--  SUB
        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;
```

Here is the simulation

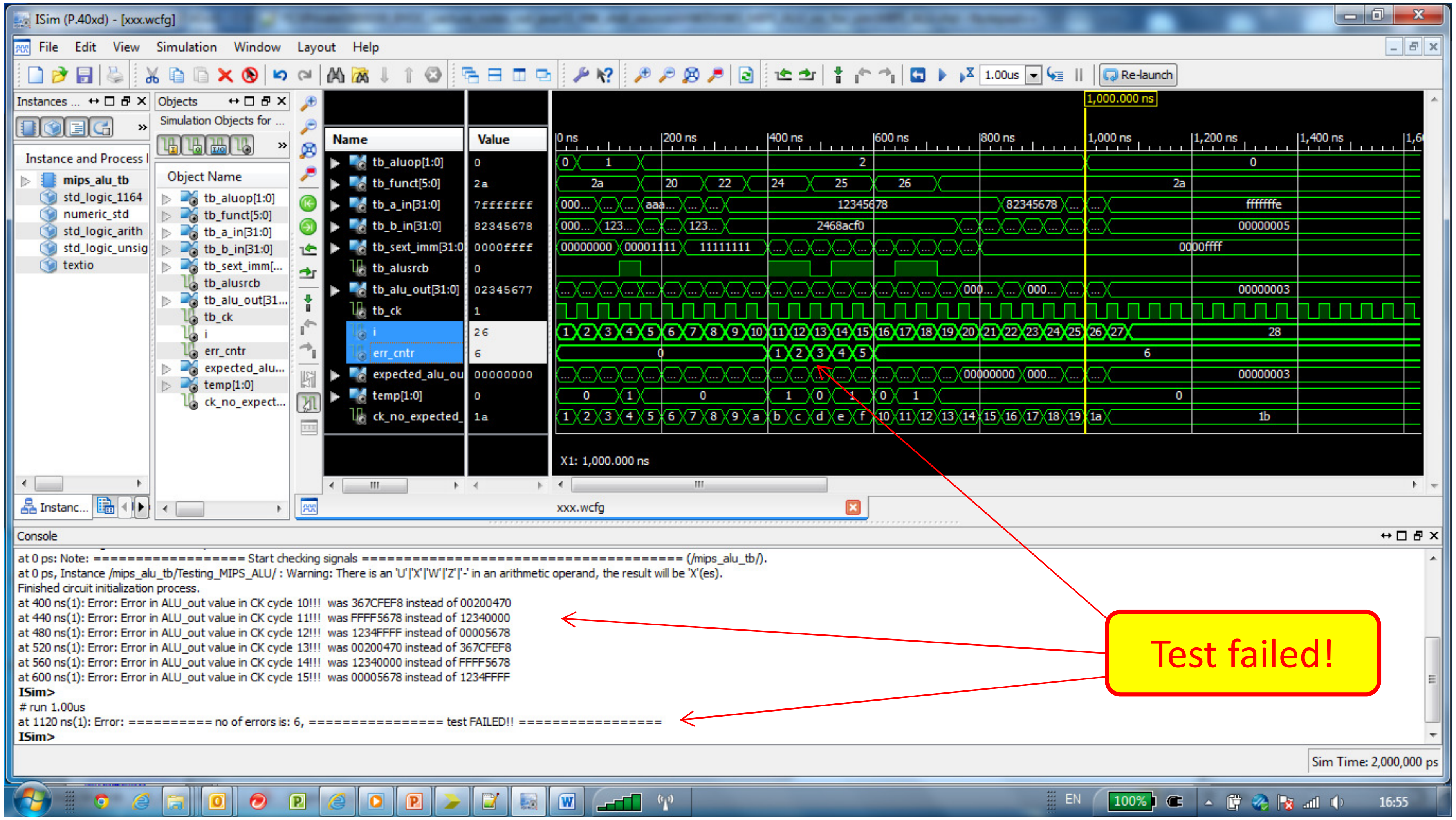


```
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
at 0 ps: Note: ================== Start checking signals ======================================== (/mips_alu_tb/).
at 0 ps, Instance /mips_alu_tb/Testing_MIPS_ALU/ : Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
Finished circuit initialization process.
ISim>
# run 1.00us
at 1120 ns(1): Note: ============ Test pass!! =============== (/mips_alu_tb/).
ISim>
```

Here is the ALU cntrol code with errors

```
-- ALU
      process(ALUOP, Funct, ORI)
      begin
            if ORI = '1' then
                        ALU_cmd <= b"001"; -- FUNCT=OR
            elsif  ALUOP = b"00" then
                        ALU_cmd <= b"010"; -- ADD
            elsif ALUOP= b"01" then
                                    ALU_cmd <= b"110";--  SUB
            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"001"; -- FUNCT=AND
                        elsif Funct = b"100101" then
                                    ALU_cmd <= b"000"; -- 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;
```

Here is the simulation now

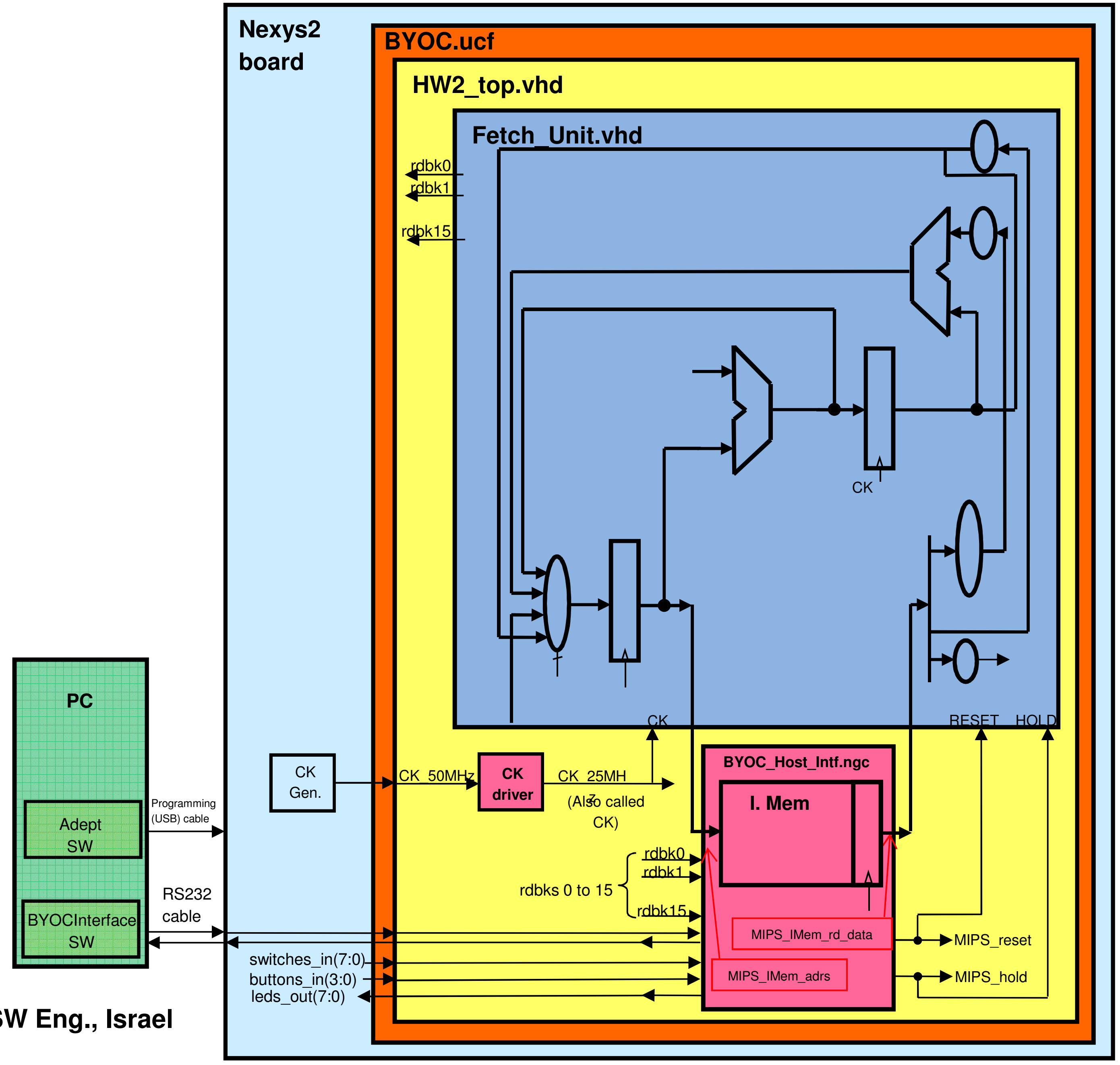


```
at 0 ps: Note: ================== Start checking signals ====================================== (/mips_alu_tb/).
at 0 ps, Instance /mips_alu_tb/Testing_MIPS_ALU/ : Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
Finished circuit initialization process.
at 400 ns(1): Error: Error in ALU_out value in CK cycle 10!!!  was 367CFEF8 instead of 00200470
at 440 ns(1): Error: Error in ALU_out value in CK cycle 11!!!  was FFFF5678 instead of 12340000
at 480 ns(1): Error: Error in ALU_out value in CK cycle 12!!!  was 1234FFFF instead of 00005678
at 520 ns(1): Error: Error in ALU_out value in CK cycle 13!!!  was 00200470 instead of 367CFEF8
at 560 ns(1): Error: Error in ALU_out value in CK cycle 14!!!  was 12340000 instead of FFFF5678
at 600 ns(1): Error: Error in ALU_out value in CK cycle 15!!!  was 00005678 instead of 1234FFFF
ISim>
# run 1.00us
at 1120 ns(1): Error: ========== no of errors is: 6, ================ test FAILED!! =================
ISim>
```

BYOC course infrastructure

- **Support in implementation step**
  - The BIT file is loaded to the board via a SW application supplied by the board manufacturer (Adept by Digilent)
  - Instructions of what to omit (TB signals, TB.vhd file etc.) when migrating from simulation to implementation are given
  - The students get the implementation version of the two pre-prepared components – the clock driver and the BYOC_Host_Intf
  - A User Constraints File (UCF) describing the connections of the i/o signals to actual FPGA pins is also given
  - The BYOC_Host_Intf has an RS232 interface connection to a PC. A special application, the BYOCInterface SW, allows loading of IMem with code. It also allows to run the design in single-clock mode and display 16 values of 32 bit rdbk signals outputted from the design
  - Compare files for this rdbk data is also given. Again, we control which part of the design we want the students to compare

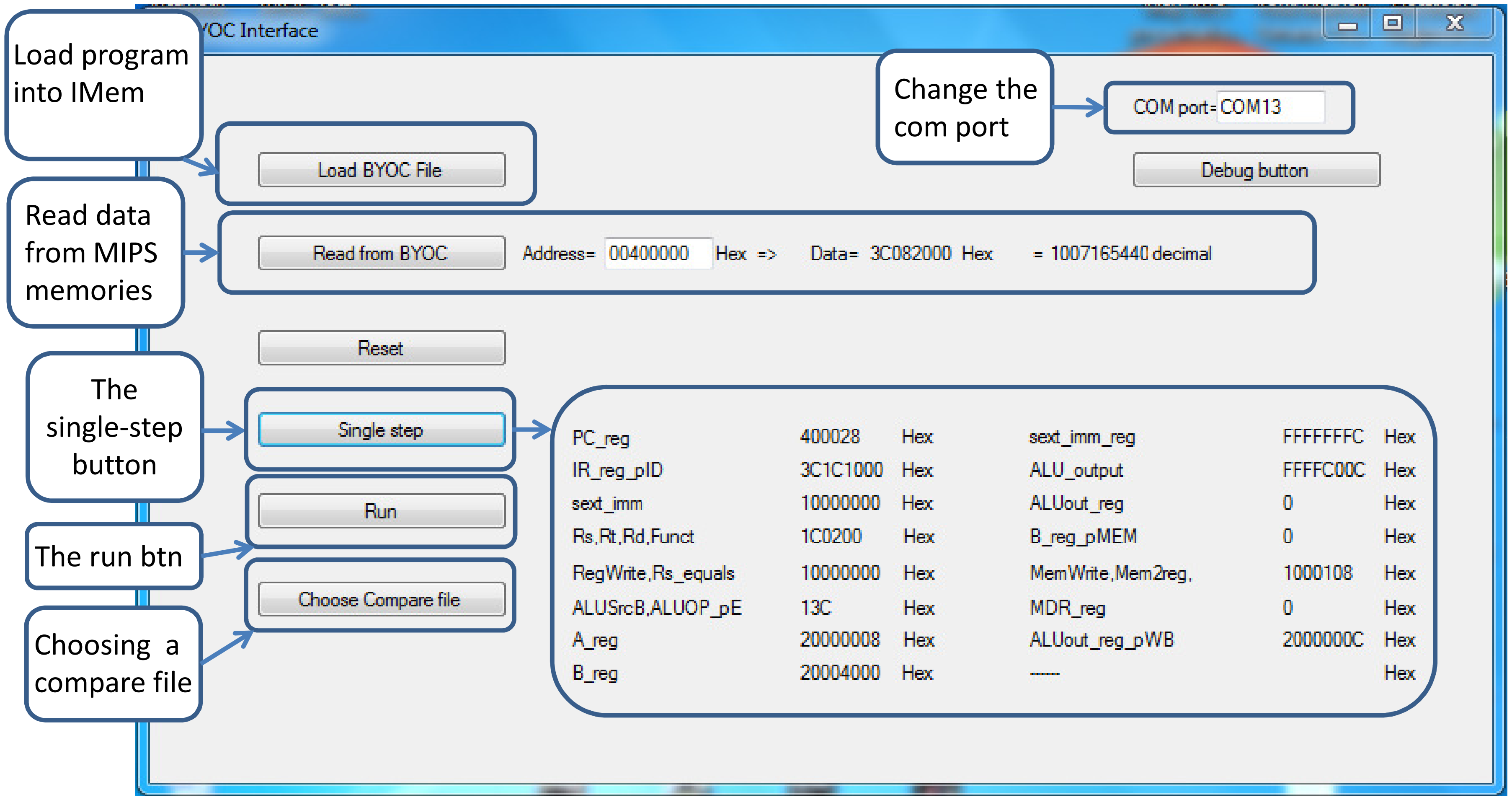
The Fetch Unit Implementation project



Kineret SW Eng., Israel

16/2/2016

BYOCInterface SW panel

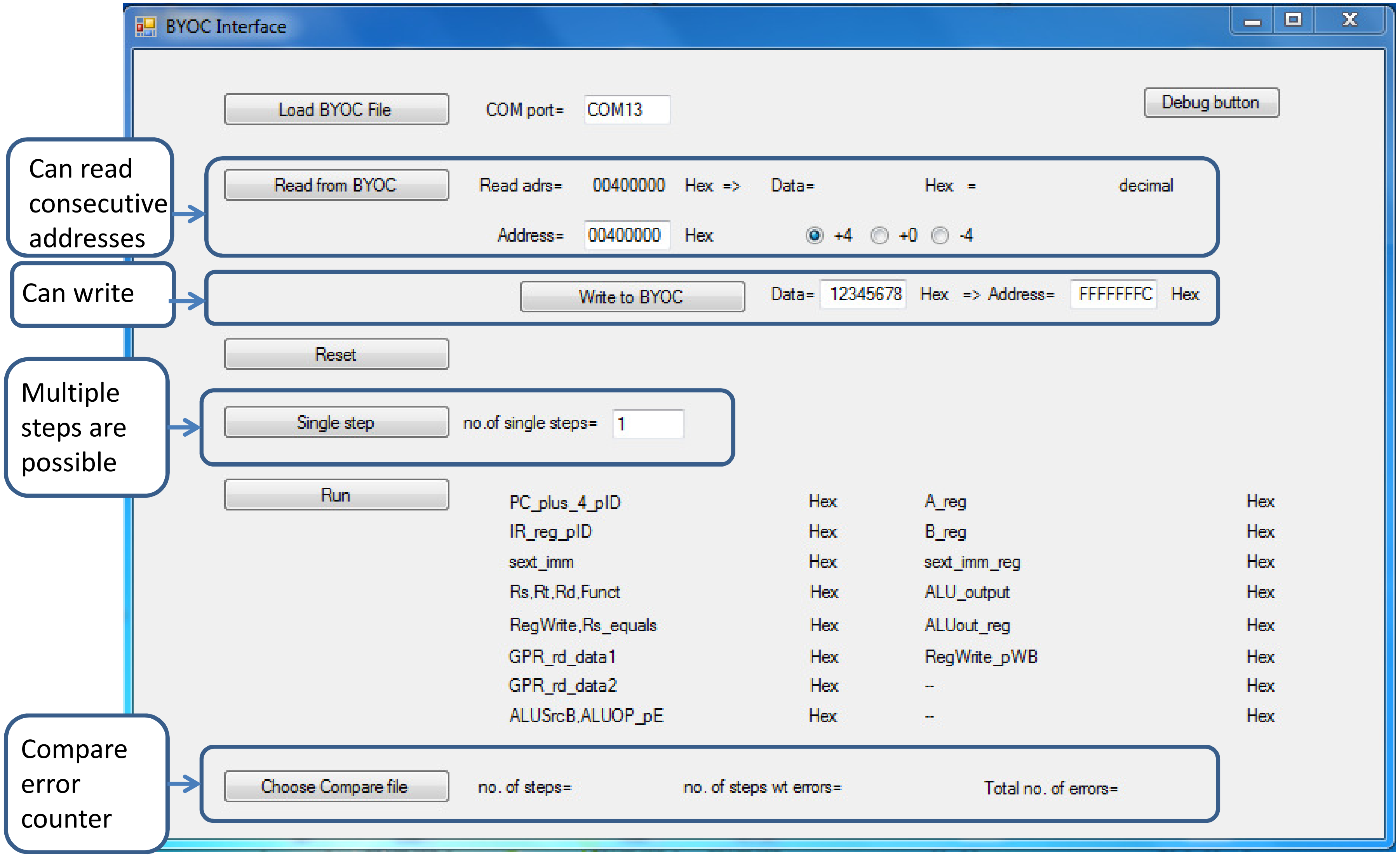


COM port= COM13

Address= 00400000 Hex => Data= 3C082000 Hex = 1007165440 decimal

```
PC_reg              400028     Hex    sext_imm_reg         FFFFFFFC  Hex
IR_reg_pID          3C1C1000   Hex    ALU_output           FFFFC00C  Hex
sext_imm            10000000   Hex    ALUout_reg           0         Hex
Rs,Rt,Rd,Funct      1C0200     Hex    B_reg_pMEM           0         Hex
RegWrite,Rs_equals  10000000   Hex    MemWrite,Mem2reg,    1000108   Hex
ALUSrcB,ALUOP_pE    13C        Hex    MDR_reg              0         Hex
A_reg               20000008   Hex    ALUout_reg_pWB       2000000C  Hex
B_reg               20004000   Hex    ------                         Hex
```

The updated BYOCInterface SW panel



Can read consecutive addresses

Can write

Multiple steps are possible

Compare error counter

BYOC Interface

Load BYOC File COM port= COM13 Debug button

Read from BYOC Read adrs= 00400000 Hex => Data= Hex = decimal

Address= 00400000 Hex [✓ +4] +0 -4

Write to BYOC Data= 12345678 Hex => Address= FFFFFFFC Hex

Reset

Single step no.of single steps= 1

Run

PC_plus_4_pID	Hex	A_reg	Hex
IR_reg_pID	Hex	B_reg	Hex
sext_imm	Hex	sext_imm_reg	Hex
Rs,Rt,Rd,Funct	Hex	ALU_output	Hex
RegWrite,Rs_equals	Hex	ALUout_reg	Hex
GPR_rd_data1	Hex	RegWrite_pWB	Hex
GPR_rd_data2	Hex	--	Hex
ALUSrcB,ALUOP_pE	Hex	--	Hex

Choose Compare file no. of steps= no. of steps wt errors= Total no. of errors=

BYOC course unique features

- We teach the entire HW design process
  - Design & coding (inc. syntax check)
  - Logic simulation
  - Implementation & testing
- We have **total control** on the amount of effort required from the student:
  - In **design** – we decide what "empty" files are given
  - In **simulation** – we determine the parts automatically tested
  - In **implementation** – we determine what is compared
- This is a **great** platform for HW & SW projects in Computer Architecture (adding Floating point, super-scalar, etc.)

Conclusion

- In this paper we described a lab course in which the students actually implement a simplified pipelined MIPS CPU in VHDL
- The course leads the student to build the CPU in a step by step approach that makes it easy to understand the CPU structure
- The course teaches process of designing and testing an FPGA design
- The infrastructure we built allows full control on the effort level required from the student during the design, simulation, and the implementation phases. Thus we can adjust the course for different populations – from Computer Science students to Electrical Engineering students
- This course can be a great platform for many projects related to computer architecture

Thank you

Backup slides

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    (note that Rd=0)
```

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

I-type

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

6	5	5	16
OPCODE	Rs	Rt	imm

j-type

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

6	26
OPCODE	26 bit imm

Empty slide