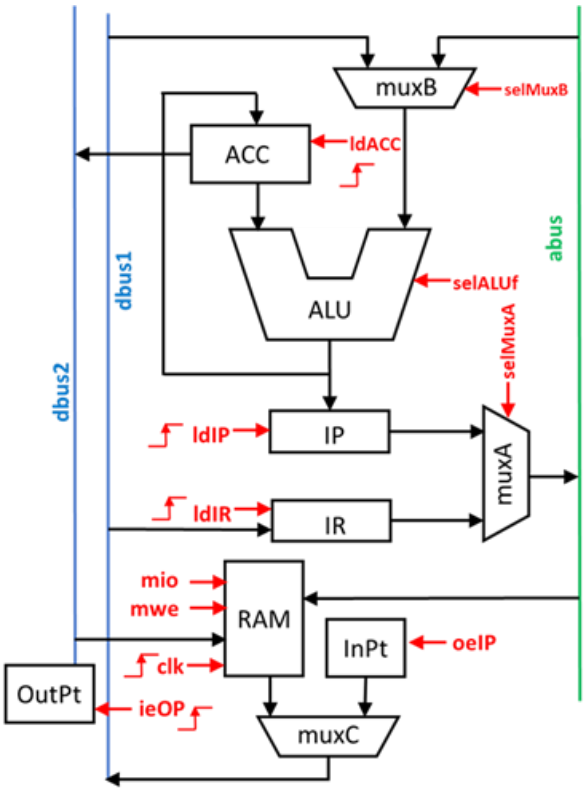
Comp3402 Synthesizing and Simulating CPU2S

C.B.Price January 2024

Purpose	To apply our understanding of digital circuit synthesis using VHDL to a synthesis of our own CPU
Files Required	Summary of VHDL syntax you need can be found here.
ILO Contribution	LO 5
Send to Me	nix
Homework	Read chapter 14

Structure of CPU2S

Here is the complete CPU structure showing the components we shall synthesize using VHDL.



- **ALU** Arithmetic Logic Unit. Does adds, subtracts and other maths ops.
- **Acc**umulator. 16-bit register, gets its input from ALU, outputs onto **dbus2**.
- **muX**A/B/C. Multiplexers (switches) selects one of two inputs according to **selMux**A/B/C signal.
- **IP** Instruction pointer. Counter that starts at 0 and increases by 1 each clock. Used to step through memory
- **IR** Instruction Register, 16-bits. Used to hold current encoded instruction.
- **RAM** 32 words of 16 bits.
- **InPt** 16-bit input port
- **OutPt** 16-bit output port
- **abus** 16-bit address bus
- **dbus1** data bus from memory/InPt upwards
- **dbus2** databus downward into memory/OutPt

Section A. Synthesizing and Simulating the CPU2S Components

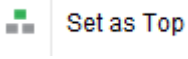
For each of the following components you will

- Complete the component VHDL code template.
- Check your component code using the testbench provided.

Please note, words in **bold** are the names of signals in each Component.

Here is a list of Design Sources you will find

Also, here are the associated Simulation Sources (Test-benches). You have to select the one which belongs to your current Design Source. Here's how. Say you want to use **mux_tb**. Right click this line and then choose

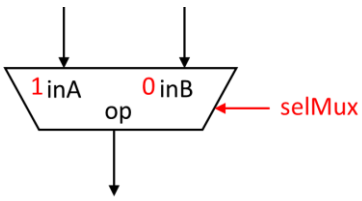


from the dialogue box.

---

1. The MUX

Here is a MUX. It has two inputs **InA** and **InB** and one output. There is also a selection input, **sel**



The MUX output **op** should be **inA** when **sel** is high otherwise, it should be **inB**.

(a) Open up the file **ISE_mux.vhd** and code this behaviour. Note we are not in a sequential *process* block, so we cannot use if-then-else. Instead we must use a **when … else**. Synthesize the MUX.

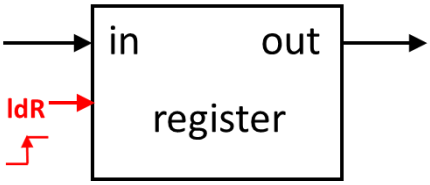
(b) Select the testbench file **ISE_mux_tb** and check the behaviour of your MUX.

---

2. Registers

Registers are like D-flipflops. You remember, the input only gets through to the output at the rising edge of a clock pulse. In fact our 16-bit registers can be thought of as 16 D-flipflops in parallel, though we won't code that, but let Vivado do the work for us.

So here's a register with input and output and a clock signal

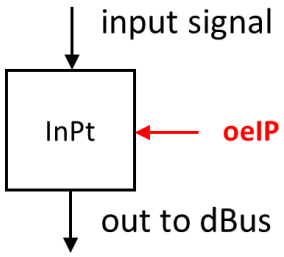


(a) Open up the file **ISE_reg.vhd** and code the required behaviour. You may want to look at your D-flipflop code. The only real difference is that here **data_in** and **data_out** are 16-bit wide, they are declared as logic *vectors*. Run Synthesis.

(b) Select the testbench **IDE_reg_tb.vhd** and run the simulation. You'll probably get some **UUUU** which means undefined logic level. Probably a consequence of a stand-alone test.

3. Input Port

The input port is designed to capture external data and feed it into the CPU, it is 16-bits wide, but it is not a register. It only passes its input through to its output when a control signal is high. The control signal is shown in red and means '**o**utput **e**nable **I**nput **P**ort'.



(a) Open up the file **ISE_InOutPort.vhd.** You must add a line or two of code to obtain the desired behaviour. This line will start with **data_out <=** to assign a value to the output signals.

The Port output **data_out** should be **data_in** when **load** is high. In all other cases it should be low. There is no process block, so you will have to use the good old **when … else** construct. To assign a value of zero then use the syntax **X"0000"** which specifies a 16-bit hexadecimal value of 0. Alternatively you could write **"0000000000000000"**.

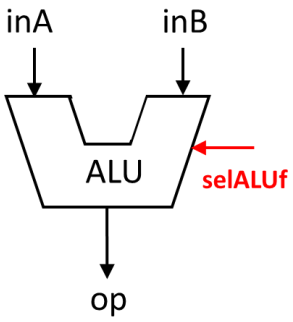
There is another way of coding this alternative. Here it is abstractly. I guess you can work out what it means.

out <= in when (condition) else (others => '0')

(b) Select the testbench **ISE_IOPort.vhd** and test your synthesized circuit.

4. The ALU

Here's the ALU which has two inputs and also a control signal **selALUf** which selects the function the ALU is instructed to perform.



(a) Code up the ALU functionality, using the table of what the **op** will be according to the **sel_ALUf** signal which directs the ALU what to do. You'll need a chain of **when … else … when**

sel_ALUf =	op
"001"	**inB**
"010"	**inB + 1**
"011"	**inA + inB**
"100"	**inA - inB**

after your final else you should probably include the following to catch any user programming errors:
else (others => '0'); The table shows that our ALU has a limited set of arithmetic operations; the last two are add and subtract. The first is a 'pass-through' mode on **inB** and the second increments **inB**. You'll understand these when we assemble the entire CPU.

(b) In addition to the line **op <= …** you will also need to complete the line **zero <= …** This is the 'zero flag' which is used in branch instructions. The zero flag is set to '1' when **inA** is zero, else the flag is '0'.

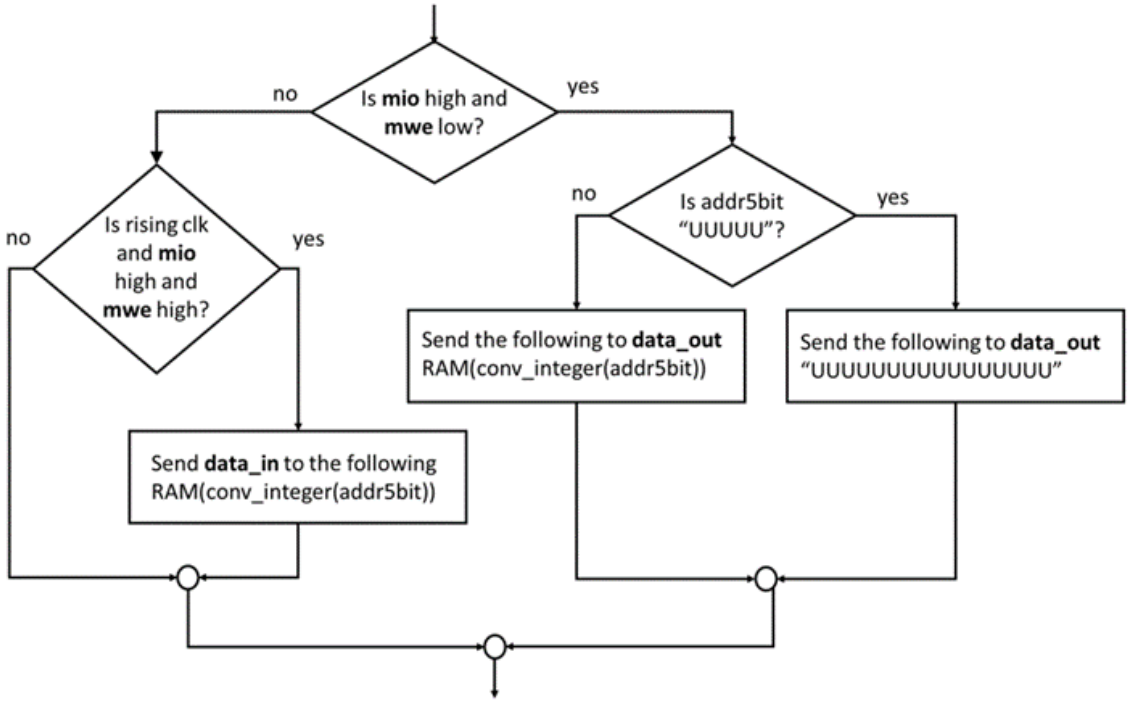
(c) Pull up the testbench **ISE_alu_tb.vhd** and check out your synthesized circuit.

---

5. Memory – RAM

This needs thought, concentration, a southerly wind, and faith.
[HINT1] The code in chapter 13 page 14 will help, but this is certainly not the whole story.
See also Chapter 14.6.3.

The solution is best expressed through this flow diagram,


Is **mio** high and **mwe** low?
no
yes
Is rising clk and **mio** high and **mwe** high?
no
yes
Send **data_in** to the following RAM(conv_integer(addr5bit))
Is addr5bit "UUUUU"?
no
yes
Send the following to **data_out** RAM(conv_integer(addr5bit))
Send the following to **data_out** "UUUUUUUUUUUUUUUU"

Section B. Programming our CPU2S

This will use the **NoC_CPU2S** file where all the individual components have been assembled and connected. You will only need to change the **compRAM** component.

---

1. A First Program

Let's investigate a program to input a number (could come from switches), then add a number from memory, then output the number (could go to LEDs). Here's the program

IN	8000	*Input into **ACC***
ADD 11	2011	*Add data from memory address 11 (hex) to the **ACC***
OUT	9000	*Send data from **ACC** to the output port*
HLT	7000	*Halt processing.*

The left shows the mnemonics, the right shows the opcodes you must type into RAM code-segment. Open the **ISE_RAM.vhd** design source, and you'll see the code-segment

```
27      -- code segment
28      X"0000",
29      X"0011",
30      X"0000",
31      X"0000",
32      X"0000",
33      X"0000",
```

(a) Replace the first 4 rows with the opcodes for the program. Do not add or delete rows. So the first row will become **X"8000",**

You can find the data segment further down; it looks like this.

```
44      -- data segment
45      X"0002", X"0003", X"0004", X"0005",
46      X"0001", X"0001", X"0002", X"0006",
47      X"0000", X"0000", X"0000", X"0000",
48      X"0000", X"0000", X"0000", X"0000"
```

So your instruction **ADD 11** will get the data at address 11 (hex) which is 17 (dec). That's the **X"0003"** on line 45.

(b) Synthesize your design.

(c) Run the testbench **NoC_CPU2S_tb_micro** and you'll get a waveform. Your job is to interpret this. Remember the program does an input into ACC, then adds a number to ACC then outputs the ACC to the output port. I suggest you take a snip of the waveform, put it into a DrawCanvas in Word and annotate the canvas.

(d) Here's some basic checks

(i) Check the instruction pointer **IP** increments 0, 1, 2, 3, …
(ii) Check when each instruction is loaded into the instruction register **IR** and add the instruction mnemonic alongside the op-code in the IR.

(e) Now let's follow the data. We shall concentrate on the accumulator **ACC** since this is where inputs, outputs and the results of ALU operations are stored. Remember the rising edge of the clk is when data is input into the **ACC.**

(i) Find out the value of the data on the input port.
(ii) Find out then this is loaded into the **ACC**. Label the appropriate rising clock edge.

(iii) Look for the **ADD 11** instruction in the **IR.**
(iv) Find out when this is loaded into the **ACC.** Label the appropriate rising clock edge.

(v) Find out when the result is sent to the output port.
(vi) Label the appropriate rising clock edge. (Remember the output port, like the accumulator inputs its data only on a rising clock edge.

2. Optional Programs

Here's the full instruction set. The 'aa' in the opcode column is an address in hex.

LDA addr	00aa	*load **ACC** with data from memory at address aa(hex)*
STO addr	10aa	*Store **ACC** into memory at address aa*
IN	8000	*Input into **ACC***
ADD addr	20aa	*Add data from memory address 11 (hex) to the **ACC***
OUT	9000	*Send data from **ACC** to the output port*
JMP addr	40aa	*Set **IP** to code segment address aa (i.e. jump execution there)*
HLT	7000	*Halt processing.*

You might want to investigate the following and work out what they do. Note Program 1 does not halt!

Program 1	IN
	ADD 19 (hex)
	JMP 1

Program 2	LDA 10
	ADD 11
	STO 12
	OUT
	HLT