

## Software Lab 1 : Minimal 'CISC' Processor (MCP)

The aim of this lab is to examine the internal operations of one possible hardware realisation of the SimpleCPU processor. This hardware architecture is defined as a VHDL simulation model, allowing you view the processor's operation as a waveform diagram i.e. to see each individual signal, data bus and register value. VHDL is a hardware description language i.e. a textual description, defining the hardware units within the processor and how they are connected. In addition to simulations this model can also be synthesised i.e. converted into a configuration file and used to configure physical hardware devices e.g. Field Programmable Gate Arrays (FPGA).

**Note**, the implementation you will be using is based on a Minimal CISC Processor (MCP) I wrote many moons ago. Processing performance was not a design goal for this CPU, the main objective was to reduce hardware usage. The same instruction-set could also be implemented as a RISC architecture, as shown in:

[http://www.simplecpudesign.com/simple\\_cpu\\_v1/index.html](http://www.simplecpudesign.com/simple_cpu_v1/index.html)

The key point to note is that an instruction-set can be implemented in a number of different hardware architectures, each optimised for a particular application domain.

A VHDL project called `mcp` has already been created and can be downloaded from the module web page (link under lab script).

Using your preferred web browser download the file `mcp.zip` to `c:\temp`. **Note**, you can run this simulation from your home directory, however, network speeds may cause it to run **VERY** slowly. Right click on this file selecting 'Extract all...' to unzip it. To start the Vivado project navigator left click on the Windows icon in the bottom left of the screen and type :

 -> Vivado 2017.2

This may take a few minutes to load, eventually you should see the main GUI, as shown in figure 1. To open the downloaded project click on the Open Project icon, or the pull down:

File -> open project

This will open the 'Open project' box, browse to the directory where you unzipped this project and select the file `mcp.xpr`. Again, this may take a few minutes to open.

Within the main project window you will be able to access the VHDL model files, double click on the memory component `MEM`, this will open the VHDL text file `ram.vhd` in the main panel, as shown in figure 2. This VHDL description defines a simple read / write memory, capable of storing 256 x 16bits. As the processor is based on a Von Neumann architecture it will be used to store instructions and data. As with the `CPUSim` model the first instruction is stored at address 0 i.e. its boot or reset vector.

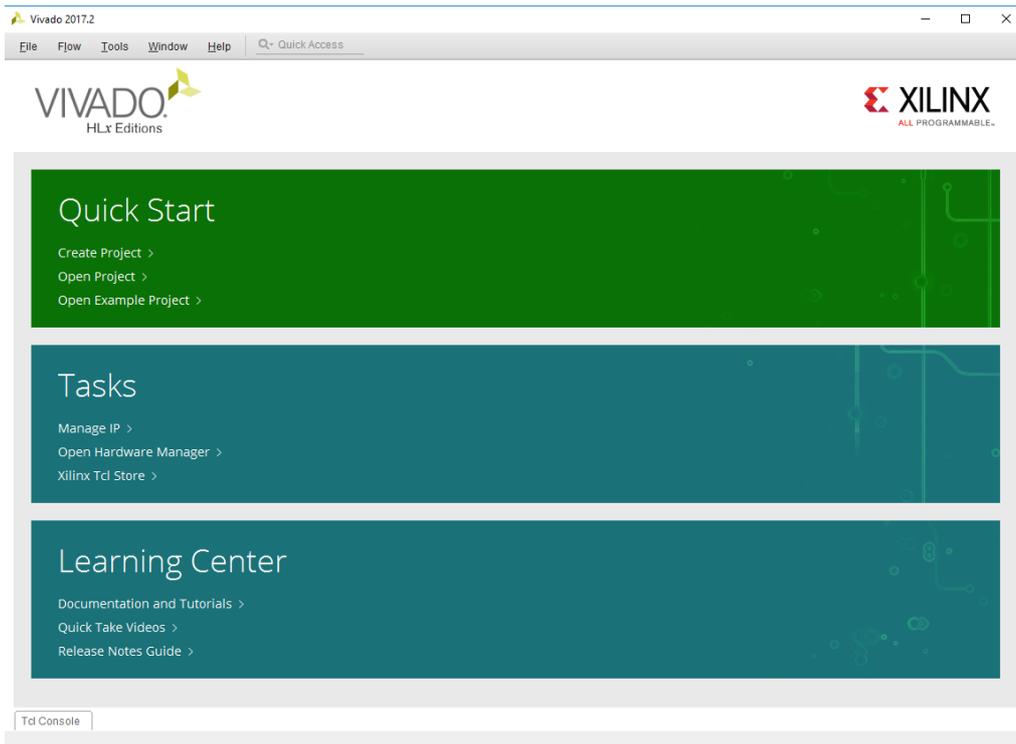


Figure 1: Vivado main window

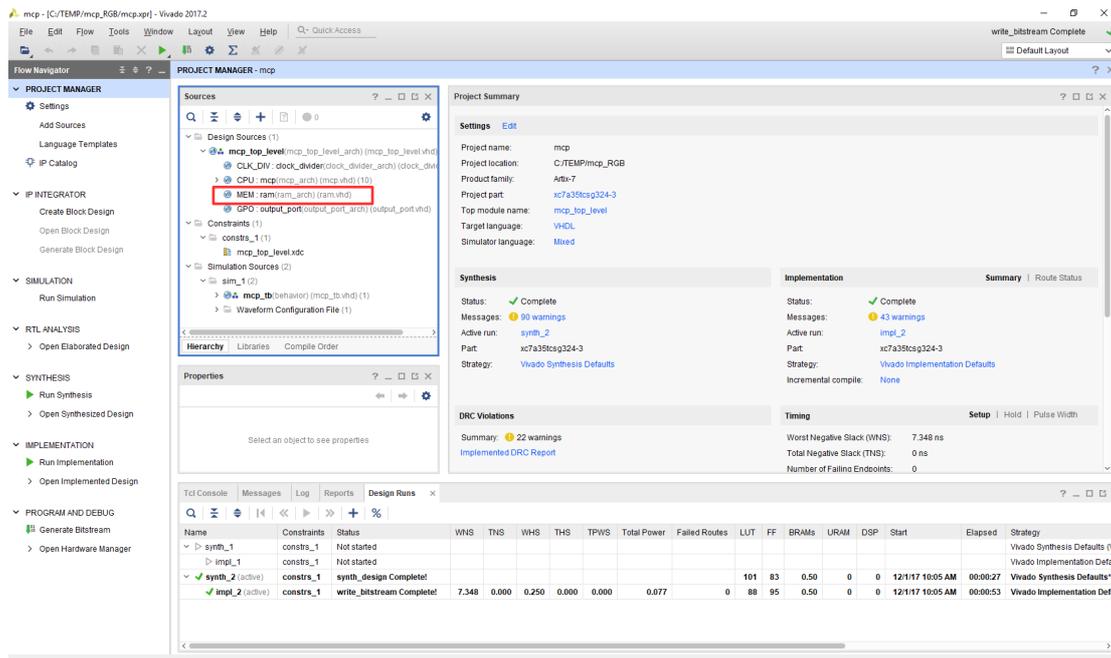


Figure 2: project files

### Task 1

Click on the `ram.vhd` text file in the main panel and scroll down to line 44, as shown in figure 3. This section defines this hardware component's interface (ports) i.e. the physical pins that are need on the actual hardware device:

- `clk` : clock, control signal, synchronising memory transactions with the rest of the CPU's hardware.

- `addr` : address bus, uni-directional, input, 8bits, selecting the memory location to be read or written to.
- `din` : data input, uni-directional, input, 16bits, data from the processor to be written to selected memory location.
- `dout` : data output, uni-directional, output, 16bits, data requested by the processor from the selected memory location.
- `rw` : read / write control signal, uni-directional, input, 1bit, from the processor, a logic 1 indicates that the memory transaction is a READ, a logic 0 indicates that it is a WRITE.

**Note**, for some memory technologies the data bus is a single, bi-directional bus. This reduces the number of physical pins needed on the device, as the same pins are used as both inputs and outputs (multiplexed), but this does complicate its operations i.e. need to connect and disconnect different drivers. For the hardware we will be using (FPGAs) these types of bi-directional buses are not generally supported.

```

44 ENTITY ram IS
45   GENERIC(
46     ADDR_WIDTH : INTEGER := 8;
47     DATA_WIDTH : INTEGER := 16 );
48   PORT (
49     clk : IN  STD_LOGIC;
50     addr : IN  STD_LOGIC_VECTOR( ADDR_WIDTH-1 DOWNTO 0 );
51     din  : IN  STD_LOGIC_VECTOR( DATA_WIDTH-1 DOWNTO 0 );
52     dout : OUT STD_LOGIC_VECTOR( DATA_WIDTH-1 DOWNTO 0 );
53     rw   : IN  STD_LOGIC
54   );
55 END ram;

```

Figure 3: VHDL interface

```

84 SIGNAL data_ram : DATA_RAM_TABLE := DATA_RAM_TABLE'
85 (
86   DATA_RAM_WORD' ("0000000000000000"), --00
87   DATA_RAM_WORD' ("0000000000000000"), --01
88   DATA_RAM_WORD' ("0000000000000000"), --02
89   DATA_RAM_WORD' ("0000000000000000"), --03
90   DATA_RAM_WORD' ("0000000000000000"), --04
91   DATA_RAM_WORD' ("0000000000000000"), --05
92   DATA_RAM_WORD' ("0000000000000000"), --06
93   DATA_RAM_WORD' ("0000000000000000"), --07
94   DATA_RAM_WORD' ("0000000000000000"), --08
95   DATA_RAM_WORD' ("0000000000000000"), --09

```

Figure 4: memory locations

Next, scroll down to line 84, as shown in figure 4. This section of VHDL defines the contents of each memory location. Data values are highlighted in **PURPLE** (or blue), addresses are included as comments in **GREY** e.g. --01. To program this processor you will need to manually edit these binary data strings with the correct machine code instructions or data values i.e. value  $10_{10}$  = binary value "0000000000001010".

The instructions supported by this processor are equivalent to the instruction formats used in the CPUSim laboratories and discussed in the associated lectures. A full list of the supported instructions is shown in figure 5. The processor uses a 16bit fixed

length instruction, 4bit opcode, followed by an 8bit immediate value (KK), data memory address (PP) or an instruction memory address (AA). Jump instructions also contain a 2bit bit field, selecting the conditional flag used.

Load	ACC	kk	:	0000	XXXX	KKKKKKKK
Add	ACC	kk	:	0100	XXXX	KKKKKKKK
Add	ACC	pp	:	1100	XXXX	PPPPPPPP
Input	ACC	pp	:	1010	XXXX	PPPPPPPP
Output	ACC	pp	:	1110	XXXX	PPPPPPPP
Jump	U	aa	:	1000	XXXX	AAAAAAAA
Jump	Z	aa	:	1001	00XX	AAAAAAAA
Jump	C	aa	:	1001	10XX	AAAAAAAA
Jump	NZ	aa	:	1001	01XX	AAAAAAAA
Jump	NC	aa	:	1001	11XX	AAAAAAAA

Figure 5: instruction set

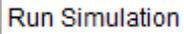
**IMPORTANT**, the main difference between this processor and the one simulated in CPUSim is that memory is not byte addressable i.e. each address represents a 16bit value. Make sure you understand how this will affect your program.

## Task 2

Edit the `ram.vhd` text file inserting the required machine code for the assembly language program below. The required machine code is shown on the right.

start:	ADDR	DATA
input A	0	1010 0000 00000100
add 01	1	0100 0000 00000001
output A	2	1110 0000 00000100
jump start	3	1000 0000 00000000
A: .data 2 0	4	0000 0000 00000000

Make sure you understand how each instruction and data value are represented. To save these edits click on the save icon  or press CTRL-S.

To simulate this processor and its program click on the  icon, within the Simulation panel (middle, left side). Then select 'Run behavioural simulation' from the options given. This will open 'Launch Runs' window, ensure that the radio button 'Launch runs on local host' is selected and click OK, status information is shown in the top right of the main window, after a small delay a waveform trace as shown in figure 6 will appear.

By default the hardware simulation is run for 1us, however, this may not be long enough to see the processor executing all of the instructions in a program. To step through the simulation click on the  icon, this will step through the simulation in 1us steps, time steps defined in the text box. To restart the simulation click on the  icon, this will return the simulation to time 0s. These icons are in a block in the top middle of the GUI i.e. 



**Question:** how long does an instruction take to execute i.e. the number of clock cycles? Does each instruction take the same amount of time to execute? Why do some instructions take longer than others?

**Hint,** look at the IR and PC values to estimate time.

**Question:** modify your program so that it adds 0xAA to variable A on each loop. Run the simulation for 15ms (15,000us). What is the value in the ACC at this simulation time. Do you understand why the value is 0x52?

### Task 3

In addition to a memory device this processor also has a 16bit output port i.e. the processor can control the state of 16 wires, setting their individual values to either a logic '1' or a logic '0'. The hardware used to implement this functionality is memory mapped to addresses 0xFF and 0xFE i.e. its functionality is triggered when the processor writes to memory address 0xFF or 0xFE. To identify when this operation is performed by the processor the address decoder logic shown in figure 8 is used.

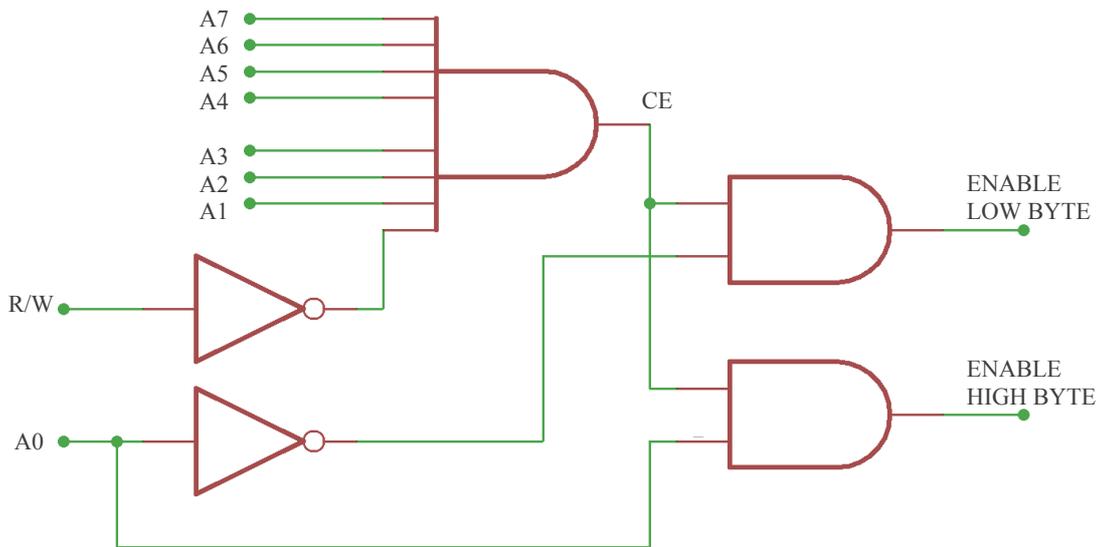


Figure 8: address decoding logic

When the address bus lines A7 – A1 are all logic '1' and the read/write (R/W) control line is a logic '0' the CE line is enabled. This is then combined with A0, enables either the low or high byte on the output port to be updated .

**Note,** the output port is just a bank of 16 flip-flops, their D inputs are connected to the data bus (D7–D0), their Q outputs connect to external wires (LEDs). These signals can not be read by the processor i.e. are not connected to the data bus. The flip-flops also have an enable line, if this is low the clock (update signal) is ignored, eight flip-flops are connect to ENABLE\_LOW\_BYTE, eight to ENABLE\_HIGH\_BYTE.

**Question:** can you see why this hardware will only generate enable signals for memory transactions that write to addresses 0xFE and 0xFF?

Edit the `ram.vhd` text file inserting the required machine code for the assembly language program below.

ADDR	INSTRUCTION	DESCRIPTION
0	load 0x00	ACC <- 0xFF
1	output 0xFE	M[0xFE] <- ACC
2	load 0x01	ACC <- 0
3	output 0xFE	M[0xFE] <- ACC
4	jump 0	repeat

**Question:** what does this program do? Next, modify this program to implement the following pseudo code:

```
LOOP:
    WRITE TO OUTPUT PORT 0xFFFF
    WRITE TO OUTPUT PORT 0xA0A0
    WRITE TO OUTPUT PORT 0x0505
    JUMP LOOP
```

Rerun the simulator to confirm that the output port is updated with the correct values.

**Question:** why will the program produce intermediate results i.e. values other than 0xFFFF, 0xA0A0 and 0x0505? Is there a software solution to prevent this? Is there a hardware solution to prevent this?

#### Task 4

To control the speed at which the output port is updated we will need to implement a software delay loop i.e. a piece of code that effectively wastes processor cycles, delaying the progression of the program.

Edit the `ram.vhd` text file adding the required machine code to implement the following program:

ADDR	INSTRUCTION	DESCRIPTION
0	load 0	ACC <- 0x00
1	output 0x09	M[0x09] <- ACC
2	load 0	ACC <- 0x00
3	add 1	ACC <- ACC + 1
4	jump NC 3	loop until overflow
5	input 0x09	ACC <- M[0x09]
6	add 1	ACC <- ACC + 1
7	output 0x09	M[0x09] <- ACC
8	jump 2	repeat
9	DATA	variable COUNT

**Question:** can you see why the rate at which the variable COUNT is updated is delayed?

If the processor's clock speed is approximately 6.25KHz, write a program to turn the output port's pins on and off every 0.25 seconds. Rerun the simulator to confirm that the output port now updating at a slower rate.

When your program is working correctly you can convert this design into a format that can be downloaded into a FPGA i.e. actual hardware. Connected the output port are four RGB LEDs and four ordinary LEDs, as shown in figure 9, a configuration file connecting VHDL signal names to actual FPGA pins.

Driving a logic '1' onto an output pin will cause the associated LED to be illuminated. In addition to these outputs there are also four inputs (*sw*). These are not connected to the processor, but are logically ORed with output port bits 0,1,2 and 3. This allows you to test if the FPGA has been configured correctly i.e. if one of these switches is moved into the logic '1' position an LED will be illuminated.

```

6 #RGB LEDs
7
8 set_property -dict { PACKAGE_PIN E1      IOSTANDARD LVCMOS33 } [get_ports { output[0] }]; #led0_b
9 set_property -dict { PACKAGE_PIN F6      IOSTANDARD LVCMOS33 } [get_ports { output[1] }]; #led0_g
10 set_property -dict { PACKAGE_PIN G6      IOSTANDARD LVCMOS33 } [get_ports { output[2] }]; #led0_r
11 set_property -dict { PACKAGE_PIN G4      IOSTANDARD LVCMOS33 } [get_ports { output[3] }]; #led1_b
12 set_property -dict { PACKAGE_PIN J4      IOSTANDARD LVCMOS33 } [get_ports { output[4] }]; #led1_g
13 set_property -dict { PACKAGE_PIN G3      IOSTANDARD LVCMOS33 } [get_ports { output[5] }]; #led1_r
14 set_property -dict { PACKAGE_PIN H4      IOSTANDARD LVCMOS33 } [get_ports { output[6] }]; #led2_b
15 set_property -dict { PACKAGE_PIN J2      IOSTANDARD LVCMOS33 } [get_ports { output[7] }]; #led2_g
16 set_property -dict { PACKAGE_PIN J3      IOSTANDARD LVCMOS33 } [get_ports { output[8] }]; #led2_r
17 set_property -dict { PACKAGE_PIN K2      IOSTANDARD LVCMOS33 } [get_ports { output[9] }]; #led3_b
18 set_property -dict { PACKAGE_PIN H6      IOSTANDARD LVCMOS33 } [get_ports { output[10] }]; #led3_g
19 set_property -dict { PACKAGE_PIN K1      IOSTANDARD LVCMOS33 } [get_ports { output[11] }]; #led3_r
20
21 #LEDs
22
23 set_property -dict { PACKAGE_PIN H5      IOSTANDARD LVCMOS33 } [get_ports { output[12] }]; #led
24 set_property -dict { PACKAGE_PIN J5      IOSTANDARD LVCMOS33 } [get_ports { output[13] }]; #led
25 set_property -dict { PACKAGE_PIN T9      IOSTANDARD LVCMOS33 } [get_ports { output[14] }]; #led
26 set_property -dict { PACKAGE_PIN T10     IOSTANDARD LVCMOS33 } [get_ports { output[15] }]; #led
27
28 #Slide SW
29
30 set_property -dict { PACKAGE_PIN A8      IOSTANDARD LVCMOS33 } [get_ports { sw[0] }]; #sw0
31 set_property -dict { PACKAGE_PIN C11     IOSTANDARD LVCMOS33 } [get_ports { sw[1] }]; #sw1
32 set_property -dict { PACKAGE_PIN C10     IOSTANDARD LVCMOS33 } [get_ports { sw[2] }]; #sw2
33 set_property -dict { PACKAGE_PIN A10     IOSTANDARD LVCMOS33 } [get_ports { sw[3] }]; #sw3
34

```

Figure 9: Pin configuration file - LEDs and Switches

## Task 5

To implement this system click on the  **Run Synthesis** icon. Whilst the software tools are converting the VHDL models used in the simulation into hardware i.e. logic gates & registers, a progress bar is displayed in the top right of the screen

 **Running synth\_design**  , a more detailed view can be seen by clicking on the  **Log** tab at the bottom of the screen.

When this process is complete a *Synthesis Completed* box will appear, select the *Run Implementation* button then click on OK to continue. Alternatively you can click on the  **Run Implementation** icon. Again a progress bar is displayed in the top right of the screen.

The implementation phase selects which logic gates and registers in the FPGA should be used to construct this system. It then allocates routing resources (wires) within the hardware to actually implement the required circuits. When complete the *Implementation Completed* box will appear. Select the *Generate Bitstream* option, then click on OK to continue. Alternatively, you can click on the  **Generate Bitstream** icon under the *Program And Debug* icon.

When the FPGA configuration file (bit stream) is complete the *Bitstream Generation Completed* box will appear. Select *Open Hardware Manager*, then click on OK to continue. Alternatively, you can click on the **> Open Hardware Manager** icon.

You can now program the FPGA board shown in 10. These boards are available from boxes from the back of the lab. They do **NOT** need an external power supply, simply plug the USB cable into the front of the PC.

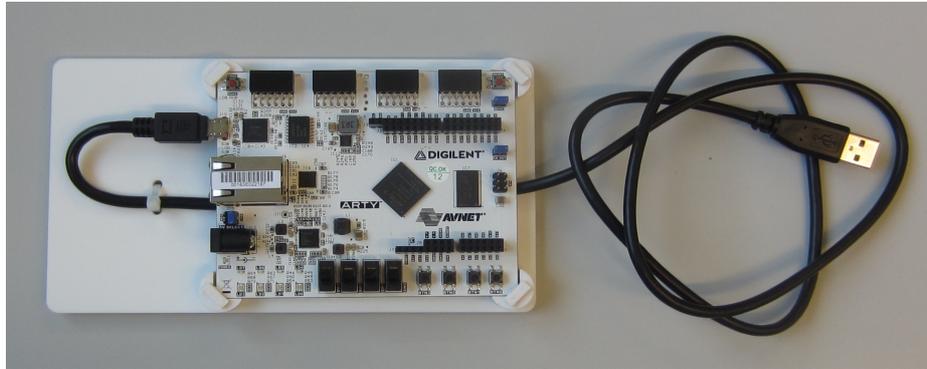


Figure 10: FPGA board

On the side panel under the **> Open Hardware Manager** icon, click on the **Open Target** icon, selecting **Auto Connect**. The software should detect the Digilent FPGA, allowing you to click on the **Program Device** icon, allowing you to select and program the Xilinx XC7a35t\_0 FPGA on this board. This will open a *Program Device* box, allowing you to select the bitstream configuration file to be downloaded i.e. the .bit file. By default previously generated file should be selected:

```
<root folder>/mcp/mcp.runs/impl_2/mcp_top_level.bit
```

Click on the *Program* button to configure the FPGA. Once programmed try pressing one or more of the switches, as shown in figure 11.

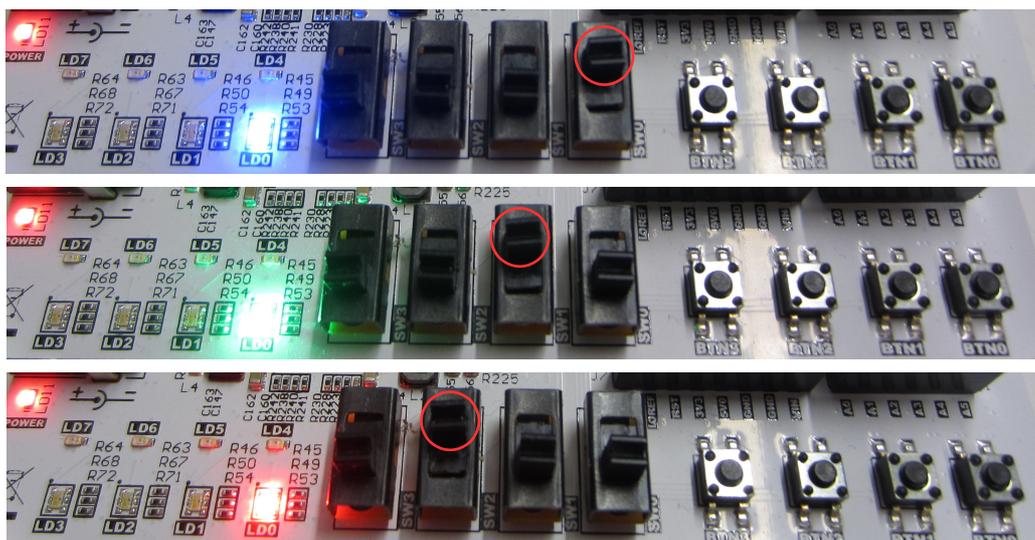


Figure 11: FPGA board test switches

These switches have been combined into the FPGA design as shown in figure 12. These four signals are logically ORed with GPO bits 0 – 3, allowing you to control the first two RGB LEDs. These switches can not be accessed by the processor as it does not have an input port. Their function is to confirm that an FPGA design has been correctly downloaded onto the board i.e. if you can not change these LEDs a configuration file has not been downloaded into the FPGA.

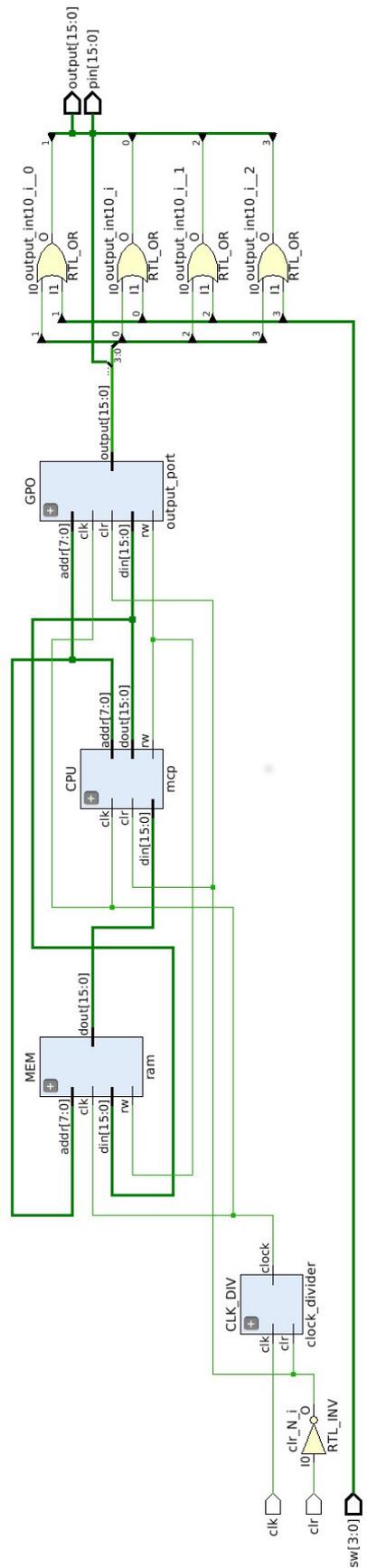


Figure 12: FPGA design