Laboratory 3 : SimpleCPU_v1a Parallel IO

The aim of this lab is to demonstrate how a general purpose processor can be used to replace application specific hardware. The first commercial processor developed was the Intel 4004. This IC and its support chips were developed in 1971 as a replacement for a set of Application Specific Integrated Circuits (ASIC), that were to be used in a range of desktop calculators i.e. rather than designing a set of three ASICs, the Intel engineers designed one set of ICs that could be reconfigured using software to perform different tasks. To allow the processor to interface to the world around it we need some General Purpose Input Output (GPIO) devices. These allow the processor to access external sensors (input devices) and actuators (output devices). Then using control rules defined in software implement the desired system functions. For this practical we will be updating the bug trap implemented in laboratory 5 and relaunching it as the iTrap :). This new and improved bug trap replaces the previous hardwired circuits with the simpleCPU processor, allowing additional features to be added using software. At the end of this practical you will understand how to:

- Use GPIO ports: input / output (I/O)
- Design polling based systems i.e. software-driven I/O.
- Implement time delays in software.
- Evaluate the trade-offs involved in implementing system functions in software or application specific hardware i.e. flexibility vs hardware efficiency.

An ISE project called `bug_trap_v3`, has been created and can be downloaded from the webpage. Using your preferred web browser download this zip file to `c:\temp`. Right click on this file selecting 'Extract all…' to unzip it. To start this practical click on the start button and select the ISE project navigator.

 `-> Xilinx 64bit project navigator`

This may take a few seconds (a minute when the network is busy) to start. To open this project left click on the `File` pull down within the project navigator window :

```
File -> Open Project
```

Then browse to the directory where you unzipped this project and select : `bug_trap_v3.xise`. Next, within the Hierarchy window double click on the top level schematic `top_level.sch`, this is the same circuit used in laboratory 5 (Flip-flops and registers practical). Left click on the `bug_trap_controller` symbol, then click on the push into symbol icon  to enter the schematic shown in figure 1. Finally, highlight the `computer` symbol and push into this symbol to view the simpleCPU system, shown in figure 2.

Task 1

To interface the simpleCPU system to the real world it has been equipped with two GPIO ports: `PORTA` and `PORTB`. These are 8bit interface devices, having an 8bit input port and an 8bit output port i.e. 16 interface lines each. These devices are attached to the processors address, data and control buses, therefore, from the processor's point of view they are just "memory" devices i.e. data can be accessed

using the `LOAD` and `STORE` instructions. Each "storage" element within the GPIO port is assigned an address i.e. mapped to a memory location, so that the processor can access this data using the absolute addressing mode.

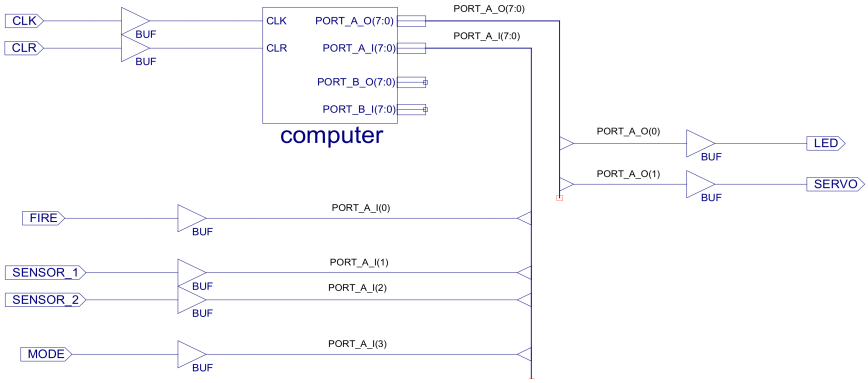

Figure 1 : SimpleCPU bug trap controller

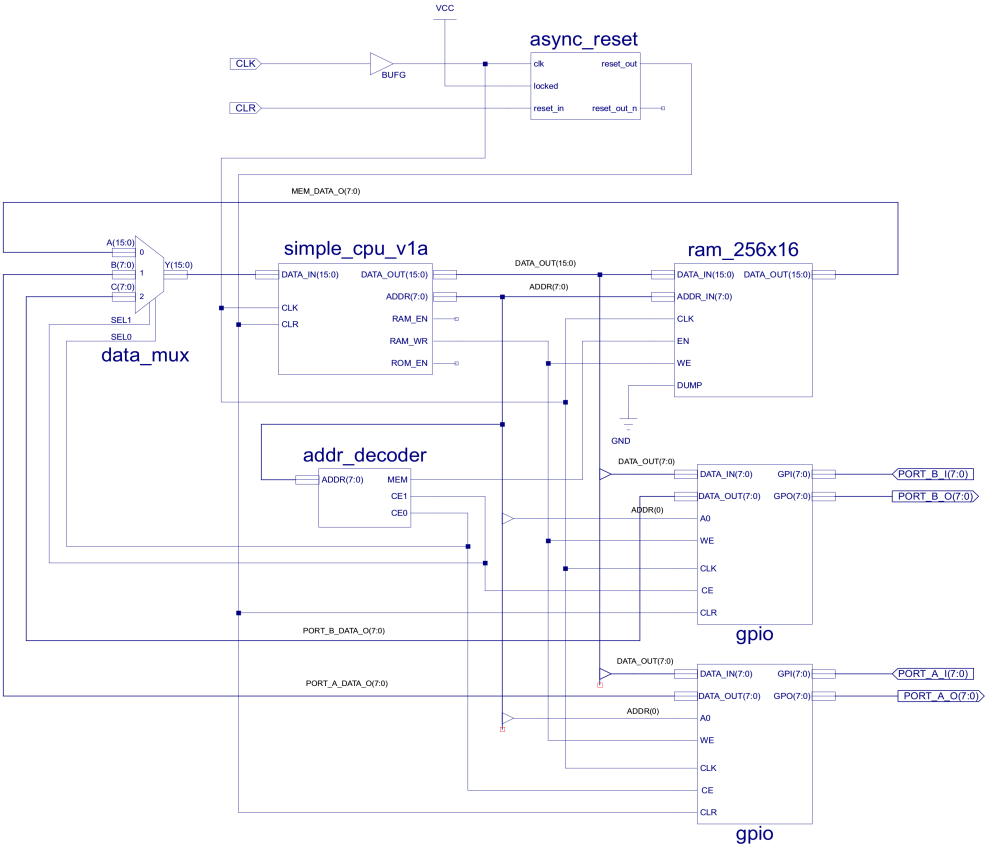

Figure 2 : SimpleCPU with two GPIO ports

The GPIO port contains two 8bit addressable locations, selected between using address bus bit-0. If data is read from the GPIO port when address bus bit-0 is :

- 0 : data is read from the input port i.e. the current value on the General Purpose Input (GPI) port's input pins.
- 1 : data is read from the output port i.e. the value previous written to the General Purpose Output (GPO) port by the program.

If data is written to a GPIO port it will be written to the output register, updating the value stored on the GPO pins.

Note, changing address bit-0 does not affect where this data is written.

The GPIO port's internal registers can be mapped to any of the 256 addressable memory locations. However, as the processor's boot vector i.e. the address of the first instruction, is address zero, GPIO ports are typically assigned addresses at the top of the memory map (away from program code and data), as shown in figure 3.

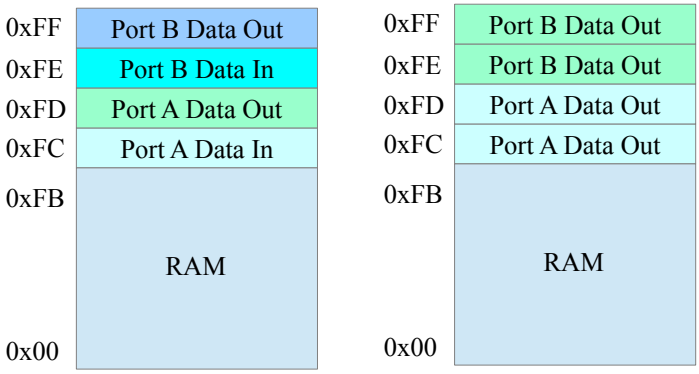

Figure 3 : Read (left) and Write (right) memory maps.

Read or writing to address range :

- 0x00 (0000 00 00) to 0xFC (1111 10 11) will access memory (RAM).
- 0xFC (1111 11 00) to 0xFD (1111 11 01) will access GPIO port A.
- 0xFE (1111 11 10) to 0xFF (1111 11 11) will access GPIO port B.

To identify the address range accessed by the processor the `addr_decoder` component is used, as shown in figure 4.

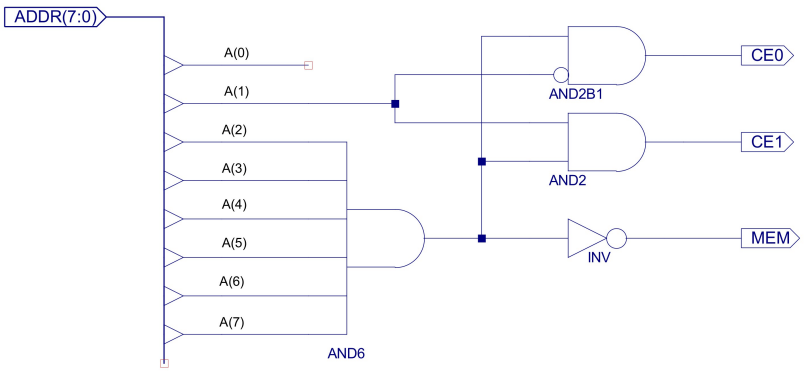

Figure 4 : Address decoder

This component performs the logical `AND` of the top six bits of the address bus. If any of these bits are a logic 0 the processor is accessing memory (RAM). If they are all logic 1 the processor is accessing a GPIO port. Address line `ADDR(1)` then selects between GPIO port `A` and `B`. Address line `ADDR(0)` is used within the GPIO port to control its internal register and multiplexer as shown in figure 5. The truth table for the address decoding logic is shown in figure 6.

The addition of the two GPIO ports means that multiple components now need to be connected to the processor's buses i.e. memory (RAM), GPIO port-A and port-B. Multiple components can be directly connected to the processors output signals

(control bus) and output buses (address and data-out) e.g. the data-out bus from the processor can be wired to the data-in ports on each component as there is only one component (processor) driving a signal onto these shared wires.

Note, driving a signal onto an input does take some electrical power, therefore, there is a limit to the number of components that can be connected to these shared buses etc. This limit is technology dependent, for this FPGA a rough rule of thumb is to try and limit any one output driver to 100 gate inputs.

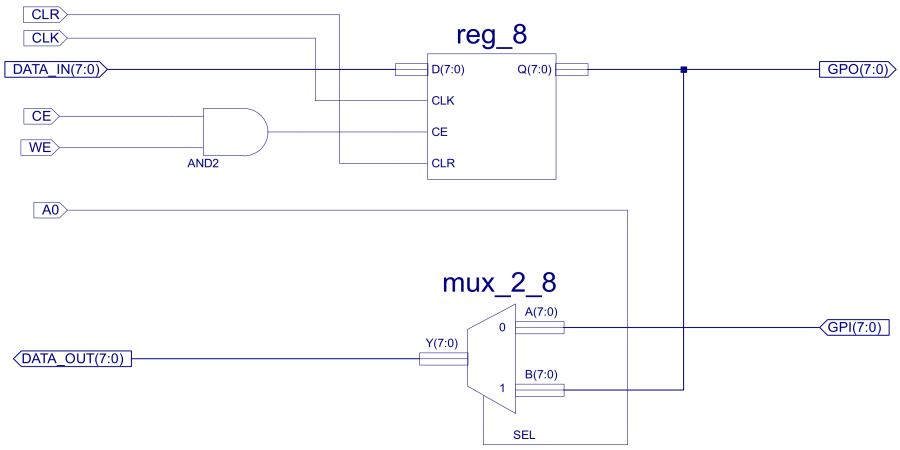

Figure 5 : GPIO port

Address Min	Addres Max	MEM	CE0	CE1
0x00	0xFB	1	0	0
0xFC	0xFD	0	1	0
0xFE	0xFF	0	0	1

Figure 6 : Address decoder truth table

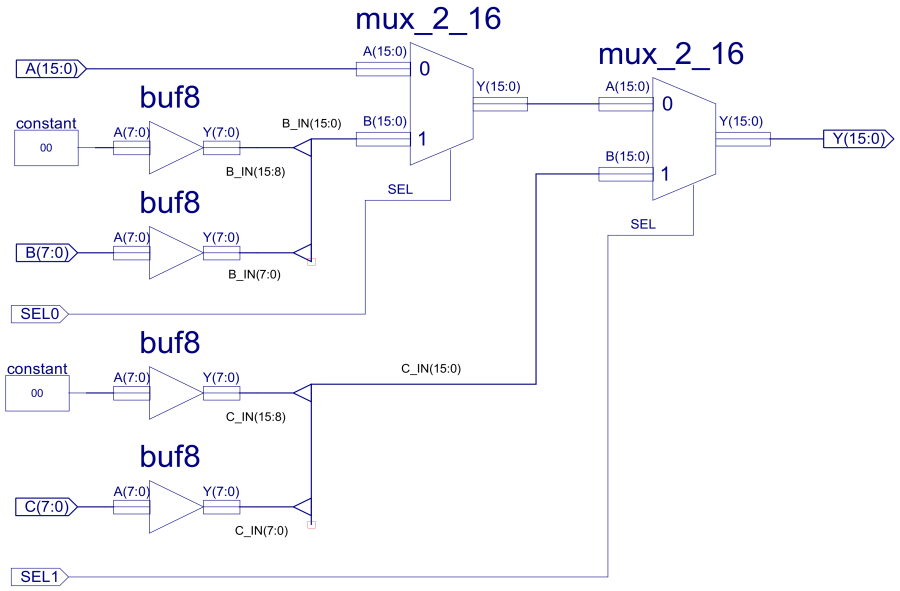

Figure 7 : Data multiplexer

However, a problem does occur when we try connect the data-out bus of each component onto the data-in port of the processor. The multiple signal drivers from these different buses i.e. memory (RAM), GPIO port-A and GPIO port-B, can **not** be directly connected to the same wires i.e. multiple readers is ok, multiple writers is bad. Therefore, the data multiplexer shown in figure 7 is used to switch the output buses of these components onto the share data-in bus. Again, the selection of which input source is connected to the processor is controlled by the address decoding logic.

Note, the logical 1's and 0's used to encode binary values on each wire are represented as different voltages. Connecting multiple signal drivers onto the same wire could result in a short circuit i.e. trying to drive a logic 1 and a logic 0 onto the same wire at the same time. This will result in large amounts of power being dissipated, which can lead to permanent damage to the silicon.

The simpleCPU is connected to the bug trap hardware via the `virtual_wires` component using GPIO port A, as shown in figures 1 and 8.

PORT A INPUT		PORT A OUTPUT	
Bit	Description	Bit	Description
0	Fire	0	Servo
1	Sensor 1	1	LED
2	Sensor 2	2	NU
3	Mode	3	NU
4	OSC	4	NU
5	NU	5	NU
6	NU	6	NU
7	NU	7	NU

Figure 8 : GPIO pins

The pseudo below can now be implemented in software, as shown in figure 9.

```
DESCRIPTION                ASSEMBLY CODE
IF (FIRE)                  start:
THEN                            load 0xFC
  TURN OFF LED                  and 0x01
  CLOSE TRAP                    jumpz fire
ELSE                       reset:
  TURN ON LED                   move 0x02
  OPEN TRAP                     store 0xFC
ENDIF                           jumpu start
                           fire:
                                move 0x01
                                store 0xFC
                                jumpu start
```

Figure 9 : test code

Task : examine the test code in figure 9, make sure you understand how this code implements the required control rules described in pseudo code.

Hint, refer back to figure 3, what is address 0xFC mapped to? What is connected to GPIO port A input bit 0? What is connected to GPIO port A output bit 0 and bit 1?

To assemble this program i.e. generate the machine code that will be loaded into memory, a python based assembler has been written : `simpleCPUv1a_as.py`, this code and supporting files can be downloaded for the the VLE: `assembler.zip`. Using your preferred browser download this zip file, save and uncompress in the current project directory. This will create a subdirectory called `assembler`.

Next, launch a command prompt, click on the start button and select :

 -> `cmd`

Within the new command prompt change to the current project directory using the `cd` command. Alternatively, type "`cd `" at the command line, then within a file browser drag the current project directory folder into the command prompt, this will automatically add the correct path string. Finally `cd` into the previously created `assembler` directory, then into the `windows` subdirectory.

Note, before changing directory you may need to first change to the local hard-disk by typing "`c:`" at the command line. You will need a space between the `cd` command and the path, also within the file browser you will need to go up one folder level so that you can drag the project folder.

Next, launch the text editor Notepad++ (or your preferred text editor)

 -> `Notepad++`

Enter the assembler code shown in figure 9, save to the file: `test.asm`, in the previously created assembler subdirectory. The command line parameters supported by this assembler are shown below:

```
Usage: simpleCPUv1a_as.py -i <input_file.asm>
                          -o <output_file>
                          -a <address_offset>
                          -t <input_file_type>
```

To assemble this program at the command prompt enter :

```
simpleCPUv1a_as.py -i test -o test
```

Note, the file extensions are automatically added by the assembler. This will generate the following output files:

- `tmp.asm` : temporary assembly code file, comments and additional white spaces removed. Labels replaced with their numerical values.
- `test.asc` : ASCII hex file for EPROM programmer. Not used in this lab.
- `test.dat` : plain text file, instruction address (decimal) followed by

machine code (binary). Useful for debugging

- `test.mem` : plain text file, instruction address (hex) followed by machine code (hex) reversed nibble format i.e. most significant nibble first. Used by linker to generate memory image file.

These files can not be directly loaded into the FPGA memory. To allocate this machine code to specific memory blocks a linker is used. In general a linker takes one or more assembled object files and libraries to produce a final executable program. As we are using a very simple assembler, this linker could be more accurately referred to as a loader. To convert the machine code generated by the assembler into a format that can be loaded into memory at the command prompt enter :

```
simpleCPUv1a_ld.py -i test
```

The linker reads the file: `test.mem`, producing the VHDL file `memory.vhd` that will be used during the synthesis process to initialise the computer's memory. To add this file to the current project left click on the `Project` pull down within the project navigator window :

```
Project -> Add Source
```

This will open the Add Source window. Browse to the project directory and open the file : `memory.vhd`.

To verify the operation of this circuit a VHDL test bench has already been added to this project. Click on the 'Simulation' radio button, ensuring 'Behavioral' simulation is selected, as previously described. Next, click (highlight) on the testbench file `bug_trap_controller_TB`, this will update the 'Processes for' source window, then double left click on 'Simulate Behavioural Model' to launch the simulation.

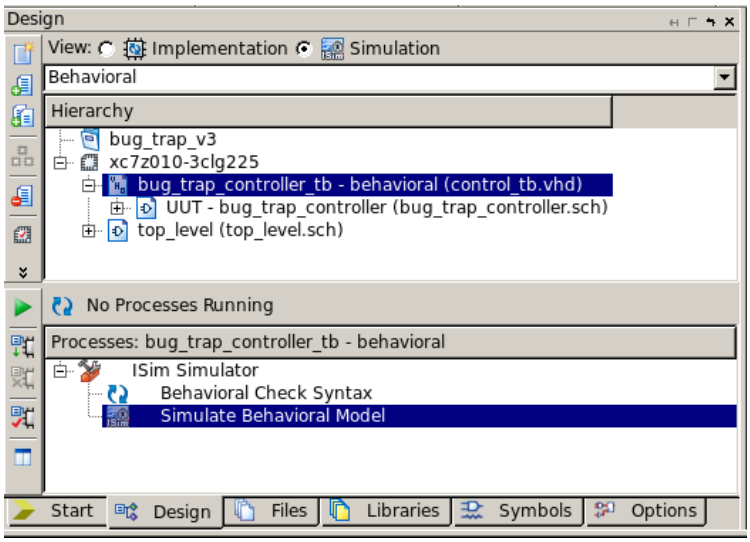

Figure 10: launching decoder simulation

The VHDL simulator allows the operation of the circuit to be checked by examining its waveform timing diagrams shown in figure 11 and 12. This test bench repeatedly pulses the `FIRE` signal low, simulating the `RED` button being pressed. The software

detects this by examining bit-0 of input port A, if low it updates the `SERVO` and `LED` outputs as described by the pseudo code in figure 9.

To automatic load the required simulation signals as shown in figure 12, a waveform configurations script (`.wcfg` file) has been included in the project directory. Within the `ISim` windows left click on:

```
File -> Open
```

Select the `test.wcfg` file and click `Open`. Next, click on the reset icon, as shown in figure 11, then repeatedly click on the `step` icon to simulate this hardware and software for approximately 15 us. To view the complete waveform click on the `zoom-fit` icon.

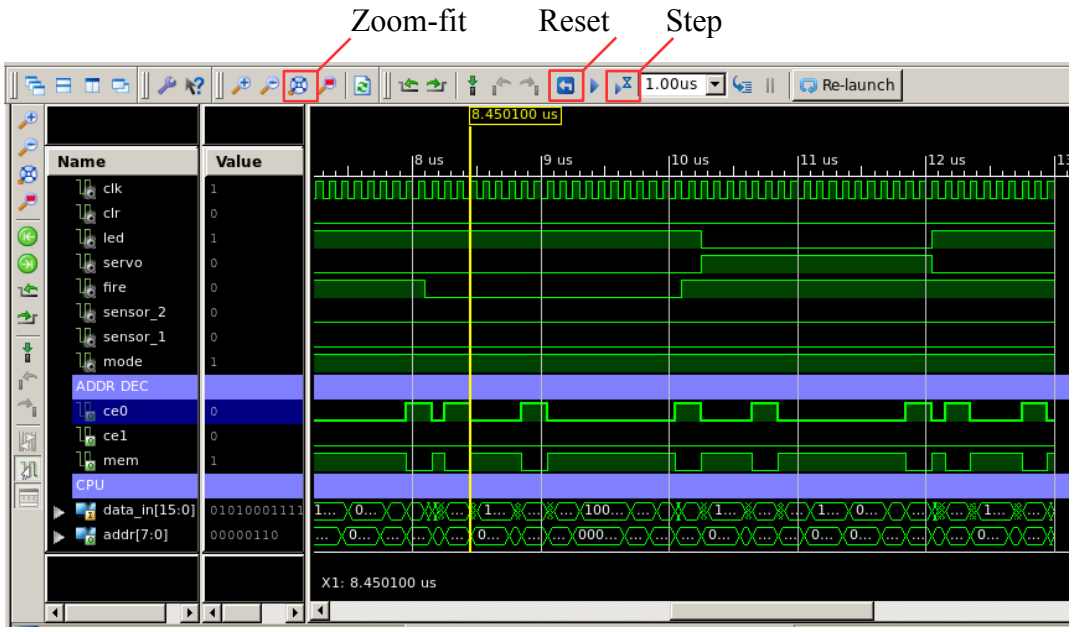

Figure 11: GPIO simulation

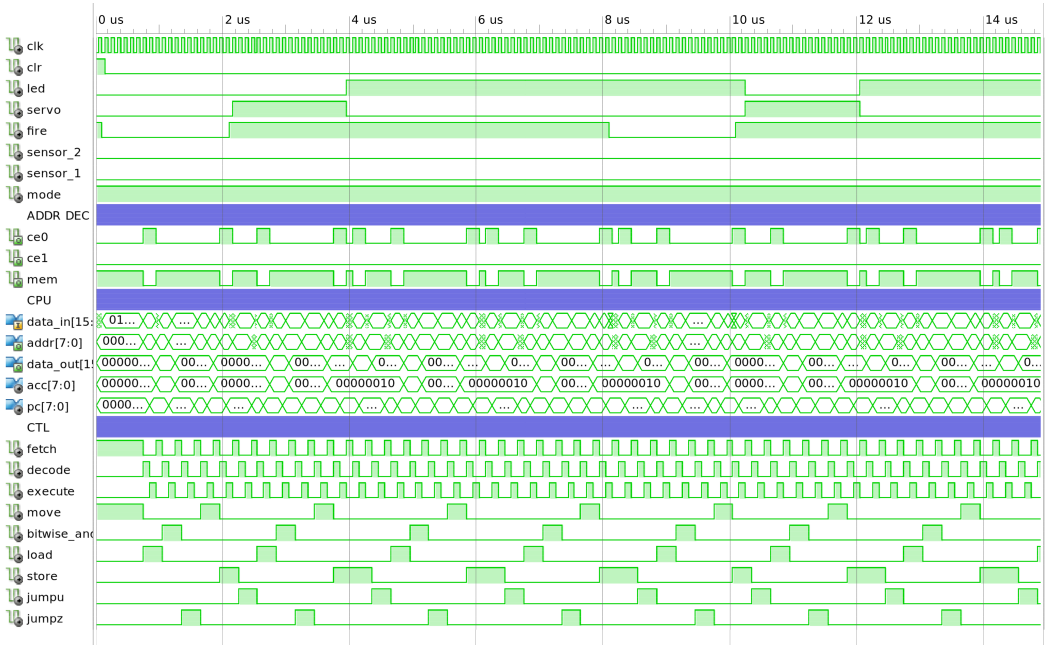

Figure 12: full GPIO simulation

Task : examine the simulation wave-forms, identify when each instruction is executed and when the GPIO port is accessed.

To generate the `bit` file that will be used to configure the FPGA click on the 'Implementation' radio button and select the top level design file `top_level.sch` within the Hierarchy panel. This will update the Processes panel below, double click on the "Generate Programming File" icon, as shown in figure 13.

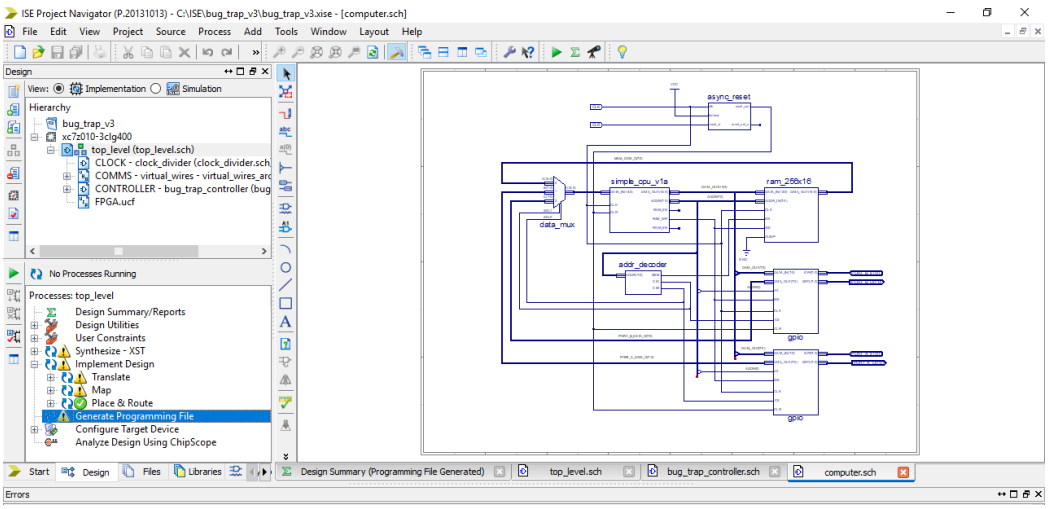

Figure 13 : generating bit file

Plug in the power supply module and connect the USB cable, then configure the FPGA as described in Laboratory script 1, pages 9 to 11 i.e. using the `IMPACT` download tools and the bit file: `top_level.bit`. If all has gone well you should now be able to control the net position and LED status using the RED push button.

Note, if working correctly when the RED push switch is pressed the RED LED is turned off and the trap is closed, otherwise the RED LED is turned on and the trap opened.

CPUSIM

```
MACRO clr
  move 0
ENDM

MACRO set
  move 0xFF
ENDM
```

M4

```
define( clr,`move 0x00')

define( set,`move 0xFF')
```

Figure 14 : CLR, SET macros

Task 2

Most assemblers use a pre-processor to define constants and macros. Included within the downloaded zip file is the M4 pre-processor :

https://en.wikipedia.org/wiki/M4_(computer_language)

If you look through some of the examples on the wiki-page you can see that this pre-processor is actually quite powerful i.e. the power and confusion of recursive

programming. Therefore, we can now define more complex macros than those implemented in CPUSim. The syntax of the M4 pre-processor is a little different to that used in CPUSim, consider the `CLR` or `SET` macros defined in laboratory 7 shown in figure 14.

Note, The quotes for the M4 pre-processor are a matched pair of single quotes "`" and "'", they are different.

You can again pass parameters to a macro, you can also now do some pre-processing within the macro using the `eval` function, supported operators are shown in figure 15.

+ -	unary plus and minus
**	exponent
* / %	multiplication, division, modulo
+ -	addition and subtraction
<< >>	shift up or down
== != < <= >= >	relational
!	logical not (converts non-zero to 0, 0 to 1)
~	bitwise not
&	bitwise and
^	bitwise exclusive or
|	bitwise or
&&	logical and
||	logical or

Figure 15 : eval operators

Consider the new bitwise-OR macro (`ora`) below. This performs the opposite function to the the bitwise-AND i.e. the bitwise-AND is used to clear bit positions, the bitwise-OR can be used to set bit positions.

```
define( ora,`and eval($1 ^ 255)
add $1')
```

We can call this macro in our assembly language program :

```
ORIGINAL          EVAL          FINAL
start:            XOR           start:
  load 0xFC       11110000        load 0xFC
  ora 0xF0        11111111        and 0x0F
  store 0xFC      00001111        add 0xF0
                                  store 0xFC
```

Note, parameters are positional in calling macro, labelled `$1`, `$2`, `$3` etc. If we need to pass two parameters to the `ora` macro we would write this in the program as `ora 0xAA 0xBB`, then within the macro the variable `$1=0xAA` and `$2=0xBB`.

Task : make sure you understand how the bitwise-OR macro has been implemented using `AND` and `ADD` instructions?

Hint, what does XORing the passed parameter 0xF0 ($1) do to its value. What is clearing a register and then adding a value to it equivalent to?
These macros are stored in the file `simpleCPUv1a.m4` and can be passed to the M4 pre-processor along with the assembly language text file, as shown below:

```
m4 simpleCPUv1a.m4 test.asm > code.asm
```

The pre-processor replaces the macro names with the specified code, saving the resulting text to the file `code.asm`. To illustrate this in practice consider the original test program shown in figure 9. This can be rewritten using macros as shown in figure 16.

```
M4
define(GPIO, `0xFC')

define(turnOnLED, `load eval(GPIO+1)
and 0xFD
add 2
store GPIO')

define(turnOffLED, `load eval(GPIO+1)
and 0xFD
store GPIO')

define(closeTrap, `load eval(GPIO+1)
and 0xFE
add 1
store GPIO')

define(openTrap, `load eval(GPIO+1)
and 0xFE
store GPIO')

define(buttonPressed, `load GPIO
and 0x01
jumpz $1')
```

```
DESCRIPTION              ASSEMBLY CODE
IF (FIRE)                start:
THEN                          buttonPressed fire
     TURN ON LED         reset:
     CLOSE TRAP                turnOffLED
ELSE                           openTrap
     TURN OFF LED              jumpu start
     OPEN TRAP           fire:
ENDIF                          turnOnLED
                               closeTrap
                               jumpu start
```

Figure 16 : test code

Using macros significantly improves the readability of the program. Rather than using

the value of the GPIO's memory mapped address: 0xFC, we can now use the symbol `GPIO`. Each identified system function is then implemented as a macro, abstracting away from the underlying implementation.

Task : the file `simpleCPUv1a.m4` can be downloaded from the VLE. Save this file to the previously created `windows` subdirectory . Enter the assembly code shown in 16, saving it to the file `test.asm`.

To generate the memory image file `memory.vhd`, at the command prompt run the following commands:

```
m4 simpleCPUv1a.m4 test.asm > code.asm
simpleCPUv1a_as.py -i code -o code
simpleCPUv1a_ld.py -i code
```

Regenerate the configuration bit file, upload into the FPGA and confirm that the trap operates correctly.

Note, to help confirm that you have downloaded a new bit file I have reversed the LEDs state, in this new version the RED LED is turn on when the RED push switch is pressed.

Task 3

In previous bug traps time delays have been used to control the servo's operations e.g. stamping (raise and lower the net every second) and a four second delay. These functions can be implemented in software using software delays based on the example code shown in figure 17.

PESUDO CODE

```
delay:
  lowCount = 0
  highCount = 0
  WHILE highCount >0
  LOOP
    WHILE lowCount >0
      lowCount = lowCount+1
    END LOOP
    highCount = highCount+1
  END LOOP
```

ASSEMBLER CODE

```
delay:
  move 0x00
  store lowCount
  store highCount
innerLoop:
  load lowCount
  add 0x01
  store lowCount
  jumpnz innerLoop
outerLoop:
  load highCount
  add 0x01
  store highCount
  jumpnz innerLoop
stop:
  jumpu stop
```

Figure 17 : software delay

This code uses two variables stored in memory `lowCount` and `highCount` to implement a 16bit counter. **Note**, the addresses of these symbolic names can be hard-coded e.g. replaced with the values `0xE0` and `0xE1`. Alternatively, they can be

defined using a macro e.g. as the GPIO address in the previous macros. Initially these variables are zeroed. The `lowCount` is repeatedly incremented until it overflows i.e. 0xFF+1=0x00. The `highCount` is then incremented and the process repeated, the delay finishing when an overflow is detected in the `highCount` variable.

Task : if the processor's system clock is running at 10MHz and each instruction takes three clock cycles to complete i.e. one clock cycle for each instruction micro-instructions phase (FDE), how long will it take to complete this delay? What size of delay counter will be required to implement a 4 second delay function?

Max Delay = ((start_code) + ((innerLoop_code × 256) + outerLoop_delay) × 256)
× period × 3

If you would like to confirm your solution refer to Appendix A.

Task : define a new macro `stampOnBug`. This macro should close the trap for approximately 0.75 seconds, then open the trap for approximately 0.75 seconds, before returning back to the main control loop. Update the test code to match that shown in figure 18, regenerate the configuration bit file, upload into the FPGA and confirm that the trap operates correctly.

```
DESCRIPTION                ASSEMBLY CODE
IF (FIRE)                  start:
THEN                            buttonPressed fire
     TURN OFF LED          reset:
     STAMP ON BUG                turnOffLED
ELSE                             openTrap
     TURN ON LED                 jumpu start
     OPEN TRAP             fire:
ENDIF                            turnOnLED
                                 stampOnBug
                                 jumpu start
```

Figure 18 : test code

Hints, you may wish to define a new macro `longDelay` to implement the software delay loop. Variables used in this code can be hard coded e.g. `lowCount` and `highCount` could be assigned the addresses 0xE0 and 0xE1 etc. The macro `longDelay` will be called twice i.e. delays when trap is open/closed, therefore, you will also need to consider how you will generate unique labels e.g. you can't have two instructions with the same label e.g. `innerLoop` etc. If you would like to confirm your solution refer to Appendix B.

Task : if the processor is performing the `stampOnBug` function what will happen if the `fire` button is released i.e. how long will it take for the software to respond to this change of input state? How could this program be modified to improve responsiveness? **Hint**, where could you perform the button test?

Note, this is a key consideration when moving from a hardware based system to a software based one. Hardware by its nature is a parallel processing based one i.e. all logic gates are operating at the same time, whereas software is sequential i.e. one

instruction is executed at a time. Therefore, the speed at which the processor can access input and output ports will have a significant impact on system performance.

Task 4

To test your programming skills re-implement the control rules from laboratory 5, the pseudo code description is shown in figure 19. The stamping function could be implemented using software delay loops from the previous task. Alternatively, to reduce code size the `OSC` input from the hardware clock divider can be used. This signal can be accessed on input port bit 4, you will need to detect the transitions on this signal and update the servo accordingly. If you would like to confirm your answer one possible solution is shown in Appendix C.

Hint, to test the state of each input use a bit-wise `AND` instruction to test for zero and `SUB` instruction to implement a compare function i.e. is the `ACC` equal to a particular value. These combined with the appropriate conditional jump instructions will implement the required `IF-THEN-ELSE` tree. Remember that the two push switches and sensors are active low i.e. they will output a logic 0 when pushed and a logic 1 when not.

Note, this program is an example of polling, or polled operations. This is where a program explicitly tests the state of an I/O device e.g. switches, then updates its internal state, or external actuators e.g. LED, servo etc. This technique is therefore sometimes called software-driven I/O.

Task : rewrite the assembly code used to implement the control rules shown in figure 19 using macros, defining new macros that will test if the size of the bug is either small or large.

```
DESCRIPTION
IF (AUTOMATIC)
THEN
  TURN OFF LED
  IF (LARGE)
  THEN
    STAMP ON BUG
  ELSIF (SMALL)
  THEN
    CLOSE TRAP
  ELSE
    OPEN TRAP
  ENDIF
ELSE
  TURN ON LED
  IF (FIRE)
  THEN
    CLOSE TRAP
  ELSE
    OPEN TRAP
  ENDIF
ENDIF
```

Figure 19 : Control rules

Task 5

The assembler and linker are both written in python, not the most robust pieces of code i've ever written, but they do their job :). Have a look at this code can you identify what each section of code does? Could you modify the assembler to add a new instruction?

Hint, you will need to do this for the open assessment.

For more information on this code refer to:

http://www.simplecpudesign.com/simple_cpu_v1a_assembler/index.html

Summary

One of the main advantages of a general purpose processor is its flexibility. System functions that would be difficult or expensive to implement in application specific hardware can be emulated in software. However, this flexibility does come at the cost of processing performance. Instructions are processed using the fetch-decode-execute cycle, therefore, for two out of these three phases the processor is not actually processing data i.e. crunching numbers. The fetch-decode portion of this state-machine implements “admin” / “house keeping” operations, that allow this flexibility i.e. needed to setup / configure the processor's architecture to process an instruction. When a system is hard-wired i.e. we remove this flexibility and implement the system in custom hardware, we remove (reduce) the need for these types of operations, such that the processing elements just process data. Minimising these types of “admin” operations significantly increases processing performance. When designing a system we tend to take a hardware / software co-design approach, using cheap / flexible software based solutions where possible, but where required e.g. to meet timing requirements, we will add additional hardware units / custom hardware accelerators. Therefore, you tend not to see true general purpose processors i.e. a processor that is good at everything, rather application specific processor's with additional hardware units to match their intended application domain.

Appendix A : delay calculation

Max Delay = ((start_code) + ((innerLoop_code × 256) + outerLoop_delay) × 256)
× period × 3

= (3 + ((4 × 256) + 4) × 256) × 100^{-9} × 3 = 0.0789 seconds

To implement a four second delay we could need to call this software delay loop:

Four Second Delay = 4 ÷ 0.0789 = 50 times

This could be implemented as a third outer FOR loop:

```
FOR I IN RANGE 0 TO 50
  delay()
END LOOP
```

Appendix B : stampOnBug macro

The variables LOW_COUNT and HIGH_COUNT are assigned fixed addressing within the M4 macro script. This delay function is called within the stampOnBug macro twice, therefore, it is passed the Ids “1” and “2” to ensure unique label names. The lonDelay macro is called 10 times using the LOOP_COUNT variable to produce the 0.75 second delay. Trap controlled by the openTrap and closeTrap macros.

```
define(LOW_COUNT, `0xE0')
define(HIGH_COUNT, `0xE1')
define(LOOP_COUNT, `0xE2')

define(longDelay, `move 0
store LOW_COUNT
store HIGH_COUNT

innerLoop$1:
load LOW_COUNT
add 1
store LOW_COUNT
jumpnz innerLoop$1

outerLoop$1:
load HIGH_COUNT
add 1
store HIGH_COUNT
jumpnz innerLoop$1'
)

define(stampOnBug, `closeTrap
move 10
store LOOP_COUNT

closedWait:
longDelay( 1 )
load LOOP_COUNT
sub 1
store LOOP_COUNT
jumpnz closedWait

openTrap
move 10
store LOOP_COUNT

openWait:
longDelay( 2 )
load LOOP_COUNT
sub 1
store LOOP_COUNT
jumpnz openWait'
)
```

Appendix C : Improved control rules

```
DESCRIPTION
IF (AUTOMATIC)
THEN
  TURN OFF LED
  IF (LARGE)
  THEN
    STAMP ON BUG
  ELSIF (SMALL)
  THEN
    CLOSE TRAP
  ELSE
    OPEN TRAP
  ENDIF
ELSE
  TURN ON LED
  IF (FIRE)
  THEN
    CLOSE TRAP
  ELSE
    OPEN TRAP
  ENDIF
ENDIF

BIT     FUNCTION
0   1   FIRE
1   2   SENSOR1
2   4   SENSOR1
3   8   MODE
4   16  OSC
```

```
ASSEMBLY CODE
start:
  load 0xFC
  and 0x08
  jumpz manual
auto:
  load 0xFC
  and 0x06
  jumpz large
  sub 0x06
  jumpz auto
small:
  move 0x03
  store 0xFC
  jumpu start
large:
  move 0x03
  store 0xFC
  load 0xFC
  and 0x06
  jumpnz start

waitLow:
  load 0xFC
  and 0x10
  jumpz waitLow

   move 0x02
   store 0xFC
waitHigh:
  load 0xFC
  and 0x10
  sub 0x10
  jumpz waitHigh
  jump large
manual:
  load 0xFC
  and 0x01
  jumpz fire
reset:
  move 0x02
  store 0xFC
  jumpu start
fire:
  move 0x03
  store 0xFC
  jumpu start
```