SYS1 Laboratory 2 : Logic gates – hardwired controllers

The aim of this lab is to demonstrate the operation of basic logic gates, their functions and how they can be combined to implement different functions i.e. the fundamental building blocks of any computer. Each time a computer executes an instruction it has to "reconfigure" its hardware, selecting the correct data paths to access the required operands (data) and functional units to process this data. These control circuits are commonly implemented using a hardwired controller i.e. a combinatorial logic circuit specifically designed for a particular instruction set. At the end of this practical you will understand how to:

- Define combinatorial logic circuits using schematic capture entry.
- Design simple hardwired controllers from high level pseudo code descriptions.
- Implement combinatorial logic circuits to process input sensor data to control output actuators.
- Understand how multiplexer components are used within a processor.

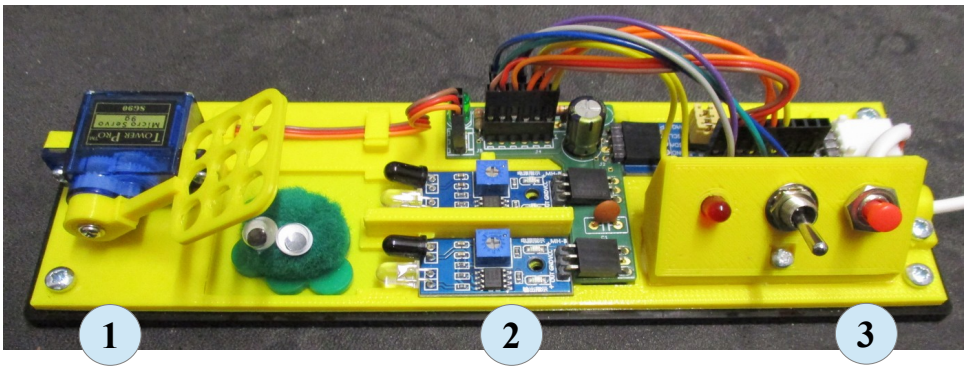


Figure 1 : Bug trap

The University is overrun with cockroaches (bugs) you are required to design a hardwired controller for an automatic bug trap. The bug trap hardware has already been designed as shown in figure 1. This hardware has three main components :

1) Net: a high powered servo motor connected to a square net via a connecting arm. When energised the servo pushes the net down into the closed position, trapping the cockroach. Otherwise it lifts the net up into the open position.
2) Base: this section contains two infra-red LED (illuminated) sensor modules i.e. front and rear sensor, that detect when a cockroach is in the trap. These are interfaced to the FPGA via an I2C GPIO expander.
3) Control panel: user console containing a **RED** status LED, a toggle switch and a **RED** push switch.

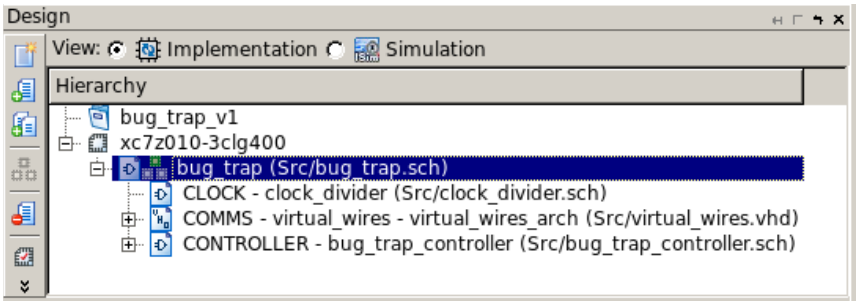
To allow you to control this hardware an ISE project called `bug_trap_v1` has been created and can be downloaded from the module's VLE page. Using your preferred web browser download this zip file to `C:\Users\<USER NAME>`. Right click on this file selecting 'Extract all…' to unzip it. To start this practical click on the start button and select the ISE project navigator.

-> `Xilinx 64-bit 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_v1.xise`. Next, within the Hierarchy window double click on the top level schematic `bug_trap.sch`, as shown below:



This will open the schematic shown in figure 2.

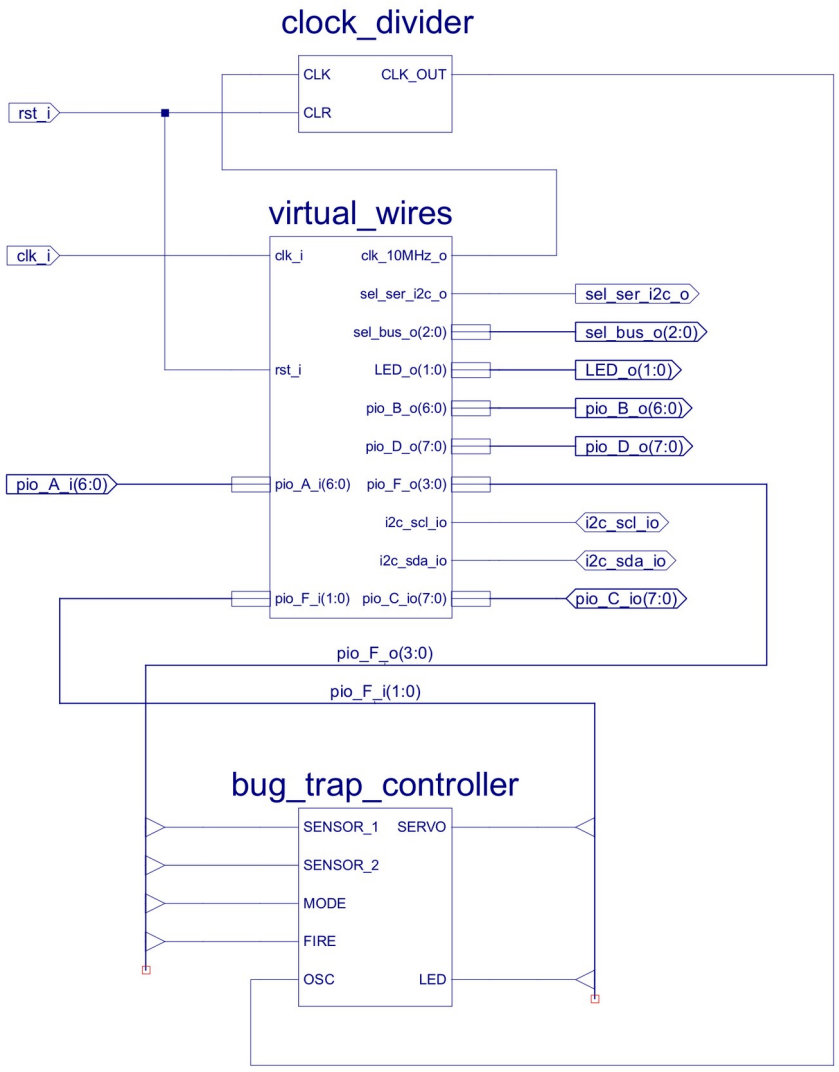
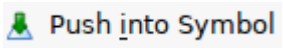


Figure 2 : top level schematic, bug_trap.sch

This schematic is made up of three components:

- Bug_Trap_Controller : a combinatorial logic circuit that you will be incrementally developing during this laboratory, processing the input sensor data to control output actuators. This component's interface contains the signals:
  - Sensor 1 : front sensor, infra-red module 1=empty, 0=bug present.
  - Sensor 2 : rear sensor, infra-red module 1=empty, 0=bug present.
  - Mode : toggle switch, 0=down position, 1=up position.
  - Fire : push switch, 0=pressed, 1=not pressed
  - OSC : clock oscillator, frequency approximately 1Hz
  - Servo : servo motor controlling net position, 0=up, 1=down
  - LED : status display, 0=off, 1=on
- Virtual_Wires : the seven signals controlling the bug trap (listed above) are not directly connected to the FPGA board as individual wires, rather signals are transferred between the FPGA board and the bug trap hardware using a serial Inter IC Communications (I2C) Bus, significantly reducing the number of wires in the white connecting cable. **Note**, the reason for this abstraction was due to limited IO lines, this is not visible to the bug trap controller. In addition to implementing this communications link this component also updates the seven segment LED display with the status of these signals.
- Clock_Divider : the FPGA board has an on board oscillator producing a 10MHz square wave clock. This is divided down to a slower frequency to produce a 1Hz clock on the CLK_OUT (`OSC`) pin. This signal will be used later to control the servo motor's position.

To test that the bug trap and the FPGA board are working correctly a simple controller has already been implemented. To open this schematic single left click on the `Bug_Trap_Controller` component, this will highlight it **RED**, then on the side toolbar click on the “push into” symbol icon:



This will open the `Bug_Trap_Controller` schematic shown in figure 3. In this circuit the **RED** push switch is connected through an inverter to the servo and the front panel LED. When this switch is pressed a logic 0 is generated on the FIRE signal, this is inverted to produce a logic 1 which will move the servo into the down position and illuminate the **RED** LED.

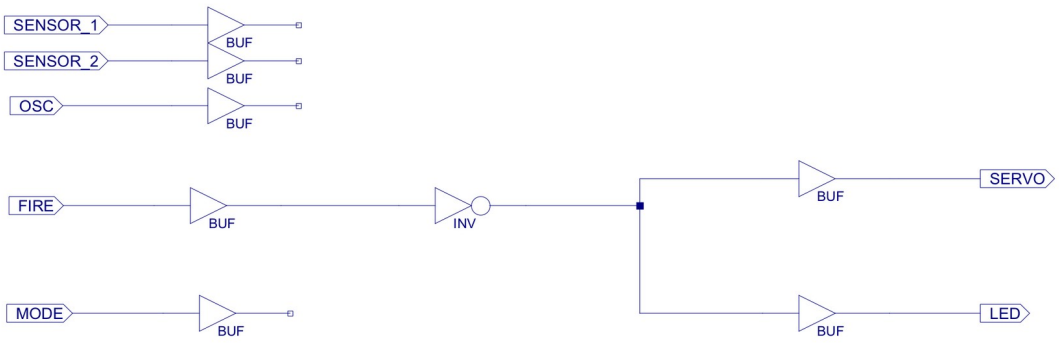


Figure 3 : top level schematic, bug_trap.sch

Left click on the top level schematic (highlight), then double click on "Generate Programming File" to produce the configuration bit file, as shown in figure 4. Plug in the power supply module and connect the USB cable, then configure the FPGA as described in Laboratory script 1, page 10, but using the `bug_trap.bit` file.

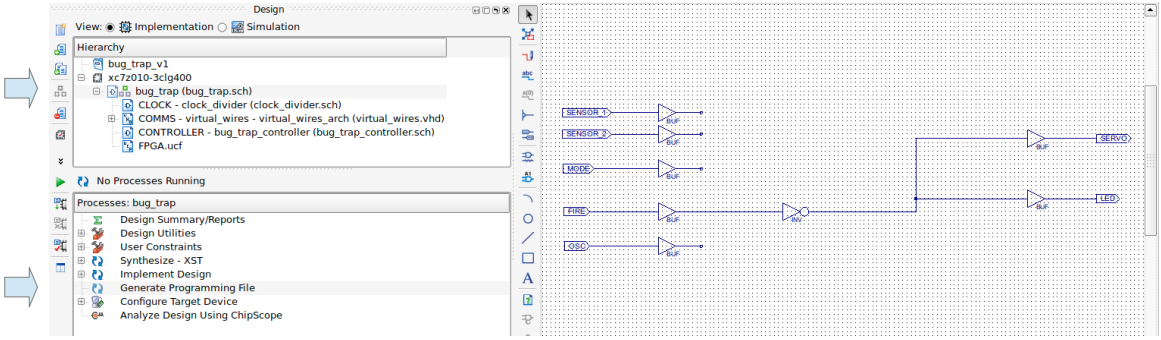


Figure 4 : generate programming file.

When configured the seven segment LED display will count down from 9 to 0, then the servo arm will initialise into the UP position.

Task : press the **RED** push switch on the front panel, if the hardware is connected correctly the net will move into the down position and the LED will be illuminated. The seven segment LED display on the interface PCB should also change as the sensors and buttons are activated.

You may need to adjust the net position. Use a small Phillips screwdriver to stop servo rotating

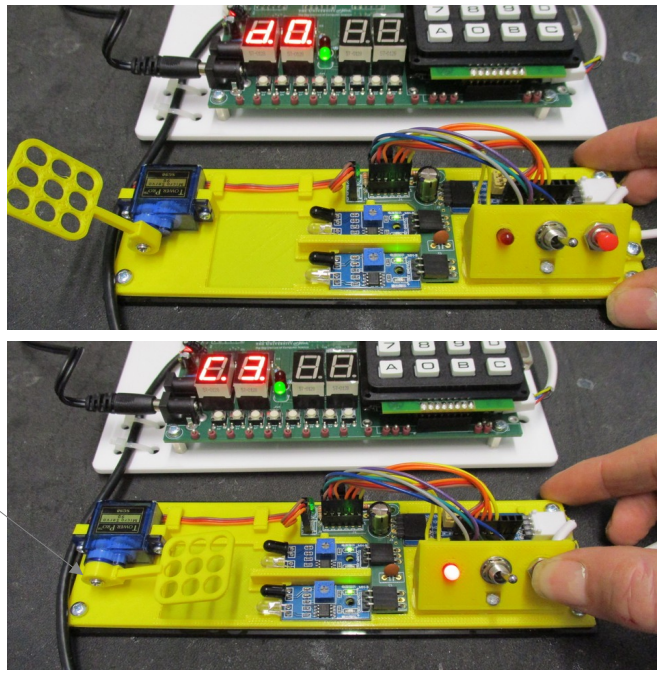


Figure 5 : bug trap operation

Task 1

The data transferred across the I2C bus can be viewed using the oscilloscope. Turn on the scope as described in the Laboratory 1 and connect it to the test point shown in figure 6. Set the vertical voltage range to 2V per division and the horizontal time base to 50us per division, a typical packet transfer is shown in figure 7.

Tip, if you do not see the waveform in figure 7 try pressing the '`Default Setup`' and '`Auto-scale`' buttons. You will need to reposition the trace using the small horizontal position knob.

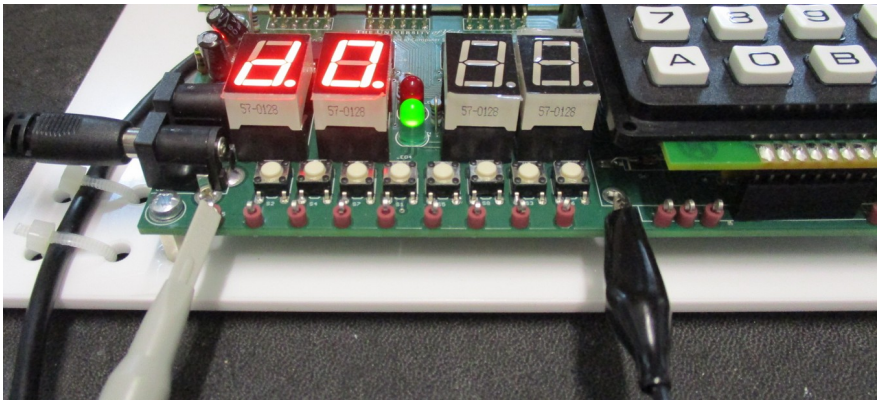


Figure 6 : scope test points.

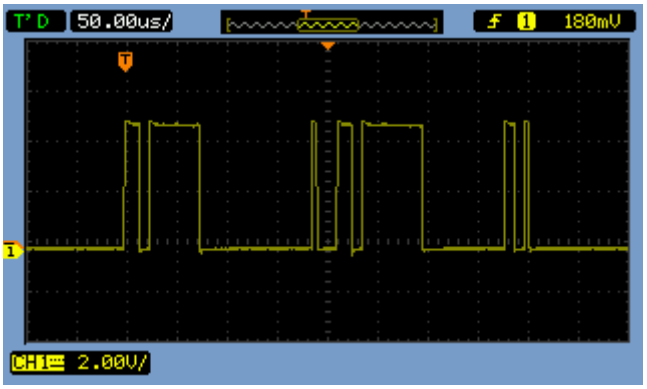


Figure 7 : I2C data packets

These I2C data packets are used by the FPGA to read the state of the various input sensors and to control the output actuators.

Task : how often are these packets transferred i.e. how quickly is the bug trap hardware updated? **Tip**, you will need to increase the timebase on the scope (ms).

Note, you should see quick changes in the data packet bit values when the push button is pressed (not persistent). We will not be using the I2C bus directly in these laboratories, but we will be looking at different examples of serial data buses in later lectures i.e. buses that transfer data one bit at a time across a single wire. If you would like more background information on the I2C bus a pdf document is available on the VLE in `files.zip`.

From these packets the state of the four input sensors is read and displayed on the seven segment display as a hexadecimal value, as shown in figure 8.

Task : what bit of this four-bit value is each input sensor connected to? Activate each infra-red sensor and switch in turn, record the displayed hexadecimal values (base 16), from this you can determine what bit represents what sensor.

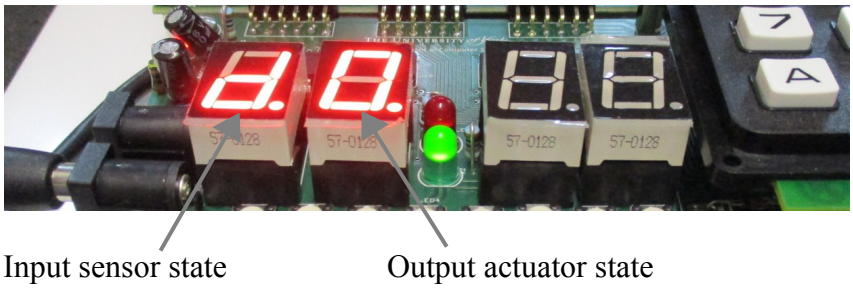


Figure 8 : input and output states

Hint, the RED push button is connected to bit 0. As previously discussed when pressed this bit will be set to a logic zero. Therefore, depending on the state of the other three bits the display will be set to the value 0, 2, 4, 6, 8, A, C or E and when it is not pressed the value 1, 3, 5, 7, 9, B, D, or F. A hexadecimal to binary conversion table is shown in figure 9. If you are really, really, really stuck, refer to Appendix A.

IMPORTANT : make you understand this before proceeding.

Note, if the infra-red sensors do not change the state of the LED display you will need adjusted their sensitivity. This can be done by turning the blue variable resistor on each module using a small screwdriver (desk draw). For more information refer to Appendix B. If you are not sure how to do this do ask for assistance.

Dec	Hex	Bin	Dec	Hex	Bin
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

Figure 9 : Decimal to Hexadecimal conversion tables

Task 2

The current version of the `Bug_Trap_Controller` implements the following control rules / pseudo code, only using the **RED** push switch i.e. the `FIRE` button:

```
DESCRIPTION                    LOGIC
IF FIRE                        SERVO = NOT FIRE
THEN                           LED   = NOT FIRE
     CLOSE TRAP
     TURN ON LED
ELSE
     OPEN TRAP
     TURN OFF LED
```

Remember, the FIRE switch is active LOW, the SERVO and LED are active HIGH. To allow the trap to work at night i.e. without human assistance, the front and back infra-red sensors (on base) need to be incorporated into these control rules.

In the Bug_Trap_Controller component SENSOR_1 is the rear sensor and SENSOR_2 is the front sensor as shown in figure 10. The control rules can be rewritten as:

```
DESCRIPTION                    LOGIC
IF SENSOR_1 OR SENSOR_2        SERVO = ( (NOT SENSOR_1) OR
THEN                                     (NOT SENSOR_2) )
     CLOSE TRAP
     TURN ON LED               LED   = ( (NOT SENSOR_1) OR
ELSE                                     (NOT SENSOR_2) )
     OPEN TRAP
     TURN OFF LED
```

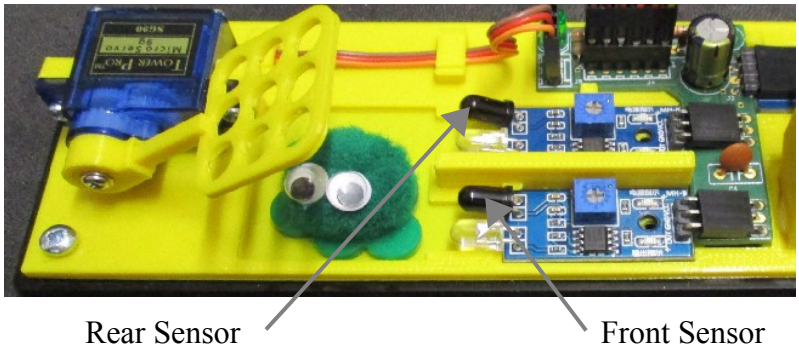


Figure 10 : front and rear infra-red sensors

These control rules could be implemented using an OR gate, as shown in figure 11.

Task : update the schematic bug_trap_controller.sch to match the new circuit diagram shown in figure 11. **Tip**, you will need to delete some of the connecting wires and add an additional INV and a OR2 component from the LOGIC component category.

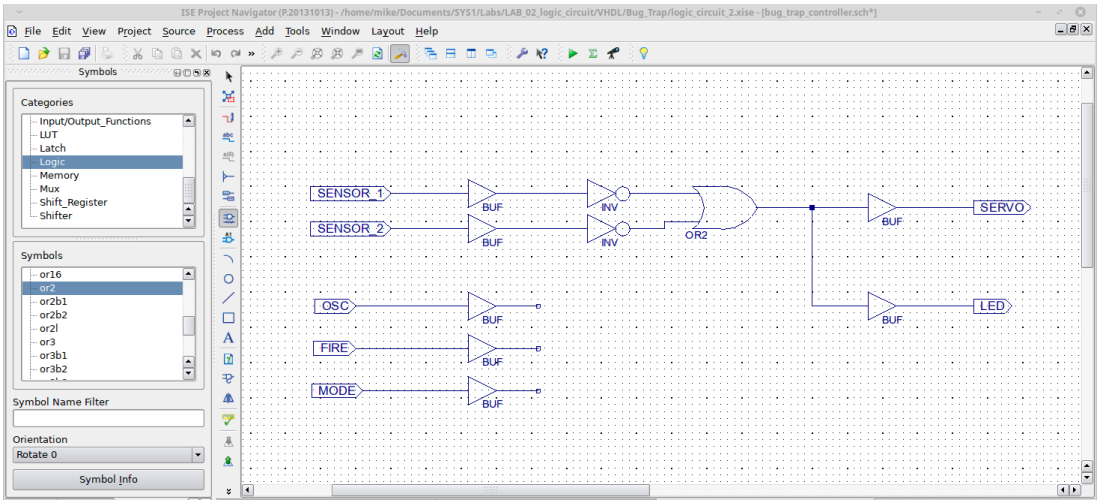


Figure 11 : front and rear infra-red sensors

This circuit can also be represented as the truth table shown in figure 12.

Sensor 1	Sensor 2	NOT Sensor 1	NOT Sensor 2	Servo	LED
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

Figure 12 : truth table

Task : what single logic gate can be used to implement the same function i.e. replace the two `INV` and `OR2` components?

When you have finished entering this circuit left click on the save icon  in the top left toolbar in the main window.

Note, logic gates have inputs (on the left) and outputs (on the right) as shown in figure 13. When connecting these logic gates together remember these following rules.

- Input connected to an Input -> Ok, but nothing is producing a signal
- Output connected to an Input -> Good, signals processed by a logic gate
- Output connected to an Output -> **VERY** bad, two outputs could try to drive different signals onto the same wire.

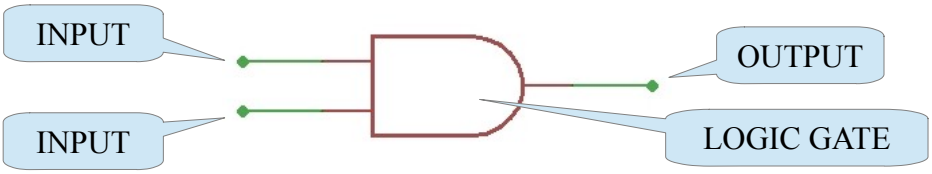


Figure 13: AND gate, inputs and outputs

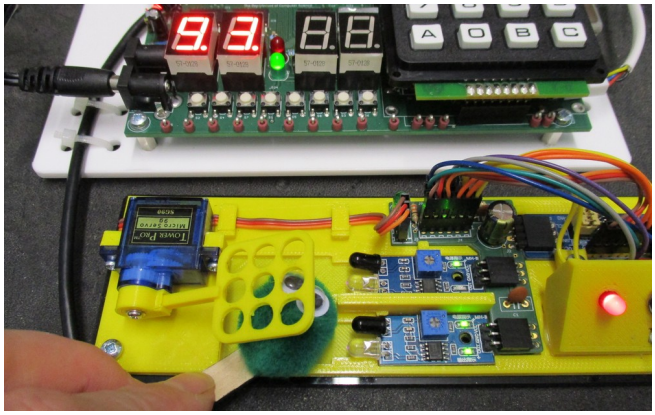


Figure 14: automatic triggering

To upload this new design into the FPGA double click on "Generate Programming File" to produce a new configuration bit file, then configure the FPGA as described in Laboratory script 1, using the `bug_trap.bit` file. If the hardware is configured correctly you should now be able to automatically capture your bug, as shown in figure 14.

Tip, if it seems that the FPGA has not been updated with the new control rules click on `Project -> Cleanup Project Files` and regenerate the bit file.

Task 3

To combine the automatic and manual control rules we need to include the toggle switch to select which set of rules should be active i.e. a MODE switch. The selected mode will be indicated by the LED:

- MODE=0 : manual mode, LED turned on, net controlled by the push button
- MODE=1 : automatic mode, LED turned off, net controlled by either the front or the rear infra-red sensor.

These combined control rules can be expressed as:

```
DESCRIPTION
IF MANUAL
THEN
  TURN ON LED
  IF FIRE
    CLOSE TRAP
  ELSE
    OPEN TRAP
ELSE
  TURN OFF LED
  IF SENSOR_1 OR SENSOR_2
    CLOSE TRAP
  ELSE
    OPEN TRAP
```

These two sets of rules could be rephrased, combined as shown below:

```
DESCRIPTION
IF ( (MANUAL AND FIRE) OR
     (AUTOMATIC AND (SENSOR_1 OR SENSOR_2)) )
THEN
  CLOSE TRAP
ELSE
  OPEN TRAP

IF MANUAL
THEN
  TURN ON LED
ELSE
  TURN OFF LED
```

These control rules can also be represented as the truth table shown in figure 15. When `MODE`=0 i.e. manual mode, only the `FIRE` button will activate the trap. When `MODE`=1 i.e. automatic mode, either `SENSOR` will activate the trap.

Note, remember these input sensors and switches are active when they produce a logic 0 and inactive when they produce a logic 1 i.e. we need INV.

Task : convert the above control rules into a logic circuit.

Tip, in the previous description each bracket term ( ) is the output of a logic gate. Knowing this break the descriptions below into a series of logic gate circuits, then assemble them into the complete circuit. This circuit should implement the truth table shown in figure 15.

```
W = NOT SENSOR_1 OR NOT SENSOR_2      AUTOMATIC = MODE
X = AUTOMATIC AND W                   MANUAL = NOT MODE
Y = MANUAL AND NOT FIRE
Z = Y OR X
```

Mode	Sensor 1	Sensor 2	Fire	Servo	LED
0	0	0	0	1	1
0	0	0	1	0	1
0	0	1	0	1	1
0	0	1	1	0	1
0	1	0	0	1	1
0	1	0	1	0	1
0	1	1	0	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0

INPUTS OUTPUTS

Figure 15: truth table for controller with automatic and manual modes

Task 4

If all has gone well you should have a circuit comparable to that shown in Appendix C. **Note**, this is not the only possible solution.

Task : update the schematic `bug_trap_controller.sch` to match your new circuit diagram. Double click on “Generate Programming File” to produce the configuration bit file. Next. configure the FPGA as described in Laboratory script 1,

using the `bug_trap.bit` file. Test that the circuit performs the specified control rules as described in figure 15.

Task 5

At the heart of the previous controller is a multiplexer, selecting between the manual and automatic control rules, as shown in figure 16. This is a key building block of any CPU.

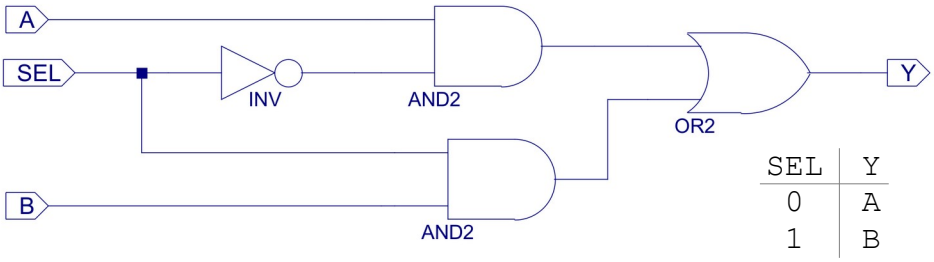


Figure 16: multiplexer

This higher level functionality can be implemented as a new `mux_2` component i.e. rather than keep drawing the circuit in figure 16, we can use a multiplexer symbol.

A new ISE project called `bug_trap_v2` has been created using this new component and can be downloaded from the module's VLE page. Using your preferred web browser download this zip file to `C:\Users\<USER NAME>`. Right click on this file selecting 'Extract all…' to unzip it.

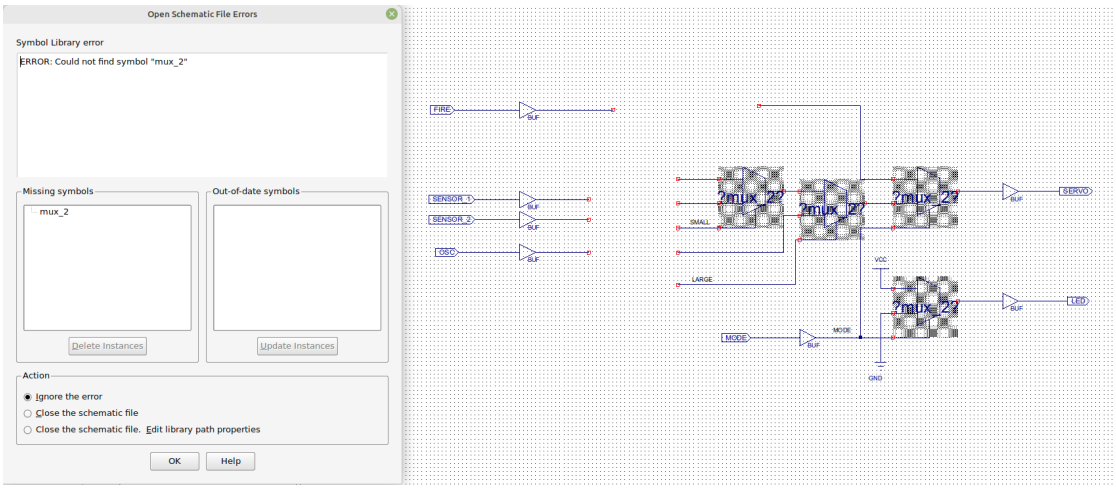


Figure 17: bug_trap_controller schematic – missing components.

Open this new project, then open the `Bug_Trap_Controller` schematic shown in figure 17. This example illustrates what happen when a schematic containing a missing component i.e. `mux_2`.

IMPORTANT : never click OK, always close the 'Open Schematic File Errors' pop up window by clicking on the icon. Then close the schematic.

The missing `mux_2` component is defined by its schematic (`.sch`) and associated circuit symbol (`.sym`), these can be downloaded from the VLE.

IMPORTANT, symbol files (`.sym`) must always be stored in the top level project directory i.e. the directory containing the project file (`.xise`).

Using your preferred web browser download from the VLE the file: `files.zip`. Unzip this file and copy the files `mux_2.sch` and `mux_2.sym` to your project directory.

Before you can use this new component you must first add this component to the ISE project. To add this component to the project left click on:

```
Project -> Add Source
```

This will open the 'Add Source Wizard', browse to the project directory and select the file `mux_2.sch`, then click Open. Then click Ok to confirm.

Re-open the `Bug_Trap_Controller` schematic it should now look like figure 18.

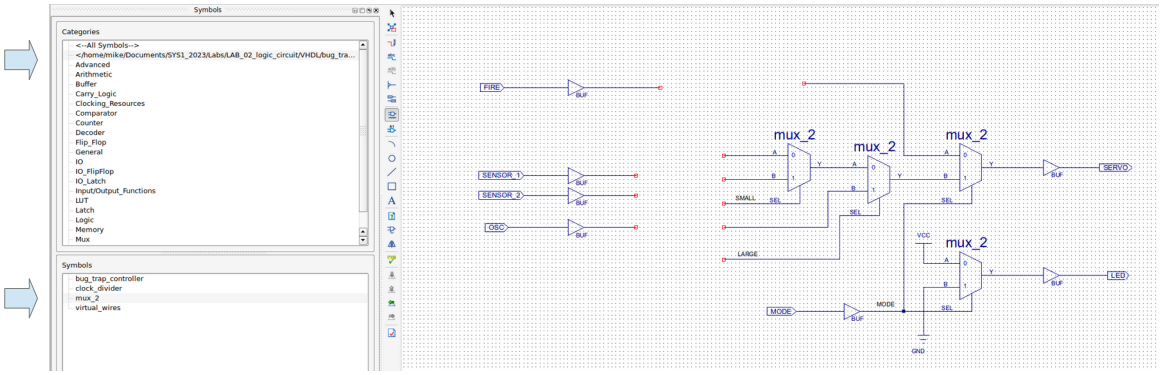


Figure 18: bug_trap_controller schematic

If you need to add a new `mux_2` component to the schematic left click on the "Add Symbol" icon  , within the Categories box you will see the current project directory listed, an example (my laptop) is shown in figure 18, this will be machine dependent. Select the `mux_2` component and add it to the schematic, as previously described.

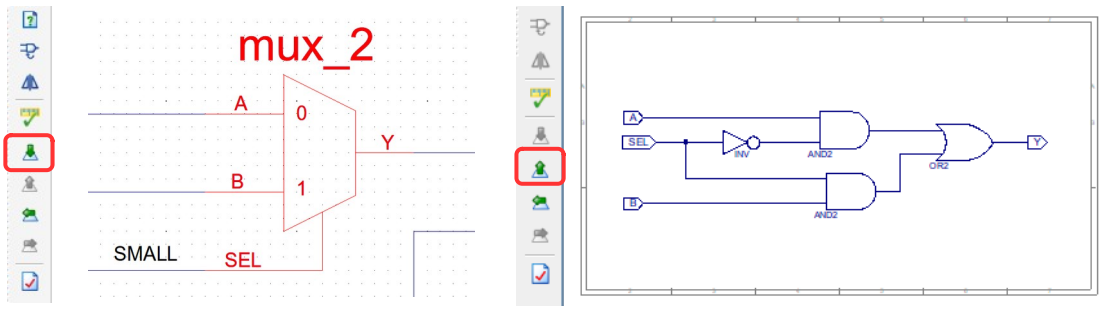
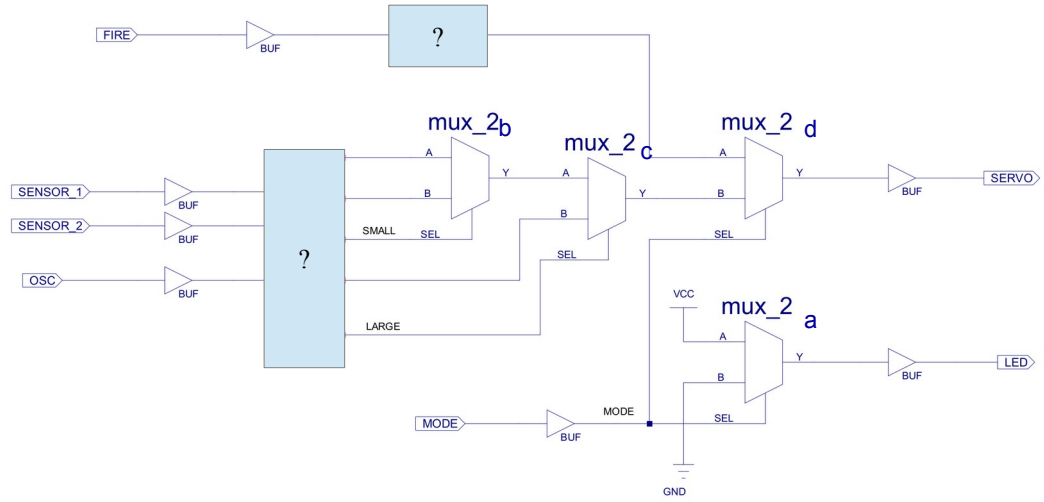


Figure 19: `mux_2` symbol (left), schematic (right)

To see the `mux_2` component's schematic left click on the `mux_2` symbol, then click on the Push into Symbol icon. This will open the `mux_2.sch` file. To return to the higher level schematic click on the Pop to Calling Schematic icon.




```
DESCRIPTION                          LOGIC
IF (AUTOMATIC)                       MANUAL = NOT MODE
THEN                                 AUTOMATIC = MODE
  TURN OFF LED
  IF (LARGE)                         LARGE = BOTH SENSORS
  THEN                                       ARE ZERO
    STAMP ON BUG
  ELSIF (SMALL)                      SMALL = ONLY ONE
  THEN                                       SENSOR IS ZERO
    CLOSE TRAP
  ELSE                               FIRE = ZERO WHEN
    OPEN TRAP                               PRESSED
ELSE
  TURN ON LED                        OSC = LOGIC 1 FOR 0.5s
  IF (FIRE)                                LOGIC 0 FOR 0.5s
  THEN
    CLOSE TRAP
  ELSE
    OPEN TRAP
```

Figure 20: multiplexer based implementation

A new breed of heavy weight cockroaches have been discovered. These are longer and more powerful than your typical bug. If caught these cockroaches can push their way through the trap i.e. as we all know cockroaches can not walk backwards :). Therefore, the bug trap controller circuit must be updated to implement the following rules:

- If the cockroach is small i.e. only covers one of the sensors, the net should hold the cockroach in the trap (to minimize the mess).
- If the cockroach is large i.e. covers both of the sensors, the net should repeatedly stamp on the bug to ensure it does not "escape". The 1Hz `OSC` clock input has been provided to help produce this behavior.

These rules can be implemented in a number of different ways. However, you could consider each decision point i.e. `IF` statement, shown in figure 20 as an input to a multiplexer.

Note, the symbols for constant logic 1 (`VCC`) and a constant logic 0 (`GND`) can be found in the `General` symbol category.

Multiplexer `mux_2a` controls the LED. The `MODE` signal from the toggle switch selecting a constant logic 1 (`VCC`) when in manual mode (`MODE`=0) and a constant logic 0 (`GND`) when in automatic mode (`MODE`=1).

Multiplexers `mux_2b` and `mux_2c` implement the automatic control rules. If the cockroach is `LARGE` input `B` of `mux_2c` is selected, else input `A`. This input is connected to the output of `mux_2b`, selecting between the `SMALL` and no bug present rules.

Multiplexer `mux_2d` controls the servo, selecting between the manual control rules on its `A` input and the automatic control rules on its `B` input.

Task : edit the schematic shown in figure 20 replacing the ? boxes with the correct logic gates / circuits to implement the required control rules.

Tip, you need to select the correct logic gates to generate these control signals:

- LARGE : produce a logic 1 when the bug triggers both sensors at the same time, else a logic 0.
- SMALL : produce a logic 1 when either sensor is triggered, but not if both are triggered, else a logic 0.

To help simplify the logic design you may want to invert the input sensors and switch signals to make them active high, such that a sensor produces a logic 1 when a bug is in the trap rather than a logic 0. The `OSC` input produces a square wave i.e. a signal that repeatedly alternates between a logical 1 and a logical 0. You can use this signal to repeatedly squash (stamps on) the bug instead of just holding it.

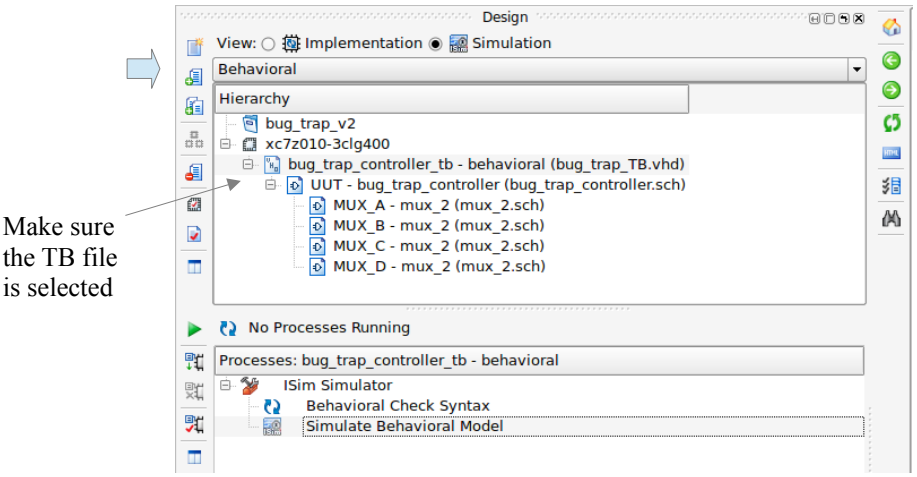


Figure 21: launching simulation

To verify the operation of the bug controller schematic a VHDL test bench

`bug_trap_tb.vhd` has been added to this project.

Left click on the 'Design' tab (project navigator window, bottom left) to display project 'sources'. Click on the 'Simulation' radio button, ensuring 'Behavioral' simulation is selected, as shown in figure 21.

Left click (highlight) on the testbench file `bug_trap_TB`, this will update the 'Processes for' source window, then double left click on 'Simulate Behavioural Model' to launch the simulation.

The VHDL simulator allows the operation of the circuit to be checked by examining its waveform timing diagram. Update the simulation time to **4us,** then click on the Run simulation for icon. If operating correctly the Servo and LED signals should match those shown in figure 23.

IMPORTANT, make sure you understand what this wave diagram is showing you as we will be using these types of simulations in most labs.

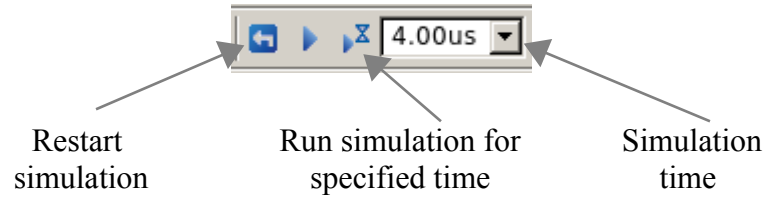


Figure 22: simulation controls

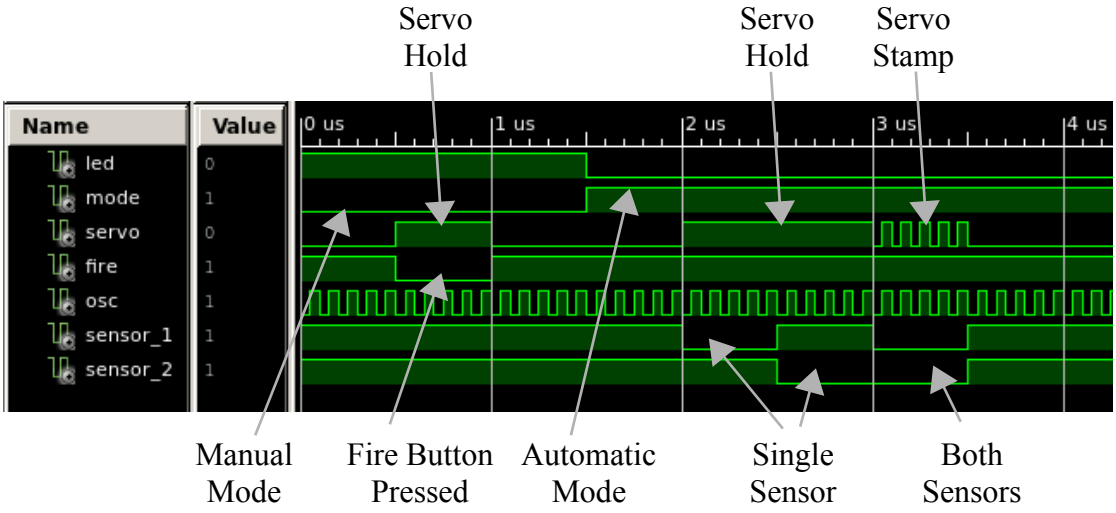


Figure 23: multiplexer based simulation

To upload this new design into the FPGA double click on "Generate Programming File" to produce a new configuration bit file, then configure the FPGA as described in Laboratory script 1, using the `bug_trap.bit` file. Download this into the FPGA to confirm your solution is working correctly.

Summary

The multiplexer (MUX) is one of the main building blocks found within any computer, an electronic "switch" used to route data from one point in a processor to another. Within the bug trap circuit it is used to select either the manual or automatic control logic, as shown in figure 24.

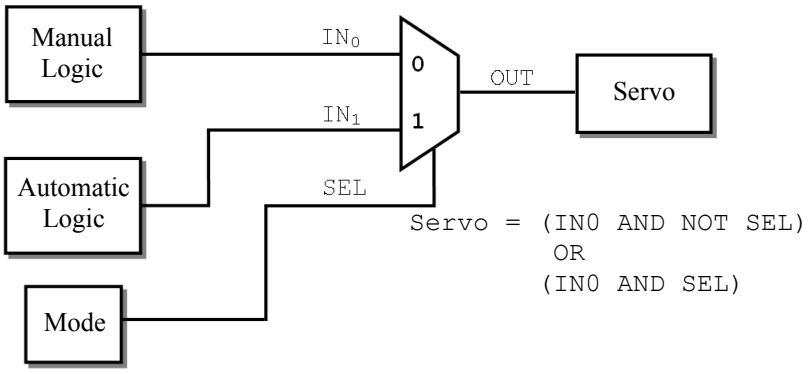


Figure 24: multiplexer

To simplify circuit diagrams a multiplexer is commonly represented using a distinctive trapezoid shaped component symbol. The operation of this multiplexer is controlled by its select (SEL) signal. This signal and its inverse are combined with AND gates, as shown in figure 16, therefore, one AND gate will always have a logic 0 on its input allowing the SEL signal to mask one multiplexer input and select data from the other, as shown by these logical relationships:

```
A AND 0 = 0
A AND 1 = A
```

The final OR gate acts as a joining function merging the outputs of the two AND gates onto a single output. As only a single AND gate can produce a logical 1 at any time, this simple circuit can select either input A or input B to drive its output Y. The truth table of this two input multiplexer is shown in figure 25.

SEL	A	B	Y	Description
0	0	0	0	Select A
0	0	1	0	Select A
0	1	0	1	Select A
0	1	1	1	Select B
1	0	0	0	Select B
1	0	1	1	Select B
1	1	0	0	Select B
1	1	1	1	Select B

Figure 25: Multiplexer truth table

The combination of control logic and multiplexers form the core of most computers i.e. hardware circuits that can work out what each instruction should do and then select the required data to be processed.

IMPORTANT, if your Xilinx project directory is on the `C` drive don't forget to copy it to your home directory e.g. `H` drive, otherwise you may loose your work.

Appendix A : Inputs and Outputs

Sensors

Bit 0	:	Push button	(0=Pushed)
Bit 1	:	Toggle Switch	(0=Down / 1=Up)
Bit 2	:	Front infra-red sensor	(0=Bug)
Bit 3	:	Back infra-red sensor	(0=Bug)
Bit 4 - 7	:	Not used	

Actuators

Bit 0	:	Servo	(0=Up / 1=Down)
Bit 1	:	LED	(0=Off / 1=On)
Bit 2 - 7	:	Not used	

Appendix B : adjusting infra-red sensor sensitivity

Using a small Phillips screwdriver adjust the variable resistor shown in figure B1. Rotating clockwise reduces sensor sensitivity. Rotating anticlockwise increases sensor sensitivity.

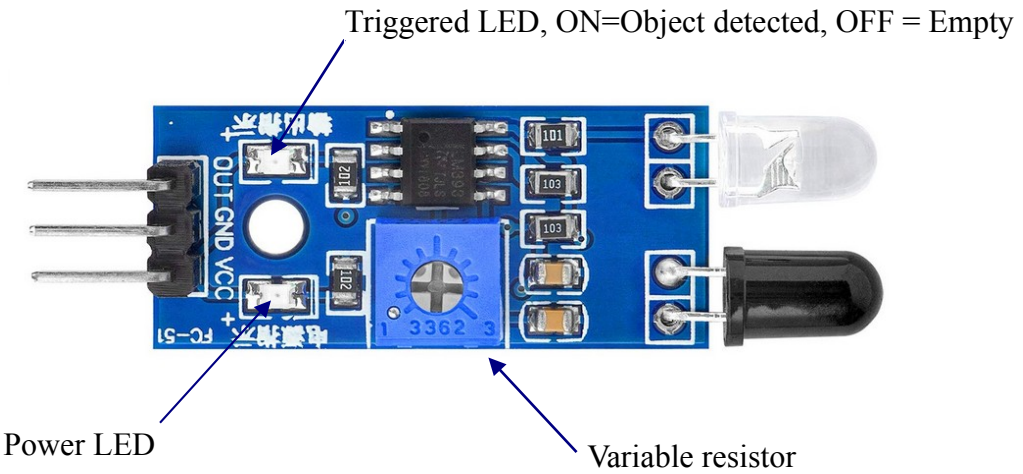


Figure B1: infra-red sensor module

Appendix C : Hardwired controller

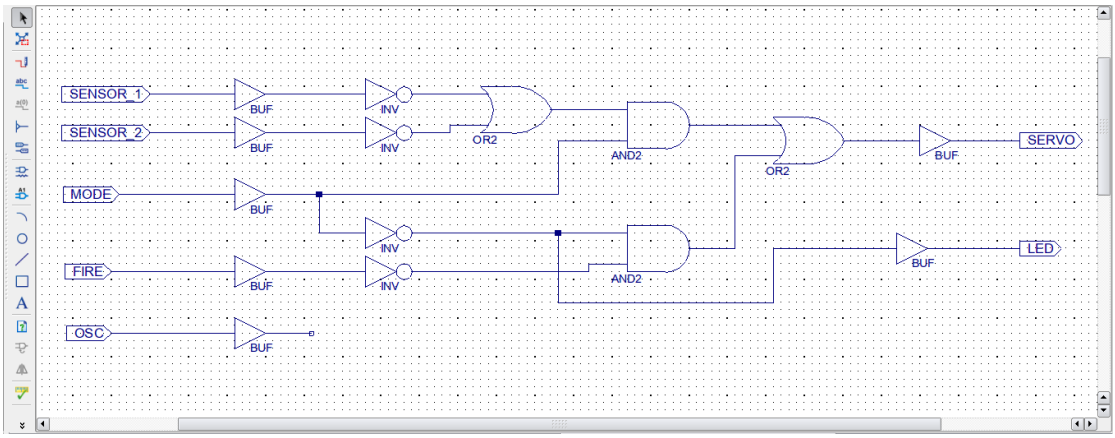


Figure C1: hardwired controller schematic