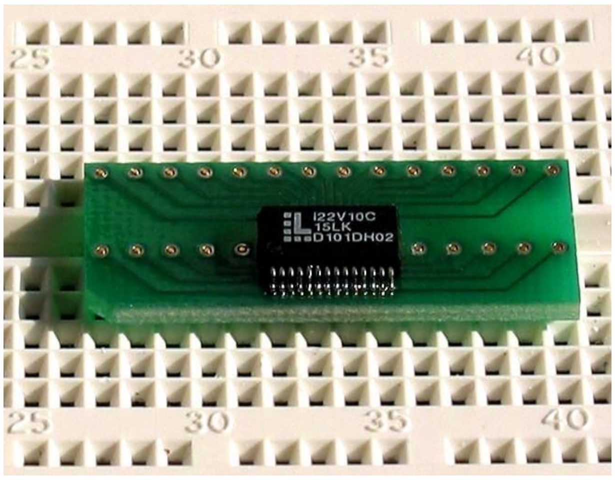
X5

Programmable Logic Devices

This exercise builds on the basics of digital system description in a hardware description language (SystemVerilog) introduced in lectures, allowing you to become familiar with PLD programming, simulation and testing. The PLD used is the ispGAL22V10, an electrically re-programmable device. This has the advantage that mistakes can be rapidly and cheaply corrected.



Schedule

Preparation time	:	3 hours
Lab time	:	3 hours

Items provided

Tools	:	n/a
Components	:	hook-up wire
Equipment	:	ispGAL22V10, Digital Test-Bed
Software	:	ModelSim, Synplify, ispLever

Items to bring

Essentials. A full list is available on the Laboratory website at https://secure.ecs.soton.ac.uk/notes/ellabs/databook/essentials/

Before you come to the lab, it is essential that you read through this document and complete ***all*** of the preparation work in section 2. If possible, prepare for the lab with your usual lab partner. Only preparation which is recorded in your laboratory logbook will contribute towards your mark for this exercise. There is no objection to several students working together on preparation, as long as all understand the results of that work. Before starting your preparation, read through all sections of these notes so that you are fully aware of what you will have to do in the lab.

Academic Integrity – *If you undertake the preparation jointly with other students, it is important that you acknowledge this fact in your logbook. Similarly, you may want to use sources from the internet or books to help answer some of the questions. Again, record any sources in your logbook.*

Revision History

September 04, 2013	Geoff Merrett (gvm)	Minor amendments for 2013/14
September 13, 2012	Mark Zwolinski (mz)	Revised for 2012/13
March, 2011	Geoff Merrett (gvm)	Heavily revised for phased implementation

1 Aims, Learning Outcomes and Outline

This laboratory exercise aims to:

- Introduce you to programmable logic devices
- Give you experience of SystemVerilog, an advanced Hardware Description Language, and state machines
- Give you an opportunity to design and test a digital system using a programmable logic device (PLD)

Having successfully completed the lab, you will be able to:

- understand the facilities and limitations of a simple PLD
- use advanced commercial software tools: Modelsim, Synplify Pro and ispLever to produce SystemVerilog code for a PLD design, simulate it, debug and synthesise into hardware.
- design a simple state machine, describe it in SystemVerilog, synthesise its circuit with Synplify Pro and implement it on a PLD with ispLever.

This laboratory provides an opportunity to program and test a programmable logic device (PLD) implementation of a state machine.

The exercise builds on the basics of digital system description in a hardware description language (SystemVerilog) introduced in lectures allowing you to become familiar with PLD programming, simulation and testing. The PLD used is the ispGAL22V10, an electrically re-programmable device. This has the advantage that mistakes can be rapidly and cheaply corrected.

The software tools installed in the Electronics Laboratory for ispGAL designs are:

- Modelsim from Mentor Graphics – a digital system simulator,
- Synplify Pro from Synopsys – an advanced digital synthesis tool,
- ispLever from Lattice Semiconductor Corp – a development system for ispGAL PLD devices.

You should have already completed the *SystemVerilog Walkthrough* (available through the ELEC1202 notes page on the web) to familiarise yourself with the design software tools.

Full information on ispGAL chips, including the ispGAL22V10 Data Sheet, is included on the Laboratory website at https://secure.ecs.soton.ac.uk/notes/ellabs/databook/pld, which also includes a link for downloading a starter version of ispLever.

Keep a complete record of your SystemVerilog source code, Modelsim simulation results, synthesised RTL diagrams and other details of the design and tests (successful or not) in your laboratory logbook. This includes sticking a printout of at least the final version of each source code module into the record of this exercise.

2 Preparation

Read through the course handbook statement on safety and safe working practices, and your copy of the standard operating procedure. Make sure that you understand how to work safely. Read through this document so you are aware of what you will be expected to do in the lab.

2.1 Filestore

For all System Verilog exercises, it is essential that you set up a suitable working directory/folder on your server account, which maps as drive H: on ECS Windows PCs. Do not store files on your Desktop or the C: drive, and do not use these locations for working directory structures. USB drives should be used for backup copies of your work, but should not be used as working filestore.

The software you will be using creates sub-directories and a great many files within the working directory structure. Some of these directories are arranged to provide "version control" – this means you need to take care that you always access and work on the most recent version of relevant files. Do not simply accept default file locations offered to you by the software when creating projects – apart from anything else these are almost invariably on the C: drive, which must not be used for such files.

Make sure Windows is configured so that file extensions are visible – you need to be able to find files generated by the software, and without visible extensions this can be very difficult. In the root of your H: drive, click on "Organise" → "Folder and search options". Pick the View tab, **un**check "Hide extensions for known file types" then click "Apply to Folders".

2.2 SystemVerilog Walkthrough

If you have not already done so, complete the *SystemVerilog Walkthrough* that is available through the ELEC1202 notes page on the web.

2.3 The ispGAL22V10

Find the ispGAL22V10 datasheet on the Laboratory web site. Read it carefully and answer the following questions in your logbook:

- *What is the difference between the SCLK signal (pin 1) and the CLK signal (pin 2)?*
- *How many of the IC's pins can be used as inputs, and how many as outputs?*
- *In how many ways can one of the IC's output macrocells be configured?*

2.4 SystemVerilog

Study the examples in your ELEC1202 notes and complete the following tasks:

a) **counter1**: a 4-bit binary counter

- The following SystemVerilog code represents a counter where a four-bit value 'volume' is incremented each clock cycle.

```
// simple counter for X6
module counter1(
  input logic clk, n_reset,
  output logic [3:0] volume);

  always_ff @ (posedge clk, negedge n_reset)
  if (~n_reset)
  volume <= 0;
  else
  volume <= volume + 1;
endmodule
```

- Draw a pseudo-ASM chart for this system (refer to the principles of a pseudo-ASM chart as per https://secure.ecs.soton.ac.uk/notes/ellabs/databook/pld/pld_tip.htm).

- Write a suitable SystemVerilog testbench for **module counter1** above. Refer to your ELEC1202 notes for suitable examples. Simulate your code using Modelsim on an ECS PC or using the ECS Undergraduate Compute Server via VPN.
- Run Synplify Pro on an ECS PC to synthesise **module counter1** into hardware targeting the Lattice ispGAL22V10 (as shown in section 3.2 of these notes). Print the RTL circuit diagram generated by Synplify and include it in your logbook.

b) **counter2:** a 4-bit binary up/down counter

- Extend your SystemVerilog for **module counter1** by adding two inputs: **up** and **down**. You should name this file **counter2**. The output **volume** should be incremented each cycle whenever input **up** is '1' and input **down** is '0', and decremented each cycle that **up** is '0' and **down** is '1'. When both **up** and **down** are '0', the volume does not change. When they are both '1' the priority is given to **up**.
- Write a suitable testbench for **module counter2**, simulate your code, and synthesise it into hardware. Record the SystemVerilog, simulation waveforms, and RTL circuit diagram in your logbook.

c) **volume_asm:** the state machine controller

- Develop an ASM chart for the 'CONTROLLER FSM' state machine shown in Figure 1, and described in Section 3.1. Write a SystemVerilog module (name it: **volume_asm**) for the state machine.

2.5 Laboratory equipment

You will be powering and exercising your PLD designs with the Digital Test-Bench. Find the User Guide for this unit on the laboratory "databook" Website, and decide which of its signal sources are best suited to the various PLD inputs, and how best to monitor your PLD's outputs.

---

3 Laboratory work – implementing a state machine on a PLD

At the start of the laboratory session, check that you can access your pre-prepared SystemVerilog files and can run the software tools: Modelsim, Synplify Pro and ispLever. If you have any problems tell a supervisor straight away. A kit for this exercise will be on your bench: an anti-static box containing an ispGAL22V10 – this IC will have been tested before issue, so if it doesn't work the problem is probably due to your code or construction!

The aim of this exercise is to design, simulate, implement and test a digital volume controller, described in Section 3.1. The exercises in this lab are designed to achieve this, by starting with a simple design and building up the software incrementally.

3.1 Full specification of the final volume control module

The PLD is to implement a state machine to control the sound level in a digital mixer. Figure 1 shows the block diagram of the controller that enables the sound level to be easily controlled with a couple of push buttons, **up** and **down**. When **up** is pressed the controller increments the sound level by 1 and when **down** is pressed, the sound level is decremented by 1.

When the **mode** switch is '1', it enables the volume to reach its maximum value (15). When **mode** is '0', the maximum volume is limited to (9). If the minimum sound level (0) occurs during decrement mode or maximum sound level (9/15) occurs during increment mode, further change must be inhibited since a limit has been reached.

The system should be implemented on a single PLD (where **increment** and **decrement** are internal signals), and should comprise two parts:

- a counter (COUNTER in Figure 1) which is implemented by the code you develop for **module counter4** (Section 3.5).
- a controller state machine (CONTROLLER FSM in Figure 1) which ensures that the signal **increment** is only '1' for a single clock cycle when the button **up** is pressed, and the signal **decrement** is only '1' for a single clock cycle when the button **down** is pressed.

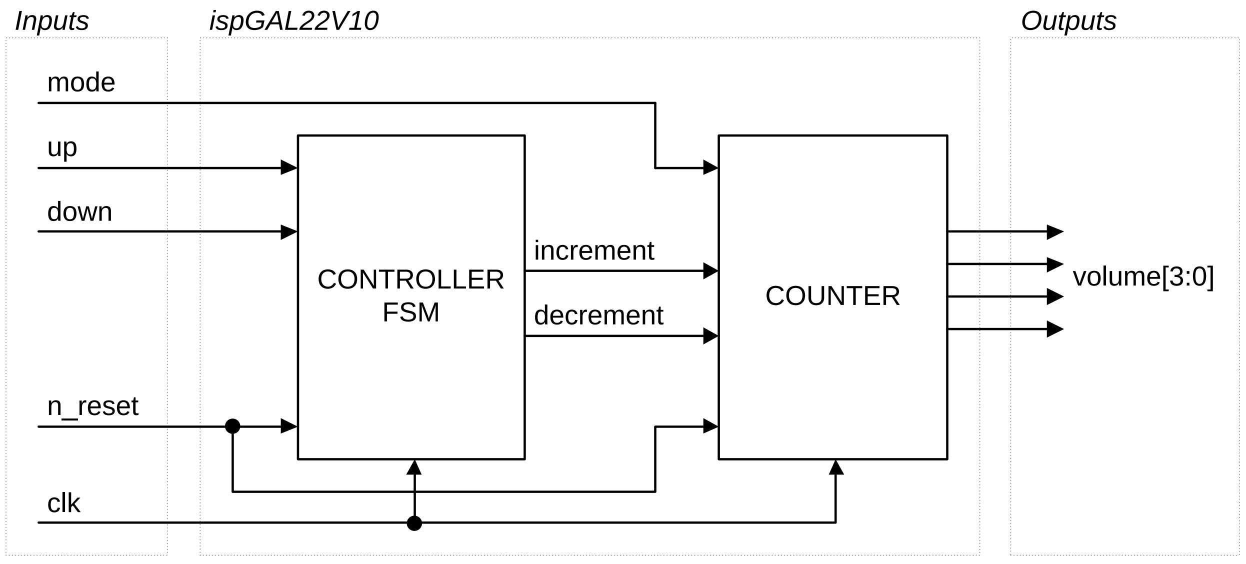
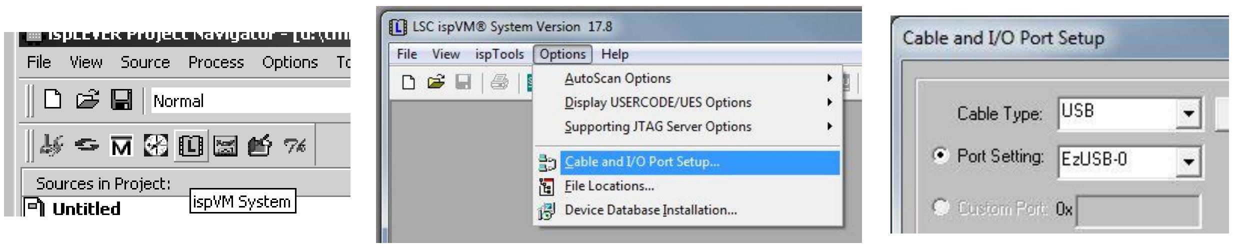


FIGURE 1: Volume controller comprising an FSM and an up/down counter

You should partition your module into (at least) two processes to implement this.

3.2 counter1: 4-bit binary counter

Using the SystemVerilog module **counter1** and the testbench you developed in your preparation work, confirm that the design compiles without error and simulates correctly in Modelsim. Run Synplify Pro to synthesise module **counter1** into hardware. When you are satisfied that the simulation results indicate a correct operation of the system and the synthesis completes successfully, process the EDIF (counter1.edf) file generated by Synplify Pro with Lattice ispLever, to produce a JEDEC file which can be used to program the chip as per Section 4 of the *In System Programmability Notes* on the ECS Laboratory web pages.


ispLEVER Project Navigator
File View Source Process Options To
Normal
Sources in Project:
Untitled
ispVM System
LSC ispVM® System Version 17.8
File View ispTools Options Help
AutoScan Options
Display USERCODE/UES Options
Supporting JTAG Server Options
Cable and I/O Port Setup...
File Locations...
Device Database Installation...
Cable and I/O Port Setup
Custom Port: 0x

Cable Type: USB
Port Setting: EzUSB-0

Figure 2. Selecting Lattice ispVM from the ispLever Project Navigator toolbar and setting the appropriate cable type.

When you start ispLever Project Navigator Classic, create a new EDIF project, browse to the counter.edf file created for your design by Synplify Pro and include it in the project. Make sure that you select the correct device family: GAL, device: ispGAL22V10 (look at the writing on your chip to identify the actual model number) and package SSOP (you may need to tick the 'show obsolete devices' box in order to list the GAL devices. Full information on this procedure is set out in detail in a walkthrough available on the Web:

https://secure.ecs.soton.ac.uk/notes/ellabs/databook/pld/edif22v10.htm

You can run the Lattice download utility "ispVM" either directly from the Windows "CAD" programs menu (go via "Lattice Semiconductor") or from the icon on the ispLever toolbar (see Figure 2). Plug your ispGAL22V10 into your prototyping board, locate the protoboard onto the tray of the Digital Test-Bed, and wire up the PLD to the Digital Test-Bed power rails and download adaptor ready for downloading your design – refer to the *In System Programmability Notes* to see how the various signals should be connected. Make a note in your logbook of which pins on the 22V10 have been assigned to which signals. Switch on the Digital Test-Bed and run the download software. Use the ispVM "scan" option. This should detect that there is a 22V10 in the in-system programming chain. If it doesn't, then you have made an error in your wiring or ispVM is not configured for the correct download cable type (see Figure 2 for details). Download your design.

Using the Digital Test-Bed's sources and monitoring signals, test the behaviour of your PLD, using the standard 2Hz clock.

Are the waveforms the same as those produced by simulation?

3.3 counter2: 4-bit binary up/down counter

Repeat the above process for the SystemVerilog you developed for **counter2**. Note that the Digital Test-Bed has two debounced pushbuttons available. You should use the debounced pushbuttons in your tests to drive the inputs **up** and **down**.

As before, make a note of which pins on the 22V10 have been assigned to which signals.

Are the pin assignments for ***counter2*** *the same as for* ***counter1****?*

When synthesising your design using Synplify Pro, you can use a constraints file to ensure a fixed pin allocation each time that you modify and re-synthesise your design. If a constraints file is not used, Synplify Pro will allocate signals to pins automatically and the allocation information can be viewed in ispLever. Remember that your design has a better chance of fitting into the PLD if a constraints file is not used. However, the disadvantage of not using a constraints file is that each time you modify and re-synthesise your design it might be necessary to rewire connections to your PLD pins on the protoboard should the pin allocation change. An example constraints file for **counter1** is shown below.

```
# ispGAL22V10-15LK constraint file for counter1
# counter1.sdc, 11 Mar 2010
# Note that vectors MUST be declared with signals as vector,
# pins as comma-separated list

# Attributes
define_attribute          {clk} loc {P2}
define_attribute          {n_reset} loc {P3}
define_attribute          {volume[3:0]} loc {P26,P25,P24,P23}
```

Re-synthesise your designs for **counter1** and **counter2**, using a suitable constraint file so that you do not need to rewire connections to your PLD pins. You will need to modify the above example.

How do the pin assignments for ***counter1*** *and* ***counter2*** *now compare? Why is this useful?*

3.4 counter3: 4-bit binary up/down counter without roll-over/under

Extend the SystemVerilog that you developed for **counter2** by preventing the output from "rolling over" (i.e. incrementing halts when the count value is 15 (or 4'b1111) and decrementing halts when the count value is 0). You may need to modify your testbench to check for this in simulation. Simulate your code, and synthesise it into hardware.

3.5 counter4: 4-bit binary/BCD up/down counter without roll-over/under

Extend your SystemVerilog for **counter3** by adding an input **mode**. When **mode** is '1', the output **volume** should increment and decrement through the full range of 4-bit binary number. When **mode** is '0', the output **volume** should only use 4-bit BCD (binary coded decimal) values. You should write a suitable testbench for **module counter4**, simulate your code, and synthesise it into hardware.

What happens if the input ***mode*** *switches from '1' to '0' while the output* ***volume*** *was greater than 9 (i.e. an illegal BCD state)? You should ensure that, in this scenario, your system automatically jumps to the BCD maximum.*

3.6 volume: the final volume control module

Develop a simple SystemVerilog module named **module volume,** comprising both the counter (which you developed and tested as **counter4**) and the controller state machine (which you developed in your preparation), as illustrated in Figure 1. The header of your module should be:

```
module volume(
  input logic clk, n_reset, up, down, mode,
  output logic [3:0] volume);
```

Write a suitable testbench to test the volume control system prior to synthesis in SynplifyPro.

Connect a 20-30Hz clock signal from the bench function generator as the system clock. Make sure the levels produced are "TTL-compatible". A slower clock signal will enable you to observe the state sequence in the FSM which might be helpful when debugging errors. Check that the PLD operates correctly at the selected clock speed.

Does your design act in all respects as per the functional specification of section 3.1?

---

4 Optional Additional Work

Marks will only be awarded for this section if you have already completed all of Section 3 to an excellent standard and with excellent understanding.

Use ispLever to produce a JEDEC file for the simple traffic light controller state machine that you designed in lab T3. Using a slow clock (or manually clocking the system using a switch), the state machine should repeat the sequence RED, RED+AMBER, GREEN, AMBER. Connect its RED, GREEN and AMBER outputs to the corresponding inputs of the Traffic Light toy (available from the laboratory support staff). Program the PLD and demonstrate the controller's operation in hardware.

Next, drive the system using a fast clock. Clearly this is not very practically useful, so you need to add a new signal 'DELAY' to the state machine to control the timing and dictate when the traffic lights should change phase. Note – you should not use this additional signal to clock the system! Connect a switch to the 'DELAY' input and demonstrate the correct operation of the traffic light sequence.