1 Course overview

1.1 The Course .................................................. 3
1.2 Approach Take in the Course ................................. 3
1.3 Learning Outcomes .......................................... 4
1.4 Hardware Description Language ............................. 4
1.5 Laboratories .................................................. 5
1.6 Lab rules ..................................................... 5
1.7 Assessment .................................................... 5
1.8 Our RTL Flagship System: MU0 ............................. 6
1.9 Curriculum .................................................... 6
1.10 Software Used in the Course ................................. 7
1.11 Hardware Used in the Course ............................... 8
1.12 Individual Project .......................................... 9
1.13 Documentation .............................................. 9

2 Sequential Datapaths ........................................ 11

2.1 Introduction ................................................ 11
2.2 Sequential Datapath ......................................... 11
2.3 The Sequential Datapath Abstraction ...................... 13
2.4 Sequential Datapath Design Examples ..................... 14
   2.4.1 A Timer or Signal Generator ......................... 14
   2.4.2 A One’s Counter ....................................... 18
   2.4.3 DVB-T ETSI EN 300 744 V1.4.1 (2001-01) .......... 18
2.5 Final Comments about Datapaths ........................... 19
1 Course overview

1.1 The Course

This is a first course on digital electronics for undergraduate engineers. The course has no prerequisites apart from a basic understanding of electronics and high school maths. The course is strongly lab and design oriented and is assessed accordingly.

1.2 Approach Take in the Course

The course takes a somewhat novel approach to digital design. In digital electronics, hardware behaviour is fundamentally predictable and there are a limited set of non-ideal performance limits that need to be controlled. As usual the main issue in the implementation of complex designs is that the simple gate level description of devices proves to be a complex and inappropriate starting point for designs. The approach we take is based on the simplication of complex electronics circuits by a process of abstraction of already familiar concepts. This is an exciting new approach to the teaching of DSM. New approaches are necessary if young engineers of the digital age are to effortlessly produce the complex designs that are now common place.

This approach involves the abstraction of a highly complex electronic design into a higher level system concept for which an accurate and reproducible circuit model exists. The abstractions we use are as follows.

1. *A combinational block* - a device that responds instantaneously to inputs (save propagation delays)

2. *A sequential block* - a device that responds to inputs only at the tick of a clock (save setup and hold times)

3. *A Finite State Machine* - a simple electronic model of controller or a Markov chain model. An FSM or stae machine is a system which moves through a set of states in response to inputs. In each state the system provides a set of outputs but has no memory of its prior states and is incapable of storing data across state transitions.

4. *A Sequential Datapath* - a circuit that performs a mathematical algorithm (can consist of combinational or sequential sub-blocks). It is controlled by a control word that acts like the line numbers of a computer program. Most often the control word is some kind of simple FSM such as a counter.

5. *A Register Transfer Level (RTL) system* - a circuit involving a FSM and a datapath. Here the FSM provides the control word so that the datapath can be transitioned through a complex sequence of states.
We will assume that all hardware can be treated as an interconnection of these fundamental blocks. That is: all systems can be built by interconnecting a set of Verilog modules or schematics that describe these fundamental blocks together. In addition we will make hardware that is clock synchronous to a unique system clock and all datapath processes are pipelined: designed as a sequence of synchronised processes. Asynchronous designs are not permitted in this course (although we will visit them).

1.3 Learning Outcomes

In this course you will learn to design complex digital systems. You will obtain experience in using CAD programs to produce complex digital hardware in Verilog HDL. You will obtain experience in the use of Field Programmable Gate Array (FPGA) and microprocessor development boards, electronics.

1. Use top-down design to translate system requirements into a practical digital design.
2. Design complex digital systems and implement these in programmable logic.
3. Learn to build and use microprocessors
4. Learn fixed point arithmetic.
5. Learn how to program in C and Verilog HDL.
6. Learn to use ISE WebPACK to realise complex digital designs in FPGAs.
7. Learn about other free and open source design software.
8. Learn practical electronics testbench skills and the ability to communicate appropriately via a lab notebook.

1.4 Hardware Description Language

In this course we will use a very limited set of synthesisable Verilog HDL for our designs. It turns out that Verilog simulations using a restricted set of syntactical heuristics can accurately model the above-mentioned abstractions. The significant fact is that the Verilog simulations will accurately model the hardware actual behaviour. The Verilog can also be used to deduce circuit schematics, though the latter will be mainly useful for summarising designs because a schematic by itself has no predictive power.
1.5 Laboratories

There is one three hour session of either a CLAB (Computer lab or tutorial) or a HLAB (hardware lab) each week. The CLABS and HLABS are conducted in alternate weeks and in different locations - Ian Ross 103 might be available during CLAB weeks. As modern digital design requires the use of complex software tools (in our case Xilinx ISE WebPACK 9.2i) CLABS will be devoted to the use of this software while hardware labs will involve implementations in hardware. All CLABS have assessable quizzes at the beginning of the lab. In all labs it is preferable for you to work alone. Due to a lack of PC work stations in HLABs, students with laptops are strongly encouraged to bring these along

In this course you will build strongly on acquired knowledge and techniques. All students must keep a HLAB notebook and a flash drive for digital designs. **Retain and reuse your old designs and document them thoroughly.** HLABs will be assessed on your notebook documentation and the success of your designs.

If you wish to use the labs out of the regular lab times then please contact me at least one week in advance. You may also buy your own hardware and work at home. Hardware can be ordered via the website


1.6 Lab rules

The following rules must be obeyed at all times.

- Do any prelab work before coming to the HLABs
- You must keep a neat and simple lab notebook that documents the date, notes and results
- It will not be possible to recover marks for missed labs
- Save your own files to a flash drive before leaving the lab
- Delete your computer files and clean up your work area before you leave
- Turn off equipment when leaving the lab.
- No food, no drink in the labs and no bags etc in the aisles

1.7 Assessment

Assessment is diverse: 10% CLAB quizzes, 20% Hlabs, 30% Exam and 40% individual project.
1.8 Our RTL Flagship System: MU0

A flagship RTL example will guide us in our thinking - The MU0 microprocessor softcore of the University of Manchester

http://intranet.cs.man.ac.uk/apt/publications/books/ARMsysArch/1stEd.php

MU0 will allow us to transition transparently from digital design to the Microprocessors and C and assembly language programming. MU0 is a relatively complex and useful processor soft core, comparable to many currently available microcontrollers. You should aim to obtain a good understanding of the full details of the operation of MU0 very early in the course.

![Block diagram of the MU0 microprocessor. Note that this is just the datapath. In addition there is a simple two state FSM. The states are (i) fetch and store and (ii) Decode and execute.](image)

1.9 Curriculum

The follow is the list of topics to be covered.

1. Course overview and microprocessor MU0
2. Numbers Systems
3. Verilog HDL
4. Digital system components using schematics and Verilog
5. Combinational logic abstraction. Karnaugh maps
6. Case studies of combinational circuits
7. The Sequential circuit abstraction
8. Sequential circuits, counters, registers, memories
9. Non-ideal effects in digital circuits
10. Finite State Machines
11. Design of FSMs
12. Datapath abstraction
13. Design of Datapaths
14. Register Transfer Level System (RTL) abstraction
15. The PICOBLAZE Softcore
16. Assembly Language programming
17. C programming
18. ARM microprocessors

1.10 Software Used in the Course

We use FREE and OPEN SOURCE software that you can use on your home or laptop PCs. All the software can be downloaded from

http://engnet.anu.edu.au/DEcourses/engn3213/Software

A DVD can be made available.

The main software packages are as follows.

1. Xilinx ISE WebPACK 9.2i is a complete design suite for Xilinx FPGAs including simulation, synthesis and device download. ISE WebPACK 9.2i is the main package used in the course and it is essential for you to obtain a personal copy for home use. Mastering ISE is the main technical achievement of the course. ISE WebPACK can be downloaded from the Xilinx website

www.xilinx.com
1. ICARUS Verilog is a command line tool and CYGWIN (the unix shell for windows) has been installed on IC and IR computers.

2. ICARUS Verilog is a very popular open source command line tool for simulating Verilog. It can be obtained from

http://www.icarus.com/eda/verilog

ICARUS is a very low overhead program and you should use it regularly. You can use ICARUS VERILOG with the WINDOWS CMD prompt.

3. GTKwave is an open source tool that allows ICARUS output to be displayed as traces (logic state analyser)

4. XAPP213 is the tool suite needed to create the PICOBLAZE embedded microprocessor softcore in SPARTAN FPGAs. It contains the KCSMP PICOBLAZE assembly language to VHDL compiler.

5. PCCOMP is a C-compiler for PICOBLAZE. It compiles C-code into PICOBLAZE assembly language

6. picoblazeIDE is a simulation environment for PICOBLAZE.

7. Keil compiler for ARM

8. The MinGW is the minimalist GNU C-compiler for WINDOWS. It is suggested as a means to obtain practice in C programming on your PC.

XAPP213, PCCOMP and picoblazeIDE do not need installation and are not installed on the uni computers. Download these from the course website and take them to labs.

1.11 Hardware Used in the Course

In this course we will heavily use the Spartan 3E Development board

http://www.digilentinc.com/Products/Detail.cfm?Prod=S3EBOARD

This board includes an XC3S500E FPGA. Peripherals include PS/2, switches, buttons, LEDs, LCD, ADC, DAC, VGA port and Ethernet connectivity. The boards can be ordered through Blackbox Consulting in Sydney

www.blackboxconsulting.com.au

them to send you the US to Oz mains adaptor.

The ARM2368 controller:

http://www.futurlec.com/ARM2368_Controller.shtml
1.12 Individual Project

There will be an individual project requiring a report of length \( \leq 20 \) pages plus hardware designs to be submitted at the end of semester. There is a choice of two projects: usually a reverse polish calculator (complex RTL system) and a CDMA wireless terminal (complex datapath system). The project is a significant, complex design to test your digital skills and generally requires substantial independent research. The project usually starts around Easter when you have the minimum design skills and ends at the end of semester.

Each student will write an individual report about their original work. The project will be implemented on the SPARTAN 3E boards using only digital logic. With the exception of DCMs (digital clock multipliers) and ROMs, no IP (Intellectual property) cores or HDL found on the web will be permitted. Project specifications will be provided by Easter. The RTL Design of MU0 in lectures will be the best early indicator of the magnitude of the project, although the project will also have graded milestones and a significant interface component.

1.13 Documentation

Datasheets and manuals for the hardware and numerous detailed articles on digital systems are available on the course website

Figure 3: The Philips ARM 2368 microcontroller development board.
2 Sequential Datapaths

2.1 Introduction

Up until now we have learnt to model circuits using three abstractions: combinational, sequential and FSM. FSMs describe controllers and are therefore useful for control systems. The physical model of an FSM is a Markov Chain - a very intuitive physical concept. In a traditional Markov Chain a system moves through a sequence of states in response to inputs and provide outputs only when in a state. Markov chains are sequential systems. They take up their states on time steps of an external clock. Markov Chains have no memory. They retain no record of previous outputs or inputs. All you know is that if a Markov chain finds itself in a particular state at any time, then you know that there is a finite set of states from which must have come on the previous time step and a finite set set to which it can transition on the next time step. You have already seen a VERILOG model of an FSM.

In this chapter we learn another useful abstraction: the sequential datapath. Like FSMs sequential datapath (SD) model processes with which you are already quite familiar: algorithms.

Fig. 4 shows a simple conceptual view of a datapath as would be applicable to that of MU0. In designing a datapath we assume that a control word plus inputs is provided as well as an input to the system. In this chapter we consider how to design a simple but practical datapath: a sequential datapath.

2.2 Sequential Datapath

A datapath usually serves the purpose of evaluating some mathematical function. Consider evaluating the following polynomial given an input 32 bit number \( x \). Assume that the coefficients are stored in memory.

\[
P(x) = \sum_{k=0}^{7} p_k x^k
\]  

(1)

There is more than one way that a datapath could be implemented to execute this function. Eqn. 1 can rewritten as follows

\[
P(x) = (((((p_7x + p_6)x + p_5)x + p_4)x + p_3)x + p_2)x + p_1)x + p_0
\]  

(2)

or,

\[
P(x) = (x^2)(x^2)[x^2(p_7x + p_6) + (p_5x + p_4)] + x^2(p_3x + p_2) + (xp_1 + p_0)
\]  

(3)

Fig. 5 shows at the left a flow diagram which executes the algorithm of eqn. 2 and the right a flow diagram that executes eqn. 3. These flow diagrams are generally referred to
as execution graphs. The particular execution graphs of Fig. 5 are termed sequential execution graphs because the algorithm is executed purely in sequence.

The sequential execution graph is the simplest algorithm to implement because the algorithm executes in steps that can be sequenced. The algorithm completes the current step before moving to the next. Generally speaking this process is slow but is usually unintensive on hardware.

An execution graph that executes faster is shown in Fig. 6. This architecture has steps that are executed in parallel or concurrently and are referred to as group sequential execution graphs. Group sequential execution implementations are more hardware intensive but execute more rapidly. When a group of steps require inputs that are not also the outputs of other steps in the group then these steps can be executed in parallel. We have already seen a similar hardware / speed trade-off when discussing combinational versus word serial implementations of circuits. The idea in datapaths is similar except that the algorithm datapath models considered are all sequential in nature.

Finally we note that any concurrent execution graph can be converted into an equivalent
2.3 The Sequential Datapath Abstraction

Sequential execution graphs are easier to develop due to ease of tracking data transformations. We design them using an algorithmic design approach.

One way to control sequencing is to use an FSM in which each node or group of nodes in the execution graph corresponds to a state. There is one important difference however. FSMs intrinsically have no memory yet datapaths need to store data during processing. For example in the above example of a sequential execution graph evaluation of a polynomial we could imagine each step in the execution being a state in a FSM but the computational results at the various steps would have to stored along the way for subsequent processing.

Be careful to use proper memories such as the register \texttt{vreg16.v} in MU0. VERILOG \texttt{regs} are NOT REGISTERS otherwise FSMs would have memory!

The simplest scenario would be to let the data be stored and read from memories or registers on the active clock edge while using combinational components to do data processing in between clock transitions. The processing would then naturally occur during the states while memory accesses would occur at the state transitions. We have already seen this in
action with MU0 where the memories are the program and data memory and the three registers: accumulator, instruction register and program counter.

The algorithm could be controlled by a state machine (FSM) as in the case of MU0 but this may not always be necessary. If the steps occur from start to finish in fixed order without conditional branches then a counter could perform the same task and make a much easier design. Remember that a counter is just a special case of an FSM but for simple up counters the implementation may not need a full FSM VERILOG implementation.

2.4 Sequential Datapath Design Examples

In a pure algorithmic datapath we have to design a counter or FSM with some memory elements. Maybe we should consider algorithmic datapaths for simple tasks with the more complex datapaths promoted toRTLs.

Note that the control word of Fig. 4 corresponds to the line numbers of the algorithm.

2.4.1 A Timer or Signal Generator

Design a sequential datapath to execute an algorithm to timeout a binary counter at a preset count. A signal generator or frequency synthesiser is a slightly more complex
device that outputs any number of signals at (usually but not always fixed) predetermined intervals. Examples are direct digital synthesisers (DDS) and Coordinate Rotating Digital Computers (CORDICS).

The variables (some of which have to be stored in registers for example) are

- **TIC**: The input bit from the calling device that starts the timer
- **TARGET**: the input word from the calling device that is the count value for the timeout.
- **CNTR**: An internal variable that indicates the current counter value
- **TOC**: The output bit that indicates the calling device that the count has completed.

It should be clear that TIC and TOC do not need to be stored because they are only ever valid after the active edge of the clock. TARGET and the current value of the CNTR need to be remembered during the execution of the algorithm. For this we use a register, \textit{vreg8} in a similar fashion to the \textit{vreg16} of MU0.

\begin{verbatim}
if(TIC)
\end{verbatim}
[1] Store TARGET
[2] initialise CNTR = 0
  while( |(Target $\XOR$ CNTR) )
[3] CNTR = CNTR + 1
  end while
[4] TOC=1
end if

This is a normal FSM design with FSM VERILOG coding style. There are three modules in the timer, timer.v which contains the FSM, vreg8.c the register and a testbench. The modules are shown below. We store the current value of the CNTR in the variable CNT. The construct is similar to the way the program counter worked in MU0. Since TARGET is only available in the initial state we need to catch it and store it in the variable Tlocal. The algorithm begins by initialising the registers containing CNTR and Tlocal. The states are given the variable name line as they correspond to the line numbers of the algorithm.

```verilog
module timer(clk, TIC, TARGET, TOC);
  input clk;
  input TIC;
  input [7:0] TARGET;
  output TOC;
  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  reg [1:0] line;
  reg [1:0] lnext;
  reg TOC;
  reg [7:0] CNTR;
  wire [7:0] CNT;
  wire [7:0] Tlocal;
  reg ent,rst,enc, rsc;
  vreg8 Target( clk, Tlocal, TARGET, ent, rst);
  vreg8 Count( clk, CNT, CNTR, enc, rsc);
  assign cond = |(Tlocal^CNTR);
  always@(*posedge clk)
    line <= lnext;
  always @(*posedge clk)
    lnext <= S1;
    if(TIC) begin
      lnext <= S1;
    end else begin
      lnext <= S2;
    end
    end
  S0: begin
    if(TIC) begin
      lnext <= S1;
      end else begin
      lnext <= S0;
      end
    end
  S1: begin
    if(cond) begin
      lnext <= S1;
      end else begin
      lnext <= S2;
      end
    end
  S2: lnext <= S0;
    default lnext <= S0;
    endcase
endmodule
```
case(line)
S0:
begin
if(TIC) begin
ent = 1'b1;
rst = 1'b0; //Set Tlocal = TARGET
TOC = 1'b0;
end else begin
ent = 1'b1;
rst = 1'b1; //Reset Tlocal = 0
end
end
S1:
begin
if( |(Tlocal^CNTR) ) begin
CNTR = CNT + 8'h01; //increment the counter
ent = 1'b0;
rst = 1'b0; //Store CNTR
end else begin
CNTR = 8'h00;
end
end
S2:
begin
TOC = 1'b1;
end
default: TOC = 1'b0;
endcase // case (line)
end // always @ (*)
endmodule // timer

module vreg8( clk, q, d, en, rs );
output [7:0] q;
input [7:0] d;
input clk;
input en;
input rs;
reg [7:0] state;
assign q = state;
always @(posedge clk) begin
if(en && ~rs) state <= d;
else if(en && rs) state <= 16'h0;
else state <= state;
end
endmodule // v_reg16

module TB_timer;
reg clk;
reg TIC;
reg [7:0] TARGET;
wire TOC;
integer i;
timer tmr(clk, TIC, TARGET, TOC);
initial begin
#clk = 0;
#TIC = 1'b0;
#clk = 0;
#TIC = 1'b1;
#TARGET = 8'b00010000;
#clk = 0;
#clk = 1;
end
Such a mechanical method for the design of datapaths allows for the design of relatively complex systems. However if the FSM is little more than a counter as in this case then much simpler solutions can be obtained. Consider the simpler solution shown in Fig. 10.

The Verilog implementation is correspondingly simpler.

2.4.2 A One’s Counter

Using a datapath with a register file, design a one’s counter that will count the number of 1’s in an input bitstream and return the result after completion.

2.4.3 DVB-T ETSI EN 300 744 V1.4.1 (2001-01)

The following viewgraphs show the block diagram of a terrestrial DVB (Digital Video Broadcast) transmitter The system as shown in the diagram is all digital and could be implemented in FPGA. If so implemented then the configuration could be modified and reloaded. Such an implementation would constitute a Software Defined Radio.

Two particular system blocks have been highlighted:

The Transport Multiplexer is a data scrambler. Its job is to randomise the data stream. This device has the property that repassing the scrambled data through the device reconstitutes the original data stream. There are identical data scramblers at the transmitter and the receiver.

The second subsystem is the inner encoder. It is a convolutional encoder that imbues the data stream with memory as a result of a tapped delay line. By memory one means that the data symbol at any symbol time is dependent upon the values of previous data symbols. Such memory can be exploited by a convolutional decoder in order to achieve forward error correction. The algorithm in use for convolutional decoding in modern DVB receivers is the Viterbi algorithm.

The system is defined as the functional block of equipment performing the adaptation of the baseband TV signals from the output of the MPEG-2 transport multiplexer, to the
terrestrial channel characteristics. The following processes shall be applied to the data stream (see figure 1):

- transport multiplex adaptation and randomization for energy dispersal; outer coding (i.e. Reed-Solomon code); outer interleaving (i.e. convolutional interleaving); inner coding (i.e. punctured convolutional code); inner interleaving; mapping and modulation; Orthogonal Frequency Division Multiplexing (OFDM) transmission.

The system is directly compatible with MPEG-2 coded TV signals ISO/IEC 13818 [1].

To ensure adequate binary transitions, the data of the input MPEG-2 multiplex shall be randomized in accordance with the configurations depicted below.

The system shall allow for a range of punctured convolutional codes, based on a mother convolutional code of rate 1/2 with 64 states.

### 2.5 Final Comments about Datapaths

FSM provides a good solution to a sequential datapath. But this model differs from a simple FSM in two important ways.

1. Some variables need to be stored across states of the FSM. For this we need memory devices such as registers.

2. If any processing has to be done prior to storage then the computations must be performed combinatorially while the state is valid.

Often an FSM is overkill. Most of the steps in the algorithm proceed automatically without a control flag input. Some sequential datapaths may only need a counter if the algorithm proceeds without jumps or has thousands of unconditioned steps. Concurrent and group sequential datapath designs may be required for high speed algorithms to meet timing constraints.
Figure 8: FSM of the timer.
Figure 9: GTKWAVE traces of the timer with a target of 0x10.

Figure 10: Simpler implementation of the timer
Figure 11: Block diagram of a Digital Video Broadcast transmitter.

Figure 12: Scrambler (transmit multiplexer) digital schematic.

The polynomial for the Pseudo Random Binary Sequence (PRBS) generator shall be (see note):

$$1 + X^{14} + X^{15}$$
Figure 13: The 64 state convolutional encoder for the inner code.
References