Digital Systems
EEE4084F

FINAL EXAM

15 June 2017

3 hours

Examination Prepared by:
Simon Winberg

Last Modified: 16-Jun-2018

REGULATIONS

This is a closed-book exam. Scan through the questions quickly before starting, so that you can plan your strategy for answering the questions. If you are caught cheating, you will be referred to University Court for expulsion procedures. Answer on the answer sheets provided. Make sure that you **put** your **student name and student number,** the course code **EEE4084F** and a title **Final Exam** on your answer sheet(s). Answer each section on a separate page.

DO NOT TURN OVER UNTIL YOU ARE TOLD TO

Exam Structure

Marked out of 120 marks / 180 minutes. Time per mark = 1min 30sec

Section 1	Section 2	Section 3	Appendices
Short Answers *(4 x 12 mark questions)* [50 marks] pg 2	Multiple choice *(6x4-mark questions + 3x2-mark true/false q's)* [30 marks] pg 4	Long Answers *(2x question HDL & MPI)* [40 marks] pg 6	A: Formulae B: Verilog cheatsheet C: MPI cheatsheet pg 8

RULES

- You must write your name and student number on each answer book.
- Write the question numbers attempted on the cover of each book.

NB ⇨
- Start **each section on a new page**.
- Make sure that you cross out material you do not want marked. Your first attempt at any question will be marked if two answers are found.
- Use a part of your script to plan the facts for your written replies to questions, so that you produce carefully constructed responses.
- Answer all questions, and note that the time for each question relates to the marks allocated.

Section 1: Short Answers [50 marks]

Question 1.1 [12 marks]

This question relates to the grid network shown in Figure 1 below, which comprises 18 of the same type of nodes, where each node only connects to its nearest neighbors via a 1Gbps link. This is essentially a non-uniform configuration because it is not a square and some of the nodes (e.g. corner nodes) have less links than others. Answer the following…

(a) The concept of *bisection bandwidth* is useful in gauging the performance of networked systems. Briefly explain the concept of bisectional bandwidth and why it is useful in judging the network performance design of a high performance computer system. [5 marks]

(b) Discuss, and calculate a bandwidth value, for the bisection bandwidth of the network shown in Figure 1. Assume each link has bandwidth of 1Gbps. Explain (using a rough diagram if needed, or a listing links to show the) how you decided the 'bisection' would be done in this case. [5 marks]. (*Hint:* Appendix A has formulas that might, or might not, be useful.)

(c) If each link in the network is 1Gbps then what is the maximum speed P1 can continuously stream data to P18, and P18 stream back to P1, assuming a 0us (i.e. zero) time cost for each node to route data from one link to another[1] [2 marks].

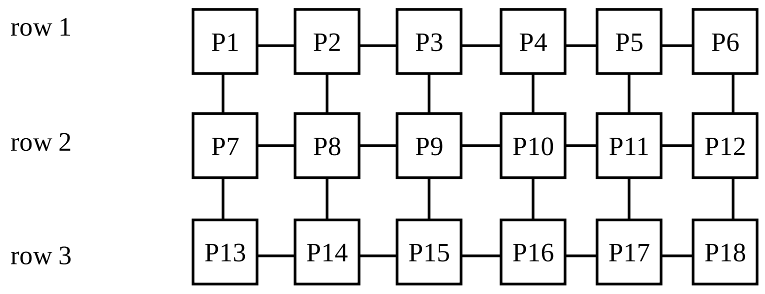


Figure 1 Cluster Topology for Question 1

[12 marks]

Question 1.2 [10 marks]

Answer the following questions and sub-questions concerning computation approaches…

(a) Many RC platforms are built using FPGAs. But they often also include a CPLD or PLA as well in the platform design. Explain the following…
i. What is the main difference between an FPGA and a PLA? [1 marks]
ii. Explain why a PLA (or a CPLD) in a reconfigurable computing system is often needed as configuration / glue logic whereas FPGA(s) are more commonly used for the actual computation (e.g., digital filtering) operations. [2 marks]

(b) Developing FPGA code can get you part of the way to an ASIC. What are some of the further difficulties (name at least one) and risks (name at least two) in taking an FPGA design further to become an ASIC? [4 marks]

(c) Contrast some of the benefit offered by using parallel code over sequential code for an embedded microprocessor-based or microcontroller-based solution. [3 marks]

[1] Consider that a connection from P1 to P3 routed via P2 still gives a 1Gbps performance; but if P1 sends two streams at the same time to both P2 and to P3 via P2, then there is only a 500MHz performance for both P1-P2 stream and for the P1-P2-P3 stream).

Question 1.3 [12 marks]

The design and implementation of applications for parallel and heterogeneous computing programs can be considered to involve seven major steps as listed below:

1. Understand the problem
2. Partitioning
3. Decomposition & Granularity
4. Communications
5. Identify data dependencies
6. Synchronization
7. Load balancing
8. Performance analysis and tuning

Answer the following:

(a) In terms of the step "Understanding the problem" the typical approach starts with establishing requirements for the system and later identifying what is termed critical "hotspots" where most of the parallelization work is done. Motivate why this identification of hotspots and parallelization of them is, all things considered, probably a more effective strategy than attempting to develop an entirely parallel solution for the problem. [3 marks]

(b) The spiral model provides a different perspective on the procedures, showing a more cyclic view. Yet both models are generally accurate, especially for large system development. Briefly describe the spiral model, indicating how it models the progression of development and mention (at least three) major activities that are commonly iterated in this model. Provide a rough figure to assist your explanation.[5 marks]

(c) The concept of granularity is sometimes misunderstood. Help clarify the term by answering the following: If the sequence of data {x1, x2, x3, … x*N*} has to be passed through each of *M* processors nodes in a cluster to compute the result (considering N large, M > 10, N >> M), would this computing problem be classified as *course grained* or *fine grained* granularity? (One word answer is fine.) [2 mark]

Question 1.4 [16 marks]

This question concerns MPI and, interestingly enough, machine learning. There are a range of views on ML, the main two pretty much: a) the machine learning romantics who see it as fabulous and behaving like real humans (and in some ways the technology can, wow!), and then there's the pragmatists who have probably had a difficult relationship with the technology and spent many days of their lives trying to get it to work effectively for their problems. For this question it doesn't really matter which view you hold.

Assume you've implemented ML methods which use two C functions: *learn* that trains your ML algorithm, and *infer* that given an input predicts a result. Somehow the *lean* and *infer* functions automatically load up whatever data is needed for them to work and *lean* will save effects of the learning when the program quits.

Although you aren't a pragmatists, let's consider that you have nevertheless found that your algorithm seems to work much of the time, but for some inputs the training just doesn't take and it gives silly predictions – even those the same inputs might work perfectly on a differently trained instance. Therefore you've decided to implement a voting system, using MPI as the mechanism to implement this on a cluster of systems so that you can run multiple instances of the ML algorithms which will send their predictions to the master node (ID 0, also referred to as 'rank' in the MPI nomenclature) to be displayed to the user.

The prototype of the functions are given below, they both need to be given an input vector ***x*** (assume this is a fixed length array of 3 integers) and a single output integer result (*y*).

```
void learn ( int x[3], int y );  // ML training alg: input x is array of 3 ints, and y is the output
void infer ( int x[3], int *y ); // ML inference alg: x is input array, y is predicted output (call by reference)
```

The program (let's call it *mipp*[2]) works by sending it a command line as follows:

mipp learn x1 x2 x3 y	: train the program with 3 inputs x1, x2, x3 and an output y
mipp pred x1 x2 x3	: predict and display an output for the 3 inputs x1..x3

(a) First sketch out the solution briefly in your exam book. Show a simple block diagram of how the system would work. [4 marks]

(b) Put together a *main()* function that works as follows, using MPI functions[3] where needed to implement the parallelism and message passing: [12 marks]

```
1. Initialize the MPI library.
2. Write out the ID of this process in the cluster.
3. If this node is the master node (ID==0) then
   o Set variable learning = 0
   o If the first command line argument is "learn" then
      ▪ Set learning = 1
     else
      ▪ Set learning = 0
   o Read in next 3 strings from command line, convert to ints and put into array x[0]..x[2]
   o If learning then covert the last string from command line into a integer and save in y
   o If learning==1 then
      ▪ send message LEARN to all other nodes with x and y as data
     else
      ▪ send message INFER to all other notes with x as data
      ▪ receive back the results for the other nodes and print these out
      ▪ {BONUS: report the majority vote, i.e. decide the most likely y result from the
        various y results that were sent back from the other nodes. [up to 5 bonus marks]}
4. If this node is not the master then
   o Wait for a message to be received, read message ID into msgid
   o If msgid == LEARN then
      ▪ Get x and y from the data, run lean(x,y) and then exit
     else
      ▪ Get x from the data, run infer(x,y) and send a message to the master to return y
```

[2] For Machine-learning Integer Prediction Program (MIPP)
[3] You may be delighted to have noticed the MPI Cheat Sheet in the appendix.

Section 2: Multiple Choice [30 marks]

NOTE: Choose only one option (i.e. either a, b, c, d or e) for each question in this section.
Questions are 5 marks each.

Q2.1 A FLASH ADC tends to need a lot of comparators, but it is quick. As an example calculate how many comparators are needed for a 6-bit FLASH ADC? (using the classic FLASH ADC design).

(a) 6.

(b) 31.

(c) 32.

(d) 63.

(e) 255.

[5 marks]

Q2.2 In the first seminar the paper "The Landscape of Parallel Computing Research: A view from Berkeley" was presented. It was highlighted that for processor cores "small is beautiful". Which of these statements accurately reflects the article's advice on multi-core processor architecture design:

(a) A small number of big cores should be used.

(b) A small processor gives better performance, so processor design should be kept simple.

(c) Many small cores tends to give better performance (e.g. MIPS) per unit area than a few big cores taking an equivalent area.

(d) A big cluster of small cores will almost always run faster, for high granularity problems, than a small cluster of big cores.

(e) It is unlikely that special purpose processors or reconfigurable coprocessors will break the classic treatise of Amdahl's law.

[5 marks]

Q2.3 What is an accurate definition of communication latency…

(a) The time it takes to send messages between tasks.

(b) Time taken to send a minimal length message from one task to another.

(c) The overhead of time involved in generating a message to be sent.

(d) The time taken to interpret a message once it is received.

(e) The time taken on average between the first point of generating the message and the last point of responding to the message on the receiver side.

Q2.4 Aspinall defines 3x dimensions for the spaces of parallel processing these are…

(a) Process level, granularity, and degree of parallelism.

(b) Selection, dispersion, collation.

(c) Granularity, parallelism and response.

(d) Process creation, arbitration, termination.

(e) Scheduling, execution, completion.

Q2.5 Answer true or false to each question below (each answer is 2 marks).

(a) DeepQA is a natural language processing system.

(b) A GPU is only able to process graphics data, in particular two dimensional matrices, vectors and pixels.

(c) The real-world performance measure is a major telling factor in determining whether or not it was worth the effort to develop a parallel solution.

(d) Amdahl's law states the number of transistors per square inch on ICs doubles every 18 months.

(e) A CPU tends to support more threads than a GPU.

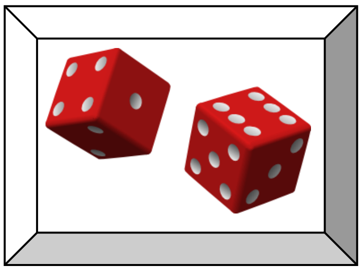
[10 marks]

Section 3: Long Answers [40 marks]

This section involves one rather lengthy question divided into four sub-questions

Virtual Craps Box (VCB)

Innovative machine for playing street dice in transit or in a cramped environment



Hey, here's a game that can be a lot of fun! At least if you're playing for matchsticks or virtual stakes. Yes, I know it is often played in casinos or for real money, but if you do that then that's your bad!

> *Disclaimer, small text and stuff: I am strongly against gambling but I am all for games to help sharpen your mind!*

So what is this question about? I want you to build a craps machine in the exam. Ha ha! No, only joking! But I would like you to prove some design expertise and Verilog programming. So, let's get started…

Description of the VCB:

The VCB basically is a box that you can shake about, throw, and catch. And while you are abusing it in that way, it is generating some random numbers, simulated dice roles to be precise. How does it do that? It has three digital accelerometers, for acceleration in the X, Y, Z axes. So when you shake it about the accelerometers will essentially be generating random values in its low order bits. Furthermore there are two die displays (what?! Look below, it's not to show if the machine is dead or alive – you should know the singular of 'dice' by now). A die display accept a 3-bit input, and displays a number 1 – 6, in the traditional way that a dice shows numbers – which is why it actually has 7 LEDs instead of 6 as you initially thought. The input values that map from the input bits to the die display is shown in the figure below (input bits in base two shown above or below the corresponding display):

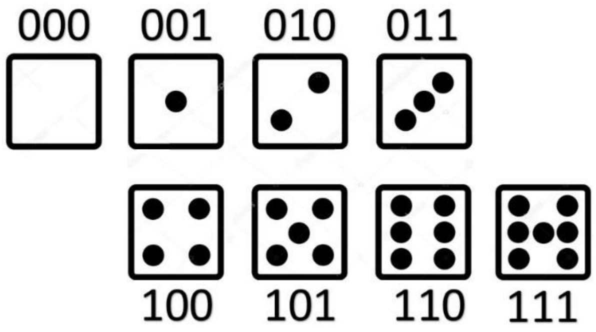


Figure 2: Die Display mapping of input bits (numbers shown in base 2) to display (LEDS illuminated).

The accelerometers are packaged in one module has a 2-bit input bus to select which axis you want (the *axis* input) and an 8-bit output that writes out the measured acceleration for that access (the *aval* output). It's easy to use (assume the UFC is already completed to link the pins of these modules to the FPGA).

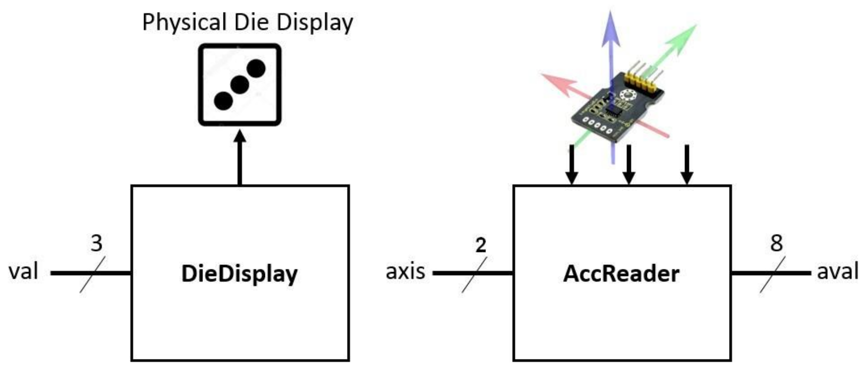


Figure 3: the COTS modules that the VCB utilizes and are wired to the FPGA.

On startup just assume the two Die Displays should show 77.

The system has an additional two devices: a green 'ready' LED and a beeper. The beeper is only utilized if the user is being annoying and it has two input lines to set its state (00=no sound, 01=slow beeping, 10=fast beeping, 11=buzzing). The ready LED and the beeper are off initially. The system enters Warmup mode after the displays have been initialized.

There are three modes:

- *Warmup*, mode 0: basically just juggling the 'dice' for a few seconds (10s to be precise). Just generating random numbers, using the **random process** given below. User not allowed to 'throw' (or in the craps jargon 'shoot') in this mode – technically the user can throw the dice in this mode also but the throw doesn't count, it doesn't change mode. The 'ready' LED is off in this mode. If Result mode had been activated the display that was given in that mode remains the same in this mode.
- *Shuffling*, mode 1: this is 'warming up' the dice beyond the warmup stage which is totally optional. The **random process** continues to run. If you warm up for more than 50s then it should start 'slow beeping' (to warn you that "really you're being annoying now, just throw!"). If you continue on for another 10s then it starts 'fast beeping' (i.e. to say "no fun, you've taking too long, time up, pass it on!" for 3s) and you have to pass it to the next person and it will reset (back to mode 0). The 'ready' LED is on in this mode, the DieDisplay modules just show fast-changing random numbers (see **display sample process** below).
- *Throw*, mode 2: this is triggered by the user causing an acceleration reading that has |x+y+z|>128 (i.e. you've thrown it or knocked it hard – note that by x, y, z I mean the x is the *aval* value returned for the x-axis, y for the *aval* for the y-axis etc.). The system moves to *result* mode.
- *Result*, mode 3: in this mode the system does a reading (same as display process below) and displays the result on its two DieDisplay modules. It doesn't run random process. The system sticks in this mode for 5s and then goes back to the Warmup mode.

The **random process** works as follows:

This process describes how random numbers are generated from the accelerometers. Initially set the 24-bit rval = 0. The following set of operations are performed every 256th tick of the fast clock (clk):

- Select X axis (axis=0) and read the value and add it to rval, i.e. rval = rval + aval
- Select Y axis (axis=1) and read the value and add it to rval, i.e. rval = rval + 16*aval
- Select Z axis (axis=2) and read the value and add it to rval, i.e. rval = rval + 64*aval [4]

[4] Technically it should cycle between which axis is given a greater weighing, but let's ignore that for now.

Display sample process should run every 100ms (if not in warmup mode):

- Generate the two dice values D1 and D2 such that
    - D1=((rval&0xFF) mod 6)+1,   (you can assume there is a mod6 module available)
    - D2 =(((rval shr 8)&0xFF) mod 6)+1
- Display the D1 and D2 values to the Die Displays (doing pin assignments as needed)

TODO…

See next page for a starting point for the vcb top level entity (i.e. by top level entity this means it is the parent of all other modules and will connect up to the external IO pins).

You essentially need to implement the aspects requested in the subquestions below in Verilog. You can assume you start coding from after the “over to you” comment given in the code on the next page.

You do *not need to rewrite the code already provided*. If you need to add another module, you can do so, but start on a separate page so that it is clear that it is not part of the top level entity.

NB: ANSWER ALL 4 QUESTIONS 3.1-3.4 BELOW!!!

Question 3.1 [12 marks]

Implement a state machine (always loop) or module for the **random process**. If you are going to implement a separate module then please indicate how it would be connected in the top level entity.

[12 mark]

Question 3.2 [10 marks]

Implement the **display process**. Note that it must trigger every 100ms when in shuffling mode (mode 1). You should be writing to the necessary pins (val1, val2) as needed. (not you don’t need to implement the annoyance monitor, see below, for this question).

[10 mark]

Question 3.3 [10 marks]

Implement the ‘annoyance’ monitor for shuffling mode (mode 1). i.e. if the user is shuffling for too long (>=50s) then it starts show beeping (setting buzzer=1) and then if the user continues on doing that for another 10s then it needs to fast beep and timeout as explained in the operation above.

(you don’t need to repeat any code given for 3.2, you can just say ‘add on’ or you can do 3.2 and 3.3 together in one answer.)

[10 mark]

Question 3.4 [8 marks] *ESSAY QUESTION* (for those struggling with code)

Explain in English how you would implement the last two modes, i.e. the throw and result mode. You do not need to provide code, but you can provide some code snippets if you think that will help and/or you can provide a sketch to help illustrate your explanation (around ½ page explanation should do).

[8 mark]

[total marks for this section: 40]

Here is a starting point for the implementation:

```
//-----------------------------------------------------
// Design Name : VCB
// File Name   : vcb.v
// Function    : Visual Craps Box toplevel module
//-----------------------------------------------------

module mod6 ( input [23:0] val, output [2:0] result );
  // assume implementation for mod6 is given, returns val%6
endmodule

module vcb    (
  // Inputs
  clk     ,  // a clock line, runs at 10MHz
  sclk    ,  // slow clock, runs at 1KHz (useful timing in ms obviously!)
  aval    ,  // reading for selected accelerometer axis
  reset   ,  // reset Input
  // Outputs
  axis    ,  // accelerometer axis select line
  val1    ,  // first  die display
  val2    ,  // second die display
  ready   ,  // ready LED
  buzzer     // 2-bit buzzer output lines
  );
//------------Input Ports--------------
   input       clk;
   input       sclk;
   input [7:0] aval;
   input       reset;
//----------Output Ports--------------
   output reg [1:0] axis;
   output reg [2:0] val1;
   output reg [2:0] val2;
   output reg       ready;
   output reg [1:0] buzzer;

//------------Internal Variables--------
   reg mode [2:0] = 0; // set mode to ready
   reg rval [23:0]= 0; // set rval to 0
   // here is some space to add more variables

//-------------Code Starts Here-------

   // over to you for the rest :-)

endmodule
```

END OF EXAMINATION

Appendix A: Formulae

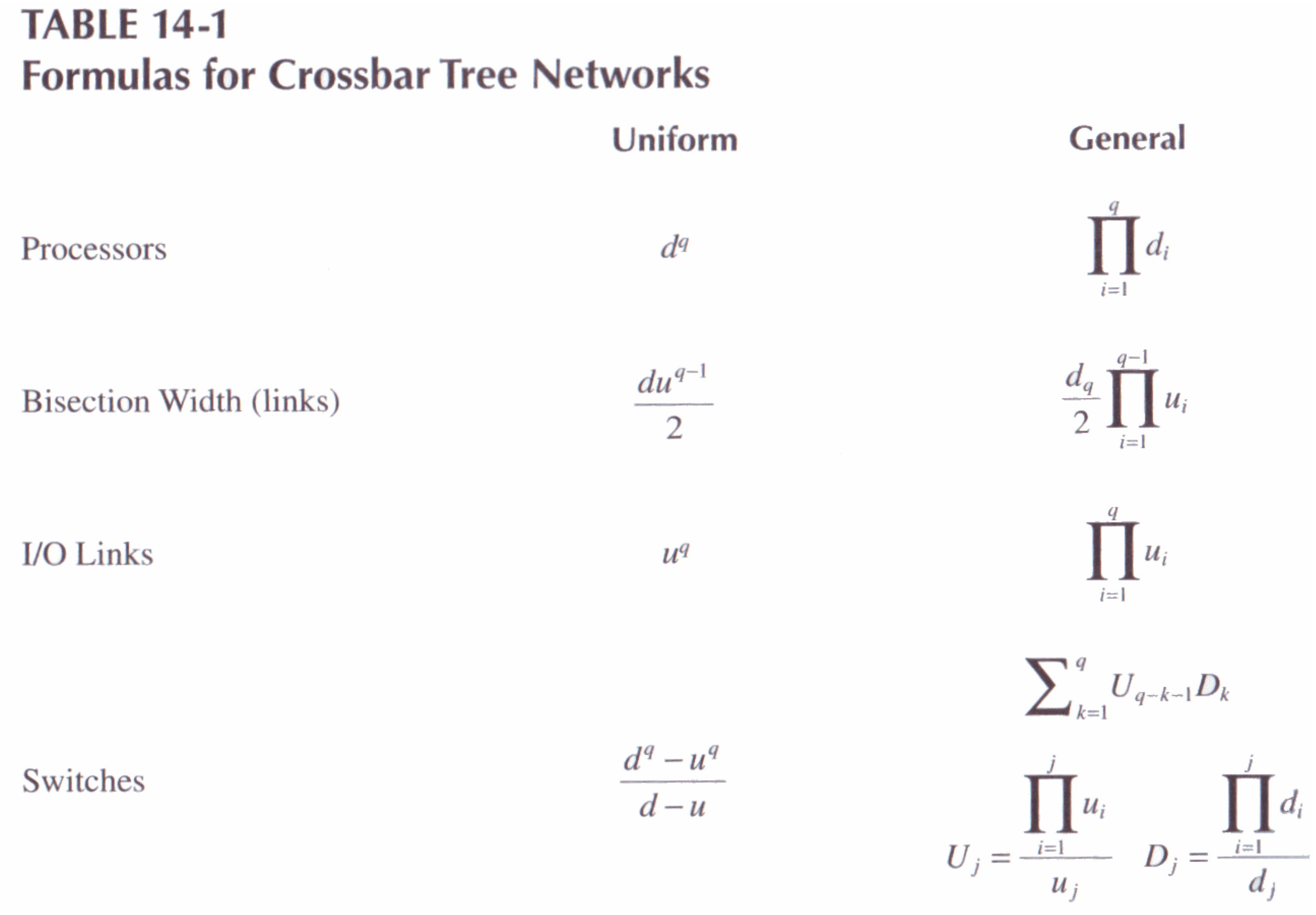


TABLE 14-1
Formulas for Crossbar Tree Networks

	Uniform	General
Processors	d^q	$\prod_{i=1}^{q} d_i$
Bisection Width (links)	$\frac{du^{q-1}}{2}$	$\frac{d_q}{2}\prod_{i=1}^{q-1} u_i$
I/O Links	u^q	$\prod_{i=1}^{q} u_i$
Switches	$\frac{d^q - u^q}{d - u}$	$\sum_{k=1}^{q} U_{q-k-1} D_k$ $U_j = \frac{\prod_{i=1}^{j} u_i}{u_j}$ $D_j = \frac{\prod_{i=1}^{j} d_i}{d_j}$

Table from pg. 291 of Martinez, Bond and Vai 2008

Appendix B: Verilog Cheat sheet

Numbers and constants

Example: 4-bit constant 10 in binary, hex and in decimal: 4’b1010 == 4’ha -- 4’d10

(numbers are unsigned by default)

Concatenation of bits using {}

4’b1011 == {2’b10 , 2’b11}

Constants are declared using parameter:

parameter myparam = 51

Operators

Arithmetic: and (+), subtract (-), multiply (*), divide (/) and modulus (%) all provided.

Shift: left (<<), shift right (>>)

Relational ops: equal (==), not-equal (!=), less-than (<), less-than or equal (<=), greater-than (>), greater-than or equal (>=).

Bitwise ops: and (&), or (|), xor (ˆ), not (˜)

Logical operators: and (&&) or (||) not (!) note that these work as in C, e.g. (2 && 1) == 1

Bit reduction operators: [n] n=bit to extract

Conditional operator: ? to multiplex result

Example: (a==1)? funcif1 : funcif0

The above is equivalent to:

```
  ((a==1) && funcif1)
  || ((a!=1) && funcif0)
```

Registers and wires

Declaring a 4 bit wire with index starting at 0:

wire [3:0] w;

Declaring an 8 bit register:

reg [7:0] r;

Declaring a 32 element memory 8 bits wide:

reg [7:0] mem [0:31]

Bit extract example:

r[5:2]   returns 4 bits between pos 2 to 5 inclusive

Assignment

Assignment to wires uses the assign primitive outside an always block, e.g.:

assign mywire = a & b

Registers are assigned to inside an always block which specifies where the clock comes from, e.g.:

```
always@(posedge myclock)
    cnt = cnt + 1;
```

Blocking vs. unblocking assignment <= vs. =

The <= assignment operator is non-blocking (i.e. if use in an always@(posedge) it will be performed on every positive edge. If you have many non-blocking assignments they will all updated in parallel. The <= operator must be used inside an always block – you can’t use it in an assign statement.

The blocking assignment operator = can be used in either an assign block or an always block. But it causes assignments to be performed in sequential order. This tends to result in slower circuits, so avoid using it (especially for synthesized circuits) unless you have to.

Case and if statements

Case and if statements are used inside an always block to conditionally update state. e.g.:

```
always @(posedge clock)
  if (add1 && add2) r <= r+3;
  else if (add2) r <= r+2;
  else if(add1) r <= r+1;
```

Note that we don’t need to specify what happens when add1 and add2 are both false since the default behavior is that r will not be updated. Equivalent function using a case statement:

```
always @(posedge clock)
  case({add2,add1})
  2’b11  : r <= r+3;
  2’b10  : r <= r+2;
  2’b01  : r <= r+1;
  default: r <= r;
endcase
```

Module declarations

Modules pass inputs, outputs as wires by default.

```
module ModName (
  output reg [3:0] result,  // register output
  input [1:0] bitsin,  input clk, inout bidirectnl  );
 … code …
endmodule
```

Verilog Simulation / ISIM commands

```
$display ("a string to display");
$monitor ("like printf. Vals: %d %b", decv,bitv);
#100   // wait 100ns or simulation moments
$finish  // end simulation
```

Appendix C: MPI Cheat sheet

Initialize & Finalize	
MPI_Init(&argc, &argv);	Called at start of program before using any other MPI functions
MPI_Finalize();	Called once done with using MPI
Information about the collaborating nodes	
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);	Returns the process ID of the current process (a value between 0 and number_processors-1 inclusive)
MPI_Comm_size (MPI_COMM_WORLD, &num_procs);	Returns number of processes available, typically called after MPI_Init.
MPI_Get_processor_name (&name, &result_length)	Returns the name of the host running on (generally not used)
Passing Messages	
MPI_Send (message, message_size, MPI_CHAR, destination, tag, MPI_COMM_WORLD);	Send a message from the current process to another process (indicated by the destination ID). If message is bytes (MPI_CHAR) message_size is size in bytes. MPI_CHAR can be replaced with other type e.g. MPI_INT but I put char because that's more general.
MPI_Recv (message, max_message_size_in_byes+1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status)	Receives a message on the current process from another process. Note re source parameter: Set source = MPI_ANY_SOURCE to wait on and receive a message from any source. Otherwise source is set to the ID of the processor to receive a packet from. MPI_CHAR can be replaced with other type e.g. MPI_INT but I put char because that's more general.
MPI_Bcast(message, message_size, MPI_CHAR, source, MPI_COMM_WORLD);	Broadcasts message from current process to all other processors.
MPI_Reduce(&value, &value_collect, count, type, MPI_Op, server, MPI_COMM_WORLD);	Performs a reduction (e.g., summation or find min) of a variable on all processes, sending the result to a single process. Type could be e.g. MPI_CHAR. Built in ops: PI_MAX, MPI_MIN, MPI_PROD, MPI_SUM, MPI_LAND, MPI_LOR, MPI_LXOR, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC in place of MPI_Op
MPI_Allreduce(&value, &value_collect, count, type, MPI_Op, MPI_COMM_WORLD);	Performs a reduction of a variable on all processes, and sends result to all processes (takes longer)
MPI Data Types	
MPI_CHAR	char
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double

Comment on the datatypes used:

Remember that the message is a void pointer, and count is the size in bytes of the data pointed to by message.

```
int MPI_Send(const void *message, int count, MPI_Datatype datatype, int dest, int tag,
             MPI_Comm comm);
```