A Handbook of Parts for ECE 241/253 Circuits

Revision : 1.5

Paul Chow
Edward S. Rogers, Sr. Department of Electrical
and Computer Engineering
University of Toronto

September 16, 2020

Contents

Chapter 1

Introduction

In the earliest days, hardware design was done by wiring together a number of logic chips, where each chip provided some function such as a logic gate or something larger such as an adder. A logic designer had to rely on whatever logic functions were provided by the chip vendors. The catalogue of available chips was provided as a *handbook*, which described for each chip the functionality, the inputs and outputs available on each pin of the chip and the electrical characteristics, such as current draw and the delay of the logic from input to output. One of the most famous handbooks was the *The TTL Data Book for Design Engineers* from Texas Instruments. If you like to refurbish old arcade games, you will find that handbook very useful!

The purpose of this handbook is to provide you with all the *parts* or components that you will need to do the labs for ECE241/253. We start with the pinouts for the discrete chips that you will use in the first two labs. The remaining chapters describe the logic functions that you will need and how to express them in Verilog. Unlike with discrete chips, Verilog is much more flexible. You can describe arbitrary logic functions in Verilog, effectively creating your own *virtual chips* that you can wire together in your Verilog code.

All the Verilog constructs needed to do the ECE241/253 labs are shown in this handbook. You will still need the textbook for explanations. In fact, you should be able to build any logic circuit using only the Verilog described in this handbook. Verilog is a much more complicated language. More advanced features can allow you to describe circuits more compactly and flexibly, but they are not necessary for ECE241/253. Verilog also has features that cannot be used to create logic circuits. Any attempt to use

those features will likely cause you much grief, so beware if you decide to try other Verilog constructs that are not taught in ECE241/253.

1.1 Organization

The chapters of this handbook are organized into the major topics for fundamental hardware design.

In the spirit of the old handbooks, when a logic function or component is described, we first show a schematic or block diagram and then the equivalent Verilog.

Chapter 2

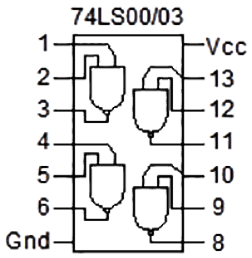
Discrete Chips

In the 1970's, the largest chips had on the order of a few thousand transistors. Most logic design was done with discrete chips that provided a few logic gates and some larger functions like adders. All the labs in the equivalent course to ECE241/253 at that time used these discrete chips. In Labs 1 and 2 of ECE241/253, you will build some simple logic circuits using these types of chips. An important take away from these labs is that logic design is done by connecting the outputs of logic functions in one chip to the input of logic functions in the same or different chip using wires.

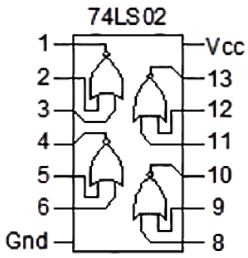
You will soon be learning to use Verilog, which is really a textual description of logic circuits where you have the flexibility of describing arbitrary logic functions and connecting them together. Essentially, you can create your own library of *virtual chips*, called Verilog *modules*. You are not restricted to the kinds of chips that some manufacturer provides. However, the analogy to connecting physical discrete chips into circuits using wires still holds. You are just doing the wiring *virtually* using Verilog. To be successful at writing Verilog, it is important to keep this analogy in mind. **Most users of Verilog struggle when they start trying to write *programs* as they would do when writing a C program.** Verilog is **NOT** for writing programs. It is for describing circuits.

In this chapter, we provide the pinouts of the discrete chips available in the labs. The pinouts show which pins of each chip are used to access the inputs and outputs of the gates in each chip.

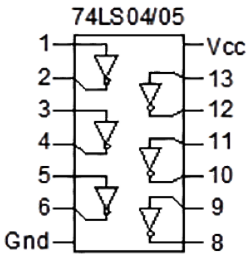
Pin-out of Selected TTL Chips



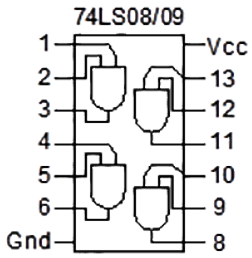
Quad 2-input NAND



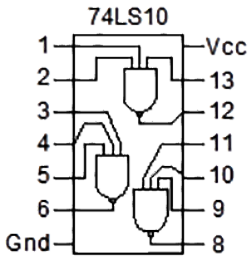
Quad 2-input NOR



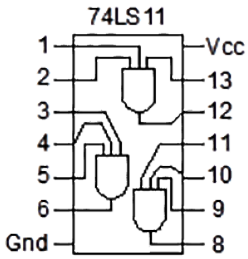
Hex INVERTER



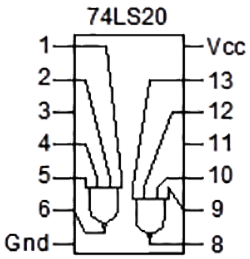
Quad 2-input AND



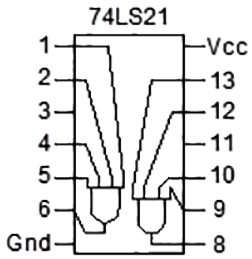
Triple 3-input NAND



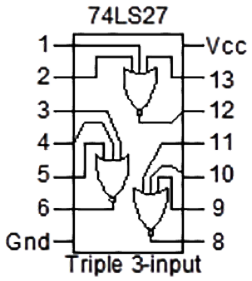
Triple 3-input AND



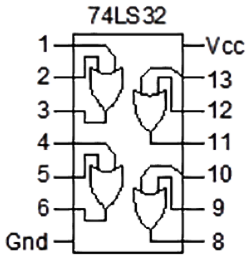
Dual 4-input NAND



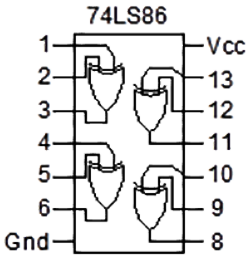
Dual 4-input AND



Triple 3-input NOR



Quad 2-input OR



Quad 2-input XOR

Chapter 3

Virtual Chips

When writing Verilog, it is important to think in terms of blocks of circuits that are wired together, as you do in Lab 1 with the discrete chips. In Verilog, there are two ways to express blocks of logic, each with its own rules for use. These are *modules* and *always blocks*.

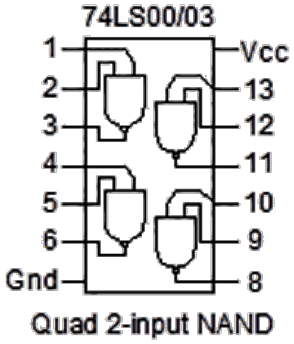
3.1 Modules

A *module* is the closest analogy to a discrete physical chip. It has explicit input and output pins, or in Verilog jargon, *ports*. Some ports can be bidirectional, meaning the port is sometimes an input and sometimes an output. Bidirectional ports are not discussed here. Consult the textbook if you think you need to use this capability.

There is always at least one module in any Verilog design, which is the top-level module encompassing the entire design. *Modules* can also be replicated, i.e., multiple copies of the same circuit can be *instantiated* into a larger circuit. Section 4.3 discusses this.

Here we show a module that defines the function equivalent to a 7400 chip.

Schematic



Verilog

Note that in Verilog, we do not worry about the power and ground connections.

```
module v7400 (input pin1, pin2, pin4, pin5, pin13,
                    pin12, pin10, pin9,
              output pin3, pin6, pin11, pin8);

   assign pin3 = !(pin1 & pin2);
   assign pin6 = !(pin4 & pin5);
   assign pin11 = !(pin13 & pin12);
   assign pin8 = !(pin10 & pin11);

endmodule // v7400
```

3.2 Always Blocks

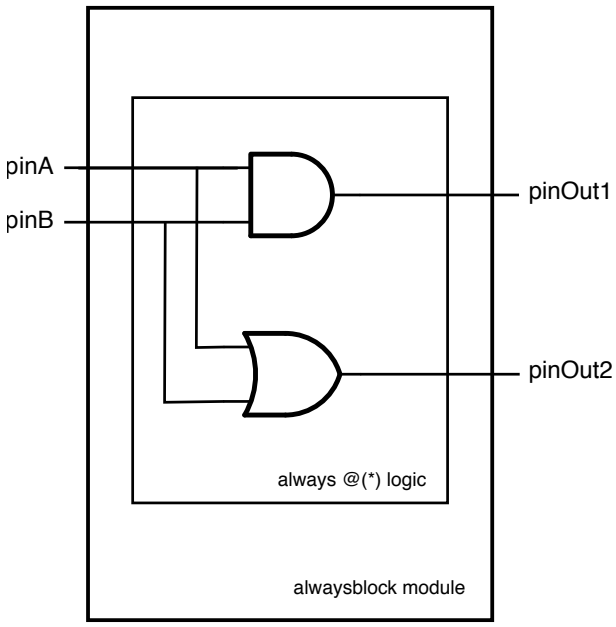
Like a *module*, the *always* block also represents a block of logic, but its use is different from modules. A *module* can be copied and used multiple times in your design, but not an *always* block. The *always* block encapsulates *procedural statements*, which are simulated sequentially, i.e., you must read the statements inside the always block in the order they are written to achieve the correct behaviour. Some constructs in Verilog must be described inside an *always* block.

Since an *always* block describes a block of logic, it also has inputs and outputs. The sensitivity list of the *always* block represents the inputs of the block. For combinational logic, the sensitivity list should just be expressed as '*'. This will avoid issues with simulation not matching the synthesized logic. The explanation has to do with peculiarities of Verilog and will not be elaborated further here. The sensitivity list for sequential logic will be discussed in Chapter 6.

Unlike with *modules*, the outputs of an *always* block are not explicitly declared as outputs. Any symbol that appears on the left-hand side of an assignment expression in an *always* block must be declared as a `reg`, and that symbol can be used as an output of the block.

Note that the declaration `reg` has nothing to do with the *register* logic block. This is just another confusing feature of Verilog to remember.

Schematic



Verilog

```
module alwaysblock (input pinA, pinB,
                    output pinOut1, pinOut2);

   reg pinOut1, pinOut2

   always @(*)  // Use * for sensitivity list of
                // combinational logic
     begin
        // pinOut1 and pinOut2 are declared reg
        //   so they are outputs of this always block
        pinOut1 = pinA & pinB;
        pinOut2 = pinA | pinB;
     end

endmodule // alwaysblock
```

Chapter 4

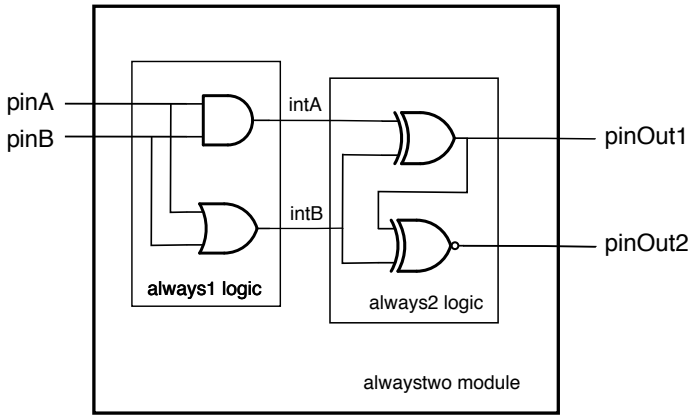
Building Bigger Circuits

4.1 Wiring Circuits Together

When connecting gates in chips together on the protoboard in Lab 1, you used physical wires. In Verilog, there is also the concept of wires, and thankfully you can declare a signal name as a `wire`. You can then use that wire to connect the output of one block to the input of another block. With *always* blocks, you can take an output of the block and connect it directly to an input without declaring the intermediate wire.

4.2 Using the Outputs of an Always Block

Schematic



Verilog

```
module alwaystwo (input pinA, pinB,
                  output reg pinOut1, pinOut2);
   reg  intA, intB;

   always @(*)  // always1
     begin
        // intA and intB are declared reg
        //   so they are outputs of this always block
        intA = pinA & pinB;
        intB = pinA | pinB;
     end

   always @(*)  // always2
     begin
        // pinOut1 and pinOut2 are declared reg
        //   so they are outputs of this always block
        pinOut1 = intA ^ intB;
        pinOut2 = intB ~^ pinOut1;
     end
endmodule // alwaystwo
```

4.3 Hierarchical Design

A *module* defines a block of logic, or a sub-circuit, that can be copied and used several times in the same overall circuit. This allows you to do hierarchical and incremental design. Hierarchy means that you can have modules instantiated inside other modules to whatever depth you need. The incremental design comes from being able to test modules individually and then assemble them into larger circuits having confidence that if there are problems, it is more likely due to interfacing problems between the modules than the internal behaviour of the individual modules. Once you have a library of working modules, designing larger circuits is easier because you can instantiate large blocks of logic assuming that they are already working. An example might be a module that describes an entire microprocessor. You can instantiate that microprocessor in your circuit and not have to worry (as much) whether it works properly. This is equivalent to buying a microprocessor chip from Intel and using it to build a new motherboard for your computer.

You will use *wires* to connect *modules* together.

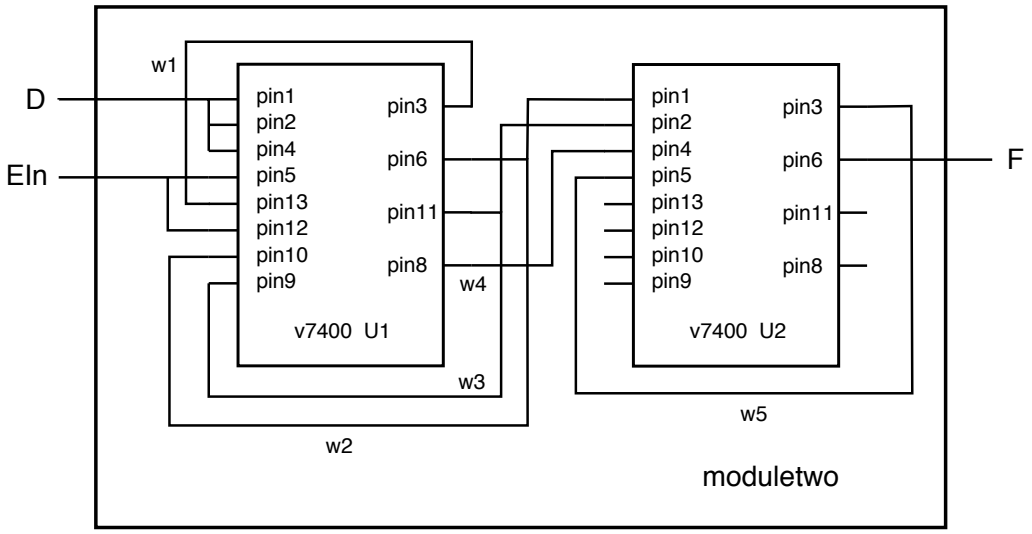
4.3.1 Module Instantiation

You should think about a *module* as the design, or template, for a chip. When designing with discrete chips, if I need six two-input NAND gates, then I will need to use two 7400 chips because each chip only provides four NAND gates. In a similar way, in Verilog, I can create a copy of the circuit described in a module by *instantiating* the module inside a higher-level module.

When instantiating a module you have to make a correlation between the ports of the module and the wires used connect to the ports of the instantiated module. There are two ways to do this. One is by the order of the wires specified in the instantiation. This means that that order specified in the instantiation must follow the order as defined in the module port list. The textbook uses the order approach, which is okay when you have a small number of ports, but is prone to error, such as mixing up the orders. Consider what might happen with a larger design where there might be hundreds of ports. To avoid this ordering problem, the latest Verilog standards use a *named* approach. The example code will show both approaches.

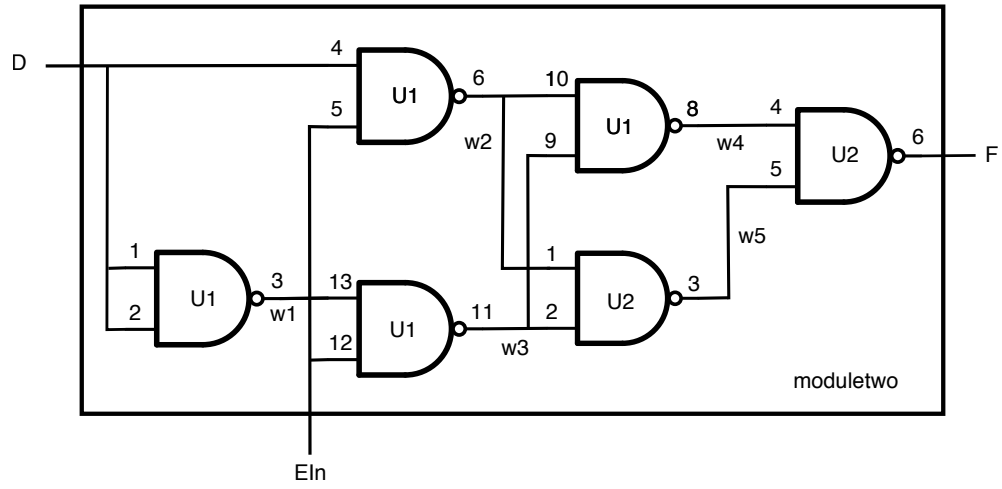
4.3.2 Connecting (Wiring) Modules Together

Schematic 1



Schematic 2

Sometimes drawing the modules as just blocks is not as meaningful. This version of the schematic is better if you want to understand the circuit functionality.



Verilog

```
module moduletwo (input D, EIn,
                  output F);

   // Declare some wires for making connections
   wire   w1, w2, w3, w4, w5;

   // Instantiate the first module using ordered ports
   // Look at the schematic and the 7400 chip/module
   // description to understand the ordering
   v7400 U1(D, D, D, EIn, w1, EIn, w2, w3, w1, w2, w3, w4);

   // Instantiate the second module using named ports
   // You do not have to worry about ordering
   // Note that not all inputs are used since only
   // two gates are needed
   v7400 U2(.pin1(w2),.pin2(w3),.pin3(w5),.pin4(w4),
     .pin5(w5),.pin6(F));

endmodule // moduletwo


module v7400 (input pin1, pin2, pin4, pin5, pin13, pin12,
                    pin10, pin9,
      output pin3, pin6, pin11, pin8);

   assign pin3 = !(pin1 & pin2);
   assign pin6 = !(pin4 & pin5);
   assign pin11 = !(pin13 & pin12);
   assign pin8 = !(pin10 & pin11);

endmodule // v7400
```

Chapter 5

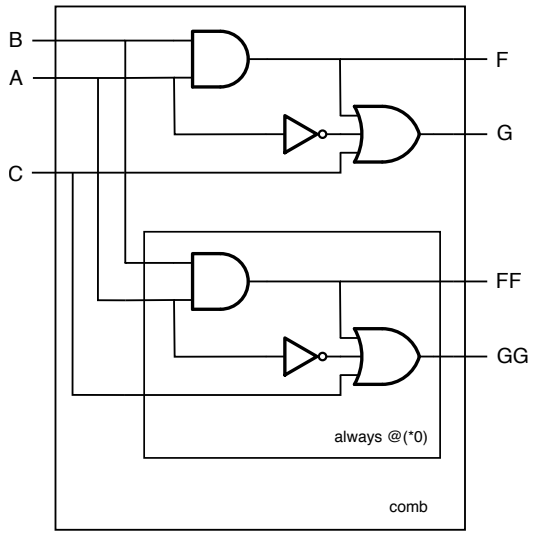
Combinational Logic

Combinational logic is any logic where the output is strictly a function of the inputs. Verilog has various ways to express combinational logic from basic logic expressions to larger constructs, such as multiplexers. How you express them depends on whether you do it inside or outside of an *always* block. Inside an *always* block you use *procedural* statements and outside an *always* block you use *continuous assignment* statements. Some constructs can only be used inside *procedural* statements.

5.1 Logic Functions

Logic functions describe a number of gates that operate on input signals to form a logic output. There are numerous logic operators, such as *and*, *or* and *not*. The textbook describes all the Verilog operators in Section 4.6.5 and Appendix A.7.

Schematic



Verilog

```
module comb (input A, B, C,
             output F, G, FF, GG);
   // These are used in the always block
   reg   FF, GG;

   // Logic functions using continuous assignment statements

   assign F = A & B;
   assign G = F | ~A | C;

   // Logic functions using procedural statements

   always @(*)
     begin
        FF = A & B;
        GG = FF | ~A | C;
     end

endmodule // comb
```

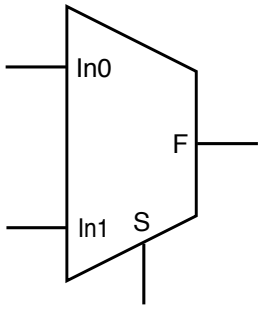
5.2 The Multiplexer

The output of a multiplexer selects one of the N inputs according to the selection value S. A multiplexer can be expressed using conditional statements, *if* statements, or *case* statements. Conditional and *if* statements are best used for just 2-input multiplexers. While larger multiplexers can also be expressed with conditional and *if* statements, it requires using nested statements, which are not easy to read and understand. Larger multiplexers are most clearly expressed using *case* statements.

5.2.1 2:1 Multiplexers

Schematic

The logic with outputs **F**, **FP**, **FI** and **FC** in the Verilog module are all 2:1 multiplexers as shown here.



S	F
0	In0
1	In1

Verilog

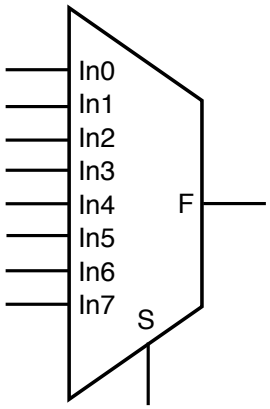
```
module mux2 (input In0, In1, S,
             output F, FP, FI, FC);

   // These are used in the always block

   reg   FP, FI, FC;

   // 2:1 mux using continuous assignment statements

   assign F = S ? In1 : In0;


   // Using procedural statements

   always @(*)  // mux2s
     begin
        // conditional procedural statement
        FP = S ? In1 : In0;

        // if statement
        if (S)
          FI = In1;
        else
          FI = In0;

        // case statement
        case(S)
          0: FC = In0;
          1: FC = In1;
          default: FC = In0:  // To avoid latches
        endcase // case (S)
     end // always @ (*)

endmodule // mux2
```

5.2.2 Larger Multplexers

Multiplexers larger than 2:1 are best expressed using the case statement.

Observe the use of the `default` statement in the case statement. This is to avoid another Verilog feature where simulations may not match the synthesized logic unless you use the *default* statement. The explanation is left for the lectures.

Schematic



S	F
0	In0
1	In1
2	In2
3	In3
4	In4
5	In5
6	In6
7	In7

Verilog

```
module mux8 (input In0, In1, In2, In3, In4, In5, In6, In7,
             input [2:0] S,
             output          F);
   // This is used in the always block
   reg    F;

   always @(*)  // mux8
     begin
        case(S)
          0: F = In0;
          1: F = In1;
          2: F = In2;
          3: F = In3;
          4: F = In4;
          5: F = In5;
          6: F = In6;
          7: F = In7;
          default: F = In0;  // To avoid latches
        endcase // case (S)
     end // always @ (*)

endmodule // mux8
```

5.3 Other Common Combinational Building Blocks

There are other building blocks such as encoders, decoders, and comparators that you may want to use. Consult the textbook for examples.

Chapter 6

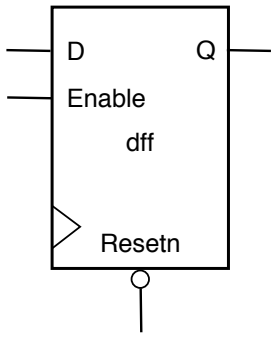
Sequential Logic

Sequential logic is logic that has state. The main sequential logic elements used are flip flops, registers and counters. The examples shown here will show all the code structures that you might use to do the ECE241/253 labs.

6.1 D Flip Flop

There are many types of flip flops and many configurations of those flip flops in terms of the clock polarity, the style of reset, and enables.

Schematic



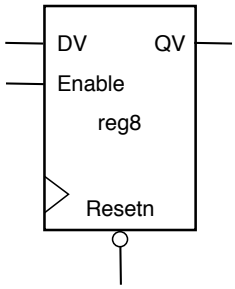
Verilog

```
module DFF (input D, Enable, Resetn, Clock,
            output Q);

   // Used in the always block
   reg  Q;

   // D flip flop
   // Synchronous reset
   // with enable
   always @(posedge Clock)
     begin
        if(!Resetn)
          Q <= 0;
        else if(Enable)
          Q <= D;
     end

endmodule // DFF
```

6.2 Registers

Observe the similarity to the D flip flop code.

Schematic



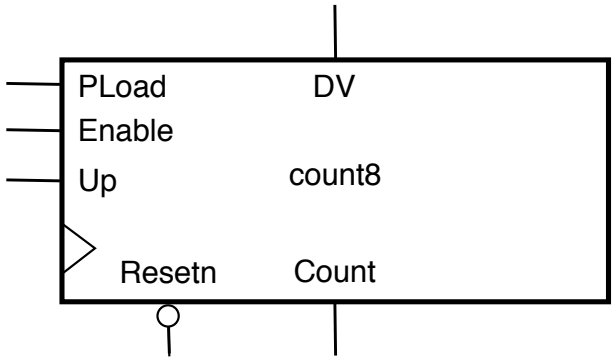
Verilog

```
module reg8 (input Enable, Resetn, Clock,
             input [7:0] DV,
             output [7:0] QV);

   // Used in the always block
   reg         QV;

   // 8-bit register
   // Synchronous reset
   // with enable
   always @(posedge Clock)
     begin
        if(!Resetn)
          QV <= 0;
        else if(Enable)
          QV <= DV;
     end
endmodule // reg8
```

6.3 Counters

There are many types of counters. The example here is an up/down binary counter with enable and pre-load features. Consult the textbook for other types of counters. This example shows most of the useful constructs you can use for building counters.

Schematic



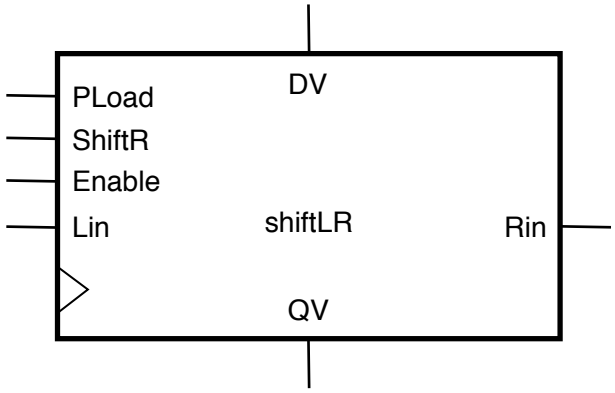
Verilog

```
module count8 (input Enable, Resetn, Clock, Up, PLoad,
               input [7:0] DV,
               output [7:0] Count);

   // Used in the always block
   reg         Count;

   // 8-bit up/down counter
   // Synchronous reset
   // Parallel load
   // with enable - no counting unless enabled
   always @(posedge Clock)
     begin
        if(!Resetn)
          Count <= 0;
        else if(PLoad) // Parallel load
          Count <= DV;
        else if(Up & Enable)  // Count up
          Count <= Count + 1;
        else if(Enable) // Count down
          Count <= Count - 1;
     end
endmodule // count8
```

6.4 Shift Registers

A shift register is a register where the bits are usually shifted one position either left or right. There are several ways to express the circuit and many possible features. The example here is simple to understand, but not the most compact way to express the behaviour. For the purposes of the ECE241/253 labs, you should strive for clear and understandable code that works versus the most compact code that may not work! The example shift register shown here has an enable and can shift either left or right.

Schematic



Verilog

```
module shiftLR (input Enable, Clock, PLoad, ShiftR, Lin, Rin,
                input [3:0] DV,
                output [3:0] QV);

   // Used in the always blocks
   reg [3:0]        QV, Count;


   // 4-bit shift left/right register
   // with enable
   // with parallel load
   always @(posedge Clock)
     begin
        if(PLoad)
          QV <= DV;
        else if(Enable & ShiftR) // Shift right
          begin
             QV[3] <= Lin;
             QV[2] <= QV[3];
             QV[1] <= QV[2];
             QV[0] <= QV[1];
          end
        else if(Enable) // Shift left
          begin
             QV[3] <= QV[2];
             QV[2] <= QV[1];
             QV[1] <= QV[0];
             QV[0] <= Rin;
          end
     end // always @ (posedge Clock)

endmodule // shiftLR
```

Chapter 7

Finite State Machines

In the broadest definition, any sequential circuit, such as a counter or shift register, is a finite state machine (FSM). However, when building digital systems we generally talk about an FSM as the important construct used for building control circuitry. While you can write code without an explicit FSM, you then rely on the synthesis tool to infer your intent, and it usually gets it wrong. It is better to make your intent explicit and clear. Your FSM code should always be written to explicitly show the standard parts of an FSM: The next-state logic, the state register and the output logic. Figure 6.5 in the textbook shows the generic structure of an FSM and your Verilog code should reflect the same structure.

The basic concepts of FSMs are described well in the textbook but there is a step from FSMs describing sequence detectors to using FSMs to control something more useful. In this chapter, we show some example uses of FSMs that you might encounter in ECE241/253.

7.1 Controlling a Datapath

To be completed.

7.2 Communicating FSMs

When you have larger circuits, it is usually better to have a number of smaller FSMs instead of one very large one. Smaller FSMs are easier to understand

and generally run faster than larger ones. However, to coordinate multiple FSMs, some communication between them is required.

To be completed.

Chapter 8

Arithmetic

When you need to do arithmetic in Verilog, it is possible to just use the operators '+', '-' and '*' for addition, subtraction and multiplication, respectively, instead of instantiating the respective circuit modules. Note that division is not included here because it is a very complex circuit. You should also be familiar with the concepts of bit widths, unsigned numbers, signed numbers using two's complement notation and sign extension. A short paper is attached that gives examples of how to write correct code using arithmetic expressions with signed numbers. You should follow the examples given for Verilog 2001.

8.1 Beware When Using Multiple Bit Widths

Verilog has rules about how to handle simple expressions when the operands and LHS of an expression have different widths. For example, consider the addition expression in the following code:

```
module addern (cin, A, B, S);
   parameter n = 4;
   input cin;
   input [n-1:0] A, B;
   output reg [2:0] S;

   always @(*)
     S = A + B + cin;

endmodule // addern
```

The LHS is three bits, `A` and `B` are four bits and `cin` is one bit. The Verilog rule for dealing with this is:

> Widen all operands and the LHS to the same width as the widest width using padding with 0's or sign extension for signed numbers. Do arithmetic in the widest width and then drop the most significant bits, if necessary.

The dropping of bits is where you may run into problems. For the above example, the widest operand is four bits, so the expression will be evaluated making all operands four bits, but the result is only three bits. You may get incorrect results because of this.

8.2 Signed Arithmetic in Verilog

Signed Arithmetic in Verilog 2001 – Opportunities and Hazards

Dr. Greg Tumbush, Starkey Labs, Colorado Springs, CO

Introduction

Starkey Labs is in the business of designing and manufacturing hearing aids. The new digital hearings aids we design at the Starkey Labs Colorado IC Design Center utilize very complex DSP algorithms implemented in both software and hardware accelerators. The predominant data type used in these algorithms is signed. The format of the signed type is two's complement. The designation of signed and two's complement is used interchangeably throughout this document.

Verilog 2001 provides a very rich set of new signed data types. However, there are issues when performing operations such as sign extension, truncation or rounding, saturation, addition, and multiplication with signed values. These new data types (in theory) free the designer from worrying about some of these signed data type issues. More compact and readable code should result. However, in the spirit of Verilog, usage of this new functionality is "user beware"! Arithmetic manipulation between mixes of signed and unsigned may simulate and synthesize in unintended ways. Assignments between differently sized types may also not result in what the designer intended. Does the usage of signed data types in arithmetic operations result in smaller or larger circuits?

Verilog 1995 provides only one signed data type, integer. The rule is that if any operand in an expression is unsigned the operation is considered to be unsigned. The rule still applies for Verilog 2001 but now all regs, wires, and ports can be signed. In addition, a numeric value can be designated with a 's similar to the 'h hex designation. Signed functions are also supported as well as the type casting operators *$signed* and *$unsigned*. There are many new rules about when an operation is unsigned, and some may surprise you!

In this paper I will provide code examples of how the new signed data types can be used to create more compact code if some simple rules are followed. RTL and gate level simulation results of add and multiply operations using mixtures of signed and unsigned data types will be provided. Area results from synthesis using Design Compiler 2003.12 will be presented to compare efficiencies of these operations. Synthesis warnings that should be investigated thoroughly will be explained. Suggestions for improvement in the Verilog 2001 language, such as saturation support, will also be provided.

Signed Data Types

Table 1 demonstrates the conversion of a decimal value to a signed 3-bit value in 2's complement format. A 3-bit signed value would be declared using Verilog 2001 as *signed [2:0] A;*.

Decimal Value	Signed Representation
3	3'b011
2	3'b010
1	3'b001
0	3'b000
-1	3'b111
-2	3'b110
-3	3'b101
-4	3'b100

Table 1: Decimal to 3-bit Signed

Type Casting

The casting operators, *$unsigned* and *$signed*, only have effect when casting a smaller bit width to a larger bit. Casting using *$unsigned(signal_name)* will zero fill the input. For example *A = $unsigned(B)* will zero fill B and assign it to A. Casting using *$signed(signal_name)* will sign extend the input. For example, *A = $signed(B)*. If the sign bit is X or Z the value will be sign extended using X or Z, respectively. Assigning to a smaller bit width signal will simply truncate the necessary MSB's as usual. Casting to the same bit width will have no effect other than to remove synthesis warnings.

Signed Based Values

The only way to declare a signed value in Verilog 1995 was to declare it as an integer which limited the size of the value to 32-bits only[1]. Verilog 2001 provides the *'s* construct for declaring and specifying a sized value as signed. For example, 2 represented as a 3-bit signed hex value would be specified as *3'sh2*. Somewhat confusing is specifying negative signed values. The value -4 represented as a 3-bit signed hex value would be specified as *-3'sh4*. A decimal number is always signed.

Signed Addition

Adding two values that are n-bits wide will produce a n+1 bit wide result. The signed values must be sign

extended. For example, adding -2 (3'b110) to 3 (3'b011) will result in 1 (4'b0001). See the example in Figure 1.

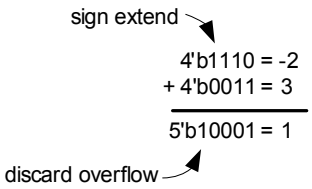


Figure 1: Basic Signed Addition Example

To do this addition using Verilog-1995 constructs we could use the code in Code Example 1.

```
module add_signed_1995 (
                        input [2:0] A,
                        input [2:0] B,
                        output [3:0] Sum
                        );
  assign                Sum = {A[2],A} + {B[2],B};
endmodule // add_signed_1995
```

Code Example 1: Addition - Verilog 1995

Or we can use the new signed type and get the code in Code Example 2.

```
module add_signed_2001 (
                        input signed [2:0] A,
                        input signed [2:0] B,
                        output signed [3:0] Sum
                        );
  assign                Sum = A + B;
endmodule // add_signed_2001
```

Code Example 2: Addition - Verilog 2001

Both adders are exactly the same size. So you will get the same results without having to worry about manually doing the sign extension.

Problems creep up when mixing signed and unsigned. Consider adding two 3-bit values with a 1-bit carry in. See Code Example 3 for a valid solution using Verilog 1995

```
module add_carry_signed_1995 (
                        input [2:0] A,
                        input [2:0] B,
                        input carry_in,
                        output [3:0] Sum
                        );
  assign                Sum = {A[2],A} + {B[2],B} + carry_in;
endmodule // add_carry_signed_1995
```

Code Example 3: Add with Carry - Verilog 1995

Intuitively we would create Code Example 4 to use signed types. However, when synthesized the following warning occurs: *signed to unsigned conversion occurs. (VER-318)* In addition there is a functional error. Due to the *carry_in* being unsigned the operation is unsigned and neither the *A* nor *B* operand is sign extended properly as in Figure 1.

```
module add_carry_signed_2001 (
                        input signed [2:0] A,
                        input signed [2:0] B,
                        input carry_in,
                        output signed [3:0] Sum
                        );
  assign                Sum = A + B + carry_in;
endmodule // add_carry_signed_2001
```

Code Example 4: Addition with Carry – Incorrect

We can avoid the synthesis warning by using *assign Sum = A + B + $signed(carry_in).* But this creates a different functional error. What happens if carry_in = 1? In this case the *$signed* operator sign extends the carry_in so it now equals 4'b1111 and we would have been subtracting 1 instead of adding 1. A similar functional error occurs if we use Code Example 4 but declare *carry_in* to be a signed input. See Code Example 5 for a valid solution. Using this code we avoid the synthesis warning and sign extend *carry_in* correctly with 0's.

```
module add_carry_signed_final (
                        input signed [2:0] A,
                        input signed [2:0] B,
                        input carry_in,
                        output signed [3:0] Sum
                        );
  assign      Sum = A + B + $signed({1'b0,carry_in});
endmodule // add_carry_signed_final
```

Code Example 5: Add with Carry - Correct

The code in Code Example 1 and Code Example 2 simulate the same with both RTL and gate level verilog. They are also the same size. The code in Code Example 3 and Code Example 5 simulate the same using both RTL and synthesized gate level verilog. They are also the same size. Code Example 4 is smaller in area but functionally incorrect.

Signed Multiplication

Multiplying two values that are n-bits wide will produce a 2n bit wide result. For example, multiplying -3 (3'b101) by 2 (3'b010) should result in -6 (6'b111010). The multiplier (second factor) is examined bit by bit right to left (least significant to most significant bit) to determine if the multiplicand (first factor) is to be added to the partial result. If so, the multiplicand is aligned so that the least significant bit is under the correct multiplier bit position. If

the multiplicand is negative it must be sign extended. However, if the MSB of the multiplier is 1, the multiplicand is actually subtracted. Recall that subtraction is the same as invert and increment. See the example in Figure 2.

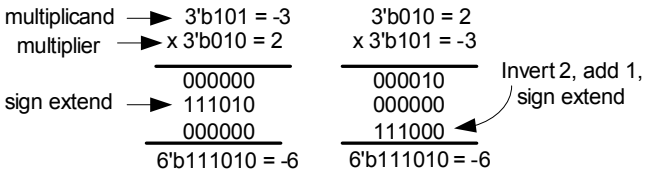


Figure 2: Signed Multiply Examples

Using Verilog-1995 constructs the code required to multiply two 3-bit signed values is in Code Example 6.

```
module mult_signed_1995 (
                input [2:0] a,
                input [2:0] b,
                output [5:0] prod
                );
  wire [5:0] prod_intermediate0;
  wire [5:0] prod_intermediate1;
  wire [5:0] prod_intermediate2;
  wire [2:0] inv_add1;
  assign     prod_intermediate0 = b[0] ? {{3{a[2]}}, a}       : 6'b0;
  assign     prod_intermediate1 = b[1] ? {{2{a[2]}}, a, 1'b0} : 6'b0;
  // Do the invert and add1 of a.
  assign     inv_add1 = ~a + 1'b1;
  assign     prod_intermediate2 = b[2] ? {{1{inv_add1[2]}},
                                          inv_add1, 2'b0} : 6'b0;
  assign     prod =  prod_intermediate0 +  prod_intermediate1 +
                                           prod_intermediate2;
endmodule
```

Code Example 6: Signed Multiply - Verilog 1995

Or we can use the new signed type and write the code in Code Example 7.

```
module mult_signed_2001 (
                input  signed [2:0] a,
                input  signed [2:0] b,
                output signed [5:0] prod
                );
   assign       prod = a*b;
endmodule
```

Code Example 7: Signed Multiply - Verilog 2001

Now, lets multiply a signed value by an unsigned value. Using Verilog 1995 constructs the code in Code Example 8 results. Now if we multiply –3 (3'b101) by 2 (3'b010) as usual with Code Example 8 we get –6 (6'b111010). When using a multiplier with one operand unsigned be sure of the range of input to the unsigned operand. If we tried to multiply 2 (3'b010) by –3 (3'b101) we would get 0xA because –3 is actually 5 unsigned. Note that because the multiplicand is unsigned this code is more compact and results in a smaller size multiplier.

```
module mult_signed_unsigned_1995 (
                input [2:0] a,
                input [2:0] b,
                output [5:0] prod
                );
  wire [5:0] prod_intermediate0;
  wire [5:0] prod_intermediate1;
  wire [5:0] prod_intermediate2;
  assign     prod_intermediate0 = b[0] ? {{3{a[2]}}, a}       : 6'b0;
  assign     prod_intermediate1 = b[1] ? {{2{a[2]}}, a, 1'b0} : 6'b0;
  assign     prod_intermediate2 = b[2] ? {{1{a[2]}}, a, 2'b0} : 6'b0;
  assign     prod =  prod_intermediate0 +  prod_intermediate1 +
                                           prod_intermediate2;
endmodule
```

Code Example 8: Signed by Unsigned Multiply - Verilog 1995

After migrating to Verilog 2001 we might be tempted to use Code Example 9. However, recall the rule that if any operand of an operation is unsigned the entire operation is unsigned. When synthesized the following warning occurs: *signed to unsigned conversion occurs. (VER-318).* Now if we multiply –3 (3'b101) by 2 (3'b010) as usual with this code we get 0xA (6'b001010). The reason for this is that since we mixed signed with unsigned we actually multiplied 5 by 2 and got 0xA since the operation is considered unsigned.

```
module mult_signed_unsigned_2001 (
                input  signed [2:0] a,
                input  [2:0] b,
                output signed [5:0] prod
                );
  assign       prod = a*b;
endmodule
```

Code Example 9: Signed by Unsigned Multiply - Incorrect

How about trying Code Example 10? This works for multiplying -2x3=-6 but what about if the MSB of our unsigned number = 1? In this case the multiplier is sign extended which is also incorrect. For the operation -2x7 we would get actually get 2 while the correct answer is 0x-E (6'b110010).

```
module mult_signed_unsigned_2001 (
                input  signed [2:0] a,
                input  [2:0] b,
                output signed [5:0] prod
                );
  assign       prod = a*$signed(b);
endmodule
```

Code Example 10: Signed by Unsigned Multiply - Still Incorrect

The correct answer to this problem follows from Code Example 5. Using this code we avoid the synthesis warning and sign extend *b* correctly with 0's. The correct code is in Code Example 11.

```
module mult_signed_unsigned_2001 (
                  input  signed [2:0] a,
                  input  [2:0] b,
                  output signed [5:0] prod
                  );
  assign        prod = a*$signed({1'b0,b});
endmodule
```

Code Example 11: Signed by Unsigned Multiply - Correct

Code Example 7 synthesizes to about 18% smaller than Code Example 6. I believe that this is because synthesis found a better implementation. There is no reason why we cannot replicate this size by more careful hand coding. The RTL and gate level implementation simulate the same.

Code Example 11 synthesizes to about 6% smaller than the code in Code Example 8. Once again, I believe that this is because synthesis found a better implementation. There is no reason why we cannot replicate this size by more careful hand coding. The RTL and gate level implementation simulate the same. The code in Code Example 9 and Code Example 10 are smaller in area but are functionally incorrect.

What is an expression?

The *Verilog-2001 LRM* states that to evaluate an expression "Coerce the type of each operand of the expression (excepting those which are self-determined) to the type of the expression"[2]. The question is what is an expression? Consider Code Example 12 which is directly from a Synopsys's SolvNet article 002590[3]. There are two ways to look at this code. It could be considered as two expressions, a signed multiply and then an unsigned addition. It can also be considered as one expression, an unsigned multiply followed by an unsigned addition. Results will differ in each case.

```
module mult_add (
            input signed [3:0] in1, in2,
             input [3:0] in3,
             output [7:0] o1;
              );
assign o1 =  in1 * in2 + in3;
endmodule
```

Code Example 12: Multiply and Add

It was reported that older versions of Design Compiler considered this code as two expressions while some simulators at the time considered it as one. Newer version of Design Compiler and ModelSim consider this code as one expression, alleviating a very worrisome simulation/synthesis mismatch. This issue is slated to be clarified in the upcoming Verilog 2005 LRM.

Rules for Expression Types

Located in the Verilog 2001 LRM but worth repeating here are the rules for determining the resulting type of an expression. The following operations are unsigned regardless of the operands.

1. Bit-select results
2. Part-select results, even if the entire vector is selected.
3. Concatenation results
4. Comparison results

Signed Shifting

Shifting of signed values creates another problem for Verilog 1995. Consider a signed negative value that is right shifted. The positions vacated by the right shift will be filled in with zeros which is incorrect. Instead, the sign bit should be used for vacated bits. A new operator >>> is introduced in Verilog 2001 to accomplish exactly this. A signed left shift operator (<<<) is also provided for language consistency[1].

Signed Saturation

In this section we present a concept that is widely used in DSP math but is not easily accomplished in Verilog. While sign extension is used when assigning a smaller bit-width variable to a larger bit-width variable, the opposite is accomplished using saturation. The possible outcomes of saturation are max positive indicating positive overflow, max negative indicating negative underflow, and simply dropping the appropriate number of bits starting at the MSB.

Saturation is accomplished by examining the number of bits to saturate plus 1 starting at the MSB. If all of these bits are the same drop the number of bits to saturate. If these bits are different examine the MSB. If the MSB is 0 go to max positive, else go to max negative. A module *sat* to accomplish this is in Code Example 13. Usage of this module is in Code Example 14.

```
module sat (sat_in, sat_out);

parameter IN_SIZE = 21;  // Default is to saturate 22 bits to 21 bits
parameter OUT_SIZE = 20;
input      [IN_SIZE:0] sat_in;
output reg [OUT_SIZE:0] sat_out;

wire [OUT_SIZE:0]   max_pos = {1'b0,{OUT_SIZE{1'b1}}};
wire [OUT_SIZE:0]   max_neg = {1'b1,{OUT_SIZE{1'b0}}};

always @* begin
   // Are the bits to be saturated + 1 the same?
   if ((sat_in[IN_SIZE:OUT_SIZE]=={IN_SIZE-OUT_SIZE+1{1'b0}}) ||
     (sat_in[IN_SIZE:OUT_SIZE]=={IN_SIZE-OUT_SIZE+1{1'b1}}))
           sat_out = sat_in[OUT_SIZE:0];
    else if (sat_in[IN_SIZE])  // neg underflow.  go to max neg
           sat_out = max_neg;
    else // pos overflow, go to max pos
           sat_out = max_pos;
  end
endmodule
```

Code Example 13: Saturation Module

```
wire signed [4:0] A, B, C;
reg signed [2:0] D, E, F;

A = 5'sb11101;
B = 5'sb01001;
C = 5'sb10001;

// Drop two MSB's. D will equal 3'sb101
sat  #(.IN_SIZE(4), .OUT_SIZE(2))  satA (.sat_in(A), .sat_out(D));

// Go to max positive.  E will equal 3'sb011
sat  #(.IN_SIZE(4), .OUT_SIZE(2))  satB
                (.sat_in(B), .sat_out(E));

// Go to max negative.  F will equal 3'sb100
sat  #(.IN_SIZE(4), .OUT_SIZE(2))  satC
                (.sat_in(C), .sat_out(F));
```

Code Example 14: Use of Saturation Module

Summary

This paper strove to give the user a strong background on the use of signed types using the Verilog 2001 language. The proper use of type casting, addition, multiplication, shifting, and truncation was presented. In addition, an example of a signed saturation module along with examples of it's use were included.

Proper use of the new signed capability in Verilog 2001 can be summarized by a few basic rules.

1. If any operand in an operation is unsigned the entire operation is unsigned[2].
2. Investigate fully all *signed to unsigned conversion occurs. (VER-318)* synthesis warnings. These point to incorrect functionality
3. All signed operands will be signed extended to match the size of the largest signed operand.
4. Type casting using *$unsigned* will make the operation unsigned. The operand will be sign extended with 0's if necessary.
5. Type casting using *$signed* make the operand signed. The operand will be sign extended with 1's if necessary. Pad the operand with a single 0 before the cast if this is not desired.
6. Expression type depends only on the operands or operation, it does not depend of the LHS of the expression.

References

1. S. Sutherland. *Verilog 2001 A Guide to the New Features of the Verilog Hardware Description Language.* Kluwer Academic Publishers

2. IEEE P1364-2005/D3. *Draft Standard for Verilog® Hardware Description Language.*

3. Synopsys Inc, *Synopsys Solvnet,* solvnet.synopsys.com