

# Practical Logic and Processor Design with Verilog

Spring 2022.0 Edition

Jay Brockman

Copyright © Jay Brockman, 2018-2022

## Version History

Spring 2021.1	Rewrote sections of Chapter 3 on albaCore to better follow lectures and provide more detailed explanations and examples.
Spring 2021.0	<p>Added new chapter on Programmer's View of a Computer (Chapter 2)</p> <p>Created separate chapter on albaCore Microprocessor: Instructions and Programming (Chapter 3), including new material on using the albaCore assembler and simulator</p> <p>Fixed various other typos from earlier versions.</p>
Spring 2022.0	<p>Chapter 3: Added summary table of albaCore instruction</p> <p>Chapter 6: Added an introductory section on behavioral modeling and moved section on event-driven simulation to the end of the chapter</p> <p>Chapter 7: Changed my_dff_tb to uses port names rather than port order in uut instance</p> <p>Chapter 13: Wrote section on albaCore system and moved Verilog example of system to front of chapter with added detail. Renamed alu_op signal to match figure.</p>

# Contents

1	Introduction.....	9
1.1	About This Document.....	9
1.2	References.....	10
2	Programmer's View of a Computer.....	11
2.1	What is a Computer?.....	11
2.2	Binary Representation of Data in C .....	12
2.3	Bit-Level Boolean Operations in C.....	15
2.4	Memory Organization in C .....	16
2.4.1	Memory Addresses and Pointers.....	16
2.4.2	Arrays.....	17
2.4.3	Byte Order and Type Casting.....	20
2.4.4	Memory Segments: Text, Data, Stack, and Heap .....	22
2.5	Assembly Language and Machine Code.....	24
2.5.1	Original C Program .....	24
2.5.2	Assembly Language Program .....	25
2.5.3	Machine Code for Executable Program .....	27
2.5.4	A Program that Prints Its Own Machine Code.....	28
3	albaCore Microprocessor: Instructions and Programming .....	30
3.1	Basics of Computer Architecture and Organization .....	30
3.2	The albaCore Instruction Set.....	31
3.2.1	Overview: Memory, Registers, and Instructions.....	31
3.2.2	albaCore Assembler and Simulator.....	33
3.2.3	Arithmetic/Logic Instructions .....	34
3.2.4	Data Transfer Instructions.....	37
3.2.5	Branch Instructions .....	44
3.2.6	Jump Instructions .....	47
3.3	Writing Longer Assembly Language Programs.....	49
3.3.1	Using C goto Statements to Structure Assembly Programs .....	49
3.3.2	Maxfinder.....	50
3.4	Function Calls .....	52
3.4.1	Simple Function Call with Parameters Passed via Registers .....	52
3.4.2	The Call Stack.....	53
3.4.3	Example: Using Stack Alone .....	56
3.4.4	Example: Modern RISC Approach Using Registers and Stack .....	59
3.4.5	Recursive Summation .....	63

4	Logic and Processor Design Technology.....	64
4.1	Form and Function.....	64
4.2	Using Switches to Build Logic Functions.....	65
4.3	CMOS Logic.....	67
4.4	Evolution of Digital Circuit Technology .....	72
4.4.1	Discrete Transistors.....	72
4.4.2	Small Scale Integrated Circuits and the 7400 Series Logic Chips .....	72
4.4.3	Microprocessors and VLSI .....	72
4.4.4	Programmable Logic Devices .....	74
4.5	Evolution of Digital Logic Design Methodology .....	76
4.6	Terasic Altera DE2-115 Board .....	77
5	Getting Started with Verilog .....	80
5.1	Hardware Description vs. Programming Languages .....	80
5.2	Modules.....	80
5.3	Expressing Behavior Using the <code>assign</code> Statement.....	81
5.4	Expressing Structure: Connecting Components.....	89
5.4.1	Modules, Instances, Ports, and Wires .....	89
5.4.2	Instantiating and Connecting Modules by Port Name .....	89
5.4.3	Instantiating and Connecting Modules by Port Order.....	94
5.4.4	Built-In Primitive Logic Gates.....	94
5.5	Simulating Module Behavior Using a Testbench .....	96
5.5.1	Testing Real Hardware.....	96
5.5.2	Generating Test Inputs in Verilog: <code>initial</code> Procedure Block and <code>reg</code> Data Type.....	98
5.5.3	Connecting a Unit-Under-Test (UUT) to the Signal Generator .....	100
5.6	Multibit Signals.....	101
5.6.1	Vector Notation.....	101
5.6.2	Representation of Numbers.....	102
5.6.3	Arithmetic and Relational Operators .....	103
5.6.4	Reduction Operators .....	104
5.6.5	Rearranging Bits with the Concatenation Operator .....	105
5.7	Example: Ripple Carry Adder.....	107
5.7.1	Full Adder Using <code>assign</code> Statements .....	107
5.7.2	Structural Implementation.....	108
5.7.3	Testbench and Simulation Results .....	108
5.8	Design Flow: Compilation and Simulation.....	109
5.8.1	Design Flow Overview .....	109

5.8.2	Example: 5-Input OR Gate .....	111
5.8.3	Example: 4-Bit Adder .....	116
6	Behavioral Modeling of Combinational Logic .....	120
6.1	First Look at <code>always</code> Block .....	120
6.2	Multiplexor Using <code>if else</code> Statement .....	121
6.3	Comparator Using Nested <code>if else</code> Statements.....	121
6.4	Multiplexor Using <code>case</code> Statement .....	123
6.5	4-Bit Wide Multiplexor Using <code>case</code> Statement .....	124
6.6	Decoder Using <code>case</code> Statement.....	125
6.7	Majority Gate Bit Concatenation and <code>case</code> Statement.....	125
6.8	7-Segment Hex Digit Display Decoder.....	126
6.9	Demultiplexor: A Really Important Example to Understand! .....	128
6.10	Demultiplexor the Wrong Way: A Common Mistake to Avoid!.....	131
6.11	Putting It All Together: Router .....	133
6.12	Event-Driven Simulation: <code>always, assign, initial</code> .....	137
6.12.1	Step-by-Step Example.....	137
6.12.2	Modeling Circuit Delay .....	140
7	Sequential Logic .....	142
7.1	D Flip Flop .....	143
7.1.1	Modeling Edge-Triggered Behavior .....	143
7.1.2	Synthesized RTL Netlist .....	144
7.1.3	Testbench with Clock.....	144
7.2	D Flip Flop with Reset .....	145
7.3	D Flip Flop with Load Enable.....	148
7.4	Power Up Values .....	149
7.5	Multibit Registers.....	150
7.6	Counters .....	152
7.6.1	Incrementing Counter with Reset and Enable.....	152
7.6.2	Loadable Decrementing Saturating Counter.....	153
7.7	Frequency Division by a Power of 2.....	154
7.8	Timers .....	155
7.9	Square Wave Generator .....	156
7.10	Shift Register .....	158
7.11	Pseudo-Random Number Generation Using a Linear-Feedback Shift Register .....	159
7.12	Edge Detector.....	162
8	Verilog Signal Assignment .....	164

8.1	Blocking vs. Non-Blocking Signal Assignment .....	164
8.2	Good: Blocking Assignment Operator ( = ) and Combinational Logic .....	165
8.3	Bad: Blocking Assignment Operator ( = ) and Clocked Logic .....	165
8.4	Good: Non-Blocking Assignment Operator ( <= ) and Clocked Logic .....	167
9	Memory.....	169
9.1	RAM: Random Access Memory.....	169
9.1.1	Structure and Operation .....	169
9.1.2	Structural Verilog Implementation .....	171
9.1.3	Behavioral Verilog Implementation Using Arrays .....	174
9.1.4	Reading Memory Synchronous with Clock .....	177
9.1.5	Why You Should Use Synchronous Memory .....	178
9.2	ROM: Read-Only Memory .....	180
9.2.1	Using <b>initial</b> Block to Initialize ROM Contents.....	180
9.2.2	Using <b>for</b> Loop to Initialize ROM Contents.....	182
9.3	Using \$readmemh and \$readmemb to Initialize Memory from a File .....	183
9.4	An X-Y Image RAM.....	184
9.5	Multiport X-Y Image Memory for VGA Applications .....	185
9.6	Altera Parametrized Memory Module Library .....	187
10	Finite State Machines.....	190
10.1	Example State Machine Diagram and Equations.....	190
10.2	Low-Level Implementation: Combinational Logic and State Register as Separate Modules...	192
10.2.1	Structural Interconnection.....	192
10.2.2	State Register .....	193
10.2.3	Next State and Output Combinational Logic .....	194
10.3	Low-Level Implementation: Combinational Logic as <b>assign</b> Statements.....	195
10.4	Preferred High-Level Implementation: Combinational Logic as <b>case</b> Statement .....	196
10.5	Simulating FSM Behavior .....	198
10.6	DE2-115 Board Single-Stepping Demonstration.....	199
11	Register-Transfer Level (RTL) Design.....	200
11.1	A Simple Example .....	200
11.1.1	Example Datapath Design.....	200
11.1.2	Example: Calculating Additive Inverse .....	202
11.1.3	Example: Countdown.....	207
11.1.4	DE2-115 Board Single-Step Demonstration.....	212
11.2	Memory Interface: Maxfinder.....	214
11.2.1	Overview and HLSM .....	214

11.2.2	Datapath Design.....	215
11.2.3	Controller Design.....	219
11.2.4	Complete Processor: Connecting Controller and Datapath.....	221
11.2.5	Complete System: Connecting Processor to Memory.....	222
11.2.6	Testbench and Simulation Results .....	223
11.3	Animation Example: Obstacle Course .....	225
11.3.1	Overview and HLSM .....	225
11.3.2	Datapath Design.....	227
11.3.3	Controller Design.....	230
11.3.4	Complete Processor: Connecting Controller and Datapath.....	235
11.3.5	Image RAM .....	237
11.3.6	Complete System: Connecting Processor to Image RAM and VGA Controller .....	237
11.3.7	Simulating and Debugging Image Memory .....	239
12	Using the Terasic/Altera DE2-115.....	242
12.1	Top-Level Module Ports and Pin Definitions .....	242
12.2	Switches, Buttons, LEDs, and 7-Segment Displays .....	243
12.3	Switch Debouncing.....	244
12.3.1	Theory .....	244
12.3.2	Debouncing Circuit Design.....	245
12.3.3	Demo Using Terasic DE2-115 Board Pushbutton .....	246
12.4	LCD Display .....	247
12.4.1	Displaying a Static Text Message.....	248
12.4.2	Mixing Formatted Text and Data.....	250
12.4.3	Changing Location of Displayed Characters .....	252
12.4.4	Storing Message in a RAM .....	254
12.5	VGA Monitor.....	258
12.5.1	The VGA XY Controller.....	258
12.5.2	Display Image from ROM .....	259
12.5.3	bmp2mem: Converting BMP Files to readmemh Files.....	261
12.5.4	Animation Strategies.....	262
12.5.5	Raster Graphics: Plotting Individual Pixels in an Image RAM .....	262
12.5.6	Sprite Graphics: Switching Between Multiple Images .....	267
12.6	PS/2 Keyboard .....	270
12.6.1	Scan Codes.....	270
12.6.2	Packet Format .....	270
12.6.3	Serial-to-Parallel Conversion Using a Shift Register.....	271

12.6.4	Complete Keycode Recognizer.....	273
12.6.5	Buffering a Keypress Signal Until You Need It .....	278
12.6.6	Example: Typing Characters from Keyboard to LCD Display.....	279
12.7	Audio.....	284
12.7.1	University of Toronto Audio Controller .....	284
12.7.2	University of Toronto Audio and Video-In Configuration Module.....	287
12.7.3	Square Wave Synthesis Demo .....	288
12.7.4	Microphone to Line-Out Demo.....	290
13	albaCore Microprocessor: Hardware Implementation .....	293
13.1	albaCore System and Instruction Set Review .....	293
13.2	Verilog Implementation of the Top-Level albaCore System .....	295
13.2.1	Complete System: Connecting Processor and Memory .....	295
13.2.2	Loading a Program into Memory .....	296
13.2.3	Testbench and Simulation.....	298
13.2.4	Memory-Mapped Input/Output.....	300
13.3	5-Stage RISC Instruction Processing Cycle.....	304
13.4	HLSM .....	304
13.5	Datapath Design.....	306
13.5.1	Identifying Sources for Variables and Defining Flags.....	306
13.5.2	Arithmetic Logic Unit (ALU) .....	307
13.5.3	Instruction Fetch and Decode .....	309
13.5.4	Register Fetch and Instruction Execution .....	309
13.5.5	Load/Store Instruction Memory Stage .....	310
13.5.6	PC Update and Branch/Jump Instruction Execution.....	311
13.5.7	Write Back .....	311
13.5.8	Complete Datapath.....	312
13.5.9	Verilog Implementation of Datapath .....	312
13.6	Controller Design.....	315
13.6.1	HLSM with Control Inputs and Flags.....	315
13.6.2	Verilog Implementation .....	317
13.7	Complete Processor: Connecting Controller and Datapath.....	321
14	Appendix: Method of Complements.....	323
14.1	10's Complement Representation for Single-Digit Numbers .....	323
14.1.1	Implementing Subtraction with Addition.....	324
14.1.2	Overflow with Single-Digit 10's Complement Addition.....	324
14.2	10's Complement of Multidigit Numbers .....	325

14.2.1	Overflow with Multidigit 10's Complement Addition .....	325
14.2.2	Calculation Trick: 10's Complement as 9's Complement + 1 .....	326
14.3	Binary Numbers and 2's Complement.....	327
14.3.1	Representation.....	327
14.3.2	Calculation Trick: 2's Complement as 1's Complement + 1 .....	327
14.3.3	Conversion Rules Using the Invert + 1 Trick .....	328
14.3.4	Overflow in 2's Complement Addition.....	328
15	Appendix: Verilog Language Rules.....	330

# 1 Introduction

## 1.1 About This Document

This is an evolving, living document that introduces the practice of digital logic design using the Verilog hardware description language in the context on an introductory logic design course, CSE 20221 at the University of Notre Dame. It is based on the lecture notes and examples developed by the author over the past couple of years. The Verilog HDL has many language features that can be confusing to beginners. Further, many of the available books on Verilog assume that the reader is already familiar with the basics of logic design. This book is not a comprehensive Verilog guide or manual—good examples of those are available elsewhere. The goal of this book is to introduce “just enough” Verilog in a methodical way so that it can serve as a useful tool while learning how to design digital systems.

**A bit about the author:** I first began designing digital circuits as an undergrad at Brown University in the 1970s and was actually a TA for the digital design lab my senior year. After graduation, I worked as a product engineer at Intel for the first half of the 1980s, primarily in design, test, and manufacturing of chips that were the forerunners of today’s flash memory. From there, I went back to grad school at Carnegie Mellon University where I got my Master’s and Ph.D. degrees in Electrical and Computer Engineering in the field of computer-aided design and simulation of integrated circuits. I joined the newly-formed Department of Computer Science and Engineering at the University of Notre Dame as a faculty member in 1992, where I’ve taught courses in digital logic design, computer architecture, VLSI design (and many others, including software courses) ever since. I have worked on projects with many computer hardware companies, including Intel, IBM, HP, Motorola, Samsung, and more. I co-founded Emu Technology (now Lucata, Inc.) with Peter Kogge of Notre Dame and Ed Upchurch of Caltech to commercialize a novel memory-side processor technology for high-performance, big data analytics that grew from our research. We opened our first office in South Bend, Indiana in 2009 and have been growing since.

I’ve been designing digital systems for around 40 years, and while the students that I teach today are generally quicker than I am, I make up for that with experience, especially the experience of getting burned when I’ve cut corners. You will find many examples of digital hardware descriptions in Verilog out there on the web, some better than others and some downright awful. It can be difficult for a novice to recognize the difference between a good and bad design. The approach that I lay out in this book is based on the way that I really design things, building on what I’ve learned from other experts in both academia and industry and adding my own twist on some things. It’s not the only approach and there are alternative ways of achieving the same ends, but it has worked for me and I’m happy to pass it along to you. For someone getting started with this book, I strongly recommend that you stick with this approach and master it, understanding its strengths (and identifying its weaknesses), before considering a different style.

**Why I’m making this book available for free:** I have great respect for traditional textbook publishers. They provide a valuable service in finding good authors, helping them edit their material with extensive advice from reviewers, and then marketing and selling the product and paying the author royalty fees. For many authors who may spend years writing a textbook, this is the only way that they can be compensated for their efforts. For this reason, I’m a strong advocate of buying new textbooks when they exist (and keeping them for future reference), since this is the only way that the authors get paid. I’ve written a conventional textbook, *Introduction to Engineering: Modeling and Problem Solving*, published by John Wiley & Sons. It was a five-year labor of love and while very taxing, all-in-all it was a very good experience. While that book was a very intentional effort, this book snuck up on me. It began as a set of class notes and it was only after I’d written more than 100 pages that I began to see it as a book. At this stage, I don’t have the patience to go through a full editing cycle with a publisher—even though the

book would benefit greatly from it—nor am I concerned about royalties, so I've decided to just make the book available for free in the hope that readers would find it useful. Note that the book is still copyrighted, however, and you may not reproduce material from this book in either print or electronic form without proper citation for any purpose.

## 1.2 References

### Textbooks

Verilog has a lot of language features and there are many different styles of writing Verilog models in use. IMHO, the best digital logic design textbooks that incorporate Verilog are those written by Mike Ciletti. He has a very disciplined approach to writing Verilog that is less error-prone and easier to follow than many other approaches. My own approach to Verilog is mostly based on Ciletti's.

- **M. Morris Mano and Michael D. Ciletti. *Digital Design, 6<sup>th</sup> Edition. Pearson.*** Excellent introductory textbook with an introduction to the Verilog, VHDL, and SystemVerilog hardware description languages. Covers most of the same topics as the Vahid Digital Design zyBook with detailed explanations and lots of examples. The coverage of Verilog itself is not as deep as his other books.
- **Michael D. Ciletti. *Advanced Digital Design with the Verilog HDL, 2<sup>nd</sup> Edition. Pearson.*** Very comprehensive text. If you're serious about becoming a digital system designer, consider getting a copy of this.
- **Michael D. Ciletti. *Starter's Guide to Verilog 2001. Pearson.*** Good language reference that incorporates Ciletti's disciplined approach.

### Online Resources

- **ASIC World's Verilog Pages:** <http://www.asic-world.com/verilog> Great set of tutorials, examples, and pointer to other resources. Very highly recommended—I'd view this as required reading.
- **HDL Works Verilog Reference Guide:** [https://www.hdlworks.com/hdl\\_corner/verilog\\_ref](https://www.hdlworks.com/hdl_corner/verilog_ref) Clickable online language reference manual, seems to be mostly based on the 1995 standard but still extremely useful.

### Altera Documentation and Resources

- **LPM Library Quick Reference Guide:**  
[https://www.altera.com/en\\_US/pdfs/literature/catalogs/lpm.pdf](https://www.altera.com/en_US/pdfs/literature/catalogs/lpm.pdf)
- **Resources from Dr. John Loomis, University of Dayton**  
<http://www.johnloomis.org/>  
Great set of resources under the links “Altera DE2 Board” and “DE2 Digital Lab Projects”. I use his LCD display interface.
- **Resources from Jonathan Rose, University of Toronto, ECE241**  
[http://www.eecg.utoronto.ca/~jayar/ece241\\_06F/](http://www.eecg.utoronto.ca/~jayar/ece241_06F/)  
Lots of stuff at this link. Hasn't been updated since 2006, but IMHO still sets a high bar for what's possible in a project-oriented introductory Digital Logic Design course. I use his VGA display materials.

## 2 Programmer's View of a Computer

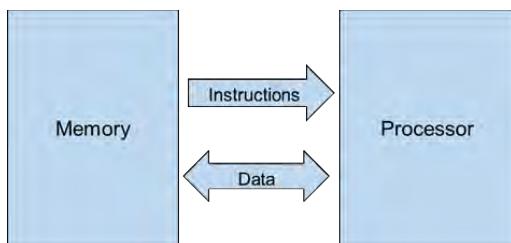
### 2.1 What is a Computer?

The main purpose of this course is to get a basic understanding of how a computer works from the hardware perspective. The term “computer” itself can be difficult to pin down, and its definition has evolved over time. The first recorded use of the term “computer” dates back to the 1600’s to refer to *people* who perform calculations, a usage that continued through the 1940s and 1950s as typified by the women who performed calculations for the space program portrayed in the movie *Hidden Figures*. Today, we define a computer as a *machine* that processes information. The design of computing machines has changed greatly over time, ranging from mechanical calculators to quantum computers. The focus of this course is on the inner workings of the computing machine that is ubiquitous today: the programmable digital electronic computer.

- **programmable:** The computer executes a sequence of encoded instructions called a *computer program* stored along with data in memory.
- **digital:** Both instructions and data are encoded as sequences of the binary digits or *bits* 1 and 0, and we use the principles of Boolean logic to define operations on data. *Logic Design* is the process of using Boolean logic to design digital computer circuits.
- **electronic:** The machinery that interprets the computer and executes a computer program and that operates on the data is built from electronic switches called *transistors*.

As with other types of systems, ranging from airplanes to social media platforms, engineers use two key principles to manage the complexity of designing computer hardware: modularity and abstraction. *Modularity* is the ability to break down a complex system into individual modules or components that can be designed and tested separately before integrating them together. *Abstraction* is the process of generalizing a particular case or class of cases such that we can describe a component by its interface and behavior without worrying about the internal details of its implementation. We can think of modularity and abstraction as tools for managing the breadth and the depth of a complex system: at the top level of abstraction, we think of the system as a set of interconnected, high-level modules, where the insides of each of these components are composed of a set of lower level modules, continuing on down in a hierarchy of layers until we get to a concrete set of primitive components.

At the top level of abstraction, we can think of a computer as a system with two components: a *processor* and a *memory*.



The memory is a large array that contains both the program and data represented in binary. The index into the array is called the memory *address*. At each address is stored a *word* of data that is typically 8, 16, 32, or 64 bits in length. The processor is the engine that actually runs the program by running through a repeating cycle of steps:

- read an instruction from memory
- decode the instruction and execute it, reading and writing data values as needed
- determine the address of the next instruction and repeat

Given this simple, high-level model for a computer, we can start to gain insights into some of the underlying implementation details by doing some forensic detective work on a computer program. In particular, we will consider the following questions:

- How are numbers and letters represented with binary data?
- What are some common arithmetic and Boolean logic operations on data?
- Where in memory (at what addresses) are data values stored?
- Where and how is the program stored in memory?

## 2.2 Binary Representation of Data in C

In a programming language, a *data type* is a way of describing the representation and properties of a variable, including how many bits, how to interpret them, and what operations can be performed on them. In C, an `int` is a 32-bit signed integer with a 2's complement representation. We can verify this by printing an `int` in both hex and decimal format using the `printf` function. Consider the following 32-bit 2's complement integer expressed in binary and hexadecimal:

binary:	0000 0000 0000 0000 0000 0000 1010 1011
hex:	0 0 0 0 0 0 a b

Since this number has a 0 in the most-significant bit, it is a positive number. The decimal value of this number is

$$1 \times 2^7 + 1 \times 2^5 + 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = 171$$

We can display this 32-bit number in both hex and decimal formats by using `printf` with format specifier `%x` for hex and `%d` for decimal:

```
#include <stdio.h>
int main ()
{
    int v = 0xab;
    printf("%x\n", v);
    printf("%d\n", v);
}
```

This prints

ab
171

Now consider the following 32-bit 2's complement integer expressed in binary and hexadecimal:

binary:	1111 1111 1111 1111 1111 1111 1010 1011
hex:	f f f f f f a b

Since this number has a 1 in the most-significant bit, it is negative. Thus to express it in signed decimal format, we need to find its additive inverse (which will be positive) and attach a negative sign. To find the additive inverse of a 2's complement number, we invert the bits and add 1:

```

original:      1111 1111 1111 1111 1111 1111 1010 1011
inverted:      0000 0000 0000 0000 0000 0000 0101 0100
inverted + 1: 0000 0000 0000 0000 0000 0000 0101 0101

```

The decimal value of the additive inverse of the original number is

$$1 \times 2^6 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 = 85$$

Thus the signed decimal representation of `0xffffffffab` is -85. The following C program demonstrates this:

```
#include <stdio.h>
int main ()
{
    int v = 0xffffffffab;
    printf("%x\n", v);
    printf("%d\n", v);
}
```

This prints

```
fffffab
-85
```

C provides the `unsigned` data type to be able to use 32 bits to represent positive integers. The `printf` format specifier `%u` displays 32-bit values as unsigned integers.

```
#include <stdio.h>
int main ()
{
    unsigned v = 0xffffffffab;
    printf("%x\n", v);
    printf("%u\n", v);
}
```

This prints

```
fffffab
4294967211
```

Note that regardless of whether the hex value `0xffffffffab` is assigned to an `int` or an `unsigned`, the 32-bit pattern stored in memory is the same. In fact, by using the appropriate format specifier, we can print either an `int` or an `unsigned` in either format. The difference is in how the program interprets it, as illustrated by the following program:

```
#include <stdio.h>
int main ()
{
    int      v_int = 0xffffffffab;
    unsigned v_uns = 0xffffffffab;

    printf("v_int: %x (hex)  %d (signed)  %u (unsigned)\n", v_int, v_int, v_int);
    printf("v_uns: %x (hex)  %d (signed)  %u (unsigned)\n", v_uns, v_uns, v_uns);
```

```

if (v_int < 0)
    printf("v_int is negative\n");
else
    printf("v_int is positive\n");

if (v_uds < 0)
    printf("v_uds is negative\n");
else
    printf("v_uds is positive\n");
}

```

This prints

```

v_int: ffffffab (hex) -85 (signed) 4294967211 (unsigned)
v_uds: ffffffab (hex) -85 (signed) 4294967211 (unsigned)
v_int is negative
v_uds is positive

```

The `int` and `unsigned` C data types are both 32 bits long. We can confirm this with the C `sizeof` operator, which determines the size of a variable or data type in *bytes*, where a byte is 8 bits. In addition to `int` and `unsigned`, C has the data types `char`, `short`, `long`. We can use `sizeof` to determine the length of each of these in bytes; note that these lengths may be different for different C compilers.

```

#include <stdio.h>
int main ()
{
    char v_char;
    short v_short;
    int v_int;
    long v_long;

    printf("size of v_char: %d bytes\n", sizeof(v_char));
    printf("size of v_short: %d bytes\n", sizeof(v_short));
    printf("size of v_int: %d bytes\n", sizeof(v_int));
    printf("size of v_long: %d bytes\n", sizeof(v_long));
}

```

This prints:

```

size of v_char: 1 bytes
size of v_short: 2 bytes
size of v_int: 4 bytes
size of v_long: 8 bytes

```

The C data type `char`, which is 1 byte (8 bits) in length is so named because that is the length of a text character represented in ASCII code (American Standard Code for Information Interchange, see [asciicode.com](http://www.asciicode.com)). We can display a `char` in ASCII format using the `%c` format specifier. Note, however, that an ASCII character is just one interpretation of an 8-bit `char`; it can still be interpreted as a numerical value as illustrated below:

```

#include <stdio.h>
int main ()
{

```

```

char v_char = 0x4a;

printf("v_char: %x (hex) %d (dec) %c (ascii)\n",
       v_char, v_char, v_char);
}

```

This prints:

```
v_char: 4a (hex) 74 (dec) J (ascii)
```

## 2.3 Bit-Level Boolean Operations in C

C provides the ability to perform bit-level Boolean operations on data. These operators include:

- $\sim$  bitwise logical inverse (NOT)
- & bitwise logical AND
- | bitwise logical OR
- bitwise logical XOR

As a first illustration of bitwise Boolean logic operations, consider the difference between arithmetic negation with the  $-$  operator and logical inversion with the  $\sim$  operator. Recall that for 2's complement integers, the arithmetic negation or additive inverse of a number is equal to the bitwise logical inversion plus 1. Thus, consider the 32-bit representation for 1,  $\sim 1$ , and  $-1$ :

```

1: 0000 0000 0000 0000 0000 0000 0000 0001 (0x1)
~1: 1111 1111 1111 1111 1111 1111 1111 1110 (0xfffffffffe)
-1: 1111 1111 1111 1111 1111 1111 1111 1111 (0xffffffff)

```

The following C program illustrates this:

```

#include <stdio.h>
int main ()
{
    int v = 1;
    printf("~1: %x (hex) %d (dec)\n", ~v, ~v);
    printf("-1: %x (hex) %d (dec)\n", -v, -v);
}

```

This prints:

```

~1: ffffffe (hex) -2 (dec)
-1: ffffffff (hex) -1 (dec)

```

The bitwise AND and OR operations are particularly useful for setting bits within a binary number to 1s (by OR-ing with a 1) or 0s (by AND-ing with a 0). For example:

```

v:      1010 1010 1010 1010 1010 1010 1010 1010 (0aaaaaaaa)
mask:   1111 0000 1111 0000 1111 0000 1111 0000 (0xf0f0f0f0)
v & mask: 1010 0000 1010 0000 1010 0000 1010 0000 (0xa0a0a0a0)
v | mask: 1111 1010 1111 1010 1111 1010 1111 1010 (0x1a1a1a1a)

```

The following C program illustrates this:

```

#include <stdio.h>
int main ()
{
    int v      = 0aaaaaaaaa;
    int mask = 0xf0f0f0f0;
    printf("v & mask: %x\n", v & mask);
    printf("v | mask: %x\n", v | mask);
}

```

This prints:

```
v & mask: a0a0a0a0
v | mask: fafafafa
```

Another bit-level operation is shifting the bits in a variable to the left or right. The C shift operators are as follows:

---

a << b shifts the bits of a to the left by b bits, shifting in zeros from the right

a >> b shifts the bits of a to the right by b bits, shifting in copies of the most significant bit from the left (which preserves the sign of an `int`)

Note that shifting a number by  $b$  bits to the left multiplies it by  $2^b$  and shifting it by  $b$  bits to the right divides it by  $2^b$ , as illustrated by the 8-bit example below:

```

a:      0000 1100  0x0c (hex)  12 (dec)
a << 2: 0011 0000  0x30 (hex)  48 (dec)
a >> 2: 0000 0011  0x03 (hex)  3 (dec)

```

In C:

```

#include <stdio.h>
int main ()
{
    char a = 12;
    printf("a << 2: %x (hex) %d (dec)\n", a << 2, a << 2);
    printf("a >> 2: %x (hex) %d (dec)\n", a >> 2, a >> 2);
}

```

This prints:

```
a << 2: 30 (hex) 48 (dec)
a >> 2: 3 (hex) 3 (dec)
```

## 2.4 Memory Organization in C

### 2.4.1 Memory Addresses and Pointers

As stated earlier, a computer memory is like a large array, where the memory *address* serves as the index into the array, and where each array location holds a data word of some fixed width. For most modern computer systems, the size of a data word at each address is 1 byte (8 bits); this is known as *byte-addressable* memory. For a 64-bit Intel processor, a memory address is a 64-bit number ranging from 0 to  $2^{64} - 1$ , where each memory location contains 1 byte of data. Thus the maximum memory address in

hexadecimal is `ffffffffffffffff` or in decimal 18,446,744,073,709,551,615 ( $1.8 \times 10^{19}$ ). This is a huge number and in practice, no computer actually has anywhere near that much physical memory, but because of a concept called *virtual memory*, program data and binary-encoded instructions can be spread through that entire range, and then mapped to a smaller amount of physical memory. Since each memory location only holds 1 byte of data, an 8-bit C `char` can fit into one location, but longer data types including `int`, `short`, and `long` require several memory locations spanning multiple consecutive memory addresses—we'll take a closer look at this soon.

In C, a *pointer* is just another name for a memory address. We can determine the address of a variable using the address operator `&`. The pointer dereference operator `*` determines the contents of a memory location given an address. Thus given a variable `v`, the address of `v` is determined by `&v`, and the content of `v` is determined by `*(%v)`. The following C program example uses these operations to display the address and content of a `char`, where the format specifier `%p` for “pointer” is used to display a 64-bit address in hexadecimal format.

```
#include <stdio.h>
int main ()
{
    char v = 7;
    printf("&v:      %p\n", &v);
    printf("*(&v): %d\n", *(&v));
}
```

This prints

```
&v:      0x7ffd5764dd6f
*(&v): 7
```

The system assigns the address `0x7ffd5764dd6f` as the location of `v` when the program runs. Note that this address can be different on different systems, and even on different runs on the same system.

## 2.4.2 Arrays

In C, an array is a consecutive set of memory locations. When we declare an array in C, for example, a `char` array `A` of length 3.

```
char A[3] = {30, 31, 32};
```

The system reserves 3 consecutive memory locations starting at address `A`. The type of variable `A` is an address or pointer to a `char`. Note that using name of an array variable in an expression in a C program is very different from using the name of a scalar variable. Whereas the name of a scalar variable evaluates to the *value* of the variable, the name of an array variable evaluates to the *address* of the variable. Thus, if `A` is the starting address of an array, `A+1` and `A+2` are the addresses of the next 2 consecutive elements.

To get to the value of an array variable, we need to dereference that address. The C language provides two equivalent mechanisms for accessing elements of an array: the indexing operator `[ ]` or the pointer dereference operator `*`. While the indexing operator `[ ]` is the more familiar notation, the pointer dereference operator `*` provides more insight into the underlying hardware. The following table illustrates the use of either notation for the addresses and data associated with array `A`:

	Address	Data
1st element of A	<code>A</code> or <code>&amp;A[0]</code>	<code>*A</code> or <code>A[0]</code>
2nd element of A	<code>A+1</code> or <code>&amp;A[1]</code>	<code>*(A+1)</code> or <code>A[1]</code>

3rd element of A A+2 or &A[2] \*(A+2) or A[2]

The following C program illustrates the two alternative approaches to accessing an array and prints the actual addresses where the array elements are stored.

```
#include <stdio.h>
int main ()
{
    char A[3] = {0x1a, 0x2a, 0x3a};
    printf("array index notation\n");
    printf("A[0]:  addr: %p  data: %d\n", &A[0], A[0]);
    printf("A[1]:  addr: %p  data: %d\n", &A[1], A[1]);
    printf("A[2]:  addr: %p  data: %d\n", &A[2], A[2]);

    printf("pointer notation\n");
    printf("A:      addr: %p  data: %d\n", A, *(A));
    printf("A+1:    addr: %p  data: %d\n", A+1, *(A+1));
    printf("A+2:    addr: %p  data: %d\n", A+2, *(A+2));
}
```

This prints:

```
array index notation
A[0]:  addr: 0xffff87e71f8d  data: 1a
A[1]:  addr: 0xffff87e71f8e  data: 2a
A[2]:  addr: 0xffff87e71f8f  data: 3a
pointer notation
A:      addr: 0xffff87e71f8d  data: 1a
A+1:    addr: 0xffff87e71f8e  data: 2a
A+2:    addr: 0xffff87e71f8f  data: 3a
```

The table below shows the location of array A in memory in a diagram called an *address map*. The *address space* of the map is the full range of 64-bit addresses from 0 to  $2^{64} - 1$ , shown in the diagram with the lowest address on the bottom, with the 64-bit addresses expressed as 16 digit hex values. As we can see, the 3 elements of A are stored in 3 consecutive memory locations, with addresses ending in the hex digits d, e, and f. The name of the array variable, “A,” serves as a label that refers to the starting address of the array.

Address Expression	Address	Data
	0xffffffffffffffffff	
	...	
A+2 or &A[2]	0x00007fff87e71f8f	0x3a
A+1 or &A[1]	0x00007fff87e71f8e	0x2a
A or &A[0]	0x00007fff87e71f8d	0x1a
	...	
	0x0000000000000000	

In the previous example, we looked at how an array of 8-bit `char` data is stored in memory. Since the memory is byte-addressable and a `char` is exactly 1 byte, each element of the array fits exactly into 1 memory location. But what happens with an array of `ints`, where each element is 32 bits wide and takes 4 bytes to store? To find out, we'll examine the results of the following C program where `B` is an array of `ints`:

```
#include <stdio.h>
int main ()
{
    int B[3] = {0x1a1b1c1d, 0x2a2b2c2d, 0x3a3b3c3d};
    printf("array index notation\n");
    printf("B[0]:  addr: %p  data: %x\n", &B[0], B[0]);
    printf("B[1]:  addr: %p  data: %x\n", &B[1], B[1]);
    printf("B[2]:  addr: %p  data: %x\n", &B[2], B[2]);

    printf("pointer notation\n");
    printf("B:      addr: %p  data: %x\n", B, *(B));
    printf("B+1:    addr: %p  data: %x\n", B+1, *(B+1));
    printf("B+2:    addr: %p  data: %x\n", B+2, *(B+2));
}
```

This prints:

```
array index notation
B[0]:  addr: 0xffff3596e024  data: 1a1b1c1d
B[1]:  addr: 0xffff3596e028  data: 2a2b2c2d
B[2]:  addr: 0xffff3596e02c  data: 3a3b3c3d
pointer notation
B:      addr: 0xffff3596e024  data: 1a1b1c1d
B+1:    addr: 0xffff3596e028  data: 2a2b2c2d
B+2:    addr: 0xffff3596e02c  data: 3a3b3c3d
```

Here, we see that successive elements of the array are located at every *fourth* address, meaning that it takes 4 consecutive addresses to store 1 `int`. In an Intel processor, an `int` is stored in memory with its least-significant byte at the lowest address, so the memory map with array `B` is as follows:

Address Expression	Address	Data
	0xffffffffffffffffff	
	...	
	0x00007fff3596e02f	0x3a
	0x00007fff3596e02e	0x3b
	0x00007fff3596e02d	0x3c
B+2 or &B[2]	0x00007fff3596e02c	0x3d
	0x00007fff3596e02b	0x2a
	0x00007fff3596e02a	0x2b
	0x00007fff3596e029	0x2c
B+1 or &B[2]	0x00007fff3596e028	0x2d
	0x00007fff3596e027	0x1a

	0x00007fff3596e026	0x1b
	0x00007fff3596e025	0x1c
B or &B[0]	0x00007fff3596e024	0x1d
	...	
	0x0000000000000000	

From the memory map, we see that the 3 elements of `int` array `B` are mapped to 12 consecutive byte addresses ranging `0x00007fff3596e024–2f`. One subtle point from this example is that the expression `B+1` doesn't simply add 1 to the address of `B`—instead it adds 4. This is how “pointer arithmetic” works in C; it adds the size of the data type in bytes to the address, not just 1 address. Thus `B+1` points to the second element of array `B` and `B+2` points to the third element.

### 2.4.3 Byte Order and Type Casting

Another subtle point about the memory map: how can we be sure that `ints` are stored in 4 consecutive memory locations with the least-significant byte at the lowest address? This arrangement of bytes is known as “little endian;” in fact, some systems store `ints` and longer data types with the most-significant byte at lowest address, an arrangement known as “big endian.” To confirm, we'll trick the computer into thinking that an `int` variable is actually an array of 4 `chars` and then print the addresses of the array elements and their contents.

In C, we can define a variable as a pointer type and assign to point to the address of another variable. In the following example, we define a variable `v_ptr` of type `int` pointer (`int *`) and assign to it the address of variable `v` that is of type `int`. When we dereference pointer `v_ptr`, we expect to get the value of `v`.

```
#include <stdio.h>
int main ()
{
    int    v = 0x1a2b3c4d;
    int *v_ptr = &v;

    printf("&v: %p    v:      %x\n", &v, v);
    printf("v_ptr: %p    *v_ptr: %x\n", &v_ptr, *v_ptr);
}
```

This prints:

```
&v: 0x7ffd905c0684    v: 1a2b3c4d
v_ptr: 0x7ffd905c0684    *v_ptr: 1a2b3c4d
```

which shows that `v_ptr` contains the address of `v`.

Using a technique called *type casting*, we can assign a pointer to one type of variable to a pointer to another type of variable. Using type casting, we can define a variable of type `int` and then create a pointer to it as if it were a `char`. The following C program demonstrates this.

```
#include <stdio.h>
int main ()
{
    int    v          = 0x1a2b3c4d;
```

```

char *v_char_ptr = (char *) &v;

printf("&v: %p v: %x\n", &v, v);
printf("v_char_ptr: %p *v_char_ptr: %x\n", v_char_ptr, *v_char_ptr);

```

This prints:

```

&v: 0x7ffe4cffc6c4 v: 1a2b3c4d
v_char_ptr: 0x7ffe4cffc6c4 *v_char_ptr: 4d

```

The results of this demonstrates that `v_char_ptr` indeed points to the location of `v`. When we dereference the pointer `v_char_ptr`, we get the least significant byte of `v`, `0x4d`. The following C program extends this example to view all 4 bytes of `v` and the addresses where they are stored as if it were an array of `char`s.

```

#include <stdio.h>
int main ()
{
    int v = 0x1a2b3c4d;
    int *v_int_ptr = &v;
    char *v_char_ptr = (char *) &v;

    printf("v as int: addr: %p data: %x\n", v_int_ptr, *v_int_ptr);
    printf("v as array of char:\n");
    printf("addr: %p data: %x\n", v_char_ptr, *v_char_ptr);
    printf("addr: %p data: %x\n", v_char_ptr+1, *(v_char_ptr+1));
    printf("addr: %p data: %x\n", v_char_ptr+2, *(v_char_ptr+2));
    printf("addr: %p data: %x\n", v_char_ptr+3, *(v_char_ptr+3));
}

```

This prints:

```

v as int: addr: 0x7fffa47c5c0c data: 1a2b3c4d
v as array of char:
    addr: 0x7fffa47c5c0c data: 4d
    addr: 0x7fffa47c5c0d data: 3c
    addr: 0x7fffa47c5c0e data: 2b
    addr: 0x7fffa47c5c0f data: 1a

```

As expected, the 4 bytes of `int` variable `v` are stored at 4 consecutive addresses, with the least significant byte at the lowest address.

Thus far, we've observed the following regarding the placement of data from a C program in memory:

- The system memory has  $2^{64}$  16-bit address
- Each memory address points to 1 byte of data
- Data types such as `int` that are more than 1 byte long are stored in consecutive addresses, with the least-significant byte of data stored at the lowest address.
- The elements of an array are stored in consecutive memory locations.

## 2.4.4 Memory Segments: Text, Data, Stack, and Heap

Next, we'll consider the question of where the system places data within the entire 64-bit address space. In our examples so far, all of the data has been placed in the neighborhood of address `0x7fff00000000`, but is there a further pattern to this. To gain some insight, we'll write a C program to print the addresses to data defined in several different ways:

- local variables inside of functions (including `main`)
- global variables
- data allocated dynamically using `malloc`
- the starting addresses of functions

The final element in this list—the starting addresses of functions—is curious. As mentioned earlier, the instructions of a computer program are themselves encoded as binary data and stored in memory. The instructions for each function are stored in a contiguous block of consecutive memory locations. The name a function is actually a pointer, of type *function pointer* that holds the starting address of the function in memory. We can print the value of a function pointer in the same way that we can print the value of any other pointer.

The following C program prints the addresses of these different kinds of data. The program has 3 functions, `main`, `fcn1`, and `fcn2`, where `main` calls `fcn1`, which in turn calls `fcn2`. Each of the 3 functions has a local variable, named `v_main`, `v_fcn1`, and `v_fcn2`. The program has a global variable `v_global`. `main` also uses `malloc` to dynamically allocate data, the address of which is stored in the local pointer variable `p_malloc`.

```
#include <stdio.h>
#include <stdlib.h>
int v_global;

void fcn2 ()
{
    int v_fcn2;
    printf("&v_fcn2:      %016p\n", &v_fcn2);
}

void fcn1 ()
{
    int v_fcn1;
    printf("&v_fcn1:      %016p\n", &v_fcn1);
    fcn2();
}

int main ()
{
    int v_main;
    int *p_malloc = (int *) malloc(sizeof(int));
    fcn1();
    printf("&v_main:      %016p\n", &v_main);
    printf("&v_global:     %016p\n", &v_global);
    printf("p_malloc:      %016p\n", p_malloc);
    printf("main:         %016p\n", main);
```

```

    printf("fcn1:      %016p\n", fcn1);
    printf("fcn2:      %016p\n", fcn2);
}

```

The program uses the format specifier `%016p` to print the pointer addresses as 16 digit hex values padded with leading 0s, which makes it easier to interpret the results. The printed values are:

```

&v_fcn1:      0x007ffc4a2c115c
&v_fcn2:      0x007ffc4a2c113c
&v_main:      0x007ffc4a2c1174
&v_global:    0x0000000040403c
p_malloc:     0x00000000bd0260
main:         0x0000000040117e
fcn1:         0x00000000401153
fcn2:         0x00000000401132

```

The table below sorts and groups these addresses in the memory map:

Address Expression	Address	Segment
	0xffffffffffffffffffff	
	...	
&v_main	0x00007ffc4a2c1174	stack
&v_fcn1	0x00007ffc4a2c115c	
&v_fcn2	0x00007ffc4a2c113c	
	...	
p_malloc	0x0000000000bd0260	heap
	...	
&v_global	0x000000000040403c	data
	...	
main	0x000000000040117e	text
fcn1	0x0000000000401153	
fcn2	0x0000000000401132	
	...	
	0x0000000000000000	

The table reveals the typical memory organization for a C program in the Linux operating system, where different types of data are grouped together in regions of memory called *segments*. From bottom to top, these segments are as follows:

- **Text Segment:** At the bottom of the memory are the pointers to the 3 function, `main`, `fcn1`, and `fcn2`. If we were to look at the data values at these addresses—which we will do later—we would find the binary-encoded text of program instructions.
- **Data Segment:** The compiler places global variables in the data segment just above the functions in the memory map.

- **Heap:** The heap is the data segment reserved for dynamically allocated data using `malloc` or related utilities. It is located just above the data segment. It grows upward, where successive calls to `malloc` allocate data at increasingly higher addresses until space is freed up using `free`.
- **Stack:** Local variables inside functions are allocated in a data segment called the stack. Whenever a function is called, a new block of memory is allocated on the stack called at *stack frame*. The base of the stack is located towards the top of the memory map and grows downward. The stack frame for `main` is above `fcn1`, which is above the stack from for `fcn2`, since this is the order in which the functions are called. The base of the stack is far above the heap in the address space, so that the two won't collide as the stack grows downward and the heap grows upward.

## 2.5 Assembly Language and Machine Code

As discussed at the end of the last section, the text segment of a computer program in memory contains a binary-encoded version of the program instructions. These instructions are *not* the same instructions found in the C program. Rather, every microprocessor has a set of basic instructions called its *instruction set* that specifies the primitive operations that the microprocessor can perform, and the program stored in memory is a translation of the C—or other high-level language—program into the native instruction set of the microprocessor. Different microprocessors such as Intel or ARM processors have different instruction sets, and the design of the instruction set has important implications on the processor's performance and efficiency. In this section, we will take a first look at the steps involved in translating a C program to a processor's native instruction set, and how the translated program is stored in memory. The next section takes an in-depth look at the instruction set of a particular microprocessor and considers it's implications on the hardware

Each instruction in a processor's instruction set has both a text and binary representation. The text representation is called *assembly language* and the binary representation is called *machine code*. When you compile a C program, it goes through a multistep process of first translating the C program to assembly language, then translating the assembly language to machine code, and then linking the machine code of your program with the machine code of libraries of other functions needed to produce a complete binary program that can be loaded into memory and executed.

### 2.5.1 Original C Program

We will use a simple example of a program `addsub.c` that calls a function to perform some simple arithmetic operations to illustrate each of these steps.

```
// addsub.c
#include <stdio.h>

int addsub(int a, int b, int c)
{
    return a + b - c;
}

int main()
{
    int y = addsub(3, 5, 1);
    printf("y: %d\n", y);
}
```

Ordinarily, we would compile the program to an executable `a.out` and run it:

```
$ gcc addsub.c  
$ ./a.out
```

This produces the output

```
y: 7
```

## 2.5.2 Assembly Language Program

Rather than compiling the program all the way to the executable, we could just compile it to an assembly language program using the `-S` compiler option

```
$ gcc -S addsub.c
```

This produces the assembly language program `addsub.s`. When compiled for an Intel processor running Linux, the Intel x86 assembly language program is as shown below. At this point, it is not necessary to *understand* the assembly language program, but rather to make a few observations. First, it is much longer than the C program, where a single C requires multiple assembly language instructions to implement. Second, it is much less abstract than the C program; where the C program uses variables, the assembly language program uses memory addresses and a small number of hardware storage locations called registers inside the processor.

```
$ cat addsub.s  
    .file    "addsub.c"  
    .text  
    .globl   addsub  
    .type    addsub, @function  
  
addsub:  
    .LFB0:  
        .cfi_startproc  
        pushq   %rbp  
        .cfi_def_cfa_offset 16  
        .cfi_offset 6, -16  
        movq   %rsp, %rbp  
        .cfi_def_cfa_register 6  
        movl   %edi, -4(%rbp)  
        movl   %esi, -8(%rbp)  
        movl   %edx, -12(%rbp)  
        movl   -8(%rbp), %eax  
        movl   -4(%rbp), %edx  
        addl   %edx, %eax  
        subl   -12(%rbp), %eax  
        popq   %rbp  
        .cfi_def_cfa 7, 8  
        ret  
        .cfi_endproc  
  
.LFE0:
```

Statements starting with a ‘.’ are not instructions but directives that define symbols or otherwise give hints on how to assemble the program.

Names followed by ‘:’ are labels for memory addresses where either instructions or data are stored. This is the starting address of the `addsub` function.

Names starting with ‘%’ refer to limited, temporary storage locations inside the processor hardware called *registers*.

Register names inside of ‘( )’ refer to memory addresses.

`mov` instructions copy data between registers or between registers and memory.

Here’s where the add and subtract happens, as 2 separate instructions.

```

.size    addsub, .-addsub
.section      .rodata
.LC0:
.string "y: %d\n"
.text
.globl main
.type  main, @function
main:
.LFB1:
.cfi_startproc
pushq  %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq   %rsp, %rbp
.cfi_def_cfa_register 6
subq   $16, %rsp
movl   $1, %edx
movl   $5, %esi
movl   $3, %edi
call   addsub
movl   %eax, -4(%rbp)
movl   -4(%rbp), %eax
movl   %eax, %esi
movl   $.LC0, %edi
movl   $0, %eax
call   printf
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE1:
.size  main, .-main
.ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)"
.section .note.GNU-stack,"",@progbits

```

.LC0 is a label for the memory location where the `printf` string is stored.

Here's the starting address of `main`.

Here's where `main` calls `addsub`.

The `printf` string at memory location with label `.LC0` is copied to a register in preparation for the call to `printf`.

In the above example, we compiled `addsub.c` for an Intel x86 processor. If we compiled it for a different processor with a different instruction set, we would get a different assembly language program. For example, here is the assembly code for the `addsub` function for an ARM processor, which is found in many mobile devices. Although there are some functional similarities, the instructions and syntax are different.

```

addsub:
sub    sp, sp, #16
str    w0, [sp, 12]
str    w1, [sp, 8]
str    w2, [sp, 4]
ldr    w1, [sp, 12]
ldr    w0, [sp, 8]
add    w1, w1, w0
ldr    w0, [sp, 4]

```

`sp` is a “stack pointer” register with the address of the stack in memory. `w0`, `w1`, and `w2` are “general purpose” registers.

`str` and `ldr` store and load values between registers and memory.

Here's the addition, after the function arguments have been loaded into registers, followed by the subtraction.

```

sub      w0, w1, w0
add      sp, sp, 16
ret

```

### 2.5.3 Machine Code for Executable Program

Given the Intel x86 assembly language version of the program, the next step is to translate it to the binary machine code. We do this by compiling the assembly language program to produce the executable file `a.out`.

```
$ gcc addsub.s
```

The executable file `a.out` is a binary file—a bunch of 1's and 0's—that contains some header information about how the file is structured, followed by machine code. The file isn't human-readable, but we can use a tool called a disassembler to translate the executable file back to assembly language, showing the bytes of machine code associated with each assembly language instructions, and where they are stored in memory.

```
$ objdump -d a.out
```

The first thing you will notice is that the disassembled `a.out` contains a lot more code than the assembly language program `addsub.s`. This is because to make the program executable, the compiler need to add additional code for initialization and calls to library functions such as `printf`. Scrolling down through the output, you reach the sections with the code for the functions `addsub` and `main`. The output is arranged in three columns. The first column is the memory address, the second column is the bytes of machine code (1 byte is 2 hex digits) stored in memory beginning at that address, and the third column is the assembly language instruction corresponding to the disassembled machine code.

Thus, from the output below:

- The `addsub` function starts at memory address (in hex) `40052d`. The byte `55` is stored at that address, which is the machine code for the instruction `push %rbp`.
- The bytes `48, 89, e5` are stored at the next 3 address `40052e`, `40052f`, and `400530`, which is the machine code for the instruction `mov %rsp,%rbp`.
- The machine code for the next instruction, `mov %edi,-0x4(%rbp)`, starts at address `400531`, consisting of the 3 bytes `89 7d fc`.
- Note that the machine code for different instructions has different numbers of bytes, ranging from 1 byte for a `push` instruction to 5 bytes for a `callq` instruction.

<i>Memory Addresses</i>	<i>Machine Code (bytes in hex)</i>	<i>Disassembled Assembly Language Instructions</i>
<hr/>		
<code>000000000040052d &lt;addsub&gt;:</code>		
<code>40052d:</code>	<code>55</code>	<code>push %rbp</code>
<code>40052e:</code>	<code>48 89 e5</code>	<code>mov %rsp,%rbp</code>
<code>400531:</code>	<code>89 7d fc</code>	<code>mov %edi,-0x4(%rbp)</code>
<code>400534:</code>	<code>89 75 f8</code>	<code>mov %esi,-0x8(%rbp)</code>
<code>400537:</code>	<code>89 55 f4</code>	<code>mov %edx,-0xc(%rbp)</code>
<code>40053a:</code>	<code>8b 45 f8</code>	<code>mov -0x8(%rbp),%eax</code>
<code>40053d:</code>	<code>8b 55 fc</code>	<code>mov -0x4(%rbp),%edx</code>
<code>400540:</code>	<code>01 d0</code>	<code>add %edx,%eax</code>
<code>400542:</code>	<code>2b 45 f4</code>	<code>sub -0xc(%rbp),%eax</code>
<code>400545:</code>	<code>5d</code>	<code>pop %rbp</code>

```

400546:    c3                      retq
0000000000400547 <main>:
400547:    55                      push   %rbp
400548:    48 89 e5                mov    %rsp,%rbp
40054b:    48 83 ec 10             sub    $0x10,%rsp
40054f:    ba 01 00 00 00           mov    $0x1,%edx
400554:    be 05 00 00 00           mov    $0x5,%esi
400559:    bf 03 00 00 00           mov    $0x3,%edi
40055e:    e8 ca ff ff ff         callq  40052d <addsub>
400563:    89 45 fc                mov    %eax,-0x4(%rbp)
400566:    8b 45 fc                mov    -0x4(%rbp),%eax
400569:    89 c6                  mov    %eax,%esi
40056b:    bf 10 06 40 00           mov    $0x400610,%edi
400570:    b8 00 00 00 00           mov    $0x0,%eax
400575:    e8 96 fe ff ff         callq  400410 <printf@plt>
40057a:    c9                      leaveq 
40057b:    c3                      retq
40057c:    0f 1f 40 00             nopl   0x0(%rax)

```

## 2.5.4 A Program that Prints Its Own Machine Code

Earlier in Section 1.2.3, we noted that in C, the name of a function is a pointer to the starting address of that function in memory. We now know that the machine code for the function is stored in memory starting at that address. So, if we dereference the function pointer and print the bytes stored at that address, we should see the machine code for the function.

The following C function takes a function pointer as an input and prints the values of 16 sequential bytes in memory, starting from that address. We will use the data type `void *` to indicate a generic function pointer. In order to be able to dereference this pointer and print the data stored there in hex, we need to trick the computer into thinking that this is a pointer to a numeric value. In C, the data type for a byte—that is, an 8-bit unsigned value—is `unsigned char`. So, we will cast the function pointer of type `void *` to a pointer to a byte of type `unsigned char *`.

```

void print_machine_code(void *fcn)
{
    // print the function pointer address
    printf("%p: ", fcn);

    // Cast the function pointer to unsigned char pointer to
    // trick the computer into letting us dereference it
    // to get the byte stored there
    unsigned char *fcn_as_charptr = (unsigned char *) fcn;

    // Print the 1st 16 bytes of machine code of the function
    int i;
    for (i = 0; i < 16; i++) {
        printf("%02x ", *(fcn_as_charptr + i));
    }
    printf("\n");
}

```

To test this, we'll add the function `print_machine_code` to `addsub.c` and print the machine code for the functions `addsub` and `main`.

```
int main()
{
    int y = addsub(3, 5, 1);
    printf("y: %d\n", y);
    printf("machine code for addsub\n");
    print_machine_code(addsub);
    printf("machine code for main\n");
    print_machine_code(main);
}
```

Compiling and running the program prints the following:

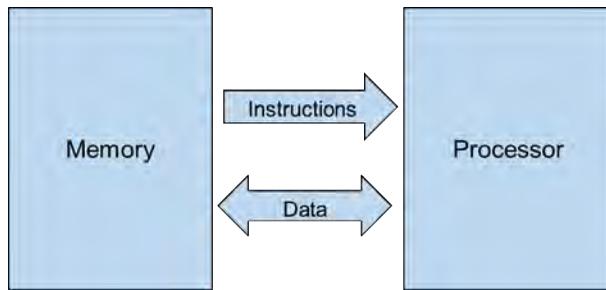
```
y: 7
machine code for addsub
0x4005bd: 55 48 89 e5 89 7d fc 89 75 f8 89 55 f4 8b 45 f8
machine code for main
0x400644: 55 48 89 e5 48 83 ec 10 ba 01 00 00 00 be 05 00
```

This is the same machine code for functions `addsub` and `main` that we saw when we disassembled the executable `a.out`. The starting addresses of the two functions are slightly different because the C source program was modified to include the new function.

## 3 albaCore Microprocessor: Instructions and Programming

### 3.1 Basics of Computer Architecture and Organization

Though the technology has changed dramatically, the basic ideas of computer architecture and organization have been in place since the 1930s and 1940s. Fundamentally, a computer system has two main hardware components: *memory* that stores instructions and data, and a *processor* that reads instructions from memory and then executes these instructions to process the data.

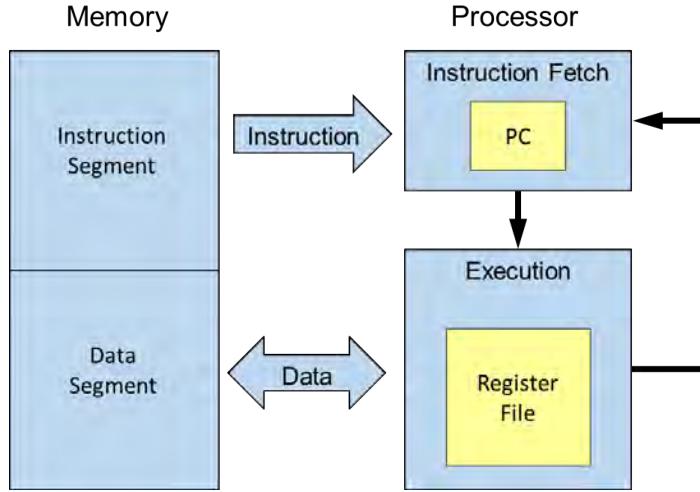


The instructions that a processor executes aren't the statements found in a high-level language such as C, Java, or Python. Instead, they are low-level instructions called machine code that perform fairly primitive operations. This basic set of instructions, called the instruction set architecture (ISA), varies from processor to processor, with some processors having a very large and complex set of instructions and others having a simpler set. A software tool called a *compiler* translates programs from a high-level language down to machine code. (A kind of compiler that does the translation on the fly, as happens with Python, is called an *interpreter*). Machine code is a binary code—a bunch of 1s and 0s—that is decoded by the processor. A text representation of these low-level instructions is called *assembly language*, and a tool that translates assembly language to machine code is called an *assembler*.

In addition to translating high-level languages to machine code, another important role of a compiler is figuring out where to place instructions and data in memory. For security purposes, instructions and data are typically separated into different memory *segments*. Computer systems would be extremely vulnerable to attack if a user program could easily change machine code in memory.

At a very high level, we can think of the processor hardware organization as having two main stages: an *instruction fetch* stage and an *execution* stage.

- *Instruction fetch stage*: As the name implies, the instruction fetch stage is responsible for fetching instructions from memory. It uses a register called a **program counter (PC)** to keep track of which memory address to fetch the next instruction from. This could be the next sequential address, or elsewhere in memory such as when processing conditional statements or loops.
- *Execution stage*: This stage executes the instructions that process the data in memory. Because transferring data between the processor and memory can be slow, the execution stage typically contains a **register file** that is much smaller and faster than memory (typically 16-32 data words) to hold frequently used variables and intermediate results.



## 3.2 The albaCore Instruction Set

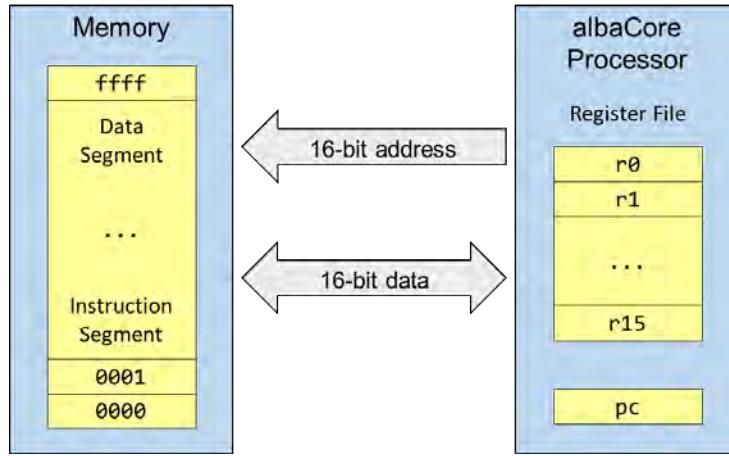
### 3.2.1 Overview: Memory, Registers, and Instructions

As discussed earlier, instruction sets vary from processor to processor, but there are certain common approaches. One particularly common style of instruction set architecture is called a *Reduced Instruction Set Computer (RISC)*. The original intent behind RISC processors was to harmonize the design of the processor and the compiler, with the goal of simplifying both while achieving good performance. The most common example of a RISC processor is the ARM (Advanced RISC Machine), which is found in all kinds of products from smartphones to the Raspberry PI. Here we introduce an example RISC processor instruction set: Intel has its “Core” Architecture, we have “albaCore”. albaCore is a very simple, yet functionally complete 16-bit RISC microprocessor, based on the MIPS architecture. A main design goal for albaCore was for it to be powerful enough to be able to run programs compiled from a significant subset of the C language, including integer arithmetic and logic operations and function calls, but simple enough so that the complete instruction set specification can fit on one page.

The albaCore memory system has 16-bit addresses and 16-bit data. This means that the memory can contain up to  $2^{16}$  or 65,536 16-bit words of data. Data is treated as 16-bit, 2’s complement signed integers. All machine code instructions are also encoded as 16-bit words.

albaCore has 16 general purpose registers, each 16-bits wide. In the albaCore assembly language, we name these r0–r15.

albaCore has a 16-bit program counter register, named pc.



albaCore has a total of only 16 carefully chosen instructions, which are sufficient for implementing many of the features of a program compiled from C. Like most RISC processors, albaCore has five categories of instructions.

- Arithmetic / Logical Instructions
  - Arithmetic (addition, subtraction, usually but not always multiplication/division) and Boolean logic operations.
  - Use registers or constants immediate values as operands
- Data Transfer Instructions (Loads and Stores)
  - Load register from data memory
  - Store register to data memory
- Branch
  - Change the PC to some new address relative to the current PC value. Branches can be conditional on some relation between registers (less than, greater than, zero, negative) or unconditional. Branch instructions are used to support if-else statements and loops in a high-level language such as C.
- Jump
  - Change PC to an absolute new location. Jump instructions are used to support function calls in a high-level language such as C.
- Special
  - Quit execution

The complete albaCore instruction set, which will be explained over the next few sections, is summarized in the table below.

instruction	name	operation	16-bit encoding			
			15:12	11:8	7:4	3:0
add rw, ra, rb	add	$rw \leftarrow ra + rb$	0x0	rw	ra	rb
sub rw, ra, rb	subtract	$rw \leftarrow ra - rb$	0x1	rw	ra	rb
and rw, ra, rb	bitwise and	$rw \leftarrow ra \& rb$	0x2	rw	ra	rb
or rw, ra, rb	bitwise or	$rw \leftarrow ra   rb$	0x3	rw	ra	rb
not rw, ra	bitwise not	$rw \leftarrow \sim ra$	0x4	rw	ra	rb
shl rw, ra, rb	shift left	$rw \leftarrow ra \ll rb$	0x5	rw	ra	rb
shr rw, ra, rb	shift right	$rw \leftarrow ra \gg rb$	0x6	rw	ra	rb
ldi rw, imm8	load immediate	$rw \leftarrow imm8$	0x7	rw	imm8	
ld rw, rb, imm4	load (from mem)	$rw \leftarrow M[rb+imm4]$	0x8	rw	imm4	rb
st ra, rb, imm4	store (to mem)	$M[rb+imm4] \leftarrow ra$	0x9	imm4	ra	rb
br imm8	branch (uncond)	$PC \leftarrow PC + \text{SignEx}(imm8)$	0xA	imm8		X
bz rb, imm8	branch if zero	<pre>if (rb == 0)     PC \leftarrow PC + \text{SignEx}(imm8) else     PC \leftarrow PC + 1</pre>	0xB	imm8		rb
bn rb, imm8	branch if negative	<pre>if (rb &lt; 0)     PC \leftarrow PC + \text{SignEx}(imm8) else     PC \leftarrow PC + 1</pre>	0xC	imm8		rb
jal imm12	jump and link	$PC \leftarrow \{PC[15:12], imm12\}$ $r15 \leftarrow PC + 1$	0xD	imm12		
jr ra	jump register	$PC \leftarrow ra$	0xE	X	ra	X
quit	quit	$PC \leftarrow PC$	0xF	X	X	X

### 3.2.2 albaCore Assembler and Simulator

To facilitate the writing of albaCore programs and to demonstrate how they run on the processor, we have developed an albaCore assembler and simulator. An *assembler* is a tool that translates a program written in *assembly language* to machine code. In general, the assembly language for a microprocessor consists of the textual form of the instruction set, along with additional assembler directives that determine how instructions and data organized in memory segments. The albaCore assembler `albaasmh` translates albaCore assembly language programs into a hexadecimal representation of the machine code. The albaCore simulator `albasimh` runs the assembled machine code program for observing changes in register and memory locations after the execution of each instruction.

The executable files for the albaCore assembler and simulator are installed on the CSE department servers. Type the following at the Linux command line to add them to your path. You should also add it to your `.bashrc` file so that you don't need to retype it at the command line each time you log in.

```
export PATH=$PATH:/escnfs/courses/sp22-cse-20221.01/public/bin
```

To test that you now can access the tools type

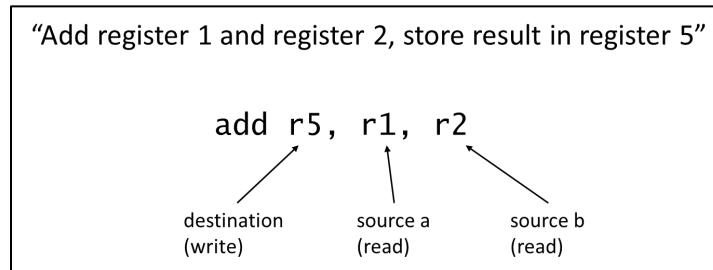
```
which albasimh
```

and it should display the path to the tools.

### 3.2.3 Arithmetic/Logic Instructions

#### 3.2.3.1 Operation and Instruction Encoding

The first category of albaCore instructions are the arithmetic and Boolean logic instructions. They take 3 register operands: 2 source operands and 1 destination (result) operand. An example is the `add` instruction is given below:



The table below lists the seven arithmetic and logic instructions for addition, subtraction, bitwise Boolean operations, and shifting, along with the load immediate instruction, and quit instructions. Note that these account for more than half of all of the albaCore instructions. The first three columns of the table describe the instructions and their operation, and the remaining columns describe how the instruction is encoded into 16 bits. The operations are described using a notation known as *register transfer* operations, where the destination register is on the left side of a left arrow ( $\leftarrow$ ) and the operation as a function of source registers is on the right side of the arrow.

instruction	name	operation	16-bit encoding			
			15:12	11:8	7:4	3:0
add rw, ra, rb	add	$rw \leftarrow ra + rb$	0x0	rw	ra	rb
sub rw, ra, rb	subtract	$rw \leftarrow ra - rb$	0x1	rw	ra	rb
and rw, ra, rb	bitwise and	$rw \leftarrow ra \& rb$	0x2	rw	ra	rb
or rw, ra, rb	bitwise or	$rw \leftarrow ra   rb$	0x3	rw	ra	rb
not rw, ra	bitwise not	$rw \leftarrow \sim ra$	0x4	rw	ra	x

<code>shl rw, ra, rb</code>	shift left	$rw \leftarrow ra \ll rb$	0x5	<code>rw</code>	<code>ra</code>	<code>rb</code>
<code>shr rw, ra, rb</code>	shift right	$rw \leftarrow ra \gg rb$	0x6	<code>rw</code>	<code>ra</code>	<code>rb</code>
<code>ldi rw, imm8</code>	load immediate	$rw \leftarrow imm8$	0x7	<code>rw</code>	<code>imm8</code>	
<code>quit</code>	quit	$PC \leftarrow PC$	0xF	X	X	X

The five operations `sub`, `and`, `or`, `shl`, and `shr` all follow the same pattern as the `add` example above, where the value written to the destination register `rw`, where `rw` is any register `r0-r15` is a function of any two sources registers `ra` and `rb`. The `not` instruction is similar, but has only one source register `ra`.

All albaCore instructions are encoded into 16 bits. The 4 most-significant bits (15:12) are the *operation code (opcode)* that identifies the instruction. Depending on the instruction, the remaining 12 bits have different interpretations. For each of the arithmetic/logic instructions, bits 11:8 specify the destination register `rw`, bits 7:4 specify `ra` and bits 3:0 specify `rb`. Examples of encodings for several instructions are shown below.

instruction	encoding
<code>add r5, r1, r2</code>	0512
<code>or r10, r10, r4</code>	3aa4
<code>not r7, r12</code>	r7c0

The load immediate instruction (`ldi`) loads an 8-bit constant into the 8 least-significant bits (7:0) of a 16-bit register and sets the upper bits of the register to zeros. The reason why the constant is only 8 bits rather than 16 is because all machine code instructions are 16 bits long, and there isn't room in the encoding to represent a longer constant. Example encodings of `ldi` instructions are shown below.

instruction	encoding
<code>ldi r8, 9</code>	7809
<code>ldi r10, 0xef</code>	7aef

The quit instruction terminates execution of the program. It does not have any destination or source registers and its 16-bit encoding only uses the opcode field.

instruction	encoding
<code>quit</code>	f000

The following examples illustrate the translation of some simple C statements into albaCore assembly language, assuming that variables `x`, `y`, and `z` are stored in registers `r1`, `r2`, and `r3`, respectively:

C	albaCore assembly
<code>x = 5;</code>	<code>ldi r1, 5</code>
<code>x = y + z;</code>	<code>add r1, r2, r3</code>
<code>x = x + y - z;</code>	<code>add r1, r1, r2</code> <code>sub r1, r1, r3</code>

Since the load immediate instruction `ldi` only takes an 8-bit constant as an argument, it requires several steps to load a 16-bit constant into a register. This is done by loading the upper byte of the constant into the register, shifting it 8 bits to the left, and then using an `or` instruction to set the lower bits as shown below.

C	albaCore assembly
<code>x = 0abcd;</code>	<pre> ldi r1, 0xab    // write high byte to r1 as 0x00ab ldi r0, 8       // write shift amount 8 to r0 shl r1, r1, r0 // shift r1 left 8 bits as 0xab00 ldi r0, 0xcd    // low byte to r0 as 0x00cd or  r1, r1, r0  // or low and high bytes together as 0abcd </pre>

### 3.2.3.2 Assembling and Simulating a Program

Like the albaCore processor itself, the albaCore assembly language and its directives, as well as the simulation tools were designed to be minimally sufficient and easy to use. As a first example, consider the following program `simple_demo.s`, where the “`.s`” extension is the convention for an assembly language program.

```

// albaCore simple example
.text
ldi r0, 5
ldi r1, 0xf
add r2, r0, r1
quit

```

albaCore assembly language programs recognize two memory segments: a text segment for program instructions and a data segment for variables and other forms of data, including the stack. The `.text` assembler directive denotes that the next lines are instructions that should be assembled into the text segment in memory. Constants may be specified in either decimal or hex, where hex constants have the prefix `0x` as in C/C++. Comments are also preceded with `//` or between `/* */` as in C/C++.

To assemble the program, type:

```
albaasmh simple_demo.s simple_demo.mem
```

This will produce a text file `simple_demo.mem` contains a *memory image* of the assembled program that the binary data in hexadecimal format that should be loaded into memory to run the program. Each row of the memory image file specifies an address and the data stored at that address, with the following format:

```
@address 16_bit_hex_value // translated_assembly_instruction or label
```

The memory image file `ex0_simple.mem` is as follows:

```
$ cat simple_demo.mem
// .text
@0000 7005 // ldi r0, 5
@0001 710f // ldi r1, 15
```

```

@0002 0201 // add r2, r0, r1
@0003 f000 // sys 0

// .data

```

The text segment with the machine code for the program instructions starts at address **0000** and ends at address **0003**. The program doesn't use the data segment, which would start right after the text segment.

To simulate the program in interactive mode, type:

```
albasimh -i simple_demo.mem
```

Type ‘h’ to get a list of commands:

```

h      help
q      quit
s      step 1 instruction
c      continue to end
v      toggle verbose instruction mode
rh     display registers (hex)
rd     display registers (dec)
l [n]   list up to n instructions (default 4 if n unspecified)
memh [addr]    display memory contents at addr (hex), addr is in hex
memd [addr]    display memory contents at addr (dec), addr is in hex

```

Type ‘s’ to single-step through the program or ‘c’ to continue execution until the program quits. The resulting trace of the program is as follows:

0000: 7005 ldi r0, 5	r0 = 0x5 (5)
0001: 710f ldi r1, 15	r1 = 0xf (15)
0002: 0201 add r2, r0, r1	r2 = 0x14 (20)
0003: f000 sys, 0	

### 3.2.4 Data Transfer Instructions

#### 3.2.4.1 Operation and Instruction Encoding

albaCore had two data transfer instructions: load (**ld**), which loads a register with the contents of a memory address and store (**st**), which stores data from a register to a memory address. For both load and store, the memory address is specified as the value in the source register **rb** plus a 4-bit offset. This way of specifying the address, called *base + offset*, is particularly handy for accessing values in a small array of up to 16 elements. The albaCore load and store instructions are specified in the table below.

instruction	name	operation	16-bit encoding			
			15:12	11:8	7:4	3:0
ld rw, rb, imm4	load (from mem)	$rw \leftarrow M[rb+imm4]$	0x8	rw	imm4	rb
st ra, rb, imm4	store (to mem)	$M[rb+imm4] \leftarrow ra$	0x9	imm4	ra	rb

As a first example of a data transfer instruction, suppose that we want to store the value 5 at memory address `0xf0`. To do this, we would write the value `0xf0` into a register to serve as the base address. We would write the value 5 into another register to serve as the source to be stored in memory. Then we could use a store instruction with an offset of `0` to write the source register value to the base address + `0`.

```
ldi r0, 0xf0      // base address
ldi r1, 5         // source value
st  r1, r0, 0     // M[r0+0] ← r1
```

We could read the value from memory back into another register with a load instruction

```
ld  r2, r0, 0    // r2 ← M[r0+0]
```

The encoding of load and store instructions is somewhat less intuitive than the encoding of the arithmetic/logic instructions. Like all albaCore instructions, the four most significant bits `15:12` are the opcode. Both load and store use the `rb` field in bits `3:0`, same as the arithmetic/logic instructions. Only the load instruction needs a destination register `rw`, since it reads a value from memory and writes it to a register. We use the same field for `rw` as we did for the arithmetic/logic instruction, bits `11:8`. Only the store instruction need a source register `ra`, since it writes a value from a source register to memory, and similarly, we use the same field for this as we did for the arithmetic/logic instructions, bits `7:4`. This leaves different fields remaining for the offset for the two instructions, bits `7:4` for the load and bits `11:8` for the store. Below are examples of encodings for two load and store instructions.

instruction	operation	encoding
<code>ld r3, r5, 0</code>	$r3 \leftarrow M[r5+0]$	8305
<code>st r3, r5, 0</code>	$M[r5+0] \leftarrow r3$	9035

### 3.2.4.2 Accessing Small Arrays Using Offsets

As mentioned earlier, base + offset addressing is primarily useful for reading and writing to small arrays of up to 16 elements. To illustrate, suppose that we want to write the values 7, 8, and 9 to successive memory addresses `0xf0`, `0xf1`, and `0xf2`. The simplest way to do this would be to use the same base address `0xf0` and just changed the offset for each of the three store instructions. Here is a complete albaCore assembly language program for this operation.

```
.text
ldi r0, 0xf0
ldi r1, 7
st  r1, r0, 0
ldi r1, 8
st  r1, r0, 1
ldi r1, 9
st  r1, r0, 2
quit
```

The simulated execution trace is as follows.

0000: 70f0 ldi r0, 240	r0 = 0xf0 (240)
------------------------	-----------------

0001: 7107 ldi r1, 7	r1 = 0x7 (7)
0002: 9010 st r1, r0, 0	M[00f0] = 0x0007 (7)
0003: 7108 ldi r1, 8	r1 = 0x8 (8)
0004: 9110 st r1, r0, 1	M[00f1] = 0x0008 (8)
0005: 7109 ldi r1, 9	r1 = 0x9 (9)
0006: 9210 st r1, r0, 2	M[00f2] = 0x0009 (9)
0007: f000 sys, 0	

In order to store data at some known constant memory address, we must first get the address into `rb`, typically using a load immediate (`ldi`) instruction. For example, the following instruction sequence stores the contents of register `r2` at address `0xff`:

```
ldi r0, 0xff
st r2, r0, 0
```

Address `0xff` is the highest address that we can represent in 8 bits. In order to access a higher address, we need to be able to get a full 16 bit constant value into a register. We can do this with a sequence of shift and or operations. For example, the following instruction sequence stores the contents of register `r2` at address `0x01ff`:

```
ldi r0, 8          // write 8 (shift amount) into r0
ldi r1, 0x01        // write 0x01 into r1
shl r1, r1, r0      // shift r1 left by 8 bits to 0x0100
ldi r0, 0xff        // write 0xff into r0
or r1, r1, r0        // or r0 with r1 so that r1 equals 0x01ff
st r2, r1, 0        // store r2 at 0x01ff
```

### 3.2.4.3 Data Segment, Labels, and `high( )`, `low( )` Directives

The albaCore assembler uses the data segment for storing variables and other data in memory. Before illustrating the use of the data segment, we will first review how to store a value at a fixed location in memory. The following example `st_demo.s` stores the value 7 at address `0abcd`. Recall that since the address `0abcd` is 16 bits and the load immediate instruction `ldi` only takes an 8-bit argument, we need to load the high and low bytes of the address into a register separately and arrange them together using `shl` and `or` instructions.

```
// st_demo.s
// Store the value 7 at address 0abcd
.text
ldi r0, 0xab    // high byte of address
ldi r1, 8
shl r0, r0, r1
ldi r1, 0xcd    // low byte of address
or r0, r0, r1
ldi r2, 7        // data value
st r2, r0, 0
quit
```

Keeping track of fixed memory locations for variables is inconvenient not only for human assembly language programmers, but even more so for a compiler that generates assembly code. A better approach

is to use labels to refer to memory locations in the data segment, and let the assembler figure out the memory addresses associated with those labels.

The following assembly language program `st_label_demo.s` demonstrates the use of labels for memory addresses in the data segment.

```
// st_label_demo.s
// Store the value 7 at address with label v
.data
v: 0xcafe

.text
ldi r0, high(v) // high byte of address
ldi r1, 8
shl r0, r0, r1
ldi r1, low(v) // low byte of address
or r0, r0, r1
ldi r2, 7        // data value
st r2, r0, 0
quit
```

The `.data` assembly language directive declares that the following lines of the program are labeled memory locations that should be placed in the data segment. The next line

`v: 0xcafe`

declares `v` as a label for a memory address in the data segment, with an initial value of `0xcafe` stored at the address.

The figure below illustrates the memory organization for the assembled program. The program has two memory segments, text and data. The text segment starts at address 0 and continues for as many address as needed to hold the program instructions. The data segment starts at the next address after the top of the text segment. The label `v` corresponds to the first address in the data segment, and the initial value stored there is 0.

label	address	value	segment
v:	bottom of data segment ffff	cafe	data
	top of text segment 0000	program instructions	text

The programmer (or compiler) doesn't easily know what the address of label `v` is, since that depends on the number of instructions in the program, but still needs to be able to access the high and low bytes of that unknown address in order to get the address into a register prior to performing the store instruction. The albaCore assembly language provides two directives, `high(Label)` and `low(Label)` to get the high and low bytes of an address associated with a label. Note that these are assembler directives and are *not* albaCore instructions. They are evaluated by the assembler, not by the processor.

Here is the memory image file `st_label_demo.mem` after the program is assembled:

```
$ albaasmh st_label_demo.s st_label_demo.mem
$ cat st_label_demo.mem
// .text
@0000 7000 // ldi r0, 0    target label: v
@0001 7108 // ldi r1, 8
@0002 5001 // shl r0, r0, r1
@0003 7108 // ldi r1, 8    target label: v
@0004 3001 // or r0, r0, r1
@0005 7207 // ldi r2, 7
@0006 9020 // st r2, r0, 0
@0007 f000 // sys 0

// .data
@0008 cafe //      label: v
```

The diagram shows the assembly code with annotations. Two boxes with arrows point to specific instructions. The first box points to the instruction at address 0000, which is `ldi r0, 0`. It contains the text `target label: v` and `high(v) = 00`. The second box points to the instruction at address 0003, which is `ldi r1, 8`. It contains the text `target label: v` and `low(v) = 08`.

The text segment goes from address `0000` through address `0007`. The data segment begins at address `0008`, which is the address corresponding to the label `v`. For the `ldi` instruction at address `0000`, the directive `high(v)` was replaced with the high byte of the address of `v`, which is `0`, and `low(v)` was replaced with the low byte of the address of `v`, which is `8`.

The simulated execution trace shows the value `7` stored at memory address `0008`.

```
$ albasimh -i st_label_demo.mem
albasim> c
0000: 7000 ldi r0, 0          | r0 = 0x0 (0)
0001: 7108 ldi r1, 8          | r1 = 0x8 (8)
0002: 5001 shl r0, r0, r1    | r0 = 0x0 (0)
0003: 7108 ldi r1, 8          | r1 = 0x8 (8)
0004: 3001 or r0, r0, r1     | r0 = 0x8 (8)
0005: 7207 ldi r2, 7          | r2 = 0x7 (7)
0006: 9020 st r2, r0, 0       | M[0008] = 0x0007 (7)
0007: f000 sys, 0
```

You can also initialize a series of memory locations with values, starting at a labeled address. In this case, we can think of the label as the starting address of an array. In the following example, 4 successive memory addresses starting at the address with label `A` are initialized with the values `1, 2, 3, 4` prior to declaring the label `v`.

```
// st_label_demo.s
// Store the value 7 at address with label v
.data
A: 1, 2, 3, 4
v: 0xcafe

.text
ldi r0, high(v) // high byte of address
ldi r1, 8
```

```

shl r0, r0, r1
ldi r1, low(v) // low byte of address
or r0, r0, r1
ldi r2, 7 // data value
st r2, r0, 0
quit

```

In assembled memory image file, we note that the 4 values starting at label A are found in addresses **0008** through **000b** and that the label **v** now corresponds to address **000c**, or decimal value **12**. The assembler directives **high(v)** and **low(v)** are evaluated by the assembler to reflect this change.

```

$ albaasmh st_label_array_demo.s st_label_array_demo.mem
$ cat st_label_array_demo.mem

```

```

// .text
@0000 7000 // ldi r0, 0      target label: v   high(v) = 00
@0001 7108 // ldi r1, 8
@0002 5001 // shl r0, r0, r1
@0003 710c // ldi r1, 12      target label: v   low(v) = 0c (decimal 12)
@0004 3001 // or r0, r0, r1
@0005 7207 // ldi r2, 7
@0006 9020 // st r2, r0, 0
@0007 f000 // sys 0

// .data
@0008 0001 // label: A
@0009 0002 //
@000a 0003 //
@000b 0004 //
@000c cafe //      label: v

```

The simulated execution now shows the value 7 stored at memory address **000c**

```

$ albasimh -i st_label_array_demo.mem
albasim> c
0000: 7000 ldi r0, 0          | r0 = 0x0 (0)
0001: 7108 ldi r1, 8          | r1 = 0x8 (8)
0002: 5001 shl r0, r0, r1    | r0 = 0x0 (0)
0003: 710c ldi r1, 12          | r1 = 0xc (12)
0004: 3001 or r0, r0, r1      | r0 = 0xc (12)
0005: 7207 ldi r2, 7          | r2 = 0x7 (7)
0006: 9020 st r2, r0, 0        | M[000c] = 0x0007 (7)

```

### 3.2.4.4 Example: Arithmetic Operations on Array Elements

This example illustrates an albaCore assembly language implementation of the following C program:

```
int A[3] = {3, 4, 0};
```

```

main()
{
    A[2] = A[0] + A[1];
}

```

The assembly language program places the array **A** in the data segment and the instructions in the text segment of memory.

```

.data
A: 3, 4, 0

.text
ldi r0, high(A)      // r0 = A
ldi r1, 8
shl r0, r0, r1
ldi r1, low(A)
or r0, r0, r1
ld r1, r0, 0          // r1 = A[0]
ld r2, r0, 1          // r2 = A[1]
add r1, r1, r2        // r1 = r1 + r2
st r1, r0, 2          // A[3] = r1
quit

```

The assembled machine code memory image file is shown below. The directives **high(A)** and **low(A)** are replaced with the starting address of the array **A** in the data segment, which is 10 decimal (0x000a).

```

// .text
@0000 7000 // ldi r0, 0           target label: A
@0001 7108 // ldi r1, 8
@0002 5001 // shl r0, r0, r1
@0003 710a // ldi r1, 10         target label: A
@0004 3001 // or r0, r0, r1
@0005 8100 // ld r1, r0, 0
@0006 8210 // ld r2, r0, 1
@0007 0112 // add r1, r1, r2
@0008 9210 // st r1, r0, 2
@0009 f000 // sys 0

// .data
@000a 0003 // label: A
@000b 0004 //
@000c 0000 //

```

The simulated execution trace is as follows:

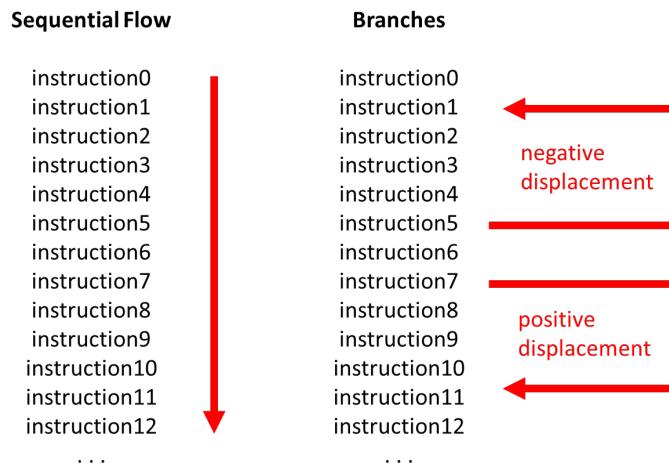
0000: 7000 ldi r0, 0	r0 = 0x0 (0)
0001: 7108 ldi r1, 8	r1 = 0x8 (8)
0002: 5001 shl r0, r0, r1	r0 = 0x0 (0)
0003: 710a ldi r1, 10	r1 = 0xa (10)
0004: 3001 or r0, r0, r1	r0 = 0xa (10)

0005: 8100	ld r1, r0, 0	r1 = 0x3 (3)
0006: 8210	ld r2, r0, 1	r2 = 0x4 (4)
0007: 0112	add r1, r1, r2	r1 = 0x7 (7)
0008: 9210	st r1, r0, 2	M[000c] = 0x0007 (7)
0009: f000	sys, 0	

### 3.2.5 Branch Instructions

#### 3.2.5.1 Operation and Instruction Encoding

Branch instructions specify the address after the branch is taken in terms of a *displacement* from the current pc value. This displacement can be positive or negative.



Branch instructions can either be unconditional or conditional. For conditional branches, if the branch condition is met, then program execution continues with the instruction at memory address  $pc + \text{displacement}$ . Otherwise, it continues at the next sequential address. albaCore has three branch instructions: branch unconditionally (**br**), branch if register **rb** equals zero (**bz**), and branch if register **rb** is negative (**bn**). The displacement in albaCore is an 8-bit 2's complement signed integer, **imm8**. The notation **SignEx(imm8)** means that the 8-bit immediate value **imm8** should be interpreted as a signed value and *sign-extended* to 16-bits before adding it to PC. This simply means that if bit 7 is a 0, the number is positive and bits 15:8 are set to 0 and if bit 7 is a 1, then the number is negative and bits 15:8 are set to 1. (Note that this is different from the offsets for load and store instructions, which are always considered to be positive).

instruction	name	operation	16-bit encoding			
			15:12	11:8	7:4	3:0
br imm8	branch (uncond)	$PC \leftarrow PC + \text{SignEx}(\text{imm8})$	0xA	imm8	X	
bz rb, imm8	branch if zero	$\text{if } (rb == 0)$ $PC \leftarrow PC + \text{SignEx}(\text{imm8})$ $\text{else}$ $PC \leftarrow PC + 1$	0xB	imm8	rb	
bn rb, imm8	branch if negative	$\text{if } (rb < 0)$	0xC	imm8	rb	

		PC $\leftarrow$ PC + SignEx(imm8) else PC $\leftarrow$ PC + 1			
--	--	---	--	--	--

The 8-bit displacement is encoded in bits 11:4 of the instruction. Below are example encodings of branch instructions.

instruction	encoding	
br 2	a020	
bz r3, 2	b023	
bn r3, -2	cfe3	displacement 0xfe = -2

To illustrate the use of branches, consider the translation of the C code for a `while` loop into albaCore assembly language.

```
i = 4;
while (i >= 0)
    i = i - 1;
x = 0xff;
```

The `while` loop is translated to two branch statements, a conditional branch `bn` at the top at `loop_cond` that will break out of the loop to `continue` when the branch condition is met, and an unconditional branch `br` at the bottom that branches back to testing the loop condition at the top. The displacement for the conditional branch is positive (3 instructions from the branch to `continue`) and the displacement for the unconditional branch is negative (-2 instructions from the branch to `loop_cond`).

```
.text
    ldi r1, 1      // let r1 be constant 1
    ldi r2, 4      // let r2 be i; i = 4
loop_cond: bn r2, 3      // if (i < 0) go to continue (ahead 3)
            sub r2, r2, r1 // i = i - 1
            br -2          // go to loop_cond (back 2)
continue: ldi r3, 0xff // let r3 be x; x = 0xff
         quit
```

The assembled machine code in the memory image file is as follows:

```
// .text
@0000 7101 // ldi r1, 1
@0001 7204 // ldi r2, 4
@0002 c032 // bn r2, 3      label: loop_cond
@0003 1221 // sub r2, r2, r1
@0004 afe0 // br -2
@0005 73ff // ldi r3, 255   label: continue
@0006 f000 // sys 0
```

Note that the conditional branch instruction `bn` is at memory address 2 and the unconditional branch instruction `br` is at address 4. We can clearly observe the behavior of the two branches in the simulated execution trace, where the conditional branch at address 2 is not taken, and the program counter

increments by 1, as long as the value in r2 is greater than or equal to 0. Further, the program counter always branches from address 4 back to address 2 as a result of the unconditional branch with a displacement of -2. Once  $r2 = 0xffff$ , the conditional branch at address 2 is finally taken, and the program counter increments by a displacement of 3 to address 5.

0000: 7101 ldi r1, 1	r1 = 0x1 (1)
0001: 7204 ldi r2, 4	r2 = 0x4 (4)
0002: c032 bn r2, 3	
0003: 1221 sub r2, r2, r1	r2 = 0x3 (3)
0004: afe0 br -2	
0002: c032 bn r2, 3	
0003: 1221 sub r2, r2, r1	r2 = 0x2 (2)
0004: afe0 br -2	
0002: c032 bn r2, 3	
0003: 1221 sub r2, r2, r1	r2 = 0x1 (1)
0004: afe0 br -2	
0002: c032 bn r2, 3	
0003: 1221 sub r2, r2, r1	r2 = 0x0 (0)
0004: afe0 br -2	
0002: c032 bn r2, 3	
0003: 1221 sub r2, r2, r1	r2 = 0xffff (-1)
0004: afe0 br -2	
0002: c032 bn r2, 3	
0005: 73ff ldi r3, 255	r3 = 0xff (255)
0006: f000 sys, 0	

### 3.2.5.2 Use of Labels for Assembly Language Programming

Keeping track of branch displacements manually is tedious; fortunately, the assembler can automate this by using labels. Here is the same example rewritten with labels:

```
.text
ldi      r1, 1           // let r1 be constant 1
          ldi r2, 4           // let r2 be i; i = 4
loop_cond: bn r2, continue // if (i < 0) go to continue
            sub r2, r2, r1   // i = i - 1
            br loop_cond      // go to loop_cond
continue: ldi r3, 0xff     // let r3 be x; x = 0xff
          quit
```

The assembled machine code memory image file shows the labels replaced with the appropriate displacements.

```
// .text
@0000 7101 // ldi r1, 1
@0001 7204 // ldi r2, 4
@0002 c032 // bn r2, 3      label: loop_cond    target label: continue
@0003 1221 // sub r2, r2, r1
@0004 afe0 // br -2          target label: loop_cond
@0005 73ff // ldi r3, 255   label: continue
@0006 f000 // sys 0
```

## 3.2.6 Jump Instructions

### 3.2.6.1 Operation and Instruction Encoding

While branch instructions change the program counter to a new location that is some displacement away from the current locations, jump instructions change the program counter to some new absolute address. Jump instructions are used primarily to implement procedure calls and returns.

The jump register instruction (**jr**) changes the program counter to the 16-bit value contained in source register **ra**.

The jump and link instruction (**jal**) provides the mechanism for calling procedures and knowing where to return from them. It does two things:

- it changes the program counter to an absolute target location that is specified with the instruction
- it saves the value of the old program counter plus 1 in register 15.

instruction	name	operation	16-bit encoding				
			15:12	11:8	7:4	3:0	
<b>jal imm12</b>	jump and link	$PC \leftarrow \{PC[15:12], imm12\}$ $r15 \leftarrow PC + 1$	0xD	imm12			
<b>jr ra</b>	jump register	$PC \leftarrow ra$	0xE	X	ra	X	

Some example instruction encodings are as follows:

instruction	encoding
<b>jal 0xabc</b>	dabc
<b>jr r15</b>	e0f0

Because all albaCore instructions are encoded into 16 bits, there are only 12 bits available for encoding the target address for **jal**. In order to form a complete 16-bit address for the target, albaCore takes the 4 most significant bits of the program counter. For example, if **jal 0xabc** was called while the program counter was **0x005a**, then the program counter would jump to **0x0abc**. If the program counter was **0x105a**, however, then the program counter would jump to **0x1abc**.

The following example illustrates a procedure call and return using **jal** and **jr** that could be the result of compiling the following C program:

```
int main()
{
    int x = 5;
    int y = mult2(x);
}

int mult2(x)
{
    return x + x;
```

```
}
```

First, we will show the albaCore assembly language program with the address of the function `mult2` hardcoded; after, we will show how the use of labels makes writing the assembly program simpler. The numbers in square brackets are the addresses of where the instructions will be placed in memory. In this example, the function `mult2` starts at address 4. When the program reaches the `jal` instruction at address 2, it stores `pc + 1`, which is 2, in `r15` and sets `pc` to the target address 4. Execution continues at the instruction at address 4, where it does an add and stores the result in `r0`. The `jr` instruction at address 5 sets the `pc` to the value in `r15`, which is the return address following the call.

```
.text
[0]:      ldi r1, 5
[1]:      jal 4          // call: jump to 4, save 2 in r15
[2]:      or  r2, r0, r0 // copy r0 to r2 (return value from call)
[3]:      quit
[4]: mult2: add r0, r1, r1 // r0 = 2*r1 (return value)
[5]:      jr r15        // return: jump to address in r15 (2)
```

The assembled machine code memory image file is as follows:

```
// .text
@0000 7105 // ldi r1, 5
@0001 d004 // jal 0x4
@0002 3200 // or r2, r0, r0
@0003 f000 // sys 0
@0004 0011 // add r0, r1, r1    label: mult2
@0005 e0f0 // bn r15
```

The simulated execution trace shows the call-and-return behavior. When the `jal 0x4` is executed at address 1, the next address 2 is copied to `r15` as the return address, and execution continues at address 4. When the `jr r15` is executed, the value in `r15`, which is 2, is copied back to the program counter and execution continues at address 2.

0000: 7105 ldi r1, 5	r1 = 0x5 (5)
0001: d004 jal, 4	r15 = 0x2 (2)
0004: 0011 add r0, r1, r1	r0 = 0xa (10)
0005: e0f0 jr, r15	
0002: 3200 or r2, r0, r0	r2 = 0xa (10)
0003: f000 sys, 0	

### 3.2.6.2 Use of Labels for Assembly Language Programming

Using the jump-and-link instruction to call a function requires knowledge of where each function is stored in memory. The albaCore assembler can automatically determine this with labels. Instead of specifying the argument to `jal` as a numerical target address, you can instead specify the label of the starting instruction of the function. Here is the example from above rewritten with a label as an argument to `jal`.

```
.text
main: ldi r1, 5
      jal mult2      // call mult2
```

```

        or r2, r1, r1 // copy r1 to r2
        quit

mult2: add r0, r1, r1 // r0 = 2*r1
       jr r15          // return

```

The assembled machine code memory image file replaces the label with the appropriate target address.

```

// .text
@0000 7105 // ldi r1, 5      label: main
@0001 d004 // jal 0x4        target label: mult2
@0002 3211 // or r2, r1, r1
@0003 f000 // sys 0
@0004 0011 // add r0, r1, r1 label: mult2
@0005 e0f0 // bn r15

```

### 3.3 Writing Longer Assembly Language Programs

#### 3.3.1 Using C goto Statements to Structure Assembly Programs

Let's state upfront: it is very uncommon for humans to write assembly language programs by hand. In nearly all cases, compilers do a far superior job in translating a program from a high-level language such as C to assembly language than a human programmer possibly could. The purpose of this and following sections is *not* to turn you into an assembly language programmer as an end in itself, but rather to use assembly language programming as a vehicle for understanding how processors and programs work at a very basic level.

A useful approach to manually translating C programs to assembly language is to express if-else statements and loops using C `goto` statements and labels. Many programmers don't even know that these exist and they are generally discouraged because they operate at such a low level of abstraction when compared to alternatives such as `while` and `for` statements. But this is precisely why they are useful as a guide for writing assembly language programs—they are similar to assembly language instructions.

To illustrate the process, we will take an `if-else` statement and translate it to `goto` statements and labels, and then translate this code to albaCore assembly language. The original `if-else` code is given below:

```

int x = 1;
int y = 2;
int z;

if (x < y)
    z = 1;
else
    z = 2;
x = x + z;

```

Our goal in rewriting the code with `goto` statements and labels is to get it close to albaCore assembly language, taking into consideration the features and limitations of the albaCore instruction set, while still being a valid C program that we can compile and test. In this example, we write the code to anticipate

using the conditional branch-if-negative instruction `bn` and unconditional branch `br`. To use the `bn` instruction, we will need to test the value of a register, so we replace the condition ( $x < y$ ) with the difference  $x - y$  and store the result in a temporary variable `tmp` that we can test.

```

int tmp;
int x;
int y;
int z;

tmp = x - y;
if (tmp < 0) goto if_block;
else_block:
z = 2;
goto cont;
if_block:
z = 1;
cont:
x = x + z;

```

Given this form, we can now translate to assembly language where each line of the C program maps directly to short sequence of assembly language instructions. A recommended technique is to first copy C program with `goto` statements into the assembly language file and make them comments, define which registers to use for each of the local variables, and then fill in the assembly language instructions following the commented-out C statements.

```

//      int tmp;    // r0
ldi r1, 1      //      int x = 1; // r1
ldi r2, 2      //      int y = 2; // r2
//      int z;      // r3
//
sub r0, r1, r2 //      tmp = x - y;
bn  r0, if_block //      if (tmp < 0) goto if_block;
else_block:     // else_block:
    ldi r3, 2   //      z = 2;
    br  cont    //      goto cont;
if_block:       // if_block:
    ldi r3, 1   //      z = 1;
cont:           // cont:
    add r1, r1, r3 //      x = x + z

```

### 3.3.2 Maxfinder

The Maxfinder is a longer program that demonstrates operations on an array of data in global memory. The objective of the program is to find the maximum value in an array of positive integers. Here is a complete C program using a `while` loop and `if` statements for comparisons that we will first translate to a program using `goto` statements and then translate to albaCore assembly language.

```
#include <stdio.h>
int A[8] = {3, 1, 4, 2, 15, 5, 32, 9};
```

```

int main()
{
    int max = 0;
    int i = 0;
    while (i < 8) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    printf("%d\n", max);
}

```

Here is the translation of program to C **goto** form. As with the **if-else** example in the previous section, the goal is to bring the program close to an albaCore assembly language program, considering the features and limitations of the instruction set. In particular, this program targets the way that albaCore reads values from memory using the load instruction, **ld**. To use **ld**, the base memory address needs to be in a register and the result of the load will go into another register. The **ld** instruction also takes an offset as a parameter, but we can't take advantage of that for this example. The reason is that the offset is a constant that is specified as a field of the instruction; it is not possible to change it under program control as we need to do with the index **i**. The program also targets the albaCore conditional branch instructions **bz** and **bn**, and the unconditional branch instruction **br**.

```

#include <stdio.h>
int A[8] = {3, 1, 4, 2, 15, 5, 32, 9};

int main()
{
    int tmp;
    int max = 0;
    int i = 0;
    int size = 8;
    int d;           // emulates register with data read from memory
    int *a = A;      // emulates writing the starting address of A in a register

while_condition:
    tmp = size - i;
    if (tmp == 0) goto done;
    d = a[i];
    tmp = d - max;
    if (tmp < 0) goto update_i;
    max = d;
update_i:
    i = i + 1;
    goto while_condition;
done:
    printf("%d\n", max);
}

```

Here is the translation of the C program in goto form to albaCore assembly language. In place of the `printf` statement, the program copies the final value of `max` to register `r0` to make it easy to spot in simulation.

```
.data
        // int A[8] = {3, 1, 4, 2, 15, 5, 32, 9};
A: 3, 1, 4, 2, 15, 5, 32, 9

.text
main:           // int main()
                //
                // {
                //     int tmp;          // r0
ldi r1, 0       //     int max = 0;      // r1
ldi r2, 0       //     int i = 0;       // r2
ldi r3, 8       //     int size = 8;    // r3
                //     int d;          // r4
ldi r5, high(A) //     int *a = A    // r5
ldi r0, 8
shl r5, r5, r0
ldi r0, low(A)
or r5, r5, r0
                //
while_condition: // while_condition:
sub r0, r3, r2 //     tmp = size - i;
bz r0, done    //     if (tmp == 0) goto done;
add r0, r5, r2 //     d = a[i];
ld r4, r0, 0
sub r0, r4, r1 //     tmp = d - max;
bn r0, update_i //     if (tmp < 0) goto update_i;
or r1, r4, r4 //     max = d;
update_i:       // update_i:
ldi r0, 1       //     i = i + 1;
add r2, r2, r0
br while_condition //     goto while_condition;
done:           // done:
or r0, r1, r1   //     printf("%d\n", max);
quit            // }
```

## 3.4 Function Calls

### 3.4.1 Simple Function Call with Parameters Passed via Registers

One of the chief mechanisms that we use to make programs modular is functions. albaCore uses the jump-and-link (`jal`) and jump register (`jr`) instructions to call and return from functions.

Function calls also involve passing parameters from the function that makes the call (the *caller*) to the function that is called (the *callee*) and then returning values from the callee back to the caller. The example below illustrates the simplest mechanism for passing parameters and the return value using registers. In this example, registers `r1` and `r2` are used for passing parameters from the caller to the

callee and register `r0` is used to pass the return value back from the callee to the caller. Register `r15` contains the return address set by `jal` during the call and used by `jr` to return from the callee to the caller.

```
int A = {6, 4, 0}
main
{
    A[2] = add(A[0], A[1]);
}

int add(int a, int b)
{
    return(a + b);
}
```

variable	register
<code>retval</code>	<code>r0</code>
<code>a</code>	<code>r1</code>
<code>b</code>	<code>r2</code>
<code>A</code>	<code>r3</code>

address	label	pseudocode	assembly	hex
0	main:	<code>A = 32</code>	<code>ldi r3, 32</code>	7320
1		<code>a = M[A]</code>	<code>ld r1, r3, 0</code>	8103
2		<code>b = M[A+1]</code>	<code>ld r2, r3, 1</code>	8213
3		<code>call add</code>	<code>jal add</code>	D008
4		<code>M[A+2] = retval</code>	<code>st r0, r3, 2</code>	9203
5		<code>quit</code>	<code>quit</code>	F000
8	add:	<code>retval = a + b</code>	<code>add r0, r1, r2</code>	0012
9		<code>return</code>	<code>jr r15</code>	E0F0
32	A:			6
33				4
34				

### 3.4.2 The Call Stack

The previous example illustrated a simple function call and return, with parameters, the return value, and the return address passed via registers. While this approach worked for this simple example, a closer examination reveals that it isn't scalable to larger, more complex, and more realistic programs:

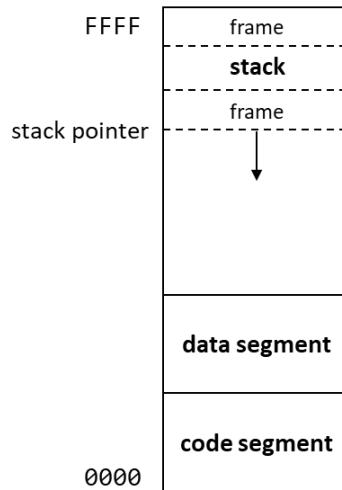
- **Overwritten return address:** A first concern is the use of register `r15` for storing the return address. Suppose that function `main` uses `jal` to call function `func1`. As long as `func1` doesn't call some other function, `func1` could return to `main` using a `jr r15` instruction. But if `func1` does call another function, say `func2`, then the `jal` call to `func2` would overwrite the value of `r15` with the return address to `func1`, and the value of the return address back to `main` would be lost.

- **Register conflict and modular programming:** The problem of the return address `r15` getting clobbered by subsequent function calls is indicative of the general problem of only having a finite number of registers to store variables. Even if we try to keep register usage between functions separate, eventually all of the registers will be used up. The whole point of using functions in programming is to be able to break large programs down into smaller, more manageable modules that can be written *independently* of each other. Even if we had a very large number of registers, it violates the principle of modularity if one function needs to be aware of which registers other functions are using so that it doesn't clobber them.
- **Not enough registers:** Finally, some functions may require more storage for local variables than the register set provides.

To overcome these issues, we need a structured way of allocating memory to each function call that meets all of its storage needs. The standard approach is to allocate memory for function calls on a *call stack* (or just “the stack”), where the space allocated for each function call is known as a *stack frame*. Each stack frame contains space for holding local variables, for backing up the return address, and for passing parameters and returning values between function calls.

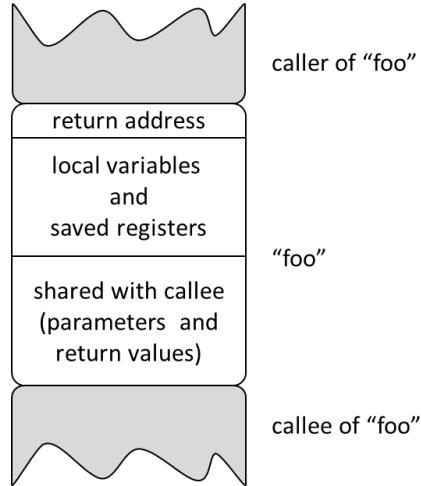
Note: Modern RISC processor conventions use a mix of registers and the stack for passing values between functions. In this section we focus on using the stack alone and later will show an example using a more modern approach that involves registers as well as the stack.

When a function is called, a new frame for that function is “pushed” onto the stack, and when the function returns, it is “popped” off of the stack. The *stack pointer* indicates the address where new frames are to be added. The stack is typically located at the upper end of the memory address space and grows downward, so that frames are pushed by decrementing the stack pointer and then popped by incrementing it.



The figure below illustrates the layout of a stack frame for a call of function `foo` somewhere in the middle of the stack. Above `foo`'s frame is the frame of the function that called `foo`—its *caller*—and below is the frame of a function that `foo` calls—its *callee*. The frame is organized into slots for different kinds of information. At the top of the frame is a copy of `foo`'s return address, so that it knows where to return to its caller. Below that is space for local variables and saved registers. At the bottom is space for sharing data between `foo` and its callee, specifically, values of parameters passed from `foo` to its callee,

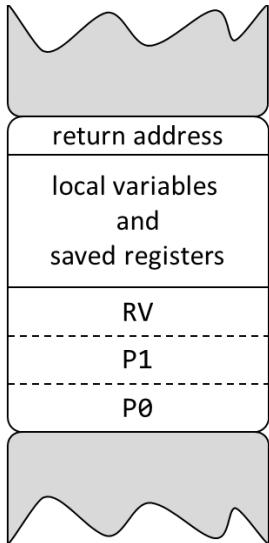
as well as space for the callee's return value. Before calling its callee, `foo` will copy the values of the parameters passed to the callee into the bottom of its own frame. After the call, the callee will reach up into `foo`'s frame to get the values of the parameters. Before it returns, the callee will similarly write the return value back into `foo`'s frame. Note that the callee doesn't need to know anything about `foo` other than expecting to find the values of parameters passed to it and an empty slot for placing its return value in the memory locations immediately above its own frame.



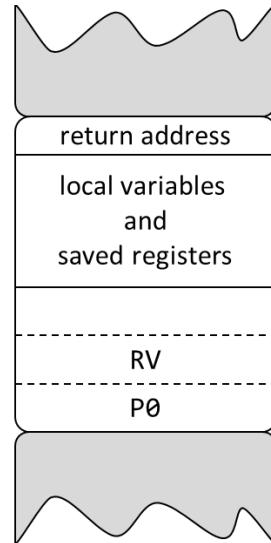
For the bottom part of the frame that is shared with callees, we'll adopt the ordering convention that from the bottom up, slots for parameters to the callee come first, followed by the slot for the return value. Note that if a function calls several callees, its frame must provide enough space to handle the callee with the most parameters. For example, suppose that `foo` calls 2 callees: `callee_A` with 2 input parameters and `callee_B` with 1 input parameter, as illustrated by the C program below.

```
int foo(int P0, int P1)
{
    int RV;
    RV = callee_A(P0, P1);
    RV = callee_B(P0);
}
```

`foo`'s frame would need to provide enough space for the 2 arguments and the return value for `callee_A`, and when `foo` calls `callee_B`, it would leave part of that space unused, as illustrated below:



stack during call from foo to callee\_A



stack during call from foo to callee\_B

### 3.4.3 Example: Using Stack Alone

#### 3.4.3.1 Sample C Program

To illustrate the use of the stack, consider the following C program with 3 functions, where `main` calls `func1`, which in turn calls `func2`.

```
int A;

main()
{
    A = func1(3, 4);
    exit;
}

int func1(int x, int y)
{
    return (func2(x) + y);
}

int func2(int n)
(
    return (n + n);
}
```

#### 3.4.3.2 Overview of Execution and Stack Behavior

Below is an outline of the execution of the program, detailing interactions with the stack. The example provides snapshots of the stack at significant points during the execution.

### Inside main:

- `main` pushes a frame onto the stack with space for 3 words: the 2 parameters (`P0, P1`) passed to callee `func1` and the return value (`RV`) received from `func1`. (Since `main` isn't returning anywhere, it doesn't need a slot for a return address in its frame).
- `main` writes the value 3 into the callee parameter 0 slot and the value 4 into the callee parameter 1 slot
- `main` calls `func1`

Stack			
main:	sp+2	RV	
	sp+1	P1	4
	sp	P0	3

### Inside func1:

- `func1` pushes a frame onto the stack with space for 3 words: the return address (`RA`), the 1 parameter (`P0`) passed to callee `func2`, and the return value (`RV`) from `func2`
- `func1` saves the return address in the top slot in its frame
- `func1` reads the value of its first parameter (`x`) from `main`'s frame and then copies it to the parameter 0 slot of its own frame in preparation for the call to `func2` as `n`
- `func1` calls `func2`

Stack			
main:	sp+5	RV	
	sp+4	P1	4
	sp+3	P0	3
func1:	sp+2	RA	r15
	sp+1	RV	
	sp	P0	3

### Inside func2:

- `func2` doesn't need to push a frame onto the stack since it doesn't call any other functions and doesn't require additional space for local variables
- `func2` reads the value of its parameter from `func1`'s frame
- `func2` performs the shift operation and writes the result to the return value slot in `func1`'s frame
- `func2` returns execution to the return address in `func1`

Stack			
main:	sp+5	RV	
	sp+4	P1	4
	sp+3	P0	3
func1:	sp+2	RA	r15
	sp+1	RV	12
	sp	P0	3

### Back inside func1:

- `func1` reads the value of its second parameter (`y`) from the parameter 1 slot of `main`'s frame
- `func1` reads the return value from the call to `func2` from its own frame and adds it to `y`
- `func1` writes the result of the addition to the return value slot in `main`'s frame

- `func1` copies the return address from its frame back to a register
- `func1` pops its frame
- `func1` returns execution to the return address in `main`

Stack			
main:	sp+2	RV	16
	sp+1	P1	4
	sp	P0	3

### Back inside main

- `main` reads the return value from its own frame
- `main` writes the value to global variable A
- The program quits execution

### 3.4.3.3 albaCore Program

Below is a complete implementation of the example C program in albaCore. In order to accentuate the role of the stack, the implementation uses a minimal set of registers.

register	usage
r0	temporary values
r1	temporary values
r14	stack pointer (sp)
r15	return value (rv)

Addr	Label	Comment	Pseudocode	Assembly	Hex
0	main:	initialize sp to address 255	sp = 255	ldi r14, 255	7EFF
1		push frame onto stack with space for 2 parameters and return value	r0 = 3 (FSIZE = 3)	ldi r0, 3	7003
2			sp = sp-r0	sub r14, r14, r0	1EE0
3		pass 3 to P0 slot in frame	r0 = 3	ldi r0, 3	7003
4			sp[P0] = r0	st r0, r14, 0	900E
5		pass 4 to P1 slot in frame	r0 = 4	ldi r0, 4	7004
6			sp[P1] = r0	st r0, r14, 1	910E
7		call func1	call func1	jal func1	D00E
8		get get func1 result from return value slot in frame	r0 = sp[RV]	ld r0, r14, 2	802E
9		store func1 result to address of global variable A	r1 = &A	ldi r1, 64	7140
10			M[r1] = r0	st r0, r1, 0	9001

11			r0 = FSIZE	ldi r0, 3	7003
12		pop frame	sp = sp+r0	add r14, r14, r0	0EE0
13		quit	quit	quit	F000
14	func1:	push frame with space for 1 parameter, return value, and return address	r0 = FSIZE (FSIZE = 3)	ldi r0, 3	7003
15			sp = sp-FSIZE	sub r14, r14, r0	1EE0
16		save return address in frame	sp[RA] = ra	st r15, r14, 2	92FE
17		receive value of x from P0 slot in caller's frame	r0 = sp[FSIZE+P0]	ld r0, r14, 3	803E
18		pass value of x to P0 slot in frame	sp[P0] = r0	st r0, r14, 0	900E
19		call func2	call func2	jal func2	D01C
20		get func2 return value from slot in frame	r0 = sp[RV]	ld r0, r14, 1	801E
21		receive value of y from P1 slot in caller's frame	r1 = sp[FSIZE+P1]	ld r1, r14, 4	814E
22		add func2 return value + value of y	r0 = r0 + r1	add r0, r0, r1	0001
23		pass addition result back to return value slot in caller's frame	sp[FSIZE+RV] = r0	st r0, r14, 5	950E
24		restore return address	ra = sp[RA]	ld r15, r14, 2	8F2E
25			r0 = FSIZE	ldi r0, 3	7003
26		pop frame	sp = sp+r0	add r14, r14, r0	0EE0
27		return to caller	return	jr r15	E0F0
28	func2:	receive value of n from P0 slot in caller's frame	r0 = sp[FSIZE+P0] (FSIZE = 0)	ld r0, r14, 0	800E
29		calculate n + n	r0 = r0 + r0	add r0, r0, r0	0000
30		pass shift result back to return value slot in caller's frame	sp[FSIZE+RV] = r0	st r0, r14, 1	910E
31		return to caller	return	jr r15	E0F0

### 3.4.4 Example: Modern RISC Approach Using Registers and Stack

#### 3.4.4.1 Caller-Saved vs. Callee-Saved Registers

The previous example showed how the stack could be used to handle all values passed between caller and callee functions, including arguments passed from the caller to callee and the return value passed by the callee back to the caller. There's a lot of overhead, however, in copying values back and forth between registers and the stack. One of the advents of RISC processors over earlier architectures was to increase the number of registers in order to reduce the use of the stack. As a result, the modern approach is to adopt a convention that uses a mix of registers and the stack for passing values between functions. It also

establishes rules for who is responsible for saving registers back and forth to the stack—the caller or the callee—before they get clobbered.

Before looking at how parameters are passed between functions, we consider strategies for saving and restoring registers on the stack so that their values aren't clobbered. Basically, there are two approaches:

- *caller-saved*: the “watch-out-for-your-own-tail” approach where the caller backs up registers that it is using to its own frame before a call—just in case the callee modifies them—and then restores the register values from its frame after the call.
- *callee-saved*: the “good neighbor” approach where a callee backs up register values to its frame before it modifies them and then restores them before returning to the caller, so that the registers appear undisturbed to the caller.

Typical RISC systems including ARM and MIPS use a mix of both strategies and designate some registers as caller-saved and others as callee-saved. If a function uses a callee-saved register, then it is obligated to save it to the stack before modifying it and then to restore it before returning to the caller. If a function uses a caller-saved register, then it only needs to back it up on the stack if it cares about the value being clobbered by a callee. In many cases, the caller doesn't care about the value being clobbered by a callee, since it won't be using the value of that register again after the call. The general convention is to use caller-saved registers as *temporary* registers where it doesn't matter if the value is clobbered by a function call, and to only use callee-saved registers for those cases where the value needs to be preserved across a function call.

### 3.4.4.2 albaCore Register Usage Conventions

Given this background, we're ready to define a register usage convention for albaCore that is based on the MIPS processor. We give each register name an alias that reflects its usage.

register	alias	usage convention
r0	rv	return value, may also be used as temporary. Caller-saved.
r1-r4	a0-a3	function arguments, used to pass values from caller to callee. Caller-saved
r5-r9	t0-t4	temporary/caller-saved, general-purpose registers
r10-r13	s0-s3	saved/callee-saved, general-purpose registers
r14	sp	stack pointer, callee-saved
r15	ra	return address, callee-saved

Usage notes:

- Up to the first 4 arguments of a function call are passed via a0-a3. If a function has more than 4 arguments, additional arguments are passed via the stack.
- A callee will typically clobber the values in a0-a3, especially if it itself uses them to call another function. If a caller wants to preserve these values across a call to a callee for later use, it should either copy them to saved registers s0-s3 (common) or save them in its own stack frame (typically done only after saved registers are used up).
- Register rv is the return value, which is typically modified by a callee, since that is its purpose. If a function has more than 1 return value (C-like languages don't support this), additional return values are passed via the stack.
- The return address register ra (r15) is the only register that has a “hardwired” usage, since the jal instruction explicitly modifies the value of this register. All other registers are technically general-purpose and their usage is by convention.
- The stack pointer sp (r14) is callee-saved, in that a callee allocates space for its own stack frame (pushing a frame onto the stack) by decrementing sp upon entrance, and then deallocates the

frame (popping the frame from the stack) by incrementing `sp` to where it was the caller just before it returns.

### 3.4.4.3 Overview of Execution

Implementing a C program using both register and the stack to pass parameters between functions requires considerably fewer steps than using the stack alone. This section gives an overview of the execution of the same program as before, using the new calling convention. For convenience, the C program is repeated below.

```
int A;

main()
{
    A = func1(3, 4);
    exit;
}

int func1(int x, int y)
{
    return (func2(x) + y);
}

int func2(int n)
{
    return (n + n);
}
```

#### Inside `main`:

- `main` doesn't need a frame because parameters will be passed by registers `a0-a1` and its callee's return value will be passed through register `rv`. Further, it doesn't need to back up the return address `ra` because it doesn't return anywhere and it doesn't need to back up any callee-saved registers because it doesn't have a caller.
- `main` writes the value 3 into `a0` slot and the value 4 into `a1`
- `main` calls `func1`

Stack	
<code>sp:</code>	initialized, but nothing on stack yet

#### Inside `func1`:

- `func1` pushes a frame onto the stack with space for 2 words: the return address (`ra`) and a copy of callee-saved register `s0`, which it will be modifying. (In this case, `func1`'s caller, `main`, didn't use `s0`, but because of the principle of modularity, there is no way that `func1` could know this).
- `func1` saves the return address `ra` in the top slot in its frame
- `func1` saves `s0` into the next slot in its frame.
- In preparation for calling `func2`, `func1` copies `a1` (parameter `y`) into `s0`, since it will need this value after the call and there are no guarantees that `func2` won't modify `a1`. (It

turns out that it doesn't, but because of the principle of modularity, there is no way that `func1` could know this).

- The first parameter to `func1` (passed as `x`) will be passed as the sole parameter to `func2` (as `n`). The value is already in `a0`, so nothing needs to be done.
- `func1` calls `func2`

Stack		
func1:	sp+1	ra (r15)
	sp	s0 (r10)

#### Inside `func2`:

- `func2` doesn't need to push a frame onto the stack since it doesn't call any other functions and doesn't require additional space for local variables
- `func2` performs the shift operation on register `a0` and writes the result to register `rv`
- `func2` returns execution to `func1` using the return address in `ra (r15)`.

Stack		
func1:	sp+1	ra (r15)
	sp	s0 (r10)

#### Back inside `func1`:

- `func1` reads the return value from the call to `func2` from register `rv`, adds it to the value of `y` in register `s0`, and stores the result in `rv`
- `func1` restores the value of callee-saved register `s0` from its frame
- `func1` restores the return address from its frame back to `r15`
- `func1` pops its frame
- `func1` returns execution to the return address in `main`

Stack	
sp:	nothing on stack

#### Back inside `main`

- `main` writes the value in return value register `rv` (`r0`) to global variable `A`
- The program quits execution

### 3.4.4.4 albaCore Program

The complete albaCore program using the modern calling convention involving both registers and the stack is given below:

Addr	Label	Comment	Pseudocode	Assembly	Hex
0	main:	initialize sp	sp = 255	ldi r14, 255	7EFF
1		pass 3 as parameter in a0	a0 = 3	ldi r1, 3	7103
2		pass 4 as parameter in a1	a1 = 4	ldi r2, 4	7204
3		call func1	call func1	jal 7	D007
4		get address of A	t0 = &A	ldi r5, 64	7540
5		store rv in A	M[t0] = rv	st r0, r5, 0	9005
6		quit	quit	quit	F000

7	func1:	push frame with space for rv and s0	t0 = 2	ldi r5, 2	7502
8			sp = sp-t0	sub sp, sp, r5	1EE5
9		save ra in frame	sp[1] = ra	st r15, r14, 1	91FE
10		save s0 in frame	sp[0] = s0	st r10, r14, 0	90AE
11		copy a1 to s0	s0 = a1	and r10, r2, r2	2A22
12		call func2	call func2	jal 19	D013
13		set rv to rv from func2 + s0	rv = rv + s0	add r0, r0, r10	000A
14		restore s0	s0 = sp[0]	ld r10, r14, 0	8A0E
15		restore ra	ra = sp[1]	ld r15, r14, 1	8F1E
16		pop frame	t0 = 2	ldi r5, 2	7502
17			sp = sp + t0	add r14, r14, r5	0EE5
18		return to caller	return	jr r15	E0F0
19	func2:	set rv = a0 + a0	rv = a0 + a0	add r0, r1, r1	0011
20		return to caller	return	jr r15	E0F0

### 3.4.5 Recursive Summation

This example illustrates the use of the stack for recursion. There are 2 registers saved on the stack, the return address and the previous value of n.

I'll write more about what's going on in this example later!

```
int x;

main
{
    x = sum(5);
    exit;
}

int sum(n)
{
    if (n == 0)
        return 0;
    else
        return (n + sum(n - 1));
}
```

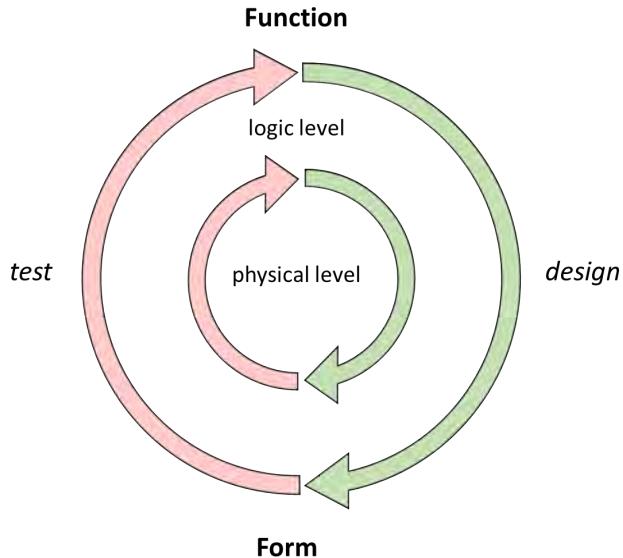
Variable	Definition	Register	Initial Value
n		r3	
prev_n		r4	
rtn_val		r0	
const_1		r1	
const_2		r2	
sp	stack pointer	r14	
rtn_addr		r15	
x	global ptr	r8	

Addr	Label	Pseudocode	Assembly	Hex
0		const_1 = 1	ldi r1, 1	7101
1		const_2 = 2	ldi r2, 2	7202
2	main:	sp = 255	ldi r14, 255	7EFF
3		n = N	ldi r3, 5	7305
4		call sum	jal 8	D008
5		x = 64	ldi r8, 64	7840
6		M[x] = rtn_val	st r0, r8, 0	9008
7		quit	quit	F000
8	sum:	sp = sp - 2	sub r14, r14, r2	1EE2
9		M[sp] = rtn_addr	st r15, r14, 0	90FE
10		M[sp+1] = prev_n	st r4, r14, 1	914E
11		if (n == 0) goto rtn_0	bz r3, rtn_0	B093
12		prev_n = n	and r4, r3, r3	2433
13		n = n - 1	sub r3, r3, r1	1331
14		call sum	jal sum	D008
15		rtn_val = rtn_val + prev_n	add r0, r0, r4	0004
16		prev_n = M[sp+1]	ld r4, r14, 1	841E
17		rtn_addr = M[sp]	ld r15, r14, 0	8F0E
18		sp = sp + 2	add r14, r14, r2	0EE2
19		return	jr r15	E0F0
20	rtn_0:	rtn_val = 0	ldi r0, 0	7000
21		return	jr r15	E0F0

## 4 Logic and Processor Design Technology

### 4.1 Form and Function

Digital logic design is the process of coming up with the *form* of a computing machine that performs a certain logic *function*. The design process is iterative. It starts with an idea for the desired function. Then, the designer *designs* a possible form. Given this initial implementation, the designer *tests* the function to see if it matches what is desired, and if not, changes the form, repeating as necessary.



Digital system design proceeds through two distinct phases or levels: first the logic level and then the physical level. Design at the logic level is about developing an essentially mathematical form for the system using abstract logical operations and components. For example, at the logic level, designing an adder might involve coming up with an arrangement of AND, OR, and inversion operations that correctly performs binary addition. Design at the physical level is about mapping the logical design to a particular implementation technology and making sure that it meets physical constraints on area, speed, and power.

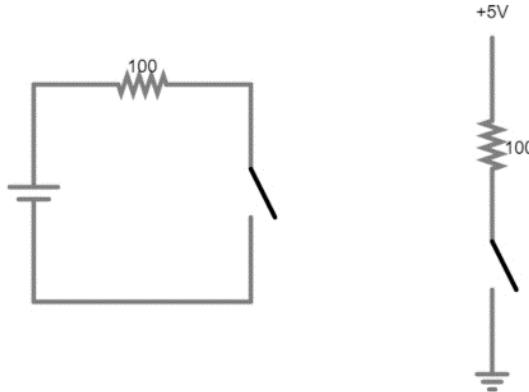
## 4.2 Using Switches to Build Logic Functions

The mathematics of logic design is based on the algebra of symbolic logic developed by George Boole in the 1840s and 1850s—Boolean algebra—but there is no evidence that Boole ever conceived of building computing machines based upon his system. The first electrical calculating machines in the 1880s with Herman Hollerith’s invention of the punched card tabulator, which was used to process the 1890 census. Hollerith’s invention used switches to detect holes in the cards, which activated electromagnets that rotated mechanical counters. Hollerith’s technology became the basis for founding the Computing-Tabulating-Recording Company in 1911, later renamed as International Business Machines (IBM).

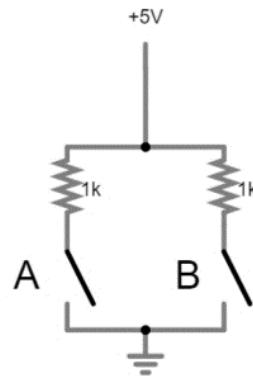
Welsh physicist C.E. Wynn-Williams first published the idea of using an electronic switch for binary arithmetic in 1931. Between 1934-36, NEC engineer Akira Nakashima published a series of papers showing that series and parallel switching circuits could implement logical AND and OR operations and further developed an algebra using the symbols for multiplication and addition to represent these symbols, unaware that Boole had developed such an algebra nearly 50 years earlier. Finally, in 1937, 21-year-old MIT graduate student Claude Shannon finally made the connection between Boolean algebra, binary arithmetic, and electrical switching circuits. Shannon’s master’s thesis, *A Symbolic Analysis of Relay and Switching Circuits* showed how switching circuits could implement all functions described by Boolean algebra, and the modern age of digital system design began.

The most basic approach to implementing Boolean logic functions with switches is using combinations of series and parallel switches to control the flow of current around a loop. Below is a basic circuit with a battery, a resistor, and a switch. When the switch is closed, current flows around the loop. The figure on the left explicitly shows the battery; the figure on the right, which is the more common notation for drawing schematics for digital circuits, shows the positive end of the battery as a supply voltage and the negative end as a ground symbol, but doesn’t explicitly show the battery itself. When the switch is closed, current flows from the supply voltage to ground. If we let the variable A represent the statement,

“the switch is closed” and variable Y represent the statement, “current is flowing,” then this circuit implements the function  $Y = A$ .



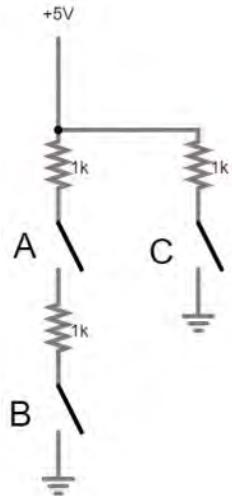
The circuit below has 2 switches wired in parallel. If we let variable A represent the statement, “switch A is closed,” variable B represent the statement, “switch B is closed,” and variable Y represent the statement, “current is flowing,” then this circuit implements the function  $Y = A \text{ or } B$ .



The circuit below has 2 switches wired in series. If we let variable A represent the statement, “switch A is closed,” variable B represent the statement, “switch B is closed,” and variable Y represent the statement, “current is flowing,” then this circuit implements the function  $Y = A \text{ and } B$ .

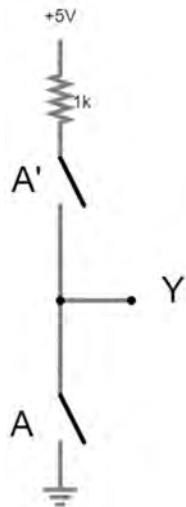


We can combine series and parallel connections to implement more complex Boolean functions. The following circuit implements the function  $Y = (A \text{ and } B) \text{ or } C$ .



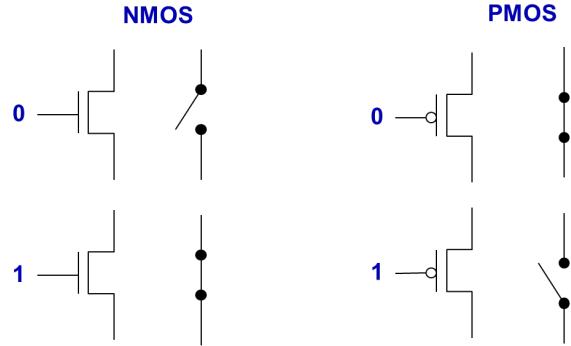
### 4.3 CMOS Logic

Previously, we used the presence or absence of current flowing through a circuit to indicate whether a logic statement was true or false. The problem with this approach is that as long as the statement is true, current *is* flowing through the circuit, consuming power from the supply. A better approach that consumes less power is to use switches to connect the output of a circuit to a high or low voltage, without actually completing a loop. The following circuit illustrates this concept.

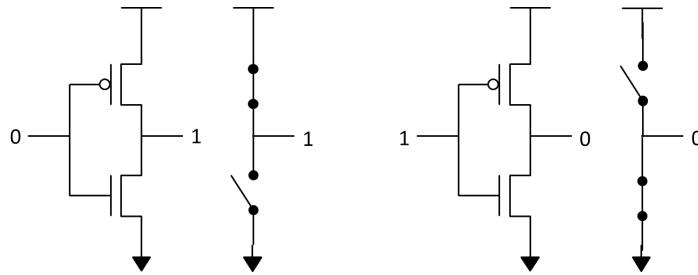


The circuit has 2 complementary switches controlled by variables  $A$  and  $A'$ , such that either one switch is closed or the other, but never both at the same time. The output  $Y$  is connected at the point between the switches. If  $A$  is true, then switch  $A$  is closed and switch  $A'$  is open, connecting  $Y$  to ground. If  $A$  is false, then switch  $A'$  is closed and switch  $A$  is open, connecting  $Y$  to the positive supply voltage. If a high output implies “true” and a low output implies “false,” then this circuit implements an inverter,  $Y = A'$ .

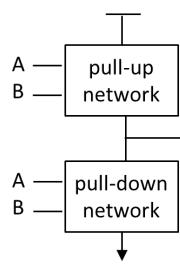
CMOS circuits use complementary transistors as voltage controlled switches to implement Boolean functions. CMOS stands for complementary metal-oxide-semiconductor, which is the technology used to fabricate the transistors. There are 2 kinds of CMOS transistors. NMOS transistors are switches that are closed when there is a high voltage on the input (called the “gate”) and open when there is a low voltage. PMOS transistors are the opposite, closed when there is a low voltage on the gate and open when there is a high voltage.



Below is a CMOS inverter

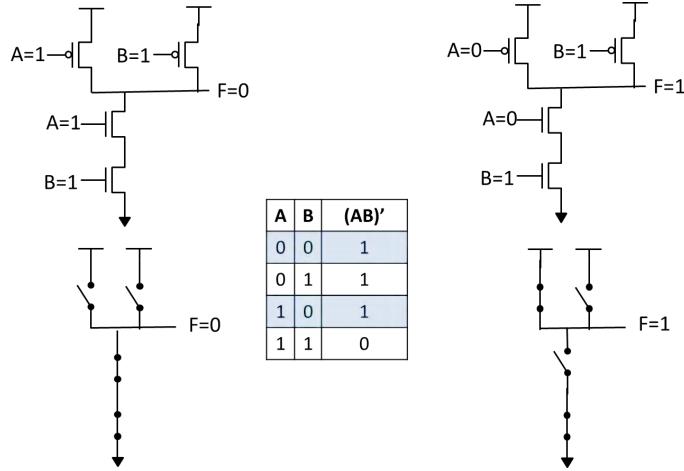


In general, we build CMOS circuits by using PMOS transistors to implement the logic for the conducting path to the positive supply voltage—called the “pull-up network”—and NMOS transistors to implement the logic for the conducting path to ground—called the “pull-down network.” We never use NMOS devices in the pull-up network or PMOS devices in the pull-down network, because for electronic reasons beyond the scope of this discussion, NMOS devices make poor paths to the positive supply and PMOS devices make poor paths to ground.

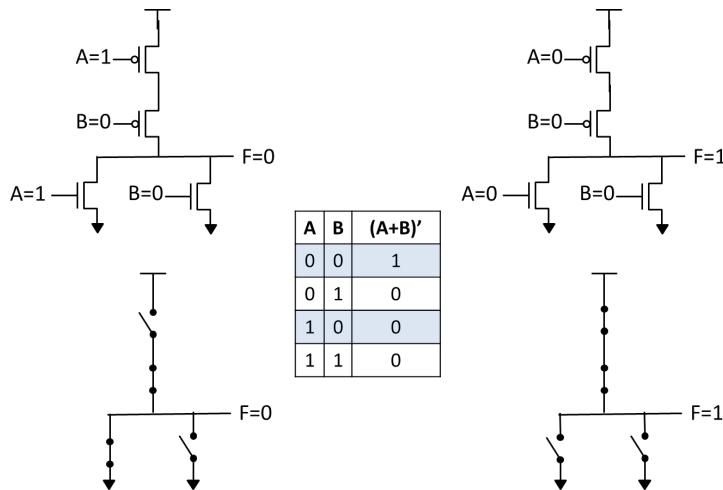


- Pull-up network to supply voltage V<sub>cc</sub>
  - PMOS only
- Pull-down network to ground
  - NMOS only
- **One is conducting while the other isn't**
- Pull-up network (PMOS) and pull-down network (NMOS) are logical complements
- CMOS: complementary metal-oxide-semiconductor

A NAND gate has NMOS devices in series in the pull-down network and PMOS devices in parallel in the pull up network.



A NOR gate has NMOS devices in parallel in the pull-down network and PMOS devices in series in the pull-up network.



We can explain the logic of the pull-up and pull-down networks using DeMorgan's Law. For a NAND gate, the pull-down network is conducting, making the output a logical 0, when A = 1 and B = 1.

$$F' = AB$$

Conversely, the pull-up network is conducting, making the output a logical 1, when A = 0 or B = 0.

$$F = A' + B'$$

These two statements are equivalent by DeMorgan's Law

$$(AB)' = A' + B'$$

Similarly, for a NOR gate, the pull-down network is conducting, making the output a logical 0, when A = 1 or B = 1.

$$F' = A+B$$

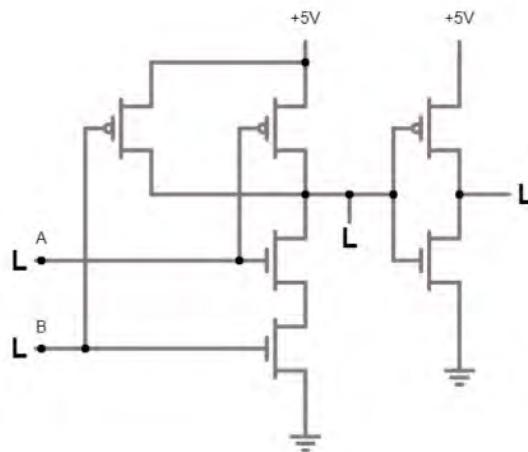
Conversely, the pull-up network is conducting, making the output a logical 1, when A = 0 and B = 0.

$$F = A'B'$$

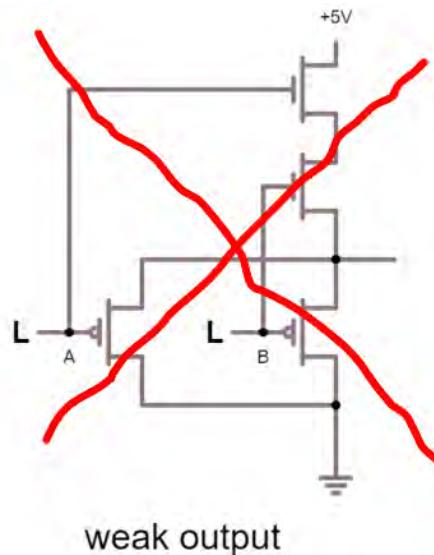
These two statements are equivalent by DeMorgan's Law

$$(A+B)' = A'B'$$

In order to implement an AND gate, place an inverter after the output of a NAND gate:



Don't be tempted to build an AND gate by putting NMOS devices in the pull-up network and PMOS devices in the pull-down network—remember that we said that NMOS devices make poor paths to the positive supply and PMOS device make poor paths to ground.



Here's a general methodology for implementing a CMOS circuit for a Boolean logic function in several variables:

- 1. Start with equation in inverted form  $f = (g)'$**

If equation is already in inverted form, you're fine  
If not, you'll need to add an inverter

Don't worry about whether inputs are inverted or not

Example:

$$f = (a' b + c)'$$

already in inverted form,  $g = a' b + c$

$$f = a' b + c$$

not in inverted form, so we'll let  $g = a' b + c$  then add inverter later

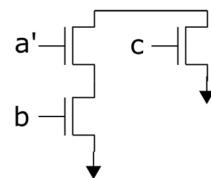
## 2. Design the pull-down network

Function  $g$  represents the logic when the output is 0, that is, when the pull-down network should be conducting

Implement pull-down network for  $g$  with NMOS transistors

AND: series connections

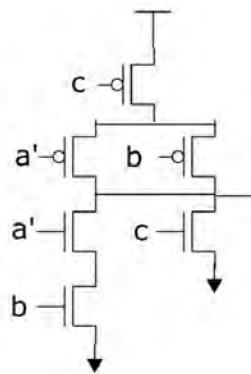
OR: parallel connections



## 3. Implement pull-up network as complement of pull-down network

PMOS transistors

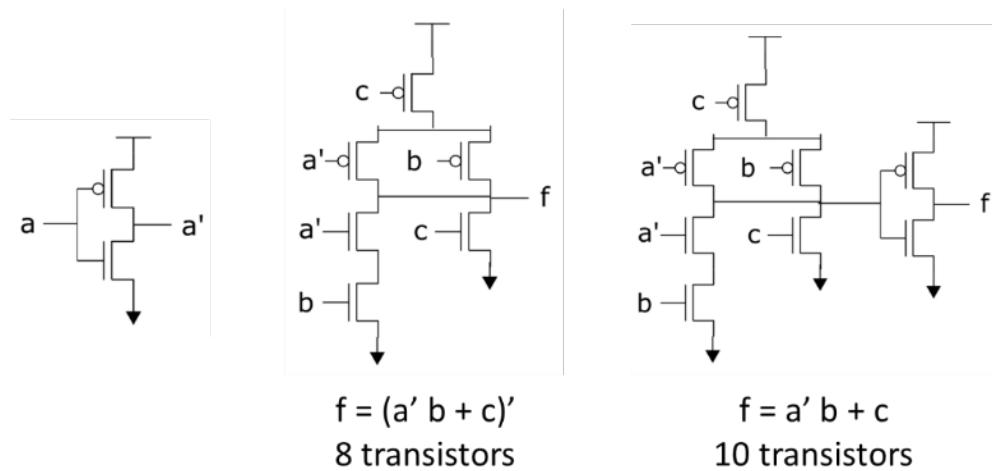
exchange serial and parallel connections



## 4. Add inverters as necessary

Add an inverter to get  $a'$  from  $a$

If  $f$  was not naturally inverting, add an inverter on the output



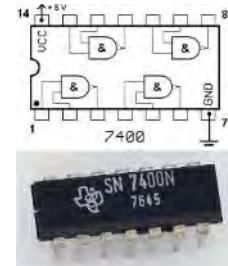
## 4.4 Evolution of Digital Circuit Technology

### 4.4.1 Discrete Transistors

Switch technology for digital logic systems evolved rapidly between the 1930s and 1950s from electromechanical relays, to vacuum tubes, to transistors. William Shockley, John Bardeen, and Walter Brattain demonstrated the first working transistor at AT&T Bell Laboratories in 1947. “Discrete” transistors—single transistors in a metal or plastic package with 3 wires—began appearing in commercial products shortly thereafter. By the mid-1950, researchers began working on technology for integrating more than one transistor onto a single chip. Jack Kilby of Texas Instruments demonstrated the first working integrated circuit in 1958. A few months later, Robert Noyce of Fairchild Semiconductor improved upon Kilby’s design, producing the first silicon integrated circuit using photolithographic masks to pattern aluminum interconnections between transistors.

### 4.4.2 Small Scale Integrated Circuits and the 7400 Series Logic Chips

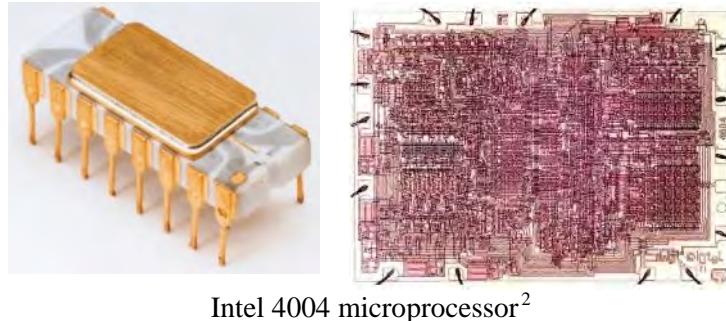
In 1964, Texas Instruments launched the 7400 series of logic chips. These were small-scale integrated circuits (SSI) with a few basic logic gates or simple digital building blocks. The first product in the family was the 7400 quad NAND gate, with 4 2-input NAND gates in a 14-pin package.<sup>1</sup> Other chips in the series include the other basic logic gates, multiplexors, adders, simple storage elements (flip-flops), and counters. The 7400 series chips were used to build many of the computer systems of the 1960s and 1970s and are still in use today.



### 4.4.3 Microprocessors and VLSI

In 1968, Noyce, along with Gordon Moore and Andy Grove, left Fairchild to form Intel Corporation. Intel’s first commercial products were memory chips, which dominated their sales volume into the 1980s. In 1971, Intel produced the first commercially available single-chip computer processor integrated circuit, the 4004 microprocessor.

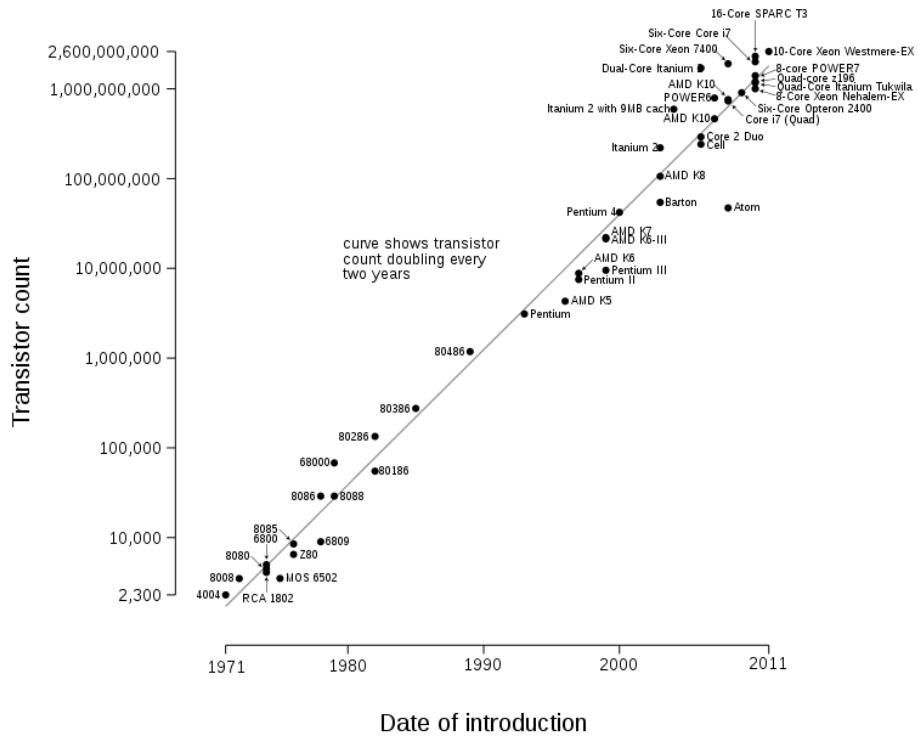
<sup>1</sup> 7400 image from <https://upload.wikimedia.org/wikipedia/commons/2/26/7400.jpg>



Intel 4004 microprocessor<sup>2</sup>

The Intel 4004 contained 2,250 transistors and ran at a frequency of 500 kHz. Since then, both the number of transistors and speed of digital integrated circuits has grown exponentially over time. This rate of growth is known as Moore's Law, after Intel founder Gordon Moore, who first predicted in 1965 that the number of components in an integrated circuit would double every year.<sup>3</sup>

### Microprocessor Transistor Counts 1971-2011 & Moore's Law



Transistor count and speed are important factors in digital design technology, but so is cost. Very-large-scale integrated circuits (VLSI) have dramatically reduced the cost per transistor for high-volume chips. Today, microprocessors with a few hundred million transistors can be purchased retail for a few hundred dollars, or around 1 million transistors per dollar. Compare that with the cost today of a single 7400 quad NAND gate available today for around \$0.50 or a single discrete transistor for around \$0.40. But while VLSI chips can make very complex digital systems very inexpensive when produced in very large quantities, the cost of bringing the *first chip* to market is extremely expensive. While prototyping services exist that make it possible to fabricate a chip with a million or so transistors for a few hundred

<sup>2</sup> <https://en.wikichip.org/wiki/intel/mcs-4/4004>

<sup>3</sup> [https://en.wikipedia.org/wiki/Moore%27s\\_law](https://en.wikipedia.org/wiki/Moore%27s_law)

dollars, it can cost \$10 million or more to design and fabricate an application-specific integrated circuit (ASIC) that's ready for market. And that's not to mention the risk involved with the chip having bugs!

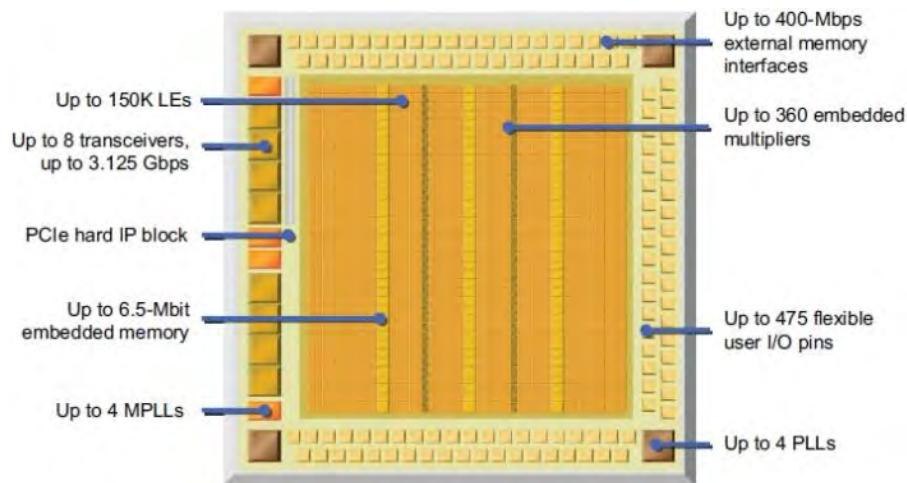
#### 4.4.4 Programmable Logic Devices

Programmable logic devices are packaged chips that take advantage of most of the transistor density and performance of integrated circuits, but without the cost and risk of producing a custom chip for each design. As the name implies, programmable logic devices can be “programmed” or configured to implement different logic functions in hardware. There is an overhead to making programmable logic devices configurable—it may take 10-100 times as many transistors to implement a design using programmable logic versus a custom integrated circuit, but with that come the advantages of being able to reprogram the device with a new design, not to mention avoiding the cost of designing and fabricating a custom integrated circuit.

Altera was founded in 1983 and produced the industry’s first programmable logic device in 1984.<sup>4</sup> That first device, the EP300, could be programmed by writing to erasable memory cells—the forerunner to today’s flash memory—to configure different sum-of-products logic expressions.<sup>5</sup> With the EP300, logic designers could replace a bunch 7400 series logic chips with a single device.

Xilinx introduced the industry’s first field-programmable gate array (FPGA) in 1985. Altera introduced their first FPGA in 1992. As opposed to the earlier generation of programmable logic devices that could only implement basic combinational logic expressions, FPGAs were developed for implementing complete digital systems. Field-programmable gate arrays (FPGAs) are composed of arrays of configurable logic elements (LEs) along with a configurable interconnection network. Many FPGAs also contain blocks of memory and special-purpose hardwired modules such as multipliers and even complete microprocessors that would be inefficient (though possible) to implement with configurable logic elements.

The floorplan of the Altera Cyclone IV E family of FPGAs is shown below.



The Cyclone IV E FPGA on the DE2-115 board, the EP4CE115F29C7N, is one of the largest FPGAs in the family, with:

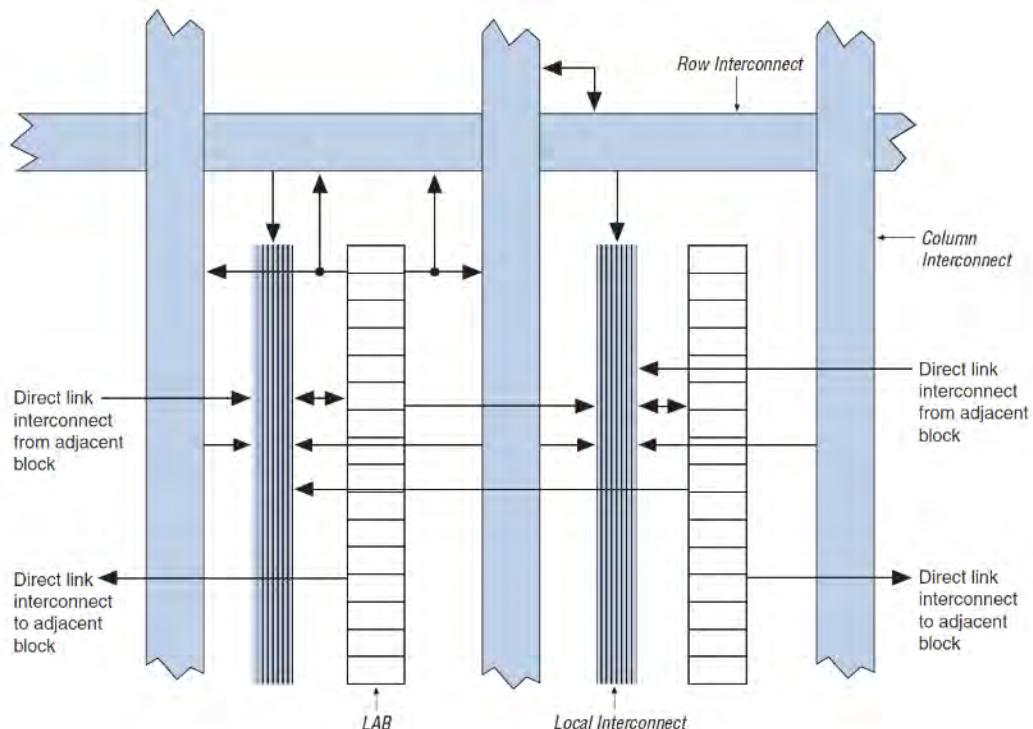
- 114K logic elements (LEs)

<sup>4</sup> [https://en.wikipedia.org/wiki/Field-programmable\\_gate\\_array](https://en.wikipedia.org/wiki/Field-programmable_gate_array)

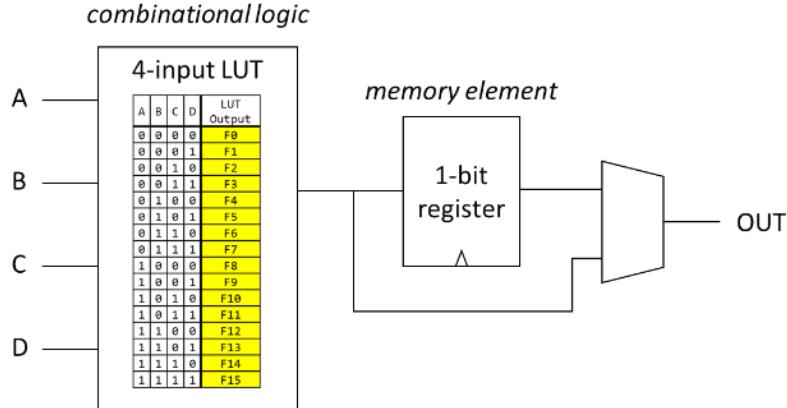
<sup>5</sup> [https://www.altera.com/solutions/technology/system-design/articles/\\_2013/in-the-beginning.html](https://www.altera.com/solutions/technology/system-design/articles/_2013/in-the-beginning.html)

- 3.9 Mbits of embedded memory,
- 266 embedded multipliers
- 528 user I/O pins

The figure below shows the Cyclone IV logic structure, which consists of arrays of logic elements called logic array blocks (LAB), banks of programmable local interconnect for making connections between LEs in the same LAB, and row and column global interconnect for connecting LEs across the chip:



Each logic element (LE) contains the basic building blocks for both combinational and sequential logic circuits. The figure below shows a simplified view of the insides of a logic element (LE). The heart of a LE is a configurable 4-input look-up table (LUT). A LUT is basically a hardware implementation of a 16-entry truth table that can be configured to behave as any 4-input (or less) combinational logic element, depending on the values loaded into the table by the user. A 1-bit register connected to the LUT output serves as a basic memory element for building sequential systems. The LE output can be configured to come from either the LUT or the register. The LE inputs and output are connected to the local and global interconnect network via configurable switches for building large systems from these elements.



## 4.5 Evolution of Digital Logic Design Methodology

As described early, the digital system design process has two distinct phases or levels:

- **logic level:** designing an essentially mathematical representation of a system using abstract logical components such as basic logic gates or memory elements
- **physical level:** implementing the logic level design with a particular technology, ranging from individual transistors to FPGAs

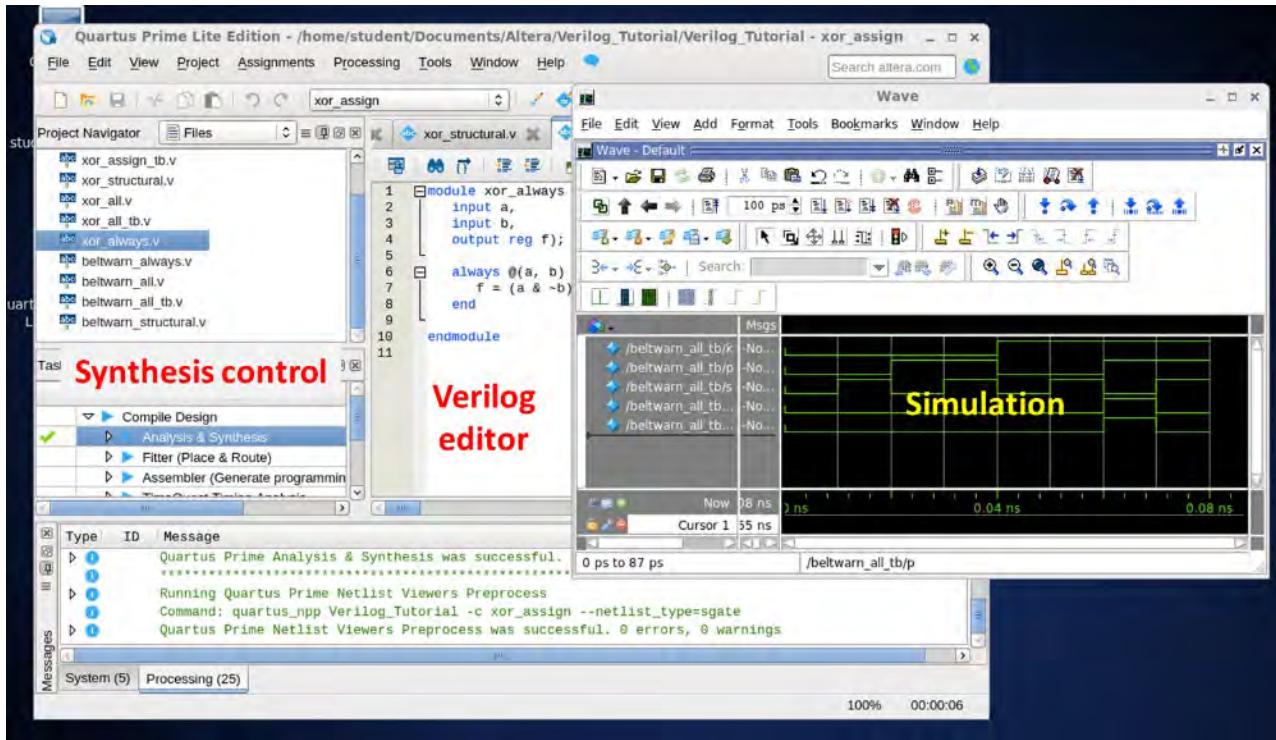
Before 1980, much of digital design at both levels was done by hand. Logic designers would work out the logic equations for each component of the system, then optimize the equations by applying the laws of Boolean algebra or using Karnaugh maps. At the physical level, IC designers would manually figure out where to manually place transistors on chips and how to optimally connect them to get the densest designs with the shortest wires.

The beginning of the 1980s saw the growth of computer-aided design (CAD) tools for electronic design automation (EDA). This included logic design tools that could automatically minimize logic equations and physical design tools that could automatically place and route physical components. One of the biggest advances was the introduction of *hardware description languages* (HDLs). The U.S. Department of Defense began funding the development of VHDL in 1981. Verilog was released by Gateway Design Automation in 1986.

Hardware description languages such as Verilog and VHDL provide a means for describing the design of a digital system at the logic level. They serve a dual purpose in the hardware design flow:

- as input to a *simulator* for testing the function of the logical design
- as input to a *synthesis tool* that can automatically map the description into a particular implementation technology

The Altera Quartus package brings together a fully-integrated suite of HDL-driven CAD tools for FPGA design:



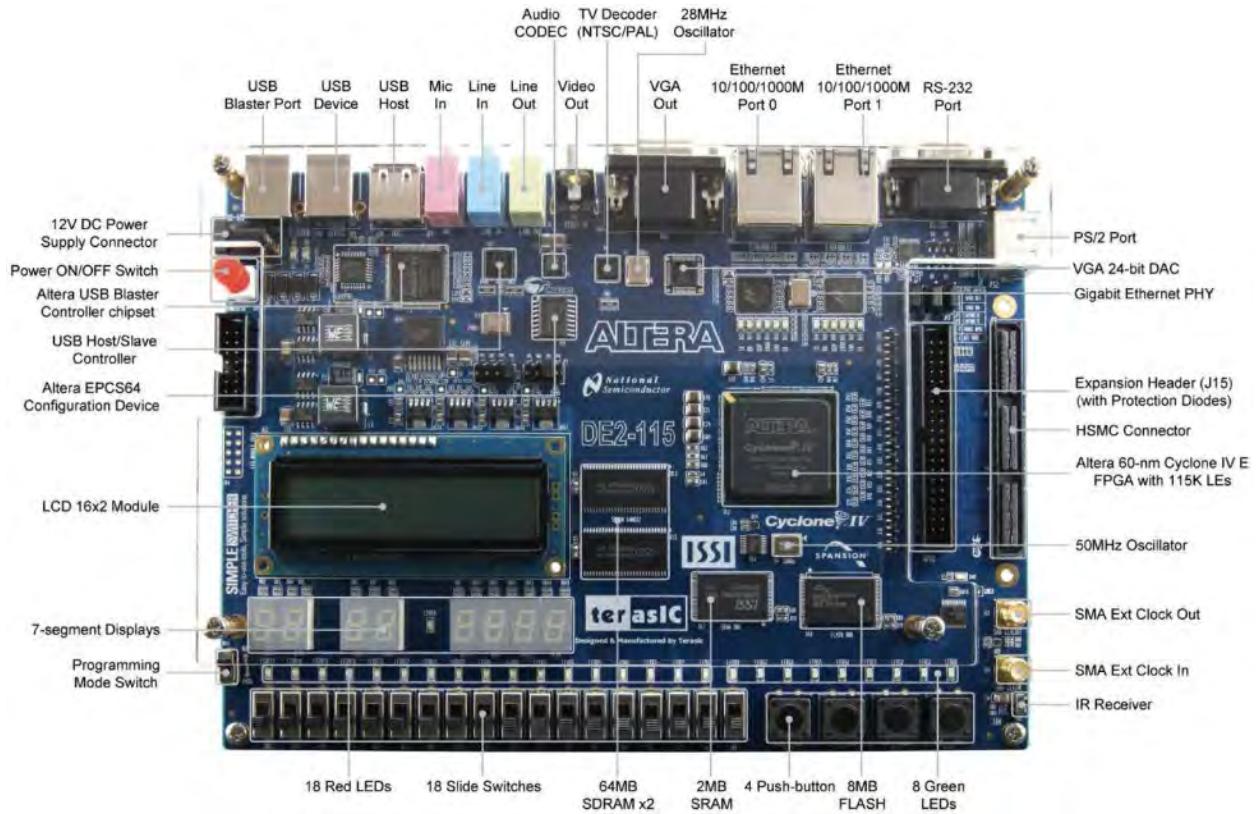
The combination of HDLs and FPGAs has dramatically changed the face of digital system design. The high productivity of this platform makes it possible for even novice designers to implement systems that are far more complex than was possible before.

From an educational perspective, however, there is a pitfall to avoid: between HDLs and FPGAs, it's easy for beginners to lose sight of the fact that they are actually designing hardware, and not writing software. It may be tempting to approach a hardware application such as a video game console as if it were a programming assignment, but ***don't do it!*** Control and data structures that work well in software oftentimes don't make sense in hardware. There is a discipline to hardware design that differs from software design, and it is important to maintain that discipline even when designing logic using a hardware description language. We repeat:

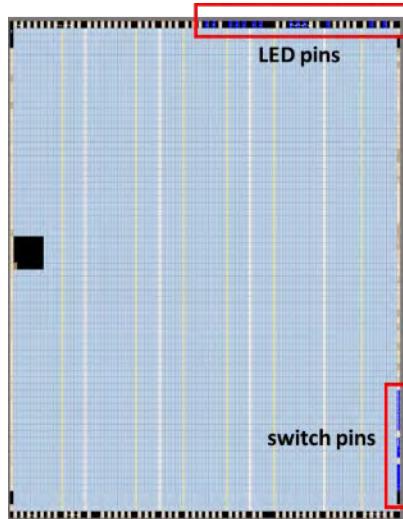
***Verilog is a hardware description language (HDL), not a programming language.*** Verilog has language features that are unique to describing the structure and behavior of hardware that don't have counterparts in a programming language like C. Even though some Verilog statements may *look* like C statements, they may do very different things, or worse, do similar things with subtle differences.

## 4.6 Terasic Altera DE2-115 Board

The Terasic DE2-115 board provides an assortment of input/output devices connected to an Altera EP4CE115F29C7N FPGA. The board is pictured below:

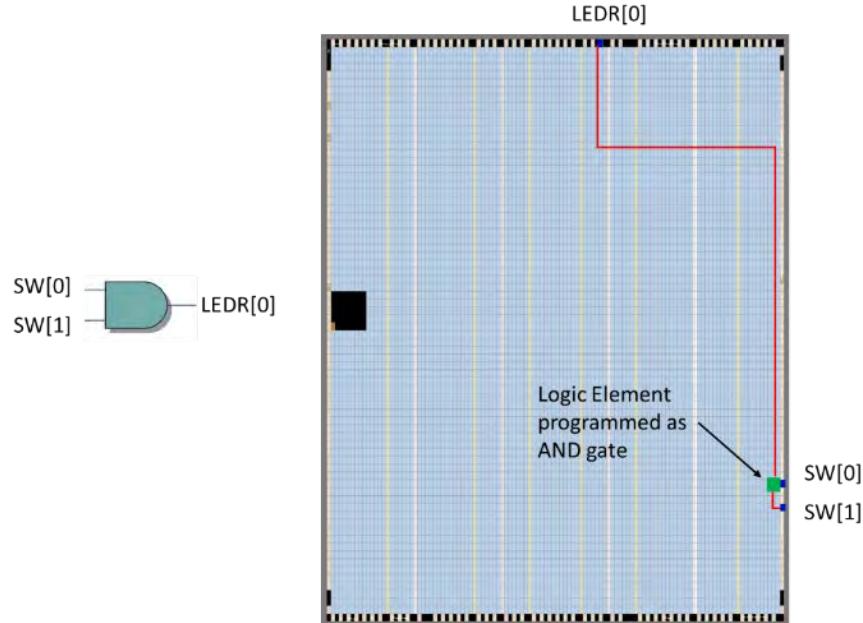


Wiring traces on the DE2-115 board connect specific pins on the FPGA to specific I/O devices. For example, as illustrated below, the LEDs on the board are connected to pins on the top edge of the FPGA, while the switches are connected to pins on the right side.

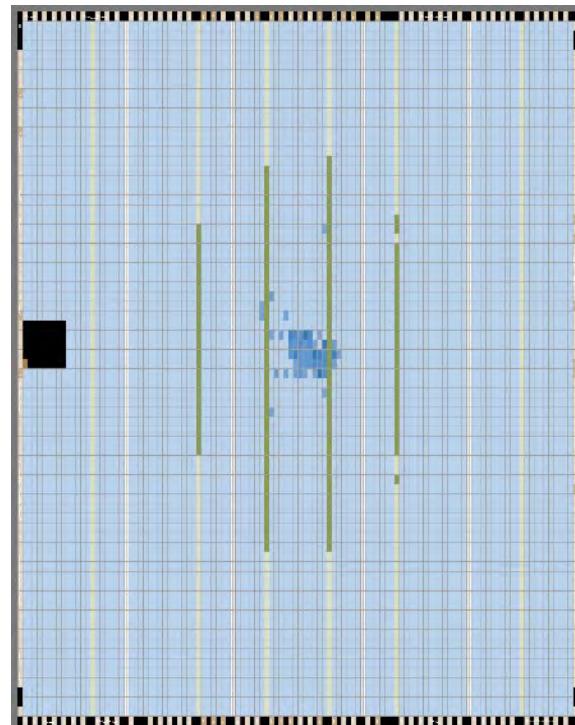


When compiling a design for the DE2-115 board, it is critical to let the Altera Quartus compiler know which ports in the Verilog model correspond to which pins in the FPGA—otherwise the compiler will randomly assign pins to the ports. A *pin assignment file* imported into the design contains the correct port-pin mappings for the board. Thus a design for the DE2-115 consisting of a 2-input AND gate with

inputs connected to switches SW[0] and SW[1] and output connected to red LED LEDR[0] would be placed and routed into the FPGA as follows:



Note that this design uses an insignificant portion of the total resources on the FPGA: only 1/114,480 logic elements and only I/O 3/529 pins. By comparison, the layout of the complete 16-bit albaCore microprocessor detailed later in these notes) uses 564/114,480 logic elements—still less than 1%—along with 128 KB of memory (25% of the block memory available) and 51 I/O pins



# 5 Getting Started with Verilog

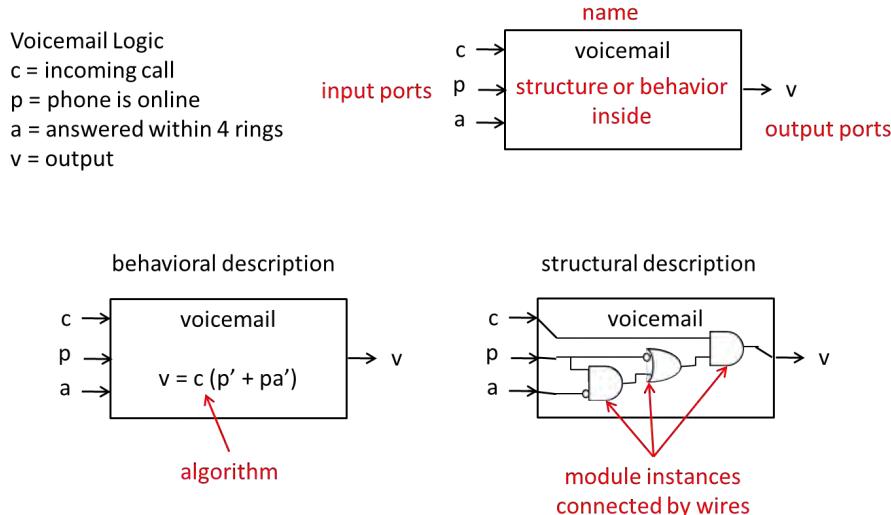
## 5.1 Hardware Description vs. Programming Languages

**Verilog is a hardware description language (HDL), not a programming language.** Verilog has language features that are unique to describing the structure and behavior of hardware that don't have counterparts in a programming language like C. Even though some Verilog statements may *look* like C statements, they may do very different things, or worse, do similar things with subtle differences.

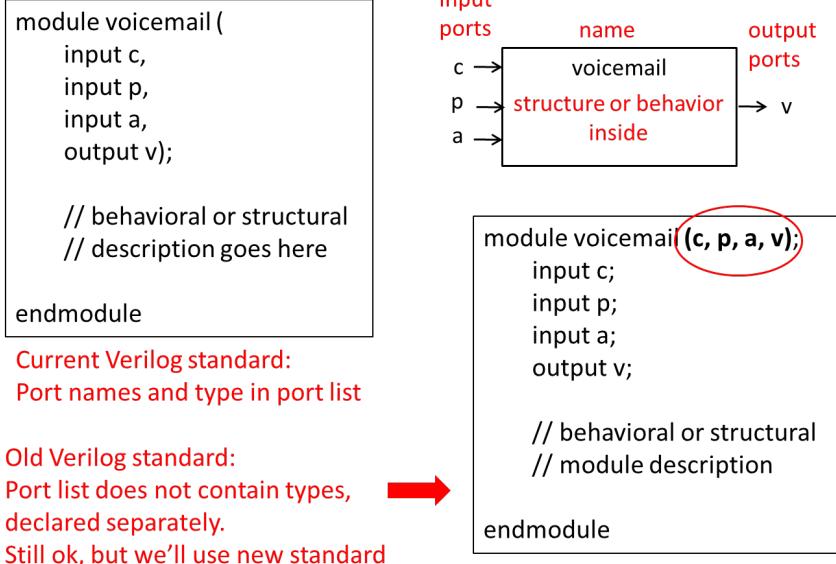
Our advice: make a clean break from thinking like a programmer and start thinking like a hardware designer. Don't ever say, "I'm writing a Verilog program." Instead say, "I'm developing a Verilog model of a hardware system."

## 5.2 Modules

Modules are the basic building blocks of hardware system descriptions in Verilog. Modules have an *interface* that consists of the ports for connecting one module to other modules and an interior or *body* that describes what the module does. The body can either be a *behavioral* description, which specifies algorithmically what the module does, or a *structural* description, which is a circuit consisting of interconnections of instances of other modules.



A module interface specifies the name of the module and the list of ports. A module declaration begins with the keyword `module` and ends with the keyword `endmodule`. Module ports are the inputs and outputs of the module. Ports in the port list are specified by stating the direction (input, output, or inout), its width (ports can be multiple bits wide but for now we'll just consider single bits), and its name. Two ways of specifying the port list are both in common usage.



The current Verilog standard (Verilog-2001) declares the direction, width, and name of each port all together in a comma-separated list enclosed in parentheses, much like the way that parameter types and names are listed in a C or C++ function. An older standard (Verilog-1995) just lists the names of the ports in the list, but then specifies the width and direction outside of the list (much as an older C standard declared parameters). For this class, we will use the newer standard of declaring the directions and widths with the name, but you will still encounter the older standard often.

### 5.3 Expressing Behavior Using the assign Statement

Verilog offers several ways of expressing the behavior of circuits. The simplest form of behavioral description is the **assign** statement. An assign statement is like using a soldering iron on logic operators to create a circuit—it permanently connects logic operators to ports and internal wires.

Verilog expressions use the following logic operators (it also has arithmetic, relational, and bit manipulation operators that we will introduce later):

and	&
or	
xor	^
not	~

As a first example, consider a Verilog model of a NAND gate using an **assign** statement.

```

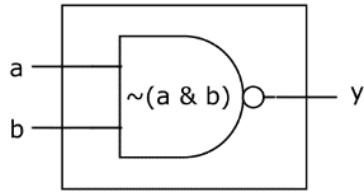
module nand_simple (
    input    a,
    input    b,
    output   y
);

    assign y = ~(a & b);

endmodule

```

Here, the `assign` statement “solders” the circuit expressed by  $\sim(a \& b)$  to the output port `y`.



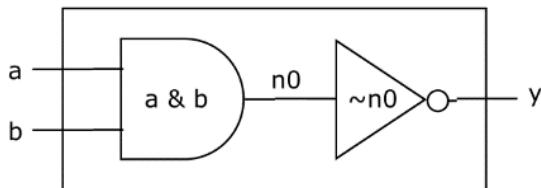
We can use multiple assign statements to represent separate subcircuits. The following example shows an alternative way of modeling a NAND gate, using two subcircuits connected by a wire.

```
module nand_wired (
    input    a,
    input    b,
    output   y
);

    wire n0;

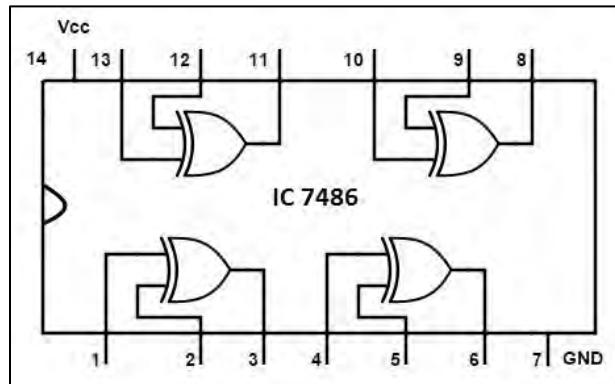
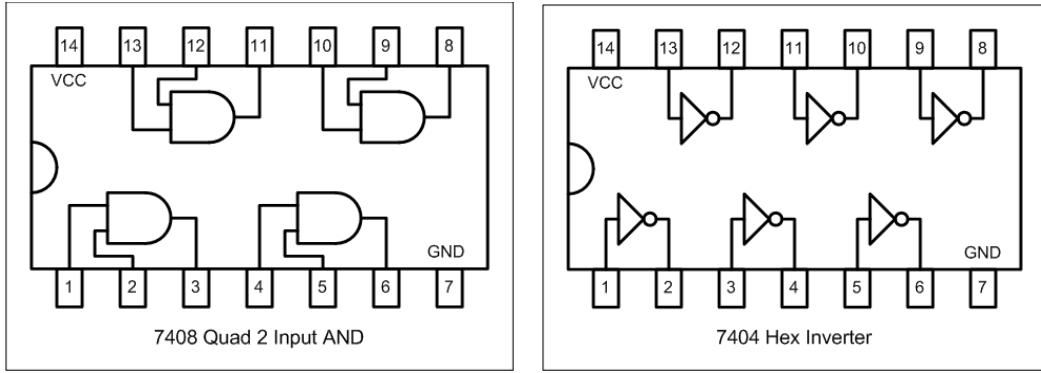
    assign y = ~n0;
    assign n0 = a & b;

endmodule
```

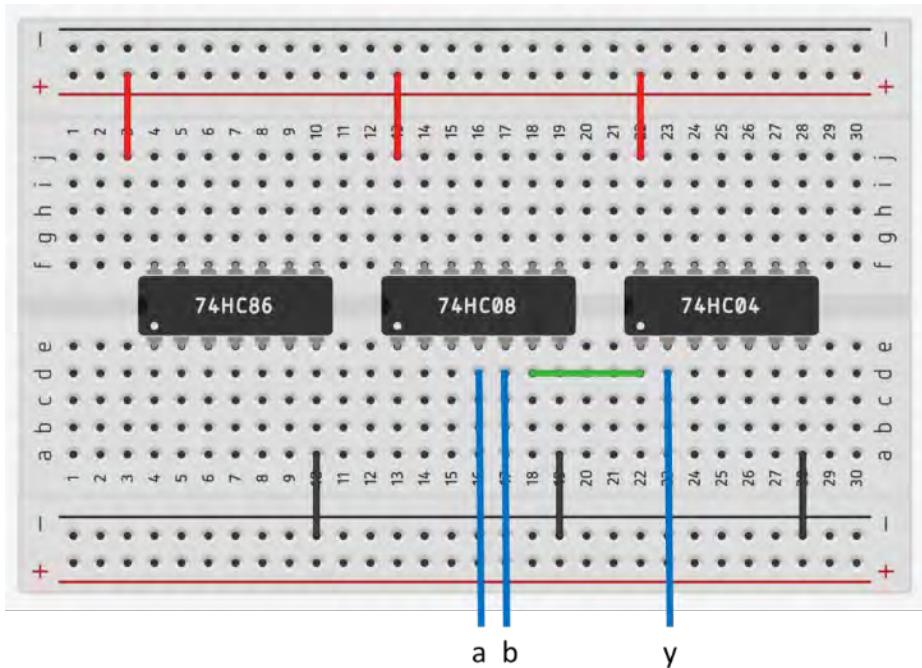


The Verilog data type `wire` models a physical wire in a circuit. Module ports are also implicitly of type `wire`. It is important to remember that ***Verilog is a hardware description language and not a programming language***. The `wire` data type may look like a variable in a programming language like C but it is not the same thing. (Note that Verilog does have data types that behave more like variables in programming languages that we will consider later, but `wire` is not one of these). To better appreciate the differences between Verilog `wires` and `assign` statements and C variables and assignments statements, let's look at some examples with real hardware and how to model them.

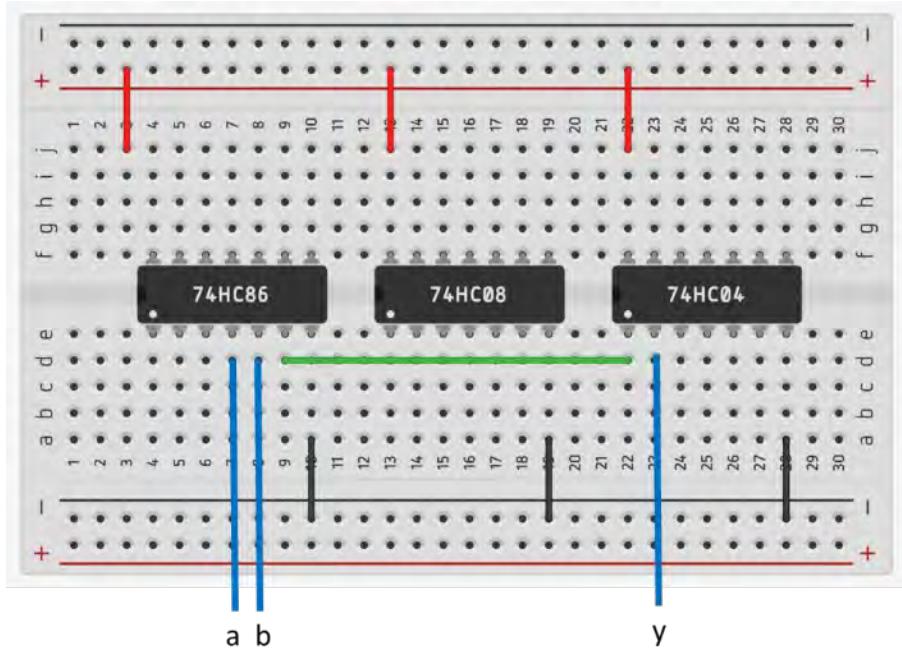
The 7400 series of small-scale and medium-scale integrated circuits (SSI and MSI circuits, as opposed to today’s very-large-scale VLSI circuits) was the main technology for implementing digital systems in the 1960s and 1970s. Still available today, these chips contain a small handful of simple digital components, such as basic logic gates, multiplexors, 4-bit adders, flip-flops, and 4-bit counters. The pinouts for 3 typical 7400-series chips, the 7404 hex inverter, the 7408 quad AND gate, and the 7406 quad XOR gate are shown below:



The figure below shows an actual hardware implementation of the NAND circuit using the high-speed CMOS (HC) versions of the 7408 and 7404:



The red and black wires are the power and ground connections. The blue wires are the input and output ports. The green wire connects the output of one of the AND gates in the 7408 to the input of one of the inverters in the 7404. Note that while the green wire can carry different values (logical 1 or 0 represented by high or low voltages), we can't change the function carried by the wire—namely connecting the output of an AND gate to the input of an inverter—without ripping out the wire and reconnecting it to a different chip. For example, we could implement an XNOR gate by reconnecting the left side of the wire to the output of one of the XOR gates in the 7486:



In a C program, we can reassign the results of different expressions to a variable with successive assignment statements. For example:

```
int xnor(int a, int b)
{
    int n0, y;

    n0 = a & b;
    n0 = a ^ b;
    y = ~n0;
    return y;
}
```

Here, the results of the first assignment to `n0` is overwritten by the second assignment to `n0`. Unlike assigning variables in C, `assign` statements in Verilog make a permanent connection. For example, consider the following Verilog model:

```
module bad_2_drivers (
    input    a,
    input    b,
    output   y
);
```

```

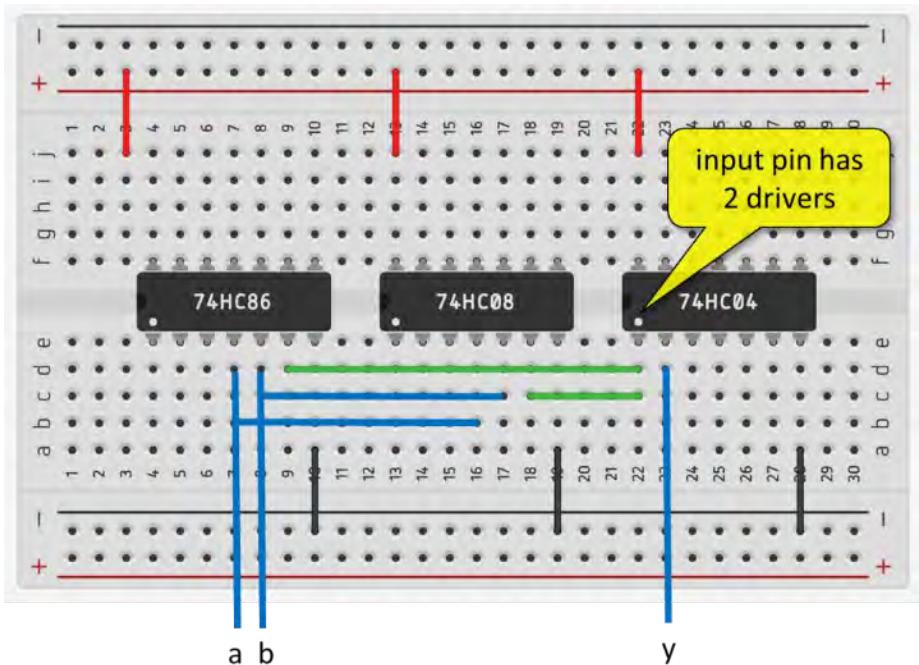
wire n0;

assign y = ~n0;
assign n0 = a & b;
assign n0 = a ^ b;

endmodule

```

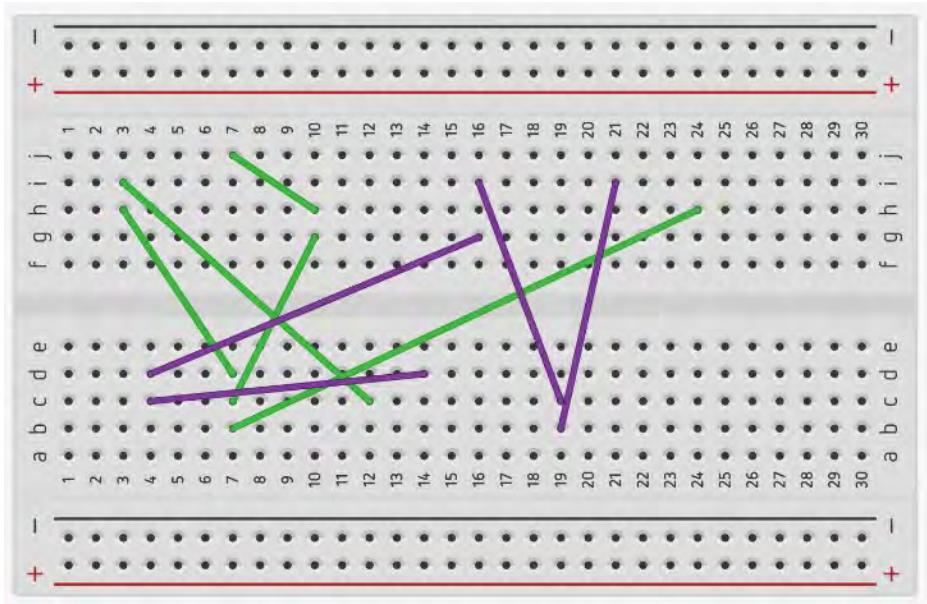
Rather than first assigning the AND expression to `wire n0` and then re-assigning the XOR expression to `n0`, this module describes a hardware configuration where the results of both the AND and the XOR are assigned to the same wire. Here's the situation in real hardware:



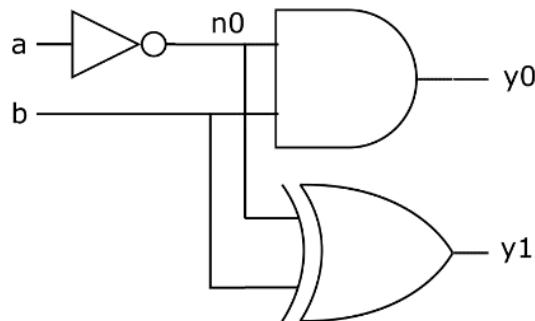
The two green wire segments in this circuit are really just part of one “wire” since they are connected together at pin 1 of the 7404 chip. Moreover, this is problematic since both the 7486 and the 7408 are trying to drive the same pin on the 7404. For example, if the 7486 is trying to output a low voltage (logical 0) and the 7408 is trying to output a high voltage (logical 1) onto the same wire, then the voltage on the wire will be somewhere in between and the logical sense is indeterminate. Verilog synthesis tools will generally flag this as an error, even though it is syntactically correct. Altera Quartus gives the following error when synthesizing the module `bad_2_drivers`:

```
Error (10028): Can't resolve multiple constant drivers for net "y" at bad_2_drivers.v(11)
Error (10029): Constant driver at bad_2_drivers.v(10)
```

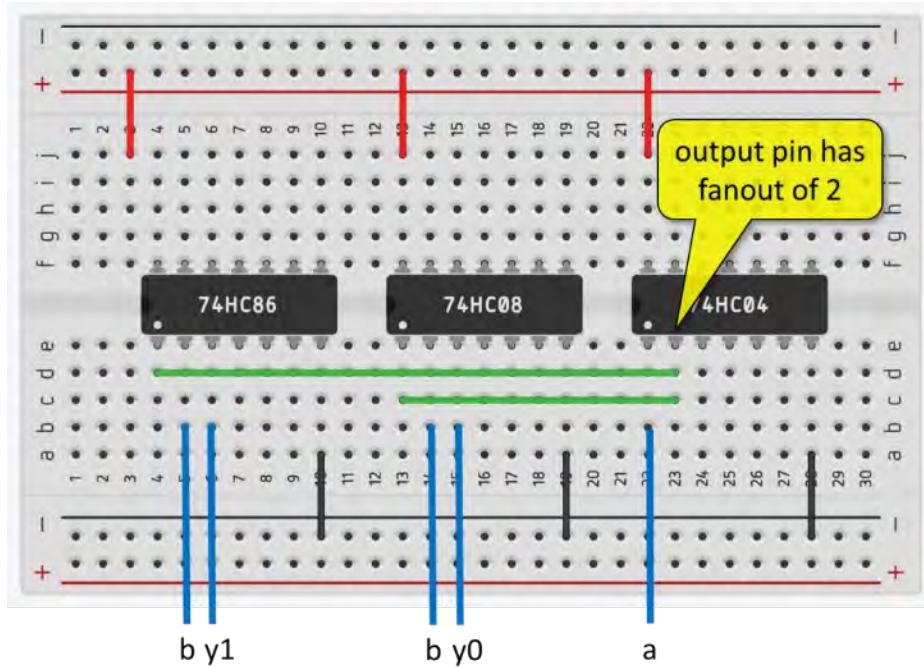
Many students, especially non-electrical engineers (I'm talking about you, CS majors) find this confusing. The Verilog `wire` data type is actually better thought of as a connection to a net in a circuit that is at a single electrical potential, rather than a “wire”. Even though the circuit below has lots of wire segments, there are only 2 electrically separate “wires” (green and purple).



While a single wire can't have multiple drivers (connected to the outputs of multiple logic gates), you can connect a common wire to the inputs of multiple logic gates. The number of different inputs connected to a single driver is called the *fan-out* of the driver. For example, the output of an inverter can drive an input of an AND gate and an input of an inverter over (segments of) the single wire  $n0$ . The inverter thus has a fan-out of 2 over wire  $n0$ :



An implementation with 7400-series chips would be as follows, where the inverter output on pin 2 of the 7404 is connected to both the input of an AND gate on pin 1 of the 7408 and the input of an XOR gate on pin 1 of the 7486:



A Verilog model for the circuit is:

```
module fanout (
    input    a,
    input    b,
    output   y0,
    output   y1
);

    wire n0;

    assign n0 = ~a;
    assign y0 = n0 & b;
    assign y1 = n0 ^ b;

endmodule
```

As a shortcut, Verilog lets you declare a wire and assign a value to it in a single statement, for example:

```
module nand_wired_shortcut (
    input    a,
    input    b,
    output   y
);

    wire    n0 = a & b; // declare wire and assign expression to it
    assign y  = ~n0;

endmodule
```

Note that this is not the same as initializing a variable, which implies that the value of the variable can be changed later. When you assign an expression to a wire, it represents a permanent physical connection to the wire.

A common Verilog syntax error is forgetting the `assign` keyword when connecting expressions to wires:

```
module bad_no_assign (
    input    a,
    input    b,
    output   y
);

    wire n0;

    y = ~n0;      // syntax error: missing assign
    n0 = a & b; // syntax error: missing assign

endmodule
```

Altera Quartus flags this with the following error message that doesn't quite tell you what your mistake really was:

```
Error (10170): Verilog HDL syntax error at bad_no_assign.v(9) near text: "=";
expecting ".", or "(".
```

A really nasty “feature” of Verilog syntax is that it will automatically declare a `wire` if you use a name in an expression for a `wire` that wasn’t previously declared. You should never take advantage of this feature! It typically crops up when you misspell a `wire` name in an expression and can lead to some really difficult bugs to track down unless you know how to spot them. For example, Verilog will not flag this as an error:

```
module bad_implicit_wire (
    input    a,
    input    b,
    output   y
);

    wire n0;

    assign y = ~n0;
    assign n0_misspelled = a & b; // meant n0 but misspelled it!!!

endmodule
```

This will connect `a & b` to a dangling wire named `n0_misspelled` which isn’t what we meant. Altera Quartus will give a warning, but not an error:

```
Warning (10236): Verilog HDL Implicit Net warning at bad_implicit_wire.v(10):
created implicit net for "n0_misspelled"
```

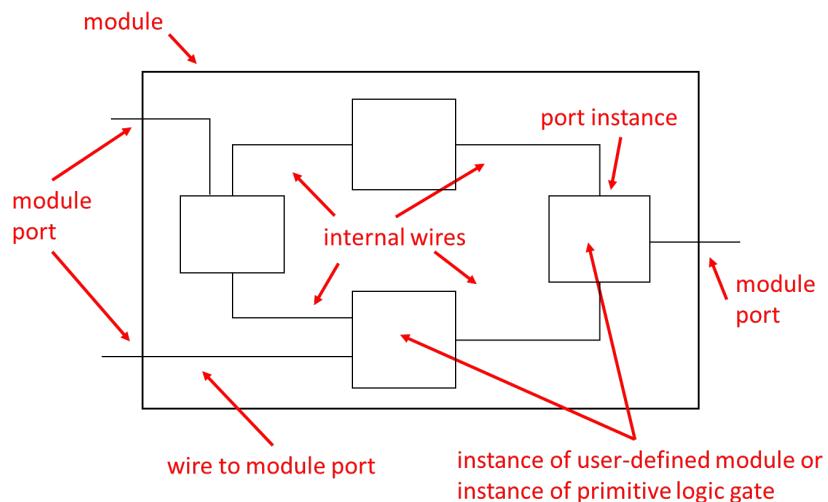
Fortunately, if you miss the warning, dangling wires are easy to spot in simulation if you know where to look.

## 5.4 Expressing Structure: Connecting Components

In principle, we could model a complete digital system in Verilog by connecting a bunch simple logic gates using wires and assign statements in a single module. In practice, however, this would be unwieldy for all but the simplest designs: difficult to read, write, debug, and maintain in much the same way that a C program would be unwieldy if you packed the entire program into `main` without calling any other functions. To manage this complexity, designers in all fields learn to decompose complex systems into modular subsystems that can be designed and tested independently before connecting them together into the complete complex system. Further, if the submodules themselves are complex, they too may be decomposed into submodules. This modular “boxes-in-boxes” organization is called a *hierarchy*. We find examples of hierarchies in all sorts of manmade systems, ranging from computer programs, to automobiles, to governments and corporations. In Verilog, we express design hierarchy by breaking complex systems down into **modules** and wiring them together.

### 5.4.1 Modules, Instances, Ports, and Wires

A Verilog structural model consists of instances of modules connected together by wires attached to their ports. (Verilog also has built-in primitive logic gates—e.g. and, or, not, etc.—that can be wired up like instances of modules, but these are rarely used in practice).



Terminology:

module	a user-defined component that has ports and a body, where the body can be either a description of structure or behavior
module port	port defined for a module as seen from the perspective of the body of that module
instance	a component of a structure, either an instance of a module or an instance of a primitive logic gate
port instance	port on an instance of a module or on an instance of a primitive logic gate
wire	used to make connections to port instances, may be internal or connect to module ports

### 5.4.2 Instantiating and Connecting Modules by Port Name

To illustrate structural modeling in Verilog, we'll implement a new version of the NAND gate by connecting an instance of an AND gate module to an instance of an inverter module, rather than lumping

them together in a single module with `assign` statements as we did earlier. Before we can do this, we need to define modules for an AND gate and an inverter. Since the name `and` is a reserved keyword in Verilog (for a built-in primitive), we'll call these modules `my_and` and `my_inv` and implement them as behavioral models using `assign` statements.

```
module my_and (
    input    a,
    input    b,
    output   y
);

    assign y = a & b;

endmodule

module my_inv (
    input    a,
    output   y
);

    assign y = ~a;

endmodule
```

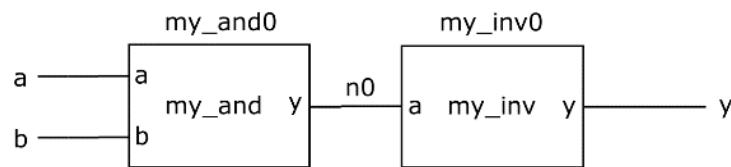
For completeness, we'll also define a behavioral model for an OR gate that we'll use later.

```
module my_or (
    input    a,
    input    b,
    output   y
);

    assign y = a | b;

endmodule
```

Given this library of module building blocks, the next step is to draw a schematic for the NAND gate and label all instances and wires with unique names.



The Verilog structural model for the NAND gate is basically just a formal text description of the schematic with the following sections:

- module name and port interface
- internal wire declarations

- module instance declarations with connections to port instances

Here's a complete structural model for a NAND gate using instances of modules `my_and` and `my_inv`:

```
module nand_structural (
    input    a,
    input    b,
    output   y
);

    wire n0;

    my_and my_and0 (
        .a (a),
        .b (b),
        .y (n0)
    );

    my_inv my_inv0 (
        .a (n0),
        .y (y)
    );

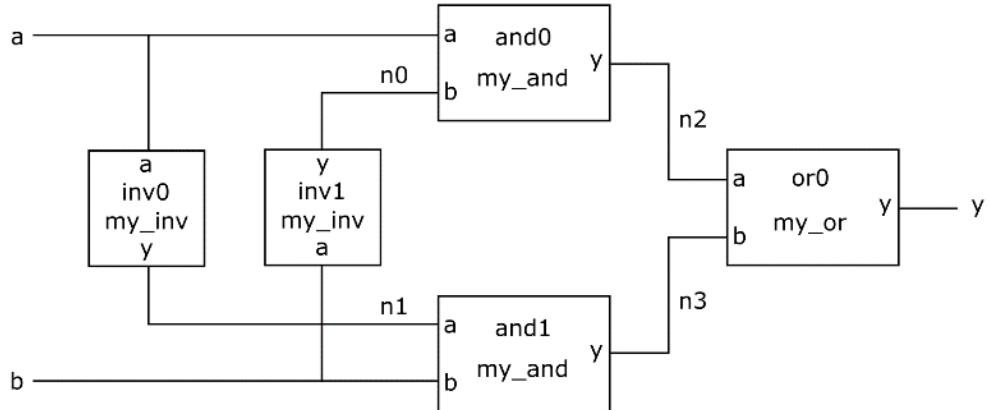
endmodule
```

The syntax of a module instance declaration is:

```
instance-module-name instance-name (
    .instance-port-name (wire-or-module-port-name),
    .
    .
    .instance-port-name (wire-or-module-port-name)
);
```

As a slightly more complex example that demonstrates hierarchy, we'll implement a structural model of an XNOR gate, composed of a structural model of XOR gate built from AND gates, OR gates, and inverters, connected to another inverter.

First, we draw the schematic for the XOR gate and label the instances and wires:



A structural model for the XOR gate is as follows:

```

module xor_structural (
    input    a,
    input    b,
    output   y
);

    wire n0, n1, n2, n3;

    my_inv inv0 (
        .a (a),
        .y (n1)
    );

    my_inv inv1 (
        .a (b),
        .y (n0)
    );

    my_and and0 (
        .a (a),
        .b (n0),
        .y (n2)
    );

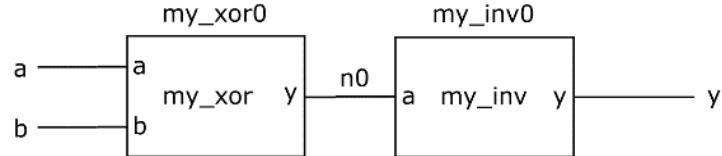
    my_and and1 (
        .a (n1),
        .b (b),
        .y (n3)
    );

    my_or or0 (
        .a (n2),
        .b (n3),
        .y (y)
    );

```

```
endmodule
```

Next, we draw the schematic for the XNOR gate



Finally, we implement the Verilog structural model for the XNOR gate from the schematic:

```
module xnor_structural (
    input    a,
    input    b,
    output   y
);

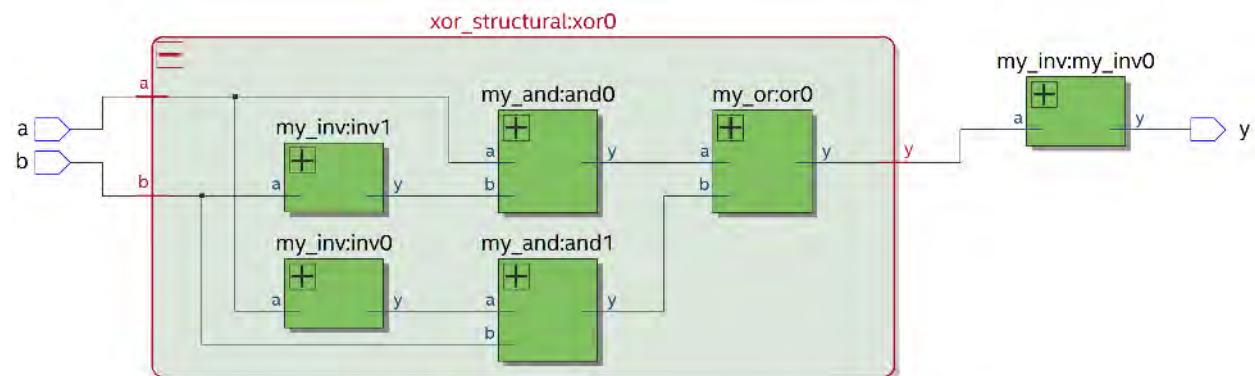
    wire n0;

    xor_structural xor0 (
        .a (a),
        .b (b),
        .y (n0)
    );

    my_inv my_inv0 (
        .a (n0),
        .y (y)
    );

endmodule
```

The complete design shown as synthesized by Quartus is



### 5.4.3 Instantiating and Connecting Modules by Port Order

There's a shorthand way for instantiating and connecting modules in a structural Verilog model. It's much more prone to errors than the longhand approach and we don't recommend using it. But it is commonly used, especially for instantiating modules with a small number of ports, so we present it for reference.

The shorthand is this: as an alternative to connecting wires to port instances by port name, you can list the connected wires in order that the ports were declared in the definition of the module being instantiated. For example, in the module `my_inv`, the ports were declared in the order `a`, `y` and in the module `my_xor`, the ports were declared in the order `a`, `b`, `y`. Using the alternative shorthand for connecting wire to ports, we could rewrite the module `my_xnor` as:

```
module xnor_shorthand (
    input    a,
    input    b,
    output   y
);

    wire n0;

    xor_structural xor0 (a, b, n0);

    // shorthand for:
    // xor_structural xor0 (
    //     .a (a),
    //     .b (b),
    //     .y (n0)
    // );

    my_inv inv0 (n0, y);

    // shorthand for:
    // my_inv inv0 (
    //     .a (n0),
    //     .y (y)
    // );

endmodule
```

While more convenient to type, the problem with the shorthand is that it makes the code harder to read. It also is easy to make the mistake of listing port connections in the wrong order or leaving one out, which can be hard bugs to find and fix. In short, avoid the shorthand and stick with connecting ports by name.

### 5.4.4 Built-In Primitive Logic Gates

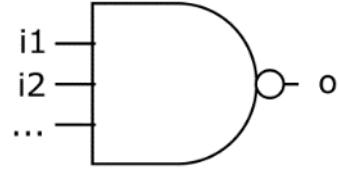
Verilog has a number of built-in primitive logic gates for and, or, not, etc. In practice these are rarely used, since it is usually more convenient to use `assign` statement. Because you may run into them, we describe their usage here, but don't recommend using them in your designs.

The first type of built-in logic primitive we'll consider are the basic logic gates with one output and multiple inputs: `and`, `nand`, `or`, `nor`, `xor`, `xnor`. You can think of a multiple input built-in primitive as being like a module with a single output port and two or optionally more inputs, for example:

```
built-in-primitive nand (
    output o,
    input i1,
    input i2,
    input i3, optional
    . . . );

    // Body that implements nand truth table

end-built-in-primitive
```



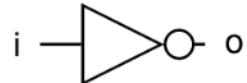
Note that for the built-in primitives, the output is the first port in the port list, followed by the inputs.

The built-in primitive for an inverter is `not`, and we'll consider it as having a single output and single input, where the output is first and the input is last in the port list.<sup>6</sup>

```
built-in-primitive not (
    output o,
    input i,
    . . . );

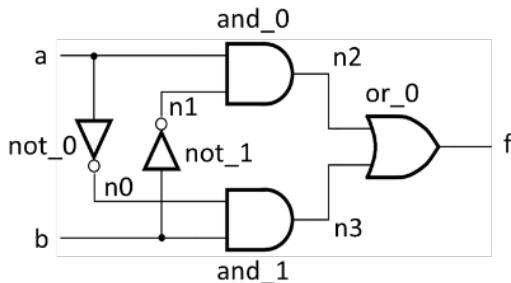
    // Body that implements inverter truth table

end-built-in-primitive
```



We can implement structural models of circuits using the built-in primitive logic gates by instantiating them and connecting their ports to wires in port order, just like the shorthand for instantiating and connecting user-defined modules. (Built-in primitive logic gates don't have named ports, so you can't connect the ports by name.)

As an example, we implement an XOR gate using the built-in `and`, `or`, and `not` primitives. A schematic for the XOR gate is:



We describe the structure in Verilog by first declaring the wires and then declaring each of the instances of components. This design has 4 internal wires, `n0`, `n1`, `n2`, `n3` which we declare as follows:

---

<sup>6</sup> The `not` primitive actually supports multiple copies of its output, where the outputs come first in the port list and the single input is last, but we'll ignore that for the purposes of these notes.

```
wire n0, n1, n2, n3;
```

We instantiate or declare an instance of a primitive component by specifying the type of component or module, its instance name, and the wires connected to each of its ports. The wires connections to the instance ports must be listed in the order specified in the component definition. For the built-in primitive logic gates, recall that the output is listed first, followed by the inputs. For example, and gate instance `and0` has its output connected to wire `n2` and its inputs connected to module port `a` and wire `n1`. The Verilog instance declaration is:

```
and and_0(n2, a, n1);  
  ↑   ↑   ↑  
module type instance name wire connections to ports
```

The complete Verilog implementation of the module, which we name `xor_primitives`, is

```
module xor_primitives (  
    input a,  
    input b,  
    output f  
);  
  
    wire n0, n1, n2, n3;  
  
    not not_0 (n0, a);  
    not not_1 (n1, b);  
    and and_0 (n2, a, n1);  
    and and_1 (n3, n0, b);  
    or  or_0   (f, n2, n3);  
  
endmodule
```

Note that when instantiating a built-in primitive, the instance name is optional. Thus we could replace

not not\_0 (n0, a);

with

not (n0, a);

Of course, it's much simpler to implement same logic using a single `assign` statement and no internal wires, which is why the built-in primitives are seldom used:

```
assign y = a & ~b | ~a & b;
```

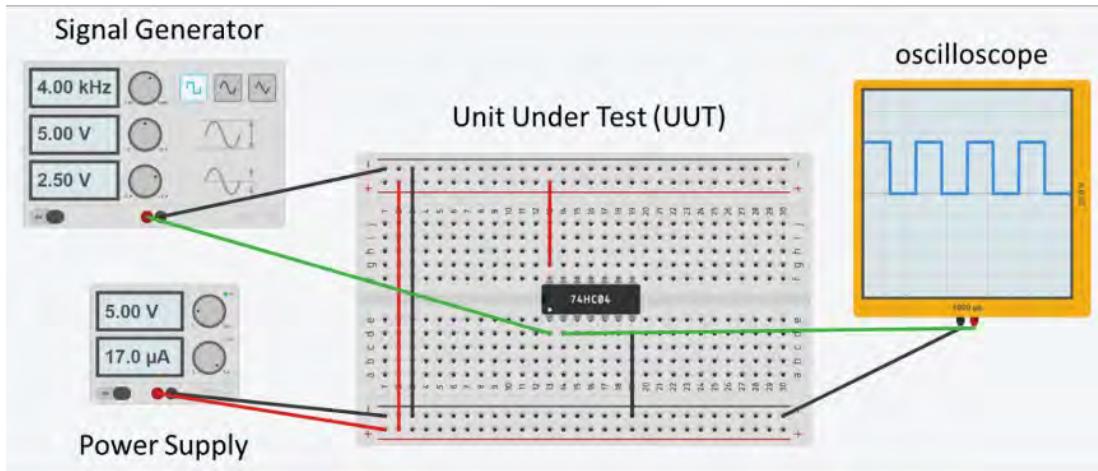
## 5.5 Simulating Module Behavior Using a Testbench

### 5.5.1 Testing Real Hardware

In the real, physical world, circuit designers use a setup called a *testbench* to test circuit behavior. Illustrated below, a testbench has several components:

- the circuit being tested, called the *unit under test (UUT)*
- a *signal generator* that generates the sequence of input signal values to the UUT

- an *oscilloscope* for viewing input and output signals from the UUT
- a *power supply* that provides electrical power to the UUT



The goal in testing a circuit is to verify that it is working properly and if it isn't—more likely than not the first time that you test something that you just built—to identify what the problem is so that you can fix it. For a combinational logic circuit, the most straightforward way to see what is and isn't working in the circuit is to apply all possible combinations of inputs to the circuits and compare the actual output values with the expected output values.

Suppose that the UUT is a 2-input combinational logic circuit such as a NAND, or XOR circuit, with inputs *a* and *b*. There are several reasons why a newly-designed circuit may not work properly, including getting the logic equations wrong or wiring the circuit up incorrectly. For a logic circuit with 2 inputs, there are 4 possible combinations of input values. This set of input combinations is sometimes called the *test stimuli*, *test pattern*, or *test vectors*. As we sequence through the set of combinations in the test pattern, we want to be sure that we allow enough time for the input values to propagate to the outputs of the circuit before moving on to the next combination. If we assume that the propagation delay is less than 10ns, a possible test procedure would be as follows:

```

start test
  set a = 0, set b = 0
  wait 10 ns, set a = 0, set b = 1
  wait 10 ns, set a = 1, set b = 0
  wait 10 ns, set a = 1, set b = 1
  wait 10 ns (for last test to finish)
end test

```

Verilog provides a means for constructing testbenches for simulating circuit behavior. Since Verilog only models the logical behavior of circuits and doesn't deal with currents and voltages, the testbench doesn't need to provide a power supply. The "oscilloscope" for viewing waveforms is part of the simulation environment and doesn't need to be included in the Verilog testbench either. A Verilog testbench is a module that contains the remaining two parts:

- an instance of a module that is the UUT
- a procedure for generating input signals to the UUT

We already know how to create an instance of a module; in the next section, we consider how to generate the input signals.

### 5.5.2 Generating Test Inputs in Verilog: initial Procedure Block and reg Data Type

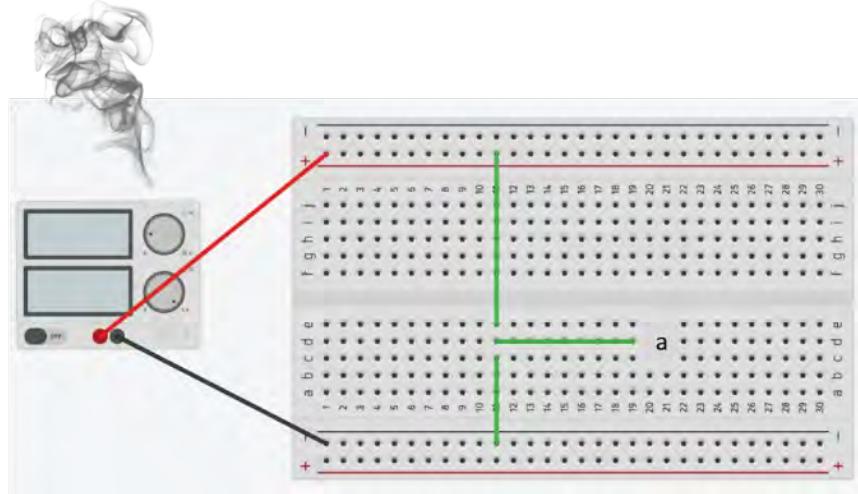
Observe that in the above test procedure, we change the values of input signals `a` and `b` every 10ns. In order to model this behavior in Verilog, we need some way of assigning new values to signals. Verilog `wires` and `assign` statements clearly won't work, since they create permanent connections. In fact, the following module,

```
module shortcircuit ();
    wire a;

    assign a = 0;
    assign a = 1;

endmodule
```

would model this configuration in hardware, which effectively creates a short circuit across the power supply:



In order to generate input signals, we need to introduce two new Verilog language features: a way of writing a procedure that runs at the start of a simulation and a way of representing signals that can be assigned values inside of the procedure. The type of procedure we will use is a Verilog `initial` block and the type of signal we will use is the register data type, `reg`.

While we've taken some pains to show that Verilog `wires` and `assign` statements are dedicated to modeling hardware and have no real counterparts in general purpose programming languages, Verilog `reg` signals and procedures have *some* similarities with C variables and functions, but also some significant differences. Whereas `wires` just pass values along, `regs` essentially *are* variables that store values that can be modified. Like C functions, Verilog procedures contain a sequence of statements. Also like in C, these statements include programming constructs such as variable (`reg`) assignment, `if-else` and `case` conditionals, and `for` and `while` loops. Unlike C functions, however, Verilog

procedures don't call each other and don't pass parameters. In fact, *very much unlike C functions*, the Verilog procedures inside of a module all run in parallel. For people used to programming in a language like C, this requires a real change in mindset and takes some getting used to. But this feature is extremely valuable for modeling the behavior of components of a hardware system, where once the system is powered on, all of the components start running at once.

From a simulation perspective, an **assign** statement continuously monitors the signals on the right hand side of the assignment operator (=) and recalculates the value of the signal on the left-hand side whenever one of the right-hand side values changes. By contrast, statements inside of an **initial** block evaluate once, in sequence from top to bottom beginning at the start of a simulation. As mentioned earlier, Verilog provides a rich set of statement types for modeling algorithmic behavior within a procedure that we will introduce later, but for now, we only need **reg** variable assignment statements. The syntax of an **initial** block for simulation is as follows:

```
initial begin
    reg-assignment; . . . reg-assignment;
    #delay reg-assignment; . . . reg-assignment;
    #delay reg-assignment; . . . reg-assignment;
    .
    .
    .
    #delay reg-assignment; . . . reg-assignment;
end
```

Given all this background, here's an example of a Verilog module named **signal\_generator** that uses an **initial** procedure and **reg** variables to generate the set of test signals described above:

```
`timescale 1ns/1ns
module signal_generator ();
    reg a;
    reg b;

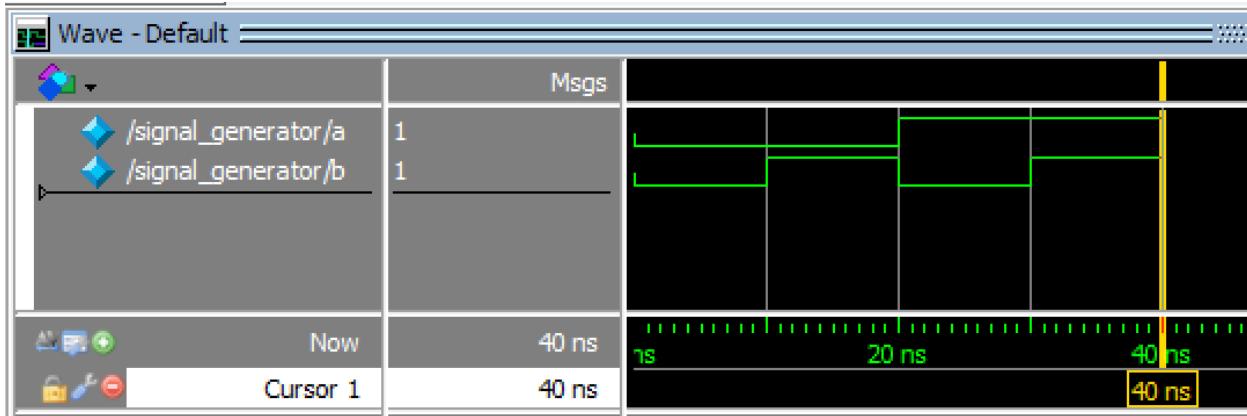
    initial begin
        a = 0; b = 0;
        #10 a = 0; b = 1;
        #10 a = 1; b = 0;
        #10 a = 1; b = 1;
        #10;
    end

endmodule
```

Notes:

- The directive `timescale 1ns/1ns defines the time units for the simulation. Note that the punctuation mark at the beginning of the directive is the “backward apostrophe,” located on the same key as the tilde (~) on most keyboards.
- The module doesn't have any ports; it generates its own signals internally.
- The two signals to be generated, **a** and **b**, are declared as **reg** data types.
- After the first set of signal assignments in the initial block, the remaining signal assignments are preceded by a delay of 10 time units (defined as **1ns** by the **timescale** directive)

Simulating this module with the ModelSim Verilog simulator produces the following waveforms:



### 5.5.3 Connecting a Unit-Under-Test (UUT) to the Signal Generator

To complete the testbench, we connect the unit-under-test (UUT) to signal generator. Here's a complete testbench for our first NAND gate design, `nand_simple`:

```
`timescale 1ns/1ns
module nand_simple_tb ();
    reg a;
    reg b;
    wire y;

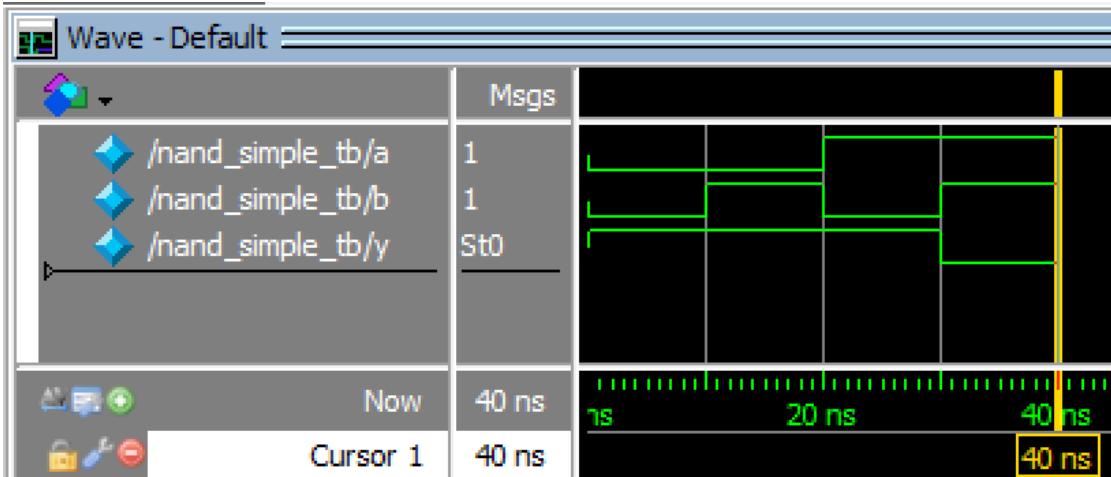
    nand_simple uut (
        .a(a),
        .b(b),
        .y(y)
    );

    initial begin
        a = 0; b = 0;
        #10 a = 0; b = 1;
        #10 a = 1; b = 0;
        #10 a = 1; b = 1;
        #10;
    end

endmodule
```

Notes:

- The testbench is a module with no ports since all signals are generated internally.
- The name of the testbench `nand_simple_tb` is the name of the UUT with “\_tb” appended to the end. This is the standard naming convention that you should follow—some tools require that testbench names end in “\_tb”.
- By convention, we name the instance of the UUT in a testbench “uut”.
- The `uut` input ports are connected to `regs` with the signals generated by the `initial` block.
- The `uut` output port is connected to a `wire`.



## 5.6 Multibit Signals

### 5.6.1 Vector Notation

Thus far, we've only considered Verilog signals that are single bits. Most digital systems, however, involve quantities represented by multiple bits, such as a 32-bit integer or an 8-bit ASCII letter. Verilog represents multibit quantities using vector notation. The range—and implicitly the size—of a vector is specified in square brackets as follows:

[*starting-bit-index*:*ending-bit-index*]

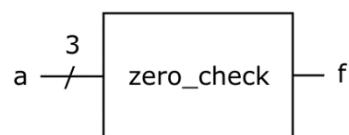
For example, a vector of 8 wires named *w*, with a starting index of 0 and an ending index of 7 is declared as

```
wire [0:7] w;
```

We refer to a single wire by its index in square brackets. For example, the first wire in vector *w* is *w[0]*. We can also refer to a range of wires, such as *w[3:5]*. Unlike an array in a programming language like C, Verilog vectors may be specified in either ascending or descending order. Descending order is very commonly used for vectors that represent numbers, where the most-significant bit (MSB) is the first element in the vector and the least-significant bit (LSB) is the last.

The following is an example of a module that checks if the value of a 3-bit signal *a* is zero by testing if each of the individual bits is zero.

```
module zero_check(
    input [2:0] a,
    output     f
);
    assign f = ~a[2] & ~a[1] & ~a[0];
endmodule
```



## 5.6.2 Representation of Numbers

Multibit numbers can be specified in binary, decimal, or hexadecimal. The specification format includes the number of bits and the base.

- Base specifiers:
  - b or B binary
  - d or D decimal (default)
  - o or O octal
  - h or H hexadecimal
- Examples:
  - 8'b1001\_1101 // Use underscore for readability
  - 32'HABCD // Pads with leading 0s
  - Note Unsigned numbers are stored as integers (at least 32 bits)

The numerical values of ASCII characters can be written as letters in quotes, e.g. "A" is equivalent to 8'd65 or 8'h40.

Example:

```
module numbers_tb();
    reg [7:0] n;

    initial begin
        n = 8'b10101010;
        #10 n = 8'b1010_1011;
        #10 n = 8'hcd;
        #10 n = 8'd33;
        #10 n = -8'd1;
        #10 n = "a";
        #10 n = "A";
        #10;
    end

endmodule
```

Simulation result with binary radix:

/numbers_tb/n	01000001	(10101010)	10101011	11001101	00100001	11111111	01100001	01000001
---------------	----------	------------	----------	----------	----------	----------	----------	----------

Simulation result with hexadecimal radix:

/numbers_tb/n	41	aa	ab	cd	21	ff	61	41
---------------	----	----	----	----	----	----	----	----

Simulation result with unsigned radix:

/numbers_tb/n	65	(170)	171	205	33	255	97	65
---------------	----	-------	-----	-----	----	-----	----	----

Simulation result with decimal (signed) radix:

/numbers_tb/n	65	-86	-85	-51	33	-1	97	65
---------------	----	-----	-----	-----	----	----	----	----

Simulation result with ASCII radix:

/numbers_tb/n	A	a	<	I	!	y	a	A
---------------	---	---	---	---	---	---	---	---

### 5.6.3 Arithmetic and Relational Operators

Verilog arithmetic and relational operators apply to multibit signals.

#### Arithmetic operators

addition	+	subtraction	-
multiplication	*	division	/
modulo	%	power	**

Example:

```
module adder4 (
    input [3:0] a,
    input [3:0] b,
    output [3:0] s
);

    assign s = a + b;

endmodule
```

#### Relational Operators

equal	==	not equal	!=
less than	<	less than or equal to	<=
greater than	>	greater than or equal to	>=

Example:

```
module zero_check_equal(
    input [2:0] a,
    output     f
);

    assign f = a == 3'b0;

endmodule
```

The testbench for the `zero_check_equal` module illustrates the use of multibit signals and vector notation

```
module zero_check_tb();
    reg [2:0]    a;
    wire        f;
```

```

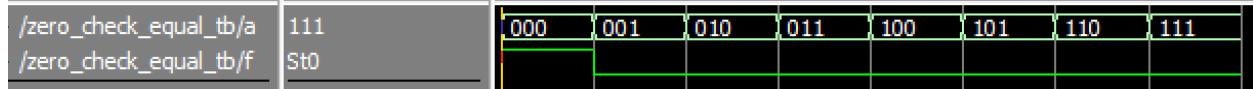
zero_check_uut(a, f);

initial begin
a = 3'b000; // 3-bit, binary value
#10 a = 3'b001;
#10 a = 3'b010;
#10 a = 3'b011;
#10 a = 3'b100;
#10 a = 3'b101;
#10 a = 3'b110;
#10 a = 3'b111;
#10;
end

endmodule

```

Simulation result:



#### 5.6.4 Reduction Operators

Reduction operators perform a bitwise logic operation across all the bits of a vector. The reduction operator is a unary operator like “~”, that is you specify the operator followed by the single operand.

##### Reduction operators

and	&	nand	$\sim\&$
or		nor	$\sim $
xor	$\wedge$	xnor	$\sim\wedge$ or $\wedge\sim$

Example:

```

module reduction (
    input [1:0] in,
    output      out_and,
    output      out_nand,
    output      out_or,
    output      out_nor,
    output      out_xor,
    output      out_xnor
);

    assign out_and  = &in;
    assign out_nand = ~&in;
    assign out_or   = |in;
    assign out_nor  = ~|in;
    assign out_xor  = ^in;
    assign out_xnor = ~^in;

```

```
endmodule
```

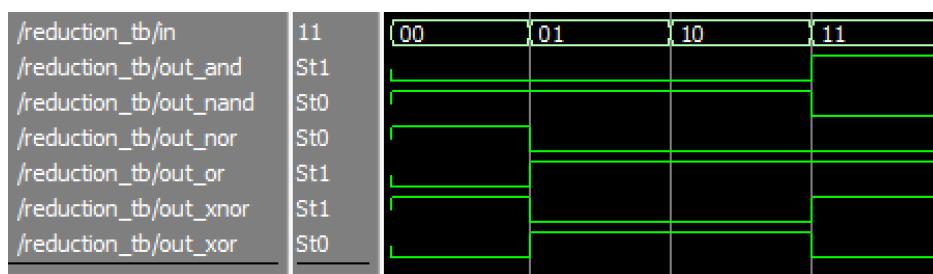
Testbench and simulation results:

```
module reduction_tb ();
    reg [1:0] in;
    wire      out_and;
    wire      out_nand;
    wire      out_or;
    wire      out_nor;
    wire      out_xor;
    wire      out_xnor;

    reduction uut (
        .in          (in),
        .out_and     (out_and),
        .out_nand   (out_nand),
        .out_or      (out_or),
        .out_nor     (out_nor),
        .out_xor     (out_xor),
        .out_xnor   (out_xnor)
    );

    initial begin
        in = 2'b00;
        #10 in = 2'b01;
        #10 in = 2'b10;
        #10 in = 2'b11;
        #10;
    end

endmodule
```



### 5.6.5 Rearranging Bits with the Concatenation Operator

“Bit Swizzling” is rearranging the bits in a multibit signal into a different order, or composing bits from different sources into a multibit signal. This is done using the Verilog concatenation operator of curly brackets, { }.

### 5.6.5.1 Example: Concatenating Vectors

The following example creates a 3-bit signal  $y$  composed of the 1-bit signal  $a$  followed by the 2-bit signal  $b$ :

```
module concat (
    input      a,
    input [1:0] b,
    output [2:0] y
);

    assign y = {a, b};

endmodule
```

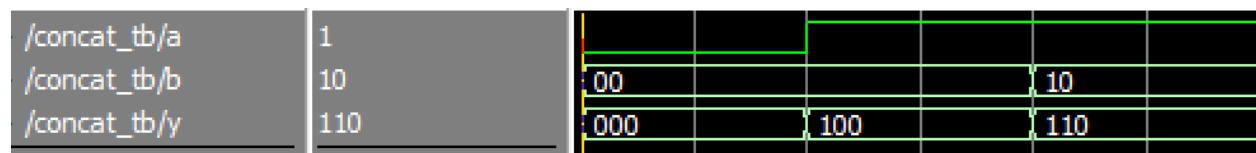
Testbench and simulation result:

```
module concat_tb ();
    reg      a;
    reg [1:0] b;
    wire [2:0] y;

    concat uut (
        .a (a),
        .b (b),
        .y (y)
    );

    initial begin
        a = 1'b0;  b = 2'b00;
        #10 a = 1'b1;  b = 2'b00;
        #10 a = 1'b1;  b = 2'b10;
        #10;
    end

endmodule
```



### 5.6.5.2 Example: Shifting

This example takes the 8-bit value  $a$  and shifts it to the left 2 places to produce a new 8-bit value  $y$ . The shift left operation discards the 2 most-significant bits of  $a$  and replaces the 2 least-significant bits of  $a$  with 0s.

```
module shift_left (
    input [7:0] a,
```

```

    output [7:0] y
  );
  assign y = {a[5:0], 2'b00};
endmodule

```

Testbench and simulation results:

```

module shift_left_tb ();
  reg [7:0] a;
  wire [7:0] y;

  shift_left uut (
    .a (a),
    .y (y)
  );

  initial begin
    a = 8'b00000000;
    #10 a = 8'b11110001;
    #10;
  end
endmodule

```

/shift_left_tb/a	11110001	00000000	11110001
/shift_left_tb/y	11000100	00000000	11000100

## 5.7 Example: Ripple Carry Adder

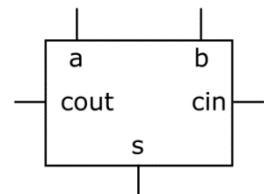
This example illustrates a hierarchical design using multibit vectors to implement a 3-bit ripple carry adder made up of 3 full adders. The full adders each operate on 1 bit and are wired together to produce a 3-bit adder.

### 5.7.1 Full Adder Using assign Statements

```

module full_adder (
  input a,
  input b,
  input cin,
  output s,
  output cout);
  assign s = a ^ b ^ cin;
  assign cout = a & b | b & cin | a & cin;
endmodule

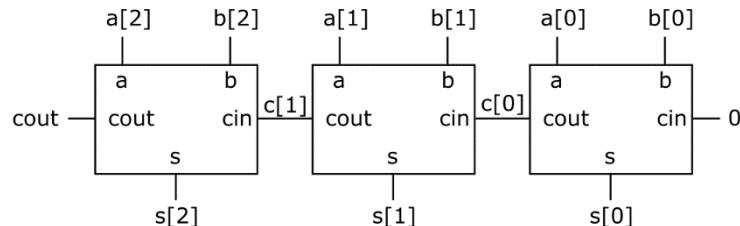
```



### 5.7.2 Structural Implementation

We use the preferred format of mapping signals to port instances by name to wire 3 full-adders together to build a 3-bit ripple carry adder. We implement the carry chain by passing a 0 into the carry-in of the least significant bit, 2 bits of internal wire  $c$ , and then map the final carry out to the `cout` port of the module.

```
module ripple3(
    input [2:0] a,
    input [2:0] b,
    output [2:0] s,
    output cout);
    wire [1:0] c;
    full_adder fa0 (
        .a      (a[0]),
        .b      (b[0]),
        .cin    (0),
        .s      (s[0]),
        .cout   (c[0])
    );
    full_adder fa1 (
        .a      (a[1]),
        .b      (b[1]),
        .cin    (c[0]),
        .s      (s[1]),
        .cout   (c[1])
    );
    full_adder fa2 (
        .a      (a[2]),
        .b      (b[2]),
        .cin    (c[1]),
        .s      (s[2]),
        .cout   (cout)
    );
endmodule
```



### 5.7.3 Testbench and Simulation Results

```
`timescale 1 ns/1 ns
```

```
module ripple3_tb();
    reg [2:0] a;
    reg [2:0] b;
    wire [2:0] s;
    wire cout;

    ripple3 uut (

```

```

.a(a),
.b(b),
.s(s),
.cout(cout)
);

initial begin
    a = 3'd0; b = 3'd0;
#10 a = 3'd0; b = 3'd0;
#10 a = 3'd1; b = 3'd0;
#10 a = 3'd1; b = 3'd1;
#10 a = 3'd0; b = 3'd1;
#10 a = 3'd3; b = 3'd4;
#10 a = 3'd3; b = 3'd2;
#10 a = 3'd4; b = 3'd3;
#10 a = 3'd4; b = 3'd5;
#10;
end

endmodule

```

Simulation results with binary radix:

/ripple3_tb/a	100	000	001	000	011	100	
/ripple3_tb/b	101	000		001		100	010
/ripple3_tb/cout	St1				010	011	101
/ripple3_tb/s	001	000	001	010	001	111	101

Simulation results with unsigned radix:

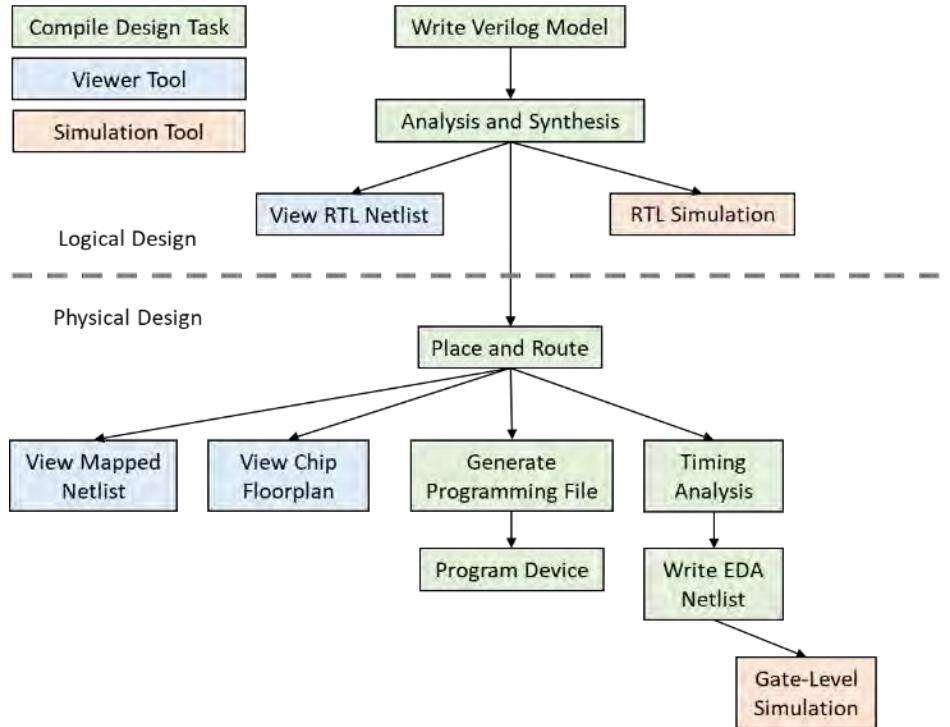
/ripple3_tb/a	4	0	1	0	3	4	
/ripple3_tb/b	5	0		1	4	2	3
/ripple3_tb/cout	1				5		
/ripple3_tb/s	1	0	1	2	1	7	5

## 5.8 Design Flow: Compilation and Simulation

### 5.8.1 Design Flow Overview

Say it again: *Verilog is a hardware description language (HDL), not a programming language.* Verilog has language features that are unique to describing the structure and behavior of hardware that don't have counterparts in a programming language like C. Even though some Verilog statements may *look* like C statements, they may do very different things, or worse, do similar things with subtle differences.

The figure below illustrates a basic design flow using Altera Quartus, starting from writing a Verilog hardware description model through generating an FPGA programming file and programming the FPGA device on the board.



The top half of the flow is *logical design*—developing a Verilog model that is logically correct. The bottom half of the flow is *physical design*—mapping the logical design to physical locations in the FPGA. There are three types of color-coded steps in the flow:

- Compile Design Tasks (green)
  - **Write Verilog Model:** use the text editor to write the Verilog
  - **Analysis and Synthesis:** first step in the compilation process, translates the Verilog model to what is called a *register-transfer level* (RTL) model composed of abstract combinational logic components (e.g. basic logic gates, multiplexors, decoders, adders) and memory storage elements (registers, not the same as a Verilog `reg` variable)
  - **Place and Route:** map the RTL model to specific logic elements (LEs), I/O pins, and other hardware resources inside the FPGA.
  - **Generate Programming File:** translate the placed-and-routed design to a bit stream file that can be downloaded to the FPGA over the USB BitBlaster cable.
  - **Timing Analysis:** generate a sorted list of delays through circuit paths.
  - **Write EDA Netlist:** generate a netlist file that is annotated with circuit path delays
- Viewer Tools (blue)
  - **View RTL Netlist:** generate schematic of the RTL model
  - **View Mapped Netlist:** generate schematic of design after it has been mapped into LEs in the FPGA. The netlist contains details inside of the LEs, specifically look-up tables (LUTs), multiplexors, and registers.
  - **View Chip Floorplan:** use the Chip Planner tool to view the physical layout of LEs, I/O pins and other hardware resources after place-and-route.
- Simulation Tools (orange)
  - **RTL Simulation:** simulate the RTL-level design without any physical delay
  - **Gate-Level Simulation:** simulate the physical gate-level netlist post place-and-route. Gives details of delays through LE components, specifically LUTs, registers, and multiplexors.

The fastest way to get to a working design on the FPGA board is to validate as much of the design as possible at the logic level before moving on to physical design. This means viewing the RTL netlist to make sure that it matches the intended structure and running RTL simulations with testbenches to check the logical behavior. Generally, it is *much* easier to identify and fix bugs in simulation than it is on the board.

As a rule, you should only move on to physical design after you are confident that you have a working design at the logic level. After running place-and-route, you can view the mapped netlist and run gate-level simulation to diagnose any timing problems. For this course, we generally won't worry about timing problems—and hence won't run gate-level simulation—because we'll be using a conservatively slow clock. Viewing the chip floorplan can help diagnose hardware resource utilization problems, but given that the FPGA has over 114K logic elements, it is highly unlikely that any of your designs will use more than a few percent of the total available LEs. Hence we generally won't use the Chip Planner tool, either. In summary, once you are confident that you have a working design at the logic level, you can safely run the remainder of the compilation tasks: place-and-route, generating a programming file, and programming the FPGA.

Even though we won't be using gate-level simulation or viewing the mapped netlist or chip floorplan in this course, it is still instructive to run through a couple of examples using these steps in order to get a better idea of what really happens when you compile a Verilog model for an FPGA.

## 5.8.2 Example: 5-Input OR Gate

As a first example, we'll look at the design of a 5-input OR gate, where the inputs are connected to switches and the output is connected to an LED on the DE2-115 board. As simple as the design seems, it is interesting because the FPGA logic elements (LEs) contain 4-input look-up tables (LUTs), so one LE won't be sufficient.

### 5.8.2.1 Verilog Model

Here's the Verilog model for the OR gate:

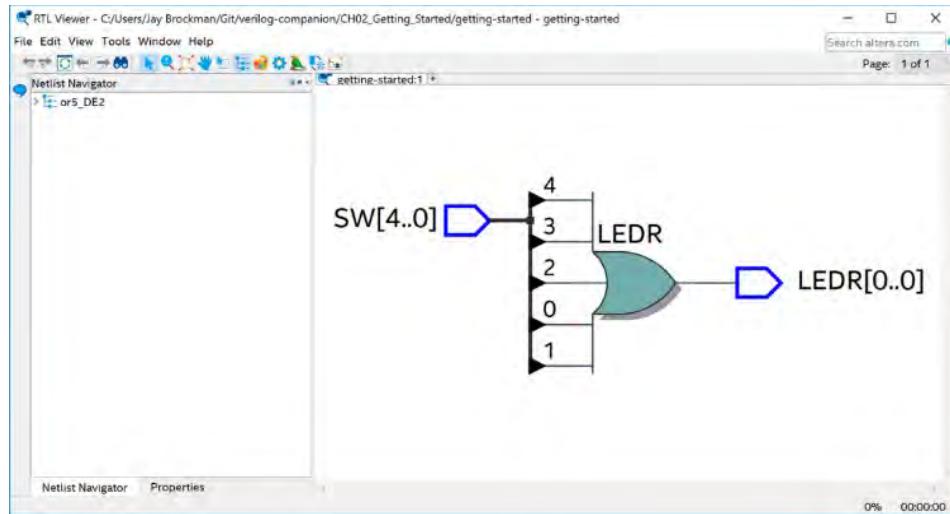
```
module or5_DE2 (
    input [4:0] SW,
    output [0:0] LEDR
);

    assign LEDR[0] = SW[0] | SW[1] | SW[2] | SW[3] | SW[4];

endmodule
```

### 5.8.2.2 RTL Netlist

The Analysis and Synthesis task generates the RTL Netlist as a single 5-input OR gate.



### 5.8.2.3 RTL Simulation

We write a testbench that will stimulate a change in the output of the OR gate. The test vectors consist of the inputs initially set to all 0s and then change the least-significant input to a 1.

```

`timescale 1ns/1ns
module or5_DE2_tb ();
    reg [4:0] SW;
    wire [0:0] LEDR;

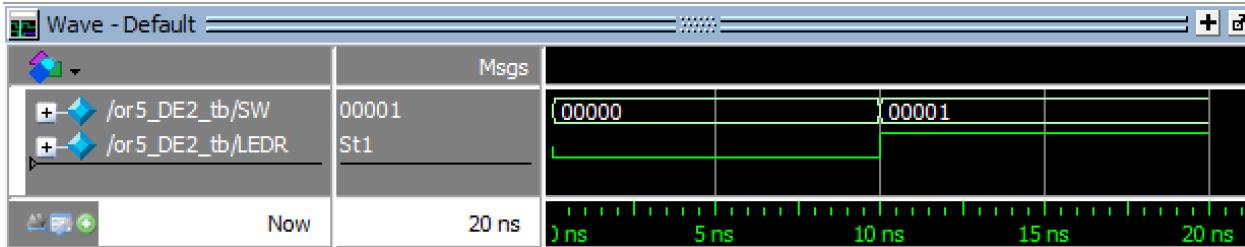
    or5_DE2 uut (
        .SW(SW),
        .LEDR(LEDR)
    );

    initial begin
        SW = 5'b00000;
        #10 SW = 5'b00001;
        #20;
    end

endmodule

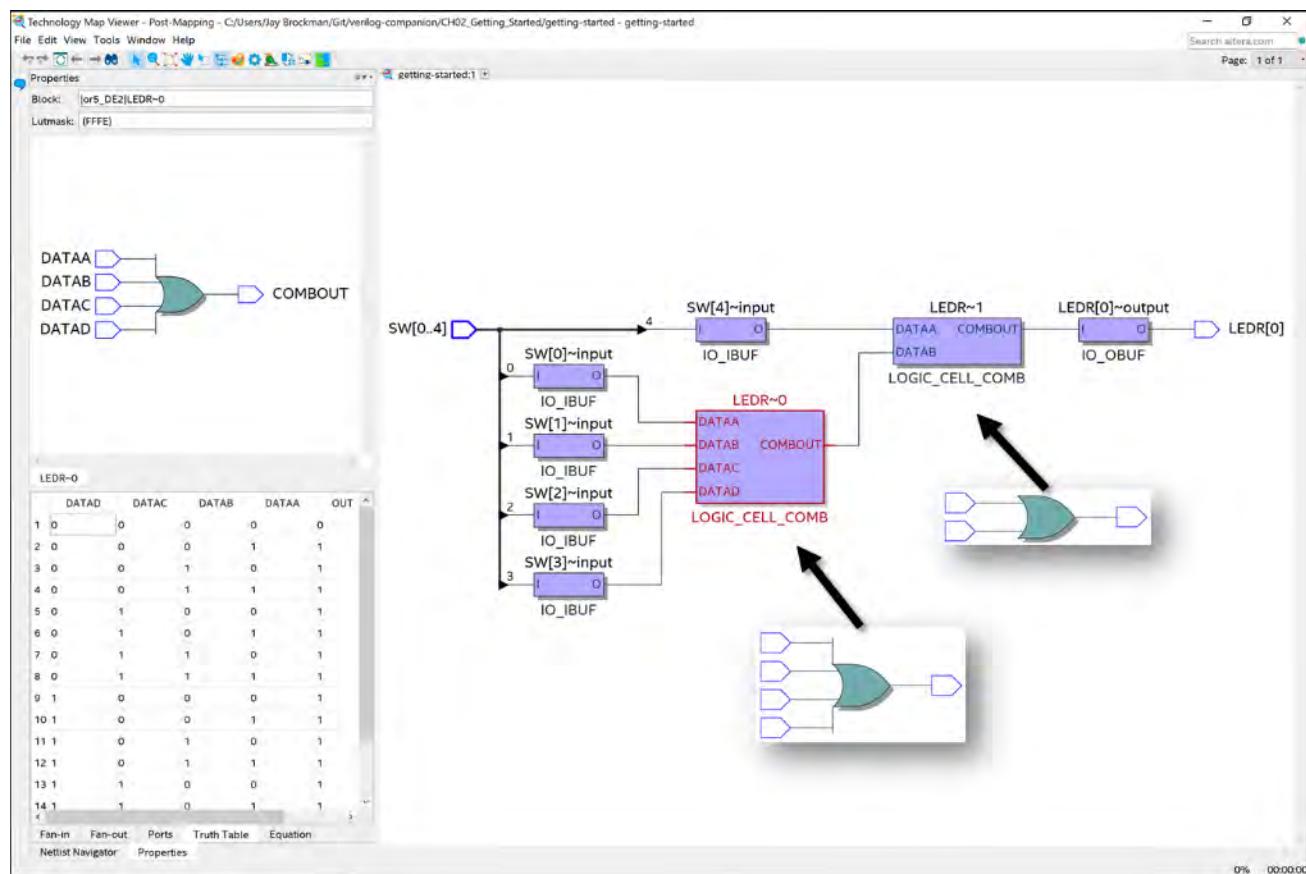
```

RTL Simulation gives the expected result, where the output LEDR changes from a 0 to a 1 when the inputs SW change from all 0s to having least-significant bit SW[0] be a 1. Note that in RTL Simulation, outputs respond instantaneously to changes in input values—there is zero delay.



### 5.8.2.4 Technology-Mapped Netlist

A logic element in the FPGA contains a 4-input LUT, so a 5-input OR gate won't be able to fit into a single LE. After running Place and Route, we view the technology-mapped netlist to see how the OR gate was implemented. The Technology Map Viewer (Post-Mapping) shows the following:

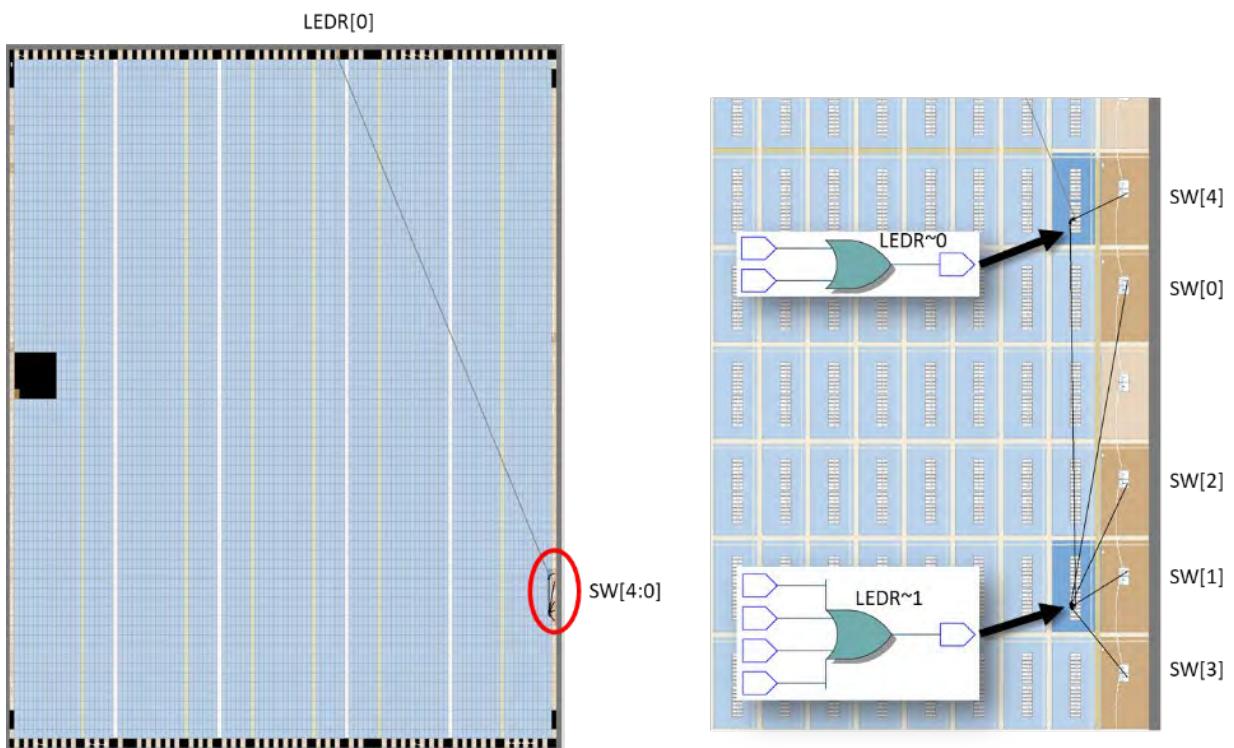


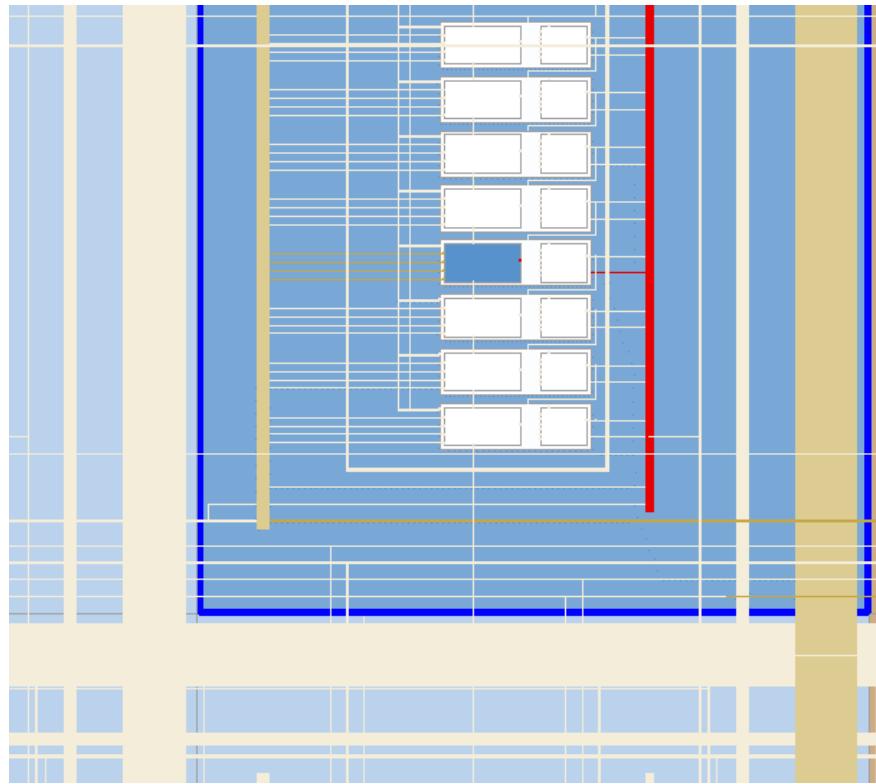
We see that the 5-input OR gate was mapped into 2 LEs, shown on the netlist schematic as **LOGIC\_CELL\_COMB** with instance names **LEDR~0** and **LEDR~1**. Highlighting instance **LEDR~0** shows its properties including a logic symbol and truth table in the pane in the left side of the viewer. We see that **LEDR~0** implements a 4-input OR gate. Similarly, **LEDR~1** implements a 2-input OR gate.

The technology-mapped netlist also include input/output buffers **IO\_BUF** on each of the input and output pins. The purpose of these is provide extra current for driving long wires on or off chip. As we will see, these buffer can add significantly to the delay through the circuit.

### 5.8.2.5 Chip Plan View

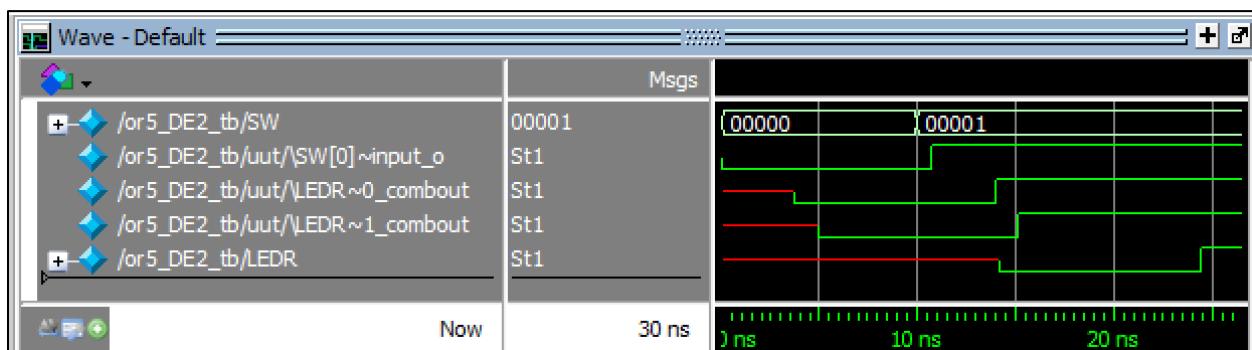
The view in the Chip Planner tool gives the physical layout of the FPGA, showing where the LEs and I/O pins have been placed and how connections between them are routed. From the figure below, we see that the 2 LEs were placed in the lower-left corner of the FPGA, near the pin connections for the switches. As a result of this placement, the connections to the switch input pins  $SW[4:0]$  are very short, but there is a long route to the output pin  $LEDR[0]$ . The thin black interconnection lines, sometimes called “rat’s nest” connections indicate what is connected to what, but don’t indicate the actual usage of horizontal and vertical routing channels. Zooming in more closely to logic element  $LEDR\sim 1$  shows details of the routing channel connections as red and gold colored wires.



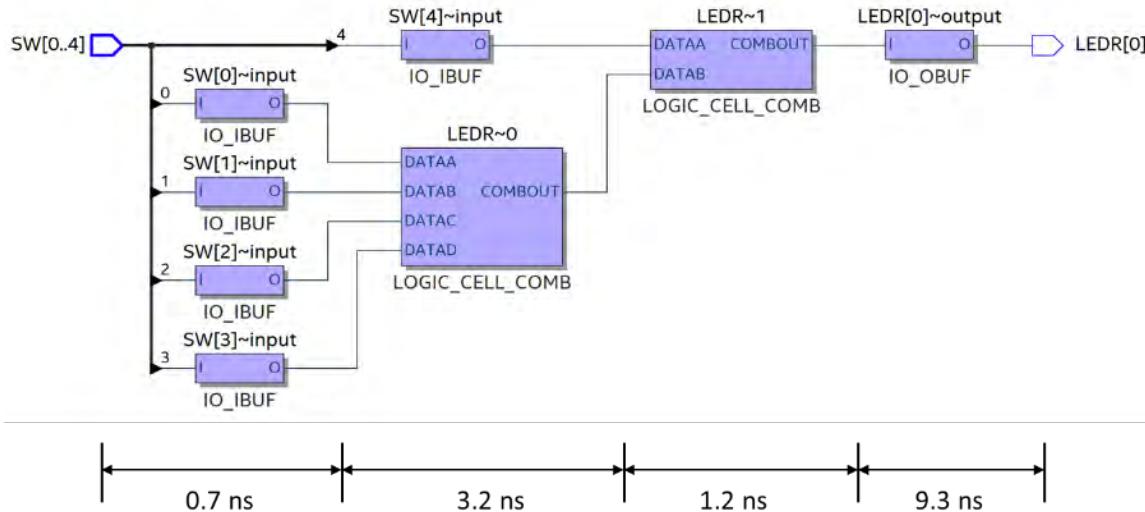


### 5.8.2.6 Gate-Level Simulation

Unlike RTL Simulation, in which outputs respond instantaneously to input changes with zero delay, Gate-Level Simulation models simulate the delays through logic elements as well as the delays through the interconnect network. The Gate-Level Simulation output below shows the waveforms at 5 points in the technology mapped netlist: (1) at the SW input pins, (2) at the output of the SW[0] input buffer, (3) at the output of 4-input OR gate LEDR~0, (4) at the output of the 2-input OR gate LEDR~1, and (5) at the output pin LEDR[0].



For convenience, we note the delays through each of the stages of the circuit on the technology-mapped netlist schematic.



The propagation delay through each of the stages of the circuit is on the order of a few nanoseconds. Interestingly, the longest delay is from the output of the second LE with the 2-input OR gate  $LEDR\sim 1$ , through the output buffer, to the output pin, which takes 9.3 ns. The reason that this is so much slower than the other stages is because of the delay of the long route from where the LEs were placed in the FPGA to the output pin on the top of the chip.

### 5.8.3 Example: 4-Bit Adder

As another example, we consider each of the steps in the design of a 4-bit adder. What makes this example interesting is seeing how the hardware compiler deals with the Verilog addition operator “+”.

#### 5.8.3.1 Verilog Model

The Verilog model for a 4-bit adder using a simple assign statement is repeated below:

```
module adder4 (
    input [3:0] a,
    input [3:0] b,
    output [3:0] s
);

    assign s = a + b;

endmodule
```

We'll need a testbench that will stimulate the slowest path in the circuit when we run gate-level simulation. This would be a set of test inputs that causes a carry signal to ripple through all the stages of the adder. We can do this by adding 4'b0001 to 4'b1111:

```
`timescale 1ns/1ns
module adder4_tb ();
    reg [3:0] a;
    reg [3:0] b;
    wire [3:0] s;
```

```

adder4 uut (
    .a (a),
    .b (b),
    .s (s)
);

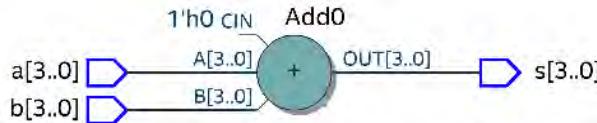
initial begin
    a = 4'b1111;  b = 4'b0000;
    #10 b = 4'b0001;
    #10;
end

endmodule

```

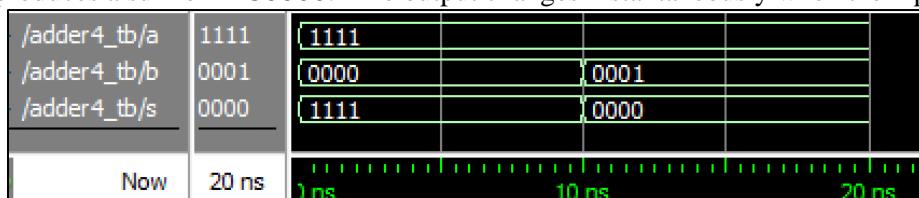
### 5.8.3.2 RTL Netlist

Analysis and Synthesis compiles the Verilog model to an RTL netlist. In this case, the compiler recognizes that the Verilog model can be mapped to a 4-bit adder in its library of abstract components.



### 5.8.3.3 RTL Simulation

RTL Simulation show the logical behavior of the design with zero delay. As expected, adding  $4'b0001$  to  $4'b1111$  produces a sum of  $4'b0000$ . The output changes instantaneously when the input changes.

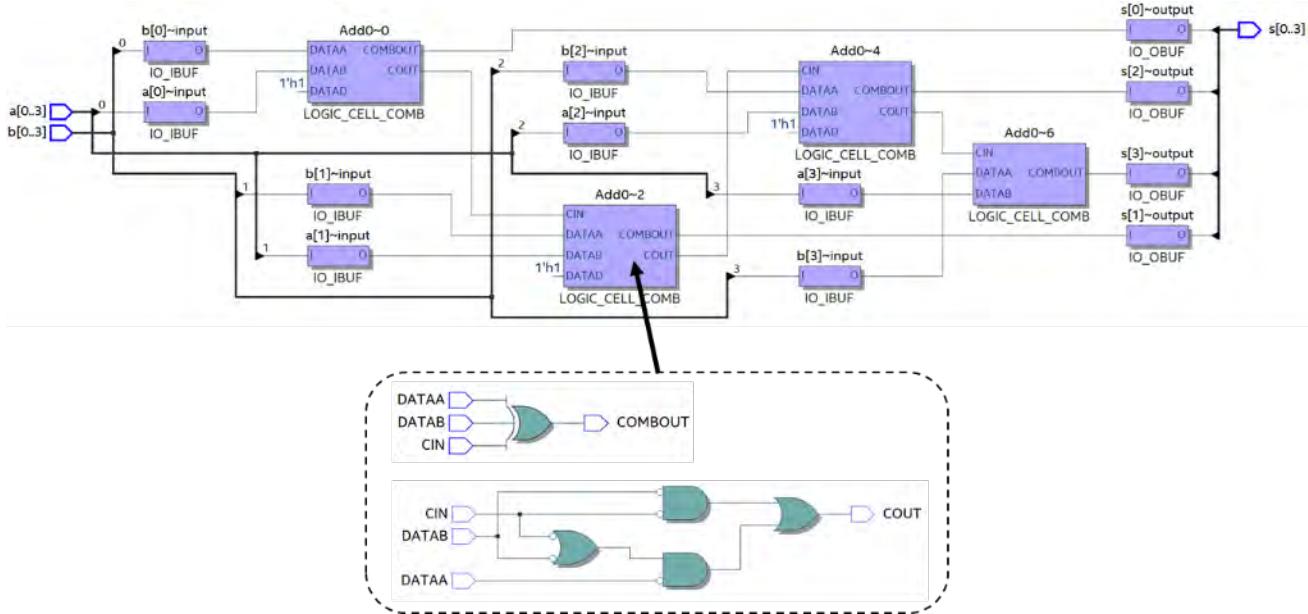


### 5.8.3.4 Technology-Mapped Netlist

This is where things get really interesting. After running Place and Route, we see that the hardware compiler implements the 4-bit adder with LEs as a 4-stage ripple-carry adder. Each of the 4 instances of `LOGIC_CELL_COMB` is a full adder (except the first stage which is a half adder) that contains LUTs that implement a sum function (3-input XOR) and a carry-out function.<sup>7</sup>

---

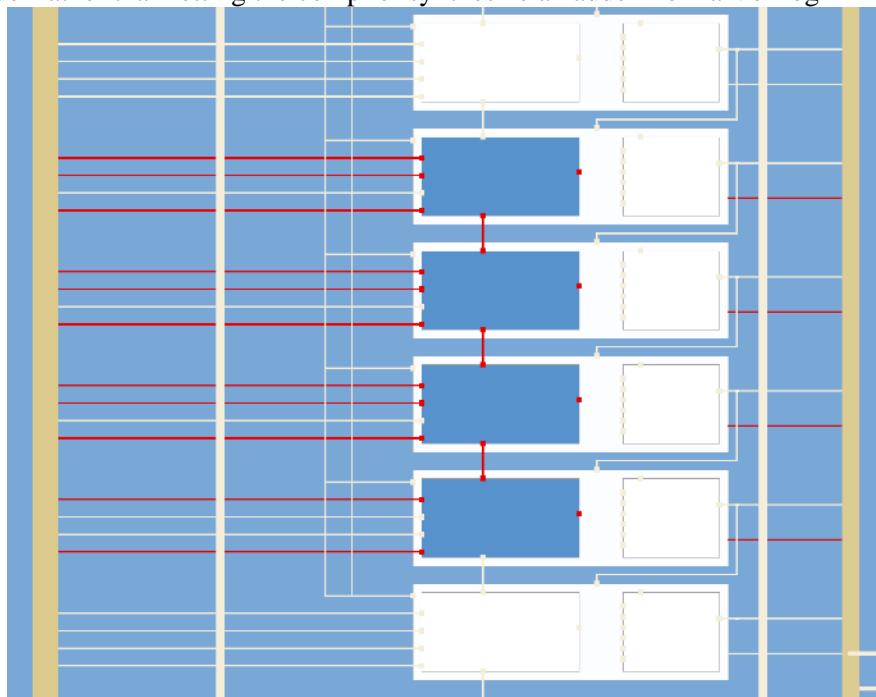
<sup>7</sup> If you look really closely, you will see that the carry-out circuit actually implements  $\sim COUT$ . The compiler is taking advantage of a known optimization of carry chains that reduces the delay by using the inverse of the carry signal through each stage.



### 5.8.3.5 Chip Plan View

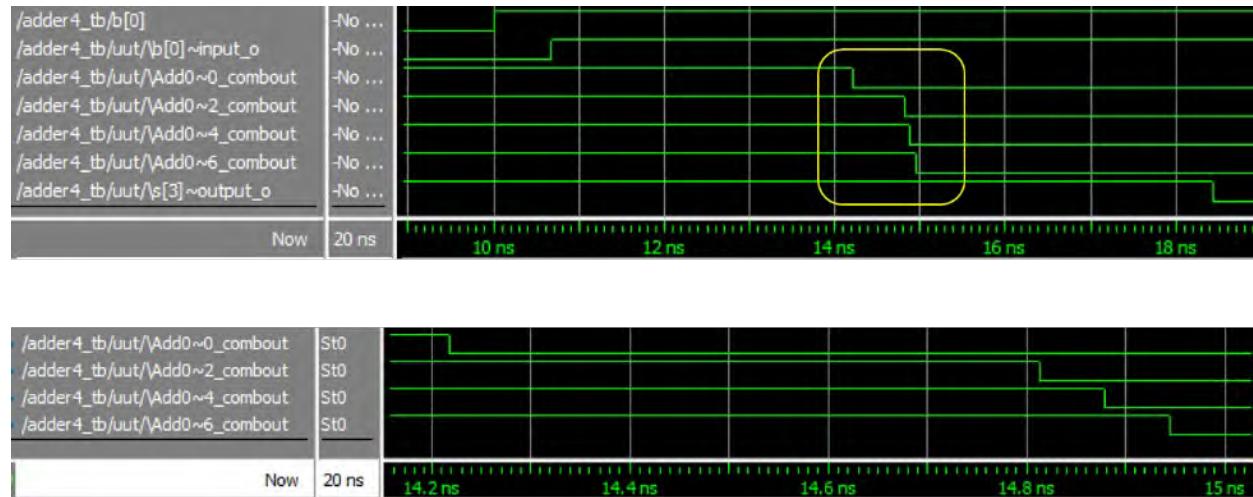
The floorplan viewed in the Chip Planner tool shows how the ripple-carry adder is mapped into LEs in the FPGA. We see that they are efficiently placed into 4 adjacent LEs in the same logic array block (LAB). We stated earlier that LEs in Altera Cyclone IV E contain a single 4-input LUT. Actually, they can also be configured into what is called “adder mode” with 2, 3-input LUTS for the special case when the compiler knows that it is implementing an adder. It also takes advantage special routing tracks between adjacent LEs for the carry chain.

Note that the compiler wouldn't have used these optimizations for adders if we had implemented our own ripple carry adder rather than letting the compiler synthesize an adder from a Verilog “+” operator.



### 5.8.3.6 Gate-Level Simulation

The gate-level simulation shows the rippling of the carry signal through the adder stages. The zoomed-in view of the highlighted region shows that the delay through the first stage is around 0.5 ns and that the delay through the remaining stages is less than 0.1 ns each.



# 6 Behavioral Modeling of Combinational Logic

Remember that *Verilog is a hardware description language (HDL), not a programming language*. In this Chapter, we will examine a Verilog language construct called an `always` block for describing the behavior of hardware that has no counterpart in a sequential, procedural programming language such as C. Perhaps the key difference between the behavior of hardware and the execution of a computer program is that all of the hardware components in a system are running all of the time, whereas the instructions in a C program execute sequentially one at a time. The Verilog `always` block provides a way for describing the behavior of continuously running hardware components.

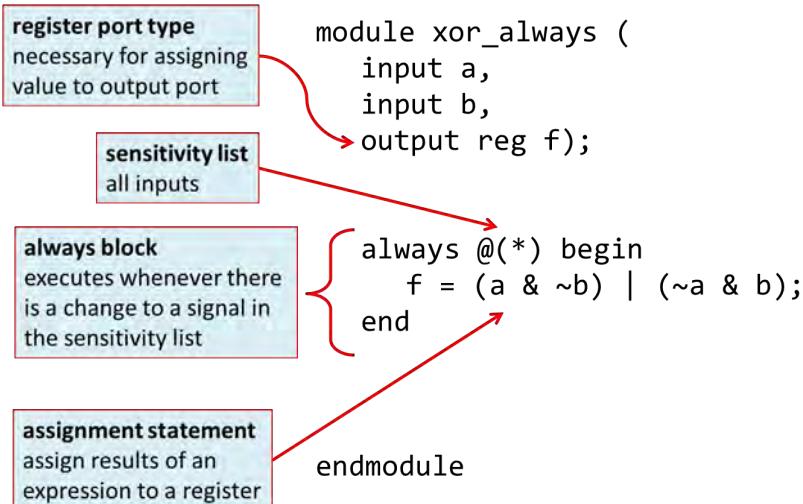
## 6.1 First Look at `always` Block

The `assign` statement is a convenient way of expressing the behavior of a module, but its use is limited to cases where the value of the assignment can be stated as a simple arithmetic or logic expression. Verilog can express more complex behaviors algorithmically using control structures such as `if` statements, `case` statements, and loops similar to those found in programming languages. A Verilog `always` block provides a means for expressing these more complex behaviors that a simple `assign` statement can't.

Like the `initial` block that we've already seen, a Verilog `always` block provides a way for writing procedures that describe behavior. The key difference between an `initial` block and an `always` block is that while an `initial` block runs once starting at the beginning of a simulation, an `always` block is "always" running. In relation to real hardware, if an `initial` block models the behavior of a signal generator in a testbench, running through a sequence of operations once, an `always` block models the behavior of the unit under test and its subcomponents, always ready to respond to changes in its inputs.

An `always` block models event-driven behavior, where the body of the block "fires" and evaluates in response to a triggering event, which is typically when the value of an input signal changes. We say that the `always` block is *sensitive* to changes in a set of signals, called the *sensitivity list*.

For now, we'll show how an `always` block works using a simple assignment, and then introduce its use with more complex control structures later. The example below shows an XOR gate module implemented using an `always` block.



An `always` block is overkill for this example, which in practice would be better served by a simpler `assign`, but it illustrates the basic operation. The block starts with the `always` keyword. The body of the block, which fires upon a triggering event, is nested between `begin` and `end` keywords. (If there is only a single statement, then the `begin` and `end` keywords are optional.) The sensitivity list is specified by an `@` sign followed by a list of signals that the outputs are “sensitive” to. For combinational logic, this is all of the inputs, which is denoted by `*`.<sup>8</sup>

Finally, the output port `f` is declared using the `reg` keyword. This is required for any signal that is assigned a value from within an `always` block. Many new Verilog users find this confusing: why do you need the `reg` keyword when you assign a value from inside an `always` block but not from an `assign` statement? The reason is that an `assign` statement connects a logic expression to a `wire` and by default, ports are `wires`. In contrast, an `always` block is a procedure that assigns values to variables, and a `wire` is not a variable. Thus we need to change the type of the output port from the default type `wire` to `reg`.

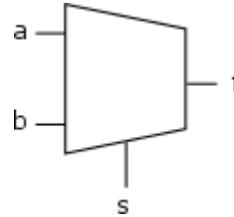
## 6.2 Multiplexor Using if else Statement

The syntax of the Verilog `if` `else` statement is similar to that found in other languages. If there is more than one statement to be executed if a condition is true, then they must be enclosed by `begin` and `end`, otherwise these keywords are optional. Note that for the module below, the sensitivity list consists of all the inputs.

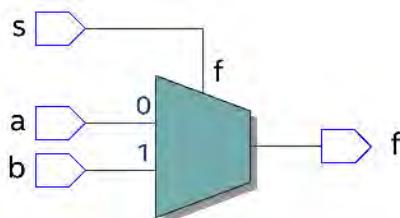
```
module mux_2x1(
    input a, b, s,
    output reg f
);

    always @(*)
        if (s == 0)
            f = a;
        else
            f = b;

endmodule
```



Following synthesis, we see that the RTL netlist is indeed a multiplexor:



## 6.3 Comparator Using Nested if else Statements

This comparator compares two 4-bit numbers `a` and `b` and sets one of three outputs `gt`, `lt`, or `eq` high depending upon whether  $a > b$ ,  $a < b$ , or  $a == b$ .

---

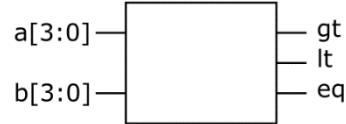
<sup>8</sup> The sensitivity list could also list all of the input signals explicitly, e.g. `@(a,b)`. This can lead to really nasty bugs if a signal is left out, so we will always use the `@(*)` notation for combinational logic.

```

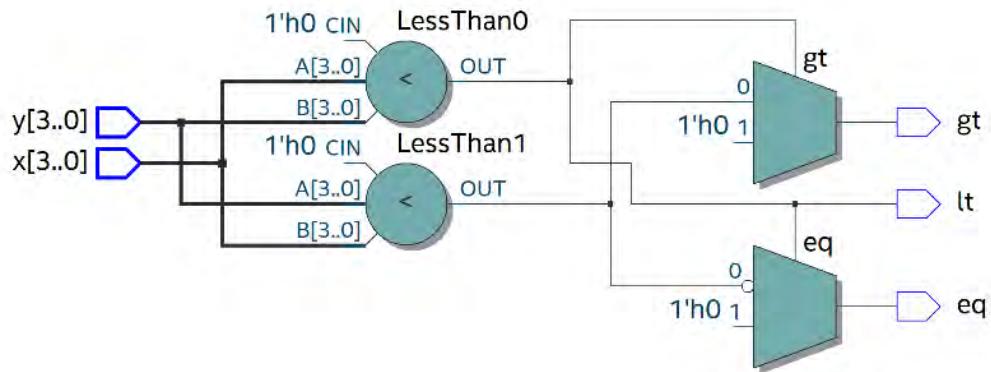
module unsigned_comparator (
    input [3:0] a, b,
    output reg gt, lt, eq
);

    always @(*)
        if (a < b) begin
            gt = 0;  lt = 1;  eq = 0;
        end
        else if (a > b) begin
            gt = 1;  lt = 0;  eq = 0;
        end
        else begin
            gt = 0;  lt = 0;  eq = 1;
        end
    endmodule

```



After synthesis, the RTL netlist uses 2 “less than” primitives with opposite inputs for the comparisons and multiplexors to implement the **if-else** statements. Note that the bubble on the 0 input of the lower multiplexor means that the input is inverted.



A testbench and simulation results are as follows:

```

module unsigned_comparator_tb ();
    reg [3:0] x;
    reg [3:0] y;
    wire      gt;
    wire      lt;
    wire      eq;

    unsigned_comparator uut (
        .x      (x),
        .y      (y),
        .gt     (gt),
        .lt     (lt),
        .eq     (eq)
    );

```

```

);
initial begin
    x = 5;  y = 7;
#10 x = 7;  y = 5;
#10 x = 5;  y = 5;
#10;
end
endmodule

```



## 6.4 Multiplexor Using case Statement

The syntax of a Verilog `case` statement is as follows:

```

case (expression)
  match-value: begin
    statements
  end
  match-value: begin
    statements
  end
  .
  .
  default: begin
    statements
  end
endcase

```

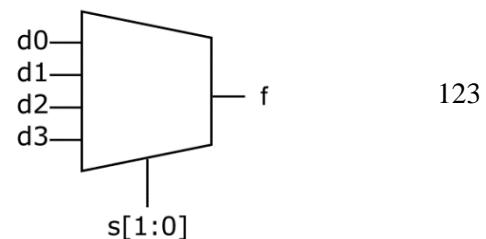
The `case` statement evaluates `expression` and then compares its value with the `match-values`. If it finds a match, it evaluates the `statements` for that case. If there is only a single statement for that case, then the enclosing `begin` and `end` are optional. If there are no matches, it evaluates the `default` case.

Here's an example of a 1-bit wide, 4-to-1 multiplexor using a `case` statement: The `expression` to be matched is the 2-bit wide signal `s`. The `match-values` consist of the 4 possible values of a 2-bit signal. There is no `default` specified because all possible cases are covered.

```

module mux4 (
  input d0, d1, d2, d3,
  input [1:0] s,

```



```

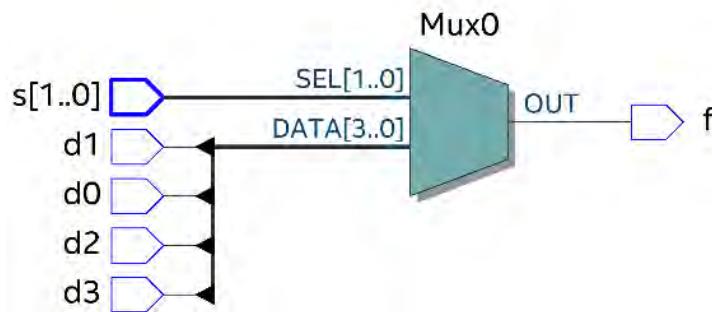
output reg f);

always @(*)
  case (s)
    0: f = d0;
    1: f = d1;
    2: f = d2;
    3: f = d3;
  endcase

endmodule

```

Synthesis maps this to 4:1 multiplexor primitive. Note that RTL netlist viewer draws the symbol for the multiplexor with the select input on the left side rather than on the top or bottom.



## 6.5 4-Bit Wide Multiplexor Using case Statement

The previous example is easily modified to a 4-bit wide 4-to-1 multiplexor:

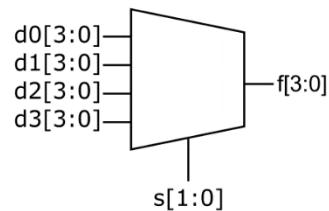
```

module mux_4x4_case(
  input [3:0] d0, d1, d2, d3,
  input [1:0] s,
  output reg [3:0] f);

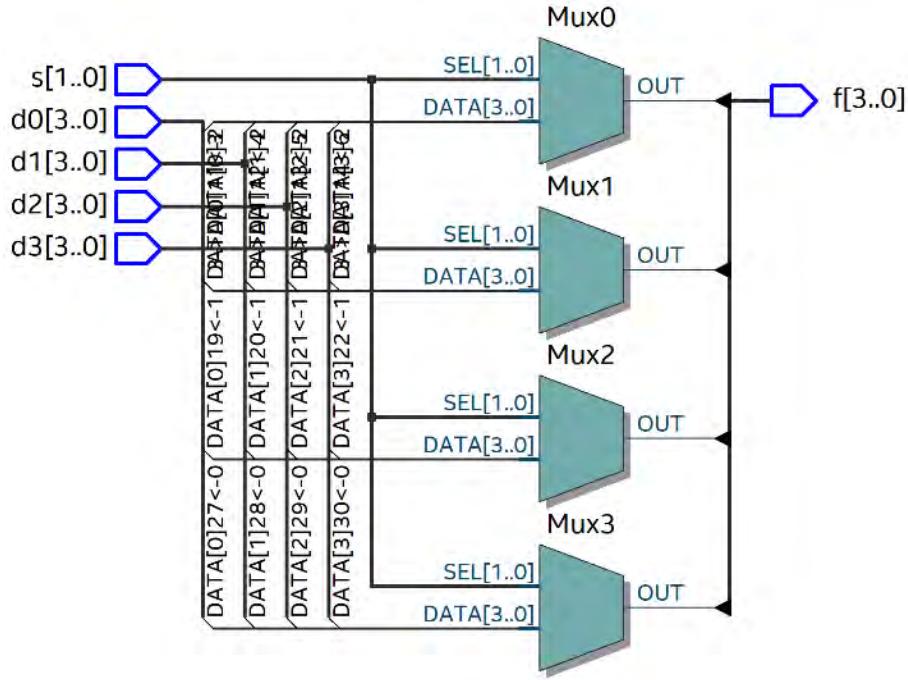
  always @(*)
    case (s)
      0: f = d0;
      1: f = d1;
      2: f = d2;
      3: f = d3;
    endcase

endmodule

```



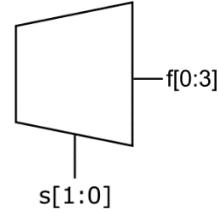
This synthesizes to 4 1-bit wide 4:1 multiplexors wired in parallel



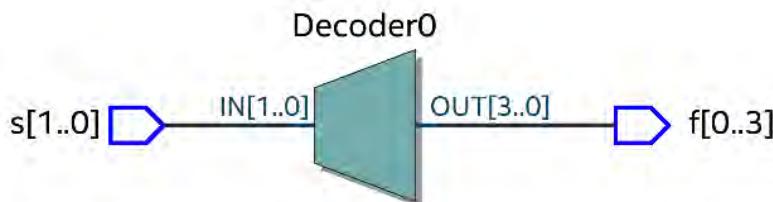
## 6.6 Decoder Using case Statement

```
module decoder_2x4_case (
    input [1:0] s,
    output reg [0:3] f // bits in ascending order (for example)
);

always @(*)
    case (s)
        0: f = 4'b1000;
        1: f = 4'b0100;
        2: f = 4'b0010;
        3: f = 4'b0001;
    endcase
endmodule
```



This synthesizes to a decoder. Note that RTL netlist viewer draws the symbol for the decoder with the select input on the left side rather than on the top or bottom.



## 6.7 Majority Gate Bit Concatenation and case Statement

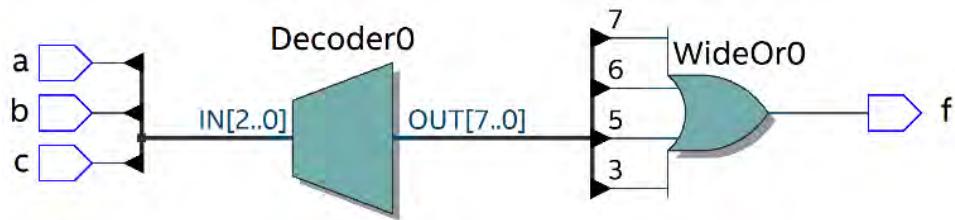
A majority gate returns a value of 1 if the majority of the inputs have a value of 1. In this example, the inputs are 3 individual 1-bit values. In order to take advantage of a `case` statement to implement the

logic, we concatenate the 3 individual bits into a 3 bit vector using the curly brackets { } concatenation operator.

```
module majority_case (
    input a, b, c,
    output reg f);

    always @(*)
        case({a, b, c})
            3'b000: f = 0;
            3'b001: f = 0;
            3'b010: f = 0;
            3'b011: f = 1;
            3'b100: f = 0;
            3'b101: f = 1;
            3'b110: f = 1;
            3'b111: f = 1;
        endcase
endmodule
```

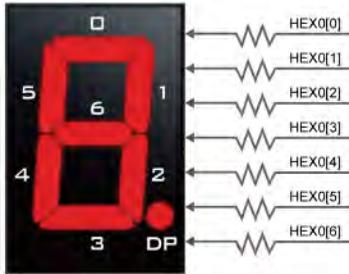
The RTL netlist is interesting, and illustrates how the synthesis tool often maps **case** statements to decoders.



The symbol for the decoder instance **Decoder0** has 3 select inputs **IN[3:0]** and 8 outputs **OUT[7:0]**. The numbering of the 8 decoder outputs corresponds to the values of the select input combinations, e.g. **IN = 3'b000** selects **OUT[0]**, **IN = 3'b001** selects **OUT[1]**, etc. In this example, the synthesized result takes the OR of the 4 cases of the decoder output where **f** should equal 1, namely when **IN = 3'b011**, **IN = 3'b101**, **IN = 3'b110**, or **IN = 3'b111**. In short, what the synthesis tool did was use a decoder and an OR gate to implement a sum-of-products Boolean expression.

## 6.8 7-Segment Hex Digit Display Decoder

**case** statements are the most convenient way of modeling component behavior as a truth table. A 7-segment hex digit decoder is a good example of this. This module translates the 4-bit input **in[3:0]** to a pattern of 7 illuminated segments **f[6:0]** on the display that form the corresponding hex digit. Note that segments are turned on when the output value for that segment is a 0 and turned off when the output value for the segment is a 1.



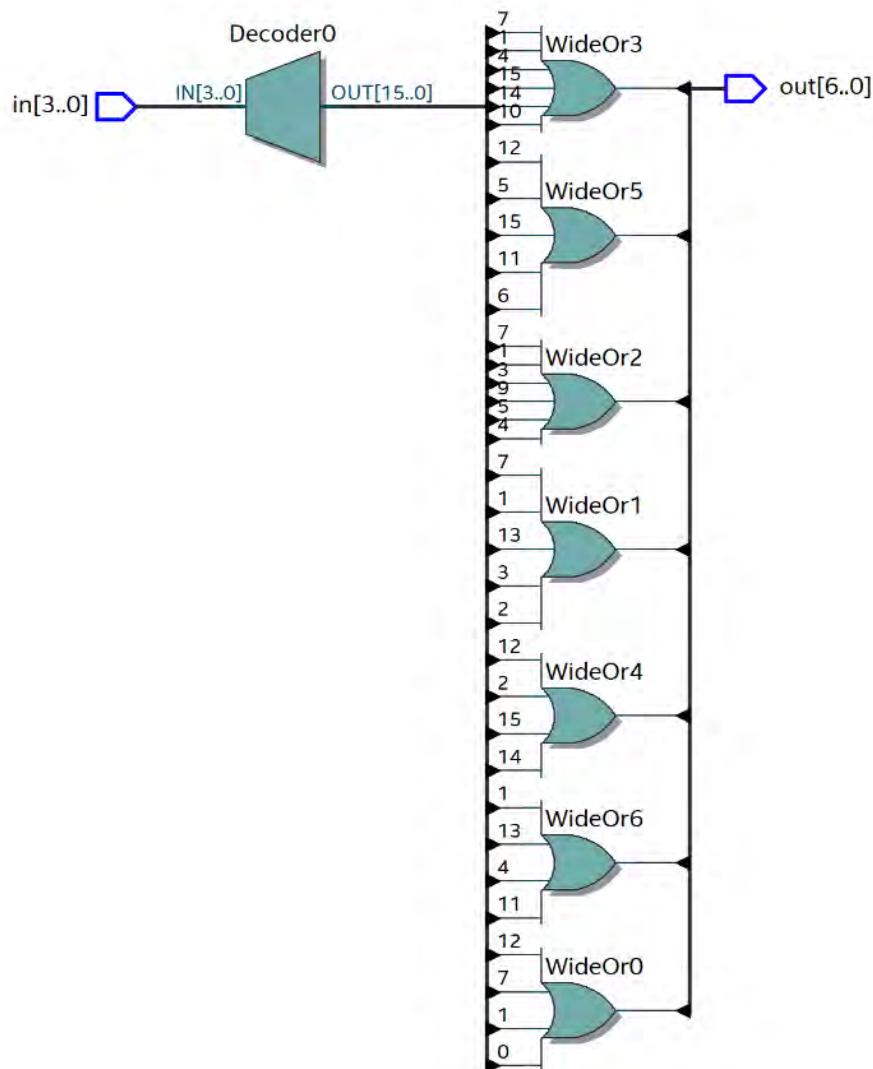
```

module hexdigit (
    input [3:0] in,
    output reg [6:0] out
);

    always @(*) begin
        case (in)
            4'h0: out = 7'b1000000;
            4'h1: out = 7'b1111001;
            4'h2: out = 7'b0100100;
            4'h3: out = 7'b0110000;
            4'h4: out = 7'b0011001;
            4'h5: out = 7'b0010010;
            4'h6: out = 7'b0000010;
            4'h7: out = 7'b1111000;
            4'h8: out = 7'b0000000;
            4'h9: out = 7'b0010000;
            4'ha: out = 7'b0001000;
            4'hb: out = 7'b0000011;
            4'hc: out = 7'b1000110;
            4'hd: out = 7'b0100001;
            4'he: out = 7'b0000110;
            4'hf: out = 7'b0001110;
        endcase
    end
endmodule

```

Similar to the previous majority gate example, the synthesis tool implements each of the 7 outputs as a sum-of-products, using a decoder to generate the product terms.



## 6.9 Demultiplexor: A Really Important Example to Understand!

Remember:

**Verilog is a hardware description language (HDL), not a programming language.** Verilog has language features that are unique to describing the structure and behavior of hardware that don't have counterparts in a programming language like C. Even though some Verilog statements may *look* like C statements, they may do very different things, or worse, do similar things with subtle differences.

This behavioral model for a demultiplexor is a good example of how a construct in Verilog that is similar to a construct in C doesn't quite do the same thing.

First, here's a simple English description of what a demultiplexor does: an n-output demultiplexor routes an input value  $d$  to one of the outputs as determined by select signal  $s$ . The values of the unselected outputs are  $\theta$ .

Here's one possible Verilog model for a 4-output demultiplexor. For each case of the select value, it sets the values of each of the 4 outputs:

```

module demux4_verbose (
    input      d,
    input [1:0] s,
    output reg y0,
    output reg y1,
    output reg y2,
    output reg y3
);

    always @(*) begin
        case (s)
            0: begin y0 = d; y1 = 0; y2 = 0; y3 = 0; end
            1: begin y0 = 0; y1 = d; y2 = 0; y3 = 0; end
            2: begin y0 = 0; y1 = 0; y2 = d; y3 = 0; end
            3: begin y0 = 0; y1 = 0; y2 = 0; y3 = d; end
        endcase
    end
endmodule

```

A more compact way of modeling the behavior is to initialize the values of each of the outputs to 0, and then overwrite the value of the selected output with d:

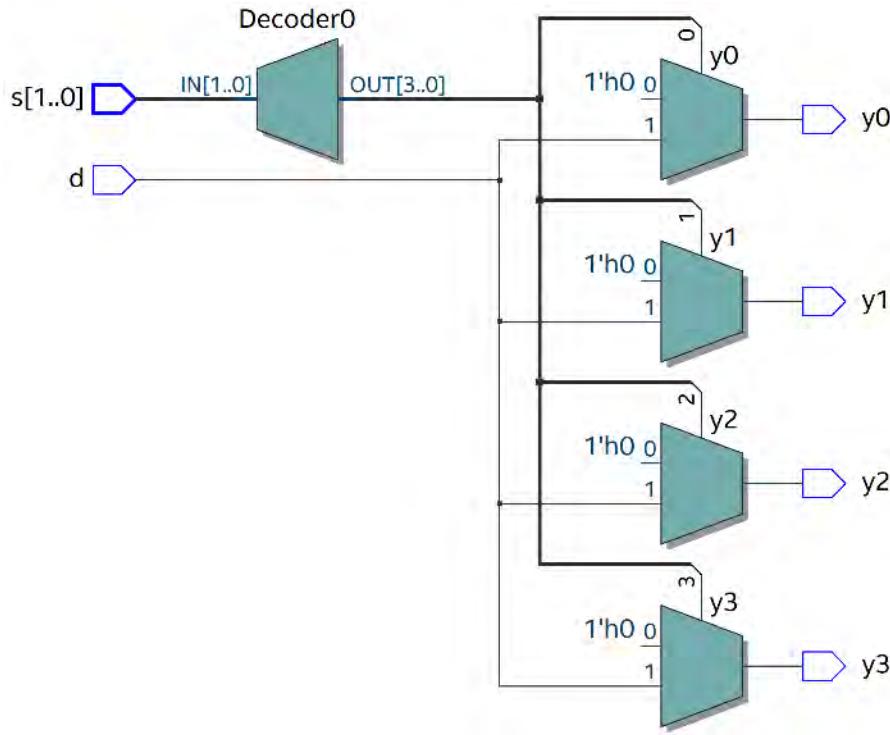
```

module demux4 (
    input      d,
    input [1:0] s,
    output reg y0,
    output reg y1,
    output reg y2,
    output reg y3
);

    always @(*) begin
        y0 = 0; y1 = 0; y2 = 0; y3 = 0;
        case (s)
            0: y0 = d;
            1: y1 = d;
            2: y2 = d;
            3: y3 = d;
        endcase
    end
endmodule

```

For both versions, the synthesizer produces the exact same RTL netlist, using a decoder to select an output and multiplexors to select whether that output gets the value of d or 0 (look at it carefully and convince yourself that it is doing the right thing):



Note that even though the second version of the Verilog model initialized the Y outputs to 0, there is absolutely nothing in the synthesized RTL netlist that indicates this. If you are confused by this, remember that *Verilog is a hardware description language, not a programming language*. While the statements in the `always` block are evaluated in order, they also occur in zero time. This behavioral model of a demultiplexor is *not specifying that the hardware has to set all of the y outputs to 0 before setting one of them to the value of d*. Rather, what it is saying is that when the `always` block fires (as a result of a change in the value of d or s), the hardware should assign d to one of the outputs and all of the others should be 0. It is the responsibility of the synthesizer to determine a hardware configuration that does this efficiently.

Furthermore—and this is a really important point—even though the signals y0, y1, y2, and y3 are `regs`, and that we initialized their values to 0 and then changed one of them, this doesn’t imply that the hardware that we are modelling should store anything. The demultiplexor is strictly a combinational logic component and not a sequential logic component and it shouldn’t contain any kind of memory element.

Simulation shows this combinational behavior. Below is a testbench that tests all combinations of select inputs to the demultiplexor, first with d = 1 and then with d = 0:

```
module demux4_tb ();
    reg      d;
    reg [1:0] s;
    wire     y0;
    wire     y1;
    wire     y2;
    wire     y3;

    demux4 uut (
        .d(d),
        .s(s),
        .y0(y0),
        .y1(y1),
        .y2(y2),
        .y3(y3)
    );

```

```

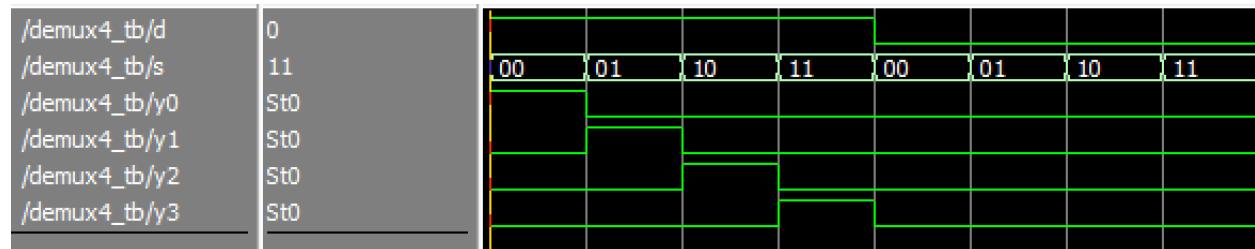
        .d      (d),
        .s      (s),
        .y0    (y0),
        .y1    (y1),
        .y2    (y2),
        .y3    (y3)
    );

initial begin
    s = 0;  d = 1;
#10 s = 1;
#10 s = 2;
#10 s = 3;
#10 s = 0;  d = 0;
#10 s = 1;
#10 s = 2;
#10 s = 3;
#10;
end

endmodule

```

Observe that for the first set of inputs, where  $s = 0$  and  $d = 0$ , the value of  $y0$  does *not* go to 0 before being set to 1:



## 6.10 Demultiplexor the Wrong Way: A Common Mistake to Avoid!

As mentioned in the last section, a demultiplexor is a strictly combinational logic component—it shouldn't store any values over time, just combine a set of inputs to produce an output, like a mathematical function would do. The following Verilog module *does not* describe a module with this behavior:

```

module bad_demux4 (
    input      d,
    input [1:0] s,
    output reg y0,
    output reg y1,
    output reg y2,
    output reg y3
);

```

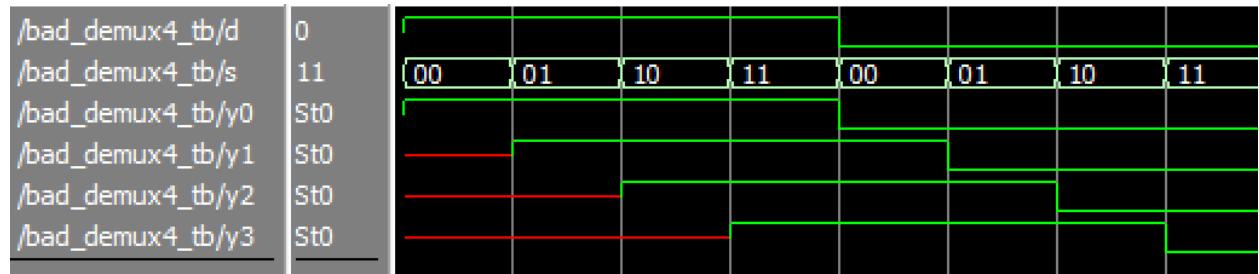
```

always @(*) begin
  case (s)
    0: y0 = d;
    1: y1 = d;
    2: y2 = d;
    3: y3 = d;
  endcase
end

endmodule

```

Here's the simulation result, using the same set of test inputs as for the working demultiplexor in the previous section:

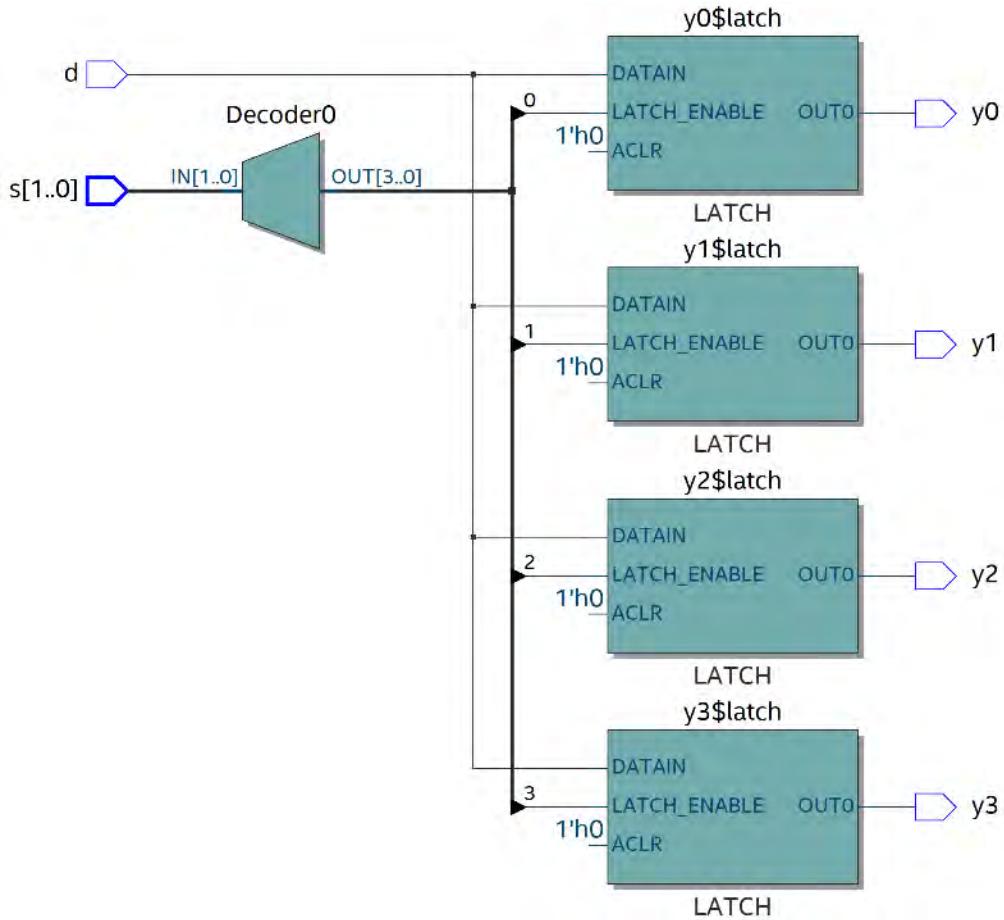


What's wrong with this result?

- The values of outputs  $y_1$ ,  $y_2$ , and  $y_3$  are unknown until they are selected, rather than 0.
- Once an output has been selected, it gets the value of input  $d$  and stays there, even if other outputs have been selected, rather than going to 0 when not selected. In other words, the module is exhibiting memory behavior, even though it shouldn't.

The problem is that the module doesn't define the values for the unselected outputs. (In `demux4_verbose`, each of the output values were defined for each case. In `demux4`, they were initialized to 0 before the `case` statement.) As a result, output values are undefined until they are selected. Since nothing causes the value of the outputs to change until they are selected, they retain their previous values.

What kind of hardware will produce this behavior? To find out, we look at the synthesized RTL netlist:



We observed that the design contains a LATCH on each output. Though we haven't studied them yet, a latch is a type of memory element, which shouldn't exist in a combinational logic circuit. If we look back at the Quartus flow messages, we see that synthesis produced the following warnings:

```

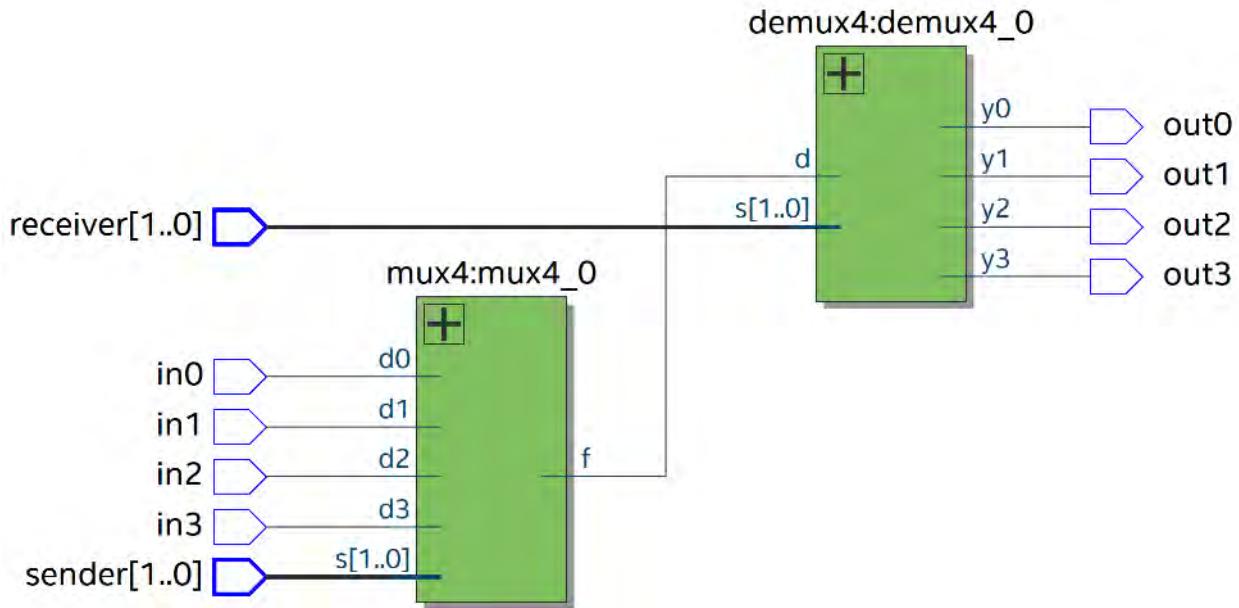
▲ 10240 Verilog HDL Always Construct warning at bad_demux4.v(10): inferring latch(es) for variable "y3"
▲ 10240 Verilog HDL Always Construct warning at bad_demux4.v(10): inferring latch(es) for variable "y2"
▲ 10240 Verilog HDL Always Construct warning at bad_demux4.v(10): inferring latch(es) for variable "y1"
▲ 10240 Verilog HDL Always Construct warning at bad_demux4.v(10): inferring latch(es) for variable "y0"

```

When we introduce sequential circuit design, we will use a different kind of memory element from a latch. In short, *your designs should never contain latches*. If they do, it is almost certainly because you left an output undefined for certain cases.

## 6.11 Putting It All Together: Router

A router is a component that routes one of a set of possible sources (senders) to one of a set of possible destinations (receivers). Sometimes called “switchboxes,” routers are very common building blocks for a wide variety of digital systems, ranging from computer networks down to configurable interconnects inside of FPGAs. A simple way of building a router is to use a multiplexer to select the sender and then use a demultiplexer to route the sender to the receiver, as illustrated below:



The wire connecting the multiplexor output to the demultiplexor input is called a *bus*.

```

module router (
    input  [1:0] sender,
    input      in0,
    input      in1,
    input      in2,
    input      in3,
    input  [1:0] receiver,
    output     out0,
    output     out1,
    output     out2,
    output     out3
);

wire bus;

mux4 mux4_0 (
    .d0 (in0),
    .d1 (in1),
    .d2 (in2),
    .d3 (in3),
    .s  (sender),
    .f  (bus)
);

demux4 demux4_0 (
    .d  (bus),
    .s  (receiver),
    .y0 (out0),
    .y1 (out1),
    .y2 (out2),
    .y3 (out3)
);

```

```
    .y2 (out2),  
    .y3 (out3)  
);
```

```
endmodule
```

Testbench and simulation result:

```
module router_tb ();  
  
reg      [1:0] sender;  
reg      in0;  
reg      in1;  
reg      in2;  
reg      in3;  
reg      [1:0] receiver;  
wire     out0;  
wire     out1;  
wire     out2;  
wire     out3;  
  
router uut (  
    .sender      (sender),  
    .in0        (in0),  
    .in1        (in1),  
    .in2        (in2),  
    .in3        (in3),  
    .receiver    (receiver),  
    .out0       (out0),  
    .out1       (out1),  
    .out2       (out2),  
    .out3       (out3)  
);  
  
initial begin  
    in0 = 1'b1;  in1 = 1'b0;  in2 = 1'b1;  in3 = 1'b0;  
    sender = 2'd0;  receiver = 2'd0;  
    #10 receiver = 2'd3;  
    #10 sender = 2'd2;  receiver = 2'd1;  
    #10;  
    $stop;  
end  
  
endmodule
```

/router_tb/in0	1								
/router_tb/in1	0								
/router_tb/in2	1								
/router_tb/in3	0								
/router_tb/sender	2	0							2
/router_tb/uut/bus	St1								
/router_tb/receiver	1	0							1
/router_tb/out0	0								
/router_tb/out1	1								
/router_tb/out2	0								
/router_tb/out3	0								

Demonstration on DE2-115 board:

```

module router_DE2 (
    input [7:0] SW,
    output [7:0] LEDR,
    output [3:0] LEDG,
    output [6:0] HEX0,
    output [6:0] HEX1
);

assign LEDR[7:0] = SW[7:0];

router n (
    .sender (SW[7:6]),
    .in0 (SW[0]),
    .in1 (SW[1]),
    .in2 (SW[2]),
    .in3 (SW[3]),
    .receiver (SW[5:4]),
    .out0 (LEDG[0]),
    .out1 (LEDG[1]),
    .out2 (LEDG[2]),
    .out3 (LEDG[3]),
);
hexdigit h1 (
    .in ({2'h0, SW[7:6]}),
    .out (HEX1)
);
hexdigit h0 (
    .in ({2'h0, SW[5:4]}),
    .out (HEX0)
);
endmodule

```

## 6.12 Event-Driven Simulation: `always`, `assign`, `initial`

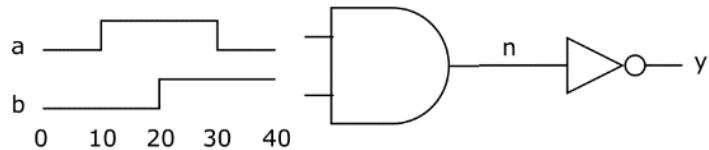
Verilog uses an *event-driven* execution and simulation model that is very different from the execution model that is common in programming languages such as C. In C, programs are organized into functions, and execution proceeds when one function calls another. In Verilog, execution is organized into a set of processes that are all running in parallel that respond to triggering *events* posted on a shared, time-sorted *event list*. Processes respond to triggering events by posting new events to the list.

In Verilog, `always`, `assign`, and `initial` statements are three different ways of declaring processes:

- `always` statements explicitly state which events triggers the process to evaluate through its sensitivity list. An `always` block expresses *cyclic* behavior. When a signal in the sensitivity list changes, the statements in the body execute to the end and then the block cycles back to the top to wait for the next change to a signal in the sensitivity list.
- `assign` statements don't have sensitivity lists, but are implicitly sensitive to changes in any signal that appears on the right-hand side of the “`=`”. They exhibit *continuous* behavior.
- `initial` statements aren't sensitive to triggering events (and hence don't have sensitivity lists), but can generate events that trigger other processes. They automatically run once, starting at the beginning of a simulation.

### 6.12.1 Step-by-Step Example

To illustrate, let's examine the evaluation of a Verilog model for the system below:



We can write a Verilog model in a single testbench module that uses an `initial` block to generate signals `a` and `b`, an `assign` statement to model the AND gate, and an `always` statement to model the inverter. (Note that we wouldn't normally put the system implementation and the signal generation into the same module—this is just for illustration purposes).

```
`timescale 1ns/1ns
module and_inv_tb( );
    reg a;
    reg b;
    reg y;
    wire n;

    assign n = a & b;

    always @(*)
        y = ~n;

    initial begin
        a = 0; b = 0;
        #10 a = 1;
        #10 b = 1;
    end
endmodule
```

```

#10 a = 0;
#10;
end

endmodule

```

At the beginning of the simulation, the initial block posts 2 events to the event list (the value “x” means “unknown”):

time (ns)	process	signal	transition
0	initial	a	x → 0
0	initial	b	x → 0

The transition on signals **a** and **b** causes the **assign** statement modeling the AND gate to fire and post a transition on signal **n** to the event list:

time (ns)	process	signal	transition
0	initial	a	x → 0
0	initial	b	x → 0
0	assign (and)	n	x → 0

This in turn causes the **always** statement modeling the inverter to fire and post a transition on signal **y** to the list:

time (ns)	process	signal	transition
0	initial	a	x → 0
0	initial	b	x → 0
0	assign (and)	n	x → 0
0	always (inv)	y	x → 1

There is no additional activity at 0 ns, so the simulation advances. After a delay of 10 ns, the **initial** block posts a transition on signal **a**:

time (ns)	process	signal	transition
0	initial	a	x → 0
0	initial	b	x → 0
0	assign (and)	n	x → 0
0	always (inv)	y	x → 1
10	initial	a	0 → 1

This causes the assign statement modeling the AND gate to fire, but it doesn’t post any transitions to the event list, because the output of the AND gate, signal **n**, stays at 0.

This simulation advances, and after a delay of 10 ns, the **initial** block posts a transition on signal **b**:

time (ns)	process	signal	transition
0	initial	a	x → 0
0	initial	b	x → 0
0	assign (and)	n	x → 0

0	<code>always (inv)</code>	y	$x \rightarrow 1$
10	<code>initial</code>	a	$0 \rightarrow 1$
20	<code>initial</code>	b	$0 \rightarrow 1$

The transition on b causes the `assign` statement modeling the AND gate to fire and post a transition on signal n:

time (ns)	process	signal	transition
0	<code>initial</code>	a	$x \rightarrow 0$
0	<code>initial</code>	b	$x \rightarrow 0$
0	<code>assign (and)</code>	n	$x \rightarrow 0$
0	<code>always (inv)</code>	y	$x \rightarrow 1$
10	<code>initial</code>	a	$0 \rightarrow 1$
20	<code>initial</code>	b	$0 \rightarrow 1$
20	<code>assign (and)</code>	n	$0 \rightarrow 1$

The transition on n causes the `always` statement modeling the inverter to fire and post a transition on signal y:

time (ns)	process	signal	transition
0	<code>initial</code>	a	$x \rightarrow 0$
0	<code>initial</code>	b	$x \rightarrow 0$
0	<code>assign (and)</code>	n	$x \rightarrow 0$
0	<code>always (inv)</code>	y	$x \rightarrow 1$
10	<code>initial</code>	a	$0 \rightarrow 1$
20	<code>initial</code>	b	$0 \rightarrow 1$
20	<code>assign (and)</code>	n	$0 \rightarrow 1$
20	<code>always (inv)</code>	y	$1 \rightarrow 0$

There is no further activity at 20 ns, so the simulation advances to the next event, when the `initial` block posts a transition on signal a:

time (ns)	process	signal	transition
0	<code>initial</code>	a	$x \rightarrow 0$
0	<code>initial</code>	b	$x \rightarrow 0$
0	<code>assign (and)</code>	n	$x \rightarrow 0$
0	<code>always (inv)</code>	y	$x \rightarrow 1$
10	<code>initial</code>	a	$0 \rightarrow 1$
20	<code>initial</code>	b	$0 \rightarrow 1$
20	<code>assign (and)</code>	n	$0 \rightarrow 1$
20	<code>always (inv)</code>	y	$1 \rightarrow 0$
30	<code>initial</code>	a	$1 \rightarrow 0$

The transition on signal a causes the `assign` statement modeling the AND gate to fire and post a transition on signal n:

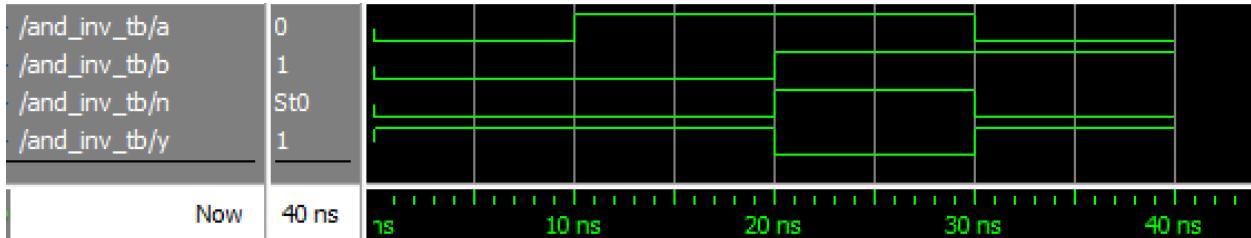
time (ns)	process	signal	transition
0	<code>initial</code>	a	$x \rightarrow 0$

0	initial	b	$x \rightarrow 0$
0	assign (and)	n	$x \rightarrow 0$
0	always (inv)	y	$x \rightarrow 1$
10	initial	a	$0 \rightarrow 1$
20	initial	b	$0 \rightarrow 1$
20	assign (and)	n	$0 \rightarrow 1$
20	always (inv)	y	$1 \rightarrow 0$
30	initial	a	$1 \rightarrow 0$
30	assign (and)	n	$1 \rightarrow 0$

The transition on signal n causes the `always` statement modeling the inverter to fire and post a transition on signal y:

time (ns)	process	signal	transition
0	initial	a	$x \rightarrow 0$
0	initial	b	$x \rightarrow 0$
0	assign (and)	n	$x \rightarrow 0$
0	always (inv)	y	$x \rightarrow 1$
10	initial	a	$0 \rightarrow 1$
20	initial	b	$0 \rightarrow 1$
20	assign (and)	n	$0 \rightarrow 1$
20	always (inv)	y	$1 \rightarrow 0$
30	initial	a	$1 \rightarrow 0$
30	assign (and)	n	$1 \rightarrow 0$
30	always (inv)	y	$0 \rightarrow 1$

The initial block finishes and there are no more events scheduled, so the simulation finishes. The waveform plot shows all the transitions:



### 6.12.2 Modeling Circuit Delay

In the previous example, we saw how transitions on signals had a rippling effect, causing the `assign` and `always` processes to fire, which in turn posted more transitions on the event list. While the `assign` and `always` processes fired in the proper order, as dictated by circuit connections through `reg` and `wire` signals, there was zero propagation delay through the AND gate and the inverter.

At this point, you may be wondering how gate-level simulation models propagation delay through the circuit. The answer is that the delay operator “#” that we used in the `initial` block can also be used within `assign` and `always` statements to delay execution. During the timing analysis and netlist writer steps, the compiler inserts delays into the model, based on the placement and routing of the synthesized design. Suppose, for example that the delay through the AND gate was determined to be 2 ns and the

delay through the inverter was determined to be 1 ns. These delays could be added to the Verilog model as follows:

```

`timescale 1ns/1ns
module and_inv_delay_tb ( );
    reg    a;
    reg    b;
    reg    y;
    wire   n;

    assign #2 n = a & b;

    always @(*)
        #1 y = ~n;

    initial begin
        a = 0;  b = 0;
        #10 a = 1;
        #10 b = 1;
        #10 a = 0;
        #10;
    end

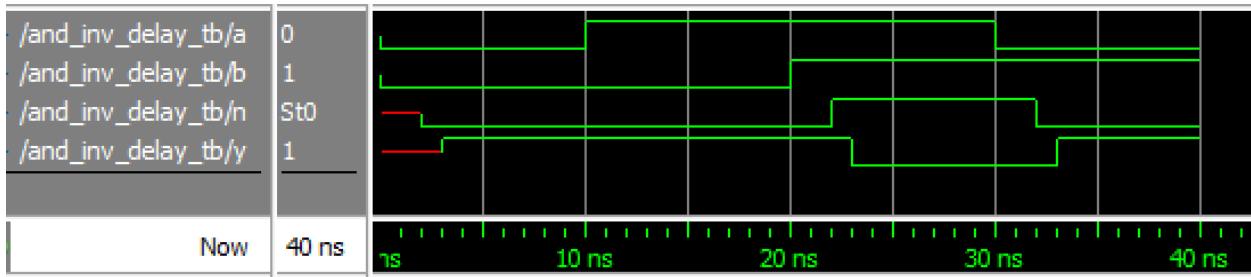
endmodule

```

The addition of the delays into the `assign` and `always` statements doesn't change when these processes are *triggered*, but it does *change when they evaluate and post transitions to the event list*. For example, the `assign` statement that models the AND gate will still trigger when either signals `a` or `b` change value, but it will wait for 2 ns before assigning a value to `n`. Similarly, the `always` block that models the inverter will still fire when `n` changes but will wait for 1 ns before changing the value of `y`. This changes posting to the event list as follows:

time (ns)	process	signal	transition
0	initial	a	x → 0
0	initial	b	x → 0
2	assign (and)	n	x → 0
3	always (inv)	y	x → 1
10	initial	a	0 → 1
20	initial	b	0 → 1
22	assign (and)	n	0 → 1
23	always (inv)	y	1 → 0
30	initial	a	1 → 0
32	assign (and)	n	1 → 0
33	always (inv)	y	0 → 1

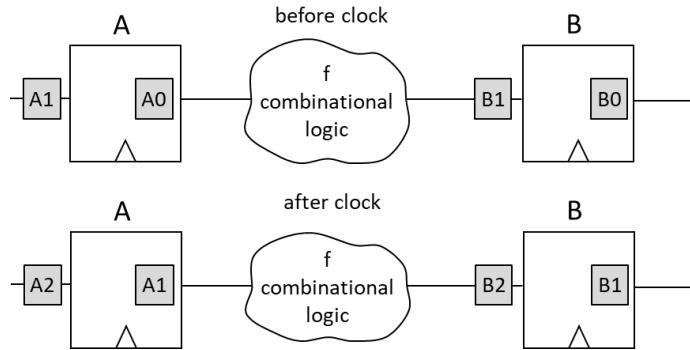
The resulting waveforms are shown below:



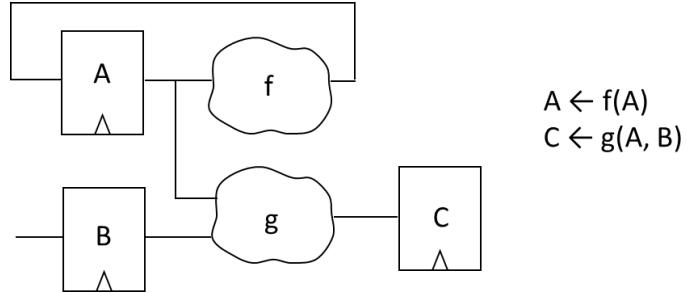
## 7 Sequential Logic

Whereas combinational logic involves evaluating arithmetic and logic expressions, sequential logic is logic that involves storing values between operations. The outputs of a combinational logic circuit change immediately (in reality after a brief propagation delay) after the inputs to the circuit change. By contrast, we will always synchronize the changes to the outputs of a sequential circuit to a clock. To see the difference between a combinational and sequential circuit, consider the difference between an adder and a counter. An adder, which is a combinational logic circuit, takes two values and evaluates their sum. As soon as one of the inputs to the adder changes, the output of the adder follows. A counter, on the other hand, maintains an internally-stored running count. The counter only increments at the edge of a clock.

The figure below illustrates the basic structure and operation of a simple sequential circuit. The circuit has two clocked storage elements called flip flops or registers with some combinational logic between them. Before the clock, register A contains value  $A_0$  and register B contains value  $B_0$ , which appear at the outputs of the registers. Value  $A_0$  is transformed by the combinational logic  $f$  to produce value  $B_1$  that is sitting at the input of register B and value  $A_1$  is sitting at the input of register A. At the rising edge of the clock,  $A_1$  is written into register A and  $B_1$  is written into register B, and new values  $A_2$  and  $B_2$  are produced, to be written at next rising clock edge. Thus values in the system go through a sequence of transitions, all synchronized to the clock.



The type of operations performed by the sequential circuits are called *register-transfer operations*. The circuit above performs the register-transfer operation  $B \leftarrow f(A)$ , where A is the *source register* for operation and B is the *destination*. More generally, sequential circuits can support multiple register transfer operations that incorporate feedback loops and combinations of values from multiple registers as shown below. As we will see the main consideration when modeling sequential circuits in Verilog is to state what register-transfer operations happen at the rising edge of the clock.

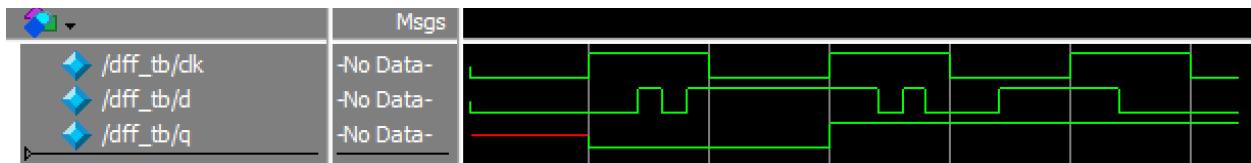
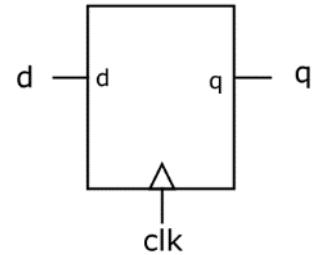


This chapter begins with how we model the basic storage element, the flip flop, in Verilog and proceeds to examine how to add features to a flip flop, and then how to model other common sequential components such as counters, timers, and more.

## 7.1 D Flip Flop

A D flip flop (DFF) is the most basic edge-triggered, sequential logic storage element. A flip flop has an input data port  $d$ , and output data port  $q$ , and an input clock  $clk$ . On the symbol for a flip flop, an edge-triggered clock is usually represented by a triangle.

On the rising edge of  $clk$ , the flip flop stores the input  $d$  and it appears on the output  $q$ . Data storage only happens on the rising clock edge; in other words, if the data on the input  $d$  is changing, the value of  $q$  won't change as long as there is no rising clock edge.



### 7.1.1 Modeling Edge-Triggered Behavior

The key behavior of a flip flop—storing the data on the input port  $d$  and having it appear at the output port  $q$ —only happens on a rising edge of the clock  $clk$ . Thus the process of assigning the value of  $d$  to  $q$  is only sensitive to the positive edge of the clock. This is unlike the models of the combinational logic circuits that we've seen thus far, which were sensitive to changes in the values of any of the input. The fact that a flip flop is sensitive to the positive edge of  $clk$  but is *not* sensitive to changes in  $d$  is what give it its distinctive storage behavior.

Below is a Verilog model for a D flip flop, `my_dff`. The first thing to notice is that the sensitivity list contains only the clock signal  $clk$ , with the qualifier keyword `posedge`. This means that the body of the `always` block fires when there is a 0 to 1 transition  $clk$ . (If we wanted the flip flop to store data on the negative or 1 to 0 transitions of the clock, we could use the qualifier `negedge`). Importantly, the data input  $d$  is not part of the sensitivity list; otherwise, a new value would be assigned to  $q$  whenever the value of  $d$  changes regardless of the clock, which is not the behavior that we want.

```
module my_dff (
    input d,
    input clk,
    output reg q
```

```

);
always @(posedge clk)
q <= d;
endmodule

```

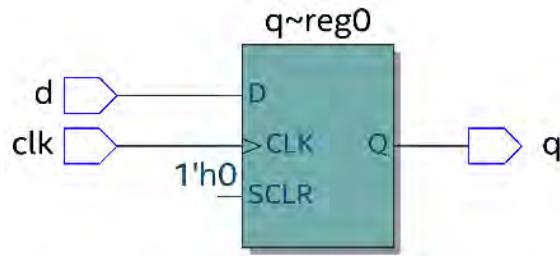
The next important thing to notice is the use of the *non-blocking signal assignment operator*, `<=`. **This is not a less-than-or-equal-to operator, which unfortunately uses the same symbol.** (The compiler will interpret the meaning of this symbol, based on context). Think of this symbol as representing an arrow that denotes an assignment that occurs immediately after the rising clock edge. We'll go into detail about the differences between the blocking assignment operator “`=`” that we've seen before and the non-blocking operator “`<=`”, but for now we'll just keep a simple rule in mind and worry about understanding why later:

assignment type:	blocking	non-blocking
syntax:	<code>y = a</code>	<code>q &lt;= d</code>
usage:	combinational logic only	sequential (clocked) only

Finally, the output port `q` is of type `reg`, because it is assigned a value from within an `always` block.

### 7.1.2 Synthesized RTL Netlist

The Verilog model synthesizes as an edge-triggered D flip-flop in the RTL netlist. The library component has a synchronous clear input, `SCLR`, which is hardwired to 0 (we'll talk about clear inputs soon).

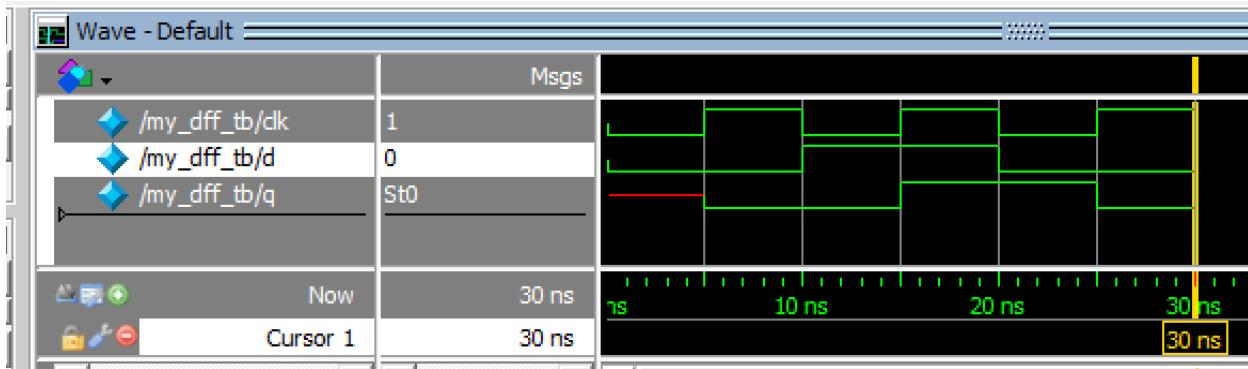


### 7.1.3 Testbench with Clock

In order to simulate a sequential circuit like a flip flop, we need to model a clock and data inputs in the testbench. There are a number of styles for doing this, but here's a recommended approach.

- set up a clock with an initial value of 0 that flips its value every  $n$  time units. Thus there will be a rising clock edge at every odd multiple of  $n$  time units.
- change data inputs at the midpoints between each rising clock edge. Thus if you initialize the data inputs at time zero and then change them every  $2n$  time units, the data input transitions will be at the even multiples of  $n$  time units.

The following input and output waveforms for a D flip flop illustrate this behavior for  $n = 5$  ns, where the rising edges on clock signal `clk` occur at 5 ns, 15 ns, and 25 ns. The data on input data signal `d` changes at 10 ns and 20 ns. The output data `q` changes on the rising edges of the clock.



The following testbench for a D flip flop illustrates a way to model both the clock and data inputs. The testbench contains two processes, an `always` block that generates the clock and an `initial` block that generates input data stimuli. Note that the initial value for the clock is defined at the beginning of the `initial` block. The `$stop` directive is necessary to make sure that the simulation ends after the last data input; without it, the `always` block for the clock would cause the simulation to never end.

```
`timescale 1ns/1ns
module my_dff_tb ();
    reg d;
    reg clk;
    wire q;

    my_dff uut (
        .d(d),
        .clk(clk),
        .q(q)
    );

    always
        #5 clk = ~clk;

    initial begin
        clk = 0; d = 0;
        #10 d = 1;
        #10 d = 0;
        #10;
        $stop;
    end

endmodule
```

## 7.2 D Flip Flop with Reset

A common feature added to a flip flop is a reset signal that will set the flip flop to a known state, typically 0. Resets can be *synchronous* with the clock, that is, occur on the rising clock edge, or *asynchronous*, that is, occur as soon as the reset signal is applied. For this class, we'll only use synchronous resets.

The following Verilog module illustrates a DFF with synchronous reset. Note that the reset signal `rst` is not part of the sensitivity list—this is what makes it synchronous.

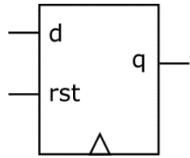
```

module dff_RST (
    input clk,
    input d,
    input rst,
    output reg q);

    always @(posedge clk) begin
        if (rst)
            q <= 0;
        else
            q <= d;
    end

endmodule

```



Here is a testbench showing the use of the reset signal and the resulting waveforms. Note how `q` resets to 0 on the first rising clock edge after `rst` is asserted to 1.

```

module dff_RST_tb ();
    reg clk;
    reg d;
    reg rst;
    wire q;

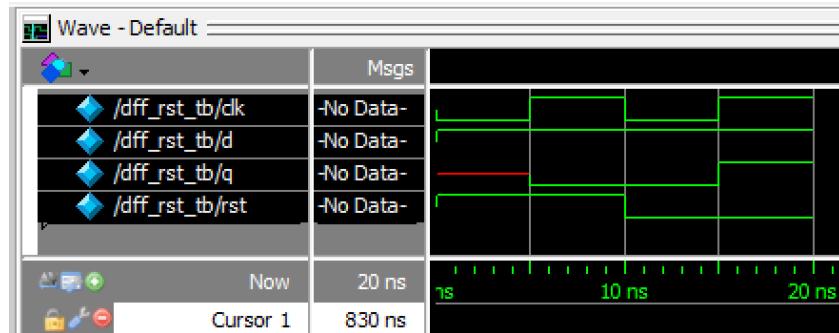
    dff_RST uut (
        .clk  (clk),
        .d    (d),
        .rst  (rst),
        .q    (q)
    );

    always
        #5 clk = ~clk;

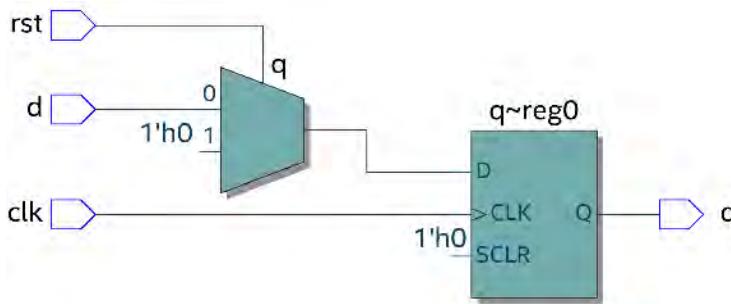
    initial begin
        clk = 0; d = 1;  rst = 1;
        #10 rst = 0;
        #10;
        $stop;
    end

endmodule

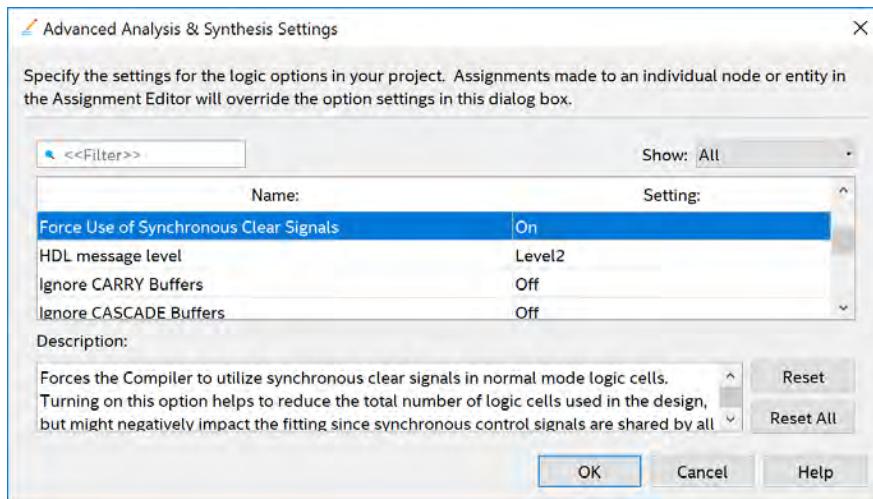
```



The synthesized result is curious. Even though the Quartus RTL library of abstract components has a register with a synchronous clear, the RTL netlist for the synthesized results uses a multiplexor to implement the clear logic, rather than using the existing pin.

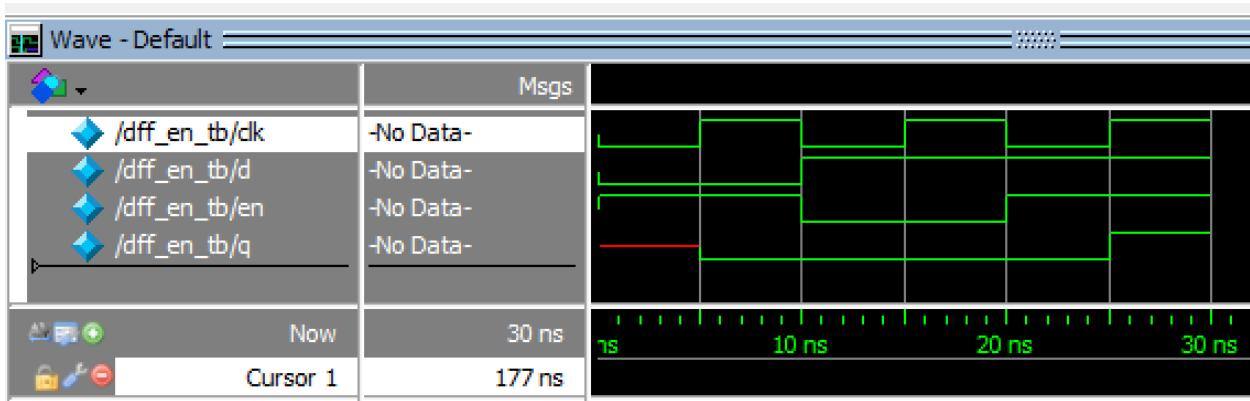
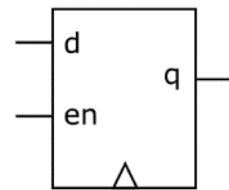


On the other hand, during placement and routing, if the option “Force Use of Synchronous Clear Signals” is enabled, then the place and route will take advantage of the synchronous clear input on the register inside of the logic element. In general, we won’t worry about this.



### 7.3 D Flip Flop with Load Enable

Another common feature is a load enable signal that inhibits new data from being loaded into the flip flop unless the enable signal is asserted. This behavior is illustrated in the waveforms below, where the output q only change on a rising clock edge if the enable signal en is a 1.



This behavior is easily modeled in Verilog by using an `if` statement conditional on the load enable signal to determine whether or not to assign the value of `d` to `q`.

```
module dff_en (
    input clk,
    input d,
    input en,
    output reg q);

    always @(posedge clk)
        if (en)
            q <= d;

endmodule
```

Here's the testbench that generated the above waveforms:

```
`timescale 1ns/1ns

module dff_en_tb ();
    reg clk;
    reg d;
    reg en;
    wire q;

    dff_en uut (
        .clk  (clk),
        .d    (d),
        .en   (en),
        .q    (q)
```

```

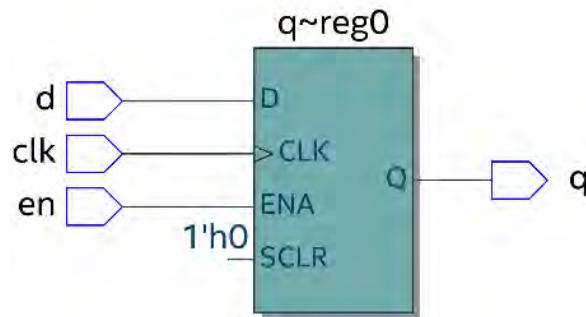
);
always
#5 clk = ~clk;

initial begin
clk = 0; d = 0; en = 1;
#10 d = 1; en = 0;
#10 en = 1;
#10
$stop;
end

endmodule

```

The synthesized netlist takes advantage of the enable pin on the RTL library register component.



## 7.4 Power Up Values

In general, when flip flop hardware is first powered up, it may enter a state where it either stores a 1 or a 0. The simulated waveforms in the previous section shows this as an unknown value, represented by a red line. Special hardware may be designed, however, that ensures that flip flops power up to a known state. For example, flip flops in the logic elements of Intel FPGAs are designed to always power up to a low (0) logic level.<sup>9</sup> While the hardware powers up to a zero, the Verilog simulation environment (ModelSim) does not recognize this fact, which is the reason why the simulation shows an unknown value at the flip flop output until a known value is written into it.

It is very important to hardware designers to have flip flops power up into a known state. In fact, without having this feature built into the flip flop circuitry, designers would have to use separate reset logic and then assert the reset signal to force the system into a known state. Sometimes, however, it would be useful to have flip flops be able to power up to a high value (1) rather than low value (0). The physical hardware inside the flip flops in an Intel FPGA can't accommodate that, but it is possible to "fake it" by adding inverters to the flip flop input and output.

The Intel Quartus supports specifying power up values for flip flops by initializing `reg` variables using an initial statement as shown below.

```
module dff_initialized (
```

---

<sup>9</sup> Intel Quartus Prime Standard Edition Handbook Volume 1, for Design Suite 17.1, QPS5V1 | 2017.10.06, pg. 840.

```

input clk,
input d,
output reg q;

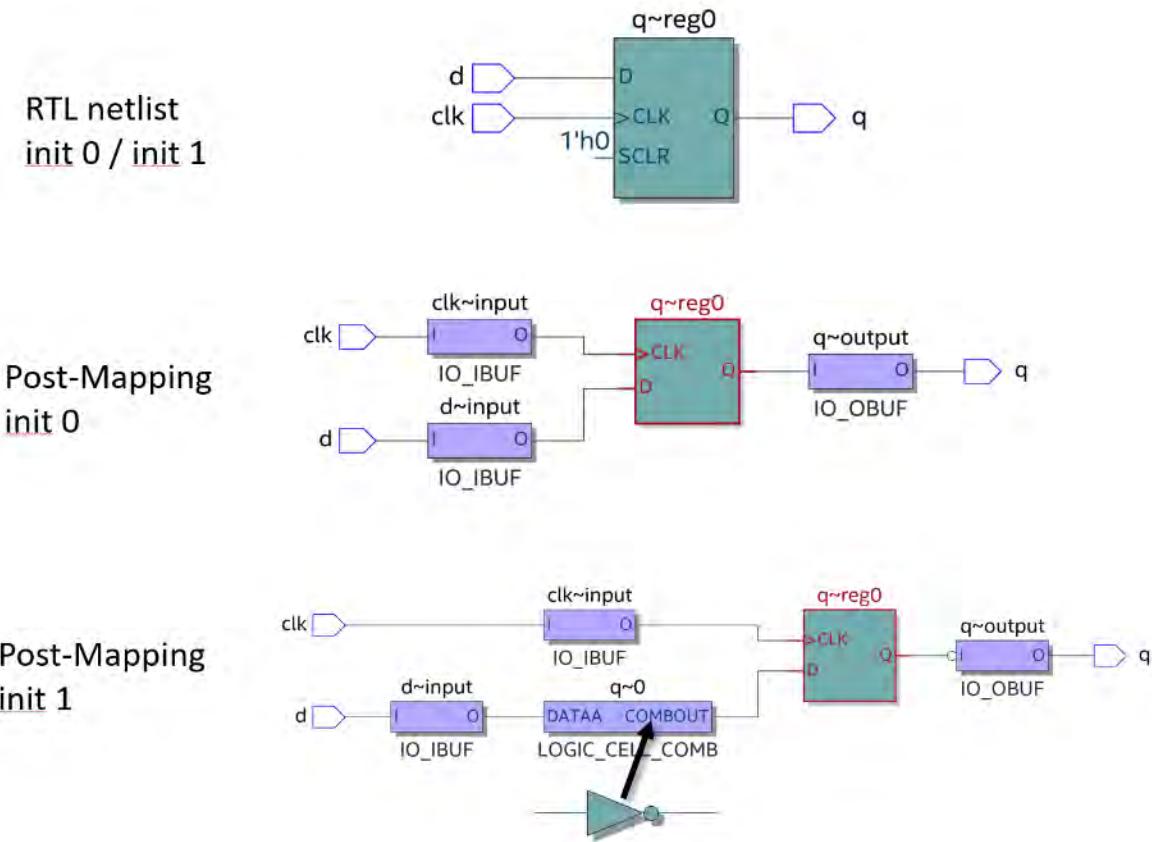
initial q = 1'b1; // or 1'b0

always @(posedge clk)
q <= d;

endmodule

```

Initializing the `reg` variable causes two important things to happen. From the simulation perspective, the simulation will show a known value for the flip flop from the start of the simulation. From the synthesis perspective, the synthesis tools will add the necessary inverters to flip flop inputs and outputs to “fake” an initial value of 1. These inverters won’t show up in the RTL netlist, but they are visible in the technology mapped netlist after assigning the flip flop to specific logic elements in the FPGA fabric. In the figure below, the inverter on the input is included in combinational logic inserted before the flip flop, and the inverter after shows as a bubble on the input of an input/output buffer (IO\_BUF) after the flip flop.



## 7.5 Multibit Registers

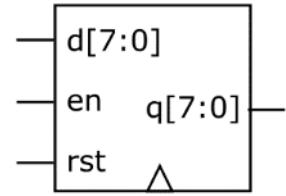
A register is like a flip flop except that it can store arrays of bits. (Physically, a register may be implemented as an array of flip flops). The Verilog model for a register is much like the model of a flip

flop, except that it uses multibit vectors for its input and output data. Here is Verilog model for an 8-bit register with reset and enable inputs.

```
module reg8_en_rst (
    input      clk,
    input [7:0] d,
    input      en,
    input      rst,
    output reg [7:0] q);

    always @ (posedge clk)
        if (rst)
            q <= 0;
        else if (en)
            q <= d;

endmodule
```



A testbench and simulated waveforms are below:

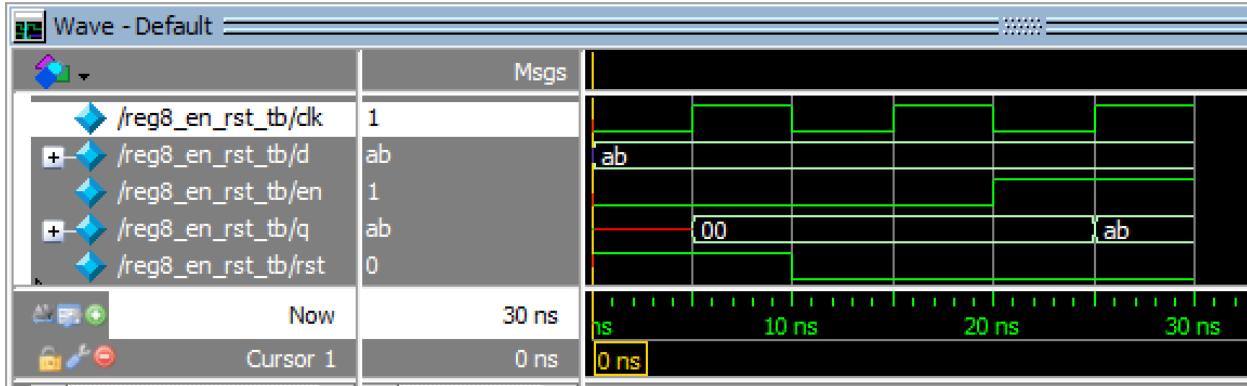
```
`timescale 1ns/1ns
module reg8_en_rst_tb ();
    reg      clk;
    reg [7:0] d;
    reg      en;
    reg      rst;
    wire [7:0] q;

    reg8_en_rst uut (
        .clk  (clk),
        .d    (d),
        .en   (en),
        .rst  (rst),
        .q    (q)
    );

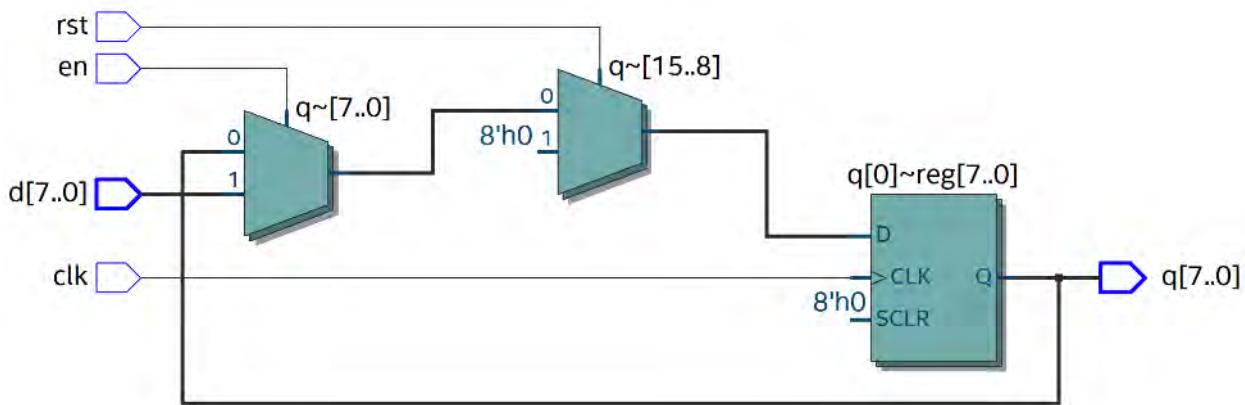
    always #5 clk = ~clk;

    initial begin
        clk = 0; en = 0; rst = 1; d = 8'hab;
        #10 rst = 0;
        #10 en = 1;
        #10;
        $stop;
    end

endmodule
```



This time, synthesis used multiplexors to implement both the reset and enable logic:



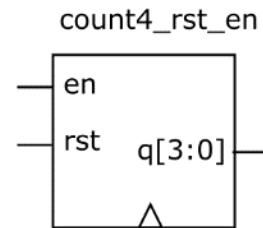
## 7.6 Counters

A counter is a component that stores a value that increments (or decrements) on clock edges. Like registers, counters can be designed with many different features, such as reset, enable, direction control, etc.

### 7.6.1 Incrementing Counter with Reset and Enable

Here is a basic 4-bit incrementing counter with reset and count enable inputs. The counter wraps around to zero after it reaches the maximum value of 15.

```
module count4_RST_EN (
    input clk,
    input rst,
    input en,
    output reg [3:0] q);
    initial q = 0;
    always @(posedge clk)
        if (rst)
            q <= 0;
        else if (en)
```

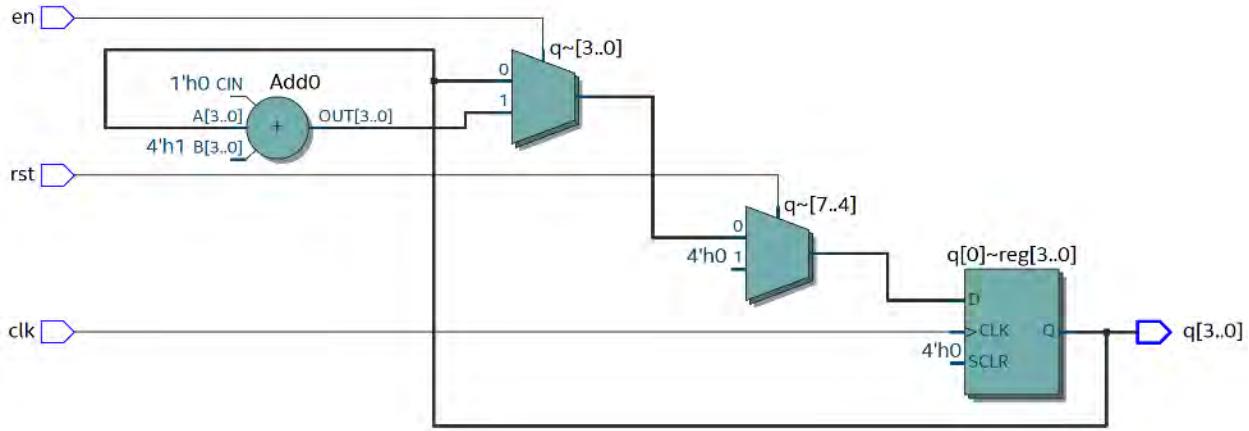


```

q <= q + 1;
endmodule

```

The synthesized result uses an adder and multiplexors to implement the logic, as shown in the RTL netlist:



## 7.6.2 Loadable Decrementing Saturating Counter

A saturating counter is one that stops counting once it reaches a minimum or maximum level. This example is of a counter that can be loaded with some initial value  $d$  using a load enable signal  $ld$  and then counts down to 0 and saturates there until a new value is loaded. A count enable signal  $en$  inhibits counting unless it is asserted as a 1.

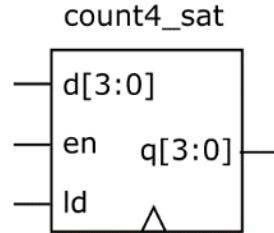
```

module count4_sat (
    input clk,
    input [3:0] d,
    input ld,
    input en,
    output reg [3:0] q);

    initial q = 0;

    always @(posedge clk)
        if (ld)
            q <= d;
        else if (en && q != 0)
            q <= q - 1;
endmodule

```



Testbench and waveforms:

```

`timescale 1ns/1ns
module count4_sat_tb ();
    reg clk;
    reg [3:0] d;

```

```

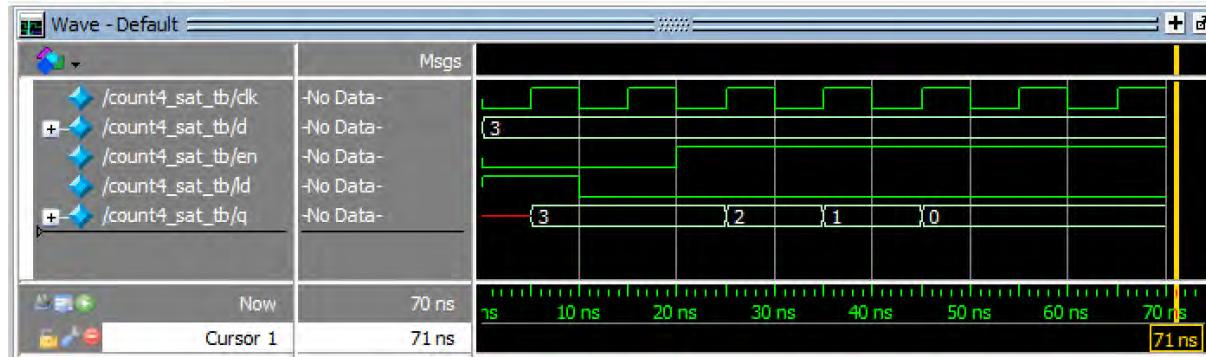
reg ld;
reg en;
wire [3:0] q;

count4_sat uut (
    .clk  (clk),
    .d    (d),
    .ld   (ld),
    .en   (en),
    .q    (q)
);

always #5 clk = ~clk;

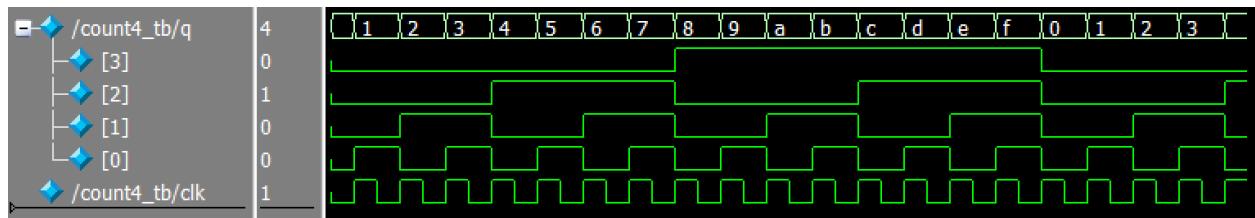
initial begin
    clk = 0;  ld = 1;  d = 4'd3;  en = 0;
    #10 ld = 0;
    #10 en = 1;
    #50 $stop;
end
endmodule

```



## 7.7 Frequency Division by a Power of 2

Sometimes we have a clock signal that is available at a certain frequency, but we need a clock at a lower frequency. The process of generating a lower frequency clock from a higher frequency clock is called *clock division*. Dividing the clock by a power of 2 is easily accomplished with a counter. Consider the output of a 4 bit counter:



The least-significant bit of the counter changes at 1/2 the frequency of the clock, the 2<sup>nd</sup> least-significant bit changes at 1/4 the frequency of the clock, the 3<sup>rd</sup> least-significant bit at 1/8, and so on. Thus we can

divide the clock frequency by  $2^n$  by tapping the most-significant bit of an n-bit counter. The following Verilog module uses a 2-bit counter to divide the clock frequency by a factor of 4.

```
module freq_div4 (
    input clkin,
    output clkout
);

reg [1:0] count = 0;

assign clkout = count[1];

always @(posedge clkin)
    count <= count + 1;

endmodule
```

## 7.8 Timers

Timers are used to produce a pulse after a certain number of clock ticks. Timers are often used as part of a real-time clock, for example, to produce a pulse once every second. Inside, a timer is similar to a counter, except rather than outputting the current value of the count, it instead indicates when the count has reached a certain limit. The following example is a timer that produces a pulse one cycle wide when the count reaches 5. It uses two separate processes to model this behavior: an `always` block for the counter and an `assign` statement to test the limit.

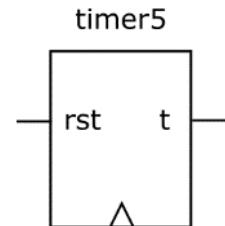
```
module timer5 (
    input clk,
    input rst,
    output t);

reg [2:0] count = 0;

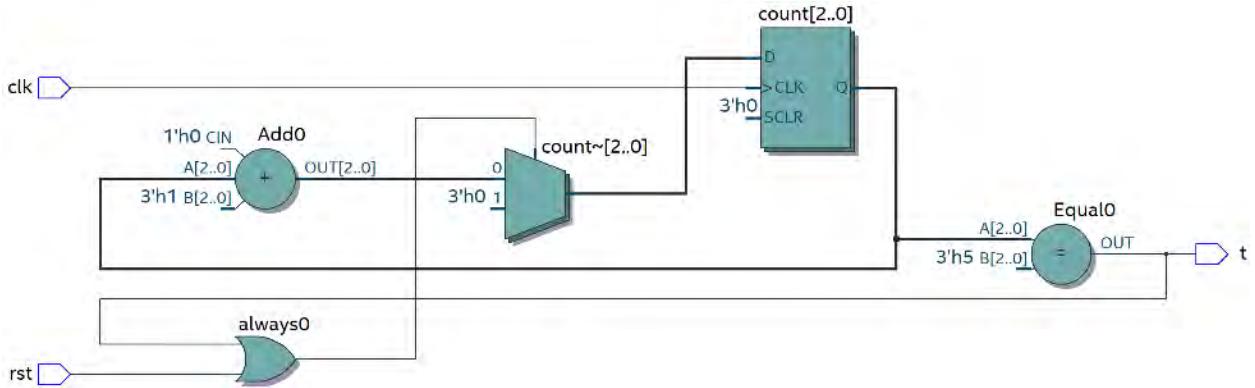
assign t = (count == 3'd5);

always @(posedge clk) begin
    if (rst | t)
        count <= 0;
    else
        count <= count + 1;
end

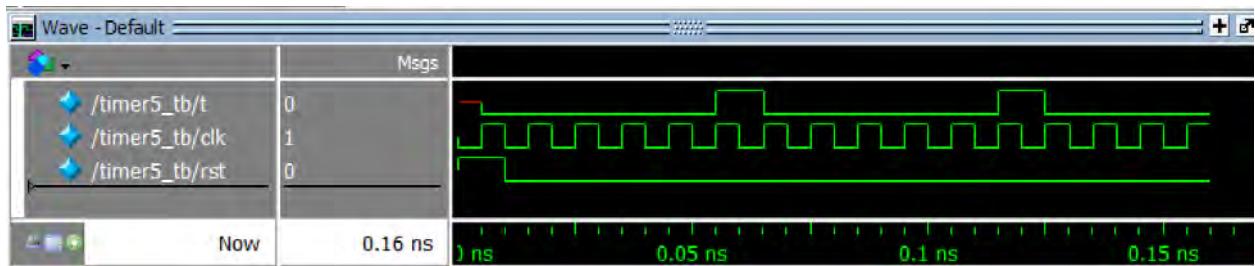
endmodule
```



The synthesized result shows the counter functionality, followed by the combinational logic from the `assign` statement at the counter register output:

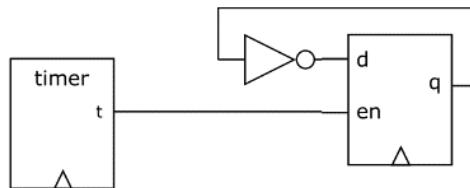


The simulated waveforms are shown below.



## 7.9 Square Wave Generator

A timer generates a train of pulses at a certain frequency, where that frequency is determined by the limit of the counter inside. The shape of the waveform generated by a timer is very asymmetric, where it is only high for one clock cycle and low for the rest of each period. For some application, it is preferable to produce a symmetric square wave that high for half of the period and low for the other half. A simple way to generate a square wave of a certain frequency is to use a timer to enable a flip flop that toggles its output.



Here's a Verilog model that combines the complete functionality of a square wave generator into a single module. It uses separate `always` blocks for the count register inside of the timer and the toggling flip flop. It also uses the Verilog `parameter` statement to define the counter limit, which is half of the period of the generate square wave. A `parameter` isn't part of the hardware like a `reg` or `wire`; rather it's simply a way of defining a constant used by the module that helps make the code more readable.

```

module square_wave_generator (
    input clk,
    output reg q);

```

```

parameter HALF_PERIOD = 3'd5;

initial q = 0;
reg [3:0] count = 0;

// limit check logic
wire at_limit = (count == HALF_PERIOD);

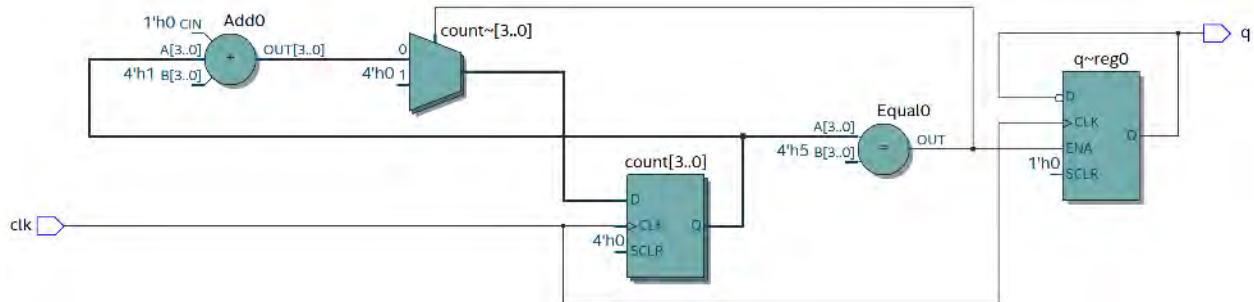
// counter process
always @(posedge clk)
  if (at_limit)
    count <= 0;
  else
    count <= count + 1;

// toggling flip flop process
always @(posedge clk)
  if (at_limit)
    q <= ~q;

endmodule

```

The complete synthesized model is shown below:



An alternative Verilog implementation combines all of the logic into a single `always` block—it will synthesize into the same RTL netlist:

```

module square_wave_generator_v2 (
  input clk,
  output reg q
);

parameter HALF_PERIOD = 3'd5;

initial q = 0;
reg [3:0] count = 0;

always @(posedge clk) begin
  if (count == HALF_PERIOD) begin

```

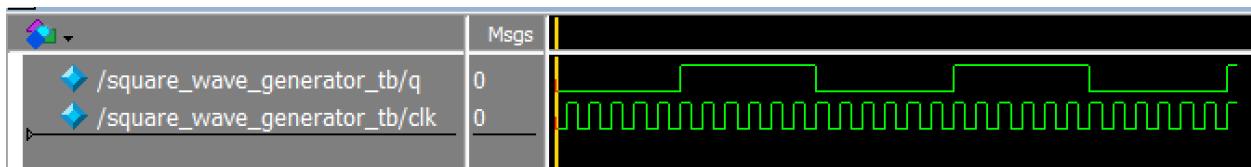
```

        count <= 0;
        q <= ~q;
    end
    else begin
        count <= count + 1;
    end
end

endmodule

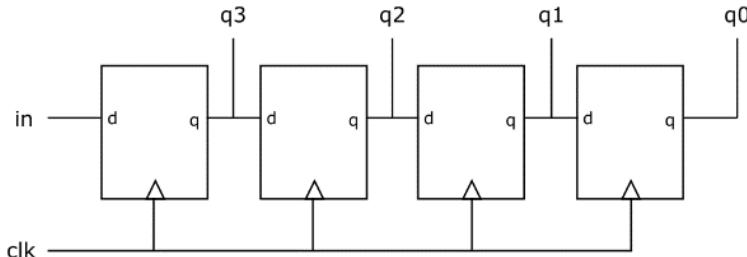
```

Here's the simulated output:



## 7.10 Shift Register

A shift register is a series of flip flops connected in a chain. A typical application of a shift register is converting a stream of serial data to parallel outputs. The following shift register would take a 4-bit serial packet arriving least-significant bit first through port `in` and convert it to 4-bit parallel data output `q[3:0]`. At each rising clock edge, the shift register receives a new most-significant bit and shifts the old bits to the right, dropping off the old least-significant bit.



The standard way of modeling the behavior of a shift register in Verilog is using the concatenation operator `{ }`.

```

module shiftreg4_msb_in (
    input clk,
    input in,
    output reg [3:0] q
);

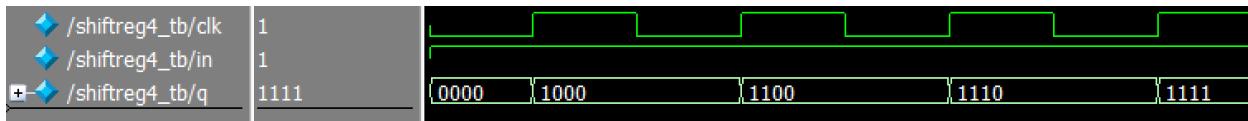
initial q = 0;

always @(posedge clk)
    q <= {in, q[3:1]};

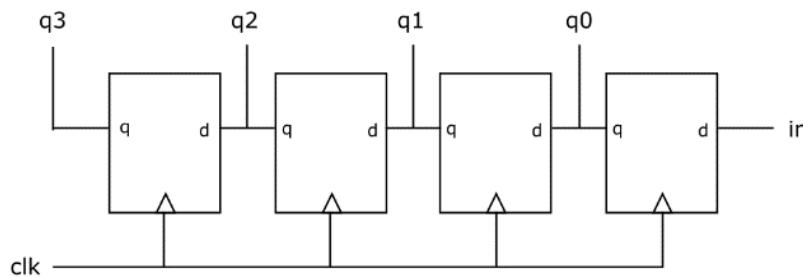
endmodule

```

Here's the results of simulating shifting in 1s:



A shift register could also shift bits in from the least significant bit, dropping off the most-significant bit (note that d is on the right side of the flip flop symbol and q is on the left):

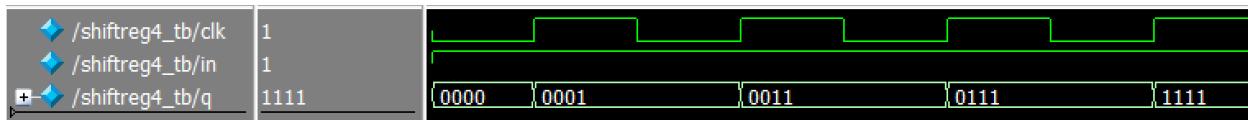


```
module shiftreg4_lsb_in (
    input clk,
    input in,
    output reg [3:0] q
);

initial q = 0;

always @(posedge clk)
    q <= {q[2:0], in};

endmodule
```



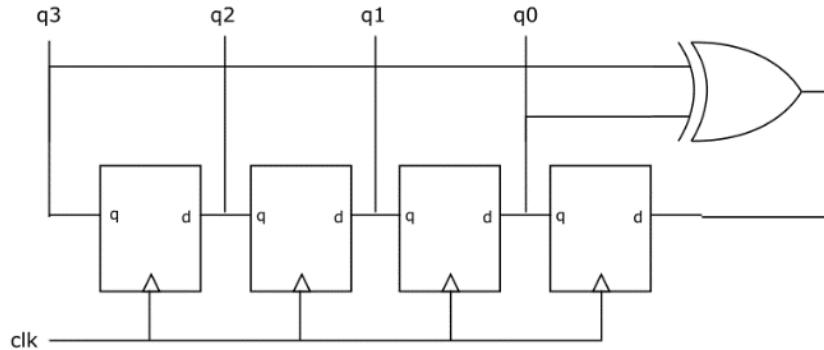
## 7.11 Pseudo-Random Number Generation Using a Linear-Feedback Shift Register

Many digital hardware systems, including games, cryptography, and more, require random numbers. Upon determining that they need a random number generator for their designs, students will often ask, “does Verilog have a random number command,” similar to the ones they’ve seen in programming languages like C. It does, but it’s not synthesizable into hardware. The question should be, “how do you implement a random number generator in hardware?”

There are a number of approaches, but an important one to understand is a component known as a *linear-feedback shift register (LFSR)*. An LFSR is basically a shift register whose input is the XOR combination of a specially-chosen set of bits from the current contents of the shift register. This set of bits, known as “taps” into the shift register, are chosen so that an n-bit LFSR will generate a series of  $2^{(n-1)}$  pseudo-random numbers without repetition. After the end of the series, the pattern repeats (which is why the

sequence is pseudo-random rather than random). The mathematics behind the selection of taps is beyond the scope of this text, but it is based on finite or Galois fields.

Below is an example of a 4-bit LFSR, with bits shifting from the least-significant to most-significant direction. The taps are at bits 3 and 0, and the XOR combination  $q[3] \wedge q[0]$  is fed back into the least-significant bit position of the shift register.



This LFSR generates the following repeating 4-bit unsigned decimal sequence:

1, 3, 7, 15, 14, 13, 10, 5, 11, 6, 12, 9, 2, 4, 8

Note that the sequence is initialized to 1 ( $4'b0001$ ). If it were initialized to 0, the LFSR would never change, since  $0 \wedge 0$  is 0.

A Verilog model for the 4-bit LFSR is:

```
module lfsr4 (
    input clk,
    output reg [3:0] q);

    initial q = 4'd1;

    always @(posedge clk)
        q <= {q[2:0], q[3]^q[0]};

endmodule
```

The following table lists sets of taps for LFSRs up to 32 bits wide.<sup>10</sup>

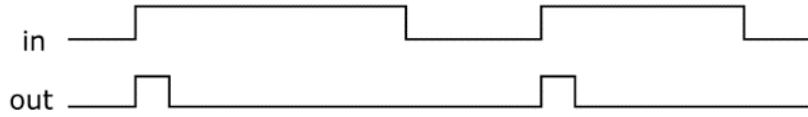
Width	Taps	Width	Taps	Width	Taps	Width	Taps
1	--	9	3,8	17	2, 16	25	2, 24
2	0, 1	10	2, 9	18	8, 17	26	0, 1, 5, 25
3	0, 2	11	1, 10	19	6, 17	27	0, 1, 4, 26
4	0, 3	12	0, 3, 5, 11	20	0, 1, 4, 18	28	2, 27
5	1, 4	13	0, 2, 3, 12	21	2, 19	29	1, 28
6	0, 5	14	0, 2, 4, 13	22	1, 20	30	0, 3, 5, 29

<sup>10</sup> Max Maxfield, "Tutorial: Linear-Feedback Shift Registers Part 1," *EE Times*, Dec 20, 2006, [https://www.eetimes.com/document.asp?doc\\_id=1274550&page\\_number=1](https://www.eetimes.com/document.asp?doc_id=1274550&page_number=1)

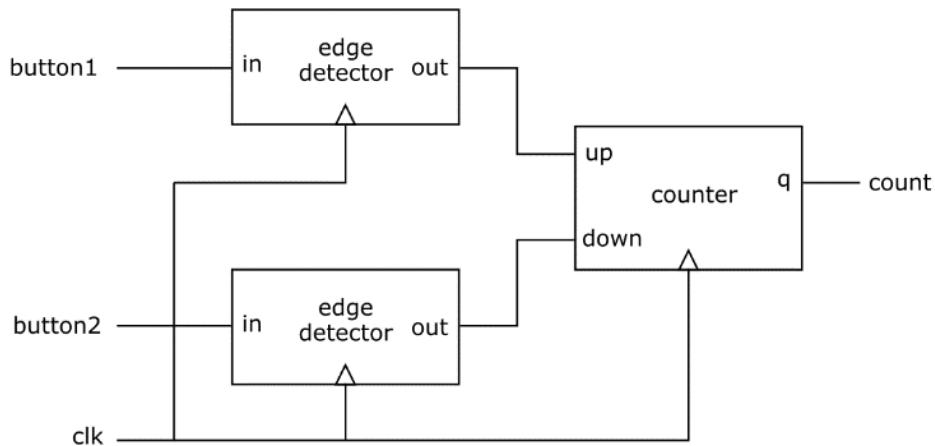
7	0, 6	15	0, 14	23	4, 22	31	2, 30
8	1, 2, 3, 7	16	1, 2, 4, 15	24	0, 2, 3, 23	32	1, 5, 6, 31

## 7.12 Edge Detector

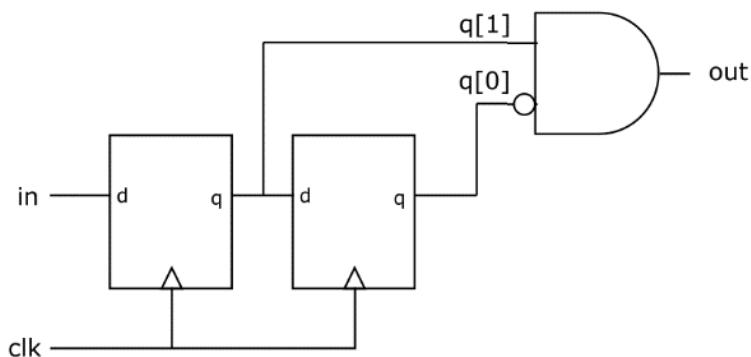
An *edge detector* is a sequential circuit that detects when there is a change in a stream of data from a 0 to a 1 (or from a 1 to a 0) and that produces a pulse one clock cycle wide when the rising (or falling) edge occurs. The following waveforms illustrate the behavior of an edge detector, where *in* is the original input data stream and *out* is the output that detects when a rising edge occurs.



As an application of an edge detector, consider a system with a counter and 2 buttons, such that the counter should increment by 1 when button1 is pressed and then decrement by 1 when button2 is pressed. An initial idea may be to try to write a Verilog module with the buttons as clock inputs to counter and to somehow use combinational logic to determine which button was pressed, but in practice this won't work, since sequential circuits are synthesized into flip flops and combinational logic and flip flops can only have a single clock input. Instead, a clean way to implement the system is to design a counter with a single clock input and separate up and down count enables, and then use edge detectors to convert button presses into pulses exactly one cycle wide, as shown below. (Note that real buttons should also be debounced before connecting them to the edge detectors.)



The standard way of implementing an edge detector is to use a 2-stage shift register with some logic to compare the values in each of the stages as shown below.



The output of the circuit will only be a 1 when  $q[1]$  equals 1 and  $q[0]$  equals 0, that is, when two adjacent data samples transition from 0 to 1.

We leave it as an exercise for the reader to write the Verilog code for the edge detector.

## 8 Verilog Signal Assignment



One of the more confusing aspects of Verilog is how values get assigned to signals. We use the term signal to mean any way of passing bit values between parts of a digital circuit. The language provides a variety of signal assignment mechanisms with subtle differences in behavior that can lead to subtle bugs in your models. Fortunately, there are some simple rules of thumb to follow that can help you avoid getting bitten.

### 8.1 Blocking vs. Non-Blocking Signal Assignment

We've seen earlier that Verilog has two different signal assignment operators, the *blocking* assignment operator (`=`) and the *non-blocking* assignment operator (`<=`). The semantics of the blocking assignment operator are similar to the assignment operator in many procedural programming languages such as C. Given a series of blocking assignments in a `begin-end` block, each one completes before moving on to the next one. The non-blocking assignment operator is something not found in a language like C; it evaluates all of the right-hand-sides before making the assignment, and then makes all of the assignments concurrently. Because we design digital systems so that all state transitions happen synchronously with the edge of the clock, the non-blocking assignment operator (`<=`) is better for modeling the behavior of sequential (clocked) circuits. On the other hand, the blocking assignment operator (`=`) is typically more natural for expressing the behavior of combinational circuits. In this sections, we give examples that illustrate the advantages and pitfalls of blocking and non-blocking assignments in combinational and sequential logic. It can get to be confusing, but it all boils down to a simple rule of thumb:

- Use the blocking assignment operator (`=`) for modeling combinational logic
- Use the non-blocking operator (`<=`) for clocked, sequential logic

Use for Combinational Logic	Use for Sequential (Clocked) Logic
blocking: complete each assignment before moving on to next statement	non-blocking: evaluate each RHS without waiting for assignment, then make all assignments concurrently
<pre>begin   LHS<sub>1</sub> = RHS<sub>1</sub>;   LHS<sub>2</sub> = RHS<sub>2</sub>;   LHS<sub>3</sub> = RHS<sub>3</sub>; end</pre>	<pre>begin   LHS<sub>1</sub> &lt;= RHS<sub>1</sub>;   LHS<sub>2</sub> &lt;= RHS<sub>2</sub>;   LHS<sub>3</sub> &lt;= RHS<sub>3</sub>; end</pre>
evaluate $RHS_1$ , assign to $LHS_1$ , evaluate $RHS_2$ , assign to $LHS_2$ , evaluate $RHS_3$ , assign to $LHS_3$	evaluate $RHS_1$ , evaluate $RHS_2$ , evaluate $RHS_3$ , assign right-hand sides to left-hand-sides

## 8.2 Good: Blocking Assignment Operator (=) and Combinational Logic

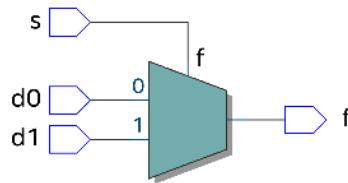
Blocking assignments work well for modeling combinational logic. They are particularly well-suited to situations where the programmer may assign a variable a temporary value that is later overwritten. Consider the following model for a 2-to-1 multiplexor that initially assigns `d0` to the output `f` and then overwrites this with `d1` if `s` is equal to 1.

```
module mux2_overwrite (
    input      d0,
    input      d1,
    input      s,
    output reg f);

    always @(*) begin
        f = d0;
        if (s)
            f = d1;
    end

endmodule
```

This implementation may not be as obvious as using an `if-else` statement, but it still works and will properly synthesize as a multiplexor. Here's the synthesized RTL netlist:



## 8.3 Bad: Blocking Assignment Operator (=) and Clocked Logic

While great for combinational logic, the blocking assignment operator (`=`) is best avoided for clocked, sequential circuits. This is because with the blocking operator, the order of assignments matters, whereas in sequential circuits, we think of all assignments happening concurrently on the edge of the clock.

To illustrate, consider the design of a chain of three registers known as a *shift register*, since values shift down the chain on each rising clock edge.

The following module contains 3 register variables, `Q`, `R1`, and `R2` and uses blocking assignments to describe the behavior of the shift register.

```
module register_block_back_to_front (
    input D, CLK,
    output reg Q);

    reg R1, R2;
```

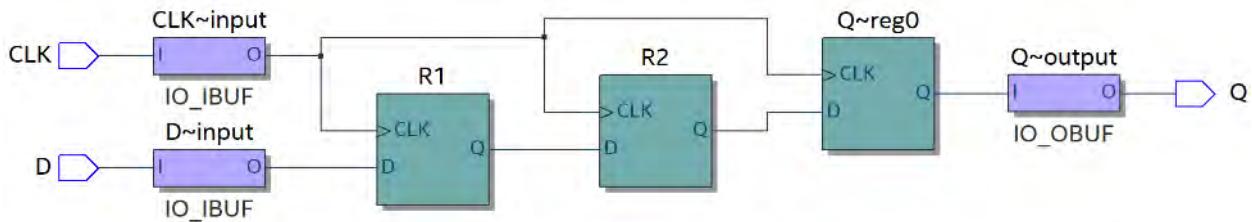
```

always @(posedge CLK)
begin
    Q = R2;
    R2 = R1;
    R1 = D;
end

endmodule

```

On the rising clock edge, Q gets the value of R2, then R2 gets the value of R1, the R1 gets the input value D. This is the behavior that we want, and the module will synthesize as a chain of three registers as expected. Here's the gate-level netlist, after place-and-route, which includes the input and output buffers (the layout of the RTL netlist was correct but a bit strange and harder to understand):



Consider what happens, however, if we reverse the order of the assignments in the `always` block.

```

module register_block_front_to_back (
    input D, CLK,
    output reg Q);

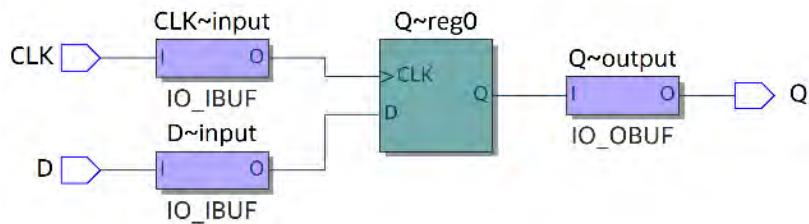
    reg R1, R2;

    always @(posedge CLK)
    begin
        R1 = D;
        R2 = R1;
        Q = R2;
    end

endmodule

```

In this case, R1 first gets the value of D. After that completes, R2 gets the value of R1, which is now the same as D. Finally, Q gets the value of R2, which is also now the same as D. Rather than producing a chain of three registers, the synthesis tool will produce just one register, since on a rising clock edge, the overall behavior is to assign the value of D to Q, and R1 and R2 just act as temporary variables.



This behavior is what we described, but it is not what we intended. The blocking operator is ripe for these kinds of mistakes and is best avoided for sequential logic—the non-blocking assignment operator is a much better choice.

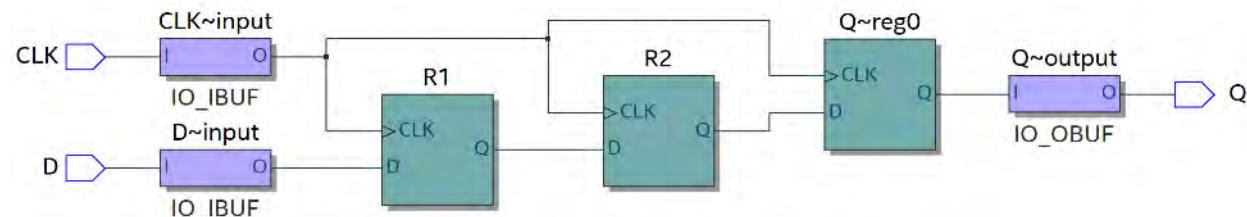
## 8.4 Good: Non-Blocking Assignment Operator ( $<=$ ) and Clocked Logic

In clocked, sequential circuits, we think of flip-flop or register assignments as all happening concurrently at the clock edge. This kind of concurrent behavior is exactly what the non-blocking assignment operator expresses. When using the non-blocking operator ( $<=$ ) to describe the behavior of a shift register, the order in which the assignments are written doesn't matter, since they all happen concurrently. The current values of D, R1, and R2 are first all evaluated and then they are all assigned their new values. Regardless of the order, the two examples below will both properly synthesize as a shift register.

```
module register_nonblock_front_to_back (
    input D, CLK,
    output reg Q);

    reg R1, R2;

    always @(posedge CLK)
    begin
        R1 <= D;
        R2 <= R1;
        Q <= R2;
    end
endmodule
```



```
module register_nonblock_back_to_front (
    input D, CLK,
    output reg Q);

    reg R1, R2;

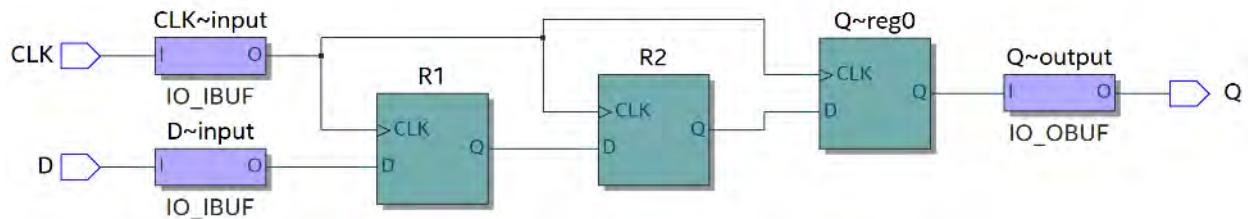
    always @(posedge CLK)
    begin
        R1 <= D;
        R2 <= R1;
        Q <= R2;
    end
endmodule
```

```

always @(posedge CLK)
begin
    Q  <= R2;
    R2 <= R1;
    R1 <= D;
end

endmodule

```



# 9 Memory

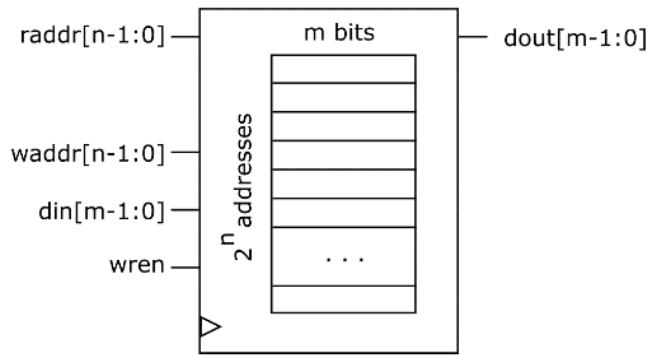
A *memory* is an indexed array of data storage locations. An index into a memory array is called an *address*. A random-access memory or *RAM* can be both read and written, while a read-only memory or *ROM* has predefined data values that cannot be modified.

## 9.1 RAM: Random Access Memory

A register file is a type of RAM composed of an array of registers. Register files typically have fewer storage locations than other kinds of RAM (and are hence faster) and have separate ports for read and write addresses, and may have multiple read address ports so that multiple locations can be read simultaneously.

### 9.1.1 Structure and Operation

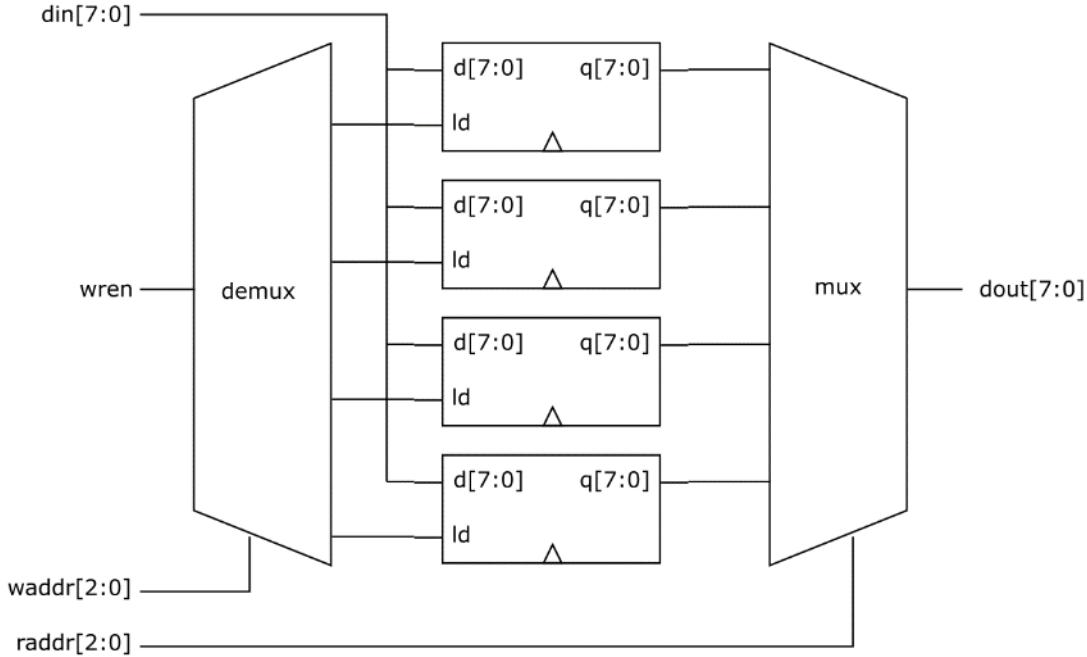
The figure below illustrates a basic RAM with one read port and one write port. Inside the register file is an array of  $2^n$  registers, each  $m$  bits wide.



The inputs and outputs are defined as follows:

Port	Direction	Description
Write Address (waddr)	input	Which register to write to
Read Address (raddr)	input	Which register to read from
Data In (din)	input	Data to be written to register at waddr
Data Out (dout)	output	Data read from register at raddr
Write Enable (wren)	input	Enable or inhibit writing to registers
Clock (clk)	input	Writing is synchronous with clock

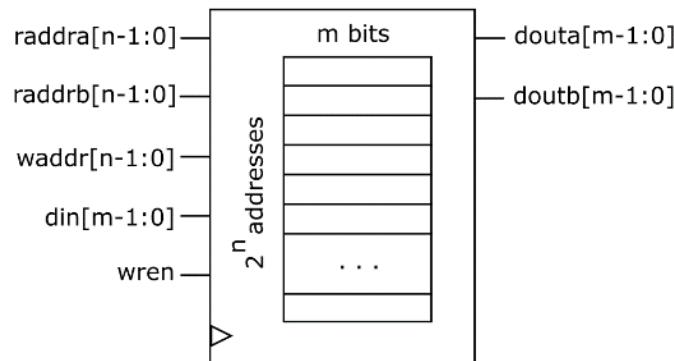
The following figure illustrates a basic structure for a register file containing 4 registers, each 8 bits wide.



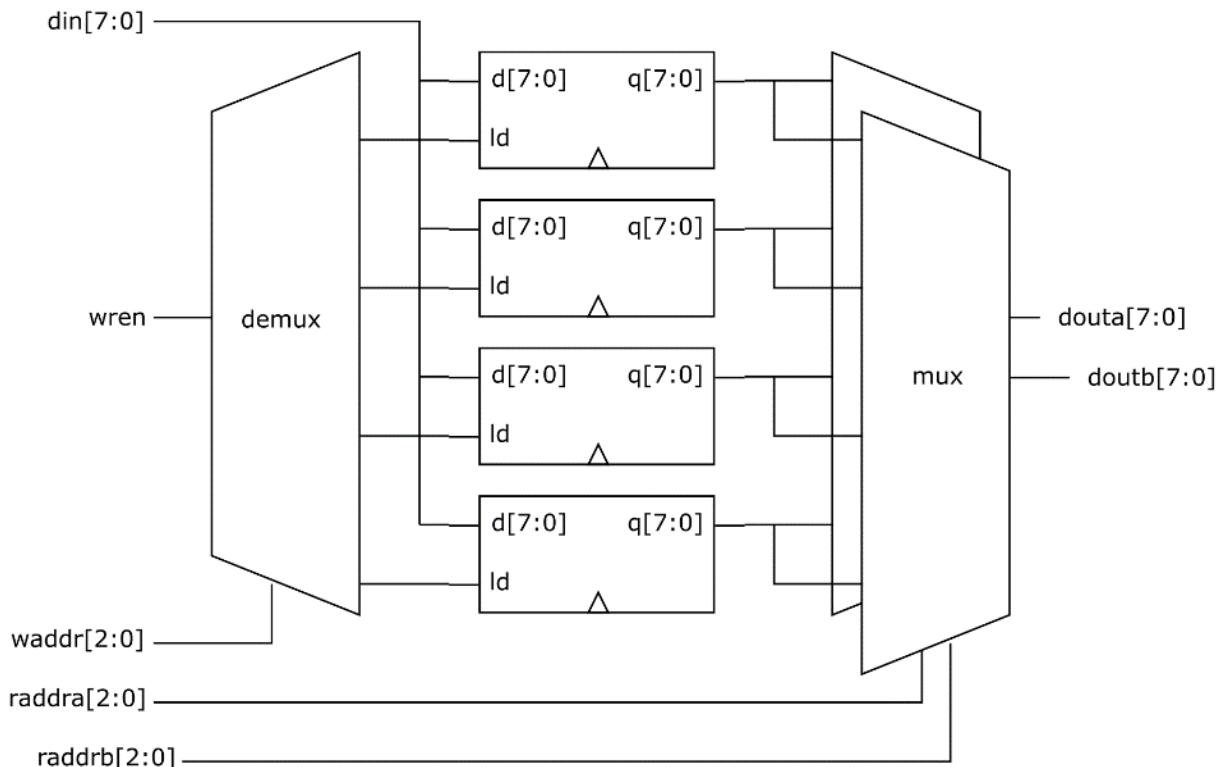
*Read Operation:* The outputs of the registers are connected to a 4-to-1 multiplexor, with the read address **raddr[1:0]** connected to the select lines of the multiplexor. Reading the register file simply consists of selecting which of the 4 register outputs to route to the output port **dout[7:0]**.

*Write Operation:* The register file data input port **din[7:0]** is connected to the data input **d[7:0]** of each of the 4 registers. A demultiplexor routes the value of the write enable signal **wren** to one of the 4 registers, as selected by the write address **waddr[1:0]**. The actual write occurs on the rising clock edge.

Adding a second read port makes it possible to read two values from two different (or the same) register simultaneously. An example of an application that uses this feature would be a computer that reads the values of two variables, adds them, and writes the result to a third variable. The figure below shows a register file with 2 read address ports **raddra** and **raddrb**, with corresponding data out ports **douta** and **doutb**.



Internally, adding a second read port involves adding a second output multiplexor.



### 9.1.2 Structural Verilog Implementation

#### 9.1.2.1 Register

```
module register4 (
    input [3:0] din,
    input ld,
    input clk,
    output reg [3:0] dout
);

    always @(posedge clk)
        if (ld)
            dout <= din;

endmodule
```

#### 9.1.2.2 Multiplexor

```
module mux4x4 (
    input [3:0] d0,
    input [3:0] d1,
    input [3:0] d2,
    input [3:0] d3,
    input [1:0] s,
    output reg [3:0] y
);
```

```

always @(*)
  case (s)
    0: y = d0;
    1: y = d1;
    2: y = d2;
    3: y = d3;
  endcase

endmodule

```

### 9.1.2.3 Demultiplexor

```

module demux4 (
  input [1:0] s,
  input en,
  output reg [0:3] y
);

  always @(*) begin
    y = 4'b0000;
    if (en) begin
      case (s)
        0: y = 4'b1000;
        1: y = 4'b0100;
        2: y = 4'b0010;
        3: y = 4'b0001;
      endcase
    end
  end
end

```

### 9.1.2.4 Complete Register File

```

module regfile_1r1w (
  input [1:0] raddr,
  input [1:0] waddr,
  input [3:0] din,
  input wren,
  input clk,
  output [3:0] dout
);

  wire [0:3] ld;
  wire [3:0] dout0, dout1, dout2, dout3;

  register4 register4_0 (
    .din      (din),
    .ld       (ld[0]),
    .clk      (clk),
    .dout     (dout0)
  );

```

```

register4 register4_1 (
    .din      (din),
    .ld       (ld[1]),
    .clk      (clk),
    .dout     (dout1)
);

register4 register4_2 (
    .din      (din),
    .ld       (ld[2]),
    .clk      (clk),
    .dout     (dout2)
);

register4 register4_3 (
    .din      (din),
    .ld       (ld[3]),
    .clk      (clk),
    .dout     (dout3)
);

demux4 demux4_0 (
    .s        (waddr),
    .en       (wren),
    .y        (ld)
);

mux4x4 mux4x4_0 (
    .d0      (dout0),
    .d1      (dout1),
    .d2      (dout2),
    .d3      (dout3),
    .s        (raddr),
    .y        (dout)
);

endmodule

```

### 9.1.2.5 Testbench

```

`timescale 1ns/1ns
module regfile_1r1w_tb ();
    reg [1:0] raddr;
    reg [1:0] waddr;
    reg [3:0] din;
    reg wren;
    reg clk;
    wire [3:0] dout;

```

```

regfile_1r1w uut (
    .raddr  (raddr),
    .waddr  (waddr),
    .din    (din),
    .wren   (wren),
    .clk    (clk),
    .dout   (dout)
);

always
    #5 clk = ~clk;

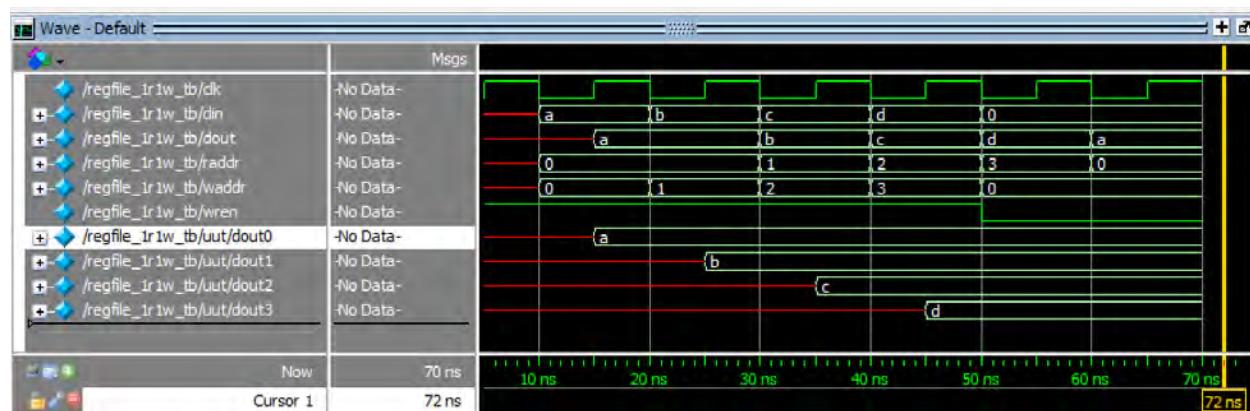
initial begin
    clk = 0; wren = 1;
    // write to each register and read on next cycle
    #10 raddr = 0; waddr = 0; din = 4'hA;
    #10 raddr = 0; waddr = 1; din = 4'hB;
    #10 raddr = 1; waddr = 2; din = 4'hC;
    #10 raddr = 2; waddr = 3; din = 4'hD;

    // disable wren and read back reg3
    #10 raddr = 3; waddr = 0; din = 0;
    wren = 0;

    // read back reg0 and make sure it didn't change
    #10 raddr = 0;
    #10 $stop;
end

endmodule

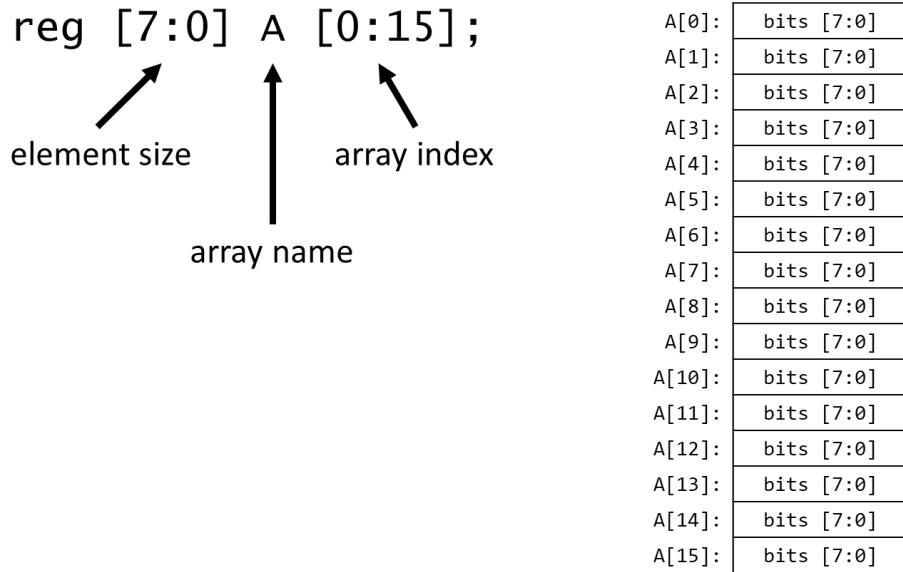
```



### 9.1.3 Behavioral Verilog Implementation Using Arrays

The previous section showed how to model the *structure* of a register file. Verilog also provides a mechanism for modeling the *behavior* of a register file using *arrays*. Most synthesis tools, including Altera Quartus, are able to automatically infer or generate a memory structure with register cells, decoders, and multiplexors given a behavioral description using arrays.

An array declaration in Verilog specifies three things: the bit range of each element in the array, the name of the array, and the range of the index into the array. The example below shows the declaration of an array named A that contains 16 elements, each of which is 8 bits wide. In Verilog, vectors may be declared in either ascending or descending order. In the model for a memory, the bits of the data elements are typically defined in descending order, while the index into the array is defined in ascending order.



Below is a behavioral model for a register file with 1 write port and 1 read port using an array. The model has two processes—the write process and the read process—running simultaneously. The write process is synchronous with the rising edge of the clock while the read process is asynchronous, meaning that dout gets a new value whenever raddr changes, independent of the clock.

```
module regfile_1r1w_behav (
    input      clk,
    input      wren,
    input [7:0] din,
    input [3:0] raddr,
    input [3:0] waddr,
    output [7:0] dout
);

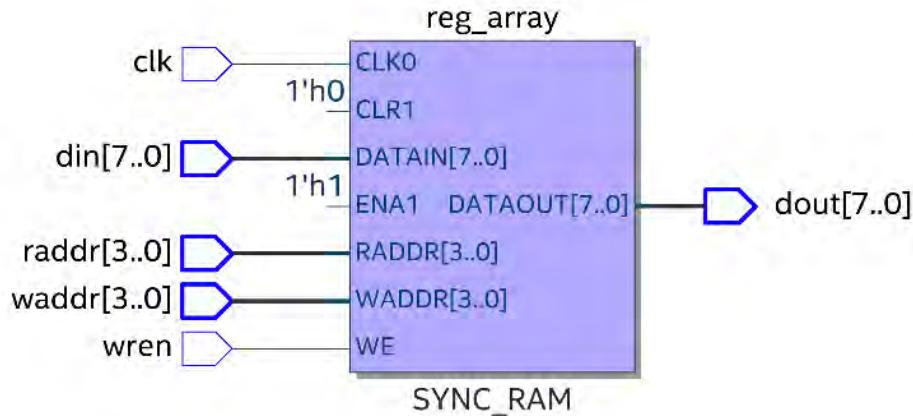
reg [7:0] reg_array [0:15];

// write process
always @(posedge clk)
    if (wren)
        reg_array[waddr] <= din;

// read process
assign dout = reg_array[raddr];

endmodule
```

The Quartus synthesis tools recognize this model as a RAM and the RTL netlist contains an abstract `SYNC_RAM` primitive:



To add a second read port, we add a second `assign` statement that runs simultaneously with the other two processes:

```
module regfile_2r1w_behav (
    input      clk,
    input      wren,
    input [7:0] din,
    input [3:0] raddr_a,
    input [3:0] raddr_b,
    input [3:0] waddr,
    output [7:0] dout_a,
    output [7:0] dout_b
);

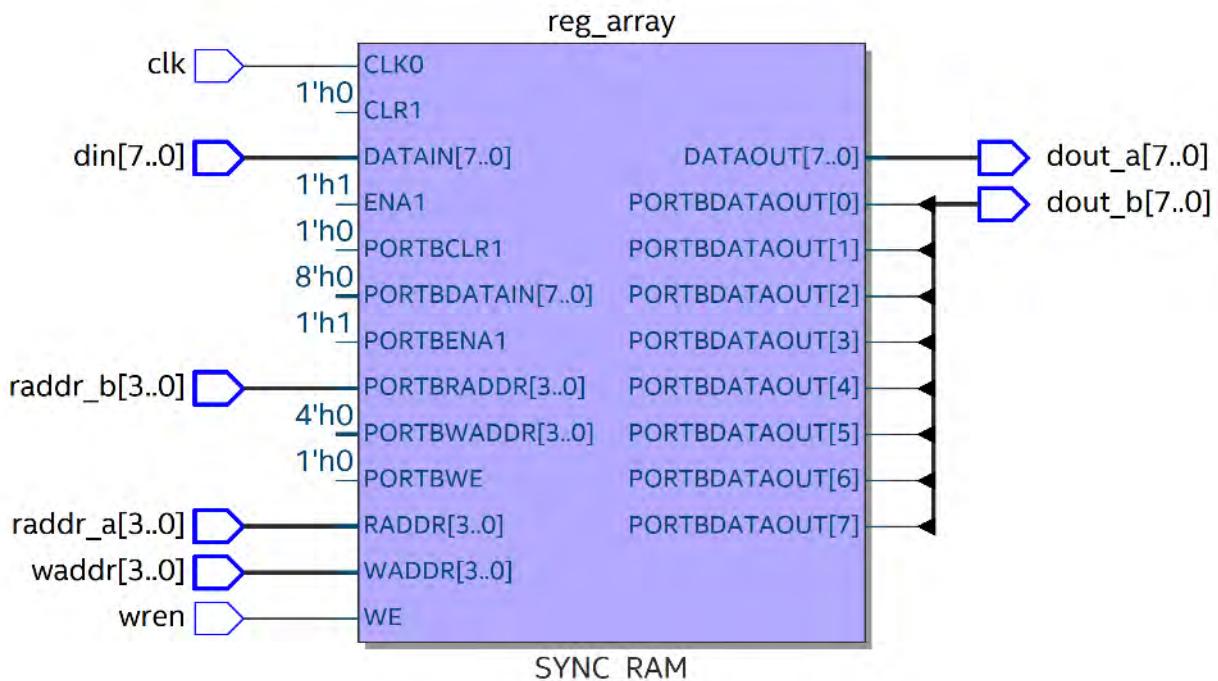
reg [7:0] reg_array [0:15];

always @(posedge clk)
    if (wren)
        reg_array[waddr] <= din;

assign dout_a = reg_array[raddr_a];
assign dout_b = reg_array[raddr_b];

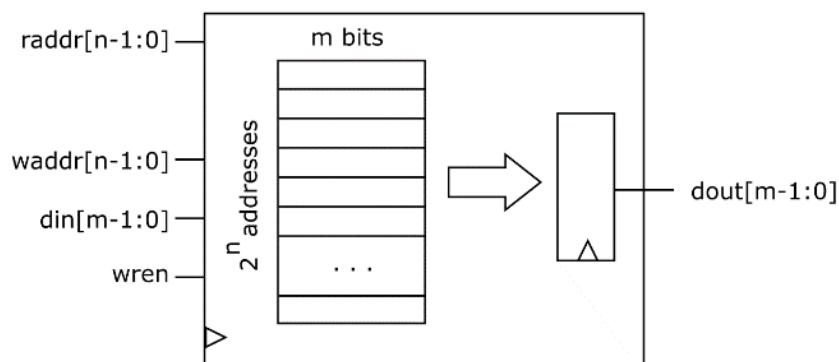
endmodule
```

The synthesized result uses an abstract `SYNC_RAM` primitive with dual ports:



#### 9.1.4 Reading Memory Synchronous with Clock

In the previous register file example, writes happened synchronous with the rising clock edge while reads happened whenever the address changed, asynchronously with respect to the clock. In larger memories, it is common for reads to also occur synchronous to the clock edge. Structurally, this is equivalent to placing a clocked register as a buffer just before the memory output.



We can model this in Verilog by placing the read operation inside of the `always` block sensitive to the clock edge, rather than using an `assign` statement that is independent of the clock. The following is a behavioral model of RAM containing 256, 16-bit words, with synchronous reads and writes:

```
module ram_256x16 (
    input      [7:0]  read_addr,
    input      [7:0]  write_addr,
    input      [15:0] din,
    input          clk,
```

```

input          we,
output reg [15:0] dout);

reg [15:0] mem_array [0:255];

always @(posedge clk) begin
  if (we)
    mem_array[write_addr] <= din;
  dout <= mem_array[read_addr];
end

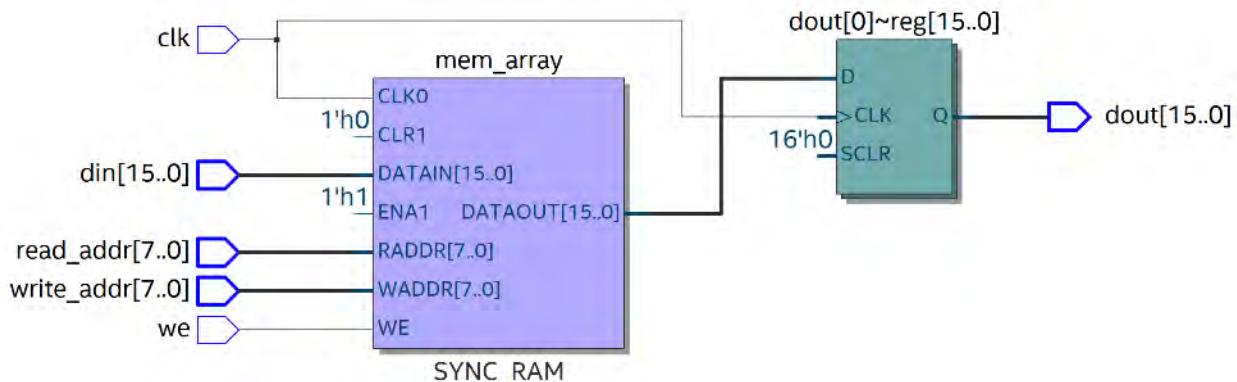
endmodule

```

Note that in a RAM with synchronous reads, the new data doesn't become available until the next rising clock edge. In other words, the data isn't available until the clock cycle *after* the read address changes.



The RTL netlist of the synthesized results adds a register after the data output port of the abstract SYNC\_RAM:



### 9.1.5 Why You Should Use Synchronous Memory

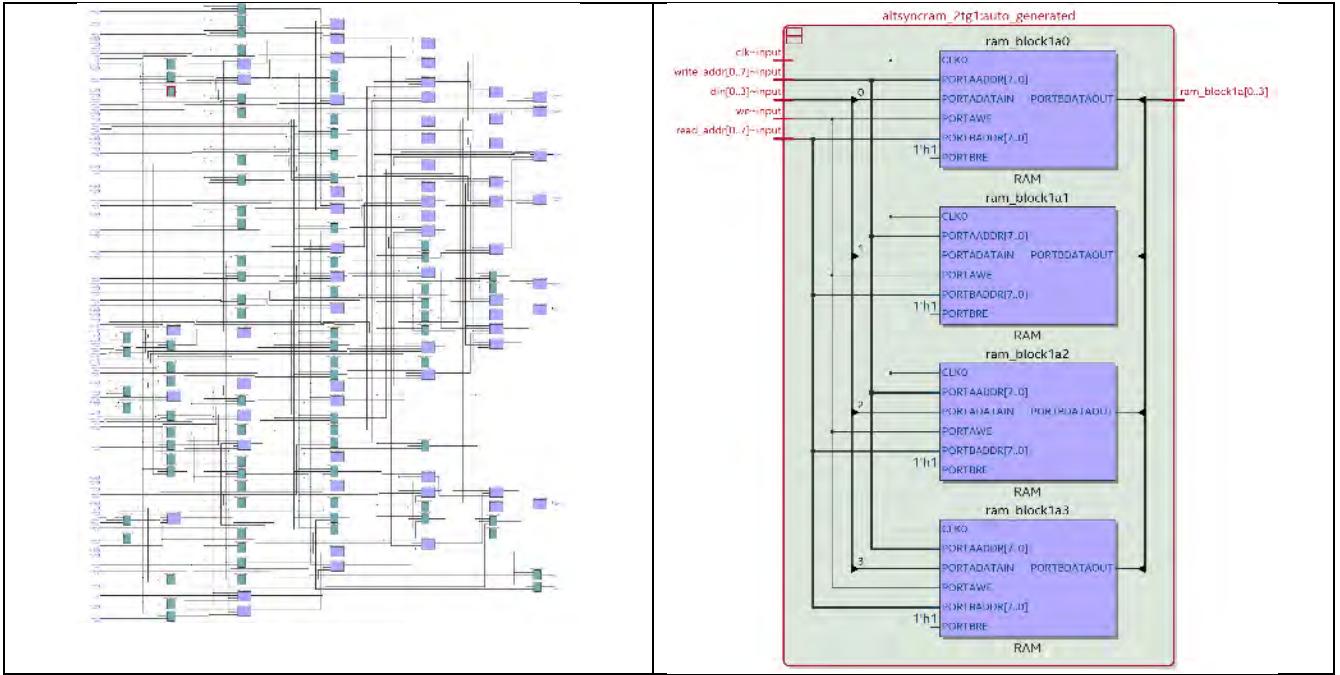
In the previous sections we looked at 2 versions of a RAM, one with reads that didn't depend on the clock and the output data appeared after the address changed, and a second version with synchronous reads where the output data appeared after the next rising clock edge. The version with asynchronous reads synthesized to the abstract module SYNC\_RAM (oddly named, since it supports asynchronous reads), while the version with synchronous reads synthesized to SYNC\_RAM with a register attached to the output. So at the RTL level, it *looks* like synchronous memory has a slightly more complicated implementation, since it

has the extra register. But looks at the RTL level can be deceiving—the reality is that at the physical level, a synchronous memory has a much more efficient implementation, because it is able to take advantage of the built-in memory banks inside of FPGA that have synchronous reads, whereas an asynchronous RAM is synthesized into a large collection of individual flip flops, much like the structural design of a register file that was presented at the beginning of this chapter.

The figure below illustrates the differences between the synthesis results for 256x4 asynchronous RAM vs. synchronous RAM in an Intel Cyclone IV E FPGA. In short, the synchronous RAM synthesizes very efficiently into 4 built-in physical RAM blocks (1 for each bit of a 4-bit data element) without using any logic elements, the asynchronous RAM synthesizes into 1,439 logic elements and doesn't use any built-in RAM blocks.

In summary, unless there is a very compelling reason to use asynchronous memory, you should pretty much always use synchronous memory in your designs.

Asynchronous RAM	Synchronous RAM
Verilog Source	
<pre>module ram_256x4_async (     input      [7:0]  read_addr,     input      [7:0]  write_addr,     input      [3:0]   din,     input          clk,     input          we,     output     [3:0]   dout);      reg [3:0] mem_array [0:255];      always @(posedge clk)         if (we)             mem_array[write_addr] &lt;= din;      assign dout = mem_array[read_addr];  endmodule</pre>	<pre>module ram_256x4_sync (     input      [7:0]  read_addr,     input      [7:0]  write_addr,     input      [3:0]   din,     input          clk,     input          we,     output reg   [3:0]   dout);      reg [3:0] mem_array [0:255];      always @(posedge clk) begin         if (we)             mem_array[write_addr] &lt;= din;         dout &lt;= mem_array[read_addr];     end  endmodule</pre>
RTL Netlist	
Post Mapping Physical Netlist	



Implementation Statistics			
Top-level Entity Name	ram_256x4_async	Top-level Entity Name	ram_256x4_sync
Family	Cyclone IV E	Family	Cyclone IV E
Device	EP4CE115F29C7	Device	EP4CE115F29C7
Timing Models	Final	Timing Models	Final
Total logic elements	1,439 / 114,480 ( 1 % )	Total logic elements	0 / 114,480 ( 0 % )
Total registers	1024	Total registers	0
Total pins	26 / 529 ( 5 % )	Total pins	26 / 529 ( 5 % )
Total virtual pins	0	Total virtual pins	0
Total memory bits	0 / 3,981,312 ( 0 % )	Total memory bits	1,024 / 3,981,312 ( < 1 % )
Embedded Multiplier 9-bit elements	0 / 532 ( 0 % )	Embedded Multiplier 9-bit elements	0 / 532 ( 0 % )
Total PLLs	0 / 4 ( 0 % )	Total PLLs	0 / 4 ( 0 % )

## 9.2 ROM: Read-Only Memory

A read-only memory or ROM is a memory that stores a predefined set of data values that can't be modified. Since the contents can't be modified, there is no need for data input or write enable ports. Like RAM, reads can be either synchronous or asynchronous with respect to the clock.

### 9.2.1 Using initial Block to Initialize ROM Contents

A common way of initializing ROM contents is to use an `initial` block. The following example illustrates this for a 256 by 16-bit ROM with asynchronous reads:

```
module async_rom_256x16 (
    input      [7:0]  addr,
    output     [15:0]  dout
);

    reg [15:0] mem_array [0:255];
```

```

initial begin
mem_array[0] = 16'haaaa;
mem_array[1] = 16'hbbbb;
mem_array[2] = 16'hcccc;
mem_array[3] = 16'hdddd;
end

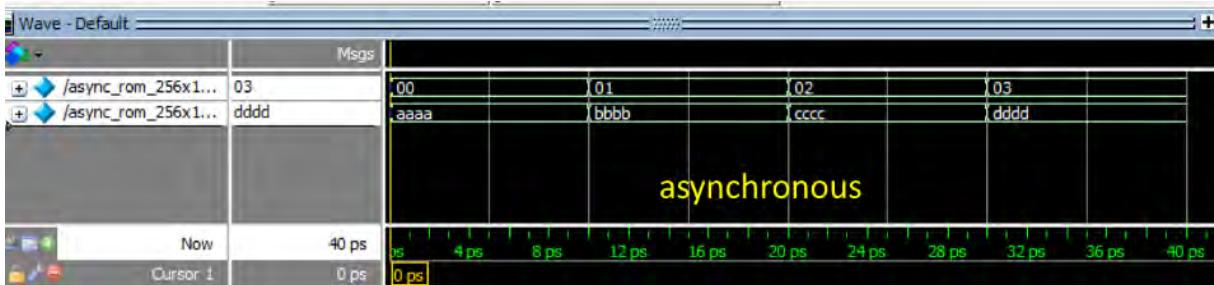
assign dout = mem_array[addr];

endmodule

```

Note that you can also initialize the contents of a RAM the same way.

In a ROM with asynchronous read, the output data changes when the address changes:



Here's a version with synchronous reads:

```

module synch_rom_256x16 (
    input      [7:0]    addr,
    input      clk,
    output reg [15:0]  dout
);

reg [15:0] mem_array [0:255];

initial begin
    mem_array[0] = 16'haaaa;
    mem_array[1] = 16'hbbbb;
    mem_array[2] = 16'hcccc;
    mem_array[3] = 16'hdddd;
end

always @(posedge clk)
    dout <= mem_array[addr];

endmodule

```

In a ROM with synchronous reads, the data changes on the rising clock edge after the address changes:



### 9.2.2 Using for Loop to Initialize ROM Contents

For a large memory, it is impractical to initialize all of the memory locations individually using an `initial` block. Typically, we want to initialize a memory to all zeros (or maybe all ones) and then just set some locations to a different value. One way to do this in Verilog is by using a `for` loop to set the default values, and then individually change the values of locations that differ from the default.

The syntax of a Verilog `for` loop is the same as that for the C programming language. The loop counter is of type `integer`, which is a way of expressing a variable in Verilog that is not a hardware signal.

```
module rom_256x16 (
    input      [7:0]    addr,
    input      clk,
    output reg [15:0]  dout
);

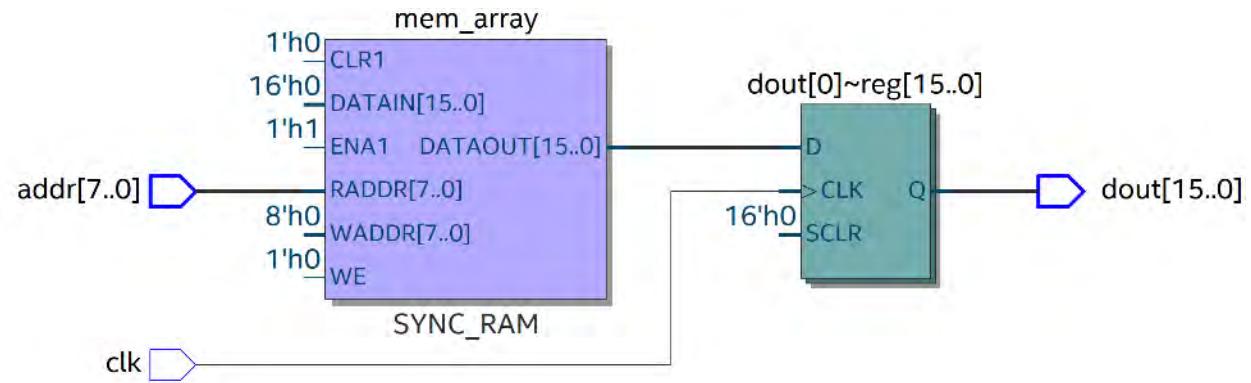
reg [15:0] mem_array [0:255];

integer i;      // a variable, not a signal
initial begin
    // initialize default values
    for (i = 0; i < 256; i = i + 1)
        mem_array[i] = 0;
    // change values for non-default locations
    mem_array[0] = 16'hfaaa;
end

always @(posedge clk)
    dout <= mem_array[addr];

endmodule
```

Here's the RTL netlist of the synthesized result. Note that the `for` loop doesn't create any additional hardware, and there is no evidence of it in the netlist—it is just part of the initialization of the memory.



### 9.3 Using \$readmemh and \$readmemb to Initialize Memory from a File

Another way to initialize memory is to read data from a file. Verilog provides two commands for doing this: `$readmemh` for reading hexadecimal data and `$readmemb` for reading binary data. The following is an example of a synchronous RAM with 64 entries, where each entry is 8 bits wide, that initializes the memory array M with hex values contained in the file `memdata.txt`:

```
module ram_64x8_readmemh (
    input      clk,
    input      wren,
    input [7:0] din,
    input [5:0] addr,
    output reg [7:0] dout
);

reg [7:0] M [0:63];
initial $readmemh("memdata.txt", M);

always @(posedge clk) begin
    if (wren)
        M[addr] <= din;
    dout <= M[addr];
end

endmodule
```

The memory data files are simple text files, where values are separated by whitespace characters (mix of space, tab, or newline). For example, `memdata.txt` may be formatted as:

```
ab cd 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 ef
```

where  $M[0] = ab$ ,  $M[1] = cd$ ,  $M[63] = ef$  and the remaining entries equal 00.

**Important note:** The \$readmemh and \$readmemb commands work for both synthesis and simulation. However, when using Quartus and ModelSim, you will need **two copies** of the data files so that the tools can find them. For synthesis, the memory data file needs to be placed in the same folder as the Verilog source file (or in the top-level Quartus project folder). For simulation, a copy of the memory data file must be placed in the `simulation/modelsim` folder under the top-level Quartus project folder.

## 9.4 An X-Y Image RAM

The physical memory arrays available in the FPGA only have a 1-dimensional addressing scheme—that is the memory has a linear array of addresses, and there is an element stored at each address. In some applications, however, it may be convenient to be able to address memory as an array with 2 (or more) dimensions. Such is the case in working with images, where each pixel has an x (column number) and y (row number) coordinate.

We can implement a 2 dimensional memory with separate x and y addresses on top of a 1-dimensional memory array by using combinational logic to translate row and column numbers into a single linear address space. For example, suppose that we want to store an image that is 160 pixels in width and 120 pixels in height. The total number of pixels is  $160 \times 160 = 19,200$ . Thus the entire image could fit in a 1-dimensional memory array with addresses [0:19199]. The typical way of storing the image in a 1-dimensional memory is row-by-row, where row 0 is in addresses [0:159], row 1 is in addresses [160:319], row 2 is in addresses [320:479], and so on. The formula for determining the address of a pixel in column x and row y is thus:

$$\text{address} = (160) y + x$$

The following Verilog module calculates the address from the x and y coordinates and then uses this to address a 1-dimensiona RAM array with 1 read and 1 write port, where pixels are represented as 3-bit RGB (red-green-blue) color values:

```
module image_ram_1r1w (
    input      clk,
    input      wren,
    input [2:0] din,    // 3 RGB bits per pixel
    input [7:0] x,      // 8 bits for up to 160
    input [6:0] y,      // 7 bits for up to 120
    output reg [2:0] dout
);

wire [14:0] addr;      // 15 bits for up to 19199
assign addr = 160*y + x;

reg [2:0] M [0:19199];
initial $readmemh("testpattern.mem", M);

always @(posedge clk) begin
    if (wren)
        M[addr] <= din;
```

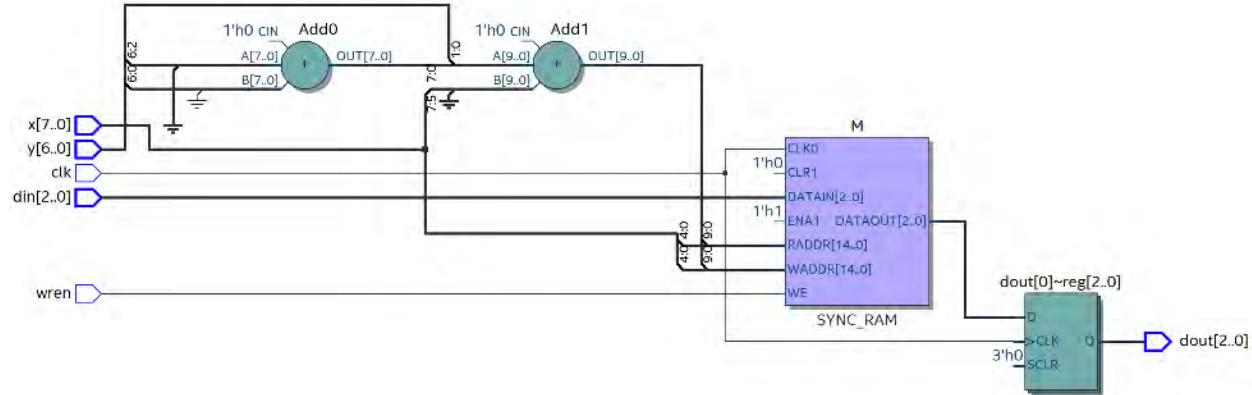
```

dout <= M[addr];
end

endmodule

```

Here is the RTL netlist for the synthesized result:



As expected, the synthesis tool inferred a built-in RAM array from the Verilog description, with combinational logic included to calculate the read address RADDR and write address WADDR of the RAM. The register added at the DATAOUT port of the RAM is because reads are synchronous with the clock. What is surprising is that the combinational logic only has adders but no multiplier. This is because multiplication by a constant can be implemented as shift-and-add operations without the need for a slower and more costly general-purpose multiplier circuit. Specifically, since  $8'd160 = 8'b10100000$ , we can express  $160*x$  as

$$(x * 8'b10000000) + (x * 6'b100000) = \{x, 7'b0\} + \{x, 5'b0\}$$

The synthesized result takes the proper bits from  $x$  and shifts them to the appropriate bits of the adder inputs.

## 9.5 Multiport X-Y Image Memory for VGA Applications

For applications that display an image on VGA display, we need an image memory with multiple read and write ports. This is because the user application is typically writing and reading memory locations simultaneously while the VGA controller is reading it to display the image. Moreover, the user application and the VGA controller are typically accessing the RAM at different clock rates; for the VGA to work properly, it must read the RAM at 25 MHz, while the user application may operate at 50 MHz.

The following is a Verilog model for an X-Y image RAM with a read and write port for a user processor and a separate read port for the VGA controller. There are separate clocks for the user processor and the VGA ports; this behavior is modeled using two `always` blocks, one for each clock:

```

module image_ram_2r1w (
    input          wren,
    input          clk_proc,
    input          clk_vga,
    input [7:0]    x_proc,
    input [6:0]    y_proc,

```

```

input      [7:0] x_vga,
input      [6:0] y_vga,
input      [2:0] din,
output reg  [2:0] dout_proc,
output reg  [2:0] dout_vga
);

parameter IMAGE_FILE = "blank.mem";

wire [14:0] addr_proc = 160*y_proc + x_proc;
wire [14:0] addr_vga = 160*y_vga + x_vga;

reg [2:0] ram [0:19199];
initial $readmemh(IMAGE_FILE, ram);

always @(posedge clk_proc) begin
    if (wren)
        ram[addr_proc] <= din;

    dout_proc <= ram[addr_proc];
end

always @(posedge clk_vga) begin
    dout_vga <= ram[addr_vga];
end

endmodule

```

The module uses a parameter `IMAGE_FILE` to specify the data file read by `$readmemh`. The main advantage of this is that it makes it possible to specify the file with the `defparam` command when the module is instantiated, rather than having to modify the contents of the file itself. For example, to change the file to “`obstacle_course.mem`”, we could instantiate the module as:

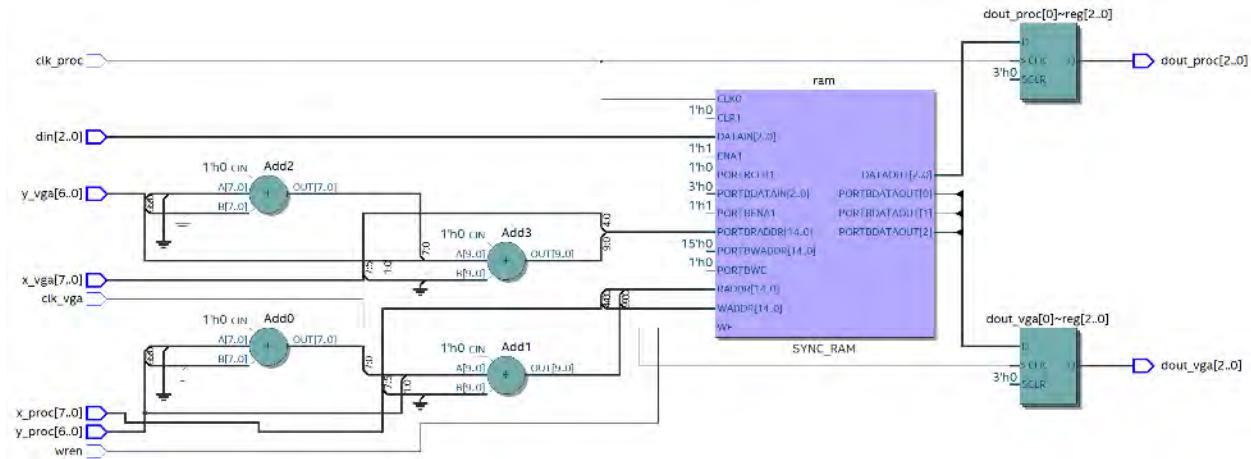
```

image_ram_2r1w image (
    .wren      (plot),
    .clk_proc   (CLOCK_50),
    .clk_vga    (VGA_CLK),
    .x_proc     (x_proc),
    .y_proc     (y_proc),
    .x_vga      (x_vga),
    .y_vga      (y_vga),
    .din        (plot_color),
    .dout_proc   (image_color),
    .dout_vga    (vga_color)
);
defparam image.IMAGE_FILE = "obstacle_course.mem";

```

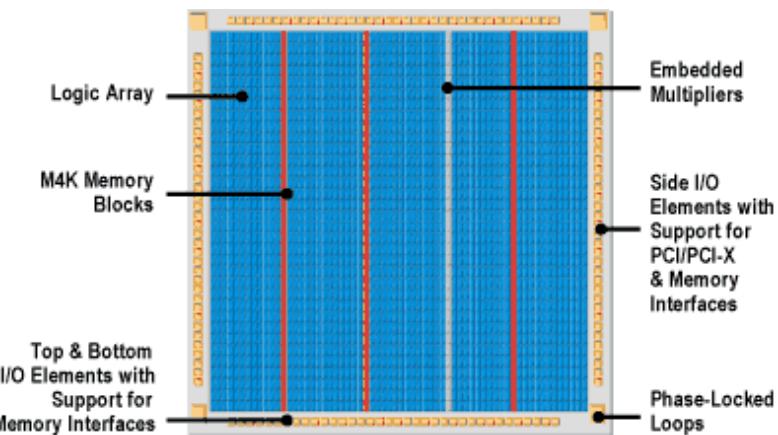
This feature is especially useful for applications where there are multiple instances of `image_ram_2r1w`, each initialized with a different image.

Here’s the synthesized result, which uses a dual-port built in RAM primitive with 2 clock inputs:



## 9.6 Altera Parametrized Memory Module Library

The Altera Cyclone FPGA series contains tens of thousands of logic elements consisting of configurable logic blocks and registers. It also contains hardwired memory blocks and other custom hardware such as multipliers that are available for use in designs. When a user gives a behavioral description of a memory, the synthesis tool could either generate a memory from registers and logic blocks or it could use one of the hardwired memory blocks. In order to make it easier to use the hardwired memory blocks efficiently, Altera provides a *library of parametrized modules (LPM)* that designers can include in their Verilog models. Detailed usage of the LPM is a somewhat advanced topic that goes beyond the scope of this course, but because some of the Altera/Terasic DE2-115 libraries make use of LPM modules, we'll touch on the basics.



The LPM modules take advantage of a Verilog language feature called parameters, which unlike ports are not part of the hardware but rather provide a way to specify a property of the hardware, such as the width of a bus. Here's an example of 2 modules from the library:

**lpm\_ram\_dq**

Parameterized Random Access Memory with Separate Input and Output Ports

The `lpm_ram_dq` function can be used as either synchronous or asynchronous random access memory. The `lpm_ram_dq` function has separate input and output data buses.

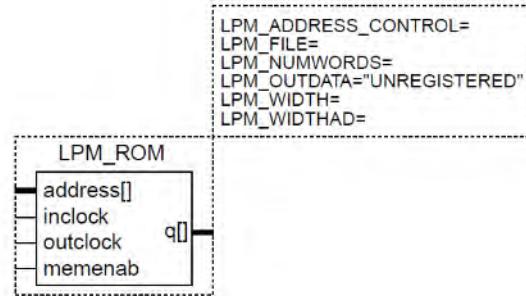
```
LPM_ADDRESS_CONTROL=
LPM_FILE=
LPM_INDATA=
LPM_NUMWORDS=
LPM_OUTDATA="UNREGISTERED"
LPM_WIDTH=
LPM_WIDTHAD=
```

```
LPM_RAM_DQ
  data[]      q[]
  address[]   inclock
  we          outclock
```

## lpm\_rom

Parameterized Read-Only Memory

The `lpm_rom` function can be used as either synchronous or asynchronous read-only memory.



Parameter values are defined using the Verilog `defparam` statement. Here's an example of a design that uses an instance of `lpm_rom`. The `defparam` statements following the instantiation of the module from the library define parameters of the instances:

- data width
- address width
- whether or not an extra register should be included at the outputs
- file containing ROM contents

```
module lpm_rom_demo (
    input      [7:0]    addr,
    input          clock,
    output     [15:0]   q);

    lpm_rom rom (
        .address (addr),
        .inclock (clock),
        .q       (q)
    );
    defparam rom.lpm_width = 16;
    defparam rom.lpm_widthad = 8;
    defparam rom.lpm_outdata = "UNREGISTERED";
    defparam rom.lpm_file = "rom_contents.mif";

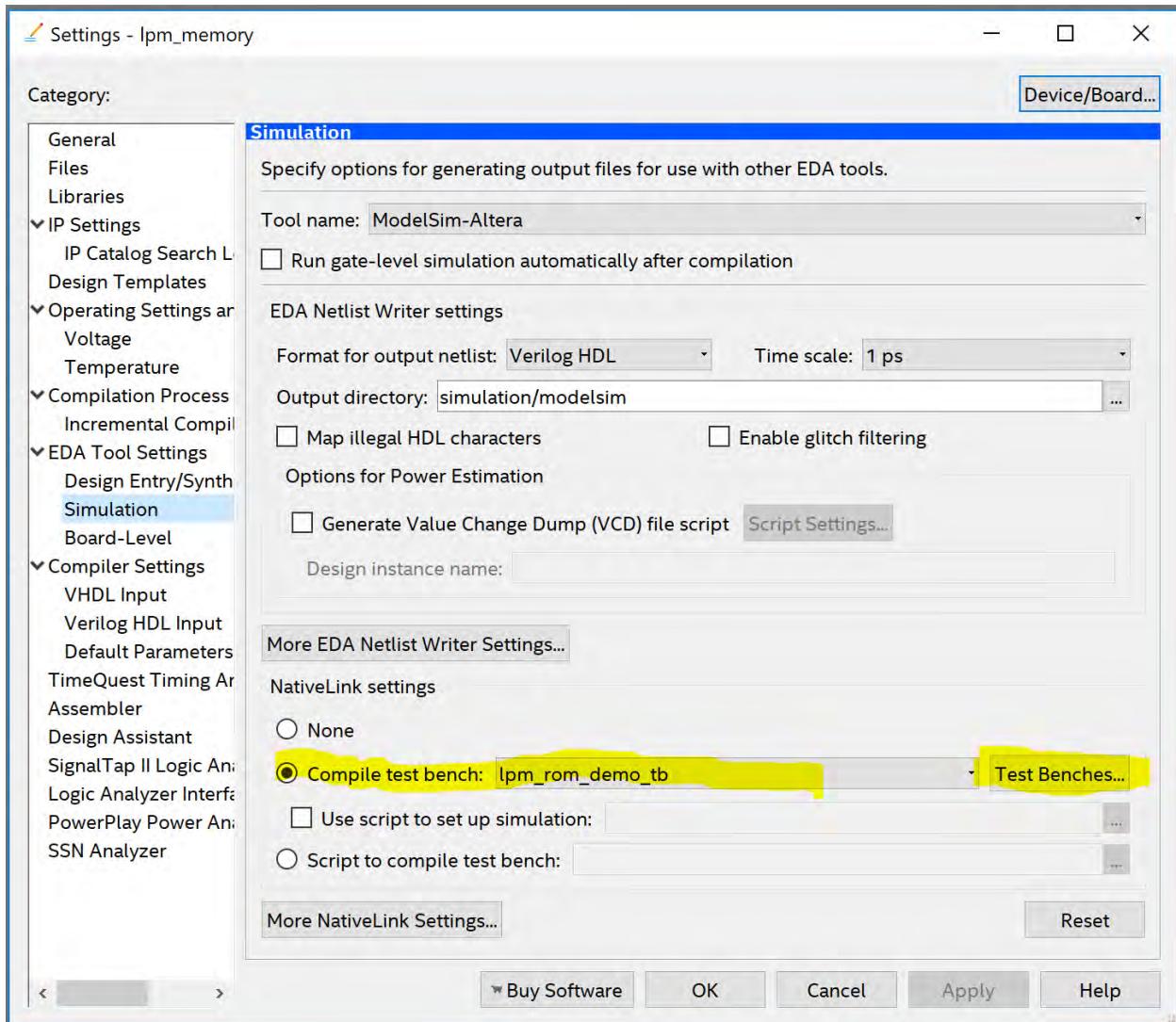
endmodule
```

The file `rom_contents.mif` is a *memory image file*. Here's an example to illustrate the syntax:

```
WIDTH=16;
DEPTH=256;
ADDRESS_RADIX=UNS;
DATA_RADIX=UNS;
CONTENT BEGIN
    0      :  0;
    1      :  1;
    2      :  2;
    3      :  3;
    4      :  4;
    5      :  5;
```

```
[6..255] : 0;
END;
```

**Note:** When using Modelsim to simulate an LPM component, you must use NativeLink to compile and run the testbench from Quartus—it is not sufficient to compile the testbench in Modelsim alone as it will not find the simulation models for the LPM components. To use NativeLink select Assignments > Settings from the Quartus main menu and from the popup select Simulation under EDA Tool Settings. Click on the “Testbenches...” button add a new testbench.



Also, the memory image files (.mif) must be included in the project. If the simulation still has problems finding them, move them to the top-level directory of the project

Complete documentation on the Altera LPM library may be found at  
[https://www.altera.com/en\\_US/pdfs/literature/catalogs/lpm.pdf](https://www.altera.com/en_US/pdfs/literature/catalogs/lpm.pdf)

# 10 Finite State Machines

## Example Quartus Project: fsm

Finite state machines (FSMs) are the primary mechanisms for implementing controllers in digital systems. In this section we look at the implementation of an FSM in Verilog. In the olden days before HDLs and modern synthesis tools, FSM design was a fairly labor intensive process involving the following steps:

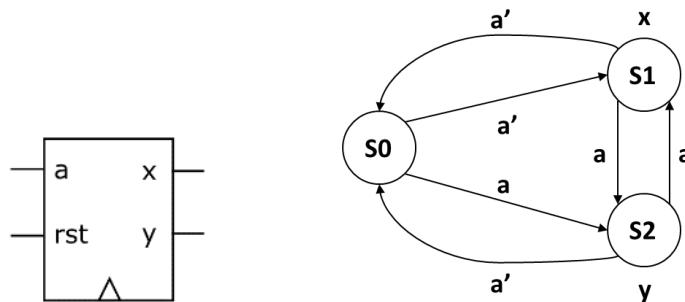
1. Draw an FSM diagram
2. Encode the states in the diagram as binary state numbers.
3. Construct a state transition table, which serves as the truth table for the next state and output logic.
4. Minimize the next state and output logic
5. Implement the next state and output logic as a combinational logic circuit.
6. Connect the next state and output combinational logic with the state registers to complete the implementation

Now with modern CAD tools, there is no need to minimize the logic equations manually. Moreover, if the FSM behavior is modeled in a particular way in Verilog using a `case` statement, synthesis tools will recognize that this is an FSM and deal with it in a special way.

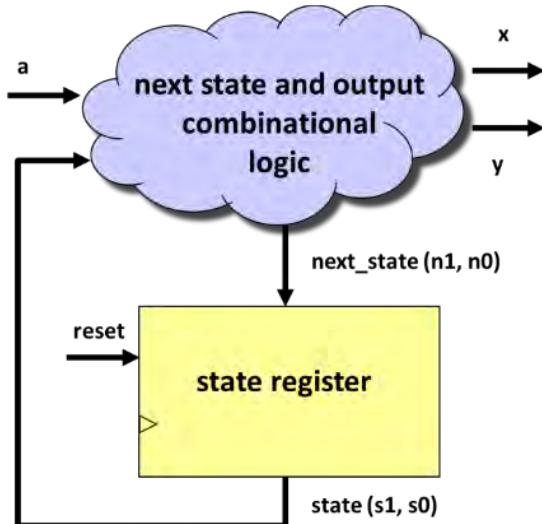
In this section will illustrate the complete process of implementing an FSM in Verilog, using both “old school” and “modern” methodologies. The “old school” approach involves explicitly deriving the next state and output combinational logic equations, while the “modern” approach uses a `case` statement that synthesis tools recognize. The “old school” low-level approach is good for understanding the low-level details of an FSM, but the “modern” high-level approach is what you should use in practice with an actual design.

### 10.1 Example State Machine Diagram and Equations

As an example system for FSM implementation, consider the following 3-state system with input  $a$  and outputs  $x$  and  $y$ :



The architecture of the FSM is:



The state table for the FSM is given below. Note that even though there is no state assigned to  $(s_1, s_0) = (1, 1)$ , it still must be accounted for in the table.

state	s1	s0	a	n1	n0	x	y
s0	0	0	0	0	1	0	0
s0	0	0	1	1	0	0	0
s1	0	1	0	0	0	1	0
s1	0	1	1	1	0	1	0
s2	1	0	0	0	0	0	1
s2	1	0	1	0	1	0	1
--	1	1	0	0	0	0	0
--	1	1	1	0	0	0	0

The minimized logic equations for the next state and output combinational logic are:

$$n1 = a s1'$$

a \ s1 s0	00	01	11	10
0				
1	1	1		

$$n0 = a' s1' s0' + a s1 s0'$$

a \ s1 s0	00	01	11	10
0				
1				1

$$x = s1' s0$$

a \ s1 s0	00	01	11	10
0				
1		1		

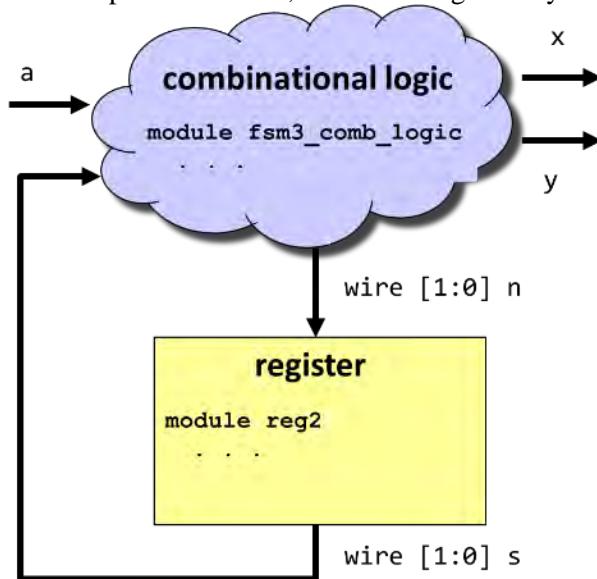
$$y = s1 s0'$$

a \ s1 s0	00	01	11	10
0				
1				1

## 10.2 Low-Level Implementation: Combinational Logic and State Register as Separate Modules

### 10.2.1 Structural Interconnection

The first “old school” low-level implementation is to implement the next state and output combinational logic and the state register in two separate modules, connected together by wires.



```

module fsm3_struct (
    input    clk,
    input    reset,
    input    a,
    output   x,
    output   y);

    wire [1:0] n;

```

```

wire [1:0] s;

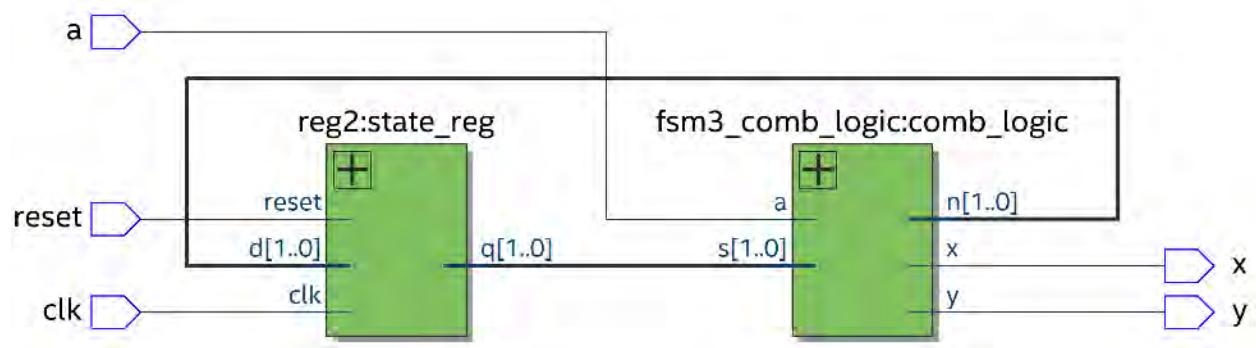
reg2 state_reg (
    .clk      (clk),
    .reset    (reset),
    .d        (n),
    .q        (s)
);

fsm3_comb_logic comb_logic (
    .a        (a),
    .s        (s),
    .n        (n),
    .x        (x),
    .y        (y)
);

endmodule

```

This synthesizes as expected:



## 10.2.2 State Register

The register synthesizes as flip flops with a multiplexor to handle the reset:

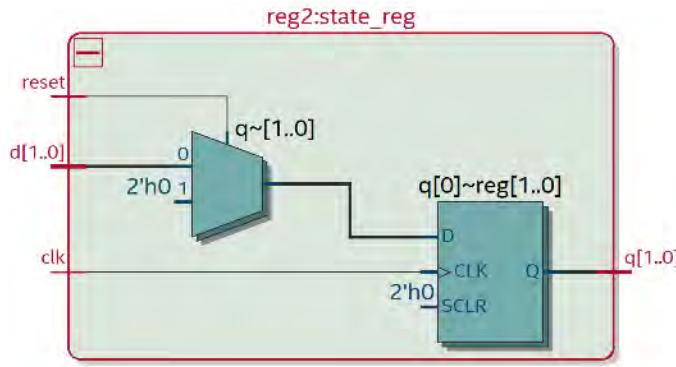
```

module reg2 (
    input          clk,
    input          reset,
    input [1:0]    d,
    output reg [1:0] q);

    always @(posedge clk)
        if (reset)
            q <= 2'h0;
        else
            q <= d;

endmodule

```



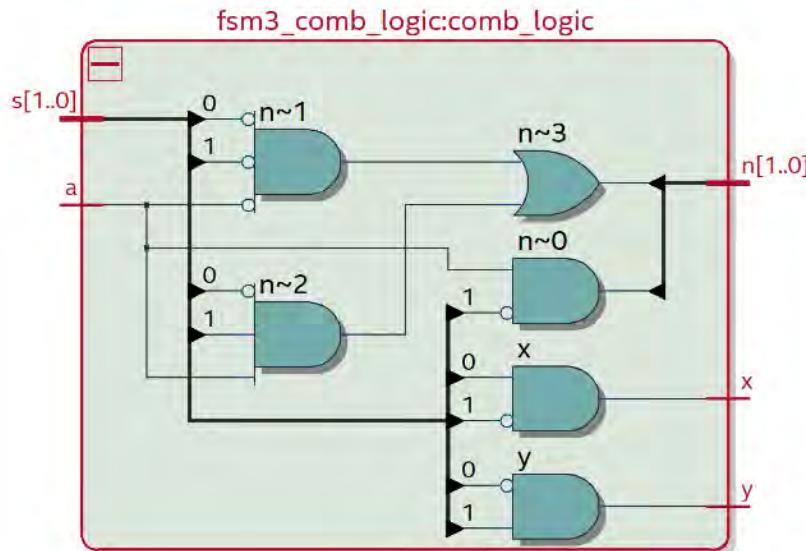
### 10.2.3 Next State and Output Combinational Logic

The combinational logic synthesizes as basic logic gates:

```
module fsm3_comb_logic (
    input      a,
    input [1:0] s,
    output [1:0] n,
    output      x,
    output      y);

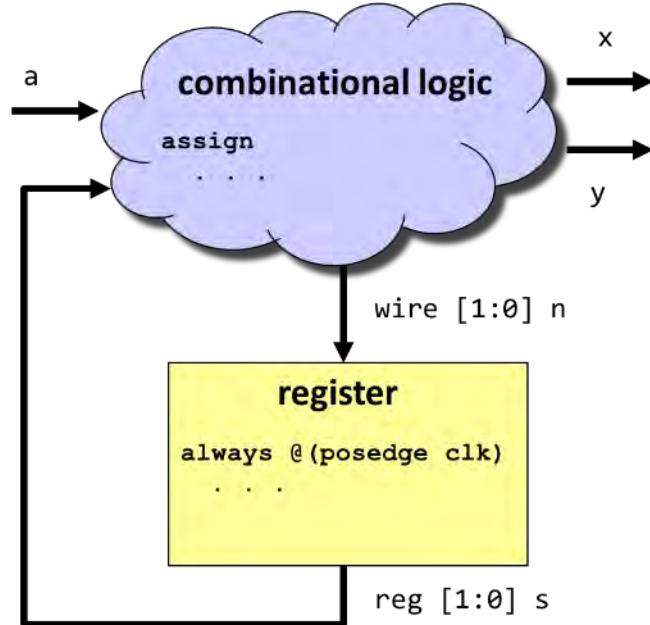
    assign n[1] = a & ~s[1];
    assign n[0] = ~a & ~s[1] & ~s[0] | a & s[1] & ~s[0];
    assign x    = ~s[1] & s[0];
    assign y    = s[1] & ~s[0];

endmodule
```



### 10.3 Low-Level Implementation: Combinational Logic as assign Statements

The second “old school” low-level approach is to implement both the next state/output logic and the state register in a single module as two processes running concurrently. The state register uses an `always` block and the next state/output logic uses `assign` statements.



```

module fsm3_assign (
    input      clk,
    input      reset,
    input      a,
    output     x,
    output     y);

    reg  [1:0] s;
    wire [1:0] n;

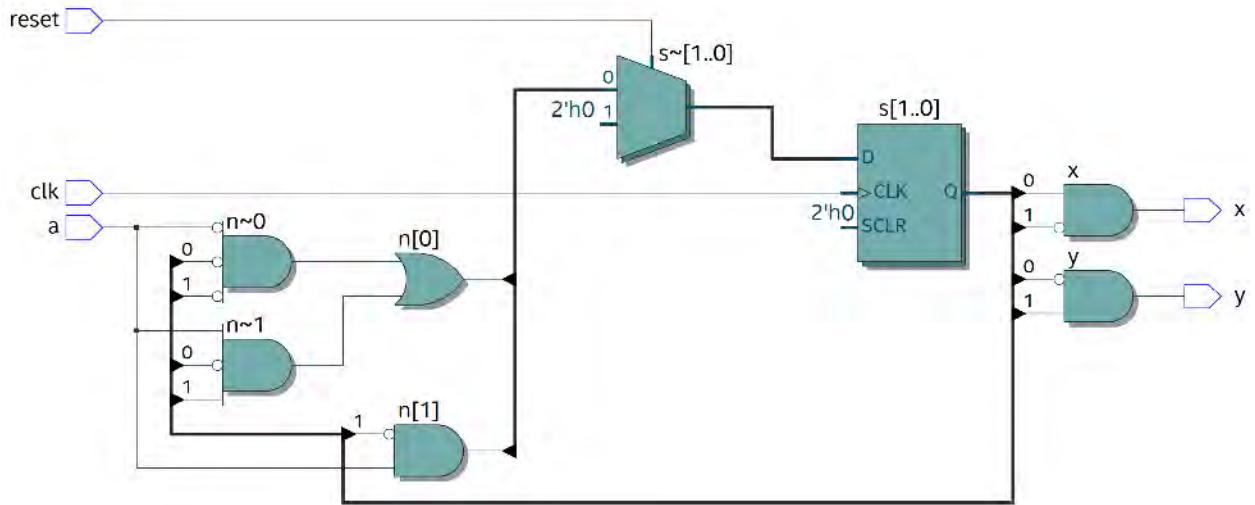
    always @(posedge clk)
        if (reset)
            s <= 2'h0;
        else
            s <= n;

        assign n[1] = a & ~s[1];
        assign n[0] = ~a & ~s[1] & ~s[0] | a & s[1] & ~s[0];
        assign x    = ~s[1] & s[0];
        assign y    = s[1] & ~s[0];

```

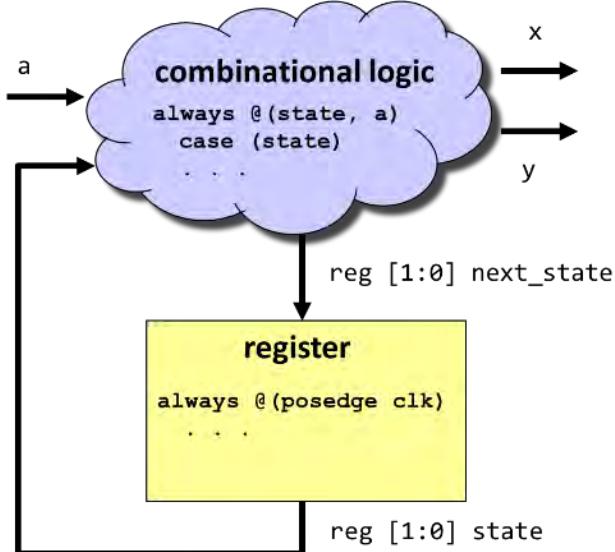
endmodule

It synthesizes as expected:



## 10.4 Preferred High-Level Implementation: Combinational Logic as case Statement

The preferred, “modern” high-level approach to implementing an FSM in Verilog such that synthesis tools will recognize it is to use a single module with two concurrent processes: an `always` block sensitive to the clock edge for the state register and another `always` block using a `case` statement for the next state and output logic.



```
module fsm3_case (
    input      clk,
    input      reset,
    input      a,
    output reg x,
    output reg y);

parameter S0 = 2'h0;
parameter S1 = 2'h1;
```

```

parameter S2 = 2'h2;

reg [1:0] state, next_state;

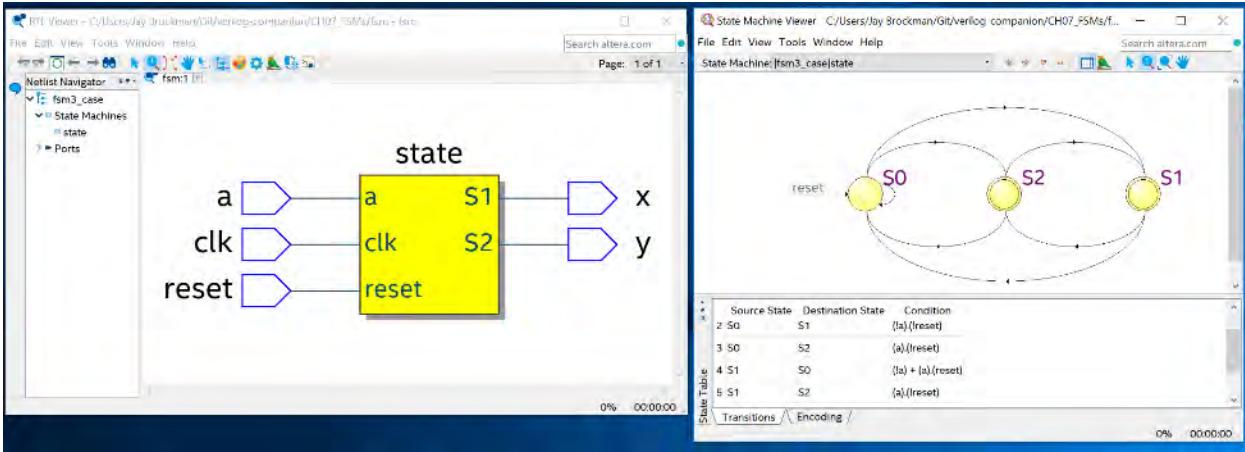
always @(posedge clk)
  if (reset)
    state <= S0;
  else
    state <= next_state;

always @(*) begin
  x = 0;
  y = 0;
  case (state)
    S0: begin
      if (a)
        next_state = S2;
      else
        next_state = S1;
    end
    S1: begin
      x = 1;
      if (a)
        next_state = S2;
      else
        next_state = S0;
    end
    S2: begin
      y = 1;
      if (a)
        next_state = S1;
      else
        next_state = S0;
    end
    default: begin
      next_state = S0;
    end
  endcase
end

endmodule

```

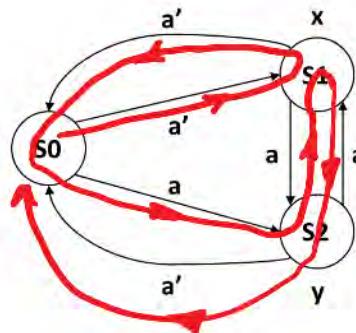
Using this modeling style produces a very different result from synthesis. The synthesis tool recognizes the model as an FSM and treats it accordingly. When viewing the RTL netlist in Altera Quartus, rather than showing flip flops and logic gates, it instead shows that this is a state machine with a corresponding FSM diagram.



## 10.5 Simulating FSM Behavior

To simulate the 3-state FSM we use a testbench that sets the values of input **a** prior to each rising clock edge such that all the states and transitions are visited. We begin the simulation by resetting the FSM by holding **reset** high for 1 clock cycle, ensuring that the initial state is state 0.

The following testbench visits the states of fsm3\_case in the order illustrated below:



```
module fsm3_tb ();
    reg clk;
    reg reset;
    reg a;
    wire x;
    wire y;

    fsm3_case uut (
        .clk      (clk),
        .reset    (reset),
        .a        (a),
        .x        (x),
        .y        (y)
    );

    always #5 clk = ~clk;

```

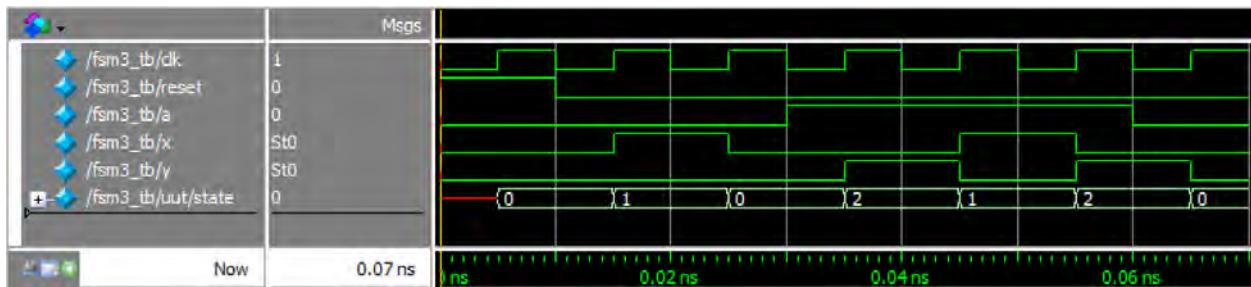
```

initial begin
    clk = 0;  reset = 1;  a = 0;
    #10 reset = 0; // S0
    #10;           // S1
    #10 a = 1;    // S0
    #10;           // S2
    #10;           // S1
    #10 a = 0;    // S2
    #10 $stop;     // S0
end

endmodule

```

The simulation results are as follows:



## 10.6 DE2-115 Board Single-Stepping Demonstration

Module `fsm3_DE2` connects the inputs, outputs, and state register of a case statement implementation of `fsm3` to buttons, switches, LED, and a 7-segment display of the DE2-115 board. The clock of the FSM is controlled by pushbutton `KEY[0]`, so that you can single-step around the FSM by setting the input `a` with a switch and then press a button to advance to the next state.

Because of mechanical bouncing that can cause the pushbutton to produce several pulses when it is pushed, `KEY[0]` is debounced using module `debounce` to produce a clean clock signal. (See the section on Debouncing in this document).

The connections to the DE2-115 board are as follows:

<b>fsm3 Signal</b>	<b>DE2-115 Connection</b>
<code>clk</code>	<code>KEY[0]</code>
<code>reset</code>	<code>KEY[1]</code> (must be held while pushing <code>KEY[0]</code> to reset)
<code>a</code>	<code>SW[0], LEDR[0]</code>
<code>state[1:0]</code>	<code>HEX0</code>
<code>x</code>	<code>LEDG[1]</code>
<code>y</code>	<code>LEDG[0]</code>

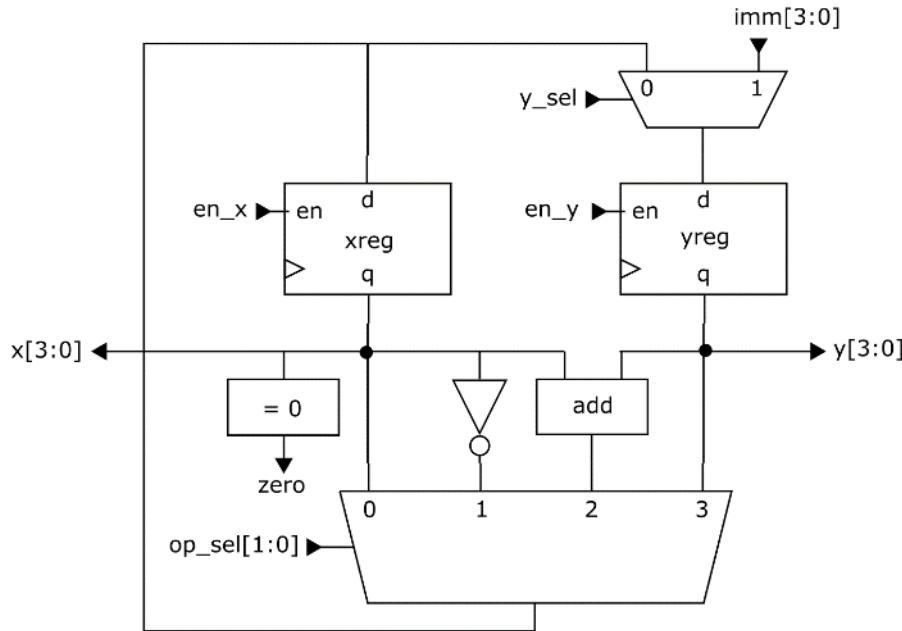
# 11 Register-Transfer Level (RTL) Design

## 11.1 A Simple Example

Example Quartus Project: `rtl_example`

### 11.1.1 Example Datapath Design

Consider the datapath below with inputs and outputs as follows:



Signal Type	Name	Description
Data In	imm[3:0]	immediate data
Data Out	x[3:0]	x
	y[3:0]	y
Control Points (input)	en_x	x register write enable
	en_y	y register write enable
	y_sel	y multiplexor select
	op[1:0]	operation multiplexor select
Flags (output)	zero	x == 0

Note that in order to load an immediate value `imm` into `x`, it must first be loaded into `y` and then transferred to `x`.

The Verilog code for the module `datapath` is as follows:

```
module datapath (
    input      clk,
    input [3:0] imm,
    input [1:0] op_sel,
```

```

input      en_x,
input      en_y,
input      y_sel,
output [3:0] x,
output [3:0] y,
output      zero
);

wire [3:0] d_y;
wire [3:0] f;

wire [3:0] op0 = x;
wire [3:0] op1 = ~x;
wire [3:0] op2 = x + y;
wire [3:0] op3 = y;

assign zero = (x == 4'h0);

reg4 xreg (
    .clk  (clk),
    .en   (en_x),
    .d    (f),
    .q    (x)
);

reg4 yreg (
    .clk  (clk),
    .en   (en_y),
    .d    (d_y),
    .q    (y)
);

mux2x4 y_mux (
    .d0  (f),
    .d1  (imm),
    .s   (y_sel),
    .y   (d_y)
);

mux4x4 op_mux (
    .d0  (op0),
    .d1  (op1),
    .d2  (op2),
    .d3  (op3),
    .s   (op_sel),
    .y   (f)
);

endmodule

```

## 11.1.2 Example: Calculating Additive Inverse

### 11.1.2.1 RTL Operations

As a first illustration of register-transfer operations with the example datapath, we use it to calculate the additive inverse of a number, add it to the original number, and verify that the result is 0. We do this with the following steps:

RTL Operation	y_sel	op_sel[1:0]	en_x	en_y	Description
$y \leftarrow 5$	1	0	0	1	load 5 into x (needs to be loaded into y then copied to x due to datapath)
$x \leftarrow y$	0	3	1	0	
$x \leftarrow \sim x$	0	1	1	0	replace x with 2's complement
$y \leftarrow 1$	1	0	0	1	additive inverse
$x \leftarrow x + y$	0	2	1	0	(x should contain $-4'd5$ or $4'ha$ )
$y \leftarrow 5$	1	0	0	1	load 5 into y
$x \leftarrow x + y$	0	2	1	0	replace x with x + y (x should contain 0, zero flag should be 1)

### 11.1.2.2 Manual Calculation Using Testbench

The following Verilog testbench implements the steps in the table above:

```

module datapath_tb ();
    reg      clk;
    reg [3:0] imm;
    reg [1:0] op_sel;
    reg      en_x;
    reg      en_y;
    reg      y_sel;
    wire [3:0] x;
    wire [3:0] y;
    wire      zero;

    datapath uut (
        .clk      (clk),
        .imm      (imm),
        .op_sel   (op_sel),
        .en_x     (en_x),
        .en_y     (en_y),
        .y_sel    (y_sel),
        .x        (x),
        .y        (y),
        .zero     (zero)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0;
    end

```

```

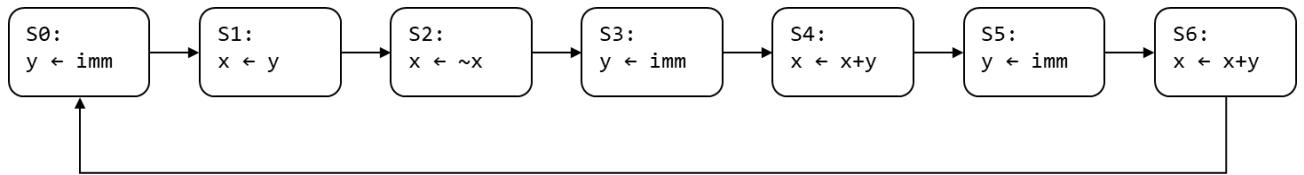
#10 imm = 5;  y_sel = 1;  op_sel = 0;  en_x = 0;  en_y = 1; // y <= 5
#10 imm = 0;  y_sel = 0;  op_sel = 3;  en_x = 1;  en_y = 0; // x <= y
#10 imm = 0;  y_sel = 0;  op_sel = 1;  en_x = 1;  en_y = 0; // x <= ~x
#10 imm = 1;  y_sel = 1;  op_sel = 0;  en_x = 0;  en_y = 1; // y <= 1
#10 imm = 0;  y_sel = 0;  op_sel = 2;  en_x = 1;  en_y = 0; // x <= x+y
#10 imm = 5;  y_sel = 1;  op_sel = 0;  en_x = 0;  en_y = 1; // y <= 5
#10 imm = 0;  y_sel = 0;  op_sel = 2;  en_x = 1;  en_y = 0; // x <= x+y
#10 $stop;
end

endmodule

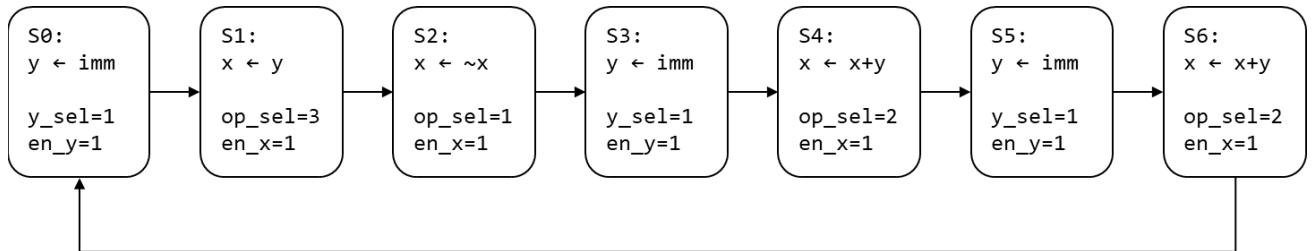
```

### 11.1.2.3 HLSM and FSM

The following high-level state machine (HLSM) describes the sequence of RTL operations for calculating an additive inverse and verifying the result. Note that the HLSM does not include the immediate values (`imm`) read as data inputs to the datapath—these will be supplied externally during the appropriate state.



We augment the HLSM with controller output values (datapath control point values) to obtain a finite-state machine (FSM).



Verilog code for the FSM is as follows:

```

module additive_inverse_fsm (
    input          clk,
    input          reset,
    output reg [1:0] op_sel,
    output reg      en_x,
    output reg      en_y,
    output reg      y_sel
);

parameter S0  = 3'd0;
parameter S1  = 3'd1;
parameter S2  = 3'd2;
parameter S3  = 3'd3;

```

```

parameter S4    = 3'd4;
parameter S5    = 3'd5;
parameter S6    = 3'd6;

reg [2:0] state, next_state;

always @(posedge clk)
  if (reset)
    state <= S0;
  else
    state <= next_state;

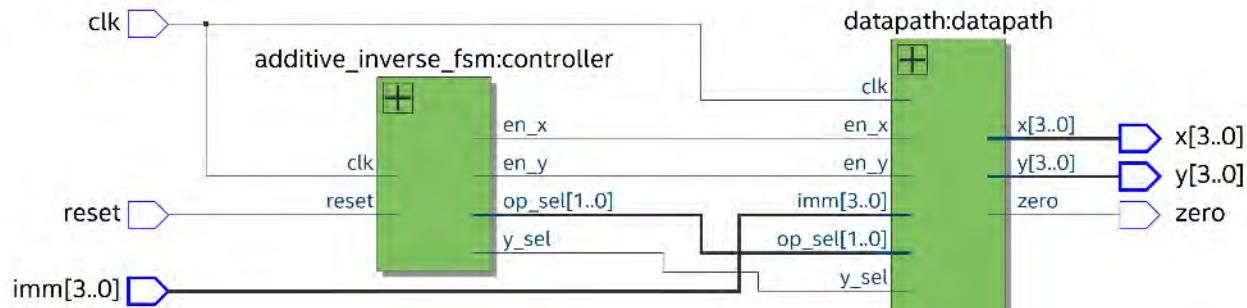
always @(*) begin
  op_sel    = 0;
  en_x      = 0;
  en_y      = 0;
  y_sel     = 0;
  case (state)
    S0: begin
      y_sel = 1;  en_y = 1;
      next_state = S1;
    end
    S1: begin
      op_sel = 3;  en_x = 1;
      next_state = S2;
    end
    S2: begin
      op_sel = 1;  en_x = 1;
      next_state = S3;
    end
    S3: begin
      y_sel = 1;  en_y = 1;
      next_state = S4;
    end
    S4: begin
      op_sel = 2;  en_x = 1;
      next_state = S5;
    end
    S5: begin
      y_sel = 1;  en_y = 1;
      next_state = S6;
    end
    S6: begin
      op_sel = 2;  en_x = 1;
      next_state = S0;
    end
    default
      next_state = S0;
  endcase
end

```

```
endmodule
```

#### 11.1.2.4 Complete Processor: Connecting FSM Controller to Datapath

The following is a schematic of the complete additive inverse processor consisting of the FSM controller connected to the datapath:



The Verilog code for the processor is as follows:

```
module additive_inverse_processor (
    input      clk,
    input      reset,
    input [3:0] imm,
    output [3:0] x,
    output [3:0] y,
    output      zero
);

    wire [1:0] op_sel;
    wire      en_x;
    wire      en_y;
    wire      y_sel;

    datapath datapath (
        .clk      (clk),
        .imm      (imm),
        .op_sel   (op_sel),
        .en_x    (en_x),
        .en_y    (en_y),
        .y Sel  (y Sel),
        .x       (x),
        .y       (y),
        .zero    (zero)
    );

    additive_inverse_fsm controller (
        .clk      (clk),
        .reset   (reset),
        .op_sel   (op_sel),
        .en_x    (en_x),
        .y Sel  (y Sel)
    );

```

```

        .en_y      (en_y),
        .y_sel     (y_sel)
    );
endmodule

```

### 11.1.2.5 Simulating Processor Behavior

The following testbench exercises the `additive_inverse_processor`. Values for the immediate input imm are provided as needed during the proper cycles.

```

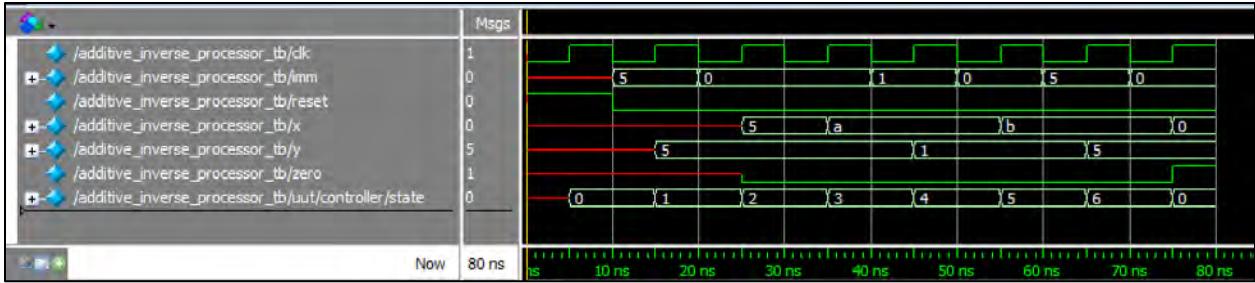
`timescale 1ns/1ns
module additive_inverse_processor_tb ();
    reg            clk;
    reg            reset;
    reg [3:0]      imm;
    wire [3:0]     x;
    wire [3:0]     y;
    wire           zero;

    additive_inverse_processor uut (
        .clk      (clk),
        .reset    (reset),
        .imm     (imm),
        .x       (x),
        .y       (y),
        .zero    (zero)
    );
    always #5 clk = ~clk;

    initial begin
        clk = 0;  reset = 1;
        #10 imm = 5; reset = 0; // S0
        #10 imm = 0;          // S1
        #10 imm = 0;          // S2
        #10 imm = 1;          // S3
        #10 imm = 0;          // S4
        #10 imm = 5;          // S5
        #10 imm = 0;          // S6
        #10 $stop;
    end
endmodule

```

Simulation results with the top-level ports and the controller `state` are shown below:



### 11.1.3 Example: Countdown

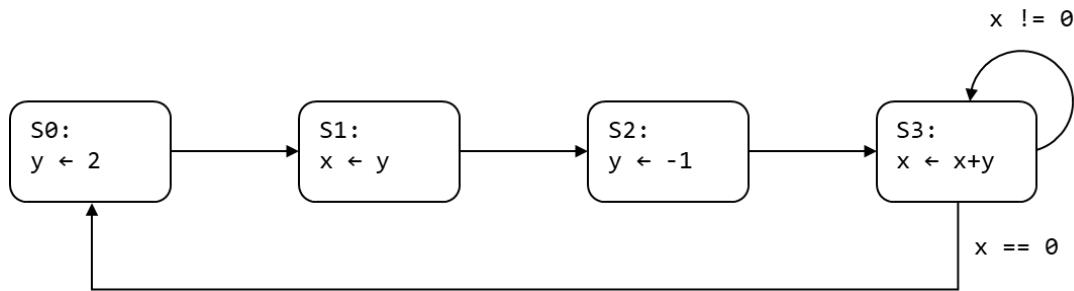
To illustrate looping and the use of a flag, as a second example we use the same datapath to perform a countdown procedure. A representation of the procedure in a programming language such as C would be:

```
while (1) {
    y = 2;
    x = y;
    y = -1
    while (x != 0)
        x = x + y;
    x = x + y;
}
```

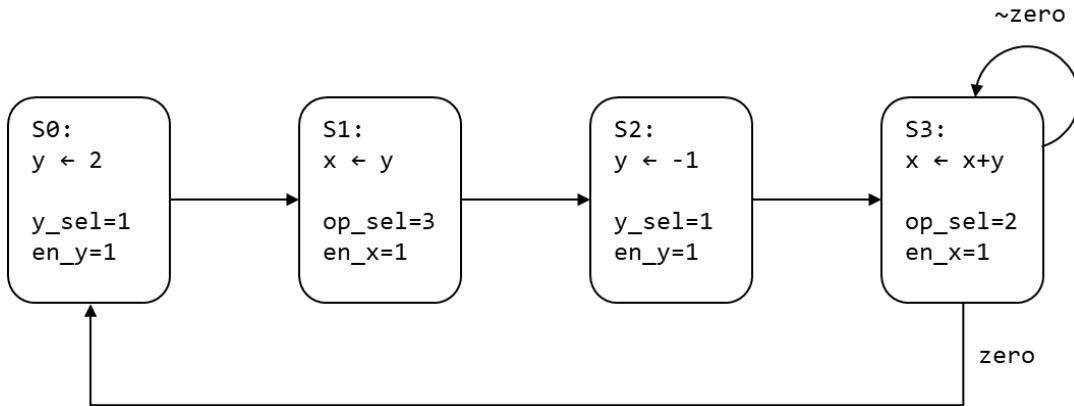
$x$  would thus count down through values 2, 1, 0, -1 and then repeat the process. Later we will show how to modify this so that  $x$  only counts down to 0 before starting over.

#### 11.1.3.1 HLSM and FSM

An HLSM for the countdown procedure is shown below:



To obtain the FSM for the controller from the HLSM, we add the controller output (datapath control point) values for each of the register-transfer operations at each state and replace the conditions that test if  $x == 0$  or  $x != 0$  with the zero flag from the datapath:



The Verilog code for the countdown FSM is as follows:

```

module countdown_fsm (
    input          clk,
    input          reset,
    input          zero,
    output reg [1:0] op_sel,
    output reg      en_x,
    output reg      en_y,
    output reg      y_sel
);

parameter S0    = 2'd0;
parameter S1    = 2'd1;
parameter S2    = 2'd2;
parameter S3    = 2'd3;

reg [1:0] state, next_state;

always @(posedge clk)
if (reset)
    state <= S0;
else
    state <= next_state;

always @(*) begin
    op_sel    = 0;
    en_x     = 0;
    en_y     = 0;
    y_sel    = 0;
    case (state)
        S0: begin
            y_sel = 1;  en_y = 1;
            next_state = S1;
        end
        S1: begin
            op_sel = 3;  en_x = 1;
            next_state = S2;
        end
        S2: begin
            y_sel = 1;  en_y = 1;
            next_state = S3;
        end
        S3: begin
            en_x = 1;
            if (zero)
                next_state = S0;
            else
                next_state = S3;
        end
    endcase
end

```

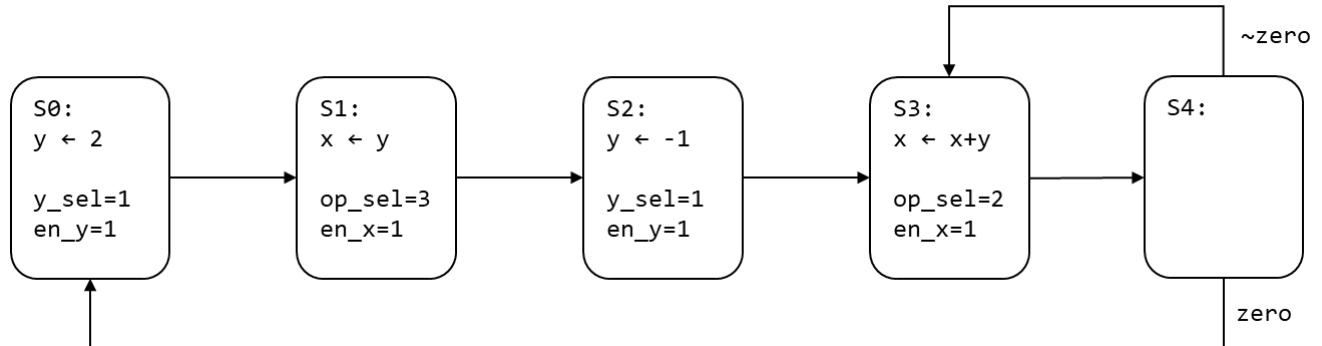
```

        next_state = S2;
    end
    S2: begin
        y_sel = 1;  en_y = 1;
        next_state = S3;
    end
    S3: begin
        op_sel = 2;  en_x = 1;
        if (zero)
            next_state = S0;
        else
            next_state = S3;
    end
    default
        next_state = S0;
    endcase
end

endmodule

```

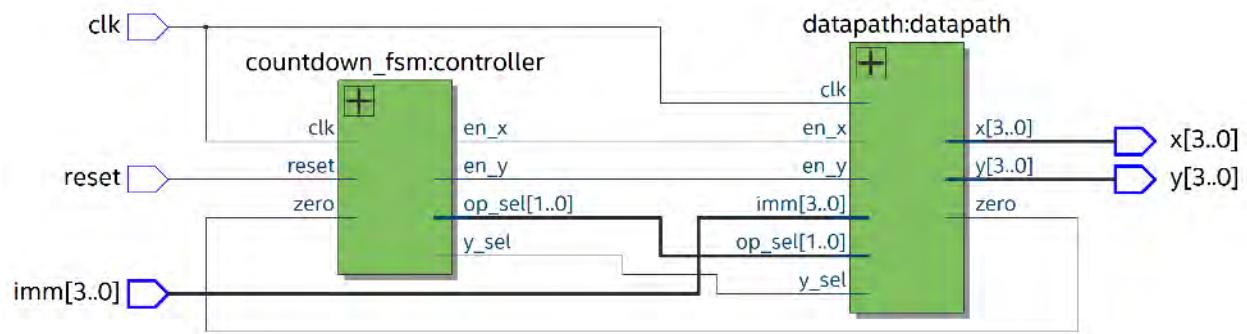
Note that when the system is in state S3, it will always execute the RTL operation  $x \leftarrow x+y$  on the next rising clock edge, regardless of the value of the `zero` flag. As a result,  $x$  will count down to -1. If we wanted  $x$  to stop decrementing at 0 using the same datapath, we could add another state to the HLSM and FSM that separates the decrement operation from the zero check.



Here, the new state S4 doesn't do any register writes—it just provides a state for testing whether  $x == 0$  after the RTL operation  $x \leftarrow x+y$  has completed in state S3. In this version, each iteration of the countdown goes through 2 states instead of just 1, meaning that the process will take longer to complete.

### 11.1.3.2 Complete Processor: Connecting FSM Controller to Datapath

The schematic for the complete processor with the controller and datapath, which now includes the zero flag from the datapath connected back to the controller is shown below:



The Verilog code is as follows:

```

module countdown_processor (
    input      clk,
    input      reset,
    input [3:0] imm,
    output [3:0] x,
    output [3:0] y
);

wire [1:0] op_sel;
wire      en_x;
wire      en_y;
wire      y_sel;
wire      zero;

datapath datapath (
    .clk      (clk),
    .imm      (imm),
    .op_sel   (op_sel),
    .en_x     (en_x),
    .en_y     (en_y),
    .y_sel    (y_sel),
    .x        (x),
    .y        (y),
    .zero     (zero)
);

countdown_fsm controller (
    .clk      (clk),
    .reset    (reset),
    .zero    (zero),
    .op_sel   (op_sel),
    .en_x     (en_x),
    .en_y     (en_y),
    .y_sel    (y_sel)
);

```

```
endmodule
```

### 11.1.3.3 Simulating Processor Behavior

The following is a Verilog testbench for the countdown processor

```
module countdown_processor_tb ();
    reg             clk;
    reg             reset;
    reg [3:0]       imm;
    wire [3:0]      x;
    wire [3:0]      y;

    countdown_processor uut (
        .clk     (clk),
        .reset   (reset),
        .imm    (imm),
        .x      (x),
        .y      (y)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0;  reset = 1;
        #10 imm = 2; reset = 0; // S0
        #10 imm = 0;           // S1
        #10 imm = -1;          // S2
        #10 imm = 0;           // S3
        #30 $stop;            // run 3 more iterations of S3 before stopping
    end

endmodule
```

Simulation results with the top-level ports, zero flag, and the controller state are shown below:

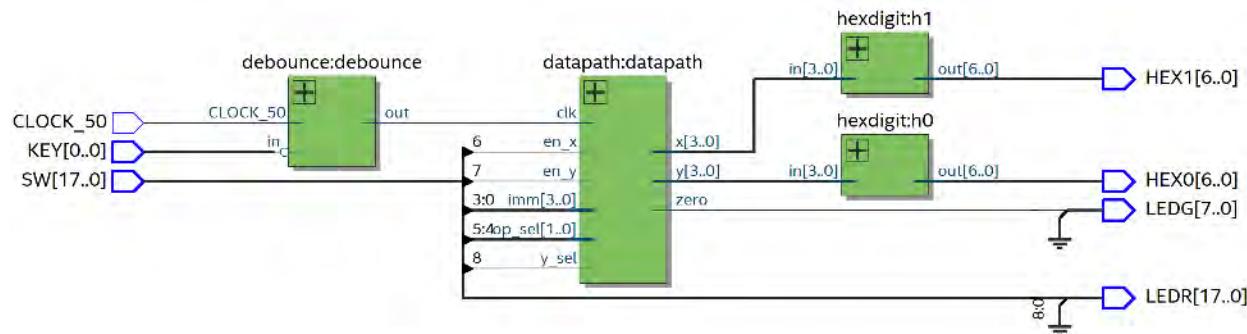


## 11.1.4 DE2-115 Board Single-Step Demonstration

### 11.1.4.1 Single-Stepping Datapath Operation

The module `datapath_DE2` connects the ports of module `datapath` to switches, buttons, LEDs, and 7-segment displays on the DE2-115 board. The `datapath` clock `clk` is controlled by the pushbutton `KEY[0]`, so that you can “single-step” through the operation of `datapath` by setting switches for the control points and then pressing the pushbutton to produce a clock edge that effects the register-transfer operation.

Because of mechanical bouncing that can cause the pushbutton to produce several pulses when it is pushed, `KEY[0]` is debounced using module `debounce` to produce a clean clock signal. (See the section on Debouncing in this document).



The inputs and outputs of the module is as follows:

Datapath Port	DE2-115 Connection
<i>inputs</i>	
<code>clk</code>	<code>KEY[0]</code> (through debouncer)
<code>imm[3:0]</code>	<code>SW[3:0]</code>
<code>op_sel[1:0]</code>	<code>SW[5:4]</code>
<code>en_x</code>	<code>SW[6]</code>
<code>en_y</code>	<code>SW[7]</code>
<code>y_sel</code>	<code>SW[8]</code>
<i>outputs</i>	
<code>x[3:0]</code>	<code>HEX1</code>
<code>y[3:0]</code>	<code>HEX0</code>
<code>zero</code>	<code>LEDG[0]</code>

### 11.1.4.2 Single-Stepping Countdown Processor Operation

Module `countdown_processor_DE2` connects the inputs, outputs, control signals, and state register of the `countdown_processor` to buttons, switches, LEDs, and 7-segment displays of the `countdown_processor` so you can single-step through its operation. The connections between the processor and the board are as follows:

Processor Signal	DE2-115 Connection
<i>inputs</i>	
<code>clk</code>	<code>KEY[0]</code> (through debouncer)

<b>reset</b>	KEY[1] (must be held down while KEY[0] is pressed to reset)
<b>imm[3:0]</b>	SW[3:0]
<i>control points</i>	
<b>op_sel[1:0]</b>	LEDG[7:6]
<b>en_x</b>	LEDG[5]
<b>en_y</b>	LEDG[4]
<b>y_sel</b>	LEDG[3]
<i>flags</i>	
<b>zero</b>	LEDG[0]
<i>controller state</i>	
<b>state[1:0]</b>	HEX2 (uses port <b>state_disp</b> added to controller for display purposes)
<i>data output</i>	
<b>x[3:0]</b>	HEX1
<b>y[3:0]</b>	HEX0

To run the demonstrate, enter appropriate values for **imm** as required for each state, click **KEY[0]** to clock to the next state, and watch the outputs on the LEDs and 7-segment displays. Required inputs for **imm** on SW[3:0] are:

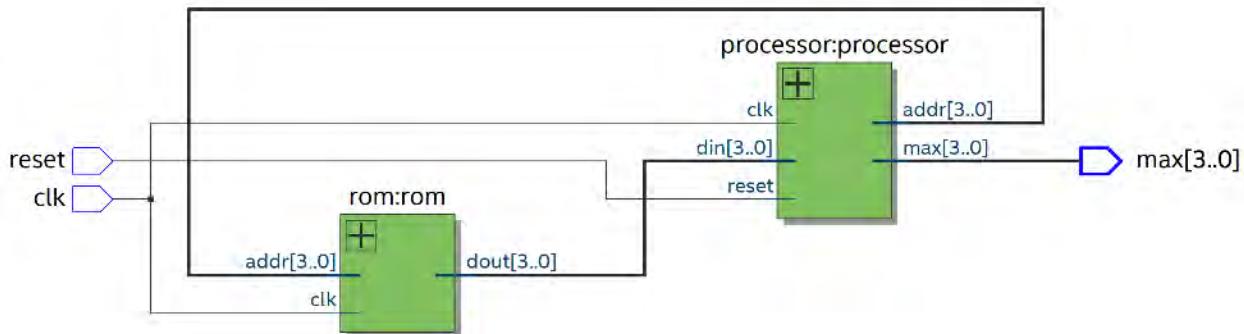
- state 0: initial value for decrementing (2 in the example)
- state 1: doesn't matter
- state 2: -1 (4'hF)
- state 3: doesn't matter

## 11.2 Memory Interface: Maxfinder

Example Quartus Project: maxfinder

### 11.2.1 Overview and HLSM

This example illustrates the design of a processor that interacts with memory to determine the maximum value in the memory. The overall system is illustrated below:



The system has two components: a ROM that contains the data values and a processor connected to the memory. The processor has the following ports:

Port	Description
clk	clock
reset	system reset
addr[3:0]	address sent to memory
din[3:0]	data read from memory
max[3:0]	running value of maximum

The ROM is a synchronous memory that can be represented as an array:

`rom[addr]`

Because the ROM is synchronous, the processor must wait a cycle after an address is asserted before the data out is available.

The processor reads each of the memory addresses in succession and if the current value is greater than the current maximum value, it resets the maximum value. The process continues until the maximum memory address is reached. The algorithm is described as a HLSM in table form below:

state	actions	transitions
INIT	<code>addr &lt;- 0</code> <code>max &lt;- 0</code>	<code>goto READ_MEM</code>
READ_MEM	<code>din &lt;- rom[addr]</code>	<code>goto CHECK_MAX</code>
CHECK_MAX		<code>if (din &gt; max) goto UPDATE_MAX</code> <code>else goto CHECK_LAST_ADDRESS</code>

UPDATE_MAX	$\max \leftarrow \text{din}$	goto CHECK_LAST_ADDRESS
CHECK_LAST_ADDRESS	$\text{addr} \leftarrow \text{addr} + 1$	if ( $\text{addr} == \text{LASTADDR}$ ) goto END else goto READ_MEM
END		goto END

Given the HLSM, the next steps are to determine:

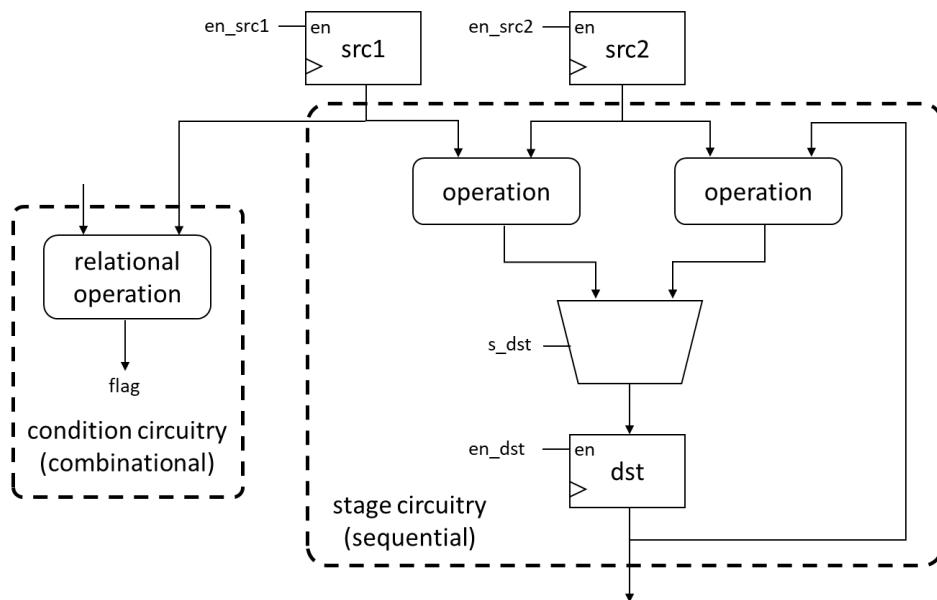
- a datapath that supports each of the individual RTL operations and that produces the conditions for transitions
- FSM controller

## 11.2.2 Datapath Design

### 11.2.2.1 Breaking Datapath Down to Stages and Conditions

One approach to designing a datapath that supports the required RTL operations and that produces the required conditions is as follows:

- **RTL operations:** break the overall datapath into *stages*, where each stage consists of a destination register and the combinational logic that feeds it. Further, we can think of the combinational logic as having two parts: arithmetic or logic operations between the outputs of some source registers (from other stages or the destination of this state), followed by a multiplexor that selects which result should feed the destination register. Since it contains both combinational logic and a register, each stage is a sequential circuit.
- **Conditions:** express each condition as a relational operation involving one or more registers. The condition circuitry is combinational logic.



Note that this approach to RTL design doesn't necessarily yield the most efficient design. Since each stage is treated separately, it could result in the duplication of components that might otherwise be shared between operations. It also doesn't take into consideration combinational logic delays that might cause a particular stage to be very slow and hence limit the clock rate. Nevertheless, this is a good, methodical

approach when circuit size and speed aren't much of a concern, as is the case with the FPGA designs for this course.

### 11.2.2.2 Stage Design

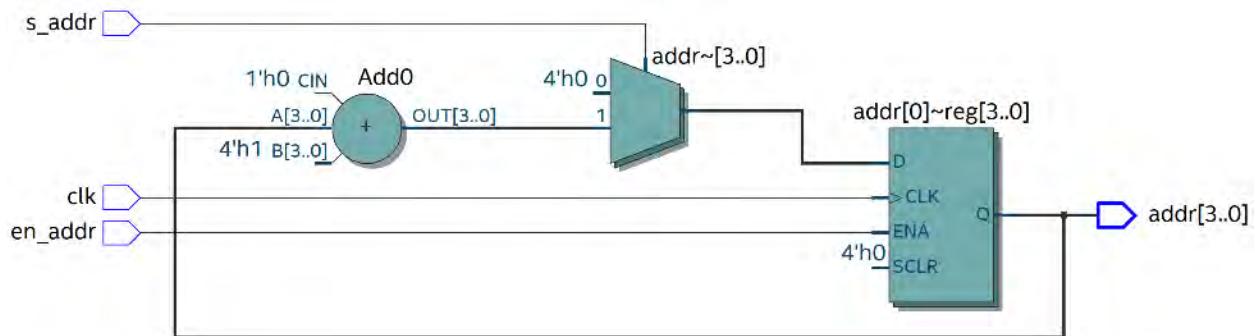
The first step in datapath stage design is to list the stages by destination register. Then for each stage, enumerate the operations that feed it. Each stage can then be implemented as a selection between operations. By convention, if the name of the destination register is `dst`, we name the enable signal for the register `en_dst` and the select input for the multiplexor that feeds it `s_dst`. The number before each source indicates the value of the select line `s_dst` to select that source. These enable and select signals will be inputs to the datapath provided by the controller FSM. Analysis of the HLSM for the Maxfinder yields the following stages with associated control signals:

destination	sources	control signals
addr	0: 0 1: <code>addr + 1</code>	<code>en_addr</code> <code>s_addr</code>
din	<code>rom[addr]</code>	
max	0: 0 1: <code>din</code>	<code>en_max</code> <code>s_max</code>

At this point, we could implement each stage in Verilog as a structural composition of components for arithmetic/logic operations, a multiplexor, and a register. A much more convenient approach, however, is to write a single Verilog block that captures the behavior of the stage as a whole. For example, the following `always` block describes the behavior of the `addr` stage:

```
always @(posedge clk)
  if (en_addr)
    if (s_addr)
      addr <= addr + 1;
    else
      addr <= 0;
```

Note that this synthesizes as an adder, a multiplexor, and a register:



### 11.2.2.3 Condition Logic

After the stages are defined, the next step in the datapath design is to enumerate all of the conditions on transitions in the HLSM and define flags for each. These flags will be datapath outputs that feed back to the controller.

conditions	flag
din > max	din_gt_max
addr == LASTADDR	addr_eq_last

The flag logic can generally be implemented in Verilog as assign statements:

```
assign din_gt_max      = din > max;
assign addr_eq_last    = addr == LASTADDR;
```

### 11.2.2.4 Complete Datapath

The only tasks remaining before completing the datapath design is to determine the port interface and to declare registers as either associated with ports or as internal. The port interface for the datapath consists of the following:

- System clock
  - clk
- Control inputs (select lines and register write enables)
  - s\_addr, en\_addr
  - s\_max, en\_max
- Flag outputs
  - din\_gt\_max
  - addr\_eq\_last
- External/processor data outputs
  - addr
  - max
- External/processor data inputs
  - din (from memory)

Thus of the stage registers, `addr` and `max` will be declared with the ports, and there are no internal registers. The complete Verilog model of the datapath is as follows:

```
module datapath (
    input          clk,
    input          en_addr,
    input          en_max,
    input          s_addr,
    input          s_max,
    input [3:0]    din,
    output reg [3:0] addr,
    output reg [3:0] max,
    output         din_gt_max,
    output         addr_eq_last
);
```

```

);

/*****************
 *          Parameter Declarations
 */
parameter LASTADDR = 4'hf;

/*****************
 *          Internal Wires and Registers Declarations
 */
// Internal Wires
// Internal Registers

/*****************
 *          Sequential Logic
 */
always @(posedge clk)
  if (en_addr)
    if (s_addr)
      addr <= addr + 1;
    else
      addr <= 0;

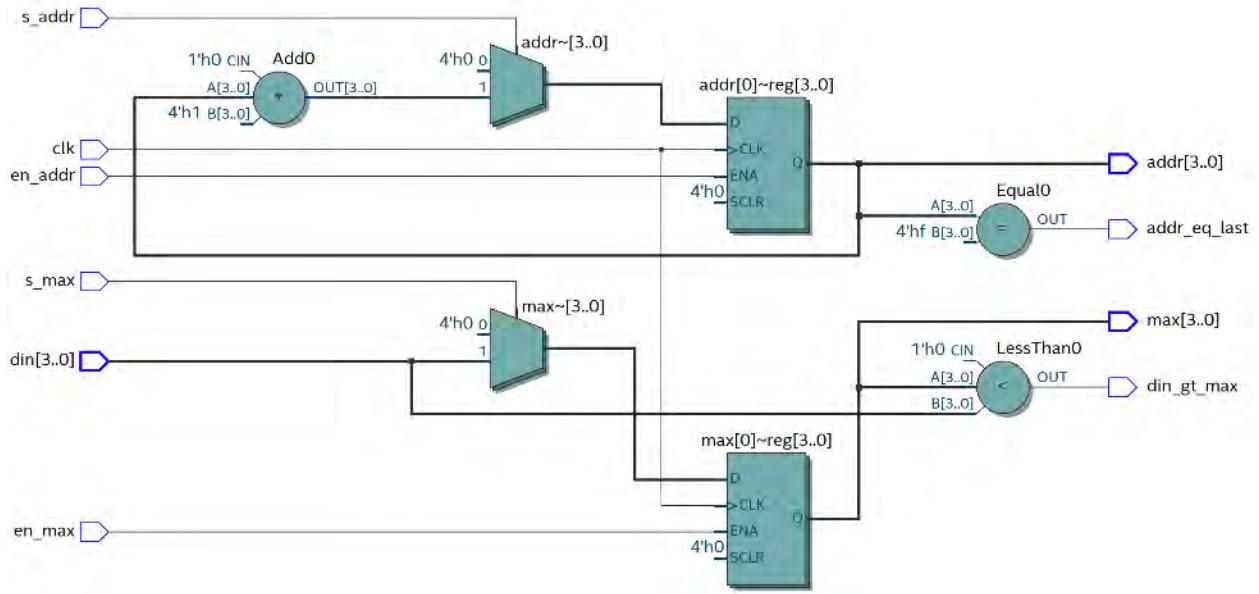
always @(posedge clk)
  if (en_max)
    if (s_max)
      max <= din;
    else
      max <= 0;

/*****************
 *          Combinational Logic
 */
assign din_gt_max    = din > max;
assign addr_eq_last  = addr == LASTADDR;

/*****************
 *          Internal Modules
 */
endmodule

```

The RTL netlist of the synthesized datapath is shown below:



### 11.2.3 Controller Design

Given the complete datapath design, we transform the HLSM to an FSM for the controller as follows:

- Specify control point values (select and enable signals) for each state as controller outputs to enact the appropriate RTL operations in the datapath
- Use flag values from the datapath as controller inputs to specify transition conditions

We add this information to the HLSM table from above:

state	actions	transitions
INIT	$\text{addr} \leftarrow 0$ $\text{max} \leftarrow 0$  $\text{s\_addr} = 0; \text{en\_addr} = 1;$ $\text{s\_max} = 0; \text{en\_max} = 1;$	goto READ_MEM
READ_MEM	$\text{din} \leftarrow \text{rom}[\text{addr}]$	goto CHECK_MAX
CHECK_MAX		if ( $\text{din}_{gt\_max}$ ) goto UPDATE_MAX else goto CHECK_LAST_ADDRESS
UPDATE_MAX	$\text{max} \leftarrow \text{din}$  $\text{s\_max} = 1; \text{en\_max} = 1;$	goto CHECK_LAST_ADDRESS
CHECK_LAST_ADDRESS	$\text{addr} \leftarrow \text{addr} + 1$  $\text{s\_addr} = 1; \text{en\_addr} = 1;$	if ( $\text{addr}_{eq\_last}$ ) goto END else goto READ_MEM

END		goto END
-----	--	----------

Given this table, implementation of the FSM in Verilog is straightforward. The complete Verilog model for the controller is given below:

```

module controller (
    input      clk,
    input      reset,
    output reg en_addr,
    output reg en_max,
    output reg s_addr,
    output reg s_max,
    input      din_gt_max,
    input      addr_eq_last
);

parameter INIT          = 3'd0;
parameter READ_MEM       = 3'd1;
parameter CHECK_MAX      = 3'd2;
parameter UPDATE_MAX     = 3'd3;
parameter CHECK_LAST_ADDR = 3'd4;
parameter END            = 3'd5;

reg [2:0] state, next_state;

always @(posedge clk)
    if (reset)
        state <= INIT;
    else
        state <= next_state;

always @(*) begin
    en_addr = 0;
    en_max = 0;
    s_addr = 0;
    s_max = 0;
    next_state = INIT;
    case (state)
        INIT: begin
            en_addr = 1;
            en_max = 1;
            next_state = READ_MEM;
        end
        READ_MEM:      begin
            next_state = CHECK_MAX;
        end
        CHECK_MAX:    begin
            if (din_gt_max)
                next_state = UPDATE_MAX;
        end
    endcase
end

```

```

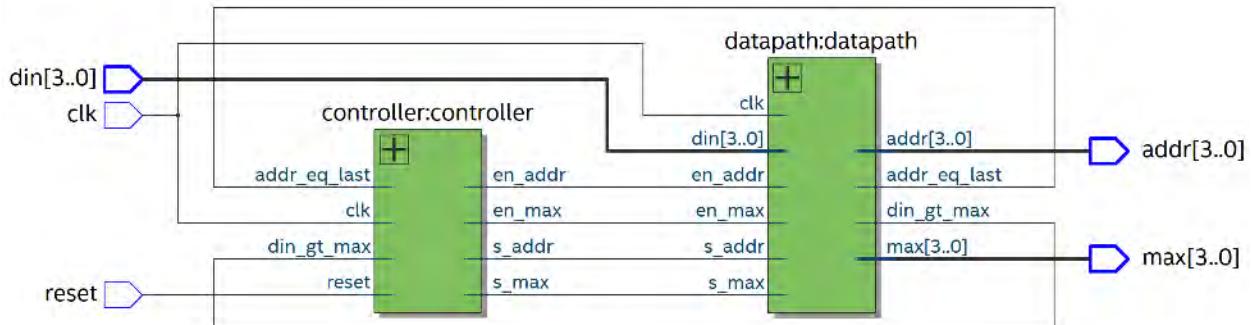
        else
            next_state = CHECK_LAST_ADDR;
    end
    UPDATE_MAX:      begin
        en_max = 1;
        s_max = 1;
        next_state = CHECK_LAST_ADDR;
    end
    CHECK_LAST_ADDR: begin
        en_addr = 1;
        s_addr = 1;
        if (addr_eq_last)
            next_state = END;
        else
            next_state = READ_MEM;
    end
    END:           begin
        next_state = END;
    end
    default: ;
endcase
end

endmodule

```

#### 11.2.4 Complete Processor: Connecting Controller and Datapath

The schematic and Verilog model for the complete processor with controller and datapath is given below:



```

module processor (
    input          clk,
    input          reset,
    input  [3:0] din,
    output [3:0] addr,
    output [3:0] max
);

wire en_addr;
wire en_max;

```

```

wire s_addr;
wire s_max;
wire din_gt_max;
wire addr_eq_last;

controller controller (
    .clk          (clk),
    .reset        (reset),
    .en_addr     (en_addr),
    .en_max      (en_max),
    .s_addr      (s_addr),
    .s_max       (s_max),
    .din_gt_max (din_gt_max),
    .addr_eq_last (addr_eq_last)
);

datapath datapath (
    .clk          (clk),
    .en_addr     (en_addr),
    .en_max      (en_max),
    .s_addr      (s_addr),
    .s_max       (s_max),
    .din         (din),
    .addr        (addr),
    .max         (max),
    .din_gt_max (din_gt_max),
    .addr_eq_last (addr_eq_last)
);

endmodule

```

### 11.2.5 Complete System: Connecting Processor to Memory

The Verilog model for the complete system connecting the processor to the ROM, along with the model for the ROM itself, is given below:

```

module maxfinder (
    input      clk,
    input      reset,
    output [3:0] max
);

wire [3:0] memdata;
wire [3:0] addr;

processor processor (
    .clk      (clk),
    .reset    (reset),
    .din     (memdata),
    .addr    (addr),
    .max     (max)

```

```

);
rom rom (
    .clk      (clk),
    .addr     (addr),
    .dout    (memdata)
);
endmodule

module rom (
    input          clk,
    input [3:0]    addr,
    output reg [3:0] dout
);

reg [3:0] mem [0:15];
initial begin
    mem[0]  = 0;
    mem[1]  = 1;
    mem[2]  = 0;
    mem[3]  = 7;
    mem[4]  = 0;
    mem[5]  = 0;
    mem[6]  = 0;
    mem[7]  = 0;
    mem[8]  = 0;
    mem[9]  = 5;
    mem[10] = 0;
    mem[11] = 0;
    mem[12] = 10;
    mem[13] = 0;
    mem[14] = 0;
    mem[15] = 0;
end

always @(posedge clk)
dout <= mem[addr];
endmodule

```

### 11.2.6 Testbench and Simulation Results

A testbench for the Maxfinder system is given below. It takes advantage of what is known as a *hierarchical signal* in Verilog to determine when the processor has reached the END state. Hierarchical signals let you tap into external signals by descending the instance hierarchy, where instance names (not module names) at each level are separated by dots. Note that hierarchical signals can only be used in testbenches—they won’t synthesize. The initial block uses a while loop to check when the `state` signal in the controller equals END (`4'd5`), incrementing the simulation time by `#10` until it does.

```

module maxfinder_tb ();
    reg          clk;
    reg          reset;
    wire [3:0]   max;

    maxfinder uut (
        .clk      (clk),
        .reset    (reset),
        .max     (max)
    );
    always #5 clk = ~clk;

    initial begin
        clk = 0;  reset = 1;
        #10 reset = 0;
        while (uut.processor.controller.state != 3'd5)
            #10;
        #10 $stop;
    end
endmodule

```

The simulation result below shows how the simulation ends one cycle after the controller reaches the END state where `uut/processor/controller/state` equals 5.

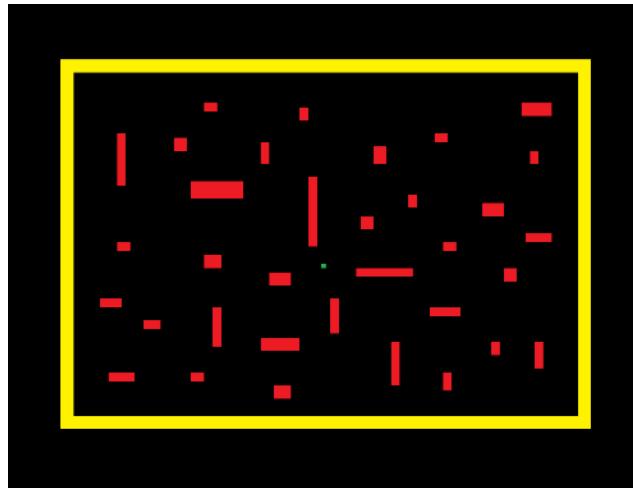


## 11.3 Animation Example: Obstacle Course

Example Quartus Project: obstacle

### 11.3.1 Overview and HLSM

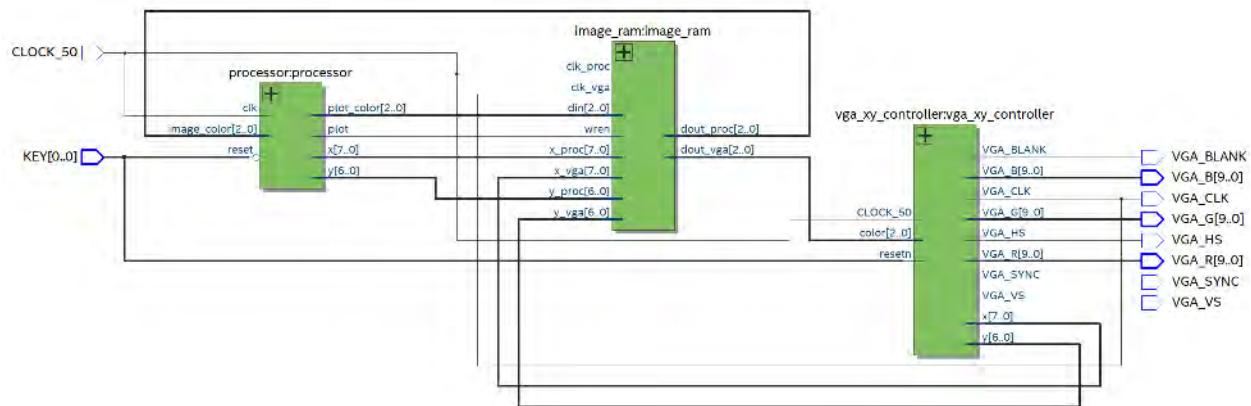
As a more complex example, we examine the design of a processor that animates the motion of a particle through a set of obstacles, where the particle moves in a diagonal path and “bounces off” the obstacles and borders when it hits them. The display is illustrated below:



This example uses the same basic RTL design methodology used in the Maxfinder, but introduces several new considerations:

- interfacing with the VGA controller `vga_xy_controller` for the DE2-115 board
- interfacing with an image memory with 2 read ports and 1 write port, where 1 read port and 1 write port is used by the processor and the other read port is used by the `vga_xy_controller`

The schematic below shows the top-level system with the processor, VGA controller, and connections to the DE2-115 input/output pins. A complete description of the VGA controller is provided in the Chapter on the DE2-115 board.



The processor has the following ports:

Port	Direction	Description
CLOCK_50	input	DE2-115 50 MHz clock
reset	input	system reset
x[7:0]	input	x-coordinate of pixel in image memory
y[6:0]	input	y-coordinate of pixel in image memory
image_color[2:0]	input	3-bit RGB pixel color read from memory
plot_color[2:0]	output	3-bit RGB pixel color written to memory
plot	output	write enable to plot pixel in image memory

The basic algorithm implemented by the processor is as follows:

1. Initialize the position and direction of the particle, as well as initialize a timer that is used to control the speed of the animation
2. Wait for the animation speed timer
3. “Erase” the pixel from its current position by redrawing it as black (the background color)
4. Depending on the direction of horizontal motion, read the pixels in the image memory to the left or right of the current position to see if there is an obstacle in front of the particle.
5. If there is an obstacle in front of it, reverse the direction of horizontal motion.
6. Similarly, depending on the direction of vertical motion, read the pixels in the obstacle memory above or below the current position.
7. If there is an obstacle in front of it, reverse the direction of vertical motion.
8. Update the position of the particle by 1 horizontal unit and 1 vertical unit in the directions of motion.
9. Plot the particle as a green pixel on the VGA display

The high-level state machine for the Obstacle application will use the following variables and component interfaces:

xpos ypos	coordinates of current position of particle
xdir ydir	current horizontal and vertical direction of particle (0 = decreasing position, 1 = increasing position.)
timer	used to count delay cycles that control animation rate
plot	write enable to plot pixel on VGA display
image(x, y)	interface to image memory
image_out	register at read output of image memory

The high-level state machine (HLSM) is presented below in table form:

state	actions	transitions
INIT	xdir $\leftarrow$ init_xdir ydir $\leftarrow$ init_ydir xpos $\leftarrow$ init_xpos ypos $\leftarrow$ init_ypos timer $\leftarrow$ 0	goto WAIT_TIMER
WAIT_TIMER	timer $\leftarrow$ timer+1	if (timer == TIME_LIMIT) goto ERASE else goto WAIT_TIMER
ERASE	image(xpos,ypos) $\leftarrow$ BLACK timer $\leftarrow$ 0	if (xdir == RIGHT) goto LOOK_RIGHT else goto LOOK_LEFT
LOOK_LEFT	image_out $\leftarrow$ image(xpos-1,ypos)	goto TEST_X_OBSTACLE

LOOK_RIGHT	<code>image_out ← image(xpos+1,ypos)</code>	<code>goto TEST_X_OBSTACLE</code>
TEST_X_OBSTACLE		<code>if (image_out != BLACK) goto CHANGE_XDIR else if (ydir == DOWN) goto LOOK_DOWN else goto LOOK_UP</code>
CHANGE_XDIR	<code>xdir ← ~xdir</code>	<code>if (ydir == DOWN) goto LOOK_DOWN else goto LOOK_UP</code>
LOOK_UP	<code>image_out ← image(xpos,ypos-1)</code>	<code>goto TEST_Y_OBSTACLE</code>
LOOK_DOWN	<code>image_out ← image(xpos,ypos+1)</code>	<code>goto TEST_Y_OBSTACLE</code>
TEST_Y_OBSTACLE		<code>if (image_out != BLACK) goto CHANGE_YDIR else if (xdir == RIGHT) goto INCREMENT_XPOS else goto DECREMENT_XPOS</code>
CHANGE_YDIR	<code>ydir ← ~ydir</code>	<code>if (xdir == RIGHT) goto INCREMENT_XPOS else goto DECREMENT_XPOS</code>
DECREMENT_XPOS	<code>xpos ← xpos-1</code>	<code>if (ydir == DOWN) goto INCREMENT_YPOS else goto DECREMENT_YPOS</code>
INCREMENT_XPOS	<code>xpos ← xpos+1</code>	<code>if (ydir == DOWN) goto INCREMENT_YPOS else goto DECREMENT_YPOS</code>
DECREMENT_YPOS	<code>ypos ← ypos-1</code>	<code>goto DRAW</code>
INCREMENT_YPOS	<code>ypos ← ypos+1</code>	<code>goto DRAW</code>
DRAW	<code>image(xpos, ypos) ← green</code>	<code>goto WAIT_TIMER</code>

For the most part, the HLSM follows the same conventions as Maxfinder example with a few notable differences in several of the states:

- The processor writes to the image memory in states ERASE and DRAW.
- The processor reads a 3-bit pixel color from the image memory in states LOOK\_LEFT, LOOK\_RIGHT, LOOK\_UP, and LOOK\_DOWN. The image memory has *synchronous reads*, meaning that data appears at the memory output `image_out` following the next rising clock edge. As a result, we need to wait until the next states TEST\_X\_OBSTACLE or TEST\_Y\_OBSTACLE before we use the memory data to determine if there is an obstacle present. Note that this is different from the behavior of the ROM in the Maxfinder example that had asynchronous reads, where the output data appears in the same cycle that the address is presented to the memory, after a combinational logic delay that is shorter than a clock cycle.

### 11.3.2 Datapath Design

Datapath design follows the same basic process that was used for the Maxfinder—please see that example for a complete description of the methodology. The sections below for the obstacle processor present the relevant tables with minimal explanation.

#### 11.3.2.1 Stage Design

Analysis of the HLSM for the obstacle processor yields the following stages with associated control signals:

destination	sources	control signals
xdir	0: init_xdir 1: ~xdir	en_xdir s_xdir
ydir	0: init_ydir 1: ~ydir	en_ydir s_ydir
xpos	0: init_xpos 1: xpos-1	en_xpos s_xpos

	2: xpos+1	
ypos	0: init_ypos 1: ypos-1 2: ypos+1	en_ypos s_ypos
timer	0: 0 1: timer+1	en_timer s_timer
image_color	0: image(xpos,ypos-1) 1: image(xpos,ypos+1) 2: image(xpos-1,ypos) 3: image(xpos+1,ypos)	s_x s_y
plot_color	0: BLACK 1: GREEN	s_plot_color plot

### 11.3.2.2 Condition Logic

The HLSM contains the following conditions that must correspond to flags produced by the datapath:

condition	flag
timer == TIME_LIMIT	timer_done
xdir == RIGHT	xdir
ydir == DOWN	ydir
image_color != BLACK	obstacle

### 11.3.2.3 Complete Datapath

The Verilog model for the complete datapath is given below.

```
module datapath (
    input          clk,
    input          en_xpos,
    input [1:0]    s_xpos,
    input          en_ypos,
    input [1:0]    s_ypos,
    input          en_xdir,
    input          s_xdir,
    input          en_ydir,
    input          s_ydir,
    input          en_timer,
    input          s_timer,
    input          s_image_color,
    input [1:0]    s_x,
    input [1:0]    s_y,
    output reg    xdir,
    output reg    ydir,
    output        timer_done,
    output        obstacle,
    output [7:0]   x,
    output [6:0]   y,
    output [2:0]   plot_color,
```

```

input      [2:0]  image_color,
input          plot
);

/*
 *                               Parameter Declarations
 */
parameter TIMER_LIMIT = 26'd1_000_000;
// parameter TIMER_LIMIT = 26'd2;

/*
 *                               Internal Wire and Register Declarations
 */
reg  [25:0]  timer;
reg  [7:0]    xpos;
reg  [6:0]    ypos;

/*
 *                               Sequential Logic
 */
always @(posedge clk)
  if (en_xdir)
    if (s_xdir)
      xdir <= ~xdir;
    else
      xdir <= 1;

always @(posedge clk)
  if (en_ydir)
    if (s_ydir)
      ydir <= ~ydir;
    else
      ydir <= 1;

always @(posedge clk)
  if (en_xpos)
    case (s_xpos)
      0: xpos <= 8'd80;
      1: xpos <= xpos - 1;
      2: xpos <= xpos + 1;
      default: xpos <= 0;
    endcase

always @(posedge clk)
  if (en_ypos)
    case (s_ypos)
      0: ypos <= 7'd60;
      1: ypos <= ypos - 1;
      2: ypos <= ypos + 1;

```

```

        default: ypos <= 0;
    endcase

    always @(posedge clk)
        if (en_timer)
            if (s_timer)
                timer <= timer + 1;
            else
                timer <= 0;

//*****************************************************************************
/*
 *          Combinational Logic
 */
//****************************************************************************/
// obstacle memory coordinate addresses
assign x =
    s_x == 0 ? xpos :
    s_x == 1 ? xpos - 1 :
    xpos + 1;

assign y =
    s_y == 0 ? ypos :
    s_y == 1 ? ypos - 1 :
    ypos + 1;

// pixel color to VGA adapter
assign plot_color = s_image_color ? 3'b010 : 3'b000;

// flags
assign timer_done = timer == TIMER_LIMIT;
assign obstacle    = image_color != 3'b000;

endmodule

```

The module `image_ram` uses an Altera library of parameterized modules component, `lpm_ram_dq`. The file is included in the project but are not shown here.

### 11.3.3 Controller Design

The following table adds control signals and condition flags to the HLSM table from above:

state	actions	transitions
INIT	<code>xdir &lt;- init_xdir</code> <code>ydir &lt;- init_ydir</code> <code>xpos &lt;- init_xpos</code> <code>ypos &lt;- init_ypos</code> <code>timer &lt;- 0</code>  <code>s_xdir = 0; en_xdir = 1;</code> <code>s_ydir = 0; en_ydir = 1;</code>	<code>goto WAIT_TIMER</code>

	<code>s_xpos = 0; en_xpos = 1; s_ypos = 0; en_ypos = 1; s_timer = 0; en_timer = 1;</code>	
WAIT_TIMER	<code>timer ← timer+1  s_timer = 1; en_timer;</code>	<code>if(timer_done) goto ERASE else goto WAIT_TIMER</code>
ERASE	<code>image(xpos,ypos) ← BLACK timer ← 0  plot = 1; s_plot_color = 0; s_timer = 0; en_timer = 1;</code>	<code>if (xdir) goto LOOK_RIGHT else goto LOOK_LEFT</code>
LOOK_LEFT	<code>image_color ← image(xpos-1,ypos)  s_x = 1; s_y = 0;</code>	<code>goto TEST_X_OBSTACLE</code>
LOOK_RIGHT	<code>image_color ← image(xpos+1,ypos)  s_x = 2; s_y = 0;</code>	<code>goto TEST_X_OBSTACLE</code>
TEST_X_OBSTACLE		<code>if (obstacle) goto CHANGE_XDIR else if (ydir) goto LOOK_DOWN else goto LOOK_UP</code>
CHANGE_XDIR	<code>xdir ← ~xdir  s_xdir = 1; en_xdir;</code>	<code>if (ydir) goto LOOK_DOWN else goto LOOK_UP</code>
LOOK_UP	<code>image_color ← image(xpos,ypos-1)  s_x = 0; s_y = 1;</code>	<code>goto TEST_Y_OBSTACLE</code>
LOOK_DOWN	<code>image_color ← image(xpos,ypos+1)  s_x = 0; s_y = 2;</code>	<code>goto TEST_Y_OBSTACLE</code>
TEST_Y_OBSTACLE		<code>if (obstacle) goto CHANGE_YDIR else if (xdir) goto INCREMENT_XPOS else goto DECREMENT_XPOS</code>
CHANGE_YDIR	<code>ydir ← ~ydir  s_ydir = 1; en_ydir;</code>	<code>if (xdir) goto INCREMENT_XPOS else goto DECREMENT_XPOS</code>
DECREMENT_XPOS	<code>xpos ← xpos-1  s_xpos = 1; en_xpos;</code>	<code>if (ydir) goto INCREMENT_YPOS else goto DECREMENT_YPOS</code>
INCREMENT_XPOS	<code>xpos ← xpos+1  s_xpos = 2; en_xpos;</code>	<code>if (ydir) goto INCREMENT_YPOS else goto DECREMENT_YPOS</code>
DECREMENT_YPOS	<code>ypos ← ypos-1  s_ypos = 1; en_ypos;</code>	<code>goto DRAW</code>
INCREMENT_YPOS	<code>ypos ← ypos+1  s_ypos = 2; en_ypos;</code>	<code>goto DRAW</code>
DRAW	<code>image(xpos, ypos) ← green  s_x = 0; s_y = 0; s_image_color = 1; plot = 1;</code>	<code>goto WAIT_TIMER</code>

The Verilog model for the controller is given below:

```

module controller (
    input                  clk,
    input                  reset,
    output reg             en_xpos,
    output reg [1:0]        s_xpos,
    output reg             en_ypos,
    output reg [1:0]        s_ypos,
    output reg             en_xdir,
    output reg             s_xdir,
    output reg             en_ydir,
    output reg             s_ydir,
    output reg             en_timer,
    output reg             s_timer,
    output reg             s_image_color,
    output reg [1:0]        s_x,
    output reg [1:0]        s_y,
    output reg             plot,
    input                  xdir,
    input                  ydir,
    input                  timer_done,
    input                  obstacle
);

parameter INIT          = 4'd0;
parameter WAIT_TIMER    = 4'd1;
parameter ERASE          = 4'd2;
parameter LOOK_LEFT      = 4'd3;
parameter LOOK_RIGHT     = 4'd4;
parameter TEST_X_OBSTACLE = 4'd5;
parameter CHANGE_XDIR    = 4'd6;
parameter LOOK_UP         = 4'd7;
parameter LOOK_DOWN       = 4'd8;
parameter TEST_Y_OBSTACLE = 4'd9;
parameter CHANGE_YDIR     = 4'd10;
parameter DECREMENT_XPOS = 4'd11;
parameter INCREMENT_XPOS = 4'd12;
parameter DECREMENT_YPOS = 4'd13;
parameter INCREMENT_YPOS = 4'd14;
parameter DRAW            = 4'd15;

reg [3:0] state = 0, next_state;

always @(posedge clk)
    if (reset)
        state <= INIT;
    else
        state <= next_state;

always @(*) begin
    en_xpos = 0;

```

```

s_xpos    = 0;
en_ypos   = 0;
s_ypos    = 0;
en_xdir   = 0;
s_xdir    = 0;
en_ydir   = 0;
s_ydir    = 0;
en_timer  = 0;
s_timer   = 0;
s_image_color = 0;
s_x       = 0;
s_y       = 0;
plot      = 0;
next_state = INIT;
case (state)
  INIT           : begin
    s_xdir = 0; en_xdir = 1;
    s_ydir = 0; en_ydir = 1;
    s_xpos = 0; en_xpos = 1;
    s_ypos = 0; en_ypos = 1;
    s_timer = 0; en_timer = 1;
    next_state = WAIT_TIMER;
  end
  WAIT_TIMER     : begin
    s_image_color = 1;
    plot = 1;
    s_timer = 1; en_timer = 1;
    if (timer_done) next_state = ERASE;
    else             next_state = WAIT_TIMER;
  end
  ERASE          : begin
    plot = 1;   s_image_color = 0;
    s_timer = 0; en_timer = 1;
    if (xdir)   next_state = LOOK_RIGHT;
    else         next_state = LOOK_LEFT;
  end
  LOOK_LEFT      : begin
    s_x = 1;   s_y = 0;
    next_state = TEST_X_OBSTACLE;
  end
  LOOK_RIGHT     : begin
    s_x = 2;   s_y = 0;
    next_state = TEST_X_OBSTACLE;
  end
  TEST_X_OBSTACLE : begin
    if (obstacle) next_state = CHANGE_XDIR;
    else if (ydir) next_state = LOOK_DOWN;
    else           next_state = LOOK_UP;
  end
  CHANGE_XDIR    : begin

```

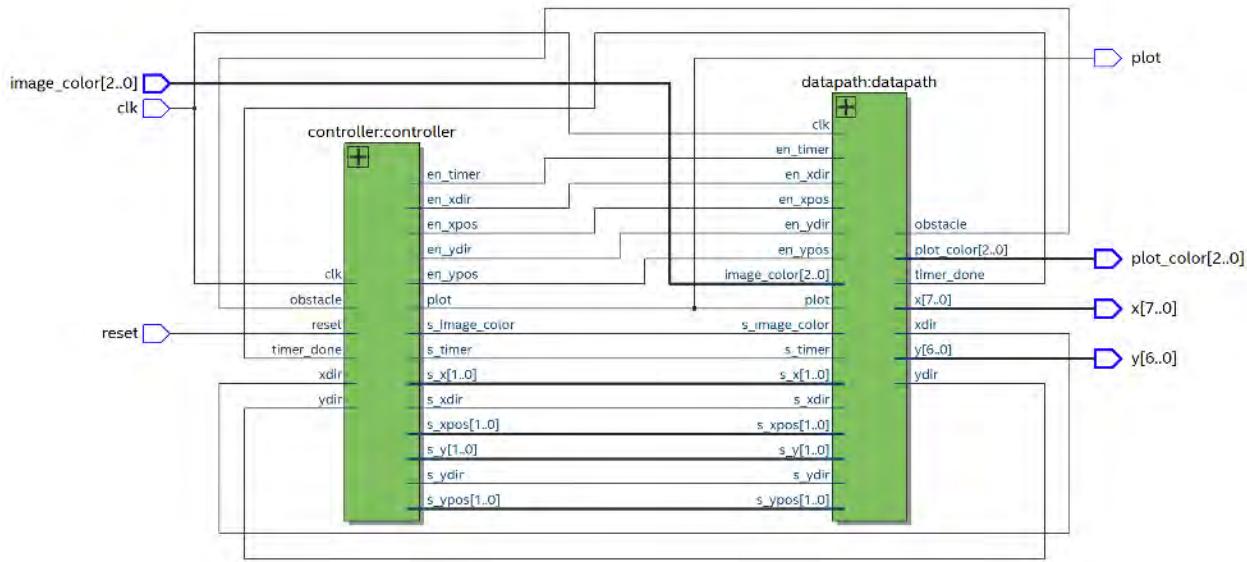
```

    s_xdir = 1; en_xdir = 1;
    if (ydir)      next_state = LOOK_DOWN;
    else          next_state = LOOK_UP;
end
LOOK_UP           : begin
    s_x = 0; s_y = 1;
    next_state = TEST_Y_OBSTACLE;
end
LOOK_DOWN         : begin
    s_x = 0; s_y = 2;
    next_state = TEST_Y_OBSTACLE;
end
TEST_Y_OBSTACLE  : begin
    if (obstacle)   next_state = CHANGE_YDIR;
    else if (xdir)  next_state = INCREMENT_XPOS;
    else            next_state = DECREMENT_XPOS;
end
CHANGE_YDIR       : begin
    s_ydir = 1; en_ydir = 1;
    if (xdir)      next_state = INCREMENT_XPOS;
    else            next_state = DECREMENT_XPOS;
end
DECREMENT_XPOS   : begin
    s_xpos = 1; en_xpos = 1;
    if (ydir)      next_state = INCREMENT_YPOS;
    else            next_state = DECREMENT_YPOS;
end
INCREMENT_XPOS   : begin
    s_xpos = 2; en_xpos = 1;
    if (ydir)      next_state = INCREMENT_YPOS;
    else            next_state = DECREMENT_YPOS;
end
DECREMENT_YPOS   : begin
    s_ypos = 1; en_ypos = 1;
    next_state = DRAW;
end
INCREMENT_YPOS   : begin
    s_ypos = 2; en_ypos = 1;
    next_state = DRAW;
end
DRAW              : begin
    s_image_color = 1; plot = 1;
    next_state = WAIT_TIMER;
end
default           ::;
endcase
end
endmodule

```

### 11.3.4 Complete Processor: Connecting Controller and Datapath

The schematic and complete Verilog code for the processor is given below:



```
module processor (
    input          clk,
    input          reset,
    output [7:0]   x,
    output [6:0]   y,
    output [2:0]   plot_color,
    output         plot,
    input [2:0]   image_color
);

wire      en_xpos;
wire [1:0] s_xpos;
wire      en_ypos;
wire [1:0] s_ypos;
wire      en_xdir;
wire      s_xdir;
wire      en_ydir;
wire      s_ydir;
wire      en_timer;
wire      s_timer;
wire      s_image_color;
wire [1:0] s_x;
wire [1:0] s_y;
wire      xdir;
wire      ydir;
wire      timer_done;
wire      obstacle;

controller controller (
    .image_color(image_color),
    .clk(clk),
    .reset(reset),
    .obstacle(obstacle),
    .plot(plot),
    .s_image_color(s_image_color),
    .s_timer(s_timer),
    .s_x(s_x),
    .s_y(s_y),
    .s_xdir(s_xdir),
    .s_xpos(s_xpos),
    .s_ydir(s_ydir),
    .s_ypos(s_ypos),
    .en_xdir(en_xdir),
    .en_xpos(en_xpos),
    .en_ydir(en_ydir),
    .en_ypos(en_ypos),
    .en_timer(en_timer)
);
```

```

    .clk          (clk          ),
    .reset        (reset         ),
    .en_xpos      (en_xpos      ),
    .s_xpos       (s_xpos       ),
    .en_ypos      (en_ypos      ),
    .s_ypos       (s_ypos       ),
    .en_xdir      (en_xdir      ),
    .s_xdir       (s_xdir       ),
    .en_ydir      (en_ydir      ),
    .s_ydir       (s_ydir       ),
    .en_timer     (en_timer     ),
    .s_timer      (s_timer      ),
    .s_image_color (s_image_color),
    .s_x          (s_x          ),
    .s_y          (s_y          ),
    .plot         (plot         ),
    .xdir         (xdir         ),
    .ydir         (ydir         ),
    .timer_done   (timer_done   ),
    .obstacle     (obstacle     )
);

datopath datapath (
    .clk          (clk          ),
    .en_xpos      (en_xpos      ),
    .s_xpos       (s_xpos       ),
    .en_ypos      (en_ypos      ),
    .s_ypos       (s_ypos       ),
    .en_xdir      (en_xdir      ),
    .s_xdir       (s_xdir       ),
    .en_ydir      (en_ydir      ),
    .s_ydir       (s_ydir       ),
    .en_timer     (en_timer     ),
    .s_timer      (s_timer      ),
    .s_image_color (s_image_color),
    .s_x          (s_x          ),
    .s_y          (s_y          ),
    .xdir         (xdir         ),
    .ydir         (ydir         ),
    .timer_done   (timer_done   ),
    .obstacle     (obstacle     ),
    .x            (x            ),
    .y            (y            ),
    .plot_color   (plot_color   ),
    .image_color  (image_color  ),
    .plot         (plot         )
);

endmodule

```

### 11.3.5 Image RAM

The image RAM has 2 read ports and 1 write port. It takes x and y image coordinates as input and translates them into RAM addresses. The first read port connection comes from the processor, to check if a pixel at a given location is an obstacle. The second read port connection comes from the VGA controller to display the image. The write port connection is from the processor, for the purpose of plotting to the image. The processor and VGA ports have separate clocks, since the processor runs at 50 MHz while the VGA controller must run at 25 MHz. The VGA controller provides the clock for the VGA port.

```
module image_ram (
    input                  wren,
    input                  clk_proc,
    input                  clk_vga,
    input [7:0]             x_proc,
    input [6:0]             y_proc,
    input [7:0]             x_vga,
    input [6:0]             y_vga,
    input [2:0]              din,
    output reg [2:0]         dout_proc,
    output reg [2:0]         dout_vga
);

parameter IMAGE_FILE = "obstacle_course.mem";

wire [14:0] addr_proc = 160*y_proc + x_proc;
wire [14:0] addr_vga = 160*y_vga + x_vga;

reg [2:0] ram [0:19199];
initial $readmemh(IMAGE_FILE, ram);

always @(posedge clk_proc) begin
    if (wren)
        ram[addr_proc] <= din;

    dout_proc <= ram[addr_proc];
end

always @(posedge clk_vga) begin
    dout_vga <= ram[addr_vga];
end

endmodule
```

### 11.3.6 Complete System: Connecting Processor to Image RAM and VGA Controller

The Verilog for the top-level module that connects the processor to the image RAM, VGA controller and DE2-115 pins is given below:

```
module obstacle_DE2 (
```

```

input      CLOCK_50,           // 50 MHz
input [0:0] KEY,
output    VGA_CLK,            // VGA Clock
output    VGA_HS,             // VGA H_SYNC
output    VGA_VS,             // VGA V_SYNC
output    VGA_BLANK,          // VGA BLANK
output    VGA_SYNC,           // VGA SYNC
output [9:0] VGA_R,           // VGA Red[9:0]
output [9:0] VGA_G,           // VGA Green[9:0]
output [9:0] VGA_B            // VGA Blue[9:0]
);

wire [2:0] plot_color;
wire [7:0] x_proc;
wire [6:0] y_proc;
wire [7:0] x_vga;
wire [6:0] y_vga;
wire     plot;
wire [2:0] image_color;
wire [2:0] vga_color;

processor processor (
    .clk      (CLOCK_50),
    .reset    (~KEY[0]),
    .x        (x_proc),
    .y        (y_proc),
    .plot_color (plot_color),
    .plot     (plot),
    .image_color (image_color)
);
image_ram image_ram (
    .wren     (plot),
    .clk_proc (CLOCK_50),
    .clk_vga  (VGA_CLK),
    .x_proc   (x_proc),
    .y_proc   (y_proc),
    .x_vga    (x_vga),
    .y_vga    (y_vga),
    .din      (plot_color),
    .dout_proc (image_color),
    .dout_vga  (vga_color)
);
defparam image_ram.IMAGE_FILE = "obstacle_course.mem";

vga_xy_controller vga_xy_controller (
    .CLOCK_50   (CLOCK_50),
    .resetn    (~KEY[0]),
    .color      (vga_color),
    .x         (x_vga),

```

```

.y          (y_vga),
.VGA_R     (VGA_R),
.VGA_G     (VGA_G),
.VGA_B     (VGA_B),
.VGA_HS    (VGA_HS),
.VGA_VS    (VGA_VS),
.VGA_BLANK (VGA_BLANK),
.VGA_SYNC   (VGA_SYNC),
.VGA_CLK    (VGA_CLK)
);

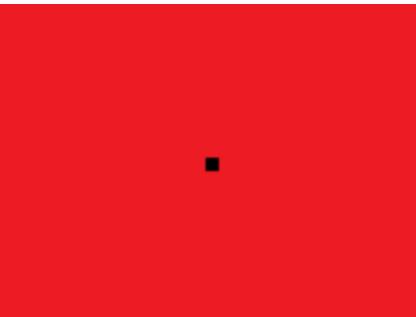
endmodule

```

### 11.3.7 Simulating and Debugging Image Memory

This example uses the obstacle course example to illustrate how you can simulate a complete processor design in ModelSim, including seeing the interactions with the obstacle memory containing a background image. This is extremely valuable for debugging, as you can see what happens with all of the signals in every state. Here are the steps I took.

**Step 1:** Create a background image that will force interesting interactions to happen quickly, for example, an obstacle course where the ball will hit an obstacle very soon, so that you don't have to simulate too many cycles to see what happens when the ball encounters an obstacle.



Here's what I used for the obstacle course. It's basically a big obstacle with a tiny hole a few pixels wide in the middle, centered at (80, 60). I start the ball in the middle of the hole so that it will collide with the obstacle in a few steps. I call the file "tiny\_obstacle"

**Step 2:** Use `bmp2mem` to convert the `tiny_obstacle.bmp` BMP file from Paint to a `.mem` file as usual.

**Step 3:** Copy the `.mem` file into the `simulation/modelsim` folder under your main Quartus project folder.

**Step 4:** Change the timer limit to a small number so you don't have to count to 1 million before getting out of the `WAIT_TIMER` state.

```
// parameter TIMER_LIMIT = 26'd1_000_000;
parameter TIMER_LIMIT = 26'd2;
```

**Step 5:** Create a testbench for the processor as you normally would that runs for some number of cycles.

```
`timescale 1ns/1ns
module processor_tb ();
  reg      clk;
```

```

reg      reset;
wire [7:0] x;
wire [6:0] y;
wire [2:0] plot_color;
wire      plot;
wire [2:0] image_color;
wire [2:0] vga_color;

processor processor (
    .clk      (clk),
    .reset    (reset),
    .x        (x),
    .y        (y),
    .plot_color (plot_color),
    .plot     (plot),
    .image_color (image_color)
);

image_ram image_ram (
    .wren      (plot),
    .clk_proc   (clk),
    .clk_vga    (1'b0),
    .x_proc     (x),
    .y_proc     (y),
    .x_vga      (8'b0),
    .y_vga      (7'b0),
    .din        (plot_color),
    .dout_proc   (image_color),
    .dout_vga    (vga_color)
);
defparam image_ram.IMAGE_FILE = "tiny_obstacle.mem";

always #5 clk = ~clk;

initial begin
    clk = 0;  reset = 1;
    #10 reset = 0;
    #10000 $stop;
end

endmodule

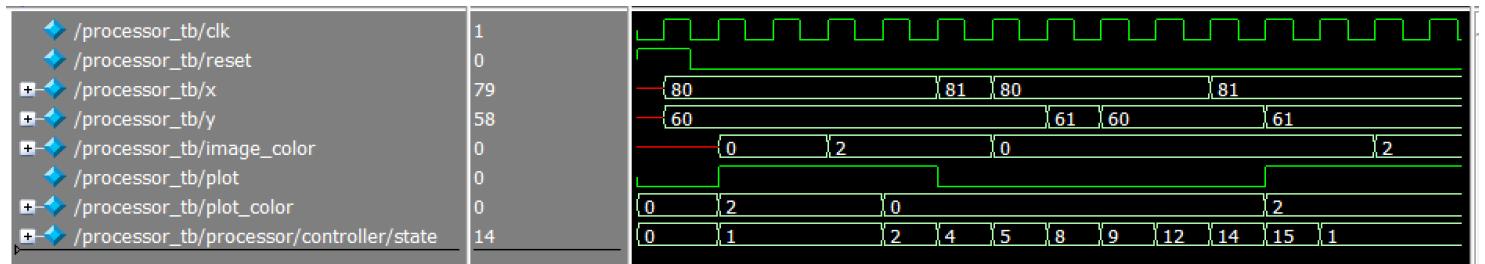
```

**Step 5:** Compile your design through Analysis & Synthesis in Quartus

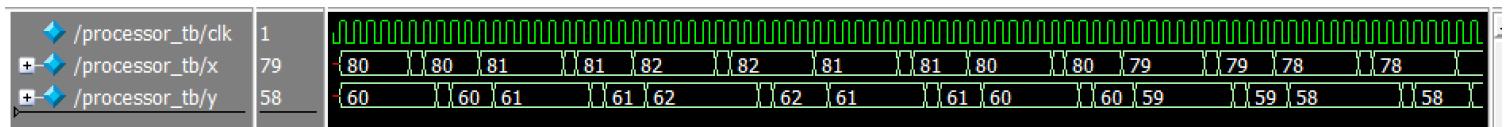
**Step 6:** Launch ModelSim from Quartus as you normally would.

### Step 7: Use ModelSim to find your bugs!

Here's an example of waveforms showing not only the top-level processor signals, but also the state in the controller and the x and y addresses and output (image\_color) from image memory. Note how the controller stays in state 1, WAIT\_TIMER for 3 cycles, but is only in the other states for 1 cycle.



Zooming out, and looking at coordinates x, y of the ball, you can clearly see the ball bouncing around in the small hole centered upon (80, 60).



## 12 Using the Terasic/Altera DE2-115

This chapter is evolving documentation on Terasic/Altera DE2-115 board relevant to this course that is not (clearly) documented elsewhere. It is being continually updated as needed. The goal isn't to replace the official documentation provided by Terasic, just to document additional material, tips, etc. that will be helpful for labs and projects.

Here are links to the official Terasic resources:

- DE2-115 Board Homepage: <https://www.terasic.com.tw/cgi-bin/page/archive.pl?No=30>
- DE2-115 Documentation and Sample Projects: <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=30&PartNo=4>

### 12.1 Top-Level Module Ports and Pin Definitions

The ports of the top-level module of a Verilog design for the DE2-115 board must correspond to the names of FPGA pins on the board. All names to FPGA pins for board I/O in this document refer to the official Terasic/Altera pin table documentation found on their website:

- [https://www.terasic.com.tw/cgi-bin/page/archive\\_download.pl?Language=English&No=30&FID=3bc5410e25f903b5d6210c07ff584eb2](https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=30&FID=3bc5410e25f903b5d6210c07ff584eb2)

Here's an example module header that uses most of the pins of interest for this course (there are lots more for other devices on the board):

```
module top (
    input          CLOCK_50,    // 50 MHz clock
    input [17:0]   SW,         // switches
    input [3:0]    KEY,        // pushbuttons
    // LEDs
    output [17:0]  LEDR,       // red
    output [7:0]   LEDG,       // green
    // 7-segment display digits
    output [6:0]   HEX0,
    output [6:0]   HEX1,
    output [6:0]   HEX2,
    output [6:0]   HEX3,
    output [6:0]   HEX4,
    output [6:0]   HEX5,
    output [6:0]   HEX6,
    output [6:0]   HEX7,
    // VGA monitor interface
    output        VGA_CLK,
    output        VGA_HS,
    output        VGA_VS,
    output        VGA_BLANK,
    output        VGA_SYNC,
    output [9:0]   VGA_R,
    output [9:0]   VGA_G,
    output [9:0]   VGA_B,
```

```

// LCD display interface
output      LCD_ON,      // LCD Power ON/OFF
output      LCD_BLON,     // LCD Back Light ON/OFF
output      LCD_RW,       // LCD read/write control
output      LCD_EN,       // LCD Enable
output      LCD_RS,       // LCD Command/Data Select
inout [7:0]  LCD_DATA,    // LCD Data bus 8 bits
// PS2 keyboard/mouse interface
inout      PS2_CLK,
inout      PS2_DAT
);

```

## 12.2 Switches, Buttons, LEDs, and 7-Segment Displays

### Example Quartus Project: board\_demo

The DE2-115 board switches, buttons, LEDs, and 7-segment displays are located on the board and named as shown below:



Notes:

- The LEDs have a faint glow (neither fully on nor fully off) if they are not connected in the design. If this bothers you, you should include all of the LEDs in the design and tie the ones that you are not using to 0.
- The 7 segments of each of the hex digits are active low, meaning that connecting them to 0 turns them on and 1 turns them off.
- The pushbuttons KEY[3:0] are active low, meaning that they output a 0 if the button is pressed and a 1 if not pressed.

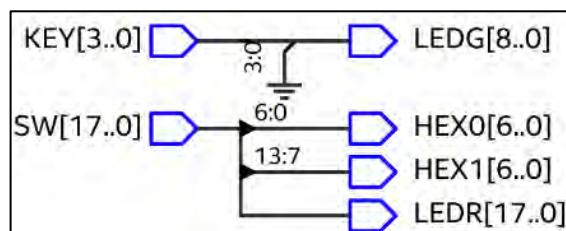
Here's a simple top-level module that connects switches and buttons to LED's and 7-segment displays.

```

module board_demo (
  input  [17:0]  SW,
  input  [3:0]   KEY,
  output [17:0]  LEDR,
  output [8:0]   LEDG,
  output [6:0]   HEX0,
  output [6:0]   HEX1
);

  assign LEDR[17:0] = SW[17:0];
  assign LEDG[3:0] = KEY[3:0];
  assign HEX0[6:0] = SW[6:0];

```



```

assign HEX1[6:0] = SW[13:7];
endmodule

```

## 12.3 Switch Debouncing

Example Quartus Project: debounce\_demo

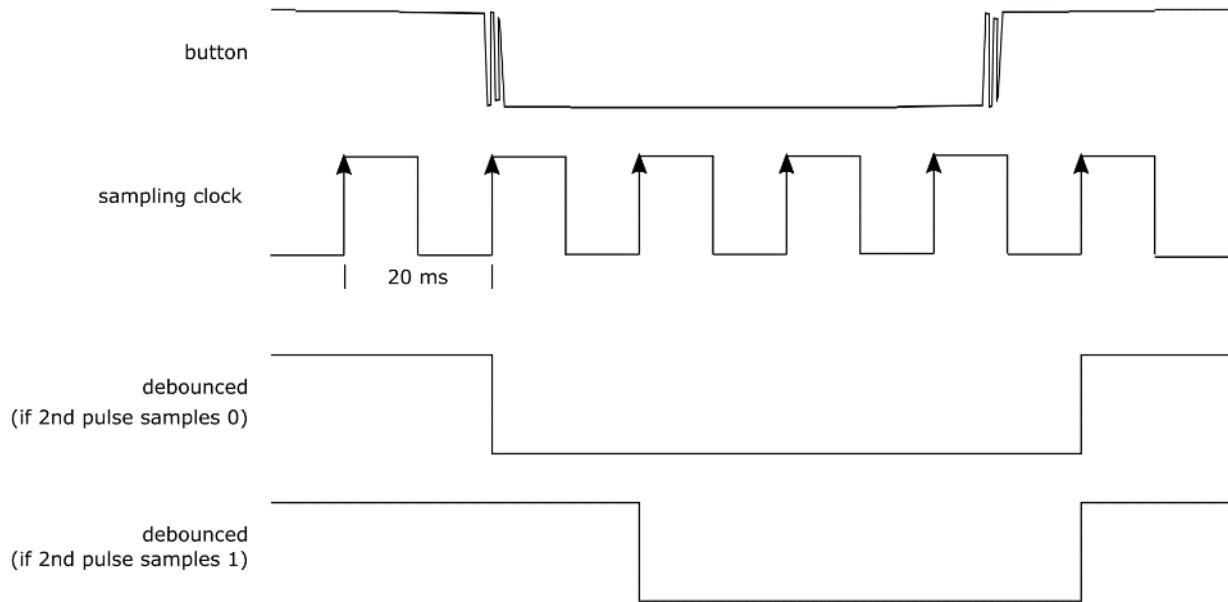
### 12.3.1 Theory

Mechanical buttons and switches are subject to an effect called *bouncing* where when you open or close them, because of their springiness, they bounce between states a few times before staying open or closed. This can be a problem in digital systems, especially if you want to use a button as a clock, because each time you press the button rather than getting one clock edge, you make actually get an indeterminate number of edges. Because of this, buttons and switches must be *debounced* before you can use one reliably as a clock.



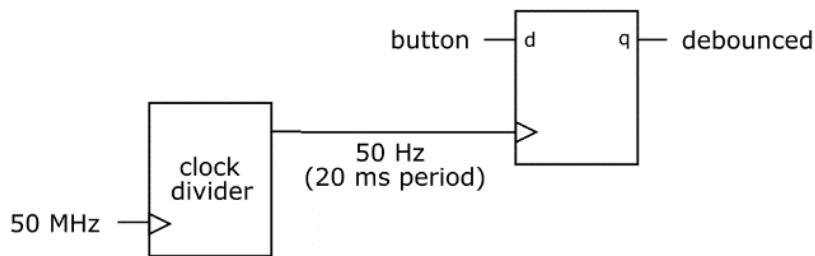
There are a number of ways of debouncing switches in hardware and an internet search will turn up a number of different design for debouncing modules. The basic idea behind all of them is to sample the value of the switch and then wait long enough for the bouncing to stop before checking the value again. A typical value for the wait time is 20 ms. A conceptually simple approach is to sample the button using a clock with a period of approximately 20 ms (50 Hz frequency) as illustrated below.

Samples taken away from the time that the button is bouncing produce obvious values for the debounced result. If a sample is taken during the time that the switch is bouncing, it doesn't matter whether the sample is a 1 or a zero. Either way, the debounced output will detect a single button press with the result synchronized to the sampling clock, although it may be shifted by one cycle of sampling clock.



### 12.3.2 Debouncing Circuit Design

A schematic for the debouncing circuit is shown below:



Verilog code for the debouncing circuit is given below:

```

module debounce (
    input      CLOCK_50,
    input      in,
    output reg  out
);

reg [19:0] count;

always @(posedge CLOCK_50)
    count <= count + 1'b1;

always @(posedge count[19])
    out <= in;

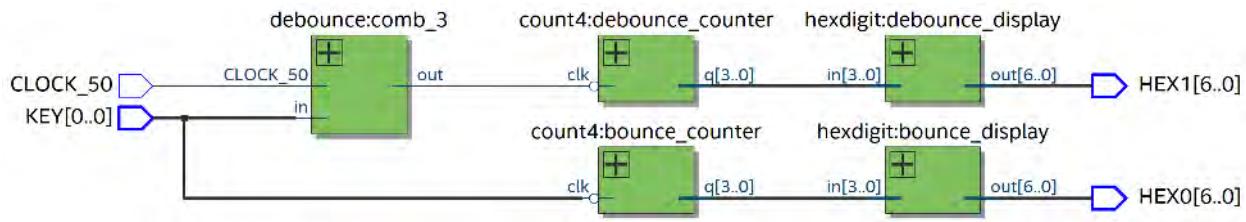
```

```
endmodule
```

The first `always` block is the clock divider and the second is the sampling flip-flop. The input clock of 50 MHz must be divided by 1 million to produce an output clock of 50 Hz. If we approximate 1 million as  $2^{20}$ , we can use the most significant bit of a 20-bit counter as the divided clock.

### 12.3.3 Demo Using Terasic DE2-115 Board Pushbutton

To demonstrate the effects of switch bounce and the use of a debouncing circuit, we use a pushbutton on the DE2-115 board both directly and through a debouncer to clock two counters and display the output of each counter on seven-segment displays as shown below.



The Verilog code for the demo is as follows:

```
module debounce_demo (
    input      CLOCK_50,
    input      [0:0] KEY,
    output     [6:0] HEX0,
    output     [6:0] HEX1
);

    wire      key_debounced;
    wire [3:0] bounce_count;
    wire [3:0] debounce_count;

    debounce (
        .CLOCK_50  (CLOCK_50),
        .in        (KEY[0]),
        .out       (key_debounced)
    );

    count4 bounce_counter (
        .clk      (~KEY[0]),
        .q        (bounce_count)
    );

    count4 debounce_counter (
        .clk      (~key_debounced),
        .q        (debounce_count)
    );

    hexdigit bounce_display (
```

```

        .in      (bounce_count),
        .out     (HEX0)
    );

hexdigit debounce_display (
    .in      (bounce_count),
    .out     (HEX1)
);

endmodule

module count4 (
    input clk,
    output reg [3:0] q
);

    always @(posedge clk)
        q <= q + 1;

endmodule

```

Because of switch bounce, the counter that is connected directly to the pushbutton will occasionally increment more than once with each press of KEY[0], while the counter connected to the button through the debouncer should reliably increment by 1 each time.

## 12.4 LCD Display

### Example Quartus Project: LCD\_Demo

The LCD display on the DE2-115 board can display 32 characters in 2 rows of 16 characters each. The DE2-115 has a Crystalfontz CFAH1602B-TMC-JP LCD display, which uses the HD44780U Dot Matrix LCD Controller/Driver chip. Writing a Verilog module that communicates with the LCD controller/driver chip requires a fairly advanced understanding of digital system design that would go beyond the scope of this course. Terasic provides sample Verilog code that uses this module, but IMHO this is still a bit more complex than it needs to be. Instead, I have written a simpler interface module called `LCD_Controller`, that uses the Verilog `LCD_Display` module from John Loomis of the University of Dayton<sup>11</sup>, who in turn based his implementation on the design by James Hamblen of GA Tech that was originally written in VHDL.<sup>12</sup>

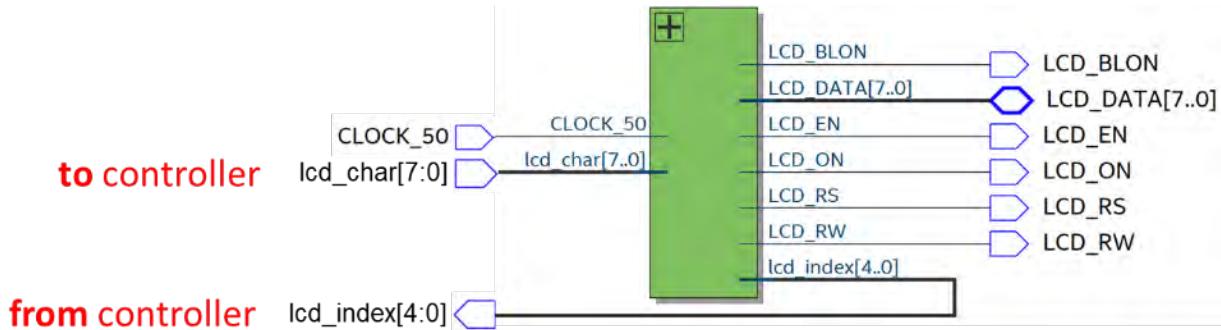
The `LCD_Controller` module considers the LCD display as an array of 31 characters, where index 0:15 in the top row and index 16:31 is the bottom row.



<sup>11</sup> <http://www.johnloomis.org/digitallab/lcdlab/lcdlab3/lcdlab3.html>

<sup>12</sup> <http://hamblen.ece.gatech.edu/DE2/>

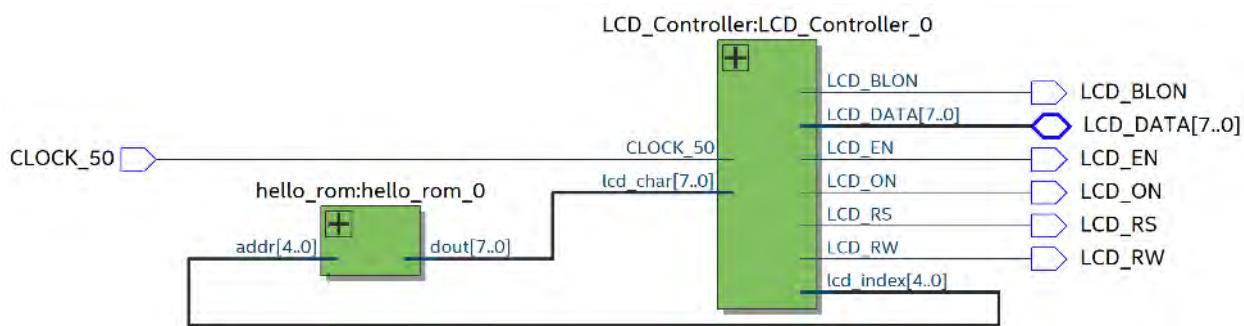
The LCD\_Controller interface is pictured below. Internally, it continuously cycles through the 32 indices `lcd_index[4:0]` and outputs these to synchronous with the board 50 MHz clock, `CLOCK_50`. It expects the user to provide an 8-bit value, `lcd_char[7:0]` that represents the character to display at that index. The module automatically controls the remaining ports that connect to the LCD display on the Terasic board.



Port	Direction	Description
<code>CLOCK_50</code>	input	50 MHz clock
<code>lcd_char[7:0]</code>	input	ASCII character or hex digit in the range 0-F
<code>lcd_index[4:0]</code>	output	Index where to display character
<code>LCD_BLON</code> , <code>LCD_DATA</code> , <code>LCD_EN</code> , <code>LCD_ON</code> , <code>LCD_RS</code> , <code>LCD_RW</code>	input/output	Connection to LCD display on Terasic board

#### 12.4.1 Displaying a Static Text Message

The simplest application of `LCD_Controller` is to connect it to a lookup table or read-only memory (ROM) that contains a message, where `lcd_index` connects to the ROM's address port and `lcd_char` connects to the ROM's data out port, and the message in the ROM will “magically” appear on the display.



Below is Verilog model for a sample ROM, `hello_rom`, that will display the message

```
Hello  
World-3A
```

Note that the character displayed by `LCD_Controller` can either be represented as an 8-bit ASCII character or as a numerical value in the range `8'b00000000-8'b00001111`, in which case it will display the corresponding hex digit.

```

module hello_rom (
    input      [4:0] addr,
    output reg   [7:0] dout
);

    always @(*)
        case (addr)
            // Line 1 starts at 0
            0:  dout = "H";
            1:  dout = "e";
            2:  dout = "l";
            3:  dout = "l";
            4:  dout = "o";
            // Line 2 starts at 16
            16: dout = "W";
            17: dout = "o";
            18: dout = "r";
            19: dout = "l";
            20: dout = "d";
            21: dout = "-";
            22: dout = "3";    // ASCII value for number
            23: dout = 8'd10;  // will display hex digit A
            // Blank spaces for all unspecified locations
            default: dout = " ";
        endcase
endmodule

```

The Verilog module `hello_rom_de2` is a complete top-level design that connects `hello_rom` to `LCD_Controller`. In order to use it, your design must also include the modules `LCD_Display` and `reset_delay`, which are used inside of `LCD_Controller`.

```

module hello_rom_de2 (
    input      CLOCK_50,
    output     LCD_ON,
    output     LCD_BLON,
    output     LCD_RW,
    output     LCD_EN,
    output     LCD_RS,
    inout    [7:0] LCD_DATA
);

    wire [4:0] lcd_index;
    wire [7:0] lcd_char;

    hello_rom hello_rom_0 (

```

```

    .addr      (lcd_index),
    .dout     (lcd_char)
);

LCD_Controller LCD_Controller_0 (
    .lcd_char   (lcd_char),
    .lcd_index  (lcd_index),
    .CLOCK_50   (CLOCK_50 ),
    .LCD_ON     (LCD_ON   ),
    .LCD_BLON   (LCD_BLON ),
    .LCD_RW     (LCD_RW   ),
    .LCD_EN     (LCD_EN   ),
    .LCD_RS     (LCD_RS   ),
    .LCD_DATA   (LCD_DATA )
);

endmodule

```

### 12.4.2 Mixing Formatted Text and Data

Probably the most common use of the LCD display is to display a mix of pre-formatted text and input data. This is easily done by designing lookup-table logic that outputs static characters for some index values and data values for others.

The module `text_and_data` takes 8-bit input value `n[7:0]` and displays it as both a hexadecimal value and as a display character. For example, if `n` equals `8'h61`, it would display

```
HEX: 61
char: a
```

The Verilog model for the module is as follows:

```

module text_and_data (
    input      [7:0] n,
    input      [4:0] lcd_index,
    output reg  [7:0] lcd_char
);

always @(*)
    case (lcd_index)
        0: lcd_char = "H";
        1: lcd_char = "E";
        2: lcd_char = "X";
        3: lcd_char = ":";
        4: lcd_char = " ";
        5: lcd_char = {4'b0, n[7:4]};
        6: lcd_char = {4'b0, n[3:0]};

        16: lcd_char = "c";
        17: lcd_char = "h";
        18: lcd_char = "a";
    endcase
endmodule

```

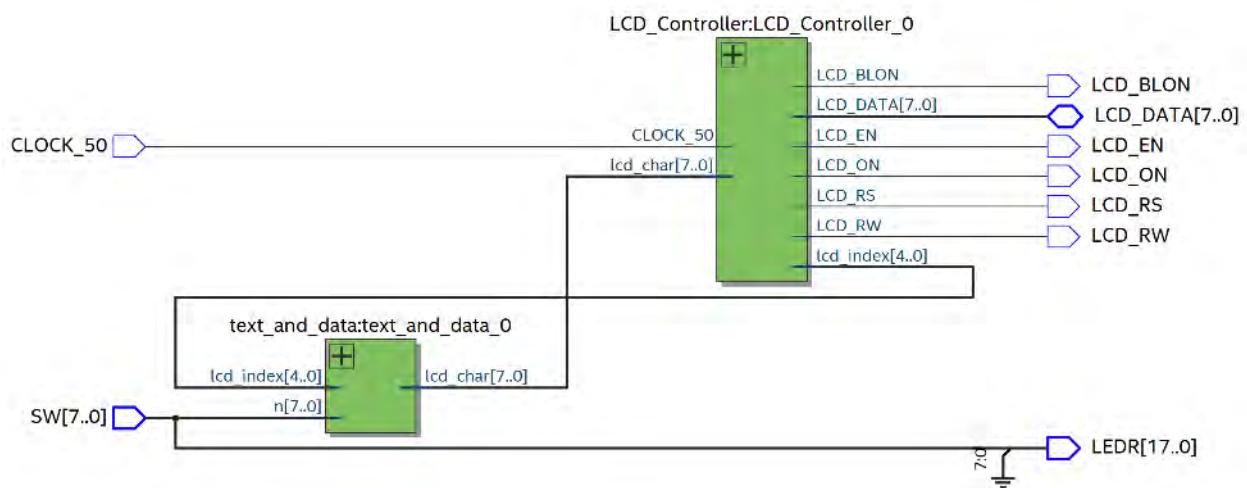
```

19: lcd_char = "r";
20: lcd_char = ":";
21: lcd_char = " ";
22: lcd_char = n;
default: lcd_char = " ";
endcase
endmodule

```

Note that because the output of the module is 8 bits wide, in order to display the hex digits, it pads a 4-bit value out to 8 bits with leading zeros.

The top-level DE2-115 module `text_and_data_de2` connects the data value `n[7:0]` to switches `SW[7:0]`.



```

module text_and_data_de2 (
    input [7:0] SW,
    output [17:0] LEDR,
    input CLOCK_50,
    output LCD_ON,
    output LCD_BLON,
    output LCD_RW,
    output LCD_EN,
    output LCD_RS,
    inout [7:0] LCD_DATA
);

    wire [4:0] lcd_index;
    wire [7:0] lcd_char;

    assign LEDR[17:8] = 0;
    assign LEDR[7:0] = SW[7:0];

    text_and_data text_and_data_0 (
        .n(SW[7:0]),

```

```

    .lcd_index  (lcd_index),
    .lcd_char   (lcd_char)
);

LCD_Controller LCD_Controller_0 (
    .lcd_char   (lcd_char),
    .lcd_index  (lcd_index),
    .CLOCK_50   (CLOCK_50 ),
    .LCD_ON     (LCD_ON   ),
    .LCD_BLON   (LCD_BLON ),
    .LCD_RW     (LCD_RW   ),
    .LCD_EN     (LCD_EN   ),
    .LCD_RS     (LCD_RS   ),
    .LCD_DATA   (LCD_DATA )
);

endmodule

```

### 12.4.3 Changing Location of Displayed Characters

The previous examples displayed characters at fixed locations specified by a table lookup using a Verilog `case` statement switching off of the LCD display index. This example module `character_and_location` shows how to change the display location of `character` by comparing `lcd_index` with a specified `location` value. This approach can be extended for use in scrolling applications, etc.

```

module character_and_location (
    input      [4:0] lcd_index,
    input      [7:0] character,
    input      [4:0] location,
    output reg  [7:0] dout
);

    always @(*)
        if (location == lcd_index)
            dout = character;
        else
            dout = " ";

```

endmodule

The top-level DE2-115 module `character_and_location_de2` demonstrates the use of this approach, with the `character` and `location` inputs connected to switches and displayed on HEX displays. The following table specifies the DE2-115 I/O connections:

<b>character_and_location port</b>	<b>DE2-115 Input</b>	<b>Displayed on</b>
location[4:0]	SW[12:8]	HEX7, HEX6
character[7:0]	SW[7:0]	HEX5, HEX4

```

module character_and_location_de2 (
    input          CLOCK_50,
    input  [12:0]   SW,
    output [6:0]    HEX4,HEX5,HEX6,HEX7,
    output [8:0]    LEDG,
    output [17:0]   LEDR,

    //LCD Module 16X2
    output         LCD_ON,
    output         LCD_BLON,
    output         LCD_RW,
    output         LCD_EN,
    output         LCD_RS,
    inout [7:0]    LCD_DATA
);

// Send used switches to red leds
assign LEDR[12:0] = SW[12:0];

wire [4:0] lcd_index;
wire [4:0] location = SW[12:8];
wire [7:0] lcd_char;
wire [7:0] character = SW[7:0];

character_and_location c_and_l (
    .lcd_index (lcd_index),
    .character (character),
    .location  (location),
    .dout      (lcd_char)
);

LCD_Controller LCD_Controller_0 (
    .lcd_char (lcd_char),
    .lcd_index (lcd_index),
    .CLOCK_50  (CLOCK_50 ),
    .LCD_ON    (LCD_ON   ),
    .LCD_BLON  (LCD_BLON ),
    .LCD_RW    (LCD_RW   ),
    .LCD_EN    (LCD_EN   ),
    .LCD_RS    (LCD_RS   ),
    .LCD_DATA  (LCD_DATA )
);

hexdigit addr1(
    .in        ({3'b0, location[4]}),
    .out       (HEX7)
);

hexdigit addr0(
    .in        (location[3:0]),

```

```

        .out      (HEX6)
    );

hexdigit data1(
    .in       (character[7:4]),
    .out      (HEX5)
);

hexdigit data0(
    .in       (character[3:0]),
    .out      (HEX4)
);

// blank unused LEDs and 7-segment digits
assign LEDR[17:13] = 5'h0;
assign LEDG[8:0]   = 9'h0;

endmodule

```

#### 12.4.4 Storing Message in a RAM

Another application is to store a message in a RAM that can be overwritten as needed. Below is a `hello_ram` module, which has 5-bit read and write address busses and 8-bit data. It is initialized to display “Hello World”.

```

module hello_ram (
    input      clk,
    input [4:0] raddr,
    input [7:0] din,
    input [4:0] waddr,
    input      we,
    output [7:0] dout
);

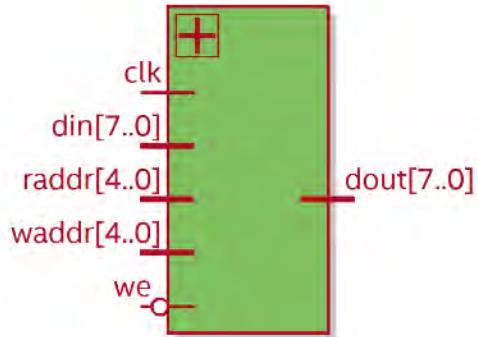
reg [7:0] m [0:31];

always@(posedge clk)
    if (we)
        m[waddr] <= din;

assign dout = m[raddr];

initial begin
    // Line 1
    m[8'h00] = "H";
    m[8'h01] = "e";
    m[8'h02] = "l";
    m[8'h03] = "l";
    m[8'h04] = "o";
    m[8'h05] = " ";
    m[8'h06] = " ";

```



```

m[8'h07] = " ";
m[8'h08] = " ";
m[8'h09] = " ";
m[8'h0a] = " ";
m[8'h0b] = " ";
m[8'h0c] = " ";
m[8'h0d] = " ";
m[8'h0e] = " ";
m[8'h0f] = " ";
// Line 2
m[8'h10] = "W";
m[8'h11] = "o";
m[8'h12] = "r";
m[8'h13] = "l";
m[8'h14] = "d";
m[8'h15] = " ";
m[8'h16] = " ";
m[8'h17] = " ";
m[8'h18] = " ";
m[8'h19] = " ";
m[8'h1a] = " ";
m[8'h1b] = " ";
m[8'h1c] = " ";
m[8'h1d] = " ";
m[8'h1e] = " ";
m[8'h1f] = " ";
end

```

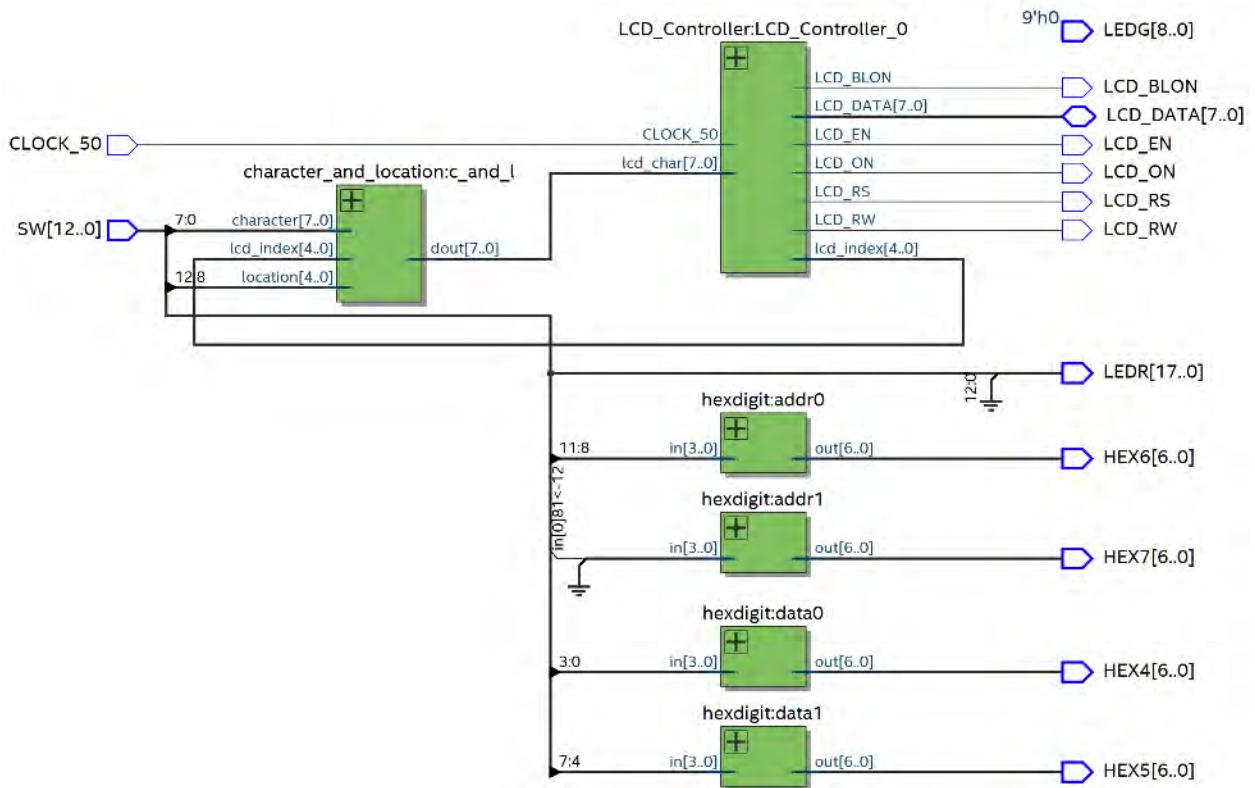
```
endmodule
```

The top-level DE2-115 module `hello_ram_de2` demonstrates the use of a character RAM, where the user can write values to `hello_ram` using switches and buttons. The `hello_ram` address is displayed on HEX7 and HEX6 and the input data is displayed on HEX5 and HEX4. The assignments of switches and buttons is as follows:

Signal Description	LCD_ram Pin	DE2-115 Input	Displayed on
character data in	din[7:0]	SW[7:0]	HEX5, HEX4
write address	waddr[4:0]	SW[12:8]	HEX7, HEX6
write enable	we	~KEY[0]	
50 MHz clock	clk	CLOCK_50	

To write the letter “J” (ASCII value `8'h4A`) to the first character on the top row of the LCD display to change the message to “Jello World,”

1. set `SW[7:0]` to `8'h4A`
2. set `SW[12:8]` to `5'h0`
3. press `KEY[0]`



The Verilog for the top-level module is as follows:

```

module character_and_location_de2 (
    input      CLOCK_50,
    input [12:0] SW,
    output [6:0] HEX4, HEX5, HEX6, HEX7,
    output [8:0] LEDG,
    output [17:0] LEDR,
    //LCD Module 16X2
    output      LCD_ON,
    output      LCD_BLON,
    output      LCD_RW,
    output      LCD_EN,
    output      LCD_RS,
    inout [7:0]  LCD_DATA
);

// Send used switches to red leds
assign LEDR[12:0] = SW[12:0];

wire [4:0] lcd_index;
wire [4:0] location = SW[12:8];
wire [7:0] lcd_char;
wire [7:0] character = SW[7:0];

```

```

character_and_location c_and_1 (
    .lcd_index  (lcd_index),
    .character   (character),
    .location    (location),
    .dout        (lcd_char)
);
LCD_Controller LCD_Controller_0 (
    .lcd_char    (lcd_char),
    .lcd_index   (lcd_index),
    .CLOCK_50    (CLOCK_50 ),
    .LCD_ON      (LCD_ON   ),
    .LCD_BLON   (LCD_BLON ),
    .LCD_RW      (LCD_RW   ),
    .LCD_EN      (LCD_EN   ),
    .LCD_RS      (LCD_RS   ),
    .LCD_DATA    (LCD_DATA )
);
hexdigit addr1(
    .in          ({3'b0, location[4]}),
    .out         (HEX7)
);
hexdigit addr0(
    .in          (location[3:0]),
    .out         (HEX6)
);
hexdigit data1(
    .in          (character[7:4]),
    .out         (HEX5)
);
hexdigit data0(
    .in          (character[3:0]),
    .out         (HEX4)
);
// blank unused LEDs and 7-segment digits
assign LEDR[17:13] = 5'h0;
assign LEDG[8:0]   = 9'h0;
endmodule

```

## 12.5 VGA Monitor

### 12.5.1 The VGA XY Controller

Example Quartus Project: `vga_xy_demo`

The VGA standard uses a precisely-timed set of signals to scan an image across the rows and columns of a display. We have developed a Verilog module called `vga_xy_controller` that provides a simple interface for plotting pixels at a specified XY location without having to understand the details of the VGA protocol. The implementation of `vga_xy_controller` is based on the VGA adapter developed by Jonathan Rose at the University of Toronto ([http://www.eecg.utoronto.ca/~jayar/ece241\\_08F/vga/](http://www.eecg.utoronto.ca/~jayar/ece241_08F/vga/)).

Similar to the LCD display controller described earlier, the `vga_xy_controller` continuously cycles through a range of X and Y coordinates, and it is the responsibility of the user application to provide the color of the pixel to display at that address. Typically, this is done by using the X and Y coordinates generated by `vga_xy_controller` to address a memory that contains a bitmapped image that you want to display.

The `vga_xy_controller` makes several simplifications to reduce the amount of memory required and to improve the display update speed. First, it uses a monitor resolution of 160 columns (X) by 120 rows (Y), so that each pixel of your application is actually a block of native pixels on the display. Second, it represents colors using a 3-bit RGB value, so that you only have 8 color options, shown below:

3-bit RGB value	Color
0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	yellow
7	white

The `vga_xy_controller` has ports for interfacing with the user application, as well as ports that drive the DE2-115 board hardware. The X, Y origin (0,0) is the top-left corner of the display. The user-side ports are as follows:

Port	Description
input <code>CLOCK_50</code>	50 MHz DE2-115 board clock
input <code>reset_n</code>	controller reset, active low, typically set to <code>1'b1</code>
input [2:0] <code>color</code>	3-bit RGB value
output <code>VGA_CLK</code>	generated by controller for synchronizing plot operations
output [7:0] <code>x</code>	x-coordinate in range 0-159 left-to-right, generated by controller
output [6:0] <code>y</code>	y-coordinate in range 0-119 top-to-bottom, generated by controller

The remaining board-side ports are connected to the DE2-115 hardware ports and you generally don't need to worry about the details of what they do:

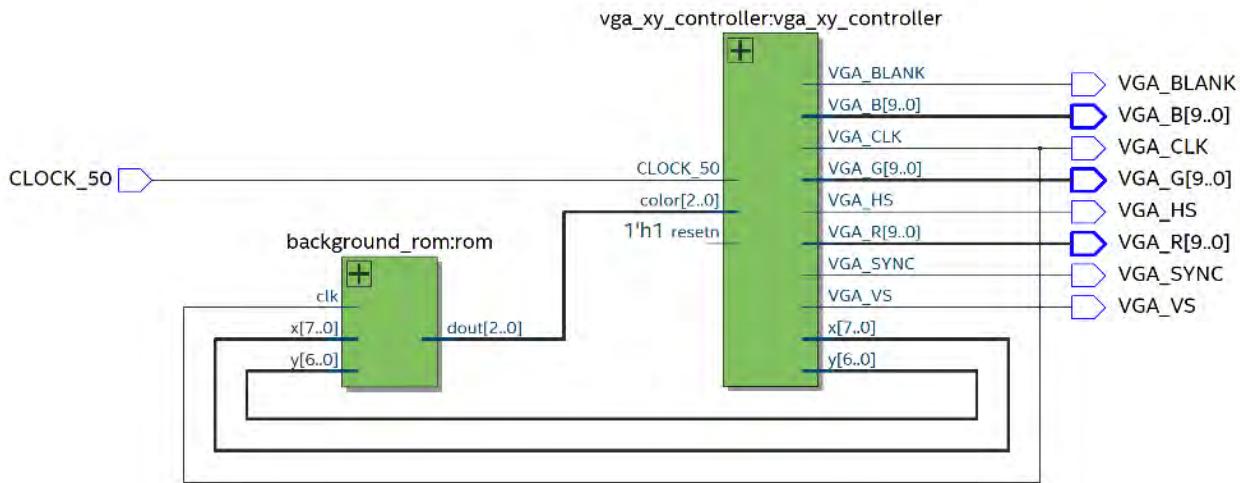
Port	Description
output <code>VGA_BLANK</code>	active-low screen blank, always set to 1

<code>output [9:0] VGA_R</code>	connection to 10-bit DAC for red
<code>output [9:0] VGA_G</code>	connection to 10-bit DAC for green
<code>output [9:0] VGA_B</code>	connection to 10-bit DAC for blue
<code>output VGA_SYNC</code>	overall monitor synchronization
<code>output VGA_HS</code>	horizontal sync
<code>output VGA_VS</code>	vertical sync

The internals of `vga_xy_controller` have been carefully tuned for proper operation. You should *never* modify any part of the module unless you really know what you are doing!

### 12.5.2 Display Image from ROM

The RTL netlist below illustrates the most basic functionality of the `vga_xy_controller`, namely, displaying a static image from an image ROM.



As described in an earlier chapter, rather than having a single address port, an image ROM has separate x and y addresses, that specify the column and row of a pixel. In this example, with assume an image that is 160 pixels wide by 120 pixels tall, with a 3-bit RGB value for each pixel. The Verilog model for the ROM below calculates the address into a memory array from the x and y values. It uses a parameter `IMAGE_FILE` to specify the initial ROM contents.

```
module background_rom (
    input          clk,
    input [7:0]    x,
    input [6:0]    y,
    output reg [2:0] dout
);
parameter IMAGE_FILE = "blank.mem";

wire [14:0] addr = 160*y + x;

reg [2:0] mem [0:19199];
initial $readmemh(IMAGE_FILE, mem);

always @(posedge clk)
    dout <= mem[addr];
```

```
endmodule
```

The VGA controller, `vga_xy_controller`, generates the sequence of x and y addresses for the ROM, much like the LCD controller generated the index for the LCD, and the user application was responsible for providing the character to display at that index. The `vga_xy_controller` also generates a 25 MHz clock, `VGA_CLK`, which is required for proper VGA display timing.

The Verilog model for the system is shown below. Note that it uses the `defparam` command to override the default memory initialization file in the background ROM `rom`.

```
module vga_rom_de2 (
    input      CLOCK_50,          // 50 MHz
    output     VGA_CLK,           // VGA Clock
    output     VGA_HS,            // VGA H_SYNC
    output     VGA_VS,            // VGA V_SYNC
    output     VGA_BLANK,         // VGA BLANK
    output     VGA_SYNC,          // VGA SYNC
    output [9:0] VGA_R,           // VGA Red[9:0]
    output [9:0] VGA_G,           // VGA Green[9:0]
    output [9:0] VGA_B            // VGA Blue[9:0]
);

wire [7:0] x_vga;
wire [6:0] y_vga;
wire [2:0] color_vga;

background_rom rom (
    .clk      (VGA_CLK),
    .x        (x_vga),
    .y        (y_vga),
    .dout     (color_vga)
);
defparam rom.IMAGE_FILE = "testpattern.mem";

vga_xy_controller vga_xy_controller (
    .CLOCK_50      (CLOCK_50),
    .resetn        (1'b1),
    .color         (color_vga),
    .x             (x_vga),
    .y             (y_vga),
    .VGA_R          (VGA_R),
    .VGA_G          (VGA_G),
    .VGA_B          (VGA_B),
    .VGA_HS         (VGA_HS),
    .VGA_VS         (VGA_VS),
    .VGA_BLANK      (VGA_BLANK),
    .VGA_SYNC       (VGA_SYNC),
    .VGA_CLK        (VGA_CLK)
);
```

```
endmodule
```

### 12.5.3 bmp2mem: Converting BMP Files to readmemh Files

Each of the memories in this Section use \$readmemh to read a text file with initial memory values. More detail on \$readmemh and the required file format can be found in the Chapter on memory. The utility **bmp2mem** provides a convenient way of converting bitmap (.bmp) files to this format. A Windows executable version of the program may be found in the course folder on Google Drive. It is most convenient to put a copy of **bmp2mem** in the same folder where you keep your bitmap images, unless you are comfortable modifying your PATH environment variable.

At present **bmp2mem** is a DOS command line program. It takes a 24-bit bitmap file (8 bits per color) as input and produces a text file with 3-bit RGB values in hexadecimal format as output. In the conversion, it rounds 8-bit per color to 1-bit per color, where 8-bit values below 128 are rounded to 0 and above 128 are rounded to 1. It works on images of any size. For background images, be sure that your canvas size in Paint, Paint3D or other drawing program is 160x120 pixels. To run it, type the command

```
bmp2mem file.bmp
```

to produce a file **file.mem**.

To open a DOS command window in Windows, type **cmd** in the search bar to locate the “Command Prompt” application and run it. Once the Command Prompt window opens, you need to navigate to the directory where your bitmap files and **bmp2mem** executable are.

Here are some basic DOS commands for navigating the file system:

Command	Description	Similar Linux Command
<b>cd</b>	change directory	<b>cd</b>
<b>cd G:\ (C:\)</b>	change to G drive (C drive)	<b>cd /</b>
<b>dir</b>	list directory	<b>ls</b>
<b>ren</b>	rename file	<b>mv</b>
<b>del</b>	delete file	<b>rm</b>
<b>copy</b>	copy file	<b>cp</b>

Note that DOS uses a backward slash (\) as a path separator, while Linux uses a forward slash (/).

#### 12.5.4 Animation Strategies

There are two distinct strategies for producing animations using the `vga_xy_controller`. The first approach, called *raster animation*, involves replacing the ROM from the previous example with a RAM and then writing to the RAM at specific XY location to change the display. The second approach, called *sprite animation*, involves having multiple ROMs that each contain a different image and switching between them, depending upon the XY coordinate generated by `vga_xy_controller`. Both approaches have their advantages and disadvantages, depending upon the application. It is also possible to combine the two.

**Raster animation** using a RAM is best for applications that operate on individual pixels and where you want to save changes made to the image. A simple example would be a digital Etch-a-Sketch, where you might move a cursor around by pressing buttons (that control counters) and light up the pixels along the cursor's path. Raster graphics would be necessary for image processing applications such as Photoshop-like filtering. Raster graphics would also be appropriate for applications where individual pixels are generated algorithmically, such as Conway's Game of Life or other cellular automaton.

**Sprite animation** is best suited for applications where a block of pixels representing a glyph, icon, or other complex image generically called a "sprite" needs to pop up or move against a fixed background. Such images may include Pac-Man-like game characters, playing cards, or individual text characters. Raster graphics is generally cumbersome for this, since images need to be drawn or copied one pixel at a time. The basic idea behind implementing sprite graphics is that as the `vga_xy_controller` scans the rows and columns of the display, for each pixel location, the application determines whether the background image or sprite (or one of several sprites) should be displayed at that location. This may sound complicated, but in practice, it just involves using combinational logic to compare the sprite coordinates with the XY coordinates generated by the controller, and using the result to multiplex among multiple images stored in separate ROMs. A big advantage of this approach is that the user application can change sprite coordinates (say, under the control of a counter) and the display will automatically reflect the change. The disadvantage of sprite animation is that you need to keep track of the positions of all of the sprites individually, with individual registers or memory storing the coordinates.

Note that *you cannot change an image on a VGA display while your application is running by reading a different image file* using `$readmemh`, which is only for initializing memory contents when the design is compiled. There is no simple way of reading a file on the host computer from the DE2-115 board—any bitmapped images that you want to display from the DE2-115 board need to be initialized onto memories before you download the design to the board.

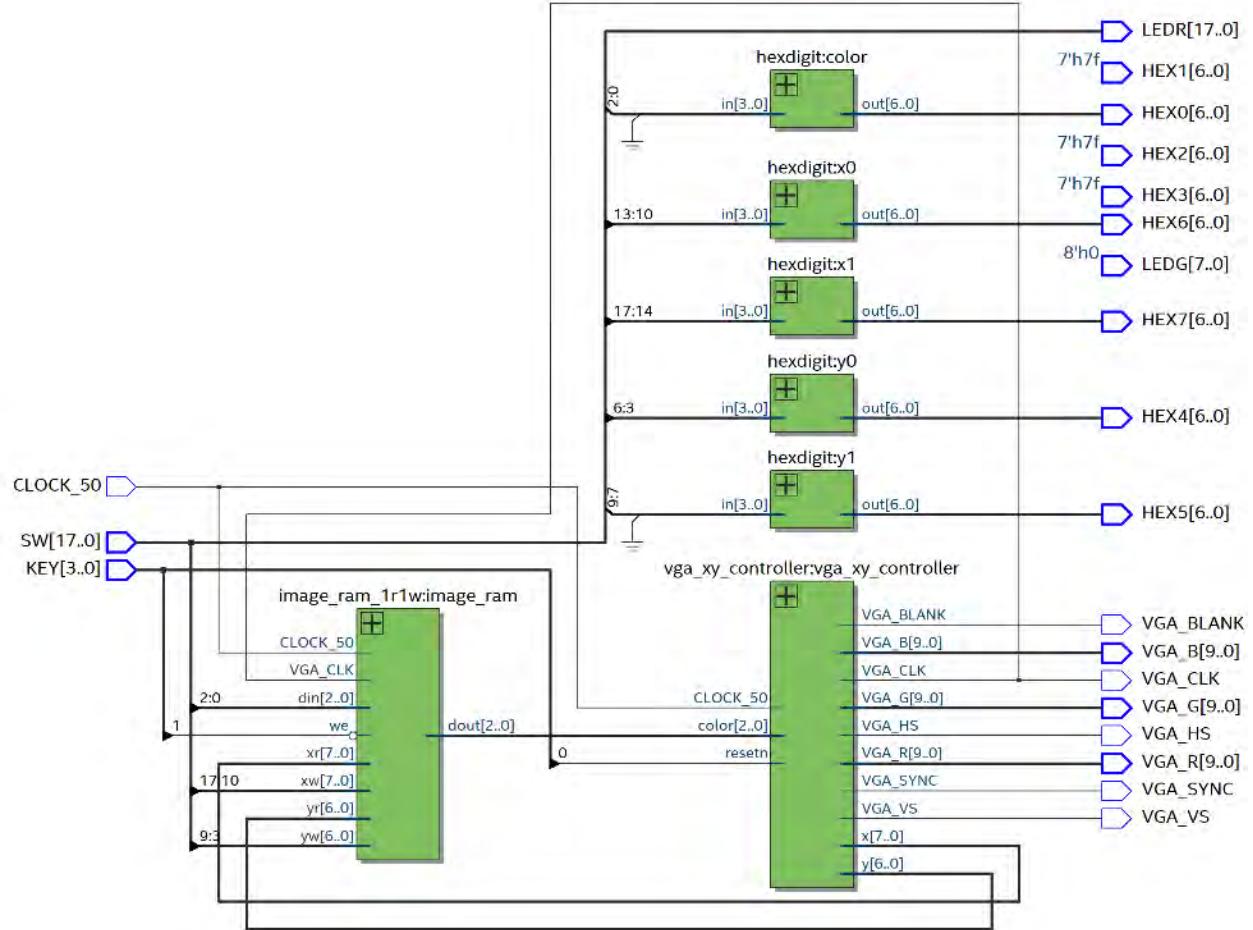
#### 12.5.5 Raster Graphics: Plotting Individual Pixels in an Image RAM

Using a RAM to store an image instead of a ROM makes it possible to change the color of individual pixels in the image by writing to the RAM. The following example illustrates this by connecting the address ports of an XY image RAM, as well as the 3-bit pixel color to switches, and the RAM write enable to a push button to manually light up individual pixels.

The DE2-115 port connections are as follows:

Demo Signal	Input Device	Display (hex format)
<code>reset_n</code>	<code>KEY[1]</code>	
<code>CLOCK_50</code>	<code>CLOCK_50</code>	
<code>color</code>	<code>SW[2:0]</code>	<code>HEX0</code>
<code>x</code>	<code>SW[17:10]</code>	<code>HEX7, HEX6</code>

y	SW[9:3]	HEX5, HEX4
plot	~KEY[0]	



The image RAM **image\_ram\_1r1w** has one read port and one write port. The read port is controlled by the **vga\_xy\_controller** and the write port is connected to buttons and switches. Note that the read and write ports have separate clocks: the read port is clocked by **VGA\_CLK** which is 25 MHz as required by the VGA spec, while the write clock uses the 50 MHz **CLOCK\_50**.

The Verilog model for **image\_ram\_1r1w** is as follows:

```
module image_ram_1r1w (
    input              CLOCK_50,
    input              VGA_CLK,
    input              we,
    input [2:0] din,
    input [7:0] xw,
    input [6:0] yw,
    input [7:0] xr,
    input [6:0] yr,
    output reg [2:0] dout
);
```

```

);
parameter IMAGE_FILE = "blank.mem";

wire [14:0] waddr = 160*yw + xw;
wire [14:0] raddr = 160*yr + xr;

reg [2:0] mem [0:19199];
initial $readmemh(IMAGE_FILE, mem);

always @(posedge CLOCK_50)
  if (we)
    mem[waddr] <= din;

always @(posedge VGA_CLK)
  dout <= mem[raddr];

endmodule

```

The Verilog model for the complete system connected to ports on the DE2-115 board is as follows:

```

module vga_ram_de2 (
  input      CLOCK_50,           // 50 MHz
  input [17:0] SW,
  input [3:0]  KEY,
  output [17:0] LEDR,
  output [7:0]  LEDG,
  output [6:0]  HEX0,
  output [6:0]  HEX1,
  output [6:0]  HEX2,
  output [6:0]  HEX3,
  output [6:0]  HEX4,
  output [6:0]  HEX5,
  output [6:0]  HEX6,
  output [6:0]  HEX7,
  output      VGA_CLK,          // VGA Clock
  output      VGA_HS,           // VGA H_SYNC
  output      VGA_VS,           // VGA V_SYNC
  output      VGA_BLANK,         // VGA BLANK
  output      VGA_SYNC,          // VGA SYNC
  output [9:0] VGA_R,            // VGA Red[9:0]
  output [9:0] VGA_G,            // VGA Green[9:0]
  output [9:0] VGA_B             // VGA Blue[9:0]
);

wire [7:0]  x_vga;
wire [6:0]  y_vga;
wire [2:0]  color_vga;

assign LEDR = SW;

```

```

assign LEDG = 0;
assign HEX1 = 7'h7f, HEX2 = 7'h7f, HEX3 = 7'h7f;

hexdigit x1 (
    .in    (SW[17:14]),
    .out   (HEX7)
);

hexdigit x0 (
    .in    (SW[13:10]),
    .out   (HEX6)
);

hexdigit y1 (
    .in    ({1'b0, SW[9:7]}),
    .out   (HEX5)
);

hexdigit y0 (
    .in    (SW[6:3]),
    .out   (HEX4)
);

hexdigit color (
    .in    ({1'b0, SW[2:0]}),
    .out   (HEX0)
);

image_ram_1r1w image_ram (
    .CLOCK_50      (CLOCK_50),
    .VGA_CLK       (VGA_CLK),
    .we            (~KEY[1]),
    .din           (SW[2:0]),
    .xw            (SW[17:10]),
    .yw            (SW[9:3]),
    .xr            (x_vga),
    .yr            (y_vga),
    .dout          (color_vga)
);

vga_xy_controller vga_xy_controller (
    .CLOCK_50      (CLOCK_50),
    .resetn        (KEY[0]),
    .color         (color_vga),
    .x             (x_vga),
    .y             (y_vga),
    .VGA_R         (VGA_R),
    .VGA_G         (VGA_G),
    .VGA_B         (VGA_B),
    .VGA_HS        (VGA_HS),

```

```

.VGA_VS      (VGA_VS),
.VGA_BLANK   (VGA_BLANK),
.VGA_SYNC    (VGA_SYNC),
.VGA_CLK     (VGA_CLK)
);

endmodule

```

To run the demo, follow these steps:

1. Connect the DE2-115 board to a VGA monitor using the VGA port in the middle of the back of the board (do not confuse this with the serial port that looks similar in the corner of the back of the board). When you turn it on, you should see the DE2-115 splash page appear on the display.
2. Synthesize the project `vga_ram_de2`, generate a programming file, and download it to the DE2-115. The VGA monitor should now display a blank background.
3. Make sure that all of the switches on the board are initially in the down position.
4. Set `SW[2:0]` to the up position and press `KEY[0]`. A white pixel should light up at (0, 0) in the top left corner of the board. Change the settings for `SW[2:0]` to pick a different color and press `KEY[0]` again. The color settings are listed in the table below and their values will appear on `HEX0`.

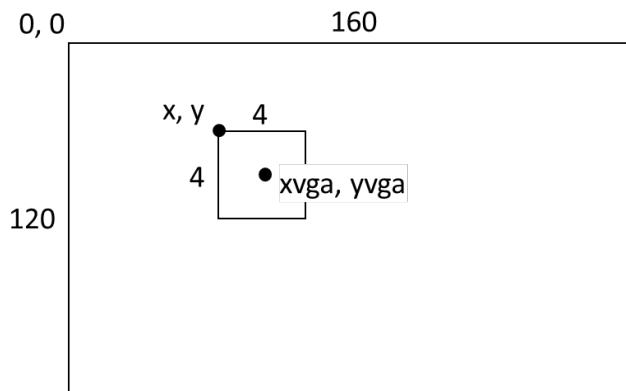
<code>SW[2:0]</code>	Color
0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	yellow
7	white

5. Use `SW[17:10]` to set the x coordinate and `SW[9:3]` to set the y coordinate for a pixel at (1, 1). The x coordinate should appear on hex displays (in hexadecimal) `HEX7`, `HEX6` and the y coordinate on `HEX5`, `HEX4`. Press `KEY[0]` to plot the pixel. Experiment by plotting pixels in various locations at different coordinates. Note that the maximum displayable x coordinate is 159 (hex 9F) and y coordinate is 119 (hex 77).

### 12.5.6 Sprite Graphics: Switching Between Multiple Images

In video game parlance, a “sprite” is a fixed 2D image that moves over a fixed background—think of the graphic animations in old school video games such as PAC-MAN and Donkey Kong. The basic idea behind sprite animation is that the background image and sprite image(s) are stored in separate memories. While the background image is the size of the full screen (160x120 pixels in our case), the sprite is much smaller and can be stored in a smaller memory, say 16x16 pixels. Given that the position of the sprite is known relative to the origin of the background image, as the VGA controller scans over the set of x,y coordinates of the display, combinational logic can be used to switch between the background memory and the sprite memory to determine which contains the pixel color that should be displayed at each coordinate.

For example, suppose that we have a 160x120 background image and a 4x4 sprite that we want to appear over the background at position x,y.



The following logic express determines if ( $xvga, yvga$ ) is inside of the sprite:

$$(xvga \geq x) \&\& (xvga < (x + 4)) \&\& (yvga \geq y) \&\& (yvga < (y + 4))$$

The background image and the sprite are stored in two different ROMs of different sizes. For the background ROM, the x-address is 8 bits and ranges 0-160 and the y-address is 7 bits and ranges 0-120. For the sprite ROM, both the x- and y-addresses are 4 bits and range 0-15.

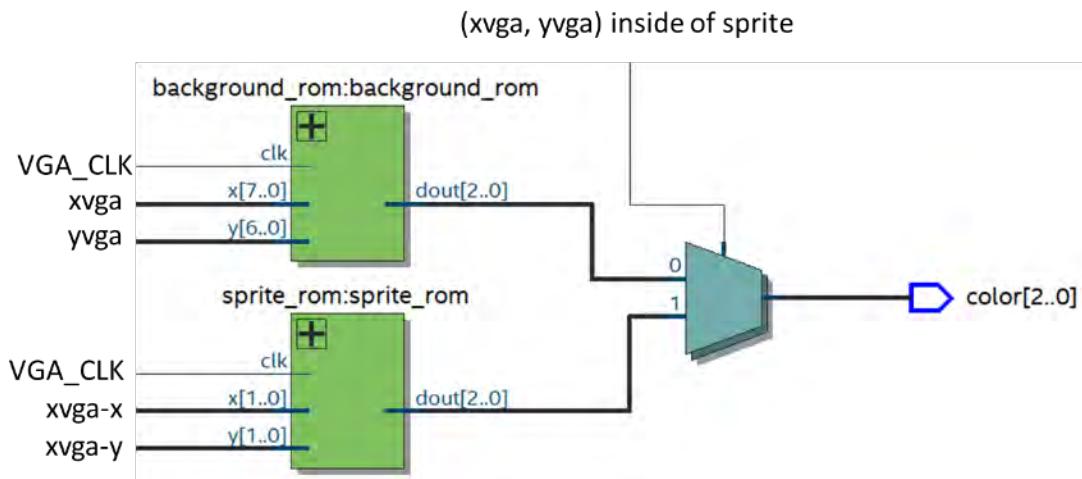
If ( $xvga, yvga$ ) is not inside of the sprite, then the pixel color in background memory at coordinates ( $xvga, yvga$ ) should be displayed. If ( $xvga, yvga$ ) is inside of the sprite, then the pixel color in the sprite memory at coordinates ( $xs, ys$ ) relative to the sprite origin should be displayed. From the diagram above, we can see that:

$$\begin{aligned} xs &= xvga - x \\ ys &= yvga - y \end{aligned}$$

Thus we can summarize the basic logic of the system as follows:

```
if (xvga, yvga) inside of sprite
    display color from sprite ROM at (xs, ys)
else
    display color from background ROM at (xvga, yvga)
```

We can implement this logic in hardware using a multiplexor:



The following Verilog module describes the complete hardware for displaying a sprite over a background image:

```

module sprite_test (
    input      VGA_CLK,
    input [7:0] x,
    input [6:0] y,
    input [7:0] xvga,
    input [6:0] yvga,
    output reg [2:0] color
);

wire [2:0] sprite_color, background_color;
wire [7:0] xs = xvga - x;
wire [6:0] ys = yvga - y;

always @(*) begin
    if ((xvga >= x) && (xvga < (x + 4)) && (yvga >= y) && (yvga < (y + 4)))
        color = sprite_color;
    else
        color = background_color;
end

sprite_rom sprite_rom (
    .clk (VGA_CLK),
    .x   (xs[1:0]),
    .y   (ys[1:0]),
    .dout (sprite_color)
);

background_rom background_rom (
    .clk (VGA_CLK),
    .x   (xvga),

```

```

    .y      (yvga),
    .dout  (background_color)
);
endmodule

```

The following Verilog module shows the complete connections of `sprite_test` to the `vga_xy_controller` and the DE2-115 board, using switches `SW[3:0]` to set the least-significant bits of the x-coordinate of the sprite and `SW[7:4]` to set the least-significant bits of the y-coordinate, with the most significant bits of both all set to 0.

```

module sprite_de2 (
    input      CLOCK_50,           // 50 MHz
    input [7:0] SW,
    input [0:0] KEY,
    output [17:0] LEDR,
    output     VGA_CLK,           // VGA Clock
    output     VGA_HS,            // VGA H_SYNC
    output     VGA_VS,            // VGA V_SYNC
    output     VGA_BLANK,          // VGA BLANK
    output     VGA_SYNC,           // VGA SYNC
    output [9:0] VGA_R,            // VGA Red[9:0]
    output [9:0] VGA_G,            // VGA Green[9:0]
    output [9:0] VGA_B             // VGA Blue[9:0]
);

wire [7:0] xvga;
wire [6:0] yvga;
wire [2:0] color;

assign LEDR [7:0] = SW[7:0];
assign LEDR [17:8] = 0;

sprite_test sprite_test (
    .VGA_CLK (VGA_CLK),
    .x      ({4'b0, SW[3:0]}),
    .y      ({3'b0, SW[7:4]}),
    .xvga  (xvga),
    .yvga  (yvga),
    .color   (color)
);

vga_xy_controller vga_xy_controller (
    .CLOCK_50      (CLOCK_50),
    .resetn        (KEY[0]),
    .color         (color),
    .x            (xvga),
    .y            (yvga),
    .VGA_R         (VGA_R),
    .VGA_G         (VGA_G),

```

```

.VGA_B          (VGA_B),
.VGA_HS         (VGA_HS),
.VGA_VS         (VGA_VS),
.VGA_BLANK      (VGA_BLANK),
.VGA_SYNC        (VGA_SYNC),
.VGA_CLK         (VGA_CLK)
);

endmodule

```

## 12.6 PS/2 Keyboard

### 12.6.1 Scan Codes

Each key on a PS/2 keyboard has an associated scan code, shown below in hex. For most of the keys, the scan code is 1 byte (8 bits). Some keys, such as the arrow keys, use an extended scan code that is 2 bytes, where the first byte is E0.

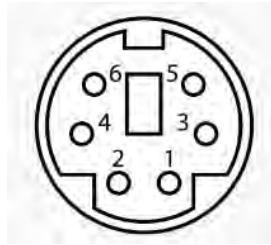
ESC 76	F1 05	F2 06	F3 04	F4 0C	F5 03	F6 0B	F7 83	F8 0A	F9 01	F10 09	F11 78	F12 07	↑ E0 75
~ 0E 16	!@ 1E 26	# 25	\$ 2E	% 2E 6^ 36	& 3D	* 3E	( 46	) 45	-_= 4E 55	=+ 55	BackSpace ← 66	→ E0 74	
TAB 0D	Q 15	W 1D	E 24	R 2D	T 2C	Y 35	U 3C	I 43	O 44	P 4D	[{ 54	}] 5B	\  5D
Caps Lock 58	A 1C	S 1B	D 23	F 2B	G 34	H 33	J 3B	K 42	L 4B	:: 4C	" 52	Enter ↓ 5A	↓ E0 72
Shift 12	Z 1Z	X 22	C 21	V 2A	B 32	N 31	M 3A	,< 41	/? 4A	Shift 59			
Ctrl 14	Alt 11				Space 29					Alt E0 11	Ctrl E0 14		

When a key with a non-extended scan code is pressed, the keyboard sends the 1-byte scan code in a serial data packet. When the key is released, it sends a packet with the data byte F0 followed by another packet with the scan code. For example, when the "A" key is pressed, the keyboard sends a packet with byte 1C and when it is released, it sends 2 packets F0 1C.

When a key with an extended scan code is pressed, it sends the scan code as 2 packets. When the key is released, it sends the two packets E0 F0 followed by the scan code. For example, when the up arrow is pressed it sends the two packets E0 75. When it is released, the keyboard sends the 3 packets E0 F0 75.

### 12.6.2 Packet Format

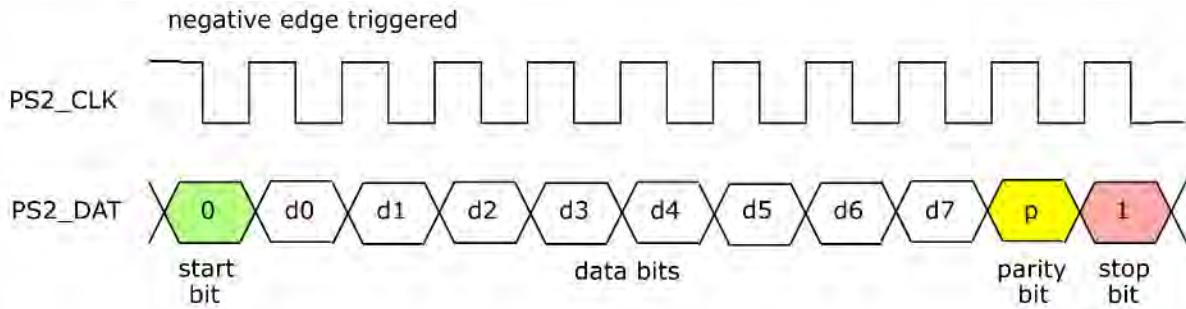
The PS/2 connector has 6 pins:



- 1: PS2\_DAT
- 2: not connected
- 3: ground
- 4: Vcc
- 5: PS2\_CLK
- 6: not connected

Packets are transmitted serially using the clock and data pins, where the data pin PS2\_DAT sends the data packet and the clock pin PS2\_CLK sends a negative edge-triggered clock centered on each bit of data. Data packets consist of 11 bits:

- start bit (0)
- data bits, least significant first
- parity bit (true if odd number of data bits = 1)
- stop bit (1)



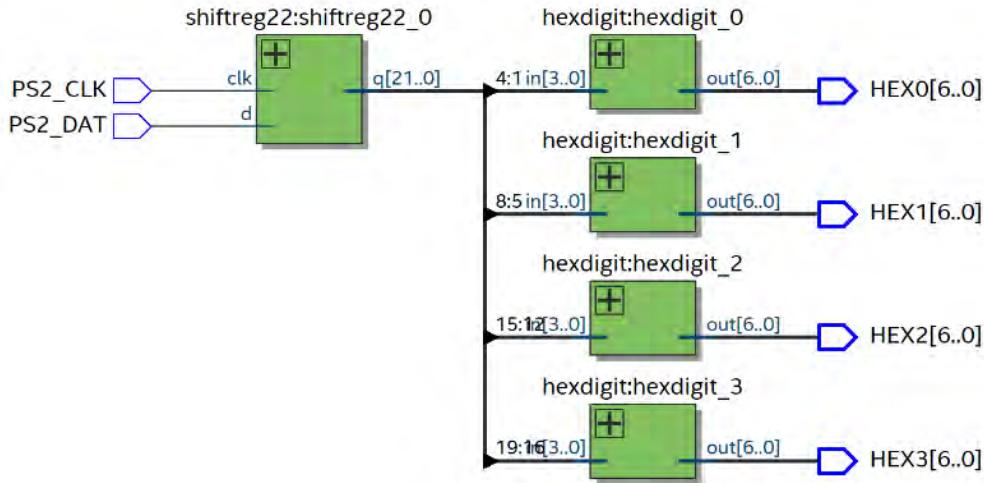
### 12.6.3 Serial-to-Parallel Conversion Using a Shift Register

Example Quartus Project: ps2\_shift\_register

#### 12.6.3.1 Simple but Naïve Approach

The standard way of receiving and using a serial packet is to convert the serial data to parallel data using a shift register. The simplest approach—although it has shortcomings we'll address later—is to use PS2\_CLK as a negative edge-triggered clock to the shift register and PS2\_DAT as the data in, and view the outputs of each of the stages of the shift register in parallel. To illustrate, we use a 22-bit shift register that is long enough to hold 2 successive packets and display the data bytes as hex digits. Shift register bits [19:16] displayed on HEX3 and bits [15:12] displayed on HEX2 are the 2 hex digits of the most-recently-sent scan code and bits [8:5] displayed on HEX1 and bits [4:1] displayed on HEX0 are the 2 hex digits of the previously-sent scan code. Thus, for example, a press and release of key “A” would result in F0 1C displayed on HEX3 through HEX0.

scan code byte 2 packet														scan code byte 1 packet													
21	20	19	18	17	16	15	14	13	12	1	0	9	8	7	6	5	4	3	2	1	0						
1	p	d7	d6	d5	d4	d3	d2	d1	d0	0	1	p	d7	d6	d5	d4	d3	d2	d1	d0	0						
HEX3						HEX2						HEX1						HEX0									



The Verilog code for this example is shown below:

```

module ps2_keyboard_demo (
    inout          PS2_CLK,
    inout          PS2_DAT,
    output [6:0]   HEX0,
    output [6:0]   HEX1,
    output [6:0]   HEX2,
    output [6:0]   HEX3);

    wire [21:0] q;

    shiftreg22 shiftreg22_0 (PS2_CLK, PS2_DAT, q);
    hexdigit hexdigit_3 (q[19:16], HEX3);
    hexdigit hexdigit_2 (q[15:12], HEX2);
    hexdigit hexdigit_1 (q[8:5], HEX1);
    hexdigit hexdigit_0 (q[4:1], HEX0);

endmodule

module shiftreg22 (
    input clk,
    input d,
    output reg [21:0] q);

    always @(negedge clk)
        q <= {d, q[21:1]};

endmodule

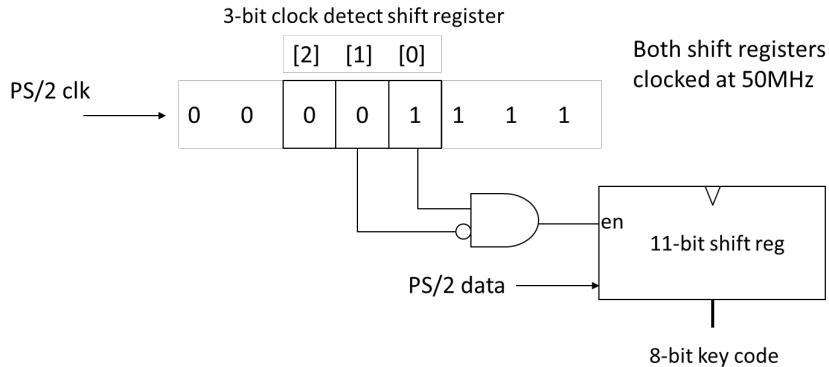
```

### 12.6.3.2 Synchronizing Keyboard Data to 50MHz System Clock

While the approach in the previous section does successfully acquire the serial PS/2 data and convert it to a parallel representation, the problem is that this operation isn't synchronous with the 50MHz system

clock, **CLOCK\_50**, which makes it difficult to reliably integrate the keyboard with the rest of the system that is all synchronized with the rising edge of **CLOCK\_50**.

One way to synchronize keyboard input with the system clock is to treat the **PS\_CLK** as data rather than as a clock, look for when it changes from a 1 to a 0, and then during that cycle acquire the data.



**PS2\_CLK** feeds into a 3-bit shift register that is clocked with **CLOCK\_50**. An **and** gate monitors bits [0] and [1] of the shift register and when they contain the values 1 and 0 respectively, this produces a pulse for one clock cycle that signals that a falling edge on **PS2\_CLK** was detected. This pulse enables a second shift register, also clocked with **CLOCK\_50**, that stores **PS2\_DATA**.

## 12.6.4 Complete Keycode Recognizer

Example Quartus Project: `ps2_keycode_recognizer`

### 12.6.4.1 Usage and Demonstration

Altera provides a **PS2\_Controller** module for communicating with a PS/2 keyboard or mouse.

It takes **PS2\_CLK**, **PS2\_DAT**, **CLK\_50**, and **reset** as input, and outputs the 8-bit **received\_data[7:0]** along with a 1 cycle wide pulse **received\_data\_en** when **received\_data[7:0]** is valid.<sup>13</sup>

It still takes as many as 3 data packets to receive a keycode, and while the Altera **PS2\_Controller** module simplifies communication with the keyboard, it is still the user's responsibility to recognize when a complete keycode is transmitted and to determine whether it represents a press, release, or extended key event. I've written a **keycode\_recognizer** module that uses a finite state machine to automate this process.

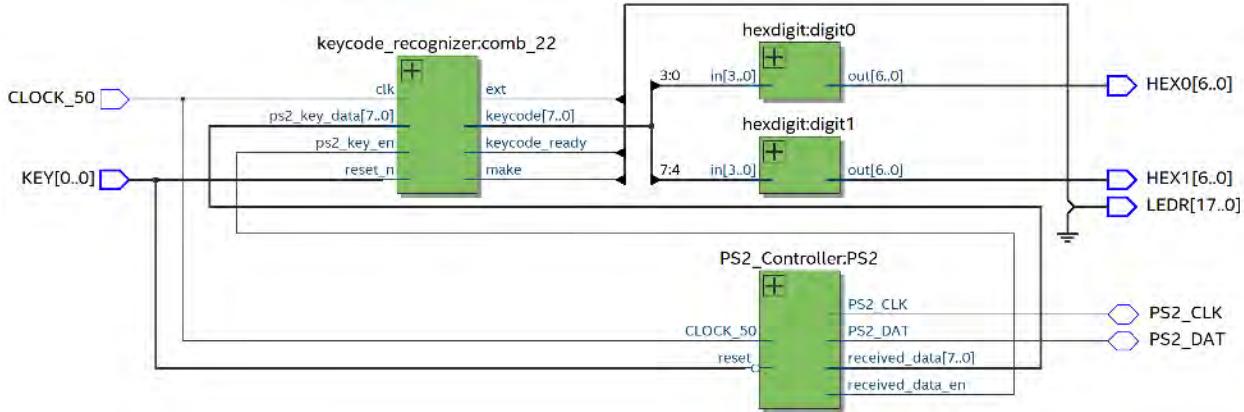
The ports on **keycode\_recognizer** are as follows:

Name	Direction	Description
<b>clk</b>	input	system clock (connect to <b>CLOCK_50</b> )
<b>reset_n</b>	input	active-low reset
<b>ps2_key_en</b>	input	enable signal from Altera <b>PS2_Controller</b>
<b>ps2_key_data[7:0]</b>	input	data from <b>PS2_Controller</b>
<b>keycode[7:0]</b>	output	recognized keycode data
<b>ext</b>	output	1 if extended keycode, 0 otherwise
<b>make</b>	output	1 if key press (make), 0 if key release (break)

<sup>13</sup> It also provides a bunch more ports for sending data to a PS/2 device. See Altera documentation for details.

<b>keycode_ready</b>	<b>output</b>	1 cycle wide pulse when keycode has been recognized
----------------------	---------------	---

The module `keycode_recognizer_demo` demonstrates the use of `keycode_recognizer` with the Altera PS2\_Controller.



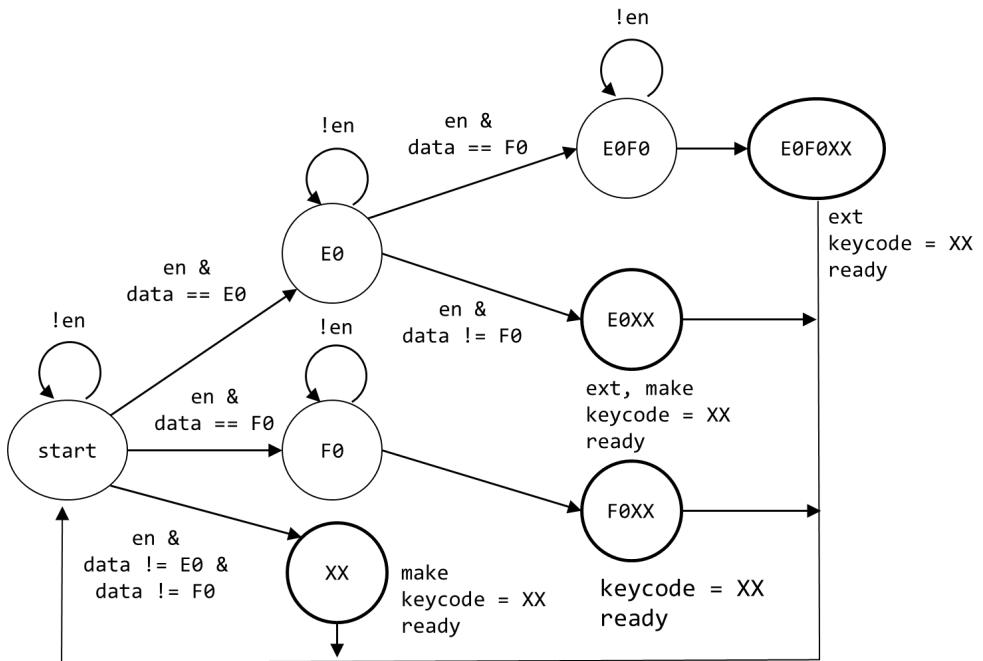
1. The module `PS2_Controller` uses modules `Altera_UP_PS2_Data_In` and `Altera_UP_PS2_Command_Out`. These must also be included in the project for it to synthesize.
2. 8-bit keycode data is displayed on `HEX1-HEX0`.
3. `make` is displayed on `LEDR[0]`, which is active following a key press event
4. `ext` is displayed on `LEDR[1]`, which is active when an extended key (such as an arrow key or “Ctrl”) is pressed or released.
5. `keycode_ready` is connected to `LEDR[2]`, but since it is only high for 1 clock cycle (1/50,000,000 seconds) you won’t actually see it blink.

#### 12.6.4.2 Recognizer Finite State Machine

`keycode_recognizer` uses a finite state machine to recognize a keyboard event. Keycodes transmission require 1, 2, or 3 bytes of data, depending on whether a key is pressed or released, or whether it is an extended key. There are 4 possible keycode formats, where XX represents the actual byte of keycode data:

Format	Meaning	make	ext
XX	press non-extended	1	0
F0XX	release non-extended	0	0
E0XX	press extended	1	1
E0F0XX	release extended	0	1

The figure below shows the finite state machine for recognizing each of these keycode formats. It begins in the start state and waits for `ps2_key_en` (labeled en). When en does go high, depending on the value of `ps2_key_data[7:0]` (labelled data), it transitions to either state E0, F0, or XX. If data does not equal either F0 or E0, then the keyboard event must have been a press of a non-extended key. At that point, it sets `make` to 1, outputs XX on `keycode`, and sets `keycode_ready` (labelled ready) to 1 before transitioning back to the `start` state. For the other cases, the FSM continues to ready PS/2 packets until it recognizes one of the formats in the table above.



The Verilog code for the FSM is given below. Note that the FSM outputs `keycode`, `ext`, `make`, and `keycode_ready` are buffered in registers (stored in flip flops) rather than being passed to the outside world directly as combinational logic.

```

module keycode_recognizer (
    input          clk,
    input          reset_n,
    input          ps2_key_en,
    input [7:0]    ps2_key_data,
    output reg [7:0] keycode,
    output reg     ext,
    output reg     make,
    output reg     keycode_ready
);

reg [7:0]    keycode_d;
reg          ext_d;
reg          make_d;
reg          keycode_ready_d;

always @(posedge clk) begin
    keycode_ready_d <= keycode_ready;
    if (keycode_ready_d) begin
        keycode_d      <= keycode;
        ext_d          <= ext;
        make_d         <= make;
    end
end

```

```

parameter S_START      = 3'd0;
parameter S_F0          = 3'd1;
parameter S_E0          = 3'd2;
parameter S_E0F0        = 3'd3;
parameter S_XX          = 3'd4;
parameter S_F0XX        = 3'd5;
parameter S_E0XX        = 3'd6;
parameter S_E0F0XX      = 3'd7;

reg [2:0] state = S_START;
reg [2:0] next_state;

always @(posedge clk)
  if (!reset_n)
    state <= S_START;
  else
    state <= next_state;

always @(*) begin
  keycode_d           = 0;
  ext_d               = 0;
  make_d              = 0;
  keycode_ready_d    = 0;
  case (state)
    S_START: begin
      if (!ps2_key_en)
        next_state = S_START;
      else begin
        if (ps2_key_data == 8'hE0)
          next_state = S_E0;
        else if (ps2_key_data == 8'hF0)
          next_state = S_F0;
        else
          next_state = S_XX;
      end
    end
    S_F0: begin
      if (!ps2_key_en)
        next_state = S_F0;
      else begin
        next_state = S_F0XX;
      end
    end
    S_E0: begin
      if (!ps2_key_en)
        next_state = S_E0;
      else begin
        if (ps2_key_data == 8'hF0)

```

```

        next_state = S_E0F0;
    else
        next_state = S_E0XX;
    end
end

S_E0F0: begin
    if (!ps2_key_en)
        next_state = S_E0F0;
    else begin
        next_state = S_E0F0XX;
    end
end

S_XX: begin
    keycode_d = ps2_key_data;
    make_d      = 1;
    keycode_ready_d = 1;
    next_state   = S_START;
end

S_F0XX: begin
    keycode_d      = ps2_key_data;
    keycode_ready_d = 1;
    next_state     = S_START;
end

S_E0XX: begin
    keycode_d      = ps2_key_data;
    make_d      = 1;
    ext_d       = 1;
    keycode_ready_d = 1;
    next_state   = S_START;
end

S_E0F0XX: begin
    keycode_d      = ps2_key_data;
    ext_d       = 1;
    keycode_ready_d = 1;
    next_state   = S_START;
end

default: next_state = S_START;
endcase
end
endmodule

```

### 12.6.5 Buffering a Keypress Signal Until You Need It

In the `keycode_recognizer` module, the `keycode_ready` port only goes high for 1 clock cycle when a key is pressed (or released) and a code is recognized. For some applications, however, it is necessary to “hold on” to the ready signal until it is needed. For example, in a video game application that uses keyboard input, a play may press a key at any time, but the game finite state machine (FSM) may not be in a state that is ready to use the keyboard data at the time that the key is pressed. To solve this problem, it is necessary to store the fact that a keypress was received in a buffer register until the time that it is needed by the FSM and then have the FSM clear the buffer after it has used the data to be ready for the next keypress.

A Verilog model for the `keypress_buffer` is shown below. It is similar to a D flip flop with clear and enable ports, except that it always writes a 1 when enable is asserted, rather than having a data input.

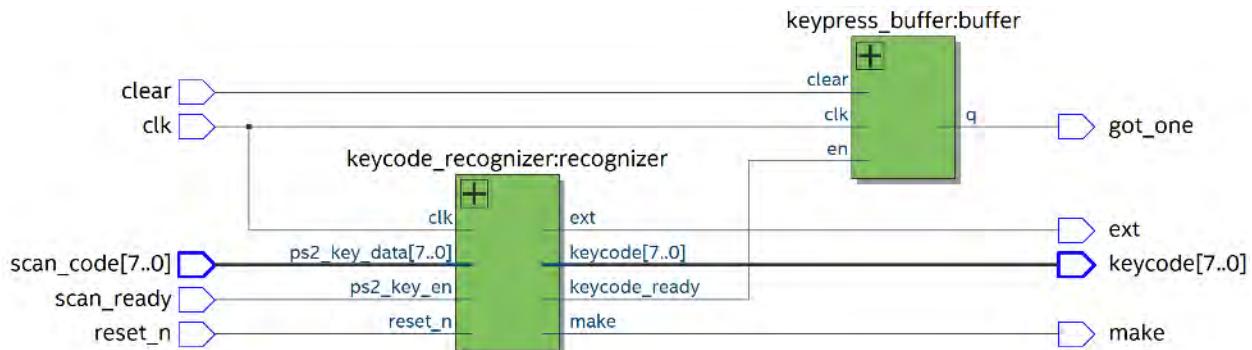
```
module keypress_buffer (
    input      clk,
    input      en,
    input      clear,
    output reg q);

    initial q = 0;

    always @(posedge clk)
        if (clear)
            q <= 0;
        else if (en)
            q <= 1;

endmodule
```

The schematic below illustrates the use of the `keypress_buffer`. The `keycode_ready` signal from the `keycode_recognizer` is connected to the `en` input of the buffer. When `keycode_ready` is high, the buffer will output a 1 signaling that it “got one”. The `got_one` signal will stay high until the `clear` signal is asserted, presumably after the application has processed the keycode.

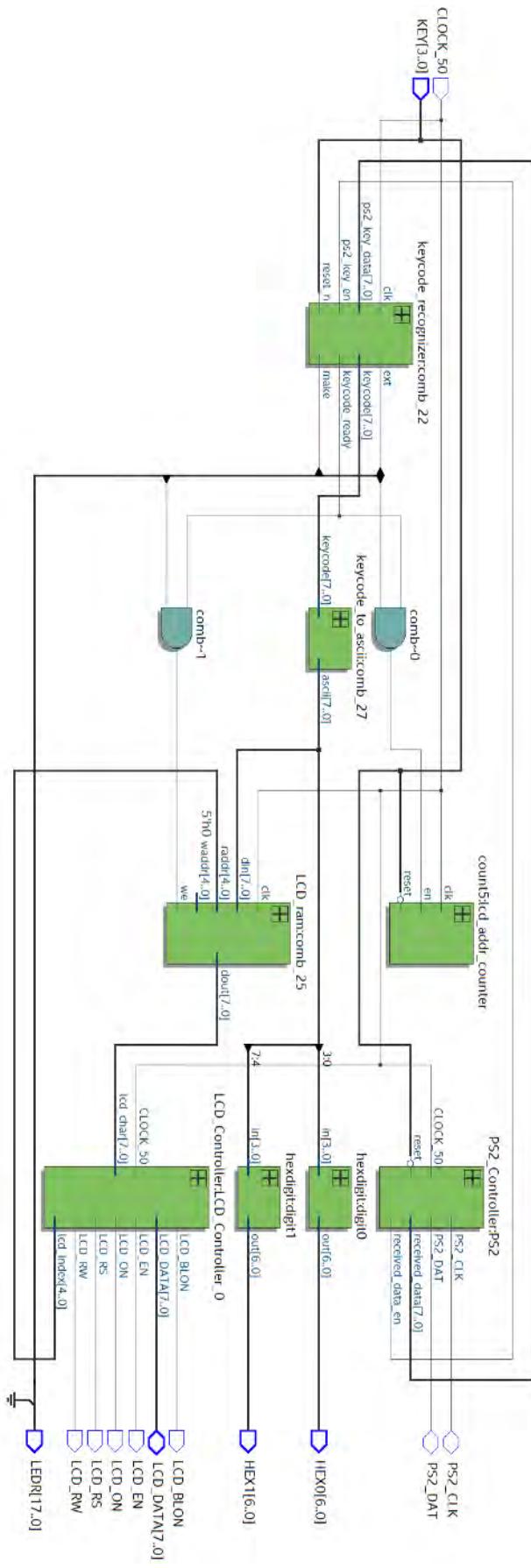


## 12.6.6 Example: Typing Characters from Keyboard to LCD Display

### Example Quartus Project: ps2\_keyboard\_to\_lcd

As a more complete example of the use of the `keycode_recognizer`, the following top-level DE2-115 module `ps2_keyboard_to_lcd_de2` takes input from the PS/2 keyboard, converts it to ASCII characters, and displays them on the LCD display.

- A combinational logic module, `keycode_to_ascii`, translates PS/2 keycodes to ASCII codes for letter, number, and space keys. The module outputs “.” for any other key.
- The ASCII character produced by `keycode_to_ascii` is written to the `LCD_ram`. The write enable of `LCD_ram` is asserted when `keycode_ready` and `make` from the `keycode_recognizer` are both 1, that is, when a keycode from a key press has been recognized.
- The write address `waddr` of the `LCD_ram` is tied to a 5-bit counter, `count5`. The enable input `en` of `count5` is also asserted when `keycode_ready` and `make` from the `keycode_recognizer` are both 1, meaning that the `LCD_ram` write address increments following each write.
- The ASCII values written to `LCD_ram` are displayed as hexadecimal digits on `HEX1` and `HEX0`.
- The LCD display interface module `LCD_Controller` provides the read address for `LCD_ram` as the index `lcd_index` where the character is to be displayed on the LCD and receives the data output from the `LCD_ram` as the character to be displayed `lcd_char`.
- The demonstration starts with an initial message on the LCD that is overwritten as you type on the keyboard.



The Verilog code for modules `keycode_to_ascii` and `ps2_keyboard_to_lcd_de2` are as follows:

```
module keycode_to_ascii (
    input      [7:0] keycode,
    output reg   [7:0] ascii
);

    always @(*)
        case (keycode)
            8'h1C: ascii = "A";
            8'h32: ascii = "B";
            8'h21: ascii = "C";
            8'h23: ascii = "D";
            8'h24: ascii = "E";
            8'h2B: ascii = "F";
            8'h34: ascii = "G";
            8'h33: ascii = "H";
            8'h43: ascii = "I";
            8'h3B: ascii = "J";
            8'h42: ascii = "K";
            8'h4B: ascii = "L";
            8'h3A: ascii = "M";
            8'h31: ascii = "N";
            8'h44: ascii = "O";
            8'h4D: ascii = "P";
            8'h15: ascii = "Q";
            8'h2D: ascii = "R";
            8'h1B: ascii = "S";
            8'h2C: ascii = "T";
            8'h3C: ascii = "U";
            8'h2A: ascii = "V";
            8'h1D: ascii = "W";
            8'h22: ascii = "X";
            8'h35: ascii = "Y";
            8'h1A: ascii = "Z";
            8'h45: ascii = "0";
            8'h16: ascii = "1";
            8'h1E: ascii = "2";
            8'h26: ascii = "3";
            8'h25: ascii = "4";
            8'h2E: ascii = "5";
            8'h36: ascii = "6";
            8'h3D: ascii = "7";
            8'h3E: ascii = "8";
            8'h46: ascii = "9";
            8'h29: ascii = " ";
            default: ascii = ".";
        endcase
    endmodule
```

```

module ps2_keyboard_to_lcd_de2 (
    input          CLOCK_50,
    input  [3:0]    KEY,
    inout         PS2_CLK,
    inout         PS2_DAT,
    output  [6:0]   HEX0,
    output  [6:0]   HEX1,
    output  [17:0]  LEDR,
    output          LCD_ON,
    output          LCD_BLON,
    output          LCD_RW,
    output          LCD_EN,
    output          LCD_RS,
    inout  [7:0]   LCD_DATA
);

wire      [7:0] ps2_key_data;
wire      ps2_key_en;
wire      keycode_ready;
wire      [7:0] keycode;
wire      [7:0] ascii;
wire      [7:0] lcd_char;
wire      [4:0] lcd_index;
wire      ext;
wire      make;
wire      [4:0] waddr;

assign LEDR[17:2] = 0;
assign LEDR[1]    = ext;
assign LEDR[0]    = make;

PS2_Controller PS2 (
    .CLOCK_50          (CLOCK_50),
    .reset             (~KEY[0]),
    .PS2_CLK           (PS2_CLK),
    .PS2_DAT           (PS2_DAT),
    .received_data     (ps2_key_data),
    .received_data_en  (ps2_key_en)
);

keycode_recognizer (
    .clk              (CLOCK_50),
    .reset_n          (KEY[0]),
    .ps2_key_en       (ps2_key_en),
    .ps2_key_data     (ps2_key_data),
    .keycode          (keycode),
    .ext              (ext),
    .make             (make),
    .keycode_ready    (keycode_ready)
);

```

```

count5 lcd_addr_counter (
    .clk      (CLOCK_50),
    .reset    (~KEY[0]),
    .en       (keycode_ready & make),
    .q        (waddr)
);

LCD_ram (
    .clk      (CLOCK_50),
    .raddr   (lcd_index),
    .din     (ascii),
    .waddr   (waddr),
    .we      (keycode_ready & make),
    .dout    (lcd_char)
);

LCD_Controller LCD_Controller_0 (
    .lcd_char  (lcd_char),
    .lcd_index (lcd_index),
    .CLOCK_50  (CLOCK_50 ),
    .LCD_ON    (LCD_ON   ),
    .LCD_BLON  (LCD_BLON ),
    .LCD_RW    (LCD_RW   ),
    .LCD_EN    (LCD_EN   ),
    .LCD_RS    (LCD_RS   ),
    .LCD_DATA  (LCD_DATA )
);

 keycode_to_ascii (
    .keycode      (keycode),
    .ascii        (ascii)
);

hexdigit digit1 (
    .in          (ascii[7:4]),
    .out         (HEX1)
);

hexdigit digit0 (
    .in          (ascii[3:0]),
    .out         (HEX0)
);

endmodule

```

## 12.7 Audio

Example Quartus Project: **Audio\_Demo**

### 12.7.1 University of Toronto Audio Controller

The University of Toronto has developed an audio controller module, built upon Altera University Program audio modules that simplifies the input and output of audio signals on the Terasic/Altera DE2-115 board. The following is reprinted with permission from Jonathan Rose from:

[http://www.eecg.toronto.edu/~jayar/ece241\\_08F/AudioVideoCores/audio/audio.html](http://www.eecg.toronto.edu/~jayar/ece241_08F/AudioVideoCores/audio/audio.html)

The audio controller provides a simple interface to the Audio CODEC chip present on the DE2 board. The controller handles the data transmission to and from the chip. The chip configuration is handled by the separate configuration module. The configuration module must be instantiated separately when using the audio controller.

To use the University of Toronto audio controller and configuration module, include the following files in your design:

- Altera\_UP\_Audio\_Bit\_Counter.v
- Altera\_UP\_Audio\_In\_Deserializer.v
- Altera\_UP\_Audio\_Out\_Serializer.v
- Altera\_UP\_Clock\_Edge.v
- Altera\_UP\_SYNC\_FIFO.v
- Audio\_Clock.v
- Audio\_Controller.v
- avconf.v
- I2C\_Controller.v

#### 12.7.1.1 Controller Interface

##### 12.7.1.1 Port Descriptions

The audio controller **Audio\_Controller** interface is illustrated below, with inputs shown on the left, outputs and bidirectional lines on the right. By default, **AUDIO\_DATA\_WIDTH** is equal to 32.



- `CLOCK_50` - system clock input, must be 50MHz for the timing control to work properly.
- `reset` - the active-high reset.
- `AUD_ADCDAT`, `AUD_DACDAT`, `AUD_BCLK`, `AUD_ADCLRCK`, `AUD_DACLRCK`, `I2C_SDAT`, `I2C_SCLK` and `AUD_XCK` - off-chip lines to be connected to the correspondingly named DE2-115 pins
- `clear_audio_in_memory` - clear the audio input buffer.
- `clear_audio_out_memory` - clear the audio output buffer.

#### Ports for receiving data

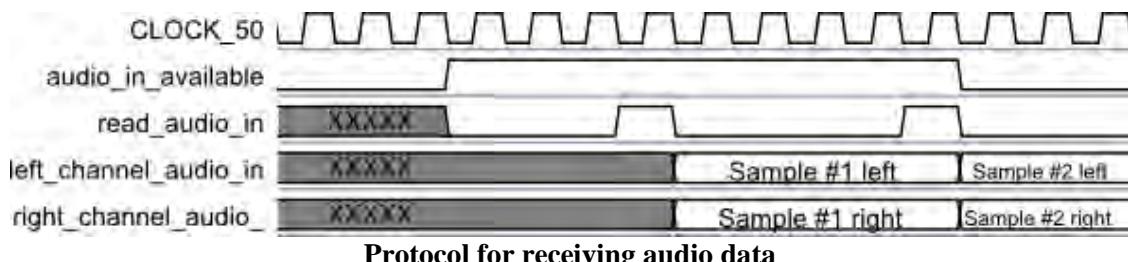
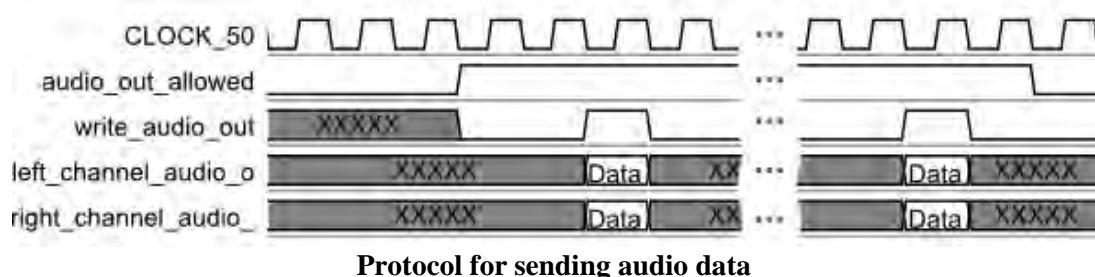
- `left_channel_audio_in` and `right_channel_audio_in` - Audio data received from the external source.
- `read_audio_in` - Read enable signal. The audio input data, if available, will be placed on the data lines on the next clock cycle. This is a level-sensitive signal, which means a new sample of data will be retrieved on every edge, as long as `read_audio_in` is high.
- `audio_in_available` - indicates whether the input data is available or not. Reads will have no effect unless this signal is high.

#### Ports for sending data

- `left_channel_audio_out` and `right_channel_audio_out` - Audio data for playback.
- `write_audio_out` - enable signal for writing the new data. Level-sensitive, data is written on every clock edge when this signal is high.
- `audio_out_allowed` - indicates when the data may be written. Write will have no effect unless this signal is high.

### **12.7.1.1.2 Interface Description**

`Audio_Controller` is capable of a full-duplex audio input and output. The data ports are 32-bit wide by default and are connected to the data buffers. The `clear_audio_in_memory` and `clear_audio_out_memory` signals may be used to clear the buffers, and the `audio_in_available` and `audio_out_allowed` signals indicate the availability of the data (in the case of input) or the free space (in the case of output) in the buffers. The data itself is a signed integer representing one audio sample. All the signals are synchronized to the same clock. The interface protocol for sending and receiving data is illustrated below:



### 12.7.1.2 Controller Operation

The audio controller consists of two main parts: the input module and the output module. This section will provide a short overview of each.

The audio input and output modules consist of the shift registers connected to the data buffers.

In the case of the audio output, the data received from the user is buffered and then shifted-out to the audio chip at the appropriate rate. The audio chip then feeds this data directly to the DACs.

In the case of the audio input, the process is reversed: the data received from the audio chip is shifted-in and placed in the data buffers. The data comes directly from the ADCs on-board the audio chip.

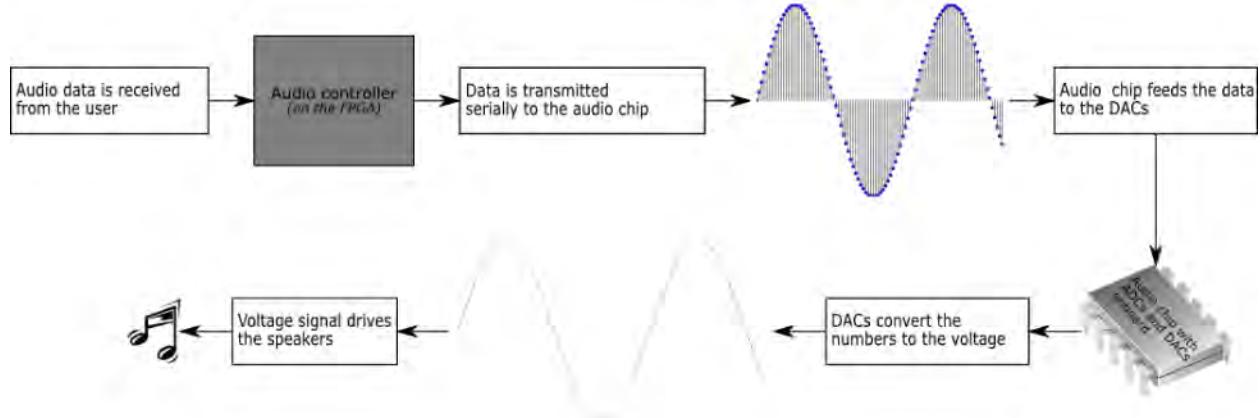
### 12.7.1.3 Audio Data Format and Analog-Digital Conversion

The audio controller uses the raw pulse code modulated (PCM) data streams, both for the input and for the output. The PCM data stream is essentially a sequence of numbers representing the intensity of the signal at a given moment. Each of these numbers is called a sample. The sound may be represented (i.e., sampled) by a sequence of the samples. This sequence has a frequency associated with it, which tells the rate at which the original signal was sampled. This sampling rate is necessary for the correct signal reconstruction.

For the audio controller the default sampling rate is 48kHz with a default sample size of 32 bits. Furthermore, there are 2 channels both for the input and for the output. The sampling rate and sample size may be changed at configuration time if necessary. The audio samples are presented as the 2's-complement signed integers.

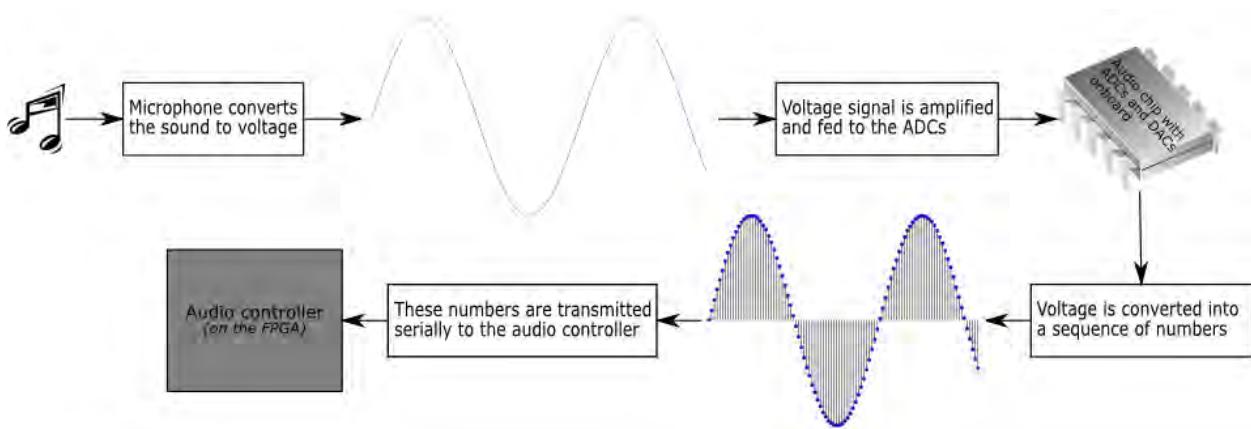
#### 12.7.1.3.1 Digital to Analog Conversion

For the audio playback, the PCM data is fed directly to the DACs, which convert the value to a voltage. This analog voltage output is connected to the Line-out jack on the DE2 board which can then drive the headphones or the speakers. This process is illustrated below:



#### 12.7.1.3.2 Analog to Digital Conversion

For the audio input (or recording), the process is reversed. You can have either the microphone connected to the Mic jack on the DE2 board, or some other device connected to the Line-in jack (but not both at the same time). These jacks are connected to the inputs of the ADCs, which convert the voltage to the digital value. This value is then transmitted to the audio controller. This process is illustrated below:



## 12.7.2 University of Toronto Audio and Video-In Configuration Module

The following is reprinted with permission from Jonathan Rose of the University of Toronto from

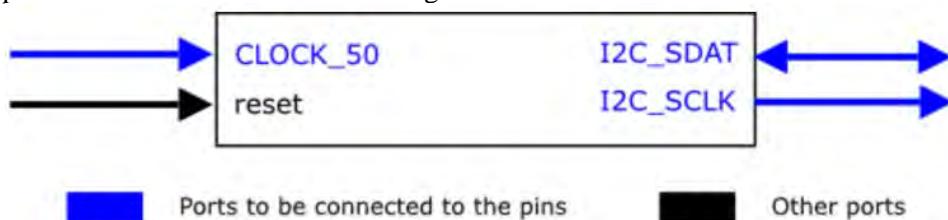
[http://www.eecg.toronto.edu/~jayar/ece241\\_08F/AudioVideoCores/avconf/avconf.html](http://www.eecg.toronto.edu/~jayar/ece241_08F/AudioVideoCores/avconf/avconf.html)

The configuration module performs the startup configuration for both audio and video decoder chips. It is needed if you are using the audio controller or the video-in controller.

### 12.7.2.1 Module Interface

#### 12.7.2.1.1 Port Descriptions

The audio and video-in configuration module `avconf` interface is illustrated below, with inputs shown on the left, outputs and bidirectional lines on the right.



- `CLOCK_50` - system clock input, must be 50MHz for the timing control to work properly.
- `reset` - the active-high reset.
- `I2C_SDAT` and `I2C_SCLK` - off-chip lines to be connected to the correspondingly named pins

#### 12.7.2.1.2 Module Parameters

- `USE_MIC_INPUT` - Select the audio source: 0 to select Line-in, 1 to select Microphone.

#### 12.7.2.1.3 Interface Description

The module is completely self-contained and only needs to be instantiated in your circuit for the configuration to be performed. The configuration is performed automatically every time the module is reset.

### 12.7.2.2 Module Operation

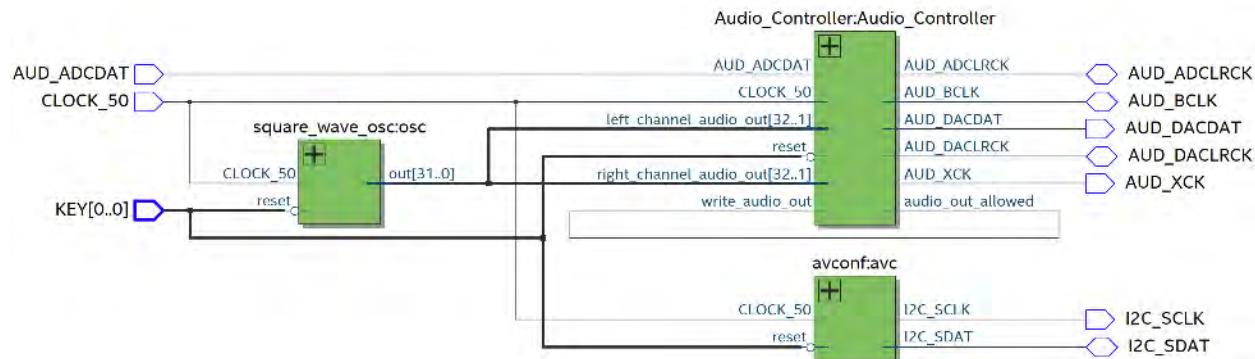
The configuration module performs the configuration for both the audio and video chips using the same data bus. The default parameters are hard coded into the module, and described in the audio controller and the video-in controller documentation. The default parameters will be acceptable for most users, and therefore the understanding or modification of the configuration module is not required. By default, `avconf` sets the ADC and DAC sample clock rates (AUD\_ADCLRCK and AUD\_DACLRCK) to 48 kHz.

### 12.7.3 Square Wave Synthesis Demo

This example shows how to generate a 440 Hz (concert A pitch) square wave tone using a digital oscillator and the University of Toronto audio modules.

#### 12.7.3.1 Top-Level Design and Operation

A schematic of the top-level module `Square_Wave_Gen_Demo` is shown below:



The output `out` of the oscillator `square_wave_osc` is a 32-bit signal representing the height of the wave at each point in time. The oscillator output connects to the left and right audio output channels of the `Audio_Controller`, `left_channel_audio_out` and `right_channel_audio_out`. The `Audio_Controller` output `audio_out_allowed` is connected back around to the input `write_audio_out` so that audio output data is only written when the controller is ready to accept it. The remaining port on `Audio_Controller`, specifically those involved with audio input from either the microphone or line-in are unused and not shown. The audio-video configuration module `avconf` is required to load the audio controller with default settings via the I<sup>2</sup>C channel at system startup.

To hear the tone, plug earbuds or a speaker in to the Line-Out (green) jack on the DE2-115 board.

The Verilog code for the complete `Square_Wave_Gen_Demo` is:

```
module Square_Wave_Gen_Demo (
    input      CLOCK_50,
    input      AUD_ADCDAT,
    inout     AUD_BCLK,
    inout     AUD_ADCLRCK,
    inout     AUD_DACLRCK,
    inout     I2C_SDAT,
    output     AUD_XCK,
    output     AUD_DACDAT,
    output     I2C_SCLK
);
```

```

wire           audio_out_allowed;
wire      [31:0]   osc_out;

square_wave_osc osc (
    .CLOCK_50          (CLOCK_50),
    .out               (osc_out)
);

Audio_Controller Audio_Controller (
    // Inputs
    .CLOCK_50          (CLOCK_50),
    .reset              (1'b0),
    .left_channel_audio_out (osc_out),
    .right_channel_audio_out (osc_out),
    .write_audio_out    (audio_out_allowed),
    .AUD_ADCDAT        (AUD_ADCDAT),
    // Bidirectionals
    .AUD_BCLK           (AUD_BCLK),
    .AUD_ADCLRCK       (AUD_ADCLRCK),
    .AUD_DACLRCK       (AUD_DACLRCK),
    // Outputs
    .audio_out_allowed (audio_out_allowed),
    .AUD_XCK            (AUD_XCK),
    .AUD_DACDAT         (AUD_DACDAT),
);
avconf avc (
    .I2C_SCLK           (I2C_SCLK),
    .I2C_SDAT           (I2C_SDAT),
    .CLOCK_50            (CLOCK_50),
    .reset               (1'b0)
);
endmodule

```

### 12.7.3.2 Oscillator Design

The oscillator `square_wave_osc` uses a counter to generate a waveform that flips the output between a high and low value and 1 every `HALF_PERIOD` clock cycles. Since the input clock, `CLOCK_50` is 50 MHz and the desired output frequency is 440 Hz,

$$\text{HALF\_PERIOD} = (50,000,000/440) / 2 = 56,818 \text{ cycles}$$

The actual output `out` of the oscillator is a 32-bit signed value representing the amplitude of the wave that goes to the left and right audio output ports of the `Audio_Controller`. We've selected an `AMPLITUDE` of 10,000, which is loud enough to be heard.

The Verilog code for `square_wave_osc` is as follows:

```

module square_wave_osc (
    input      CLOCK_50,
    output reg [31:0] out);

initial out = 0;

parameter HALF_PERIOD    = 20'd56_818;    // for 440 Hz (50,000,000/440)/2
parameter AMPLITUDE       = 32'd10_000_000;
reg [19:0] count = 0;

always @(posedge CLOCK_50)
    if (count == HALF_PERIOD)
        count <= 0;
    else
        count <= count + 1;

always @(posedge CLOCK_50)
    if (count == HALF_PERIOD)
        if (out == 0)
            out <= AMPLITUDE;
        else
            out <= 0;

endmodule

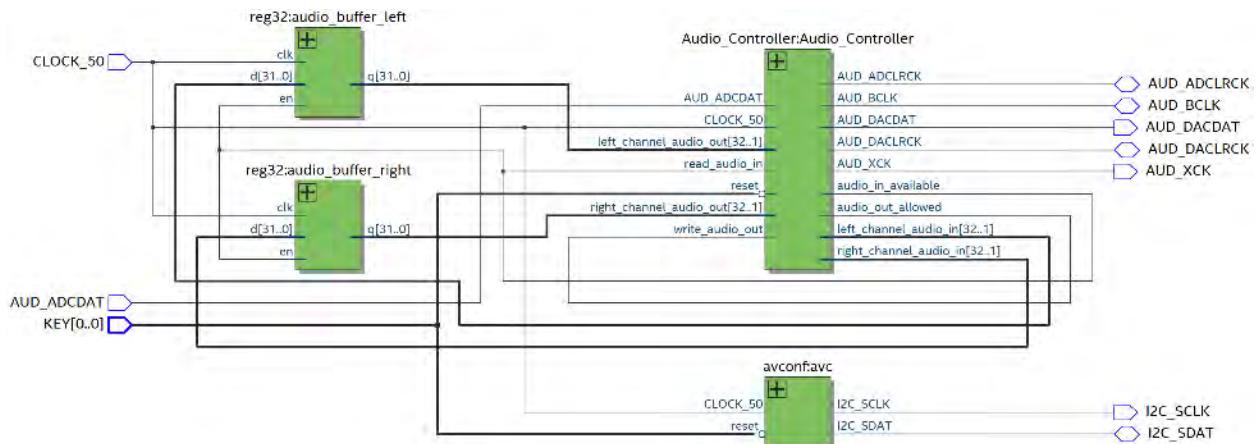
```

#### 12.7.4 Microphone to Line-Out Demo

This example demonstrates the process of acquiring audio from a microphone (or line-in), doing something with it, and then outputting it through the line-out jack. Specifically, the `Audio_Controller` first reads the audio input from the microphone, converts the analog voltage to a 32-bit digital value, and writes the data to a buffer register. It then reads the 32-bit value from the buffer register, converts it back to an analog signal and outputs it to line-out. Note that in this example, “doing something” with the data consists of simply holding it in the buffer and then writing it back out; a more complex application that builds on this example would operate on the data before writing it back, such as by filtering it, distorting it, or recording it in memory.

To run the demo on the board, connect a microphone to the microphone (red) jack and earbuds or speakers to the line-out (green) jack on the DE2-115 board. Note that a second pair of earbuds can be used as a serviceable microphone!

A schematic of the top-level module `Mic_to_Line_Out_Demo` is shown below:



The key connections are:

- `left_channel_audio_in` and `right_channel_audio_in` from the `Audio_Controller` feed into left and right buffer registers
- the outputs of the buffer registers feed into `left_channel_audio_out` and `right_channel_audio_out` of the `Audio_Controller`
- `audio_in_available` of the `Audio_Controller` is connected to the write enable `en` of the buffer registers as well as to `read_audio_in` of the `Audio_Controller`
- `audio_out_allowed` of the `Audio_Controller` is connected to `write_audio_out` of the `Audio_Controller`

The complete Verilog code for the top-level module is given below. Note that the parameter `USE_MIC_INPUT` of `avconf` is set to 1. If it were, set to 0, then the `Audio_Controller` would read audio data from line-in instead.

```

module Mic_to_Line_Out_Demo (
    input      CLOCK_50,
    input [0:0] KEY,
    input      AUD_ADCDAT,
    inout     AUD_BCLK,
    inout     AUD_ADCLRCK,
    inout     AUD_DACLRCK,
    inout     I2C_SDAT,
    output    AUD_XCK,
    output    AUD_DACDAT,
    output    I2C_SCLK
);

    wire          audio_in_available;
    wire [31:0]    left_channel_audio_in;
    wire [31:0]    right_channel_audio_in;
    wire          audio_out_allowed;
    wire [31:0]    left_channel_audio_out;
    wire [31:0]    right_channel_audio_out;

    reg32 audio_buffer_left (
        .clk           (CLOCK_50),

```

```

    .en                      (audio_in_available),
    .d                       (left_channel_audio_in),
    .q                       (left_channel_audio_out)
);

reg32 audio_buffer_right (
    .clk                     (CLOCK_50),
    .en                      (audio_in_available),
    .d                       (right_channel_audio_in),
    .q                       (right_channel_audio_out)
);

Audio_Controller Audio_Controller (
    // Inputs
    .CLOCK_50                (CLOCK_50),
    .reset                    (~KEY[0]),
    .read_audio_in            (audio_in_available),
    .left_channel_audio_out   (left_channel_audio_out),
    .right_channel_audio_out  (right_channel_audio_out),
    .write_audio_out          (audio_out_allowed),
    .AUD_ADCDAT              (AUD_ADCDAT),
    // Bidirectionals
    .AUD_BCLK                 (AUD_BCLK),
    .AUD_ADCLRCK              (AUD_ADCLRCK),
    .AUD_DACLRCK              (AUD_DACLRCK),
    // Outputs
    .audio_in_available       (audio_in_available),
    .left_channel_audio_in    (left_channel_audio_in),
    .right_channel_audio_in   (right_channel_audio_in),
    .audio_out_allowed        (audio_out_allowed),
    .AUD_XCK                  (AUD_XCK),
    .AUD_DACDAT               (AUD_DACDAT),
);

avconf #(.USE_MIC_INPUT(1)) avc (
    .I2C_SCLK                 (I2C_SCLK),
    .I2C_SDAT                 (I2C_SDAT),
    .CLOCK_50                  (CLOCK_50),
    .reset                     (~KEY[0])
);

endmodule

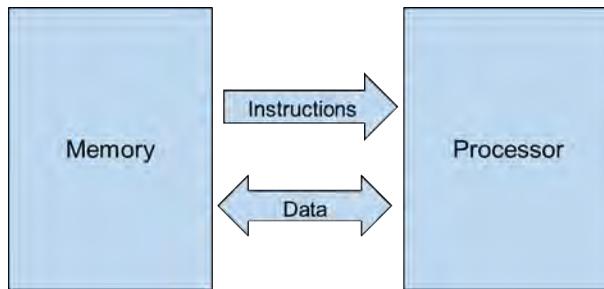
```

# 13 albaCore Microprocessor: Hardware Implementation

To this point, we have covered the basics of practical digital system design in Verilog all the way from basic AND, OR, and NOT gates up through the implementation of high-level state machines using a finite state machine controller and a custom datapath. Now, we are ready to go back to our starting point and take an in-depth look at the inner workings of a programmable digital electronic computer—specifically our albaCore microprocessor. Like our earlier Maxfinder example, an albaCore system consists of a processor connected to a memory, and the processor consists of a datapath and finite state machine controller. While Maxfinder implemented one dedicated application, however, albaCore is a programmable processor that executes programs stored as machine code in memory. In this Chapter, we will look at a complete implementation of an albaCore hardware system in Verilog, starting with the overall system of a processor with memory and showing how to connect I/O devices. From there, we will dig into the design of the datapath and controller for the processor.

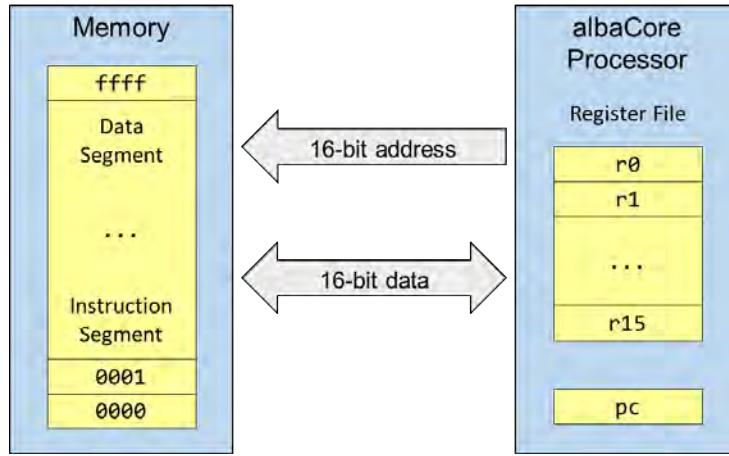
## 13.1 albaCore System and Instruction Set Review

Recall that fundamentally, a computer system has two main hardware components: *memory* that stores instructions and data, and a *processor* that reads instructions from memory and then executes these instructions to process the data.



A main design goal for albaCore was for it to be powerful enough to be able to run programs compiled from a significant subset of the C language, including integer arithmetic and logic operations and function calls, but simple enough so that the complete instruction set specification can fit on one page.

- The albaCore memory system has 16-bit addresses and 16-bit data. This means that the memory can contain up to  $2^{16}$  or 65,536 16-bit words of data. Data is treated as 16-bit, 2's complement signed integers. All machine code instructions are also encoded as 16-bit words.
- albaCore has 16 general purpose registers, each 16-bits wide. In the albaCore assembly language, we name these `r0-r15`.
- albaCore has a 16-bit program counter register, named `pc`.



albaCore has a total of only 16 carefully chosen instructions, which are sufficient for implementing many of the features of a program compiled from C. Like most RISC processors, albaCore has five categories of instructions.

- Arithmetic / Logical Instructions
  - Arithmetic (addition, subtraction, usually but not always multiplication/division) and Boolean logic operations.
  - Use registers or constants immediate values as operands
- Data Transfer Instructions (Loads and Stores)
  - Load register from data memory
  - Store register to data memory
- Branch
  - Change the PC to some new address relative to the current PC value. Branches can be conditional on some relation between registers (less than, greater than, zero, negative) or unconditional. Branch instructions are used to support if-else statements and loops in a high-level language such as C.
- Jump
  - Change PC to an absolute new location. Jump instructions are used to support function calls in a high-level language such as C.
- Special
  - Quit execution

Here's the complete albaCore instruction set:

instruction	name	operation	16-bit encoding			
			15:12	11:8	7:4	3:0
add rw, ra, rb	add	$rw \leftarrow ra + rb$	0x0	rw	ra	rb
sub rw, ra, rb	subtract	$rw \leftarrow ra - rb$	0x1	rw	ra	rb
and rw, ra, rb	bitwise and	$rw \leftarrow ra \& rb$	0x2	rw	ra	rb
or rw, ra, rb	bitwise or	$rw \leftarrow ra   rb$	0x3	rw	ra	rb

not rw, ra	bitwise not	$rw \leftarrow \sim ra$	0x4	rw	ra	rb
shl rw, ra, rb	shift left	$rw \leftarrow ra \ll rb$	0x5	rw	ra	rb
shr rw, ra, rb	shift right	$rw \leftarrow ra \gg rb$	0x6	rw	ra	rb
ldi rw, imm8	load immediate	$rw \leftarrow imm8$	0x7	rw	imm8	
ld rw, rb, imm4	load (from mem)	$rw \leftarrow M[rb+imm4]$	0x8	rw	imm4	rb
st ra, rb, imm4	store (to mem)	$M[rb+imm4] \leftarrow ra$	0x9	imm4	ra	rb
br imm8	branch (uncond)	$PC \leftarrow PC + \text{SignEx}(imm8)$	0xA	imm8		X
bz rb, imm8	branch if zero	<pre>if (rb == 0)     PC ← PC + SignEx(imm8) else     PC ← PC + 1</pre>	0xB	imm8		rb
bn rb, imm8	branch if negative	<pre>if (rb &lt; 0)     PC ← PC + SignEx(imm8) else     PC ← PC + 1</pre>	0xC	imm8		rb
jal imm12	jump and link	$PC \leftarrow \{PC[15:12], imm12\}$ $r15 \leftarrow PC + 1$	0xD	imm12		
jr ra	jump register	$PC \leftarrow ra$	0xE	X	ra	X
quit	quit	$PC \leftarrow PC$	0xF	X	X	X

## 13.2 Verilog Implementation of the Top-Level albaCore System

### 13.2.1 Complete System: Connecting Processor and Memory

At the top level, an albaCore system consists of a processor connected to a memory.

```
module system (
    input          clk,
    input          reset
);

    wire [15:0] proc_din;
    wire [15:0] addr;
    wire [15:0] proc_dout;
    wire        we;

    processor processor (
        .clk      (clk      ),
        .reset    (reset    ),
        .din      (proc_din ),
        .addr     (addr     ),
        .dout     (proc_dout),
        .we       (we       )
    );

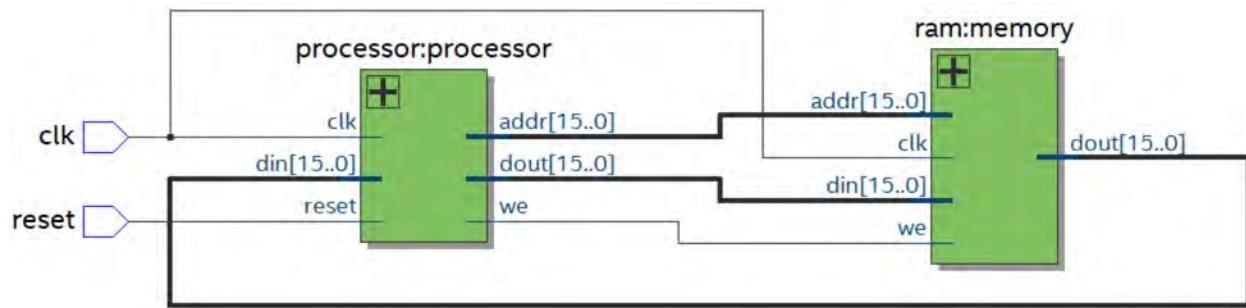
```

```

);
ram memory (
    .clk      (clk      ),
    .addr     (addr     ),
    .din      (proc_dout),
    .we       (we       ),
    .dout     (proc_din )
);
endmodule

```

The RTL netlist of the synthesized design is:



### 13.2.2 Loading a Program into Memory

In order to run a program on the albaCore processor, we need to load it into memory. We do this by initializing the memory locations in the RAM, either directly in Verilog or more conveniently, by using the `$readmemh` directive to read the memory data from a text file. The `$readmemh` approach is especially convenient since the albaCore assembler tools produce a file in this format. Both approaches are shown below. For the example, we'll use the array manipulation example presented earlier, repeated below for convenience:

```

main()
{
    A[2] = A[0] + A[1];
}

```

Variable/register usage:

variable	description	register	initial value
A	base address of A[]	r2	
t0	temp	r0	
t1	temp	r1	

Instructions:

Addr	Label	Pseudocode	Assembly	Hex
0	main:	A = 32	ldi r2, 32	7220
1		t0 = M[A+0]	ld r0, r2, 0	8002
2		t1 = M[A+1]	ld r1, r2, 1	8112
3		t0 = t0 + t1	add r0, r0, r1	0001

4		M[A+2] = t0	st r0, r2, 2	9202
5		quit	quit	F000

Data:

Addr	Data
32	3
33	4
34	0

### 13.2.2.1 Initializing Directly in Verilog

```
module ram (
    input clk,
    input [15:0] addr,
    input [15:0] din,
    input we,
    output reg [15:0] dout
);

reg [15:0] M [0:65535];

initial begin
    M[0] = 16'h7220;
    M[1] = 16'h8002;
    M[2] = 16'h8112;
    M[3] = 16'h0001;
    M[4] = 16'h9202;
    M[5] = 16'hF000;
    M[32] = 16'h3;
    M[33] = 16'h4;
    M[34] = 16'h0;
end

always @(posedge clk)
if (we)
    M[addr] <= din;

always @(posedge clk)
dout <= M[addr];

endmodule
```

### 13.2.2.2 Using \$readmemh

```
module ram (
    input clk,
```

```

input [15:0] addr,
input [15:0] din,
input we,
output reg [15:0] dout
);

reg [15:0] M [0:65535];
initial $readmemh("array.mem", M);

always @(posedge clk)
if (we)
M[addr] <= din;

always @(posedge clk)
dout <= M[addr];

endmodule

```

The file `array.mem` is given below, where the syntax for each row is `@address data`, with both values specified in hex.

```

@0 7220
@1 8002
@2 8112
@3 0001
@4 9202
@5 f000
@20 3    // 32 decimal = 20 hex
@21 4
@22 0

```

### 13.2.3 Testbench and Simulation

Like the Maxfinder, the albaCore processor has a datapath and a finite state machine controller inside, and within the controller there is a “quit” state. This state is reached when albaCore executes a quit instruction and is named `EX_QUIT` and happens to have an encoded state number of `5'd18` (as we will see later in the controller design). As we did with the Maxfinder testbench, we use a Verilog hierarchical signal and a `while` loop to detect when the processor reaches the `EX_QUIT` state in the albaCore testbench.

```

`timescale 1ns/1ns

module system_tb ();
  reg          clk;
  reg          reset;

  parameter EX_QUIT = 5'd18;

  system uut (
    .clk        (clk),

```

```

    .reset      (reset)
);

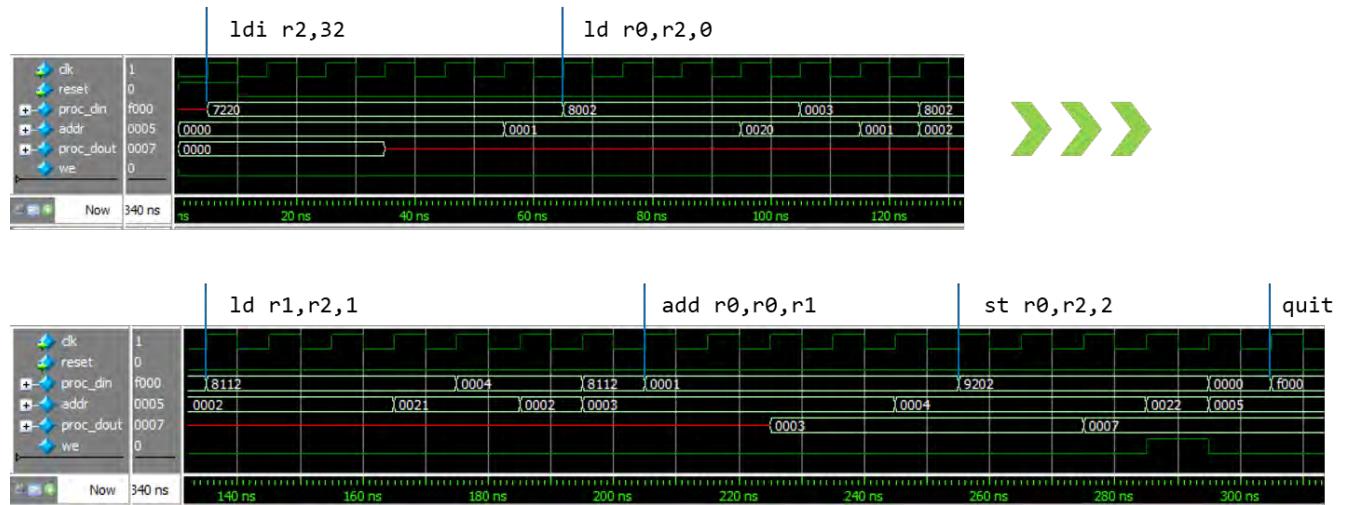
always #5 clk = ~clk;

initial begin
    clk = 0;  reset = 1;
    #10 reset = 0;
    while (uut.processor.controller.state != EX_QUIT)
        #10;
    #10 $stop;
end

endmodule

```

Simulating the testbench produces the following waveforms:



The first thing we observe is that it took 35 clock cycles (350 ns at 100 ns per cycle) for the program to reach the **quit** instruction and that each instruction took multiple clock cycles to execute. Before each instruction, the processor places the address of the current instruction on the **addr** bus, and the memory responds by placing the machine code for that instruction on the **proc\_din** bus. Then, depending upon the type of instruction, the processor goes through a series of steps that output different values onto **addr**, **proc\_dout**, and memory write enable signal **we**.

- The load immediate instruction **ldi** doesn't require any further communication with memory, since it writes the immediate value encoded into the instruction itself to the register file (not shown in the waveforms). At the end of the instruction, it places the address of the next instruction on **addr** (as do all instructions).
- The two load instructions **ld** assert the address where data is to be read from memory on **addr**, and the memory responds by placing that data on **proc\_din**.
- The **add** instruction doesn't require any further communication with memory. (It happens to put a value on **proc\_dout**, but this can be ignored as the memory doesn't do anything with it).

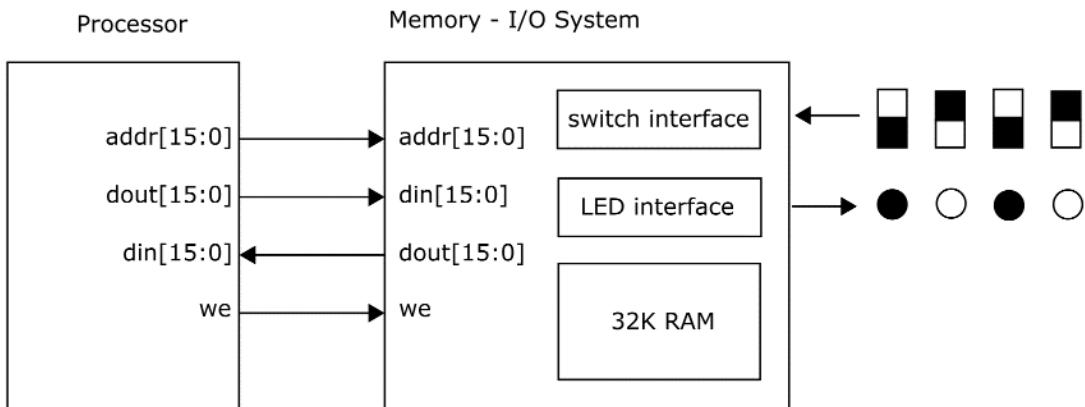
- The store instruction `st` places the address where data is to be stored in memory on `addr` and places the value of the data on `proc_dout`. It then sets the write enable `we` high for 1 clock cycle so that the data is written to memory.

Later in this chapter, we will show how to design a datapath and FSM controller for albaCore that supports these operations.

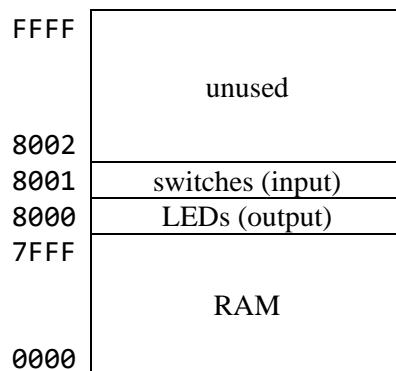
### 13.2.4 Memory-Mapped Input/Output

How do you connect and communicate between a microprocessor such as albaCore to input and output devices ranging from switches and LEDs to keyboards and monitors? The answer is to treat them like memory so that you can read data from them using load instructions and send data to them using store instructions. This approach to interfacing a microprocessor and input/output devices is called *memory-mapped I/O*.

With memory-mapped I/O, the full range of addresses—called the address space—is mapped into regions where some regions are actual memory and other regions correspond to I/O devices. To illustrate, consider an albaCore system that has 32K words of RAM, a set of switches as an input device and a set of LEDs as an output device.



These devices can be mapped to ranges of memory addresses as follows:



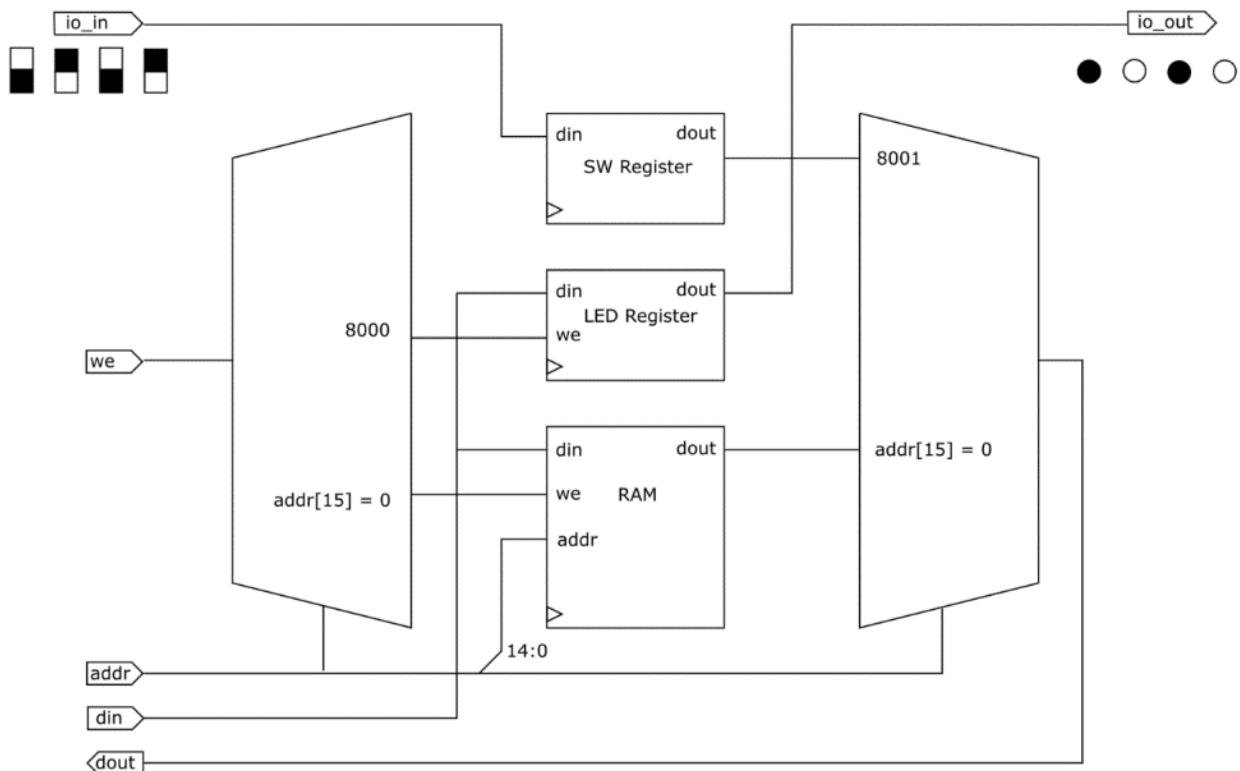
Thus loads and stores to the lower 32K addresses (0000-7FFF) are reads and writes to RAM, stores to address 8000 light up LEDs, and loads from address 8001 read values from switches.

The figure below shows the interface logic circuitry for the memory-mapped RAM and I/O devices. Registers are used to buffer the output values to the LEDs and the input values from the switches. A decoder controlled by the processor address selects which device to write to on a store instruction:

- if  $\text{addr}[15] == 0$  then the write enable for the RAM is selected
  - else if  $\text{addr} == 16'h8000$  then the write enable for the LED register is selected.

Similarly, a multiplexor selects which device to read from on a load

- if  $addr[15] == 0$  then the mux selects the RAM output
  - else if  $addr == 16'h8001$  then the mux selects the switches



A Verilog model for the memory-mapped RAM and I/O interface is given below.

```

module memory_map (
    input                      clk,
    input [15:0]                addr,
    input [15:0]                din,
    input                       we,
    output reg [15:0]           dout,
    input [3:0]                 io_in,
    output [3:0]                io_out
);

wire [15:0] dout_ram;
reg          we_ram;
reg          we_io_out;

```

```

reg [3:0] q_io_out;
reg [3:0] q_io_in;

assign io_out = q_io_out;

// I/O Output Register
always @(posedge clk)
  if (we_io_out)
    q_io_out <= din[3:0];

// I/O Input Register
always @(posedge clk)
  q_io_in <= io_in;

ram32k ram (
  .clk      (clk),
  .addr     (addr[14:0]),
  .din      (din),
  .we       (we_ram),
  .dout     (dout_ram)
);

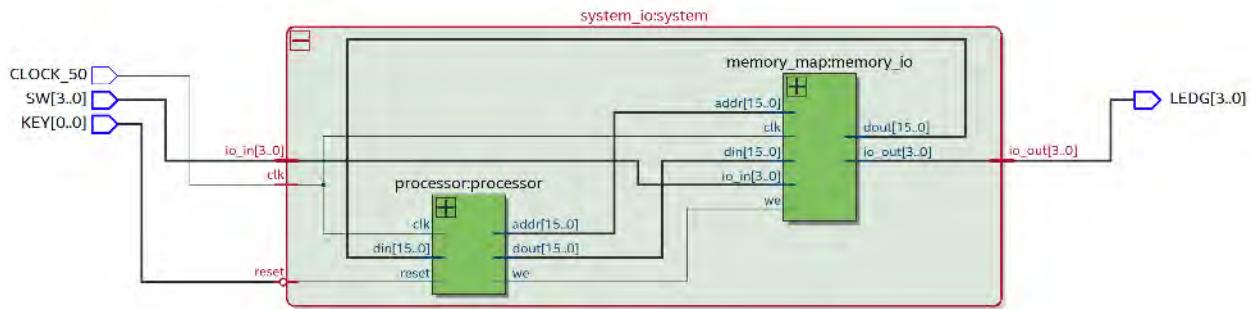
// Write Enable Decoder
always @(*) begin
  we_ram      = 0;
  we_io_out   = 0;
  if (addr[15] == 0)
    we_ram = we;
  else
    if (addr == 16'h8000)
      we_io_out = we;
end

// Output Multiplexor
always @(*) begin
  if (addr[15] == 0)
    dout = dout_ram;
  else if (addr == 16'h8001)
    dout = {12'h0, q_io_in};
  else
    dout = 16'h0;
end

endmodule

```

A schematic of the fully-integrated system is shown below:



To test the system, we use a program that repeatedly reads the switches and then outputs the value read to the LEDs:

Variable	Register
A	r0
s	r1
data	r8

Address	Label	Pseudocode	Assembly	Hex
		s = 8	ldi r1, 8	7108
	loop:	A = 0x80	ldi r0, 0x80	7080
		A = A << 8	shl r0, r0, r1	5001
		data = M[A+1] // read switches	ld r8, r0, 1	8810
		M[A] = data // write LEDs	st r8, r0, 0	9080
		goto loop	br -4	AFC0

### 13.3 5-Stage RISC Instruction Processing Cycle

When designing programmable processors, computer architects typically break the instruction processing cycle down into stages. One of the most common schemes used in simple processor designs is a 5-stage processing cycle. We will use this scheme for albaCore, with each stage defined below:

1. **Instruction Fetch:** Read an instruction from memory
2. **Instruction Decode/Register Fetch:** Break the instruction down into fields. Use the values in the `ra` and `rb` fields to read source operands from the register file, whether the instruction actually needs to use them or not. Use the `opcode` field to determine what to do next for each instruction type.
3. **Execute:** Perform arithmetic, logic, or comparison operations.
4. **Memory:** Access memory for load and store instructions
5. **Write Back:** Write results back into the register file at the location specified by the `rw` field if required by the instruction

The following table spells these steps out in more detail for the arithmetic/logic, data transfer, and branch instruction types.

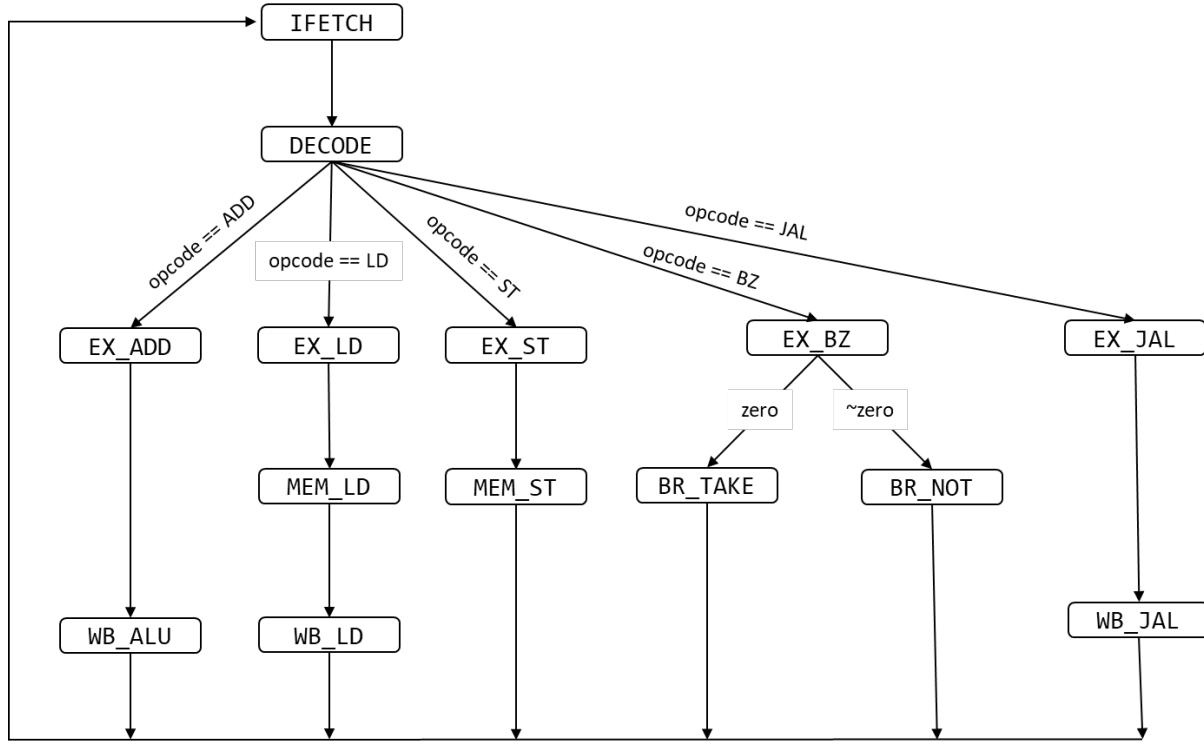
	arithmetic/logic	load	store	branch
<b>Instruction Fetch</b>	Read instruction at <code>pc</code> from memory			
<b>Decode/ Register Fetch</b>	Break instruction down into fields Read operands from register file at <code>ra</code> and <code>rb</code> Use <code>opcode</code> to decide what type of instruction this is and what to do next			
<b>Execute</b>	Perform arithmetic/logic operation on values read from register file	Calculate memory address as <code>rb</code> + offset	Calculate memory address as <code>rb</code> + offset	Make comparisons on values read from register file  Update <code>pc</code> based on whether branch is taken or not
<b>Memory</b>		Read data from memory	Write data to memory  Increment <code>pc</code>	
<b>Write Back</b>	Write results from execute step back to register file at <code>rw</code>  Increment <code>pc</code>	Write value read from memory back to register file at <code>rw</code>  Increment <code>pc</code>		

Note that not all instruction types use all steps. The last step for a given instruction is responsible for updating the program counter (`pc`), either to the next sequential address, or to some displacement off the `pc` for a taken branch.

### 13.4 HLSM

Given this conceptual framework of a 5-stage instruction processing cycle, we can develop an HLSM for the albaCore processor. The figure below gives a high level view of the flow. After fetching the instruction, during the DECODE state, the machine conditionally transitions to one of a set of execution

states—one for each instruction type—depending upon the value of the opcode found in the 4 most-significant bits of the instruction. From that point, the flow through the remaining stages is specific to the type of instruction. For the branch instructions, the execution state checks the flag value and then transitions to one of two states depending upon whether or not the branch is taken or not. For a jump-and-link, the execution state calculates the return address, while the write-back state both writes the return address to  $r15$  and updates the program counter to the jump target address.



Before constructing a complete HLSM table, we need to define the variables that will ultimately become registers in the datapath. These are listed in the table below:

$R[rn]$	register $rn$ in register file
$pc$	program counter
$din$	data read from memory (processor input port)
$inst$	instruction register
$a$	value read from register $ra$
$b$	value read from register $rb$
$f$	result of an arithmetic/logic operations
$mdr$	memory data register

Because we'll be using a synchronous memory (to take advantage of the embedded memory modules in the FPGA), each stage that reads memory—fetching an instruction and loading data—will take 2 cycles/states: one to read the memory itself and another to store the result in a register in the processor.

Given the flow diagram and variable definitions, the HLSM table for albaCore is presented below:

state	actions	transitions
IFETCH	$din \leftarrow M[pc]$	goto IFETCH2

IFETCH2	$inst \leftarrow din$	goto DECODE
DECODE	$a \leftarrow R[ra]$ $b \leftarrow R[rb]$	case (op) op: goto EX_OP
EX_ADD	$f \leftarrow a + b$	goto WB_ALU
EX_SUB	$f \leftarrow a - b$	goto WB_ALU
EX_AND	$f \leftarrow a \& b$	goto WB_ALU
EX_OR	$f \leftarrow a   b$	goto WB_ALU
EX_NOT	$f \leftarrow \sim a$	goto WB_ALU
EX_SHL	$f \leftarrow a \ll b$	goto WB_ALU
EX SHR	$f \leftarrow a \gg b$	goto WB_ALU
EX_LDI	$f \leftarrow imm$	goto WB_ALU
EX_LD	$f \leftarrow b + offset\_ld$	goto MEM_LD
EX_ST	$f \leftarrow b + offset\_st$	goto MEM_ST
EX_BR	$pc \leftarrow pc + disp$	goto IFETCH
EX_BZ		if (b == 0) goto BR_TAKE else goto BR_NOT
EX_BN		if (b < 0) goto BR_TAKE else goto BR_NOT
EX_JAL	$f \leftarrow pc + 1$	goto WB_JAL
EX_JR	$pc \leftarrow a$	goto IFETCH
EX_QUIT		goto QUIT
MEM_LD	$din \leftarrow M[f]$	goto MEM_LD2
MEM_LD2	$mdr \leftarrow din$	goto WB_LD
MEM_ST	$M[f] \leftarrow a$ $pc \leftarrow pc + 1$	goto IFETCH
WB_ALU	$R[rw] \leftarrow f$ $pc \leftarrow pc + 1$	goto IFETCH
WB_LD	$R[rw] \leftarrow mdr$ $pc \leftarrow pc + 1$	goto IFETCH
WB_JAL	$R[15] \leftarrow f$ $pc \leftarrow \{pc[15:12], target\}$	
BR_TAKE	$pc \leftarrow pc + disp$	goto IFETCH
BR_NOT	$pc \leftarrow pc + 1$	goto IFETCH

## 13.5 Datapath Design

### 13.5.1 Identifying Sources for Variables and Defining Flags

Given the HLSM table, we move on to designing a datapath that supports each of the register transfer operations. As we've done with the Maxfinder and other examples, we start by enumerating the sources that produce values for each of the variables:

destination	sources	signals
inst	$M[pc]$	en_inst
a	$R[ra]$	en_a
b	$R[rb]$	en_b
f	$a + b$ $a - b$ $a \& b$ $a   b$ $\sim a$ $a \ll b$	en_f alu_op

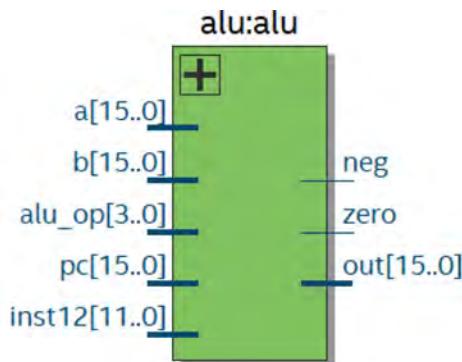
	$a >> b$ imm $b + \text{offset\_ld}$ $b + \text{offset\_st}$ $pc + 1$	
mdr	$M[f]$	en_mdr
$M[f]$	a	s_addr mem_we
$R[rw]$	0: f 1: mdr	we_regfile s_rw s_regfile_din
$R[15]$	0: f mdr	we_regfile s_rw we_regfile
pc	$pc + 1$ $pc + \text{disp}$ $\{pc[15:12], \text{target}\}$ $R[ra]$	en_pc alu_op

Next, we define flags associated with each of the conditional transitions:

condition	flag
$b == 0$	zero
$b < 0$	neg

### 13.5.2 Arithmetic Logic Unit (ALU)

In our earlier examples of processor design such as the Maxfinder, we went straight from the RTL source/destination table and flag definitions to Verilog models for the datapath. For albaCore, we'll take a different approach and construct the datapath around a single Arithmetic/Logic Unit (ALU) that performs all of the arithmetic and logical operations required, including operations for the arithmetic/logic instructions, calculating the memory address for loads and stores, making comparisons for conditional branches, and calculating the new program counter value for branches and jumps. The symbol for the ALU is given below:



The ALU has data inputs from the pc, a, and b registers, as well as the fields from the 12 least-significant bits of the instruction, `inst12` that contains the load/store offsets, branch displacement, shift amount, and jump target. It has a 4-bit control input `op` that selects the operation. As outputs, it has the 16-bit result `out` and the two flags, `neg` and `zero`.

The following table lists the ALU functions for each value of `op[3:0]`:

alu_op[3:0]	operation	usage
0	out = a + b	add
1	out = a - b	sub
2	out = a & b	and
3	out = a   b	or
4	out = ~a	not
5	out = a << b	shl
6	out = a >> b	shr
7	out = {8'h00, imm}	ldi
8	out = b + {8'd0, offset_ld}	address calculation for ld
9	out = b + {8'd0, offset_st}	address calculation for st
10	out = pc + 1	incrementing pc
11	out = pc + SignExtend(disp)	update pc for taken branches
12	out = {pc[15:12], target}	update pc for jal
13	out = a	update pc for jr

The Verilog model for the ALU is given below:

```

module alu (
    input      [15:0] a,
    input      [15:0] b,
    input      [3:0]  alu_op,
    input      [11:0] inst12,
    input      [15:0] pc,
    output reg [15:0] out,
    output      neg,
    output      zero
);

wire [7:0]  imm      = inst12[7:0];
wire [7:0]  disp     = inst12[11:4];
wire [3:0]  offset_ld = inst12[7:4];
wire [3:0]  offset_st = inst12[11:8];

assign neg          = b[15];
assign zero         = b == 0;

always @(*)
    case (alu_op)
        0:  out = a + b;
        1:  out = a - b;
        2:  out = a & b;
        3:  out = a | b;
        4:  out = ~a;
        5:  out = a << b;
        6:  out = a >> b;
        7:  out = {8'h00, imm};

```

```

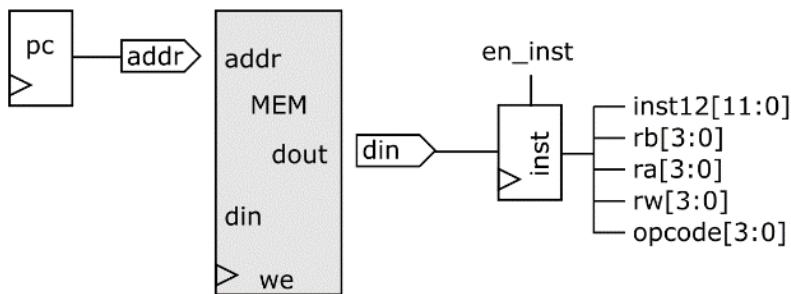
8:  out = b + {8'd0, offset_ld};
9:  out = b + {8'd0, offset_st};
10: out = pc + 1;
11: out = pc + ( disp[7] ? {8'hff, disp} : {8'h00, disp} );
12: out = {pc[15:12], inst12};
13: out = a;
default: out = 0;
endcase

endmodule

```

### 13.5.3 Instruction Fetch and Decode

The first two stages in the processing cycle, instruction fetch and decode, involve reading an instruction from memory at the address specified by the program counter `pc`, and then breaking the instruction down into fields. The structure required for this is shown in the figure below. The field `inst12` is the 12 least-significant bits of the instruction, which contains the branch displacement, memory offsets, and jump target, depending on the instruction. Note that the memory is an external component connected to the processor and is not actually part of the datapath, but it is shown to illustrate the port connections. Here, `pc` is connected to the processor address port `addr` and data out from memory is connected to the processor data in port, `din`.



The Verilog snippet below shows the breakdown of the the 16-bit instruction `inst[15:0]` into fields. Note that some of the fields are overlapping, such as `inst12` and `ra` or `rb`, since these bit ranges have different meanings for different instructions.

```

wire [15:0] inst;

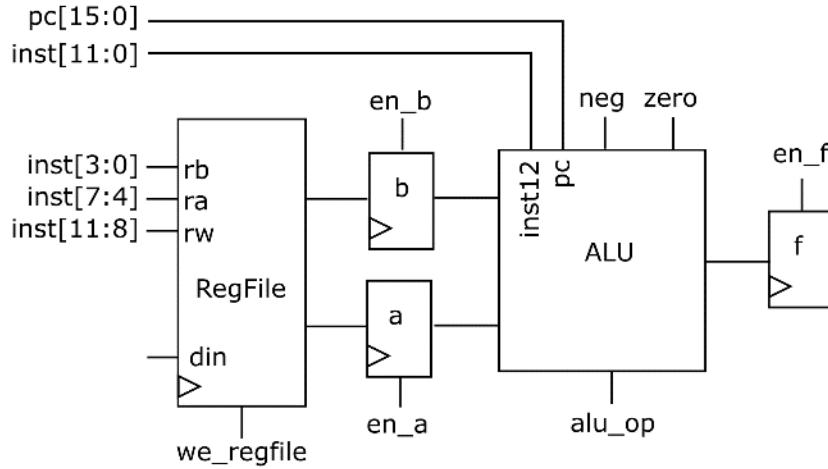
assign      opcode      = inst[15:12];
wire [3:0]  rw          = inst[11:8];
wire [3:0]  ra          = inst[7:4];
wire [3:0]  rb          = inst[3:0];
wire [11:0] inst12     = inst[11:0];

```

### 13.5.4 Register Fetch and Instruction Execution

The figure below illustrates the datapath structure required for register fetch and instruction execution stages. During register fetch (which happens during the same cycle as instruction decode), data from the register file at locations `ra` and `rb` are stored in datapath registers `a` and `b`. During the execution stage, operations among registers `a` and `b` and fields from the instruction are processed by the ALU and stored in

the **f** register. For a jump-and-link instruction, the execution stage uses the ALU to calculate the return address  $pc + 1$ .

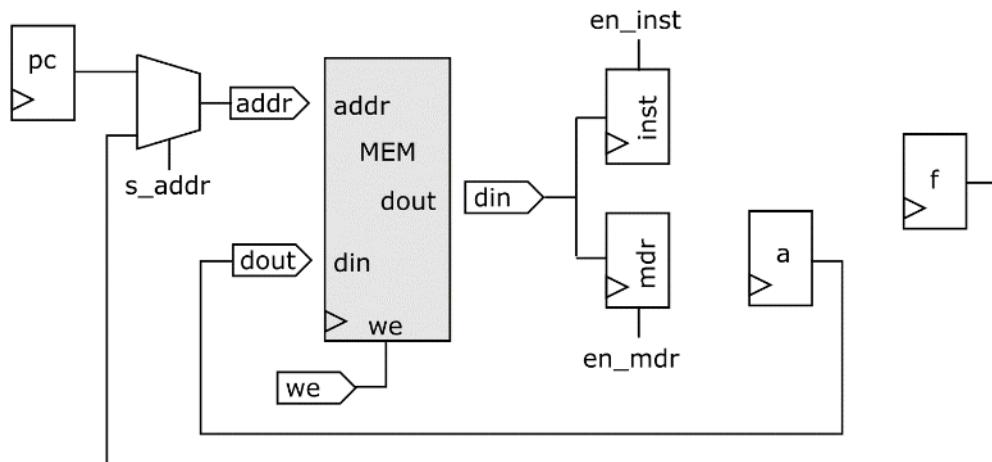


### 13.5.5 Load/Store Instruction Memory Stage

Because both data and instructions are stored in memory, connections to the memory address (`addr`), data out (`dout`), and data in (`din`) ports need to support the memory stage of the load and store instructions, as well as the instruction fetch stage.

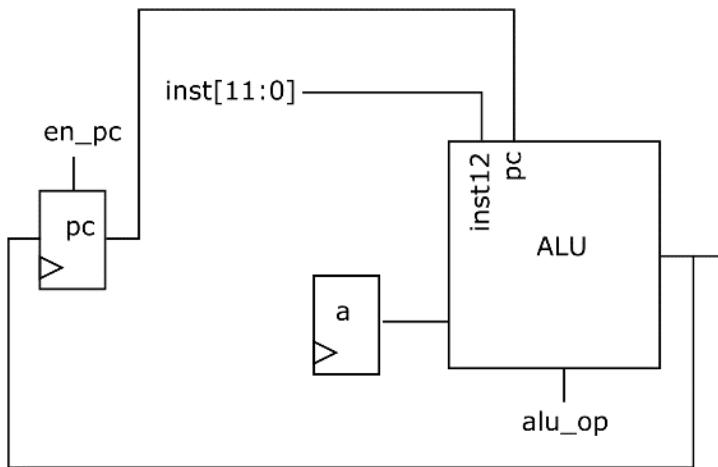
- During the execution stage of the load (ld) instruction, data is read from memory at the address specified by the f register and stored in the memory data register, mdr.
  - During the execution stage of the store instruction (st), data from the a register is written to memory at the address specified by the f register.

A multiplexor on the **addr** port selects between the **pc** for instruction fetch and the **f** register for loads. The memory output port connects to both the instruction register **inst** and **mdr** via the processor **din** port.



### 13.5.6 PC Update and Branch/Jump Instruction Execution

For a taken branch instruction, an 8-bit 2's complement displacement taken from least-significant bits of the instruction is added to the program counter. For the jump register (*jr*) instruction, the PC gets the value from *ra*. For all other cases, the pc is incremented by 1. The ALU calculates the next PC value for each of these cases, depending upon the *alu\_op*.



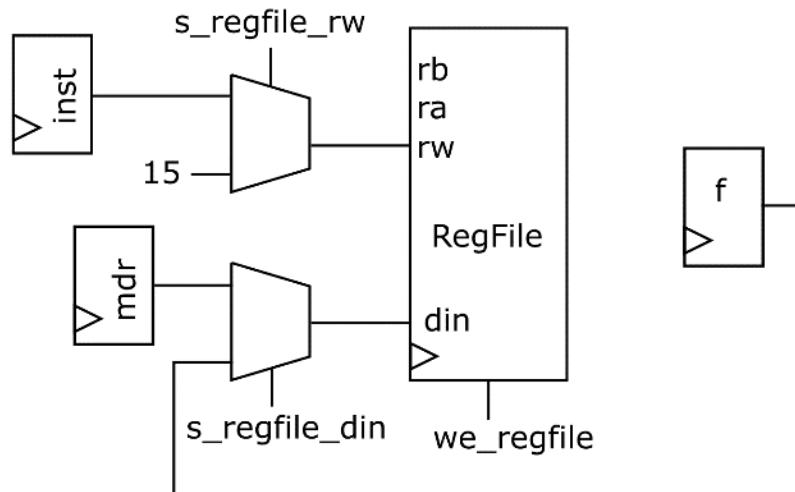
Because the displacement can be positive or negative, it needs to be sign-extended to 16-bits before adding it to the *pc* to preserve the sign. This is done by padding the displacement with leading 1s or 0s, depending on the sign of the most significant bit, within the ALU.

### 13.5.7 Write Back

Three types of instructions write values back to the register file:

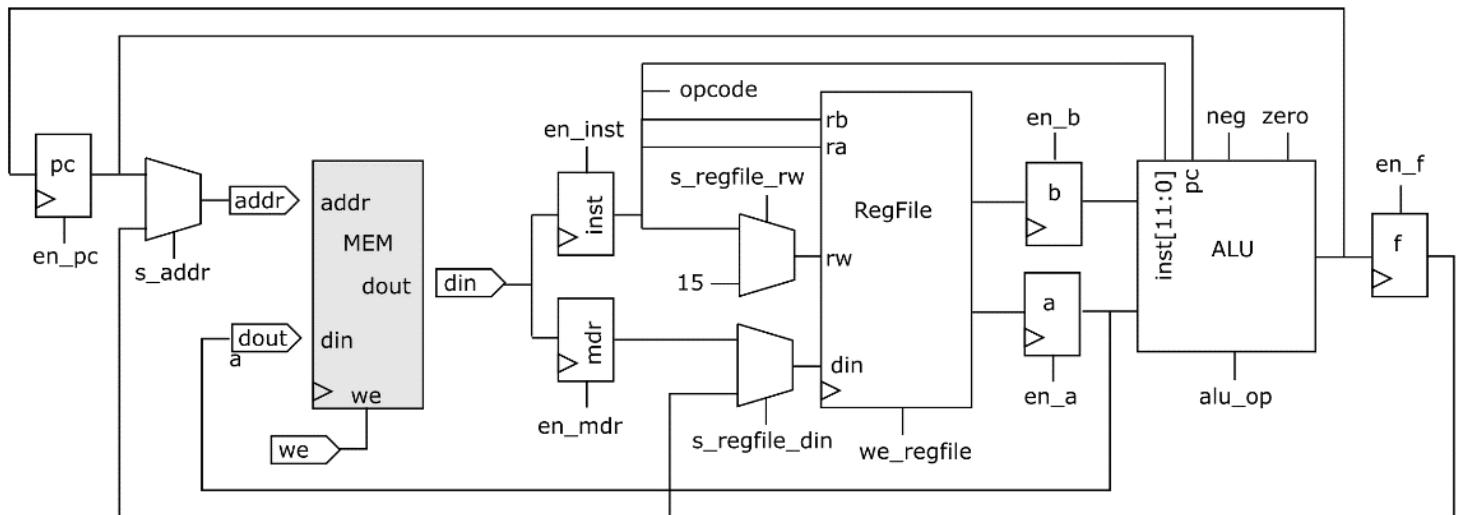
- arithmetic/logic instructions write the value in register *f* produced by the ALU to *rw* specified by bits 11:8 of the instruction (*inst[11:8]*)
- the load instruction (*ld*) write the value read from memory in the *mdr* to *rw* specified by bits 11:8 of the instruction (*inst[11:8]*)
- the jump-and-link instruction (*jal*) writes the value in register *f* to register *r15*

Since both the write back data and register number can come from different sources, this requires multiplexors on both the *din* and *rw* ports of the register file. The circuitry for this is shown below:



### 13.5.8 Complete Datapath

The figure below shows the complete albaCore datapath design that stiches together the supporting structure for each of the processing stages.



### 13.5.9 Verilog Implementation of Datapath

The Verilog models for the remaining datapath modules are given below.

#### 13.5.9.1 Register File

```

module regfile (
    input          clk,
    input [15:0]   din,
    input [3:0]    waddr,
    input [3:0]    raddra,
    input [3:0]    raddrb,
    input          we,
    output [15:0]  douta,
    output [15:0]  doutb
);

reg [15:0] M [0:15];

always @(posedge clk)
if (we)
    M[waddr] <= din;

assign douta = M[raddra];
assign doutb = M[raddrb];

endmodule

```

### 13.5.9.2 16-Bit Register

```
module reg16(
    input                  clk,
    input                  en,
    input [15:0]      d,
    output reg [15:0] q
);

initial q = 0;

always @(posedge clk)
    if (en)
        q <= d;

endmodule
```

### 13.5.9.3 Fully Connected Datapath Structure

```
module datapath (
    input                  clk,
    input                  s_addr,
    input                  en_inst,
    input                  en_a,
    input                  en_b,
    input [3:0]      alu_op,
    input                  en_f,
    input                  en_mdr,
    input                  s_regfile_din,
    input                  we_regfile,
    input                  s_regfile_rw,
    input                  en_pc,
    output [3:0]     opcode,
    output                  zero,
    output                  neg,
    input [15:0]     din,
    output [15:0]     addr,
    output [15:0]     dout
);

wire [15:0] inst;
wire [11:0] inst12          = inst[11:0];
assign      opcode           = inst[15:12];
wire [3:0]   rw              = inst[11:8];
wire [3:0]   ra              = inst[7:4];
wire [3:0]   rb              = inst[3:0];
wire [3:0]   regfile_rw;
wire [15:0] regfile_din;
wire [15:0] regfile_a;
wire [15:0] a;
wire [15:0] regfile_b;
```

```

wire [15:0] b;
wire [15:0] next_pc;
wire [15:0] pc;
wire [15:0] alu_out;
wire [15:0] f;
wire [15:0] mdr;

assign dout      = a;
assign regfile_rw = s_regfile_rw ? 4'd15 : rw;
assign regfile_din = s_regfile_din ? mdr : f;
assign addr      = s_addr ? f : pc;

regfile regfile (
    .clk      (clk),
    .din      (regfile_din),
    .waddr    (regfile_rw),
    .raddra   (ra),
    .raddrb   (rb),
    .we       (we_regfile),
    .douta   (regfile_a),
    .doutb   (regfile_b)
);

reg16 pc_reg (
    .clk      (clk),
    .en       (en_pc),
    .d        (alu_out),
    .q        (pc)
);

reg16 inst_reg (
    .clk      (clk),
    .en       (en_inst),
    .d        (din),
    .q        (inst)
);

reg16 a_reg (
    .clk      (clk),
    .en       (en_a),
    .d        (regfile_a),
    .q        (a)
);

reg16 b_reg (
    .clk      (clk),
    .en       (en_b),
    .d        (regfile_b),
    .q        (b)
);

```

```

reg16 f_reg (
    .clk      (clk),
    .en       (en_f),
    .d        (alu_out),
    .q        (f)
);

reg16 mdr_reg (
    .clk      (clk),
    .en       (en_mdr),
    .d        (din),
    .q        (mdr)
);

alu alu (
    .a        (a),
    .b        (b),
    .alu_op   (alu_op),
    .inst12  (inst12),
    .pc       (pc),
    .out      (alu_out),
    .neg      (neg),
    .zero     (zero)
);

endmodule

```

## 13.6 Controller Design

### 13.6.1 HLSM with Control Inputs and Flags

Given the full elaboration of the datapath with control signals and flags, we can now annotate the HLSM table with values for the control inputs and flags:

state	actions	transitions
IFETCH	$din \leftarrow \text{readMem}(pc)$ $s\_addr = 0;$	goto IFETCH2
IFETCH2	$inst \leftarrow din$ $en\_inst = 1;$	goto DECODE
DECODE	$a \leftarrow R[ra]$ $b \leftarrow R[rb]$ $en\_a = 1;$ $en\_b = 1;$	case (op) op: goto EX_OP
EX_ADD	$f \leftarrow a + b$ $alu\_op = 0; en\_f = 1;$	goto WB_ALU

EX_SUB	$f \leftarrow a - b$  alu_op = 1; en_f = 1;	goto WB_ALU
EX_AND	$f \leftarrow a \& b$  alu_op = 2; en_f = 1;	goto WB_ALU
EX_OR	$f \leftarrow a   b$  alu_op = 3; en_f = 1;	goto WB_ALU
EX_NOT	$f \leftarrow \sim a$  alu_op = 4; en_f = 1;	goto WB_ALU
EX_SHL	$f \leftarrow a \ll b$  alu_op = 5; en_f = 1;	goto WB_ALU
EX SHR	$f \leftarrow a \gg b$  alu_op = 6; en_f = 1;	goto WB_ALU
EX_LDI	$f \leftarrow \text{imm}$  alu_op = 7; en_f = 1;	goto WB_ALU
EX_LD	$f \leftarrow b + \text{offset\_ld}$  alu_op = 8; en_f = 1;	goto MEM_LD
EX_ST	$f \leftarrow b + \text{offset\_st}$  alu_op = 9; en_f = 1;	goto MEM_ST
EX_BR	$pc \leftarrow pc + \text{disp}$  alu_op = 11; en_pc = 1;	goto IFETCH
EX_BZ		if (zero) goto BR_TAKE else goto BR_NOT
EX_BN		if (neg) goto BR_TAKE else goto BR_NOT
EX_JAL	$f \leftarrow pc + 1$  alu_op = 10; en_f = 1;	goto WB_JAL
EX_JR	$pc \leftarrow a$  alu_op = 13; en_pc = 1;	goto IFETCH
EX_QUIT		goto EX_QUIT
MEM_LD	$din \leftarrow M[f]$  s_addr = 1;	goto MEM_LD2
MEM_LD2	$mdr \leftarrow din$  en_mdr = 1;	goto WB_LD
MEM_ST	$M[f] \leftarrow a$ $pc \leftarrow pc + 1$  s_addr = 1; mem_wb = 1; alu_op = 10; pc_en = 1;	goto IFETCH
WB_ALU	$R[rw] \leftarrow f$ $pc \leftarrow pc + 1$	goto IFETCH

	s_regfile_rw = 0; s_regfile_din = 0; we_regfile = 1; alu_op = 10; en_pc = 1;	
WB_LD	R[rw] ← mdr pc ← pc + 1  s_regfile_rw = 0; s_regfile_din = 1; we_regfile = 1; alu_op = 10; en_pc = 1;	goto IFETCH
WB_JAL	R[15] ← f pc ← {pc[15:12], target}  s_regfile_rw = 1; s_regfile_din = 0; we_regfile = 1; alu_op = 12; en_pc = 1;	goto IFETCH
BR_TAKE	pc ← pc + disp  alu_op = 11; en_pc = 1;	goto IFETCH
BR_NOT	pc ← pc + 1  alu_op = 10; en_pc = 1;	goto IFETCH

### 13.6.2 Verilog Implementation

```
module controller (
    input          clk,
    input          reset,
    output reg    s_addr,
    output reg    en_inst,
    output reg    en_a,
    output reg    en_b,
    output reg [3:0] alu_op,
    output reg    en_f,
    output reg    en_mdr,
    output reg    s_regfile_din,
    output reg    we_regfile,
    output reg    s_regfile_rw,
    output reg    en_pc,
    input   [3:0]  opcode,
    input          zero,
    input          neg,
    output reg    we_mem
);

parameter IFETCH  = 5'd0;
parameter IFETCH2 = 5'd1;
parameter DECODE  = 5'd2;
parameter EX_ADD  = 5'd3;
parameter EX_SUB  = 5'd4;
parameter EX_AND  = 5'd5;
parameter EX_OR   = 5'd6;
parameter EX_NOT  = 5'd7;
```

```

parameter EX_SHL  = 5'd8;
parameter EX SHR  = 5'd9;
parameter EX_LDI  = 5'd10;
parameter EX_LD   = 5'd11;
parameter EX_ST   = 5'd12;
parameter EX_BR   = 5'd13;
parameter EX_BZ   = 5'd14;
parameter EX_BN   = 5'd15;
parameter EX_JAL  = 5'd16;
parameter EX_JR   = 5'd17;
parameter EX_QUIT = 5'd18;
parameter MEM_LD  = 5'd19;
parameter MEM_LD2 = 5'd20;
parameter MEM_ST  = 5'd21;
parameter WB_ALU  = 5'd22;
parameter WB_LD   = 5'd23;
parameter WB_JAL  = 5'd24;
parameter BR_TAKE = 5'd25;
parameter BR_NOT  = 5'd26;

reg [4:0] state, next_state;

always @(posedge clk)
  if (reset)
    state <= IFETCH;
  else
    state <= next_state;

always @(*) begin
  s_addr      = 0;
  en_inst     = 0;
  en_a        = 0;
  en_b        = 0;
  alu_op      = 0;
  en_f         = 0;
  en_mdr      = 0;
  we_mem      = 0;
  s_regfile_din = 0;
  we_regfile  = 0;
  s_regfile_rw = 0;
  en_pc       = 0;
  next_state   = EX_QUIT;
  case (state)
    IFETCH : begin
      s_addr = 0;
      next_state = IFETCH2;
    end
    IFETCH2 : begin
      en_inst = 1;
      next_state = DECODE;
    end
  endcase
end

```

```

    end
    DECODE : begin
        en_a = 1; en_b = 1;
        case (opcode)
            0: next_state = EX_ADD;
            1: next_state = EX_SUB;
            2: next_state = EX_AND;
            3: next_state = EX_OR;
            4: next_state = EX_NOT;
            5: next_state = EX_SHL;
            6: next_state = EX SHR;
            7: next_state = EX_LDI;
            8: next_state = EX_LD;
            9: next_state = EX_ST;
            10: next_state = EX_BR;
            11: next_state = EX_BZ;
            12: next_state = EX_BN;
            13: next_state = EX_JAL;
            14: next_state = EX_JR;
            default: next_state = EX_QUIT;
        endcase
    end
    EX_ADD : begin
        alu_op = 0; en_f = 1;
        next_state = WB_ALU;
    end
    EX_SUB : begin
        alu_op = 1; en_f = 1;
        next_state = WB_ALU;
    end
    EX_AND : begin
        alu_op = 2; en_f = 1;
        next_state = WB_ALU;
    end
    EX_OR : begin
        alu_op = 3; en_f = 1;
        next_state = WB_ALU;
    end
    EX_NOT : begin
        alu_op = 4; en_f = 1;
        next_state = WB_ALU;
    end
    EX_SHL : begin
        alu_op = 5; en_f = 1;
        next_state = WB_ALU;
    end
    EX_SHR : begin
        alu_op = 6; en_f = 1;
        next_state = WB_ALU;
    end

```

```

EX_LDI  : begin
    alu_op = 7;  en_f = 1;
    next_state = WB_ALU;
end
EX_LD   : begin
    alu_op = 8;  en_f = 1;
    next_state = MEM_LD;
end
EX_ST   : begin
    alu_op = 9;  en_f = 1;
    next_state = MEM_ST;
end
EX_BR   : begin
    alu_op = 11; en_pc = 1;
    next_state = IFETCH;
end
EX_BZ   : begin
    if (zero) next_state = BR_TAKE;
    else next_state = BR_NOT;
end
EX_BN   : begin
    if (neg) next_state = BR_TAKE;
    else next_state = BR_NOT;
end
EX_JAL  : begin
    alu_op = 10; en_f = 1;
    next_state = WB_JAL;
end
EX_JR   : begin
    alu_op = 13; en_pc = 1;
    next_state = IFETCH;
end
EX_QUIT : begin
    next_state = EX_QUIT;
end
MEM_LD  : begin
    s_addr = 1;
    next_state = MEM_LD2;
end
MEM_LD2 : begin
    en_mdr = 1;
    next_state = WB_LD;
end
MEM_ST  : begin
    we_mem = 1;  s_addr = 1;
    alu_op = 10; en_pc = 1;
    next_state = IFETCH;
end
WB_ALU  : begin
    s_Regfile_rw = 0;  s_Regfile_din = 0;  we_Regfile = 1;

```

```

        alu_op = 10; en_pc = 1;
        next_state = IFETCH;
    end
    WB_LD : begin
        s_regfile_rw = 0; s_regfile_din = 1; we_regfile = 1;
        alu_op = 10; en_pc = 1;
        next_state = IFETCH;
    end
    WB_JAL : begin
        s_regfile_rw = 1; s_regfile_din = 0; we_regfile = 1;
        alu_op = 12; en_pc = 1;
        next_state = IFETCH;
    end
    BR_TAKE : begin
        alu_op = 11; en_pc = 1;
        next_state = IFETCH;
    end
    BR_NOT : begin
        alu_op = 10; en_pc = 1;
        next_state = IFETCH;
    end
    default : begin
    end
endcase
end

endmodule

```

### 13.7 Complete Processor: Connecting Controller and Datapath

```

module processor (
    input          clk,
    input          reset,
    input [15:0]   din,
    output [15:0]  addr,
    output [15:0]  dout,
    output         we
);

    wire          s_addr;
    wire          en_inst;
    wire          en_a;
    wire          en_b;
    wire [2:0]    alu_op;
    wire          en_f;
    wire          en_mdr;
    wire          s_regfile_din;
    wire          we_regfile;
    wire          s_next_pc;

```

```

wire      en_pc;
wire [3:0]  opcode;
wire      zero;
wire      neg;

controller controller (
    .clk      (clk      ),
    .reset    (reset    ),
    .s_addr   (s_addr   ),
    .en_inst  (en_inst  ),
    .en_a     (en_a     ),
    .en_b     (en_b     ),
    .alu_op   (alu_op   ),
    .en_f     (en_f     ),
    .en_mdr   (en_mdr   ),
    .s_regfile_din (s_regfile_din),
    .we_regfile (we_regfile ),
    .s_next_pc (s_next_pc ),
    .en_pc    (en_pc    ),
    .opcode   (opcode   ),
    .zero     (zero     ),
    .neg      (neg      ),
    .we_mem   (we      )
);

datapath datapath (
    .clk      (clk      ),
    .s_addr   (s_addr   ),
    .en_inst  (en_inst  ),
    .en_a     (en_a     ),
    .en_b     (en_b     ),
    .alu_op   (alu_op   ),
    .en_f     (en_f     ),
    .en_mdr   (en_mdr   ),
    .s_regfile_din (s_regfile_din),
    .we_regfile (we_regfile ),
    .s_next_pc (s_next_pc ),
    .en_pc    (en_pc    ),
    .opcode   (opcode   ),
    .zero     (zero     ),
    .neg      (neg      ),
    .din      (din      ),
    .addr     (addr     ),
    .dout    (dout     )
);

endmodule

```

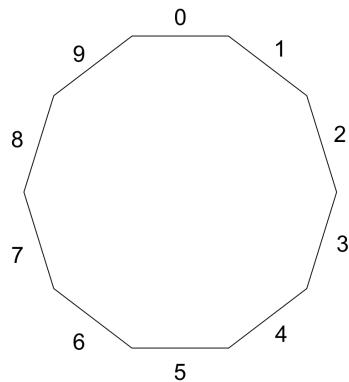
## 14 Appendix: Method of Complements

**The challenge:** Can we use a symbol system and machine designed for addition of non-negative integers to represent negative numbers and implement subtraction?

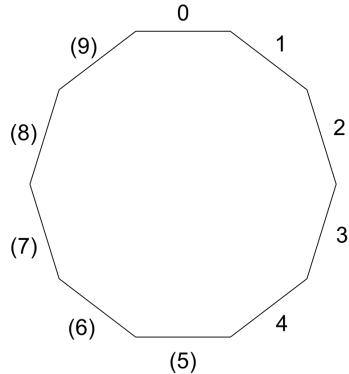
**The answer:** Yes, by using the method of complements.

### 14.1 10's Complement Representation for Single-Digit Numbers

The method of complements has been around for quite some time, and was a common technique for implementing subtraction on mechanical adding machines. The basic idea is that given a range of non-negative integers 0 to N, instead of having all of them represent their common non-negative values, let half of them represent negative numbers. To illustrate, consider a single wheel from a base 10 adding machine with digits 0-9:

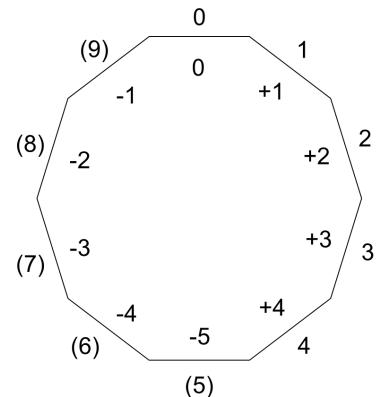


Instead of having all of the digits represent non-negative values, we'll let half of them in parentheses represent negative values, such that the faces labeled with digits 0-4 represent non-negative values, and the faces labeled with digits 5-9 represent negative values:



In order to preserve the convention that values increase in the clockwise direction (at least until we get to the transition from positive to negative between 4 and 5), we assign the value -1 to the face immediately to the left of 0, -2 to the face to the left of that, and so on. This assignment is known as 10's complement representation, because the magnitude of the negative values equals 10 minus the face value. More generally, the 10's complement of a value n in the range 1-9 is equal to  $10 - n$ .

value	1	2	3	4	5	6	7	8	9
10's complement	9	8	7	6	5	4	3	2	1



The rules for determining the sign and magnitude of a number using 10's complement representation are as follows:

- If a face is labeled with a number  $n$  in the range 0-4, then the sign is (+) and the magnitude is the value of the label  $n$ .
- If a face is labeled with a number  $n$  in the range 5-9, then the sign is (-) and the magnitude is the 10's complement of the value of the label ( $10-n$ ).

### 14.1.1 Implementing Subtraction with Addition

Given this assignment, it's easy to see how adding  $n$  to a number corresponds to moving  $n$  faces in the clockwise direction and subtracting  $n$  corresponds to moving  $n$  faces in the counter clockwise direction. But the really cool part is that you can implement subtraction through addition of negative numbers. For example, suppose that we want to subtract  $3 - 1$ . This is the same as  $3 + (-1)$ . Since the value of -1 is represented by the digit 9, we would move 9 faces in the clockwise direction, which brings us to the face labeled 2, which is the correct answer. This works because the digits wrap around the wheel. Another way to think about this is to add  $3 + 9 = 12$  and then discard the carry out, since the wheel only goes to 9. Using this approach, note further that the sum of a number and its 10's complement is always zero (assuming that the single-digit 10's complement of 0 is 0).

$$\begin{array}{r}
 3 \\
 + 9 \\
 \hline
 \end{array}$$

discard the carry out → 1 2

### 14.1.2 Overflow with Single-Digit 10's Complement Addition

The system breaks down if an addition operation crosses between the face labeled 4, which represents the value +4 and the face labeled 5, which represents the value -5. The important thing to realize is that with 10's complement, the faces labeled with digits 0-9 represent the values in the range -5 through +4. Even though there are faces *labeled* with digits greater than 4, the system does not represent any *values* greater than 4.

For example, suppose that we try to add  $3 + 3$ . Moving 3 faces clockwise from the face labeled 3 brings us to the face labeled 6. But this represents the value -4, not the value +6, so the answer is incorrect. This type of error, where the correct value is outside of the range that can be represented by the system, is known as *overflow*. Overflow is fairly easy to detect:

- If the sum of 2 non-negative numbers (faces labeled 0-4) results in a negative number (faces labeled 5-9), then overflow has occurred.
- If the sum of 2 negative numbers (faces labeled 5-9) results in a positive number (faces labeled 0-4), then overflow has occurred.
- The sum of a positive and negative number can never result in overflow, since the magnitude of the sum will always be less than or equal to the magnitude of either addend and hence be within the range of representation.

Here are a few examples to illustrate:

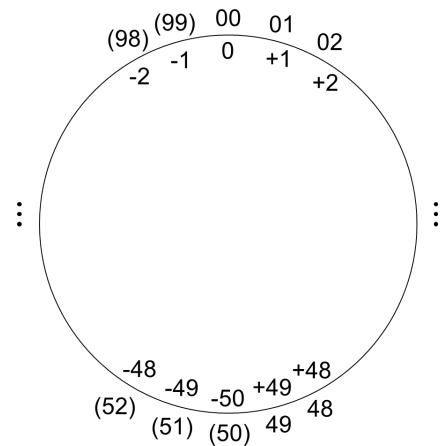
$3 + 2 = 5$	overflow, because the face labeled 5 represents the value -5 and +5 is out of range. More simply, the sum of 2 positives can't be negative
$4 + 7 = 1$	no overflow, because the face labeled 7 represents value -3 and $4 + (-3) = 1$ . The sum of a positive and negative cannot overflow
$6 + 7 = 3$	overflow, because the sum of 2 negatives can't be positive

## 14.2 10's Complement of Multidigit Numbers

The method of complements also applies to larger numbers with more than one digit. Suppose we had a single wheel with 100 faces labeled 00-99. Faces 00-49 would represent the non-negative values in the range 0 through +49 and the faces labeled 50-99 would represent negative values in the range -50 through -1.

The rules for determining the sign and magnitude of a 2-digit number using 10's complement representation are as follows:

- If a face is labeled with a number n in the range 0-49, then the sign is (+) and the magnitude is the value of the label n.
- If a face is labeled with a number n in the range 50-99, then the sign is (-) and the magnitude is the 2-digit 10's complement of the value of the label (100-n).



Here are a few examples of numbers in 2-digit 10's complement representation and their corresponding values in signed decimal (base 10 sign-magnitude) representation.

10's complement	sign	magnitude	signed decimal
35	+	35	+35
99	-	100 - 99	-1
65	-	100 - 65	-35

### 14.2.1 Overflow with Multidigit 10's Complement Addition

Here are a few examples of addition, indicating whether or not there is overflow:

35 + 30 = 65	overflow, because the sum of 2 positives can't be negative
10 + 65 = 75	no overflow, because the sum of a positive and negative can't overflow. The signed decimal representation of this is +10 + (-35) = -25 which is correct.
65 + 55 = 20	overflow, because the sum of 2 negatives can't be positive

We can generalize the method of 10's complement to any number of digits. The rules for determining the sign and magnitude of an m-digit number using 10's complement representation are as follows:

- If a face is labeled with a number n whose most-significant digit is in the range 0-4, then the sign is (+) and the magnitude is the value of the label n.
- If a face is labeled with a number n whose most significant digit is in the range 5-9, then the sign is (-) and the magnitude is the m-digit 10's complement of the value of the label ( $10^m - n$ ).

Here are a few examples to illustrate:

10's complement	digits (m)	sign	magnitude	signed decimal
421	3	+	421	+421
9999	4	-	$10^4 - 9999$	-1
84	2	-	$10^2 - 84$	-16

### 14.2.2 Calculation Trick: 10's Complement as 9's Complement + 1

Suppose that we have a 4-digit mechanical calculator and we want to perform the following subtraction:  $4371 - 0123$ . We could do this by taking the 10's complement of 0123 and then adding that to 4371. But to find the 10's complement of 0123 we would need to subtract it from 10000, so we're still faced with having to do a subtraction, and worse, we don't have enough digits to represent 10000. Not to mention, how do you borrow? So what do we do?

Fortunately, there is a very simple and clever trick to finding the 10's complement of a 4-digit number. Because  $10000 = 9999 + 1$ , we can take the 9's complement of the number and then add 1 to the result. The 9's complement of a number  $n$  in the range 0-9 is equal to  $9 - n$ . This still involves subtraction, but on a digit-wise basis, it's easy to do in your head. Further, because 9 is the largest digit value, you will never have to borrow from the next column when subtracting a number from 9999, and you can take the 9's complement of each digit independently. Thus we could calculate the 10's complement of 0123 as follows:

original value:	0123
9's complement:	9876
10's complement (9's complement + 1):	9877

To complete the subtraction, add  $4371 + 9877 = 4248$  (ignoring the final carry out), which is correct result for the subtraction  $4371 - 0123$ .

In summary, using the method of complements on a mechanical adding machine, we could calculate the subtraction (minuend – subtrahend) with the following steps:

1. Enter the minuend
2. Enter the 9's complement of the subtrahend and add it to the sum
3. Enter 1 and add it to the sum.

The approach of determining the 10's complement of a number as the 9's complement + 1 applies to any number of decimal digits. In order to simplify the second step, some mechanical adding machines such as the Comptometer<sup>14</sup> included the 9's complement of a number in a smaller font on each key:



<sup>14</sup> <http://www.hpmuseum.org/adddir.htm>

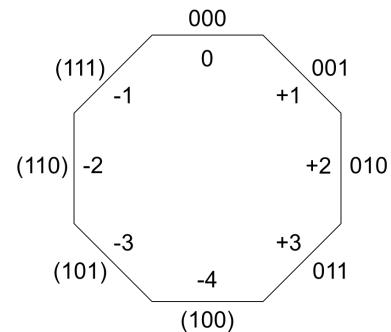
## 14.3 Binary Numbers and 2's Complement

### 14.3.1 Representation

The method of complements applies to binary numbers as well as decimal numbers, and 2's complement representation is the standard way of representing signed integers in most computer systems. As with 10's complement for mechanical calculators, the big advantage of 2's complement in digital computers is that you have a clean, consistent way of representing negative values and that you can use an adder to do subtraction.

Here's the representation and sign-magnitude interpretation of 3-bit binary 2's complement numbers. Similar to the decimal number examples, the first half of the range of label values 000-011 represent non-negative values and the second half 100-111 represent negative values. The rules for determining the sign and magnitude of an m-bit binary number using 2's complement representation are similar to those we used for decimal 10's complement numbers.

- If a face is labeled with a number  $n$  whose most-significant digit is a 0, then the sign is (+) and the magnitude is the value of the label  $n$ .
- If a face is labeled with a number  $n$  whose most significant digit is a 1, then the sign is (-) and the magnitude is the m-digit 2's complement of the value of the label ( $2^m - n$ ).



Here are a few examples to illustrate:

2's complement	digits (m)	sign	magnitude	signed decimal
0101	4	+	5	+5
1110	4	-	$2^4 - 14$	-2
11110	5	-	$2^5 - 30$	-2

### 14.3.2 Calculation Trick: 2's Complement as 1's Complement + 1

Just as we could simplify the computation of the 10's complement of a decimal value as the 9's complement + 1, we can also simplify the computation of a 2's complement binary number as the 1's complement + 1. The reasoning is the same as with 10's complement. The 2's complement of 4-bit binary number  $n$  would be 10000 -  $n$ . But  $10000 = 1111 + 1$ , so we can calculate the 2's complement of  $n$  as  $(1111 - n) + 1$ , where  $(1111 - n)$  is the 4-bit 1's complement of  $n$ .

The 1's complement of a single binary digit (bit)  $n$  is  $1 - n$ , which is simply the bitwise logical inverse of  $n$ ,  $n'$ . So to obtain the 2's complement of an m-bit binary number, follow this simple rule:

The 2's complement of an m-bit binary number  $n$  is  $n' + 1$

Here's an example to illustrate.

Start with the 4-bit 2's complement binary number 0101. This has signed decimal value +5:	0101
To get the 2's complement of that number, invert the bits and add 1:  $\begin{array}{r} 1010 \\ 1 \\ \hline \end{array}$	1010 1 ---- 1011
Add the original number to its 2's complement. The sum should be 0 (discarding the final carry out):  $\begin{array}{r} 0101 \\ 1011 \\ \hline \end{array}$	0101 1011 ---- 0000
The 2's complement of the 2's complement of 0101 should be 0101. We check this by taking the bitwise inverse of 1011 then adding 1:  $\begin{array}{r} 0100 \\ 1 \\ \hline \end{array}$	0100 1 ---- 0101

### 14.3.3 Conversion Rules Using the Invert + 1 Trick

Converting a 2's complement binary number to signed decimal representation:

- If the number has a leading 0 then its sign is (+) and the magnitude is the value of the number
- If the number has a leading 1 then its sign is (-) and the magnitude is the inverse + 1

Converting a signed decimal number to 2's complement binary representation:

- If the sign is positive, convert the magnitude value to binary and pad it with a leading 0
- If the sign is negative, convert the magnitude to binary, invert it, and add 1

### 14.3.4 Overflow in 2's Complement Addition

As with 10's complement decimal numbers, arithmetic overflow occurs when the sum of two 2's complement numbers is out of the range of the representation. The rules for overflow with 2's complement numbers are similar to those for 10's complement numbers:

- If the sum of 2 non-negative numbers (most significant bit is a 0) results in a negative number (most significant bit is a 1), then overflow has occurred.
- If the sum of 2 negative numbers (most significant bit is a 1) results in a positive number (most significant bit is a 0), then overflow has occurred.
- The sum of a positive and negative number can never result in overflow, since the magnitude of the sum will always be less than or equal to the magnitude of either addend and hence be within the range of representation.

We can boil this down to something even simpler:

- If the most significant bit of number is the same and the most significant bit of their sum is different (discarding the final carry-out), then overflow has occurred.
- Overflow cannot occur when adding two numbers whose most significant bits are different.

Examples:

0101 + 0100 = 1001	overflow, the sum of 2 positives can't be negative
1110 + 1000 = 0110	overflow, the sum of 2 negatives can't be positive
1110 + 0011 = 0001	<p>no overflow, since the sum of a positive and negative will always be in range.</p> <p>The signed decimal representation of this problem is <math>-2 + 3 = 1</math></p> <p>Note that just because there was a carry out of the most-significant bit that was discarded, that doesn't mean that there was overflow—this is just the natural “wrapping around” from negative to positive numbers.</p>

## 15 Appendix: Verilog Language Rules

- Verilog is a case sensitive language (with a few exceptions)
- Identifiers (space-free sequence of symbols)
  - upper and lower case letters from the alphabet
  - digits (0, 1, ..., 9)
  - underscore ( \_ )
  - \$ symbol (only for system tasks and functions)
  - Max length of 1024 symbols
- Terminate lines with semicolon (;)
- Single line comments: // A single-line comment goes here
- Multi-line comments:  
/\* Do not /\* nest multi-line comments \*/ like this \*/

The following is the set of reserved Verilog keywords. You cannot use reserved keywords as identifiers for the names of signals, modules, etc.

always	end	initial	output	rtran	tranif1
and	endcase	inout	parameter	rtranif0	tri
assign	endfunction	input	pmos	rtranif1	tri0
begin	endmodule	integer	posedge	scalared	tri1
buf	endprimitive	join	primitive	small	triand
bufif0	endspecify	large	pull0	specify	trior
bufif1	endtable	macromodule	pull1	specparam	trireg
case	endtask	medium	pulldown	strength	vectored
casex	event	module	pullup	strong0	wait
casez	for	nand	rcmos	strong1	wand
cmos	force	negedge	real	supply0	weak0
deassign	forever	nmos	realtime	supply1	weak1
default	fork	nor	reg	table	while
defparam	function	not	release	task	wire
disable	highz0	notif0	repeat	time	wor
edge	highz1	notif1	rnmos	tran	xnor
else	if	or	rnmos	tranif0	xor