

# What are the Fundamentals of Digital Design?

Richard E. Haskell and Darrin M. Hanna  
CSE Dept., Oakland University, Rochester, Michigan 48309

## ABSTRACT

Everyone agrees that engineering programs should teach the fundamentals. The disagreement comes in agreeing on what the fundamentals are. Perhaps we can agree that the fundamentals are those concepts, theories, and practices that have long-term staying power and are still useful decades into the future. How does such a viewpoint apply to the teaching of digital design? In this paper we will argue that it is the *behavioral specifications* of digital components and systems that have not changed and are therefore the fundamentals that should be the centerpiece of courses in digital design.

This paper describes a sophomore/junior course in computer hardware design that we have taught at Oakland University for the past six years. In this course the students begin by studying basic logic gates and circuits and then proceed to design a complete 16-bit stack-based microprocessor using VHDL and implement it in a Xilinx FPGA. During the last three weeks of the course the students, working in groups of 3-4, complete a project in which they write a software program and compile it to execute on their custom microprocessor that they have designed. These projects have ranged from video games to a real-time software debugger. The only way that all of this material can be included in a single course is to focus on the *fundamentals*.

## Introduction

The first digital circuits used relays, the original key to speaking binary, the fundamental language of computing. But no one would argue that relays, or the electromagnetic theory on which they rely, are fundamental to digital design today. The same could be said of vacuum tubes, transistors, TTL, CMOS, PLDs, CPLDs, FPGAs, ASICs, or any other implementation mode that is popular at a particular time. Inasmuch as any logic circuit can be made using AND, OR, NAND, NOR, and NOT gates representing high-level logic compared to their transistor fabric they have been considered fundamental for some time. In fact, any logic circuit can be made from only NAND gates or from only NOR gates, escalating their status to *universal gates*. If one peeks inside some modern silicon system such as an FPGA one looks in vain for any of these gates as they are concealed by the FPGA's fundamental lookup tables within the configurable logic blocks (CLBs). Most digital logic courses spend considerable time on reducing logic equations particularly by using Karnaugh maps to derive reduced, fundamental Boolean logic. But engineers who design modern digital circuits for a living never use Karnaugh maps. The vast majority of digital systems today are designed by using either Verilog or VHDL, the present-day fundamental hardware description languages. Surely the syntax and semantics of a particular hardware description language no matter how universally used and which may be replaced by a "better" hardware description language in the future, can not be really fundamental. So what is it that has remained constant over the last several decades of digital design? In this paper we will argue that it is the *behavioral specifications* of digital components and systems that have not changed and are therefore the fundamentals that should be the centerpiece of courses in digital design.

For example, the important thing to know about a 2 x 1 multiplexer with inputs  $A$ ,  $B$ , and  $S$ , and output  $Z$  is that the output  $Z$  is equal to  $A$  when  $S = 0$  and is equal to  $B$  when  $S = 1$ . It is much less important to know what particular arrangement of gates can implement this multiplexer, because your particular implementation, for example in an FPGA, may not contain any gates at all. This viewpoint has important implications of what and how topics should be taught in a digital design course.

In this paper we describe a sophomore/junior course in computer hardware design that we have taught at Oakland University for the past six years. In this course the students begin by studying basic logic gates, combinational logic circuits, and sequential circuits by focusing on the *behavior* of these circuits. The students then proceed to design a complete 16-bit stack-based microprocessor using VHDL and implement it in a Xilinx FPGA. At the end of the course the students, working in groups of 3-4, complete a project in which they write a software program and compile it to execute on their custom microprocessor that they have designed. These projects have ranged from video games to a real-time software debugger. The only way that all of this material can be included in a single course is to focus on the *fundamentals*.

### **Behavior, Implementation, and History**

The course, *CSE 378, Computer Hardware Design*, is a junior-level course taken by all computer engineering and computer science majors at Oakland University. At the beginning of the course we tell the students that everything we cover will fall into one of three categories: 1) the *behavior* of a digital circuit, system, or component (which we argue is the most *fundamental* in the sense of having long-term staying power); 2) the *implementation* of a particular digital circuit, system, or component (which is fun and uses the latest technology, which will likely be replaced by a newer technology that is even more fun next year); and *history* (such as how the “totem-pole” output of a TTL chip works) that is, well, history. At each stage in the course we urge the students to decide into which category the particular topic of the day falls. Learning material in the third category (*history*) gives the students the perspective of understanding how we got to the current state of affairs. Learning material in the second category (*implementation*) will help the student get a job next year. Learning material in the first category (*behavior*) will help the student get a job ten years from now.

How do we describe the *behavior* of digital circuits? We describe it in words. For example,

- The output of an AND gate is HIGH only if all inputs are HIGH.
- The output of an OR gate is LOW only if all inputs are LOW.
- The output of a NAND gate is LOW only if all inputs are HIGH.
- The output of a NOR gate is HIGH only if all inputs are LOW.

For larger circuits it is convenient to use some type of hardware description language (HDL) to describe the behavior of the circuit. We chose to use VHDL but Verilog (or even C) could also be used. Using VHDL or Verilog has the advantage of being able to simulate and synthesis the designs using widely available tools. We use Aldec Active-HDL for simulation and the Xilinx ISE Project Navigator for synthesis to Xilinx FPGAs. In this course, each student purchases the Spartan-3 board from Digilent.<sup>1</sup>

## The Fundamentals of Combinational and Sequential Circuits

The fundamentals of basic combinational and sequential circuits are introduced by describing their behavior in terms of VHDL statements within a VHDL architecture. For example, an 8-line, 2 x 1 multiplexer is described in Figure 1, a 4-bit adder is described in Figure 2, and a 4-bit shifter is described in Figure 3. The VHDL code in all of these examples can directly be simulated and synthesized to an FPGA.

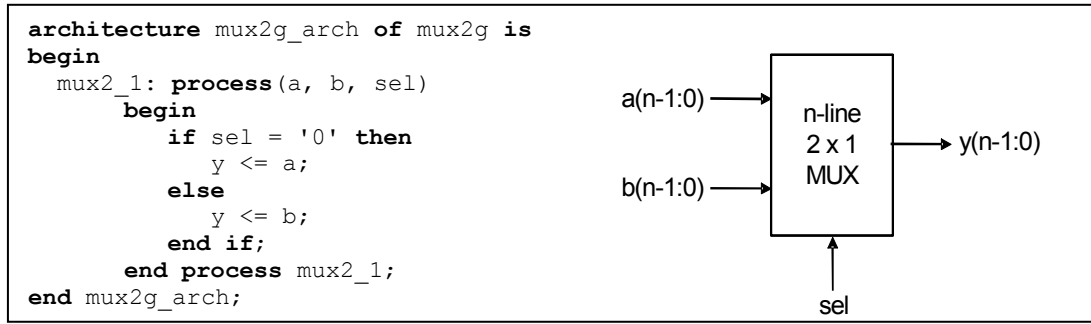


Fig. 1 Describing the behavior of a 2 x 1 multiplexer

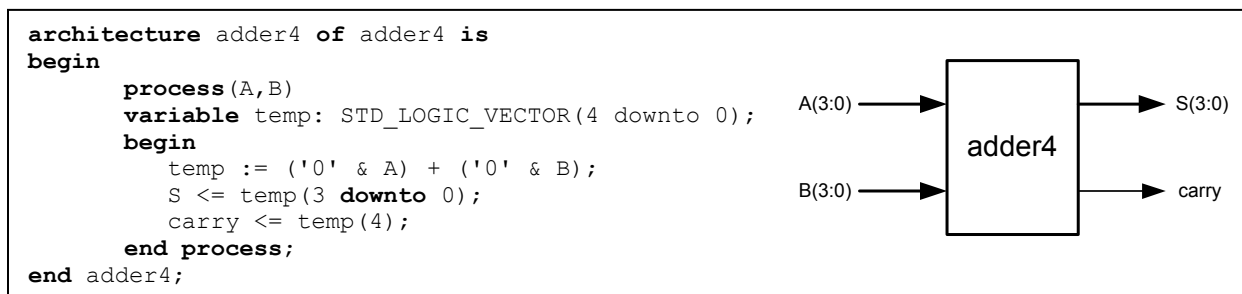


Fig. 2 Describing the behavior of a 4-bit adder

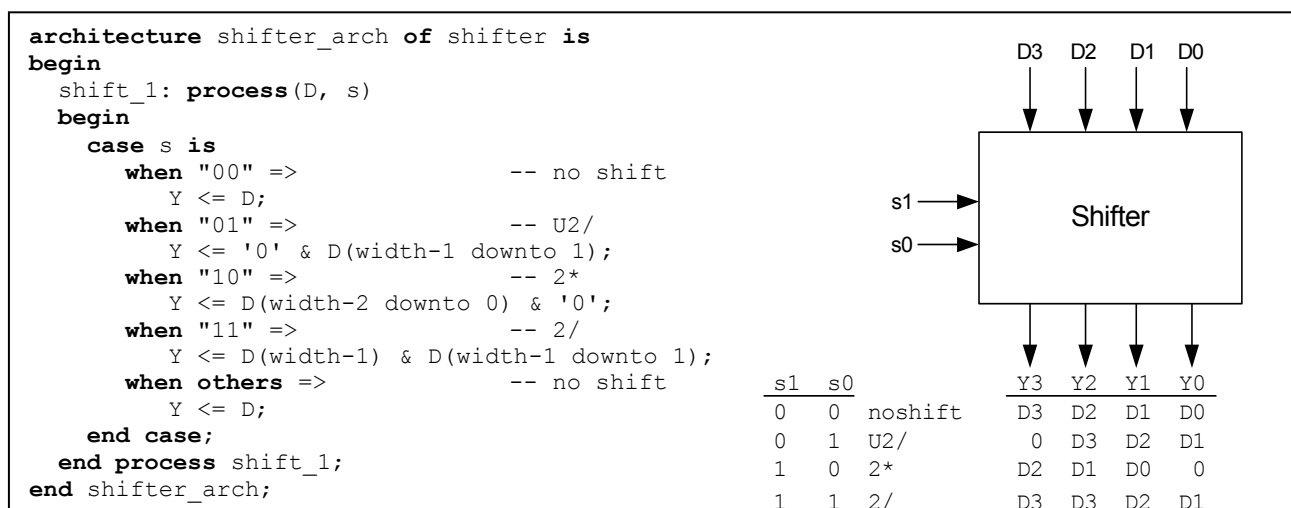


Fig. 3 Describing the behavior of a 4-bit shifter

As another example, the “Shift and Add 3” algorithm for converting an 8-bit binary number to BCD is shown in Figure 4. The VHDL description of this behavior is shown in Figure 5 and the result of the simulation of this code is shown in Figure 6.

Binary-to-BCD Conversion:									
1. Shift the binary number left one bit.									
2. If 8 shifts have taken place, the BCD number is in the <i>Hundreds, Tens, and Units</i> column.									
3. If the binary value in any of the BCD columns is 5 or greater, add 3 to that value in that BCD column.									
4. Go to 1.									

Operation	Hundreds	Tens	Units	Binary						
HEX										
Start					F			F		
Shift 1				1	1	1	1	1	1	1
Shift 2				1	1		1	1	1	1
Shift 3				1	1	1	1	1	1	1
Add 3				1	0	1	0	1	1	1
Shift 4			1	0	1	0	1	1	1	1
Add 3			1	1	0	0	0	1	1	1
Shift 5			1	1	0	0	0	1	1	1
Shift 6			1	1	0	0	1	1	1	1
Add 3			1	0	0	1	1	0	1	1
Shift 7	1		0	0	1	0	0	1	1	1
Add 3	1		0	0	1	0	1	0	1	1
Shift 8	1	0	0	1	0	1	0	1	0	1
BCD	<b>2</b>		<b>5</b>		<b>5</b>					
P	9 8	7	4	3	0					
z	17 16	15	12	11	8	7	4	3	0	

Fig. 4 “Shift and add 3” algorithm for converting binary to BCD

```

architecture binbcd_arch of binbcd is
begin
  bcd1: process(B)

    variable z: STD_LOGIC_VECTOR (17 downto 0);

  begin
    for i in 0 to 17 loop
      z(i) := '0';
    end loop;
    z(10 downto 3) := B;
    for i in 0 to 4 loop
      if z(11 downto 8) > 4 then
        z(11 downto 8) := z(11 downto 8) + 3;
      end if;
      if z(15 downto 12) > 4 then
        z(15 downto 12) := z(15 downto 12) + 3;
      end if;
      z(17 downto 1) := z(16 downto 0);
    end loop;
    P <= z(17 downto 8);
  end process bcd1;
end binbcd_arch;

```

Fig. 5 VHDL behavior of “Shift and add 3” algorithm in Fig. 4

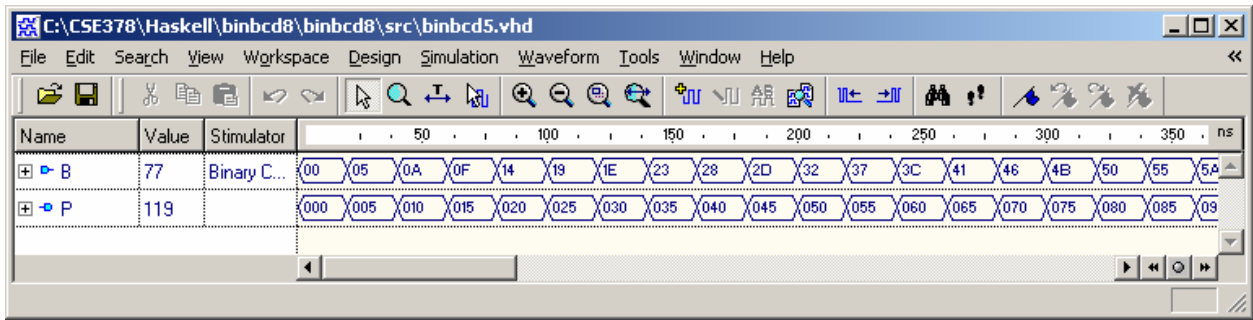


Fig. 6 Simulation of binary-to-BCD VHDL code in Fig. 5

The behavior of an R-S latch is given in Figure 7 and its simulation is shown in Figure 8. Note that this behavior is different from an R-S latch made from crossed NOR gates in that it doesn't have a "disallowed state" in which both  $Q$  and  $\text{NOT } Q$  are zero when  $R$  and  $S$  are both 1. In our case the output  $Q$  remains unchanged. The students make a truth table for this version of an R-S latch and show how to implement one in terms of AND, OR, and NOT gates.

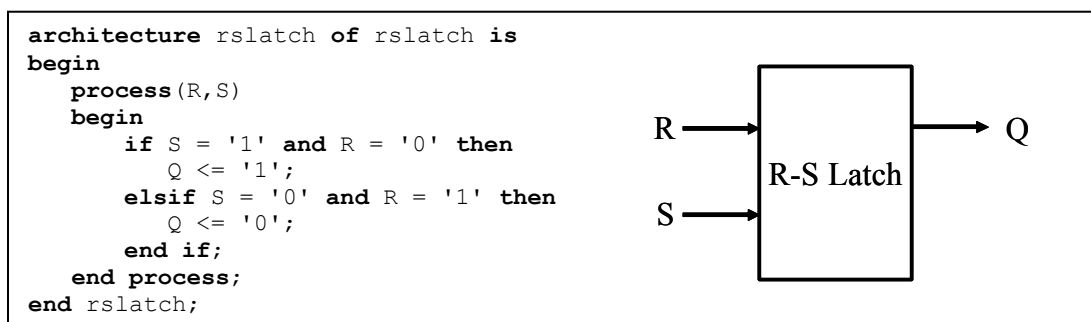


Fig. 7 A behavioral description of an R-S latch

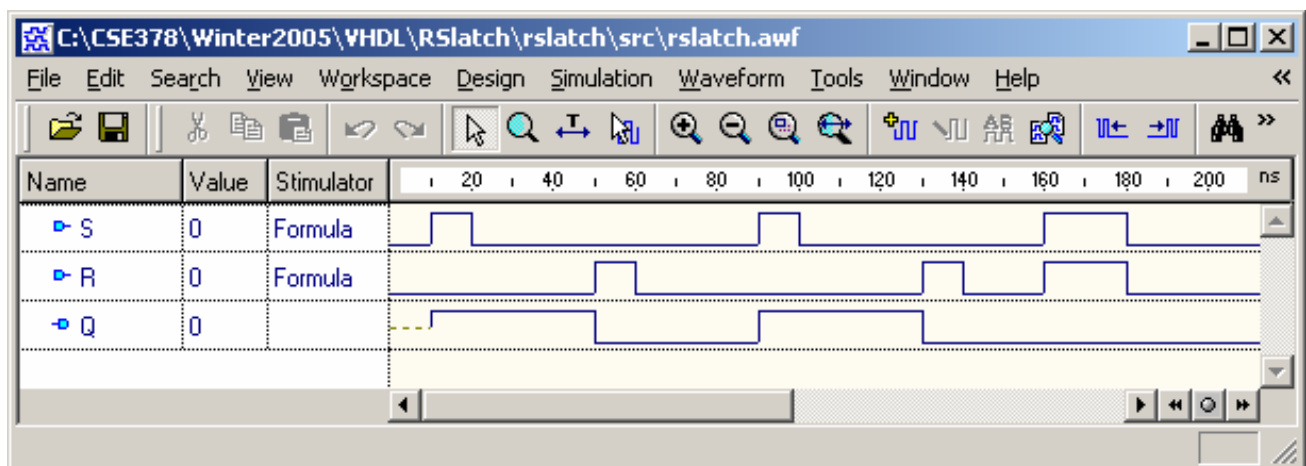


Fig. 8 Simulation of the R-S latch shown in Fig. 7

The behavior of an  $n$ -bit register with an asynchronous clear and a *load* input is given in Figure 9. The behavior of an  $n$ -bit counter that can be used as a program counter is given in Figure 10. This counter has an asynchronous clear and *load* and *inc* inputs.

In addition to these examples the students use VHDL to describe the behavior of 7-segment decoders, comparators, decoders, Gray code converters, arithmetic logic units, ROMs, D latches, D flip-flops, and shift registers. For each of these cases they can synthesize the design to a Xilinx Spartan 3 FPGA by adding a wrapper that includes the switch and pushbutton inputs and the LED and 4-digit 7-segment display of their Digilent Spartan-3 board.

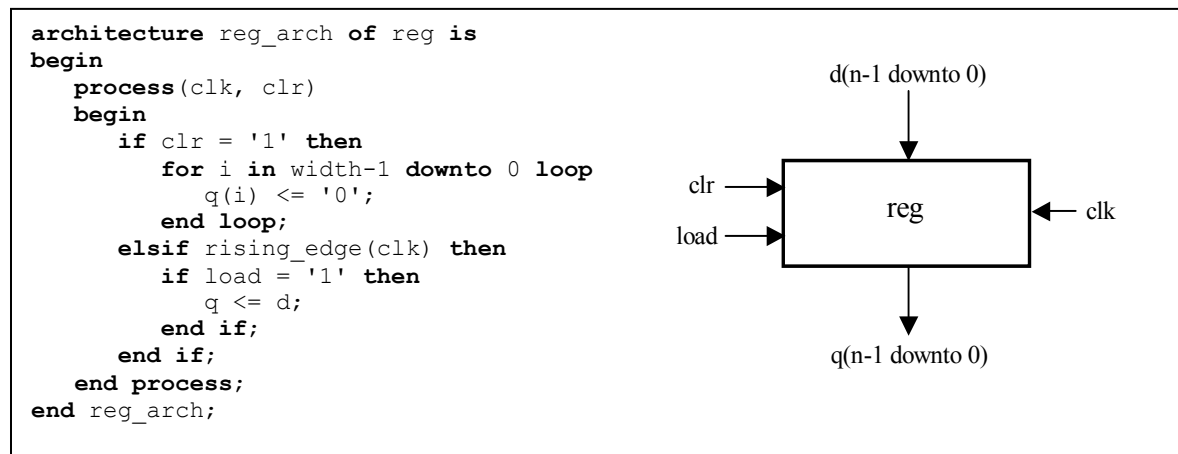


Fig. 9 Describing the behavior of an  $n$ -bit register

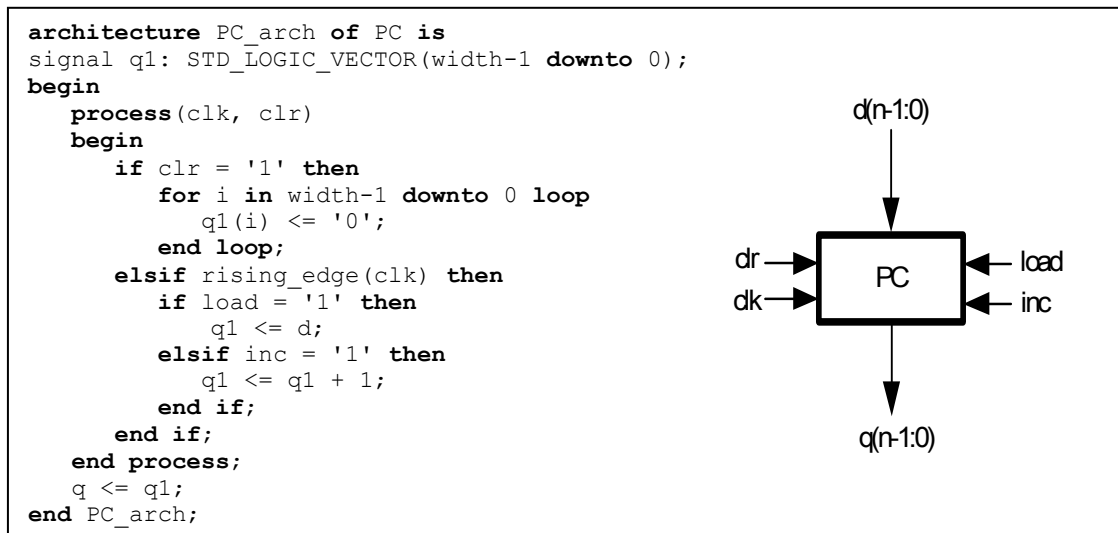


Fig. 10 Describing the behavior of an  $n$ -bit program counter

## The Design of a Stack-Based Microprocessor Core

Forth is a programming language that uses a data stack and postfix notation. Everything in Forth is a word and every word is a module that does something useful. Forth words accept parameters on the data stack, execute themselves, and return the answers back on the data stack.

In the junior-level course, *CSE 378, Computer Hardware Design*, students design and implement the FC16 Forth core shown in Figure 11. This microprocessor core contains four main components, the data stack, *DataStack*, the function unit, *Funit16*, the return stack, *ReturnStack*, and the controller, *FC16\_control*. The FC16 also contains a program counter, *PC*, whose output, *P*, containing the address of the next instructions, is the input to the program ROM outside the FC16. The output of the ROM is the signal, *M*, which can be loaded into the instruction register, *IR*, pushed onto the data stack through the multiplexer, *Tmux*, or loaded into the program counter, *PC*, through the multiplexer, *Pmux*. A detailed description of this Forth core is given in the references.<sup>2,3</sup>

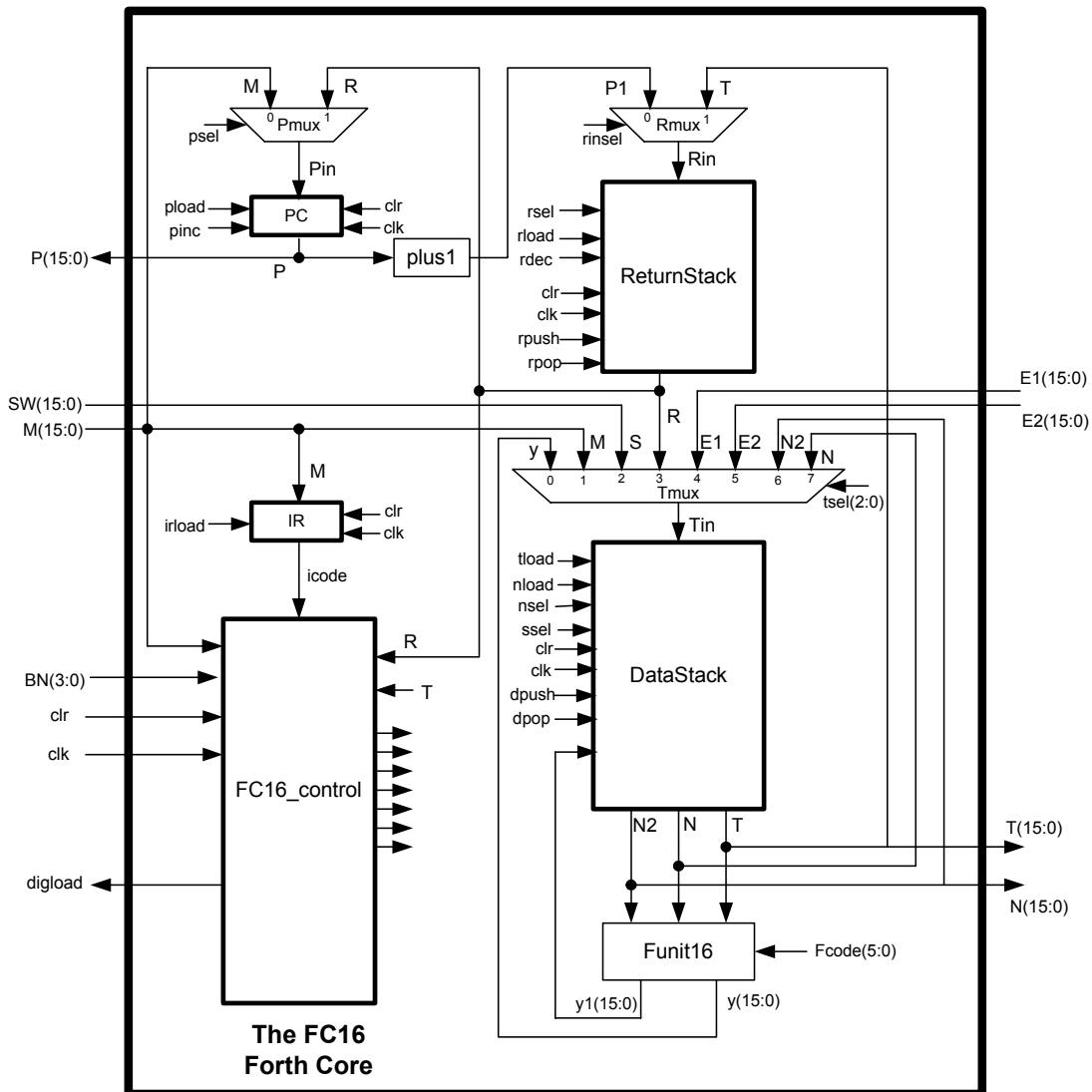


Fig. 11 The FC16 Core that executes Forth instructions

The FC16 data stack is a modified 32x16 stack whose architecture is shown in Figure 12. The FC16 data stack shown in Figure 12 consists of two 16-bit registers for the top and second elements of the data stack followed by a 32x16 stack implemented using a dual-port RAM. These registers, *Treg* and *Nreg*, serve as ‘false top’ and ‘false second’ elements in the data stack, respectively. This architecture is necessary to support single-clock-cycle execution of Forth instructions such as ROT involving the top three stack elements.

Forth programs can easily be compiled to hardware by translating the program to VHDL code for a ROM that contains the corresponding FC16 instructions. A C++ program that translates Forth programs to 68HC12 assembly language is described in Haskell.<sup>4</sup> A modification of this C++ program has been used to produce a VHDL ROM array directly from a Forth program. This makes it easy to quickly change programs, compile them to a VHDL ROM, and download them to the FPGA for testing.

The FC-16 microprocessor core contains over 60 Forth instructions, 75% of which execute in a single clock cycle. Forth programs written for the FC-16 typically execute at 25 MHz. After completing the design and implementation of the FC-16 core the students work in groups of 2 or 3 to design and implement a project that involves the use of the FC-16 perhaps with addition of new hardware and new instructions.

### Student Projects

Using the FC16 core developed throughout the course, CSE 378 students produce final projects in groups of three or four during the last three weeks. The project guidelines require students to develop software for a derivative of the FC16 core made by any modifications they need to implement their project. Since Forth is a modular high-level language combined with the knowledge and experience they’ve obtained from

this FPGA-based hardware design course, substantial projects have been designed, implemented, tested, and demonstrated in only three weeks. In addition to creating the completed, working project, each group must deliver a 15-minute PowerPoint presentation, demonstrate the project in class, submit a written report detailing the processor and their project, and construct a creative poster for public display. Some of these projects are described below.

### *A Real-time Forth Compiler and Debugger*

Development of Forth programs for the FC16 microprocessor requires that compiled software is implemented in a program ROM. This ROM and FC16 project must then be compiled using ISE for synthesis with each program change. In this project, students made hardware modifications and developed a Visual Basic application to streamline software development for the FC16.

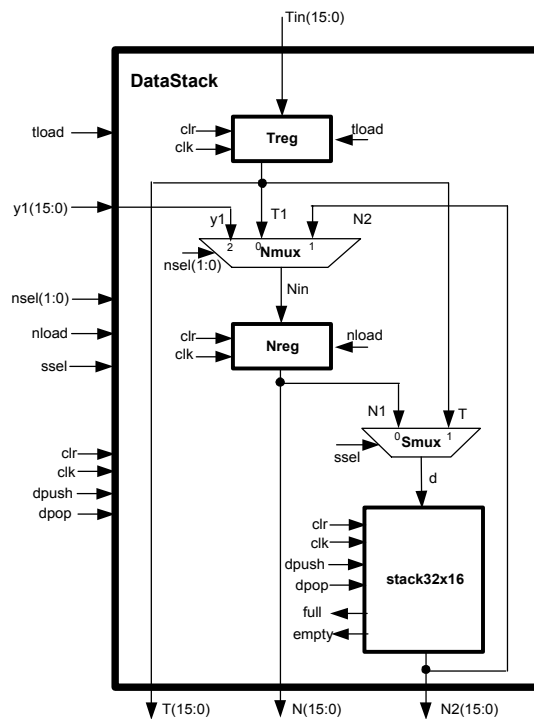


Fig. 12 The data stack

Through the use of parallel FC16 processors implemented on the Spartan2E FPGA and the Visual Basic application; real-time compilation, execution, and debugging are realized for rapid development and testing of Forth code. Figure 13 shows a screenshot of the Visual Basic interactive debugger.

This design required a modified FC16 with minor modifications to a slave FC16 that executes the programmer's code. This makes it easy for future students to replace the FC16 with a different processor and still use this interactive programmer and debugger. Figure 14 shows a diagram of the modifications made to the FC16 processor shown previously in Figure 11 for the real-time programmer and debugger. The *CD* component is the compiler-debugger. The *SLAVE* component is the FC16

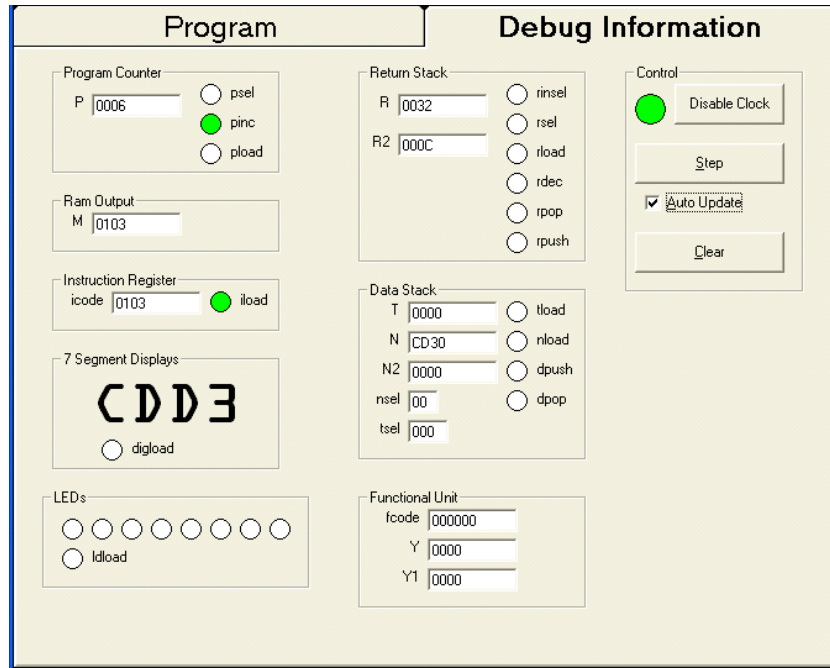


Fig. 13 A real-time FC16 debugger in Visual Basic

processor shown in Figure 11 with some extra output signals connected to the *CD*. All of these modifications were designed and implemented by the student group.

Figure 15 shows the hardware required inside the compiler-debugger, *CD*. This component is also a modified FC16 core.

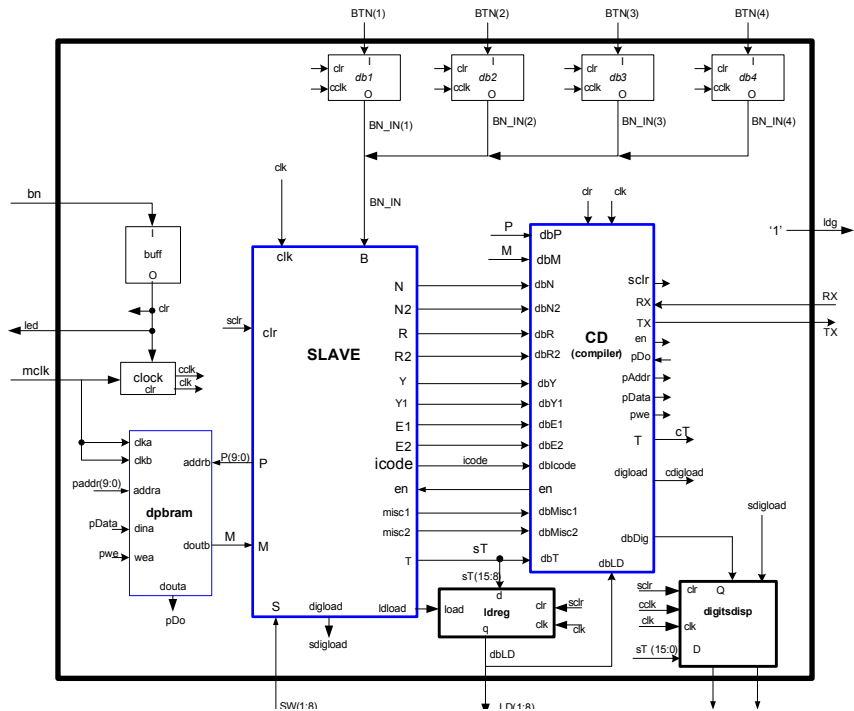


Fig. 14 The Hardware for the Real-time Forth Compiler-Debugger and FC16 Microcontroller

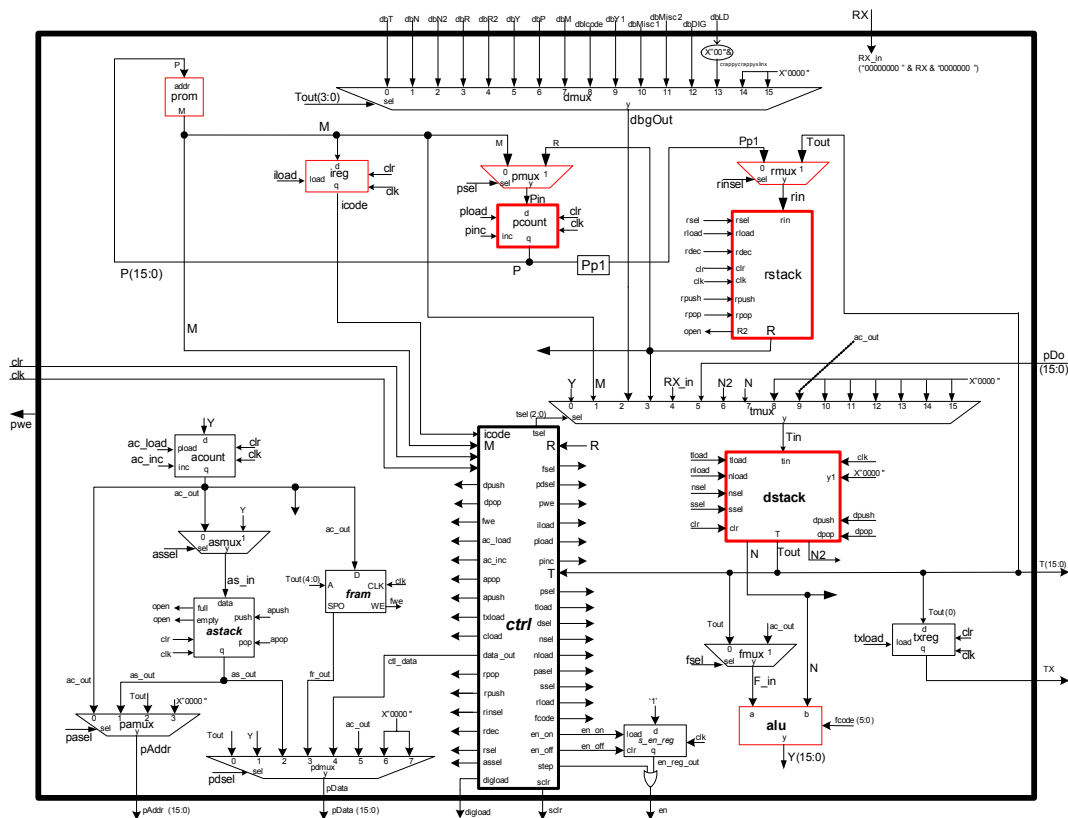


Figure 15 Inside the Compiler-Debugger Component, CD

### A Video Game: Tetris

In this project, students implemented a hardware video driver that controls the horizontal sync, vertical sync, red, green, and blue signals for a VGA monitor and implemented the game of Tetris. They used a Nintendo game controller interfaced to the Digilent Digilab IIE Protoboard using open pins to allow players to rotate game pieces and accelerate the piece downward. These students also used the FPGA block RAM and created a Tetris Control Component to implement the non-trivial game logic. Figure 16 shows a block diagram of the hardware developed for this project and a screenshot of the VGA game in action.

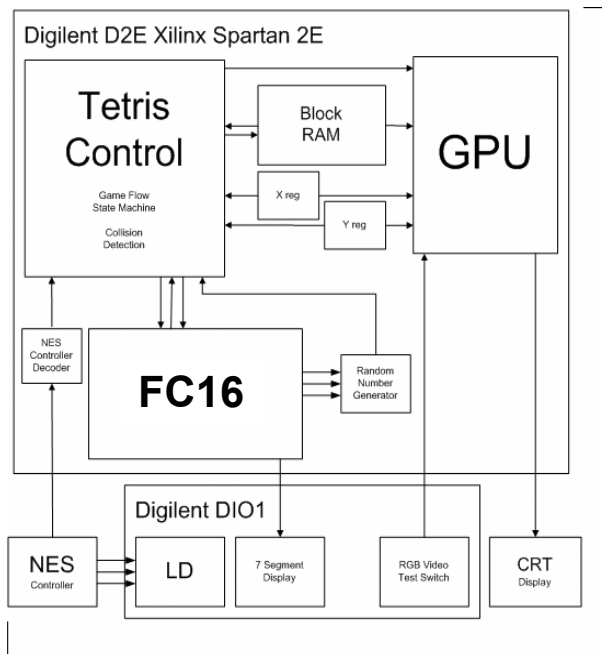


Fig. 16 A Hardware Block Diagram for the FC16 Tetris Video Game

## The Card Gallery

Students implemented a card gallery comprised of a generic card gaming engine and four card-game modules. The games are played using the VGA screen controlled by the video driver designed by the students and implemented in the FPGA and the buttons on the Digilent Digilab IIE Protoboard. The four games implemented were High-Low, Black Jack, In Between, and War. Figure 17 shows a block diagram of the Card Gallery including a screenshot of one of the games, In Between, in action.

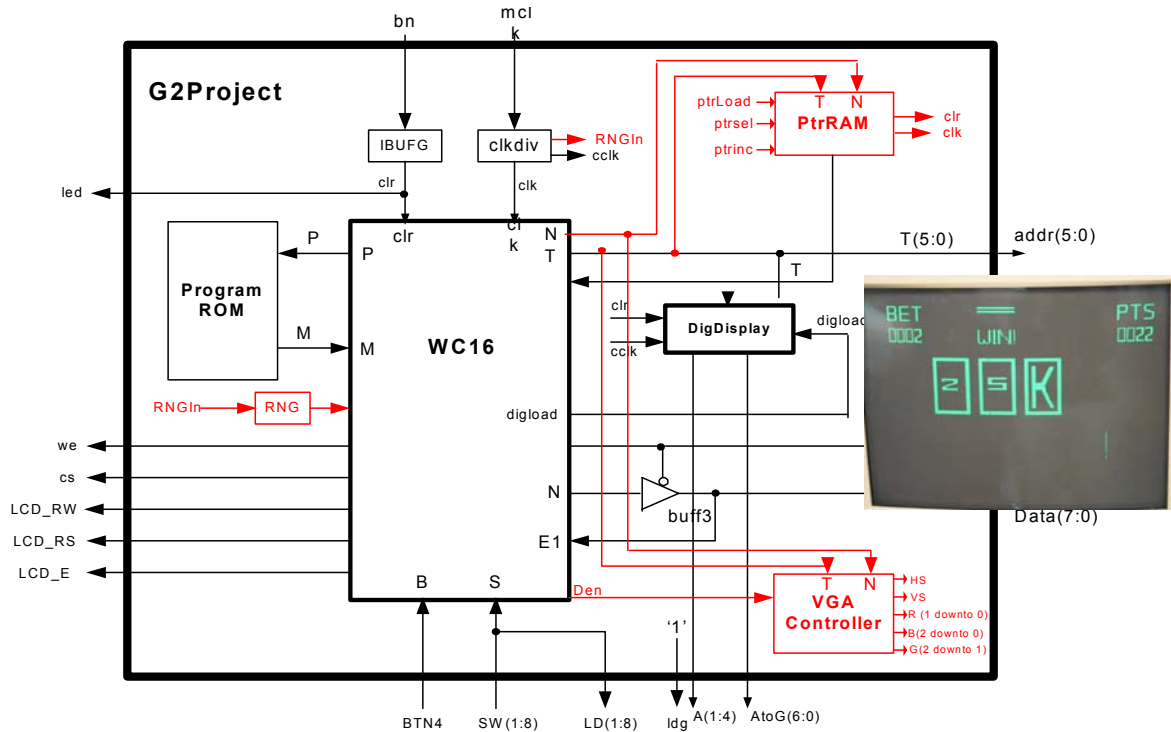


Fig. 17 The Card Gallery Block Diagram and Screenshot

All of these projects represent computer hardware design employing the fundamentals of digital logic, the behavior of digital systems. These student projects were implemented on modern RAM-based FPGA technology using VHDL, simulated by Aldec's ActiveHDL, and synthesized using Xilinx's ISE Foundation Tools.

## Summary

In this paper we have argued that it is the *behavioral specification* of digital circuits that remains constant over time and is therefore more fundamental than any particular implementation technology. We have showed how this approach allows students in a junior-level course in computer hardware design to design and implement a complete 16-bit microprocessor core from scratch and then write a sophisticated high-level software program to perform some useful task and execute this program on their personally-designed microprocessor core.

## References

1. [www.digilentinc.com](http://www.digilentinc.com).
2. Haskell, R. E. and D. M. Hanna, "Rapid Prototyping using a Microprocessor Core on a Spartan II FPGA," Proc. of the International Conference on Embedded Systems and Applications, ESA'03, pp. 49-55, Las Vegas, Nevada, USA, June 23-26, 2003.
3. Haskell, R. E. and D. M. Hanna, "A VHDL Forth Core for FPGAs," Microprocessors and Microsystems, Vol. 28/3 pp. 115-125, Apr 2004.
4. Haskell, R. E., *Design of Embedded Systems Using 68HC12/11 Microcontrollers*, Prentice Hall, Upper Saddle River, NJ, 2000.

RICHARD E. HASKELL is Professor of Engineering in the Department of Computer Science and Engineering at Oakland University. He is the author of 15 books and has taught numerous undergraduate and graduate courses including courses in microprocessors, embedded systems and digital design using VHDL.

DARRIN M. HANNA is Assistant Professor of Engineering in the Department of Computer Science and Engineering at Oakland University. His primary area of research is pattern recognition and artificial intelligence and embedded systems.