Microprocessors and Microsystems, Vol. 28/3 pp. 115-125, Apr 2004.

A VHDL Forth Core for FPGAs

Richard E. Haskell and Darrin M. Hanna
Computer Science and Engineering Department
Oakland University
Rochester, Michigan 48309

Abstract

The Forth programming language is typically implemented to run on some particular microprocessor. Several Forth engines have been designed that execute Forth instructions directly, typically in a single clock cycle. With the advent of high density FPGAs it has become feasible to implement a high-performance Forth core in an FPGA. This paper describes the design of a Forth core using VHDL that has been implemented on a Xilinx Spartan II FPGA. Examples are presented of high-level Forth programs that are compiled to VHDL code that implements a ROM embedded in the FPGA. The use of a Forth core in an FPGA allows for rapid prototyping of digital systems. Experiments show that an identical Forth program for the Sieve of Eratosthenes executes nearly 30 times faster on the FPGA Forth core than on a 68HC12 microcontroller at the same clock speed. This same program executes over 6 times faster on the FPGA Forth core than an equivalent compiled C program run on the same 68HC12. The Forth core is available as an EDIF file at www.tigs.com/fc16, which can be included in a VHDL project and uses approximately 30% of a Spartan II FPGA.

Keywords: Forth core, VHDL, FPGA, Stack-based microprocessor, rapid prototyping

1. Introduction

Forth has been implemented on many microprocessors including the Motorola 68HC12 [8]. As the density of an FPGA (in terms of the number of equivalent gates) has increased while its cost has decreased, it is becoming feasible to consider putting all functions, including a microprocessor core, into the same FPGA forming a true System-on-a-Chip (SOC). The software running on the microprocessor core would also be stored in the form of instructions in the same FPGA.

Forth is a programming language that uses a data stack and postfix notation. Chuck Moore invented Forth in the late 1960s while programming minicomputers in assembly language. His idea was to create a simple system that would allow him to write many more useful programs than he could using assembly language. The essence of Forth is simplicity—always try to do things in the simplest possible way. Forth is a way of thinking about problems in a modular way. It is modular in the extreme. Everything in Forth is a word and every word is a module that does something useful. There is an action associated with Forth words. The words execute themselves. Forth words accept parameters on the data stack, execute themselves, and return the answers back on the data stack.

Forth has been implemented in a number of different ways. Chuck Moore's original Forth had what is called an *indirect-threaded* inner interpreter. Other Forths have used what is called a *direct-threaded* inner interpreter. These inner interpreters get executed every time you go from one Forth word to the next; i.e. all the time. A unique version of Forth called *WHYP* (pronounced *whip*) has recently been described in a book on embedded systems [8]. WHYP stands for Words to Help You Program. WHYP is what is called a *subroutine-threaded* Forth. This means that the subroutine calling mechanism that is built into the 68HC12 is used to go from one WHYP word to the next. In other words, WHYP words are 68HC12 subroutines.

Inasmuch as Forth (and WHYP) programs consist of sequences of words, the most often executed instruction is a call to the next word; i.e. executing the inner interpreter (NEXT) in traditional Forths, or calling a subroutine in WHYP. Over 25% of the execution time of a typical Forth program is used up in calling the next word [15]. To overcome this problem, Chuck Moore designed a computer chip, called NOVIX, in the mid-eighties, which could call the next word (equivalent to a subroutine call) in a single clock cycle [5]. Many of the Forth primitive instructions would also execute in a single clock cycle. The design of the NOVIX chip was eventually sold to Harris Semiconductor where it was redesigned as the RTX 2000 [6]. Similar 32-bit Forth engines were also developed [12,13,15]. In the late eighties Chuck

Moore designed a 32-bit microprocessor called ShBoom that had 64 8-bit instructions and was designed to interface to DRAM [17]. Later, Chuck Moore and C. H. Ting designed the MuP21 that has been described by Ting [18,19]. In 1999 we designed the W8X microcontroller [7] that was based on ideas developed in these early Forth engines. It was designed using VHDL [1] and has been implemented in a Xilinx FPGA by students in a junior-level course at Oakland University [9]. A variation of the W8X, the W8Z, that implements only those instructions used in a particular program has also been implemented on FPGAs [10].

This paper describes the design of a complete 16-bit Forth core that has been implemented on a Xilinx Spartan II FPGA. Section 2 describes the overall architecture of the F16 Forth Core. The data stack and data stack instructions are described in Section 3. The function unit, which implements arithmetic, logical, shifting, and relational instructions is detailed in Section 4. The operation of the return stack and the return stack instructions are discussed in Section 5. The operation of the control unit is described in Section 6. Some examples of using this Forth core for rapid prototyping [11] in a Xilinx Spartan II FPGA are given in Section 7. Experimental results showing that an identical Forth program for the Sieve of Eratosthenes executes over 25 times faster on the FPGA Forth core than on a 68HC12 microcontroller at the same clock speed are given in Section 8. The operation of the FC16 Forth core is summarized in Section 9.

2. The FC16 Forth Core

The FC16 is a high-performance microprocessor that can be implemented on an FPGA to execute embedded programs. The overall structure of the FC16 is shown in Figure 1. The data busses in this figure are 16 bits wide and each instruction is a 16-bit word.

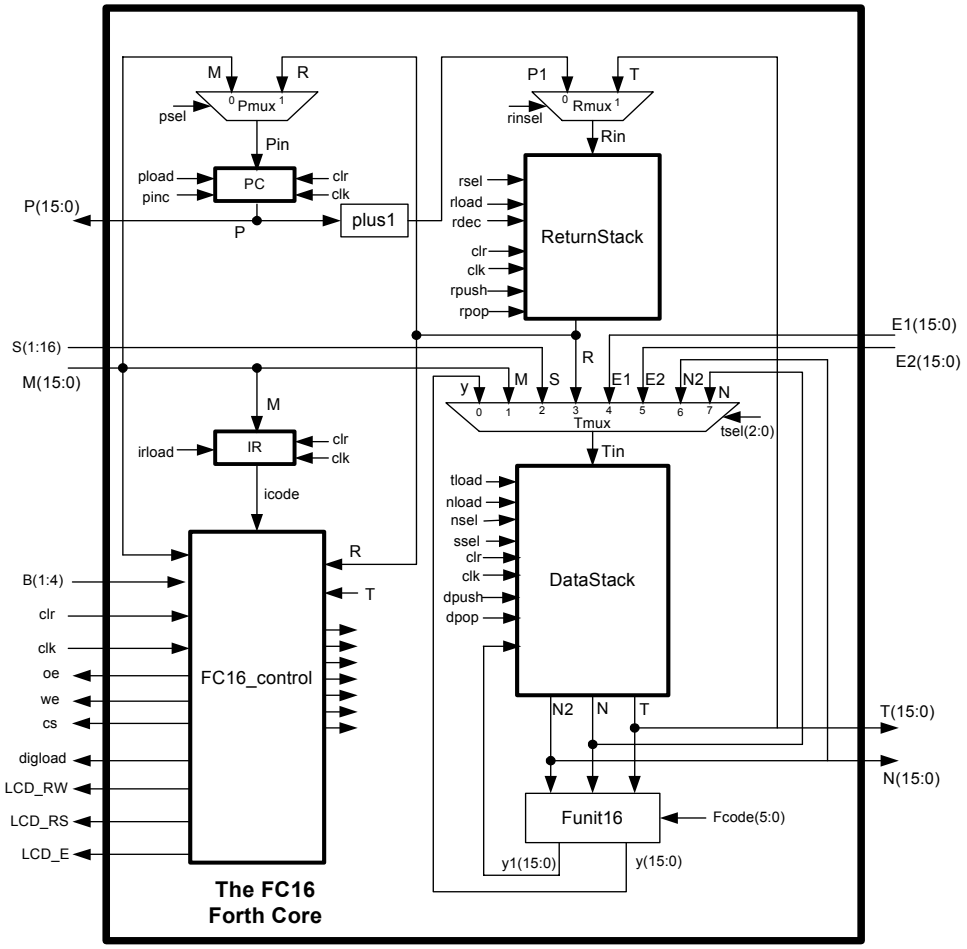

Figure 1 Functional diagram of the FC16 Forth core

The FC16 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 shown outside the FC16 core in Figure 2. 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*.

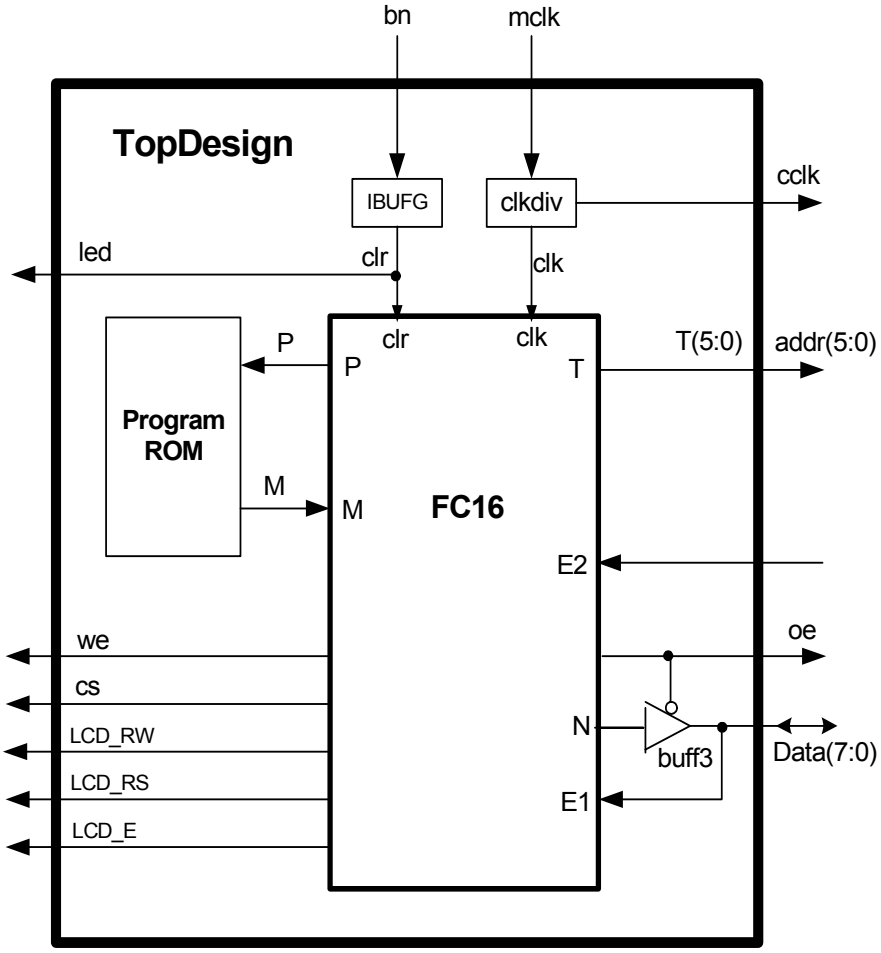

Figure 2 Example of a top-level design using the FC16 Forth core

The example of using the FC16 core shown in Figure 2 represents the top-level VHDL design that was downloaded to a Xilinx Spartan IIE FPGA on a Digilab D2 development board produced by Digilent, Inc. [2]. Figure 2 shows the signals needed to interface the Digilab D2 development board to the DIO2 peripheral board developed by Digilent, Inc. [2]. The Digilab DIO2 board features sixteen LEDs, a 16 x 2 liquid crystal display, eight switches, fifteen pushbuttons, four 7-segment displays, a VGA port, and a PS/2 port. An example of using the FC16 core to make a calculator on the DIO2 peripheral board is given in [11].

Other memory and I/O modules could be added to the top-level design shown in Figure 2. For example, a RAM module would input data from the *N* bus (the second element on the data stack) and the address from the *T* bus (the top element on the data stack). The output of the RAM would be fed back to the top of the data stack through the *E2* bus. The write enable signal, *we*, would be used to write data to the RAM module. A ROM module containing constant data would connect its address input to the *T* bus and

its output to the *E2* bus. Special Forth words for accessing these RAM and ROM modules will be described in Section 6.

The top of the data stack can be loaded from eight different signals through the 8-to-1 multiplexer, *Tmux*, shown in Figure 1. One of these signals is *S*, which can be connected to external switches. The instruction *S@* will push the value of *S* onto the data stack. The next section provides a more detailed description of the operation of the data stack.

3. The Data Stack

The FC16 data stack is a modified 32x16 stack. Table 1 shows the basic stack operations performed by the FC16. The architecture of this data stack is shown in Figure 3. Figure 4 shows a 32x16 stack implemented using a 32x16 LogiCore dual-port RAM controlled by a stack controller. The stack controller implements the stack as a traditional stack with push and pop instructions including full and empty flags. When *push* is '1' and *pop* is '0', the stack pushes the value at *d(15:0)* to the write address, *wr_addr,* the memory address that represents the next empty location in memory. Both *wr_addr* and the read address, *rd_addr*, are simultaneously decremented. After the operation is complete, the output *q(15:0)* contains the value on top of the stack. When *pop* is '1' and *push* is '0', both the read and write addresses are incremented. Unlike a traditional stack, when both *pop* and *push* are '1', the top element is replaced with *d(15:0)* without pushing the stack.

Table 1 FC16 Data Stack Operations

Opcode	Name	Function
0000	NOP	No operation
0001	DUP	Duplicate T and push data stack. N <= T; N2 <= N
0002	SWAP	Exchange T and N. T <= N; N <= T
0003	DROP	Drop T and pop data stack. T <= N; N <= N2
0004	OVER	Duplicate N into T and push data stack. T <= N; N <= T; N2 <= N
0005	ROT	Rotate top 3 elements on stack clockwise. T <= N2; N <= T; N2 <= N
0006	-ROT	Rotate top 3 elements on stack counter-clockwise. T <= N; N <= N2; N2 <= T
0007	NIP	Drop N and pop rest of data stack. T is unchanged. N <= N2
0008	TUCK	Duplicate T into N and push rest of data stack. N2 <= T
0009	ROT_DROP	Drop N2 and pop rest of data stack. T and N are unchanged. Equivalent to ROT DROP
000A	ROT_DROP_SWAP	Drop N2 and pop rest of data stack. T and N are exchanged. Equivalent to ROT DROP SWAP

The FC16 data stack shown in Figure 3 consists of two 16-bit registers for the top and second elements of the data stack followed by the modified 32x16 stack shown in Figure 4. 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 instructions involving the top three stack elements.

The input to the top register, *Treg*, can be from one of eight possible sources using the 8-to-1 multiplexer shown in Figure 1. The input to the second element in the register stack, *Nreg*, can be from either *Treg*, one of the outputs from the function unit, *y1*, or the top of the modified stack, *stack32x16*. The data stack instructions operate at most on the top three elements of the data stack. The modifications to *stack32x16* described above are necessary to support operations involving the third stack element. The instruction *ROT*, for example, moves the value in *Treg* to *Nreg*, the value in

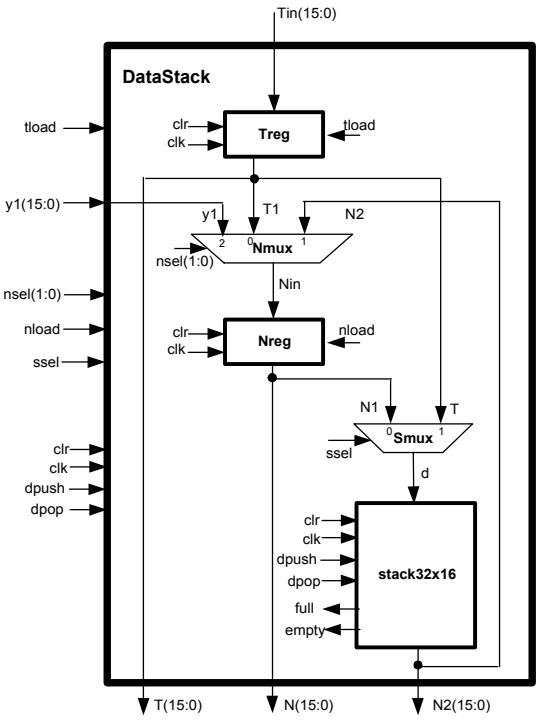

Figure 3 The data stack

Nreg to the top of the *stack32x16* (the third element in the data stack) and the value on the top of the *stack32x16* to *Treg*. This has the effect of rotating the top three elements of the data stack. For this instruction, *T1* is multiplexed into *Nreg*, *N* is multiplexed into the *stack32x16*, and *N2* is externally multiplexed into *Treg*. In this case the top of the modified stack is replaced with the value in *Nreg* without pushing or popping the stack. To allow this operation, *wr2_addr* is multiplexed between *wr_addr*, the next available empty space in memory for writing and *rd_addr*, the present top of the stack, shown in Figure 4. The FC16 data stack can execute all stack operations listed in Table 1 in a single clock cycle while using chip real estate efficiently.

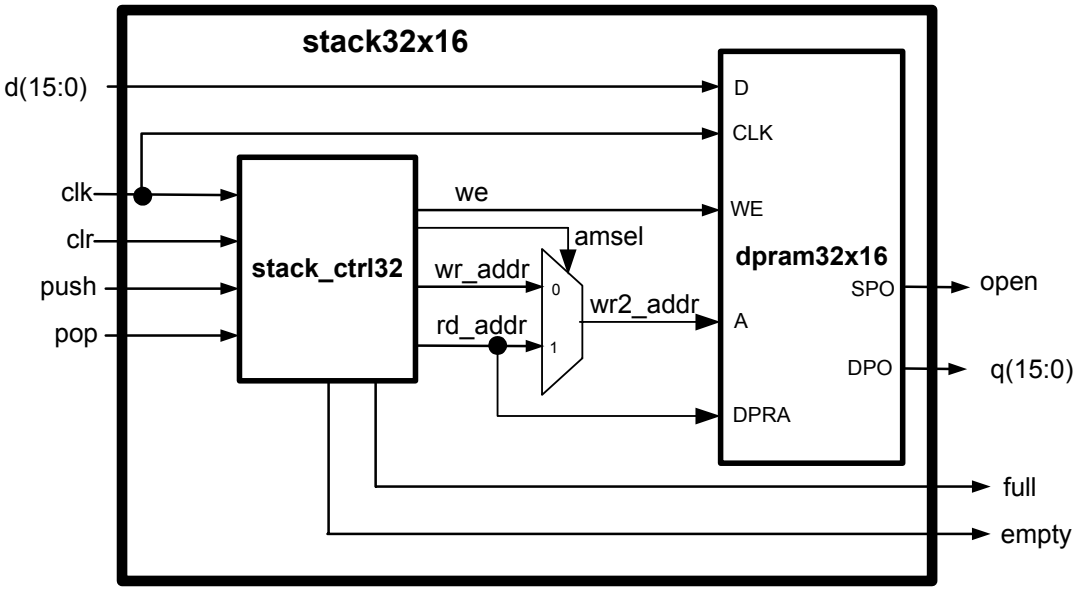

Figure 4 Stack created by a dual-port RAM

4. The Function Unit

The function unit performs arithmetic, logical, shifting, and relational operations on the top elements of the data stack. *T, N,* and *N2*, the top three elements of the data stack, respectively, and a 6-bit function selection signal, *Fcode*, are inputs to the function unit. Table 2 shows the instructions for the function unit. The function unit has two 16-bit outputs *y(15:0)* and *y1(15:0)* as shown in Figure 1. The primary output, *y*, is multiplexed into the top of the data stack for performing unary and binary operations. For operations having answers larger than 16 bits, such as multiplication or division, *y1* is input into the data stack and multiplexed into *Nreg* so that *Treg:Nreg* will contain the 32 bit answer.

The arithmetic and logical operations operate on the top elements of the stack and output the result to be placed on top of the stack. The shifting operations operate on values from the top of the stack. The relational operators output X“FFFF” or X”0000” if the top two elements of the stack are or are not accordingly related, respectively. Among these instructions are two instructions *MPP* and *SHLDC* for implementing multiplication and division, respectively. Listing 1 contains a program that will multiply the value in *Treg* by the value in *Nreg* and place the 32-bit product in *Treg:Nreg* in 19 clock cycles using partial product multiplication, *MPP*. Listing 2 contains a program that will divide a 32 bit numerator located in *N2:N* by a 16 bit denominator located in *T* and place the quotient in *Treg* and the remainder in *Nreg* in 18 clock cycles. The FC16 executes all of the instructions in Table 2 in a single clock cycle.

Table 2 Instructions for the FC16 Function Unit

Opcode	Name	Function
0010	+	Pop N and add it to T
0011	-	Pop T and subtract it from N
0012	1+	Add 1 to T
0013	1-	Subtract 1 from T
0014	INVERT	Complement all bits of T
0015	AND	Pop N1 and AND it to T
0016	OR	Pop N1 and AND it to T
0017	XOR	Pop N1 and AND it to T
0018	2*	Logic shift left T
0019	U2/	Logic shift right T
001A	2/	Arithmetic shift right T
001B	RSHIFT	Pop T and shift N1 T bits to the right
001C	LSHIFT	Pop T and shift N1 T bits to the left
001D	mpp	multiply partial product (used for multiplication)
001E	shldc	shift left and decrement conditionally (used for division)
0020	TRUE	Set all bits in T to '1'
0021	FALSE	Clear all bits in T to '0'
0022	NOT 0=	TRUE if all bits in T are '0'
0023	0<	TRUE if sign bit of T is '1'
0024	U>	T <= TRUE if N > T (unsigned), else T <= FALSE
0025	U<	T <= TRUE if N < T (unsigned), else T <= FALSE
0026	=	T <= TRUE if N = T, else T <= FALSE
0027	U>=	T <= TRUE if N >= T (unsigned), else T <= FALSE
0028	U<=	T <= TRUE if N1 <= T (unsigned), else T <= FALSE
0029	<>	T <= TRUE if N /= T, else T <= FALSE
002A	>	T <= TRUE if N1 > T (signed), else T <= FALSE
002B	<	T <= TRUE if N1 < T (signed), else T <= FALSE
002C	>=	T <= TRUE if N1 >= T (signed), else T <= FALSE
002D	<=	T <= TRUE if N1 <= T (signed), else T <= FALSE

Listing 1 Unsigned Multiplication

```
LIT X"0000"
MPP MPP MPP MPP MPP MPP MPP MPP
MPP MPP MPP MPP MPP MPP MPP MPP
ROT_DROP
```

Listing 2 Unsigned Division

```
-ROT
SHLDC SHLDC SHLDC SHLDC SHLDC SHLDC SHLDC SHLDC
SHLDC SHLDC SHLDC SHLDC SHLDC SHLDC SHLDC SHLDC
ROT_DROP_SWAP
```

5. The Return Stack

The FC16 return stack, shown in Figure 5, is a modified 32x16 stack made from a *stack32x16* described in Section 3, and a single register, *R*. Table 3 shows the return stack instructions. The *R* register serves as a 'false top' of the return stack with multiplexed inputs and the option to decrement the registered output. The instruction *DRJNE* decrements the value on the top of the return stack and jumps to an address in memory if the value is not equal to zero. If the top of the return stack is equal to zero, execution proceeds to the next valid instruction in the program. This instruction is used to implement the *NEXT* in a *FOR…NEXT* loop.

The top-of-stack output, *R(15:0)*, is multiplexed to the top of the data stack and to the program counter, *PC,* as shown in Figure 1. The input to the return stack can be either the top of the data stack or

the program counter plus one. These inputs make it possible to push values from the data stack to the return stack and to push the return address of a subroutine call. The *RET* instruction at the end of a subroutine pops the address from the return stack into the program counter.

Table 3 FC16 Return Stack Operations

Opcode	Name	Function
0030	>R	"To-R" Pop T and push it on return stack
0031	R>	"R-from" Pop return stack R and push it into T
0032	R@	"R-fetch" Copy R to T and push register stack
0033	R>DROP	"R-from-drop" Pop return stack R and throw it away
0103	DRJNE	Decrement R and jump if R is not zero
0104	CALL (:)	Call subroutine (colon)
0105	RET (;)	Subroutine return (semi-colon)

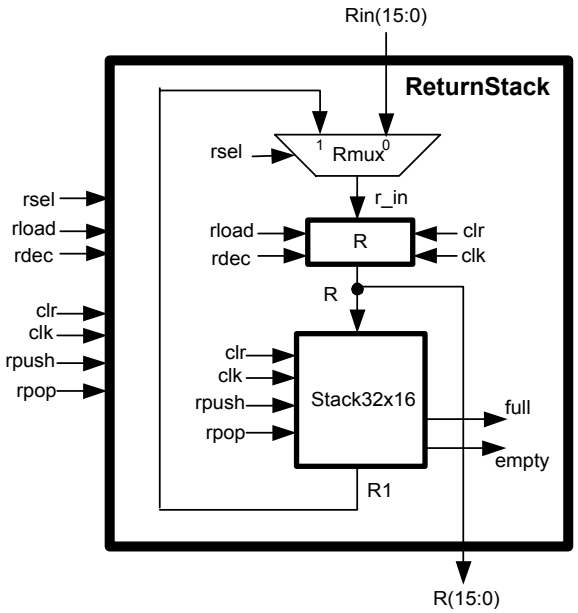

Figure 5 The return stack

6. The Controller

The module *FC16_control* shown in Figure 1 is a control unit implemented as a Mealy state machine. This state machine has three states: *Fetch*, *Execute*, and *Execute-Fetch*. Figure 6 shows the state-transition diagram for the controller. This controller begins in the fetch state to 'fetch' the next instruction from the external program ROM. If the instruction requires only a single clock cycle to execute, the current instruction is executed and the next instruction is read from the program ROM in the Execute-Fetch state. The instructions continue to be executed and fetched at the same time an instruction that requires more than one clock cycle is fetched. Instructions with inline data or addresses, for example, are two clock-cycle instructions, one to execute the instruction and one to fetch the next instruction while ignoring the inline information.

These multi-cycle instructions have been assigned opcodes with a '1' in the 8th bit position. For instructions requiring multiple clock cycles, the controller executes the current instruction without 'fetching' the next word from the program ROM. Following the last clock cycle the controller returns to the fetch state to 'fetch' the next instruction.

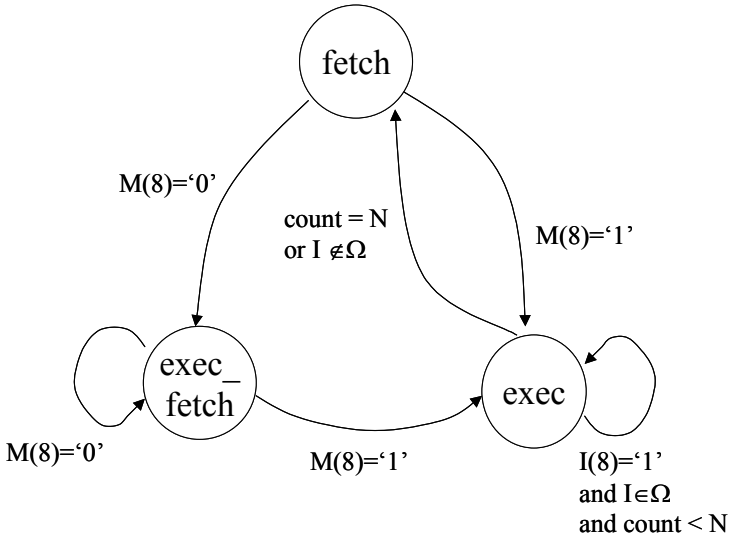

Figure 6 State diagram for the controller

For the instructions related to the function unit, the six least significant bits of the instruction corresponds directly to the function select signal, *Fcode(5:0)*. The controller sets the *load*, *push*, and *pop* data stack signals appropriately to perform arithmetic, relational, logical, or shifting operations on the top elements of the data stack. In the same clock cycle the result, output by the function unit, is placed on top of the data stack and the operands are removed. Instructions that have not been introduced yet in this paper are given in Table 4.

Table 4 Other Instructions

Opcode	Name	Function	Number of Clock Cycles
0034	@	Fetch the byte at address T in RAM and load it into T	1
0036	ROM@	Fetch the byte at address T in ROM and load it into T	1
0037	S@	Fetch the 8-bit byte from Port S and load it into T	1
0038	DIO2@	Fetch the 8-bit byte from the DIO2 data bus and load it into T.	1
0039	DIO2!	Store the byte in N at the DIO2 address in T. Pop both T and N	3
0100	LIT	Load inline literal to T and push data stack	2
0101	JMP	Jump to inline address	2
0102	JZ	Jump if all bits in T are '0' and pop T	2
0106	JB1LO	Jump if input pin B1 is LO	2
0107	JB2LO	Jump if input pin B2 is LO	2
0108	JB3LO	Jump if input pin B3 is LO	2
0109	JB4LO	Jump if input pin B4 is LO	2
010A	JB1HI	Jump if input pin B1 is HI	2
010B	JB2HI	Jump if input pin B1 is HI	2
010C	JB3HI	Jump if input pin B1 is HI	2
010D	JB4HI	Jump if input pin B1 is HI	2
010E	RAMSTORE	Store the byte in N at the address in T. Pop both T and N	2

For branching instructions, the program counter, *PC* is loaded with the inline address, *M*. Most multiple cycle instructions execute on the first clock cycle and fetch the next instruction to avoid executing inline information as an instruction. In some cases, multiple clock-cycle instructions will require multiple cycles of execution. For example, the instruction, *RAMSTORE*, requires two clock cycles. In the first clock cycle, the data in *Nreg* is stored in the external RAM at the address in *Treg* and the address in *Treg* is popped from the data stack. During the second clock cycle, the leftover data which is now in *Treg* is popped from the data stack. Instructions of this type, denoted as belonging to set Ω in Figure 6, remain in the *Execute* state with different control outputs for each clock cycle as necessary. This implementation easily extends the FC16 to support N-clock cycle instructions.

7. Programming Example

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 [8]. 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.

We will illustrate this process by writing a Forth program to access the buttons, LEDs, and 7-segment displays on the Digilent DIO2 peripheral board [2]. Figure 7a shows a block diagram of the DIO2 peripheral board that connects to the D2 prototyping board shown in Figure 7b via connectors A and B.

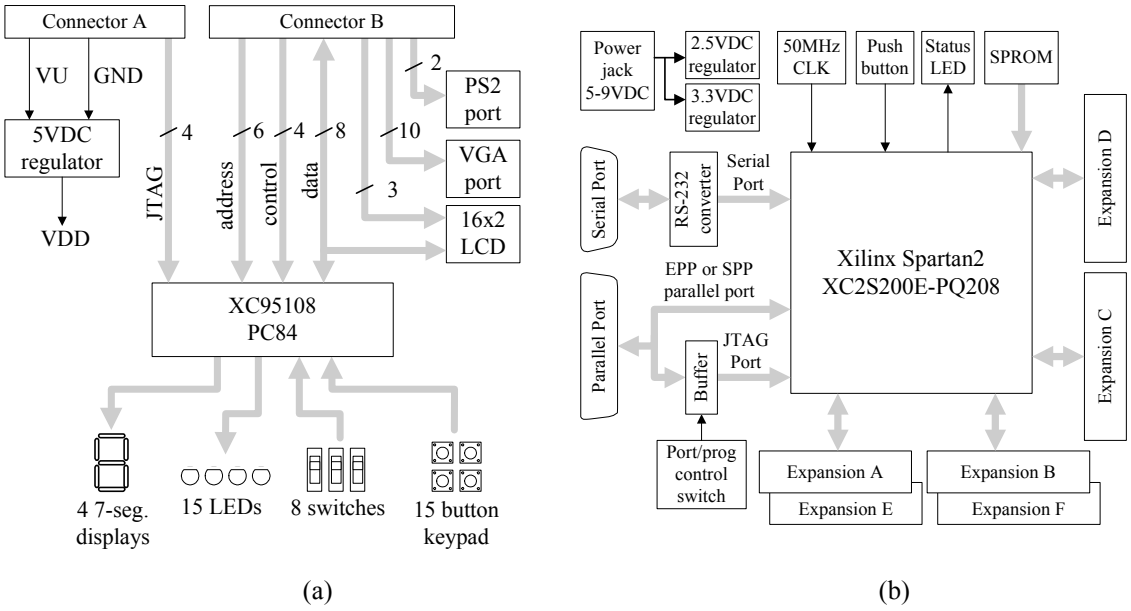

Figure 7: Block diagrams of (a) the DIO2 Peripheral Board and (b) the D2 development board (Courtesy of Digilent, Inc.)

Since the DIO2 peripheral board offers interfaces to many input and output devices, a system bus is used to interface between the Spartan II and the devices. The system bus is programmed on a Xilinx 95108 CPLD. Note that the Spartan II in Figure 7b will communicate with the XC95108 in Figure 7a using a 6-bit address bus, an 8-bit bidirectional data bus, and 4 control signals. These control signals include *cclk* (for scanning the four 7-segment displays), *cs* (chip select), *oe* (output enable), and *we* (write enable).

Table 5 shows how to access the buttons, switches, leds, and 7-segment displays on the DIO2 peripheral board. The *oe* signal controls the direction of the data bus. Writing data to the CPLD occurs on the falling edge of *we*. The 16-bit value in *sseg_reg(15:0)* will automatically be displayed as a 4-digit hex value on the 7-segment displays by the hardware in the CPLD. These four 7-segment displays are "refreshed" at the *cclk* rate. For example, a 190 Hz clock counted down from the 50 MHz clock on the D2 board works well.

A test program that demonstrates how to access the buttons, LEDs, and 7-segment displays on the DIO2 board is shown in Listing 3. As a matter of notation, the parentheses following a Forth word shows the data stack contents before and after the word is executed in the form (*before* -- *after*). In each case the top of the stack is on the right. The word *D2DIG! (n --)* takes the 16-bit value, *n*, in *T* and stores the high byte at the DIO2 address 7 and the low byte at the DIO2 address 6. As shown in Table 5 this will store *n* in the DIO2 *sseg_reg* which will then display the corresponding four hex digits on the four 7-segment displays. Note that the word *D2DIG! (n --)* expects *n* on the top of the stack and then pops this value when the word is executed. In a similar way the word *D2LD! (n --)* will display the 16-bit value in *T* on

the 16 LEDs (see Table 5) and then pop *T*. The right-most LED displays *T(15)* and the left-most LED displays *T(0)*.

Table 5 Accessing the DIO2 Peripheral Board

cs	oe	we	addr(5:0)	data(7:0)
1	1	0	xxxx00	btns(7:0)
1	1	0	xxxx01	'0'&btns(14:8)
1	1	0	xxxx1x	switchs
1	0	↓	000100	leds(7:0)
1	0	↓	000101	leds(15:8)
1	0	↓	000110	sseg_reg(7:0)
1	0	↓	000111	sseg_reg(15:8)

Listing 3 DIO2 buttons, LEDs, and 7-seg displays

```
\ Test of DIO2 buttons, LEDs, and 7-seg displays

: D2DIG!        ( n -- )              \ Display n on 7-segment displays
                DUP 8 RSHIFT                  \ n nHI
                7 DIO2!                       \ display nHI
                6 DIO2! ;                     \ display nLO

: D2LD!         ( n -- )              \ Display n on the 16 LEDs
                DUP 8 RSHIFT                  \ n nHI
                5 DIO2!                       \ display nHI
                4 DIO2! ;                     \ display nLO

: get.BTN2      ( -- n )              \ Push 15-button bit mask to T
                1 DIO2@                       \ btns(15:8)
                8 LSHIFT
                0 DIO2@                       \ btns(7:0)
                OR ;

: waitBTN2        ( -- n)             \ Wait to push a button and get mask
                  BEGIN                       \ wait to lift finger
                    get.BTN2 0=
                  UNTIL
                  BEGIN                       \ wait to press button
                    get.BTN2
                  UNTIL
                  get.BTN2      ;             \  get buttons

: but>num   ( n1 -- n2 )              \ convert button bit mask to button no.
                15 FOR                        \ loop 15 times
                   DUP 1 =
                   IF                         \ value matches
                        R>                    \ get loop value
                        15 SWAP -             \ find index
                        1 >R                  \ break out of loop
                   ELSE
                        U2/                   \ Shift button value
                   THEN
                NEXT
                NIP ;                         \ remove extra 1 from N

: main  ( -- )                        \ main program
                BEGIN
                  waitBTN2                    \ wait to push BTN2
                  DUP D2LD!                   \ display on LEDs
                  but>num                     \ find button number
                  D2DIG!                      \ display on 7-seg display
                AGAIN ;
```

The word *get.BTN2 (-- n)* first reads *btns(15:8)* using *DIO2@*, shifts this value 8 bits to the left, then reads *btns(7:0*) and *ORs* it with the shifted *btns(15:8)* to produce a single 16-bit value, *n*, that contains *btns(15:0)*. This value, *n*, is pushed into T.

The word *waitBTN2 (-- n)* will first wait to make sure no button is being pressed, then wait until any button is pressed, and finally read that button using *get.BTN2*. Note that the value left on the data stack is a 16-bit value with a single bit set corresponding to the button pressed. The word *but>num (n1 – n2)* can be used to convert this value, *n1*, into a button number, *n2*, between 0 and 15. This word loops through all 15 bit positions and uses the fact that the loop counter in a *FOR…NEXT* loop is kept on the top of the return stack.

The main program in Listing 3 waits for a button to be pressed, displays the bit position of the button on the LEDs, and displays the button number on the 7-segment displays. The *BEGIN…AGAIN* loop causes this process to continue indefinitely.

The Forth program in Listing 3 is compiled to the VHDL code shown in Listing 4 using a compiler adapted from a C++ program described in [8]. The VHDL code shown in Listing 4 is the contents of the program ROM shown in Figure 2.

Note that the FC16 program in Listing 4 begins with a jump to the main program at address 47. The main word is always the last word defined in a Forth program such that the addresses of all previously defined words will be known.

The Forth *IF* statement is compiled to the FC16 *JZ* instruction, which jumps if the flag in *T* is false. The *JZ* instruction is also used for the Forth *WHILE* and *UNTIL* instructions. The Forth *ELSE* statement is compiled to the FC16 *JMP* instruction, as is the Forth word *AGAIN*.

The FC16 core is implemented in VHDL in a junior engineering course at Oakland University. Students then use this core to implement a project of their choosing. A partial list of student projects done in the past year is given in Table 6.

Table 6 Partial list of student projects that use the FC16 core

A Decimal Calculator
A Morse Code Transceiver
A Digital Fan Controller
Encryption and Decryption Engine
A Text Pad using the VGA Monitor
Venom8y VGA Video Game
Home Security System
Chess AI
Music Generator and Display
Hex Education
The Number Matching Game
Digital Battleship
Guess the Number
Slot Machine
Simon Game in Reverse
Mind Teaser

Listing 4 VHDL code generated from the Forth program in Listing 3

```
type rom_array is array (NATURAL range <>)
of STD_LOGIC_VECTOR (15 downto 0);
constant rom: rom_array := (
       JMP,           --0
       X"0047",       --1
-- D2DIG!
       dup,           --2
       LIT,           --3
       X"0008",       --4
       rshift,        --5
       LIT,           --6
       X"0007",       --7
       DIO2store,     --8
       LIT,           --9
       X"0006",       --a
       DIO2store,     --b
       RET,           --c
-- D2LD!
       dup,           --d
       LIT,           --e
       X"0008",       --f
       rshift,        --10
       LIT,           --11
       X"0005",       --12
       DIO2store,     --13
       LIT,           --14
       X"0004",       --15
       DIO2store,     --16
       RET,           --17
-- get.BTN2
       LIT,           --18
       X"0001",       --19
       DIO2fetch,     --1a
       LIT,           --1b
       X"0008",       --1c
       lshift,        --1d
       LIT,           --1e
       X"0000",       --1f
       DIO2fetch,     --20
       orr,           --21
       RET,           --22
-- waitBTN2
       CALL,          --23
       X"0018",       --24
       zeroequal,     --25
       JZ,            --26
       X"0023",       --27
       CALL,          --28
       X"0018",       --29
       JZ,            --2a
       X"0028",       --2b
       CALL,          --2c
       X"0018",       --2d
       RET,           --2e
```

```
-- btn>num
       LIT,           --2f
       X"000f",       --30
       tor,           --31
       dup,           --32
       LIT,           --33
       X"0001",       --34
       eq,            --35
       JZ,            --36
       X"0042",       --37
       rfrom,         --38
       LIT,           --39
       X"000f",       --3a
       swap,          --3b
       minus,         --3c
       LIT,           --3d
       X"0001",       --3e
       tor,           --3f
       JMP,           --40
       X"0043",       --41
       u2slash,       --42
       drjne,         --43
       X"0032",       --44
       nip,           --45
       RET,           --46
-- main
       CALL,          --47
       X"0023",       --48
       dup,           --49
       CALL,          --4a
       X"000d",       --4b
       CALL,          --4c
       X"002f",       --4d
       CALL,          --4e
       X"0002",       --4f
       JMP,           --50
       X"0047",       --51
       X"0000"        --52
       );
```

8. Experimental Results

Experiments were conducted to measure the speed of a program running on the FC16 Forth core in a Xilinx Spartan IIE FPGA compared with the same program running on a Motorola 68HC12 microcontroller. A Forth program implementing the Sieve of Eratosthenes [3, 4], which found the 308 prime numbers between 3 and 2039, was run on the FC16 Forth core in a Xilinx Spartan IIE (x2s200e) FPGA. The exact same Forth program was also run on a Motorola 68HC12 microcontroller using the subroutine-threaded Forth language WHYP described in [8]. The identical algorithm was also written in C and compiled to tight HC12 assembly language code using the ImageCraft HC12 ANSI C Tools V6.15A [14]. In each of these three cases a counter was used to directly measure the number of clock cycles used to execute the program. The results are summarized in Table 7. Note that the FC16 Forth core is over 6 times faster than an equivalent C-assembly language implementation and nearly 30 times faster than the same Forth program running on an HC12 microcontroller. Table 7 shows the actual time required to execute the program using an 8 MHz clock on the HC12. The FC16 Forth core was actually run at 25 MHz on the Xilinx FPGA as indicated in Table 7. The FC16 Forth core is available as an EDIF file at *www.tigs.com/fc16*.

Table 7 Relative speed of FPGA Forth core

Experiment	#clock cycles	Relative Speed				Time (ms) @8MHz	Time (ms) @25MHz
FC16 - FPGA	60,299	1	0.034	0.151	8.83	7.54	2.41
Forth – HC12	1,763,369	29.244	1	4.419	258.26	220.42	----
C – HC12	399,031	6.618	0.226	1	58.44	49.88	----
FPGA-VHDL	6,828	0.113	0.004	0.017	1	0.85	0.273

In addition to these experiments a direct hardware implementation of the Sieve of Eratosthenes algorithm was developed on the Xilinx Spartan IIE FPGA using VHDL. This implementation included a datapath containing two registers, three counters, an adder, a shift-adder, two comparators, a multiplexer, and a 1024 x 1 block RAM module. A state machine containing 8 states that implement the Sieve of Eratosthenes algorithm controlled the datapath. The result of this implementation is shown in the last row of Table 7. Not only is it faster than the FC16 Forth core by nearly a factor of 9, but it also uses only 94 CLB slices (3%) of the FPGA compared with 734 CLB slices (31%) for the FC16 Forth core implementation. This direct hardware implementation was also run at 25 MHz as indicated in Table 7.

This confirms that a hardware only implementation will generally run faster and take up less space than using a processor core that executes a software program. The reason is that only those instructions and hardware actually needed for the algorithm need to be implemented. On the other hand a processor core will have implemented a complete set of instructions, only a fraction of which are needed for any particular algorithm. The advantage of a processor core is ease of implementing an algorithm at the expense of speed and area. Thus, processor cores are particularly useful for rapid prototyping [11].

9. Summary

The FC16 is a high-performance Forth core that has been implemented on a Xilinx Spartan II FPGA. Of the 63 Forth instructions that have been implemented, 51 of them execute in a single clock cycle. Forth has always been an extensible language in the sense that the programmer defines new words that get added to the dictionary and essentially become a part of the language. With the development of this flexible Forth core that is used in an FPGA, the Forth hardware has also become extensible. It is an easy matter for the user to add new hardware instructions that will perform specific operations on new hardware I/O modules. For example, it is possible to make the top of stack, *T*, a shift register that could interface to external serial devices using the standard SPI interface [9]. By including a timer module, the FC16 could become a very useful high-performance, low-cost microcontroller. Adding a UART would allow interactive communication with the FC16 through a standard asynchronous serial line. Many of these types of modules are available as precompiled LogiCore modules. Many projects have been completed using the FC16 Forth core, which illustrate the effectiveness of Forth as a rapid prototyping tool.

Experiments show that an identical Forth program for the Sieve of Eratosthenes executes nearly 30 times faster on the FPGA Forth core than on a 68HC12 microcontroller at the same clock speed. This same program executes over 6 times faster on the FPGA Forth core than an equivalent compiled C program run on the same 68HC12. A direct hardware implementation of the Sieve of Eratosthenes algorithm runs nearly 9 times faster than on the FC16 Forth core.

10. References

1. Ashenden, P. J., *The Designer's Guide to VHDL*, Morgan Kaufmann, San Francisco, 1996.
2. Digilent, Inc., *http://www.digilentinc.com*.
3. Geere, R., *Forth: The NEXT Step*, Addison-Wesley Publishing Co., Reading, MA, 1986.
4. Gilbreath, J. and Gilbreath, G., BYTE Magazine, p. 283, Jan. 1983.
5. Golden, J., Moore, C. H., and Brodie, L., "Fast Processor Chip Takes Its Instructions Directly from Forth," *Electronic Design*, March 21, 1985, pp. 127-138.
6. Hand, T., "The Harris RTX 2000 Microcontroller," *Journal of Forth Application and Research*, Vol. 6, No. 1, pp. 5-13, 1990.
7. Haskell, R. E., "WHYP on a Chip—A VHDL Model for a High-Performance Forth Engine," Technical Report No. 9911-1, CSE Dept., Oakland University, Rochester, MI, Nov. 1999.
8. Haskell, R. E., *Design of Embedded Systems Using 68HC12/11 Microcontrollers*, Prentice Hall, Upper Saddle River, NJ, 2000.
9. Haskell, R. E., and D. M. Hanna, "Implementing a Forth Engine Microcontroller on a Xilinx FPGA," Looking Forward – The IEEE Computer Society's Student Newsletter (A Supplement to Computer), Vol. 8, No. 1, Spring 2000.
10. Haskell, R. E. and D. M. Hanna, "An elastic microprocessor core for Xilinx FPGAs," Second IEEE Electro/Information Technology Conference, Oakland University, Rochester, MI, June 7-9, 2001.
11. Haskell, R. E. and D. M. Hanna, "Rapid Prototyping using a Microprocessor Core on a Spartan II FPGA," Proc. 2003 International Conference on Embedded Systems and Applications, Las Vegas, NV, pp. 49-55, June 23-26, 2003.
12. Hayes, J. R., Fraeman, M.E., Williams, R. L., and Zaremba, T., "A 32-Bit Forth Microprocessor," *Journal of Forth Application and Research*, Vol. 5, No. 1, pp. 39-48, 1987.
13. Hayes, J. and Lee, S., "The Architecture of the SC32 Forth Engine," *Journal of Forth Application and Research*, Vol. 5, No. 4, pp. 49-71, 1989.
14. ImageCraft, *http://www.imagecraft.com/software/*.
15. Koopman, Jr., P., "32-Bit RTX Chip Prototype," *Journal of Forth Application and Research*, Vol. 5, No. 2, pp. 331-335, 1988.
16. Koopman, Jr., P., *Stack Computers – the new wave*, Originally published by Ellis Horwood in 1989, now available on *http://www-2.cs.cmu.edu/~koopman/stack.html*.
17. Moore, C., "ShBoom on ShBoom: A Microcosm of Software and Hardware Tools," Proc. 1990 Rochester Forth Conference, pp. 21-27, June 12-15, 1990.
18. Ting, C. H., "P Series of Microprocessors," in *More on Forth Engines*, Vol. 22, pp. 1-17, Sept. 1997.
19. Ting, C. H., "P16 Microprocessor Design in VHDL," in *More on Forth Engines*, Vol. 22, pp. 44-51, Sept. 1997.