An Elastic Microprocessor Core for Xilinx FPGAs

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

Abstract

An embedded system based on the use of an FPGA such as the Xilinx Spartan or 4000 series FPGA is typically designed using a hardware description language such as VHDL. On the other hand an embedded system based on the use of a microcontroller such as the Motorola 68HC12 is typically designed by writing software in either assembly language or a high-level language such as Forth or C. When an embedded system contains both a microprocessor and an FPGA a decision must be made on how to partition the system into its hardware and software components. This decision can have a profound effect on the overall system cost. The longer the irrevocable decision of how to partition the hardware and software can be delayed, the better is the chance to keep overall system cost to a minimum. This paper describes the design of a microprocessor core that has been implemented in a Xilinx FPGA. Software programs written in Forth can be compiled to VHDL code that uses the minimum amount of hardware to implement the microprocessor core to run the program. The entire embedded system, including both the microprocessor core and additional hardware, is implemented using VHDL, allowing a unified approach to design, simulation, and testing.

1. Introduction

A typical embedded system contains a microprocessor and possibly an FPGA. Microcontrollers such as the Motorola 68HC12 contain integrated modules for implementing parallel and serial I/O, A/D conversion, and various timing functions [1]. While these functions are convenient to have available, they may not provide all the hardware functions needed for a particular design. Such a microcontroller will certainly include some functions that are not actually used in a particular design.

As the size 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.

VHDL is used to design all the hardware, including the microprocessor core, that is synthesized to the FPGA. The microprocessor core is a stack-based computer that will efficiently execute Forth code. A program written in Forth can be translated to a VHDL program that will synthesize only the hardware necessary to implement the particular Forth program. Because the Forth software program ends up as just more VHDL code that can be simulated and synthesized with the rest of the hardware design, the boundary between hardware and software has become almost entirely obliterated. This has the advantage of delaying (or avoiding) the

hardware/software partition decision. This is possible because the same person is designing both the hardware and software as a unified whole. Changes can be made at any point in the design process as simulations and synthesis tests provide information about speed and area tradeoffs.

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. In this sense they are very object-oriented. We send words parameters on the data stack and ask the words to execute themselves and send us the answers back on the data stack. We really don't care how the word does it -- once we have written it and tested it so we know that it works.

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 [1]. 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 what is used to go from one WHYP word to the next. In other words, WHYP words are just regular 68HC12 subroutines.

Inasmuch as Forth (and WHYP) programs consist of sequence of words, the most often executed instruction is a call to the next word. This means executing the inner interpreter (NEXT) in traditional Forths, or calling a subroutine in WHYP. Up to 25% of the execution time of a typical Forth program is used up in calling the next word. 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 [2]. 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 [3]. Similar 32-bit Forth engines were also developed [4-6]. 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 [7]. Later Chuck Moore and C. H. Ting designed the MuP21 that has been described by Ting [8-9]. In 1999 the author designed the W8X microcontroller [10] that was based on ideas developed in these early Forth engines. It was designed using VHDL [11] and has been implemented in a Xilinx FPGA by students in a junior-level course at Oakland University [12].

This paper describes an enhanced W8X, the W8Z, that will fit in a low-cost Xilinx FPGA and can be used as a microprocessor core for a high-performance embedded system using only a Xilinx FPGA. It contains over 40 ANSII standard Forth words, all of which execute in a single clock cycle. It also contains a number of specialized instructions that make it easy to interface to the Digilab XL development board [13].

2. The W8Z Microcontroller

The W8Z is a high-performance microprocessor that can be implemented to perform useful functions on a Xilinx Spartan series FPGA. The overall structure of the W8Z is shown in Figure 1. The data busses in this figure are 8 bits wide and each instruction contains 8 bits. The heart of the W8Z is a register stack, *reg_stack*, that contains four registers interconnected in a complete crossbar fashion. The function unit, *Funit*, implements 32 arithmetic, logical, Boolean, and shifting operations. Software programs are stored in a 16-bit program ROM, *Wrom*, with two separate outputs containing the upper byte and lower byte of the ROM word. This word can contain either two instructions or a jump instruction plus the jump address. This technique allows all instructions, including jump instructions, to be executed in a single clock cycle.

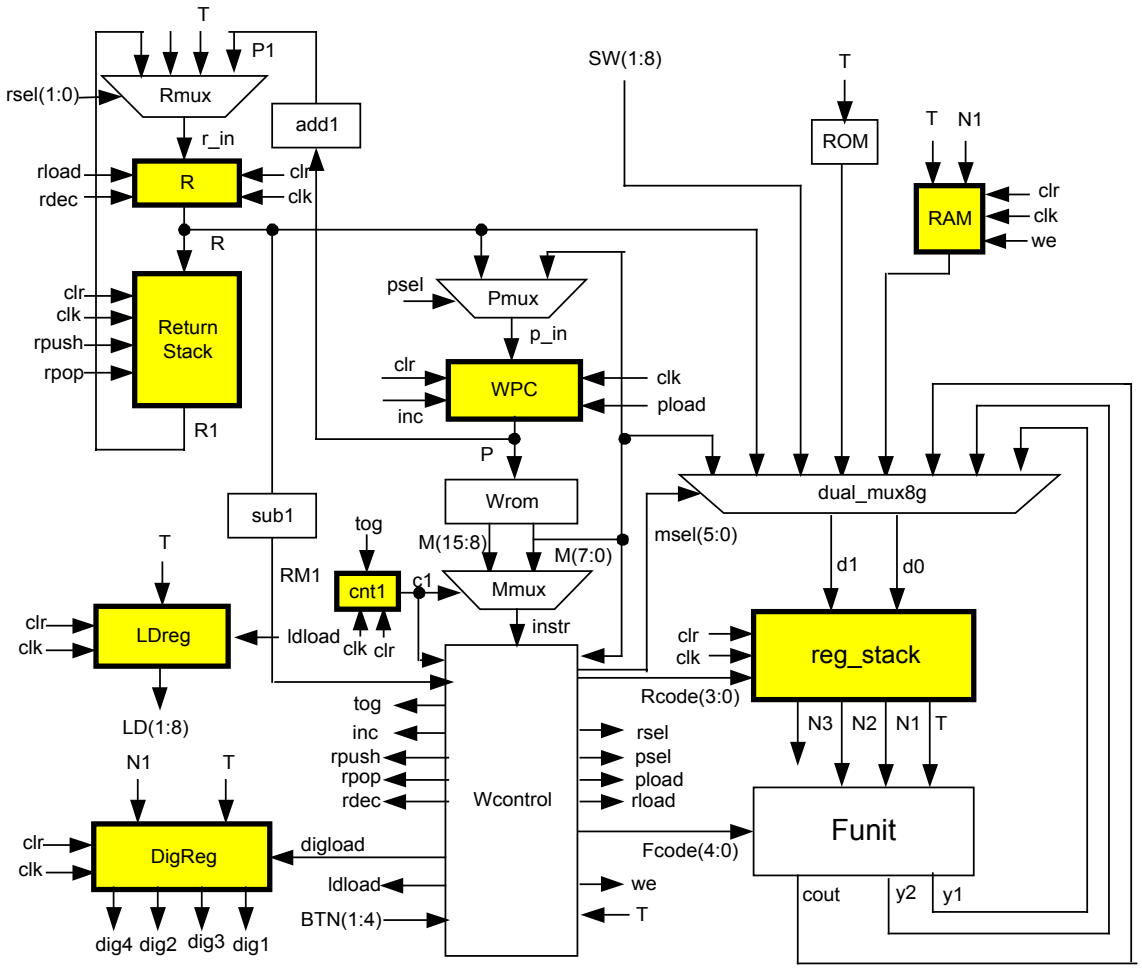

Figure 1 The W8Z Microprocessor Core

The program ROM address comes from the program counter, *WPC*, that can be incremented by 1 when *inc* = '1'. The program counter is loaded from the output of the multiplexer, *Pmux*, when *pload* = '1'.

The control unit, *Wcontrol*, is entirely combinational. This eliminates the need for a separate instruction register. The input to the control unit is the 8-bit instruction, *instr*, from the multiplexer, *Mmux*, that selects either the upper or lower byte or the *Wrom* output. The select input, *c1*, to *Mmux* is the output of a 1-bit counter that toggles between '0' and '1' when *tog* = '1'.

A return stack is used to hold return addresses during subroutine calls. The top of the return stack is held in the register, *R*, which is decremented by 1 when *rdec* = '1'. The rest of the return stack is implemented as a LogiBLOX dual-port RAM. A byte is pushed onto this return stack when *rpush* = '1' and is popped from the return stack when *rpop* = '1'.

The input to the top of the register stack is *d0* and can come from one of eight possible sources through a dual 8-to-1 multiplexer. The input to the second element in the register stack is *d1* and can also come from one of eight possible sources through the same dual 8-to-1 multiplexer. The *d0* output of this multiplexer is controlled by *msel(2:0)* and the *d1* output of the multiplexer is controlled by *msel(5:3)*.

3. The Register Stack

The heart of the W8Z is the register stack, *reg_stack*, shown in Figure 2. The register stack consists of four 8-bit registers that are completely interconnected using four 4-to-1 multiplexers as shown in Figure 2.

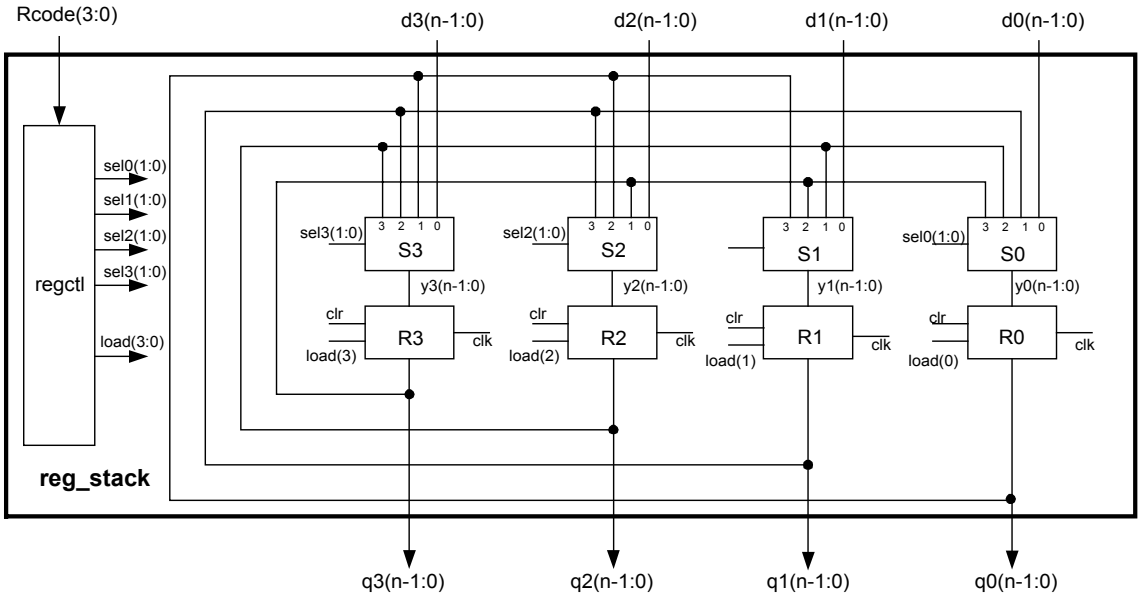

Figure 2 The Register Stack

The controller, *regctl*, in the register stack has a 4-bit input, *Rcode(3:0)*, and produces the control signals *load(3:0), sel0(1:0), sel1(1:0), sel2(1:0),* and *sel3(1:0)* to implement the 16 W8Z instructions shown in Table 1. Note that the value of *Rcode(3:0)* is the same as the lower 4 bits of the W8Z opcode for each of the 16 instructions given in Table 1.

Table 1 -- Register Instructions

Hex Opcode	Rcode(3:0)	Name	Function
20	0	NOP	No operation
21	1	DUP	Duplicate T and push data stack. N1 <= T; N2 <= N1; N3 <= N2;
22	2	SWAP	Exchange T and N1. T <= N1; N1 <= T;
23	3	DROP	Drop T and pop data stack. T <= N1; N1 <= N2; N2 <= N3;
24	4	OVER	Duplicate N1 into T and push data stack. T <= N1; N1 <= T; N2 <= N1; N3 <= N2;
25	5	ROT	Rotate top 3 elements on stack clockwise. T <= N2; N1 <= T; N2 <= N1;
26	6	-ROT	Rotate top 3 elements on stack counter-clockwise. T <= N1; N1 <= N2; N2 <= T;
27	7	NIP	Drop N1 and pop rest of data stack. T is unchanged. N1 <= N2; N2 <= N3;
28	8	TUCK	Duplicate T into N2 and push rest of data stack. N2 <= T; N3 <= N2;
29	9	ROT_DROP	Same as ROT DROP N2 <= N3;
2A	A	ROT_DROP_SWAP	Same as ROT DROP SWAP T <= N1; N1 <= T; N2 <= N3;
2B	B	d0_push	Load T from d0 and push data stack. T <= d0; N1 <= T; N2 <= N1; N3 <= N2;
2C	C	d0_load	Load T from d0. N1, N2, and N3 are unchanged. T <= d0;
2D	D	d0_pop	Load T from d0 and pop data stack. T <= d0; N1 <= N2; N2 <= N3;
2E	E	d01_load	Load T from d0 and load N1 from d1. T <= d0; N1 <= d1; N2 and N3 are unchanged.
2F	F	drop2	Drop T and N1 and pop data stack. T <= N2; N1 <= N3;

4. The Function Unit

The function unit, *Funit*, shown in Figure 1 has the top three elements on the register stack, *T*, *N1*, and *N2* as inputs. The outputs of *Funit* are the two 8-bit busses, *y1* and *y2*, and the single-bit carry value, *cout*. The 5-bit input, Fcode(4:0), is used to select one of the 32 W8X instructions shown in Tables 2, 3, and 4. Table 2 contains 14 conditional instructions that leave either a TRUE ("11111111") value or a FALSE ("00000000") value on the top of the register stack.

Table 2 -- Conditional Instructions

Hex Opcode	Name	Function
00	NOP	No operation
01	U>	T <= TRUE if N1 > T (unsigned), else T <= FALSE Fcode <= "00000";
02	U<	T <= TRUE if N1 < T (unsigned), else T <= FALSE Fcode <= "00001";
03	=	T <= TRUE if N1 = T, else T <= FALSE Fcode <= "00010";
04	U>=	T <= TRUE if N1 >= T (unsigned), else T <= FALSE Fcode <= "00011";
05	U<=	T <= TRUE if N1 <= T (unsigned), else T <= FALSE Fcode <= "00100";
06	<>	T <= TRUE if N1 /= T, else T <= FALSE Fcode <= "00101";
07	TRUE	Set all bits in T to '1'. Fcode <= "00110";
08	FALSE	Clear all bits in T to '0'. Fcode <= "00111";
09	>	T <= TRUE if N1 > T (signed), else T <= FALSE Fcode <= "01000";
0A	<	T <= TRUE if N1 < T (signed), else T <= FALSE Fcode <= "01001";
0B	>=	T <= TRUE if N1 >= T (signed), else T <= FALSE Fcode <= "01010";
0C	<=	T <= TRUE if N1 <= T (signed), else T <= FALSE Fcode <= "01011";
0D	NOT 0=	TRUE if all bits in T are '0'. Fcode <= "01100";
0E	0<	TRUE if sign bit of T is '1'. Fcode <= "01101";

Table 3 contains 8 logic and shift instructions. Table 4 contains 10 ALU instructions. Note that except for the last four instructions in Table 4, the value of *Fcode(4:0)* is the same as the W8Z opcode for all instructions in Tables 2, 3, and 4. Opcode 1E is a multiply partial product instruction (*mpp*) that will be described in Section 3.1. The instruction *shldc* (opcode = 1F) is used for division and will be described in Section 3.2.

Table 3 -- Logic and Shift Instructions

Hex Opcode	Name	Function
10	AND	Pop N1 and AND it to T. Fcode <= "10000";
11	OR	Pop N1 and OR it to T. Fcode <= "10001";
12	XOR	Pop N1 and XOR it to T. Fcode <= "10010";
13	2*	Arithmetic shift T left 1 bit. Fcode <= "10011";
14	U2/	Logic shift T right 1 bit. Fcode <= "10100";
15	2/	Arithmetic shift T right 1 bit. Fcode <= "10101";
16	RSHIFT	Pop T and shift N1 T bits to the right. Fcode <= "10110";
17	LSHIFT	Pop T and shift N1 T bits to the left. Fcode <= "10111";

Table 4 -- ALU Instructions

Hex Opcode	Name	Function
18	+	Pop N1 and add it to T. Fcode <= "11000";
19	-	Subtract T from N1 and pop the data stack. Fcode <= "11001";
1A	SWAP-	Subtract N1 from T and pop the data stack. Fcode <= "11010";
1B	1+	Increment T by 1. Fcode <= "11011";
1C	1-	Decrement T by 1. Fcode <= "11100";
1D	INVERT	Complement all bits of T. Fcode <= "11101";
1E	mpp	Multiply partial product If N1(0) = 1, add N2 to T and shift T:N1, else shift T:N1
1F	shldc	Shift left for division conditionally. If T > N1, subtract N1 from T and set N1(0) to '1'
0F	C>T	Push cout on the data stack

5. Program ROM and Control

The program ROM contains 16-bit words. This means that each ROM word can contain two 1-byte W8Z instructions, or a single 2-byte W8Z instruction as shown in Figure 5. The opcode of a 2-byte instruction must be in the left byte of a ROM word with an address in the right byte. This means that a 1-byte instruction between two 2-byte instructions must be padded with an X"FF" in the right byte as shown in Figure 5. This X"FF" is not executed as a NOP but is used by the controller to jump to the following 2-byte instruction when the 1-byte instruction is executed.

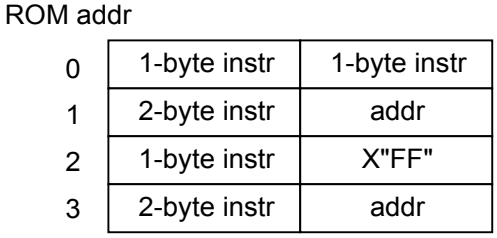

Figure 5 ROM Instruction Layout

The 16-bit output of the ROM is divided into two 8-bit signals, *M(15:8)* and *M(7:0)*. One of these two signals is sent to the controller, *Wcontrol*, through the multiplexer, *Mmux*. This multiplexer is controlled by the output of a 1-bit counter, *c1*, that toggles on the rising edge of the clock when the signal *tog* = '1'. When *c1* = '0', the mux output, *instr*, is *M(15:8)*. When *c1* = '1', the mux output, *instr*, is M(7:0). For a 2-byte instruction (such as a jump instruction), *tog* = '0'. For a 1-byte instruction, *tog* = '1' unless the instruction is in the left byte, *M(15:8),* and a X"FF" is in the right byte, *M(7:0).* In that case, *tog* = '0'.

The 8-bit address of the program ROM, *Wrom*, comes from the program counter, *WPC*, shown in the center of Figure 1. This program counter must increment by 1 when a non-jump 2-byte instruction (such as LIT) is executed, or when a 1-byte instruction in the right byte, *M(7:0)*, is executed, or when a 1-byte instruction in the left byte, *M(15:8)*, is executed and the right byte contains a X"FF". This right byte, *M(7:0)*, is an input to *Wcontrol* so that the controller can detect when this value is X"FF".

Table 5 shows three 2-byte W8Z instructions. The instruction *LIT* will push the value in *M(7:0)* onto the top of the data stack, *T*, through *dual_mux8g*. The instruction *JMP* will load the value of *M(7:0)* into the program counter, *WPC*, through *Pmux*. The instruction *JZ* will load the value in *M(7:0)* into the program counter, *WPC*, if the value in *T* is "00000000".

Table 5 – LIT and Jump 2-Byte Instructions

Hex Opcode	Name	Function
34	LIT	Load inline literal to T and push data stack.
40	JMP	Jump to inline address
41	JZ	Jump to inline address if all bits in T are '0'

6. The Return Stack

The return stack is shown in the upper-left of Figure 1. The value on the top of the return stack is stored in the register, *R*. The rest of the return stack is implemented by the module, *return_stack*, shown in Figure 6. This module consists of an 8 x 16 LogiBLOX dual-port RAM, *dpram8x16*, and a control unit, *stack_addr*. The control unit, *stack_addr*, produces the two addresses to the dual_port RAM, *wr_addr* and *rd_addr*. When a byte is pushed on the stack the byte is stored at the address, *wr_addr*, and then *wr_addr* is decremented by 1. The address, *rd_addr*, is equal to *wr_addr* + 1 as shown in Figure 7. The output *q(7:0)* in Figure 6 is the value at address*, rd_addr*; that is, the value that was just pushed on the stack. This corresponds to *R1* in Figure 1. The VHDL code for the module *stack_addr* in Figure 6 is given in Listing 1.

Table 6 lists seven W8Z instructions that involve the return stack. The instructions *>R*, *R>,* and *R@* are used to move a byte between *T* and *R*. The instructions *>R* and *DRJNE* can be used to implement a *FOR…NEXT* in WHYP as shown in Figure 8. When a *CALL* instruction is executed the subroutine address in *M(7:0)* is loaded in the program counter, *WPC*, and the return address, *P1 = P* + 2, is pushed on the return stack. When the *RET* (return from subroutine) instruction is executed, the return address in *R* is loaded into the program counter, *WPC*.

Table 6 -- Return Stack Instructions

Hex Opcode	Name	Function
30	>R	"To-R" Pop T and push it on return stack.
31	R>	"R-from" Pop return stack R and push it into T.
32	R@	"R-fetch" Copy R to T and push register stack
33	R>DROP	"R-from-drop" Pop return stack R and throw it away
42	DRJNE	Decrement R and jump if R is not zero
43	CALL	Call subroutine
44	RET	Subroutine return

7. W8Z Memory and I/O

The RAM shown in the upper-right of Figure 1 can be a scratchpad RAM implemented with a LogiBLOX module. Data can be written to the RAM using the WHYP word *C! (data addr --)* shown in Table 7. Data can be read from the RAM using the WHYP word *C@ (addr – data)* shown in Table 7. The ROM shown in the upper-right of Figure 1 can be used to hold permanent data and implemented with a LogiBLOX ROM module. Data can be read from the ROM using the WHYP word *ROM@ (addr – data)* shown in Table 7.

Table 7 – Memory and I/O Instructions

Hex Opcode	Name	Function
35	C@	Fetch the byte at address T in RAM and load it into T
36	C!	Store the byte in N1 at the address T. Pop both T and N1.
37	ROM@	Fetch the byte at address T in ROM and load it into T.
38	SW@	Fetch the 8 switch readings and load it into T.
39	DIG!	Write the 4 hex digits in N1:T to the digit register DigReg
3A	LD!	Store the byte in T to the LED register LDreg.
45	JNBTN1	Jump if BTN1 is not pushed
46	JNBTN2	Jump if BTN2 is not pushed
47	JNBTN3	Jump if BTN3 is not pushed
48	JNBTN4	Jump if BTN4 is not pushed
49	JBTN1	Jump if BTN1 is pushed
4A	JBTN2	Jump if BTN2 is pushed
4B	JBTN3	Jump if BTN3 is pushed
4C	JBTN4	Jump if BTN4 is pushed

The WHYP word *SW@ (-- n)* shown in Table 7 can be used to read the 8 switch settings on the Digilab board into *T*. The WHYP word *DIG! (N1 T --)* can be used to latch the 16-bit value in *N1:T* into the 16-bit register *DigReg*. By connecting the outputs of this register to the module *step_display* shown in Figure 8, the hex values in *N1* and *T* can be displayed until a new value is stored in *DigReg*. The eight bits in *T* can be displayed on the 8 LEDs on the

Digilab board by storing *T* in the 8-bit register, *LDreg*, using the WHYP word *LD! (t --)* shown in Table 7.

The signals from the four pushbuttons, BTN(1:4), are inputs to the module Wcontrol as shown in Figure 1. These signals can be used to implement the eight jump instructions shown in Table 7. For example, the following two instructions can be used to wait for a button 3 to be pressed.

```
4  JBTN3,   X"04",
5  JNBTN3,  X"05",
```

The first instruction at ROM address 4 will branch on itself until the button has been released from a possible previous pressing. The second instruction at ROM address 5 will branch on itself until the button is pressed.

8. Sample W8Z Program

The program ROM, *Wrom*, contains the program to be executed. To test the W8X the Forth program shown in Figure 8 was converted to W8Z code as shown in the VHDL program given in Listing 2. This program first stores a hex $80 in the register, *LDreg*, connected to the 8 LEDs. This value is arithmetic shifted right 8 times which will light up all 8 LEDs. The second FOR...NEXT loop stores a $7F (which will turn off the leftmost LED) and then logic shifts this value right 8 times which will turn off each LED in sequence. This process is then repeated endlessly.

This Forth program can be converted to W8Z instructions using the coding form shown in Figure 10. This is done in Figure 11 and then copied to the VHDL ROM program shown in Listing 2.

```
HEX
: MAIN ( -- )
  BEGIN
     80
     8 FOR
       DUP LD! DELAY 2/
     NEXT
     DROP 7F
     8 FOR
       DUP LD! DELAY U2/
     NEXT
     DROP
   AGAIN ;

: DELAY  ( -- )
   5 FOR NEXT ;
```

Figure 9 Test Forth Program

9. Summary

The W8Z is a high-performance microprocessor core that has been implemented on a Xilinx Spartan FPGA. All of the 72 W8Z instructions given in Tables 1 – 7 are executed in a single clock cycle. Of these 72 instructions, only 11 are actually used to implement the Forth program shown in Figure 9. This means that only these 11 instructions need to be implemented in VHDL when synthesizing the microprocessor core for this program. That is, the microprocessor core is elastic in the sense that its size grows and shrinks to accommodate a particular program. In this way the minimum amount of hardware resources are used for every program.

Other hardware components can easily be added to this design. 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. By including a timer module, the W8Z could become a very useful high-performance, low-cost microcontroller. Adding a UART would allow interactive communication with the W8Z through a standard asynchronous serial line.

10. References

1. Haskell, R. E., *Design of Embedded Systems Using 68HC12/11 Microcontrollers*, Prentice Hall, Upper Saddle River, NJ, 2000.
2. 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.
3. Hand, T., "The Harris RTX 2000 Microcontroller," *Journal of Forth Application and Research*, Vol. 6, No. 1, pp. 5-13, 1990.
4. Koopman, Jr., P., "32-Bit RTX Chip Prototype," *Journal of Forth Application and Research*, Vol. 5, No. 2, pp. 331-335, 1988.
5. 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.
6. 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.
7. Moore, C., "ShBoom on ShBoom: A Microcosm of Software and Hardware Tools," Proc. 1990 Rochester Forth Conference, pp. 21-27, June 12-15, 1990.
8. Ting, C. H., "P Series of Microprocessors," in *More on Forth Engines*, Vol. 22, pp. 1-17, Sept. 1997.
9. Ting, C. H., "P16 Microprocessor Design in VHDL," in *More on Forth Engines*, Vol. 22, pp. 44-51, Sept. 1997.
10. 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.
11. Ashenden, P. J., *The Designer's Guide to VHDL*, Morgan Kaufmann, San Francisco, 1996.
12. 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.
13. http://www.digilent.cc/ (Visited on March 1, 2001).