Implementing a Forth Engine Microcontroller on a Xilinx FPGA

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

Introduction

In a junior-level course in computer hardware design it is useful to have students design a real microprocessor and implement it in an FPGA. Most real microprocessors can be categorized as having either a complex instruction set computer (CISC) architecture or a reduced instruction set computer (RISC) architecture [1]. Both of these architectures involve a set of registers and multiple addressing modes. A simpler architecture that is easier to implement in an FPGA is a stack-based processor in which all arithmetic and logical operations are performed on the top two elements of a data stack. This architecture is ideally suited for efficiently executing high-level Forth programs.

Forth is a programming language invented by Chuck Moore 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 by writing his programs in 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 new book on using the Motorola 68HC12 microcontroller in embedded systems [2]. WHYP stands for *W*ords to *H*elp *Y*ou *P*rogram. 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 [3]. 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 [4]. Similar 32-bit Forth engines were also developed [5-7]. 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 [8]. Later Chuck Moore and C. H. Ting designed the MuP21 that has been described by Ting [9, 10]. The W8X microcontroller described in this article is based on ideas developed in these early Forth engines. It is designed using VHDL [11] and has been implemented in a Xilinx FPGA by students in a junior-level course at Oakland University.

The W8X Microcontroller

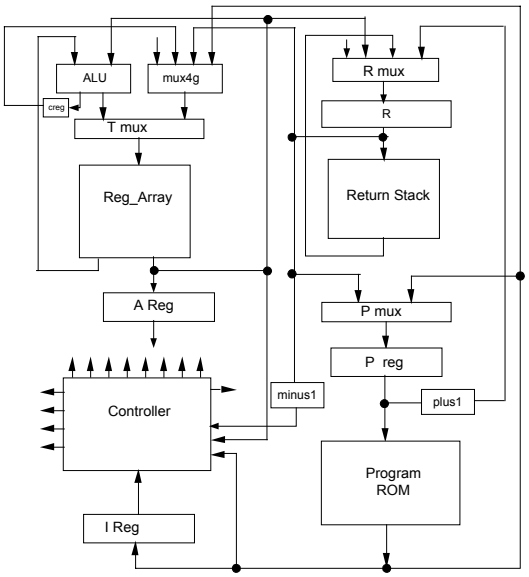

Figure 1: The W8X Microprocessor

The W8X is a high-performance microcontroller that can be implemented to perform useful functions on a Xilinx 4000 series FPGA. The overall structure of the W8X is shown in Figure 1. The data busses in this figure are 8 bits wide and each instruction contains 8 bits. The W8X instruction set is given in Table 1.

Table 1: W8X Instruction Set

Opcode	Name	Function
00	DUP	Duplicate T and push data stack.
01	DROP	Drop T and pop data stack.
02	SWAP	Exchange T and N1.
03	NIP	Drop N1 and pop rest of data stack.
04	ROT	Rotate top 3 elements on stack clockwise.
05	MROT	Rotate top 3 elements on stack counter-clockwise.
06	OVER	Duplicate N1 into T and push data stack.
07	TUCK	Duplicate T into N2 and push rest of data stack.
08	NOP	No operation
09	TOR	"To-R" Pop T and push it on return stack.
0A	RFROM	"R-from" Pop return stack R and push it into T.
0B	RFETCH	"R-fetch" Copy R to T and push register stack
10	LSL	Logic shift left T
11	ASR	Arithmetic shift right T
12	LSR	Logic shift right T
13	ROTR	Rotate right T (carry unchanged)
14	ROTL	Rotate left T (carry unchanged)
20	ZEROS	Clear all bits in T to '0'.
21	PLUS	Pop N1 and add it to T.
22	MINUS	Pop T and subtract it from N1.
23	ANDD	Pop N1 and AND it to T.
24	ORR	Pop N1 and AND it to T.
25	XORR	Pop N1 and AND it to T.
26	INVERT	Complement all bits of T.
27	ONES	Set all bits in T to '1'.
28	ZEQUAL	TRUE if all bits in T are '0'.
29	ZLESS	TRUE if sign bit of T is '1'.
2A	CTOT	Push carry bit to top of register stack
2B	1PLUS	Add 1 to T
2C	1MINUS	Subtract 1 from T
2D	mpp	Multiply partial product
2E	shld	Shift left T and N1 for division
2F	subc	If T > N1, subtract N1 from T and set N1(0) to '1'
40	LIT	Load inline literal to T and push data stack.
31	C@	Fetch the byte at addr T in RAM and load it into T
32	C!	Store the byte in N1 at the address T
41	JMP	Jump to inline address
42	JZ	Jump if all bits in T are '0'
43	JNC	Jump if carry is cleared
44	DRJNE	Decrement R and jump if R is not zero
45	CALL	Call subroutine
46	RET	Subroutine return

The data stack in the W8x is a register array designed with four multiplexers combined with four registers. The multiplexer for a stack register switches the output from any of the other three registers to its input as shown in Figure 2.

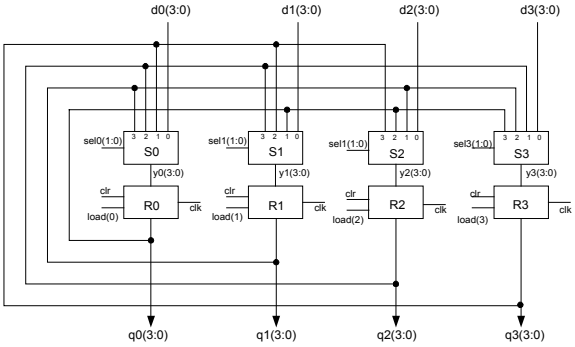

Figure 2: A Register Array Data Stack

This provides flexibility to implement the stack instructions in one clock cycle. Although each of the stack registers have the capability, by design, to be loaded, the W8x only uses the load provided to the register designated as the top of the stack. This design provides an 8x4 register array data stack and an 8x16 return stack. The return stack is not an array-bases stack since the flexibility to manipulate individual stack data is not necessary to obtain single clock-cycle instructions. The input to the data stack is multiplexed from two sources, the ALU and a 4-

input multiplexer. The ALU performs the operations shown below in Table 2.

Table 2: W8X ALU Operations

ALU Select	Operation
"000"	all zeros
"001"	a + b
"010"	b - a
"011"	a and b
"100"	a or b
"101"	a xor b
"110"	not a
"111"	all ones

The 8-bit inputs into the ALU are T and N, the top and second elements in the stack, respectively. The 4-input multiplexer provides an external signal, the carry out from the ALU, the top of the return stack, and the current value in the program memory addressed by the program counter. The program for the W8X is stored in a program ROM. The ROM is addressed by the program counter which can be loaded with a value from the return stack or from memory for return from subroutine instructions or instructions that may jump to an inline address. The return stack can be loaded with values from the top of the data stack and the program counter plus one. The W8X microprocessor is controlled by a microcontroller and instruction register. The microcontroller is the mealy state machine shown below in Figure 3.

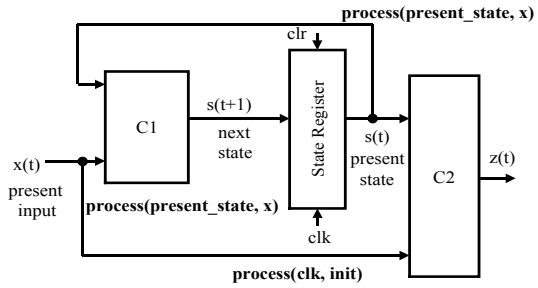

Figure 3: The W8X State Machine

This state machine has three states: Fetch, Execute, and Execute-Fetch. A portion of its VHDL implementation is shown in Figure 4. The controller begins in the fetch state to 'fetch' the next instruction from the program ROM. If the instruction requires only a single clock instruction 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 until it is presented with an instruction that requires more than one clock cycle.

```
C1: process(current_state, M)
begin
  case current_state is
    when fetch =>
      if M(6) = '1' then
        next_state <= exec;
      else
        next_state <= exec_fetch;
      end if;
    when exec_fetch =>
      if M(6) = '1' then
        next_state <= exec;
      else
        next_state <= exec_fetch;
      end if;
    when exec =>
       next_state <= fetch;
  end case;
end process C1;
```

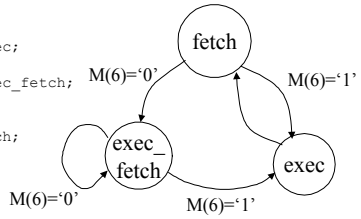

Figure 4: Three-State VHDL implementation

These multi-cycle instructions have been assigned opcodes with a '1' in the 6th bit position, the second most significant bit of the opcode. For instructions requiring two clock cycles, the controller executes the current instruction without 'fetching' the next word from the program ROM. On the second clock cycle, the microcontroller returns to the fetch state to 'fetch' the next instruction.

Extending the W8X

In addition to implementing, simulating, and synthesizing the W8X controller in VHDL on a 4000 series FPGA, students are required to complete a design project that extends the W8X controller. For example, the top of stack register in the Register Array was altered to be an SPI (serial peripheral interface). The new SPI top of stack register is shown below in Figure 5.

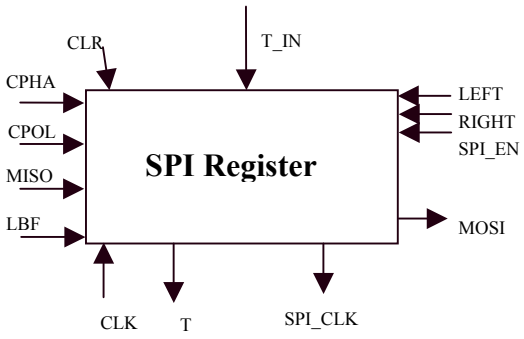

Figure 5: An SPI Register

This SPI register was implemented as the top of the data stack. An instruction, SPI was added to the instruction set. Obviously, this instruction requires more than one or two clock cycles. The microcontroller was modified to remain in the execute state for the current number of clock cycles

to complete the SPI transmission. The SPI runs in all four standard SPI modes: active high and low with CPHA = 1 and active high and low with CPHA = 0. With this extension, the W8X will now accept data through the SPI directly into the top of the stack. This design was synthesized on a Xilinx 4010 FPGA and simulated using the Aldec Active VHDL simulator. A simulation is shown below in Figure 6.

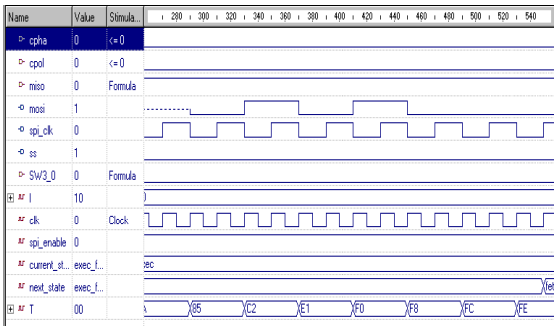

Figure 6: SPI Simulation CPHA=0 CPOL=0

References

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