ECE 4710 Computer Hardware Design D. M. Hanna

Lab 7 – A Data Stack and Return Stack using RAM - Stack Instructions

This lab contains 3 parts and should be done in groups of 2. If there is an odd number of people, you may form a group of 3, but ideally this should be done in pairs. In part A you will implement a stack and use it to create a Data Stack. In part B you will use the stack from part A and create a Return Stack and instructions for multiplication and division. In Part C you will compare executing the factorial algorithm on the WC16 to computing the factorial using the Special Purpose Processor created in Lab 3.

**PART A: A 32x16 Data Stack**

A 32 x 16 stack, *stack32x16*, is shown in Figure 7A.1. The 16-bit output *q(15:0)* always contains the value of the top element on the stack. If the stack is empty then *empty* will be '1' and the values on *q(15:0)* will be meaningless and no action is taken on the rising edge of the clock. If *clr* is '1' then on the rising edge of the clock input, *clk*, the stack becomes empty regardless of the values of *push* or *pop*. If *push* is '1' and *pop* is '0' then on the rising edge of the clock input *clk* the 16-bit input *d(15:0)* will be pushed on the stack. If *pop* is '1' and *push* is '0' then on the rising edge of the clock input *clk* the top element on the stack will be "popped" and the next element on the stack will be moved to the top of the stack. If both *push* and *pop* are '1', then on the rising edge of the clock input *clk* the 16-bit input *d(15:0)* will overwrite the top element of the stack without pushing other values down the stack. If both *push* and *pop* are zero then no action is taken on the rising edge of the clock. Once 32 values have been pushed on the stack the *full* bit is set to '1' and no further values can be pushed on the stack until some are popped.

A behavioral VHDL model of such a stack that uses flip-flops to implement the RAM will not yield a very efficient implementation. It is always more efficient to implement memory in Xilinx FPGAs using the table lookup distributed RAMs that are part of each Slice in the Artix-7 FPGA. The Artix-7 also has a block RAM, but this block RAM can have only registered outputs in the Artix-7. We will need to use non-registered outputs from a dual-port RAM, and must therefore use the distributed RAM in the Artix-7. You can do this by using an IP Core Memory dual-port distributed RAM module.



An RTL model of the stack, called *stack32x16*, which uses an IP Core 32 x 16 dual-port distributed RAM module called *dpram32x16*, and a second entity called *stack\_ctrl32* is shown in Figure 7A.2. The address *wr\_addr* will point to the next address to be written to when *push* is '1' and *pop* is ‘0’. The address *rd\_addr* will point to the current top of the stack whose value is on the output *q*(15:0). In general, (except when the stack is full) the value of *rd\_addr* will be one more than the value of *wr\_addr*. When a value is pushed on the stack, *wr\_addr* is decremented and when a value is popped from the stack, *rd\_addr* is incremented. The use of a dual-port RAM allows us to read and write from different locations on the same clock cycle. The mux select signal, *amsel*, will be equal to ‘1’ when *push* = ‘1’ and *pop* = ‘1’. This will allow writing the new data to *rd\_addr*. The VHDL description for the control module *stack\_ctrl32* is given in Listing 8A.1.

Fig. 7A.1 A 32 x 16 stack

A 32 x 16 data stack is shown in Figure 7A.3. Note that it includes the stack module, *stack32x16*, plus the two 16-bit registers, *T* and *N*. The register *T* contains the top element on the data stack and the register *N* contains the second (or next) element on the data stack. The top element in the stack module *stack32x16* is *N2* and represents the third element on the data stack. Separating out the top two elements of the data stack, *T* and *N*, will allow us to implement several stack manipulation instructions involving the top three elements of the data stack (*T*, *N*, and *N2*). It may look as if the total depth of the data stack in Figure 7A.3 is 34 rather than 32. However, only 32 elements can actually be pushed on the stack because when the first two elements are pushed into *T* and *N*, the previous values in *T* and *N* (presumably zeros) have been pushed into the *stack32x16* module. Therefore, as soon as 32 values are pushed onto the data stack, it will be full.

**library** IEEE;**use** IEEE.std\_logic\_1164.**all**;**use** IEEE.std\_logic\_unsigned.**all**;**entity** stack\_ctrl **is** **port** ( clr: **in** STD\_LOGIC;

clk: **in** STD\_LOGIC;

push: **in** STD\_LOGIC;

pop: **in** STD\_LOGIC;

we: **out** STD\_LOGIC;

amsel: **out** STD\_LOGIC;

wr\_addr: **out** STD\_LOGIC\_VECTOR(4 **downto** 0);

rd\_addr: **out** STD\_LOGIC\_VECTOR(4 **downto** 0);

full: **out** STD\_LOGIC;

empty: **out** STD\_LOGIC

);

**end** stack\_ctrl;

**architecture** stack\_ctrl\_arch **of** stack\_ctrl **is**

**signal** full\_flag, empty\_flag: STD\_LOGIC;

**begin**

stk: **process**(clr, clk, push, pop, full\_flag, empty\_flag)

**variable** push\_addr, pop\_addr: STD\_LOGIC\_VECTOR(4 downto 0);

**begin**

**if** clr = '1' **then**

push\_addr := "11111";

pop\_addr := "00000";

empty\_flag <= '1';

full\_flag <= '0';

wr\_addr <= "11111";

rd\_addr <= "00000";

full <= full\_flag;

empty <= empty\_flag;

**elsif** clk'event and clk = '1' **then**

**if** push = '1' **then**

**if** pop = '0' **then**

**if** full\_flag = '0' **then**

push\_addr := push\_addr - 1;

pop\_addr := push\_addr + 1;

empty\_flag <= '0';

**if** push\_addr = "11111" **then**

full\_flag <= '1';

push\_addr := "00000";

**end if**;

**end if**;

**end if**;

**elsif** pop = '1' **then**

**if** empty\_flag = '0' **then**

pop\_addr := pop\_addr + 1;

**if** full\_flag = '0' **then**

push\_addr := push\_addr + 1;

**end if**;

full\_flag <= '0';

**if** pop\_addr = "00000" **then**

empty\_flag <= '1';

**end if**;

**end if**;

**end if**;

wr\_addr <= push\_addr;

rd\_addr <= pop\_addr;

**end if**;

full <= full\_flag;

empty <= empty\_flag;

**if** push = '1' and full\_flag = '0' **then**

we <= '1';

**else**

we <= '0';

**end if**;

**if** push = '1' and pop = '1' **then**

amsel <= '1';

**else**

amsel <= '0';

**end if**;

**end process** stk;

**end** stack\_ctrl\_arch;

**Listing 7A.1** stack\_ctrl.vhd



Fig. 7A.2 Implementing an 32 x 16 stack using an 32 x 16 dual-port distributed RAM



Fig. 7A.3 A 32 x 16 data stack

**Data Stack Instructions**

Table 7A.1 is a list of the data stack instructions that involve *T*, *N*, and *N2*. These instructions are included in the package *opcodes.vhd* used in Lab 6.

Table 7A.1 Data Stack Instructions

|  |  |  |
| --- | --- | --- |
| Hex 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 N2 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 WC16B WHYP core is shown in Figure 7A.4 and the top-level design for Lab 7A is shown in Figure 7A.5. Note that the program ROM and the *DigDisplay* module are taken outside the WC16B core.

1. Write a structural VHDL program for the *stack32x16* model shown in Figure 7A.1 using the structure shown in Figure 7A.2. Use an IP Core distributed dual-port RAM for the Artix-7 FPGA to implement the *dpram32x16* module. Print out the listing for *stack32x16*.
2. Write a structural VHDL program for the module *DataStack* shown in Figure 7A.3. Simulate the module *DataStack*. In your test bench, create a process that is a counter for the stimulator for *Tin* that starts at 0000 and increments in steps of 1; this is similar to the clock process except that it will be a 16-bit counter. Continue pushing values on the stack until the stack becomes full, and then pop all values off the stack. Submit a screenshot of the waveform and hand in the listing for *DataStack*.
3. Write a structural VHDL program for the module *WC16B* shown in Figure 7A.4. Write the VHDL program for the controller *WC16B\_control* (modify *Pcontrol.vhd* from Lab 6). Implement all of the instructions in *Funit1* (from Lab 6) plus all the new data stack instructions in Table 7A.1.
4. Write a structural VHDL program for the top-level design *Lab7A* shown in Figure 7A.4. Use a program ROM, *prom8A*, created by compiling the WHYP program *addsub.whp*.



Fig. 7A.4 The WC16B WHYP Core

**WC16B**

clk

clr

T

S

B

P

M

mclk

BTN(3)

**Program**

**ROM**

P

M

AN(3:0)

AtoG(6:0)

T

BTN(0)

SW(7:0)

**Lab7A**

digload

LD(7:0)

clkdiv

cclk

clr

clk

debounce

x7seg

clr

cclk

DIGreg

clk

clr

digload

xin

Fig. 7A.5 Lab7A top-level design

1. Simulate the design by writing a test bench for the simulator. Set the signal *clr* to 1 at 0 ns and 0 at 5 ns. Apply a 100 MHz clock stimulator to the *clk* signal. Set the signal *SW[7:0]* to 16#12 at 0 ns, 16#34 at 500 ns, 16#56 at 1000 ns, and 16#78 at 1500 ns. Set *BTN0* to ‘0’ at 0 ns, ‘1’ at 200 ns, ‘0’ at 300 ns, ‘1’ at 700 ns, and ‘0’ at 800 ns, ‘1’ at 1200 ns, ‘0’ at 1300 ns, ‘1’ at 1700 ns, and ‘0’ at 1800 ns, ‘1’ at 2200 ns, ‘0’ at 2300 ns, ‘1’ at 2700 ns, and ‘0’ at 2800 ns, ‘1’ at 3200 ns, ‘0’ at 3300 ns. Run the simulator for 4000 ns. Turn in a screenshot of the waveform and listings of the files *Lab7A.vhd*, *stack32x16.vhd*, *datastack.vhd*, *WC16B.vhd*, *WC16B\_control.vhd* and *Prom8A.vhd*.
2. Synthesize and implement the *Lab7A* project and download the .*bit* file to the Nexys-4 board. Note that you must load a *copy* of your *opcodes.vhd* file to the working directory of your Vivado project. Also, you must generate an IP Core for the dual-port RAM. Test the program by pushing X”1234” and X”5678” onto the data stack from the switches and then press *BTN(0)* three times to display the sum, difference and AND operations. Demonstrate the operation of *Lab7A* to your lab instructor.

PART B: **ReturnStack**



A 32 x 16 return stack is shown in Figure 7B.1. Note that it includes the stack module, *stack32x16*, from Part A of this lab plus the 16-bit register *R* that contains the top element on the return stack. A WHYP core that contains this return stack is shown in Figure 7B.2.

Fig. 7B.1 A Return Stack

Table 7B.1 is a list of the return stack instructions and Table 7B.2 shows additional transfer instructions. These instructions are included in the package *opcodes.vhd* used in Lab 6.

1. Write a structural VHDL program for the module *ReturnStack* shown in Figure 7B.1. Simulate the module *ReturnStack* using Aldec-HDL. Use a counter for the stimulator for *Rin* that starts at 0000 and increments in steps of 1. Continue pushing values on the stack until the stack becomes full, and then pop all values off the stack. Turn in a screen shot of the waveform and the listing for *ReturnStack*.

.

Table 7B.1 Return Stack and I/O Instructions

|  |  |  |
| --- | --- | --- |
| Code | 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 |
| 0039 | LD! | “LD store” Store 8-bits of T into the 8-bit LD register |

Table 7B.2 Transfer Instructions

|  |  |  |
| --- | --- | --- |
| Code | Name | Function |
| 0103 | DRJNE | Decrement R and jump if R is not zero |
| 0104 | CALL | Call subroutine |
| 0105 | RET | Subroutine return |

**The WC16 Core: Putting it all together**

A final version of the WC16 core is shown in Figure 7B.2. Note that it includes a new *Funit*, called *Funit2*, which includes the multiplication and division primitives, *mpp* and *shldc*, shown in Table 7B.3.



Fig. 7B.2 The WC16 WHYP Core

Table 7B.4 shows a new I/O instruction and Table 7B.5 is a list of additional transfer and multi-cycle instructions. These instructions are included in the package *opcodes.vhd* used in Lab 6.

The top-level design for Lab 7B is shown in Figure 7B.3.

Table 7B.3 New Funit Instructions

|  |  |  |
| --- | --- | --- |
| Code | Name | Function |
| 001D | mpp | multiply partial product (used for multiplication) |
| 001E | shldc | shift left and decrement conditionally (used for division) |

Table 7B.4 New I/O Instructions

|  |  |  |
| --- | --- | --- |
| Code | Name | Function |
| 0034 | @ | “Fetch” Push contents of RAM address, *T*, into *T* via *E1*. |

Table 7B.5 New Transfer and Multi-Cycle Instructions

|  |  |  |
| --- | --- | --- |
| Code | Name | Function |
| 0106 | JB1LO | Jump if input pin B1 is LO |
| 0107 | JB2LO | Jump if input pin B2 is LO |
| 0108 | JB3LO | Jump if input pin B3 is LO |
| 0109 | JB0LO | Jump if input pin B4 is LO |
| 010A | JB1HI | Jump if input pin B1 is HI |
| 010B | JB2HI | Jump if input pin B1 is HI |
| 010C | JB3HI | Jump if input pin B1 is HI |
| 010D | JB0HI | Jump if input pin B1 is HI |
| 010E | ! | “Store” Write the word in *N* at the address in *T*. Pop both *T* and *N*. |



Fig. 3 Lab 7B Top-level Design

**Multiplication and Division**

Multiplication of a 16-bit unsigned number in *T* by a 16-bit unsigned number in *N* producing a 32-bit product in *T:N* can be accomplished by adding the variables shown in Figure 7B.4 to the process in *Funit2* and by implementing the *Funit2* operation *mpp* as shown in Figure 7B.5.

**variable** AVector: STD\_LOGIC\_VECTOR (width **downto** 0);

**variable** BVector: STD\_LOGIC\_VECTOR (width **downto** 0);

**variable** CVector: STD\_LOGIC\_VECTOR (width **downto** 0);

**variable** yVector: STD\_LOGIC\_VECTOR (width **downto** 0);

**variable** y1\_tmp: STD\_LOGIC\_VECTOR (width-1 **downto** 0);

Fig. 7B.4 Variables to add to process in Funit2

Fig. 7B.5 Implementation of *mpp* and *shldc* in *Funit2*

when "011101" => -- *mpp*

**if** b(0) = '1' **then**

yVector := AVector + CVector;

**else**

yVector := AVector;

**end if**;

y <= yVector(width **downto** 1);

y1 <= yVector(0) & b(width-1 **downto** 1);

when "011110" => -- *shldc*

yVector := a & b(width-1);

y1\_tmp := b(width-2 **downto** 0) & '0';

**if** yVector >= CVector **then**

yVector := yVector - CVector;

y1\_tmp(0) := '1';

**end if**;

y <= yVector(width-1 **downto** 0);

y1 <= y1\_tmp;

The WHYP word *UM\* ( u1 u2 - upL upH )* can then be implemented using the colon definition shown in Figure 7B.6. Note that this multiplication will take 23 clock cycles to execute, including the CALL and RET instructions (2 clock cycles each).

: UM\* ( u1 u2 - upL upH )

0

mpp mpp mpp mpp

mpp mpp mpp mpp

mpp mpp mpp mpp

mpp mpp mpp mpp

ROT\_DROP ;

Fig. 7B.6 The WHYP multiplication word, *UM\**

Dividing a 32-bit unsigned number in *N2:N* by a 16-bit unsigned number in *T* producing a 16-bit quotient in *T* and a 16-bit remainder in N can be accomplished by adding the variables shown in Figure 7B.4 to the process in *Funit2* and by implementing the *Funit2* operation *shldc* as shown in Figure 7B.5.

The WHYP word *UM/MOD ( unL unH ud -- ur uq )* can then be implemented using the colon definition shown in Figure 7B.7. Note that this division will take 22 clock cycles to execute, including the CALL and RET instructions.

: UM/MOD ( unL unH ud -- ur uq )

-ROT

shldc shldc shldc shldc

shldc shldc shldc shldc

shldc shldc shldc shldc

shldc shldc shldc shldc

ROT\_DROP\_SWAP ;

Fig. 7B.5 The WHYP division word, *UM/MOD*

**Lab 7B**

1. Modify the function unit, *Funit1*, used in part A to include the new operations, *mpp* and *shldc*. Call the new function unit *Funit2*. Note that this function unit has three input busses and two output busses.
2. Write a structural VHDL program for the module *WC16* shown in Figure 7B.2. Write the VHDL program for the controller *WC16\_control* (modify *WC16B\_control.vhd* from part A). In addition to all of the instructions from Part A, implement the new instructions in Tables 1 - 3.
3. Write a structural VHDL program for the top-level design *Lab7B* shown in Figure 3.
4. Test your *Lab7B* module by compiling each of the following WHYP programs (available on the web) using the program WC16.EXE (available on the web) that will produce the VHDL code to insert in your ROM program and then download the program to the Nexys-4 board. Demonstrate each program to your lab instructor as you get it to work.

**sdigtest.whp** -- tests the switches and 7-segment displays

**hex2asc.whp** -- converts hex to ascii

**minmax.whp** -- finds the min and max of two numbers

**mul.whp** -- multiplies two 16-bit numbers

**div.whp** -- divides a 32-bit number by a 16-bit number

**fact16.whp** -- computes the factorial of a number

**leap.whp** -- checks if a year is a leap year

**PART C: Comparing Factorial on the WC16 with a Factorial SPP**

In Lab 3, you designed a special-purpose processor comprised of a state machine and data path to compute the factorial of an input. Now that you have completed implementing the WC16 processor, compare computing 5! On the WC16 with computing it using the SPP.

1. How many clock cycles does it take in each?
2. What is the critical path timing of each? What is the maximum frequency clock with which you can drive each?
3. How much space does each take up on the Artix-7 FPGA?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | # Clock Cycles to Compute 5! | Critical Path Timing (ns) | Maximum Clock Frequency | Space required on FPGA |
| WC16 |  |  |  |  |
| Special-Purpose Hardware |  |  |  |  |