EE 209 Project Part 2 – Blaze of Glory

1 Introduction

In this project you will integrate your heap hardware engine with a soft-core processor and glue logic so that the heap engine can be used by software executing on the processor. This lab should be completed **INDIVIDUALLY**!

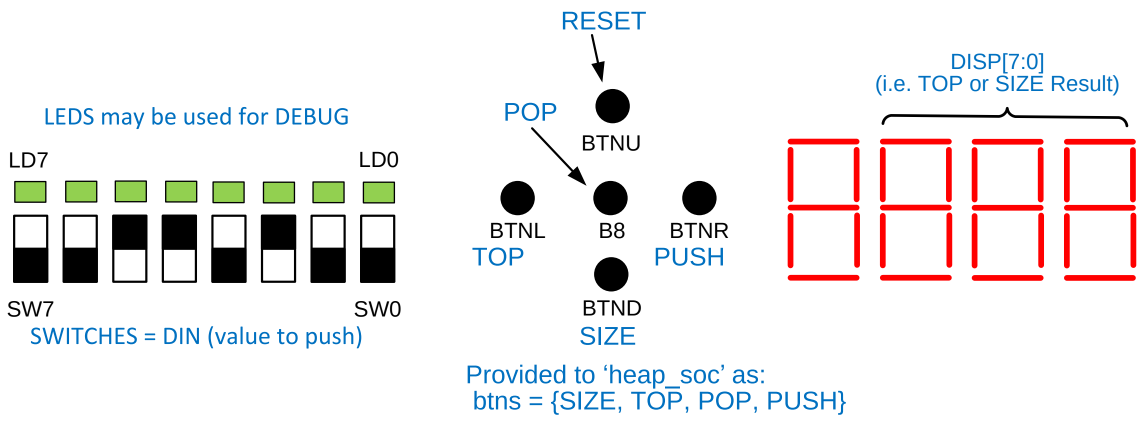
2 What you will learn

This lab is intended to teach you how custom hardware integrates with a processor core and software similar to what you'd see on a modern system-on-chip (SoC) design.

3 Background Information and Notes

1. **Overview**

   The user will be allowed to choose from 4 operations: SIZE, TOP, POP, and PUSH using the buttons on the FPGA board. For a PUSH operation, the data value to be added (i.e. DIN) will be the current value of the SWITCHES. After completion of either of the 3 operations: TOP, PUSH, and POP the top value of the heap should be displayed on the 7-segment displays. If the SIZE operation is chosen, then the current number of elements in the heap should be displayed on the 7-segment displays. Note: When the heap becomes empty, we will still display whatever is in the memory element 1 (i.e. the old top). The I/O mapping is shown below.


RESET
DISP[7:0]
(i.e. TOP or SIZE Result)
LEDS may be used for DEBUG
POP
BTNU
LD7
LD0
BTNL
TOP
B8
BTNR
PUSH
SW7
SW0
SWITCHES = DIN (value to push)
BTND
SIZE
Provided to 'heap_soc' as:
btns = {SIZE, TOP, POP, PUSH}

   Essentially, you'll write software to poll the push-buttons and when you find a pressed button, have your software initiate the operation, poll to see when the operation is done, acknowledge the operation is done, and then read and display the result of the operation (either the TOP or SIZE value) on the 7-segment displays. At this point, you will repeat and start polling on the pushbuttons again, repeating the entire process. The button mapping is shown below.

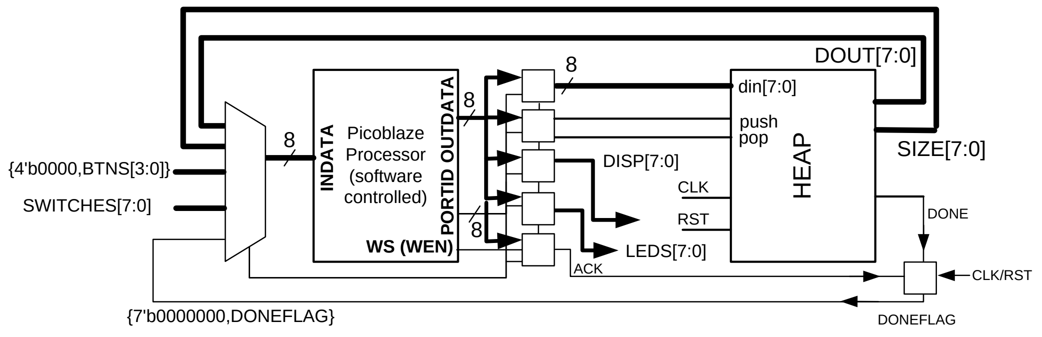
PushButton (BTN[3:0])	**Desired Operation**
4'b1000	Read and display the SIZE of the heap
4'b0100	Read and display the TOP element of the heap
4'b0010	Pop the heap
4'b0001	Push a new value (din=SWITCHES) to the heap

2. PicoBlaze Soft-core Processor

The PicoBlaze is a soft-core processor design available from Xilinx that can be synthesized easily to work on Xilinx' FPGA products. A soft-core processor means the actual HW for the processor logic is not already built on the FPGA but instead it must be synthesized into the appropriate FPGA form and then the FPGA fabric "implements" the processor logic. This is unlike some of Xilinx's (and other vendors') new product lines that have several processor cores built directly onto the FPGA chip and surrounded by the traditional FPGA fabric which can be used for custom logic processing.

We call the soft-core a **3rd party IP (Intellectual Property)** because many designs for specific processing tasks already exist and are sold by 3rd party companies to be integrated into your design and placed on an FPGA or ASIC. A **System-on-chip** is one that comprises software running on a general purpose processor along with custom hardware for specific tasks. You will implement a small SoC here.

The PicoBlaze processor is an 8-bit microcontroller with 16 registers (names **s0-sf**) and 8-bit address, input data, and output data ports for communicating with external logic. More details for this important aspect are discussed in the next sections, but below is a block diagram of the system. The heap engine is essentially what you completed in the previous part of this project. The Picoblaze will be provided to you but you will need to add all the periphery logic needed to interface the two designs. Please refer to this diagram below and flesh it out in more detail as exactly what will be needed in each of the boxes.



3. **Exposing Your Hardware I/O to Software: Addressable registers**
To interface HW with software we use the normal input/output reads/writes the processor can perform and add "addressable" registers on the periphery of the CPU which will interface to our custom hardware. In this case the CPU will need to provide the DIN[7:0] input to the heap engine as well as the PUSH and POP signals. In addition, the CPU can retrieve results by reading (input) again using addresses to identify what value is desired. In the PicoBlaze this input/output read/write capability is performed by the instructions: INPUT and OUTPUT

```
INPUT   sx, hex_addr
OUTPUT  sx, hex_addr
```

The input instruction will output an 8-bit address known as a PORTID (specify it in hex in your assembly). Your logic will need to examine this PORTID (or some subset of its bits) to control what value is selected (generally via some muxing) and placed at the INDATA[7:0] input of the processor. It is your responsibility to add the muxes and use the PORTID address bits to select the correct value for the processor to read. Whatever is read is placed into the register **sX** for further examination by your software. We will ask you to use the following address/memory map:

Address / PORTID (in binary)	**Value to Select**	**Description**
0000 0000	{4'b0000, btn[3:0]}	4 Push buttons from the FPGA = {SIZE, TOP, POP, PUSH}.
0000 0001	switches[7:0]	DIN value to be used for PUSH operations
0000 0010	{7'b0000000,doneflag}	Done flag indicating a PUSH or POP is complete
0000 0011	dout[7:0]	TOP element from the heap
0000 0100	size[7:0]	Number of elements in the heap

The OUTPUT instruction outputs the provided 8-bit address/PORTID as well as the provided data from the register **sX** on to the OUTDATA bits. In addition a 1-bit signal: **write strobe (aka 'wen' = write enable)** is asserted for a clock cycle and can be used as a register enable along with appropriate PORTID bits. First we will assign a unique address to each register of information we want our processor to be able to write to/assign.

Address / PORTID (in binary)	**Register/Signals to Generate**	**Description**
0000 0001	din[7:0]	Value to be pushed to your heap
0000 0010	Start signals: 8’b0000 0010 = POP 8’b0000 0001 = PUSH	The POP and PUSH start signals should share the same address with the POP signal being bit 1 and PUSH being bit 0
0000 0100	Acknowledge: Any data value is acceptable	Any value written to this address should cause the DONEFLAG to be reset
0000 1000	disp(7:0)	Whatever value is stored here will be displayed on the FPGA’s 7-segment displays
0001 0000	leds(7:0)	*See Note below

* Any value on this output will be displayed on the discrete LEDs (above the switches). This can be used for 2 purposes: 1.) Debugging: You can add output statements in your software program to output a certain pattern to let you know you got to that point in your code. 2.) Operation Count: Count and display (in binary) the number of operations (SIZE, TOP, POP, or PUSH) performed by the user). This will help us know your software is operating correctly and can also be helpful in debugging.

Your final submission must display the operation count on the LEDs. See the software implementation below.

Now by placing an 8-bit register or flip-flop and adding logic that looks at the appropriate bits of PORTID as well as the write strobe ('wen') signal we can make the register only capture data when the processor outputs to that specified register. The data to be captured is available on the OUTDATA bits.

4. The Software Program

With your memory map in place and the basic concept of what hardware is needed to interface the processor to the heap engine, we can now write software to control it. Below is a C program that ***DESCRIBES*** what your assembly code should do. You may not want to implement it 1-for-1 but make the assembly code perform equivalently.

```
// In C, if we know the address of an I/O device in an embedded
// processor we can just make our own pointer to it.
//
// Pointers to inputs
#define BTNS (const unsigned char*) (0x00)
#define SWITCHES (const unsigned char*) (0x01)
#define DONE (const unsigned char*) (0x02)
#define DOUT (const unsigned char*) (0x03)
#define SIZE (const unsigned char*) (0x04)

// Pointers to outputs
#define DIN (unsigned char*) (0x01)
#define START (unsigned char*) (0x02)
#define ACK (unsigned char*) (0x04)
#define DISPLAY (unsigned char*) (0x08)
#define LEDS (unsigned char*) (0x10)

// Actual embedded code that you should translate to
// assembly in any *equivalent* manner.  It does not need
// to be a 1-to-1 translation (i.e. variables here can
// just be some register that you choose in the PicoBlaze).
unsigned char opcnt = 0;
unsigned char btnval;
while (true)
{
    ++opcnt;
    *LEDS = opcnt;  // display opcnt on LEDs
    while( (btnval = *BTNS) == 0x00) {}  // poll the buttons
    if(btnval & 0x08) // if SIZE op
    {
        *DISPLAY = *SIZE;
    }
    else if(btnval & 0x04) // if TOP op
    {
        *DISPLAY = *DOUT;
    }
    else
    {
        unsigned char myop;
        if(btnval & 0x02) // if POP op
        {
            myop = 0x02;
        }
        else if(btnval & 0x01) // if PUSH op
        {
            *DIN = *SWITCHES;
            myop = 0x01;
        }
        *START = myop;  // start the heap engine
        // Poll for completion of the operation
        while( *DONE == 0 ) {}
        *ACK = 1;   // acknowledge the operation to turn
                    // off the done flag
        *DISPLAY = *DOUT;  // show the top value
    }
    // one human button press can be long so we should wait
    // for the button press that started at the top of this
    // iteration to complete so that one press doesn't cause
    // multiple operations
    while( (btnval = *BTNS) != 0x00) {}

}
```

5. Select Instruction review

Below is an introduction to some of the Picoblaze assembly instructions. More info can be found in the Picoblaze PDF posted with this assignment.

Instruction	Description
COMPARE sX, YY (YY = 8-bit hex const.)	Compares the number in the sX register with the constant YY by subtracting and determines if the result is Zero or NotZero (to be followed by an appropriate Jump)
COMPARE sX, SY	Compares the number in the sX register with the number in the sY register by subtracting and determines if the result is Zero or NotZero (to be followed by an appropriate Jump)
AND sX, YY (YY = 8-bit hex const)	Performs bitwise AND of sX and YY. **Sets the Z flag if the result of the ANDing is 0.**
JUMP Z, Label	Jump to the instruction with the given Label if the previous COMPARE result was Zero
JUMP NZ, Label	Jump to the instruction with the given Label if the previous COMPARE result was NotZero
LOAD sX, YY (YY = 8-bit hex constant)	Loads the 8-bit constant YY into the register sX
INPUT sX, YY (YY = 8-bit PORT ID)	Outputs the address YY on the PORTID output of the Picoblaze processor and then captures whatever data is present on the INDATA input placing it in sX
OUTPUT sX, YY (YY = 8-bit PORT ID)	Outputs the address YY on the PORTID output of the Picoblaze processor and the data in register sX on the OUTDATA bits of the processor. It also asserts the write strobe ('wen') signal of the processor for a clock signal to indicate when your logic should capture the output data.

6. Compiling your Picoblaze Assembler

The Picoblaze assembler (i.e. converts your human-readable assembly description to a hardware-level description given as a .vhd file) is done by a Python script. Rather than making each person install python and get the system working we instead have provided a web-service to do this for you. If you simply go to:

http://bits.usc.edu/codedrop/?course=ee209-sp17&assignment=pasm&auth=Google#

and login with your USC credentials you can upload a .psm file, click "Check my submission" and scroll down to the results. If successful, you should see a link to the output `heap2_prog.vhd` file. You can then save that file to your Xilinx project directory, replacing the default one provided to you in the skeleton project. If the assembly file has syntax errors they should be shown to you on the webpage. Try to fix the problem and re-upload the file and repeat the process.

4 Prelab

Ensure your part1 of the heap engine is working before you attempt part 2.

5 Procedure

Be sure you have read the Background Notes and Information before you start this project.

1. Download the skeleton project: heap2.zip and extract it to a folder.
2. Find your 'ctrlpush.v' and 'ctrlpop.v' files and all other Verilog files (reg8e.v, dff1s.v, …, etc.) and copy them from the Project Part 1 folder and copy it to your 'heap2' project folder. ****DO NOT COPY the heap.v**** file as we have provided a slightly update version of this file that adds logic to produce the DONE signal when a push is performed on a full heap or a pop on an empty heap. This will be necessary for your Picoblaze software to execute correctly. <u>**So please ensure you do not overwrite this new version of heap.v with your old version**</u>.
3. Then open the 'heap2' project, go to the Project menu and click "Add Source". Be sure you navigate to your heap2 project folder that you just extract the project and find all your Verilog (.v) files and add them to the project.
4. In heap_soc.v ('student_design') we have provided the Picoblaze processor component, the program memory, and an instance of your heap engine. You should now add the peripheral registers needed to receive output from your Picoblaze processor and provide input to your heap engine. You will need to ensure the registers are only enabled when the desired port ID (address) is output by the Picoblaze. Add the muxing necessary to read values from your heap engine or the buttons and switches into your processor.
5. Write the assembler program 'heap2_prog.psm' that will control your square root engine. It is located in your project folder. Edit it in an editor and then assemble it on our provided website and download the generated heap2_prog.vhdl file into your 'heap2' project folder (replacing the previous one).
6. Change to 'Simulation' view and use our provided testbench program to simulate your design. We provide tasks (functions) to perform the various

operations. In the stimulus input block, please add calls to exercise various sequences of operations. We have started this for you by calling a push of the value 7 and then a size operation. Feel free to add or alter the testbench as needed.

7. Synthesize, Implement, and generate the programming file for the FPGA. During lab office hours download your .bit file to the FPGA and examine the operation on the FPGA by pressing the any of the four push buttons. The output 7-segment displays should be alternating display of the input X and the output DISP value (at around 1 second each).
8. Demo to your TA & submit your .psm file and other indicated files [heap_soc.v, heap_soc_tb.v, heap2_prog.psm, heap2_prog.vhdl, and heap.v, ctrlpush.v, ctrlpop.v] using the provided link from our course website page.

6 EE 209 Project Part 2 Grading Rubric

Student Name: ____

TA Initials of DEMO: ______________________________

Item	Outcome	Score	Max.
Periphery Logic			
• Correct implementation of done input muxing	Yes / No		1
• Correct implementation of dout input muxing	Yes / No		1
• Correct implementation of size input muxing	Yes / No		1
• Correct implementation of btns input muxing	Yes / No		1
• Correct implementation of switches input muxing	Yes / No		1
• Correct implementation of din output	Yes / No		1
• Correct implementation of push/pop output	Yes / No		1
• Correct implementation of ack output / doneflag reset	Yes / No		2
• Correct implementation of disp output	Yes / No		1
• Correct implementation of leds output	Yes / No		1
ASM			
• Assembly file wait for a button press	Yes / No		1
• Assembly file correctly performs SIZE	Yes / No		1
• Assembly file correctly performs TOP	Yes / No		1
• Assembly file correctly performs PUSH	Yes / No		2
• Assembly file correctly performs POP	Yes / No		2
Test Cases			
• Works for instructor test cases	___ / 6		6
SubTotal			25
Late Deductions (-5 pts. per day)			
Total			25
Open Ended Comments:			