EE 209 Project Part 1 – Heaping it on

1 Introduction

In this project you will complete the control unit and datapath for a HW-based priority queue utilizing a heap data structure. You should work on this lab INDIVIDUALLY!

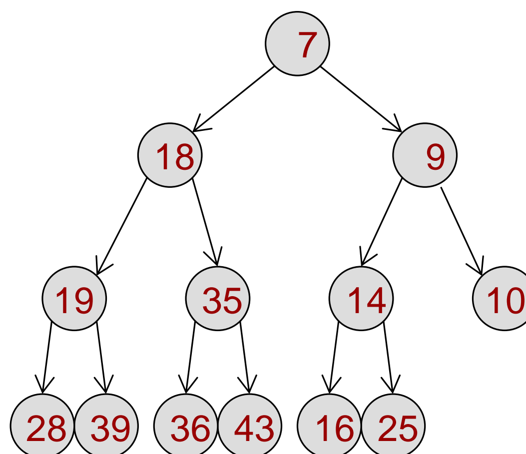
2 What you will learn

This lab is intended to teach you how to implement state machines, use datapath components and perform a non-trivial digital design.

3 Background Information and Notes

1. **The Data Structure**

   A priority queue is a common data structure that allows values to be inserted in any order but only allows retrieval and removal of the smallest element in the priority queue. While there are many implementations for a priority queue, one of the simplest and most efficient is a **binary heap**. A binary heap is a binary tree that stores the values in the priority queue, but also guarantees that the "heap"property is maintained for all nodes. The **heap property** states that: *a parent must be less than both of its children*, but no ordering constraint exists between the two children. If that property holds then the smallest element must live at the root (top) of the binary tree and can easily be accessed when needed. However, when inserting and removing new values we must take care to maintain the heap property.



A binary heap. Each parent is smaller than both children.

Push: Push is a fancy name for adding an element to the heap. When a new element is added we ALWAYS add it to the bottom left-most free location in the tree. By always placing it there, it may be the number is smaller than its parent. So, what we will do is repeatedly compare the new child with its parent and if it is

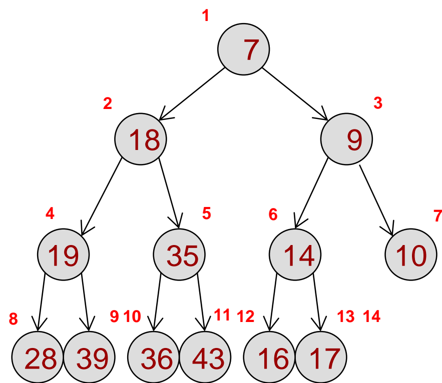
smaller swap the two. The child can continue to move up and swap places with its parent until it reaches the root of the tree or until it encounters a parent that is smaller than the child. See the website link below for visualization purposes.

Pop: Pop is a fancy name for removing an element from the heap. When we perform a pop operation we ALWAYS remove the top/root element. With it gone we need someone to take its place. So, though it may seem strange at the moment, we actually take the bottom-left most value in the tree (which is likely a "large" number and consider it to be the new root. That would likely violate the heap property and so what we will do is compare it to its smallest child (whichever is smaller of its left and right child). If the smallest child is less than the parent we swap their locations and repeat the process on the value that got demoted (i.e. the old parent). This process of comparing a parent to its smallest child and swapping them can continue until the node has no children (i.e. is a leaf node) or until neither of its children are smaller than it.

Storing the heap in an array: Finally, to store the tree it turns out we can use a simple array (and thus memory device for our logic implementation). For a binary tree we will place the root at location 1 in the array/memory (leaving location 0 empty and never using it). By placing the root at location 1 in the array/memory it allows simple arithmetic to find the location of a node's left or right child or its parent. Given a node at index, *idx*, we can find its children and parent by applying the following formula:

$$parent(idx) = idx/2$$
$$left(idx) = 2*idx$$
$$right(idx) = 2*idx+1$$

Below is a sample heap and you can verify the formula.



0	1	2	3	4	5	6	7	8	9	10	11	12	13
em	7	18	9	19	35	14	10	28	39	36	43	16	17

Parent(5) = 5/2 = 2
Left(5) = 2*5 = 10
Right(5) = 2*5+1 = 11

At this point if you feel unsure of how a heap would work we strongly encourage you to play with this visualization tool:
https://www.cs.usfca.edu/~galles/visualization/Heap.html

Finally, the push and pop pseudocode is described below. (Assume a '*size*' variable is maintained from operation to operation and is initialized to 0 upon reset/startup.

```
Push(val)
// precondition:
//    the heap is not full
-----------------------------
idx = ++size;
// while a parent exists
while( idx > 1 ){
   parent = M[idx/2];
   // check if we should
   // promote val
   if(val < parent){
      M[idx] = parent;
      idx = idx / 2;
   }
   else break;
}
// Place val in its correct
// location
M[idx] = val;
```

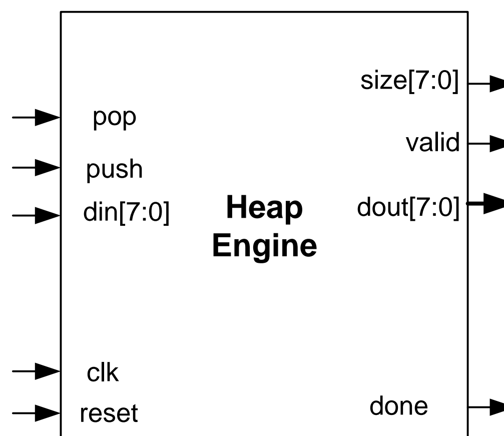
```
Pop()
// precondition:
//    the heap is not empty
-----------------------------
parent = M[size--];
idx = 1; cidx = 2;
// while left child exists
while(cidx <= size){
  child = M[cidx];
  // see if right child exists
  // and is smaller
  if(cidx+1 <= size){
    if(M[cidx+1] < child){
      child = M[cidx+1];
      cidx = cidx + 1;
  } }
  // see if the smaller child
  // is less than the parent
  if(child < parent){
    M[idx] = child;
    idx = cidx;
    cidx = 2*cidx;
  }
  else break;

}
M[idx] = parent;
```

2. Block Diagram

Our heap implementation will allow storage of up to 255 unsigned 8-bit numbers. The next page shows the high level block diagram of the system you are to design. We will provide a good deal of support logic that will allow you to design the push and pop controllers separately while our logic handles interfacing the two controllers to the actual memory the array is stored in. In addition, our logic will provide the 'size' counter and you can provide a simple 'increment' or 'decrement' signal to us. Essentially, you can break your design into two pieces: the push logic and the pop logic, design them separately and our logic should handle the rest. The key task of your controllers is to supply the correct signals to the control the array/memory.

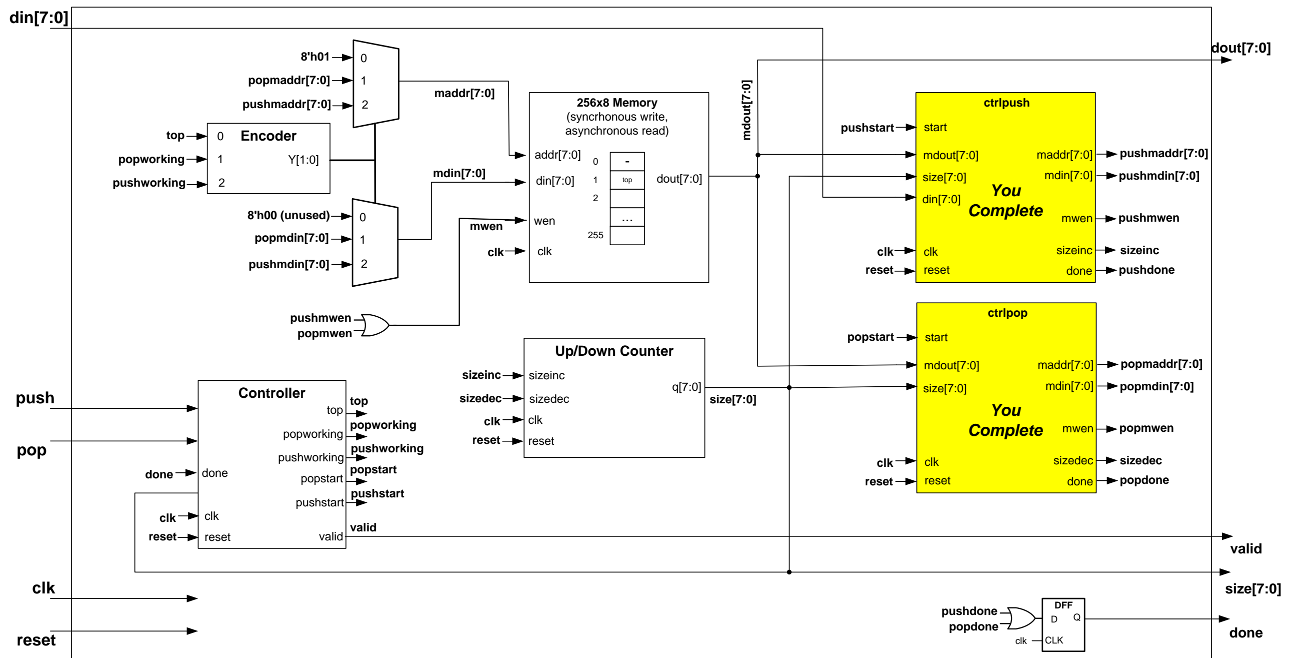
The primary inputs and outputs of the system are described below:

- **push** and **pop**: Control signals which should be high for a clock cycle to start the respective operation.
- **din:** For push operations a data value input (**din**) should be given with the new number to be added to the heap.
- **dout:** When not processing a push/pop operation this should output the top/smallest number stored in the heap; during a push/pop operation dout is arbitrary/undefined .
- **done**: Should be high for 1 cycle once a push/pop operation is completed.
- **size:** The current number of elements in the heap.
- **valid**: A signal indicating the **dout** number is valid. This signal will be false when there are no numbers in the heap or while a push/pop operation is in process; it will be true otherwise. This is really a convenience signal for you when looking at waveforms. We could remove it and just realize that the time between when we assert **push/pop** and the assertion of the **done** signal will yield undefined **dout** values.



3. The Provided Interface Logic

To make the design more manageable we have split the push and pop controllers into separate subdesigns and then provided all the high-level control/interface logic needed to share the memory and produce the combined outputs. In this way you can design each subcontroller (push and pop controller) separately with the idea that they fully “own” the memory and our provided logic will multiplex appropriately. Thus as you design each subcontroller, assume your inputs and outputs (mdout, maddr, mdin, and mwen) are solely under your control.

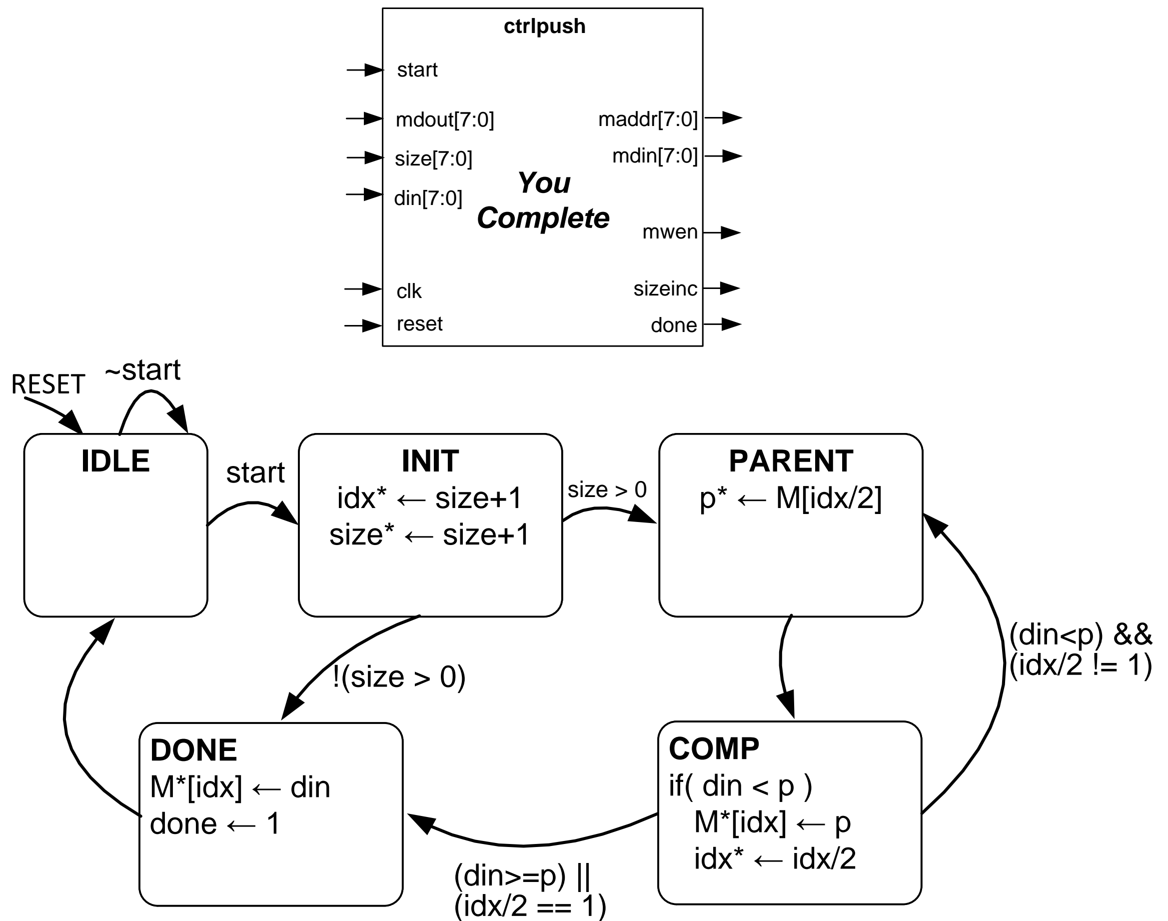


The memory where we will store are array/heap of data is a 256 row by 8 column memory. It is a "single-ported" memory meaning it can only perform one operation per cycle (either read or write but not both simultaneously). The memory allows asynchronous read (value is available on the same clock as we supply the address) but synchronous write (i.e. write takes place on the trailing clock edge of the cycle in which we assert **mwen**. The memory will perform **writes** when **mwen=1** and **reads** when **mwen=0** (by default).

The size counter is an 8-bit up/down counter that will perform Q*=Q+1 when **sizeinc** is asserted and Q*=Q-1 when **sizedec** is asserted. You should not assert both sizeinc and sizedec at the same time. The size count can be used to not only indicate how many elements are stored but also as the address/index where the last element is located. This will be useful for the push and pop operations.

4. The Push Controller

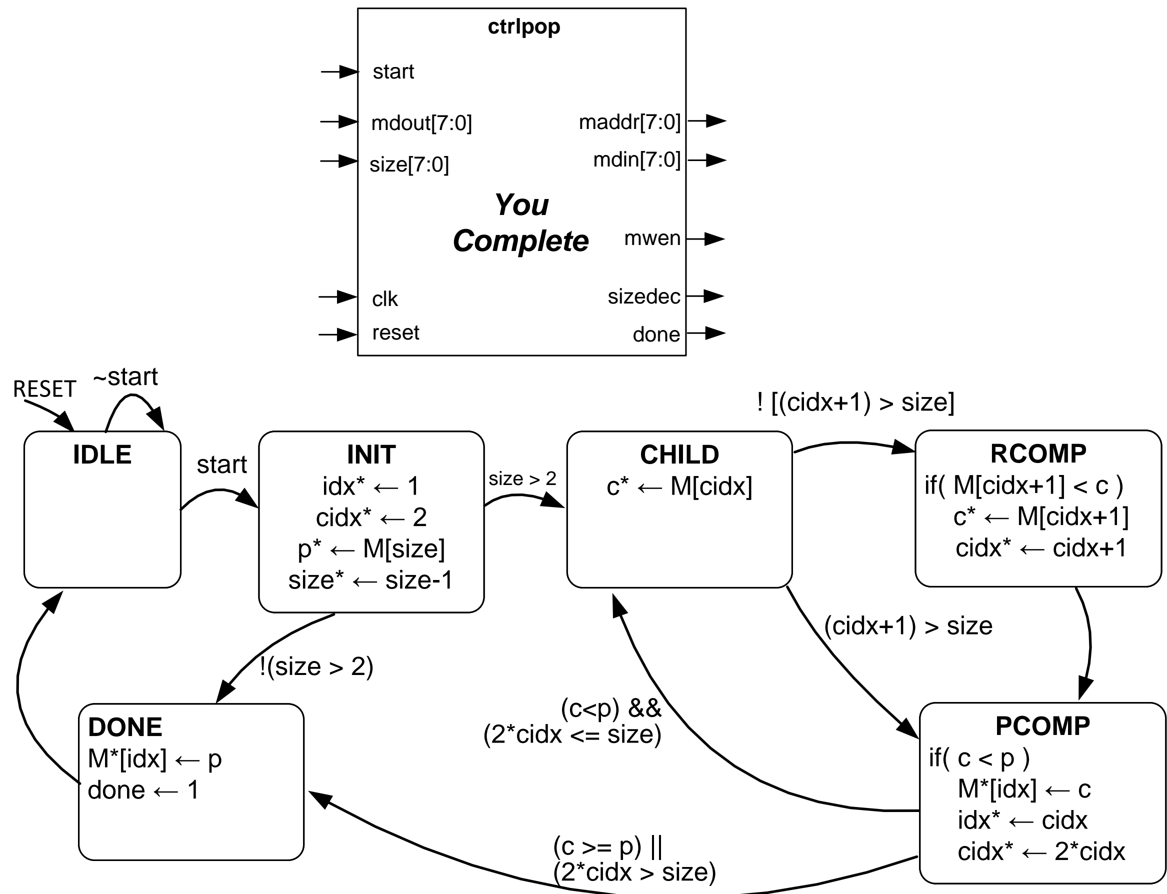
The push controller should take a new data value (**din**), write it to the end of the memory array and then perform the process of moving it up the tree if it is less than its parent. The main issue in doing this is the memory allows only one operation (read or write) per clock. So, we need some state machine/sequencing to order those operations. Similarly, we'll need an address/index (to track the current location and its parent location. Finally, we'll need to compare **din** to the parent at each step so we'll need a register to store the parent. A state diagram showing the major register, counter, and memory operations is shown below.



In the above state diagram, realize that M[*loc*] refers to the value of the memory at address, *loc*. IDX is a value you will maintain as the current address/index of the added element. P is a register that can store the *value* of the parent item (not its index which is easily derived as idx/2). Remember that **size** is maintained in the logic we provided and not in your push controller. Thus, you only need to assert the **sizeinc** output which will cause the size counter to increment on the next cycle. You will need to study the state diagram and the operations performed in each to derive a datapath for your push controller and then implement it in Verilog in `ctrlpush.v`. You will also implement the state machine controller (for the above state diagram) in the same file.

5. **The Pop Controller**

   The pop controller should copy the last value in the array over the value to be removed at address/index 1 of the memory. From there it should iterative move that value (a.k.a. the parent) down swapping it with its smallest child if in fact that child is smaller than the parent. A state diagram showing the major register, counter, and memory operations is shown below.



In the above state diagram, realize that M[*loc*] refers to the value of the memory at address, *loc*. IDX is a value you will maintain as the current address/index of the "parent" element. P is a register that can store the *value* of the parent item. CIDX is a value that should store the index/address of the smallest child and c is a register that can store the value of the smallest child. Remember that **size** is maintained in the logic we provided and not in your pop controller. Thus, you only need to assert the **sizedec** output which will cause the size counter to decrement on the next cycle. You will need to study the state diagram and the operations performed in each to derive a datapath for your pop controller and then implement it in Verilog in `ctrlpop.v`. You will also implement the state machine controller (for the above state diagram) in the same file.

6. **Helpful Verilog Notes:**

Concatenation: To concatenate different sets of bits to form a large bus simply place the signals in { … } separate by commas (i.e. assign X = { A[3:0], B[3:0} would create an 8-bit signal with X[7:4] provided from A and X[3:0] provided from B. You can also concatenate constants (e.g. `x = { A[3:0], 4'b0000 }`).

Multiply and Dividing by 2: One can multiply or divide a number X by 2^n by simply shifting X left or right (respectively) by n-bits. Think how you can use the concatenation Verilog tip above to easily produce 2*X or X/2.

Comparison and addition: Rather than using dedicated adders and comparators we can use the Verilog operators in our assign statements. For example if we want a signal to indicate true/false if (X+1) > 2 we could simply declare a wire such as 'xp1_gt_2' and then use an assign statement like:
`assign xp1_gt_2 = ((X[7:0] + 1) > 2)`. The synthesis tool will infer that an incrementer and comparator are needed. This should save you from having to wire up a lot of adders and comparators. Though we have provided an 8-bit adder and comparator module in `adder8.v` and `comp8.v`.

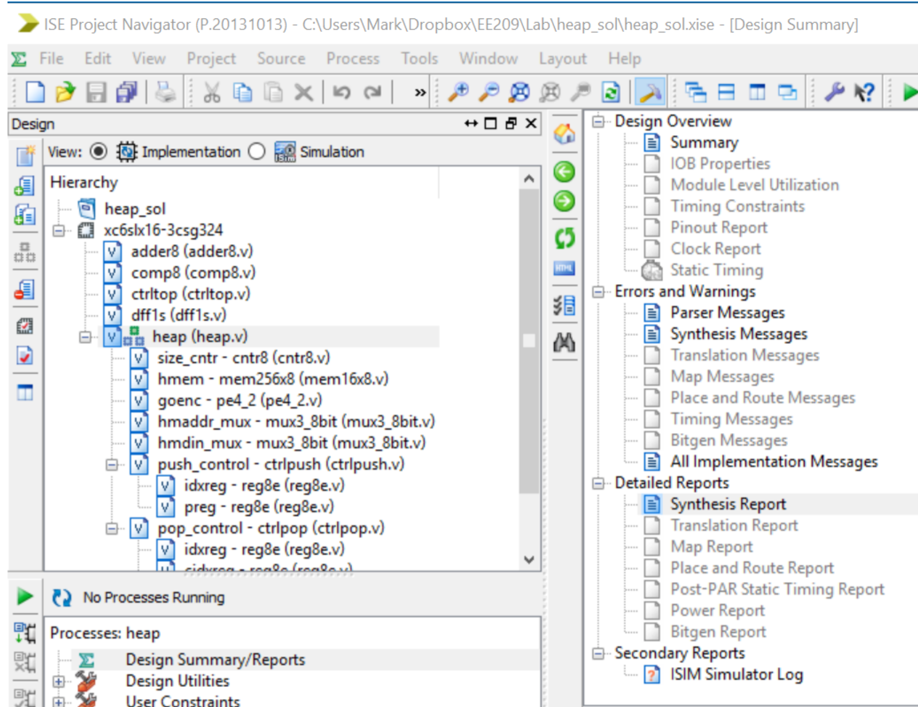
Registers & DFFs: We provide an 8-bit register (`reg8e.v`) and a DFF (`dff1s.v`) for you to use. Tip: Use a 1-hot state machine approach to design the control unit FSMs.

4 Procedure

You will design and implement the push and pop controllers to complete the overall priority queue / heap engine. We have provided the start of a testbench with functions (aka 'tasks' in Verilog) that will generate the input/output sequence needed to push or pop a value from the heap engine. For part 1 we will only use simulation to ensure a working design (no implementation on the FPGA board).

1. Download the heap1 project .zip. Extract the files to a folder.
2. Examine the top-level logic in `heap.v`
3. Implement the datapath and FSM control for the push controller in `ctrlpush.v.`
4. Implement the datapath and FSM control for the pop controller in `ctrlpop.v.`
5. If you need to add modules not provided, just add their definition in ctrlpush.v or ctrlpop.v.
6. Understand and then update the provided testbench `heap_tb.v` to simulate your design. We suggest testing cases where values are placed in different orders, performing pushes and pops when the last item is a left child and when it is a right child, interspersing pushes and pops, making the heap go empty by popping values and then adding more. Verify the correctness of your design. **HINT: First build the push controller and test that (make the pop controller outputs 0s) first to ensure you get it working before building and testing the pop controller.**

7. Once you are satisfied your design works, go back to the ‘Implementation’ view, select your top-level `heap.v` file, and synthesize your design. Ensure it synthesizes without errors (warnings are okay). Then, at the top of the processes pane, click on “Design Summary/Reports” and in the right window select “Synthesis Report” (see the screen capture below). Scroll through and read the output of this report. In particular, look at the “Advanced HDL Synthesis” about 1/2 of the way down and look at the blocks/macros that are used. Continue to scroll and find the ‘Device Utilization summary’ which indicates the size/area of your design (i.e. how much of the FPGA’s resources your design is using). Below that will be the timing report which will list the critical path and its delay (which sets the frequency at which you could run this design). Answer the questions in the review section below through your examination of these areas.



8. Submit your design files: ctrlpush.v, ctrlpop.v, and heap_tb.v as well as your answers to the review questions below (in a file name **`review.txt`**) via the link on the website.

5 Review

1. In the synthesis report, how many flip-flops does your design use? How many adders/subtractors are used? How many comparators are used? In the device summary section, how many slices LUTs are used? What percentage is that number vs. the total available on the chip?
2. Examine the timing report. What is the minimum period and maximum frequency at which we can run this design?

6 EE 209 Project Part 1 Grading Rubric

Student Name: ______________________________

Item	Outcome	Score	Max.
Push Correctness • FSM NSL is correctly described • Datapath is correctly described	 Yes / No Yes / No		 1 1
Pop Correctness • FSM NSL is correctly described • Datapath is correctly described	 Yes / No Yes / No		 1 1
Instructor Test Cases • Test Case 1 (Separate) • Test Case 2 (Integrated)	 Yes / No Yes / No		 2 3
Review Questions are accurate.	Yes / No		1
SubTotal			10
Late Deductions (-1 pts. per day)			
Total			10
Open Ended Comments:			