EECS 151/251A Homework 10

Due Wed, April 24$^{\text{th}}$, 2024

Introduction

This homework is meant to test your understanding of parallelism and HW accelerator design. There are five total questions. Please check Ed first if you have any questions.

Problem 1

We would like to evaluate the list processor architectures in the lecture slides using the FOM (figure of merits). The FOM here is defined as:

$$\frac{\frac{\text{Nodes}}{\text{Cycle}} \cdot F}{\sqrt{\text{Cost}}}$$

For this problem, ignore the delay and cost of control logic. Use 16-bit components for the sum part (15-bit is enough though).

component	delay (ns)	cost
N-bit register with CE	$t_{clk-q} = t_{setup} = 0.5$	$32N + 4$
N-bit multiplexer	1	$3N + 12$
N-bit adder	$2\log_2(N) + 2$	$8N - 1$
Memory	10 for async read	0
N-bit zero comparator	$0.5\log_2(N)$	$4N - 4$

1. Draw a circuit diagram for an 8-bit zero comparator based on the delay and cost in the table.
2. Justify the delay and cost equations for each component (i.e explain why could these be realistic based on our FOM).
3. Formulate the FOM in terms of NPC (nodes per cycle), T (clock period), and the number of components shown below.

component	number
8-bit register with CE	a
16-bit register with CE	b
8-bit multiplexer	c
16-bit multiplexer	d
8-bit adder	e
16-bit adder	f
8-bit zero comparator	g

4. Fill out the following table for the list processor architectures in the lecture slides.

Architecture	NPC	T (ns)	a	b	c	d	e	f	g	FOM
1										
2										
3										
4										

5. Would it help if nodes are aligned in memory for the first architecture? The address of the next node is always stored at an even address. Explain why or why not.

6. Design a list processor with 32×16 memory and maximize the FOM. The memory stores the value and the next address of a node at the same address and has 10 ns acync-read delay. Use 8-bit components for the address part. No need to design a control logic. What is the maximized FOM of your design?

Solution:

1. If you compare with Zero, you essentially want to do a 8-bit NOR (only output 1 if all of the bits are 0). OR is commutative, so you can do a tree structure of ORs (NORs + inverters) with just a NOR at the end. This will be logarithmic with respect to the delay, and linear with respect to the time.

2. In this case, the cost is proportional to the number of transistors. An N-Bit register is essentially 2N Flip FLops, which is linear in N. An N-bit multiplexer has constant delay (if using transmission gates) and is linear in the logic. An N-bit Adder can be made using a Carry Look ahead logic, which is logarithmic in the delay and linear for the number of components (tree of adders). Memory is off-chip so there is no cost, and an N-bit zero comparator, as made in part 1, is logarithmic in delay and linear in the number of transistors.

3. Cost = 260a + 516b + 36c + 48d + 63e + 127f + 28g
   So, the FOM is :

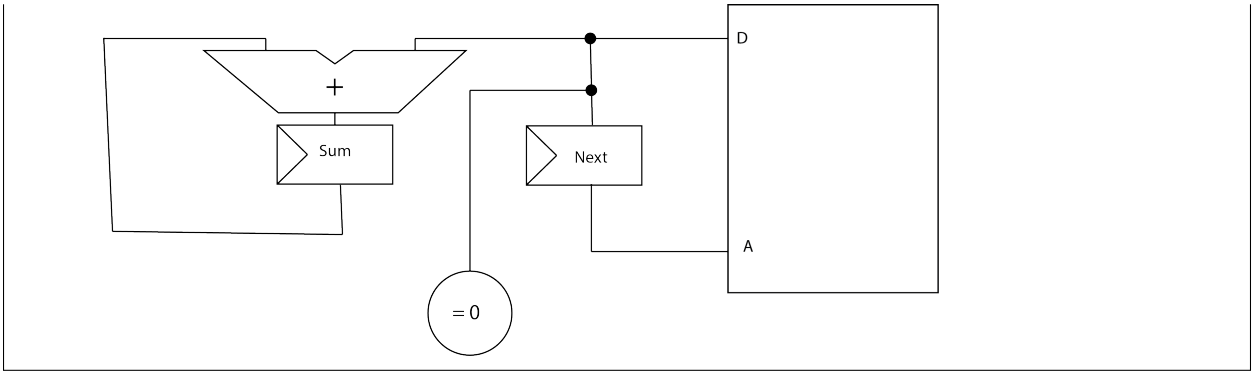
$$\frac{\frac{\text{Nodes}}{\text{Cycle}}}{\sqrt{260\text{a} + 516\text{b} + 36\text{c} + 48\text{d} + 63\text{e} + 127\text{f} + 28\text{g}} \cdot T}$$

4. The table looks like:

Architecture	NPC	T (ns)	a	b	c	d	e	f	g	FOM
1	0.5	31	1	1	2	1	1	1	1	483243
2	0.5	23	2	1	3	1	1	1	1	578938
3	0.5	23	2	1	3	2	0	1	1	582042
4	0.5	13	3	1	5	2	0	1	1	925508

5. Yes, it would help because the NUMA addition can be eliminated.

6. We can load both the NEXT and data in one load
   {NEXT, X} ← Memory[NEXT], SUM ← SUM + X;
   In this case, we can do the entire load and sum in one cycle, so our NPC is 1. Our critical path is equivalent to one register read, one memory access, one 16-bit sum, and one register write = 20ns. We no longer have an 8-bit adder or any multiplexers. We still need the zero comparator, 8-bit register, 16-bit register, and 16-bit adder. So our FOM is 1638683, which far surpasses even the fastest 8-bit rd port processor.


+
Sum
Next
D
A
= 0

Problem 2 - 251 only, Extra Credit for 151

For this problem, you will design HW to implement the **add** function for an 8-bit Binary Search Tree. A Binary Search Tree is an ordered data structure comprised of nodes. Each nodes has at most 2 children. For every node n, the right child of n contains data that is greater than the data at n, while the left child contains data that is smaller. **add** takes as input an 8-bit input, *data*, and traverses through the tree nodes to place the data point. At every node, if the data is greater than the data at that node, you will then visit the **right** child node of that node. If the data is than the data at that node you will travel to the **left** child node of that node. If you receive data that is equal to the current node, then you should not add the data. If the node you wish to travel to does not exist, then you will get a free address and create a new node there. In order to get a free address, you will have access to a separate data structure, **free list**, which will be a Linked List (similar to the one discussed in lecture). You should return whether or not you added the data to the tree.

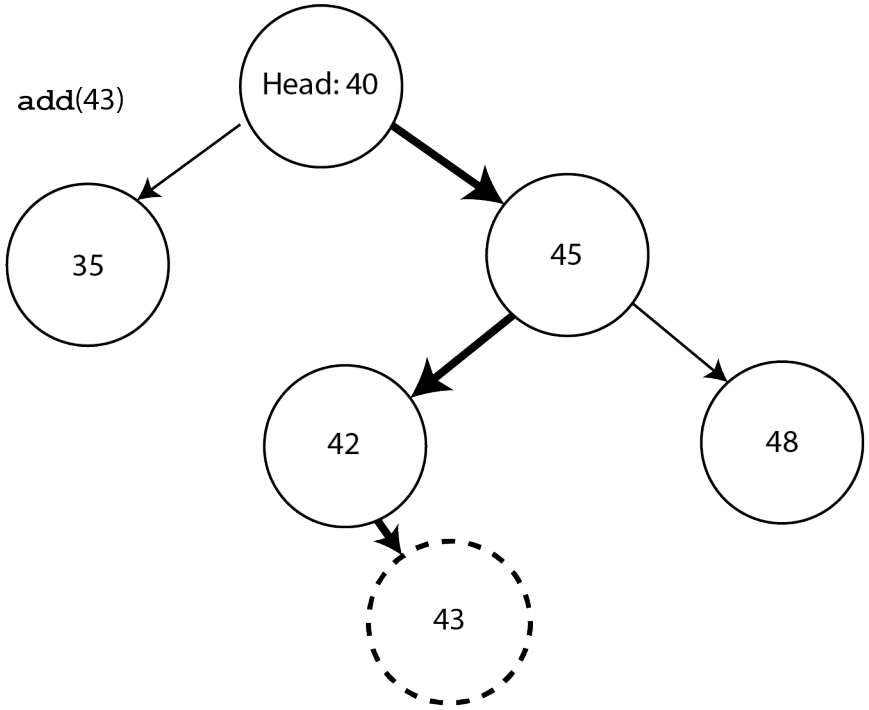


Figure 1: Adding the number 43 to the Binary Search Tree. You first start at the head, and since 43 is greater than 40, you travel to the right child node which has 45. Then you travel to 42, which has no right child. You then must create a new node by getting the free address, assigning that memory location to 42's right child pointer, and then setting the data of the corresponding node to 43. You also return a 1 to signify that you added a new node.

Here are some things to note:

1. You are given a 256 x 8-bit memory with 1 8-bit read port and 1 8-bit write port.
2. Assume there is already some data in the tree.

3. Each node in the free list is 2 8-bit words. Word 0 contains the address of the next element in the free list. Word 1 contains the data (free address available for use).
4. A Binary Search Tree node has 3 words. Word 0 has the address of the right child node, while Word 1 has the address of the left child node. Word 2 contains the data in the node. You can assume a child node does not exist if the corresponding address is 0.
5. The free list points to chunks of memory where 3 words exist contiguously. You do not have to worry about blocks not being big enough to fit a full tree node.
6. Adding a tree node means that you do the following:
   (a) You set the corresponding left/right pointer of the parent node to the address you receive from the free list.
   (b) You create a new node at the correct address and zero out the left and right child pointers.
   (c) You add the inputted data as the data of the node you just created.
   (d) You update the pointer to the free list and return.

Please do the following:

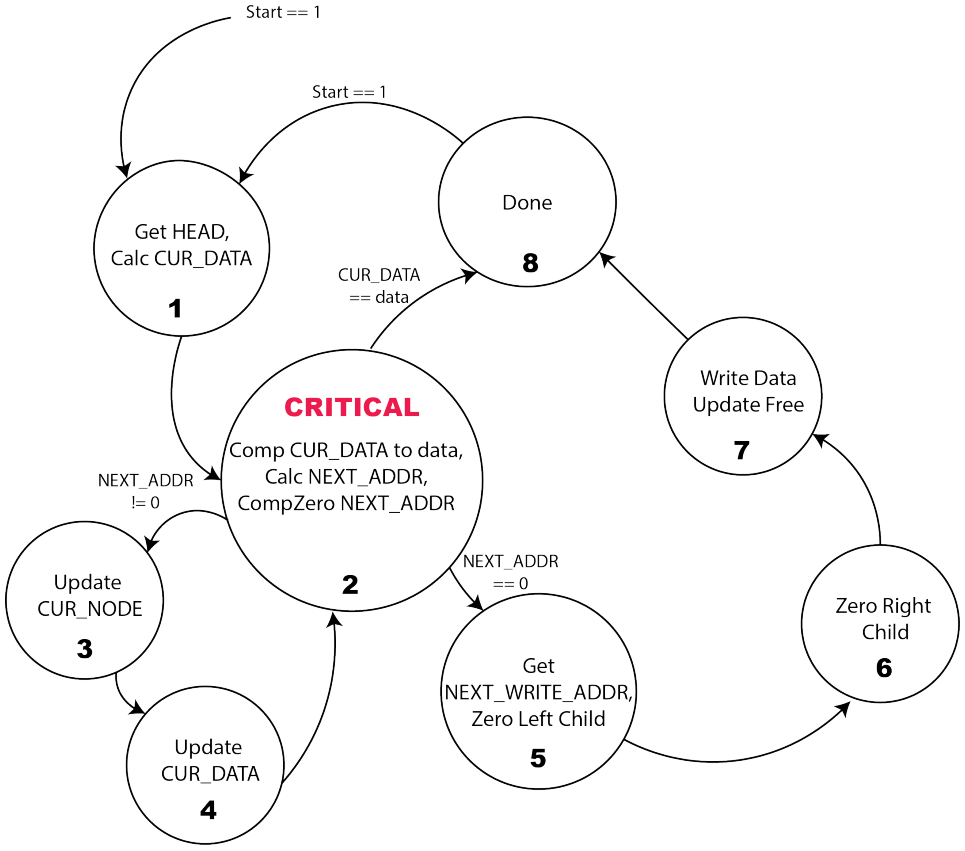
1. Write the **add** function as described using RT notation. You can use assume the current address of the free list pointer is stored in a register *FREE*. The head of the Binary Search Tree is stored in a register *HEAD*. Assume that you start working when the START signal is high, as we did in lecture. You should have one loop, designated by a repeat block. Make sure to maintain the correctness of the free list.
2. What is the minimal number of states you need for the Finite State Machine for the Controller (keep in mind what all you can do in a single cycle!)? Draw a high level Finite State Machine (In each state, write what operations are happening, explain what outcomes lead to what transitions). Make sure you draw the diagram clearly.
3. What operations from the critical path could be moved to a different state? If you were to move those operations, what extra HW do you need?
4. Given the following sequence of adds to the tree in Figure 1:
   add(20), **add**(20), **add**(20),**add**(25), **add**(35)
   Which of the following architectural changes would most significantly improve the performance in terms of the average **latency** (in terms of cycles) of an an **add** request?
   (a) Adding 1 read port
   (b) Adding 1 write port
   (c) Modifying the read and write ports to be 24 bits wide (3 words)

Solution:

1. `If (START == 1) {`
   `DONE, WROTE <- 0;`
   `CUR_NODE <- HEAD;`
   `CUR_DATA <- MEM[HEAD + 2];`

```
repeat {
if (data < CUR_DATA){
NEXT_ADDR <- MEM[CUR_NODE];
}
else:  NEXT_ADDR <- MEM[CUR_NODE + 1];
if (NEXT_ADDR == 0) {
    NEXT_WRITE_ADDR <- MEM[FREE + 1];
    MEM[NEXT_WRITE_ADDR], NEXT_WRITE_ADDR + 1] <- 0;
    MEM[NEXT_WRITE_ADDR + 2] <- data;
    FREE <- MEM[FREE];
    WROTE, DONE <- 1;
}
CUR_NODE <- MEM[NEXT_ADDR];
CUR_DATA <- MEM[CUR_NODE + 2];
} until (DONE == 1);
}
```

2. You will need 8 states including the DONE state. You can only do 1 memory read and write per state which limits how much you can do.



3. The Critical stage is labeled in the FSM. In this stage, the critical path is an 8-bit comparator (note that you are doing less than and equal to, this will be two comparisons but they can be done in parallel), an addition (CUR_NODE + 1), a Memory read, and then finally a zero comparator. One thing we can do is move the CUR_DATA comparison into the 2 states that lead into it. This requires one extra 1-bit register to keep track

of the comparison output (note that if the data is equal to CUR_DATA, you can move directly to the DONE state). If this stage is still the critical path, we can move the CUR_NODE + 1 calculation into the two previous states as well at the cost of an 8-bit register to hold the output of the addition. Note that this will probably not help as it will make one of the other states you are moving into slower.
You can draw the FSM differently to have a different critical stage, answers will vary.

4. The latency in cycles of an add request corresponds to the number of states visited. The first add inserts a 20 as the left child of the 35 and visits 9 nodes (takes 9 cycles). The next two adds both take 9 cycles as the 20 now is in the tree. The 25 adds as the right node of the 20, and it takes 12 cycles. Then, the 35 takes 6 cycles since it is in the tree as well. The average number of cycles is 9 cycles.

   (a) If we add a second read port, we can move state 3 and 5 into state 2. Now the instructions take 7, 7, 7, 9, and 5 cycles respectively, which brings us an average of 7 cycles per instruction.

   (b) If we add a write port, we can combine state 6 into state 7. This will save us 1 cycle every time we write a node, which is in 2 of the instructions. This brings our average to 8.6 cycles per instructions.

   (c) If we can write 3 bytes at a time, this will help us combine Zero Left Child, Zero Right Child, and Writing the Data into a single state. Reading 3 bytes at a time can help remove some addition, but will not save us any cycles. Therefore, we save one state again every time we write, which gives us 8.6 cycles per instruction.

Problem 3:

You will be designing a 2-way set associative cache using a single read and write port RAM. If both ways are full, the cache always kicks out the data from Way 0. If the data is dirty (has been written to by the CPU), then the data is written to memory. The cache has 16 lines (8 lines per set) which are 128 bits wide. Both memory address and data words are 32 bits wide. The interface between the CPU and cache is shown below.

Between CPU and cache:

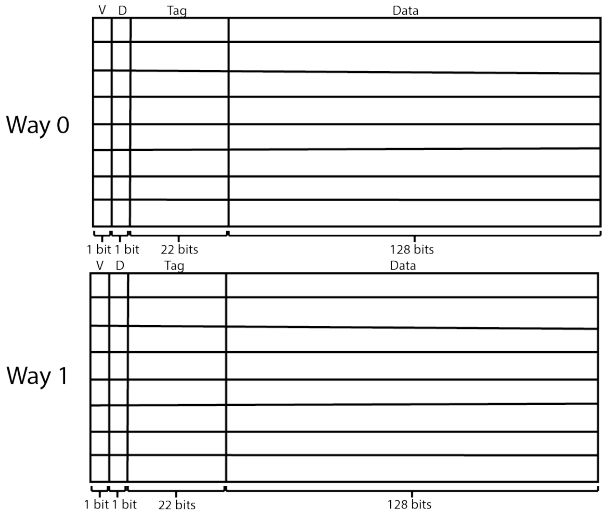
- Read
  - CPU waits til `cache_ready` is high.
  - CPU sets `ren` to 1 and `addr` to the source address.
  - Cache reads those signals immediately. The cache then checks both ways and set `valid` to 1 and `dout` to the data in case of a cache hit.
  - On a miss, the caches sets `cache_ready` to low and checks if both ways are valid. If both are valid, then the cache must perform a memory write with the data from way 0 if it is dirty. If both are invalid, you may read into either way. If one is invalid, you must read into that way.
  - The cache then performs a memory read into the appropriate way.
  - When memory returns the data (`mem_valid` is 1), cache sets `valid` to 1 and `dout` to the read data. The cache additionally set `cache_ready` high again.
  - Cache stores the tag and data at the next positive clock edge.
- Write
  - CPU waits til `cache_ready` is high.
  - CPU sets `wen` to 1, `addr` to the destination address, and `din` to the data to write.
  - On a hit, cache stores the tag and the data. It also sets the dirty bit to 1.
  - On a miss, the caches sets `cache_ready` to low and checks if both ways are valid. If both are valid, then the cache must perform a memory write with the data from way 0 if it is dirty. If both are invalid, you may read into either way. If one is invalid, you must read into that way.
  - The cache then performs a memory read into the appropriate way (must read the the rest of the block) and then writes the data. It must set the corresponding dirty bit high.
  - When the cache is done writing the data, it sets `cache_ready` high again.

1. Draw the structure of the cache. Make sure to show the two ways, the tags for each line, as well as any additional flag bits needed. How big (in bits) is the full cache (include the size of the tag, flag bits, and data)?
2. Draw the FSM for the cache controller. Assume the memory responds with a `mem_ready` signal on a memory read. Caches do not need to wait on a memory write. Assume to read from memory you set the `mem_rd` signal high (and then set it low on receive) and to write to memory you set `mem_write` high (and then you can set it low on the next clock signal).

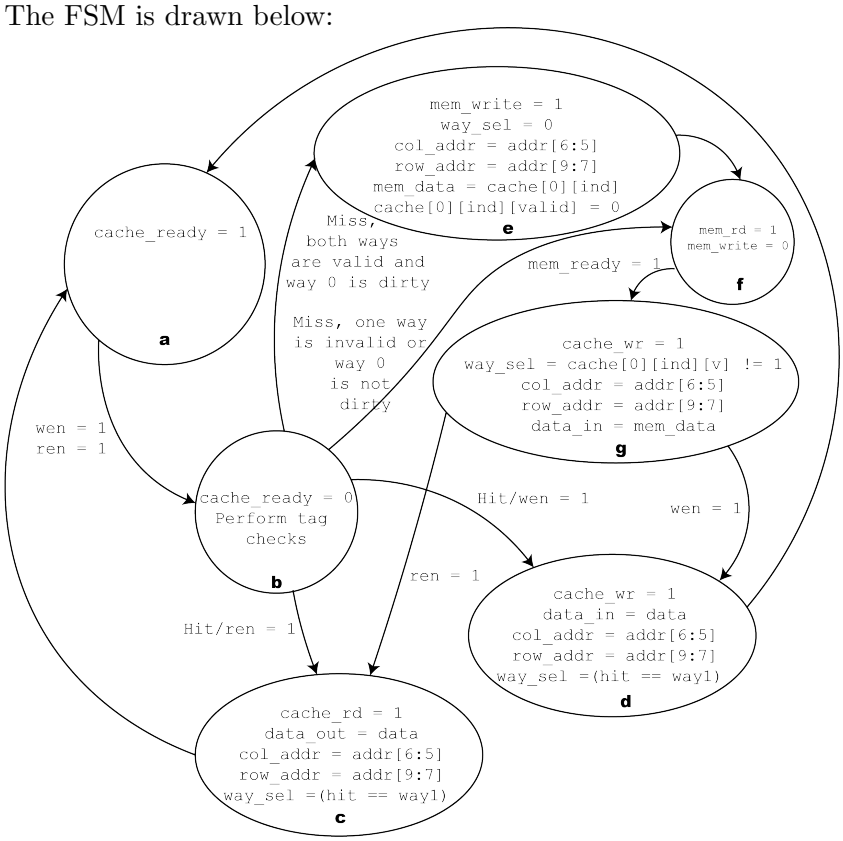
3. Draw a full block diagram with the complete controller logic. You may use logic gates, multiplexers, arithmetic blocks and flip-flops.

Solution:

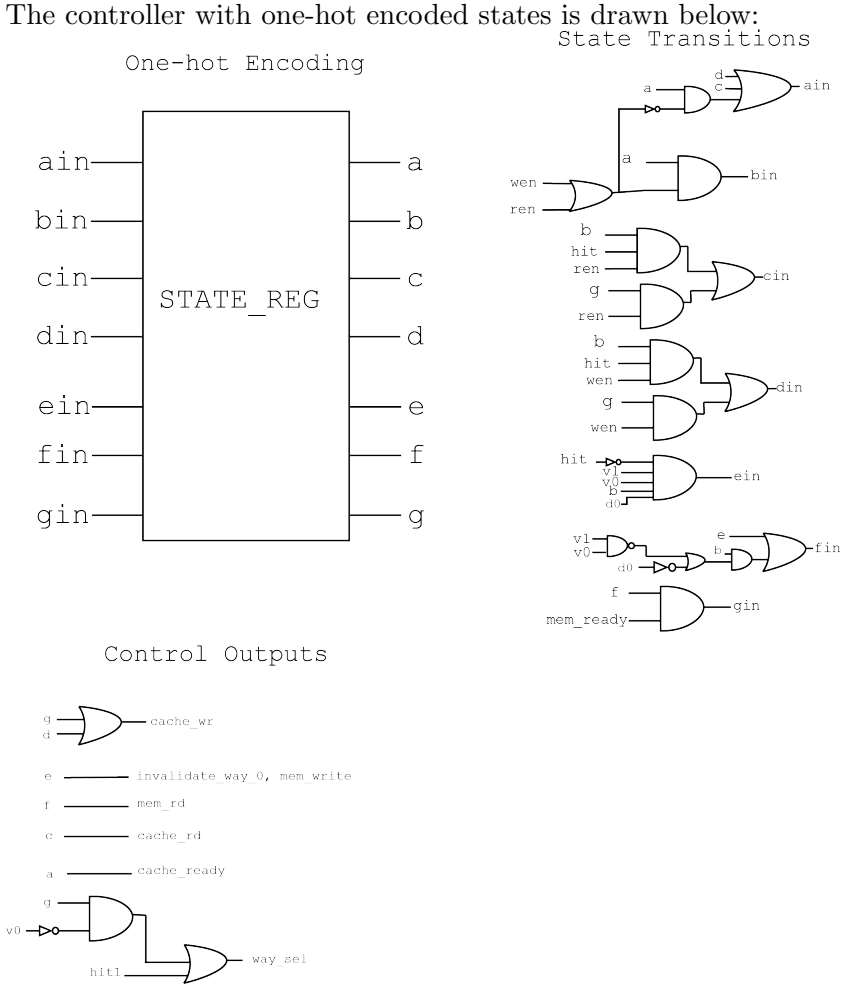
1. If we assume byte-addressed, there are 4 bits needed for the block offset and the 3 bits needed to index, there are 25 bits in the tag. Then, there are an extra 2 bits for the valid and dirty bits. Therefore the tag array has 16 x 27 = 432 bits and the data array has 16 x 128 = 2048 bits. The total number of bits is 2480 bits.



2. The FSM is drawn below:

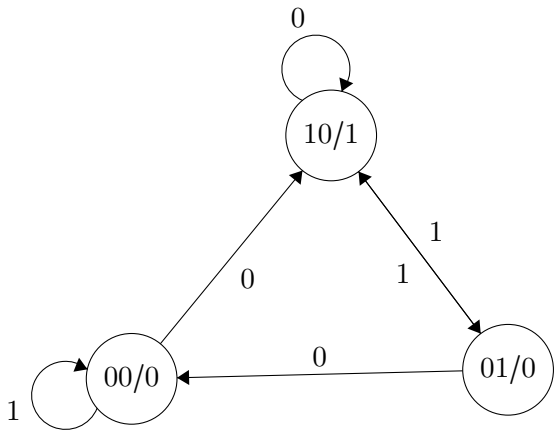


3. The controller with one-hot encoded states is drawn below:



Problem 4

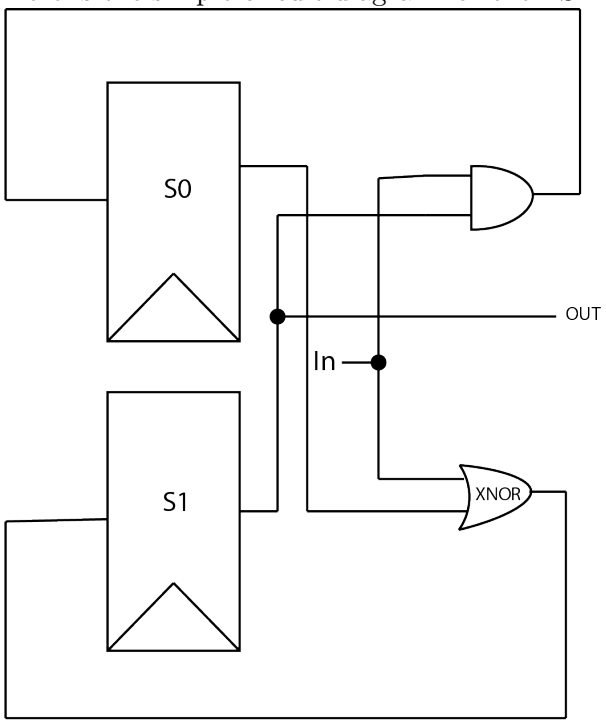
Given the following FSM:



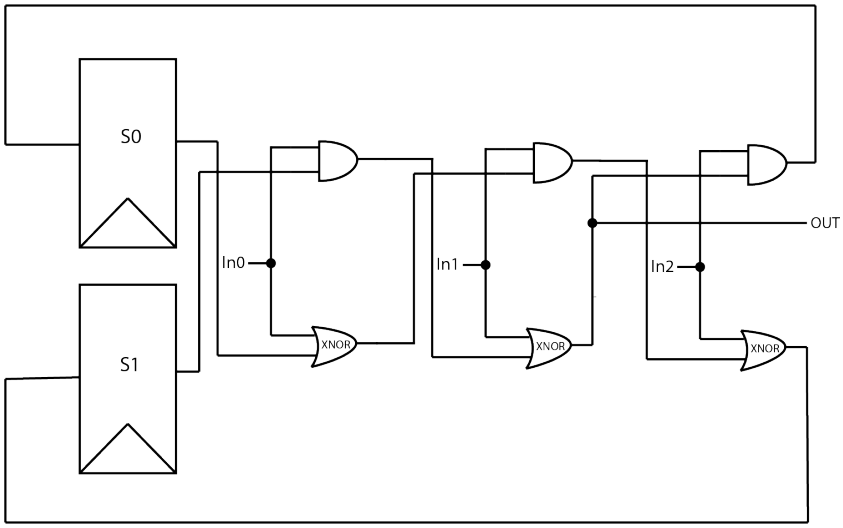
1. Draw the circuit diagram for the FSM using binary encoded states.
2. Draw the circuit diagram for the FSM if we receive 3 bits per cycle (you only need to output the final output). Make sure use loop unrolling.

Solution:

1. Here is the simple circuit diagram for the FSM:



2. Here is the unrolled, unoptimized diagram for the 3-input FSM:



In order to simplify, we can actually look at the 5 variable truth table using S0, S1, In0, In1, and In2 in order to perform a logical reduction. You can do this with a 5-variable KMap (but you didn't have to for this problem!).

Problem 5

```
char* strstr(const char *main_str, const char *substr);
```

`strstr` is a C function that takes 2 strings, `main_str` and `substr` and outputs a 1 if `substr` is in `main_str`.

For this problem, assume that you are designing an accelerator connected to a 1-byte wide memory block. Assume the accelerator is given the address of the two strings. The strings are formatted as traditional ASCII characters followed by a null character. Perform the following:

1. Write a RT Notation for a simple version of this accelerator.
2. Draw the datapath and the state transfer diagram of the controller.
3. Write a RT Notation for a performance-optimized design.
4. Describe the revised datapath for performance optimization.

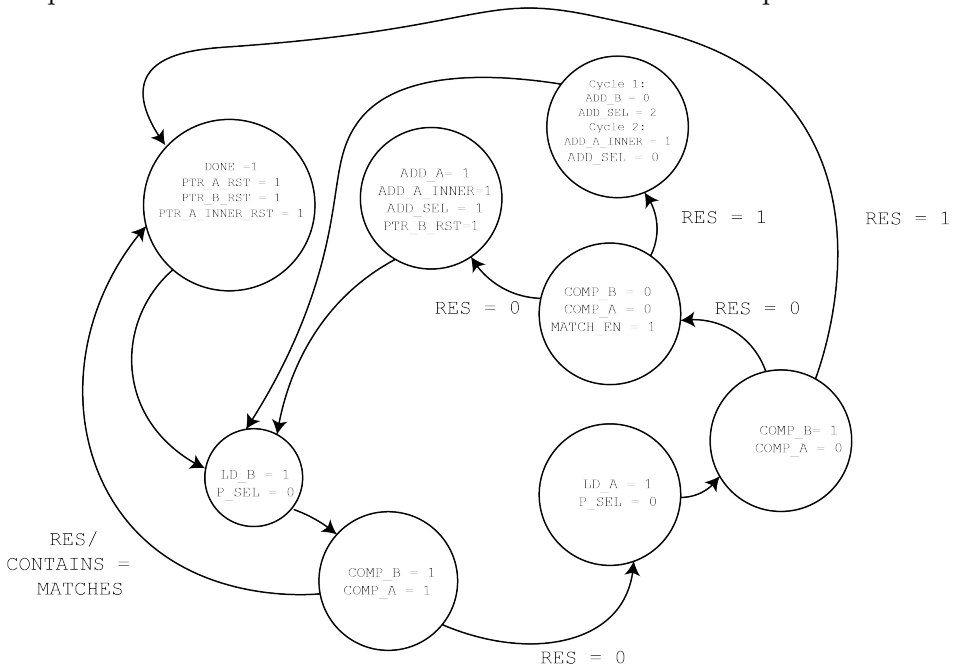
Solution:

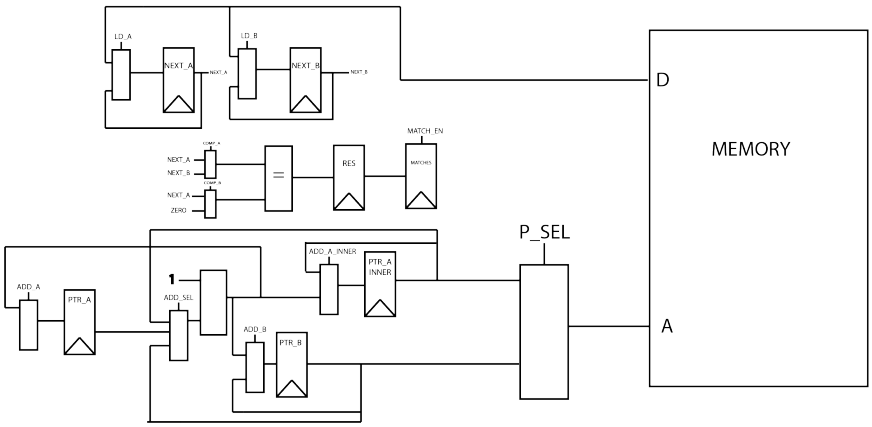
1.
```
If (START == 1) {
  DONE, CONTAINS, MATCHES <- 0;
  PTR_A <- main_str;
  PTR_B <- substr;
  PTR_A_INNER <- main_str;
  repeat {
      DONE_INNER <- 0;
      repeat {
          NEXT_B <- MEM[PTR_B];
          NEXT_A <- MEM[PTR_A_INNER];
          if (NEXT_B == NULL) {
              CONTAINS <- MATCHES;
              DONE <- MATCHES;
              DONE_INNER <- 1;
          }

          if (NEXT_A == NULL) {
              DONE <- 1;
              CONTAINS <- 0;
          } else {
              if (NEXT_A == NEXT_B) {
                  MATCHES <- 1;
                  PTR_B <- PTR_B + 1;
                  PTR_A_INNER <- PTR_A_INNER + 1;
              } else {
                  MATCHES <- 0;
                  PTR_B <- substr;
                  DONE_INNER <- 1;
              }
          }
      } UNTIL (DONE_INNER == 1);
```

```
        PTR_A_INNER <- PTR_A + 1;
        PTR_A <- PTR_A + 1;
        } UNTIL (DONE == 1);
    }
```

2. We put each if block into its own state. The FSM and Datapath will be:





3. We can calculate NEXT_B in the previous states. This allows us to not spend an extra cycle reading NEXT_B. We can also perform both comparisons at once, as well as multiple adds at once at the expense of more HW.

```
If (START == 1) {
// Load NEXT_B here, need to route substr to Address
DONE, CONTAINS, MATCHES <- 0;
PTR_A <- main_str;
PTR_B <- substr;
PTR_A_INNER <- main_str;
```

```
NEXT_B <- MEM[substr];
repeat {
    DONE_INNER <- 0;
    repeat {
        // Still need a cycle to calculate NEXT_A,
        // can calculate some DONEs. Need 2 comparators here
        NEXT_A <- MEM[PTR_A_INNER];
        DONE <- (MATCHES && NEXT_B == NULL)
        CONTAINS <- MATCHES
        DONE_INNER <- NEXT_B == NULL
        // Do both comparisons in one cycle.
        // Can also increment PTR_B pointers
        DONE <- NEXT_A == NULL
        MATCHES <- NEXT_A == NEXT_B
        DONE_INNER <- !MATCHES
        if (MATCHES) {
            PTR_B <- PTR_B + 1;
        } else {
            PTR_B <- substr
        }
        // Read NEXT_B and increment PTR_A_INNER if needed
        NEXT_B <- MEM[PTR_B];
        PTR_A_INNER <- PTR_A_INNER + 1;
    } UNTIL (DONE_INNER == 1);
    PTR_A_INNER <- PTR_A + 1;
    PTR_A <- PTR_A + 1;
    } UNTIL (DONE == 1);
}
```

4. The revised datapath has the controls are slightly different. There are far fewer states now, but you need to be able to do 2 comparisons at once. You additionally need 1 AND gate and extra connections to route correctly. This datapath also requires additonal MUXes to select inputs for comparators, and adders.