EECS 151/251A Homework 9

Due Tuesday, Apr 20nd, 2021

For this Homework

Please include a short (1-2 sentence) explanation with your answer, unless otherwise noted.

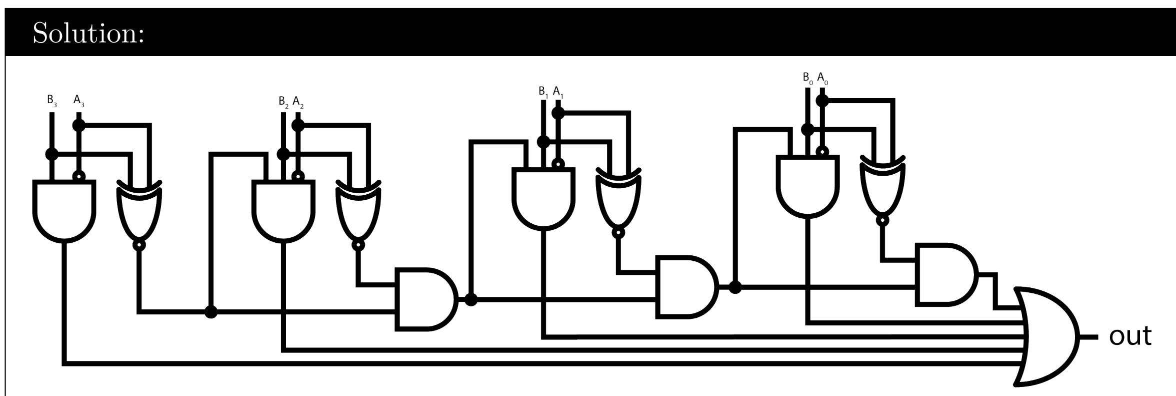
Problem 1: Unsigned Equality Checker

Consider a circuit that compares two N-bit unsigned integers, A and B, and generates a single output bit f, which is equal to 1 if and only if $A \leq B$.

There are two common ways to implement this function. One is to use subtractors to determine the smaller number, but a circuit optimized for the comparison will be simpler and potentially faster. Such a circuit could also be implemented by computing $(A > B)'$, but there is a way to implement this function directly.

(a) Derive the simplest circuit that achieves the direct implementation of the $A \leq B$ function. This circuit will have $O(N)$ cost and delay. Draw such a circuit for $N = 4$. Show your work.

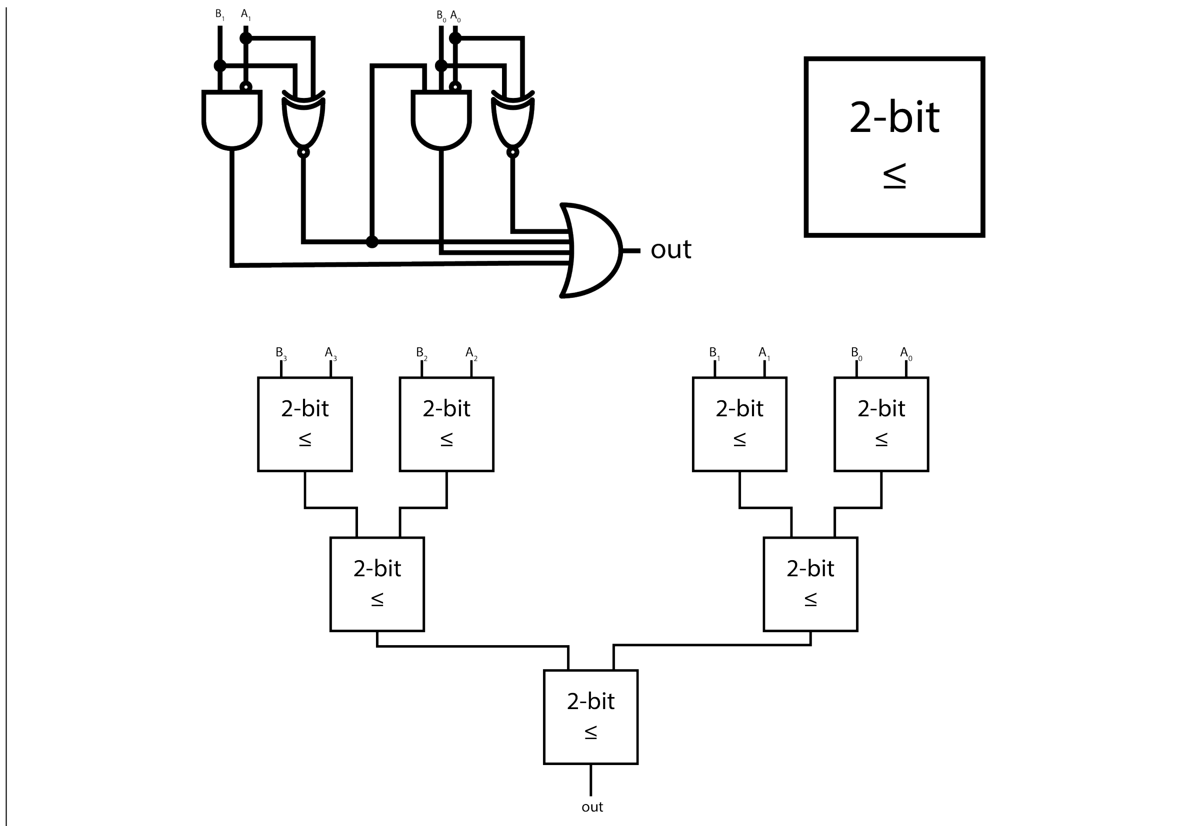
Solution:



(b) Derive a strategy for improving the performance of the circuit from part (a) to have $O(\log(N)$ delay and $O(N)$ cost. Draw a circuit illustrating the structure of your design for $N = 8$.

Solution:

A tree of 2-bit elements will be faster.



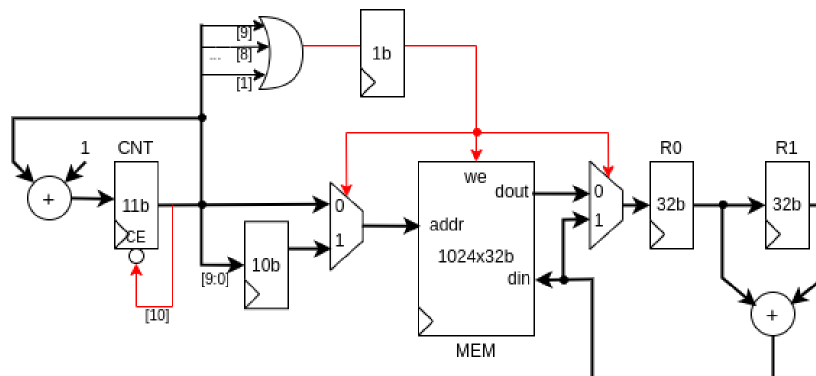
Problem 2: Fibonacci Computer

Given a 1024x32b single-port synchronous read and write memory block, we'd like to design a circuit to compute 1024 Fibonacci elements and store to the memory block using the *minimal* number of cycles. The algorithm description of the problem is as follows.

```
for (i = 2; i < 1024; i = i + 1)
  fib[i] = fib[i - 1] + fib[i - 2]
```

The first two locations of the memory block are initialized to 0 and 1, respectively. Draw the block diagram of your hardware circuit including both datapath (computation, read/write to the memory block) and control (setting up clock enables, memory write enable, and MUX selectors). You may use any logic gates, MUXes, arithmetic units, comparators, registers as you like.

Solution:



At a first glance, it would seem that we could not compute a new Fibonacci element at every cycle due to the resource constraint (single memory port) and the loop-carried dependency (data hazard from an iteration to the next).

We use a length-two shift register to alleviate the resource constraint, since a new element only depends on the last two. We spend a few cycles initially to read the first two elements from the memory to the registers R0 and R1. We also add a data forwarding path to resolve data hazard (memory read-after-write). After that ($cnt_value \geq 2$), the circuit computes new element and write to the Memory block every clock cycle (and we don't need to read from the Memory block anymore). The control logic is simplified due to pipelining. Some control points are pipelined to ensure that the write data aligns with the write address in a cycle. This extra complexity is due to the synchronous read nature of the given Memory block.

This would take 1022 cycles plus some additional cycles to set up R0 and R1 at the beginning.

(Another acceptable solution is to initialize R0 and R1 with the content from the memory, so that one never needs to read from the Memory block initially.)

RT Language description

```
CNT <- 0;
// We use the temporary variable ("tmp") to denote that
// reading from the Memory block takes one cycle
CNT <- CNT + 1, CNT1 <- CNT, tmp <- MEM[CNT], R0 <- tmp, R1 <- R0;
CNT <- CNT + 1, CNT1 <- CNT, tmp <- MEM[CNT], R0 <- tmp, R1 <- R0;
CNT <- CNT + 1, CNT1 <- CNT, tmp <- MEM[CNT], R0 <- tmp, R1 <- R0;
repeat {
  if (CNT < 1024) {
    CNT  <- CNT + 1,
    MEM[CNT1] <- R0 + R1,
  }
  CNT1 <- CNT,
  R0   <- R0 + R1,
  R1   <- R0;
```

```
} until (1);
```

Problem 3: Modulo Scheduling

The lead designer of the chip you are working on has asked you to implement the following operation,

$$Y_i = \text{shift_l}_2(\text{shift_l}_2(A_i) + \text{shift_l}_2(B_i) + C_i)$$

You are given the following blocks to work with:

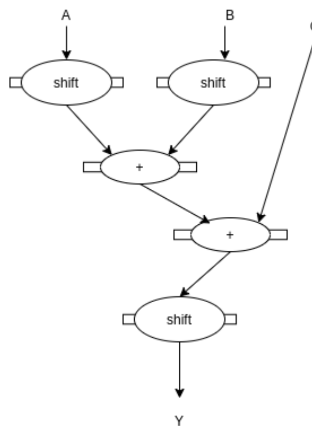
- A memory block with 1 synchronous read port and 1 synchronous write port
- A multiplier block pipeline to two stages
- Single-cycle shifter that shifts 2 bits to the left
- Single-cycle adder

A, B, and C are stored in memory, and Y should be written back to memory once the computation is done.

(a) Using modulo scheduling, show how you would optimally schedule the computation with your datapath.

(b) Draw an extended version of the hardware utilization chart covering at least 3 iterations of the calculation.

Solution:



Naive ASAP (as-soon-as possible) scheduling w/o loop pipelining

Cycle	0	1	2	3	4	5	6	7	8	9	10
shifter		A0	B0				Y0				
adder				A0+B0	+C0						
mem_rd	A0	B0	C0						A1	B1	C1
mem_wr								Y0			

...

First iteration | Second iteration

Modulo scheduling

Cycle	m	m+1	m+2
shifter	Y(i-2)	Ai	Bi
adder	A(i-1)+B(i-1)	+C(i-1)	
mem_rd	Ai	Bi	Ci
mem_wr		Y(i-2)	

Hardware utilization chart (with at least 3 iterations)

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13
shifter		A0	B0		A1	B1	Y0	A2	B2	Y1	A3	B3	Y2	A4
adder				A0+B0	+C0		A1+B1	+C1		A2+B2	+C2		A3+B3	+C3
mem_rd	A0	B0	C0	A1	B1	C1	A2	B2	C2	A3	B3	C3	A4	B4
mem_wr								Y0			Y1			Y2

...

(This solution is based on the assumption that we only have one single-stage shifter and one single-stage adder.)

Follow the modulo scheduling procedure, we first calculate the minimal length of characteristic section (or initiation interval – II).

Per iteration, we need to do three shift operations, three load operations, and one store operations. As we only have one shifter, and the memory block only has one read port, the II should not be less than 3. We start with II=3. The three memory loads are scheduled in consecutive cycles. Other operations are scheduled as soon as possible (after the result of the previous operations are ready). The remaining operations that do not fit in one characteristic section will be wrapped around, and we decrement their iteration subscripts. If there is a conflict (between a scheduling operation from future iteration with the scheduled operation of the current iteration), we would have to increase the II by 1, and start over. In this case, there is no conflict, so II=3 and we have the modulo scheduling solution as shown in the figure above. In other words, we can only start a new iteration every 3 cycles.

As seen from the hardware utilization chart, starting from the fourth cycle we begin to see some overlapping execution between the operations from the first iteration and the second iteration.

Problem 4: `strncmp` Accelerator

```
int strncmp( const char* str1, const char* str2, size_t num );
```

`strncmp` is a C function that compares up to `num` characters of the string `str1` to those in the string `str2`. The function begins comparing the first character of each string. If they are equal, the function continues with each following pair until the characters differ, until it reaches a terminating null character, or until `num` characters have been matched, whichever occurs first.

The function returns the following values:

< 0	The first character that does not match has a lower value in `str1` than in `str2`
0	The contents of both strings are equal
> 0	The first character that does not match has a greater value in `str1` than in `str2`

For this problem assume that the accelerator connects to a 1 byte-wide memory block with a single asynchronous memory read/synchronous write port, and the memory is initialized with two strings. The strings are stored in standard C format with one ASCII character per byte, terminated by a null character. The address of each of the two strings are inputs to the accelerator, and the accelerator will output the return value according to the return behavior detailed previously. The accelerator will assert both the return value and a corresponding "valid" bit once the accelerator finishes computation.

Design the accelerator with the best performance (total time to compare). Start with a simple (less cost) design, and then optimize for performance.

(a) Write a RT Language description of a simple version of the hardware.

(b) Draw the datapath and the state transfer diagram of the controller.

(c) Write a RT Language description for a performance-optimized design.

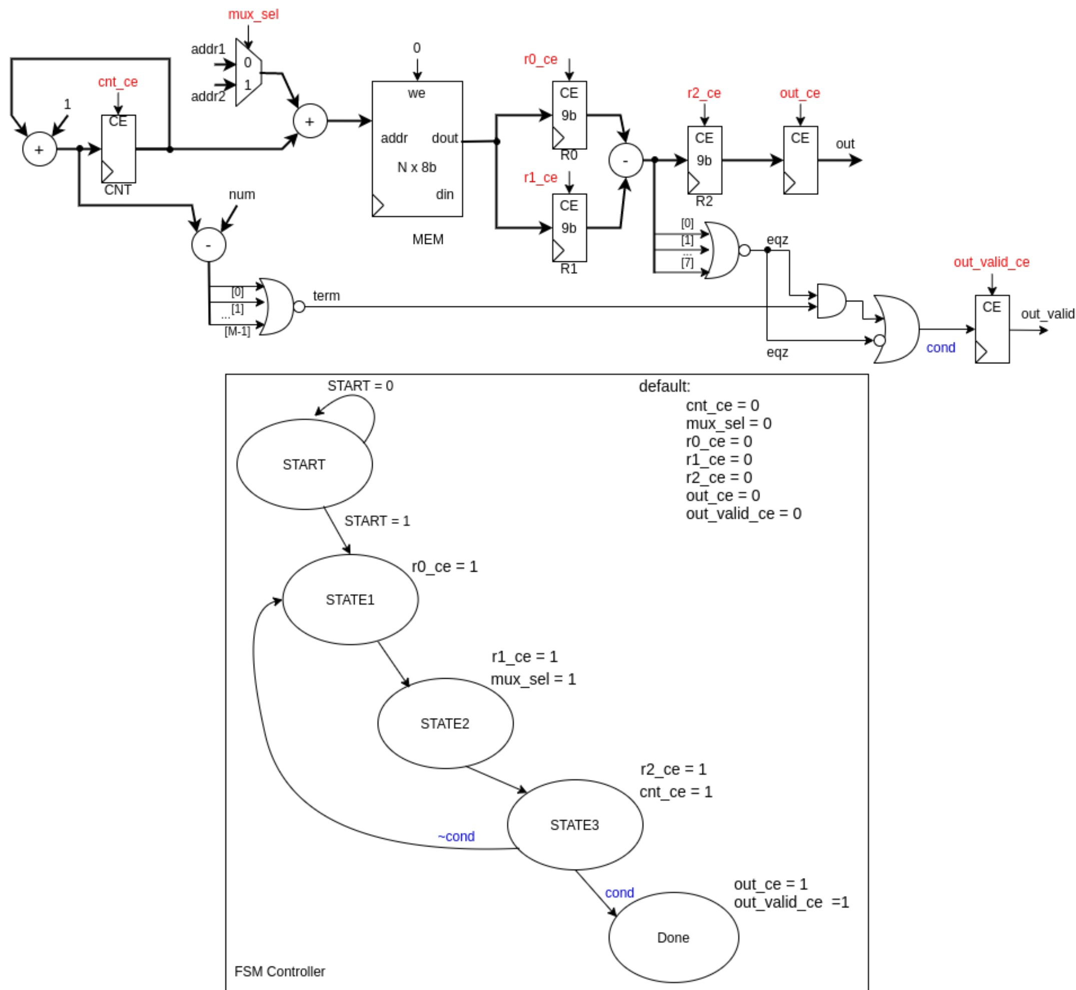
(d) Describe the revised datapath for performance optimization.

Solution:

(a) Answers may vary! Below is the RT Language description of a naive un-pipelined version that one might come up (not necessarily less cost). When writing RT Lang code, we only need to write assignments to synchronous elements, the control signals are implicit. Also note that regarding the semantics of the repeat construct, the repeat condition check happens in the same cycle with the final state from the statements in the body of the iteration (sort of like the semantics in software programming language). Therefore, we use the new values (next) of `R2` and `cnt` for the terminating condition, not the current values.

```
if (START == 1) cnt <- 0, out_valid <- 0;
repeat {
  R0 <- Mem[addr1 + cnt];
  R1 <- Mem[addr2 + cnt];
  R2 <- R0 - R1, cnt <- cnt + 1;
until ((R2 != 0) || (cnt == num));
out <- R2, out_valid <- 1;
```

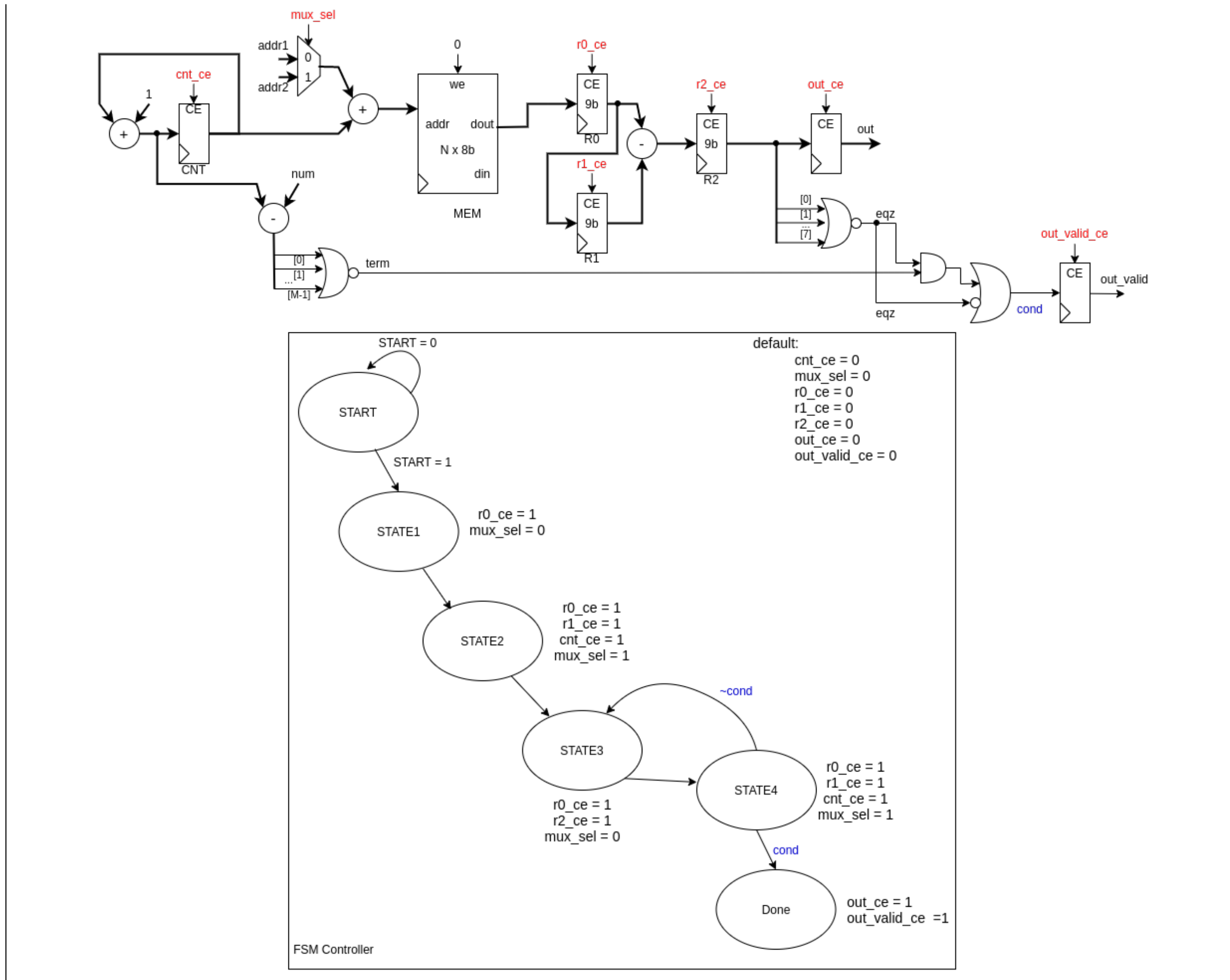
(b) The datapath and state transfer diagram of the controller are shown as follows. The control points (MUX selectors, clock enables) are highlighted in red.



(c) Answers may vary. One could optimize the performance by overlapping reading a new character with the comparison of the current pair of characters. The two registers `R0` and `R1` are cascaded to reduce the fan-out from the Memory data output. In addition, we could also pipeline the checking condition to minimize the critical path.

```
if (START == 1) cnt <- 0, out_valid <- 0;
R0 <- Mem[addr1 + cnt];
R1 <- R0, R0 <- Mem[addr2 + cnt], cnt <- cnt + 1;
repeat {
  R2 <- R1 - R0, R0 <- Mem[addr1 + cnt];
  R1 <- R0, R0 <- Mem[addr2 + cnt], cnt <- cnt + 1;
until ((R2 != 0) || (cnt == num));
out <- R1, out_valid <- 1;
```

(d) The revised datapath along with the FSM controller are shown as follows.


mux_sel
addr1
addr2
0
1
cnt_ce
1
CE
CNT
+
+
0
we
addr
dout
N x 8b
din
MEM
r0_ce
CE
9b
R0
r1_ce
CE
9b
R1
-
r2_ce
CE
9b
R2
out_ce
CE
out
num
-
eqz
term
out_valid_ce
CE
out_valid
cond
eqz
START = 0
START
START = 1
STATE1
r0_ce = 1
mux_sel = 0
STATE2
r0_ce = 1
r1_ce = 1
cnt_ce = 1
mux_sel = 1
~cond
STATE3
r0_ce = 1
r2_ce = 1
mux_sel = 0
STATE4
r0_ce = 1
r1_ce = 1
cnt_ce = 1
mux_sel = 1
cond
Done
out_ce = 1
out_valid_ce =1
default:
cnt_ce = 0
mux_sel = 0
r0_ce = 0
r1_ce = 0
r2_ce = 0
out_ce = 0
out_valid_ce = 0
FSM Controller