Multiple Choice:

[Chapter 2]

1. Given the inputs (*zx, nx, xy, ny, f, no*) = (1, 1, 1, 1, 1, 0), what is the output from the Hack ALU?
    a. -1
    b. -2
    c. –x + -y
    d. 0

Solution: B;
zx=1 sets x to 0, and *nx*=1 does a bitwise inversion on x, resulting in the two's-complement value of -1. The same is done for y, resulting in -1 as well.
f=1 specifies to perform addition, so x + y = -1 + -1 = -2.
no=0 means do not bitwise invert the output, so the final result is -2.

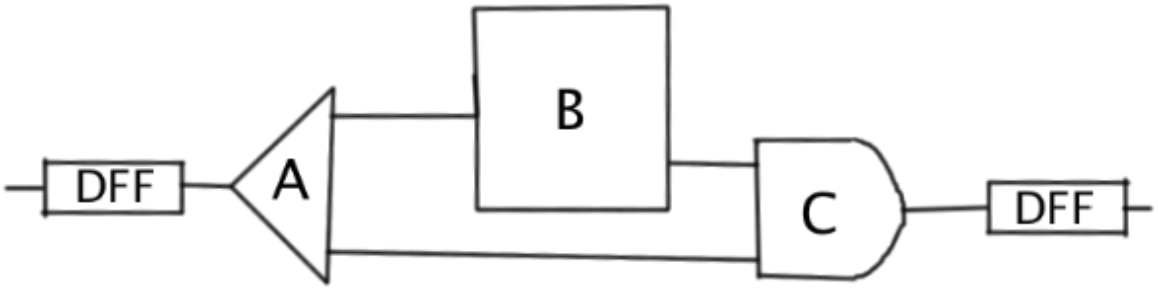
[Chapter 3]

2. Consider a 1-bit register (Bit) where if load(t-1) then out(t)=in(t-1), else out(t)=out(t-1). If we know that the register will be storing a new value more often than not (that is, out(t)=in(t-1) most of the time), how can the load bit of the register be re-implemented for better power consumption?
    a. Set the default load bit to 1 all the time.
    b. Set the default load bit to 0 all the time.
    c. Set load bit to 1 for out(t)=in(t-1), else set load bit to 0 for out(t)=out(t-1).
    d. Set load bit to 1 for out(t)=out(t-1), else set load bit to 0 for out(t)=in(t-1).

Solution: D;
Since we know that the register will be storing a new value most of the time, we can reduce power consumption by re-designing the chip such that the new value can be stored if the load bit is 0, else set the load bit to 1 to keep storing the internal value. To set the load bit to 1 corresponds to turning the transistor ON, which requires that there be enough voltage to surpass the voltage threshold. By having the load bit be 0 to store a new value (which is the more common operation), it is not necessary to generate enough voltage to surpass the voltage threshold since the transistor can just be OFF.

[Chapter 3]

3. What is the minimum number of clock cycles needed for the sequential chip below to perform correctly?
   For the generic sequential chip below, A takes 3 clock cycles to execute, B takes 5 clock cycles to execute, and C takes 1 clock cycle to execute.



   a. 4
   b. 8
   c. 9
   d. 11

Solution: C;
The longest required clock cycle to complete an instruction is 9 clock cycles (A→B→C).

[Chapter 4]

4. Most computer architectures have the concept of a no-operation instruction, NOP, which has no effect on the state of the CPU except increment the program counter to address the next instruction. Which of the following Hack C-instructions (i.e. a specific pattern of 16 bits) is equivalent to a NOP? (Here X stands for any single bit value)

   a. 111X XXXX XX00 0000
   b. 0XXX X000 1010 X100
   c. 111X XXXX XX11 1000
   d. 111X XXXX XX11 1111

Solution: A;
The instruction must not store the ALU's result anywhere nor does it cause the CPU to jump to a different instruction than the next one. Looking at the C-instruction's format, we need to set $d_1\ d_2\ d_3$ to *0 0 0*, and $j_1\ j_2\ j_3$ also to *0 0 0*. It does not matter what the ALU is instructed to do since its result will never be used. There is no destination for the ALU result, so the CPU's state is not modified. By specifying the $j_1\ j_2\ j_3$ bits equal 0 0 0, it ensures that no jump is performed, so the PC will increment and load the next instruction.

Exercises:

[Chapter 1]

1. Given 16-bit inputs a[16] and b[16], the Equivalence function returns 1 when a[16] and b[16] are identical, else returns 0. Using only the logic gates we've built so far and a 4WayAnd gate, which you can assume has already been implemented, design a HDL implementation of the 16-bitwise Equivalence chip (called EQUAL16) with the following interface:
   IN a[16], b[16];
   OUT out;
   - The 4WayAnd has the following interface:
     IN a, b, c, d;
     OUT out;

Solution: See attached *EQUAL16.hdl* file

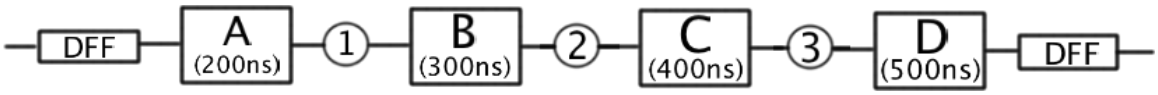
[Chapter 1]

2. The XNOR gate is a logic gate whose function is the inverse of the XOR gate. Using only the nand gates, design a HDL implementation of the XNOR chip (called XNOR) with the following interface:
   IN a, b;
   OUT out;

Solution: See attached *XNOR.hdl* file

[Chapter 3]

3. Consider the diagram below. Positions 1, 2, 3 are possible places to put a DFF. A requires 200 nanoseconds to execute its function, B requires 300 nanoseconds, C requires 400 nanoseconds, and D requires 500 nanoseconds. Where and how many DFFs should be placed in the diagram for maximum speed and throughput? (Note: Pipelining is taken into account). Explain why.



Solution: Two DFFs should be placed in Position 2 and in Position 3 to maximize speed and throughput.

To prove this, we consider all the possible cases and use the process of elimination. Since there are 3 possible positions to put a DFF, it follows that there are 3! = 6 possible cases.

Case 1: Three DFFs in Position 1, Position 2, and Position 3
If three DFFs are used, then the minimum clock cycle timing must be 500ns since the bottleneck is D. We now compute the speed it takes to complete three instructions.

Instruction 1	500ns (200ns)	500ns (300ns)	500ns (400ns)	500ns (500ns)		
Instruction 2		500ns (200ns)	500ns (300ns)	500ns (400ns)	500ns (500ns)	
Instruction 3			500ns (200ns)	500ns (300ns)	500ns (400ns)	500ns (500ns)

From the chart above, we see that to compute three instructions takes 3000ns, or 1000ns/instruction. However, throughput is bad since A and B requires only 200ns and 300ns respectively, yet the clock is set to 500ns.

Case 2: Two DFFs in Position 1 and Position 2
The minimum clock cycle timing is 900ns since the bottleneck is C+D.

Instruction 1	900ns (200ns)	900ns (300ns)	900ns (900ns)		
Instruction 2		900ns (200ns)	900ns (300ns)	900ns (900ns)	
Instruction 3			900ns (200ns)	900ns (300ns)	900ns (900ns)

Computing three instructions requires 4500ns, or 1500ns/instruction. Also, we can see that throughput is much worse than Case 1.

Case 3: Two DFFs in Position 2 and Position 3

The minimum clock cycle timing is 500ns since the bottleneck is (A+B) and D.

Instruction 1	500ns (500ns)	500ns (400ns)	500ns (500ns)		
Instruction 2		500ns (500ns)	500ns (400ns)	500ns (500ns)	
Instruction 3			500ns (500ns)	500ns (400ns)	500ns (500ns)

Computing three instructions requires 2500ns, or approximately 833ns/instruction. Also, given the clock cycle timing the throughput is very good.

Case 4, 5, and 6: One DFF in Position 1, Position 2, or Position 3

We consider all three cases at once.

(Case 4): If the DFF was placed in Position 1, the minimum clock cycle timing must be B+C+D=1200ns.

Instruction 1	1200ns (200ns)	1200ns (1200ns)		
Instruction 2		1200ns (200ns)	1200ns (1200ns)	
Instruction 3			1200ns (200ns)	1200ns (1200ns)

Speed: 1600ns/instruction

(Case 5): If the DFF was placed in Position 2, the minimum clock cycle timing must be C+D=900ns.

Instruction 1	900ns (500ns)	900ns (900ns)		
Instruction 2		900ns (500ns)	900ns (900ns)	
Instruction 3			900ns (500ns)	900ns (900ns)

Speed: 1200ns/instruction

(Case 6): If the DFF was placed in Position 3, the minimum clock cycle timing must be A+B+C=900ns.

Instruction 1	900ns (900ns)	900ns (500ns)		
Instruction 2		900ns (900ns)	900ns (500ns)	
Instruction 3			900ns (900ns)	900ns (500ns)

Speed: 1200ns/instruction

We see that Case 4, 5, and 6 all have lower speeds and throughput than Case 3 (with Case 4 having the slowest speed and throughput out of all the cases). Thus, Case 3, putting two DFFs in Position 2 and in Position 3, results in the fastest speed and best throughput.

[Chapter 4]

4. Write the following program using Hack assembly.
   Modular Arithmetic Program (mod.asm): The inputs of this program are the current values stored in R0 and R1 (i.e. the top two RAM locations). The program computes the modulus R0%R1 and stores the result in R2. We assume (in this program) that R0>=0 and R1>=1.

Solution: See attached *mod.asm* file.

Additional Exercises

These are some exercises that I thought up but decided not to use since they are pretty complicated, and I did not have time to implement the solution.

- Since the Adder built in the previous project was implemented using a ripple-carry effect, it can be pretty slow. A Carry Look-ahead Adder improves speed by reducing the amount of time needed to determine the carry bits. Design a HDL implementation of a 4-bit Carry Look-ahead Adder.
  *Hint: Requires four 1-bit Full Adders and one 4-bit Carry Look-ahead Unit.
  (Even for a 4-bit CLA, there is already a lot of wiring involved, so a 16-bit would be probably too difficult for the scope of this class.)
- WriteLetters.asm:
  This is an extension of the fill.asm program. Instead of filling the entire screen black when a key is pressed, draw the key that is pressed on to the screen. For example, if K is pressed, then the program draws the letter K to the screen. When the key is released, the screen whitens and the letter that was previously drawn is erased.