**UNCA CSCI 320**
**Exam 2 Spring 2016**
19 April, 2016

This is a closed book and closed notes exam. It is to be turned in by ~6:00 PM.

Communication with anyone other than the instructor is not allowed during the exam. Furthermore, calculators, cell phones, and any other electronic or communication devices may not be used during this exam. Anyone needing a break during the exam must leave their exam with the instructor. Cell phones or computers may not be used during breaks.

*If you want partial credit for imperfect answers, explain the reason for your answer!*

Name:_____

**Problem 1 (8 points) Two's complement to decimal conversion**
Convert the following four five-bit *two's complement* numbers into signed decimal representation.

| | |
|---|---|
| 10101 | 11111 |
| 01010 | 10000 |

**Problem 2 (6 points) The full adder**
Place either a truth table for a full adder or a gate-level implementation for a full adder in the space below. (If you have no idea, look at the example SystemVerilog programs.)

**Problem 3 (8 points) x86-64**
Two rather interesting x86-64 instructions, which appeared in Homework 11, are given in the table below. Write a couple of sentences of about what each of these instructions do and why they are frequently used in compiler-generated code.

| |
|---|
| `xorl %edx, %edx` |
| `movl (%rdi,%rdx,4), %ecx` |

**Problem 4 (12 points) SystemVerilog**
Suppose SystemVerilog variables `W`, `X`, `Y` and `Z` are declared as follows

```
logic [3:0] W ;
logic [3:0] X ;
logic [3:0] Y ;
logic [3:0] Z ;
```

What are their values after the following assignments are executed in an `initial` block? Use the blank space for writing their values and showing your work.

```
initial begin

  W = 4'd3 | 4'b1001 ;


  X = W ^ 6 ;


  {Y[1:0], Y[3:2]} = W + W ;


  Z = {W, X[2:0]} ;

end
```

**Problem 5 (12 points) cache**
The Intel 486DX was a 32-bit computer introduced in 1989. The 486DX had a 4-way set associative 8k (8192) byte cache and each block (line, in Intel parlance) of the cache was 16 bytes.

If you prefer the textbook's word oriented terminology: The 486DX had a 2k word cache with blocks of 4 words where each word is 4 bytes.

Now answer some subquestions.

5A: How many blocks (or lines) did the 486DX cache have?

5B: How many sets (or rows) did the 486DX cache have? (It is a tad confusing that rows and lines have different meanings.)

5C: The 32-bit address is divided into three fields: tag, index, and offset. How many bits are each allocated in each of these fields?

5D: What are the tag, index, and offset fields of the address `0xABBA2468`?

**Problem 6 (4 points) Effective access time**
Suppose it takes the 486DX 3 clock cycles to read data from the cache and 500 clock cycles to read data from RAM. Further suppose the average hit rate is 90%. What is the effective access for reading memory? Assume that an attempt is made to read the cache before an access is made to RAM.

**Problem 7 (12 points) Adding numbers, signed and unsigned**
Addition overflow occurs when you add two numbers and the result is out of range.

First of all, what is the range of numbers that can be represented by six-bit unsigned numbers?

What is the range of numbers that can be represent by six-bit signed (twos-complement) numbers.

Add the four pairs of six-bit numbers below. If the result is out-of-range for signed addition, write ***signed overflow*** in the box. If the result is out-of-range for unsigned addition, write ***unsigned overflow*** in the box.

| | |
|---|---|
| 010111<br>+ 001001 | 010110<br>+ 000111 |
| 111011<br>+ 111000 | 110001<br>+ 111111 |

**Problem 8 (8 points)**

Your assignment is to make an odd parity checker FSM with a single data input, `a`, and a single data output, `y`. It also has a reset input, `reset`, and a clock input, `clk`. The FSM is, as expected, an asynchronously reset Moore machine.

The output of the FSM should be 1 if and only if the number of 1's received since the FSM was reset is an odd number. So, if the input sequence was 10101, the output should be 1; and, if the input sequence was 10001, the output should be 0.

To save your self some writing (and me some reading), just take the example and scratch out the stuff that you don't want and scratch in some stuff that you do want.

*Minimize the scratching out.*

```
// 4.31: patternMoore
module patternMoore(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate <= S0;
          else   nextstate <= S1;
      S1: if (a) nextstate <= S2;
          else   nextstate <= S1;
      S2: if (a) nextstate <= S0;
          else   nextstate <= S1;
      default:   nextstate <= S0;
    endcase

  // output logic
  assign y = (state == S2);
endmodule
```

**Problem 9 (8 points)**

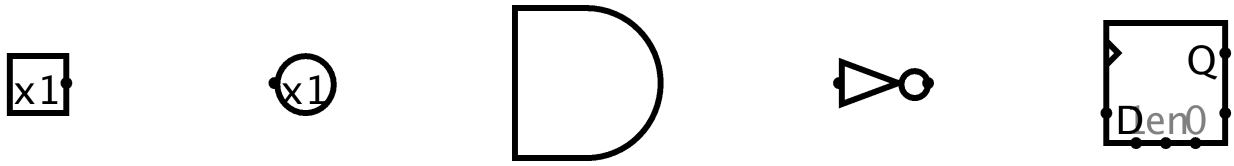
I'm sure that you, as required, followed the suggest style in fixing up the last problem. But admit it, you'd be ridiculed in CSCI 202 for using an FSM to implement a parity checker. All you really need is a single Java boolean that records if the parity so far is odd or even. Or you could use a Java integer and the XOR (^),

This time use a *single internal variable* to keep up with the parity. I've grayed out all the stuff you shouldn't be using. You need to replace all the gray places.

```
// 4.31: patternMoore
module patternMoore(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate <= S0;
          else   nextstate <= S1;
      S1: if (a) nextstate <= S2;
          else   nextstate <= S1;
      S2: if (a) nextstate <= S0;
          else   nextstate <= S1;
      default:   nextstate <= S0;
    endcase

  // output logic
  assign y = (state == S2);
endmodule
```

**Problem 10 (12 points) Gate implementation**
The two SystemVerilog modules in the last two problems should have the same externally observable behavior, though one should be much easier for a CSCI major to read.

In this page, you should implement the sequential odd parity FSM with the elements resembling the usual input and output pads, combinational circuit elements (AND, NOT, *etc*), and D flip-flops of Logisim. Here as of some examples of taken from Logisim.




The circuit should be pretty simple. You are encouraged to include a next state table and an output table with your answer.

**Problem 11 (10 points) Contamination and propagation delays**
The table below gives the contamination and propagation delays of three gates.

| Gate | Contamination | Propagation |
|---|---|---|
| NOT (inverter) | 10 ps | 15 ps |
| Two-input NOR | 20 ps | 25 ps |
| Two-input AND | 25 ps | 30 ps |
| Two-input OR | 30 ps | 40 ps |

What is the contamination and propagation delays for the circuit shown in Problem 3. (Show your work by drawing some numbers on the connections between gates.)