Name:_____

EECS150: Components and Design Techniques for Digital Systems

University of California
Dept. of Electrical Engineering and Computer Sciences

Mid Term 2 – version A Fall 2004

Last name: _____ First name_____
Student ID: _____ Login: _____

Lab meeting time: _____ TA's name: _____
(Sorry to ask this next question, but with 100 students packed closely together there may be a wide range of behavior.)
Student to my left is _____ _____
Student to my right is _____ _____

No notes. No calculators! This booklet contains 9 numbered pages, including room to show your work. Please, no extra stray pieces of paper. The exam contains 5 substantive questions and 100 points, so just over 1 point per minute. Browse through the questions before you start. You have 1.5 hours, so relax, work thoughtfully and give clear answers. Good luck!

I certify that my answers to this exam are my own work. If I am taking this exam early, I certify that I shall not discuss the exam questions, the exam answers, or the content of the exam with anyone until after the scheduled exam time. If I am taking this exam in scheduled time, I certify that I have not discussed the exam with anyone who took it early.

Signature: _____

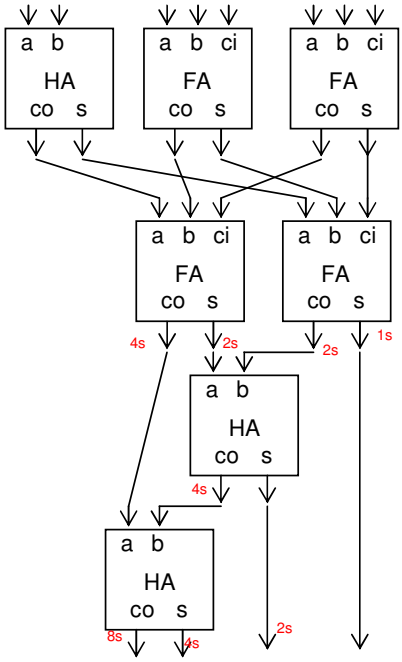
Problem 1 [15]	
Problem 2 [20]	
Problem 3 [15]	
Problem 4 [20]	
Problem 5 [30]	
Total [100]	

Name:_____

Problem 1 (15). Arithmetic Circuit Design

a. You are to build a POPCOUNT unit, shown below, which takes a single 8-bit input and produces a 4-bit output that is the number of bits that are 1 in the input. You are to construct it out of Full Adder (FA) and Half Adder (HA) blocks, not logic gates. Use as few FA's and HA's as possible. Clearly label your design.

b. If you generalize your approach, how does the critical path of your design compare to that of an n-bit ripple carry adder? n-bit CLA?



Many solutions put all inputs into a Carry Propagate Adder.
The important part of the solution was to correctly combine bits with like place values. That is to say bits with a 2^0 significance cannot be grouped with bits with a 2^1 significance…
Other possible solutions involved using a balanced binary tree of 2, 2 and then 3 bit adders. However many people claimed O(log(n)) or O(n) performance, however the real performance of this tree is slightly less than $O(\log^2(n))$
Make sure you understand what a half-adder and full-adder are. Neither is a ripple carry adder.

Name:_____

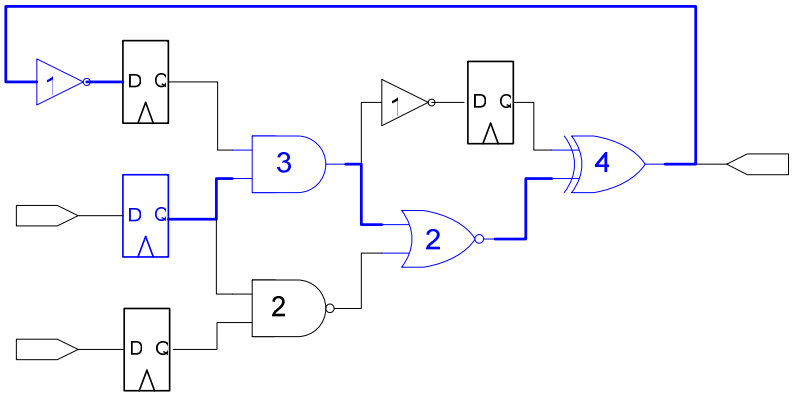
Problem 2 (20). Circuit Delays

Determine the maximum clock rate for the circuit shown below. Assume the following:

(1) The primitive inverter delay is 100 ps. All wire delays are 0.
(2) The primitive delay of each gate is the number specified inside the gate. It is in units of primitive inverter delays.
(3) The actual delay of a gate in the circuit is a linear function of its primitive gate delay and its fanout: *Actual delay = primitive delay + 0.25 * (# of fanouts)*
   a. Any logic gate, flip-flop or output port counts as one fanout of a gate.
   b. For example, the XOR gate in the circuit below has a fanout of 2 (output port and inverter).
   c. The two inverters in the circuit have a fanout of 1 (flip-flop input).
   d. Flip-flop outputs incur fanout-related delays, just like gates.

(4) Flip-flop setup time and clock-to-Q time is 2 inverter delays.
(5) The maximum skew between any two clock inputs is 50 ps.

State clearly any other assumptions you make. **Show your work.**



Longest path: FF-AND-NOR-XOR-INV-FF
Delay = 3 + 0.25(2) + 2 + 0.25(1) + 4 + 0.25(2) + 1 + 0.25(1) = 11.5 inv delays
A path is defined and any series of combinational gates from any flip-flop output to any flip-flop input. This is not necessarily the delay around a loop. A path CANNOT GO THROUGH TWO FLIP-FLOPS.

FF delay = 2 + 0.25(2) + 2 = 4.5 (clock-to-Q + fanout delay + setup)

Clock skew = 0.5
Only applies if the path starts and ends at a different register.

Total delay = 11.5 + 4.5 + 0.5 = 16.5 inv delays or 1650 ps.
Maximum Frequency = 1/1650ps = 606MHz

Problem 3 (15). Memory

Describe each of the steps involved in a DRAM read operation. (We are looking for a single brief sentence or two for each major step.)

- Prior to the read, `OE_l` and `WE_l` are disasserted (`OE_l = 1, WE_l = 1`) and the DRAM chip is essentially inactive.
- Row address is provided. After row address becomes valid, RAS is asserted (RAS = 1)
- Column address is provided. After Column address becomes valid, CAS is asserted (CAS = 1)
- Output enable is asserted (OE_l = 0)
- Value of selected bits causes a small change in the precharged value on the line which is detected and amplified by the output driver or "sense amp"
- After a delay (of length specified by the data sheet), output data is latched
- Internally, the values read from the selected row are restored
- RAS and CAS are de-asserted

Name:_____

Problem 4 (20). Understanding verilog

It is your job to find all 10 errors in the verilog fragment below, and suggest corrections for each error. The exact function of the FSM doesn't matter.

An **error** is counted as anything where **a single continuous block of text is missing or wrong**. This means that an error may span multiple lines, but there will be no correct text between the lines.

```
module ShiftRegister(Clock, Reset, SIn, POut);
      input       Clock, SIn, Reset;
      output      [7:0] POut;

      reg   [7:0] POut;

      always @ (posedge Clock) begin
            if (Reset) POut <=       8'h00;
            else POut <=             {POut[6:0], SIn};
      end
endmodule

module FSM(In, Out, OutValid, Clock, Reset);
      input             In, Clock, Reset;
      output      [7:0] Out;
      output            OutValid;

      reg   [1:0]       CurrentState, NextState;
      reg               ShiftReset, OutValid;

      ShiftRegister     Shifter(    .SIn(In),
                                    .POut(Out),
                                    .Clock(Clock),
                                    .Reset(ShiftReset | Reset));

      parameter   STATE_Idle =          2'h0,
                  STATE_A =             2'h1,
                  STATE_B =             2'h2;

      always @ (posedge Clock) begin
            if (Reset) CurrentState <=          STATE_Idle;
            else CurrentState <=                NextState;
      end

      always @ (CurrentStat e or Out or In) begin
            NextState =                         CurrentState;
            OutValid =                          1'b0;
            ShiftReset =                        1'b0;

            case (CurrentState)
                  STATE_Idle: begin
                        if (In) begin
                              NextState =       STATE_A;
                              ShiftReset =      1'b1;
                        end
                  end
                  STATE_A: begin
```

Comment [GDG1]: Reset was neglected as an input. This was not an intentional mistake, but you will still receive points for it.

Comment [GDG2]: Must be declared as a "`reg`" if it is to be a register later.

Comment [GDG3]: Used to be "`Reset or Sin`" which doesn't work. We need a register.

Comment [GDG4]: `Reset` used to be left out. Every register needs to have a "`Reset`" input.

Comment [GDG5]: Should be non-blocking assignment. Blocking assignment only works for combinational `always` blocks.

Comment [GDG6]: We forgot a semicolon here. This was not a purposeful error on our part, but you still get credit if you caught it.

Comment [GDG7]: `NextState` must also be a "`reg`" even though it is not a register. This is because it is assigned in an `always` block.

Comment [GDG8]: These connections are not in the same order as the port list for the `ShiftRegister` module. That means you need to specify which connection goes to which port. (Or reorder the connections)

Comment [GDG9]: The signals `Out` and `In` were missing from this sensitivity list.

Comment [GDG10]: Some default value should be specified here in case a proper value is not given later. This will prevent latches.

```
                        if (Out[0] & ~Out[7]) begin
                              NextState =       STATE_B;
                              OutValid =        1'b1;
                        end
                  end
                  STATE_B: begin
                        ShiftReset =            1'b1;
                        if (~|Out) NextState =  STATE_Idle;
                  end
                  default: begin
                        NextState =             2'bxx;
                        OutValid =              1'bx;
                        ShiftReset =            1'bx;
                  end
            endcase
      end
endmodule
```

Comment [GDG11]: Used to be "`Valid`" which is not an actual signal

Comment [GDG12]: FSMs must always include a default block to ensure that 1) there are no latches and 2) the logic generated by this `always` block can be optimized properly.

Name:_____

Problem 5 (30). Controller design

In 1976 Seymore Cray introduced the Cray-1 Supercomputer that ran at an amazing 80 MHz and had all the characteristics that we associate with modern RISC processors. It also introduced the concept of vector registers, which are only beginning to appear in modern designs. In this problem you will design a portion of a vector unit.

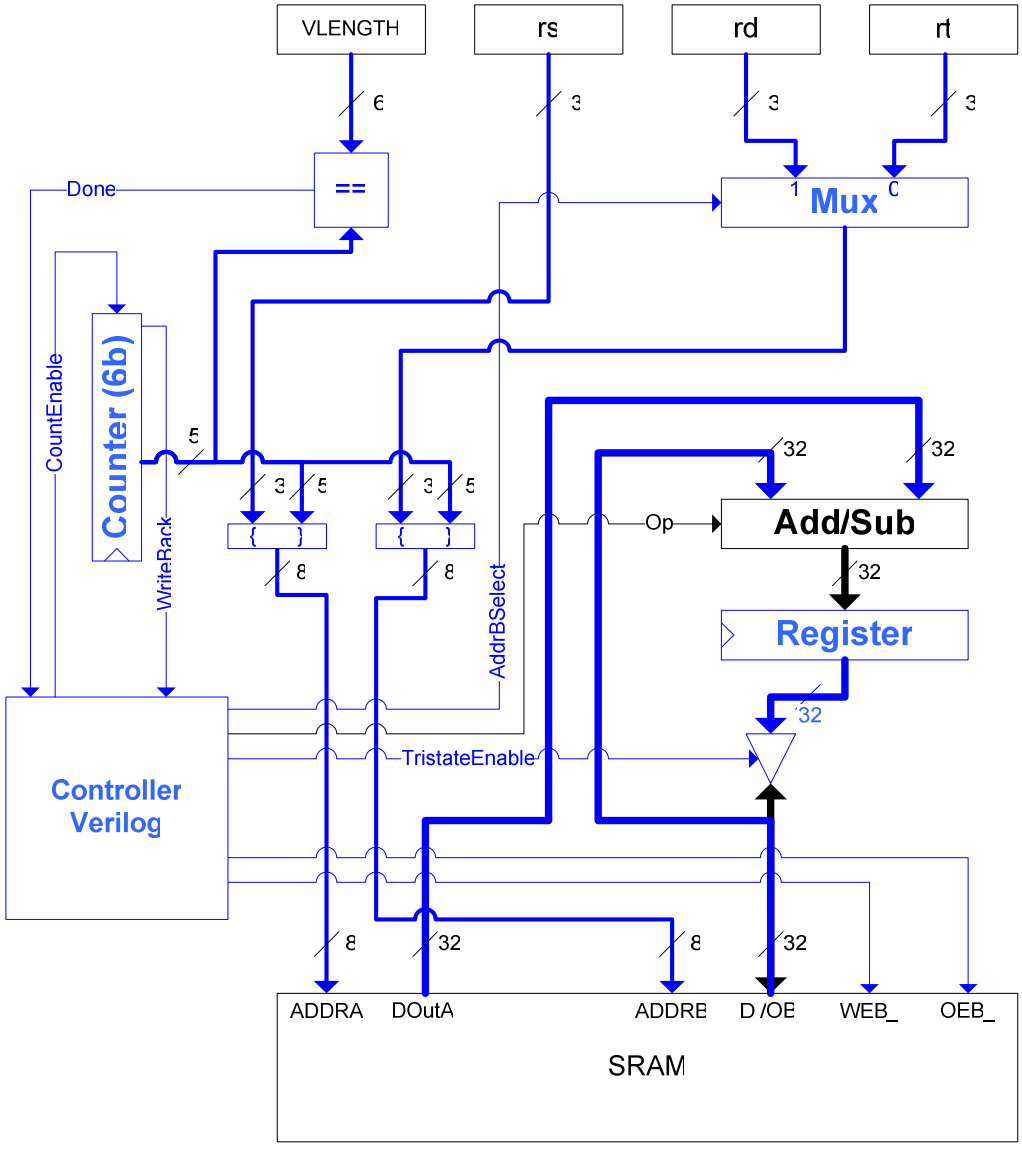
Your machine has 8 vector registers, each containing 32 words of 32 bits in width. It has an adder unit that takes two 32-bit inputs and generates a 32-bit result. An additional `VLENGTH` register specifies the number of elements that are to be added. Thus the operation: `VADD rd, rs, rt` implements

```
FOR i from 0 to VLENGTH-1
      REG[rd[i]] := REG[rs[i]] + REG[rt[i]]
```

a. Starting with the datapath skeleton shown below, you are to implement a controller for the `VADD` operation.

The vector registers are stored in a synchronous SRAM with two ports: port A is a read port and port B is a read/write port. The WriteEnable input (`WEB_`) indicates whether the operation on port B is a write. The vector register numbers for the sources and destination, as well as `VLENGTH`, are provided to you in registers.

Complete the datapath with MUXes, tristates, wires, counters, latches, adders, flipflops or other components as required to implement the `VADD` operation.



b. Starting with the verilog skeleton below, implement your controller for your datapath.

```
module VADD(      Clock, Reset,
                  WEB_, OEB_,
                  TristateEnable, Op, AddrBSelect,
                  WriteBack, CountEnable, Done);

      input             Clock, Reset;

      output            WEB_, OEB_;
      output            TristateEnable, Op, AddrBSelect;
      input             WriteBack;
      output            CountEnable;
      input             Done;

      parameter         OP_Add =          1'b0,
                        OP_Sub =          1'b1;

      assign      WEB_ =                  ~(WriteBack & ~Done);
      assign      OEB_ =                  WriteBack | Done;

      assign      TristateEnable =        WriteBack;
      assign      Op =                    OP_Add;
      assign      AddrBSelect =           WriteBack;

      assign      CountEnable =           ~Done;
endmodule
```

For this problem we accepted answers with either this simple Verilog based controller, and the more complicated datapath shown above or a similar solution but with a simpler datapath and a more complicated controller. The solution here shows the ideal division between datapath and controller, however we did not require you to divide it this way. In a real system the division would depend also on the other commands which had to be implemented on the datapath.

Those of you who took 3 cycles per addition also received full credit

Grading Scale

Datapath

- 3 - 3 Data busses to addr
- 3 - Tri-state for D/IOB
- 3 - Concatenate addresses
- 3 - Mux for the ADDRB line
- 1 - Correct operation (Add/Sub)
- 3 - Comparator for counter

Control

- 3 - Counter
- 3 - Read State/Write State
- 2 - Idle State/Done State