ECE-327 Solution to Midterm

2018t1 (Winter)

All requests for re-marks must be submitted by email to Mark Aagaard before noon on Friday March 2.

		Total Marks	**Approx. Time**	**Page**
Q1	VHDL Semantics	20	10	2
Q2	Code Optimization	20	15	3
Q3	FSM	20	15	7
Q4	DFD	25	18	8
Q5	Memory	15	10	9
Totals		100	68	

Q1 (20 Marks) VHDL Semantics

(*estimated time: 10 minutes*)

In VHDL, when do assignments to signals become visible?

Answer:

At the beginning of the next simulation cycle.

Marking:

5 marks *correct answer*
3 marks *in next simulation cycle*
3 marks *beginning of next delta cycle*
1 mark *in next delta cycle*
1 mark *registered: next clock cycle, comb: immediately*

Over the past few years, you have been very successful at creating new hardware description languages that are variations of VHDL (Synflopsys, Why Wait, *etc.*). You have earned billions of dollars and are now retired, living on your own tropical island. Despite your luxurious life of leisure, you still have the urge to explore new hardware description languages.

Your next language, the Island Description Language for Electronics (Idle) uses the same syntax as VHDL, but changes the semantics so that assignments to signals become visible *at the end of the simulation cycle in which the assignment was executed*.

If you compare Idle and VHDL by simulating the same program in an Idle simulator and a VHDL simulator, will each signal have the same value in both simulators at the beginning of every nanosecond of simulation? **For full marks, you must justify your answer.**

Answer:

All *programs will have the same behaviour in VHDL and Idle.*

This solution is longer than necessary for an exam answer.

The behaviour of a program in VHDL and Idle will be identical, except that the first *simulation step in a simulation cycle in VHDL will become the* last *simulation step in the previous simulation cycle in Idle.*

The visible value of each signal at the end of a simulation round will be the same in VHDL and Idle. Time increments at the beginning of a simulation round. At the beginning of each nanosecond, the visible value of each signal will be the same in VHDL and Idle

Marking:

+5 marks *updating at end of one sim cycle is same as updating at beginning of next sim cycle*
+5 marks *time increments at beginning of simulation round*
+5 marks *all values are stable before a simulation round ends and next one starts*
+5 marks *Simulation cycles are instantaneous, so difference between Idle and VHDL won't be visible at the nanosecond level.*

Q2 (20 Marks) Code Optimization

(*estimated time: 15 minutes*)

In this question, you will analyze the VHDL program `payette`, whose entity is on this page and whose architecture is on the next page.

NOTES:

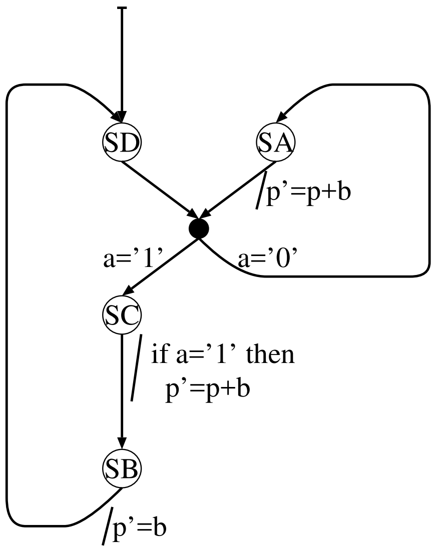
1. Signal and state names are *not* related to the purpose of the signal or state, except `clk`, `reset`, and `state`.

Q2a (5 Marks) Encoding

Based upon the ECE-327 guidelines, which state-encoding should be used with `payette`? **For full marks, you must justify your answer.**

Answer:

One-hot*: The system uses an ASAP parcel schedule and does not have inter-parcel variables, which are the two criteria for valid-bits. A one-hot encoding allows us to test if we are in a state by testing just one bit (a wire).*



Show your encoding for the states:

```
SA  "0001"
SB  "0010"
SC  "0100"
SD  "1000"
```

Marking:

+2 marks	*Choice of state encoding*
+2 marks	*Justification*
+1 marks	*Encoding of values*

Q2b (15 Marks) Optimizations

On the next page, use the recommended state encoding from Q2a to optimize the code.

NOTES:

1. Your optimizations shall *not* affect the clock-cycle by clock-cycle behaviour of `p`, `y`, or `z`.
2. You may change, add, and delete lines of code.

```
entity payette is
  port ( clk, reset : in std_logic;
         a : in std_logic;
         b : in signed( 7 downto 0 );
         y : out std_logic;
         z : out signed( 7 downto 0 )
       );
end entity;
```

Original:

```
architecture main of payette is
  signal p : signed( 7 downto 0 );
  type state_ty is ( SA, SB, SC, SD );
  signal state : state_ty;
begin

  process begin
    wait until rising_edge(clk);
    if reset = '1' then
      state <= SD;
    else
      if state = SD or state = SA then
        if a = '0' then
          state <= SA;
        else
          state <= SC;
        end if;
      elsif state = SC then
        state <= SB;
      else
        state <= SD;
      end if;
    end if;
  end process;

  process begin
    wait until rising_edge(clk);
    if state = SB then
      p <= b;
    elsif state = SA
          or (state = SC and a='1')
    then
      p <= p + b;
    end if;
  end process;

  z <= p;
  y <=   '1' when p > 17
    else '0';

end architecture;
```

Answer:

```
architecture main of payette is
  signal p : signed( 7 downto 0 );
  signal state :
    std_logic_vector( 3 downto 0 );
begin

  process begin
    wait until rising_edge(clk);
    if reset = '1' then
      state <= "1000";
    else
      state(0) <= (state(0) or state(3))
                  and not a;
      state(2) <= (state(0) or state(3))
                  and a;
      state(1) <= state(2);
      state(3) <= state(1);
    end if;




  end process;

  process begin
    wait until rising_edge(clk);
    if state(1) = '1' then
      p <= b;
    elsif state(0) = '1'
          or (state(2) = '1' and a='1')
    then
      p <= p + b;
    end if;
  end process;

  z <= p;
  y <=   '1' when p > 17
    else '0';

end architecture;
```

Marking:

- **+2 marks** *State declaration*
- **+2 marks** *Reset state*
  - *Matches encoding*
  - *Correct value of SD from Q2a*
- **+3 marks** *Next state: control structure*
  - *Use Boolean expressions*
  - *If use if-then-else, must:*
    - *Synthesize to same hardware as Boolean expressions*
    - *State bits are set to both `'1'` and `'0'`.*
- **+3 marks** *Next state: test individual bits*
- **+3 marks** *Next state: assign to individual bits*
- **+2 marks** *Code for `p` is correct and optimal*

Q3 (20 Marks) FSM

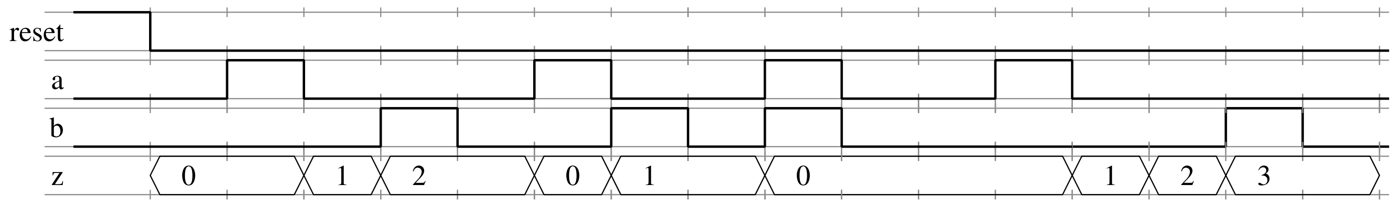
(*estimated time: 15 minutes*)

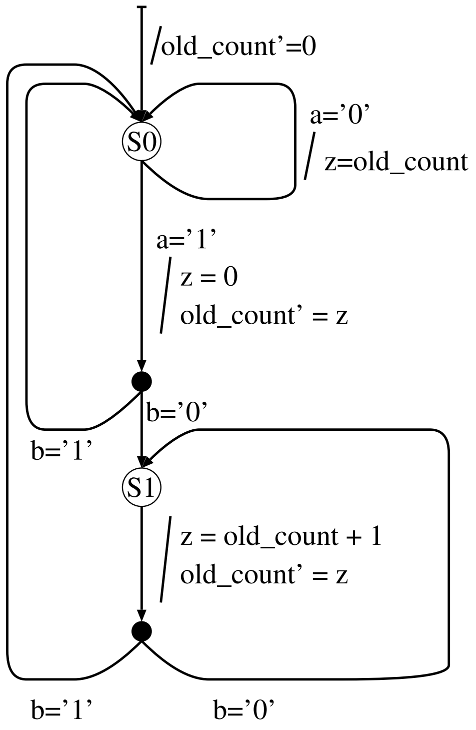
Draw a state machine that counts the number of clock cycles between `a='1'` and `b='1'`.

NOTES:

1. The system has three inputs: `reset`, `a`, and `b`.
2. The system has one output: `z`.
3. From the first clock cycle when `reset` is deasserted (changes from `'1'` to `'0'`) up to but not including the clock cycle when `a='1'`, `z` shall be `0`.
4. In the clock cycle when `a='1'`, `z` shall be `0`.
5. From the clock cycle after `a='1'` up to and including the clock cycle when `b='1'`, `z` shall increment in each clock cycle.
6. From the clock cycle after `b='1'` up to but not including the clock cycle when `a='1'`, `z` shall hold its value constant, showing the number of clock cycles between `a='1'` and `b='1'`.
7. After `a` changes from `'1'` to `'0'`, it is guaranteed to remain `'0'` until the clock cycle after `b='1'`.
8. Marks will be earned for the state machine being syntactically correct, functionally correct, simple and elegant, and drawn neatly.

Example execution:





Marking:

- **2 marks** *state transitions cover all possibilities and are mutually exclusive*
- **2 marks** *correct syntax and use of registered and combinational assignments*
- **2 marks** *correct syntax and use of conditions and assignments*
- **2 marks** *reset is done correctly, `z=0` after reset, and holds value*
- **2 marks** *test `a='1'` and `b='1'` in same clock cycle*
- **2 marks** *`z` holds its value after `b='1'`*
- **2 marks** *`z` updated in same clock cycle as `a` and `b`*
- **2 marks** *`z` gets correct count*
- **2 marks** *simplicity and elegance of design*
- **2 marks** *neatness of drawing*

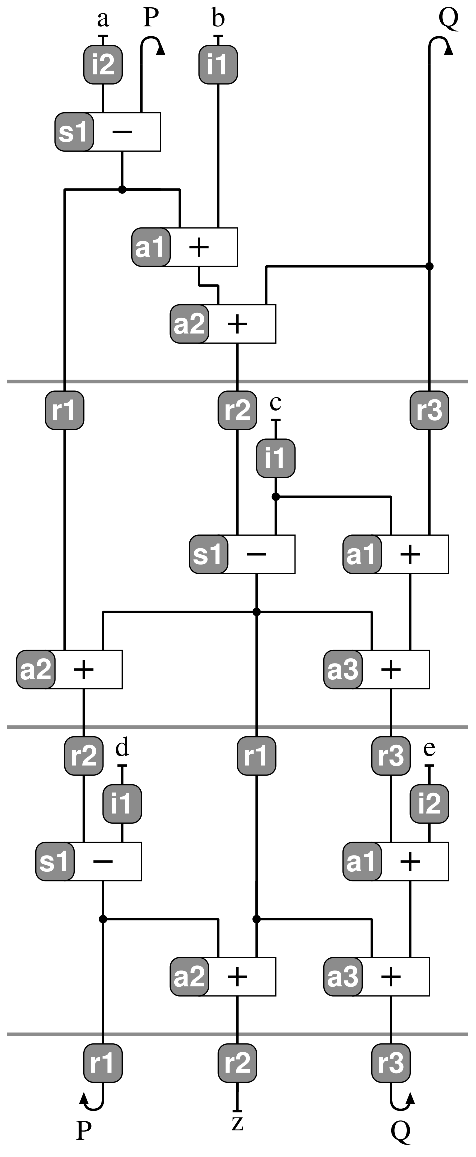
Q4 (25 Marks) DFD

(*estimated time: 18 minutes*)

Complete the allocation for the dataflow diagram below, draw the control table, and analyze your allocation.

NOTES:

1. Optimization goal: minimize number of multiplexers
2. You may *not* use any scheduling optimizations.
3. You may *not* change any of the existing allocations.
4. The *only* algebraic optimization you may use is commutativity.



Control Table

	a1		a2		a3		s1		r1		r2		r3		
	src1	src1	src1	src1	src1	src1	src1	src1	ce	d	ce	d	ce	d	
idle	—	—	—	—	—	—	—	—	—	—	—	—	0	—	
0	s1	i1	a1	r3	—	—	i2	r1	1	s1	1	a2	0	—	
1	~~i1~~ r3	~~r3~~ i1	~~r1~~ s1	~~s1~~ r1	s1	a1	r2	i1	1	s1	1	a2	1	a3	
2	r3	i1	s1	r1	r1	a1	r2	i2	1	s1	1	a2	1	a3	total
mux	1	0	1	1	1	0	1	2		0		0		0	7

Analysis

2:1 Multiplexers	6
Latency	3
Throughput	1/3
Clock period	sub + 2add + flop

Marking:

+13 marks *DFD*

- **13 marks** *7 mux solution*
- **12 marks** *8 mux solution*
- **11 marks** *9 mux solution*
- **10 marks** *10 or more mux solution*

+8 marks *Control table*

+4 marks *Analysis*

Q5 (15 Marks) Memory

(*estimated time: 10 minutes*)

Q5a (8 Marks) Optimization

Optimize the pseudocode below as if it were to be used as the specification for a dataflow diagram.

NOTES:

1. Optimization goals, in order of *decreasing* priority:
   (a) Minimize overall latency of the program
   (b) Minimize latency to z
2. It is guaranteed that `a` < `d`.
3. The variables `a`, `b`, `c`, and `d` do *not* change their values during execution.
4. Use the standard type of dual-port memory from ECE-327.
5. Scheduling and other optimizations are allowed, as long as they do *not* affect the final values of `M` or `z`.

```
M[a+1] = b
M[a+2] = c
M[d]   = M[a+2]
M[d+1] = M[a]   + M[a+2]
e      = M[a]
z      = M[e]
```

Answer:

```
M[a+1] = b
M[a+2] = c
M[d]   = c
e      = M[a]
M[d+1] = e    + c
z      = M[e]
```

Marking:

+2 marks *replace first `M[a+2]` with `c`*
+2 marks *replace second `M[a+2]` with `c`*
+2 marks *replace `M[a]` with `e`*
+2 marks *move `e` up*

Q5b (5 Marks) Latency

If you were to design a DFD to implement your optimized pseudocode, what would be the minimum possible latency to `z` and the minimum possible overall latency?

Latency to `z`	5
Overall latency	5

Marking:

5 marks *both latencies are correct*
3 marks *both latencies are incorrect but consistent with each other*
3 marks *one latency is correct and one is incorrect*
1 mark *both latencies are incorrect and are inconsistent*

Q5c (2 Marks) Single-Port Memory

If you switch to single-port memory, will you need to increase the overall latency of the program? **For full marks, you must justify your answer.**

Answer:
Yes. With dual port memory we can do a read and an independent write in the same clock cycle. This allows us to do line 3 (write to `M[3]`) and line 4 (read from `M[a]`) in the same clock cycle. Using single port memory would add increase the latency by one clock cycle.

Marking:

+2 marks *somplete and correct justification*

+1 mark *substantial correct information*