Eliot's Guide to Getting a Bare Pass in ECE327

VHDL (Legal / Synthesizable / Bad Practice)

Legal: The code will compile w/o errors.
Look for:
- typos
- incorrect syntax or structure
- timing-based statements in combinational processes

Synthesizable: The code can turn into hardware
Look for:
- initial values
- wait for X ns
- variables, bits, and booleans
- clocked process starting with an assignment (should be wait)
- different wait conditions
- multiple if rising-edge
- if rising edge and wait in the same process
- if rising edge with else
- a loop that has both combinational and clocked paths
- wait inside a for loop

Bad Practice: Your code is shit
Look for:
- latches (conditionals without else)
- asynchronous resets
- combinational loops (signal dependent on self)
- using a data signal as a clock
- clock as a data signal

- tri-state buffers and using 'Z'
- inout or buffer type
- multiple drivers (assigning signal in multiple processes)

Delta Cycle Simulations

1) Time is before 0. Initialize.
   - all processes set to resume
   - all signals set to default value
2) Time is 0.
   - execute all processes that rely on time change by calculating projected values, then suspend
   - execute all processes reliant on "other" values, calculating the projected value based on the "other" values' actual values, then suspend
   - note that since time has increased, this is not a delta cycle
3) If all processes are suspended, and there are projected values that differ from the actual values, copy the projected into the actual
   - resume, execute, change data, then suspend any process dependent on the changed value
   - begin next delta cycle for next process once all of the above are suspended
4) If all processes are suspended, and there are no changed projected values, increment time
5) Continue steps 3/4 until no values change, even when we keep incrementing time

Register-Transfer Level Simulation

1) Pre-process processes
   - separate into timed (wait for x ns), clocked (if rising edge) and combinational
   - deconstruct combinational processes such that they only mutate one signal per process
   - sort them based on which ones are dependent on other data signals (no dependency on other data signals would be at the top)
2) For each unit of actual time
   a) run timed processes in any order, reading OLD values of signals
   b) run clocked processes in any order, reading NEW timing values and OLD data values
   c) run combinational processes in sorted order, reading NEW values of signals

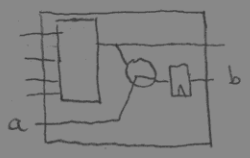
Mapping Circuits onto FPGA Cells

Essentially the idea is just to traverse backward from the output, stuffing as much combinational circuitry as you can into a single LUT, with 4 or less inputs.

One useful trick: any output of a flip flop MUST be the output of an FPGA cell

Another: any input that is directly flopped must have a direct connection to a flip flop,
ie.

a —|▯|— b must be

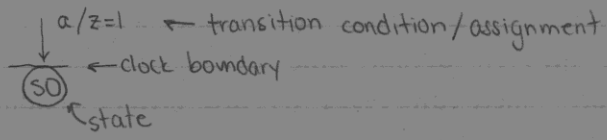


We're only allowed to use carry in/out for arithmetic circuits.

Area of Arithmetic Circuits

Skip this. Haven't encountered any exams that ask this kind of thing. We're looking for a pass, not for a 100%.

ECE327 State Machines



$z=1$ is a combinational (instant) assignment, while $z'=1$ is a registered (one cycle delayed) assignment

	Binary	One-Hot	Thermometer	Gray
S0	00	0001	0001	00
S1	01	0010	0011	01
S2	10	0100	0111	11
S3	11	1000	1111	10

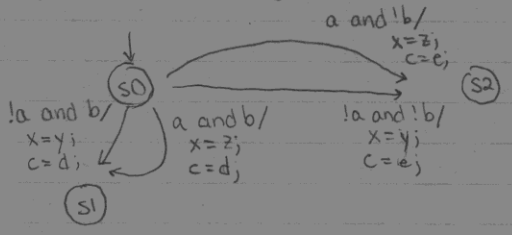
Remember to take care of representations of invalid states if your number of states isn't a power of 2.

types of machines - explicit - current
- one signal for current state
- state changed in a clocked process

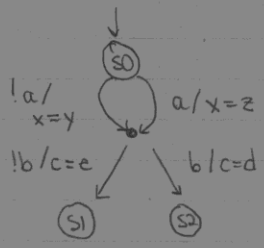
- explicit - current - next
- two signals, one for current, one for next
- next state set as combinational
- current state flopped from next as a clocked process

- implicit
  - no state signal
  - multiple waits (one for each state) in a process

If a signal is assigned multiple times in one transition, just take the last assigned value. You can also split up multiple assignments to multiple 'transient' states



can be turned into:



Dataflow Diagrams

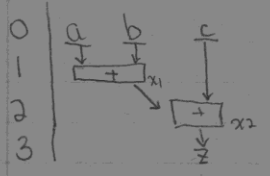
Basically you need to look at order of operations and figure out what values are dependent on what.

Inputs go on row 0. Each row thereafter is one clock cycle.

Each logic block can only take two inputs. Outputs of logical blocks can only be used the cycle after.

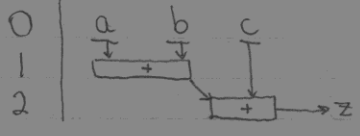
Registered (input/output) means they can only be read one cycle after

ex $z = a + b + c$



clock	0	1	2	3
a/b/c	V			
x_1		V		
x_2			V	
z				V

Registered input
Registered output



clock	0	1	2
a/b/c	V		
x_1		V	
x_2			V
z			V

Registered input
Combinational out

Latency = number of rows to get an output (including input row)

Components = number of logic blocks

Registers = number of signals that cross clock boundaries (horizontal lines)

Unconnected signal tails in a cycle = number of inputs

Unconnected signal heads in a cycle = number of outputs

You can put more logic blocks than 1 in a row, at the cost of increasing the amount of hardware required (but also decreasing latency).

Throughput

Throughput = parcels per clock cycle
= 1 parcel per X clock cycles

Max Throughput = max rate of parcels per cycle
= 1 / (min bubbles +1), generally

Actual Throughput ≤ Max (obviously)

We use greek letters to track parcels - all things labelled α are part of the α parcel.

Max thoroughput is determined by latency, pipelined, or superscalar (multiple parcels per clock cycle)

System	Thoroughput (Max)
Unpipelined	1/latency
Pipelined, not SS	Y/latency ≤ x ≤ 1
Fully pipelined, not SS	1
Superscalar	>1