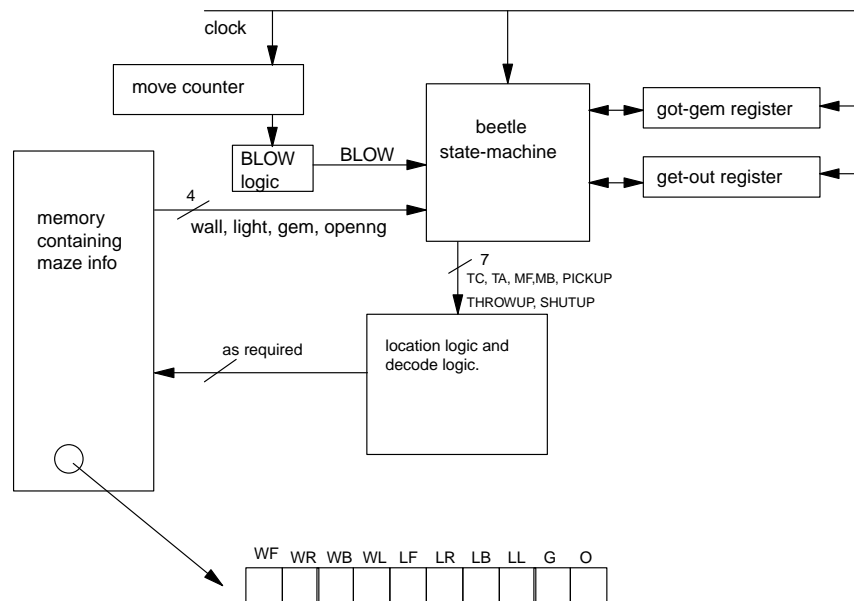


Case study

- 1) a) look for plausible examples of test-patterns that could drive a state machine through key sequences of events. This could use their own state machine design from Q2.a. (4)
 In general, testing using input patterns can prove that the response of the system are correct for a given input stream. For example you can test that finding a gem results in all further gems being ignored until the current gem is delivered. (2)
 Developing a thorough set of test sequences would be tedious and error prone, if done manually. (1)
 It would not be possible to test whether a robot could find an opening within the required 50 moves after BLOW goes active. (1)
 Disadvantage is that you are controlling the sequence of events. There is no feedback from the robot to manipulate the environment. Static testing. (2)
 Credit any reasonable discussion.
- b) This is a much more complex problem as the environment has to be modelled and interfaced to the robot. There are probably a number of ways to implement this so credit should be given for sensible suggestions. There is no requirement that it should all be modelled in a hardware description language so plausible explanations of interfacing the hardware to a C program should also be credited.
 One possible solution is to design, in vhdl, something akin to the diagram below. Not complete but gives an indication of a possible solution. This would require no change to the original state machine, it would become a component in a larger model.



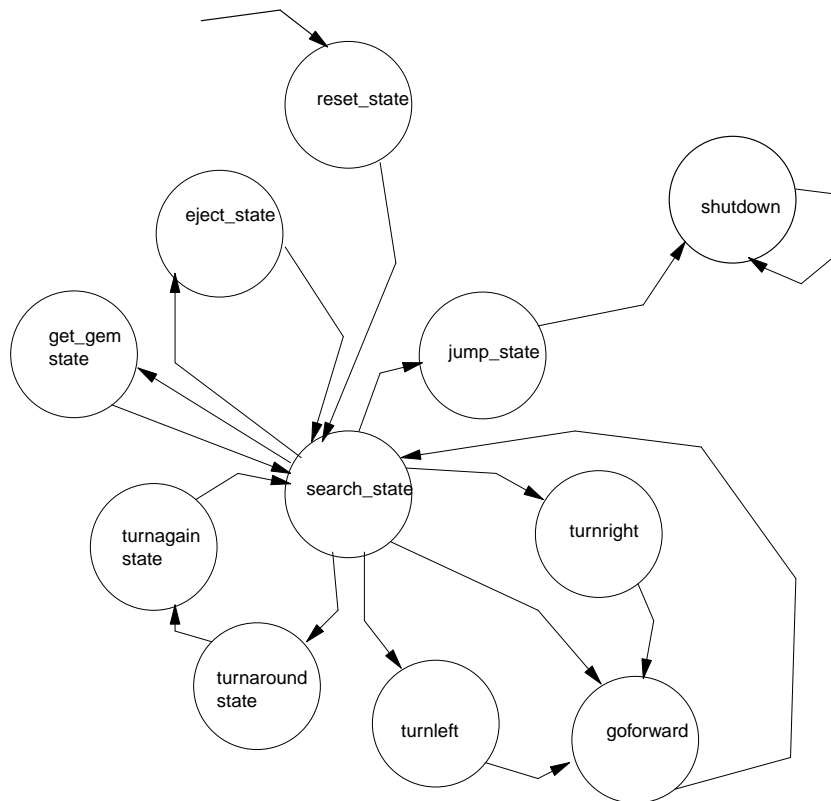
The system would need to keep track of current location and orientation. Location could map to a memory address, as it is custom hardware we can use 10 bits per address, 4 bits for walls, 4 bits for light, 1 bit gem, 1 bit opening. wall and light values can be rotated for different orientations. movements can map into memory address offsets and inc/dec. By decoding the movements/actions of the robot, the system can access a new input pattern from memory. This would allow for dynamic testing under simulation. The toolkit that has been used in the lab sessions supports generation of ROM and RAM VHDL models capable of being simulated. The VHDL entity, for the above example could contain the following inputs and outputs. This would allow for monitoring of the behaviour of the design under test. marks should be assigned as 10+ for sketch and written description and up to 5 for the

entity

depending on complexity.

```
entity sim_env is
  port (
    clock    : in  bit;
    reset    : in  bit;
    BLOW     : in  bit; -- this could be driven from the test-pattern.
    address  : out bit_vector(n downto 0); -- size to mem-size
    actions  : out bit_vector(6 downto 0) -- monitor behaviour
  )
end sim_env;
```

2) A possible state transition diagram is



A set of states such as this can be fairly easily derived from the spec 5 . Some of the events are straightforward, such as the mutually exclusive light signals. The wall sensors are not specified as mutually exclusive and it is possible that several wall sensors will indicate clear space at the same time, therefore some means of making them mutually exclusive is needed. look for evidence of resolving this in the design 3. +2 marks for elegant design

b) The entity should be derivable from the original spec + any extras that they have added to resolve design issues. 3 marks for matching to their fsm design, 2 marks for syntax. c) look for evidence of clear partitioning of the design. There is no exact answer as it is dependent on thier design. A possible solution could be a top level structural design with behavioural and structural subcomponents.

3) give marks according to quality ogf their discussion, up to 5. For the second part, the answer is that it will go round in circles 1, remainin marks on possible solutions. This is a possibly non-trivial problem in finding an algorithm that will never get stuck. One possible

solution would be to toggle a value so that it alternates between left and right turns. The priorities would be : always go forward if possible otherwise go left or right depending on the state of the toggle flag. invert toggle flag after moving. the equations would be

$$\begin{aligned}
 MF &= LF.(BLOW+haveagem) + !WF \\
 TL &= LL.(BLOW+haveagem) + !WL.WR.WF + !WL.!WR.toggle \\
 TR &= LR.(BLOW+haveagem) + !WR.WL.WF + !WL.!WR.!toggle \\
 MB &= LB.(BLOW+haveagem) + WF.WL.WR
 \end{aligned}$$

This could still result in looping paths that repeat.

Section 2

- 4) a) This kind of behaviour is common in real hardware devices and in simulation when the time units are fine-grained enough, or when clocked too fast 3. example of 74ls74 where setup time is 20nS, hold time 5nS and propogation delay is ≈ 40 ns 3. First glitch on q_bar is generated by the following :

B goes low when clock \uparrow as A and clear are high already
 therefore
 C \downarrow when clk \uparrow causing q_bar \uparrow

explanations similar to this 4 marks. The second & third glitches are the same but are merged into the correct behaviour of q_bar. 2 marks

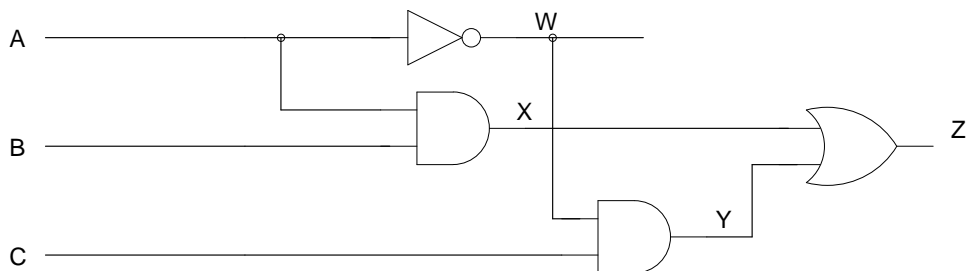
A solution is to use slower gates on the output stage, ie replace the delay on the expressions for QI and QbarI with

```

QI    <= not(prst and B and QbarI) after 2 ns;
QbarI <= not(QI and clear and C) after 2 ns;
3 marks
  
```

- b) Main issues are ascertaining setup, hold times etc 1. ensuring that the simulator uses appropriate delays in the feedback 2, eg use of -fd *time* flag to the simulator, simulator lock-up if no timing given 1.

- 5) a)



- b) something like this, the important thing is that the inputs are held high for long enough as propagation delay A→W is 4ns, and that there are changes on the a and b inputs 3. syntax 2.

```

in a;
in b;
in c;
out z;

begin
< 0 ns > p: 1 1 1 ?*;
< +1 ns > p: 1 1 1 ?*;
< +1 ns > p: 1 1 1 ?*;
< +1 ns > p: 1 1 1 ?*;
< +1 ns > p: 0 0 1 ?*;
< +1 ns > p: 0 0 1 ?*;
< +1 ns > p: 0 0 1 ?*;
< +1 ns > p: 0 0 1 ?*;
< +1 ns > p: 1 1 1 ?*;
< +1 ns > p: 1 1 1 ?*;
< +1 ns > p: 1 1 1 ?*;
< +1 ns > p: 1 1 1 ?*;
< +1 ns > p: 0 1 1 ?*;
< +1 ns > p: 0 1 1 ?*;
< +1 ns > p: 0 1 1 ?*;
< +1 ns > p: 0 1 1 ?*;
end;

```

c) the unit of 'non-time' taken by the simulator to calculate signal values to enable concurrent hardware to be simulated as a sequential series of steps 2.

d) The transitions should map their test-pattern. Credit either a written description in the form of ...

at time 0ns

a value of 1 on A causes a transaction of (0, 2ns) to be scheduled for W.

a value of 1 on A and B causes a transaction (1, 1ns) to be scheduled for X

...

...

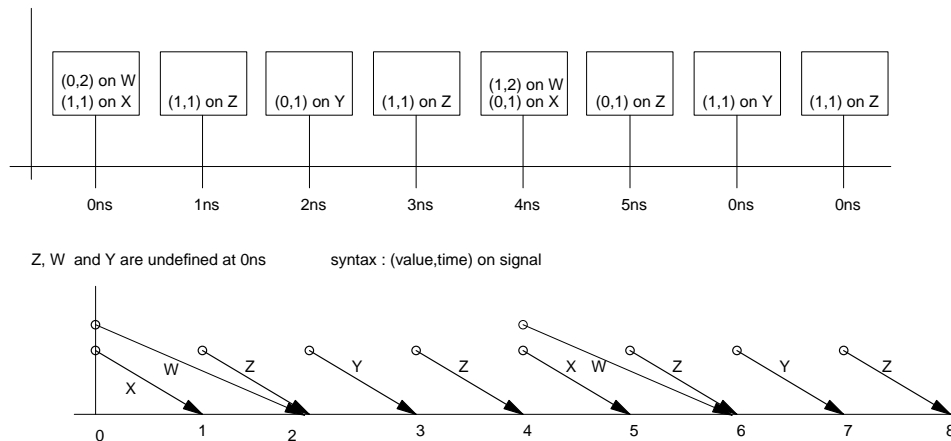
at time 2ns

W, which was previously undefined, goes to 0, this causes a transaction of (0, 1) to be scheduled for Y.

etc.

credit matching to pattern 5, reference to undefined value on W and Y 2, correct transitions 3.

or diagrams similar to



- 6) a) need to learn ... Design entry tools such as schematic capture or state machine capture for a more traditional hardware approach that can output VHDL. Text editor, with support for the language, eg emacs, to speed up code entry and syntax checking. Revision Control or Project support to keep code versions in order 3.

Simulator for testing the behaviour of the design, waveform viewer for test output analysis, static timing analysis tools. Issues around language subsets and converting data-flow descriptions to structural descriptions, Managing environment variables for library selection etc. 3

conversion to synthesisable code. It may be necessary to convert to a standard interchange format such as EDIF or SPICE in order to port the code to the CPLD synthesis program, unless the CPLD compiler can compile VHDL, but even then this will only be a subset of the language. 'Fitter' to take the output of the synthesis tool and program the device. Timing analysis tools for the 'fitted' design, hardware testing tools 3.

optimisation issues at all levels, optimising early in the process leaves the 'fitter' less opportunity to optimise for the target CPLD. critical paths, no-minimise paths eg extra gates added to cover glitches which could get optimised out. optimise for speed versus area? 3

Difficulties in moving from a data-flow description that is not synthesisable to a structural one that is. Use of language subsets to permit automatic mapping of data-flow into structural. Software engineers learning to think from a hardware perspective even though VHDL looks like a programming language. understanding the concurrent behaviour of hardware. 3.

b) credit examples from the lab sessions. software state machines are similar to data-flow vhdl descriptions but with subtle differences. The key points are that hardware state machines are more akin to control units, often the state value (bit pattern) will be used to drive outputs directly (Moore state machines). No memory in the sense of software variables. VHDL state machines can be described as case statements similarly to C but can be described in ways that are not synthesisable. structural vhdl requires mapping of the state machine excitation expressions, karnaugh maps etc. need to understand different state machine encodings, one-hot, mustang etc. 5