CS2002 Practical 4 - Stable Circuits

By: 150013565 Tutor: Juliana Bowles

Date: 10th April 2018

Abstract

A digital circuit with feedback is called stable if the value of each wire (which can be 0 or 1) remains unchanged over time. We say a circuit stabilises if eventually it becomes stable.

A 'circuit description language', CDL, is introduced to represent a circuit. By modelling this circuit in code, we can simulate circuits and determine whether they stablise or not.

Stage 1 - Modelling the CDL

The code for modelling the CDL is found in *circuit.h* and *circuit.c*. The data structure used to store the CDL is linked list. When processing the CDL, there is a struct for wires - Wire - and a struct for gates - Gates. A linked list to store all of the wires - WireList - and another linked list to store all of the gates - GateList.

The Wire struct stores two integer variables current value and previous value to store the current value and previous value of the wire. This is done to generate the output table. The Wire also stores the name of the wire and a boolean to represent whether it is an IN wire. The name of the wire is stored so that it can be uniquely identified in the list of wires since the specification specifies that all wires are uniquely named. The boolean value is used to determine output of the results of Stage 3 and used to calculate the max number of states a circuit can reach. The Wire also contains a reference to another wire which is used for the linked list.

The Gate struct contains contains three pointers to three different wires; two input wires and one output wire. This is done so that when the program updates the wires, the gates will automatically updated with the new values of the wires. The output wire is also stored so that when the gate is evaluated, the output wire will be updated with its new value. This pointer based solution makes the whole program a lot easier to code as I do not need to keep track which wires go to which gates - the gates know which wires go to them. The Gate struct also contains a string to represent what type of gate it is i.e. AND, NOR, XOR etc. It also contains a reference to another gate which is used for the linked list.

The WireList and GateList are practically the same structure, they store references to the head and tail of the linked list. The head is used to store the beginning of the linked list so that all the members in the list are accessible. The tail is used to add structs to the end of the linked list, this is done so that the CDL can be processed in the order it was written in.

The CDL is parsed using standard in. The input is read line by line using scanf. The read in line is then passed to *processLine* which determines how many words there are in the line and does a switch statement. This is done because, for example, an *IN* input will only have two words. Knowing this, the program can be process this as an "in" wire without needing to do any string operations such as *strcmp*. The same logic applies for three and four words. Three words is always {*output wire*} {*gate*} {*input wire*} and four words, it is always {*output wire*} {*gate*} {*input wire one*} {*input wire two*}.

Once this has been done, Wires and Gates can be created and added to their respective Wirelist or Gatelist.

Stage 2 - Simulating circuits

In my solution, to simulate a circuit is all done within one method. First the max time of the circuit is calculated. This max time is 2n̂umber of wires. This is because each wire has 2 potential states 0 or 1 and with each new state another permutation of 0 and 1 can be combined thus 2n̂umber of wires.

Once max time is calculated, the permutation array of inputs is calculated. If there are defined *IN* wires, finding all possible combinations of those wires is calculated. For example for two wires, the combination is {*0,0*}, {*0,1*}, {*1,0*}, {*1,1*}. Looking closely, this set represents the binary representation of {*0, 1, 2, 3*}. Therefore to generate all possible permutations, we must get the 2n̂umber of *IN* wires to get the total number of permutations.

Once all permutations are found, we go loop through all possible combinations. Inside the loop there is another loop which runs from 0 to max time representing the time of the circuit. At t=0, all wires are set to 0 and at t=1, all values of the wires are initiated. To initiate all wires, the permutation array is used to set the wires to the specified value in the permutation array. At t ¿= 2, the program updates the wires and then evaluate the gates.

To update the wires, the program traverses through the WireList. For each Wire in the list, the Wire's previous value is changed to the current value of the Wire. By updating the wires first, it ensures the gates behave in a synchronous manner.

To evaluate the gates, the program traverses through the GateList. For each Gate in the list, the Gate takes the input wires and evaluates them to produce an output depending on the type of gate it is. For example, if the gate was an AND gate and its input wires had previous values 1 and 0, the output wire would evaluate and set the output wire's current value to 0.

Once all gates have been evaluated, the result of the circuit is sent to an output array which is a 2D array which keeps track of all the wires and how they change as time goes on.

Stage 3 - Truth tables

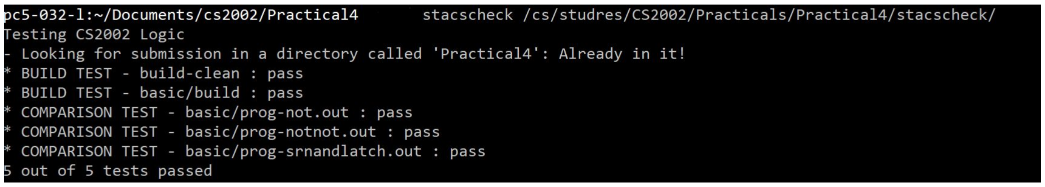
Once the circuit has finished simulating, we must calculate whether the circuit stabilises or not. To do this, we check the output table.

The method *checkStable* checks whether an *out* wire exists as if it does we only need to check the *out* row in the output table and see whether the last value in the row is the same as the second last value. If it is, then the circuit stabilizes. In the case where it does stabilise, the program returns the value it stabilises at, if not it returns 2 to signify that the circuit does not stabilise.
If an out wire does not exist, the output table iterates through its rows and checks whether the last value in the row is equal to the second last value in the row. If the circuit does stabilise it returns 0 as the specification requires, if it does not stabilise, the program returns 2 to signify that the circuit does not stabilise.

Once a circuit has been determined to stabilise or not, the program prints out the output table as required by the specification - printing all input wires, plus the result of the stabilisation check.

Testing

The program passes the stacshceck test provided:



```
pc5-032-l:~/Documents/cs2002/Practical4      stacscheck /cs/studres/CS2002/Practicals/Practical4/stacscheck/
Testing CS2002 Logic
- Looking for submission in a directory called 'Practical4': Already in it!
* BUILD TEST - build-clean : pass
* BUILD TEST - basic/build : pass
* COMPARISON TEST - basic/prog-not.out : pass
* COMPARISON TEST - basic/prog-notnot.out : pass
* COMPARISON TEST - basic/prog-srnandlatch.out : pass
5 out of 5 tests passed
```

Figure 1: Output given by stacscheck

The gates were individually checked to confirm that they do indeed work, this was cross checked against the lecture slides [2].

Gate.	Input A	Input B	Actual Output	Expected Output
NOT	0	NULL	1	1
NOT	1	NULL	0	0
AND	0	0	0	0
AND	0	1	0	0
AND	1	0	0	0
AND	1	1	1	1
NAND	0	0	1	1
NAND	0	1	1	1
NAND	1	0	1	1
NAND	1	1	0	0
OR	0	0	0	0
OR	0	1	1	1
OR	1	0	1	1
OR	1	1	1	1
NOR	0	0	1	1
NOR	0	1	0	0
NOR	1	0	0	0
NOR	1	1	0	0
XOR	0	0	0	0
XOR	0	1	1	1
XOR	1	0	1	1
XOR	1	1	0	0
EQ	0	0	1	1
EQ	0	1	1	1
EQ	1	0	0	0
EQ	1	1	1	1

I also manually tested the circuit using the SR latches provided in lecture slides, tutorials [3] and online [1] but it is difficult verify the results due to the assumptions defined in the specification.

Images of gate testing and latch testing are found at the end of this pdf.

Evaluation

One aspect of my solution I could have implemented better is generating the output table when simulating the circuit. I don't need to generate a massive table for checking whether a circuit is stable or not, I could just check whether the current value of each wire is equal to its previous value. This would be more space efficient since I will not be generating a massive 2D array. What I could have done better however, is pattern checking. When building up my output table rather than generating all outputs until the max time, an algorithm could be designed to detect whether a column has already been seen before. If it has, then the circuit will not stabilise. Using the example in the specification, on page three it describes a circuit as

```
w0 NOT w3

w1 XOR w1 w0

w2 XOR w2 w1

w3 XOR w3 w2
```

This generates the output table:

t	0	1	2	3	4	5	...
w0	0	1	1	1	1	0	...
w1	0	0	1	0	1	0	...
w2	0	0	0	1	1	0	...
w3	0	0	0	0	1	0	...

Evidently the output at *t=0* is equivalent at *t=5*. One way to detect this would be to keep track of all states seen before and compare it to the new state generated. If this new state has been seen before then the circuit does not stabilise. The caveat to this would be if the circuit did stabilise. In which case an if statement has to be written to check if the current state of the wires is equivalent to the previous state, if they are equal the circuit stabilises.

An outline for this algorithm would be as follows:

```
if (current state is equal to last previous state then circuit stabilises
and return 1) else go through each column in previous state array
and compare it to current state array, if a duplicate is found return
-1 to represent not stable or return 0 to represent that the current
state is a new state
```

Another problem with the output table is scaling. With large amount of user defined wires it may run overflow.

Another problem with my implementation that could be better is duplication of code. The GateList and WireList structure are the almost identical and the method of adding items into the list is the same. I could have generalised the code so that it wouldn't be writing the same code twice.

Conclusion

Overall this practical was extremely challenging. The hardest part of this practical for me was modelling the CDL in code. I struggled with finding a good way of representing the circuit in code.

I initially started with a graph theory approach - every wire and gate is a node - vertices - and everything is connected - edges. However, this lead to a potential problem of evaluating gates multiple times making the code redundant and wasting CPU resources.

This happens because one wire could potentially connect to multiple gates, modelling this behaviour would require a node containing an array of the other nodes it is connected to. For example, the algorithm could visit node1 which is a wire, node1 is connected to 3 other nodes which are gates. The first gate is chosen and for discussion purposes, lets assume it is an AND gate. The AND gate requires two wires, node1 and node5, by evaluating this gate, I have already evaluated part of node5. If the algorithm would proceed to evaluate node5 and evaluate the same AND gate processed earlier.

With small CDLs this isn't really an issue. However, when the CDL is larger with more wires and more gates, this could result in multiple duplicate calculations leading to longer run times and wasting CPU time.

I changed my solution to modelling the CDL shortly after discovering this and after it was becoming more and more complex to debug maintain this graph theory approach.

From this practical I have learned that planning well in advance is something that I need to get better at. I spent a lot of time trying to implement a graph theory approach for it to only go to waste.

Extensions

File IO

I initially misread the practical specification, I thought the program was designed to read from a *.txt* file and output to a *.txt* file. Rather than removing the code altogether, I have included it as part of an extension. This extension isn't incredibly useful since you can pipe the *.txt* file in using bash, however, the functionality is there.

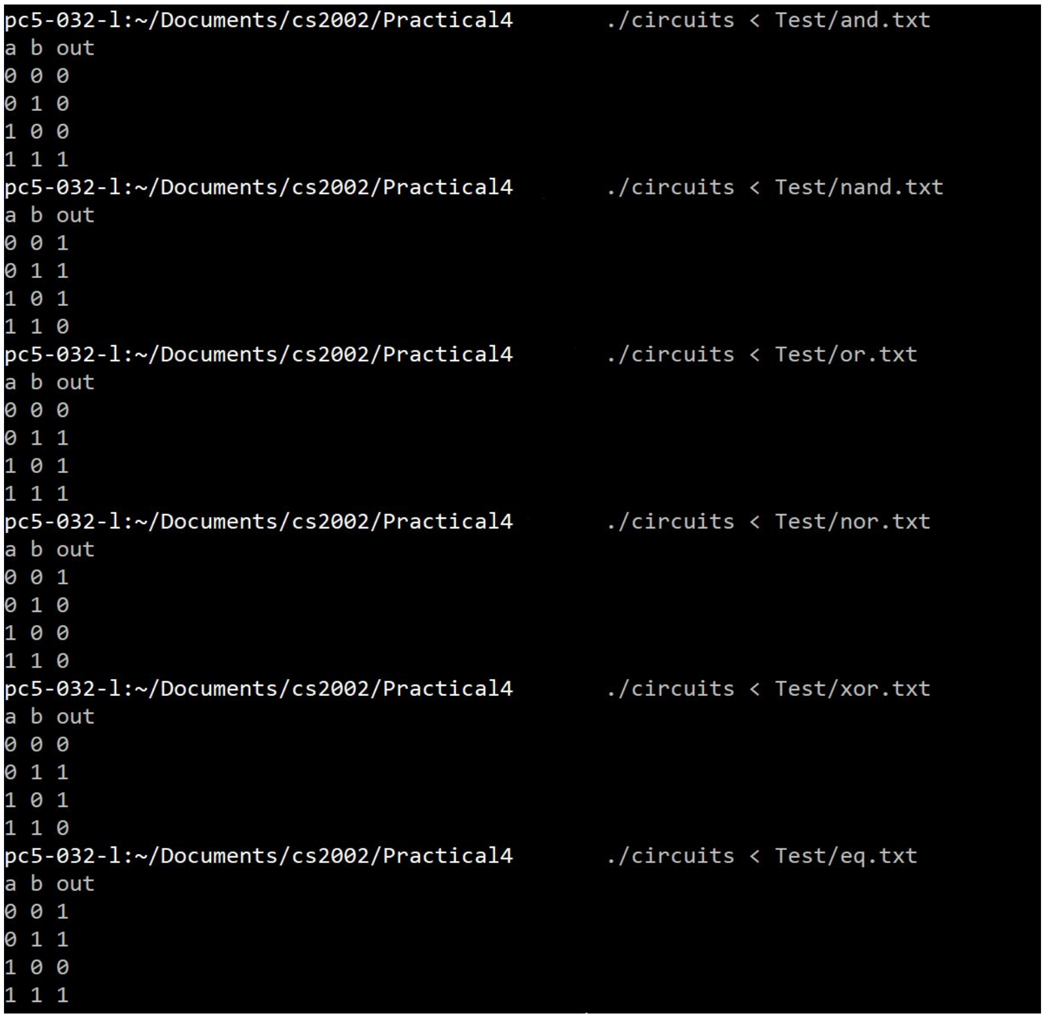
To run this extension simply run: `circuits input.txt output.txt`

References

[1] ELECTRONICS TUTORIALS. Sequential logic circuits. `https://www.electronics-tutorials.ws/sequential/seq_1.html`, 2018. [Online; accessed 10-April-2018].

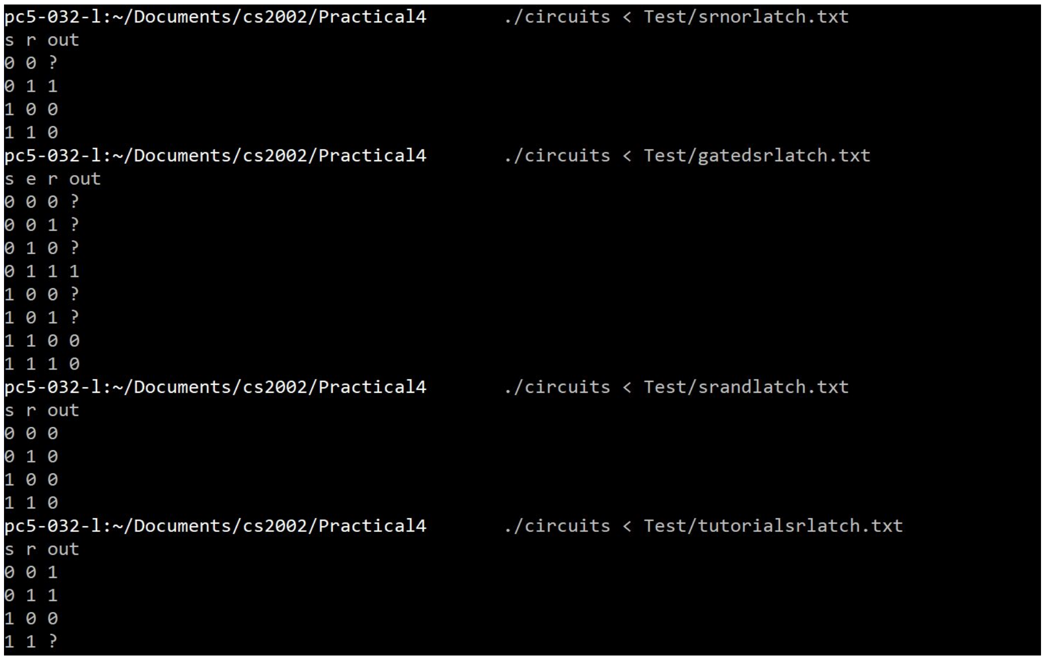
[2] STUDRES. CS2002 Lecture 10. `https://studres.cs.st-andrews.ac.uk/CS2002/Lectures/2018-03-09-LA10.pdf`, 2018. [Online; accessed 10-April-2018].

[3] STUDRES. CS2002 Tutorial Week 8. `https://studres.cs.st-andrews.ac.uk/CS2002/Tutorials/TutorialWeek8.pdf`, 2018. [Online; accessed 10-April-2018].



```
pc5-032-l:~/Documents/cs2002/Practical4       ./circuits < Test/and.txt
a b out
0 0 0
0 1 0
1 0 0
1 1 1
pc5-032-l:~/Documents/cs2002/Practical4       ./circuits < Test/nand.txt
a b out
0 0 1
0 1 1
1 0 1
1 1 0
pc5-032-l:~/Documents/cs2002/Practical4       ./circuits < Test/or.txt
a b out
0 0 0
0 1 1
1 0 1
1 1 1
pc5-032-l:~/Documents/cs2002/Practical4       ./circuits < Test/nor.txt
a b out
0 0 1
0 1 0
1 0 0
1 1 0
pc5-032-l:~/Documents/cs2002/Practical4       ./circuits < Test/xor.txt
a b out
0 0 0
0 1 1
1 0 1
1 1 0
pc5-032-l:~/Documents/cs2002/Practical4       ./circuits < Test/eq.txt
a b out
0 0 1
0 1 1
1 0 0
1 1 1
```

Figure 2: Manually testing the gates



```
pc5-032-l:~/Documents/cs2002/Practical4       ./circuits < Test/srnorlatch.txt
s r out
0 0 ?
0 1 1
1 0 0
1 1 0
pc5-032-l:~/Documents/cs2002/Practical4       ./circuits < Test/gatedsrlatch.txt
s e r out
0 0 0 ?
0 0 1 ?
0 1 0 ?
0 1 1 1
1 0 0 ?
1 0 1 ?
1 1 0 0
1 1 1 0
pc5-032-l:~/Documents/cs2002/Practical4       ./circuits < Test/srandlatch.txt
s r out
0 0 0
0 1 0
1 0 0
1 1 0
pc5-032-l:~/Documents/cs2002/Practical4       ./circuits < Test/tutorialsrlatch.txt
s r out
0 0 1
0 1 1
1 0 0
1 1 ?
```

Figure 3: Manually testing other latches found online