Logic CAD of VLSI Design - 046880

Topic: Event-Driven Simulation

Background

Interpreted simulators use parser generated database to help perform the simulation. The general flow of this procedure is illustrated in the following figure:

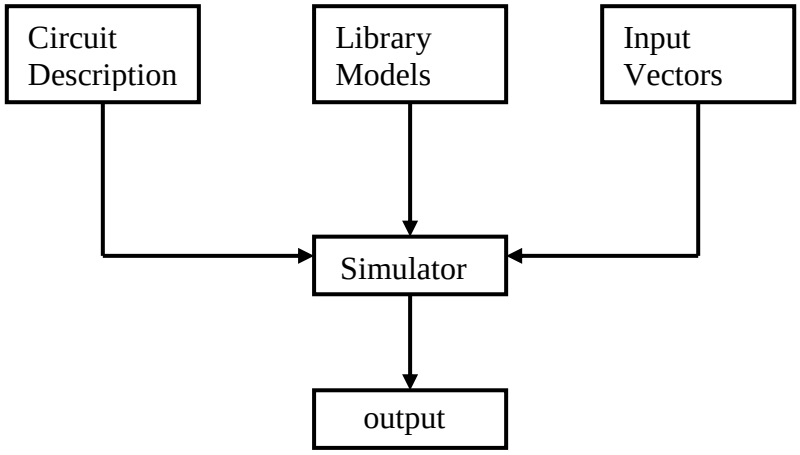


Figure 1 Interpreted Simulation Flow

In this technique, the circuit netlist is read by a parser and the appropriate data structures are then dynamically created in memory. A separate simulation kernel performs the simulation, by operating on the data structures and using information supplied by the library models, in accordance with the user-supplied input vectors.

In compiled code simulators, all gates are evaluated once for each input vector. However, if an input of a gate does not change, its output will remain the same and there is no need to simulate it again. This suggests a method for improving the speed of the simulation by reducing the number of gates evaluated per input vector.

Interpreted simulators usually employ an **event-driven** scheduling algorithm to determine which gates are to be evaluated. The main function of this algorithm is to detect events and to schedule gate simulations in response to them. If no events occur then no gates will be simulated.

All nodes have a current value associated with them. If an event causes a node value (a gate's output) to change, the node's current value will be updated at some future time. In the **unit delay model,** the change in the net value will occur one unit of time later, i.e. a new event will be scheduled for processing in the next time unit. At any particular time, gates are evaluated by processing all events scheduled for the particular time slot and using current node values i.e. values computed in the previous time slot. It is assumed that an arbitrary number of simulated time units can occur between successive input vectors. The simulator moves on to the next vector, only when it has finished processing all events directly or indirectly created by the previous input vector.

Input Vectors and Signal parsing

The test vectors file format holds on each line a hexadecimal string which encodes the values of the concatenated set of input signals. Each line on the file represents a single time unit.
For example, given the signals file includes the following lines:

```
a[3:0]
b[1:0]
clk
```

Each line in the test vectors file should be encoded as hexadecimal numbers of the following binary number: clk,b[1],b[0],a[3],a[2],a[1],a[0]. The LSB is on the right.

Note that when the number of bits is not a multiple of 4 the MSB extra bits on the most significant digit should be ignored.

For example, a vectors file:

```
38
63
```

Represents the two input assignments:
2'h38 = 8'b00111000, clk=0 → a=4'b1000, b=2'b11, clk=1'b0
2'h63 = 8'b01100011, clk=1 → a=4'b0011, b=2'b10, clk=1'b1

A reference for VCD file format could be found at :
http://en.wikipedia.org/wiki/Value_change_dump

Detailed Tasks

You are required to write a program that accepts

a. a set of gate-level Verilog files and the name of the top-level module
b. a tests signals and vectors files.

The program should read all input vectors and should simulate the circuit for each input vector. You should assume a single time unit is taken by each line of the input vectors file. The output of the program is a VCD file holding at least all the signals on the interface of the top module.

The simulation program will need to read in a signals file, a test vector file, and apply the test vector to the Verilog model inputs. The simulation results should be written into a VCD file named ***<top-cell-name>.vcd*** and should include all the interface signals of the design (meaning all the external signals).

In your code you can utilize, but does not have to, the library provided in the hcm/hcmvcd directory described below. You are also encouraged to use the library provided in hcm/sigvec to parse the pair of signals and vector files described below. You can assume that hcm/hcmvcd and hcm/sigvec are working correctly and you can implement your code on the data which recived using these libraries.

[Dry] Questions Part 1

Before you start writing you program, answer the following questions in your report:

1. Describe the data structures (including details of linked lists and queues) you will use to hold the elements in the netlist.
2. Describe (with the aid of a flow diagram or pseudo-code) the algorithm you use to control scheduling of gate evaluation (your event queue, etc.).
3. Explain how time is processed by your simulator. Is there a modeling of the time within a single cycle (time unit) ?
4. Are there any restrictions on the type of input circuits (constructed with the basic gates) for your simulator ? Explain. Make sure **you do not** restrict simple cases, that can be easily handled by event driven simulators.
5. Describe (with the aid of a flow diagram or pseudo-code) the algorithm of the complete simulator program.

[Wet] Programming Requirements and Execution

Write the program you described in "Questions – Part 1". You are expected to use HCM code for parsing the Verilog files and data model.
The program should work in the following manner after all the setup:

1. Accept a vector and apply it's value to the circuit.
2. Simulate the circuit.
3. Write the results to the VCD file.

All the files you need are given to you in the VM in directory wet02. you should change stdcell.v accordung to the instructions **below.**

Simplifications of the program:

1. The simulator can hardcode the function of the primitive cells. It is only required to recognize the "primitives" that are included in the file ***stdcell.v*** of the HCM code.

2. You are not required to implement smart "Initialization" or deal with ternary logic (0,1,X). Instead you can assume all flip flops are initialized to 0 before the simulation starts.

3. You may, and probably should, use the flattener code provided with HCM to code your simulator on a "Flat" folded model. "Scalability" for very large models is not required. NOTE: don't write out flat verilog and read it again – simply use the generated flat cell. (the syntax of the written out flat veriog is not perfectly legal).

StdCell.v and DFF

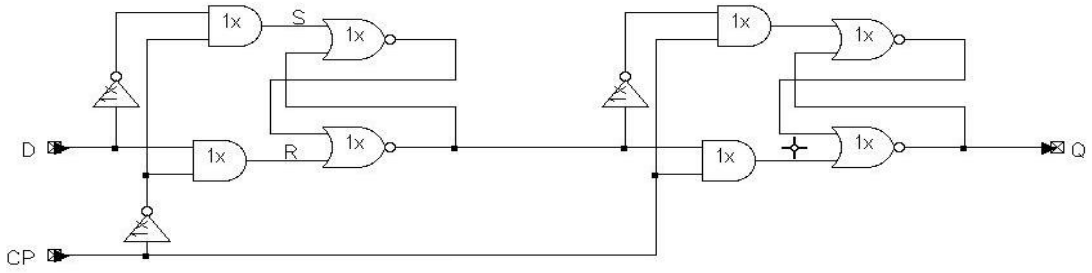


Fig. 2

1. Fig. 2 illustrates a gate level implementation of the DFF. Change the stdcell.v file to include this description for dff module and your program to be able to analyze it correctly. Your simulator must be able to evaluate this circuit ! (modify if not). Put your stdcell.v in your work folder and use this file instead the one in the ISCAS-85 folder.

2. If you have modified the algorithms you described in Question 1, explain how and why. If you did not need to change your program, explain what were the features that were critical for handling the circuit in figure 2 and were not required when the DFF was modeled as block box.

Note – In the end, your code should be able to simulate both cases:
1) DFF as a primitive without internal implementation – the function of the DFF should be implemented in your code. (The test will be using the stdcell.v from the folder ISCAS-85 where the DFF does not contain implementation).

2) DFF are implemented according to Fig.2 which means they are not primitives (you should write the implementation to a stdcell.v file which you add to your subbmition). Hence, your code should be able to handle the loops of the DFF implementation (you can base your code on the names and implementation of the DFF which you submit in the stdcell.v).

Note – your stdcell.v should be named stdcell_FF.v when submitting.

[Dry] Questions – Part 2

After you have finished implementing and testing your program answer the following questions in your report:

6.1 What is the complexity of your algorithm with respect to the number of gates in the circuit ? Suggest methods that could be used to optimize your simulator in order to reduce the run time.

6.2 Give a detailed list of the functions your program performs (in terms of procedures called) each time a gate is simulated.

6.3 Give a detailed list of the functions your program is required to perform (in terms of procedures called) each time an event is added or removed from the event queue.

Stage A – Build and write your own programs with HCM

1. **cd HCM**
2. **make** – not needed if you followed the workshop
3. **cd wet02**
4. Write your code in HW2ex1.cc file.
5. **make**
   NOTE: the tests for this exercise are automatic!
   If your submission will not compile on the virtual machine – it will be grated ZERO!

Stage B – Test your own programs with HCM

After finishing Stage A, make sure you run the following command from wet02 directory.

Now you are ready to generate the required output files for self-testing before submitting your work.

To generate TopLevel2806.vcd and TopLevel3540.vcd files run the following lines in the Virtual machine, in "wet02" directory.

- **./event_sim TopLevel2806 c2806.sig.txt c2806.vec.txt stdcell.v c2806.v**
- **./event_sim TopLevel3540 c3540.sig.txt c3540.vec.txt stdcell.v c3540.v**

We provided you the files TopLevel2806.vcd and TopLevel3540.vcd to compare with your outputs.

Stage C – Create your submission

After finishing Stage B, make sure your wet02 directory include only the .cc files, the stdcell_FF.v and your final report.
Run the following command from wet02/ directory -

1. **cd ..**
2. **tar czf wet02_<id1>_<id2>.tar.gz wet02**
   (<id1> and <id2> are the id numbers of the students)
3. Upload wet02_<id1>_<id2>.tar.gz to Moodle
   (Example of the file name: wet02_123456789_147852369.tar.gz)

- Make sure that only the .cc, stdcell_FF.v and your report files are in the folder !

Note: Bold lines are execution commands for the Linux environment (LUX).
Note: The following will be taken into consideration when grading your assignment:

a. The correct modeling of the behavior of the circuit.
b. Clear answers to all dry part questions.
c. The quality of the code and the documentation.

Topic	Event-Driven Simulation
Exercise owner	Yoav Cohen - yoavneta1@technion.ac.il
Given Code files	HW2_ex1.cc Makefile
Given Input verilog files	c3540.v c2806.v
Given Input vector files	c3540.vec.txt c2806.vec.txt
Given Input signal files	c3540.sig.txt c2806.sig.txt
Given Extra Verilog files	stdcell.v
Given Output files for self-testing	TopLevel3540.vcd TopLevel2806.vcd
Files to submit	HW2_ex1.cc stdcell_FF.v HW2.pdf
Submission format	wet02_ < id1 > _ < id2 >.tar.gz
NOTE: the tests for this exercise are automatic! wrong submission format will be grated ZERO!	

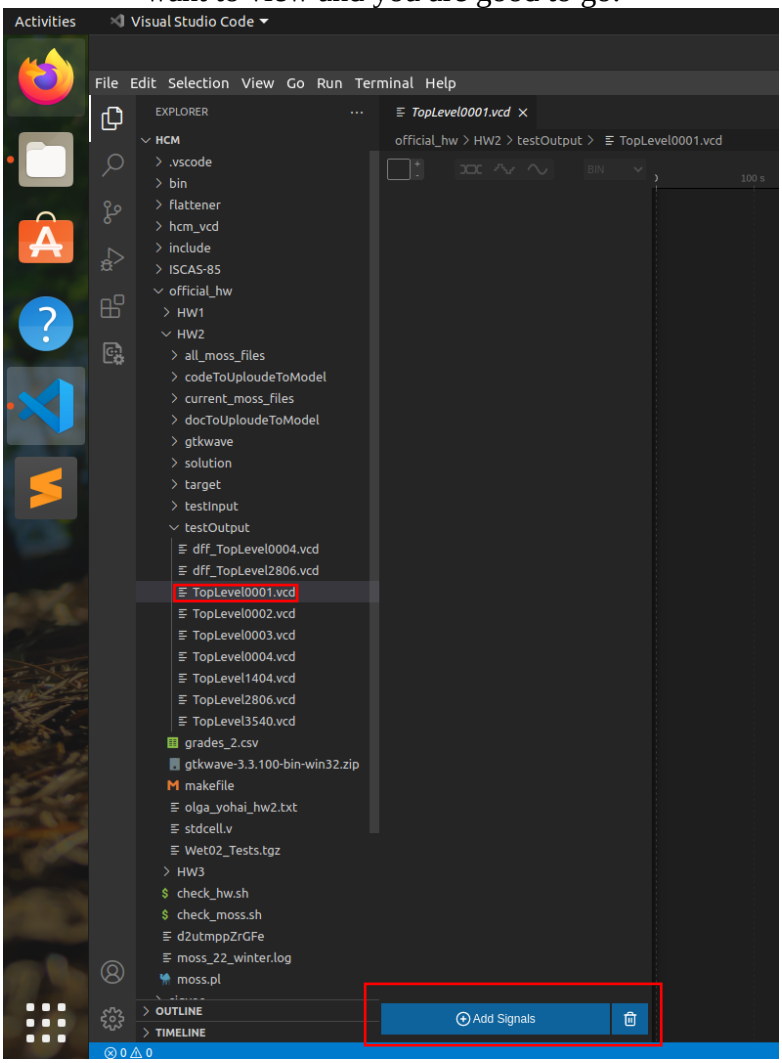
Questions about this WET exercise should be posted on Moodle. You required to follow the answers of the course staff and write your code according to the answers.

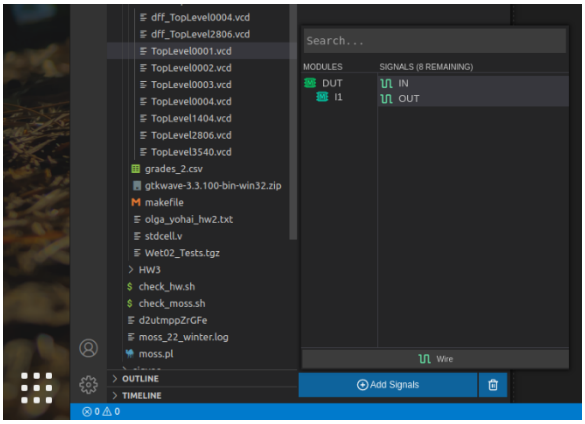
Good luck!

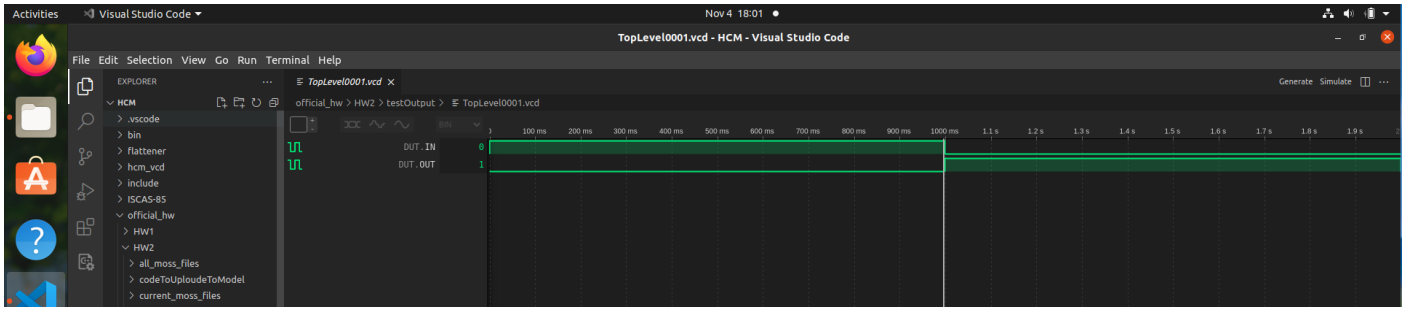
View VCD files as Waves using the Virtual machine

Once your program is running and generate VCD files you can use the built in WaveTrace Extension in the Visual Studio code you have installed in you VM.

In Visual Studio code open the VCD file you want to inspect, add the signals you want to view and you are good to go:







VCD Formatter Library

To reduce the overhead of learning the details of VCD file format we provide you with an implementation of a VCD formatter that is tailored to HCM objects.

The code is located in the hcm/hcmvcd directory where the header file you need to read is hcmvcd.h and an example program named main.cc show how to use it.

The formatter is provided as an object that once instantiated will write the VCD file header for the hierarchy below the given cell. After initialization, the user can call 2 functions:

- changeValue(nodeCtx, newVal) : given an INTERNAL node context (node and its parent instance list) and a value – write the VCD line describing the new value
- changeTime(t) : given the time, as unsigned int, write a new time line into the file

NOTE that VCD is only supporting nodes that are not connected above the cell. In other words: only nodes that do not have ports.

The below code section is given cellName and the topCell pointer to that cell folded model. It calls a fucntion to randomize some value changes and write the VCD file accordingly. See the header file for exact definition of the hcmNodeCtx.

```
vcdFormatter vcd(cellName + ".vcd", topCell, globalNodes);
if (!vcd.good()) {
  printf("-E- Could not create vcdFormatter for cell: %s\n",
        cellName.c_str());
  exit(1);
}

map< const hcmNodeCtx, bool, cmpNodeCtx > valByNodeCtx;
list<const hcmInstance*> noInsts;

// randomize some changes of values and write them out
for (unsigned int t = 1; t < 100; t++) {
  for (int i = 0;  i < 20; i++) {
    hcmNodeCtx *nodeCtx = getRandomNodeCtx(topCell, noInsts, globalNodes);
    if (nodeCtx) {
      if (valByNodeCtx.find(*nodeCtx) == valByNodeCtx.end()) {
        valByNodeCtx[*nodeCtx] = 1;
      }
      bool newVal = !valByNodeCtx[*nodeCtx];
      valByNodeCtx[*nodeCtx] = newVal;
      vcd.changeValue(nodeCtx, newVal);
    }
  }
  vcd.changeTime(t);
}
```

Signals and Vector Files Parsing Library

To reduce the overhead of writing code to parse the signals list and vector files and prevent bugs in the input section, we provide you with an implementation of a parser for the two files that provides a simple API to read each line of the vectors file ad obtain signal values.

The code is located in the hcm/sigvec directory where the header file you need to read is hcmsigvec.h and an example program named main.cc show how to use it.

The library provids an object named hcmSigVec that once instantiated will open up the given signals and vectors files provided in the constructor, prepare the list of signals and be ready for further calls to read each line of the vectors file and answer queries about specific signals value.

The following code is taken from the main.cc of that same directory and extended with some explanations.

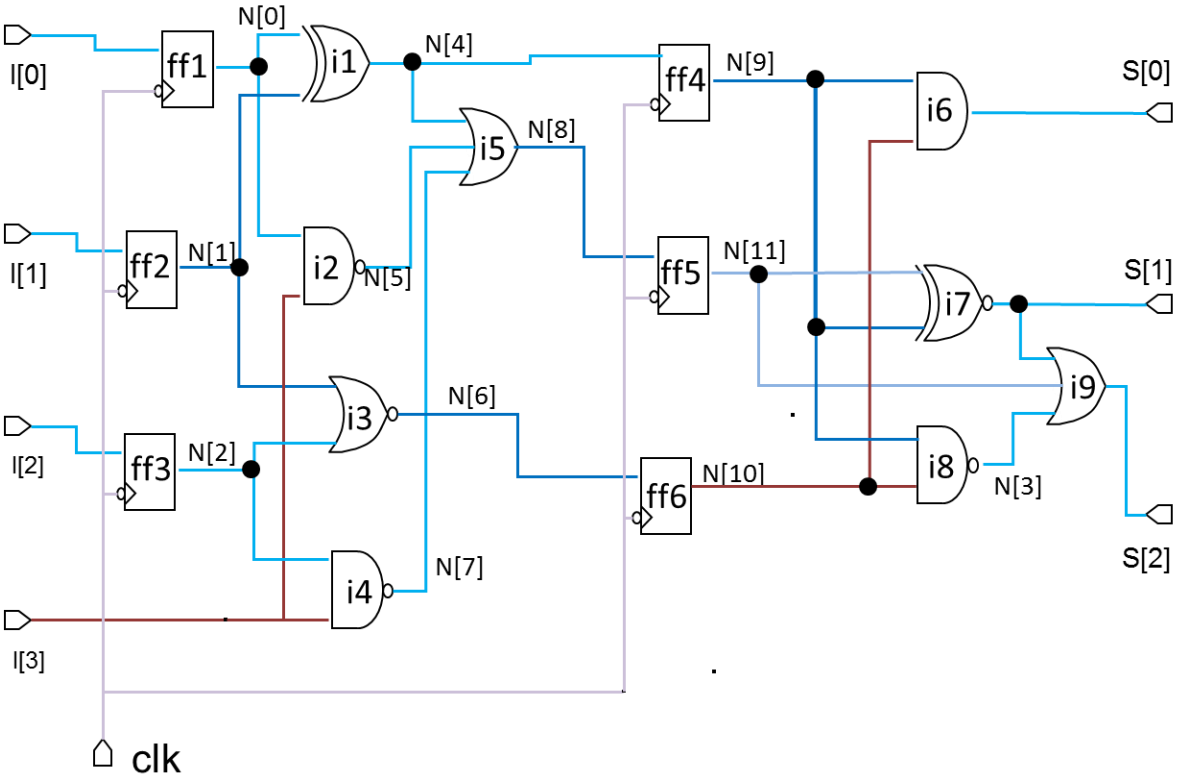
```
#include “hcmsigvec.h”
// instantiate the Signals and Vectors parser
hcmSigVec parser(sigsFileName, vecsFileName, verbose);

// obtain and print the list of signals extracted from the sigsFileName
parser.getSignals(sigs);
for (set<string>::iterator I= sigs.begin(); I != sigs.end(); I++) {
      cout << "SIG: " << (*I) << endl;
}

// read the vectors file one line at a time until the eof
while (parser.readVector() == 0) {
      for (set<string>::iterator I= sigs.begin(); I != sigs.end(); I++) {
            string name = (*I);
            bool val;
            parser.getSigValue(name, val);
            cout << "  " << name << " = " << (val? "1" : "0")  << endl;
      }
}
```

NOTES:

- Your Makefile will need to add to the include path the sigvec dir **-I$(HCMPATH)/sigvec** such that #include “hcmsigvec.h” will work.
- Similarly you also need to include the libhcmsigvec.so path and library: LDFLAGS will need to contain also: **-L$(HCMPATH)/sigvec -lhcmsigvec**

SEQ1 Circuit

This is an example circuit of some logic and state.



ISCAS-89 Circuit c3540
Provided in: http://web.eecs.umich.edu/~jhayes/iscas.restore/c3540/c3540.html

This benchmark is an 8-bit ALU that can perform binary and BCD arithmetic operations as well as logic and shift operations. Logic operations are intermixed with arithmetic ones, much as in the TTL 74181. BCD addition is done via a two's-complement adder by adding 6 to both digits of the first operand, and then subtracting 6 from the digits of the result if they do not generate a carry.