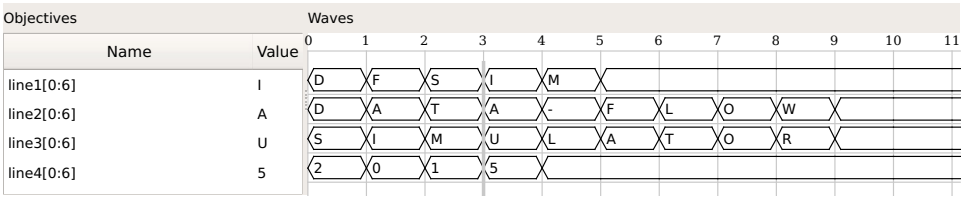
Excel@FIT 2015
http://excel.fit.vutbr.cz

An Interactive Simulator For Data-flow Graphs

David Kovařík*



Abstract

Data-flow graphs are, for their native support of high level of parallelism, often used by hardware designers. However, such graph representation is also very useful for performing deeper analysis of the design (including functional or formal verification). Simulator presented in this paper is a support tool for verification environment HADES. The goal of the simulator is to perform an efficient simulation of a verified model and to enhance user's knowledge about the model and its behavior. To perform a simulation efficiently, we introduce a specific simulation algorithm which saves computation time by eliminating redundant evaluations. The simulator is equiped with several output interfaces connected to a single simulation core. One output interface provides direct simulation output in text format. The second is also textual, but allows user to enter commands in order to control the simulation. Finally, the third forms a graphical interface in order to visualize results of simulation process. Thus, the simulator provides a scriptable command line interface to let users write automated tests as well as a powerful visualization tool for users to better understand behavior of the model.

Keywords: Modeling — Simulation — Data-flow graphs — Visualisation

Supplementary Material: Demonstration Video

*xkovar66@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Data-flow graphs [1] are commonly used as an input for advanced hardware analysis such as formal or functional verification. Output of verification process is usually provided in form of counter-examples which might be hard to associate reversely with the analyzed model. Simulator `dfsim`presented in this paper was created as a support tool for verification environment `HADES` [2]. Its goal is to help a user to better understand a verified design and to help with locating a specific parts of its simulation (e.g. faulty behavior of a design). The purpose of the simulator is to decrease necessary time and increase quality of the whole verification process.

Our work focuses on creating suitable representation of an internal simulation model and an efficient simulation algorithm. Equally important is presentation of simulation results to a user. Our goal is to provide textual interface for command line usage, which allows a user to create automated tests, as well as a powerful graphical interface for visualization of the simulation process. The reason to provide an interface for automated scripts is to give a user a tool to automatically detect behavior patterns of a design. On the other side, visualization part is meant to help a user to better understand behavior of a design. To make working with the simulator as efficient as possible, we

want to let a user control both simulation process and internal model's state interactively.

Currently, there are many tools for simulating hardware designs. One of the well-known is ModelSim [3]. It is a very complex and powerful proprietary tool for simulation of hardware description languages such as VHDL and Verilog. The simulation can be controlled in graphical user interface or by automated scripts.

Another example is GHDL [4], an open-source simulator of VHDL. It compiles VHDL source into an executable file, which then performs the simulation. GHDL does not provide any form of visualization. An output of the simulation is given in VCD format [5]. It is an ASCII-based file format, which records value changes of observed entities. To increase usability with other simulation or analysis tools, the presented simulator uses VCD file format as a default output format for performed simulations.

Icarus Verilog [6] is an open-source simulation and synthesis tool for several standards of Verilog. It operates as a compiler, compiling Verilog source code into a target format. It is designed to be extensible, thus it supports plug-in installation. As with GHDL, it does not contain an interface for visualization of the simulation. Thus, the visualization depends on other tools.

An example of a visualization tool is GTKWave [7]. It is designed to efficiently visualize large amount of entities. It supports variety of input file formats (including VCD format mentioned before). Its graphical interface was an inspiration for the visualization part of the presented simulator.

In `dfsim`, we designed graph representation of an internal model which provides an easy way to perform computational operations. We also developed an efficient simulation algorithm, which does not perform unnecessary evaluations and saves computing time. We decided to split output interface of the simulator into several standalone interfaces, so each can focus on different purpose. The first interface is purely textual and does not provide interactive tools. The next one is also purely textual, however it is interactive and can be used to run automated scripts. Both of these interfaces produces simulation results in VCD format, so they can be use by third-party tools. Finally, we also implemented graphical interface to visualize the simulation process in form of waveform graphs.

By introducing the simulator, we want to increase quality and decrease necessary time of verification process of microprocessor's designs. Its prototype implementation has been used to verify counter-examples found by verification environment HADES. During development we also detected several flaws in VAM file format specification.

2. Data-Flow Graph Paradigm

Data-flow graph [8] is a directed graph where each node represents a function and each arch is a data path among them. One node can interact with another if and only if they are connected with a data path – no *side effects* are allowed (e.g. usage of global variables, ...). In general, an arch represents a FIFO queue, where *tokens* (units of data) are stored.

Data-flow graphs (DFG) are *data-driven*, meaning that each node can *fire* (perform its computation) whenever there is sufficient amount of tokens on its input arcs. A node with no input arch can fire any time. It implies that multiple nodes can fire simultaneously. Therefore, DFG are said to be *untimed*. It means that timing is irrelevant for node's firing, because it is driven by presence of data (there is no clock signal or program counter). It does not mean that node's firing is performed in zero time. Since there is no central element to control the execution, a *schedule*, which tells when each node should fire, has to be introduced.

Number of data tokens consumed and produced by node's firing can be specified for each node. DFG is *synchronous* (SDFG) if the numbers are specified explicitly for each node and do not change in run-time. This type of DFG allows to create a static schedule which does not change. It saves computation time (the schedule does not need to be re-assembled in run-time), but it makes SDFG unsuitable for control-flow constructs (like *if-then-else statement*). On the other hand, *asynchronous* DFG allows to model control-flow constructs, but their schedule has to be modified in run-time.

3. Building an Internal Model

Before simulation itself can be executed, there are several major steps which needs to be done. It includes reading an input model, parsing it, building an internal model, and creating a schedule which defines the order of node firing.

3.1 Input Model Format

Reference input format for `dfsim`is `VAM` [9]. It is both language and model which describes single-pipelined microprocessors on the *register transfer level*.

Model in VAM is a graph where node represents either a storage (register or memory) or a functional circuit. Signals, which interconnects nodes, represent directed arcs of a graph. A single signal can connect multiple nodes, however two nodes can not write to

the same signal. A storage is able to keep its value for unspecified amount of time. They have zero read and unit write delays.

When a functional node fires, it uses its input signals to evaluate a function it represents. The result of the evaluation is propagated to all its output signals. Both function evaluation and value propagation is zero-delayed. Thus, an input model must not include loops formed only from functional nodes.

VAM format is also used as an input format for HADES. Since dfsimis a support tool for HADES, it is most reasonable to use same input formats. However, some constructs and attributes of VAM model are not important for the simulation, therefore they are ignored. An example of VAM description of 4-bit counter is shown in Figure 1.

```
(model counter
  (sig dataIn  4)
  (sig dataOut 4)
  (sig const_1 1)

  (fnode f_enable
    (input )
    (output const_1)
    (assign (:= const_1 1))
  )

  (fnode f_incr
    (input dataOut)
    (output dataIn)
    (assign
     (:= dataIn (+ dataOut 1))
    )

  (reg cnt 4
    (we const_1)
    (d dataIn)
    (q dataOut)
  )
)
```

Figure 1. An example of a VAM model describing 4-bit counter.

An input source is parsed and an abstract syntax tree (AST) is built. An AST represents a structure of a described model and is language independent. So although VAM is used as a default input language, support for new languages is easy to add. New parser must only produce an AST with the expected structure. An internal simulation model is then created by AST traversal.

3.2 Internal Simulation Model

On the top level, an input model and an internal simulation model are isomorphic. Each input model object (register, signal, ...) has its representation in the simulation model and all connections are preserved, too. We classify model objects into two groups. Each model object is either a state or a stateless element. State elements can store a value, while stateless elements are able to fire (to compute a new value and produce it to node's outputs). Registers and memories are state elements. Also signals are considered state elements, because one can read or write their value. Therefore, they could also be referred as variables within the model. Functional nodes are stateless[1]. An example of an internal graph representation of the previously described VAM model is shown in Figure 2.

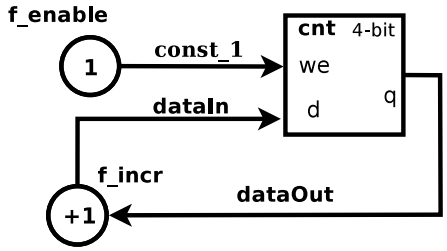


Figure 2. An internal graph model of the 4-bit counter described in Figure 1. Circles represents functional nodes, squares are storages and edges are signals.

3.3 Schedule of the Model

Since data–flow graph paradigm allows nodes to fire whenever it has sufficient data on its input edges, a schedule (described in Section 2) has to be introduced. The schedule should control an order of nodes' execution, thus it can only contain stateless elements which are able to fire and produce new value.

To create a schedule, each storage node is decomposed into separated stateless elements. These elements are called *input and output interfaces*. When an input interface fires, a value of its parent storage can be updated. When an output interface fires, a value of its parent storage is propagated to all its output signals. An input/output interface of a register is a set of all its input/output signals, respectively. An input interface of a memory unit is represented by a write port, while read port represents an output interface. Such decomposition is possible, because all storages are state elements and they have non-zero write delay. It means that their input and output is not directly connected. An example of the decomposed model from Figure 2 is shown in Figure 3.

After storages are decomposed, we consider stateless elements and signals only. By doing that, we get a new graph where any path going through a storage is disconnected. Such graph can be passed to a modified version of a topological sorting algorithm [10]. The output of the algorithm is a sequence, in which nodes

[1] Zero-delayed cycles are not allowed in the graph.

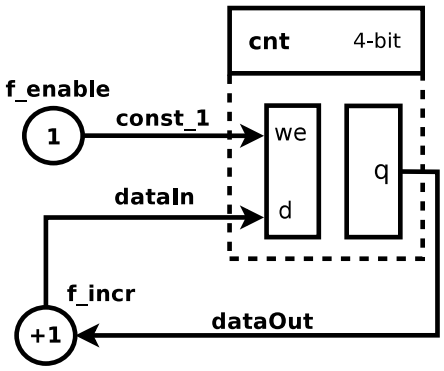


Figure 3. Decomposition of the model from Figure 2.

should fire. The algorithm takes each stateless element with no input signals and puts it into the resulting sequence. Then it removes the element and all its output signals from the graph[2]. This routine is repeated until the graph is empty. As in Section 2, we call the resulting sequence a schedule.

The schedule guarantees that each node will fire if and only if all of its predecessors have already fired. This is a significant observation. When a node fires, it does not need to traverse the graph to obtain a proper value of its input, because the value has already been calculated. For example, in Figure 3, the input interface of `cnt` register fires only after functional nodes `f_enable` and `f_incr` have fired. Node `f_incr` fires after the output interface of `cnt` has fired. Both register's output interface and functional nodes with no input can fire anytime, because they have no further dependencies.

As mentioned in Section 2, an execution of the model is driven by presence data, not by time. Moreover, there is no entity to measure time. Therefore, we define a time unit to be one iteration over a model's schedule. It means that each node fires at most once during a single time unit. after the last element of the schedule has fired, next time moment begins.

4. Simulation Process

The simulation process consists of two phases – initialization and execution. Both of them works with several structures which are either necessary for the simulation itself or for optimization purposes. Besides model and schedule, simulator uses *a next–event calendar* and *model state logger*. A calendar is used to schedule events to a specific time and perform these events, when given time is reached. A model state logger logs a state of a model as it changes in time.

[2]We extended the algorithm to support signals which can connect multiple nodes.

4.1 Simulation Initialization

When an internal model and its schedule is built, an initialization of simulation process begins.

If the constructed schedule of the internal model contains stateless elements with no input signals (there has to be at least one such element, otherwise the graph can not be topologically sorted), they are removed from the schedule. If the removed element is a functional node, it means that its output value can never change (data-flow paradigm does not allow any side effects). If the element is a storage output interface, its value can only change when the value of the storage changes. If that happens, output interface firing event is scheduled to the calendar. Therefore, their firing is scheduled to the calendar only once, at the first time moment of the simulation.

In the next step of initialization, initial values of all storages are set. Default initial value for all storages (whether it is aregister or a memory cell) is zero. The default value of each storage can be overridden (explicitly for each storage) by applying a special file containing new initial values. The file can be seen as a sequential program where each line defines new initial value of a single storage in form of a triplet `type, name, value`. For example line `reg cnt 10` sets initial value of register `cnt` to `10`.

4.2 Simulation Execution

After the initialization, the simulation itself can begin. Algorithm 1 shows a simplified version of the used simulation algorithm. Each simulation time, all events scheduled in the calendar for the current time are performed. Such event can, for example, update a value of a storage or explicitly force firing of a node. Then each node of the schedule is taken and processed. Each type of node has different *firing routine* and *firing condition*. A firing routine is an operation which is performed when the node fires. A node fires if and only if its firing condition is valid. For example, functional node fires only if a value of any of its input signals has changed. When all nodes from the scheduled have been processed, simulation time is incremented.

A support for *conditional execution* is currently in development. It will allow simulator to run simulation process until a certain condition or an event occurs. It is going to be implemented in form of `run until <condition>` command, which a user can enter to execute the simulation. This feature will significantly increase simulator's ability of detecting flaws in models.

Input: Model, Schedule, Calendar
Let t is current simulation time;
while $t < STOP\_TIME$ **do**
  for *each event scheduled at time t* **do**
    Remove event from calendar;
    Perform the event's action;
  for *each node n in schedule* **do**
    if *n is a functional node* **then**
      if *value of any input changed* **then**
        evaluate function of n;
        propagate value to outputs;
    else if *n is an input interface* **then**
      if *value of any input changed* **then**
        perform next-state function;
        schedule *update-storage-value event* to calendar;
    else if *n is an output interface* **then**
      if *value of storage changed* **then**
        propagate value to outputs;
  $t := t + 1$

Algorithm 1: A simplified version of simulation algorithm.

5. Output Interfaces

We decided not to create a single output interface with too many features which becomes hard to use. Instead, we created a *simulation framework* where new output interfaces can be created and easily connected to a single simulation core. With this approach, a user can create new output interface, which precisely meets his requirements for current task. It is not necessary to compromise among other interfaces. The approach guarantees the same simulation results among all the output interfaces. So far, we created three standalone output interfaces.

5.1 Textual Output Interface

This output interface provides direct results of the simulation in textual form in VCD format so the the results can be used be analyzed by other tools.

The reason for this output interface is to provide a user with a simple command line interface. It is useful when a user does not explicitly need to control the simulation or perform interactive operations with the model.

5.2 Interactive Textual Output Interface

Similarly to the previously mentioned interface, this is an interface with purely textual output, however it provides a user with a console, through which a user can enter commands and retrieve their results.

We created a set of commands to control both simulation process and model's state. An example of a simulation control command is `run` which forces the simulation core to perform specified number of steps. For managing model's state there are, for example, commands to force storage's value or to retrieve its state from the past. Demonstration of the interactivity is shown in Figure 4.

```
>>> init counter.init /* initialize from file */
>>> step /* perform single simulation step */
>>> dumpreg cnt bin /* Get binary value of register cnt */
0b0000
>>> step
>>> dumpreg cnt bin
0b0001
>>> run 5 /* perform five simulation steps */
>>> dumpreg cnt dec
6
>>> setreg cnt 10 /* Set value of register cnt to 10 */
>>> dumpreg cnt dec
10
```

Figure 4. Demonstration of output interface's interactivity. A user can enter commands to observe/control simulation. The string `'>>>'` is a console prompt.

This interface does not provide VCD output by default, but it can be generated by explicit usage of command `export`. The goal is to provide an interface for automated scripts. User can write a sequential program consisting of simulator's commands, pass it as an input of the simulator and then analyze the output.

5.3 Interactive Graphical Output Interface

This interface forms graphical environment which visualizes the simulation process in form of digital waveform graphs. An example of a simple plot is shown in Figure 5.

Its main goal is to help a user to better understand an investigated model or to manually detect flaws in it. The interface has a built-in console which works the same way as it does in the interactive textual interface. The difference is in their purpose. In this case it is not meant to allow a user to run scripts, but to give a user an easy way to observe a state of the model.

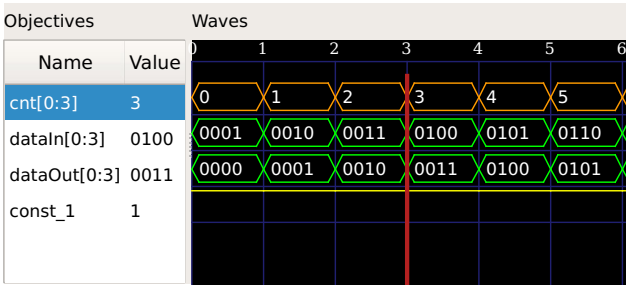


Figure 5. Visualisation of simulation progress with digital waveforms.

6. Practical Application of the Simulator

We conducted series of experiments on several models in order to demonstrate and verify simulator's functionality. Simulation results of the examples were manually checked and found correct, therefore these examples were included into a test suite of the simulator. Besides simple circuits, we performed simulation of a model of an 8-bit processor with 3 pipeline stages, but the manual verification of the results is not finished yet. However, the simulator provides promising results, so far.

We also simulated large models of 32-bit processors with several memory units. However, in this case the simulator was not able to efficiently manage 2^{32} memory cells used by the model. Therefore, a new memory cell management system is planned to be implemented.

7. Conclusions

The paper presented an interactive simulator for dataflow graphs, a support tool for verification environment HADES [2]. The first part of the project was to design representation of an internal simulation model and an efficient simulation algorithm for it. The other part of the work focused on designing proper output interfaces for the simulation to satisfy all requirements.

We created graph-oriented internal representation for loaded models and an efficient simulation algorithm which does not perform unnecessary evaluations and saves computing time. We equiped the simulator with several ways of simulation output presentation. Each has its specific benefits to users, whether it means scriptable command line interface or visualization of the results.

Simulator's purpose is to help a user to better understand a verified microprocessor design and to allow writing automated tests of the design. It leads to decreasing time and increasing quality of the whole verification process. Its prototype implementation has been used to validate counter-examples found by verification tool HADES. The simulator is also used to support testing in project which targets to create a compiler from Verilog/VHDL to VAM.

There are several useful features that are planned to be implemented in the future. As mentioned in Section 4.2, we currently work on implementation of `run until <condition>` command which should increase ability of detecting flaws in designs. We also plan to extend the set of supported commands for new useful features. Several memory optimizations are planned, too.

References

[1] E. A. Lee and D. G. Messerschmitt. Synchronous data flow [online]. http://ptolemy.eecs.berkeley.edu/publications/papers/87/synchdataflow/synchdataflow.pdf. Accessed. 2015-04-02.

[2] Hades hardware verification tool. http://www.fit.vutbr.cz/research/groups/verifit/tools/hades/. Accessed. 2015-03-21.

[3] ModelSim ASIC and FPGA Design. http://www.mentor.com/products/fv/modelsim/. Accessed. 2015-04-01.

[4] Ghdl homepage. http://home.gna.org/ghdl/, 2005. Accessed. 2015-04-01.

[5] The value change dump file. http://support.ema-eda.com/search/eslfiles/default/main/sl_legacy_releaseinfo/staging/sl3/release_info/psd142/vlogref/chap20.html, 1999. Accessed. 2015-04-03.

[6] Icarus verilog. http://iverilog.icarus.com/. Accessed. 2015-04-16.

[7] Gtkwave 3.3 wave analyzer user's guide. http://gtkwave.sourceforge.net/gtkwave.pdf, 2014. Accessed. 2015-04-01.

[8] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[9] Variable-assignment-model (vam) – structure and language [online]. http://www.fit.vutbr.cz/research/groups/verifit/tools/hades/download/language.txt, 2012-01-24. Accessed. 2015-03-21.

[10] M. M. Priya. Topological sorting. http://www.cs.iit.edu/~cs560/fall_2012/Research_Paper_Topological_sorting/Topological%20sorting.pdf. Accessed. 2015-04-03.