CESNET technical report number 34/2007

Design Specification as a Basis to Hardware Simulation

Ondřej Kuzník, Daniel Jakubík

18.12.2007

1 Abstract

While developing a hardware design, especially programmable hardware, it has proven useful to detect the most critical properties of a concept prior to implementing it. If a simulation method is chosen to gain such knowledge it is vital that it be able to create and adapt a suitable model on the fly. In this technical report we describe an approach meeting that criteria. It leads to an easily amendable behavioural simulation model readily applicable to any phase of hardware development. It then allows us to obtain fair estimates of many properties of the design in progress. This way enables to point out a shortfall very soon while the cost of reviewing the implementation or even the specification is still low. As the implementation becomes available, the model can then easily be adjusted and later even reused if the design becomes part of a more complex structure. The methodology is demonstrated on a recent simulation of a high-speed network design.

Keywords:simulation of hardware, FPGA, NIFIC, SIMLIB

2 Introduction

When designing any software or hardware product, a developer has to devise an algorithm that fits its application. That means it is required to be free of bugs, be able to cope with any reasonable input, comply with required standards and meet both time and memory constraints implied. Many approaches have been used to assure that all these criteria are met.

Besides verification and testing which are commonly used in many projects to check correctness in the process of designing and developing hardware, it has proven useful to have some insight in its critical properties of a concept prior to implementing it. The necessary information can be acquired by simulation which generally means creation of a simplified model of the said system so that experiments can commence. Creating a model from a specification that has

not yet been implemented is a task that has to be done by hand. Since every evolution of the design means changing the model too an approach that allows for easy modification or recreation of the model will eventually pay out.

In this report an approach fulfilling these demands is presented on an example of a high speed network design. It has been developed under Liberouter project, mostly for models based on simulation library for C++ SIMLIB[1] but should be applicable when creating any behavioural simulation model.

3 Modelling hardware designs

Very often the hardware design is well structured mostly with clear hierarchy between the components and a clear flow of processed data taking advantage of pipelining and parallelism where possible. The topmost levels of this partitioning are already decided and stabilised in the earliest phases of the design and thus a more or less precise model can be built according to the specification when no or only a little part of the implementation is ready.

If such a model is made, as the work on implementing the design continues, more details are known about each of its parts that can be included in the model to find out more about the expected behaviour. This assumes that the model be set up of a similar set of components as the design itself and that the components can either be specified further at any time or replaced by a more precise representation with little effort spent on subsequent changes to the model so that it can accommodate the new version. These requirements are perfectly satisfied when we look at the concept of object oriented programming. If provided with a modelling environment capable of nesting components inside components and supporting parallelism of a type similar to the one found in hardware, we should be able to create and maintain such a model.

3.1 SIMLIB

The name SIMLIB/C++ is an abbreviation of “SIMulation LIBrary for C++” and has been developed by Petr Peringer at the Faculty of Information Technology in Brno since 1991. It is a general purpose simulation environment containing basic tools for continuous, discrete, mixed and fuzzy models. Because the models are created directly in C++ language, it lets us use all of its features and other advantages connected with object oriented programming.

The components can be described and put into a hierarchy based on a relation of abstraction/specification. One of the aspects of this approach being that it brings the ability for a concurrency, an important aspect for a hardware simulation.

[1] *http://www.fit.vutbr.cz/ peringer/SIMLIB/*

The code defining a SIMLIB simulation can be divided in two parts, definition of the components and the experiment set up (the main function of the program). The set up part takes care of creating and initialising the model and activating the simulation and the components describe its structure and interactions that affect the resulting behaviour.

The library components that can be used in discrete simulations can be divided into active and passive ones. The active are limited to the Event and Process class derivatives. The variety of passive components is much wider, the most prominent being a Facility, Store and Barrier classes.

The behavior of active components should be strictly parallel, similar to the behavior of some hardware components, and the SIMLIB seems to process them in such way. In fact, the SIMLIB core executed them in quasi parallel fashion, which means that the central scheduler switches among them when they reach certain breakpoints, effectively forming a cooperative multitasking system.

Each of the passive components gathers statistical data throughout the experiment that can be written to a file after finishing its termination. They are presented in a table, so they are easy to read but harder to interpret and process further without sophisticated preprocessing.

4 NIFIC design simulation model

NIFIC[2] stands for a "Network interface card capable of packet classification, filtering and forwarding". It has been developed as a part of the Liberouter[3] project. To ease future development, the team has decided to reimplement this device based on a unified design platform NetCOPE [MT06] using components that have been substantially improved since the design was first built. At the time it was being finished, there were several aspects of the design in need of clearing out. Those aspects were the information on how does the maximum available memory at certain points affect the behaviour of the whole system and the amount of memory needed at those points for the design to reliably operate at least at the desired speed.

The interface had not yet been operational so it could not be directly tested in real conditions nor in test cases but enough information was known about the relevant components to issue a creation of an abstract model of the design whose simulation would provide the team with enough information regarding this topic.

[2] *http://www.liberouter.org/nific.php*
[3] *http://www.liberouter.org*

4.1 Description of the NIFIC

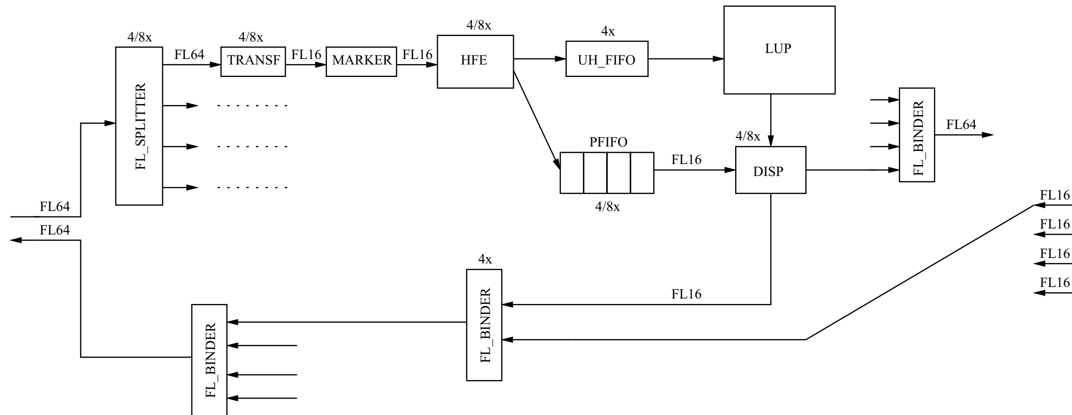

Figure 1: The components of NIFIC and the interconnections between them

The interface is equipped with four network connections. Every packet received on the link gets to the marker through an interconnection interface, whose endpoint is drawn at the top of Figure 1. Marker then checks that packet's CRC and timestamps it, forwarding it to further processing which is the Header Field Extractor (HFE) unit. There are four or eight of these extractors and other components that the HFE connects to. This is why the packet is sent to HFE that has the least amount of unprocessed data. The HFE then parses the packet for information based on what data is relevant for the nanoprogram set up in the Look-Up Processor (LUP). The data is stored in Unified Header format into the UH_FIFO queue. While the packet is being parsed and relevant information stored in UH_FIFO, the raw data is forwarded into Packet FIFO (PFIFO).

If for any reason the HFE cannot write the parsed information or packet contents to the appropriate queue, it suspends processing the packet and stops writing to the second queue until the pending write operation has succeeded. When the HFE has finished parsing the packet, LUP can begin processing the data according to the criteria its nanoprogram has been configured for using the data HFE encoded into the unified header.

The LUP has an exceptional status among the NIFIC components. While same amount of just about every component is four or eight (depending on the settings), so that even partially processed packets can be fed to the next component and don't get mixed up, there is only one LUP instance classifying the incoming packets from every source and then notifying the appropriate dispatcher (DISP) whether the packet should be dropped, sent to the computer's operating system and/or forwarded to the network and through which connections.

There are two message queues between LUP and each of the dispatchers. One leading directly from the LUP where all messages are stored before there is a

free space for it in the second, dispatcher's own, message queue. The message waits in this queue until the dispatcher has finished processing the previous one and is ready. A packet can be flagged for dropping, then its contents are immediately flushed from the PFIFO and nothing else happens. If the packet is meant to be processed further, the dispatcher reads the raw packet data from the PFIFO and copies it to the appropriate output buffers.

4.2 Principles behind the model

Building the NIFIC model, the first thing to do was separating the design from the data (internet packets in this case) that were processed by it. All information related to the packet is thus represented by one structure common to the whole model, the packet itself, and a component handling it should behave like it sees only parts of the packet that are accessible to it. This way almost every communication between two components could be abstract, e.g. the HFE component will not have to generate a Unified Header of a packet (contents of which depend on the settings loaded into the design at boot time) and then simulate its transmission to UH_FIFO, it just sends a structure containing this packet (or in C++ a pointer to that structure) and UH_FIFO blocks the amount of space it would normally take up.

At this moment we should concentrate on deciding the overall scope of the model. There are obviously several different parts that have no or only one connection to others and form a single logical block, in the NIFIC design there are four such blocks, the first one is responsible for reception and integrity checking of a packet and then feed it to the HFE units. Most of this block is not included in the figure save the splitter, transformer and marker components. The second block is the NIFIC core, HFE, FIFOs, the look-up processor and dispatchers which is a natural choice as those components process a packet and control whether it should enter another blocks, the third block is the NetCOPE interface serving as a connection between the core/network on one side and the operating system of the computer this card is plugged in. The last block is the connection between the NIFIC core and the Ethernet network containing binders shown in the figure that blend the flows of packets from dispatchers and operating system into one ultimately leading to the transmitters. In most designs these blocks should look just like a set of components bound together by easily describable connections and thus form a higher abstraction layer.

Because of the strict boundary between the active and passive components in SIMLIB environment, the magnitude of a model created by following only these guidelines would be much too great as each component that is not entirely passive should be represented by several processes/events plus the stores, facilities or whatever is needed to hold its state. As the internal state of each component contributes to the state of a whole model, these parts cannot be

easily eliminated, so the only thing remaining to be simplified is their behaviour. Another separation has been created – a layer put between the two proposed at the beginning of this section (components and the data), representing the workflow inside the design. Operating on both of them it is driving all the behaviour of the model.

4.3 Only relevant components

The purpose of a model we were building was to evaluate the constraints that apply to the core parameters so that the interface can handle any flow of data that can possibly come from the four Ethernet connections. As the only new part was the NIFIC and the rest represents a NetCOPE architecture that is built to comply with these conditions and simulation of which was none of the goals, we can abstract from as much of its parts as we think will not affect the precision of the results. This suits us well because several of the high level components we divided the design into do not seem to affect the core performance.

In this case we decided to leave the component that receives packets from network and delivers them to the input of one of the HFEs completely abstract reflecting only the ideal conditions, namely when there are only correct packets received from the network. Then it only merges the flows and distributes packets to the appropriate HFEs. The discussion over the detail of modelling of the transmitting component had been somewhat longer. If the stream of data produced by the dispatchers and directed by the look-up processor is too fast for even one of the four connections, the common route connecting them to the entry point of the component will start filling and when all the buffers are filled, will slow down every Dispatcher's work. Yet we were trying to determine the behaviour of the device at a stabilised state. If any of the connections is over-utilised, no matter how much space is there in the buffers, it will eventually be depleted. This means we only have to care about modelling the network connections themselves, the route in between can be represented as a simple connection, just like the previous component is. Finally there is the block standing between the core and operating system. Since its inability to process the flow of data will mean the same as the network connection's and the limit (defined by the OS's workload at that time) can change anytime, we have to suppose that there is none.

This way we have decided what is and what is not important at the highest level of abstraction. The only component that is fully relevant is the NIFIC core, where the same process we described earlier in this section took place, leaving some components rather abstract (HFEs that only forward data and FIFOs acting like the simplest store would) and some very detailed (the dispatchers). The last step to creating a model is actual implementation, which is highly dependent on the environment used.

4.4 Gathering the results

Once we have a simulation model we need to run experiments to validate it and gather the information we want. To accomplish these goals an experiment has to produce some data so that we have something to operate on. In our case we needed to measure if the actual throughput meets the projected criteria and watch the utilisation of memory segments. In the case of short time experiments like model validation it is easier to analyse the complete history of a relevant variable. When we are curious about the stable state properties we can either read the state at the moment the experiment is terminated or build our own summary. At this point the SIMLIB has saved us much work because each passive component maintains such a summary.

4.5 Results

At first we used parameters that were most likely to occur in a common set up along with a reasonable reserve when it came to deciding the amount of maximum available memory at the critical points we were to investigate. The resulting characteristics then served as a reference for judging the sensitivity of the model to changing values of each parameter. These parameter sensitivity checks showed that the device does not require as much space in the buffers as initially assumed. In reality, the lower bound on the minimum space is defined directly by the maximum size of a packet when it comes to PFIFO and one pending message when it comes to the FIFOs between LUP and each dispatcher.

Checking the model's sensitivity to the other parameter values revealed yet other interesting properties of the model (and as has already been confirmed, also the modelled interface). One of the findings was that LUP can be a bottleneck beyond certain time it takes to process a packet since it has to serve all traffic received on the interface. The other properties that the model has, are a robust design of the dispatchers (that is what ultimately allows the FIFOs to be as small as possible) and a (mis)configuration possibility such that lowers the maximum throughput of the interface to half the ideal value. This can happen when the LUP decides based on more data, than the HFE units can provide.

Table 1 and Table 2 represent behaviour of the model when all four Ethernet interfaces supply it with packets at full speed. Average time between receiving and ejecting each packet can serve as an indicator of the the interface's load, because from the queuing theory when a load of a system approaches its maximum capacity, the average time spent waiting quickly climbs up towards infinity.

HFEs	PFIFO cap.	LUP data	LUP delay	LUP FIFO	Disp. FIFO	Avg. delay
4	4096 B	128 B	80 ticks	4 mess.	2 mess.	984 ticks
4	2048 B	128 B	80 ticks	1 mess.	1 mess.	984 ticks
4	2048 B	128 B	160 ticks	1 mess.	1 mess.	1109 ticks
4	2048 B	128 B	250 ticks	1 mess.	1 mess.	1775 ticks
4	2048 B	128 B	260 ticks	1 mess.	1 mess.	2692 ticks
4	2048 B	128 B	265 ticks	1 mess.	1 mess.	infinite

Table 1: Response of the model when stressed by Ethernet packets of all sizes.

HFEs	PFIFO cap.	LUP data	LUP delay	LUP FIFO	Disp. FIFO	Avg. delay
8	4096 B	128 B	80 ticks	4 mess.	2 mess.	infinite
8	4096 B	128 B	16 ticks	4 mess.	2 mess.	143 ticks
8	4096 B	256 B	16 ticks	4 mess.	2 mess.	207 ticks
4	4096 B	128 B	16 ticks	4 mess.	2 mess.	146 ticks
4	4096 B	256 B	16 ticks	4 mess.	2 mess.	infinite

Table 2: Response of the model when stressed by a continuous stream of shortest packets.

5 Simulation model of bus system in NetCOPE

The other project within the Liberouter project is engaged in development of the NetCOPE platform. It forms the basic infrastructure inside the FPGA chip and defines an interface for communication among modules. Using this platform can make the applications built in top of that faster in future.

In addition, the platform should be able to process the data in hardware (and eventually transmit them to the operating system) significantly more quickly then present applications. To achieve this goal it is necessary to find out the maximal throughput from and to the adapter.

The simulation model of this bus architecture was created just for the verification of the critical properties such as speed of packet processing and packet transmission or the already mentioned throughput.

5.1 Architecture description of NetCOPE buses

The bus architecture of the NetCOPE platform consists of three different types of buses.

Internal bus: This bus has been designed to provide high throughput for the components connected to the host PCI interface. Every component on this bus can operate both in Master mode and Slave mode.

Local bus: It interfaces most of the common components that do not need high bandwidth. The components connected to this bus can only work in Slave mode. That means that they can have only passive behavior and they do not initiate reading or writing transaction by themselves.

Control bus: As the name suggests, this bus is reserved for control data transfered between the generic Bus Master controller in the form of the PowerPC and Bus Master component that sends data to the RAM or vice versa.

More information about this architecture is available in [MT06].

5.2 Building the model

While modelling an efficient system, it is generally possible to abstract from the real implementation and focus only on particular actions of the system, their order (including parallel or exclusive processing), and duration of particular actions. A time event or a previous action finishing are then the typical initiators of an action.

At the beginning of the modelling process, it is necessary to decide the structure of the constructed model.

The first thing was to separate the data from the rest of the functional model as has been described above (Section 3.2). For that purpose, we have adapted the class Packet, which was developed within the modelling of the NIFIC, and especially, designed for reusing in other models. In our case each instance of this class represents one packet processed by the system. The packets are processed by components of the system.

As seen by incoming packets the system's behavior is divided in two parts. The first (TX part) represents the relay from RAM memory to the FPGA and the second (RX part) represents the relay from the FPGA to RAM. Both data streams are nearly independent and meet only in few components, making it possible to split the model according to these streams. Modelling the system from the view of the data stream corresponds to the base idea of SIMLIB, where active elements are the processes and devices behave passively. The system can be split more finely according to the type of the bus on which the data are being exchanged. Then the main classes can be deduce from that.

In the subsequent step, two mutually independent generators were created. These produce packets received in the RX or TX direction. These represent the results of processes which take place behind the boundaries of the modelled system. The buses were easy to model as a mere delay. It was just necessary to ensure that the maximum bandwidth of the transport channel cannot be exceeded and that they will arrive in the same order they were sent. Because

the Programmable DMA Controller (PDMA) was not the center of our interest, we could accept another simplification, when we supposed that PowerPC processor can cope with all incoming requests and all memories in PDMA have sufficient capacity. Thanks to this abstraction PDMA could be modelled as an event, which loads requests from the appropriate queues in a periodic intervals. Not modelling the physical data paths allows us to abstract from many components like endpoints (interfacing component and buses).

5.3 Results

While running the simulation two essential properties were observed. The first one was the utilization of every component's capacity, the second one was the delay between a packet entered the system and when it has been successfully transmitted outside the system. Unfortunately, the obtained results were not as favourable as we had hoped for.

The data gained so far show, that throughput of the current architecture is about 250–350 Mb/s, while the minimum throughput should not be below 2 Gb/s, as modelled. The problem is caused by inadequate speed of PowerPC, which is not able to serve all requests and serves as a bottleneck of the whole system.

The simulation model can be run with different parameters. So the next step was to find their optimal settings. It was shown that the performance of the model does not depend on the capacity of the queues and of the buffers.

The simulation had another use. It was to find the appropriate settings of the system components so that its throughput was the best possible. Different changes to the design and its settings were proposed. One possible solution to this problem could be the separation of processing the TX and RX directions as shown by a modified model.

6 Conclusion

In this report we described a way of abstraction, which quite easily allows representing all important features of the design while the less important can be simplified so otherwise very complicated and vast systems can be represented in a neat and comprehensible model. This way simulation models can be created and maintained quickly and used at any stage of the design process.

The approach was demonstrated on a simulation of high-speed network design and a bus subsystem and can also be used when creating a behavioural model of a hardware design or other well structured system. Moreover, the simulation framework established during this simulation process (i.e. NetCOPE component models, other generic components or extensions to the SIMLIB environment)

can be reused in future simulation of new versions of this component and in simulation of other similar designs.

The results of each model clearly pointed out the most prominent limitations of both systems. We either defined the constraints under which the system fits the intended purpose (in NIFIC), or proposed a change to the design that can help achieving the desired functionality (for the internal bus). The model implementations are stored in the project Subversion repository. Since this repository is not publicly readable, both models are also be available from author's home page[4].

In comparison to testing or verification, which are applicable mainly in advanced phases of development, the simulation, with respect to the presented procedures stated in this report, brings designers useful tool which can help them already when drafting a system and subsequently in any state of its development.

References

[MT06] Martínek T., Tobola, J.: *Interconnection System for the NetCOPE Platform*. Technical report 34/2006[5], Praha: CESNET, 2006.

[4] *http://www.liberouter.org/ kuznik/models/*
[5] *http://www.cesnet.cz/doc/techzpravy/2006/netcope-interconnection/*