High level synthesis methodology from C to FPGA used for a network protocol communication.

M. Diaby, M. Tuna, J-L. Desbarbieux, F. Wajsburt
University Pierre et Marie Curie
LIP6 Laboratory, Dept ASIM
4, Place Jussieu, 75252 Paris Cedex 05, France
{mouhamadou.diaby, matthieu.tuna, jean-lou.desbarbieux, franck.wajsburt}@lip6.fr

Abstract

This paper presents a "Kahn Process Network" methodology based on the DISYDENT platform (DIgital SYstem Design ENvironmenT). The system is described by a set of communicating Kahn processes. This processes are C POSIX threads representing both software and hardware tasks. Each thread communicates with the others using channel-read / channel-write primitives. Thus, the system can be validated efficiently and quickly by software. System's realization consists of synthesizing hardware tasks to RTL-VHDL language. This step is automated from C task to FPGA mapping. This paper shows the method's effectiveness through the realization of a network controller on FPGA enabling communication between two Linux stations.

Keyword

High Level Synthesis, KPN, Hardware/Software codesign, FPGA, Code Generator, VHDL

1. Introduction

The complexity of today's application specific integrated circuits steadily increases. Their design and implementation is getting more and more complex [6]. This is a problem for integrated circuits for which high quality is required. Moreover, "time-to-market" constraints are more and more aggressive [4].

Though hardware description languages (HDLs), such as VHDL and Verilog, have became considerably popular in system design, there is an increasing need for more intuitive methods for system specification. Moreover, high level system design requires specification method of higher level of abstraction than HDLs which are based on hardware model. Whenever there is a higher level specification, it needs a new step: once a satisfactory system model is established, it has to be implemented. This operation is quite critical [7].The model may be misunderstood or some implementation choices may change the system behavior.

The proposed methodology starts by the description based on the Kahn processes network formal representation [5]. The set of communicating processes are C POSIX threads representing both software and hardware tasks. This means threads communications using channels read / channels write primitives. The hardware implementation is automated by a code-generation tool, needing a C source and generating the physical description of the circuit in RTL-VHDL language.

The paper is composed of 2 main sections, organized as follows. Section 2 presents the methodology and the work environment : Dysident. In section 3 the methodology is applied in real case: the conception of a protocol test platform. The last section presents the results and a discussion.

2. Methodology

This methodology helps to go from the high level description to the actual prototype implementation. It's based on the DYSIDENT environment and most specifically on the use of DPN (Dysident Process Network) library and the UGH tool (User Guided High level synthesis). Figure 1 represents the design flow of our method. Each step is detailed below. First, we study the DPN model, then follows the implementation stage, lingering over UGH, and finally to obtain the prototype.

2.1. Software DPN model

DPN stands for "Dysident Process Network", DPN [1] is a library that provides communication channels. It allows

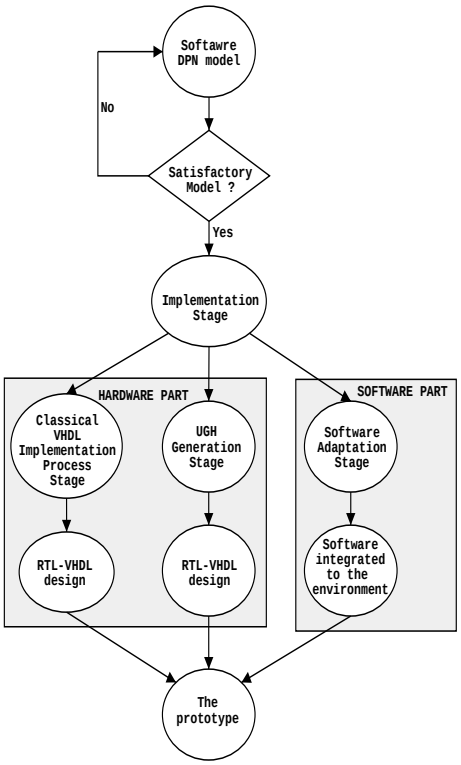

Figure 1. The method design flow

to specify and simulate a parallel application described by a communicating tasks graph applying the Kahn processes model. Thus, DPN library is on one hand a theoretical model for the parallelism and the communications between tasks, and on the other hand an implementation that uses C POSIX thread for simulating the tasks parallel execution. The C Posix thread model hardware tasks as well as software tasks. Thus, the described system represents the whole application.

Figure 2 presents an example: we have two processes (hcf and work) and two FIFOs (hcf2work, work2hcf).

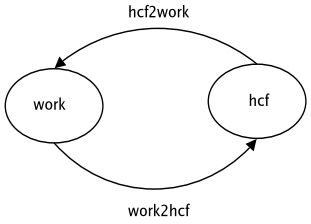

Figure 2. The DPN model of the example

This is the corresponding DPN 'C' model:

```
pthread_t hcf_tid,work_tid;
  Channel* hcf2work,work2hcf;
  /* creation of the channel */
  work2hcf= channelInit(DEPTH, 4);
  hcf2work= channelInit(DEPTH, 4);
  /* creation of the hcf and work process */
  pthread_create(&hcf_tid, 0, hcf, 0);
  pthread_create(&work_tid, 0, work, 0);
  /* waiting for the end of work process */
  pthread_join(work_tid, NULL);
/*
The hcf process repeats forever:
1) read 2 words from the work2hcf channel,
2) compute their hcf,
3) write the result into the hcf2work channel.
*/
  void hcf()
  {
      int a, b;
      while(1)
      {
          channelRead(work2hcf, a);
          channelRead(work2hcf, b);
          while (a != b)
          {
              if (a < b)
                  b = b - a;
              else
                  a = a - b;
          }
          channelWrite(hcf2work, &a);
      }
  }
/*
*/
  void work()
  {
      int in,out;
      in = production_number();
      channelWrite(work2hcf,in,1);
      in = production_number();
      channelWrite(work2hcf,in,1);
      channelRead(hcf2work,out,1);
      printf("out = %d", out);
  }
```

Simulation validate the description.

2.2. Implementation stage

Following the design flow chart, the next step is a critical section : the implementation. It can be divided in two main parts : hardware part and software part. The hardware part's implementation consists to obtain the RTL-VHDL code. This can be made in by an automatic code generator : UGH, or by a "manual" classical way. The software implementation consists to adapt the code at the environment in which it will be submerged.

2.2.1 Classical VHDL implementation process

The "manual" classical way : following the classical VHDL implementation process i.e. to hand write the RTL-VHDL code and simulate. It is also possible to do design reuse if the bloc already exists in VHDL.

2.2.2 UGH

The UGH tool [3] is a complete framework for high level synthesis and is the cornerstone of our method. In our case,

high level synthesis means translation of a sequential behavioral description (C language) to hardware description (VHDL).

In model UGH, the task communicates with the others tasks using a basic asynchronous handshake protocol (typically FIFO). As indicated by its name (User Guided), the UGH synthesis approach relies on user hints. Such hints are necessary: the module interface (the number of communication channels) and the resource allocations by defining a data path. This clearly means that the designer needs to have designing skills to take the correct decisions. This information is contained in the ddp (Draft DataPath) file, for helping UGH to drive the generation of the data path of the circuit. However, the less you help UGH, the more it takes decision instead of you. To resume, UGH needs two input files: the 'C' source file and the 'DDP' file.

The 'C' source file

The 'C' input of UGH describes the functional behavior of the circuit. In order to synthesize it, UGH places restrictions on the 'C' and so defines a 'C' subset. Nevertheless, the 'C' description is still perfectly valid and may be compiled. For running it, you must provide to the linker the I/O routines. Those for the Disydent Process Network are already provided. Such runs are useful to verify functionally the 'C' description. To perform the synthesis, the description must contain the bit sizes of the registers. This means that the basic UGH 'C' types are:

- int1 , int2 , ... , int32 : the intN type defines an integer of N type bits. These types extend 'C' types char , short , int , long.
- uintx for unsigned types.

In the rest of the description, it is advised to use them instead of standard 'C' types, in order to allow the synthesis process to restrict the number of bits to its minimum. For the compound types, you can define 'C' typedef, structure and union, using the types intN , uintN . Arrays are allowed, pointers are forbidden. The communication channels must be declared as global variables of types ugh_inChannel1, ugh_inChannel2, ... , ugh_inChannel32 for the input channels and ugh_outChannel1, ugh_outChannel2, ... , ugh_outChannel32 for the output channels. The global variables are the registers of the tasks. The types of the variables are either the basic UGH ones (int1, ... , uint1, ...) or compound ones built on these types. Array variables are allowed (RAMs), pointer variables are forbidden. Simple and array variable can be initialized (VarType var=...;) if and only if they are declared "static const". They are then read only. The behavior of the task is enclosed in the ugh_main(void) function. This function is the entry point used for synthesis. The behavior is described in standard 'C' language without pointers:

- variable declarations
- control statements: if, for, while, break, continue, goto, ...
- logic operators: $||$, $\&\&$, ...
- operators on the basic UGH types: all the 'C' operators like $+, -, *, /, \ >>, !=, \ <=,$ ...
- function calls: the functions must be inlined, the classical parameters are allowed even references arguments

Nevertheless there are the following restrictions:

- floats and doubles are forbidden
- pointer variables are forbidden

In our method, the 'C' UGH input is the DPN file after modifications to be UGH compliant. For instance, a UGH'C' synthesizable description of our HCF task is shown below:

```
ugh_inchannel32  fifo_in;
ugh_outchannel32 fifo_out;

HCF()
{
   uint32 a;
   uint32 b;

   while(1)
   {
       ugh_read(fifo_in, &a);
       ugh_read(fifo_in, &b);

       while (a != b)
       {
           if (a < b)
               b = b - a;
           else
               a = a - b;
       }

       ugh_write(fifo_out, &a);
    }
}
```

A comparison with the DPN model shows minor differences between the two files.

The 'DDP' file

The 'DDP' file gives the draft of the data path. Like a data path, it consists of an interface and of a set of cells connected together. Nevertheless, it is a draft. So the following rules allow to simplify the description:

- The interface is reduced to the data connectors necessary to perform I/O operation with the external world.
- The connectors of the cells of the library have no size. Therefore you can connect the connectors of the cells without taking care of their bit sizes.
- Control inputs (clock, write enable, ...) of the cells are omitted.

- An input connector can be driven by several output connectors so multiplexers are not required.

Figure 3 represents the dreamed data path of our hcf:

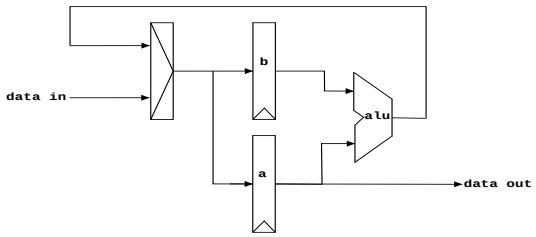

Figure 3. The dreamed data path

Then, here is the code of the hcf.ddp file:

```
MODEL HCF  (inchannel fifo_in;
            outchannel fifo_out)
{

    DFF    a;
    DFF    b;

    ALU alu1;

    a.d = alu1.s, fifo_in;
    b.d =  alu1.s, fifo_in;

    alu1.a = a.q;
    alu1.b = b.q;

    fifo_out = a.q
}
```

The separation of the behavior (C file) and the draft data path (DDP file) permits to test several differents architectures to fit the best to the needs. In other words, UGH permits architectural exploration. As said before, the less you help UGH, the more it take decision instead of you. That means the DDP file can be almost empty, only the interface and the registers are mandatory.

2.2.3 the software part

The software adaptation has two sides:

- the communication with the hardware : for example, if the hardware is mapped on the FPGA, the communication with the FPGA is needed (PCI bus, JTAG or specific). If the software is embedded, the communication is hardware to software fifos or software to hardware fifos.
- the immersion in the environment (Linux, windows, embedded system ...)

2.3 The prototype

The prototype is realized when all parts are connected in the correct environment. If the partitionning in the hardware part between the task generated by UGH and the others is judicious, this means that the task generated by UGH is task liable to change, then if a change appears in this file, obtain the new reliable prototype is quick and easy.

3. Case study: realization of a network controller on FPGA enabling communication between two Linux stations

The aim is to realize a full-size protocol test platform. This platform has to enable the experimental evaluation of ZCSP's protocol (Zero Copy Secured Protocol) [2]. We will experiment the method described above on this project to get rapidly a prototype of the platform. This platform consists of two communicating Linux stations. They are connected by a point-to-point full duplex link plugged to their PCI card. The PCI card is a PLDA PCI20K-PROD C and disposes of an APEX20K FPGA (from EP20K400FC672-1XV family). This FPGA has 16640 logic elements and 212992 memory bits (400k typical gates).

3.1 System description and DPN model

The first step of the method is to code the DPN model of the system. Figure 4 represents this system.

This system represents two communicating machines. Each machine has a network controller (the hardware part with the FPGA). It consists of PCI core, TX, RX and WD tasks resuming the protocol and PB and PX tasks charged of the communication to the other machine. All these tasks communicate by mean of a daisy-chain. This part will be mapped on the FPGA. The Ping-Pong task represents the application using the network controller. Ping-Pong and Memory tasks model the PC environment necessary.

In order to understand the later implementation choices, this section roughly describes each tasks of the system:

- **the daisy-chain**: this subset is the bus of the system. In fact, the APEX EP20K FPGA doesn't have tri-state components and so, we can't realize a standard bus. This daisy-chain is based on multiplexers. The processes of the system will communicate with each others using the daisy-chain. Each operator is connected to the bus through a 'Wrapper'. The wrapper is responsible for the communication between its task and the bus. Figure 5 shows the implementation of the daisy-chain. The information transits on the bus through the pipe (just a register). At each cycle, the information propagates from pipe to pipe.
- **the PCI core**: this entity symbolizing the PCI core will enable communication between the FPGA and the external world. This PCI core enables the classical access to the configuration space and memory space.

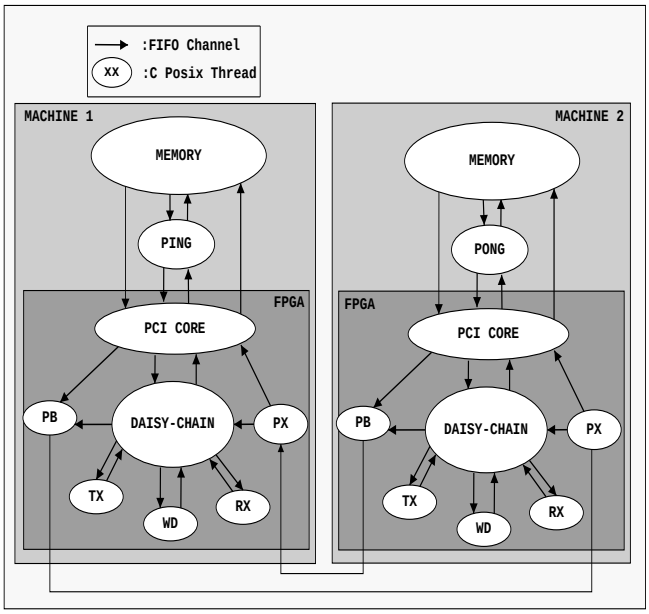

Figure 4. The system: two machines communicating with each other. Each machine is composed of two parts: the software representing by PING/PONG threads and the hardware on the FPGA.

- **Memory**: this task simulates the PC's memory.
- **PB, PX**: these two tasks respectively build packets to send to the network (Packet Builder) and extract information from packets from network (Packet Extractor). PB task of machine 1 is connected to PX task of machine 2 and vice-versa to enable communication between them.
- **TX, RX et WD**: these three tasks describe ZCSP protocol. TX is the transmission task, RX the reception task and WD manage timeouts.
- **PING et PONG**: This tasks represent the software of the system. They realizes a ping-pong between the two machines.

The whole tasks have been described with DPN library. All this tasks simulate hardware level and the necessary software application. This level of description permits a rapid compilation, rapid system execution (C compiled) and simple debugging.

Note: The PCI core physical description in RTL-VHDL already exists. The PCI core described at this level is a DPN simulable simple version, i.e. it communicates with memory and ping-pong with channelRead/channelWrite. In the DPN model, we don't need to add the PCI protocol to the PCI core. We only want to access the memory.

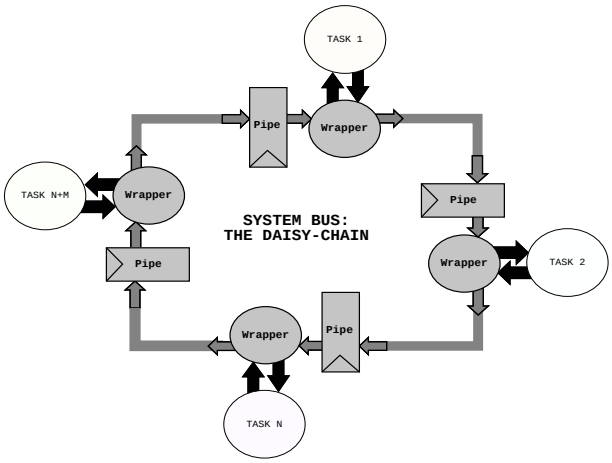

Figure 5. Daisy-Chain model

3.2 Implementation stage

After the validation of the DPN model, we can then begin the implementation stage. This step is the most decisive in rapid prototyping.

3.2.1 Hardware synthesis

For the tasks representing the hardware, the DPN model and the RTL-VHDL description must have the same behavior. In our model, we have to find a partitionning between tasks to synthesize with UGH and tasks to "hand-write". In our case, the VHDL description of the PCI core already exists. Moreover, the PCI core as well as PB, PX and the daisy chain tasks are cycle accurate. Thus, two categories can be defined:

- the cycle accurate tasks: Pb, Px, Daisy-chain, PCI core
- ZCSP tasks: TX, RX and WD

The cycle accurate tasks

PCI core is provided. For the daisy-chain, wrappers are simple combinational components and pipes are just registers. So, the simple design of the daisy-chain enables an easy implementation in RTL-VHDL. PB and PX tasks manipulate a lot of data, it's why their designs must be optimized. Nevertheless, their designs are simple because the manipulated data have been pre-processed by the tasks containing the mindness of the system. Each state machine has less than 10 states.

ZCSP tasks

TX, RX and WD represent ZCSP protocol and so the network controller conductor. Any protocol is first described by a flow chart. The translation from a flow chart to a high level language (C language) is easy. In opposite, translation to a hardware description language (VHDL) is quite hard and long because of the considerable number of states (near hundred). They don't manipulate a lot of data but their behavior is complex. So, we get their description in hardware language with UGH. DPN files must be adapted, replacing channelwrite/channelread by ugh_write/ugh_read and determining the variables size (intN, uintN) and add the entry point: ugh_main. All ddp files are nearly empty in order to let UGH do its own choices.

Software integration

The software integration corresponds to the immersion of the ping-pong thread in the environment. In the DPN model, ping-pong threads communicated directly with the PCI core. The ping-pong thread communicates with the PCI card through an added software brick: the PCI core Linux driver. Obviously, the communication with the memory changes compared to the classical way.

3.3 Mapping and results

When the whole tasks is synthesized in RTL-VHDL, this description can be mapped on the FPGA. The tools used is ALTERA QUARTUS II version 2.0. After the mapping of the hardware tasks on the FPGA and the installation of the software application (driver and ping-pong programs) on the two machines, the platform is ready to work. The results with our network controller:

- full PCI throughput (no wait state) : more than 100 MB/s
- circuit frequency : 33 MHz
- fpga occupation : 80%

The table below presents for each ZCSP tasks the number of C lines written, the caracteristics of code generated by UGH (number of line and state of the state machine).

	TX	RX	WD
C lines nbr	390	375	232
VHDL lines nbr	3052	1088	955
states nbr	440	143	34
logic cells nbr	3363	2488	954
memory bits nbr	0	0	4592

TX represents 20% of the FPGA surface, RX 25% and WD 7%. They represent more than 50% of the global circuit. TX, RX and WD are the core of the hardware part.

4 Discussion

As we have seen, the three ZCSP tasks represent more than 50% of the circuit because of their complexity and the code generated by UGH is not optimal. Nevertheless ZCSP protocol is still in development. So, the ZCSP tasks (particularly the transmission and the reception task) will be modified. The new prototype is reliable and quickly obtained. In this modification loop, the longest phase is using QUARTUS to map on the FPGA (about 45 minutes). In our case, the interest to have a C model for the ZCSP tasks permits to translate easily to Promela language for model checking.

5 Conclusion

Implementation phase is a critical step. This paper presents a methodology in which an application written in language C is mapped onto a target platform composed of a CPU and an FPGA in an automatic way. UGH is the heart of our methodology. This tool allows us to quickly go from an application specification in C to an hardware specification in VHDL. The system is described with DPN model which applies Kahn processes model. In this model, memory is distributed and each task runs independently of the other tasks. So the partionning is realized easily and so we can get an efficient mapping onto the CPU and the FPGA. The paper shows the method's effectiveness through the realization of network controller on FPGA enabling communication between two Linux stations

References

[1] I. Auge, F. Donnet, P. Gomez, D. Hommais, and F. Petrot. Disydent : A pragmatic approach to the design of embedded systems. *DATE02 Designers' Forum, University Booth Demonstration*, page 255, March 2002.

[2] V. Beaudenon, E. Encrenaz, and J.-L. Desbarbieux. Design validation of zcsp with spin. *Third International Conference on Application of Concurrency to System Design (ACSD'03)*, page 102, June 2003.

[3] F. Donnet. User guided high level synthesis. *PhD Thesis of University Paris 6*, January 2004.

[4] M. Hunt and J. Rowson. Blocking in a system on a chip. *IEEE Spectrum*, November 1996.

[5] G. Kahn. The semantics of a simple language for parallel programming. *In Information Processing: Proceedings of the IFIP Congress*, page 74, 1974.

[6] N. Leveson. Software engineering: Stretching the limits of complexity. *Communcations of the ACM*, 40(2):129–131, February 1997.

[7] S. Murphy, P. Gunningberg, and J. Kelly. Implementing protocols with multiple specifications : Experiences with estelle, lotos and sdl. *Testing and Verification*, June 1989.