Master Thesis: Development of a Runtime Reconfigurable Application

Ronald Hoogma[1]

September 28, 2004

[1]Thesis advisor: drs. S.A. Achterop

Abstract

Hardware development is an interesting research area. At first only ASICs (application-specific integrated circuits), ready-made circuits, were used. As technology advanced, FPGAs (Field Programmable Gate Arrays) were developed. FPGAs allow to be reconfigured on the fly. This thesis covers the development of a Runtime Reconfigurable Application. It discusses a global model of an RTR-application along with its associated problems. As part of the thesis an RTR-application doing CRC-checks is developed.

Contents

1 Introduction **7**
- 1.1 Introduction . . . 7
- 1.2 FPGAs explained . . . 9

2 Related work **15**
- 2.1 Introduction . . . 15
- 2.2 A hardware genetic algorithm for the Travelling Salesman . . . 15
- 2.3 Multitasking in hardware-software codesign . . . 19
- 2.4 Dynamic hardware plug-in with Partial RTR . . . 22
- 2.5 A Java virtual machine for RTR computing . . . 28
- 2.6 Conclusions . . . 35

3 System Model **37**
- 3.1 What problems does the RTR-application need to solve? . . . 37
- 3.2 What are the software and hardware requirements? . . . 38
- 3.3 How does the global model look like? . . . 38
- 3.4 Switching between hardware and software . . . 39
- 3.5 Switching status in large applications . . . 41
- 3.6 Summary . . . 41

4 Using the Virtex FPGA: JBits **43**
- 4.1 Overview Components JBits . . . 43
- 4.2 Setting up a skeleton JBits program . . . 43
  - 4.2.1 Setting up communication with the FPGA . . . 44
  - 4.2.2 Modifying the bitstream and writing to the FPGA . . . 44
  - 4.2.3 Reading back the bitstream from the FPGA . . . 45
- 4.3 Writing and debugging a counter program . . . 45
- 4.4 Cores and routing . . . 47
- 4.5 The status of a Core . . . 51
- 4.6 Summary . . . 51

5 Developing the RTR application **53**
- 5.1 Functionality . . . 53
- 5.2 ReconfigInterface . . . 54
  - 5.2.1 CoreStorage . . . 54
  - 5.2.2 CoreList . . . 55
  - 5.2.3 WireManager . . . 56
  - 5.2.4 ReconfigInterface internal functions . . . 56
- 5.3 UserProgram . . . 57
  - 5.3.1 CRC: Switching between software and hardware . . . 58
  - 5.3.2 Software to hardware translation and vice versa . . . 59
  - 5.3.3 CommandLine . . . 60

5.4 Scenarios 61
5.5 Conclusions and future improvements 61

A References **63**

Chapter 1

Introduction

1.1 Introduction

In todays society a lot of hardware is used: in computers, television, cars and microwaves. This hardware consists mostly of ASICs (application-specific integrated circuits). In, for example, a computer the processor (CPU) inside is an ASIC. It is a static circuit, it has been produced in a factory and it will not change functionality; it will not become more or less than it is. In a microwave the ASIC might be a simple circuit that counts down the time of the timer that sets how long the food has yet to be prepared. All these circuits will not change functionality.
An FPGA on the other hand is a different kind of circuit. FPGA is the abbreviation for Field Programmable Gate Array. An FPGA consists of an array of logic cells. Each of these logic cells can independently take a certain function, for example an OR or AND port. The array is logic cells is interconnected by a matrix of wires and switches (which control which wires are connected) to allow for communication or data transfer between the logic cells. In contrast to an ASIC, the FPGA *can* change functionality. Depending on the type of FPGA, most or all logic cells can be (re)configured to change from one function to another. This is what the "Field Programmable" implies in FPGA: the functionality can be configured by the end user (in the field) and not in the factory.

The FPGAs available today can roughly be divided into two groups: FPGAs with coarse grained logic cells and FPGAs with fine grained logic cells. Fine grained FPGAs are made up mainly of gates or transistors or small macro cells. Coarse grained FPGAs however are made up of larger macro cells. These large macro cells can be made up of, for example, *flip flops* and *look up tables (LUTs)* which make up for combinatorial logic functions. Macro cells themselves often also contain switches (also known as *multiplexers (MUXs)*) which allow for different uses of the macro cell. The most renown manufacturers of FPGAs are Xilinx (produces mainly coarse grained FPGAs) and Altera (mainly fine grained FPGAs).

After having explained the basic architecture/functionality of an FPGA, one would wonder what their main advantages are over ASICs. Why would one want to use an FPGA over an ASIC? The most common use of FPGAs today is to serve for prototyping of ASICs. First a prototype of a circuit is produced on an FPGA, should it happen that the current prototype still contains errors, then the prototype can be debugged and the FPGA be reprogrammed within seconds to continue testing.
Another advantage is that if an FPGA product is on the market and after a while it appears that it still contains a bug, then the FPGA product is easily reconfigured to correct the bug, whereas a product with an ASIC would need its entire ASIC replaced. A third advantage is that FPGAs have a shorter time-to-market; while a product is still under development, the FPGA itself can already be produced in a factory and quickly be programmed when the product is ready. With an

ASIC, the entire product needs to be completely developed before it can be produced in a factory, thus it takes longer to produce. Another advantage of an FPGA is that it can be used to maintain superiority over competitors. Should a competitor release a superior product, then it is possible to maintain working on ones own product and release a superior FPGA program in a few days time. The last advantage will be that with FPGAs it is also possible to radically change a product. For example it is possible to add a mp3 player to a cell phone.

Having covered what an FPGA is and what its advantages are, it is time to return to the topic of the thesis. The thesis is about developing a program that makes use of the reconfiguring ability of the FPGA. But not just that, it is also about the cohesion between software and hardware. The program should be able to write functions to the FPGA and then run these functions in hardware on the FPGA. A function that is running in the FPGA can either be removed from the FPGA while it is running or when it is done executing. If the function is removed from the hardware while it was executing, then it should be possible to continue executing this function in software. If the function is removed when it is done executing, its calculation results are returned to the program and another function will be able to take its place inside the FPGA.

The requirements just mentioned may sound interesting, but what is its purpose? Why or for what purpose would one need such functionality? An example can be a future cell phone with MP3 playing capability. Say you're listening to an mp3 on your cell phone. To play the mp3, the cell phone needs to be executing a specific function (also known as *codec* from now on) to play the mp3. The cell phone was a cheap one, so it only has very small fast work-memory and a large slow-memory chip. Say someone decided to call you, when that happens you want your cell to ring your favorite ringtone. Since the ringtone is not stored in an mp3 format, several things will happen. First the mp3 stops playing. The cell phone keeps track where the mp3 stopped playing and writes this data to the slow memory module inside the cell phone. Then it retrieves the codec that is needed to play your ringtone and put this codec in the fast work-memory where once the mp3 codec was located. Now the cell phone is able to ring your ringtone. When you answer the phone, the ringtone stops playing and another, yet again different codec, is loaded to play the sounds the person is making on the other side of the phone(wireless)line. When the conversation is finished, the mp3 codec is put back into its place, it retrieves the data it stored where it kept track of how far it was playing the mp3, and it continues playing the mp3 at the place where it broke off at the beginning of the phone call.

The above is deals with a theoretical cell phone, but it does show the power of the FPGA. Another, more theoretical, application of this system is a computer with an FPGA device attached. The computer needs to execute three very processor-intensive functions. The FPGA can execute these functions as well and moreover, it can execute these functions faster since it is using dedicated hardware (the FPGA). The only problem is that only two functions fit on the FPGA. To handle this problem, it places the first two functions on the FPGA and runs the third function in software. At a certain point in time, the first function, which is running on the FPGA, needs the results of the third function. A possibility now is to remove the first function from the FPGA (and record its current status), then replace the first function inside the FPGA by function three. Of course function three had already done some calculation in software, so these results are also send to the function three inside the FPGA. Function three continues running in hardware until it is done. When it is done function three is removed from the FPGA, along with the final result it calculated and function one is put back into its place which receives its previous status along with the results of function three. Function one and function two calculate until they're done calculating and pass the result back to the main program which decides what to do with it. In this example, the main advantage of using the FPGA is the extra speed it gives. The FPGA is able to do this because unlike the host processor in the computer (which is running an OS), it does not have to deal with cache misses, difficult addressing calculations or Operating System overhead. In practise it has been proven that a low clock-speed FPGA can beat a fast workstation because of these advantages.

1.2 FPGAs explained

Before diving too deep into the subject of this thesis, it is good to have an understanding of the way an FPGA works. In this section the FPGA will be examined in greater detail. The FPGA used for this thesis is a coarse grained FPGA: the Xilinx Virtex. To start it should be said that there are many different Virtex FPGAs, usually the only main difference is the size in how many CLBs/IOBs or RAM is added to the FPGA. The following description of the CLBs/IOBs/RAM however is the same for every Virtex device.

The general layout of the Virtex is as follows:

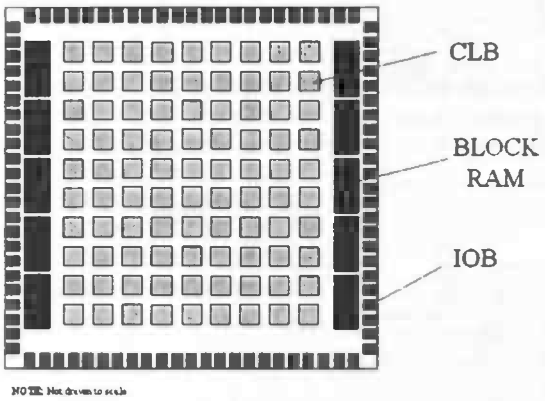


Figure 1.1: Overview Virtex FPGA

An array of logic cells (called CLBs or Configurable Logic Block) is surrounded by IOBs (Input Output Block) and BRAM (Block RAM). In the following subsections these elements will be explained.

Configurable Logic Block

The array-elements in figure 1.1 are CLBs. The CLBs define the functionality of a design, figure 1.2 contains a schematic of a Virtex CLB.

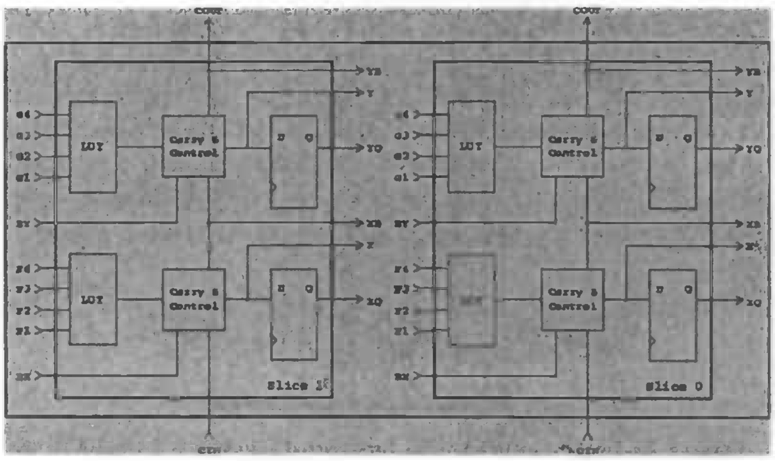


Figure 1.2: Virtex CLB

The CLB can be split into two parts (also called *slices*) by splitting it in the middle. Figure 1.3 shows a single slice.

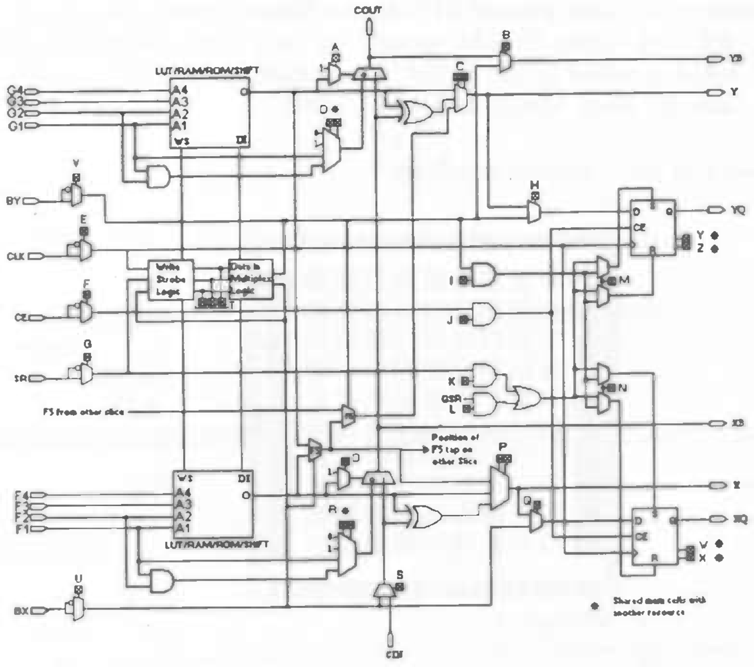


Figure 1.3: Detailed overview of a Virtex CLB slice

Figure 1.3 shows two LUTs (Look Up Table or Logic Function) inside a slice. In the figure each LUT has four input pins. For the G LUT this will be input pins G1 till G4 and for the F LUT this will be F1 till F4. The LUT is a 16-bit look up table in which one can define a function. For example, if one wanted to make the following function ($F1$ *and* $F2$ *and* $F3$) *or* $F4$ one would have the following functionality table:

F1	F2	F3	F4	output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Table 1.1: Logic table for the function ($F1$ *and* $F2$ *and* $F3$) *or* $F4$

In this instance the LUT would have the following configuration:

```
{0,1,0,1,
 0,1,0,1,
 0,1,0,1,
 0,1,1,1}
```

The outputs of these LUTs can be used as input for the H LUT.
Each LUT in figure 1.3 is connected to a D-type flipflop (1-bit memory cell). This flipflop can be set up as a latch (which means it puts through the input value directly) or as a flipflop (means it will put the value through at the end of each clockcycle). The flipflop uses the SR input wire to set the flipflop and the BX/BY wire for reset.

Multiplexers

As can be seen in figure 1.3, several multiplexers appear in the figure, indicated by the letters A till Z. These MUXs configure the CLB. To give an idea what different MUXs do, consider this one:

- A S0Control.YCarrySelect.YCarrySelect
  - S0Control.YCarrySelect.CARRY
  - S0Control.YCarrySelect.LUT_CONTROL

The *A* MUX, called *S0Control.YCarrySelect.YCarrySelect* in the JBits SDK has two different settings, these are *S0Control.YCarrySelect.CARRY* and *S0Control.YCarrySelect.LUT_CONTROL*. If the MUX is set to the former setting, then it will output the value of the Carry chain. If it's set to the latter setting, the MUX will output the value of the LUT output. A complete list of all possible configurations can be found in the JBits v2.8 documentation.

Input Output Block

The IOB serves as a connection between the FPGA and the board it is plugged into. The IOB has the following schematic:

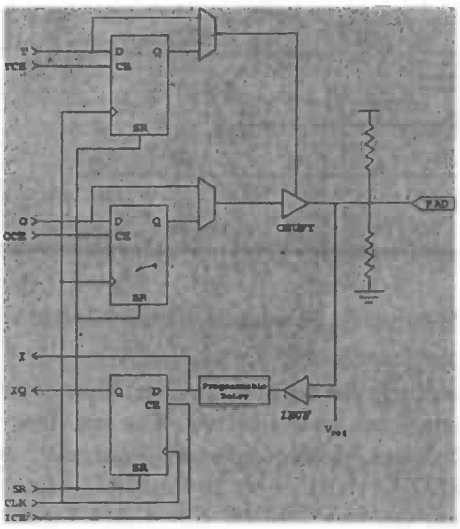


Figure 1.4: Schematic of a Virtex IOB

Each IOB has three pins: one input pin from board to FPGA, one output pin from FPGA to board and a tristate pin. Again there are several MUXs to control the behavior of the IOB.

Sample program: 2-bits counter

This section will discuss a small example of a 2-bits counter program using the FPGA hardware. The counter will have the size of one CLB slice. The YQ and YB wire of a slice of a CLB are used as output wires. The output should be as follows:

Clockcycle	Y	YQ
0	0	0
1	0	1
2	1	0
3	1	1

Table 1.2: Value of outputwires for the counter program

Figure 1.5 shows a slice of a Virtex CLB with all the connections drawn between the pins to construct a 2 bit counter.

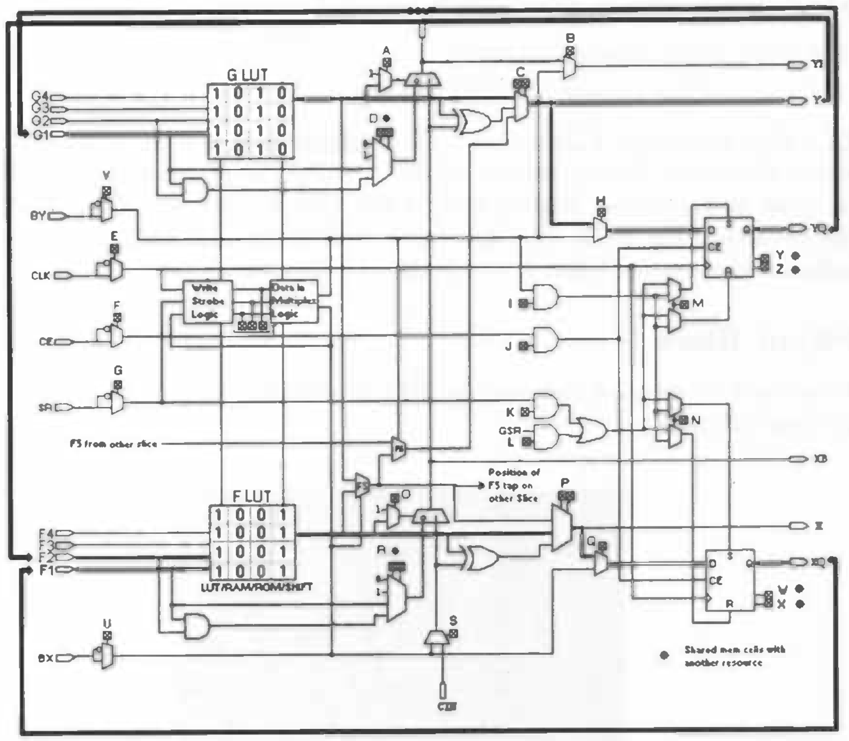


Figure 1.5: Schematic Virtex two-bit counter

The blue lines are connecting pins with each other. How to connect these pins will be covered later, but for now assume they can be connected. The red lines show which internal lines of the slice are important; the thin black lines are not used right now. As one can see the red lines come across a few MUXs. These MUXs must be set to the appropriate value. Following the line coming from the G LUT, it first runs into the C MUX. The C MUX selects which of the three input lines it should give as output. For this counter it should be set the output the red line. After the C MUX the red line also comes across the H MUX. This MUX should also give the value of the incoming red line as output.As one can see the G LUT is configured with the function: *not* $G1$ and the F LUT is the function: *not* $F1$ *xor* $F2$.

The graph in figure 1.6 shows what the values the internal pins will have as clockcycles pass by.

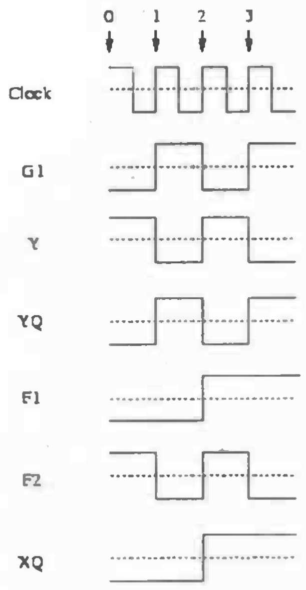


Figure 1.6: Behavior of internal pins for a two-bit counter

As can be seen in figure 1.6, the output given by {YQ,XQ} behaves like a counter.

Chapter 2

Related work

2.1 Introduction

The following sections will discuss relevant papers to this thesis. The first paper: *A hardware genetic algorithm for the Travelling Salesman Problem on Splash 2* shows how an FPGA is used for executing genetic algorithms. While it only concentrates on compile-time reconfiguration, it does give a good example of a problem that can also be solved using a Runtime Reconfigurable Application.
The second paper: *Multitasking in hardware-software co-design for reconfigurable computer* concentrates on developing a model for an RTR-application, though it only remains very abstract and depends on a High-level Language to generate a hardware and a software version for a certain task.
The third paper: *Dynamic hardware plug-in with Partial Run-Time Reconfiguration* shows an interface in which plugins can be inserted. This paper concentrates purely on the hardware side.
The fourth and last paper: *A Java virtual machine for Runtime reconfigurable computing* concentrates on executing a java program as fast as possible by executing some features of the java program in software and other features in hardware.

2.2 A hardware genetic algorithm for the Travelling Salesman

`http://splish.ee.byu.edu/docs/spga.ps`

In this article, the authors, Paul Graham and Brent Nelson describe how to implement a genetic algorithm to solve the travelling salesman problem on a Splash 2 board which contains 30 Xilinx 4010 FPGAs.

Genetic Algorithms

During its operation, the GA algorithm maintains a collection of candidate solutions (also called *population*). Each candidate has a fitness associated with it which indicates its measure of quality. The algorithm selects candidates from the current generation to propagate into the next generation. This propagation may be to simply copy candidates from the current generation to the next or to combine pairs of candidates (called *crossover*). This is comparable to natural mating systems: characteristics from both parents are used to form a new candidate. The selection for candidates to copy or crossover is randomized, but biased towards candidates with higher fitnesses so each new generation is likely to have candidates with higher fitnesses. To prevent converging to local minima, an operation called *mutation* randomly perturbs solutions to yield new ones not otherwise related to existing solutions.

A Genetic Algorithm for the Travelling Salesman Problem

The travelling salesman problem involves finding the shortest path for a salesman through a collection of n cities, visiting each city exactly once and returning to the starting city. Each candidate solution to this problem consists of an ordered list describing the sequence of the cities visited. This list is called a *tour*. To solve a TSP, the object is to find the shortest tour, so the fitness can be defined as the length of the tour. Crossover is performed by taking 2 tours (tour A and tour B), cut both tours in two at a random cutpoint. Offspring child number 1 will consist of the head of tour A with the tail of tour B with cities not contained in the head of tour A. Offspring child number 2 will consist of the head of tour B with the tail of tour A with cities not contained in the head of tour B. Mutation is performed by reversing the list of cities visited of a subtour of a candidate solution. The endpoints of this subtour are chosen randomly.

The above algorithm lends itself to parallelization in at least two ways. Several copies of this algorithm could be run in parallel and take the best result of these runs. Another cooperate model could be applied where islands of computation are executed in parallel but periodically exchange solutions with one another through *migration*. This way local minima are avoided.

Splash 2

The Splash 2 is a reconfigurable computer hosted on a Sun Sparc and consists of an interface board and a collection of processor arrays, each processor consists of 4 Xilinx 4010s FPGAs. Spash 2 is programmed using VHDL.

Basic Algorithm Implementation

Each processor consists of 4 FPGAs and their memories and are arranged in a bidirectional pipeline as indicated in figure 2.1.

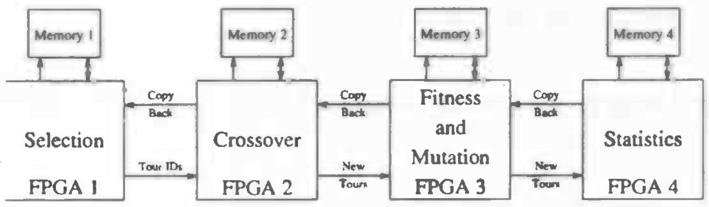


Figure 2.1: A single genetic algorithm processor

The functions of the FPGA are as follows:

- FPGA 1 chooses tour pairs randomly from memory and passes these to FPGA 2
- FPGA 2 has two choices when given a tour pair. It can either copy the tour pair unchanged to FPGA 3 or perform a crossover and then pass them on. Good results prove to occur at crossover chances between 10% and 60%.
- FPGA 3 calculates the new fitness values for tours formed by crossover. It also randomly selects tours for mutation, mutates them and sends them to FPGA 4.
- FPGA 4 writes the new population to memory and records the best and worst tours for this generation and the best tour to date.

All candidate tours are processed by this algorithm to form a new generation. This process is repeated until it has finished a number of generations determined by the user.

Num of cities	Pop. size	Crossover Probability (%)	Mutation Probability (%)	Ave. Exec. Time Hardware	Ave. Exec. Time Software	Soft / hard
24	128	10	10	4.38	43.7	9.97
24	256	10	10	11.23	118.7	10.57
120	256	60	10	295	1999.9	6.78

Table 2.1: Comparison of hardware and software Execution times

The above algorithm was run in hardware at a frequency of 11 Mhz. It was also run in software on a 125 Mhz HP PA-RISC workstation. The results can be seen in the table 2.1.

The hardware and software implementation both earned solutions of equal quality. From the table can be seen that the hardware implementation is significantly faster. The modest software performance is attributed to operating system overhead, TLB and cache misses, complex addressing calculations and strict sequential thread of control. On the other hand the hardware implementation uses a custom 4-stage pipeline, archives nearly 100% memory bandwidth utilization and incurs no overheard for the operating system, address calculations or cache misses.

Two parallel implementations

In the implementation described only 4 FPGAs were used. Whereas the author of the article has access to a Splash 2 board with 30 FPGAs. Because of this it is possible to run 7 additional copies of the program which results in an eight-fold increase in search rate. This model is called the trivially parallel model.
A more interesting approach is the island model. In this model several searches are conducted simultaneously with periodic migration of solutions between search islands. Figure 2.2 gives a diagram of this model.

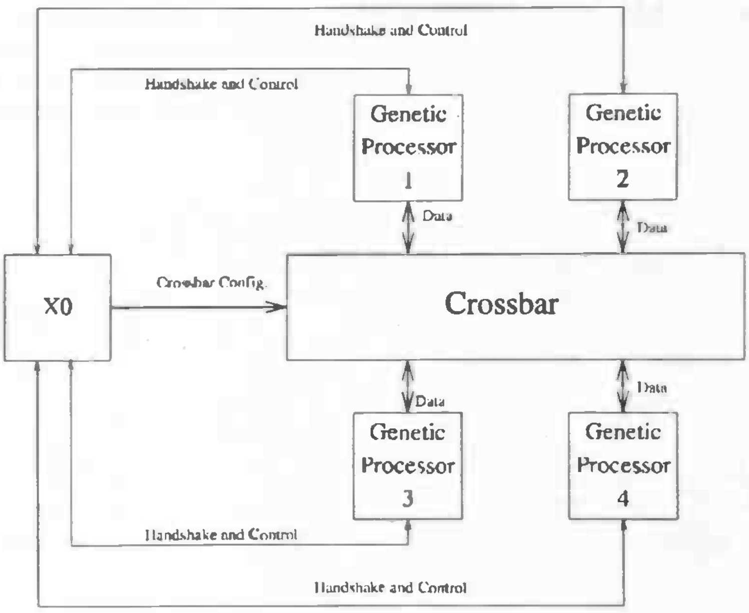


Figure 2.2: The island parallel model

During the migration, each island broadcasts a subset of its tours to the other islands via the crossbar. The receiving islands replace random solutions in their own population with the received migrated tours.

Performance results

Three computation models were compared, these were the 1-processor design, the 4 and 8-processor trivially parallel designs and a 4-processor island model. It was shown that more processors is always better, but marginally so. For long runs (3.5 billion cycles) the 8-processor parallel design gives an answer about 4% better than answer than the single processor design. The 4-processor island model performs best and is about 6% better than the single processor design. This model also finds solutions the fastest. At 400 cycles the 4-processor island has found a solution which requires 990 million cycles for the 8-processor model and 1.7 billion cycles for the single processor model.

Conclusions

- Modest hardware resources outperformed a state-of-the-art workstation. The SPGA with its custom design avoids operating overhead, TLB and cache misses, and complex addressing calculations. In addition, it makes use of pipelining to achieve parallel execution.
- The individual data objects manipulated by the algorithm are small, mostly 8- or 16-bits with a few 24-bits values. This is a good match for the computation and storage capabilities of today's FPGAs.
- The arithmetic requirements of Splash 2 are modest, consisting of small word additions, subtractions and comparisons. This is a good match for an FPGA.
- The additional work requires to create parallel implementations of the algorithm is minimal. The parallel versions managed to find 'good' solutions in far less time then the single processor version.
- The genetic algorithm keeps searching for better solutions, whereas other search algorithms usually stop at a specific solution. Hardware describes as herein may be required to take advantage of this for long execution cycles.

2.3 Multitasking in hardware-software codesign

In this paper the authors, T. Wiangtong, P.Y.K. Cheung and W. Luk describe a System model for executing tasks in hardware and software. In this section I present a subtraction of this paper.

`http://www.ee.mut.ac.th/research/2003/PS_PPfinal.pdf`

Introduction

Most reconfigurable systems contain both hardware and software tasks which must work cooperatively with each other. In most previous research work done in hardware-sofware codesigning, a lot of realistic issues as bus and memory contentions are ignored. Moreover, traditional formulation of the partitioning and scheduling problem employs a hardware model that is completely different from that used in software. This is understandable because the hardware tasks are implemented separately from the software, with different optimalization constraints.

This paper reports a new method of constructing hardware tasks in a codesign system that makes them more compatible with software tasks while retaining the benefit of concurrence as found in hardware implementations. At the same time, it exploits the benefits of software: modularity, cohesion and structured approach. In this paper a previously presented partitioning and scheduling algorithm is applied to the reconfigurable computer system known as UltraSONIC.

The novel contributions of this paper are: 1) a new way of structuring and modelling hardware tasks in a codesign system; 2) a partitioning and scheduling algorithm that employs this method; 3) applying the algorithm to a realistic reconfigurable computer system; 4) evaluation and verification of the model using the reconfigurable computer.

System model

Figure 2.3 shows the system model used in this paper. In this model, a single processor (software) resource SW capable of multitasking, and a number of concurrent hardware processing elements (PE0 to PEn) which are implemented on FPGAs, are used. System constraints as shared resource conflicts, reconfiguration time of the FPGAs and communication time are taken into account.

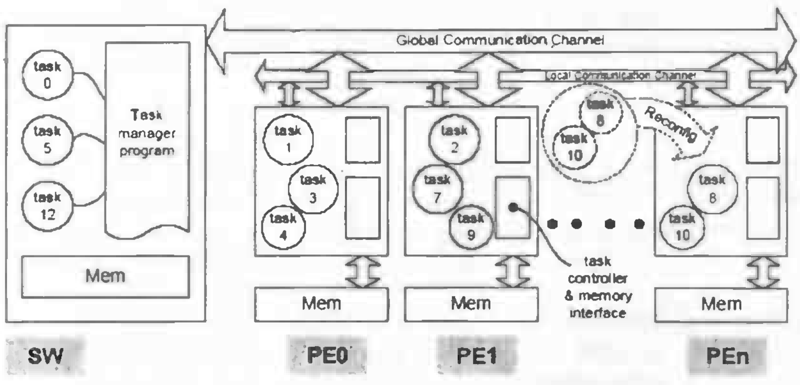


Figure 2.3: Codesign System Modelling

The assumptions in this model are:

- tasks implemented in each hardware PE are coarse grain tasks representing blocks, loops or functions.

- local memory is a single port, and only one task can access the local memory at any given time.
- tasks for a PE can be dynamically swapped in through dynamic reconfiguration.
- each task can fit into a single PE.
- multiple tasks residing in a PE execute sequentially.
- tasks residing across different PEs can execute concurrently.
- a global communication channel is available for the processor and the PE to communicate with each other.
- local communication channels are available for neighboring PEs to communicate with each other in a pipeline ring.
- a task manager program in the software processor is used to manage all tasks, software *and* hardware.

This system model makes the hardware tasks look very much like software tasks. In this way, management of task scheduling, task swapping and task allocation can be done in a uniform manner regardless whether it is a software or hardware task. Concurrency is not affected as long as concurrent tasks are mapped onto separate PEs.

Codedesign environment

Figure 2.4 represents the codesign of the environment of our system. The system to be implemented is assumed to be described in s suitable high level language which is then mapped to a directed acyclic graph (DAG). In the DAG, tasks are represented by nodes and communications by edges. The nodes are then divided into hardware or software tasks by the algorithm described in a previous paper [4]. After having obtaining this partitioning information, a task manager (TM) program can be produced (figure 2.5).

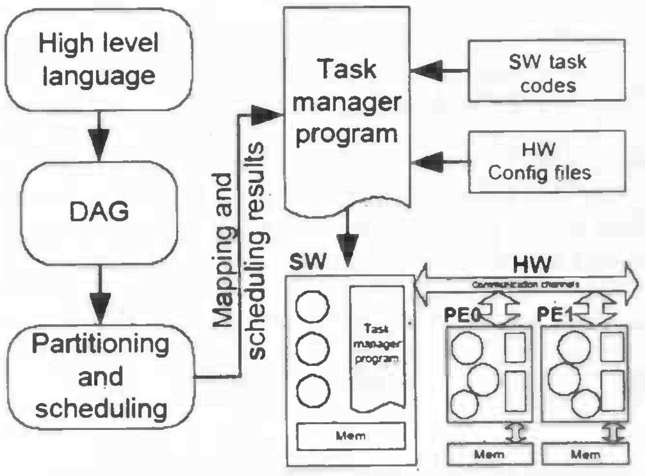


Figure 2.4: The codesign environment

The Task Manager program is responsible for managing both software and hardware tasks. It controls the sequencing of all the tasks, transfer of data and the synchronization between them,

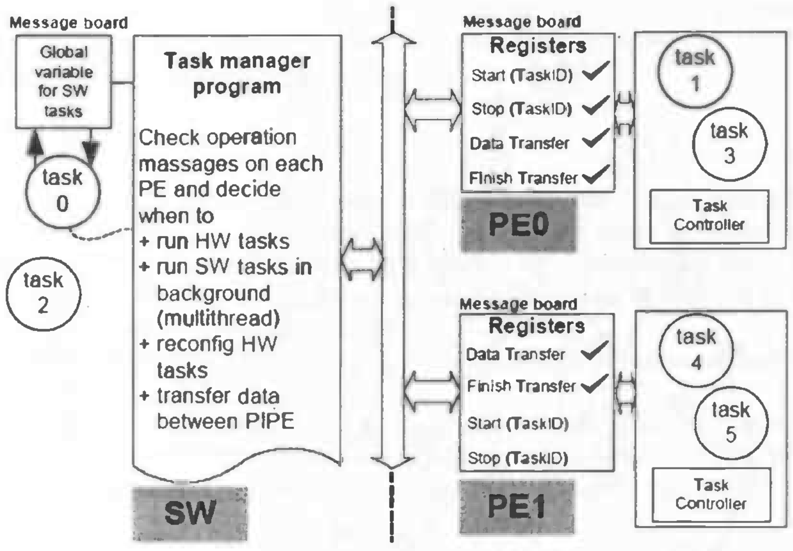


Figure 2.5: The task manager

and the dynamic reconfiguration of the FPGA. As can be seen in figure 2.5, the TM communicates with a Local Task Controller on each PE to assert control. Each PE has access to a message board to retrieve/write commands/status from the TM.

The system uses a message-based protocol when running a process. Using this method, the tasks on each PE are run independently by the TM. Hence the system runs asynchronously at system level. In task-level however, functions are executed synchronously, thus making execution time predictable.

Verification in reconfigurable computer

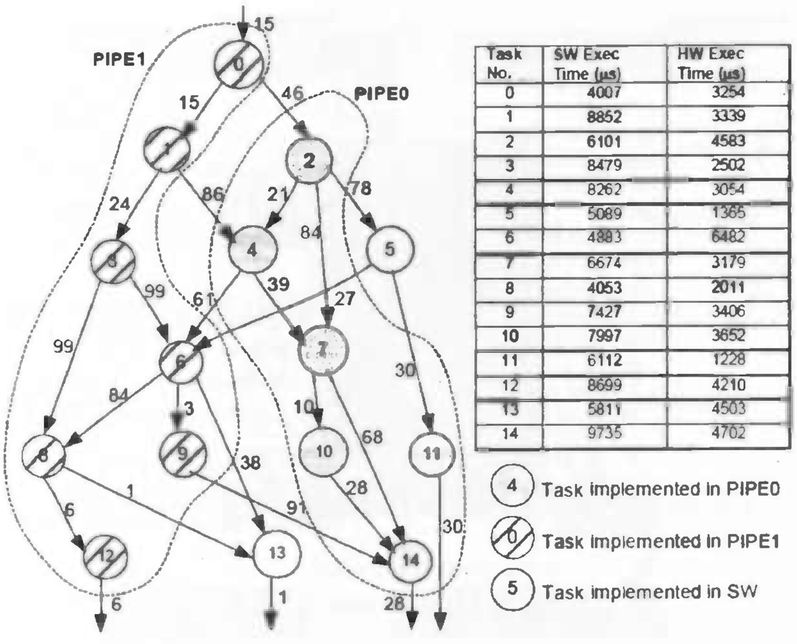
In order to verify the practicality of our model and the effectiveness of the partitioning/scheduling algorithm, the system described was implemented on the UltraSONIC. UltraSONIC employs PIPE busses (PIPE0 to PIPE15) for global communication between software and hardware.

Results

In order to verify that the implemented system works, the partitioning and scheduling program was applied to a randomly generated DAG which contained 15 tasks. This is shown in figure 2.6 alongside with the execution time of each task in software and hardware respectively. Numbers on the edges is the amount of data (KB) transferred between tasks.

As can be seen in figure 2.6, tasks $0, 1, 3, 6, 8, 9, 12$ are implemented in PIPE1 and tasks $2, 4, 7, 10, 11, 14$ in PIPE0. Tasks 5 and 13 are implemented in the host processor.

The DAG is artificially generated and is used to verify the operation of the entire system by emulating each task with a dummy hardware or software module. Each dummy task module mimics the exact input and output data transfer and takes the same amount of time to execute as the actual system.



Task No.	SW Exec Time (µs)	HW Exec Time (µs)
0	4007	3254
1	8852	3339
2	6101	4583
3	8479	2502
4	8262	3054
5	5089	1365
6	4883	6482
7	6674	3179
8	4053	2011
9	7427	3406
10	7997	3652
11	6112	1228
12	8699	4210
13	5811	4503
14	9735	4702

Figure 2.6: Implemented DAG after partitioning

2.4 Dynamic hardware plug-in with Partial RTR

In this paper the authors Edson L. Harta, John W. Lockwood, David E. Taylor and David Parlour describe how to make use of runtime reconfigurability using Dynamic Hardware Plug-in (DHP) modules to implement this on their Field Programmable Port Extender (FXP) which can be used in hardware packet processing applications.

`http://www.arl.wustl.edu/arl/projects/fpx/references/DAC2002_DHP_24_2.pdf`

Introduction

FPGAs can make use of their reconfigurability in two ways: Compile-Time Reconfiguration (CTR) or Run-Time Reconfiguration (RTR). CTR systems do not change their FPGA-configuration during their lifetime. RTR systems can change their configuration using either full reconfiguration or partial reconfiguration.
The research in this paper focuses on partial RTR in an FPGA for use with hardware packet processing applications using Dynamic Hardware Plug-in (DHP) modules. A DHP is defined as a module which can be loaded into or removed from a running FPGA without disturbing other circuits operating in it. This is particulary useful in packet processing applications where it is not desirable to suspend operation of a network during reprogramming.
The Field Programmable Port Extender (FPX) is the FPGA-based prototyping platform used in the Washington University Gigabit Switch.

Partial reconfiguration

Partial reconfiguration allows an FPGA to implement multiple functions and to change those functions while the system is running. The FPGA is partitioned into a static infrastructure region and a number of DHP sites. The infrastructure connects each DHP site to shared resources and/or other DHP sites.

The layout and interface specification for the DHPs depends on the architecture and behavior of the FPGA. The characteristics of the Xilinx VIRTEX-E family are described below. Following this description, the tool used to generate the partial configuration bitfile is presented, along with the requirements for the interface between the static infrastructure and the DHP sites, and the methodology to generate the configuration bitfiles used by PARBIT.

Virtex-E architecture

The application presented in this paper targets a device from the Xilinx Virtex-E family. Programmable Input/Output Blocks (IOBs) around the edge of the array are used to communicate with off-chip resources. The interior consists of a matrix of Configurable Logic Blocks (CLBs) containing: lookup tables, flip-flops and programmable interconnect. A number of columns in the CLB matrix are replaced with Block SelectRAMs.
Virtex-E configuration bits are organized in columns corresponding to a column of the FPGA's logic resources.
To configure the VIRTEX-E FPGA, a series of bits, divided into fields of commands and data, are loaded into the device. Each one of the configuration columns are divided in smaller slices, called frames. A frame is the smallest part of the configuration memory that can be written or read.

PARBIT

In order to handle the partially reconfiguring a tool called PARBIT has been developed. To execute a reconfigure command, the tool utilizes the original bitfile and the target bitfile and parameters given by the user. These parameters include the block coordinates of the logic implemented on a source FPGA, the coordinates of the area for a partially programmed target FPGA, and the programming options. The tool reads the original configuration bitfile and copies to the partial bitfile only the configuration bits related to the area defined by the user. The target bitfile is used by PARBIT to copy the configuration bits that are inside a column specified by the user, but outside the partial reconfigurable area. This happens due to the fact that one frame takes all the rows of a column and the partial reconfigurable area is smaller than a whole column.
PARBIT allows arbitrary block regions of a compiled design to be re-targeted into any similar size region of an FPGA.

Gasket Interface

DHP modules which are downloaded into the FPX need communication with the infrastructure logic on the FPGA. This communication is done using interconnection points. These points are connected by special wires called antennas. Gaskets allow DHP modules to be implemented with only minor modification to the standard FPGA design flow. When building a DHP, PAR (Place and Route) is made to only make use of included routes. It may not use excluded routes. Figure 2.7 shows how antennas cross out of the DHP free routing zone.

Generation of bitfiles

In order to generate bitfiles which can be used by PARBIT to create the partial bitfile, it is necessary to follow a few rules regarding the logical names of the entities that make up the logic within the FPGA. These rules are applied during the synthesis, routing, and placement of the FPGAs that hold the infrastructure circuit and the DHP modules.

Synthesis

The infrastructure bitfile contains the fixed logic area in the FPGA, encompassing all of the I/O pads, signals and flops that interface to the module, and logic that make up the on-chip system. In the VHDL file, there is one entity, called "INFRA", that contains the infrastructure, and one or

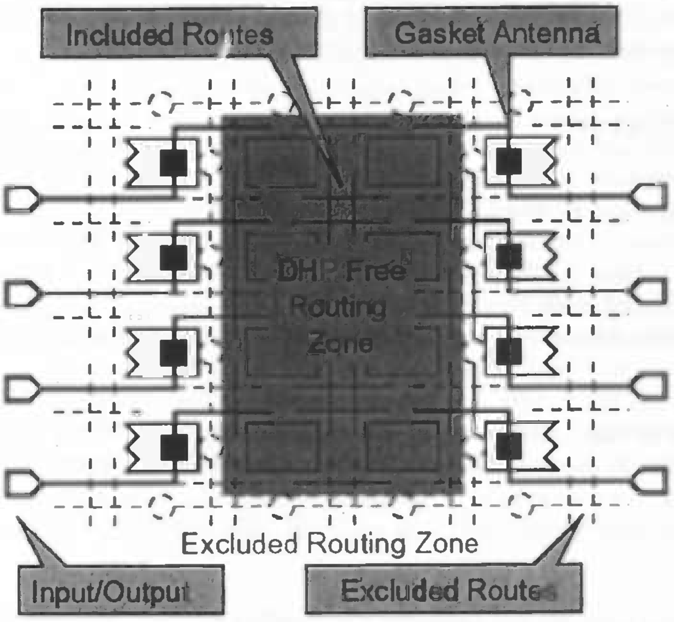


Figure 2.7: Gasket antennas

more entities, called "GASKET", which contain the flops used to interface with each user module. Additional VHDL files can be used for generating additional constraints for locking the gasket flops in fixed positions inside the FPGA and to reserve space for the DHP modules into the infrastructure. The bitfile for the DHP contains the description of a module and is generated in a similar way. The difference is that there is one user module entity, called "DHP", connected to one GASKET entity, as shown in Figure 2.8. For the GASKET entity in the design, the constraints are set so that this GASKET is placed in the same position as the first GASKET of the infrastructure design.

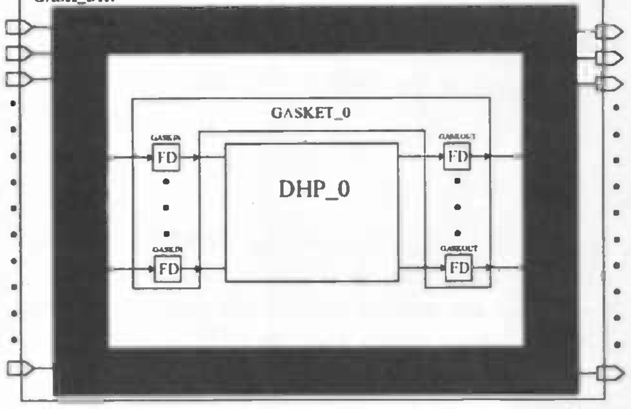


Figure 2.8: Logical Design Entities

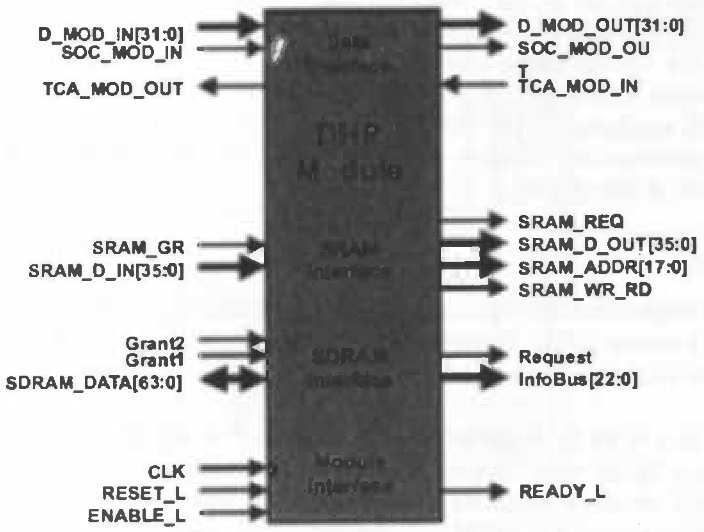


Figure 2.9: Modular component of FPX

DHP implementation on the FPX

Placement

The placement of infrastructure and DHP can be accomplished with conventional FPGA placer tools, that have the ability to constrain logic to specific regions of the array. It should keep infrastructure logic out of the DHP sites, and DHP logic is confined to use the appropriate rectangular area.

Routing

To make sure there is no interference between DHPs and infrastructure as new DHPs are being configured, the nets in a design are sorted into one of three categories and then routed with special constraints as follows:

- All nets which are internal to the DHPs do not cross the boundary of the DHP site.
- All nets which are completely contained in the infrastructure are routed using any wiring resource that is not used for use in routing DHPs.
- Nets which cross the the boundaries between the infrastructure and the DHP site are forced to follow the same path for each DHP. These are the gasket antennas which are identical for every DHP

The FPX platform

The Field-programmable Port Extender enables the rapid prototype and deployment of hardware components for modern routers and firewalls. The system allows new packet processing functions to be quickly prototyped as DHP modules in hardware, then downloaded into reconfigurable logic over the network. All functions on the FPX are implemented with FPGAs. The core functionality of the FPX is implemented on the Networking Interface Device (NID) and on part of the Reprogrammable Application Device (RAD). The NID is a Xilinx XCV600E FPGA that contains the control logic to reconfigure regions of the RAD. The RAD is a Xilinx XCV2000E FPGA that holds the DHP modules.

In order to reprogram a **RAD** module, the NID implements a reliable protocol to fill the contents of the an SRAM with configuration data that is sent over the network. A final control cell is sent to NID to initiate the reprogramming of **RAD** using the contents of the reprogram memory.
The NID also contains a network switch that forwards individual traffic flows between network interfaces and DHP modules on the RAD. This combination of partial reconfiguration control logic and per-flow routing circuits allow the FPX install new DHP modules without affecting the operation of the rest of the system.

Modular Logic

DHPs are used to implement application-specific functionality on the FPX. Multiple DHPs can run in parallel on a single FPGA device and data flows may pass through these plug-ins at the same time. In order to support a broad spectrum of applications, DHPs can access off-chip memory resources.
The modular interface of an FPX component is shown in Figure 2.9. Data arrives at and departs from a module over a 32-bit wide Utopia interface. The module provides two interfaces to off-chip memory. The SRAM interface supports transfer of 36-bit wide data to and from off-chip SRAM. The Synchronous Dynamic RAM (SDRAM) interface provides a 64-bit wide interface to off-chip memory.

DHP implementation on the FPX

Figure 2.10 shows the infrastructure of the RAD when viewed by the Xilinx FPGA Editor. As one can see the infrastructure is put on the left and right of the FPGA; leaving space for two DHP modules in the middle. Figure 2.11 shows the floorplan for the the FPGA circuit which implements a DHP. Note that input/output signals are routed to I/O pins so that standard design flows for synthesis and simulation of the FPGA circuit can be followed.
Current research efforts focus on implementing of enhanced on-chip gasket interfaces that allow for additional types of signals.

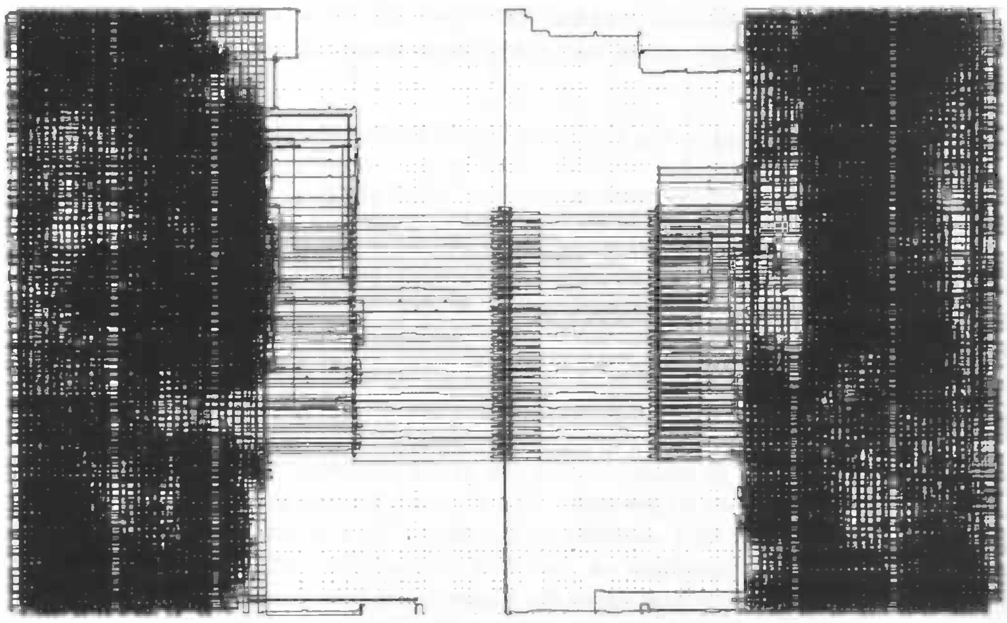


Figure 2.10: Infrastructure of the RAD (a Xilinx Virtex 2000E FPGA)

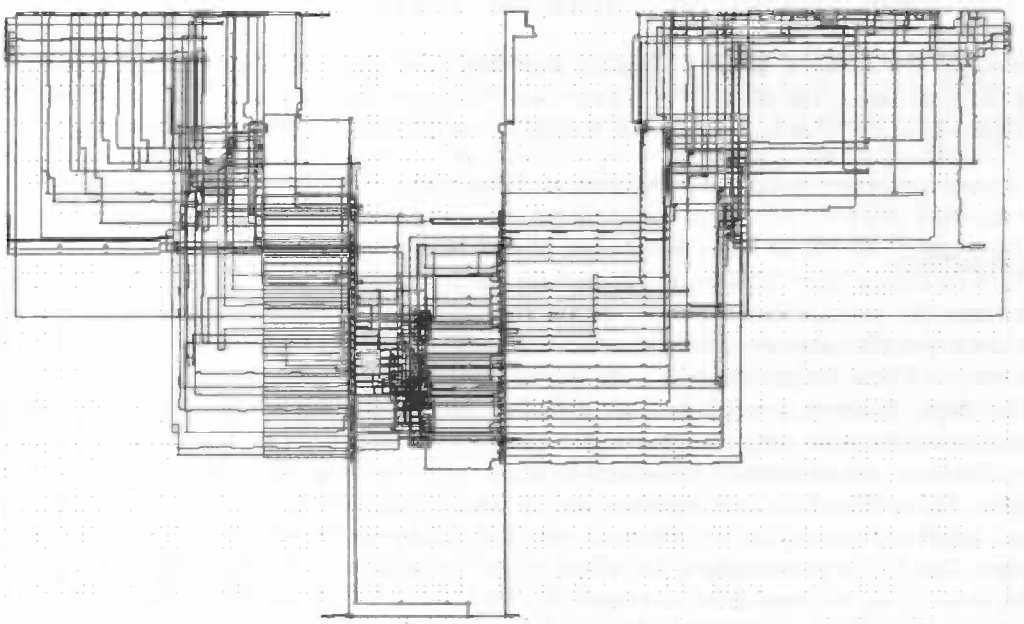


Figure 2.11: DHP floorplan of the RAD

Conclusions

A technique for designing partial RTR systems on an FPGA was demonstrated. The methodology uses PARBIT to generate the partial bitfiles. The interface between the DHP plug-ins and the infrastructure, called a gasket, is able to lock fixed interconnection points between infrastructure logic and dynamically reconfigured regions of an FPGA. The approach demonstrated reduces the time to implement a new hardware module due to the fact that the modules are pre-compiled and that the reconfiguration is performed in hardware.

2.5 A Java virtual machine for RTR computing

In this paper the authors, Brian Greskamp and Ron Sass, describe a new Reconfigurable Computing (RC) platform, the RTR-JVM which makes use of ordinary Java programs and uses online algorithms to decide whether the feature should be executed in hardware or software.

`http://www.nomotive.org/rcc/RC-JVM-profiling.pdf`

Introduction

The demand for computational power is growing rapidly. To date, most performance improvements of computational power have stemmed from improvements of the theoretical Von Neumann Architecture. These designs execute a sequenced stream of instructions from a fixed instruction set. This paper however, concentrates on architectures which do not have a fixed instruction set limitation; architectures which are reconfigurable.
RC architectures are of interest because they have been shown to speed up a wide range of applications. RC architectures are characterized by having a limited amount of programmable logic in which arbitrary circuits can be executed very fast. However not all algorithms perform well in hardware. The RC implementation described in this paper aims at handling both cases.
The RC system can be considered to consist of a traditional Von Neumann processor ("processor" for short) and an FPGA as shown in figure 2.12.

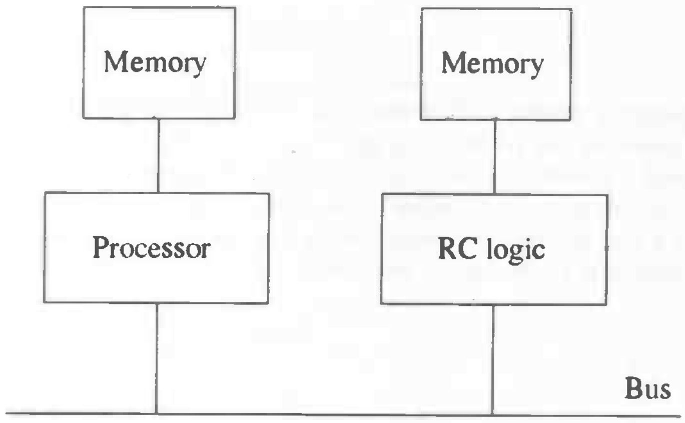


Figure 2.12: A simplified RC system

The program that should be run on the RC system should be partitioned between processor and RC logic. The smallest parts of the program which may move between software and hardware are defined as *features.* A feature may, for example, be a Java class file or a basic block. The set of features resident in hardware at any given time is called the *feature set.* All non-resident features are executed in software.

Background

The main problem with current RC systems is that they are difficult to program. This is caused by the need for the programmer to design a separate design for software and hardware algorithms. On top of this, the design methodologies between software and hardware are vastly different in concept. One way of solving this problem is to have a high level language which is able to generate

both software and hardware designs. Current tools like Transmogrifier and Handel-C are already capable of doing this.
Another problem is that the RC system needs to partition the features between hardware and software. The challenge is to find an optimal feature set which consists of the features which contribute to the greatest overall speedup of the application. Note that the optimal feature set is a function of both time and the input to the program. Also notice that due to the limited capacity of the RC logic it is not possible for the feature set to consist of all features.
Current approaches to solve the partitioning problem usually deal with browsing through the program at compile-time to search for hot spots and map these areas to the RC logic or to put all features in the RC logic. Clearly the latter approach does not work for large programs whereas the first approach can be difficult and inefficient especially when the control flow is complex. These problems can be overcome by using a *online architecture* which makes use of additional information gained during program execution to select, synthesize and instantiate hardware features.
The process which constructs and continually updates the feature set is called the *feature selection algorithm*. This algorithm must solve an *online problem*: instructions are revealed one step at a time and the algorithm must predict future inputs based on the past ones.

Technology primer

This section describes the terminology used in following sections.

The Java Virtual Machine

Java is a popular programming language. When a computer needs to execute a java program, this is done in a *Java Virtual Machine*. There are two kinds of JVMs. The simplest interpretive JVM fetches one java instruction at a time and executes a series of actual machine instructions for each one. The other "Just In Time" (JIT) JVM compiles the VM code to native code before executing it, which yields a performance increase.

FPGAs

An FPGA contains an array of identical logic units called Combinatorial Logic Blocks (CLBs) as shown in figure 2.13. Circuits are realized by programming the routing network to connect the appropriate CLBs in the desired way. This routing network is a two dimensional array of switch elements. The state of a switch is stored in a configuration RAM cell. To reconfigure an FPGA, different configuration data is stored in the RAM cells. The data to program the configuration RAM is called the configuration bitstream.

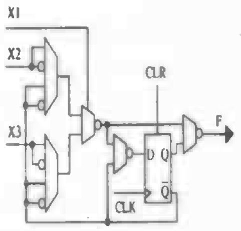


Figure 2.13: FPGA internal structure

To generate a configuration bitstream from a high-level language, *synthesis* software is used. This software first synthesizes a program described in a hardware description language (HDL) into a netlist which describes the circuit in terms of interconnected components. The next phase, "place and route", maps the netlist to a specific FPGA architecture. The synthesis can take up to many hours to complete, whereas transferring the resulting bitstream to the FPGA only requires microseconds.

The RTR-JVM

In order to test the proposed feature selection algorithms, a prototype online RC platform was developed called the "Runtime Reconfigurable Java Virtual Machine"(RTR-JVM). An overview of the RTR-JVM is shown in figure 2.14. As can be seen, all methods are intercepted by the profiler module which takes care of collecting performance data which can be used by the feature replacement thread to decide which features move in and out of the hardware. The dispatcher takes care of transferring arguments to and from the hardware. The diagram also shows the feature set of which it is important to note that it is read-only: no new features are synthesized at runtime.

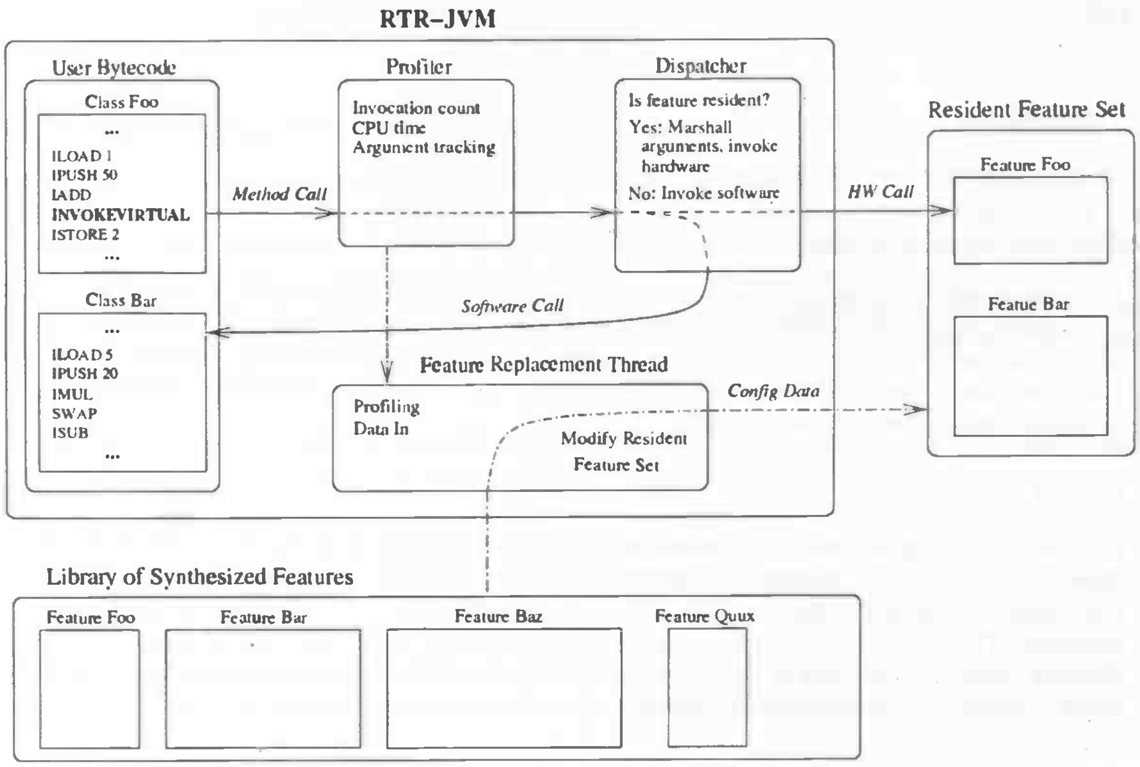


Figure 2.14: RTR-JVM block diagram

Limitation and assumptions

The RTR-JVM is based on the Kaffe v.1.0.7 JVM, chosen for its open license and stability on the Linux platform. The interpretive JVM engine was chosen for ease of implementation. The Forge Java-to-Verilog synthesis tool from Xilinx was chosen to generate Verilog VHDL files from Java .class files. This tool does require that all references to objects and arrays inside the java class file are resolvable in the class constructor. In addition it is not possible to use floating-point variables and classes must be a leaf in the cell graph: they may not call methods out of their own class.
The Synopsys synthesis tool chain is used to link custom VHDL "glue logic" components which allow for the host to communicate with the Forge-generated cores and the resulting design is synthesized with XST, the Xilinx Synthesis Tool. The resulting configuration bitstream can be used

to program the FPGA with. Currently, all instantiable features are pre-generated before JVM startup.

Feature Selection Algorithm

The goal of the algorithm is to determine **H**, the optimal feature set. Intuition suggests it is desirable to select features that:

- use a lot of processor time
- do enough computational work to offset communication overhead (argument passing)
- have a fast hardware implementation
- have a hardware implementation which doesn't use many resources

The last two items indicate it is necessary to know the resource requirements and throughput of the hardware component in advance. Resource requirements are expressed in the amount of *slots* it takes up. A slot is the smallest allocable RC unit. In the prototype, each slot consists of an entire FPGA. The amount of slots a feature f takes is referred to as $slots(f)$ and the amount of time a feature takes to execute in hardware as $T_{hw}(f)$.
In addition to the previously mentioned hardware profiling information, information about the software-resident classes must be gathered as well so that the hardware implementations can be compared with the software counterparts. For this purpose, $rate(f)$, the number of invocations per second incurred by the class and $T_{sw}(f)$, the amount of CPU time expended per software invocation of the class, are recorded. These statistics are updated at each time slice.
Using these statistics it is possible to construct the optimal feature set called **H**. The prospective speedup can be expressed as the product of the feature invocation frequency and the per-call time saved when the feature is run in hardware. This yields $S(f)$ as result. The factor d is necessary to compensate for the reduction in invocation rate that would occur if all features were in software.

$$d = 1 + \sum_{F \in H} rate(f) * (T_{sw}(f) - T_{hw}(f))$$

$$S(f) = \frac{rate(f) * (T_{sw}(f) - T_{hw}(f))}{d}$$

Since only a basis for comparing the relative merits of each feature is needed, a more simpler version can be constructed which is proportional to the theoretical speedup:

$$S'(f) = rate(f) * (T_{sw}(f) - T_{hw}(f))$$

To get a view on the cost per feature, the pseudo-speedup $S'(f)$ is normalized with respect to the number of slots required. Finally, to prevent trashing of classes with equal merit, the metric of the features with are hardware-resident are multiplied with the constant β.

$$M(f) = \frac{S'(f)}{slots(f)} * \beta \text{ if } f \in \text{h}$$
$$M(f) = \frac{S'(f)}{slots(f)} \text{ otherwise}$$

The final metric M has units of $\frac{speedup}{slot}$. After obtaining M for each feature, the new feature set can be determined according the following steps:

1. Create a temporary structure to hold the new feature set.
2. Create a table of classes sorted in order of decreasing M.

3. Traverse the table from top to bottom and proceeding until all RC resources have been exhausted, add classes to the new feature set.

4. Synchronize the new feature set with the hardware, revert expired features to their software implementations.

This process works fine, even though there are a few things to take into account:

- The T_{sw} value can not be updated while it is running in hardware, instead its value from the previous time slice is used.

- There is an enforced lockout period at the interpreter startup during which no reconfigurable hardware may be synthesized or instantiated in order to let the running statistics to stabilize.

- Hardware swap events are allowed only to occur every 100mS to ensure the cost of copying the configuration bitstream to the FPGA remains negligible.

Communicating with Hardware

Communication from and to the RC resources over a slow bus as the PCI bus proved to be a challenging task. A traditional approach to group operands, results and configuration data into packets and take advantage of the DMA proved to be inefficient. This approach works well for large packet sizes, but not for transferring small amount of data. An intelligent mechanism to select which transfer approach to use would be the best, but in meanwhile a temporary will do. Each instantiable feature has a shared object associated with it which exports stub functions. These functions are:

- enter_hardware: Called to load the bitfile into an FPGA and to perform initial configuration of the hardware. Additional system resources can be mapped at this time.

- leave_hardware: Called when a class leaves hardware. State information is retrieved and written back into the software object data structures. Resources held by the hardware are released.

- call_X: Called whenever a hardware-resident feature X is invoked. The appropriate state is loaded into the hardware, operands are transmitted, and the result is returned.

The *state information* mentioned above refers to the data associated with a particular feature instance. Currently the state is only stored in software structures of the JVM and must be synchronized with the hardware after a method is applied to a new object.
Currently the stub methods mentioned above must be hand-coded on a per-class basis, but nothing precludes automatization.

Results

Prototype Hardware Platform

The RTR-JMV platform consists of a ACE2 Reconfigurable Computing card which is installed on an x86-based Linux PC. The ACE2 card carries two Xilinx 4085 FPGAs for a total of 6200 CLBs. Since only small data transfers are involved when invoking a feature, the Linux device driver was modified to do memory mapped transactions (instead of DMA) to gain some extra performance.

Experiments

To make the RTR-JVM successful, communication overhead between software and RC resources should be kept to a minimum. Three communication methods were considered: DMA, PIO and memory-mapping. DMA makes optimal use of the bus bandwidth by transferring the data across the bus in blocks, but each transfer requires many cycles to prepare. It requires involvement of the operating system as well (system call) and suffers from high latency. Using PIO mode, each word of data is across the bus separately, yielding lower bandwidth and also lower latency. A system call is still required. Memory mapping is a technique that gives an application direct access to memory so that transfers can be performed without a system call. As in PIO mode, bandwidth is suboptimal, but latency is minimal. Figure 2.15 shows the relative speeds of each method when transferring three words of data.

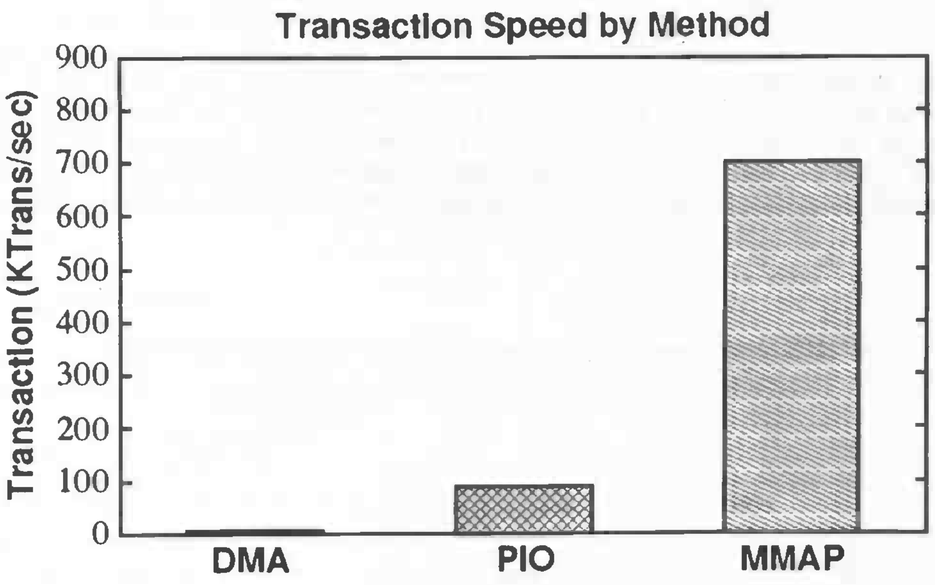


Figure 2.15: Small packet transfer rate by method

Experiments showed that when the RTR-JVM was run on a Pentium-III benchmarking system, the RTR-JVM scored 1.19 on the Scimark2 benchmark whereas the Kaffe-1.0.7 scored 1.17 (higher is better). Clearly the profiling overhead in the RTR-JVM is negligible.
The RTR-JVM was run with only one synthesization feature: an adder class that returns the addition of two integers. The communication-to-computation overhead ratio is very high. Considering this, the JVM ran a loop of six million add() calls in 31.8 seconds as opposed to 29.0 seconds using software.

A non-trivial test kernel

The RC-JVM was tested on a more complex environment as well. This simple test kernel models the task of securely transferring data over a network as shown in figure 2.16. It consists of four synthesization features: a DES encrypter, a DES decrypter, a Hamming code generator and a Hamming code verifier. A test driver generates random data and feeds it to the transmit and receive paths, with $RX\%$ of the generated packets traversing the receive path and the remainder following the transmit path.

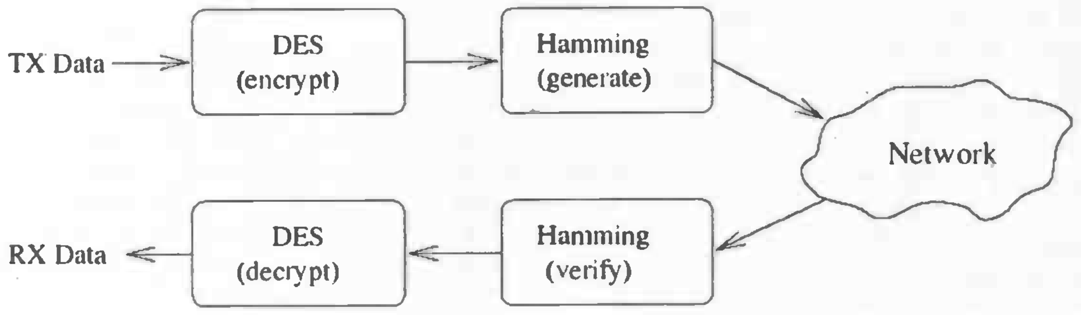


Figure 2.16: Test kernel with four instantiable features

All four of the synthesization features experienced a speedup in their execution in hardware. The speedup factors are approx. 40 for the encryption features, 2 for the Hamming code generator and 3 for the Hamming code verifier. Figure 2.17 shows the time to process eight million bytes of data for both the RTR-JVM and the unmodified Kaffe interpretive JVM. As can be seen, the average speedup is about 40, sometimes increasing to 50 if few packets are transmitted or being received.

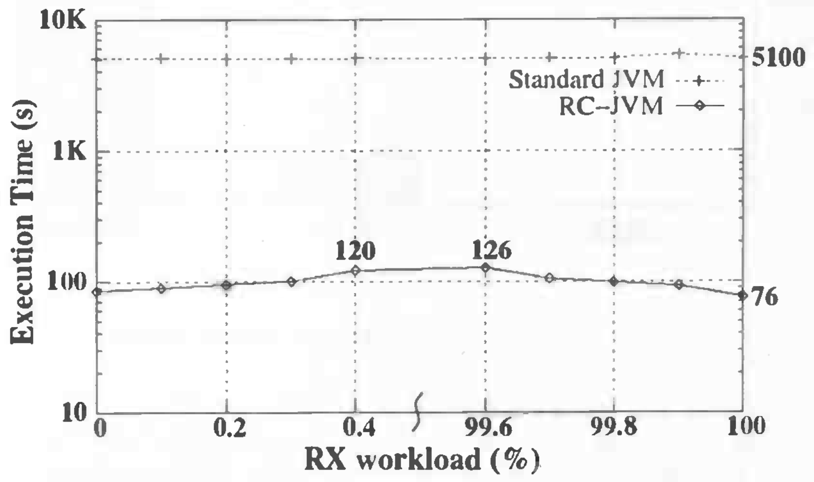


Figure 2.17: Network kernel benchmark execution times

Performance Limitations

In many cases, the limited communication bandwidth will negate the speedup. The connecting bus is clearly a major limitation in current designs, but new platforms might move the RC resources closer to the processor so this is not a great concern anymore.

Further work

In this section a number of oversimplifications in the RTR-JVM are redressed.

Runtime Feature Synthesis

Till this far, it has been assumed that the program features have been synthesized at compile-time. This approach is time-consuming and impairs portability. Run-time feature synthesis solves this problem, but synthesis tools need to improve significantly before runtime synthesis becomes practical.

Memory access from hardware features

Many features take the same input often. For these functions it might be better to write the into directly into the hardware instead of passing the input to the function.

Non-leaf features

For the RTR-JVM it has been assumed that classes implemented in hardware make no method calls outside of the class. Obviously, this is an oversimplification. Hardware features should be able to invoke other features residing in either hardware or software.

JIT compilation

Right now the RTR-JVM is based on the interpretive JVM which is slower than the JIT code translation. This will speed up the software execution, but decrease the speedup delivery by the RC hardware.

Partial reconfiguration

The current prototype only supports one feature per FPGA. Newer FPGAs allow to reprogram only a part of the FPGA, this allows for putting multiple features on a single FPGA.

Constant value propagation

By invoking a profiler callback for each feature invocation, a history of arguments passed to each feature could be maintained. Analysis might reveal that certain arguments remain constant. For such features it might be lucrative to write a specialized feature which executes faster.

Conclusions

An RC machine with runtime profiling capabilities can provide both ease of programming and considerable speedup. Many of the problems encountered during implementation have already been solved or are actively being researched. The integration of of CPUs and reconfigurable fabrics on a single chip will eliminate the communication barrier and open RC to a wide range of bandwidth-intensive applications.
Online algorithms are ideally suited to RC application because they can help maximize the use of limited hardware resources and eliminates the guesswork associated with static feature assignment. These advantages come at a very low cost; in the demonstration system, the overhead of the profiler is unmeasurable. With future work planned to develop a JIT-based machine and improved data transfer mechanism, the RTR-JVM may soon be able to accelerate a wide range of practical applications.

2.6 Conclusions

Having discussed these four papers, several conclusions can be drawn about them. The first paper has shown a good example about a compile-time reconfigurable application. The FPGA is configured just once and does not change functionality. One can imagine however that this application can very well be executed in an RTR-environment. Each Genetic Algorithm Island

can be thought of as a task, which in turn can be executed in either hardware or software. Another interesting point in this paper is that is proves that a slow FPGA can have a better performance than a fast workstation.
The second paper covers a model for an RTR-application. The problem with this model is that it hides behind a high level language which is defined to be able to generate both hardware and software version of a task. It is not possible to switch tasks in the middle of executing them. The RTR- application that will be developed during this thesis will overcome these limitations.
The third paper discusses a model for a RTR-firewall. It mostly concentrates on the hardware restrictions and the hardware interface between gasket and DHP module. It shows how powerful an RTR system can be for processing data and how easy it can be to add extra functionality using DHP modules. The gasket interface looks interesting, but in the end it was not needed in the final RTR-application developed in this thesis.
The fourth and last paper shows how to implement an RTR-system in a Java application where java classes are executed in either hardware or software. An algorithm to select which feature should be executed on the FPGA is presented along with the results of several communication protocols between the RTR-application and the FPGA. Again this RTR-application does not provide for switching tasks while the application is executing them, something that this thesis will discuss.

Chapter 3

System Model

This chapter concentrates on discussing a generic system model for a Runtime Reconfigurable application. To develop such a model, first a good look is given to the functionality which RTR-application needs to contain and which requirements exist on both the hardware and software part of the RTR-application. An important part of the RTR-application is the ability that allows it to switch the status of a task between hardware and software versions. A closer look will be taking at the switching ability and as well at associated problems.

3.1 What problems does the RTR-application need to solve?

What problem does the RTR-application need to solve? Take a situation in which several tasks need to be executed. Both hardware (FPGA) and software is available to execute these tasks. The optimal solution would be to execute all these tasks in parallel in hardware since in general tasks can be executed faster in hardware than in software because hardware has no operating system overhead or suffer from cache misses. However, hardware has the drawback it is more expensive than software and therefore only limited capacity is available. Herein lies the problem: how can optimal use be made of both hardware and software to complete the available tasks in the minimum amount of time? A scheduling algorithm can make the best choices between executing tasks in hardware or software. But the scheduling is not what this thesis concentrates on. Instead it concentrates on how to develop a runtime reconfigurable application in general, discussing the problems associated with the RTR-application. As a proof-of-concept a RTR-application is developed in this thesis.

But where does RTR or Runtime Reconfigurable stand for? On an FPGA there are two possible ways of reconfiguring:

- Static Reconfiguring. This means that after a FPGA is produced it is configured with a program and it will not change functionality again.
- Runtime Reconfiguring. While the application is running, the FPGA is changing functionality.

The Runtime Reconfiguring part is used to switch tasks from software to hardware and vice versa. This makes up for the following requirements:

The program should be able to write tasks to the FPGA and run these tasks in hardware on the FPGA. The task that is running on the FPGA can either be removed from the FPGA while it is running or when it is done executing. If the task is removed from the hardware while it was executing, then it should be possible to continue executing this task in software. If the task is removed when it is done executing, its calculation results are returned to the program and another task will be able to take its place inside the FPGA.

3.2 What are the software and hardware requirements?

It should be obvious that an FPGA is needed, since this is the part that will be able to execute tasks faster compared to software because the hardware version doesn't suffer any overhead from an operating system, does not suffer from cache misses or page faults and does not have complex memory addressing either. Basically just about any FPGA will do for a RTR application. The most important aspects about the FPGA are:

- The time required to reconfigure the FPGA should be as low as possible. How fast an FPGA can be reconfigured is one of the factors to determine how feasible it is to use in an RTR system.
- Some FPGAs (like the Xilinx Virtex) can be divided into parts which makes it possible to reconfigure just one part, instead of the complete FPGA. This speeds up reconfiguration time as well.
- It should be possible to obtain the status of the FPGA (the values of its flipflops). Obtaining the status is important since when the status of the hardware function is known, it is possible to put this status into the software version of the same function to allow it to continue execution.

Usually an FPGA resides on a board which can be connected to, for example, a PC or other hardware device by a parallel cable or serial bus. Therefore a communication interface is required between hardware and software.

As mentioned before, the RTR application which is run in software which needs to communicate with the FPGA. Several possibilities to communicate with the FPGA exist. The most standard approach is to have the software part of an RTR application to run on a PC, while it communicates with the FPGA via a parallel cable or via a PCI card in which the FPGA is plugged into.
The software will need to be able to communicate with the FPGA. How this communication takes place largely depends on way the FPGA is connected with the software (parallel cable or PCI bus).

3.3 How does the global model look like?

Figure 3.1 gives an overview of the global model designed. There is a clear difference between the hardware (FPGA) and the software (MainApplication) side. The MainApplication is divided into two parts: UserProgram and ReconfigInterface.

ReconfigInterface is a class which contains a pool of re-usable functions which provide easy access to the FPGA (for example putting tasks onto the FPGA and removing them). The other part (called *UserProgram*) consists of code for the actual application: it takes care of executing the tasks, switching tasks back to software or hardware. Basically this part contains the program that needs to be executed, whereas the other part provides for easy means to execute the program. The main benefit of this structure is that if someone wants to write a new application, he can reuse the *ReconfigInterface* and only write a new *UserProgram*. As well as if another FPGA is used, only the *ReconfigInterface* needs to be adjusted, while the *UserProgram* can stay the same. In the model a task is characterized by three parts:

- Software version. This part is the software version of the task.
- Hardware version. This is a hardware version of the task. Most likely (if communication between UserProgram and the FPGA is needed) it is also accompanied by a class file that takes care of the communication between the UserProgram and the FPGA.

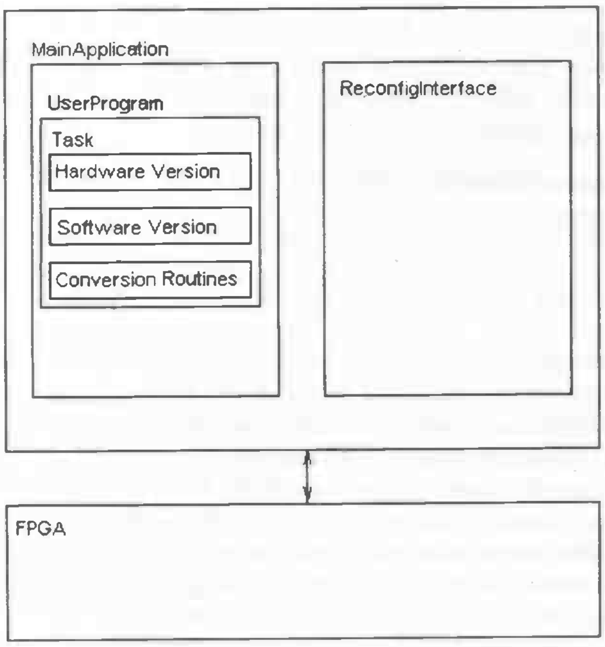


Figure 3.1: Global Model RTR Application

- Conversion routines between the tasks' Hardware and Software status. If the software version of a task has been running, has been stopped and needs to be continued in hardware, then the status of the software version has to be put into the hardware version. Vice versa for hardware to software

The most difficult part here are the conversion routines between hardware and software status. Special care must be taken to make sure that every variable in the software version can be put into the hardware version as well. The next few sections will discuss this in greater detail.

3.4 Switching between hardware and software

As an example on how the switching from hardware to software occurs, this section will discuss how the switching technique is applied to a 4-bit CRC counter. The UserProgram written is a 4-bit CRC counter of which the algorithm is as follows:

$(\sum_{i=0}^{length(array)} array[i]) \ mod \ 2^4$

The CRC algorithm is typically used for integrity-tests. The reason for using this algorithm is because it is easy to write and it has a clearly definable state. In the UserProgram two versions of this CRC algorithm exist: one in software and another one in hardware. The software variant can be represented as follows:

```
int result = 0;
for (i=index;i<CRCarray.length;i++) {
  result = (result + CRCarray[i]) % 16;
  index++;
}
```

In this piece of code, the CRC value of the elements in the array *CRCarray* is calculated. The status can be represented by the *result* and the *index* variables. The *result* variable indicates the

CRC value calculated at the index *index* in the *CRCarray*.
¿From this piece of code it is possible to derive a hardware version of the algorithm. Since the additions are the heaviest work this algorithm has to execute, let this part to be executed in hardware. The following algorithm can then be derived:

```
for (i=index;i<CRCarray.length;i++) {
  write(FPGA, CRCarray[i]);
  index++;
}
Read(FPGA, result)
```

In this algorithm a value from the *CRCarray* at index i is sent to the FPGA. The FPGA has to add the value it receives to the previously sent values. This continues until all the values in the array are sent to the FPGA and added. In the final step the value is read from the FPGA to transfer the resulting value back to the UserProgram.
As can be concluded from this piece of code: the FPGA is only responsible for adding the values it is receiving. A communication thread needs to exist to take care for sending input-values to the FPGA. Basically this communication thread consists of the *for* loop mentioned in the piece of code above and the write and read statement to read and write the values to the FPGA.
A hardware version of a task can be described as a program run in hardware plus a communication thread run in software. The description of above piece of code is not complete without the actual hardware program as well. This hardware program is represented in figure figure 3.2.

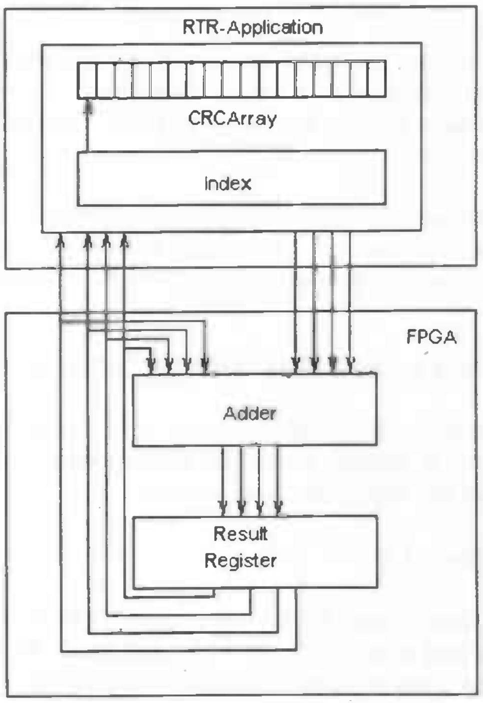


Figure 3.2: Hardware version CRC

The hardware CRC version receives its input values from the RTR-application. These values are transferred to the FPGA by for example a parallel port. The input-value is added to the value stored in the register, after which it can add another input value. Since this is a fairly basic calculation, adding one value can be done in one clockcycle. The status of the hardware version can be defined by the value of the register and the *index* variable on the RTR application.
To be able to switch executing tasks from hardware to software and vice versa it it necessary to be able to switch the status from software to hardware and back. A good conversion function for

the CRC example mentioned would be able to put the software *result* variable into the hardware register and the software *index* variable into the communication thread of the hardware version. This is all that needs to be taken care of to switch the status between hardware and software.

3.5 Switching status in large applications

In the previous section an example was given on how to change the status for a small algorithm. Though when big applications are written, the tasks will not always be as small as the CRC algorithm. This section will discuss which problems will occur when the status has to transferred between large software and hardware versions of a task.

When tasks get larger the more difficult it is likely to be to switch the status. Think of for example a Fourier transformation algorithm. The first problem occurs when such an algorithm has to be stopped in software so it can (for example) continue running on the FPGA in hardware. It is possible to derive a usable status from the Fourier algorithm when it has finished 25% of it's calculation? Most likely not. A solution to this would be to raise a *stop_algorithm* flag which will be raised when the algorithm needs to stop executing. The algorithm will only check for this flag when it is in a clearly definable state which can be transferred to hardware. In pseudocode it looks like this:

```
function FourierTransformation() {

  calculateTill25Percent();
  CheckForStopFlag();
  calculateTill75Percent();
  CheckForStopFlag();
  CalculateTillAlgorithmDone();

  ReturnResultValues();
}
```

Using this technique it is possible to stop the algorithm in a sensible state. This example was aimed at a software version of an algorithm. But it can apply just as well to a hardware version where an inputIOB has been set to a high to indicate the hardware version of the algorithm needs to be stopped.

Another more difficult problem is to actually find a clearly definable state that can be transferred between hardware and software versions where the *CheckForStopFlag* function calls can be put. In this thesis separate hardware and software versions of algorithms are developed so that both are optimized for performance. The nature of both hardware and software algorithms are very different from each other so that it might not be possible to find a variable that occurs in both hardware and software version. To make sure the status is transferrable several solutions exist. The first solution would be to use a language which can generate both hardware and software versions of an algorithm which have a transferrable status. The problem is developing such language which might be a nice project for another thesis. In this thesis special care is being taken so that the status is transferrable. This is the responsibility of the programmer who is writing both hardware and software version of a task.

3.6 Summary

In order to get the best performance out of an FPGA, the reconfiguration time should be kept as low as possible. Some FPGAs only allow to reconfigure a part instead of the whole which further lowers reconfiguration time. To be able to transfer a function between software and hardware, it should be able to determine its status. On the software side not many requirements exist, except for the FPGA to be able to communicate with the FPGA through an interface. A global model of an RTR-application was discussed and a description about the CRC algorithm used in the CRC RTR application has been given. Special care should be taken so that the status of a software version of a task can be put into a hardware version of that same task and vice versa. This will be increasingly difficult as larger tasks are developed.

Chapter 4

Using the Virtex FPGA: JBits

In the previous chapter a global model was discussed along with the requirements of software and hardware versions of tasks. This chapter will make a start at the development of an RTR-Application by discussing its most important tool used: JBits v2.8. JBits is a Software Developers Kit (SDK) which contains a set of APIs to create and modify Xilinx FPGA bitsteams from java code. This chapter describes JBits in closer detail, covering its components, its terminology and how to use it in programming.

4.1 Overview Components JBits

The JBits SDK features the following main components:

- XHWIF (Xilinx Hardware Interface) This interface describes the characteristics for a Virtex device. This allows tools like BoardScope to use any Virtex of which a XHWIF interface is defined without any adjustments. The XHWIF describes the layout of an FPGA and includes methods to read from and write to the FPGA. The main advantage of this layer is that a software program will be able to run on every hardware device which has a XHWIF defined.

- XVPI (Xilinx Virtex Portable Interface) This interface is used for reconfiguring the boards.

- JRoute JRoute is a set of java classes which take care of the routing within an FPGA. It allows for connecting routing resources to each other.

- VirtexDS (Virtex Device Simulator) This is a simulator which provides users to test their applications without the need of the actual hardware. The Simulator can be used in combination with BoardScope.

- BoardScope BoardScope is a graphical debugger for Virtex devices. It allows the user to see the internal state (e.g. LUTs) and output signals of CLB as well as the routing, routing density and a power graph. The power graph shows how much power a certain CLB is using at a certain time.

4.2 Setting up a skeleton JBits program

To start with programming in JBits, a small example skeleton program is presented that writes data to the FPGA and reads the same data back from the FPGA.

4.2.1 Setting up communication with the FPGA

The most important part of a JBits program is the JBits object. The JBits object is a bitstream which represents a hardware configuration of an FPGA: the configuration of the CLBs/LUTs/IOBs and the routing configuration.
To write the JBits object to the FPGA, the XHWIF communication interface is needed. This interface takes care of all reading and writing to the FPGA. The XHWIF interface of a board is retrieved in the following way:

```
String boardName ="VirtexDS:xcv50";
XHWIF board =XHWIF.Get(boardName);
```

This example will run a Virtex Device Simulator for the xvc50 FPGA as indicated in the lines above. In case one actually owns the hardware FPGA, *boardName* can be $XCV50$, but if you lack the hardware and want to use the simulator, then *boardName* should have $VirtexDS$: in front of the actual devicename.
Next thing to do it to make the XHWIF interface connect to the VirtexDS/Hardware. The XHWIF interface supports connecting to hardware remotely placed on the network. This feature will not be used in this thesis. Nevertheless it is necessary to connect to the interface as if it were remotely placed in a network.

```
  /* Get the remote host name */
  String remoteHostName = XHWIF.GetRemoteHostName(boardName);

  /* Get the remote port number */
  int port = XHWIF.GetPort(boardName);

  /* Connect to the hardware */
  int result = board.connect(remoteHostName, port);
```

This will establish a connection to the FPGA device.

4.2.2 Modifying the bitstream and writing to the FPGA

Modifying the configuration of the FPGA takes several steps. First the JBits object must be initialized, then configured and finally written to the FPGA.

It starts with retrieving the JBits object. To retrieve it, pass the *deviceType* to the JBits constructor:

```
deviceType = Devices.XCV300;
JBits jbits = new JBits(deviceType);
```

After creating the JBits object it should be initialized with the NULL-bitstream. This means all CLBs/IOBs/routing will be reset from the bitstream. The JBits package comes with several null-bitstreams. The null bitstream for the XVC50 is provided with the JBits SDK and is called $null50GCLK1.bit$.

```
String inputFileName = "null50GCK1.bit";
jbits.read(inputFileName);
```

Having created a clean bitstream, it is possible to make some changes to the *jbits* bitstream, for example:

```
  int row=0;int col=0;
  int[] LUTVALUE = Util.InvertIntArray(new int[] {0,0,0,0, 0,0,0,0, 1,1,1,1, 1,1,1,1});
  jbits.set(row, col, LUT.SLICE0_F, LUTVALUE);
  jbits.set(row, col, S1Control.Y.Y, S1Control.Y.GOUT);
```

In this small example the LUT of slice 0 of the CLB at location (0,0) will be given the value $\{0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1\}$. As one can see this value is inverted before being written using *jbits.set* to the bitstream. This is an obligatory thing: the value loaded into the LUT should always be the inverse of what you want. This inverse can be obtained by calling *Util.InvertIntArray*.
The second call the *jbits.set* sets the C-MUX, slice 1 of the CLB at (0,0) (as can be seen in figure 1.3) to put the input from the LUT through as output.

After modifying the *jbits* object it is time to write this bitstream to the Virtex device. This is also referred to as reconfiguring the board or just plain *reconfig*.

```
public void reconfig() {
   byte[] configBitstream = jbits.getPartial();
   if (configBitstream == null) {
     System.out.println("The bitstream did not change, so not reconfigured");
   } else {
     System.out.println("Length configBitstream: " + configBitstream.length);
     board.setConfiguration(0, configBitstream);
   }
}
```

JBits will determine the changes made between the last configuration sent to the device and the present configuration in the *JBits* Object using the *getPartial*() method. From this difference it creates a series of configuration-packets as output which are send to the FPGA using the *setConfiguration* method.

4.2.3 Reading back the bitstream from the FPGA

To check if the reconfig was successful one can do a readback. A readback is to read the complete configuration of CLBs/IOBs/routing/etc back into the JBits object. It should be done this way:

```
byte[] readbackCommand = ReadbackCommand.getClbConfig( deviceType );
board.setConfiguration( 0, readbackCommand );
byte[] readbackData = board.getConfiguration( 0, ReadbackCommand.getReadLength()*4);
jbits.parsePartial( readbackCommand, readbackData );
```

First a readbackCommand is created for this specific deviceType, then the command is sent to the board, which will result in a lot of readbackData. This data is then to be parsed back into a JBits object using the parsePartial method.
To be sure the *LUTVALUE* was set correctly and that it still has the value it was set to before it was retrieved from the JBits object:

```
int[] lutvalue =
  Util.InvertIntArray(jbits.get(row,col,LUT.SLICE0_F));
```

4.3 Writing and debugging a counter program

In section 1.2 a hardware layout of a small counter program was provided. This section will cover on how to configure the FPGA to implement this counter. To make sure the counter is running perfectly, the results are checked using the BoardScope debugger. The BoardScope debugger is very clear laid out debugger which shows all the information that one needs like the internal state of a CLB and its routing. Apart from that it has a few extra features like showing a power graph of how much power a CLB consumes and it features a routing density viewer as well.

To get the counter counting, the actions we need to take according to section 1.2 are:

- set the G-LUT to *NOT G*1
- set the F-LUT to *NOT*(*G*1 *XOR G*2)
- set C, H and Q-MUX to output the top-input
- set the P-MUX to output the middle input
- set both flipflops to act as flipflops and not as latches
- connect YQ to G1
- connect XQ to F1
- connect Y to F2
- connect the clock to the slice for the flipflop

Connecting Pins can be done using the JRoute interface easily by calling:

```
JRoute jroute = new JRoute(jbits);
jroute.route(new Pin(Pin.CLB, row,col, CenterWires.S1_YQ),
             new Pin(Pin.CLB, row,col, CenterWires.S1_G1));
```

The JRoute interface however, has been superseded by the JRoute2 interface. The main difference is that JRoute2 uses a different technique to do the routing and that it now supports IOB/BRAM routing. The JRoute2 interface does bring a lot of extra work for connecting two simple pins; instead of being able to connect two pins with one JBits command, one now needs about four. For now JRoute will be used in this example to keep things simple.

To enable the clock for a certain CLB at (*row*, *col*), the following call can be used:

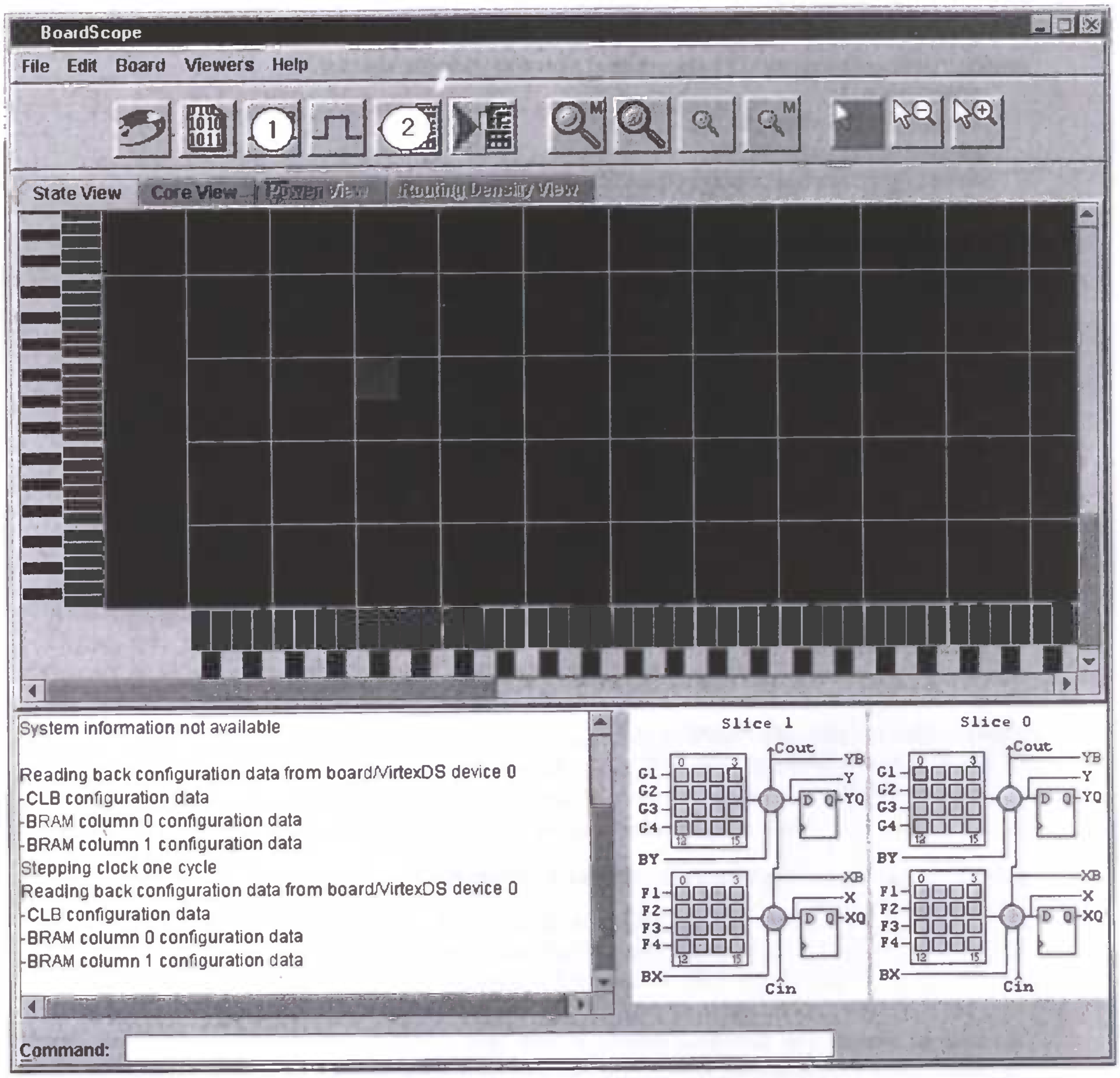
```
Bitstream.set(row,col,S1Clk.S1Clk,S1Clk.GCLK1);
```

This will cause Slice 1 of CLB(row,col) to connect to GCLK1. The Virtex has one central clock, but four points that one can use to connect to the clock, these are GCLK0, GCKL1, GCLK2 and GCLK3. Usually GCLK1 is used.

The next objective is to include BoardScope for debugging purposes. To get this working, a different way to connect to the board is needed. This time a XHWIFWithEvents interface instead of the usual XHWIF interface is required. The XHWIFWithEvents allows a few more features and BoardScope is one of them.

```
XHWIFConnection xc = new XHWIFConnection();
board = xc.getXHWIFWithEvents();
BoardScope boardscope = new BoardScope(xc);
boardscope.show();
xc.connectToBoard( boardName );
```

When running this program, BoardScope will pop up. Figure 4.1 shows what BoardScope looks like after passing one clockcycle. The middle-part of BoardScope shows the layout of the FPGA and its state. When the XQ or YQ pin outputs a high, the color of that part of the CLB will show up green. To do a clockcycle press button 1. This will do a clockcycle and update the view. By just pressing button 2, BoardScope will only update the view without doing a clockcycle. In the lower-right corner is shows the internal details of the CLB. By clicking on one of its inputs or outputs (for example F1,F2,F3,F4, BY,YQ) it will return its value (1 or 0) and the wires that it is connected to.



System information not available

Reading back configuration data from board/VirtexDS device 0
-CLB configuration data
-BRAM column 0 configuration data
-BRAM column 1 configuration data
Stepping clock one cycle
Reading back configuration data from board/VirtexDS device 0
-CLB configuration data
-BRAM column 0 configuration data
-BRAM column 1 configuration data

Figure 4.1: Screenshot of boardscope

4.4 Cores and routing

The problem with just setting LUTs and connecting CLBs like in the previous example is that there is no hierarchy, no architecture. To solve this problem, Cores were introduced into JBits. Cores, also known as RTPCores (Run-Time Parameterizable Cores) provide means to wrap a function into a package and to provide the user with ready-made functions. JBits comes with several Cores. A few examples of these cores are: Adders, Subtractors, Constants, Multiplier, Register and Decoder.
But what do these cores actually look like? Below one can find a template for a core.

```
import com.xilinx.JBits.CoreTemplate.*;

public class MyCoreTemplate extends RTPCore {
```

```
public MyCoreTemplate(String name) throws CoreException {
   super(name);
   setHeight(calcHeight());
   setWidth(calcWidth());
   setHeightGran(calcHeightGran());
   setWidthGran(calcWidthGran());
} /* end constructor */

public static int calcHeightGran() {
// return the granualty of the core
   return Gran.CLB;
   }

public static int calcWidthGran() {
// return the granualty of the core
   return Gran.CLB;
}

public static int calcHeight() {
// return the height of the core
   return height;
}

public static int calcWidth() {
// return the width of the core
   return width;
}

public final void implement() throws CoreException {
// implement core
}
```

As can be seen in the template above, a core only needs to implement a few basic functions to define the width and height of the core. The *calcWidthGran*() and *calcHeightGran*() functions are used to indicate if the width and height are measured in either CLBs, Slices or Logic Elements.
The main function of the core is the *implement*() function. This function should take care of setting all the appropriate MUXs, LUTs and routing within the core itself. Using this template it is possible to produce any kind of stand-alone core.
But that is not enough, cores should be able to communicate with each other as well. There are several structures for communication:

- Port equates a signal (net or bus) that is external to a core with a signal that is internal to the core. Ports have a direction and width, where the port width is inferred by the width of the external signal, the width of the internal signal, or the number of internal pins. A WidthMismatchException is thrown if these width indicators disagree.

- Pin associates one or more physical resources with a port. Primitive cores define internal pins for their ports, rather than internal signals. An example of a pin is a LUT input.

- Net is a named set of source and sink ports within the scope of a single RTPCore.

- Bus is an indexed multiset of nets that are within the scope of a single RTPCore.

Figure 4.2 below gives a graphical overview of this structure.

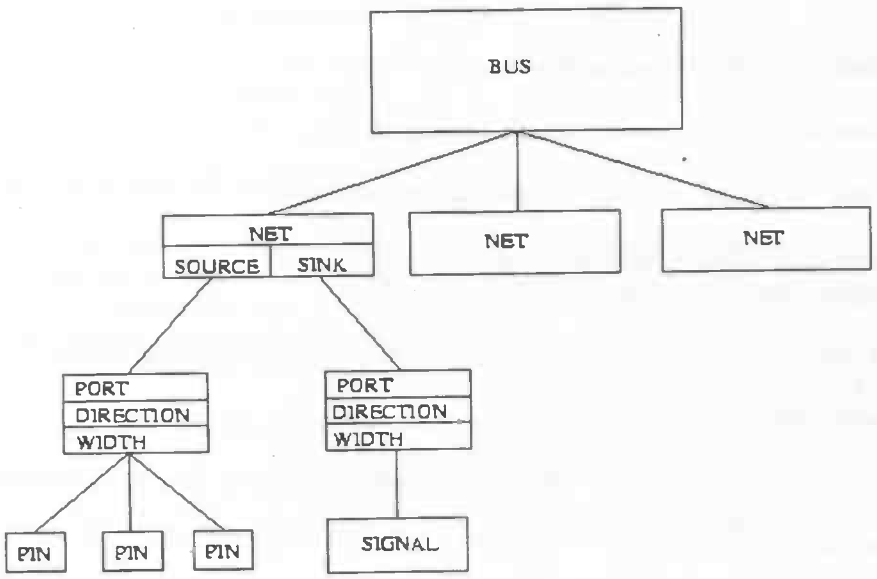


Figure 4.2: Overview routing structures

A bus contains several nets, each of these nets has a source and a sink port defined. Each port has a width and a direction property. The width property indicates the number of pins the port contains and the direction indicated whether the Port should give its value as an input or an output to the core. A port can either contain an array of pins or it is equal to a signal. A signal is equal to a Bus or Net.

To show how this should be used in JBits a small example is presented in which a Core (called the MainCore) has two subcores (SubCore1 and SubCore2). SubCore1 should have a bus that outputs to SubCore2 and SubCore2 should have a bus as well that outputs to SubCore1 as presented in figure 4.3.

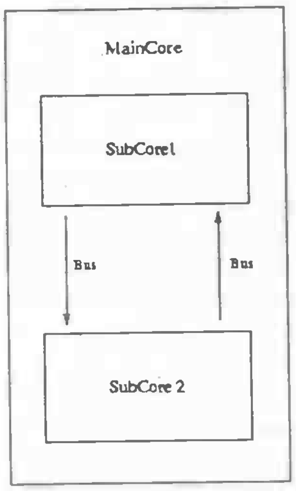


Figure 4.3: Structure example program

The JBits code for implementing this communication Busses can be done in the following way (only code related to routing the Busses is shown):

```
public class MainCore extends RTPCore {

public MainCore(String name) throws CoreException { }

public final void implement() throws CoreException {

  Bus fromSubCore1To2 = newBus("from1to2",4);
  Bus fromSubCore2To1 = newBus("from2to1",4);

  SubCore sub1 = new SubCore("SubCore1",fromSubCore2To1,fromSubCore1To2);
  addChild(sub1,Place.LOWER_LEFT);
  sub1.implement();

  SubCore sub1 = new SubCore("SubCore2",fromSubCore2To1,fromSubCore1To2);
  addChild(sub1,Place.LOWER_LEFT);
  sub1.implement();

  Bitstream.connect(fromSubCore2To1);
  Bitstream.connect(fromSubCore1To2);
 }
}

public class SubCore extends RTPCore {

private Port inport; private Port outport;

public SubCore(String name, Bus inputBus, Bus outputBus) throws
CoreException {
      inport = newInputPort("inport", inputBus);
      outport = newOutputPort("outport", outputBus);
}

public final void implement() throws CoreException {
  inport.setPin(3,new Pin(Pin.CLB, row,col, CenterWires.S1_G1));
  inport.setPin(2,new Pin(Pin.CLB, row,col, CenterWires.S1_G2));
  inport.setPin(1,new Pin(Pin.CLB, row,col, CenterWires.S1_G3));
  inport.setPin(0,new Pin(Pin.CLB, row,col, CenterWires.S1_G4));

  outport.setPin(3,new Pin(Pin.CLB, row,col, CenterWires.S1_Y));
  outport.setPin(2,new Pin(Pin.CLB, row,col, CenterWires.S1_X));
  outport.setPin(1,new Pin(Pin.CLB, row,col, CenterWires.S1_YB));
  outport.setPin(0,new Pin(Pin.CLB, row,col, CenterWires.S1_XB));
 }
}
```

As shown in the example, the necessary code involves:

- Create the bus in MainCore which encapsulates the subcores using newBus
- The busses should be passed as parameters to the subcores

- The subcores should get a Port from these busses using either newInputPort or newOutput-Port
- In the *implement*() of the subcores, Pins should be assigned to the Port
- Finally the command to actually make the routing connections should be made in the Main-Core using *Bitstream.connect*(*Bus*).

4.5 The status of a Core

One of the requirements of the RTR application is that it can start running a core on the FPGA, stop this core, remove it from the FPGA and continue running it at a later point in time. To be able to do this, the status of a core needs to be defined; identify which flipflops indicate which variables in the software version of the core.
The status of a core located on the FPGA can be divided into two parts:

- *Physical part* This part consist of all routing-connections, MUX and LUT settings.
- *Volatile part* This part consists of all the flipflops and BRAM.

In this thesis the BRAM part of the FPGA is not used, so that only leaves flipflops as the volatile part.

There are several possibilities to read the status of a core and store it. It is possible to read the physical part of a core and store all the settings in a data-structure (e.g. array). The negative aspect of storing the physical part this way is that information about a core is thrown away. A core usually contains a structure consisting of subcores and identification names, this extra information is thrown away when only storing the wire/MUX/LUT information. Another, cleaner, approach is to make use of the java core file already created. The core file already contains all information needed about the physical part. Every time the physical part of a core is needed again it suffices to load the java core file onto the FPGA again.

Putting back the volatile state (flipflops) back into a core is more difficult. JBits itself cannot set the value of flipflops during the configuration of the FPGA. A work-around is needed to do this. The RTR-application sets the flipflops in the following way: first the LUTs of the flipflops that need to be set are configured to output a high. Next the MUXs are set so that the output of the LUTs cause the flipflops to be set the next clockcycle. After this clockcycle the flipflops have their original state back. To completely restore a core back to its original state, one can now put the physical part back as well; the core will continue running from its previous state.

4.6 Summary

This chapter has shown how to use JBits by covering how one can set up a skeleton JBits program which is able to connect to the FPGA, put a program on the FPGA and read and write to the bitstream. How to debug an example program was shown as well as important datastructures like Cores and Routingstructures. A final part of this section covered how the status of a core can be read from the FPGA and put back onto it.

Chapter 5

Developing the RTR application

This chapter covers the actual development of a runtime reconfigurable application. The application that has been chosen for this thesis is a RTR application that calculates 4-bit CRC values out of an array. This chapter will cover the development process from looking at the functionality required again, filling in the details in the global model for an RTR-application as discussed in chapter 3 which are the ReconfigInterface, CoreStorage, CoreList, WireManager, ReconfigInterface internal functions, UserProgram, CommandLine and the switching between hardware and software status specific to the CRC Application. A few scenarios of how the CRC application works will be presented along with final conclusions and future recommendations.

5.1 Functionality

The functionality for a generic RTR-application was mentioned in chapter 3. The application written does limit itself a bit in comparison to the generic model by limiting itself to only two functions that may be used in the application. In short the functionality of the CRC RTR-application will be as follows:

- Write two functions to the board using cores
- Run these functions on the FPGA until either
  - A core is done calculating, it should let the software know it is done and transfer the results
  - A core is not done calculating, but the user decides it needs the space on the FPGA for another core to put in its place, the possibilities are:
    - Stop the core, store it in software and wait till there is enough space to put the core on the FPGA or
    - Stop the core and continue executing it in software

To be able to provide this functionality, the application needs a structure. In chapter 3, a global model was described in figure 3.1. The model specific for the CRC RTR application is depicted in figure 5.1

In the current model, the division between UserProgram and ReconfigInterface is maintained. In the following sections the functionality of the ReconfigInterface will be covered and next the UserProgram of the RTR application will be discussed.

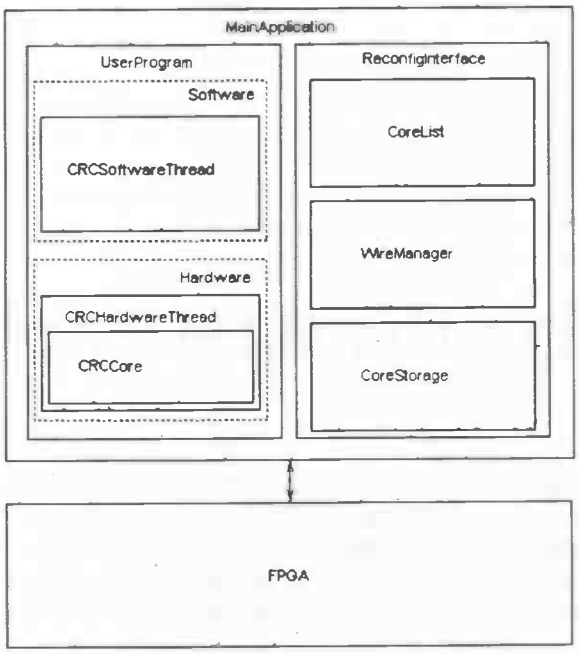


Figure 5.1: Detailed model of the CRC RTR Application

5.2 ReconfigInterface

The purpose of the ReconfigInterface is to provide for easy access to the FPGA and a class to store any functions that will be re-usable in future projects. The internal classes developed for this Interface are:

- CoreStorage
- CoreList
- WireManager

These classes will be discussed in the following subsections.

5.2.1 CoreStorage

Eventually a core will be removed from the FPGA and it will have to be stored into the software-memory. For this purpose the CoreStorage class has been developed. CoreStorage will read the entire status of a core into a data-structure. This status contains extra information like position of the core in the FPGA and the name of the core and the className of the core. The CoreStorage class looks like this:

```
public class CoreStorage {

  int[][] settings;          // array used to store the values of the
                             // flipflops in for this core
  String coreClassName = ""; // classname of the core
  String coreName = "";      // name of the core


public void nullCore(JBits jbits)
// This core resets both MUXs, LUTs and routing of the core
public void nullRouting(JBits jbits, int row, int col)
// This method removes all routing of the CLB at location (row, col)
// in the JBits object

public void nullCLB(JBits jbits, int row, int col)
// This method resets all MUXs and LUTs of the CLB at location (row, col)
// in the JBits object

public void getSettings(JBits jbits)
// This method retrieves the status of the flipflops and stores this
// status in the settings[][] array.

public void setStatus(JBits jbits, int Prow, int Pcol)
// This method sets the status stored in the settings array back into the
// JBits object at location (Prow, Pcol).

public CoreStorage(RTPCore core)
// This is the CoreStorage constructor. It takes care of storing the
// class, core, height, width and location of the core.
}
```

To make use of this class, one should call the CoreStorage constructor with as argument an implemented Core on the FPGA. In the constructor the classname of the Core, its size and location will be retrieved and stored in the object. By calling the *getSettings* method, with as argument a JBits object, the object will read the volatile status of the flipflops from the bitstream and store these in the *int*[][] *settings* array. To remove the current core from the bitstream one calls nullCore to remove the MUX and LUT settings and the internal routing to the core. The Core now has been completely removed from the JBits object. The FPGA can now be used to run another core or do something else.
If the stored core needs to be put back onto the FPGA to resume executing it, the first thing to do will be to put back the status of the flipflops using the *setSettings* method. Next a *reconfig*() to write the bitstream to the board and a clockstep is necessary to let the flipflops assume its appropriate value. The only thing left is to put the hardware layout of the core back which can be done by calling the Cores class file again (calling the class with the name className which is stored in the CoreStorage object).

5.2.2 CoreList

Since multiple cores will be used, running in either software or hardware, it is preferable to include a datastructure for managing cores. This is the sole purpose of this class. In this class all the properties of a core are defined, these are:

- *coreName* The name of the core
- *coreClass* The class the core belongs to
- *rtpcore* The RTPCore class associated with this core
- *coreStorage* If the core has been removed from the FPGA, this is where it is stored.
- *inbus* The InputBus associated with this core
- *outbus* The OutputBus associated with this core
- *widthInputBus* Number of pins in the inbus
- *widthOutputBus* Number of pins in the outbus

5.2.3 WireManager

Cores need communication with the outside world (e.g. software) to exchange data. The first idea was to split the FPGA in several parts: a part (called *communication core*) that would always stay in the FPGA to take care of the communication needs of running cores and a part for the storage of (running) cores. Whenever a core is added to the FPGA it should be connected to the communication core. This communication core would, being connected to IOB pins, take care of sending the cores signals to the software program using the IOB pins. This approach is especially useful when there are limited IOB pins available, when the output of several cores needs to go through one single IOB bus.
In the current situation using the Virtex FPGA however, plenty of IOB pins are available on the FPGA, more than will be used by the RTR application, therefore another approach was chosen. In the current approach every core can dynamically request IOB pins.
To provide this functionality, the WireManager interface was developed. This interface handles all IOB related tasks. Using the WireManager it is possible for a core to simply request an IOB Bus using the $getInputBus(int\ width)$ or $getOutputBus(int\ width)$ method. This will return a bus suitable for either incoming or outgoing signals from the Core. The WireManager interface keeps an internal list of which IOB pins of the FPGA have been allocated which have not. Every time a bus is requested, it will return a bus with a certain name. In the event that the core needs to be removed from the FPGA, one can remove the bus from the FPGA by calling $removeBus(JRoute\ jroute, Bus\ rbus)$.
Another IOB related task the WireManager is performing is to keep track of the values of the the IOB pins. This functionality was added because the JBits ClientSimulator class (which is needed to retrieve the values of IOB pins) did not prove to be thread-safe. If multiple threads try to retrieve the value of a pin, the ClientSimulator would crash. To prevent such problems, the WireManager contains an array in which it keeps track of the values of the IOB pins. An internal thread is continually updating these values. If a part of the program wishes to retrieve the value of an IOB pin, it can retrieve the latest value from the WireManager internal array.

5.2.4 ReconfigInterface internal functions

The ReconfigInterface also contains a number of internal functions. Below a pseudocode overview of the ReconfigInterface class is presented.

```
public class ReconfigInterface

CoreList[] coreList = new CoreList[5];
 // reserve space for 5 cores in memory

public void InitializeFPGA() {
 // initialize the FPGA

public int probeBusValue(Bus bus)
 // return the value on the Bus bus

public void loadBitstream(String inputFileName)
 // load a bitfile into the jbits object

public void startUpVirtexDS()
 // start up the virtex, part of the initialization process

public void steps(int s)
 // do s clockcycles in the simulator

public void reconfig()
```

```
 // reconfigure the fpga: write the jbits object to the FPGA

public Bus[] AddCore(String classCore, String coreName, int
widthInputBus, int widthOutputBus,int x, int y)
 // Add a core with the class classCore and name coreName with
 // the width of the inputbus equal to widthInputBus and the width of
 // the outputbus equal to widthOutputBus and pass an input and
 // output bus along as well at location (x,y) of te jbits object

public Bus[] RestoreCore(String coreName, int x, int y)
 // Put the core with coreName back in the jbits object at
 // location (x,y). Request the input and outputbus from the
 // WireManager and update the CoreList structure.

public void RemoveCore(String coreName)
 // Remove a core from the jbits object and put it in a
 // CoreStorage object. Also fill/update the CoreList item

private int findCoreIndex(String coreName)
 // Finds the index of the core with coreName in the CoreList

public void Disconnect()
 // Disconnect from the board

public void doReset() throws Exception
 // reset the board

public void Step()
 // does a clockstep in the simulator

public void readBack() throws Exception
  // readback data

}
```

As can be seen in the pseudocode, the internal functions provide for easy adding, removing and restoring (to put a core that was removed from the FPGA, back onto the FPGA) a core along with functions to initialize and reconfigure the FPGA.

5.3 UserProgram

This section will explain about the actual CRC application developed to make use of the ReconfigInterface that switches cores onto and from the FPGA.

The UserProgram developed in the CRC RTR Application is a 4-bit CRC counter. In the UserProgram two arrays filled with 4-bit values are created and for these arrays the CRC value (result of 4-bit addition) needs to be calculated. The reason for choosing this algorithm is because it will involve IOB communication, can be run in multiple instances on the FPGA and is relatively easy to write. Since I want to have full testing capabilities, I decided on a commandline-driven interface. This allows for easy testing different combinations of switching cores from and to software/hardware. In this commandline the user is able to give commands on what the program needs to execute (e.g. remove a core from hardware etc). The commands the user can give will be covered later in greater detail.

In short, the UserProgram can be spit up into several parts. First the UserProgram itself is a commandline parser. It takes care of executing the commands it is fed. Secondly the UserProgram can be divided in a software and a hardware part. The software part consist of a class which executes the CRC algorithm in software. The hardware part consists of a JBits core file which is used to put the hardware representation of the algorithm onto the FPGA and of a class which takes care of the communication between the UserProgram and the Core on the FPGA. Figure 5.2 represents this structure. The following sections will cover these parts in more detail.

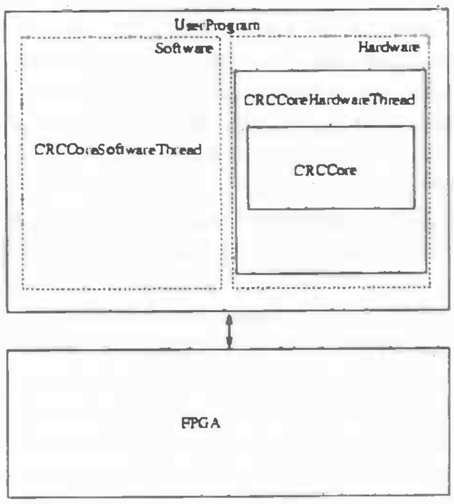


Figure 5.2: Schematic representation UserProgram

5.3.1 CRC: Switching between software and hardware

In chapter 3 a global description was given on how the switching between executing a task in hardware and software takes place. The CRC algorithm, the software version of the CRC algorithm and the hardware version were discussed. This section will look at how this is implemented in the CRC RTR Application.

Software

The software version implemented is exactly the same as the version mentioned in chapter 3.

```
for (i=index;((i<CRCarray.length) and (StopFlag == false));i++) {
  result = (result + CRCarray[i]) % 16;
  index++;
}
```

The variable *index* indicates the index in the *CRCarray* where it needs to start counting. This small algorithm was made to run in a looping thread called the CRCSoftwareThread until either it is done calculating or until the UserProgram sets the *StopFlag* variable to *true* which would make the Thread stop as well. The thread will send the index in the array and the result calculated this far to the UserProgram.

Hardware

The hardware variant of the CRC algorithm is more difficult to write because communication between software and hardware needs to be taken into account as well. The hardware algorithm in chapter 3 can be detailed a bit more for the CRC Application:

```
for (i=index;((i<CRCarray.length) and (StopFlag == false));i++) {
  write(FPGA, CRCarray[i]);
  index++;
}
Read(FPGA, result)
```

In the implementation of this code, the CRCHardwareThread covers sending inputvalues to the FPGA until either it is done calculating or *StopFlag* is set to *true*. When the hardware algorithm needs to stop, no more inputvalues are fed to the hardwarecore. This way the hardwarecore will end up in a clearly definable state which can be transferred to software. The detailed tasks of the CRCHardwareThread which has been implemented are as follows:

- If the hardware core outputs a 0, then set a new 4-bit values onto the cores input IOBs and set the Control Pin to 1 and wait till the input IOB changes.
- If the hardware core outputs a 1, then set the Control Pin to 0, this will cause the core to add the current value on the input IOBs to the current calculated value and wait till the input IOB changes.
- If this CRCHardwareThread needs to stop calculating, then finish the current calculation calculation-cycle and return the index in the array and the current result value to the User-Program.

This sums up the functionality in the CRCHardwareThread. Next the CRCCore itself is covered. A schematic of this core is represented in figure 5.3.

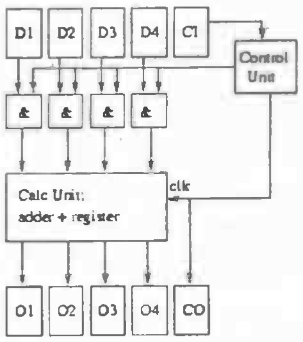


Figure 5.3: Hardware layout CRCCore

As one can see there are five input IOBs and five output IOBs. Four input IOBs (D1 till D4) are used for the input of new data, and four output IOBs (O1 till O4) are used to communicate the current CRC value (as indicated by the *result* variable in the software CRC algorithm) to the outside world. One input IOB (CI) and one output IOB (CO) are connected to a Control Unit. This Control Unit is necessary to indicate to the CRCHardwareThread whether the core is ready to be fed another 4-bit value to run its CRC algorithm on. Due to the CRCHardwareThread setting the input to the Control Unit from 0 to 1 to 0 etc, it can be used like a clock. The Control Unit is connected to the clock pin of the Calc Unit. Every time on a downward flank (from 1 to 0), the Calc Unit will add the value on the input but to the value stored in its register.

5.3.2 Software to hardware translation and vice versa

Having covered the software and hardware version of the CRC algorithm, it is time to cover how a running software CRC thread can be stopped and be put to continue running in hardware.

First consider software to hardware translation first. As mentioned before the most important variables of the software version piece of code are: *result* and *index*. When the software CRC thread receives a signal to stop, it sends these variables to the UserProgram. The UserProgram will store them till either:

- a new CRC software thread is created and needs to continue where the previous CRC software thread stopped.
- a hardware core is created and these variables need to be put into hardware.

The first of these two cases is the easiest. The stored variables from the first CRC Software thread can simply be communicated to the new software thread with a few assignments. The second case is more difficult. The problem resides in identifying the variables found in software in the hardware core; where are the *index* and *result* variables found in hardware? Having written a relatively simple CRC Core, these variables are relatively easily found. As mentioned before the hardware part of the algorithm is divided into a CRCHardwareThread and the CRCCore. The CRCHardwareThread feeds the CRCCore new input values to run the CRC algorithm on. The *index* variable needs to be transmitted to CRCHardwareThread so that it knows which value to send next to the core. The core however needs to have a few flipflops adjusted to reflect the *result* variable. The flipflops that need to be adjusted in the *Calc Unit* block are shown in figure 5.3. The Calc Unit consists of adder and a register unit, the register unit needs to be set to reflect the *result* variable. To be able to do this, the exact location of the flipflops used for the register unit need to be located on the FPGA and then be set accordingly to the value of *result*. The flipflops location was easily found since I wrote the core myself, in which I had to specify the exact location of the register unit. The procedure to set the flipflops as mentioned in section 4.5: *The status of a Core* can be used.
This covered software to hardware translation. Hardware to software works just about the same. The difference is that variables like *index* and *result* need to be generated from the status of a core. Since the location of the register unit is known, its flipflop values can be read out to produce the *register* value. The *index* value is store in the respective CRCHardwareThread.

Since only a simple CRC Core is used, it is fairly easy to go from software to hardware and vice versa. If however larger cores will be used and more difficult designs, it will become increasingly difficult, if not impossible, to identify software variables in a hardware core.

5.3.3 CommandLine

As mentioned before the UserProgram also features a commandline which the user can use to completely control the program. In the current application, the commandline was only programmed to know two CRCCores with the names CRC1 and CRC2.
The commandline features the following commands:

- "load hardware". This command will bring up a menu to load a CRCCore onto the FPGA. The user can specify the name and location of this core on the FPGA.
- "start hardware thread". This command asks the user for the name of a core which it will start a hardware thread for. In the current example the hardware thread used (CRCHardwareThread) will be started for a core specified by the user.
- "set hardware status". When using this command the user needs to specify a CoreName and location. At this location, the flipflops will be set so that the temporary calculation value of the core is represented.
- "stop hardware". This command asks for a corename and removes this core from the FPGA. The core is stored in an CoreStorage object within the CoreList object.

- "restore hardware". This command asks for a corename along with location and restores the hardware configuration of that core at the given location. Mind that this command should be given AFTER "set hardware status".
- "start software". This command starts a software version of the core CRCCore. Again the user is allowed to specify the name of the core.
- "stop software". This command asks for a corename and stops this software thread.
- "help". Shows a list of the possible command which can be given.

5.4 Scenarios

To illustrate how the CRC RTR-application is working, a few scenarios are presented here. One can start by starting a software CRC thread using the command "start software". This thread will calculate the full length of the CRC array which results in a value X as answer. Next a CRC core is put on the FPGA using "start hardware". A hardware thread ("start hardware thread") is started to calculate the results of an entire array, which results in the same value X. This indicates that both the software version and the hardware version result in the same value X, which is an indication that the application is working correctly.

Next the more advanced routines of the application are tested to see if software to hardware translation and vice versa works as well. This is done by starting up a software thread ("start software"), stopping this thread after one third of the calculation ("stop software"), setting the flipflops ("set hardware status") to reflect the status where the software thread stopped, put the core onto the FPGA ("load hardware") and starting its hardware thread ("start hardware thread"). The hardware thread can now continue running for another third of the calculation process, after that it will be stopped using the command "stop hardware" and a software thread can be start up using "start software" again to continue calculating where the hardware thread left off. When it is done calculating, the same result value X will be given. The UserProgram was made to test having multiple cores running, so the same test was repeated with another hardware core running which resulted in another correct value X.

5.5 Conclusions and future improvements

In this chapter the entire RTR-application was covered: its structure, its classes, the CRC algorithm in both hardware and software and the translation between them. The goal of this thesis has been met: covering the development of writing a RTR-application. The Virtex proved to be suitable for the application, however it would have been even more suitable if it allowed to set flipflop values during a reconfigure which currently is not supported (also see section 4.5: the status of a core). JBits v2.8 is a perfect interface to have full control over the FPGA. To have this much control is both a blessing and a curse. The good part is that you can do everything in the *exact* way you want it, full control. The bad side is that programming cores using JBits is difficult since one has to give command on how to program the core at CLB/flipflop/pin level. Because of this it is easy to lose the overview of how a core is working. This is especially true if you also need to consider the values of the MUXs in order to know what a core does. A solution to this problem would be to develop a core in a high level language like VHDL and then be able to put this VHDL core on the FPGA using JBits. The current problem however is that it is not possible to use cores created by the Xilinx Foundation tools (used program a core in VHDL) in JBits. My recommendation for anyone writing a RTR-application would my to use a low level language/interface to have full control over the FPGA, while using a high level language like VHDL to be able to program more complex cores than for example a CRC Core.
The RTR-application is developed to do CRC checks. However the hardware core needs to have its input values transmitted by the CRCHardwareThread to the FPGA. As was noticed during

testing, communication between software and the FPGA takes a lot of time. The reasons for this communication to take such a long amount of time is because there is a thread continually running to check if the FPGA is outputting new values on its IOB pins using busy waiting. Instead of busy waiting other faster algorithms can be used, but their executing time will still be slow compared to the input request rate of the FPGA. Therefore to get the best speed out of the FPGA, communication between FPGA and software should be minimized.

Future improvements

Several improvements can be made to improve the versatility and speed of the RTR-application. In short I aimed at dividing the RTR application in two parts, one for the actual application programmer (*UserProgram*) and one part for storing re-usable functions (*ReconfigInterface*). In order to make the life of the programmer who is developing the UserProgram as easy as possible, one can make improvements to ReconfigInterface. Currently it is necessary to specify where a core needs to be placed. Instead, a placing algorithm can be developed to place the core in the best suitable location on the FPGA instead so that the UserProgram programmer does not have to specify the location anymore. This will be especially useful if multiple UserPrograms will be put on the FPGA by different applications (sharing a single FPGA).
Another improvement can be made in the WireManager class. Currently it only uses IOB pins from the bottom IOB row to allocate IOBs to requests from cores. This can be expanded to have all IOBs allocatable, but not only that, the algorithm can also consider which pins are closest to which cores and allocate them accordingly.
Currently the RTR-application depends on the user (which enters commands in the commandline) to tell when a core needs to be removed from the FPGA and whether it will run in hardware or software or not. Instead of letting a user decide this, it can be preferable to use a scheduling algorithm to make these decisions. However a scheduling algorithm brings its own problems along because: how can a scheduling algorithm predict how long a core will need to execute before it is done? Usually this is not possible unless the core has been tested on its running time. The prediction algorithm will need to guess whether it is feasible to execute a task in hardware or software.
The previous mentioned improvements for the RTR-application are not going to be a very big problem to implement. A more difficult part will be the software to hardware translation and vice versa (as pointed out in section 5.3.2). This will especially hold true if a core is developed using a high level language because this way the user will not know where important registers/variables in the core are located or in which flipflops these values can be found. This is considering the case that the software variables also exist in some way in the hardware version of the algorithm. However since software and hardware differ fundamentally variables in software do not have to occur in hardware. This problem can cover a thesis by itself.
Another difficult problem to solve is whether it is more feasible to run an algorithm in hardware or software, or run maybe a tiny part of a software algorithm in hardware instead of it whole. Which parts of an algorithm perform well in software and which part performs better in hardware? Again these questions can cover an entire thesis.

Appendix A

References

1 Paul Graham and Brent Nelson, *A Hardware Genetic Algorithm for the Traveling Salesman Problem on Splash 2*, 5th International Workshop on Field Programmable Logic and Applications, pp 352-361, August 1995, Oxford, England.

2 T. Wiangtong, P.Y.K. Cheung and W. Luk3, *Multitasking in Hardware-Software Codesign for Reconfigurable Computer*, Proceedings of the 2003 International Symposium on Circuits and Systems, ISCAS '03,Volume: 5, 25-28 May 2003

3 Edson L. Horta, John W. Lockwood, David E. Taylor and David Parlour, *Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration*, Proceedings of the 39th Design Automation Conference, DAC 2002, New Orleans, LA, USA, June 10-14, 2002.

4 Brian Greskamp and Ron Sass, *A Java Virtual Machine for Runtime Reconfigurable Computing*,Parallel Architecture Research Lab Holcombe Department of Electrical & Computer Engineering Clemson University, SouthEast Conference 2003

5 Xilinx, Information about Xilinx 4005 FPGA and Xilinx Virtex Series.
Online: http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp

6 Xilinx, JBits SDK. Online: http://www.xilinx.com/products/jbits/