Reconfigurable FPGA Processor

Andreas Romer, Rolf Enzler, Didier Cottet, Gerhard Tröster
Swiss Federal Institute of Technology
Electronics Laboratory
CH-8092 Zürich
E-mail: andreas.romer@ieee.org or enzler@ife.ee.ethz.ch

Technical Report, February 2000

Abstract

Progress in the FPGA technology have made partial reconfigurable devices appear on the market. The device size is also getting larger pushed by the progress in CMOS technology. There are already devices on the market that have reached the one million gate equivalent, thus allowing to map whole processors on FPGAs. In this report a concept of a dynamic reconfigurable processor is being introduced that includes a very simple accumulator processor. It can allocate functional units through a reconfiguration process into the reconfigurable part, in order to be able to execute an instruction. The concept is validated with an implementation on the Virtex device of Xilinx. The Virtex and the XC6200 devices of Xilinx are described in detail, and their design flows are explained. A proposal is made for a design flow to create a locally constraint partial reconfiguration for the Virtex. An in VHDL written implementation of the processor onto the Virtex resulted in a clock frequency of 55 MHz with only 300 CLB slices used of a Virtex device. This is less than a tenth of a XCV300 and equals to about 5440 gate equivalents.

1 Introduction

Solutions for some computational problems are mainly driven by the demanded speed and the budget available. If the highest possible speed is asked for and the algorithm is known, often the only possible solution is to design an expensive ASIC. The ASIC will certainly fulfill the speed criteria, but if a slight change in the algorithm occurs, the same ASIC cannot be used. There has to be a complicated and expensive redesign. On the other hand, when the required speed is not critical, a slower and cheaper variant is possible. A General Purpose Processor (GPP) will be fast enough to master the task. The advantage of this solution is that a change in the algorithm can easily be overcome with a change in the software and therefore the GPP can be used again. It is clearly that the GPP has more flexibility than the ASIC, but is slower.

Besides the speed and the flexibility, the used area is also an interesting point of comparison. Todays GPP try to get a lot of parallelism out of the software. They are superscalar and pipelined. However, in order to be able to support this features a lot of controlling overhead is necessary. But even with a lot of controlling the flushing of a pipeline and idle execution units can not be avoided due to data dependencies in the instructions. The size of the latest GPP is simply huge. They cover a die area of more than 22 millions transistors. On the other hand ASICs are usually small, since they are application specific and therefore do not need the large controlling overhead of GPP.

Is it now possible to combine those features to a single device? Is it possible to have a flexible architecture on a device that implements some application specific algorithms? Will this solution be faster than a GPP?

Recent attempts to find new ways to speed up computation take advantage of the FPGAs, since they offer the speed similar to an ASIC with a flexibility equal to or even higher than a GPP. Implementation approaches of such reconfigurable systems have revealed that the algorithms implemented on the FPGAs can outperform GPPs. This research area is named reconfigurable computing (RC). A first approach is to reconfigure the FPGA at the beginning of an application, also called compile time reconfiguration. Such
static reconfiguration have been implemented and lead to considerable speedups of this applications. However, the static reconfiguration is not really flexible, since the reconfiguration cannot take place during execution. In [4] the Run Time Reconfiguration (RTR) is discussed more closely. The RTR maps execution units according to its necessity into the FPGA. To support this feature the FPGA needs to be partial reconfigurable or the system is compromised of several FPGAs which are not partial reconfigurable and each reconfiguration process will reload at least one whole FPGA. Several RTR systems have already been designed on FPGAs ([5] and [1]) and also revealed high speedups over GPPs.

But this approaches, to implement the whole design on an FPGA are not really efficient, since the reconfiguration process needs some hardware, which in turn is static and therefore only wastes FPGA space. The static logic could be implemented on an ASIC and the RTR part is implemented on an FPGA. But this would lead to delays caused by going off the chip. The better solution would be to have an FPGA core surrounded by an ASIC. This is what [2] and [3] did in their project.

2 The Basic Concept

This chapter will explain the basic concept of the reconfigurable processor.

Figure 1 shows the five main parts of the reconfigurable processor: the CPU with some static instructions, the Reconfigurable Part, the Reconfiguration controller and the two information tables. In the FU Information table information about the instruction (FU) is stored. Similarly, in the Reconfigurable Part Information table the state of the reconfigurable part is stored. Each instruction that will be fetched to the whole system will first arrive at the CPU with some static instructions. This is a very simple CPU that has a few instructions statically implemented, but is fully functional on its own. Subsequently, the FU Information table will be named FUTable and the Reconfigurable Part Information table will be named SIDTable. There are three different occurrences that are possible now:

1. The fetched instruction is statically implemented:
   In this case the code is simply executed in the CPU without further implications.

2. The fetched instruction is recognized as a reconfigurable instruction and is already present in the reconfigurable part:
   The CPU has to know where the instruction is located in the reconfigurable part. This information is stored for each reconfigurable instruction in the FUTable. Additional information (e.g. how many cycles the instruction needs to execute) about the instruction is also stored in this table. When the CPU knows the exact location in the reconfigurable part of the instruction, the CPU can send the data to it and the execution can begin. This data is sent to the reconfigurable part through a Standard Interface.

3. The fetched instruction is recognized as a reconfigurable instruction but is not present in the reconfigurable part:
   The FUTable will reveal that this instruction is not present in the reconfigurable part. Again the additional information of the instruction is loaded from the FUTable. Then the Reconfiguration Controller starts the reconfiguration process. It decides where the reconfiguration of the instruction will take place. If there is another instruction present at that location, that will consequently be replaced, the FUTable needs to be updated for this replaced instruction. Also the location of the new instruction in the reconfigurable part needs to be written to the FUTable. The Reconfiguration Controller gets the address of the reconfiguration location from the SIDTable and the address of the bitstream in the main memory from the FUTable. Then, all the information is known in order to start the reconfiguration process. When the reconfiguration process is completed, the instruction can be executed.

3 Implemented Processor

The implemented processor has an ACU architecture. Figure 2 on the next page shows the data flow.

\[3\text{Accumulator: the data is stored back into the register where it came from}\]
Figure 2. The architecture of the implemented processor. The four following stages are distinguished:

- In the Instruction Fetch stage, the instruction at the Program Counter (PC) is loaded into the IOReg. Also the current PC is incremented by eight (the instruction word width is 64 bit).

- In the Instruction Decode / Register Fetch stage the instruction is decoded and the data from the IOReg is loaded into the three registers according to the decoding.

- The Execution / Address Calculation stage computes the result according to the instruction and the next PC is chosen with the Flags. Whether the Compare unit or a FU is used, is decided by the decoded instruction. The FU appears to the processor as a normal computation unit. The result of the execution is again stored to the ACU.

- Finally, in the Writeback stage, the ACU can be stored to the address in the IOReg.

The implemented processor computes one instruction concurrently. But with some additions and modifications, this processor could be made to execute multiple instructions through pipeling the data flow at the three dashed lines. The datawidth of the implemented processor is eight. This size was kept low in order to minimize the size of the design. The addresswidth of 26 bits yields to a total addressable memory size of 64Mbyte. The processor has five flags.

The controlling units of the reconfigurable unit are more complicated than for a normal processor. Figure 3 shows the control flow of the reconfigurable processor.

The Execution Controller is the only controlling unit that is in common with a static processor. It decodes the instruction and sets the enable signals according to the decoding, in order to load the data into the three registers. At the end of the execution phase the Execution Controller enables the ACU and the Flags register to have the results stored in them.
All the components that are inside the black framed box are units that are necessary for the reconfigurable feature. The information about the FU is stored in the FUTable. Upon execution start, the FPGABaseAddress, the Counter and the SID is loaded from the FUTable. The SID selects the right FU. For the execution, the Counter is counted backwards until zero is reached. However, when the FU is not present (this is indicated with a "000" in the SID) the Execution Controller activates the Reconfiguration Controller. The Reconfiguration Controller gets the right information from the SIDTable. The gotten FPGAAddress indicates the location inside the FPGA. The StartReconfigurationS signal starts the reconfiguration process. The reconfiguration stream at the address FPGABaseAddress is loaded into the FPGA at address FPGAAddress. When the reconfiguration process is finished the Reconfiguration Controller get the signal ReconfigurationDoneS, which in return generate the DoneS signal. Then, the new FU is available for the execution of the instruction.

4 Implementation on the Virtex

Virtex devices feature a regular architecture that comprises an array of CLBs surrounded by programmable IOBs, all interconnected by a rich hierarchy of fast, versatile routing resources. The abundance of routing resources (the Virtex is produced in a 5-layer-metal 0.22-µm CMOS process) permits the Virtex family to accommodate even the largest and most complex designs. Virtex FPGAs are SRAM-based, and are customized by loading configuration data into internal memory cells. In some modes, the FPGA reads its own configuration data from an external PROM. Otherwise, the configuration data is written into the FPGA. Virtex devices provide better performance than previous generations of FPGA. Designs can achieve synchronous system clock rates up to 200 MHz including I/O. The Virtex is the successor of the XC6200 idea: a lot of features found on the Virtex were already available on the XC6200. But the fine-grained architecture of the XC6200 has been abandoned and has been replaced by an architecture that balances speed and density (e.g. dedicated carry logic for high-speed arithmetic and dedicated multiplier are supported). The fact that the standard software of Xilinx (the same as for the XC4000) is being used to generate the bitstream, allows a broader acceptance on the market for the Virtex. The lack to support the standard software for designing was one of the reasons why the general user never used the XC6200. Xilinx obviously learned from its mistakes.

The Virtex’ CLBs interconnect through a general routing matrix (GRM) (see figure 4). The GRM comprises an array of routing switches located at the intersections of horizontal and vertical routing channels. The VersaRing (see figure 5) I/O interface provides additional routing resources around the periphery of the device. This routing improves I/O routability and facilitates pin locking.

Please note that the above mentioned 3-state buffers, that drive dedicated segmentable horizontal routing resources are perfectly fit to design a partial reconfigurable design. While the logic in the CLBs and some potential routing is reconfigured, the horizontal routes, that are driven by the 3-state buffers may remain untouched through the partial reconfiguration and allow therefore data to cross a region of partial reconfiguration during the reconfiguration process. The only potential risk is that more than one 3-state buffers could drive the route during reconfiguration, since the logic levels of the enables are undefined during reconfiguration of the enable logic. In reality, this risk does not exist, since the 3-state buffers are implemented as multiplexers.
Figure 6. Structure of the synthesizeable processor

<table>
<thead>
<tr>
<th>Controllers</th>
<th>Controlled Entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle</td>
<td>Registers</td>
</tr>
<tr>
<td>Compare</td>
<td>MainController</td>
</tr>
<tr>
<td>PCCalculate</td>
<td>IOInterface</td>
</tr>
<tr>
<td></td>
<td>ExecutionControl</td>
</tr>
<tr>
<td></td>
<td>ReconfigurationController</td>
</tr>
</tbody>
</table>

Table 1. Synthesis results

<table>
<thead>
<tr>
<th>Maximum clock frequency [MHz]</th>
<th>55</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CLB slices</td>
<td>294</td>
</tr>
<tr>
<td>Number of flip flops</td>
<td>187</td>
</tr>
<tr>
<td>Number of 3-state buffers</td>
<td>64</td>
</tr>
<tr>
<td>Number of IOBS</td>
<td>250</td>
</tr>
<tr>
<td>Gate equivalents</td>
<td>5435</td>
</tr>
</tbody>
</table>

Figure 6 shows the synthesizeable design. The memory, FUTable and the SIDTable have not been included, because they are written in behavioral VHDL. Since all the signal, that are not used in the design (especially all control and data signal to the FUTable and SIDTable) are connected to a pad, the synthesized design gives only a rough estimation of the performance. The states for the state machines in the controllers are all one hot\(^4\) encoded, as advised by the Synopsys Synthesis Design Guide in order to get a higher performance. The attributes are inserted after the state declaration in the VHDL code.

The synthesis with the place and route of the synthesizeable design seen in figure 6, resulted in the numbers given in table 1. A CLB slice is half of a CLB, which is equal to two LCs. For a XCV300 device this would only cover about 10% of the CLBs. So there would be plenty of space left in order to have a big reconfigurable part.

\(^4\)The statevector is represented by only one one for each state. Therefore a state machine with five states would need five flip flops

References


