Master Thesis

A Parameterizable Processor for Reconfigurable Computing on FPGAs

Author(s):
Koster, Stephan

Publication Date:
2015

Permanent Link:
https://doi.org/10.3929/ethz-a-010475960

Rights / License:
In Copyright - Non-Commercial Use Permitted
Master Thesis

A Parameterizable Processor for Reconfigurable Computing on FPGAs

by
Stephan Koster

May 2015

Advisors:
Prof. Thomas Gross
Dr. Felix Friedrich

Computer Systems Institute

Department of Computer Science
Swiss Federal Institute of Technology (ETH), Zurich, Switzerland
ETH-Zentrum, IFW
CH-8092 Zürich
6 Conclusions
6.1 Float and Mul ........................................... 27
6.2 Nonblocking IO ........................................... 27
6.3 Instruction Width ....................................... 27
6.4 Clock Speed ............................................. 28
6.5 Is Parametrization Practical? ......................... 28

7 Perspectives .............................................. 29

A Hardware Details ......................................... 31
A.1 TRM Instruction Set ...................................... 31
A.1.1 Instruction Formats ................................... 31
A.1.2 Floating Point ........................................ 33
A.2 Floating Point .......................................... 33
A.2.1 Old System ............................................ 33
A.2.2 New System ............................................ 33
A.3 Optional Multiplier ...................................... 34
A.4 Instruction Width ....................................... 35
A.5 N Bit IM .................................................. 36
A.6 IO Subsystem ............................................ 38

B Compiler Details .......................................... 41
B.1 Overview ................................................ 41
B.1.1 Frontend .............................................. 41
B.1.2 Backends .............................................. 41
B.1.3 Intermediate and TRM Backend ..................... 42
B.1.4 Active Cells Backend ................................ 42
B.1.5 Hardware Descriptors ............................... 42
B.1.6 Linker ................................................. 42
B.1.7 Static Linker .......................................... 42
B.1.8 Hardware Generation and Deployment ............ 42
B.1.9 Integrating Instruction Width ....................... 43
B.1.10 Integrating Memory Configuration ................. 43
B.1.11 Changes to Floating Point Support ................ 43
B.1.12 Integrating the Optional Multiplication Unit .... 43

C Persistent Issues ......................................... 45

D Changelist ................................................. 47

E Task Description and Declaration of Originality .... 51
Acknowledgements

I would like to thank Dr. Felix Friedrich and Prof. Dr. Thomas Gross for supervising my thesis and for support, advice and the opportunity to work on this project.
I would also like to thank Dr. Oleksii Morozov for his input and giving me the motivating feeling this work will find some real-world applications.
Finally, I extend my gratitude to VIS for the generous supply of free coffee.

Stephan Koster,
Zurich, May 2015
Motivation

In compiler design, there is a general assumption that the instruction set and its particular implementation in hardware have certain fixed characteristics that the compiler has to exploit or work around in order to generate good assembly. What constitutes good assembly is dependent on the demands of the programmer, expressed as compiler directives, such as best execution speed, minimal code size, lowest energy consumption and so on. Ideally, the programmer delivers his code to the compiler, chooses an instruction set and compiler directives, then lets the compiler do all the hardware dependent optimization. However, when the compiler’s output is not satisfactory, the programmer has to start considering the peculiarities of the instruction set to coax the compiler into generating the desired assembly or even write assembly by hand.

We try to improve this process with a parametrized processor. Essentially, this enables us to turn the tables and generate the best instruction set for a given program instead of the other way round.

Obviously, it is impractical to fabricate a custom microchip for every program we write and emulating a custom processor on a commodity processor will not result in a practically useful system. Instead, we implement our processor on an FPGA chip which gives us less performance than custom silicon but can be reconfigured with different hardware designs. As a starting point we take the Active Cells system and its standard processor the TRM[1]. This system handles code generation and deployment for us. It enables us to easily build a network of processors on our FPGA.
Chapter 1

Introduction

In this thesis, we explore the parametrization of a processor implemented on an FPGA. The parameter values can be set in a high level program and matching assembly is automatically generated.

As a target for this experiment we use the TRM (Tiny Register Machine) processor embedded in the Active Cells framework. We measure the impact of the new parameters on various metrics.

1.1 Active Cells

Active Cells is a system developed by the Native Systems group at ETH Zurich[1] to easily describe and instantiate a network of processing nodes on an FPGA (Field Programmable Gate Array). The network topology and the tasks of each node are described in dialects of the Oberon language. The system has applications in high performance computing and embedded systems[3].

Nodes in the network, called Cells, are implemented as a processor running a program each, communicating with each other through Channels. There is also a facility to incorporate special nodes in the network that are not implemented as a processor running a program, but rather as arbitrary logic. These special nodes are known as Engines. There is no need for task switching, lock contention or processes, and all peripherals are handled through Engines, so the operating system vanishes.

Engines are used for computation intensive tasks with high throughput requirements and as IO endpoints.

Once the topology consisting of Cells, Engines and Channels have been implemented, the programs executed by the Cells can be updated without running through the entire hardware generation cycle, see Figure [1.1]. This is called fast path compilation and makes the development process more convenient and faster.
1.2 TRM

1.2.1 Architecture Selection

When we started off, we had a variety of options to base the design of our parameterizable processor on.

- We could start with a 32 bit RISC architecture as proposed by Niklaus Wirth\[7\], which has a particularly regular instruction set. It is a simple general purpose architecture, but not especially optimized for implementation on FPGAs.

- The TRM\[5\] (Tiny Register Machine) is a Harvard architecture with a data width of 32 bits, an instruction width of 18 bits and a bare bones instruction set of just 16 different instructions. The instruction size and a few other peculiarities come from its optimization for Xilinx FPGAs\[6\]. Versions of the TRM are the standard in the Active Cells system. The tight instruction word size has proven useful for small projects, but also lead to problems in the past which makes this an attractive target for parametrization.

- We could also have created a new processor from scratch. This would have given us maximum flexibility to try out even very unconventional avenues of parametrization. For instance, a processor could be built of more or less freely pluggable modules with a standardized interface to implement the different operations.

We ultimately decided to build on top of the TRM because it was the least risky option with the most potential for immediate use in practice should our results...
be encouraging.
The TRM is a RISC architecture with a very small number of instruction formats and strong regularity which leads to a frugal hardware design. The particular implementation we used as a baseline is the TRM0, which can execute all operations on registers in a single cycle, load/store operations and jumps in at most two. A write-up of all instruction formats can be found in Appendix A.1. Versions with pipelining have been tried out in the past, but did not sufficiently improved performance\cite{4}, so there is no pipelining in the version of the design we used.

1.3 Contributions of this Thesis

In this thesis, we introduce a parametrized TRM to the Active Cells system. We implement the following hardware parameters and the according compiler changes:

- Optional floating point support. Floating point support is already implemented, but we introduce a parameter to toggle it on or off.
- Optional multiplication unit and corresponding runtime procedure to replace its functionality when turned off.
- Optional non blocking IO support. This includes an overhaul of the IO system that directly integrates AXI stream channels in the TRM which is cleaner than the previous solution and enables much higher throughput.
- Free choice of instruction width and instruction memory configuration.

We then run experiments to test the impact of these new parameters.
Chapter 2

Parametrization

2.1 Why Parameters?

With the introduction of new parameters, we want to create flexibility in the hardware design so it can achieve certain design goals. From a software engineer’s point of view, that goal is usually a program’s execution time, but for a resource-constrained system like Active Cells, we need to consider other metrics as well.

- When modules other than TRMs on the FPGA need to use the same clock\(^1\) the maximum achievable clock speed is often limited by the TRM. A TRM with suitable parameters may work at higher clock speeds.
- On-chip memory is limited. In the active cells system, this memory must hold both data and program memory for each cell. Well chosen parameters may lead to better code density and thus use less memory for the same program.
- An FPGA chip contains a limited number of programmable registers and look up tables (LUTs) connected by a switching fabric. A good design should try to use as few of these resources as possible.
- Execution time is still important of course. Improvements on the other metrics often lead to a trade off with execution time though, because we generally need more instructions to execute the same task when we remove hardware support.

2.2 What Parameters?

There is a variety of ways parameters can change a hardware design. Many parameters just turn a feature on or off, for example floating point support. In our processor, this is the most common kind of parameters. Even though there is no nuance in the presence or absence of any single feature, if there are enough of these parameters, the combination of features can still lead

\(^1\)Separate clock domains can often give great benefits to overall throughput but are notoriously hard to get right
to a tight fit.
Particularly interesting to us is another category of parameters, those with integer values. Examples for such parameters are cache size, number of cores, number of IO channels, number of registers and instruction width. Since most parameters lead to a tradeoff between different design goals, variation in integer valued parameters can sometimes find a sweet spot where the most efficient configuration with regards to a particular design goal lies. We have implemented flexible instruction width, mainly because the instruction width was a limitation in the previous TRM hardware design.

In the Active Cells system, we have the possibility to instantiate hardware engines that communicate over communication channels with cells implemented as TRMs. In the interest of efficiency, we could include such engines directly in the TRM. This would make sense whenever a Cell is basically just communicating with that one engine, so the inclusion of the engine could remove the detour through the TRM's IO and the channel. Using an engine in a TRM would basically mean introducing a new instruction that delivers a word of data to the engine. It would probably make sense to introduce some kind of DMA engine in a so equipped TRM so an entire region of data memory could be fed to the engine without executing multiple TRM instructions. We chose not to pursue this path because we saw the possibility to make the existing IO subsystem almost as fast (see A.6).

A more effective use of modularity would be to reduce the processor to just a core that decodes instructions and holds a few registers, but delegates all computation to a set of modules that are connected to the core through some kind of unified interface. This would allow us to customize the available instructions to a high degree by adding many CISC-like features on demand. It would also be more future-proof, because it would for example allow new features of future FPGAs to be included in the processor more easily. We have not worked on this idea much, because effectively using these features requires a much different compiler, though we recommend exploring this approach in the future. In particular, we think a set of optional conditional move instructions could be beneficial.

2.3 How to Implement Parameters

There are a variety of techniques how parameters provided in the high level description can arrive in hardware

- The old mechanism for floating point support simply exchanged the entire Verilog source file for the TRM if the option was active. This mechanism is obviously not scalable to a greater number of parameters because there would have to be a source file for every combination of options. Integer-valued parameters are not possible with this mechanism.

- We could rely on Verilog code generated by the Active Cells compiler. This is a very powerful technique, but it would be very hard to understand and change the system in the future. Currently generated code is limited to just the top level module that instantiates and connects all cells and engines.
2.3 How to Implement Parameters

• Verilog preprocessor statements allow the effective code to be different depending on compile time parameters. This has the advantage that depending on compile time definitions, the code can be manipulated arbitrarily. In the extreme case, it would be possible to have two entirely different modules in the same file and select one with the preprocessor. Unfortunately, the semantics of preprocessor statements do not directly support instantiating the same module multiple times with different parameters. There is a known workaround, but it would involve an extra level of generated code and relies on the compile order of source files. Also, having distinct chunks of code for different processor versions would push us towards having a separate code section for every combination of parameters which again does not scale well.

• Verilog parameters are another way to introduce parameters into the design. Unlike preprocessor statements, they do not allow pieces of code to be really deleted or changed, just the definition of constants depending on the parameter choice at the instantiation of the module. With a few programming tricks, these constants can be just as powerful as the preprocessor. For example, if we want to delete a submodule depending on a parameter, we can set all inputs to the submodule to the logical 'and' of the input and the enabling constant. The synthesizer is smart enough to not generate any extra hardware when the parameter is set to '1' and removes the unneeded submodule in the netlist trimming phase if it is set to '0'.

We use this mechanism exclusively even though it leads to somewhat less readable code compared to the other methods.
Chapter 3

Implementation

3.1 Floating Point Support

Traditionally, floating point numbers are supported in hardware by providing a set of special arithmetic operations, sometimes using a separate set of registers. However, in the TRM architecture, the opcode has only 4 bits, so there can be only 16 distinct operations. This is not enough to provide a complete set of arithmetic instructions for both integer and floating point arithmetic. Also, a flag bit designating any arithmetic instruction as floating point or integer would cut into the already very tight space for immediates. However, in instructions for binary register operations (that includes all arithmetic) without immediate, there are some free bits. The TRM uses these to flag the operation as a floating point instruction. When no hardware support for floating point operations is present, operations on floating point variables are translated into calls to runtime procedures that emulate the operation with integer arithmetic.

In the old system, floating point support was already optional by a mechanism that used an entirely different hdl source file for the TRM if the floating point option was set. We chose to keep the old modus of floating point instructions where arithmetic instructions are flagged as floating point by a free bit in instructions without immediate. We merged the two versions of the TRM into a single module that has a parameter so it can be instantiated with or without floating point support. We could also have changed the system to optionally have a larger opcode width so we could introduce a separate set of floating point instructions. This would have given the user an option for a tradeoff between floating point support and size of immediates. We did not do so because we felt the current system already gives the same benefit without drawbacks. If the instruction set was more complex, our chosen solution would probably not suffice though.

3.2 Optional Multiplication

We noticed that many programs don’t use the entire instruction set. It therefore makes sense to remove the logic for unnecessary instructions in the processor. The multiplication unit was a particularly good target since it is a distinct hard-
ware module and it is relatively easy for the Active Cells programmer to see when it is not needed. If the multiplication unit is removed and some multiplications are still present in the program, the multiplication unit is emulated with a runtime procedure using shifts and additions. This is obviously much slower and uses more code memory than a simple assembly instruction. **The savings in hardware resources by optional removal of the multiplication unit are substantial, the resource use per TRM is cut roughly in half**, see chapter 4.2.3.

After we remove floating point support and the multiplication unit, it makes not much sense to reduce the instruction set further as we hit diminishing returns.

### 3.3 IO Support

The IO interface of the TRM resembles that of common microcontrollers. It features a 32 pin in- and output bus with an 8 bit IO address channel and a read- and write enable signal. However, in the Active Cells system, TRMs are not exposed to the outside, but are exclusively connected to other TRMs and hardware engines through channels following, in recent versions of Active Cells, the AXI stream standard. In the AXI stream standard, a 'valid' flag travels with the data from the sender to the recipient, and a 'ready' flag the other way. A data transfer takes place in every cycle where both the valid and ready flags are set. To attach the old TRM interface to these channels, a wrapper was in place that used the top two bits of the IO address to access the input valid and output ready flags. To perform a channel write for example, a runtime procedure would check the output ready flag and spin until it was set, then perform the data write. Nonblocking reads and writes were performed in the same fashion, just without the spinning.

**We improved this system by incorporating blocking IO directly into the TRM.** Now when we perform an IO write, the TRM stalls until the recipient is ready. **This way, we can get rid of the runtime procedures for blocking send and receive entirely, allowing us to transfer up to one word per cycle**. The old interface can be easily reintroduced with a simple wrapper that ties the output ready and input valid signals of the TRM to constant high.

Using just blocking IO, we can build Cell nets equivalent to Kahn networks. However, in practice we sometimes need to use nonblocking IO, for example to peek on a channel whether data is available. We introduced optional support for nonblocking send and receive operations. If a nonblocking IO operation fails, for example if a nonblocking send is executed when the recipient is not ready, a processor flag register in the TRM is set. This flag causes a branch condition in the next cycle to evaluate to false which we can use to build a runtime procedure. For a more detailed explanation, see Appendix A.6.

---

1 The AXI standard also specifies burst size signals which we omit
2 Provided the data is already stored in registers. A typical use case where data is read from data memory and pushed to a channel in a loop would work out to at least 4 cycles per word transferred
3.4 Flexible Instruction Width

3.4.1 Bringing the Parameter to Hardware

We introduced a parameter to customize the instruction width. Previously, the instruction width was fixed to 18 bit with two such words stored in every line of memory. This was designed this way because the memory primitives available on our board support 36 bit words. The narrow instruction width leaves only few bits for uses like immediates and jump offsets. The instruction width parameter removes these limitations. The instruction width parameter gets used in the hardware description mostly in the preamble of the TRM source where we derive several local parameters from it. Those local parameters are used later to extract features from the current instruction word. In contrast, the previous solution had relied on hardcoding. Some examples of the new code can be found in Appendix A.4

3.4.2 Instruction Width in the Compiler

In the compiler, we had to introduce an instruction set class and instantiate an object for every cell. This instruction set object gets initiated with the chosen instruction width and is used to generate assembly words accordingly. The same object is also consulted every time the compiler has to make a decision on whether an immediate or jump offset fits into the instruction word or not. Some difficulties arose because the compiler is set up to produce an intermediate assembly. When the intermediate assembly is generated, there must be an instruction set object with the correct instruction width present to validate immediates, but after the intermediate code is emitted and parsed for the second phase we need the instruction set object again to generate assembly. We had to include the selected (or default) instruction width in the intermediate code sections to get around this problem.

3.4.3 Instruction Memory

As mentioned previously, the old solution had an instruction width of 18 bit so it could pack two instruction words into a line of the block RAMs of our FPGA. If we change the instruction width, this obviously no longer works. We had a selection of options on how to implement an N-bit instruction memory.

- A first possible solution is using Block Ram IP cores provided by the FPGA manufacturer. These IP cores promise arbitrary width and depth memory optimized for the particular FPGA model. However, patching in the memory contents after bitstream generation is not supported for the current generation of these IP cores.

- We could continue to use the manual block RAM instantiation but pack just one word per line, throwing all unneeded bits away. This is the easiest solution, but very wasteful.

3 proprietary hardware modules
• We could define the instruction memory as a Verilog array and let the synthesis tools figure out how to implement it in hardware. This would make the code more portable in theory because we do not have to deal with vendor-specific instantiation of hardware primitives. Unfortunately the tools to set the memory contents after bitstream generation as needed by the fast-path compilation of Active Cells cannot reliably deal with such an array. We could partly get around this limitation by specifying an initialization file in during synthesis, but this would sacrifice the Active Cells fast path compilation.

• We could manually instantiate block RAMs but cover them with some glue logic so words can be distributed over multiple RAMs. We have to be careful to retain single-cycle memory access. This is the option we eventually took, for more details see Appendix A.5.

The restructured program memory also requires us to change how we deploy the assembled program. Because we chose a method in which we determine manually how instruction words are gathered from potentially multiple block RAMs, we can just do the opposite in the final stages of the compiler. We leveraged existing functions that write the contents of a file to the bitstream. To this purpose, we introduced an intermediate hex file that stores one word per line no matter the memory configuration. This is also a more generic and readable solution than what had been done previously.

3.5 New Possibilities Through Flexible Instruction Width

We have options to remove so many parts of the already limited TRM, that we run into diminishing returns if we want to remove even more. Therefore it makes sense to add features instead. However due to the tight instruction coding, we run out of instruction bits to signal the presence of potential new features. This can obviously be remedied by making the instruction width wider using the facilities introduced for the flexible instruction width. This would lead to a whole new area of tradeoffs of instruction width versus functionality or functionality versus immediate size. To properly use these additional features, we would need a much more optimizing compiler though that can for instance determine when it would be beneficial to use conditional moves, vector instructions and so on. This is however outside the scope of this thesis.
Chapter 4

Experiments

We checked how the different newly introduced parameters influenced the value of several metrics in a series of experiments. Most parameters lead to a tradeoff between hardware resource use and program memory size.

4.1 Test Set

The effects of on/off parameters can be characterized with a simple test and can be easily explained. Instruction width however demands that we try different values with a variety of input programs because the precise effect depends on the instructions chosen. We have created a set of Active Cell programs to exercise the system.

Table 4.1 shows the helper Cells and Engines that we reused between several test cases, Table 4.2 gives a short description of every test case.

<table>
<thead>
<tr>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPO</td>
<td>A hardware engine that constantly reads data from a channel, stores it in a</td>
</tr>
<tr>
<td></td>
<td>register and pushes the register contents to the LEDs</td>
</tr>
<tr>
<td>Timer</td>
<td>A hardware engine that holds a countdown that can be set on a command channel</td>
</tr>
<tr>
<td>Pulse giver</td>
<td>A program for the TRM that repeatedly counts to a large number and sends a</td>
</tr>
<tr>
<td>Display driver</td>
<td>The controller TRM uses the timer to get the precise timings to control the LCD display through a GPO unit.</td>
</tr>
<tr>
<td>1 TRM, 1 Timer, GPO</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Utility Components for Tests
## Experiments

### Integrated tests

<table>
<thead>
<tr>
<th>Name</th>
<th>Test Number</th>
<th>Cells</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super simple</td>
<td>00</td>
<td>1 TRM, GPO</td>
<td>Statically display a number on the GPO LEDs.</td>
</tr>
<tr>
<td>Simple</td>
<td>01</td>
<td>1 TRM, GPO</td>
<td>Loop to a large number, when number is reached, advance LED</td>
</tr>
<tr>
<td>Double</td>
<td>02</td>
<td>2 TRM, GPO</td>
<td>Two TRMs with different parameters, one producing data, one passing it through to the GPO</td>
</tr>
<tr>
<td>Double</td>
<td>03</td>
<td>1 TRM, UART</td>
<td>Tests UART communication. Not used.</td>
</tr>
<tr>
<td>Chained</td>
<td>04</td>
<td>6TRM, GPO</td>
<td>Source generates pulses, 4 passthrough Cells read the pulse and pass it on, Sink takes final pulse and toggles the GPO LEDs.</td>
</tr>
<tr>
<td>Float</td>
<td>05</td>
<td>1 TRM, GPO</td>
<td>Similar to test 01, but uses floating point arithmetic</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>06</td>
<td>2 TRMs, GPO</td>
<td>Displays fibonacci numbers on the LEDs every time the pulse generator Cell fires.</td>
</tr>
<tr>
<td>GCD</td>
<td>07</td>
<td>4 TRMs, GPO</td>
<td>A problem generator listens to a pulse generator. Every time the pulse generator fires, the problem generator pulls two numbers from a pseudorandom number generator (extra cell), passes them to the GCD calculator (euclid’s algorithm) and feeds the output to the LEDs.</td>
</tr>
<tr>
<td>Display</td>
<td>08</td>
<td>2 TRMs, Timer, LCD</td>
<td>One TRM reads data from a channel and controls the LCD screen, the other TRM feeds the channel.</td>
</tr>
<tr>
<td>Eigval</td>
<td>09</td>
<td>3 TRMs, LCD, Timer</td>
<td>1 TRM generates a test matrix and sends it to another TRM that calculates the eigenvalue. The first TRM then sends the result to a third TRM that prints it to the display.</td>
</tr>
<tr>
<td>Matrix Mul</td>
<td>10</td>
<td>3 TRMs, LCD, Timer</td>
<td>Test generator TRM sends matrix to multiplier TRM, then sends result to display controller.</td>
</tr>
<tr>
<td>Large</td>
<td>11</td>
<td>1 TRM</td>
<td>Code generated by a script, consists of hundreds of lines of the form a[44] := 44;</td>
</tr>
<tr>
<td>Quicksort</td>
<td>12</td>
<td>2 TRMs, LCD, Timer</td>
<td>1 TRM initializes a matrix using pseudorandom numbers, sorts the array then sends the results to a display driver</td>
</tr>
</tbody>
</table>

Table 4.2: Short Description of Test Programs
4.2 Data

4.2.1 Instruction Width

We performed a series of experiments to develop an understanding how different instruction widths influence the use of program memory. In general, shorter instruction width leads to more instruction words, but each word occupies less space. If we assume we can pack instruction words of odd widths efficiently into the given memory primitives, we can measure program length in bits. In practice, our n-bit instruction memory implementation is not perfectly efficient for all instruction widths, so some of the gains are lost. For a more detailed explanation see A.3.

We tried a selection of instruction widths for every program from our test set. We would like to highlight the following results to show a spectrum of behavior with different programs:

- Figure 4.1 shows the impact of the chosen instruction width on assembly size in a very simple program.
- Figure 4.10 shows the same metrics, but measured on our artificial example that is designed to create many literal values of various sizes.
- Finally, 4.11 shows the same for a more realistic program of considerable size, in this case a program that generates random numbers and then sorts them.

We also measured the effect of instruction width on hardware resource use. For this purpose, we checked the occupied LUTs (look up tables) and REGs (slice registers) at different instruction widths. We performed this check on two different Cell nets, one with a single TRM (test 0) and one with 6 (test 4). The results are shown in Figure 4.12 and Figure 4.13.

4.2.2 Floating Point

The impact of the floating point unit is straightforward: Resource use per TRM takes a hit, but the runtime procedures to emulate floating point operations can be omitted. We also save the call to the runtime procedures (backing up registers, setting up stackframe etc.) for each floating point operation. We explore these effects using a simple test case with a single Cell performing a simple multiplication. The results are summarized in Table 4.3. All other options are set to defaults, so instruction width is 18 bits with 2 words per line, multiplication is allowed and nonblocking IO is off. A manual review of the assembly confirms that the extra length of the assembly is entirely due to the additional runtime procedures and a call to the fmul procedure. The call consumes 6 instruction words to move arguments to and from the stack, so the program would grow longer by 6 instructions for each floating point instruction that is emulated. Performance is obviously much worse since single cycle instructions are replaced by more than 80 word long subroutines.

We measured the difference in performance with and without floating point support for all floating point operations. The way we measured captures some extra instructions besides the targeted floating point operation, so all execution time values are slightly overestimated. The error applies to all values equally, so
Figure 4.1: Code density in Test00 with differing instruction width

Figure 4.2: Code density in Test01 with differing instruction width
Figure 4.3: Code density in Test04 with differing instruction width

Figure 4.4: Code density in Test05 with differing instruction width
Figure 4.5: Code density in Test06 with differing instruction width

Figure 4.6: Code density in Test07 with differing instruction width
Figure 4.7: Code density in Test08 with differing instruction width

Figure 4.8: Code density in Test09 with differing instruction width
Figure 4.9: Code density in Test10 with differing instruction width

Figure 4.10: Code density in Test11 with differing instruction width
Figure 4.11: Code density in Test12 with differing instruction width

Figure 4.12: Hardware use in Test00 with differing instruction width
they remain comparable. Table 4.4 shows the results. As we see, the difference is dramatic.

4.2.3 Multiplication

Similar to the modules that enable floating point operations, the multiplication module takes up a considerable amount of hardware resources and its replacement by a runtime procedure is also fairly costly. In Table 4.5 we see that by removing the multiplication unit, we can cut our FPGA resource use almost in half. As a tradeoff, the multiplication runtime procedure occupies about 100 words, with every call to this procedure using another 6 words.

The execution time of the emulated multiplication is much longer than when done in hardware. We performed an experiment to check the difference in prac-
### 4.2 Data

<table>
<thead>
<tr>
<th>Multiplication Unit</th>
<th>REGs</th>
<th>LUTs</th>
<th>Program size in words</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>131</td>
<td>615</td>
<td>34</td>
</tr>
<tr>
<td>OFF</td>
<td>59</td>
<td>562</td>
<td>142</td>
</tr>
</tbody>
</table>

Table 4.5: Resource use of Multiplication Unit

<table>
<thead>
<tr>
<th>Multiplication Unit</th>
<th>5 * 5</th>
<th>0xFFFFFFF * 0xFFFFFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>34 Cycles</td>
<td>34 Cycles</td>
</tr>
<tr>
<td>OFF</td>
<td>105 Cycles</td>
<td>250 Cycles</td>
</tr>
</tbody>
</table>

Table 4.6: Performance impact of Multiplication Unit

For this experiment, we use a timer engine. We simply reset the timer, perform a multiplication, then stop the timer and output the value to the LEDs. This also captures some instructions besides the multiplication itself, but it allows a good comparison between the two situations. The implementation of the runtime procedure uses a shift and add algorithm where the execution time is dependent on the number of set bits in the first factor so we check for a normal case where two small numbers are multiplied and the worst case. Table 4.6 shows the results. In the worst case, hardware acceleration causes a difference in execution time of over 200 cycles. A more optimized version of the multiplication routine could possibly shrink this gap.

### 4.2.4 IO Subsystem

Native support of AXI stream channels is in many ways superior to the previous solution where a wrapper and runtime procedures were used to stimulate the AXI stream channels between cells, see Appendix A.6. We compared hardware resource consumption in a simple design with a single TRM with standard parameters. The results are shown in Table 4.7. Turning off support for non-blocking IO gives a slight improvement in hardware resources.

We have also prepared some experiments to showcase the improved maximum throughput versus the previous solution. For this purpose, we use a design with two Cells: A source Cell that sends data on a channel as fast as possible (hand-crafted assembly) and a Cell that reads this channel in a tight loop with both old and new hardware and the corresponding runtime procedures. We were able to further improve the performance of the new solution by manually inlining the runtime procedure by editing the assembly, which unfortunately leaves some no-ops in the loop. Results are summarized in Table 4.8. Further improvements are possible if we integrate IO functionality more tightly in the compiler.

<table>
<thead>
<tr>
<th>Nonblocking Support</th>
<th>REGs</th>
<th>LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>130</td>
<td>612</td>
</tr>
<tr>
<td>OFF</td>
<td>129</td>
<td>590</td>
</tr>
</tbody>
</table>

Table 4.7: Resource use of Nonblocking IO
24 Experiments

<table>
<thead>
<tr>
<th>IO version</th>
<th>Cycles per loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>old</td>
<td>52 Cycles</td>
</tr>
<tr>
<td>new with runtime</td>
<td>39 Cycles</td>
</tr>
<tr>
<td>new inlining</td>
<td>27 Cycles</td>
</tr>
</tbody>
</table>

Table 4.8: Performance impact of new IO subsystem

<table>
<thead>
<tr>
<th>Version</th>
<th>REGs</th>
<th>LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non blocking IO capable</td>
<td>131</td>
<td>615</td>
</tr>
<tr>
<td>New IO system</td>
<td>131</td>
<td>636</td>
</tr>
<tr>
<td>Flexible Instruction Width only</td>
<td>133</td>
<td>653</td>
</tr>
<tr>
<td>Old without float</td>
<td>129</td>
<td>665</td>
</tr>
<tr>
<td>Old with float</td>
<td>459</td>
<td>1232</td>
</tr>
<tr>
<td>Non blocking IO capable, float on</td>
<td>460</td>
<td>1232</td>
</tr>
</tbody>
</table>

Table 4.9: Resource use of different TRM versions in standard config

4.2.5 Legacy Versus New

In the development of the new hardware, we were always mindful not to introduce unnecessary hardware resource use. As a result, the new hardware with standard parameters is at least as frugal as the old solution. We compare the full feature version and intermediate versions with fewer parameters. The parameters are always set so the design approximates the original non parameterized version as close as possible. Results are summarized in Table 4.9. We see that resource consumption differences between the different versions are negligible.
Chapter 5

Discussion

5.1 Hardware Resources

On an FPGA, all logic is implemented using look up tables and some 'hard' functional units such as adders that are connected by a programmable switching fabric. The block RAM elements that hold our program and data memory and the DSP elements that perform floating point calculations are all hard units. Depending on the FPGA model, the number of these resources available varies. The Xilinx Virtex 5 FPGA we used for this project has enough hardware resources for over 100 TRMs. Still, these resources should not be wasted for several reasons:

- Less resources remain to implement custom logic (Engines)
- The time to synthesize a hardware design is dependent on the resource consumption of the design
- More resources used means more energy consumption
- More chip area used means longer signal paths and lower maximum clock speed.
- If very few resources are used, the design may be migrated to a smaller, cheaper FPGA.

5.2 Code Density

On embedded systems, memory is often very limited. It is therefore important to consider how much memory is needed to fulfill a given task. In our system, size restrictions arise because the program and data memory are placed on the FPGAs integrated block RAMs. Specifically on the version of the Virtex5 FPGA we are using, the total block RAM capacity is 148 blocks of 1024 words each or just over 60 kilobytes\(^1\). If we want to run complex programs or instantiate many cores, we must be careful not to waste this resource. Part of the responsibility is on the programmer writing the Active Cells specification. We can however help by providing options to fit the processor close to the use case.

\(^1\)http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf
5.2.1 Instruction Selection

Often, there is a tradeoff between use of memory and use of other hardware resources (mostly LUTs). For example if we turn off the FPU, we save the logic for fast floating point calculations, but we waste more storage space for the procedure emulating the missing function and the calls to this function. The same is true when integer multiplication is turned off. Emulated operations are also by necessity much slower because we replace a single instruction with several.

5.2.2 Instruction Width

We performed various experiments to evaluate the impact of instruction width on hardware resources and program size. Hardware resources other than block RAM seem to be only lightly influenced by instruction width. Program size and by extension block RAM consumption however is. A program will take up more words when instruction width is shorter because the compiler has to insert extra instructions to get around the shorter immediates. On the other hand, each individual word is shorter, so the program length measured in bits tends to go down with smaller word width. The two effects cancel each other out at different word widths depending on the individual program. In practice we cannot always exploit this sweet spot because our implementation of the n-bit memory is wasteful for certain word widths, see Appendix A.5.
Chapter 6

Conclusions

We have introduced a selection of parameters to a simple processor, the TRM. The selection of the parameter values comes down to a tradeoff between different design goals in all cases.

6.1 Float and Mul

Floating point support and the multiplication unit have similar tradeoffs. Both use up a significant amount of hardware resources which may not be justifiable in applications where the respective features are not performance critical. The absence of these features in hardware increases code memory consumption slightly. This is likely not problematic for most programs.

On the other hand, if floating point arithmetic or integer multiplication is used on performance critical code paths in a cell without the respective hardware support, the program will run much slower. The performance difference may shrink somewhat if the runtime procedures were more optimized, but the procedure will always be much slower than the same operations in hardware. Since in general it is hard for a compiler to determine which code paths are important for the programmer’s goals, we think it is best to leave the decision over the use of these modules to the programmer.

6.2 Nonblocking IO

Support for nonblocking IO leads to a very small change in hardware resource usage. It is however very easy for the compiler to figure out whether this support is needed: The parameter must be set if a nonblocking communication statement exists in a Cell, otherwise it is superfluous. We therefore propose that the selection of this parameter be automatized in future versions of the tool chain.

6.3 Instruction Width

A wider instruction width leads fewer assembly instructions, but every single word occupies more memory. In most programs, the number of words stops sinking at a certain instruction width. From that point on, a wider instruction
width is pure waste because the bits per word increase without leading to fewer words. The total number of bits is usually at a minimum before the number of words completely stops decreasing. For none of the programs we tested is this minimum reached at an instruction width of 18 bits which was the only option previously. The relationship between instruction width and number of words depends on the immediates and jump offsets in the assembly. Most realistic programs lead to relatively few immediates and those that do show up are often small values that already fit into narrow instruction words. Jump offsets however occur very frequently in more complex programs and their values depend on the total program size and the arrangement of basic blocks on which the programmer has almost no influence.

Since the impact of instruction width on a particular program’s memory footprint is so hard to predict, we recommend trying multiple widths, but only when standard settings do not lead to satisfactory results, similar to the normal optimization workflow.

6.4 Clock Speed

We checked the tool outputs for ’maximum clock speed’ with various designs and parameters. Maximum clock speed is calculated from the longest signal path from register to register. Unfortunately, this bottleneck does not appear to be correlated with the selection of any of the parameters, so we could not see any improvements in clock speed. Still, the smaller footprint of properly parameterized cells can make the place and route tasks easier. This could save large designs that occupy most of the FPGA from having to reduce clock frequency.

6.5 Is Parametrization Practical?

We would certainly recommend using parameters in a microprocessor implementation on an FPGA.

For complex processor designs with many pipeline stages, caches, branch prediction and other advanced features, it may be hard to include some parameters, but on an FPGA high performance tasks should be executed by a dedicated circuit anyway. Simple processors like our TRM are relatively easy to parameterize. Carefully chosen parameter values can lead to significant improvements versus a one size fits all solution. In our case, we were able to compare the parametrized design versus the previous fixed version and show that the parameters do not lead to inherently less efficient hardware designs. The parameterized HDL source code is certainly more complex, but it is more manageable in the long run than multiple versions of the same module. A parameterized processor is therefore a winning proposition if one is willing to spend the extra work needed to introduce parameters.
Chapter 7

Perspectives

We have taken a few first steps towards a system of compiler, processor and framework that can generate the best hardware and code for a process description in a high level language. Our current system still only has a small number of options to configure the processor and they all have to be set by hand by the programmer. Also the compiler still has much room to improve. We think the following features could be of interest, in no particular order:

- **Cache size**: For this, we would first have to introduce a memory hierarchy though with program and data memory in DRAM. It would probably make sense to create a new architecture for this though instead of radically changing the TRM.

- **Pipelining**: There were some previous experiments to introduce pipelining into the TRM architecture. This seems to be an interesting target for an optional feature, but the TRM does not seem to benefit much from it in terms of maximal clock speed.

- **Save opcode bits**: When enough instructions are never needed, we could save a bit of opcode, or conversely introduce more opcode bits to enable additional instructions. This would entail reassigning all opcodes both in software and hardware. It may be necessary to move use some generated Verilog code to realize this.

- **Extra instructions**: Together with the previous extension, a selection of new instructions could be introduced such as conditional moves, memory to memory operands etc. To properly exploit these, the compiler would have to change significantly. We would also have to start changing the opcode width if there are more than 16 distinct instructions.

- **In general, an optimizing compiler that ideally also chooses the hardware parameters automatically would really help make the parameters practical. Currently, the programmer has to either know the hardware in detail or try many parameter combinations.**

- **At some point we were contemplating a completely different architecture that consisted of a minimal controller core and multiple instruction modules with standardized interfaces. This system could also include other**
components such as DMA engines which would enable a cell to saturate a hardware engine with data.
Appendix A

Hardware Details

A.1 TRM Instruction Set

A.1.1 Instruction Formats

Common

All instructions start with a 4 bit opcode occupying the top four bits. The user selectable instruction width defaults to 18 bits. In the instruction formats, we denote the selectable width as \( w \).

<table>
<thead>
<tr>
<th>w-1</th>
<th>w-4</th>
<th>w-5</th>
<th>w-8</th>
<th>other bits</th>
<th>0</th>
</tr>
</thead>
</table>

Register Instructions

All register instructions perform an operation between a source register or immediate and a target register. The result is stored in the target register. Bit \( w - 7 \) signals what the lower \( w - 8 \) bits signify. If it is 0, the instruction tail contains a \( w - 8 \) bit immediate. If it is 1, the lowest 3 bits designate a source register. In the latter case, some bits are unused. Some of them have meaning in the floating point and vector versions of the instruction set: Bit \( w - 11 \) signals a vector instruction, bit \( w - 12 \) a floating point instruction.

<table>
<thead>
<tr>
<th>w-1</th>
<th>w-4</th>
<th>w-5</th>
<th>w-8</th>
<th>1</th>
<th>Immediate</th>
<th>0</th>
</tr>
</thead>
</table>

This dictates the minimum width of an instruction word: \( 4 + 3 + 1 + 1 + 1 + 3 = 13 \) bits. If we disallow vector instructions, we can go down to 12, if we assume we have no floating point support either, we can go down to 11. Immediates for register instructions would then be restricted to just 3 bits. In practice the compiler fails on most programs if the instruction width drops under 14 bits.

Load/Store

The load and store functions calculate a data memory address from the contents of the source register and an 8 bit offset. The load instruction then sets the des-
destination register to the memory contents at that address. The store instruction
instead reads the contents of the destination register (a misnomer in this case)
and stores it to the data memory at the calculated address.

<table>
<thead>
<tr>
<th>w-1</th>
<th>w-4</th>
<th>w-5</th>
<th>w-8</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>trg</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When we change the instruction width, the offset changes size correspondingly.

**Branch**

The control flow is defined with a selection of branch instructions. All branch
instructions except the function return use instruction pointer relative addresses
as jump targets.

**Conditional Branches** There are a total of 16 different conditions encoded
in the 4 bits below the opcode.

<table>
<thead>
<tr>
<th>w-1</th>
<th>w-4</th>
<th>w-5</th>
<th>w-8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>cond</td>
<td>offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The condition code consists of 3 bits to select a condition and 1 bit to switch
to the negation of the condition. For example ‘equal’ and ‘not equal’ have
the same first 3 bits of condition code. The conditions are evaluated from the
various flag registers set by the arithmetic unit.

<table>
<thead>
<tr>
<th>Code (bin)</th>
<th>Mnemonic</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>equal(zero)</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>not equal</td>
</tr>
<tr>
<td>0010</td>
<td>CS</td>
<td>carry set</td>
</tr>
<tr>
<td>0011</td>
<td>CC</td>
<td>carry clear</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>negative (minus)</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>positive (plus)</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>overflow set</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>overflow clear</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>high</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>less or same</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>greater or equal</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>less than</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>greater than</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>less or equal</td>
</tr>
<tr>
<td>1110</td>
<td>T</td>
<td>true</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>false</td>
</tr>
</tbody>
</table>

When we added support for nonblocking IO, we also introduced a flag register
to remember success or failure of the nonblocking IO operation. This flag is
considered when evaluating the EQ/NEQ condition.

**Branch and Link** The branch and link instruction takes no condition, instead
the branch offset is 14 bits long. The instruction causes a jump to position ‘In-
struction pointer+offset’ and stores the current instruction pointer in a register.

<table>
<thead>
<tr>
<th>w-1</th>
<th>w-4</th>
<th>w-5</th>
<th>w-8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A.2 Floating Point

Special
There are also special instructions to fetch the high 32 bits of integer multiplication and fetch the interrupt status. They are implemented as special move instructions. Since the high bits are rarely needed and interrupts are not used in the Active Cells system, we don’t have to go into detail here.

A.1.2 Floating Point
Floating point values require some special instructions. In the TRM, floating point values are only supported in register operations without immediate. This way, the instruction set exploits that there are some unused bits in that kind of instruction. By convention, bit w-8 signals the absence of an immediate, so it’s always 1 for floating point operations. Bit w-9 signals a vector operation in a specialized version of the TRM which we did not use. Bit w-10 marks the presence of a floating point operation. If the hardware does not support floating point instructions, these extra flag bits are ignored.

A.2 Floating Point

A.2.1 Old System
In the previously used approach, floating point support was toggled by including different hdl source files for the TRM in the autogenerated project. We noticed that when both Cells with and without floating point unit were present, the project contained two sources for the same module name which causes errors and warnings when building by hand. Due to a quirk in the synthesizer, when building through the batchfile generated by the Active Cells pipeline, the source with floating point support has precedence and all TRMs in the Cell net acquire floating point support. Note that this wastes hardware resources. Oddly when the hdl project is opened with the Xilinx ISE editor, the precedence order of the sources is reversed and thus no Cell gets floating point support when we build this way. This is obviously not the intended behavior for this system. Also, juggling different versions of the TRM is a clunky process prone to failure since changes to one version are not automatically carried over to the other. We therefore revamped the floating point support completely.

A.2.2 New System
In the new system, all TRMs are generated from the same source with different Verilog parameters. We use Verilog parameters to remove unneeded features. To this end, we exploit that Verilog parameters are evaluated before synthesis starts. Therefore, if we use the parameter to set all inputs to a module to zero and ignore the outputs, the module will be removed after synthesis in the netlist trimming phase. Here is an example showing the floating point multiplier and adder:

```verilog
wire [DW-1:0] fpA, fpB;
//if no FP support, inputs fixed to 0
assign fpA=(FloatingPoint==0?0:A);
```
assign fpB=(FloatingPoint==0?0:B);

//instantiating floating point multiplier
fp_multiplier #(.IN_DELAY(0),
    .INT_MULTIPLY_DELAY(2),
    .OUT_DELAY(1))
i_fp_multiplier(
    .clk(clk), .rst(rst&FloatingPoint!=0),
    .A(fpA), .B(fpB),
    //if no FP support never start,
    //else at FMUL instruction
    .start(FMUL & ~FMUL_reg & (FloatingPoint!=0)),
    .R(fp_mul_R), //result
    .overflow(overflow_fmul),
    .underflow(underflow_fmul),
    .busy(),
    .out_valid(fmul_ready)
);

//FP adder/horizontal adder
//2+0+1 = 3 / 8 cycles for FP addition.
//non-pipelined (8 cycles)
fp_selectable_adder
    #(.PIPELINED(1'b0), .WITH_DSP_SLICE(0),
    .IN_DELAY(2), .INT_ADD_DELAY(0),
    .OUT_DELAY(1))
i_fp_adder {
    .clk(clk),
    .rst(rst&FloatingPoint!=0),
    .A(fpB),
    .B(fpA),
    .operation(FSUB), //FSUB
    //if no FP support, never start,
    //else FADD or FSUB
    .start((FADD | FSUB) &
        ~FADD_FSUB_reg &
        (FloatingPoint!=0)),
    .R(fp_add_sub_R),
    .overflow(overflow_fadd),
    .underflow(underflow_fadd),
    .busy(),
    .out_valid(fadd_ready)
);

A.3 Optional Multiplier

For the optional Multiplier, we use a similar trick. Because the multiplication
is instantiated differently, we can use a slightly simpler technique. Instead of
setting all in- and outputs to the multiplication unit, we just tie the signal that
marks the presence of a multiplication statement to zero when multiplication
A.4 Instruction Width

is forbidden through the parameter. This is sufficient to make the synthesizer recognize the multiplication unit as dead and remove it in the pruning phase.

```markdown
\\ assignment to the MUL signal
\note MUL is always 0 if HasMul is 0
assign MUL = (op == 8) & (¬IR[ISIMM] | ¬IR[ISFLOAT]) & HasMul;
```

```markdown
\\ Instantiation of MUL unit
\Note Reset is always active when MUL is constant 0
Multiplier mulUnit (.CLK (clk),
.RST (¬MUL),
.A ({{3{A[31]}}}, A)),
.B ({{3{B[31]}}}, B)),
.stall (stallM),
.mulRes (mulRes));
```

A.4 Instruction Width

The instruction width parameter must be taken into account in almost all statements that deal with the instruction word or parts thereof. For example, the instruction word as most of the TRM sees it used to be declared as:

```markdown
register IR[17:0]
```

But now it is

```markdown
register IR[insW−1:0]
```

Decoding uses offsets from the MSB or LSB to find features in the instruction word. The offsets themselves are represented as local parameters and combinations thereof. The local parameters are defined as follows:

```markdown
// nonzero to activate fpu
parameter FloatingPoint = 0;
// zero to disable integer multiplication
parameter HasMul = 1;
// number of IM (instruction mem) blocks of 1024 l each
parameter IMB = 2;
// number of DM (data mem) blocks of 1024 words each
parameter DMB = 2;

parameter BaseMem = 1;
parameter BaseDiv = 1;

localparam OPCODEW=4; // width of opcode
// number of bits to select a register.
localparam REGSELW=3;
// instruction width.
parameter PW = 18;

// nonzero to enable
Hardware Details

```verilog
parameter HasNonBlockingIO = 0;

// is immediate? bit 10 in TRM0
localparam ISIMM = PW−1−OPCODE_W−REGSEL_W;
// start of tail of the instruction.
// may contain the immediate
localparam OPTAIL = ISIMM − 1;
// is vector instruction? bit 9 in trm0
localparam ISVEC = ISIMM − 1;
// is float instruction? bit 8 in trm0
localparam ISFLOAT = ISIMM − 2;

// immediate for arith. instructions is
// [OPTAIL:0] (18 bit PW⇒ 10 bit)
// offset for load/store is
// [ISIMM:REGSEL_W]( 18 bit PW⇒ 8 bit)

// old NOP: hardcoded 'never jump'
// new NOP: assembled from localparams
localparam NOP = 18'b11111111110000000000;

// output of several block RAM primitives, then divides this line into words of
// which it selects one using the last address bits. The current implementation
// can assemble at most 3 RAM primitives side by side, but it should be simple
// to enable more. Figure A.1 shows an example where three RAM blocks side
// by side are divided into 4 words. The RAM primitives themselves are also not
// instantiated directly but through a wrapper that logically places several RAM
// blocks behind each other in case the program is too long. A single RAM block
// on the Xilinx Virtex5 FPGA we used holds 1024 lines of 36 bits each.
// Because we wrote the logic to read from the RAM primitives ourselves, we can simply
// perform the reverse in the compiler when it generates the memory initialization
// files from the assembly.
```

A.5 N Bit IM

Our n bit instruction memory implementation assembles a line from the data output of several block RAM primitives, then divides this line into words of which it selects one using the last address bits. The current implementation can assemble at most 3 RAM primitives side by side, but it should be simple to enable more. Figure A.1 shows an example where three RAM blocks side by side are divided into 4 words. The RAM primitives themselves are also not instantiated directly but through a wrapper that logically places several RAM blocks behind each other in case the program is too long. A single RAM block on the Xilinx Virtex5 FPGA we used holds 1024 lines of 36 bits each. Because we wrote the logic to read from the RAM primitives ourselves, we can simply perform the reverse in the compiler when it generates the memory initialization files from the assembly.

```verilog
// dout ports of the 3 underlying RAMB36
wire [35:0] base0;
wire [35:0] base1;
wire [35:0] base2;
```

localparam baseW =
  baseSel == 0? 36:
  baseSel == 1? 72:
  108;
wire [baseW−1:0] base;
// assemble composite base.
// base0 yields lowest order bits.
assign base =
  baseSel == 0? base0 :
  baseSel == 1? {base1, base0} :
  {base2, base1, base0}
;
// divide base into parts
localparam baseDivW = baseW / finalDiv;
wire [baseDivW−1:0] finald0, finald1, finald2, finald3;
// first chunk (lowest order bits of BASE)
assign finald0 = base [baseDivW−1:0];
// second chunk or constant zero
assign finald1 = finalDiv > 1? base [2*baseDivW−1:baseDivW−1]: 0;
// third chunk or constant zero
assign finald2 = finalDiv > 3? base [3*baseDivW−1:baseDivW+2]: 0;
// fourth chunk or constant zero
assign finald3 = finalDiv > 3? base [4*baseDivW−1:baseDivW+3]: 0;
wire [finalW−1:0] final;
// determine what to put into result based on finalDiv
  // parameter and last bits of address
localparam omittedBits =
  finalDiv == 1? 0:
  finalDiv == 2? 1:
  2;
// Ensure at least 1 bit selector signal

Figure A.1: Example demonstrating 4 words per line distributed into 3 columns.
Hardware Details

// Necessary to compile.
localparam selectorW=omittedBits+(omittedBits==0?1:0);
reg [selectorW-1:0] selector =0;
always@(posedge clk) begin
  if (omittedBits==0)
    selector <=0;
  else
    selector <=pmadr[selectorW-1:0];
end
assign final=
  finalDiv==1? finald0[finalW-1:0]:
  finalDiv==2? ( 
    selector==0?finald0[finalW-1:0]:
    finald1[finalW-1:0]
  );
  finalDiv==4
  ( 
    selector==(2'b00)?finald0[finalW-1:0]:
    selector==(2'b01)?finald1[finalW-1:0]:
    selector==(2'b10)?finald2[finalW-1:0]:
    finald3[finalW-1:0]
  );
assign pmout=final;

A.6 IO Subsystem

The TRMs old IO system would always write to the IO port when a memory
write to a specific address took place. In the Active Cells system however, we
connect cells using AXI stream ports in which a write is only legal if both the
receiver signal it’s ready and the sender flags its data as valid at the same time.
Therefore the old system had to use a wrapper to translate between the native
IO and AXI stream. Specifically, a memory write or read to the channel Address
(0xFFFFFC0) would cause the contents of the source register to be posted on
the data lines and the last 6 bits of the address to be posted as channel selector.
If they were zero, the word would be written straight to the AXI channel, ready
or not. To ensure the channel was ready before sending, the TRM had to first
make a memory read where the top bit of the channel selector is set. Conversely,
a receive operation first checked for the presence of valid input data by doing
an IO read such that the channel selector’s second to top bit was set.
Consequently, there were only 4 of the originally 6 bits left to actually select be-
tween channels. Also, waiting for valid data or ready output was accomplished
by the TRM spinning in a tight loop. In the worst case, this behavior may even
cause the TRM to write to an unready channel because the AXI standard does
not guarantee that ready channels stay ready, so the ready signal could have
dropped between the loop and the actual send operation.
In our new system, the IO subsystem is directly connected to an AXI stream
in and output port, each 32 bits wide and with a complementary 6 bit channel
selector. An IO write operation now causes the TRM to stall until the output
channel is ready and an IO read causes a stall until the input channel shows valid data. With this mechanism, it is possible to write up to one word per cycle provided the data to write is in a register.

Support for nonblocking IO can be toggled through the parameter HasNonBlockingIO which, like the other parameters, is set at the instantiation of the TRM.

//dmadd=data memory address
//DAW=Data Address Width. Data addresses have DAW bits
//note if dmadr[DAW] is set, the address is beyond legal //data mem.
//IO writes can only happen for writes without immediate

assign blockingIO= dmadr[DAW] &dmadr[6]& ¬IR[ISIMM];
assign blockingSend=blockingIO&ST;
assign blockingRec=blockingIO&LDR;

//send:
// pass value to out
// stall until outReady
// as long as we are in this state, valid stays high
assign outData=B;
assign outDest=dmadr[7:0];
assign outValid=blockingSend;
//if out ready, outValid will stay high for exactly 1 cycle

//if we're trying to send now, stall until success
//processor will stay in this state until success
assign IOS Stall=(blockingSend&¬outReady)
  | (blockingRec&¬inValid);

//receive: stall until inValid
//register file reads the inData when blockingRec set.
assign inReady=blockingRec;

With support for nonblocking IO, the code gets more complicated, but performs the same function in principle. When a nonblocking IO operation is performed, the nonBlokingFailure register gets set. This register is consulted when evaluating a jump condition in the next cycle.

assign blockingIO= dmadr[DAW] & ¬IR[ISIMM] &
  !(HasNonBlockingIO&dmadr[nonBlockingFlagLoc]);
assign blockingSend=blockingIO&ST;
assign blockingRec=blockingIO&LDR;
assign outData=B;
assign outDest=dmadr[7:0];
assign inDest=dmadr[7:0];
assign outValid=blockingSend| nonBlockingSend;
assign IOS Stall=(blockingSend&¬outReady)
(blockingRec&¬inValid);

// inValue is handled regmux assignment
assign inReady=blockingRec|nonBlockingRec;

// if HasNonBlockingIO is not set
// all the nonblocking related signals are constant zero
assign nonBlockingIO=dmadr[DAW]
 & ¬IR[ISIMM] & HasNonBlockingIO
 &dmadr[nonBlockingFlagLoc];
assign nonBlockingRec=nonBlockingIO&LDR;
reg nonBlockFailureReg=0;
always@(posedge clk)
 nonBlockFailureReg<=nonBlockingRec
 &¬inValid | nonBlockingSend
 &¬outReady;

assign nonBlockFailure=nonBlockFailureReg;
assign nonBlockingSend=nonBlockingIO&ST;
Appendix B

Compiler Details

B.1 Overview

The compiler infrastructure for Active Cells is based upon Oberon’s Fox Compiler. This compiler has a strong division between frontend and backend. As usual, the frontend contains the parser and checker. The resulting syntax tree is then handed to one of several backends. For an Active Cells program, there are three backends involved. First intermediate generates intermediate code which is then reparsed so the TRM backend can create the TRM assembly. Then, in a separate step (new user command), the Active Cells tools module reads the hardware specification, makes the HDL tools generate an actual hardware design for the FPGA (bitstream), then patches in the program and data memory contents and finally programs the FPGA. If the hardware design has not changed, the hardware generation steps are left out and the tools skip straight to patching the bitstream.

B.1.1 Frontend

We only had to change the frontend to allow the new boolean parameter for optional multiplication. No changes had to be made for all other new parameters since integer valued parameters can already have arbitrary names. They are represented as a ’parameter’ child of a ’cell’ node in the syntax tree and contain name and value of the parameter as strings. We do however have to check for the names of the new parameters when we prepare the Active Cells specification. In the semantic checker, the parameter values get copied to the Active Cells specification. Later parts of the compilation can then check for the values of these new features in the Active Cells specification if compilation depends on them.

B.1.2 Backends

The backends are tasked with turning the syntax tree into assembly.
B.1.3 Intermediate and TRM Backend

The TRM backend is responsible for the programs that will run in the instantiated Cells that are implemented as TRMs. It works closely with the Intermediate backend, because the intermediate backend currently has to check the feasibility of immediates and call runtime procedures etc.

B.1.4 Active Cells Backend

The Active Cells backend builds the Active Cells specification that essentially defines the hardware.

B.1.5 Hardware Descriptors

The mapping from data data structures in the compiler to real hardware currently works through a collection of xml files. We had to add the new parameters to these hardware descriptors. To give the system some compatibility with older hardware descriptors, we set the default values for the new parameters in the semantic checker.

B.1.6 Linker

The linker takes the prepared section objects and assembles them into a continuous instruction stream. For this it uses functions from the static linker.

B.1.7 Static Linker

The static linker reads the instruction stream and writes a temporary binary file with one word per line. This file is read later when the initialization files for the program and data memory are prepared. In previous versions, this intermediate file already stored two words per line.

B.1.8 Hardware Generation and Deployment

When the user tries to deploy a design for the first time or has changed something in the Active Cells specification that requires a hardware change, the toolchain must generate a new bitstream to configure the FPGA. In this phase, the core compiler is not used any more. Instead, the Active Cell Tools module reads the Active Cells specification that was generated in the final compilation step. The specification is first modified to get rid of hierarchical nested networks (flattening). The flattened specification is then turned into a top level Verilog source that instantiates the various TRMs, engines and channels. The exact parameters and connections of every module are currently defined in a set of XML files that the hardware generator reads beforehand. When the top level source is generated, a project file for the synthesis tools is generated. This project file points to all hdl sources that are used in the specified Cell net. The hardware generation module then starts the synthesizer and waits until a bitstream is ready.

To initialize the program memory contents, another command line tool is used to patch the bitstream with memory files. These memory files are prepared beforehand by reading the binary file from the static linker and splitting it. If
n-bit memory is used, we first split the file into column files as mentioned in A.5. These column files are then split into memory files of 1024 lines each to fit the block RAM primitives of our hardware. The aforementioned command line tool then reads these files and patches their contents into the bitstream for the FPGA.

### B.1.9 Integrating Instruction Width

To support the n-bit instruction width, we had to change some core structures of the compiler. Previously, the instruction set was represented by a globally accessible module that held a static list of instruction formats. We changed this to a proper object. When a backend wants to output assembly or use any other facility of the instruction set, it first has to generate an instruction set object with the appropriate instruction width. This works fine because the compilation steps through the program Cell by Cell. The greatest complication was that we have to pass the right instruction width on to functions used in a Cell even if they come from different files, such as the runtime procedures.

### B.1.10 Integrating Memory Configuration

We have not automated the selection of instruction memory parameters. This means the Active Cells programmer has to choose how many 'side by side' block RAM elements are combined into lines and into how many parts that line is divided. This gives the programmer more control and is more suitable for testing the system. The integration itself is straightforward, the programmer supplied parameters are simply forwarded to the hardware generator. The hardware generator inserts the parameter values when it writes the top level Verilog module. When it is time to initialize the program memory, the hardware generator copies the assembly file over and splits it into column files of 36 bit width each using the reverse process the hardware uses to select an instruction word at runtime. In detail, a static linker function is used that takes instruction words, pads them, assembles them to lines and then divides these lines into 36 bit chunks that are then written to the appropriate files, as shown in Figure A.1. These column files are then further split into files that initialize a single memory block which on our FPGA is 1024 lines long.

### B.1.11 Changes to Floating Point Support

As mentioned in chapter 3.1 we changed the mechanism for the optional floating point support. Previously, the whole verilog source for the TRM was replaced, now we just pass an appropriate Verilog parameter. Since a mechanism to make floating point support optional was already present, we only had to make few changes in the XML hardware descriptors to support the new approach in the compiler.

### B.1.12 Integrating the Optional Multiplication Unit

To further explore hardware customization, we examined which parts in the instruction set we could turn off when not needed similar to the FPU. Since the instruction set is already very basic, the only promising candidates to remove
were the barrel shifter and the multiplication unit. Since shifts are used in many places in the compiler and the multiplication unit consumes more hardware resources than the barrel shifter, we decided to choose the multiplication unit. In Practice, this means we have to introduce a new parameter to the hardware module. The implementation itself is straightforward, we simply assign constant zero to the signal that signifies a multiplication operation when the parameter is set. To get the parameter through the compiler infrastructure, we add a parameter declaration to the TRM architecture description XML. When the multiplication unit is absent, we replace the multiplication instruction with a runtime procedure that uses a loop with shifts ans adds to emulate the multiplication.
Appendix C

Persistent Issues

During the changes to the system, we encountered a number of persistent issues, mostly with the compiler.

- The generation of intermediate code is not strictly necessary in our system, after all it is generated, parsed and translated into assembly following a single user command.

- Sometimes, the compiler continues to execute even after a function in the backend causes a trap. It then usually runs into a trap later in an unrelated part. The original trap still shows up in the error log, so the issue is an annoyance to the compiler developer at most.

- Some n-bit assembly is emitted while building the intermediate code. It appears to be mostly related to various initializations.

- Much more 'section' objects are generated than actually needed. When the constructor of the section objects is instrumented, it becomes obvious that many more section objects are generated than sections occur in the final program. This is obviously wasteful and makes the concept these objects stand for unclear.

- Long jumps that go backwards sometimes lead to infinite loops. This problem occurs occasionally in moderately complex programs compiled to a very short instruction word width. As a consequence of the short instruction word, the immediate for relative jumps is also very short and so the fixup mechanism that tries to enable jumps longer than supported by the jump offset bits.

- The register bank of the TRM is instantiated from hardware primitives as 32 columns consisting of a 16 bit dual port RAM hardware primitive. This forces the synthesizer to always use the hardware resources for 16 registers. However, 8 of these registers are set aside for the use during an interrupt which in the active cells system can never happen, the interrupt lines are not connected to anything. Due to the way the registers are instantiated, the unnecessary registers can not be trimmed away and waste resources. This could be remedied by changing the instantiation of the register file, either to hardware primitives that actually match our use or
some more generic solution. For example we could instantiate the registers as a generic Verilog array so the synthesizer can infer the hardware needed. Even if that for some reason should not work, we could at least completely get rid of interrupts and make the extra 8 registers available as normal registers. This would call for an extra register selection bit which leads to a new potentially interesting trade off.

- The maximal number of channels going in or out of a TRM is currently limited to 256 in the hardware, but the compiler does not support more than 16. This is a left-over of the old IO subsystem which had originally designated 6 address bits as channel addresses, then reserved 2 of them to access the valid and ready bits of AXI channels. The hardware could support even more channels since the channel selection comes from an IO address which in turn comes out of a 32 bit register. All addresses that do not point into a cell’s data memory are potentially available as IO addresses. Since the total amount of block RAMs on a high end FPGA currently totals to just a few megabytes, there is no good reason to limit ourselves to such a small IO space. The only limitation in practice should be the ability to select from so many data sources within a single cycle, which is synthesized as a multiplexer. Since large multiplexers are relatively slow, this will eventually run into timing problems. The system currently is not close to this boundary though, so there is much potential.

- The way we currently assemble the n-bit instruction memory is not entirely satisfactory, because we still rely on hardware primitives which are vendor and even chip specific. There are some methods to instantiate and initialize memory elements that could get around this limitation.
Appendix D

Changelist

- TRM0.v and FTRM.v replaced with TRMNINIONB.v
- TRMNIO.v introduced. New wrapper for the TRM, basically a straight pass-through
- NBitIM.v introduced. Contains the logic to stitch side by side base columns together
- 36bitBase.v introduced. Contains logic to stitch blocks to columns
- TRM-XC5V-XC6V.achc.xml changed
- XUPV5.achc.xml introduced. Architecture for the XUPV5 LX110T board.
- Lcd.achc.xml introduced
- LCD.v introduced
- ActiveCellsComponents.Mod changed
- AxisChannels.Mod changed
- Fox2.Tool changed (compile order)
- FoxActiveCells2.Mod changed
- FixActiveCells2Backend.Mod changed
- FoxIntermediateLinker2.Mod changed
- FoxTRMAssemble.Mod changed
- FoxTRMBackend.Mod changed
- FoxTRMTools.Mod changed
- TRMRuntime.Mod changed: added MUL procedure
- FoxBackend.Mod changed
- FoxGlobal.Mod changed
- FoxSemanticChecker.Mod changed
- StaticLinker.Mod changed
Bibliography


Appendix E

Task Description and Declaration of Originality
Objective

The Active Cells approach to reconfigurable computing features a programming model and a tool-chain that permits a mapping of data-stream type applications to systems on chip on FPGAs. The system on chip consists of compute kernels and communication channels. There are two types of kernels: fully fledged yet simple multi-purpose processors and specialized engines. Processors can be chosen from some variants such as with or without Floating Point or Vector Unit.

The comparison of different processors that have been developed in this context reveals the different strengths and weaknesses of the approaches chosen. For example, the TRM (Tiny Register Machine) is very well suited for small programs and provides a high code density for small-sized programs because of its 18-bit instruction set. However, for larger programs 32 or even 36 bits may have been the better choice. The TRM0 is a particularly small processor that comes without divider, multiplier and interrupt subsystem and is therefore particularly resource efficient, provided that the software does not require any of the “missing” features. TRM0 has been used for an implementation of a DDR-RAM driver that did not require such features.

The choice of the processor was, until now, driven by the programmer’s choice by ways of a property. For example

\[
\text{Adder} = \text{cell} \{\text{FPU}\} (\text{in1, in2: port in; outPort: port out}) ...
\]

Such a property has implications on both, the generated hardware on FPGA and the generated instruction stream.

At the core of this thesis is the design and implementation of a reconfigurable, parameterizable processor on FPGA. A parameterisation of a processor will have implications on the code that can run on the processor, i.e. it is unavoidable that the compiler backend is changed in parallel. We currently envision the following steps (increasing difficulty, not all would be part of one master thesis)

1.) Generic, unified interface in Verilog for different processor types, such as TRM, TRM0, TRM with / without FPU etc.
2.) Adaption of the compiler tool-chain in order to be able to parameterize the code generation and processor implementation (existing but not with respect to the unified interface)
3.) More fine-grained features of a processor. For example instruction bit widths. Controlling compiler backend with such features. This constitutes the most innovative part of the thesis.

4.) Experiments with different variations of code / parameters, measurement of used resources. Objective: find a heuristic for an automatic choice of such parameters.

5.) Automated choice of the “right” processor according to a given target

Minimally required is the implementation of one processor type that is configurable with respect to one major characteristic such as the instruction width or such as processor capabilities. For such a processor the compiler backend has to be adapted accordingly.

Assignments of Tasks
The following tasks should be finished during the master thesis

1. Getting acquainted with the tool-chain / development environment. [6 Weeks]
   a. Getting familiar with the architecture of TRM [1 Week]
   b. Implementation of a first simple version of a parameterizable processor (e.g. FPU on / off) in Verilog / adapt tool-chain (Monolithic versus Component-Based) [3 Week]
   c. Determine how fine-grained this can be done in a first step [2 Week]

2. Implementation of a parameterizable processor at a suitable granularity. In principle two possible approaches [10 Weeks]
   - Fix an instruction set encoding, vary with connected components
   - Design of a finer grained configurability of the processor instruction set
     a. Adaption of the compiler tool-chain w.r.t. instruction set / special instructions required

3. Case study [6 Weeks]

4. Thesis writing [4 Weeks]
Eigenständigkeitserklärung


Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

A PARAMETERIZABLE PROCESSOR FOR RECONFIGURABLE COMPUTING ON FPGAS

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):
KOSTER

Vorname(n):
STEPHAN

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigett“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum
Zürich, 18. Mai 2015

Unterschrift(en)

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie für die Verfassung der Arbeit.