The Creation of a Flexible, Functional Simulation Generator for the Montium Tile Processor

Master's Thesis
by

L. Ordelmans

July 2, 2007

graduation committee:
Prof. Dr. Ir. G.J.M. Smit
Dr. Ir. A.B.J. Kokkeler
Dr. Ir. L.T. Smit
Ir. K.L. Hofstra

Computer Architecture for Embedded Systems
Department of Computer Science
University of Twente

Voorwoord

Dit verslag beschrijft het werk dat ik heb uitgevoerd ter afronding van mijn Master opleiding Computer Science aan de Universiteit Twente. Het beschrijft het ontwerp en de implementatie van een simulator voor de Montium, een herconfigureerbare chip, waarbij gebruik gemaakt wordt van code generatie.
Ik heb veel plezier gehad tijdens het werken aan deze opdracht, met name toen de eerste delen begonnen te werken en ik de principes onder de knie kreeg.
Toen ik begon met afstuderen zou ik eigenlijk een andere opdracht uitvoeren, het was de bedoeling dat ik een DSP algorithme zou gaan *mappen* op de zojuist genoemde chip. Vrij snel ontdekte ik echter dat deze opdracht mij helemaal niet lag, en na een tijd getwijfeld te hebben besloot ik hiermee bij mijn begeleider aan te kloppen, en had ik mij er reeds bij neergelegd dat ik opnieuw op zoek moest naar een afstudeerplek. Tot mijn aangename verassing bleek het team van Recore Systems best bereid was om samen met mij een nieuwe opdracht uit te zoeken. Na ongeveer een goed deel van de middag ideeën te hebben uitgewisseld, ben ik mij vervolgens gaan verdiepen in de nieuwe opdracht, waarvan u het eindverslag nu aan het lezen bent. Ik wil dan ook graag de mensen van Recore Systems bedanken voor het mogelijk maken van deze opdracht en alle ondersteuning tijdens mijn afstudeer periode. Met name Lodewijk, Klaas, Paul en Gerard bij wie ik terecht kon met vragen over mijn opdracht danwel over de Montium, maar ook de rest van het team voor de gezellige tijd. Tevens wil ik natuurlijk mijn begeleiders van de UT, Gerard Smit en André Kokkeler bedanken voor hun inbreng tijdens de maandelijkse voortgangsgesprekken.
Ter afsluiting wil ik nog bedanken mijn moeder en Inge, mijn vriendin, voor hun steun tijdens mijn toch best lange studieweg van ruim 6 jaar.

Enschede, Juni 2007
Luke Ordelmans

Abstract

Simulation is an important tool for (DSP) software developers. In order to test, debug, analyze and improve algorithms the developer needs to be able to see how his work gets executed by the target system. This thesis describes the design and implementation of a functional simulator created for the Montium Tile Processor, a domain specific reconfigurable accelerator, that makes use of code-generation to achieve the speed of a binary compiled simulator but preserves the flexibility of an interpretive one. The simulation generator uses a binary configuration compiled for the Montium TP together with some (optional) design parameters of the specific Montium instance to generate program source code that is compilable on a general purpose desktop computer. By implementing this simulation generator in Java SE 6, the program will be portable between different machines and operating systems. Another benefit from this choice is that it becomes possible for the generated simulation to be compiled and instantiated internally (without leaving the generator to start a external compiler) and instantly, totally invisible for the end-user, effectively making the generator itself work as a flexible and fast simulator. In order to make interaction with the simulator as easy as possible, a graphical user interface was build around it. This gives the developer the possibility to edit his source code, compile it and simulate it all in a single, easy to use, environment.

Benchmarks show that this new simulation approach is a factor of 10 times faster than the existing interpretive simulator, while providing more flexibility in terms of Montium design parameters (directly available to the end-user). Benchmarks also show that even though using C instead of Java as the target language for code-generation, will result in somewhat faster simulations, the difference in performance isn't big enough to provide a good reason to abandon the portability and extendability benefits provided by using Java.

Contents

1 Introduction **13**
- 1.1 The Montium Tile Processor 13
- 1.2 Assignment 18
- 1.3 Structure of this report 19

2 Related Work **21**
- 2.1 Simulation techniques 21
  - 2.1.1 (V)HDL simulators 21
  - 2.1.2 Instruction level simulators 22
  - 2.1.3 Binary compiled simulation 22
- 2.2 The ‘Simsation’ simulator 23
- 2.3 The Montium TP Simulation Generator 23

3 Design **25**
- 3.1 Benefits of a functional simulator 25
- 3.2 Simulation generation 25
- 3.3 Target Language 27
- 3.4 Required functionality (API) 28
- 3.5 Software Design 28
- 3.6 Flexibility 29
  - 3.6.1 Montium design parameters 29
  - 3.6.2 Functional behaviour 29
- 3.7 Initial State and end-results 30
- 3.8 The optimization steps 30
  - 3.8.1 Straight forward optimizations 30
  - 3.8.2 Look-back optimizations 31
  - 3.8.3 Look-ahead optimizations 31

4 Implementation **33**
- 4.1 Configuration parser 34
- 4.2 Sequencer 35
  - 4.2.1 Cycle code generation 35
  - 4.2.2 State Machine 35

4.2.3 Switch/case bottleneck issue 36
4.3 Code generation 37
4.3.1 Decoders 38
4.3.2 Global and Local Interconnections 39
4.3.3 Streaming IO 40
4.3.4 Register Files 41
4.3.5 Address Generation Units 42
4.3.6 Code generation visitor for the ALUs 43
4.4 Code generation summarized 48
4.5 Code Compilation process 49
4.5.1 Compilation of Java code 49
4.5.2 Compilation of C code 49
4.6 Backwards stepping via snapshots 50
4.7 Optimizations 50
4.7.1 Data dependencies (external) 51
4.7.2 Data dependencies (internal) 52
4.7.3 Selective code generation 52
4.7.4 Hotspot or Loop detection 54

5 Communication and Configuration Unit 57
5.1 The Hydra CCU 57
5.2 Simulating the CCU 59

6 Reconfigurable fabric of the Annabelle 63

7 Results 67
7.1 Flexibility 67
7.1.1 Runtime flexibility 67
7.1.2 Compile time flexibility 68
7.2 Simulation Performance 68
7.2.1 Motivation and Overview 68
7.2.2 Benchmark setup 69
7.2.3 Performance Java versus C Output 70
7.2.4 Performance compared to ‘Simsation’ without CCU . 75
7.2.5 Performance compared to ‘Simsation’ with CCU . . . 76
7.2.6 Improvements achieved by optimizations 77
7.2.7 Running multiple simulations in parallel 80

8 Conclusion and Recommendations 81
8.1 Conclusion 81
8.2 Recommendations for Future Work 81
8.2.1 Simulation of a complete Annabelle SoC 82
8.2.2 Design Parameters in CFG Files 82
8.2.3 ‘Undefined’ values 83

8.2.4 Generation of Java byte-code 83
8.2.5 More integration with other tools 84
8.2.6 Create a complete test-suite 84
8.2.7 Create a framework to integrate the simulation generator with a Scripting environment 84

A The Simulator API **87**

B Montium Design Parameters **91**

C Simulation Datastructure **93**

D FU Helper Methods **95**

E Source used for Benchmarking the CCU **97**

List of Figures

1.1 Montium Tile Processor and CCU 14
1.2 Simplified schematic of the Arithmetic and Logic Unit 15
1.3 Register File inside an ALU 15
1.4 Control of an Montium ALU 17

3.1 Code Generation schematic 26
3.2 The Simulation Generation process using Java (above) or C . 27
3.3 Simplified UML overview of the Simulator design 29

4.1 Simulation Generator overview 33
4.2 Register File CR (4 positions) 34
4.3 Register File CR (8 positions) 35
4.4 Small piece of the ALU datapath 44
4.5 Functional Units in the ALU datapath 46
4.6 Transitions from type 1 instructions 51
4.7 Transition from type 2 instructions 51
4.8 Transition from type 3 instruction 51
4.9 Selective code generation for RF read addresses 53
4.10 Selective code generation for an ALU 54

5.1 the Montium Tile connected to a NoC router 57
5.2 State diagram for the CCU[1] 58

6.1 Reconfigurable fabric of the Annabelle chip 63
6.2 UML overview of approach 1 64
6.3 UML overview of approach 2 65
6.4 Graphical representation of a router configuration. 66

7.1 Simulation speed for FIR algorithms 71
7.2 Simulation speed for FFT algorithms 72
7.3 Simulation speed for the 1920-points FFT with input-scaling 73
7.4 Optimization Effects on Performance (non-streaming) 79

8.1 Block diagram of the Annabelle chip 82

List of Listings

4.1 Structural description of a Register File configuration register 35
4.2 Main simulator loop - the step() method 36
4.3 Generated switch/case tree 37
4.4 Example generated code for individual interconnects, transfers a value from memory 9 to register A of the first processing part 39
4.5 Example generated code for combined interconnects 39
4.6 Example generated code for Streaming IO input 41
4.7 Example generated code for Streaming IO output 41
4.8 Example generated code for a Register File 41
4.9 Example generated AGU code 42
4.10 Code describing the datapath fragment shown in Figure 4.4 . 44
4.11 Example generated ALU code 1 46
4.12 Example generated ALU code 2 46
4.13 Example generated ALU code 3 46
4.14 Example generated code for an entire cycle 48
6.1 Simulating the reconfigurable part of the Annabelle 66
6.2 Example router configuration file 66
7.1 Example parametric datapath 68
7.2 Benchmark Loop 69
7.3 Benchmark Loop for Simsation 75
7.4 Benchmark Loop for Simsation (CCU) 76
A.1 Simulation API 87
C.1 Basic Simulation Datastructure 93
D.1 FU Helper Methods 95
E.1 CDL Source used for Benchmarking the CCU 97

Glossary

AGU	Address Generation Unit
ALU	Arithmetic and Logic Unit
API	Application Programming Interface
CCU	Communication and Configuration Unit
DSP	Digital Signal Processing
DSRA	Domain Specific Reconfigurable Accelerator
FFT	Fast Fourier Transformation
FIR	Finite Impulse Response
GPI	General Purpose Input
GPO	General Purpose Output
GUI	Graphical User Interface
ISA	Instruction Set Architecture
MAC	Multiply Accumulate
Montium TP	Montium Tile Processor
NoC	Network-on-Chip
PC	Program Counter: the address indicating where a processor is in its instruction sequence
PPA	Processing Part Array
SB	Status Bits
SIO	Streaming Input and Output
SoC	System-on-Chip
UML	Unified Modeling Language
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

1.1 The Montium Tile Processor

Battery powered mobile devices nowadays tend to be given more and more functionality. To support this functionality the demand for more processing power and flexibility increases while energy consumption needs to be kept at a minimum. For addressing this problem the Chameleon System-on-Chip template was designed. In the Chameleon SoC, heterogeneous processing tiles are connected via a network-on-chip. The essence of this idea is that the processing of a task is performed by a tile that has the best support for that specific kind of task[1]. The Montium Tile processor[1][2] is a domain specific reconfigurable accelerator, DSRA, for a Chameleon System-on-Chip. It is less flexible than a general purpose processor, but more efficient in doing the specific tasks it is targeted for. The target application domain the Montium TP was designed for is the domain of 16-bit DSP algorithms like Finite Impulse Response (FIR-) filters and Fast Fourier Transformations (FFTs). The Montium TP is capable of doing multiple calculations in a single clock-cycle by using 5 rich ALUs, specifically designed for DSP algorithms, in parallel. Every one of these ALUs can do some logic functions, a Multiply-Accumulate (MAC) and a butterfly operation all together in a single clock-cycle. A Montium Tile consist of a Montium Tile processor and a Communication and Configuration Unit (CCU) connecting the Tile within the SoC. Within the *Smart chips for Smart Surroundings* (4S) project [4] the *Annabelle* prototype chip was developed. The *Annabelle* consist of, among other things, an ARM926 general purpose processor with a 5-layer AMBA bus, 4 Montium Tile Processors, a Viterbi decoder, two digital down converters (DDCs), memory and external connections[3].

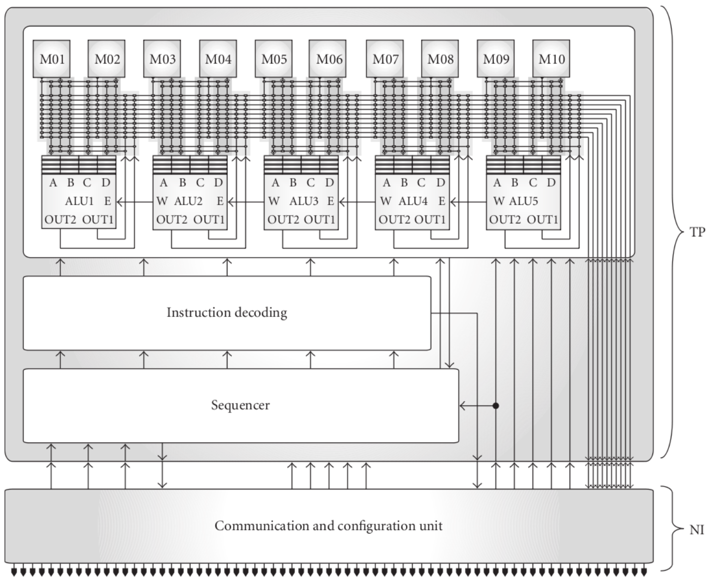


Figure 1.1: Montium Tile Processor and CCU

Figure 1.1 show the Montium Tile Processor together with the Communication and Configuration Unit. An single ALU together with its register files and two memories is called a Processing Part (PP). The five ALUs together are referred to as the Processing Part Array (PPA).

The Montium ALUs

Figure 1.2 shows a simplified schematic of the Montium's Arithmetic and Logic Unit (ALU). To accommodate many DSP operations that work on more than two operands, e.g. a Multiply-Accumulate (MAC) operation works on three operands, the Montium ALU has four input operands (most ALUs have only two input operands). Each of these input operands has a private register file, which cannot be bypassed, and can be written by multiple sources (e.g. memories or interconnects). Every cycle the Montium ALU produces two outputs which are directly connected to the interconnect. The ALU consist of an upper and a lower level. The upper level contains four function units, these function units implement general arithmetic and logic functions. The lower level contains an MAC and a butterfly unit typically used in many DSP algorithms. Each ALU has a single status bit output that can be tested by the sequencer that controls the Processing Part Array (PPA).

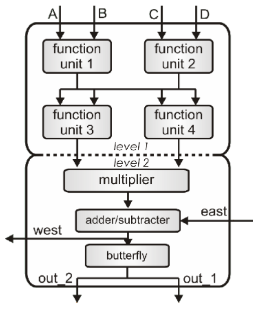


Figure 1.2: Simplified schematic of the Arithmetic and Logic Unit

remark: To minimize the number of configuration registers used, the Montium compiler will try to merge instructions whenever possible. Example given: Assume an algorithm that needs a ALU to performs a MAC operation every other clock cycle, but doesn't need the ALU the remaining cycle. The Montium configuration will simple contain a single configuration for the ALU, containing the MAC operation. The unwanted result, generated every clock cycle the MAC isn't needed, simply gets disregarded, since it will not be used/written anywhere.

Register Files

Every ALU in the Montium Tile has four register files, one for every single input A, B, C and D. Every one of these register files can hold up to four 16 bit values. Each register file is controlled by a read address, a write address and a write enable signal.

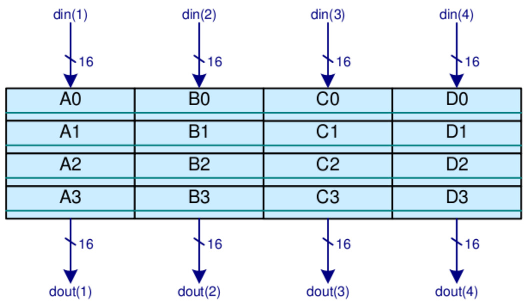


Figure 1.3: Register File inside an ALU

The Montium AGUs

There are 10 local memories on the Tile, every PP has two local memories, denoted as left-hand side memory and right-hand side memory. Every memory in the Montium has its own reconfigurable Address Generation Unit (AGU). These AGUs can generate simple memory access sequences typically used in DSP algorithms. Operations that these AGUs include are, among others, incrementation, bit-reversal, apply-ing and-masks.

Interconnect System

The Montium Processing Part Array has a reconfigurable interconnect for flexible routing of data within the Tile, that can use a different configuration every clock cycle. There are 10 Global Busses (GB01..GB10) used for inter-process communication and every PP has a local bus connecting the ALU to its local register files and two local memories. In total there are 20 data sources in the PPA (10 memories and 10 ALU outputs) and 30 data sinks (10 memories and 20 register files). In addition to on-tile communication the CCU can use the Global Busses to connect the tile to the outside world, every cycle at most 4 inputs and 4 outputs can enter and exit the Tile via the CCU.

Control in the Montium TP

The combinations of concurrent functions the five ALUs can perform in a single clock cycle is called a pattern. The flexibility of the PPA results in a vast amount of possible patterns. The programmability of the PPA is limited for efficiency reasons. For example[7] take the control of an ALU (see Figure 1.2). In the Montium each ALU has 37 control signals, resulting in 2^{37} possible function patterns per ALU. In practise however, only a few combinations are actually used. The functions an ALU needs to execute a stored in *ALU instruction registers*. Each ALU has 8 of these registers, at runtime every clock cycle one of these registers is selected to control the function of the ALU. An *ALU decoder register*, which is also a configuration register, determines which ALU instruction register is been selected for every ALU. As there are five ALUs on the PPA, there are 8^5 different combinations possible. However in practise not all of these are actually used for one application. Therefore, there are only 32 ALU decoder registers in the Montium.

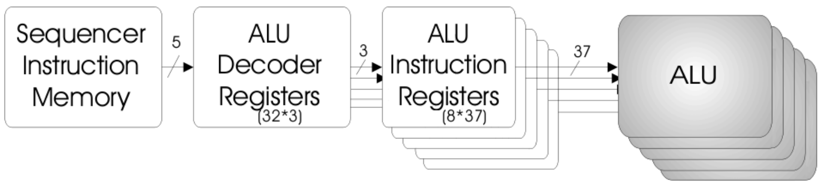


Figure 1.4: Control of an Montium ALU

Every cycle the sequencer instruction selects a ALU decoder register which will select an ALU instruction register for every ALU. In summary the 185 (5 x 37) control signals for the ALUs are reduces to 5 signals for selection of the ALU decoder register. This same two layered scheme is also used for the memories, register files and interconnect configurations.

The Montium Sequencer

The Montium sequencer is basically a state machine, that in every clock selects a register for every decoder (ALU decoder, memory decoder, register decoder and interconnect decoder). The current address in the sequencer program, called the Program Counter (PC), specifies which register to select in every decoder. The flow trough this sequencer program can be influenced by the sequencer instruction and arguments. The Montium sequencer has a fixed instruction set:

encoding	mnemonic	description
000	JCC	Jump Condition Code
001	JNC	Jump Not Condition (code)
010	LLC	Load Loop Counter
011	LOOP	Loop
100	SIG	General purpose IO signaling
101	CCC	Call Condition Code
110	CNC	Call Not Condition (code)
111	RET	Return

The branch instructions (JCC, JNC, CCC and CNC) can use the ALU status outputs, handshake signals from the CCU and internal sequencer flags. Although the sequencer supports conditional jumps, their usage should be kept to a minimum for optimum performance. Algorithms that require a lot of conditional code can better be implemented on a general purpose processor. The Montium Tile has four 11-bit loop counters that can be used either individually or combined in pairs. A simple Montium sequencer program could look like this:

PC	sequencer instruction	arguments	description
0	JNC	GPI_0 0	wait for "Data Valid" (GPI_0)
1	JNC	TRUE 0	single cycle
2	LLC	0 1022	load LC_0 with 1022
3	LOOP	0 3	loop on LC_0 (to PC 3)
4	SIG	0 1	set GPO_0
5	SIG	1 0	clear GPO_0
6	JCC	TRUE	jump to beginning

1.2 Assignment

Development and study of algorithm mapping onto a coarse grained reconfigurable architecture requires the possibility to verify and debug the produced code. Several reasons exist why testing on a real chip is not always a good option:

1. A development board may not always be available to a developer, since these development boards are expensive to produce.
2. During the development process the developer needs to be able to see how his code is being processed, how variable values change over time, where data comes from and goes to (the developer needs to be able to look inside the contents of the chips registers and memories).
3. The developer needs to be able to observe and verify the implementation effects of his algorithm, especially in the field of DSP algorithms some effects may only become visible or reliable after extensive simulation (e.g. quantization effects). For these reasons a fast simulator of the target architecture is needed.

The simulator currently available for the Montium TP, 'Simsation', has some disadvantages:

1. It is not very flexible for the end-user. While compiler options are available to developers to change certain attributes of the Montium TP instance, these changes do not work on the current simulator (in fact a binary compiled with custom parameters will not work in the current simulator at all).

2. Since it simulates the internals of the architecture it has to do a lot of (extra) work not really interesting to a software developer who just wants to verify the results of his program. Speed especially becomes an important issue when algorithm precision needs to be analyzed, since verifying (average) bit-error-rates of certain scaling or quantization effects often requires simulating many billions of cycles.

3. It proofed to be somewhat burdensome to implement a Graphical User Interface on top of it. Communication with the simulator is possible only via a (telnet) socket and thus a lot of message passing and parsing needs to be done. Exporting a public API could greatly simplify this process.

The goal of this project is to research, design, implement and test a functional simulation generator for the Montium TP which has to be flexible in terms of architecture parameters and fast in execution.

1.3 Structure of this report

After this introduction, in Chapter 2, some related work will briefly be summarized to place this work in perspective to what has already been done in similar research projects. In Chapter 3 some design choices made in the early stage of the project will be discussed. After the design choices are made clear the implementation of the simulation generation process is explained in Chapter 4, there the reader can also find examples of generated code blocks. In Chapter 5 the Communication and Configuration Unit of the Montium Tile will be discussed. Chapter 6 will show how the created simulation generator is extended to simulate the Annabelle prototype chip, created within the 4S project. Chapter 7 will present the results achieved in this project, including benchmarks and comparisons. These results will be followed by some conclusions and recommendations in Chapter 8 which will be the final chapter in this report.

Chapter 2

Related Work

Probably since the day digital chips where first introduced there has been a need for software that can simulate those chips. This need comes for the same reasons that where already mentioned in the introduction: the unavailability of real hardware and the possibility for the developer to see inside the chip while a program is running. So a lot of research has been performed in the field of simulators. This chapter summarizes some research projects related to this project in order to give some perspective.

2.1 Simulation techniques

Several different approaches can be identified for creating simulators: (1) (V)HDL Simulators, (2) Instruction Level Simulators and (3) Binary compiled simulations

2.1.1 (V)HDL simulators

Simulations based upon VHDL or Verilog hardware description languages emulate all signals that exist in the real hardware thus provide a real close one-to-one relation to the actual chip. This is particularly useful when the correct functional behaviour of the chip has to be examined. This approach can also be used for examining real elapsed time or energy consumption of the chip (or the specific implementation of an algorithm). The main disadvantage of this approach is that it is really slow, the simulator will typically require many thousands of host cycles to simulate a single target cycle.
An approach close to HDL simulation was researched by Aly and Salem [8], [9]. They build *RTLJava*, an RTL (Register Transfer Level) simulator written in Java. They use Java's built-in multithreading and observability features to deal with concurrency, parallel execution of statements, and reactivity problems that most HDL simulators cope with. This approach results in a simulator that resembles the hardware closely, but is easier to

modify, operate and link to other software than a VHDL simulation. A disadvantage is that it has to do a lot of update calls to all the primitive objects to simulate a single cycle.

2.1.2 Instruction level simulators

Instruction level simulators usually act as op-code interpreters, a target program is executed by decoding every op-code instruction sequentially, just before execution (just as the real chip would do). However, the internals of the simulator do not necessarily resemble the real chip at all. Because all decoding and interpreting steps have to be done at runtime, these simulators tend to perform rather poorly.
A big problem often faced when creating instruction level simulators lies in the fact that it is quite a lot of work to develop one for every newly developed chip. *Sleipnir* [10] is a tool that eases writing IL-(instruction level) simulators. It makes the development of IL-simulators easier by only requiring a description of the target chip. It works by compiling a machine description language into C source code files, which can be compiled to an executable simulator. The generated simulator is an interpretive simulator, meaning it has to decode instructions at runtime. Togawa et al. [11] also use code-generation to generate an instruction level simulator for DSP type processors. Their main goal was to include support for packed SIMD instructions.

2.1.3 Binary compiled simulation

In case of binary compiled simulation the simulator generates a native binary for the host platform directly based upon the input program meant for the target chip. So for every target program a new unique simulator binary will be generated. This approach generally results in very fast simulations, but very often lacks end-user flexibility because the hardware description of the target platform is usually embedded inside the binary generator.
Pees et al. [12] designed a compiled binary simulator based on the machine description language LISA. Later this work was extended by Nohl et al. [13] with their so-called JIT-CSS, Just-In-Time Cache Compiler, technique to regain the flexibility of an interpretive simulator. It worked by pre-compiling all instruction blocks beforehand, placing them in a cache and then run the simulation just like an 'interpretive' simulator would but instead of interpreting every instruction it will fetch the desired one from the compiler cache.

2.2 The ‘Simsation’ simulator

Another project closely related to this project is the currently existing Montium TP simulator, ‘Simsation’, already mentioned in the Introduction. ‘Simsation’ is an interpretive simulator that simulates all internal signals that exist inside the Montium TP hardware architecture. The user has to navigate through a tree like structure to locate the different variables of interest (or create scripts to do so). Though this simulator works correctly, it has a few drawbacks: (1) It lacks end-user flexibility in terms of Montium design parameters, (2) It is not really fast, and finally (3) There is no Graphical User Interface available (and it is not designed so that one can easily be added on top of it).

2.3 The Montium TP Simulation Generator

In this project we will try to combine the flexibility of interpretive simulators with the performance of binary simulation. We do this by creating a simulation generator that can generate and compile new binary simulations internally (based on a binary target program and design parameters for the target). Because in the case of the Montium Tile Processor we have an entire *configuration*, ($\approx$ the program), available beforehand, we can generate and compile code not just for single instructions, but for the entire configuration at once. Another difference with Instruction Set Architectures, or ISA, is that ISA based systems all start every cycle by decoding the next instruction, fetching operands, doing some calculation on the operands and finally writing the result back to some memory entity. The Montium TP however, being a coarse-grained reconfigurable chip, doesn’t have this sequential behaviour so obviously available.

Chapter 3

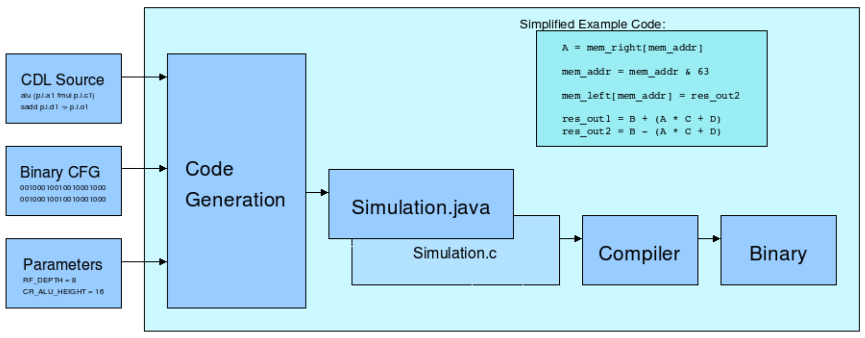
Design

3.1 Benefits of a functional simulator

A functional simulator simulates the functional behaviour of its target platform, in contrast to simulating all internal signals. This means that the functional behaviour can be implemented in a way that is efficient on the host architecture, for example a multiplier that uses multiple steps in the VHDL description, with or without intermediate results, would also take multiple steps to simulate in a VHDL simulator approach but in a functional simulator it would just be passed to the host processor as a single multiply instruction with two operands. Obviously this functional approach results in much better performance of the resulting simulator.

3.2 Simulation generation

Instead of simulating a Montium program by interpreting a configuration cycle for cycle we instead generate Java code for the configuration, compile it (internally) and start an instance of this newly created program. This gives us the best of two worlds of simulators. It provides the flexibility of an interpreter, because we can use runtime variables. But it should also give us the performance of a binary compiled simulation, because we in fact generate a compilable version of the current program and compile it. The generator will thus use two inputs: (1) the binary Montium configuration and (2) the specific Montium Design Parameters (if the latter is omitted, the generator will continue with the default Montium Design Parameters).



```
alu (p.i.a1 fmul p.i.c1)
sadd p.i.d1 -> p.i.o1
```

```
0010001001001000 1000
0010001001001000 1000
```

```
RF_DEPTH = 8
CR_ALU_HEIGHT = 16
```

```
A = mem_right[mem_addr]

mem_addr = mem_addr & 63

mem_left[mem_addr] = res_out2

res_out1 = B + (A * C + D)
res_out2 = B - (A * C + D)
```

Figure 3.1: Code Generation schematic

Figure 3.1 shows schematically how the Simulation Generation process works. The generator has up to three inputs:

1. CFG, the pseudo-binary Montium configuration as is generated by the Montium Compiler (mandatory input).

2. PRM, the Montium Design Parameters (optional, as there are default parameters).

3. CDL Source, the source code that was used to create the binary Montium configuration (optional, not actually used in the code-generation process but only used for visualization).

Though it would have been possible to build a simulation generator that uses CDL source as primary input, there are several reasons why the compiled configuration file was chosen for this:

- Several steps the compiler performs are needed before simulation generation. If CDL source would be the primary input for the simulation generator these steps would have to be duplicated:
  - Semantic checking of the source code.
  - Allocation of global- and local interconnects.
- By using the binary configuration format as input the simulation generator will also be able to simulate configurations compiled with future (higher level language) compilers.

3.3 Target Language

Because the final simulator needs to be portable, Java was chosen as the target language. All end-users will need to have, in order to use the simulator, is a JDK installed on their system. No external libraries will be required. Using Java also makes it easy to separate the program functionality from the user interface, making it easy to embed the simulation generator in a future integrated development environment or maybe even embed it a some sort of scriptable environment.
Another important reason to prefer Java is the new *compiler* feature introduced in JDK 6 SE. This new internal compiler feature makes it possible to create a simulation generator that will perform the code generation and compilation process **internally**, hiding these details from end-users. This means the end-user doesn't need to understand, or even know about this code generation and compilation process, nor does he need to make explicit calls to a compiler or build tool. In case of C code generation a complete C toolchain (e.g. GCC and make) is required on the running machine.
However, since C is still assumed to be faster than Java in program execution, the generator will be designed so that it can produce either Java or C code. Benchmarks will be performed to see whether Java truly is a viable choice.

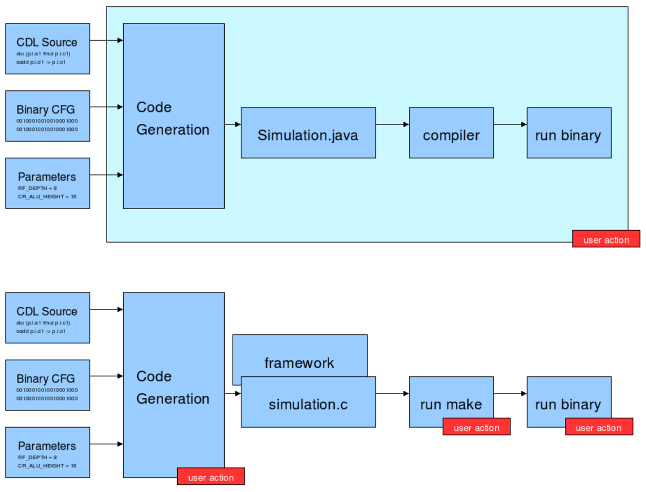


Figure 3.2: The Simulation Generation process using Java (above) or C

3.4 Required functionality (API)

For operation of the simulation the generated simulator will have to export a public accessible API. This API will provide the end-user, via a graphical interface or even directly from an Java program, with basic controls needed to manipulate a running simulation. The basic API should in some way provide at least the following functionality to the end-user:

run(int n)	run n cycles
step()	run a single cycle
back()	step back a single cycle
back(int n)	step back n cycles
get(name)	show the current value of: - a register - a memory location - a status bit
set(name)	override the current value of: - a register - a memory location - a status bit
watch(name)	break upon the value change of: - a register - a memory location - a status bit

Please note that this is not the final implemented API. For a complete overview of the final implemented API see appendix A.

3.5 Software Design

Figure 3.3 is a simplified UML view of how the final **generated** Simulator is build, the *ASimulation* class is an abstract class that is needed so we can refer to Simulation methods inside the encapsulating program (e.g. make calls to *getMem(int m, int l)* from inside the GUI classes) without having an actual Simulation class instantiated, or even compiled. The *Simulation* class is the most important class, this is the class that has to be generated by the generation process. It will implicitly contain the Montium design parameters and the Montium Configuration that were passed as inputs to the simulation generator. The SimulationController is responsible for loading and managing the Simulation instance. It also keeps track of cycles simulated (the Simulation doesn't care about cycles), and provides some higher-level interactions with the Simulation (e.g. parse command-line commands containing *Strings* like *setReg(pp1 A 6)* or running until a certain value changes). The *SequencerStateMachine* is responsible for calculating the new *Program Counter* every cycle, based upon the current PC, the tile SB (status bits), the current instruction and the arguments given.

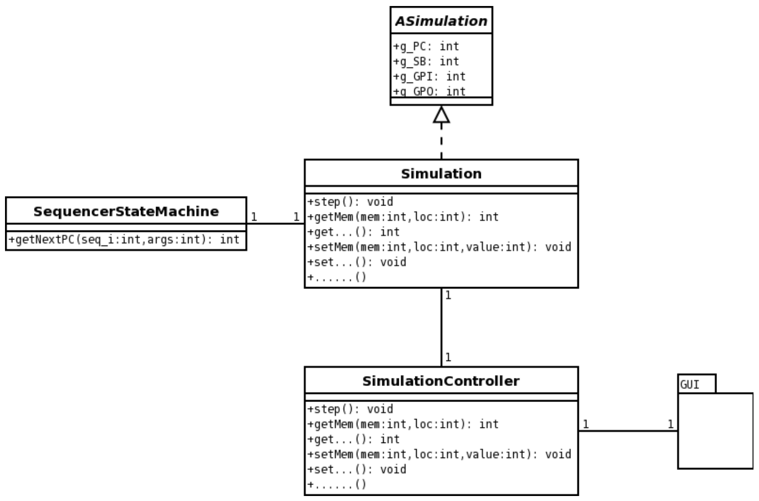


Figure 3.3: Simplified UML overview of the Simulator design

3.6 Flexibility

One of the two goals of this new simulator was to obtain more flexibility with regard to Montium design parameters and internal functional behaviour.

3.6.1 Montium design parameters

The currently available Montium compiler supports several parameters describing characteristics of a Montium Tile instance. This flexibility should also be available in the simulator, not just to the engineers at *Recore Systems* that have access to the simulator source code, but also to their clients. Thus these parameters should be runtime configurable.
These parameters should include, but not necessarily be limited to, the height of decoder and configuration registers, the width of memory addresses (thus the size of memories) and the depth of register files. For a complete list or currently implemented parameters see appendix B.

3.6.2 Functional behaviour

Besides parametric changes it should also be possible to easily extend/change the functionality of the functional units of the Tile. Even making changes to the datapath should stay simple enough to be performed whenever needed.

Examples of such changes could be removal of one or both levels of functional units or extending the level 1 functional units with MIN and MAX function (by default only available in the level 2 functional units). Because clients probably shouldn't be allowed complete freedom to the datapath functionality, this flexibility is not necessary at runtime. Moreover implementing this kind of flexibility at runtime would require some sort of specification language for describing this functional behaviour, and using that language would require skills similar to normal programming skills. Rather than inventing a new specification language one could just as well use Java code itself for the specification, assuming it is possible to isolate this code in a single place. This code isolation can be achieved by using the Visitor pattern for code generation of ALU expressions and creating *helper functions* for the different operation in the Functional Units.

3.7 Initial State and end-results

For testing of algorithms it is often useful to start a simulator with all the memories and register files filled with some initial values. For fast correctness testing it can also be useful to compare the final contents of all memories and registers with some pre-defined *expected results*. Finally, to produce some figures about results it can be convenient to be able to write all memory- and register contents to easily parseable output files at any time during the simulation. For these input and output files it was chosen to use the same format the currently available simulator uses, so that these are interchangeable and don't have to be created twice. The format is quite simple, the files have an hexadecimal address-value pair on every line.

3.8 The optimization steps

One of the prospected advantages of functional simulation in contrast with full signal simulation, combined with the fact that we analyze the entire configuration beforehand, is that now several optimization steps can be performed. Everything that has no effect on the final state change of the simulator can be removed from the generated code.

3.8.1 Straight forward optimizations

The most straight-forward form of optimization lies in disregarding statements that don't change anything to begin with. For example calculating the output of a functional unit that isn't really used can be skipped.

3.8.2 Look-back optimizations

All transition from one line in the sequencer program to the next can be deduction from the configuration. This means that already at generation time the generator can identify what the previous PC could have been. If for all these predecessors a certain read address were the same as for the current PC, than the simulation code for that assignment can be left out in the code for this PC (this will probably remove close to 20 assignments in many cycles, since in many cycles the register file read addresses remain the same).

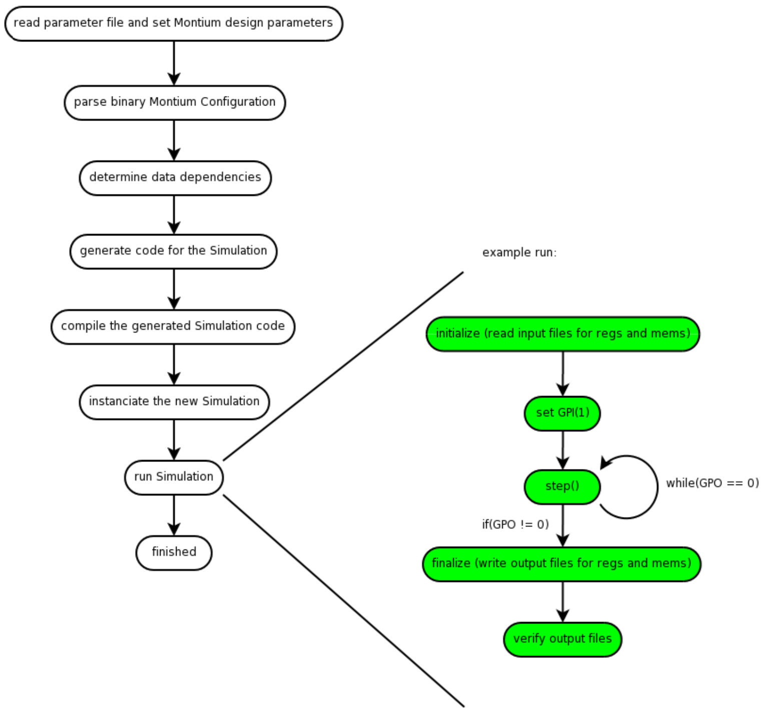
3.8.3 Look-ahead optimizations

This is probably the most effective optimization scheme, for it is capable of removing big blocks of unnecessary calculations. The Montium Tile processor generates 10 outputs for its 5 ALUs, but many of these outputs are never actually written anywhere. At generation time it should be possible for the generator to check all possible *next-states* of the current state, if none of these *followers* write a specific output to a memory, register or global interconnect we can remove the calculations that provide that specific output without changing the end-results of the simulation.

Chapter 4

Implementation

This Chapter describes the implementation of the Simulation Generator. The overall flow of a simulation is shown in Figure 4.1. The white boxes represent steps in the Simulation Generator and the colored boxes are an example of how a Simulation run could look like.



read parameter file and set Montium design parameters
parse binary Montium Configuration
determine data dependencies
generate code for the Simulation
compile the generated Simulation code
instanciate the new Simulation
run Simulation
finished
example run:
initialize (read input files for regs and mems)
set GPI(1)
step()
while(GPO == 0)
if(GPO != 0)
finalize (write output files for regs and mems)
verify output files

Figure 4.1: Simulation Generator overview

4.1 Configuration parser

The Montium compiler produces pseudo-binary output files containing a Montium TP configuration which can be loaded into a Montium processor via its configuration interface. Because the configuration data bus of the Montium is 16 bits wide but configuration lines can vary in width, e.g. an ALU configuration line is currently 37-bits wide, two different views of the configuration entity exist: (1) the normal view, in which every entity has it's own specific width and (2) the configuration view, in which every entity is at most 16-bits wide.
The pseudo-binary configuration file contains the 16-bits configuration view, but for processing the information and building a simulator the normal view is needed. So the first task the simulation generator has when it receives a new configuration is decoding this flat configuration view into a datastructure for which all information can be retrieved easily and selectively. Later, during the code generation process, the internal datastructure, created during this parse, can be called to return individual register contents within the entire Montium configuration. e.g. assuming the datastructure for the Montium configuration is stored in *config* then:

```
Bv_SequencerInstruction instr = (Bv_SequencerInstruction)config.getImSeq()[N];
```

will assign the N^{th} line in the sequencer program, and subsequently calling:

```
int alu_i = instr.get("alu_i");
```

will assign the active line number to select in the *Alu Decoder*. Similar calls can be made to fetch any entity within the loaded configuration.
Because the simulation generator has to be flexible when it comes to Montium design parameters, the conversion from the configuration view to the internal datastructure has to take into account these parameters al well. If for example the design parameters state that the register files should have a depth of 8 (instead of the default depth of 4) the conversion from configuration view to normal view has to realize that for addressing within these register files now 3 bits are required. So, consequently an *register file configuration line* will be 4 bits wider as well (see examples, one bit extra for both read addresses and one bit extra for both write addresses). So, the simulation generator will have to 'know' how all the design parameters affect the configuration space.

rfA_rd	rfA_wr	rfA_we	rfB_rd	rfB_wr	rfB_we
00	00	0	11	00	0

Figure 4.2: An example Register File configuration line, for 4 position deep register files

rfA_rd	rfA_wr	rfA_we	rfB_rd	rfB_wr	rfB_we
000	000	0	101	000	0

Figure 4.3: An example Register File configuration line, for 8 position deep register files

The defaults for all these column names and sizes are stored in respectively the *Constants* class and the *StructureVariables* class, and are overridden (if necessary) during the reading of the Montium design parameter file (passed to the generator program as a start-up parameter). A single structure definition looks like Listing 4.1.

```
CR_RFAB_COLNAMES = { "rfA_rd", "rfA_wr", "rfA_we", "rfB_rd", "rfB_wr", "rfB_we" };
CR_RFAB_COLWIDTH = { 2, 2, 1, 2, 2, 1 };
```

Listing 4.1: Structural description of a Register File configuration register

4.2 Sequencer

Simulation of the Montium *sequencer* consist of two very distinct parts. The first sequencer related task is collecting all actions for the Tile to perform for a certain line in the sequencer program. This part is done by the simulation generator. The second sequencer related task is the runtime task to provide the simulation with the next address in the sequencer program, or the PC, to execute.

4.2.1 Cycle code generation

For every line in the sequencer program a block of code is generated that is the functional equivalent of what the real Montium Tile would do during execution of this sequencer line. This step in code generation will create by far the biggest and most important part of the *Simulation* class. Whenever the **step()** function is called the simulator will check the current PC (Program Counter), execute the code generated for that specific sequencer instruction: update memory, register and global bus values, calculate required ALU outputs, etc and finally return. All the information about actions to perform in a certain PC cycle can be found in the binary configuration. This process is thoroughly described later on.

4.2.2 State Machine

The second sequencer related task, providing the simulation with the next PC, depends on data that is only available at runtime, during a simulation 'run'. This cannot be done at generation time, so for this purpose a *SequencerStateMachine* class is created which will be instantiated by the simulation at runtime. All this class does is generate the new PC and update the Loop Counters and the General Purpose Out signals whenever necessary.

4.2.3 Switch/case bottleneck issue

Because the first thing that needs to happen when stepping into the next cycle is jumping to the code specific for the active sequencer line, the naive way of building up the step function is to create one huge switch/-case statement containing the code blocks for all possible sequencer lines.

```
public void step(){
  switch(PC){
    case 0:
      ...
    case N:
  }
}
```

Listing 4.2: Main simulator loop - the step() method

This approach would however create a lot of runtime overhead, because most cases would not be taken (a program that uses all sequencer lines in an evenly distributed fashion would produce N/2 jump misses on average for every cycle). This is in fact a known problem/drawback of big switch/case statements in today's software world. C in fact has a solution for this: create an array of function pointers and use that as a table to get the jump addresses from. To find out whether this would pose a big problem in our generated simulation some benchmarks where performed on two different approaches:

1. Use C function pointers as a reference.
2. Use a single huge switch/case statement, as described above.
3. Use so-called anonymous classes extending some interface that has a run() method, and place these anonymous classes in an array (this resembles the C function pointer approach).

	minutes	seconds	calls/sec	score
C (function pointers)	9.48	588	1.13E+11	100
Anonymous Classes	16.58	1018	6.55E+10	57.76
One big Switch Case	13.41	821	8.12E+10	71.62

Table 4.1: Benchmark results 'anonymous classes' vs. switch

The above benchmarks where executed with 256 different case statements, with a single line of actual code inside them. To prevent the compiler from optimizing the step out of the actual execution the code was setup to return an integer depending on input values, which is again passed to the next

step() call. The results of this benchmark showed that using anonymous classes was not a solution to our problem, but it also showed that using the switch/case wasn't as bad as expected. So it was decided to proceed with this approach (hoping to optimize it somewhat more later on).
Because Java has a hard limit on the maximum length a method can be the switch/case statement was split up in two levels. In the first level *step()* jump to *step1()* if the PC to execute next lies between 0 and 15, jump to *step2()* if it lies between 16 and 31, and so on for all possible lines in the sequencer program. Mid-way during the project it was discovered that this approach indeed still held back performance by some sort of bottleneck issue, so some different variations on this approach where tested and finally some extra trial-and-error testing showed that the best approach was in fact:

- Create a tree with a maximum of four levels, balancing the code blocks amongst the number of required levels (if there are just 4 lines in the sequencer program, create only one level, if there are 12 lines create 2 levels with 3 leaves in all the second level methods.... and so on (when all 256 sequencer configurations are used this will give a 4 level tree with 4 'leaves' in the final levels).

```
public void step(){
  switch(PC){
    case 0: ... case 63:
      return step_1_0()
    ...
          public void step_1_0(){
            case 0: .. case 15:
              return step_2_0_0()
            ...

                public void step_2_0_0(){
                    case 0: .. case 3:
                      return step_3_0_0_0();
                    ...

                        public void step_3_0_0_0(){
                            case 0:
                              /* code for PC 0 here */
                                      ...
                            case 3:
                              /* code for PC 3 here */
                        }
```

Listing 4.3: Generated switch/case tree

4.3 Code generation

When the code generation process starts first the basic simulation necessities are written to the *Simulation* class. These necessities contain the following distinct elements:

- Montium Tile datastructure, placeholders for all internal variables like memories, registers and intermediate results.

- Simulation initialization functions, methods that take care of reading input files, writing output files, resetting the simulation and so forth.

- Simulation API, these are the get() and set() functions used to obtain/manipulate the Tile's state information during simulation
- Helper functions, that implement the special functions inside the Functional Units that can not be expressed in 'in-fix' java notation, these helper functions also include status bit update methods.

The full Tile datastructure is given in Appendix C and the Function Unit Helper functions can be found in Appendix D. The other methods are not included in this report, because they have little value to people other that the software maintainer of this project. When these basic simulation necessities are written to a Simulation class, the Sequencer code generation process is started. This will add the code blocks for every sequencer line present in the loaded configuration. The configuration however, contains all the actions without any notion of precedence. In order to create a sequential piece of java code for every block, a sequence[1] in which things should happen has to be created:

1. check if tile blocked by IO (if so **break** without change)
2. Fetch value from lane_in lanes
3. Write values to lane_out lanes
4. Calculate next Program Counter
5. Reset Status Bits
6. Set read addresses for the register files
7. Write registers, using the correct write addresses
8. Undo previous bit-reversal and calculate new AGU outputs
9. Write memories
10. Calculate ALU outputs

4.3.1 Decoders

The decoders of the Montium Tile processor only select the active addresses for different configuration registers. So they do not need to become a real part of the simulation at runtime. For every cycle for which the generator needs to generate code it will select all the decoder lines to use, based upon the active sequencer line. With these decoder lines the generator can select the correct configuration register for every entity (Register File, ALU, AGU or Interconnect) it should generate code for.

[1]The correct sequence is not uniquely defined, several correct variations also exist.

4.3.2 Global and Local Interconnections

For transferring data from the different data sources to data sinks the Montium PPA (Processing Part Array) has a configurable interconnect that can be reconfigured every clock cycle. Several components of this interconnect can be configured individually:

- The active *global interconnect* configuration registers define the sources for the global busses.
- The active *streaming IO* configuration register defines whether any of the global busses is connected to the outside world via in- and output lanes.
- The active *local interconnect to memory* configuration registers define the sources for local busses to the local memories of the processing parts.
- The active *local interconnect to register* configuration registers define the sources for local busses to the local register files of the processing parts.

Sequentially resolving these connections individually would result in code like in Listing 4.4 for a single write action.

```
// gint -> connect memory 9 to global bus 2
GB02 = mem_left[4][memL_address[4]];

// lint2r -> connect global bus 2 to the local interconnect to
//           register of the first processing part
lint_rfA_connection[0] = GB02;

// rfAB -> write the value on the local interconnect to register
//         to register A address 0
rfA[0][rfA_wr_pointer[0]] = lint_rfA_connection[0];
```

Listing 4.4: Example generated code for individual interconnects, transfers a value from memory 9 to register A of the first processing part

Because all this information is available at generation time, and since we are only interested in simulation of the functional behaviour, this generated code can be reduced to a single assignment like shown in Listing 4.5.
For every write actions in a Register or Memory, and for every Streaming output, the source that produced the value to the interconnect can be discovered by combining the information in the active *local interconnect to memory*, *local interconnect to register*, *global interconnect* and *streaming IO* configuration registers mentioned above.

```
rfA[0][rfA_wr_pointer[0]] = mem_left[4][memL_address[4]];
```

Listing 4.5: Example generated code for combined interconnects

Because this source determination has to be done at several places in the code, namely for all Register writes, Memory writes and Streaming IO outputs the *GlobalInterconnect* class was created. Given the *target* in the specific local interconnect (or 'lane2gb'), the active configuration for the global interconnects and the active streaming IO configuration, it returns a String representing the source variable inside the generated Simulation class (this String is then immediately added to the generated code). These Strings returned by the *GlobalInterconnect* can be any of:

"mem_left[0..4][memL_address[0..4]]"	any left-hand side memory.
"mem_right[0..4][memR_address[0..4]]"	any right-hand side memory.
"res_out1[0..4]"	the 1st result of any ALU.
"res_out2[0..4]"	the 2nd result of any ALU.
"GB[0..9]"	any global bus (indirectly connected via the Streaming IO configuration (see next section)).

4.3.3 Streaming IO

To allow streaming algorithms, the Montium Tile is connected to a NoC (Network-on-Chip), via a CCU (Communication and Configuration Unit). One of the tasks of this CCU is providing data to- and consuming data from the global busses in the Montium. The CCU part of this process consist of adding data to- and reading data from a FIFO buffer. In streaming algorithms the Montium configuration decides when to connect a global bus to an external lane. Via this connection the Montium can now receive data from the CCU output buffer or write data to the CCU input buffer.
During code generation this means that when streaming IO is enabled for a certain line in the sequencer program, the code generator needs to check whether a certain global bus is connected to a lane input and if so generate code for fetching the next value from the CCU buffer and assign it to the specific global bus variable (global busses are represented by a single variable in the simulator). If later in that cycle that global bus is used as a source for a register or memory write, we can simple assign the variable name for that bus. Also some code has to be added that breaks out of the current execution if there is no value received from the CCU. That way the simulator would remain executing the same block over-and-over until the next value is available thus simulating blocked IO. Code generated in a cycle using streaming IO input will look like Listing 4.6.

```
if (lane_in_ready[0] == false){
  return;              // blocked by IO
} // 'else':
GB[0] = lane_in_buf[0];  // get value from lane_in buffer to GB01
if (lane_in_ready[1] == false){
  return;              // blocked by IO
} // 'else':
GB[1] = lane_in_buf[1];  // get value from lane_in buffer to GB02
  ...
mem_left[1][memL_addr[1]] = GB[1];
  ...
mem_left[0][memL_addr[0]] = GB[0];
  ...
```

Listing 4.6: Example generated code for Streaming IO input

If any of the *gb2lane* columns in the active streaming IO configuration is non-zero, then code will be generated for placing an output value on that specific lane, the CCU will retrieve this value after the Tile is done for this cycle. Code generated in a cycle using streaming IO output will look like this:

```
FType = 0;                          // set flit-type for data to CCU
lane_out_buf[0] = res_out1[0];       // set data ready for reading by CCU
data_waiting[0] = true;              // inform the CCU there is new data
```

Listing 4.7: Example generated code for Streaming IO output

4.3.4 Register Files

Every ALU in the Montium Tile has four register files, one for every single input A, B, C and D. The current Tile has four positions for every one of these register files (the compiler and the simulator can vary this depth, by altering the RF_DEPTH parameter).
Before calculations can be done the simulator has to know the active read and write positions in these register files. They can be obtained from configuration registers *cr_rfab* and *cr_rfcd* of the processing part, selected by the Register decoder. Write sources can be decoded by combining the information in the local interconnect to register, the global interconnect and the streaming IO configurations. For a single processing part this will result in code blocks like this one:

```
rfA_rd_pointer[4] = 0;
rfB_rd_pointer[4] = 0;
rfC_rd_pointer[4] = 0;
rfD_rd_pointer[4] = 0;
rfA[4][0] = res_out2[4];                         // LB2
rfB[4][0] = mem_right[0][memR_address[0]];       // GB06
rfC[4][0] = res_out1[4];                         // LB1
rfD[4][0] = mem_right[2][memR_address[2]];       // GB07
```

Listing 4.8: Example generated code for a Register File

4.3.5 Address Generation Units

The Montium Tile processor contains a reconfigurable Address Generation Unit or AGU for every memory on the tile. These AGUs are capable of generating simple addressing schemes often used in DSP algorithms[1].
The code generation for the AGUs works according to the following sequence: (all references to registers refer to configuration registers, their values are retrievable from the currently loaded Montium configuration.)

1. check (the *add_offset* register to see) if the *sel_offset* register for this cycle should be added to the previous address or be used directly as new offset. And add the corresponding code to the simulation.

2. add code to the simulation that applies the AND mask from the *sel_mask* register to the memory address

3. check (the *add_base* register to see) if the *sel_base* register for this cycle should be added to the previous base-register or be used directly as new base. Add the corresponding code to the simulation.

4. check if *load_addr* is set for this cycle, if so then decode the source location from the local and global interconnect registers and add code for using either the low, if *load_low* is set, or the high part of received 16-bit value as new memory address.

5. save a backup of the address register and apply bit-reversal on the output address if *br_width* $!= 0$ (This backup is restored in the next cycle just before the AGU will produce a new output).

6. check if the *we*, write enable, bit is set in the current memory configuration register and if so, decode the source from the local interconnect to memory, the global interconnect and the streaming IO configuration registers values and write this to the active memory address.

The code generated for a single AGU could look like this:

```
memL_address[4] = (memL_address[4] + 1) & MEM_MINUS_ONE;
memL_address[4] = (memL_address[4] & 63);
memL_base[4] = 1;
memL_address[4] = (memL_address[4]) | (memL_base[4] << MEM_WIDTH_WO_BASE);
prev_memL_address[4] = memL_address[4];
memL_address[4] = BitUtil.reverseBits(memL_address[4], 6, MEM_ADDR_WIDTH);
```

Listing 4.9: Example generated AGU code

4.3.6 Code generation visitor for the ALUs

As mentioned before in Section~refsect:funcflex, the visitor pattern was used to create so-called abstract syntax trees (ASTs) of the ALU instructions. This way all code related to the code-generation can be kept in one single place, in the *JavaCodeGenerationVisitor*. AST nodes have been made for:

name	parents	description
IntNode	none	fixed integer value
NullNode	none	fixed '0'
VarNode	none	a variable name accessible in the Simulation class, e.g. "rfA[pp][rfa_rd_pointer[pp]]"
Add	in0, in1, enable	addition node
FU_std_lvl1	in0, in1, ctf	top-level Functional Unit
FU_std_lvl2	in0, in1, ctf	second level Functional Unit
Mult	in0, in1, sel	multiplication node
Mux	in0 .. in*N*, sel	multiplexer node
RtMux	in0, in1	multiplexer node, that can only make a decision at runtime
Convert115to216	in0	add precision bits
Convert115to416	in0	add precision bits
Convert170to171	in0	add precision bits
Convert216to316	in0	add precision bits
Convert316to416	in0	add precision bits
ConvertEnable	in0, status	enables a parent node, or pass '0'
GetHigh	in0	node that takes high part of a 32-bit multiplier result (fixed-point multiplication)
Saturate32to17	in0	saturates a 32-bit multiplier result to 17.0 fixed-point value (integer multiplication)
Saturateto216	in0	saturates the east-west result to 2.16
Scale	in0, sel	scale the end result if requested
Roundto415	in0	rounds the end result
Saturateto115	in0	saturates the end result to 1.15

Table 4.2: Overview of AST Nodes

These AST nodes are interconnected in *Alu.java* to represent the current datapath structure. This datapath structuring code remains quite readable, so that future modifications to the datapath can be implemented in the simulation generator with minimal effort. It's even possible to extend this structure with conditional connections, creating runtime flexibility in the datapath. Listing 4.10 shows an example of the ALU connection code, for the fragment of the Montium datapath shown in Figure 4.4, the entire datapath can be described in this very syntax:

```
// level 2.mux
mux_mX = new Mux( new Wire(new NullNode()), wA, wB,
                  new Wire(fu[2]), selmx
                );
mux_mY = new Mux( new Wire(new NullNode()), wC, wD,
                  new Wire(fu[3]), selmy
                );
// level 2.multiplier
mult = new Mult( new Wire(mux_mX), new Wire(mux_mY),
                  selmw
                );
// level 2.converters and muxes left
conv[0] = new Saturate32to17(new Wire(mult));
conv[1] = new GetHigh(new Wire(mult));
conv[2] = new Convert115to216(new Wire(mux_mX));
conv[3] = new Convert115to216(new Wire(mux_mY));
conv[4] = new Convert170to171(new Wire(conv[0]));
mux_mW = new Mux( new Wire(conv[2]), new Wire(conv[3]),
                  new Wire(conv[4]), new Wire(conv[1]),
                  selmw
                );
```

Listing 4.10: Code describing the datapath fragment shown in Figure 4.4

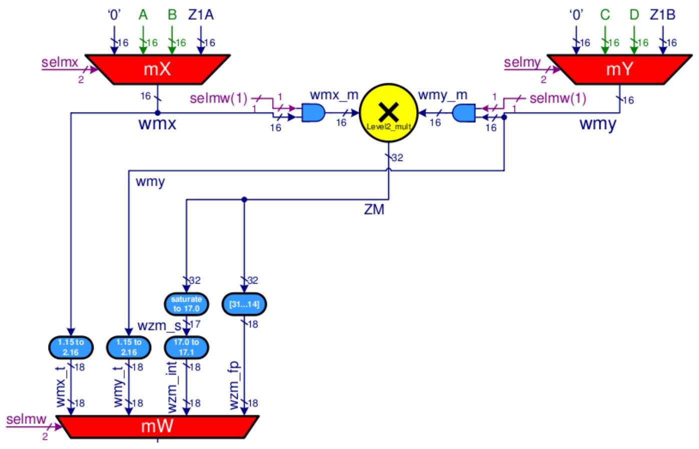


Figure 4.4: Small piece of the ALU datapath

An ALU code block can be generated simply by setting all selection signals from the ALU configuration and then calling the **visit()** method on the two lowest multiplexers selecting the ALU outputs (res_out1 and res_out2). The AST is then build bottom-up, meaning the code generation visitor will traverse upwards in the datapath, only generating code for Nodes it actually passes.

example
Take the above datapath image and assume that: $selmx = 01$, $selmy = 11$ and $selmw = 11$ (this means that the output of the multiplexer mux_mW should become the 18-bit fixed-point value of $A * Z1B$).

If *mux_mW* is now visited by the visitor, it will traverse up to the *GetHigh* converter Node, because *selmw* selects the 4th input of the multiplexer. Next the visitor will traverse to the *mult* Node and from there it will traverse to the *mux_mX* Node and the *mux_mY* Node. From the *mux_mX* Node it will traverse to the *wA* Node, because *selmx* selects the second input of the multiplier. From there the visitor will traverse no further, because it has reached the top of this AST path (the *wA* Node is a *VarNode* that only generates code and has no parent nodes). From the *mux_mY* node it will traverse up to the *FU4* Node outside the range of the above image.

Just before the *Sequencer* starts generating code for all the lines in the sequencer program all ALUs are ‘asked’ to generate code for every instruction in their configuration registers and save them in the so-called *CodeKeeper*. This way it is not necessary to run the code generation visitor over-and-over again for all sequencer lines that re-use a certain ALU configuration. At code generation time the *Sequencer* just retrieves the correct code block from the *CodeKeeper* and writes that down. Every ALU generates up to three outputs, first it will generate an expression for the *ZA* output just below the accumulator in the MAC level of the ALU. This output is used not only in the butterfly level of this ALU but also as an *east* input for the neighbouring ALU. The other two expressions that an ALU generates are for the two *res_out* outputs. The final code generated for these are pretty straightforward Java expressions, example blocks of ALU code generated for a cycle could look like these[2]:
(1) generated expression for an ALU that performs the calculations:

$$\begin{aligned} D + (A\,fmul\,C - east) &\rightarrow res\_out1 \\ D - (A\,fmul\,C - east) &\rightarrow res\_out2 \end{aligned}$$

[2]In the code listings ‘rfX_rd_pointer’ variables are abbreviated by ‘Xptr’ to improve readability

```
ZA[0] = (((((rfA[0][Aptr[0]])*(rfC[0][Cptr[0]])) >> 14))-((saturate216(ZA[1]))));
res_out1[0] = ((saturate115((((((rfD[0][Dptr[0]]) << 1)+ZA[0])) + 1) >> 1)));
res_out2[0] = ((saturate115((((((rfD[0][Dptr[0]]) << 1)-ZA[0])) + 1) >> 1)));
```

Listing 4.11: Example generated ALU code 1

(2) generated expression for an ALU that performs the calculations:

$$\begin{aligned} B + (A\, imul\, (C\, sadd\, D)) &\rightarrow res\_out1 \\ C\, sadd\, D &\rightarrow res\_out2 \end{aligned}$$

```
ZA[0] = ((((rfA[0][Aptr[0]])*(sadd(rfC[0][Cptr[0]],rfD[0][Dptr[0]]))) >> 14))+(0);
res_out1[0] = ((saturate115((((((rfB[0][Bptr[0]]) << 1)+ZA[0])) + 1) >> 1)));
res_out2[0] = (sadd(rfC[0][Cptr[0]],rfD[0][Dptr[0]]));
```

Listing 4.12: Example generated ALU code 2

(3) generated expression for an ALU that performs the calculations:

$$\begin{aligned} C\, sadd\, D &\rightarrow res\_out1 \\ A\, sadd\, B &\rightarrow res\_out2 \end{aligned}$$

```
ZA[0] = ((((0) << 1))+(0));
res_out1[0] = (sadd(rfC[4][Cptr[4]],rfD[4][Dptr[4]]));
res_out2[0] = (setZeroNeg(sadd_wsb(rfA[4][Aptr[4]],rfB[4][Bptr[4]], 4), 4, 0));
```

Listing 4.13: Example generated ALU code 3

Status Bits

The first level of the Montium TP ALU has four functional units that can do operation like shifting, saturated addition, exclusive OR and so on. All of these Functional units have two 16-bit inputs and a single 16-bit output, this means that things like overflow or underflow can occur.

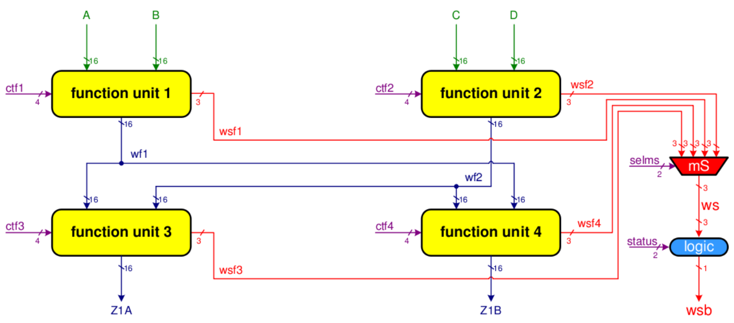


Figure 4.5: Functional Units in the ALU datapath

These overflow conditions are not visible from the result a Unit generates, therefore the Functional Units also generate 3 status bits, respectively for *zero*, *negative* and *overflow*. These status bits can be used in the DSP algorithm to choose between inputs for the seconds level of the ALU datapath and/or to control the program flow of the algorithm (jump to a different section of the algorithm).
Every cycle at most one Functional Unit is selected for generating the status bit for the PP via the *selms* part of the active ALU configuration register and the meaning of the status bit is configured via the *status* part. Because the simulation generator has this information beforehand we only need to generate code for generation of the status bits for the FU that is selected by *selms*. In the third example (Listing 4.13) the part *setZeroNeg(sadd_wsb())* comes from the fact that that specific FU, performing that calculation, is selected for setting the **ws** status bits, for this ALU. The outer *setZeroNeg()* function checks a value for being zero or being negative and also updates the final single status bit (**wsb**) for this PP. The inner *sadd_wsb()* does exactly the same as the normal FU helper function *sadd()* but in addition to that it also sets the status bit for overflow if saturation has occurred. Similarly there are also extended helper functions for other operations that can affect a status bit.

4.4 Code generation summarized

To give an impression of the code generated for a single clockcycle please see the example shown in Listing 4.14. The example shows a cycle where two inputs, read from input lanes, are assigned to the registers of four processing parts. The input sample from lane0 goes to reg.A of PP1 and PP3, denoted by rfA[0] and rfA[2] in the code, and the input sample from lane1 goes to reg.A of PP2 and PP4, rfA[1] and rfA[3] in the code. The reg.C registers of these four processing parts are filled with results calculated in the previous cycle. AGU outputs are calculated for the 10 memory units. And finally outputs are calculated for processing parts 1,2 and 3 and for the west output of processing part 4 (since that is needed as an east input for PP3).

```
if (lane_in_ready[0] == false){
  return g_PC; // blocked by IO
}
GB[0] = lane_in_buf[0];          // get value from lane_in buffer
if (lane_in_ready[1] == false){
  return g_PC; // blocked by IO
}
GB[1] = lane_in_buf[1];          // get value from lane_in buffer
FType = 0;                       // set flit-type for data to CCU
// Cycle done, calculate new PC, and GPO
ssm.getNextPC(1, 0);
g_SB = 0; ws[0] = 0; ws[1] = 0; ws[2] = 0; ws[3] = 0; ws[4] = 0;
rfA[3][0] = GB[1];               // GB02 from CCU
lane_in_ready[1] = false;        // streaming value consumed.
rfC[3][0] = res_out1[0];         // GB03
rfA[2][0] = GB[0];               // GB01 from CCU
lane_in_ready[0] = false;        // streaming value consumed.
rfC[2][0] = res_out1[2];         // LB1
rfA[1][0] = GB[1];               // GB02 from CCU
lane_in_ready[1] = false;        // streaming value consumed.
rfC[1][0] = res_out1[2];         // GB04
rfA[0][0] = GB[0];               // GB01 from CCU
lane_in_ready[0] = false;        // streaming value consumed.
rfC[0][0] = res_out1[0];         // LB1
memL_address[4] = (memL_address[4] & 1023);
memL_address[4] = (memL_address[4]) | (memL_base[4] << MEM_WIDTH_WO_BASE);
prev_memL_address[4] = memL_address[4];
memR_address[4] = (memR_address[4] & 1023);
memR_address[4] = (memR_address[4]) | (memR_base[4] << MEM_WIDTH_WO_BASE);
prev_memR_address[4] = memR_address[4];

  // code for memories 3, 2 and 1 was cut out here

memL_address[0] = (memL_address[0] & 1023);
memL_address[0] = (memL_address[0]) | (memL_base[0] << MEM_WIDTH_WO_BASE);
prev_memL_address[0] = memL_address[0];
memR_address[0] = (memR_address[0] & 1023);
memR_address[0] = (memR_address[0]) | (memR_base[0] << MEM_WIDTH_WO_BASE);
prev_memR_address[0] = memR_address[0];
ZA[3] = (((((rfA[3][Aptr[3]])*(rfC[3][Cptr[3]])) >> 14))+(0));
ZA[2] = (((((rfA[2][Aptr[2]])*(rfC[2][Cptr[2]])) >> 14))+((saturate216(ZA[3]))));
res_out1[2] = ((saturate115(((((0 << 1)+ZA[2])) + 1) >> 1)));
res_out2[2] = (setZeroNeg(rfB[2][Bptr[2]], 2, 0));
ZA[1] = (((((rfA[1][Aptr[1]])*(rfC[1][Cptr[1]])) >> 14))+(0));
res_out2[1] = (setZeroNeg(rfB[1][Bptr[1]], 1, 0));
ZA[0] = (((((rfA[0][Aptr[0]])*(rfC[0][Cptr[0]])) >> 14))-((saturate216(ZA[1]))));
res_out1[0] = ((saturate115(((((0 << 1)+ZA[0])) + 1) >> 1)));
```

Listing 4.14: Example generated code for an entire cycle

4.5 Code Compilation process

As mentioned in the Design, Chapter 3, the simulation generator produces source code that is functional equivalent to the Montium TP program to be simulated. There are some slight differences in doing this for Java code and doing this for C code (or any other target language for that matter).

4.5.1 Compilation of Java code

In case of Java code generation we can keep the process of code generation and compilation internal to the simulation generator, invisible for the outside world. To make this happen, all generated code is appended to a single String, this String is than placed inside a *JavaSourceFromString* Object that extends the normal *SimpleJavaFileObject*. This Object is added to the *modules* list of a *JavaCompiler.CompilationTask* and that task will be given to the *JavaCompiler*. For keeping the resulting **class** Object in memory as well the *JavaCompiler* is also given a *RAMFileManager* that has a *RAMClassLoader*. Now, every time we need an instance of this internally kept class we can call:

```
Class sim_class = manager.getClassLoader(null).loadClass("Simulation");
sim = (ASimulation)sim_class.newInstance();
```

4.5.2 Compilation of C code

When instead of generating Java code we want C code to be generated, and compile a native system binary executable from that C source, this it what happens: First there is a Framework containing the sources and header files that do not change from simulation to simulation. This Framework contains:

Makefile	build information.
bit_util.c, bit_util.h	bit level utility functions. (e.g. bit-reversal)
sim_util.c, sim_util.h	sim level utility functions. (e.g. reading initial mems from file)
sequencer.c, sequencer.h	calculation of next PC.
run_sim.c, run_sim.h	runs sim until GPO changes.

Next the generated Java cycle code is converted to valid C code. This is basically a find and replace algorithm that seeks for certain Java specific statements and replaces them with C equivalents. This is possible since the generated cycle source for the *Simulator* class contains only expressions that have one-to-one equivalents in C (no Object usage, no try/catch statements and so on). This cycle code, now valid C code, is added to static datastructures for storing the simulation state, and helper functions for the functional units, and written to a file.

All these files are written to a temporary directory, **make** is called to build the simulation, the generated executable *'run_sim'* is moved to the path the simulation generator was started in and the temporary directory containing the source files is removed.
It is the end-users responsibility to have a working *make*, *GCC compiler* and C development libraries available on the system.

4.6 Backwards stepping via snapshots

Debugging an algorithm often involves running a Simulation until a certain event, e.g. a register change, occurs and then stepping back one, or a few, cycles to see where this change originates from. Normally stepping back a single cycle from cycle number **N** in the simulator would mean: resetting the simulator, reloading the initial values and running **N-1** cycles. This can become slow when **N** becomes large. For this reasons the *Simulation-Controller* is given a so-called *TileStateStore* to save snapshots of states the simulation was previously in. The number of states to remember can be controlled by a parametric value, and is a trade-off between memory usage and fast back-stepping to many different locations. When the CCU is included in the simulation the CCU will also keep a *CCUStateStore* for recalling of previous states.
Now whenever a **back()** or an **undo()** is called, first the *StateStores* will be checked for availability of the state requested, if the requested state (identified by the number of cycles the CCU resp. the Tile has stepped at that point) is available then is will be restored, else the normal procedure of reset, reïnitialize and running **N-1** steps is performed.

4.7 Optimizations

To perform the optimizations mentioned in Section 3.8, we need to calculate possible predecessors (and followers) for every state. The possible transition from the current state to the next depends on the current Program Counter ($\approx$ sequencer instruction), the Tile SB (Status Bits) and the GPI (General Purpose Inputs). Since these are not available at generation time, we can only know where we possibly **could** end up after this cycle. There are three distinct scenarios for determining the possible next states, depending on the current sequencer instruction:

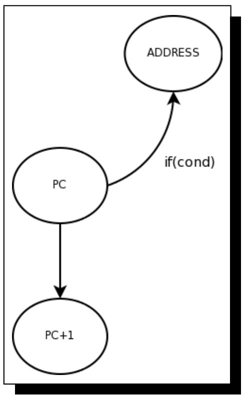


Figure 4.6: Transitions from type 1 instructions

(1) Sequencer instructions JCC, JNC, LOOP, CCC and CNC
Most sequencer instructions have two possible next-state addresses, one is given by the parameter of the sequencer instruction and the other one is the next line in the sequencer program. For example a **JCC** can either go to *PC+1* if the condition evaluates to true or to *address* if the condition evaluates to false. If the argument holds a fixed TRUE or FALSE, we can determine the single possible next-state. Otherwise, when the evaluation of the condition is not available at generation time, we have to add both to the list of possible next-states.

(2) Sequencer instructions LLC, SIG
If for the current line in the sequencer program the selected sequencer instruction is either **LLC** (Load Loop Counter) or **SIG** (set or clear Signals) the single available transition is to the next sequencer line *PC+1*. So for this state the only possible transition to the next-state is to state *PC+1*, and this state is a possible predecessor of state *PC+1* (but not necessarily the only predecessor).

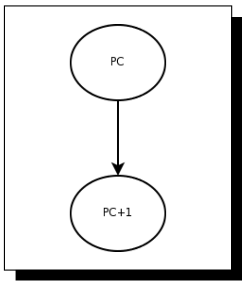


Figure 4.7: Transition from type 2 instructions

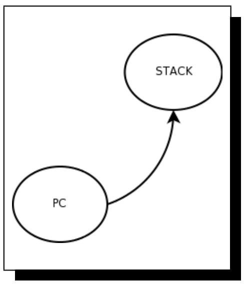


Figure 4.8: Transition from type 3 instruction

(3) Sequencer instruction RET
If for the current line in the sequencer program the selected sequencer instruction is **RET** (Return from a call) the single available transition is to the sequencer address on the stack, since the stack is not a know value at generation time we cannot determine the next-state from the information available in this state. We can however check the entire configuration for all **CCC** and **CNC** instructions and add them as possible next-states, because we know that we will jump to one of these states.

4.7.1 Data dependencies (external)

Now that we know the possible next-states for a certain state we can eliminate unnecessary code. To do that we first collect all data dependencies of the possible next-states. The data dependencies of a cycle are all outputs that are written to a register, memory or streamed to a lane_out (via the global bus). The union of data dependencies of all possible next-states is all that needs to be calculated in this state.

example:
state with sequencer line PC has possible next states: *S1* and *S2*.
S1 has data dependencies for: $(pp1.res\_out1, pp4.res\_out1)$
S2 only has data dependencies for: $(pp1.res\_out2, pp2.res\_out2)$
this means that in state PC we need to add code for the calculation of:

$$(pp1.res\_out1, pp4.res\_out1) \cup (pp1.res\_out2, pp2.res\_out2) = \\ (pp1.res\_out1, pp1.res\_out2, pp2.res\_out2, pp4.res\_out1)$$

and that we don't need to add code for the calculation of:

$$(pp2.res\_out1, pp3.res\_out1, pp3.res\_out2, \\ pp4.res\_out2, pp5.res\_out1, pp5.res\_out2)$$

4.7.2 Data dependencies (internal)

The east-west connection between the different ALUs inside the Montium Tile can also create data dependencies inside a state itself. For example if PP1 needs an input from its neighbour, PP2, this will create an internal dependency on PP2. However if the results pp2.res_out1 and pp2.res_out2 are not member of the external data dependency lists, it is not necessary to calculate the entire PP2 ALU. In that case we can suffice by just calculating the function units and the multiply accumulate (we can thus skip the butterfly, scaling and rounding that happened in lower part of level 2).

4.7.3 Selective code generation

Now that we know both the data dependencies for every possible line in the sequencer program and we know all state transitions for every line in the sequencer program, we can selectively decide to generate specific blocks of code.

Selective code generation for the Register Files

Instead of blindly assigning all read addresses for all register files, we can now check the read addresses in the preceding states. If in every of these preceding states a certain register file read address is the same as the one we would generate code for in this state then we can omit generating the assignment. If however in any single one of the possible predecessor state the read address is different, then we have to generate the assignment code here. Since for many algorithms register file read addresses don't alter for every state transition this selective code generation can save up to 20 assignments per cycle. Figure 4.9 shows the general idea. For the state at the bottom only one read address really needs to be assigned, namely *rfB*, because this is the only one that had a different value in one of the predecessors (the state in the middle reads from position 1 instead of position 0 for rfB).

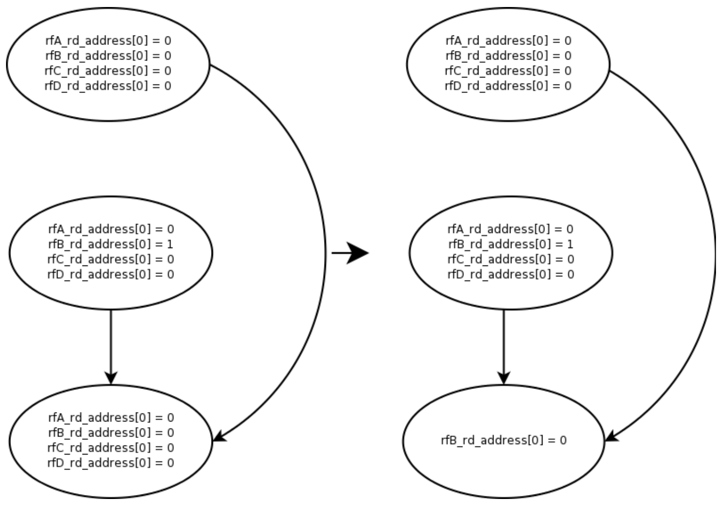


Figure 4.9: Example of selective code generation for RF read addresses

Even though this optimization step removes a lot of lines in the generated code, it will not provide a big speed-up, since all the lines are just single assignments, already taking very little time compared to other work that has to be done in a cycle (and it is not unthinkable either that the Java compiler will optimize some of these as well).

Selective code generation for the ALUs

The Montium Tile processor generates 2 outputs for each ALU, or 10 outputs in total every single cycle. Not all of these outputs are always used by the algorithm currently running on the Tile, or in terms of Montium actions: they are not written anywhere the next cycle.
For ALU outputs, not listed in any of the data dependency lists of possible successor states, we can skip the generation of code. In the example in Figure 4.10 we can see that the lowest state only writes one of the two results produced by the upper state to memory. Since the lower state is the only successor for the upper state, we can omit calculation of the second result res_out2[1] in the upper state without changing the end results the algorithm will finally produce.

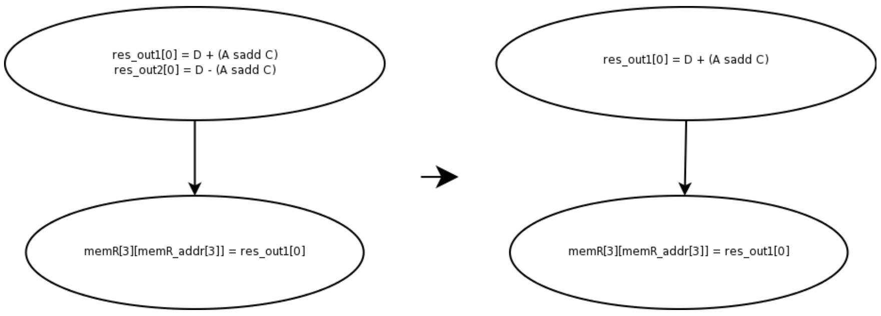

res_out1[0] = D + (A sadd C)
res_out2[0] = D - (A sadd C)
memR[3][memR_addr[3]] = res_out1[0]
res_out1[0] = D + (A sadd C)
memR[3][memR_addr[3]] = res_out1[0]

Figure 4.10: Example of selective code generation for an ALU

Because the code omitted by this optimization step actually contains heavy workloads, these are all actual calculations that otherwise have to be processed by the host computer running the simulator, this optimization should give us some measurable speed-up.

Selective code generation for the AGUs

Bit reversals inside the Address generation Units do not affect the address register in the real Montium Chip, rather they only affect the output of the ALU during the current cycle [2]. To implement this in the simulator an 'prev_addr' variable was introduced for every AGU, whenever the AGU should apply bit-reversal it first saves the current address to this 'prev_addr'. Every cycle this 'prev_addr' is used to override the current address variable just before calculation of the new address. Since we now posses a list of possible predecessors to the current state we can do this undo-ing selectively, only when in a previous state actual bit-reversal took place. Again, just like the selective code generation for register files this will not provide a substantial speed-up.

4.7.4 Hotspot or Loop detection

When a Montium Configuration becomes bigger, in other words when the sequencer program will have more unique configuration lines, the tree mentioned in Listing 4.3 becomes more saturated. This means that every cycle the simulation has to call 4 levels of **step()** methods before reaching the code to be executed. Because many DSP algorithm kernels consist of a single (or a few) small loops it might show prudent to identify the most heavily called *PC* and move this code block into the top-level **step()** method. This is what the *Hotspot* optimizer in the simulation generator does, it performs a dry loop of the sequencer program, meaning it does the loop without processing any data and thus without taking status changes into account, and

counts how many times it executed every line in the sequencer program. This of course does not necessarily provide a completely accurate invocation counts, but should in many cases be able to find the most frequently called instruction correctly. When done the simulator can place the line with the highest execution count into the top-level of the generated tree (this is very likely the most heavily called PC in a real simulation as well, so this should reduce the method call overhead somewhat in the simulation depending on the simulated algorithm).
Since most streaming algorithm only terminate after the last data sample is received from the CCU (by looping on a sequencer line until an interrupt occurs) the Optimization step does not work well for these algorithms (because the dry run doesn't terminate), therefore this step is skipped when the CCU is used during simulation.

Chapter 5

Communication and Configuration Unit

5.1 The Hydra CCU

For communication with the outside world and configuration of the Tile Processor the Montium Tile is equipped with a Communication and Configuration Unit, or CCU (see Figure 1.1). This CCU is again connected to the network-on-chip via a router (see Figure 5.1). In the Annabelle SoC this CCU is implemented by the *Hydra* CCU[5].

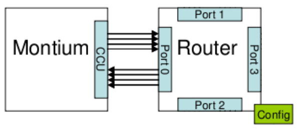


Figure 5.1: the Montium Tile connected to a NoC router

The state diagram for the CCU tasks are shown in Figure 5.2 and consists of: Configuring the Montium TP, loading initial data into the memories and registers of the Montium TP via DMA, executing the algorithm (putting the Tile in run mode), and finally retrieving results from the Montium TP's memories and registers. When the Tile is in run mode, the CCU is responsible for feeding the Montium TP with streaming input data and retrieving streaming output data from the Montium TP and forwarding that data to the NoC.
Communication units between the CCU and the routers are called *flits*, a flit contains a 2-bit *FT*, flit-type, field and a 16-bit *data* field, together creating an 18-bit atomic unit. Every clockcycle every channel can transport a

single flit. The *Hydra* CCU discussed here, has 8 of these physical channels for communication with the network: 4 ingoing channels and 4 outgoing channels.

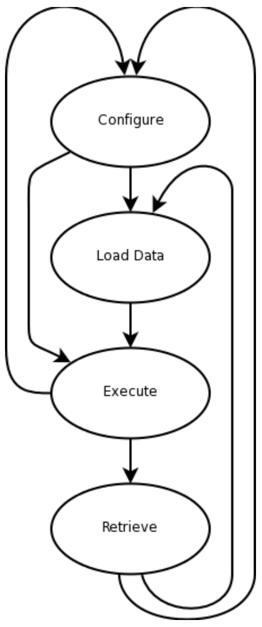


Figure 5.2: State diagram for the CCU[1]

Besides the (obvious needed) data flits, the CCU recognizes 3 other flit types, making the total list of flits:

FT	flit type	Function
00	data	carries user data
01	address	carries an address
10	tail	marks the end of a message
11	command	carries a command for the CCU

Table 5.1: Overview of flit types

Command flits change the state of the CCU, they affect the way the CCU will handle the rest of the incoming data (or message). The *Hydra* understands single flit messages as well as multi-flit messages.

An overview of supported messages is shown in Table 5.2. For a more detailed description please refer to the "Hydra Design Specification" [5] In the current implementation of the CCU, command messages are only supported via lane0. Input lanes 1, 2 and 3 are only used for loading data during **load** and for streaming data during **run**.

Encoding	Command	Message
000	Start configuring	(cmd_start_cfg)((addr)(data)$^+$)$^+$(tail)
001	Start loading data	(cmd_start_load)((addr)(data)$^+$)$^+$(tail)
010	Start retrieving data	(cmd_start_retr)((addr)(data))$^+$(tail)
011	Get GPo	(cmd_get_gpo)
100	Run	(cmd_run)
101	Idle	(cmd_idle)
110	Reset	(cmd_reset)
111	Unused	

Table 5.2: Overview of commands and corresponding messages

5.2 Simulating the CCU

After realization of the simulation generator for the Tile Processor, the simulator is extended to simulate the *Hydra* CCU as well. The input files that the CCU simulator will accept are the same input files as used by the current simulator, this again to improve portability of test-suites between the two simulators. The format is quite simple:every line in a lane_in (.lix) file contains a hexadecimal flit-type and a hexadecimal data value. Every cycle the CCU (simulator) handles the next flit (=line) available from every active lane.
A new flit is processed every cycle. However the Tile Processor (simulation) doesn't necessarily consume data every cycle. Therefore a FIFO buffer is in place between the CCU (simulation) and the Simulation. If the Simulation needs an input but the FIFO doesn't provide one, the Simulation will retry, simulating blocked IO transfers. In hardware all flits go through this FIFO, but since the simulated Tile Processor can only exist after a complete configuration was found and processed by the simulation generator (only then a Simulation class exists), only data meant for streaming during **run** mode is buffered.
Loading and retrieving data can be simulated by calling methods from the Simulation API directly, thus instead of simulating the DMA process, the way it happens inside the real chip, the simulated CCU can just decode the destination address and call the appropriate **get()** or **set()** function from the Simulator.

Below the description is given of how commands are processed by the simulated CCU.

000 - Start configuring

1. Clear the configuration space 'flat_config'.
2. Add all incoming data from lane0 to the 'flat_config' (cycle for cycle).
3. When the tail flit arrives pass the 'flat_config' to the configuration parser and pass the MontiumConfig 'cfg' to the SimulationController. The SimulationController will generate and instantiate a new Simulation instance.

001 - Start loading data

1. Read the target address flit next cycle.
2. Decode the target address flit from previous cycle and write the value to the Simulation with setMem() or setReg() and increment the target address.
3. (a) If the next flit is a data flit then decode the (incremented) target address and write the value to the Simulation with setMem() or setReg() and increment the target address again and continue from (3).
   (b) If the next flit is an address flit then store this new target address and continue from (2).

010 - Start retrieving data

1. Read the start address from the next address flit.
2. Read the 'num_results' field from the next flit and store it in a *"requested results"* counter.
3. Retrieve a result every cycle, by calling the appropriate get() function, and decrement the counter, until all requested results are retrieved.

011 - Get GPo

1. Call sim.getGPO() (from SimulationController) and place the result on lane_out immediately (this represents the General Purpose Output of the Tile Processor).

100 - Run

1. Enable CCU ‘run_mode’.
2. Enable all lanes for streaming IO.
3. Call sim.setGPI(run_param) (from SimulationController).
4. From now on include a sim.step() in every cycle.

101 - Idle

1. Enable ‘hold_mode’, don’t take sim.step() actions anymore.

110 - Reset

1. Enable ‘hold_mode’, don’t take sim.step() actions anymore.
2. Reset all datastructures and variables, also drop Simulation instance.

Chapter 6

Reconfigurable fabric of the Annabelle

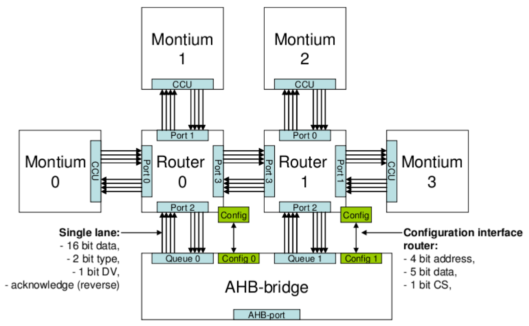


Figure 6.1: Reconfigurable fabric of the Annabelle chip

As mentioned in the introduction, within the *4S* project an *'Annabelle'* prototype chip was developed. This Annabelle contains four Montium Tiles together with other (processing and non-processing) elements. The reconfigurable part of the Annabelle (the four Montium Tiles and the two routers) can be implemented by glue-ing 4 simulation generators together.
Since the simulator is completely single-threaded running 2 simulations in parallel on a dual-core host system runs approximately as fast as a single simulation on a single core (see benchmarks in chapter 7). This should also scale for more than 2 cores, thus the reconfigurable part of the Annabelle chip can probably be simulated quite effectively on a multi-core system.

There are two plausible alternatives for implementing a simulator for the Annabelle reconfigurable fabric, with a trade-off between flexibility and run-time overhead:

1. Run 4 simulations in parallel (thus on 4 separate Java Virtual Machines, JVMs) and create some *router* entity that reads the output data generated by all simulations and passes this as inputs to destination simulations if required. This approach shouldn't be too much of an effort to implement. The only thing that is needed for this to work is the mechanism to pass data from one simulation to the next. This approach is very flexible because a Simulator instance can easily be exchanged with some other program, as long as that program can use the same input/output formats.
   There are however two reasons why this approach wouldn't work very well: (1) all data leaving a Simulation is converted to hexadecimal output and thus needs to be converted back to integer values in the receiving Simulation, this is slow. (2) using 4 separate Virtual Machines would introduce a lot of unnecessary overhead, since it is also possible to run all Simulations on a single JVM.

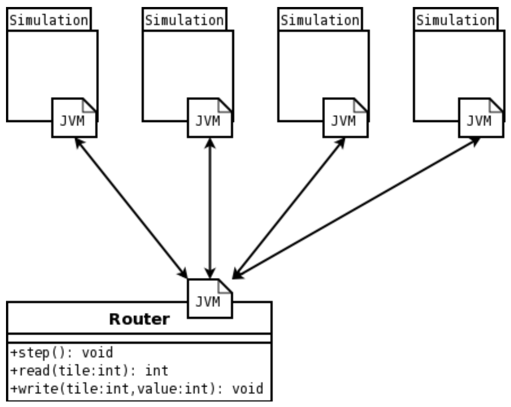


Figure 6.2: UML overview of approach 1

2. A different approach would be to create a class that instantiates 2 router entities and 4 simulations (or actually simulation generators). Important is that these simulations are implemented as separate **Threads**, otherwise the entire Annabelle simulation would become singe-threaded. This approach should be more efficient because it would only require a single JVM, and it can just pass the integers from one thread to another.

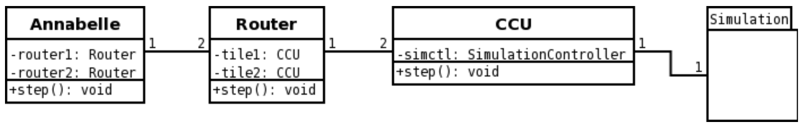


Figure 6.3: UML overview of approach 2

Because simulation speed remains an important issue, the latter alternative is chosen (Use one JVM and start multiple Threads).
The routers in the Annabelle NoC are circuit switched routers[6]. This means that once configured, the routes are fixed (until reconfiguration of the router). Thus entities on the NoC do not need to specify where data is meant to go (packages do not need addressing information). Every router has 4 ports and every port has 4 input lanes and 4 output lanes. The configuration of a router contains a (port,input_lane) tuple for every (router) output lane (where *port* is relative to the port of the output lane). Output lane indexes are calculated via: $port\_lane = port\_out * 4 + lane\_out$ (e.g. port(1)lane_out(2) will have index 6). (port,input_lane) tuples are encoded according to the following scheme:

Encoding	Description
- - - -0	all_zero
00 001	port_out+1 mod 4, lane 0
00 011	port_out+1 mod 4, lane 1
00 101	port_out+1 mod 4, lane 2
00 111	port_out+1 mod 4, lane 3
01 001	port_out+2 mod 4, lane 0
01 011	port_out+2 mod 4, lane 1
01 101	port_out+2 mod 4, lane 2
01 111	port_out+2 mod 4, lane 3
10 001	port_out+3 mod 4, lane 0
10 011	port_out+3 mod 4, lane 1
10 101	port_out+3 mod 4, lane 2
10 111	port_out+3 mod 4, lane 3

Table 6.1: Router port_lane configuration scheme

Simulation of the reconfigurable fabric in pseudo code looks like:

```
    router[0,1].configure(filename)
    while (!finished) do
        router[0,1].getData()
        for (lines in config) do
            router[0,1].copyData(source, dest)
        end
        tile[0,1,2,3].step()
    end
    writeResults()
```

Listing 6.1: Simulating the reconfigurable part of the Annabelle

The simulation will use up to 10 input files:

- 2 router configurations (mandatory)
- Up to 8 input files for Queues 0 and 1 (see Figure 8.1).

The simulation will produce 8 output files for Queues 0 and 1. Together with the input files the simulator will need to know how many cycles it has to run. This parameter can be given on the commandline as well. The input and output files for the lanes are identical to those described in Section 5.2. The router configuration files will contain hexadecimal addresses, data tuples similar to these lane files. An example of a router configuration could look like this:

```
0x0 0x09    // lane 0 will connect to port 2 lane 0 (TP0_in0 <- Q0_out0)
0x4 0x03    // lane 4 will connect to port 2 lane 2 (TP1_in0 <- Q0_out2)
0x8 0x09    // lane 8 will connect to port 0 lane 0 (Q0_in0 <- TP0_out0)
0x9 0x13    // lane 9 will connect to port 1 lane 0 (Q0_in1 <- TP1_out0)
```

Listing 6.2: Example router configuration file

Figure 6.4 shows the above router configuration in a graphical representation.

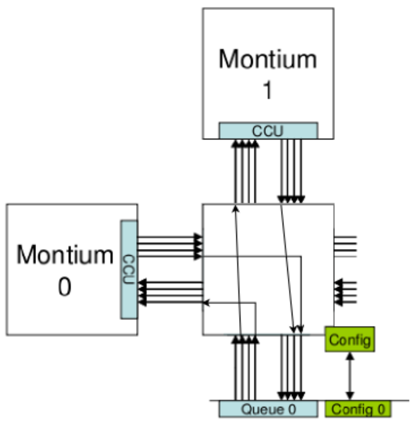


Figure 6.4: Graphical representation of a router configuration.

Chapter 7

Results

The initial goal of this project was to research, design, implement and test a functional simulation generator for the Montium TP which is flexible in terms of architecture parameters and fast in execution (See Chapter 1). The next section discusses the flexibility that was achieved. To determine whether the performance goal was achieved, the simulator generator was benchmarked and compared to the existing 'Simsation' simulator, the results of these benchmarks will be presented in Section 7.2.

7.1 Flexibility

Flexibility in the created simulator can be divided into two different types of flexibility. The first important kind of flexibility is defined by the parameters that can be altered at runtime, by end-users. The second form of flexibility is the flexibility in the software design of the simulator, how easy can it be extended to include function X and how easy is it to embed the simulator in other programs.

7.1.1 Runtime flexibility

For runtime flexibility there was a well defined list created prior to the implementation of the project. All the parameters available in the compiler should also become available in the simulator and it should also be possible to adapt the register file depths of the simulated Tile at runtime. Both these requirements are met by the created simulator and further more it is also possible to change the number of loop counters. In the final design it is also taken into account that future versions might want the number of processing parts and the datapath width to be variable at runtime.

7.1.2 Compile time flexibility

To make extending or altering the datapath functionality flexible the, Visitor pattern was used. The result is that all the code that has to be generated for the ALUs can be found in a single, quite well readable class: the *JavaCodeGenerationVisitor*. Changes to the datapath layout can be made quite fast by simple adjustments in the *ALU* class (see Listing 4.10). New functionality in the functional units need two additions: (1) add a new fu_xyz() helper method in the *FUBaseCode* class (2) add code generation for the correct method call in the *JavaCodeGenerationVisitor*.
These three classes could also be extended to include some conditional statement, creating a simulator that can simulate different instances of Montium Tiles. Listing 7.1 show how easy it is to implement a runtime variable into the simulation generator, that can remove the Functional Units from the simulated Tile.

```
MontiumParameterReader.java:
  TILE_WITHOUT_FUS = parameters.getProperty(TILE_WITHOUT_FUS).equals("true");

Alu.java:
  NullNode nn = new NullNode();
  if (TILE_WITHOUT_FUS){
       mux[0] = new Mux(new Wire(nn, wA, wB, nn, nn));
       mux[1] = new Mux(new Wire(nn, wC, wD, nn, nn));
  } else {
       fu[0] = new FU_std_lvl1(wA, wB, ctf[0], fulogic1);
       fu[1] = new FU_std_lvl1(wC, wD, ctf[1], fulogic2);
       fu[2] = new FU_std_lvl2(new Wire(fu[0]), new Wire(fu[1]), ctf[2], fulogic3);
       fu[3] = new FU_std_lvl2(new Wire(fu[1]), new Wire(fu[0]), ctf[3], fulogic4);

       mux[0] = new Mux(new Wire(nn, wA, wB, new Wire(fu[2]), sel[2]));
       mux[1] = new Mux(new Wire(nn, wC, wD, new Wire(fu[3]), sel[3]));
  }
```

Listing 7.1: Example parametric datapath

7.2 Simulation Performance

7.2.1 Motivation and Overview

Several benchmarks were performed in order to answer the questions:

1. Is Java a viable alternative for creating fast simulations (for Montium TP programs)?
2. Do code generation and functional simulation indeed result in faster simulations?

In the following sections first the benchmark setup will be described. After that, in Section 7.2.3, the performance of simulations generated in Java versus simulations generated in C will be compared. Sections 7.2.4 and 7.2.5 will compare the speed of the new simulation generator with the speed of the already existing 'Simsation' simulator. Finally in Section 7.2.6 the effects of the optimization steps will be presented.

7.2.2 Benchmark setup

- Because different algorithms may behave differently with respect to speed, several different types of algorithms where benchmarked, some variations of a Finite Input Response (FIR) filters, some variations of a Fast Fourier Transforms (FFT) and some other algorithms that where available.

- Most algorithms finish in a few thousands of cycles, therefore for benchmarking they are placed within a loop. Pseudo code for the loop looks like this:

```
init();
while(loop++ < parameter){
    startTime = System.nanoTime();
    while (cycles++ < CYCLES_FOR_ONE_ITERATION){
        step();
    }
    stopTime = System.nanoTime();
    runTime += (stopTime - startTime);
    cycles = 0;
    reset();
}
```

Listing 7.2: Benchmark Loop

- The benchmarks were all performed on the same system, a Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz, which is in fact a dual core CPU but given the single-threaded nature of the simulations one core remains idle when benchmarking.

- All benchmarks were performed after office-hours, to ensure a minimum load on the test system.

- All benchmarks were perform threefold, each time keeping the best result. This choice was made because this eliminates benches where the test-system was performing background tasks during the benchmark. The three different iterations where executed with much time in between, so that (long) background tasks would influence iterations of different benchmarks and not a single one over and over.

- All C sources were compiled with GCC and the following flags: '-Wall -pedantic -std=c99 -O3'

- The C performance was measured using the RDTSC [1] instruction inside the CPU. This gives a cycle-count measurement which, together with the clock speed of the processor, can be translated to a time measurement.

[1]RDTSC stands for Read Time Stamp Counter, in which the TSC is a 64-bit cycle counter available in Pentium class systems

- The Java performance was measured using the System.nanoTime() call, which, according to the JDK documentation, uses the best available timer available in the system.

7.2.3 Performance Java versus C Output

The main difference between C and Java execution lies in the fact that the C program is a native compiled binary, ready to run directly on the system CPU when invoked, and the Java program consists of Java byte-code, which will at first be interpreted by the Java Virtual Machine and compiled at runtime when the JVM detects multiple invocations of code-blocks (Just-in-Time compilation).
This means that the Java version will likely have a slow start-up performance, which will increase until the point where (almost) all code-blocks are actually compiled.
Benchmarks quickly showed that the number of simulated cycles has little or no effect on the performance of the C program. The following performance results where achieved for the different algorithms:

Algorithm	Speed
FIR 5-taps	~7500 kHz
FIR 20-taps	~9800 kHz
FFT 64-points	~6700 kHz
FFT 1024-points	~7500 kHz
FFT 1920-points	~7500 kHz

The fact that the 64-points FFT runs slower than the 1024- and 1920-points FFTs can probably be attributed to the fact that the 64-points FFT is finished in just 205 cycles, so benchmarking will have a relative large overhead restarting the algorithm over-and-over again.
To identify the point where all code is compiled by the JIT compiler, all algorithms where benchmarked multiple times with increasing runtimes (varying from a few seconds up to 5 minutes). In the results of the more complex algorithms one can clearly see the effect of the Just-in-Time compiler. After some time the average performance shows asymptotic behaviour, at the asymptote one can assume (almost) al code has been compiled.

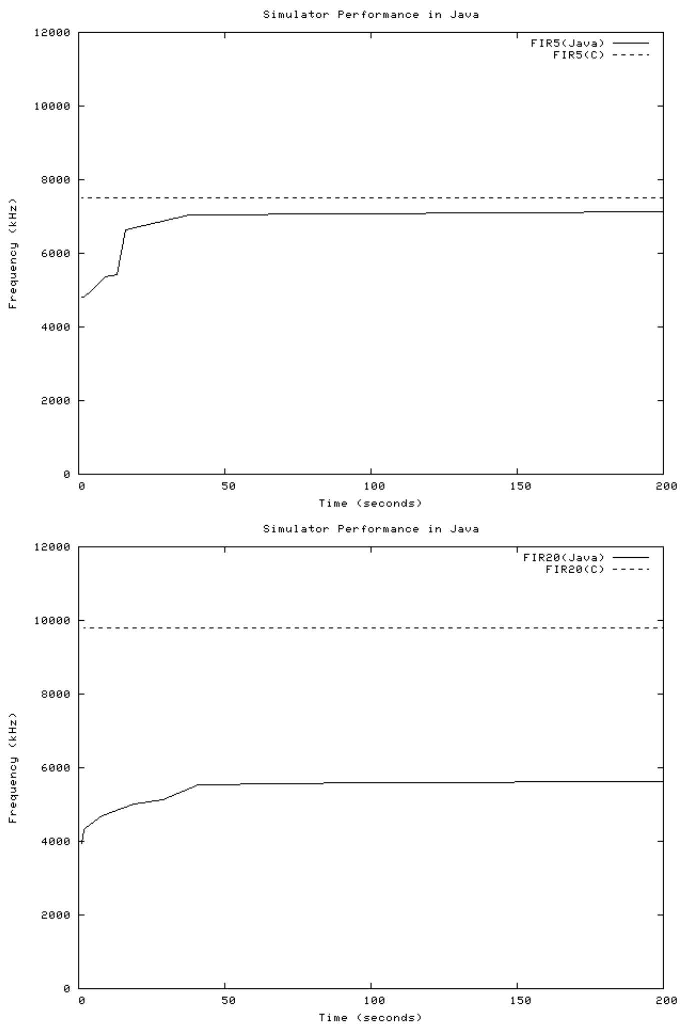
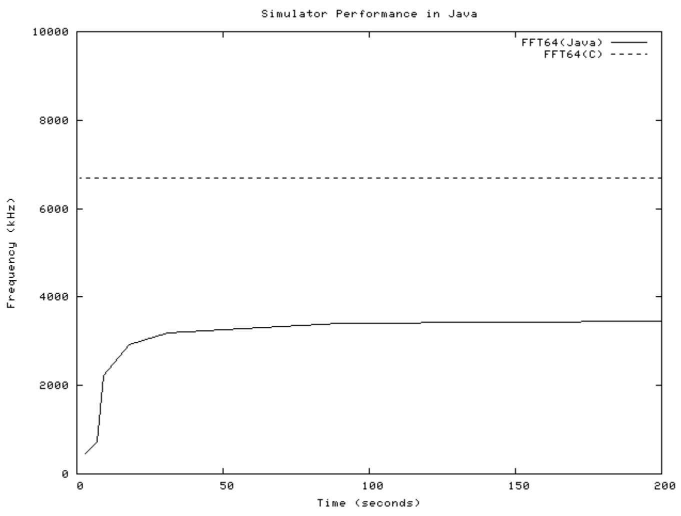


Figure 7.1: Simulation speed for the 5- and 20-taps FIR

The (somewhat unexpected) speed difference between the 5 taps FIR and the 20 taps FIR in Java can probably be attributed to two facts: (1) the switch/case structure (the FIR 5 will only have two levels in the tree while the FIR 20 has 3 levels and (2) the FIR 5 spends 99 % in a single sequencer instruction, which is therefore moved to the first branch in the switch-case tree whereas the FIR 20 spends 'only' 12 % in its most often executed sequencer instruction (See Section 7.2.6). For some reason it seems Java suffers more from these than C does).



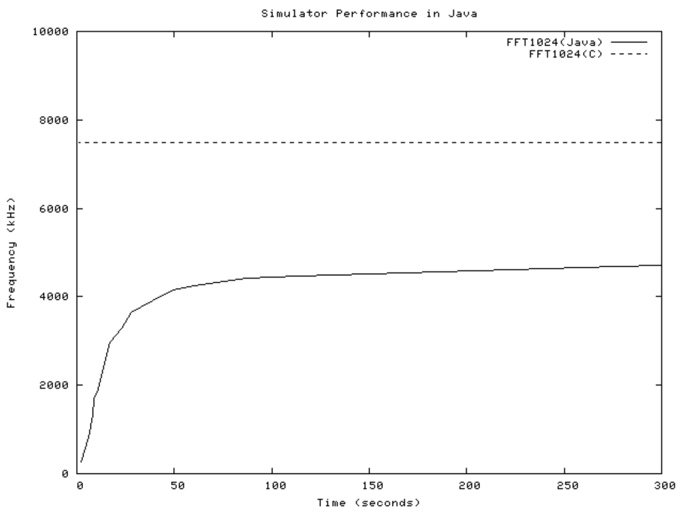


Figure 7.2: Simulation speed for the 64- and 1024-points FFTs

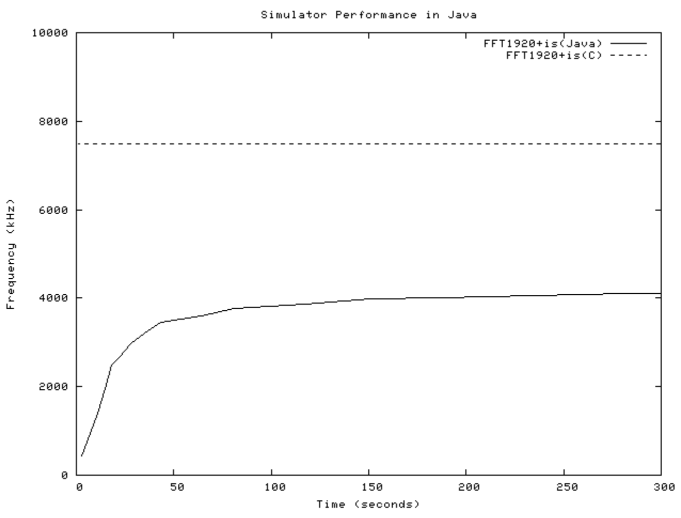


Figure 7.3: Simulation speed for the 1920-points FFT with input-scaling

Summarize Java versus C performance

The above results show that using C as target language provides faster simulations. Neglecting the JIT startup time Table 7.1 summarizes the difference, defined as $\frac{perf_c - perf_{java}}{perf_c} * 100\%$, in performance running the algorithms on the two alternative languages.

Algorithm	C	Java	difference
FIR 5	~7500 kHz	~7200 kHz	4%
FIR 20	~9800 kHz	~5700 kHz	42%
FFT 64-points	~6700 kHz	~3500 kHz	48%
FFT 1024-points	~7500 kHz	~4700 kHz	38%
FFT 1920-points	~7500 kHz	~4100 kHz	45%

Table 7.1: Summarized performance results C versus Java

In the Table one can see that the difference in performance never exceeds a factor of 50%. Performance differences usually only matter when they reach factors in the order of 10 times, because small differences can easily be overcome by using a faster system. Now recalling the benefits of using Java (see Section 3.3): (1) portability, (2) extendability and (3) the hidden generation/compilation process we conclude that Java is a viable choice for implementation of the Simulation Generator. In search for the causes in speed difference between C and Java, some new, final-day changes where introduced to the Java version of the simulation generator:

1. The two-dimensional arrays used to store memory and register variables are handled different by Java and C. In C a two dimension array is always like a matrix, meaning all rows have the same number of elements, thus the C compiler can assign a single block of memory. In Java an array of array-elements is created, which introduces a extra lookup step. Also in Java, because every array (≈row) can have a different number of elements, an extra range-check has to be done. To overcome this all two-dimensional array were refactored to single dimensional arrays (thus mem_left[NUM_PP][MEM_SIZE] → mem_left[NUM_PP * MEM_SIZE]).

2. Despite all testing with the switch-case (see Section 4.2.3), the best option was overlooked: create a single method for every line in the sequencer program and add calls to these from a single switch-case in **step()**. Further investigation showed the Java Virtual Machine knows two different types of switch-case bytecode instructions, the *tableswitch* creates a jumptable eliminating the problem completely. A second benefit of this approach lies in the fact that the JIT compiler compiles entire methods only, thus a block of code in its own method will be compiled sooner than four blocks sharing a single method.

Preliminary tests showed these two changes already improved average simulation speed with approximately 20% on average for the FIR and FFT algorithms. More tweaking, testing and benchmarking can probably eliminate more of the performance gap between te C and the Java version.

7.2.4 Performance compared to ‘Simsation’ without CCU

For benchmarking ‘Simsation’, the same test system was used. The following application versions where used for the benchmarks:

```
Simsation:
  1.9.19 (May 31 2007 11:45:23)
  Copyright 2007 Recore Systems BV

simgen:
  Simulation Generator for Montium TP - build 10.1541
  Build Date: Wed Jun 13 15:20:55 CEST 2007
```

For Simsation a simple ‘.sim’ script was created to loop the different benchmarks:

```
func benchmark1(Path, Name, Cycles, Iterations)
  reset()

  init(Path, Name)
  mark("start")
  var starttime = time()
  var i
  for i = 0 ; i < Iterations ; i++ do
    run(Cycles)
    recall("start")
  end
  var res = time() - starttime
  var total = Cycles * Iterations
  print (Name ## ":␣" ## total ## "␣cycles␣in␣" ## res ## "␣seconds.\n")
end
```

Listing 7.3: Benchmark Loop for Simsation

All benchmarks where again performed in three-fold, each time keeping the best result. Benchmarking the five non-streaming algorithms gave to following results (Simulation Generator results are based on Java code generation):

Algorithm	Simsation	Simulation Generator	speed-up
FIR 5	∼270 kHz	∼7200 kHz	27x
FIR 20	∼267 kHz	∼5700 kHz	21x
FFT 64-points	∼230 kHz	∼3500 kHz	15x
FFT 1024-points	∼251 kHz	∼4700 kHz	19x
FFT 1920-points	∼239 kHz	∼4100 kHz	17x
		avg:	19.8x

Table 7.2: Simulation generator versus ‘Simsation’

7.2.5 Performance compared to 'Simsation' with CCU

For artificially creating benchmarks that run a significant amount of time again a loop was used. The script for the Simsation loop is given in Listing 7.4.

```
func benchmark1(Name, Iterations)
  reset()
  ccu_init_fast(Name)
  \sys\io_enable = true

  print("Loading␣configuration...\n")
  while \tile\in\gpi == 0 do
    run(1)
  end
  mark("configured")

  var clocks = 0
  while (\tile\out\gpo == 0)&&(\sys\clock <1000000) do
    run(1)
    clocks++
  end

  recall("configured")

  var starttime = time()
  var i
  var j
  for i = 0 ; i < Iterations ; i++ do
    for j = 0 ; j < clocks ; j++ do
      run(1)
    end
    recall("configured")
  end

  var res = time() - starttime
  var total = Iterations * clocks
  print (Name ## ":␣" ## total ## "␣cycles␣in␣" ## res ## "␣seconds.\n")
end
```

Listing 7.4: Benchmark Loop for Simsation (CCU)

The script probably requires some explanation though:

- First the simulator is reset and the lane input files are read into a buffer.
- Next the simulator is run until a valid configuration is loaded, this state is saved (so that during benchmarking we don't include configuration time, only real algorithm running time).
- Next the simulation is run until we receive a change in GPO, in other words the run-time (in cycles) for a single algorithm run is determined.
- We restore the simulator state saved previously.
- Now the benchmark loop starts:
  - run the number of cycles needed for the algorithm to finish.
  - restore the saved state.
  - loop
- finally the result is printed to the console.

The result of benchmarking the Simulation Generation in combination with the CCU are shown in Table 7.3.

Algorithm	Simsation	Simulation Generator	speed-up
Streaming FIR	~29 kHz	~1208 kHz	42x
Streaming FFT	~27 kHz	~85 kHz[1]	3x
DRM offset correction	~27 kHz	~856 kHz	32x
DRM offset estimation	~27 kHz	~2695 kHz	100x
DRM Viterbi	~27 kHz	~2836 kHz	100x
Rotation cordic	~26 kHz	~72 kHz[1]	2.8x
Vectoring cordic	~26 kHz	~73 kHz[1]	2.8x

Table 7.3: Simulation generator versus 'Simsation' (streaming)

[1] Because the streaming FFT and the two cordic algorithm kernels only need a few cycles (resp. 52, 19 and 19 cycles) before completion, the overhead of reloading the state after each iteration becomes relatively large. That is why these benchmarks show bad results compared the longer algorithms. This is probably also the case for the Streaming FIR and the DRM offset correction algorithms (resp. 1026 and 794 cycles per run). To verify this assumption, a new algorithm was written in CDL code that performs FIR like calculations (source code can be found in Appendix E).

Algorithm	Simsation	Simulation Generator	speed-up
Long Streaming FIR	~28 kHz	~2680 kHz	96x

Table 7.4: Simulation generator versus 'Simsation' (streaming)

This indeed shows the performance of the new simulation generator, with CCU simulation, to be around 2600 kHz assuming the algorithm runs for a reasonable amount of time (more than 60 seconds).

7.2.6 Improvements achieved by optimizations

The optimization steps in Section 4.7 follow two different principles. The first principle, **selective code generation**, includes selective code generation for: (1) register file pointer (=read address) assignments, (2) selective memory address restorations and most importantly (3) the selective generation of arithmetic expressions for the ALU outputs. The other principle, the **hotspot detection**, doesn't affect the amount of code generated but only moves a block of source code to a different location for faster execution. The first principle works for streaming and non-streaming algorithms, the second only works when simulating non-streaming algorithms (see Section 4.7.4). Benchmarks in this Chapter are done with:

1. Optimizations disabled.
2. Selective code generation optimizations enabled.
3. Selective code generation optimizations and the hotspot detection enabled. (only for non-streaming algorithms)

Tables 7.5 and 7.7 show some statistics for the tested algorithms/Montium configurations. They should provide some idea of the complexity of the algorithms. Tables 7.6 and 7.8 show some statistics about the optimization results for the same algorithms. The columns in Tables 7.6 and 7.8 show (from left to right): (1) the algorithm name, (2) the number arithmetic expression needed to do all calculations, (3) the number of expressions that could be removed from these by the optimization process, (4) the percentage of expressions the optimizer removed, (5) the sequencer instructions identifier to be executed most often together with the number of invocations to that sequencer instruction during the detection run and finally (6) the percentage the algorithm is expected to be executing that *hotspot* instruction.

Algorithm	#lines in seq. program	#lines of code generated	#cycles/run
FIR 5	7	1131	1027
FIR 20	22	4910	2055
FFT 64-points	86	10892	205
FFT 1024-points	174	20594	5141
FFT 1920-points	204	22253	15035

Table 7.5: Statistics (non-streaming) algorithms

Algorithm	#arith.	#removed	%	*HS*:	#inv.	%
FIR 5	60	38	63%	3:	1023	99%
FIR 20	495	167	34%	6:	256	12%
FFT 64-points	1275	601	47%	4:	15	7%
FFT 1024-points	2550	1196	47%	4:	255	5%
FFT 1920-points	3165	1581	49%	18:	1536	10%

Table 7.6: Optimization Effects (non-streaming algorithms)

Algorithm	#lines in seq. program	#lines of code generated	#cycles/run (without config)
Streaming FIR	6	1067	1025
Streaming FFT	31	4016	52
Streaming DMA	103	11792	
DRM offset correction	14	1698	794
DRM offset estimation	34	3782	10956
DRM Viterbi	80	9009	6238
Rotation cordic	10	1341	19
Vectoring cordic	10	1320	19

Table 7.7: Statistics (streaming) algorithms

Algorithm	#arith.	#removed	%
Streaming FIR	60	38	63%
Streaming FFT	420	244	58%
Streaming DMA	1815	1435	79%
DRM offset correction	135	72	53%
DRM offset estimation	435	229	52%
DRM Viterbi	1260	1011	80%
Rotation cordic	71	51	71%
Vectoring cordic	75	55	73%

Table 7.8: Optimization Effects (streaming algorithms)

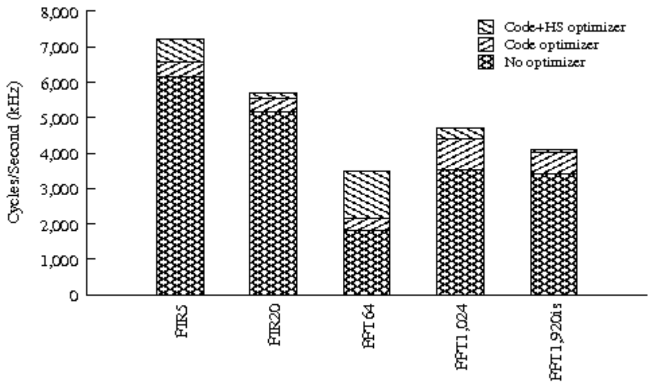


Figure 7.4: Optimization Effects on Performance (non-streaming)

Figure 7.4 shows the effect the two optimizations have on the non-streaming algorithms. On average one could say that both optimization steps improve simulation speed by 15% (see Table 7.9).

Algorithm	NO_OPT	OPT	OPT+HS	gain OPT	gain HS
FIR 5	6140	6580	7200	7.2 %	9.4 %
FIR 20	5170	5550	5700	7.4 %	2.7 %
FFT 64-points	1811	2150	3500	18.7 %	62.8 %
FFT 1024-points	3506	4400	4700	25.5 %	6.8 %
FFT 1920-points	3420	4000	4100	17.0 %	2.5 %
			avg:	15.1 %	16.9 %

Table 7.9: Optimization Effects on Performance (kHz) (non-streaming)

7.2.7 Running multiple simulations in parallel

The Montium Tile chip is designed to be part of a multi-processing System-on-Chip processor, e.g. the Annabelle prototype chip has four Montium Tiles in its reconfigurable part. Therefore it is useful if multiple instances of the simulator can be run in parallel. Given the fact that modern days general purpose processors used in simulation host systems become multi-cored systems as well, it seems reasonable to use these multiple cores to simulate multiple Montium instances. Some benchmarks where performed to test the scalability of the simulator on a multi-core system (a dual core Intel Core 2 Duo was used).

Algorithm	1 simulation	2 simulations	4 simulations
FIR 5	7200 kHz	2x ∼7100 kHz	4x ∼3500 kHz
FFT 1920-points	4100 kHz	2x ∼3800 kHz	4x ∼1900 kHz

Table 7.10: Running multiple instances in parallel

The above table shows the performance of the simulator scales nice per extra host processor core, and divides nicely by two if two instances are run on a single core.

Chapter 8

Conclusion and Recommendations

8.1 Conclusion

The benchmarks presented in Chapter 7.2 show that a simulator was created that can simulate the Montium Tile Processor at speeds of several millions of simulated cycles per second. The provided results show the simulation generation technique to be on average almost 20 times faster than the existing simulator. When the CCU is included in the simulation, to perform streaming simulations, the difference in performance becomes even bigger.
All runtime flexibility requirements were met in the simulation generator. Later an example was given how the flexibility could even be extended to include flexibility in terms of datapath connectivity.
In the overall software design it was tried to keep everything flexible and modular, extending the software with new components, user interfaces, and functionality should rise no fundamental problems.

8.2 Recommendations for Future Work

Though the simulation generator is feature complete, including the simulation of the CCU, and all initial goals set for the project where achieved some ideas for future extensions remain available. The next pages will present some ideas.

8.2.1 Simulation of a complete Annabelle SoC

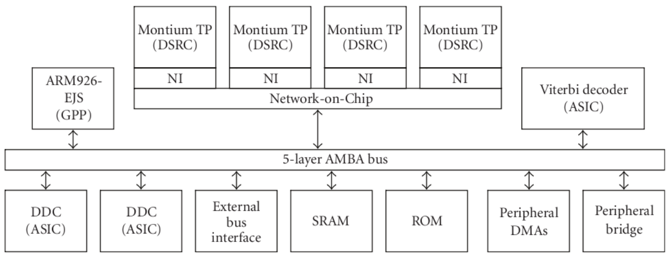


Figure 8.1: Block diagram of the Annabelle chip

In Chapter 6 the implementation of a Simulator for the reconfigurable part of the Annabelle System-on-Chip is described. The entire Annabelle however consists of more components. An idea for future projects could be the creation on a single simulation environment that simulates this entire SoC. Probably the most interesting step would be to including an ARM simulator. That way applications could be simulated that run on the ARM and delegate computational intensive blocks/kernels to the Montium Tiles.

8.2.2 Design Parameters in CFG Files

Now the compiler and the simulator both have support for runtime configurable Montium design parameters, a consistency issue arises. A binary, compiled for a Montium instance with a sequencer size of 256 positions, will only function correctly in the simulator if this same design parameter is given to the simulator, otherwise the configuration parser will fail. A simple but effective way to fix this would be to extend the CFG format with a header containing those parameters. The compiler can add these parameters to the CFG and the simulator can simply read them. This assures the compiler and the simulator will use identical design parameters. Of course this line should be ignored (or left out) during the configuration of a real chip.
Because this header is not passed to a real Montium chip, it it not necessarily limited to a 16 bit value. A simple suggestion for the format of such a header could be:

```
# ver | CR_ALU_HEIGHT | CR_GINT_HEIGHT | ..... ..... | DATAPATH_WIDTH | NUM_PP
```

where the '#' will tell the simulator that this is a parameter line, 'ver' indicates the (VHDL) version of the Montium Tile (maybe useful in the future) and the other parameters are the design parameters from Appendix B. The header for the default Montium instance would become:

```
# 01.09.01 | 8 | 4 | 8 | 8 | ..... ..... | 4 | 4 | 16 | 5
```

Another option would be to only include parameters that differ from the default parameters, something like: `# 01.09.01 | 14:512` could for example mean: assign 512 to parameter 14 (where all parameters would get a unique identifier).

8.2.3 'Undefined' values

In the current implementation primitive integers are used for all internal variables like memories, registers and so on. In Java this means they all get to be initialized to be 0. In real hardware however an uninitialized location can have any value. These initial 0's can possible hide some problems in developed algorithms from the developer. To detect these problems, the simulator should be extended with the notion of uninitialized variables, several implementations can be thought of for realizing this:

- Assign all variables with (pseudo-)random values at start-up, this will however make the debugging process for the developer more cumbersome, because he can not identify a produced variable value from a randomly assigned one thus he will have difficulties keeping track of what the simulation has really produced. So this approach is probably not a good idea.
- Initialize all these variables to some fixed value that lies outside the range of actually allowed values, e.g. the Java constant for the maximum value of a integer (Integer.MAX_VALUE). This has the advantage that all arithmetic operations on them remain valid, thus there will be no immediate effect on simulation performance. A big disadvantage however is that multiplying this value with 0, or adding 1 to this value will result in a valid value that lies within the allowed range again (Thus cannot be identified as invalid anymore). One could design some checks for this though.
- Keep track of all assigned locations in memories and registers and give warning messages when the simulator accesses a variable that was never assigned. This is probably the best alternative, it will however have some impact on the simulator speed so it is probably a good idea to make it available as a start-up parameter for the simulation generator, allowing the end-user to choose between speed and safety.

8.2.4 Generation of Java byte-code

The project was intended to support different target languages, and was implemented to generate either Java or C code. The Java version needs to compile the java code internally. This results in a dependency on the Java

Development Kit (JDK) to be installed on the system. An alternative for this could be to generate Java byte-code instead of normal java code during the generation process. The dependency would then be reduced to a Java Runtime Environment (JRE) and as an added bonus it would remove the time needed for compilation whenever a new configuration is loaded into the Simulator.

8.2.5 More integration with other tools

For a DSP developer, as for every programmer, development becomes easier and thus less time consuming if the tools he needs are quickly accessible and are easy to familiarize with. For this, the available tools, like the compiler, the simulator(s) and a source editor should be integrated into a single familiar environment. Recore Systems is already looking into this right now.

8.2.6 Create a complete test-suite

To detect errors or bugs in the simulator(s), now and after future extensions, it is necessary to have access to a test suite that provides full coverage of all functions available. Right now there is a basic test-bench that can be used to perform some tests, but this test-bench does not yet provide full coverage of all capabilities within the Montium TP. An ideal test suite would probably be one that will work inside both the 'Simsation' simulator and in the simulation generator described in this report. This might however be difficult because that would mean that the entire test-suite has to be written in the form of small Montium configurations (the only 'language' both simulators understand).

8.2.7 Create a framework to integrate the simulation generator with a Scripting environment

Something many developers would probably like to have, and would also be useful for creating the test-suite mentioned above, is the possibility to embed the simulator in a some sort of scriptable environment. For several scripting languages there already are Java bindings available which should make it possible to create some interaction framework for the simulation generator, examples of available scripting languages are python[15] and Perl[16]. Another option would be to use *BeanShell*[14] a scripting environment specially designed for Java that understands standard Java statements, expressions, and method declarations.

Bibliography

[1] Paul M. Heysters *Coarse-Grained Reconfigurable Processors - Flexibility meets efficiency.* Ph.D. Dissertation, University of Twente, Enschede, The Netherlands, September 2004, ISBN 90-365-2076-2.

[2] Paul M. Heysters *Montium Tile Processor Design Specification.* Draft 01.07.xx, 24-Jun-05

[3] Gerard J. M. Smit, Andre B. J. Kokkeler, Pascal T. Wolkotte, Philip K. F. Holzenspies, Marcel D. van de Burgwal, and Paul M. Heysters *The Chameleon Architecture for Streaming DSP Applications.* EURASIP Journal on Embedded Systems, Volume 2007, Article ID 78082, 10 pages, doi:10.1155/2007/78082

[4] G.J.M. Smit, E.Schuler, J.E.Becker, J.Quevremont, and W. Brugger *Overview of the 4S project.* Proceedings of the International Symposium on System-on-Chip (SoC), pp. 70 - 73, Tampere, Finland, November 2005.

[5] M.D. van de Burgwal *Hydra Design Specification - multiplexer version.* Draft, version 01.09.xx, 21-Aug-2005, Last saved 22-Feb-2006

[6] P. Wolkotte *Circuit switched router design specification.* Draft, version 02.04.xx, 19-April-2005

[7] Guo, Y. and Hoede, C. and Smit, G.J.M. *A Column Arrangement Algorithm for a Coarse-grained Reconfigurable Architecture.* In: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'06), 26-29 Jun 2005, Las Vegas, Nevada, USA. pp. 117-122. CSREA Press. ISBN 1-932415-74-2

[8] S.G. Aly and A.M. Salem *Observability-based RTL simulation using Java.* System-on-Chip for Real-Time Applications, 2004. Proceedings. 4th IEEE International Workshop on, Volume , Issue , 19-21 July 2004 Page(s): 179 - 182

[9] S.G. Aly and A.M. Salem *Modeling and simulation of digital circuits using JAVA.* Electrical, Electronic and Computer Engineering, 2004. ICEEC apos;04. 2004 International Conference on, Volume , Issue , 5-7 Sept. 2004 Page(s): 47 - 52

[10] Tor E. Jeremiassen *Sleipnir. An instruction-level simulator generator.* Computer Design, 2000. Proceedings. 2000 International Conference on, Volume , Issue , 2000 Page(s):23 - 31

[11] Nozomu Togawa, Kyosuke Kasahara, Yuichiro Miyaoka, Jinku Choi, Masao Yanagisawa and Tatsuo Ohtsuki *A Retargetable Simulator Generator for DSP Processor Cores with Packed SIMD-type Instructions.* IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences Vol.E86-A No.12 pp.3099-3109, 2003/12/01, ISSN: 0916-8508

[12] Stefan Pees, Andreas Hoffmann and Heinrich Meyr *Retargetable compiled simulation of embedded processors using a machine description language.* ACM Transactions on Design Automation of Electronic Systems (TODAES) archive, Volume 5 , Issue 4, pages: 815 - 834 , 2000, ISSN:1084-4309

[13] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr and A. Hoffmann *A universal technique for fast and flexible instruction-set architecture simulation.* Design Automation Conference, 2002. Proceedings. 39th Volume , Issue , 2002 Page(s): 22 - 27

[14] *BeanShell - Lightweight Scripting for Java*
http://www.beanshell.org/

[15] *The Jython Project*
http://www.jython.org/

[16] *Inline::Java*
http://aspn.activestate.com/ASPN/CodeDoc/Inline-Java/Java.html

Appendix A

The Simulator API

```
/**
 * @return the name of this simulation
 */
public String getName();

/**
 * @return the generation date/time of this simulation
 */
public String getTime();

/*
 * @return the General Purpose Output bits
 */
public int getGPO();

/*
 * @return the General Purpose Input bits
 */
public int getGPI();

/*
 * @return the Program Counter to be executed
 */
public int getPC();

/*
 * @return the Tile Status Bits
 */
public int getSB();

/*
 * @param mem the selected memory
 * @param address the selected address
 * @return the value in mem[address]
 */
public int getMem(int mem, int address);

/*
 * @param mem the selected memory
 * @param address the selected address
 * @return the value in expected_mem[address]
 */
public int getExpect(int mem, int address);

/*
 * @param pp the selected processing part
 * @param rf the selected register file -> 0=A .. 3=D
 * @return the value in reg[rf][current_pointer]
 */
public int getReg(int pp, int rf);

/*
 * @param pp the selected processing part
 * @param rf the selected register file -> 0=A .. 3=D
 * @return the current pointer for reg[rf]
 */
public int getRfPtr(int pp, int rf);

/*
```

```
 * @param pp the selected processing part
 * @param rf the selected register file -> 0=A .. 3=D
 * @param addr the selected address
 * @return the value in reg[rf][addr]
 */
public int getReg(int pp, int rf, int addr);

/*
 * @param alu the selected ALU
 * @return the value for ZA[alu] (east-west)
 */
public int getZA(int alu);

/*
 * @param alu the selected ALU
 * @param res the selected result -> 1 or 2
 * @return the value for res_out'res'[alu]
 */
public int getResOut(int alu, int res);

/*
 * @param sel the selected Loop Counter -> 1..4
 * @return the value for lc[sel]
 */
public int getLC(int sel);

/*
 * @param lane the selected lane_out
 * @return the value for lane_out[lane] (tile->ccu)
 */
public int getLane(int lane);

/*
 * @param lane the selected lane_in
 * @return the value for lane_in[lane] (ccu->tile)
 */
public int getLaneIn(int lane);

/*
 * @param lane the selecten lane_in
 * @param val the value to put on the lane
 * @return true if succeeded
 */
public boolean putLane(int lane, int val);

/*
 * @param lane the selected lane_out
 * @return the flit type for the lane_out data
 */
public int getLaneFlit(int lane);

/*
 * @param lane the selected lane_out
 * @return true if there is data available
 */
public boolean isDataWaiting(int lane);

/*
 * @param which the selected memory
 * @return an entire memory -> 0..9
 */
public int[] getmem(int which);

/*
 * Set Tile state, for snapshot recovery
 */
public void setTileState(TileState state);

/*
 * @return the current Tile state
 */
public TileState getTileState();

/*
 * @return all errorous memory locations
 */
public int[][] verify();

/*
 * reset the internal Tile state
 */
public void init();

/*
```

```
 * write all memories and registers
 * to files in /output/
 */
public void dump();

/*
 * fast reset, without reloading
 */
public void clear();

/*
 * set GPI value
 */
public void setGPI(int val);

/*
 * set a single bit in GPI
 */
public void setGPIbit(int pos);

/*
 * clear a single bit in GPI
 */
public void clrGPIbit(int pos);

/*
 * override an active register value
 */
public boolean setReg(int pp, int rf, int val);

/*
 * override any register value
 */
public boolean
setReg(int pp, int rf, int addr, int val);

/*
 * override any memory value
 */
public boolean
setMem(int mem, int addr, int val);

/*
 * override an east-west connection
 */
public boolean setEast(int pp, int val);
public boolean setWest(int pp, int val);

/*
 * get the ouput of an AGU
 * @param mem the selected AGU
 */
public int getAddr(int mem);

/*
 * step a single cycle, update the entire
 * internal state
 */
public int step();
```
Listing A.1: Simulation API

Appendix B

Montium Design Parameters

Parameters supported by both the compiler and the simulation generator:

parameter	default	description
CR_ALU_HEIGHT	8	Height of CR_ALU Reg.
CR_GINT_HEIGHT	4	Height of CR_GINT Reg.
CR_LINTM_HEIGHT	8	Height of CR_LINTM Reg.
CR_LINTR_HEIGHT	8	Height of CR_LINTR Reg.
CR_MEML_HEIGHT	16	Height of CR_MEML Reg.
CR_MEMR_HEIGHT	16	Height of CR_MEMR Reg.
CR_RFAB_HEIGHT	16	Height of CR_RFAB Reg.
CR_RFCD_HEIGHT	16	Height of CR_RFCD Reg.
CR_SIO_HEIGHT	8	Height of CR_SIO Reg.
DEC_ALU_HEIGHT	32	Height of DEC_ALU Reg.
DEC_GINT_HEIGHT	32	Height of DEC_GINT Reg.
DEC_LINT_HEIGHT	32	Height of DEC_LINT Reg.
DEC_MEM_HEIGHT	64	Height of DEC_MEM Reg.
DEC_REG_HEIGHT	32	Height of DEC_REG Reg.
IM_SEQ_HEIGHT	256	Height of IM_SEQ Reg.
IR_AGU_BASEL_HEIGHT	4	Height of IR_AGU_BASEL Reg.
IR_AGU_BASER_HEIGHT	4	Height of IR_AGU_BASER Reg.
IR_AGU_BASE_WIDTH	4	Width of IR_AGU_BASE Reg.
IR_AGU_MASKL_HEIGHT	4	Height of IR_AGU_MASKL Reg.
IR_AGU_MASKR_HEIGHT	4	Height of IR_AGU_MASKR Reg.
IR_AGU_OFFSL_HEIGHT	16	Height of IR_AGU_OFFSL Reg.
IR_AGU_OFFSR_HEIGHT	16	Height of IR_AGU_OFFSR Reg.
MEM_ADDR_WIDTH	10	Width of memory addresses.

Parameters supported at runtime by the simulation generator, but not currently supporter by the compiler:

parameter	default	description
NUM_LOOP_COUNTERS	4	The number of loop counters in the tile.
RF_DEPTH	4	Depth of the register files.

Parameters the simulation generator can ‘easily’ be extended with:

parameter	default	description
DATAPATH_WIDTH	16	The width of the datapath in bits.
NUM_PP	5	The number of Processing Parts in a Tile.

Appendix C

Simulation Datastructure

```
 /* A name for this Simulation */
public String program_name, program_path and program_date;

 /* Constants for this Montium TP instance */
private static final int DATAPATH_WIDTH = 16;
private static final int DATAPATH_MAXINT = 32767;
private static final int DATAPATH_MININT = -32768;
private static final int MEM_ADDR_WIDTH = 10;
private static final int MEM_WIDTH_WO_BASE = 6;
private static final int MEM_MINUS_ONE = 1023;

 /* Montium TP Variables, HEIGTH = 256, NUM_LOOP_COUNTERS = 4 */
private SequencerStateMachine ssm = new SequencerStateMachine(256, 4);

 /* Global Interconnects, for CCU communication */
private int[] GB = { 0,0,0,0,0,0,0,0,0,0 };
private int[] lane_out_buf = new int[4];
private boolean[] data_waiting = new boolean[4];
private int[] lane_in_buf = new int[4];
private int FType = 0;
private boolean lane_in_ready[] = { false, false, false, false };

 /* output latches for 5PPs */
private int[] res_out1 = new int[5];
private int[] res_out2 = new int[5];

 /* 'east' inputs 5PPs */
private int[] ZA = new int[6];

 /* temp variables for wsb storage for 5PPs */
private int[] wsbFU = new int[5];

 /* register files for 5PPs, 4 locations deep */
private int[][] rfA = new int[5][4];
private int[][] rfB = new int[5][4];
private int[][] rfC = new int[5][4];
private int[][] rfD = new int[5][4];

 /* register file pointers for 5PPs */
private int[] rfA_rd_pointer = new int[5];
private int[] rfB_rd_pointer = new int[5];
private int[] rfC_rd_pointer = new int[5];
private int[] rfD_rd_pointer = new int[5];

 /* Local Memories for 5PPs */
private int[][] mem_left = new int[5][1<<MEM_ADDR_WIDTH];
private int[][] mem_right = new int[5][1<<MEM_ADDR_WIDTH];

 /* Memory Address Registers for 5PPs */
private int[] memL_address = new int[5];  // current address registers
private int[] memR_address = new int[5];
private int[] memL_base = new int[5];    // current base registers
private int[] memR_base = new int[5];
```

Listing C.1: Basic Simulation Datastructure

Appendix D

FU Helper Methods

```
private int[] sbmask = { 1,2,4,8,16 };
private int[] ws = new int[5];

private final int runtimeMux(int a, int b, int pp){
  if (BitUtil.checkBit(g_SB,pp)) { return b; }
  return a;
}

private final int setZeroNeg(int value, int pp, int status){
  if (value == 0) { ws[pp] |= 4; } // zero: 100
  else if (value < 0) { ws[pp] |= 2; }  // negative: 010
    switch(status){
    case 0: if (ws[pp] >= 4)        { g_SB |= sbmask[pp]; }; // =0 (1--)
        break;
    case 1: if (ws[pp] < 2)         { g_SB |= sbmask[pp]; }; // >0 (00-)
        break;
    case 2: if ((ws[pp] & 2) == 0) { g_SB |= sbmask[pp]; }; // >=0 (-0-)
        break;
    default: if ((ws[pp] & 1) == 1) { g_SB |= sbmask[pp]; }; // (--1)
        break;
  }
  return value;
}

private final int fu_lsl(int a, int b){
  int shift = b & 15;
  if (b == 0) { return a; }
  int bits = a & DATAPATH_MAXINT;
  int result = BitUtil.getBits(bits<<shift,0,16);
  int sign = (result > DATAPATH_MAXINT) ? -1 : 1;
  int fill = (sign == -1) ?
      (result | (-1 ^ (DATAPATH_MAXINT))) :
      BitUtil.getBits(result,0,15);
  return fill;
}

private final int fu_lsr(int a, int b){
  return a >> (b & 15);
}

private final int fu_asl(int a, int b){
  return fu_lsl(a, b);
}

private final int fu_asr(int a, int b){
  int shift = b & 15;
  if (a >= 0){ return a >> shift; }
  else { return (a >> shift) | (-1 << (DATAPATH_WIDTH - b)); }
}

private final int fu_min(int a, int b){
  return Math.min(a,b);
}

private final int fu_max(int a, int b){
  return Math.max(a,b);
}
```

```
private final int fu_add_wsb(int a, int b, int pp){
  int result = a+b;
  if ( ((a>0) && (b>0) && (result<0)) ||
       ((a<0) && (b<0) && (result>0)) ){
         ws[pp] = 1;
  }
  return result;
}

private final int fu_add(int a, int b){
  return a+b;
}

private final int fu_sadd_wsb(int a, int b, int pp){
  int result = Math.min(a+b, DATAPATH_MAXINT);
  if ( (a+b) > result ){ ws[pp] = 1; }
  return result;
}

private final int fu_sadd(int a, int b){
  return Math.min(a+b, DATAPATH_MAXINT);
}

private final int fu_neg_wsb(int a, int b, int pp){
  int result = -a;
  if ( (result & DATAPATH_MAXINT) == (DATAPATH_MAXINT) ){ ws[pp] |= 1; }
  return result;
}

private final int fu_neg(int a, int b){
  return -a;
}
```

Listing D.1: FU Helper Methods

Appendix E

Source used for Benchmarking the CCU

```
// Multiply-accumlate operation
proc mac
  rep i <- 1 2 3 4 5
    alu (p.i.a1 fmul p.i.c1) sadd p.i.d1 -> p.i.o1
  end
end

//-------------------------------

start:        clock
                frz
                jnc    gpi1 start
              default
                weak   agu p1m1 p5m1 =0 |=0
              clock
                agu    p1m1=0

                llc    lc3 100
loop1:        clock
                llc    lc2 1022
                mov    ext2 -> p1a1 p2a1 p3d1 p4c1 p5d1
loop2:        clock
                llc    lc1 1022
                mov    ext2 -> p1c1 p2d1 p3c1 p4d1 p5a1
macloop:      clock
                mov    ext2 -> p1d1 p2c1 p3a1 p4a1 p5c1
                call   mac
                mov    data p1o1 -> ext2
                mov    data p2o1 -> ext3
                mov    data p3o1 -> ext4
                loop lc1 macloop
              clock
                loop lc2 loop2
              clock
                loop lc3 loop1

              clock
                set    gpo1       // signal 'Done'
              clock
                frz               // freeze the PPA
                clr    gpo1       // clear 'Done'
              clock
                frz               // freeze the PPA
                jmp    start      // jump to start of program
// end
```

Listing E.1: CDL Source used for Benchmarking the CCU