Bachelor Thesis

Design and Implementation of a Dataflow-Processor for Synchronous Programs

Stefan Dyckmans and Adrian Willenbücher

17.03.2008

Supervisors:
Prof. Dr. Klaus Schneider
Dr. Jens Brandt

Embedded Systems Group
Department of Computer Science
University of Kaiserslautern

Hereby we declare that we have independently composed the thesis at hand. The sources and auxiliary means used have been marked in the text and are exhaustively given in the bibliography.

17.03.2008 – Kaiserslautern

Stefan Dyckmans and Adrian Willenbücher

Abstract

Synchronous languages are well suited to implement reactive systems, which are increasingly widespread and important. Traditionally, these systems were either synthesized for hardware or compiled for execution on a microprocessor; however, both approaches possess significant disadvantages.

In this thesis we present a new processor architecture for the execution of synchronous programs. Our architecture is dataflow-oriented and can exploit the parallelism inherent in those programs. Based on this concept, we designed a processor and wrote a compiler which reads synchronous programs in the form of guarded actions and translates them for the processor. Furthermore, a simulator has been implemented in software, which can be used to measure the performance of the processor and the effects of optimizations.

Synchrone Sprachen eignen sich gut zur Realisierung von reaktiven Systemen, welche zunehmend verbreitet und bedeutend sind. Bisher wurden solche Systeme entweder für Hardware synthetisiert oder zur Ausführung auf einem Mikroprozessor kompiliert; beide Ansätze weisen jedoch erhebliche Nachteile auf.

In dieser Arbeit stellen wir eine neue Prozessorarchitektur zur Ausführung von synchronen Programmen vor. Unsere Architektur ist datenflussorientiert und kann die Parallelität dieser Programme nutzen. Darauf basierend haben wir einen Prozessor entworfen und einen Compiler geschrieben, der synchrone Programme in Form von Guarded Actions liest und für den Prozessor übersetzt. Des Weiteren wurde ein Simulator in Software implementiert, mit dessen Hilfe die Performanz des Prozessors und die Auswirkungen von Optimierungen gemessen werden können.

Contents

Chapter 1

Preface

1.1 Goal

Synchronous languages are well suited to implement reactive systems. These synchronous programs can, due to their structure, be synthesized for hardware (either a custom implementation, or an FPGA), or compiled and executed by a microprocessor. However, these approaches have disadvantages:

- When implemented in silicon, the reactive system is fixed, i.e. it is not possible to change the program once the chip has been produced. Also, development cycles for silicon implementations are very long and expensive.
- Although FPGAs solve these problems, their disadvantages are higher costs per item (for high quantities), lower performance, higher power consumption and larger die sizes, compared to custom-made chips.
- Microprocessors, albeit cheap and reprogrammable, are less power efficient and slower than silicon implementations, thus making them unsuitable for some embedded systems. Furthermore, synchronous programs exhibit a lot of parallelism, which is wasted when they are executed sequentially by a microprocessor.

For certain applications, there are strict requirements for performance or power consumption, so that other aspects (like programmability) are impaired or have to be sacrificed. For example, if a certain reactive system in a car has high demands to reaction delay and thus can't be realized with a microprocessor, it has to be implemented in silicon. If this implementation turns out to be erroneous, the chip has to be replaced in every sold car, and a new chip has to be designed and fabricated, resulting in very high costs.

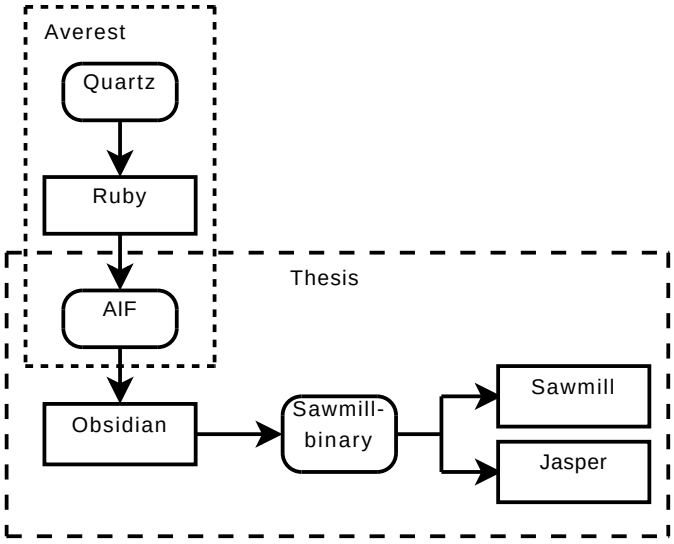


Figure 1.1: Tools and formats related to this thesis. The dotted box contains the Averest components, the dashed box shows the components in this thesis, as well as the relationships between them.

It was our goal to create a processor architecture that is inspired by data flow computers and tailored to synchronous programs. For this architecture, we designed an implementation for an FPGA ("Sawmill"), a software simulator for this implementation ("Jasper") and a compiler that transforms synchronous programs in the form of guarded actions to binaries suitable for execution by our processor ("Obsidian"). These components are designed to work together with the Averest framework [1], as depicted in figure 1.1. Obsidian reads AIF-modules and compiles them to binaries. These can be executed either by Sawmill or by Jasper.

1.2 Reactive Systems

To understand synchronous languages, first the concept of reactive systems has to be explained. According to [10], a system is reactive related to the inputs if it leads to an output determined through the input and the state of the system. Reactive systems are nonterminating, and interact continuously with their environment. Thereby the action is initiated by the environment. Also reactive system satisfies strong real time requirements (see also [11]).

[1] The Averest Framework, `http://www.averest.org`

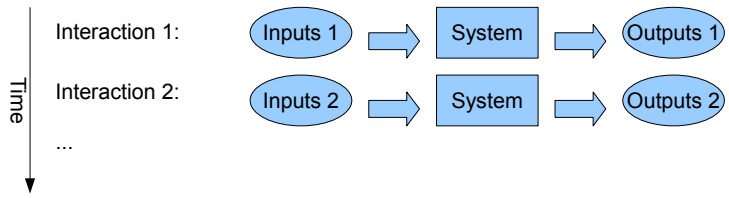


Figure 1.2: Interactions in a reactive system

Lots of embedded system, for example systems in a car that observe the vehicle state, like ESC (Electronic Stability Control) and so on, are reactive systems. But they can be found in communications protocols or operating systems as well.

1.2.1 Real Time System

Reactive systems have a exactly defined time response. Therefore they are real time systems. A real time system is a system that assures a response within a certain time interval. Strong real time system interprets the excess of the specified time as failure.

1.2.2 Concurrency

Normally reactive systems are realized in a concurrent way, at least when they are embedded systems. This means that several output values are evaluated at the same time.

1.2.3 Continuity

A transitional system reads the input variables, computes the outputs and terminates. In contrast to that, reactives systems read the inputs in every interaction, and compute a new output (as shown in figure 1.2).

1.2.4 Perfect Synchrony

Is it is talked about perfect synchrony, if all reactions of a system are executed in zero time. This means that all outputs are generated at the same time as the inputs appear. Of course this only an idealized model. In reality zero time means that the output is produced, before the next interactions starts.

1.3 Synchronous Languages

A Synchronous language is a programming language, which is especially dedicated to describe reactive systems. There are several synchronous languages, the most common is Esterel (for further information see [3]). This thesis however is based on Quartz (see also [12]).

Synchronous languages are based on the concepts of

- perfect synchrony, as explained in the former section
- logical time, which counts only the number of interactions

These concepts are represented by so called macro- and microsteps (see also [10]).

1.3.1 Macro-/Microsteps

A macrostep is the representation of an interaction, thus it contains the whole process of evaluating all outputs. Thereby all outputs are calculated theoretically in parallel. It is assumed, that during such a macrostep, all variables, either they are inputs, outputs or local have the same value all the time.

The evaluation thereby is done in microsteps, which are basic instructions like an addition for example. These microsteps do not consume time in this model and are all calculated at the same moment.

1.3.2 Averest Interchange Format

This thesis sets up upon AIF, a file format at an intermediate level. AIF documents are created by the Quartz compiler Ruby. This chapter gives a short overview about the AIF format.

AIF is based on XML, hence it is suitable for all operating systems and readable by the most programing languages. Every AIF document starts with an header, followed by a root element `<aif>...</aif>`, which enclose an AIF-module. The header only provides information about the used XML-version.

Every module has a name and consists of several parts:

- declarations
- control signals
- definitions

- data flow
- control flow
- tasks

The declaration part consist of a list, in which every variable, that is used in this module, is described and assigned to an identifier. The list provides information about the variable type, which can be boolean, bitvector, natural or integer, the storage type, which can either be memorized or event, the declaration, which can be output, (controlled) input, local, or label. Labels are boolean values and indicate whether the controll flow rest at the labeled position or not.

In next part, several control signals are declared. But the most of them are not considered in this thesis, and not mentioned here. The two considered are:

- *goAlpha* : Condition to activate the module, here only marked in the first macro step.
- *goEta* : Condition to enter the module

The next part is a list of definitions. It contains key-value pairs, variables with the belonging evaluation instruction. Definitions can be used as abbreviations in the rest of the module.

The next section provides information about the data flow. It is split in two sections, the surface and the depth, but they are handled the same way by Obsidian. The data flow consist of a list of guarded action of the form

$$\gamma, \delta := \tau$$

where γ is a boolean expression, δ is a basic expression and τ a expression of the same type. δ has to be local or output.

The control flow is a list of unguarded actions of the form

$$\delta := \tau$$

where δ is a basic expression and τ a expression of the same type. δ has to be a declared control-flow location (label). Here also a difference between surface and depth is made, which does not matter for this thesis.

Tasks are not considered in this thesis.

For further information see also [4].

1.3.3 Causality

Within a macrostep, variables have exactly one value. Since all microsteps within a macrostep are considered to be executed instantaneously and at the same time, a microstep can read a variable of which the value is determined later in the program. For example, in the following program fragment,

```
pause;
x = y + 5;
y = z * 3;
pause;
```

`x` is assigned a value that depends on the value of `y`, `y` is assigned in a later microstep; these assignments can be reordered, so that sequential execution delivers the correct results. However, this technique does not work if there are causality cycles between the microsteps:

```
pause;
x = a ? y + 1 : 0;
y = !a ? x - 1 : 0;
pause;
```

In this case, the assignments can not be statically (i.e., at compile-time) reordered to a correct execution sequence; hence, this is called a "static causality cycle". However, they can be reordered dynamically (at run-time), given the value of `a`: if `a` is 1, `y` has to be assigned first, otherwise `x`. If reordering is not possible at all, a "dynamic causality cycle" exists:

```
pause;
x = a ? y + 1 : 0;
y = a ? x - 1 : 0;
pause;
```

1.4 Data Flow Processors

Data flow computers do not execute sequential programs, but they can execute data flow graphs. A data flow graph is a directed graph, where the nodes contain the operations and the edges carry tokens, which represent values, an example for such a graph is shown in figure 1.3. When enough incoming edges carry a token, then the node can execute his operation. This is called it can fire. When a node fires, it consumes tokens on the incoming edges and produces new tokens on the outgoing ones. Thereby a successive node can be enabled to fire himself (see also [14]).

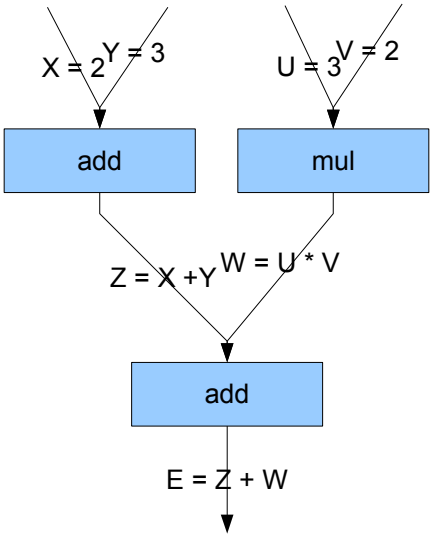


Figure 1.3: Example of a data flow graph

The problem for a data flow computer is now to find a schedule for a given data flow graph. Theoretical after a node has fired, all other nodes have to be checked either they are ready or not. Of course this is impossible. Therefore tokens and instructions are separated and stored in their own queues. Now the difficulty for the processor is to find matching tokens and instructions. This is called the token matching problem. Solutions for this can be found for example in [9] or [1].

Data flow computers are efficient when executing several threads, but have poor performance when executing a single thread. Thus, they are well suited for reactive systems, because here, in general, exists a lot of concurrency.

1.4.1 Demand Driven

Demand driven means, that an instruction is only executed, if its result is requested by an other instruction that is scheduled to execution (see also [13]). This behavior can be well suited through recursive evaluation. As a little example consider the following instructions:

```
inst 1: out := x + y;
inst 2: x   := 2 + 3;
inst 3: z   := 3 * 4;
```

```
inst 4: y   := 2 * 3;
```

Here `inst 1` has to executed because it defines an output. The processor detects that he needs x and y, and thus `inst 2` and `inst 4` are scheduled for execution. Then `inst 2` and `inst 4` can be executed, because their arguments are constant. After that, `inst 1` can be executed. `inst 3` is simply ignored. A common sequential processor would have executed all four instructions.

1.5 Terms

The following terms will be used in various parts of this thesis and are defined here.

- Compiler: the compiler presented in this thesis ("Obsidian") which transforms AIF-modules to Sawmill-programs.
- Sawmill-program: a binary which is created by Obsidian and which can be executed by Sawmill or Jasper.
- SL-compiler: compiler that generates guarded actions from synchronous programs; `Ruby` in the Averest Framework.

Chapter 2

Execution

Execution of Sawmill-programs is carried out macrostep-oriented; these correspond directly to the macrosteps in the source program. The length of a macrostep, that is, the time required to evaluate all expression trees that have to be evaluated, is not constant. The concept of microsteps is not relevant, since suspension or abortion of statements is handled by the SL-compiler, and static dependency cycles (that can occur between different microsteps within the same macrostep) are handled by Obsidian.

2.1 Expression Trees

Central to the execution of Sawmill-programs are expression trees. These are directed, ordered trees, with the nodes being operators, and the arcs representing data dependencies, i.e. every operator has arcs pointing to those nodes that serve as its arguments. Expression trees are ordered, since for some operators, the order of their arguments is relevant (e.g., for division or selection, see section A). Operators do not have side effects (except for some reading and all writing operators). The term "operator" has been chosen deliberately to show the contrast to instructions in conventional architectures; moreover, they are not "executed" but "evaluated".

The leafs of an expression tree are operators without non-constant arguments, for example operators that return a constant value, or operators that return the value of a register (the index of the register is a constant; see section 2.2).

Some operators do not always need all their arguments to compute their results. For example, the second argument to the "or"-operator does not have to be evaluated if its first argument is true - the result will be true in every case. This demand-driven evaluation of expression trees allows more

efficient computation, since only those subexpressions are evaluated, of which the results are needed. Furthermore, it is crucial for the handling of static dependency cycles (see section 3.4).

2.2 Registers

Registers are storage cells that can be read and/or written by an expression tree. They should not be confused with the registers in conventional, imperative architectures: they do not store values between operators, but rather between expression trees or between macrosteps.

Sawmill-programs consist of five sets of registers:

- input registers
- delayed registers
- eager immediate registers
- lazy immediate registers
- output registers

Each register has an associated expression tree to evaluate its value for the current macrostep. Input registers are an exception to this, their values are provided by an external source. Every type of register comes in two flavors: bit registers and word registers. Bit registers are not treated differently from word registers during evaluation; however, they take up much less space in memory. In synchronous programs, especially control flow locations are boolean variables, so a considerable effect should be expected. Words are 16 bits long in our implementation; this length is not dictated by our architecture.

Every delayed register reserves two memory locations within the processor: one that can only be read, and one which receives the result of its expression tree. After every macrostep, these two locations are swapped, so that in the next macro step the processor will read the value that has been written in this step. Their name stems from their purpose of implementing variables with delayed assignments.

Immediate registers reserve special memory locations: in addition to their value, they are also associated with two flags, one to indicate whether their value for the current macrostep has already been determined (`valid`-flag), and one to indicate whether their expression tree is currently being evaluated by the processor (`busy`-flag). If an expression tree requests (i.e. reads) the

value of an immediate register, and neither the `busy`- nor the `valid`-flag is set, its expression tree is scheduled for evaluation (see section A.1 for further information). If the `busy`-flag is set, evaluation of the expression tree that requested the register is stalled until the flag is cleared (i.e., the value is available); however, the processor can evaluate another expression tree if it has to wait for the flag to be cleared. At the beginning of a macrostep, both flags are cleared for all registers. Two types of immediate registers exist:

- eager immediate registers are guarantueed to be evaluated in every macro step, even if their value is not requested by another expression tree; they keep their value across macro step boundaries.
- lazy immediate registers are evaluated only if their value is requested by another expression tree; their value is discarded after the current macro step.

These registers are termed "immediate" because they are not suited to represent variables with delayed assignments (there are exceptions, see chapter 3 for details).

The value of an output register is sent to an external sink; it can not be read from an expression tree.

The order in which evaluation of registers is initiated does not matter, however it is recommended that the output registers are evaluated first. Although the time required to for the current macro step will not change, in general the latency of the processor's reaction to an input will be decreased.

2.3 Evaluation

Evaluation of expression trees is demand-driven, hence it starts at the root of the tree. The exact proceeding can be abstracted to demand-driven dataflow semantics. When an operator is evaluated, it requests that the operators that serve as its arguments are scheduled for evaluation; this is done recursively, until a leaf operator is evaluated.

When all arguments of an operator are available, it computes the result and sends it to the operator that requested it (the operator is said to "fire"). Operators that under certain circumstances can fire even though not all of their arguments are available (like the "or"-operator) request only the value of their first argument. When this value is available, the operator decides whether the value of the second argument is required, and requests it if this is the case.

Evaluating an expression tree this way, i.e. using a fully-fledged dataflow processor, is, however, highly inefficient for the following reasons:

- When an expression tree is serialized in preorder, the processor can evaluate it using a stack-based approach, as will be explained later in this chapter. This way, the destination for a value token that is produced by an operator is implicitly given; furthermore, the source of an argument also does not need to be explicitly encoded in the serialized expression tree. An operator in our architecture is only one byte long (with some exceptions, like register-reading operators), while still offering the flexibility of extending the operator set.

  The nodes in dataflow processors, in contrast, need to encode the address of the source and/or the destination nodes, which leads to longer instruction words. For example, the Monsoon dataflow computer ([7]) has an instruction-length of 32 bits. Operators in expression trees exhibit very bad temporal locality (since they are evaluated at most once per macrostep), leading to frequent cache misses. Considering that DDR memory modules deliver at most 128 bits per cycle, evaluation would be limited to about four operators per cycle, thus the potential parallelism inherent in synchronous programs would be wasted.

- After an operator fires, there is always at most one other operator that can compute its own result. Determining this operator is trivial (it is always the parent of the operator that fires), as is the decision whether it actually has all arguments available that are needed to compute the result. This way, there is no token matching necessary.

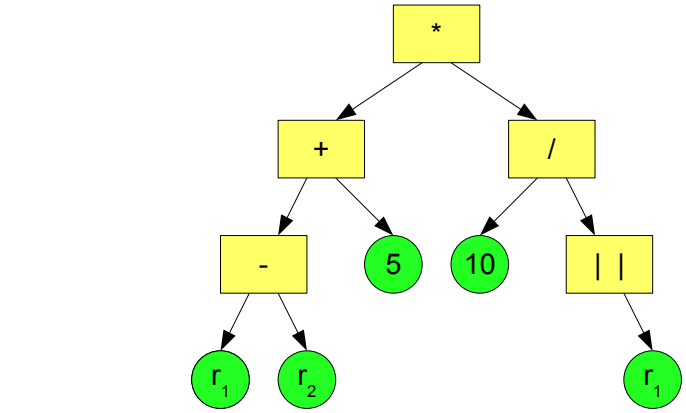
- Since every operator that fires produces exactly one token (and every operator fires at most once per macrostep, because there are no loops in an expression tree), no race condition between tokens on an arc can occur. Therefore, techniques like queues, acknowledge edges or frame tags are not needed.

2.3.1 Tree Serialization

The way that the operators of an expression trees are serialized by the compiler is a key prerequisite to the simplicity of evaluation. As mentioned above, the trees are traversed in preorder: the opcode of the operator is written first, then the serialized first argument, then the serialized second argument. For example, the expression tree for

$$((r_1 - r_2) + 5) * (10/|r_1|)$$

is shown in figure 2.1(a), its serialization in figure 2.1(b). Operators that request evaluation of some of their arguments based on a condition (like,



(a) Tree

(b) Serialization

Figure 2.1: Example expression

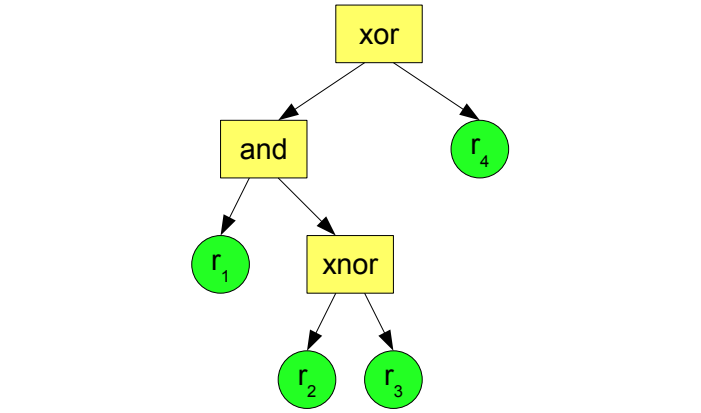
for example, short-circuit boolean operators) need to know the length of the optional argument's expression tree in order to determine the next operator. This is accomplished by encoding its length (in bytes) in the stream right before the tree itself. Figure 2.2(a) shows the tree of the expression

$$(r_1 \wedge (r_2 = r_3)) \oplus r_4$$

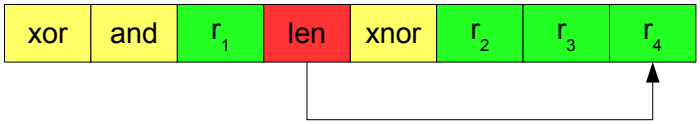
Its serialization is shown in figure 2.2(b). The and-operator requests the evaluation of its second argument only if the first one evaluates to 1. The length of the second argument's expression tree is contained in the *len*-field after the first argument's tree; it implicitely points to the next operator that the processor has to read in case the and-operator short-circuits (in this example this is the second argument to the parent of the and-operator).

2.3.2 Tree Traversion

While the operators in the stream are being read, the processor builds two stacks: an operator-stack that stores the operators that have been read but not yet evaluated, and a data-stack that stores the results of an operator's first argument; the latter has a valid-flag associated with every entry in the stack to indicate whether the first argument has already been evaluated (1) or not (0). The operator-stack thus always represents the current branch



(a) Tree



(b) Serialization

Figure 2.2: Example expression containing short-circuit operators

of the expression tree that is traversed. Figure 2.3 shows the step-by-step evaluation of the example expression tree from figure 2.1.

1. A multiply-operator is read from the stream and pushed on the operator-stack.

2. An add-operator is read and pushed on the stack. A data cell, which will receive the result of the add-operator, is pushed on the data-stack.

3. A subtract-operator and a data cell are pushed on the stack.

4. A read-operator (for register r_1) and a data cell are pushed on the stack. The read-operator is evaluated and its result (val_1 will be written to the top of the data-stack; since the operator at the top of the operator-stack completes in this step, it is popped.

5. A read-operator (for register r_2) is pushed on the stack. It is evaluated and its result (val_2 will be written to a temporary storage cell; since the operator at the top of the operator-stack completes in this step, it is popped.

6. The top of the operator-stack can compute its result, therefore, no operator is read; instead, the top of each stack is popped. The result

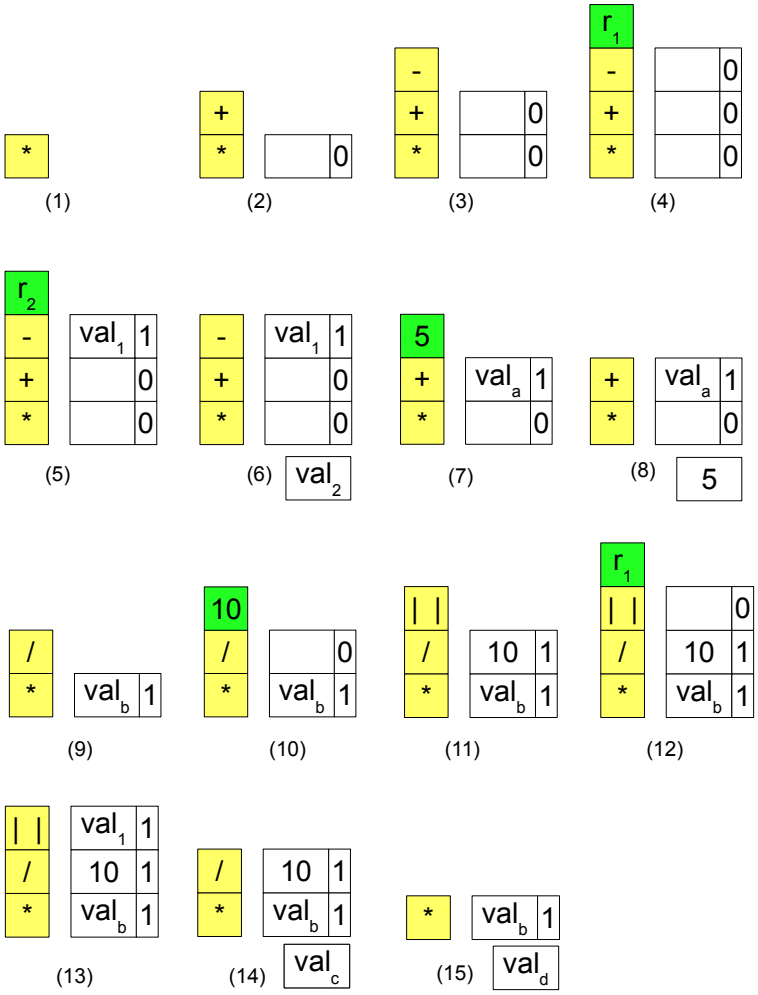


Figure 2.3: Processing an example expression. The colored stack is the operator-stack, the other one is the data-stack.

of the subtract-operator ($val_1 - val_2 = val_a$) will be written to the new top of the data-stack.

7. A constant-operator (for the constant 5) is pushed on the stack. It is evaluated and its result will be written to the temporary storage cell; the operator completes in this step and is popped.

8. The add-operator at the top of the stack can compute its value, so no operator is read and the top of each stack is popped. Its result ($val_a - 5 = val_b$) will be written to the new top of the data-stack.

9. A divide-operator is read and pushed on the stack.

10. A constant-operator (for the constant 10) is pushed on the stack, as well as a new data cell. The operator is evaluated and its result will be written to the new top of the data-stack. The operator completes, therefore the operator-stack will be popped.

11. An absolute-operator is read and pushed on the stack.

12. A read-operator and a data cell are pushed on the respective stack. The operator is evaluated and its result (val_1) will be written to the top of the data-stack. It completes in this step and is therefore popped from the stack.

13. The top of the operator-stack can compute its value, therefore both stacks are popped. The result ($val_c = |val_1|$) is written to the temporary storage cell.

14. The top of the operator-stack can compute its value, therefore both stacks are popped. The result ($val_d = 10/val_c$) is written to the temporary storage cell.

15. The top of the operator-stack can compute its value, therefore both stacks are popped. The result ($val_b * val_d$) is the result of the expression tree.

Figure 2.4 shows the step-by-step evaluation of a short-circuiting operator. The expression tree from figure 2.2, under the assumption that $r_1 = 0$, is used for an example.

1. An xor-operator is read from the stream and pushed on the operator-stack.

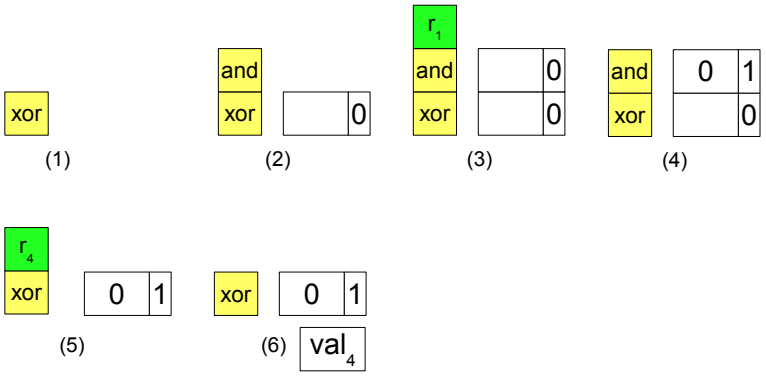


Figure 2.4: Processing an example expression. The and-operator short-circuits, hence its second argument is not evaluated.

2. An and-operator is read and pushed on the stack. A data cell, which will receive the result of the and-operator, is pushed on the data-stack.

3. A read-operator (for register r_1) and a data cell are pushed on the stack. The read-operator is evaluated and its result (0 will be written to the top of the data-stack; since the operator at the top of the operator-stack completes in this step, it is popped.

4. The and-operator can already compute its result (because its first argument is 0); hence the top of each stack is popped. Since the and-operator short-circuits, the processor reads the length of the seconds argument's expression tree and skips to the next operator that has to be read.

5. A read-operator (for register r_4) is read and pushed on the stack. It is evaluated and its result (val_4 will be written to a temporary storage cell; since the operator at the top of the operator-stack completes in this step, it is popped.

6. The xor-operator can compute its value, therefore both stacks are popped. The result ($0 \oplus val_4$) is the result of the expression tree.

Chapter 3

Compilation

The compiler component, called Obsidian, reads an AIF 2.0 document and creates a sawmill program, which can be run by sawmill or jasper. In this section the compilation process is described.

Obsidian is written in Java and can be started with one or several AIF documents, but will compile one after an other. First the document is read into an AIF module.

3.1 AIF module

An AIF module represents a document in AIF 2.0 format. The AIF module parses the whole document, which is described in section 1.3.2, and saves input and local variables, including labels, and output variables. The parsing process is shown in the following sections.

3.1.1 Declarations

In the first step the declarations are handled. The AIF module reads and saves the name and the storage type, which can be memorized or event. In case of input variables the storage type is ignored. Also the the size in bit of the variable is calculated and stored. This is not important now, but could be used for further optimization, as explained in chapter 6.

Labels are transformed into memorized bit vector variables of length 1.

Also the controll signal are read, whereas output signals are ignored. The input signals, goAlpha and goEta, are stored and will be transformed, so that they are set to true during the first macrostep. The other input signals are ignored.

3.1.2 Definitions

The definitions are also parsed. Their names are saved together with their expression trees in the AIF module, the parsing process of the trees is explained in 3.1.4. Also the result type is determined here by the expression tree and saved as well.

3.1.3 Actions

The next step is to parse the actions. For every variable the actions of surface and depth are read. The compiler makes no differences between them. But immediate and delayed actions are treated separately. For every action is checked whether there is a guard or not. If there is none, a pseudo guard (opReadConst(1)) is inserted, this means that this action will be executed always, because the guard is alway one respectively true. If the action has a guard, then the expression tree that represents the guard is imported with the function explained in the next section.

For every variable two list are created now, one with the belonging immediate actions, and one with the delayed actions.

3.1.4 Importing the trees

The import of the trees from the AIF file is meanly done by the function determinOp. Therefor the complete XML structure of the AIF module and the current element is assigned. The function works recursively. First the type of the operation is determined, which can either be natural, integer or boolean. Next the number of input arguments is detected. Now determinOp is called with the child elements and so on. Also the length of the result in bits is calculated for further optimization, as described in chapter 6.

3.2 Compiling an AIF module

The second important class in the compilation progress is the class "program". An object of this class represents a sawmill program and provides the functions to create such a program out of an AIF module.

The sawmill program is represented by the following structures:

- A mapping between variables and their initial values
- A mapping between the variables and the belonging immediate actions
- A mapping between the variables and the belonging delayed actions

- A list of all registers, including input, output, eager immediate, lazy immediate and delayed registers
- And a mapping between the variables and the registers

The compilation process is divided in the following described parts.

3.2.1 Extract Initial Values

First the initial values for every value are determined. Therefor, all actions are scanned for guards with goAlpha. Here has to mentioned, that goAlpha and goEta are true in the first macro step and false in all of the following. These controll signals are thus no input registers and are will not be checked during the execution.

If the belonging action consists only of an constant, this constant is inserted into the map for variables / initials values and the guard, as well as the action are deleted. An example from the program Squares.aif as listed in appendix C:

```
<actions><id name="i"/>
...
<guard><blVal><id name="_goAlpha"/></blVal></guard>
                                    <natConst val="0u"/>
...
```

Thus the intial value for the variable `i` is `0`.

3.2.2 Contract Actions

This function searches for identical assignments to the same variable, but with different guards. An example from Squares.aif:

```
<id name="s"/>
  <delayed>
    <guard><blVal><id name="_goAlpha"/></blVal></guard>
                  <natVal><id name="_var3"/></natVal>
  </delayed>
...
<id name="s"/>
   <delayed>
    <guard><blVal><id name="ell1"/></blVal></guard>
                  <natVal><id name="_var3"/></natVal>
   </delayed>
```

Here the variable s gets two times the same assignments, only the guards differ.

These assignments are combined in the following form:

```
( guard1 || guard2 ) s = expr
```

This function is not essential for the compilation process, and the resulting program would work without, but it provides a little speed up.

3.2.3 Create Trees

This function creates the expressions trees for every local and output variable, input variables, of course, do not have an expression tree. All guarded actions are combined under an select expression (see section A.3), which is the root element of the tree. Immediate and delayed actions are considered seperatly and two tree for each variable are crated and saved.

3.2.4 Parentize Trees

At this moment, every element of the tree, due to the mechanism of determineOp, explained in 3.1.4, knows his children. Unfortunately none of the elements knows his parent, but some functions, for example propagateDefs in the next section, require this. Therefor the function parentize trees determines recursively the parent of each node. First the root node tells his children that he is their parent, and then tells them to do the same with their children and so on. The parent node is saved in every element of the tree.

3.2.5 Propagate Definitions

In the AIF module is saved which elements reference to a certain definition or variable. Now every definition that is referenced only once is inserted at the point where the reference take place. This function need to know the parents of the nodes, because the parent node has to informed, that his child has changed(see also figure 3.1).

3.3 Mapping Variables

The variables in a Quartz program, along with their guarded actions, have to be mapped to registers and expression trees in Sawmill-programs.

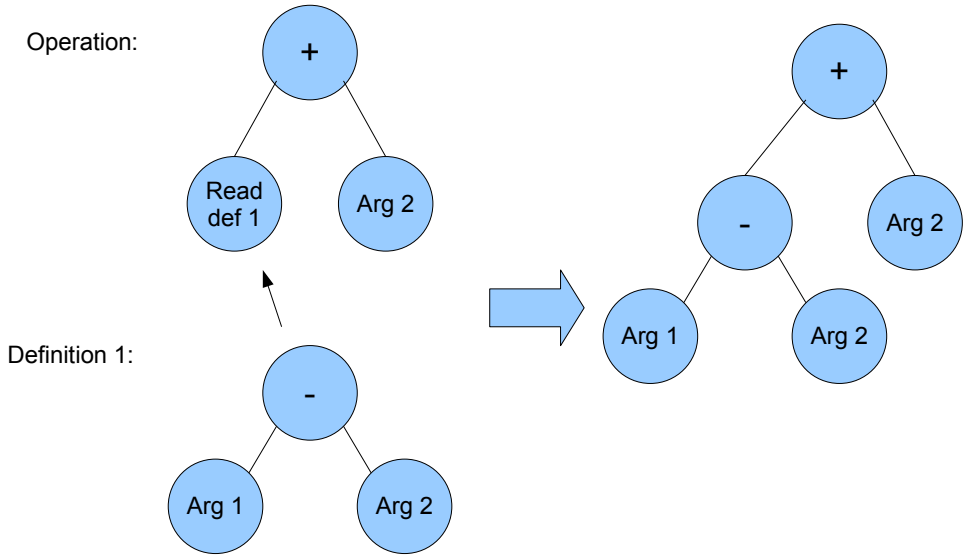


Figure 3.1: Obisidian: propagation of definitions

Note that expressions for immediate assignments of variables (e.g., `x=expr`) can not read this variable, since this would imply that the value of the variable is used before it is known, which results in a causality cycle.

All immediate respectively delayed guarded actions of a variable can be combined to a single statement. For example, the guarded actions
$\gamma_1 : x = \tau_1$
$\gamma_2 : x = \tau_2$
can be combined to the statement

```
if( gamma1 || gamma2 )
  x = gamma1 ? tau1 : tau2;
```

which has the general form

```
if(cond)
  x = expr;
```

If x had an unconditional action (other than its default value for the macrostep), its statement would have the form

```
x = expr;
```

Depending on the form of its statement, every variable is mapped to a register with a certain type and a certain expression tree.

Every local variable (control flow locations are treated as local event variables) in a Quartz program belongs to exactly one of the following categories:

- variable x, `x=expr;`
  The value does not need to be remembered across macrostep boundaries, therefore x is mapped to a lazy immediate register r, with `r=expr`

- event variable x, `if(cond) x=expr;`
  Mapped to a lazy immediate register r, `r=cond ? expr : default(x)`

- memorized variable x, `if(cond) x=expr;`
  Mapped to an eager immediate register r, `r=cond ? expr : r`

- variable x, `next(x)=expr;`
  Mapped to a delayed register r, `r=expr`

- event variable x, `if(cond) next(x)=expr;`
  Mapped to a delayed register r, `r=cond ? : expr : default(x)`

- memorized variable x, `if(cond) next(x)=expr;`
  Mapped to a delayed register r, `r=cond ? expr : r`

- event variable x, `if(cond1) x=expr1; if(cond2) next(x)=expr2;`
  Mapped to a lazy immediate register r1 and a delayed register r2; reads of this variable are mapped to r1.
  `r1=cond1 ? expr1 : r2,`
  `r2=cond2 ? expr2 : default(x)`

- memorized variable x, `if(cond1) x=expr1; if(cond2) next(x)=expr2;`
  Mapped to a lazy immediate register r1 and a delayed register r2; reads of this variable are mapped to r1.
  `r1=cond1 ? expr1 : r2,`
  `r2=cond2 ? expr2 : r1`

Note that reads of a delayed register always return the delayed value from the previous macro step; moreover, they do not trigger an evaluation of the variable's expression tree.

Input variables and controlled input variables are mapped to an input register. Output variables are replaced by a local variable, which is mapped according to the rules described above; this is done because output registers do not keep their value across macrosteps. An output register is added which simply reads the value of the corresponding register to which the local

variable is mapped. In case the output variable does not keep its value across macrosteps (either because it is an event variable without delayed assignments, or because it is unconditionally assigned an immediate value in every macrostep), the expression tree is directly associated with the output register.

3.4 Resolving Static Cycles

Synchronous programs can exhibit static causality cycles between different microsteps of the same macrostep, for example in the following code fragment (a, b, c and d are supposed to be input variables and thus do not depend on x or y):

```
x = cond1 ? y + 1 : a + b;
y = cond2 ? x - 1 : c + d;
```

Statically, the value of x depends on the value of y, and vice versa. However, a dynamic cycle exists if $cond1 \wedge cond2$ holds for the current macrostep. Jasper does not handle dynamic causality cycles, it is the duty of the SL-compiler to do this; therefore, it does not try to find, let alone resolve, dynamic cycles.

Static cycles (that are not dynamic cycles) on the other hand, can be resolved by (1), evaluating only the guarded action that is enabled for the current microstep and (2), evaluating only those subexpressions within selective expressions, of which the result will actually be used (in the above example, this would by `a + b` if cond1 is false, or `x - 1` if cond2 is true). This strategy works iff the guards respectively the conditions in a selective expression are not part of the static cycle.

Both requirements are automatically fulfilled:

1. the guarded actions for a variable are combined to a single expression tree using a select-operator, and

2. selective expressions are implemented using the select-operator, too.

The select-operator has the guaranteed property of evaluating only the action which is enabled (i.e., the one of which the corresponding guard evaluates to 1).

3.5 Completing the Compilation Process

Now the compilation process is finished. The byte code for Jasper and Sawmill can be created by replacing the operations by their opcode, which

you can find in appendix A and generating a binary. The form of the binary is described in appendix B and C.2.

Chapter 4

Hardware Architecture

The processor we designed is called "Sawmill", its name is inspired by its characteristic trait of processing (expression) trees. In this chapter, its microarchitecture is described; due to lack of time, we were not able to implement it on an FPGA.

Figure 4.1 gives an overview of the main components of Sawmill.

- Tree Processing Units (TPU) evaluate register trees.
- The operator server provides the operator streams for the TPUs.
- The register server schedules registers for evaluation by a TPU and handles "read register"-requests.
- The program loader is responsible for loading a program from external memory.

The processor does not provide a way to directly communicate with an external system, it can read inputs and write outputs from/to RAM only. The reason for this decision is that the only interface that is implementable without much effort is a serial port, which by far can't handle the amount of traffic that is expected to be generated. For example, an RS-232 port provides a maximum speed of 115,000 bits/second ([2]). If our processor runs at 150 MHz and a program has one input bit and one output bit, this would mean that a macrostep is 1300 cycles long (under the assumption that the port is full-duplex and the input of the next macrostep is available before the output of the current one has been computed), so the processing speed would be severely limited by IO. In a real application, the communication would be handled by other means; for example, a reactive system in a car could interface a CAN-bus ([5]).

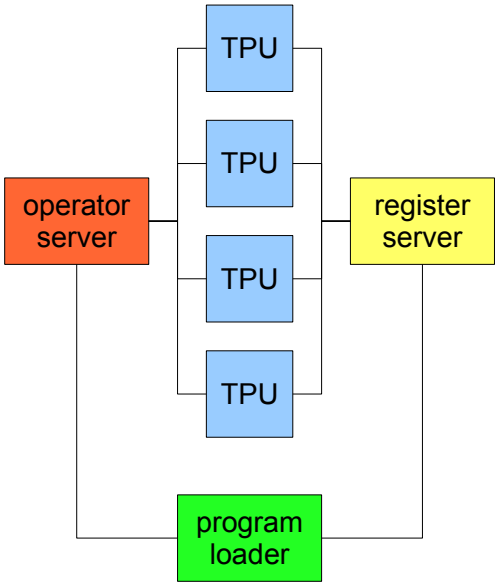


Figure 4.1: Overview of Sawmill

4.1 TPU

The processor contains four TPUs; the number of TPUs is, however, in theory only limited by chip area, and in practice by the number of operators that the operator server can provide per cycle, and, to a lesser extent, the number of registers that the register server can read, write and schedule per cycle.

A TPU (see figure 4.2) consists of

- a small buffer which holds up to 8 bytes. Whenever it contains 4 bytes or less, the controller requests 4 more bytes from the operator server. A possible future optimization would be to associate those requests with a priority (depending on the state of the buffer), to prevent stalling other TPUs if they need the next bytes more urgently.

- an on-chip RAM for the combined operator-/data-stack (and a stack pointer SP to indicate the top of the stack), as well as a temporary storage cell as described in chapter 2.

- an operator unit (OpU), that determines the next action to take (e.g., push stack, pop stack, short-circuit and jump to a certain position in the operator stream, request value of a certain register, etc.) and computes the result of the operator at the top of the stack (if all needed

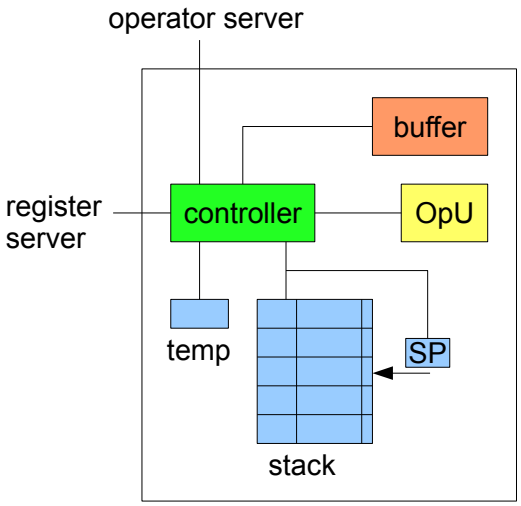


Figure 4.2: Tree Processing Unit

arguments are available). This unit is purely combinational, the state of the other components is updated by the controller.

- a controller, which coordinates the other components. It also contains an operator pointer (OP), which holds the address of the next byte to fetch from the operator stream (similar to the instruction pointer or program counter in common processors).

It takes some time to access the stack; computing the result of an operator in the same cycle as a read of one of the operands from the stack would be slow. Therefore, the controller caches the top-most and second-to-top entries of the stack.

When a TPU reads a register which is not available yet (i.e., its valid-flag is zero), it requests another register from the register server for evaluation (if possible, this will be the one that it tried to read). The TPU will simply start evaluating the expression tree of the new register on top of the stack; the index of a new register is stored together with its write-operator on the stack, this way the TPU always knows which register to write to when it finishes evaluating an expression tree. After the new register has been evaluated, the processor repeats the read that had failed.

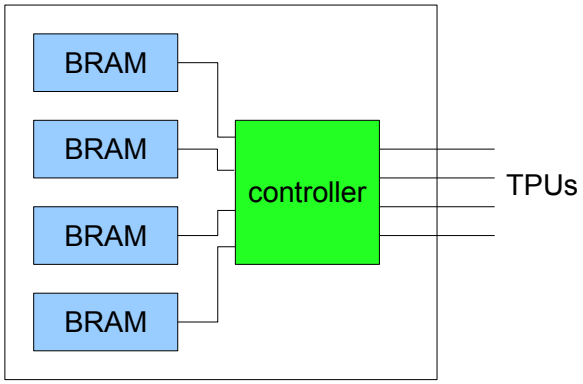


Figure 4.3: Operator Server

4.2 Operator Server

The operator server (see figure 4.3) keeps all operator streams in on-chip RAM ("BRAM"); keeping them in external memory would be too slow. Our processor has four of these BRAMs (although this number is variable) and every BRAM has two independent read-ports, so at most eight requests could be handled per cycle. It is important to interleave the data between the BRAMs, so that the code of shorter programs is spread evenly among them.

The controller handles the requests from the TPUs, including arbitration in case of a conflict, and maps addresses to locations in the corresponding BRAM.

4.3 Register Server

Requests for reading, writing or scheduling a register are handled by the register server (see figure 4.4). It contains the following components:

- the address of the expression tree for every register (except input registers). These are the addresses used by the operator server.
- the values of the registers.
- the busy-/valid-flags for each register (except input registers).
- a counter that tracks the number of registers that have been evaluated in this macrostep so far; only delayed, output and eager immediate registers are counted.

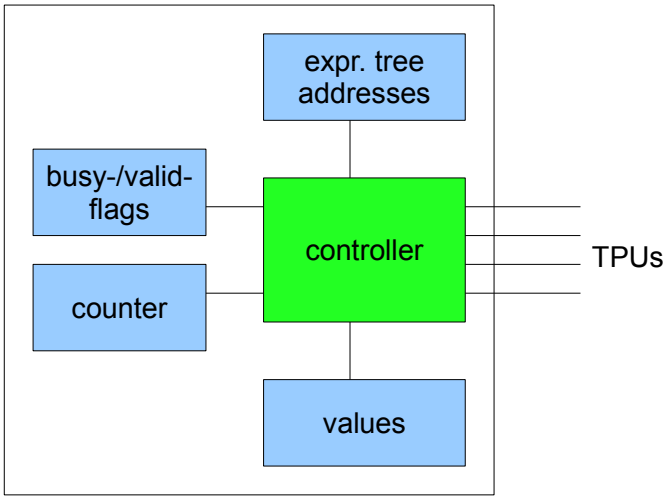


Figure 4.4: Register Server

When a TPU writes a register, its valid-flag is set, its busy-flag cleared and its value is written. Also, a counter is incremented, if applicable. The value of the counter is compared to the number of registers that have to be evaluated every macrostep; if they match, the values of the output registers are copied to an external RAM, the counters are reset, all flags are cleared, the values of each delayed register are swapped and the new values of the input registers are read from RAM.

TPUs that read a register ask the register server for its value. If the corresponding valid-flag is set, the controller simply returns the value. If the busy-flag of the register is set, the controller tells the TPU to wait (causing it to request an arbitrary register). If neither flag is set, the controller tells the TPU to evaluate the register it requested.

When a TPU is idle, it requests a register from the register server. The controller chooses a register which is not currently being evaluated (i.e., both flags are zero). To determine such a register, it maintains a small queue that contains candidates for evaluation.

4.4 Program Loader

Upon power-on, or when requested externally, the program loader reads a Sawmill-binary (see appendix B) from an external memory. It reads the initial values of the delayed and eager immediate registers and writes them to their respective locations. The expression trees are stored in the BRAMs

of the operator server; finally it calculates the addresses of the trees for each register and writes them to the register server.

Chapter 5

Simulation

This section describes the behavior of the software simulation for Sawmill, Jasper. It was not the main goal for Jasper to run an Obsidian program as fast as possible, but to simulate the Sawmill architecture as close as possible, that mean every cycle in Sawmill should be an cycle in Jasper as well. Our reason for this decision was to be able to test different speed ups for Jasper respectively Sawmill without implement them in hardware.

Jasper consist basically of multiple identical all purpose execution units, a set of registers, a controll unit and a unit, which analyzes the program.

5.1 Program Flow

After being started, Jasper reads the given input file, and loads its content to a byte stream. Afterward Jasper initiates the execution units and the register set. Then it calls an analyzing component to analyze the content of the input file. After the input file is completely read, the controll unit is activated.

5.1.1 Analyzing the Program

The analyzing component reads the in chapter 2 and appendix B defined program structure. It reads the quantity of each register type and allocate space for them in the register set. Then it reads the expression tree of every register and store it together with the register in the register set, so they are accessible from every part of Jasper.

5.2 Controll Unit

At the beginning of its work the controll units form a queue of all those registers, that have to be evaluated in every macro step, these are all delayed and output registers, as well as the eager temporary registers, as explained in chapter 2. This step is done exactly once. In this section these register will called 'all times registers'.

After that the controll unit begins with its actual work, which is controlling the execution units. This work is composed of two parts:

- dispatching the all times registers to the execution units and
- make the executions unit execute one step of their expression tree

The controll unit take one register after another out of the queued all times registers and try to assign it to an execution unit. If the actual execution unit is busy at the moment, the controll unit orders it to go ahead one step in its expression tree and try to the assign the register to next unit. If an assignment was successful the next register of the queue is taken.

A step thereby can be one of the following:

- read an operation from the tree and resolve its opcode
- read the index of a register and the value of the register
- read the first byte of a constant
- read the second byte of a constant
- execute an operation, and save the result in the instruction stack, and check the top of stack either it is ready or not

After the controll unit has checked all executions unit the cycles counter is increased by one, because in the hardware implementation the described act is done in parallel within one step.

Another task of the controll unit is to assign new all times registers to execution units, which are waiting for busy registers, or to tell them that they have to stall.

Also the controll unit change the status of the registers after every macro step. The temporary registers are marked as invalid and the delayed registers are swapped.

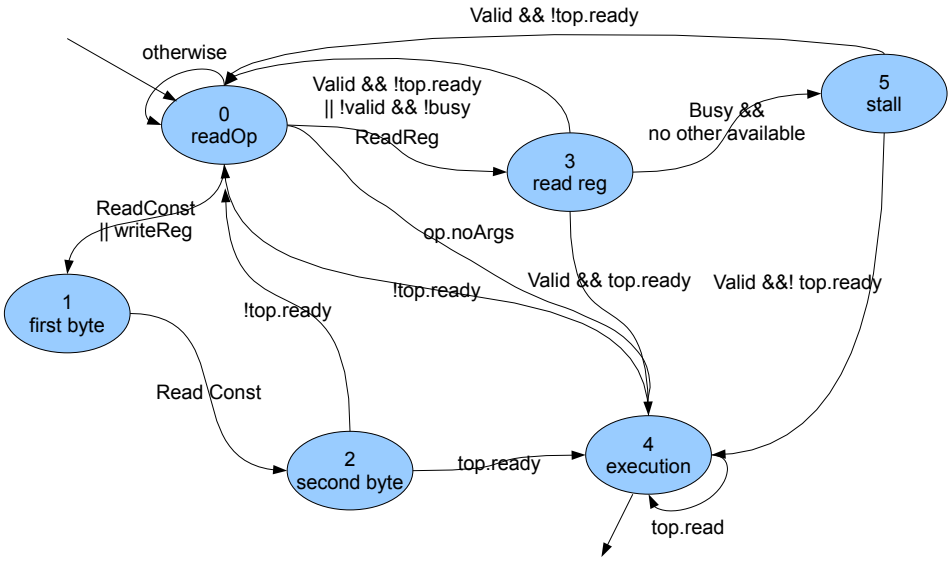


Figure 5.1: Internal state machine of a jasper execution unit

5.3 Execution Units

Jasper has several general purpose execution units. Their actual behavior depends on their internal state, the content of their dispatched expression tree, and the content of their internal instruction stack.

5.3.1 Internal state of the instruction units

The state machine that represents the internal state of an execution unit consists of 6 states, which correspond generally speaking to the actions of a microstep. Also the status changes every microstep.

State 0, is also the initial state of an execution unit. In this state a byte, which represents an opcode, is read from the tree. Then the execution unit call an external function which resolve the opcode an return a corresponding operator object. If this object is neither a read constant operator nor an operator which deals with registers, it is lay on the top of the instruction stack and marked as non-ready, in this case the following state is again 0. If it is a read constant operation or a write register the next state is 1. If it is an read register instruction the following state is 3.

A special case at this state form the operators, that have no arguments

and do not read a constant. At this moment this is only one operation, namely "first Step" (see chapter A). In this case the operation is marked as ready, placed on the stack and the next state is 4.

State 1, in this state a byte is read from the tree and saved in the actual instruction. It can be either the upper 8 bits of a 16 bit constant, if the instruction is a read constant operation, or it is the index of a register in a write register operation, which is afterward placed on the stack. In the first case the next state is 2, in the other one it is 0.

State 2. Here, the second byte of an read constant instruction is read from the tree. The byte is interpreted as the lower 8 bits of the value. The instruction is now ready an can be executed, which here simply means the constant is written to the stack. If the top instruction on the stack is now ready the next state is 4, else it is 0.

State 3, the read register state. First the index of the register, which is one byte, is read from the tree. Then the state of the register is checked. If the value of the register is already valid, which is always be the case at delayed and input registers, the value is read an stored in the stack, if the top operator of the stack is ready now, the next state is 4 else it is 0.

If the register is not valid and not busy, the actual instruction unit will evaluate the register itself. This behavior is a form of demand driven evaluation. To do this the following is done: The read instruction is put on the stack, the corresponding register is marked as busy, the actual expression tree is stored in a queue and replaced by the new expression tree, the instruction stack must not be changed. The next state is 0.

If the register is not valid but busy, which mean an other execution unit evaluates it right now, the execution unit ask the controll unit for an other expression tree to evaluate. If all other registers are already evaluated for this macro step or busy, the controll unit orders the execution unit to stall, and the next state is 5. Otherwise the unit gets a new register to work on and the same procedure like described for non-busy registers has to be done.The next state in this case in 0.

State 4, is the execution state. The top instruction is read and deleted from the stack. Then its execution function is called and the received result is saved in the stack. If the top instruction on the stack is ready by now, the next state will be 4 again, else it is 0. If the executed instruction is a write register instruction, the evaluating of the corresponding tree is finished and it is marked as valid. Either is was the one that has been assigned by

the control unit, then the execution unit has finished and signal that to the controll unit. Or it was a self taken register, than this expression tree is finished and the execution unit can return to his former task, therefor the old expression tree is loaded from the queue and the read register operator is read from the stack. The next state is 4 again, because the register to read from is now valid.

State 5. This state only waits for the former busy register to change to valid. If this happens, the value of the register is read and saved to the stack. If the top instruction gets ready by this, the next state is 4 else it is 0. If the register remains busy, also the state remains 5.

The execution unit signals its current status to the controll unit by returning an integer after every step. The status can be one of the following:

- 0: Everything fine
- 1: Finished the dispatched expression tree, awaiting a new one next step.
- 2: Waiting for a busy register, need a new expression tree to work on.

5.3.2 Instruction Stack

Every execution has its own instruction stack. Every time an operator is read out of the expression tree that can not be executed right now (and that should nearly all operator, except of: read constant, read register or the first step operator), are put on the stack. In contrast to Sawmill, every execution unit need only one stack. The instruction stack does not only hold the operators them self, but also their arguments, insofar they have already been evaluated.

5.3.3 Updating the stack

In the description of the internal status, several times is mention that a result is put on the stack. Thanks of the preeorder notation of the trees, the result of an executed instruction correspond automatically to an argument of the top instruction of the stack. So, if a result is put on the stack it is checked, how much arguments the instruction has, and if the left one is not valid, the result is stored there, the argument marked itself valid. Else the result is stored in the right argument, and this is marked valid too. Now the operator is executable.

Also the updating stack function handles the short circuit detection. Therefor every time a left argument is set, it is checked if this operation is one of those that are generally able to be shorted. If this is the case, then the next byte of the tree indicates the length of the expression tree for the right argument. If the operator is executable now, for example it is an And-operation and the left argument is 0, the length of the right tree is read and the position in the expression tree is forwarded of this value. In the following step the operator is executed. Otherwise, this byte is ignored an in the next step, the evaluation of the right tree will begin.

Furthermore this function also handles the execution of the select operator, which is described in section A.3. Select is treated as an two argument operator, where the guards represent the left argument and the actions the right one. If a guard is evaluated completely it calls the updating stack function with either true or false (respectively 1 or 0), the function discover that the top of the stack is a select operator. If the result of the tree is true, the left argument of the select operator is set to true and marked as valid. The next byte of the tree is ignored, because it represents the length of the belonging action. In the next step the evaluation of the action will begin.

If the result of the guard is false, the next byte of the tree is read and the tree pointer is increased by this value. The left argument remains invalid. And the next guard will be evaluated. Since the guard of the default action can be treated the same way as all other guards, and the default action has the same structure as the others, the case that all guards evaluate to false, must not be considered in a special manner.

After the action is completely evaluated, the updating stack function is called with the result of the action. The right argument of the select operator is set to this result and marked as valid. Then the next byte of the tree, which represents the length of the remaining select statement, is read, and the position in the expression tree is increased by this value. The select operator is now ready to be executed, which happened in the next step. Executing the select operator simply means the right argument, which represents the result of the action is returned.

5.4 Register

The register set of Jasper consist of the ten specified types of registers, as explained in chapter 2. Their individual quantity is delivered by the input program. The registers are visible from all parts of Jasper. In composition to every register the belonging expression tree is stored. In general every register type has the same behavior. Every register can be written and read,

also they have flags to indicate their status, a valid flag, which indicates that their value is valid for this macro step and a busy flag, which indicates that the register is evaluated right now.

There are some exception from this behavior. Input register can only be read during a macro step. Output registers can only be written. From the point of view of the execution units, the value of delayed registers is always valid, and they are never busy, although the controll unit can see their real status. Also the value of the input registers is always marked valid.

5.5 Operators

In Jasper all operators are implemented as described in Appendix A, with the exception, that all results of the operators are 16-bit integers. For boolean values 1 stands for true and 0 for false.

5.6 In and Outputs

5.6.1 Inputs

Normally the inputs are created by the environment, here this is of course not possible. For this reason Jasper supports different methods to handle the input variables. The desired method is chosen with a parameter at the start after the program file. The following methods are available:

- *never* : The input variables can not be entered and will be assumed as 0 respectively as false in every macro step.
- *once* : The input variables can be entered before the first macrostep. These values remain the same for every macrostep.
- *always* : The input variables can be entered before every macro step.

5.6.2 Outputs

Also two variants for the outputs are available. They are also chosen as parameter at program start:

- *verbose* : The outputs are written to the console after every macro step.
- *quiet* : The outputs are not visible.

Chapter 6

Conclusions

In this Bachelor thesis, we presented a new processor architecture for synchronous programs. This architecture is characteristic in that it evaluates expression trees in a demand-driven dataflow-oriented manner and provides the exploitation of the parallelism inherent in synchronous languages. We described the exact proceedings of the execution of programs, as well as the data structures needed for this task. To show the practical feasability of our concept, we designed a hardware implementation for an FPGA. For this design, we wrote a simulator that can be used to profile the execution of programs on our processor. In order to transform an AIF-module to a Sawmill-program, a compiler, which also provides certain optimizations, has been written. All three tools are designed to work together with the Averest framework.

Our architecture combines the advantages of microprocessors and custom chips:

- It has the potential to be more power efficient than a microprocessor, because frequency can be traded for parallelization [1]. This property is important for embedded systems which have a limited energy supply at their disposal.
- It is cheaper than custom-made chips, since its non-recurring engineering costs for hardware can be distributed due to higher production quantities.
- It can be fast, since it exploits the parallelism of synchronous programs.
- It is reprogrammable and programs can be loaded from an external memory.

[1] $P = CV^2F$, with C being the average switching capacity per cycle. A lower frequency allows lower voltage, thereby saving energy per cycle (see [8])

Future work and research has to be conducted to explore the full potential and practical usefulness of our concept, especially regarding optimizations of the hardware design and extensions to our compiler.

6.1 Compiler

Currently, our compiler provides very few optimizations. For example, the signals *goAlpha*, *goEta* are assumed to be 1 during the first macrostep, and 0 otherwise; *suspend* and *abort* are assumed to be constantly 0. This allows constant folding (typically within guards) and potential dead code elimination ([6]). Furthermore, constant propagation can result in the removal of guards: in some programs, certain control flow locations are always active after the first macrostep (this is the case with *ell*1 in appendix C), so their value can be replaced with 1, leading to further constant folding and/or propagation.

Other potential optimizations can result from using the type information that the compiler deduces for every operator from its arguments. For example, if the argument *arg* in `opSegS(`*arg*`, 4, 0)` is known to be a 5 bit signed number, the result of the operator is identical to its argument and it can therefore be replaced.

The simulator could be used to create profiling information for a program, which the compiler uses for optimizations, especially concerning short-circuit operators. For example, if the second argument to an and-operator turns out to be zero in many cases, the arguments can be swapped, resulting in shorter execution times on average.

6.2 Hardware

Apart from implementing the processor in hardware, as described in chapter 4, real-world programs have to be executed on it, to determine the bottlenecks. The register and the operator server are the two components that limit the parallelism of the processor (given sufficiently complex programs), testing would reveal how many TPUs can be supported by the current concept in practice. A way has to be found to circumvent those limitations, for example by dividing the registers among two register servers.

Due to the stack-like nature of tree processing, computation of sequential operators generally can not overlap, therefore pipelining within a single TPU is not possible. However, two (or more) TPUs could share certain execution resources using pipeline interleaving, thereby saving chip area.

6.3 Bottom-up Tree Traversion

Evaluation of expression trees as described in chapter 2 is not optimal, since two steps are needed per operator: one step to push it on the stack, and one step to compute the result (and pop the stack). Thus, during evaluation of a tree, a TPU stores every operator of the tree on its stack, an operation which itsself does not contribute to computation.

Another approach to tree traversion is more efficient: expression trees are serialized in postorder; this way, when an operator is read from the operator stream, it is guaranteed that its arguments have already been evaluated, and the result of the operator can be computed immediately. This also has the advantage that no operator-stack is needed. Every operator that is evaluated consumes the appropriate number of values from the data-stack, and pushes its result on it.

This evaluation strategy takes approximately half the number of steps as the one we implemented. The compiler does not need to be adapted, except the part that serializes the expression trees. A further advantage is that the number of arguments for a non-short-circuiting operator is not limited, for example a fused-multiply-add operator with three arguments would be possible. However, an efficient way to handle short-circuit operators has yet to be found.

Appendix A

Operators

Operators can have constant and variable arguments. The values of constant arguments are encoded in the instruction stream. Variable arguments are subtree expressions, of which the value is determined at runtime.

All opcodes are given as binary numbers; a dash ("-") indicates a don't-care-bit, processors should not rely on the value of those bits. "1st byte" respectively "2nd byte" denotes the first respectively second byte that follows the opcode in the operator stream (if applicable). The "length-byte" is the byte that preceeds the serialization of the second argument's expression tree, as described in section 2.3.1

A.1 Reading Operators

All operators in this section are leaf-operations, i.e. their operands are constants. Table A.1 lists their encodings.

Opcode	1st byte	2nd byte	Operator
000w000-	`index`	N/A	opReadInput*
000w010-	`index`	N/A	opReadDel*
000w100-	`index`	N/A	opReadLazyImm*
000w110e	`index`	N/A	opReadEagerImm*, e = `eval`
000-001-	MSB(`c`)	LSB(`c`)	opConst
000-011-	N/A	N/A	opFirstStep

Table A.1: Encodings for reading operators. "w" is 1 iff the operator reads a word-sized register.

A.1.1 Read Input Register

The operators

- `opReadInputBit(index)`
- `opReadInputWord(index)`

read the current value of the input bit register respectively the input word register with the given index.

A.1.2 Read Delayed Register

The operators

- `opReadDelBit(index)`
- `opReadDelWord(index)`

read the delayed bit register respectively the delayed word register with the given index.

A.1.3 Read Immediate Register

The operators

- `opReadLazyImmBit(index)`
- `opReadLazyImmWord(index)`
- `opReadEagerImmBit(index, eval)`
- `opReadEagerImmWord(index, eval)`

read the lazy/eager immediate bit register respectively the lazy/eager immediate word register with the given index; the *Bit-variants set the upper bits of the result to zero. If the corresponding `valid`- and `busy`-flags of the register are not set (and `eval` is true in the *Eager*-variants), the expression tree belonging to the register is scheduled for evaluation and the operation will not return until the value of the register has been computed. If `eval` is false in the *Eager*-variants, the current value of the register is returned and no expression tree is scheduled for evaluation; this should be used only to read the value of the immediate register belonging to the current expression tree.

Opcode	1st byte	Operator
001w000-	`index`	opWriteOutput*
001w010-	`index`	opWriteDel*
001w100-	`index`	opWriteLazyImm*
001w110-	`index`	opWriteEagerImm*

Table A.2: Encodings for writing operators. "w" is 1 iff the operator writes a word-sized register.

A.1.4 First Macrostep

The operator `opFirstStep()` returns 1 if the current macrostep is the first macrostep (after power-on or a reset), otherwise 0. It is used to implement the "goAlpha" and "goEta" signals, which are assumed to be 1 during the first macrostep.

A.1.5 Load Constant

The operator `opConst(c)` returns a constant word with value `c`.

A.2 Writing Operators

The operators in this section do not return a value and hence have to be the root of an expression tree. Table A.2 lists their encodings.

A.2.1 Write Delayed Register

The operators

- `opWriteDelBit(index,` arg_1`)`
- `opWriteDelWord(index,` arg_1`)`

write their argument arg_1 to the delayed bit register respectively the delayed word register with the given index.

A.2.2 Write Immediate Register

The operators

- `opWriteLazyImmBit(index,` arg_1`)`

- `opWriteLazyImmWord(index,` arg_1`)`
- `opWriteEagerImmBit(index,` arg_1`)`
- `opWriteEagerImmWord(index,` arg_1`)`

write their argument arg_1 to the lazy/eager immediate bit register respectively the lazy/eager immediate word register with the given index. The `valid`-flag of the register is set to true, its `busy`-flag is set to false.

A.2.3 Write output register

The operators

- `opWriteOutputBit(index,` arg_1`)`
- `opWriteOutputWord(index,` arg_1`)`

write their argument arg_1 to the output bit register respectively the output word register with the given index.

A.3 Select Operators

The operator
`opSelect( (`$guard_1$`,` $action_1$`), ..., (`$guard_n$`,` $action_n$`),` $default$ `)`
evaluates and returns the first argument $action_k$ for which $guard_k$ evaluates to 1 (only the least significant bit of the result is considered); if no guard is true, $default$ is evaluated and returned.

It is guaranteed that

- $guard_k$ is evaluated iff $\forall_{0 \leq i < k} lsb(guard_i) = 0$
- $action_k$ is evaluated iff $lsb(guard_k) = 1 \wedge \forall_{0 \leq i < k} lsb(guard_i) = 0$
- $default$ is evaluated iff $\forall_{0 \leq i \leq n} lsb(guard_i) = 0$

These properties are needed to handle static causality cycles.

This operator is serialized in the following way:

- opcode: 111-000- (one byte)
- $guard_1$
- length of $action_1$ (one byte)

Opcode	length-byte	Operator
010-000-	*len*	opAndSC
010-001-	*len*	opCImpSC
010-010-	*len*	opImpSC
010-011-	*len*	opOrSC

Table A.3: Encodings for boolean logical operators. *len* is the length of the second argument's expression tree.

- $action_1$
- length of rest of opSelect (one byte)
- ...
- $guard_n$
- length of $action_n$ (one byte)
- $action_n$
- length of rest of opSelect (one byte)
- opConst(1)
- length of $default$ (one byte)
- $default$
- 00000000 (one byte)

A.4 Boolean Logical Operators

All operators in this section use short-circuit evaluation, i.e. the second argument will be evaluated iff the result is not available after the first argument is known. They only use the LSB of their inputs; the upper bits of their result are set to zero. Logical `xor`, `xnor` and `not` are not available; instead, their bitvector counterparts have to be used. Table A.3 lists their encodings.

The operator `opAndSC(`arg_1`,` arg_2`)` computes the logical conjunction of its two arguments. The second one is evaluated iff the LSB of the first argument evaluates to 1.

The operator `opOrSC(`arg_1`, `arg_2`)` computes the logical disjunction of its two arguments. The second one is evaluated iff the LSB of the first argument evaluates to 0.

The operator `opImpSC(`arg_1`, `arg_2`)` computes the logical implication

$$arg_1 \rightarrow arg_2$$

. The second argument is evaluated iff the LSB of the first one evaluates to 1.

The operator `opCImpSC(`arg_1`, `arg_2`)` computes the logical implication

$$arg_1 \leftarrow arg_2$$

. The second argument is evaluated iff the LSB of the first one evaluates to 0. This operator is functionally equivalent to `opImpSC(`arg_2`, `arg_1`)` and can be used if $arg_1[0] = 1$ is deemed more probable than $arg_2[0] = 0$.

A.5 Bitvector Logical Operators

The following operators are available:

- `opNot(`arg_1`)`
- `opAnd(`arg_1`, `arg_2`)`
- `opOr(`arg_1`, `arg_2`)`
- `opImp(`arg_1`, `arg_2`)`
- `opXor(`arg_1`, `arg_2`)`
- `opXnor(`arg_1`, `arg_2`)`

The operators in this section employ an eager evaluation strategy. Although some operators wouldn't always have to compute their second argument (e.g. `opAnd` with a first argument of all 0s), these cases are considered unlikely enough to be neglected. They all operate on the full word size. Table A.4 lists their encodings.

A.6 Misc. Bitvector Operators

This section comprises miscellaneous bitvector operators. Table A.5 lists their encodings.

Opcode	Operator
011-000-	opAnd
011-010-	opImp
011-011-	opOr
011-100-	opXnor
011-101-	opXor
011-111-	opNot

Table A.4: Encodings for bitvector logical operators.

Opcode	1st byte	Operator
110-000-	`len`	opConcat
110-001-	`len`	opReverse
110s010-	`c`	opSeg*

Table A.5: Encodings for misc. bitvector operators. "s" is 1 for `opSegS`, and 0 for `opSegU`. `c = msb << 4 | lsb`.

A.6.1 Concatenation

The operator `opConcat(`arg_1`, `arg_2`, len)` concatenates the lower $16 -$ `len` bits of arg_1 and the lower `len` bits of arg_2, as shown in figure A.1. `len` is a constant, with the constraint $0 \leq$ `len` < 16.

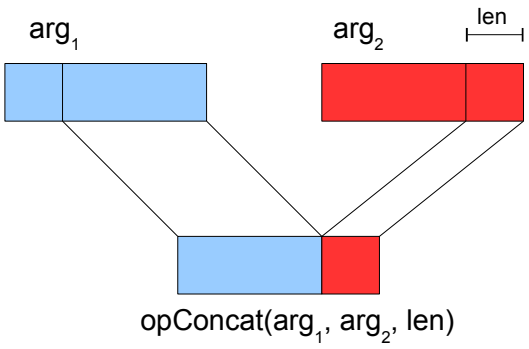


Figure A.1: Operation of opConcat

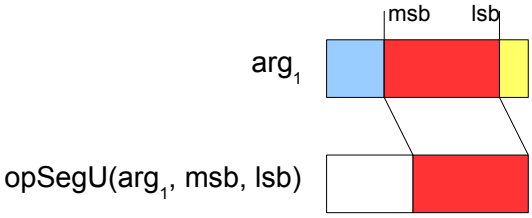


Figure A.2: Operation of opSegU

A.6.2 Bit Order Reversal

The operator `opReverse(`arg_1`, len)` reverses the order of the lower `len` bits of arg_1 and fills the upper $16 - $ `len` bits with zeros. `len` is a constant, with the constraint $0 \leq$ `len` < 16.

A.6.3 Bitvector Extraction and Zero/Sign Extension

The operators

- `opSegU(`arg_1`, lsb, msb)`
- `opSegS(`arg_1`, lsb, msb)`

extract a segment of a word and zero/sign extend it (see figure A.2). `lsb` and `msb` are constants, with the constraint $0 \leq$ `lsb` $\leq$ `msb` < 16.

A.7 Comparison Operators

All comparison operators set the LSB of their result according to the outcome of the comparison (true $\equiv 1$, false $\equiv 0$); all other bits are set to 0. Table A.6 lists these operators, table A.7 lists their encodings.

A.8 Arithmetic Operators

Arithmetic operators truncate their results if they are longer than 16 bits. Table A.8 lists their encodings.

A.8.1 Absolute Value

The operator `opAbs(`arg_1`)` interprets its argument as 16 bit signed integer and returns its absolute value. Note that the result of `opAbs(`-2^{16-1}`)` is undefined.

Operator	Compare relation
`opEQ(`arg_1`,` arg_2`)`	$arg_1 = arg_2$
`opNE(`arg_1`,` arg_2`)`	$arg_1 \neq arg_2$
`opLTS(`arg_1`,` arg_2`)`	$arg_1 < arg_2$, signed
`opLES(`arg_1`,` arg_2`)`	$arg_1 \leq arg_2$, signed
`opLTU(`arg_1`,` arg_2`)`	$arg_1 < arg_2$, unsigned
`opLEU(`arg_1`,` arg_2`)`	$arg_1 \leq arg_2$, unsigned

Table A.6: Comparison operators

Opcode	Operator
101-000-	opEQ
101-001-	opNE
1011100-	opLTS
1011101-	opLES
1010110-	opLTU
1010111-	opLEU

Table A.7: Encodings for comparison operators.

Opcode	Operator
100-000-	opAdd
100s001-	opSub
100s010-	opMul
100s011-	opDiv
100s100-	opMod
100-111-	opAbs

Table A.8: Encodings for arithmetic operators. s is 1 iff the operation is signed.

A.8.2 Addition

The operator `opAdd(`arg_1`, `arg_2`)` adds arg_1 and arg_2.

A.8.3 Subtraction

The operators

- `opSubS(`$args_1$`, `arg_2`)`
- `opSubU(`$args_1$`, `arg_2`)`

subtract arg_2 from arg_1. In accordance to the definition of the subtraction operator for natural numbers in [12], the `opSubU(`$args_1$`, `arg_2`)` returns 0 if $arg_2 > arg_1$.

A.8.4 Multiplication, Division, Modulo

The operators

- `opMulS(`arg_1`, `arg_2`)`
- `opDivS(`arg_1`, `arg_2`)`
- `opModS(`arg_1`, `arg_2`)`

perform multiplication, division and modulo operations on signed operands. Their counterparts

- `opMulU(`arg_1`, `arg_2`)`
- `opDivU(`arg_1`, `arg_2`)`
- `opModU(`arg_1`, `arg_2`)`

perform the same operations on unsigned operands. The exact semantics can be found in [12].

Appendix B

Sawmill File Format

Obsidian outputs programs in a specific file format that is suitable for being loaded by Sawmill; Jasper can read this format, too. Tables B.1 sequentially lists the fields of a binary.

number of input bit registers (1 byte)
number of input word registers (1 byte)
number of delayed bit registers (1 byte)
number of delayed word registers (1 byte)
number of eager immediate bit registers (1 byte)
number of eager immediate word registers (1 byte)
number of lazy immediate bit registers (1 byte)
number of lazy immediate word registers (1 byte)
number of output bit registers (1 byte)
number of output word registers (1 byte)
initial values of delayed bit registers
initial values of delayed word registers
initial values of eager immediate bit registers
initial values of eager immediate word registers
length of the following tree (1 byte)
expression tree for delayed bit register 0
length of the following tree (1 byte)
expression tree for delayed bit register 1
...
length of the following tree (1 byte)
expression tree for delayed word register 0
...
length of the following tree (1 byte)

expression tree for eager immediate bit register 0
...
length of the following tree (1 byte)
expression tree for eager immediate word register 0
...
length of the following tree (1 byte)
expression tree for lazy immediate bit register 0
...
length of the following tree (1 byte)
expression tree for lazy immediate word register 0
...
length of the following tree (1 byte)
expression tree for output bit register 0
...
length of the following tree (1 byte)
expression tree for output word register 0
...

Table B.1: Sawmill file format

Appendix C

An Example for Compilation and Simulation

In this chapter the result of the compilation of squares.aif with Obsidian is shown.

C.1 squares.aif

The following lists the relevant parts of squares.aif.

```
<declarations>
  <local name="i" storage="memorized"><nat size="1000u"/></local>
  <output name="s" storage="memorized"><nat size="1000u"/></output>
  <label name="ell1"/>
</declarations>

<definitions>
  <def name="_var1"><natMul><natConst val="2u" />
     <natVal><id name="i"/></natVal></natMul></def>
  <def name="_var2"><natAdd><natVal><id name="_var1"/>
     </natVal><natConst val="1u" /></natAdd></def>
  <def name="_var3"><natAdd><natVal><id name="s"/>
     </natVal><natVal><id name="_var2"/></natVal></natAdd></def>
  <def name="_var4"><natAdd><natVal><id name="i"/>
     </natVal><natConst val="1u"/></natAdd></def>
</definitions>

<controlFlow>
<surface>
```

```
  <actions><id name="ell1"/>
  <immediate>
    <guard><blFalse /></guard> <blTrue/>
  </immediate>
  <delayed>
    <guard><blVal><id name="_goEta"/></blVal></guard> <blTrue/>
  </delayed>
  </actions>
</surface>

<depth>
  <actions><id name="ell1"/>
  <delayed>
    <guard><blVal><id name="ell1"/></blVal></guard> <blTrue/>
  </delayed>
  </actions>
</depth>
</controlFlow>

<dataFlow>
<surface>
  <actions><id name="i"/>
  <immediate>
    <guard><blVal><id name="_goAlpha"/></blVal></guard>
        <natConst val="0u"/>
  </immediate>
  <delayed>
    <guard><blVal><id name="_goAlpha"/></blVal></guard>
        <natVal><id name="_var4"/></natVal>
  </delayed>
  </actions>
  <actions><id name="s"/>
  <immediate>
    <guard><blVal><id name="_goAlpha"/></blVal></guard>
        <natConst val="0u"/>
  </immediate>
  <delayed>
    <guard><blVal><id name="_goAlpha"/></blVal></guard>
       <natVal><id name="_var3"/></natVal>
  </delayed>
  </actions>
```

```
</surface>

<depth>
  <actions><id name="i"/>
  <delayed>
    <guard><blVal><id name="ell1"/></blVal></guard>
       <natVal><id name="_var4"/></natVal>
  </delayed>
  </actions>
  <actions><id name="s"/>
  <delayed>
    <guard><blVal><id name="ell1"/></blVal></guard>
       <natVal><id name="_var3"/></natVal>
  </delayed>
  </actions>
</depth>
</dataFlow>
```

C.2 squares.aif.smp

C.2.1 Header

In the header information about the register and their initial values can be found, in this case:

- 1 delayed bit register, initial value 0 (= false) represents the variable ell1
- 2 delayed word register, initial value 0 represents the variable i and the local version of s
- 1 output word register, represents the output version of s

C.2.2 Expression Trees

In this part of the file the trees, their length and the constants are saved. In the listing, values in brackets either mean a register index, or the length of the following tree where needed. In the following the expression trees of squares.aif.smp are listed.

macrostep	output	1 unit	2 units
1	0	62	32
2	1	68	34
3	4	68	34
4	9	68	34
5	16	68	34

Table C.1: Simulation of square.aif.smp with Jasper

```
Delayed Bit Tree[length: 20 Byte]:
opWriteDelBit[0]( opSelect( Or(opFirstStep,[2]opReadDelBit[0]):
    [3] opReadConst(1),[8](opReadConst(1):[2]readDelBit(0))[0]))

Delayed Word Tree[length: 30 Byte]:
opWriteDelWord[0]( opSelect( Or(opFirstStep,[2]opReadDelBit[0]):
[13] opAdd(opReadDelWord[0],opAdd(opMulu(opReadConst[2],
opReadDelWord[1]),opConst(1))),[8]opConst(1):[2]opReadDelWord[0],[0]))

Delayed Word Tree[length: 23 Byte]:
opWriteDelWord[1]( opSelect( Or(opFirstStep,[2]opReadDelBit[0]):
    [6] opAdd(opReadDelWord[1],opConst(1)),[8]opConst(1):[2]
    opReadDelWord[1],[0]))

Output Word Tree[length: 4 Byte]:
opWriteOutWord[0]( opReadDelWord[0] )
```

C.2.3 Simulation with Jasper

In table C.1 the output of the first five macrosteps and the used cycles are shown in dependency to the number of execution unit.

As you can see, the produced outputs are the expected ones. In each first step less cycles are used, this is so, because the operation opFirstStep is evaluated to one and thus the short circuit ability of opOr can be used. Also it is apparent, that the number of cycles correspond well to the number of used execution units.

Appendix D

Segmentation

Because this bachelor thesis is written by two people, here is a short listing who wrote which part. If not mentioned all subsections are included.

- Adrian Willenbücher
  - 1.3.3 Causality
  - 2 Execution
  - 3.3 Mapping Variables
  - 3.4 Resolving Static Cycles
  - 4 Hardware Architecture
  - 6.1 Compiler
  - 6.2 Hardware
  - 6.3 Bottom-up Tree Traversion
  - Appendix A Operators
  - Appendix B Sawmill File Format
- Stefan Dyckmans
  - 1.2 Reactive Systems
  - 1.3 Synchronous Languages except 1.3.3
  - 1.4 Data Flow Processors
  - 3 Compilation except 3.3 and 3.4
  - 5 Simulation
  - Appendix C An Example for Compilation and Simulation

- Together
  - 1.1 Goal
  - 6 Conclusion except 6.1, 6.2 and 6.3

Bibliography

[1] Arvind, D. Culler, R. Iannucci, V. Kathail, K. Pingali, and R. Thomas. The tagged token dataflow architecture. Technical report, Cambridge, Massachussetts, USA, 1984.

[2] J. Axelson. *Serial Port Complete: Programming and Circuits for RS-232 and RS-485 Links*. Lakeview Research LLC, 1998.

[3] G. Berry and G. Gonthier. *The ESTEREL synchronous programming language: design, semantics, implementation*. Elsevier, 1992.

[4] J. Brandt. The averest intermediate format revision 2.0. Internal report, Department of Computer Science, University of Kaiserslautern, 2007.

[5] W. Lawrenz. *Can System Engineering: From Theory to Practical Applications*. Springer, 1997.

[6] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[7] G.M. Papadopoulos and D.E. Culler. Monsoon: an explicit token store architecture. Technical Report CSG-MEMO 306, MIT Lab for Computer Science, Cambridge, Massachussetts, USA, 1990.

[8] J.M. Rabaey, A. Chandrakasan, and B. Nicolic. *Digital Integrated Circuits*. Prentice Hall, 1996.

[9] K. Schneider. Script of the lecture 'parallel computers'. 2008.

[10] K. Schneider. Script of the lecture 'system description languages'. 2008.

[11] K. Schneider. Script of the lecture 'verification of reactive systems'. 2008.

[12] K. Schneider. The synchronous programming language Quartz. Internal Report (to appear), Department of Computer Science, University of Kaiserslautern, 2008.

[13] P.S. Treleaven, D.R. Brownbridge, and R.P. Hopkins. Data-driven and demand-driven computer architecture. *ACM Computing Surveys*, 14(1):93–143, March 1982.

[14] T. Ungerer. *Datenflussrechner*. Teubner, 1993.