Modular Operational Semantic Specification of Transport Triggered Architectures

Jon Mountjoy * Pieter Hartel † Henk Corporaal ‡

September 22, 1996

Abstract

The formal specification of hardware at the instruction level is a daunting task. The complexity, size and intricacies of most instruction sets makes this task even more difficult. However, the benefits of such a specification can be quite rewarding: a precise, unambiguous description is provided for each instruction, a basis for proving the correctness of code transformations is made available, and the specification can be animated, providing a simulator. This paper proposes a high level structural operational semantic (S.O.S.) specification for the class of transport triggered architectures. These architectures are simple, powerful, flexible and modular and can exploit very fine grained parallelism. The S.O.S. is novel in that it follows the structure of the architecture, and by doing so inherits the modularity of the architecture.

1 Introduction

The precise definition of programming languages is important; ambiguities in programming language definitions were rife before the introduction of formal techniques of semantic specification. Likewise, the specification of compilers is becoming more precise, relying on the correct incremental transformation of programs written in a high level language to some low level machine language. There is also much work on verifying hardware and formal hardware description languages. However, the formal specification and verification of low level machine languages, lying somewhere between architecture and language, is lagging behind.

It is perhaps due to the size and intricacies of most processors that the formalisation is so difficult. For instance, the specification of a large portion of the MC68020 instruction set described in [2] took approximately 80 pages of text. This difficulty is unfortunate, as a manageable specification has many uses:

1. The specification can serve as a rigorous *hardware description* of the architecture. The formal meaning given to the machine code in this way eliminates ambiguities. This can then serve as documentation for VLSI designers and compiler writers.

2. Having a formal semantics helps us to prove code *transformations* or optimizations correct, or to clarify the conditions under which such transformations can be applied.

3. The semantics can act as a *design validation*, serving as a formal description of an intended architecture for the user and the architect. This is more relevant if application-specific processors are the target as different possible configurations of the architecture need to be expressed.

*Department of Computer Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
†Department of Electronics & Computer Science,University of Southampton,Highfield,Southampton SO17 1BJ,United Kingdom
‡Department of Electronic Engineering,Delft University of Technology,Mekelweg 4,2628 CD Delft,The Netherlands

4. Animating (implementing) the semantics will yield a prototype of the machine architecture and instruction set. This tool can then serve as the basis for *verification and validation* of hardware and compiler. Looked at the other way, the hardware can be used to validate the semantic model.

In general, an abstract semantic description of a low level machine architecture may be non-trivial due to the complexity of the architecture. The class of *transport triggered architectures*(TTAs) [6] were introduced as modular, parallel, application-specific processors. This extensible, VLIW-like architecture, has parallel data moves as the only operations. All that is visible at the instruction level are a set of registers whose values can be moved to other registers. The architecture shows much promise [3, 5, 6, 4], and although seemingly simple, is powerful and flexible. Since the machine itself is at a low level (data transports are exposed, pipelines are visible, instructions have to be schedulled), compilers for the architecture are by necessity more complex [7] making the precise documentation of the machine even more important.

The inherent modularity of transport triggered architectures lends itself to formal modelling not easily applied to other processors. In this paper we provide an operational semantic description for a class of TTAs. This is based on structural operational semantics [16]. This semantic framework is simple, readable, and easily implemented. It can also be used to model a fine grained parallel system, which makes it suited to the task at hand.

A requirement of the semantics is that it must be easily extensible. If we wish to prototype various configurations of architectures (which is the case since we are dealing with application-specific processors), we need the semantics to be modular with respect to the architecture: small changes in the architecture should not mean extensive changes to the semantics. With structural operational semantics, one usually builds the semantics following the structure of the syntax of a language. This paper builds the semantics of the architecture after the structure of the architecture; in effect, the semantics is a synthesis of the essential ingredients of the architecture. We believe this approach to be novel.

Another ingredient of a good semantic description is that it models at the 'correct' level. A model should not be too fine, describing circuit level when the user of the model has no use of this information. Likewise, it should not be too coarse, eliminating information that could be useful. We have sought to find a balance which will result in a model capable of fulfilling points (1–4) above. By necessity, the model is then at a much higher level than those found in the formal methods used in high-level design and verification of VLSI circuits. There are *hardware verification* languages, where one takes a gate-level description of a circuit and derives a model thereof. Some kind of model checking (usually using Boyer-Moore provers) can then be used to ensure that the model is consistent with another specification of its behaviour. There are also languages used for a formal *hardware specification* such as VHDL. These languages typically address design issues at the lower end of the spectrum (close to the hardware). The language Ruby [11] allows one to specify circuits in a high-level formal language, and 'calculate' circuits. We hope that our higher level specification can serve as a base for linking the specifications at the various levels. In addition, if used as a design validation then we would ultimately like to prove that a model based on the hardware circuits fulfills the formal specification as laid out in this paper. Our methodology thus involves analysing the architecture at the particular level of interest and extracting a language design. This language is then given a semantics. It is the aim of this paper to show the use of this methodology and the resulting semantics.

We begin by describing the basic concepts behind transport triggered architectures in Section 2. A typical TTA is given, and this serves as the basis for the semantic description provided in section 3. Section 4 shows how several variations of the architecture can be modelled, and how the semantics is animated. Section 5 examines future work, and section 6 concludes this paper.

2 Transport Triggered Architectures

This paper concerns the modelling of transport triggered architectures. Since transport triggered architectures are a relatively new concept, we first introduce the architectures. We then provide a more detailed example of a typical TTA which is used in the subsequent sections. The description will focus on the salient points of the architecture (control flow, parallel execution etc.), which are apparent in the semantic description. This section concludes with an analysis of the architecture which leads to the derivation of a syntax for the machine.

2.1 Basic Concepts

TTAs can be compared to VLIW architectures. In both cases the instructions are horizontally encoded; i.e. each instruction has a number of fields. Whereas fields for VLIWs specify RISC like operations, for TTAs they specify the required data transports. These transports may trigger operations as side effects. Programming transports adds an extra level of control to the code generator, and enables new optimizations; in particular, it allows us to eliminate many superfluous data transports to and from the register files and to reduce the on-chip connectivity. TTAs are fully introduced in [6, 9].

A compiler views a TTA as a collection of function units (FUs), register files (RFs), *move buses*, and *sockets*; see figure 1. FUs perform operations, RFs provide general purpose registers for temporary fast accessible storage, the network of move buses performs data transports between the FUs and RFs, and sockets interface FUs and RFs to move buses. Normally, each socket is connected to a different FU input/output or RF port.

To illustrate TTA programming, consider the execution of the following three instructions on a conventional machine:

```
add  r1, r2, r3  /* r1 = r2 + r3 */
sub  r4, r2, r6  /* r4 = r2 - r6 */
st   r4, r1      /* store r4 at address r1 */
```

These operations can be translated into the following two TTA instructions, given a suitable architecture with four busses:

```
r2→ add_o    || r3→ add_t    || r2→ sub_o || r6->sub_t
add_r→ st_o  || sub_r→ st_t  || nop       || nop
```

Each instruction is composed of four operations and each operation is destined for a single bus. These operations will then be executed in parallel. An increase in the number of busses increases the amount of potential parallelism. In the first instruction the four operands of the add and subtract operations are moved from general purpose registers to the inputs of the functional units which perform addition and subtraction. The add and subtract unit then, have two input sockets each. Output sockets, which will hold the result of the operation, are also available. In the second instruction the results of the add and subtract operations are moved from the FUs that performed them to the FU that performs the store operation. From this small example we already observe a few advantages of TTAs. The results of the add and subtract operations are not written back to the RF and the operands of the store operation are not read from the RF. The former saves RF write accesses and data transports, the latter saves RF read accesses. Since TTAs do not couple RF ports directly to FUs, as is the case for many VLIW and superscalar architectures with bypassing networks, the freed resources can be used for other operations. This gives TTAs a better hardware utilization, which implies less hardware for the same performance compared to conventional architectures. From the example,

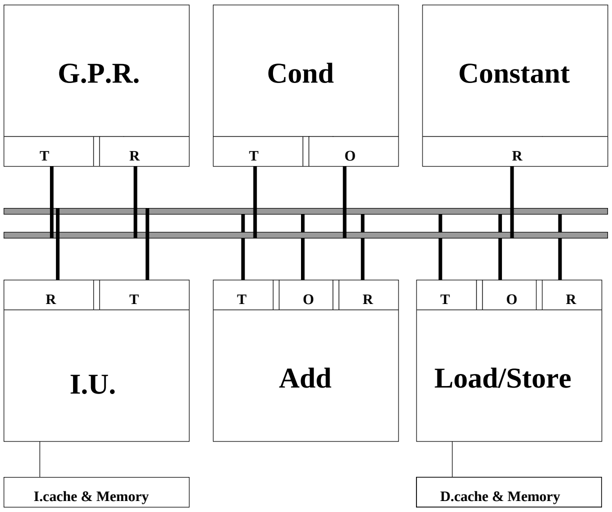

Figure 1: An Abstract Operational view of a TTA with Trigger(T), Operand(O) and Result(R) registers indicated.

we can see that to program the TTA efficiently, sufficiently many independent operations have to be found which can be executed in each cycle. If this cannot be done, the operation slots can be filled with no-ops.

The interconnection network may be fully connected, in which case every socket is connected to all move buses, or partially connected. A fully connected interconnection network simplifies the code generation task, but it is likely to result in a high bus load on the move buses, which affects the achievable cycle time. Therefore, in practice, the network will be partially connected and the compiler will be responsible for using the available connections as well as possible.

Conditional execution is provided by guarding every move with the result of a conditional unit. Control flow is provided by exposing the program counter as a register of the instruction unit. Writing to it with an address, forces a jump to this address.

2.2 A Detailed Look

Figure 1 gives an operational view of a possible transport triggered architecture. It consists of a number of functional and register units and an interconnection network. In this example, we have only provided an addition, comparison, general purpose register and load/store unit. We also assume a fully connected network of two busses. A more general architecture may have additional functional units for multiplication, division, shift and store. Instead of providing a single register, it would more typically provide a register file of several registers. Note that these would be independent and modular extensions: all that has to change is the addition of the functional unit and connections to the busses. More busses can be added, and the interconnection network need not be fully connected. The architecture presented here is sufficiently general to cover all of the important aspects in the modelling. Section 4 addresses extensions.

Registers

We will adopt the convention of labeling the input and output sockets of the functional units by the unit name and a letter, either ‘O’, ‘T’ or ‘R’. We will also refer to these as registers. An operand register (O), is used to supply a datum to a functional unit. The trigger register (T), is also used to supply a datum to the functional unit. In addition, it may set a flag enabling the functional unit to start its computation. When the functional unit produces its result, it is usually placed in a result register (R). We will always refer to registers provided by a register unit for temporary storage as general purpose registers to avoid ambiguity.

It is important to note that the only aspect of the functional unit made available to the programmer are the registers and data transports between them. The architecture is thus *transport triggered*, as opposed to the more conventional *operation triggered* architectures where operations are issued and data transports occur as possible side effects of these operations. Small constants are made available by embedding them in the operation word. A constant unit (seen operationally) then reads the instruction off the bus and makes the embedded constant available on its result register. A register unit provides a register. Writing a value to the trigger will make it available for reading from its result register in the next cycle.

Since it is not always possible to fill an instruction completely with useful operations, a NOP operation is provided.

Control Flow and Conditional Execution

Conditional execution is provided by *guarding* every data transport. The guard condition is taken as the output of the conditional functional unit. The output of a conditional unit is a boolean which is not made available as a register. Instead, it is used internally to control squash lines on the various busses. If a bus is squashed, then the data transport taking place on that bus is cancelled. Flow control is made possible by making the program counter visible as a register, (the instruction unit trigger and result registers). This program counter automatically increments every cycle (unless written to). Writing to the instruction unit trigger register (INS^T) forces a jump to the location written.

The following fragment of code demonstrates these ideas. The first instruction simultaneously moves the value 10 to the conditional units operand, and 20 to the trigger, starting the conditional unit comparison. If the result of this comparison is true, then a jump is made to location 7.

$$
\begin{array}{lcl}
10 \rightarrow \mathsf{CND}^O & \| & 20 \rightarrow \mathsf{CND}^T \\
\vdots & & \\
[Check]\ 7 \rightarrow \mathsf{INS}^T & \| & \mathsf{NOP}
\end{array}
$$

We will use the word "Check" to denote that the move is conditionally guarded on a value being true, or "Always" to denote that the move is not guarded and will always succeed. For clarity, we sometimes omit the annotation if the guard is unconditional. We will also assume that the conditional unit returns true if the operand register is less than or equal to the trigger register.

Pipelined Functional Units

A functional unit may be pipelined. We will assume a semi virtual-time latching scheme which means that pipelines run synchronously to the instruction stream; each time an instruction is issued, the FU pipelines progress a step. In addition, only moves to the trigger register start new operations. Note that in this scheme a result register can be intentionally overwritten without ever being read. A true virtual-time latching scheme would have pipelines advancing every cycle, eliminating the need for a special trigger register. The various pipelining schemes and their impact on instruction scheduling are discussed in [4]. Pipelining and its implications for the semantics is discussed in sections 4.2 and 4.3.

Locking

Transport triggered architectures may support exceptions and locking. Locking is needed for instance, if one wants an architecture with a memory which has variable access times. If a read on the store is not available, the machine can be locked until the read can proceed. Locking is further discussed in section 4.

Advantages

In summary, we list some of the advantages of transport triggered architectures:

Short cycle times The cycle time is bounded only by the register to register transfer time, since the data transports and the operations are independent. Decoding is also done locally at the individual functional units.

Exploitable parallelism TTAs can exploit parallelism at both the operation (multiple FUs) and data transport (multiple busses) level. In addition, the exploitable instruction level parallelism can be increased by either increasing the transport capacity or the number of FUs.

More scheduling freedom Dividing conventional operations into simpler data transports increases the available parallelism.

Performance Scalability Performance scales by adding multiple FUs to an existing architecture, pipelining existing FUs, or by increasing the data transport capacity. These extensions are also modular.

Simplicity TTAs have a simple and regular hardware organization. This makes them suitable for application-specific processors.

2.3 Deriving a Syntax

Our methodology of assigning a structural operational semantics to the architecture is based on extracting details from the architecture at an appropriate level of interest. Analysing the architecture, we can immediately see a hierarchy involved in the execution of an instruction: an instruction is composed of operations, operations are guarded moves, and moves are from source to target registers. This tidy separation is unfortunately more difficult to find in conventional architectures.

We will ultimately assign meanings to programs written in the machine language, and so we synthesize a syntax for this language keeping the hierarchy of the architecture in mind. A program then, is represented as a sequence of instructions, an instruction as two operations (recall that the architecture we are focussing on illustrated in figure 1 has two busses), an operation being either a NOP or a guarded move and a move by a source and target register specification. Figure 2 illustrates this.

It is a strength of the architecture that it allows this hierarchical division. Different instantiations of the architecture do not change this fundamental structure, but only the components. Thus adding a bus of FU is mirrored easily in the syntax.

Note that the load/store unit is capable of two actions (loading and storing); the physical architecture uses the opcode in the operation to distinguish between them. We have separated them at the syntax (and thus semantics) level as there is no reason to complicate the system by modelling this explicitly. The point to be noted is that they are the same physical resource, and so cannot both be used at the same time. One can prevent this by using a *static semantics*, as explained in subsection 3.5.

We are now in a position to give a formal semantics to the language.

3 Semantic Description

We now give a semantic description of the example architecture presented in section 2. The architecture as described is parallel (many moves can execute at once) and modular (it can be extended by adding functional units or busses). The execution of an instruction consists of the parallel execution of the operations of that

$$
\begin{array}{lcl}
\mathit{Operation} \ni O & ::= & \mathsf{NOP} \\
& | & G : S : T \\
\\
\mathit{Guard} \ni G & ::= & \mathit{Check} \\
& | & \mathit{Always} \\
\\
\mathit{Source} \ni S & ::= & \mathsf{CNST\text{-}c} \\
& | & \mathsf{ADD}^R \\
& | & \mathsf{INS}^R \\
& | & \mathsf{REG}^R \\
& | & \mathsf{LD}^R \\
\\
\mathit{Target} \ni T & ::= & \mathsf{ADD}^O \\
& | & \mathsf{ADD}^T \\
& | & \mathsf{INS}^T \\
& | & \mathsf{REG}^T \\
& | & \mathsf{CND}^O \\
& | & \mathsf{CND}^T \\
& | & \mathsf{ST}^O \\
& | & \mathsf{ST}^T \\
& | & \mathsf{LD}^T \\
\\
\mathit{Constants} \ni \mathsf{c} & ::= & 0|1|\ldots|255 \\
\mathit{Instruction} \ni I & ::= & (O||O) \\
\\
\mathit{Program} \ni P & ::= & (I, I, \ldots, I)
\end{array}
$$

Figure 2: Abstract Syntax of the Architectural Language

instruction. The abstract syntax of the architecture reflects these issues. After the execution of an instruction the functional units update their state (if there is information in the pipeline, or if a result register has been written to). Initially we will make certain assumptions about the architecture to simplify the description, and address the extensions required to the semantics to cope with the full architecture in a later section.

A structural operational semantics specifies transition relations on various configurations. A configuration is usually a tuple, consisting of the instruction (or part thereof) to be executed and a state. Sometimes it is just a value. The state is a representation of the state of the registers and memory locations.

3.1 Source

We begin by defining the binary transition relation which concerns configurations containing source registers:

$$\stackrel{s}{\rightarrow} \in (\mathit{Source} \times \mathit{State}) \times \mathit{Value}$$

where Source is the set of source registers taken from figure 2. That is, we define a transition relation $\stackrel{s}{\rightarrow}$ between a source register and state, and a value. This relation, together with a relation on the target register, is used to build up the meaning of an operation.

We can now begin to define the rules for the various source constructs. The meaning of a constant c is the value c of the constant. The constant unit therefore acts as the identity on the value. This is exactly what happens in the architecture, where the constant is read from the instruction word. We write:

(1) $$\frac{S = \mathsf{CNST\text{-}c}}{\langle S, s\rangle \xrightarrow{s} c}$$

This can be read as: If the source register is a CNST-c (the premise), then the meaning of the configuration $< S, s >$ is the value c (the conclusion). We always write the premise above the rule, and conclusion beneath. Section 4.1 explores an alternative approach to constants.

If we are dealing with the source register of the add unit, then we define the meaning of the result register as the value of the result register in the state.

(2) $$\frac{S = \mathsf{ADD}^R}{< S, s > \xrightarrow{s} s[\mathsf{ADD}^R]}$$

We use the notation $s[r]$ to denote the value of r in state s. The value returned above then is the value of the result register of the add unit. We will later return to defining the actual components in the state. For now, we can consider it as a mapping between registers and values.

All other source register moves are modelled in this way. For instance, we define the rule for a general purpose register as:

(3) $$\frac{S = \mathsf{REG}^R}{< S, s > \xrightarrow{s} s[\mathsf{REG}^R]}$$

and the rule for the load unit as:

(4) $$\frac{S = \mathsf{LD}^R}{< S, s > \xrightarrow{s} s[\mathsf{LD}^R]}$$

3.2 Targets

The transition relation for target operands is given by:

$$\xrightarrow{t} \in (Target \times Value \times State) \times Substitution$$

where $Target$ is the set of target registers. We will define the rules for the add unit.

We first need to define a *substitution*. Intuitively, a substitution holds a list of registers and values to which these registers are to be mapped. Later, we will see how these substitutions are used to update the state of the machine. For now, they can be thought of as reminders that the state is to be updated. We will write ϕ for the empty substitution. A more formal motivation is given in a later subsection.

The meaning of a target register and value is given by a substitution.

(5) $$\frac{T = \mathsf{ADD}^O}{< T, v, s > \xrightarrow{t} [(\mathsf{ADD}^O, v)]}$$

Thus in the above rule, the substitution indicates that the operand register should be mapped to the value v. The use of the substitution will become clear once we address the semantics for instructions.

Recall that we are dealing with a semi virtual-time architecture in which pipelines only progress synchronously with virtual time, and if the unit has been 'triggered'. Moving to an operand register results in a single substitution. Moving to a trigger register results in the additional setting of a flag, which is meant to indicate that the unit has been triggered. A similar mechanism occurs in the hardware where flags are used at the pipeline stages to indicate whether valid data has been latched.

$$(6) \qquad \frac{T = \mathsf{ADD}^T}{< T, v, s > \xrightarrow{t} [(\mathsf{ADD}^T, v), (\mathsf{ADD}^F, True)]}$$

Note that the register ADD^F is something introduced to model the behaviour of the architecture. Although also present in the architecture, it is not visible to the user.

The other target transitions are modelled in exactly the same manner.

3.3 Operations

We can now model an operation. This relation is ultimately defined in terms of the above two transitions, the source register transition and the target register transition.

The operation transition relation, $\xrightarrow{o}$, is defined as:

$$\xrightarrow{o} \in (Operation \times State) \times Substitution$$

The semantic description is formed on the structure of an operation as defined in figure 2. We begin with the NOP. Intuitively, the execution of the NOP should leave the state unchanged. We therefore model the meaning of this operation, in a state s, as the empty substitution (when this is later applied to the state, the state will remain unchanged).

$$(7) \qquad \frac{O = \mathsf{NOP}}{< O, s > \xrightarrow{o} \phi}$$

The other forms of operations involve guards. If the guard is "Always", then we will always execute the move. This involves a transition for the source and the target register, and we use the relations defined in the previous two sections. Ultimately, a substitution is yielded.

$$(8) \qquad \frac{O = Always : S : T \quad < S, s > \xrightarrow{s} v \quad < T, v, s > \xrightarrow{t} nv}{< O, s > \xrightarrow{o} nv}$$

Thus a transition ($\xrightarrow{s}$) is made to nv in the case that the source register produces a value v, and the target transition using this value produces a substitution nv. The rule can be read as: if $< S, s > \xrightarrow{s} v$ and $< T, v, s > \xrightarrow{t} nv$ then $< Always : S : T, s > \xrightarrow{o} nv$.

If the guard is "Check", then the execution of the move only takes place if the value in the conditional unit is true; otherwise nothing happens. The false case is defined by:

$$(9) \qquad \frac{O = Check : S : T}{< O, s > \xrightarrow{o} \phi} \text{, if } Guard(s) = False$$

It is easy to see the following:

$$\forall s \in State.Guard(s) = False \quad \Rightarrow \quad < Check : S : T, s > \xrightarrow{o} \phi \text{ and } < \mathsf{NOP}, s > \xrightarrow{o} \phi$$

That is, a transition of an operation with a failed guard and a NOP yield the same (empty) substitution. If, while optimizing code, one can prove that a guard is always *False*, then by the above proposition we can replace it with a NOP. Although this may seem trivial, it is a fact derived from the semantics.

The *Guard* function used in the above rules takes the supplied state and yields the result of the conditional units comparison. It is of type $Guard :: State \to Bool$ and defined as follows:

$$Guard(s) = \begin{cases} True & \text{, if } (s[\mathsf{CND}^R] = 1) \\ False & \text{, otherwise} \end{cases}$$

We write $s[\mathsf{CND}^R]$ for the value of the CND^R register in the state s.

A guard which succeeds has the same semantics as a non-conditional move with the same source and target registers, as can be seen by comparing rules 8 and 10.

$$(10) \qquad \frac{O = Check : S : T \quad < S, s > \stackrel{s}{\to} v \quad < T, v, s > \stackrel{t}{\to} nv}{< O, s > \stackrel{o}{\to} nv} \text{, if } Guard(s) = True$$

Note that, in the definition of the *Guard* function, we use the result register of the conditional unit, a register which cannot be explicitly read by the user. Instead of modelling complex squash lines and timings, the hardware has been abstracted and replaced by a 'pseudo' register.

3.4 Using substitutions

The previous sections used substitutions to represent eventual changes to the state. We now motivate the use of substitutions instead of the more conventional techniques discussed below. A key feature of the semantics is that it describes a parallel architecture. It is natural to expect that the semantics also be parallel, but at the same time deterministic. This is a notorious problem, as we have to merge two states which change in parallel.

One possible solution is to model the parallel execution of two operations in some state s by providing rules allowing the interleaving execution of either operation. This would be written as:

$$\frac{< O_1, s > \to < O_1', s' >}{< O_1 \parallel O_2, s > \to < O_1' \parallel O_2, s' >} \quad \frac{< O_2, s > \to < O_2', s' >}{< O_1 \parallel O_2, s > \to < O_1 \parallel O_2', s' >}$$

Another solution is to assume the existence of a clever merge operator, written here as $+$, which will only merge those portions of the state that have been changed:

$$\frac{< O_1, s > \to s_1 \quad < O_2, s > \to s_2}{< O_1 \parallel O_2, s > \to s_1 + s_2}$$

We can, however, do better than this. Since we know that each operation will not use the same resources (we have the so called disjointness requirement of [17])[1] and that the changes to the state that each operation induces are mutually exclusive (see subsection 3.5), we model operations as being substitutions. That is, (name,value) pairs that can later be applied to the state. We thus model the parallel execution as:

$$\frac{< O_1, s > \to nv_1 \quad < O_2, s > \to nv_2}{< O_1 \parallel O_2, s > \to (s \uplus nv_1) \uplus nv_2}$$

The union operator, $\uplus$, used above takes as arguments a state and substitution, and yields a new updated state. Its type is $\uplus :: State \to Substitution \to State$, and its behaviour is such that for a register r:

$$(s \uplus p)[r] = \begin{cases} (p[r]) & \text{if } (r \in p) \\ (s[r]) & \text{otherwise} \end{cases}$$

[1] The disjointness requirement is present in all well-schedulled programs. Indeed, this is a property of schedulling.

Note that for an empty substitution, the p above would always be empty giving us the equality $(s \uplus \phi)[r] = (s[r])$. That is, $s \uplus \phi = s$.

It is easy to see that if nv_1 and nv_2 are disjoint, then $((s \uplus nv_1) \uplus nv_2)[r] = ((s \uplus nv_2) \uplus nv_1)[r]$.

Armed with this technique, we can now proceed to give a semantics to the rest of the architecture.

3.5 Instruction

We are finally in a position to define the meaning of an instruction. The transition relation for instructions is given by:

$$\xrightarrow{i} \in (Instruction \times State) \times State$$

In defining the semantics of an instruction, we are faced with a choice. The TTA is parallel. However, all "well scheduled" operations that execute in an instruction are independent and will use separate resources. Indeed, this is part of the task of the scheduler. We can then either assume that we are given code which is scheduled correctly, or try and explicitly model incorrect code. In the former scenario, we have:

$$\text{(11)} \qquad \frac{I = (O_1 \parallel O_2) \quad < O_1, s > \xrightarrow{o} nv_1 \quad < O_2, s > \xrightarrow{o} nv_2}{< I, s > \xrightarrow{i} (s \uplus nv_1) \uplus nv_2}$$

The meaning of an instruction is the final state achieved by updating the initial state with the substitutions yielded by the individual operations. Note that, because we have assumed nv_1 and nv_2 to be disjoint, the final state update can take place in an arbitrary order. The task of ensuring that operations yield mutually exclusive substitutions can be given to a *static semantics*. This semantics can include rules checking the well-formedness of the instructions. In effect, it checks that the code is conflict free, a property that would be guaranteed if a scheduler were used. Among other things it can ensure that the maximum size of constants is not exceeded, and perform resource checks. If the architecture being modelling never had a fully-connected network, then the check on which moves are valid can also be incorporated in the static semantics. Implementing the semantics with explicit checking would involve replacing rule 11 by rules which check that the substitutions nv_1 and nv_2 are disjoint, and yield a special terminal configuration, $\perp$, indicating failure. In terms of the hardware, two operations writing to the same register yield an undefined result. For the case of instructions, this would be:

$$\text{(11-a)} \qquad \frac{I = (O_1 \parallel O_2) \quad < O_1, s > \xrightarrow{o} nv_1 \quad < O_2, s > \xrightarrow{o} nv_2}{< I, s > \xrightarrow{i} \perp} \text{ , if } \neg\ Mutex(nv_1, nv_2)$$

$$\text{(11-b)} \qquad \frac{I = (O_1 \parallel O_2) \quad < O_1, s > \xrightarrow{o} nv_1 \quad < O_2, s > \xrightarrow{o} nv_2}{< I, s > \xrightarrow{i} (s \uplus nv_1) \uplus nv_2} \text{ , if } Mutex(nv_1, nv_2)$$

Adopting this form of semantics allows one to reason about the machine at a slightly lower level. For instance, one can now prove that the result of the instruction
$Always$: $\mathsf{CNST\text{-}23}$: ADD^T $\parallel$ $Always$: $\mathsf{CNST\text{-}5}$: ADD^T is always undefined.

As a concrete example of the use of the semantics, we prove that the result of executing the instruction NOP $\parallel$ $Always$: $\mathsf{CNST\text{-}10}$: ADD^O is only a substitution changing the value of the operand register for the add unit to 10. We do this by constructing a *derivation tree*, applying the semantic rules:

$$\frac{<\mathsf{NOP}, s> \xrightarrow{o} \phi \qquad \dfrac{<\mathsf{CNST\text{-}10}, s> \xrightarrow{s} 10 \quad <\mathsf{ADD}^O, 10, s> \xrightarrow{t} [(\mathsf{ADD}^O, 10)]}{<Always : \mathsf{CNST\text{-}10} : \mathsf{ADD}^O, s> \xrightarrow{o} [(\mathsf{ADD}^O, 10)]}}{<\mathsf{NOP} \parallel (Always : \mathsf{CNST\text{-}10} : \mathsf{ADD}^O), s> \xrightarrow{i} (s \uplus \phi) \uplus [(\mathsf{ADD}^O, 10)]}$$

We have thus used the semantics to prove that the above instruction only introduces one change to the state; the operand of the add unit becomes a 10 (since, by the empty substitution, $(s \uplus \phi) \uplus [(\mathsf{ADD}^O, 10)] = s \uplus [(\mathsf{ADD}^O, 10)]$).

Similarly, we can derive that $< Always : \mathsf{CNST\text{-}10} : \mathsf{ADD}^O \parallel \mathsf{NOP}, s > \xrightarrow{i} s \uplus [(\mathsf{ADD}^O, 10)]$ showing that these two instructions are equivalent.

We can go further, and prove that for all operations O, $< O \parallel \mathsf{NOP} > \xrightarrow{i} s'$ and $< \mathsf{NOP} \parallel O > \xrightarrow{i} s''$ and $s' = s''$. We prove this by showing that the resulting states are equal. For the first instruction, we have:

$$\frac{\dfrac{\cdots}{<O, s> \xrightarrow{o} sv} \quad <\mathsf{NOP}, s> \xrightarrow{o} \phi}{<O \parallel \mathsf{NOP}, s> \xrightarrow{i} (s \uplus sv) \uplus \phi}$$

For the second, in the same state s, we have:

$$\frac{<\mathsf{NOP}, s> \xrightarrow{o} \phi \quad \dfrac{\cdots}{<O, s> \xrightarrow{o} sv}}{<\mathsf{NOP} \parallel O, s> \xrightarrow{i} (s \uplus \phi) \uplus sv}$$

By a simple calculation, we have that $(s \uplus sv) \uplus \phi = s \uplus sv$ and $(s \uplus \phi) \uplus sv = s \uplus sv$, showing that the final states are indeed equal.

3.6 Program

Extending the semantics of an instruction in a natural way, we can provide a semantics for a program. This relation makes use of an environment, $\mathcal{C}$, which stores the program. For clarity, we have written the program counter as PC instead of the more explicit $s[\mathsf{INS}^R]$. The transition relation for programs is given by:

$$\xrightarrow{p} \in (State \times State)$$

$$(12) \qquad \frac{\mathcal{C}[PC] = (O_1 \parallel O_2) \quad s' = s \uplus [(\mathsf{INS}^T, PC + 1)] \quad <O_1 \parallel O_2, s'> \xrightarrow{i} s'' \quad s'' \xrightarrow{u} s'''}{\mathcal{C} \vdash s \xrightarrow{p} s'''} \neg Halt(PC)$$

We can read this as: If in environment $\mathcal{C}$ the program counter of the current state is not a terminating one (we have not reached the end of the code stream), then we can look up the instruction at the program counter. A new state, s' is created which has the program counter updated. This is in accordance to the physical architecture which updates the program counter automatically. Using the new state s', a transition is made on the instruction, and the resulting state is updated via the update transition ($\xrightarrow{u}$). Recall that we have been modelling a semi-virtual time system: all updates to the state of the machine due to pipeline changes etc. are performed before the execution of the next instruction. This is exactly what we have modelled above. In fact, each transition of $\xrightarrow{p}$ can be thought of as a virtual time cycle. Finally, we define some halting condition on the program counter which will indicate when the program has finished execution.

An initial state is assumed, that maps the program counter to the first location and every other register to zero.

With a suitable update function (one with a latency of 1 for the add unit) we could prove (by constructing a derivation tree) that the value of the result register in the add unit after a suitable number of cycles would be 10 after executing the following code:

$$\begin{array}{ll} 1: & (Always : \mathsf{CNST\text{-}5} : \mathsf{ADD}^O) \parallel (Always : \mathsf{CNST\text{-}15} : \mathsf{ADD}^T) \\ 2: & (Always : \mathsf{CNST\text{-}5} : \mathsf{ADD}^T) \parallel \mathsf{NOP} \end{array}$$

3.7 Specifying Updates

Finally, we need to specify the update transition. Since the functional units are independent, an update on the state entails an update on all the components of the architecture (all the FUs and RUs). We will assume that we can split all of the components of the state. That is, all state describing the register unit will be called REG etc. Using this, we can specify the update of the state of the entire machine as the separate modular updates of its components.

$$\frac{s = (INS, REG, \ldots, CND) \quad INS \xrightarrow{u^i} INS' \cdots CND \xrightarrow{u^c} CND' \quad s' = (INS', REG', \ldots, CND')}{s \xrightarrow{u} s'} \tag{13}$$

Since we are using a semi-virtual time pipelining scheme, state updates need only be done at one time in the execution of each instruction, namely before the next instruction issue. We begin by illustrating the specification of the register unit. The triggering of a register unit moves the trigger value to the result value. If the unit has not been triggered, then we do nothing. We use the notation $REG = [(\mathsf{REG}^F, False), \ldots]$ to indicate the requirement that the state component of the general purpose register maps the flag to $False$.

$$\frac{REG = [(\mathsf{REG}^F, False), \ldots]}{REG \xrightarrow{u^r} REG} \tag{14}$$

This can be read as follows. The update of a state REG, where the state contains a mapping of REG^F to $False$, is just the state itself. There is no change in the state of the register functional unit if it has not been triggered.

$$\frac{REG = [(\mathsf{REG}^T, v_t), (\mathsf{REG}^F, True), \ldots]}{REG \xrightarrow{u^r} REG \uplus [(\mathsf{REG}^R, v_t), (\mathsf{REG}^F, False)]} \tag{15}$$

If however, the state has the flag register set, then we must yield a new state where this register is reset, and wherein the result register has the value of the trigger register.

The update of the instruction unit is the identity if nothing has been written to the trigger register, or the result register is updated with the value of the trigger register if written. The following two rules illustrate this.

$$\frac{INS = [(\mathsf{INS}^F, False), \ldots]}{INS \xrightarrow{u^i} INS} \tag{16}$$

$$\frac{INS = [(\mathsf{INS}^F, True), (\mathsf{INS}^T, v_t), \ldots]}{INS \xrightarrow{u^i} INS \uplus [(\mathsf{INS}^R, v_t), (\mathsf{INS}^F, False)]} \tag{17}$$

In modelling the load/store unit, we need to keep the state of the memory. Instead of dragging an environment around holding this information, we follow [12] and consider one register as special: if applied to an address, it will yield the value stored at that address. Using this, we could model the load as:

$$\frac{LD = [(\mathsf{LD}^F, False), \ldots]}{LD \xrightarrow{u^l} LD} \tag{18}$$

$$\frac{LD = [(\mathsf{LD}^F, True), (\mathsf{LD}^T, a)]}{LD \xrightarrow{u^l} LD \uplus [(\mathsf{LD}^F, False), (\mathsf{LD}^R, \mathsf{LD}^{MEM}[a])]} \tag{19}$$

We finish this section by illustrating the semantics of the conditional unit update. The specification of the other units is very similar to this, and will be omitted.

We first handle the case of no change:

$$\frac{CND = [(\mathsf{CND}^F, False), \ldots]}{CND \xrightarrow{u^i} CND} \tag{20}$$

Assuming that the conditional unit places a $True$ in the result register if the operand is less than or equal to the trigger, we then have:

$$\frac{CND = [(\mathsf{CND}^F, True), (\mathsf{CND}^O, v_o), (\mathsf{CND}^T, v_t), \ldots]}{CND \xrightarrow{u^i} CND \uplus [(\mathsf{CND}^R, v_o \leq v_t), (\mathsf{CND}^F, False)]} \tag{21}$$

4 Extensions and Animation

We now address extensions to the base semantics, by looking at a proposal for handling long constants, and the addition of pipelining. Throughout this paper, emphasis has been placed on the modularity of the semantics. We will see that the extensions to the architecture are indeed reflected by modular extensions of the semantics. The resulting semantic framework is sufficiently general to model existing TTAs such as the MOVE32INT [6].

4.1 Long Constants

The architecture as described in section 2 has constants embedded in the operation. If the instruction length is not to increase, this constant is by necessity small. An alternative technique is to provide a different instruction format. The proposed format extends the previous format by adding a new type of instruction, which is a long constant move to a register. The syntax for instructions in figure 2 is then replaced with:

$$\begin{array}{rcl} Instruction \ni I & ::= & O \parallel O \\ & | & \mathsf{LONGCNST\text{-}c} : T \end{array}$$

Corresponding semantic rules have to be added for this case:

$$\frac{S = \mathsf{LONGCNST\text{-}c}}{< S, s > \stackrel{s}{\rightarrow} c} \tag{22}$$

$$\frac{I = \mathsf{LONGCNST\text{-}c} : T \quad < \mathsf{LONGCNST\text{-}c}, s > \stackrel{s}{\rightarrow} v \quad < T, v, s > \stackrel{t}{\rightarrow} nv}{< I, s > \stackrel{i}{\rightarrow} s \uplus nv} \tag{23}$$

This completes the changes necessary to model an architecture with long constants. As can be seen, only a change in the syntax and two additions to the semantics were necessary, highlighting the benefit of the modular approach.

The new architecture semantics could be compared to the old in various ways. One way is to prove that the architectures are equivalent in some sense. For example, that we could simulate moving a long constant to a register by adding two small constants together in the register. Another comparison would be to look at generated code and compare the efficiency of the two strategies. Unfortunately, the scheduler plays a large rôle in this area, and one cannot provide hard evidence independent of any scheduler. However, an animation of the architectures would provide a test bed for these experiments.

4.2 A Pipelined Add Unit

We examine the consequences of adding a single pipeline stage to the add unit. The only change that has to be made to the semantic description is at the update function for the add unit. We introduce two new pseudo registers, $\mathsf{ADD_2}^F$ and $\mathsf{ADD_2}^R$. Intuitively, after these are triggered the addition will take place, and the result will not be placed in the result register but in register $\mathsf{ADD_2}^R$. The associated flag will also be set to indicate that data is sitting in the pipeline stage. The pipeline stages progress with virtual time, and so every update, if a pipeline stage has data, it will move the data to the output register. The real architecture keeps a valid bit associated with each pipeline stage, similar to what we do here. Again, the pseudo registers are not visible to the user of the architecture, but similar mechanisms are implemented in the hardware.

The update for the modified functional unit can then be described by four rules. If there is no data in the pipeline, and the unit has not been triggered, then an update is an identity on the state.

$$\frac{ADD = [(\mathsf{ADD}^F, False), (\mathsf{ADD_2}^F, False), \ldots]}{ADD \stackrel{u^a}{\rightarrow} ADD}$$

If, however, the pipeline contains data, then this is fed through to the result register.

$$\frac{ADD = [(\mathsf{ADD}^F, False), (\mathsf{ADD_2}^R, a2r), (\mathsf{ADD_2}^F, True), \ldots]}{ADD \stackrel{u^a}{\rightarrow} ADD[(\mathsf{ADD_2}^F, False), (\mathsf{ADD}^R, a2r)]}$$

If the pipeline contains no data, and the unit is triggered, then we have:

$$\frac{ADD = [(\mathsf{ADD}^F, True), (\mathsf{ADD}^O, ao), (\mathsf{ADD}^R, ar), (\mathsf{ADD_2}^F, False), \ldots]}{ADD \stackrel{u^a}{\rightarrow} ADD[(\mathsf{ADD_2}^F, True), (\mathsf{ADD_2}^R, (ao + ar)), (\mathsf{ADD}^F, False)]}$$

There is also a case for when there is both a triggered register, and the pipeline contains data, which we omit as it it is similar to the previous rules. The above technique for modelling pipelines has the disadvantage that we need a number of rules exponential in the depth of the pipeline. This can be avoided by introducing a list of operand/flag values which is cycled on every update. Indeed, this is how the semantics has been animated(see section 4.5).

Again, the modularity of the architecture and semantics is shown by the need for only localised changes.

4.3 Other pipelining schemes

We have thus far only considered functional units employing a semi virtual-time latching scheme. This leads to a simplification of the semantics as updates only have to occur at one time in the cycle, after any possible changes made by executing an instruction.

Other schemes are discussed in [4], and we briefly examine these.

In a true virtual-time latching scheme, all stages except the first progress synchronously with virtual-time. That it, there is no distinction between the operand and trigger registers. Modelling this scheme would simplify our presentation, as there is no need to keep flag registers indicating whether a stage has valid data or not. In addition, the update relation simplifies as we update irrespective of whether anything has been written.

Thus the rules 14 and 15 for instance, would be replaced by:

$$\frac{REG = [(\mathsf{REG}^T, v_t)]}{REG \stackrel{u^r}{\rightarrow} REG \uplus [(\mathsf{REG}^R, v_t)]}$$

indicating that the update is to take place every time a general update occurs (which is synchronous with virtual time).

In the hybrid latching scheme [4], stages only advance if they will not overwrite any unused values. In this case, the result value of a pipelined add unit will not be overwritten by a succeeding add until it has been read. This behaviour can be modelled by having the update functions examine the flags more carefully before updating the state of the unit. Unfortunately this scheme also complicates the semantics somewhat. Whenever a trigger or operand register is written to, it has to be checked that it does not contain valid values. If it does, a lock is needed. Such checking is unfortunate as we no longer have the separation of state update/access and FU access.

4.4 Locking

If a memory unit uses locking when a cache miss occurs, then the entire machine will stall until the datum is available (a global lock would be in effect). Irrespective of whether a cache miss occurred or not, the machine should still behave in the same way (of course, the machine will take longer to produce the result: a non-functional aspect). Our model only captures virtual time aspects. Modelling the cycle impact of locking could be handled by having a trigger relation returning not only a substitution, but also a cycle count. This cycle count would be greater than zero in the event of some modelled cache miss.

The semantic framework presented here does not model non-functional aspects very well. Indeed, we have purposefully abstracted away from modelling control signals, squash lines and locking. A lower-level semantics of the VLSI circuitry would take this into account. However, animating the semantics provides us with one way of retrieving some non-functional behaviour, if properly modelled as suggested above.

(CNST-10 : ADD^O	$\|$	NOP)
(CNST-20 : ADD^T	$\|$	NOP)
(ADD^R : REG^T	$\|$	NOP)
(REG^R : ADD^O	$\|$	NOP)
(CNST-30 : ADD^T	$\|$	NOP)
(ADD^R : REG^T	$\|$	NOP)

Figure 3: Sequential Unoptimised Code

4.5 Determinism and Animation

The above operational semantics can be proved to be *deterministic* by structural induction. By deterministic, we mean that for all choices of I, s, s' and s'' we have:

$$< I, s > \xrightarrow{i} s' \;\&\; < I, s > \xrightarrow{i} s'' \Rightarrow s' = s''$$

That is, given an instruction and a state we can unambiguously determine the following state. As this is the case, we can capture all of the semantics rules by functions which can easily be implemented. The extended operational semantics given above has been implemented in the functional language Haskell [15], and provides a useful prototype of the architecture. By 'running' the semantics we can get an idea of what state changes occur after each virtual cycle and update. We can also gather statistics, and some non-functional aspects such as the number of virtual time cycles.

Here is an excerpt from the code implementing the transition rule on an operation with an 'always' guard (Rule 8), illustrating the proximity of the semantics to the animation:

```
> transition (Oper ALWAYS src trg) s  = let v  = tran_s src s
>                                           nv = tran_t trg v s
>                                       in
>                                           nv
```

In addition, techniques are available which can adapt the operational semantics to create reasonably efficient abstract machines [8]. The animated semantics can also provide verification of compilers and hardware. The current implementation allows the user to examine the state of the machine after every instruction issue. It can be extended by annotating the transitions to gather statistics, such as cycle counts. If a parameterisable compiler was available, one could evaluate various architectures by changing the implementation of the semantics accordingly and running the programs. The animation thus provides a tool for examining the non-functional behaviour of the semantics. The animation can also be used to some degree to validate the semantics (given a hardware implementation), or verify the hardware (using the semantics as a base reference specification).

4.6 Proving correctness of code transformations

A compiler for a VLIW machine will at some time in the compile cycle produce sequential code which is later optimised and schedulled into parallel code. One optimisation which can be applied to TTAs is the removal of redundant register file writes. Figure 3 illustrates a sequential piece of code which adds two numbers, writes the result to a register file and then adds 30 to this value writing the final result. We assume a latency of one, and place NOPs to emphasize that the code is unschedulled. We will call this code $\mathcal{C}^u$. Figure 4

$$\begin{array}{lcl}
(\mathsf{CNST\text{-}10} : \mathsf{ADD}^O & \| & \mathsf{CNST\text{-}20} : \mathsf{ADD}^T) \\
(\mathsf{ADD}^R : \mathsf{ADD}^O & \| & \mathsf{CNST\text{-}30} : \mathsf{ADD}^T) \\
(\mathsf{ADD}^R : \mathsf{REG}^T & \| & \mathsf{NOP})
\end{array}$$

Figure 4: Schedulled Optimised Code

illustrates the code after schedulling and the removing of the redundant register file write. We will call this code $\mathcal{C}^o$.

These two sequences of code can be shown to be equivalent in the sense that they produce the same final state given the same initial state. [2] The proof involves the construction of a number of derivation trees. For each of the sequences of code $\mathcal{C}^o$ and $\mathcal{C}^u$ a number of program transition steps can be derived. To prove the first step in $\mathcal{C}^u$ for instance, we would construct the derivation tree for $\mathcal{C}^u \vdash s_i \xrightarrow{p} s$, which would involve the construction of the derivation tree for the instruction similar to those in subsection 3.5. Since the final states will be the same, we conclude that the two pieces of code are equivalent. This allows us to verify the correctness of the program optimisations and schedulling. However, the current framework only allows us to prove equivalence of given pieces of code. By building on the semantics following [1] we can prove equivalence of arbitrary pieces of code. By doing this, we hope to prove the various optimisation techniques and schedulling of code to be correct.

5 Future Work

There is much related work in the area of hardware description and modelling languages. However, much of this work is based on a lower level of description, for instance VHDL or Boyer-Moore theorem provers applied to low level descriptions [2, 10]. Recently there has been work on hardware description languages with a good formal semantic footing, for instance HML [14] and [13] where functional languages are used as description languages. We believe that a gap exists in the specification languages lying between languages and hardware, and that the system presented in this paper goes some way towards bridging this.

We have already seen examples of using the given semantics to prove small properties about programs or instructions. One would often like to prove properties about programs in general. Aiken [1] builds on a transition semantics for parallel execution by constructing a notion of an execution trace. The execution trace is defined as the successive states that a program goes through while executing. These states are just those described by our transition relation ($\xrightarrow{p}$) in rule 12. Using this, he develops provably correct code transformations which can be used to prove percolation scheduling correct. Although Aiken's framework has to be extended (he uses a machine with unlimited resources, and his semantics is a general one for parallel execution) we believe that our proof framework will be useful in proving similar properties for scheduling techniques for TTAs. Future work will these aspects of the semantics.

6 Conclusions

This paper has introduced a technique for modelling architectures based on their abstract syntax. The model is based on an operational semantics derived from the structure of the machine, and is in essence a semantics of the instruction set. The hierarchical and modular structure of transport triggered architectures allows for such a modular syntax and semantics, and a TTA was used as an example to illustrate the techniques. In

[2]The final states will of course differ in the value of the program counter.

principle, these could be applied to other VLIW-like architectures which share this hierarchical structure. Since the description is manageable and readable, we have provided a precise specification of the architecture which can be used by compiler writers and as a formal design document. The semantics has been animated, providing a simulator on which to explore the design space of the architecture. It can also be used to verify the behaviour of physical architectures.

7 Acknowledgements

The authors thank Marcel Beemster and Hugh McEvoy for their copious comments which greatly improved the paper. The first author is supported by the Netherlands Computer Science Research Foundation with financial support from the Netherlands Organization for Scientific Research (NWO).

References

[1] A. Aiken. *Compaction-Based Parallelization*. PhD thesis, Department of Computer Science, Cornell University, 1988.

[2] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, 1996.

[3] H. Corporaal. Evaluating transport triggered architectures for scalar applications. *Microprocessing and Microprogramming*, (38):45–52, 1993.

[4] H. Corporaal. *Transport Triggered Architectures: Design and Evaluation*. PhD thesis, Technische Universiteit Delft, 1995.

[5] H. Corporaal and H. J. M. Mulder. Move: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, New Mexico, 1991. IEEE Computer Society Press, Los Alamitos, California.

[6] H. Corporaal and P van der Arend. MOVE32INT, a sea of gates realization of a high performance transport triggered architecture. *Microprocessor and Microprogramming*, 38:53–60, 1993.

[7] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19 of *SIGPLAN Notices*, pages 37–47, 1984.

[8] J. Hannan. Operational semantics-directed compilers and machine architectures. *Transactions on Programming Languages and Systems*, 16(4):1215–1247, 1994.

[9] J. Hoogerbrugge and H. Corporaal. Comparing software pipelining for an operantion-triggered and a transport-triggered architecture. Technical report, Delft University of Technology, 1993.

[10] W. A. Hunt and B. C. Brock. The verification of a bit-slice ALU. In M. Leeser and G. Brown, editors, *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 282–306. Springer-Verlag, 1989.

[11] G. Jones and M. Sheeran. Circuit design in Ruby. In J Staunstrup, editor, *Formal methods for VLSI design*, pages 13–70. North-Holland, 1990.

[12] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *2nd Operating System Design and Implementation (OSDI)*, Seattle, Washington, 1996.

[13] J. T. O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In J. Launchbury and P. Sansom, editors, *Functional Programming, Glasgow 1992*, Workshops in Computing. Springer-Verlag, 1992.

[14] J. O'Leary, M Linderman, M leeser, and M Aagaard. HML: A hardware description language based on standard ML. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of the 11th IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL'93)*, pages 327–334. IFIP, North-Holland, 1993.

[15] J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. D. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, S. Peyton Jones, A. Reid, and P. Wadler. Report on the programming language Haskell (version 1.3). Technical Report YALEU/DCS/RR-1106, Yale University, 1996.

[16] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department , Aarhus University, Denmark, 1981.

[17] G. D. Plotkin. An operational semantics for CSP. Technical Report CSR-114-82, University of Edinburgh, 1982.