Distributed Processing, IEEE Computer Society Press, p. 493--500,
Jan 24-26th, 1996, Braga, Portugal.

Using ITL and Tempura for Large Scale Specification and Simulation*

Antonio Cau† Hussein Zedan† Nick Coleman Ben Moszkowski

School of Computing
and Mathematical Sciences
Liverpool John Moores University
Liverpool L3 3AF, UK

Department of Electrical
and Electronic Engineering
University of Newcastle upon Tyne
Newcastle NE1 7RU, UK

Abstract

ITL and Tempura are used for respectively the formal specification and simulation of a large scale system, namely the general purpose multi-threaded dataflow processor EP/3. This paper shows that this processor can be specified concisely within ITL and simulated with Tempura. But it also discusses some problems encountered during the specification and simulation, and indicates what should be added to solve those problems.

1 Introduction

There has been a considerable debate about the use and relevance of formal methods in the development of computing systems (both software and hardware). Some claim that formal methods offer a complete solution to the problems encountered in such development. Others put the claim that formal methods are of little or no use, or that their utility is severely hindered by their cost.

However, it would be over-enthusiastic to claim that a formal development technique could provide a panacea for the problem involved in developing useful computer systems. Indeed, there are too many aspects which are not amenable to formal representation or reasoning. For example, it is hard to envisage a way in which the process of requirements elicitation could be totally formalised; it is true that some requirements can be denoted using a formal notation, and possible inconsistencies found, but the completeness of the requirements (with respect to client's intentions) cannot be formally proven.

As digital devices are deployed in a growing number of high integrity applications there is increased anxiety about the dependability of such systems. Furthermore, the rapid growth of the VLSI market has meant that manufacturers are under pressure to deliver increasingly complex, reliable and cost-effective products within a short time scale. Formal techniques have clearly had an impact on the design of safety critical systems, and have been shown to be commercially advantageous as demonstrated in the production of Inmos IMS T800 floating point unit [8]. We believe that using formal techniques in the production of systems should be viewed as a means of delivering correctness (with respect to requirements) and hence enhanced quality.

One such formal technique is ITL which has been developed over a range of years and has been applied to a various number of systems. However it has not been applied to the design of large scale hardware systems. The aim of this paper is an analysis and discussion of the benefits of the use of ITL and its associated executable language Tempura [10] for such large scale hardware systems. Our chosen hardware is a general-purpose multithreaded dataflow computer known as EP/3 (Event Processor/3) [3]. EP/3 is intended primarily as a vehicle for exploratory research in high-performance computer structures.

1.1 Related work

There have been a plethora of formalisms proposed and used in conjunction with digital system specification and verification. A complete overview of these formalisms is outside the scope of the present paper, however, in this section we will only highlight some of the well known formalism classes used in this field.

A number of hardware specific calculi have been developed and used. Some of these are supported by theorem proving or other checking tools. Barrow's seminal work on VERIFY [1] (a Prolog program for checking the correctness of finite state machines) and Milne's Circle (a calculus based on CCS) [9] for specifying and analysing circuit behaviour.

General purpose logics have been proposed. The

*Supported by EPSRC Research Grant GR/K25922

†Current address: Department of Computer Science, de Montfort University, Leicester LE1 9BH, UK

Boyer-Moore theorem prover is a notable example [2]. Recently, CLInc has demonstrated that Boyer-Moore can be used successfully for non-trivial hardware verification cases. Higher order logic was first used by Hanna and Daeche who developed the VERITAS theorem proving system [6]. HOL [5] is also a machine-oriented formulation of higher logic based on Church's lambda calculus.

Algebraic specification languages have also been used; a notable example is the use of OBJ specifications with hardware. OBJ-T [13] version of the language was used to specify and test hardware building blocks. UMIST OBJ [4] has also been used to specify simple devices with theorem proving support from REVE [7].

Computer hardware description languages were the first textual descriptive techniques to be used in the design of hardware. Examples include ELLA [12] and VHDL.

Many ideas originating from reactive systems theory including temporal logics, are relevant to the specification and verification of synchronous and asynchronous digital systems. Although propositional temporal logic can be used for reasoning about hardware, interval temporal logic is particularly interesting for hardware verification.

To our best knowledge, there have not been any serious attempts to specify, verify and design a large-scale hardware system in interval temporal logic.

1.2 Paper Organisation

In Section 2 we will describe the EP/3 processor architecture and in section 3 we will present an overview of ITL and its associated programming language Tempura. The specification and simulation of EP/3 in ITL are discussed in section 4. We give our evaluations in section 5 and indicate future work.

2 The EP/3 Processor

The system used in this case-study is the Event Processor [3]. Its processing elements are based on the multithreaded principle, in which a main memory is available for explicit storage of data (known as static operands), in addition to the normal circulation of tokens (flow operands). The processors are designed for individual high speed (150MHz with ≈ 1 cycle instruction execution) with the facility for assembly into a small multiprocessing array with shared memory. The latter aims at almost 100% load-balancing efficiency by dynamic scheduling at instruction thread level.

For the purpose of this case-study, a single processing element will be described, as shown in figure 1. It consists of a circular pipeline in which the 8 units communicate via highways[1]. Each unit will be described

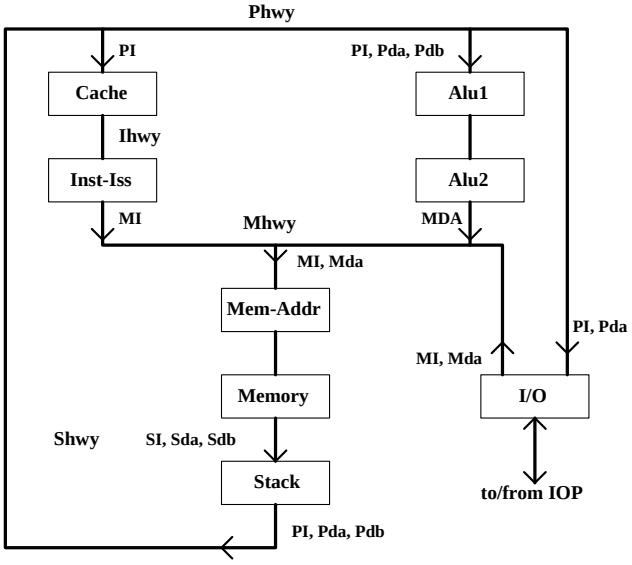

Figure 1: The EP/3 processor

below.

1: The Instruction Issue unit (Inst-Iss) receives instructions from the Cache unit. It will decode these instructions and issue these decoded instructions onto the MI field of the memory highway (Mhwy). All instructions consists of a Command Field and a Destination Field. In the case of multi-target instructions, the Inst-Iss unit will issue the instruction once for each destination, using separate cycle for each. An Interlock signal from the control highway (Zhwy) will prevent any further input into the Inst-Iss unit during this time.

2: The Alu1 unit receives executable parcels of work from the Stack unit via the processor highway (Phwy). These parcels consist of a command field PI and data operands Pda and Pdb. If the PI is a logical or arithmetic command then the Alu1 unit will calculate the low-order of the result within one cycle and send this to the Alu2 unit together with the command otherwise it will send one of the operands plus the command field to the Alu2 unit.

3: The Alu2 unit receives an operand and a command from the Alu1 unit. If the command is a shift or rotate instruction it will execute it on the operand within one cycle and send the result onto the Mda field of the Mhwy otherwise it will complete the high-order

[1]Note: the Zhwy in figure 1 is omitted because it is connected to each unit.

of an arithmetic operation from the Alu1 unit and send the result onto the Mda field of the Mhwy.

4: The Memory Address unit (Mem-Addr) receives from the Mhwy the instruction and its flow operand. The Mem-Addr unit calculates the effective address of the static operand. This address together with the instruction and the flow operand are sent to the Memory unit.

5: The Memory unit will fetch the contents of the address received from the Mem-Addr unit and will issue them (Sdb) together with the instruction (SI) and the flow operand (Sda) onto the stack highway (Shwy).

6: The Stack unit receives complete executable parcels of work destined for the Alu units via the Phwy, but in order to buffer variations in the rate of flow between the Mhwy and Phwy the Stack unit is interposed. If the Interlock signal from the Zhwy is present it will store the parcel, and wait until some future time when the Phwy is clear but nothing is present on the Shwy. It will then issue any stored instruction parcels. In the absence of an Interlock signal, the Stack unit will pass the input parcel from the Shwy straight through to the Phwy. The Stack unit also plays a special role during cache and i/o operations (see the description of the I/O unit).

7: The Cache unit receives via the Phwy instructions PI. The destination field of PI is used by the Cache unit to fetch the corresponding target instruction which is sent to the Inst-Iss unit. When a target instruction is requested which is not present in the Cache unit, a stream of memory accesses must be made to fill the relevant cache block. This loading function will be performed by the I/O unit.

8: The I/O unit is an interface to an external I/O processor IOP. It accesses the Memory unit by issueing dummy instructions, together with a flow operand, onto the Mhwy. These cause the Memory unit either to read out a word onto Sdb, or to write the flow operand. In the case of a read, the data issued by the Memory unit enters the Stack unit. A second function performed by the I/O unit is related to cache loading. When a target instruction is requested which is not present in the Cache unit, a stream of memory accesses must be made to fill the relevant cache block. Since the I/O unit already contains hardware for generating memory accesses, it is convenient to use the unit for this purpose. The PI field is therefore carried into the I/O unit, which contains the cache tag memory and address comparator. When a cache miss is detected by the I/O unit, the Interlock signal is asserted and several consecutive memory accesses are made onto the Mhwy. The requested data is passed out of the Memory unit into the Stack unit, as before. I/O and cache operations are identified by a special tag on the Mhwy and Shwy; this causes the Stack unit to ignore any such data.

I/O transfers and cache loading each involve serial operations, and are somewhat too complex to handle with hardwired control. The I/O is therefore microcoded.

3 Interval Temporal Logic and Tempura

This section describes the syntax and informal semantics of the Interval Temporal Logic (ITL) and gives the syntax of the executable part of ITL, i.e., the Tempura language. For a more succinct exposition and the formal semantics see [10].

An interval is considered to be a finite sequence of states, where a state is a mapping from variables to their values. The length of an interval is equal to one less than the number of states in the interval (i.e., a one state interval has length 0).

The syntax of ITL is defined in Table 1 where a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol, p is a predicate symbol.

Table 1: Syntax of ITL

Expressions	
$exp ::=$	$a \mid A \mid g(exp_1, \ldots, exp_n) \mid \iota a\colon f$
Formulas	
$f ::=$	$p(exp_1, \ldots, exp_n) \mid exp_1 = exp_2 \mid \neg f$ $\mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2 \mid f^*$

The informal semantics of the most interesting constructs are as follows:

- $\iota a\colon f$: the value of a such that f holds.
- $\forall v \bullet f$: for all v such that f holds.
- skip: unit interval (length 1).
- $f_1 ; f_2$: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix.
- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds.

Frequently used abbreviations are listed in table 2.

Tempura is an executable subset of ITL, its syntax resembles that of ITL. It has as data-structures

Table 2: Frequently used abbreviations

$f_1 \supset f_2$	$\stackrel{\text{def}}{\equiv} \neg(f_1 \wedge \neg f_2)$
$f_1 \vee f_2$	$\stackrel{\text{def}}{\equiv} \neg(\neg p_1 \wedge \neg p_2)$
$f_1 \equiv f_2$	$\stackrel{\text{def}}{\equiv} (f_1 \supset f_2) \wedge (f_1 \supset f_2)$
$\exists v \bullet f$	$\stackrel{\text{def}}{\equiv} \neg\forall v \bullet \neg f$
$\bigcirc f$	$\stackrel{\text{def}}{\equiv}$ skip ; f (next f)
$\Diamond f$	$\stackrel{\text{def}}{\equiv} true\,; f$ (sometimes f)
$\Box f$	$\stackrel{\text{def}}{\equiv} \neg \Diamond \neg f$ (always f)
$\circledcirc f$	$\stackrel{\text{def}}{\equiv} \neg \bigcirc \neg f$ (weak next f)
if f_0 then f_1 else f_2	$\stackrel{\text{def}}{\equiv} (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$
$A := exp$	$\stackrel{\text{def}}{\equiv} \bigcirc A = exp$
$more$	$\stackrel{\text{def}}{\equiv} \bigcirc true$ (non-empty interval)
empty	$\stackrel{\text{def}}{\equiv} \neg more$ (empty interval)
$fin\ f$	$\stackrel{\text{def}}{\equiv} \Diamond$(empty $\wedge f$)
while f_0 do f_1	$\stackrel{\text{def}}{\equiv} (f_0 \wedge f_1)^* \wedge fin\ \neg f_0$
repeat f_0 until f_1	$\stackrel{\text{def}}{\equiv} f_0$; (while $\neg f_1$ do f_0)

integers and booleans and the list construct to built more complex ones. The standard operations on expressions are available like $+, -, *, /, div, mod, =, >$ $, or, and,$. The basic statements are as follows with b a formula without temporal operators (with the corresponding ITL construct in parentheses behind it): f_1 *and* f_2 ($f_1 \wedge f_2$), $A := exp$ ($A := exp$), *more* (*more*), empty (empty), *sometimes* ($\Diamond$), *always* ($\Box$), *true* (*true*) , *false* (*false*), if b then f_1 else f_2 (if b then f_1 else f_2), while b do f (while b do f), repeat b until f (repeat b until f). See [10] for more statements. With the $define\ name(p_1, \ldots, p_n) = f$ and $define\ name(p_1, \ldots, p_n) = exp$ one can define "procedures" and functions (recursive definitions are allowed).

4 The Specification and Simulation of the EP/3

This section describes some parts of the specification of the EP/3 in ITL. The whole specification consists of a temporal formula of about 3400 lines, so this won't be given.

4.1 The Specification

The description of the processor from which we started to write the formal specification, consisted of what is given in section 2 plus a Pascal program that simulates the processor. The first formal specification was therefore of sequential nature, i.e., the units of the processor were sequentially composed. This however was not a faithful specification of the processor because each unit should work in parallel. Luckily the specification was such that the transformation into a "parallel" version was not so difficult. Only the interlock signal caused some difficulties. The specification of the processor is a big ITL formula of about 3400 lines. The general structure of the formula is as follows:

$$
\begin{array}{l}
epsim(IO) \stackrel{\text{def}}{\equiv} \\
\exists Stopped, Zlocka, Zlockb, Zvalid, Zioreq, \\
\quad Phwy, Ihwy, Mhwy, Shwy : (\\
\quad init() \quad \wedge \\
\quad \text{repeat} (\\
\qquad \text{skip} \quad \wedge \\
\qquad cache(Phwy, Zlockb \vee Zlocka, \quad Ihwy) \quad \wedge \\
\qquad instis(Ihwy, Zlocka, Zvalid, Zioreq, \\
\qquad\qquad Mhwy, Zlockb, Stopped) \quad \wedge \\
\qquad alu(Phwy, Zlockb \vee Zlocka, Zioreq, \\
\qquad\qquad Mhwy, Zvalid) \quad \wedge \\
\qquad io(IO, Phwy, Zvalid, Zlockb, \\
\qquad\qquad Mhwy, Zioreq, Zlocka) \quad \wedge \\
\qquad mem(Mhwy, \quad Shwy) \quad \wedge \\
\qquad stk(Shwy, Zlockb \vee Zlocka, \quad Phwy) \\
\quad) \text{ until } Stopped \\
)
\end{array}
$$

The `init()` formula initializes the values of the variables. The `repeat ( S ) until Stopped` repeats "executing" the `S` formula until `Stopped` is true which is the case if an stop instruction is executed by the processor. The `S` formula is an $\wedge$ (and) formula of 7 sub-formulas which means that these sub-formulas are "executed" in parallel.

- The skip formula describes that we use an interval of two states namely the state of the processor before and after each clock cycle.
- The $cache(Phwy, Zlockb \vee Zlocka, \quad Ihwy)$ formula describes the behavior of the Cache unit. It has as inputs the processor highway $Phwy$ and the interlock signal which consists of the logical or of respectively the lock signal $Zlocka$ issued by the I/O unit and the lock signal $Zlockb$ issued by the Inst-Iss unit. This means that both units can lock the pipeline. The Cache outputs on the instruction highway $Ihwy$.
- The $instis(Ihwy, Zlocka, Zvalid, Zioreq, Mhwy, Zlockb, Stopped)$ formula describes the behavior of the Inst-Iss unit. It has as inputs the instruction

highway $Ihwy$ and the lock signal $Zlocka$ from the I/O unit. It has furthermore as input the signal $Zvalid$ from the Alu unit indicating if data is valid, and the signal $Zioreq$ from the I/O unit indicating if it is allowed to output on the memory highway $Mhwy$. It issues also the lock signal $Zlockb$ for interlocking. If the stop instruction is issued the *Stopped* signal will be on.

- The $alu(Phwy, Zlockb \vee Zlocka, Zioreq, Mhwy, Zvalid)$ formula describes the behavior of the Alu1 and Alu2 unit. It is defined as

$$\begin{array}{l} \exists Alu2in \bullet (\\ \quad alu1(Phwy, Zlockb \vee Zlocka, Alu2in) \quad \wedge \\ \quad alu2(Alu2in, Zlockb \vee Zlocka, Zioreq, \\ \qquad Mhwy, Zvalid) \\ ). \end{array}$$

  So the Alu1 unit has as inputs the processor highway $Phwy$ and the interlock signal $Zlockb \vee Zlocka$. It outputs $Alu2in$ to the Alu2 unit[2]. The Alu2 unit has as further inputs the interlock signal $Zlockb \vee Zlocka$ and the signal $Zioreq$ from the I/O unit indicating if it is allowed to output on the memory highway $Mhwy$. It outputs furthermore the signal $Zvalid$ indicating if the computed data is valid.

- The $io(IO, Phwy, Zvalid, Zlockb, Mhwy, Zioreq, Zlocka)$ formula describes the behavior of the I/O unit. It has as inputs the stream IO from the external I/O processor IOP, and the processor highway $Phwy$, and the signal $Zvalid$ from the Alu unit, and the lock signal $Zlockb$ from the Inst-Iss unit. It outputs on the memory highway $Mhwy$ and it generates the signal $Zioreq$ during this output preventing that the Alu and the Inst-Iss unit generate output on the $Mhwy$. It also generates the lock signal $Zlocka$ for interlock.

- The $mem(Mhwy, Shwy)$ formula describes the behavior of the Mem-Addr and Memory unit. It is defined as

$$\begin{array}{l} \exists Mem2in \bullet (\\ \quad mem1(Mhwy, Mem2in) \quad \wedge \\ \quad mem2(Mem2in, Shwy) \\ ). \end{array}$$

  $mem1(Mhwy, Mem2in)$ describes the Mem-Addr unit and has as input the memory highway $Mhwy$ and as output $Mem2in$ to the Memory unit. The Memory unit is described by the $mem2(Mem2in, Shwy)$ formula and has as output the stack highway $Shwy$.

- The $stk(Shwy, Zlockb \vee Zlocka, Phwy)$ formula describes the behavior of the Stack unit. It has as inputs the stack highway $Shwy$ and the interlock signal $Zlockb \vee Zlocka$ and it outputs on the processor highway $Phwy$.

[2]The current specification of the Alu units is such that Alu1 computes the arithmetic and logical operations as opposed to the informal specification of section 2

4.2 The Simulation

We will simulate the execution of a small machine code program on the processor. This program consists of two threads running in parallel on the processor, one thread subtracts from 0 the value 31 and the other thread adds 31 to 0. The machine code program is as follows:

```
1.      QS   F1,Z /X2 /X1
2. X1   S    F2,F1
3.      WS   F2,Y1 /END
4. X2   A    F3,F1
5.      WS   F3,Y2 /END
6. END  STOP
7. Z    DCW 31
8. Y1
9. Y2
```

The program performs the following: 1. load flow F1 with the value of Z (i.e., 31) and "goto" /X1 and /X2 , 2. subtract from flow F2 the value of flow F1, 3. write the value of flow F2 to Y1 and goto END, 4. add to flow F3 the value of flow F1, 5. write the value of flow F3 to Y2 and goto END, 6. stop the computation, 7. the variable Z (with value 31), 8. the variable Y1, 9. the variable Y2. The stages of the pipeline are illustrated in figure 2 where

A	is	Cache and Alu1
B	is	Inst-Iss and Alu2
C	is	Mem-Addr
D	is	Memory
E	is	Stack

and the numbers correspond to the instructions.

At stage 1 the Inst-Iss unit gets the first instruction from the Cache and because the instruction has two destinations this instruction is split into two instructions (1.1 and 1.2). It first issues the instruction with the first destination to the Mem-Addr unit (stage 2) and then the same instruction with the second destination (stage 3). Meanwhile the Mem-Addr has sent the first instruction to the Memory (stage 3) where the contents of the Z variable are fetched. The contents of the flows and the value of the Z variable are sent to the Stack unit (stage 4) and the Memory unit fetches the contents of the Z variable for the second instruction.

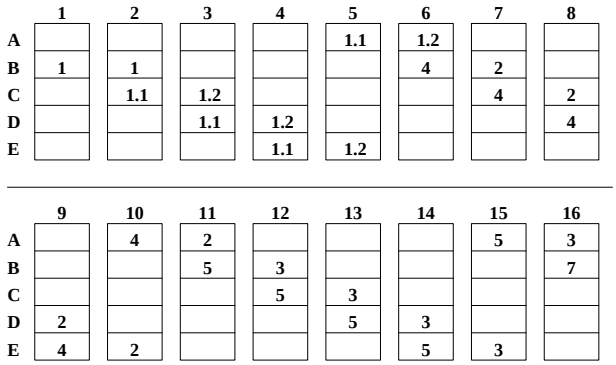

Figure 2: The stages of the pipeline during execution.

The Stack sends the first instruction to the Alu1 and the Cache (stage 5). The Cache fetches the destination instruction where to the Alu unit should send the result of the first instruction (this instruction is `4. X2 A F3,F1`) and sends it to the Inst-Iss unit (stage 6). At stage 7 this instruction is sent to the Mem-Addr unit together with the result of the Alu unit (flow F1 is loaded with the contents of the Z variable). Meanwhile the Cache has fetched the destination instruction for the second instruction (namely 2. X1 S F2,F1) and this is sent at stage 8 to the Mem-Addr unit together with the result of the Alu1 unit (the same as the first instruction). These two "destination" instructions migrate through the Mem-Addr, the Memory and the Stack unit (stages 7-10) towards the Alu1 unit and the Cache. The same procedure as before is followed: the Cache fetches the destination instruction and the Alu unit computes the result. For the first instruction this result is "add the contents of flow F1 to the contents of flow F3 and write them to flow F3" (stage 12) and the destination instruction is `5. WS F3,Y2 /END`. For the second instruction this result is "subtract the contents of flow F1 from flow F2 and write them to flow F2" (stage 13). The two destination instructions again migrate through the Mem-Addr and the Memory unit. The Memory unit writes at the arrival of the first instruction (stage 13) the contents of flow F3 to variable $Y2$ and at the arrival of the first instruction (stage 14) the contents of flow F2 to variable $Y1$. The two instructions then migrates through the Stack to arrive at the Alu1 and the Cache unit (stages 15-16) where the Cache fetches the destination instruction of the first instruction, i.e., `6. END STOP` and sends it to the Inst-Iss unit (stage 16). Because this is the halt instruction the processor stops. The output as generated by the simulator is shown in the appendix.

This sample machine program works fine on the processor but there is a problem. This problem is you can be observe when you execute the following program:

```
1. X  AS F0,Y /X /X
2. Y  DCW 1
```

This program adds 1 to flow F0 and then sends the result twice to itself (it constructs a binary tree of adding instructions).

The stages of the pipeline are illustrated in figure 3. Whenever an instruction enters the Inst-Iss (B) it will be split into 2 instructions. Because the pipeline has a length five this will eventually result in a situation (for instance stage 12) that there is "no room" in the pipeline; the instruction (i.e., 1.2.1) in the Stack (E) unit will then be put on the stack. In this example because the stack is of finite length, this will result eventually in a stack overflow. This problem should then be solved on operating system level.

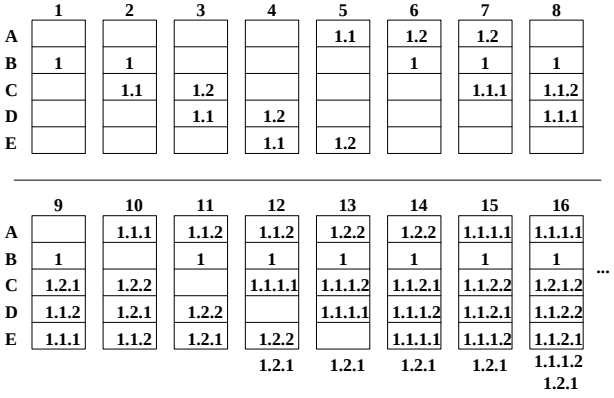

Figure 3: The stages of the pipeline during execution.

5 Evaluation and Future Work

This section evaluates ITL and Tempura as vehicle for the specification and simulation of large scale systems. It also indicates future work.

5.1 Evaluation

ITL and Tempura are suitable for the specification and simulation of the EP/3, i.e., for large scale systems. But the following problems were encountered during the specification of the processor:

- In ITL/Tempura only integers, booleans and lists are available as data-types. One would like to have

the data-structures of Pascal like records, reals, files etc. Although these could be simulated using lists, it would ease usability if these data-types were explicitly expressed.

- The units of the processor communicate with each other over the highways. This is modeled with shared variables. The use of special kind of variables like channels is more appropriate. Plus allowing communication actions to be explicitly expressed.

- In normal programming languages like Pascal, a program like `if y>0 then x:=x+1` the value of `x` increases by 1 if `y>0` and `x` doesn't change otherwise. In ITL/Tempura the information that `x` doesn't change has to be coded explicitly: `if y>0 then x:=x+1 else x:=x`, i.e., one has to state explicitly that a variable doesn't change. If one has to update one memory cell, this will be a very costly operation. This is the so-called framing problem.

- Within ITL/Tempura is it hard to model timing constrains like delay or time-out. In the specification of the processor each clock cycle corresponds to a state in ITL, i.e., the length of the interval corresponds to the number of cycles of the processor. We couldn't model properties like: the Alu unit computes the result within 1 clock cycle.

A problem that did not show up in this example is whether to extend Tempura (the executable part of ITL) with high level constructs like the "or" for nondeterminism. The advantage is that more specifications become executable but a disadvantage is that the simulator becomes then very complicated. It may be better to construct a refinement tool that refines a high level specification (written in ITL) into a executable specification (written in Tempura).

5.2 Future work

Future work will consists in extending ITL/Tempura with more data-structures and data-types, and constructs for the description of communicating and timing constrains. Furthermore the framing problem should be solved, this will result in an increase of the speed of the simulator because an update of the memory would then only cost 1 statement instead of as many as there are memory cells. Also we will investigate the use of the PVS[11] system as refinement tool for high level specifications written in ITL to executable specifications written in Tempura.

An issue that isn't addressed in this paper is the correctness of the EP/3, i.e., the proof that certain properties like the pipeline doesn't "overwrite" an unit when there is for some reason "no room" in the pipeline. This kind of properties should be formalized within ITL. With the proof system of ITL these properties then can be proven.

References

1. H. Barrow. Proving the correctness of hardware designs. In *VLSI Design*, pages 64–77, July 1984.
2. R. Boyer and J. Moore. *A Computational Logic*. Academic Press, 1979.
3. J. Coleman. A high speed data-flow processing element and its performance compared to a von Neumann mainframe. In *IEEE 7th Int'l. Parallel Processing Symp*, pages 24–33, Newport Beach, California, 1993.
4. R. Gallimore, D. Coleman, and V. Stavridou. UMIST OBJ: a language for executable program specifications. *Comp. J.*, pages 413–421, 1989.
5. M. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report Report 68, University of Cambridge, 1985.
6. F. Hanna and N. Daeche. Specification and verification using higher order logic. In *Proc. of 7th Intern. Symp. on Computer Hardware Description Languages and Applications*, pages 418–443, 1985.
7. P. Lescanne. Computer experiments with the REVE term rewriting system generator. In *Proc. 10th ACM Symp. on PPL*, 1983.
8. D. May, G. Barrett, and D. Shepherd. Designing chips that work. In C. Hoare and M. Gordon, editors, *Mechanised Reasoning and Hardware Design*, pages 3–18. Prentice Hall, 1992.
9. G. Milne. Circal: a calculus for circuit description. *INTEGRATION. VLSI J.*, pages 121–160, 1983.
10. B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge Univ. Press, Cambridge, UK, 1986.
11. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *International Conference on Automated Deduction*, volume LNAI 607, pages 748–752, Saratoga, NY, June 1992. Springer Verlag.
12. Praxis Systems. *The ELLA User Manual*, 2.0 edition, 1986.
13. A. Sampaio and K. Parsaye-Ghomi. The formal specification and testing of expanded hardware building blocks. In *Proc. Computer Science Conf*, Rolla, MO, 1981.

A Simulator output of first example

The simulator has a more detailed output namely the values of the registers of the various units of the processor before the start of the cycle. The format is as follows of the output is given in table 3.

Table 3: Format of the simulator output

```
Cyc Zlck Zirq Zvld
NR   B    B    B
--------ralu1in ralu1s F3-------F2-------F1-------F0-ralu1a --ralu1b
 7 6 5 4 3 2 1 0    B     3 2 1 0  3 2 1 0  3 2 1 0  3 2 1 0  3 2 1 0
------------rii1 -rii1s ------------------------------ralu2 riiidest
 7 6 5 4 3 2 1 0    B     3 2 1 0  3 2 1 0  3 2 1 0  3 2 1 0    D
---------rmem1in rmem1s ------------------------------rmem1
 7 6 5 4 3 2 1 0    B     3 2 1 0  3 2 1 0  3 2 1 0  3 2 1 0
---------rmem2in rmem2s ------------------------------rmem2 --rmem2b
 7 6 5 4 3 2 1 0    B     3 2 1 0  3 2 1 0  3 2 1 0  3 2 1 0  3 2 1 0
---------rstk1in rstk1s ------------------------------rstk1 --rstk1b
 7 6 5 4 3 2 1 0    B     3 2 1 0  3 2 1 0  3 2 1 0  3 2 1 0  3 2 1 0
```

where NR is a positive integer, B a boolean and D one of the values 0, 1 or 2 (the destination index). The first line describes the registers comprising the Alu1, i.e., the instruction, the active, the flow and the static operand registers. The second line describes the registers comprising the Inst-Iss and the Alu2. The third line describes the registers comprising the Mem-Addr. The fourth line describes the registers comprising the Memory. And the fifth and last line describes the registers comprising the Stack.

The output generated by the simulator of the most interesting cycles is given in the following table. Stage i (in figure 2) corresponds with cycle $20 + i$. Variables $Z, Y1$ and $Y2$ corresponds to respectively Memory locations 00000020, 00000024 and 00000028.

```
21  .   .   .
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
81100020F012F108  T  00000000 00000000 00000000 00000000     0
0000000000000000  .  00000000 00000000 00000000 00000000
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
0000000000000000  .  00000000 00000000 00000000 00000000 00000000

22  T   .   T
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
81100020F012F108  T  00000000 00000000 00000000 00000000     1
81100020F0000012  T  00000000 00000000 00000000 00000000
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
0000000000000000  .  00000000 00000000 00000000 00000000 00000000

23  .   .   T
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
81100020F012F108  .  00000000 00000000 00000000 00000000     0
81100020F1000008  T  00000000 00000000 00000000 00000000
81100020F0000012  T  00000000 00000000 00000000 00000000 00000000
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
Reading from Memory[00000020] the value 0000001F

24  .   .   .
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
81100020F012F108  .  00000000 00000000 00000000 00000000     0
81100020F0000012  .  00000000 00000000 00000000 00000000
81100020F1000008  T  00000000 00000000 00000000 00000000 00000000
81100020F0000012  T  00000000 00000000 00000000 00000000 0000001F
Reading from Memory[00000020] the value 0000001F

25  .   .   .
81100020F0000012  T  00000000 00000000 00000000 00000000 0000001F
81100020F012F108  .  00000000 00000000 00000000 00000000     0
81100020F0000012  .  00000000 00000000 00000000 00000000
81100020F0000012  .  00000000 00000000 00000000 00000000 00000000
81100020F1000008  T  00000000 00000000 00000000 00000000 0000001F

26  .   .   .
81100020F1000008  T  00000000 00000000 00000000 00000000 0000001F
0231F1168F300028  T  00000000 00000000 00000000 00000000     0
81100020F0000012  .  00000000 00000000 00000000 00000000
81100020F0000012  .  00000000 00000000 00000000 00000000 00000000
0000000000000000  .  00000000 00000000 00000000 00000000 00000000

27  .   .   T
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
0321F10C8F200024  T  00000000 00000000 00000000 00000000     0
02310000F1000016  T  00000000 00000000 0000001F 00000000
81100020F0000012  .  00000000 00000000 00000000 00000000 00000000
0000000000000000  .  00000000 00000000 00000000 00000000 00000000

28  .   .   T
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
81100020F012F108  .  00000000 00000000 00000000 00000000     0
03210000F100000C  T  00000000 00000000 0000001F 00000000
02310000F1000016  T  00000000 00000000 0000001F 00000000 0000001F
0000000000000000  .  00000000 00000000 00000000 00000000 00000000

29  .   .   .
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
81100020F012F108  .  00000000 00000000 00000000 00000000     0
81100020F0000012  .  00000000 00000000 00000000 00000000
03210000F100000C  T  00000000 00000000 0000001F 00000000 0000001F
02310000F1000016  T  00000000 00000000 0000001F 00000000 0000001F

30  .   .   .
02310000F1000016  T  00000000 00000000 0000001F 00000000 0000001F
81100020F012F108  .  00000000 00000000 00000000 00000000     0
81100020F0000012  .  00000000 00000000 00000000 00000000
81100020F0000012  .  00000000 00000000 00000000 00000000 00000000
03210000F100000C  T  00000000 00000000 0000001F 00000000 0000001F

31  .   .   .
03210000F100000C  T  00000000 00000000 0000001F 00000000 0000001F
8F300028F11C7F00  T  00000000 00000000 0000001F 00000000     0
81100020F0000012  .  00000000 00000000 00000000 00000000
81100020F0000012  .  00000000 00000000 00000000 00000000 00000000
0000000000000000  .  00000000 00000000 00000000 00000000 00000000

32  .   .   T
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
8F200024F11C0231  T  00000000 00000000 0000001F 00000000     0
8F300028F100001C  T  0000001F 00000000 0000001F 00000000
81100020F0000012  .  00000000 00000000 00000000 00000000 00000000
0000000000000000  .  00000000 00000000 00000000 00000000 00000000

33  .   .   T
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
81100020F012F108  .  00000000 00000000 00000000 00000000     0
8F200024F100001C  T  00000000 FFFFFFE1 0000001F 00000000
8F300028F100001C  T  0000001F 00000000 0000001F 00000000 0000001F
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
Writing to Memory[00000028] the value 0000001F

34  .   .   .
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
81100020F012F108  .  00000000 00000000 00000000 00000000     0
81100020F0000012  .  00000000 00000000 00000000 00000000
8F200024F100001C  T  00000000 FFFFFFE1 0000001F 00000000 FFFFFFE1
8F300028F100001C  T  0000001F 00000000 0000001F 00000000 00000000
Writing to Memory[00000024] the value FFFFFFE1

35  .   .   .
8F300028F100001C  T  0000001F 00000000 0000001F 00000000 00000000
81100020F012F108  .  00000000 00000000 00000000 00000000     0
81100020F0000012  .  00000000 00000000 00000000 00000000
81100020F0000012  .  00000000 00000000 00000000 00000000 00000000
8F200024F100001C  T  00000000 FFFFFFE1 0000001F 00000000 00000000

36  .   .   .
8F200024F100001C  T  00000000 FFFFFFE1 0000001F 00000000 00000000
7F0000000000001F  T  0000001F 00000000 0000001F 00000000     0
81100020F0000012  .  00000000 00000000 00000000 00000000
81100020F0000012  .  00000000 00000000 00000000 00000000 00000000
0000000000000000  .  00000000 00000000 00000000 00000000 00000000
Stop Instruction Executed
Done!  Computation length:  36.  Total Passes:  160.
Total Reductions: 2045605 (1725696 successful).
Cumulative Statistics
Total Cycles = 36 Processing Cycles = 34 during which
Instructions Executed = 6 Mean Cycles / Instruction = 5
```