78

Using Transformations and Verification in Circuit Design

James B. Saxe, Stephen J. Garland,
John V. Guttag, James J. Horning

September 25, 1991

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984 — their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Using Transformations and Verification in Circuit Design

James B. Saxe, Stephen J. Garland, John V. Guttag, James J. Horning

September 25, 1991

John Guttag and Stephen Garland are at the Laboratory for Computer Science of the Massachusetts Institute of Technology. They were supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988, and by the National Science Foundation under grant CCR-8910848.

Authors' Abstract

We show how machine-checked verification can support an approach to circuit design based on transformations. This approach starts with a conceptually simple (but inefficient) initial design and uses a combination of *ad hoc* and algorithmic transformations to produce a design that is more efficient (but more complex).

We present an example in which we start with a simplified CPU design and derive an efficient pipelined form, including circuitry for reverting the effects of partially executed instructions when a successful branch is detected late in the pipeline. The algorithmic stage of our derivation applies a transformation, retiming, that has been proven to preserve functional behavior in the general case. The *ad hoc* stage requires special justification, which we supply in the form of a machine-checked formal verification.

Contents

1 Introduction 1

2 An example circuit 2

2.1 Specification of a minimal processor 3

2.2 An *ad hoc* transformation 6

2.3 An algorithmic transformation 8

3 A manual correctness proof 9

3.1 Overview of the proof 9

3.2 Givens 9

3.2.1 The implementation equations 9

3.2.2 Required properties of components 10

3.3 The goals: specification equations 11

3.4 The proof 12

3.4.1 The level map 12

3.4.2 Validating the specification history 14

4 A machine-checked proof 17

4.1 Declarations 17

4.2 Assertions 18

4.3 Proving the specification equations 20

5 Discussion 23

5.1 Distance from a real circuit 23

5.2 Debugging and maintaining machine-checked proofs 24

5.3 Development, debugging, and generalization of our example proof 25

References 27

1 Introduction

This report presents an integrated approach to designing and verifying circuits. In this approach one starts with a straightforward circuit design, whose semantics are easily understood. This design is then improved—under some measure such as cost, efficiency, or functionality—by a series of transformations. Each transformation may serve either to effect an improvement directly or to enable further transformations.

The transformations used are of two kinds: algorithmic and *ad hoc.* An algorithmic transformation is one that can be mechanically applied to any circuit satisfying some mechanically-checkable preconditions, and that is guaranteed to have a specific effect—or, more commonly, lack of effect—on the functional behavior of any circuit to which it is applicable. When we apply an *ad hoc* transformation, on the other hand, we must produce a specific proof that the behavior of the transformed circuit has some desired relation to that of the original circuit. Doing this is often difficult, and is the main subject of this report.

Three main issues must be considered when doing such proofs:

1. *The cost of performing the proof.* The primary cost of verification is people time, not machine time. Often, the most important factor in determining the cost is how much and what kind of work must be redone when changes are made to the circuit design or to the properties to be proved.

2. *The soundness of the proof.* A proof can increase one's confidence that some property holds; it cannot provide a guarantee. Proofs, like circuits, can have bugs in them.

3. *The relevance of the proof.* For a proof to be relevant it must be based upon axioms that accurately model the circuit about which properties are being proved. Furthermore, the theorem that is proved must accurately model some property that it is desirable for the circuit to have.

We approach the first issue in two ways. First, by making use of algorithmic transformations, we reduce the difficulty of the proofs to be done. Second, we provide a theorem prover called the Larch Prover (LP) [3], that provides assistance in finding proofs. Nevertheless, the cost is still high, in part because it requires someone who understands both the circuit design and the proof technology to develop the proof.

LP also plays a major part in our approach to the soundness issue. In verifying circuits (or programs) the mass of detail that must be managed increases the likelihood of error. Experience indicates that when a proof is not machine checked, it is highly likely that cases are omitted, undocumented assumptions are relied upon, and clerical errors are made [4, 5, 9] Even machine-checked proofs may not be sound; there is always the chance that the program doing the checking is flawed.

The relevance issue is the hardest to deal with. We cannot prove theorems about physical devices such as circuits, only about abstractions of such devices. The relevance of our proofs rests in part on how accurately the abstractions model the circuits. Of course, this problem is not limited to verification. The same problem arises in other kinds of analysis, such as simulation. One way to increase the likely relevance is to develop a suite of tools for specification, simulation, verification, and fabrication, all using a common source language. Driving all tools off the same source files, would increase the likelihood that results from tools such as simulators and verifiers will be about the circuit that actually gets built.[1]

A key technical question to resolve in building such tools is exactly what proof obligation is entailed when verifying an *ad hoc* transformation. This question is particularly complicated in our setting. While using the same language for expressing specifications and implementations facilitates hierarchical decomposition, it raises several questions about defining what it means for an implementation to satisfy its specification. In particular,

1. Must the implementation take the same number of clock cycles as the specification to perform each function?

2. Which of the components appearing in the specification must also appear in the implementation? That is, which parts of the specification are part of its interface to the outside world, and which parts are artifacts of our choice of description—virtual components that need not correspond to any localized part of the implementation?

3. Once one has defined the interface of the specification, what aspects of the I/O behavior of the specification must be preserved in the implementation?

In the next section we give an example of circuit design using *ad hoc* and algorithmic transformations. In Section 3 we show how to formalize the claim that one of the transformations is correct, that is, that the transformed circuit description implements the original circuit description. In carrying out this formalization, we address each of the three questions just listed. We also show how the claim of correctness can be proved. In Section 4 we describe the formal verification of the example transformation using the Larch Prover. In Section 5 we discuss some of the lessons we have learned from our experience with machine-checked formal verification.

2 An example circuit

This section describes the derivation of an optimized circuit design by the application of both *ad hoc* and algorithmic transformations to a simple initial design. We describe the

[1] The work presented in [10] has similar goals. However, it starts from a different design method, one in which specifications and implementations are written in distinct languages. This leads to significant technical differences.

circuits using (a) informal explanatory text, (b) diagrams, and (c) equations suitable for use in formal verification. We also explain how to get from (b) to (c).

2.1 Specification of a minimal processor

Figure 1 shows an initial ("specification") design for a stripped-down computer. In the diagram, lines represent data paths, rectangles represent *registers* (clocked memory elements), and other shapes represent combinational logic elements.

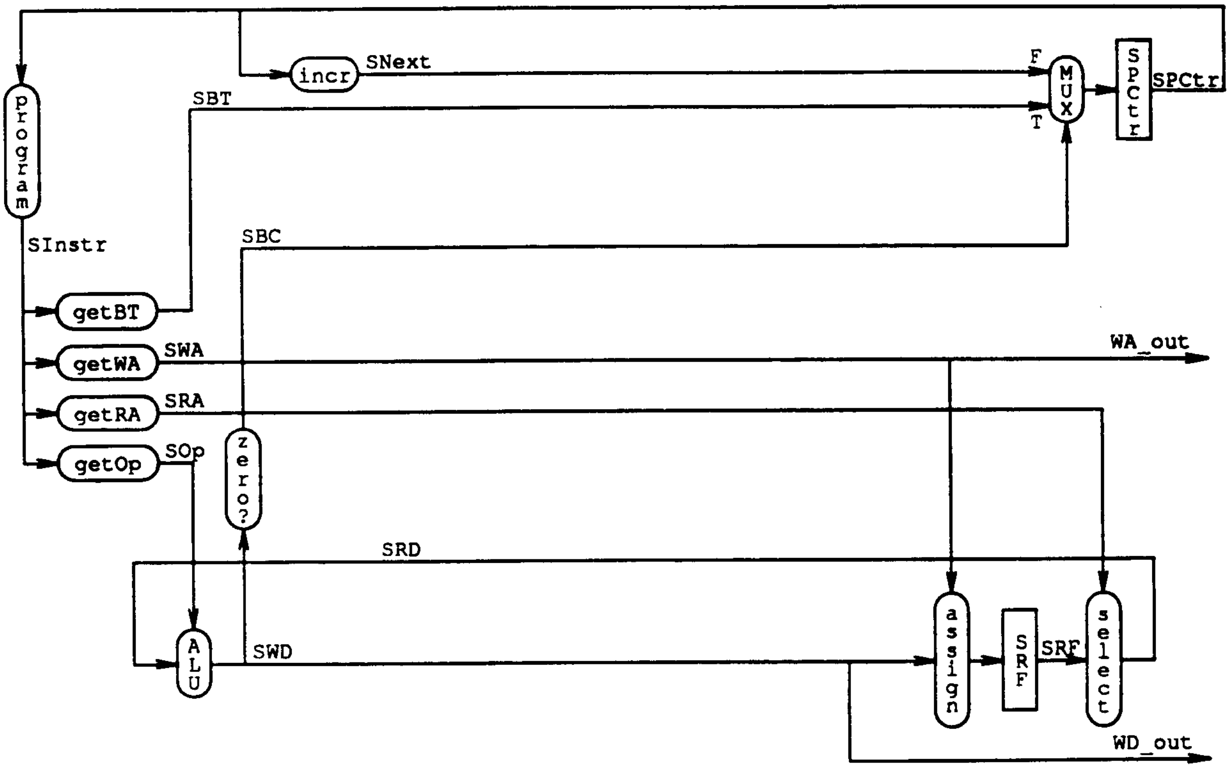

Figure 1. The initial, simple processor design.

Registers are presumed to be driven by a global clock, not shown in the diagram. At each tick of the global clock, each register begins to assert at its output whatever value was present at its input just before the clock tick. The changes in the register outputs then propagate through the combinational logic to produce new register inputs, which will be captured at the next tick of the clock.

An *execution history* of a circuit consists of a sequence of values for each data path in the circuit. By convention, we use the label of a data path to denote the sequence of values that appear on it in this history.[2] Similarly, we use the same label for a register and its output data path. We write `P.t` for the stable value on data path `P` at the end of time step `t`.

Each circuit element places a constraint on execution histories, which can be expressed as an equation relating values on the data paths it connects. Combinational elements constrain values at the same time step, while registers constrain values at consecutive time steps.

In each clock cycle, our example computer executes one instruction of the form "destination register becomes function of source register; branch on zero to target instruction." The next few paragraphs follow the instruction execution process in detail, giving both informal commentary and equations. We use here a subset of the notations used by the Larch Prover (LP) [3], but systematically elide details needed by LP, such as declarations. These details will be discussed in Section 4.

At the beginning of time step `t`, an instruction address, `SPCtr.t` (the "S" is for specification) is clocked out of the program counter (the rectangle in the upper right hand corner of the diagram). We model the program itself as a block of combinational logic (think of it as a ROM) that takes addresses as input and produces instructions as output. Thus the instruction fetched in time step `t` is

```
SInstr.t == program(SPCtr.t)
```

The instruction is then decoded into its four components, namely the read address, the write address, the ALU operation, and the branch target.

```
SRA.t == getRA(SInstr.t)
SWA.t == getWA(SInstr.t)
SOp.t == getOp(SInstr.t)
SBT.t == getBT(SInstr.t)
```

The read address `SRA.t` is used to read a data value `SRD.t` from the register file (modeled as a single large register `SRF`):

```
SRD.t == select(SRF.t, SRA.t)
```

Using the read data and the operation part `SOp.t` of the current instruction, the ALU computes result `SWD.t` (for "write data"):

```
SWD.t == ALU(SOp.t, SRD.t)
```

This result is stored back into the register file at the write address:

```
SRF.(t+1) == assign(SRF.t, SWA.t, SWD.t)
```

Meanwhile, the result is also compared to zero. According to the result `SBC.t` (for "branch control") of this comparison, the new value of the program counter is either set to

[2] Since we will not have occasion, in this report, to reason about two specific execution histories of the same circuit at the same time, this convention introduces no ambiguity.

the branch target or computed by incrementing the current program counter:

```
SBC.t == (SWD.t) = zero
SNext.t == incr(SPCtr.t)
SPCtr.(t+1) == if(SBC.t, SBT.t, SNext.t)
```

where `if(B, E1, E2)` is read "if `B` then `E1` else `E2`." The `if` and "`=`" operators are among the few built-in operators of LP. The others are the boolean operators "`&`" (and), "`|`" (or), "`=>`" (implies), "`<=>`" (iff), `not`, `true`, and `false`. For technical reasons, LP uses two different symbols `=` and `==` to denote comparison for equality. The symbol `==` is used only as the top level connective in equations; syntactically, it binds more loosely than `=` and other infix operators.

The machine communicates to the external world by making all writes to the register file visible at an external interface:

```
WA_out.t == SWA.t
WD_out.t == SWD.t
```

Since these interface signals are common to the "specification" circuit described here and the "implementation" circuit described below, we do not apply the convention of starting their names with `S`.

The speed at which a circuit such as this can run is bounded by the requirement that all the combinational logic outputs must have time to settle to stable values in the interval between one clock tick and the next. In Figure 1 there is a long combinational path that starts at the program counter and goes through the instruction fetch logic, then through the decode logic to select the read address, then through the register file reading logic, then through the ALU, and then through the register write logic. Another long path starts in the same way, going from the program counter through the instruction fetch and decode logic, the register read logic, and the ALU, and then continues through the zero-tester and the branch selection multiplexer. In a physical implementation based directly on Figure 1, one of these two long paths would probably be the critical path limiting the clock speed.

Circuits containing long combinational paths can sometimes be sped up by retiming [7, 8], a technique by which registers are removed from some points in the circuit and inserted at other points according to rules that guarantee preservation of functional behavior. Intuitively, if there are cyclic paths through the circuit on which the registers are distributed unevenly with respect to the combinational logic, retiming can distribute the registers more evenly, thus reducing the maximal combinational delays in the circuit.

Unfortunately, our example circuit in Figure 1 is not a good candidate for improvement by retiming. One of the long combinational paths mentioned above leads from the output of the program counter back to the input of the program counter. The attempt to "distribute the registers more evenly" around a cycle is not likely to be fruitful when there is only one register to distribute.

In the next section, we will transform Figure 1 by *ad hoc* methods to yield a new CPU, shown in Figure 2, that executes the same instruction set. The new CPU is more complicated (and probably slower) than that in Figure 1, but it has a higher ratio of registers to combinational components along its critical feedback cycles, enabling retiming to produce a machine, shown in Figure 3, that is faster than that in Figure 1.

2.2 An *ad hoc* transformation

Roughly speaking, the design of the pipelined machine in Figure 2 is based on the assumption that branches (times for which `SPCtr.(t+1) == SBT.t`) are less common than straight line execution (`SPCtr.(t+1) == incr(SPCtr.t)`). When execution of an instruction results in a branch, the machine doesn't "notice" the branch until several time steps have elapsed, during which it has started executing the next few instructions along the straight line path "in the shadow of the branch." Upon detection of the branch condition, the effects of the shadowed instructions are undone and program execution continues along the branching path. The next few paragraphs describe the derivation of Figure 2 from Figure 1 in more detail.

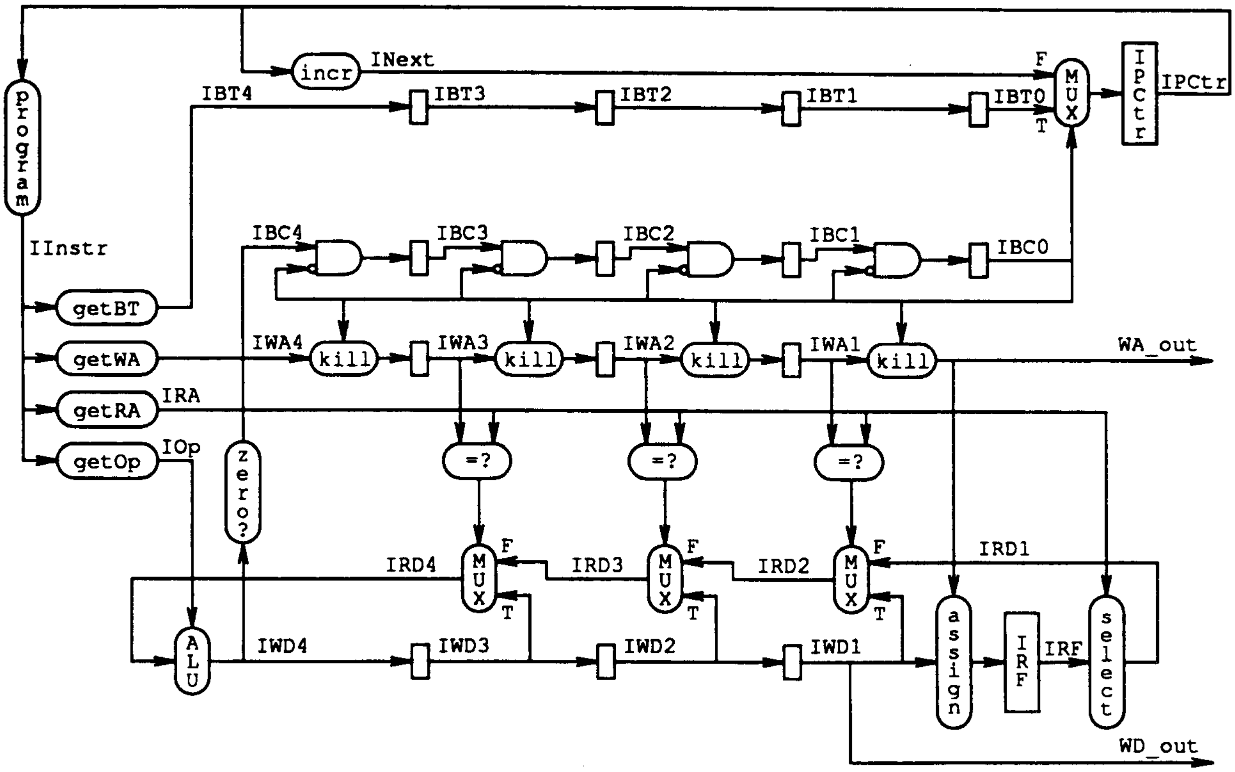

Figure 2. A pipelined processor design, derived from Figure 1 by an *ad hoc* transformation.

We begin by introducing four registers IBC3, IBC2, IBC1, and IBC0, ("I" for implementation) between the zero-comparator and the branch multiplexer, thereby delaying for four clock cycles all decisions about whether or not to branch. In order to have the decision about whether to branch and the appropriate destination of the potential branch arrive at the branch multiplexer at the same time, we also introduce four registers IBT3, IBT2, IBT1, and IBT0 on the path by which branch targets are communicated from the instruction decode logic to the branch multiplexer.

If these were the only changes we made, we would have produced a machine whose instruction set differed from that of Figure 1 by having delayed branching. Whenever the machine executed an instruction that should result in a branch, the next four instructions along the straight-line execution path would be executed before control transferred to the branch target. In order to restore the semantics of Figure 1, we must introduce additional logic to insure that no effects of these (shadowed) instructions become permanent.

In order to postpone commitment of potentially shadowed register file writes, we introduce delaying registers IWA3, IWA2, and IWA1 for the write addresses, and IWD3, IWD2, and IWD1 for the write data destined to the register file. Whenever a branch is taken, write suppression logic changes all queued write addresses to a special addresses, null. Writing to null causes no change to the register file.

Suppose that some instruction writes to a particular address in the register file and the next instruction in the normal execution sequence reads that same address. If both instructions turn out to be permanent (i.e., not shadowed), we want to be sure that the data read by the second instruction is the same as that written by the first, even though that data has not yet been written to the actual register file. For this purpose, we introduce a series of address-comparitors and read data multiplexers (known as "read-bypass" logic) to allow the results of reading the register file to reflect any pending writes.

Finally, in the case of a branch we must suppress not only the register file writes of shadowed instructions, but also any branches which may have been initiated by shadowed instructions. The gates that drive the inputs of registers IBC3, ... , IBC0 serve this purpose.

The interface signals WA_out and WD_out are supplied from the head of the write queue so that shadowed writes will have had their addresses set to null before they become visible at the interface.

Although the machine in Figure 2 is pipelined, the work done for each instruction is very unevenly distributed among the stages of the pipe. If we look for the longest combinational paths in Figure 2, we will see that the minimal clock period for a circuit based directly on Figure 2 would not be much less (and might well be greater, depending on the relative speeds of register assignment and the read-bypass circuitry) than the minimal clock period of a circuit based directly on Figure 1. Since the Figure 2 circuit also loses four cycles to shadowed instructions on every branch, we seem to have bought worse performance at the price of added complexity. In the next section, we show how retiming can be used to reduce the clock period by changing the distribution of registers in the circuit so as to balance the pipeline.

2.3 An algorithmic transformation

Figure 3 shows a circuit that results from retiming Figure 2. If the circuits of Figures 2 and 3 are run side by side with appropriately corresponding initial conditions, then each combinational element of Figure 3 will execute the same sequence of computations as the corresponding element of Figure 2, but shifted later or earlier in time by some number of clock ticks. The small numbers next to the combinational components in Figure 3 indicate their time shifts with respect to the corresponding components of Figure 2, with positive numbers denoting lags and negative numbers denoting leads.

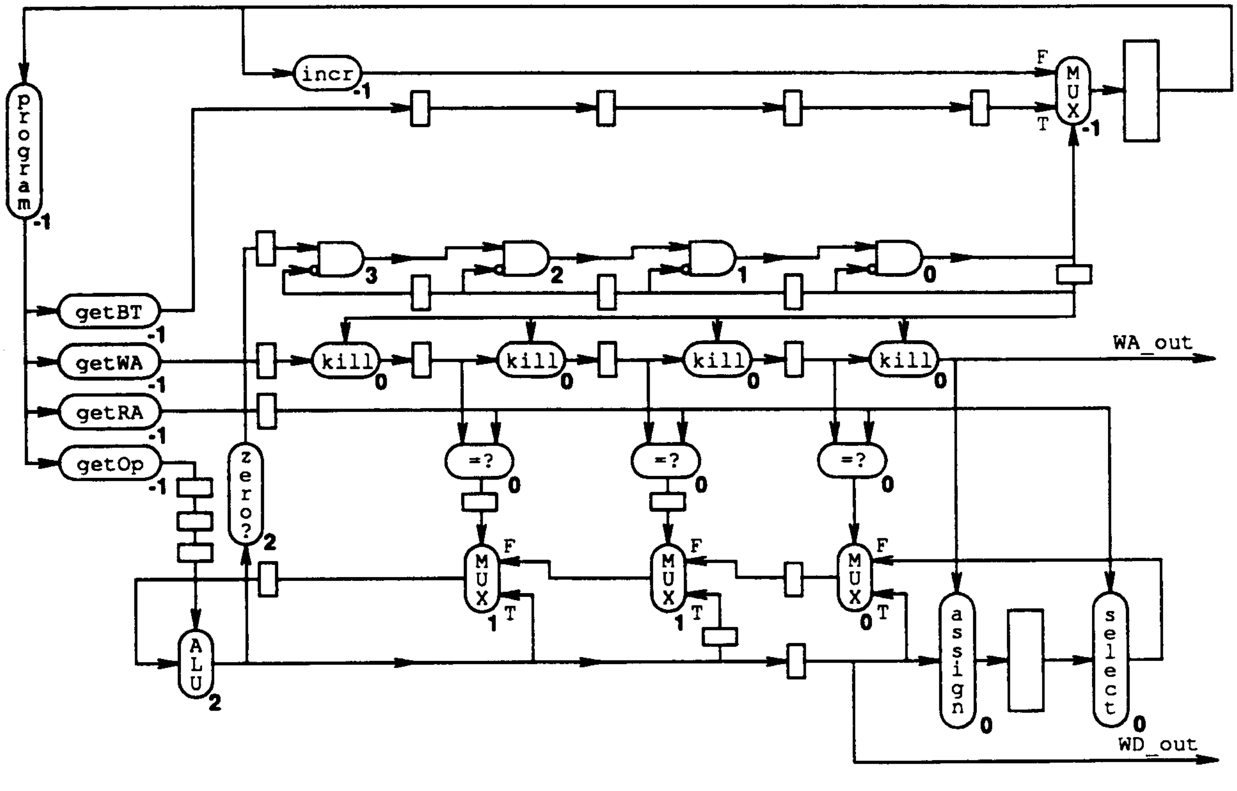

Figure 3. A pipelined processor design, derived from Figure 2 by retiming.

In Figure 3 there is only a relatively small amount of combinational logic on the path from the output of any register to the input of the next register, implying that a circuit built according to Figure 3 can safely run at a much higher clock speed than one based directly on Figure 1. If the frequency of successful branches is sufficiently low, this faster clock speed will more than compensate for the cycles lost to shadowed instructions.

The retiming transformation used to produce Figure 3 from Figure 2 can be performed algorithmically [8] and is guaranteed to preserve circuit behavior [7]. The *ad hoc* transformation used to produce Figure 2 from Figure 1 is another matter. Our description in Section 2.1 of the derivation of Figure 2 constitutes an informal argument for its correctness, but is by no means a rigorous proof. A cautious designer would want more convincing evidence that our *ad hoc* transformation is indeed valid. In the next section, we show how the correctness of such a transformation can be proved.

3 A manual correctness proof

Description I *implements* description S if any observable behavior satisfying I also satisfies S. So to demonstrate that our *implementation* (Figure 2) implements our *specification* (approximately given by Figure 1—see Section 3.3) we must demonstrate that for any legal execution history of Figure 2 there exists a corresponding legal execution history of the specification.

3.1 Overview of the proof

We start by presenting the *givens* for the proof: the *implementation equations*, describing the constraints on an arbitrary legal history of the implementation implied by Figure 2, and the *required properties* of various operators such as **select**, **assign**, and **kill**. Then we state the *goals* to be proved. These are the *specification equations* derived from Figure 1, each of which must be satisfied. The proof proper comprises a *level map*, defining a specification history in terms of the implementation history, and a proof of *satisfaction*, showing that each goal is implied by the givens plus the level map.

3.2 Givens

3.2.1 The implementation equations

Each implementation equation describes the behavior of a register and/or some combinational elements. Identifiers starting with `I` denote the sequences of values that appear on the correspondingly labeled data paths of Figure 2 during some arbitrary execution history; the variable `t` denotes an arbitrary time step, modeled as a natural number; and the infix operator "`.`" returns the element of its left (sequence) argument indexed by its right (time) argument.

```
INext.t     == incr(IPCtr.t)
IInstr.t    == program(IPCtr.t)
IBT4.t      == getBT(IInstr.t)
```

```
IBT3.(t+1)  == IBT4.t
IBT2.(t+1)  == IBT3.t
IBT1.(t+1)  == IBT2.t
IBT0.(t+1)  == IBT1.t
IRA.t       == getRA(IInstr.t)
IRD1.t      == select(IRF.t, IRA.t)
IRD2.t      == if((IWA1.t)=(IRA.t), IWD1.t, IRD1.t)
IRD3.t      == if((IWA2.t)=(IRA.t), IWD2.t, IRD2.t)
IRD4.t      == if((IWA3.t)=(IRA.t), IWD3.t, IRD3.t)
IWA4.t      == getWA(IInstr.t)
IWA3.(t+1)  == kill(IWA4.t, IBC0.t)
IWA2.(t+1)  == kill(IWA3.t, IBC0.t)
IWA1.(t+1)  == kill(IWA2.t, IBC0.t)
IRF.(t+1)   == assign(IRF.t, kill(IWA1.t, IBC0.t), IWD1.t)
IOp.t       == getOp(IInstr.t)
IWD4.t      == ALU(IOp.t, IRD4.t)
IWD3.(t+1)  == IWD4.t
IWD2.(t+1)  == IWD3.t
IWD1.(t+1)  == IWD2.t
IBC4.t      == (IWD4.t) = zero
IBC3.(t+1)  == (IBC4.t) & not(IBC0.t)
IBC2.(t+1)  == (IBC3.t) & not(IBC0.t)
IBC1.(t+1)  == (IBC2.t) & not(IBC0.t)
IBC0.(t+1)  == (IBC1.t) & not(IBC0.t)
IPCtr.(t+1) == if(IBC0.t, IBT0.t, INext.t)
WA_out.t == kill(IWA1.t, IBC0.t)
WD_out.t == IWD1.t
```

3.2.2 Required properties of components

The circuit in Figure 2, relies on certain properties of **assign**, **select**, **kill**, and **getRA**. We must state them explicitly, since we will use them in the proof. Our proof does not depend on the properties of many other operators, such as ALU and incr, and they can be regarded as parameters, or free variables, of the specification and implementation.

Writing to address null must leave the register file unchanged.

```
assign(x, null, y) == x
```

Writing to an address other than null and then reading from the same address must yield the data last written, and writing to one address must not change the contents of any other address.

```
select(assign(x, y1, z), y2) ==
    if((y1 = y2) & not(y2 = null), z, select(x, y2))
```

The kill function used in the write suppression logic is a simple conditional.

```
kill(x, y) == if(y, null, x)
```

No instruction attempts to read from the address null.

```
getRA(program(x)) = null == false
```

If such a read were attempted, it might retrieve the data word from a suppressed write recently queued by a shadowed instruction. Although this assumption seems obvious in retrospect, we overlooked it in our first attempt at a formal proof.

3.3 The goals: specification equations

The specification equations play a different role than the implementation equations. The implementation equations are assumptions of the verification; the specification equations are the theorems that we must prove to show the correctness of the implementation. More precisely, we would like to show that it is possible to choose an execution history for Figure 1 consisting of sequences SNext, SPCtr, SWA, ... such that these equations hold. Such an execution history would constitute an explanation of how Figure 1 could have produced the behavior produced by Figure 2, as observed on WA_out and WD_out. Unfortunately, such an execution history may not exist.

There are several tedious reasons why there might not be an execution history of Figure 1 to explain an arbitrary execution history of Figure 2. As we modified Figure 1 to produce Figure 2, we loosened the specification in a number of ways that were implicit in the informal derivation of Section 2.1 but must be made explicit if we are to state (and prove) a valid theorem about the possible behaviors of the implementation. Being forced to be explicit is one of the benefits of constructing a formal proof.

If the initial contents of the implementation registers are arbitrary, there can be a startup transient of up to three clock cycles (needed to flush the initial contents of the write queue) during which the behavior of the implementation may not reflect anything allowed by the specification.[3] In a real machine, of course, additional circuitry would be used to initialize the machine state. In the interest of brevity, however, we choose the alternative course of simply not observing the implementation during its first three cycles, by systematically using t+3 rather than t in all the specification equations.

The processor drawn in Figure 1 executes an instruction every clock cycle. The processor

[3] The transformation to Figure 3 introduces an additional start-up transient of at most one cycle, arising from the fact that some components have lags of -1 time step with respect to Figure 2, which is one less than the lag (0) of the interface.

in Figure 2, on the other hand, takes five cycles to execute any instruction that actually branches—the four extra cycles being spent on shadowed instructions. In order to allow these extra cycles, we augment the specification by introducing a sequence, `Stalled`, of booleans indicating cycles on which the system is "stalled," and modify the equations describing the register file and program counter to indicate that the register file's contents are unchanged during such cycles.

```
SRF.(t+4)    == if(Stalled.(t+3),
                   SRF.(t+3), assign(SRF.(t+3), SWA.(t+3), SWD.(t+3)))
SPCtr.(t+4)  == if(Stalled.(t+3),
                   SPCtr.(t+3), if(SBC.(t+3), SBT.(t+3), SNext.(t+3)))
```

The specification of behavior at the interface must also be modified to account for values that may be seen during stalled cycles. Specifically, the write address to the register file is required to be `null` (so that it won't be seen at the interface as a genuine write), and the write data is irrelevant, leading to the interface equations:

```
WA_out.(t+3) == if(Stalled.(t+3), null, SWD.(t+3))
((WD_out.(t+3)) = (SWD.(t+3))) | (Stalled.(t+3)) == true
```

The remaining specification equations simply describe constraints imposed by registers and combinational elements in Figure 1.

```
SNext.(t+3)  == incr(SPCtr.(t+3))
SInstr.(t+3) == program(SPCtr.(t+3))
SBT.(t+3)    == getBT(SInstr.(t+3))
SRA.(t+3)    == getRA(SInstr.(t+3))
SRD.(t+3)    == select(SRF.(t+3), SRA.(t+3))
SWA.(t+3)    == getWA(SInstr.(t+3))
SOp.(t+3)    == getOp(SInstr.(t+3))
SWD.(t+3)    == ALU(SOp.(t+3), SRD.(t+3))
SBC.(t+3)    == (SWD.(t+3)) = zero
```

3.4 The proof

3.4.1 The level map

We next construct a specification history to account for the observed behavior of an arbitrary implementation history. Such a history can be given by a *level map*—a set of equations defining a specification history in terms of a given implementation history.

Note that the definitions of the sequences, `SRF`, `SPCtr`, etc., comprising the specification history must be just that—definitions. If a "level map" employs multiple or circular definitions of some component of the specification history, then it may implicitly introduce new constraints on the implementation history, or may even be outright inconsistent. Technically speaking, the theory obtained by adding the level map equations to the previously assumed theory (consisting of the implementation equations, the required operator properties, and the standard rules for arithmetic, booleans, and sequences) must be a *conservative extension* of the previously assumed theory. That is, any theorem that doesn't involve the newly introduced symbols, SRF, SPCtr, etc., and that is provable in the new system must be provable in the old system.

```
SRF.t           == IRF.t
SPCtr.(t+3)     == if(IBC0.t, IBT0.t,
                       if(IBC1.t, IBT1.t,
                          if(IBC2.t, IBT2.t,
                             if(IBC3.t, IBT3.t,
                                IPCtr.t
                   ) ) ) )
SNext.t         == incr(SPCtr.t)
SInstr.t        == program(SPCtr.t)
SBT.t           == getBT(SInstr.t)
SWA.t           == getWA(SInstr.t)
SRA.t           == getRA(SInstr.t)
SOp.t           == getOp(SInstr.t)
SRD.t           == select(IRF.t, SRA.t)
SWD.t           == ALU(SOp.t, SRD.t)
SBC.t           == (SWD.t) = zero
Stalled.(t+3)   == (IBC0.t) | (IBC1.t) | (IBC2.t) | (IBC3.t)
```

We insure the conservativeness of our level map by following a certain syntactic style. For each component sequence of the specification history, there is exactly one defining equation, with that component on its left hand side and with only previously defined terms (possibly including previously defined specification history components) on the right hand side. To illustrate these ideas in a simpler context, consider a proof in which we have introduced integers a and b but have not yet used the name c. A legitimate next step would be to say, (1) “Let c == a + b.” It would not be legitimate to say (2) “Let c == a + a and also let c == b + b,” or (3) “Let c == a + c - b.” We know by the syntactic form of (1)—namely, “Let c == ⟨*expression containing neither* c *nor any free variable*⟩”—that it is conservative. Multiple definitions, such as (2), or a circular definition, such as (3), may implicitly introduce unwarranted assumptions about a and b (in these examples, the assumption a == b). As a final example consider (4) “Let a == b + c + 1.” While (4) is technically conservative, it does not have the syntactic form of a definition of c. The conservativeness of (4) cannot be deduced by mere examination of its form, but depends on properties of integer addition; if the universe of discourse were the natural numbers rather than the integers, (4) would embody the assumption a > b.[4]

[4] A fine point: Since the operators “.” and “+” appear on the left hand sides of the “definitions” in our

There is no set method for deriving a level map. We arrived at this one by understanding why the system in Figure 2 works and applying brain power. Note that an ill-chosen level map may lead to failure to verify a correct implementation, but, so long as it is conservative, cannot lead to "verification" of an incorrect implementation.

3.4.2 Validating the specification history

All that remains is to prove that the execution history given by the level map satisfies the specification equations. There are thirteen specification equations to prove. Nine of them are simply instances of level map equations. The other four do not follow quite so trivially, but can be proved by case analysis on the values of some of the boolean signals in the specification and implementation equations. We give here a sampling of the sort of reasoning used. The non-trivial goals are:

```
SRF.(t+4) ==
  if(Stalled.(t+3), SRF.(t+3), assign(SRF.(t+3), SWA.(t+3), SWD.(t+3)))
SPCtr.(t+4) ==
  if(Stalled.(t+3), SPCtr.(t+3), if(SBC.(t+3), SBT.(t+3), SNext.(t+3)))
WA_out.(t+3) == if(Stalled.(t+3), null, SWA.(t+3))
(Stalled.(t+3)) | ((WD_out.(t+3)) = (SWD.(t+3))) == true
```

We may proceed by considering first the case in which time step t+3 is a non-stalling cycle and then the case of a stalling cycle.

We consider first the non-stalling case, in which we assume

```
Stalled.(t+3) == false
```

This assumption lets us reduce our four goals to

```
SRF.(t+4) == assign(SRF.(t+3), SWA.(t+3), SWD.(t+3))
SPCtr.(t+4) == if(SBC.(t+3), SBT.(t+3), SNext.(t+3))
WA_out.(t+3) == SWA.(t+3)
WD_out.(t+3) == SWD.(t+3)
```

It also allows us to make some forward inferences. By expanding the definition of Stalled in the level map, we have

```
IBC0.t == false
IBC1.t == false
IBC2.t == false
IBC3.t == false
```

level map, the legitimacy of these definitions technically depends on certain properties of sequences and of natural numbers not explicitly stated above. Specifically, (1) every function from indices (natural numbers) to scalars is the characteristic function of some sequence, and (2) addition of the constant 3 to distinct augends gives distinct results.

Expanding the definition of SPCtr then gives

```
SPCtr.(t+3) == IPCtr.t
```

from which we obtain[5]

```
SNext.(t+3) == incr(SPCtr.(t+3)) == incr(IPCtr.t) == INext.t
```

and similarly

```
SInstr.(t+3) == IInstr.t
SBT.(t+3)    == IBT4.t
SWA.(t+3)    == IWA4.t
SRA.(t+3)    == IRA.t
SOp.(t+3)    == IOp.t
```

The rest of the specification state components for time t+3 also map (in this non-stalled case) directly to particular implementation state components for time t, but the calculations necessary to confirm this take a bit more work.

We start by considering the implementation equations for the registers in the branch condition pipeline IBC0, ..., IBC3. Note that

```
IBC0.(t+1) == (IBC1.t) & not(IBC0.t)
           == false & not(false)
           == false
```

Similarly, we obtain

```
IBC0.(t+2) == IBC1.(t+1) == false
IBC0.(t+3) == IBC1.(t+2) == IBC2.(t+1) == false
```

Since IBC0 is false at steps t through t+3, we find that the write addresses in the pipeline at time t propagate to the register file assignment logic without change:

```
WA_out.t       == kill(IWA1.t, IBC0.t)
               == IWA1.t
WA_out.(t+1) == IWA1.(t+1) == IWA2.t
WA_out.(t+2) == IWA1.(t+2) == IWA2.(t+1) == IWA3.t
WA_out.(t+3) == IWA1.(t+3) == IWA2.(t+2) == IWA3.(t+1) == IWA4.t
```

Since we already had

```
SWA.(t+3) == IWA4.t
```

we obtain one of our four goals for the not-stalled case:

[5] The shorthand used here, in which we write "E1 == E2 == E3 == E4" to indicate the line of reasoning "E1 == E2; E2 == E3; E3 == E4; therefore E1 == E4" is not used by LP, which does not produce—and cannot parse—any "equation" containing more than one "==" sign.

```
WA_out.(t+3) == IWA4.t == SWA.(t+3)
```

For the remaining three goals, we next consider the level map for the register file and its relation to the read-bypass equations.

If we repeatedly apply the implementation equation for the implementation register file IRF to expand the level map definition for the specification register file SRF, we get

```
SRF.(t+3) == IRF.(t+3)
          == assign(IRF.(t+2),
                    kill(IWA1.(t+2), IBCO.(t+2)),
                    IWD1.(t+2))
          == assign(IRF.(t+2), IWA1.(t+2), IWD2.(t+1))
          == assign(IRF.(t+2), IWA3.t, IWD3.t)
          == ...
          == assign(assign(assign(IRF.t, IWA1.t, IWD1.t),
                               IWA2.t, IWD2.t),
                    IWA3.t, IWD3.t)
```

If we start expanding the definition of SRD for time step t+3, we get

```
SRD.(t+3) == select(SRF.(t+3), SRA.(t+3))
          == select(assign(IRF.(t+2), IWA3.t, IWD3.t) IRA.t)
          == if((IRA.t = IWA3.t) & not((IRA.t) = null),
                IWD3.t, select(IRF.(t+2), IRA.t))
```

Since

```
IRA.t == getRA(IInstr.t) == getRA(program(IPCtr.t))
```

we know

```
(IRA.t) = null == false
```

So the above simplifies further to

```
SRD.(t+3) == if((IRA.t = IWA3.t), IWD3.t, select(IRF.(t+2), IRA.t))
```

Similarly expanding the subexpression `select(IRF.(t+2), IRA.t)`, we eventually get

```
SRD.(t+3) == if((IRA.t = IWA3.t), IWD3.t,
                if((IRA.t = IWA2.t), IWD2.t,
                   if((IRA.t = IWA1.t), IWD1.t,
                      select(IRF.t, IRA.t))))
```

The right hand side of this equation is exactly what we get by applying the implementation equations for the combinational logic that computes IRD4.t. So we have

```
SRD.(t+3) == IRD4.t
```

which we can use to obtain

```
SWD.(t+3) == ALU(SOp.(t+3), SRD.(t+3))
          == ALU(IOp.t, IRD4.t)
          == IWD4.t
```

The equations for the write data pipeline give us

```
IWD1.(t+3) == IWD2.(t+2) == IWD3.(t+1) == IWD4.t
```

So we have

```
WD_out.(t+3) == IWD1.(t+3) == IWD4.t == SWD.(t+3)
```

and another of our four goals is achieved.

The remaining two goals can be proved in a similar fashion. We would then turn to the stalling case, which turns out to need nested case analysis on the position of the first queued branch in the pipeline, but is basically similar. It should be clear by now why such manual proofs are extremely tedious and error-prone. Much of the detailed analysis, formula manipulation, and checking can and should be automated. The next section discusses one way to do that.

4 A machine-checked proof

In this section, we present a slightly abridged script for a formal verification of the Figure 2 circuit using the Larch Prover (LP). While we include brief explanations of the LP commands used in this example, this section is not intended as a tutorial on LP. Readers who want to learn more about LP are referred to [3].

4.1 Declarations

In LP, all user identifiers must be declared, with signatures, before use. Here are some examples of declarations used in our formal verification.

```
declare sort Nat              % The sort of natural numbers
declare variable t: Nat       % Used for time steps
declare sorts
  RAddr,                      % The sort of register addresses
  IAddr,                      % Instruction addresses
  Data,                       % Data words
  Instr,                      % Instructions
  RAddr_seq,                  % Sequences of register addresses
  IAddr_seq,                  % Sequences of instruction addresses
```

```
    Data_seq,                  % Sequences of data words
    Instr_seq                  % Sequences of instructions
    ..                         % .. marks the end of a multi-line LP command.
  declare operators
    .: RAddr_seq, Nat -> RAddr  % The name "." is overloaded to denote several
    .: IAddr_seq, Nat -> IAddr  % operators for indexing sequences of different
    .: Data_seq, Nat -> Data    % sorts.  LP determines which one is meant
    .: Instr_seq, Nat -> Instr  % on the basis of the sorts of the arguments.
    ..                          % LP has no built-in rules about sequences.
  declare variable
    xInstr: Instr               % xInstr is a variable denoting an arbitrary
    ..                          % instruction.
  declare operators
    program: IAddr -> Instr     % Instruction fetch.  Takes an IAddr as
                                %   argument and returns an Instr
    incr: IAddr -> IAddr        % Increment.
    ..
  declare operators
    IPCtr, INext: -> IAddr_seq  % Constants in LP are modeled as nullary
    IInstr: -> Instr_seq        % operators.  Our proof starts by picking
    ..                          % an arbitrary "fixed" execution history.
  declare operators
    0, 1, 2, 3, 4, 5, 6, 7: -> Nat
    zero: -> Data
    ..
```

It is important to understand the distinction between variables (such as **t** and **xInstr**) and constants (modeled as nullary operators, such a **IPCtr** and **zero**). An occurrence of a variable in an LP equation indicates that the equation holds for all values of that variable's sort. An occurrence of a constant indicates only that the equation holds for the particular value denoted by the constant.

4.2 Assertions

Having declared the necessary identifiers, we then assert the implementation equations, required operator properties, and level map definition, as described in Section 3.[6] The proof also uses some simple facts about arithmetic. Normally these would come from a standard library but for the sake of completeness, we give here all those that are actually used.

[6] We must admit here that the conservativeness of the level map is not machine-checked. While we know how to remedy this deficiency, to do so with the current version of LP would require a style of proof rather more cumbersome, both to use and to explain, than the one we use in this report. Instead we have chosen to rely on the syntactic conventions described in Section 3.4.1.

```
set order left-to-right

assert        % definitions of numerals
  2 == 1 + 1
  3 == 2 + 1
  4 == 3 + 1
  5 == 4 + 1
  6 == 5 + 1
  7 == 6 + 1
  ..
assert ac +   % "+" is associative and commutative

assert        % implementation equations
  INext.t == incr(IPCtr.t)
  % etc.,  as given in Section 3.2.1
  WD_out.t == IWD1.t
  ..

assert        % required properties of operators (Section 3.2.2)
  assign(xRFile, null, xData) == xRFile
  select(assign(xRFile, xRAddr, xData),yRAddr) ==
    if( (xRAddr = yRAddr) & not(yRAddr = null),
        xData,
        select(xRFile, yRAddr)
      )
  kill(xRAddr, xBool) == if(xBool, null, xRAddr)
  getRA(program(xIAddr)) = null == false
  ..

assert % level map
  SRF.t         == IRF.t
  SPCtr.(t+3)   == if(IBC0.t, IBT0.t,
                       if(IBC1.t, IBT1.t,
                          if(IBC2.t, IBT2.t,
                             if(IBC3.t, IBT3.t,
                                IPCtr.t
                      )  )  )  )
  % etc., as given in Section 3.4.1)
  Stalled.(t+3) == (IBC0.t) | (IBC1.t) | (IBC2.t) | (IBC3.t)
  ..
```

Much of LP's deductive system is based upon orienting equations into rewrite rules. The command "`set order left-to-right`" causes LP to produce rules that rewrite expressions matching the left hand sides of the equations above into corresponding instances of the right hand sides, rather than the other way around. While the left-to-right ordering rule works for our example proof, such user-chosen orderings are potentially dangerous because they may lead to non-terminating rewriting systems. For this reason, LP includes automated strategies for producing rewriting systems that are guaranteed to terminate [1, 3]. One such strategy can be caused to orient all the equations above in the left-to-right direction (thereby certifying the termination of the resulting rewriting system), but only if the user supplies appropriate hints. Further discussion of such ordering hints is beyond the scope of this report; the reader is referred to [3] for a thorough discussion of LP's ordering facilities and the theory behind them.

4.3 Proving the specification equations

All that remains is to prove the modified specification equations from Section 3.3. LP gets the easy ones without any user guidance:

```
prove SNext.(t+3) == incr(SPCtr.(t+3))
qed
prove SInstr.(t+3) == program(SPCtr.(t+3))
qed
prove SBT.(t+3) == getBT(SInstr.(t+3))
qed
prove SRA.(t+3) == getRA(SInstr.(t+3))
qed
prove SRD.(t+3) == select(SRF.(t+3), SRA.(t+3))
qed
prove ((SWA.(t+3)) = getWA(SInstr.(t+3)))
qed
prove SOp.(t+3) == getOp(SInstr.(t+3))
qed
prove SWD.(t+3) == ALU(SOp.(t+3), SRD.(t+3))
qed
prove SBC.(t+3) == (SWD.(t+3)) = zero
qed
```

The qed command causes LP to check whether there are any outstanding unproven conjectures on its stack. If there are, LP displays an error message, and, if the failing qed came from a file, returns to interactive mode.

For some of the proofs LP requires a bit of assistance from the user. There are a variety of ways in which this can be provided. For this example, the only guidance needed is for the user to suggest appropriate uses of case analysis. The necessary commands exhibit several features of LP, which we will explain presently.

```
set box-checking on  % This command is explained on the next page.
prove
  SPCtr.(t+4) ==
    if(Stalled.(t+3), SPCtr.(t+3), if(SBC.(t+3), SBT.(t+3), SNext.(t+3)))
  ..
  resume by case IBC0.t
  <> 2 subgoals for proof by cases
    [] case IBC0 . tc
    resume by case IBC1.tc
    <> 2 subgoals for proof by cases
      [] case IBC1 . tc
      resume by case IBC2.tc
      <> 2 subgoals for proof by cases
        [] case IBC2 . tc
        [] case not(IBC2 . tc)
      [] case not(IBC1 . tc)
    [] case not(IBC0 . tc)
  [] conjecture
qed
```

The command `resume by case E` (where the *case expression* `E` is of sort `boolean`) causes LP to replace the goal of proving the current conjecture with two subgoals: proving the conjecture under the assumption that the case expression is true and proving the conjecture under the assumption that the case expression is false.[7] These assumptions are known as "case hypotheses" in their respective arms of the proof.

A technical issue arises when the case expression contains a variable. Since variables are implicitly universally quantified, the cases

```
IBC0.t == true   % for all t
```

and

```
IBC0.t == false  % for all t
```

do not cover all possible situations. The data path `IBC0` might carry the value `true` on some time steps and `false` on others. In order to make the case analysis sound, LP replaces each variable in the case expression with a so-far-unused constant (`tc` in this case). All occurrences of these *case variables* in the conjecture are likewise replaced with the corresponding constants. The soundness of this procedure follows from the observation that, if some property can be proven for a particular time step of `tc` about which no special information is given, then that property must hold at every time step. Since, within any arm of the case analysis, LP is working on a conjecture in which the variable `t` has been replaced by the

[7] LP also supports proofs by multi-way case analyses, where the set of cases examined must be proven to be exhaustive. We do not happen to use this facility in this report.

constant tc, nested **resume by case** commands use case expressions that likewise contain tc in place of t.

The lines beginning with boxes ([]) and diamonds (<>) are annotations to the proof. When LP is run with its "box-checking" mode turned on, and is reading commands from a file, it expects to see a line beginning with a diamond (and an appropriate number) every time it introduces one or more subgoals, and it expects to see a line beginning with a box every time a subgoal is discharged. Otherwise, it displays an error message and returns to interactive mode. This feature, like the qed command described above, is quite useful when one is replaying a slightly modified version of an old command file, since it notifies the user of the first deviation of a proof from its expected course, rather than letting LP proceed to apply additional commands in an inappropriate context. Note that if the box and diamond lines were removed from the proof script above, it would be impossible to tell (even using indentation as a hint) whether the cases analysis on IBC1.tc was intended to be applied within the case IBC0.tc == **true** or within the case IBC0.tc == **false**.

The remaining pieces of the proof proceed similarly to those above. Here they are, with the box and diamond lines omitted:

```
set box-checking off
prove
  SRF.(t+4) ==
    if(Stalled.(t+3), SRF.(t+3), assign(SRF.(t+3), SWA.(t+3), SWD.(t+3)))
  ..
  resume by case IBC0.(t)
    resume by case IBC1.tc
      resume by case IBC2.tc
        resume by case IBC3.tc
qed
prove WA_out.(t+3) == if(Stalled.(t+3), null, SWA.(t+3))
  resume by case IBC0.(t)
    resume by case IBC1.tc
      resume by case IBC2.tc
qed
prove
  (Stalled.(t+3)) | ((WD_out.(t+3)) = (SWD.(t+3))) % "== true" is implicit.
  ..
  resume by case IBC0.(t)
    resume by case IBC1.tc
      resume by case IBC2.tc
        resume by case IBC3.tc
qed
```

The preceding theorems cover only safety properties of Figure 2, showing that every step of its execution (after the start-up transient) corresponds either to a step of Figure 1's execution or to a stall. It would also be useful to know that the implementation cannot simulate stalling forever. The following commands suffice to prove that, once the start-up transient is over. the circuit can never stall for more than four consecutive cycles—the time needed to clear the pipe of shadowed instructions after a successful branch.

```
prove
  not( (Stalled.(t+3)) & (Stalled.(t+4)) & (Stalled.(t+5)) &
       (Stalled.(t+6)) & (Stalled.(t+7))
     )
  ..
  resume by case IBC0.(t)
    resume by case IBC1.tc
      resume by case IBC2.tc
        resume by case IBC3.tc
qed
```

5 Discussion

5.1 Distance from a real circuit

The example presented in this report is at a level of abstraction somewhat removed from actual circuits. The details we have abstracted away fall into two broad classes:

1. We have been intentionally silent about some aspects of the circuit design, including
   - The kinds of data carried on internal data paths; for example, how big are the instruction and register address spaces?
   - The semantics of the operators; for example, how many ALU operation codes are there and what does each ALU operation code mean?
   - The encodings of the data presented at the interface; for example, how is a data word encoded in bits.
2. We have said nothing about the physical realization of the circuit. In particular, we have not discussed what design rules must be obeyed in order to guarantee that the physical circuit behaves consistently with the equations describing it.

We have avoided the first class of details by treating many aspects of the design as parameters. In effect, we have designed a class of circuits. In a real application, the specification writer would at some point instantiate the design parameters by requiring a particular word size, ALU semantics, etc. The soundness of our algorithmic transformations and our verification of the *ad hoc* transformation do not depend upon how these parameters are instantiated. They are independent not only of the implementation of, say, the ALU, but even of its specification. It is only necessary that Figure 3 be instantiated with the *same* word size, ALU specification, etc. as Figure 1.

In contrast, the second class of details is not easily dealt with within our framework. Recall that each implementation equation in Section 3.2.1 corresponds to one or a few components in the Figure 2 diagram. We assume that if the physical components are properly fabricated and connected to each other in accordance with the diagram, then the resulting system will obey the equations. In general, however, the signals at the inputs and outputs of a physical component (for example, an AND gate) can be guaranteed to satisfy the corresponding equation (`out.t == (in1.t) & (in2.t)`) only if the system as a whole obeys certain design rules. For example, the outputs of different components must not be shorted together, and the clock period must be long enough to allow all combinational outputs to achieve stable values between successive ticks.

5.2 Debugging and maintaining machine-checked proofs

In our experience with mechanical proof checking, a few things stand out:

- Almost every "theorem" we try to prove isn't.
- If we don't understand why a theorem is valid, there is very little chance of discovering a mechanical proof.
- Even if we *do* understand why a theorem is valid, the first proofs we attempt are likely to be flawed.

Consequently, we want our proof checker to have a number of properties:

- It should assist in the interactive development of proofs.
- It should quickly detect invalid proof steps, and provide feedback that will help the user discover the error.
- To the greatest extent possible, it should "fail fast" on non-theorems, that is, stop and complain rather than automatically indulging in complex or time-consuming heuristics. There's a very good chance that simple proofs aren't working because the conjecture just isn't true.
- It should make it easy to formulate proof scripts that will be robust. It is frequently necessary to replay the proof of a theorem after small changes in axioms, the proofs

of previous theorems, or the statement of the theorem itself. These should not require development of a completely new proof script.

- During replay, it should monitor the correspondence between the script and the progress of the proof, and stop as soon as a divergence is detected. This realization led us to the implementation of the <>, [], and qed commands described in Section 4.3.

5.3 Development, debugging, and generalization of our example proof

The circuits in Figures 1, 2, and 3 were originally designed by Jim Saxe to illustrate the application of successive transformations in the systematic design of high-speed digital circuits by deriving Figure 3 from Figure 1 by way of Figure 2. Although he was convinced of the "equivalence" of Figures 1 and 2, because he had carefully derived the latter from the former, he wasn't sure how to prove it formally. So he solicited help from Leslie Lamport, who, in the course of an afternoon, generated both a formal statement of the theorem to be proved and a sketch of a formal proof.[8] Leslie suggested that greater confidence in the proof could be gained by using a term rewriting system some of us had been building, Reve [2, 6], to perform the symbol manipulation needed to complete his proof sketch.

It took us about three man-weeks to come up with our first machine-checked proof based on the sketch we were given. We had to fix a number of small errors that are practically a litany of what goes wrong in hand proofs: there was a missing parenthesis in one of the formulas; there was an off-by-one error in a loop; the restriction that no instruction may attempt to read from the address `null` was left unstated; one case (branch taken) was ignored in the analysis; the proof used an invariant that we had to strengthen. We also spent time debugging our axiomatization of the circuit. In some places the prover was very helpful in doing this; for example, an inconsistent specification of the `assign` and `select` operators was detected by the prover. In other places, it was less helpful.

It is interesting that all the problems we found were errors in the hand proof or in our translation of the hand proof to Reve notation, and not in the machine design. It really was the proof, and not the design, that we debugged. It is important to note, however, that in constructing the proof we were forced to think carefully about the specification of the circuit, making explicit a variety of assumptions upon which the correctness of the design depends. For example, the specification used in our proof included the possibility of a start-up transient and the possibility of occasional stalls. If the circuit described by Figure 1 were to be used in a context where such behaviors were not acceptable, then a physical implementation based on Figure 3 would not be acceptable. Making such assumptions explicit is an important benefit of machine-checked verification.

[8] Both the original statement of the theorem and the structure of the proof differ significantly from the ones given in this report, although they are of similar complexity.

A key feature of our formalization of start-up transients and stalls was its nondeterminism. For example, the specification equations in Section 3.3 do not, for any given time step `t`, determine `Stalled.(t+1)` as a function of `SPctr.t` and `SRF.t`. It was by making stalling on any cycle optional, rather than mandatory or forbidden, that we allowed physical implementations based on Figures 2 or 3 without forbidding an implementation based directly of Figure 1.

Figure 2 was produced by adding four stages of pipelining to Figure 1, one of which is partially optimized away. One would expect similar correctness proofs to work for circuits derived by adding more or fewer stages of pipelining. Our experiments in this direction gave precisely the results we expected: It was an entirely straightforward, albeit tedious, task to modify all the circuit descriptions, conjectures, and proofs given above so that they would work with different numbers of stages; but the time and space requirements of such proofs rose exponentially with the number of stages. Under a reformulation of the circuit descriptions using single LP symbols to represent arrays of circuit components, we were able to prove, by induction on the depth, the correctness of versions of Figure 2 with arbitrary-depth pipelines.

When we produced our first machine-checked verification of Figure 2, based on Leslie Lamport's proof sketch, our tools were not as good as they are now. In fact, much of the early evolution of LP from Reve was motivated by the example we have presented here. Also, we had very little experience with mechanical proofs of this nature. However, despite improvements in our tools and our increased experience, it still seems harder to construct machine-checked proofs than hand proofs, not least because the machine has an annoying habit of rejecting plausible but erroneous arguments.

Acknowledgments

We thank Urban Engberg and Cynthia Hibbard for helpful comments on earlier drafts of this report.

References

[1] David Detlefs and Randy Forgaard, "A procedure for automatically proving the termination of a set of rewrite rules," *Proceedings of the First International Conference on Rewriting Techniques and Applications*, Dijon, France, *Lecture Notes in Computer Science* 202, Springer-Verlag, May 1985, pp. 255–270.

[2] Randy Forgaard and John V. Guttag, "REVE: A term rewriting system generator with a failure-resistant Knuth-Bendix," *Proceedings of a Workshop on Term Rewriting* (ed. by D. Kapur and D. Musser), April 1984, pp. 5–31.

[3] Stephen J. Garland and John V. Guttag, "A guide to LP, the Larch Prover," in preparation.

[4] Stephen J. Garland, John V. Guttag, and James J. Horning, "Debugging Larch Shared Language specifications," *IEEE Transactions on Software Engineering*, Vol. 16, No. 9, September 1990, pp 1044–1057.

[5] Stephen J. Garland, John V. Guttag, and Jørgen Staunstrup, "Verification of VLSI circuits using LP," *Proceedings of the IFIP WG 10.2 Conference on the Fusion of Hardware Design and Verification*, North Holland, 1988, pp. 329–345.

[6] Pierre Lescanne, "REVE: a rewrite rule laboratory," *Proceedings of the 8th International Conference on Automated Deduction*, Oxford, England, *Lecture Notes in Computer Science* 230, Springer-Verlag, July 1986, pp. 695–696.

[7] Charles E. Leiserson and James B. Saxe, "Optimizing synchronous systems," *Journal of VLSI and Computer Systems*, Vol. 1, No. 1, Spring 1983, pp. 41–67.

[8] Charles E. Leiserson and James B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, Vol. 6, No. 1, 1991, pp. 5–35.

[9] John Rushby and Friedrich von Henke, "Formal verification of the interactive convergence clock synchronization algorithm using EHDM," SRI International report SRI-CSL-89-3, February, 1989.

[10] Jørgen Staunstrup and Mark Greenstreet, "Synchronized transitions," in Jørgen Staunstrup, ed., *Formal Methods for VLSI Design*, North-Holland/Elsevier, 1990, pp. 71–129.

SRC Research Reports

The following pages list the titles of our research reports. You can access the list of abstracts on gatekeeper.pa.dec.com via anonymous ftp. The pathname of the list is /usr/spool/ftppublic/pub/DEC/srcabstracts.list.

If you would like to order reports electronically, please send mail to src-report@src.dec.com.

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Research Report 1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Research Report 2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Research Report 3, January 11, 1985.

"Eliminating **go to**'s while Preserving Program Structure."
Lyle Ramshaw.
Research Report 4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Research Report 5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Research Report 6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Research Report 7, November 14, 1985.
Revised October 31, 1986.

"On Interprocess Communication."
Leslie Lamport.
Research Report 8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Research Report 9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Research Report 10, May 1, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Research Report 11, May 5, 1986.

"Fractional Cascading."
Bernard Chazelle and Leonidas J. Guibas.
Research Report 12, June 23, 1986.

"Retiming Synchronous Circuitry."
Charles E. Leiserson and James B. Saxe.
Research Report 13, August 20, 1986.

"An $O(n^2)$ Shortest Path Algorithm for a Non-Rotating Convex Body."
John Hershberger and Leonidas J. Guibas.
Research Report 14, November 27, 1986.

"A Simple Approach to Specifying Concurrent Systems."
Leslie Lamport.
Research Report 15, December 25, 1986.
Revised January 26, 1988.

"A Generalization of Dijkstra's Calculus."
Greg Nelson.
Research Report 16, April 2, 1987.

"*win* and *sin*: Predicate Transformers for Concurrency."
Leslie Lamport.
Research Report 17, May 1, 1987.
Revised September 16, 1988.

"Synchronizing Time Servers."
Leslie Lamport.
Research Report 18, June 1, 1987.
Temporarily withdrawn to be rewritten.

"Blossoming: A Connect-the-Dots Approach to Splines."
Lyle Ramshaw.
Research Report 19, June 21, 1987.

"Synchronization Primitives for a Multiprocessor: A Formal Specification."
A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin.
Research Report 20, August 20, 1987.

"Evolving the UNIX System Interface to Support Multithreaded Programs."
Paul R. McJones and Garret F. Swart.
Research Report 21, September 28, 1987.

"Building User Interfaces by Direct Manipulation."
Luca Cardelli.
Research Report 22, October 2, 1987.

"Firefly: A Multiprocessor Workstation."
C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr.
Research Report 23, December 30, 1987.

"A Simple and Efficient Implementation for Small Databases."
Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber.
Research Report 24, January 30, 1988.

"Real-time Concurrent Collection on Stock Multiprocessors."
John R. Ellis, Kai Li, and Andrew W. Appel.
Research Report 25, February 14, 1988.

"Parallel Compilation on a Tightly Coupled Multiprocessor."
Mark Thierry Vandevoorde.
Research Report 26, March 1, 1988.

"Concurrent Reading and Writing of Clocks."
Leslie Lamport.
Research Report 27, April 1, 1988.

"A Theorem on Atomicity in Distributed Algorithms."
Leslie Lamport.
Research Report 28, May 1, 1988.

"The Existence of Refinement Mappings."
Martín Abadi and Leslie Lamport.
Research Report 29, August 14, 1988.

"The Power of Temporal Proofs."
Martín Abadi.
Research Report 30, August 15, 1988.

"Modula-3 Report."
Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson.
Research Report 31, August 25, 1988.
This report has been superseded by Research Report 52.

"Bounds on the Cover Time."
Andrei Broder and Anna Karlin.
Research Report 32, October 15, 1988.

"A Two-view Document Editor with User-definable Document Structure."
Kenneth Brooks.
Research Report 33, November 1, 1988.

"Blossoms are Polar Forms."
Lyle Ramshaw.
Research Report 34, January 2, 1989.

"An Introduction to Programming with Threads."
Andrew Birrell.
Research Report 35, January 6, 1989.

"Primitives for Computational Geometry."
Jorge Stolfi.
Research Report 36, January 27, 1989.

"Ruler, Compass, and Computer: The Design and Analysis of Geometric Algorithms."
Leonidas J. Guibas and Jorge Stolfi.
Research Report 37, February 14, 1989.

"Can fair choice be added to Dijkstra's calculus?"
Manfred Broy and Greg Nelson.
Research Report 38, February 16, 1989.

"A Logic of Authentication."
Michael Burrows, Martín Abadi, and Roger Needham.
Research Report 39, February 28, 1989.
Revised February 22, 1990.

"Implementing Exceptions in C."
Eric S. Roberts.
Research Report 40, March 21, 1989.

"Evaluating the Performance of Software Cache Coherence."
Susan Owicki and Anant Agarwal.
Research Report 41, March 31, 1989.

"WorkCrews: An Abstraction for Controlling Parallelism."
Eric S. Roberts and Mark T. Vandevoorde.
Research Report 42, April 2, 1989.

"Performance of Firefly RPC."
Michael D. Schroeder and Michael Burrows.
Research Report 43, April 15, 1989.

"Pretending Atomicity."
Leslie Lamport and Fred B. Schneider.
Research Report 44, May 1, 1989.

"Typeful Programming."
Luca Cardelli.
Research Report 45, May 24, 1989.

"An Algorithm for Data Replication."
Timothy Mann, Andy Hisgen, and Garret Swart.
Research Report 46, June 1, 1989.

"Dynamic Typing in a Statically Typed Language."
Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin.
Research Report 47, June 10, 1989.

"Operations on Records."
Luca Cardelli and John C. Mitchell.
Research Report 48, August 25, 1989.

"The Part-Time Parliament."
Leslie Lamport.
Research Report 49, September 1, 1989.

"An Efficient Algorithm for Finding the CSG Representation of a Simple Polygon."
David Dobkin, Leonidas Guibas, John Hershberger, and Jack Snoeyink.
Research Report 50a, September 10, 1989.

"Boolean Formulæ for Simple Polygons" (video).
John Hershberger and Marc H. Brown.
Research Report 50b, September 10, 1989.

"Experience with the Firefly Multiprocessor Workstation."
Susan Owicki.
Research Report 51, September 15, 1989.

"Modula-3 Report (revised)."
Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson.
Research Report 52, November 1, 1989.

"IO Streams: Abstract Types, Real Programs."
Mark R. Brown and Greg Nelson.
Research Report 53, November 15, 1989.

"Explicit Substitutions."
Martín Abadi, Luca Cardelli, Pierre-Louis Curien, Jean-Jacques Lévy.
Research Report 54, February 6, 1990.

"A Semantic Basis for Quest."
Luca Cardelli and Giuseppe Longo.
Research Report 55, February 14, 1990.

"Abstract Types and the Dot Notation."
Luca Cardelli and Xavier Leroy.
Research Report 56, March 10, 1990.

"A Temporal Logic of Actions."
Leslie Lamport.
Research Report 57, April 1, 1990.

"Report on the Larch Shared Language: Version 2.3"
John V. Guttag, James J. Horning, Andrés Modet.
Research Report 58, April 14, 1990.

"Autonet: a High-speed, Self-configuring Local Area Network Using Point-to-point Links."
Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, Charles P. Thacker.
Research Report 59, April 30, 1990.

"Debugging Larch Shared Language Specifications."
Stephen J. Garland, John V. Guttag, and James J. Horning.
Research Report 60, July 4, 1990.

"In Memoriam: J.C.R. Licklider 1915–1990."
Research Report 61, August 7, 1990.

"Subtyping Recursive Types"
Roberto M. Amadio and Luca Cardelli.
Research Report 62, August 14, 1990.

"Heap Usage in the Topaz Environment"
John DeTreville.
Research Report 63, August 20, 1990.

"Experience with Concurrent Garbage Collectors for Modula-2+"
John DeTreville.
Research Report 64, August 20, 1990.

"An Axiomatization of Lamport's Temporal Logic of Actions"
Martín Abadi.
Research Report 65, October 12, 1990.

"Composing Specifications"
Martín Abadi and Leslie Lamport.
Research Report 66, October 10, 1990.

"Authentication and Delegation with Smart-cards"
M. Abadi, M. Burrows, C. Kaufman, B. Lampson.
Research Report 67, October 22, 1990.

Research Report 68: in preparation.

Research Report 69: in preparation.

"A Calculus for Access Control in Distributed Systems"
M. Abadi, M. Burrows, B. Lampson, G. Plotkin.
Research Report 70, February 28, 1991.

"Trading Space for Time in Undirected s-t Connectivity"
Andrei Z. Broder, Anna R. Karlin, Prabhakar Raghavan, Eli Upfal.
Research Report 71, May 7, 1991.

"LM3: A Larch Interface Language for Modula-3. A Definition and Introduction. Version 1.0"
Kevin D. Jones.
Research Report 72, June 10, 1991.

"Decidability and Expressiveness for First-Order Logics of Probability"
Martín Abadi and Joseph Y. Halpern.
Research Report 73, June 18, 1991.

"Introduction to LCL, A Larch/C Interface Language"
J.V. Guttag and J.J. Horning.
Research Report 74, July 24, 1991.

Research Report 75: in preparation.

"Color and Sound in Algorithm Animation"
Marc H. Brown and John Hershberger.
Research Report 76a, August 30, 1991.

"An Anthology of Algorithm Animations using Zeus" (video).
Marc H. Brown.
Research Report 76b, August 30, 1991.

"Automatic Reconfiguration in Autonet"
Thomas L. Rodeheffer and Michael D. Schroeder.
Research Report 77, September 18, 1991.

78 **Using Transformations and Verification in Circuit Design**

James B. Saxe, Stephen J. Garland, John V. Guttag, James J. Horning

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301