Acta Informatica **33,** 421–456 (1996)

Algebraic Models of Microprocessors: Architecture and Organisation

N. A. Harman and J. V. Tucker
Department of Computer Science,
University College of Swansea,
Swansea SA2 8PP,
WALES,
United Kingdom.

1. INTRODUCTION.

1.1. Overview. In this paper we present a general algebraic method for modeling microprocessors at different levels of abstraction, and for expressing formally the relationships between each level. We consider algebraic models of microprocessors at levels of abstraction determined by time and by details of construction. The algebraic models are intended to isolate some fundamental features of the scientific structure of microprocessor computation; by which we mean formal models of their behaviour over time, and of the manipulation and representation of data. We are particularly interested in the computational rôle of clocks. The algebraic methods developed: (i) provide a basis for the formal, modular decomposition of the description of microprocessors, including correctness criteria; and (ii) support equational specification and verification techniques for the design of microprocessors that are relevant to a wide range of specification languages and theorem provers.

A clock is a means of dividing time into (not necessarily equal) segments; these segments are defined by steps or stages in a computation or process. We use this idea as follows. The method is to model a computer by means of the iteration of a map $f : A \to A$ in discrete time $T = \{0, 1, 2, \ldots\}$, defined by

$$F : T \times A \to A,$$

$$F(t, a) = f^t(a),$$

for $t \in T$, $a \in A$. The set A is made to model the state of the computer and the function f to model the next-state map; thus $F(t, a)$ is the state of the computer at time $t \in T$ on operating from initial state $a \in A$. The computer is then modeled by the algebra

$$(A, T \mid F).$$

The nature of the set A of states and clock T is determined by the level of abstraction of the computer. A typical clock would be the *system clock*. However, we may also consider *instruction clocks*, where each cycle represents the execution of a single instruction, or indeed any other division of time. Multiple clocks may exist in a single specification, and clocks may be related formally. We will consider a computer at two levels of abstraction.

We show how to give an equational specification of this type of algebra using the fact that F is defined by primitive recursive equations over the algebra

$$(A, T \mid f, 0, t + 1).$$

In turn this algebra is constructed by building A and f from a simpler many-sorted algebra

$$(B_1, \ldots, B_n \mid \sigma_1, \ldots, \sigma_m)$$

which is an abstract data type for microprocessor specification.

We show that this *iterated map method* is a systematic technique that decomposes the modeling of the computer into easily understood stages, represented by algebras, so that each model can be equationally specified using initial algebra semantics.

We illustrate our algebraic tools by specifying a simple computer (based on the DEC PDP-8). We develop a specification at the programmer's level, and consider the implementation of this specification at the microcode level. We consider the correctness of the microcode level specification with respect to the programmer's level specification.

Our computer at the programmer's level specification includes

- arithmetic and logic operations;
- conditional operations;
- branches; and
- subroutines.

Our microcode level representation includes

- a two-bus datapath;
- a microprogrammed controller;
- conditional microinstructions; and
- a single-level microinstruction subroutine mechanism.

With these modest constructs only a short account of the verification of the processor is possible. In this first encounter with the methods we have chosen not to consider input and output, though we indicate how the methods can be adapted.

1.2. Background. The mathematical models describe precisely microprocessor structure in a way that is independent of formally defined specification languages, or input languages to theorem provers and proof checkers. By using algebraic methods we obtain semantical models and systems of equations that are independent of specific machine-reasoning systems. The idea is that models of a system are necessary before any attempt to *encode* the system into any specific input language for, for example, a theorem prover. Models of systems, such as presented in this paper, will be representable in, and processable by, most machine-oriented reasoning systems.

Related work on the formal specification and verification of microprocessors includes the following.

- A simple computer, known as "Gordon's computer", has been considered on a number of occasions. Gordon [1983b] describes the specification, high-level implementation and verification of the computer using LCF-LSM (Gordon [1983a]); Joyce [1987] describes and verifies the the same computer, using HOL (Gordon [1987]); and Stavridou [1993] uses OBJ3 (Goguen and Winkler [1988]), though only performs a partial verification.
- The *Viper* microprocessor has been specified, designed and partially verified using HOL: see, for example, Cohn [1987], Cullyer [1987a] and Cullyer [1987b].
- An implementation of Landin's SECD machine (Landin [1963]) is specified, designed and verified, in HOL: see, for example, Graham [1992], Graham and Birtwistle [1990], and Birtwistle and Graham [1990].
- The FM8501, a PDP-11-based processor, and the significantly more complex FM9001 have been specified, designed and verified using the Boyer-Moore theorem prover (Boyer and Moore [1988]): see Hunt [1986], Hunt [1989], Hunt [1992] and Hunt [1994]. The FM9001 is also considered in Bose and Johnson [1993].
- Parts of the Inmos Transputers T800 and T9000 have been specified, designed and verified using an Occam-based transformation system: see May et al [1992] and Roscoe [1992].
- Additionally, the following are of interest: Geser [1989], which discusses the Intel 8085 processor, and also Windley [1993], and Chazarain and Collavizza [1993].

A primary characteristic of the work summarised above is the dedication to a specific language, theorem prover, or proof assistant. A specific aim of the work described in this paper is to be independent of specific languages and systems. Furthermore, the verifications presented in the work above are long and difficult, often amounting to a *tour de force* in the application of a particular machine-based verification tool. Such techniques would be of limited use in routine verification attempts.

For work on hardware specification and verification in general, see, for example, Gordon [1987], Cohn and Gordon [1990], Milne [1989], Milne [1990], Melham [1988], McEvoy and Tucker [1990], Subrahmanyam [1988], Johnson and Zhu [1991], and Zhu and Johnson [1991], Weijland [1990], Hanna and Daeche [1993], and Melham [1993].

This work is part of a project on the theory of verifiable *synchronous concurrent algorithms*. A synchronous concurrent algorithm (SCA) is an algebraic model of parallel deterministic computation where the parallelism and determinism is expressed using discrete space and discrete time. The work on formal specification, of which this is a part, is intended to support the work on SCAs by providing a basis for the formal verification of parallel systems modeled as SCAs. The work of the SCA group encompasses methodologies for specification and design; formal analysis; manipulation and verification of specifications and designs; and software tools. For further information see Tucker [1991] and Thompson and Tucker [1991]. For work on case studies on hardware, see Harman and Tucker [1988a], Harman and Tucker [1988b], Harman and Tucker [1990], Harman [1989], and Harman and Tucker [1992a]. In particular, Harman and Tucker [1988b] and chapter 7 of Harman [1989] are concerned with the specification of computers.

1.3. Contents. The structure of this paper is as follows. In §2 we discuss algebraic tools for modeling and equational specification. In §3 we give a general account of the application of the iterated map algebras to the modeling of program execution and implementation in digital computers, and the relations between models of levels of abstraction. In §4 we outline the informal specification of a simple computer. In §5 we develop an algebraic specification of this computer. In §6 we propose an implementation of this computer. In §7, we consider the relationship between the specifications of the computer and of the controller, and show how the different levels of timing abstraction of each can be formally related. In §8 we outline the process of verifying the correctness of one level of abstraction with respect to its predecessor. In §9 we sketch how our model may be extended to accommodate direct input-output to and from a processor. In §10 we discuss problems and further work.

The authors would like to thank J M Emmett, M Summerfield and three referees for useful comments on a draft of this paper.

2. BASIC ALGEBRAIC TOOLS.

In this section, we introduce the algebraic ideas we need for writing algebraic specifications of computers.

2.1. Abstract Data Types. The theory of modeling *abstract data types* by *many-sorted algebras* has been extensively covered in the literature: see, for example, Meinke and Tucker [1992], Wechler [1991], Wirsing [1990] and Ehrig and Mahr [1985].

A *many-sorted algebra* A consists of one or more non-empty *carrier sets* $A_1, \ldots, A_s$, constants $c_1,\ldots,c_p \in A_i$, $i \in \{1,\ldots,s\}$, and operations f_i of the form

$$f_i : A_{s_1} \times \cdots \times A_{s_n} \to A_{s_{n+1}},$$

where $i = 1, \ldots, q$ and $s_1, \ldots, s_n, s_{n+1} \in \{1, \ldots, s\}$.

2.2. Clocks. A clock is an algebra $(T \mid 0, t+1)$ where $T = \{0, 1, \ldots\}$ identifies time intervals called *clock cycles*, 0 denotes the first clock cycle, and $t+1$ allows us to count clock cycles.

2.3. Streams. A *stream* $s \in [T \to A]$ is a function from a clock T to a set A of data items. A stream is a formal representation of time-separated data items. Given stream $s \in [T \to A]$, clock cycle $t \in T$, then $s(t)$ represents the data item arriving on stream s at time t.

2.4. Iterated Maps and Primitive Recursion. Let A be a non-empty set and $f : A \to A$ be a map, and form the algebra $(A \mid f)$. Let T be a clock and consider the function

$$F : T \times A \to A$$

that iterates f as follows: for $t \in T$, $a \in A$

$$\begin{aligned} F(0, a) &= a \\ F(t+1, a) &= f(F(t, a)). \end{aligned}$$

A solution to this set E_F of equations is

$$F(t, a) = f^t(a)$$

that generates the sequence, called an *orbit*,

$$a,\ f(a),\ f^2(a), \ldots,\ f^t(a), \ldots$$

and which will be used to represent the behaviour of a computer in time, starting from an initial state a.

The equations of E_F are primitive recursive equations over the algebra $(A, T \mid f, 0, t+1)$, see Tucker and Zucker [1988], Tucker [1991].

2.5. A Decomposition. Let $A = A_1 \times \cdots \times A_n$, where each A_i, $i = 1,\ldots,n$ is a non-empty set. Consider the function $F : T \times A \to A$ as defined in §2.4. The function F can be rewritten

$$F(t, a_1, \ldots, a_n) = (F_1(t, a_1, \ldots, a_n), \ldots, F_n(t, a_1, \ldots, a_n))$$

for $t \in T$, $a_i \in A_i$, and where $F_i : T \times A_1 \times \cdots \times A_n \to A_i$ is the i^{th} component function for F, $1 \leq i \leq n$.

Furthermore, the function $f : A \to A$, whose iteration defines F, can be rewritten

$$f(a_1, \ldots, a_n) = (f_1(a_1, \ldots, a_n), \ldots, f_n(a_1, \ldots, a_n))$$

for $a_i \in A_i$, and where $f_i : A_1 \times \cdots \times A_n \to A_i$ is the i^{th} component function for f, $1 \leq i \leq n$.

Substituting these component functions for F and f we may rewrite the equations E_F in §2.4 as follows:

$$\begin{aligned} F_1(0, a_1, \ldots, a_n) &= a_1, \\ &\vdots \qquad\qquad \vdots \\ F_n(0, a_1, \ldots, a_n) &= a_n, \\ F_1(t+1, a_1, \ldots, a_n) &= f_1(F_1(t, a_1, \ldots, a_n), \ldots, F_n(t, a_1, \ldots, a_n)), \\ &\vdots \qquad\qquad \vdots \\ F_n(t+1, a_1, \ldots, a_n) &= f_n(F_1(t, a_1, \ldots, a_n), \ldots, F_n(t, a_1, \ldots, a_n)). \end{aligned}$$

Equations $E_{F_1 \ldots F_n}$ are (simultaneous) primitive recursive equations over the algebra

$$(A_1, \ldots, A_n \mid f_1, \ldots, f_n, 0, t+1).$$

See Tucker and Zucker [1993]. The following result is discussed further in Thompson and Tucker [1991] and Tucker and Zucker [1988].

2.5.1. Lemma. *Suppose that* (Σ_0, E_0) *is an equational (or conditional equational) specification of*

$$(A_1, \ldots, A_n \mid f_1, \ldots, f_n)$$

possibly containing hidden sets and functions, under initial algebra semantics. Then

$$(\Sigma_0 \cup \Sigma_T \cup \{F_1, \ldots, F_n\}, E_0 \cup E_{F_1,\ldots,F_n})$$

is an equational (or conditional equational) specification of

$$(A_1, \ldots, A_n, T \mid f_1, \ldots, f_n, 0, t+1, F_1, \ldots, F_n)$$

and hence of

$$(A_1, \ldots, A_n, T \mid F_1, \ldots, F_n)$$

under initial algebra semantics.

In fact, the specification can be chosen to have useful term rewriting properties (e.g. orthogonality and completeness).

2.6. Comparing Iterated Maps. Consider the iteration of two maps $f : A \to A$ and $g : B \to B$ defined by

$$F : T \times A \to A \text{ and } G : S \times B \to B$$

where

$$F(t, a) = f^t(a) \text{ and } G(s, b) = g^s(b)$$

for $t \in T$, $a \in A$, $s \in S$ and $b \in B$. We compare these iterations below.

2.6.1. A General Case. We say G *simulates* F if there are maps $\alpha : T \times A \to S \times B$ and $\beta : B \to A$. such that the following diagram commutes.

$$\begin{array}{ccc} T \times A & \xrightarrow{F} & A \\ \downarrow \alpha & & \uparrow \beta \\ S \times B & \xrightarrow{G} & B \end{array}$$

This means for $t \in T$, $a \in A$

$$F(t, a) = \beta(G(\alpha_1(t, a), \alpha_2(t, a))),$$

and, equivalently, given the definition of F and G,

$$f^t(a) = \beta(g^{\alpha_1(t,a)}(\alpha_2(t, a))).$$

2.6.2. Time-Independent State Comparison. The comparison we have is of a more specific form where time is state-dependent, but state is not time-dependent. The map α has the following coordinate functions α_1, α_2 respectively:

$$\overline{\lambda} : T \times A \to S$$
$$\phi : A \to B$$

and we denote the map β

$$\psi : B \to A.$$

In addition, we expect that

$$\psi(\phi(a)) = a.$$

We say G simulates F with respect to $(\overline{\lambda}, \phi)$ and ψ if the following diagram commutes.

$$\begin{array}{ccc} T \times A & \xrightarrow{F} & A \\ \downarrow{\scriptstyle (\overline{\lambda},\phi)} & & \uparrow{\scriptstyle \psi} \\ S \times B & \xrightarrow{G} & B \end{array}$$

Hence for $t \in T$, $a \in A$

$$F(t,a) = \psi(G(\overline{\lambda}(t,a), \phi(a))),$$

and equivalently

$$f^t(a) = \psi(g^{\overline{\lambda}(t,a)}(\phi(a))).$$

2.7. Retimings. Specifications may contain multiple clocks which are not equivalent in speed. Also, each cycle of a clock T need not be the same length relative to another (e.g. standard) clock R: that is, clocks can be irregular. In §2.6 we saw that time on clock S is determined by both time on clock T, and state A.

2.7.1. State-Independent Retimings. In order to relate multiple clocks we introduce *retimings*. Let T and R be two clocks. A retiming $\lambda : T \to R$ is a surjective, monotonic map. We denote the set of retimings from clock T to clock R by $Ret(T,R)$. A typical retiming is illustrated in Fig. 1.

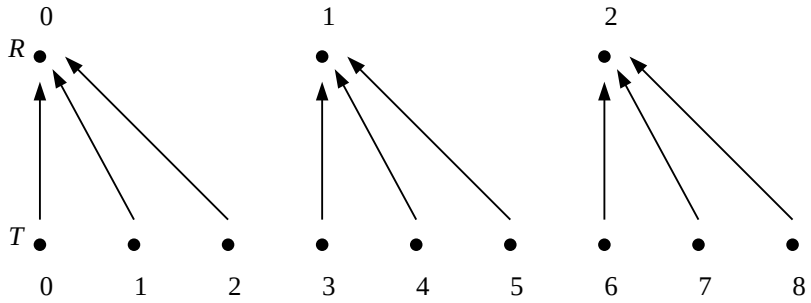

Fig. 1. A Retiming Between Clocks T and R.

Example. Observe that in Fig. 1 every cycle of clock R corresponds with the same number $k \in \mathbf{N}^+$ cycles of clock T. Clock T runs at exactly k times the rate of clock R, and $\lambda(t) = \lfloor t/k \rfloor$. We say such retimings are *linear* of *length* k.

For each retiming λ there is a corresponding *immersion* $\overline{\lambda} : R \to T$,

$$\overline{\lambda}(r) = (\mu\, t \in T)[\lambda(t) = r].$$

The function $L : [R \to \mathbf{N}^+] \to Ret(T,R)$ defines a retiming in which each clock cycle $L(x)(r)$ lasts $x(r)$ cycles of clock T.

$$L(a)(t) = \begin{cases} 0, & \text{if } t = 0; \\ (\mu\, r \in R)[a(0) + a(1) + \cdots + a(r) > t], & \text{if } t > 0. \end{cases}$$

The function $sch : Ret(S,T) \to [[T \to A] \to [S \to A]]$ *schedules* a stream $a \in [T \to A]$ by retiming λ, and is defined by

$$sch(\lambda)(a)(s) = a\lambda(s).$$

2.7.2. State-Dependent Retimings. Retimings may also be *state-dependent*. That is, for each state $\sigma \in B$, we may define a retiming $\lambda(-, \sigma) : T \to R$. An example of such a retiming would be that between the instruction clock T and system clock S of a microprocessor, where the number of cycles of S corresponding to each cycle of T will depend on the precise instruction to be executed, and hence on the state of the machine. Precisely such a case is illustrated in §2.6.2, where F represents the microprocessor at the level of the instruction clock T, and G represents the level of the system clock S.

Further formal tools can be developed from retimings: see Harman [1989], Harman and Tucker [1990], Harman and Tucker [1988a], and Harman and Tucker [1992a].

3. COMPUTERS AS STATE TRANSFORMERS.

We will begin by giving computers *state transformer* or *programmer's model* specifications PM, that progressively update the state of the machine. By *state*, we mean those aspects of the internal structure of the computer visible to the *programmer*. That is, the registers and memory that can be *explicitly* modified. This model is sometimes called the *architecture* of the computer (Stallings [1987]). There is no requirement that these registers and/or memory actually exist in the form seen in PM (for example, in the MC68000 each 32-bit register is implemented as a pair of physically separated 16-bit registers: see Anceau [1986]). However, the machine must behave *as if the state components of PM were physically present*. In an actual design, there will be additional registers, hidden from the programmer, that are necessary for the implementation of the machine. These registers are not properly part of the specification. Additionally, we include primary memory in a state transformer specification even when specifying, say, a microprocessor, where memory is *not* an integral part of the device.

The programmer's model specification is followed by an *abstract circuit design* representation AC. The abstract circuit design is a high-level representation of the implementation of the microprocessor. Typically, it will consist of a decomposition into *controller*, *datapath* and *memory*. Each of the components of the decomposition may contain state information that is not present in PM.

A hierarchy of specification models may be seen emerging. First, the programmer's model PM, followed by an abstract circuit design AC, and further modeling the implementation in successively greater detail.

3.1. Models and Abstraction in the Design of a Computer. The rôle of the programmer's model specification PM is to form part of a specification of some microprocessor implementation I, for which we may require a *correctness proof*. We can ask: what is I, and how is it constructed?

In the case of a complex device the design would proceed in *stages*, with each successive stage a *refinement* of its predecessor, terminating in a set of *chip masks*, or other representation of the final hardware. We also admit the possibility of false trails in the design process: ideas that later are seen as inappropriate, and explorations of alternate designs. The design process will not be a simple trail of increasing detail: see Harman [1989], Harman and Tucker [1992a]. In the basic case however, we can postulate an initial abstract circuit design AC, followed by one or more stages of *detailed circuit design*, and culminating in the final product: the chip mask-set, circuit board design, etc, whose characteristics are uniquely determined by a physical system.

We may consider how far into the design process we can profitably apply formal tools for proving correctness. Ultimately, in the design process, we must encounter real physical devices: in current technologies, these will be transistors. The behaviour of transistors, and other electronic components, is complex and so are accurate mathematical models of their behaviour. Simple models of transistor behaviour exist, but are not always accurate (Gordon [1987]), and generally contain minimal timing information. Accurate behaviour models are useful for limited simulation, but are unsuitable for formal reasoning. Furthermore, it can be argued that the *practical* usefulness (we would not dispute the *scientific*

value) of formal methods applied to circuits containing small numbers of transistors is limited.

While it may not be profitable to apply formal tools to all stages of the design process, we may certainly apply them to the abstract circuit design AC, and possibly to one or more stages of detailed circuit design. In the case of a microprocessor, what does AC consist of? We can partition a typical microprocessor into two main components: *control* and *datapath*. Additionally, since the initial programmer's model specification contained *memory* M, we must include M in AC. However, we are not concerned with the *design* of M, and we can employ the same specification as used in PM. The abstract circuit design phase consists of specifying the control section CT and the datapath DP, and composing these together with M to form AC. Then we must show that AC correctly implements PM. We may continue the design process by subdividing CT, DP and M and specifying their components until we reach practical limits imposed by the physical behaviour of devices.

3.2. Computer as an Iterated Map. The behaviour of a computer is described by an infinite sequence of states over time. The state comprises the contents of the machine's registers and memory. Additionally, there may be streams for input and output, see §9.

3.2.1. Operational Semantics. Let C be the set of states. Starting from some *initial state* $\sigma \in C$, in which a program is held in memory and the *program counter* points to the start of this program, the machine state evolves in time as controlled by a *next state function*

$$comp : C \to C,$$

and a clock T. At every cycle t of the clock, $comp$ is applied to the current state σ to generate a new state $comp(\sigma)$. Hence the evolution of the state of the machine from time $0 \in T$ to time $t \in T$ is represented by

$$\sigma,\ comp(\sigma),\ comp(comp(\sigma)),\ \ldots,\ comp^t(\sigma).$$

The machine is therefore represented by

$$COMP : T \times C \to C$$

defined by the equations

$$\begin{aligned} COMP(0, \sigma) &= \sigma, \\ COMP(t+1, \sigma) &= comp(COMP(t, \sigma)). \end{aligned}$$

A solution $COMP$ has the form

$$COMP(t, \sigma) = comp^t(\sigma).$$

3.2.2. Algebraic Representation. We know from §2.4 that the computer is represented by a simultaneous primitive recursive function $COMP$ over the algebra

$$(C, T \mid comp, 0, t+1),$$

which we call the *next-state algebra*. The algebra

$$(C, T \mid COMP)$$

we call the *state algebra*. The algebra used to build the next-state algebra we call the *machine algebra*. The state algebra represents the machine and is equationally specified given an equational specification of the next-state algebra. In turn the next-state algebra should be equationally specified given an equational specification of the machine algebra.

3.2.3. A Note on Time. The speed of clock T determines the level of timing abstraction of $COMP$. Typically, in the programmer's model, T will be the *instruction clock*, where each clock cycle represents the execution of a single instruction. Since instructions take differing amounts of "real" time to execute, the instruction clock will be irregular with respect to the system clock. The system clock is at a lower level of timing abstraction than the instruction clock. Each cycle of the system clock represents one cycle of the clock signal used to control the computer or microprocessor. Another possible clock is the *memory clock*, in which each clock cycle corresponds with one memory access. The level of temporal abstraction of the memory clock lies between the system and instruction clocks. In §7 we will show how to formally map from the instruction clock to the system clock in the case of our simple computer.

3.3. A Decomposition: General Case. Following §2.5, we may specify a particular computer architecture at any level of abstraction by defining the state set C and next state function $comp : C \to C$. State set C will normally be a cartesian product $C_1 \times \cdots \times C_n$ where each C_i, $i = 1,\ldots,n$ represents registers and memory. The definition of $comp$ then reduces to the definition of the coordinate functions $comp_i : C_1 \times \cdots \times C_n \to C_i$, for $i = 1,\ldots,n$ from which $comp$ is constructed:

$$comp(c_1,\ldots,c_n) = (comp_1(c_1,\ldots,c_n),\ldots,comp_n(c_1,\ldots,c_n))$$

and hence $COMP : T \times C \to C$:

$$\begin{aligned}
COMP_1(0,c_1,\ldots,c_n) &= c_1,\\
&\vdots \qquad\qquad \vdots\\
COMP_n(0,c_1,\ldots,c_n) &= c_n,\\
COMP_1(t+1,c_1,\ldots,c_n) &= comp_1(COMP_1(t,c_1,\ldots,c_n),\ldots,COMP_n(t,c_1,\ldots,c_n)),\\
&\vdots \qquad\qquad \vdots\\
COMP_n(t+1,c_1,\ldots,c_n) &= comp_n(COMP_1(t,c_1,\ldots,c_n),\ldots,COMP_n(t,c_1,\ldots,c_n)).
\end{aligned}$$

The carriers $C_1,\ldots, C_n$ and the functions $comp_1,\ldots, comp_n$ of the state algebra are constructed from a machine algebra at a lower level of data abstraction (see §3.5).

3.4. A Decomposition: Abstract Circuit Design. Following the ideas of §2.5 and §3.1, consider a decomposition of the state of an abstract circuit representation of a computer into the states of its controller, datapath and memory. Let Ct be the set of states of the controller, Dp be the set of states of the datapath and Mem be the set of states of the memory, and suppose that

$$C = Ct \times Dp \times Mem$$

and hence for appropriate component functions

$$COMP_{AC}(t,c,d,m) = (CT(t,c,d,m), DP(t,c,d,m), MEM(t,c,m))$$

for all $t \in T$, $c \in Ct$, $d \in Dp$, $m \in Mem$. If the next state function has component functions

$$comp_{AC}(c,d,m) = (ct(c,d,m), dp(c,d,m), mem(c,d,m))$$

for $c \in Ct$, $d \in Dp$, $m \in Mem$, then we may rewrite the equations for the computer.

$$\begin{aligned}
&CT(0,c,d,m) = c,\\
&DP(0,c,d,m) = d,\\
&MEM(0,c,d,m) = m,\\
&CT(s+1,c,d,m) = ct(CT(s,c,d,m), DP(s,c,d,m), MEM(s,c,d,m)),\\
&DP(s+1,c,d,m) = dp(CT(s,c,d,m), DP(s,c,d,m), MEM(s,c,d,m)),\\
&MEM(s+1,c,d,m) = mem(CT(s,c,d,m), DP(s,c,d,m), MEM(s,c,d,m)),
\end{aligned}$$

The next-state algebra becomes

$$(Ct, Dp, Mem, T \mid ct, dp, mem, 0, t+1).$$

3.5. Machine Algebras. The machine algebra consists of carriers (typically vectors of *bits*) and functions representing ALU operations. A typical machine algebra for a microprocessor would be constructed as follows. Let the algebra Bit consist of carrier $\{0,1\}$, constants 0 and 1, and *logical operations* such as *and*, *or*, *not*, and so on (the precise choice depending on the architecture). Let $W_n = \{0,1\}^n$. The machine algebra M is constructed by adding to Bit carriers consisting of vectors of Bit, and special-purpose operations on these vectors: typically

$$(Bit, W_{32}, W_{16}, W_8, \mid and, or, not, add, shift, sub, mul, \ldots),$$

the precise choice of carriers and operations again depending on the architecture (see §5.1).

3.6. Correctness. Given a programmer's model PM of a computer, and an abstract circuit design AC intended to implement PM, what conditions must be met if we are to say that *AC correctly implements PM*?

The programmer's model specification PM describes how the state $\sigma \in C_{PM}$ of the machine *evolves* over time. Each component of $\sigma \in C_{PM}$ may, potentially, change during each cycle of clock T, where each cycle of T represents one instruction. The execution of an instruction is determined by a next-state function $pmcomp : C_{PM} \to C_{PM}$, and so

$$PM(t, \sigma) = pmcomp^t(\sigma).$$

The abstract circuit design AC behaves in a similar way, except now the state $\sigma' \in C_{AC}$ is enlarged to include registers/memory required to implement PM, and the controlling system clock S is *faster* than clock T. Again, computation is represented by a next-state function $accomp : C_{AC} \to C_{AC}$, and so

$$AC(s, \sigma') = accomp^s(\sigma').$$

Although an element σ' of C_{AC} will contain structure and information not in C_{PM}, at the start of the execution of any instruction, that information will refer to the execution of the *previous* instruction, or will be invariant between instructions (for example, the microcode memory). Any correct implementation of PM will assume that, at the start of the execution of any instruction, those parts of C_{AC} not in C_{PM} either contain "junk", or contain constant information. Therefore, at the start of any sequence of instructions we can assume that the contents of those parts of $\sigma' \in C_{AC}$ not in C_{PM} contain either arbitrary or pre-defined values.

Thus, suppose there exists a a projection function $\pi : C_{AC} \to C_{PM}$, which projects out those parts of C_{AC} also in C_{PM}, and furthermore, that there is a function $\alpha_x : C_{PM} \to C_{AC}$ which "pads" those elements of C_{AC} not in C_{PM} with appropriate values x. We may suppose that for all $\sigma \in C_{PM}$,

$$\pi\alpha_x(\sigma) = \sigma.$$

Informally then, *AC correctly implements PM* if, given an initial starting state $\sigma \in C_{PM}$, for any corresponding starting state $\alpha_x(\sigma) \in C_{AC}$, then for each cycle of clock S corresponding with the start of a cycle of clock T, we can compute a time $\overline{\lambda}(t, \sigma)$, such that $\pi(AC(\overline{\lambda}(t, \sigma), \alpha_x(\sigma)))$ is the state of PM at clock cycle t.

Recalling §2.6, we formalise the above correctness condition as follows. Given $PM : T \times C_{PM} \to C_{PM}$ and $AC : S \times C_{AC} \to C_{AC}$, we require the following diagram to commute.

$$\begin{array}{ccc} T \times C_{PM} & \xrightarrow{PM} & C_{PM} \\ \downarrow \phi & & \uparrow \pi \\ S \times C_{AC} & \xrightarrow{AC} & C_{AC} \end{array}$$

where $\phi : T \times C_{PM} \to S \times C_{AC}$ is defined by

$$\phi(t, \sigma) = (\overline{\lambda}(t, \sigma), \alpha_x(\sigma)),$$

$\alpha_x : C_{PM} \to C_{AC}$ "pads" C_{PM} to C_{AC}, $\overline{\lambda} : T \times C_{PM} \to S$, and $\pi : C_{AC} \to C_{PM}$ projects out those parts of C_{AC} also present in C_{PM}. Alternatively, we can express the correctness condition as follows: for all x,

$$PM(t, \sigma) = \pi(AC(\overline{\lambda}(t, \sigma), \alpha_x(\sigma))).$$

Recall the definitions of PM and AC as the iterated maps

$$pmcomp : C_{PM} \to C_{PM} \text{ and } accomp : C_{AC} \to C_{AC}$$

respectively. Then the condition for correctness becomes: for all $t \in T$, $\sigma \in C_{PM}$ and any padding values x

$$pmcomp^t(\sigma) = \pi(accomp^{\overline{\lambda}(t,\sigma)}(\alpha_x(\sigma))).$$

Observe that $\overline{\lambda} : T \times C_{PM} \to S$ qualifies as the immersion of a state-dependent retiming (§2.6.2).

4. INFORMAL SPECIFICATION.

The machine we will specify is based on the DEC PDP-8 (Bell et al [1978]). This machine has been used previously as a design example: for instance, see Florentin [1991], Barbacci [1982]. In this section, we will sketch the informal specification of the machine. Full informal specifications can be found in the literature. We will omit a number of straightforward features of the PDP-8 that are tedious to specify: specifically, the *operate* instruction, and the *I-O* instruction. The former deals with a range of trivial functions that do not require a memory operand (e.g., clear accumulator). The latter deals with input-output via *streams* (Harman [1989], Harman and Tucker [1990]), though these may be added (see §9 and Harman and Tucker [1994a]).

4.1. Registers, Word Size and Instruction Format. The PDP-8 uses a 12-bit word, giving an address space of $2^{12} (= 4096)$ words. There is a single *accumulator* ACC, a single-bit *link* register L, used for overflow detection and shifting, and a 12-bit *program counter* PC. All instructions are one word long, with format: 3-bit opcode; indirection bit; page bit; and 7-bit page offset. The page offset only allows 128 words to be addressed, hence the indirection and page-structured memory are used to allow access to the entire memory. Memory is divided into 128-word pages, and the page offset can either access a word in the first page of memory, or the page of memory containing the current instruction, depending on the *page bit*. The *indirection bit* allows indirect memory addressing.

4.2. Instruction Set. We will specify the following instructions.

(i) *AND*: Bitwise-and the accumulator with the specified memory operand. The result is stored in the accumulator.
(ii) *TAD*: Two's complement add of the accumulator and the specified memory operand. The result is stored in the accumulator.
(iii) *ISZ*: Increment the accumulator. If the new accumulator value is zero, skip the next instruction.
(iv) *DCA*: Deposit the accumulator contents at the specified memory location, then clear the accumulator.
(v) *JSR*: Subroutine call. Subroutine calls store the return address in the first word of the subroutine, and jump to the following word. To return, an indirect jump though the first word of the subroutine is necessary.
(vi) *JMP*: Jump unconditionally to the word specified by the memory operand.
(vii) Undefined operation codes cause an *exception*, with the value of the program counter replaced by an *exception branch address*.

5. FORMAL SPECIFICATION.

We will construct a formal specification of the machine informally described in §4 in the manner of §3.2 and §3.4. We will not give a complete formal specification, but supply sufficient information for the reader in possession of an informal specification to complete the process.

Following §3.2 we may construct the state algebra

$$(C, T \mid COMP)$$

of the machine as follows:

$$\begin{aligned} &COMP : T \times C \to C, \\ &COMP(0, c, m) = c, m \\ &COMP(t+1, c, m) = comp(COMP(t, c, m)). \end{aligned}$$

To proceed, we must define the state set C, together with the next-state function *comp*. There are two 12-bit registers ACC and PC, and a single-bit register L, so we define the state of the CPU

$$Cpu = A \times A \times Bit,$$

where $A = W_{12}$. Memory consists of 2^{12} 12-bit words. Hence we define

$$Mem = [A \to A].$$

Thus

$$C = Cpu \times Mem = A \times A \times Bit \times [A \to A]$$

(see §3).

5.1. The Machine Algebra. Following the process outlined in §3.5, we define the machine algebra for our example. Our machine will require the following carriers. Let $A = W_{12}$ represent memory words, memory addresses and the accumulator. Let $La = W_{13}$ represent the result of additions. Additionally, we require Bit, the booleans $\mathbf{B}$ and the natural numbers $\mathbf{N}$, and the functions $\wedge : A^2 \to A$, $\vee : A^2 \to A$ and $add : A^2 \to La$.

In addition to ALU operations, we add the successor function for $\mathbf{N}$, together with some functions for manipulating bit vectors, converting bit vectors to $\mathbf{N}$, and performing memory substitution.

The *bit field extraction functions* $bits^{W_i}_{j/l} : W_i \to W_i$, are defined by

$$bits^{W_i}_{j/l}(a_1, \ldots, a_i) = (0, \ldots, 0, a_j, \ldots, a_l),$$

$1 \le j \le l \le i$. For example, consider the case when $i = 12$ (i.e. $W_i = A$ above), $j = 6$, and $l = 12$. Then

$$bits^{A}_{6/12}(a_1, \ldots, a_{12}) = (0, 0, 0, 0, 0, 0, a_6, \ldots, a_{12}).$$

The function $pn : A \to \mathbf{N}$ interprets a bit vector $(a_1, \ldots, a_l) \in A$ as a natural number, using the usual binary representation of positive integers.

The function $sub : [A \to A] \times A \times A \to [A \to A]$ is the *memory substitution function*, defined by

$$sub(m, a, l)(i) = \begin{cases} m(i), & \text{if } l \neq i; \\ a, & \text{if } l = i. \end{cases}$$

We will use $m[a/l]$ to stand for $sub(m, a, l)$. Summarising, the machine algebra is as follows:

$$(Bit, A, La, A \to A, \mathbf{B}, \mathbf{N} \mid \wedge, \vee, add, succ, bits^{A}_{j/l}, pn, sub, cases, =),$$

with $j/l \in \{1/3, 6/12, 4/4, 5/5\}$.

5.2. Field Extraction Functions. We define the following functions to extract the various fields of an instruction. The *opcode* field is extracted by $op : A \to \mathbf{N}$, defined by

$$op(a) = pn(bits^A_{1/3}(a)).$$

We use pn to convert the three opcode bits into a natural number. This makes the specification easier to read.

The *page offset* field is extracted by $pgoff : A \to A$, defined by

$$pgoff(a) = bits^A_{6/12}(a).$$

The *indirection bit* is extracted by $indir : A \to \mathbf{B}$, defined by

$$indir(a) = \begin{cases} tt, & \text{if } bits^A_{4/4}(a) = 1; \\ ff, & \text{if } bits^A_{4/4}(a) = 0. \end{cases}$$

The *page bit* is extracted by $pgbit : A \to \mathbf{B}$, defined by

$$pgbit(a) = \begin{cases} tt, & \text{if } bits^A_{5/5}(a) = 1; \\ ff, & \text{if } bits^A_{5/5}(a) = 0. \end{cases}$$

5.3. Next-State Function. We define the next state function

$$comp : A \times A \times Bit \times [A \to A] \to A \times A \times Bit \times [A \to A]$$

as follows:

$$comp(a, pc, l, m) =$$

$$\begin{cases} (a \wedge mval(pc, m), pc + 1, l, m), & \text{if } op(m(pc)) = 0;\ (i) \\ (bits^{La}_{1/12}(add(a, l, mval(pc, m))), pc + 1, & \text{if } op(m(pc)) = 1;\ (ii) \\ \quad add_{13}(a, l, mval(pc, m)), m) & \\ (a, tskip(pc, m), l, & \text{if } op(m(pc)) = 2;\ (iii) \\ \quad m[mval(pc, m) + 1/maddr(pc, m)]), & \\ (0, pc + 1, l, & \text{if } op(m(pc)) = 3;\ (iv) \\ m[a/maddr(pc, m)]), & \\ (a, maddr(pc, m) + 1, l, m[pc + 1/maddr(pc, m)]), & \text{if } op(m(pc)) = 4;\ (v) \\ (a, maddr(pc, m), l, m), & \text{if } op(m(pc)) = 5;\ (vi) \\ (a, EXCEPTIONADDR, l, m), & \text{if } op(m(pc)) = 6\ (vii) \\ & \text{or } op(m(pc)) = 7, \end{cases}$$

where $EXCEPTIONADDR = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$, and $mval$, $maddr$ and $tskip$ are defined below. We explain each case briefly below.

- (i) AND: the contents of accumulator a are replaced with the bitwise AND of the initial value of accumulator a and the memory operand $mval(pc, m)$. The program counter pc is incremented.
- (ii) TAD: accumulator a is replaced by the least-significant 12 bits of the sum of a and $mval(pc, m)$, and l is replaced by the most significant bit of this sum. The program counter pc is incremented.
- (iii) ISZ: the memory word indicated by $maddr(pc, m)$ is incremented by one, and the program counter is incremented by one or two (by $tskip(pc, m)$), depending on the new value of the memory word.

(*iv*) DCA: the accumulator a is stored at the memory word indicated by $maddr(pc, m)$ and the accumulator is set to zero. The program counter is incremented.

(*v*) JSR: the program counter pc plus one is stored at memory word $maddr(pc, m)$, and the value of pc is replaced by $mval(pc, m)+1$.

(*vi*) JMP: the value of pc is replaced by $mval(pc, m)$.

(*vii*) Operation code values of 6 or 7 represent illegal instructions, and generate an *exception*, resulting in a branch to special memory address *EXCEPTIONADDR*. It is the responsibility of the user to ensure appropriate error-handling code is present at this address.

It now remains to define the subfunctions of *comp*.

5.4. Memory Access. We need both memory address ($maddr$) and read memory ($mval$) functions. We define $maddr : A \times Mem \to A$ as follows:

$$maddr(pc, m) =$$
$$\begin{cases} pgoff(m(pc)), & \text{if } pgbit(m(pc)) = ff \text{ and } indir(m(pc)) = ff; \ (i) \\ pgoff(m(pc)) \vee & \text{if } pgbit(m(pc)) = tt \text{ and } indir(m(pc)) = ff; \ (ii) \\ (pc \wedge ADDRMSK), & \\ m(pgoff(m(pc))), & \text{if } pgbit(m(pc)) = ff \text{ and } indir(m(pc)) = tt; \ (iii) \\ m(pgoff(m(pc)) \vee & \text{if } pgbit(m(pc)) = tt \text{ and } indir(m(pc)) = tt. \ (iv) \\ (pc \wedge ADDRMSK)), & \end{cases}$$

where $ADDRMSK = (1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0)$. In cases (*i*) and (*iii*) we are accessing memory words in page zero. In cases (*ii*) and (*iv*) we are accessing memory words in the current page, and so must extract the most significant five bits of pc and use them to extend the page offset. In cases (*iii*) and (*iv*), we are using indirect addressing, and so must make a memory access.

We define $mval : A \times Mem \to A$ as follows.

$$mval(pc, m) = m(maddr(pc, m)).$$

We define $tskip : A \times M \to A$ as follows.

$$tskip(pc, m) = \begin{cases} pc + 1, & \text{if } mval(pc, m) + 1 \neq 0; \\ pc + 2, & \text{if } mval(pc, m) + 1 = 0. \end{cases}$$

5.5. Algebraic Structure of a Microprocessor. It is easy to check that function *comp* is polynomial over the machine algebra, and that therefore *COMP* is primitive recursive over the machine algebra. Since the machine algebra is easily algebraically specified, the state algebra of the computer is algebraically specified by lemma 2.5.1.

6. A MICROPROGRAMMED IMPLEMENTATION.

We now consider how *COMP* specified in §5 may be implemented. We will proceed in the manner outlined in §3.1. First, we will decide on, and informally specify, an appropriate datapath *DP*, and controller *CT*. Then we will show how *DP* and *CT* may be formalised as iterated maps.

6.1. The Datapath. We will use the two-bus datapath, illustrated in Fig. 2.

As well as the user-visible registers ACC, L and PC, there are four further registers, two buses A and B, the ALU and an *incrementer* for PC. The buses A and B are 13-bit-wide connection paths between components of the datapath. The two buses A and B are not equivalent, in that not all components are connected to both buses (see Fig. 2). The *memory address register* MAR holds memory addresses, and is linked to the *address bus*. The *memory buffer register* MBR holds data to be read/written from/to memory, and is linked to the *data bus*. The *instruction register* IR holds the current instruction, and is

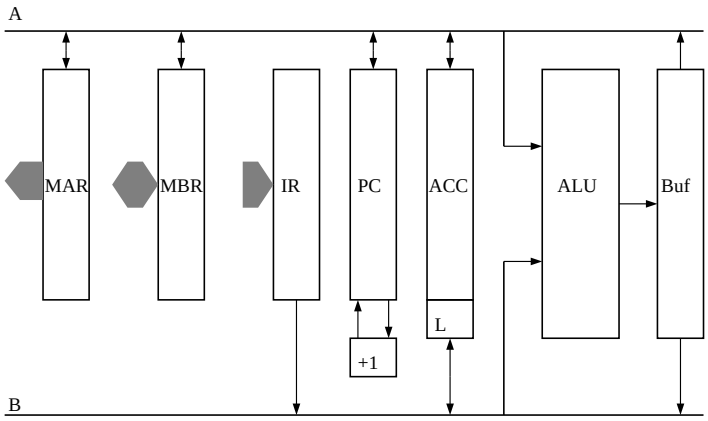

Fig. 2. The Datapath DP.

linked separately to the data bus. The *ALU buffer register* is used as a store for ALU results. Additionally, the results of ALU *test* operations are available to the controller.

To control the datapath, a total of 24 separate control signals are required. These are classified as follows.

- Two control memory: MEM.rd and MEM.wr.
- Two control MAR: MAR.A.rd and MAR.A.wr.
- Four control MBR: MBR.A.rd, MBR.A.wr, MBR.data.rd, and MBR.data.wr.
- Two control IR: IR.data.rd and IR.B.wr.
- Three control PC: PC.A.rd, PC.A.wr and PC.increment.
- Five control ACC: ACC.A.rd, ACC.A.wr, ACC.B.rd, ACC.B.wr and L.A.rd.
- Four select the appropriate ALU operation.
- Two control the ALU buffer: BUF.A.wr and BUF.B.wr.

The general format of these signals is

register name.bus name.operation name,

where *operation* is rd (*read*) or wr (*write*), except in a few self-explanatory cases.

6.2. The Controller. We will use a microcoded controller *CT* to control *DP*. To avoid confusion, we will use *microinstruction* to refer to instructions of the microprogrammed controller, and *instruction* to refer to instructions of *COMP*.

Each microinstruction will have three fields. The *control* field will control the operation of *DP*, and will be 24 bits long (see §6.1); the *sequencing* field will determine which microinstruction will be executed next, and will be four bits long; and the *address* field will contain the address of the next microinstruction in the event of a branch or subroutine call. Branch addresses will be eight bits long.

The sequencing field bits will allow the following microinstruction sequences: *next sequential microinstruction*; *unconditional or conditional jump*; *unconditional or conditional subroutine*; and *decode next instruction*. The last is used when the sequence of microinstructions to execute the current machine instruction is exhausted, and we must start a new machine instruction. The following table shows the allowable combinations of the four control bits.

next I	Sub. Rout.	Jump	Cond	
0	0	0	0	Do the next sequential microinstruction.
0	0	1	0	Unconditional jump to the branch address.
0	0	1	1	Conditional Jump to the branch address.
0	1	0	0	Unconditional Subroutine call to the branch address.
0	1	0	1	Conditional Subroutine call to the branch address.
0	1	1	0	Return from Subroutine.
1	0	0	0	Start the next programmer's level instruction.

Note that conditional jumps and subroutine calls are based on the results of ALU *test* operations. Further note that only one level of subroutine call is allowed, as the return addresses will be stored in a register. Fig. 3. illustrates the microprogrammed controller.

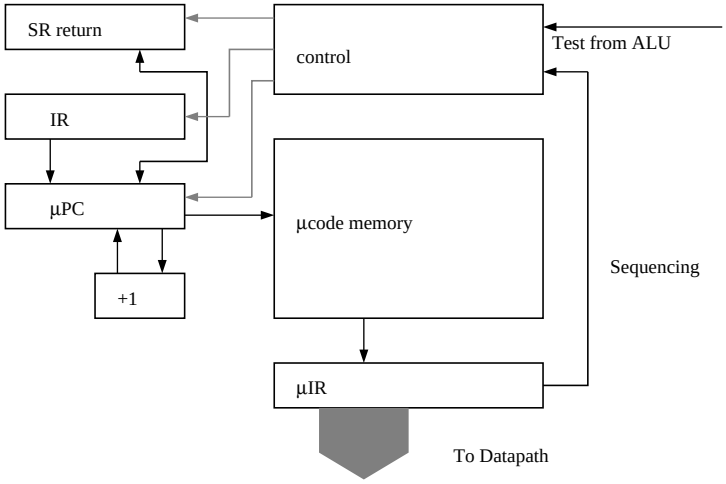

Fig. 3. The Controller.

6.3. Formal Specification of DP and CT. We now consider the formal specification of DP and CT as iterated maps. We wish to construct an algebra

$$(Ct, Dp, Mem, S \mid CT, DP, SMEM),$$

where Ct is the state of the controller CT, Dp is the state of the datapath DP, Mem is the memory as in §5, S is the system clock which is faster than instruction clock T, and CT, DP and $SMEM$ are iterated maps representing CT, DP and memory over clock S.

$$CT : S \times Ct \times Dp \times Mem \to Ct,$$
$$DP : S \times Ct \times Dp \times Mem \to Dp,$$
$$SMEM : S \times Ct \times Dp \times Mem \to Mem,$$

are defined by

$$CT(0, c, d, m) = c,$$
$$DP(0, c, d, m) = d,$$
$$SMEM(0, c, d, m) = m,$$
$$CT(s+1, c, d, m) = ct(CT(s, c, d, m), DP(s, c, d, m), SMEM(s, c, d, m)),$$
$$DP(s+1, c, d, m) = dp(CT(s, c, d, m), DP(s, c, d, m), SMEM(s, c, d, m)),$$
$$SMEM(s+1, c, d, m) = smem(CT(s, c, d, m), DP(s, c, d, m), SMEM(s, c, d, m)).$$

To proceed, we must define state sets Ct and Dp, as well as next-state functions ct, dp and $smem$.

6.4. Datapath and Controller State Sets. First, we consider Dp. Observe in fig. 2 that the state of DP consists of six registers of size A (ACC, PC, MAR, MBR, IR and ALUBUF), and the single-bit register L. Additionally, there is a single-bit $ALU\ test$ register. Hence:

$$Dp = A \times A \times A \times A \times A \times Bit \times A \times Bit.$$

Now, we consider Ct. Observe in fig. 3 that the state of CT consists of a microprogram counter μPC, a subroutine return address register SR, and the microcode memory. Let $\mu PC = W_8$, let $\mu IR = W_{36}$, and let $cntrl = W_{24}$ be the control part of the microinstruction word. Let $[\mu PC \to \mu IR]$ be the microprogram memory. However, not all possible microprogram words are meaningful: for example, it is only possible for a single register to write to a particular bus at one time. We restrict the allowable combinations of bits in a microinstruction as follows. First, we define functions for each bit in the microcode word:

$$MAR.A.wr : \mu IR \to \mathbf{B}, \ldots, BUF.B.wr : \mu IR \to \mathbf{B}$$

These functions are *true* if and only if the corresponding control bit in the current microinstruction word is true (see §6.1 for a list of control bits). We say a set B of bits in a microinstruction is *disjoint* if and only if at most one of the corresponding functions $MAR.A.wr : \mu IR \to \mathbf{B}$,...,$BUF.B.wr : \mu IR \to \mathbf{B}$ are true for the bits in B.

We can define a number of *mutual exclusion functions*

$$Mx_{abus} : \mu IR \to \mathbf{B}, \ldots, Mx_{mbr} : \mu IR \to \mathbf{B}$$

to restrict the microprogram memory. We define Mx_{abus} below: the remaining four functions Mx_{bbus},..., Mx_{mbr} are similar.

$$Mx_{abus}(\mu ir) = \begin{cases} tt, & \text{if } MAR.A.wr(\mu ir),\ MBR.A.wr(\mu ir), \\ & PC.A.wr(\mu ir),\ ACC.A.wr(\mu ir), \\ & ALUBUF.A.wr(\mu ir) \text{ disjoint;} \\ ff, & \text{otherwise.} \end{cases}$$

We can now define the set of allowable microinstruction memories μM:

$$\begin{aligned} \mu M = \{\mu m \in [\mu PC \to \mu IR] \mid \forall \mu pc \in \mu PC, \\ & Mx_{abus}(\mu m(\mu pc)) \text{ and } Mx_{bbus}(\mu m(\mu pc)) \text{ and} \\ & Mx_{acc}(\mu m(\mu pc)) \text{ and } Mx_{pc}(\mu m(\mu pc)) \text{ and} \\ & Mx_{mbr}(\mu m(\mu pc))\} \end{aligned}$$

We can now define Ct:

$$Ct = \mu PC \times \mu PC \times \mu M.$$

We construct the abstract circuit design (see §3.1) of the computer $\mu COMP$ in the manner of §3.4. We may define $\mu COMP : S \times Ct \times Dp \times Mem \to Ct \times Dp \times Mem$ as follows:

$$\mu COMP(s, c, d, m) = (CT(s, c, d, m), DP(s, c, d, m), SMEM(s, c, d, m)).$$

It remains to define the next-state functions dp, ct and $smem$ of the datapath, controller and memory.

6.5. Datapath Next-State Function. We wish to define the datapath next state function

$$dp : Ct \times Dp \times Mem \to Dp.$$

Recall fig. 2: the datapath transfers data (including ALU operations results) between registers and memory via two busses A and B. A simple and convenient way to model this is via two functions *dpwr* which transfers data *to* the two busses, and *dprd* which transfers data *from* the busses. We proceed as follows. First we define a new state-set Dp', which extends Dp by adding the two busses A and B:

$$Dp' = Dp \times La \times La,$$

where

$$Dp = A \times A \times A \times A \times A \times Bit \times La \times Bit.$$

Recall from §3.5 that $La = W_{13}$. Then we define *dpwr* and *dprd*:

$$dpwr : Ct \times Dp' \times Mem \to Dp',$$
$$dprd : Ct \times Dp' \times Mem \to Dp'.$$

We can then define *dp* in terms of *dpwr* and *dprd* as follows:

$$dp(c, d, m) = \pi_d(dprd(c, dpwr(c, \alpha_d(d), m), m)),$$

where $\pi_d : Dp' \to Dp$ is the following projection

$$\pi_d(a, pc, mar, mbr, ir, l, alubuf, test, abus, bbus) = (a, pc, mar, mbr, ir, l, alubuf, test),$$

and $\alpha_d : Dp \to Dp'$ is the padding function

$$\alpha_d(a, pc, mar, mbr, ir, l, alubuf, test) = (a, pc, mar, mbr, ir, l, alubuf, test, \alpha_{d1}, \alpha_{d2}),$$

for arbitrary constants α_{d1}, $\alpha_{d2} \in La$. Observe that $\pi_d(\alpha_d(d)) = d$.

It remains to define *dpwr* and *dprd*. Since busses A and B can potentially operate in parallel, we will define the coordinate functions of *dpwr* and *dprd* separately.

We introduce two groups of functions to manipulate bit vectors. The *trimming functions* $trim_{W_i}^{W_j} : W_j \to W_i$, $j \geq i$, are defined by

$$trim_{W_i}^{W_j}(a_1, \ldots, a_j) = (a_{j-i}, \ldots, a_j).$$

The *padding functions* $pad_{W_i}^{W_j} : W_j \to W_i$, $j \leq i$, are defined by

$$pad_{W_i}^{W_j}(a_1, \ldots, a_j) = (0, \ldots, 0, a_1, \ldots, a_j).$$

Since *dpwr* is concerned with writing from the registers to busses A and B, the contents of a,...,*test* remain unchanged. Hence we just project out the appropriate elements of the state of the extended datapath $d' \in Dp'$.

$$dpwr_a(c, a, pc, mar, mbr, ir, l, alubuf, test, abus, bbus, m) = a,$$
$$\vdots \qquad\qquad \vdots$$
$$dpwr_{test}(c, a, pc, mar, mbr, ir, l, alubuf, test, abus, bbus, m) = test.$$

We define the coordinate functions for $abus$ and $bbus$ as follows.

$$dpwr_{abus}(\mu pc, sr, \mu m, a, pc, mar, mbr, ir, l, alubuf, test, abus, bbus, m) =$$
$$\begin{cases} pad_A^{La}(mar), & \text{if } MAR.A.wr(\mu m(\mu pc)); \\ pad_A^{La}(mbr), & \text{if } MBR.A.wr(\mu m(\mu pc)); \\ pad_A^{La}(pc), & \text{if } PC.A.wr(\mu m(\mu pc)); \\ pad_A^{La}(a), & \text{if } ACC.A.wr(\mu m(\mu pc)); \\ alubuf, & \text{if } ALUBUF.A.wr(\mu m(\mu pc)); \\ abus, & \text{otherwise,} \end{cases}$$

$$dpwr_{bbus}(\mu pc, sr, \mu m, a, pc, mar, mbr, ir, l, alubuf, test, abus, bbus, m) =$$
$$\begin{cases} pad_A^{La}(ir), & \text{if } IR.B.wr(\mu m(\mu pc)); \\ pad_A^{La}(a), & \text{if } ACC.B.wr(\mu m(\mu pc)); \\ alubuf, & \text{if } ALUBUF.B.wr(\mu m(\mu pc)); \\ bbus, & \text{otherwise.} \end{cases}$$

We can define $dprd$ in a similar way, except now $abus$ and $bbus$ remain unchanged, and a—$test$ may be modified, depending on the current microinstruction word.

$$dprd_{abus}(c, a, pc, mar, mbr, ir, l, alubuf, test, abus, bbus, m) = abus,$$
$$dprd_{bbus}(c, a, pc, mar, mbr, ir, l, alubuf, test, abus, bbus, m) = bbus.$$

We omit the definitions of the five functions $dprd_{pc}, \ldots, dprd_l$, which are similar to $dprd_a$ below.

$$dprd_a(\mu pc, sr, \mu m, a, pc, mar, mbr, ir, l, alubuf, test, abus, bbus, m) =$$
$$\begin{cases} trim_A^{La}(abus), & \text{if } ACC.A.rd(\mu m(\mu pc)); \\ trim_A^{La}(bbus), & \text{if } ACC.B.rd(\mu m(\mu pc)); \\ a, & \text{otherwise.} \end{cases}$$

$$dprd_{alubuf}(\mu pc, sr, \mu m, a, pc, mar, mbr, ir, l, alubuf, test, abus, bbus, m) =$$
$$\begin{cases} aluop(abus, bbus, \mu m(\mu pc)), & \text{if } doaluop(\mu m(\mu pc)); \\ alubuf, & \text{otherwise.} \end{cases}$$

$$dprd_{test}(\mu pc, sr, \mu m, a, pc, mar, mbr, ir, l, alubuf, test, abus, bbus, m) =$$
$$\begin{cases} alutest(abus, bbus, \mu m(\mu pc)), & \text{if } doaluop(\mu m(\mu pc)); \\ test, & \text{otherwise.} \end{cases}$$

Function $doaluop : \mu IR \to \mathbf{B}$ is true if any of those bits in the control word that control the ALU are set. The function $aluop : La \times La \times \mu IR \to La$ performs the operation indicated by a control word on the contents of the A and B busses. Similarly, the function $alutest : La \times La \times \mu IR \to Bit$ sets the ALU test bit based on the result of an ALU operation.

6.6. Controller Next-State Function. We now consider the controller next state function

$$ct : Ct \times Dp \times Mem \to Ct.$$

Recall that the state of the controller is

$$Ct = \mu PC \times \mu PC \times \mu M.$$

The following functions are used to divide up the microinstruction word. The address portion of the microinstruction word $addr : \mu IR \to \mu PC$ is defined by

$$addr(i) = trim^{\mu IR}_{\mu PC}(i),$$

The control portion of the microinstruction word $control : \mu IR \to cntrl$ is defined by

$$control(i) = trim^{\mu IR}_{cntrl}(bits^{\mu IR}_{1/24}(i)).$$

The *next instruction bit* of the microinstruction word $nexti : \mu IR \to \mathbf{B}$ is defined by

$$nexti(i) = \begin{cases} tt, & \text{if } bits^{\mu IR}_{25/25}(i) = 1; \\ ff, & \text{if } bits^{\mu IR}_{25/25}(i) = 0. \end{cases}$$

Similarly for *jump*, *subrout* and *cond* (bits 26, 27 and 28).

$ct(\mu pc, sr, \mu m, a, pc, mar, mbr, ir, l, alubuf, test, m) =$

$$\begin{cases} (\mu pc + 1, sr, \mu m), & \text{if } nexti(\mu m(\mu pc)) = ff \text{ and} \\ & (subrout(\mu m(\mu pc)) = ff \text{ and } jump(\mu m(\mu pc)) = ff) \text{ or} \\ & ((subrout(\mu m(\mu pc)) = tt \text{ or } jump(\mu m(\mu pc)) = tt \text{ and} \\ & cond(\mu m(\mu pc)) = tt \text{ and } test(s) = ff); \quad (i) \\ (addr(\mu m(\mu pc)), sr, \mu m), & \text{if } nexti(\mu m(\mu pc)) = ff \text{ and} \\ & subrout(\mu m(\mu pc)) = ff \text{ and} \\ & (jump(\mu m(\mu pc)) = tt \text{ and } cond(\mu m(\mu pc)) = ff \text{ or} \\ & jump(\mu m(\mu pc)) = tt \text{ and } cond(\mu m(\mu pc)) = tt \text{ and} \\ & test(s) = tt); \quad (ii) \\ (addr(\mu m(\mu pc)), pc + 1, \mu m), & \text{if } nexti(\mu m(\mu pc)) = ff \text{ and} \\ & jump(\mu m(\mu pc)) = ff \text{ and} \\ & (subrout(\mu m(\mu pc)) = tt \text{ and } cond(\mu m(\mu pc)) = ff \text{ or} \\ & subrout(\mu m(\mu pc)) = tt \text{ and } cond(\mu m(\mu pc)) = tt \text{ and} \\ & test(s) = tt); \quad (iii) \\ (sr, sr, \mu m), & \text{if } nexti(\mu m(\mu pc)) = ff \text{ and} \\ & jump(\mu m(\mu pc)) = tt \text{ and } subrout(\mu m(\mu pc)) = tt; \quad (iv) \\ (decode(ir), sr, \mu m) & \text{if } nexti(\mu m(\mu pc)) = tt; \quad (v) \end{cases}$$

Each case is explained below.

(i) If the microinstruction is not a branch or subroutine (or is a failed conditional branch or subroutine), and not the start of the next machine instruction, increment μpc.

(ii) If the microinstruction is an unconditional branch, or a successful conditional branch, set μpc to the *addr* field of the microinstruction.

(iii) If the microinstruction is an unconditional subroutine call, or a successful conditional subroutine call, set μpc to the *addr* field of the microinstruction, and store the old value of $\mu pc + 1$ in sr.

(iv) If both the *jump* and *subrout* fields are set, this is a subroutine return, so set μpc to sr.

(v) This is the start of the next machine instruction, so set μpc to *decode* of the current instruction, where $decode : A \to \mu PC$ is a function we will not define, but which maps machine instructions to some starting address in the microinstruction memory.

6.6.1. Microprogram Memory. To complete the specification, we should define the contents of the microprogram memory. This is not possible here, because it would be too long. However, we present a sample implementation of the AND instruction using page zero indirect addressing. A pseudo-assembly language format is used, where the first column contains address labels, the second column contains the control signals to be activated, and the third column contains control and next address information. Each line represents the activity occurring during a single cycle of the system clock S. Parallel activation of multiple control signals a and b is indicated by $a \mid b$, and a blank third column indicates that the next sequential microinstruction is to be executed after the current one.

AND0i:	IR.B.wr|ALU.comp-page-zero|PC.inc	Subroutine ind-addr
	MBR.A.wr|ACC.B.wr|ALU.AND	
	ACC.B.rd|BUF.B.wr|PC.A.wr|MAR.A.rd	
	IR.data.rd|MEM.rd	next instruction
	⋮	
ind-addr:	MAR.A.rd|BUF.A.wr	
	MBR.data.rd|MEM.rd	
	MAR.A.rd|MBR.A.wr	
	MBR.data.rd|MEM.rd	return

6.6.2. Execution Time Bounds. Eight microinstructions are required to execute this particular instruction. As a minimal condition on an implementation of a microprocessor, we require all microinstruction sequences implementing machine instructions to terminate. We can define the *execution time bound function*

$$Ex : Cpu \times Mem \to S$$

as an upper bound on the number of cycles of clock S required for each machine instruction. In the case of the single instruction defined above

$$Ex(a, pc, l, m) =$$

$$\begin{cases} \vdots & \vdots \\ 8, & \text{if } op(m(pc)) = 0 \text{ and } pgbit(m(pc)) = \mathit{ff} \text{ and } indir(m(pc) = tt; \\ \vdots & \vdots \end{cases}$$

In the example above, there is a constant bound on the number of cycles of S. In general, this will not always be the case. A weaker condition is that Ex be *primitive recursive*. We can postulate machine instructions that are terminating but will not have a primitive recursive bound on the number of system clock cycles required: consider, for example, a machine instruction for Ackerman's function. However, such instructions will be extremely rare in practice.

6.7. Memory Next-State Function. Finally, we consider the memory next state function

$$smem : Ct \times Dp \times Mem \to Mem.$$

We proceed in a manner similar to that used for the *mem* next state function in §5.3. New values to be stored in memory are held in the memory buffer register *mbr*. The addresses at which they are to be stored are held in the memory address register *mar*. We define *smem* as follows:

$$smem(\mu pc, sr, \mu m, a, pc, mar, mbr, ir, l, alubuf, test, m) =$$

$$\begin{cases} m, & \text{if } \neg MBR.data.wr(\mu m(\mu pc)); \\ m[mbr/mar], & \text{if } MBR.data.wr(\mu m(\mu pc)). \end{cases}$$

7. CORRECTNESS OF THE IMPLEMENTATION.

Recall §3.6. For $\mu COMP$ (§6.3) to be a correct implementation of $COMP$ the following diagram must commute:

$$\begin{array}{ccc} T \times Cpu \times Mem & \xrightarrow{COMP} & Cpu \times Mem \\ \downarrow \phi & & \uparrow \pi \\ S \times Ct \times Dp \times Mem & \xrightarrow{\mu COMP} & Ct \times Dp \times Mem \end{array}$$

where $\pi : Ct \times Dp \times Mem \to Cpu \times Mem$ is the following projection function

$$\pi(\mu pc, sr, \mu m, a, pc, mar, mbr, ir, l, alubuf, alutest, m) = (a, pc, l, m),$$

and $\phi : T \times Cpu \times Mem \to S \times Ct \times Dp \times Mem$ is defined by

$$\phi(t, c, m) = (\overline{\lambda}(t, c, m), \alpha_x(c, m)),$$

where $\alpha_x : Cpu \times Mem \to Ct \times Dp \times Mem$ is a function which pads Cpu to construct $Ct \times Dp$. Recall from §3.6 that the contents of $Ct \times Dp$ not in Cpu must be either invariant between instructions, or will be the result of the execution of a previous instruction. In either case, we may pad Cpu with constants: in the case of invariant elements (in this case, the microcode memory) the precise value of the constants is important; in other cases, arbitrary values are sufficient. The function α_x is defined by

$$\alpha_x(a, pc, l, m) = (x_{\mu pc}, x_{sr}, x_{\mu m}, a, pc, x_{mar}, x_{mbr}, x_{ir}, l, x_{alubuf}, x_{alutest}, m),$$

with $x_{\mu pc} \in \mu PC$, $x_{sr} \in \mu PC$, $x_{\mu m} \in \mu M$, $x_{mar} \in A$, $x_{mbr} \in A$, $x_{ir} \in A$, $x_{alubuf} \in A$, $x_{alutest} \in A$. Note that the microcode memory is represented by $x_{\mu m}$.

It remains to define $\overline{\lambda} : T \times Cpu \times Mem \to S$. Recalling §2.7, we observe that $\overline{\lambda}$ is the immersion of a state-dependent retiming $\lambda : S \times Cpu \times Mem \to T$. We will define $\overline{\lambda}$ by constructing the corresponding retiming λ, and taking $\overline{\lambda}$ to be the immersion of λ.

We define $XTIME : Cpu \times Mem \to [T \to \mathbf{N}^+]$, so that

$$\lambda(s, c, m) = L(XTIME(c, m))(s),$$

where $L : [T \to \mathbf{N}^+] \to Ret(S, T)$ is defined in §2.7. $XTIME$ generates a stream of natural numbers, such that given a starting state $(c, m) \in Cpu \times Mem$, $XTIME(c, m)(t)$ represents the number of system clock cycles required to execute the current instruction at time t. $XTIME$ is defined by

$$XTIME(c, m)(t) = xtime(COMP(t - 1, c, m)).$$

The time taken for the instruction executed at time t depends on the state of programmer's model $COMP$ at time $t - 1$. Function $xtime : Cpu \times Mem \to \mathbf{N}^+$ determines how many cycles of clock S the next instruction in state $c \in Cpu$, $m \in Mem$ will take to be executed by the abstract circuit design $\mu COMP$.

$$xtime(c, m) = (\mu\ s \leq Ex(c, m))[\pi_{next}(\mu COMP(s, \alpha_x(c), m))],$$

where $\pi_{next} : Ct \times Dp \times Mem \to \mathbf{B}$ is a function that determines when execution of the current instruction is complete, by examining the appropriate part of the microinstruction word, and is defined by

$$\pi_{next}(\mu pc, sr, \mu m, a, pc, mar, mbr, ir, l, alubuf, test, m) = nexti(\mu m(\mu pc)),$$

where $nexti$ is defined in §6.2. and $Ex : Cpu \times Mem \to S$ is the *execution time bound function* defined in §6.6.

8. OUTLINE VERIFICATION.

Given the correctness conditions defined in §7, we now outline the verification of the Abstract Circuit Model description $\mu COMP$ with respect to the Programmer's Model description $COMP$. The proof is too long to include in full. However, we will give sufficient detail to allow the reader to complete the process.

A key feature is that $COMP$ and $\mu COMP$ are iterated maps because it allows us to simplify considerably the verification. First, we will add several concepts and results about iterated maps that will be used to organise the proof. Then we show how these concepts can be used in verifying implementations of iterated map systems in general. Finally we sketch the verification of $COMP$. For a study of the general verification process, and in particular the proofs of Lemmas 8.1.1, 8.1.2, and 8.1.3, see Harman and Tucker [1994b].

8.1. Time-Consistent Iterated Maps. A function $F : T \times A \to A$ is *time-consistent* if, and only if,

$$F(t_1 + t_2, a) = F(t_1, F(t_2, a)).$$

8.1.1. Lemma. *If F is of the form*

$$F(t, a) = f^t(a)$$

then F is time-consistent.

8.2. Uniform Retimings. Given a time-consistent function $F : T \times A \to A$, a state-dependent retiming $\lambda : S \times A \to T$, with immersion $\overline{\lambda} : T \times A \to S$ is said to be *uniform* if and only if $\overline{\lambda}$ is of the form

$$\begin{aligned} &\overline{\lambda}(0, a) = 0, \\ &\overline{\lambda}(t + 1, a) = h(F(t, a)) + \overline{\lambda}(t, a). \end{aligned}$$

Informally, the number of cycles of clock S corresponding with any cycle $t \in T$ is independent of the actual value of t, and is solely a function of the state of F at time t, given starting state a.

8.2.1. Lemma. *Given $F : T \times A \to A$ a time-consistent function, and $\lambda : S \times A \to T$ a uniform retiming, then*

$$F(\overline{\lambda}(t_1 + t_2, a), a) = F(\overline{\lambda}(t_1, F(\overline{\lambda}(t_2, a), a)), F(\overline{\lambda}(t_2, a), a)).$$

8.2.2. Lemma. *Retiming λ in §7 is uniform.*

8.3. Verification Process. Given iterated maps $F : T \times A \to A$, and $G : S \times B \to B$, and appropriate maps $\overline{\lambda}$, ϕ and ψ (see §2.6.2), we require the following diagram to commute.

$$\begin{array}{ccc} T \times A & \xrightarrow{F} & A \\ \downarrow{\scriptstyle (\overline{\lambda}, \phi)} & & \uparrow{\scriptstyle \psi} \\ S \times B & \xrightarrow{G} & B \end{array}$$

Equivalently, for all $t \in T$, $a \in A$, we require

$$F(t, a) = \psi(G(\overline{\lambda}(t, a), \phi(a))).$$

Our verification strategy will be induction over time. First we must prove the base case:

$$F(0, a) = \psi(G(0, \phi(a))). \tag{1}$$

(Recall from §2.7 that $\lambda(0) = 0$, and hence $\overline{\lambda}(0) = 0$.) The correctness of (1) should be straightforward to establish: expanding the definitions of F and G, (1) reduces to

$$a = \psi(\phi(a)).$$

The induction hypothesis is

$$F(t, a) = \psi(G(\overline{\lambda}(t, a), \phi(a))), \tag{2}$$

and hence, the induction step is

$$F(t + 1, a) = \psi(G(\overline{\lambda}(t + 1, a), \phi(a))). \tag{3}$$

In addition, we have the following two identities, from §8.1 and §8.2:

(i) Given that F is an iterated map, then F is time-consistent and hence:

$$F(t_1 + t_2, a) = F(t_1, F(t_2, a)). \tag{4}$$

(ii) Given that G is an iterated map, and hence is time-consistent, and λ is a uniform retiming, then:

$$G(\overline{\lambda}(t_1 + t_2, a), \phi(a)) = G(\overline{\lambda}(t_1, \psi(G(\overline{\lambda}(t_2, a), \phi(a)))), \phi(\psi(G(\overline{\lambda}(t_2, a), \phi(a))))). \tag{5}$$

8.3.1. Lemma. *Given iterated maps F and G, and uniform retiming λ, then the following are equivalent.*

(i) For all $t \in T$, $a \in A$,

$$F(t, a) = \psi(G(\overline{\lambda}(t, a), \phi(a)).$$

(ii) For all $a \in A$,

$$F(0, a) = \psi(G(0, \phi(a)),$$

and

$$F(1, a) = \psi(G(\overline{\lambda}(1, a), \phi(a)). \tag{6}$$

Proof. Trivially, (i) implies (ii). To show that (ii) implies (i), we prove the induction step (3) above. Starting with the left hand side,

$$\begin{aligned} F(t + 1, a) &= F(t, F(1, a)), && \text{using (4)} \\ &= F(t, \psi(G(\overline{\lambda}(1, a), \phi(a)))), && \text{using (6)} \\ &= \psi(G(\overline{\lambda}(t, G(\overline{\lambda}(1, a), \phi(a)))), \phi(\psi(G(\overline{\lambda}(1, a), \phi(a))))) && \text{using (2)} \end{aligned}$$

Now considering the right-hand side,

$$\psi(G(\overline{\lambda}(t + 1, a), \phi(a))) = \psi(G(\overline{\lambda}(t, G(\overline{\lambda}(1, a), \phi(a)))), \phi(\psi(G(\overline{\lambda}(1, a), \phi(a))))). \quad \text{using (5)}$$

8.3.2. Corollary. Given iterated map representations of a microprocessor F and G at different levels of abstraction, with clocks T and S related by uniform retiming λ, to verify that G implements F it is sufficient to show that F and G are equivalent at times $t = s = 0$ and $t = 1$, $s = \lambda(1, a)$.

8.4. Verification. From §8.3 we can see that to verify that $\mu COMP$ correctly implements $COMP$, we must show that for any state $(c, m) \in Cpu \times Mem$ of CPU, that

$$COMP(0, c, m) = \pi(\mu COMP(\phi(0, c, m))), \tag{1}$$

and that

$$COMP(1, c, m) = \pi(\mu COMP(\phi(1, c, m))). \tag{2}$$

Substituting in (1) for $COMP$ and $\mu COMP$, we obtain

$$(c, m) = \pi(\alpha_x(c, m)).$$

Inspection of the definitions of π and α_x (§7) shows this to be clearly true. To prove (2), we observe that the actions of $COMP$ and $\mu COMP$ are dictated by the values of the *opcode* $op : A \to \mathbf{N}$, the *indirection bit* $indir : A \to \mathbf{B}$, and the *page bit* $pgbit : A \to \mathbf{B}$ (§5.2), giving 32 possible outcomes. In the case that the opcode is two (Increment and Skip if Zero), the instruction may either increment the program counter by one or two, giving a further four cases, for a total of 36. To prove (2), we need to consider each of these 36 cases. For problems of this size, manual verification is quite practical (though of course subject to human error). For more complex processors, machine support is required. Given the multiple-case structure of this example, and of other microprocessors, appropriately-directed machine verification should be quite practical, even with large examples.

Consider the example instruction of §6.6.1: the AND instruction with indirect page zero addressing. In this case, $op(m(pc)) = 0$, $indir(m(pc)) = tt$, and $pgbit(m(pc)) = ff$. Evaluating $COMP(1, a, pc, l, m)$:

$$\begin{aligned} COMP(1, a, pc, l, m) &= comp(a, pc, l, m), \\ &= (a \wedge mval(pc, m), pc + 1, l, m), \\ &= (a \wedge m(maddr(pc, m)), pc + 1, l, m), \\ &= (a \wedge m(pgoff(m(pc))), pc + 1, l, m). \end{aligned}$$

Evaluating $\pi(\mu COMP(\phi(1, a, pc, l, m)))$:

$$\begin{aligned} \pi(\mu COMP(\phi(1, a, pc, l, m))) &= \pi(\mu COMP(\overline{\lambda}(1, a, pc, l, m), \alpha_x(a, pc, l, m))), \\ &\vdots \end{aligned}$$

To complete this example, and the remainder of the verification, it would be necessary to define the remaining functions (for example, $aluop : La \times La \times \mu IR \to La$ from §6.5) and the remainder of the microcode.

9. ADDING INPUT-OUTPUT.

The example processor considered in this paper has no provision for input-output; this is clearly unrealistic. Adding input-output to our model however is not difficult. Let T be a clock, A be a set of states, $[T \to W]$ be a set of *input streams*, and X be a set of *output values*. We model a computer with inputs and outputs:

$$\begin{aligned} &F : T \times A \times [T \to W] \to A \times X, \\ &F_1(0, a, w) = a, \\ &F_1(t + 1, a, w) = f(F_1(t, a, w), w(t)), \\ &F_2(t, a, w) = out(F_1(t, a, w)). \end{aligned}$$

The output of F has two components, representing the state and output at time t, with initial state a and input stream w. We introduce a new function $out : A \to X$ to compute the output at time t: a function of the state at time t. Typically, out will be relatively simple; for example, projecting out part of the state. The next state function $f : A \times W \to A$ at time $t+1$ is now a function not only of the state of the processor at time t, but also of any input that arrived at time t.

The correctness condition (§7) is now as follows. Given clocks T and S, state sets A and B, input stream sets $[T \to W]$ and $[S \to Y]$, and output sets X and Z, we say iterated map $F : T \times A \times [T \to W] \to A \times X$ is correctly implemented by iterated map $G : S \times B \times [S \to Y] \to B \times Z$ if the following diagram commutes:

$$\begin{array}{ccc} T \times A \times [T \to W] & \xrightarrow{F} & A \times X \\ \Big\downarrow (\overline{\lambda}, \phi, sch) & & \Big\uparrow (\psi, \chi) \\ S \times B \times [S \to Y] & \xrightarrow{G} & B \times Z \end{array}$$

for state-dependent retiming immersion $\overline{\lambda} : T \times A \times [T \to W] \to S$, padding function $\phi : A \to B$, projection function $\psi : B \to A$, and output map $\chi : Z \to X$. Recall the definition of sch from §2.7.

For $t \in T$, $a \in A$ and $w \in [T \to A]$, we require

$$F(t, a, w) = \psi(G\overline{\lambda}(t, a, w), \phi(a), sch(\lambda(a, w))(w)).$$

Clearly, we require lemma 8.3.1 to still hold. This is in fact the case: see Harman and Tucker [1994a]. In addition, we must show

$$F_2(t, a, w) = \chi(G_2(\overline{\lambda}(t, a, w), \phi(a), sch(\lambda(a, w))(w)))$$

in order to complete the verification. Given that the output functions are generally simple, this will usually be straightforward to establish: see also Harman and Tucker [1994a].

The mathematical modeling can be carried through. However the logical complexity of the models, and of any reasoning about the models and specifications, has increased: the streams turn our models and specifications into higher equational models and specifications.

10. FURTHER CONSIDERATIONS.

We have shown how computers may be algebraically modeled as iterated maps at different levels of abstraction, and defined what it means for a lower-level algebraic model to correctly implement a higher-level specification. Additionally, we have applied our algebraic tools to a case study.

The use of algebras provides a mathematically clear approach to the modular description of computers and, hence, to the analysis of their structure. By lemma 2.5.1 all of these algebraic models can be equationally specified by using initial algebra semantics. The study of both the algebraic models, their equational specifications, and modularisation principles must be taken further.

Verification, as well as input-output, will be addressed in more detail in a further paper (Harman and Tucker [1994a]). Additionally, it is necessary to consider features of real machines that have been omitted from this case study. Examples include exceptions and pipelined implementations. The process of modeling and verification of the components of the computer can continue with DP and CT, each of which may be further subdivided.

The methods and tools presented have a number of advantages over other techniques. Primarily, our algebraic tools are independent of specific machine-based languages, theorem provers and proof assistants, while at the same time being encodable in a range of such tools. Secondly, we emphasise models of explicit time at multiple levels of abstraction.

Our models of time give rise to proof obligations that are simple in comparison with those of other attempts at microprocessor verification, allowing more complex verification case studies to be attempted.

The algebraic tools will form part of the foundations of a machine-readable language for describing microprocessors (and other hardware and physical systems) in a modular way. The language will be developed as part of the Esprit Working Group NADA (No 00 85 33).

11. REFERENCES.

F Anceau, *The Architecture of Microprocessors*, Addison-Wesley, (1986).

M Barbacci, "An Introduction to ISPS", pp.23 — 32 in *Computer Structures: Principles and Examples*, ed. D P Siewiorek, C G Bell, A Newell, (1982).

C G Bell, J C Mudge, and J E McNamara, *Computer Engineering: A DEC View of Hardware Systems Design*, Digital Press, (1978).

G Birtwistle and B Graham, "Verifying SECD in HOL", pp.129 — 177 in *Formal Methods for VLSI Design*, ed. J Staunstrup, North-Holland, (1990).

B Bose and S D Johnson, "DDD-FM9001: Derivation of a Verified Microprocessor", pp.191 — 202 in *Correct Hardware Design and Verification Methods*, ed. G Milne, L Pierre, Lecture Notes in Computer Science 683, Springer-Verlag, (1993).

R S Boyer and J S Moore, *A Computational Logic Handbook*, Academic Press, (1988).

J Chazarain and H Collavizza, "Combining Symbolic Evaluation and Object-Oriented Approach for Verifying Processor-Like Architectures at the RT Level", pp.109 — 121 in *Correct Hardware Design and Verification Methods*, ed. G Milne, L Pierre, Lecture Notes in Computer Science 683, Springer-Verlag, (1993).

A Cohn and M Gordon, "A Mechanized Proof of Correctness of a Simple Counter", pp.65 — 96 in *Theoretical Foundations for VLSI Design*, ed. K McEvoy and J V Tucker, Cambridge University Press Tracts in Theoretical Computer Science 10, (1990).

A Cohn, "A Proof of Correctness of the VIPER Microprocessor: the First Levels", pp.27 — 72 in *VLSI Specification, Verification and Synthesis*, ed. G Birtwistle and P A Subrahmanyam, Kluwer Academic Publishers, (1987).

W J Cullyer, "Implementing Safety Critical Systems: the Viper Microprocessor", pp.1 — 26 in *VLSI Specification, Verification, and Synthesis*, ed. G Birtwistle and P A Subrahmanyam, Kluwer Academic Publishers, (1987).

W J Cullyer, "Application of Formal Methods to the VIPER Microprocessor", IEE Proceedings, 134 E, 3, pp.133 — 141 (May 1987).

H Ehrig and B Mahr, *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*, EATCS Monograph vol. 6, Springer-Verlag, (1985).

J S Florentin, *Microprogrammed Systems Design*, Macmillan, (1991).

A Geser, "A Specification of the Intel 8085 Microprocessor: A Case Study.", pp.347 — 402 in *Algebraic Methods: Theory, Tools and Applications*, ed. M Wirsing and J A Bergstra, Lecture Notes in Computer Science 394, Springer-Verlag, (1989).

J A Goguen and T Winkler, "Introducing *OBJ3*", Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, Menlo Park, California, (1988).

M Gordon, "LCF-LSM, a System for Specifying and Verifying Hardware", Technical Report No. 41, Computer Laboratory, University of Cambridge, (1983).

M Gordon, "Proving a Computer Correct with the LCF-LSM Hardware Verification System", Technical Report No. 42, Computer Laboratory, University of Cambridge, (1983).

M Gordon, "HOL: A Proof Generating System for Higher-Order Logic", pp.73 — 128 in *VLSI Specification, Verification and Synthesis*, ed. G Birtwistle and P A Subrahmanyam, Kluwer Academic Publishers, (1987).

B Graham and G Birtwistle, "Formalising the Design of an SECD Chip", pp.40 — 66 in *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, ed. M Leeser and G Brown, Lecture Notes in Computer Science 408, Springer Verlag, (1990).

B Graham, *The SECD Microprocessor: a Verification Case Study*, Kluwer, (1992).

K Hanna and N Daeche, "Strongly-Typed Theory of Structures and Behaviours", pp.39 — 54 in *Correct Hardware Design and Verification Methods*, ed. G Milne, L Pierre, Lecture Notes in Computer Science 683, Springer-Verlag, (1993).

N A Harman and J V Tucker, "Clocks, Retimings, and the Formal Specification of a UART", pp.375 — 396 in *The Fusion of Hardware Design and Verification*, ed. G J Milne, North-Holland, (1988).

N A Harman and J V Tucker, "Formal Specification and the Design of Verifiable Computers", pp.500 — 503 in *Proceedings of the 1988 UK IT Conference, University College Swansea*, IEE, (1988).

N A Harman and J V Tucker, "The Formal Specification of a Digital Correlator I: Abstract User Specification", pp.161 — 262 in *Theoretical Foundations for VLSI Design*, ed. K McEvoy and J V Tucker, Cambridge University Press Tracts in Theoretical Computer Science 10, (1990).

N A Harman and J V Tucker, "Consistent Refinements of Specifications for Digital Systems", pp.273 — 295 in *Correct Hardware Design Methodologies*, ed. P Prinetto and P Camurati, North-Holland, (1992).

N A Harman and J V Tucker, "Specification, Design and Verification of a Simple Computer", in preparation, (1994).

N A Harman and J V Tucker, "A Model of Timing Abstraction for Synchronous Digital Hardware", in preparation, (1994).

N A Harman, "Formal Specifications for Digital Systems", Ph.D. Thesis, School of Computer Studies, University of Leeds, (1989).

W A Hunt, "FM8501: A Verified Microprocessor", The University of Texas at Austin Institute for Computing Science technical report 47, (1986).

W A Hunt, "Microprocessor Design Verification", Journal of Automated Reasoning, 5, 4, pp.429 — 460 (1989).

W Hunt, "A Formal HDL and its use in the FM9001 Verification", in *Mechanized Reasoning in Hardware Design*, ed. C A R Hoare and M Gordon, Prentice-Hall, (1992).

W Hunt, *FM8501: A Verified Microprocessor*, Lecture Notes in Artificial Intelligence 795, Springer-Verlag, (1994).

S D Johnson and Z Zhu, "An Algebraic Approach to Hardware Specification and Derivation", in *Applied Formal Methods for Correct VLSI Design*, ed. L Claesen, Elsevier, (1991).

J Joyce, "Formal Verification and Implementation of a Microprocessor", pp.129 — 159 in *VLSI Specification, Verification and Synthesis*, ed. G Birtwistle and P A Subrahmanyam, Kluwer Academic Publishers, (1987).

P Landin, "On the Mechanical Evaluation of Expressions", Computer Journal, 6, pp.308 — 320 (1963).

D May, G Barrett, and D Shepard, "Designing Chips that Work", Philosophical Transactions of the Royal Society A, 339, pp.3 — 19 (1992).

K McEvoy and J V Tucker, "On Theoretical Foundations for Hardware Design", pp.1 — 64 in *Theoretical Foundations for VLSI Design*, ed. K McEvoy and J V Tucker, Cambridge University Press Tracts in Theoretical Computer Science 10, (1990).

K Meinke and J V Tucker, "Universal Algebra", pp.189 — 411 in *Handbook of Logic in Computer Science*, ed. S Abramsky, D Gabbay, T S E Maibaum, Oxford University Press, (1992).

T Melham, "Using Recursive Types to Reason about Hardware in Higher Order Logic", pp.27 — 50 in *The Fusion of Hardware Design and Verification*, ed. G J Milne, North-Holland, (1988).

T F Melham, *Higher Order Logic and Hardware Verification*, Cambridge University Press Tracts in Theoretical Computer Science 31, (1993).

G J Milne, "Timing Constraints: Formalising their Description and Verification", in *Proceedings of Computer Hardware Description Languages and their Applications*, North-Holland, (1989).

G J Milne, "The Formal Description and Verification of Hardware Timing", University of Strathclyde Computer Science Report HDV–8–90, (1990).

W Roscoe, "Occam in the Specification and Verification of Microprocessors", Philosophical Transactions of the Royal Society A, 339, pp.137 — 151 (1992).

W Stallings, *Computer Organisation and Architecture: Principles of Function and Structure*, Macmillan, (1987).

V Stavridou, *Formal Specification of Digital Systems*, Cambridge University Press Tracts in Theoretical Computer Science 37, (1993).

P A Subrahmanyam, "Contextual Constraints, Temporal Abstraction and Observational Equivalence in VLSI Design", pp.159 — 184 in *The Fusion of Hardware Design and Verification*, ed. G J Milne, North-Holland, (1988).

B C Thompson and J V Tucker, "Equational Specification of Synchronous Concurrent Algebras and Architectures", Department of Computer Science CSR 9.91, University College Swansea, (1991).

J V Tucker and J I Zucker, *Program Correctness over Abstract Data Types with Error State Semantics*, North-Holland, (1988).

J V Tucker and J I Zucker, "Generalised Computability and Algebraic Specifications for Abstract Data Types", In preparation, (1993).

J V Tucker, "Theory of Computation and Specification over Abstract Data Types and its Applications", pp.1 — 40 in *Logic, Algebra and Computation*, ed. F L Bauer, Springer, (1991).

W Wechler, "Universal Algebra for Computer Scientists", EATCS Monograph, Springer-Verlag, Berlin, (1991.).

W P Weijland, "Verification of a Systolic Algorithm in Process Algebra", in *Theoretical Foundations for VLSI Design*, ed. K McEvoy and J V Tucker, Cambridge University Press Tracts in Theoretical Computer Science 10, (1990).

P Windley, "A Theory of Generic Intepreters", pp.122 — 134 in *Correct Hardware Design and Verification Methods*, ed. G Milne, L Pierre, Lecture Notes in Computer Science 683, Springer-Verlag, (1993).

M Wirsing, "Algebraic Specification", pp.675 — 788 in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, ed. J van Leeuwen, Elsevier, (1990).

Z Zhu and S D Johnson, "An Example of Interactive Hardware Transformation", Indiana University, Computer Science Department (draft), (1991).