An Instruction Set Process Calculus

Shiu-Kai Chin[1] and Jang Dae Kim[2]

[1] EE/CS Dept., Syracuse University, Syracuse, NY 13244
skchin@syr.edu
[2] National Semiconductor, 2900 Semiconductor Dr., Santa Clara, CA 95052
jdkim@galaxy.nsc.com

Abstract. We have created a calculus for reasoning about hardware and firmware at the algorithmic state machine (ASM) and instruction-set processor (ISP) levels of description. The calculus is a value-passing process algebra that extends the Mealy machine model to include parallel composition. It supports reasoning about the composed behavior of synchronous ASM and ISP components and microcode. We present an overview of the calculus and its application including an example showing the equivalence of a microcoded machine to its target instruction set specified by both ASM and ISP descriptions. The calculus, its properties, and the examples have been deeply embedded, proved, and verified as conservative extensions to the logic of the Higher Order Logic (HOL90) theorem prover.

1 Introduction

The *Instruction-set Process Calculus* (IspCal) is a calculus for describing processes at the algorithmic and instruction-set levels. Its purpose is to support compositional reasoning about the equivalence between *structures* of processes and *behaviors*.

The focus of IspCal on synchronous composition is similar to Milner's SCCS [11], Milne's Circuit Calculus (CIRCAL) [10], and Esterel [3]. IspCal differs from previous work in that it targets **instruction set processor** (ISP) [1] descriptions by having value-passing, delayed assignment of register values, and a semantics tuned to support the Mealy model of computation. This view is consistent with synchronous hardware design. The processes that are described are **algorithmic state machines** (ASM) [4][7] (e.g., hardware flowcharts), or ISP descriptions stating how instructions change registers and outputs.

Our objectives for IspCal are threefold: (1) to make connections to existing computer engineering design models by extending the Mealy machine model of hardware to reason about composed ISP and ASM descriptions; (2) to reason about correctness of microcoded architectures; and (3) to show equivalence between structural hardware and microcode descriptions and ASM or ISP specifications. IspCal describes behavior at a *cycle simulation* level, (i.e., changes occur on a clock-cycle basis).

IspCal consists of a *behavioral* language and a *structural* language. The behavioral language typically is used to describe ASM or ISP-level behavior. It

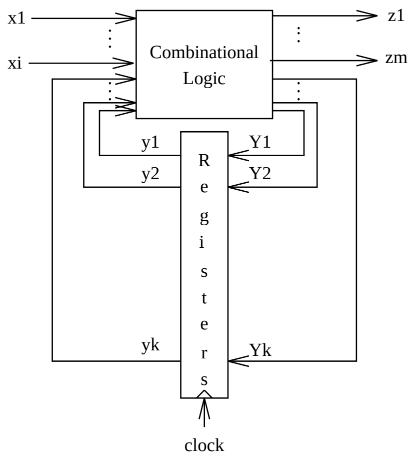

Fig. 1. Finite-State Machine

includes operators such as prefix, summation, guarded expressions, and mutual recursion. The structural language typically is used to describe implementations consisting of composed ASM or ISP processes. It includes operators for parallel composition and hiding.

The syntax and semantics of IspCal, all the equational laws, examples, and proofs have been done as conservative extensions to the higher-order logic of the HOL [6] theorem-prover. The techniques of defining IspCal within HOL are similar to the work done by Nesi [13]. Details of IspCal in HOL and the HOL proofs of the examples are found in `http://www.cat.syr.edu/~chin/`

The rest of this paper is organized as follows. Section 2 describes the Mealy machine model of hardware. Section 3 describes the syntax and semantics of behavioral IspCal expressions while Section 4 describes the syntax and semantics of structural IspCal expressions. A microprogrammed controller is described in Section 5, and a datapath for a CPU is described in Section 6. The controller and datapath are combined in Section 7 to form a CPU. The conclusions are in Section 8.

2 The Mealy Machine Model

The behaviors described by IspCal are an extension to the sequential Mealy machine model [8]. The model sketched in Figure 1 is used widely by computer engineers and circuit designers. Sequential machine behavior is specified as a sequence of events occurring at discrete points in time corresponding to clock cycles.

The machine has inputs $x_1 \ldots x_i$ and outputs $z_1 \ldots z_m$. The *present state* of the machine is given by state variables $y_1 \ldots y_k$. The *next state* of the machine is given by next-state variables $Y_1 \ldots Y_k$. The state register holding $y_1 \ldots y_k$

updates its contents with $Y_1 \ldots Y_k$ when *clock* has an *active* edge. The outputs $z_1 \ldots z_m$ and the next-state variables $Y_1 \ldots Y_k$ are functions of inputs $x_1 \ldots x_i$ and present-state variables $y_1 \ldots y_k$.

More formally, a *synchronous sequential machine* M is described by a 5-tuple

$$M = (I, O, S, \delta, \lambda),$$

where I, O, and S are finite, non-empty sets of *inputs, outputs*, and *states*, $\delta : (I \times S) \rightarrow S$ is a state-transition function taking elements of I and S and returning a state, and $\lambda : (I \times S) \rightarrow O$ is the output function taking elements of I and S as inputs and returning an element of O.

3 Behavioral IspCal

IspCal extends the Mealy machine model by adding several CCS-like [12] [11] operators such as parallel composition, choice, hiding, and value-passing. *Actions* are sets of input, output, and state assignments occurring *simultaneously* within a single clock cycle.

3.1 Syntax

The abstract syntax of behavioral IspCal uses expressions from the following syntactic domains: **BExp**, the set of boolean expressions be; **DExp**, the set of data expressions d; **Dvar**, the set of data identifiers x; **Port**, the set of port names p; **Event**, the set of events e; **BIsp**, the set of behavioral IspCal terms P; and **Var**, the set of process identifiers X.

We take for granted the syntax of data expressions, port names, and process and data identifiers. They are formally defined in our HOL implementation and are omitted here for reasons of space. **Data expressions** include: *Boolean expressions* with operators such as not, and, or, etc.; *indexed arrays*, (e.g. if x is an array then $x[i]$ is the i^{th} element of array x); *arithmetic expressions* with arithmetic operators such as addition, subtraction, multiplication, and division; and *command expressions* corresponding to (micro) instructions. We use the HOL notion for **conditional expressions:** *if A then B else C* is written as $A \rightarrow B \mid C$ where $(true \rightarrow B \mid C) = B$ and $(false \rightarrow B \mid C) = C$.

Event expressions have three forms: $p?x$ (inputs to variables x through ports p); $p!d$ (values d assigned to ports p); and $x := d$ (assignment of data values d to state variables x). Thus their abstract syntax can be given as follows:

$$e ::= p?x \mid p!d \mid x := d$$

Prefix action expressions are finite sets of event expressions.

$$pre ::= \{e_0; e_1; e_2; \ldots; e_k\}$$

The abstract syntax of **process expressions** is as follows:

$$P ::= pre.P \mid P_1 + P_2 \mid be \rightarrow P \mid X \mid Proc\, X\, Pl$$

Process list expressions appear within the definition of recursive process expressions. They are finite sets of expressions $X_i \Longleftarrow P_i$

$$Pl ::= \{X_0 \Longleftarrow P_0; X_1 \Longleftarrow P_1; X_2 \Longleftarrow P_2; \ldots; X_n \Longleftarrow P_n\}$$

We implicitly assume that there is only one definition for each process identifier X_i in the set. Process terms P_i cannot be recursive terms of the form $Proc\ X\ Pl$. As mutual recursion is supported, there is no need for nested recursion.

3.2 Substitution of Process Identifiers

$P[P'/X']$ denotes the term obtained by substituting process term P' for every occurrence of X' in process term P. The substitution rules are defined recursively as follows:

$$\begin{array}{l}
(pre.P)[P'/X'] = pre.(P[P'/X]')\\
(P1 + P2)[P'/X'] = (P1[P'/X']) + (P2[P'/X'])\\
(b \rightarrow P)[P'/X'] = b \rightarrow (P[P'/X'])\\
X[P'/X'] = \begin{cases} P' \text{ if } X = X' \\ X \text{ otherwise} \end{cases}
\end{array}$$

Expanding recursive process terms $Proc\ X\ Pl$ is done by substitution of process expressions P_i into free process identifiers X_i. As nested recursion is prohibited in the syntax, we don't have to specify the case for $Proc\ X\ Pl$ in the definition of process substitution. Multiple parallel (simultaneous) substitution of free process identifiers $(X'_0, X'_1, X'_2, \ldots, X'_m)$ in a behavioral IspCal term P with new process terms $(P'_0, P'_1, P'_2, \ldots, P'_m)$, respectively, is denoted by $P[P'_0/X'_0, P'_1/X'_1, P'_2/X'_2, \ldots, P'_m/X'_m]$.

3.3 Semantics

An inductively defined labeled-transition relation defines the semantics of behavioral IspCal. Inductively defined relations are defined by *rules*. The labeled-transition relations are the least relations satisfying the rules. This corresponds to the way relations are defined inductively in the HOL theorem-prover [6] by Melham [9].

Let V be the set of data values, v. A *store* σ is a mapping from data identifiers x to data values v. That is,

$$\sigma : Var \rightarrow V$$

Each store σ specifies a value, written $\sigma(x)$, for data identifier x. For any store σ, $\sigma[x \mapsto v]$ is the store that agrees with σ except it maps the identifier x to value v.

We assume the evaluation semantics are given for data and Boolean expressions. (This is described in detail as part of our HOL implementation but is omitted here for space reasons). We write $\langle d, \sigma \rangle \longrightarrow^* v$ to indicate that data expression d given store σ evaluates to v. We use similar notation for the evaluation of Boolean expressions. We let $[\![exp]\!]\sigma$ denote the value of exp when evaluated with store σ.

Execution of an assignment statement $x := d$ given store σ results in an update of store σ to $\sigma[x \mapsto v]$ where $v = [\![d]\!]\sigma$. Execution of multiple assignment statements results in multiple updates $\sigma[\ldots, x_i \mapsto v_i, \ldots]$ where $v_i = [\![d_i]\!]\sigma$.

The operational semantics describe how a configuration — a process paired with a store — evolves to another configuration. A store keeps track of bindings of data identifiers of values.

The set of configurations *Cfg* is $Cfg = \{\langle P, \sigma\rangle | P \in BIsp \wedge \sigma \in store\}$ where σ is a store containing free data or state variables in P. The operational semantics is defined by a labeled transition system over process configurations.

An **event** is a member of the set $\{p.v | p \in Port \wedge v \in V\}$ where V is the set of data values. Intuitively, the event $p.v$ represents the presence of the value v on port p.

An **action** is simply a finite set of events $\{p_1.v_1; \ldots; p_n.v_n\}$.

A transition relation $\alpha \stackrel{act}{\Longrightarrow} \beta$ means the configuration α may perform the action *act* and become β. The transition label *act* is a subset of events $Event = \{p.v \mid p \in Port \wedge v \in V\}$. The labeled transition relation $\stackrel{act}{\Longrightarrow}$ is a subset of $Cfg \times Cfg$, where *act* is the transition label.

Four rules define $\Longrightarrow$, a family of relations indexed by actions. They are defined as the least relations satisfying the rules.

1. The prefix rule describes the behavior of a Mealy machine.
$$\frac{\forall i : k < i \leq n.\ \langle d_i, \sigma[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]\rangle \longrightarrow^* v_i \quad \begin{array}{l} \forall i\, j.\ p_i = p_j \supset v_i = v_j \\ \forall i\, j.\ x_i = x_j \supset v_i = v_j \end{array}}{\begin{array}{c}\langle\{p_1?x_1; \ldots; p_k?x_k; p_{k+1}!d_{k+1}; \ldots; p_m!d_m; x_{m+1} := d_{m+1}; \ldots; x_n := d_n\}.P, \sigma\rangle \\ \stackrel{\{p_1.v_1;\ldots;p_m.v_m\}}{\Longrightarrow} \\ \langle P, \sigma[x_{m+1} \mapsto v_{m+1}, \ldots, x_n \mapsto v_n]\rangle\end{array}}$$
Inputs $x_1, \ldots, x_k$ appear on ports $p_1, \ldots, p_k$. Ports $p_{k+1}, \ldots, p_m$ have output values. The expressions determining the output values are $d_{k+1}, \ldots, d_m$. The state variables are $x_{m+1}, \ldots, x_n$. The updated values of $x_{m+1}, \ldots, x_n$ are determined by expressions $d_{m+1}, \ldots, d_n$. Just as in the Mealy machine model, the values of the output and state expressions depend on the input values assigned to $x_1, \ldots, x_k$ and on the values of the state variables $x_{m+1}, \ldots, x_n$ in store σ. Each output and state expression d_i is evaluated given the input values $v_1, \ldots, v_k$ and the values of free state variables in d_i stored in σ. This is represented by $\forall i : k < i \leq n.\ \langle d_i, \sigma[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]\rangle \longrightarrow^* v_i$.
The conditions when $p_i = p_j$ and $x_i = x_j$ mean ports and state variables have only one value at a time. Notice that instantiation of input events does not change the store; only assignment statements can update stores and updates are in effect only after labeled transitions.
2. The second rule defines summation similar to CCS [12]. Intuitively, the process $P_1 + P_2$ can choose to behave like either P_1 or P_2.

$$\frac{\langle P_1,\sigma\rangle \overset{act}{\Longrightarrow} \langle P_1',\sigma'\rangle}{\langle P_1+P_2,\sigma\rangle \overset{act}{\Longrightarrow} \langle P_1',\sigma'\rangle} \qquad \frac{\langle P_2,\sigma\rangle \overset{act}{\Longrightarrow} \langle P_2',\sigma'\rangle}{\langle P_1+P_2,\sigma\rangle \overset{act}{\Longrightarrow} \langle P_2',\sigma'\rangle}$$

3. The third rule defines the behavior of *guarded* expressions.

$$\frac{\langle P,\sigma\rangle \overset{act}{\Longrightarrow} \langle P',\sigma'\rangle \quad \langle be,\sigma\rangle \longrightarrow^{*} true}{\langle be \rightarrow P,\sigma\rangle \overset{act}{\Longrightarrow} \langle P',\sigma'\rangle}$$

4. The final rule defines *expansion* of a recursive process *Proc X Pl*, as similar to the **fix** operator of SCCS [11], when there is exactly one instance of $X \Leftarrow P_X(X_1,\ldots,X_n)$ in Pl defining X. As free occurrences of process identifiers X_i in the body P_X represent recursion, they are replaced by recursive process terms $ProcX_i\,Pl$ in which X_i are the initial process identifiers of the new recursion terms.

$$\frac{\langle P_X[ProcX_1Pl/X_1,\ldots,ProcX_nPl/X_n],\sigma\rangle \overset{act}{\Longrightarrow} \langle P',\sigma'\rangle}{\langle Proc\ X\ Pl,\sigma\rangle \overset{act}{\Longrightarrow} \langle P',\sigma'\rangle} \quad X \text{ is uniquely defined in } Pl \text{ by } X \Leftarrow P_X(X_1,\ldots,X_n)$$

3.4 Bisimulation Equivalence

Two behavioral IspCal configurations $\langle P_1,\sigma_1\rangle$ and $\langle P_2,\sigma_2\rangle$ are equivalent if they are **bisimilar**, written $\langle P_1,\sigma_1\rangle \sim \langle P_2,\sigma_2\rangle$. Our definition of bisimulation corresponds to Milner's [12].

$$\langle P_1,\sigma_1\rangle \sim \langle P_2,\sigma_2\rangle =_{def} \exists\, \mathcal{R}.\ Bisim\ \mathcal{R} \wedge (\langle P_1,\sigma_1\rangle,\langle P_2,\sigma_2\rangle) \in \mathcal{R}$$

where $\mathcal{R} \subseteq Cfg \times Cfg$ identifying bisimilar states or processes. $\mathcal{R}$ is a **bisimulation**, written $Bisim\ \mathcal{R}$, when for all P_1, P_2, σ_1, σ_2, if $(\langle P_1,\sigma_1\rangle,\langle P_2,\sigma_2\rangle) \in \mathcal{R}$ then for all actions a:

(a) $\langle P_1,\sigma_1\rangle \overset{a}{\Rightarrow} \langle P_1',\sigma_1'\rangle$ implies $\exists P_2'\ \sigma_2'.\langle P_2,\sigma_2\rangle \overset{a}{\Rightarrow} \langle P_2',\sigma_2'\rangle \wedge (\langle P_1',\sigma_1'\rangle,\langle P_2',\sigma_2'\rangle) \in \mathcal{R}$.
(b) $\langle P_2,\sigma_2\rangle \overset{a}{\Rightarrow} \langle P_2',\sigma_2'\rangle$ implies $\exists P_1'\ \sigma_1'.\langle P_1,\sigma_1\rangle \overset{a}{\Rightarrow} \langle P_1',\sigma_1'\rangle \wedge (\langle P_1',\sigma_1'\rangle,\langle P_2',\sigma_2'\rangle) \in \mathcal{R}$.

Bisimilar configurations make transitions that preserve bisimilarity. Given two configurations, if $\langle P_1,\sigma_1\rangle \sim \langle P_2,\sigma_2\rangle$, then for any action a,

(a) $\langle P_1,\sigma_1\rangle \overset{a}{\Rightarrow} \langle P_1',\sigma_1'\rangle$ implies $\exists P_2'\sigma_2'.\langle P_2,\sigma_2\rangle \overset{a}{\Rightarrow} \langle P_2',\sigma_2'\rangle \wedge \langle P_1',\sigma_1'\rangle \sim \langle P_2',\sigma_2'\rangle$
(b) $\langle P_2,\sigma_2\rangle \overset{a}{\Rightarrow} \langle P_2',\sigma_2'\rangle$ implies $\exists P_1'\ \sigma_1'.\ \langle P_1,\sigma_1\rangle \overset{a}{\Rightarrow} \langle P_1',\sigma_1'\rangle \wedge \langle P_1',\sigma_1'\rangle \sim \langle P_2',\sigma_2'\rangle$.

Bisimilarity is an equivalence relation.

(Commutivity and Associativity) For all behavioral IspCal terms P, Q, R, and store σ,

1. $\langle P+Q,\sigma\rangle \sim \langle Q+P,\sigma\rangle$
2. $\langle P+(Q+R),\sigma\rangle \sim \langle (P+Q)+R,\sigma\rangle$

(Substitutivity) For all behavioral IspCal terms P, Q, R and store σ,

1. $\langle P,\sigma\rangle \sim \langle Q,\sigma\rangle \supset \langle P+R,\sigma\rangle \sim \langle Q+R,\sigma\rangle$
2. $\langle P,\sigma\rangle \sim \langle Q,\sigma\rangle \supset \langle b \rightarrow P,\sigma\rangle \sim \langle b \rightarrow Q,\sigma\rangle$
3. $X \Leftarrow P_X(X_1,\ldots,X_n) \in Pl\ \wedge\ (\forall X \Leftarrow P_i, X \Leftarrow P_j \in Pl.\ P_i = P_j) \supset$
$\langle Proc\ X\ Pl,\sigma\rangle \sim \langle P_X\,[Proc\,X_1\ Pl/X_1,\ldots,Proc\,X_n\ Pl/X_n],\sigma\rangle$

4 Structural IspCal

4.1 Syntax

Structural IspCal (SIsp) terms are constructed from behavioral IspCal terms. A basic SIsp term $(Cell\ P\ \sigma\ s)$ consists of a behavioral IspCal term P, its (local) store σ, and a port set s declaring all the ports process P may use. Structural IspCal terms, C_1 and C_2, are composed synchronously by $C_1 \parallel C_2$. SIsp term $C \setminus s$ hides ports s of the component C. The abstract syntax for SIsp terms is:

$$C ::= Cell\ P\ \sigma\ s \mid C_1 \| C_2 \mid C \setminus s$$
$$s ::= \{p_0; p_1; p_2; \ldots; p_n\}$$

where $P \in BIsp$, σ is a local store for P, $p \in Port$, and $s \subseteq Port$.

4.2 Sort

The *sort* of a structural IspCal term C is the set of all possible real events $p.v$ that the term C may perform. We define a recursive function $SORT$ that computes the sort of structural IspCal terms as follows:

$$SORT(Cell\ P\ \sigma\ s) = \{p.v \mid p \in s \wedge v \in V\}$$
$$SORT(C_1 \| C_2) = SORT(C_1) \cup SORT(C_2)$$
$$SORT(C \setminus s) = SORT(C) - \{p.v \mid p \in s \wedge v \in V\}$$

4.3 Operational Semantics of Structural IspCal Terms

The semantics of structural IspCal terms is defined by a labeled transition system $\stackrel{act}{\Longrightarrow} \subseteq SIsp \times SIsp$. As in the case of behavioral IspCal, $C_1 \stackrel{act}{\Longrightarrow} C_2$ means that the component C_1 may perform the action act and become C_2. The rules for the labeled transition system for structural IspCal are as follows:

1. $$\frac{\langle P, \sigma\rangle \stackrel{act}{\Longrightarrow} \langle P', \sigma'\rangle}{Cell\ P\ \sigma\ s \stackrel{act}{\Longrightarrow} Cell\ P'\ \sigma'\ s}\ act \subseteq \{p.v \mid p \in s \wedge v \in V\}$$
   Basic cells behave according to their process definitions.
2. $$\frac{C \stackrel{act}{\Longrightarrow} C'}{(C \setminus s) \stackrel{act - \{p.v \mid p \in s \wedge v \in V\}}{\Longrightarrow} (C' \setminus s)}$$
   Cells with hidden ports make the same transitions as the same cells without hidden ports except the events corresponding to the hidden ports are missing.
3. $$\frac{C_1 \stackrel{a_1}{\Longrightarrow} C_1' \quad C_2 \stackrel{a_2}{\Longrightarrow} C_2'}{(C_1 \parallel C_2) \stackrel{a_1 \cup a_2}{\Longrightarrow} (C_1' \parallel C_2')}\ a_1 \cap SORT(C_2) = a_2 \cap SORT(C_1)$$
   Synchronous composition has the union of its components' actions as its transition label. The components make transitions in lock-step synchronism.

The side condition for the composition rule is the *interacting condition*. It requires events on shared ports of C_1 and C_2 to be the same. Intuitively, when two components are composed with $\|$, similarly named ports of C_1 and C_2 are connected together and both components share the same clock.

4.4 Bisimulation Equivalence

Bisimulation relations for SIsp terms are defined as they are for BIsp configurations. $\mathcal{R}$ is a bisimulation when for all SIsp terms C_1, C_2, if $(C_1, C_2) \in \mathcal{R}$ then for all actions a:
(a) $C_1 \overset{a}{\Rightarrow} C_1'$ implies $\exists C_2'.\, C_2 \overset{a}{\Rightarrow} C_2' \wedge (C_1', C_2') \in \mathcal{R}$.
(b) $C_2 \overset{a}{\Rightarrow} C_2'$ implies $\exists C_1'.\, C_1 \overset{a}{\Rightarrow} C_1' \wedge (C_1', C_2') \in \mathcal{R}$.

As before, bisimilarity for SIsp processes is defined as

$$C_1 \sim C_2 =_{def} \exists \mathcal{R}.\, Bisim\ \mathcal{R} \wedge (C_1, C_2) \in \mathcal{R}$$

The properties of bisimilarity for SIsp processes are the same as the properties for bisimilarity for BIsp configurations. If $C_1 \sim C_2$ then for any action a,
(a) $C_1 \overset{a}{\Rightarrow} C_1'$ implies $\exists C_2'.\ C_2 \overset{a}{\Rightarrow} C_2' \wedge\ C_1' \sim C2'$
(b) $C_2 \overset{a}{\Rightarrow} C_2'$ implies $\exists C_1'.\ C_1 \overset{a}{\Rightarrow} C_1' \wedge\ C_1' \sim C_2'$.

The following properties are essential for hardware composition and hold for SIsp bisimulation relations.

(**Commutivity and Associativity**) For all structural IspCal agents C, C_1, C_2, and C_3,
1. $(C_1 \parallel C_2) \sim (C_2 \parallel C_1)$
2. $((C_1 \parallel C_2) \parallel C_3) \sim (C_1 \parallel (C_2 \parallel C_3))$

(**Substitutivity**) For all structural IspCal agents C_1, C_2, C_3 and stores σ_1, σ_2,
1. $\langle P1, \sigma_1 \rangle \sim \langle P2, \sigma_2 \rangle \supset (Cell\ P_1\ \sigma_1\ s) \sim (Cell\ P_2\ \sigma_2\ s)$
2. $C_1 \sim C_2 \wedge (SORT(C_1) = SORT(C_2)) \supset (C_1 \parallel C_3) \sim (C_2 \parallel C_3)$
3. $C_1 \sim C_2 \supset (C_1 \setminus s) \sim (C_2 \setminus s)$

5 Microprogrammed Controller

We describe a microprogrammed controller using both an ASM model and an ISP model. The two models are proved equivalent, provided they start from similar states. We implement the controller using a sequencer and a ROM. The implementation is proved equivalent to both the ASM and ISP models of the controller. This controller is then used in Section 7 to implement a PDP-8 instruction set machine [2].

5.1 User's Model

The controller has an input port *op* and an output port *control*. The ASM description of the microprogrammed controller uses symbolic control states in place of registers in the control part. It has 27 such control states in the following ASM description of the controller. Figure 2 shows the state diagram of the controller. The IspCal description is below. Note, *op* is used both as a port name and as a variable name. When *op* appears in data expressions *op* is a variable.

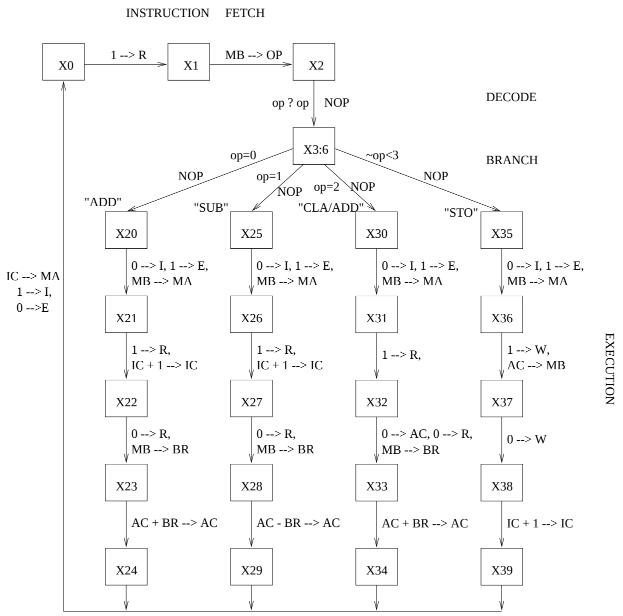

Fig. 2. Flow chart for the microprogrammed controller

Input and output events are in every prefix in *micro_asm* (even if the input values are not used) as ports on physical devices always have some value.

$$
\begin{array}{l}
micro\_asm =_{def} \{ \\
\quad X_0 \Leftarrow\!= \{op?op; control!SET\ R\}.X_1; \\
\quad X_1 \Leftarrow\!= \{op?op; control!MB\ INTO\ OP\}.X_2; \\
\quad X_2 \Leftarrow\!= \{op?op; op := op; control!NOP\}. \\
\qquad\qquad ((op = 0 \rightarrow X_3) + (op = 1 \rightarrow X_4) + (op = 2 \rightarrow X_5) + (\neg op < 3 \rightarrow X_6)); \\
\quad X_3 \Leftarrow\!= \{op?op; control!NOP\}.X_{20}; \\
\quad X_4 \Leftarrow\!= \{op?op; control!NOP\}.X_{25}; \\
\quad X_5 \Leftarrow\!= \{op?op; control!NOP\}.X_{30}; \\
\quad X_6 \Leftarrow\!= \{op?op; control!NOP\}.X_{35}; \\
\quad X_{20} \Leftarrow\!= \{op?op; control!RESET\ I, SET\ E, MB\ TO\ MA\}.X_{21}; \\
\quad X_{21} \Leftarrow\!= \{op?op; control!SET\ R, INCR\ IC\}.X_{22}; \\
\quad X_{22} \Leftarrow\!= \{op?op; control!RESET\ R, MB\ INTO\ BR\}.X_{23}; \\
\quad X_{23} \Leftarrow\!= \{op?op; control!ADD\}.X_{24}; \\
\quad X_{24} \Leftarrow\!= \{op?op; control!IC\ INTO\ MA, SET\ I, RESET\ E\}.X_0; \\
\quad X_{25} \Leftarrow\!= \{op?op; control!MB\ TO\ MA, RESET\ I, SET\ E\}.X_{26}; \\
\quad X_{26} \Leftarrow\!= \{op?op; control!SET\ R, INCR\ IC\}.X_{27}; \\
\quad X_{27} \Leftarrow\!= \{op?op; control!MB\ INTO\ BR, RESET\ R\}.X_{28}; \\
\quad X_{28} \Leftarrow\!= \{op?op; control!SUBTRACT\}.X_{29}; \\
\quad X_{29} \Leftarrow\!= \{op?op; control!IC\ INTO\ MA, SET\ I, RESET\ E\}.X_0; \\
\quad X_{30} \Leftarrow\!= \{op?op; control!MB\ TO\ MA, RESET\ I, SET\ E\}.X_{31}; \\
\quad X_{31} \Leftarrow\!= \{op?op; control!SET\ R\}.X_{32}; \\
\quad X_{32} \Leftarrow\!= \{op?op; control!MB\ INTO\ BR, RESET\ AC, RESET\ R\}.X_{33}; \\
\quad X_{33} \Leftarrow\!= \{op?op; control!ADD\}.X_{34}; \\
\quad X_{34} \Leftarrow\!= \{op?op; control!IC\ INTO\ MA, SET\ I, RESET\ E\}.X_0; \\
\quad X_{35} \Leftarrow\!= \{op?op; control!RESET\ I, SET\ E, MB\ TO\ MA\}.X_{36}; \\
\quad X_{36} \Leftarrow\!= \{op?op; control!SET\ W, AC\ INTO\ MB\}.X_{37}; \\
\quad X_{37} \Leftarrow\!= \{op?op; control!RESET\ W\}.X_{38}; \\
\quad X_{38} \Leftarrow\!= \{op?op; control!INCR\ IC\}.X_{39}; \\
\quad X_{39} \Leftarrow\!= \{op?op; control!IC\ INTO\ MA, SET\ I, RESET\ E\}.X_0; \\
\}
\end{array}
$$

The successor of $\langle Proc\, X_2\ micro\_asm, \sigma\rangle$, denoted as $X_{3:6}$ in Figure 2, is $\langle (op = 0 \rightarrow Proc\, X_3\, micro\_asm) + (op = 1 \rightarrow Proc\, X_4\, micro\_asm) + (op = 2 \rightarrow Proc\, X_5\, micro\_asm) + (\neg(op < 3) \rightarrow Proc\, X_6\, micro\_asm), \sigma[op \mapsto v]\rangle$ for some value v, which behaves similar to $Proc\ X_3\ micro\_asm$, $Proc\ X_4\ micro\_asm$, $Proc\ X_5\ micro\_asm$, or $Proc\ X_6\ micro\_asm$, depending on the value v received in state X_2.

The instruction-set processor description of the controller is defined by $micro\_isp$ shown below. Note, the data expression that updates the value of iar is a nested *if-then-else* expression of the form $A \rightarrow B \mid C$.

$$
\begin{array}{l}
micro\_isp =_{def} Proc\,X\ \{ \\
\quad X \Longleftarrow \{op?op; control!control[iar]; \\
\qquad\qquad iar := ((uinst[iar] = RESET\ IAR) \rightarrow 0 \mid \\
\qquad\qquad\qquad ((uinst[iar] = C\ INTO\ IAR) \rightarrow uaddr[iar] \mid \\
\qquad\qquad\qquad\quad ((uinst[iar] = INCR\ IAR) \rightarrow iar + 1 \mid \\
\qquad\qquad\qquad\qquad (op < 3 \rightarrow op + iar + 1 \mid 3 + iar + 1))))\}.X \\
\}
\end{array}
$$

$micro\_isp$ has one control state X. It has an input port op and an output port $control$. Similar to $micro\_asm$, op is used both as a port name and as a variable name. During each cycle, the input port op is read, the output port $control$ is assigned the value $\llbracket control[iar]\rrbracket\sigma$ and iar is updated depending on the values of op and $\llbracket uinst[iar]\rrbracket\sigma$. iar corresponds to the *micro-instruction address register*. It is used as an index to point to values in arrays *control*, *uinst*, and *uaddr*. Arrays *control*, *uinst*, and *uaddr* correspond to *control, microinstruction*, and *next micro-address fields* in a microprogram stored in σ.

To relate the ISP description to the ASM description, a microprogram should be in the store for the ISP description. The following predicate $is\_micropgm$ specifies the stored microprogram by explicitly stating the values of the control, micro-instruction, and next micro-address fields. Notice the correspondence between the values of *control* in the microprogram with the values assigned to the *control* port in *micro_asm*. The predicate *is_micropgm* is an abstract description of the microprogram in σ.

$$
\begin{array}{l}
\forall\sigma.\ is\_micropgm\ \sigma =_{def} \\
\quad (\llbracket uinst[0]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[0]\rrbracket\sigma = SET\ R) \wedge \\
\quad (\llbracket uinst[1]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[1]\rrbracket\sigma = MB\ INTO\ OP) \wedge \\
\quad (\llbracket uinst[2]\rrbracket\sigma = ADD\ OP\ TO\ IAR) \wedge (\llbracket control[2]\rrbracket\sigma = NOP) \wedge \\
\quad (\llbracket uinst[3]\rrbracket\sigma = C\ INTO\ IAR) \wedge (uaddr[3]\rrbracket\sigma = 20) \wedge (\llbracket control[3]\rrbracket\sigma = NOP) \wedge \\
\quad (\llbracket uinst[4]\rrbracket\sigma = C\ INTO\ IAR) \wedge (uaddr[4]\rrbracket\sigma = 25) \wedge (\llbracket control[4]\rrbracket\sigma = NOP) \wedge \\
\quad (\llbracket uinst[5]\rrbracket\sigma = C\ INTO\ IAR) \wedge (uaddr[5]\rrbracket\sigma = 30) \wedge (\llbracket control[5]\rrbracket\sigma = NOP) \wedge \\
\quad (\llbracket uinst[6]\rrbracket\sigma = C\ INTO\ IAR) \wedge (uaddr[6]\rrbracket\sigma = 35) \wedge (\llbracket control[6]\rrbracket\sigma = NOP) \wedge \\
\quad (\llbracket uinst[20]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[20]\rrbracket\sigma = RESET\ I, SET\ E, MB\ TO\ MA) \wedge \\
\quad (\llbracket uinst[21]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[21]\rrbracket\sigma = SET\ R, INCR\ IC) \wedge \\
\quad (\llbracket uinst[22]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[22]\rrbracket\sigma = RESET\ R, MB\ INTO\ BR) \wedge \\
\quad (\llbracket uinst[23]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[23]\rrbracket\sigma = ADD) \wedge \\
\quad (\llbracket uinst[24]\rrbracket\sigma = RESET\ IAR) \wedge (\llbracket control[24]\rrbracket\sigma = IC\ INTO\ MA, SET\ I, RESET\ E) \wedge \\
\quad (\llbracket uinst[25]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[25]\rrbracket\sigma = MB\ INTO\ MA, RESET\ I, SET\ E) \wedge \\
\quad (\llbracket uinst[26]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[26]\rrbracket\sigma = SET\ R, INCR\ IC) \wedge \\
\quad (\llbracket uinst[27]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[27]\rrbracket\sigma = MB\ INTO\ BR, RESET\ R) \wedge \\
\quad (\llbracket uinst[28]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[28]\rrbracket\sigma = SUBTRACT) \wedge \\
\quad (\llbracket uinst[29]\rrbracket\sigma = RESET\ IAR) \wedge (\llbracket control[29]\rrbracket\sigma = IC\ INTO\ MA, SET\ I, RESET\ E) \wedge \\
\quad (\llbracket uinst[30]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[30]\rrbracket\sigma = MB\ INTO\ MA, RESET\ I, SET\ E) \wedge \\
\quad (\llbracket uinst[31]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[31]\rrbracket\sigma = SET\ R) \wedge \\
\quad (\llbracket uinst[32]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[32]\rrbracket\sigma = MB\ INTO\ BR, RESET\ AC, RESET\ R) \wedge \\
\quad (\llbracket uinst[33]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[33]\rrbracket\sigma = ADD) \wedge \\
\quad (\llbracket uinst[34]\rrbracket\sigma = RESET\ IAR) \wedge (\llbracket control[34]\rrbracket\sigma = IC\ INTO\ MA, SET\ I, RESET\ E) \wedge \\
\quad (\llbracket uinst[35]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[35]\rrbracket\sigma = RESET\ I, SET\ E, MB\ INTO\ MA) \wedge \\
\quad (\llbracket uinst[36]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[36]\rrbracket\sigma = SET\ W, AC\ INTO\ MB) \wedge \\
\quad (\llbracket uinst[37]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[37]\rrbracket\sigma = RESET\ W) \wedge \\
\quad (\llbracket uinst[38]\rrbracket\sigma = INCR\ IAR) \wedge (\llbracket control[38]\rrbracket\sigma = INCR\ IC) \wedge \\
\quad (\llbracket uinst[39]\rrbracket\sigma = RESET\ IAR) \wedge (\llbracket control[39]\rrbracket\sigma = IC\ INTO\ MA, SET\ I, RESET\ E)
\end{array}
$$

With the microprogram in the store of the ISP description, we can prove that $micro\_isp$ is equivalent to $Proc\,X_0\,micro\_asm$ if the ISP model starts at instruction address 0.

$$\vdash \forall\sigma_1\sigma_2.(\llbracket iar\rrbracket\sigma_1 = 0) \wedge is\_micropgm\sigma_1 \supset \langle micro\_isp, \sigma_1\rangle \sim \langle ProcX_0micro\_asm, \sigma_2\rangle$$

We proved the following relation $\mathcal{R}$ is a bisimulation, i.e. $\vdash$ *Bisim* $\mathcal{R}$, for any σ_1' and σ_2'. $\mathcal{R}$ must be given by the designer.

$$
\begin{array}{l}
\mathcal{R} = \{(\langle micro\_isp, \sigma_1'\rangle, \langle Proc\, X_0\ micro\_asm, \sigma_2'\rangle) \mid ([\![IAR]\!]\sigma_1' = 0) \wedge is\_micropgm(\sigma_1')\} \ \cup \\
\quad \{(\langle micro\_isp, \sigma_1'\rangle, \langle Proc\, X_1\ micro\_asm, \sigma_2'\rangle) \mid ([\![IAR]\!]\sigma_1' = 1) \wedge is\_micropgm(\sigma_1')\} \ \cup \\
\quad \{(\langle micro\_isp, \sigma_1'\rangle, \langle Proc\, X_2\ micro\_asm, \sigma_2'\rangle) \mid ([\![IAR]\!]\sigma_1' = 2) \wedge is\_micropgm(\sigma_1')\} \ \cup \\
\quad \{(\langle micro\_isp, \sigma_1'\rangle, \langle (op = 0) \rightarrow Proc\, X_3\, micro\_asm + (op = 1) \rightarrow Proc\, X_4\, micro\_asm + \\
\qquad (op = 2) \rightarrow Proc\, X_5\ micro\_asm + \neg(op < 3) \rightarrow Proc\, X_6\ micro\_asm, \sigma_2'\rangle) \mid \\
\qquad ([\![IAR]\!]\sigma_1' = 3) \wedge ([\![op]\!]\sigma_1' = 0) \wedge is\_micropgm(\sigma_1')\} \ \cup \\
\quad \{(\langle micro\_isp, \sigma_1'\rangle, \langle (op = 0) \rightarrow Proc\, X_3\, micro\_asm + (op = 1) \rightarrow Proc\, X_4\, micro\_asm + \\
\qquad (op = 2) \rightarrow Proc\, X_5\ micro\_asm + \neg(op < 3) \rightarrow Proc\, X_6\ micro\_asm, \sigma_2'\rangle) \mid \\
\qquad ([\![IAR]\!]\sigma_1' = 4) \wedge ([\![op]\!]\sigma_1' = 1) \wedge is\_micropgm(\sigma_1')\} \ \cup \\
\quad \{(\langle micro\_isp, \sigma_1'\rangle, \langle (op = 0) \rightarrow Proc\, X_3\, micro\_asm + (op = 1) \rightarrow Proc\, X_4\, micro\_asm + \\
\qquad (op = 2) \rightarrow Proc\, X_5\ micro\_asm + \neg(op < 3) \rightarrow Proc\, X_6\ micro\_asm, \sigma_2'\rangle) \mid \\
\qquad ([\![IAR]\!]\sigma_1' = 5) \wedge ([\![op]\!]\sigma_1' = 2) \wedge is\_micropgm(\sigma_1')\} \ \cup \\
\quad \{(\langle micro\_isp, \sigma_1'\rangle, \langle (op = 0) \rightarrow Proc\, X_3\, micro\_asm + (op = 1) \rightarrow Proc\, X_4\, micro\_asm + \\
\qquad (op = 2) \rightarrow Proc\, X_5\ micro\_asm + \neg(op < 3) \rightarrow Proc\, X_6\ micro\_asm, \sigma_2'\rangle) \mid \\
\qquad ([\![IAR]\!]\sigma_1' = 6) \wedge \neg([\![op]\!]\sigma_1') < 3) \wedge is\_micropgm(\sigma_1')\} \ \cup \\
\quad \{(\langle micro\_isp, \sigma_1'\rangle, \langle Proc\, X_i\ micro\_asm, \sigma_2'\rangle) \mid is\_micropgm(\sigma_1') \wedge \\
\qquad\qquad\qquad\qquad ([\![IAR]\!]\sigma_1' = i) \ \ for\ i\ = 20, \ldots, 39\}
\end{array}
$$

5.2 Controller Implementation

The controller is implemented using a sequencer and a read-only memory (ROM). The sequencer and ROM are composed to implement the controller as shown in Figure 3. The difference between this description and *micro_isp* is that *micro_isp* did not describe any implementation details of how the ROM was controlled by the sequencer. In particular, the details of how values for the *addr, uinst,* and *uaddr* ports *internal* to the implementation of *micro_imp* being computed were not stated.

The behavior of the sequencer and ROM are described as ISP models. The sequencer has an instruction address register *iar*. It has input ports *op, uinst,* and *uaddr*, and an output port *addr*. The sequencer in IspCal is defined as:

$$
\begin{array}{l}
microseq =_{def} Proc\, X\ \{ \\
\quad X \Longleftarrow \{op?op; uinst?uinst; uaddr?uaddr; addr!iar; \\
\qquad\quad iar := ((uinst = RESET\, IAR) \rightarrow 0 \mid \\
\qquad\qquad\quad ((uinst = C\, INTO\, IAR) \rightarrow uaddr \mid \\
\qquad\qquad\quad ((uinst = INCR\, IAR) \rightarrow iar + 1 \mid \\
\qquad\qquad\quad (op < 3 \rightarrow op + iar + 1 \mid 3 + iar + 1))))\}.X \\
\}
\end{array}
$$

uinst is the micro-instruction from the ROM. *uaddr* is the next micro-address from the ROM. The ROM is addressed by *addr*.

The microprogram ROM, *microrom*, has three output ports — *uinst* for current microinstruction code, *uaddr* for the next address, and *control* for the micro-control output.

$$
\begin{array}{l}
microrom =_{def} Proc\, X\ \{ \\
\quad X \Leftarrow \{addr?addr; uinst!uinst[addr]; uaddr!uaddr[addr]; control!control[addr]\}.X \\
\}
\end{array}
$$

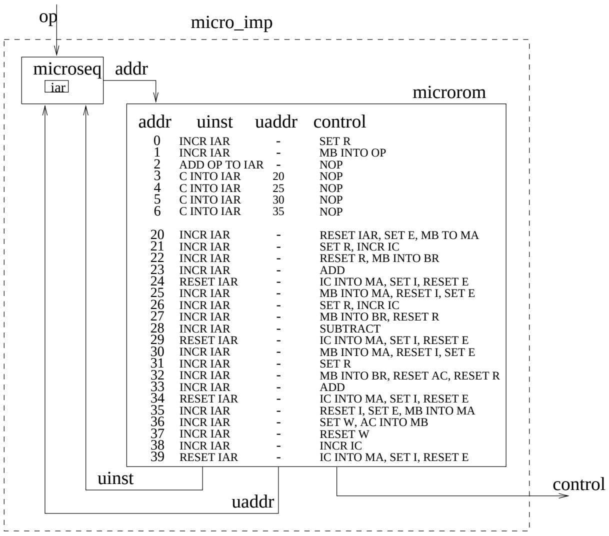

Fig. 3. Structural implementation of the controller

The term *microrom* only defines how memory is read. The actual memory contents are in store σ. *micro_imp* is defined to be the composition of the sequencer and the ROM with the internal ports *uinst, uaddr,* and *addr* hidden.

$$
\begin{aligned}
&micro\_imp\ \sigma_1\ \sigma_2 =_{def}\\
&\qquad (Cell\ microseq\ \sigma_1\ \{op; uinst; uaddr; addr\} \parallel\\
&\qquad\ Cell\ microrom\ \sigma_2\ \{addr; uinst; uaddr; control\}) \setminus \{uinst; uaddr; addr\}
\end{aligned}
$$

The following theorem asserts the equivalence between the structural implementation *micro_imp* and the ISP description of the controller *micro_isp*, if they start from similar memory states.

$$
\begin{aligned}
&\vdash \forall \sigma\ \sigma_1\ \sigma_2.\\
&\qquad ([\![iar]\!]\sigma = [\![iar]\!]\sigma_1)\ \wedge (\forall i.[\![uinst[i]]\!]\sigma = [\![uinst[i]]\!]\sigma_2)\ \wedge\\
&\qquad (\forall i.[\![uaddr[i]]\!]\sigma = [\![uaddr[i]]\!]\sigma_2)\ \wedge (\forall i.[\![control[i]]\!]\sigma = [\![control[i]]\!]\sigma_2)\ \supset\\
&\qquad (micro\_imp\ \sigma_1\ \sigma_2) \sim (Cell\ micro\_isp\ \sigma\ \{op; control\})
\end{aligned}
$$

The bisimulation relation $\mathcal{R}$ between *micro_imp* and *micro_isp* is given below. Note that any pair $(micro\_imp\ \sigma_1\sigma_2, Cell\ micro\_isp\ \sigma\{op; control\})$ is contained in the relation $\mathcal{R}$ for any σ, σ_1, and σ_2 as long as the stores σ, σ_1, and σ_2 satisfy the constraints in the set specification of $\mathcal{R}$ shown below.

$$
\begin{aligned}
\mathcal{R} = \{&(micro\_imp\ \sigma_1\sigma_2, Cell\ micro\_isp\ \sigma\{op; control\}) \mid\\
&([\![iar]\!]\sigma = [\![iar]\!]\sigma_1) \wedge (\forall\, i.\ [\![uinst[i]]\!]\sigma = [\![uinst[i]]\!]\sigma_2) \wedge\\
&(\forall\, i.\ [\![uaddr[i]]\!]\sigma = [\![uaddr[i]]\!]\sigma_2) \wedge (\forall\, i.\ [\![control[i]]\!]\sigma = [\![control[i]]\!]\sigma_2)\}
\end{aligned}
$$

By transitivity of the bisimulation relation, the implementation is also equivalent with the ASM description of the controller provided the microprogram is

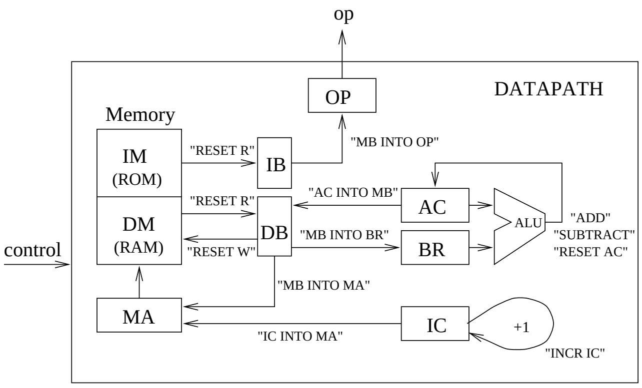

Fig. 4. Datapath Registers and their Connection

in the store for the implementation. The above theorem does not constrain σ in any way as the behavior of *micro_asm* is unaffected by σ.

$$\begin{array}{l} \vdash \forall \sigma\, \sigma_1\, \sigma_2. \\ \quad ([\![iar]\!]\sigma_1 = 0) \;\wedge\; is\_micropgm\, \sigma_2 \supset \\ \quad (micro\_imp\, \sigma_1\, \sigma_2) \sim (Cell\ (Proc\, X0\, micro\_asm)\, \sigma\, \{op; control\}) \end{array}$$

6 A Datapath Example

Datapath modules are simple to describe in IspCal using value-passing. The example in this section is modeled after Harris 6100 microprocessor in [2]. Our simplified model is shown in Figure 4.

Main memory is divided into instruction memory (IM) and data memory (DM). The read-only instruction memory holds the operation (OP code) portion of instructions. The data memory holds instruction addresses and data. Data are read from or written to the data memory. Registers IB and DB are buffer registers for IM and DM, respectively. The remaining registers are the instruction register (OP), memory address register (MA), instruction counter (IC), accumulator (AC) and another register (BR).

The instruction counter (IC) keeps track of program instructions. Normally IC is incremented with each instruction execution. The contents of the counter are transferred into MA at the beginning of each instruction cycle. The MA register provides memory address to both IM and DM.

When an instruction word is read from the memory, the OP-code portion of the word is stored in the OP register later to determine what instruction is to be performed. The memory address register points to the memory location to be read from or written to.

We modeled only 13 micro-control instructions for the datapath in this example. All the registers in the datapath store numerical values in this model. The OP register also stores numbers because the micro-controller performs arithmetic operations on the value received from the OP register. Note that the *op* port always outputs the value of the OP register. This reflects the fact that a physical port always carries some value on it unless we explicitly hide the port.

As all the registers are visible, the IspCal model of the datapath is an ISP machine,(i.e., a single-state ASM). The ISP model shown below has a single control state X. All registers are state variables. The data and instruction memories are indexed variables.

$$
\begin{array}{l}
datapath =_{def} Proc\, X\, \{ \\
\quad X \Longleftarrow \{control!SET\ R; op!OP; DB := DM[MA]; IB := IM[MA]\}.X + \\
\qquad \{control!MB\ INTO\ OP; op!OP; OP := IB\}.X + \\
\qquad \{control!NOP; op!OP\}.X + \\
\qquad \{control!RESET\ I, SET\ E, MB\ INTO\ MA; op!OP; MA := DB\}.X + \\
\qquad \{control!SET\ R, INCR\ IC; op!OP; IC := IC + 1; DB := DM[MA]; IB := IM[MA]\}.X + \\
\qquad \{control!RESET\ R, MB\ INTO\ BR; op!OP; BR := DB\}.X + \\
\qquad \{control!ADD; op!OP; AC := AC + BR\}.X + \\
\qquad \{control!IC\ INTO\ MA, SET\ I, RESET\ E; op!OP; MA := IC\}.X + \\
\qquad \{control!SUBTRACT; op!OP; AC := AC - BR\}.X + \\
\qquad \{control!MB\ INTO\ BR, RESET\ AC, RESET\ R; op!OP; AC := 0; BR := DB\}.X + \\
\qquad \{control!SET\ W, AC\ INTO\ MB; op!OP; DB := AC\}.X + \\
\qquad \{control!RESET\ W; op!OP; DM[MA] := DB\}.X + \\
\qquad \{control!INCR\ IC; op!OP; IC := IC + 1\}.X \\
\quad \}
\end{array}
$$

There are 13 micro-control instructions that can appear on the *control* port. All instructions except NOP involve register updates. The microinstruction command placed on the *control* port (from the controller) determines which instruction to perform.

Two types of lemmas together characterize the behavior of *datapath*. Lemmas of the first type describe the state transition for each instruction. An example of the first lemma type is shown below for the microinstruction $SET\ R$. (The other microinstructions have similar lemmas).

$$
\begin{array}{l}
\vdash \forall \sigma.\ \langle datapath, \sigma \rangle \stackrel{\{control.SETR; op.[\![OP]\!]\sigma\}}{\Longrightarrow} \\
\qquad \langle datapath, \sigma[DB \mapsto [\![DM[MA]]\!]\sigma; IB \mapsto [\![IM[MA]]\!]\sigma] \rangle
\end{array}
$$

This lemma says the configuration $\langle datapath, \sigma \rangle$ can make a $\{control.SET\ R; op.[\![OP]\!]\sigma\}$ move to a new configuration $\langle datapath, \sigma[DB \mapsto [\![DM[MA]]\!]\sigma; IB \mapsto [\![IM[MA]]\!]\sigma] \rangle$.

The second type of lemma lists all possible states reachable by a single transition. In this case, if a transition was made, the transition corresponded to one of the 13 microinstructions.

$$
\begin{array}{l}
\vdash \forall \sigma a P \sigma'. \langle datapath, \sigma \rangle \stackrel{a}{\Longrightarrow} \langle P, \sigma' \rangle \supset \\
\ (P = datapath) \wedge \\
\ ((a = \{control.SETR; op.[\![OP]\!]\sigma\}) \wedge \\
\qquad (\sigma' = \sigma[DB \mapsto [\![DM[MA]]\!]\sigma; IB \mapsto [\![IM[MA]]\!]\sigma]) \vee \\
\quad (a = \{control.MBINTOOP; op.[\![OP]\!]\sigma\}) \wedge (\sigma' = \sigma[OP \mapsto [\![IB]\!]\sigma]) \vee \\
\quad (a = \{control.NOP; op.[\![OP]\!]\sigma\}) \wedge (\sigma' = \sigma) \vee \\
\quad (a = \{control.RESETI, SETE, MBINTOMA; op.[\![OP]\!]\sigma\}) \wedge \\
\qquad (\sigma' = \sigma[MA \mapsto [\![DB]\!]\sigma]) \vee \\
\quad (a = \{control.SETR, INCRIC; op.[\![OP]\!]\sigma\}) \wedge \\
\qquad (\sigma' = \sigma[IC \mapsto [\![IC]\!]\sigma + 1; DB \mapsto [\![DM[MA]]\!]\sigma; IB \mapsto [\![IM[MA]]\!]\sigma]) \vee \\
\quad (a = \{control.RESETR, MBINTOBR; op.[\![OP]\!]\sigma\}) \wedge (\sigma' = \sigma[BR \mapsto [\![DB]\!]\sigma]) \vee \\
\quad (a = \{control.ADD; op.[\![OP]\!]\sigma\}) \wedge (\sigma' = \sigma[AC \mapsto [\![AC]\!]\sigma + [\![BR]\!]\sigma]) \vee \\
\quad (a = \{control.ICINTOMA, SETI, RESETE; op.[\![OP]\!]\sigma\}) \wedge \\
\qquad (\sigma' = \sigma[MA \mapsto [\![IC]\!]\sigma]) \vee \\
\quad (a = \{control.SUBTRACT; op.[\![OP]\!]\sigma\}) \wedge \\
\qquad (\sigma' = \sigma[AC \mapsto [\![AC]\!]\sigma - [\![BR]\!]\sigma]) \vee \\
\quad (a = \{control.MBINTOBR, RESETAC, RESETR; op.[\![OP]\!]\sigma\}) \wedge \\
\qquad (\sigma' = \sigma[AC \mapsto 0; BR \mapsto [\![DB]\!]\sigma]) \vee \\
\quad (a = \{control.SETW, ACINTOMB; op.[\![OP]\!]\sigma\}) \wedge (\sigma' = \sigma[DB \mapsto [\![AC]\!]\sigma]) \vee \\
\quad (a = \{control.RESETW; op.[\![OP]\!]\sigma\}) \wedge (\sigma' = \sigma[DM[[\![MA]\!]\sigma] \mapsto [\![DB]\!]\sigma]) \vee \\
\quad (a = \{control.INCRIC; op.[\![OP]\!]\sigma\}) \wedge (\sigma' = \sigma[IC \mapsto [\![IC]\!]\sigma + 1]))
\end{array}
$$

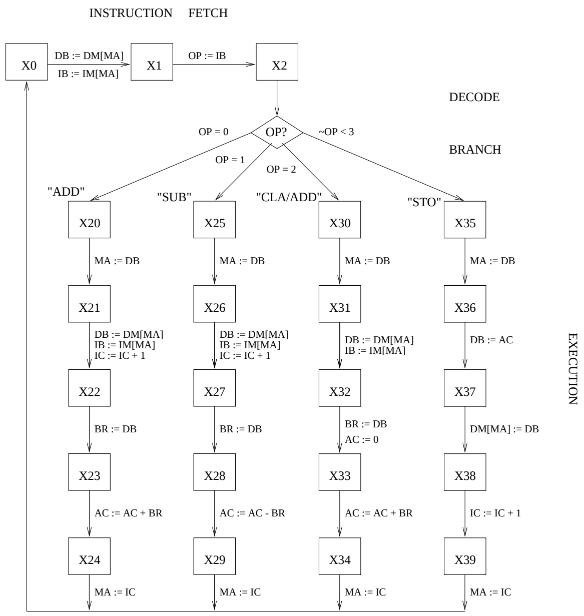

Fig. 5. Behavioral ASM description of the CPU

The two lemma types are used when reasoning about bisimulation.

7 A CPU Example

Composing the controller and datapath implements a PDP-8 instruction set machine [2]. The instruction set implemented here includes *ADD*, *SUBTRACT*, *CLEAR AND ADD*, and *STORE*. We show that the composition of the controller and datapath is bisimilar to the instruction set behavior as specified by an ASM. The behavioral ASM model is shown in Figure 5. Its representation as an IspCal term *top_asm* is shown below. The structural model is shown in Figure 6.

$$
\begin{array}{l}
top\_asm =_{def} \{ \\
\quad X_0 \Longleftarrow \{op!OP; DB := DM[MA]; IB := IM[MA]\}.X_1; \\
\quad X_1 \Longleftarrow \{op!OP; OP := IB\}.X_2; \\
\quad X_2 \Longleftarrow \{op!OP\}.((OP=0 \rightarrow X_3) + (OP=1 \rightarrow X_4) + (OP=2 \rightarrow X_5) + (\neg\, OP < 3 \rightarrow X_6)); \\
\quad X_3 \Longleftarrow \{op!OP\}.X_{20}; \\
\quad X_4 \Longleftarrow \{op!OP\}.X_{25}; \\
\quad X_5 \Longleftarrow \{op!OP\}.X_{30}; \\
\quad X_6 \Longleftarrow \{op!OP\}.X_{35}; \\
\quad X_{20} \Longleftarrow \{op!OP; MA := DM\}.X_{21}; \\
\quad X_{21} \Longleftarrow \{op!OP; IC := IC + 1; DB := DM[MA]; IB := IM[MA]\}.X_{22}; \\
\quad X_{22} \Longleftarrow \{op!OP; BR := DM\}.X_{23}; \\
\quad X_{23} \Longleftarrow \{op!OP; AC := AC + BR\}.X_{24}; \\
\quad X_{24} \Longleftarrow \{op!OP; MA := IC\}.X_0; \\
\quad X_{25} \Longleftarrow \{op!OP; MA := DM\}.X_{26}; \\
\quad X_{26} \Longleftarrow \{op!OP; IC := IC + 1; DB := DM[MA]; IB := IM[MA]\}.X_{27}; \\
\quad X_{27} \Longleftarrow \{op!OP; BR := DM\}.X_{28}; \\
\quad X_{28} \Longleftarrow \{op!OP; AC := AC - BR\}.X_{29}; \\
\quad X_{29} \Longleftarrow \{op!OP; MA := IC\}.X_0; \\
\quad X_{30} \Longleftarrow \{op!OP; MA := DM\}.X_{31}; \\
\quad X_{31} \Longleftarrow \{op!OP; DB := DM[MA]; IB := IM[MA]\}.X_{32}; \\
\quad X_{32} \Longleftarrow \{op!OP; AC := 0; BR := DM\}.X_{33}; \\
\quad X_{33} \Longleftarrow \{op!OP; AC := AC + BR\}.X_{34}; \\
\quad X_{34} \Longleftarrow \{op!OP; MA := IC\}.X_0;
\end{array}
$$

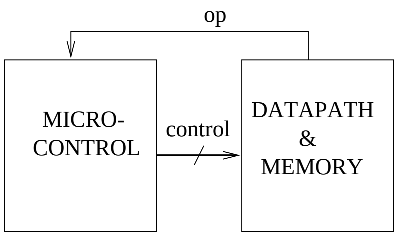

Fig. 6. An Implementation of the CPU

$X_{35} \Longleftarrow \{op!OP; MA := DM\}.X_{36};$
$X_{36} \Longleftarrow \{op!OP; DB := AC\}.X_{37};$
$X_{37} \Longleftarrow \{op!OP; DM[MA] := DM\}.X_{38};$
$X_{38} \Longleftarrow \{op!OP; IC := IC + 1\}.X_{39};$
$X_{39} \Longleftarrow \{op!OP; MA := IC\}.X_0$
}

top_asm has 27 control states. For each state, we prove two lemmas that characterize the state. The first lemma describes the state transition. The second lemma describes all possible states reachable in one transition. The characterization lemmas for the control state X_0 are as follows.

$$\begin{array}{l} \vdash \forall\, \sigma.\, Cell(Proc\, X_0\; top\_asm)\; \sigma\; \{op\} \overset{\{op.[\![OP]\!]\sigma\}}{\Longrightarrow} \\ \qquad Cell(Proc\, X_1\; top\_asm)\sigma[DB \mapsto [\![DM[MA]]\!]\sigma; IB \mapsto [\![IM[MA]]\!]\sigma]\;\{op\} \\ \vdash \forall\, \sigma\; a\; C.\, Cell(Proc\, X_0\; top\_asm)\; \sigma\; \{op\} \overset{a}{\Longrightarrow} C \supset \\ \qquad (a = \{op.[\![OP]\!]\sigma\}) \wedge \\ \qquad (C = Cell(ProcX_1 top\_asm)\sigma[DB \mapsto [\![DM[MA]]\!]\sigma; IB \mapsto [\![IM[MA]]\!]\sigma]\{op\}) \end{array}$$

Similar characterization lemmas are proved for the rest of the control states. The branching operation needs some explanation. The characterization lemmas for the state X_2 are shown below.

$$\begin{array}{l} \vdash \forall\, \sigma.\, Cell(Proc\, X_2\; top\_asm)\; \sigma\; \{op\} \overset{\{op.[\![OP]\!]\sigma\}}{\Longrightarrow} \\ \qquad Cell((OP = 0 \rightarrow Proc\, X_3\; top\_asm) + (OP = 1 \rightarrow Proc\, X_4\; top\_asm) + \\ \qquad\qquad (OP = 2 \rightarrow Proc\, X_5\; top\_asm) + (\neg OP < 3 \rightarrow Proc\, X_6\; top\_asm)) \\ \qquad \sigma[DB \mapsto [\![DM[MA]]\!]\sigma; IB \mapsto [\![IM[MA]]\!]\sigma]\;\{op\} \\ \vdash \forall\, \sigma\; a\; C.\, Cell(Proc\, X_2\; top\_asm)\; \sigma\; \{op\} \overset{a}{\Longrightarrow} C \supset \\ \qquad (a = \{op.[\![OP]\!]\sigma\}) \wedge \\ \qquad (C = Cell((OP{=}0 \rightarrow ProcX_3 top\_asm) + (OP{=}1 \rightarrow ProcX_4 top\_asm) + \\ \qquad\qquad (OP{=}2 \rightarrow ProcX_5 top\_asm) + (\neg OP{<}3 \rightarrow ProcX_6 top\_asm))\sigma\{op\}) \end{array}$$

The successor of the process $Cell(ProcX_2 top\_asm)\sigma\{op\}$ is a conditional process

$$\begin{array}{l} Cell((OP{=}0 \rightarrow Proc\, X_3\; top\_asm) + (OP{=}1 \rightarrow Proc\, X_4\; top\_asm) + \\ \qquad (OP{=}2 \rightarrow Proc\, X_5\; top\_asm) + (\neg OP{<}3 \rightarrow Proc\, X_6\; top\_asm))\; \sigma\; \{op\}, \end{array}$$

rather than process $ProcX_3 top\_asm$, $ProcX_4 top\_asm$, $ProcX_5 top\_asm$, or $Proc\; X_6\; top\_asm$. The conditional process, however, behaves like them depending on the stored value for the state variable *OP*. If *OP* is 0, for example, it behaves like $Proc\, X_3\; top\_asm$ as shown in the following lemmas.

$$
\begin{array}{l}
\vdash \forall\, \sigma.\ (\llbracket OP \rrbracket \sigma = 0)\ \supset \\
\quad Cell(((OP = 0) \rightarrow ProcX_3 top\_asm) + ((OP = 1) \rightarrow ProcX_4 top\_asm) + \\
\qquad ((OP = 2) \rightarrow ProcX_5 top\_asm) + \neg(OP < 3) \rightarrow ProcX_6 top\_asm)\sigma\{op\} \\
\quad \xRightarrow{\{op.\llbracket OP \rrbracket \sigma\}} Cell(Proc\, X_{20}\ top\_asm)\ \sigma\ \ \{op\} \\
\vdash \forall\, \sigma\, a\, C.\ (\llbracket OP \rrbracket \sigma = 0) \supset \\
\quad Cell(((OP = 0) \rightarrow ProcX_3 top\_asm) + ((OP = 1) \rightarrow ProcX_4 top\_asm) + \\
\qquad ((OP = 2) \rightarrow ProcX_5 top\_asm) + \neg(OP < 3) \rightarrow ProcX_6 top\_asm)\sigma\{op\} \\
\quad \xRightarrow{a} C\ \ \supset \\
\quad (a = \{op.\llbracket OP \rrbracket \sigma\}) \wedge (C = Cell(Proc\, X_{20}\ top\_asm)\ \sigma\ \{op\})
\end{array}
$$

The ASM description of the microprogrammed controller and the datapath of the previous section are composed together with their (local) stores σ_1 and σ_2. The *control* port is abstracted away as the *control* signal is invisible at the top-level.

$$
\begin{array}{l}
micro\_asm\_datapath\ X\ \sigma_1\ \sigma_2 =_{def} \\
\quad (Cell(ProcXmicro\_asm)\sigma_1\{op; control\} \| Celldatapath\sigma_2\{op; control\}) \backslash \{control\}
\end{array}
$$

We prove the bisimulation equivalence between the specification and implementation using all the characterization lemmas. The correctness theorem is shown below.

$$\vdash \forall\, \sigma\, \sigma_1.\ Cell\,(Proc\, X_0\ top\_asm)\ \sigma\ \{op\}\ \sim\ micro\_asm\_datapath\ X_0\ \sigma_1\ \sigma$$

The following relation $\mathcal{R}$ is proved to be a bisimulation. The above pair is contained in $\mathcal{R}$ where $\sigma_1, \sigma \in Store$.

$$
\begin{array}{l}
\mathcal{R} = \{(Cell\,(Proc\, X_0\ top\_asm)\ \sigma\{op\}, micro\_asm\_datapath\ X_0\sigma_1\sigma) \mid for\ any\ \sigma,\ \sigma_1\} \cup \\
\quad \{(Cell\,(Proc\, X_1\ top\_asm)\ \sigma\{op\}, micro\_asm\_datapath\ X_1\sigma_1\sigma) \mid for\ any\ \sigma,\ \sigma_1\} \cup \\
\quad \{(Cell\,(Proc\, X_2\ top\_asm)\ \sigma\ \{op\}, micro\_asm\_datapath\ X_2\ \sigma_1\ \sigma) \mid \llbracket OP \rrbracket \sigma \\
\qquad = \llbracket IB \rrbracket \sigma\ and\ any\ \sigma_1\} \cup \\
\quad \{(Cell\,((OP = 0 \rightarrow Proc\, X_3\ top\_asm) + (OP = 1 \rightarrow Proc\, X_4\ top\_asm) + \\
\qquad\quad (OP = 2 \rightarrow Proc\, X_5\ top\_asm) + (NegOP < 3 \rightarrow Proc\, X_6\ top\_asm)) \\
\qquad \sigma\ \{op\}, \\
\qquad Cell\,((op = 0 \rightarrow Proc\, X_3\ micro\_asm) + (op = 1 \rightarrow Proc\, X_4\ micro\_asm) + \\
\qquad\quad (op = 2 \rightarrow Proc\, X_5\ micro\_asm) + (Negop < 3 \rightarrow Proc\, X_6\ micro\_asm)) \\
\qquad \sigma_1\ \{op; control\}\ \| \\
\qquad datapath\ \sigma\{op; control\} \setminus \{control\}) \mid \llbracket op \rrbracket \sigma_1 = \llbracket OP \rrbracket \sigma\ and \\
\qquad \llbracket OP \rrbracket \sigma = \llbracket IB \rrbracket \sigma\} \cup \\
\quad \{(Cell\,(Proc\, X_i\ top\_asm)\ \sigma\{op\}, micro\_asm\_datapath\ X_i\ \sigma_1\ \sigma) \mid for\ i = \\
\qquad 20, \ldots, 39\ and\ any\ \sigma\ and\ \sigma_1\}
\end{array}
$$

8 Conclusions

Siewiorek, Bell, and Newell [5] identified several levels of design abstraction including the instruction-set processor (ISP) and register-transfer levels (RTL). Previous work at the ISP and RTL levels [1][7] is missing a means to reason *compositionally* about *structures* of ISP and RTL components in an algebraic fashion.

The contribution of IspCal is the capability to reason compositionally at the RTL and ISP levels. It can show equivalence between *structures* and *behavior* specified by ASM or ISP descriptions. As IspCal is a process algebra, it is straightforward to define a modal logic for it using the methods outlined by Stirling [14].

Acknowledgments Many thanks go to Susan Older who helped us with the high-level description of the semantics of IspCal. This work was partially supported by the New York State Center for Advanced Technology in Computer Applications and Software Engineering (CASE).

References

1. M. R. Barbacci. Instruction Set Processor Specifications ISPS: The Notation and its Applications. *IEEE Trans. Comp.*, C-30(7), January 1981.
2. Thomas C. Bartee. *Digital Computer Fundamentals — Sixth Edition*. McGraw-Hill, New York, 1984.
3. Gerard Berry. The Foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.
4. Christopher R. Clare. *Designing Logic Systems Using State Machines*. McGraw-Hill, 1973.
5. D. Siewiorek, C. Bell, and A. Newell. *Computer Structures — Principles and Examples*. McGraw-Hill, New York, 1982.
6. M.J.C. Gordon. A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subramanyam, editors, *VLSI specification, verification and synthesis*. Kluwer, 1987.
7. Gordon L. Smith, Ralph Bahnsen, and Harry Halliwell. Boolean Comparison of Hardware and Flowcharts. *IBM J. Res. Develop.*, 26(1):106 – 116, January 1982.
8. Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, 1982.
9. T. F. Melham. A Package for Inductive Relation Definitions in HOL. In *Proceedings of the 1991 International Tutorial and Workshop on the HOL Theorem Proving system*. IEEE Computer Society Press, Davis, California, August 1991.
10. George Milne. Circal and the representation of communication, concurrency and time. In *ACM Transactions on Programming Languages and Systems*, April 1985.
11. Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267 – 310, 1983.
12. Robin Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
13. Monica Nesi. A Formalization of the Process Algebra CCS in Higher Order Logic. Technical Report 278, University of Cambridge, December 1992.
14. Colin Stirling. Modal and Temporal Logics for Processes. In *Logics for Concurrency*, number 1043 in Lecture Notes in Computer Science, pages 149 – 237. Springer-Verlag, 1996.