Extending VLSI Design with Higher-Order Logic

Anand Chavan, Shiu-Kai Chin,
Shahid Ikram and Jang Dae Kim
Dept. of Electrical & Computer Engineering
Syracuse University
Syracuse, NY 13244-1240

Juin-Yeu Lu
Product Development Automation
National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara, CA 95052-8090

Abstract

Extending VLSI CAD with higher-order logic integrates formal verification with synthesis. The benefits of doing so are: 1) relating instruction-set descriptions to implementations, 2) designing at a higher level of abstraction than at the level of schematics, 3) verifying by proof, 4) reusing verified parameterized designs, 5) automatically compiling designs in higher-order logic to parameterized cell generators and layouts, and 6) validating electrical and functional properties by simulation. Such an integration is demonstrated by linking the Cambridge Higher-Order Logic (HOL) theorem-prover with the Mentor Graphics GDT design environment. We illustrate its application by creating a parameterized macro-cell generator for an n-bit *Am2910 microprogram sequencer, whose design is formally verified with respect to its instruction-set architecture specification.*

1 Introduction

This paper reports on our experience integrating formal verification with VLSI design. Our long term goal is to extend current simulation-based design techniques with theorem-based design techniques using higher-order logic. As the level of device integration increases, simulation and testing alone are less able to verify the correctness of designs. As complexity increases, correct, reusable, and parametric designs become more important for managing complexity.

The purpose of this effort was to see if we could integrate our theorem-based design tools with our VLSI design tools to support a practical design process. We wanted to simulate an industrial design experience using formal methods and CAD tools on a relatively simple design. We selected the Am2910 microprogram sequencer [1] because its top-level description combined both arithmetic functions with state transition behavior. Our specific objectives were to:

1. Create a working design within two months.
2. Create a reusable parameterized cell generator for the design.
3. Formally verify the correctness of the design.
4. Support simulation and hardware modeling early in the design.
5. Have the verification and design activities concurrent with each other.

We achieved all of our objectives. The Am2910 instruction-set description appears in Table 1. Four graduate students working half-time over two months completed the layout, cell generators, and formal verification of the Am2910. The tools used were the Mentor Graphics GDT CAD tool suite [12] and the Cambridge Higher-Order Logic (HOL) theorem-prover [7]. Cell generator programs in GDT are written in the L programming language, [12]. Our HOL2L compiler translates HOL design descriptions to cell generator programs in GDT [10]. While we used GDT and HOL specifically, other reasoning and VLSI CAD tools can be used as reported by Kumar, Schneider, and Kropf, [9].

Once the instruction-set descriptions are verified against gate-level descriptions, these gate level descriptions can be used with gate-to-gate verification tools like Chrysalis [5] to verify modified or optimized gate-level descriptions. This methodology is being integrated into the National Semiconductor design flows.

The layout appears in Figure 1. The time distribution for our design of the Am2910 was as follows.

- $t + 2$ weeks, all formal definitions were complete.
- $t + 3$ weeks, all formal definitions for components were compiled into GDT cell generator programs.
- $t + 4$ weeks, all components were simulated.
- $t + 8$ weeks, Am2910 was formally verified, corresponding GDT cell generator and layout were complete.

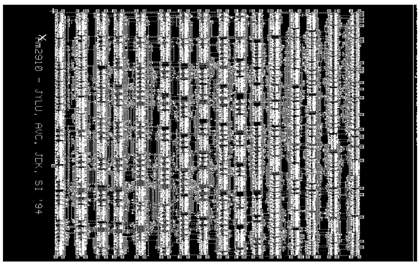

Figure 1: Am2910 layout

The division of labor was as follows. Jang Dae Kim was primarily responsible for the HOL verifications. He was assisted by Shahid Ikram. Juin-Yeu Lu was primarily responsible for GDT and the HOL2L compiler. He was assisted by Anand Chavan. Shiu-Kai Chin served as “program manager” and arbiter for major design decisions and conflicts. All personnel are engineers with backgrounds in VLSI design and formal verification. Chin, Ikram, Kim, and Lu are experienced HOL users. Chavan at the time understood the notation of higher-order logic but had no experience in theorem-proving. Chin, Lu, and Kim had prior industrial careers.

Most design errors were caught early in the design process by simulating the compiled formal design descriptions. Three errors with the register-counter were missed by simulation but caught by formal verification. These errors involved zero detection, an incorrect control signal connection to the instruction decoder, and an incorrect connection from the register-counter output to the output multiplexer.

The remainder of the paper describes the design process, the specification and verification techniques used, and the tools we developed to link theorems in HOL to parameterized cell generators in GDT.

2 Design Process

The actual design process was a “middle-out” one starting with a top-level functional decomposition as shown in Figure 2. From this intermediate-level of description, we started the concurrent activities of design and verification – the design activity moving “downwards” towards gate-level descriptions; the verification activity moving “upwards” towards verifying the instruction-set description against the intermediate block diagram description.

A diagram of the process flow is given in Figure 3. It supports design at the gate, register-transfer, and instruction-set levels as defined by Bell and Newell, [14]. The process starts on the left-hand side with the Am2910 instruction-set architecture, high-level block

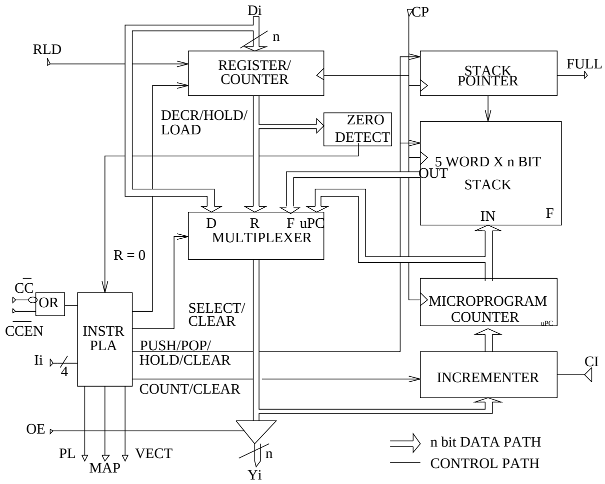

Figure 2: Am2910 Block Diagram

diagram, and the basic GDT standard-cell library. The sequence of activities is as follows where the activity numbers are associated with the numbered blocks and arcs in Figure 3.

1. From the instruction-set description given in Table 1, a corresponding formal definition is created. This process is largely automatic and is discussed in detail later.

2. From the high-level block diagram in Figure 2, the major components of the Am2910 are defined. For each of these components, a definition corresponding to its instruction-set architecture is made, e.g. the *stack* is defined in terms of its instructions *push, pop, hold*, and *clear*.

3. The high-level block diagram is formally verified against the instruction-set architecture specification. If any mistakes are found, the component specifications and interconnections are modified and passed to the designers. Verification proceeds concurrently with the design activities.

4. From the GDT standard-cell library, a set of gates is formally defined. The correctness of the HOL gate library is verified against the GDT cells by exhaustive simulation using Lsim in GDT.

5. The gate-level module implementations are formally defined. This is where most of the design occurs. For example, a basic multiplexer cell is defined and verified. Then an iterative array of multiplexer cells is defined corresponding to an *n-bit* multiplexer cell generator.

Table 1: Am2910 Instruction Set

Hex	Instruction	Name	Reg Ctr Value	Fail - see 1 Y	Fail - see 1 STACK	Pass - see 2 Y	Pass - see 2 STACK	Reg Ctr	Enable
0	JZ	Jump Zero	X	0	Clear	0	Clear	Hold	PL
1	CJS	Cond JSB PL	X	PC	Hold	D	Push	Hold	PL
2	JMAP	JUMP MAP	X	D	Hold	D	Hold	Hold	MAP
3	CJP	COND JUMP PL	X	PC	Hold	D	Hold	Hold	PL
4	PUSH	PUSH/Cond Ld Ctr	X	PC	Push	PC	Push	see 3	PL
5	JSRP	Cond JSB R/PL	X	R	Push	D	Push	Hold	PL
6	CJV	Cond Jump Vect	X	PC	Hold	D	Hold	Hold	VECT
7	JRP	Cond Jump R/PL	X	R	Hold	D	Hold	Hold	PL
8	RFCT	Repeat Loop, $Ctr \neq 0$	$\neq 0$	F	Hold	F	Hold	Dec	PL
			$= 0$	PC	Pop	PC	Pop	Hold	PL
9	RPCT	Repeat PL, $Ctr \neq 0$	$\neq 0$	D	Hold	D	Hold	Dec	PL
			$= 0$	PC	Hold	PC	Hold	Hold	PL
A	CRTN	Cond Rtn	X	PC	Hold	F	Pop	Hold	PL
B	CJPP	Cond Jump PL & Pop	X	PC	Hold	D	Pop	Hold	PL
C	LDCT	Ld Ctr & Continue	X	PC	Hold	PC	Hold	Load	PL
D	LOOP	Test End Loop	X	F	Hold	PC	Pop	Hold	PL
E	CONT	Continue	X	PC	Hold	PC	Hold	Hold	PL
F	TWB	Three-Way Branch	$\neq 0$	F	Hold	PC	Pop	Dec	PL
			$= 0$	D	Pop	PC	Pop	Hold	PL

1. Fail = $\overline{CCEN}$ = False and $\overline{CC}$ = True
2. Pass = $\overline{CCEN}$ = True or $\overline{CC}$ = False
3. if $RLD\_B = True$ then Hold; else Load

6. As each module implementation is defined, it is compiled into its corresponding cell generator in GDT. This is done by the HOL2L compiler described in [10].

7. The cell generator is used to create modules in GDT at the logic, transistor, and layout levels. Components are simulated. These simulations are a quick (but incomplete) check for correctness. They also allow for electrical characterization of each module. Errors found at this level are used to change the module definitions before formal verification is started. After each change, the module definitions are recompiled into their L cell generators and re-simulated. Performance information is fed back so implementations are changed if necessary.

8. If desired, hardware models can be created using the L2XNF compiler [4, 18] which maps L net lists to Xilinx FPGA net lists.

9. Once the module definitions are stable, they are verified in HOL against their instruction-set specifications.

10. The implementation-level module definitions as well as the entire Am2910 description are proved correct with respect to the instruction-set architecture using the prior correctness of the block diagram.

While formal verification is initially expensive for simple designs, all the proved properties are re-usable. These properties enable incremental verification which cannot be done by using simulation approaches. Having formally verified components at the instruction-set level and above simplifies system-level design.

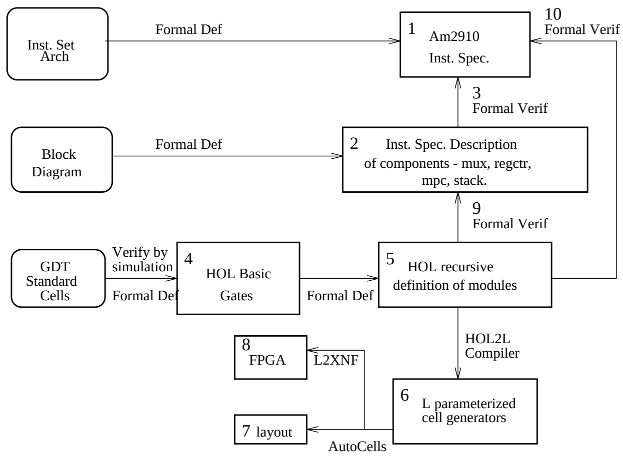

Figure 3: Process Flow

3 Instruction-Set Specification and Verification

In this section, we describe how we translate tabular instruction-set architecture (ISA) descriptions into *inductively defined relations* – i.e. relations defined by rules which generate elements satisfying the relation, [3]. From ISA descriptions we define instruction *syntax* and instruction *semantics*.

The syntax of an instruction set is defined as a (fully enumerated) data type in the logic. It describes the *forms* instructions can take. This is mostly automatic and leads to useful properties like structural induction over instructions, the uniqueness of instructions, and cases of instructions.

The behavior or *operational semantics* of an instruction set describes how outputs and memory are changed for each instruction, memory state, and set of inputs, [13]. The semantics are given as *rules.* These rules describe how the new memory state and outputs are *generated* by each instruction and input.

In the following subsections, first we show an example of how a set of rules is translated to an inductively defined relation. Second, we describe how *induction* over the transition rules is used. Third, we show how the syntax of an instruction-set is defined. Fourth, we show how the operational semantics is defined for a simple example. Fifth, we describe the correctness theorem relating the top-level ISA specification to the composition of the component ISA descriptions.

3.1 Inductive Definitions

A simple way to illustrate the use of inductive definitions is to define the sets of even and odd (natural) numbers. The rules for defining even numbers are given below.

$$\frac{}{Even(0)} \tag{1}$$

$$\frac{Even(n)}{Even(n+2)} \tag{2}$$

The informal interpretation is, "0 is even and if n is even, so is $n+2$."

The rules themselves are translated into the logical formulas shown below. Standard predicate calculus notation is used where $\supset$ denotes implication, $\forall$ is *for-all*, and $\exists$ is *exists*. $\vdash$ denotes *theorems*. In general, theorems have the form $\Gamma \vdash t$ where Γ is a list of formulas constituting the *assumptions*; t is a logical formula which is the *conclusion* guaranteed to be true whenever all the assumptions are true. Definitions are denoted by $\vdash_{def}$. The rules, *"0 is even and if n is even so is n+2,"* are given by:

$$Even(0) \tag{3}$$

$$\forall n.Even(n) \supset Even(n+2) \tag{4}$$

The notion that the rules given are the *only* rules means *Even* is defined as the *smallest* relation satisfying the rules – the natural numbers $\{0, 2, 4, \ldots\}$. If we did not exclude all but the smallest set of numbers satisfying the rules, then the *entire* set of natural numbers would also satisfy the rules.

The stipulation that we are interested in only the smallest set is handled by a third formula. The third formula states that the set we are interested in is contained by *all* sets satisfying the rules, i.e. *Even* is an implicant of all relations satisfying the rules. This can be thought of as the *intersection* of all possible solutions. The formula which states this is as follows.

$$\begin{aligned} \forall P.P(0) \wedge (\forall n.P(n) \supset P(n+2)) \\ \supset (\forall n.Even(n) \supset P(n)) \end{aligned} \tag{5}$$

The above formula states for all relations P satisfying the given rules, whenever $Even(n)$ is true, then so is $P(n)$. This means *Even* is contained by all relations P satisfying the rules. Taking the three statements about *Even* is its complete inductive definition. (Note, in HOL parenthesis are sometimes omitted around function applications, e.g. *Even 0 = Even(0).* Applications are *left associative*, e.g. *f a b = ((f(a))b).*

```
⊢def  (Even 0 ∧ (∀n. Even n ⊃  Even(n + 2))) ∧
       (∀P. P 0 ∧ (∀n. P n ⊃  P(n + 2)) ⊃
      (∀n. Even n ⊃  P n))
```

The rules for generating *odd* numbers are as follows.

$$\frac{}{Odd(1)} \tag{6}$$

$$\frac{Odd(n)}{Odd(n+2)} \tag{7}$$

This leads to the inductive definition of *Odd* shown below.

```
⊢def  (Odd 1 ∧ (∀n. Odd n ⊃  Odd(n + 2))) ∧
       (∀P. P 1 ∧ (∀n. P n ⊃  P(n + 2)) ⊃
      (∀n. Odd n ⊃  P n))
```

Rules like the ones above are automatically translated to inductive definitions by HOL, [3].

3.2 Rule Induction

One of the advantages of using inductive definitions is the automatic derivation of theorems based on the rules. For example, an even number n is 0 or there is an even number n' and $n = n' + 2$.

$$\begin{aligned} \forall n.Even(n) = (n = 0) \quad \vee \\ (\exists n'.(n = n' + 2) \wedge Even(n')) \end{aligned} \tag{8}$$

Similarly, a property P can be proved to hold for all even numbers if P holds for all the rules defining even numbers. This way of doing proofs is called *rule induction*, [3]. The rule induction formula for *Even* is as follows.

$$\forall P.P(0) \wedge (\forall n.Even(n) \wedge P(n) \supset P(n+2)) \supset (\forall n.Even(n) \supset P(n)) \quad (9)$$

An example of using rule induction is shown below. Suppose we wish to prove:

$$\forall n.Even(n) \supset Odd(n+1) \quad (10)$$

From the induction theorem on *Even*, we can decompose the goal into two subgoals based on each of the rules defining *Even*. The first subgoal is for $n = 0$; the second subgoal corresponds to the case where n is even and we have to show that $(n+2)+1$ is *Odd*.

$$Odd(0+1) \quad (11)$$

$$Even(n) \quad \wedge \quad Odd(n+1) \supset Odd((n+2)+1) \quad (12)$$

The first subgoal is proved as a direct consequence of the definition of *Odd*. The second subgoal is derivable from the definition of *Odd* after rewriting $Odd((n+2)+1)$ as $Odd((n+1)+2)$ and using the assumption $Odd(n+1)$.

While the example here is simple, rule induction is useful for proving properties of a design whose behavior is described by rules. For example, a design is "correct" if it exhibits "correct" behavior over all transition rules.

3.3 Instruction Syntax

Based on Table 1, the top-level instruction set for the Am2910 is given by the data type definition:

$$\begin{aligned} &Inst = \\ &\quad JZ \mid CJS \mid JMAP \mid CJP \mid PUSH \mid JSRP \mid CJV \mid \\ &\quad JRP \mid RFCT \mid RPCT \mid CRTN \mid CJPP \mid LDCT \mid \\ &\quad LOOP \mid CONT \mid TWB \end{aligned}$$

The effect of the above data type definition is to introduce into the logic a set or *type* called *Inst* consisting of all the Am2910 top-level instructions.

In a similar fashion, the instruction-sets for the stack, register-counter, multiplexer, and microprogram counter are introduced as follows.

$$\begin{aligned} stack &= POP \mid SPUSH \mid HOLD \mid CLEAR \\ regc &= DEC \mid RHOLD \mid LOAD \\ mux &= Dir \mid Reg \mid Sta \mid Mpc \mid Sel0 \\ mpc &= MCLEAR \mid COUNT \end{aligned}$$

Once the instruction sets have been introduced, useful theorems like instruction cases, induction, and uniqueness over the instructions are proved automatically. For example, the cases, induction, and uniqueness theorems for the microprogram counter appear below. Similar theorems for all the other instruction sets are proved automatically by HOL as described by Melham in [11].

$$\forall m.(m = MCLEAR) \vee (m = COUNT) \quad (13)$$

$$\forall P.P(MCLEAR) \wedge P(COUNT) \supset (\forall m.P(m)) \quad (14)$$

$$\neg(MCLEAR = COUNT) \quad (15)$$

3.4 Instruction Semantics

After the instruction set for a component has been defined, the rules governing its instruction set behavior are defined. For example, consider Table 2 describing the instruction-set behavior of the register-counter.

Table 2: Register-Counter Instruction Set

Mnemonic	Condition	Counter
DEC	$ctr \neq 0$	$ctr - 1$
RHOLD	don't care	ctr
LOAD	don't care	D_i

Table 2 states that *DEC* causes the counter to be decremented by one if it is not already zero; *RHOLD* causes no change in counter state; and *LOAD* changes the state of the counter to input D_i. Notice that *DEC* is not defined when $ctr = 0$, so we must ensure that the instruction decoder does not issue a *DEC* instruction when $ctr = 0$.

The corresponding set of rules specifying the behavior of the register-counter follow. The rules inductively define the transition relation $\stackrel{rcexe}{\rightarrow}$ between the register-counter instruction, input D_i, the state of the counter ctr, and the next-state of the counter.

$$\frac{\neg(ctr = 0)}{(DEC\ D_i\ ctr) \stackrel{rcexe}{\rightarrow} (ctr - 1)} \quad (16)$$

$$\frac{}{(RHOLD\ D_i\ ctr) \stackrel{rcexe}{\rightarrow} ctr} \quad (17)$$

$$\frac{}{(LOAD\ D_i\ ctr) \stackrel{rcexe}{\rightarrow} D_i} \quad (18)$$

The inductive definition corresponding to the rules is as follows.

```
⊢def  ((∀ctr. ¬(ctr = 0) ⊃
        (∀Di. rcexe DEC Di ctr(ctr − 1))) ∧
        (∀Di ctr. rcexe RHOLD Di ctr ctr) ∧
        (∀Di ctr. rcexe LOAD Di ctr Di)) ∧
(∀P.
 (∀ctr. ¬(ctr = 0) ⊃  (∀Di. P DEC Di ctr(ctr − 1))) ∧
 (∀Di ctr. P RHOLD Di ctr ctr) ∧
 (∀Di ctr. P LOAD Di ctr Di) ⊃
 (∀C Di ctr newctr. rcexe C Di ctr newctr ⊃
  P C Di ctr newctr))
```

Note that in all the rules defining the instruction-set behavior of the register-counter, the instructions are symbolic and the data are natural numbers. Boolean encodings of instructions and numbers are not used at this level. They are part of the lower-level implementation.

The multiplexer, stack, microprogram counter, and top-level Am2910 description are defined in a manner similar to the register-counter. For example, the rule describing the *next-state* behavior of the Am2910 for a *CONTinue* instruction, as defined in Table 1, is shown below in (22). The state of the Am2910 is given by *regc* – the register-counter; *mpc* – the microprogram counter; and *st* – the stack. The next state is determined by the current state; the instruction *CONT*; the active-low condition code enable and the condition code *CCEN_B* and *CC_B*; D_i – the direct data input; C_i – the carry input into the incrementer; and *RLD_B* – the active-low register-counter load control line. The functions $HoldCtr$, $mpcCnt$, and $Hold$ which operate on the register-counter, microprogram counter, and stack, are defined below. Note: $cond \rightarrow A|B$ means "*if cond is true then A else B.*" (22) says that if the stack length is less-than-or-equal to 5, then the $CONT$ instruction will cause the register-counter contents to be held subject to *RLD_B*; *mpc* will be incremented if $C_i = 1$; and the stack *st* remains unchanged. The word length of the Am2910 is parameterized by n.

$$HoldCtr(RLD\_B, Ctr, Di) = RLD\_B \rightarrow Ctr \mid Di \quad (19)$$

$$mpcCnt(mpc, Ci, n) = (mpc + Ci) MOD (2^{n+1}) \quad (20)$$

$$Hold(st) = st \quad (21)$$

$$\frac{LENGTH(st) \leq 5}{\begin{pmatrix} n\ CONT \\ CCEN\_B \\ CC\_B\ Di \\ Ci\ RLD\_B \\ regc\ mpc\ st \end{pmatrix} \xrightarrow{nextstate} \begin{pmatrix} (HoldCtr \\ (RLD\_B, regc, Di)) \\ (mpcCnt \\ (mpc, Ci, n)) \\ (Hold(st)) \end{pmatrix}} \quad (22)$$

The instruction decoder is defined as a large *if-then-else* expression. Each Am2910 instruction is broken down into individual component instructions depending on the input signals. A fragment of the definition corresponding to the *RFCT* instruction is shown below. Note the register-counter instruction depends on whether the register-counter value is zero or not.

$$\begin{array}{l} \quad \ldots \\ (C = RFCT) \rightarrow \\ \qquad ((rc\_inst = (RLD\_B = F) \rightarrow \\ \qquad LOAD \mid ((Zero = T) \rightarrow RHOLD \mid DEC)) \wedge \\ \qquad (st\_inst = (Zero = T) \rightarrow POP \mid HOLD) \wedge \\ \qquad (pc\_inst = COUNT) \wedge \\ \qquad (mx\_inst = (Zero = T) \rightarrow Mpc \mid Sta) \wedge \\ \qquad (PL\_B = F) \wedge (MAP\_B = T) \wedge \\ \qquad (VECT\_B = T)) \\ \quad \ldots \end{array}$$

3.5 Instruction-set Verification

Once the top-level instruction-set descriptions for the Am2910 and its components are defined, the design at the instruction-set level can be verified. This amounts to verifying that the behavior of the block diagram in Figure 2 is completely covered by the top-level instruction-set behavior specified by Table 1.

Our top-level instruction-set behavioral description had six modules: *zero detector, instruction decoder, register-counter, multiplexer, program counter,* and *stack*. They are denoted by the following terms.

$$\begin{array}{l} zero\ detector = zero\_detect(rc, zero) \\ instruction\ decoder = \\ \qquad Inst\_decode \\ \qquad (C, CCEN\_B, CC\_B, RLD\_B, zero) \\ \qquad (rc\_inst, mx\_inst, pc\_inst, st\_inst, PL\_B, \\ \qquad\ MAP\_B, VECT\_B) \\ program\ counter = mpcexe\ n\ pc\_inst\ ci\ Y\ new\_pc \\ register\ counter = \\ \quad ((rc\_inst = DEC) \rightarrow \\ \quad (\neg(rc = 0) \supset rcexe\ rc\_inst\ din\ rc\ new\_rc) \mid \\ \qquad rcexe\ rc\_inst\ din\ rc\ new\_rc) \\ multiplexer = muxexe\ mx\_inst\ din\ rc\ st\_out\ pc\ Y \\ stack = \\ \qquad ((LENGTH st) \leq 5 \supset \\ \qquad (NULL\ st \rightarrow \\ \qquad sta0exe\ st\_inst\ 5\ pc\ st\ new\_st\ full\_b \mid \\ \qquad staexe\ st\_inst\ 5\ pc\ st\ new\_st\ st\_out\ full\_b)) \wedge \end{array}$$

$((st\_inst = CLEAR) \supset$
$\quad staCexe\ st\_inst\ 5\ pc\ st\ new\_st)$

The instruction-set behavior of the Am2910 is denoted by:

$$
\begin{aligned}
&Am2910Spec = \\
&\quad Am2910\_spec \\
&\quad (n, C, CCEN\_B, CC\_B, din, ci, RLD\_B, rc, pc, st) \\
&\quad (Y, PL\_B, MAP\_B, VECT\_B, full\_b) \\
&\quad (new\_rc, new\_pc, new\_st)
\end{aligned}
$$

The instruction-set correctness theorem has the form (*zero detector*) $\wedge$ (*instruction decoder*) $\wedge$ (*register counter*) $\wedge$ (*multiplexer*) $\wedge$ (*program counter*) $\wedge$ (*stack*) $\supset$ (*Am2910Spec*). Instruction-set composition is obtained by $\wedge$, i.e. the logical conjunction of all the predicates constraining the input/output/next-state behavior as described by Gordon in [8]. The actual correctness theorem *Compose_correct* as it appears in HOL is as follows:

```
Compose_correct
⊢  zero_detect(rc,zero) ∧
 Inst_decode(C,CCEN_B,CC_B,RLD_B,zero)
  (rc_inst,mx_inst,pc_inst,st_inst,PL_B,MAP_B,VECT_B) ∧
 ((rc_inst = DEC) →
  ¬(rc = 0) ⊃  rcexe rc_inst din rc new_rc) |
  rcexe rc_inst din rc new_rc) ∧
  muxexe mx_inst din rc st_out pc Y ∧
  mpcexe n pc_inst ci Y new_pc ∧
  ((LENGTH st) ≤ 5 ⊃
   (NULL st →
     sta0exe st_inst 5 pc st new_st full_b |
     staexe st_inst 5 pc st new_st st_out full_b)) ∧
  ((st_inst = CLEAR) ⊃  staCexe st_inst 5 pc st new_st)
  ⊃
 Am2910_spec(n,C,CCEN_B,CC_B,din,ci,RLD_B,rc,pc,st)
  (Y,PL_B,MAP_B,VECT_B,full_b)(new_rc,new_pc,new_st)
```

While the details of the proof cannot be presented here due to space limitations, the theorem was proved by 1) doing a case analysis on all the top-level Am2910 instructions, 2) rewriting to get the individual component-level instructions from the instruction decoder, 3) rewriting the Am2910 specification and component definitions to relate implementation and specification behavior. The proof was highly repetitive and it is likely the parameterized techniques described by Windley in [16] could be used.

4 Module Descriptions

In this section, we show how parameterized implementations are described. As an example, we

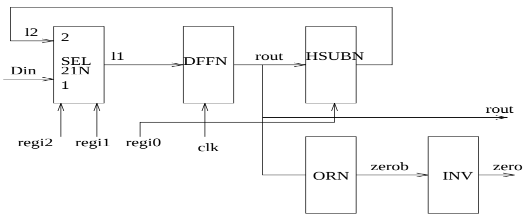

Figure 4: Register-Counter Implementation

show some of the details of the register-counter whose instruction-set behavior was described previously.

The register-counter implementation is built around three principal components: an n-bit 2-1 multiplexer – *SEL2N*, an n-bit register – *DFFN*, and an n-bit half subtracter – *HSUBN*. Zero detection is computed by an array of *OR* gates – *ORN*, and an inverter – *INV* to generate an active high zero flag. Figure 4 shows the block diagram.

The HOL definition appears below. All inputs are boolean signals, i.e. functions from time to booleans where time is modeled discretely by the natural numbers. Internal signals are hidden from view by existential quantification, [8]. All other signals are universally quantified. The *DEC, RHOLD,* and *LOAD* commands are encoded by the signals *regi0, regi1* and *regi2*. The direct data input is *din*. The output is *rout*, the clock is *clk*, and the zero flag is *zero*. n is the word length parameter.

```
REGCNTR
  ⊢def  ∀n din clk zero regi0 regi1 regi2 rout.
      REGCNTR n din clk zero regi0 regi1 regi2 rout =
      (∃l1 l2 zin reginxt zerob.
        SEL21N n din l2 regi1 regi2 l1 ∧
        DFFN n l1 clk rout ∧
        gnd zin ∧
        ORN n rout zin zerob ∧
        inv(zerob,zero) ∧
        HSUBN n rout regi0 l2 reginxt)
```

REGCNTR is an n-bit register which is itself constructed from other n-bit components. Ultimately, these components have a recursive definition corresponding to an iterative array of cells. For example, an iterative array of *or* gates is created by *ORN*. A two-input *or* gate is a three port device with inputs x and y and an output *out*. It exists as part of the GDT standard-cell library and is defined as follows.

$$or(x, y, out) = \forall t. out(t) = x(t) \vee y(t) \tag{23}$$

$$\forall x\ y\ out.ORN\ 0\ x\ y\ out = or(x0, y, out) \quad (24)$$

$$\forall\ n\ x\ y\ out.ORN(n+1)\ x\ y\ out = \quad (25)$$

$$\exists l1.ORN\ n\ x\ y\ l1 \wedge or(x(n+1), l1, out) \quad (26)$$

Given the definition of *ORN*, the expression *ORN 2 x y out* is equivalent to $or(x(0), y, l1) \wedge or(x(1), l1, l2) \wedge or(x(2), l2, out)$, i.e. a linear array of *or* gates with internal lines *l1* and *l2*.

5 Layout Compilation

Once the components are defined in HOL, they are compiled into cell generator programs that are executed by GDT to create layouts. The details of how objects in HOL are related to objects in GDT is described by Lu and Chin in [10] so we only give a brief description below. While the compilation is straightforward, it is crucial to the design process as it allows us to rapidly simulate logic and layouts derived from formal definitions within GDT using the Lsim simulator. Such simulations validate the definitions and provide timing information. Changes in design are quickly implemented since schematic capture is completely bypassed. The layout in Figure 1 is generated by the top-level Am2910 cell generator in GDT.

The correspondence between HOL objects and GDT objects is as follows.

- Universally quantified (∀) variables are input or output ports designated by `IN` or `OUT`.
- Existentially quantified variables (∃) are internal signals designated by `SIG` or `WIRE`.
- Predicates (e.g. *inv*) correspond to cell module names or instances (e.g. `INST inv inv[0]`, "`inv[0]` is an instance of `inv`").
- Recursive definitions correspond to iterative module generators.

Cell generators in GDT are written in the L language, [12]. L is a C-like language which describes cell connectivity. An example L program for the *ORN* module generator is shown below with a default size of 12.

```
SCHEMATIC ORN (INT N = 12)
{IN y;
OUT out;
INT I;
I = 0;
WHILE(I < N) (
   IN x[I];
   INST or or[I];
   WIRE or[I].in[0] TO x[I];
   IF(I == 0) WIRE or[0].in[1] TO y;
   IF(I != 0) WIRE or[I-1].out TO or[I].in[1];
   IF(I == N - 1) WIRE or[I].out TO out;
 I++; )}
```

6 Cell to Instruction-set Level Verification

At this point in the design process, the collection of components at the ISA level has been shown to implement the top-level ISA specification, and the gate-level cell definitions have been compiled into layouts and simulated. What remains to be done is to verify that each of the gate-level definitions satisfy their ISA definitions.

As an example, we show how the register-counter implementation in Section 4 is related to its ISA behavioral description in Section 3.5. The signal names used in the register-counter implementation are n-bit. Given a signal f, the n^{th}-bit of f at time t is $f(n)(t)$. Eventually, we will compute the numerical value of a word from a *list* of boolean values where the most-significant bit appears at the head of the list and the least-significant bit is $f(0)(t)$. The HOL definition of the function *word* which given a signal f creates a list of boolean values at time t appears below. It builds the list of values least-significant-bit first. (Note: *CONS* is a list operator which pushes elements onto the front of lists, e.g. *CONS 1 [2;3] = [1;2;3]*; and *SUC* increments numbers by one, e.g. *SUC 0 = 1*).

$$\forall t\ f.word\ 0\ t\ f = [f\ 0\ t] \quad (27)$$

$$\forall n\ t\ f.word(SUC\ n)t\ f =$$

$$CONS(f(SUC\ n)t)(word\ n\ t\ f)) \quad (28)$$

Given a signal, say D_i, its numerical value at time t is computed using functions provided by the HOL *word* library created by Wong, [17]. Specifically,

$$Di = NVAL\ BV\ 2(WORD(word\ n\ t\ din)) \quad (29)$$

where BV is a function mapping *true* and *false* to 1 and 0; $WORD$ takes a list of booleans and maps it into an object of type *word*; and $NVAL$ is a higher-order function using BV to map the elements of the word $WORD(word\ n\ t\ din)$ into 1s and 0s which in turn is interpreted as a base 2 number. For example, $NVAL\ BV\ 2\ WORD[T;T;F] = 6$.

The boolean encodings of the three register-counter instructions is given by *rcinst_abs* defined below.

```
rcinst_abs
⊢def  ∀c0 c1 c2.rcinst_abs(c0,c1,c2) =
       ((c0,c1,c2 = F,T,T) →
        DEC | ((c0,c1,c2 = F,T,F) →  RHOLD | LOAD))
```

Valid register-counter instruction codes are defined by *IS_rcinst*.

```
IS_rcinst
⊢def  ∀c0 c1 c2.IS_rcinst(c0,c1,c2) =
       (c0,c1,c2 = F,T,T) ∨ (c0,c1,c2 = F,T,F) ∨
       (c0,c1 = T,F)
```

The numerical values of D_i, ctr, and new_ctr are defined below.

$$Di = NVAL\ BV\ 2(WORD(word\ n\ t\ din)) \quad (30)$$

$$ctr = NVAL\ BV\ 2(WORD(word\ n\ t\ rout)) \quad (31)$$

$$new\_ctr = NVAL\ BV\ 2(WORD(word\ n\ (t+1)\ rout)) \quad (32)$$

$$C = rcinst\_abs(rc0\ t, rc1\ t, rc2\ t) \quad (33)$$

Using the above interpretations, the following theorem holds between the register-counter implementation *REGCNTR* and its ISA behavioral description given by $((C = DEC) \rightarrow (\ (ctr = 0) \supset rcexe\ C\ Di\ ctr\ new\_ctr) \mid rcexe\ C\ Di\ ctr\ new\_ctr))$. Note that this is the same ISA description as given in Section 3.5.

$$
\begin{aligned}
&REGCNTR\ n\ din\ clk\ zero\ rc2\ rc0\ rc1\ rout \supset\\
&(\forall t. IS\_rcinst(rc0\ t, rc1\ t, rc2\ t) \supset\\
&\ ((C = DEC) \rightarrow\\
&\qquad (\neg(ctr = 0) \supset rcexe\ C\ Di\ ctr\ new\_ctr) \mid\\
&\qquad rcexe\ C\ Di\ ctr\ new\_ctr))
\end{aligned}
$$

The proof of the above theorem is not given here. The general procedure is 1) to prove the correctness of the n-bit components like *ORN* using induction on n followed by 2) proving the *implementation* $\supset$ *specification* theorem using case analysis on the instructions. Having the *IS_rcinst* predicate reduces the number of cases to only the valid instruction encodings.

Similar theorems are proved relating the remaining component implementations to their ISA behavioral descriptions. By transitivity using the correctness of the composed ISA behavior, the composition of the gate-level descriptions satisfies the top-level ISA specification. At this point, the verification is complete from top-to-bottom and parameterized cell generators and layouts exist for the design.

7 Conclusion

The strategy underlying our approach is based on

1. Letting each tool do what it is best at. This means using HOL to verify high-level properties, letting cell generation and instantiation occur in GDT instead of HOL, and doing performance analysis using Lsim in GDT.

2. Quickly checking designs by simulation and postponing costly activities like verification until designs have been validated by simulation.

3. Making formal verification a non-recurring cost by creating parameterized cell generators.

Our experience leads us to believe that incorporating higher-order logic into the design process is practical and enables reliable and reusable designs to be created quickly. The use of inductive definitions works well for describing instruction-set behavior. Such an approach should work for finite-state machines as well since they can be specified in an operational manner. Our approach compiling cell generators from formal definitions saves considerable time and effort when compared with approaches using schematic capture. Formal verification of behavior requires explicit definitions of initial conditions, interpretations, and abstraction functions. Such explicitness enables the correct re-use and embedding of components in larger designs.

One remaining issue is the correctness of the HOL2L compiler. We plan to address this using techniques similar to net list comparison checkers which compare net lists to layouts. Net lists created by interpreting the parameterized gate-level formal descriptions can be compared with net lists created by cell generators. This approach seems more practical when compared with alternatives like assigning an arbitrary formal semantics to the L language and verifying the compiler. This approach has been used by Verkest, Claesen, and De Man [15] in their integration of the Boyer-Moore prover [2] with the Cathedral-II silicon compiler, [6].

Based on the success of this effort, a similar process is being used to design a 32-bit processor for Rome Laboratory, Griffiss Air Force Base, NY. This effort extends the work reported here by incorporating test hardware such as boundary scan registers and controllers into the formal design process. Fabrication of the processor at MOSIS should occur by the end of the year. The level-of-effort on this project is two students working half time.

8 Acknowledgments

This work was supported in part by the NY State Center for Advanced Technology in Computer Applic-

ations and Software Engineering (CASE) at Syracuse University.

References

[1] Advanced Micro Devices Inc., *AMD Microprocessor Logic and Interface - Am2900 family data book*, Sunnyvale, CA, 1985.

[2] R.S. Boyer, J.S. Moore, *A Computational Logic*, Academic Press, New York, 1979.

[3] J. Camilleri, T. Melham, *Reasoning with Inductively Defined Relations in the HOL Theorem Prover,* University of Cambridge Technical Report, Computer Laboratory, University of Cambridge, August 1992.

[4] A. V. Chavan, *Technology mapping of a standard-cell design library to an FPGA design environment*, CASE Center Technical Report No. 9320, Syracuse University, Syracuse, New York, 1994.

[5] Chrysalis Symbolic Design, Inc., Reference Manuals, Release 1.05, Andover, MA, July 1994.

[6] H. De Man, J. Rabaey, P. Six, L. Claesen, "Cathedral-II: A Silicon Compiler for Digital Signal Processing," *IEEE Design and Test of Computers*, Vol. 3, No. 6, pp. 73–85, December, 1986.

[7] M.J.C. Gordon, "A proof generating system for higher-order logic," in *VLSI specification, verification and synthesis*, edited by G. Birtwistle and P. A. Subramanyam, Kluwer, 1987.

[8] M.J.C. Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware," in *Formal Aspects of VLSI Design,* edited by G. Milne & P.A. Subrahmanyam, North Holland, 1986.

[9] R. Kumar, K. Schneider, T. Kropf, "Structuring and Automating Hardware Proofs in a Higher-Order Theorem-Proving Environment," *Formal Methods in System Design*, 2, pp. 165–223, 1993.

[10] J. Y. Lu, S. K. Chin, "Linking HOL to a VLSI CAD system," *Higher Order Logic Theorem Proving and Its Applications*, Lecture Notes in Computer Science 780, Springer-Verlag, Berlin Heidelberg 1994

[11] T. Melham, "Automating Recursive Type Definitions in Higher Order Logic," *Current Trends in Hardware Verification and Automated Theorem Proving,* edited by G. Birtwistle and P. Subrahmanyam, Spinger-Verlag, 1989, pp. 341-386.

[12] Mentor Graphics Inc., *GDT Led, Lx Standard Cell, Explorer Lsim V.5.3 users manuals*, San Jose, CA, 1990.

[13] H. R. Nielson,F. Nielson, *Semantics with Applications*, John Wiley, New York, 1992.

[14] D.Siewiorek, C. Bell, A. Newell, *Computer Structures - Principles and Examples*, McGraw-Hill, New York, 1982.

[15] D. Verkest, L. Claesen, H. De Man, "Correctness Proofs of Parameterized Hardware Modules in the Cathedral-II Synthesis Environment," in *European Design Automation Conference, EDAC'90*, pp. 62–66. Glasgow, Scotland, March 1990.

[16] P.J. Windley, "Formal Modeling and Verification of Microprocessors," *IEEE Trans. Comp.*, Vol. 44, No. 1, Jan. 1995, pp. 54–72.

[17] W. Wong, "Modelling bit vectors in HOL: the word library," Proc. of 6th Intl. HOL Users Group Workshop 1993, Vancouver, B.C, Canada, August 1993, Springer-Verlag, New York, 1994.

[18] Xilinx Inc., *Xilinx users manual and Programmable gate array data book*, San Jose, CA, 1990.