A State Machine Language Supporting Integer Inequalities Implemented with Ordered Ternary Decision Diagrams

Glenn Jennings, Per Lindgren and Joachim Isaksson
Division of Computer Engineering
Luleå Institute of Technology
S-971 87 Luleå, Sweden
glenn@sm.luth.se

Abstract

A state-machine language supporting both integer and Boolean input and output signals permits unified description throughout the design process, permits rapid interactive debugging, generates efficient behavioral simulation code, and produces compact but non-compromised output for logic optimization. The compiler is built upon Ordered Ternary Decision Diagrams, which insures correct simulation behavior in the presence of "unknown" input values. Mapping of integer tests into OTDDs, and the effects of OTDD bit ordering on simulation code size and efficiency, are briefly discussed.

1 Introduction

The first step toward an optimized PLA or logic structure is the compilation of the designer's original symbolic description. However, the compiler is more than just the front-end to the logic optimizer, it is the first line of defense as well. If the compiler is inefficient, then it will be slow in detecting logical errors, such as the error shown in Figure 1. This would decrease the productivity of the designer, who must be allowed to iterate rapidly until the description contains no more such errors. The compiler must also deliver the description to the optimizer in as minimal a format as possible so as to avoid overloading the optimizer, but at the same time it must *not* attempt any optimizations itself, that is, it must not remove any of the opportunities for optimization which are present in the original description.

Finally, if the compiler is also part of a simulation environment, then it may have the task of *quickly* generating *efficient behavioral code* from the designer's description. In that case the compiler may choose to avoid the optimizer altogether, since good optimizers seldom have fast execution times. Therefore the burdens on the compiler are not only considerable, but as seen in the list above, they can even be in conflict with one another.

We encountered all of those requirements, and as if those were not enough, it also appeared that we had to work in both Boolean *and integer* domains. This is because our compiler is part of a register transfer level synthesis environment called GRTL [Jen91a] [Jen91b] which allows circuit interconnections (that is, "signals") to be of type *integer* as well as *bits* (or vectors of bits). The integer signal abstraction has proven valuable to us in the early stages of design, since it allows us to specify a signal even though we do not yet know how many bits that signal is going to require.

```
;----------------------------------
;  example. 4 bits in, 7 bits out.
;
IFIN      1XX0          OUTPUT 110XXXX
IFIN      00XX OR X110  OUTPUT XX10X01
OTHERWISE               OUTPUT 01X10X1
```

Figure 1: This typical state machine language tests the four-bit input vector against conditions which divide the $2^4 = 16$ possible configurations into three sets, each set having its own seven-bit output vector. "X" means "don't care." Note that the first two sets overlap ("1110" belongs to both sets), which must be detected by the state machine language compiler.

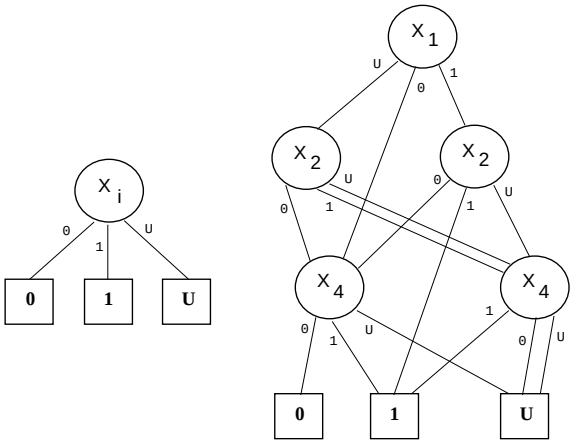

Figure 2: Simple Ordered Ternary Decision Diagrams. To left: the identity function for variable x_i. To right: the function $(x_1 \wedge x_2) \vee x_4$.

This means that, at the early stages of the design process, there may be integer signals appearing as input or output signals to state machine descriptions and other functional entities which are ultimately to be *implemented* as PLA or optimized Boolean logic. Therefore the designer must be able to manage *integer* inputs and outputs in combination with Boolean input and outputs, for these early component descriptions. Not only are there integer abstractions, but in both integer and Boolean domains there are values which represent "unknown" values during (multivalued) simulation, as in [L+88] where such a value is used as the initial simulation value for all binary signals. Current languages such as VHDL would force the designer to write in a tedious behavioral style (as in Figure 7). This not only frustrates the interface to the implementation tool, but it is also contrary to the long tradition among digital designers of using high-level finite state machine languages [BC87]. What we needed was to combine that tradition together with the idea of integer signal abstraction, and furthermore make automatic provision for handling "unknown" values in both domains. In that way, as integer signals were resolved into Boolean signals, only incremental changes would need to be made to a single description (as in Figure 3) which otherwise remains common and largely intact throughout the entire design process.

We succeeded in handling this, which we report here as the GRTL state machine language called **GPLA**. We were able to *implement* this entirely in the Boolean domain, which allowed us to make use of *Ordered Ternary Decision Diagrams* (henceforth OTDDs) (see Figure 2). These are closely related to the highly efficient OBDDs described in [Bry86]. The result is that GPLA has exceeded all our performance expectations while meeting *all* of the conflicting requirements mentioned above.

2 The Specification Language

The language itself has a conservative PLA-type specification format, where the collection of input vectors are tested against constant conditions, and output configurations (including "next state") are specified for each such condition, similar to the style of Figure 1. We must also be able to test *integer* input signals against integer expressions as seen in Figure 3. The actual grammar of GPLA (implemented with the help of the UNIX utility **yacc**) is shown in abbreviated form in Figure 4. In Figure 3 we see the first five lines declare the names, types, and (if of type binary) the widths of the logical signals, and whether they are input or output signals. After this follows a description of a state machine having two states, and each state has a conditional structure (**if**... **elsif**... **else**).

```
INSIG     int_in   INTEGER
INSIG     bin_in   BINARY  5
INSIG     isig     INTEGER
OUTSIG    int_out  INTEGER
STATESIGS int_in int_out

ON (bin_in == '--0-1)  ->
     GOTO state_a

state_a :
  IF isig > 2  ->
       REPEAT
  ELSIF (isig < 7) & (isig > 1)  ->
       NEXT
  ELSE REPEAT

  IF isig <= 2  ->
       REPEAT
  ELSIF isig != 4   ->
       GOTO state_a
  ELSE REPEAT
```

Figure 3: A two-state FSM description with integer state signals (current state = "int_in" and next state = "int_out") and two additional input signals, one binary ("bin_in") and one integer ("isig"). Only the first state is symbolically named ("state_a"). An overriding ON-condition appears before the first state.

It is the management of the input signal value space that is of interest to us here. In essence we wish to concatenate both the input state signal with the other incoming signals into one long vector, which would define for us the possible configurations we can expect

plaline	ϕ	
	| **insig** *id* **integer**	| **insig** *id* **binary** expr
	| **outsig** *id* **integer**	| **outsig** *id* **binary** expr
	| **vector** *id* binlist	| **vector** *id* boutlist
	| **condition** *id* cond	
	| **set** *id* outlist	
	| **statesigs** *int_in_id int_out_id*	
	| **statesigs** *bit_in_id bit_out_id*	
	| onstate	
	| state	
state	slabel sbody	
slabel	ϕ	
	| *id* **:**	
	| **others :**	
sbody	uncond	| ifwork
ifwork	**if** ifbody ifrest	
ifbody	cond $\mapsto$ sbody	
ifrest	moreif **else** sbody	
moreif	ϕ	| moreif **elsif** ifbody
onstate	**on** cond $\mapsto$ uncond	
uncond	**stub**	| outseq action
	| outseq	| action
outseq	**donotcare**	| outlist
action	**goto** *id*	
	| **next**	| **repeat**
cond	cond2	| cond2 $\vee$ cond2
cond2	cond3	| cond3 $\oplus$ cond3
cond3	cond4	| cond4 $\wedge$ cond4
cond4	cond5	| cond5 $\setminus$ cond5
cond5	cond6	| $\neg$ cond6
cond6	*int_in_id* = expr	| *int_in_id* $\neq$ expr
	| *int_in_id* > expr	| *int_in_id* $\geq$ expr
	| *int_in_id* < expr	| *int_in_id* $\leq$ expr
	| expr > *int_in_id*	| expr $\geq$ *int_in_id*
	| expr < *int_in_id*	| expr $\leq$ *int_in_id*
	| *cond_id*	| *bit_in_id*
	| *bit_in_id* = *cube*	| *bit_in_id* $\neq$ *cube*
	| (cond)	
outlist	outs outs2	
outs2	ϕ	| outlist
outs	*int_out_id* := expr	| *set_id*
	| *bit_out_id*	| $\neg$ *bit_out_id*
	| *bit_out_id* := *cube*	| (outlist)
binlist	u_linb bin2	
bin2	ϕ	| binlist
u_linb	*bit_in_id* [expr]	| *bit_in_id*
	| *bit_in_id* [expr : expr]	
boutlist	u_loutb bout2	
bout2	ϕ	| boutlist
u_loutb	*bit_out_id* [expr]	| *bit_out_id*
	| *bit_out_id* [expr : expr]	
expr	expr2	
	| expr + expr2	| expr $-$ expr2
expr2	expr3	| $-$ expr3
expr3	*integer_literal*	| (expr)

Figure 4: From the "yacc" grammar for GPLA.

(Figure 5). This is complicated by two things: first, the *binary* signals arriving into the *simulation* model of the code of Figure 3 are essentially three-valued (**1, 0,** and **X** indicating an unknown value), the latter which we do not permit to be tested for explicitly. Second of course is that part of our concatenated input vector consists of *integer* signals (Figure 5) so that we do not yet know how many bits those signals actually occupy. Furthermore during *simulation,* any integer value in the range of the machine could appear at that input, *and* there are reserved integer values which represent integer *unknown.* It is this mixed integer-Boolean value space which must be managed by GPLA.

3 Mapping to OTDDs

It is the conventional expression style of the language that enables us to solve this management problem entirely in the Boolean domain. Whenever an integer comparison occurs in the input text description, it consists of an integer signal being compared to an integer expression whose value is known (constant) during compilation. This is apparent from the grammar in Figure 4. We make use of this by making two passes over the text description. On the first pass we identify all the integer values which occur in comparisons against each integer signal. For example, in Figure 3 we can see that signal "isig" is involved in comparisons against the integers 1, 2, 4 and 7. These four integers partition the set of integers into eight regions, see Figure 5. If we assign a *Boolean label* to each such integer region, then that label requires only three bits to represent. In this way we have established a mapping from integer to Boolean domains, and in Figure 5 we allocate three OTDD bits to the representation of integer signal "isig." If the *input state* is represented by an integer signal, then in the first pass we count the number of states, and then create a similar mapping based on that count.

At the end of the first pass, the total number of OTDD bits is known, see Figure 6. An ordering for those bits is then chosen (see Section 5 below), and then a second pass over the text description is made to encode the input conditions as OTDDs. The *apply* function [Bry86] allows us to deal with very complex expressions very efficiently; we support logical **and, or, not, exclusive-or,** and **set difference** operators, see Figure 4. Once these are built, we make the "OTDD pass" (Figure 6) based entirely on OTDD computations, which checks the description for validity. Overlaps of the type shown in Figure 1 (as well as

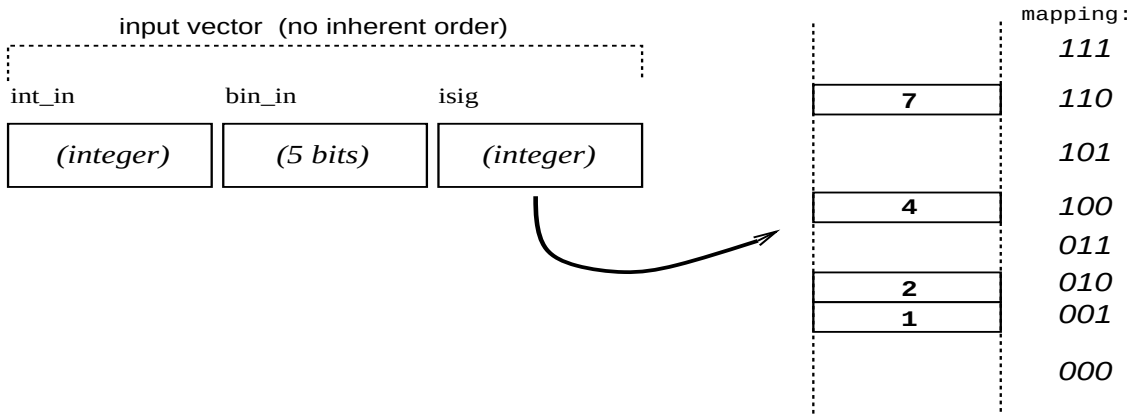

Figure 5: To the left of this Figure is the conceptual input vector formed by concatenating the inputs for the GPLA description in Figure 3; no ordering among the inputs is implied. Fragments of that vector are in the integer domain, but by analysis of the text description, the integer space belonging to each integer input signal can be broken up into a finite, and usually small, number of partitions. Determining the number of partitions for each integer input signal allows GPLA to allocate a number of bits to that signal in order to represent it in the OTDDs for the entire input vector, that is, in the Boolean domain. A unique bit configuration is then assigned to each partition of the integer domain, as shown to the right of the Figure.

empty input condition sets) are detected here, and the portion of the input vector still unused is computed. In the end the entire description is reduced to a set of pairs (A, B) where in each pair, the input condition A is represented by an OTDD, and the output configuration B for that condition is represented in a canonical form. All of the A members represent mutually exclusive fragments of the input vector (Figure 5). A compression pass follows in which all pairs having identical output members B are combined; this does not compromise opportunities for optimization, since all A are disjoint, and combined B are identical; but this minimizes the size of the compiler output. If producing simulation code only, a more aggressive compression can be applied. All that remains is to emit the compressed pairs (A, B) more or less directly as simulation code, or else into the form expected by the optimizer (which we allow only when there are no integer signals in the description, see Figure 8). However, the compiler is not allowed to perform state assignment, which is a task belonging to the optimizer. Therefore state names are emitted symbolically, or in the case of simulation code, encoded provisionally. This of course places a severe restriction on the compression pass.

```
grtl_pla starting:
filename:                          e.pla
parsing completed:        0.07 sec
first pass completed:     0.01 sec
number of otdd bits:        10
second pass completed:    0.06 sec
otdd pass completed:      0.07 sec
compression completed:    0.05 sec 2 cases
code text generated:      0.75 sec
writing code file:                 e.c
code file written:        0.25 sec
memory nodes used:         546
grtl_pla successful:      1.28 sec total
```

Figure 6: Execution statistics on a Sun 4 ELC SPARCstation for the compiling of the description in Figure 3 into the C code of Figure 7.

4 Validations and Results

We developed an "integer-PLA" test suite ranging from very simple state-free logic, to very complex finite state machines which combined integer-signal and bit-signal test conditions together. The suite included some smaller PLAs to test simple Boolean compositions $(xor, not, or, and...)$, for example

```
$if  NOT ((P < 4) XOR (Q > 7)) $then ..
```

Each member in the test suite was processed by the integer-supporting OTDD-based implementation, and the resulting simulation code was compiled and subjected to a number of different tests.

```
#include <stdio.h>
#define MX 2147438647
int _e(INT_IN,BIN_IN,ISIG,INT_OUT)
int INT_IN;
char *BIN_IN; /* 5 bits */
int ISIG;
int *INT_OUT;
{ _0: G_IDRIVE(INT_OUT,(-MX));
_1: if (BIN_IN[2] == '0') {
if (BIN_IN[0] == '0') {
if ((ISIG >= 5)||(ISIG == 3)||
(ISIG == -MX)) { goto _3; /* int_in */
} else if ((ISIG == 4)||(ISIG <= 1)) {
if ((INT_IN == 1)) { goto _2;
} else goto _3; /* int_in */
} else if ((ISIG == 2)) {
if (((INT_IN >= 0)&&(INT_IN <= 1))) { goto _2;
} else goto _3; /* int_in */
} else goto _3; /* isig */
} else if (BIN_IN[0] == '1') { goto _3;
} else { /* X */
if ((ISIG >= 5)||((ISIG >= 2)&&(ISIG <=3))||
(ISIG == -MX)) { goto _3; /* int_in */
} else if ((ISIG == 4)||(ISIG <= 1)) { goto _3; /* int_in */
} else goto _3; /* isig */ }
} else if (BIN_IN[2] == '1') {
if ((ISIG >= 5)||(ISIG == 3)||
(ISIG == -MX)) { goto _3; /* int_in */
} else if ((ISIG == 4)||(ISIG <= 1)) {
if ((INT_IN == 1)) { goto _2;
} else goto _3; /* int_in */
} else if ((ISIG == 2)) {
if (((INT_IN >= 0)&&(INT_IN <= 1))) { goto _2;
} else goto _3; /* int_in */
} else goto _3; /* isig */
} else { /* X */
if ((ISIG >= 5)||((ISIG >= 2)&&(ISIG <=3))||
(ISIG == -MX)) { goto _3; /* int_in */
} else if ((ISIG == 4)||(ISIG <= 1)) { goto _3; /* int_in */
} else goto _3; /* isig */ }
_2: G_IDRIVE(INT_OUT,(1)); return;
_3: if (BIN_IN[2] == '0') {
if (BIN_IN[0] == '0') {
if ((ISIG == 2)||(ISIG == MX)||
(ISIG == -MX)) { goto _6; /* int_in */
} else if ((ISIG >= 5)||(ISIG == 3)) {
if (((INT_IN >= 0)&&(INT_IN <= 1))) { goto _5;
} else goto _6; /* int_in */
} else if ((ISIG == 4)||(ISIG <= 1)) {
if ((INT_IN == 0)) { goto _5;
} else goto _6; /* int_in */
} else goto _6; /* isig */
} else if (BIN_IN[0] == '1') { goto _5;
} else goto _4; /* X */
} else if (BIN_IN[2] == '1') {
if ((ISIG == 2)||(ISIG == MX)||
(ISIG == -MX)) { goto _6; /* int_in */
} else if ((ISIG >= 5)||(ISIG == 3)) {
if (((INT_IN >= 0)&&(INT_IN <= 1))) { goto _5;
} else goto _6; /* int_in */
} else if ((ISIG == 4)||(ISIG <= 1)) {
if ((INT_IN == 0)) { goto _5;
} else goto _6; /* int_in */
} else goto _6; /* isig */
} else goto _4; /* X */
_5: G_IDRIVE(INT_OUT,(0)); return;
_6: ; /* fall through */
_4: G_IDRIVE(INT_OUT, MX); return;
} /* end e */
```

Figure 7: Simulation code machine-generated by GPLA for the example in Figure 3.

Since it is conceptually impossible (and even in practice, impractical) to exhaustively apply all possible integer values to each integer input signal, we instead selected a range of integer values which "covered" the domain of values used in the PLA description. For example, given a signal Q tested by

```
$if ((Q < 4) OR (Q > 7)) $then ..
```

then the values from 3 to 8 inclusive, plus the "unknown" integer values, were applied to signal Q. The tests for bit-valued signals were generated essentially exhaustively, and also included the "unknown" logic value ('X'). We started with the smaller PLAs in the suite, those used to test basic Boolean operations on integer comparisons. These were not only thoroughly tested, but the generated code was carefully scrutinized so as to discover any flaws in either theory or implementation.

Once we were convinced that the basic operations had been correctly implemented, we tested more and more complex descriptions which included multiple states, "reset" conditions including simultaneous integer and bit-signal tests, complicated $IF-ELSIF-ELSIF-$ structures, and so forth. These were tested by examining the output values produced for different sets of input values. The simulator code itself was not examined directly for these cases, partly because we had confidence in the code generation from examining the smaller cases, and partly because the code generated for the larger cases was very difficult to inspect directly (Figure 7). The output values for a number of representative cases were validated manually. However, exhaustive tests (within the integer domains selected earlier) were carried out by comparison against the results of "brute force" computations derived from the original PLA descriptions. No discrepancies were detected, although the compiler carried out some optimizations which we had not anticipated.

This integer-supporting state machine compiler is now fully integrated into the GRTL register-transfer design tool, so that it will be subjected to continual testing by computer engineering students at the Luleå Institute of Technology in northern Sweden.

5 OTDD Bit Ordering

A specimen of behavioral simulation code emitted by GPLA is shown in Figure 7. On the one hand, we would like to minimize the size of this code file, since this reduces the compilation time for the code. Unfortunately we were finding that the most compact code was not always the fastest at run-time. Run-time speed depends in part on the order in which the various comparisons are applied to the input vector. The earliest comparisons should in some sense have the most impact. We found this concept difficult to quantify, since it also involves the entropies of the individual input signals and these are in general not known to GPLA. All of these factors must be considered when choosing the ordering of the bits used in the OTDDs as discussed in Section 3.

Therefore our approach was empirical. We reasoned that the signal having the most impact would be the "current state" signal, so we placed those OTDD

```
grtl_pla starting:
filename prefix:                    b12
parsing completed:          0.09 sec
first pass completed:       0.07 sec
number of otdd bits:          15
second pass completed:     15.73 sec
otdd pass completed:        0.34 sec
compression completed:      0.00 sec 2 cases
code text generated:        7.02 sec
writing code file:                  b12.c
code file written:          0.26 sec
writing symbstate file:             b12.pla3
symbstate file written:     6.70 sec
memory nodes used:      44483
grtl_pla successful:       30.24 sec total
```

Figure 8: Execution statistics on a Sun 4 ELC SPARCstation for the compiling of the description derived from case "b12" of the LGSynth93 MCNC benchmarks, for output bit "o_6". In this case the OTDD computations to parse and combine input cubes account for most of the time spent in the second pass, leaving little work to do in the OTDD pass. Since this description contains no integer signals, GPLA then emits the PLA as disjoint cubes using symbolic state names into the file "b12.pla3".

bits first; then came bits for other integer signals; finally came the bits for binary signals. We then compared this against an ordering in which the bits for binary signals came first, then the bits for non-state integers, then the bits for the input state. Surprisingly the second ordering seemed to produce more efficient code, even though the text files were somewhat larger. At the moment we are satisfied with this result, but we expect to repeat the experiments when we have gathered together a benchmark set containing larger descriptions.

6 Conclusion

We have reported the successful implementation of a state machine language, supporting both Boolean and integer input and output signals. It detects erroneous and ambiguous input descriptions, generates behavioral simulation code, and emits non-compromised output for the logic optimizer. Simulator code behaves correctly in the presence of "unknown" input values, because the implementation is based on Ordered Ternary Decision Diagrams which explicitly treat the "unknown" case. We are still working on reducing the size of the emitted simulation code (figure 7) by using a more direct emission of the OTDD. The current implementation is now fully integrated with the GRTL design environment [Jen91a] where its fast execution times are especially important.

References

[BC87] M. C. Browne and E. M. Clarke. SML - a High Level Language for the Design and Verification of Finite State Machines. In Dominique Barrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 269–292. Elsevier Science Publishers B.V. (North-Holland), 1987.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Jen91a] G. Jennings. GRTL – a Graphical Platform for Pipelined System Design. In *Proceedings of the 1991 European Conference on Design Automation (EDAC 1991)*, pages 424–428, February 1991.

[Jen91b] G. Jennings. Reversible Functional Simulation for Digital System Design. In *Proceedings of the IEEE 1991 Custom Integrated Circuits Conference (CICC '91)*, pages 8.2.1–8.2.4, May 1991.

[L+88] M. Loughzail et al. Experience with the VHDL Environment. In *Proceedings of the 25th Design Automation Conference*, pages 28–33, June 1988.