A Simple C to Verilog Compilation Procedure for Hardware/Software Verification

Jiang Long and Robert Brayton
Berkeley Verification and Synthesis Research Center (BVSRC)
Department of EECS
University of California at Berkeley
{jlong, brayton}@eecs.berkeley.edu

ABSTRACT

The objective of this work is two-fold: (1) to build a simple trusted translator from C programs to a hardware description language (in this case Verilog) and (2) to illustrate its application to the formal verification of hardware and software systems using highly developed hardware model checking methods. To achieve the first goal, we used the LLVM compiler infrastructure to compile the C program into LLVM bytecode, and used a simple and straightforward method to translate this into a Verilog circuit. The current implementation is able to handle arbitrary loops and static array access. In the experimental section, equivalence checking was done comparing *OpenCores* VHDL hardware models of FPUs against C-coded *SoftFloat* versions. This revealed several bugs in the *OpenCores* versions. For model checking safety properties, we experimented with bit-vector benchmarks from the 2015 Software Verification Competition and checked SVA assertions derived from *VERIFIER_error* calls in the C program. This also revealed discrepancies.

1. INTRODUCTION

There has been an increasing interest in reasoning about and proving properties of C software programs. In the hardware domain, there is an increasing interest in using C as a hardware specification language. While the actual hardware is typically described in Verilog and synthesized from that, the C model represents the golden model which is used to compare behaviors. In both cases, a highly trusted translation from C to Verilog is desired.

In this work, we focus on creating a translation tool that is as simple as possible using trusted and well established intermediate tools to accomplish the task. We eschew optimization which can add complications and therefore increased potential for introducing bugs. This allows existing hardware formal verification flows such as equivalence and property checking of safety and liveness properties to be applied readily, with high confidence that the translation process has not introduced bugs and therefore the results can be trusted. We use the LLVM compiler infrastructure and a straight-forward translation of its produced LLVM bytecode to create a Verilog program from a single thread C program. At this point, an equivalence-checking Verilog (miter) model can be created using the actual hardware model. Figure 1 shows the high-level C-to-Verilog translation flow and how it fits into a C-to-RTL equivalence checking methodology using hardware verification. Such a flow can also be used in the software verification of C programs where assertions are translated into SVA and compiled into logic using, for example *Verific*[5].

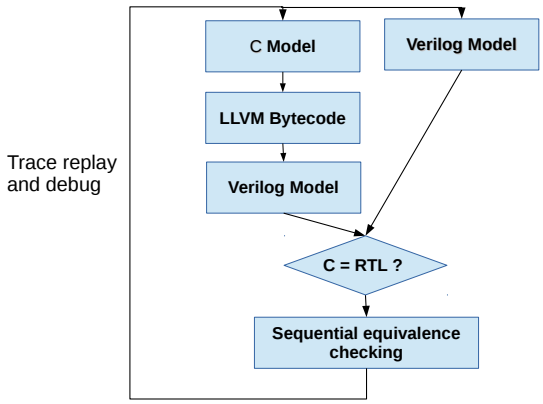

Figure 1: C vs RTL equivalence checking

1.1 Motivation

A language front-end to compile C programs into finite-state-machines can be a complex and delicate matter. There has always been concerns that the compilation procedure may incorrectly reproduce the original C program semantics. This is a real concern as illustrated in the C program in Figure 2. It is one of the test cases in the Bit-Vector category of the 2015 Software Verification Competition [8]. In this category, only integer arithmetic and control flow structures in the C language are allowed.

This test is officially classified as "unreach", i.e. the assertion is considered proven *true* by the competition committee. It is one of the initial regression tests imposed on any entry in the competition. However, the assertion fails when $n = 65536$, in which case, $n * (n + 1)$ overflows while the addition inside the loop does not. Four, [20][25][28][27], out

```
extern unsigned int __VERIFIER_nondet_uint();
int main() {
  unsigned int i, n=__VERIFIER_nondet_uint(), sn=0;
  for(i=0; i<=n; i++) {
    sn = sn + i;
  }
  __VERIFIER_assert(sn==(n*(n+1))/2 || sn == 0);
}
```

Figure 2: sum02_true-unreach-call.c

Solvers	Rank	Consistency	#Errors
ESBMC[25]	1st	Buggy	1
CBMC[20]	2nd	Buggy	1
CPAChecker[10]	3rd	Buggy	1
Beagle 1.1[27]		Buggy	1
Cascade 2.0[28]		Buggy	1
Seahorn[16]		Buggy	> 1
Ultimate Automizer [18]		Buggy	> 1
Ultimate Kojak[15]		Buggy	> 1

Table 1: 2015 Software Verification Competition: Bit-Vector category

of eight of the competition participants produced the incorrect *true* result (Participants ([20][25] were recognized as the top two winners in this category). The other four participants, [16] [15][18][10], had one or more incorrect results on other test cases. Thus none of participants was one hundred percent correct.

It is not known where the bugs got introduced into the various tool flows, but for any such tool, the very first step is to compile the C program into an internal representation of the C execution semantics. To many, this is a complex and time-consuming, yet preliminary step.

We believe simplicity results in fewer bugs. In this work, we focus on a procedure to translate C to a circuit model that is simple and less error-prone by construction. Our simple/trusted C-to-Verilog compilation should take the translation procedure out of the verification equation, and reliably bridge the gap between software and hardware. This allows existing hardware synthesis and verification techniques to be applied reliably to the software domain.

1.2 Why Verilog

We chose Verilog as the intermediate representation for the circuit model for the following reasons:

- It is almost the de facto standard for hardware description from which hardware is synthesized.
- Verilog's *always_ff* and *always_comb* blocks can be used to express a finite state machine model as sequential and combination logic respectively.
- Verilog is defined through IEEE standards. The well-defined syntax and semantics make it easier to map the C semantics into Verilog constructs.
- Synthesis and verification tool flows for Verilog can be utilized readily after the translation.
- Validation of the translation procedure can be conducted by simulating the Verilog model and comparing against the C-code program using random input vectors. In our case, we use Verilator[26] to compile the generated Verilog code back to C++.
- Verilog has enough constructs to capture the original C constructs, which is important to retain the control flow and word-level operators after the translation.

In Verilog, computation resources must be statically resolvable at synthesis time. Thus, Verilog does not support non-constant loop controls and infinite loops at the language level. In contrast, C programs support more flexible control flows with non-constant loop control variables and dynamic resource management such as memory allocation/deallocation, run-time function call-stacks and run-time parallel threads. In this work, we limit the scope of the control flow structure of C to single-thread C programs with arbitrary control flows but statically resolvable resources.

1.3 Contributions

1. We document a simple procedure, based on LLVM, by which a subset of C (dynamic memory management, recursive functions and parallelism are excluded), can be mapped into a Verilog module using the *always_ff* and *always_comb* blocks.

2. We provide experiments for C-to-RTL equivalence checking and software property checking, showing that such a flow allows existing hardware verification tool chains to be used, producing promising results.

1.4 Organization

In Section 2, we review the circuit computation model, the language characteristics of C, the LLVM[21] Bytecode, and Verilog. In Section 3, the overall tool flow and translation procedure is described. Experimental results are presented in Section 4 followed by discussions of related work in Section 5 and conclusions in Section 6.

2. BACKGROUND

2.1 Logic network and the Circuit

A logic network N is a directed graph (V, E) with node set V and edge set E. Each node is labeled with a node type and a width. A node type is one of {input, output, operator, flip-flop}. An input node has zero incoming edges; an output has zero outgoing edges, and a flip-flop has a single incoming edge. An operator node is an arithmetic function of its immediate fanin nodes.

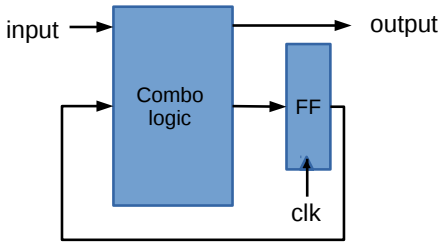

Figure 3: A single-clock synchronous circuit

Such a logic network can express the functionality of a synchronous sequential circuit shown in Figure 3, which contains inputs, outputs, combinational logic, flip-flops and a single clock input.

It has the following characteristics:

- The logic network topology is statically determined and does not change during evaluation of the network.
- The circuit evaluates at every clock cycle and infinitely over time.

The goal is to map a C program into a logic network such that an evaluation of it produces the same input/output behavior as the original C program.

2.2 Verilog HDL

Verilog is an almost de facto standard used by industry for specifying hardware designs. Table 2 summarizes its language constructs. The *always_ff* block is used to describe sequential logic where flip-flops are inferred, while *always_comb* is for combinational logic. According to Verilog's event-driven semantics, for the circuit in Figure 3, the *always* blocks are evaluated every clock cycle, and the computation is infinite over time.

Language elements	Verilog Constructs
variables	wire /reg
control flow	if/else/case
sequential block	always_ff @(posedge clk)
combo block	always_comb
operators	$+, -, *, /, \&, \|, etc.$

Table 2: Verilog language elements

2.3 The C Programming Language

The C language has two major features. One is the control flow structure that defines the order of computation; the other is resource allocation, which can be dynamic at run-time. A C program can be viewed as a circuit-like computation model with dynamic resource allocation. Conceptually, resources refer to those computation elements that either hold the value of a variable, or implement some arithmetic function. Static resources are allocated at compile time, while dynamic resources can be acquired and released during run-time. Table 3 lists the major C features and their category, static or dynamic.

Language Elements	C Constructs	Resource
variable	local/global	static
control flow	if/case/for/while	static
memory	malloc/free	dynamic
parallelism	pthread	dynamic
call stack	function calls	dynamic
pointers	pointer arithmetic	aliasing

Table 3: C language elements

Pointers are simply aliases for objects in the C program. Although they can be complex to reason about, there is no resource allocation involved in their use, and thus not a fundamental barrier in translating to a circuit model. On the other hand, run-time function calls dynamically allocate resources for the stack space to hold local variables. Nested non-recursive function calls can be eliminated at compile time by function inlining. Dynamic threads also require run-time acquisition of local variable space and operator nodes. Although the behavior of loops can vary at run time, the computational resources for variable space can be statically determined during compilation. Therefore if the resources required for a C program to execute can be statically determined, a corresponding circuit model for it exists.

In Section 3, we demonstrate a method for mapping a C program into a circuit model if the program does not have any of the dynamic features, namely, dynamic memories, pthreads and recursive functions. Mapping into Verilog then becomes almost verbatim.

2.4 LLVM Bytecode

C-to-Verilog is done in two steps, the first using the LLVM compiler infrastructure to create a LLVM bytecode representation. The second is an almost verbatim translation from this to Verilog. The LLVM bytecode has a static single assignment (SSA) [14] form, in which each variable is assigned exactly once and is defined before it is used. In SSA form, *use-def* chains are explicit and each contains a single element. A *use* is a location where a variable value is referenced; while a *def* is the location where a variable value is assigned. A *use-def* chain is used for data-flow analysis. A basic block is defined as a maximum sequence of instructions that has no branching or return statement, except the last one. Such a SSA program is viewed as a control flow graph (CFG) with basic-blocks as nodes, while edges between basic-blocks are derived from branching/return instructions.

```
int factorial(int n ) {
   int ret = 1;
   for ( int i = 1;i<=n;i++)
       ret = ret * n;
   return ret;
}
```

Figure 4: C function

Figure 5 is the control flow graph derived from compiling the C function in Figure 4. The top block labeled *entry* is the starting point of the function execution. The for loop in Figure 4 is converted to a loop of three basic blocks in Figure 5, labeled: $for.cond$, $for.body$, and $for.inc$. All loop constructs, $for, while, do-while$, are rewritten using branching instructions for uniform treatment. Such a LLVM bytecode program has the following characteristics:

- Every symbol in the bytecode instructions is of an LLVM *Value* type, referred to as an SSA variable in the rest of the paper.
- Each basic-block is uniquely identified by an LLVM *Label*, which is also an LLVM *Value* type.

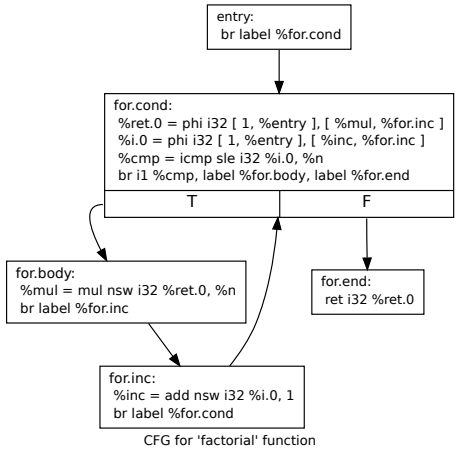

Figure 5: LLVM CFG

- The last instruction for each BasicBlock is either a *BranchInst* or a *ReturnInst* instruction.
- An LLVM *PhiNode* structure is used to handle the values that are obtained from different incoming branches.
- Non-recursive function calls can be eliminated through inlining.

In the SSA form used in the LLVM bytecode, there is no distinction between global and local variables. All variables have a unique name. Variable types are reduced to basic signed/unsigned integers and floating point numbers (in this work, we currently don't consider floating point arithmetic). Each basic block has a unique label. The order of computation is captured by the CFG (control-flow-graph). Only uninitialized variables can be used before being defined. Without dynamic memory allocation, all pointer access can be eliminated through a pre-processing step to map pointer references to array indexing. Illegal memory access through pointers becomes illegal array indexing. After the preprocessing step, the LLVM bytecode is a pure SSA form without side-effects caused by pointers.

Such a pure SSA form is very close to a circuit logic network. In particular, resources are allocated for each SSA variable through the variable's unique naming. To convert to a logic network model, one needs to ensure that the Verilog execution has the same sequence of LLVM instructions with the same *use-def* chain ordering as the original CFG.

3. TRANSLATING LLVM BYTECODE TO VERILOG

The treatment for translating LLVM arithmetic instructions into the corresponding Verilog operators is mostly a syntactic rewrite, so it is skipped. The following sections focus on two steps in the translation procedure to Verilog:

1. Variable declaration: Create Verilog variables for each SSA variable.
2. Create *always* blocks to capture the computation sequence of the LLVM CFG.

3.1 Loop Free Bytecode

For loop free Bytecode, the CFG is acyclic. An evaluation of the function annotates the CFG graph with values at each node. This is the same as evaluating each node in topological order from the entry block. Such a CFG can be viewed as a logic network, which maps directly into a Verilog module consisting of a single always_comb block.

Figure 6 shows the skeleton of such a Verilog module, which implements the function $y = f(x)$. For each SSA variable, a Verilog bit-vector is declared, and for each LLVM Label, a single bit is declared. For example, the 32-bit integer $\%ret.0$ and label $for.cond$ in Figure 5 are mapped to the following Verilog:

```
reg [31:0]  v_ret_0;
reg  label_for_cond ;
```

During initialization, SSA variables are initialized as specified, and all label variables are set to 0 except the very first (entry) basic block's label, which is set to 1, marking the starting point of the evaluation. Label variables are set to 1 by branch instructions, indicating that the corresponding basic-block has been entered. Uninitialized variables are left as dangling wires in the Verilog result. A *PhiNode* is translated into a case statement to compute the values. Therefore, a topological traversal of the bytecode CFG creates the $always\_comb$ block that implements a function $y = f(x)$.

```
module cmodel_f( x, y) ;
input x;
output y;
// Variable declarations
reg ... // label vars
reg ... // ssa vars

always_comb begin
// Initialization section
...
// SSA statement section
... //topological order of the CFG
end
endmodule
```

Figure 6: Loop-free translation to a Verilog *always_comb* block

3.2 Bytecode with Loops

For a CFG with loops, the *use-def* chain will have loops. To preserve the original *use-def* chain semantics, we use the sequential logic template as shown in Figure 7. At each clock cycle, exactly one basic-block is evaluated. In addition to input/output ports for x and y, it has *clock*, *reset* and *start* signals, and output *ready*. The function is evaluated over time: the value of x is captured when *start* transitions from 0 to 1 while the output value y is captured when *ready* transitions from 0 to 1. Conceptually the block computes $y@ready = f(x@start)$. The *reset* signal clears the state of the block and brings it into an *idle* state. The number of cycles from *start* to *ready* is variable depending on the functionality of f over the input value x.

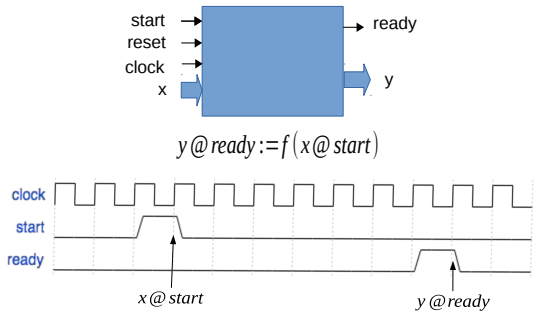

Figure 7: Verilog Model

During the variable-declaration phase of Verilog, a single bit (referred as the *label*-signal) is declared for each LLVM label while two bit-vectors of the same width are allocated for each SSA variable. One of the bit-vectors, referred as the *latched*-signal, is assigned exactly once in the *always_ff* block, and the other, referred as the *temp*-signal, is assigned exactly once in the *always_comb* block.

At each clock cycle, the Verilog module is configured to evaluate exactly one basic block. In such a setting, the *temp*-signal is used in a *use-def* chain within the same basic block to capture the immediate result of an instruction, while the *latched*-signal is used for a *use-def* chain between different basic blocks. For each basic block, the Verilog module creates an *always_comb* block, and the assignment for each instruction is translated into a Verilog blocking-assignment to the corresponding *temp*-signal.

There is only one *always_ff* block in a Verilog module. Each LLVM basic block, is followed by an 'if' block, created to manage the assignments for *label*_signals and *latched*_signals from the *temp*_signals.

```
if (v_label) begin
 v_label <= 1'b0;
 v_latched <= v_temp;
 ...
 v_next_label <= ... ;
end
```

The above *v_next_label* is the *label*_signal referenced in the *branch* (br) statement at the end of the basic block, indicating which basic-block to evaluate in the next clock cycle. For illustration, Figure 8 shows the Verilog code for the basic block *for.cond* in Figure 5 (The % in the LLVM symbol names is removed in the Verilog code).

A *PhiNode* is treated differently. Flip-flops are introduced for *PhiNodes* and assigned in each of the incoming basic blocks prior to entering the current basic block. The choice of using a *latched*-signal or *temp*-signal is determined through *use-def* analysis within each basic block. Within each basic block, a use-before-def value uses the *latched*-signal, while def-after-use values use the *temp*-signals. A Verilog blocking-assignment evaluates the left-hand-side of the assignment immediately which corresponds to the semantics of an SSA variable value defined earlier in the same basic block.

From Verilog synthesis semantics and coding style in the

```
always_comb begin
  temp_cmp =
     $signed(latched_i) < $signed( latched_n) ;
end
always_ff @(posedge clk) begin
  ...
  if( label_for_cond ) begin
    label_for_cond <= 1'b0;
    latched_cmp <= temp_cmp;
    label_for_body <= temp_cmp;
    label_for_end <= ! temp_cmp;
  end
  ...
end
```

Figure 8: Translated Verilog for basic-block *for.cond*

generated Verilog module, variables assigned in the *always_ff* produce flip-flops, while those in the *always_comb* are part of the combinational logic of the circuit. The above translation may create unnecessary *temp*-signals and *latched*-signals because the corresponding SSA variable may not be referenced outside of the basic block where it is assigned. However, those unused signals are usually deleted by a subsequent Verilog tool flow. This allows the translation procedure at this stage to be greatly simplified.

Finally, we initialize the label for the entry basic block to 1 after *start* is set to 1 and set *ready* to 1 in the 'if' block where the corresponding basic-block has a return statement. Because of the SSA form of LLVM bytecode, the above procedure creates a Verilog module that captures the original function $y = f(x)$ through $y@ready$ and $x@start$.

3.3 Assertions

In the above framework, a C assertion $assert(a)$ is translated into SVA[6] with the following form:

```
assert property (@(posedge clk) label_t|-> a);
```

A termination condition of the C program is a liveness check on the Verilog module:

```
assert property (@(posedge clk)
        start|-> eventually ready);
```

Memory errors in C that result in signal SIGSEGV will cause the C program to stop. In the Verilog model, such an error will result in an un-defined state in the circuit. Extra logic or assertions can be added to detect such an error condition when it happens. For error-free C programs, the above translation captures the exact semantics of the C program.

3.4 Creating a Verilog Miter for Equivalence Checking

It is useful to point out that in creating a miter between a Verilog hardware description that will be used for synthesis and the Verilog produced by our procedure as the golden model for comparison, a miter compares outputs at each

clock cycle. The two models need not be cycle comparable so it is necessary to make the miter output not flag an error when the *ready* is not asserted. To make this happen, extra logic is needed in the miter to make the comparison only when *ready* is asserted each time.

4. EXPERIMENTS

The following experiments were conducted on a 32-core Intel Xeon 2.6GHz machine running Ubuntu Linux. Timeout was set to 900 seconds for the formal engine *suprove*(*super_prove*) from ABC[1], which was the winner of the HWMMC'14, single-output safety category [4]; *suprove* is a parallel proof engine that uses multiple algorithms.

4.1 FPU Verification : C vs. RTL equivalence checking

In this experiment, we conducted FPU verification of two OpenCores FPU designs: *fpu_100* and *fpu_double*. Both were designed in Verilog and were intended to implement and conform to the IEEE 754 standard: *fpu_100* is a 32-bit and *fpu_double* a 64-bit floating point unit. A floating point number is represented using 3 components starting from the left of a 32- or 64- bit-vector (i.e. MSB): a single-bit sign bit, an exponent, and a mantissa. In such a number system, two special numbers, *sNaN* and *qNaN*, are used to represent non-real numbers (not-a-number) - useful for handling exceptions. For both *sNaN* and *qNaN*, the exponent needs be all 1's and the mantissa is non-zero.

Both designs support add, subtract, multiply, divide, and square-root operations. Conforming to the IEEE standards, all operators raise exception flags if the computation underflows, overflows, or produces inexact results. Both designs are configured to run in the *round_to_nearest_even* mode and are pipe-lined implementations. They have the same start-ready model as in Figure 7, where an input is driven when *start* is asserted, and the result of the FPU operation is available when *ready* is asserted, a number of cycles later after *start*.

Both comparisons are against the C model from the SoftFloat[17] C library, which is the de facto standard for floating point implementations. After the C translation to a Verilog model, a miter model is constructed between the generated Verilog model and the OpenCore model design, comparing the outputs when both *ready* signals are asserted. The run-time for each equivalence checking problem was set to timeout at 900 seconds.

No.	Opcode	Inputs	Softfloat Result	RTL Result	Pipeline Depth
1	add32	matched			6
2	mul32	000017f0 * 43360000	001104a0	00008825	10
3	div32	4fdbc9bf / ff800000	80000000	ff800000	33
4	sqrt32	0000f7c0	1e321421	1dfbd75a	33

Table 4: fpu_100 : 32-bit FPU

The Verilog models for opcodes *add32* and *mul32* are combinational, while they are sequential for *div32* and *sqrt32*. Table 4 shows the results of comparing the translated SoftFloat program with *fpu_100*. It turns out that only for *add32* does *fpu_100* match with SoftFloat, while the three other opcodes all have mismatches as shown in the table.

These equivalence checking problems were done under the constraint that no exception flag is generated in the Verilog models, because in the IEEE standard, when an exception is raised, the result is not defined in certain situations. The input numbers and mismatching output results are captured in Table 4. Column six is the pipeline depth for the corresponding operation in the *fpu_100* implementation.

We also compared *fpu_double* with SoftFloat's 64-bit FPU routines. In addition to comparing the results of the operation, we also conducted equivalence checking of the exception generation logic. Table 5 shows the experiments for opcodes *add*, *subtract* and *multiply*. All three models generated from the C functions are combinational. Tests 1 and 3 produce different output values with no exception raised, while tests 2, 4, and 5 produce the same output value for the *add*, *subtract* or *multiply* operations, but the SoftFloat library functions generate underflow or inexact exceptions while *fpu_double* does not.

Both *fpu_double* and *fpu_100* were completed several years ago and reportedly have been incorporated into silicon and FPGAs. From the *OpenCores* repository, each benchmark includes a random test framework which compares it to *SoftFloat* for validation. Each have passed millions of random test vectors. However, from the counterexamples, we observe that the input numbers that cause the mismatches either contain 10+ consecutive 0s or 1s (e.g. the *NaN* numbers), or the values of both numbers are close to each other. These situations are very unlikely to be generated by a random number generator and thus a random test bench would most likely miss those scenarios; sequential equivalence checking is much more effective in exposing discrepancies. As far as we know, this is the first time these bugs have been reported for *OpenCores* designs.

4.2 Software Verification of Safety Properties

We conducted C verification on the bit-vector benchmarks of the 2015 Software Verification Competition[8].

Although all participants have bugs, we chose to compare against *CPAChecker* because it is an unbounded solver while the other two winners are bounded model checkers. We thought its result is more comparable although it does produce an incorrect TRUE result on the function in Figure 9.

In the C program under verification, the dummy function __VERIFIER_nondet_int used to indicate a random number generator, was converted into new free primary inputs in the Verilog modules. The calls to the error flagging function, *__VERIFIER_error*, was converted into SVA assertions. VeriABC[22], which interfaces with Verific[5] was then used to compile the generated Verilog and SVA into an AIG[11]. *suprove* from ABC [1] was used to prove or dis-prove the target properties.

No.	Opcode	Inputs	Softfloat Result	RTL Result	Pipeline Depth
1	add64	fff8000000000000 + 7ff8000000000000	7ff8000000000000	fff8000000000000	20
2	add64	d172dd2000000000 + 4175c97000000000	inexact	no exception	20
3	sub64	fdf88d1ffffe7ba4 - fff88d20001e45dc	fff88d20001e45dc	7ff88d20001e45dc	21
4	sub64	d130600000400000 - b131a0020000a41f	inexact	no exception	21
5	mul64	8000006e00008194 * 8000001e9e8d5048	inexact , underflow	no exception	24

Table 5: fpu_double: 64-bit FPU

```
int main(void) {
  unsigned int x = 10;
  while (x >= 10) {
    x += 2;
  }
  __VERIFIER_assert(x % 2);
}
```

Figure 9: test:bitvector-loops/overflow_false-unreach-call1.i

Figure 10 shows the results. CPAChecker's results were from the official competition website, which were run on a 3.4Ghz Intel Quad-core i7 platform, while ours were run on an 32-core Intel Xeon 2.6Ghz host. The second and third columns show ABC's *suprove* results, while the forth and fifth columns are the published results from the competition's website. Of the 46 tests, ABC resolved 30 while *CPAchecker* resolved 40. There are 4 tests that ABC resolved but *CPAChecker* did not. The capital letters, TRUE and FALSE, are indicating the test is solved by only one of the two solvers being compared. The results seem rather promising in that *suprove* is only optimized to run on hardware model checking problems that have been bit-blasted without utilizing any software related heuristics or word-level information.

So far, we believe our proof results are more trust-worthy because the C-compilation procedure is simple and less error-prone and no inconsistency of the results have been found yet.

Test	suprove	Time	CPA	Time
byte_add_1_true-unreach-call	timeout	900	TRUE	34.56
byte_add_2_true-unreach-call	timeout	900	TRUE	78.33
byte_add_false-unreach-call	false	7.70	false	63.58
diamond_false-unreach-call2	false	0.48	false	1.78
gcd_1_true-unreach-call	true	2.14	true	2.29
gcd_2_true-unreach-call	true	13.82	true	2.31
gcd_3_true-unreach-call	true	10.12	true	2.26
gcd_4_true-unreach-call	true	0.12	true	1.38
implicitunsignedconversion_false-u	false	0.10	false	1.59
implicitunsignedconversion_true-un	true	0.09	true	1.40
integerpromotion_false-unreach-cal	false	0.11	false	2.30
integerpromotion_true-unreach-call	true	0.09	true	1.57
interleave_bits_true-unreach-call	true	0.66	true	16.63
jain_1_true-unreach-call	true	0.10	true	2.52
jain_2_true-unreach-call	true	0.10	true	2.61
jain_4_true-unreach-call	true	0.11	true	2.59
jain_5_true-unreach-call	TRUE	0.10	timeout	930.61
jain_6_true-unreach-call	true	0.12	true	2.65
jain_7_true-unreach-call	true	0.12	true	4.97
modulus_true-unreach-call	TRUE	35.57	timeout	903.46
num_conversion_1_true-unreach-ca	true	0.09	true	1.40
num_conversion_2_true-unreach-ca	true	0.90	true	17.07
overflow_false-unreach-call1	timeout	900	TRUE	122.57
parity_true-unreach-call	timeout	900	timeout	906.63
s3_clnt_1_false-unreach-call.BV	false	99.51	false	5.75
s3_clnt_1_true-unreach-call.BV	timeout	900	TRUE	32.85
s3_clnt_2_false-unreach-call.BV	timeout	900	FALSE	87.28
s3_clnt_2_true-unreach-call.BV	timeout	900	TRUE	29.56
s3_clnt_3_false-unreach-call.BV	false	16.77	false	4.48
s3_clnt_3_true-unreach-call.BV	timeout	900	TRUE	34.83
s3_srvr_1_alt_true-unreach-call	TRUE	518.21	aborted	128.31
s3_srvr_1_true-unreach-call.BV	timeout	900	TRUE	63.91
s3_srvr_2_alt_true-unreach-call	timeout	900	TRUE	62.97
s3_srvr_2_true-unreach-call.BV	timeout	900	TRUE	62.91
s3_srvr_3_alt_true-unreach-call	timeout	900	TRUE	63.48
s3_srvr_3_true-unreach-call.BV	timeout	900	TRUE	63.86
signextension2_false-unreach-call	false	0.08	false	2.12
signextension2_true-unreach-call	true	0.09	true	1.54
signextension_false-unreach-call	false	0.12	false	2.18
signextension_true-unreach-call	true	0.09	true	1.56
soft_float_1_true-unreach-call	timeout	900	TRUE	11.35
soft_float_2_true-unreach-call	true	119.53	true	12.70
soft_float_3_true-unreach-call	TRUE	300.46	timeout	930.46
soft_float_4_true-unreach-call	timeout	900	TRUE	63.56
soft_float_5_true-unreach-call	true	138.73	true	13.25
sum02_true-unreach-call	timeout	900	timeout	903.20

Figure 10: Software Verification Benchmark: bitvector category

5. RELATED WORKS

Because we are building a finite state machine model directly from a C program, it is more suitable to compare our work with those software verification flows that do symbolic exploration of the state space using model checking algorithms.

The SLAM model checker [7] introduced Boolean programs – imperative programs where each variable is Boolean – as an intermediate language to represent program abstractions. A tool flow was created to convert C programs and predicates to Boolean programs and then to employ follow-up model checking using abstraction/refinement.

The Blast [9] model checker implements an optimization of CEGAR (counter example guided abstraction refinement) called lazy abstraction. The internal model of the C program is built incrementally, based on an error trace and an unrolling of the CFG. A similar internal representation is used in the IMPACT [23] model checker.

The C verifiers f-soft [19] and CPAChecker [10] internally build a finite state machine model using their own internal representations. The finite state machine definition is conceptually the same as the circuit model. Our approach is simpler because Verilog has well defined syntax and semantics. LLBMC [24] also uses the LLVM infrastructure to translate the C program for bounded model checking through loop unrolling.

In the hardware design area, high level synthesis tools Catapult [2] and Forte [3] build circuits from a subset of the C language. Their primary focus is to optimize the generated hardware to satisfy user-defined timing, power, and area constraints. In contrast, our goal is for verification purposes; the translation procedure does not consider memories and assumes an unlimited number of flip-flops to use in the Verilog model; optimization is not the concern at the translation stage because abstraction and optimization can be conducted later, on the Verilog model instead of on the original C program.

The tools AutoPilot [29] and LegUp [12] translate a C program for FPGA synthesis or hardware and software co-simulation. They use the LLVM framework and translate LLVM bytecode into Verilog. There is no detailed description of the underlying translation algorithm. By looking at the generated Verilog from LegUp, their principle seems to be similar to ours, except they use memory components to allocate memory spaces for each variable and introduce memory access latency for each read/write of variable values while the control flow graph is instruction based rather than basic block based.

The tool in [13] processes a C program into an SSA-like internal format and conducts bounded model checking for C to RTL equivalence checking. Our model can be used for both bounded and unbounded model checking and our implementation is much simpler based on the LLVM infrastructure.

6. CONCLUSIONS

In this paper, we observed that the gap between a C program and a circuit computational model is due to using dynamic resources in the C language: run-time memory allocation/de-allocation, run-time function call stacks, and parallel threads. We build a simple translation procedure for a single thread non-recursive C program to a semantically equivalent Bytecode network using the LLVM compiler. The mapping from LLVM bytecode to a Verilog module is then almost verbatim. Experiments show promising and more consistent results compared to existing software verification approaches.

7. REFERENCES

[1] ABC - a system for sequential synthesis and verification. Berkeley Verification and Synthesis Research Center, http://www.bvsrc.org.

[2] Calypto® Catapult Design Product. http://www.calypto.com.

[3] Forte design systems. http://www.forteds.com.

[4] Hardware model checking competition 2014.

[5] Verific Design Automation: http://www.verific.com.

[6] *1800: IEEE Standard for System Verilog - Unified Hardware Design, Specification, and Verification Language*. IEEE Computer Society, 2005.

[7] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

[8] D. Beyer. Software verification and verifiable witnesses. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–416. Springer, 2015.

[9] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[10] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification*, pages 184–190. Springer, 2011.

[11] A. Biere, K. Heljanko, and S. Wieringa. Aiger 1.9 and beyond. *Available at http://fmv.jku.at/hwmcc12/beyond1.pdf*, 2012.

[12] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[13] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Design Automation Conference, 2003. Proceedings*, pages 368–371. IEEE, 2003.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[15] E. Ermis, A. Nutz, D. Dietsch, J. Hoenicke, and A. Podelski. Ultimate kojak. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 421–423. Springer, 2014.

[16] A. Gurfinkel, T. Kahsai, and J. A. Navas. Seahorn: A framework for verifying c programs (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–450. Springer, 2015.

[17] J. Hauser. Softfloat. *available from http://www.jhauser.us/arithmetic/SoftFloat.html*, 2002.

[18] M. Heizmann, J. Christ, D. Dietsch, J. Hoenicke, M. Lindenmann, B. Musa, C. Schilling, S. Wissert, and A. Podelski. Ultimate automizer with unsatisfiable cores. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 418–420. Springer, 2014.

[19] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform. In *Computer Aided Verification*, pages 301–306. Springer, 2005.

[20] D. Kroening and M. Tautschnig. Cbmc–c bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[21] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[22] J. Long, S. Ray, B. Sterin, A. Mishchenko, and R. Brayton. Enhancing abc for ltl stabilization verification of systemverilog/vhdl models. *DIFTS*, 2011.

[23] K. L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, pages 123–136. Springer, 2006.

[24] F. Merz, S. Falke, and C. Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *Verified Software: Theories, Tools, Experiments*, pages 146–161. Springer, 2012.

[25] J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, and B. Fischer. Esbmc 1.22. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 405–407. Springer, 2014.

[26] W. Snyder, P. Wasson, and D. Galbi. Verilator: Convert verilog code to c++/systemc, 2012.

[27] D. Wang, C. Zhang, G. Chen, Y. Peng, F. He, M. Gu, and J. Sun. Beagle. 2015.

[28] W. Wang, C. Barrett, and T. Wies. Cascade 2.0. In *Verification, Model Checking, and Abstract Interpretation*, pages 142–160. Springer, 2014.

[29] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. Autopilot: A platform-based esl synthesis system. In *High-Level Synthesis*, pages 99–112. Springer, 2008.