v2c – A Verilog to C Translator[⋆]

Rajdeep Mukherjee[1], Michael Tautschnig[2], and Daniel Kroening[1]

[1] University of Oxford, UK
[2] Queen Mary, University of London
{rajdeep.mukherjee,kroening}@cs.ox.ac.uk, mt@eecs.qmul.ac.uk

Abstract. We present *v2c*, a tool for translating Verilog to C. The tool accepts synthesizable Verilog as input and generates a word-level C program as an output, which we call the *software netlist*. The generated program is cycle-accurate and bit precise. The translation is based on the synthesis semantics of Verilog. There are several use cases for *v2c*, ranging from hardware property verification, co-verification to simulation and equivalence checking. This paper gives details of the translation and demonstrates the utility of the tool.

1 Introduction

At the bit level, formal property verification for hardware is scalable to circuits up to the block level but runs out of capacity for SoC-level or full-chip designs. Verification at the word level promises more efficient reasoning, and thus better scalability. However, unlike the AIGER format that is used to represent bit-level netlists, there is no standard format to represent circuits at the word level. In this paper, we argue that hardware circuits given in Verilog can be represented at the word level by encoding them as C programs, which we call a *software netlist*. To this end, we present a Verilog to C translator which we name *v2c*. Given a Verilog RTL design, *v2c* applies the synthesis semantics to automatically generate an equivalent C program. The tool is available online at http://www.cprover.org/hardware/v2c/.

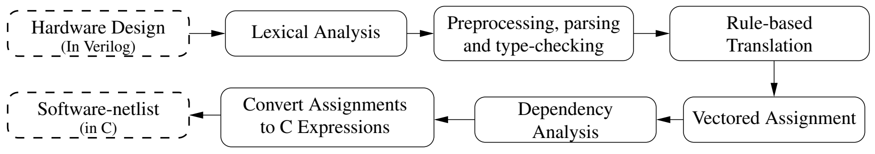
The primary motivation for the transition from bit level to word level is to gain scalability [5, 6]. The exploitation of high-level structures for better reasoning is a standard goal in hardware verification. We propose to take one further step: the automatic translation of hardware circuits to a software netlist model in C allows us to leverage advanced software verification techniques such as abstract interpretation and loop acceleration, which have never been applied in conventional bit-level hardware verification.

Verilog and C share many common operators. However, Verilog offers a number of additional operators like part-select, bit-select from vectors, concatenation and reduction operators, which are not available in C. Additionally, Verilog statements like the initial block, the always block, the generate statement, procedural assignment (blocking, non-blocking) and continuous assignment are not supported in C. Further, Verilog offers 4-valued data-types. These non-trivial constructs, combined with parallelism, make the translation of Verilog to C challenging.

[⋆] Supported by ERC project 280053 and the SRC task 2269.001.

Although SoC designs are increasingly written at a higher level of abstraction [3,4], there is still a significant body of existing design IP blocks that are written in VHDL or Verilog. Our tool *v2c* allows rapid generation of software netlist models from hardware IPs given in Verilog RTL. Other tools like VTOC [2] or Verilator [3] also generate C/C++ code; however, VTOC was not obtainable; the code generated by Verilator is suitable for simulation only and is not amenable to formal analysis.

2 v2c – The Verilog RTL to C Translator



Hardware Design (In Verilog)
Lexical Analysis
Preprocessing, parsing and type-checking
Rule-based Translation
Vectored Assignment
Dependency Analysis
Convert Assignments to C Expressions
Software-netlist (in C)

Fig. 1. Translation stages in *v2c*

Figure 1 illustrates the translation steps of *v2c*. The front-end phase performs macro preprocessing, parses Verilog and checks the types. The front-end supports the 1364-2005 IEEE Standard for Verilog HDL. It generates a type-annotated parse-tree, which is passed to the translation phase. During the translation phase, the tool applies the synthesis semantics and performs a rule-based translation following the Verilog module hierarchy. The rule-based translation produces vectored assignments by mapping bit-operations to equivalent shift and mask operations and performs a global dependency analysis to determine inter-module and intra-modular dependencies. The translation phase is followed by the code-generation phase, where the intermediate vectored expressions and translated module items are converted into C expressions. Note that we refrain from any optimizations or abstractions to obtain a correct and trustworthy output.

Software Netlist: A *software netlist SN* is a four-tuple $\langle L, A, l_0, l_e \rangle$, where L is the finite set of locations for modeling the program counter in the corresponding sequential code, $l_0 \in L$ is the initial location, $l_e \in L$ is the error location and $A \subseteq L \times M \times L$ is the control flow automation. The edges in A are labelled with a quantifier-free first-order formula M over program variables, which encode an assignment or an assume statement. The formula M is defined by five-tuples $\langle In, Out, Seq, Comb, Asgn \rangle$, where *In*, *Out*, *Seq*, *Comb* are *input*, *output*, *sequential/state-holding* and *combinational/stateless* signals, respectively. *Asgn* is a finite set of assignments to *Out*, *Seq* and *Comb* where

– $Asgn ::= CAsgn | SAsgn$
– $CAsgn ::= (V_c = bvExpr) | (V_c = bool)$, $V_c \in Comb \uplus Out$

[3] http://www.veripool.org/wiki/verilator

– $SAsgn ::= (V_s = bvExpr)|(V_s = bool),\, V_s \in Seq$
– $bvExpr ::= bv_{const}|bv_{var}|ITE(cond, bv_1 \ldots bv_n)|bv_{op}(bv_1 \ldots bv_n), cond \in bool, bv_i \in \{bv_{const}, bv_{var}\}$
– $bool ::= true|false|\neg b|b_1 \wedge b_2|b_1 \vee b_2|bv_{rel}\{b_1 \ldots b_n\}$

2.1 Translating Verilog Module Items

Data Model: The data model in Verilog is significantly different from C. Each bit of a C integer value can have only two states, namely 0 and 1. Bits in Verilog HDL can take one of four values, namely 0, 1, *X* and *Z*. A value of 0 represents low voltage and value of 1 represents high voltage. Further, the values *X* and *Z* represent an unknown logic state and a high impedance value, respectively. The simplest synthesis semantics for *X* is treating it as a "don't-care" assignment, which allows the synthesis to choose a 0 or 1 to further improve logic minimization. *v2c* treats *X* and *Z* values to be non-deterministic.

Registers, wires, parameters and constants: Verilog supports structural data types called *nets*, which are *wire* and *reg*. The value of wire variables changes continuously as the input value changes. By contrast, the *reg* types hold their values until another value is assigned to them. A structure containing all state holding elements of a module is declared in C to store the register variables. Wires are declared as local variables in C. Verilog parameters are constants, which are frequently used to specify the width of variables. Parameters are declared as constants in C. Verilog also allows the definition of translation unit constants using the `` `define `` construct, e.g., `` `define STATE 2'b00; ``, which is the same as the #define preprocessor directives in C.

Variable declaration: Variables of specific bit-width (register, wire) in Verilog are translated to the next largest native data type in C such as `char`, `short int`, `long`, `long long`, etc.

Always and Initial blocks: Always blocks are the concurrent statements, which execute when a variable in the sensitivity list changes. The statements enclosed inside the always block within *begin* . . . *end* constructs are executed in parallel or sequentially depending on whether it is a non-blocking or blocking statement, respectively. The behaviour of an initial block is the same as that of an always block, except that they are executed exactly once, before the execution of any always block. Figure 2 and Figure 3 demonstrate the translation of Verilog always blocks. All code snippets are partial due to space limitations.

Module Hierarchy with Input/Output Port: The communication between modules takes place through *ports*. Ports can be input only, output only and inout. Figure 2 gives an example of a Verilog module hierarchy on the left and the translated code block in C on the right. The output ports are passed as reference to reflect the changes in the parent module. The generated C code preserves the module hierarchy of the RTL. Structurally identical code often aids debugging, as identifying corresponding C/RTL operations is easier.

Module Hierarchy with Input/Output Ports

```
module top(in1, in2);
input [3:0] in1, in2;
wire [3:0] o1, o2;
and A1 (in1, in2, o1, o2);
and A2 (.c(o1),.d(o2),.a(o1),.b(in2));
endmodule
// Module Definition
module and1(a, b, c, d);
input [3:0] a, b;
output [3:0] c, d;
reg [3:0] c;
always @(*) begin
 c = a & b;
end
assign d = 1;
endmodule
```

Data and Function definitions in C

```
struct state_elements_and {
 unsigned char c; };
struct state_elements_and sand;
void and(unsigned char a, unsigned char b,
unsigned char *c, unsigned char *d) {
 *d = 1; sand.c = a & b;
}
void top(unsigned char in1, unsigned char in2) {
 unsigned char o1,o2;
 and(in1, in2, &o1, &o2);
 and(o1, in2, &o1, &o2);
}
void main() {
 unsigned char in1,in2;
 top(in1, in2);
}
```

Fig. 2. Handling Module hierarchy with I/O ports

Procedural Assignments: Procedural assignments are used within Verilog always and initial blocks and are of two types: *blocking* and *non-blocking*. Blocking assignments are executed in sequential order. However, the effect of blocking assignments is visible immediately, whereas the effect of non-blocking assignments is delayed until all events triggered are processed. This form of parallelism in procedural assignments are modeled in *v2c* by first storing the value of register variables in auxiliary variables in the beginning of the clock cycle. Each read access to the register variables are then replaced by these auxiliary variables. This ensures that an assignment to a register variable do not influence subsequent procedural assignments. Figure 3 illustrates the translation of procedural assignments (given at the top) to the equivalent C semantics (given at the bottom).

Non-blocking assignment

```
reg [7:0] x,y,z;
wire in = 1'b1;
always @(posedge clk) begin
 x <= in;
 y <= x;
 z <= y;
end
```

```
struct smain {
unsigned char x,y,z; } sm;
unsigned char xs,ys,zs;
 _Bool in = 1;
// save register variables
 xs=sm.x;ys=sm.y;zs=sm.z;
// update register variables
 sm.x = in;
 sm.y = xs;
 sm.z = ys;
```

Blocking assignment

```
reg [7:0] x,y,z;
wire in = 1'b1;
always @(posedge clk) begin
 x = in;
 y = x;
 z = y;
end
```

```
struct smain {
unsigned char x,y,z;}sm;
 _Bool in = 1;
// clocked block
 sm.x = in;
 sm.y = sm.x;
 sm.z = sm.y;
```

Continuous assignment

```
wire in;
reg a,b,t;
wire a = in;
wire c = b; wire d = c;
always @(posedge clk) begin
 b <= a;
 t <= b;
end
```

```
struct smain {
_Bool a,b,t; } sm;
_Bool in,c,d,as,bs,cs,ds,ts;
sm.a = in;//continuous assign
// save register variables
as=sm.a;bs=sm.b;ts=sm.t;
// clocked block
sm.b = as; sm.t = bs;
// continuous assignment
c = sm.b; d = c;
```

Fig. 3. Tanslation of non-blocking, blocking and continuous assignments

Continuous Assignment: The continuous assignment is used to assign a value to a wire. Continuous assignments are concurrent statements, which are immediately triggered when there is any change in any of the signals used on the right-hand side. The translation of continuous assignments are discussed next.

2.2 Dependency Analysis

v2c performs intra-modular dependency analysis to correctly model the dependencies between the combinatorial and sequential blocks. Let us consider the following three cases for dependency analysis which are demonstrated with an example in Figure 3.

1. A variable, say x, appearing in continuous assignment, say A, is updated directly by the input signal and the same variable is read inside an always block. The continuous assignment A is placed before the always block to capture any change to the input signal and subsequently propagate the updated value of x to the translated always block.
2. A variable, say x, assigned in a continuous assignment statement, say A, appears in the right-hand side of another continuous assignment statement, say B. In this case, the variable assignment A is placed before the other assignment B which reads x.
3. A variable, say x, appearing in the right-hand side of a continuous assignment, say A, is driven by an always block. This gives an ordering where the continuous assignment is placed after the always block to capture the updated value of x.

For designs with inter-modular combinatorial paths or combinatorial loops, the combinatorial signals (wire variables) may settle after several executions before the next clock cycle. The combinatorial exchanges between modules depends on the stability condition for the combinatorial signals and thus it is necessary to execute the combinatorial logic until the stability condition is reached. Determining such stability condition for large circuits is hard. An alternative way to handle combinatorial exchanges between modules is by using assumptions over the signals that encode combinatorial logic in the respective modules following synthesis semantics. An example using the latter approach is given at http://www.cprover.org/hardware/v2c/.

Bit-precise code generation: *v2c* generates a bit-precise software netlist model in C. The tool automatically handles complex bit-level operators in Verilog like bit-select or part-select operators from a vector, concatenation operators, reduction OR and other operators. *v2c* retains the word-level structure of the Verilog RTL and generates vectored expressions. Figure 4 shows Verilog code (at the top) and the generated C expressions (at the bottom), which are combinations of bit-wise and arithmetic operators like bit-wise OR, AND, multiplication, subtraction, shifts and other C operators.

3 Equivalence of Hardware and software Netlist

We have applied *v2c* to a range of Verilog RTL circuits, which were obtained from different sources. We have observed that the translation produces the correct output.

Bit-select	Part-select (SystemVerilog)	Concatenation

wire [7:0] in1,in2;
reg [7:0] out1,out2;
out1[7:5] = in1[4:2];
out2[6] = in2[4];
	reg [31:0] in, out;
	for(i=0;i<=3;i++) **begin**
	out[8*i +: 8]=in[8*i +: 8];
	end
		wire [7:0] in1, in2;
		reg [9:0] out;
		out = {in2[5:2],in1[6:1]};

unsigned char in1,in2;
struct smain {
 unsigned char out1,out2; } sm;
sm.out1 = sm.out1 & 0x1f |
(((in1 & 0x1c)>>2)<<5);
sm.out2 = (sm.out2 & 0xbf)|
(((in2 & 0x10)>>4)<<6);
	struct smain {
	 unsigned int in,out; } sm;
	for(i=0;i<=3;i++) {
	 x=8*i+(8−1); y=8*i;
	 sm.out=(sm.out&!(2^31−2^y))
	 |(sm.in&(2^31−2^y)); }
		unsigned char in1,in2;
		struct smain {
		 unsigned char out; } sm;
		sm.out = (((in2 >> 2)
		 & 0xF) << 6)|
		 ((in1 >> 1) & 0x3F);

Fig. 4. Handling Bit-select, part-select from vectors and concatenation operator

While we do not have a formal proof of correctness, experiments have shown that for property verification, valid safety properties are proven to be *k*-inductive for the same value of *k* in the hardware and software netlist models. Conversely, for unsafe designs, the bug is found in the same cycle for both the models.

4 Implementation

We have implemented *v2c* in C++ on top of the *CPROVER* framework [1]. We make a pre-compiled static-binary for Linux available at http://www.cprover.org/hardware/v2c/. We also provide several benchmarks in Verilog and the corresponding software netlist models in C, which can be used for simulation, property verification or equivalence checking. Currently, *v2c* does not support multi-clock designs, transparent latches and designs with combinatorial loops.

5 Conclusion and Future Work

This paper presents a tool for translating Verilog RTL to C. The generated software netlist can be used as word-level representation for hardware circuits in Verilog RTL. This design representation allows us to leverage advanced software verification techniques for hardware verification. In the future, we plan to handle combinatorial feedback between modules and also support a richer subset of SystemVerilog assertions for property specification.

References

1. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004)
2. Greaves, D.J.: A Verilog to C compiler. In: RSP. pp. 122–127. IEEE Computer Society (2000)
3. Keating, M.: The Simple Art of SoC Design. Springer (2011)
4. Liu, L., Vasudevan, S.: Scaling input stimulus generation through hybrid static and dynamic analysis of RTL. ACM TODAES 20(1), 4:1–4:33 (2014)
5. Mukherjee, R., Kroening, D., Melham, T.: Hardware verification using software analyzers. In: ISVLSI (2015)
6. Mukherjee, R., Schrammel, P., Kroening, D., Melham, T.: Unbounded safety verification for hardware using software analyzers. In: DATE (2016)