# Using Bitvector Program Generators For Digital Circuit Synthesis

Quentin Roussel
*School of Computing*
*National University of Singapore*
e1374335@u.nus.edu

***Abstract*—Efficient hardware synthesis from high-level logical specifications is a critical challenge in modern digital system design. This work focuses on generating non-trivial combinational logic blocks and Moore machines from logical constraints, leveraging an off-the-shelf loop-free bitvector program generator. The proposed methodology transforms time-dependent specifications into time-independent representations by encoding temporal dependencies, such as references to past inputs and outputs, into the machine state. The state is implemented as a shift register that retains only the necessary history required by the specification.**

**Two key combinational blocks are synthesized: one for state transition logic, computing the next state based on the current state and inputs, and another for output computation, generating outputs based solely on the current state. For specifications referencing past outputs, we propose an adjusted architecture where the output computation is integrated with the state update to capture the required dependencies. The framework ensures modularity, enabling efficient synthesis of Moore machines directly implementable in Hardware Description Languages (HDLs) like SystemVerilog.**

***Index Terms*—Hardware Description Language, Moore Machines, Hardware Synthesis, Temporal Logic, SystemVerilog**

## I. Introduction

In modern digital systems, hardware design often requires precise implementation of complex combinational and state-dependent logic. Hardware Description Languages (HDLs) such as SystemVerilog, Verilog, and VHDL are commonly used to describe and implement such systems. However, manually crafting efficient HDL code for intricate logic or systems involving complex logical constraints can be time-consuming and error-prone.

Combinational logic forms the backbone of digital systems, where outputs depend solely on current inputs and predefined logical relationships. For non-trivial combinational logic, such as computing arithmetic expressions, signal transformations, or custom operations, defining accurate and efficient HDL code is particularly challenging. Tools like program synthesis have shown promise in automating the generation of such logic from high-level specifications.

In this work, we aim to leverage an off-the-shelf loop-free bitvector program generator, such as **Brahma** [2], to synthesize non-trivial combinational blocks directly from user-defined logical specifications. **Brahma** [2] is well-suited for generating compact, correct-by-construction programs that map bitvector inputs to an output based on logical constraints. However, Brahma operates under certain limitations: it does not inherently handle time-dependent behavior.

To extend Brahma's applicability to HDL synthesis, we introduce a methodology for pre- and post-processing user specifications:

- **Pre-processing:** Temporal dependencies, such as references to past inputs and outputs, are encoded into a finite state captured by Moore machine structures. This transforms the problem into a combinational one by treating the current state as an extended input.
- **Post-processing:** The output from Brahma is integrated into an HDL framework, where combinational logic synthesized by Brahma operates between the pre-computed state and output.

While this approach is inspired by temporal logic frameworks like Linear Temporal Logic (LTL), we focus on a practical subset of specifications that reference a finite number of past inputs and outputs. By encoding temporal history into state variables, we convert stateful and temporal problems into purely combinational ones, allowing Brahma to synthesize non-trivial combinational logic blocks efficiently.

The contributions of this work are as follows:

1) A methodology for leveraging loop-free bitvector program synthesis tools to generate non-trivial combinational blocks from logical specifications.
2) A framework for pre-processing temporal logic specifications, encoding temporal history into state variables for Moore machine-like structures.
3) Demonstrations of the approach through examples such as rolling averages and maximum value tracking, showcasing its ability to handle practical hardware design problems.

The remainder of this paper is structured as follows. Section II provides background on HDLs, combinational logic, Brahma, and related work. Section III formalizes the problem and introduces motivating examples. Section IV describes the methodology, including state encoding and combinational block generation. Section V presents results and evaluates the approach. Finally, Sections VI and VII discuss limitations, future work, and conclude the paper.

## II. Background and related work

The synthesis of hardware designs from logical specifications has been extensively studied, with significant contributions spanning various methodologies, including finite state machine (FSM) synthesis, logic-based synthesis, and modern program synthesis frameworks.

### A. FSM Synthesis and Moore Machines

Klimowicz et al. [3] proposed methodologies for synthesizing finite state machines by integrating Moore and Mealy machine models. Their approach emphasized reducing the number of states while preserving timing constraints, resulting in efficient designs for FPGAs and CPLDs. These methods form a foundational basis for FSM-based synthesis, particularly Moore machines, which rely solely on the state for outputs, ensuring stability and predictability in hardware implementations.

### B. Logical Specifications for Reactive Systems

Bloem et al. [1] introduced techniques for synthesizing reactive systems using generalized reactivity (GR(1)) logic, a fragment of linear temporal logic (LTL). Their work focused on synthesizing modular designs by separating assumptions about the environment from guarantees about the system. This structured approach highlights the utility of formal logic in creating predictable and modular systems. However, such logic can often lead to overly complex synthesis processes for certain applications.

### C. Loop-Free Program Synthesis

The Brahma framework, introduced by Gulwani et al. [2], focuses on synthesizing loop-free bitvector programs by reducing the problem to constraint solving. This approach leverages SMT solvers to generate efficient combinational circuits and arithmetic designs, particularly useful for handling modular libraries of operations. Brahma's ability to synthesize non-trivial arithmetic and bitwise logic designs from specifications closely aligns with our focus on synthesizing arithmetic-constrained Moore machines.

### D. HDL Code Generation and Machine Learning

Recent advancements in HDL code generation have incorporated machine learning techniques to automate complex design processes. Sun et al. [5] explored classification-based methods to guide HDL generation, leveraging large language models (LLMs) for combinational and sequential logic tasks. These methods showcase the potential of AI-assisted design tools in simplifying hardware synthesis while maintaining correctness.

### E. Bridging LTL and Hardware Design

The translation of LTL specifications into hardware has been extensively studied. Piterman et al. [4] emphasized symbolic representation and automata theory to synthesize hardware from high-level temporal logic. These methods focus on ensuring the correctness and efficiency of state-based designs but often require handling complex GR(1)-style specifications.

### F. Contributions of This Work

This work builds on these foundations by focusing on logical constraints that are less complex than GR(1) logic but include more intricate arithmetic relationships. The key contributions of this work are:

- Simplifying temporal dependencies by encoding past inputs and outputs as state variables in shift registers.
- Leveraging off-the-shelf bitvector program synthesis tools to generate combinational blocks that compute state transitions and outputs.
- Producing hardware designs with readable HDL code, emphasizing clarity and maintainability.
- Bridging high-level logical constraints and low-level HDL synthesis while enabling arithmetic-constrained Moore machine synthesis.

By targeting specifications that blend arithmetic complexity with manageable temporal dependencies, this work offers a novel approach to synthesizing stateful digital systems while maintaining simplicity and readability in the resulting code.

## III. Problem definition

### A. Program synthesis problem spesification

The goal of this work is to synthesize non-trivial combinational logic blocks from high-level logical specifications that may include dependencies on past inputs or outputs. To formalize this problem, we begin by defining the notation for the inputs, outputs, and internal state of the system.

The system's **inputs** are represented as a time series of $k$-bit vectors, denoted as $\vec{I}(t) = \{I_0(t), I_1(t), \ldots, I_{k-1}(t)\}$, where $I_i(t)$ is the value of the $i$th input signal at discrete time $t$. The **outputs** are similarly defined as $k$-bit vectors, denoted as $O(t)$, computed at each time step based on the current inputs and potentially previous inputs and outputs.

The program synthesis problem is defined by a logical constraint $\phi_{spec}$, the inputs $\vec{I}$, the output $O$, and a library of functions $\mathcal{L}$. Formally, the specification $\phi_{spec}$ establishes the relationship between the inputs, outputs, and possibly their past values. It is expressed as:

$$\phi_{spec}(\vec{I}(t \leq T), O(t \leq T), O(T+1)),$$

where $t \leq T$ indicates that the specification may depend on inputs and outputs up to time $T$. The constraint $\phi_{spec}$ can include references to a finite number of past input values $\vec{I}(t-n)$ and output values $O(t-n)$ for $n > 0$, introducing temporal dependencies that need to be addressed.

In this formulation, the function library $\mathcal{L}$ is a set of specifications representing all the bitvector operations permitted by the target language, in this case, SystemVerilog. This library defines the operations available for constructing the desired combinational logic, such as arithmetic, bitwise, and comparison operations. The full library used can be found in Table I.

TABLE I
SYSTEMVERILOG OPERATIONS AND THEIR LOGICAL SPECIFICATIONS.

| Operation | SystemVerilog Syntax | Logical Specification |
|---|---|---|
| Bitwise AND | `O = I_1 & I_2;` | $O[i] = I_1[i] \wedge I_2[i]$ |
| Bitwise OR | `O = I_1 \| I_2;` | $O[i] = I_1[i] \vee I_2[i]$ |
| Bitwise XOR | `O = I_1 ^ I_2;` | $O[i] = I_1[i] \oplus I_2[i]$ |
| Bitwise NOT | `O = ~I_1;` | $O[i] = \neg I_1[i]$ |
| Reduction AND | `O = &I_1;` | $O = I_1[0] \wedge I_1[1] \wedge \cdots \wedge I_1[N-1]$ |
| Reduction OR | `O = \|I_1;` | $O = I_1[0] \vee I_1[1] \vee \cdots \vee I_1[N-1]$ |
| Reduction XOR | `O = ^I_1;` | $O = I_1[0] \oplus I_1[1] \oplus \cdots \oplus I_1[N-1]$ |
| Shift Left | `O = I_1 << N;` | $O[i] = I_1[i-N]$ if $i \geq N$, 0 otherwise |
| Shift Right | `O = I_1 >> N;` | $O[i] = I_1[i+N]$ if $i+N <$ size$(I_1)$, 0 otherwise |
| Arithmetic Addition | `O = I_1 + I_2;` | $O = I_1 + I_2$ |
| Arithmetic Subtraction | `O = I_1 - I_2;` | $O = I_1 - I_2$ |
| Arithmetic Multiplication | `O = I_1 * I_2;` | $O = I_1 \times I_2$ |
| Arithmetic Division | `O = I_1 / I_2;` | $O = I_1 \div I_2$ |
| Arithmetic Modulo | `O = I_1 % I_2;` | $O = I_1 \mod I_2$ |
| Less Than | `O = (I_1 < I_2);` | $O = \begin{cases} 1, & \text{if } I_1 < I_2 \\ 0, & \text{otherwise} \end{cases}$ |
| Greater Than | `O = (I_1 > I_2);` | $O = \begin{cases} 1, & \text{if } I_1 > I_2 \\ 0, & \text{otherwise} \end{cases}$ |
| Switch (If-Then-Else) | `O = (I_3) ? I_1 : I_2;` | $O = \begin{cases} I_1, & \text{if } I_3 \neq 0 \\ I_2, & \text{else} \end{cases}$ |

### B. Output Format

To simplify the problem, we focus on a specific form of modules: the implementation of Moore machines. A Moore machine is a finite state machine where the outputs depend solely on the current state, not on the inputs. This structure ensures predictable and stable outputs, making it well-suited for hardware synthesis.

The program we propose will generate two distinct combinational blocks: First, a **state transition block**, which computes the next state of the machine $\vec{S}(T+1)$ based on the current state $\vec{S}(T)$ and inputs $\vec{I}(T)$. This part will be generated algorithmically as described in IV. Secondly, an **output computation block**, which computes the outputs of the machine based solely on the current state, this block will be generated using a bitvector program synthesiser.

The general structure of the program is illustrated in Fig. 1

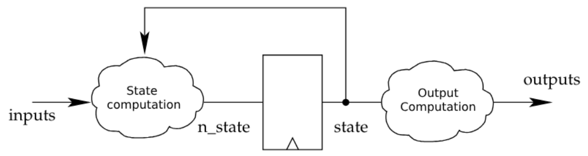

Fig. 1. Architecture of the program

## IV. METHODOLOGY

### A. New State Computation

The key idea is to eliminate time dependency by storing past inputs directly in the machine state. This allows the state to act as a shift register that retains only the necessary amount of past values for each input or output referenced in the specification. To compute the exact number of past values to keep we parse the specification $\phi_{spec}$ and for each intpus and for the output, find the reference to the oldest sample. Formally, let $\tau_i = \max\{t \in \mathbb{N} | I_i(T-t) \text{ appears in } \phi_{spec}\}$ . In the simpler case where the specification contains references only to past inputs, the new state can be computed using Algorithm 1.

**Algorithm 1** New state computation for storing past inputs

```
1: Parse $\phi_{spec}$ and computing the $\tau$ values
2: for $i = 0$ to $k-1$ do
3:     if $\tau_i > 0$ then
4:         $S_{i,0} \leftarrow I_i$
5:         for $j = 1$ to $\tau_i - 1$ do
6:             $S_{i,j} \leftarrow S_{i,j-1}$
7:         end for
8:     end if
9: end for
```

However, if the specification includes references to past outputs, those outputs must also be incorporated into the state. This adjustment requires moving the output computation block alongside the state computation block so that the outputs can be added to the shift register. As for the intups we note $\tau_o = \max\{t \in \mathbb{N} | O(T-t) \text{ appears in } \phi_{spec}\}$ The updated architecture reflecting this adjustment is shown in Fig. 2. The algorithm for this more complex scenario is presented in Algorithm 2. This algorithm references the new output computation based on the current state, this commbinational block is the one generated as described in IV-B.

### B. Output Computation

The output computation determines how the outputs of the system are derived from the state. The process involves three

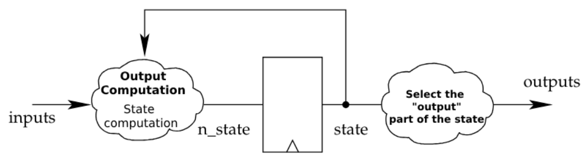

Fig. 2. Modified architecture of the program to store past outputs

**Algorithm 2** New state computation

```
1: Compute $\tau_o$
2: run Algorithm 1
3: if $\tau_o > 0$ then
4:     for $j = 1$ to $\tau_o - 1$ do
5:         $S_{k+1,j} \leftarrow S_{k+1,j-1}$
6:     end for
7: end if
```

main steps: preprocessing the logical specification, solving the synthesis constraint, and postprocessing the resulting program.

## *C. Preprocessing*

The way **Brahma** works is by taking a program specification defined by:

- a **specification** $\left\langle \vec{I}, O, \phi_{spec}(\vec{I}, O) \right\rangle$ where $\vec{I}$ is the set of input variables, $O$ is the set of output variables, and $\phi_{spec}(\vec{I}, O)$ is a logical formula describing the relation between the output variables and the input variables.
- a **library** $\{\left\langle \vec{I_i}, O_i, \phi_i(\vec{I_i}, O_i) \right\rangle | 1 \leq i \leq n\}$ representing the available instructions the program can use.

The function library used will be the one defined in Table I, but the inputs, outputs, and constraints must first be adapted. To achieve this, a new program specification is created based on the problem specification introduced in Section III. The new logical specification $\hat{\phi}_{spec}$ is create by replacing any references to past inputs or outputs in $\phi_{spec}$ with generic variables. For instance, a reference to $O(T-2)$ in $\phi_{spec}$ is replaced with the variable $S_{k+1,2}$.

Formally, this Brahma-compatible specification can be written as :

$$\hat{\vec{I}} := \vec{S} = \bigcup_{0 \leq i < k} \bigcup_{1 \leq j < \tau_i} S_{i,j}$$

$$\hat{O} := O$$

$$\hat{\phi}_{spec} := \phi_{spec} \text{ where } \forall 0 \leq i < k, \forall 1 \leq j < \tau_i, I_i(T-j) \text{ is replaced by } S_{i,j}. \quad (1)$$

*a) Solving the Synthesis Constraint:* Once preprocessed, the new specification is passed to the bitvector program generator brahma. This tool uses SMT solvers to synthesize a combinational logic program that satisfies the logical constraints of $\hat{\phi}_{spec}$. The resulting program is correct by construction and minimizes unnecessary operations.

*b) Postprocessing:* The output of the synthesis tool is then translated into SystemVerilog code. Each variable in the synthesized program is mapped to corresponding shift registers or state variables. The resulting code is placed inside an `always_comb` block in the SystemVerilog template, ensuring that the output is updated dynamically based on the current state.

For example, given the specification:

$$O = (I_1(t-1) \wedge I_1(t-2)) - 1$$

The synthesized logic may produce:

```
always_comb begin
    v1 = (I1_reg[1] & I1_reg[2]);
    O = v1 - 1;
end
```

Then this block is added to the rest of the module to produce the final code.

## V. Experimental Results

The generator has been tested on programs with simple specifications due to time and computation power constraints. In this section, we present three example cases to demonstrate the functionality and modularity of the generated SystemVerilog code.

### *A. Case 1: Simple Logic Specification*

For the specification:

$$\phi_{spec} := O(t) = (I_1(t-1) \wedge I_2(t-2)) - 1$$

the generated code is structured into two distinct parts: the shift register block and the combinational computation block. Below is the complete generated SystemVerilog code:

```
module GeneratedModule (
    input logic clk,
    input logic [31:0] I1,
    input logic [31:0] I2,
    output logic [31:0] O
);

    logic [31:0] I1_reg [1:1];
    always_ff @(posedge clk) begin
        for (int i = 1; i > 1; i = i -
         ↪  1) begin
            I1_reg[i] <= I1_reg[i-1];
        end
        I1_reg[1] <= I1;
    end

    logic [31:0] I2_reg [1:2];
    always_ff @(posedge clk) begin
        for (int i = 2; i > 1; i = i -
         ↪  1) begin
            I2_reg[i] <= I2_reg[i-1];
        end
        I2_reg[1] <= I2;
    end

    always_comb begin
        v1 = (I2_reg[1] & I1_reg[2]);
        O = v1 - 1;
    end

endmodule
```

This example demonstrates how the generator splits the specification into shift registers to handle temporal dependencies and a combinational block for logic computation.

## B. Case 2: Averager with Past Inputs

For a 4-value rolling average, with the specification:

$$\phi_{spec} := O(t) = \frac{1}{4}\left(I_1(t-1) + I_1(t-2) + I_1(t-3) + I_1(t-4)\right)$$

the generator produces the following combinational block:

```
always_comb begin
    v1 = I1_reg[1] + I1_reg[2];
    v2 = I1_reg[3] + I1_reg[4];
    v3 = (v1 + v2) >> 2;
    O = v3
end
```

The use of bit-shift for division ensures efficient implementation while maintaining accuracy. This demonstrates how the generator handles arithmetic operations across multiple past inputs.

## C. Case 3: Minimum Value with Past Outputs

For a design that references past outputs, such as a minimum tracker:

$$\phi_{spec} := O(t) = \min(O(t-1), I_1(t))$$

the generator includes a new shift register for past outputs, as shown below:

```
    logic [31:0] O_reg [1:1];
    always_ff @(posedge clk) begin
        for (int i = 1; i > 1; i = i -
         ↪  1) begin
            O_reg[i] <= O_reg[i-1];
        end
        O_reg[1] <= I1;
    end
```

The corresponding combinational block is:

```
always_comb begin
    v1 = (O_reg[1] < I1)
    v2 = v1 ? O_reg[1] : I1;
    O = v2 ;
end
```

This case demonstrates the generator's capability to handle designs with dependencies on past outputs by integrating additional state variables.

The examples illustrate that the generator is capable of producing modular and functional SystemVerilog code for a variety of use cases, including arithmetic computations, logical operations, and state tracking. However, the complexity of the combinational blocks is currently limited by the capabilities of the underlying bitvector program generator (**Brahma**). Despite these limitations, the generated designs are efficient, readable, and well-suited for practical hardware synthesis tasks.

## VI. Limitations and Future Work

While the proposed methodology provides a robust framework for synthesizing combinational logic and Moore machines from logical specifications, several opportunities for enhancement remain. This section explores potential directions for future work.

### A. Expanding Temporal Logic Support

The current approach is limited to handling specifications with finite temporal dependencies, where references to past inputs and outputs are explicitly bounded. To broaden the applicability of the methodology, future work could focus on extending support to more expressive temporal logics, such as generalized reactivity (GR(1)) logic. GR(1) specifications allow for nested and complex temporal relationships, enabling the synthesis of systems with more intricate state-based behaviors. Incorporating such logic would involve refining the preprocessing step to efficiently encode nested temporal dependencies into the machine state.

### B. Support for Multiple Outputs

The framework currently targets designs with a single output. However, many practical applications require generating modules with multiple outputs. Supporting multiple outputs would involve optimizing the synthesis process to identify and share common computations among outputs. This enhancement could reduce the overall hardware resource utilization and improve performance. Additionally, the SystemVerilog templates could be extended to accommodate multiple combinational blocks, one for each output, while preserving modularity and readability.

### C. Automated Specification Translation

An important goal for future work is to improve the accessibility of the synthesis process by enabling automated translation of high-level specifications into the required logical form. This could involve developing user-friendly interfaces where designers specify desired behaviors in natural language or high-level temporal logic, which are then automatically translated into the input format for the synthesis tool. Such automation would reduce the barrier to entry for non-experts and streamline the design process.

### D. Integration with Formal Verification

While the synthesized designs are correct-by-construction with respect to the input specification, integrating formal verification tools would provide an additional layer of assurance. Future work could explore the automatic generation of formal properties, derived from the original specification, for verification against the synthesized HDL code. This would ensure that the resulting designs not only meet functional requirements but also adhere to safety, liveness, and other critical correctness properties.

### E. Resource and Timing Constraints

To make the framework more practical for hardware implementations, future work could introduce the ability to limit hardware resources, such as the maximum number of Look-Up Tables (LUTs), or enforce timing constraints, such as the maximum critical path length. This could be achieved by incorporating resource and timing estimates into the synthesis process and adapting the optimization algorithm to prioritize designs that meet these constraints. Such enhancements would allow designers to balance performance and resource usage more effectively, particularly for applications targeting resource-constrained environments like FPGAs.

## VII. Conclusion

This paper presented a methodology for synthesizing non-trivial combinational logic and Moore machines directly from logical specifications. By using an off-the-shelf loop-free bitvector program generator, we demonstrated how time-dependent specifications can be transformed into purely combinational forms through efficient state encoding. The proposed approach introduces a modular framework where temporal dependencies, such as past inputs and outputs, are encoded in shift registers, enabling seamless integration with existing synthesis tools.

The resulting SystemVerilog code effectively separate state management from output computation, ensuring both clarity of the generated hardware descriptions.

While the current framework is limited to finite temporal dependencies and manageable specifications, it lays the groundwork for future exploration of more complex temporal logic synthesis. Enhancements such as GR(1)-style specification handling, optimization of state encoding, and integration with automated verification tools represent promising directions for future work.

In conclusion, this work offers a practical solution for synthesizing stateful digital systems that satisfy a given logical constrain. By automating the translation of logical specifications into hardware description languages, this methodology simplifies the design process, reduces human error, and accelerates the development of robust and efficient digital systems.

## References

[1] Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., and Sa'ar, Y. Synthesis of reactive(1) designs. In *Journal of Computer and System Sciences* (2012), vol. 78, Elsevier, pp. 911–938.

[2] Gulwani, S., and Srivastava, S. Synthesis of loop-free programs. *ACM SIGPLAN Notices 44*, 1 (2009), 62–73.

[3] Klimowicz, D., Jemielniak, A., and Dobrowolski, D. Finite state machine synthesis for fpga and cpld. In *2012 IEEE International Symposium on Industrial Electronics* (2012), pp. 1282–1287.

[4] Piterman, N., Pnueli, A., and Sa'ar, Y. Synthesis of reactive(1) designs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)* (2006), vol. 3855 of *Lecture Notes in Computer Science*, Springer, pp. 364–380.

[5] Sun, Y., Yang, J., Chen, L., Chen, F., and Yu, Z. Guided hdl code generation with classification-assisted large language models. In *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2023), pp. 1–8.