PREPRINT: "R. Sinha and H. Patel, "synASM: A High-level Synthesis Framework that Supports Explicit Parallel and Timed Constructs," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 31, no. 10, pp. 1508-1521, Oct. 2012."

synASM: A High-level Synthesis Framework with Support For Parallel and Timed Constructs

Rohit Sinha *Student Member, IEEE,* and Hiren D. Patel *Member, IEEE*

***Abstract*—This work presents a high-level synthesis framework called synASM that synthesizes Abstract State Machines (ASMs) to VHDL for FPGAs. In particular, this work focuses on the specification, scheduling, and synthesis of parallel and timed constructs. ASMs possess well-defined formal semantics for sequential, and parallel computation, and their composition. We extend ASMs to support the specification of timing requirements, which we call timed constructs. We also describe the composition of timed constructs with sequential and parallel computation. A key contribution of this work is the extension of the force-directed scheduling algorithm to support both parallel and timed constructs. We implement the synthesis back-end in synASM that targets FPGAs. Our experiments show improvements of up to 52% in LUT usage and 34% in total area for certain examples.**

***Index Terms*—High-level synthesis, parallel and timed constructs, force-directed scheduling.**

I. Introduction

HIGH-LEVEL synthesis (HLS) addresses the challenge of generating hardware designs from algorithmic specifications. The popular choice of language used for algorithmic specification is either C or some C-like variant. The reason for this choice is a pragmatic one: most designers can program using C, which means they do not need to learn a new language to use a HLS methodology. This allows designers with a wide spectrum of expertise to design hardware without the laborious tasks involved in traditional RTL methodologies. Examples of some C-based HLS frameworks are Handel-C [1], [2], AutoESL [3], SPARK [4], LegUp [5], Mentor's Catapult C, and Synphony C compiler.

Although C-like languages are the preferred language for algorithmic specifications, they have also been criticized as not being suitable for hardware designs [6], [7]. We find that there are three major criticisms. The first major criticism is that C-like languages impose sequential semantics whereas hardware is inherently parallel. Therefore, synthesis tools need to automatically extract parallelism from the sequential specifications [8]. The second criticism is that C-like languages do not provide mechanisms to control the timing behaviour [7] of a design. That is, the time at which an output is available cannot be clearly defined or easily deciphered. This is a considerable concern for designers that are building hardware components of a larger system that must behave with specific timing behaviours. By abstracting away the notion of time, C-like languages do not natively provide any mechanisms to make the trade-offs between area and latency, which we find are necessary for designing efficient hardware. Therefore, designers are left at the mercy of HLS frameworks to decide the performance of the resulting hardware circuit. The third criticism is that formal methods and verification are not integrated into HLS methodologies. As a result, existing tools and methods for model-checking, automated testbench generation and equivalence checking cannot be trivially leveraged.

In response to these criticisms, there are several efforts that extend C-like languages with constructs to express parallel computation [2], [8], timing models [2], and that integrate formal methods [9]. For example, Handel-C extends C with a par construct for explicit parallelism, and Kiwi [8] leverages the concurrency mechanism in .NET for parallel specifications. These constructs allow the designer to explicitly make space and time trade-offs, and potentially expose parallelism that would otherwise be difficult to extract. Another innovation by Handel-C is its simple timing model; each assignment takes one clock cycle. This enables designers to clearly specify the timing requirements of their design. Most C-based HLS methodologies do not provide a timing model. In addition, seamless integration with formal methods also largely remains absent. Note that while the aforementioned synthesis frameworks provide some support for explicit parallelism and/or control over the timing behaviour of the design, they do not present its impact on scheduling algorithms.

Traditional hardware description languages (HDL)s such as VHDL and Verilog provide the ability to specify parallelism and timing. Designers express parallelism through concurrent processes and statements, and timing through FSMs. As a result, it is possible to only use HDLs to control the parallelism and the temporal behaviours. This, however, requires the designer to make scheduling and resource sharing decisions that optimizes the hardware for the metric of interest such as latency, and resource savings. HLS methodologies relinquish the designer of this burden by automating the scheduling and allocation, which is one of the primary advantages of HLS. Nonetheless, modern HLS methodologies typically abstract away both the notion of parallelism and time away from the designers. This makes efficient automatic synthesis very difficult. It is our opinion that by providing HLS methodologies with simple methods for expressing certain timing requirements and parallel behaviours in combination with completely sequential behaviours provides a balance between the two proposed extremes. We attempt to provide this balance in this work.

To this end, we explore the benefits of parallel and timed constructs for scheduling with the focus on resource savings in HLS. Our specification uses ASMs, which are a form of concurrent state machines that are straightforward for both

software and hardware engineers to understand. Even though we use ASMs for specification, the techniques we present in this article may be applied to other C-like languages that support parallel and timed constructs. ASMs allow specifying both sequential and parallel computation with clear definition of their composition [10]. They have formal operational semantics making ASM specifications executable, which is necessary for simulation. There is a significant amount of prior work that uses ASMs for verification [11], [12], [13], giving formal semantics [14], and automated testbench generation [15], [16], [17]. While we can leverage these works, and integrate it into our HLS framework, we restrict the focus of this article on the synthesis of ASMs to hardware circuits, the use of parallel and timed constructs, and their scheduling [18], [19].

Even though ASMs allow specifying parallel computation through the **par** construct, they do not provide support for specifying timing requirements. Therefore, we extend ASMs by defining the syntax and its appropriate semantics to support timed constructs to allow designers the ability to control the timing behaviour of the specification [19]. Specifically, we introduce **tseqblock** and **nseqblock** timed constructs. **tseqblock** provides a cycle-by-cycle granularity on each statement, and **nseqblock** enables a coarse-grained end-to-end timing granularity on the enclosed statements. Providing support for explicit timing may be considered too low-level for a HLS methodology; however, we contend that this is a critical component to hardware design for the above mentioned reasons. Furthermore, by providing timing requirements through timed constructs, our scheduling algorithms can perform optimizations while honoring these requirements. This is a key feature of the synASM framework.

Our synthesis framework accepts the extended ASMs as a specification, and it generates synthesizable VHDL. This synthesis back-end introduces a timed force-directed scheduling (TFDS) algorithm, which maximizes resource sharing across parallel and timed constructs. Furthermore, we perform optimizations that exploit the parallel and timed constructs. Through experimentation, we show that there are advantages of allowing designers to guide the synthesis engine by explicitly exposing parallelism, and prescribing timing requirements in the specification. In particular, we focus on the LUT usage metric, but we also present total area reductions. Our experiments show improvements of up to 52% in LUT usage and 34% in total area for certain examples. We also compare our synthesis results with Handel-C [1], [2].

A. Main Contributions

The main contributions of this work are the following.

- We present a HLS methodology that uses ASMs as its specification language, and generates synthesizable VHDL called synASM.
- We extend the definition of ASMs to provide support for timed constructs, and their composition with sequential and parallel constructs.
- We define an algorithm TIME that computes clock cycle values for timed constructs.
- We present a new definition of the mobility function in FDS to support these timed constructs, which we call timed FDS (TFDS).
- We present two optimizations for the parallel composition of timed constructs, and the parallel composition of a mixture of timed and untimed constructs that aid TFDS in further resource savings.

II. Related Work

A. HLS Frameworks

In recent years, we have seen a surged interest in HLS methodologies [6] with various academic and commercial HLS methodologies and tools emerging. We focus on a subset of the HLS methodologies based on C-like languages.

The Handel-C [1], [2] methodology extends the C language with extensions to support parallel computation via the par construct and it includes a timing model. The par construct allows designers to specify parallel computation, which otherwise would be difficult to automatically extract from a pure sequential specification. The underlying model of parallel computation is that of communicating sequential processes (CSPs). CSPs consist of independent processes that interact with each other through fully interlocked message passing. The timing model is simple: each assignment synthesizes to a cycle in hardware. While this allows a method to describe computations occurring at each cycle, we find it to be restrictive. We observe that there are certain segments of the computation where a designer may want to describe computation happening at each cycle, but there are cases where the designer may want end-to-end guarantees on the number of clock cycles spent in a piece of computation. Consequently, synASM provides the flexibility of specifying timing requirements through **tseqblock**s and **nseqblock**s as well as untimed computation using **seqblock**s. This flexibility does not exist in Handel-C. In addition, Handel-C allows explicit parallelism with the par construct; however, it requires the designer to ensure that the parallel statements are free of conflicts [20]. For a complex combination of parallel computation, we find that it is difficult to detect potential state conflicts through inspection. An advantage that ASMs have with respect to the **par** block is that there is a clear semantics of the state conflicts, which we adhere to during synthesis. These can also be detected during simulation of ASMs.

The Kiwi framework takes a similar approach to Handel-C in that it incorporates system-level concurrency abstractions into its HLS methodology. However, Kiwi is based on the C# language [8], and it uses threads, monitors and mutexes (concurrency mechanisms in C#) to specify parallel computation. While the idea of presenting parallel computation in the specification is valuable, Kiwi lacks the ability to specify timing behaviours.

The LegUp [5] open source tool generates hardware accelerators from C code. LegUp is unique in that it synthesizes the specifications to hybrid architectures containing hardware accelerators, and FPGA-based MIPS soft cores. It also automatically generates the necessary communication logic over a standard bus. AutoESL [3], and Forte's SystemC

Cynthesizer [21] support synthesis from SystemC algorithmic specifications. This addresses the lack of concurrency in C-like languages, but the requirement to learn the discrete-event semantics for algorithmic specification is time consuming for non-hardware experts.

Bluespec synthesizes RTL from the Bluespec SystemVerilog language [22], [23], which is based on the concurrent action-oriented specification (CAOS) model of computation. A CAOS specification consists of a set of guarded atomic actions or rules. An implementation of CAOS specification executes rules (whose predicates are enabled) one at a time, in a non-deterministic order. The Bluespec compiler optimizes the implementation by firing multiple rules in parallel while still preserving the correctness of the program. Thus, explicit parallelism is not supported in Bluespec, and the compiler conservatively analyzes rules to investigate parallelism amongst them.

B. Scheduling

Timed constructs impose specific timing constraints on hardware designs. For that reason, we choose a time-constrained scheduling algorithm to minimize functional units in a design with fixed number of clock cycles. Different techniques for time-constrained scheduling include mathematical programming (eg. Integer Linear Programming) and constructive heuristics (eg. force-directed scheduling (FDS)) [24]. Another approach to the scheduling problem is resource-constrained scheduling, which minimizes the number of clock cycles without exceeding the specified number of functional units. List-based scheduling is an example of this approach. as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) are also list based scheduling algorithms, but they do not limit the number of functional units [25]. Although the force-directed list scheduling technique incorporates hardware constraints from the user, it does not guarantee that the schedule will satisfy the timing constraints [26].

Integer linear programming (ILP) formulates the scheduling problem as an optimization problem to minimize functional units. Although it solves for the global minimum, ILP does not scale with the problem size. Therefore, this method is only practical for scheduling operations for small designs.

FDS is a heuristic method for time-constrained scheduling. It minimizes functional units in a design by efficiently sharing them amongst operations in different clock cycles [24]. The advantages are: 1) FDS extracts parallelism amongst sequential operations within a control/data flow graph (CDFG), 2) FDS can be extended to support timing constraints by extending the definition of mobility, and 3) FDS is computationally inexpensive compared to ILP for very little loss in quality of results. For these reasons, we select the FDS scheduling algorithm. synASM warns the user if the schedule violates the hardware constraints.

III. Background

A. Abstract State Machines

The ASM model of computation comprises of a set of transition rules that describe the evolution of the state. A transition rule (or simply called a rule) is of the form:

$$\text{if } \textit{Guard} \text{ then } \textit{Updates}$$

where *Guard* evaluates a Boolean expression and *Updates* is a finite set of assignments. An update is of the form:

$$f(a_1, a_2, ..., a_n) := a_0$$

where a_0 to a_n are arguments and f supplied with its corresponding arguments denotes the location to update with the value of argument a_0. A step in this model of computation first evaluates the arguments a_0 to a_n for their values, which we denote as $v(a_0)$ to $v(a_n)$. Once the values are evaluated, the assignments in the *Updates* set are computed. Therefore, $f(v(a_1), v(a_2), ..., v(a_n))$ gets the value of $v(a_0)$.

When the *Guard* of multiple transition rules evaluate to true, the assignments in the *Updates* set are applied simultaneously. This is known as a run of an ASM. Notice that simultaneous updates to the same location result in conflicts. A conflict is a result of a conflicting *Updates* set. This occurs when the same location is scheduled to receive two different values. For example, the *Updates* set $\{f(v(a_1), , ..., v(a_n)) := v(a_0), f(v(a_1), ..., v(a_n)) := v(a_1)\}$ such that $v(a_0) \neq v(a_1)$ is a conflicting update set.

A basic ASM contains a set of rules, where each rule produces a *Updates* set. In the synchronous model of execution, all rules in the ASM specification are scheduled to execute in a step. This union of *Updates* (of all rules) indicates the next state values of the ASM.

B. CoreASM Engine

CoreASM [27] is a Java-based open-source framework for modeling and simulating ASM specifications. It is designed with a plugin-based software architecture that makes extending the framework simple. Aside from its support for multiple schedulers for synchronous and asynchronous ASMs, it also has a plug-in for the formal verification of ASM specifications. CoreASM's software architecture has four components: the parser, the abstract storage, the interpreter, and the scheduler. Each of these components can be extended. This includes extending CoreASM with scheduling policies, datatypes and a type system, back-end code generators, and syntactical and semantical additions. The extensible nature of CoreASM makes it an ideal candidate upon which we build our HLS methodology and framework.

IV. Synthesis Framework Design Flow

Figure 1 shows our HLS methodology. The first stage in our methodology involves the algorithmic specification using ASMs. This is the behavioural specification of the intended design. The designer can use the parallel and timed constructs to explore space and time trade-offs, and guide the synthesis process to generate efficient hardware. While Figure 1 shows a path from C-like languages to ASMs, we currently write our specifications directly using ASMs. We are in the process of implementing a translator that converts a C specification into ASMs, but that is not the focus of this article. We extend the CoreASM [27] framework with the synthesis back-end that

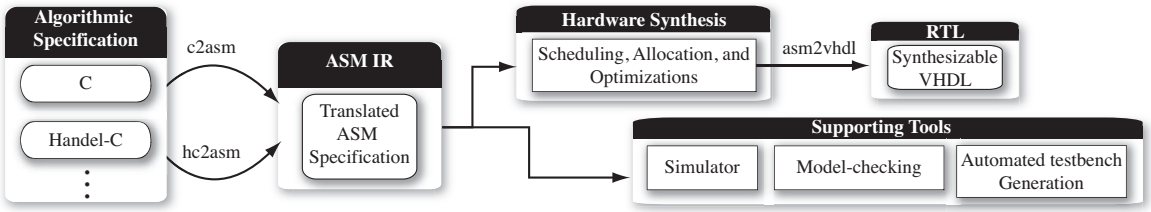


Fig. 1: Design flow of the proposed HLS methodology

performs scheduling and allocation, and generates synthesizable VHDL. synASM transforms the ASM specification into a control/data flow graph (CDFG). Unlike traditional CDFG structures, we extend it to support a combination of parallel and timed constructs. The next stages perform scheduling via the timed FDS algorithm followed by allocation. The scheduling algorithm partitions the CDFG into subgraphs, and determines the starting clock cycle of the operations in the subgraphs. The final stage generates synthesizable VHDL for the entire design, which we then target onto the Altera DE2 FPGA platform.

V. Synthesis from ASMs

Our back-end supports the core constructs of ASMs as implemented by CoreASM [27]. We include support for basic datatypes such as Boolean, numbers, enumerations and function elements.

TABLE I: Synthesizable constructs in synASM

Construct	Syntax
Block	**par** $statement_1$ ... $statement_n$ **endpar** **seqblock** $statement_1$ ... $statement_n$ **endseqblock** **tseqblock** $statement_1$ ... $statement_n$ **endtseqblock** **nseqblock** $statement_1$ ... $statement_n$ **endnseqblock**
Forall	**forall** element **in** domain **with** guard **do** statement
Conditional	**if** guard **then** $statement_1$ **else** $statement_2$
Macro Rule	rule(a_1,...,a_n)
While	**while** (guard) statement
Update	function(arguments) := value

Case	**case** var **of** {
	a_1 : $statement_1$
	a_n : $statement_n$
	}

A. Synthesis

Table I shows the set of synthesizable statements. A statement can itself be defined using statements. For example, the conditional statement has two possible branches: the taken and the not taken. The behaviour of each of these branches is also defined using statements. Note that a statement can nest multiple statements. These are typically denoted using the **seqblock** and **par** block statements. Each block statement produces an *Updates* set, and based on the type of block statement, the *Updates* set are combined. We now describe some of the important statements, and their synthesis to hardware.

1) Enumerations: Enumeration (enums) elements in ASMs are used for arguments and return values. Functions that return enums must store the binary representation of the elements in FPGA registers. Our synthesis tool incrementally assigns numeric values to the enumeration elements starting from 0. Since enums may also be used as function arguments, and functions define state of the design, their numeric value must be non-negative to allow indexing the array of registers synthesized for functions.

2) Functions: State is denotationally defined using function declarations in ASMs. The function declaration denotes the domain and range for which the state is defined. For example, the two functions in ASM Spec. 1 describe the states for a register of size 8 bits unsigned (`x`), and an array addressed using 8 bits unsigned that stores 32-bit words signed (`weights`). We synthesize these functions as an array of registers. These can then be placed onto block ram cells or logic cell flip-flops depending on the memory/state mapping optimization.

ASM Spec. 1 Example of Function Declarations.

```
1 function x: → UNSIGN_NUMBER8
2 function weights : UNSIGN_NUMBER8 → SIGN_NUMBER32
```

3) Parallel Block Statement: The **par** block statement contains statements that describe parallel computation. For the example in ASM Spec. 2 each statement produces an *Updates* set. The *Updates* set of the **par** block statement is computed as the union of the *Updates* of its constituent statements. For this example, the *Updates* set for the **par** statement is {(x,5+c), (y,6+c)}. During synthesis, these statements generate parallel hardware. The synthesis of this example generates two adders. The first adder computes $5 + c$ and stores the result in state `x`, and the second adder computes $6 + c$ and writes the result to state `y`. Note that c is a constant. Both these additions happen in the same clock cycle.

4) Forall Statement: The **forall** statement also describes parallel computation. It does this by enumerating over all the elements in the domain (refer to Table I) and evaluating the statements within the **forall** statement with these enumerations. For the example shown in ASM Spec. 3, the following statements are evaluated in parallel: *medfilt(0,0)*, *medfilt(0,1)*, ..., *medfilt(639,479)*. This allows designers to concisely represent a hardware design that contains replicated blocks for performing the same operation. Our synthesis of the **forall** statement performs loop unrolling such that the median operation for each pixel occurs in parallel. We take the

ASM Spec. 2 par

```
1 par
2   x := 5 + c
3   y := 6 + c
4 endpar
```

ASM Spec. 3 forall

```
1 forall row in [0 .. 639] do {
2   forall col in [0 .. 479] do {
3     medfilt(row,col)
4   }
5 }
```

ASM Spec. 4 seqblock

```
1 seqblock
2   y := 1
3   x := 1
4   x := 2
5 endseqblock
```

ASM Spec. 5 nseqblock

```
1 nseqblock[5]
2   x := 1
3   y := x + 2
4   z := 3
5 endnseqblock
```

statement after the **do** keyword and generate an instance of that statement for each enumeration. The generated HDL contains dedicated hardware computing *medfilt()* for each pixel data. Note that *medfilt()* is an invocation of another rule, which are called macro rules. We explain macro rules in Section V-A5.

5) Macro Rule Call: A macro rule call is a statement that invokes another rule. Macro rules enable modular design by 1) treating each rule as a component of the design and 2) allowing component reuse across design entities. The invocation of *medfilt(...)* is an example of a macro rule call. The computation inside the macro rule is scheduled as any other statement. The type of the enclosing block (either **par** or **seqblock/tseqblock**) governs whether the macro rule code executes sequentially or in parallel with respect to other statements. Our synthesis inlines the behaviour in the macro rules at the location of their use in the ASM specification. The primary reason for inlining is that it enables scheduling algorithms to make better optimization decisions by analyzing a larger fraction of the program.

6) Sequential Block Statement: A sequential block statement executes statements in program order. ASM Spec. 4 shows an example using the **seqblock** statement. In this example, the statements on lines 2 and 3 produce a combined *Updates* set of $\{$(x,1), (y,1)$\}$. The statement on line 4 updates the value of x to 2 resulting in $\{$(x,2),(y,1)$\}$. Therefore, the sequential block yields $\{$(x,2), (y,1)$\}$ as the *Updates* set. A sequential specification such as this has no well-defined timing model. As a result, analysis techniques extract data dependencies to create a partial order amongst the operations. Scheduling of these operations give the clock cycle at which the operations occur. This is followed by resource allocation and binding. There are several well-known techniques such as ASAP, FDS, etc. that can be used with the sequential block statement.

7) Timed Sequential Block Statement: Timed sequential block statements have the same semantics as **seqblock** statements with the exception that we incorporate a timing model for synthesis. The timing model is simple: each update to a state takes one clock cycle. Since all constructs in synASM are eventually composed of update statements, the designer can constrain individual state updates to specific clock cycles. Consider ASM Spec. 4 with **tseqblock** instead of **seqblock** (lines 1 and 5). In this modified example, the sequential block yields a combined *Updates* set of $\{$(y,1,0),(x,1,1),(x,2,2)$\}$. Notice that the tuple now contains a cycle time value at the end. For instance, (y,1,0) means that $y := 1$ happens in clock cycle 0. With a well-defined timing model, a timed sequential block is subject to time-constrained scheduling such as force-directed scheduling. Next, we synthesize **tseqblock** statements as a finite state machine (FSM). This FSM schedules each assignment to consecutive clock cycles.

8) N-Timed Sequential Block Construct: A **nseqblock** presents a relaxed timing model compared to the **tseqblock**. **nseqblock** accepts an additional parameter n that guarantees the constituent operations occur within n clock cycles. Therefore, n specifies a hard deadline on the entire block's timing behaviour. ASM Spec. 5 shows an example of **nseqblock** with a deadline of 5 clock cycles. The clock cycle assignment of each operation is bounded by the deadline, and is affected by the data dependencies with other operations in the **nseqblock**. Therefore, at the end of the ASM step, the *Updates* set becomes $\{$(x,1,0,3),(y,3,1,4),(z,3,0,4)$\}$. The read-after-write dependency between *x := 1* and *y := x + 2* forces the computation of y to occur at least one cycle after the update to x. Hence, the upper bound for the update to x is 3 to allow at least one cycle for the computation of y. The final schedule is driven by optimizations that reduce LUT usage.

9) Parallel Block with Timed Sequential Block Statements: A caveat of specifying parallel computation is the potential for data races. The *Updates* set of the **par** block statement is computed as the union of the *Updates* of its constituent statements. However, an inconsistent *Updates* set for the **par** block surrounding **tseqblock** statements occurs if and only if the same state elements are written with different values in the same clock cycle. Consider ASM Spec. 6. In this example, the individual *Updates* sets for the two **tseqblock** statements would be $\{$(x,1,0),(y,2,1)$\}$ and $\{$(z,3,0),(x,4,1)$\}$, respectively. The surrounding **par** block composes these *Updates* sets to form $\{$(x,1,0),(z,3,0),(x,4,1),(y,2,1)$\}$. Notice that there is no conflict in this specification as x gets written by parallel statements at different clock cycles. If, however, the assignment to x in the second **tseqblock** was done in line 7, then the specification would result in a conflict.

10) Parallel Sequential and Timed Sequential Blocks: Updates produced by **seqblock** statements are not associated with any time value. A surrounding **par** block considers an *Updates* set as inconsistent if and only if parallel **seqblock** computations write different values to the same state element. As an example, let us replace the second **tseqblock** in ASM Spec. 6 with **seqblock**. The individual *Updates* sets for the two blocks would then be $\{$(x,1,0),(y,2,1)$\}$ and $\{$(z,3),(x,4)$\}$, respectively. The *Updates* set for the **seqblock** does not define when the operations occur; thereby, allowing them to occur at any clock cycle. Therefore, the union of these *Updates* set is in conflict.

11) Preserving Sequential Composition of Updates: Recall that states are updated at the end of a step of an ASM run. Therefore, with parallel composition of sequential blocks, we must prevent one **seqblock** from affecting others. For our example in ASM Spec. 7, the end of a step happens when

ASM Spec. 6 par with **tseqblock**.

```
1  par
2  tseqblock
3    x := 1
4    y := 2
5  endtseqblock
6  tseqblock
7    z := 3
8    x := 4
9  endtseqblock
10 endpar
```

ASM Spec. 7 Temporary Registers.

```
1  par
2    // Assume x, y, and z are initially 0
3    seqblock
4      x := 1
5      y := y + 1
6    endseqblock
7    seqblock
8      y := 2
9    endseqblock
10 endpar
```

ASM Spec. 8 Mobility in FDS

```
1  seqblock
2    w := k * 1
3    x := k * 2
4  endseqblock
```

ASM Spec. 9 par with **tseqblock**

```
1  par
2  tseqblock
3    w := k + 1
4    x := k * 2
5  endtseqblock
6  tseqblock
7    y := k + 3
8    z := k * 4
9  endtseqblock
10 endpar
```

all parallel statements within the **par** block are evaluated. Therefore, we expect the assignment in line 5 to produce a partial *Updates* set {(y,1)}. However, each **seqblock** statement updates the same instance of state y (i.e. the same register). Now, assume that the framework schedules each operation within a **seqblock** to consecutive clock cycles of the FSM. When the execution reaches line 5, y evaluates to 3 because line 8 modified the value of y in the previous clock cycle of the FSM. The assignment in line 8 has incorrectly become visible before the end of the ASM step.

To preserve the ASM semantics, we must ensure that the assignments within the sequential blocks do not immediately update the global state. As a result, we introduce intermediate states to hold these values before computing the updates to the global state. At the point where the execution flow splits into the parallel statements, the current state is registered into the temporary state. Each statement in the parallel block operates on its local states. Then, at the end of the parallel block statement, the consistency check logic combines the updates from each of the statements. This is a semantically correct implementation of the ASM consistency check semantics.

B. Checking for Consistent Updates

In order for us to detect potential data races, we generate hardware that checks for the consistency of updates at runtime. During execution, if an inconsistent update is identified, the hardware asserts a signal identifying that a conflict occurred. Note that this is only required during verification and debugging of the hardware design. Therefore, we provide a setting that disables the generation of the consistency check hardware.

Algorithm 1 details the internals of consistency check logic for a statement *S*. Consistency check logic introduces another set of signals, hereafter referred to as check bits. In this algorithm, *W* denotes the set of functions being written by statement *S*. Set *B* contains the statements nested within *S*. In addition to executing the specification, the first cycle registers the global state into temporary registers for every parallel statement (*preserve*). For each block statement, the algorithm only generates temporary registers for functions in W that are also in the W of at least one other statement. The algorithm then recursively generates the consistency check logic for each statement. The recursion guarantees that the algorithm works for nested **par** statements. For each function in W, *GenerateCC* replaces each access of that function inside the statement to use the temporary register instead of global state. The read operations must access temporary registers because the temporary state may evolve inside a sequential block statement. A **tseqblock** has the added responsibility of tagging each constituent update with a clock count value. Additionally, *GenerateCC* generates a check bit via *generateCheckBit* for each update statement. We only generate check bits if the function being updated is present in the W set of at least one other statement.

Algorithm 1: $generateCC(S)$

```
   /* Initial declarations                                  */
1  W ← functions written by S
2  B ← nested statements within S
3  if (type(S) = par) then
4  |   preserve(W)
5  |   foreach statement ∈ B do
6  |   |   generateCC(statement)
7  |   end
8  |   compose(W)
9  end
10 if (type(S) = tseqblock) then
11 |   foreach statement ∈ B do
12 |   |   addClockTag(statement)
13 |   |   generateCC(statement)
14 |   end
15 end
16 if (type(S) = Update) then
17 |   generateCheckBit(W)
18 end
19 return
```

Different parallel statements can update the same state (line 5 and 8 in ASM Spec. 7). For such cases, we allocate multiple check bits for that state and assert them if the corresponding updates do take place (they could be guarded). The check bits for each state function in W are represented as a bit vector. The *compose* step generates hardware to detect if the bit vector contains more than one asserted bit, which denotes a conflict. *compose* must also check for the cycle count tags based on the consistency semantics described earlier. We adhere to the timing constraints imposed by **tseqblock** by doing the checks simultaneously with last assignment of the **par** block. Instead of asserting a check bit in the last cycle, we drive the consistency check hardware with the same guards as the **Update** statement. Thus, we remove the need for consuming another cycle for consistency check.

Table II shows an execution of ASM Spec. 7 with the consistency check (CC) hardware. Notice that we use two check bits for y because two parallel statements modify it. Since all three assignments take place, the execution results in conflicting updates for state y. We identify this by the 2 asserted bits in the check_y bit vector. Our implementation for identifying multiple set bits is to determine if the check_y bit vector is a power of two as follows: conflict = not ((check_y

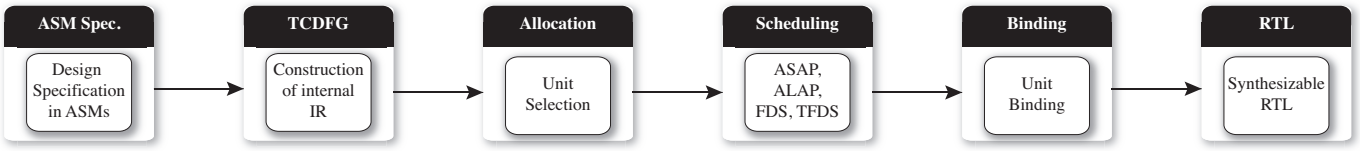


Fig. 2: Hardware synthesis backend

TABLE II: Execution with consistency check

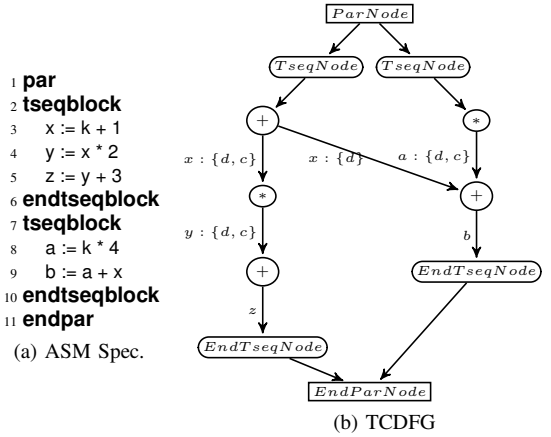
cycle i	**cycle** $i+1$
y_0:=y y_1:= y	
x:=1 y_1:=2	y_0:=y_0+1
check_x:=1 check_y(1):=1	check_y(0) := 1
	pow2(check_y)

```
& check_y - 1) = "00").
```

VI. Extending CDFGs to Support Parallel and Timed Specifications

We support explicit parallelism using ASM **par** block construct, and timed constructs with **tseqblock** and **nseqblock**. To capture a design using these constructs, we construct a control/data flow graph (CDFG). Scheduling and allocation are operations on this CDFG representation. However, conventional CDFGs [28] do not provide a method for supporting parallel and timed constructs. This is because traditional HLS methods automatically extract parallelism from sequential specifications, and there is no ability to specify explicitly parallel and timed behaviours. As a result, we extend CDFGs with special nodes and edge labels that enable us to use the extended CDFG for scheduling. We term these extended CDFGs timed CDFGs (TCDFG)s as described in Definition 1. The synthesis engine back-end flow is shown in Figure 2.

Definition 1. *A TCDFG is a directed acyclic graph* $G = (V, E)$ *where* $V = \{(i, V_T) : i \in \mathbb{Z}^+$ *s.t.* i *is unique*, $V_T \in T_V\}$ *is the set of nodes and* $E = \{(v_i, v_j, E_T) : v_i, v_j \in V, E_T \subseteq T_E\}$ *is the set of edges.*



(a) ASM Spec.

(b) TCDFG

Fig. 3: Example of ASM spec. and its corresponding TCDFG

We define the set of node types as $T_V = \{Operation, CondNode, CaseNode, LoopNode, SeqNode, ParNode, EndParNode, TseqNode, EndTseqNode, NseqNode, EndNseqNode\}$. Notice that we extend the set of node types with six new node types: $ParNode$, $TseqNode$, $NseqNode$, and their corresponding ending nodes. We also extend the set of edge types T_E with two types: d for data dependency edges, and c for signifying that the next node occurs in the next clock cycle. Therefore, $T_E = \{d, c\}$. For convenience, we use a function $type(x)$ where $x \in V \cup E$ to return the node type or the set of associated edge types. The scheduling and allocation algorithms operate on the TCDFG structure. Refer to Figure 3b for the TCDFG representation of the ASM specification in Figure 3a. The labels on edges describe whether the edges are of type data dependence, control, or both. Control edges only exist within a **tseqblock** construct. The outgoing edges from nodes of type $Operation$ also denote the state locations being updated by the operations. For example, the edge labelled $y : \{d, c\}$ denotes the following: 1) the multiplication writes to the state y, 2) there is data dependency between $y := x*2$ and $z := y+3$, and 3) the update $z := y+3$ occurs one clock cycle after $y := x * 2$. synASM also extends the semantics of **par** block constructs that nest **tseqblock** constructs. Specifically, there can be data dependencies amongst operations within parallel **tseqblock** constructs. We use clock cycle assignments to ensure that an operation is only data dependent on an earlier parallel operation. The specification in Figure 3a has a read-after-write data dependency between $x := k+1$ (cycle 0) and $b := a + x$ (cycle 1).

VII. Scheduling Operations in Timed Constructs

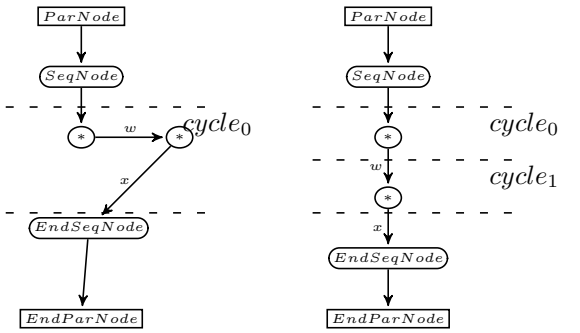
Data-path operations are performed using functional units (such as adders, multipliers, etc.). These operations are scheduled such that the synthesized hardware obeys design constraints and objectives. We present the definition of a schedule in Definition 2.

Definition 2. *A schedule of a TCDFG is a function* $\rho : V_{ops} \rightarrow \mathbb{Z}^+$ *where* $\rho(v_i) = c_i$ *is the operation's start clock cycle time such that* $c_i \leq c_j, \forall i, j \in \mathbb{Z}^+ : (v_i, v_j) \in E$ *and* $V_{ops} = \{v : v \in V, type(v) = Operation\}$.

Our primary goal in scheduling is to reduce LUT usage by sharing resources. With timed constructs in HLS, we take the approach of time-constrained scheduling. To this end, we use FDS to balance operations across all clock cycles such that functional units used in one clock cycle are reused in other clock cycles. FDS is appropriate because it optimizes resource sharing.

A. Force-Directed Scheduling

FDS uses as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) schedules of a CDFG for scheduling. ASAP sorts the operations such that they are placed in the earliest possible clock cycle. On the other hand, ALAP sorts the operations topologically such that each operation is assigned to the latest possible clock cycle. FDS defines the difference between the ASAP and ALAP schedule for any operation as the mobility. The mobility determines the likelihood of an operation being assigned to any clock cycle in that range. The final assignment is done by the schedule that allows the most resource sharing. We illustrate the impact of mobility on resource sharing with the example in ASM Spec. 8. If the design latency is fixed to 1 cycle, both ASAP and ALAP schedule the multiplications to cycle 0. The lack of data dependency between the two multiplications allows parallel execution. In this scenario, we say that the mobility is 0 cycles for each operation because the difference between ASAP and ALAP is 0. With 0 mobility, the schedule is forced to use 2 multipliers (Figure 4a), otherwise known as the cost of schedule. Now, let us increase the design latency to 2 cycles. The ASAP algorithm schedules both multiplications to cycle 0, and the ALAP algorithm schedules them both to cycle 1. FDS has the option of scheduling both multiplications to cycle 0, or both to cycle 1, or one to cycle 0 and one to cycle 1. In this case, FDS schedules the multiplications to different cycles in order to share the multiplier. Therefore, increasing the mobility from 0 to 1 clock cycle permits us to share the multiplier for both multiplications (Figure 4b). Hence, the higher the operation's mobility, the more opportunities there exist for resource sharing optimizations. We refer the reader to [26] for further details on FDS.



(a) Mobility: 0 cycles (b) Mobility: 1 cycle

Fig. 4: FDS scheduling with different mobility

B. Extending FDS to Support Timed Constructs

In order to use FDS scheduling for timed constructs, we need to make two extensions to traditional FDS: 1) we need to compute an additional schedule specific for timed constructs, and 2) we need to extend the definition of mobility.

1) Scheduling Timed Constructs: We call this the TIME schedule. TIME maps all operations within **tseqblock** and **nseqblock** constructs to clock cycles as shown in Algorithm 2. The arguments to TIME are the TCDFG G, starting clock cycle (usually 0), and the empty set S denoting the schedule. At completion of TIME, S is populated with tuples that map operations to clock cycles. Since operations within **par** execute in parallel, we invoke TIME with the same time parameter t for each constituent operation. The timed constructs schedule operations in program order to consecutive cycle values. As an example, we invoke TIME (G, 0, S) on ASM Spec. 9. The surrounding **par** block schedules both **tseqblock** constructs starting from cycle 0. The schedule S of this rule is $\{(w := k{+}1, 0), (x := k{*}2, 1), (y := k{+}3, 0), (z := k{*}4, 1)\}$. We use Definition 2 to extract the clock cycle value for each node in the following way: $\rho_{TIME}(v_i)$ where v_i is the node corresponding to $x := k + 2$ returns clock cycle value of 1.

Algorithm 2: $TIME(G, t, S)$

```
Let duration ← 0 be the duration
Let subNodes ← [] be an empty list
Let v_i be the root node of G
if (type(v_i) = ParNode) then
    foreach v_j ∈ {v_k : (v_i, v_k) ∈ E} do
        duration ← max(duration, TIME(v_j, t, S))
    end
    return duration
end
if (type(v_i) = TseqNode ∨ type(v_i) = NseqNode) then
    subNodes ← getSubNodeList(v_i)
    foreach v_i in subNodes do
        duration ← duration + TIME(v_j, duration, S)
        t ← t + 1
    end
    return duration
end
if (type(v_i) = Operation) then
    S ← S ∪ {(v_i, t)}
    return 1
end
return duration
```

2) Extending the Definition of Mobility: An operation's mobility depends on the type of enclosing block. For instance, $w := k + 1$ in ASM Spec. 9 is enclosed by **tseqblock**, which means it has the mobility defined by **TseqNode**.

Definition 3. *The mobility of an operation is a function* $mob : V \rightarrow \mathbb{Z}^+ \times \mathbb{Z}^+$ *where*

$$mob(v_i) = \begin{cases} (\rho_{ASAP}(v_i), \rho_{ALAP}(v_i)) \\ \qquad : type(v_i) = \textbf{\textit{SeqNode}} \\ (\rho_{ASAP}(v_i), \rho_{TIME}(v_i)) \\ \qquad : type(v_i) = \textbf{\textit{TseqNode}} \\ (max(\rho_{ASAP}(v_i), \rho_{TIME}(top(v_i)) + n) \\ \qquad : type(v_i) = \textbf{\textit{NseqNode}} \end{cases}$$

where $v_i \in V_{ops}$ *and* $top(v_i)$ *returns the first operation node within that* ***nseqblock****.*

Notice that we subscript the scheduling algorithm ρ to denote the algorithm used for scheduling. Therefore, $\rho_{ASAP}(v_i)$ returns the scheduled clock cycle value for node v_i using ASAP scheduling. The mobility of **seqblock** is bound by the ASAP and ALAP schedules. These are appropriate because ASAP and ALAP optimize based on data dependencies, and they are independent of timing constraints. Notice that this is the traditional definition of mobility in FDS [26]. However,

our definition extends this for timed constructs. The **tseqblock** uses the TIME scheduling algorithm, which imposes a strict constraint on when each operation must occur. This means that the mobility of an operation in **tseqblock** is 0 because the lower and upper bounds are both the same. We can, however, optimize the lower bound to the ASAP schedule by pre-computing the operations and temporarily storing the result. Details of this technique are described in Section VII-C. The mobility of each operation within an **nseqblock** is simply the parameter n because an operation can be scheduled in any of the n cycles. However, due to data dependencies, we can not schedule an operation within a **nseqblock** prior to its ASAP schedule.

3) A timed extension to FDS: We formulate our definition of the TFDS algorithm using Definition 3 of mobility.

Definition 4. *Timed forced-directed scheduling (TFDS) is force-directed scheduling with mobility being the mob function from Definition 3.*

C. TFDS for Parallel Composition of Timed Blocks

synASM supports a strict timing model with **tseqblock** constructs. The semantics of **tseqblock** constrains each update to a specific clock cycle. For the example in ASM Spec. 9, assume that k is a state variable with initial value 0. The two **tseqblock** constructs produce *Updates* sets $\{(w, 1, 0), (x, 2, 1)\}$ and $\{(y, 3, 0), (z, 4, 1)\}$, respectively. This means w gets the value of 1 at cycle 0, x gets the value of 2 at cycle 1, and so on. Therefore, **tseqblock** restricts when updates to state occur.

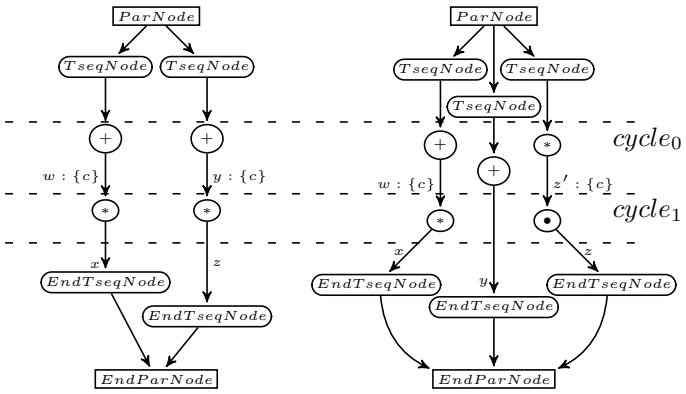
We utilize TFDS to optimize the synthesis of **tseqblock** constructs. However, to better assist TFDS, we perform an optimization that increases the mobility of the operations in **tseqblock**s. In particular, we perform computations earlier and temporarily store the result in a register. We then update the state from the register at the clock cycle specified by the TIME schedule. The TIME schedule specifies the clock cycle based on the timing semantics of **tseqblock**. This preserves the semantics of **tseqblock** constructs, and it increases mobility.

We illustrate this optimization with the example in Figure 5 for ASM Spec. 9. Figure 5(a) shows the unoptimized schedule, which executes the operation in the same clock cycle as the corresponding state update. This requires 2 adders and 2 multipliers. On the other hand, with the optimization, TFDS moves $k * 4$ into cycle 0 because it is not dependent on an earlier operation. Notice that we store the computed value in a temporary register, and update z at the clock cycle denoted by the **tseqblock**. Doing so allows the multiplier to be used in both clock cycles. This optimization transforms the TCDFG by adding a special node to denote the registering (a circle with a dot in the middle). The node symbolizes the assignment $(z := z')$ to state z from the temporary register z'. We show the optimized schedule in Figure 5(b) that uses 2 adders, 1 multiplier, and 1 register.

Algorithm 3 details this optimization. We first compute the TIME schedule and the TFDS schedule. The TIME schedule provides the clock cycle value at which the state updates of the operations must occur, and the TFDS schedule provides a clock cycle value when the computation must occur. Whenever the latter (c_i) is less than the former (c_j), we perform this optimization. We reflect this optimization in Definition 3 where the lower bound of the mobility of **tseqblock** constructs is determined by the ASAP schedule instead of the TIME schedule.

Algorithm 3: $timedOpt(G)$

```
Let S1 ← {(v, ρ_TIME(v)) : ∀v ∈ V} be TIME schedule
Let S2 ← TFDS(G) be the timed FDS schedule
foreach v ∈ {V : type(v) = Operation} do
    Let (v_i, c_i) ∈ S2, (v_j, c_j) ∈ S1
    if (type(v) = TseqNode ∧ c_i < c_j) then
        register(v)
    end
end
```



(a) Unoptimized TCDFG (b) Optimized TCDFG

Fig. 5: Optimization for parallel timed blocks

D. TFDS for Parallel Composition of Timed and Untimed Blocks

We schedule the composition of **seqblock** and **tseqblock** constructs such that we share the resources amongst the enclosed operations. Since the **seqblock** construct has no timing model, we are able to explore area versus latency trade-offs. For example, we can share a multiplier for operations within parallel **tseqblock** and **seqblock** by delaying the multiplication in the **seqblock**. Our approach is broken into three steps: 1) schedule timed operations within **tseqblock** constructs using TFDS, 2) schedule operations within **seqblock** constructs that can share functional units with one of the timed operations, and 3) delay remaining operations within **seqblock** such that they occur after all **tseqblock** operations have finished.

We demonstrate this optimization for ASM Spec. 10. In step 1, we schedule **tseqblock** operations $w := k + 1$ and $x := k * 2$ to cycles 0 and 1, respectively. In step 2, our algorithm schedules $y := k + 3$ from **seqblock** into cycle 1 to reuse the adder assigned in cycle 0. In step 3, we delay the remaining operation $z := k * 4$ into clock cycle 2. The optimized design in Figure 6a uses 1 adder and 1 multiplier and has a latency of 3 clock cycles. Contrast this with another valid schedule in Figure 6b which has identical resource

utilization but has a latency of 4 clock cycles. This hardware is a result of a scheduling algorithm that does not optimize for latency. On the other hand, our algorithm generates a schedule with the minimum possible resources, and then minimizes latency without increasing resources. However, if a trade-off exists, we choose LUT usage optimization over latency.

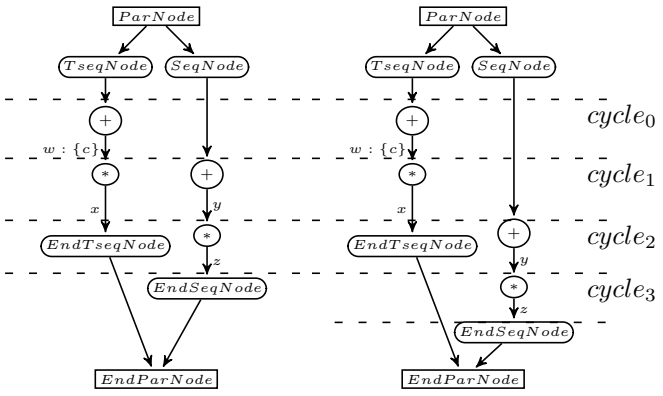
Algorithm 4: $timedUntimedOpt(G)$

Let $ParNodeSet$ be the set of $ParNode$ nodes that enclose $TseqNode$ and $SeqNode$ nodes
Let $B \leftarrow \emptyset$ be a temporary set of nodes
Let $tseqNodes \leftarrow \emptyset$ be an empty set of **TseqNodes**
Let $opNodes \leftarrow []$ be an empty list of **Operation** nodes
Let $seqNodes \leftarrow \emptyset$ be an empty set of **SeqNodes**
Let $Partial \leftarrow \emptyset$ be a partial schedule
Let $maxClock \leftarrow 0$ be the maximum clock cycle value
foreach $v_i \in ParNodeSet$ **do**
  $tseqNodes \leftarrow \{v_k : \forall(v_i, v_k) \in E \wedge (type(v_k) = TseqNode \vee type(v_k) = NseqNode)\}$
  $Partial \leftarrow TFDS(copySubGraph(tseqNodes))$
end
$maxClock \leftarrow getMaxClock(Partial)$
$B \leftarrow tseqNodes$
foreach $v_i \in ParNodeset$ **do**
  $seqNodes \leftarrow \{v_k : \forall(v_i, v_k) \in E \wedge type(v_k) = SeqNode\}$
  foreach $v_j \in seqNodes$ **do**
    $opNodes \leftarrow getSubNodeList(v_j)$
    foreach v_l ***in*** $opNodes$ **do**
      $New \leftarrow TFDS(copySubGraph(B \cup \{v_l\}))$
      if $cost(Partial) = cost(New)$ **then**
        $Partial \leftarrow New$
        $B \leftarrow B \cup \{v_l\}$
      else
        $Partial \leftarrow scheduleAfter(maxClock, Partial, v_l)$
      end
    end
  end
end

Algorithm 4 describes this optimization. In step 1, we first accumulate operations within **tseqblock** constructs in the set *tseqNodes*. We then schedule them using Algorithm 3 and store the schedule in *Partial*. *getMaxClock* returns the highest clock cycle value assigned to any timed operation. We use *maxClock* to delay **seqblock** operations in step 3. In step 2, we accumulate the vertices within **seqblock** constructs in the set *seqNodes*. We add each unscheduled operation in *seqNodes* one-by-one to the existing TFDS schedule. In this process, the operations are added to a cycle between 0 and *maxClock*. We only accept the new schedule if it does not increase the design *cost*. That is, it is able to use an available functional unit. This allows the operation to utilize existing resources from previous clock cycles. If the operation increases our cost, we jump to step 3 and delay this operation to clock cycles after *maxClock*. We implement the *scheduleAfter* utility to include this operation into the *Partial* schedule.

ASM Spec. 10 par with **tseqblock** and **seqblock**

```
1  par
2  tseqblock
3    w := k + 1
4    x := k * 2
5  endtseqblock
6  seqblock
7    y := k + 3
8    z := k * 4
9  endseqblock
10 endpar
```



(a) Latency: 3 cycles (b) Latency: 4 cycles

Fig. 6: Optimization for parallel timed and untimed blocks

VIII. Case Study: Differential Equation Solver

We present a case study of a differential equation solver in ASM Spec. 11. The inputs are denoted as functions in lines 1 to 5, and the outputs are in lines 6 to 8. We also need intermediate variables (also states) during the computation, they are listed in lines 9 to 13. At this point, it is valuable to discuss the design flow from writing this specification to synthesizing the design on the FPGA chip. Although this ASM contains a testbench rule, it is only required during simulation. During simulation, all inputs and outputs to the solver are modified to be internal state by changing the monitored and controlled functions to regular functions. The testbench in ASM Spec. 11 first sets values of all module inputs, and then invokes the Solve macro rule. Prior to synthesizing this design for the FPGA, the testbench may be removed from the specification or suffixed with **_nosyn** to inform the synthesis engine to not synthesize that macro rule. synASM generates VHDL described by the Solve rule, leaving the input (monitored) ports and output (controlled) ports dangling. This allows the Quartus synthesis flow to assign them to pins on the FPGA board. The remaining functions (intermediate variables) are assigned to flip-flops. Note that the specification also contains InitRule, which is the first rule to run during execution. It performs state initialization, including assigning agents to their corresponding rules.

The solver uses a parallel composition of timed and untimed sequential constructs. The TIME schedule for the timed operations is also listed as comments. The strict timing constraints on operations within **tseqblock** ensures that specific computations finish prior to the cycles in which their results are

ASM Spec. 11 Specification of differential equation solver

```
1  function monitored  i_x : → SIGN_NUMBER8
2  function monitored  i_dx : → SIGN_NUMBER8
3  function monitored  i_y : → SIGN_NUMBER8
4  function monitored  i_u : → SIGN_NUMBER8
5  function monitored  i_a : → UNSIGN_NUMBER8
6  function controlled  o_x : → SIGN_NUMBER8
7  function controlled  o_y : → SIGN_NUMBER8
8  function controlled  o_u : → UNSIGN_NUMBER8
9  function  x1 : → SIGN_NUMBER8
10 function  y1 : → SIGN_NUMBER16
11 function  u1 : → SIGN_NUMBER24
12 function  t1 : → SIGN_NUMBER8
13 function  t2 : → SIGN_NUMBER16
14
15 derived  c1 = 3
16 derived  c2 = 3
17
18 universe  Agents = {tb}
19
20 init  InitRule
21
22 rule InitRule = {
23     program(self) := undef
24     program(tb) := @testbench
25 }
26
27 rule Solve = {
28     while ( x < a) do {
29     par
30         tseqblock
31             u1:= u * dx   /*  cycle 0  */
32             u1:= u1 * t1  /*  cycle 1  */
33             u1:= u - u1   /*  cycle 2  */
34             u1:= u1 - t2  /*  cycle 3  */
35         endtseqblock
36         tseqblock
37             t1:= c1 * x   /*  cycle 0  */
38             t2:= c2 * y   /*  cycle 1  */
39             t2:= t2 * dx  /*  cycle 2  */
40         endtseqblock
41         seqblock
42             y1:= u * dx
43             y1:= y + y1
44         endseqblock
45         seqblock
46             x1:= x + dx
47         endseqblock
48     endpar
49     }
50 }
51
52 rule testbench = {
53     seqblock
54         par
55             u := 1
56             dx := 2
57             a := 3
58             x := 4
59             y := 5
60         endpar
61         Solve
62     endseqblock
63 }
```

TABLE III: Schedule for differential equation solver

	tseqblock		nseqblock	
Cycle	Timed	Untimed	Timed	Untimed
0	$u1:*, t1:*$	$x1:+$	$u1:*$	$x1:+$
1	$u1:*, t2:*$		$t1:*$	
2	$u1:-, t2:*$	$y1:*$	$u1:*$	
3	$u1:-$		$u1:-, t2:*$	
4		$y1:+$	$t2:*$	
5			$u1:-$	$y1:*$
6				$y1:+$

ASM Spec. 12 Partial spec. of differential equation solver

```
1 nseqblock[6]
2       u1:= u * dx
3       t1:= c1 * x
4       u1:= u1 * t1
5       u1:= u - u1
6       t2:= c2 * y
7       t2:= t2 * dx
8       u1:= u1 - t2
9 endnseqblock
```

used. We relax the timing constraints on **seqblock** operations to achieve LUT usage savings. We illustrate the schedule in Table III. In step 1 of Algorithm 4, we first schedule all **tseqblock** operations. Next in step 2, we schedule operations within **seqblock** to share the already instantiated functional units. For example, $y1 := u * dx$ uses the multiplier used by either $u1 := u1 * t1$ or $t2 := c2 * y$. We denote this using the entry $y1 : *$ in cycle 2 in Table III. In step 3, we delay the **seqblock** operations (that would have allocated additional resources if executed concurrently with **tseqblock** operations) to later clock cycles. The addition $y1 := y + y1$ is scheduled in cycle 4 to share the subtractor used by $u1 := u1 - t2$ in cycle 3. synASM uses the same functional unit for addition and subtraction. Our schedule for this specification uses 2 multipliers, 1 adder/subtractor, and 1 comparator. Note that the effective computation of $u1$ is actually $u1 * dx * t1 - t2$, but we split up the computation as shown in the specification. This is done solely for illustration purposes, and we could use other constructs such as the **nseqblock** with an argument of 4 to let the synthesis tool discover the best schedule for these operations.

The **tseqblock** construct forces the hardware to exhibit specific timing behaviours, and therefore limits scheduling optimizations. To get additional resource savings, we leverage the **nseqblock** construct with higher latency than the **tseqblock** construct. The partial ASM Spec. 12 uses the **nseqblock** construct in place of the two **tseqblock** constructs (lines 30 – 40 in ASM Spec. 11). The operations within the **nseqblock** construct have 6 cycles to finish execution. They need 1 adder/subtractor, 1 multiplier, and 1 comparator in total. The goal of Algorithm 4 is to maintain this design cost for untimed operations. Algorithm 4 delays untimed operation $y1 := y+y1$ by 1 cycle to reuse the adder from previous cycles, thus maintaining the cost.

Figure 7 shows a block diagram of the generated hardware for ASM Spec. 11. Since the functional units are shared by different operations, we multiplex the inputs that are driven by either state variables or constants. The multiplexors are controlled by a finite state machine labeled `controller`. The state is updated once the results are computed. Prior to updating the state, we perform a consistency check (using `check` logic) which prevents conflicting writes to the same state location. The details of consistency check are described in [18].

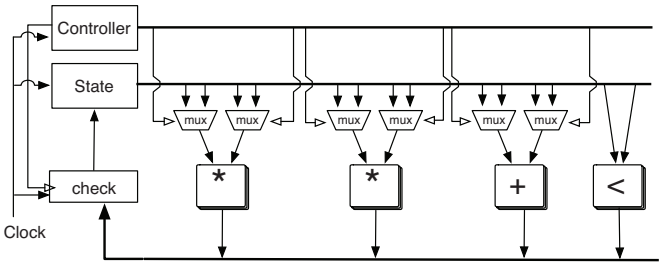


Fig. 7: Block diagram of generated hardware

IX. Results

Table IV presents our experimental designs consisting of designs from the CHStone benchmark suite [29], and in-house case studies. We convert these designs into ASM specifications. However, the original designs do not have explicit timing and parallelism in them. Consequently, we adapt these designs by appropriately adding parallel and timed constructs we expect designers identify most naturally. These benchmarks cover a variety of computations that include control-intensive (eg. microprocessor) and data-intensive (eg. AES encryption and decryption) types. We synthesize the designs using the Altera Quartus II for the Altera DE2 FPGA platform. We validate the correct operation of the designs by driving these designs with a set of input vectors and comparing the outputs

TABLE IV: Synthesis results of example designs

Strictly Constrained Designs	With CC Logic			Unoptimized			Optimized			Handel-C		
	LUTs	FFs	MHz	LUTs	FFs	MHz	LUTs	FFs	MHz	LUTs	FFs	MHz
Parallel FIR	504	102	86.1	248	102	101.3	248	102	101.3	281	140	134.4
Sequential FIR	271	102	94.9	**271**	102	94.9	**183**	102	91.4	208	133	121.3
Kalman Filter	1334	783	125.1	310	783	126.4	266	802	118.3	304	781	97.3
Diff. Eq. Solver	119	62	139.6	**119**	62	139.6	**86**	62	135.2	100	87	171.0
Edge Detector	167	1021	137.8	167	1021	137.8	138	1033	133.5	158	1050	145.4
AES encryption	1346	2653	38.5	834	2653	43.2	756	2722	40.8	864	2826	80.8
AES decryption	1398	2716	55.9	786	2716	61.0	739	2803	56.2	839	2826	76.0
Blowfish	162	1842	113.5	162	1842	113.5	162	1842	112.7	192	1883	109.8
Loosely Constrained Designs	With CC Logic			Unoptimized			Optimized			Handel-C		
	LUTs	FFs	MHz	LUTs	FFs	MHz	LUTs	FFs	MHz	LUTs	FFs	MHz
Parallel FIR	504	102	86.1	248	104	128.1	180	116	121.4	–	–	–
Sequential FIR	271	102	94.9	**271**	102	94.9	**185**	102	91.2	–	–	–
Kalman Filter	1334	783	125.1	310	784	126.1	178	811	113.2	–	–	–
Diff. Eq. Solver	119	62	139.6	**119**	62	**138.7**	**57**	62	**129.0**	–	–	–
Edge Detector	167	1021	137.8	167	1023	139.8	97	1037	127.6	–	–	–
AES encryption	1346	2653	38.5	834	2653	43.2	693	2701	41.2	–	–	–
AES decryption	1398	2716	55.9	786	2717	60.9	615	2789	57.7	–	–	–
Blowfish	162	1842	113.5	162	1842	114.1	162	1844	113.4	–	–	–

TABLE V: The reductions in LUTs, FFs and total area of the optimized circuits when compared to unoptimized and that generated using Handel-C

	Strictly Constrained						Loosely Constrained		
	Over Unopt.			Over Handel-C			Over Unopt.		
	LUTs %	FFs %	Area	LUTs %	FFs %	Area	LUTs %	FFs %	Area
Parallel FIR	0%	0%	0%	12%	27%	17%	27%	-9%	17%
Sequential FIR	32%	0%	24%	12%	23%	3%	32%	0%	23%
Kalmann Filter	14%	-2%	2%	13%	-3%	2%	43%	-3%	10%
Diff. Eq. Solver	28%	0%	18%	14%	29%	21%	52%	0%	34%
Edge Detector	17%	-1%	1%	3%	2%	2%	42%	-1%	5%
AES Encryption	9%	-3%	0%	13%	4%	6%	17%	-2%	3%
AES Decryption	6%	-3%	-1%	12%	1%	3%	22%	-3%	3%
Blowfish	0%	0%	0%	24%	2%	4%	0%	0%	0%

from the ASM simulation, VHDL simulation, and the output we get from running it on the FPGA.

We provide two versions of the benchmarks: strictly and loosely constrained. The strictly constrained versions have timing requirements using **tseqblock**s, and loosely constrained versions replace the **tseqblock**s with **nseqblock**s. These two versions incorporate timing requirements at different granularities. **tseqblock**s provide a strict timing requirement on each operation enclosed in the block, and **nseqblock**s give a coarse-grained timing requirement for the entire block. This allows designers who just need coarse-grained timing requirements to be met to avoid having to encode it using **tseqblock**s. These two different versions lend themselves to varying opportunities for optimizations. This brings us to unoptimized and optimized. The difference between unoptimized and optimized is the application of the algorithms that perform optimizations (Algorithm 3 and 4). The unoptimized version does not apply Algorithms 3 and 4, but the optimized version does. Table IV shows the results. For example, the case study presented in ASM Spec. 11 is the strictly constrained version. Alternatively, the partial ASM Spec. 12 shows a snippet of the loosely constrained version that uses **nseqblock** instead of **tseqblock**. We highlight the results in Table IV for this case study. In Table V, we compare the optimized version of the benchmark against versions that are unoptimized for both strictly and loosely timed, and those that are implemented using Handel-C. Notice that Table V shows the reduction in LUTs, FFs and total area.

Table IV also presents synthesis results from Handel-C. We compare with Handel-C because it supports the specification of parallelism and timing. However, the timing model supported by Handel-C is that of strictly constrained only. Hence, the entries for the loosely constrained section of the table does not contain results. We manually translate our benchmark designs into Handel-C specifications such that the parallelism and timing requirements are the same between the optimized version and the Handel-C specification. We validate the correctness of the Handel-C implementations through simulation and synthesis to the Altera DE2 FPGA. We notice that the clock frequency of designs synthesized using Handel-C are higher than when synthesizing designs with synASM. By investigating the resulting design, we attribute the primary reason to be the aggressive optimizations such as automatic pipelining that Handel-C does, and synASM does not support at the moment.

The top half of the table represents synthesis from strictly constrained designs, and the lower half represents synthesis from loosely constrained designs. For each version, we report LUTs, FFs and total area usage, and frequency for optimized and unoptimized circuits. We do not report on the latency of each benchmark because of its minute execution times. As mentioned earlier, the optimized circuits undergo optimizations presented in Algorithms 3 and 4. The loosely constrained circuits have a lower LUT count for most of the benchmarks.

This is because **nseqblock**s in these benchmarks allow the optimizations to take effect by scheduling operations earlier or later within the range (number of cycles in which the block must complete) specified in cycles. This is different for strictly constrained circuits because **tseqblock** explicitly dictates that the outputs of the operations must be available at specific cycles. This shows that coarse-grained timing requirements provide better opportunities for resource sharing. Notice that the sequential FIR shows similar LUT usage for both strictly and loosely constrained circuits. This is because the benchmark makes no use of the parallel constructs; hence, there is no scope for optimizations by either algorithm.

Although sharing functional units reduces overall design LUT usage, we notice a decrease in the maximum clock frequency for most benchmarks. For example, the differential equation solver case study for the loosely constrained version drops by approximately 10Mhz. This decrease is due to additional multiplexors at the inputs of each unit, which lengthens the critical path. The exception is the parallel FIR design for the strictly constrained version. This is because the strictly constrained version contains a combinational path featuring an adder being driven by a multiplier. In the loosely constrained version, the outputs of the adder and multiplier are registered; thus, reducing register-to-register critical path. There are also more flip-flops in the loosely constrained version for some examples. These flip-flops are inserted by the optimization in Algorithm 3.

Furthermore, we also synthesize circuits with consistency check logic, and present the respective synthesis results in the CC logic section of table IV. CC refers to designs that have the consistency check enabled. Note that the generation of consistency check logic can be disabled. The consistency check logic is purely combinational; therefore, it only increases the LUT usage. In addition, the LUT usage incurred by consistency check is same for both optimized and unoptimized designs. The optimization does not impact either the timing or values of state updates, but only when the computation is performed. The consistency check only evaluates the timing and values of the state updates. The LUT usage measurement includes logic cells for the ASM scheduler implementation. Overall, we are able to reduce design LUT usage when compared to the unoptimized version. For some designs we saw a reduction of approximately 50%. The percentage improvement is evaluated without the CC logic, as the consistency check may be disabled later in the design cycle.

We observe from Table V that there is a positive reduction in LUT usage for most designs. The benchmarks that show an increase in FFs (by having a negative reduction) increase the number of FFs because of the optimization in Algorithm 3 that pre-computes and stores the result. We also document the total area reduction by summing up the LUTs and FFs. From our experiments, we experience a total area reduction ranging from 3% to 34% with the exception of Blowfish that shows no reductions. This is because the Blowfish benchmark is primarily sequential. Hence, there is no room for leveraging the parallelism to optimize the circuit any further.

X. Discussion and Future Work

We currently identify five limitations with the synASM framework, which provide opportunities for future work. The first limitation is that synASM maps the state of an ASM into flip-flops on the FPGA. This means that synASM does not map state elements onto any of the other memories that are available on the platform. The second limitation is that we require every operation to take one cycle. This means that synASM does not support automatic synthesis of multicycle operations and automatic pipelining. We require the designer to explicitly pipeline and split long operations into multiple cycles. The third limitation is that we do not support synthesis of floating point operations. This prevents us from performing particular types of operations. The fourth limitation is the support of scheduling algorithms that optimize over different metrics. For now, we only support TFDS, which focuses on maximizing resource sharing. However, it is often important to optimize over other constraints such as frequency, and latency. The fifth limitation is that we require the designer to specify using ASMs. This prevents the use of legacy C and C++ code with the existing framework.

Our plan for the future is to address the above limitations. We are currently exploring methods to automatically partition the state specified in an ASM, and perform memory scheduling that incorporate the explicit parallelism and timing requirements. We are also looking at supporting multicycle operations. However, attempting to perform an operation in multiple cycles that is enclosed in **tseqblock** is non-trivial. We must ensure that with multicycle operations can still preserve the timing requirements and program order constraints in the specification. Hence, we have to appropriately identify opportunities for making operations execute over multiple cycles. We have ongoing work on building a data-type library that supports floating point numbers. In addition, we are investigating the use of high-level data structures such as sets, lists, and tuples for synthesis. To provide a flexible scheduling back-end, we are looking to incorporate system of difference constraints (SDC) scheduling [30] as an alternate scheduling that allows the designer to adjust their constraints. We are currently implementing a translator that converts a C-like language to ASMs. Other avenues of future work include designing an automatic type inference system for hardware data-types, static analysis techniques to conservatively determine state conflicts within parallel and timed constructs, and integrating an automated testbench generator.

XI. Conclusion

This paper proposes synASM, a high-level synthesis framework based on ASMs that focuses on the specification, synthesis, and scheduling of parallel and timed constructs. The contributions to specification, and scheduling of the parallel and timed constructs is independent of the actual language of choice. We select ASMs for its precise, and formal semantics, but the proposed techniques for parallel and timed constructs can be extended to other languages. In this work, we describe synASM's core constructs, and its synthesis to hardware. We present extensions to the force-directed scheduling algorithm

to incorporate explicit parallel constructs, timed constructs, and their compositions in order to improve resource sharing of the resulting hardware. To do this, we augment the definition of CDFGs, redefine mobility, and we present two optimizations that utilize resources across parallel computations to assist in making better scheduling decisions. We call the resulting scheduling algorithm timed FDS. Our experiments on a variety of example designs show that by allowing the designers to specify the inherent parallelism available in the application, there are resource savings to be made. We show savings up to 50% on LUTs and 34% on total area on some of our designs.

REFERENCES

[1] A. E. Abdallah and J. Hawkins, "Formal behavioural synthesis of handel-c parallel hardware implementations from functional specifications," in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9 - Volume 9*, ser. HICSS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 278.1–.

[2] Mentor Graphics, "Handel-C High-level Synthesis." [Online]. Available: "http://www.mentor.com/products/fpga/handel-c/"

[3] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, *AutoPilot: a platform-based ESL synthesis system*. Springer, 2008, pp. 99–112.

[4] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark : A high-level synthesis framework for applying parallelizing compiler transformations," *International Conference on VLSI Design*, vol. 0, p. 461, 2003.

[5] 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 International Symposium on Field Programmable Gate Arrays (FPGA)*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36.

[6] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 18–25, july-aug. 2009.

[7] S. A. Edwards, "The challenges of synthesizing hardware from c-like languages," *IEEE Des. Test*, vol. 23, no. 5, pp. 375–386, Sep. 2006.

[8] S. Singh and D. J. Greaves, "Kiwi: Synthesis of fpga circuits from parallel programs," in *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 3–12. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2008.46

[9] S. Kundu, S. Lerner, and R. K. Gupta, "Translation validation of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 4, pp. 566–579, Apr. 2010.

[10] E. Börger and R. Stärk, *Abstract State Machines – A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[11] K. Winter, "Model checking for abstract state machines," *Journal of Universal Computer Science*, vol. 3, pp. 689–701, 1997.

[12] M. Spielmann, "Automatic verification of abstract state machines," in *Proceedings of the 11th International Conference on Computer Aided Verification*, ser. CAV '99. London, UK, UK: Springer-Verlag, 1999, pp. 431–442.

[13] M. Veanes, N. Bjørner, Y. Gurevich, and W. Schulte, "Symbolic bounded model checking of abstract state machines," *International Journal Software and Informatics*, vol. 3, no. 2-3, pp. 149–170, 2009.

[14] J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, and W. Mueller, "The simulation semantics of systemc," in *Proceedings of IEEE Design, Automation and Test in Europe*, ser. DATE '01. Piscataway, NJ, USA: IEEE Press, 2001, pp. 64–70.

[15] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating finite state machines from abstract state machines," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 112–122.

[16] A. Cavarra, "Data flow analysis and testing of abstract state machines," in *Abstract State Machines, B and Z*, ser. Lecture Notes in Computer Science, E. Barger, M. Butler, J. Bowen, and P. Boca, Eds. Springer Berlin / Heidelberg, 2008, vol. 5238, pp. 85–97.

[17] H. D. Patel and S. K. Shukla, "Model-driven validation of systemc designs," in *Proceedings of the 44th annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 29–34.

[18] R. Sinha and H. D. Patel, "Abstract state machines as an intermediate representation for high-level synthesis," in *Proceedings of IEEE Design, Automation Test in Europe Conference Exhibition*, 3 2011, pp. 1–6.

[19] ——, "Extending force-directed scheduling with explicit parallel and timed constructs for high-level synthesis," in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 214–217.

[20] Mentor Graphics, "Handel-C High-level Synthesis Manual." [Online]. Available: "http://www.mentor.com/products/fpga/handel-c/upload/handelc-reference.pdf"

[21] Forte Design Systems, "Cynthesizer and high-level design." [Online]. Available: "http://www.forteds.com/highlevelsynthesis/"

[22] D. L. Rosenband and Arvind, "Modular scheduling of guarded atomic actions," in *Proceedings of the 41st annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: ACM, 2004, pp. 55–60.

[23] J. Hoe, "Operation-centric hardware description and synthesis," *proceedings of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 9, pp. 1277–1288, 2004.

[24] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-level synthesis: introduction to chip and system design*. Norwell, MA, USA: Kluwer Academic Publishers, 1992.

[25] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, 1st ed. McGraw-Hill Higher Education, 1994.

[26] P. Paulin and J. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, jun 1989.

[27] R. Farahbod, V. Gervasi, and U. Glasser, "CoreASM: An extensible ASM execution engine," *Fundamenta Informaticae*, vol. 77, no. 1, pp. 71–103, 2007.

[28] R. Namballa, N. Ranganathan, and A. Ejnioui, "Control and data flow graph extraction for high-level synthesis," *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, p. 187, 2004.

[29] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[30] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *Proceedings of the 43rd annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: ACM, 2006, pp. 433–438.

Rohit Sinha received his BASc degree in Computer Engineering from the University of Waterloo, Waterloo, Canada in 2011. He is currently pursuing his Ph.D. at the University of California, Berkeley in department of Electrical Engineering and Computer Engineering. His research focus is centred around applying automated formal methods to problems in computer security, embedded systems, and hardware verification.

Hiren D. Patel received his Ph.D. degree in Computer Engineering from Virginia Polytechnic Institute and State University (Virginia Tech), Blacksburg in 2007. He was then a post doctoral fellow at U. C. Berkeley from 2007 to 2009. He is currently an assistant professor in the Electrical and Computer Engineering department at the University of Waterloo, Waterloo, Canada. His research interests are in models of computation, real-time embedded systems, and computer architecture.