Abstract State Machines as an Intermediate Representation for High-level Synthesis

Rohit Sinha
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
rsinha@uwaterloo.ca

Hiren D. Patel
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
hdpatel@uwaterloo.ca

***Abstract*—This work presents a high-level synthesis methodology that uses the abstract state machines (ASMs) formalism as an intermediate representation (IR). We perform scheduling and allocation on this IR, and generate synthesizable VHDL. We have the following advantages when using ASMs as an IR: 1) it allows the specification of both sequential and parallel computation, 2) it supports an extension of a clean timing model based on an interpretation of the sequential semantics, and 3) it has well-defined formal semantics, which allows the integration of formal methods into the methodology. While we specify our designs using ASMs, we do not mandate this. Instead, one can create translators that convert the algorithmic specifications from C-like languages into their equivalent ASM specifications. This makes the hardware synthesis transparent to the designer. We experiment our methodology with examples of a FIR, microprocessor, and an edge detecteor. We synthesize these designs and validate our designs on an FPGA.**

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 PICO [1], Handel-C [2], AutoPilot [3], SPARK [4], Catapult C, and SynphonyC.

Although C-like languages are the preferred language for algorithmic specifications, they have also been criticized as not being suitable for hardware designs [5], [6]. 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, which is equivalent to automatic software parallelization [7].

The second criticism is that C-like languages do not provide mechanisms to control the timing behaviour [6] 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. 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 leveraged.

In response to these criticisms, there are several efforts that extend C-like languages with constructs to express parallel computation [2], [7], timing models [2] and the integration of formal methods [8]. For example, Handel-C extends C with a `par` construct for explicit parallelism, and Kiwi [7] leverages the concurrency mechanism in .NET for parallel specifications. These constructs allow the designer to explicitly make space and time tradeoffs, and 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. Many other C-based HLS methodologies do not provide a timing model. In addition, seamless integration with formal methods also largely remains absent.

We find that it is essential for an HLS methodology to address the above three criticisms for its further success. Given that C-like languages are the language of choice for algorithmic specification, language extensions to allow parallel specifications and control over timing are becoming more prevalent [7]. As a result, our objective with this work is to propose an intermediate representation (IR) that incorporates and supports such extensions. In particular, we want to 1) support the specification of sequential and parallel computation, 2) introduce a timing model, and 3) integrate formal methods into our HLS methodology. Our HLS methodology uses abstract state machines (ASMs) as an IR to accomplish these objectives. Note that we do not require the initial algorithmic specification to be described in ASMs. Instead, translators can convert the algorithmic specification into ASMs, which we can then analyze, optimize and synthesize into hardware. This makes the ASMs transparent to the designers.

ASMs are a form of concurrent state machines that are straightforward for both software and hardware engineers to understand. They also have formal operational semantics. This means that all ASM specifications are executable, which

978-3-9810801-7-9/DATE11/©2011 EDAA

is necessary for simulation. Furthermore, we can leverage existing tools and methods to incorporate formal methods, verification and automated testbench generation to the HLS methodology. ASMs allow specifying both sequential and parallel computation with clear definition of their composition. This is necessary for hardware designs because the IR can naturally represent the parallel computation to be performed in hardware. In addition, the execution semantics of ASMs ensure that race conditions on shared states for parallel computation blocks caused by simultaneous conflicting updates are identified.

A. Main Contributions

The main contributions of this work are listed below:

- An ASM-based IR for a HLS methodology that supports the specification of parallel computation, and a timing model.
- A synthesis back-end that generates synthesizable VHDL from ASMs.

II. Related Work

In recent years, we have seen a surged interest in HLS methodologies [5] 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 [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 timing model is simple: each assignment synthesizes to a cycle in hardware. The abilities to specify parallel computation, and the timing behaviour are important characteristics for a hardware design.

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 [7], 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. In addition, both Kiwi and Handel-C provide no evidence that formal methods can be easily integrated.

The Program-In, Chip-Out (PICO) [1] system presents a framework based on the Kahn process network (KPN) formalism, and an architecture template. The KPN semantics allows the compiler to represent parallel behaviour as a set of processes implementing a sequential algorithm interconnected with unbounded FIFOs. PICO also implements facilities for testbench generation, and software generation. However, one of the fundamental challenges in generating hardware from KPNs is bounding the size of the infinite FIFOs between processes. Currently, PICO requires the user to define constraints on each of the FIFOs, which is similar to defining the bounds on memory usage.

AutoPilot [3] and Forte's SystemC Cynthesizer 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. Once again, there is limited integration of formal methods and verification into their methodologies.

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:

if *Guard* then *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 [9] 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 create our HLS tool. In particular, we make minor extensions to the language, and we implement a back-end that generates synthesizable VHDL from ASM specifications.

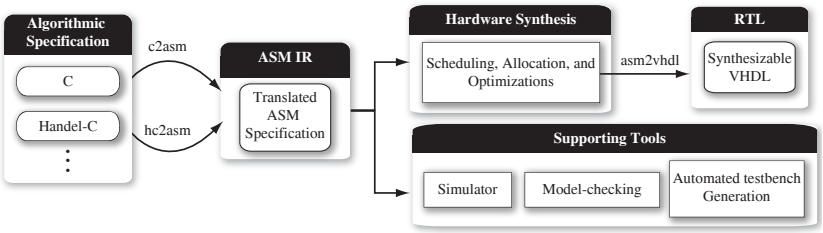

Fig. 1. Design Flow of the Proposed HLS Methodology

IV. Design flow using CoreASM

Figure 1 shows our HLS methodology. The first stage in our methodology involves the algorithmic specification in either a C-like language or in ASMs. While translators can automatically convert the specifications from C-like languages into the ASM IR, we currently write our specifications directly using ASMs. We extend the CoreASM [9] framework with the synthesis back-end that performs scheduling and allocation, and generates synthesizable VHDL. We also make extensions to the language to support timed sequential blocks, which provide a timing model for the synthesis.

V. Synthesis from ASMs

Our back-end supports the core constructs of ASMs as implemented by CoreASM [9]. We include support for basic datatypes such as Boolean, numbers, enumerations and function elements. We are currently implementing datatypes to support bit-wise arithmetic as well.

Statements	Syntax
Block	**par** $statement_1$ ... $statement_n$ **endpar** **seqblock** $statement_1$ ... $statement_n$ **endseqblock** **tseqblock** $statement_1$ ... $statement_n$ **endtseqblock**
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 : $statement_n$
	}

TABLE I
Synthesizable Statements

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, which 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)}.

ASM Spec. 2 Example of **par** Block Statement.

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

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. Several discrete filter designs and image processing algorithms leverage replication to maximize throughput. Our example computes the median filter for each pixel in an input image of size 640 by 480.

ASM Spec. 3 Example of the **forall** Statement.

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

Our synthesis of the **forall** statement essentially performs loop unrolling such that the median operation for each pixel occurs in parallel. We take the 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. This is useful in SIMD type hardware designs. 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 a top-level design and 2) allowing component reuse across design entities. The invocation of `medfilt(...)` is an example of macro rule call. The computation inside the macro rule is scheduled as any other statement. The type of 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, at the end of an ASM step, the sequential block yields {(x,2), (y,1)} as the *Updates* set.

A sequential specification such as the above 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, Force Directed Scheduling, etc. that we support with the sequential block statement.

ASM Spec. 4 Example of **seqblock** Statement.

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

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 unless embedded with a parallel statement. Consider ASM Spec. 4 with **tseqblock** instead of **seqblock**. 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) 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 this definition for a cycle-accurate ASM Spec. 5. 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 in different clock cycles.

ASM Spec. 5 Example of **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
```

9) 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. 5 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.

10) 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.

ASM Spec. 6 Example for Temporary Registers

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

For our example in ASM Spec. 6, the end of a step happens when 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 its value 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

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
```

`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.

Different parallel statements can update the same state (line 5 and 8 in ASM Spec. 6). 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. 6 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 executes results

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)

TABLE II
EXECUTION WITH CONSISTENCY CHECK

in conflicting updates for state `y`. We identify this by the 2 asserted bits in `check_y` bit vector. Our implementation for identifying multiple set bits is to determine if `check_y` bit vector is a power of two as follows: `conflict = not ( (check_y & check_y - 1) = "00").`

VI. EXPERIMENTS

We present a case study of an 8-tap FIR design using our HLS methodology. ASM Spec. 7 shows the ASM IR. The function declarations from line 1 to line 5 define the states of the hardware design. The `Tap` rule defines the operation of the filter, and the `ShiftRightOne` shifts the input `window` right by one in parallel. The `DoFIR` rule performs the actual FIR operation by first performing the shift, which is followed by eight tap operations in parallel.

ASM Spec. 7 A Parallel Specification of FIR

```
1  function i : → UNSIGN_NUMBER8
2  function z : → UNSIGN_NUMBER8
3  function y : UNSIGN_NUMBER8 → SIGN_NUMBER32
4  function window : UNSIGN_NUMBER8 → SIGN_NUMBER32
5  function weights : UNSIGN_NUMBER8 → SIGN_NUMBER32
6  derived weights_sz = 8
7  derived window_sz = 8
8
9  rule Tap(a, x, yv, o) = {
10     o:= x * a + yv
11 }
12
13 rule ShiftRightOne(firstInput) = {
14     forall i in [1 .. window_sz] do {
15         window(i + 1) := window(i)
16     }
17     window(1) := firstInput
18 }
19
20 rule DoFIR(newInput) = {
21    tseqblock
22     ShiftRightOne(newInput)
23     forall z in [1 .. weights_sz] do {
24         Tap(weights(z), window(z), y(z), y(z))
25     }
26    endtseqblock
27 }
```

Figure 2 shows a block diagram of the hardware generated using our HLS methodology. Notice that the **forall** statements in lines 14 and 23 generate parallel hardware. The **tseqblock** mandates that the shift operation occurs in a cycle prior to the tap computation.

VII. RESULTS

We target our example designs onto an Altera DE2 FPGA. Table III shows the results. CC refers to designs that have the consistency check enabled. We also present results of the generated hardware when we disable the CC. The area measurement includes logic cells for the ASM scheduler implementation.

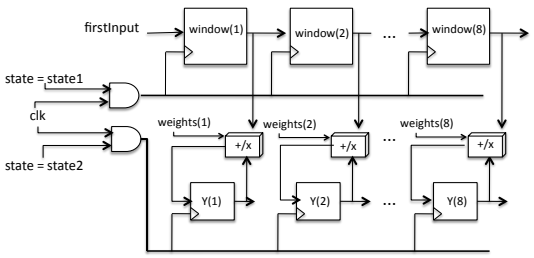

Fig. 2. Block Diagram of Generated Hardware for FIR

Designs	With CC			Without CC		
	LUTS	FFs	MHz	LUTS	FFs	MHz
Microprocessor	7771	6303	98.1	180	101	121.4
SequentialFIR	17,752	12,257	83.7	293	153	91.4
ParallelFIR	243	139	88.1	168	90	101.3
FourPlaceQueens	224	131	146.8	87	48	154.1
Loop Unroll	123	58	163.1	105	54	163.1

TABLE III
SYNTHESIS RESULTS OF EXAMPLE DESIGNS

VIII. CONCLUSION

In this paper, we propose using ASMs as an IR for the specification of parallel computation, an extension of the sequential semantics for a timing model, and incorporating formal methods into the methodology. We describe the core constructs, and its synthesis to hardware. We evaluate our methodology with a variety of examples. Currently, we are investigating optimization opportunities across multiple parallel block statements, and the inclusion of sets and lists as data structures for synthesis.

REFERENCES

[1] V. Kathail, S. Aditya, R. Schreiber, B. Rau, D. Cronquist, and M. Sivaraman, "PICO: automatically designing custom computers," *Computer*, pp. 39–47, 2002.

[2] Mentor Graphics, "Handel-c high-level synthesis," 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," *High-Level Synthesis*, pp. 99–112, 2008.

[4] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations," 2003.

[5] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Design & Test of Computers*, pp. 18–25, 2009.

[6] S. Edwards, "The challenges of synthesizing hardware from C-like languages," *IEEE Design and Test of Computers*, vol. 23, no. 5, pp. 375–386, 2006.

[7] S. Singh and D. Greaves, "Kiwi: Synthesis of FPGA circuits from parallel programs," in *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, 2008, pp. 3–12.

[8] 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, 2010.

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