Automatic Generation of Architecture-Level Models from RTL Designs for Processors and Accelerators

Yu Zeng, Aarti Gupta, Sharad Malik
Princeton University, Princeton, USA
yuzeng@princeton.edu, aartig@cs.princeton.edu, sharad@princeton.edu

***Abstract*—Hardware platforms comprise general-purpose processors and application-specific accelerators. Unlike processors, application-specific accelerators often do not have clearly specified architecture-level models/specifications (the instruction set architecture or ISA). This poses challenges to the development and verification/validation of firmware/software for these accelerators. Manually writing architecture-level models takes great effort and is error-prone. When Register-Transfer Level (RTL) designs are available, they can be a source from which to automatically derive the architecture-level models. In this work, we propose an approach for automatically generating architecture-level models for processors as well as accelerators from their RTL designs. In previous work we showed how to automatically extract the architectural state variables (ASVs) from RTL designs. (These are the state variables that are persistent across instructions.) In this work we present an algorithm for generating the update functions of the model: how the ASVs and outputs are updated by each instruction. Experiments on several processors and accelerators demonstrate that our approach can cover a wide range of hardware features and generate high-quality architecture-level models within reasonable time.**

***Index Terms*—Hardware modelling, accelerators, architectural abstraction**

I. Introduction

General-purpose processors have an architecture-level model in the form of an instruction set architecture (ISA) [1], [2] which serves as the software/hardware interface for firmware/software development and verification/validation. With Moore's Law slowing down, application-specific accelerators are increasingly being used to meet the power-performance requirements of emerging applications. Recent work has presented the Instruction-Level Abstraction (ILA) as the software/hardware interface for accelerators [3]. The ILA generalizes the notion of ISAs to include accelerators.

Processor ISAs are mostly manually written by hardware experts [1], [2]. Although this takes great effort, this is acceptable since processor ISAs are widely used and have been stable for many years. Similarly, the ILAs are largely manually written. (While there is some work on synthesizing ILAs from templates, manual effort is needed in defining the templates [4].) This may be a significant burden for accelerators which may not have the volume or life-span compared to processors. When the Register-Transfer Level (RTL) implementation of a hardware design is available, it can be used to extract the architecture-level model. (We use "ISA/ILA" and "architecture-level model" interchangeably.) The extracted model is consistent with the RTL by construction, and thus can be used with high confidence for software development and verification/validation for processors/accelerators.

In this work, we propose an automated approach for extracting architecture-level models from RTL designs of processors and accelerators. While this work primarily targets accelerators, some legacy processors/micro-controllers may also have no architecture-level model available, and they can benefit from our approach. We build on our previous work on extracting the architecture-state variables (ASVs) from RTL designs [5]. Given the ASVs, the remaining task in determining the architecture-level model is the extraction of the state update functions: how the ASVs and outputs are updated by each instruction. We present an algorithm for generating the update functions using the LLVM [6] framework. The inputs to our algorithm are the instruction encoding and any constraints for issuing these instructions. This information is necessary for correctly using the hardware and thus available in its user manual. We demonstrate the efficacy of our approach through five non-trivial designs: three processors and two accelerators, sourced from the OpenCores website or Github. (Our open-source tool is available at: https://github.com/yuzeng2333/autoGenILA.)

The contributions of this work are as follows:

- To the best of our knowledge, this is the first work that automatically generates architecture-level models for both general-purpose processors and accelerators.
- We propose an approach using the LLVM framework for generating and simplifying the state update functions.
- We provide an experimental evaluation of our approach with five non-trivial designs. We show that our approach can deal with a wide range of hardware features (pipelines, hierarchy/reuse, internal data buffers, etc.) and the generated high-quality architecture-level models show much faster simulation speed than RTL-based simulation.

II. Preliminaries

A. Architecture-State Variables

For a processor/accelerator, the architecture-state variables (ASVs) [5] are the variables which are persistent across instructions, i.e., they determine the results of all future instructions. Thus, ASVs are the only state variables that are needed in the architecture-level model. For example, ASVs for a general purpose processor such as RISC-V include the program counter, general-purpose registers (GPR), the addressable memory, etc.

B. Architecture-Level Models

We assume that the encoding for the valid instructions of our target hardware are already given. For a given instruction set $Inst := \{inst_i\}$, we define the architecture-level model as the tuple: $\mathcal{A} = \langle S, Init, I, O, D, U \rangle$, where

- S is a vector of architecture-state variables (ASVs) with space $\mathbb{S}$,
- $Init$ is a vector of initial values for the variables in S,
- I is a vector of input variables with space $\mathbb{I}$,
- O is a vector of output variables with space $\mathbb{O}$,
- $D := \mathbb{I} \rightarrow Inst$, is a decoding function that maps an input to a specific instruction,
- $U := \{U_i : (\mathbb{S} \times \mathbb{I}) \rightarrow \langle \mathbb{S}, \mathbb{O} \rangle\}$ is the set of update functions to the ASVs and outputs. Each instruction $inst_i \in Inst$ has a corresponding update function U_i.

The architecture-level model specifies a transition system in the usual way: all variables in S are initialized with values in $Init$, then for input I, the decode function determines which instruction it is, and the corresponding update function in U updates S and O.

To generate the architecture-level models, the components in the tuple need to be determined automatically. S is often provided in the user manual of the hardware. Further, our previous work [5] has provided methods for automated extraction of ASVs from RTL designs. $Init$ is usually the vector of reset values, which can be easily determined using Yosys [7] or RTL simulation. I and O are already identified in the RTL design. The instruction encoding needed for the decode function is provided as part of the hardware user manual. Further, the automated generation of decoding functions has also been extensively studied [8]. For hardware with RISC-like instructions, decoding is usually simple. However, to the best of our knowledge, there has never been any previous work towards automatic generation of the update functions U. This is the focus of our work.

C. LLVM and Optimization Passes

LLVM [6] is a modular and reusable compiler framework widely used by industry and academia. The IR of LLVM is a Static Single Assignment (SSA) based representation that supports a wide range of programming languages. It supports integers of any width. In our work, we use the LLVM IR to represent the generated update functions. To optimize programs for performance and code size, many optimization passes are provided by LLVM, including dead code elimination, instruction combination, etc. In our approach, the passes are used to abstract the RTL model by eliminating details not relevant to the architecture-level model.

D. Scope of this Work

Our approach targets both processors and accelerators. For accelerators, we focus on Loosely-Coupled Accelerators (LCA) [9], which use Memory-Mapped Input/Output interfaces and are widely used in today's SoCs. LCAs share an important property with processors: they have a clear definition of when instructions are valid. Further, we focus on RTL designs with a single clock domain. Designs with multiple clock domains of different frequencies require alignment of events that happen in different domains, which is currently not supported, but this work could be extended in that direction.

III. Basic Update Function Generation Algorithm

In this section, we give the basic algorithm for automatically generating the update functions from RTL designs. This needs to address three major challenges:

- **Removing the Clock.** RTL designs are cycle-accurate. Clock signals and related behavior should be removed in the update functions.
- **Separating Operations.** The RTL design implements the functionality of all instructions, while an update function only specifies the operations of a single instruction. These operations need to be separated from others.
- **Eliminating Details.** The RTL design contains micro-architecture level or implementation details, which should be eliminated in the architecture-level update functions.

Previous works [10] [11] on automatically converting RTL designs to higher-level models largely address the first challenge. These works analyze the conditions for state transitions in the Extended-FSM from the RTL and merge the states to simplify the model. However, it is difficult to address the second and third challenges above by analyzing the state transition conditions. *Instead, we convert the task of update function generation to a program simplification problem and solve it using powerful LLVM optimization passes.*

A. Converting RTL Statements to Update Functions

The execution of an instruction in RTL usually takes multiple clock cycles to finish. In each of these cycles, some registers get updated with new values as intermediate steps in the whole instruction. Therefore, to get the update function that spans multiple cycles, we can unroll the RTL design and merge the intermediate steps across the execution cycles. This converts the operations that occur in multiple clock cycles to a sequence of steps that constitute the update functions.

1) Extracting Update Functions: A Simple Example

We illustrate this idea with a simple example before presenting the pseudo-code for our algorithm. The RTL design in Listing 1 counts the number of non-zero bits in the register `word`. It has two instructions: for instruction I (Inst.I), the input `optn` (abbreviation for "option") has value 1 and register `word` is assigned the value of `dataIn`; in instruction II (Inst.II) where `optn` has value 2, it counts how many non-zero bits there are in `word` and stores the result in `result`. For the architecture-level model of this design, D (the decoding function) is straightforward, S (the set of ASVs) consists of `word` and `result`, whereas `counter` is a temporary variable. We now need to generate the update functions to complete the architecture-level model.

```
module number_of_positive_bits (
  input clk,            input [1:0]  optn,
  input [1:0] dataIn,
  output reg [1:0] result
);
reg [1:0] word; reg [1:0] counter = 0;
wire word_nxt    = optn==1    ? dataIn:word;
wire counter_nxt= optn==2    ? 2 :
                  counter>0 ? counter-1:0;
wire result_nxt = optn==2    ? 0 :
```

```
11                      counter>0 ?
12                      result+word[counter-1] :
13                      result;
14 always @(posedge clk)
15   {result, word, counter}
16     <= {result_nxt, word_nxt, counter_nxt};
```

Listing 1: Verilog code for a two-instruction module that counts the non-zero bits in a word

Consider the derivation of the update function for `result` by Inst.II. Since `word` has only 2 bits, Inst.II ends after 3 cycles. Starting from the statement of `result` (line 15 & 16), we unroll the RTL backwards for 3 cycles and merge the register updates, i.e., replace variables in the update expressions with their own update expressions. Fig. 1 shows the unrolling order starting from cycle 3, and Listing 2 shows all the unrolled statements. Note that in the unrolling, a variable in the code may be instantiated in multiple clock cycles. We use the suffix "`_#c`" to denote a variable's instantiation in cycle `c`. In Fig. 1, "assign1 (...)" represents the assignment to `result_#3`, which depends on the variables in cycle 2, as shown in the line that follows. Each register has its own assignment, and the variables that it depends on are circled with the same color as its assignment.

We continue the unrolling for registers, but not for inputs since their values are provided by the external environment. Since we are interested in only the update function of this instruction, we assume that the inputs following the instruction of interest are equivalent to a NOP instruction, and do not interfere with the execution of the instruction of interest. We assume such NOP instructions are available. The unrolling ends at variables of cycle 0. From these unrolled statements, we can see that the value of `result_#3` can be determined by the variables at cycle 0, together with inputs (`optn`, `dataIn`) at all other cycles.

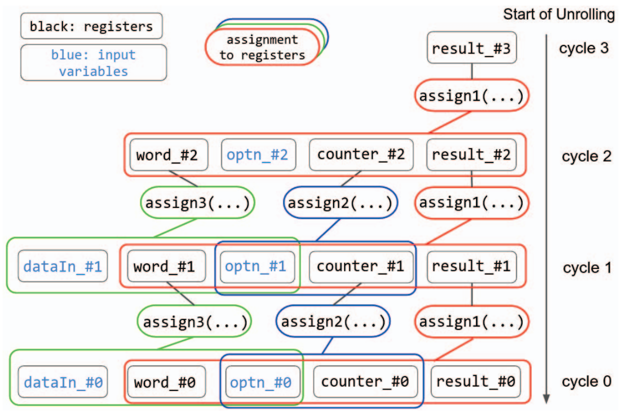

Fig. 1: Unrolling backwards starting from result_#3

```
result_#3 = optn_#2==2 ? 0 : counter_#2>0 ?
            result_#2+word_#2[counter_#2-1]
            : result_#2
counter_#2= optn_#1==2 ? 2 : counter_#1>0 ?
            counter_#1-1 : 0
result_#2 = optn_#1==2 ? 0 : counter_#1>0 ?
            result_#1+word_#1[counter_#1-1]
            : result_#1
word_#2   = optn_#1==1 ? dataIn_#1 : word_#1
counter_#1= optn_#0==2 ? 2 : counter_#0>0 ?
            counter_#0 - 1 : 0
result_#1 = optn_#0==2 ? 0 : counter_#0>0 ?
            result_#0+word_#0[counter_#0-1]
            : result_#0
word_#1   = optn_#0==1 ? dataIn_#0 : word_#0
```

Listing 2: Assignment to the registers in each cycle by unrolling

The unrolled statements in Listing 2 show the assignment to `result_#3` for all possible values of the variables it depends on. However, we are only interested in the update by Inst.II. To obtain the specific update function, we can replace the input `optn_#0` with the value "2" for Inst.II. As discussed earlier, `optn_#1` and `optn_#2` will be 0, which correspond to NOP instructions for this module. With these constants, these statements can be further simplified by applying constant folding, as shown below in the Listing 3.

```
counter_#1 = 2
result_#1  = 0
word_#1    = word_#0
counter_#2 = counter_#1 - 1 = 1
result_#2  = 0 + word_#1[1] = word_#1[1]
word_#2    = word_#1
result_#3  = result_#2 + word_#2[0]
           = word_#0[1] + word_#0[0]
```

Listing 3: Simplified assignment to registers

The last statement in Listing 3 is the update function of interest. This only takes `word_#0` as its argument, which is an ASV assigned by Inst.I. Further, the non-ASV register `counter` is eliminated during simplification. This is consistent with our expectation since the use of `counter` is an implementation detail. So the third challenge, "Eliminating Details" is addressed here with program simplification. In the above, we also successfully separated the operations for Inst.II from those of Inst.I by restricting the unrolling to the cone of influence of the operations of Inst.II. This shows that simplifying unrolled statements with input values from one instruction can remove operations for other instructions. So the second challenge, "Separating Operations," is also addressed.

In some cases, the simplified update function may still have non-ASVs among its arguments. These are the implementation/micro-architectural variables in the RTL design. As mentioned, ASVs are sufficient to determine the updated value – the non-ASVs may determine the performance (for example number of cycles taken), but cannot determine the update function. Thus, non-ASV variables may be replaced with explicit values. The values assigned to the non-ASV variables must be those that can be obtained for some sequence of instructions. We use the reset state which meets this requirement for the non-ASV values – essentially we consider the instruction of interest as the first instruction that is executed.

2) Pseudo-Code for the Algorithm

Next, we present the pseudo-code for our algorithm. We use LLVM IR as the translation target from RTL and to represent the update function. Bit-vectors are converted to integers of the same width. To make the translation easier, the original Verilog code is preprocessed by Yosys [7] to generate simplified Verilog code, which uses a subset of Verilog syntax.

Algorithm 1: Generate Update Function

```
Input: target variable $v$, instruction encoding for $Inst$,
       instruction encoding for $NOP$ instruction, upper
       bound of execution delay of $Inst$: bound, initial
       values for non-ASV variables: $INIT$
Output: Update function for $v$ by instruction $Inst$
1  Function Update_Function_Gen(var, cycle):
2      if $cycle > 0$ then
3          if var is input signal then
4              return value for var in $NOP$
5          else
6              parse the statement of var as:
                 $var = operator(arg1, \dots, argN)$
7              cycle_nxt = (var is register) ? $cycle - 1$ : cycle
8              arg1_nxt = Update_Function_Gen(arg1,
                 cycle_nxt)
9              ...
10             argN_nxt = Update_Function_Gen(argN,
                 cycle_nxt)
11             return operator(arg1_nxt, ..., argN_nxt)
12     else
13         if var is input signal then
14             return value for var in $Inst$
15         else if var is a non-ASV then
16             return value for var in $INIT$
17         else
18             return symbolic value for var
19     end
20 Update_Function_Gen(v, bound)
```

Algorithm 1 generates the update function for a target variable v for instruction $Inst$. The instruction encoding for $Inst$ specifies values for each input signal, either explicitly or symbolically. Usually the control signals/opcode are explicit while input data are symbolic. As discussed before, the values of the inputs at cycles after $Inst$ is issued are such that they do not affect the normal execution of $Inst$, i.e., they effectively represent NOP instructions. Here, *bound* is an upper bound on the number of execution cycles for $Inst$, which can be obtained from the user manual. $INIT$ denotes the initial values for non-ASV variables. As discussed, they are the reset state values and can be obtained from simulations.

The top-level call in the algorithm is shown at line 20, which uses the given bound for the instruction of interest. In the recursive function (line 1), the first argument var is a variable in RTL, and the second argument $cycle$ represents the clock cycle for which we are deriving its value. When $cycle$ is greater than 0 (line 2), which means $Inst$ is executing, input variables are assigned values as in the NOP instruction. Otherwise, we recursively call the function for all the operands of the statement for var (line 8 to line 10). If var is a register, $cycle$ is decremented by 1 for the recursive calls on its operands, since the value of the register depends on the operands at the previous cycle. If $cycle$ is 0, i.e., the cycle that $Inst$ is issued, the recursive function calls stop. In this case, if var is an input, it is assigned the corresponding value from the encoding of $Inst$, which can either be explicit or symbolic. If var is non-ASV, it is assigned its value in $INIT$. Otherwise, the symbolic value of the ASV is used.

Many hardware designs can process instructions in parallel, e.g., pipelined processors. Our algorithm works for these designs by enforcing the later-issued inputs to be NOP. The reasoning is that the update function should be *the same regardless of the context of execution*, e.g., whether or not the other instructions are running in parallel. Therefore, we derive the update functions in the most simple context with a single instruction being executed.

B. Simplifying Update Functions with LLVM Optimizations

For complex designs, the update function generated by the algorithm can be complex for a couple of reasons. First, many micro-architecture details may still exist in the generated update function. Second, the recursive function in our algorithm is called on every variable that is used as an operand. However, some of them may not be needed, e.g., if the condition variable for a "? :" operator is always true or false for this instruction, then one branch would never be selected. Both the micro-architecture level details and unused variables can be seen as redundant code. For example, data buffering, a common technique used in hardware for data reuse, can be seen as temporary variables that can be optimized out. We use the powerful optimization passes in LLVM to optimize the update functions to remove such redundant code.

IV. Enhancements to the Basic Algorithm

The following enhancements are provided to deal with specific hardware features.

A. Buffers for Output Variables

Our basic algorithm assumes that the target variables keep their updated value on instruction completion. This is true for ASVs, but for output variables, the assumption may not hold. For example, a sequence of data values could be output on the same data port in multiple cycles. We need the update function for the entire sequence. Our approach to deal with this problem is to connect a buffer (written in Verilog) at the output ports which stores the sequence of output values. The registers in the buffer are then used as our target variables.

B. Dealing with Hierarchy in the RTL

Most designs have hierarchies, where sub-modules may be reused multiple times. We would like to maintain this hierarchy in the update functions to reduce the code size by converting the sub-modules to sub-functions. However, since the values of the inputs to sub-modules are unknown, the sub-functions cannot be simplified separately using constant folding. We solve this problem by simplifying the sub-functions together with the top function with the top-module inputs. This allows constant folding to include the sub-modules.

C. Treating Internal Memories as Global Arrays

Memories may be external to the given RTL design or internal, e.g., buffers/scratchpads. Internal memories are conveniently modeled as global array-type variables in LLVM IR as opposed to modules with a separate addressing mechanism.

V. Implementation and Experiments

We applied our algorithm to three RISC-V processors and two accelerators shown in Table. I. The ASVs for all designs except VTA are determined using the algorithms in our previous work [5]. VTA does not have any register-type ASV – only the internal buffers are its ASVs. The decode functions are simple "switch" operations based on some bit fields in the instruction encoding. The generated update functions, represented as LLVM IR, are simplified with the "O1" optimization level.

Designs	Pico RISCV	Pipelined-RISCV	SuperScalar-RISCV	AES	VTA
# Register ASVs	37	46	61	5	0
# Instructions	34	42	42	5	11
# Lines of Verilog	2014	7526	10049	1111	22999
# Internal Mem Bits	0	0	0	0	3.8 MB

TABLE I: Statistics of RTL designs used in experiments

Table I provides the design statistics. PicoRISCV [12] is a size-optimized RISC-V core that implements the RV32I instruction set. It has no pipelines but uses a seven-state control finite state machine (FSM). It tests if our algorithm can deal with complex FSMs. Pipelined-RISCV [13] is a five-stage pipelined RISC-V core which tests the effectiveness of our algorithm on pipelines. SuperScalar-RISCV [14] is a superscalar (dual-issue) core with a seven-stage pipeline and branch prediction. It tests how our algorithm performs on complex processor designs. "AES" [15] is a cryptographic engine, which implements the Advanced Encryption Standard. It is a highly hierarchical design: with five levels from the top module to the basic sub-modules, and the sub-modules are reused multiple times. It tests if our approach can maintain the hierarchy to reduce code size for the update functions generated. "VTA" [16] is a programmable deep learning accelerator that can perform dense linear algebra operations. It has multiple internal memories, which are used to store the loaded data and weights for neural networks. This tests if our algorithm can handle internal buffers as global arrays.

All the experiments are run on a computer with Intel Core i7-8665U CPU @ 1.9GHZ and 32GB of memory. The experiment results for the designs are listed in Table II. For testing correctness, the C++ based architecture-level models with the LLVM-IR based update functions are all compiled to executable machine code files using Clang. We executed one million randomly generated valid instructions with the architecture-level models, and compared the results with compiled simulations for the RTL designs using Verilator [17]. The simulation results are a complete match. To show the advantage of the architecture-level models, we also compared the simulation time using architecture-level models vs. the Verilator-based RTL simulations. The previous related works [10], [11] are only released with proprietary tools, so we are unable to compare our approach with them.

A. Discussion of Results

PicoRISCV. The generated update functions for PicoRISCV are as simple as its ISA. The FSM-related details are removed successfully. The code size is larger than expected because the bit-select operations, which are not supported by LLVM IR, need to be replaced by combinations of bit-shift and truncate operations. The simulation speed-up is smaller than in other designs because it is a relatively simple core and there is little room for improvement.

Designs	Pico RISCV	Pipelined-RISCV	SuperScalar-RISCV	AES	VTA
# Update Functions	394	363	363	21	18
# Lines On Average	166	1721	7399 (2332)	239	7932
# Max Lines	2293	3877	14786 (4864)	4475	10602
Total Generation (hr:min:sec)	24:24:15	41:33:21	75:24:19 (71:09:13)	00:49:00	04:50:05
Avg Generation (hr:min:sec)	00:03:43	00:06:52	00:12:28 (00:11:46)	00:02:20	00:16:07
Total Simplification (hr:min:sec)	00:37:21	03:39:25	04:48:42 (04:09:03)	00:05:32	00:04:23
Avg Simplification (hr:min:sec)	00:00:06	00:00:36	00:00:48 (00:00:41)	00:00:16	00:00:15
Simulation Speed-Up	4.5	15.2	1.15 (9.26)	27.3	9.41

TABLE II: Experiment results. "# Update Functions" shows the number of update functions generated. "# Lines On Average" shows the average number of lines of LLVM IR in the update functions. "# Max Lines" shows the number of lines in the largest function. "Total (Avg) Generation" shows the total (average) time of function generation. "Total (Avg) Simplification" shows the total (average) time for function simplification using LLVM. "Simulation Speed-Up" shows the speed-up of architecture-level simulation using the generated model compared to RTL simulation using Verilator. For SuperScalar-RISCV, the data outside/inside parentheses are for the configuration with branch prediction enabled/disabled.

Pipelined-RISCV. The generated update functions for the internal ASVs (e.g., program counter) are very close to those of PicoRISCV. However, the update functions for the outputs are quite complex. PicoRISCV is directly connected to a single memory, while Pipelined-RISCV is connected to the instruction/data cache. The update functions contain cache operations such as invalidation which increases the average code size. The simulation speed-up is larger than PicoRISCV, because Pipelined-RISCV has more micro-architecture level features which are eliminated.

SuperScalar-RISCV. This has two configurations – branch prediction enabled or disabled. With the enabled configuration, the update functions are not significantly simplified. For the other two RISCV processors, the simplification mostly comes from constant folding due to the explicit input instruction value, e.g., the decoder is greatly simplified with this. However, with branch prediction, the decoder does not take the instruction value directly – the predicted program counter (PC) for the instruction is first checked for correctness based on some internal registers. Since these internal registers are symbolic, the logic for the decoder cannot be simplified with constant folding. With branch prediction disabled, the generated update functions are significantly simplified and the overall simulation speed-up is higher (9.26 vs. 1.15).

AES. The "# Max Lines" row represents the update function for the encryption operation with 4475 lines of code. Although the encryption operation is significantly more complex than the instructions for RISC-V, its size is just slightly larger. This is because our approach successfully maintains the hierarchy of the design: sub-modules are converted to sub-functions and reused. The AES encryption is implemented with a 21-stage pipeline. The update functions eliminate the intermediate pipeline results which provides a large simulation speed-up.

VTA. VTA has the most complicated update functions – reflected in the maximum number of lines of code among the five designs. In the update functions for VTA, all the internal buffers are converted to global data arrays by our program.

LLVM Passes. We inspected the running time with the "time-passes" option of LLVM-OPT. The "InstCombine" pass takes the longest time in nearly all the optimizations. It combines redundant instructions to form fewer, simple instructions. This is consistent with our earlier observation that the update functions generated from the RTL contain redundant code. Other important passes include "ModuleInlinerWrapper", "IP-SCCP" and "EarlyCSE".

Summary. The update functions are generated for all designs within reasonable time. Except for SuperScalar-RISCV, the simulation is several times faster than Verilator-based simulation, one of the fastest RTL simulators. This provides evidence that the architecture-level models generated in this work provide high-quality abstractions.

VI. Limitations and Future Work

Some designs have a "hardware loop" where the same operations are repeated and the number of execution cycles depends on some ASVs or inputs. For example, the matrix multiplication instruction in VTA uses a "hardware loop" where the number of execution cycles depend on the bits of the instruction which specify the size of the matrices. The most effective representation for the update function is a software loop. This requires a "loop-reroll" optimization pass which detects the repeated operations and converts them to loops. However, the native "loop-reroll" pass in LLVM is quite limited and cannot detect complex loops. Due to this, we constrain the sizes of matrices when generating the update functions for VTA. For future work, we plan to add a customized "loop-reroll" pass.

VII. Related Work

Manually-Specified ISAs. Several domain-specific languages have been used to manually specify ISAs. For example, Sail [1] has been used to specify the ISAs for ARM, RISC-V, etc. The K-framework [2] is used to specify the ISAs for all the user-level instructions of x86-64. However, as the authors admit, manually writing specifications is "a daunting effort" [2]. ILAng [18] has been used to manually model the ISA for both processors and accelerators.

Automatically-Generated ISAs. There has been some work on automatic generation of ISAs. Stratified synthesis [19], used to generate x86 ISA specifications, synthesizes semantically equivalent programs with a base set of instructions to model more complicated instructions. This method does not cover all instructions and is only applicable to processors. Another work [4] uses template-based program synthesis to generate ILA-based architecture-level models. This method requires users to provide the synthesis templates – our approach is fully automatic.

Other Types of Hardware Models. A^{2}T [10] first converts the RTL design to an Extended-FSM, which is then simplified to a Transaction-Level Model (TLM) by merging the states. Another follow-up work [11] further converts the RTL to C++ programs. While both the TLM and C++ programs are higher-level models, they are not architecture-level models.

VIII. Conclusions

In this paper, we present the first work on the automatic generation of architecture-level models from RTL designs for both processors and accelerators. The RTL design is unrolled for each clock cycle of the instruction execution and then converted to LLVM-IR-based programs, which are simplified with the LLVM optimization passes. Experiments on processors and accelerators show that high-quality architecture-level models are generated. These higher-level models can significantly enhance the development and verification of software/firmware running on these designs [2] [18].

References

[1] A. Armstrong, T. Bauereiß, B. Campbell, A. D. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. R. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS," in *Proc. POPL*, 2019.

[2] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Rosu, "A complete formal semantics of x86-64 user-level instruction set architecture," in *Proc. PLDI*, 2019.

[3] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-Level Abstraction: A uniform specification for system-on-chip (SoC) verification," *TODAES*, Dec 2018.

[4] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik, "Template-based synthesis of instruction-level abstractions for SoC verification," in *Proc. FMCAD*, 2015.

[5] Y. Zeng, B.-Y. Huang, H. Zhang, A. Gupta, and S. Malik, "Generating architecture-level abstractions from RTL designs for processors and accelerators, Part I: Determining architectural state variables," in *Proc. ICCAD*, 2021.

[6] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis amp; transformation," in *Proc. CGO*, 2004.

[7] C. Wolf, "Yosys open synthesis suite." https://github.com/YosysHQ/yosys. Accessed: 2021-12-12.

[8] X. Xu, J. Wu, Y. Wang, Z. Yin, and P. Li, "Automatic generation and validation of instruction encoders and decoders," in *Proc. CAV*, 2021.

[9] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich CMPs," in *Proc. DAC*, 2012.

[10] N. Bombieri, F. Fummi, and G. Pravadelli, "Automatic abstraction of RTL IPs into equivalent TLM descriptions," *IEEE Transactions on Computers*, vol. 60, pp. 1730–1743, Dec 2011.

[11] N. Bombieri, F. Fummi, and S. Vinco, "A methodology to recover RTL IP functionality for automatic generation of SW applications," *TODAES*, June 2015.

[12] C. Wolf. https://github.com/YosysHQ/picorv32. Accessed: 2021-12-12.

[13] Ultraembedded, "RISC-V core." https://github.com/ultraembedded/riscv. Accessed: 2021-12-12.

[14] Ultraembedded, "biriscv." https://github.com/ultraembedded/biriscv. Accessed: 2021-12-12.

[15] H. Hsing, "Aes." https://opencores.org/projects/tiny_aes. Accessed: 2021-12-12.

[16] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "A hardware–software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, 2019.

[17] Veripool, "Verilator homepage." https://www.veripool.org/verilator/. Accessed: 2021-12-16.

[18] B.-Y. Huang, H. Zhang, A. Gupta, and S. Malik, "ILAng: A modeling and verification platform for SoCs using instruction-level abstractions," in *Proc. TACAS*, 2019.

[19] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: Automatically learning the x86-64 instruction set," in *Proc. PLDI*, 2016.