Automating Formal Verification of Customized Soft–Processors

Kong Woei Susanto
Department of Computing
Imperial College London
Email: susanto@computer.org

Wayne Luk
Department of Computing
Imperial College London
Email: wl@imperial.ac.uk

***Abstract*—Soft–processors, instruction processors implemented in FPGA technology, are often customizable to support domain-specific optimization. However the correctness of customized soft–processors, executing the associated machine code, is often not obvious. This paper proposes a novel approach for verifying the implementation of an application program for a customized soft–processor, based on the ACL2 theorem prover. The correctness proof involves verifying a machine code program executing on the target hardware device against a high-level specification of the application program. We illustrate the proposed approach with several case studies, showing how processors with different custom instructions and with different number of pipelined stages can be automatically produced and verified; such processors have a range of trade-offs in performance, size, power and energy consumption to meet different requirements.**

I. Introduction

As chip fabrication costs continue to rise, FPGA technology becomes increasingly popular. Advanced FPGAs can now support complex electronic designs; implementing FPGA-based soft–processors is a promising option, since: (a) they can support execution of existing machine code programs with a large code base, (b) existing software development tools can be used for generating and debugging soft–processor machine code, (c) there is no need for place-and-route, which takes longer time as FPGAs become more complex, (d) soft–processor architecture can be customized to one or more machine code programs, to exploit information about the application or the implementation platform; for instance, a soft–processor can be simplified if it does not need to execute all the instructions in a given instruction set, and frequently-occurred instruction fragments can be made into a single custom instruction to avoid multiple instruction fetch and decode.

Because of the above benefits, FPGA vendors have developed soft–processors such as the PicoBlaze and MicroBlaze from Xilinx and the Nios from Altera. Soft–processors are widely used, particularly in embedded systems [18]; they have also been an active topic of research [6],[8],[12],[19],[20]. However, while there has been previous work on verifying instruction processors [9] and reconfigurable cores [17], we are not aware of prior work on verification of soft–processors.

The verification of systems involving both hardware and software has been recognised to be a difficult problem. Indeed one of the grand challenges from the formal methods community is to develop techniques and methodologies for mechanically verify an embedded system application [13]. One possible solution is to verify the correctness of an application program along with the hardware on which the program is executed [4]. The verification is targeted towards a system with both hardware and software, rather than performing separate verifications for software and hardware.

This paper proposes an approach to verifying the correct execution of a machine code program for a customized soft–processor. The proposed approach involves modelling the execution of a soft–processor based on operational semantics [16], which provides a formalism for reasoning about the state-transition behaviour of the processor when executing a given machine code program. Given that the processor is initialized properly, the verification would show that, upon completion of the machine code program, the final state of the processor would meet the functional specification of the application program. Our research includes developing design templates for soft–processors in Verilog, and supporting various ways of customizing soft–processors for a Processor Generator.

This paper presents two forms of soft–processors customization: adoption of custom instructions, and customizable number of processor pipeline stages. We have developed a prototype of our design and verification system based on the ACL2 theorem prover [10]. Our approach is complementary to previous work [7] on processor verification—we do not focus on providing a complete hardware model, but rather on verifying that the execution of a specific machine code program on a specific processor meets the specification of that program.

The rest of the paper is organized as follows. Section II discusses related work. Section III covers our design and verification approach. Section IV presents our soft–processor description. Section V describes the formalism for capturing soft–processors in the ACL2 theorem prover. Section VI contains a selection of case studies illustrating our approach. Finally, Section VII provides a summary and future work.

II. Related Work

For almost a decade, Moore has proposed a grand challenge in embedded system verification [13]. However, few has responded to his proposal. Many researchers choose to focus

978-1-4577-1740-6/11/$26.00 © 2011 IEEE

on verifying either complex hardware [9] or complex software systems [11].

Our work is inspired by the work of Chadpadhyay [5], Ray [16], and Fox [7]. Chadpadhyay et al. developed a design environment to model a wide range of processor architectures and to generate hardware for custom instructions. We extend this idea by developing an approach for formal verification that supports customization of soft–processors. Ray et al. described three strategies for program verification. We adapt one of the strategies for verifying programs targeting customized soft–processors. Fox et al. proposed a de-compilation procedure to verifying programs for an ARM processor. This procedure involves developing theorems about specific instructions in the program. In contrast, we adopt a generic approach by developing, where possible, techniques that can verify a variety of processors.

III. Design and Verification Approach

This section describes our proposed design and verification approach. The design and verification flow is shown in Fig. 1.

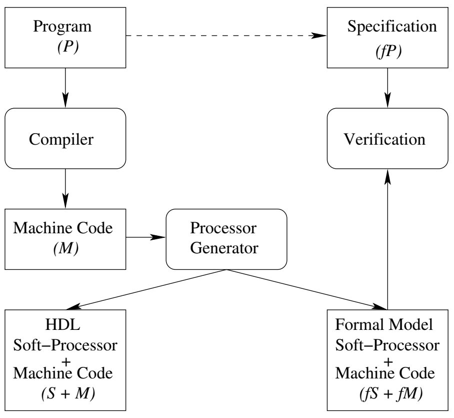

Fig. 1.
Design and Verification Approach.

Given one or more high-level programs, P, we aim to provide a soft–processor with appropriate customization for those programs, and a script for verifying that the execution of a machine code program, M, meets the corresponding specification of a high-level program. The machine code programs can be produced by hand or by a compiler; since compiler verification is difficult, most compilers in wide-spread use are not verified. There is no guarantee that the compilation produced a correct machine code.

The Processor Generator uses the machine code program M to customize a soft–processor template and generate a customized soft–processor design. The machine code characterizes the minimum number of soft–processor components that need to be implemented. Unused components are omitted during the soft–processor generation, resulting in an optimized system, $S + M$.

In our approach, a soft–processor includes various customizations to meet requirements of application developers. This paper covers simple means of customization, such as:

- the support for custom instructions, each of which has the effect of multiple instructions implementing frequently-occurring code fragments,
- processors with different amounts of pipeline stages; since pipelining improves throughput but could have lengthen latency, an optimal design could depend on specific application requirements.

Advanced customizations include different ways of multi-threading [6] and vector processing [19],[20]. These are beyond the scope of this paper, but we hope to discuss them in a future publication.

Eventually, we hope to automate soft–processor customization, generation, and verification. Ideally, the generation of an optimally customized processor is accompanied by a compiler for that processor, together with a verification script. The verification script covers a formal model of the customized processor, fS; it takes the functional specification of a high-level program, fP, the corresponding machine code model fM and the assertions about the machine code program, and verifies that the execution of the machine code on fS meets the functional specification fP. In our case the specification fS is a functional description defined recursively [15]. Assertions are properties about the program that provide the correctness criteria. We adopt a systematic translation between the HDL implementation and the formal model as the base for correct translation. There is a direct mapping between the two representations. Every module in the hardware description has its formal representation. The module structure is also maintained. More details on the mapping between these descriptions will be presented in Section V.

The proposed verification approach is based on a program verification formalism called operational semantics [16]. The evaluation target is a triple: a machine state `s`, the machine code program fM, and soft–processor fS. The machine-state `s` captures the values stored in the machine's registers and memory. The program fM is the application program represented in machine code. The processor fS is the formal model of the customized processor. The behavior of application program is interpreted through evaluating these triple data using two functions, `next` and `eval`. The `next` function models a one-step execution of the machine, analogous to executing the soft-processor for a single clock cycle. The `eval` function is a sequence of `n` step execution, analogous to executing the soft–processor for `n` clock cycles. The definition of `eval` function is shown in Eq. 1.

$$eval([s, fM, fS], n) \triangleq \begin{cases} eval(next\,[s, fM, fS], n-1) & \text{n} > 0 \\ [s, fM, fS] & \text{otherwise} \end{cases} \tag{1}$$

Program correctness is based on conditions defined by three predicates: `pre`, `post`, and `exit`. A predicate defines the conditions about the properties of the state `s` at a certain time. The predicate `pre` defines the initial/pre-condition of the program. It is a condition which is true before we evaluate the program fM. The predicate `post` defines the post-condition after completing program evaluation. It contains the condition after the program is completed. It also defines program termination condition. The predicate `exit` defines the final correctness evaluation criteria against the specification. The total correctness of the program is based on partial correctness of the program and its termination condition. The termination condition of the program is as follows:

$$\forall s : pre[s, fM, fS] \Rightarrow (\exists n : exit([s, fM, fS], n)) \tag{2}$$

The termination condition in Eq. 2 says that:

for any initial state, s, that satisfies the predicate pre, there exists n, a number of steps. When we execute the system for n steps from the initial condition, the resulting state satisfies the condition defined by predicate exit.

The correctness theorem for a program subroutine is as follows:

$$\begin{aligned}\forall s :\ & pre[s, fM, fS] \wedge exit(eval([s, fM, fS], n)) \\ & \Rightarrow post(eval([s, fM, fS], n))\end{aligned} \tag{3}$$

The correctness criterion is based on a state s that satisfies the pre-condition, $pre[s, fM, fS]$, and eventually reaches the exit state, $exit(eval([s, fM, fS], n))$, and satisfies post-condition, $post(eval([s, fM, fS], n))$.

Reasoning about the program involves defining the assertion properties, `assert`, of the system that are always true at a given point, called `cutpoint`. The `assert` and `cutpoint` statements guarantee that if there is a condition of a `cutpoint` that satisfies `assert`, the following cutpoints will also satisfy `assert` properties. The next reachable cutpoint from a state s is defined by the function $nextc$. The verification condition generator (VCG) for assertion reasoning is defined in the following theorems:

$$\forall s : pre[s, fM, fS] \Rightarrow cut[s, fM, fS] \wedge assert[s, fM, fS] \tag{4}$$

$$\forall s : exit[s, fM, fS] \Rightarrow cut[s, fM, fS] \tag{5}$$

$$\forall s : exit[s, fM, fS] \wedge assert[s, fM, fS] \Rightarrow post[s, fM, fS] \tag{6}$$

$$\begin{aligned}\forall s : & cut[s, fM, fS] \wedge assert[s, fM, fS] \wedge \neg exit[s, fM, fS] \\ & \Rightarrow assert(nextc(next[s, fM, fS])\end{aligned} \tag{7}$$

Eq. 4–7 state that:

for any state s that satisfies pre, state s must be a cutpoint and satisfies the condition of assert (Eq. 4).
An exit state s is a cutpoint (Eq. 5).
If state s is a cutpoint and state s satisfies the condition of assert, then state s must satisfy post (Eq. 6).
If state s is a cutpoint, not exit and satisfies the condition assert, then the next reachable cutpoint must satisfy assert (Eq. 7).

IV. Processor description

Our soft–processor is influenced by ARM-7 [1] and Xilinx PicoBlaze [2]. We adopt PicoBlaze's data-path architecture and ARM-7's instruction set as our soft-processor template. The data-path block diagram of the soft–processor is shown in Fig. 2. The main components of a soft–processor are the on–chip Random Access Memory (`on-chip RAM`), on–chip Read Only Memory (`on-chip ROM`) (includes the machine code program), registers (include program counter, instruction register, and pipeline registers), the ALU, the decoder, and the Input/Output (`I/O`) ports.

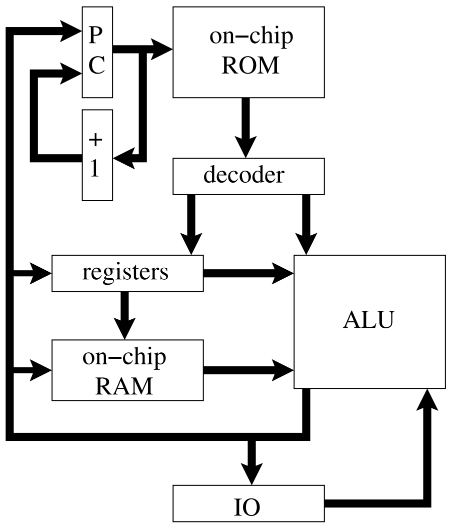

Fig. 2.
Soft–processor data-path block diagram.

In our architecture, program and data are represented in separate memory modules. Application program is stored in the on–chip ROM block. Application data are stored in the on–chip RAM block. We eliminate indirect addressing process by implementing two independent addressing registers, Program Counter register `PC` for the on–chip ROM and Memory address register `Madd` for the on–chip RAM.

The data-path operation has straightforward implementation. Program counter `PC` defines the address of current instruction to be fetched. This instruction is immediately decoded by the decoder and results in the ALU control logic and registers and memory addressing. All the data are fed to ALU for data manipulation operations. The result of ALU operation is a write back operations to either registers, on–chip RAM, I/O, or PC update.

Custom instructions and customizable pipelines. In our data-path architecture description, the effect of instruction set and pipeline customization are localized to a particular block while maintaining the generic implementation for the remainder. For example, the instruction set customization mainly affects the ALU module. Extending instruction set means more functionality covered by ALU module, while the rest of modules remains the same. The pipeline customization adds more register layers on the decoder. Similar to the instruction set customization, the remainder of the modules are kept intact.

We have also developed an instruction set template where designer can customize and extend pre-built instruction sets with a new instruction. Every instruction can use up to four generic registers (`Rd, Rn, Rm, Rs`), memory address register (`Madd`) and an immediate data (*Imm*). The register `Rn, Rm, Rs` are the source/operand registers. The register `Rd` is solely used as the target destination register. An instruction is defined by a tuple of triple of (assembly code construct, destination, operation). For example, A logical AND instruction is defined as `((AND Rd Rm Rn) Rd (logand Rm Rn))`. A new instruction functionality is restricted to the derivation of the predefined arithmetic and logical constructs.

The soft–processor's Verilog description follows a processor template module construct. At the bottom level, there are four groups of modules; application program, ALU, decoder, and auxiliary. The application program module (`prog_list`) contains the application machine code. The `ALU` module contains the functionality that is implemented and possibly used to compute instructions from the application program. The decoder module is a group of modules that decode current instruction into control signals, immediate data (used by ALU) and address registers (`decode_Rn`, `decode_reg_control`, `etc`). The auxiliary module is the program counter incrementer function. The `decode_Rn` and `prog_list` modules Verilog descriptions are as follows:

```
module decode_Rn (inst, Rn);
 input  [31:0] inst;
 output [3:0]  Rn;
 assign Rn = inst[19:16];
endmodule
module prog_list (pc_data, inst);
 input  [9:0]  pc_data;
 output [31:0] inst;
 assign inst = ...
              (pc_data == 12) ? ... : // MV  R0  R15
              ...
              (pc_data == 15) ? ... : // BZ  R0  #6
              ...
              (pc_data == 20) ? ... : // B   #-5
              (pc_data == 21) ? ... : // WRO R2
              ...
endmodule
```

The top level `core` module contains Input/Output (I/O) interfaces, modules connectivity, registers (includes `PC`, `MAdd`, `Regn`, `MEMn` and a write-back control sequence. The `core` module Verilog description of the processor is as follows:

```
module core(di, clock, do, Oflag);
 ...
 reg [31:0]    REGn[0:15], MEMn[0:1023], PC, do ;
 reg [9:0]     Madd;
 reg           Oflag;
 ...
 prog_list    i_prog_list           ...
 reg_control  i_decode_reg_control ...
 alu          i_alu                 ...
 ...
 always @(posedge clock)
  begin
   IC_pc = pc_next_data;
   if (opcode == 'WRO) Do = alu_data;
   ...
   if (reg_control) REGn[Rd_add] = alu_data;
  end
endmodule
```

The above is used as the template for our soft–processor generator (Fig. 1).

The introduction of custom instruction often changes ALU functionality, and may affect how the write back process is performed. The changes on internal operations will be reflected in a new machine code program, and a new processor implementation. For example, we merge (LDRd R_3) and (ADD R_2 R_2 R_3) into a single instruction (ADDd R_2 R_2). The `LDRd` instruction loads data from on–chip memory and store it in register R_3. The `ADD` instruction adds the value of R_2 and R_3, and store the result in register R_2. The new instruction `ADDd` performs an addition operation of register R_2 and data from on–chip memory, and stores the result in register R_2. We introduce this new instruction to the Processor Generator by submitting the new instruction construct `((ADDd Rd Rm) Rd (plus<32s> Rm Madd))`.

Another customization concerns having different number of pipeline stages in a processor. Adding pipeline stages affects decoder and write back operations, if the number of pipeline stages is customizable. As an example, consider a three-stage pipeline architecture, fetch-decode-execute, that we have implemented. The three-stage pipeline uses two variables $\texttt{pipe}_0$ and $\texttt{pipe}_1$, to capture the pipeline organization and a control variable corresponds to a finite-state-machine (`fsm`). The `fsm` variable is used to address pipeline hazard when the program counter and processor mode are changed. When the PC is changed, the instructions stored in pipeline registers are no longer valid and need to be flushed. The processor mode is then changed from execution mode to pipeline flush mode. The pipeline flush mode is represented by a non-zero value in the `fsm` variable.

V. Formal Modelling and Verification

Our processor generator produces a formal model of the soft–processor system (Fig. 1). We use the ACL2 theorem prover as our verification environment. We describe briefly below how the formal model in ACL2 is constructed from the Verilog module against its formal model function. All variables and wires in Verilog are declared with explicit size. They can have a single bit or a bit–vectors of size `n`. The formal model has three data-types: boolean, unsigned-byte, and signed-byte. Similarly, a single bit variable/wire is represented as of type boolean. The bit vector of size `n` is represented as either a unsigned–byte or a signed-byte with size `n`.

Every module in the Verilog description is represented as a function in ACL2. We map the relational representation in Verilog into a functional relation representation in the formal model in ACL2. Module instantiation corresponds to function evaluation. For example, module `prog_list` contains the machine code. It has one input argument: the address of program counter (`pc_data`). The result of evaluating this function will be discussed later. The `prog_list` function in ACL2 is as follows:

```
(defun prog_list (pc_data)
  (declare (type (unsigned-byte 10) pc_data))
  (case pc_data
    ...
    (12 27262991)
    ...
    (15 184550406)
    ...
    (20 167772165)
    (21 100794384)
    ...))
```

The top level module `core` contains three groups of operations: register declaration, module instantiation, and data-storage operation. Consider memory allocation for Verilog and ACL2. In the Verilog description, the memory covers the registers. Registers are used in the state machine of the processor: each register has a one-to-one mapping in ACL2 as a state variable. The state machine representation in ACL2 is as follows:

```
(defstobj machine-state
  (DIn   :type (signed-byte 32))
  (ProgC :type (unsigned-byte 10))
  (MEMn  :type (array(signed-byte 32)(1024)))
  (Madd  :type (unsigned-byte 10))
  (REGn  :type (array(signed-byte 32)(16)))
  (DOut  :type (signed-byte 32))
  (FlagO :type T))
```

There are two differences between these representations. First, all `core` module Verilog arguments, both input and output ports, are represented as state variables. In the core algorithm verification, it is not necessary for users to provide vector inputs as they are represented by symbolic constants. On the other hand, functions in ACL2 can be used in two ways: as descriptions in formal logic, or as executable specifications to produce simulation results. The input port is useful during simulation—real data can be used and the ACL2 formal model can be simulated using test-vectors. Second, an insight on how the registers are used is needed when constructing a state variable. Specifically, the variables which are declared as bit–vectors. All variables used in data-manipulation need to be declared to have type of signed-byte. Other bit-vectors variables/wires, such as instructions, are declared of type unsigned–byte.

The second operation in the `core` module is module instantiation. Modelling module instantiation is straight forward. Instantiations are represented as a sequence of function declaration. For example, the instantiation of `prog_list` module with input `pc_data` and output `inst` is as follows:

```
Verilog: prog_list
          i_prog_list (.pc_data (pc_data), .inst (inst));
ACL2:    (inst (prog_list pc_data))
```

The final operation in the `core` module is data-storage operation. In most cases, the result of data manipulation from `ALU` needs to be write–back either to the register, RAM, PC, or I/O. These process is controlled by clock function. We implement the `clock` function of HDL as the `state-update` operation. The `state-update` is a process where a sequence of operation is update the content of state-register is performed. A sketch of the `core` module in ACL2 is as follows:

```
(defun core (st)
  (let*
    (...
     (inst        (prog_list      pc_data))
     (reg_control (decode_reg_control ...))
     (alu_data    (alu                ...))
     ...
     (st (update-progc pc_next_data st))
     (st (if (equal opcode *WRO*)(update-do alu_data st)st))
     ...
     (st (if reg_control (update-regn alu_data st) st)))
    st))
```

The correctness of an application program is verified using `core` formal model as the execution engine. The verification script for the application program is organized into two parts. The first part contains a collection of theorems about the instruction set and the processor organization, which are independent of the code that it executes. The second part contains the assertions about the application program. We have developed a template for defining theorems about customizing instructions of a soft–processor.

Theorems about the processor are deduced from the processor's formal description. They contains specific features about the processor that have been proven against its formal description. These theorems can be grouped into three categories: module theorems, ALU theorems, and next-state theorems. The module theorems are focused on maintaining the correctness of the processor organization which includes the interconnection between modules. For example, input port of `prog_list` module is connected to `PC` register through `pc_data` wire. The output of the module is connected to `decoder` through `inst` wire. The (`pc_data`) and (`inst`) wire interconnection is defined to be bit–vectors of size 10 and 32 respectively. The `prog_list` ACL2 function explicitly constrains the input argument by declaring the input to have a unsigned-byte type with size of 10. The ACL2 statement is as follows: `(declare (type (unsigned-byte 10) pc_data)))`. In this functional representation, there is no explicit statement about the outputs. We show that the function satisfy output constraints by proving a theorem about the function. The theorem is as follows:

```
(defthm prog_list_gives_p32
  (implies (unsigned-byte-p 10 pc_data)
           (unsigned-byte-p 32 (prog_list pc_data))))
```

The theorem states that

for any given input-data with the type of unsigned-byte and size of 10, the result of evaluating `prog_list` *function with the input–data is of unsigned-byte type and size 32.*

The ALU theorems are a group of theorems about data-manipulations. Consider a 32-bit addition function `+32bvs`, one of the theorems of this function is as follows:

```
(defthm +bv32-0
  (implies (and (integerp x)
                (< x (expt 2 31))
                (<= (- (expt 2 31)) x))
           (equal (+bv32s x 0) x)))
```

The theorem states that:

for any integer x *and the value is less than* 2^{31} *and greater or equal than* -2^{31}*, then the result of adding* x *to 0 is* x*.*

A set of basic arithmetic (addition, subtraction) and logical (XOR, AND, OR) operations library has been developed for the processor. It includes the functional definition of each operations and its corresponding theorems. The theorems set the rules and constraints on how data-manipulation operations will correctly work. The library is sufficient to handle any derivative instruction customization using predefined operators. No new theorem are needed to handle permutation of the predefined operator.

The last group is the next–state theorems. The next–state theorems are a single step evaluation of the instructions. These theorems are used by ACL2 as rewrite rules in application program evaluation. One theorem is needed for every instruction implemented in a processor. We prove a general next-step theorem of instruction rather than proving each machine code [7]. For example, the next step theorem for the addition instruction, *ADD Rd Rm Rn*, is as follows:

$$\begin{aligned} \forall s.\ \ & statep(s) \wedge curr - opc = *ADD* \\ & \Rightarrow s([PC] = [PC] + 1; [Rd] = [Rm] + [Rn]) \end{aligned} \tag{8}$$

Eq. 8 states that

for any state s that satisfies the state predicate and the current instruction opcode is ∗ADD∗, the next state evaluation would increase the program counter PC value by 1 and the result of adding the value of register Rm and register Rn would be copied to register Rd.

Changes in soft–processors model due to customization are also reflected in the formal models and their supporting theorems. In most cases, both instruction and pipeline customizations only required a straight forward adaptation in the formal model descriptions. As long as custom instructions only use combination of pre-define operations, we only need to proof the equivalent next-state theorems of the instructions. For example: adding new instruction $ADDd$, as described in Section IV, requires only one additional theorem about the next-state for $ADDr$. The theorem is as follows:

$$\begin{aligned} \forall s.\ \ & statep(s) \wedge (curr - opc = *ADDd*) \\ & \Rightarrow s([PC] = [PC] + 1; [Rd] = [Rm] + [Madd]) \end{aligned} \tag{9}$$

When a new operator is introduced as part of the customization, a theory (set of definitions and theorems) about the operator is also needed.

One major difference between a pipelined and a non-pipelined processor is about information evaluated during the decode cycle. In a non-pipelined architecture, this information is immediately calculated from the current instruction. In a pipelined architecture, the execution relies on control information evaluated and stored during the decode cycle. The relations between current instruction and control information are lost. We define such relationship information in the $correct\_decode$ constraint. The constraint is shown in Eq. 10.

$$\begin{aligned} & correct\_decode = \\ & (opc = (dec\_opc\ pipe0)) \wedge (reg\_ctrl = (dec\_reg\_ctrl\ opc)) \wedge \\ & (pc\_ctrl = (dec\_pc\_ctrl\ opc)) \wedge (mem\_ctrl = (dec\_mem\_ctrl\ opc)) \end{aligned} \tag{10}$$

The equation states that the *opc*, *pc_ctrl*, reg_ctrl, and mem_ctrl are the results from decoding an instruction in $pipe0$. The next-state theorem assumes that the processor's state s satisfies these constraints. All updates during execution need to be specified in the next-state theorem. This includes the flow between fetch and decode. A new instruction is fetched and stored in $pipe1$. Meanwhile the instruction in $pipe1$ is moved to the next pipeline register. This instruction is also decoded and the results are stored in their respective registers *opc*, pc_ctrl, etc. The next-state theorem for ADD is as follows:

$$\begin{aligned} & \forall s.statep(s) \wedge (curr - opc = *ADD*) \wedge correct\_decode(s) \wedge (fsm = 0) \\ & \quad \Rightarrow s([PC] = [PC] + 1;\ \ [Rd] = [Rm] + [Rn];\ \ decode(pipe1); \\ & \qquad\qquad pipe0 = pipe1;\ \ pipe1 = new\_instruction) \end{aligned} \tag{11}$$

Pipeline customization introduces processor mode to handle pipeline hazards. Existing theorems are only correct when the processor is in execution mode. This constraint is propagated to all next-state theorems. Furthermore, theorems about execution result when the processor in a flush mode need to be included in the next-state theorems group.

The remaining task, elaborated in the next section, is to develop the assertions for an application program.

VI. Case Study

In this case study, we present our approach to verify an application program and show how the verification script can easily be adapted to support processor customization. We illustrate our approach with an application program `SUM`. The program calculates the summation of an array of $(n + 1)$ elements: $\sum_{i=0}^{n} X_i$.

A high-level functional description of `SUM` is captured in a recursive style [15]. This description acts as the high-level specification in the verification of the application program. The `sum_spec` functional description is as follows:

```
(defun sum_spec (X n)
  (cond ((zp n) (nth 0 X))
        (t (plus<32s> (sum_spec X (- n 1)) (nth n X)))))
```

The function `plus<32s>` is a high-level representation for the 32-bit addition function similar to `+bv32s`. The `sum_spec` function accumulates the first `(n+1)` elements of the `X` array.The application program machine code resides in the program memory/Read Only Memory (ROM). Separating program memory and data memory avoid the complication of a self modifying program and ease the verification process. In general, the machine code implementation includes initial operation to read data from external environment and store them in the local memory/Random Access Memory (RAM). After reading all data, the summation process will be started. In this verification, we only consider the main summation algorithm, omitting the process of reading data from external environment. We assume that the data have been correctly read and stored in RAM. The machine code of the summation function is as follows:

```
        ...
:PRE    (MVIM  #0)        ;Madd = 0
        (LDRd  R2)        ;R2 = [Madd]
:LOOP   (BZ    R0 #6)     ;if (R0=0) PC=:END
        (SUBI  R0 R0 #1)  ;R0 = R0 - 1
        (ADDIM #1)        ;Madd = Madd + 1
        (LDRd  R3)        ;R3 = [Madd]
        (ADD   R2 R2 R3)  ;R2 = R2 + R3
        (B     #-5)       ;PC = :LOOP
:END    (WRO   R2)        ;Dout = R2
:EXIT   ...
```

The machine code program reads data from memory, adds them together and stores the result in register `R2`. The process is repeated until the content of register `R0` is 0. The final step is to send the result to the output port using the *WRO* instruction. Note that `MVIM` and `ADDIM` are respectively instructions for moving and adding an immediate constant C, specified as #`C`, to the memory address register `Madd`.

This machine code is used to reduce the instruction set table and ALU functionalities. Only instructions that are being used will be implemented in the processor. The simplest processor is configured as a non-pipelined architecture, and optional instruction set extension is not used.

Proving the correctness of the operation requires a general assumption about the data. One assumption is that both the specification and the formal processor model operate on an array `X` of size `(n+1)`. Another assumption is that the array data are stored in the `MEMn` of the state machines. We define a *data_assume* function that states that array `X` is represented in the specification and the formal model.

To verify the correctness of the implementation, one needs to define the assertion's cutpoints. Cutpoints are the reference points that link the specification and implementation. In the summation function, we define four cutpoints denoted by the labels in the machine code: `PRE`, `LOOP`, `EXIT`, and `END`.

`PRE` contains condition of program operation. In this example, the program operates on a data array of size `(n+1)`. This means that at least one datum is available when the core algorithm is ready to be executed. `PRE` has two assumptions: the symbolic value `n` is already stored in register `R0`, and register `R0` has to contain a non-negative integer.

`LOOP` contains the invariant properties of the iterative program. It links the result of iterative evaluation of the program against its specification. We state that when the program counter reach `LOOP`, the value of n will always equal or greater than zero and the value of Register `R0` is equal or less than `n` (has been reduced at least by one). There are two more properties of interest: the data pointer which indicates the amount of data processed, and the data container which stores the summation results. The amount of data which has been processed is denoted by `(n - R0)` and stored in Register `MAdd`. The container register `R2` is equal to the summation of `(n - R0)` data.

`END` is the post loop termination. It contains the final evaluation of the program for an array of size `n+1`. Here we know that the register `R0` has the value zero and the data container register `R2` has the final result of data summation.

`EXIT` describes the final state of the system after the program is completed. The evaluation result of the `sum_spec` function is presented at the data-output variable/port `DOut`.

A summary of the assertions for these cutpoints is shown in Table I. The correctness is obtained by formally proving that the operational semantics of the processor model satisfies the condition defined by the assertions and the data constraints.

In the `SUM` program, we perform instructions customization by merging `LRDd` and `ADD` instructions as `ADDd`. Since the customization did not introduce new arithmetic or logical

PC	Assertion
PRE	$0 \leq R_0 \wedge R_0 = n$
LOOP	$0 \leq n \wedge R_0 \leq n \wedge$ Madd = (n - R_0) $\wedge$ R_2 = SUM (X, n - R_0)
END	$R_0 = 0 \wedge R_2$ = SUM (X, n)
EXIT	DOut = SUM (X, n)

TABLE I
SUM: NON-PIPELINE ASSERTIONS.

definitions, only `next-state` theorem associated with this new instruction need to be proved. The criteria of correctness remain the same; existing verification script can be used with minor modification. Specifically, adjusting the value of predicates `LOOP`, `END`, and `EXIT` to reflect changes in address for instruction allocations. The assertions shown in Table I can be reused without any modification.

The second customization involves adding a three-stage pipeline. Pipelining uses more registers to store information from each pipeline stage. The general top-level correctness criteria remain unchanged. Internally, additional assertion constraints are needed on the newly-added state variables. Table II shows the assertion for a three-stage pipeline architecture. For clarity, we represent instruction machine code with its assembly representation.

PC	Assertion
PRE	$0 \leq R_0 \wedge R_0 = n \wedge$ fsm = 0 $\wedge$ opcode = *MVIM* $\wedge$ $pipe_0$ = (MVIM #0) $\wedge$ $pipe_1$ = (LDRd R2)
LOOP	$0 \leq n \wedge R_0 \leq n \wedge$ fsm = 0 $\wedge$ opcode = *BZ* $\wedge$ Madd = (n - R_0) $\wedge$ R2 = SUMn (X, n - R_0) $\wedge$ $pipe_0$ = (BZ R_0 #6) $\wedge$ $pipe_1$ = (SUBI R_0 R_0 #1)
END	$R_0 = 0 \wedge$ fsm = 0 $\wedge$ opcode = *WRO* $\wedge$ $pipe_1$ = 0 $\wedge$ $pipe_0$ = (WRO R_2) $\wedge$ R2 = SUMn (X, n)
EXIT	DO = SUMn (X, n)

TABLE II
SUM: PIPELINE ASSERTIONS.

Pipeline customization affects the assertion properties (`PRE, LOOP, END`). While all non-pipeline properties described in Table I are preserved, additional information is needed. Specifically, information about pipeline properties is captured in registers `fsm`, `opcode`, `pipe`$_0$, and `pipe`$_1$. The assertions indicate that the processor is about to execute instructions defined as cutpoints. In a non–pipelined architecture, the predicate `PC` is the memory address of current instruction. In a three-stage pipeline, this relation has changed to `PC+2`. The immediate relation information is lost between `PC` and current instruction. One way to rectify this is by adding an assertion on the condition of the pipeline and opcode. One common property is the processor mode defined by the `fsm` register. It states that `fsm` has the value of 0, which means the processor is in execution mode.

We have developed unpipelined soft–processors for three application programs: Fibonnaci (`FIB`), Summation function (`SUM`), and Boolean Inner Product (`BIP`), as well as soft–processors with custom instructions (`SUMc, BIPc, BIP+`) and with three-stage pipelining (`FIB-3, SUM-3, SUMc-3, BIPc-3, BIP+-3`). The suffix "c" and "+" denote respectively customization by merging two instructions, and customization by replacing logical operation with arithmetic operation. Suffix "`-3`" denotes three-stage pipelining. Our approach involves developing re-usable verification scripts. One can often start verifying the correctness of an application program using the simplest processor architecture. Properties from this simple processor can be reused in subsequent customizations. We conduct three kinds of experiments: swapping/replacing arithmetic/logical operators, customizing instruction by merging, and customizing with a three-stage pipeline. Swapping logical operator `XOR` to arithmetic operator `plus<32s>` does not change the assertion. The verification script is immediately applicable without any changes. The last two customization experiments confirm the approach that we previously discussed.

Performance measurements are obtained by synthesizing the Verilog processor descriptions using Xilinx ISE tools. We base resource usage and performance estimation on a Xilinx xc5vlx30 device. The ACL2 verification is carried out on a system with an Intel i7 processor and 6 GB memory. The results are shown in Table III.

Design	LUT/FF/ RAM	Speed (MHz)	Power (mW)	Time (μs)	Energy (nJ)	Verify (s)
`FIB`	241/43/0	163.3	0.8	4.3	3.44	17
`FIB-3`	286/75/0	207.6	0.9	4.3	3.87	90
`SUM`	363/54/2	137.0	1.5	4.4	6.6	35
`SUM-3`	374/107/2	192.2	1.8	4.2	7.56	105
`SUMc`	367/53/2	137.3	1.7	3.6	6.12	35
`SUMc-3`	364/108/2	183.9	1.9	3.8	7.22	105
`BIP`	433/53/3	137.1	1.6	5.8	9.28	58
`BIP-3`	434/107/3	198.9	2.2	5.0	11	168
`BIPc`	434/53/3	143.0	1.9	4.9	9.31	58
`BIPc-3`	450/112/3	213.1	2.1	4.2	8.82	168
`BIP+`	426/53/3	134.4	1.7	5.9	10.03	59
`BIP+-3`	434/111/3	215.1	2.0	4.6	9.2	170

TABLE III
RESULTS FROM THREE CASE STUDIES.

The above results (except for the ACL2 verification which verify all possible cases) cover generation of 100 Fibonacci numbers, summation of 100 numbers, and inner product of two vectors with 100 booleans. They show that, for computing Fibonacci, the unpipelined `FIB` is best; although `FIB-3` has a faster clock, it suffers from pipeline flushing. For summation, the unpipelined `SUMc` with custom instruction turns out to be fastest and consumes the least energy, although it consumes more power than the unpipelined `SUM`. For inner product, `BIP` consumes the least power, but the pipelined `BIPc-3` with custom instruction is 38% faster and 5% more energy efficient than `BIP`, although it consumes 31% more power than `BIP`. For all our cases, ACL2 completes the verification in fewer than 3 minutes.

VII. Summary

We have presented a verification approach for customized soft–processors. There are two distinct flows in this approach: design flow and verification flow. In the design flow, an application program is compiled producing the machine code. This machine code is used to customize the instruction set by stripping the unused functionality. A customizable soft–processor template is used to generate application-specific processor implementation in Verilog and its corresponding formal model. In the verification flow, the correctness of a machine code program is obtained by evaluating the program in the target soft–processor system and verifying it against the high-level specification. We have presented two soft–processor customization techniques, instruction customization and pipelining customization, to demonstrate our approach.

Although our current prototypes are simple, they show that the proposed approach is viable and promising. Further research includes supporting a variety of processors, customizations, and applications, as well as exploring the logical foundation of this approach and the opportunities for its further automation.

VIII. Acknowledgement

The support of UK EPSRC, the European Union Seventh Framework Programme under Grant agreement number 248976 and 257906, the HiPEAC European Network of Excellence, Maxeler Technologies, and Xilinx Inc. is gratefully acknowledged.

References

[1] ARM. ARM-7 Datasheet, DDI 0020C, December 1994.
[2] Xilinx. PicoBlaze 8-bit Embedded Microcontroller User Guide, UG129, June 2011.
[3] S. Bard and N.F. Rafla. Reducing Power Consumption in FPGAs by Pipelining. *Proc. Midwest Symp. on Cir. and Sys.*, 2008.
[4] W. Bevier, W. Hunt, J.S. Moore and W. Young. An Approach to System Verification. Technical Report 41, *Computational Logic Inc.*, 1989.
[5] A. Chattpadhyay, H. Myer and R. Leupers. LISA: A Uniform ADL for Embedded Processor Modeling, Implementation, and Software Toolsuite Generation. *Processor Description Languages*, Elsevier, 2008.
[6] R. Dimond, O. Mencer and W. Luk. Application-Specific Customisation of Multi-Threaded Soft Processors. *IEE Proc. Computers and Digital Tech.*, 153(3):173–180, 2006.
[7] A. Fox and M. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. *Proc. Int. conf. on Interactive Theorem Proving*, 2010.
[8] A. Hagiescu and W.F. Wong. Co-synthesis of FPGA-Based Application-Specific Floating Point SIMD Accelerators. *Proc. FPGA*, 2011.
[9] W. Hunt and S. Sword. Centaur Technology Media Unit Verification, Case study: Floating–Point Addition. *Proc. Int. Conf. on Computer Aided Verif.*, 2009.
[10] M. Kaufmann, J.S. Moore and R. Boyer. ACL2 version 4.1 (2010). http://www.cs.utexas.edu/∼moore/acl2
[11] G. Klein et al. seL4: Formal Verification of an OS Kernel. *Proc. 22nd ACM Symp. on Operating Sys. Principles*, 2009.
[12] S.K. Lam, T. Srikanthan and C.T. Clarke. Architecture-Aware Technique for Mapping Area-Time Efficient Custom Instructions onto FPGAs. *IEEE Trans. on Computers*, 60(5):680–692, 2011.
[13] J.S. Moore. A Grand Challenge Proposal for Formal Methods: A Verified Stack. *Proc. 10th Anni. Coll. of UNU IIST*, 2002.
[14] J.S. Moore. Symbolic Simulation: An ACL2 Approach. *Proc. Int. Conf. on Formal Methods in Computer-Aided Design*, 1998.
[15] M. Myreen and M. Gordon. Transforming Programs into Recursive Functions. *Proc. Brazilian Symp. on Formal Methods*, 2008.
[16] S. Ray et al. A Mechanical Analysis of Program Verification Strategies. *Journal of Automatic Reasoning*. 40(4), 2008.
[17] S. Singh and C.J. Lillieroth. Formal Verification of Reconfigurable Cores. *Proc. FCCM*, IEEE Computer Society Press, 1999.
[18] J.G. Tong et al. Soft-Core Processors for Embedded Systems. *Proc. Int. Conf. on Microelectronics*, 2006.
[19] P. Yiannacouras, J. Rose and J. G. Steffan. VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors, *Proc. CASES*, 2008.
[20] J. Yu et al. Vector Processing as a Soft Processor Accelerator. *ACM Trans. on Reconfig. Tech. and Sys.* 2(2), 2009.