Verifying Pipelines with BAT*

Panagiotis Manolios and Sudarshan K. Srinivasan

1 Introduction

In this chapter, we show how to use the Bit-level Analysis Tool (BAT) [21, 22, 20, 4] for hardware verification. The BAT system has been used in the analysis of systems ranging from cryptographic hash functions to machine code to biological systems to large component-based software systems [24, 13, 19], but here we focus on one application: verification of pipelined hardware systems. This chapter brings together results from previous work in a self-contained way, and is intended as a starting point for someone who is interested in using automatic formal verification tools to prove the correctness of hardware or low-level software. The structure and examples in this chapter are based on previous work by the authors that showed how to use the ACL2 theorem proving system [8] to model and verify pipelined machines [12].

Hardware systems are ubiquitous and are an integral part of safety-critical and security-critical systems. Ensuring the correct functioning of hardware is therefore of paramount importance as failure of deployed systems can lead to loss of life and treasure. A well known example is the bug that was found in the floating point division (FDIV) unit of the Intel Pentium processor and that led to a 475 million

Panagiotis Manolios
College of Computer and Information Science
Northeastern University
360 Huntington Avenue, Boston, MA 02115 U.S.A.
e-mail: pete@ccs.neu.edu

Sudarshan K. Srinivasan
Department of Electrical and Computer Engineering
North Dakota State University
1411 Centennial Blvd., Fargo, ND 58105-5285 U.S.A.
e-mail: sudarshan.srinivasan@ndsu.edu

* This research was funded in part by NASA Cooperative Agreement NNX08AE37A and NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

dollar write-off by Intel. Estimates show that a similar bug in the current generation of Intel processors would cost the processor company about 12 billion dollars [1].

One of the key optimizations used in hardware systems is pipelining. Pipelining is used extensively in hardware designs, including both mainstream and embedded microprocessor designs, multi-core systems, cache coherence protocols, memory interfaces, etc. Therefore, the verification of pipelines is an important, ubiquitous problem in hardware verification and has received a lot of attention from the research community.

Pipelines are essentially assembly lines. Just like it is much more efficient to build cars using an assembly line, it is also much more efficient to break up the execution of processor instructions into well-defined stages, *e.g.*, fetch, decode, and execute. In this way, at any point in time there can be multiple instructions being executed simultaneously, in parallel and in various stages of completion. Furthermore, in order to extract maximum performance from pipelines, synchronization between the various instructions being executed in parallel is required. This synchronization between instructions, memories, and register files is provided by complex pipeline controllers. This added complexity makes the design and verification of pipelines a challenging problem.

We use the BAT system [22] for pipelined machine verification for several reasons. The BAT specification language [21] is designed as a synthesizable HDL with formal semantics and can therefore be used to construct bit-level pipelined machine models amenable to formal analysis. The decision procedure incorporated in BAT includes a memory abstraction algorithm and memory rewriting techniques and can therefore deal with verification problems that involve large memories [20]. Also, the BAT decision procedure uses an efficient circuit to CNF compiler, which drastically improves efficiency [23, 4].

The notion of correctness that we use for pipelined machines is based on Well-Founded Equivalence Bisimulation (WEB) refinement [10, 11]. There are several attractive properties of refinement. The instruction set architecture (`ISA`) is used as the specification. Both safety and liveness are accounted for. The refinement map (a function used to relate states of the pipelined machine with states of its `ISA`) is a parameter of the framework and can therefore be studied and optimized to improve efficiency [14, 17, 16, 7]. Refinement is a compositional notion, a property that can be exploited to deal with scalability issues [15].

The rest of the chapter is organized as follows. Section 2 describes the BAT system, including the BAT specification language and the BAT decision procedure. Section 3 describes a 3-stage pipelined machine example and its `ISA`, and also shows how to model these machines using BAT. In section 4, we provide an overview of the notion of correctness we use, which is based on refinement. Section 5 shows how to verify pipelines with the BAT system, using the example of the 3-stage pipeline. Section 6 provides an overview of techniques to cope with the efficiency and scalability issues that arise when reasoning about more complex pipelined systems. Conclusions are given in Section 7.

2 Bit-Level Analysis Tool

The Bit-level Analysis Tool (BAT) is a system for solving verification problems arising from hardware, software, and security. BAT is designed to be used as a bounded model checker and *k*-induction engine for Register Transfer Level (RTL) models. At the core of the system is a decision procedure for quantifier-free formulas over the extensional theory of fixed-size bit-vectors and fixed-size bit-vector arrays (memories). BAT also incorporates a specification language that can be used to model hardware designs at the word-level and to express linear temporal logic (LTL) properties. In this section, we describe the BAT specification language and provide a brief overview of the BAT decision procedure.

2.1 BAT Specification Language

The BAT specification language is strongly typed and includes a type inference algorithm. BAT takes as input a machine description and LTL specification, and tries to either find a counterexample requiring no more steps than a user provided upper bound, or tries to prove no such counterexample exists. While BAT accepts various file formats, a commonly used format for the machine specification requires the following four sections: `:vars`, `:init`, `:trans`, and `:spec`. These correspond to the declaration of the variables making up the machine state, a Boolean formula describing valid initial states, a Boolean formula describing the transition relation, and an LTL formula giving the desired formula, respectively. In this section, we describe the main features of the language. For a complete description, see the BAT Web page [21].

2.1.1 Data Types

The BAT language is strongly typed. Variables are either bit-vectors or memories. The `:vars` section is a list of variable declarations that specify the types of each variable. Each variable declaration is either: (1) A symbol corresponding to the variable name, in which case the variable is a bit-vector of one bit (*e.g.*, `x`). (2) A list with 2 elements, a variable name and a positive integer, in which case the variable is a bit-vector of the given number of bits (*e.g.*, `(x 4)` is a bit-vector of 4 bits). (3) A list with 3 elements, a variable name and two positive integers, specifying that the variable is a memory with the given word size and number of words (*e.g.*, `(x 8 4)` is a memory with 8 4-bit words).

A `:vars` section then looks like this: `(:vars (x 2) y (z 8 16))`. In addition to variables, there are bit-vector and integer constants. Bit-vectors can be given in binary, hex, or octal. For example, numbers in binary start with `0b` and are followed by an arbitrary sequence of 0s and 1s.

Integers are represented by signed bit-vectors. The size of the bit-vector is determined by BAT's type-inferencing mechanism. The appropriate size is determined by the context in which the integer is used. For example, if `x` is a 4-bit bit-vector, then if we bitwise-and it with 3, it is written as `(and x 3)`. Then in this context, 3 is represented by the bit-vector `0b0011`, since bit-vectors that are bitwise-anded together must be of the same type. The only restriction in this case is that the integer must be representable in signed binary notation (2's compliment) in the number of bits dictated by the context.

2.1.2 Primitives

BAT supports primitives for Boolean, arithmetic, and memory operations. All the basic bitwise Boolean functions are provided. The functions `and`, `or`, and `xor` all take an arbitrary number of arguments and perform the appropriate operations. In addition, -> (implication), and <-> (iff) take exactly two arguments. The `not` function takes exactly one argument. All of these functions take bit-vectors of the same size, and return a bit-vector of that size.

Arithmetic operations include =, <, >, <= (less than or equal to), >= (greater than or equal to), `add`, `sub`, `inc`, and `dec`. BAT contains bit-vector related functions as well. These include different kind of shift and rotate operations, concatenation, and (signed) extension. For example, the `cat` function concatenates bit-vectors, returning a bit-vector with size equal to the sum of the inputs to the `cat` function. The most significant bits are to the left so the earlier arguments to the `cat` formula are more significant than the later arguments.

Memories have to be treated with care because the obvious translation that converts formulas involving memories to propositional logic leads to an exponential blow-up. The BAT system introduced a decision procedure for memories that leads to greatly reduced SAT problems [20]. The memory-specific BAT functions are `get` and `set`. The `get` function takes a memory and a bit-vector and returns the word of the memory addressed by the bit-vector. The `set` function takes a memory and two bit-vectors. It returns a memory equivalent to the original memory except that the word addressed by the first bit-vector is set to the value of the second bit-vector. In both cases the size of the addresses must be equal to the ceiling of the log of the number of words in the memory, and in the case of the `set` the size of the last argument must be equal to the words size of the memory. Memories can be directly compared for equality using = (type checking makes sure that they have the same type, *i.e.*, that they have the same word size and the same number of elements). In a similar way, they type of an `if` can be a memory (type checking again checks that the then and else cases have the same type).

2.1.3 Expressions

BAT supports several constructs to build bit-vector and bit-vector memory expressions. Conditional statements include `if` and `cond`. The `if` statement takes three arguments: the first is the test and must be a 1-bit bit-vector. The second and third arguments are the then and else clauses respectively and must be the same type. `Cond` statements are convenient for expressing a series of `if` statements. For example, a `cond` statement that returns -1 if $x < y$, 1 if $x > y$ and 0 otherwise is shown below:

```
(cond ((< x y) -1)
      ((> x y) 1)
      (0b1     0))
```

BAT provides a way to return multiple values from an expression (this becomes helpful in conjunction with user-defined functions). This is done simply by wrapping a sequence of values in an `mv` form:

```
(mv (+ a b) (set m x y))
```

This returns both the sum and the result of the set form.

The most complex construct of the BAT language is local. In its simplest form, it operates like a `let*` in Lisp. The following implementation of an ALU slice demonstrates one of the more complex abilities of the local.

```
(local ((nb (xor bnegate b))
        (res0 (and a nb))
        (res1 (or a nb))
        (((cout 1) (res2 1)) (fa a nb cin)))
   (cat cout (mux-4 res0 res1 res2 1u op))))
```

Here, the last binding binds variables `cout` and `res2` simultaneously. It declares each to be 1 bit, and binds them to the 2-bit output of the `fa` function (a user-defined function). This splits up the output of the `fa` function between `cout` and `res2` according to their sizes. Another feature of the `local` is illustrated by the following.

```
(local ((c 2))
       (((t0 (c 0))
         (alu-slice (a 0) (b 0) bnegate bnegate op))
        ((t1 (c 1))
         (alu-slice (a 1) (b 1) t0 bnegate op))
        (zero (= c 0)))
    (cat t1 c zero))
```

Here an extra argument appears at the beginning of the `local`. This is a list of bit-vector variable declarations. The idea is that these variables can be bound by bits

and pieces through the bindings. The first binding binds several values, as in the last example. However, in this example the second value being bound is not a variable, but a bit of the variable, `c`, declared in the first argument to the `local`. Likewise, the other bit of `c` is set in the second binding. It is also possible to set a sequence of bits in a similar way by giving 2 integers: `((c 0 1) (and a b))`.

Finally, it is possible to set multiple values to the result of an `mv` form:

```
(local ((aa mm) (mv (inc a) (set m a b)))
    (set mm c aa))
```

Here the types of the variables being bound are inferred from the type of the `mv` form.

2.1.4 User Defined Functions

In addition to the `:vars`, `:init`, `:trans`, and `:spec` sections of a specification, the user can define his or her own functions in the `:functions` section. Consider the following example.

```
(:functions
 (alu-output
  (32)
  ((op 4) (val1 32) (val2 32))
  (cond ((= op 0) (bits (+ val1 val2) 0 31))
        ((= op 1) (bits (- val1 val2) 0 31))
        (1b1 (bits (and val1 val2) 0 31)))))
```

The functions section takes a list of function definitions. In this example, we define one function. A function definition starts with the function name. Our function is called `alu-output`. The second element in the definition is the type. This is a list containing 1 positive integer for a bit-vector function (`alu-output`, for example returns a 32-bit bit-vector), 2 positive integers if the return type is a memory, and a list of 1 integer lists and 2 integer lists if multiple values are returned. For example `((1) (8 4))` would specify that the function returns a 1-bit bit-vector and a memory with 8 4-bit words. The third part of a function definition is a list of its arguments. This is just a list of variable definitions just like the ones in the `:vars` section. In the case of `alu-output`, the inputs are `op` (a 4-bit bit-vector), `val1` (a 32-bit bit-vector), and `val2` (another 32-bit bit-vector). The final element of a function definition is the function body. Its return type must be compatible with that of the function.

2.1.5 Specification Formats

BAT takes specifications in one of three formats. The first is a machine description for bounded model checking. A file in this format contains three items. The first is the keyword ":machine" (without the quotes). The second is the machine description (described below). The third is a natural number which represents the number of steps you want BAT to check the property for.

The other two formats are very similar. They are used to check if a formula holds for some values of the variables (existential), or if a formula holds for all values of the variables (universal). These files contain 4 items. The first is either ":exists" or ":forall" (without the quotes). The next is a list of variable declarations for the formula. The third is a list of function definitions for use in the formula (this can be () if there are no functions). The final argument is the formula itself, which is over the variables and functions declared earlier in the file.

For examples of all these formats, see the BAT Web page [21].

2.1.6 Other Language Features

Since the BAT language is an s-expression based language implemented in Lisp, it is easy to develop parametrized models. We routinely use Lisp functions that take in a set of input parameters and generate BAT models.

BAT also has support for defining constants, LTL temporal operators, and a number of other primitive operators not discussed here. We point the reader to the BAT Web page for detailed documentation on the BAT specification language [21].

2.2 BAT Decision Procedure

As we saw in the earlier section, a BAT specification includes a model and a property about the model that BAT attempts to verify. The BAT decision procedure translates the input specification to a Boolean formula in Conjunctive Normal Form (CNF). The CNF formula is then checked using a SAT solver. In the common case, where we are checking validity, if the CNF formula is found to be unsatisfiable, then this corresponds to a formal proof that the user-provided property is valid. If the CNF formula is satisfiable, then the satisfying assignment is used to construct a counterexample for the input property.

The translation from the input specification to CNF is performed using four high-level compilation steps and is based on a novel data structure for representing circuits known as the NICE dag, because it is a dag that contains Negations, Ites (If-Then-Else operators), Conjunctions, and Equivalences [4]. In the first step, functions are inlined, constants are propagated, and a range of other simplifications are performed. The output of the first step is a NICE dag that also includes `next` operators, memory variables, and memory operators. In the second step, the transition relation

is unrolled for as many steps as specified by the specification. This eliminates the `next` operators, resulting in a NICE dag with memory variables and memory operators. In the third step, BAT uses its own decision procedure for the extensional theory of arrays to reduce memories [20], which are then eliminated by replacing memory variables and memory operators with Boolean circuits, resulting in a NICE dag. In the fourth step, the NICE dag is translated to a SAT problem in CNF format.

2.2.1 Memory Abstraction

BAT incorporates an automatic, sound, and complete memory abstraction algorithm [20]. The algorithm allows BAT to handle verification problems that involve models with large memories, but with correctness properties that include only a small number of memory references. The verification of pipelined microprocessor models is an example of such verification problems.

The key idea of the abstraction algorithm is to reduce the size of a memory to a size that is comparable to the number of unique accesses (both read and write) to that memory. The insight here is that if in a correctness property, there are only 10 unique accesses to a memory with say 2^{32} words, it is enough to reason about a reduced version of the memory whose resulting size is just larger than 10, to check the property. Therefore, the original memory size can be drastically reduced. Note however that care has to taken when performing the reduction because a memory access could be a symbolic reference, *i.e.*, an access that could reference any one of a large number of words in the memory. Another complication is that we allow memories to be directly compared in any context, *i.e.*, we have to support an extensional theory of arrays.

The efficiency of memory abstraction depends on the size of the reduced memories, which in turn depends on the number of unique memory access. However, because of nested memory operations, it is often hard to determine if two different memory references correspond to the same symbolic reference. To improve the efficiency of the abstraction, BAT incorporates automated term-rewriting techniques employing a number of rewrite rules that are used to simplify expressions with memory operators. The simplifications performed by rewriting help to identify equivalent memory references thereby improving the efficiency of memory abstraction.

2.2.2 Efficient Translation To CNF

CNF generation can significantly affect SAT solving times. BAT introduced a new linear-time CNF generation algorithm, and extensive experiments, have show that our algorithm leads to faster SAT solving times and smaller CNF than existing approaches. Our CNF generation algorithm is based on NICE dags, which subsume And-Inverter Graphs (AIGs) and are designed to provide better normal forms at linear complexity. The details are beyond the scope of this chapter, but are described in detail elsewhere [4].

3 ISA and Pipelined Machine Models

In this section, we show how to model a simple instruction set architecture and a 3-stage pipelined implementation of this instruction set architecture using the BAT specification language. We start by defining `ISA`, a sequential machine that directly implements the instruction set architecture. We then define `MA`, a 3-stage pipelined implementation (the microarchitecture machine). As stated previously, the models are based on our previous work on using ACL2 for hardware verification [12]. Those models in turn are based on Sawada's simple machine [27] and our subsequent related machines [9].

The instructions in the `ISA` have 4 components, including an opcode, a destination register, and two source registers. The pipelined `MA` machine is shown in Figure 1. The functionality of the `ISA` is split into 3 stages so that each of the stages can operate in parallel on different instructions. Registers, known as pipeline latches, are used to separate the stages. The pipeline latches hold the intermediate results generated in a stage. The `MA` machine has two pipeline latches, latch 1 and latch 2 as shown in the figure. The three stages of our `MA` machine are fetch, set-up, and write. In the fetch stage, an instruction is fetched from memory using the program counter as the address, and is stored in latch 1. In the set-up stage, the source operands are retrieved from the register file and stored in latch 2, along with the rest of the instruction. In the write stage, the appropriate operation is performed by the ALU (arithmetic and logic unit), and the result of the ALU operation is stored in the destination register specified by the destination address of the instruction.

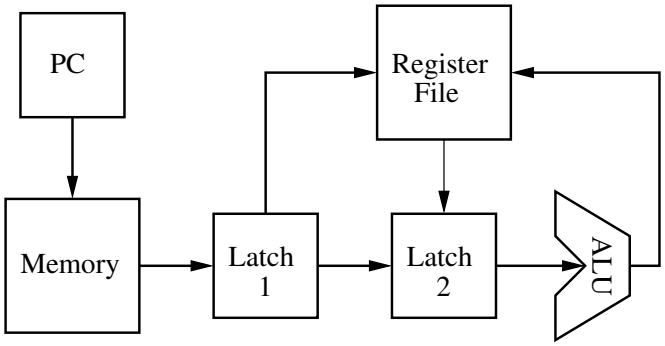

Fig. 1 Our simple 3-stage pipelined machine.

Consider a simple example, where the contents of the memory is as follows.

Inst

0	add	rb	ra	ra
1	add	ra	rb	ra

The following traces are obtained when the two-line code segment is executed on the ISA and MA machines. Note that we only show the values of the program counter and the contents of registers ra and rb.

Clock	ISA	MA	Inst 0	Inst 1
0	(0, (1,1))	(0, (1,1))		
1	(1, (1,2))	(1, (1,1))	Fetch	
2	(2, (3,2))	(2, (1,1))	Set−up	Fetch
3		(2, (1,2))	Write	Stall
4		(−, (1,2))		Set−up
5		(−, (3,2))		Write

The rows correspond to steps of the machines, *e.g.*, row Clock 0 corresponds to the initial state, Clock 1 to the next state, and so on. The ISA and MA columns contain the relevant parts of the state of the machines: a pair consisting of the PC and the register file (itself a pair consisting of registers ra and rb). The final two columns indicate what stage the instructions are in (only applicable to the MA machine).

The PC in the initial state (in row Clock 0) of the ISA machine is 0. The values of the registers ra and rb are 1. The next state of the ISA machine (row Clock 1) is obtained after executing instruction "Inst 0". In this state, the PC is incremented to 1, and the sum of the values stored in registers ra and rb (2) is computed and stored in rb. In the second clock cycle, instruction "Inst 1" is executed. The PC is again incremented to 2. The sum of the values stored in registers ra and rb (3) is computed and stored in ra.

In the initial state of the MA machine, the PC is 0. We assume that the two latches are initially empty. In the first clock cycle, "Inst 0" is fetched and the PC is incremented. In the second clock cycle, "Inst 1" is fetched, the PC is incremented again, and "Inst 0" proceeds to the set-up stage. In the third clock cycle, "Inst 0" completes and updates register rb with the correct value (as can be seen from the MA column). However, during this cycle, "Inst 1" cannot proceed, as it requires the rb value computed by "Inst 0", and therefore is stalled and remains in the fetch stage. In the next clock cycle, "Inst 1" moves to set-up, as it can obtain the the rb value it requires from the register file, which has now been updated by "Inst 0". In the fifth clock cycle, "Inst 1" completes and updates register ra.

3.1 ISA Definition

We now consider how to define the ISA and MA machines using BAT. The first machine we define is a 32-bit ISA *i.e.*, the data path is 32 bits. The main function is isa-step, a function that steps the ISA machine, *i.e.*, it takes an ISA state and returns the next ISA state. The definition of isa-step follows.

```
(isa-step
 ((32) (4294967296 32)
  (4294967296 100) (4294967296 32))
 ((pc 32) (regs 4294967296 32)
  (imem 4294967296 100) (dmem 4294967296 32))
 (local
  ((inst (get imem pc))
   (op (opcode inst))
   (rc (dest-c inst))
   (ra (src-a inst))
   (rb (src-b inst)))
  (cond ((= op 0) (isa-add rc ra rb pc regs imem dmem))
        ;; REGS[rc] := REGS[ra] + REGS[rb]
        ((= op 1) (isa-sub rc ra rb pc regs imem dmem))
        ;; REGS[rc] := REGS[ra] - REGS[rb]
        ((= op 2) (isa-and rc ra rb pc regs imem dmem))
        ;; REGS[rc] := REGS[ra] and REGS[rb]
        ((= op 3) (isa-load rc ra pc regs imem dmem))
        ;; REGS[rc] := MEM[ra]
        ((= op 4) (isa-loadi rc ra pc regs imem dmem))
        ;; REGS[rc] := MEM[REGS[ra]]
        ((= op 5) (isa-store ra rb pc regs imem dmem))
        ;; MEM[REGS[ra]] := REGS[rb]
        ((= op 6) (isa-bez ra rb pc regs imem dmem))
        ;; REGS[ra]=0 -> pc:=pc+REGS[rb]
        ((= op 7) (isa-jump ra pc regs imem dmem))
        ;; pc:=REGS[ra]
        (1b1 (isa-default pc regs imem dmem)))))
```

In the above function `regs` refers to the register file, `imem` is the instruction memory, and `dmem` is the data memory. The function fetches the instruction from the instruction memory, which is a bit-vector. Then it uses decode functions `opcode`, `dest-c`, `src-a`, and `src-b` to decode the instruction. The opcode is then used to figure out what action to take. For example, in the case of an add instruction, the next `ISA` state is `(isa-add rc ra rb pc regs imem dmem)`, where `isa-add` provides the semantics of add instructions. The definition of `isa-add` is given below.

```
(isa-add
   ((32) (4294967296 32)
         (4294967296 100) (4294967296 32))
   ((rc 32) (ra 32) (rb 32) (pc 32) (regs 4294967296 32)
    (imem 4294967296 100) (dmem 4294967296 32))
   (mv (bits (+ pc 1) 0 31)
       (add-rc ra rb rc regs)
```

```
       imem
       dmem))


(add-rc (4294967296 32)
   ((ra 32) (rb 32) (rc 32) (regs 4294967296 32))
   (set regs
        rc
        (bits (+ (get regs ra) (get regs rb)) 0 31)))
```

Notice that the program counter is incremented and the register file is updated by setting the value of register rc to the sum of the values in registers ra and rb. This happens in function `add-rc`.

The other ALU instructions are similarly defined. We now show how to define the semantics of the rest of the instructions. The semantics of the load instructions are shown next.

```
(isa-loadi
 ((32) (4294967296 32)
  (4294967296 100) (4294967296 32))
 ((rc 32) (ra 32) (pc 32) (regs 4294967296 32)
  (imem 4294967296 100) (dmem 4294967296 32))
 (mv (bits (+ pc 1) 0 31)
     (load-rc (get regs ra) rc regs dmem)
     imem
     dmem))


(load-rc
 (4294967296 32)
 ((ad 32) (rc 32) (regs 4294967296 32)
  (dmem 4294967296 32))
 (set regs rc (get dmem ad)))


(isa-load
 ((32) (4294967296 32)
  (4294967296 100) (4294967296 32))
 ((rc 32) (ad 32) (pc 32) (regs 4294967296 32)
  (imem 4294967296 100) (dmem 4294967296 32))
 (mv (bits (+ pc 1) 0 31)
     (load-rc ad rc regs dmem)
     imem
     dmem))
```

The semantics of the store instruction is given by `isa-store`.

```
(isa-store
 ((32) (4294967296 32)
  (4294967296 100) (4294967296 32))
 ((ra 32) (rb 32) (pc 32) (regs 4294967296 32)
  (imem 4294967296 100) (dmem 4294967296 32))
 (mv (bits (+ pc 1) 0 31)
     regs
     imem
     (store ra rb regs dmem)))


(store
 (4294967296 32)
 ((ra 32) (rb 32) (regs 4294967296 32)
  (dmem 4294967296 32))
 (set dmem (get regs ra) (get regs rb)))
```

Jump and branch instructions follow.

```
(isa-jump
 ((32) (4294967296 32)
  (4294967296 100) (4294967296 32))
 ((ra 32) (pc 32) (regs 4294967296 32)
  (imem 4294967296 100) (dmem 4294967296 32))
 (mv (bits (get regs ra) 0 31)
     regs
     imem
     dmem))


(isa-bez ((32) (4294967296 32)
 (4294967296 100) (4294967296 32))
 ((ra 32) (rb 32) (pc 32) (regs 4294967296 32)
  (imem 4294967296 100) (dmem 4294967296 32))
 (mv (bez ra rb regs pc)
     regs
     imem
     dmem))


(bez
 (32)
 ((ra 32) (rb 32) (regs 4294967296 32) (pc 32))
 (cond ((= (get regs ra) 0)
        (bits (+ pc (bits (get regs rb) 0 31)) 0 31))
       (1b1 (bits (+ pc 1) 0 31))))
```

No-ops are handled by `isa-default`.

```
(isa-default
 ((32) (4294967296 32)
  (4294967296 100) (4294967296 32))
 ((pc 32) (regs 4294967296 32)
  (imem 4294967296 100) (dmem 4294967296 32))
 (mv (bits (+ pc 1) 0 31)
     regs
     imem
     dmem))
```

3.2 MA Definition

The `MA` machine is a pipelined machine with three stages that implements the instruction set architecture of the `ISA` machine. Therefore, the `ISA` machine can be thought of as a specification of the `MA` machine. The `MA` machine contains a PC, a register file, a memory, and two pipeline latches. The latches are used to implement pipelining and stores intermediate results generated in each stage. The first latch contains a flag which indicates if the latch is valid, an opcode, the target register, and two source registers. The second latch contains a flag as before, an opcode, the target register, and the values of the two source registers. The definition of `ma-step` follows.

```
(ma-step
 ((298) (4294967296 32)
  (4294967296 100) (4294967296 32))
 ((ma 298) (regs 4294967296 32)
  (imem 4294967296 100) (dmem 4294967296 32))
 (mv
  (cat
   (step-latch2 ma regs)
   (step-latch1 ma imem)
   (step-pc ma regs imem))
   (step-regs ma regs dmem)
   imem
   (step-dmem ma dmem)))
```

The `ma-step` function works by calling functions that given one of the `MA` components return the next state value of that component. Note that this is very

different from `isa-step`, which calls functions, based on the type of the next instruction, that return the complete next `isa` state.

Below, we show how the register file is updated. If latch2 is valid, then if we have an ALU instruction, the output of the ALU is used to update register rc. Otherwise, if we have a load instruction, then we update register rc with the appropriate word from memory.

```
(step-regs
 (4294967296 32)
 ((ma 298) (regs 4294967296 32) (dmem 4294967296 32))
   (local
    ((validp (getvalidp2 ma))
     (op (getop2 ma))
     (rc (getrc2 ma))
     (ra-val (getra-val2 ma))
     (rb-val (getrb-val2 ma)))
    (cond ((and validp (alu-opp op))
           (set regs rc (alu-output op ra-val rb-val)))
          ((and validp (load-opp op))
           (set regs rc (get dmem ra-val)))
          (1b1 regs))))


(alu-opp
 (1)
 ((op 4))
 (or (= op 0) (= op 1) (= op 2)))


(load-opp
 (1)
 ((op 4))
 (or (= op 3) (= op 4)))


(alu-output
 (32)
 ((op 4) (val1 32) (val2 32))
   (cond ((= op 0) (bits (+ val1 val2) 0 31))
         ((= op 1) (bits (- val1 val2) 0 31))
         (1b1 (bits (and val1 val2) 0 31))))
```

Next, we describe how latch 2 is updated. Latch 2 is invalidated if latch 1 will be stalled or if latch 1 is not valid. Otherwise, we copy the opcode and rc fields from latch1 and read the contents of registers rb and ra, except for load instructions. We use a history variable pch2 to record the value of the PC value corresponding to the

instruction in latch 2. A similar history variable, pch1, is used in latch 1 to record the PC value corresponding to the instruction in latch 1. Note that history variables do not affect the computation of the machine. They are used primarily to aid the proof process.

```
(step-latch2
 (133)
 ((ma 298) (regs 4294967296 32))
   (local ((l1op (getop1 ma)))
    (cond ((= (or (not (getvalidp1 ma))
                  (stall-l1p ma)) 1b1)
           (cat (getpch2 ma)
                (getrb-val2 ma)
                (getra-val2 ma)
                (getrc2 ma)
                (getop2 ma)
                1b0))
          (1b1
           (cat (getpch1 ma)
                (get regs (getrb1 ma))
                (cond ((= l1op 3) (getra1 ma))
                      (1b1 (get regs (getra1 ma))))
                (getrc1 ma)
                l1op
                1b1)))))
```

Latch 1 is updated as follows. If it is stalled, it retains its previous contents. If it is invalidated, its flag is set to false. Otherwise, the next instruction is fetched from memory and stored in latch 1. The PC of the instruction is stored in pch1. Latch 1 is stalled when the instruction in latch 1 requires a value computed by the instruction in latch 2. Latch 1 is invalidated if it contains any branch instruction (because the jump address cannot be determined yet) or if latch 2 contains a bez instruction (again, the jump address cannot be determined for bez instructions until the instruction has made its way through the pipeline, whereas the jump address for jump instructions can be computed during the second stage of the machine).

```
(step-latch1 (133) ((ma 298) (imem 4294967296 100))
   (local
    ((latch1 (getlatch1 ma))
     (inst (get imem (getppc ma))))
    (cond ((= (stall-l1p ma) 1b1) latch1)
          ((= (invalidate-l1p ma) 1b1)
           (cat (getpch1 ma)
                (getrb1 ma)
                (getra1 ma)
                (getrc1 ma)
```

```
                (getop1 ma)
                1b0))
          (1b1
           (cat (getppc ma)
                (src-b inst)
                (src-a inst)
                (dest-c inst)
                (opcode inst)
                1b1)))))
```

The function `stall-l1p` determines when to stall latch 1.

```
(stall-l1p (1) ((ma 298))
 (local
  ((l1validp (getvalidp1 ma))
   (l1op (getop1 ma))
   (l2op (getop2 ma))
   (l2validp (getvalidp2 ma))
   (l2rc (getrc2 ma))
   (l1ra (getra1 ma))
   (l1rb (getrb1 ma)))
  (and l2validp l1validp (rc-activep l2op)
       (or (= l1ra l2rc)
           (and (uses-rbp l1op) (= l1rb l2rc))))))


(rc-activep (1) ((op 4))
 (or (alu-opp op) (load-opp op)))


(uses-rbp (1) ((op 4))
 (or (alu-opp op) (= op 5) (= op 6)))
```

The function `invalidate-l1p` determines when latch 1 should be invalidated.

```
(invalidate-l1p (1) ((ma 298))
 (local
  ((l1validp (getvalidp1 ma))
   (l1op (getop1 ma))
   (l2op (getop2 ma))
   (l2validp (getvalidp2 ma)))
  (or (and l1validp (or (= l1op 6) (= l1op 7)))
      (and l2validp (= l2op 6)))))
```

Memory is updated only when we have a store instruction, in which case we update the memory appropriately.

```
(step-dmem
 (4294967296 32)
 ((ma 298) (dmem 4294967296 32))
   (local
    ((l2validp (getvalidp2 ma))
     (l2op (getop2 ma))
     (l2ra-val (getra-val2 ma))
     (l2rb-val (getrb-val2 ma)))
    (cond ((= (and l2validp (= l2op 5)) 1b1)
           (set dmem l2ra-val l2rb-val))
          (1b1 dmem))))
```

Finally, the PC is updated as follows. If latch 1 stalls, then the PC is not modified. Otherwise, if latch 1 is invalidated, then if this is due to a bez instruction in latch2, the jump address can be now be determined, so the program counter is updated as per the semantics of the bez instruction. Otherwise, if the invalidation is due to a jump instruction in latch 1, the jump address can be computed and the program counter is set to this address. The only other possibility is that the invalidation is due to a bez instruction in latch 1; in this case the jump address has not yet been determined, so the pc is not modified. Note, this simple machine does not have a branch predictor. If the invalidate signal does not hold, then we increment the program counter unless we are fetching a branch instruction.

```
(step-pc (32)
 ((ma 298) (regs 4294967296 32) (imem 4294967296 100))
   (local
    ((pc (getppc ma))
     (inst (get imem pc))
     (op (opcode inst))
     (l1op (getop1 ma))
     (l2op (getop2 ma))
     (l2validp (getvalidp2 ma))
     (l2ra-val (getra-val2 ma))
     (l2rb-val (getrb-val2 ma)))
    (cond ((stall-l1p ma) pc)
          ((invalidate-l1p ma)
           (cond
            ((and l2validp (= l2op 6))
             (cond
              ((= l2ra-val 0)
               (bits (alu-output 0 pc l2rb-val) 0 31))
              (1b1 (bits (+ pc 1) 0 31))))
            ((= l1op 7)
```

```
        (bits (get regs (getra1 ma)) 0 31))
       (1b1 pc)))
((or (= op 6) (= op 7)) pc)
(1b1 (bits (+ pc 1) 0 31)))))
```

4 Refinement

In the previous section, we saw how one can model a pipelined machine and its instruction set architecture in BAT. We now discuss how to verify such machines. Consider the partial traces of the ISA and MA machines on the simple two-line code fragment from the previous section (add rb ra ra followed by add ra rb ra). We are only showing the value of the program counter and the contents of registers ra and rb.

ISA	**MA**		**MA**		**MA**
(0, (1,1))	(0, (1,1))		(0, (1,1))		(0, (1,1))
(1, (1,2))	(1, (1,1))	⟶	(0, (1,1))	⟶	(1, (1,2))
(2, (3,2))	(2, (1,1))	Commit	(0, (1,1))	Remove	(2, (3,2))
	(2, (1,2))	PC	(1, (1,2))	Stutter	
	(–, (1,2))		(1, (1,2))		
	(–, (3,2))		(2, (3,2))		

Notice that the PC differs in the two traces and this occurs because the pipeline, initially empty, is being filled and the PC points to the next instruction to fetch. If the PC were to point to the next instruction to commit (*i.e.*, the next instruction to complete), then we would get the trace shown in column 3. Notice that in column 3, the PC does not change from 0 to 1 until Inst 0 is committed in which case the next instruction to commit is Inst 1. We now have a trace that is the same as the ISA trace except for stuttering; after removing the stuttering we have, in column 4, the ISA trace.

We now formalize the above and start with the notion of a refinement map, a function that maps MA states to ISA states. In the above example we mapped MA states to ISA states by transforming the PC. Proving correctness amounts to relating MA states with the ISA states they map to under the refinement map and proving a WEB (Well-founded Equivalence Bisimulation). Proving a WEB guarantees that MA states and related ISA states have related computations up to finite stuttering. This is a strong notion of equivalence, *e.g.*, a consequence is that the two machines satisfy the same CTL$^* \setminus \mathsf{X}$[2]. This includes the class of next-time free safety and liveness

[2] CTL*is a branching-time temporal logic; CTL$^* \setminus \mathsf{X}$ is CTL* without the next-time operator X.

(including fairness) properties, *e.g.*, one such property is that the MA machine cannot deadlock (because the ISA machine cannot deadlock).

Why "up to finite stuttering"? Because we are comparing machines at different levels of abstraction: the pipelined machine is a low-level implementation of the high-level ISA specification. When comparing systems at different levels of abstraction, it is often the case that the low-level system requires several steps to match a single step of the high-level system.

Why use a refinement map? Because there may be components in one system that do not appear in the other, *e.g.*, the MA machine has latches but the ISA machine does not. In addition, data can be represented in different ways, *e.g.*, a pipelined machine might use binary numbers whereas its instruction set architecture might use a decimal representation. Yet another reason is that components present in both systems may have different behaviors, as is the case with the PC above. Notice that the refinement map affects how MA and ISA states are related, not the behavior of the MA machine. The theory of refinement we present is based on transition systems (TSs). A TS, $\mathscr{M}$, is a triple $\langle S, \dashrightarrow, L\rangle$, consisting of a set of states, S, a left-total transition relation, $\dashrightarrow \subseteq S^2$, and a labeling function L whose domain is S and where $L.s$ (we sometimes use an infix dot to denote function application) corresponds to what is "visible" at state s. Clearly, the ISA and MA machines can be thought of as transition systems (TS).

Our notion of refinement is based on the following definition of stuttering bisimulation [2], where by $fp(\sigma, s)$ we mean that σ is a fullpath (infinite path) starting at s, and by $match(B, \sigma, \delta)$ we mean that the fullpaths σ and δ are equivalent sequences up to finite stuttering (repetition of states).

Definition 1. $B \subseteq S \times S$ is a stuttering bisimulation (STB) on TS $\mathscr{M} = \langle S, \dashrightarrow, L\rangle$ iff B is an equivalence relation and for all s, w such that sBw:

$$\begin{array}{ll} (\text{Stb1}) & L.s = L.w \\ (\text{Stb2}) & \langle \forall \sigma : fp(\sigma, s) : \langle \exists \delta : fp(\delta, w) : match(B, \sigma, \delta)\rangle\rangle \end{array}$$

Browne, Clarke, and Grumberg have shown that states that are stuttering bisimilar satisfy the same next-time-free temporal logic formulas [2].

Lemma 1. *Let B be an STB on $\mathscr{M}$ and let sBw. For any* CTL$^* \setminus$ X *formula f, $\mathscr{M}, w \models f$ iff $\mathscr{M}, s \models f$.*

We note that stuttering bisimulation differs from weak bisimulation [25] in that weak bisimulation allows infinite stuttering. Stuttering is a common phenomenon when comparing systems at different levels of abstraction, *e.g.*, if the pipeline is empty, MA will require several steps to complete an instruction, whereas ISA completes an instruction during every step. Distinguishing between infinite and finite stuttering is important, because (among other things) we want to distinguish deadlock from stutter.

When we say that MA refines ISA, we mean that in the disjoint union ($\uplus$) of the two systems, there is an STB that relates every pair of states w, s such that w is an MA state and $r(w) = s$.

Definition 2. (STB Refinement) Let $\mathscr{M} = \langle S, \dashrightarrow, L\rangle$, $\mathscr{M}' = \langle S', \dashrightarrow', L'\rangle$, and $r : S \to S'$. We say that $\mathscr{M}$ is a STB refinement of $\mathscr{M}'$ with respect to refinement map r, written $\mathscr{M} \approx_r \mathscr{M}'$, if there exists a relation, B, such that $\langle \forall s \in S :: sBr.s\rangle$ and B is an STB on the TS $\langle S \uplus S', \dashrightarrow \uplus \dashrightarrow', \mathscr{L}\rangle$, where $\mathscr{L}.s = L'.s$ for s an S' state and $\mathscr{L}.s = L'(r.s)$ otherwise.

STB refinement is a generally applicable notion. However, since it is based on bisimulation, it is often too strong a notion and in this case refinement based on stuttering simulation should be used (see [10, 11]). The reader may be surprised that STB refinement theorems can be proved in the context of pipelined machine verification; after all, features such as branch prediction can lead to nondeterministic pipelined machines, whereas the `ISA` is deterministic. While this is true, the pipelined machine is related to the `ISA` via a refinement map that hides the pipeline; when viewed in this way, the nondeterminism is masked and we can prove that the two systems are stuttering bisimilar (with respect to the `ISA` visible components).

A major shortcoming of the above formulation of refinement is that it requires reasoning about infinite paths, something that is difficult to automate [26]. In [10], WEB-refinement, an equivalent formulation is given that requires only local reasoning, involving only `MA` states, the `ISA` states they map to under the refinement map, and their successor states.

Definition 3. $B \subseteq S \times S$ is a WEB on TS $\mathscr{M} = \langle S, \dashrightarrow, L\rangle$ iff:

(1) B is an equivalence relation on S; and

(2) $\langle \forall s, w \in S :: sBw \;\Rightarrow\; L(s) = L(w)\rangle$; and

(3) There exist functions $erankl : S \times S \to \mathbb{N}, erankt : S \to W$,
such that $\langle W, \prec\rangle$ is well-founded, and
$\langle \forall s, u, w \in S :: sBw \;\wedge\; s \dashrightarrow u \;\Rightarrow$
(a) $\langle \exists v :: w \dashrightarrow v \;\wedge\; uBv\rangle \;\vee$
(b) $(uBw \;\wedge\; erankt(u) \prec erankt(s)) \;\vee$
(c) $\langle \exists v :: w \dashrightarrow v \;\wedge\; sBv \;\wedge\; erankl(v,u) < erankl(w,u)\rangle\rangle$

We call a pair $\langle rank, \langle W, \prec\rangle\rangle$ satisfying condition 3 in the above definition, a well-founded witness. The third WEB condition guarantees that related states have the same computations up to stuttering. If states s and w are in the same class and s can transit to u, then one of the following holds.

1. The transition can be matched with no stutter, in which case, u is matched by a step from w.
2. The transition can be matched but there is stutter on the left (from s), in which case, u and w are in the same class and the rank function decreases (to guarantee that w is forced to take a step eventually).
3. The transition can be matched but there is stutter on the right (from w), in which case, there is some successor v of w in the same class as s and the rank function decreases (to guarantee that u is eventually matched).

To prove a relation is a WEB, note that reasoning about single steps of $\dashrightarrow$ suffices. In addition we can often get by with a rank function of one argument.

Note that the notion of WEB refinement is independent of the refinement map used. For example, we can use the standard flushing refinement map [3], where MA states are mapped to ISA states by executing all partially completed instructions without fetching any new instructions, and then projecting out the ISA visible components. In previous work, we have explored the use of other refinement maps, *e.g.*, in [17, 16, 7], we present new classes of refinement maps that can provide several orders of magnitude improvements in verification times over the standard flushing-based refinement maps. In this paper, however, we use the commitment refinement map, introduced in [9].

A very important property of WEB refinement is that it is compositional, something that we have exploited in several different contexts [15, 18].

Theorem 1. *(Composition) If* $\mathscr{M} \approx_r \mathscr{M}'$ *and* $\mathscr{M}' \approx_q \mathscr{M}''$ *then* $\mathscr{M} \approx_{r;q} \mathscr{M}''$.

Above, $r;q$ denotes composition, *i.e.*, $(r;q)(s) = q(r.s)$.

From the above theorem we can derive several other composition results; for example:

Theorem 2. *(Composition)*

$$\frac{\texttt{MA} \approx_r \cdots \approx_q \texttt{ISA} \qquad \texttt{ISA} \parallel P \vdash \varphi}{\texttt{MA} \parallel P \vdash \varphi}$$

In this form, the above rule exactly matches the compositional proof rules in [5]. The above theorem states that to prove $\texttt{MA} \parallel P \vdash \varphi$ (that MA, the pipelined machine, executing program P satisfies property φ, a property over the ISA visible state), it suffices to prove $\texttt{MA} \approx \texttt{ISA}$ and $\texttt{ISA} \parallel P \vdash \varphi$: that MA refines ISA (which can be done using a sequence of refinement proofs) and that ISA, executing P, satisfies φ. That is, we can prove that code running on the pipelined machine is correct, by first proving that the pipelined machine refines the instruction set architecture and then proving that the software running on the instruction set—not on the pipelined machine—is correct.

5 Verification

This section describes how BAT is used to verify the 3-stage pipelined machine given in Section 3. Note that the definition of WEBs given in Section 4 cannot be directly expressed in the BAT specification language. Therefore, we first strengthen the WEB refinement proof obligation such that we obtain a statement that is expressible as a quantifier-free formula over the extensional theory of fixed-size bit-vectors and fixed-size bit-vector arrays (memories), the kind of formulas that BAT decides.

We first define the equivalence classes of B to consist of an ISA state and all the MA states whose image under the refinement map r is the ISA state. As a result,

condition 2 of the WEB refinement definition clearly holds. Since an `ISA` machine never stutters with respect to the `MA` machine, the second disjunct of the third condition in the WEB definition can be ignored. Also, the `ISA` machine is deterministic, and the `MA` machine if not deterministic, can be transformed to a deterministic machine using oracle variables [11]. Using these simplifications and after some symbolic manipulation, Condition 3 of the WEB definition can be strengthened to the following core refinement-based correctness formula, where *rank* is a function that maps states of `MA` into the natural numbers.

$$\begin{aligned}\langle \forall w \in \texttt{MA} :: \langle \forall s,u,v :: \quad & s = r(w) \ \wedge \ u = \texttt{ISA-step}(s) \ \wedge \\ & v = \texttt{MA-step}(w) \ \wedge \ u \neq r(v) \\ \Longrightarrow \quad & s = r(v) \ \wedge \ \mathit{rank}(v) < \mathit{rank}(w) \rangle \rangle\end{aligned}$$

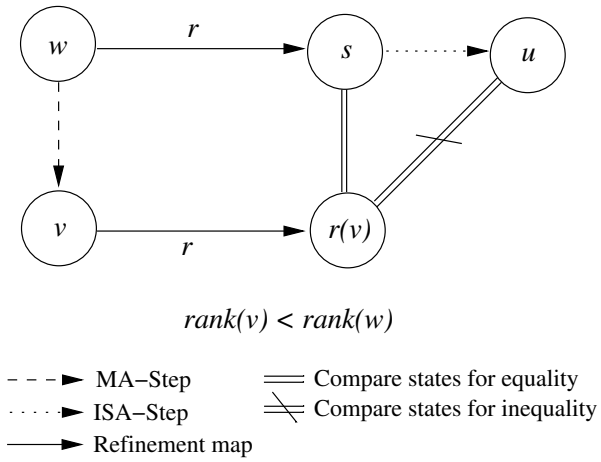

Fig. 2 Diagram shows the core theorem.

The correctness formula shown above is also depicted in Figure 2. In the formula above, if `MA` is the set of all reachable `MA` states, `MA-step` is a step of the `MA` machine, and `ISA-step` is a step of the `ISA` machine, then proving the above formula guarantees that the `MA` machine refines the `ISA` machine. In the formula above, w is an `MA` state and v (also an `MA` state) is a successor of w. s is an `ISA` state obtained by applying the refinement map r to w and u (also an `ISA` state) is a successor of s. The formula states that if applying the refinement map r to v does not result in the `ISA` state u, then $r(v)$ must be equal to s and the rank of v should decrease w.r.t. the rank of w. Also, the proof obligation relating s and v can be thought of as the safety component, and the proof obligation $\mathit{rank}(v) < \mathit{rank}(w)$ can be thought of as the liveness component.

If the ISA and MA models are described at the bit-level, then the core refinement-based correctness formula relating these models is in fact expressible in the logic that BAT decides.

5.1 Refinement Map Definitions

To check the core refinement-based correctness formula using BAT, two witness functions are required, a refinement map and a rank function. There are many different ways in which these witness functions can be defined. In this section, we describe one approach.

The following function is a recognizer for "good" MA states.

```
(good-ma (1)
 ((ma 298) (regs 4294967296 32) (imem 4294967296 100)
  (dmem 4294967296 32))
 (local
  (((nma nregs nimem ndmem)
    (committed-ma ma regs imem dmem))
   ((nma1 nregs1 nimem1 ndmem1)
    (ma-step nma nregs nimem ndmem))
   ((nma2 nregs2 nimem2 ndmem2)
    (ma-step nma1 nregs1 nimem1 ndmem1)))
  (cond ((getvalidp2 ma)
         (equiv-ma nma2 nregs2 nimem2 ndmem2
                   ma regs imem dmem))
        ((getvalidp1 ma)
         (equiv-ma nma1 nregs1 nimem1 ndmem1
                   ma regs imem dmem))
        (1b1 1b1))))
```

The "good" MA states (also known as reachable states) are states that are reachable from the reset states (states in which the pipeline latches are empty). The reason for using a recognizer for reachable states is that unreachable states can be inconsistent and interfere with verification by raising spurious counterexamples. A state in which a pipeline latch has an add instruction, when there are no add instructions in memory is an example of an inconsistent unreachable state. We check for reachable states by stepping the *committed state*, the state obtained by invalidating all partially completed instructions and altering the program counter so that it points to the next instruction to commit.

```
(committed-ma
 ((298) (4294967296 32)
  (4294967296 100) (4294967296 32))
 ((ma 298) (regs 4294967296 32) (imem 4294967296 100)
  (dmem 4294967296 32))
```

```
 (local ((inst (get imem (getppc ma))))
  (mv
   (cat
    (getpch2 ma) (getrb-val2 ma)
    (getra-val2 ma) (getrc2 ma)
    (getop2 ma) 1b0
    (getppc ma) (src-b inst)
    (src-a inst) (dest-c inst)
    (opcode inst) 1b0
    (committed-pc ma))
   regs imem dmem)))
```

The program counter (PC) of the committed state is the PC of the instruction in the first valid latch. Each latch has a history variable that stores the PC value corresponding to the instruction in that latch. Therefore, the PC of the committed state can be obtained from the history variables.

```
(committed-pc (32) ((ma 298))
 (cond ((getvalidp2 ma) (getpch2 ma))
       ((getvalidp1 ma) (getpch1 ma))
       (1b1 (getppc ma))))
```

The `equiv-MA` function is used to check if two `MA` states are equal. Note however that if latch 1 in both states are invalid, then the contents of latch 1 in both states are not compared for equality. Latch 2 is also compared similarly.

The `committed-MA` function invalidates partially executed instructions in the pipeline and essentially rolls back the program counter to correspond with the next instruction to be committed. The consistent states of `MA` are determined by checking that they are reachable from the committed states within two steps. The refinement map is defined as follows.

```
(ma-to-isa
 ((32) (4294967296 32)
  (4294967296 100) (4294967296 32))
 ((ma 298) (regs 4294967296 32)
  (imem 4294967296 100) (dmem 4294967296 32))
 (local (((nma nregs nimem ndmem)
          (committed-ma ma regs imem dmem)))
  (mv (getppc nma) nregs nimem ndmem)))
```

We also need a rank function to check for liveness, which is given by the `ma-rank` function. Note that this rank function is designed to work with the refinement map we defined. If another refinement map is used, then another rank may be required. `ma-rank` defines the rank of an `MA` state as the number of steps required to reach a state in which `MA` is ready to commit an instruction. If latch 2 is valid, an instruction will be committed in the next step. If latch 2 is invalid and latch 1 is valid, `MA` will commit an in two steps. If both latches are invalid, then then `MA` should commit an instruction in three steps.

```
(ma-rank (3) ((ma 298))
 (cond ((getvalidp2 ma) 0)
       ((getvalidp1 ma) 1)
       (1b1 2)))
```

Now, we can state the core theorem for the 3-stage pipelined machine, which is given by the function `commitment-theorem`.

```
(commitment-theorem (1)
 ((w-ma 298) (w-regs 4294967296 32)
  (w-imem 4294967296 100) (w-dmem 4294967296 32))
 (local
  (((s-pc s-regs s-imem s-dmem)
    (ma-to-isa w-ma w-regs w-imem w-dmem))
   ((v-ma v-regs v-imem v-dmem)
    (ma-step w-ma w-regs w-imem w-dmem))
   ((u-pc u-regs u-imem u-dmem)
    (isa-step s-pc s-regs s-imem s-dmem))
   ((rv-pc rv-regs rv-imem rv-dmem)
    (ma-to-isa v-ma v-regs v-imem v-dmem)))
  (-> (good-ma w-ma w-regs w-imem w-dmem)
   (and (good-ma v-ma v-regs v-imem v-dmem)
        (or (and (= rv-pc u-pc)
                 (= rv-regs u-regs)
                 (= rv-imem u-imem)
                 (= rv-dmem u-dmem))
            (and (= rv-pc s-pc)
                 (= rv-regs s-regs)
                 (= rv-imem s-imem)
                 (= rv-dmem s-dmem)
                 (< (ma-rank v-ma)
                    (ma-rank w-ma)))))))))
```

The commitment theorem also includes an inductive proof for the "good" `MA` invariant, *i.e.*, we check that if we step `MA` from any good state, then the successor of that state will also be good. Next, the property that we ask BAT to check is shown below. We declare a symbolic `MA` state in the `(:vars)` section. The symbolic state essentially corresponds to the set of syntactically all possible `MA` states. In the `(:spec)` section, we ask BAT to check if the `commitment-theorem` for all the `MA` states, which corresponds to the core theorem applied to the "good" `MA` states and an inductive invariance proof for the "good" `MA` invariant.

```
(:vars (mastate 298) (regs 4294967296 32)
       (imem 4294967296 100) (dmem 4294967296 32))
(:spec (commitment-theorem mastate regs imem dmem)))
```

Table 1 shows the verification times and CNF statistics for the verification of five 3-stage processor models using BAT. The models are obtained by varying the size of

the data path and the number of words in the register file and memories. Note that the original 3-stage model was parametrized, and the models for the experiments were generated by varying the parameters. The models are given the name "DLX3-n", where "n" indicates the size of the data path and the size of the program counter. The instruction memory, the data memory, and the register file each have 2^n words. The experiments were conducted on a 1.8GHz Intel (R) Core(TM) Duo CPU, with an L1 cache size of 2048KB. The SAT problems generated by BAT were checked using version 1.14 of the MiniSat SAT solver [6].

Table 1 Verification times and CNF statistics

Processor Model	Verification Times [sec]		CNF Statistics		
	MiniSat	Total (BAT)	Variables	Clauses	Literals
`DLX3-2`	0.10	0.32	363	1,862	9,914
`DLX3-4`	0.20	0.49	790	3,972	23,743
`DLX3-8`	0.47	1.01	1,486	7,536	46,599
`DLX3-16`	2.04	3.20	2,878	14,664	93,559
`DLX3-32`	6.03	8.63	5,662	28,920	192,471

6 Scaling to More Complex Designs

The formal proof of correctness for the 3-stage pipelined machine required stating the refinement correctness formula in the BAT specification language. BAT was then able to automatically prove the refinement theorem relating the 3-stage pipelined machine and its `ISA`. However, a big challenge in verifying pipelined machines using decision procedures is that as the complexity of the machine increases, the verification times are known to increase exponentially [19]. An alternate approach to verifying pipelined machines is based on using general purpose theorem provers. More complex designs can be handled using theorem provers, but a heroic effort is typically required on the part of the expert user effort to carry out refinement-based correctness proofs for pipelined machines [18]. In this section, we discuss some techniques for handling the scalability issues when using decision procedures for pipelined machine verification.

6.1 Efficient Refinement Maps

One of the advantages of using the WEB refinement framework is that the refinement map is factored out and can be studied independently. In Section 5, the commitment refinement map was described. There are other approaches to define the refinement map as well. Another well known approach to define the refinement map

based on flushing, the idea being that partially executed instructions in the pipeline latches of a pipelined machine state are forced to complete without fetching any new instructions. Projecting out the programmer-visible components in the resulting state gives the `ISA` state.

There are several more approaches to define the refinement map that have been found to be computationally more efficient. One approach is the Greatest Fixpoint invariant based commitment [16]. The idea here is to define the invariant that characterizes the set of reachable states in a computationally more efficient way. A second approach is collapsed flushing, which is an optimization of the flushing refinement map [7]. A third approach is intermediate refinement maps, that combine both flushing and commitment by choosing a point midway in the pipeline and committing all the latches before that point and flushing all the latches after that point [17]. This approach is also known to improve scalability and efficiency.

6.2 Compositional Reasoning

Refinement is a compositional notion as described in Section 4. The idea with compositional reasoning is to decompose the refinement correctness proof into smaller manageable pieces that can be efficiently handled using a decision procedure such as BAT. Another advantage with compositional reasoning is that the counter examples generated are smaller and more localized, making it easier to debug the design. A method for decomposing refinement proofs for pipelined machines has been developed in [15]. Proof rules are also provided to combine the smaller decomposed proofs to construct the refinement proof for the pipelined machine being verified.

6.3 Combining Theorem proving and Decision Procedures

The BAT decision procedures directly handles the verification problem at the RTL. One approach to handle scalability issues to to abstract and verify the pipelined machine at the term-level. The drawback however is that the final correctness result is only about the abstract model and the formal connection with the RTL model is lost. Hybrid approaches that exploit the refinement framework and use both theorem proving and decision procedures have been developed to address this problem [18]. The idea is to use the theorem prover to formally reduce the verification problem at the RTL to an abstract verification problem, which can then be handled by a decision procedure. The approach scales better for some complex machines, but is much less automatic than using a decision procedure like BAT.

6.4 Parametrization

An advantage of using BAT is that the models can be easily parametrized. This provides an effective debugging mechanism. The idea is based on the fact that models with smaller data path widths lead to computationally more tractable verification problems. For example, the verification of a 32-bit pipelined machine with many pipeline stages may not be tractable, but BAT could probably verify a 2-bit or 4-bit version of the model. While verifying a 4-bit version of the model does not guarantee correctness, a majority of the bugs (example control bugs that do not depend on the width of the data path) will be exposed. Generating a 4-bit version of a 32-bit model is easy to accomplish if the model is parametrized.

7 Conclusions

In this chapter, we described how to use the BAT system to verify that pipelined machines refine their instruction set architectures. The notion of correctness that we used is based on WEB refinement. We showed how to strengthen the WEB refinement condition to obtain a statement in the BAT specification language, for which BAT includes a decision procedure. This allows us to automatically check that the pipelined machine satisfies the same safety and liveness properties as its specification, the instruction set architecture. If there is a bug, then BAT will provide a counterexample. We also discussed various techniques to deal with more complex designs.

While much of the focus of pipelined machine verification has been in verifying microprocessor pipelines, these techniques can also be used to reason about other domains in which pipelines occur. Examples include cache coherence protocols and memory interfaces that use load and store buffers.

BAT is not limited to proving properties of pipelines. Any system that can be modeled using BAT's synthesizable hardware description language can be analyzed using BAT. This includes verification problems arising in both hardware and software, embedded systems, cryptographic hash functions, biological systems, and the assembly of large component-based software systems.

References

1. Bentley, B.: Validating a modern microprocessor (2005). See URL `http://-www.cav2005.inf.ed.ac.uk/bentley_CAV_07_08_2005.ppt`
2. Browne, M., Clarke, E.M., Grumberg, O.: Characterizing finite Kripke structures in propositional temporal logic. Theoretical Computer Science **59** (1988)
3. Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: Computer-Aided Verification (CAV '94), *LNCS*, vol. 818, pp. 68–80. Springer-Verlag (1994)

4. Chambers, B., Manolios, P., Vroon, D.: Faster sat solving with better cnf generation. In: Design, Automation and Test in Europe, DATE, pp. 1590–1595. IEEE (2009)
5. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
6. Een, N., Sorensson, N.: The minisat page. See URL `http://minisat.se/Main.html`
7. Kane, R., Manolios, P., Srinivasan, S.K.: Monolithic verification of deep pipelines with collapsed flushing. In: G.G.E. Gielen (ed.) Design, Automation and Test in Europe, (DATE'06), pp. 1234–1239. European Design and Automation Association, Leuven, Belgium (2006)
8. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
9. Manolios, P.: Correctness of pipelined machines. In: W.A. Hunt Jr., S.D. Johnson (eds.) Formal Methods in Computer-Aided Design–FMCAD 2000, *LNCS*, vol. 1954, pp. 161–178. Springer-Verlag (2000)
10. Manolios, P.: Mechanical verification of reactive systems. Ph.D. thesis, University of Texas at Austin (2001). See URL `http://www.ccs.neu.edu/~pete/research.html`
11. Manolios, P.: A compositional theory of refinement for branching time. In: D. Geist, E. Tronci (eds.) 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, *LNCS*, vol. 2860, pp. 304–318. Springer-Verlag (2003)
12. Manolios, P.: Refinement and Theorem Proving. International School on Formal Methods for the Design of Computer, Communication, and Software Systems: Hardware Verification. Springer Verlag, LNCS Series (2006)
13. Manolios, P., Oms, M.G., Valls, S.O.: Checking pedigree consistency with pcs. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS, *Lecture Notes in Computer Science*, vol. 4424, pp. 339–342. Springer (2007)
14. Manolios, P., Srinivasan, S.K.: Automatic verification of safety and liveness for xscale-like processor models using web refinements. In: Design, Automation and Test in Europe Conference and Exposition (DATE'04), pp. 168–175. IEEE Computer Society (2004)
15. Manolios, P., Srinivasan, S.K.: A complete compositional reasoning framework for the efficient verification of pipelined machines. In: International Conference on Computer-Aided Design (ICCAD'05), pp. 863–870. IEEE Computer Society (2005)
16. Manolios, P., Srinivasan, S.K.: A computationally efficient method based on commitment refinement maps for verifying pipelined machines. In: Formal Methods and Models for Co-Design (MEMOCODE'05), pp. 188–197. IEEE (2005)
17. Manolios, P., Srinivasan, S.K.: Refinement maps for efficient verification of processor models. In: Design, Automation and Test in Europe (DATE'05), pp. 1304–1309. IEEE Computer Society (2005)
18. Manolios, P., Srinivasan, S.K.: Verification of executable pipelined machines with bit-level interfaces. In: International Conference on Computer-Aided Design (ICCAD'05), pp. 855–862. IEEE Computer Society (2005)
19. Manolios, P., Srinivasan, S.K.: A refinement-based compositional reasoning framework for pipelined machine verification. IEEE Trans. VLSI Syst. **16**(4), 353–364 (2008)
20. Manolios, P., Srinivasan, S.K., Vroon, D.: Automatic memory reductions for rtl model verification. In: S. Hassoun (ed.) International Conference on Computer-Aided Design (ICCAD'06), pp. 786–793. ACM (2006)
21. Manolios, P., Srinivasan, S.K., Vroon, D.: BAT: The Bit-level Analysis Tool (2006). Available from `http://www.ccs.neu.edu/~pete/bat/`
22. Manolios, P., Srinivasan, S.K., Vroon, D.: BAT: The bit-level analysis tool. In: International Conference Computer Aided Verification (CAV'07) (2007)
23. Manolios, P., Vroon, D.: Efficient circuit to cnf conversion. In: J. Marques-Silva, K.A. Sakallah (eds.) International Conference Theory and Applications of Satisfiability Testing (SAT'07), *Lecture Notes in Computer Science*, vol. 4501, pp. 4–9. Springer (2007)
24. Manolios, P., Vroon, D., Subramanian, G.: Automating component-based system assembly. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, pp. 61–72. ACM (2007)
25. Milner, R.: Communication and Concurrency. Prentice-Hall (1990)

26. Namjoshi, K.S.: A simple characterization of stuttering bisimulation. In: 17th Conference on Foundations of Software Technology and Theoretical Computer Science, *LNCS*, vol. 1346, pp. 284–296 (1997)
27. Sawada, J.: Verification of a simple pipelined machine model. In: M. Kaufmann, P. Manolios, J.S. Moore (eds.) Computer-Aided Reasoning: ACL2 Case Studies, pp. 137–150. Kluwer Academic Publishers (2000)