ILA Tutorial : Template, Synthesis, and Verification

Last updated: 09/08/2016

This document provides three tutorials for ILA-based verification.

1. **Tutorial 1: ILA template writing and synthesis** covers template writing, simulator interface requirements, and synthesis.
2. **Tutorial 2: Manual ILA construction** covers how to use the ILA API as a programming language to build/modify ILA.
3. **Tutorial 3: Property specification and bounded model checking** covers how to specify property on a given model, how to create a golden model, and how to use the embedded bounded model checking feature.

Package Requirements:

python: version 2.7 or above

Z3: version 4.4.2

boost: version 1.60.0

ILA package

Files:

alu_[template/sim/interface/modify/bmc].py

synthesis.py

ALU.ila

ALU_moore.ila

Tutorial: ILA template writing and synthesis

In this tutorial, we will use an ALU as the example. As shown in Figure 1, the ALU has 8 8-bit registers and take one 16-bit input. Within the input, bits [15:8] is the *opcode*, and bits [7:0] is the *immediate*. It support three possible operations: addition, multiplication, and subtraction. The two operands can be either one of the 8 registers or the immediate, depending on the value of *opcode*. The output has 8 bits.

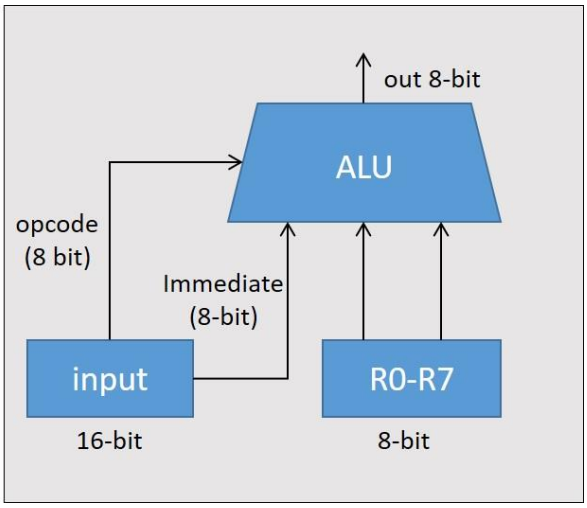

Figure 1: ALU structure.

A simulator for this ALU is provided in the file *alu_sim.py*.

Step 1: Write a template for the ALU [alu_template.py]

Our ILA package has python API that allows you to write template, synthesize, or manually construct ILA in a python environment. Once you install the package, you can then import the package.

- import ila

The first thing to do is to create a container for the model. Here you can specify a name for the model.

- m = ila.Abstraction('tutorial')

Once we have a container, we can then create/specified states in the container. For example, in the ALU example, there are eight registers, one input, and one output.

- reg0 = m.reg('reg0', 8)
- …
- instr = m.reg('input', 8)
- out = m.reg('output', 8)

Based on our understanding of the ALU design, now we can write a template for the model. The template is used to capture the possible behaviors of each state. That is to say, you are going to write down all possible state update functions for each state that you want to synthesize.

We know that the ALU output is the result of adding/subtracting/multiplying two operands. The operands can be either one of the registers or the immediate. Therefore, the below is one possible template for the output.

- imm = instr[7:0]
- src1 = ila.choice('src1', [reg0, reg1, …, imm])
- src2 = ila.choice('src2', [reg0, reg1, …, imm])
- ADD = src1 + src2
- SUB = src1 – src2
- MUL = src1 * src2
- out_nxt = ila.choice('out_nxt', [ADD, SUB, MUL])

The synthesis primitive *choice* is used to specify a set of possible values. The synthesis algorithm will configure which one in the *choice* is the correct one under every decoding case. We will explain how synthesis works and what it means by decode very soon. More details about the ILA syntax and semantics are provided in the above ILA manual.

After we define the template for the state, we have to assign it to the model container.

- m.set_next('output', out_nxt)

So far we have already defined the structure of the model and the template (possible update functions) for the states. The next step is to tell the synthesis algorithm how to synthesize the model. Consider the processor's execution model, the processor *fetch* the instruction, *decode* the instruction, and then *execute* depends on the decoded result. Our synthesis algorithm follow the same pattern. It tries to figure out what is the correct *execute* function for each *decode* result. During synthesis, it tries to configure all the unspecified holes in the template, such as the *choice*, under different decode cases. That is to say, every decode case can have different state update function. In our ALU example, the decode cases should be all the possible opcode.

- op = instr[15:8]
- m.decode_exprs = [op == i for i in xrange(0, 0x100)]

Now we have the template model.

Step 2: Prepare the simulator interface. [alu_interface.py]

The synthesis algorithm requires a simulator as an oracle to configure all the unspecified holes in the template. Therefore, we have to write an interface for the simulator to be used by the synthesis algorithm. Our simulator for the ALU example is provided in the file *alu_sim.py*.

Required simulator usage for synthesis:

1. Input: a dictionary for all states. s_in = {'reg0' : 0, 'reg1' : 1, …, 'input' : 0x0201}
2. Output: a dictionary for all states. s_out = {'reg0' : 0, 'reg1' : 1, …, 'output' : 1}
3. The state name in the dictionary should have the same name as specified in the model.

Ex. s_out = sim(s_in)

Usage for the given simulator:

1. Create a simulator object. alu = ALU()
2. Set the value for the 8 registers. alu.setStates([0, 1, 0, 1, 0, 1, 0, 1])
3. Simulate with an input. out = alu.simulate(0x0201)

An example interface for the provided simulator is provided in the file *sim_interface.py*.

Step 3: Use the simulator and the template to synthesize ILA. [synthesize.py]

Once you have the simulator interface and the template, you can then synthesize the model by using the function *synthesize*. The synthesis is done per state, specified by the state name.

- m.synthesize('output', sim)

Note that the synthesis of every state is independent. Parallel synthesis is a possible way to save time.

You can export the synthesis results for each state to a file.

- m.exportOne(m.get_next('output'), 'out.ila')

You can also export the whole model to a file.

- m.exportAll('alu.ila')

Tutorial: Manual ILA construction.

In this tutorial we will reuse the ALU ILA, stored in the file *ALU.ila*, which we build in ***Tutorial: ILA template writing and synthesis*** as our example. We will demonstrate how to manually massage the ILA and build the ILA without using synthesis techniques. This skill is useful for integrating and patching the models.

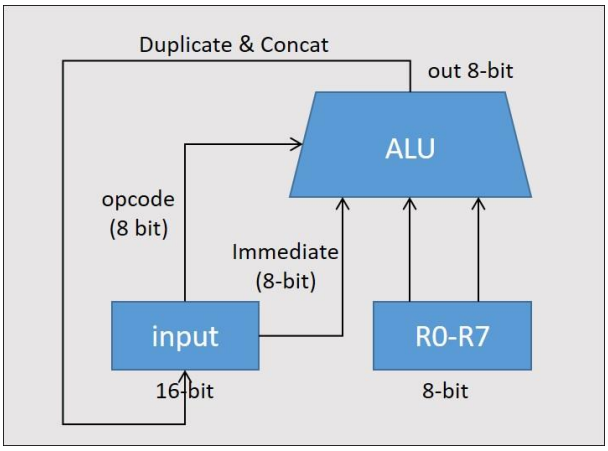

Figure 2: Modified ALU.

Figure 2 shows how we want to modified the ILA. Here we will take the synthesized ALU ILA, and change it into a Moore machine. Note that we already specified and synthesized the next state function for the state *output*. Now we want to set the next state functions for the state *input* and *reg*s.

The next state of the 16 bits state *input* comes from concatenating two copies of the 8 bits *output*. That is to say, if the next state value of *output* is 0x1a, then the next state value of *input* is 0x1a1a. All eight registers remain unchanged.

Step 1: Import ILA from file. [alu_modify.py]

There are two ways to reuse the synthesis result.

1. Create a new ILA container and create all the states. Import synthesis result for each state from file.
   - m = ila.Abstraction('alu')
   - m.reg('reg0', 8)
   - …
   - m.reg('output', 8)
   - out_nxt = m.importOne('out.ila')
   - m.set_next('output')
2. Create a new ILA container. Import whole model from file.
   - m = ila.Abstraction('alu')
   - m.importAll('ALU.ila')

Step 2: Manually define state update functions. [alu_modify.py]

After the model is imported, the state update function for the state *output* has been fully specified. However, we still have to define the transition relation for *input* and the eight registers. The only difference between

manual ILA construction and template writing is that you cannot use synthesis primitives, such as *choice*, *inrange*, *readslice*, and etc.

- reg0_nxt = m.getreg(‘reg0’)
- m.set_next(‘reg0’, reg0_nxt)
- out_nxt = m.get_next(‘output’)
- inp_nxt = ila.concat(out_nxt, out_nxt)
- m.set_next(‘input’, inp_nxt)

The ILA operator *getreg* will create a symbolic link to the state with the specified name. The ILA operator *concat* will concatenate two bitvector. After defining the state update function, we use *set_next* to assign to the model container.

Now we have a complete model where all states have their all state update functions. The model is ready for verification.

Tutorial: Property specification and bounded model checking

In this tutorial we will reuse the ALU ILA, stored in the file *ALU_moore.ila*, which we build in ***Tutorial: ILA template writing and synthesis*** as the example. We will demonstrate how to specify property in the ILA model and use bounded model checking technique to check the property.

If the ILA is synthesized correctly, it should be functionally equivalent to the simulator. Based on the simulator implementation, we will try to prove the following property:

If the initial value for the state *input* is 0, then the state *input* and *output* should always be 0.

This property should be true. If input is 0, then opcode and immediate are both 0. Under the case where opcode is 0, source 1 is the immediate, source 2 is reg1, and the operation is multiplication. Therefore, output should be 0 no matter what are the values of the eight registers. Later on we will try to use bounded model checking to check that this property holds in our model.

Step 1: Create state for property predicates. [alu_bmc.py]

Our bounded model checking framework will check the value of a certain state in the model after some number of steps. Therefore, the easiest way to specify properties to the model is through creating a new state to represent the property predicate.

- prop = m.bit('prop')

Here we create a new one-bit Boolean state called *prop*. This state will be used to indicate whether the property holds or not, by defining the following update function.

- prop_nxt = prop & (m.getreg('input') == 0x0)
- m.set_next('prop', prop_nxt)

Step 2: Set initial condition. [alu_bmc.py]

In addition to specifying the next state function, we also need to specify the initial condition.

- m.set_init('prop', m.bool(True))
- m.set_init('instr', m.const(0x0, 16))

Step 3: Create golden model. [alu_bmc.py]

Our bounded model checking function can be used to check the equivalence between two different models. However, here we only use it to check property on one model. Therefore, we can simply create a golden model where the state *prop* always hold true.

- golden = ila.Abstraction('golden')
- g_prop = golden.bit('prop')
- golden.set_next('prop', golden.bool(True))
- golden.set_init('prop', golden.bool(True))

Step 4: Bounded model check the property. [alu_bmc.py]

To call the bounded model checking function, simply specify the models, states, and steps.

- print ila.bmc(10, m, prop, 1, golden, g_prop)

If the property hold, then you will see a “True” being printed on the command prompt, as Figure 3 shows. If we remove the initial condition which specify the state *input* to 0, then you will see a “False” printed on the command prompt. Besides the false, it will also print out the assignment for all the states that fail the checking as Figure 4 shows. This counter example can help you debug the reason why the property does not hold.

```
boyuanhu@~/workspace/release/ilaLab$ time python alu_bmc.py
True

real    0m28.737s
user    0m27.146s
sys     0m0.660s
```

Figure 3: Property holds within a certain number of steps.

```
boyuanhu@~/workspace/release/ilaLab$ time python alu_bmc.py
(define-fun reg2_ () (_ BitVec 8)
  #x70)
(define-fun reg6_ () (_ BitVec 8)
  #x66)
(define-fun reg5_ () (_ BitVec 8)
  #x36)
(define-fun output_ () (_ BitVec 8)
  #x00)
(define-fun reg4_ () (_ BitVec 8)
  #x26)
(define-fun input_ () (_ BitVec 16)
  #x125b)
(define-fun reg3_ () (_ BitVec 8)
  #x14)
(define-fun reg7_ () (_ BitVec 8)
  #x00)
(define-fun reg0_ () (_ BitVec 8)
  #xe3)
(define-fun reg1_ () (_ BitVec 8)
  #x20)
false
true
False

real    1m4.671s
user    1m0.616s
sys     0m1.424s
```

Figure 4: Property not hold and the counter example.