Institute for System Programming of the Russian Academy of Sciences

Getting started with MicroTESK test program generator by the example of miniMIPS architecture

Moscow 2015

Contents

Introduction

Getting started document is intended to explain how to generate template based test programs with MicroTESK generator by the example of miniMIPS architecture.

MicroTESK is a reconfigurable (retargetable and extendable) model-based test program generator (TPG) for microprocessors and other programmable devices. The generator is customized with the help of instruction-set architecture (ISA) specifications and configuration files, which describe parameters of the microprocessor subsystems (pipeline, memory and others). The suggested approach eases the model development and makes it possible to apply the model-based testing in the early design stages when the microprocessor architecture is frequently modified.

The framework is applicable to a wide range of microprocessor architectures including RISC(ARM, MIPS, SPARC, etc.), CISC (x86, etc.), VLIW/EPIC (Elbrus, Itanium, etc.), DSP, GPU, etc.

The key idea of the reconfigurable TPG is that configuration of a generator for a concrete microprocessor architecture is implemented as easy as possible. It is obvious that to fulfill this requirement, a generator should be divided into two parts: (1) a platform-independent component (so-called core or engine) and (2) a component describing all platform-specific features (model or configuration). Such kinds of TPG tools are usually called model-based test program generators.

Test program generation is fulfilled in the following way. A verification engineer provides a test template of the scenario that should occur in the test program. A TPG tool generates a program by formulating and then solving constraints for each instruction of the test template or for groups of instructions. It also randomizes values of the unspecified parameters to produce different test program variations for the template given. The constraints are formulated in terms of the model, which consists of the microprocessor model and test coverage model. The first of them defines syntax and semantics of the target design's instructions, while the second one describes special conditions (known as test situations) to be achieved when verifying the design.

The generation core solves constraints by using external or built-in solvers and assigns the values to the instructions' operands. Then, it interprets the instructions under processing (for example, by using a microprocessor simulator) and takes the next group of instructions. Having performance limitations, TPG tools generally do not process test templates as a whole. Instead, they split them up into small pieces and formulate several Constraint Satisfaction Problems (CSPs). It is clear that such a technique can fail to find an exact solution of the joint CSP even when one is available, but it works well for test programs with low-density dependencies between instructions.

Our goal is to reach a new level in the TPG configuration flexibility. In order to do it, we suggest applying so-called architecture description languages (ADL), which are commonly used for developing microprocessor simulators (nML, ISDL, EXPRESSION, etc.). In the approach suggested, a microprocessor ISA is described in some ADL and then automatically translated into a microprocessor model together with a test coverage model. Fine tuning of the generator is performed by the subsystem-specific configuration files. Such an approach is especially useful in the early design stages when the project is frequently modified and rapid generator reconfiguration is required. The current version of the tool supports ISA specification (in nML) and manual development of test program templates (in Ruby).

The suggested approach eases the model development and makes it possible to apply the model-based testing in the early design stages when the microprocessor architecture is frequently modified. The MicroTESK installation guide can be found in
http://forge.ispras.ru/projects/microtesk/wiki/Installation_Guide.

Generation steps

The test program generation with MicroTESK consists of 4 main steps: 1) specification of the instruction set for the purpose microprocessor architecture, 2) microprocessor model generation based on specification described, 3) test templates description, 4) test program generation based on templates.

Instruction set specification

To describe a purpose microprocessor architecture MicroTESK supports ADLs, at that moment only nML/Sim-nML language. Thus instruction set specification should be specified in nML/Sim-nML.

Instruction set specification includes register specification, memory specification and instructions specification. Instruction set specification for microprocessor starts with register description.

Registers

In miniMIPS the following registers are available: GPR, COP_0_R, HI, LO, CIA.
Registers GPR and COP_0_R are 32 bits wide.

The data types description is necessary for register description. To define 32 bits wide register we need first to define 32-bit type.

Data types

Data types are defined as bit-vectors having different wide. These bit-vectors are interpreted as signed or unsigned numbers. To define 32-bit type we should write

type TYPE_NAME = card(32).

For example,

type BYTE = card(8)

Data type is declared which name is BYTE. This type has 8 bits and it is interpreted as unsigned number.

type WORD = int(32)

Data type is declared which name is WORD. It has 32 bits and is interpreted as signed number.

We defined the data type. After that we can define the register file where each element is a bit-vector of the defined type. For this purpose 'reg' expression is used. For example, define the register GPR in miniMIPS:

reg GPR [32, WORD]

Here GPR register of type WORD and having 32 elements is declared.

reg HI [WORD]

Register HI of type WORD is declared.

Using such constructions we could define GPR and HI register files of miniMIPS microprocessor. The examples of other register files description of miniMIPS microprocessor one can find in:
https://forge.ispras.ru/svn/microtesk/trunk/microtesk/src/main/arch/minimips/model/minimips.nml.

Memory

The next step of set instruction specification is memory specification. Main memory in miniMIPS supports 4 kilobyte addressing. Address has 16 bits wide. To define this memory we use mem construction. The number of elements in memory one can set with expression in square brackets. For example,

```
mem M [2 ** 16, WORD]
```

Line memory is set as array with name M which type is WORD with 2 ** 16 elements.

Constants and labels

One can set constants and labels with 'let' expression:

```
let REGISTER_INDEX_SIZE = 5
```

Constant REGISTER_INDEX_SIZE is declared, its value equals 5.

Variables

Variables could be necessary for instruction description. They can be defined with 'var' expression. For example, define the variable 'temp' in miniMIPS as 33-bits signed integer:

```
var temp[int(33)]
```

After defining registers and memory we can switch to instruction description.

Instructions

Instructions are described using operations and operands. The operations are specified as 'op-rules' whereas the operands are specified as parameters to 'op-rules'. The types of parameters that define the operands are the addressing modes specified using 'mode-reles'. 'Mode' and 'Op-rules' are arranged hierarchically using production rules. There are two kinds of production rules, OR rule and AND rule.

Operations *op*

Both the production rules for 'op-rules' are as follows.

OR rule

op n = n_1 | n_2 | n_3 . . .

For example, define add sub mov operation using or rule:

Add_sub_mov = Add | Sub | Mov

AND rule

op n_0 (p_1 : t_1, p_2 : t_2, p_3 : t_3 . . .)

a_1 = e_1 a_2 = e_2 a_3 = e_3 . . .

where t_i are tokens, interpreted as the type of p_i parameter. Each pair (a_i e_i) distinguishes the attribute and corresponding definition for n_0 symbol.

For example, in the expression

op y (rd: REG)

REG is a token for rd parameter.

Addressing *modes*

Mode rules are nearly analogous to above described op rules. Keyword 'mode' is used to define mode rules.

The attributes

Attributes are used to describe properties of instructions and addressing modes. There are some important predefined attributes.

Syntax-attribute describes textual (assembler) syntax of the instruction and evaluates to a string value. Image-attribute describes binary coding of the instruction. Action-attribute describes semantics of the instruction in terms of sequence of register transfer statements.

Root instruction

One can describe an instruction set in microprocessor by hierarchical tree like structure. The hierarchical structure facilitates sharing of description among related instructions in the instruction set. Any part from the root node to a leaf node constructs an individual instruction description. Each non-leaf node contains certain attributes, which can be shared by its descendants.

This structure facilitates description of processors having more than one instruction set. The root node provides an abstraction of the complete instruction set. Each node in between the root and the leaf nodes represents a set of instructions having certain common features, such as numeric instruction and load/store instruction.

The root instruction is named as 'instruction'. Assembler format of operation is set in the field 'syntax' (command.syntax). Binary coding is set in the field 'image' (command.image). Root instruction operation is described in 'action'.

```
op instruction (command: Operations)
  syntax = command.syntax
  image  = command.image
  action = {
    GPR[0] = coerce(WORD, 0);

    // PC is set based on the result of the previous instruction execution.
    // This is needed to simulate MIPS delay slot.
    if BRANCH == 0 then
      CIA = CIA + 4;
    else
      CIA = JMPADDR;
    endif;

    BRANCH = 0;
    command.action;
  }
```

Instruction *add*

Let us consider the description of arithmetic operation addition (*add*).

```
op add (rd: REG, rs: REG, rt: REG)
syntax = format("add %s, %s, %s", rd.syntax, rs.syntax, rt.syntax)
image = format("000000%s%s%s00000100000", rs.image, rt.image, rd.image)
action = {
temp = rs<31>::rs + rt<31>::rt;
if temp<32> != temp<31> then
exception("IntegerOverflow");
else
rd = temp<31..0>;
endif;
}
```

The signature of addition operation is set by the expression:

op add (rd: REG, rs: REG, rt: REG)

An action of the instruction is described in the field 'action'.

As we can see, describing an instruction based on an instruction set manual is a relatively easy task that can be performed by a verification engineer who does not have significant programming skills.

Instruction *beq*

The example of unconditional transfer instruction is below:

```
op beq (rs: REG, rt: REG, imm: SHORT)
syntax = format("beq %s, %s, %<label>d", rs.syntax, rt.syntax, imm)
image = format("000100%s%s%d", rs.image, rt.image, imm)
action = {
if rt == rs then
BRANCH = 1;
JMPADDR = CIA + 4 * imm + 4;
endif;
}
```

The examples of other instructions for miniMIPS architecture one can find also in svn.

The groups of operations

If an operation (instruction) has a common features with other operation, one can define the common operation for them. Then instructions having equal features will inherit one with other. In this case the instruction is defined as a set of operations which have one node.

One can use 'or' rule to describe a group of instructions. For example, in miniMIPS define the root instruction 'Operation' as a set of several instructions:

```
op Operations =  add
                |addi
                |addiu
                |addu
                |and
                |andi
```

```
            |beq
            |bgez
            |bgezal
            |bgtz
```

Then the root instruction will be one for each of above instructions.

So, the microprocessor instruction set can be easily described. After that we can feed this description to SinmNL translator in MicroTESK. The translator builds the inner representation (on Java) which is need for building the microprocessor model.

Microprocessor model generation

Based on the microprocessor specification (in Sim-nML) described above we can generate Java model of miniMIPS microprocessor with MicroTESK.

The MicroTESK installation instruction is described in
http://forge.ispras.ru/projects/microtesk/wiki/Installation_Guide.

Run compile.bat script:

```
bin\compile.bat arch\minimips\model\minimips.nml
```

This is script output:

```
Buildfile: C:\Programs\1\bin\build.xml

clean:
   [delete] Deleting directory C:\Programs\1\gen

BUILD SUCCESSFUL
Total time: 0 seconds
Translating: arch\minimips\model\minimips.nml
Model name: minimips
Included: arch\minimips\model\minimips.nml
Buildfile: C:\Programs\1\bin\build.xml

build:
    [mkdir] Created dir: C:\Programs\1\gen\bin
    [javac] Compiling 58 source files to C:\Programs\1\gen\bin
    [mkdir] Created dir: C:\Programs\1\gen\src\resources
      [jar] Building jar: C:\Programs\1\lib\jars\models.jar

BUILD SUCCESSFUL
Total time: 1 second
```

Thus we have generated Java model of miniMIPS microprocessor. This model can be found in C:\Programs\1\lib\jars и называется models.jar.

The next step is test templates writing.

Test template writing

Test template is the test scenario which should be executed by microprocessor model. Template consists of the instruction set and is written in Ruby. User writes the test template and feed it to MicroTESK input to generate test programs.

The sequence of instructions in test template is executed by generated Java model of the microprocessor and then is translated to the assembler code. Thus the test programs are generated.

The test template consists of two parts, the first one is a base template and the second one is a heritor of base template. If the set of test templates which contain some common parts is produced but not only one template hence all these parts are placed in the base class and then will be used. For that purpose the base template is created.

Template sample

Let us define the base template for miniMIPS.

```
require ENV['TEMPLATE']

class MiniMipsBaseTemplate < Template

  def initialize
    super
    # Initialize settings here
  end

  def pre
    data_config(:text => '.data', :target => 'M', :addressableSize => 8) {
      define_type :id => :byte, :text => '.byte', :type => type('card', 8)
      define_type :id => :half, :text => '.half', :type => type('card', 16)
      define_type :id => :word, :text => '.word', :type => type('card', 32)

      define_space :id => :space, :text => '.space', :fillWith => 0
      define_ascii_string :id => :ascii, :text => '.ascii', :zeroTerm => false
      define_ascii_string :id => :asciiz, :text => '.asciiz', :zeroTerm => true
    }

    #
    # The code below specifies an instruction sequence that writes a value
    # to the specified register (target) via the REG addressing mode.
    #
    preparator(:target => 'REG') {
      lui  target, value(16, 31)
      addi target, target, value(0, 15)
    }
  end
```

```
def post
  # Place your finalization code here
end

# Alias for the NOP instruction (MIPS idiom)
def nop
  sll zero, zero, 0
end

# Aliases for accessing General-Purpose Registers
#   Name    Number Usage                Preserved?
#   $zero      0   Constant zero
#   $at        1   Reserved (assembler)
#   $v0-$v1   2-3  Function result
#   $a0-$a3   4-7  Function arguments
#   $t0-$t7  8-15  Temporaries
#   $s0-$s7  16-23 Saved                    yes
#   $t8-$t9  24-25 Temporaries
#   $k0-$k1  26-27 Reserved (OS)
#   $gp       28   Global pointer           yes
#   $sp       29   Stack pointer            yes
#   $fp       30   Frame pointer            yes
#   $ra       31   Return address           yes

def zero
  reg(0)
end

def at
  reg(1)
end

def v0
  reg(2)
end

def v1
  reg(3)
end

def a0
  reg(4)
end

def a1
  reg(5)
end

def a2
```

```
  reg(6)
end

def a3
  reg(7)
end

def t0
  reg(8)
end

def t1
  reg(9)
end

def t2
  reg(10)
end

def t3
  reg(11)
end

def t4
  reg(12)
end

def t5
  reg(13)
end

def t6
  reg(14)
end

def t7
  reg(15)
end

def s0
  reg(16)
end

def s1
  reg(17)
end

def s2
  reg(18)
end
```

```
def s3
  reg(19)
end

def s4
  reg(20)
end

def s5
  reg(21)
end

def s6
  reg(22)
end

def s7
  reg(23)
end

def t8
  reg(24)
end

def t9
  reg(25)
end

def k0
  reg(26)
end

def k1
  reg(27)
end

def gp
  reg(28)
end

def sp
  reg(29)
end

def fp
  reg(30)
end

def ra
```

```
    reg(31)
  end

  # Shortcut methods to access memory resources in debug messages

  def gpr(index)
    location('GPR', index)
  end

  def mem(index)
    location('M', index)
  end
end
```

Then define the heritor of the base template. For example, describe the situations with overflow for integer addition and subtraction.

```
 class IntExceptionTemplate < MiniMipsBaseTemplate

  def run
    block(:combinator => 'PRODUCT', :compositor => 'RANDOM') {
      block {
        add t0, t1, t2 do situation('add', :case =>   'normal', :size => 32) end
        add t0, t1, t2 do situation('add', :case => 'overflow', :size => 32) end
      }

      block {
        sub t3, t4, t5 do situation('sub', :case =>   'normal', :size => 32) end
        sub t3, t4, t5 do situation('sub', :case => 'overflow', :size => 32) end
      }
    }
  end

end
```

Test program generation

We built the test template. Now we can feed it to MicroTESK and run it on generated Java model of miniMIPS microprocessor.

Run generate.bat script to generate the test program:

```
bin\generate.bat minimips arch/minimips/templates/int_exception.rb test.asm
```

The generated test program is in test.asm file.

The output of running generate.bat script is long enough and we show the small part of it only for one of four test situations.

```
----------------------------- Printing Test Case 1 -----------------------------

Initialization:

lui $12, 0x7044
addi $12, $12, 0xc4aa
lui $13, 0x5b68
addi $13, $13, 0xbad1
lui $9, 0x82ef
addi $9, $9, 0x796b
lui $10, 0x7f53
addi $10, $10, 0x5053

Main Code:

sub $11, $12, $13
add $8, $9, $10
```

The generated test program can be run on the simulator of the miniMIPS microprocessor model.

Conclusion

We have demonstrated how to generate the test programs based on templates with MicroTESK for miniMIPS architecture.

Easy-to-use the MicroTESK generator could be applied not only for uncomplicated architectures, such as in case considered with miniMIPS. But also the generator is useful for more complicated microprocessor architectures.