MicroTESK v2

Testing Guide

Corresponding to MicroTESK v2.5

Contents

Chapter 1

Overview

Overview.

Chapter 2

Basic Functions

2.1 Introduction

MicroTESK generates test programs on the basis of *test templates* that describe test programs to be generated in an abstract way. Test templates are created using special Ruby-based test template description language that derives all Ruby features and provides special facilities. The language is implemented as a library that implements facilities for describing test cases. Detailed information on Ruby features can be found in official documentation [2, 3].

MicroTESK uses the JRuby [4] interpreter to process test templates. This allows Ruby libraries to interact with other components of MicroTESK written in Java.

Test templates are processed in two stages:

1. Ruby code is executed to build the internal representation (a hierarchy of Java objects) of the test template.
2. The internal representation is processed with various engines to generate test cases which are then simulated on the reference model and printed to files.

This chapter describes facilities of the test template description language and supported test generation engines.

2.2 Test Template Structure

A test template is implemented as a class inherited from the `Template` library class that provides access to all features of the library. Information on

the location of the `Template` class is stored in the `TEMPLATE` environment variable. Thus, the definition of a test template class looks like this:

```
require ENV['TEMPLATE']

class MyTemplate < Template
```

Test template classes should contain implementations of the following methods:

1. `initialize` (optional) - specifies settings for the given test template;
2. `pre` (optional) - specifies the initialization code for test programs;
3. `post` (optional) - specifies the finalization code for test programs;
4. `run` - specifies the main code of test programs (test cases).

The definitions of optional methods can be skipped. In this case, the default implementations provided by the parent class will be used. The default implementation of the `initialize` method initializes the settings with default values. The default implementations of the `pre` and `post` methods do nothing.

The full interface of a test template looks as follows:

```
require ENV['TEMPLATE']

class MyTemplate < Template

  def initialize
    super
    # Initialize settings here
  end

  def pre
    # Place your initialization code here
  end

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

```
  def run
    # Place your test problem description here
  end

end
```

2.3 REUSING TEST TEMPLATES

It is possible to reuse code of existing test templates in other test templates. To do this, you need to subclass the template you want to reuse instead of the `Template` class. For example, the `MyTemplate` class below reuses code from the `MyPrepost` class that provides initialization and finalization code for similar test templates.

```
require ENV['TEMPLATE']
require_relative 'MyPrepost'

class MyTemplate < MyPrepost

  def run
    ...
  end

end
```

Another way to reuse code is creating code libraries with methods that can be called by test templates. A code library is defined as a Ruby module file and has the following structure:

```
module MyLibrary

  def method1
    ...
  end

  def method2(arg1, arg2)
    ...
  end

  def method3(arg1, arg2, arg3)
```

```
    ...
  end

end
```

To be able to use utility methods `method1`, `method2` and `method3` in a test template, the `MyLibrary` module must be included in that test template as a mixin. Once this is done, all methods of the library are available in the test template. Here is an example:

```
require ENV['TEMPLATE']
require_relative 'my_library'

class MyTemplate < Template
  include MyLibrary

  def run
    method1
    method2 arg1, arg2
    method3 arg1, arg2, arg3
  end

end
```

2.4 Test Template Settings

2.4.1 Managing Text Format

Test templates use the following settings that set up the format of generated test programs:

- `sl_comment_starts_with` - starting characters for single-line comments;
- `ml_comment_starts_with` - starting characters for multi-line comments;
- `ml_comment_ends_with` - terminating characters for multi-line comments;
- `indent_token` - indentation token;

- `separator_token` - token used in separator lines.

Here is how these settings are initialized with default values in the `Template` class:

```
@sl_comment_starts_with = "//"
@ml_comment_starts_with = "/*"
@ml_comment_ends_with   = "*/"

@indent_token     = "\t"
@separator_token = "="
```

The settings can be overridden in the `initialize` method of a test template. For example:

```
class MyTemplate < Template

  def initialize
    super

    @sl_comment_starts_with = ";"
    @ml_comment_starts_with = "/="
    @ml_comment_ends_with = "=/"

    @indent_token = "␣␣"
    @separator_token = "*"
  end
  ...
end
```

2.4.2 Managing Address Alignment

The `.align n` directive may have different interpretation for different assemblers. By default, MicroTESK assumes that it aligns an address to the next 2^n byte boundary. If this is not the case, to make MicroTESK correctly interpret it, the `alignment_in_bytes` function must be overridden in a test template. This function returns the number of bytes that correponds to n. The default implementation of the function looks like this:

```
#
# By default, align n is interpreted as alignment on 2**n byte border.
# This behavior can be overridden.
```

```
#
def alignment_in_bytes(n)
  2 ** n
end
```

2.5 TEXT PRINTING

The test template description language provides facilities for printing text messages. Text messages are printed either into the generated source code or into the simulator log. Here is the list of functions that print text:

- `newline` - adds the new line character into the test program;
- `text(format, *args)` - adds text into the test program;
- `trace(format, *args)` - prints text into the simulator execution log;
- `comment(format, *args)` - adds a comment into the test program;
- `start_comment` - starts a multi-line comment;
- `end_comment` - ends a multi-line comment.

Formatted Printing.

Functions `text`, `trace` and `comment` print formatted text. They take a format string and a variable list of arguments that provide data to be printed.

Supported argument types:

- constants;
- locations.

To specify locations to be printed (registers, memory), the `location(name, index)` function should be used. It takes the name of the memory array and the index of the selected element.

Supported format characters:

- `d` - decimal format;
- `x` or `X` - hexadecimal format (lowercase or uppercase letters);

- `s` - decimal format for constants and binary format for locations.

For example, the code below prints the `0xDEADBEEF` value as a constant and as a value stored in a register using different format characters:

```
prepare reg(1), 0xDEADBEEF
reg1 = location('GPR', 1)
text 'Constants: dec=%d, hex=0x%X, str=%s', 0xDEADBEEF, 0xDEADBEEF, 0xDEA
text 'Locations: dec=%d, hex=0x%X, str=%s', reg1, reg1, reg1
```

Here is how it will be printed:

```
Constants: dec=3735928559, hex=0xDEADBEEF, str=3735928559
Locations: dec=3735928559, hex=0xDEADBEEF, str=11011110101011011011111011
```

2.6 RANDOM DISTRIBUTIONS

Many tasks involve selection based on *random distribution*. The test template language includes constructs to describe ranges of possible values and their weights. To accomplish this task, the following functions are provided:

- `range(attrs)` - creates a range of values and its weight, which are described by the `:value` and `:bias` attribures. Values can be one of the following types:
  - *Single* value;
  - *Range* of values;
  - *Array* of values;
  - *Distribution* of values.

  The `:bias` attribute can be skipped which means default weight. Default weight is used to describe an even distribution based on ranges with equal weights.

- `dist(*ranges)` - creates a random distribution from a collection of ranges.

The code below illustrates how to create weighted distributions for integer numbers:

```
simple_dist = dist(
  range(:value => 0,          :bias => 25), # Value
  range(:value => 1..2,       :bias => 25), # Range
  range(:value => [3, 5, 7],  :bias => 50)  # Array
  )

composite_dist = dist(
  range(:value=> simple_dist, :bias => 80), # Distribution
  range(:value=> [4, 6, 8],   :bias => 20)  # Array
  )
```

Distributions are used in a number of test template features that will be described further in this chapter.

2.7 INSTRUCTION CALLS

The `pre`, `post` and `run` methods of a test template contain descriptions of instruction call sequences. Instructions are operations defined in ISA specifications which represent target assembler instructions. Operations can have arguments of three kinds:

- immediate value;
- addressing mode;
- operation.

Addressing modes encapsulate logic of reading or writing values to memory resources. For example, an addressing mode can refer to a register, a memory location or hold an immediate value. Operations are used to describe complex instructions that are composed of several operations (e.g. VLIW instructions). What arguments are suitable for specific instructions is specified in ISA specifications.

Arguments are passed to instructions and addressing modes in two ways:

- As *arrays*. This format is based on methods with a variable number of arguments. Values are expected to come in the same order as corresponding parameter definitions in specifications.
- As *hash maps*. This format implies that operations and addressing modes are parameterized with hash tables where the key is in the

name of the parameter and the value is the value to be assigned to this parameter.

The first way is more preferable as it is simpler and closer to the assembly code syntax. The code below demonstrates both ways (miniMIPS):

```
# Arrays
add reg(11), reg(9), reg(0)
# Hash maps
add :rd=>reg(:i=>11), :rs=>reg(:i=>9), :rt=>reg(:i=>0)
```

2.7.1 Aliases

Sometimes it is required to define *aliases* for addressing modes or operations invoked with certain arguments. This is needed to make a test template more human-readable. This can be done by defining in a test template Ruby functions that create instances with specific arguments. For example, the following code makes it possible to address registers `reg(0)` and `reg(1)` as `zero` and `at`:

```
def zero
  reg(0)
end

def at
  reg(1)
end
```

2.7.2 Pseudo Instructions

It is possible to specifify *pseudo instructions* that do not have correspondent operation in specifications. Such instructions print user-specified text and do not change the state of the reference model. The can be described using the following function: `pseudo(text)`. For example:

```
pseudo 'syscall'
```

2.7.3 Groups

Addressing modes and operations can be organized into *groups*. Groups are used when it is required to randomly select an addressing mode or an operation from the specified set.

Groups can be defined in specifications or in test templates. To define them in test templates, the following functions are used:

- `define_mode_group(name, distribution)` - defines an addressing mode
  group;
- `define_op_group(name, distribution)` - defined an operation group.

Both function take the `name` and `distribution` arguments that specify the group name and the distribution used to select its items. More information on distributions is in the Random Distribution section. *Notes*: (1) distribution items can be names of addressing modes and operations, by not names of groups; (2) it is not allowed to redefine existing groups.

For example, the code below creates an instruction group called `alu` that contains instructions `add`, `sub`, `and`, `or`, `nor`, and `xor` selected randomly according to the specified distribution.

```
alu_dist = dist(
  range(:value => 'add',                          :bias => 40),
  range(:value => 'sub',                          :bias => 30),
  range(:value => ['and', 'or', 'nor', 'xor'], :bias => 30))

define_op_group('alu', alu_dist)
```

The following code specifies three calls that use instructions randomly selected from the `alu` group:

```
alu t0, t1, t2
alu t3, t4, t5
alu t6, t7, t8
```

2.7.4 Test Situations

Test situations are associated with specific instruction calls and specify methods used to generate their input data. There is a wide range of data generation methods implemented by various data generation engines. Test situations are specified using the situation construct. It takes the situation name and a map of optional attributes that specify situation-specific parameters. For example, the following line of code causes input registers of the add instruction to be filled with zeros:

```
add t1, t2, t3 do situation('zero') end
```

When no situation is specified, a default situation is used. This situation places random values into input registers. It is possible to assign a custom

default situation for individual instructions and instruction groups with the set_default_situation function. For example:

```
set_default_situation 'add' do situation('zero') end
```

Situations can be selected at random. The selection is based on a distribution. This can be done by using the random_situation construct. For example:

```
sit_dist = dist(
  range(:value => situation('add.overflow')),
  range(:value => situation('add.normal')),
  range(:value => situation('zero')),
  range(:value => situation('random', :dist => int_dist))
  )

add t1, t2, t3 do random_situation(sit_dist) end
```

Unknown immediate arguments that should have their va-lues generated are specified using the "_" symbol. For example, the code below states that a random value should be added to a value stored in a random register and the result should be placed to another random register:

```
addi reg(_), reg(_), _ do situation('random') end
```

2.7.5 Registers Selection

Unknown immediate arguments of addressing modes are a special case and their values are generated in a slightly different way. Typically, they specify register indexes and are bounded by the lenght of register arrays. Often such indexes must be selected from a specific range taking into account previous selections. For example, registers are allocated at random and they must not overlap. To be able to solve such tasks, all values passed to addressing modes are tracked. The allowed value range and the method of value selection are specified in configuration files. Values are selected using the specified method before the instruction call is processed by the engine that generates data for the test situation. The selection method can be customized by using the mode_allocator function. It takes the allocation method name and a map of method-specific parameters. For example, the following code states that the output register of the add instruction must be a random register which is not used in the current test case:

```
add reg(_ mode_allocator('free')), t0, t1
```

Also, it is possible to exclude some elements from the range by using the exclude attribute. For example:

```
add reg(_ :exclude=>[1, 5, 7]), t0, t1
```

Addressing modes with specific argument values can be marked as free using the free_allocated_mode function. To free all allocated addressing modes, the free_all_allocated_modes function can be used.

2.8 Instruction Call Sequences

Instruction call sequences are described using block-like structures. Each block specifies a sequence or a collection of sequences. Blocks can be nested to construct complex sequences. The algorithm used for sequence construction depends on the type and the attributes of a block.

An individual instruction call is considered a primitive block describing a single sequence that consists of a single instruction call. A single sequence that consists of multiple calls can be described using the sequence or the atomic construct. The difference between the two is that an atomic sequence is never mixed with other instruction calls when sequences are merged. The code below demonstrates how to specify a sequence of three instruction calls:

```
sequence {
  add t0, t1, t2
  sub t3, t4, t5
  or  t6, t7, t8
}
```

A collection of sequences that are processed one by one can be specified using the iterate construct. For example, the code below describes three sequences consisting of one instruction call:

```
iterate {
  add t0, t1, t2
  sub t3, t4, t5
  or  t6, t7, t8
}
```

Sequences can be combined using the block construct. The resulting sequences are constructed by sequentially applying the following engines to sequences returned by nested blocks:

- combinator - builds combinations of sequences returned by nested blocks. Each combination is a tuple of length equal to the number of nested blocks.
- permutator - modifies combinations returned by combinator by rearranging some sequences.
- compositor - merges (multiplexes) sequences in a combination into a single sequence preserving the initial order of instructions calls in each sequence.
- rearranger - rearranges sequences constructed by compositor.
- obfuscator - modifies sequences returned by rearranger by permuting some instruction calls.

Each engine has several implementations based on different methods. It is possible to extend the list of supported methods with new implementations. Specific methods are selected by specifying corresponding block attributes. When they are not specified, default methods are applied. The format of a block structure for combining sequences looks as follows:

```
block(
  :combinator => 'combinator-name',
  :permutator => 'permutator-name',
  :compositor => 'compositor-name',
  :rearranger => 'rearranger-name',
  :obfuscator => 'obfuscator-name') {

  # Block A. 3 sequences of length 1: {A11}, {A21}, {A31}
  iterate { A11; A21; A31 }

  # Block B. 2 sequences of length 2: {B11, B12}, {B21, B22}
  iterate { sequence { B11, B12 }; sequence { B21, B22 } }

  # Block C. 1 sequence of length 3: {C11, C12, C13}
  iterate { sequence { C11; C12; C13 } }
}
```

The default method names are: diagonal for combinator, catenation for compositor, and trivial for permutator, rearranger and obfuscator. Such a combination of engines describes a collection of sequences constructed

as a concatenation of sequences returned by nested blocks. For example, sequences constructed for the block in the above example will be as follows: {A11, B11, B12, C11, C12, C13}, {A21, B21, B22, C11, C12, C13} and {A31, B11, B12, C11, C12, C13}

2.9 DATA

2.9.1 Configuration

Defining data requires the use of assembler-specific directives. Information on these directives is not included in ISA specifications and should be provided in test templates. It includes textual format of data directives and mappings between nML and assembler data types used by these directives. Configuration information on data directives is specified in the `data_config` block, which is usually placed in the `pre` method. Only one such block per a test template is allowed. Here is an example:

```
data_config(:text => '.data', :target => 'M') {
  define_type :id => :byte, :text => '.byte', :type => type('card',
  define_type :id => :half, :text => '.half', :type => type('card',
  define_type :id => :word, :text => '.word', :type => type('card',

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

The block takes the following parameters:

- `text` (compulsory) - specifies the keyword that marks the beginning of the data section in the generated test program;
- `target` (compulsory) - specifies the memory array defined in the nML specification to which data will be placed during simulation;
- `base_virtual_address` (optional) - specifies the base virtual address where data allocation starts. Default value is 0;
- `item_size` (optional) - specifies the size of a memory location unit pointed by address. Default value is 8 bits (or 1 byte).

To set up particular directives, the language provides special methods that must be called inside the block. All the methods share two common parameters: `id` and `text`. The first specifies the keyword to be used in a test template to address the directive and the second specifies how it will be printed in the test program. The current version of MicroTESK provides the following methods:

1. `define_type` - defines a directive to allocate memory for a data element of an nML data type specified by the type parameter;
2. `define_space` - defines a directive to allocate memory (one or more addressable locations) filled with a default value specified by the `fillWith` parameter;
3. `define_ascii_string` - defines a directive to allocate memory for an ASCII string terminated or not terminated with zero depending on the `zeroTerm` parameter.

The above example defines the directives `byte`, `half`, `word`, `ascii` (non-zero terminated string) and `asciiz` (zero terminated string) that place data in the memory array `M` (specified in nML using the `mem` keyword). The size of an addressable memory location is 1 byte.

2.9.2 Definitions

Data are defined using the `data` construct. Data definitions can be added to the test program source code file or placed into a separate source code file. There are two types of data definitions:

- **Global** - defined in the beginning of a test template and can be used by all test cases generated by the test template. Global data definitions can be placed in the root of the `pre` or `run` methods or methods called from these methods. Memory allocation is performed during inital processing of a test template (see stage 1 of template processing).
- **Test case level** - defined and used by specific test cases. Such definitions can be applied multiple times (e.g. when defined in preparators). Memory allocation is performed when a test case is generated (see stage 2 of template processing).

The `data` construct has two optional parameters:

- `global` - a boolean value that states that the data definition should be treated as global regardless of where it is defined.
- `separate_file` - a boolean value that states that the generated data definitions should be placed into a separate source code file.

Predefined methods

Here is the list of methods that can be used in `data` sections:

- `align` - aligns data by the amount n passed as an argument. By default, n means 2^n bytes. How to change this behaviour see here.
- `org` - sets data allocation origin. Can be used to increase the allocation address, but not to descrease it. Its parameter specifies the origin and can be used in two ways:
  1. As **obsolute** origin. In this case, it is specified as a constant value (`org 0x00001000`) and means an offset from the base virtual address.
  2. As **relative** origin. In this case, it is specified using a hash map (`org :delta => 0x10`) and means an offset from the latest data allocation.
- `label` - associates the specified label with the current address.

Configurable methods

Also, here is the list of runtime methods what has been configured in the `data_config` section in the previous example:

- `space` - increases the allocation address by the number of bytes specified by its argument. The allocated space is filled with the value which has been set up by the `define_space` method.
- `byte`, `half`, `word`
- `ascii`, `asciiz`

Here is an example:

```
data {
  org 0x00001000

  label :data1
  byte 1, 2, 3, 4

  label :data2
  half 0xDEAD, 0xBEEF

  label :data3
  word 0xDEADBEEF

  label :hello
  ascii  'Hello'

  label :world
  asciiz 'World'

  space 6
}
```

In this example, data is placed into memory. Data items are aligned by their size (1 byte, 2 bytes, 4 bytes). Strings are allocated at the byte border (addressable unit). For simplicity, in the current version of MicroTESK, memory is allocated starting from the address 0 (in the memory array of the executable model).

2.10 PREPARATORS

Preparators describe instruction sequences that place data into registers or memory accessed via the specified addressing mode. These sequences are inserted into test programs to set up the initial state of the microprocessor required by test situations. It is possible to overload preparators for specific cases (value masks, register numbers, etc). Preparators are defined in the pre method using the preparator construct, which uses the following parameters describing conditions under which it is applied:

- `target` - the name of the target addressing mode;
- `mask` (optional) - the mask that should be matched by the value in order for the preparator to be selected;

- arguments (optional) - values of the target addressing mode arguments that should be matched in order for the preparator to be selected;
- name (optional) - the name that identifies the current preparator to resolve ambiguity when there are several different preparators that have the same target, mask and arguments.

It is possible to define several variants of a preparator which are selected at random according to the specified distribution. They are described using the variant construct. It has two optional parameters:

- name (optional) - identifies the variant to make it possible to explicitly select a specific variant;
- bias - specifies the weight of the variant, can be skipped to set up an even distribution.

Here is an example of a preparator what places a value into a 32-bit register described by the REG addressing mode and two its special cases for values equal to 0x00000000 and 0xFFFFFFFF:

```
preparator(:target => 'REG') {
  variant(:bias => 25) {
    data {
      label :preparator_data
      word value
    }

    la at, :preparator_data
    lw target, 0, at
  }

  variant(:bias => 75) {
    lui target, value(16, 31)
    ori target, target, value(0, 15)
  }
}

preparator(:target => 'REG', :mask => '00000000') {
  xor target, zero, zero
```

```
}

preparator(:target => 'REG', :mask => 'FFFFFFFF') {
  nor target, zero, zero
}
```

Code inside the preparator block uses the target and value functions to access the target addressing mode and the value passed to the preparator.

Also, the prepare function can be used to explicitly insert preparators into test programs. It can be used to create composite preparators. The function has the following arguments:

- target - specifies the target addressing mode;
- value - specifies the value to be written;
- attrs (optional) - specifies the preparator name and the variant name to select a specific preparator.

For example, the following line of code places value 0xDEADBEEF into the t0 register:

```
prepare t0, 0xDEADBEEF
```

2.11 COMPARATORS

Test programs can include self-checks that check validity of the microprocessor state after a test case has been executed. These checks are instruction sequences inserted in the end of test cases which compare values stored in registers with expected values. If the values do not match control is transferred to a handler that reports an error. Expected values are produced by the MicroTESK simulator. Self-check are described using the comparator construct which has the same features as the preparator construct, but serves a different purpose. Here is an example of a comparator for 32-bit registers and its special case for value equal to 0x00000000:

```
comparator(:target => 'REG') {
  prepare target, value
  bne at, target, :check_failed
  nop
}
```

```
comparator(:target => 'REG', :mask => "00000000") {
  bne zero, target, :check_failed
  nop
}
```

2.12 EXCEPTION HANDLERS

Test programs can provide handlers of exceptions that occur during their execution. Exception handlers are descibed using the exception_handler construct. This description is also used by the MicroTESK simulator to handle exceptions. Separate exception handlers are described using the section construct nested into the exception_handler block. The section function has two arguments: org that specifies the handler's location in memory and exception that specifies names of associated exceptions. For example, the code below describes a handler for the IntegerOverflow, SystemCall and Breakpoint exceptions which resumes execution from the next instruction:

```
exception_handler {
  section(:org => 0x380, :exception => ['IntegerOverflow',
                                        'SystemCall',
                                        'Breakpoint']) {
    mfc0 ra, cop0(14)
    addi ra, ra, 4
    jr ra
    nop
  }
}
```

Chapter 3

Sequence Control

Sequence control.

Chapter 4

Data Control

Data control.

Chapter 5

Test Engines

5.1 Branch Engine

5.1.1 Parameters

- *branch_exec_limit* is an upper bound for the number of executions of a single branch instruction;
- *trace_count_limit* is an upper bound for the number of execution traces to be returned.

More information on the parameters is given in the âĂIJExecution Traces EnumerationâĂİ section.

5.1.2 Description

Functioning of the *branch* test engine includes the following steps:

1. construction of a *branch structure* of an abstract test sequence;
2. enumeration of *execution traces* of the branch structure;
3. concretization of the test sequence for each execution trace:
   a) construction of a *control* code;
   b) construction of an *initialization* code.

Let D be the size of the delay slot for an architecture under scrutiny (e.g., D=1 for MIPS, and D=0 for ARM).

Chapter 6

Simulator Memory Configuration

NOTE: Features described in the present document are subject to review. Mechanisms of address allocation for code and data sections initially were developed as separate features. Data were stored in physical memory, while instructions were stored in a separate internal tables using virtual addresses and were not actually stored and fetched from physical memory. Now these mechanisms are in the process of unification. For this reason, at the current stage, the scheme is a bit awkward.

Addresses used by MicroTESK to allocate code and data sections are configured using the following settings:

- `base_virtual_address` (by default, equals 0);
- `base_physical_address` (by default, equals 0).

The settings are initialized in the initialize method of a template in the following way:

```
def initialize
  super
  # Memory-related settings
  set_option_value 'base-virtual-address', 0x00001000
  set_option_value 'base-physical-address', 0x00001000
end
```

In addition, the configuration of physical memory and data allocation constructs is set up using the `data_config` construct. For example:

```
data_config(:text => '.data',
            :target => 'M',
            :base_virtual_address => 0x40180000) {
  ...
}
```

The construct specifies memory array used as physical memory (`:target`) and base virtual address used for data allocation (`:base_virtual_address`).

Schemes of address allocation for code and data sections are described in detail further in this document.

6.1 CODE SECTIONS

Memory modeling for code sections is optional. It is enabled using the `-fetch-decode-enabled` (`-fde`) option. If it is not enabled, instructions are not stored in physical memory and have no physical addresses. They are stored in special tables and accessed using their virtual addresses.

6.1.1 Disabled memory modeling

When memory modeling is disabled the scheme is the following. MicroTESK starts address allocations for code sections at `base_virtual_address`. The VA for the current allocation is calculated as VA for the previous code allocation + size of previous code allocation.

Allocation address can be modified using the org directive. MicroTESK allows specifying relative and absolute origins:

- Relative origin: `org :delta => n`, $VA = VA + n$;
- Absolute origin: `org n`, $VA = base\_virtual\_address + n$.

Addresses can be aligned using the align directive. By default, align n means align at the border of $2 \times n$ bytes. When code is simulated, no fetches are performed. Instead, MicroTESK extracts instructions stored in special tables using their virtual addresses.

6.1.2 Enabled memory modeling

When memory modeling is enabled, MicroTESK stores binary representation of instructions in physical memory. Also, objects describing instructions are stored in special tables in the same way as when memory modeling is disabled. The only difference is that now they are indexed using physical addresses.

Physical addresses used for allocation are calculated using the following formula:

$$PA = base\_physical\_address + origin,$$

where $origin = VA - base\_virtual\_address$

When code is simulated, instruction fetches are performed. Logic of fetches is described in MMU specifications (address translation and accesses to cache buffers). For each fetch, MicroTESK gets binary representation of the instruction and its physical address (written to the `MMU_PA` register). First, it tries to extract the instruction from the table using its physical address. If the instruction is not found (execution of self-modifying code), it tries to decode its binary representation. If decoding fails, the `invalid_instruction` operation defined in the nML specification is called (which throws a corresponding exception).

6.2 DATA SECTIONS

There two use cases of addressing data which are handled using separate mechanisms:

- Data is allocated in memory using the `data...` construct.
- Data is read or written to memory using load/store instructions.

Loads and stores use the MMU model included into the MicroTESK simulator, which involves address translation and accesses to cache buffers according to the MMU specifications.

For data allocation, MMU logic is not used.

Data are placed directly to physical memory starting from `base_physical_address`. Address allocation is controlled by the org and align directives. They work similarly to the one used for code sections, but operate with physical addresses:

- Relative origin: `org :delta => n`, $PA = PA + n$;
- Absolute origin: `org n`, $PA = base\_physical\_address + n$.

Labels in data sections are assigned virtual addresses. To do this, PA is translated into VA. Address translation does not use the MMU model. Instead, it uses a simplified scheme based on settings that works as follows:

$$VA = base\_virtual\_address + (PA - base\_physical\_address).$$

NOTE: Allocation addresses VA and PA are tracked separately for data and code sections. Therefore, it is required to take care to avoid address conflicts.

MicroTESK allows using different base addresses for code and data allocations. Data allocations can be assigned a separate base virtual address. This can be done in the following way (see the pre method of the base test template):

```
data_config(:text => '.data',
            :target => 'M',
            :base_virtual_address => 0x00002000) {
  ...
}
```

This VA is translated into PA using the simplified address translation scheme ($PA = base\_physical\_address + (VA - base\_virtual\_address)$), and the result is used as base PA for data allocations.

Example: `min_max.rb`.

NOTE: This is a simplified example. It considers PA to be equal VA. This assumption is made because the address translation mechanism is currently disabled in MMU specifications. Anyway, this should be enough to illustrate the above-described principles.

6.2.1 Settings

Setting values are as follows:

- `base_virtual_address` and `base_physical_address` use the default value 0 (are not initialized).
- `base_virtual_address` for data sections is specified as $0x40180000$ (see code below).

```
data_config(:text => '.data',
            :target => 'M',
            :base_virtual_address => 0x40180000) {
  ...
}
```

6.2.2 Prologue

Starting VA for the test programŠs main code is specified as $0x2000$ (calculated as $base\_virtual\_address + origin = 0 + 0x2000$):

```
text '.text'
text '.globl _start'
movz x0, vbar_el1_value, 0
msr vbar_el1, x0
bl :_start
org 0x2000
label :_start
```

The code is allocated in the simulatorŠs physical memory in the following way (from MicroTESK debug output):

```
0x00000000 (PA): nop (0xD503201F)
0x00000004 (PA): movz x0, #0x50, LSL #0 (0xD2800A00)
0x00000008 (PA): msr vbar_el1, x0 (0xD518C000)
0x0000000c (PA): movz x0, #0x700, LSL #0 (0xD280E000)
0x00000010 (PA): msr VBAR_EL2, x0 (0xD51CC000)
0x00000014 (PA): movz x0, #0xd50, LSL #0 (0xD281AA00)
0x00000018 (PA): msr VBAR_EL3, x0 (0xD51EC000)
0x0000001c (PA): bl _start (0x94000000)
```

6.2.3 Data section

Data sections starts at $VA = 0x40180000$ ($base\_virtual\_address + origin = 0x40180000 + 0$).

VA of `:data` is $0x40180000$ and VA of `:end` is $0x40180060$:

```
data {
  org 0x0
  label :data
  dword rand(0, 0xffff),
        rand(0, 0xffff),
```

```
        rand(0, 0xffff),
        rand(0, 0xffff),
        rand(0, 0xffff),
        rand(0, 0xffff),
        rand(0, 0xffff),
        rand(0, 0xffff),
        rand(0, 0xffff)
  label :end
  space 1
}
```

The data is allocated in the simulatorŠs physical memory in the following way (from MicroTESK debug output):

```
0x40180000 (PA): .org 0x0
0x40180000 (PA): data:
0x40180000 (PA):
  .dword 0x000000000000C64E, 0x0000000000002658,
         0x0000000000004D63, 0x000000000000DCC6,
         0x00000000000028E6, 0x00000000000043F2,
         0x0000000000008916, 0x00000000000005FD,
         0x000000000000C60F
0x40180048 (PA): end:
0x40180048 (PA): .space 1
```

6.2.4 Main code

Starting VA is set by the `org.` directive to $0x2000$ ($base\_virtual\_address + origin = 0 + 0x2000$). The PA is equal to VA. The code is allocated in the simulatorŠs physical memory in the following way (from MicroTESK debug output):

```
0x00002000 (PA): adr x0, data          (0x10000000)
0x00002004 (PA): adr x1, end           (0x10000001)
0x00002008 (PA): mov x2, x0            (0xAA0003E2)
0x0000200c (PA): ldar x6, [x2, #0]     (0xC8DFFC46)
0x00002010 (PA): mov x3, x6            (0xAA0603E3)
0x00002014 (PA): mov x4, x6            (0xAA0603E4)
0x00002018 (PA): cmp x2, x1, LSL #0    (0xEB01005F)
0x0000201c (PA): b.ge exit_for_0000    (0x5400000A)
0x00002020 (PA): ldar x6, [x2, #0]     (0xC8DFFC46)
```

```
0x00002024 (PA): cmp x6, x4, LSL #0       (0xEB0400DF)
0x00002028 (PA): b.gt new_max_value_0000 (0x5400000C)
0x0000202c (PA): cmp x6, x3, LSL #0       (0xEB0300DF)
0x00002030 (PA): b.lt new_min_value_0000 (0x5400000B)
0x00002034 (PA): add x2, x2, #8, LSL #0  (0x91002042)
0x00002038 (PA): b for_0000               (0x14000000)
0x0000203c (PA): mov x4, x6               (0xAA0603E4)
0x00002040 (PA): b next_0000              (0x14000000)
0x0 002044 (PA): mov x3, x6               (0xAA0603E3)
0x00002048 (PA): b next_0000              (0x14000000)
```

Bibliography

[1] M. Freericks. *The nML Machine Description Formalism*. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993.

[2] *Ruby site* – `http://www.ruby-lang.org`

[3] Flanagan D., Matsumoto Y. *The Ruby Programming Language*. OâĂŹReilly Media, Sebastopol, 2008.

[4] *JRuby site* – `http://jruby.org`