HAIL Language Specification and User Guide

External Technical Report Number DCL-TR-2005-0006

Wanghong Yuan, Jun Sun, Nayeem Islam
DoCoMo USA Labs
181 Metro Dr, Suite 300, San Jose, CA 95110
{yuan, jsun, nayeem}@docomolabs-usa.com

September 12, 2005

1. Overview 2
2. Language Semantics 3
   2.1 Register map description 3
   2.2 Address space description 5
   2.3 Device instantiation 7
   2.4 Invariants 7
   2.5 Others 8
3. Run-time interface 8
   3.1 Interfaces exported to driver 8
   3.2 Interfaces provided by driver 10
4. Compiler Usage 11
5. Implementation Status 12
   5.1 Current status 12
   5.2 Known bugs 12
Appendix: Syntax and Lexical Notes 13
   A. HAIL syntax 13
   B. HAIL lexical notes 17

1. Overview

HAIL (Hardware Access Interface Language) is a domain specific language for device drivers to access device registers and manipulate register bit fields. HAIL allows driver developers to describe the attributes and properties of devices and their address spaces (or buses), rather than writing low level accessing code. From the attribute specification, the HAIL compiler generates the accessing code (in the form of inline C functions) with optional debugging code. Device drivers can include and use the generated code. Figure 1 illustrates the HAIL approach.

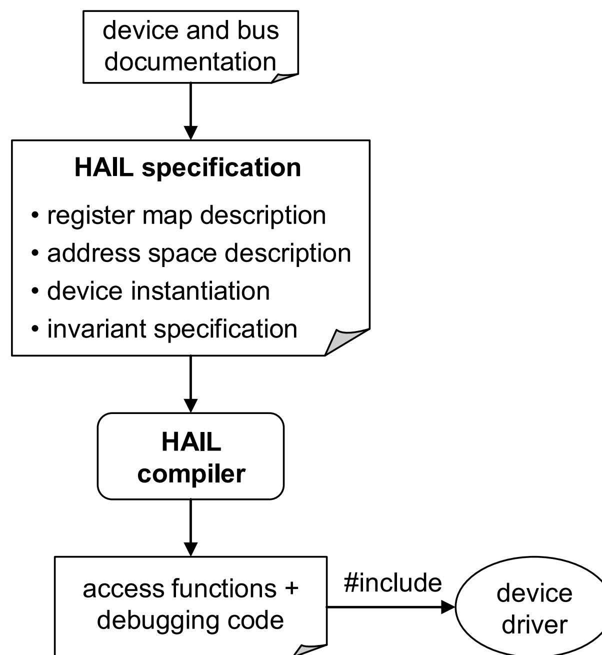

Figure 1 - Overview of HAIL

In particular, the HAIL specification consists of four parts: (1) *register map description*, which describes the various device registers and bit fields, (2) *address space description*, which describes the address spaces (or buses) where the device live and the mechanisms for accessing the spaces, (3) *device instantiation*, which describes the actual instantiation of the device in the particular system, and (4) *invariant specification*, which describes the constraints on accessing the device. The HAIL specification is usually translated from documentations on the device and its buses.

The HAIL compiler translates the specification into C code for accessing registers. In a simplified view, the generated code is a list of get_xxx()/set_xxx() for all registers and bit fields, put together in a C header file. Optionally the generated code can have run-time debugging code which catches any violation to invariant specification.

The device driver includes the generated header and uses the get_xxx()/set_xxx() function to access the device, without concerning about the underlying device base addresses, bus attributes, data width, endianess, etc.. Additionally, device driver can manipulate bit fields directly through the generated code. Consequently the driver can get rid of a large number of bit manipulations.

Refer to our paper [1] for details on the concepts of HAIL. In this manual, we describe the HAIL language specification and the compiler. Specifically, Section 2 describes semantics of HAIL language. Section 3 describes the run-time interfaces between the HAIL compiler and the device driver. Section 4 illustrates how to use the HAIL compiler. Section 5 describes the current status of HAIL implementation. Finally, the appendix gives the syntax and lexical notes of the HAIL language.

2. Language Semantics

As we mentioned above, the HAIL specification consists of four parts: *register map description*, *address space description*, *device instantiation*, and *invariant specification*. The appendix gives the detailed language syntax and lexical notes. This section explains the language semantics for the above four parts.

2.1 Register map description

Registers are the software interface of a device. A set of logically coherent registers composes a *register map*. A device can have one or more register maps. For example, a typical PCI device may have three register maps, one each for PCI configuration, PCI memory, and PCI IO space. Each register map consists of a set of registers. Each register has its attributes, such as name and access (read only, write only, and read-write), and a set of bit fields, which in turn have their own attributes. To simplify the specification, each register map has default register options that apply to all registers and bit fields by default. Figure 2 illustrates the structure of a device specification.

```
DEVICE name {
      /* one register map */
      REGISTER_MAP name {
            .= default {
               attributes
             }

            /* one register */
            .= offset {
                  register attributes such as name and access
                  /* one bit field */
                  [startbit:endbit]: bit attributes
                  …
                  /* another bit field */
                  [startbit:endbit]: bit attributes
```

```
                }
                …
                /* another register */
                .= offset {
                }
        }
        …
        /* another register map */
        REGISTER_MAP name {
        }
}
```

Figure 2 – Structure of device specification

If a device is a revision of another device, the device head is

```
DEVICE name DERIVED_FROM athoer_device_name { }
```

In this case, the new device specification copies the specification from another device and can change the specification by removing and adding a register.

We next describe the attributes of registers and bit fields in detail. Registers and bit fields have the following attributes (these attributes apply to both register and bit field, unless specified otherwise):

- offset – the offset, in number of bytes, relative to the base address of the register map. *Only applicable to registers*
- name – an identifier consisting of alphabets, underscore, and numbers
- size – how many bytes the register has. *Only applicable to registers*
- reserved – For a reserved register, the device driver should not access it. For a reserved bit field, the device driver should not use its value (that is, its read_value is dont_matter).
- access – can be read-only, write-only or read-write
- enum_list – a set of possible register values. Values not in the set are illegal. *Only applicable to bit fields*.
- read_value – can be
    - dont_matter - the driver should not use the read value of the register
    - fixed – always return the fixed value when read
    - static - multiple consecutive reads return the same value
    - volatile – multiple consecutive reads may return different value
    - volatile_se – volatile and with side effect (for example, read an interrupt status register may clear some of its bits)
- default_write – can be
    - dont_matter – whatever value can be written to the register
    - fixed – always write a fixed value
    - dont_change – The value of bit should not change when writing another bit in the same register.
    - explicit – The value of the bit must be specified when writing other bits in the same register. That is, it should not write other bits alone.

- starting bit and ending bit - the starting bit and ending bit of a bit field. *Only applicable to bit fields*

The ***default attributes*** can contain most of the above attributes except

- offset
- name
- enum_list

The ***default attributes*** has an additional attribute:

- stride – the distance, in number of bytes, between adjacent registers. If a register does not specify its offset, its offset is the sum of the stride and the offset of the previous register in the map

Note that

- The attributes of a bit field override those of a register, which in turn override the default attributes.
- If a register has read_only access, it should NOT have default_write. Likewise, if a register has write_only access, it should NOT have read_value.
- In a register, if a bit filed has dont_change in default_write, there is a WARNING if another bit filed has volatile in read_values and an ERROR if another bit filed has volatile_se in read_values.
- If a register has write_only access and one of its bit fields has dont_change in default_write, there is a WARNING since one cannot read the register to keep the bit field's value.

2.2 Address space description

In a system a device is always attached to a bus, or *address space*, which dictates how the CPU accesses the device. There are two kinds of address spaces, *memory mapped space* and *gated space.* The memory mapped space has the following structure:

```
ADDRESS_SPACE name {
      data_width=[1,2,…]
      address_size=<number of addressing bits>
      endianness=big|little
 }
```

Figure 3 – Structure of memory mapped space specification

Each memory mapped address space has the following three attributes:

- data_width – The number of bytes to access data. It is 1, 2, 4, 8, 16, or their combinations
- endianness – endian format used when data_width can be more than one byte

- address_size – bits used for address, which determines the addressing range. For example, an address space with 16bit addressing bits would have address_size=16, and there is a total of 64KB addressable bytes in this space.

A special memory mapped address space is CPU virtual space. The `CPU_virtual is predefined address space that has the same endianness as CPU, address_size being the same as int in C language, data_width is equivalent to various size of memory units CPU can directly fetch and write.` If a memory mapped space is not the CPU virtual space that the CPU can access directly, the memory mapped space needs to be mapped to the CPU virtual space. The *address map* is like the following:

```
ADDRESS_MAP name => name {
      permanet_= yes|no
      base_address = runtime|static(number)|literal (identifier)
      window_start = runtime|static(number)|literal (identifier)
      window_size = number of bytes
      endian_swap = yes | no
      map_function = an identifier, which is function name
 }
```

Figure 4 – Structure of address map specification

Each *address map* has the following attributes:

- permanent_map – boolean value that indicates if the map is permanent or not
- base_address – base address of the source space (see below for options)
- window_start – the address of the starting mapping part
- window_size – mapping window size
- endian_swap – the mapping automatically performs endian swap or not
- map_function – name for function that perform the mapping

The address value for base address and window start can has three types:

- runtime – generated in runtime. In this case, a global variable is generated. The driver is expected to set the value of the global variable before the first usage of HAIL generated access function.
- static – a fixed constant number
- literal – an identifier, which is typically a global variable or macro in the driver or OS.

Another type of address space is gated space like follows:

```
GATED_SPACE name {
      Data width
      Access function
 }
```

Figure 5 – Structure of gated space specification

A typical gated space is usually accessed through a pair of address/data registers such as many implementations of i2c bus. Each gated space has the following two attributes:

- data_width – The number of bytes to access data. It is 1, 2, 4, 8, 16, or their combinations
- access_function – function name for accessing registers.

For example, if the data_width is {1, 2} and the access_function is arcom_i2c, the HAIL compiler generates arcom_i2c_get_1 and arcom_i2c_set_1 for accessing one-byte registers and arcom_i2c_get_2 and arcom_i2c_set_2 for accessing two-byte registers. Functions arcom_i2c_get_1, arcom_i2c_set_1, arcom_i2c_get_2, and arcom_i2c_set_2 are external functions to HAIL.

2.3 Device instantiation

Device instantiation defines an instance of a device by associating every device's register map to a specific address space. When associating a register map with an address space, the instantiation specifies a base address for the register map, which can be a statically fixed address, a literal (which is defined as a macro or a variable by the driver and OS environment), or a HAIL variable (which needs to be set by driver at run-time).

```
INSTANTIATE name AS device_name {
      MULTI_INSTANCE

      /* for each register map of the device */
      map name => space name {
            base address
            stride
      }
 }
```

Figure 5 – Structure of instantiation specification

An instantiation has the following attributes:

- multi_instance – A Boolean value that indicate if the device can be instantiated multiple times or not
- register map => address space – Associate each register map of the device to an address space. The association itself has a base address and stride (the stride in the distance, in bytes, between registers in the register map). The stride value overwrites the value specified in the register map if it has one.

2.4 Invariants

Invariants are constraints that the device must satisfy at runtime. In HAIL, invariant specification consists of a list of clauses. Each clause consists of a list of events connected by sequence connectors. The following sequence connectors are defined:

- A =i> B : event A immediately followed by event B (no other event can happen in-between)
- A =o> B : event A optionally followed by event B
- A =e> B : event A eventually followed by event B, implying that B must follow A
- A =r> B : event A resets event B

Note that the same event cannot appear more than once in a sequence.

Each event has a set of actions and an optional pre-condition and post-condition for these actions.

- Each action is a read, write, or read/write of a register.
- Pre-condition and post-condition are Boolean expressions that assert values of register access:
    - R(identifier)- Read value of register or bit field
    - M(identifier)- the last read value of register or bit field. Typically applied to volatile side effect registers
    - W(identifier)- the value is about to write into the register or bit field if the term appears in the pre-condition and the identifier appears in the actions, or the value of the last write otherwise.

2.5 Others

A HAIL specification must have one and only one device specification and one device instantiation of the same device. It can have zero or more address spaces and invariants. All address spaces must be mapped into native address space through a series of mapping.

3. Run-time interface

The HAIL compiler translates the HAIL specification into low-level access code and debugging code, which can be used by device drivers. Furthermore, to support the generated code, the device driver needs to provide some functions and macros for gated address space and for debugging. We next describe the runtime interfaces in detail.

3.1 Interfaces exported to driver

The generated code consists of a set of access functions and macros for the device driver to use.

3.1.1 Access functions

For each register with read-write access, the HAIL compiler generates the following access functions:

- static inline rtype get_regname (void)
- static inline void set_regname (rtype reg_val)

where regname is the name of the register and rtype is the C data type of the register. rtype is

- u_int32_t if the register size is 4 bytes
- u_int16_t if the register size is 2 bytes
- u_int8_t if the register size is 1 bytes

For each bit field with read-write access, the HAIL compiler generates the following access functions:

- static inline type get_regname_bitname (void)
- static inline void set_regname_bitname (type reg_val)
- static inline type mem_get_regname_bitname (rtype reg_val)
- static inline type mem_set_regname_bitname (rtype *reg_val, btype bit_val)

where bitname is the name of the bit field, regname is the name of the register, rtype is the C data type of the register, and btype is the C data type of the bit field. btype is

- u_int32_t if the bit filed size is greater than 16 bits but no more than 32 bits
- u_int16_t if the bit filed size is greater than 8 bits but no more than 16 bits
- u_int8_t if the bit filed size is no more than 8 bits

Function mem_get_regname_bitname(rtype reg_val) returns the value of the bit field from reg_val, the memory cached value of the register without accessing the register. For example, if one needs to get values of two different bit fields of a register, the common way is

```
reg_val  = get_regname();
bit1_val = mem_get_regname_bit1name(reg_val);
bit2_val = mem_get_regname_bit2name(reg_val);
```

Likewise, function mem_set_regname_bitname(rtype *reg_val, btype bit_val) sets the value of the bit field in the memory cached value of the register. For example, if one needs to write multiple bit fields of a register, the common way is

```
rtype reg_val;
mem_set_regname_bit1name(&reg_val, bit1_val);
mem_set_regname_bit2name(&reg_val, bit2_val);
set_regname(reg_val);
```

Note that is the register has read-only access, only get_regname() is generated. If the register has write-only access, only set_regname() is generated. Similarly, if a bit filed has read-only access, only get_ regname_bitname() and mem_get_ regname_bitname() are generated. If the bit field has write-only access, only set_ regname_bitname() and mem_set_ regname_bitname() are generated.

3.1.2 Macros

If a register or bit field has named and enumerated value options, the HAIL compiler also generates a set of macros for the enumerated values. For example, for register IIR with the following spec,

```
.=0x08 {
     IIR;
```

```
        access = RO;
        read_value = volatile;
        [7:6]: FIFOES; enum = {NON_FIFIO=0b00, FIFO=0b11};
        [5]: EOC;
        [4]: ABL;
        [3]: TOD;
        [2:1]: ID; enum = {MSI=0b0, THRI=0b01, RDI=0b10, RLSI=0b11};
        [0]: NO_INT;
}
```

the HAIL compiler generates macros for bit field FIFOES and ID

```
/* value options for bit IIR_FIFOES */
#define IIR_NON_FIFIO   0
#define IIR_FIFO  3
/* value options for bit IIR_ID */
#define IIR_MSI   0
#define IIR_THRI  1
#define IIR_RDI   2
#define IIR_RLSI  3
```

The HAIL compiler also generates the following macros for each bit field:

- regname_bitname_MASK – the mask of the bit field
- regname_bitname_SHIFT – the shift of the bit field
- regname_bitname_VAL(x) – the value of the bit field from the parameter x. This is similar to mem_get_regname_bitname() function.

For the bit field FIFEOS in the above register, the following macros are generated

```
#define IIR_FIFOES_MASK 0xC0
#define IIR_FIFOES_SHIFT      0x6
#define IIR_FIFOES_VAL(x) (((x) << IIR_FIFOES_SHIFT) & \
                           IIR_FIFOES_MASK)
```

3.2 Interfaces provided by driver

The device driver should provide the following to HAIL generated code

- Access functions for gated space. For example,

  ```
  gated_space i2c {
        data_width = {1, 2};
        access_function = arcom_i2c;
  };
  ```

  The device driver needs to provide
  - `u_int8_t arcom_i2c_get_1(void);`
  - `void arcom_i2c_set_1(u_int8_1 val);`
  - `u_int16_t arcom_i2c_get_2(void);`
  - `void arcom_i2c_set_2(u_int16_1 val);`

- Literal address value. If the base address of an address space is literal(baseaddr)

the device driver should define a global variable *baseaddr* and initialize it.

In addition, there are some OS dependent macros needed when HAIL compiler generates debugging code. We pre-defined them for Linux and NetBSD in the hail-os-dep.h. For any other OSes you will need to supply the similar macros.

```
#if defined(__linux__)

#define HAIL_FAIL(info) printk("%s\n", info)
#define HAIL_ASSERT(x) if(!(x)) printk("assert fail %s\n", #x)
#define HAIL_DISABLE_INTERRUPT() local_irq_save()
#define HAIL_ENABLE_INTERRUPT(s) local_irq_restore(s)

#elif defined(__NetBSD__)

#include <sys/malloc.h>
#include <machine/intr.h>
#include <lib/libkern/libkern.h>

#define HAIL_FAIL(info) printf("%s\n", info)
#define HAIL_ASSERT(x) KASSERT(x)
#define HAIL_DISABLE_INTERRUPT() splhi()
#define HAIL_ENABLE_INTERRUPT(s) splx(s)

#endif
```

4. Compiler Usage

The HAIL compiler can be started as

> *hail [options] hail_spec_file*

The options are

- -checkparam: check parameters from device driver. If a register or a bit field has a fixed write value or a set of enumerated values, this option enables the generated code check if the parameter provided by device driver is valid or not.
- -checkspec: check the consistency of the HAIL spec file. This option enables HAIL generated code checks if the specification or the hardware is correct or not. For example, a bit field X in register Y has a fixed read value, the get_Y_X() checks if the value of the bit field X is the same the fixed value in the specification.
- -invariant: generate debugging code for logical and sequential invariants.
- -prefix string: add a prefix, string, to all generated macros to avoid conflicts. For the macro examples in 5.1.2, if –prefix MY is specified, the macros are

  ```
  #define MY_IIR_FIFOES_MASK 0xC0
  #define MY_IIR_FIFOES_SHIFT      0x6
  ```

```
#define MY_IIR_FIFOES_VAL(x) (((x) << MY_IIR_FIFOES_SHIFT) & \
                              MY_IIR_FIFOES_MASK)
```

5. Implementation Status

5.1 Current status

The current HAIL compiler supports all language syntax. HAIL compiler support almost all semantics except

- address space
  - address_map: HAIL DOES NOT support address map. Instead, HAIL assumes that memory mapped address spaces are directly mapped to the CPU virtual space.
  - addr_value: HAIL supports only static and literal address value, and DOES NOT support runtime value.

- Instantiatition
  - multi_instance: HAIL DOES NOT multiple instance and only supports single instance (that is, MULTI_INSTANCE = NO).
  - addr_value: HAIL supports only static and literal address value, and DOES NOT support runtime value.

5.2 Known bugs

1. Name conflict: HAIL compiler can NOT solve the name conflicts. For example, if a register has name A and has a bit filed B_C, the following functions are generated

   get_A_B_C()

   If another register has name A_B and has a bit filed C, HAIL also generates

   get_A_B_C()

References

1. J. Sun, W. Yuan, M. Kallahalla, and N. Islam, HAIL: A Language for Easy and Correct Device Access, To appear in *Proc. of ACM Conference on Embedded Software*, Jersey City, NJ, Sep 2005.

Appendix: Syntax and Lexical Notes

A. HAIL syntax

```
hail_list: /* empty */
   | hail_list hail
   ;

hail: address_space
   | address_map
   | gated_space
   | device
   | instantiate
   | invariant
   ;

/* ********************************* */
/* part 1: address space             */
/* ********************************* */

address_space: address_space_head '{' as_attr_list '}' ';'
    ;

address_space_head: ADDRESS_SPACE IDENTIFIER
    ;

as_attr_list: /* empty */
    | as_attr_list as_attr
    ;

as_attr: DATA_WIDTH '=' '{' number_list '}'';'
    | ADDRESS_SIZE '=' NUMBER ';'
    | ENDIANNESS '=' LITTLE ';'
    | ENDIANNESS '=' BIG ';'
    ;

number_list: NUMBER
    | number_list ',' NUMBER
    ;

address_map:  ADDRESS_MAP  IDENTIFIER  FOLLOW  IDENTIFIER  '{'  am_attr_list
'}' ';'
    ;

am_attr_list: /* empty */
    | am_attr_list am_attr
    ;

am_attr: PERMANET_MAP '=' BOOL ';'
    | BASE_ADDRESS '=' addr_value ';'
    | WINDOW_START '=' addr_value ';'
    | WINDOW_SIZE  '=' number ';'
    | ENDIAN_SWAP  '=' BOOL ';'
    | MAP_FUNCTION '=' IDENTIFIER ';'
    ;
```

```
addr_value: RUNTIME
    | STATIC '(' NUMBER ')'
    | LITERAL '(' IDENTIFIER ')'
    ;

gated_space: gated_space_head '{' ag_attr_list '}' ';'
    ;

gated_space_head: GATED_SPACE IDENTIFIER
    ;

ag_attr_list: /* empty */
    | ag_attr_list ag_attr
    ;

ag_attr: DATA_WIDTH '=' '{' number_list '}' ';'
    | ACCESS_FUNCTION '=' IDENTIFIER ';'
    ;

/* ******************************** */
/* part 2: instantiate              */
/* ******************************** */

instantiate: instantiate_head '{' instantiation_attr_list '}'
    ;

instantiate_head: INSTANTIATE IDENTIFIER AS IDENTIFIER
    ;

instantiation_attr_list: /* empty */
    | instantiation_attr_list instantiation_attr
    ;

instantiation_attr: MULTI_INSTANCE '=' BOOL ';'
    | inst_attr_head  '{' asso_attr '}' ';'
    ;

inst_attr_head: IDENTIFIER FOLLOW IDENTIFIER
    ;

asso_attr: BASE_ADDRESS '=' addr_value ';'
   | BASE_ADDRESS '=' addr_value ';' STRIDE '=' NUMBER ';'
   ;

/* ******************************** */
/* part 3: register map             */
/* ******************************** */

optional_semicolon: /* empty */
     | ';'
     ;

device: device_head '{' register_map_list '}' optional_semicolon
    ;

device_head: DEVICE IDENTIFIER
    | DEVICE IDENTIFIER DERIVED_FROM IDENTIFIER
```

```
    ;

register_map_list: /* empty */
    | register_map_list register_map
    ;

register_map:   map_head   '{'   default_reg_attr   register_list   '}'
optional_semicolon
    ;

map_head: REGISTER_MAP IDENTIFIER
    ;

default_reg_attr: /* empty */
    | default_reg_head '{' stride_option attribute_list reg_bit_list '}'
optional_semicolon
    ;

default_reg_head: '.''=' DEFAULT
    ;

stride_option: /* empty */
    | STRIDE '=' NUMBER ';'
    ;

register_list: /* empty */
    | register_list register
    ;

register: reg_head '{' attribute_list reg_bit_list '}' optional_semicolon
    | REMOVE IDENTIFIER ';'
    ;

reg_head: '.' '=' NUMBER
    | '.' '='
    ;

attribute_list: /* empty */
    | attribute_list attribute
    ;

attribute: RESERVED ';'
    | NAME '=' IDENTIFIER ';'
    | IDENTIFIER ';' /* shorthand for the above */
    | SIZE '=' NUMBER ';'
    | access_option ';'
    | read_value ';'
    | default_write ';'
    | ENUM '=' '{' enum_list '}'';'
    ;

access_option : ACCESS '=' READ_ONLY
    | ACCESS '=' WRITE_ONLY
    | ACCESS '=' READ_WRITE
    ;

read_value : READ_VALUE '=' DONT_MATTER
```

```
     | READ_VALUE '=' FIXED '(' NUMBER ')'
     | READ_VALUE '=' VOLATILE
     | READ_VALUE '=' VOLATILE_SE
     | READ_VALUE '=' STATIC
     ;

default_write : DEFAULT_WRITE '=' DONT_MATTER
     | DEFAULT_WRITE '=' FIXED '(' NUMBER ')'
     | DEFAULT_WRITE '=' DONT_CHANGE
     | DEFAULT_WRITE '=' NO_DEFAULT
     ;

enum_list: enum_item
     | enum_list ',' enum_item
     ;

enum_item: NUMBER
     | IDENTIFIER '=' NUMBER
     ;

reg_bit_list: /* empty */
     | reg_bit_list reg_bit
     ;

reg_bit: reg_bit_head attribute_list
     ;

reg_bit_head: '[' NUMBER ']' ':'
     | '[' NUMBER ':' NUMBER ']' ':'
     ;

/* ******************************** */
/* part 4: invariant                */
/* ******************************** */

invariant: INVARIANT '{' sequence_list '}' optional_semicolon
     ;

sequence_list: /* empty */
     | sequence_list sequence ';'
     ;

sequence: event
     | sequence connector event
     ;

connector: IMMEDIATE_FOLLOW
     | OPTION_FOLLOW
     | EVENTUAL_FOLLOW
     | RESET
     ;

event: condition '{' actions '}' condition
     ;

actions: action
    | actions ',' action
```

```
    ;

action: READ '(' IDENTIFIER ')'
    | WRITE '(' IDENTIFIER ')'
    | READ_WRITE '(' IDENTIFIER ')'
    ;

condition: /* empty*/
    | '(' bool_exp ')'
    ;

bool_exp: bool_item
    | bool_exp AND bool_exp
    | bool_exp OR bool_exp
    | NOT bool_exp
    | '(' bool_exp ')'
    ;

bool_item: bool_factor
    | bool_factor EQUAL bool_factor
    | bool_factor NOTEQUAL bool_factor
    | bool_factor LESS bool_factor
    | bool_factor GREATER bool_factor
    | bool_factor NOLESS bool_factor
    | bool_factor NOGREATER bool_factor
    ;

bool_factor: NUMBER
    | IDENTIFIER
    | READ '(' IDENTIFIER ')'
    | WRITE '(' IDENTIFIER ')'
    | MEM_READ '(' IDENTIFIER ')'
    ;
```

B. HAIL lexical notes

1) Keywords

Keyword	representation in syntax tree
as	AS
address_space	ADDRESS_SPACE
address_map	ADDRESS_MAP
gated_space	GATED_SPACE
instantiate	INSTANTIATE
data_width	DATA_WIDTH
address_size	ADDRESS_SIZE
endianness	ENDIANNESS
little	LITTLE
big	BIG
permanet_map	PERMANET_MAP
base_address	BASE_ADDRESS
window_start	WINDOW_START
window_size	WINDOW_SIZE
endian_map	ENDIAN_SWAP
map_function	MAP_FUNCTION
runtime	RUNTIME
static	STATIC

```
literal                 LITERAL
access_function         ACCESS_FUNCTION
instance_of             INSTANCE_OF
multi_instance          MULTI_INSTANCE
device                  DEVICE
register_map            REGISTER_MAP
default                 DEFAULT
stride                  STRIDE
reserved                RESERVED
name                    NAME
size                    SIZE
access                  ACCESS
read_value              READ_VALUE
default_write           DEFAULT_WRITE
idempotent              IDEMPOTENT
volatile                VOLATILE
volatile_se             VOLATILE_SE
fixed                   FIXED
dont_matter             DONT_MATTER
dont_change             DONT_CHANGE
no_default              NO_DEFAULT
derived_from            DERIVED_FROM
enum                    ENUM
remove                  REMOVE
ro|RO                   READ_ONLY
wo|WO                   WRITE_ONLY
rw|RW                   READ_WRITE
invariant               INVARIANT
"=>"                    FOLLOW
"=i>"                   IMMEDIATE_FOLLOW
"=o>"                   OPTION_FOLLOW
"=e>"                   EVENTUAL_FOLLOW
"=r>"                   RESET
m|M                     MEM_READ
r|R                     READ
w|W                     WRITE
"&&"                    AND
"||"                    OR
"!"                     NOT
"=="                    EQUAL
"!="                    NOTEQUAL
"<"                     LESS
">"                     GREATER
">="                    NOLESS
"<="                    NOGREATER
```

2) Numbers and Identifiers

- *Identifier*: start with a-zA-Z followed by 0-9a-zA-Z and underscore
- *Decimal number*: consist of digital numbers 0-9
- *Hexicial number*: start with 0x followed by 0-9a-fA-F
- *Binnary number:* start with 0b followsed by 0-1

3) Comments

HAIL accepts C-like comments that begin with “/*” and end with “*/”.