MetaHDL Reference Manual

A user guide to syntax and compiler usage

by mengxin@vlsi.zju.edu.cn

Version 0.1

built on 15:42, October 10, 2008

Institute of VLSI,
College of Electrical Engineering,
Zhejiang University

Revision History

Revision	Date	Description	Author
0.1	7/10/2008	Self Review to refine text.	MENG Xin
0.0	7/9/2008	Initial version.	MENG Xin

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

MetaHDL is a *Hardware Description Language* (HDL) aims at synthesizable digital VLSI designs (commonly known as *Register Transfer Level* (RTL) designs. MetaHDL selectively inherits SystemVerilog syntax, eliminates unnecessary variants, extends existing synthesizable language structures and adds new grammars to simplify RTL coding. Designers will find it quite intuitive and flexible when using MetaHDL. A compiler named `mhdlc` is implemented to translate MetaHDL to SystemVerilog.

1.1 Features

1. Comprehensive Preprocessor
2. Flexible declarations
3. Port inference and automatic variable declarations
4. Enhanced instantiation syntax
5. New syntax for Flip-Flop (FF) and *Finite State Mahchine* (FSM)
6. Parameter tracing
7. Automatic dependency resolving
8. Lightweight lint checking
9. Independent Verilog Parser to support IP integration
10. Rich user control syntax
11. Re-indent the generated SystemVerilog

1.2 Document Organization

In the reset of this manual, MetaHDL syntax and usage will be documented in detail, following is the organization of this document:

- chapter 2 gives many basic and important concepts about language and RTL design, all readers are expected to read this chapter carefully, otherwise, later chapters are difficult to understand.
- chapter 3 gives major syntax explanations and sample codes. After reading this chapter, readers can develop complex chips with powerful capabilities provided by MetaHDL.
- chapter 4 describes preprocessor in `mhdlc` and its directives. Designers can achieve script like code configurations by using this build-in preprocessor, instead of writing dozens of one-time scripts.
- chapter 5 lists various user control syntax that alters compiler execution.
- chapter 6 describes mechanism and operation flow of `mhdlc`.
- chapter 7 documents all command line options accepted by `mhdlc`.
- chapter 8 provides additional information for those who originally use VPerl for daily coding. Differences with VPerl are summarized there.
- Appendix A is the complete formal syntax of MetaHDL.

1.3 Download

`mhdlc` is not available publically right now (hopefully next few monthes), but anyone can goto MetaHDL Code Request to register and get invitation. Registered users will receive complete source code package and be notified on any update and new releases.

This document is built from LaTeX sources which is distributed with `mhdlc` source code. You can refer to build time to check the version of this document. Latest version in PDF format can be checked out from SVN.

1.4 Bug Report

If you find any bug of `mhdlc`, ambiguous contents or typo in this document, please contact author via mengxin@vlsi.zju.edu.cn, thanks!

Part I

Language Reference

Chapter 2

Basic Concepts

Before discussing syntax details of MetaHDL, several basic concepts should be introduced, including terminologies about *Context Free Grammar* (CFG) and philosophy of MetaHDL. These preliminary information will help understand later chapters. Readers who already familiar with CFG and VPerl can skip this chapter.

2.1 Grammar

MetaHDL is defined using CFG, which means one or more *syntactic groupings* are defined and constructed from their parts via rules. For example, one rule for making an expression might be, "An expression can be made of a minus sign and another expression". Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

In the formal grammatical rules for a language, each kind of syntactic unit or grouping is named by a *symbol*. Those which are built by grouping smaller constructs according to grammatical rules are called *nonterminal* symbols; those which can't be subdivided are called *terminal symbols* or *token types*. We call a piece of input corresponding to a single terminal symbol a *token*, and a piece corresponding to a single nonterminal symbol a *grouping*.

We can use the Verilog as an example of what symbols, terminal and nonterminal, mean. The tokens of Verilog are identifiers, constants (numeric and string), and the various keywords, arithmetic operators and punctuation marks. So the terminal symbols of a grammar for Verilog include 'identifier', 'number', 'string', plus one symbol for each keyword, operator

or punctuation mark: 'if', 'return', 'const', 'static', 'int', 'char', 'plus-sign', 'open-brace', 'close-brace', 'comma' and many more. (These tokens can be subdivided into characters, but that is a matter of lexicography, not grammar.) Here is a simple Verilog combinational block subdivided into tokens:

Listing 2.1: Verilog example for describing CFG

```
1 always_comb begin     // keyword 'always_comb', keyword 'begin'
2   if ( output_gated ) // keyword 'if', open-paren, identifier, close-paren
3     ctrl = 1'b0;      // identifier, equal-sign, based_number
4   else                // keyword 'else'
5     ctrl = i1 & i2;   // identifier, equal-sign, identifier, logic-AND, identifier
6 end                   // keyword 'end'
```

The syntactic groupings of Verilog include the expression, the statement, the declaration, and etc. These are represented in the grammar of Verilog by nonterminal symbols 'expression', 'statement', 'declaration'. The full grammar uses dozens of additional language constructs, each with its own nonterminal symbol, in order to express the meanings of them. The example above is a combinational logic; it contains one 'if-else' statement. In the statement, 'i1' and 'i2' are expressions, and so is 'i1 & i2'.

Each nonterminal symbol must have grammatical rules showing how it is made out of simpler constructs. For example, one kind of Verilog statement is the `assign` statement; this would be described with a grammar rule which reads informally as follows:

> An 'assign statement' can be made of an 'identifier', an 'equal-sign', an 'expressions' and a 'semicolon'.

2.2 MetaHDL Philosophy

RTL designs are not like other programming, there are few local variables. In additional to physical resources occupation semantics, RTL variables also represent nets or connections. Physical elements are connected via variables. Normally, there is no floating net inside modules, which means every net has sources and sink. If a net has no source, it should be module input port. If it has no sink, it should be output port. If it has both sources and sink, it is most probably an internal net. These are basic rules of port inference in `mhdlc`. Designers can override these rules by adding explicit port declarations.

Module ports is automatically inferred by compiler, but designers can always command compiler to perform port validation (or port checking) against designers' explicit declaration. In this situation, golden ports are

declared by designers, compiler compares the ports inferred and ports declared by designers, any missing or newly emerging ports are reported via warning.

In MetaHDL world, there are *only* four types of building block in synthesizable RTL designs: combinational logic, sequential logic (mostly Flip-Flop), module instantiation and FSM[1], they are called *code block* in the rest of this document. Any modules – no matter how complex it is – can be decomposed to these four structures. Module is only a physical resources wrapper with parameters to be overridden upon instantiation.

MetaHDL RTL design is a process in which designers describe functionalities using code blocks; `mhdlc` connects nets with same name and infers ports according to designers' declarations. Parameters are recognized and ports/nets are parameterized automatically.

MetaHDL also allows designers to embed script like flexible code configuration settings in RTL description in a reuse oriented design. Module logic can be fine-grained tuned before translating MetaHDL to SystemVerilog. Ports and variable declarations are *dynamically* updated according to logic configuration.

[1] FSM is essentially a mixture of combinational and sequential logic, Because it is so commonly used, we promote it to basic structure.

Chapter 3

Syntax

MetaHDL syntax born from SystemVerilog. It selectively inherits synthesizable syntax of SystemVerilog, eliminates unnecessary variants, extends module instantiation syntax, add new syntax for Flip-Flop and FSM. Verilog or SystemVerilog designers will find it quite intuitive to use MetaHDL syntax. In the rest of this chapter, major syntax are presented with examples, refer to Appendix A for complete syntax.

3.1 Expression and Statement

Just as Verilog, SystemVerilog or C/C++, *expression* is the bottom level language elements, it constructs *statement*, statement then constructs code blocks which ultimately constructs module. Expression is the same as that in synthesizable Verilog or SystemVerilog. Expression is recursively defined in BNF, prime expressions are net and constant, which leads out of the recursion.

Statement is built upon expression through assignment, conditional control or repetition. Empty statement is also allowed in MetaHDL, since it does not make any sense, warning is popped when empty statement is encountered. 'if-else', 'case', 'for' statements are supported, 'goto' statement is newly added for state transition (refer to section 3.4).

Since MetaHDL inherits SystemVerilog syntax, rules for expression and statements are same as that in SystemVerilog, refer to Appendix A for complete rule list.

3.2 Combinational Logic

Tow code blocks can be used to describe combinational logic:

- `always_comb` code block
- `assign` statement

Different from Verilog or Verilog 2000, SystemVerilog introduces `always_comb` keywords and eliminates sensitivity list, which simplifies combinational logic coding a lot. MetaHDL *only* supports `always_comb` style procedure assignment, legacy Verilog or Verilog 2000 with sensitivity list are not recognized. MetaHDL also supports `assign` continuous assignment to describe simple combinational logic. Listing 3.1 demonstrates legal and illegal combinational logic code in MetaHDL.

Listing 3.1: Combinational logic examples

```
// OK, accepted
always_comb
  if ( enabled )
    o1 = i1 | i2 | i3;
  else
    o1 = 1'b0;

// OK, accepted
assign o2 = cond ? i1 : i2;

// Illegal, wrong!!
// conventional Verilog
always @( i1 or i2 or i3 )
  if ( enabled )
    o1 = i1 | i2 | i3;
  else
    o1 = 1'b0;

// Illegal, wrong!!
// Verilog 2000 is not accepted, neither
always @(*)
  if ( enabled )
    o1 = i1 | i2 | i3;
  else
    o1 = 1'b0;
```

3.3 Flip-Flop

MetaHDL supports two Flip-Flop descriptions: one is conventional `always` block using keywords 'posedge' and 'negedge' to denote Flip-Flop description, the other is newly added syntax using 'ff' and 'endff' keywords. Listing 3.2 demonstrates new Flip-Flop syntax, Listing 3.3 is the equivalence conventional syntax, both of them are legal in `mhdlc`.

Listing 3.2: MetaHDL FF syntax

```
ff clk, rst_n;
  a_ff, i1 ? c : d, 1'b0;
  b_ff, b & c;
endff

// data path does not need reset
ff clk;
  data_ff[63:0], data[63:0];
endff
```

Listing 3.3: Legacy FF syntax

```
// OK, accepted
always_ff @ (posedge clk or negedge rst_n )
  if ( ~rst_n ) begin
    a_ff <= 1'b0;
  end
  else begin
    a_ff <= i1 ? c: d;
    b_ff <= b & c;
  end

// OK, accepted
// data path does not need reset
always_ff @ (posedge clk)
  data_ff[63:0] <= data[63:0];
```

In MetaHDL new FF syntax, keyword 'ff' is followed by two optional identifiers: one is clock name, the other is reset name. If reset name is omitted, there is no reset clause in `always_ff` block. If both of them are omitted, default name 'clock' and 'reset_n' are used, both reset clause and value refresh clause are generated.

'ff_item'[1] consists of three parts: FF name, source value expression, and reset value. Reset value is optional, when it is omitted, corresponding reset behavior is not generated in SystemVerilog.

3.4 FSM

FSM in conventional RTL design requires many constant/parameter definitions to make code readable. But these definitions are hard to maintain during develop iteration, especially for one-hot encoded FSM. MetaHDL introduces *symbol based* FSM programming paradigm that liberates designers from such frustrated situation.

fsm_block[2] is enclosed by keywords 'fsm' and 'endfsm'. Note that 'fsm' line *must* end with semi-colon, just as SystemVerilog 'sequence' or 'property' blocks. 'fsm' is followed by three identifiers: FSM name, clock name, and reset name. FSM name is used as based name of state register, '_cs' and '_ns' suffix are appended to FSM name to create current state register and next state next state register, respectively. clock and reset names are used in sequential block of FSM, which resets state register and perform current state refreshing. clock and reset names can be omitted together, and default name 'clock' and 'reset_n' will be used. State transition explicitly stated by 'goto' keyword, instead of next state assignment.

[1] Maybe you need Appendix A if you don't know what I'm saying.

[2] If you still don't know what is fsm_block, I guess you need to print out Appendix A and look up non-terminals in it when spotted.

Symbol based FSM programming allows designers to code FSM using state names, one-hot state encodings are automatically generated by `mhdlc`. Constant definitions are generated according to state names to improve code readability. To help designers eliminate state name typo, `mhdlc` will build a directed graph representing state transition during parsing, to check the connectivity of every state. Dead states and unreachable states are reported to designers for confirmation. Listing 3.4 is MetaHDL FSM description, Listing 3.5 is the corresponding SystemVerilog description, including constant definition.

Listing 3.4: FSM in MetaHDL

```
fsm cmdrx, clk, rst_n;

  cm_pim_ack = 1'b0;

  IDLE: begin
    if ( pim_cm_req ) begin
      cm_pim_ack = 1'b1;
      goto DATA;
    end
    else begin
      goto IDLE;
    end
  end

  DATA: begin
    cm_pim_ack = 1'b1;
    if ( pim_cm_eof ) begin
      cm_pim_ack = 1'b0;
      goto IDLE;
    end
    else begin
      goto DATA;
    end
  end

endfsm
```

Listing 3.5: FSM in SystemVerilog

```
// other declarations...
const logic [1:0] DATA = 2'b10;
const logic [1:0] IDLE = 2'b01;
const int _DATA_ = 1;
const int _IDLE_ = 0;

// Sequential part of
// FSM /tmp/xin_meng/mhdlc/test/a.mhdl:1.0-25.5
// /tmp/xin_meng/mhdlc/test/a.mhdl:1.0-25.5
always_ff @(posedge clk or negedge rst_n)
  if (~rst_n) begin
    cmdrx_cs <= IDLE;
  end
  else begin
    cmdrx_cs <= cmdrx_ns;
  end

// Combnational part of
// FSM /tmp/xin_meng/mhdlc/test/a.mhdl:1.0-25.5
// /tmp/xin_meng/mhdlc/test/a.mhdl:1.0-25.5
always_comb begin
  cm_pim_ack = 1'b0;
  unique case ( 1'b1 )
    cmdrx_cs[_IDLE_] : begin
      if ( pim_cm_req ) begin
        cm_pim_ack = 1'b1;
        cmdrx_ns = DATA;
      end
      else begin
        cmdrx_ns = IDLE;
      end
    end

    cmdrx_cs[_DATA_] : begin
      cm_pim_ack = 1'b1;
      if ( pim_cm_eof ) begin
        cm_pim_ack = 1'b0;
        cmdrx_ns = IDLE;
      end
      else begin
        cmdrx_ns = DATA;
      end
    end

    default: begin
      cmdrx_ns = 2'hX;
    end
  endcase
end
```

As shown in Listing 3.5, 'fsm_block' is expanded to two blocks: sequential and combinational. The former resets state register, the latter calculates

next state and controls output. Combinational part of FSM is implemented in 'unique case' statement, a bunch of constants are defined to hold state value and hot bit index.

3.5 Module Instantiation

Verilog module instantiation syntax is extended in MetaHDL, BNF is:

inst_block ::= **ID** parameter_rule instance_name connection_spec **;**

Where 'ID' is the module name to be instantiated, parameter_rule, instance_name and connection_spec are all optional. If no instance name specified, prefix 'x_' is added to module name to create instance name.

parameter_rule specifies parameter override. In addition to Verilog positioned override, *named parameter override* is added. Designers can explicitly specify which parameter should be set, rather than list all magic numbers.

In addition to Verilog connection syntax, connection_spec supports prefix, suffix and regular expression connection rules, which save a lot efforts in IP integration and top level integration. Note that prefix, suffix and regular expression rules are cumulative and applicable to all module ports, rule execution sequence is the sequence they appear.

Listing 3.7 is the module to be instantiated. Listing 3.6 instantiates `moda` several times, pay attention to the instance at line 10, prefix and suffix rules take *cumulative* effects on `x2_moda`. Listing 3.8 is the generated SystemVerilog.

Listing 3.6: Wrapper Module in MetaHDL

```
// simplest instantiation
moda;


// prefix rule
moda x1_moda ( x1_ +);

// suffix rule
// after prefix rule
moda x2_moda ( x2_ + ,
               + _22);


// Perl compatible regexp
moda x3_moda ( "s/o/out/g",
               "s/i/in/g" );
```

Listing 3.7: Module Template

```
module moda (
  i1,
  i2,
  i3,
  i4,
  i5,
  i6,
  o1,
  o2);


input i1;
input i2;
input i3;
input i4;
input i5;
input i6;
output o1;
output [1:0] o2;

endmodule
```

Listing 3.8: Wrapper module in SystemVerilog

```
// declarations...

// /tmp/xin_meng/mhdlc/test/modb.mhdl:2.0-4
moda x_moda (
               .o1 (o1),
               .i1 (i1),
               .i2 (i2),
               .o2 (o2),
               .i3 (i3),
               .i4 (i4),
               .i5 (i5),
               .i6 (i6)
          );

// /tmp/xin_meng/mhdlc/test/modb.mhdl:6.0-21
moda x1_moda (
                .o1 (x1_o1),
                .i1 (x1_i1),
                .i2 (x1_i2),
                .o2 (x1_o2),
                .i3 (x1_i3),
                .i4 (x1_i4),
                .i5 (x1_i5),
                .i6 (x1_i6)
           );

// /tmp/xin_meng/mhdlc/test/modb.mhdl:10.0-11.21
moda x2_moda (
                .o1 (x2_o1_22),
                .i1 (x2_i1_22),
                .i2 (x2_i2_22),
                .o2 (x2_o2_22),
                .i3 (x2_i3_22),
                .i4 (x2_i4_22),
                .i5 (x2_i5_22),
                .i6 (x2_i6_22)
           );

// /tmp/xin_meng/mhdlc/test/modb.mhdl:15.0-16.27
moda x3_moda (
                .o1 (out1),
                .i1 (in1),
                .i2 (in2),
                .o2 (out2),
                .i3 (in3),
                .i4 (in4),
                .i5 (in5),
                .i6 (in6)
           );
```

3.6 Parameterization

MetaHDL enables designers to creates parameterized module in two ways:

- Write parameterized module from draft.
- Build parameterized module from existing parameterized modules.

Designers declare parameters, and use them in ports or net index. `mhdlc` will automatically parameterize ports in generated declarations. If a module

to be instantiated is a parameterized module, `mhdlc` can trace parameter usage in port connections and automatically parameterize wrapper module.

Listing 3.9: `modc` in MetaHDL

```
1  parameter A = 4;
2  parameter B = 5;
3  parameter C = A + B;
4
5  assign o1[C-1:0] = {~i1[A-1:0], i2[B-1:0]};
```

Listing 3.10: `modc` in SystemVerilog

```
1  module modc (
2    i1,
3    i2,
4    o1);
5
6  parameter A = 4;
7  parameter B = 5;
8  parameter C = 4 + 5;
9
10 input [A - 1:0] i1;
11 input [B - 1:0] i2;
12 output [C - 1:0] o1;
13
14 logic [A - 1:0] i1;
15 logic [B - 1:0] i2;
16 logic [C - 1:0] o1;
17
18 // /tmp/xin_meng/mhdlc/test/modc.mhdl:5.0-42
19 assign o1[C - 1:0] = {~i1[A - 1:0], i2[B - 1:0]};
20
21 endmodule
```

Listing 3.9 is a parameterized module to be instantiated. Three parameters are defined in it, in which `C` depends on other two.

Listing 3.10 is the generated SystemVerilog code, all ports are parameterized which fits designers' intend pretty well.

Listing 3.11 is a wrapper module named 'modd' that contains three instantiations of modc with different parameter settings. 'modd' itself is parameterized by two parameters, this example demonstrates the automatic parameterization through instantiation.

First instance only overrides value of `A` via named override. Second instance uses positioned override to set values of `A` and `B`, parameters in wrapper module are used. Third instance only overrides value of `A` via named override, parameters in wrapper module are used.

Listing 3.11: Instantiation

```
parameter SETA = 8,
          SETB = 9;


modc #( A = 2 ) x0_modc ( x0_ + );

modc #( SETA, SETB ) x1_modc ( x1_ + );

modc #( A = SETA ) x2_modc (x2_ +,
                            .o1 (x2_o1[10:0]));
```

Listing 3.12: SV Instantiation

```
module modd (
// port list...
);

parameter SETA = 8;
parameter SETB = 9;

input [1:0] x0_i1;
input [4:0] x0_i2;
output [6:0] x0_o1;
input [SETA - 1:0] x1_i1;
input [SETB - 1:0] x1_i2;
output [SETA + SETB - 1:0] x1_o1;
input [SETA - 1:0] x2_i1;
input [4:0] x2_i2;
output [10:0] x2_o1;

// variable declarations...

// /tmp/xin_meng/mhdlc/test/modd.mhdl:5.0-33
modc #(
      2,      // A
      5,      // B
      2 + 5   // C
     ) x0_modc (
                .o1 (x0_o1),
                .i1 (x0_i1),
                .i2 (x0_i2)
               );

// /tmp/xin_meng/mhdlc/test/modd.mhdl:7.0-38
modc #(
      SETA,   // A
      SETB,   // B
      SETA + SETB    // C
     ) x1_modc (
                .o1 (x1_o1),
                .i1 (x1_i1),
                .i2 (x1_i2)
               );

// /tmp/xin_meng/mhdlc/test/modd.mhdl:9.0-10.18
modc #(
      SETA,   // A
      5,      // B
      SETA + 5 // C
     ) x2_modc (
                .o1 (x2_o1[10:0]),
                .i1 (x2_i1),
                .i2 (x2_i2)
               );

endmodule
```

Listing 3.12 is the generated SystemVerilog from Listing 3.11. First instance is configured by constant, new nets created by prefix rule are not parameterized.
In second instance, `A` and `B` are override by parameters in wrapper module, `C` is untouched. To preserve the parameter dependency in `modc`, compiler will record parameter usage and propagated the dependency to the wrapper module. So port `x1_o1` is parameterized to `x1_o1[SETA+SETB-1]`, which captures designers' intent perfectly.
Third instance is a mixture of parameter override and constant override. Besides, port `o1` is explicitly connected to `x2_o1[10:0]`. In this case, compiler will not parameterize `x2_o1`.

3.7 Escape from MetaHDL

MetaHDL provide keywords `rawcode` and `endrawcode` to escape code from `mhdlc`. Designers can write non-MetaHDL syntax in side this block, all contents are copied to generated SystemVerilog literally.

Chapter 4

Preprocessor

Preprocessor helps designers to embed script like code configuration directives into RTL code for reuse oriented designs to improve code integrity. Conventionally, designers are used to write one-time scripts (Perl/sed/awk/csh) to preprocess their RTL for similar project usage. This methodology is not clean enough. Verification engineers have to create additional steps in `Makefile` to preprocess code. MetaHDL's preprocessor uses Verilog style macro syntax, introduces more flow control directives that help designers perform conditional and repetitive configuration on RTL.

In addition to conventional `` `ifdef ``, `` `ifndef ``, `` `else ``, `` `define `` and `` `include `` macro directives, MetaHDL introduces `` `for ``, `` `if `` and `` `let `` to enlarge the power of preprocessor (see following examples).

Listing 4.1 is a simple Round Robin Arbiter FSM implemented in MetaHDL with facilitating preprocessor. This arbiter can respond to a configurable number of slaves, which is controlled by macro `SLV_NUM`. Once the MetaHDL code is finished, various arbiters can be generated in SystemVerilog with giving different values to `SLV_NUM` when invoke `mhdlc`.

Listing 4.1: Configurable Arbiter

```
1  fsm arb;
2
3  `for (i=1; `i<=`SLV_NUM; i++)
4  slave_grnt_`i = 1'b0;
5  `endfor
6
7  `for (i=1; `i<=`SLV_NUM; i++)
8   `let j = `i + 1
9
10  `if `i != `SLV_NUM
11 SLAVE_`i: begin
12    if ( slave_req_`i ) begin
13       slave_grnt_`i = 1'b1;
14       if ( slave_eof_`i ) begin
15          slave_grnt_`i = 1'b0;
16          goto SLAVE_`j;
17       end
18       else begin
19          goto SLAVE_`i;
20       end
21    end
22    else
23      goto SLAVE_`j;
24 end
25
26  `else
27 SLAVE_`i: begin
28    if ( slave_req_`i ) begin
29       slave_grnt_`i = 1'b1;
30       if ( slave_eof_`i ) begin
31          slave_grnt_`i = 1'b0;
32          goto SLAVE_1;
33       end
34       else
35         goto SLAVE_`i;
36    end
37    else
38      goto SLAVE_1;
39 end
40  `endif
41 `endfor
42 endfsm
```

Line 3 starts a `for directive to repetitively "write" code with slight difference. Default values of FSM output are set within this block.

Line 7 starts another `for directive to write slave handling code, one state for each slave. Since it is a Round Robin arbiter, every states perform same task: grant slave access if has request, move to next slave when current one has no request or transaction is done, roll back to the first slave when a arbitration round finishes. Line 10, 26 and 40 compose an `if block to check whether current state is for last slave.

Line 8 is `let directive used to perform arithmetic operation and calculate value of `j, which is the number of next slave.

Listing 4.2 is the generated SystemVerilog with SLV_NUM set to 4.

Listing 4.2: 4 slave arbiter SystemVerilog code

```
1  module arbiter (
2    clock,
3    reset_n,
4    slave_eof_1,
5    slave_eof_2,
6    slave_eof_3,
7    slave_eof_4,
8    slave_grnt_1,
9    slave_grnt_2,
10   slave_grnt_3,
11   slave_grnt_4,
12   slave_req_1,
13   slave_req_2,
14   slave_req_3,
15   slave_req_4);
16
17
18 input clock;
19 input reset_n;
20 input slave_eof_1;
21 input slave_eof_2;
22 input slave_eof_3;
23 input slave_eof_4;
24 output slave_grnt_1;
25 output slave_grnt_2;
26 output slave_grnt_3;
```

```
27  output slave_grnt_4;
28  input slave_req_1;
29  input slave_req_2;
30  input slave_req_3;
31  input slave_req_4;
32
33  const logic [3:0] SLAVE_1 = 4'b0001;
34  const logic [3:0] SLAVE_2 = 4'b0010;
35  const logic [3:0] SLAVE_3 = 4'b0100;
36  const logic [3:0] SLAVE_4 = 4'b1000;
37  const int _SLAVE_1_ = 0;
38  const int _SLAVE_2_ = 1;
39  const int _SLAVE_3_ = 2;
40  const int _SLAVE_4_ = 3;
41  logic [3:0] arb_cs;
42  logic [3:0] arb_ns;
43  logic clock;
44  logic reset_n;
45  logic slave_eof_1;
46  logic slave_eof_2;
47  logic slave_eof_3;
48  logic slave_eof_4;
49  logic slave_grnt_1;
50  logic slave_grnt_2;
51  logic slave_grnt_3;
52  logic slave_grnt_4;
53  logic slave_req_1;
54  logic slave_req_2;
55  logic slave_req_3;
56  logic slave_req_4;
57
58  // Sequential part of FSM /tmp/xin_meng/mhdlc/test/arbiter.mhdl:1.0-42.5
59  // /tmp/xin_meng/mhdlc/test/arbiter.mhdl:1.0-42.5
60  always_ff @(posedge clock or negedge reset_n)
61    if (~reset_n) begin
62      arb_cs <= SLAVE_1;
63    end
64    else begin
65      arb_cs <= arb_ns;
66    end
67
68  // Combnational part of FSM /tmp/xin_meng/mhdlc/test/arbiter.mhdl:1.0-42.5
69  // /tmp/xin_meng/mhdlc/test/arbiter.mhdl:1.0-42.5
70  always_comb begin
71    slave_grnt_1 = 1'b0;
72    slave_grnt_2 = 1'b0;
73    slave_grnt_3 = 1'b0;
74    slave_grnt_4 = 1'b0;
75    unique case ( 1'b1 )
76      arb_cs[_SLAVE_1_] : begin
77        if ( slave_req_1 ) begin
78          slave_grnt_1 = 1'b1;
79          if ( slave_eof_1 ) begin
80            slave_grnt_1 = 1'b0;
81            arb_ns = SLAVE_2;
82          end
83          else begin
84            arb_ns = SLAVE_1;
85          end
86        end
87        else begin
88          arb_ns = SLAVE_2;
89        end
90      end
91
92      arb_cs[_SLAVE_2_] : begin
93        if ( slave_req_2 ) begin
94          slave_grnt_2 = 1'b1;
95          if ( slave_eof_2 ) begin
96            slave_grnt_2 = 1'b0;
97            arb_ns = SLAVE_3;
98          end
99          else begin
100           arb_ns = SLAVE_2;
101         end
102       end
```

```
103       else begin
104         arb_ns = SLAVE_3;
105       end
106     end
107
108     arb_cs[_SLAVE_3_] : begin
109       if ( slave_req_3 ) begin
110         slave_grnt_3 = 1'b1;
111         if ( slave_eof_3 ) begin
112           slave_grnt_3 = 1'b0;
113           arb_ns = SLAVE_4;
114         end
115         else begin
116           arb_ns = SLAVE_3;
117         end
118       end
119       else begin
120         arb_ns = SLAVE_4;
121       end
122     end
123
124     arb_cs[_SLAVE_4_] : begin
125       if ( slave_req_4 ) begin
126         slave_grnt_4 = 1'b1;
127         if ( slave_eof_4 ) begin
128           slave_grnt_4 = 1'b0;
129           arb_ns = SLAVE_1;
130         end
131         else begin
132           arb_ns = SLAVE_4;
133         end
134       end
135       else begin
136         arb_ns = SLAVE_1;
137       end
138     end
139
140     default: begin
141       arb_ns = 4'hX;
142     end
143   endcase
144 end
145
146
147 endmodule
```

`let directive supports sufficient operators and a bunch of common used functions:

- Numeric operators: addition (+), subtraction (−), multiplication (∗), division (/), modulus (%), power (∗∗).
- Logical operators: logical AND (&&), logical OR (||), logical NOT (!).
- Bit operators: bitwise XOR (ˆ), bitwise AND (&), bitwise OR (|), shift right (>>), shift left (<<).
- Functions: log 2 (LOG2()), round up (CEIL()), round down (FLOOR()), round to nearest value (ROUND()), max of two numbers (MAX()), min of two numbers (MIN()), odd (ODD()), even (EVEN()), absolute value (ABS()).

Listing 4.3: `let usage examples

```
1  `define x 2
2
3  // NOTE `let need "="
4  `let y = `x ** 10 // now `y is 1024
5
6  `let z = LOG2(`y) // `z is 10
7  `let a = LOG2(`z) // `a is 3.321928
8  `let c = CEIL(`a) // `c is 4
9  `let f = FLOOR(`a) // `f is 3
10 `let r = ROUND(`a) // `r is 3
11
12
13 // we can concatenate macro value with other strings
14 // using "::" operator
15 assign cat_`f::k = 1'b0; // expand to ``assign cat_3k = 1'b0;''
16 assign cat_`fk = 1'b0; // Error!! macro ``fk'' is not defined
```

Listing 4.3 demonstrates some usages of `let directive. Operators and functions supported by `let directive constructs macro variables to *macro expression*, such as `a + `b, ODD(`f), `a >> 2, `f <= `d && `f != 0, which can further be evaluated by `if directive for flow controlling, just as shown in Listing 4.1 line 10 .

Chapter 5

User Control

MetaHDL provides control syntax start with keyword "`metahdl`", which interfaces with `mhdlc` and controls the runtime behavior of compiler. Designers' controls are passed to compiler via variable assignments embedded in RTL code, this variable settings are also preceded by keyword `metahdl`. Boolean variables inside compiler are set via +/− preceded by variable name, where + means "enable" and − means "disable". There are two special form of control syntax: exit syntax, and echo syntax. The former is used to command compiler exit when the statement is encountered. The latter is used to print messages on `stderr`. They are usually used with preprocessor to guarantee correct configuration settings. Refer to Appendix A for detailed formal syntax.

Working scope of all variables can be *Modular* or *Effective*. Modular variables (MVAR) take effect on entire module and are used when parsing is finished. Designers can set MVAR anywhere in source code and get the same effect. If an MVAR is assigned multiple times, last assignment wins. MVAR can have different values in different files, so file is the minimum granularity of MVAR.

Effective variables (EVAR) take effect from the point the variable is assigned and are used *during* parsing. Designers can set different values for same EVAR in different sections of source code, and make compiler treat sections differently. So the minimum granularity of EVAR is section divided by EVAR assignments.

5.1 Variable List

Following is the complete list of all compiler variables can be assigned by user control syntax, variable type (boolean or string) and variable scope (MVAR or EVAR) are listed with variable name.

modname MVAR string
Default Value: Base file name
Set the generated module name. Often used with preprocessor to distinguish modules with different configurations.

outfile MVAR string
Default Value: Base file name
Set the generated SystemVerilog file base name. Often used with preprocessor to distinguish module definition files with different configurations.

portchk MVAR boolean
Default Value: false
Enable/Disable port validation for module.

hierachydepth MVAR positive int
Default Value: 300
Maximum level of module instantiation.

clock EVAR string
Default Value: clock
Default clock name used for `ff_block` and `fsm_block`.

reset EVAR string
Default Value: reset_n
Default reset name used for `ff_block` and `fsm_block`.

multidriverchk MVAR boolean
Default Value: true
Enable/Disable multiple driver checking for module.

relexedfsm EVAR boolean
Default Value: true
Set severity of connectivity/reachability error checked in FSM. If it is true, relaxed FSM programming mode is enabled, all dead states or unreachable states are acceptable, compiler only reports warning when such states are encountered, and continues processing. if it is false, FSM programming is in strict mode, any dead state or unreachable state is considered to be fatal error, compiler will report error and stop processing if such state is checked.

exitonwarning EVAR boolean
Default Value: false
Set severity of *normal parsing warning*, such as width mismatch. If it is true, compiler exits on any warning.

exitonlintwarning MVAR boolean
Default Value: false
Warning from port validation, multiple driver checking are categorized as lint warning. If this variable is set to true, compiler will exit on any lint warning.

5.2 Example

Listing 5.1 demonstrates user control syntax with code configuration.

Listing 5.1: Demo Top Wrapper

```
1  metahdl + portchk;
2
3  `if WIDTH > 64
4  metahdl ``width can not exceed 64!'';
5  metahdl exit;
6  `endif
7
8  assign data[`WIDTH-1:0] = `WIDTH'd0;
9
10 `ifdef FPGA
11 `define target fpga
12 `else
13 `define target asic
14 `endif
15
16 metahdl modname = top_`target;
17 metahdl outfile = top_`target;
18
19 metahdl + exitonwarning;
20
21 metahdl clock = clk_125M;
22 metahdl reset = pclk_rst_n;
23
24 ff;
25   a_ff, a, 1'b0;
26   b_ff[1:0], b, 1'b0;
27 endff
28
29 metahdl - exitonwarning;
30
31 metahdl clock = clk_250M;
32 metahdl reset = dclk_rst_n;
33
34 ff;
35   c_ff, c, 1'b0;
36   d_ff, d, 1'b0;
37 endff
38
39 ff;
40   e_ff, e, 1'b0;
41   g_ff, g, 1'b0;
42 endff
```

Line 1 enables port validation in this module.

Line 3 checks value of macro `WIDTH`, forces compilation exit upon illegal values.

Line 16 and 17 alter SystemVerilog module name and output file name according to target device.

Any warning between 19 and 29 makes compiler exit. To be more specific, width mismatch between `b_ff` and `b` is considered to be fatal error.

Different clock and reset names are used for different code sections, section 23-28, section 33-42.

Part II

Compiler Usage

Chapter 6

Compilation Flow

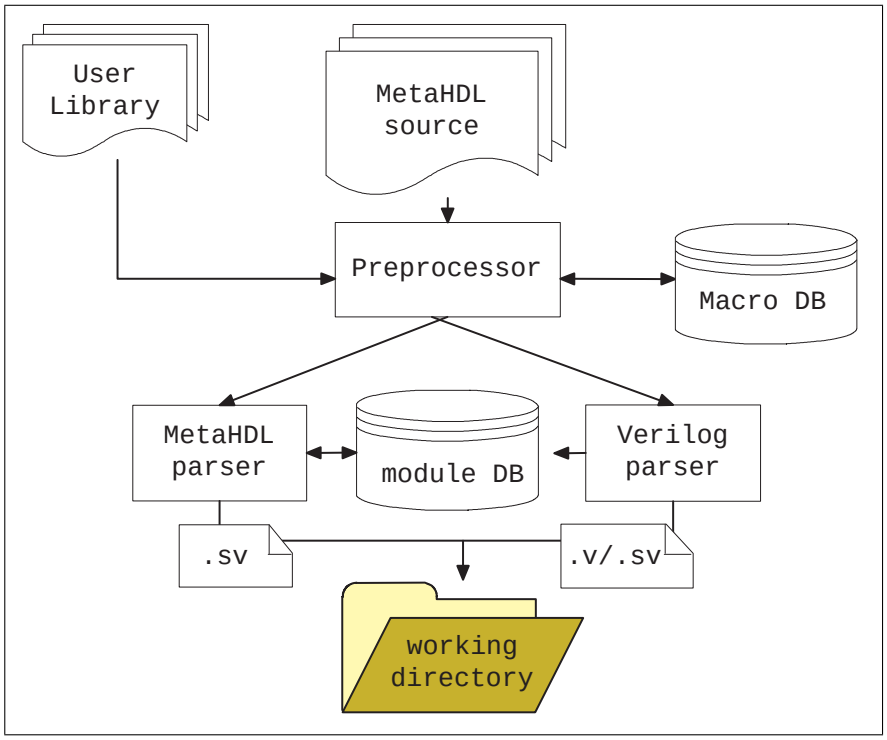

Figure 6.1: MetaHDL compilation flowchart

Figure 6.1 is MetaHDL compilation flow. `mhdlc` has a build-in directory based automatic dependency resolving capability, which means files to be

processed can be supplied in any order[1], or only provide top level wrapper module, compiler can find all instantiated modules in search path which is specified by user through command line.

Once the module is compiled, module definition is stored in module database maintained by compiler. When module instantiation statement is encountered, compiler first searches module definition in module database, if not found, compiler start automatic dependency resolving to find module definition.

Automatic dependency resolving is *comprehensive* because compiler will search MetaHDL, Verilog or SystemVerilog format module definitions in search path. Multiple definition is considered to be fatal error, and designers are responsible to fix the problem. Comprehensive dependency resolving requires traversing of search path three times for each module, this could cost very long runtime especially when search path list is very large. *Fast dependency resolving* mode can be enabled to save time. In this mode, first found module definition is used, designers are in their own risk of multiple definition. This mode can achieve 2x-4x faster than comprehensive mode.

`mhdlc` puts all generated files in centralized output directory (default name is "workdir") which can be specified by users. RTL are generated in SystemVerilog format with ".sv" as file extension. Files with ".postpp" extension are output files from preprocessor, only used for compiler debugging. Normally, only SystemVerilog files generated from MetaHDL are put in output directory, Verilog or SystemVerilog parsed by compiler are *not* copied to output directory. However, users can still ask compiler to put all touched Verilog or SystemVerilog files into output directory with command line options.

[1] More strictly, macro definition files should be provided before macro usage files, otherwise, because compiler can not expand macro without definition.

Chapter 7

Command Line Options

`mhdlc` command line captions are case sensitive, which means `-p` and `-P` are different.

Table 7.1: `mhdlc` command line options

Option	Description
`-I`	Specify single search path.
`-P`	Specify a list of search paths in a file, one path per line. Line starts with "#" is comment line and is ignored by compiler.
`-D`	Define macro just as used in GCC or VCS.
`-C`	Enable Verilog or SystemVerilog copying touched by compiler.
`-F`	Enable fast dependency resolving mode.
`-o`	Specify output directory.
`-f`	Specify a list of files to be processed, one file per line. Line starts with "#" is comment line and is ignored by compiler.
`--version`	Display version number and copyright.
`-h`	Display help information.

All other text in command line and does not start with "-" are considered to be file names to be processed.

Chapter 8

For VPerl Designers

1. `&Depend` is no longer needed since `mhdlc` automatically resolves dependency.
2. `vpmake -depend` is not needed anymore, just give `mhdlc` top level file and search path.
3. All `&Force` should be converted to standard Verilog declarations, including 2D array.
4. `&ConnRule` and `&Connect` should be converted according to MetaHDL port connect syntax.
5. `&Instance` should be converted to MetaHDL instantiation syntax.
6. `c-sky vperl_off` and `c-sky vperl_on` should be converted to `rawcode` and `endrawcode`.
7. File extension is "`.mhdl`", not "`.vp`".

Part III

Appendix

Appendix A

Formal Syntax

The formal syntax of MetaHDL is described using Backus-Naur Form (BNF). The conventions used are:

- Keywords are in lower case red text.
- Punctuation are in red text.
- A vertical bar "|" separates alternatives.
- UPPER case red text are tokens from lexer.

---

start ::= /* empty */
| start port_declaration
| start parameter_declaration
| start constant_declaration
| start variable_declaration
| start assign_block
| start combinational_block
| start legacyff_block
| start ff_block
| start fsm_block
| start inst_block
| start rawcode_block
| start metahdl_constrol

constant ::= **NUM**
| **BIN_BASED_NUM**

| **DEC_BASED_NUM**
| **HEX_BASED_NUM**

net_name ::= **ID**

net ::= net_name [expression **:** expression]
| net_name [expression]
| net_name [expression] [expression]
| net_name [expression] [expression **:** expression]
| net_name

net_lval ::= net
| **{** net_lvals **}**

net_lvals ::= net
| net_lvals **,** net

expression ::= constant
| net
| concatenation
| net_name **(** expressions **)**
| **{** expression concatenation **}**
| **(** expression **)**
| | expression
| **&** expression
| **^** expression
| **~** expression
| expression | expression
| expression **&** expression
| expression **^** expression
| expression + expression
| expression − expression
| expression ***** expression
| expression / expression
| expression % expression
| expression << expression
| expression >> expression
| expression **?** expression **:** expression
| **!** expression
| expression || expression
| expression **&&** expression
| expression < expression
| expression > expression
| expression == expression
| expression **!=** expression
| expression >= expression

```
    | expression <= expression

concatenation ::=  { expressions }

expressions ::=  expression
    | expressions , expression

statement ::=  balanced_stmt
    | unbalanced_stmt

balanced_stmt ::=  ;
    | for ( net_lval = expression ; expression ; net_lval = expression ) statement
    | begin end
    | net_lval <= expression ;
    | net_lval = expression ;
    | begin statements end
    | begin : ID statements end
    | if ( expression ) balanced_stmt else balanced_stmt
    | case_statement
    | goto ID ;

unbalanced_stmt ::=  if ( expression ) statement
    | if ( expression ) balanced_stmt else unbalanced_stmt

statements ::=  statement
    | statements statement

case_statement ::=  case_type ( expression ) case_items endcase
    | case_type ( expression ) case_items default : statement endcase

case_type ::=  case
    | casez
    | unique case
    | unique casez
    | priority case
    | priority casez

case_items ::=  case_item
    | case_items case_item

case_item ::=  expressions : statement

port_declaration ::=  port_direction net_names ;
    | port_direction [ expression : expression ] net_names ;

net_names ::=  net_name
    | net_names , net_name
```

port_direction ::= **input**
| **output**
| **inout**
| **nonport**

parameter_declaration ::= **parameter** parameter_assignments **;**

parameter_assignments ::= parameter_assignment
| parameter_assignments **,** parameter_assignment

parameter_assignment ::= **ID** **=** expression

constant_declaration ::= **const** variable_type net_name **=** expression **;**
| **const** variable_type **[** expression **:** expression **]** net_name **=** expression **;**

variable_declaration ::= variable_type net_names **;**
| variable_type **[** expression **:** expression **]** net_names **;**
| variable_type net_names **[** expression **:** expression **]** **;**
| variable_type **[** expression **:** expression **]** net_names **[** expression **:** expression **]** **;**

variable_type ::= **reg**
| **wire**
| **logic**
| **int**
| **integer**

assign_block ::= **assign** net_lval **=** expression **;**

always_keyword ::= **always**
| **always_ff**

legacyff_block ::= always_keyword **@** **(** **posedge** net_name **or** **negedge** net_name **)** statement

legacyff_block ::= always_keyword **@** **(** **posedge** net_name **)** statement

combinational_block ::= **always_comb** statement

ff_block ::= **ff** **ID** **;** ff_items **endff**
| **ff** **ID** **,** **ID** **;** ff_items **endff**
| **ff** **;** ff_items **endff**

ff_items ::= ff_item
| ff_items ff_item

ff_item ::= net_lval **,** expression **,** expression **;**
| net_lval **,** expression **;**

fsm_block ::= **fsm ID , ID , ID ;** statements fsm_items **endfsm**

fsm_block ::= **fsm ID ;** statements fsm_items **endfsm**

fsm_items ::= fsm_item
| fsm_items fsm_item

fsm_item ::= **ID :** statement

inst_block ::= **ID** parameter_rule instance_name connection_spec **;**

instance_name ::= /* empty */
| **ID**

parameter_rule ::= /* empty */
| **#** **(** parameter_override **)**

parameter_override ::= parameter_num_override
| parameter_name_override

parameter_num_override ::= expression
| parameter_num_override **,** expression

parameter_name_override ::= **ID =** expression
| parameter_name_override **, ID =** expression

connection_spec ::= /* empty */
| **(** connection_rules **)**

connection_rules ::= connection_rule
| connection_rules **,** connection_rule

connection_rule ::= **.** net_name **(** expression **)**
| **.** net_name **( )**
| **STRING**
| **+ ID**
| **ID +**

rawcode_block ::= **rawcode** verbtims **endrawcode**
| **function** verbtims **endfunction**

verbtims ::= **VERBTIM**
| verbtims **VERBTIM**

metahdl_constrol ::= **metahdl ID ;**
| **metahdl + ID ;**
| **metahdl − ID ;**

| **metahdl ID = NUM ;**
| **metahdl ID = ID ;**
| **metahdl STRING ;**

Appendix B

Change Log

B.1 Revision 0.1

1. Move `mhdlc` location and application notes into chapter 1.
2. Divide chapter 1 into sections.
3. Add document link in sharepoint.

Index