MetaHDL Manual

xinmeng@hotmail.com

2023-11-27

Contents

1 Introduction

MetaHDL (shorted as "mhdl" in this document) is an HDL aims at synthesizable digital VLSI designs (commonly known as RTL designs). mhdl 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 mhdl. A compiler named `mhdlc` is implemented to translate mhdl to SystemVerilog or Verilog

1.1 Features

1. Comprehensive Preprocessor
2. Flexible declarations
3. Port inference and automatic variable declarations
4. Enhanced instantiation syntax
5. New syntax for ff and 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 sv/v

1.2 Document Organization

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

- Basic Concepts gives many basic and important concepts behind mhdl. All readers are expected to read this chapter carefully, otherwise, later chapters are difficult to understand.
- Syntax gives major syntax explanations and sample codes. After reading this chapter, readers can develop complex chips with powerful capabilities provided by mhdl.
- Preprocessor describes preprocessor in mhdlc and support directives. Designers can achieve script-like code configurations by using this build-in preprocessor, instead of writing dozens of one-time scripts.

- User Control lists various user control variables that alter compiler execution.
- Command Line Options documents all command line options accepted by mhdlc.
- For VPerl Designers provides additional information for those who originally use vperl for daily coding. Differences with vperl are summarized there.
- formal syntax is the complete formal syntax of mhdl.

1.3 download and bug report

`mhdlc` is publically available at https://github.com/xinmeng/metahdl. if you find any bug of mhdlc, ambiguous contents or typo in this document, please file issue at github.

2 BASIC CONCEPTS

RTL designs are not like other programming, there are few local variables. In addition 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 should have source and sinks. if a net has no source, it should be module input port, which will be fed by external drivers. If it has no sink, it should be module output port, which will drive external modules' signal. If it has both source and sinks, it is most probably an internal net. These are basic rules of port inference in mhdl. Designers can override these rules by adding explicit port declarations.

Module ports are automatically inferred by compiler, and designers can ask compiler to perform port validation (or port checking) against designers' explicit declaration. In this scenario, golden ports are declared by designers, compiler compares the inferred ports and declared ports, any missing or newly emerging ports are reported as error.

In mhdl world, there are **only** four types of building blocks in synthesizable rtl designs:

1. combinational logic
2. sequential logic (mostly flip-flop)
3. module instantiation
4. FSM. Technically, FSM is essentially a mixture of combinational and sequential logic, because it is so commonly used, we promote it to a basic structure.

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 treated as a physical resources wrapper with parameters to be overridden upon instantiation.

mhdl RTL designing 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.

mhdl also allows designers to embed script-like code configurations (via preprocessor) in RTL in a reuse oriented design. Module logic can be fine-grain tuned before translating mhdl to SystemVerilog. Ports and variable declarations are dynamically updated according to logic configuration.

3
SYNTAX

mhdl 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 mhdl syntax. In the rest of this chapter, major syntax are presented with examples, refer to Formal Syntax for complete syntax.

3.1 Combinational Logic

There are two and **only two** types of code block in mhdl for coding combinational logic, as shown in Listing 3.1:

1. `assign` statement
2. `always_comb` statement

```
// 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 is NOT supported
always @( i1 or i2 or i3 )
  if ( enabled )
    o1 = i1 | i2 | i3;
  else
    o1 = 1'b0;


// Illegal, wrong!!
// Verilog 2000 is NOT supported, either
always @(*)
  if ( enabled )
    o1 = i1 | i2 | i3;
  else
    o1 = 1'b0;
```

Listing 3.1: Combinational logic example

3.2 Sequential Logic

There are two types of code block in mhdl for coding sequential logic:

1. `always_ff @()` statement, which is same in SystemVerilog
2. `ff-endff` block, which is introduced by mhdl

The troditional SystemVerilog syntax is good except its redundancy: FF variable appears twice (even more) in different clause of `if-else` branches. For a multi-bit vector variable, such redundancy is prone to typo and width mismatch.

For well-coded FF, combinational part of the FF sources should be coded in a separate code block, so the `if-else` branches can be reduced. mhdl provides a new `ff-endff` code block to reduce redundancy. FF code in following two forms (Listing 3.2) are equivalent.

```
// troditional sequential block
always_ff @ (posedge clk or negedge rst_n)
  if (!rst_n)
    a_ff <= 1'b0;
  else
    a_ff <= a;

// MetaHDL new sequential block
ff;
  a_ff, a, 1'b0;
endff
```

Listing 3.2: ff-endff code block example

`ff-endff` block can optionally specify clock and reset signal name. Usually they are ommitted to further reduce redundancy. Each line in block describes a FF. A line has three element:

1. FF variable name, `a_ff` here.
2. An expression containing the logic to update the FF, `a` here. Any expression defined in Formal Syntax are allowed here.
3. An optinally reset value. If no reset value is provided, FF variable will not be reset.

Here is example:

```
ff clk_a, clk_a_rst_n;
  a_ff, a, 1'b0;
  b_ff, b;
  c_ff, a_ff & b_ff, 1'b0;
endff

// Generated SystemVerilog from above code
always_ff @(posedge clk_a or negedge clk_a_rst_n)
  if (!clk_a_rst_n) begin
     a_ff <= 1'b0;
     c_ff <= 1'b0;
  end
  else begin
     a_ff <= a;
     b_ff <= b;
     c_ff <= a_ff & b_ff;
  end
```

Listing 3.3: ff-endff block example and corresponding SystemVerilog

3.3 FSM

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

FSM code block is enclosed by keywords `fsm`, `fsm_nc` and `endfsm`. `fsm` keyword is followed by three identifiers:

1. FSM name, which is mandatory.
2. clock signal name, which is optional.
3. reset signal name, which is optional, too.

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 signal 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 `clk` and `rst_n` will be used. State transition is explicitly stated by `goto` keyword, instead of next state assignment.

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. By checking the connectivity of every state, dead states and unreachable states are reported to designers for confirmation. **??** is mhdl FSM description, **??** is the corresponding SystemVerilog description, including constant definition.

```
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 3.4: FSM code in MetaHDL

```
// 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 cmdrx
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 cmdrx
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
```

Listing 3.5: FSM code generated in SystemVerilog

Difference between `fsm` and `fsm_nc` is Verilog generated from `fsm_nc` block will not contain the sequential block. That means designers have to manually code the sequential block. This is expecially designed for FSM with synchronous reset. **Note** that the manual crafted sequential block *must* come after the block, because and signals are only accessible after block is parsed.

3.4 Module Instantiation

SystemVerilog module instantiation syntax is extended in mhdl, BNF is shown in Formal Syntax, start from non-terminal "inst_block". Features of mhdl instantiation syntax are highlighted below:

1. Instance name is optional. Default instance name created by prefixeding `x_` on module named.
2. Port connection is optional. Default behavior is to connect ports to net with identical name.
3. Prefix and/or Suffix connection rules are allowed in port connection (see example below).
4. Regular expression connection rule is allowed in port connection (see example below).

```
module moda
  (input i1,
   input i2,
   output o1,
   output [1:0] o2
   );

endmodule
```

Listing 3.6: Module moda to be instantiated

Listing 3.7 shows different connection rules. Listing 3.8 is the generated SystemVerilog.

```
// simplest instantiation
moda;

// prefix connection rule
moda x1_moda ( x1_ +);

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

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

Listing 3.7: Instantiate moda in mhdl

```
moda x_moda
  (.i1 (i1),
   .i2 (i2),
   .o1 (o1),
   .o2 (o2) );

moda x1_moda
  ( .i1 (x1_i1),
    .i2 (x1_i2),
    .o1 (x1_o1),
    .o2 (x1_o2) );

moda x2_moda
  ( .i1 (x2_i1_22),
    .i2 (x2_i2_22),
    .o1 (x2_o1_22),
    .o2 (x2_o2_22) );

moda x3_moda
  ( .i1 (in1),
    .i2 (in2),
    .o1 (out1),
    .o2 (out2) );
```

Listing 3.8: Instaniate moda

3.5 Parameter Tracing

mhdl enables designers to creates parameterized module in two ways:

1. Write parameterized module in MetaHDl from draft.

2. Build parameterized module by instantiating parameterized modules.

To use parameter in mhdl source code, designers declare parameters, and use them in ports or net. `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 nets in wrapper module. Here are some examples.

In Listing 3.9, `modc` is created in mhdl with parameters. Listing 3.10 is the generated SystemVerilog. Ports and nets are properly parameterized.

```
parameter A = 4;
parameter B = 5;
parameter C = A + B;

assign o1[C-1:0] = {~i1[A-1:0], i2[B-1:0]};
```

Listing 3.9: Coding `modc` in mhdl with parameters

```
module modc
  ( i1, i2, o1);

   parameter A = 4;
   parameter B = 5;
   parameter C = 4 + 5;

   input [A - 1:0] i1;
   input [B - 1:0] i2;
   output [C - 1:0] o1;

   logic [A - 1:0]  i1;
   logic [B - 1:0]  i2;
   logic [C - 1:0]  o1;

   assign o1[C - 1:0] = {~i1[A - 1:0], i2[B - 1:0]};

endmodule
```

Listing 3.10: Generated modc.v with parameters

In Listing 3.11, `modc` is instantiated several times to demonstrate parameter tracing.

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

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

modc #(.A (SETA), .B (SETB)) x1_modc ( x1_ + );

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

Listing 3.11: Instantiate `modc` in mhdl with parameter override

```
module modwrapper
  ( x0_i1, x0_i2, x0_o1, x1_i1, x1_i2, x1_o1, x2_i1, x2_i2, x2_o1);

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;

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

    modc #( .A( 2 ),
            .B( 5 ),
            .C( 2 + 5 ) )
    x0_modc
      ( .i1 (x0_i1),
        .i2 (x0_i2),
        .o1 (x0_o1) );

    modc #( .A( SETA ),
            .B( SETB ),
            .C( SETA + SETB ) )
    x1_modc
      ( .i1 (x1_i1),
        .i2 (x1_i2),
        .o1 (x1_o1) );

    modc #( .A( SETA ),
            .B( 5 ),
            .C( SETA + 5 ) )
    x2_modc
      ( .i1 (x2_i1),
        .i2 (x2_i2),
        .o1 (x2_o1[10:0]) );

endmodule
```

Listing 3.12: Genrated wrapper with parameter tracing

3.6 Optional Declaration

Declaration in Verilog and SystemVerilog is mandatory, but in mhdl is optional. `mhdlc` can automatcially infer width, port directions, and variable type from a well designed synthesizable RTL code. But in some cases, designers want to constrain the inference results. This can be done by declaration statements. Usually, declaration is used in follow sceanrios:

1. Force port direction, such as

   ```
   input a;
   output b;
   nonport c;
   ```

2. 2 dimensional array, such as

   ```
   reg [31:0] unpacked_a [15:0];
   reg [31:0] [7:0] packed_b;

   always_comb begin
      // i will be declared by mhdlc
   ```

```
   for (i=0; i<8; i++)
     packed_b[i] = unpacked_a[i];
end
```

4 PREPROCESSOR

Preprocessor helps designers to embed script-like code configuration directives into RTL code for reuse oriented designs. Conventionally, designers are used to write one-time scripts (Perl/sed/awk/csh) to preprocess their RTL for using in different project. This methodology is not clean enough. Verification engineers have to create additional steps in Makefile to preprocess code. mhdl preprocessor uses SystemVerilog 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, 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 with facilitating preprocessor. This arbiter can respond to a configurable number of slaves, which is controlled by macro `SLV_NUM`. Once the code is finished, various arbiters can be generated by giving different values to `SLV_NUM`.

```
fsm arb;

`for (i=1; `i<=`SLV NUM; i++)
slave_grnt_`i = 1'b0;
`endfor

`for (i=1; `i<=`SLV NUM; i++)
 `let j = `i + 1

 `if `i != `SLV NUM
SLAVE_`i: begin
   if ( slave_req_`i ) begin
      slave_grnt_`i = 1'b1;
      if ( slave_eof_`i ) begin
         slave_grnt_`i = 1'b0;
         goto SLAVE_`j;
      end
      else begin
         goto SLAVE_`i;
      end
   end
   else
     goto SLAVE_`j;
end

 `else
SLAVE_`i: begin
   if ( slave_req_`i ) begin
      slave_grnt_`i = 1'b1;
      if ( slave_eof_`i ) begin
         slave_grnt_`i = 1'b0;
         goto SLAVE_1;
      end
      else
        goto SLAVE_`i;
   end
   else
     goto SLAVE_1;
end
 `endif
`endfor
endfsm
```

Listing 4.1: Configurable Arbiter in mhdl

A `‘for` directive is used to repetitively "write" code with slight difference. Default values of FSM output are set within this block.

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. A `‘if` block is used to check whether current state is for last slave.

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

Listing 4.2 shows te SyntemVerilog generated with `SLV_NUM=4`.

```
module arbiter (
                clock,
                reset_n,
                slave_eof_1,
                slave_eof_2,
                slave_eof_3,
                slave_eof_4,
                slave_grnt_1,
                slave_grnt_2,
                slave_grnt_3,
                slave_grnt_4,
                slave_req_1,
                slave_req_2,
                slave_req_3,
                slave_req_4);


   input clock;
   input reset_n;
   input slave_eof_1;
   input slave_eof_2;
   input slave_eof_3;
   input slave_eof_4;
   output slave_grnt_1;
   output slave_grnt_2;
   output slave_grnt_3;
   output slave_grnt_4;
   input  slave_req_1;
   input  slave_req_2;
   input  slave_req_3;
   input  slave_req_4;

   const logic [3:0] SLAVE_1 = 4'b0001;
   const logic [3:0] SLAVE_2 = 4'b0010;
   const logic [3:0] SLAVE_3 = 4'b0100;
   const logic [3:0] SLAVE_4 = 4'b1000;
   const int          _SLAVE_1_ = 0;
   const int          _SLAVE_2_ = 1;
   const int          _SLAVE_3_ = 2;
   const int          _SLAVE_4_ = 3;
   logic [3:0]        arb_cs;
   logic [3:0]       arb_ns;
   logic             clock;
   logic             reset_n;
   logic             slave_eof_1;
   logic             slave_eof_2;
   logic             slave_eof_3;
   logic             slave_eof_4;
   logic             slave_grnt_1;
   logic             slave_grnt_2;
   logic             slave_grnt_3;
   logic             slave_grnt_4;
   logic             slave_req_1;
   logic             slave_req_2;
   logic             slave_req_3;
   logic             slave_req_4;

   // Sequential part of FSM /tmp/xin_meng/mhdlc/test/arbiter.mhdl:1.0-42.5
   // /tmp/xin_meng/mhdlc/test/arbiter.mhdl:1.0-42.5
   always_ff @(posedge clock or negedge reset_n)
     if (~reset_n) begin
        arb_cs <= SLAVE_1;
     end
     else begin
        arb_cs <= arb_ns;
     end

   // Combnational part of FSM /tmp/xin_meng/mhdlc/test/arbiter.mhdl:1.0-42.5
   // /tmp/xin_meng/mhdlc/test/arbiter.mhdl:1.0-42.5
   always_comb begin
      slave_grnt_1 = 1'b0;
      slave_grnt_2 = 1'b0;
      slave_grnt_3 = 1'b0;
      slave_grnt_4 = 1'b0;
      unique case ( 1'b1 )
        arb_cs[_SLAVE_1_] : begin
           if ( slave_req_1 ) begin
              slave_grnt_1 = 1'b1;
              if ( slave_eof_1 ) begin
                 slave_grnt_1 = 1'b0;
                 arb_ns = SLAVE_2;
              end
              else begin
                 arb_ns = SLAVE_1;
              end
           end
           else begin
              arb_ns = SLAVE_2;
           end
        end

        arb_cs[_SLAVE_2_] : begin
           if ( slave_req_2 ) begin
              slave_grnt_2 = 1'b1;
              if ( slave_eof_2 ) begin
                 slave_grnt_2 = 1'b0;
                 arb_ns = SLAVE_3;
              end
              else begin
                 arb_ns = SLAVE_2;
              end
           end
           else begin
              arb_ns = SLAVE_3;
           end
        end

```

```
        arb_cs[_SLAVE_3_] : begin
           if ( slave_req_3 ) begin
              slave_grnt_3 = 1'b1;
              if ( slave_eof_3 ) begin
                 slave_grnt_3 = 1'b0;
                 arb_ns = SLAVE_4;
              end
              else begin
                 arb_ns = SLAVE_3;
              end
           end
           else begin
              arb_ns = SLAVE_4;
           end
        end

        arb_cs[_SLAVE_4_] : begin
           if ( slave_req_4 ) begin
              slave_grnt_4 = 1'b1;
              if ( slave_eof_4 ) begin
                 slave_grnt_4 = 1'b0;
                 arb_ns = SLAVE_1;
              end
              else begin
                 arb_ns = SLAVE_4;
              end
           end
           else begin
              arb_ns = SLAVE_1;
           end
        end

        default: begin
           arb_ns = 4'hX;
        end
      endcase
   end


endmodule
```

Listing 4.2: Generated Arbiter in SystemVerilog code (SLV_NUM=4)

'let directive supports following functions:

- Numeric operators: addition (+), subtraction (-), multiplication (*), division (/), modulus (%), power (**).
- Logical operators: logical AND (&&), logical OR (||), logical NOT (!).
- Bit operators: bitwise XOR (^), 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()).

```
`define x 2

// NOTE `let need "="
`let y = `x ** 10 // now `y is 1024

`let z = LOG2(`y) // `z is 10
`let a = LOG2(`z) // `a is 3.321928
`let c = CEIL(`a) // `c is 4
`let f = FLOOR(`a) // `f is 3
`let r = ROUND(`a) // `r is 3
```

```


// we can concatenate macro value with other strings
// using "::" operator
assign cat_`f::k = 1'b0; // expand to ``assign cat_3k = 1'b0;''
assign cat_`fk = 1'b0; // Error!! macro ``fk'' is not defined
```

Listing 4.3: `‘let` usage examples

5 User Control

mhdl provides control syntax start with keyword `metahdl`, which interfaces with 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 + or - preceded by variable name, where + means "enable" and - means "disable".

There are two special form of control syntax: `exit` syntax, and `message` 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.

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.

Table 5.1: Compiler Control variables

Name	Scope	Type	Default	Description
modname	MVAR	string	Base file name	Set the generated module name. Often used with preprocessor to distinguish modules with different configurations.
outfile	MVAR	string	Base file name	Set the generated Synthesizable file base name. Often used with preprocessor to distinguish module definition files with different configurations.
portchk	MVAR	boolean	false	Enable/Disable port validation for module.
hierachydepth	MVAR	postitive int	300	Maximum level of module instantiation.
clock	EVAR	string	clock	Default clock name used for `ff-endff` and `fsm-endfsm`
reset	EVAR	string	reset_n	Default reset name used for `ff-endff` and `fsm-endfsm`
relexedfsm	EVAR	boolean	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	false	Set severity of *normal parsing warning*, such as width mismatch. If it is true, compiler exits on any warning.
exitonlintwarning	EVAR	boolean	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

demonstrates user control syntax with code configuration.

- 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 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.

```
 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
```

6 Compiler Usage

6.1 Command Line Options

`mhdlc` command line is listed below:

```
% mhdlc <options> mhdl_file [mhdl_file mhdl_file ...]
```

Command line captions are case sensitive, which means `-p` and `-P` are different. Major options are listed below:

1. `-I` Specify single search path for `mhdlc` to look for `‘include` files.
2. `-o` Specify output base directory for generated files. mhdl source directory structure is mirrored under it.
3. `-D` Define macro from command line. e.g., `-DCMD_EN` defines a macro named `CMD_EN`, `-DSLV_NUM=4` defines a macro `SLV_NUM` with `4` as its value.
4. `-mb` specify mhdl source base directory. All mhdl files are searched recursively from this directory. `-mb` can only be specified once, multiple values are not allowed.
5. `-ib` specify IP base directories, which contain existing verilog designs. Multiple `-ib` can be spcified at command line.

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

6.2 For VPerl Designers

1. `&Depend` is no longer needed since automatically resolves dependency.
2. `vpmake -depend` is not needed anymore, just give top level file and search path.
3. All `&Force` should be converted to standard declarations, including 2D array.
4. `&ConnRule` and `&Connect` should be converted according to port connect syntax.
5. `&Instance` should be converted to 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` .

7 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 force_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

| start generate_block

constant ::= **STRING**

| **NUM**

| **BIN_BASED_NUM**

| **DEC_BASED_NUM**

```
    | HEX_BASED_NUM
    | FLOAT

net_name ::=  ID

net ::=  net_name [ expression : expression ]
    | net_name [ expression ]
    | net_name [ expression ] [ expression ]
    | net_name [ expression ] [ expression : expression ]
    | net_name [ expression + : expression ]
    | net_name [ expression — : expression ]
    | net_name

net_lval ::=  net
    | { net_lvals }

net_lvals ::=  net
    | net_lvals , net

expression ::=  constant
    | net
    | concatenation
    | $clog2 ( expressions )
    | net_name ( expressions )
    | { expression concatenation }
    | ( expression )
    | | expression
    | & expression
    | ˆ expression
    | ˜ expression
    | expression | expression
    | expression & expression
    | expression ˆ 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 **<=** optional_delay expression **;**

| net_lval **=** expression **;**

| **begin** statements **end**

| **begin : ID** statements **end**

| **if (** expression **)** balanced_stmt **else** balanced_stmt

| case_statement

| **goto ID ;**

optional_delay ::= **empty**

| **# NUM**

| **# FLOAT**

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

force_port_declaration ::= **force** port_declaration

port_declaration ::= port_direction net_names **;**

| port_direction **[** expression **:** expression **]** net_names **;**

| port_direction **[** expression **:** expression **] [** expression **:** expression **]** net_names **;**

net_names ::= net_name

| net_names **,** net_name

port_direction ::= **input**

| **output**

| **inout**

| **nonport**

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

| **localparam** 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 **] [** expression **:** expression **]** net_names **;**
  | variable_type **[** expression **:** expression **]** net_names **[** expression **:** expression **] ;**

variable_type ::= **wire**
  | **reg**
  | **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
  | 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_keyword ::= **fsm**
  | **fsm_nc**

fsm_header ::= fsm_keyword **ID ;**
  | fsm_keyword **ID , ID , ID ;**

fsm_block ::= fsm_header 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 +**

generate_block ::= **generate** generate_statements **endgenerate**

generate_statements ::= generate_statement

| generate_statements generate_statement

generate_statement ::= generate_balanced_statement

| generate_unbalanced_statement

generate_balanced_statement ::= assign_block

| combinational_block

| ff_block

| legacyff_block

| inst_block

| **begin : ID** generate_statements **end**

| **for (** net_lval **=** expression **;** expression **;** net_lval **=** expression **)** generate_statement

| **if (** expression **)** generate_balanced_statement **else** generate_balanced_statement

| **case (** expression **)** generate_case_items **endcase**

generate_unbalanced_statement ::= **if (** expression **)** generate_statement

| **if (** expression **)** generate_balanced_statement **else** generate_unbalanced_statement

generate_case_items ::= generate_case_item

| generate_case_items generate_case_item

generate_case_item ::= expressions **:** generate_statement

| **default :** generate_statement

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 message** verbtims **;**

| **metahdl parse** verbtims **;**