Verishort HDL

Anish Bramhandkar  Elba Garza  Scott Rogowski  Ruijie Song

December 22, 2010
Contents

1 An Introduction to Verishort 4
   1.1 Background ................................................. 4
   1.2 Related Work ............................................... 4
   1.3 Goals of Verishort ......................................... 4
      1.3.1 Short ................................................ 4
      1.3.2 Logical .............................................. 4
      1.3.3 Clean ............................................... 4

2 Understanding, Compiling, and Running a Verishort file 5

3 Reference Manual 5
   3.1 Lexical Conventions ........................................ 5
      3.1.1 Tokens ............................................... 5
      3.1.2 Comments ............................................ 5
      3.1.3 Data Types .......................................... 5
      3.1.4 Identifiers .......................................... 6
      3.1.5 Keywords ............................................ 6
      3.1.6 Numbers .............................................. 6
      3.1.7 Operators ........................................... 7
      3.1.8 Buses ............................................... 8
   3.2 Assignment ................................................. 8
      3.2.1 Assigning Wires ...................................... 8
      3.2.2 Binding Ports ....................................... 9
   3.3 Expressions & Operators ...................................... 9
      3.3.1 Concatenation, Replication & Splitting ............... 9
      3.3.2 Bitwise .............................................. 9
      3.3.3 Parenthesis .......................................... 10
      3.3.4 Reduction .......................................... 10
      3.3.5 Arithmetic .......................................... 10
      3.3.6 Bit Shifts .......................................... 11
      3.3.7 Sign Extension ...................................... 11
      3.3.8 Comparison .......................................... 11
      3.3.9 Precedence and associativity ......................... 12
   3.4 Declarations ................................................ 12
      3.4.1 Identifiers .......................................... 12
      3.4.2 Wire Declarations .................................... 12
      3.4.3 Register Declarations ................................. 13
      3.4.4 Module Declarations .................................. 13
      3.4.5 Building A Module .................................... 13
      3.4.6 Statements .......................................... 14
      3.4.7 Conditional Statements ................................. 15
      3.4.8 Case/Switch Structures ............................... 15
      3.4.9 For loops ........................................... 16
   3.5 Scope ...................................................... 16
   3.6 Preprocessor ................................................ 16
   3.7 Standard Library Modules ................................... 16
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7.1 Latches</td>
<td>17</td>
</tr>
<tr>
<td>3.7.2 Flipflops</td>
<td>17</td>
</tr>
<tr>
<td>4 Project Plan</td>
<td></td>
</tr>
<tr>
<td>4.1 Team Responsibilities</td>
<td>17</td>
</tr>
<tr>
<td>4.1.1 Round-Robin Dictatorship</td>
<td>17</td>
</tr>
<tr>
<td>4.2 Project Timeline</td>
<td>17</td>
</tr>
<tr>
<td>4.3 Software Development Environment</td>
<td>17</td>
</tr>
<tr>
<td>4.4 Project Log</td>
<td>18</td>
</tr>
<tr>
<td>5 Architectural Design</td>
<td></td>
</tr>
<tr>
<td>5.1 Architecture</td>
<td>18</td>
</tr>
<tr>
<td>5.2 Error Recovery</td>
<td>19</td>
</tr>
<tr>
<td>6 Testing Plan</td>
<td>19</td>
</tr>
<tr>
<td>7 Lessons Learned</td>
<td>19</td>
</tr>
<tr>
<td>8 Codebase</td>
<td></td>
</tr>
<tr>
<td>8.1 Ast.ml</td>
<td>21</td>
</tr>
<tr>
<td>8.2 Asttoimst.ml</td>
<td>22</td>
</tr>
<tr>
<td>8.3 Inst.ml</td>
<td>33</td>
</tr>
<tr>
<td>8.4 Imsttocode.ml</td>
<td>34</td>
</tr>
<tr>
<td>8.5 Parser.mly</td>
<td>36</td>
</tr>
<tr>
<td>8.6 Scanner.mll</td>
<td>41</td>
</tr>
<tr>
<td>9 Example files</td>
<td></td>
</tr>
<tr>
<td>9.1 Gcd.vs</td>
<td>42</td>
</tr>
<tr>
<td>9.2 Gcd.v</td>
<td>42</td>
</tr>
<tr>
<td>9.3 Stim file for gcd</td>
<td>43</td>
</tr>
<tr>
<td>9.4 Helloworld.vs</td>
<td>45</td>
</tr>
<tr>
<td>9.5 Helloworld.v</td>
<td>46</td>
</tr>
</tbody>
</table>
1 An Introduction to Verishort

1.1 Background

Verilog is a very popular hardware description language (HDL) which is widely utilized by the electronics hardware design industry. First invented and used in the early 80s at Automated Integrated design Systems, Verilog was put into the public domain and standardized by the IEEE in 1995. This initial public version of Verilog became known as Verilog-95. The language was later expanded in 2001 and 2005 to address deficiencies and add features resulting in Verilog-2001 and Verilog-2005, the most recent version. (A combined hardware description/verification language known as SystemVerilog was extended from the 2005 standard but goes beyond the scope of this manual.)

Despite its popularity, Verilog is infamous for its repetitiveness, strange grammar, and ease of bug insertion. Part of this is a factor of the nature of low-level hardware design. There is a difference between languages meant to be run using gates and latches rather than processors and memory. However, we believe that another part of this simply poor language design and can be improved.

VeriShort HDL is meant to simplify the Verilog-2005 language to make it easier to read and write. First, we have reduced repetitiveness in accordance with the DRY (Don’t repeat yourself) philosophy by simplifying module input/output syntax and instantiation. Next, we introduced some C-language features such as brackets and array-like bus descriptions. We substantially simplified synchronous logic by doing away with always syntax and replacing it with simple if statements. The list of reserved keywords has been substantially shortened in order to make VeriShort completely synthesizable and to remove rarely used features. Finally, we added a standard library of commonly used electronic components like latches, multiplexers, and decoders to further reduce the Verilog tedium.

Because of the wide adoption of Verilog and the existence of many verifiers and hardware synthesizers specific to the IEEE standards, the initial goal of VeriShort will not be to exist as a self contained HDL but rather to translate into clean synthesizable Verilog code. In support of these efforts, a translator has been started and is expected to be running by the date of December 22nd 2010.

1.2 Related Work

1.3 Goals of Verishort

1.3.1 Short

We want Verishort code to be comparibly shorter than the Verilog code it creates. We want to take away the humdrum of writing Verilog code and make it easier to write what we wish to write.

1.3.2 Logical

Verishort shouldn’t skimp on the general power of Verilog either. Therefore, we support the most common operations and help make common structures easy to write.

1.3.3 Clean

We hope to be able to write clean and understandable code that can be, at a glance, intuitively translated to its equal Verilog code. The syntax should be friendly, logical, and quick to learn.
2 Understanding, Compiling, and Running a Verishort file

This section refers to the Verishort input file `gcd.vs` found in the source tarball and included at the end of this document.

This module calculates the greatest common denominator of the eight-bit inputs `num0` and `num1` (obviously with a non-recursive algorithm as this is hardware). As with any Verishort file, the order of lines in a file is parameters, followed by declarations (parameters and registers), followed by statements (if-else blocks, clock blocks, and instantiations among other things).

To compile the Verishort compiler\(^1\), you can use the Makefile as follows:

```
make all
```

This will generate the executable file `vsc`. To compile your Verishort file, run `vsc` with your filename as the first parameter. Alternatively, the compiler will also accept standard input if no parameter is specified. If standard in is used as input, no output file can be specified and the compiler will print to standard out. If the input parameter is used, the second parameter may be the desired output file:

```
./vsc gcd.vs gcd.v
```

This file can now be used with any Verilog stim files. `gcdstim.v` is a working Verilog stimulation file that can be used to verify the output of the Verishort compiler for `gcd.vs`.

3 Reference Manual

3.1 Lexical Conventions

3.1.1 Tokens

There are 5 classes of tokens: identifiers, keywords, numbers, operators, and other separators.

Blanks, tabs, and newlines (collectively, whitespace) are ignored, except when they serve to separate tokens.

3.1.2 Comments

The characters `/*` introduce a comment, which is terminated by the characters `*/`.

The characters `//` also introduce a comment, which is terminated by the newline character. Comments do not nest. Lines marked as comments are discarded by the compiler.

3.1.3 Data Types

The primary data type is the bit, which may store the value 0 or 1. A group of bits comprises a bus. All multibit binary values are treated as two’s complement numbers.

In for-loops (see the corresponding section), the loop variable is assumed to be a simple integer (i.e., natural number).

---

\(^1\)On Unix, we require that you have OCaml and the OCaml libraries installed. On Debian-based systems, the packages are called `ocaml` and `ocaml-libs` respectively.
3.1.4 Identifiers
An identifier is a sequence of characters that represent a wire, bus, register, parameter, or module. An identifier may only include alphanumerical characters or the underscore character (\_). The first character of an identifier may not be a number.

3.1.5 Keywords
The following identifiers are reserved as keywords and may not be used for any other purpose: In this manual, keywords are bolded.

- case
- clock
- concat
- else
- for
- if
- input
- module
- negedge
- output
- parameter
- posedge
- register
- return
- reset
- wire

3.1.6 Numbers
Numbers can be either binary or integer values and are specified as follows:

- A sequence of digits, followed by a radix suffix (‘b’ required for binary but no suffix for decimal values)
- The characters 0 and 1 are valid binary digits.
- The characters 0-9 are valid integer digits.
- Extended binary numbers are like normal binary numbers, but may also use the character x as a binary digit for the value “don’t care” as in Verilog. They may only be used in case structures.
3.1.7 Operators

An operator is a token that specifies an operation on at least one operand. The operand may be an expression or a constant.

Bitwise operators:
- ~
- &
- |
- ~
- ~~
- <<
- >>

Comparison operators:
- ==
- !=
- >=
- <=
- >
- <

Arithmetic operators:
- +
- -
- *
- %

Sign extension operator:
- ,
3.1.8 Buses

A bus represents a multibit wire. The number of bits in a bus must be determinable at compile time. Buses are declared using the syntax `data_type bus_name[number_of_bits];` Where `data_type` is either `wire`, `register` or assumed to be input or output by its position in a module declaration. The number of bits must be a constant. From here, any bit in the bus may be referred to using the subscript syntax: `bus_name[bit_index]`, where `bit_index` is a constant or expression (evaluable at compile time) that yields an integer value less than the size of the bus.

A range of bits in a bus is represented by using the index of the most significant bit in the range, followed by the colon character (:) , followed by the index of the least significant bit in the range, as the subscript. For example, wires 4-8 would be referred to by `bus_name[7:3]`. Reversing this order is invalid.

3.2 Assignment

All assignments in Verishort bind wires and ports to other wires, binary values, decimal values, or registers. Registers can also be assigned a value in `if` and `case` blocks but not outside them. These can be done en masse, such as in buses (multi-bit wires) or one by one.

3.2.1 Assigning Wires

The most basic assignment is of a single wire to a single bit value:

```plaintext
wire w1 = 0;
wire w2 = ~w1; // w2 == 1b
```

A bundle of wires can be assigned to a multi-bit value, as long as the number of bits matches the number of wires in the bundle:

```plaintext
wire w3[5] = 01010b; // assigning a binary value
wire w4[4] = 10; // assigning a decimal value
wire w5[10] = concat(1b,8{0b},1b); // 1000000001b using concatenation
```

The two sides of an assignment must have the same number of bits.

This does not work and will result in an error because the left hand side and the right hand side are not the same size:

```plaintext
wire w6[10] = 10b;
```

Note that the number of bits must always be specified for more than one bit.

Subsets of buses can be assigned:

```plaintext
wire w6[5];
w6[3:0] = 4;
w6[4] = 1b; // w6 == 10100b
```
3.2.2 Binding Ports

Ports are bound to wires when instantiating a module by setting the modules parameter name equal to the wire or value to which it should be bound. Unlike assigning wires, these bindings must be in whole. The value of a port that has not been bound is assumed to be 0.

```verilog
module m1(input in1[5], in2; output out[5]) { ... } // declared somewhere
//now, inside of a calling module
wire w7[5];
m1(in1 = w6, in2 = 1b; out = w7);
```

3.3 Expressions & Operators

The logic of Verishort is described using expressions which are made up of one or more operators and operands. An operand can be either a single bit or a bus. All expressions will return a bit or bus that is then be assigned to a wire or output (see “Assignment”) or returned in an output. This section will detail operators ordered from the most basic building blocks to complex operations.

3.3.1 Concatenation, Replication & Splitting

To place two or more bits or buses together into a single bus, the concatenation syntax is used.

```verilog
wire a = 0;
wire b = 1;
wire c[2] = 01b;
//concat(a, b, c, 01b) results in 010101b
wire a1 = 1;
wire b1[4];
b1 = concat(4 { a1 }); //results in 1111b, which is equivalent to concat(a1, a1, a1, a1)
```

3.3.2 Bitwise

Bitwise operators represent the primitive AND, OR, and NOT gates. All other logical are a combination of these operations. Every bitwise operation with the exception of the NOT gate is a binary operation using infix notation with both operands being the same size which is also the size of the return value. A NOT operation will return the same number of bits as are in its single operand.

Primitive bitwise operators ordered by precedence.

```verilog
wire a = 01b;
wire b = 11b;
```

Bitwise Operations:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>~a</td>
<td>10b</td>
</tr>
<tr>
<td>AND</td>
<td>a&amp;b</td>
<td>01b</td>
</tr>
<tr>
<td>OR</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>
Full range of bitwise operators (NAND and NOR not supported):

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Equivalency</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR</td>
<td>a^b</td>
<td>(~a &amp; b)</td>
<td>(a &amp; ~b)</td>
</tr>
<tr>
<td>XNOR</td>
<td>a \textasciicircum b</td>
<td>(~a</td>
<td>b) &amp; (a</td>
</tr>
</tbody>
</table>

3.3.3 Parenthesis

Parenthesis has the highest precedence.
1|(1&0) //1
(1|1)&0 //0

3.3.4 Reduction

Reduction operators take only a single operand on their right hand side (a bus) and result in a single bit result.

wire a[3] = 010b;

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Equivalency</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND</td>
<td>\textasciicircum &amp;a</td>
<td>\textasciicircum(a[0] &amp; a[1] &amp; a[2])</td>
<td>1</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td>a</td>
<td>a[0]</td>
</tr>
<tr>
<td>NOR</td>
<td>\textasciicircum</td>
<td>a</td>
<td>\textasciicircum(a[0]</td>
</tr>
<tr>
<td>XOR</td>
<td>^a</td>
<td>a[0] ^ a[1] ^ a[2]</td>
<td>1</td>
</tr>
<tr>
<td>XNOR</td>
<td>\textasciicircum ^a</td>
<td>\textasciicircum(a[0] ^ a[1] ^ a[2])</td>
<td>0</td>
</tr>
</tbody>
</table>

3.3.5 Arithmetic

Arithmetic operators are shorthand for common equivalent but complex operations. They operate on two bits or buses which do not have to be the same size. They will return a bit or bus. All operations are done in two’s complement.

In general, the bus that receives the result must contain enough bits to hold all bits in the result, or the result of the arithmetic operation may be undefined.

wire e0[3] = 011b //equivalent to 3d and can be expanded to 0011b
wire e1[3] = 111b //equivalent to -1d and can be expanded to 1111b
wire e3[3];
wire e4[4];
<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Notes</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plus</td>
<td>e3=e0+e1</td>
<td>Overflow bits are discarded.</td>
<td>e3=010b</td>
</tr>
<tr>
<td>Minus</td>
<td>e3=e0−e1</td>
<td></td>
<td>e3 = 100b</td>
</tr>
<tr>
<td>Multiplication</td>
<td>e0*e1</td>
<td>Returns a bus that is n+m-1 long</td>
<td>10101b</td>
</tr>
<tr>
<td>Modulus</td>
<td>e0%e1</td>
<td>Returns a bus that is n long where n is the size of the second operand</td>
<td>011b</td>
</tr>
</tbody>
</table>

### 3.3.6 Bit Shifts

Shifting operations will shift the entire bus to the left or right and will discard the bits shifted off the end.

\[
e0 = 0111b; //7d  
e1 = 1111b; //-1d
\]

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Note</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left-shift</td>
<td>e0&lt;&lt;2; e1&lt;&lt;2</td>
<td>Left shift will always fill with zeros</td>
<td>1100b //−4d; 1100b</td>
</tr>
<tr>
<td>Right-shift</td>
<td>e0&gt;&gt;2; e1&gt;&gt;2</td>
<td>Right shift will always fill with the most significant bit to preserve sign</td>
<td>0001b //1d; 1111b //−1d</td>
</tr>
</tbody>
</table>

### 3.3.7 Sign Extension

The sign extension operator sign-extends the right operand to the number of bits specified by the left hand side operand in decimal. The left operand must be determinable at compile time. Attempts to specify a number of bits that is smaller than the number of bits in the right operand is a syntax error. The operation is done in twos complement.

\[
e0 = 0111b; //7d  
e1 = 1111b; //7d
\]

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Note</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign Extension</td>
<td>8’e0; 8’e1</td>
<td>Left hand side is always a decimal number.</td>
<td>0000011b; 1111111b</td>
</tr>
</tbody>
</table>

### 3.3.8 Comparison

Comparison operators will compare the values of two buses which do not need to be equally sized and will return a one bit true or false result.

\[
e0 = 0111b; //7d  
e1 = 0111b; //7d  
e2 = 1111b; //−1d
\]
<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than</td>
<td>$e_0 &lt; e_1$</td>
<td>0</td>
</tr>
<tr>
<td>Less than or equal to</td>
<td>$e_0 \leq e_1$</td>
<td>1</td>
</tr>
<tr>
<td>Greater than</td>
<td>$e_0 &gt; e_2$</td>
<td>1</td>
</tr>
<tr>
<td>Greater than or equal to</td>
<td>$e \geq e_2$</td>
<td>1</td>
</tr>
<tr>
<td>Equal to</td>
<td>$e_0 == e_1$</td>
<td>1</td>
</tr>
<tr>
<td>Not equal to</td>
<td>$e_0 \neq e_1$</td>
<td>0</td>
</tr>
</tbody>
</table>

### 3.3.9 Precedence and associativity

<table>
<thead>
<tr>
<th>Operators, in order of decreasing precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>~ + - &amp; ^ ~ ^ ^ ^ (unary) ' (sign extension)</td>
<td>right to left</td>
</tr>
<tr>
<td>* % /</td>
<td>left to right</td>
</tr>
<tr>
<td>+ - (binary)</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt;= &gt;&gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp; (binary)</td>
<td>left to right</td>
</tr>
<tr>
<td>^ ~ ^ (binary)</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td>(binary)</td>
</tr>
</tbody>
</table>

### 3.4 Declarations

There are three types of declarations in Verishort: wires, for passing values between modules; registers, for storing values between clock cycles; and modules, blocks of Verishort code offering specific functionality.

#### 3.4.1 Identifiers

Identifiers are user-friendly names, much like variable names in most programming languages. They must start with an upper- or lower-case letter followed by any sequence of letters, numbers, and underscores. No other characters are allowed in identifier names.

#### 3.4.2 Wire Declarations

Wires can be single or multi-bit (buses/bundles). They are declared using the keyword `wire` followed by a space, then an identifier, then an optional number inside square brackets. The number inside the brackets is the number of bits to be used for the bus, using 0-indexing. The MSB is at the highest bit value, i.e., at index 15 for a 16-bit wire. If the brackets and enclosed number are omitted, the wire is assumed to hold a single bit.

```verilog
wire w1; // single bit
wire w2[5]; // five bits with MSB at index 4, LSB at index 0;
```

In addition, the declared wires can take values immediately during the declaration. Following the identifier (or optional bracketed expression) another space, an equals sign, a space, a binary, decimal, or parameter value can be used:
wire w4[3] = 4d;

Once assigned, wires cannot be reassigned.

### 3.4.3 Register Declarations

Registers are declared exactly like wires but with the `register` keyword. The main difference between registers and wires are that registers can be reassigned inside of `if` and `case` blocks but may not be assigned anywhere else. They can be used to store values between clock cycles, for example.

### 3.4.4 Module Declarations

Modules are analogous to classes in Java. Modules allow code to be grouped to offer specific functionality. They are declared with the `module` keyword, followed by an identifier, followed by a parenthesized expression. The parenthesized expression contains a comma-separated list of input ports preceded by the `input` keyword, followed by a semi-colon, followed by a comma-separated list of output ports preceded by the `output` keyword. The body of the module (discussed below) followed inside braces:

```module m1(input in1, in2, in3[3]; output out[5]) { ... }
```

In addition to the user-specified inputs and outputs (see the Assignment section for information on binding ports), each module has an implicit reset input (keyword `reset`) and clock input.

Modules can be declared in any order but all code must reside in one file. Modules cannot be nested, i.e., one module cannot be declared inside of another module. However, a module can be instantiated (`Instantiating A Module` below) inside of another module as long as there are no infinitely recursive references.

### 3.4.5 Building A Module

A module is a series of parameters, declarations, assignments, logic, and conditional statements. All declarations in a module must appear at the beginning of the module, before any other statements. Here is a brief example:

```module m2(input enable, in[4]; output out[4]) {
    parameter p = 2;
    register r[4] = 0;
    if(posedge) {
        if(reset) {
            r = 0000b; // equivalent to r = 0;
        }
    } else if(enable == 1b) {
```
\[
\begin{align*}
  r &= \text{in}; \\
  \text{out} &= r;
\end{align*}
\]

This is a simple four-bit latch. On every positive clock edge, if the \texttt{enable} bit is set to 1, the output will be set to the input. If the \texttt{enable} bit is set to 0, the output will remain unchanged. However, if the module’s \texttt{reset} bit is 1 when the clock edge is detected, the register values and output will be reset to 0.

This module is reusable. Here is an example of instantiating the module from within another module:

\begin{verbatim}
module m3(input check1, check2, in[4]; output out[4]) {
    wire enabler, resetter;
    m2(enable = enabler, in = in, reset = resetter; out = out);
    if(posedge) {
        enabler = check1;
        resetter = check1 & check2;
    }
}
\end{verbatim}

The enable bit of module \texttt{m2} is 1 if input \texttt{check1} in module \texttt{m3} is 1. If both \texttt{check1} and \texttt{check2} are 1, then the module \texttt{m2} is reset. \texttt{check2} on its own does nothing.

Notice that though \texttt{reset} is not listed as an input of module \texttt{m2}, it is implicit; it can be referred to by keyword \texttt{reset}.

In addition to binding wires to outputs, subsets of outputs can be returned as in traditional languages. To indicate that a module returns \(n\) bits, put the number of bits to be returned (as in a \texttt{wire} declaration) in square brackets after the identifier, including for a single bit returned, unlike wires.

If you wish to return all or a subset of outputs (replacing or in addition to the normal outputs via binding), list them after the \texttt{return} keyword anywhere in a block. A single block may not have multiple \texttt{return} keywords.

\begin{verbatim}
module m10[5] (input in[5]; output out[3]) {
    ... \\
    out = ... ; \\
    return concat(out,in[1],in[2]);
}
\end{verbatim}

These can then be used as follows: \texttt{wire w11[5]; w11 = m10(...);}

### 3.4.6 Statements

A semicolon is necessary after a statement in Verishort. Because whitespace has no effect, it is necessary to have semicolons to signal the end of a statement.
Examples from previous sections:
wire w3 = ~w2[0];
wire c[2] = 01b;

3.4.7 Conditional Statements
Conditional statements work just as they do in the C programming languages with if and else. Following an if, an expression is placed within parenthesis. An expression returning 0 is false; all other values are true. An else block attaches itself to the closest else-less if block.

wire gate = 1;
wire b[2];
if (gate & 1b > 0b) {
    b = 10b;
}
else {
    b = 01b;
}

The power of conditional statements come in their ability to use the clock. However, the clock edge must reside in the outermost if block and may not be followed by an else clause. For example, an incrementer:

register a[8] = 0;
if (posedge) {
    a = a + 1b;
}

3.4.8 Case/Switch Structures
Case structures use the case keyword and work similarly to the switch statement in C. The main difference is that the case structure in Verishort does not provide fall-through or default behavior.

This is especially useful when the user wants to test for conditions on certain bits but does not care about the value of other bits. Those bits can be replaced with x instead of 1 or 0.

Inside of a case block, the condition is followed by a colon, then followed by the resulting statement, followed by a semicolon.

If-else statements and for-loops cannot appear inside case structures:

wire w[3] = ... // also assume a 3-bit output out
case(w) {
    1x1b : {out = 11b; out2 = 10b;};
Note that unlike C and Java, there is no `default`.

### 3.4.9 For loops

For loops in Verishort are different from for loops in C. Instead of being used to repeat a task multiple times, they are instead used to repeat tedious code. For example, to wire every other bit of the wire `a` to a module and output the result to `b`:

```plaintext
wire a[32];
wire b[16];
for (i = 0; i < 16; i = i + 1) {
    example_module(in=a[i*2], out=b[i]);
}
```

Only one loop variable is permitted, and it may not be modified inside the loop body.

### 3.5 Scope

Verishort tends to be a linear language, with very little dependence on lexical scope and linkage. That being said, Verishort still limits scope for certain conditions. Importantly, data declared within one module is not available outside of that module except for port bindings.

### 3.6 Preprocessor

Like the `#define` directive in C, the `parameter` keyword can be used in Verishort to replace numbers before compile time. For example, in the following code:

```plaintext
parameter const = 10;
wire c[4] = const;
```

The `const` identifier behaves as if it were replaced with the number `10` before compilation. As with any assignment, notice that `c` contains the correct number of bits to hold the constant.

To reduce the repetitiveness of Verilog, we automatically assume `clock` and `reset` ports in all modules. A `reset` input port is included in all modules to the effect that asserting a reset will set all registers back to 0.

### 3.7 Standard Library Modules

Because of the repetitiveness of many aspects of hardware design, the Verishort language includes standard modules of many commonly used electronic components in the hope of reducing some tedium. The definitions given below are templates. An actual module from the template can be declared with the following module declaration:

```plaintext
module jkl_16 = JKL[16];
```
After it is declared, jkl_16 may now be used as any normal module.

3.7.1 Latches

module JKL[n] (input J[n], K[n], E[n]; output Q[n], QNOT[n])
module DL[n] (input D[n], E[n]; output Q[n], QNOT[n])
module TL[n] (input T[n], E[n]; output Q[n], QNOT[n])

3.7.2 Flipflops

module DFF[n] (input D[n], S[n]; output Q[n], QNOT[n])
module TFF[n] (input T[n]; output Q[n], QNOT[n])
module JKFF[n] (input J[n], K[n]; output Q[n], QNOT[n])

4 Project Plan

4.1 Team Responsibilities

4.1.1 Round-Robin Dictatorship

At some point in this project, each of our members has been our dictator. Leadership switched hands especially when moving between important phases in the project. For example, Elba was dictator for the failed nation of Natural. Anish usurped the leadership position soon after Verishort was established. Next was Scott, when regular meetings started getting iffy during midterms. Soon after, our final and present dictator took hold, our Dear Leader, Ruijie Song.

We mostly worked together during the large blocks of time which we held our meetings. All sections were written with all members present to catch mistakes or give suggestions.

4.2 Project Timeline

Our goals timewise were as follows:

<table>
<thead>
<tr>
<th>Week of</th>
<th>Work Contributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 27, 2010</td>
<td>Submit project proposal.</td>
</tr>
<tr>
<td>November 1, 2010</td>
<td>Edit and submit LRM.</td>
</tr>
<tr>
<td>November 8, 2010</td>
<td>Complete parser and establish grammar.</td>
</tr>
<tr>
<td>November 29, 2010</td>
<td>Finish code generation and error recovery.</td>
</tr>
<tr>
<td>December 6, 2010</td>
<td>Write and test cases.</td>
</tr>
<tr>
<td>December 13, 2010</td>
<td>Write report and submit final coding.</td>
</tr>
</tbody>
</table>

4.3 Software Development Environment

For version control, our team signed up for accounts on Github.com and used git from our personal computers to collaborate on the project. All code was written in O’Caml and compiled with ocamlc, ocamllex, and ocamlyacc with accompanying makefiles to make compiling and linking easier.

As stated before, we mostly worked for large segments of time during our designated weekly meeting times, where we all got together in a room and pushed out code together that all could
see and scrutinize. Anish would connect a large 24” display to his computer, so we could see what was coded and written.

For developing in Verishort:
To simplify writing Verishort code, we created a syntax and keyword highlighter for gedit. Though we planned to reach out to other text editors like vim and emacs, as always, we did not have time for it.

4.4 Project Log

<table>
<thead>
<tr>
<th>Week of</th>
<th>Work Contributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 13, 2010</td>
<td>Our team got together, and brainstormed possible projects.</td>
</tr>
<tr>
<td>September 20, 2010</td>
<td>Natural, our first attempt at a project, was born.</td>
</tr>
<tr>
<td>September 27, 2010</td>
<td>A written proposal for Natural was submitted.</td>
</tr>
<tr>
<td>October 11, 2010</td>
<td>Met with Professor Edwards and got recommended to establish a new project.</td>
</tr>
<tr>
<td></td>
<td>Began looking into Verilog a possible project replacement.</td>
</tr>
<tr>
<td>October 18, 2010</td>
<td>Verishort, our eventual final project was born.</td>
</tr>
<tr>
<td>November 1, 2010</td>
<td>Final writing of LRM finished and submitted.</td>
</tr>
<tr>
<td>November 22, 2010</td>
<td>Parser and final grammar established and generated.</td>
</tr>
<tr>
<td>November 29, 2010</td>
<td>Code generation initiated.</td>
</tr>
<tr>
<td>December 6, 2010</td>
<td>Error recovery and code generation completed.</td>
</tr>
<tr>
<td>December 13, 2010</td>
<td>Test cases and programs created and tested.</td>
</tr>
<tr>
<td>December 20, 2010</td>
<td>Report and final check-through done.</td>
</tr>
</tbody>
</table>

5 Architectural Design

5.1 Architecture

Like any other compiler, Verishort uses a lexer, a parser, and a translator. The lexer parses the input into tokens. The parser parses the tokens to generate an abstract syntax tree for Verishort, with position information for future error reporting. It also checks for syntactical correctness. Then, the translator creates a new Verilog-friendly syntax tree by translating the original Verishort tree, accounting for semantic correctness and generating all the necessary temporaries. Finally, the code generator takes the Verilog syntax tree and outputs syntactically correct Verilog code.
5.2 Error Recovery

Syntactic errors are reported during parsing. The grammar has a few rules designed to catch specific errors, but it is difficult to anticipate them, and unexpected errors are reported immediately. The following code, for example, attempts to anticipate the problem with attaching else blocks to clock edge conditions:

```cpp
IF LPAREN condition_clock RPAREN stmt ELSE stmt { raise (Parse_Failure(‘‘Clock edge if statements may not have else clauses.’’, Parsing.symbol_start_pos ())) }
```

Semantic errors, such as assignment width mismatches, are reported during translation. Errors are reported with the input file name, line number, and character position, which is recorded during parsing and passed to the translator in the AST. The Verishort compiler is fail-fast: once an error is discovered, the compiler reports the error and terminates without attempting to continue compilation.

6 Testing Plan

We had two strategies to test our language. Our first was to build small 33 test cases. For example, there was a test for implementing multiple modules, a test for block comments, and a test for gating clocks. Initially, we planned to implement a test bench in Perl using the Posix command `diff` which would loop through these test cases to compare the output of our compiler with hand-generated expected Verilog code. This worked until we actually started running the compiler and found minor but crucial differences between the output of the compiler and the generated Verilog code. The differences did not indicate errors but rather differences in ordering and naming (we began putting an underscore in front of all variable names, for example, to avoid conflicts with Verilog reserved words). For this reason, this system was no longer worth maintaining as it was more efficient to manually verify contents due to continued differences in naming and ordering. However, the Verishort test cases remained useful and we began using them as miniature regression tests to ensure that the compiler was handling output correctly.

Our second strategy was to build small but non-trivial programs and ensure that our compiler translated them into working Verilog and ran them with stim files. Three of these files were built: helloworld, gcd, and a memoryarray. These can be run by running `make helloworld`, `make gcd`, `make memoryarray`. (These are in the Makefile for the convenience of the tester.)

7 Lessons Learned

The biggest lesson that (we imagine) is stated every year is to start early, stay ahead, and don’t wait until the last minute. There was a time crunch at the end, even though we were relatively ahead of schedule all semester long. In the end, even when it seems most of the compiler is built, there are a lot of loose ends to tie up, and random bugs to squash. Just as you may think you are finished, two or three more errors will pop up.

Concerning Verishort, we realized it was very bad to try to improve on a language that we were not entirely familiar with. We ended up having to rely on Scott, who is taking a design class which required Verilog, for every single detail on Verilog. We all knew a bit about the language, but not very specific details that needed clarifying every now and then. Had we all known and written Verilog code on a regular basis, we wouldn’t have had a huge learning curve for language knowledge. In addition, we should have had a clearer vision for what subset of the Verilog language we were
going to implement.

Granted, we had a few weeks shaven off our development time since we switched our project from Natural to Verishort, but even then, we should have chosen more carefully in that respect. It was a challenge, and the challenge overwhelmed.
8 Codebase

8.1 Ast.ml

type op = Plus | Minus | Multiply | Modulus | Eq | Ne | Ge | Gt | Le | Lt | And | Or | Xor | Nand | Nor | Xnor | Lshift | Rshift | Not

type parameter = string * int * Lexing.position

type expr =
  DLiteral of int * Lexing.position
  | BLiteral of string * Lexing.position
  | Lvalue of lvalue * Lexing.position
  | Binop of expr * op * expr * Lexing.position
  | Reduct of op * lvalue * Lexing.position
  | Unary of op * expr * Lexing.position
  | Concat of concat_item list * Lexing.position
  | Inst of string * binding_in list * binding_out list * Lexing.position
  | Reset of Lexing.position
  | Noexpr of Lexing.position
  | ConcatBLiteral of int * string
  | ConcatLvalue of int * lvalue

and concat_item =
  | ConcatBLiteral of int * string
  | ConcatLvalue of int * lvalue

and lvalue =
  Identifier of string
  | Subscript of string * expr
  | Range of string * expr * expr

and binding_in = string * expr

and binding_out = string * lvalue

type condition = Posedge | Nedge | Expression of expr

type statement =
  Nop of Lexing.position
  | Expr of expr * Lexing.position
  | Block of statement list * Lexing.position
  | If of condition * statement * statement * Lexing.position
  | Case of lvalue * case_item list * Lexing.position
  | Return of expr * Lexing.position
  | For of string * expr * expr * statement * Lexing.position
  | Assign of lvalue * expr * Lexing.position

and case_item = string * statement * Lexing.position

type decl_type = Wire | Reg

type declaration = {
  decltype : decl_type;
  declname : string;
  declwidth : int;
  init : expr;
  declpos : Lexing.position;
}

type id_with_width = string * int * Lexing.position

type mod_decl= {
  modname : string; (* Name of the module *)
  inputs : id_with_width list;
  outputs : id_with_width list;
  statements : statement list;
  parameters : parameter list;
  declarations : declaration list;
  returnwidth : int;
  libmod : bool;
  libmod_name : string;
module StringMap = Map.Make(String)

(* Environment information *)

let string_map_args map mods = List.fold_left (fun m mod1 -> StringMap.add mod1.modname (mod1.inputs, if mod1.returnwidth > 0 then ("return", mod1.returnwidth, Lexing.dummy_pos) :: mod1.outputs) else mod1.outputs) m) map mods

let string_map_params map mods = List.fold_left (fun m mod1 -> StringMap.add mod1.modname mod1.parameters m) map mods

let string_maplocals map mods = List.fold_left (fun m mod1 -> StringMap.add mod1.modname mod1.declarations m) map mods

let string_map_returns map mods = List.fold_left (fun m mod1 -> StringMap.add mod1.modname mod1.returnwidth m) map mods

let get_param_value (_, x, _) = x

let rec get_param_tuple name lst = match lst with [] -> raise Not_found | (s, i, p)::tl -> if s = name then (s, i, p) else get_param_tuple name tl

let get_param_mod_name_param_name env = get_param_value (get_param_tuple param_name mod_name env.param_map)

let rec get_arg_tuple name lst = match lst with [] -> raise Not_found | (s, i, _)::tl -> if s = name then (s, i) else get_arg_tuple name tl

let rec invert_binary_actual n x = if n < 0 then x else (x.[n] < '1' then '0' else '1'); invert_binary_actual (n-1) x and invert_binary x = invert_binary_actual (String.length x - 1) (String.copy x)

let rec add_one_actual n x = if n < 0 then x (*discard overflow bit*) else if x.[n] = '1' then (x.[n] <- '0'; add_one_actual (n-1) x) else (x.[n] <- '1'; x) and add_one x = add_one_actual (String.length x - 1) (String.copy x)
let rec eval_expr mod_name env = function
DLiteral(x, _) -> Int64.of_int x
BLiteral(x, pos) -> (try
  if x[0] = '0' then Int64.of_string("0b"^x)
  else Int64.neg(Int64.of_string("0b"^add_one(invert_binary x)))
with Failure(_, pos) -> raise (Parse.Failure("Binary literals may not exceed 64 bits.", pos)))
Lvalue(x, pos) -> (match x with
  Identifier(id) -> (try Int64.of_int(get_param mod_name id env)
    with Not_found -> raise (Parse.Failure("Expression cannot be evaluated at compile time.", pos)))
  | _ -> raise (Parse.Failure("Expression cannot be evaluated at compile time.", pos)))
Binop(e1, op, e2, pos) -> (match op with
  Plus -> Int64.add(eval_expr mod_name env e1)(eval_expr mod_name env e2)
  Minus -> Int64.sub(eval_expr mod_name env e1)(eval_expr mod_name env e2)
  Multiply -> Int64.mul(eval_expr mod_name env e1)(eval_expr mod_name env e2)
  Modulus -> Int64.rem(eval_expr mod_name env e1)(eval_expr mod_name env e2)
  Eq -> if (Int64.compare(eval_expr mod_name env e1)(eval_expr mod_name env e2)) = 0
    then Int64.one else Int64.zero
  Ne -> if (Int64.compare(eval_expr mod_name env e1)(eval_expr mod_name env e2)) <> 0
    then Int64.one else Int64.zero
  Ge -> if (Int64.compare(eval_expr mod_name env e1)(eval_expr mod_name env e2)) >= 0
    then Int64.one else Int64.zero
  Lt -> if (Int64.compare(eval_expr mod_name env e1)(eval_expr mod_name env e2)) < 0
    then Int64.one else Int64.zero
  Gt -> if (Int64.compare(eval_expr mod_name env e1)(eval_expr mod_name env e2)) > 0
    then Int64.one else Int64.zero
  And -> Int64.logand(eval_expr mod_name env e1)(eval_expr mod_name env e2)
  Or -> Int64.logor(eval_expr mod_name env e1)(eval_expr mod_name env e2)
  Xor -> Int64.logxor(eval_expr mod_name env e1)(eval_expr mod_name env e2)
  Not -> Int64.lognot(eval_expr mod_name env e1)
  Lshift -> Int64.shift_left(eval_expr mod_name env e1)(Int64.to_int(eval_expr mod_name env e2))
  Rshift -> Int64.shift_right(eval_expr mod_name env e1)(Int64.to_int(eval_expr mod_name env e2))
  | _ -> raise (Parse.Failure("Expression cannot be evaluated at compile time.", pos)))
Signext(_, _, pos) -> raise (Parse.Failure("Sign extension cannot be used in the current context.", pos))
Reduct(_, _, pos) -> raise (Parse.Failure("Expression cannot be evaluated at compile time.", pos))
Unary(op, exp, pos) -> (match op with
  Minus -> Int64.neg(eval_expr mod_name env exp)
  Plus -> (eval_expr mod_name env exp)
  | _ -> raise (Parse.Failure("Internal compiler error 002: contact manufacturer for assistance.", pos)))
Concat(_, pos) -> raise (Parse.Failure("Concatenation cannot be used in the current context.", pos))
Inst(_, _, pos) -> raise (Parse.Failure("Expression cannot be evaluated at compile time.", pos))
Reset(pos) -> raise (Parse.Failure("Expression cannot be evaluated at compile time.", pos))
Noexpr(pos) -> raise (Parse.Failure("Internal compiler error 001: contact manufacturer for assistance.", pos))
let tuples = StringMap.find mod_name env.arg_map in
get_arg_tuple arg_name((fst tuples) @ (snd tuples))

let check_modinfo lst1 mods =
List.fold_left (fun lst mod -> if List.mem mod1 modname lst then raise (Parse_Failure("Duplicate module name.", mod1.modpos)) else if mod1.returnwidth < 0 then raise (Parse_Failure("Invalid return width.", mod1.modpos)) else mod1.modname :: lst) lst1 mods

let check_unique_ids_in_module env mod1 =
let modname = mod1.modname in
let inputslist = List.fold_left (fun lst2 (name, width, pos) -> (* each input *)
  if List.mem name lst2
  then raise (Parse_Failure("Duplicate identifier.", pos))
  else name :: lst2) [] (fst (StringMap.find mod_name env.arg_map))
in let argslst = List.fold_left (fun lst2 (name, width, pos) -> (* each output *)
  if List.mem name lst2
  then raise (Parse_Failure("Duplicate identifier.", pos))
  else name :: lst2) argslst (snd (StringMap.find mod_name env.param_map))
in List.fold_left (fun lst2 decl -> (* each decl in each mod *)
  if List.mem decl.declname lst2
  then raise (Parse_Failure("Duplicate identifier.", decl.declpos))
  else decl.declname :: lst2) argslst (StringMap.find mod_name env.local_map)

let rec get_min_bit_width x =
if Int64.compare Int64.zero x = 0 || Int64.compare Int64.one x = 0 || Int64.compare Int64.minus_one x = 0 then 1
else 1 + get_min_bit_width(Int64.div x (Int64.of_int 2))

let rec get_arg_tuple name lst =
match lst with
  [] -> raise Not_found
  | (s, i, _)::tl -> if s = name then (s,i) else get_arg_tuple name tl

let get_arg_modname arg_name env =
let tuples = StringMap.find mod_name env.arg_map in
get_arg_tuple arg_name (fst tuples @ snd tuples)

let get_input_modname input_name env =
let tuples = StringMap.find mod_name env.arg_map in
get_arg_tuple input_name (fst tuples)

let get_output_modname output_name env =
let tuples = StringMap.find mod_name env.arg_map in
get_arg_tuple output_name (snd tuples)

let rec get_local_tuple name lst =
match lst with
  [] -> raise Not_found
  | hd::tl -> if hd.declname = name then (hd.declname, hd.declwidth) else get_local_tuple name tl

let rec get_local_decl name lst =
match lst with
let get_local_all mod_name local_name env =
    get_localDecl local_name (StringMap.find mod_name env.local_map)
let get_local_mod_name local_name env =
    get_localTuple local_name (StringMap.find mod_name env.local_map)

let get_lvalue_name = function
  | Identifier(n) -> n | Subscript(n, _) -> n | Range(n, _, _) -> n

let change_im_lvalue_name newname = function
  ImSubscript(_, s) -> ImSubscript(newname, s) | ImRange(_, up, lo) -> ImRange(newname, up, lo)

let check_valid_lvalue environ mod_name lvalue_id pos =
  (* arg map, local map *)
  try get_arg mod_name lvalue_id environ
  with Not_found -> try get_local mod_name lvalue_id environ with Not_found -> raise (Parse.Failure("Undefined identifier.", pos))
  let (id, width) = check_valid_lvalue environ immod.im_modname s pos in
  let subscr1 = Int64.to_int (eval_expr immod.im_modname environ expr1) in
  let subscr2 = Int64.to_int (eval_expr immod.im_modname environ expr2) in
  if subscr1 < 0 || subscr2 < 0 || subscr1 >= width || subscr2 >= width then raise (Parse.Failure("Bus index out of bounds.", pos))
  else try gtl v a l e environ modname immod lvalue pos =
            match (to_im_lvalue environ immod lvalue pos) with
            | ImSubscript(_, _) -> 1
            | ImRange(_, upper, lower) -> (upper - lower + 1)

let rec check_im_mod_local_actual name = function
  | [] -> false
  | (_, declname, _) :: tl -> if name = declname then true else check_im_mod_local_actual name tl

let get_lvalue_bit_range environ immod lvalue pos = match (to_im_lvalue environ immod lvalue pos) with
  | ImSubscript(_, s) -> (s, s)
  | ImRange(_, upper, lower) -> (upper, lower)

let to_im_op = function
  Plus -> ImPlus | Minus -> ImMinus | Multiply -> ImMultiply | Gt -> ImGt
  | Modulus -> ImModulus | Eq -> ImEq | Ne -> ImNe | Ge -> ImGe | Le -> ImLe
  | Lt -> ImLt | And -> ImAnd | Or -> ImOr | Xor -> ImXor | Nand -> ImNand
  | Nor -> ImNor | Xnor -> ImXnor | Lshift -> ImLshift | Rshift -> ImRshift | Not -> ImNot
let rec get_max_bit_width_expr environ immod expr = match expr with
  DLiteral(d, _) -> 64
| BLiteral(b, _) -> String.length b
| Lvalue(1, pos) -> get_lvalue_length environ immod 1 pos
| Binop(e1, op, e2, pos) -> (match op with
  Plus -> max (get_max_bit_width_expr environ immod e1) (get_max_bit_width_expr environ immod e2)
| Minus -> max (get_max_bit_width_expr environ immod e1) (get_max_bit_width_expr environ immod e2)
| Multiply -> (get_max_bit_width_expr environ immod e1) + (get_max_bit_width_expr environ immod e2) - 1
| Modulus -> get_max_bit_width_expr environ immod e2
| Eq -> 1 | Ne -> 1 | Ge -> 1 | Gt -> 1 | Le -> 1 | Lt -> 1
| And -> max (get_max_bit_width_expr environ immod e1) (get_max_bit_width_expr environ immod e2)
| Or -> max (get_max_bit_width_expr environ immod e1) (get_max_bit_width_expr environ immod e2)
| Xor -> max (get_max_bit_width_expr environ immod e1) (get_max_bit_width_expr environ immod e2)
| Lshift -> get_max_bit_width_expr environ immod e1 | Rshift -> get_max_bit_width_expr environ immod e1
| _ -> raise (Parse_Failure("Internal compiler error 003: contact manufacturer for assistance.", pos))

| Signext(bits, expr, _) -> bits
| Reduct(op, lvalue, _) -> 1
| Unary(_, expr, _) -> get_max_bit_width_expr environ immod expr
| Concat(lst, pos) -> List.fold_left (fun orig x -> (match x with
  | ConcatBLiteral(time, lit) -> orig + time * (String.length lit)
| ConcatLvalue(time, lvalue) -> orig + time * (get_lvalue_length environ immod lvalue pos)) 0 lst
| Inst(str, bindlst1, bindlst2, pos) -> (try StringMap.find str environ.return_map
  with Not_found -> raise (Parse_Failure("Undefined module name.", pos)))
| Reset(_) -> 1
| Noexpr(_) -> 0

let rec get_min_bit_width_expr environ immod expr = match expr with
  DLiteral(d, _) -> get_min_bit_width (Int64.of_int d)
| BLiteral(b, _) -> String.length b
| Lvalue(1, pos) -> get_lvalue_length environ immod 1 pos
| Binop(e1, op, e2, pos) -> (match op with
  Plus -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ immod e2)
| Minus -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ immod e2)
| Multiply -> (get_min_bit_width_expr environ immod e1) + (get_min_bit_width_expr environ immod e2) - 1
| Modulus -> get_min_bit_width_expr environ immod e2
| Eq -> 1 | Ne -> 1 | Ge -> 1 | Gt -> 1 | Le -> 1 | Lt -> 1
| And -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ immod e2)
| Or -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ immod e2)
| Xor -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ immod e2)
| Xnor -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ immod e2)
| Lshift -> get_min_bit_width_expr environ immod e1 | Rshift -> get_min_bit_width_expr environ immod e1
| _ -> raise (Parse_Failure("Internal compiler error 003: contact manufacturer for assistance.", pos))

| Signext(bits, expr, _) -> bits
| Reduct(op, lvalue, _) -> 1
| Unary(_, expr, _) -> get_min_bit_width_expr environ immod expr
| Concat(lst, pos) -> List.fold_left (fun orig x -> (match x with
(Translate an AST expression into an IM expression *)

let rec translate_expr_environment = match expr with
  DLiteral (b, pos) -> (immod, ImLiteral (try
    if b = '0' then Int64.of_string "$0b" b
    else Int64.neg (Int64.of_string "$0b$" (add_one (invert_binary b)))
  with Failure (_) -> raise (Parse_Failure "$Binary literals may not exceed 64 bits", pos)), String.length lit)
  Lvalue (l, pos) -> (immod, ImLvalue (to_im_lvalue environ immod l pos), count)
  Binop (el, op, e2, pos) -> let (immod1, imexp1, count1) = translate_expr_environment immod el
    count in
      conditional in
        let (immod2, imexp2, count2) = translate_expr_environment immod1 e2
        count1 in
          conditional in
            (immod2, ImBinop (imexp1, to_im_op op, imexp2), count2)
  | Signext (bits, exp, pos) -> (* generate a temporary wire to store the value of expr. Then use the concatenation syntax. *)
    let width = get_min_bit_width_expr environ immod in
    if width > bits then raise (Parse_Failure "$Cannot sign extend something into fewer bits than the original.", pos) else
      let (immod1, imexp1, count1) = translate_expr_environment immod exp count in
      conditional in
        (immod1, ImUnary (op, expr environ immod1 e2
        count1 in
          conditional in
            (immod2, ImLiteral (try
              if lit = '0' then Int64.of_string "$0b$" b
              else Int64.neg (Int64.of_string "$0b$" (add_one (invert_binary b)))
            with Failure (_) -> raise (Parse_Failure "$Binary literals may not exceed 64 bits", pos)), String.length lit)
  Unary (op, expr, pos) -> let (immod1, imexp1, count1) = translate_expr_environment immod exp count in
    conditional in
      (immod1, ImUnary (op, expr environ immod1 e2
      count1 in
        conditional in
          (immod2, ImLiteral (try
            if lit = '0' then Int64.of_string "$0b$" b
            else Int64.neg (Int64.of_string "$0b$" (add_one (invert_binary b)))
          with Failure (_) -> raise (Parse_Failure "$Binary literals may not exceed 64 bits", pos)), String.length lit)
  ConcatByLiteral (time, lit) -> if time <= 0 then raise (Parse_Failure "$Replication must be at least one time.", pos) else
    let immod3 = { immod2 with im_declarations = (ImWire "$im\_" (string_of_int count2), String.length lit - 1, 0), ImLiteral (try
        if lit = '0' then Int64.of_string "$0b$" b
        else Int64.neg (Int64.of_string "$0b$" (add_one (invert_binary lit)))
      with Failure (_) -> raise (Parse_Failure "$Binary literals may not exceed 64 bits", pos)), String.length lit)
        String.length lit - 1, 0) ) } immod2.im_assignments
    in
      (immod3, ImConcatValue (time, ImRange "$im\_" (string_of_int count2), String.length lit - 1, 0)), ImConcatValue (time, ImRange "$im\_" (string_of_int count2), String.length lit - 1, 0))
  ConcatLvalue (time, lvalue) -> if time <= 0 then raise (Parse_Failure "$Replication must be at least one time.", pos) else
    let immod3 = { immod2 with im_declarations = (ImWire "$im\_" (string_of_int count2), String.length lit - 1, 0), ImLiteral (try
      if lit = '0' then Int64.of_string "$0b$" b
      else Int64.neg (Int64.of_string "$0b$" (add_one (invert_binary lit)))
    with Failure (_) -> raise (Parse_Failure "$Binary literals may not exceed 64 bits", pos)), String.length lit)
      String.length lit - 1, 0) ) } immod2.im_assignments
  in
    (immod3, ImConcatValue (time, ImRange "$im\_" (string_of_int count2), String.length lit - 1, 0)), ImConcatValue (time, ImRange "$im\_" (string_of_int count2), String.length lit - 1, 0))
  | Inst (othermod, bindlst1, bindlst2, pos) -> if in Conditional then raise (Parse_Failure "$Modules may not be instantiated inside conditional blocks.", pos) else
    (immod1, ImConcat (List.rev concatlst1), count1)
  | Reset (_) -> (immod, ImLvalue (ImRange "$reset", 0, 0), count)
  | Noexpr (_) -> (immod, ImNoexpr, count)
  | Inst (othermod, bindlst1, bindlst2, pos) -> raise (Parse_Failure "$Modules may not be instantiated inside conditional blocks.", pos)

(* Check bindings *)
let (immod1, count1, converted_bindings_in) = convert_bindings_in environ count immod othermod bindlst1 pos in conditional in
let (immod2, count2, converted_bindings_out) = convert_bindings_out environ count immod othermod bindlst2 pos in
try
    let returnwidth = StringMap.find othermod environ.return_map in
    if returnwidth = 0 then
        (* No return value – just do instantiation *)
        ( { immod2 with im.instantiations = (othermod, converted_bindings_in, converted_bindings_out) :: immod2.im.instantiations; }, ImNoexpr, count2)
    else
        (* return value – Do the following: *)
        (* Generate a bus for this purpose with the name _im_. followed by count, then increment count... *)
        (Add binding between "return" port and the new bus, *)
        let new_bus_name = ".imexp_" (string_of_int count2) in
        let new_bindings_out = ("return", ImValue(ImRange(new_bus_name, returnwidth - 1, 0)) :: converted_bindings_out in
        ( { immod2 with im.instantiations = (othermod, converted_bindings_in, new_bindings_out) :: immod2.im.instantiations;
            im_declarations = (ImWire, new_bus_name, returnwidth) :: immod2.im.declarations;
        },
        ImValue(ImRange(new_bus_name, returnwidth - 1, 0), count2 + 1))
        with Not_found -> raise (Parse_Failure("Undefined module name.", pos))
    )
and add_param_pos startpos endpos paramname lst pos = if startpos < endpos then lst
            else let newparamplace = paramname ` (string_of_int startpos) in
                if List.memb newparamplace lst then raise (Parse_Failure("Duplicate binding.", pos))
            else add_param_pos (startpos - 1) endpos paramname (newparamplace::lst) pos
( * convert_bindings_in: env -> int -> im_moddecl -> string -> binding_in list -> im_moddecl
    * int * im_assignment list * )
(* Check: no duplicate bindings (but partial binding is OK, and do not have to bind anything ). *)
(* Expr to be assigned to will be translated into ImExpr *)
(* The expr bound to can be anything. Do not check for uninitialized wires. *)
and convert_bindings_in environ count immod othermod bindlst pos in conditional =
    let (immod1, count1, list1, _) = (List.fold_left (fun (mod1, cnt1, bnd1, lst1) (name, expr) -> (let (_, width) = get_input othermod name environ in
      let lst2 = add_param_pos (width - 1) 0 name lst1 pos in
      if width < get_min_bit_width expr environ immod expr1 then raise (Parse_Failure("Binding width mismatch.", pos))
      else if width > get_max_bit_width expr environ immod expr1 then raise (Parse_Failure("Binding width mismatch.", pos))
      else let (mod2, exp2, cnt2) = translate_expr environ mod1 expr cnt1 in conditional in
      (mod2, cnt2, (name, exp2) :: bnd1, lst2))
    ) bindlst) in (immod1, count1, list1)
( * convert_bindings_out: env -> im_moddecl -> string -> binding_out list -> im_moddecl * int
    * im_assignment list * )
(* Check: no duplicate bindings to ports (but partial binding is OK, and do not have to bind anything ). *)
(* Note that duplicate assignments to wires will result in undefined behavior. *)
(* Check that the target of the assignment is a wire – "binding" a reg to an output does not make any sense. *)
and convert_bindings_out environ count immod othermod bindlist pos =
    let (immod1, count1, list1, _, _) = (List.fold_left (fun (mod1, cnt1, bnd1, lst1) (name, lval2) ->
      try (try
          let result = get_local_all immod.im.name lval2 environ in
          if result.dectype = Reg then raise (Parse_Failure("Cannot bind output port to registers.", pos)) else ()
        with Not_found -> ignore (get_output immod.im.name (get_lvalue_name lval2) environ));
      let expr = Lvalue(lval2, Lexing.dummy_pos) in
      let (_, width) = get_output othermod name environ in
...
let lst2 = add_param_pos (width - 1) 0 name lst1 pos in
if width <> get_min_bit_width_expr environ immod expl then raise (Parse_Failure("Binding width mismatch.", pos))
else let (mod2, exp2, cnt2) = translate_expr environ modl expl (modulecnt false in
  (mod2, cnt2, (name, exp2) :: bind1. lst2))
with Not_found -> raise (Parse_Failure("Undefined identifier.", pos))
(immod, count, [], []) bindlst) in (immod1, count1, lst1)

(* translate_stmt: environ -> im_moddecl -> stmt -> int -> bool -> im_moddecl *)
(* im_always: stmt list * int *)
(* or, in other words, we are in a top-level statement, then statement *)
(* generated is returned through modification to the im_moddecl directly. *)
(* Otherwise, it is returned in the list for incorporation by a top-level statement *)
let rec translate_stmt environ immod vshstmt count in_always = match vshstmt with
  Nop(_, _) -> (immod, [], count)
| Expr(expr, _) -> let (immod1, _, count1) = translate_expr environ immod expr count in_always in (immod1, [], count1)
| Block(lst, _) -> List.fold_left (fun (immod1, stmtlst1, count1) stmt ->
    let (immod2, stmtlst2, count2) = translate_stmt environ immod stmt count1 in_always in
    (immod2, stmtlst1 @ stmtlst2, count2)) (immod, [], count) lst
| If(cond, stmt1, stmt2, pos) -> if in_always && (cond = Posedge || cond = Negedge) then raise (Parse_Failure("Clock edge conditions must be in the outermost if statement.", pos))
else (match cond with
  Posedge -> let (immod1, stmtlst1, count1) = translate_stmt environ immod stmt true in
    ( { immod1 with im_always_poseedge = immod1. im_always_poseedge @ stmtlst1 }, [], count1 )
| Negedge -> let (immod1, stmtlst1, count1) = translate_stmt environ immod stmt true in
    ( { immod1 with im_always_negedge = immod1. im_always_negedge @ stmtlst1 }, [], count1 )
| Expression(expr) -> if in_always then
  (let (immod2, stmtlst2, count2) = translate_stmt environ immod stmt true in
    let (immod3, imexpr, count3) = translate_expr environ immod expr count true in
    (immod3, [ImIf(imexpr, stmtlst1, stmtlst2)], count3))
  else let (immod1, stmtlst1, count1) = translate_stmt environ immod stmt true in
    let (immod2, stmtlst2, count2) = translate_stmt environ immod stmt true in
    let (immod3, imexpr, count3) = translate_expr environ immod expr count false in
    ( { immod3 with im_always_all = ImIf(imexpr, stmtlst1, stmtlst2) :: immod3. im_always_all }, [], count3 )
| Return(expr, pos) -> translate_stmt environ immod (Assign(Identifier("return"), expr, pos)) count in_always
| Case(lvalue, lst, pos) -> let width = get_max_bit_width_expr environ immod (Lvalue(lvalue, pos)) in
  let (newmod, newcount, newlist) = List.fold_left (fun (immod1, count1, lst1) (item, stmt, pos) ->
    if String.length item <> width then raise (Parse_Failure("Width mismatch in case statement.", pos))
  else let (immod2, stmtlst2, count2) = translate_stmt environ immod stmt true in
always then

(param immod.im
lvalue environ pos) : : (StringMap.find mod1.im
found lvalue environ newmod lvalue pos, List.
rev newlist), newcount)
else

({newmod with im_alwaysall = ImCase(to_lvalue environ
im_alwaysall}, [], newcount)

| For(id, init, cond, incr, stmt, pos) ->
| (* How this works: We add the loop variable as a parameter in the local parameter table, then translate the statement.*)
| (* Rinse, repeat. Loop for a maximum of 1024 times. *)
| (* First, make sure that id is not referring to anything else.*)
let _ = get_param immod.im.modname id environ in raise (Parse_Failure("Loop control variable must not have been used previously.", pos))
with Not_found -> (try let _ = get_arg immod.im.modname id environ in raise {
| Parse_Failure("Loop control variable must not have been used previously.", pos))
with Not_found -> (try let _ = get_local immod.im.modname id environ in raise {
| Parse_Failure("Loop control variable must not have been used previously.", pos))
with Not_found -> {
| (* compute the initialization value *)
let firstval = Int64.to_int (eval_expr immod.im.modname environ init) in
| (* build_for: int -> int -> im_moddecl -> int -> im_moddecl * im_always stmt list * int *)
let rec build_for currval loopsleft modi count = (if loopsleft < 0 then raise (Parse_Failure("For loop has run too many times.", pos)) else
| (*Add currval to local parameter table*)
let newenv = {environ with param_map = StringMap.add mod1.im.modname ((id, currval, Lexing.dummy_pos) :: (StringMap.find mod1.im.modname environ.param_map)) environ .param_map} in
let continue = eval_expr mod1.im.modname newenv cond in
if Int64.compare Int64.zero continue = 0 then (mod1, [], count)
else
| let (mod2, stmtlist2, count2) = translate_stmt newenv modi stmt count in always in
let newval = Int64.to_int (eval_expr mod1.im.modname environ incr) in
let (mod3, stmtlist3, count3) = build_for newval (loopsleft - 1) mod2 count2 in
(mod3, stmtlist2 @ stmtlist3, count3)
|)
| in
let (mod4, stmtlist, newcount) = build_for firstval 1024 immod count in
if in_always then (mod4, stmtlist, newcount)
else ( { mod4 with im_alwaysall = List.rev_append stmtlist mod4.im.alwaysall}, [],
newcount)
))))
| Assign(lvalue, expr, pos) -> {
| let ilvalue = to_lvalue environ immod lvalue pos in
| let lvalue_width = get_lvalue_length environ immod lvalue pos in
| let ilvaluename = get_lvalue_name lvalue in
| if lvalue_width < get_min_bit_width_expr environ immod expr then raise (Parse_Failure("Assignment width mismatch.", pos))
else if lvalue_width > get_max_bit_width_expr environ immod expr then raise {
| Parse_Failure("Assignment width mismatch.", pos))
else if in_always then {
try
| let decl = get_local_all immod.im.modname ilvaluename environ in
| if decl.ctype = Reg then
| let (immodl, imexpr, countl) = translate_expr environ immod expr count true in
| (immodl, [ImRegAssign(imlvalue, imexpr)], countl)
| else
| let tempregname = ".reg_" ~ ilvaluename in
| if check_im_mod_local immod tempregname then
| let (immodl, imexpr, countl) = translate_expr environ immod expr count true in
| (immodl, [ImRegAssign(change_im_lvalue_name tempregname imlvalue, imexpr)],
countl)
| else


let immod1 = { immod with im_declarations = (ImReg, tempregname, decl.declwidth) |
  :: immod.im_declarations;
  im_assignments = (ImRange(lvaluename, decl.declwidth - 1, 0), ImLvalue(ImRange(tempregname, decl.declwidth - 1, 0))) :: immod.im_assignments } in

let (immod2, imexpr, count1) = translate_expr environ immod1 expr count true in

(immod2, [ImRegAssign(change_im_lvalue_name tempregname imlvalue, imexpr)],
count1)

with Not_found -> let tempregname = ".reg." lvaluename in

let (_, width) = get_output immod.im_modname lvaluename environ in

if check_im_mod_local immod tempregname then

let (immod1, imexpr, count1) = translate_expr environ immod expr count true in

(immod1, [ImRegAssign(change_im_lvalue_name tempregname imlvalue, imexpr)],
count1)

else

let immod1 = { immod with im_declarations = (ImReg, tempregname, width) :: immod.im_declarations;
  im_assignments = (ImRange(lvaluename, width - 1, 0), ImLvalue(ImRange(tempregname, width - 1, 0))) :: immod.im_assignments } in

let (immod2, imexpr, count1) = translate_expr environ immod expr count true in

(immod2, [ImRegAssign(change_im_lvalue_name tempregname imlvalue, imexpr)],
count1)

) else (try

let decl = get_local_all immod.im_modname lvaluename environ in

if decl.dectype = Reg then raise (Parse_Failure("Cannot assign values to registers outside if and case blocks. Use wires.", pos))

else let (immod1, imexpr, count1) = translate_expr environ immod expr count false in

if imexpr = ImNoexpr then raise (Parse_Failure("Invalid right hand side value in assignment.", pos))

else ((immod1 with im_assignments = (imlvalue, imexpr) :: immod1.im_assignments),
  [], count1)

with Not_found -> let (immod1, imexpr, count1) = translate_expr environ immod expr count false in

if imexpr = ImNoexpr then raise (Parse_Failure("Invalid right hand side value in assignment.", pos))

else ((immod1 with im_assignments = (imlvalue, imexpr) :: immod1.im_assignments),
  [], count1)

)

let rec check_assignment_duplication startpos endpos paramname lst pos = if startpos < endpos then lst

else let newparamplace = paramname " (string_of_int startpos) in

if List.mem newparamplace lst then raise (Parse_Failure("Duplicate assignment of \"\" paramname \"\" (string_of_int startpos) \"\"", pos))

else add_param_pos (startpos - 1) endpos paramname (newparamplace::lst) pos

(* translate_module: env -> mod_decl -> im_mod_decl*)

let translate_module environ vshmod = if vshmod.libmod then { im_modname = vshmod.modname; im_libmod = true; im_libmod_name = vshmod.libmod_name;
  im_libmod_width = vshmod.libmod_width; im_inputs = []; im_outputs = []; im_declarations = [];
  im_instantiations = []; im_alwaysall = []; im_alwaysposedge = []; im_alwaysnegegedge = [] } else (ignore (check_unique_ids_in_module environ vshmod); (* check that all identifiers used in the module are unique *)

let ret = { im_modname = vshmod.modname; im_libmod = false; im_libmod_name = ";
  im_libmod_width = 0; im_inputs = []; im_outputs = []; im_declarations = [];
  im_instantiations = []; im_alwaysall = []; im_alwaysposedge = []; im_alwaysnegegedge = [] } in

(* build up inputs and outputs *)

let ret = { ret with im_inputs = List.map (fun (i, w, _) -> (i, w)) vshmod.inputs } in
let ret = { ret with im_outputs = List.map (fun (i, w, _) -> (i, w)) vshmod.outputs } in
(* special output for returns. Note that "return" is never a valid name because it is a
keyword! *)
let ret = { ret with im_outputs = (if vshmod.returnwidth = 0 then ret.im_outputs else ("return", vshmod.returnwidth)) :: ret.im_outputs } in
(* Build up initial declarations and initializations from the ones provided.
Initializations *must* be to an expression evaluable at compile time.
This constraint and the parameter declaration statements sequence allows us to not
worry about scope. *)
let to_im_decl_type = function Reg -> ImReg | Wire -> ImWire in
let (decls, assigns) = List.fold_left (fun (olddecl, oldassign) decl ->
  ((to_im_decl_type decl.dectype), decl.declname, decl.declwidth)) :: olddecl,
  (match decl.init with
    Noexpr(_, _) -> oldassign
  | x -> (ImRange(decl.declname, decl.declwidth - 1, 0), (let value = eval_expr vshmod.
    modname environ x in
    if get_min_bit_width value > decl.declwidth then
      raise (Parse_Failure("Overflow in initialization.", decl.declpos))
    else ImLiteral(value, decl.declwidth)) :: oldassign )) ([], []) vshmod.declarations in
let ret = { ret with im.declarations = decs; im.assignments = assigns } in
let (immod, _, _) =
  List.fold_left (fun (immodl, _, count) stmt -> translate_stmt environ immodl stmt
count false) (ret, [], 0) vshmod.statements in
let finalmod = { immod with im.alwaysall = List.rev immod.im.alwaysall;
  im.instantiate = List.rev immod.im.instantiate;
  im.assignments = List.rev immod.im.assignments } in
ignore (List.fold_left (fun lst1 lvall -> (match lvall with
  ImSubscript(name, s) -> check_assignment_duplication s s name lst1 vshmod.modpos
  | ImRange(name, up, lo) -> check_assignment_duplication up lo name lst1 vshmod.modpos
  )) []) ([List.map (fun (s, _) -> s) finalmod.im.assignments] @
  List.map (fun (_, exp) -> match exp with ImLiteral(1) -> 1
  | _ -> raise (Parse_Failure("Internal compiler error 005. Contact
    manufacturer for more information.", vshmod.modpos)))
  (List.flatten (List.map (fun (_, 1) -> 1) finalmod.im.instantiate)))}; finalmod

let set_standard_library_module_info modl = if modl.libmod then (
  if modl.libmod.width < 1 then raise (Parse_Failure("Invalid standard module width.", modl.
    modpos))
  else match modl.libmod.name with
    JLX" -> {modl with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.
      dummy_pos)];
      ("J", modl.libmod_width, Lexing.dummy_pos); ("K", modl.
      libmod_width, Lexing.dummy_pos); ("E", modl.libmod_width, Lexing.dummy_pos)];
    output = [("Q", modl.libmod_width, Lexing.dummy_pos);("QNOT", modl.
      libmod_width, Lexing.dummy_pos)];
    returnwidth = modl.libmod_width;
  |"DL" -> {modl with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.
      dummy_pos)];
      ("D", modl.libmod_width, Lexing.dummy_pos); ("E", modl.
      libmod_width, Lexing.dummy_pos)];
    output = [("Q", modl.libmod_width, Lexing.dummy_pos);("QNOT", modl.
      libmod_width, Lexing.dummy_pos)];
    returnwidth = modl.libmod_width;
  |"TL" -> {modl with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.
      dummy_pos)];
      ("T", modl.libmod_width, Lexing.dummy_pos); ("E", modl.
      libmod_width, Lexing.dummy_pos)];
    output = [("Q", modl.libmod_width, Lexing.dummy_pos);("QNOT", modl.
      libmod_width, Lexing.dummy_pos)];
    returnwidth = modl.libmod_width;
  |"DFF" -> {modl with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.
      dummy_pos)];
      ("D", modl.libmod_width, Lexing.dummy_pos); ("S", modl.
      libmod_width, Lexing.dummy_pos)];
    output = [("D", modl.libmod_width, Lexing.dummy_pos);("QNOT", modl.
      libmod_width, Lexing.dummy_pos)];
    returnwidth = modl.libmod_width; }
outputs = [("Q", mod1.libmod_width, Lexing.dummy_pos); ("QNOT", mod1.libmod_width, Lexing.dummy_pos)];
returnwidth = mod1.libmod_width; }
  | "TFF" -> {mod1 with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.dummy_pos)];
              ("T", mod1.libmod_width, Lexing.dummy_pos)];
outputs = [("Q", mod1.libmod_width, Lexing.dummy_pos); ("QNOT", mod1.libmod_width, Lexing.dummy_pos)];
returnwidth = mod1.libmod_width; }
  | "JKFF" -> {mod1 with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.dummy_pos);
                                    ("J", mod1.libmod_width, Lexing.dummy_pos); ("K", mod1.libmod_width, Lexing.dummy_pos)];
              ("J", mod1.libmod_width, Lexing.dummy_pos); ("K", mod1.libmod_width, Lexing.dummy_pos)];
outputs = [("Q", mod1.libmod_width, Lexing.dummy_pos); ("QNOT", mod1.libmod_width, Lexing.dummy_pos)];
returnwidth = mod1.libmod_width; }
  | _ -> raise (Parse.Failure("Unsupported standard module name.", mod1.modpos))
else mod
(* translate: mod_decl_list -> im_mod_decl_list *)
let translate modules =
  (* Check that the module names are consistent and the return widths are valid. *)
  let _ = check_mod_info [] modules in
  (* Set information for standard library module declarations *)
  let modules = List.map set_standard_library_module_info modules in
  (* Build the environment *)
  let environment = {
    arg_map = string_map_args StringMap.empty modules;
    param_map = string_map_params StringMap.empty modules;
    local_map = string_map_locals StringMap.empty modules;
    return_map = string_map_returns StringMap.empty modules;
  }
  in List.map (translate_module environment) modules

8.3 Imst.ml

open Int64

type im_op = ImPlus | ImMinus | ImMultiply | ImModulus | ImEq | ImNe | ImGe | ImLe | ImLt | ImGt | ImAnd | ImOr | ImXor | ImNand | ImNor | ImXnor | ImLshift | ImRshift | ImNot

type im_liter = int64 * int

and im_concat_item =
  ImConcatLit of int * im_liter
| ImConcatValue of int * im_liter
and im_liter =
  ImLiteral of im_liter
| ImValue of im_liter
| ImBinop of im_expr * im_op * im_expr
| ImReduct of im_op * im_liter
| ImUnary of im_op * im_expr
| ImConcat of im_concat_item list
| ImNoexpr
and im_concat =
  ImConcatLit of int * im_liter
| ImConcatValue of int * im_liter
and im_expr =
  im_liter
| ImSubscript of string * int
| ImRange of string * int * int

and im_assignment = im_liter * im_expr

and im_binding = string * im_expr

and im_instantiation = string * im_binding list * im_binding list

and im_always_stmt =
8.4 Imsttocode.ml

```ml
open Ast
open Imst
open Parser
open Asttoimst
open Str

type stringify = function
  ImSubscript ( id, ind ) -> ((mod_id id) ^ "[ " ^ (string_of_int ind) ^ "]")
```
```haskell
let stringify_concat = function
ImConcatLit(replications, literal) -> raise (Failure("duh."))
| ImConcatLvalue(replications, lv) -> string_of_int replications "(" " (stringify_lvalue lv) ")"

let stringify_concats lst =
let concat_string = List.map stringify_concat lst in
"(" (string STRING " ", " concat_string) ")"

let update_inst_count modname inst_map =
if StringMap.mem modname inst_map then StringMap.add modname ((StringMap.find modname inst_map) + 1) inst_map else StringMap.add modname 1 inst_map

let rec print_if_necessary str = function
[] -> "[" ", str ", (" ", str ")]
| (id, _) :: tl -> if id = str then [] else print_if_necessary str tl

let rec stringify_expression = function
ImLiteral(_,_) -> Int64.to_string x
| ImLvalue(x) -> stringify_lvalue x
| ImBinop(x, op, y) -> "(" " (stringify_expression x) " (op.to_string op) " (stringify_expression y) ")"
| ImReduct(op, y) -> "(" " (op.to_string op) " (stringify_expression y) ")"
| ImUnary(op, expr) -> "(" " (op.to_string op) " (stringify_expression expr) ")"
| ImConcat(x) -> stringify_concats x
| ImNoexpr -> ""

let print_inst out inst_map (modname, bindlst1, bindlst2) =
let new_map = update_inst_count modname inst_map in
let ind = StringMap.find modname new_map in
let output_string = (" " " modname " " " " modname " (string_of_int ind) " " ("))
| output_string out (String.concat modname ", " (List.map (fun (id, exp) -> " " id " " (" (stringify_expression exp) ")") (bindlst1 @ bindlst2)) @ (print_if_necessary "clock" bindlst1) @ (print_if_necessary "reset" bindlst1))
| output_string out "\n"
| new_map

let rec print_case out (b, stmt) = output_string out ((string_of_int (String.length b)) " ", b " ", ":\n")
| output_string out "begin\n"
| List.iter (print_statement out) stmt;
| output_string out "end\n"

and print_case_list out lst = List.iter (print_case out) lst

and print_statement out = function
ImNop -> ()
| ImIf(pred, tru, faI) -> output_string out ("if " (" " (stringify_expression pred) " ")\n")
| output_string out "begin\n"
| List.iter (print_statement out) tru;
| output_string out "end\n" else output_string out "begin\n"
| List.iter (print_statement out) faI;
| output_string out "end\n"

| ImCase(lv, csl) -> output_string out ("case(" (" " (stringify_lvalue lv) " ")\n")
| print_case_list out csl; output_string out "endcase\n"
| ImRegAssign(lv, expr) -> output_string out ((stringify_lvalue lv) "=" (stringify_expression expr) ";\n")

let print_module_sig out m =
(* print module sig *)
| (output_string out ("module " (mod.arg_list (fun (name, _) -> " " name) (inputs @ outputs)) in
| output_string out ("module " (mod.arg_list (fun (name, _) -> " " name) (inputs @ outputs)) in
| output_string out ("module " (mod.arg_list (fun (name, _) -> " " name) (inputs @ outputs)) in
| List.iter (fun (id, width) -> output_string out ("input " " (if width == 1 then " else ("[" (string_of_int (width - 1) " ",:0")") " ", " id " (";\n")}) (fst mod.args))
| List.iter (fun (id, width) -> output_string out ("output " " (if width == 1 then " else ("[" (string_of_int (width - 1) " ",:0")") " ", " id " (";\n")}) (snd mod.args))

(* print decls *)
```
let print_decl out (typ, str, width) = output_string out (if typ = ImWire then "wire" else "reg") " " " (if width == 1 then "" else ("[" " (string_of_int (width - 1)) " :0 "]") " " str " ;"
let print_assignment out (lv, expr) = output_string out ("assign " " (stringify_lvalue lv) " = " " (stringify_expression expr) " ;")

(* print a standard library module *)
let print_libmod out libname libwidth actualname =
  let filename = (Filename.current_dir_name "/stdlib/" libname ".v") in
  let chan = open_in filename in
  try
    while true ; do
      let rawline = input_line chan in
      let replname = String.global_replace (Str.regexps libname) actualname rawline in
      let replwidth = String.global_replace (Str.regexps "WIDTHMINUSONE") (if libwidth = 1 then "" else ("[" " (string_of_int (libwidth - 1)) " :0 "]") ) replname in
      output_string out (replwidth " \n")
    done ;
    with End_of_file -> close_in chan
  done;

let print_module out m =
  if m.im_libmod then print_libmod out m.im_libmod_name m.im_libmod_width m.im_modname else
    ( print_module_sig out m;
      List.iter (print_decl out) m.im_declarations;
      List.iter (print_assignment out) m.im_assignments;
      ignore (List.fold_left (print Instantiate out) StringMap.empty m.im_instantiations);
      output_string out "always @ (*) begin\n";
      List.iter (print_statement out) m.im_alwaysall;
      output_string out "if (\_reset) begin\n";
      List.iter (fun (typ, name, _) -> if typ = ImReg then output_string out ("\_ \_ name \_ = 0;") else ()) m.im_declarations;
      output_string out "end\nend\n";
      output_string out "always @ (posedge \_clock) begin\n";
      List.iter (print_statement out) m.im_alwaysposedge;
      output_string out "end\n";
      output_string out "always @ (negedge \_clock) begin\n";
      List.iter (print_statement out) m.im_alwaysnegedge;
      output_string out "end\n";
      output_string out "endmodule\n")

let _ =
  let inname = if Array.length Sys.argv > 1 then Sys.argv.(1) else "stdin" in
  let inchannel = if Array.length Sys.argv > 1 then Pervasives.open_in Sys.argv.(1) else open_stdin in
  let lexbuf = Lexing.from_channel inchannel in
  try
    let sourcecode = List.rev (Parser.program Scanner.token lexbuf) in
    List.iter (print_module outchannel) (translate sourcecode)
    with Parse.Failure(msg, pos) -> print_endline (inname " ": " (string_of_int pos.Lexing.posronym) " ": " (string_of_int (pos.Lexing.pos_lineno - pos.Lexing.pos_column)) " ": " msg )

imsttocode.ml

8.5 Parser.mly
%token CASE CLOCK CONCAT ELSE FOR IF INPUT MODULE NEGEDGE OUTPUT PARAMETER POSEDGE REG RESET RETURN WIRE
%token ASSIGN NOT OR XOR AND NOR XNOR NAND EQ NE GT GE LT LE LSHIFT RSHIFT PLUS MINUS MULTIPLY MODULUS SIGEXT
%token NOELSE UMINUS
%token <string> ID
%token <int> DLIT
%token <string> BLIT
%token <string> XLIT
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%left OR NOR
%left XOR XNOR
%left AND NAND
%left EQ NE
%left GT GE LT LE
%left LSHIFT RSHIFT
%left PLUS MINUS
%left MULTIPLY DIVIDE MODULUS
%right SIGEXT NOT UPLUS

%start program
%type <Ast.program> program

program : /
        | program moddecl { $2 ::= $1 }
        | error { raise (Parse.Failure("General error. You’re screwed.", Parsing.symbol_start_pos ( ))) }
        |
moddecl:
MODULE ID LPAREN input_output RPAREN LBRACE parameter_list decl_list stmt_list RBRACE {
    modname = $2;
    inputs = [ ("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.dummy_pos) ] @ fst $4;
    outputs = snd $4;
    statements = List.rev $9;
    parameters = $7;
    declarations = $8;
    returnwidth = 0;
    libmod = false;
    libmod_name = "";
    libmod_width = 0;
    modpos = Parsing.symbol_start_pos ( );
}

| MODULE ID LBRACKET DLIT RBRACKET LPAREN input_output RPAREN LBRACE parameter_list decl_list stmt_list RBRACE {
    modname = $2;
    inputs = [ ("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.dummy_pos) ] @ fst $7;
    outputs = snd $7;
    statements = List.rev $12;
    parameters = $10;
    declarations = $11;
    returnwidth = $4;
    libmod = false;
    libmod_name = "";
    libmod_width = 0;
    modpos = Parsing.symbol_start_pos ( );
}

| MODULE ID ASSIGN ID LBRACKET DLIT RBRACKET SEMICOLON {
    modname = $2;
    inputs = [ ];
    outputs = [ ];
    statements = [ ];
    parameters = [ ];
}
declarations = [];
returnwidth = 0;
libmod = true;
libmod_name = $4;
libmod_width = $6;
modpos = Parsing.symbol_start_pos();}

input_output:
    INPUT formsals_opt { $2, [[] ]}
| OUTPUT formsals_opt { [[] ], $2 }
| INPUT formsals_opt SEMICOLON OUTPUT formsals_opt { $2, $5 }
| error { raise (Parse_Failure("Module arguments parsing error.", Parsing.
        symbol_start_pos ( )) ) }

id_with_width:
ID LBRACKET DLIT RBRACKET { $1, $3, Parsing.symbol_start_pos () }

id_with_width_opt:
    ID { $1, 1, Parsing.symbol_start_pos () }
| id_with_width { $1 }

id_with_width_opt_list:
    id_with_width_opt { [$1] }
| id_with_width_opt_list COMMA id_with_width_opt { $3 :: $1 }

formsals_opt:
    /* nothing */ { [] }
| id_with_width_opt_list { List.rev $1 }

parameter_list:
    /* nothing */ { [] }
| parameter_list parameter_decl { $1 @ List.rev $2 }

parameter_decl:
    PARAMETER parameter_initialization_list SEMICOLON { $2 }
| error { raise (Parse_Failure("Parameter declaration error.", Parsing.symbol_start_pos
        ( )) ) }

parameter_initialization_list:
    parameter_initialization { [$1] }
| parameter_initialization_list COMMA parameter_initialization { $3 :: $1 }

parameter_initialization:
    ID ASSIGN DLIT { $1, $3, Parsing.symbol_start_pos () }
| error { raise (Parse_Failure("Parameter initialization error.", Parsing.symbol_start_pos
        ( )) ) }

decl_list:
    /* nothing */ { [] }
| decl_list decl { $1 @ List.rev $2 }

decl:
    WIRE wire_decl_with_opt_init_list SEMICOLON { $2 }
| REG reg_decl_list SEMICOLON { $2 }

wire_decl_with_opt_init_list:
    wire_decl_with_opt_init { [$1] }
| wire_decl_with_opt_init_list COMMA wire_decl_with_opt_init { $3 :: $1 }
| error { raise (Parse_Failure("Wire declaration error.", Parsing.symbol_start_pos ( )) ) }

wire_decl_with_opt_init:
    ID { [ decltype = Wire; declname = $1; declwidth = 1; init = Noexpr(Parsing.
        symbol_start_pos ( )); declpos = Parsing.symbol_start_pos () ] }
| ID LBRACKET DLIT RBRACKET { [ decltype = Wire; declname = $1; declwidth = $3; init =
        Noexpr(Parsing.symbol_start_pos ( )); declpos = Parsing.symbol_start_pos () ] }

38
ID ASSIGN expr { { decftype = Wire; declname = $1; declwidth = 1; init = $3; declpos = Parsing.symbol_start_pos () } }
ID LBRAket DLIT RBRacket ASSIGN expr { { decftype = Wire; declname = $1; declwidth = $3; init = $6; declpos = Parsing.symbol_start_pos () } }

reg_decl_list:
  reg_decl { [$1] }
  reg_decl_list COMMA reg_decl { $3 :: $1 }
| error { raise (Parse_Failure("Register declaration error.", Parsing.symbol_start_pos () )) }

reg_decl:
  ID { { decftype = Reg; declname = $1; declwidth = 1; init = Noexpr(Parsing. symbol_start_pos () ); declpos = Parsing.symbol_start_pos () } }
| ID LBRAket DLIT RBRacket { { decftype = Reg; declname = $1; declwidth = $3; init = Noexpr(Parsing. symbol_start_pos () ); declpos = Parsing.symbol_start_pos () } }
| ID ASSIGN expr { raise (Parse_Failure("Registers may not be initialized.", Parsing. symbol_start_pos () )) }
| ID LBRAket DLIT RBRacket ASSIGN expr { raise (Parse_Failure("Registers may not be initialized.", Parsing. symbol_start_pos () )) }

stmt_list:
| /* nothing */ { [] }
| stmt_list stmt { $2 :: $1 }

stmt:
| stmt: SEMICOLON { Expr($1, Parsing.symbol_start_pos () ) }
| RETURN stmt: SEMICOLON { Return($2, Parsing.symbol_start_pos () ) }
| LBRAket stmt_list RBRacket { Block(List.rev $2, Parsing.symbol_start_pos () ) }
| IF LPARENT condition_clock RPAREN stmt %prec NOELSE { If($3, $5, Nop(Parsing. symbol_start_pos () ), Parsing.symbol_start_pos () ) }
| IF LPARENT expr RPAREN stmt ELSE stmt { If(Expression($3), $5, Nop(Parsing. symbol_start_pos () ), Parsing.symbol_start_pos () ) }
| IF LPARENT expr RPAREN stmt ELSE stmt { If(Expression($3), $5, $7, Parsing. symbol_start_pos () ) }
| IF LPARENT condition_clock RPAREN stmt ELSE stmt { raise (Parse_Failure("Clock edge if statements may not have else clauses.", Parsing.symbol_start_pos () )) }
| CASE LPARENT lvalue RPAREN LBRAket case_list RBRacket { Case($3, List.rev $6, Parsing. symbol_start_pos () ) }
| FOR LPARENT ID ASSIGN expr SEMICOLON ID ASSIGN expr RPAREN stmt { if $3 <> $9 then raise (Parse_Failure("For loops must have only a single loop variable.", Parsing.symbol_start_pos () )) else For($3, $5, $7, $11, $13, Parsing.symbol_start_pos () ) }
| FOR LPARENT error RPAREN stmt { raise (Parse_Failure("Invalid for loop header.", Parsing. symbol_start_pos () )) }
| SEMICOLON { Nop(Parsing.symbol_start_pos () ) } /* empty statements */
| lvalue ASSIGN expr SEMICOLON { Assign($1, $3, Parsing.symbol_start_pos () ) }

condition_clock:
  POSEDGE { Posedge }
  NEGEDGE { Negedge }

case_list:
  case_item { [$1] }
  case_list case_item { $2 :: $1 }
| error { raise (Parse_Failure("Case statement error.", Parsing.symbol_start_pos () )) }

case_item:
  BLIT COLON stmt { $1, $3, Parsing.symbol_start_pos () }
  XLIT COLON stmt { $1, $3, Parsing.symbol_start_pos () }

lvalue:
  ID { Identifier($1) }
  ID LBRAket expr RBRacket { Subscript($1, $3) }
  ID LBRAket expr COLON expr RBRacket { Range($1, $3, $5) }
expr:
DLIT { DLiteral($1, Parsing.symbol.start_pos ()) }  
BLIT { BLiteral($1, Parsing.symbol.start_pos ()) }  
lvalue { Lvalue($1, Parsing.symbol.start_pos ()) }  
expr PLUS expr { Binop($1, Plus, $3, Parsing.symbol.start_pos ()) }  
expr MINUS expr { Binop($1, Minus, $3, Parsing.symbol.start_pos ()) }  
expr MULTIPLY expr { Binop($1, Multiply, $3, Parsing.symbol.start_pos ()) }  
expr MODULUS expr { Binop($1, Modulus, $3, Parsing.symbol.start_pos ()) }  
DLIT SIGEXT expr { Signext($1, $3, Parsing.symbol.start_pos ()) }  
expr EQ expr { Binop($1, Eq, $3, Parsing.symbol.start_pos ()) }  
expr NE expr { Binop($1, Ne, $3, Parsing.symbol.start_pos ()) }  
expr GE expr { Binop($1, Ge, $3, Parsing.symbol.start_pos ()) }  
expr GT expr { Binop($1, Gt, $3, Parsing.symbol.start_pos ()) }  
expr LE expr { Binop($1, Le, $3, Parsing.symbol.start_pos ()) }  
expr LT expr { Binop($1, Lt, $3, Parsing.symbol.start_pos ()) }  
expr AND expr { Binop($1, And, $3, Parsing.symbol.start_pos ()) }  
expr OR expr { Binop($1, Or, $3, Parsing.symbol.start_pos ()) }  
expr XOR expr { Binop($1, Xor, $3, Parsing.symbol.start_pos ()) }  
expr XNOR expr { Binop($1, Xnor, $3, Parsing.symbol.start_pos ()) }  
expr LSHIFT expr { Binop($1, Lshift, $3, Parsing.symbol.start_pos ()) }  
expr RHSIFT expr { Binop($1, Rshift, $3, Parsing.symbol.start_pos ()) }  
LPARENP expr RPAREN { $2 }  
NOT expr { Unary(Not, $2, Parsing.symbol.start_pos ()) }  
PLUS expr %prec UPLUS { Unary(Plus, $2, Parsing.symbol.start_pos ()) }  
MINUS expr %prec UMINS { Unary(Minus, $2, Parsing.symbol.start_pos ()) }  
AND lvalue %prec NOT { Reduct(And, $2, Parsing.symbol.start_pos ()) }  
OR lvalue %prec NOT { Reduct(Or, $2, Parsing.symbol.start_pos ()) }  
XOR lvalue %prec NOT { Reduct(Xor, $2, Parsing.symbol.start_pos ()) }  
XNOR lvalue %prec NOT { Reduct(Xnor, $2, Parsing.symbol.start_pos ()) }  
RESET { Reset(Parsing.symbol.start_pos ()) }  
CONCAT LPAREN concat.list RPAREN { Concat(List.rev $3, Parsing.symbol.start_pos ()) }  
| CONCAT LPAREN binding_in_list_opt SEMICOLON binding_out_list_opt RPAREN { Inst($1, List.rev $3, List.rev $5, Parsing.symbol.start_pos ()) }  
| CONCAT LPAREN concat_list RPAREN { Concat(List.rev $3, Parsing.symbol.start_pos ()) }  
concat_list:  
| concat_item { $1 }  
| concat_list COMMA concat_item { $3 :: $1 }  
| error { raise (Parse.Failure("Concatenation error.", Parsing.symbol.start_pos ()) ) }  
concat_item:  
| BLIT { ConcatBLiteral(1, $1) }  
| lvalue { ConcatLvalue(1, $1) }  
| DLIT LBRACE BLIT RBRACE { ConcatBLiteral($1, $3) }  
| DLIT LBRACE lvalue RBRACE { ConcatLvalue($1, $3) }  
binding_in_list:  
| binding_in { $1 }  
| binding_in_list COMMA binding_in { $3 :: $1 }  
| error { raise (Parse.Failure("Port binding error.", Parsing.symbol.start_pos ()) ) }  
binding_in_list_opt:  
/*nothing*/ { [] }  
| binding_in_list { $1 }  
binding_out_list:  
| binding_out { $1 }  
| binding_out_list COMMA binding_out { $3 :: $1 }  
| error { raise (Parse.Failure("Port binding error.", Parsing.symbol.start_pos ()) ) }  
binding_out_list_opt:  
/*nothing*/ { [] }  
| binding_out_list { $1 }  
ID ASSIGN expr { $1, $3 }  
| CLOCK ASSIGN expr { "clock", $3 }
8.6 Scanner.mll

```plaintext
{ open Parser

let incr_linenum lexbuf =
let pos = lexbuf.Lexing.lex_curr_p in
lexbuf.Lexing.lex_curr_p <- { pos with
  Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
  Lexing.pos_cnum = lexbuf.Lexing.pos_cnum;
}

rule token = parse
[ ' ' 't' 'r' ] { token lexbuf } (*Whitespace*)
[ 'n' { incr_linenum lexbuf; token lexbuf }
  { comment lexbuf } (*Comments*)
[ '//' { comment2 lexbuf } (*Punctuation*)
[ '(' { LPAREN }
[ ')' { RPAREN }
[ '{' { LBRACE }
[ '}' { RBRACE }
[ ';' { SEMICOLON }
[ ',' { COMMA }
[ ':' { COLON }
[ '+'] { PLUS } (*Operators*)
[ '-'] { MINUS }
[ '*'] { MULTIPLY }
[ '/'] { MODULUS }
[ '<<'] { LSHIFT }
[ '>>'] { RSHIFT }
[ '^'] { SIGEXT }
[ '~&'] { NAND }
[ '~|'] { NOR }
[ '~^'] { XNOR }
[ '~'] { NOT }
[ '&'] { AND }
[ '|'] { OR }
[ '<<'] { ASSIGN }
[ '=='] { CASE } (*Keywords*)
[ 'clock' { CLOCK }
[ 'concat' { CONCAT }
[ 'else' { ELSE }
```
9 Example files

9.1 Gcd.vs

```vhdl
module gcd (input num0[8], num1[8], start; output greatest[8], success) {
  register found;
  success = found;
  if (start) {
    if (num0<num1)
      greatest = num0;
    else
      greatest = num1;
  }
  if (posedge) {
    if (~found) {
      if (num0%greatest==0 & num1%greatest==0)
        found = 1;
      else
        greatest = greatest -1;
    }
  }
}
```

examples/gcd.vs

9.2 Gcd.v

```vhdl
module gcd (.clock, .reset, .num0, .num1, .start, .greatest, .success);
input .clock;
input .reset;
input [7:0] .num0;
input [7:0] .num1;
input .start;
```
```verilog
output [7:0] _greatest;
output _success;
reg [7:0] _reg_greatest;
reg _found;
assign _success = _found;
assign _greatest [7:0] = _reg_greatest [7:0];
always @(*) begin
  if (_start)
    begin
      if ((_num0[7:0] < _num1 [7:0]))
        begin
            _reg_greatest [7:0] = _num0 [7:0];
        end
      else
        begin
            _reg_greatest [7:0] = _num1 [7:0];
        end
    end
  else
    begin
    end
end
endmodule

always @ (posedge _clock) begin
  if ((!_found))
    begin
      if (((_num0[7:0] % _greatest [7:0]) ==0) & ((_num1[7:0] % _greatest [7:0]) ==0))
        begin
          _found = 1;
        end
    end
else
  begin
      _reg_greatest [7:0] = (_greatest [7:0] -1);
  end
end
endmodule
```

9.3 Stim file for gcd

```verilog
module stim();
  reg clk;
  reg reset;
  reg [7:0]num1;
  reg [7:0]num0;
  reg [3:0] count;
  reg start;
  wire [7:0]greatest;
  wire success;
```

// This is a very rudimentary stim file for the gcd. It will input two numbers, 36 and 24,
// and find the greatest common denominator

// instance the dut
gcd dut(
   .clock(clk),
   .reset(reset),
   .num0(num0),
   .num1(num1),
   .start(start),
   .greatest(greatest),
   .success(success)
);

initial begin
   reset = 1; // first positive edge, i.e. first cycle. Zero out the module
   #1 clk = 0;
   #1 clk = 1;
   #1 clk = 0;
   reset = 0;
   start = 1;
   num0 = 36;
   num1 = 24;
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   #1 clk = 1;
   $display("Found:%d Current greatest %d\n",success,greatest);
   #1 clk = 0;
   ...
9.4 HelloWord.vs

```vhdl
module helloWorld (input enable; output letter [8]) {
  register count [4];
  wire c [4];
  c = count;
  if (enable) {
    case (c) {
      0001b: letter = 01001000b;
      0010b: letter = 01100101b;
      0011b: letter = 01101100b;
      0100b: letter = 01101100b;
      0101b: letter = 01101111b;
      0110b: letter = 00100000b;
      #1 clk = 0;
      #1 clk = 1;
    }
  }
  #1 $finish;
  end
  endmodule
```
14 0111b: letter=01010111b;
15 1000b: letter=01101111b;
16 1001b: letter=01110010b;
17 1010b: letter=01101100b;
18 1011b: letter=01100100b;
19 1100b: letter=00100001b;
20 //default: letter=00000000b;
21 }
22 }
23 }
24 } if (posedge) {
25 count = count+1;
26 }
27 } examples/helloworld.vs

9.5 Helloworld.v

1 module .helloWorld(.clock, .reset, .enable, .letter);
2 input .clock;
3 input .reset;
4 input .enable;
5 output [7:0] .letter;
6 reg [7:0] _reg_letter;
7 wire [3:0] _c;
8 reg [3:0] _count;
9 assign _c[3:0] = _count [3:0];
10 assign _letter [7:0] = _reg_letter [7:0];
11 always @(*) begin
12 if (_enable) begin
13 4'b0001:
14 begin
15 _reg_letter [7:0]=72;
16 end
17 4'b0010:
18 begin
19 _reg_letter [7:0]=101;
20 end
21 4'b0011:
22 begin
23 _reg_letter [7:0]=108;
24 end
25 4'b0100:
26 begin
27 _reg_letter [7:0]=108;
28 end
29 4'b0101:
30 begin
31 _reg_letter [7:0]=111;
32 end
33 4'b0110:
34 begin
35 _reg_letter [7:0]=32;
36 end
37 4'b0111:
38 begin
39 _reg_letter [7:0]=87;
40 end
41 4'b1000:
42 begin
43 _reg_letter [7:0]=111;
44 end
45 4'b1001:
begin
.reg_letter [7:0] = 114;
end
4'b1010:
begin
.reg_letter [7:0] = 108;
end
4'b1011:
begin
.reg_letter [7:0] = 100;
end
4'b1100:
begin
.reg_letter [7:0] = 33;
endcase
else
begin
if (.reset) begin
.reg_letter = 0; .count = 0; end
always @(posedge .clock) begin
.count[3:0] = ( .count[3:0] + 1 );
end
always @(negedge .clock) begin
endendmodule