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

3.7.1 Latches . . . 17
3.7.2 Flipflops . . . 17

4 Project Plan **17**
4.1 Team Responsibilities . . . 17
4.1.1 Round-Robin Dictatorship . . . 17
4.2 Project Timeline . . . 17
4.3 Software Development Environment . . . 17
4.4 Project Log . . . 18

5 Architectural Design **18**
5.1 Architecture . . . 18
5.2 Error Recovery . . . 19

6 Testing Plan **19**

7 Lessons Learned **19**

8 Codebase **21**
8.1 Ast.ml . . . 21
8.2 Asttoimst.ml . . . 22
8.3 Imst.ml . . . 33
8.4 Imsttocode.ml . . . 34
8.5 Parser.mly . . . 36
8.6 Scanner.mll . . . 41

9 Example files **42**
9.1 Gcd.vs . . . 42
9.2 Gcd.v . . . 42
9.3 Stim file for gcd . . . 43
9.4 Helloworld.vs . . . 45
9.5 Helloworld.v . . . 46

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:

```
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:

```
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:

```
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:

```
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.

```
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.

```
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.

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

Bitwise Operations:

Operator	Example	Result
NOT	`~a`	`10b`
AND	`a&b`	`01b`
OR	`a|b`	`11b`

Full range of bitwise operators (NAND and NOR not supported):

<table>
<tr><th>Operator</th><th>Example</th><th>Equivalency</th><th>Results</th></tr>
<tr><td>XOR</td><td>a^b</td><td>(~a & b) | (a & ~b)</td><td></td></tr>
<tr><td>XNOR</td><td>a ~^b</td><td>(~a | b) & (a | ~b)</td><td></td></tr>
</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>
<tr><th>Operator</th><th>Example</th><th>Equivalency</th><th>Result</th></tr>
<tr><td>AND</td><td>&a</td><td>a[0] & a[1] & a[2]</td><td>0</td></tr>
<tr><td>NAND</td><td>~&a</td><td>~(a[0] & a[1] & a[2])</td><td>1</td></tr>
<tr><td>OR</td><td>|a</td><td>a[0] | a[1] | a[2]</td><td>1</td></tr>
<tr><td>NOR</td><td>~|a</td><td>~(a[0] | a[1] | a[2])</td><td>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>~^a</td><td>~(a[0] ^a[1] ^a[2])</td><td>0</td></tr>
</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];
```

Operator	Example	Notes	Result
Plus	`e3=e0+e1`	Overflow bits are discarded.	`e3=010b`
Minus	`e3=e0-e1`		`e3 = 100b`
Multiplication	`e0*e1`	Returns a bus that is n+m-1 long	`10101b`
Modulus	`e0%e1`	Returns a bus that is n long where n is the size of the second operand	`011b`

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
```

Operator	Example	Note	Result
Left-shift	`e0<<2; e1<<2`	Left shift will always fill with zeros	`1100b //-4d; 1100b`
Right-shift	`e0>>2; e1>>2`	Right shift will always fill with the most significant bit to preserve sign	`0001b //1d; 1111b //-1d`

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
```

Operator	Example	Note	Result
Sign Extension	`8'e0; 8'e1`	Left hand side is always a decimal number.	`0000011b; 11111111b`

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
```

Operator	Example	Result
Less than	e0 < e1	0
Less than or equal to	e0 <= e1	1
Greater than	e0 > e2	1
Greater than or equal to	e >= e2	1
Equal to	e0 == e1	1
Not equal to	e0 != e1	0

3.3.9 Precedence and associativity

<table>
<tr><th>Operators, in order of decreasing precedence</th><th>Associativity</th></tr>
<tr><td>~ + - & ~& ^ ~^ | ~| (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><< >></td><td>left to right</td></tr>
<tr><td>< <= > >=</td><td>left to right</td></tr>
<tr><td>== !=</td><td>left to right</td></tr>
<tr><td>& (binary)</td><td>left to right</td></tr>
<tr><td>^ ~^ (binary)</td><td>left to right</td></tr>
<tr><td>| (binary)</td><td>left to right</td></tr>
</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.

```
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) {
```

```
        r = in;
        out = r;
    }
}
```

This is a simple four-bit latch. On every positive clock edge, if the `enable` bit is set to 1, the output will be set to the input. If the `enable` bit is set to 0, the output will remain unchanged. However, if the module's `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:

```
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;
    }
}
```

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

Notice that though `reset` is not listed as an input of module `m2`, it is implicit; it can be referred to by keyword `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 `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 `return` keyword anywhere in a block. A single block may not have multiple `return` keywords.

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

These can then be used as follows: `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 doesnt 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 = 111b; out2= 110b;}
```

```
    x00b :  out = 000b;
}
```

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`:

```
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:

```
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:

```
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

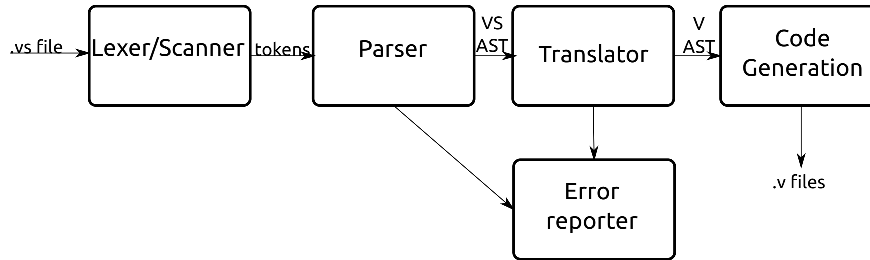
Our goals timewise were as follows:

Week of	Work Contributed
September 27, 2010	Submit project proposal.
October 18, 2010	Finish first writing of Language Reference Manual (LRM).
November 1, 2010	Edit and submit LRM.
November 8, 2010	Complete parser and establish grammar.
November 29, 2010	Finish code generation and error recovery.
December 6, 2010	Write and test cases.
December 13, 2010	Write report and submit final coding.

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

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

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 necessart 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:

```
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
  | Signext of int * 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
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 | Negedge | 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 * 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;
```

```
63   libmod_width : int;
64   modpos : Lexing.position;
65 }
66
67 type program = mod_decl list
68
69 exception Parse_Failure of string * Lexing.position
```

ast.ml

8.2 Asttoimst.ml

```
1 (* convert an AST to an IMST, with checking *)
2 open Ast
3 open Imst
4
5 module StringMap = Map.Make(String)
6
7 (* Environment information *)
8 type enviro = {
9   local_map     : declaration list StringMap.t;
10   param_map     : parameter list StringMap.t;
11   arg_map       : (id_with_width list * id_with_width list) StringMap.t;
12   return_map    : int StringMap.t
13 }
14
15 let string_map_args map mods =
16   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
17
18 let string_map_params map mods =
19   List.fold_left (fun m mod1 -> StringMap.add mod1.modname mod1.parameters m) map mods
20
21 let string_map_locals map mods =
22   List.fold_left (fun m mod1 -> StringMap.add mod1.modname mod1.declarations m) map mods
23
24 let string_map_returns map mods =
25   List.fold_left (fun m mod1 -> StringMap.add mod1.modname mod1.returnwidth m) map mods
26
27 let get_param_value (_, x, _) = x
28
29 let rec get_param_tuple name lst =
30   match lst with
31       [] -> raise Not_found
32     | (s, i, p)::tl -> if s = name then (s, i, p) else get_param_tuple name tl
33
34 let get_param mod_name param_name env =
35   get_param_value (get_param_tuple param_name (StringMap.find mod_name env.param_map))
36
37 let rec get_arg_tuple name lst =
38   match lst with
39       [] -> raise Not_found
40     | (s, i, _)::tl -> if s = name then (s,i) else get_arg_tuple name tl
41
42 let rec invert_binary_actual n x =
43   if n < 0 then x else
44   (x.[n] <- (if x.[n] = '1' then '0' else '1'); invert_binary_actual (n-1) x)
45 and invert_binary x = invert_binary_actual (String.length x - 1) (String.copy x)
46
47 let rec add_one_actual n x =
48   if n < 0 then x (*discard overflow bit*)
49   else
50   if x.[n] = '1' then (x.[n] <- '0'; add_one_actual (n-1) x) else (x.[n] <- '1'; x)
51 and add_one x = add_one_actual (String.length x -1) (String.copy x)
52
```

```
53 let rec eval_expr mod_name env = function
54     DLiteral(x, _) -> Int64.of_int x
55   | BLiteral(x, pos) -> (try
56               if x.[0] = '0' then Int64.of_string ("0b" ^ x)
57               else Int64.neg (Int64.of_string ("0b"^(add_one (invert_binary x))))
58               with Failure(_) -> raise (Parse_Failure("Binary literals may not exceed 64
                   bits", pos)))
59   | Lvalue(x, pos) -> (match x with
60         Identifier(id) -> (try Int64.of_int(get_param mod_name id env)
61                             with Not_found -> raise (Parse_Failure("Expression cannot be
                                 evaluated at compile time.", pos)))
62       | _ -> raise (Parse_Failure("Expression cannot be evaluated at compile time.", pos)))
63   | Binop(e1, op, e2, pos) -> (match op with
64       Plus -> Int64.add (eval_expr mod_name env e1) (eval_expr mod_name env e2)
65     | Minus -> Int64.sub (eval_expr mod_name env e1) (eval_expr mod_name env e2)
66     | Multiply -> Int64.mul (eval_expr mod_name env e1) (eval_expr mod_name env e2)
67     | Modulus -> Int64.rem (eval_expr mod_name env e1) (eval_expr mod_name env e2)
68     | Eq -> if (Int64.compare (eval_expr mod_name env e1) (eval_expr mod_name env e2)) = 0
         then Int64.one else Int64.zero
69     | Ne -> if (Int64.compare (eval_expr mod_name env e1) (eval_expr mod_name env e2)) <> 0
         then Int64.one else Int64.zero
70     | Ge -> if (Int64.compare (eval_expr mod_name env e1) (eval_expr mod_name env e2)) >= 0
         then Int64.one else Int64.zero
71     | Gt -> if (Int64.compare (eval_expr mod_name env e1) (eval_expr mod_name env e2)) > 0
         then Int64.one else Int64.zero
72     | Le -> if (Int64.compare (eval_expr mod_name env e1) (eval_expr mod_name env e2)) <= 0
         then Int64.one else Int64.zero
73     | Lt -> if (Int64.compare (eval_expr mod_name env e1) (eval_expr mod_name env e2)) < 0
         then Int64.one else Int64.zero
74     | And -> Int64.logand (eval_expr mod_name env e1) (eval_expr mod_name env e2)
75     | Or -> Int64.logor (eval_expr mod_name env e1) (eval_expr mod_name env e2)
76     | Xor -> Int64.logxor (eval_expr mod_name env e1) (eval_expr mod_name env e2)
77     | Nand -> Int64.lognot (Int64.logand (eval_expr mod_name env e1) (eval_expr mod_name env
          e2))
78     | Nor -> Int64.lognot (Int64.logor (eval_expr mod_name env e1) (eval_expr mod_name env
         e2))
79     | Xnor -> Int64.lognot (Int64.logxor (eval_expr mod_name env e1) (eval_expr mod_name env
          e2))
80     | Lshift -> Int64.shift_left (eval_expr mod_name env e1) (Int64.to_int (eval_expr
         mod_name env e2))
81     | Rshift -> Int64.shift_right (eval_expr mod_name env e1) (Int64.to_int (eval_expr
         mod_name env e2)) (*arithmetic shift - keep sign*)
82     | Not -> raise (Parse_Failure("Internal compiler error 003: contact manufacturer for
         assistance.", pos))
83   )
84   | Signext(_, _, pos) -> raise (Parse_Failure("Sign extension cannot be used in the current
        context.", pos))
85   | Reduct(_, _, pos) -> raise (Parse_Failure("Expression cannot be evaluated at compile
       time.", pos))
86   | Unary (op, exp, pos) -> (match op with
87     | Not -> Int64.lognot (eval_expr mod_name env exp)
88     | Plus -> (eval_expr mod_name env exp)
89     | Minus -> Int64.neg (eval_expr mod_name env exp)
90     | _ -> raise (Parse_Failure("Internal compiler error 002: contact manufacturer for
         assistance.", pos))
91     )
92   | Concat(_, pos) -> raise (Parse_Failure("Concatenation cannot be used in the current
       context.", pos))
93   | Inst(_,_,_,pos) -> raise (Parse_Failure("Expression cannot be evaluated at compile time
       .", pos))
94   | Reset(pos) -> raise (Parse_Failure("Expression cannot be evaluated at compile time.",
       pos))
95   | Noexpr(pos) -> raise (Parse_Failure("Internal compiler error 001: contact manufacturer
       for assistance.", pos))
96
97
98
99 let get_arg mod_name arg_name env =
```

```
100   let tuples = StringMap.find mod_name env.arg_map in
101   get_arg_tuple arg_name((fst tuples) @ (snd tuples) )
102
103
104 let check_mod_info lst1 mods =
105   List.fold_left (fun lst mod1 -> 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
106
107 let check_unique_ids_in_module env mod1 =
108   let mod_name = mod1.modname in
109   let inputslist = List.fold_left (fun lst2 (name, width, pos) -> (* each input *)
110       if List.mem name lst2
111       then raise (Parse_Failure ("Duplicate identifier.", pos))
112       else if width < 1
113       then raise (Parse_Failure ("Invalid width.", pos))
114       else name :: lst2) [] (fst (StringMap.find mod_name env.arg_map))
115     in let argslist = List.fold_left (fun lst2 (name, width, pos) -> (* each output *)
116       if List.mem name lst2
117       then raise (Parse_Failure ("Duplicate identifier.", pos))
118       else if width < 1
119       then raise (Parse_Failure ("Invalid width.", pos))
120       else name :: lst2) inputslist (snd (StringMap.find mod_name env.arg_map))
121     in let argsandparamslist = List.fold_left (fun lst2 (name, _, pos) ->
122       if List.mem name lst2
123       then raise (Parse_Failure ("Duplicate identifier.", pos))
124       else name :: lst2) argslist (StringMap.find mod_name env.param_map)
125     in
126     List.fold_left (fun lst2 decl -> (* each decl in each mod *)
127       if List.mem decl.declname lst2
128       then raise (Parse_Failure ("Duplicate identifier.", decl.declpos))
129       else if decl.declwidth < 1
130       then raise (Parse_Failure ("Invalid width.", decl.declpos))
131       else decl.declname :: lst2) argsandparamslist (StringMap.find mod_name env.local_map)
132
133
134 let rec get_min_bit_width x =
135   if Int64.compare Int64.zero x = 0 || Int64.compare Int64.one x = 0 || Int64.compare Int64.
        minus_one x = 0 then 1
136   else 1 + get_min_bit_width (Int64.div x (Int64.of_int 2))
137
138
139 let rec get_arg_tuple name lst =
140   match lst with
141       [] -> raise Not_found
142     | (s, i, _)::tl -> if s = name then (s,i) else get_arg_tuple name tl
143
144 let get_arg mod_name arg_name env =
145   let tuples = StringMap.find mod_name env.arg_map in
146   get_arg_tuple arg_name (fst tuples @ snd tuples)
147
148 let get_input mod_name input_name env =
149   let tuples = StringMap.find mod_name env.arg_map in
150   get_arg_tuple input_name (fst tuples)
151
152 let get_output mod_name output_name env =
153   let tuples = StringMap.find mod_name env.arg_map in
154   get_arg_tuple output_name (snd tuples)
155
156 let rec get_local_tuple name lst =
157   match lst with
158       [] -> raise Not_found
159     | hd::tl -> if hd.declname = name then (hd.declname, hd.declwidth) else get_local_tuple
          name tl
160
161 let rec get_local_decl name lst =
162   match lst with
```

```
163        [] -> raise Not_found
164      | hd::tl -> if hd.declname = name then hd else get_local_decl name tl
165
166 let get_local_all mod_name local_name env =
167   get_local_decl local_name (StringMap.find mod_name env.local_map)
168
169 let get_local mod_name local_name env =
170   get_local_tuple local_name (StringMap.find mod_name env.local_map)
171
172 let get_lvalue_name = function
173   Identifier(n) -> n | Subscript(n, _) -> n | Range(n, _, _) -> n
174
175 let change_im_lvalue_name newname = function
176   ImSubscript(_, s) -> ImSubscript(newname, s) | ImRange(_, up, lo) -> ImRange(newname, up,
        lo)
177
178 let check_valid_lvalue environ mod_name lvalue_id pos =
179   (* arg map, local map *)
180   try get_arg mod_name lvalue_id environ
181   with Not_found -> try get_local mod_name lvalue_id environ with Not_found -> raise (
        Parse_Failure("Undefined identifier.", pos))
182
183 let check_assignment_lvalue environ mod_name lvalue_id pos =
184   (* arg map, local map *)
185   try get_output mod_name lvalue_id environ
186   with Not_found -> try get_local mod_name lvalue_id environ with Not_found -> raise (
        Parse_Failure("Undefined identifier.", pos))
187
188 let to_im_lvalue environ immod lval pos = match lval with
189    Identifier(i) -> ImRange(fst (check_valid_lvalue environ immod.im_modname i pos), snd (
         check_valid_lvalue environ immod.im_modname i pos) - 1, 0)
190  | Subscript(s, expr) ->
191       let (id, width) = check_valid_lvalue environ immod.im_modname s pos in
192       let subscr = Int64.to_int (eval_expr immod.im_modname environ expr) in
193       if subscr < 0 || subscr >= width then raise (Parse_Failure("Bus index out of bounds.",
            pos)) else ImSubscript(id, subscr)
194  | Range(r, expr1, expr2) ->
195       let (id, width) = check_valid_lvalue environ immod.im_modname r pos in
196       let subscr1 = Int64.to_int (eval_expr immod.im_modname environ expr1) in
197       let subscr2 = Int64.to_int (eval_expr immod.im_modname environ expr2) in
198       if subscr1 < 0 || subscr2 < 0|| subscr1 >= width || subscr2 >= width then raise (
            Parse_Failure("Bus index out of bounds.", pos))
199       else if subscr1 < subscr2 then raise (Parse_Failure("Bus ranges must be specified from
            most significant to least significant.", pos))
200       else if subscr1 = subscr2 && width != 1 then ImSubscript(id, subscr1)
201       else ImRange(id, subscr1, subscr2)
202
203 let get_lvalue_length environ immod lvalue pos = match (to_im_lvalue environ immod lvalue
      pos) with
204        ImSubscript(_, _) -> 1
205      | ImRange(_, upper, lower) -> (upper - lower + 1)
206
207 let rec check_im_mod_local_actual name = function
208     [] -> false
209   | (_, declname, _) :: tl -> if name = declname then true else check_im_mod_local_actual
        name tl
210 and check_im_mod_local immod name = check_im_mod_local_actual name immod.im_declarations
211
212 let get_lvalue_bit_range environ immod lvalue pos = match (to_im_lvalue environ immod lvalue
       pos) with
213        ImSubscript(_, s) -> (s, s)
214      | ImRange(_, upper, lower) -> (upper, lower)
215
216 let to_im_op = function
217      Plus -> ImPlus  | Minus -> ImMinus  | Multiply -> ImMultiply | Gt -> ImGt
218    | Modulus -> ImModulus  | Eq -> ImEq  | Ne -> ImNe | Ge -> ImGe  | Le -> ImLe
219    | Lt -> ImLt  | And -> ImAnd | Or -> ImOr | Xor -> ImXor | Nand -> ImNand
220    | Nor -> ImNor | Xnor -> ImXnor | Lshift -> ImLshift | Rshift -> ImRshift | Not -> ImNot
```

```
221
222
223 let rec get_max_bit_width_expr environ immod expr = match expr with
224     DLiteral(d, _) -> 64
225   | BLiteral(b, _) -> String.length b
226   | Lvalue(l, pos) -> get_lvalue_length environ immod l pos
227   | Binop(e1, op, e2, pos) -> (match op with
228       Plus -> max (get_max_bit_width_expr environ immod  e1) (get_max_bit_width_expr environ
            immod e2)
229     | Minus -> max (get_max_bit_width_expr environ immod  e1) (get_max_bit_width_expr
          environ immod e2)
230     | Multiply -> (get_max_bit_width_expr environ immod  e1) + (get_max_bit_width_expr
          environ immod e2) - 1
231     | Modulus -> get_max_bit_width_expr environ immod e2
232     | Eq -> 1 | Ne -> 1 | Ge -> 1 | Gt -> 1 | Le -> 1 | Lt -> 1
233     | And -> max (get_max_bit_width_expr environ immod e1) (get_max_bit_width_expr environ
          immod e2)
234     | Or -> max (get_max_bit_width_expr environ immod e1) (get_max_bit_width_expr environ
          immod e2)
235     | Xor -> max (get_max_bit_width_expr environ immod e1) (get_max_bit_width_expr environ
          immod e2)
236     | Xnor -> max (get_max_bit_width_expr environ immod e1) (get_max_bit_width_expr environ
          immod e2)
237     | Lshift -> get_max_bit_width_expr environ immod e1 | Rshift -> get_max_bit_width_expr
          environ immod e1
238   | _ -> raise (Parse_Failure("Internal compiler error 003: contact manufacturer for
        assistance.", pos)))
239   | Signext(bits, expr, _) -> bits
240   | Reduct(op, lvalue, _) -> 1
241   | Unary(_, expr, _) -> get_max_bit_width_expr environ immod expr
242   | Concat(lst, pos) -> List.fold_left (fun orig x -> (match x with
243         ConcatBLiteral(time, lit) -> orig + time * (String.length lit)
244       | ConcatLvalue(time, lvalue) -> orig + time * (get_lvalue_length environ immod lvalue
            pos))) 0 lst
245   | Inst(str, bindlst1, bindlst2, pos) -> (try StringMap.find str environ.return_map
246     with Not_found -> raise (Parse_Failure("Undefined module name.", pos)))
247   | Reset(_) -> 1
248   | Noexpr(_) -> 0
249
250 let rec get_min_bit_width_expr environ immod expr = match expr with
251     DLiteral(d, _) -> get_min_bit_width (Int64.of_int d)
252   | BLiteral(b, _) -> String.length b
253   | Lvalue(l, pos) -> get_lvalue_length environ immod l pos
254   | Binop(e1, op, e2, pos) -> (match op with
255       Plus -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ
           immod e2)
256     | Minus -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ
           immod e2)
257     | Multiply -> (get_min_bit_width_expr environ immod e1) + (get_min_bit_width_expr
          environ immod e2) - 1
258     | Modulus -> get_min_bit_width_expr environ immod e2
259     | Eq -> 1 | Ne -> 1 | Ge -> 1 | Gt -> 1 | Le -> 1 | Lt -> 1
260     | And -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ
          immod e2)
261     | Or -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ
          immod e2)
262     | Xor -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ
          immod e2)
263     | Xnor -> max (get_min_bit_width_expr environ immod e1) (get_min_bit_width_expr environ
          immod e2)
264     | Lshift -> get_min_bit_width_expr environ immod e1 | Rshift -> get_min_bit_width_expr
          environ immod e1
265   | _ -> raise (Parse_Failure("Internal compiler error 003: contact manufacturer for
        assistance.", pos)))
266   | Signext(bits, expr, _) -> bits
267   | Reduct(op, lvalue, _) -> 1
268   | Unary(_, expr, _) -> get_min_bit_width_expr environ immod expr
269   | Concat(lst, pos) -> List.fold_left (fun orig x -> (match x with
```

```
270        ConcatBLiteral(time, lit) -> orig + time * (String.length lit)
271      | ConcatLvalue(time, lvalue) -> orig + time * (get_lvalue_length environ immod lvalue
           pos))) 0 lst
272   | Inst(str, bindlst1, bindlst2, pos) -> (try StringMap.find str environ.return_map
273     with Not_found -> raise (Parse_Failure("Undefined module name.", pos)))
274   | Reset(_) -> 1
275   | Noexpr(_) -> 0
276
277 (* translate_expr: environment -> im_moddecl -> expr -> bool -> im_moddecl * im_expr * int
      *)
278 (* Translate an AST expression into an IM expression *)
279 let rec translate_expr environ immod expr count in_conditional = match expr with
280     DLiteral(d, _) -> (immod, ImLiteral(Int64.of_int d, get_min_bit_width (Int64.of_int d)),
          count)
281   | BLiteral(b, pos) -> (immod, ImLiteral((try
282               if b.[0] = '0' then Int64.of_string ("0b" ^ b)
283               else Int64.neg (Int64.of_string ("0b"^(add_one (invert_binary b))))
284               with Failure(_) -> raise (Parse_Failure("Binary literals may not exceed 64
                    bits", pos))), String.length b), count)
285   | Lvalue(l, pos) -> (immod, ImLvalue(to_im_lvalue environ immod l pos), count)
286   | Binop(e1, op, e2, _) -> let (immod1, imexp1, count1) = translate_expr environ immod e1
        count in_conditional in
287                             let (immod2, imexp2, count2) = translate_expr environ immod1 e2
                                 count1 in_conditional in
288                             (immod2, ImBinop(imexp1, to_im_op op, imexp2), count2)
289   | Signext(bits, exp, pos) -> (* generate a temporary wire to store the value of expr. Then
        , use the concatentation syntax. *)
290    let width = get_min_bit_width_expr environ immod expr in
291    if width > bits then raise (Parse_Failure("Cannot sign extend something into fewer bits
         than the original.", pos)) else
292    let (immod1, imexpr1, count1) = translate_expr environ immod exp count in_conditional in
293    ({ immod1 with im_declarations = (ImWire, "_im_" ^ (string_of_int count1), width) ::
         immod1.im_declarations;
294                  im_assignments = (ImRange("_im_" ^ (string_of_int count1), width - 1, 0),
                       imexpr1) :: immod1.im_assignments},
295     ImConcat([ImConcatLvalue(bits - width, ImSubscript("_im_" ^ (string_of_int count1),
          width - 1));
296             ImConcatLvalue(1, ImRange("_im_" ^ (string_of_int count1), width - 1, 0))]),
                  count + 1)
297   | Reduct(op, lvalue, pos) -> (immod, ImReduct(to_im_op op, to_im_lvalue environ immod
        lvalue pos), count)
298   | Unary(op, expr, _) -> let (immod1, imexp1, count1) = translate_expr environ immod expr
        count in_conditional in (immod1, ImUnary(to_im_op op, imexp1), count1)
299   | Concat(lst, pos) -> let (immod1, concatlst1, count1) = List.fold_left (fun (immod2, lst2
        , count2) x -> match x with
300     ConcatBLiteral(time, lit) -> if time <= 0 then raise (Parse_Failure("Replication must be
           at least one time.", pos)) else
301     let immod3 = { immod2 with im_declarations = (ImWire, "_im_" ^ (string_of_int count2),
          String.length lit) :: immod2.im_declarations; im_assignments = (ImRange("_im_" ^ (
          string_of_int count2), String.length lit - 1, 0), ImLiteral((try
302             if lit.[0] = '0' then Int64.of_string ("0b" ^ lit)
303             else Int64.neg (Int64.of_string ("0b"^(add_one (invert_binary lit))))
304             with Failure(_) -> raise (Parse_Failure("Binary literals may not exceed 64
                    bits", pos))), String.length lit)) :: immod2.im_assignments } in
305          (immod3, ImConcatLvalue(time, ImRange("_im_" ^ (string_of_int count2), String.
              length lit - 1, 0)) :: lst2, count2 + 1)
306   | ConcatLvalue(time, lvalue) -> if time <= 0 then raise (Parse_Failure("Replication must
        be at least one time.", pos)) else
307      (immod2, ImConcatLvalue(time, to_im_lvalue environ immod lvalue pos) :: lst2, count2))
           (immod, [], count) lst in
308      (immod1, ImConcat(List.rev concatlst1), count1)
309   | Reset(_) -> (immod, ImLvalue(ImRange("reset", 0, 0)), count)
310   | Noexpr(_) -> (immod, ImNoexpr, count)
311   | Inst(othermod, bindlst1, bindlst2, pos) -> if in_conditional then raise (Parse_Failure("
        Modules may not be instantiated inside conditional blocks.", pos)) else
312     (
313     (* Check bindings *)
```

```
314      let (immod1, count1, converted_bindings_in) = convert_bindings_in environ count immod
             othermod bindlst1 pos in_conditional in
315      let (immod2, count2, converted_bindings_out) = convert_bindings_out environ count immod1
              othermod bindlst2 pos in
316      try
317        let returnwidth = StringMap.find othermod environ.return_map in
318        if returnwidth = 0 then
319          (* No return value - just do instantiation *)
320          ( { immod2 with im_instantiations = (othermod, converted_bindings_in,
                 converted_bindings_out) :: immod2.im_instantiations; }, ImNoexpr, count2)
321        else
322          (* return value - Do the following:*)
323          (* Generate a bus for this purpose with the name _im_ followed by count, then
                 increment count.. *)
324          (* Add binding between "return" port and the new bus. *)
325          let new_bus_name = "_imexp_" ^ (string_of_int count2) in
326          let new_bindings_out = ("return", ImLvalue(ImRange(new_bus_name, returnwidth - 1, 0)
                 )) :: converted_bindings_out in
327          ( { immod2 with im_instantiations = (othermod, converted_bindings_in,
                 new_bindings_out) :: immod2.im_instantiations;
328                          im_declarations = (ImWire, new_bus_name, returnwidth) :: immod2.
                                 im_declarations; },
329                          ImLvalue(ImRange(new_bus_name, returnwidth-1, 0)), count2 + 1)
330      with Not_found -> raise (Parse_Failure("Undefined module name.", pos))
331      )
332  and add_param_pos startpos endpos paramname lst pos = if startpos < endpos then lst
333            else let newparamplace = paramname ^ (string_of_int startpos) in
334                 if List.mem newparamplace lst then raise (Parse_Failure("Duplicate binding.",
                        pos))
335                 else add_param_pos (startpos - 1) endpos paramname (newparamplace::lst) pos
336
337  (* convert_bindings_in: env -> int -> im_moddecl -> string -> binding_in list -> im_moddecl
       * int * im_assignment list *)
338  (* Check: no duplicate bindings (but partial binding is OK, and do not have to bind anything
       ).*)
339  (* Expr to be assigned to will be translated into ImExpr*)
340  (* The expr bound to can be anything. Do not check for uninitialized wires. *)
341  and convert_bindings_in environ count immod othermod bindlst pos in_conditional =
342    let (immod1, count1, list1, _) = (List.fold_left (fun (mod1, cnt1, bnd1, lst1) (name, exp1
         ) -> (let (_, width) = get_input othermod name environ in
343                              let lst2 = add_param_pos (width - 1) 0 name lst1 pos in
344                              if width < get_min_bit_width_expr environ immod exp1 then raise (
                                     Parse_Failure("Binding width mismatch.", pos))
345                              else if width > get_max_bit_width_expr environ immod exp1 then
                                     raise (Parse_Failure("Binding width mismatch.", pos))
346                              else let (mod2, exp2, cnt2) = translate_expr environ mod1 exp1
                                     cnt1 in_conditional in
347                              (mod2, cnt2, (name, exp2) :: bnd1, lst2)))
348      (immod, count, [], []) bindlst) in (immod1, count1, list1)
349
350  (* convert_bindings_out: env -> im_moddecl -> string -> binding_out list -> im_moddecl * int
        * im_assignment list *)
351  (* Check: no duplicate bindings to ports (but partial binding is OK, and do not have to bind
        anything). *)
352  (* Note that duplicate assignments to wires will result in undefined behavior. *)
353  (* Check that the target of the assignment is a wire - "binding" a reg to an output does not
        make any sense. *)
354  and convert_bindings_out environ count immod othermod bindlst pos =
355    let (immod1, count1, list1 , _) = (List.fold_left (fun (mod1, cnt1, bnd1, lst1) (name,
         lval2) ->
356      try (
357      (try
358        let result = get_local_all immod.im_modname (get_lvalue_name lval2) environ in
359        if result.decltype = Reg then raise (Parse_Failure("Cannot bind output port to
               registers.", pos)) else ()
360      with Not_found -> ignore (get_output immod.im_modname (get_lvalue_name lval2) environ));
361      let exp1 = Lvalue(lval2, Lexing.dummy_pos) in
362        let (_, width) = get_output othermod name environ in
```

```
363                           let lst2 = add_param_pos (width - 1) 0 name lst1 pos in
364                           if width <> get_min_bit_width_expr environ immod exp1 then raise (
                                  Parse_Failure("Binding width mismatch.", pos))
365                           else let (mod2, exp2, cnt2) = translate_expr environ mod1 exp1
                                  cnt1 false in
366                           (mod2, cnt2, (name, exp2) :: bnd1, lst2))
367     with Not_found -> raise (Parse_Failure("Undefined identifier.", pos)))
368     (immod, count, [], []) bindlst) in (immod1, count1, list1)
369
370
371 (* translate_stmt: enviro -> im_moddecl -> stmt -> int -> bool -> im_moddecl *
        im_always_stmt list * int *)
372 (* If in_always is false and we are not currently in an always block (that is, if/case), *)
373 (* or, in other words, we are in a top-level statement, then statement *)
374 (* generated is returned through modification to the im_moddecl directly. *)
375 (* Otherwise, it is returned in the list for incorporation by a top-level statement *)
376 let rec translate_stmt environ immod vshstmt count in_always = match vshstmt
377     with Nop(_) -> (immod, [], count)
378   | Expr(expr, _) -> let (immod1, _, count1) = translate_expr environ immod expr count
          in_always in (immod1, [], count1)
379   | Block(lst, _) -> List.fold_left (fun (immod1, stmtlst1, count1) stmt ->
380                         let (immod2, stmtlst2, count2) = translate_stmt environ immod1 stmt
                                count1 in_always in
381                         (immod2, stmtlst1 @ stmtlst2, count2)) (immod, [], count) lst
382   | 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))
383                                       else (match cond with
384                                         Posedge -> let (immod1, stmtlist1, count1) =
                                                  translate_stmt environ immod stmt1 count true in
385                                                         ( { immod1 with im_alwaysposedge = immod1.
                                                                im_alwaysposedge @ stmtlist1 }, [],
                                                                count1 )
386                                       | Negedge -> let (immod1, stmtlist1, count1) =
                                              translate_stmt environ immod stmt1 count true in
387                                                         ( { immod1 with im_alwaysnegedge = immod1.
                                                                im_alwaysnegedge @ stmtlist1 }, [],
                                                                count1 )
388                                       | Expression(expr) -> if in_always then
389                                         (   let (immod1, stmtlist1, count1) = translate_stmt
                                                  environ immod stmt1 count true in
390                                             let (immod2, stmtlist2, count2) = translate_stmt
                                                    environ immod1 stmt2 count1 true in
391                                             let (immod3, imexpr, count3) = translate_expr
                                                    environ immod2 expr count true in
392                                             (immod3, [ImIf(imexpr, stmtlist1, stmtlist2)],
                                                    count3))
393                                         else
394                                             let (immod1, stmtlist1, count1) = translate_stmt
                                                    environ immod stmt1 count true in
395                                             let (immod2, stmtlist2, count2) = translate_stmt
                                                    environ immod1 stmt2 count1 true in
396                                             let (immod3, imexpr, count3) = translate_expr
                                                    environ immod2 expr count false in
397                                             ( {immod3 with im_alwaysall = ImIf(imexpr,
                                                    stmtlist1, stmtlist2) :: immod3.im_alwaysall},
                                                    [], count3)
398                                       )
399   | Return(expr, pos) -> translate_stmt environ immod (Assign(Identifier("return"), expr,
          pos)) count in_always
400   | Case(lvalue, lst, pos) -> let width = get_max_bit_width_expr environ immod (Lvalue(
          lvalue, pos)) in
401                             let (newmod, newcount, newlist) = List.fold_left (fun (immod1,
                                    count1, lst1) (item, stmt1, pos) ->
402                               if String.length item <> width then raise (Parse_Failure("
                                      Width mismatch in case statement.", pos))
403                               else let (immod2, stmtlist2, count2) = translate_stmt environ
                                      immod1 stmt1 count1 true in
```

```
404                             (immod2, count2, (item, stmtlist2) :: lst1)) (immod, count,
                                   []) lst in
405                           if in_always then
406                             (newmod, [ImCase(to_im_lvalue environ newmod lvalue pos, List.
                                   rev newlist)], newcount)
407                           else
408                             ({newmod with im_alwaysall = ImCase(to_im_lvalue environ
                                   newmod lvalue pos, List.rev newlist) :: newmod.
                                   im_alwaysall}, [], newcount)
409   | For(id, init, cond, incr, stmt, pos) ->
410     (* How this works: We add the loop variable as a parameter in the local parameter table,
           then translate the statement.*)
411     (* Rinse, repeat. Loop for a maximum of 1024 times. *)
412     (* First, make sure that id is not referring to anything else. *)
413     (try
414     let _ = get_param immod.im_modname id environ in raise (Parse_Failure("Loop control
          variable must not have been used previously.", pos))
415     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))
416     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))
417     with Not_found -> (
418       (* compute the initialization value *)
419       let firstval = Int64.to_int (eval_expr immod.im_modname environ init) in
420       (* build_for: int -> int -> im_moddecl -> int -> im_moddecl * im_always_stmt list *
            int *)
421       let rec build_for currval loopsleft mod1 count = (if loopsleft < 0 then raise (
            Parse_Failure("For loop has run too many times.", pos)) else
422         (*Add currval to local parameter table*)
423         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
424         let continue = eval_expr mod1.im_modname newenv cond in
425         if Int64.compare Int64.zero continue = 0 then (mod1, [], count)
426         else
427           let (mod2, stmtlist2, count2) = translate_stmt newenv mod1 stmt count in_always in
428           let newval = Int64.to_int (eval_expr mod1.im_modname newenv incr) in
429           let (mod3, stmtlist3, count3) = build_for newval (loopsleft - 1) mod2 count2 in
430           (mod3, stmtlist2 @ stmtlist3, count3)
431         )
432       in
433       let (mod4, stmtlist, newcount) = build_for firstval 1024 immod count in
434       if in_always then (mod4, stmtlist, newcount)
435       else ( { mod4 with im_alwaysall = List.rev_append stmtlist mod4.im_alwaysall}, [],
            newcount)
436     ))))
437   | Assign(lvalue, expr, pos) -> (
438     let imlvalue = to_im_lvalue environ immod lvalue pos in
439     let lvalue_width = get_lvalue_length environ immod lvalue pos in
440     let lvaluename = get_lvalue_name lvalue in
441     if lvalue_width < get_min_bit_width_expr environ immod expr then raise (Parse_Failure("
          Assignment width mismatch.", pos))
442     else if lvalue_width > get_max_bit_width_expr environ immod expr then raise (
          Parse_Failure("Assignment width mismatch.", pos))
443     else if in_always then (
444       try
445         let decl = get_local_all immod.im_modname lvaluename environ in
446         if decl.decltype = Reg then
447         let (immod1, imexpr, count1) = translate_expr environ immod expr count true in
448         (immod1, [ImRegAssign(imlvalue, imexpr)], count1)
449         else
450         let tempregname = "_reg_" ^ lvaluename in
451         if check_im_mod_local immod tempregname then
452           let (immod1, imexpr, count1) = translate_expr environ immod expr count true in
453             (immod1, [ImRegAssign(change_im_lvalue_name tempregname imlvalue, imexpr)],
                  count1)
454         else
```

```
455             let immod1 = { immod with im_declarations = (ImReg, tempregname, decl.declwidth)
                    :: immod.im_declarations;
456                                      im_assignments = (ImRange(lvaluename, decl.declwidth −
                                             1, 0), ImLvalue(ImRange(tempregname, decl.declwidth
                                             − 1, 0))) :: immod.im_assignments } in
457             let (immod2, imexpr, count1) = translate_expr environ immod1 expr count true in
458               (immod2, [ImRegAssign(change_im_lvalue_name tempregname imlvalue, imexpr)],
                      count1)
459
460         with Not_found −> let tempregname = "_reg_" ^ lvaluename in
461           let (_, width) = get_output immod.im_modname lvaluename environ in
462           if check_im_mod_local immod tempregname then
463             let (immod1, imexpr, count1) = translate_expr environ immod expr count true in
464               (immod1, [ImRegAssign(change_im_lvalue_name tempregname imlvalue, imexpr)],
                      count1)
465           else
466             let immod1 = { immod with im_declarations = (ImReg, tempregname, width) :: immod.
                    im_declarations;
467                                      im_assignments = (ImRange(lvaluename, width − 1, 0),
                                             ImLvalue(ImRange(tempregname, width − 1, 0))) ::
                                             immod.im_assignments } in
468             let (immod2, imexpr, count1) = translate_expr environ immod1 expr count true in
469               (immod2, [ImRegAssign(change_im_lvalue_name tempregname imlvalue, imexpr)],
                      count1)
470         )
471       else (
472         try
473           let decl = get_local_all immod.im_modname lvaluename environ in
474           if decl.decltype = Reg then raise (Parse_Failure("Cannot assign values to registers
                  outside if and case blocks. Use wires.", pos))
475           else let (immod1, imexpr, count1) = translate_expr environ immod expr count false in
476           if imexpr = ImNoexpr then raise (Parse_Failure("Invalid right hand side value in
                  assignment.", pos))
477           else ({immod1 with im_assignments = (imlvalue, imexpr) :: immod1.im_assignments},
                  [], count1)
478         with Not_found −>  let (immod1, imexpr, count1) = translate_expr environ immod expr
                count false in
479           if imexpr = ImNoexpr then raise (Parse_Failure("Invalid right hand side value in
                  assignment.", pos))
480           else ({immod1 with im_assignments = (imlvalue, imexpr) :: immod1.im_assignments},
                  [], count1)
481         )
482       )
483
484 let rec check_assignment_duplication startpos endpos paramname lst pos = if startpos <
        endpos then lst
485             else let newparamplace = paramname ^ (string_of_int startpos) in
486                  if List.mem newparamplace lst then raise (Parse_Failure("Duplicate assignment
                          of \"" ^ paramname ^ "["^ (string_of_int startpos) ^ "]\".", pos))
487                  else add_param_pos (startpos − 1) endpos paramname (newparamplace::lst) pos
488
489 (* translate_module: env −> mod_decl −> im_mod_decl*)
490 let translate_module environ vshmod =
491   if vshmod.libmod then { im_modname = vshmod.modname; im_libmod = true; im_libmod_name =
          vshmod.libmod_name;
492   im_libmod_width = vshmod.libmod_width; im_inputs = []; im_outputs = []; im_declarations =
          []; im_assignments = [];
493   im_instantiations = []; im_alwaysall=[];  im_alwaysposedge = [];  im_alwaysnegedge = []; }
494   else (
495   ignore (check_unique_ids_in_module environ vshmod); (* check that all identifiers used in
          the module are unique *)
496   let ret = { im_modname = vshmod.modname; im_libmod = false; im_libmod_name = "";
          im_libmod_width = 0;  im_inputs = [];
497   im_outputs = [];  im_declarations = []; im_assignments = [];  im_instantiations = [];
          im_alwaysall=[];
498   im_alwaysposedge = [];im_alwaysnegedge = [];} in
499   (* build up inputs and outputs *)
500   let ret = { ret with im_inputs = List.map (fun (i, w, _) −> (i, w)) vshmod.inputs } in
```

```
501   let ret = { ret with im_outputs = List.map (fun (i, w, _) -> ( i, w)) vshmod.outputs } in
502   (* special output for returns. Note that "return" is never a valid name because it is a
          keyword! *)
503   let ret = { ret with im_outputs = (if vshmod.returnwidth = 0 then ret.im_outputs else ("
          return", vshmod.returnwidth) :: ret.im_outputs) } in
504   (* Build up initial declarations and initializations from the ones provided.
505      Initializations *must* be to an expression evaluable at compile time.
506      This constraint and the parameter - declaration - statements sequence allows us to not
            worry about scope. *)
507   let to_im_decl_type = function Reg -> ImReg | Wire -> ImWire in
508   let (decls, assigns) = List.fold_left (fun (olddecl, oldassign) decl ->
509    (
510     ((to_im_decl_type decl.decltype), decl.declname, decl.declwidth) :: olddecl,
511     (match decl.init with
512          Noexpr(_) -> oldassign
513        | x -> (ImRange(decl.declname, decl.declwidth - 1, 0), (let value = eval_expr vshmod.
             modname environ x in
514                  if get_min_bit_width value > decl.declwidth then
515                    raise (Parse_Failure("Overflow in initialization.", decl.declpos))
516                  else ImLiteral(value, decl.declwidth))) :: oldassign
517    ))) ([], []) vshmod.declarations in
518     let ret = { ret with im_declarations = decls; im_assignments = assigns } in
519     let (immod, _, _) =
520       List.fold_left (fun (immod1, _, count) stmt -> translate_stmt environ immod1 stmt
            count false) (ret, [], 0) vshmod.statements in
521     let finalmod = { immod with im_alwaysall = List.rev immod.im_alwaysall;
          im_instantiations = List.rev immod.im_instantiations; im_assignments = List.rev
          immod.im_assignments } in
522      ignore (List.fold_left (fun lst1 lval1 -> (match lval1 with
523       ImSubscript(name, s) -> check_assignment_duplication s s name lst1 vshmod.modpos
524      | ImRange(name, up, lo) -> check_assignment_duplication up lo name lst1 vshmod.modpos
525      )) [] ((List.map (fun (s, _) -> s) finalmod.im_assignments) @
526     List.map (fun (_, exp) -> match exp with ImLvalue(l) -> l
527                | _ -> raise (Parse_Failure("Internal compiler error 005. Contact
                      manufacturer for more information.", vshmod.modpos)))
528       (List.flatten (List.map (fun (_, _, l) -> l) finalmod.im_instantiations)))); finalmod
529   )
530
531 let set_standard_library_module_info mod1 = if mod1.libmod then (
532   if mod1.libmod_width < 1 then raise (Parse_Failure("Invalid standard module width.", mod1.
        modpos))
533   else match mod1.libmod_name with
534   "JKL" -> {mod1 with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.
        dummy_pos);
535                                  ("J", mod1.libmod_width, Lexing.dummy_pos); ("K", mod1.
                                       libmod_width, Lexing.dummy_pos);
536                                  ("E", mod1.libmod_width, Lexing.dummy_pos)];
537                         outputs = [("Q", mod1.libmod_width, Lexing.dummy_pos);("QNOT", mod1.
                              libmod_width, Lexing.dummy_pos)];
538                         returnwidth = mod1.libmod_width; }
539   | "DL"  -> {mod1 with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.
        dummy_pos);
540                                  ("D", mod1.libmod_width, Lexing.dummy_pos); ("E", mod1.
                                       libmod_width, Lexing.dummy_pos)];
541                         outputs = [("Q", mod1.libmod_width, Lexing.dummy_pos);("QNOT", mod1.
                              libmod_width, Lexing.dummy_pos)];
542                         returnwidth = mod1.libmod_width; }
543   | "TL"  -> {mod1 with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.
        dummy_pos);
544                                  ("T", mod1.libmod_width, Lexing.dummy_pos); ("E", mod1.
                                       libmod_width, Lexing.dummy_pos)];
545                         outputs = [("Q", mod1.libmod_width, Lexing.dummy_pos);("QNOT", mod1.
                              libmod_width, Lexing.dummy_pos)];
546                         returnwidth = mod1.libmod_width; }
547   | "DFF" -> {mod1 with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.
        dummy_pos);
548                                  ("D", mod1.libmod_width, Lexing.dummy_pos); ("S", mod1.
                                       libmod_width, Lexing.dummy_pos)];
```

```
549                          outputs = [("Q", mod1.libmod_width, Lexing.dummy_pos);("QNOT", mod1.
                                 libmod_width, Lexing.dummy_pos)];
550                          returnwidth = mod1.libmod_width; }
551   | "TFF" -> {mod1 with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.
        dummy_pos);
552                                    ("T", mod1.libmod_width, Lexing.dummy_pos)];
553                          outputs = [("Q", mod1.libmod_width, Lexing.dummy_pos);("QNOT", mod1.
                                 libmod_width, Lexing.dummy_pos)];
554                          returnwidth = mod1.libmod_width; }
555   | "JKFF" -> {mod1 with inputs = [("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.
        dummy_pos);
556                                    ("J", mod1.libmod_width, Lexing.dummy_pos); ("K", mod1.
                                        libmod_width, Lexing.dummy_pos)];
557                          outputs = [("Q", mod1.libmod_width, Lexing.dummy_pos);("QNOT", mod1.
                                 libmod_width, Lexing.dummy_pos)];
558                          returnwidth = mod1.libmod_width; }
559   | _ -> raise (Parse_Failure("Unsupported standard module name.", mod1.modpos)))
560    else mod1
561 (* translate: mod_decl list -> im_mod_decl list *)
562
563 let translate modules =
564
565   (* Check that the module names are consistent and the return widths are valid. *)
566   let _ = check_mod_info [] modules in
567   (* Set information for standard library module declarations *)
568   let modules = List.map set_standard_library_module_info modules in
569   (* Build the environment *)
570   let environment = {
571   arg_map  = string_map_args StringMap.empty modules;
572   param_map = string_map_params StringMap.empty modules;
573   local_map = string_map_locals StringMap.empty modules;
574   return_map = string_map_returns StringMap.empty modules;
575   }
576   in List.map (translate_module environment) modules
```

asttoimst.ml

8.3 Imst.ml

```
1  open Int64
2
3  type im_op = ImPlus | ImMinus | ImMultiply | ImModulus | ImEq | ImNe | ImGe | ImLe | ImLt |
       ImGt | ImAnd | ImOr | ImXor | ImNand | ImNor | ImXnor | ImLshift | ImRshift | ImNot
4
5  type im_literal = int64 * int
6
7  type im_expr =
8      ImLiteral of im_literal
9    | ImLvalue of im_lvalue
10   | ImBinop of im_expr * im_op * im_expr
11   | ImReduct of im_op * im_lvalue
12   | ImUnary of im_op * im_expr
13   | ImConcat of im_concat_item list
14   | ImNoexpr
15 and im_concat_item =
16     ImConcatLit of int * im_literal
17   | ImConcatLvalue of int * im_lvalue
18 and im_lvalue =
19     ImSubscript of string * int
20   | ImRange of string * int * int
21
22 type im_assignment = im_lvalue * im_expr
23 type im_binding = string * im_expr
24 type im_instantiation = string * im_binding list * im_binding list
25
26 type im_always_stmt =
```

```
27     ImNop
28   | ImIf of im_expr * im_always_stmt list * im_always_stmt list
29   | ImCase of im_lvalue * im_case_item list
30   | ImRegAssign of im_lvalue * im_expr
31
32 and im_case_item = string * im_always_stmt list
33
34 type im_decl_type = ImWire | ImReg
35
36 type im_decl = im_decl_type * string * int
37
38 type im_mod_decl = {
39     im_modname : string;
40     im_libmod : bool;
41     im_libmod_name: string;
42     im_libmod_width: int;
43     im_inputs : (string * int) list;
44     im_outputs : (string * int) list;
45     im_declarations : im_decl list;
46     im_assignments : im_assignment list;
47     im_instantiations : im_instantiation list;
48     im_alwaysall : im_always_stmt list;
49     im_alwaysposedge : im_always_stmt list;
50     im_alwaysnegedge : im_always_stmt list;
51 }
52
53 type im_program = im_mod_decl list
```

imst.ml

8.4 Imsttocode.ml

```
1 open Ast
2 open Imst
3 open Parser
4 open Asttoimst
5 open Str
6
7 module StringMap = Map.Make(String)
8
9 (* prepend identifiers with an underscore so that we never use an Verilog keyword. *)
10 let mod_id id = "_" ^ id
11
12 let op_to_string = function
13     ImPlus -> "+"
14   | ImMinus -> "-"
15   | ImMultiply -> "*"
16   | ImModulus -> "%"
17   | ImEq -> "=="
18   | ImNe -> "!="
19   | ImGe -> ">="
20   | ImGt -> ">"
21   | ImLe -> "<="
22   | ImLt -> "<"
23   | ImAnd -> "&"
24   | ImOr -> "|"
25   | ImXor -> "^"
26   | ImNand -> "~&"
27   | ImNor -> "~|"
28   | ImXnor -> "~^"
29   | ImLshift -> "<<"
30   | ImRshift -> ">>"
31   | ImNot -> "~"
32
33 let stringify_lvalue = function
34     ImSubscript(id, ind) -> ((mod_id id) ^ "[" ^ (string_of_int ind) ^ "]")
```

```
35    | ImRange(id, ind1, ind2) -> ((mod_id id) ^ (if ((ind1 = 0) && (ind2 = 0)) then "" else
          "[" ^ (string_of_int ind1) ^ ":" ^ (string_of_int ind2) ^ "]"))
36
37  let stringify_concat = function
38      ImConcatLit(replications,literal) -> raise (Failure("duh."))
39    | ImConcatLvalue(replications, lv) -> string_of_int replications ^ "{" ^ (stringify_lvalue
          lv) ^ "}"
40
41  let stringify_concats lst =
42    let concat_string = List.map stringify_concat lst in
43
44        "{" ^ (String.concat ", " concat_string) ^ "}"
45
46  let update_inst_count modname inst_map =
47    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
48
49  let rec print_if_necessary str = function
50     [] -> ["._" ^ str ^ "(_" ^ str ^ ")"]
51     | (id, _) :: tl -> if id = str then [] else print_if_necessary str tl
52
53
54  let rec stringify_expression = function
55      ImLiteral(x,_) -> Int64.to_string x
56    | ImLvalue(x) -> stringify_lvalue x
57    | ImBinop(x, op, y) -> "(" ^ (stringify_expression x) ^ (op_to_string op) ^ (
          stringify_expression y ) ^ ")"
58    | ImReduct(op, y) -> "(" ^ (op_to_string op) ^ (stringify_lvalue y) ^ ")"
59    | ImUnary(op, expr) -> "(" ^ (op_to_string op) ^ (stringify_expression expr) ^ ")"
60    | ImConcat(x) -> stringify_concats x
61    | ImNoexpr -> ""
62
63  let print_inst out inst_map (modname, bindlst1, bindlst2) =
64      let new_map = update_inst_count modname inst_map in
65      let ind = StringMap.find modname new_map in
66      output_string out ("_" ^ modname ^ " ___" ^ modname ^ (string_of_int ind) ^ "(");
67      output_string out (String.concat ", " ((List.map (fun(id, exp) -> "._" ^ id ^ "(" ^ (
            stringify_expression exp) ^ ")") (bindlst1 @ bindlst2)) @ (
68      print_if_necessary "clock" bindlst1) @ (print_if_necessary "reset" bindlst1)));
69      output_string out ");\n";
70      new_map
71  let rec print_case out (b, stmt) = output_string out ((string_of_int (String.length b)) ^ "'
        b" ^ b ^ ": \n"); output_string out "begin\n"; List.iter (print_statement out) stmt;
        output_string out "end\n"
72  and print_case_list out lst = List.iter (print_case out) lst
73  and print_statement out = function
74      ImNop -> ()
75    | ImIf(pred,tru,fal) -> output_string out ("if (" ^ (stringify_expression pred) ^ ")\n");
          output_string out "begin\n"; List.iter (print_statement out) tru; output_string out "
          end\nelse\nbegin\n"; List.iter (print_statement out) fal; output_string out "end\n"
76    | ImCase(lv,csl) -> output_string out ("casex(" ^ (stringify_lvalue lv) ^ ")\n");
          print_case_list out csl; output_string out "endcase\n"
77    | ImRegAssign(lv, expr) -> output_string out ((stringify_lvalue lv) ^ "=" ^ (
          stringify_expression expr) ^ ";\n")
78
79  let print_module_sig out m =
80    (* print module sig *)
81    let mod_args = (m.im_inputs, m.im_outputs) in
82    let mod_arg_list (inputs, outputs) =
83    String.concat ", " (List.map (fun (name, _) -> "_" ^ name) (inputs @ outputs)) in
84    (output_string out ("module _" ^ m.im_modname ^ "(" ^ (mod_arg_list mod_args) ^ ");\n");
85    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);
86    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))
87
88    (* print decls *)
```

```
89 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 ^ ";\n")
90
91 let print_assignment out (lv, expr) = output_string out ("assign " ^ (stringify_lvalue lv) ^
        " = " ^ (stringify_expression expr) ^ ";\n")
92
93 (* print a standard library module *)
94 let print_libmod out libname libwidth actualname =
95   let filename = (Filename.current_dir_name ^ "/stdlib/" ^ libname ^ ".v") in
96   let chan = open_in filename in
97   try
98     while true; do
99       let rawline = input_line chan in
100      let replname = Str.global_replace (Str.regexp_string libname) actualname rawline in
101      let replwidth = Str.global_replace (Str.regexp_string "WIDTHMINUSONE") (if libwidth =
           1 then "" else ("[" ^ (string_of_int (libwidth - 1)) ^ ":0]")) replname in
102      output_string out (replwidth  ^ "\n")
103    done;
104  with End_of_file -> close_in chan
105
106 let print_module out m =
107   if m.im_libmod then print_libmod out m.im_libmod_name m.im_libmod_width m.im_modname else
        (
108    print_module_sig out m;
109    List.iter (print_decl out) m.im_declarations;
110    List.iter (print_assignment out) m.im_assignments;
111    ignore (List.fold_left (print_inst out) StringMap.empty m.im_instantiations);
112    output_string out "always @ (*) begin\n";
113    List.iter (print_statement out) m.im_alwaysall;
114    output_string out "if (_reset) begin\n";
115    List.iter (fun (typ, name, _) -> if typ = ImReg then output_string out ("_" ^ name ^ "=
         0;") else ()) m.im_declarations;
116    output_string out "end\nend\n";
117    output_string out "always @ (posedge _clock) begin\n";
118    List.iter (print_statement out) m.im_alwaysposedge;
119    output_string out "end\n";
120    output_string out "always @ (negedge _clock) begin\n";
121    List.iter (print_statement out) m.im_alwaysnegedge;
122    output_string out "end\n";
123    output_string out "endmodule\n")
124
125
126
127 let _ =
128   let inname = if Array.length Sys.argv > 1 then Sys.argv.(1) else "stdin" in
129   let inchannel = if Array.length Sys.argv > 1 then Pervasives.open_in Sys.argv.(1) else
        stdin in
130   let outchannel = if Array.length Sys.argv > 2 then Pervasives.open_out Sys.argv.(2) else
        stdout in
131   let lexbuf = Lexing.from_channel inchannel in
132   try
133     let sourcecode = List.rev (Parser.program Scanner.token lexbuf) in
134       List.iter (print_module outchannel) (translate sourcecode)
135   with Parse_Failure(msg, pos) -> print_endline (inname ^ ":" ^ (string_of_int pos.Lexing.
        pos_lnum) ^":" ^ (string_of_int (pos.Lexing.pos_cnum - pos.Lexing.pos_bol)) ^": " ^
        msg )
```

imsttocode.ml

8.5 Parser.mly

```
1 %{ open Ast
2 %}
3
4 %token SEMICOLON LPAREN RPAREN LBRACE RBRACE COMMA COLON LBRACKET RBRACKET EOF
```

```
5  %token CASE CLOCK CONCAT ELSE FOR IF INPUT MODULE NEGEDGE OUTPUT PARAMETER POSEDGE REG RESET
       RETURN WIRE
6  %token ASSIGN NOT OR XOR AND NOR XNOR NAND EQ NE GT GE LT LE LSHIFT RSHIFT PLUS MINUS
       MULTIPLY MODULUS SIGEXT
7  %token NOELSE UMINUS
8  %token <string> ID
9  %token <int> DLIT
10 %token <string> BLIT
11 %token <string> XLIT
12 %token EOF
13
14 %nonassoc NOELSE
15 %nonassoc ELSE
16 %left OR NOR
17 %left XOR XNOR
18 %left AND NAND
19 %left EQ NE
20 %left GT GE LT LE
21 %left LSHIFT RSHIFT
22 %left PLUS MINUS
23 %left MULTIPLY DIVIDE MODULUS
24 %right SIGEXT NOT UPLUS
25
26 %start program
27 %type <Ast.program> program
28
29 %%
30
31 program:
32     /* nothing */    {[]}
33   | program moddecl { $2 :: $1 }
34   | error { raise (Parse_Failure("General error.  You're screwed." , Parsing.
         symbol_start_pos () )) }
35
36 moddecl:
37   MODULE ID LPAREN input_output RPAREN LBRACE parameter_list decl_list stmt_list RBRACE {{
38     modname = $2;
39     inputs = [ ("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.dummy_pos)] @ fst $4;
40     outputs = snd $4;
41     statements = List.rev $9;
42     parameters = $7;
43     declarations = $8;
44     returnwidth = 0;
45     libmod = false;
46     libmod_name = "";
47     libmod_width = 0;
48     modpos = Parsing.symbol_start_pos ();
49     }}
50  | MODULE ID LBRACKET DLIT RBRACKET LPAREN input_output RPAREN LBRACE parameter_list
       decl_list stmt_list RBRACE {{
51     modname = $2;
52     inputs = [ ("clock", 1, Lexing.dummy_pos); ("reset", 1, Lexing.dummy_pos)] @ fst $7;
53     outputs = snd $7;
54     statements = List.rev $12;
55     parameters = $10;
56     declarations = $11;
57     returnwidth = $4;
58     libmod = false;
59     libmod_name = "";
60     libmod_width = 0;
61     modpos = Parsing.symbol_start_pos ();
62     }}
63  | MODULE ID ASSIGN ID LBRACKET DLIT RBRACKET SEMICOLON {{
64     modname = $2;
65     inputs = [];
66     outputs = [];
67     statements = [];
68     parameters = [];
```

```
69      declarations = [];
70      returnwidth = 0;
71      libmod = true;
72      libmod_name = $4;
73      libmod_width = $6;
74      modpos = Parsing.symbol_start_pos (); }}
75
76 input_output:
77        INPUT formals_opt  { $2, [] }
78      | OUTPUT formals_opt { [], $2 }
79      | INPUT formals_opt SEMICOLON OUTPUT formals_opt { $2, $5 }
80      | error { raise (Parse_Failure("Module arguments parsing error." , Parsing.
          symbol_start_pos () )) }
81
82 id_with_width:
83      ID LBRACKET DLIT RBRACKET { $1, $3, Parsing.symbol_start_pos () }
84
85 id_with_width_opt:
86      ID            { $1, 1, Parsing.symbol_start_pos () }
87    | id_with_width { $1 }
88
89 id_with_width_opt_list:
90      id_with_width_opt { [$1] }
91    | id_with_width_opt_list COMMA id_with_width_opt { $3 :: $1 }
92
93 formals_opt:
94      /* nothing */ { [] }
95    | id_with_width_opt_list   { List.rev $1 }
96
97
98 parameter_list:
99      /* nothing */ { [] }
100   | parameter_list parameter_decl { $1 @ List.rev $2 }
101
102 parameter_decl:
103     PARAMETER parameter_initialization_list SEMICOLON { $2 }
104   | error { raise (Parse_Failure("Parameter declaration error." , Parsing.symbol_start_pos
        () )) }
105
106 parameter_initialization_list:
107     parameter_initialization { [$1] }
108   | parameter_initialization_list COMMA parameter_initialization { $3 :: $1 }
109
110
111 parameter_initialization:
112     ID ASSIGN DLIT { $1, $3, Parsing.symbol_start_pos () }
113 | error { raise (Parse_Failure("Parameter initialization error." , Parsing.symbol_start_pos
      () )) }
114
115 decl_list:
116     /* nothing */ { [] }
117   | decl_list decl { $1 @ List.rev $2 }
118
119 decl:
120     WIRE wire_decl_with_opt_init_list SEMICOLON { $2 }
121   | REG reg_decl_list SEMICOLON { $2 }
122
123 wire_decl_with_opt_init_list:
124     wire_decl_with_opt_init { [$1] }
125   | wire_decl_with_opt_init_list COMMA wire_decl_with_opt_init { $3 :: $1 }
126   | error { raise (Parse_Failure("Wire declaration error." , Parsing.symbol_start_pos () ))
        }
127
128 wire_decl_with_opt_init:
129     ID { { decltype = Wire; declname = $1; declwidth = 1; init = Noexpr(Parsing.
          symbol_start_pos ()); declpos = Parsing.symbol_start_pos () } }
130   | ID LBRACKET DLIT RBRACKET { { decltype = Wire; declname = $1; declwidth = $3; init =
        Noexpr(Parsing.symbol_start_pos ()); declpos = Parsing.symbol_start_pos () } }
```

```
131   | ID ASSIGN expr { { decltype = Wire; declname = $1; declwidth = 1; init = $3; declpos =
        Parsing.symbol_start_pos () } }
132   | ID LBRACKET DLIT RBRACKET ASSIGN expr { { decltype = Wire; declname = $1; declwidth = $3
        ; init = $6; declpos = Parsing.symbol_start_pos () } }
133
134 reg_decl_list:
135     reg_decl { [$1] }
136   | reg_decl_list COMMA reg_decl { $3 :: $1 }
137   | error { raise (Parse_Failure("Register declaration error." , Parsing.symbol_start_pos ()
        )) }
138
139 reg_decl:
140     ID { { decltype = Reg; declname = $1; declwidth = 1; init = Noexpr(Parsing.
        symbol_start_pos ()); declpos = Parsing.symbol_start_pos () } }
141   | ID LBRACKET DLIT RBRACKET { { decltype = Reg; declname = $1; declwidth = $3; init =
        Noexpr(Parsing.symbol_start_pos ()); declpos = Parsing.symbol_start_pos () } }
142   | ID ASSIGN expr { raise (Parse_Failure("Registers may not be initialized." , Parsing.
        symbol_start_pos () ))  }
143   | ID LBRACKET DLIT RBRACKET ASSIGN expr { raise (Parse_Failure("Registers may not be
        initialized." , Parsing.symbol_start_pos () ))  }
144
145 stmt_list:
146     /* nothing */ { [] }
147   | stmt_list stmt { $2 :: $1 }
148
149 stmt:
150     expr SEMICOLON { Expr($1, Parsing.symbol_start_pos ()) }
151   | RETURN expr SEMICOLON { Return($2, Parsing.symbol_start_pos ()) }
152   | LBRACE stmt_list RBRACE { Block(List.rev $2, Parsing.symbol_start_pos ()) }
153   | IF LPAREN condition_clock RPAREN stmt %prec NOELSE { If($3, $5, Nop(Parsing.
        symbol_start_pos ()), Parsing.symbol_start_pos ()) }
154   | IF LPAREN expr RPAREN stmt %prec NOELSE { If(Expression($3), $5, Nop(Parsing.
        symbol_start_pos ()), Parsing.symbol_start_pos ()) }
155   | IF LPAREN expr RPAREN stmt ELSE stmt { If(Expression($3), $5, $7, Parsing.
        symbol_start_pos ()) }
156   | IF LPAREN condition_clock RPAREN stmt ELSE stmt { raise (Parse_Failure("Clock edge if
        statements may not have else clauses.", Parsing.symbol_start_pos ())) }
157   | CASE LPAREN lvalue RPAREN LBRACE case_list RBRACE { Case($3, List.rev $6, Parsing.
        symbol_start_pos ()) }
158   | FOR LPAREN ID ASSIGN expr SEMICOLON 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
        ()) }
159   | FOR LPAREN error RPAREN stmt { raise (Parse_Failure("Invalid for loop header.", Parsing.
        symbol_start_pos ())) }
160   | SEMICOLON { Nop(Parsing.symbol_start_pos ()) } /* empty statements */
161   | lvalue ASSIGN expr SEMICOLON { Assign($1, $3, Parsing.symbol_start_pos ()) }
162
163
164 condition_clock:
165     POSEDGE { Posedge }
166   | NEGEDGE { Negedge }
167
168 case_list:
169     case_item { [$1] }
170   | case_list case_item { $2 :: $1 }
171   | error { raise (Parse_Failure("Case statement error." , Parsing.symbol_start_pos () )) }
172
173 case_item:
174     BLIT COLON stmt { $1, $3, Parsing.symbol_start_pos () }
175   | XLIT COLON stmt { $1, $3, Parsing.symbol_start_pos () }
176
177 lvalue:
178     ID { Identifier($1) }
179   | ID LBRACKET expr RBRACKET { Subscript($1, $3) }
180   | ID LBRACKET expr COLON expr RBRACKET { Range($1, $3, $5) }
181
182 expr:
```

```
183      DLIT { DLiteral($1, Parsing.symbol_start_pos ()) }
184    | BLIT { BLiteral($1, Parsing.symbol_start_pos ()) }
185    | lvalue { Lvalue($1, Parsing.symbol_start_pos ()) }
186    | expr PLUS expr { Binop($1, Plus, $3, Parsing.symbol_start_pos ()) }
187    | expr MINUS expr { Binop($1, Minus, $3, Parsing.symbol_start_pos ()) }
188    | expr MULTIPLY expr { Binop($1, Multiply, $3, Parsing.symbol_start_pos ()) }
189    | expr MODULUS expr {Binop($1, Modulus, $3, Parsing.symbol_start_pos ())}
190    | DLIT SIGEXT expr {  Signext($1, $3, Parsing.symbol_start_pos ()) }
191    | expr EQ expr { Binop($1, Eq, $3, Parsing.symbol_start_pos ())}
192    | expr NE expr {Binop($1, Ne, $3, Parsing.symbol_start_pos ()) }
193    | expr GE expr {Binop($1, Ge, $3, Parsing.symbol_start_pos ()) }
194    | expr GT expr {Binop($1, Gt, $3, Parsing.symbol_start_pos ())}
195    | expr LE expr {Binop($1, Le, $3, Parsing.symbol_start_pos ()) }
196    | expr LT expr {Binop($1, Lt, $3, Parsing.symbol_start_pos ()) }
197    | expr AND expr { Binop($1, And, $3, Parsing.symbol_start_pos ())}
198    | expr OR expr {Binop($1, Or, $3, Parsing.symbol_start_pos ()) }
199    | expr XOR expr {Binop($1, Xor, $3, Parsing.symbol_start_pos ()) }
200    | expr XNOR expr { Binop($1, Xnor, $3, Parsing.symbol_start_pos ())}
201    | expr LSHIFT expr {Binop($1, Lshift, $3, Parsing.symbol_start_pos ()) }
202    | expr RSHIFT expr { Binop($1, Rshift, $3, Parsing.symbol_start_pos ())}
203    | LPAREN expr RPAREN { $2 }
204    | NOT expr { Unary(Not, $2, Parsing.symbol_start_pos ()) }
205    | PLUS expr %prec UPLUS { Unary(Plus, $2, Parsing.symbol_start_pos ()) }
206    | MINUS expr %prec UPLUS { Unary(Minus, $2, Parsing.symbol_start_pos ()) }
207    | AND lvalue %prec NOT {Reduct(And, $2, Parsing.symbol_start_pos ()) } /* reductions */
208    | OR lvalue %prec NOT {Reduct(Or, $2, Parsing.symbol_start_pos ()) }
209    | XOR lvalue %prec NOT {Reduct(Xor, $2, Parsing.symbol_start_pos ()) }
210    | NAND lvalue %prec NOT {Reduct(Nand, $2, Parsing.symbol_start_pos ()) }
211    | NOR lvalue %prec NOT {Reduct(Nor, $2, Parsing.symbol_start_pos ()) }
212    | XNOR lvalue %prec NOT {Reduct(Xnor, $2, Parsing.symbol_start_pos ()) }
213    | RESET { Reset(Parsing.symbol_start_pos ()) }
214    | CONCAT LPAREN concat_list RPAREN { Concat(List.rev $3, Parsing.symbol_start_pos ()) } /*
         Concatenation */
215    | ID LPAREN binding_in_list_opt SEMICOLON binding_out_list_opt RPAREN { Inst($1, List.rev
         $3, List.rev $5, Parsing.symbol_start_pos ()) } /*Module instantiation */
216
217  concat_list:
218      concat_item { [$1] }
219    | concat_list COMMA concat_item { $3 :: $1 }
220    | error { raise (Parse_Failure("Concatenation error.", Parsing.symbol_start_pos () )) }
221
222  concat_item:
223      BLIT { ConcatBLiteral(1, $1) }
224    | lvalue { ConcatLvalue(1, $1) }
225    | DLIT LBRACE BLIT RBRACE { ConcatBLiteral($1, $3) } /* duplicated blit */
226    | DLIT LBRACE lvalue RBRACE { ConcatLvalue($1, $3) } /* duplicated lvalue */
227
228  binding_in_list:
229    binding_in { [$1] }
230    | binding_in_list COMMA binding_in { $3 :: $1 }
231    | error { raise (Parse_Failure("Port binding error.", Parsing.symbol_start_pos () )) }
232
233  binding_in_list_opt:
234    /*nothing*/ { [] }
235    | binding_in_list { $1 }
236
237  binding_out_list:
238    binding_out { [$1] }
239    | binding_out_list COMMA binding_out { $3 :: $1 }
240    | error { raise (Parse_Failure("Port binding error.", Parsing.symbol_start_pos () )) }
241
242  binding_out_list_opt:
243    /*nothing*/ { [] }
244    | binding_out_list { $1 }
245
246  binding_in:
247      ID ASSIGN expr { $1, $3 }
248    | CLOCK ASSIGN expr { "clock", $3}
```

```
249    | RESET ASSIGN expr { "reset", $3}
250
251 binding_out:
252     ID ASSIGN lvalue { $1, $3 }
```

parser.mly

8.6 Scanner.mll

```
1 { open Parser
2
3 let incr_linenum lexbuf =
4 let pos = lexbuf.Lexing.lex_curr_p in
5 lexbuf.Lexing.lex_curr_p <- { pos with
6   Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
7   Lexing.pos_bol = pos.Lexing.pos_cnum;
8 }
9
10
11
12 }
13
14
15
16 rule token = parse
17   [' ' '\t' '\r' ]  { token lexbuf }        (*Whitespace*)
18   | '\n'  { incr_linenum lexbuf;token lexbuf }
19   | "/*"      { comment lexbuf }            (* Comments *)
20   | "//"      { comment2 lexbuf }
21   | '(' { LPAREN }
22   | ')' { RPAREN }          (*Punctuation*)
23   | '{' { LBRACE }
24   | '}' { RBRACE }
25   | '[' { LBRACKET }
26   | ']' { RBRACKET }
27   | ';' { SEMICOLON }
28   | ',' { COMMA }
29   | ':' { COLON }
30   | ['0' - '9']+  as var { DLIT( int_of_string var )  } (* Literals *)
31   | ['0' '1']+ 'b' as var { BLIT (String.sub var 0 (String.length var - 1) )  }
32   | ['0' '1' 'x']+ 'b' as var { XLIT(String.sub var 0 (String.length var - 1)) }
33   | '+'  { PLUS }  (* Operators *)
34   | '-'  { MINUS }
35   | '*'  { MULTIPLY }
36   | '%'  { MODULUS }
37   | "<<" { LSHIFT }
38   | ">>" { RSHIFT }
39   | '\'' { SIGEXT }
40   | "~&" { NAND }
41   | "~|" { NOR }
42   | "~^" { XNOR }
43   | '~'  { NOT }
44   | '&'  { AND }
45   | '|'  { OR }
46   | '^'  { XOR }
47   | "==" { EQ }
48   | "!=" { NE }
49   | "<=" { LE }
50   | ">=" { GE }
51   | '<'  { LT }
52   | '>'  { GT }
53   | '='  { ASSIGN }
54   | "case"      { CASE }  (*Keywords*)
55   | "clock"     { CLOCK }
56   | "concat"    { CONCAT }
57   | "else"      { ELSE }
```

```
58    | "for"        { FOR }
59    | "if"         { IF }
60    | "input"      { INPUT }
61    | "module"     { MODULE }
62    | "negedge"    { NEGEDGE }
63    | "output"     { OUTPUT }
64    | "parameter"  { PARAMETER }
65    | "posedge"    { POSEDGE }
66    | "register"   { REG }
67    | "return"     { RETURN }
68    | "reset"      { RESET }
69    | "wire"       { WIRE }
70    | eof          { EOF }                    (*EOF*)
71    | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as var { ID(var) }
72    | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
73
74  and comment = parse
75    "*/" { token lexbuf }
76  | '\n' { incr_linenum lexbuf; comment lexbuf}
77  | _    { comment lexbuf }
78
79  and comment2 = parse
80    '\n' { incr_linenum lexbuf; token lexbuf }
81  | _ {comment2 lexbuf}
```

scanner.mll

9 Example files

9.1 Gcd.vs

```
1  module gcd(input num0[8], num1[8], start; output greatest[8], success) {
2    register found;
3    success = found;
4
5    if (start) {
6      if (num0<num1)
7        greatest = num0;
8      else
9        greatest = num1;
10     }
11
12   if (posedge) {
13     if (~found) {
14       if (num0%greatest==0 & num1%greatest==0)
15         found = 1;
16       else
17         greatest = greatest -1;
18       }
19     }
20   }
```

examples/gcd.vs

9.2 Gcd.v

```
1  module _gcd(_clock, _reset, _num0, _num1, _start, _greatest, _success);
2  input _clock;
3  input _reset;
4  input [7:0] _num0;
5  input [7:0] _num1;
6  input _start;
```

```
7  output [7:0] _greatest;
8  output _success;
9  reg [7:0] __reg_greatest;
10
11 reg _found;
12
13 assign _success = _found;
14 assign _greatest[7:0] = __reg_greatest[7:0];
15 always @ (*) begin
16 if (_start)
17 begin
18 if ((_num0[7:0]<_num1[7:0]))
19 begin
20 __reg_greatest[7:0]=_num0[7:0];
21 end
22 else
23 begin
24 __reg_greatest[7:0]=_num1[7:0];
25 end
26 end
27 else
28 begin
29 end
30 if (_reset) begin
31 __reg_greatest= 0;
32 _found= 0;
33 end
34 end
35 always @ (posedge _clock) begin
36 if ((~_found))
37 begin
38 if ((((_num0[7:0]%_greatest[7:0])==0)&((_num1[7:0]%_greatest[7:0])==0)))
39 begin
40 _found=1;
41 end
42 else
43 begin
44 __reg_greatest[7:0]=(_greatest[7:0]-1);
45 end
46 end
47 else
48 begin
49 end
50 end
51 always @ (negedge _clock) begin
52 end
53 endmodule
```

examples/gcd.v

9.3 Stim file for gcd

```
1  // This is a very rudimentary stim file for the gcd. It will input two numbers, 36 and 24,
       and find the greatest common denominator
2  module stim();
3
4    // reg feeds input, wires get output
5    reg clk;
6    reg reset;
7    reg [7:0]num1;
8    reg [7:0]num0;
9    reg [3:0]count;
10   reg start;
11   wire [7:0]greatest;
12   wire success;
13
```

```
14    // instance the dut
15    _gcd dut(
16      ._clock(clk),
17      ._reset(reset),
18      ._num0(num0),
19      ._num1(num1),
20      ._start(start),
21      ._greatest(greatest),
22      ._success(success)
23      );
24
25    initial begin
26      reset = 1;
27
28      #1 clk = 0;
29      #1 clk = 1; // first positive edge, i.e. first cycle.  Zero out the module
30
31      reset = 0;
32      start = 1;
33      num0 = 36;
34      num1 = 24;
35
36      #1 clk = 0;
37      #1 clk = 1;
38    $display("Found:%d Current greatest %d\n",success,greatest);
39
40      #1 clk = 0;
41      #1 clk = 1;
42    $display("Found:%d Current greatest %d\n",success,greatest);
43      #1 clk = 0;
44      #1 clk = 1;
45    $display("Found:%d Current greatest %d\n",success,greatest);
46      #1 clk = 0;
47      #1 clk = 1;
48    $display("Found:%d Current greatest %d\n",success,greatest);
49      #1 clk = 0;
50      #1 clk = 1;
51    $display("Found:%d Current greatest %d\n",success,greatest);
52      #1 clk = 0;
53      #1 clk = 1;
54    $display("Found:%d Current greatest %d\n",success,greatest);
55      #1 clk = 0;
56      #1 clk = 1;
57    $display("Found:%d Current greatest %d\n",success,greatest);
58      #1 clk = 0;
59      #1 clk = 1;
60    $display("Found:%d Current greatest %d\n",success,greatest);
61      #1 clk = 0;
62      #1 clk = 1;
63    $display("Found:%d Current greatest %d\n",success,greatest);
64      #1 clk = 0;
65      #1 clk = 1;
66    $display("Found:%d Current greatest %d\n",success,greatest);
67      #1 clk = 0;
68      #1 clk = 1;
69    $display("Found:%d Current greatest %d\n",success,greatest);
70      #1 clk = 0;
71      #1 clk = 1;
72    $display("Found:%d Current greatest %d\n",success,greatest);
73      #1 clk = 0;
74      #1 clk = 1;
75    $display("Found:%d Current greatest %d\n",success,greatest);
76      #1 clk = 0;
77      #1 clk = 1;
78    $display("Found:%d Current greatest %d\n",success,greatest);
79      #1 clk = 0;
80      #1 clk = 1;
81    $display("Found:%d Current greatest %d\n",success,greatest);
```

```
82        #1 clk = 0;
83        #1 clk = 1;
84     $display("Found:%d Current greatest %d\n",success,greatest);
85        #1 clk = 0;
86        #1 clk = 1;
87     $display("Found:%d Current greatest %d\n",success,greatest);
88        #1 clk = 0;
89        #1 clk = 1;
90     $display("Found:%d Current greatest %d\n",success,greatest);
91        #1 clk = 0;
92        #1 clk = 1;
93     $display("Found:%d Current greatest %d\n",success,greatest);
94        #1 clk = 0;
95        #1 clk = 1;
96     $display("Found:%d Current greatest %d\n",success,greatest);
97        #1 clk = 0;
98        #1 clk = 1;
99     $display("Found:%d Current greatest %d\n",success,greatest);
100       #1 clk = 0;
101       #1 clk = 1;
102    $display("Found:%d Current greatest %d\n",success,greatest);
103       #1 clk = 0;
104       #1 clk = 1;
105    $display("Found:%d Current greatest %d\n",success,greatest);
106       #1 clk = 0;
107       #1 clk = 1;
108    $display("Found:%d Current greatest %d\n",success,greatest);
109       #1 clk = 0;
110       #1 clk = 1;
111    $display("Found:%d Current greatest %d\n",success,greatest);
112       #1 clk = 0;
113       #1 clk = 1;
114    $display("Found:%d Current greatest %d\n",success,greatest);
115       #1 clk = 0;
116       #1 clk = 1;
117    $display("Found:%d Current greatest %d\n",success,greatest);
118       #1 clk = 0;
119       #1 clk = 1;
120    $display("Found:%d Current greatest %d\n",success,greatest);
121       #1 clk = 0;
122       #1 clk = 1;
123    $display("Found:%d Current greatest %d\n",success,greatest);
124       #1 clk = 0;
125       #1 clk = 1;
126
127    #1 $finish;
128    end
129 endmodule
```

examples/gcdstim.v

9.4 Helloworld.vs

```
1  module helloWorld(input enable; output letter[8]) {
2    register count[4];
3    wire c[4];
4    c = count;
5
6    if (enable) {
7      case(c) {
8        0001b:letter=01001000b;
9        0010b:letter=01100101b;
10       0011b:letter=01101100b;
11       0100b:letter=01101100b;
12       0101b:letter=01101111b;
13       0110b:letter=00100000b;
```

```
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)
13 begin
14 casex(_c[3:0])
15 4'b0001:
16 begin
17 __reg_letter[7:0]=72;
18 end
19 4'b0010:
20 begin
21 __reg_letter[7:0]=101;
22 end
23 4'b0011:
24 begin
25 __reg_letter[7:0]=108;
26 end
27 4'b0100:
28 begin
29 __reg_letter[7:0]=108;
30 end
31 4'b0101:
32 begin
33 __reg_letter[7:0]=111;
34 end
35 4'b0110:
36 begin
37 __reg_letter[7:0]=32;
38 end
39 4'b0111:
40 begin
41 __reg_letter[7:0]=87;
42 end
43 4'b1000:
44 begin
45 __reg_letter[7:0]=111;
46 end
47 4'b1001:
```

```
48 begin
49 __reg_letter[7:0]=114;
50 end
51 4'b1010:
52 begin
53 __reg_letter[7:0]=108;
54 end
55 4'b1011:
56 begin
57 __reg_letter[7:0]=100;
58 end
59 4'b1100:
60 begin
61 __reg_letter[7:0]=33;
62 end
63 endcase
64 end
65 else
66 begin
67 end
68 if (_reset) begin
69 __reg_letter= 0;_count= 0;end
70 end
71 always @ (posedge _clock) begin
72 _count[3:0]=(_count[3:0]+1);
73 end
74 always @ (negedge _clock) begin
75 end
76 endmodule
```

examples/helloworld.v