The DIL Programming Language

Seth Copen Goldstein and Mihai Budiu
{seth,mihaib}@cs.cmu.edu

April 1, 1999
Contents

1 Introduction ................................................................. 3
  1.1 Dil and Reconfigurable Computing .................................. 3
  1.2 Overview of the Dil Language ....................................... 3

2 Dil Syntax ............................................................................. 6
  2.1 Introduction .................................................................. 6
  2.2 Syntax ......................................................................... 6
    2.2.1 Keywords ............................................................. 6
    2.2.2 Identifiers ............................................................ 6
    2.2.3 Constants ............................................................. 7
    2.2.4 Comments ............................................................. 7
  2.3 Types and Type Declarations .......................................... 7
    2.3.1 Types ................................................................. 7
  2.4 Structure of a Program ................................................ 8
    2.4.1 Module Declarations ............................................. 8
    2.4.2 Function Definition .............................................. 9
    2.4.3 Main ................................................................. 10
  2.5 Variable Declarations ................................................ 10
  2.6 Statements .................................................................... 10
    2.6.1 Assignment ......................................................... 11
    2.6.2 Function Call ....................................................... 11
    2.6.3 For Statement ....................................................... 11
  2.7 Expressions ................................................................... 12

3 Invoking the compiler ....................................................... 14
  3.1 Compiling ..................................................................... 14
    3.1.1 Generating C++ code ........................................... 15
  3.2 Using the C++ simulation file ........................................ 15
    3.2.1 Generating C Code ............................................... 16
    3.2.2 dil2c ................................................................. 16
  3.3 Example ....................................................................... 16
  3.4 Automatically generating drivers .................................... 18
  3.5 The Expressiveness of the Dil Language ......................... 19
3.6 What is the Dil Compiler Doing ........................................... 20

4 Semantics ................................................................................. 21
  4.1 Guiding principles: ............................................................. 21
  4.2 Conversions ....................................................................... 22
  4.3 Accepted input types; output types .................................... 22
  4.4 Widths .............................................................................. 23
  4.5 Examples ........................................................................... 24

5 The Dil Library ........................................................................... 26
  5.1 Introduction ......................................................................... 26
    5.1.1 The Dil Compilation Process ....................................... 26
  5.2 The Library ......................................................................... 27
  5.3 Examples ........................................................................... 29
    5.3.1 Synthesizing Non-Primitive Operations ....................... 29
    5.3.2 Synthesizing Large Operations .................................... 30
    5.3.3 Synthesizing Complex Modules ................................... 32
    5.3.4 Synthesizing new data types ....................................... 34
  5.4 Conclusion ......................................................................... 34

A The Dil Grammar ...................................................................... 37
Chapter 1

Introduction

This manual is available at

1.1 Dil and Reconfigurable Computing

The Dil language is a Data-Flow Intermediate language used to describe hardware computations. Its main target are reconfigurable computing devices. The current version of the Dil compiler (1.1.3) is targeted especially for the PipeRench CVH (Configurable Virtual Hardware) chip, developed at CMU (see also http://www.ece.cmu.edu/research/piperench for more information about this architecture). Ideally the compiler should be retargetable to other reconfigurable hardware platforms, like the Xilinx chips, but we have not attempted such a port yet.

The intended purpose of the Dil language is to be an intermediate step in the compilation from a high-level language (like C, Java or ML) towards a reconfigurable hardware platform.

The Dil compiler takes as input a circuit description written in the Dil language (presented in this document) and outputs configuration information for the target architecture. The current version of the Dil compiler supports one “real” hardware instance (PipeRench) for which it generates assembly code. (The assembly code is further processed with the CVH assembler cvhasm for generation of hardware configurations.)

Dil can also generate a C++ program which simulates the target circuit. For debugging purposes Dil can generate a “graphical” description of the circuit (in the language of the xfig 3.2 drawing program). The circuit figures in this paper are generated using the latter facility.

1.2 Overview of the Dil Language

This section gives a flavor of the capabilities of the Dil programming language. The following chapters give more details about the syntax, semantics and implementation.

Dil is a single-assignment language. This means that any variable can be assigned only once. This is natural if we think at the variables as wires which connect operators (gates). Any Dil program specifies a hardware circuit, with a set of inputs and a set of outputs. The sizes (i.e. bit-widths) of the values in the circuit can either be specified by the user or inferred by the compiler from the I/O sizes specifications (the sizes of the inputs and outputs always have to be specified).
Types

Dil variables can be either signed or unsigned. There are no fixed widths, like in conventional C-style languages; the computations are carried internally on widths large enough to ensure no precision loss (there are currently some limitations in the values the compiler can handle, but these are artifacts of the C++ implementation and should disappear in future versions when we add support for large numbers. Notably, very large constants cannot appear in the source code and cannot be generated in the simulated C program. Other than that, internally there’s no real limitation on the types or values manipulated.)

Statements

A Dil program describes a synchronous circuit. The current version of the language allows only combinatorial circuits (i.e., no feedback loops, where the value of a variable is computed in terms of itself, can be expressed), possibly containing “registers”, (delay elements), which store and delay the propagation of one value for one clock cycle. A future version will support sequential circuits in limited forms (the PipeRench architecture provides a very limited support for sequential circuits).

The only statement of the language is the delayed assignment:

\[ \text{var } <n> = \text{expression}; \]

\( n \) is the amount of delay between the computation of the value of the expression and the actual assignment. There is a special syntax for denoting an instantaneous assignment, using simply the “=” sign. The syntax \text{var } <0> = \text{expression} has a special meaning, inserting one no-op (no operation) between the wires representing the output value of the expression and \text{var}. This distinction may be important in some circumstances.

Expressions

Expressions can be built using a wide range of operators. The syntax and semantics of Dil are very similar to C; here is the full range of operators, in order of increasing priority:

- \&, |, ^, ~
- \!, +, unary +
- \!-(n): sign copy operator. This operator has been artificially introduced to support sign-extension on the PipeRench target. In the future it will probably be replaced with some library operator. This operator should never be present in user-written programs; it is synthesized internally only.
- ==, !=, &&, ||, !, <, >, <=, >=
- ?: (multiplication), \% (modulo), / (division)
- <<, >> (bit range selection), a.b (concatenation)

Compile-time Macros

The Dil language offers additional constructs for increased expressiveness. These constructs are completely evaluated at compilation time. This is natural when we think that we are actually describing a (fixed) circuit. The special constructs are:

- Arrays of variables or constants;
- An if-else construct;
- A for construct (the induction variable can be assigned many times, unlike other types of variables; the induction variable actually stands for a series of constants);
- A module construct, reminiscent of procedures (but which cannot be recursive).

A Dil program is a collection of global constant declarations (i.e. variables whose value can be evaluated at compile-time) and module definitions. The variables in a module have a scope limited to that module. All variables defined outside modules have to have constant value.
Chapter 2

DIL Syntax

2.1 Introduction

DIL is a compiler for a data-flow intermediate language, also called DIL. The DIL compiler generates configurations for PipeRench (one must use the cvhasm assembler as the last stage of the compilation). In addition, it has the option of creating a C++ simulation of the hardware circuit which can be run stand-alone. The stand-alone C++ simulation is significantly faster than cvhsim simulations.

This chapter should serve as a language. It contains some examples of DIL code. General syntax is described in section 2.2, followed by descriptions of types and type declarations. The section 2.4 describes the structure of a program and function definitions. Variable declarations are discussed in (section 2.3), and then descriptions of statements (section 2.6).

2.2 Syntax

The syntax is very similar to the C language. The complete grammar is presented in Appendix A. Every statement must be terminated by a semicolon (';'). Programs consist of declarations and statements. All variables must be declared before they are used. The statements and declarations are evaluated in the order they appear in the program.

2.2.1 Keywords

There are very few reserved words. They are:

```
else fixed for if
in out typedef
```

2.2.2 Identifiers

All identifiers must start with an alphabetic character which can be followed by alphanumeric characters and underscores. This differs from C in that a variable name cannot start with an
underscore. The identifiers beginning with lib. are reserved for the Dil library (presented in Chapter 5).

2.2.3 Constants

Currently the compiler only supports integer constants. They have the same form as C integer constants. All constants must start with a digit or a minus ('-') sign. Constants are interpreted as decimal numbers unless they start with a zero and contain only the digits '0'-'7', in which case they are octal, or start with a '0x' and consist of the 10 digits and the letters 'a'-'f', in which case they are hexadecimal.

2.2.4 Comments

Dil uses C++ style comments. A comment begins with two slashes ('//'). It is terminated by the end of the line.

2.3 Types and Type Declarations

All user defined variables and types must be defined before they are used.

2.3.1 Types

There are two basic types, signed and unsigned values. A type allows the programmer to specify the width of the variable and whether it is signed or unsigned. A type is specified as: fixed<\w, s>\1

where

w is the width of the entire variable (including sign and fractional part) or 's' indicating unknown width.

s is either 0, indicating an unsigned quantity, or 1, indicating a signed quantity, or '* for an unknown sign.

User defined types can be created using the typedef keyword. A type declaration has the following syntax:

```plaintext
typedef type newtype;
```

where

**type** is a previously declared type or a built-in type.

**newtype** is declared to have the same type as **type**.

Below are some examples using typedef to declare fixed-point types.

\1The syntax allows usage of the form fixed<a, b, c>, with 3 arguments. The keyword "fixed" and the three arguments were introduced initially to allow Dil to handle fixed-point numbers. Fixed point numbers were dropped (for now) from the language, but the syntax is still there. The third argument must always be zero if present.
typedef fixed<*,0> uint; declares uint to be unsigned integers of arbitrary width.
typedef fixed<*,1> int; declares int to be a signed integer of arbitrary width.
typedef fixed<8,1> sample; declares sample to be a signed integer of 8-bits. Thus, it has a range of [-127, 127].
typedef fixed<10,0,4> frac; This is not supported, because the fraction is not zero.

2.4 Structure of a Program

A program consists of zero or more type or modules declarations, and one or more module definitions. At least one module must be defined. Every program must define at least the module main. Separate compilation is not supported. Modules must be declared or defined before they are used.

2.4.1 Module Declarations

We will interchangeably use the term “module”, “function” and “procedure”.

A function declaration declares a function name and the arguments for the function along with the argument types. The syntax for a function declaration is:

\[ \text{name} ( \text{parameters} ) ; \]

where

\text{name} is the name of the function being declared.

\text{parameters} is a comma separated list of parameters as described below.

The syntax for \text{parameters} is:

\[ \text{param}[ , \text{param}]^* \]

where

\text{param} is the declaration of a parameter.

A parameter declaration defines the type of the parameter, the name, and finally its in/out status. Parameters are either in, out, or regular. An input parameter is an input to the function, an output parameter is an output. A regular parameter can be either of them\(^2\)

A \text{param} is:

\[ \text{type \ identifier}[ \text{inout}] \]

where

\text{type} is previously declared or built-in type.

\text{identifier} is the name of the parameter.

\text{inout} is either in or out.

An example function declaration is:

\[^2\text{The keywords in and out serve mere as comments; because of the way the evaluation proceeds the processing applied on a supplied argument is the same, even if it is labeled in, out or nothing. This part of the syntax may be revised in the future.}\]
typedef fixed<*, 0> uint;
tap(uint x in, uint xout out,
    uint y in, uint yout out,
    uint weight);

This declares a function tap which has two input values (x and y), provides two output values (xout and yout) and receives an extra parameter weight.

### 2.4.2 Function Definition

A function definition defines the parameter list for a function and the body of the function. Functions cannot be called recursively. In fact, function invocation is really macro expansion.

\[
\text{name ( parameters ) \{ statements \}}
\]

where

- **name** is the name of the function being defined.
- **parameters** is a comma separated list of parameters as described in the section 2.4.1.
- **statements** is a list of statements.

An example function definition is:

typedef fixed<*, 0> uint;
tap(uint x in, uint xout out, uint y in, uint yout out, 
    uint w)
{
    yout = y + x*w;
    xout <i= xin;
}

When a function is invoked, the actual arguments are assigned to the formal arguments, and afterwards the call statement is replaced with the function body (inlined).

As a consequence of the way the function evaluation is done, one can pass array arguments to functions like ordinary variables.

For added convenience, the functions which have only one output value can be called using the C function syntax (but not declared that way). For instance one can write:

```
max(fixed<4, 0> a in, fixed<4, 0> b in, fixed<4, 0> c out)
{
    c = (a > b) ? a : b;
}
```

...


2.4.3 Main

The main function is the function that generates the complete program. All of its parameters must be either in or out parameters, since they specify the I/O interface to the function. The types of the inputs and outputs of main have to be fully specified (i.e., they cannot have an unknown size or sign).

2.5 Variable Declarations

All variables must be declared before they are used. A variable declaration has the form:

\[ type \ (var[ = init])^\star; \]

where

- **type** is a previously declared type or a built-in type.
- **varlist** is a comma separated list of variables all declared to have type **type**.
- **init** if present is an initializer that must evaluate to a manifest constant.

Variables may be either scalar variables or arrays. The bounds on an array must evaluate to a constant at compile time. An array variable is one which has an array bounds specifier attached. Arrays are indexed starting at zero, in C style. For example:

```c

fixed<4,0> aux;
aux = max(3,2);
```

```c

type = (var[ = init])^\star;
```

**fixed<8,0> beast;**

```c

uint w[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
uint x[10];
```

declares a scalar and two arrays. beast is an unsigned 8-bit wide scalar, w is an unsigned array initialized so that \( w[i-1] = i \), x is an unsigned array that has not been initialized.

Variables are used either to represent wires, i.e., connections between operators, or to construct expressions evaluated at compile time. A variable used to represent a wire is write-once. All parameters which are either in and out must be used to represent wires. All variables which have an initializer are evaluated at compile time. Only context can indicate whether a variable is used to represent wires or not.

2.6 Statements

There are two kinds of statements: compile-time statements and run-time statements. Compile time statements are evaluated at compile time and are used essentially as pre-processor statements to aid in program construction. Run-time statements do the real work in the program.
2.6.1 Assignment

There is only one runtime statement in the language, assignment. The assignment statement has two flavors: immediate or delayed. The immediate assignment equates the right-hand side of the statement (an expression) with a variable. The delayed assignment statement assigns the current value of the right-hand side to the variable on the left after the specified delay in terms of clock ticks. The assignment statements have the form:

\[ lval = rval; \]
\[ lval <constant> = rval; \]

where

- \( lval \) is a variable.
- \( rval \) is an expression.
- \( constant \) is a manifest constant indicating the number of clock ticks before the current value of \( rval \) is assigned to the \( lval \).

For example, one tap of an FIR filter in a double-pipined design would be written as:

\[
y2 <= x1 * 2 + y1; \quad \text{// y2} \leftarrow x1 \ast 2 + y1 \text{ after 1 clock ticks}
x2 <= x1; \quad \text{// x2} \leftarrow x1 \text{ after 2 clock ticks}
\]

One may think of an assignment delayed with \( n \) units as introducing \( n \) registers between the computation of the right-hand side and the left-hand side.

The meaning of \( <0> \) is special: it means a Noop, which is different from a simple assignment in that it might take space in the final circuit.

2.6.2 Function Call

A function call indicates that the body of the function being called should be inlined into the program at the point of the call. The arguments in the call replace the parameters in the function definition. An example call using the \( \text{tap} \) function defined previously is:

\[ \text{tap(xin, x[1], 0, y[1], w[0]);} \]

Because the language does not allow any side-effects, the order of the argument evaluation does not matter. It also does not matter if the arguments are evaluated or not before substitution; the result is the same. Common subexpression elimination should guarantee that there is no penalty for substituting several times a large expression for many arguments.

2.6.3 For Statement

The \( for \) statement allows one to replicate code easily. It is also a case where a variable (the index variable) can be assigned to more than once. Such a variable does not represent a wire, it is representing a series of compile-time constants. The syntax is:

\[ \text{for ( assignment; test; increment ) block} \]

where
**assignment** is an immediate assignment statement.

**test** is an expression resulting in a 0 or non-zero value.

**increment** is an immediate assignment statement (no semicolon after it).

**block** is a brace-enclosed statement list.

Notice that, unlike C, and like in Perl, the body of the loop must be enclosed between braces.

An example for statement declared types, and functions:

```c
main(fixed<8,0> xin in, uint yout out)
{
    uint w[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    uint x[10];
    uint y[10];
    uint i;
    uint taps = 5;

    tap(xin, x[0], 0, y[0], w[taps-1]);
    for (i=1; i<(taps-1); i=i+1) {
        tap(x[i-1], x[i], y[i-1], y[i], w[taps-i-1]);
    }
    tap(x[i-1], x[i], y[i-1], yout, w[0]);
}
```

### 2.7 Expressions

The standard C operators, plus a few additional ones are allowed in expressions, with the order of precedence and associativity shown below. Operators at the top have higher precedence. All operators within horizontal lines have the same precedence. Use parenthesis when you are not sure.

When the compiler evaluates a compile-time constant, the declared width of the constant is ignored.
<table>
<thead>
<tr>
<th>operator</th>
<th>assoc</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fixed&lt;width, sign&gt;) x</td>
<td>r</td>
<td>cast x to another type</td>
</tr>
<tr>
<td>(x1)</td>
<td>l</td>
<td>x1</td>
</tr>
<tr>
<td>x1[h,l]</td>
<td>l</td>
<td>selects bits h..l from x1. zero relative to the lsb of x1.</td>
</tr>
<tr>
<td>~x1</td>
<td>l</td>
<td>bitwise not of x1</td>
</tr>
<tr>
<td>! x1</td>
<td>l</td>
<td>logical not of x1</td>
</tr>
<tr>
<td>- x1</td>
<td>l</td>
<td>-1 * x1</td>
</tr>
<tr>
<td>-(2)x+</td>
<td>l</td>
<td>The sign-copy operator applied to x</td>
</tr>
<tr>
<td>x1 &lt;&lt;= x2</td>
<td>l</td>
<td>x1 shifted right by x2 positions.</td>
</tr>
<tr>
<td>x1 &gt;&gt; x2</td>
<td>l</td>
<td>x1 shifted left by x2 positions.</td>
</tr>
<tr>
<td>x1 % x2</td>
<td>l</td>
<td>x1 mod x2</td>
</tr>
<tr>
<td>x1 / x2</td>
<td>l</td>
<td>x1 divided by x2</td>
</tr>
<tr>
<td>x1 ^ x2</td>
<td>l</td>
<td>x1 xor x2 bitwise.</td>
</tr>
<tr>
<td>x1</td>
<td></td>
<td>x2</td>
</tr>
<tr>
<td>x1 &amp;&amp; x2</td>
<td>l</td>
<td>x1 logical and x2.</td>
</tr>
<tr>
<td>x1</td>
<td>x2</td>
<td>l</td>
</tr>
<tr>
<td>x1 &amp; x2</td>
<td>l</td>
<td>x1 bitwise and with x2</td>
</tr>
<tr>
<td>x1 * x2</td>
<td>l</td>
<td>x1 * x2</td>
</tr>
<tr>
<td>x1 + x2</td>
<td>l</td>
<td>x1 + x2</td>
</tr>
<tr>
<td>x1 &gt;= x2</td>
<td>l</td>
<td>if x1 ≥ x2 then 1 otherwise 0.</td>
</tr>
<tr>
<td>x1 &lt;= x2</td>
<td>l</td>
<td>if x1 ≤ x2 then 1 otherwise 0.</td>
</tr>
<tr>
<td>x1 != x2</td>
<td>l</td>
<td>if x1 ≠ x2 then 1 otherwise 0.</td>
</tr>
<tr>
<td>x1 == x2</td>
<td>l</td>
<td>if x1 = x2 then 1 otherwise 0.</td>
</tr>
<tr>
<td>x1 &gt; x2</td>
<td>l</td>
<td>if x1 &gt; x2 then 1 otherwise 0.</td>
</tr>
<tr>
<td>x1 &lt; x2</td>
<td>l</td>
<td>if x1 &lt; x2 then 1 otherwise 0.</td>
</tr>
<tr>
<td>x1 . x2</td>
<td></td>
<td>the bitstring concatenation of x1 followed by x2.</td>
</tr>
<tr>
<td>b ? x1 : x2</td>
<td>r</td>
<td>if (b ≠ 0) x1 otherwise x2.</td>
</tr>
</tbody>
</table>
Chapter 3

Invoking the compiler

This chapter gives instructions for invoking the compiler and running the produced simulations in section 3.1. An example is presented in section 3.3. Finally, some general comments on the language and compiler are given in section 3.5.

3.1 Compiling

We will assume that the environment variable $DIL holds the path to the compiler installed distribution. For CMU the value of $DIL is /afs/ece/usr/reconf.

When invoking the compiler the one required argument is an input filename. There are several option switches:

- **A filename** read target hardware architectural parameters from the indicated file.
- **C** compile to C++ code.
- **D pass** make specified pass dump debugging information.
- **K** turn on internal self-checks (used for development; compilation is slower).
- **S pass** dump statistics about the size of the program after pass.
- **X pass** dump the graph after the given pass in the format of the VCG visualisation tool.
- **W number** Set warning and debugging level to number. The higher the number the more warning/debugging information is dumped.
- **W error** transforms warnings into errors.
- **a** compile to CVHASM code.
- **c** compile to C code.
- **f pass** skip the execution of pass.
- **h** prints out a help message and the compiler version number.
- **k** turns off internal self-checks (this is the default value).
- **o filename** basename of the output file is filename (i.e. the assembly will be written in filename.cvh, the generated C++ code in filename.cc, etc.).
- **s** reads the input from standard input.
- **t** reports execution times.
--v pass output a text dump of the graph after pass.
--w turn off all warnings. not a good idea.
--x pass output an xfig dump of the graph after pass.
--y turn on parsing debugging (yydebug).

For information about how to specify the architectural parameters please see section 5.2.

3.1.1 Generating C++ code

If C++ code is produced, the C++ file can be linked with the Dil library to produce an executable which will simulate the circuit specified in the input file. You can compile the Dil file foo.dil into C++ as follows:

dil -C -o foo foo.dil

Once you have produced a C++ simulation output file, you can compile it with g++. You will need to add a directory to the include path. Assuming you ran Dil to produce foo.cc, you can compile foo.cc as follows:


g++ -I$DIL/include -o foo.o foo.cc

You can then link the executable with the following:

g++ -L$DIL/lib object-files -ldil

3.2 Using the C++ simulation file

The C++ file created will export a function name(in-args, out-args), where name is the name of the source file in which main was defined. The in-args are the variables designated as input values to the graph arguments, i.e., they have the modifier in in their declaration. The out-args are the variables designated as output values. The in-args are passed as values, the out-args are passed as references. For example, the code fragment:

main(fixed<8,0> xin in, uint yout out) {
    ...
}

if found in the source file “fir.dil” Produces the following function:

void fir(unsigned , unsigned& );

Each time the fir function is called, the circuit is run for one clock-tick. If you don’t care about the input or output values, you can pass garbage (e.g. to drain the pipeline).

In addition to the .cc file, the compiler also produces a .h file which includes a declaration for the function and a #define for the variable LATENCY_TO_FIRST_OUTPUT which indicates how many clock ticks until the first valid output appears. This is the time taken by the first input to propagate to the last output.

If debugging information is turned on for the C/C++ code generation pass, the compiler will insert cout/printf statements in the resulting C/C++ code, which print on stdout the values of the internal variables after each clock cycle. The following command line will achieve that:
dil -C -D C++ -o foo foo.dil

Normally the C++ code will only print the variables that were present in the user input program. Some of these may be absent, due to optimizations of the code.

Printing of all the internal variables generated is achieved by using a higher debugging/warning level:

dil -C -W 2 -D C++ -o foo foo.dil

### 3.2.1 Generating C Code

Using the `-c` flag generates C code: two C files and one C++ file. The C++ file is a C++ wrapper for the C functions, which allows linking the C code generates with either a C or a C++ program. Compiling the file `foo.dil` will generate three files:

- `foo.aux.cc`: the C++ wrapper.
- `foo.c`: the C code simulating the circuit.
- `foo.h`: a header file for the C functions.

### 3.2.2 dil2c

To aid the debugging of the Dil code we have supplied a program called dil2c which generates the equivalent C++ code of a Dil program, without executing any program optimizations or transformations. This is achieved practically by skipping all the compiler passes except parsing, evaluation and code generation. Here's an example usage of this utility:

dil2c -o foo foo.dil

### 3.3 Example

The following Dil code implements an 5 tap FIR filter:

```c
// double pipeline x, single pipeline y implementation of a
// n tap fir filter on 8 bit unsigned input data.

// declare an unsigned int of unknown width
typedef fixed<*, 0> uint;

// define a function which implements a single tap
tap(uint xin in, uint xout out, 
    uint yin in, uint yout out, uint w)
{
    yout = yin + xin*w;
    xout <<= xin;
}
```

16
filter(uint x in, uint y out, uint weights, uint taps)
{
    uint i;

    // do the first tap, getting input from outside world
    tap(xin, x[0], 0, y[0], weights[taps-1]);

    // define all but the last tap
    for (i=1; i<(taps-1); i++) {
        tap(x[i-1], x[i], y[i-1], y[i], weights[taps-i-1]);
    }

    // finish up with the last tap.
    tap(x[i-1], x[i], y[i-1], yout, weights[0]);
}

// main defines the filter itself
main(fixed<8,0> xin in, uint yout out)
{
    // w has the weights of the taps
    uint w[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
    // x and y are the wires connecting the stages
    uint x[10], y[10];
    uint taps = 5;  // used to determine how many
                   // taps are in the filter

    filter(xin, yout, w, taps);  // instantiate the filter
}

At compilation Dil will optimize away some delays. To test this circuit, a driver function will need to supply input values and then get output values after some delay. Let us assume that the output program is called fir. Then a possible driver function looks like:

#include <iostream.h>
#include "fir.h"

main()
{
    unsigned i;
    unsigned out;

    for (i=0; i<LATENCY_TO_FIRST_OUTPUT; i++)
    {
        fir(i, out); // fill the pipeline
3.4 Automatically generating drivers

We have implemented a perl script which can automatically generate a main C++ function which can be used to exercise the output of the Dil compiler. This script is called gen-driver.pl\(^1\). The way this script is used is depicted in Figure 3.1.

![Diagram](image)

**Figure 3.1**: Automatically generating drivers to exercise the C code generated by Dil.

The script `gen-driver` receives as argument the header file generated by the Dil compiler. The generated file contains:

- A `main()` function which parses the command line and invokes the dil-generated function.
- A sample function to compute the input values fed each clock cycle to the dil-generated function (the values can be read from a text file).

\(^1\)We are currently enhancing this script; the current version doesn’t handle properly the signed/unsigned types.
• A sample function to compute the expected outputs of the dil-generated function. The expected outputs may be read from a text file.

• Code to print and compare the generated output with the sample outputs. The printing is invoked by using the `-v` flag, and the comparison by using the `-c` flag.

3.5 The Expressiveness of the Dil Language

This section is useful in order to understand how to write Dil programs. In particular, for the programmers used to write directly in `cvhasm`, we notice that in `Dil` the meaning of delay operators (<=) is somewhat different from the one in `cvhasm`.

Observe first that `Dil` is only capable of expressing functions, i.e. all its expressions are side-effect free. This is true because you're not allowed to assign twice to a variable, or compute a variable based on itself.\(^\text{2}\) I.e. the following are both illegal:

\[
\begin{align*}
a &= b; \\
a &<i= b+1; \\
\end{align*}
\]

// or

\[
\begin{align*}
a &<i= b; \\
b &<i= a+1; \\
\end{align*}
\]

The reason we disallow cycles is that when mapped to PipeRench they might require feedback from "lower" stripes, which is impossible in this architecture.

You can view thus a `Dil` program as a function on its input values.

It is useful (although not very precise) to think of a `Dil` program as the body of a loop, which is applied by driver program the to each element of an array. So implicitly when you write a `Dil` program which computes \(P(xin, yout)\) you should think at this program like part of the larger program, which applies \(P\) to each element of an array \(xin\). If your driver program in C looks like this:

```c
int out[elements];
in [elements];

for (i=0; i < elements; i++) // loop feeding
    P(in[i], out[i]); // ...this dil program
```

The meaning of a register in a `Dil` program is the following: a 1-unit delay register stores a value which was computed from the inputs of previous clock cycle. So, for instance, instead of writing a program like the following:

```c
for (i=0; i < elements; i++)
    Q(in[i], in[i-1], out[i]);
```

\(^\text{2}\)This will change in future releases of the compiler.
which reads twice the array in, you can state:

```c
for (i=0; i < elements; i++)
    R(in[i], out[i]);
```

where R is the Dil program

```c
main(int input in, int output out)
{
    aux <1= input;
    Q(input, aux, output);
}
```

The advantage of Q is that it reads the input array only once.

### 3.6 What is the Dil Compiler Doing

For the user knowledgeable of the PipeRench architecture and assembly language, here is a brief description of the operations performed by Dil:

- Carries a bunch of optimizations on the given code.
- Breaks the operations on large pieces of data (which don’t fit into a PipeRench stripe) into smaller ones.
- Groups together independent operations which can fit into a stripe.
- Allocates pass-registers (i.e., virtual wires) for each piece of data and PE’s for each operator.
- If necessary time-multiplexes several values in the same register by slowing down the circuit.
- Schedules the resulting stripes; this implicitly inserts delays between two consecutive stripes.
- Generate cvhasm for the resulting circuit.
Chapter 4

Semantics

As mentioned previously, the Dil language has two types of constructs:

- Run-time evaluated (assignment, expressions).
- Compile-time evaluated (if, for, modules).

This chapter describes the semantics of the run-time components of the language. For the compile-time part see the chapter on Dil syntax.

4.1 Guiding principles:

0) There are two basic types: signed and unsigned. Signed values are represented using 2’s complement.

1) The bit-width of a value is arbitrary unless specified by the input program.

2) The conversions are made in such a way to preserve the maximum of information.

3) No run-time overflow checking or signaling is done automatically; (it could however be implemented in the language).

4) For associative operators the final result should be insensitive to the order of evaluation (i.e. intermediate results in the evaluation of an expression cannot overflow). [It would be very nice to prove this; it looks true.]

5) All operators will compute their result with no loss of information.

6) If the meaning of an operator is obvious it is implemented that way (see for instance selections with negative range). If the meaning of an operator is arguable it is not defined (see modulo on negative values).

7) When speaking about “the width of a value” we mean the actual number of bits taken by a wire, i.e. including the “sign” bit.
4.2 Conversions

8) When a value conversion is made, the type is first converted and next the size. i.e.

```c
type1<size1> a;
type2<size2> b;

a = b;
```

has the semantics:

```c
type1<size1> a;
type2<size2> b;
type1<=> c; // arbitrary width: try to avoid loss of information

c = b;
a = c;
```

9) The only conversions which can cause information loss are:

1. from signed to unsigned values of any width.
2. from a wider value to a narrower value.

All such conversions will be properly signaled by warnings. (This is not yet implemented).

10) The conversions with loss of information are carried like follows:

1. Converting a value from unsigned to signed preserves the bitstring of the value.
2. Converting a value to a narrower string truncates topmost bits.

11) The conversions without loss of information are carried like follows:

1. Converting a shorter value into a longer one will sign-extend the value.
2. Converting an unsigned number into a signed one will grow the value with one extra bit for the sign which will be set to 0.

4.3 Accepted input types; output types

12) Arithmetic operators which require operands to have the same type will convert inputs to the most general type before carrying the computation. Those are

```c
+ - <= >= == != < > ?:
```
13) Here is the typing of the DIL operators inputs/result:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>any</td>
</tr>
<tr>
<td>u</td>
<td>unsigned</td>
</tr>
<tr>
<td>s</td>
<td>signed</td>
</tr>
<tr>
<td>&gt;</td>
<td>stronger type (i.e. signed if one of left/right is signed)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>left op.</th>
<th>right op.</th>
<th>result</th>
<th>observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; n=</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>[ , &amp; , ^</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>+=(n)</td>
<td>a</td>
<td>a</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unary -</td>
<td>a</td>
<td>a</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>== , ! = , &lt; , &gt; , &lt;= , &gt;=</td>
<td>a</td>
<td>a</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>, &amp;&amp;</td>
<td>a</td>
<td>a</td>
<td>u</td>
</tr>
<tr>
<td>!</td>
<td>a</td>
<td>a</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>&gt;</td>
<td>third operator is u</td>
</tr>
<tr>
<td>*</td>
<td>a</td>
<td>a</td>
<td>&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>% , /</td>
<td>u</td>
<td>u</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; , &gt;&gt;</td>
<td>a</td>
<td>u</td>
<td>&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr[ :: ]</td>
<td>a</td>
<td>a</td>
<td>&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a , b</td>
<td>a</td>
<td>a</td>
<td>&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4 Widths

14) The following operators will extend the width of the shorter input operand to match the other input.

&   | ^
-   | +
==   | !=  | <   | >   | <=  | >=
?:

15) The following operators have one-bit outputs:

==   | !=  | <   | >   | <=  | >=
&&   | ||
!

16) The following operator has the third input of size 1 bit:
17) All operators will compute their result with no loss of information (the compiler may optimize this, but the semantics has to be preserved). For instance the following operands will (probably) have outputs wider than both inputs:

\[ + - * \ll \]

18) The result of minus (unary and binary) is \textit{always} signed, no matter the input operand sign.

19) Shifts are arithmetic (i.e. on unsigned values they are logical, on signed values they perform sign-extension).

20) Division and modulo operations do not make sense on signed values. Such circumstances should be flagged as warnings. Some operations cannot have signed operands on one side (i.e. shifts).

21) Division (or modulus) by 0, unsigned subtraction of a larger number have undefined values.

22) Selection will sign-extend its input if necessary. Selections with negative ranges (e.g. \texttt{a[3,4]}) will provide no value at all (and should give warnings).

4.5 Examples

23)

\begin{verbatim}
fixed<8,1> x;
fixed<8,0> y;
x = y;
\end{verbatim}

Is computed like:

\begin{verbatim}
xxxxxxxxxx -> Oxxxxxxxxx -> xxxxxxxxx (warning)
\end{verbatim}

24)

\begin{verbatim}
fixed<8,1> x;
fixed<8,0> y;
fixed<8,1> z;

z = x+y;
\end{verbatim}

This is the same as

\begin{verbatim}
z = (fixed<8,1>)(\texttt{(fixed<9,1>)x + (fixed<9,1>)y});
\end{verbatim}
It should again be flagged as a warning.

25) The meaning of the logical operators:

\[ a \&\& b \]

is the same as

\[ (a \neq 0) \& (b \neq 0) \]
Chapter 5

Module Generation in Dil — The Dil Loader

5.1 Introduction

This chapter describes the design and implementation of a library loader for the Dil programming language. This allows the creation of Dil libraries, which can be used for generating code for non-primitive operations (i.e., the ones that cannot be directly implemented by the hardware), for decomposing operations on wide bit values into sizes manageable by the hardware and for implementing high-level modules and parameterized module instantiation.

The current Dil library is not fully debugged; some operators are not implemented properly.

5.1.1 The Dil Compilation Process

The Dil compilation process is depicted in Figure 5.1.

![Figure 5.1: The Dil Compilation Process](image-url)

These are the main steps of the compilation which involve the library:
1. Load an input file.
2. Evaluate the **main** module.
3. Load the library.
4. Extract from the library the modules for the non-primitive operations.
5. “Execute” the module code to synthesize those operations.
6. Replace the non-primitive operations with the corresponding synthesized modules.
7. Run optimizations on the obtained graph.
8. Generate assembly.

The front-end of the compiler is doing three main operations:

**Parsing:** reads the input (text) file and converts it into an abstract syntax tree.

**Evaluation and loading:** one module definition is extracted from the AST (by default this is the module called **main**), and its definition is evaluated, by expanding the **if**, **for** and module instantiation statements.

**Conversion:** the result of the evaluation is another syntax tree, which contains only assignments operations ($< n =$); this is converted to a different data structure (a graph), which is used in all sequent compiler passes.

The user-supplied input program may contain operators which are not directly implementable on the target hardware. To cope with these, the graph is subjected to a series of analyses, including **type** and **width** analysis. The width analysis computes the minimum necessary bit-width (which does not cause information loss) for the involved operations. Afterwards the library loader is invoked to synthesize the operators which cannot be directly implemented.

The compiler reads afterwards the default library file (the order of loading the user input and the library may be changed in the future, or support for several libraries might be added). This library is also a **.Dit** file, which is supposed to contain definitions of modules for all the operators which cannot be synthesized.

Each operator in the input language has a default module associated in the library, which describes how that operator is to be synthesized (the corresponding module names are hardwired in the compiler code). For each non-primitive operator the corresponding module is loaded (e.g. for synthesizing a “*” operator, the “multiplication” module is loaded).

The next section explains how the library is implemented.

### 5.2 The Library

The modules in the library are **parameterized**. For instance, the user input file might contain both a 4-bit and a 6-bit multiplier. The library has to be able to synthesize both efficiently.

To avoid name clashes with the user code, by convention all global symbols of the library and the symbols used for passing information to the library begin with the prefix **lib**.

Information is passed from the compiler to the library loader using **global constants**. By convention a set of global constants describes the environment in which each library module is to be synthesized.

Module instantiation (generation) based on following parameters:
Architectural parameters: of the target architecture. For example, for the (only) current target, the PipeRench architecture, the following variables are exported by the compiler to the library loader (prefixed with `lib`):

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pes_perStripe</td>
<td>Width of a stripe (in processing elements).</td>
</tr>
<tr>
<td>bits_perPe</td>
<td>Number of bits in a processing element.</td>
</tr>
<tr>
<td>passregs_perPe</td>
<td>Number of pass registers available in a stripe.</td>
</tr>
<tr>
<td>number_of_inputs</td>
<td>The number of input busses.</td>
</tr>
<tr>
<td>width_of_input</td>
<td>The width of an input bus.</td>
</tr>
<tr>
<td>number_of_outputs</td>
<td>The number of output global busses.</td>
</tr>
<tr>
<td>width_of_output</td>
<td>The width in bits of an output bus.</td>
</tr>
<tr>
<td>carry_chain_latency</td>
<td>Signal propagation latency of a carry chain between 2 PEs</td>
</tr>
<tr>
<td>pe_to_pe_latency</td>
<td>Propagation latency between two PEs in same stripe.</td>
</tr>
<tr>
<td>latency_I_to_O_through_one_pe_with_carry</td>
<td>Propagation delay from input to output of a PE which is also using the carry chain.</td>
</tr>
<tr>
<td>latency_I_to_O_through_one_pe_no_carry</td>
<td>Propagation delay from input to output of a PE which is not using the carry chain.</td>
</tr>
<tr>
<td>latency_I_to_Cout</td>
<td>Propagation delay from input to the carry output of a PE.</td>
</tr>
<tr>
<td>latency_Cin_to_O</td>
<td>Propagation delay from carry input to the output of a PE.</td>
</tr>
<tr>
<td>latency_Cin_to_Cout</td>
<td>Propagation through one PE carry chain.</td>
</tr>
<tr>
<td>interconnect_latency</td>
<td>Interconnect latency O-EI &amp; either between stripes or in the same stripe.</td>
</tr>
<tr>
<td>xin_to_xout</td>
<td>Delay from xinput to xoutput in a PE.</td>
</tr>
<tr>
<td>max_latency</td>
<td>Maximum allowed combinational latency within a stripe.</td>
</tr>
</tbody>
</table>

Optimization criteria: these are not yet defined. Possible ones would be: for space, for time (clock speed), for latency, etc.

Module name (operation to synthesize): as mentioned, each language operator has a corresponding (hardwired) module name.

Types of input/output variables: when a module is loaded it replaces an operator in the user program operating on values of known sizes; parameters describing these sizes are passed. Here are the parameters exported right now for each input and output of a module:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>The width in bits</td>
</tr>
<tr>
<td>log2_size</td>
<td>Integer log2 of previous value</td>
</tr>
<tr>
<td>has_constant_value</td>
<td>1 if the input is known to have constant value</td>
</tr>
</tbody>
</table>

28
Known compile-time values of I/O parameters.

The library can choose not to synthesize a module by setting the global variable `return_no_graph` to 1. For instance, the library can specify that adders on less than 16 bits should be directly implemented by setting this variable whenever it is asked to synthesize an adder on shorter values.

5.3 Examples

The figures in this part of the paper were generated with an older version of the compiler. The new version will indicate for each wire its width. At the bottom of each operator the output wire of that operator is described like follows: `wireId/sign width`, where

- `wireId` is an integer, which identifies uniquely the wire in the internal data structures of the compiler (useful mainly for debugging the compiler);
- `sign` is “+” for an unsigned magnitude, “-” for an unsigned one and “?” for a wire whose type hasn’t yet been inferred.
- `width` is the wire width, in bits. This value is 0 if the width hasn’t yet been inferred.

5.3.1 Synthesizing Non-Primitive Operations

The first role of the library is to synthesize the non-primitive operations.

Figure 5.2 shows an input program an the associated graph.

```c
typedef fixed<4,0,0> int;

main(int a in, int b in, int c out)
{
    c = a * b;
}
```

Figure 5.2: A non-primitive operation.

Here is a possible definition of the multiplier module in the library. (The current one is slightly more complex.) Notice the references to the global variable `lib_input1.size`, which is a parameter which guides how the module should be instantiated.

```c
multiplication(lib_int a in, lib_int b in, 
        lib_int c out) 
{ 
    lib_int i, partial[lib_input1.size + 1];

    partial[0] = 0;
    for (i=1; i <= lib_input1.size; i=i+1) {
```
\[
\text{partial[i]} = \text{partial[i-1]} + ((\text{b[i-1,i-1]}) \ ? \\
\quad (a \ll i) : 0);
\]

\} \\
c = \text{partial[lib_input1\_size]};
\}

After loading the library and synthesizing the multiplier the previous program will look like in Figure 5.3.

![Figure 5.3: Library synthesized multiplier.](image)

Further compiler simplifications will lead to the graph in Figure 5.4. The addition with zero has been eliminated.

A Constant Multiplier Figure 5.5 shows another example which uses exactly the same multiplier definition, but which invokes the multiplier with a constant input.

After loading the module the result will look very similar to the above non-constant multiplier; however, after simplifications the result is shown in Figure 5.6. (The result is obtained by concatenating zeros to the end of the input and adding two copies, one with 1 zero, and one with 3 zeros; the widths of the zero constants are not apparent in the figure.)

5.3.2 Synthesizing Large Operations

The second purpose of the library is to reduce operations carried on too large data values to operations on values narrow enough to fit into the target hardware (“narrow enough” both in size and in signal propagation time). We will illustrate this functionality of the library with a simple logical operator acting on 64-bit values. We will synthesize it on a fabric which cannot support values larger than 32 bits.
The library code for synthesizing large logical operators is simple: the inputs are split into bit ranges (using the selection [...] operator) which are narrow enough to fit, the logical operation is carried on each range pair, and the results are joined together again.

Here is the input program which strained the library (we concatenate 32-bit input values using the . operator).

typedef fixed<32,0> int;
typedef fixed<64,0> long;

main(int a in, int b in, int d in,
    int c out, int c1 out)
{
    long r;

    r = (b.d) & (a.d) & (a.b);
    c0 = r[63,32];
    c1 = r[31, 0];
}

The initial graph representation of this circuit is shown in Figure 5.7.

![Diagram](image)

Figure 5.7: One operation on large values. 64-bit wires represented by thick lines.

After each large “&” operation is synthesized from the library definition the graph looks like in Figure 5.8.

As you see, although no operator works on large values (except the concatenations and selections), there still are large wires in the design, which would make it unimplementable.

However, the compiler has a special pass which does routing simplification (unlike for instance the constant propagation pass which was exhibited in the constant-multiplier example). After the interconnection simplification pass is run, the above graph reduces to the one in Figure 5.9.

5.3.3 Synthesizing Complex Modules

Finally, the library can be used as repository of user-defined complex modules\(^1\). Here is a possible library fragment, which defines a general-purpose Finite Input Response filter:

```
tap(uint txin in, uint txout out, uint tyin in, uint tyout out, uint w)
```

\(^{1}\)This example is not yet supported by the current implementation of the compiler, due to some technical reasons.
Figure 5.8: Large boolean operations synthesized.

Figure 5.9: The previous graph after simplifications.
{  
  tyout = tyin + txin*w;  
  txout <1= txin;  
}

fir_filter(uint input in, uint output out, uint w, uint size)
{
  uint x[size], y[size], i;
  tap(input, x[0], 0, y[0], w[size-1]);
  for (i=1; i<(size-1); i=i+1) {
    tap(x[i-1], x[i], y[i-1], y[i], w[size-i-1]);
  }
  tap(x[i-1], x[i], y[i-1], output, w[0]);
}

The code fragment which instantiates such a filter could be:

main(int xin in, int xout out)
{
  uint taps = 5, weights[taps] = {3, 4, 5, 6, 7};
  fir_filter(xin, xout, weights, taps);
}

Notice the fact that an entire constant array is passed to the module on invocation as the weights parameter.

The filter before (Figure 5.10) and after (Figure 5.11) the simplification can be compared. Although the figures look similar in complexity, the second one has no multipliers, and can be directly synthesized!

5.3.4 Synthesizing new data types

Not illustrated here is another possible usage of the library: for implementing additional data types, like fixed point or floating point. The way this can be done is obvious using modules.

5.4 Conclusion

DIL is a powerful and versatile enough language to make parameterized modules definition an easy task. The development of the current library was very quick (less than 2 days). The effort is much reduced compared to implementing the operator decomposition directly in the C++ language (i.e. hardwiring it in the compiler). For instance the multiplier above has more than 200 lines of C++ code, but only 12 in the library!

Further explorations will show the limitations and strengths of the approach.
Figure 5.10: A fir filter synthesized from the library.
Figure 5.11: The fir filter after constant propagation.
Appendix A

The Dil Grammar

This is the complete grammar of the Dil language, stripped from the bison parser specification file.

```
program:            decls

decls:             decl
    | decls decl

stmts:             stmts stmt
    | stmt

stmt:              assign_stmt ';'
    | delay_stmt ';
    | function_stmt ';
    | if
    | forStmt
    | vdecl ';

block:             '{' stmts '}'

if_stmt:           IF '(' castexpr ')' block
    | IF '(' castexpr ')' block ELSE block

delay_stmt:        lval '<' CONSTANT '=' castexpr

assign_stmt:       lval '=' castexpr
    | lval '=' var '(' alist ')'

lval:              var
    | var '[' expr ']
```
function_stmt: var (' list ')'

for_stmt: FOR (' assign_stmt ;' castexpr ;' assign_stmt )' block

castexpr: (' type_spec ') condexpr
  | condexpr

condexpr: condexpr COMPARE condexpr
  | condexpr '<' condexpr
  | condexpr '>' condexpr
  | condexpr '?' expr ':' expr
  | expr

expr: expr '+' expr
  | expr '*' expr
  | expr ',' expr
  | expr '&' expr
  | expr '|' expr
  | expr ANDAND expr
  | expr OROR expr
  | expr '-' expr
  | expr '^' expr
  | expr '/' expr
  | expr '%' expr
  | expr LSHIFT expr
  | expr RSHIFT expr
  | unary

unary: '-' primary
  | '!' primary
  | '~' primary
  | PLUSMINUS '(' expr ')' primary
  | select

select: expr '[' expr ',' expr ']'
  | expr '[' expr ']
  | primary

primary: var
  | CONSTANT
  | '(' castexpr ')'}

alist: alist ',' expr
var:       IDENTIFIER

decl:      tdecl ';'
           | fdecl
           | gdecl '

tdecl:     TYPEDEF type_spec IDENTIFIER

gdecl:     type_spec vardecllist

vdecl:     type_spec vardecllist

vardecllist: vardecllist ',' vardecl
                       | vardecl

vardecl:   idorarray
                       | idorarray '=' initializer

initializer: '{' init_list '}'
                       | expr

init_list:  expr
                       | init_list ',' expr

voneddecl:  type_spec idorarray inout_spec

fdecl:      fspec ';'
                       | fspec '{' stmts '}

fspec:      IDENTIFIER '(' decl_io_list ')

idorarray:  IDENTIFIER
                       | IDENTIFIER '[' expr ']

decl_io_list: decl_io_list ',' voneddecl
                       | voneddecl

inout_spec: /* empty */
                       | IN
                       | OUT
type_spec: basic_type width_spec

basic_type:
  FIXED
  | TYPENAME

width_spec: /* empty */
  | '<' width_expr '>
  | '<' width_expr ',' width_expr '>
  | '<' width_expr ',' width_expr ',' width_expr '>

width_expr: ' '*
  | expr