

Sandbox

Megan Fillion
mlf2179

Gabriel Guzman
grg2117

Dimitri Leggas
ddl2133

Abstract

We present **Sandbox**, a hardware description language written for COMS W4115, Programming Languages and Translator, Fall 2017. Our goal was to build a syntactically simple yet powerful HDL that would be useful to students from a computer science background learning about digital systems. Our success in that aim should be determined by the programmer, but at minimum we bolstered our own understanding of such systems.

Contents

1	Introduction	3
1.1	Motivation	3
1.2	Goals	3
2	Language Tutorial	3
2.1	The <code>sandbox</code> Function	3
2.2	The type <code>bit</code>	4
2.3	Variable Declaration and Scope	4
2.4	Function Calls	4
2.5	Building a Circuit	5
3	Language Reference Manual	5
3.1	Lexical Elements	5
3.1.1	Identifiers	5
3.1.2	Keywords	6
3.1.3	Comments	6
3.2	<code>bit</code>	6
3.3	Operators and Lexical Conventions	7
3.3.1	Operators	7
3.3.2	Assign	7
3.3.3	Delimiters	7
3.4	Functions	7
3.4.1	Function Declarations	7
3.4.2	Function Returns	8
3.4.3	Function Calls	8
3.4.4	The <code>sandbox</code> Function	8
3.5	Variables	8
3.5.1	Locals	8
3.5.2	Globals	8
4	Project Plan	9
4.1	Language Barrier	9
4.2	Project Log	9
4.3	Roles and Responsibilities	10
4.4	Software Development Environment	10
4.5	Programming Style Guide	10
4.5.1	Indentation	10
4.5.2	Comments	10
4.5.3	<code>in</code>	11
4.5.4	Naming Conventions	11

5	Architectural Design	11
5.1	Overview	11
5.1.1	Preprocessor	12
5.1.2	Lexical Analyzer	12
5.1.3	Parser	12
5.1.4	Abstract Syntax Tree	12
5.1.5	Semantic Checking	12
5.1.6	Flattening	13
5.1.7	Code Generation	13
5.1.8	Sandbox	13
5.1.9	The <code>tic</code> Function	13
6	Testing	13
7	Lessons Learned	13
7.1	Megan	13
7.2	Gabriel	14
7.3	Dimitri	14
8	Code	15

1 Introduction

1.1 Motivation

Sandbox allows students or other electrical engineering enthusiasts to test out elementary circuits in a programming environment. Our language makes it easy and intuitive to create circuit blocks and then link them together in the desired manner. To simulate hardware construction, we did not include some imperative programming features such as loops in the Sandbox language. Therefore, users will have to link their circuit blocks in a way that creates the loop they desire.

1.2 Goals

Our goal in creating Sandbox was to create a simple and easy to use hardware description language. In some sense, we aimed for Sandbox to combine the functional elements of VHDL and the syntax of Python. We also wanted it to be simple to define circuit block function and then connection them in a visually intuitive and comprehensible manner in the coding environment.

2 Language Tutorial

Sandbox is clear, concise, and intuitive for both software and hardware engineers. But nonetheless, let's walk through the language step by step.

2.1 The sandbox Function

First of all, let's talk about Sandbox's main executable function—**sandbox**. Every .sb program has to contain a sandbox function. It is from here that programmers can call other functions they have created in their program. The inputs and outputs (yes, our functions handle multiple outputs) of **sandbox** are the i/o of the circuit being designed by the coder. Outputs of the circuit are printed to standard out or a file. As with all functions in our language, inputs are not required, but at least one output is. Consider the following simple program representing a full adder written in Sandbox:

```
(bit a, bit b, bit cin) sandbox (bit s, bit c):  
    a ^ b ^ cin -> s  
    (a & b) ^ (cin & (a ^ b)) -> c
```

Here our simple sandbox function takes in 1-bit variables **a**, **b**, and **cin** and returns 1-bit variable **s** and **c**. As you might have inferred, **bit** is a type in Sandbox, and yes, larger variables can be declared.

Assignments in Sandbox are evaluated from left to right meaning that the expression on the left hand side of the assign operator **->** will be assigned to the variable on the right hand side. Here for example, the value of **a** exclusive-or **b** exclusive-or **c** is assigned to the variable **s**.

One of the aspects that makes Sandbox so novel is that its functions can return more than one value without having to merge them into an array. Any function in sandbox will return the variables specified in the output list of the function. The sandbox function will print those outputs out for you.

From the example above, it is clear that Sandbox's syntax is similar to Python's. More specific details are covered later.

2.2 The type bit

Sandbox ultimately only has one type, the bit, but it is very powerful and can represent variables of different lengths. Here's how to declare a 1-bit variable called `a`:

```
bit a
```

Of course, we also allow the definition of multiple bit busses. Some k -bit busses are declared as

```
bit.4 b
bit.4 c
```

`b` can hold any integer between 0 and 15 inclusive, and `c` between 0 and 255 inclusive. Although busses of bits are assigned using integers, individual bits can be accessed (values are labeled in comments):

```
12 -> bit.4 u
u(0)->bit v / v is 0 /
u(1)->bit w / w is 0 /
u(2)->bit x / x is 1 /
u(3)->bit y / y is 1 /
```

We also permit accessing a sub-range of a busses (the upper bound is not inclusive)

```
w(0,3)->bit.3 y / y is 4 /
```

2.3 Variable Declaration and Scope

Sandbox supports both local and global variable declarations. Examples of local variable declaration were seen above. Globals require the label `const`:

```
bit z const
12 -> bit.4 y const
```

Note that only global variables can be declared as constants. The scope of a global variable starts when the global is declared and ends at the end of the program. It can be called in any method but if a local variable were to be declared with the same name, the local variable would be prioritized. Thus Sandbox is a statically scoped language. Each scope is associated with a function, or a block of code, and is denoted by indentation within the function.

2.4 Function Calls

The scope of a function in Sandbox starts from its declaration to the end of the program. This means that a function in Sandbox can be called anywhere in the program after its declaration. The following simple example of a half adder show how to make calls in Sandbox:

```
(bit a, bit b) add (bit sum, bit carry):
  a ^ b -> sum
  a & b -> carry
```

```
() sandbox (bit s, bit c):
  [1, 0] -> [s, c]
```

One should also note that outputs need not be explicitly returned.

2.5 Building a Circuit

Consider the following circuit:

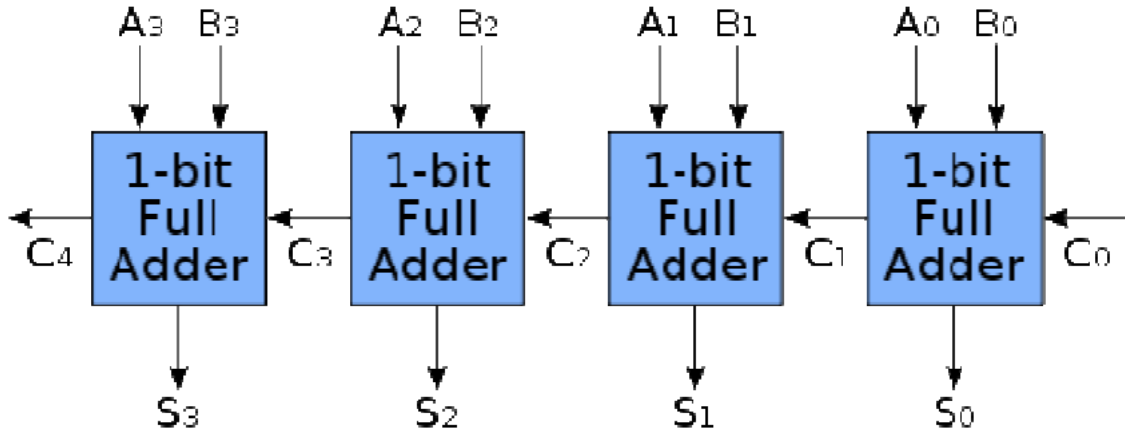


Figure 1: A 4-bit ripple-carry adder

In the code below we have created simulated the 4-bit ripple-carry adder found in Figure 1. The `fulladder` function defines the necessary logical gate operations which are then called from `sandbox`:

```
(bit a, bit b, bit cin) fulladder (bit s, bit c):  
    a ^ b ^ cin -> s  
    ( a & b ) ^ ( cin & ( a ^ b ) ) -> c  
  
(bit a.4, bit b.4, bit cin) sandbox (bit sum.4, bit cout.4):  
    [a(0), b(0), cin ] fulladder [sum(0), cout(0)]  
    [a(1), b(1), cout(0) ] fulladder [sum(1), cout(1)]  
    [a(2), b(2), cout(1) ] fulladder [sum(2), cout(2)]  
    [a(3), b(3), cout(2) ] fulladder [sum(3), cout(3)]
```

3 Language Reference Manual

The reference manual of `sandbox`, a hardware description language for writing circuits in terms of nested circuit blocks, see below.

3.1 Lexical Elements

3.1.1 Identifiers

An identifier is a letter followed by any union of lower- and upper-case letters, numbers, and underscores:

3.1.2 Keywords

The following words are reserved for language-specific use:

bit const

3.1.3 Comments

Comments in Sandbox start and end with `/`. Multiple line comments are accepted but nested comments are not. The following program shows some comments:

```
/ this function, like this comment, does nothing /
( ) doNothing (bit a):
    0 -> a

( ) sandbox (y):
    / look
      how little is
      done /
    [ ] doNothing [y]
```

3.2 bit

The Sandbox language only has one type, but it is quite powerful to say the least. The type `bit` can be used to declare variables of any size the user wants. That is, the `bit` type is used to define k -bit busses. Here's how to declare a 1-bit variable called `a`:

```
bit a
```

Of course, we also allow the definition of multiple bit busses. Some k -bit busses are declared as

```
bit.4 b
bit.4 c
```

`b` can hold any integer between 0 and 15 inclusive, and `c` between 0 and 255 inclusive. Although busses of bits are assigned using integers, individual bits can be accessed (values are labeled in comments):

```
12 -> bit.4 u
u(0)->bit v / v is 0 /
u(1)->bit w / w is 0 /
u(2)->bit x / x is 1 /
u(3)->bit y / y is 1 /
```

We also permit accessing a sub-range of a busses (the upper bound is not inclusive)

```
w(0,3)->bit.3 y / y is 4 /
```

3.3 Operators and Lexical Conventions

3.3.1 Operators

Sandbox supports the following operators:

+	-		&	^	<<	>>
<	>	<=	>=	==	=	!

3.3.2 Assign

Sandbox has two assignment operators. The first `->` is a simple assign used for creating combinational circuits. The second `-:` means to assign on the clock pulse.

3.3.3 Delimiters

The following table describes the delimiters used in Sandbox

,	Commas are necessary for the input and output parameters of a function declaration. Also necessary when calling a function with more than one input/output
:	The colon is used to mark the start of a function body
()	Parentheses are used to delimit inputs and outputs in function declarations and to access sub-busses
[]	Braces are used to delimit input and output parameters in a function call

3.4 Functions

3.4.1 Function Declarations

Functions act as code blocks, meaning they map a list of inputs to a list of outputs. Any function in sandbox will take the following form:

```
(type in_0,...,type in_n) nameOfMethod (type out_0,...,type out_n):  
    stmt1  
    stmt2
```

As seen above, the variables in the parentheses preceding the method name would represent the input argument and the ones in the succeeding parentheses would be the output arguments. The colon is used to denote the start of the function body. Any statement inside a function has to be indented as to show it is still in the scope of the function.

3.4.2 Function Returns

Sandbox, unlike many other languages, supports multiple variable return. This makes circuit designing in Sandbox simpler because circuit blocks tend to have more than one output. Also, Sandbox doesn't have a return keyword—rather, all returns are implicit.

```
(bit a, bit b) add (bit sum, bit carry):  
    a ^ b -> sum  
    a & b -> carry
```

In this add function, both sum and carry are returned from the add function. As long as the variable the user wishes to return is specified in the output parameters of a function, the value will be returned. If a variable is defined in the output parameters, it has to be assigned before the end of the execution of the function.

3.4.3 Function Calls

To call another function declared in a sandbox function, the braces [] must be used to delimit the input and output parameters of a function:

```
(bit a, bit b) sandbox (bit s, bit c):  
    [a, b] add [s, c]
```

3.4.4 The sandbox Function

The sandbox function is the main executable function of any Sandbox program. It must be located at the end of the program as it cannot recognize functions that come after it. That is because the scope of a function starts at its declaration and extends to the end of the program. Therefore, if a function located under the main function and is called in sandbox, the compiler will throw an error.

The outputs of the sandbox function will be the outputs of the program at run time and will either be printed out on the command line or into a text file. The inputs of the sandbox function can either be hard-coded or be taken from the command line at runtime.

If the sandbox function does not contain any outputs, the compiler will throw an error.

3.5 Variables

3.5.1 Locals

Local variables are variables defined in function declarations or inside a program. Their scope is their declaration until the end of the function they were defined in.

3.5.2 Globals

Global variables do exist in Sandbox but they can only be declared as constants (`const`). The scope of a global variable stretches from the point of their declaration until the end of the program. But, if a local variable is declared with the same name as a global variable, the local variable will be prioritized over the global variable:

```

/ Sandbox will output 0 and not 1 /
1 -> bit a const

```

```

() sandbox (bit out):
  0 -> bit a
  a -> out

```

4 Project Plan

4.1 Language Barrier

We were lucky, we came up with a language idea pretty quickly. During our first meeting, Dimitri mentioned how it would be cool to write a programming language where we can build simple digital circuits. From there, we started our work.

Although we believed we were on the right track, our language took a lot of rewriting to get it to the precise syntax we have now. Recall the 4-bit ripple-carry adder defined at the end of Section 2.5. It initially looked very verbose:

```

(int(1) sum, int(1) carry) fulladder (int(1) a, int(1) b, int(1) cin):
  a ^ b ^ cin -> sum
  (a && b) ^ (cin && (a ^ b)) -> carry

(int(4) s, int(1) carry) 4add (int(4) a, int(4) b, int(1) cin):
  FA0, FA1, FA2, FA3 = fulladder
  {a(0), b(0), cin} -> {FA0.a, FA0.b, FA0.cin}
  {a(1), b(1), FA0.carry} -> {FA1.a, FA1.b, FA1.cin}
  {a(2), b(2), FA1.carry} -> {FA2.a, FA2.b, FA2.cin}
  {a(3), b(3), FA2.carry} -> {FA3.a, FA3.b, FA3.cin}
  {FA0.sum, FA1.sum, FA2.sum, FA3.sum, FA3.carry} -> {s(0), s(1), s(2), s(3), carry}

(int(4) x, int(4) y, int(4) z) sandbox (int(4) s, int(1) carry):
  ADDER1, ADDER2 = 4ADD
  {x, y, 0} -> {ADDER1.a, ADDER1.b, ADDER1.cin}
  {ADDER1.s, z, ADDER1.carry} -> {ADDER2.a, ADDER2.b, ADDER2.cin}
  {ADDER2.s, ADDER2.carry} -> {s, carry}

```

4.2 Project Log

The following is a timeline of our project:

9/16	Project proposal
10/10	Language finalized
10/13	AST done
10/17	Scanner done
11/5	Parser done
10/17	Scanner done
11/1	Semantic checking done, flatten in progress
12/1	Code generation done
12/10	Flatten done
12/13	Tic function done
12/18	Regression testing done

4.3 Roles and Responsibilities

We divided the work as follows:

Megan, Gabe, Dimitri	Deciding features, grammar, testing, final report and slides
Gabe	Code repository initialization
Megan	Scanner
Dimitri and Megan	Architecture design, code generation, tic.c
Dimitri	Parser, semantic checking, flatten

4.4 Software Development Environment

Language	Purpose
Ocam 4.06.0	Primary language used for coding Sandbox compiler
Ocamlex	Ocaml lexical analysis language
Ocamlyacc	Ocaml parser language
LLVM 5.0.0	Low level virtual machine, i.e translate Sandbox into byte code
C	Used to write our tic function
Bash	Used for testing

4.5 Programming Style Guide

4.5.1 Indentation

As a group, we all agreed that one indent wins over four spaces any day. The rule was to indent when necessary. But as long as the code was readable and its function was clear, a couple of missing indents here and there was not a problem.

4.5.2 Comments

If the declaration of a function or a variable name isn't completely self-explanatory, a short comment explaining the code's function should be placed above the block. No comments should be longer than three lines on the final project. Long comments explaining a piece of code while the team is working on it is acceptable.

4.5.3 in

The `in` keyword should be used at the end of the line if it's a variable declaration or a short function declaration. If a member is writing a longer function, the `in` should be on the next line as to make debugging simpler.

4.5.4 Naming Conventions

Function and variable names should be descriptive yet short. Other than that, the user of underscore and capitalized letters is fair game. One should avoid using numbers in their variable declarations except if it pertains to the `tic` function or the different states in code generation.

5 Architectural Design

5.1 Overview

This is the overarching structure of the Sandbox compiler:

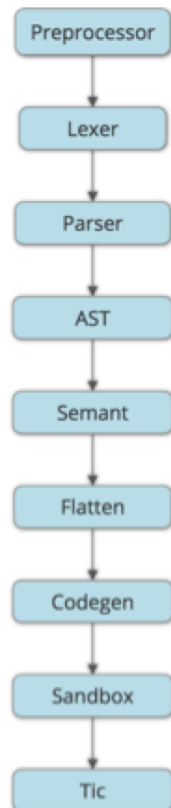


Figure 2: The architecture of the Sandbox compiler

5.1.1 Preprocessor

pre.ml takes in the code from the file passed to the compiler. Here, the preprocessor looks at indentation patterns to mark the start and the end of a function. Also, a semi colon is added at the end of every line to mark the end of a statement. The code that preprocessor produces is located in pre.txt after compiling.

5.1.2 Lexical Analyzer

lexer.mll takes in the code produced by pre.ml and tokenizes the input into lexemes. Lexer excludes comments and makes sure no variables are have the name of one of the “reserved words”. Although not all these words are defined by sandbox, we make sure variables don?t take the name of generic library functions as to prevent errors in later compiling phases. The lexer is compiled using ocamllex.

5.1.3 Parser

parser.mly takes in the tokens from Lexer and creates a tree. The parser defines the grammar and structures of the Sandbox language. parser.mly is compiled with ocaml yacc.

5.1.4 Abstract Syntax Tree

ast.ml acts as an interface between the lexer and parser and the semantic checking portion of the compiler. The ast recursively builds the tree passed from the parser and checks that the types of the arguments are correct.

5.1.5 Semantic Checking

semant.ml is where most of the output errors will originate from. The semantic checker flips through the tree produced by the AST and checks types, sizes, assignments, etc. Here is a list of errors the semantic checker catches:

- Unmatched operand sizes
- Duplicate global variable
- Global initiation
- Duplicate function name
- Duplicate in/out/local bus
- Use of undeclared identifier
- Illegal bus assignment
- Incorrect dereference
- No sandbox function
- Argument mismatch
- Return value never assigned

5.1.6 Flattening

flat.ml collapses our a program written in Sandbox into a single function that gives a list of outputs in terms of logical operations on inputs. Flatten performs a recursive walk through function calls to produce post-order expressions for the outputs of the **sandbox**. The list of operations and literals is passed to the code generation stage.

5.1.7 Code Generation

codegen.ml takes the sequence of literals and operations from the flatten stage and begins pushing them onto a stack. If an operation is encountered, the correct number of literals or expressions are popped and the necessary LLVM instruction is built, then the resulting expression is pushed back onto the stack. Because of the flattening stage, all of the instructions are built in a single function.

In order to keep track of states of values assigned on the clock pulse, for such variables codegen.ml creates two internal globals (in LLVM these are static variables) one for state 0 and one for state 1. If in state 0, it loads variable from 0 and stores into 1 and if in state 1, it loads from 1 and stores into 0. The **tic** function keeps track of the current state.

In order to have multiple returns, the function built in LLVM takes two pointers, one to inputs, and one to outputs. At the beginning of the function the values are loaded from input, and at the end the results are stored in the output array.

5.1.8 Sandbox

The sandbox.ml file calls all the functions necessary to compile the program.

5.1.9 The tic Function

tic is our only function written in C. It calls the function written in code generation, resulting in one loop through the circuit. Tic prints out the outputs of the file at each clock pulse.

6 Testing

To run our series of semantic checks and tests that pass, type:

```
> make
> ./testall.sh
```

7 Lessons Learned

7.1 Megan

Although this sounds generic, the best advice I can give is start as early as possible. Everything is going to take longer than you think and Ocaml and LLVM cannot be learnt in one day. In parallel, pick a project that you think your group can handle. Although it's nice to reach for the stars sometimes, building a compiler is going to be more complex than you think so start simple. If need be, you can always add to your language. Also, if you're dealing with a particularly hard part of your compiler, be sure to sit down and map out the architecture of your design with your

teammates. You can't believe how much easier it is to write your programs when you have a detailed outline in front you. Also, you're more likely to catch potential bugs in your program if you talk it out with your teammates beforehand.

7.2 Gabriel

Representation of objects/datatypes in a programming language needs to be explicitly drawn out and discussed before being tested. When I was trying to implement multiple bit busses I had an extremely difficult time because I had not "mapped" the representation in a diagram of the AST. In previous programming projects I had been able to get away with "figuring it out as I went," that was the case when I was trying to design types in our language.

I also learned how essential correctly configuring an environment is for developing projects like this. I spent over a week trying to set up ocaml-llvm on my Windows version of Ubuntu before resorting to the VM provided by Prof. Edwards at the start of the semester. The time I spent setting up my environment and making it usable took up much more time than it needed to, and I regret not speaking to a professor/ta to fix the issue sooner.

7.3 Dimitri

"Maybe I'm a masochist, but I really loved learning Ocaml. I'll probably use this more than I should from now on." I said this to Megan one night probably around 3 A.M., and it was as true now as it was then. In data structures, where I got a taste of Ocaml, I found it completely daunting and impossible. This class put me a little out of my comfort zone and exposed me to a new way of thinking about programming.

I learned that effective communication within a group is really hard. People do not always understand what you mean and you do not always get what they mean. Making sure everyone is on the same page conceptually and in terms of what needs to be done would have made things much easier. Starting **EARLY** would be the key to achieving this.

8 Code

```
1
2 (*
3  assumes the the .sb file is formatted correctly
4  keeps track of indentation to form blocks
5  adds semi-colons
6  will not keep track if inside comments
7 *)
8
9 open Printf
10
11 let process ic =
12   let out_file = "pre.txt" in
13   let oc = open_out out_file in
14   let rec read_lines l =
15     try read_lines ((input_line ic)::l)
16     with End_of_file -> close_in ic; l
17   in
18
19   let process_line t l =
20     if l = "" then t
21     else if l.[String.length l - 1] = '/' then (fprintf oc "%s\n" l; t)
22     else if l.[0] = '\t' then (fprintf oc "%s;\n" l; l)
23     else (
24       let tokens = String.split_on_char ' ' l
25       and opt = if t = 1 then "}\n" else "" in
26       match List.rev tokens with
27       | [] -> fprintf oc ""; t (* never matched *)
28       | hd::tl -> let last = hd and others = List.rev tl in
29         let ll = String.length last in
30         if ll > 0 then
31           (* function header *)
32           if last.[ll-1] = ':' then (
33             fprintf oc "%s%s %s{\n"
34             opt
35             (String.concat " " others)
36             (String.sub last 0 (ll-1)); 1)
37           (* then global decl *)
38           else (fprintf oc "%s%s;\n" opt l; 0)
39         else (fprintf oc "%s%s;\n" opt l; 0)
40     )
41   in let lines = List.rev(read_lines [])
42   in ignore(List.fold_left process_line 0 lines);
43   fprintf oc "}\n";
44   close_out_noerr oc;
45   open_in out_file
```

```

1  (* Abstract Syntax Tree *)
2
3  type op = Add | Sub | Lt | Gt | Lte | Gte | Eq | Neq |
4          Or | And | Xor | Shl | Shr
5
6  type uop = Not | Umin
7
8  type asn = Asn | Casn
9
10 type expr =
11   | Num of int
12   | Id of string
13   | Subbus of string * int * int
14   | Unop of uop * expr
15   | Binop of expr * op * expr
16   | Basn of expr * asn * string
17   | Subasn of expr * asn * string * int * int
18
19 type stmt =
20   | Expr of expr
21   | Call of expr list * string * expr list
22
23 type bus = { name : string; size : int; init : expr; isAsn : bool array }
24
25 type gdecl = Const of int * bus
26
27 type vdecl =
28   | Bdecl of bus
29   (* | Adecl of bus * int *)
30
31 type fbody = vdecl list * stmt list
32
33 type fdecl = {
34   portin : bus list;
35   fname  : string;
36   portout : bus list;
37   body   : fbody;
38 }
39
40 type program = gdecl list * fdecl list
41
42 (* Pretty-printing functions *)
43
44 let string_of_op = function
45   | Add -> "+"
46   | Sub -> "-"
47   | Lt  -> "<"
48   | Gt  -> ">"
49   | Lte -> "<="

```

```

50 | Gte -> ">="
51 | Eq  -> "=="
52 | Neq -> "!="
53 | Or  -> "|"
54 | And -> "&"
55 | Xor -> "^"
56 | Shl -> "<<"
57 | Shr -> ">>"
58
59 let string_of_uop = function
60 | Not -> "!"
61 | Umin -> "-"
62
63 let string_of_asn = function
64 | Asn -> "->"
65 | Casn -> "-:"
66
67 let rec string_of_expr = function
68 | Num(l) -> string_of_int l
69 | Id(s) -> s
70 | Subbus(n, i1, i2) ->
71   n ^ "(" ^
72   (if i2=i1+1 then string_of_int i1
73     else string_of_int i1 ^ ":" ^ string_of_int (i2-1))
74   ^ ")"
75 | Unop(o, e) -> string_of_uop o ^ string_of_expr e
76 | Binop(e1, o, e2) ->
77   string_of_expr e1 ^ " " ^
78   string_of_op o ^ " " ^
79   string_of_expr e2
80 | Basn(e, a, n) ->
81   string_of_expr e ^ " " ^
82   string_of_asn a ^ " " ^
83   n
84 | Subasn(e, a, n, i1, i2) ->
85   string_of_expr e ^ " " ^
86   string_of_asn a ^ " " ^
87   string_of_expr (Subbus(n, i1, i2))
88
89 let rec string_of_stmt = function
90 | Expr(expr) -> string_of_expr expr ^ "\n"
91 | Call(il, f, ol) ->
92   "[" ^
93   String.concat ", " (List.map string_of_expr il) ^ "]" ^
94   f ^ " [" ^
95   String.concat ", " (List.map string_of_expr ol) ^ "]"
96
97 let string_of_bus bus =
98   string_of_expr bus.init ^ "-> " ^
99   "bit" ^ (if bus.size = 1 then "" else "." ^ string_of_int bus.size) ^

```

```

100     " " ^ bus.name
101
102 let string_of_vdecl v = match v with
103   | Bdecl bus -> string_of_bus bus
104
105 let string_of_gdecl v = match v with
106   | Const(bus, s) -> string_of_bus bus ^ " const"
107
108 let string_of_fdecl fdecl =
109   String.concat ", " (List.map (fun b -> b.name) fdecl.portin) ^ " " ^
110   fdecl.fname ^
111   String.concat ", " (List.map (fun b -> b.name) fdecl.portout) ^ ":\n\t" ^
112   String.concat "\n\t" (List.map string_of_vdecl (fst fdecl.body)) ^ "\n\t" ^
113   String.concat "\t" (List.map string_of_stmt (snd fdecl.body))
114
115 let string_of_program (vars, funcs) =
116   String.concat "\n" (List.map string_of_gdecl (List.rev vars)) ^ "\n" ^
117   String.concat "\n" (List.map string_of_fdecl funcs)

```

```

1  (* Lexical analyzer *)
2
3  { open Parser }
4
5  rule token = parse
6    |[' ' '\t' '\r' '\n'] {token lexbuf} (* eat whitespace *)
7
8    (* binary operators *)
9    | '+'          { PLUS }
10   | '-'          { MINUS }
11   | '|'          { OR }
12   | '&'          { AND }
13   | '^'          { XOR }
14   | "<<"         { SHL }
15   | ">>"         { SHR }
16   | '<'          { LT }
17   | '>'          { GT }
18   | "<="         { LTE }
19   | ">="         { GTE }
20   | "=="         { EQ }
21   | "!="         { NEQ }
22
23   (* unary operator *)
24   | '!'          { NOT }
25
26   (* other operators *)
27   | "::"         { CAT }
28   | '.'          { DOT }
29
30   (* delimiters *)
31   | ','          { COMMA }
32   | ';'          { SEMI }
33   | ':'          { COLON }
34
35   (* assignments *)
36   | "->"         { ASSIGN }
37   | "-:"         { CLKASN }
38
39   (* scoping *)
40   | '('          { LPAREN }
41   | ')'          { RPAREN }
42   | '['          { LBRACK }
43   | ']'          { RBRACK }
44   | '{'          { LBRACE }
45   | '}'          { RBRACE }
46
47   (* key words *)
48   | "bit"        { BIT }
49

```

```

50  (* integer and string literals *)
51  | ['0'-'9']+ as n { NUM(int_of_string n) }
52  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as i { ID(i) }
53
54  (* Comments, unrecognized, and EOF *)
55  | '#'           {comment lexbuf}
56  | eof           { EOF }
57  | _             { raise (Failure("illegal character")) }
58
59  and comment = parse
60  | '\n'          {token lexbuf}
61  | eof           { raise (Failure("comment started but never finished")) }
62  | _             {comment lexbuf}

```

```

1  /* parser for sandbox */
2
3  %{ open Ast %}
4
5  /* tokens */
6  %token PLUS MINUS OR AND XOR SHL SHR
7  %token LT GT LTE GTE EQ NEQ
8  %token NOT
9  %token ASSIGN CLKASN WIRE
10 %token COMMA SEMI COLON CAT DOT
11 %token BIT
12 %token LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE
13 %token <int> NUM
14 %token <string> ID
15 %token EOF
16
17 /* precedence */
18 %left COMMA SEMI
19 %right ASSIGN CLKASN
20 %left EQ NEQ
21 %left LT GT LTE GTE
22 %left PLUS MINUS
23 %left OR
24 %left XOR
25 %left AND
26 %left SHL SHR
27 %right NOT UMIN
28
29 %start program
30 %type <Ast.program> program
31
32 %%
33
34 program: decls EOF { $1 }
35
36 decls:
37   | /* nothing */ { [], [] }
38   | decls gdecl { ($2 :: fst $1), snd $1 }
39   | decls fdecl { fst $1, ($2 :: snd $1) }
40
41 gdecl:
42   | bdecl SEMI { Const($1, $1.init) }
43
44 bdecl:
45   | init_opt ID size_opt
46     { {
47       name = $2;
48       size = $3;
49       init = $1;

```

```

50         isAsn = Array.make $3 false
51     } }
52
53 init_opt:
54     | /* nothing */      { Num 0 }
55     | expr assign        { $1 }
56
57 size_opt:
58     | /* nothing */      { 1 }
59     | DOT NUM            { $2 }
60
61
62 fdecl:
63     LPAREN port RPAREN ID LPAREN port_out RPAREN
64     LBRACE fbody RBRACE
65     { {
66         portin = $2;
67         fname  = $4;
68         portout = $6;
69         body    = List.rev (fst $9), List.rev (snd $9);
70     } }
71
72 port:
73     | /* nothing */      { [] }
74     | busses              { List.rev $1 }
75
76 port_out:
77     | /* nothing */      { raise (Failure("Empty output port list")) }
78     | busses              { List.rev $1 }
79
80 busses:
81     | bdecl                { [$1] }
82     | busses COMMA bdecl   { $3 :: $1 }
83
84 fbody:
85     | /* nothing */      { [], [] }
86     | fbody local        { ($2 :: fst $1), snd $1 }
87     | fbody stmt         { fst $1, ($2 :: snd $1) }
88
89 local:
90     | vdecl SEMI          { $1 }
91
92 vdecl:
93     | bdecl                { Bdecl($1) }
94
95 stmt:
96     | asnexpr SEMI        { Expr $1 }
97     | LBRACK actuals RBRACK ID LBRACK actuals RBRACK SEMI
98         { Call($2, $4, $6) }
99

```

```

100 asnexpr:
101   | expr assign ID { Basn($1, $2, $3) }
102   | expr assign ID LPAREN NUM COLON NUM RPAREN
103     { Subasn($1, $2, $3, $5, $7) }
104   | expr assign ID LPAREN NUM RPAREN
105     { Subasn($1, $2, $3, $5, $5+1) }
106
107 assign:
108   | ASSIGN          { Asn }
109   | CLKASN          { Casn }
110
111 actuals:
112   | /* nothing */   { [] }
113   | actual_list     { List.rev $1 }
114
115 actual_list:
116   | expr            { [$1] }
117   | actual_list COMMA expr { $3 :: $1 }
118
119 expr:
120   | NUM              { Num($1) }
121   | ID               { Id($1) }
122   | ID LPAREN NUM COLON NUM RPAREN { Subbus($1, $3, $5) }
123   | ID LPAREN NUM RPAREN           { Subbus($1, $3, $3+1) }
124   /* ADD CAT */
125   | MINUS expr %prec UMIN { Unop(Umin, $2) }
126   | NOT  expr %prec NOT   { Unop(Not, $2) }
127   | expr PLUS expr { Binop($1, Add, $3) }
128   | expr MINUS expr { Binop($1, Sub, $3) }
129   | expr EQ  expr { Binop($1, Eq, $3) }
130   | expr NEQ expr { Binop($1, Neq, $3) }
131   | expr LT  expr { Binop($1, Lt, $3) }
132   | expr LTE expr { Binop($1, Lte, $3) }
133   | expr GT  expr { Binop($1, Gt, $3) }
134   | expr GTE expr { Binop($1, Gte, $3) }
135   | expr AND expr { Binop($1, And, $3) }
136   | expr OR  expr { Binop($1, Or, $3) }
137   | expr XOR expr { Binop($1, Xor, $3) }
138   | expr SHL expr { Binop($1, Shl, $3) }
139   | expr SHR expr { Binop($1, Shr, $3) }
140   | LPAREN expr RPAREN { $2 }

```

```

1  (* Semantic Checking *)
2
3  open Ast
4
5  module StringMap = Map.Make(String)
6
7  (** HELPER FUNCTIONS **)
8
9  (* raise failure if duplicates exist *)
10 let report_duplicate exceptf list =
11     let rec helper = function
12         | n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
13         | _ :: t -> helper t
14         | [] -> ()
15     in helper (List.sort compare list)
16
17 (* give underlying bus of declarations *)
18 let gdec2b d = match d with Const(b, s) -> b
19 let vdec2b d = match d with Bdecl b -> b
20
21 (* number of bits required to describe int x *)
22 let bit_required x =
23     (* for the moment assuming x > 0 tho *)
24     let x = abs x
25     in let log2 y =
26         int_of_float ( ((log (float_of_int y)) /. (log 2.)) )
27     in (log2 x) + 1
28
29 (* raise failure if some element not equal to another *)
30 let all_eq l =
31     let rec diff d = function
32         | [] | [_] -> d
33         | hd::tl -> diff ((hd - List.nth tl 0)::d) tl
34     in let diffs = diff [] l
35     in if not (List.for_all (fun x -> x = 0) diffs) then
36         raise (Failure("invalid arguments")) else ()
37
38 (* number of bits required for result of binop *)
39 let binop_size s1 op s2 = match op with
40     | And | Or | Xor -> if s1 != s2
41         then raise(Failure("operand sizes do not match "))
42         else s1
43     | Add | Sub -> Pervasives.max s1 s2
44     | Shl | Shr -> s1
45     | Lt | Gt | Lte | Gte | Eq | Neq -> 1
46
47 (** CHECK THAT THE AST IS SEMANTICALLY CORRECT **)
48
49 (* function for checking a single assign *)

```

```

50 (* use x and y to be usable for subbus *)
51 let check_basn e es b x y = ( match e with
52   | Num _ -> if es > y-x then raise(Failure("size mismatch in " ^ b.name))
53   | Id _ | Subbus(_,_,_) | Unop(_,_) | Binop(_,_,_) -> if es != y-x
54     then raise(Failure("size mismatch in " ^ b.name)) else ()
55   | _ -> raise (Failure("illegal bus assignment: " ^ b.name)) );
56 for i = x to y-1 do if b.isAsn.(i)
57   then raise (Failure("bus " ^ b.name ^ " has more than one driver"))
58   else b.isAsn.(i) <- true
59 done
60
61
62 (* check if valid subbus *)
63 let check_subbus b x y =
64   if x >= 0 && y <= b.size && x < y then ()
65   else raise(Failure("incorrect dereference of " ^ b.name))
66
67 let check_subasn e es b x y =
68   check_subbus b x y;
69   check_basn e es b x y
70
71 (* main checking function *)
72 let check (globaldecls, functions) =
73   (* checking globals *)
74   let globals = List.map gdec2b globaldecls in
75   (* no duplicate globals *)
76   report_duplicate (fun n -> "duplicate global variable " ^ n)
77   (List.map (fun g -> g.name) globals);
78   (* globals intialized to an int *)
79   let check_global_init g = match g.init with
80     | Num _ -> ()
81     | _ -> raise (Failure ("global " ^ g.name ^ " must be initialized to an integer"))
82   in List.iter check_global_init globals;
83
84   (* checking functions *)
85   (* no duplicate functions *)
86   report_duplicate (fun n -> "duplicate function " ^ n)
87   (List.map (fun fd -> fd.fname) functions);
88
89   (* collect declared functions *)
90   let function_decls = List.fold_left
91     (fun m fd -> StringMap.add fd.fname fd m) StringMap.empty functions
92   in
93   let function_decl s = try StringMap.find s function_decls
94     with Not_found -> raise (Failure ("no function " ^ s))
95   in
96   (* ensure that sandbox defined *)
97   let _ = function_decl "sandbox"
98   in
99   (* check each function decl *)

```

```

100 let check_function func =
101   (* ensure no conflict between portin/portout/locals *)
102   let locals = func.portin @ func.portout @ (List.map vdec2b (fst func.body))
103   in report_duplicate
104   (fun n -> "duplicate in/out/local bus " ^ n ^ " in " ^ func.fname)
105   (List.map (fun b -> b.name) locals);
106   (* build symbol table for all busses visible in function *)
107   let symbols = List.fold_left (fun m b -> StringMap.add b.name b m)
108   StringMap.empty (globals @ locals)
109   in
110   let lookup s =
111     try StringMap.find s symbols
112     with Not_found -> raise (Failure ("undeclared identifier " ^ s))
113   in
114
115   (* need to ensure all outputs are assigned *)
116   let out_names = List.map (fun b -> b.name) func.portout in
117   let out_table = Hashtbl.create (2 * List.length out_names) in
118   List.iter (fun out -> Hashtbl.add out_table out false) out_names;
119
120   let check_const n =
121     if List.mem n (List.map (fun b->b.name) globals) &&
122     not (List.mem n (List.map (fun b->b.name) globals))
123     then raise(Failure("cannot change const")) else ()
124   in
125   (* returns number of bits required for expression *)
126   let rec expr = function
127     | Num x -> bit_required x
128     | Id s -> (lookup s).size
129     | Subbus(n, i1, i2) -> let b = lookup n in
130       check_subbus b i1 i2; i2-i1
131     | Unop(op, e) -> expr e
132     | Binop(e1, op, e2) -> let s1 = expr e1 and s2 = expr e2 in
133       binop_size s1 op s2
134     | Basn(e, a, n) -> let s = expr e and b = lookup n in
135       check_const n;
136       check_basn e s b 0 b.size;
137       if List.mem n out_names then Hashtbl.replace out_table n true;
138       b.size
139     | Subasn(e, a, n, i1, i2) -> let s = expr e and b = lookup n in
140       check_const n;
141       check_subasn e s b i1 i2;
142       if List.mem n out_names then Hashtbl.replace out_table n true;
143       i2-i1
144   in
145   (* returns unit if semantically valid *)
146   let rec stmt = function
147     | Expr e -> ignore(expr e)
148     | Call(inputs, n, outputs) -> let fd = function_decl n in
149       (* can only assign to busses and subbuses *)

```

```

150 List.iter (fun out -> match out with
151   | Id _ | Subbus(_,_,_) -> ()
152   | _ -> raise(Failure("only bus or subbus can be an output"))
153 ) outputs;
154 (* calls to sandbox are not permitted *)
155 if fd.fname = "sandbox" then raise(Failure("cannot call sandbox"))
156 (* do number of actuals/outputs match portin/portout *)
157 else if (List.length inputs) != (List.length fd.portin)
158   then raise(Failure("input mismatch in " ^ n))
159 else if (List.length fd.portout) != (List.length outputs)
160   then raise(Failure("output mismatch in " ^ n))
161 (* can inputs fit in portin and outputs in portout *)
162 (* accounts for shorthand function calls... *)
163 else
164   let check_ports acts port =
165     List.iter2 (fun x y -> if not (x mod y = 0) then
166       raise (Failure("invalid arguments")) else ())
167     acts port;
168     let quo = List.map2 (fun x y -> x / y) acts port
169     in all_eq quo
170   in
171   check_ports
172     (List.map expr inputs) (List.map (fun b -> b.size) fd.portin);
173   check_ports
174     (List.map expr outputs) (List.map (fun b -> b.size) fd.portout);
175
176   (* sizes accounted for, but can outputs be assigned? *)
177   let check_outasn o = match o with
178     | Id s -> let out = lookup s in
179       check_basn (Id "dummy") out.size out 0 out.size;
180       if List.mem out.name out_names
181         then Hashtbl.replace out_table out.name true
182     | Subbus(n, i1, i2) -> let out = lookup n in
183       check_subasn (Id "dummy") (i2-i1) out i1 i2;
184       if List.mem out.name out_names
185         then Hashtbl.replace out_table out.name true
186     | _ -> raise (Failure(n ^ " cannot port to these outputs "))
187   in List.iter check_outasn outputs
188
189   (* check each statement and that all outputs are assigned *)
190   in List.iter stmt (snd func.body);
191   Hashtbl.iter (fun n x -> if x then ()
192     else raise(Failure("not all outputs of " ^ func.fname ^ " assigned")))
193   out_table
194   in
195   List.iter check_function functions
196
197 (* BREAK UP BUSSES INTO BITS *)
198 (* THE CODE BELOW HERE CAUSES NO ERRORS BUT IS NOT
199   COMPLETE. THE INTENT WAS TO RETURN A MODIFIED AST

```

```

200     WHERE EVERYTHING HAS BEEN BROKEN INTO BITS
201 *)
202
203 (* convert decimal number to list of bits of given len *)
204 let d2b x len =
205     let rec dec2bin y lst = match y with
206     | 0 -> (List.rev lst)
207     | @ (Array.to_list (Array.make (len - List.length lst) 0))
208     | _ -> dec2bin (y / 2) ((y mod 2)::lst)
209     in dec2bin x []
210
211 (* convert name into list of bit names *)
212 let n2b n i1 i2 =
213     let rec name2bits n i lst =
214         if i = i1 then (n ^ "_" ^ (string_of_int i1))::lst
215         else name2bits n (i-1) ((n ^ "_" ^ (string_of_int i))::lst)
216     in name2bits n (i2-1) []
217
218 (* break unops up into bitwise unops *)
219 let break_unop uop ex = match uop with
220 | Not -> []
221 | Umin -> []
222
223 (* break binops up into bitwise binops *)
224 let break_binop e1 op e2 = []
225
226 (* function for breaking a single assign *)
227 (* use x and y to be usable for subbus *)
228 let break_basn e a b x y = ( match e with
229 | Num v -> List.map2 (fun q p -> Basn(Num q, a, p))
230 | (d2b v y) (n2b b.name x y)
231 | Id s -> List.map2 (fun q p -> Basn(Id q, a, p))
232 | (n2b s 0 y) (n2b b.name x y)
233 | Subbus(n,i1,i2) -> List.map2 (fun q p -> Basn(Id q, a, p))
234 | (n2b n i1 i2) (n2b b.name x y)
235 | Unop(uo,e1) -> []
236 | Binop(e1,o,e2) -> []
237 | _ -> raise (Failure("never reached")) )
238
239 let break_busses gb =
240     let binit = match gb.init with Num x -> x | _ -> 0 in
241     let vals = d2b binit gb.size in
242     let nams = n2b gb.name 0 gb.size in
243     List.map2 (fun n v ->
244         {
245             name = n;
246             size = 1;
247             init = Num v;
248             isAsn = [| false |]
249         }

```

```

250     ) nams vals
251
252 let break (globaldecls, functions) =
253     (* break up globals *)
254     let globals = List.map gdec2b globaldecls in
255     let broken_globals = List.concat (List.map break_busses globals) in
256
257     (* collect functions *)
258     (* let function_decls = List.fold_left
259        (fun m fd -> StringMap.add fd.fname fd m) StringMap.empty functions
260     in
261     let function_decl s = StringMap.find s function_decls
262     in *)
263     (* check each function decl *)
264     let break_function func =
265         (* ensure no conflict between portin/portout/locals *)
266         let locals = func.portin @ func.portout @ (List.map vdec2b (fst func.body)) in
267         (* let broken_locals = List.concat (List.map break_busses locals) in *)
268
269         (* build symbol table for all busses visible in function *)
270         let symbols = List.fold_left (fun m b -> StringMap.add b.name b m)
271         StringMap.empty (globals @ locals)
272         in let lookup s = StringMap.find s symbols
273         in
274         (* list of stmts on bits *)
275         let rec break_stmt = function
276             | Expr e -> (match e with
277                 | Basn(ex,a,nb) -> let b = lookup nb in
278                     break_basn ex a b 0 b.size
279                 | Subasn(ex,a,nb,i1,i2) -> let b = lookup nb in
280                     break_basn ex a b i1 i2
281                 | _ -> []
282             )
283             | Call(inputs, n, outputs) -> [] (* let fd = function_decl n in [] *)
284
285         (* check each statement and that all outputs are assigned *)
286         in List.map break_stmt (snd func.body)
287
288     in
289     broken_globals, List.map break_function functions

```

```

1
2 (* Flattening stage *)
3
4 open Ast
5
6 type node =
7   | Val of int
8   | Var of string
9   | Uo of uop
10  | Op of op
11  | As of asn
12
13 module StringMap = Map.Make(String)
14
15 let flatten (globaldecls, functions) =
16   (* globals busses *)
17   let g2b d = match d with Const(b, s) -> b in
18   let globals = List.map g2b globaldecls in
19   (* let global_names = List.map (fun b -> b.name) globals in *)
20   let globes =
21     List.map
22       (fun g -> match g.init with
23         | Num x -> g.name, x
24         | _ -> raise(Failure("never reached"))) )
25     globals
26   in
27   (* table keeping track of variable names *)
28   let var_table = Hashtbl.create 100 in
29
30   (* function declarations *)
31   let function_decls = List.fold_left
32     (fun m fd -> StringMap.add fd.fname fd m) StringMap.empty functions
33   in
34   let func_lookup n = StringMap.find n function_decls in
35   let sbd = func_lookup "sandbox"
36   in
37   let rec f2g f inexpr outexpr =
38     (* local busses *)
39     let d2b d = match d with Bdecl b -> b in
40     let ldec = List.map d2b (fst f.body) in
41     let locals = f.portin @ f.portout @ ldec in
42     (* for keeping track of naming *)
43     let track_name b = let n = b.name in
44       if Hashtbl.mem var_table n
45       then Hashtbl.replace var_table n (Hashtbl.find var_table n + 1)
46       else Hashtbl.add var_table n 0
47     in List.iter track_name locals;
48     let get_local n =
49       n ^ "_" ^ string_of_int (Hashtbl.find var_table n) in

```

```

50     (* all available busses *)
51 let loces =
52     List.concat
53     (List.map
54     (fun l -> match l.init with
55         | Num x -> [Val x; Var(get_local l.name); As(Asn)]
56         | _ -> raise(Failure("never reached")) )
57     ldec)
58 in
59     (* symbols only contains locals, globals in globals *)
60 let symbols = List.fold_left (fun m b -> StringMap.add b.name b m)
61     StringMap.empty locals
62 in
63     (* mapping formals to actuals *)
64 let f2a =
65     let formals = List.map (fun b -> b.name) f.portin in
66     List.fold_left2 (fun m f a -> StringMap.add f a m)
67     StringMap.empty formals inexpr
68 in
69     (* mapping formals to outputs *)
70 let f2o =
71     let formals = List.map (fun b -> b.name) f.portout in
72     List.fold_left2 (fun m f o -> StringMap.add f o m)
73     StringMap.empty formals outexpr
74 in
75 let rec expr2g = function
76     | Num i -> [Val i]
77     | Id s -> (* is it a local? *)
78         if StringMap.mem s symbols then
79             (if StringMap.mem s f2a
80              then [Var(StringMap.find s f2a)] (* is it a portin? *)
81              else [Var(get_local s)]) (* or just a normal local? *)
82         else [Var s] (* or a global *)
83     | Unop(o, e) -> (expr2g e) @ [Uo o]
84     | Subbus(n,e1,e2) -> []
85     | Binop(e1, o, e2) -> (expr2g e1) @ (expr2g e2) @ [Op o]
86     | Basn(e, a, n) -> let store =
87         (if StringMap.mem n f2o
88          then [Var(StringMap.find n f2o)] (* is it a portout? *)
89          else [Var(get_local n)]) (* or just a normal local? *)
90         in (expr2g e) @ store @ [As a]
91     | Subasn(e, a, n, i1, i2) -> []
92 in
93 let rec stmt2g g = function
94     | Expr e -> g @ (expr2g e)
95     | Call(ins, fn, outs) ->
96         (* let x = List.concat (List.map expr2g ins)
97         and y = List.concat (List.map expr2g outs) *)
98         let x = List.map
99         (fun a -> match a with Id s -> get_local s | _ -> raise(Failure("not handled

```



```

100         yet"))))
101     ins
102     and y = List.map
103         (fun a -> match a with Id s -> get_local s | _ -> raise(Failure("not handled
104         yet"))))
105     outs
106     in g @ (f2g (func_lookup fn) x y)
107
108     in
109     locs @ (List.fold_left stmt2g [] (snd f.body))
110
111     (* flatten sandbox, and thus the program *)
112     in let pi = List.map (fun b -> b.name ^ "_0") sbd.portin
113     in let po = List.map (fun b -> b.name ^ "_0") sbd.portout
114
115     (* in let circ_in = List.map (fun n -> Var(n)) pi
116     in let circ_out = List.map (fun n -> Var(n)) po *)
117     in
118     ( globes,
119       f2g sbd pi po,
120       pi,
121       po )
122
123     (* Pretty-printing functions *)
124
125     let string_of_node = function
126     | Val(i) -> string_of_int i
127     | Var(s) -> s
128     | Uo(u) -> string_of_uop u
129     | Op(o) -> string_of_op o
130     | As(a) -> string_of_asn a
131
132     let rec string_of_netlist = function
133     | [] -> "\n"
134     | n::tl -> string_of_node n ^ " " ^ string_of_netlist tl

```

```

1
2 (* Code Generation *)
3
4 module L = Llvml
5 module A = Ast
6 module F = Flat
7
8 module StringMap = Map.Make(String)
9
10 let translate (gl, nl, pi, po) =
11   (* setup context / module *)
12   let context = L.global_context () in
13   let the_module = L.create_module context "Sandbox"
14     and i32_t = L.i32_type context
15     and void_t = L.void_type context
16   in
17   let intyps =
18     Array.of_list( [ L.pointer_type i32_t; L.pointer_type i32_t; i32_t ] ) in
19   let mtyp = L.function_type void_t intyps in
20   let main = L.define_function "sandbox" mtyp the_module in
21   let builder = L.builder_at_end context (L.entry_block main) in
22
23   (* declare globals *)
24   List.iter
25     (fun (n, i) ->
26       ignore(L.define_global n (L.const_int i32_t i) the_module))
27     gl;
28
29   (* outputs of flipflops *)
30   let clock_asn =
31     let rec get_clock_asn fl = function
32       | [] | [_] -> fl
33       | hd::tl -> if (List.nth tl 0) = F.As(Casn)
34         then get_clock_asn (hd::fl) tl else get_clock_asn fl tl
35     in get_clock_asn [] nl
36   in
37   (* declare things that depend on state as static *)
38   let clock_vars =
39     let add_clock_variable m n =
40       let static0 = L.define_global (n ^ "__0") (L.const_int i32_t 0) the_module
41       and static1 = L.define_global (n ^ "__1") (L.const_int i32_t 0) the_module
42     in
43       L.set_linkage L.Linkage.Internal static0;
44       L.set_linkage L.Linkage.Internal static1;
45       StringMap.add n (static0, static1) m
46   in
47   List.fold_left
48     (fun m n -> match n with
49       | F.Var s -> add_clock_variable m s

```

```

50     | _ -> m (* never matched *)
51 ) StringMap.empty clock_asn
52 in
53 let clock_lookup n state = let v = StringMap.find n clock_vars
54 in if state = 0 then fst v else snd v
55 in
56
57 (* declare formals *)
58 let vars =
59     let add_formal m n p = L.set_value_name n p;
60     let local =
61         L.build_alloca
62         (if n = "sopt" then i32_t else L.pointer_type i32_t)
63         n builder
64     in
65     ignore (L.build_store p local builder);
66     StringMap.add n local m
67 in
68 (* define variables not on clock *)
69 let add_variable m n =
70     let var = L.build_alloca i32_t n builder
71     in StringMap.add n var m
72 in
73 (* add arguments *)
74 let portin = ["input"; "output"; "sopt"] in
75 let formals = List.fold_left2 add_formal StringMap.empty portin
76 (Array.to_list (L.params main))
77 in
78 List.fold_left
79 (fun m n -> match n with
80     | F.Var s -> if not (StringMap.mem s m || StringMap.mem s clock_vars)
81         then add_variable m s else m
82     | _ -> m
83 ) formals nl (* extract variable names from nl *)
84 in
85 (* Return the value for a variable or formal argument *)
86 let lookup n = StringMap.find n vars
87 in
88 (* get the state option *)
89 let sopt = lookup "sopt" in
90 (* load inputs *)
91 for i = 0 to ((List.length pi) - 1)
92 do
93     let arr = L.build_load (lookup "input") "input" builder in
94     let index = L.const_int i32_t i in
95     let ptr = L.build_in_bounds_gep arr [| index |] "" builder in
96     let inp = L.build_load ptr ("in"^(string_of_int i)) builder in
97     ignore(L.build_store inp (lookup (List.nth pi i)) builder)
98 done;
99 (* compute llvalue of flattened netlist and store outputs *)

```

```

100 let rec netlist s = function
101 | [] ->
102   (* store outputs *)
103   for i = 0 to ((List.length po) - 1 )
104   do
105     let v = List.nth po i in
106     let state = if sopt = L.const_int i32_t 0 then 0 else 1 in
107     let vv = if StringMap.mem v clock_vars
108               then clock_lookup v state
109               else lookup v
110     in
111     let out = L.build_load vv v builder in
112     let arr = L.build_load (lookup "output") "output" builder in
113     let index = L.const_int i32_t i in
114     let ptr = L.build_in_bounds_gep arr [| index |] "" builder in
115     ignore(L.build_store out ptr builder)
116   done;
117   (* need to construct the return statement *)
118   ignore(L.build_ret_void builder); builder
119 (* lookup_global name m *)
120 | n::tl -> Stack.push (match n with
121 | F.Val i -> L.const_int i32_t i
122 | F.Var v ->
123   if StringMap.mem v clock_vars then (
124     let state = (if sopt = L.const_int i32_t 0 then "__0" else "__1") in
125     let name = v ^ state in
126     let vv = match L.lookup_global name the_module with
127     | Some llv -> llv
128     | _ -> raise(Failure("never reached"))
129     in
130     (* if (List.mem v po) then (print_endline "1"; vv )else (print_endline "2";
131       L.build_load vv name builder) *)
132     if List.nth tl 0 = As(Casn) then vv else L.build_load vv name builder
133   ) else(
134     (* if List.mem v po then lookup v *)
135     if List.nth tl 0 = As(Asn) then lookup v
136     else L.build_load (lookup v) v builder
137   )
138 | F.Uo uo -> let n1 = Stack.pop s in
139   (match uo with
140   | A.Umin -> L.build_neg
141   | A.Not -> L.build_not
142   ) n1 "" builder
143 | F.Op op -> let n2 = Stack.pop s and n1 = Stack.pop s in
144   (match op with
145   | A.Add -> L.build_add n1 n2 "" builder
146   | A.Sub -> L.build_sub n1 n2 "" builder
147   | A.Lt -> L.build_icmp L.Icmp.Slt n1 n2 "" builder
148   | A.Gt -> L.build_icmp L.Icmp.Sgt n1 n2 "" builder

```

```

149         | A.Lte -> L.build_icmp L.Icmp.Sle n1 n2 "" builder
150         | A.Gte -> L.build_icmp L.Icmp.Sge n1 n2 "" builder
151         | A.Eq  -> L.build_icmp L.Icmp.Eq n1 n2 "" builder
152         | A.Neq -> L.build_icmp L.Icmp.Ne n1 n2 "" builder
153         | A.Or  -> L.build_or n1 n2 "" builder
154         | A.And -> L.build_and n1 n2 "" builder
155         | A.Xor -> L.build_xor n1 n2 "" builder
156         | A.Shl -> L.build_shl n1 n2 "" builder
157         | A.Shr -> L.build_lshr n1 n2 "" builder
158     )
159     | F.As a -> let n2 = Stack.pop s and n1 = Stack.pop s in
160         L.build_store n1 n2 builder
161     ) s;
162     netlist s tl
163 in
164
165 let (empty_stack : L.llvalue Stack.t) = Stack.create ()
166 in
167 ignore(netlist empty_stack nl);
168 the_module

```

```

1
2 (* Used for compiling *)
3
4 type action = Ast | Flatten | LLVM_IR | Compile
5
6 let _ =
7 let action = ref Compile in
8   let set_action a () = action := a in
9   let speclist = [
10     ("-a", Arg.Unit (set_action Ast), "Print the SAST");
11     ("-f", Arg.Unit (set_action Flatten), "Print the flattened net list");
12     ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
13     ("-c", Arg.Unit (set_action Compile), "Compile program");
14   ] in
15   let usage_msg = "usage: ./sandbox [-a|-f|-l|-c] [file.sb]" in
16   let channel = ref stdin in
17   Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
18   let pc = Pre.process !channel in
19   let lexbuf = Lexing.from_channel pc in
20   let ast = Parser.program Lexer.token lexbuf in
21   Semant.check ast;
22   match !action with
23   | Ast -> print_string (Ast.string_of_program ast)
24   | Flatten -> let (_,nl,_,_) = Flat.flatten ast in
25     print_string (Flat.string_of_netlist nl)
26   | LLVM_IR ->
27     print_string (Llvm.string_of_llmodule (Codegen.translate (Flat.flatten ast)))
28   | Compile -> let m = Codegen.translate (Flat.flatten ast) in
29     Llvm_analysis.assert_valid_module m;
30     print_string (Llvm.string_of_llmodule m)

```

```

1
2 // Tic Function
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7
8 extern void sandbox(int *ins, int *outs);
9
10 int main(int argc, char **argv)
11 {
12     // argv[1]...argv[argc - 3] are inputs
13     // argv[argc - 2] is number of outputs expected
14     // argv[argc - 1] is how times to loop
15     int inc = argc - 3;
16     int outc = atoi(argv[argc - 2]);
17     int loop = atoi(argv[argc - 1]);
18     int *ins = malloc(inc * sizeof(int));
19     int *outs = malloc(outc * sizeof(int));
20
21     int i;
22     for (i = 1; i <= argc - 3; i++)
23         ins[i] = atoi(argv[i]);
24
25
26     for (i = 0; i < loop; i++){
27         sandbox(ins, outs);
28         for(int j = 0; j < outc; j++)
29             printf("%d ", *(outs+j));
30         printf("\n");
31     }
32     return 0;
33 }

```
