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

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 VDHL 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:

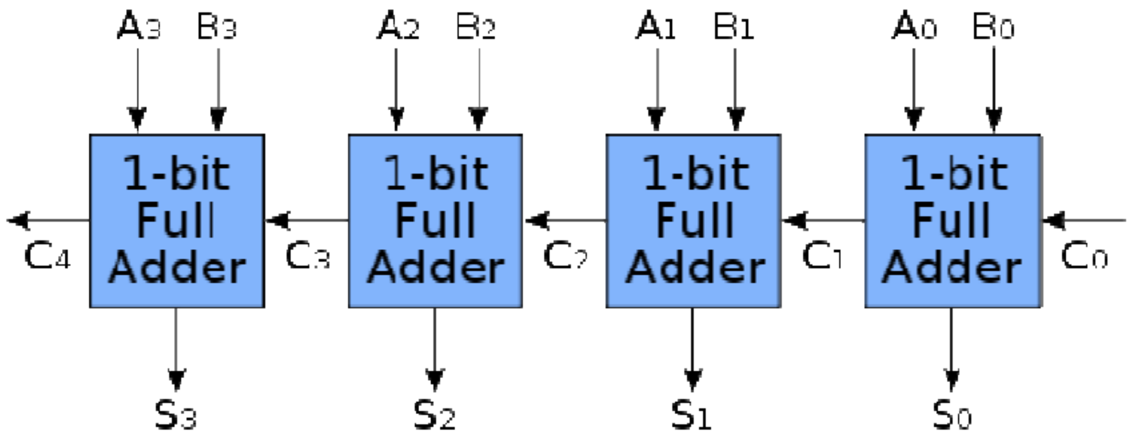

Figure 1: A 4-bit ripple-carry adder

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 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 programing 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
Ocamllex	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:

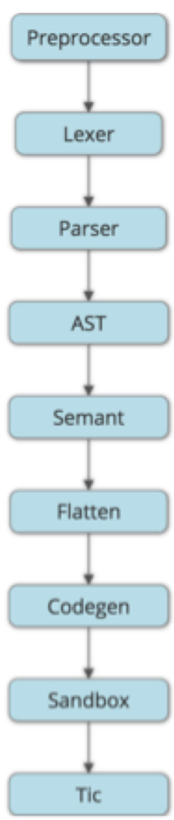

Figure 2: The architecture of the Sandbox compiler

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 ocamlyacc.

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 circut. 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 a 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 then 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

```
(*
  assumes the the .sb file is formatted correctly
  keeps track of indentation to form blocks
  adds semi-colons
  will not keep track if inside comments
*)

open Printf

let process ic =
  let out_file = "pre.txt" in
  let oc = open_out out_file in
  let rec read_lines l =
    try read_lines ((input_line ic)::l)
    with End_of_file -> close_in ic; l
  in

  let process_line t l =
    if l = "" then t
    else if l.[String.length l - 1] = '/' then (fprintf oc "%s\n" l; t)
    else if l.[0] = '\t' then (fprintf oc "%s;\n" l; 1)
    else (
      let tokens = String.split_on_char ' ' l
      and opt = if t = 1 then "}\n" else "" in
      match List.rev tokens with
        | [] -> fprintf oc ""; t (* never matched *)
        | hd::tl -> let last = hd and others = List.rev tl in
          let ll = String.length last in
          if ll > 0 then
            (* function header *)
            if last.[ll-1] = ':' then (
            fprintf oc "%s%s %s{\n"
            opt
            (String.concat " " others)
            (String.sub last 0 (ll-1)); 1)
            (* then global decl *)
            else (fprintf oc "%s%s;\n" opt l; 0)
          else (fprintf oc "%s%s;\n" opt l; 0)
    )
  in let lines = List.rev(read_lines [])
  in ignore(List.fold_left process_line 0 lines);
  fprintf oc "}\n";
  close_out_noerr oc;
  open_in out_file
```

```ocaml
(* Abstract Syntax Tree *)

type op = Add | Sub | Lt | Gt | Lte | Gte | Eq | Neq |
         Or | And | Xor | Shl | Shr

type uop = Not | Umin

type asn = Asn | Casn

type expr =
  | Num of int
  | Id of string
  | Subbus of string * int * int
  | Unop of uop * expr
  | Binop of expr * op * expr
  | Basn of expr * asn * string
  | Subasn of expr * asn * string * int * int

type stmt =
  | Expr of expr
  | Call of expr list * string * expr list

type bus = { name : string; size : int; init : expr; isAsn : bool array }

type gdecl = Const of bus * expr
  (* ensure in semant that this expr is int *)

type vdecl =
  | Bdecl of bus
  (* | Adecl of bus * int *)

type fbody = vdecl list * stmt list

type fdecl = {
    portin : bus list;
    fname  : string;
    portout : bus list;
    body    : fbody;
  }

type program = gdecl list * fdecl list

(* Pretty-printing functions *)

let string_of_op = function
  | Add -> "+"
  | Sub -> "-"
  | Lt  -> "<"
  | Gt  -> ">"
```

```
  | Lte -> "<="
  | Gte -> ">="
  | Eq  -> "=="
  | Neq -> "!="
  | Or  -> "|"
  | And -> "&"
  | Xor -> "^"
  | Shl -> "<<"
  | Shr -> ">>"

let string_of_uop = function
  | Not  -> "!"
  | Umin -> "-"

let string_of_asn = function
  | Asn  -> "->"
  | Casn -> "-:"

let rec string_of_expr = function
  | Num(l) -> string_of_int l
  | Id(s) -> s
  | Subbus(n, i1, i2) ->
     n ^ "(" ^
     (if i2=i1+1 then string_of_int i1
      else string_of_int i1 ^ ":" ^ string_of_int (i2-1))
     ^ ")"
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Binop(e1, o, e2) ->
     string_of_expr e1 ^ " " ^
     string_of_op o ^ " " ^
     string_of_expr e2
  | Basn(e, a, n) ->
     string_of_expr e ^ " " ^
     string_of_asn a ^ " " ^
     n
  | Subasn(e, a, n, i1, i2) ->
     string_of_expr e ^ " " ^
     string_of_asn a ^ " " ^
     string_of_expr (Subbus(n, i1, i2))

let rec string_of_stmt = function
  | Expr(expr) -> string_of_expr expr ^ "\n"
  | Call(il, f, ol) ->
     "[" ^
     String.concat ", " (List.map string_of_expr il) ^ "] " ^
     f ^ " [" ^
     String.concat ", " (List.map string_of_expr ol) ^ "]"

let string_of_bus bus =
  string_of_expr bus.init ^ " -> " ^
```

```
  "bit" ^ (if bus.size = 1 then "" else "." ^ string_of_int bus.size) ^
  " " ^ bus.name

let string_of_vdecl v = match v with
  | Bdecl bus -> string_of_bus bus

let string_of_gdecl v = match v with
  | Const(bus, s) -> string_of_bus bus ^ " const"

let string_of_fdecl fdecl =
  String.concat ", " (List.map (fun b -> b.name) fdecl.portin) ^ " " ^
  fdecl.fname ^
  String.concat ", " (List.map (fun b -> b.name) fdecl.portout) ^ ":\n\t" ^
  String.concat "\n\t" (List.map string_of_vdecl (fst fdecl.body)) ^ "\n\t" ^
  String.concat "\t" (List.map string_of_stmt (snd fdecl.body))

let string_of_program (vars, funcs) =
  String.concat "\n" (List.map string_of_gdecl (List.rev vars)) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

```
(* Lexical analyzer *)

{ open Parser }

rule token = parse
  |[' ' '\t' '\r' '\n'] {token lexbuf} (* eat whitespace *)

  (* binary operators *)
  | '+'             { PLUS }
  | '-'             { MINUS }
  | '|'             { OR }
  | '&'             { AND }
  | '^'             { XOR }
  | "<<"            { SHL }
  | ">>"            { SHR }
  | '<'             { LT }
  | '>'             { GT }
  | "<="            { LTE }
  | ">="            { GTE }
  | "=="            { EQ }
  | "!="            { NEQ }

  (* unary operator (also handle minus) *)
  | '!'             { NOT }

  (* other operators *)
  | "::"            { CAT }
  | '.'             { DOT }

  (* assignments *)
  | "->"        { ASSIGN }
  | "-:"            { CLKASN }

  (* delimiters *)
  | ','             { COMMA }
  | ';'             { SEMI }
  | ':'             { COLON }

  (* scoping *)
  | '('             { OPAREN }
  | ')'             { CPAREN }
  | '['             { OBRACK }
  | ']'             { CBRACK }
  | '{'             { OBRACE }
  | '}'            { CBRACE }

  (* key words *)
  | "const"         { CONST }
  | "bit"           { BIT }
```

```
  (* integer and string literals *)
  | ['0'-'9']+ as n { NUM(int_of_string n) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as i { ID(i) }

  (* Comments, unrecognized, and EOF *)
  |"/"              {comment lexbuf}
  | _               { raise (Failure("illegal character")) }
  | eof             { EOF }

and comment = parse
  | "/"         {token lexbuf}
  | eof         { raise (Failure("comment started but never finished")) }
  | _           {comment lexbuf}
```

```
/* parser for sandbox */

%{ open Ast %}

/* tokens */
%token PLUS MINUS OR AND XOR SHL SHR
%token LT GT LTE GTE EQ NEQ
%token NOT
%token ASSIGN CLKASN WIRE
%token COMMA SEMI COLON CAT DOT
%token CONST BIT
%token OPAREN CPAREN OBRACK CBRACK OBRACE CBRACE
%token <int> NUM
%token <string> ID
%token EOF

/* precedence */
%left COMMA SEMI
%right ASSIGN CLKASN
%left EQ NEQ
%left LT GT LTE GTE
%left PLUS MINUS
%left OR
%left XOR
%left AND
%left SHL SHR
%right NOT UMIN

%start program
%type <Ast.program> program

%%

program: decls EOF { $1 }

decls:
  | /* nothing */ { [], [] }
  | decls gdecl   { ($2 :: fst $1), snd $1 }
  | decls fdecl   { fst $1, ($2 :: snd $1) }

gdecl:
  | bdecl CONST SEMI { Const($1, $1.init) }

bdecl:
  | init_opt BIT size_opt ID
      { {
        name = $4;
        size = $3;
        init = $1;
```

```
      isAsn = Array.make $3 false
    } }

init_opt:
  | /* nothing */     { Num 0 }
  | expr assign       { $1 }

size_opt:
  | /* nothing */     { 1 }
  | DOT NUM           { $2 }


fdecl:
  OPAREN port CPAREN ID OPAREN port_out CPAREN
  OBRACE fbody CBRACE
    { {
      portin = $2;
      fname  = $4;
      portout = $6;
      body   = List.rev (fst $9), List.rev (snd $9);
    } }

port:
  | /* nothing */    { [] }
  | busses           { List.rev $1 }

port_out:
  | /* nothing */    { raise (Failure("Empty output port list")) }
  | busses           { List.rev $1 }

busses:
  | bdecl                 { [$1] }
  | busses COMMA bdecl    { $3 :: $1 }

fbody:
  | /* nothing */    { [], [] }
  | fbody local      { ($2 :: fst $1), snd $1 }
  | fbody stmt       { fst $1, ($2 :: snd $1) }

local:
  | vdecl SEMI       { $1 }

vdecl:
  | bdecl        { Bdecl($1) }

stmt:
  | asnexpr SEMI             { Expr $1 }
  | OBRACK actuals CBRACK ID OBRACK actuals CBRACK SEMI
      { Call($2, $4, $6) }
```

```
asnexpr:
  | expr assign ID { Basn($1, $2, $3) }
  | expr assign ID OPAREN NUM COLON NUM CPAREN
      { Subasn($1, $2, $3, $5, $7) }
  | expr assign ID OPAREN NUM CPAREN
      { Subasn($1, $2, $3, $5, $5+1) }

assign:
  | ASSIGN           { Asn }
  | CLKASN           { Casn }

actuals:
  | /* nothing */     { [] }
  | actual_list       { List.rev $1}

actual_list:
  | expr                { [$1] }
  | actual_list COMMA expr { $3 :: $1 }

expr:
  | NUM             { Num($1) }
  | ID              { Id($1) }
  | ID OPAREN NUM COLON NUM CPAREN { Subbus($1, $3, $5) }
  | ID OPAREN NUM CPAREN       { Subbus($1, $3, $3+1) }
  /* ADD CAT */
  | MINUS expr %prec UMIN  { Unop(Umin, $2) }
  | NOT  expr %prec NOT    { Unop(Not, $2) }
  | expr PLUS expr  { Binop($1, Add, $3) }
  | expr MINUS expr { Binop($1, Sub, $3) }
  | expr EQ   expr  { Binop($1, Eq, $3) }
  | expr NEQ  expr  { Binop($1, Neq, $3) }
  | expr LT   expr  { Binop($1, Lt, $3) }
  | expr LTE  expr  { Binop($1, Lte, $3) }
  | expr GT   expr  { Binop($1, Gt, $3) }
  | expr GTE  expr  { Binop($1, Gte, $3) }
  | expr AND  expr  { Binop($1, And, $3) }
  | expr OR   expr  { Binop($1, Or, $3) }
  | expr XOR  expr  { Binop($1, Xor, $3) }
  | expr SHL  expr  { Binop($1, Shl, $3) }
  | expr SHR  expr  { Binop($1, Shr, $3) }
  | OPAREN expr CPAREN     { $2 }
```

```
(* Semantic Checking *)

open Ast

module StringMap = Map.Make(String)

(*** HELPER FUNCTIONS ***)

(* raise failure if duplicates exist *)
let report_duplicate exceptf list =
  let rec helper = function
    | n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
    | _ :: t -> helper t
    | [] -> ()
  in helper (List.sort compare list)

(* give underlying bus of declarations *)
let gdec2b d = match d with Const(b, s) -> b
let vdec2b d = match d with Bdecl b -> b

(* number of bits required to describe int x *)
let bit_required x =
  (* for the moment assuming x > 0 tho *)
  let x = abs x
  in let log2 y =
    int_of_float ( ((log (float_of_int y)) /. (log 2.)) )
  in (log2 x) + 1

(* raise failure if some element not equal to another *)
let all_eq l =
  let rec diff d = function
    | [] | [_] -> d
    | hd::tl -> diff ((hd - List.nth tl 0)::d) tl
  in let diffs = diff [] l
  in if not (List.for_all (fun x -> x = 0) diffs) then
    raise (Failure("invalid arguments")) else ()

(* number of bits required for result of binop *)
let binop_size s1 op s2 = match op with
  | And | Or | Xor -> if s1 != s2
    then raise(Failure("operand sizes do not match "))
    else s1
  | Add | Sub -> Pervasives.max s1 s2
  | Shl | Shr -> s1
  | Lt | Gt | Lte | Gte | Eq | Neq -> 1

(*** CHECK THAT THE AST IS SEMANTICALLY CORRECT ***)

(* function for checking a single assign *)
```

```
(* use x and y to be usable for subbus *)
let check_basn e es b x y = ( match e with
  | Num _ -> if es > y-x then raise(Failure("size mismatch in " ^ b.name))
    else ()
  | Id _ | Subbus(_,_,_) | Unop(_,_) | Binop(_,_,_) -> if es != y-x
    then raise(Failure("size mismatch in " ^ b.name)) else ()
  | _ -> raise (Failure("illegal bus assignment: " ^ b.name)) );
  for i = x to y-1 do if b.isAsn.(i)
    then raise (Failure("bus " ^ b.name ^ " has more than one driver"))
    else b.isAsn.(i) <- true
  done

(* check if valid subbus *)
let check_subbus b x y =
  if x >= 0 && y <= b.size && x < y then ()
  else raise(Failure("incorrect dereference of " ^ b.name))

let check_subasn e es b x y =
  check_subbus b x y;
  check_basn e es b x y

(* main checking function *)
let check (globaldecls, functions) =
  (* checking globals *)
  let globals = List.map gdec2b globaldecls in
  (* no duplicate globals *)
  report_duplicate (fun n -> "duplicate global variable " ^ n)
  (List.map (fun g -> g.name) globals);
  (* globals intialized to an int *)
  let check_global_init g = match g.init with
    | Num _ -> ()
    | _ -> raise (Failure ("global " ^ g.name ^ " must be initialized to an integer"))
  in List.iter check_global_init globals;

  (* checking functions *)
  (* no duplicate functions *)
  report_duplicate (fun n -> "duplicate function " ^ n)
    (List.map (fun fd -> fd.fname) functions);

  (* collect declared functions *)
  let function_decls = List.fold_left
    (fun m fd -> StringMap.add fd.fname fd m) StringMap.empty functions
  in
  let function_decl s = try StringMap.find s function_decls
  with Not_found -> raise (Failure ("no function " ^ s))
  in
  (* ensure that sandbox defined *)
  let _ = function_decl "sandbox"
  in
  (* check each function decl *)
```

```
let check_function func =
  (* ensure no conflict between portin/portout/locals *)
  let locals = func.portin @ func.portout @ (List.map vdec2b (fst func.body))
  in report_duplicate
  (fun n -> "duplicate in/out/local bus " ^ n ^ " in " ^ func.fname)
  (List.map (fun b -> b.name) locals);
  (* build symbol table for all busses visible in function *)
  let symbols = List.fold_left (fun m b -> StringMap.add b.name b m)
  StringMap.empty (globals @ locals)
  in
  let lookup s =
    try StringMap.find s symbols
    with Not_found -> raise (Failure ("undeclared indentifier " ^ s))
  in

  (* need to ensure all outputs are assigned *)
  let out_names = List.map (fun b -> b.name) func.portout in
  let out_table = Hashtbl.create (2 * List.length out_names) in
  List.iter (fun out -> Hashtbl.add out_table out false) out_names;

  let check_const n =
    if List.mem n (List.map (fun b->b.name) globals) &&
    not (List.mem n (List.map (fun b->b.name) globals))
    then raise(Failure("cannot change const")) else ()
  in
  (* returns number of bits required for expression *)
  let rec expr = function
    | Num x -> bit_required x
    | Id s -> (lookup s).size
    | Subbus(n, i1, i2) -> let b = lookup n in
        check_subbus b i1 i2; i2-i1
    | Unop(op, e) -> expr e
    | Binop(e1, op, e2) -> let s1 = expr e1 and s2 = expr e2 in
        binop_size s1 op s2
    | Basn(e, a, n) -> let s = expr e and b = lookup n in
        check_const n;
        check_basn e s b 0 b.size;
        if List.mem n out_names then Hashtbl.replace out_table n true;
        b.size
    | Subasn(e, a, n, i1, i2) -> let s = expr e and b = lookup n in
        check_const n;
        check_subasn e s b i1 i2;
        if List.mem n out_names then Hashtbl.replace out_table n true;
        i2-i1
  in
  (* returns unit if semantically valid *)
  let rec stmt = function
    | Expr e -> ignore(expr e)
    | Call(inputs, n, outputs) -> let fd = function_decl n in
      (* can only assign to busses and subbuses *)
```

```
          List.iter (fun out -> match out with
            | Id _ | Subbus(_,_,_) -> ()
            | _ -> raise(Failure("only bus or subbus can be an output"))
          ) outputs;
          (* calls to sandbox are not permitted *)
          if fd.fname = "sandbox" then raise(Failure("cannot call sandbox"))
          (* do number of actuals/outputs match portin/portout *)
          else if (List.length inputs) != (List.length fd.portin)
            then raise(Failure("input mismatch in " ^ n))
          else if (List.length fd.portout) != (List.length outputs)
            then raise(Failure("output mismatch in " ^ n))
          (* can inputs fit in portin and outputs in portout *)
          (* accounts for shorthand function calls... *)
          else
            let check_ports acts port =
              List.iter2 (fun x y -> if not (x mod y = 0) then
                raise (Failure("invalid arguments")) else ())
              acts port;
              let quo = List.map2 (fun x y -> x / y) acts port
              in all_eq quo
            in
            check_ports
              (List.map expr inputs) (List.map (fun b -> b.size) fd.portin);
            check_ports
              (List.map expr outputs) (List.map (fun b -> b.size) fd.portout);

            (* sizes accounted for, but can outputs be assigned? *)
            let check_outasn o = match o with
              | Id s -> let out = lookup s in
                check_basn (Id "dummy") out.size out 0 out.size;
                if List.mem out.name out_names
                  then Hashtbl.replace out_table out.name true
              | Subbus(n, i1, i2) -> let out = lookup n in
                check_subasn (Id "dummy") (i2-i1) out i1 i2;
                if List.mem out.name out_names
                  then Hashtbl.replace out_table out.name true
              | _ -> raise (Failure(n ^ " cannot port to these outputs "))
            in List.iter check_outasn outputs

      (* check each statement and that all outputs are assigned *)
      in List.iter stmt (snd func.body);
      Hashtbl.iter (fun n x -> if x then ()
        else raise(Failure("not all outputs of " ^ func.fname ^ " assigned")))
      out_table
    in
    List.iter check_function functions

(* BREAK UP BUSSES INTO BITS *)
(* THE CODE BELOW HERE CAUSES NO ERRORS BUT IS NOT
   COMPLETE. THE INTENT WAS TO RETURN A MODIFIED AST
```

```
   WHERE EVERYTHNG HAS BEEN BROKEN INTO BITS
 *)

(* convert decimal number to list of bits of given len *)
let d2b x len =
  let rec dec2bin y lst = match y with
    | 0 -> (List.rev lst)
      @ (Array.to_list (Array.make (len - List.length lst) 0))
    | _ -> dec2bin (y / 2) ((y mod 2)::lst)
  in dec2bin x []

(* convert name into list of bit names *)
let n2b n i1 i2 =
  let rec name2bits n i lst =
    if i = i1 then (n ^ "_" ^ (string_of_int i1))::lst
    else name2bits n (i-1) ((n ^ "_" ^ (string_of_int i))::lst)
  in name2bits n (i2-1) []

(* break unops up into bitwise unops *)
let break_unop uop ex = match uop with
  | Not -> []
  | Umin -> []

(* break binops up into bitwise binops *)
let break_binop e1 op e2 = []

(* function for breaking a single assign *)
(* use x and y to be usable for subbus *)
let break_basn e a b x y = ( match e with
  | Num v -> List.map2 (fun q p -> Basn(Num q, a, p))
      (d2b v y) (n2b b.name x y)
  | Id s -> List.map2 (fun q p -> Basn(Id q, a, p))
      (n2b s 0 y) (n2b b.name x y)
  | Subbus(n,i1,i2) -> List.map2 (fun q p -> Basn(Id q, a, p))
      (n2b n i1 i2) (n2b b.name x y)
  | Unop(uo,e1) -> []
  | Binop(e1,o,e2) -> []
  | _ -> raise (Failure("never reached")) )

let break_busses gb =
  let binit = match gb.init with Num x -> x | _ -> 0 in
  let vals = d2b binit gb.size in
  let nams = n2b gb.name 0 gb.size in
  List.map2 (fun n v ->
    {
      name = n;
      size = 1;
      init = Num v;
      isAsn = [| false |]
    }
```

```
  ) nams vals

let break (globaldecls, functions) =
  (* break up globals *)
  let globals = List.map gdec2b globaldecls in
  let broken_globals = List.concat (List.map break_busses globals) in

  (* collect functions *)
  (* let function_decls = List.fold_left
    (fun m fd -> StringMap.add fd.fname fd m) StringMap.empty functions
  in
  let function_decl s = StringMap.find s function_decls
  in *)
  (* check each function decl *)
  let break_function func =
    (* ensure no conflict between portin/portout/locals *)
    let locals = func.portin @ func.portout @ (List.map vdec2b (fst func.body)) in
    (* let broken_locals = List.concat (List.map break_busses locals) in *)

    (* build symbol table for all busses visible in function *)
    let symbols = List.fold_left (fun m b -> StringMap.add b.name b m)
    StringMap.empty (globals @ locals)
    in let lookup s = StringMap.find s symbols
    in
    (* list of stmts on bits *)
    let rec break_stmt = function
      | Expr e -> (match e with
        | Basn(ex,a,nb) -> let b = lookup nb in
            break_basn ex a b 0 b.size
        | Subasn(ex,a,nb,i1,i2) -> let b = lookup nb in
            break_basn ex a b i1 i2
        | _ -> []
      )
      | Call(inputs, n, outputs) -> [] (* let fd = function_decl n in [] *)

    (* check each statement and that all outputs are assigned *)
    in List.map break_stmt (snd func.body)

  in
  broken_globals, List.map break_function functions
```

```
(* Flattening stage *)

open Ast

type node =
   | Val of int
   | Var of string
   | Uo of uop
   | Op of op
   | As of asn

module StringMap = Map.Make(String)

let flatten (globaldecls, functions) =
   (* globals busses *)
   let g2b d = match d with Const(b, s) -> b in
   let globals = List.map g2b globaldecls in
   (* let global_names = List.map (fun b -> b.name) globals in *)
   let globes =
      List.map
      (fun g -> match g.init with
         | Num x -> g.name, x
         | _ -> raise(Failure("never reached")) )
      globals
   in
   (* table keeping track of variable names *)
   let var_table = Hashtbl.create 100 in

   (* function declarations *)
   let function_decls = List.fold_left
   (fun m fd -> StringMap.add fd.fname fd m) StringMap.empty functions
   in
   let func_lookup n = StringMap.find n function_decls in
   let sbd = func_lookup "sandbox"
   in
   let rec f2g f inexpr outexpr =
      (* local busses *)
      let d2b d = match d with Bdecl b -> b in
      let ldec = List.map d2b (fst f.body) in
      let locals = f.portin @ f.portout @ ldec in
      (* for keeping track of naming *)
      let track_name b = let n = b.name in
         if Hashtbl.mem var_table n
         then Hashtbl.replace var_table n (Hashtbl.find var_table n + 1)
         else Hashtbl.add var_table n 0
      in List.iter track_name locals;
      let get_local n =
         n ^ "_" ^ string_of_int (Hashtbl.find var_table n) in
```

```
    (* all available busses *)
    let loces =
      List.concat
      (List.map
      (fun l -> match l.init with
        | Num x -> [Val x; Var(get_local l.name); As(Asn)]
        | _ -> raise(Failure("never reached")) )
      ldec)
    in
    (* symbols only contains locals, globals in globals *)
    let symbols = List.fold_left (fun m b -> StringMap.add b.name b m)
      StringMap.empty locals
    in
    (* mapping formals to actuals *)
    let f2a =
      let formals = List.map (fun b -> b.name) f.portin in
      List.fold_left2 (fun m f a -> StringMap.add f a m)
      StringMap.empty formals inexpr
    in
    (* mapping formals to outputs *)
    let f2o =
      let formals = List.map (fun b -> b.name) f.portout in
      List.fold_left2 (fun m f o -> StringMap.add f o m)
      StringMap.empty formals outexpr
    in
    let rec expr2g = function
      | Num i -> [Val i]
      | Id s -> (* is it a local? *)
        if StringMap.mem s symbols then
          (if StringMap.mem s f2a
          then [Var(StringMap.find s f2a)] (* is it a portin? *)
          else [Var(get_local s)]) (* or just a normal local? *)
        else [Var s] (* or a global *)
      | Unop(o, e) -> (expr2g e) @ [Uo o]
      | Subbus(n,e1,e2) -> []
      | Binop(e1, o, e2) -> (expr2g e1) @ (expr2g e2) @ [Op o]
      | Basn(e, a, n) -> let store =
          (if StringMap.mem n f2o
            then [Var(StringMap.find n f2o)] (* is it a portout? *)
          else [Var(get_local n)]) (* or just a normal local? *)
        in (expr2g e) @ store @ [As a]
      | Subasn(e, a, n, i1, i2) -> []
    in
    let rec stmt2g g = function
      | Expr e -> g @ (expr2g e)
      | Call(ins, fn, outs) ->
        (* let x = List.concat (List.map expr2g ins)
        and y = List.concat (List.map expr2g outs) *)
        let x = List.map
        (fun a -> match a with Id s -> get_local s | _ -> raise(Failure("not handled
```

```
                yet")))
            ins
            and y = List.map
            (fun a -> match a with Id s -> get_local s | _ -> raise(Failure("not handled
                yet")))
            outs
            in g @ (f2g (func_lookup fn) x y)
      in
      loces @ (List.fold_left stmt2g [] (snd f.body))

   (* flatten sandbox, and thus the program *)
   in let pi = List.map (fun b -> b.name ^ "_0") sbd.portin
   in let po = List.map (fun b -> b.name ^ "_0") sbd.portout

   (* in let circ_in = List.map (fun n -> Var(n)) pi
   in let circ_out = List.map (fun n -> Var(n)) po *)
   in
   ( globes,
      f2g sbd pi po,
      pi,
      po  )

(* Pretty-printing functions *)

let string_of_node = function
   | Val(i) -> string_of_int i
   | Var(s) -> s
   | Uo(u) -> string_of_uop u
   | Op(o) -> string_of_op o
   | As(a) -> string_of_asn a

let rec string_of_netlist = function
   | [] -> "\n"
   | n::tl -> string_of_node n ^ " " ^ string_of_netlist tl
```

```

(* Code Generation *)

module L = Llvm
module A = Ast
module F = Flat

module StringMap = Map.Make(String)

let translate (gl, nl, pi, po) =
   (* setup context / module *)
   let context = L.global_context () in
  let the_module = L.create_module context "Sandbox"
      and i32_t = L.i32_type  context
      and void_t = L.void_type context
   in
   let intyps =
      Array.of_list( [ L.pointer_type i32_t; L.pointer_type i32_t; i32_t ] ) in
   let mtyp = L.function_type void_t intyps in
   let main = L.define_function "sandbox" mtyp the_module in
   let builder = L.builder_at_end context (L.entry_block main) in

   (* declare globals *)
   List.iter
   (fun (n, i) ->
      ignore(L.define_global n (L.const_int i32_t i) the_module))
   gl;

   (* outputs of flipflops *)
   let clock_asn =
      let rec get_clock_asn fl = function
         | [] | [_] -> fl
         | hd::tl -> if (List.nth tl 0) = F.As(Casn)
            then get_clock_asn (hd::fl) tl else get_clock_asn fl tl
      in get_clock_asn [] nl
   in
   (* declare things that depend on state as static *)
   let clock_vars =
      let add_clock_variable m n =
         let static0 = L.define_global (n ^ "__0") (L.const_int i32_t 0) the_module
         and static1 = L.define_global (n ^ "__1") (L.const_int i32_t 0) the_module
         in
         L.set_linkage L.Linkage.Internal static0;
         L.set_linkage L.Linkage.Internal static1;
         StringMap.add n (static0, static1) m
      in
   List.fold_left
   (fun m n -> match n with
      | F.Var s -> add_clock_variable m s
```

```
      | _ -> m (* never matched *)
   ) StringMap.empty clock_asn
  in
  let clock_lookup n state = let v = StringMap.find n clock_vars
   in if state = 0 then fst v else snd v
  in

   (* declare formals *)
   let vars =
      let add_formal m n p = L.set_value_name n p;
         let local =
            L.build_alloca
            (if n = "sopt" then i32_t else L.pointer_type i32_t)
            n builder
            in
         ignore (L.build_store p local builder);
         StringMap.add n local m
      in
      (* define variables not on clock *)
      let add_variable m n =
         let var = L.build_alloca i32_t n builder
         in StringMap.add n var m
      in
      (* add arguments *)
      let portin = ["input"; "output"; "sopt"] in
      let formals = List.fold_left2 add_formal StringMap.empty portin
      (Array.to_list (L.params main))
      in
   List.fold_left
   (fun m n -> match n with
      | F.Var s -> if not ( StringMap.mem s m || StringMap.mem s clock_vars )
         then add_variable m s else m
      | _ -> m
   ) formals nl (* extract variable names from nl *)
  in
   (* Return the value for a variable or formal argument *)
   let lookup n = StringMap.find n vars
   in
   (* get the state option *)
   let sopt = lookup "sopt" in
   (* load inputs *)
   for i = 0 to ((List.length pi) - 1 )
      do
         let arr = L.build_load (lookup "input") "input" builder in
         let index = L.const_int i32_t i in
         let ptr = L.build_in_bounds_gep arr [| index |] "" builder in
         let inp = L.build_load ptr ("in"^(string_of_int i)) builder in
         ignore(L.build_store inp (lookup (List.nth pi i)) builder)
   done;
   (* compute llvalue of flattened netlist and store outputs *)
```

```
let rec netlist s = function
  | [] ->
    (* store outputs *)
    for i = 0 to ((List.length po) - 1 )
    do
      let v = List.nth po i in
      let state = if sopt = L.const_int i32_t 0 then 0 else 1 in
      let vv = if StringMap.mem v clock_vars
        then clock_lookup v state
        else lookup v
      in
      let out = L.build_load vv v builder in
      let arr = L.build_load (lookup "output") "output" builder in
      let index = L.const_int i32_t i in
      let ptr = L.build_in_bounds_gep arr [| index |] "" builder in
      ignore(L.build_store out ptr builder)
    done;
    (* need to construct the return statement *)
    ignore(L.build_ret_void builder); builder
  (* lookup_global name m *)
  | n::tl -> Stack.push (match n with
    | F.Val i -> L.const_int i32_t i
    | F.Var v ->
      if StringMap.mem v clock_vars then (
        let state = (if sopt = L.const_int i32_t 0 then "__0" else "__1") in
        let name = v ^ state in
       let vv = match L.lookup_global name the_module with
        | Some llv -> llv
        | _ -> raise(Failure("never reached"))
        in
       (* if (List.mem v po) then (print_endline "1"; vv )else (print_endline "2"; L.build_load vv name builder) *)
       if List.nth tl 0 = As(Casn) then vv else L.build_load vv name builder

      ) else(
        (* if List.mem v po then lookup v *)
        if List.nth tl 0 = As(Asn) then lookup v
        else L.build_load (lookup v) v builder
      )
    | F.Uo uo -> let n1 = Stack.pop s in
      (match uo with
      | A.Umin -> L.build_neg
      | A.Not -> L.build_not
      ) n1 "" builder
    | F.Op op -> let n2 = Stack.pop s and n1 = Stack.pop s in
      (match op with
      | A.Add -> L.build_add n1 n2 "" builder
      | A.Sub -> L.build_sub n1 n2 "" builder
      | A.Lt -> L.build_icmp L.Icmp.Slt n1 n2 "" builder
      | A.Gt -> L.build_icmp L.Icmp.Sgt n1 n2 "" builder
```

```
          | A.Lte -> L.build_icmp L.Icmp.Sle n1 n2 "" builder
          | A.Gte -> L.build_icmp L.Icmp.Sge n1 n2 "" builder
          | A.Eq -> L.build_icmp L.Icmp.Eq n1 n2 "" builder
          | A.Neq -> L.build_icmp L.Icmp.Ne n1 n2 "" builder
          | A.Or -> L.build_or n1 n2 "" builder
          | A.And -> L.build_and n1 n2 "" builder
          | A.Xor -> L.build_xor n1 n2 "" builder
          | A.Shl -> L.build_shl n1 n2 "" builder
          | A.Shr -> L.build_lshr n1 n2 "" builder
          )
       | F.As a -> let n2 = Stack.pop s and n1 = Stack.pop s in
          L.build_store n1 n2 builder
    ) s;
    netlist s tl
  in

  let (empty_stack : L.llvalue Stack.t) = Stack.create ()
  in
  ignore(netlist empty_stack nl);
  the_module
```

```ocaml

(* Used for compiling *)

type action = Ast | Flatten | LLVM_IR | Compile

let _ =
let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the SAST");
    ("-f", Arg.Unit (set_action Flatten), "Print the flattened net list");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile), "Compile program");
  ] in
  let usage_msg = "usage: ./sandbox [-a|-f|-l|-c] [file.sb]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
  let pc = Pre.process !channel in
  let lexbuf = Lexing.from_channel pc in
   let ast = Parser.program Lexer.token lexbuf in
   Semant.check ast;
   match !action with
      | Ast -> print_string (Ast.string_of_program ast)
      | Flatten -> let (_,nl,_,_) = Flat.flatten ast in
      print_string (Flat.string_of_netlist nl)
      | LLVM_IR ->
         print_string (Llvm.string_of_llmodule (Codegen.translate (Flat.flatten ast)))
      | Compile -> let m = Codegen.translate (Flat.flatten ast) in
      Llvm_analysis.assert_valid_module m;
      print_string (Llvm.string_of_llmodule m)
```

```c

// Tic Function

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern void sandbox(int *ins, int *outs);

int main(int argc, char **argv)
{
   // argv[1]...argv[argc - 3] are inputs
   // argv[argc - 2] is number of outputs expected
   // argv[argc - 1] is how times to loop
   int inc = argc - 3;
   int outc = atoi(argv[argc - 2]);
   int loop = atoi(argv[argc - 1]);
   int *ins = malloc(inc * sizeof(int));
   int *outs = malloc(outc * sizeof(int));

   int i;
   for (i = 1; i <= argc - 3; i++)
      ins[i] = atoi(argv[i]);


   for (i = 0; i < loop; i++){
      sandbox(ins, outs);
      for(int j = 0; j < outc; j++)
         printf("%d ", *(outs+j));
      printf("\n");
   }
   return 0;
}
```