Logisimple
A Simple Hardware Description Language

Yuanxia Lee, Sarah Walker, Kundan Guha, Hannah Pierce-Hoffman

December 21, 2017
**Team Roles**

- **Yuanxia Lee**
  - Language Guru

- **Sarah Walker**
  - Tester

- **Kundan Guha**
  - System Architect

- **Hannah Pierce-Hoffman**
  - Manager
Language Overview

- Built off primitives (AND, OR, NOT)
- Allows for easy customization
- Combinational logic
Logisimple Syntax

All statements wrapped in a "TICK" gate:

```plaintext
TICK(in1, in2){
    AND a(in1, in2);
    out = a;
}
```

Last statement of every gate is the keyword `out` followed by the output variable.
Example of a user-defined gate:

```plaintext
MYGATE{
    OR a(0,0);
    AND b(a,1);
    out = b;
}
```
Overview:

- An initial Hello World
- Only 2-input AND, OR, and NOT gates
- Multiple primitive statements
- Program represented as variable list * stmt list
- No constants

Biggest challenges:

- Where to put builder blocks
- No StringMap for variables
- No support for constants
- Needed to bypass AST and feed Netlist directly to Codegen
Hello World: Part 1

What we learned:

- Basic mechanism: allocate space for variable, load variable, perform operation, store result
- Beginning of test suite: single statements, multiple statements. Verify test by manually inspecting LLVM code.

We compiled \texttt{AND mygate(hannah, sarah);} into:
Our initial Hello World was nice, but we had a long to-do list:

- Support for constants in Codegen
- StringMaps to hold variables
- AST-to-Netlist
- User-defined gates
- Arrays
- Semantic checking
- Standard library
Include boolean LLVM type:

```python
16  +   and zero = L.const_int i8_t 0
17  +   and one = L.const_int i8_t 1 in
```

Allocate space for a boolean LLVM type:

```python
21  +     let codegen_arg a name builder =
22  +         match a with
23  +             A.Name n -> let arg = L.buildalloca i8_t n builder in
24  +                 L.build_load arg (name ^ "inp") builder
25  +             | A.Bool b -> if b then one else zero
26  +             | in
27  +                 let expr builder (A.Expr (out, gate)) =
```
StringMap to hold variables

```haskell
44     build_inputs 0 inputs StringMap.empty in
45     let codegen_arg a name builder =
46         (*let rec print_keys l = match l with
47          | [] -> print_endline "contained"
48          | (x, y) :: t -> print_bytes (x ^ " "); print_keys t in
49          ignore (print_keys (StringMap.bindings vars));*)
50     let lookup n vars = try StringMap.find n vars
51         with Not_found -> print_endline ("Key: ", n, " not found.");
52            raise Not_found in
53     let codegen_arg a vars builder =
54         StringMap.add out store_output vars
```
Bridge.ml takes the AST, walks the AST, and calls flatten on the body of the gate definition, and then outputs a Netlist object (name, inputs, list of Netlist statements, outputs).

```
let convert_to_nl gdef =

  let prog_body = gdef.A.body in

  let v_starter = [] and vdm_starter = StringMap.empty in
  let (vl_res, vdm_res) = process Mixed_list prog_body v_starter vdm_starter in

  let gdecl_map = StringMap.empty in
  let nl_stmts = T.flat vl_res vdm_res gdecl_map in

  let converted_str = special_converter gdef.out in

  let tup = (gdef.name, gdef.inp, nl_stmts, [converted_str]) in
  tup
```
Semant and Flatten

- Semant.ml performs semantic checking (e.g. redefining a gate)
- Flatten produces part of the Netlist
User-defined Gates

Use our primitives to build bigger gates.

Challenges:

- Storing new type names and gate bodies
- Need to ”mangle” names to avoid identifier conflicts
- Need to ”flatten” AST into a Netlist of primitives that Codegen can translate
Arrays hold booleans and boolean variables. Example of a boolean array "one:"

```c
bool a = 1;
bool[2] one = [a, 0];
```
Macro expansion with m4 and include keyword in Logisimple source code

Expands standard library files into Logisimple source file

Example standard library gates:

```
NAND(a,b) {
    AND x(a,b);
    NOT y(x);
    out = y;
}

NOR(a,b) {
    OR x(a,b);
    NOT y(x);
    out = y;
}

XOR(a,b) {
    OR o(a,b);
    NAND na(a,b);
    AND a(o,na);
    out = a;
}
```
Test Suite

- tester.c
  - Wrapper to pass inputs and print outputs
- compile_file.sh
  - .sim to .ll to .s to .o to executable

```bash
9   echo $file $tester "${file}.ll"
10  ./logisimple.native "${file}.sim" > "${file}.ll"
11  llc "${file}.ll"
12  gcc -c -o "${file}.o" "${file}.s"
13  gcc -c -o "${tester}.o" "${tester}.c"
14  gcc -o "${tester}" "${tester}.o" "${file}.o"
```

- Test-to-Pass
  - Testing for syntax and parsing (e.g. one primitive vs. two primitives)
  - Testing for logical operations (i.e. "Does this match its truth table?")
- Test-to-Fail
Lessons Learned

Sarah:
- Learn to love your tools
- Learn from others’ success (and failure)
- Communicate your expectations

Hannah:
- Ask for help
- Teamwork
Lessons Learned

Yuanxia:
- Early is never early enough
- Get help, and look at previous work

Kundan:
- Importance of communication
- Don’t procrastinate
Demo

LIVE DEMO