MiniJazz and netlist

Hadrien Barral <hadrien.barral@ens.fr>
Théophile Wallez <theophile.wallez@inria.fr>

1 The MiniJazz language

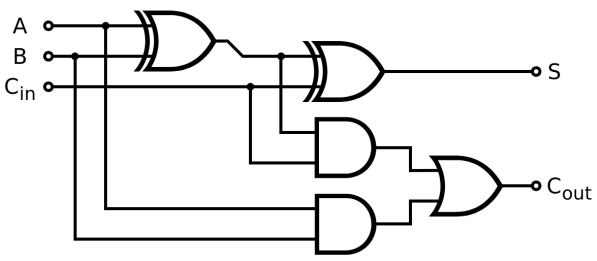
1.1 Language description

The MiniJazz language is a synchronous circuit *hardware description language* (HDL). It synthesizes an HDL description of a circuit into a netlist. Here follows an example of a *full-adder*, described using MiniJazz:

```
fulladder(a,b,c_in) = (s, c_out) where
t = a ^ b;
s = t ^ c_in;
c_out = (a & b) + (t & c_in);
end where
```

This code is compiled into the following netlist:

```
INPUT a, b, c_in
OUTPUT s, c_out
VAR
_l_10, _l_9, a, b,
c_in, c_out, s, t
IN
c_out = OR _l_9 _l_10
s = XOR t c_in
t = XOR a b
_l_9 = AND a b
_l_10 = AND t c_in
```



Block

A circuit is described by set of *blocks*. A *block* is defined by its name `f`, inputs `i_1, .., i_p`, outputs `o_1, ..., o_q` and a set of equations `D` defining the value of the outputs given its inputs. Local variables can be used without prior declaration.

```
f(i_1, ..., i_p) = (o_1, ..., o_q) where
  D
end where
```

Equations can take several forms:

- Variable assignment: `x = e` where `e` is an expression
- Block instantiation: `(y_1, ..., y_q) = g(x_1, .., x_p)`
- Conditional: **`if`** `c` **`then`** `D1` **`else`** `D2` **`end if`** using `c` is a **static** boolean expression, made of *global constant* variables declared with `const n = 32`, immediates (0, 1, 2, ...), static block parameters (see section below), combined with operators (`+, -, *, /, ^, =, <=`).

The following boolean operators can be used in the right-hand side of variable assignments (sorted by increasing arity):

- `true` or `1`: constant value
- `false` or `0`: constant value
- `not`: negation
- `&` or `and`: logical conjunction
- `xor` or `^`: exclusive or
- `+` or `or`: logical disjunction
- `nand`: *not and*
- **`mux`**: 1-bit multiplexer

Buses

A variable `x` (intput or output) can represent a ribbon of `n` wires (or equivalently, a *bus*) with the syntax `x : [n]`. The i-th wire is accessed with `x[i]` (indices start at 0). A sub-bus can be extracted using `x[i1 .. i2]`. Assuming a bus of width `n`, syntactic sugar exists for `x[i1..n-1]` (use `x[i1..]`) and `x[0..i2]` (use `x[..i2]`). Buses can be concatenated using the `.` operator. The empty bus is denoted by `[]`.

Hereafter is a short example of a block describing a 1-bit logical left shift:

```
shift_left(a:[4]) = (o:[4]) where
  o = a[1..3] . 0;
end where
```

Static block parameters

Buses have their widths specified by a *static integer expression*, made of global constant variables, immediates (0, 1, 2, ...), block parameters, combined with operators (`+, -, *, /, ^`).
Block parameters are given immediately following their name in angled brackets. Combined with conditionals and recursion, they allow for advanced use-cases. For instance, an n-bit adder can be described from a `fulladder` block and a $n-1$ bit adder:

```
adder<n>(a:[n], b:[n], c_in) = (o:[n], c_out) where
  if n = 0 then
    o = [];
    c_out = 0
  else
    (s_n1, c_n1) = adder<n-1>(a[1..], b[1..], c_in);
    (s_n, c_out) = fulladder(a[0], b[0], c_n1);
    o = s_n . s_n1
  end if
end where
```

Memory and registers

Even though it could theoretically be possible to design registers using the previously described gates (e.g. flip-flops), they introduce combinatorial cycles that hinder the simulation of the circuit. For this purpose, we provide an abstract construct that isolates those constructs into a black box to which we refer. The syntax in MiniJazz is the following:

```
x = reg(exp);
```

Similarly, volatile (RAM) and non-volatile (ROM) memory can be used in MiniJazz. Likewise, we will refer to them in an abstract form and will not convert them into gates. They can be described using the following syntax:

```
o1 = ram<addr_size, word_size>(read_addr:[addr_size],
                               write_enable:[1],
                               write_addr:[addr_size],
                               write_data:[word_size]);

o2 = rom<addr_size, word_size>(read_addr:[addr_size]);
```

where `word_size` is the memory word size (number of bits read/written). `addr_size` is the size of memory addresses (i.e. the memory contains $2^{\texttt{addr\_size}}$ words). A ROM has a single input (memory address to read) and outputs the word stored at the given address. A RAM has the same reading capabilities, but has additional inputs for writing: `write_enable` sets whether a write operation is performed at address `write_addr` with value `writedata`. Note that RAM can work like registers: it can read and write at the same address during the same cycle without introducing a combinational cycle, with the output value being the one from the previous cycle.

1.2 Compiler

The MiniJazz compiler outputs from the hardware-description an unordered *netlist*. The main block has be be specified with the `-m` option (otherwise, `main` is the default one). Below is a short example showing how to invoke the compiler:

```
> ./mjc.byte -m my_block mon_fichier.mj
```

The output file is a text-file describing a *netlist*, with extension `.net`. We cover *netlists* in the next section.

2 The *netlist* language

2.1 Concrete syntax

A *netlist* has the following layout:

```
INPUT inputs
OUTPUT outputs
VAR vars IN
  D
```

`inputs` and `outputs` are the inputs and outputs of the circuit, given as an identifier list.
`vars` gives the type of all inputs, outputs and local variables, given as a list of either:

- `x` representing a 1-bit variable,
- `x:n` representing a bus with n bits.

Finally, *D* represents the circuit logic gates. There are several type of operations.
Boolean operations:

- `x = constant`,
- `x = NOT a`,
- `x = AND a b`,
- `x = NAND a b`,
- `x = OR a b`,
- `x = MUX choice a b`, meaning that `x` is equal to `a` if `choice=0`, otherwise it is equal to `b`.

Delaying a value:

- `x = REG a`, meaning that `x` is equal to the previous value of `a`.

Operation on buses:

- `x = CONCAT a b`, representing the concatenation of the two buses `a` and `b`,
- `x = SELECT i a`, representing the selection of the `i`th bit in the bus `a`,
- `x = SLICE i1 i2 a`, representing a slice from the `i1`th bit to the `i2`th bit (inclusive) from the bus `a`,

Reading and writing the memory:

- `x = ROM addr_size word_size read_addr`,
- `x = RAM addr_size word_size read_addr write_enable write_addr write_data`.

The parameters for `ROM` and `RAM` are identical to the ones in MiniJazz:

- `addr_size`: the number of bits needed to represent an address (an integer)
- `word_size`: the number of bits that you read at once (an integer)
- `read_addr`: the read address (a variable with bus size `addr_size`)
- `write_enable`: a switch to enable writing to memory
- `write_addr`: the write address (a variable with bus size `addr_size`)
- `write_data`: the data to write in memory (a variable with bus size `word_size`)

Below is a short unordered *netlist* example for a full-adder circuit as generated by MiniJazz, and its ordered version on the right, as seen by your simulator:

```
INPUT a, b, c_in
OUTPUT s, c_out
VAR
a, b, c_in, s, c_out,
_t_1, _l_2, _l_3
IN
t_1 = XOR a b
s = XOR t_1 c_in
c_out = OR _l_2 _l_3
_l_2 = AND a b
_l_3 = AND t_1 c_in
```

```
INPUT a, b, c_in
OUTPUT s, c_out
VAR
a, b, c_in, s, c_out,
_t_1, _l_2, _l_3
IN
t_1 = XOR a b
_l_2 = AND a b
_l_3 = AND t_1 c_in
s = XOR t_1 c_in
c_out = OR _l_2 _l_3
```

2.2 Abstract syntax

The abstract syntax of the *netlist* language is given in file `netlist_ast.ml`. The lexer and the parser are given to you, ready to be used (the compilation course will teach you how to write lexers and parsers).

To convert a *netlist* to an abstract syntax tree, use the following function defined in file `netlist.ml` (input is a filename, output is the AST):

```
val read_file : string -> Netlist_ast.program
```

You can also perform the opposite operation using the function defined in file `netlist_printer.ml`:

```
val print_program : out_channel -> Netlist_ast.program -> unit
```

3 Designing your own HDL

You might not like the MiniJazz syntax, or its infamous error messages. That's fine! It is perfectly feasible to design your own HDL by writing a small library in your favorite language.

You can find an example of a HDL shallowly embedded in Python at `https://www.github.com/TWal/carotte.py`.

Make sure you are able to generate all the example netlists that you were given in the TP1.