Installing and Using Jazz, release 0.3b

Alexandre.Frey@ensmp.fr

Download

Jazz is available from http://www.cma.ensmp.fr/jazz/download.html. Download one of the following archive files:

Windows NT/9x	jazz-0.3b-x86-w32.zip
Compaq's Tru64 UNIX*	jazz-0.3b-alpha-osf.tar.gz
Sparc Solaris 2.6	jazz-0.3b-sparc-solaris2.tar.gz
Linux (glibc2)	n/a†

* Formerly Digital Unix
† Available on request.

Emacs is the preferred way of editing Jazz programs. You can get it at http://www.gnu.org/software/emacs/emacs.html. A port for Windows NT/9x is available at http://www.cs.washington.edu/homes/voelker/ntemacs.html. Note that Emacs version 20 is required.

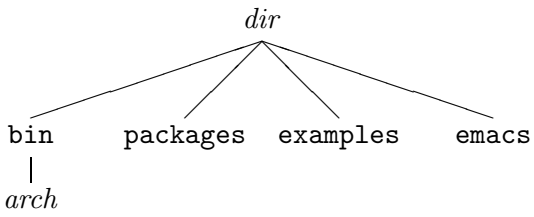
Content

Release **0.3b** of the Jazz system includes:

- The Jazz compiler and its libraries, version 0.3.5;
- The Blues circuit simulator, version 1.07 and its associated visualisation tool Rhythm, version 1.05.

Installation

1. Uncompress the archive in directory *dir*. This will create the following directory structure, where *arch* is one of `x86-w32`, `sparc-solaris2`, `alpha-osf`, `x86-linux`:



2. Add *dir*/bin/*arch* to your PATH environment variable[1].

3. The JAZZPATH environment variable is a list of directories where the Jazz compiler finds the libraries. Set it to *dir*/packages, which contains the standard Jazz packages. You can add directories containing your own packages[2].

4. An Emacs mode for editing Jazz programs is included in the distribution. To use it, add the following lines to your .emacs:

```
(setq load-path (cons "dir/emacs" load-path))
(autoload 'jazz-mode "jazz-mode" "" t)
(setq auto-mode-alist (cons '("\\.jzz$" . jazz-mode)
                            auto-mode-alist))
```

Compilation

Here is a simple Jazz source file:

```
// example.jzz
import jazz.circuit.*;
import jazz.circuit.Net.*;

// a + b + c = 2 * s + r
fun fullAdd(a, b, c) = (s, r) {
  x = a ^ b;
  s = x ^ c;
  r = mux(x, c, a);
}

// n-bit Ripple-Carry adder
```

[1] On Windows NT, open Control Panel, double-click on System, select the Environment tab, click on user variable PATH and add *dir*/bin/x86-w32; in front of its value. On Windows 9x, add set PATH=*dir*/bin/x86-w32;%PATH% at the end of your autoexec.bat.

[2] Use ':' as separator on Unix and ';' on Windows.

```
fun adder(n)(a: Net[n], b: Net[n]) = (s: Net[n+1]) {
  r[0] = constant(0);
  for (i < n) {
    (s[i], r[i+1]) = fullAdd(a[i], b[i], r[i]);
  }
  s[n] = r[n];
}

N=8;

device Adder {
  input a: Net[N], b: Net[N];
  output s: Net[N+1];

  s = adder(N)(a, b);
}

device AdderInputs {
  output a: Net[N];
  output b: Net[N];

  // generates some arbitrary inputs
  for (i < N) {
    a[i] = constant(2*i+5/(6*i + 1));
    b[i] = constant(8*3*i/(4*i+1));
  }
}

// generate the net-lists
export Adder();
export AdderInputs();
```

To compile `example.jzz`, hit `C-cC-b` under Emacs or execute `jazz example.jzz` in a command-line interpreter:

```
$ jazz example.jzz
%% The Jazz compiler version 0.3.5
%% Compilation: 1.43 s (1.56 s real)
%% Link: 0.05 s (0.04 s real)
%% Device "Adder". nets: 132, mux: 8, per: 1, ^: 16, op: 25, net/op: 5.28
%% Combinatorial depth: 9
%% Writing "./Adder.jzn"
%% Device "AdderInputs". nets: 16, per: 16, op: 16, net/op: 1.00
%% Combinatorial depth: 1
```

```
%% Writing "./AdderInputs.jzn"
%% Execution: 0.06 s (0.07 s real)
```

This file contains two device exportations, so the compilation produced two net lists: `Adder.jzn` and `AdderInputs.jzn`.

Simulation

Blues is an off-line device simulator: it takes a device net-list, simulates it for a given number of cycles, and writes the value of the output nets for each cycle in a simulation trace (`bio`). If the device has inputs, the simulator must be fed with the outputs of another device. The connection is based on the names of the nets.

For example:

```
$ blues -n 128 AdderInputs.jzn -o AdderInputs.bio
%% The Blues Simulator version 1.07
$ blues -n 128 -i AdderInputs.bio Adder.jzn -o Adder.bio
%% The Blues Simulator version 1.07
```

Use Rhythm to visualize the `.bio` files in a human-readable form. This tool generates an HTML file:

```
$ rhythm < Adder.bio > Adder.html
%% Rhythm version 1.05
```

By default, Adder.html contains the values of all input, output, and probed nets (see below, section "Debug"). You may select a subset of nets to trace and customize the output by using a configuration file:

```
$ rhythm Adder.rcf < Adder.bio > Adder.html
```

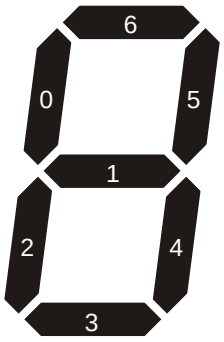
where `Adder.rcf` contains:

```
columns:
  a[0..7]: uint;
  b[0..7]: uint;
  s[0..8]: uint;
end:
```

The configuration file defines the content and format of all the columns. Content is specified by an ordered list of comma separated net names. The slice notation may also be used: `a[0..7]` is equivalent to `a[0], a[1], ..., a[7]`. This list of nets defines a integer for each cycle (least significant bits are on the left). This integer is displayed according to the format keyword:

Keyword	Semantics
`uint`	unsigned integer in decimal
`int`	signed integer in decimal
`xint`	unsigned integer in hexadecimal
`bint`	unsigned integer in binary
`net`	bit
`char`	8-bit ASCII character
`seg7`	7-segment decoder

The bit-segment mapping for 7-segment decoders is the following:



If you use 7-segment decoders, beware that the digits are displayed as images in the HTML file. These images as generated on-the-fly in a directory named `images` relative to the current directory.

Finally, if you simulate many cycles, **Rhythm** may generate unreasonably large HTML files. You can use the following options to select the cycles you want to display:

`-start` n_1	Start to display at cycle number n_1
`-stop` n_2	Display only cycles below n_2 (inclusive)
`-step` p	Select every p^{th} cycle
`-enable` *netName*	Display a cycle only if the value of *netName* is 1

Example:

```
$ rhythm -start 32 -stop 64 Adder.rcf < Adder.bio > Adder.html
```

Debug

By default, it is only possible to visualize input or output nets. Internal nets must be explicitely *probed* and given a "probe name". The syntax is:

`@Debug.probe(`⟨*net*⟩`,`⟨*netName*⟩`)`

Beware that this line may be executed several times with different nets. In this case, you must take care to generate non-ambiguous net names. For example, here how to probe the carries of `adder`:

```
// n bit Ripple-Carry adder
// with probed carries
fun adder(n)(a: Net[n], b: Net[n]) = (s: Net[n+1]) {
  r[0] = constant(0);
  for (i < n) {
    (s[i], r[i+1]) = fullAdd(a[i], b[i], r[i]);
  }
  s[n] = r[n];

  // DEBUG
  for (i < n+1) {
    @Debug.probe(r[i], format("r[%d]", i));
  }
}
```

The carries may then be referred to as `r[0]`, ..., `r[8]` in `.rcf` files.

Feedback

Send your bug reports and comments to jazz@cma.ensmp.fr.