Microprocessor Projet Presentation

Antoine Amarilli, Pablo Rauzy

ÉNS

21 january 2010

Prototype

The prototype of `scifi` is as follows:

```
Usage:
scifi -n <netlist> [-i input] [-o output] [-c nb_cycles] [-t clock] [-r rom]
scifi -s [-o output] [-c nb_cycles] [-t clock] [-r rom]
scifi -h (--help) : show this help

-s (--script)       Use this flag in a shebang to use scifi as a script
                    interpreter.

-n (--netlist)      Path to the file containing the netlist you want to
                    simulate. This argument is mandatory if -s is not present.

-i (--input)        Path to a file containing input values for the simulation.
                    Default is stdin.

-o (--output)       Path to a file to which values computed by the simulation
                    will be written. Default is stdout.

-c (--nb-cycles)    Number of cycles you want to execute.
                    Default is 1. 0 for infinity.

-a (--analyse-only) Don't do simulation if this flag is on, only parse and
                    analyse the given netlist.

-t (--clock-tick)   Interval between cycles in microseconds
                    (>= 0 and <= 1000000). Default is 0.

-r (--rom)          Path to a file with instructions in binary which will be
                    stored in the generated rom.
```

Example calls

- Here is an example command line used to run `scifi`:

  ```
  yes 0 | scifi -r rom -n main.nls -c 0 -t 7350
  ```

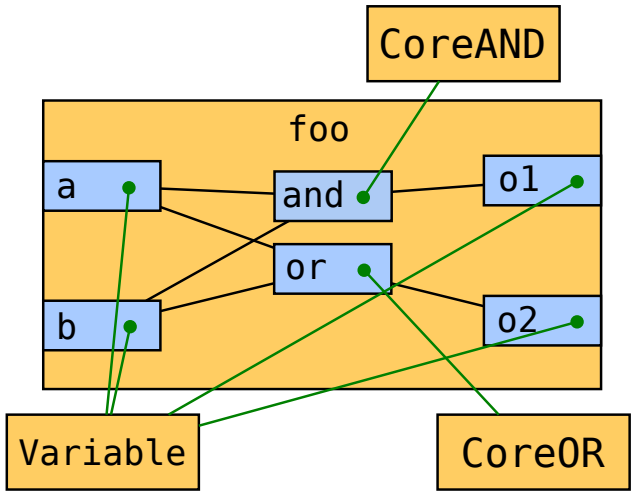
- However, we can also set the execute permission on `main.nls`, add a shebang, and write:

  ```
  yes 0 | ./main.nls
  ```

Implementation details

- scifi is coded in C in a object oriented way.
- There are two primitive types of object: `Function` and `Block`.
- The netlist's graph isn't unfolded. We use a global register table and a recursive addressing mechanism.

```
o1 o2 = foo a b
{
  o1 = and a b;
  o2 = or a b;
}
```



Netlist EBNF

The EBNF for netlists is as follows:

```
<alpha> ::= 'a'..'z' | 'A'..'Z' | '_'
NAME ::= <alpha> ( <alpha> | '0'..'9' | '-' ){0,31}
FILENAME ::= .. path to file (absolute or relative to current file) ..
<netlist> ::= ( <function> | <include_instruction> )*
<include_instruction> ::= '@i' <module_name> '"' FILENAME '"'
<function> ::= <prototype> '{' <body> '}'
<prototype> ::= <output> <output>* '=' <function_name> <input>*
<body> ::= <instruction>*
<instruction> ::= <output> <output>* '=' <function_name> <input2>* ';'
<input2> ::= <input> | <inline_instruction>
<inline_instruction> ::= '(' <function_name> <input2>* ')'
<output> ::= NAME
<function_name> ::= NAME
<input> ::= NAME
<module_name> ::= NAME
```

It is also possible to add comments.

Example netlists

Full adder

```
s r = main a b c
{
  s = xor (xor a b) c;
  r = or (and a b) (and (xor a b) c);
}
```

Minus

```
y = main x
{
  y = xor x c;
  c = reg (or x y);
}
```

Modulo 2 counter

```
s r = main x
{
  s = reg (xor x s);
  r = and x s;
}
```

2-cycle clock

```
o = main
{
  o = reg c;
  c = not (reg o);
}
```

Modulo 24 counter

```
s r = cm2 x reset
{
  s = reg (and (xor x s) (not reset));
  r = and x s;
}

a = and6 a1 a2 a3 a4 a5 a6
{
  a = and (and (and a1 a2) a3) (and (and a4 a5) a6);
}

s1 s2 s3 s4 s5 r = cm32 x reset
{
  s1 r1 = cm2 x  reset;
  s2 r2 = cm2 r1 reset;
  s3 r3 = cm2 r2 reset;
  s4 r4 = cm2 r3 reset;
  s5 r  = cm2 r4 reset;
}

s1 s2 s3 s4 s5 r = main x
{
  s1 s2 s3 s4 s5 r1 = cm32 x r;
  r = and6 s1 s2 s3 (not s4) s5 x;
}
```

General parsing strategy

- The parsing is done in one pass.
- Tokens are read one by one and data structures are created and filled progressively.
- Special characters structure the code, but are not reserved. They can be used for anything if there is no ambiguity in the context.

Obfuscated netlist

```
and=;and{}1=(){1=;(and(reg;)main);main=1;;=and)1;}(==}{(=(};})=main={)===;}
```

The templating language

To avoid lots of cut-and-paste, we wrote a very simple templating system.

Example template code

```
[i:0->3|[j:3->0|x_{i}_{j+1} ]\n]
```

Expansion result

```
x_0_4 x_0_3 x_0_2 x_0_1
x_1_4 x_1_3 x_1_2 x_1_1
x_2_4 x_2_3 x_2_2 x_2_1
x_3_4 x_3_3 x_3_2 x_3_1
```

General evaluation strategy

- For evaluation, we will need a *dependency table* which will indicate how outputs depend from inputs (directly, through a register, or not at all).
- We compute these tables recursively (going through the tree of function calls), so we can use the tables of smaller functions to compute the tables of bigger functions.
- During this pass, we also search for cycles, and report them.

Precise strategy

- We go through the blocks within a function (DFS), going from outputs to inputs, and mark them.
- If we encounter a marked block, it means that we have found a cycle.
- When inputs depend from outputs through a register, we add them to a queue, to run a DFS starting from this input (after having unmarked everything). This allows both to compute output-input dependencies for the current function, and to find cycles hidden behind a register.
- If a cycle is found, it is reported (in a very verbose way) and the simulator halts.

Example

Netlist

```
a b = f c d
{
  a = reg c;
  b = or c d;
}

z = main y
{
  z2 = and z z;
  x1 x2 = f y z2;
  z = and x1 x2;
}
```

Reporting

```
Error: [checker] reporting cycle in function "main":
Error: [checker]   'z' is the 1st input of a block of type "and" to which 'z2' is the 1st output.
Error: [checker]   'z2' is the 2nd input of a block of type "f" to which 'x2' is the 2nd output.
Error: [checker]     Here is why the 2nd input and 2nd output of "f" are connected:
Error: [checker]     'd' is the 2nd input of a block of type "or" to which 'b' is the 1st output.
Error: [checker]   'x2' is the 2nd input of a block of type "and" to which 'z' is the 1st output.
Error: [checker] cycle found in function "main"
```

General evaluation model

- Evaluation is done from outputs to inputs.
- When a block is encountered with a user-defined function, we evaluate its inputs, copy them in the definition of the function, evaluate the output, and copy its value back.
- Outputs are managed separately because of some annoying cases.

Two-pass system

- The simple strategy presented here does not work if there are cycles involving registers. If the output depends on input through internal registers, these inputs can depend directly from the output!
- Our solution is to use a two-pass system.

First pass

We just want to find the value of the output. So we evaluate inputs from which the output depends directly, and we evaluate the output, popping values from registers.

Second pass

Now, the output value is known, and we have to update the registers. So, we evaluate all inputs, and we ask for all registers to get their new value.

Optimizations

- On simple tests, the simulator works at a satisfying speed.
- However, the simulation is too slow for big circuits. To simulate the processor, the design presented so far runs at approximately two cycles per minute, even with gcc optimization flags.
- Hence, we used some optimization techniques to make the simulation faster.
- This allows us to reach speeds of roughly 600 cycles per second.
- Things would probably be much faster if we unfolded the whole circuit (ie. replacing functions by their body). Not unfolding the graph saves some memory, but wastes time.
- The optimizations are presented in chronological order.

Memoization tables

- For functions with a small number of inputs and outputs, we can speed things up with a table to store their output values.
- A global constant defines the maximal size of the tables.
- Tables are generated on the fly: when we compute a value for a memoized function, we add it to the table. However, if the value is already known, we do not compute it.
- Registers are handled as special input and outputs.

→ This optimization makes things about **3 times** faster.

RAM-ROM primitives

- Despite our initial reluctance to do so, we implemented RAM and ROM primitives in the simulator to replace our RAM-ROM netlist.

→ This optimization makes things about **60 %** faster.

Avoid useless reevaluations

- Since we only have one copy of each function, we must erase all the intermediate values when we evaluate another occurrence of the function.
- If we switch occurrences back and forth for every output, this wastes a lot of time.
- The naive version erases a function each time it evaluates an output, even if it is the same occurrence. This wastes intermediate results and makes things very slow.
- The solution is to store, in each block, the last cycle when it was evaluated, and the last occurrence. Thus, we only erase things when necessary.

→ This optimization makes things about **500 times** faster!

Optimized dependency lists

- When searching for inputs for which an output depends, adjacency lists make things faster than adjacency matrices.

→ This optimization makes things about **10 %** faster.

Partial unfolding

- There is no memory penalty if we unfold functions with only one occurrence in the whole circuit.
- Moreover, this speeds things up (it makes the circuit simpler for the simulator).

→ This optimization makes things about **30 %** faster.

Dirty tricks

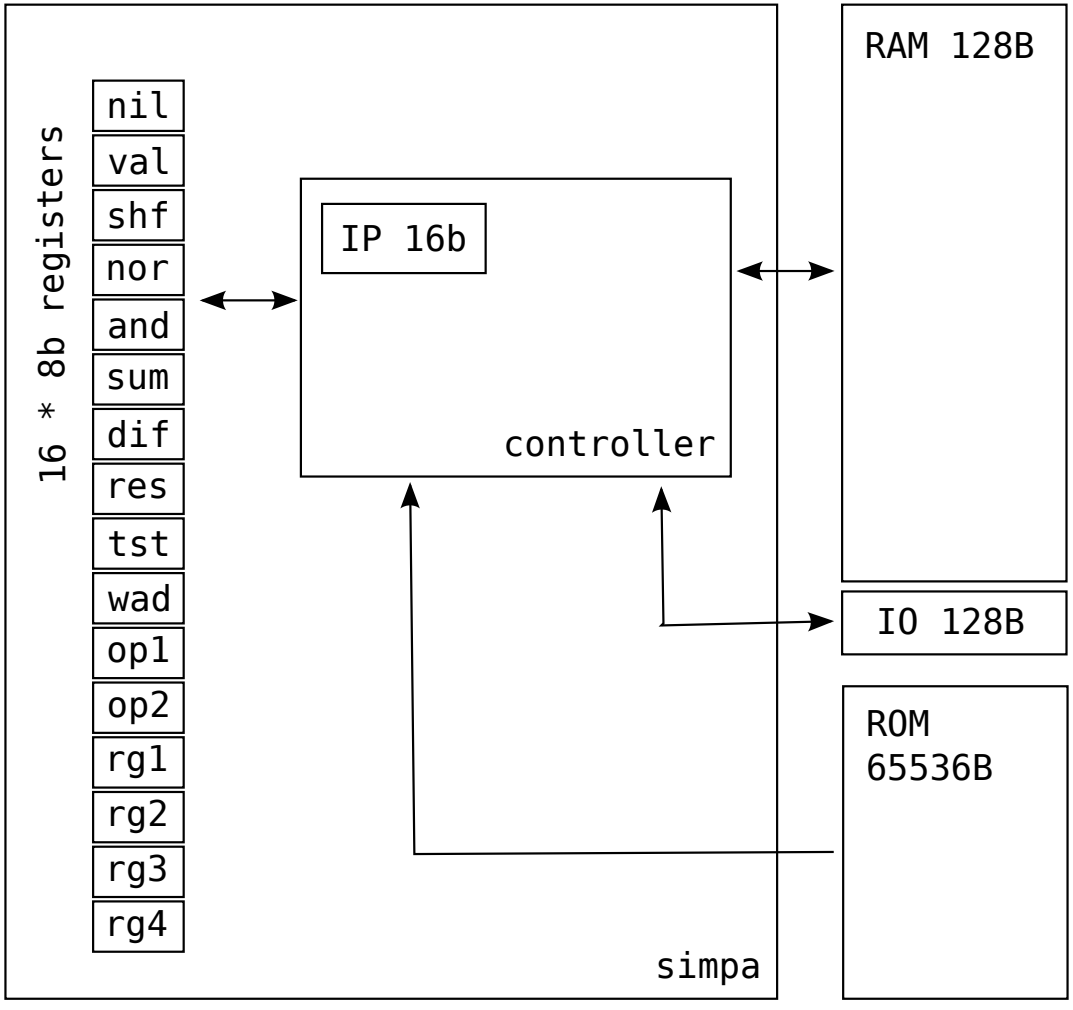
- With the C preprocessor we can replace generic accessor functions by macros, and do other such things.

→ This optimization makes things about **50 %** faster.

Some ideas we had

- A MIPS-like processor to use our own compiler?
  - → ... not really original enough.
- A processor with lambda calculus and functional programming concepts (like a Lisp machine?).
  - → ... fun but way too complicated to implement.
- At the end of the day, we decided to create a SISC (a Single Instruction Set Computer, that is).
  - → ... sound cool, and doesn't seem that hard!

Overview


16 * 8b registers
nil
val
shf
nor
and
sum
dif
res
tst
wad
op1
op2
rg1
rg2
rg3
rg4
IP 16b
controller
simpa
RAM 128B
IO 128B
ROM
65536B

Architecture

- **16** 8-bit registers:
  - **8** operation registers
  - **2** action registers
  - **2** operand registers
  - **4** storage registers
- **128** 8-bit input-output peripherals
- **128** 8-bit RAM cells
- **65536** 8-bit ROM cells
- **1** instruction (taking **2** registers as arguments)

Registers

`nil`	is always zero.
`val`	is the last instruction with bits swapped in this order: 4, 0, 1, 2, 3, 5, 6, 7.
`shf`	is op1 shifted by the number of '1's in the 7 LSBs of op2, shifting direction depends on the MSB.
`nor`	is `nor op1 op2`.
`and`	is `and op1 op2`.
`sum`	is `(op1 + op2) % 256`.
`dif`	is `op1 - op2` or 0 if the result is negative.
`rsl`	is what is read in RAM or on IO at address op1.
`tst`	is always reset to zero; the IP jumps to `(op1 << 8) + op2` if non-zero.
`wad`	is always reset to zero; op2 is written in RAM or IO at the address specified here.
`op1-2`	are the two operators.
`rg1-4`	are not special

...except `rg1` for being the first non-special one.

The *magic* instruction

- It takes two registers in parameters (`a` and `b` thereafter).
- It performs the following actions, in that order:
  1. the content of `a` is copied into `b` ;
  2. registers 0 to 6 are updated.
  3. the byte read in RAM or IO at address `op1` is copied in `rsl` ;
  4. byte in `op2` is written in RAM or IO at address `wad` ;
  5. `wad` is reset to 0 ;
  6. the `IP` jumps to `(op1 << 8) + op2` if `tst` is non-zero ;
  7. `tst` is reset to 0.

The simpasm assembly language

It offers the following functionalities:

- registers name are replaced by their binary equivalent.
- numerical constants are replaced by whatever instruction is needed to have them loaded in `val`.
- instructions "<LABEL" and ">LABEL" are replaced by their numerical constant to be loaded in `val`; the first one is for the 8 LSBs of the address of `LABEL`, the second one is for the 8 MSBs.

The assembler is written in Perl.

Simulator

For testing purposes, we wrote a processor simulator in Perl. It runs at around 26000 cycles per second, and is able to offer detailed debug output :

- state of the registers at each cycle
- IO/RAM read and write (address and value)
- jump notification

Netlist

```
@i _ "utilities.nls"
@i _ "controller.nls"

[n:1->6|[b:0->7|out_{n}_{b} ]] out_0_7
= main [n:0->127|[b:0->7|in_{n}_{b} ]]
{
  [b:0->7|instr{b} ] = rom [n:0->15|addr{n} ];

  [n:0->15|addr{n} ] [b:0->7|outval{b} ]
  [b:0->7|waddr{b} ] [b:0->7|raddr{b} ]
  = ctrl [b:0->7|instr{b} ] [b:0->7|inval{b} ];

  [b:0->7|inval{b} ] [n:0->127|[b:0->7|out_{n}_{b} ]]
  = io [b:0->7|outval{b} ] [b:0->7|raddr{b} ]
       [bb:0->7|waddr{bb} ] [n:0->127|[b:0->7|in_{n}_{b} ]];
}
```

The sasc watch

The sasc watch is coded in simpasm. Here are the big steps:

1. The ram is initialized with required values.
2. Seconds, minutes and hours are initialized to 0.
3. The time values are written in the corresponding IO.
4. An IO bit is set to 1 to tell the displayer to update the time.
5. Seconds are incremented. If < 60 then goto step 9.
6. Seconds are reset and minutes incremented. If < 60 then go to step 9.
7. Minutes are reset and hours incremented. If < 24 then go to step 9.
8. Go to step 2.
9. Wait the proper amount of cycles to have every time update lasting the same number of cycles, then goto 2.

Output format

The output consists of 6 bytes for the time (each byte is one of the 7 segments encoded digit) and one bit to notify when the time is updated.
The latter is useful to update the time display without artifacts.

Displaying the time with asciifee

asciifee is a really small program coded in C using ncurses that display time using ASCII art in a terminal.

- asciifee reads 49 bits on its stdin.
- If the last one is 0 do nothing.
- If not, change the 7 segments value according to the bits read and then refresh the window.