Conception and realization of the VIVACE architecture

PROJET DE SYSTÈME DIGITAL

A.Auvolat, E.Enguehard, J.Laurent

17 janvier 2014

The VIVACE[1] architecture is a minimalistic 16 bits RISC microprocessor architecture, largely inspired by the MIPS microprocessor.

The principal characteristics of the architecture are :
- *8 general-purpose registers*, which can hold 16 bits integers : `Z, A, B, C, D, E, F, G`
- *16 bit memory addressing*, enabling the CPU to use up to 64*kb* of memory.

In order to implement and run the architecture, the following programs have been written :
- *Netlist simulator* and *netlist optimizer*
- An *OCaml library* for generating netlists from Caml code
- The code for the *CPU implementation* (written in Caml)
- A *monitor* which is used to interact dynamically with the netlist simulator
- An *assembler* which can be used to produce the ROM files run by the CPU.

1 How to run the VIVACE cpu

1.1 Preparation

All the tools described in the introduction must first be compiled :

```
$ cd csim; make; cd ..
$ cd sched; make; cd ..
$ cd monitor; make; cd ..
$ cd asm; make; cd ..
```

To run the VIVACE CPU, type the following :

```
$ cd cpu; make
```

1.2 Monitor commands

You are now running the VIVACE CPU. The monitor accepts a few commands to control the simulation. First, you must configure the monitor to communicate with the CPU. Type :

```
t 0
s 1 19 18
d7 20 21 22 23 24 25 26 27
```

The first command sets up the tick input (a tick is sent once every second on this input by the monitor). The second command sets up the serial input/output. The third command sets up the 7-segment display (8 digits displayed). Now, use the following commands to control the simulation :

1. Virtually Infaillible VIVACE Automated Computing Environment

— `a` run the simulation at full speed
— `m` run the simulation step by step (enter an empty command to run a step)
— `f <freq>` run the simulation at fixed frequency (frequency is dynamically ajusted so this is not very accurate)
— `q` exit simulation

The CPU recieves commands on the serial input. To send a command to the CPU, use the following syntax :

```
:<cpu_command>
```

For instance :

```
:Y2014
```

These commands are essentially used to set one of the six variables `YMDhms` ; the syntax is similar to the example command given above. An empty CPU command tells the CPU to just tell us what time and what day it is.

2 Program details

2.1 Generating netlists from Caml code

We have developped a library that enables us to easily generate netlists from Caml code. The Caml code we write has the same abstraction level that MiniJazz has, but it is more comfortable to write circuits like this than with MiniJazz code.

The library functions are defined in `cpu/netlist_gen.mli`. Basically, we have created functions that build the graph of logical operations. The abstract type `t` is actually a closure to a function that adds the required equation to a netlist program being built, therefore the generation of a netlist consists in two steps : the generation of a closure graph that describes the graph of logical operations, and the execution of these closures on a program which, at the beginning, has only the circuit inputs. The equations are progressively added to the program when the closures are called.

The VIVACE CPU has been entirely realized using this library.

2.2 The VIVACE CPU

2.2.1 Control structure

The CPU is able to execute instructions that need several cycles to run. The two first cycles of an instruction's execution are used to load that instruction (16 bits have to be read, ie two bytes). Most instructions finish their execution on the second cycle, but some executions need more cycles to run :
— Load and store instructions need one or two extra cycles
— The multiplication operation needs as many cycles as the position of the most-significant non-null bit in the second operand.
— The division always runs on 16 cycles.

The execution of instructions on several cycles is implemented using a "control bit" that cycles through several steps : load instruction, various steps of instruction execution. A few of these step control bits appear in the simulator, as CPU outputs :
— `read_ilow`, `read_ihi` CPU is reading low byte/high byte of the instruction
— `ex_instr` CPU begins execution of the instruction
— `ex_finish` CPU finishes execution of the instruction (modified registers may only appear in the monitor at the next step)

2.2.2 ROM, RAM and MMIO

The CPU has uniform acces to a 64kb address space, which contains the ROM (`0x0000-0x3FFF`), MMIO (`0x4000-0x7FFF`) and the RAM (`0x8000-0xFFFF`). The `cpu_ram` (`cpu.ml`) subcircuit is basically a bunch of multiplexers that redirect reads and writes to the correct places.

The serial input/output is implemented using one input and two outputs :

— Input `ser_in` (8 bits) : when this input is non-null, the character entered is buffered by the CPU. This buffer can be read by reading MMIO byte at address `0x4100`. The buffer is reset to zero on read.

— Output `ser_in_busy` (1 bit) : signals when the input buffer is nonzero (ie a character is pending, waiting for the CPU to read and handle it).

— Output `ser_out` (8 bits) : when non-null, the CPU is sending a character to the serial output. This output can be written by writing MMIO byte at address `0x4102`.

The clock is also handled by MMIO : the CPU recieves a tick every second on input `tick`. When a tick is recieved, the tick buffer is incremented by one. This tick buffer can be read by reading MMIO word at address `0x4000`. When the word is read, the buffer is reset to zero.

The 7-segment display is also handled by MMIO : the 8 digits can be modified by writing a byte to MMIO addresses `0x4200` to `4207`.

2.2.3 The ALU

2.3 The assembler

2.4 The simulator and the monitor

The simulator is written in C for performance reasons.

The monitor is a C program, using the curses library for output to the console.

The simulator and the monitor communicate via Unix named pipes (FIFO's), which are created in the files `/tmp/sim2mon` and `/tmp/mon2sim`. The synchronization of the two programs has somewhat been problematic, due to incorrect use of `scanf` making the programs hang.

2.5 The operating system

3 Results and benchmarking