Digital System

Microprocessor project

Fabrice Ben Hamouda, Yoann Bourse, Hang Zhou

2009-2010 : Semestre 1

Abstract

This paper describes our conception of a microprocessor, for the "Systeme Digital" course. We will emphasize on the three main parts of the project, that is to say the netlist simulator, the assembly translator, and finally the microprocessor itself. Our primary aim was to be able to simulate a simple digital watch on this microprocessor. Furthermore, we chose to develop a microprocessor close to the MIPS architecture so that we could use the compiler program we have done for another course in order to compile Ocaml directly on our microprocessor.

Contents

Part I
Netlist simulator

This section describes the first step toward the simulation of a digital watch. The final result will be based on a microprocessor that we will have created, and which will require a netlist simulator, that is to say a program able to interpret and simulate logical circuits.

1 Simulator overview

1.1 Global description

This simulator was built in C++, mainly because this project immediately appeared to us under the form of classes and pointers. Since the beginning, for purposes of performance, we wanted our simulator to have two main modes : interpreting the data or compiling it. In the following subsections, you will see how we translated circuits into text, how we treated this text by C++ programs and how we configured the two modes of output.

1.2 Netlist language

We chose to use a language to describe netlists similar to the one displayed in the examples given in the description of the project. The main motivations for this choice were the clearness and the simpleness of the given language. Moreover we wanted to ease the reading of our work by the writer of the project's description. Finally, this syntax presented obvious advantages for the declaration of variables and user-defined functions. You can see below an example of our syntax being used. Note that there are a few technical specificities for practical purposes : the use of the keyword inst, the fact that a closed bracket } cannot be followed by anything. But you may skip as many lines as you want between functions.
Note that our simulator is based on the following set of logical basic instructions which provide a base to the boolean algebra : NOT, OR, AND, XOR, and the register. Register and negation have higher priority than the binary operations, and **there are no other priority rule**, so use brackets in order to avoid interpretation errors. Default interpretation is from left to right.

```
s,r = FullAdd(a,b,c){
s = a^ b^ c
r = (a & b) — ((a^ b) & c)
}
# Commentaries can be added by #
# ^ means logical XOR
# & means logical AND
# — means logical OR

o = clk2(){
o=Z(c)
c=~ Z(o)
}
# ~ means logical NOT
# Z represents a register
# Note that the brackets {} are used to introduce a customized function which can be used later with the label “inst”, meaning that you instantiate an occurrence of such a circuit.

b,c = example (a1,a2,a3){
inst b,c = FullAdd(a1,a2,a3)
}
```

We will also have to specify the value of several input variables. The syntax we chose is similar to an array whose columns are separated by a tabulation. It is possible to reproduce a line a certain amount of times using the symbol *, or skip a cycle by an empty line (therefore every empty line creates a new cycle with no forced modification). You may also try to force the value of a variable that is not an input. It will take this value of the beginning of the cycle but may be overwritten by some other functions. Below stands an example :

```
a1   a2   a3
0    0    1
0    1    0
*5
1    1    0
```

1.3 Files, classes and main functions

1.3.1 Input files

- **input.net**
  This files contains the netlist description of the used circuits. You must define custom circuits before instantiating them.

- **input.set**
  This files contains the default number of clock cycles, and the values of customized variables for each cycle.

1.3.2 Programming

- **main**
  This file is the core of the simulator, as in any C++ project. It loads the various modules (described below) whenever needed. It defines the two possible ways to use the simulator : the compilation or the interpretation. It is also in this file that while loops represent clock cycles.

- **tools**
  Here are defined several tools, especially to manipulate vectors more easily.

- **parserVar**
  This module creates an item of the settingfile class which can be seen as a flow to read the file "settings.net".

settingfile

The variable **s** contains the source file under the form of a stream.
Variable repeat record the number of line repetitions yet to come.
init_settings initializes the global settings at the beginning of the program.
forced_var is a function called every cycle in order to define the value of given variables.

- **parserNetlist**
  This file analyses input.net. The parser uses **step** variables to memorize its position in the syntax analysis : (*see the doxygen documentation for more informations*)

Creating a function :
0 -outputs- = 1 -function's name- (2 -inputs-) 3 -waiting for the bracket- {
4 -function core-
}

Instantiating a function :
-1 : no instantiating function
-2 : receive output of non instantiating function

The function **ReadFunction** analyses the text of a function : il calls the functions **ProcessExpression** which analyses a logical expression, and the functions **ProcessName** which are the core of the creation of the various functions. They are all subfunctions of **ParserNetList**, which handles as well the topological sort of the functions thanks to the subfunction **SortlistOps**.

- **graph**
  The name of this file is due to the graph structure induced during the parsing, whose consequences are present all over the program.

variable

This class represents a variable, identified by a **name and a value**. We register the list of operations of which the variable is an input and we define several functions to read or modify the variable's parameters.

registerV

This class is our mean to deal with the problem of registers : every registerV contains **two pointers towards "input" variable and "output" variable** : the function **SaveRegister()** loads the value of the input and the function **NextClockEdge()** copies it into the output. The **internal value of the register** allows an easy instantiation of functions : only registers are copied, not the combinatorial graph.

operation

An operation is the heart of a function. Every object of this class contains **two variable* arrays pointing to the inputs and outputs** of the operation, and a **functionInst pointer** heading towards the function the operation is used in.

- **functionUser**

functionUser

This class is the global one, defining every user-customized function of which the various circuits will be an instance. It contains global parameters shared by all instances of a function such as **input, outputs, registers** and **the list of operations sorted topologically**.

- **functionInst** This is a leftover of a previous version of our project, still accessible through the flag `--nogc` *(no graph copy)*. Previously, functionUser was nothing but a generic pattern, and real circuits were instances of functionInst, which used the pattern in order to compute. Yet, this system had a major problem with combinatorial loops when the output of a function was connected to one of its inputs, even through a register. In order to correct that shortcoming, we decided that a circuit would not be an instance of functionInst, but a copy of functionUser to the general functionUser which represent the whole circuit.

1.4 Executable file and flags

We allowed the use of flags in order for the user to set various option or parameters when calling the executable file resulting from the compilation of our simulator. Following syntaxes are accepted :

```
simulateur [options] netlist
simulateur -h (for help and a total list of possible flags)
```

where main options are :
`-v` : displays comments about what the program is doing (verbose)
`-q` : hides the simulation results
`-p period` : length of a clock cycle (in seconds)
`-c` : compiling mode instead of interpreting
`--cc bin` : use "bin" compiler instead of gcc `-s settings` : path to the setting file
`-n n` : number of cycles
`--column s` : size of a column
`netlist` : path to the netlist file
`--noits` : no intermediate toposort (only final)

1.5 Interpreting or compiling ?

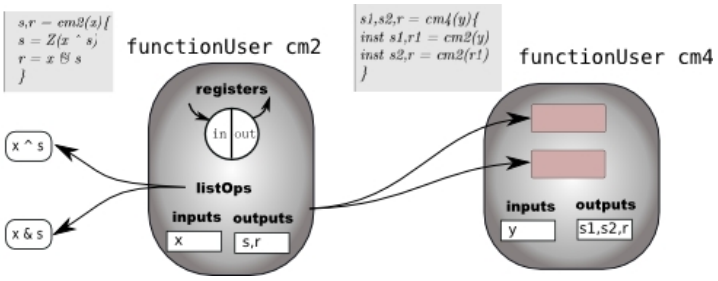
The compiling mode transforms the given circuit in a C program : sim.C and then compiles it. This way, the circuit is an independent program and therefore a lot faster. On the other hand, it is impossible to specify or to force the value of variables with a setting file. Here is an overview of the two processes :

Compiling :
- Variables declaration (*CPutDeclarations*)
- Combinatorial computing (*CPutCommands*) whole users functions are copied, not only registers.
- Save of registers (*CPutRegistersSave*)
- Next clock, restoring registers (*CPutRegistersNext*)

Interpreting :
Every time an operation invoilving a user-created function is called :
- Next clock, restoring registers (*NextClockEdge*)
- Combinatorial computing (*Eval*)
- Save registers (*SaveRegisters*)

Note that as mentioned above, there is no longer several user-created functions : they are all embedded in a global function representing the overall circuit, according to the following scheme.

1.6 Functioning scheme



2 Makefile & Compilation

2.1 Getting started

On Linux, open a command line console (in "simulateur" folder) and type :

- `make 12345`
- `make watch` : you will see a watch without microprocessor

To use compiling mode, you need GNU gcc compiler.
On Linux, just add option `SIM_NS_FLAGS=-c`.
Note that with `--cc` option you can specify options to gcc, for example optimization option like `-O3` or `-O2`.

If you want, you can use CodeBlocks : `http://www.codeblocks.org`. There is a CodeBlocks project. Just open it.

2.2 Targets and options

We have created a Makefile to automatically compile the project and transform all *.php files (*automatic generation of complex netlists, see 4.5*) into netlist **.net* files and to simulate them using if possible **.mem* and **.set* files. Important targets are :

- `all` : compile the project
- `doc` : compile this documentation
- `doxygen` : compile the doxygen documentation
- `demo` : execute all netlist
- `12345` : execute required for project (Full Adder, Minus, ...) netlist
- `watch` : print the watch (add some options to simulator to select the good period and the watch output format)
- `netlists/%.run` : execute the netlists/%.net or netlists/%.php netlist
- `democlean` : delete all generated netlists
- `clean` : delete all generated files except documentation and "simulateur" executable
- `mrproper` : delete all generated files without any exception

There are a lot of options, you can see the Makefile to have more informations :

- `TEST=yes` : compare results of simulation with saved results (in **.res* files)
- `TEST=res` : automatically generate **.res* files (you have to verify that the content is correct)
- `SIM_FLAGS` : flags for simulator
- `SIM_NS_FLAGS` : flags for simulator for target that do not use a setting file (we use it for testing the compilation mode : `SIM_NS_FLAGS=-c`)

2.3 Requirements and troubleshooting

In order for the makefile to work correctly, you will need to have installed support of **PHP**.

2.4 Speed of simulator

To have an approximative idea of the speed of our simulator, you can execute the watch without forcing the period :

- `time make netlists/watch.run SIM_FLAGS=‘‘-n 1000000 --watch’’`
- `time make netlists/watch.run SIM_FLAGS=‘‘-n 1000000 --watch -c’’`

The `time` command print the time needed to execute the simulation. You can see that compiling mode is faster than interpreting mode. There is an overhead at the beginning because of time of compilation but after that it is very fast. In addition, for real tests, it's recommended to print nothing to stdout (because it takes a lot of time) with flag `-q` for simulator and to use clock cycles.

3 Simulation results

3.1 Theoretical considerations about required set of tests

For the following circuits, at each cycle $t \in \mathbb{N}$, we call state the list $(z_1(t), ...z_m(t))$ of values of output of registers (z_k) at the beginning of the cycle. At each cycle, inputs $(i_1(t), ..., i_n(t))$ and state completely define the value of others variables since others variables only depend on value of inputs and of output of registers. In particular, state and inputs define outputs (let's set $(o_1(t), ..., o_r(t))$ be the vector of output value at cycle t) and next output of registers $(z_1(t+1), ..., z_m(t+1))$ because the next output of a register is exactly current value of input of this register. More precisely, there is one function f :

$$f : \{0,1\}^{n+m} \to \{0,1\}^{r+m}$$

so that the following condition is true :

$$(o_1(t), ..., o_r(t), z_1(t+1), ..., z_m(t+1)) = f(i_1(t), ..., i_n(t), z_1(t), ..., z_m(t))$$

States are initialized that way :

$$(z_1(0), ..., z_m(0)) = (0, ..., 0)$$

We notice that if inputs are constants (or if there is no input), $\vec{z}(t+1) = (z_1(t+1), ..., z_m(t+1))$ only depends on $\vec{z}(t) = (z_1(t), ..., z_m(t))$ and on the constant inputs. Since there are only 2^m possibilities for the values of a vector in $\{0,1\}^m$, the sequence $(\vec{z}(t))_{t\in\mathbb{N}}$ is periodic with a period less or equal than 2^m. In addition, if at cycle t and cycle t' $(t' \neq t)$, states are identical, the period is less than $t' - t$.

In particular, you can notice that if $\vec{z}(t) = \vec{z}(t+1)$ for an cycle t, the state is never modified when the circuit is in the state $\vec{z}(t)$ and when input is $i(t)$.

3.2 Full Adder

```
s,r = FullAdd(a,b,c){
s = a^ b^ c
r = (a & b) — ((a^ b) & c)
}
```

a	b	c
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
*9		

Simulation results :

	s	r
0	0	0
1	1	0
2	1	0
3	0	1
4	1	0
5	0	1
6	0	1
7	1	1
...		

This circuit doesn't involve any register : any output will require no more than one clock cycle in order to be computed. Therefore, we only need to try all the possible combinations for a, b and c (provided by magic masks).

3.3 Minus

The minus function is equivalent to negate the input and then add one to it. This means that the first 1 (the least important one) will keep the value 1. All the 0 before will not change, and after it, 0 will be turned into 1 and 1 into 0. That's the property we will have to check on our circuit.
...100$\underline{1}$00 → negation : ...011$\underline{0}$11 → incrementing : ...011$\underline{1}$00
In order to prove this function, we modified the netlist so as to print the output of the register (the

only state of the circuit). As long as input is $x = 0$, the state c is 0 (constant input). An input $x = 1$ changes the state c to 1 and output $y = 1$. After that, all bit are changed to their opposite and that never changes the state c (because $c(2) = c(3) = 1$ for input $x = 0$ and $c(3) = c(4) = 1$ for input $x = 1$). We have therefore proved correctness of the circuit with the 5 first lines of simulation results.

```
y,c = minus_b(x){
y = x ^ c
c = Z(x—y)
}
```

x
0
1
1
0
1
∗12

Simulation results :

	y	c
0	0	0
1	1	0
2	0	1
3	1	1
4	0	1
...		

3.4 Counter modulo 2

We need to check that the counter only counts clock cycles when x=1, and that $s + 2r$ corresponds to the number of such cycles, modulo 2. With complete simulation, you can see that when the input is $x = 0$, this do not change the state (because $s(0) = s(1) = 0$ and $s(2) = s(3) = 0$). Hence the number of cycle with input 0 between two cycles with input 1 has no importance. There is only left to verify that, in all states ($s = 0$ or $s = 1$), when input is 1, counter increments itself (s being the main digit and r the overflow carry) and it is the case. The 5 first lines of simulation results are sufficient to prove correctness of the circuit.

```
s,r = cm2(x){
s = Z(x ^ s)
r = x & s
}
```

x
1
0
0
1
0
1
∗11

Simulation results :

	r	s
0	0	0
1	1	0
2	1	0
3	1	1
4	0	0
5	1	1
6	0	0
7	1	1
...		
...		

3.5 Clock divider

This circuit doesn't need any input (only the specification of the number of clock cycles to execute). we need to verify that it produces a periodic output. Hence, we just need to check $2^2 = 4$ values to be sure that it is a clock divider : first 4 lines are enough.

```
o = clk2(){
o=Z(c)
c=~ Z(o)
}
```

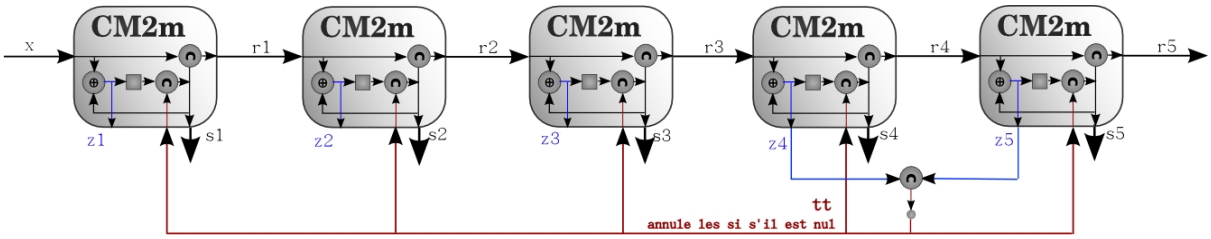
∗16

Simulation results :

	o
0	0
1	1
2	1
3	0
4	0
5	1
6	1
7	0
8	0
...	

3.6 Counter modulo 24

This counter is based on a counter modulo $32 = 2^5$ that is to say 5 CM2 in a row. We used the fact that the CM2 contain actually two different numbers : the current number in outputs s_i and the next number to come, in z_i, just before the register. The difficulty to this counter is to reset the outputs when the counter reaches 24. We decided to use an additionnal feedback condition able to reset all s_i, that is to say that we intercept the future number to come when it is $\ldots 0011000 = 24$ and replace it with $\ldots 00000 = 0[24]$.



For purposes of legibility, we detailed the first part of the input (that is to say *1 / *23 / 0 / *5*).

```
s,r,z = cm2m(x,v){
z = x ^ s
s = Z(z) & v
r = x & s
}

s5,s4,s3,s2,s1 = cm24(y){
tt = ~ (z4 & z5)
inst s1,r1,z1 = cm2m(y,tt)
inst s2,r2,z2 = cm2m(r1,tt)
inst s3,r3,z3 = cm2m(r2,tt)
inst s4,r4,z4 = cm2m(r3,tt)
inst s5,r5,z5 = cm2m(r4,tt)
}
```

```
x
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
0
0
0
0
1
*27
```

Simulation results :

	s5	s4	s3	s2	s1
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0
9	0	1	0	0	1
10	0	1	0	1	0
11	0	1	0	1	1
12	0	1	1	0	0
13	0	1	1	0	1
14	0	1	1	1	0
15	0	1	1	1	1
16	1	0	0	0	0
17	1	0	0	0	1
18	1	0	0	1	0
19	1	0	0	1	1
20	1	0	1	0	0
21	1	0	1	0	1
22	1	0	1	1	0
23	1	0	1	1	1
24	1	0	1	1	1
25	1	0	1	1	1
26	1	0	1	1	1
27	1	0	1	1	1
28	1	0	1	1	1
29	0	0	0	0	0
30	0	0	0	0	1
31	0	0	0	1	0
...					
52	1	0	1	1	1
53	0	0	0	0	0
...					

4 Further Improvements

4.1 Diagram display

To improve the legibility of the outputs, we have imagined another display of the output : instead of 0 and 1, the **--diag** flag causes the output to be a frieze where 0 is bottom and 1 is top, as displayed on the example.

4.2 The time parameter

In order to simulate the quartz periodic signal for the microprocessor, we have decided to allow the user to set the duration they want for a clock cycle. This is done thanks to an additional loop and the use of the function gettime of day.

4.3 Boolean constants

We added the possibility to use boolean constants 0 and 1 directly in the netlist, therefore avoiding the need to declare an alimentation like VDD.

4.4 RAM/ROM primitives

As a transition toward the work that is simulating a whole microprocessor, we have added to our project ROM/RAM primitives. You can instantiate ROM or RAM with the following code :

ram $d_{o1}, d_{o2} = name(a_{o2}, a_{o1}, a_{o0}, w, a_{i2}, a_{i1}, a_{i0}, d_{i1}, d_{i0})$
rom $d_{o1}, d_{o0} = name(a_{o2}, a_{o1}, a_{o0})$

First character : Data or Address
Second character : In or Out, that means for writing (in) or for reading (out)
Third character : level of bit (2 being the most significant bit)
w : write input data into input address if equal to 1

Size of address bus and data bus are determined by the number of arguments and returned values. More precisely, if there are n returned values and m arguments, data bus is n bits length and address bus is $m-n$ bits length for ROM and $\frac{m-n-1}{2}$ bits length. The order of argument is this mentioned above. Size of memory is 2^a if address bus has a bits. Names of variables have no importance.

Reading is executed before writing. You can also define RAM and ROM through a **.mem* file using the flag : `-m file`. The content of the file should follow that pattern, with the keyword **mem** :

```
mem:Name
3
2
12
mem:Name2
...
```

This uses a special functionInst called **functionMem**, which stores the data in a **data** vector.

4.5 Circuit-based digital watch

The conclusion of this simulation work was the simulation of a digital watch based on complex circuits as describes in the courses provided by Mr Vuillemin. The netlists ("simulateur/netlists/watch.php") of those gigantic circuits were created thanks to **PHP scripts** which offered a very clear way to display text in loops or recursive calls. It offered a nice transition towards the microprocessor-based digital watch, with for instance the creation of a graphic interface based on 7-segments display, activated by the flag **--watch**. This subject will be later discussed in our next report.

5 Doxygen documentation

A detailed documentation is available in the folder "simulateur/doc", describing more precisely every file, class and function. Just type `make doxygen` to compile it.

Part II
Assembly language and microprocessor specification

We will now describe the specifications and features of the microprocessor we will be using to simulate a digital watch, thanks to the netlist simulator previously conceived. Assembly programs will be translated into digital instructions that the microprocessor will interpret in order to use correctly logical circuits for computation. The translation from assembly to machine language is performed by a program which is also able to simulate the expected results of the assembly code.

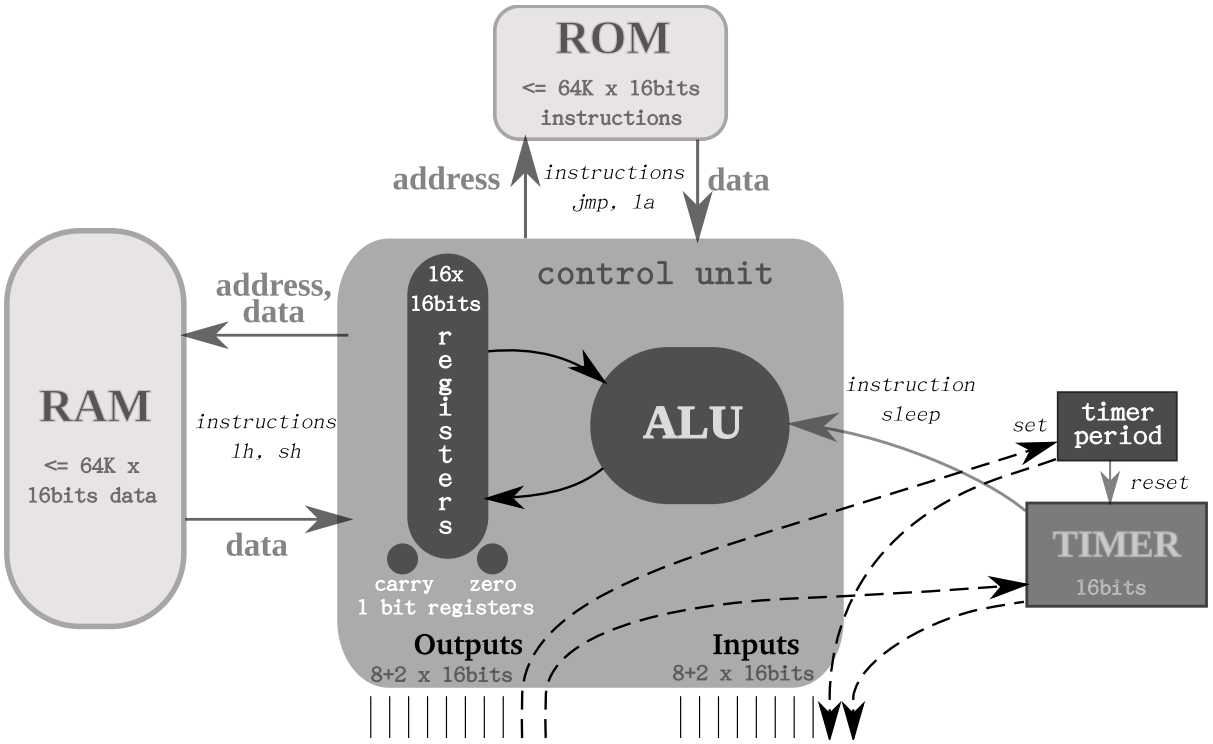
6 Overview

The main features of our microprocessor are :

- 16 registers of 16 bits
- one 16 bits clock timer
- 16 bits instructions
- one instruction per clock cycle
- up to 64 K instructions (ROM)
- up to 2×64 KB of data RAM (16 bits data length)
- up to 8 (*+2 for the timer*) output ports and 8 (*+2 for the timer*) input ports of 16 bits

We chose to develop an architecture close to MIPS, because that is the microprocessor we use in the course "Langage de programmation et compilation". Therefore, we hope to be able to combine this handmade microprocessor with the compiler we have to build for this course, so that we will have traced the whole path from CAML instruction to circuit execution. Some modifications will be needed to adapt the compiler : our microprocessor doesn't have multiplication or division instructions, and uses 16 bits integer instead of 32. Nevertheless, those changes will be minor because even if our microprocessor has fewer register and fewer instructions, the compiler will not use all MIPS registers and instructions, and most of the instructions we lack can be simulated by several of our instructions.

6.1 Schema



6.2 Registers

There are 16 registers of 16 bits, labeled by integers : W_0 to W_{15}. You may use them the way you want, but we advise you to use the following conventions, inspired from MIPS conventions :

	MIPS equivalent	Usage
W_0	$V0	expression evaluation and results of a function
W_1	$V1	expression evaluation and results of a function
W_2	$A0	argument 1
W_3	$A1	argument 2
W_4	$A2	argument 3
W_5	$A3	argument 4
W_6	$S0	saved temporary (preserved across call)
W_7	$S1	saved temporary (preserved across call)
W_8	$S2	saved temporary (preserved across call)
W_9	$T0	temporary (not preserved across call)
W_{10}	$T1	temporary (not preserved across call)
W_{11}	$T2	temporary (not preserved across call)
W_{12}	$T3	temporary (not preserved across call)
W_{13}	$SP	stack pointer
W_{14}	$FP	frame pointer
W_{15}	$AT $RA	temporary register used by pseudo-instruction (see 7.2) return address

Thanks to `jmp*` instruction with two arguments (see instructions set, section 7), you can save return address in any register (instead of only W_{15}) when you jump to another point of the assembly program. That way, you can create functions calls. The mechanism is nearly the same as in MIPS architecture (see [1]) with `jla` MIPS instruction.
Our microprocessor also uses two additional registers in conditional jumping, registers *carry* and *zero*, described below.

6.3 *Carry* and *zero* registers

One of the most important difference with MIPS architecture is the lack of conditional instructions like MIPS `beq` (see [1]). This lack is compensated by two 1 bit pseudo-registers *carry* and *zero*. They are unaccessible to the user. They are automatically updated by most instructions. We have some instructions providing jump conditioned on the value of those two registers.

6.3.1 *Carry* register

Carry register is only updated by arithmetic operations (`addu`, `subu`, `sub`, `sra`, `srl` instructions, and pseudo-instructions which use these real instructions - see instructions set, section 7).

- `addu` W_{dst} W_{src1} W_{src2} : *carry* will be set if $W_{src1} + W_{src2} \geq 2^{16}$ (values considered as unsigned)
- `subu` W_{dst} W_{src1} W_{src2} : *carry* will be set if $W_{src1} - W_{src2} \geq 0$ (values considered as unsigned)
- `sub` W_{dst} W_{src1} W_{src2} : *carry* will be set if $W_{src1} - W_{src2} \geq 0$ (values considered as signed)
- `sra` W_{dst} W_{src} : *carry* is the less significant bit of W_{src}
- `sla` W_{dst} W_{src} : *carry* is the less significant bit of W_{src}

For example, suppose that $W_0 = 10$ and $W_1 = -4 = 2^{16} - 4$:
- `addu` W_2 W_0 W_1 will set carry
- `subu` W_2 W_0 W_1 will clear carry because $10 < 2^{16} - 4$
- `sub` W_2 W_0 W_1 will set carry because $10 \geq -4$

6.3.2 *Zero* register

Zero is updated after each instruction using a destination register (W_{dst} in 7.instruction set). If the new value of W_{dst} is 0, *zero* will be set, otherwise it will be cleared.
That way, you can easily jump if two registers are equal by jumping if the result of a `xor` on those two register is 0.

6.4 External interfaces

Programs can communicate with other module of the CPU through `input` and `output` instructions. Here are the current mapping of address :

	Input	**Output**
0-7	real 16 bits input port 0-7	real 16 bits output port 0-7
8	16 bits timer *period*	
9	16 bits timer	

Default values are 0.

6.5 Timer and sleep mode

Our microprocessor has a 16 bits timer, that is to say a counter modulo *period* incremented at each clock cycle. A 0 *period* corresponds to a counter modulo 2^{16} (default). You can set its period and its value with `input` and `output` instructions (see above).

Warning : If you change the period to a value less than or equal to current timer value, the counter will not be reset. Hence it is recommended to manually reset counter (by setting it to 0) after changing its period.

You can put microprocessor into *sleep* mode with `sleep` instruction. Nothing is done until the timer reaches its period. This feature is important for the digital watch, because we need to increment seconds with the good period.

7 Instructions set

You will find below the set of instructions used by our microprocessor. Assembly programs used with our translator can use real microprocessor instructions or pseudo-instructions that will be converted into several real instructions by the translator.

We will use the following notations :

W_x	register W_x	
K (n)	constant of n bits	4, 8 or 16 bits
lo(*)	low bits (half) of *	4 or 8 bits
hi(*)	high bits (half) of *	4 or 8 bits
addr(label)	address of the label	16 bits

7.1 Real instructions

Here are real instructions, the four last columns displaying the instruction code saved in a ROM program.

ALU instructions								
					4×4 bits			
addu	W_{dst}	W_{src1}	W_{src2}	$W_{dst} \leftarrow W_{src1} + W_{src2}$	0	dst	src1	src2
adduc	W_{dst}	W_{src1}	W_{src2}	$W_{dst} \leftarrow W_{src1} + W_{src2} + carry$	1	dst	src1	src2
subu[1]	W_{dst}	W_{src1}	W_{src2}	$W_{dst} \leftarrow W_{src1} - W_{src2}$	2	dst	src1	src2
sub[1]	W_{dst}	W_{src1}	W_{src2}	$W_{dst} \leftarrow W_{src1} - W_{src2}$	3	dst	src1	src2
xor	W_{dst}	W_{src1}	W_{src2}	$W_{dst} \leftarrow W_{src1}$ xor W_{src2}	4	dst	src1	src2
or	W_{dst}	W_{src1}	W_{src2}	$W_{dst} \leftarrow W_{src1}$ or W_{src2}	5	dst	src1	src2
and	W_{dst}	W_{src1}	W_{src2}	$W_{dst} \leftarrow W_{src1}$ and W_{src2}	6	dst	src1	src2
move	W_{dst}	W_{src}		$W_{dst} \leftarrow W_{src}$	8	dst	src	0
not	W_{dst}	W_{src}		$W_{dst} \leftarrow$ not W_{src}	8	dst	src	1
sra	W_{dst}	W_{src}		$W_{dst} \leftarrow$ shift[2] right arithmetic[3] of W_{src}	8	dst	src	2
srl	W_{dst}	W_{src}		$W_{dst} \leftarrow$ shift[2] right logical of W_{src}	8	dst	src	3

Jump instructions								
jmp	W_{src}			Jump to W_{src}	9	0	src	0
jmpz	W_{src}			Jump to W_{src} if *zero*	9	0	src	2
jmpc	W_{src}			Jump to W_{src} if *carry*	9	0	src	4
jmpnz	W_{src}			Jump to W_{src} if non *zero*	9	0	src	3
jmpnc	W_{src}			Jump to W_{src} if non *carry*	9	0	src	5
jmp	W_{src}	W_{ret}		Jump to W_{src} save next address to W_{ret}	9	ret	src	8
jmpz	W_{src}	W_{ret}		Jump to W_{src} if *zero* save next address to W_{ret}	9	ret	src	10
jmpc	W_{src}	W_{ret}		Jump to W_{src} if *carry* save next address to W_{ret}	9	ret	src	12
jmpnz	W_{src}	W_{ret}		Jump to W_{src} if non *zero* save next address to W_{ret}	9	ret	src	11
jmpnc	W_{src}	W_{ret}		Jump to W_{src} if non *carry* save next address to W_{ret}	9	ret	src	13

Load/Store instructions								
lhi[4]	W_{dst}	K (8)		$hi(W_{dst}) \leftarrow$ K	10	dst	hi(K)	lo(K)
lli[3]	W_{dst}	K (8)		$lo(W_{dst}) \leftarrow$ K	11	dst	hi(K)	lo(K)
lh	W_{dst}	W_{src}		$W_{dst} \leftarrow$ RAM(W_{src})	8	dst	src	4
sh	W_{src1}	W_{src2}		RAM(W_{src2}) $\leftarrow W_{src1}$	12	0	src1	src2

Input/Output - other instructions								
output	K (4)	W_{src}		output(K) $\leftarrow W_{src}$	13	0	K	src
input	W_{dst}	K (4)		$W_{dst} \leftarrow$ input(K)	8	dst	K	5
sleep				Sleep (see 6.5)	8	0	0	8

7.2 Pseudo-instructions

Since real instructions lack some important practical functions, we have created pseudo-instructions. Every time the assembly translator sees such an instruction, this is immediately translated into real instructions.

[1] The only difference between `sub` and `subu` is the carry modification (see 6.3.1).
[2] One bit shift. Notice that left shift can be done with `addu` instruction.
[3] Arithmetic means that sign bit is copied.
[4] Instructions `lli` and `lhi` do not modify or erase the unused part of the register, respectively $hi(W_{src})$ and $lo(W_{src})$.

Warning : Some pseudo-instructions use W_{15} registers without saving it.

Load/Store pseudo-instructions				
`li`	W_{dst}	0		`xor` W_{dst} W_{src} W_{src}
`li`	W_{dst}	K (16)		`lhi` W_{dst} hi(K) `lli` W_{dst} lo(K)
`la`	W_{dst}	label		`li` W_{dst} addr(label)

Arithmetic pseudo-instructions				
`addui`	W_{dst}	K (16)	W_{src}	`li` W_{15} K `addu` W_{dst} W_{15} W_{src}
`adduci`	W_{dst}	K (16)	W_{src}	`li` W_{15} K `adduc` W_{dst} W_{15} W_{src}
`subui`	W_{dst}	K (16)	W_{src}	`li` W_{15} K `subu` W_{dst} W_{15} W_{src}
`subi`	W_{dst}	K (16)	W_{src}	`li` W_{15} K `sub` W_{dst} W_{15} W_{src}

8 Assembly translation

The second step of our project was to create a program able to translate assembly language into microprocessor instructions (16 bits integers). The program has also a simulation mode, which enable the user to overview the content of the register through the time, so that they can fix their programs.

8.1 Getting started

Like for the simulator, you may compile the assembly translator by Code Blocks or our own makefile (just type `make` in "asm" folder). You will then be able to use the program "asm" with the following syntax :

```
asm [options] sources
asm -h for help
```

Where option are :
`-v` for verbose mode
`--sim` to launch the simulation after asembly translation
`--rc` to display the content of the registers during simulation
`-p` 0 period of one cycle
`-n` 0 maximum number of cycles
`--watch` display a watch instead of just printing the contents of outputs (see bellow)

In the watch mode, output ports 0 to 5 corresponds to a 7-segment digits (each bit is a segment - the order is the traditional order : less significant bit is the upper segment). The simulator display the watch each time you wrote something in output port 0. That's why you have to update this port at the end, when you want to update the watch. Please see example "asm/tests/watch.asm" for more informations.

Notice that targets Makefile are nearly the same as simulator ones. You can so automatically run tests by typing : `make demo TEST=yes`. This will run simulator one each **.asm* file of folder "asm/tests" for 1000000 cycles and compare the results with **.res* file.

8.2 Quick tour of the program code

Here is a brief description of the way the assembly translator works. We hope it will be helpful if you ever have to look up the source code.

8.2.1 Parsing

The file is first parsed in the parser module. A simple handmade loop gets the arguments and the name of the instructions. A counter stores the number of instructions already studied so that the parser stores with every label name the number of the instruction it corresponds to. Pseudo-instructions may increment the counter more than once because they may introduce a lot of new real instructions. All those instructions are stored in a vector.

8.2.2 Handling of instructions

A map module stores the main informations about instructions and pseudo-instruction by enabling the creation of a table associating to every string name a special structure.

typedef struct instr_info

instr_type type; *Instructions are sorted by type to enable global verifications*
int code; *The code is the first of the 4 bits of the instruction code in the ROM*
int exn; *See below*
int nb_int_arg; *Number of expected registers or integer constants in the arguments.*

The purpose of *exn* is double : A positive exn is the complementary identification of the instruction, that is to say the fourth of 4 bits of the instruction code in the ROM if this bit is used to describe the instruction.
A negative *exn* (case of pseudo-instructions) contains the number of real instructions this instruction stands for.

The map module provides the general structure of the instruction, but the parser builds a specific object for every different instruction :

class instruction

friend class src_file;

private :
instr_type type;
int code; *Instruction code (4 bit integer)*
int instr_code; *ROM code (16 bit integer)*
int exn;
string name;
int arg[3]; *The initial int arguments will be completed later to form the 3 last 4 bits part of ROM instruction code*
string s_arg; *String argument for instruction dealing with labels*
int line; *Line in the assembly program to display errors*
int nb_int_arg;

public :
instruction(string name_, int cline, table_instr *table); *Constructor from the generic informations*
void print(); *Print name and arguments.*
void printc() { cout << instr_code << endl;};
void error(string msg);
void verif(table_label *labels, vector<instruction> *vect, vector<instruction>::iterator *it, bool verbose); *See below*

bool execute(int registers[18], vector<instruction>::iterator *it, vector<instruction>::iterator begin, int ram[SIZE_RAM], unsigned short inputs[NB_IO], unsigned short output[NB_IO], bool watch); *See simulation*

A simple loop on all the instructions stored in a vector enables to replace labels by the address of the instruction they correspond to, and replace pseudo-instructions by the real instructions they stand for. The ROM code of the instruction is also computed. All of this corresponds to the function *verif*.

8.2.3 Simulation

The simulation of assembler in C++ has to simulate several aspects of the real microprocessor. RAM, input and outputs are represented by tables (integer or short). So are the registers. Note that carry and zero are the 16^{th} and 17^{th} elements being used to simulate carry and zero register.
The actual simulation is executed by calling the **execute(...)** function of every instruction in the vector. It reacts differently based on the code and arguments of the function, simulating all the operations required.

8.3 Syntax

The assembly language recognized is very close to the syntax of microprocessor instructions. Below stands an example of the syntax used.

```
label1 :
label2 :    instr arg1 arg2 arg3
            instr arg1 arg2 arg3 ; comments
```

Instructions begin after a tabulation, and are separated of labels by colons. You can add comments after a semicolon. *instr* is the string corresponding to the name of the real or pseudo-instructions in section 7 page 19. arg_i are most of the time registers names, but they can also be constant integers, or label names.
You can find detailed examples in the *.asm* files of our project ("asm/tests" folder). Here are a few examples of complete instructions.

```
            addui W5 W5 1
            sub W15 W5 W6
            la W15 watch
```

8.4 Watch example

Thanks to this translator, we were able to program a digital watch in assembly language, in order to run it on our microprocessor. You may find the code of the program in the "asm/tests/watch.asm" file.
You can directly simulate the watch with the command :

```
make watch
```

The result will be printed with the 7-segment interface we introduced before.

There is also a watch which count the time from 23:59:59 to 00:00:00. Just type :

```
make watch2
```

Part III
Microprocessor realisation

This part will deal with the use of the aforementioned tools in order to create a full working microprocessor. That is to say we must now define electronic circuits that we will simulate thanks to the netlist simulator in order to run a digital watch assembly program (translated to machine language in the previous part).

9 Realisation

9.1 PHP Netlist generation

The first problem we had to deal with was that a microprocessor required complex circuits that could be huge. Writing them by hand would be an overwhelming task resulting undoubtly in many errors. We needed an automatic generation of such complex netlists. Loops and recursions were required.
On the other hand, we didn't want to add such functions to our netlist language. We wanted our netlists to represent exactly the real circuit, and not be a higher level description of the circuit. Therefore, we chose to add a special step before netlist simulation, that is to say the generation of complex netlists.
To that aim, we chose to use PHP, because it was initially designed as a preprocessor for webpages and therefore enables a very easy manipulation of text (and an easy way to output text). In a few files designed to be used as "libraries", we programmed the most important functions (lists, mux operator...) which would later be used by all our PHP netlists. We also designed a counter library, enabling the creation of any counter modulo v (used in particular for the *sleep* instruction).
PHP files are automatically compiled toward .net files. That way, the initial PHP script can benefit from loops and recursion of PHP language, but stay close to the real netlist language (thanks to the possibility to add PHP code between tags : $<?\ldots?>$ anywhere). Moreover, the final .net netlist contains an exact description of the final circuit.

9.2 Presentation of involved circuits

We designed those circuit trying to be as general as possible. Therefore, even if the circuits operate on 16 bits integers, this can easily be changed in most cases thanks to a PHP parameter.

- AddSub :
  This circuit is the core of the aritmetic operation. It gets as inputs two 16 bit integers, plus 3 extra bits : the carry, a bit containing the information of whether or not the previous numbers are unsigned, and a bit containing the information about the operation to do (addition or substraction). The output is a 16 bit integer, and a new carry.

- Registers :
  Registers are represented by a small circuit dealing with a 16 bit integer : it is a loop (the output stays the same) unless the parameter w (for write) is true, in that case the output will be changed to a 16 bit integer given in input.

- Logical operations :
  Logical operation are implemented thanks to the logical gates which apply the operation on every bit.

- Main :
  The circuit we call main has the same inputs and outputs than the microprocessor. The interpretation of the instructions is eased by an intelligent design of their syntax : the second argument is always the destination, the third one is always the source.

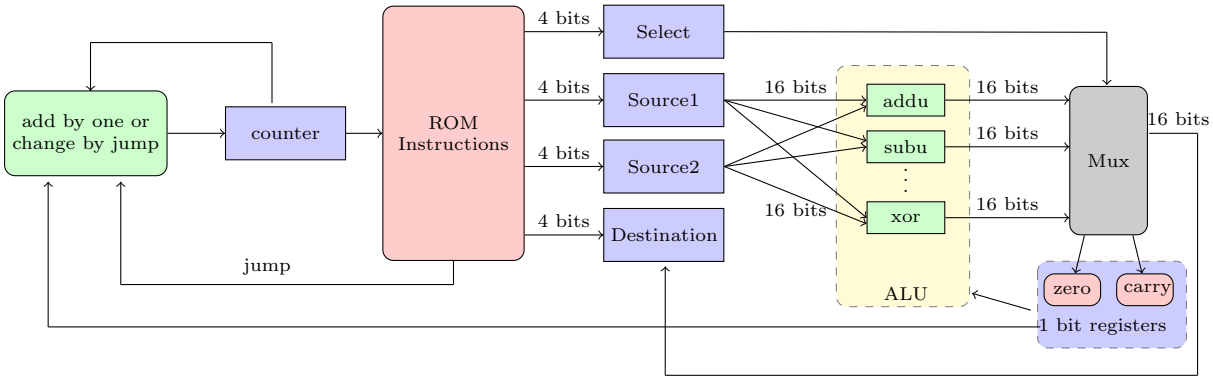


Figure 1: Netlist organization

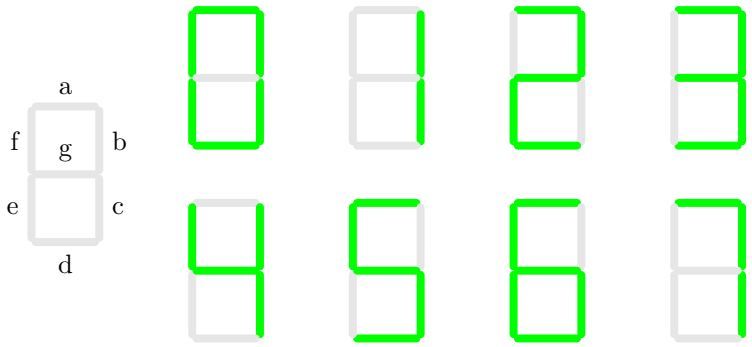
9.3 A new simulator flag

Outputs (and inputs) of microprocessor netlist are quite different from netlist watch one (see 4.5) and of any other circuit : there are 8 ports of 16 bits and two other bit : one toggles each time microprocessor has performed an output and the other each time microprocessor has performed an input.
In normal mode, simulator should print all outputs each time microprocessor has performed an output. In watch mode (graphical watch or old watch mode), simulator should update watch only when needed (and should use different outputs as netlist watch mode). That's why we have added a new flag to simulator : `--micro`.

10 Improvements

10.1 Seven Segments and Graphical Output

We have added a new mode to your simulator : graphical watch (flag `--watch2`. It's the same mode as watch mode (see 4.5) except that output is graphical (there is a screenshot page 26). You will need *WxWidgets* to compile the simulator. If you don't have this library, you can disable this new mode by adding `USE_WX=no` in `make` command line.



We use seven bits to represent segments a to g. For convenience, instead of providing each segment with two images for the opposite states, we just regard all vertical segments to be the identical, and the same for horizontal segments. In this way, we only need four patterns to represent any digit. Notice that the `colon` used to separate hour and minute (as well as minute and second) need another image, hence five different images suffice to simulate the watch.

10.2 Ocaml Compilation

Our compiler *mimo* is the work of the compilation course, while the rule is a bit different from that of MIPS in these 3 espects:

1. Instead of using 32 registers of 32 bits, here 16 registers of 16 bits suffice.
2. The `multiplication, division, modulo` instructions are replaced by software function.
3. The `conditional jump` instruction in MIPS is converting by `xor` or `sub` followed by `jmpz`, `jmpnz`, `jmpc` and `jmpnc` (the definition is in 2.1 ALU instructions) because our jump operations are relating to the registers *zero* and *carry*.
4. The `system call` instruction in MIPS is substituted by `input`, `output` and `sleep` instructions and by some software functions (for allocation data on heap for example). There are 8 gates for input and output respectively, so in this way, we can output the 7 segments information of a digit simultaneously. Notice that the `print_int` instruction in MIPS corresponds to `output in port 0`, and `print_newline` instruction corresponds to `output 1 in port 1`.

11 How to use our Simulator

The Makefile in "simulateur" folder enables you to automatically compile, assemble and simulate with microprocessor netlist any assembly or Ocaml file. Just type `make name.asm.run` or `make file.ml.run`.

Here are examples :

- `make ../asm/tests/watch.asm.run SIM_FLAGS="-n 0`[1] `--watch2"` : run the assembly watch at the maximum speed (seconds printed are not real seconds) and use graphical watch mode.
- `make ../asm/tests/watch.asm.run SIM_FLAGS="-n 0 --watch -p 0.005"` : run the assembly watch at the correct period (0.05 seconds - correspond to period loaded in period timer of microprocessor at the beginning of the assembly code) and use old watch mode[2].
- `make ../mimo/testMicro/watch.ml.run SIM_FLAGS="-n 0 --watch2 -p 0.0001 -c"` : run the Ocaml watch in compilation mode at the correct period (0.0001 seconds - correspond to period loaded in period timer of microprocessor at the beginning of the assembly code).

If you prefer, you can also use "start.py" script which enables you running any of these examples easily (type "3c" and enter to run last item for example).

12 Results

12.1 Digital watch

After we programmed in Ocaml, compiled by *mimo*, and simulated using the previous procedure, finally we get a real watch which looks like the following:

[1] `-n 0` enables you to run simulator infinitely - otherwise, it stops it after 16 cycles.

[2] In this old mode, watch is printed each time any digit is modified. Hence you can see 00:00:09, 00:00:19 and then 00:00:10. (00:00:19 appears very little time). This is the default behaviour in a real circuit because our microprocessor can't change all its outputs at the same time.

12.2 Result Comparison

We compare the execution time for *Asm*, *Netlist* and *Ocaml* mode, both with and without the *compile* option (-c) for each of them. The testing time should not be too short, otherwise the time for pretreatment of the *Compilation* mode would influence our comparison. Here we take 1 min for example: in every mode, after the watch has run for 1 min, we read the time showed at that moment to see how many seconds already passed; this number multiplied by the number of cycles per second (printed - not real second) of this mode yields the total number of instructions executed, as following :

	Compilation Mode	Interpreting Mode
Netlist (4.5)	6.0×10^6	1.5×10^6
Asm (8.4)	0.96×10^6	0.09×10^6
Ocaml (12.1)	0.96×10^6	0.10×10^6

Speed Comparison between Different Modes and Different Watches

You can see that speed of ASM watch is nearly the same as speed of Ocaml watch : it is normal since netlist is the same : this is microprocessor netlist.

Result Analysis:
Our compiler is not much optimized, as seen in the result. But the important aspect is that it works correctly in simulating our watch, and this is the first step toward an excellent simulator. The defect makes us realize the gap between our simulator (which is the work in one semester) and a top level simulator (which is improved for several decades), and arouses our interest in digital system to a great extent. In our further studies, we will try to find the answer to the improvements step by step so that one day we could catch up or even exceed the existing technology of simulators.

References

[1] Assemblers, Linkers, and the SPIM Simulator, *James R. Larus*, http://pages.cs.wisc.edu/~larus/HP_AppA.pdf, novembre 2009