Educational Simulation of the RiSC Processor

Marc Jaumain
BEAMS department, Bio Electro and Mechanical Systems,
Université Libre de Bruxelles, Belgium
mjaumain@ulb.ac.be

Michel Osée[1], Aliénor Richard[2], Alexis Vander Biest[3], Pierre Mathys[4]

Abstract **- In the context of courses related to "Architecture of microprocessors", our educational objective is to make students understand the internal dynamic mechanism of processors. Since internal measurements are not possible on such devices, simulation is the only way. Hence, we have developed our own innovating simulator with a specific focus on student interactivity. We have chosen the RiSC16 processor because it is simple but complete and has been designed for educational purposes. The simulator we propose offers different opportunities. It allows the user to define its own programs in assembly language and to see graphically the corresponding internal dynamic behaviour of the processor (interactivity). Secondly, the visualization of the architecture of the RiSC16 is enhanced by the use of colours which change depending on the activity of the different blocks. Thirdly, stepping instruction by instruction allows the user to visualize the evolution of the content of memories and registers. Furthermore, the Java language has been chosen to implement our simulator. The modularity of this language makes it easy to adapt to other processors and let several perspectives open. The simulator has been tested in real laboratory conditions and showed to be quite helpful for the students.**

Index Terms – simulator, micro architecture, processor, RiSC-16.

INTRODUCTION

For the majority of the students, the understanding of the internal working of a microprocessor is difficult to acquire in a textbook because of the static nature of the paper support. In addition, it is not possible to illustrate these concepts by practical lab work, since one cannot reach the internal signals of the microprocessors. On the contrary, a computer simulation allows the user to visualize the working procedure of a microprocessor and can thus improve the comprehension level of the students. It is now possible to visualize the execution of an instruction in details. The aim of this text is to present our simulator and to explain why and how it has been designed. Firstly, the paper deals with the context in which the simulator is used, and the options for the selection of the processor. After a brief description of the processor, we explain how the simulator works and some alternative designs of the simulator are proposed.

CONTEXT

Our simulator is part of the labs on microprocessor architecture for first year master students in electrical engineering (options electronics, telecommunications and computer science).

The prerequisites for this course are two previous courses of electronics and a third one about logical circuits. The first course is an introduction to electronics in which the students have their first contact with a microcontroller and assembly language code. The second one focuses on digital electronics: during the labs, the students program a microcontroller in C language. In the course about logical circuits, they learn how to analyse and synthesize combinatorial and sequential circuits.

The aim of the course on microprocessors architecture is to give a more advanced knowledge of the main concepts in the discipline (instruction sets architecture, pipeline, memory architecture, memory management, buses …)

The main difficulty in the study of microprocessor resides in the integration itself which hides a lot of interesting internal events. In most of the integrated development environments coupled with debuggers (or simulators), it is possible to step at instruction level and see the content of the registers but there is no way to see the progress of the instructions within the instruction cycle execution. In the textbook, the successive events within the cycle are presented like a slide show based upon the internal architecture. Since some steps of the execution of an instruction are asynchronous and other ones are synchronous, it is difficult to represent it realistically on paper. Therefore, a simulator seems to be the better tool to represent this in an interactive and dynamic way.

WHAT KIND OF PROCESSOR?

The majority of existing simulators are based on real microprocessors with complex instruction sets. This type of simulators is more aimed at people who have a good experience in the subject and want to test and debug some programs without having to download them in the target hardware. Some educational simulators are available with a

[1] Michel Osée, BEAMS department, ULB, mosee@ulb.ac.be
[2] Aliénor Richard, BEAMS department, ULB, arichard@ulb.ac.be
[3] Alexis Vender Biest, BEAMS department, ULB, avdbiest@ulb.ac.be
[4] Prof Pierre Mathys, BEAMS department, ULB, pmathys@ulb.ac.be

basic instruction set. Both types of simulator only show the evolution of the registers and the memory variables to help debugging the code. Such debuggers have already been used by students in others laboratories but in our simulator, we would like to show the internal mechanism of the microprocessor to students who are not familiar with the subject.

Two selection criteria have been considered: the architecture and the instruction set.

Concerning the architecture, the main options are Harvard type and Von Neumann type. The Von Neumann architecture has got a single data path to transfer data and code. This leads to minimal surface of silicon and reduces the complexity of design. The Harvard architecture has separate data paths for data and for instructions. This accelerates the execution because the next instruction can be fetched during the execution of the current instruction. The Von Neumann architecture has been used in general purpose processor for personal computers, while the Harvard type is more used in workstations and for real-time microprocessor like DSP. We have chosen to illustrate the Harvard architecture because we believe the separation between data and code easier to understand for the students. Besides, the overlapping of execution and opcode fetch is at the base of the pipeline concept, which is taught in the course and is illustrated in a complementary simulator.

The second criterion is the complexity of the instruction set: Reduced Instruction Set Computer (RISC) or a Complex Instruction Set Computer (CISC). The latter is characterized by a large number of instructions, in order to simplify the work of compilers and reduce the size of the code. The drawback is the size of the instruction decoder and the variable length and execution time of the instructions, which is not recommended to implement an efficient pipeline. The RISC processors have got a reduced number of instructions, all with the same length and same execution time. A disadvantage of this approach is that a complex operation is compiled in a large number of simple instructions. However the performances are excellent because the simple instructions are optimized, execute quicker than their CISC counterpart and because pipelining works properly. The RISC architecture is the best candidate from an educational point of view for the following reasons:

- the RISC architecture is easier to understand than the CISC one because the execution is more systematic.
- the display of different part of processor is easier and clearer.
- the number of instructions being less than a few tens, the students can juggle easier with the instruction set in a minimum time
- the RISC instruction set is more fitted to pipeline concept because all instructions have the same execution time

A last point considered for the choice of a microprocessor is that we wanted the microprocessor to be physically implementable in hardware. This shows that it is not just a concept, in simulation, but that it corresponds to the real working of some processor. This brings credibility to the simulator.

THE RISC-16 PROCESSOR

The processor selected is the RiSC-16, which immediately seemed to be an excellent candidate for this job. The RiSC-16, for “Ridiculously Simple Computer”, has been developed by Prof. Bruce Jacob at the University of Maryland with an educational aim. There are two implementations of this architecture, a sequential one and a pipelined one. In this paper, we just give a small description of the sequential implementation. For more information about RiSC-16, the reader is invited to refer the three documents: [1] for the instruction set, [2] for the sequential implementation and [3] for pipeline implementation.

The RiSC-16 is a RISC processor based upon Harvard architecture. As its name indicates, it is a 16 bits processor. All data and instructions are in two bytes, and so, all registers and the two memories are in short-word format. It is made up of:

- one bank of eight registers, addressable in three bits. The register 0 is read-only and contains the null value, which is quite common among RISC processors
- separated instruction and data memories. Both are addressable in sixteen bits, and hence have a capacity of 64Kwords.
- one Arithmetical-Logical Unit (ALU) that can execute three operations: addition, bitwise nand and test of equality.
- multiplexers to choose between buses.

TABLE I
INSTRUCTIONS [1]

Mnemonic	Assembly Format	Action
add	add regA, regB, regC	Add contents of regB with regC, store result in regA.
addi	addi regA, regB, Imm	Add contents of regB with Imm, store result in regA.
nand	nand regA, regB, regC	Nand contents of regB with regC, store results in regA.
lui	lui regA, Imm	Place the 10 ten bits of the 16-bit Imm into the 10 ten bits of regA, setting the bottom 6 bits of regA to zero.
sw	sw regA, regB, Imm	Store value from regA into memory. Memory address is formed by adding Imm with contents of regB.
lw	lw regA, regB, Imm	Load value from memory into regA. Memory address is formed by adding Imm with contents of regB.
beq	beq regA, regB, Imm	If the contents of regA and regB are the same, branch to the address PC+1+Imm, where PC is the address of the beq instruction.
jalr	jalr regA, regB	Branch to the address in regB. Store PC+1 into regA, where PC is the address of the jalr instruction.

- one control unit. Its functions are to decode the opcodes and to control the ALU, the multiplexers and the write function into the register bank and into data memory.
- a program counter (PC) and its incrementer.
- an instruction register containing the instruction that is being executed.
- an adder to compute jump addresses.
- two sign-extended logic blocs to convert the 7 bits immediate values into the 16 bit format.
- one left shift logic to convert the 10 bits immediate values into the 16 bit format.
- several buses to convey data between elements.
- control signals routed to the different blocs (for example, to choose the input bus of a multiplexer).

Refer to Figure 1 to see how these are connected.

The instruction set consists of 8 instructions. Table I shows their assembler format and describes their operation. This processor illustrates the RISC philosophy pushed to its maximum of simplicity. In fact, the instructions are elementary, but they are powerful enough to solve complex problems, and none instruction can be replaced by a combination of the other ones.

The students are rapidly able to master this reduced set of 8 instructions and to write small programs. A second strong point of the RiSC-16 is the small number of internal elements. This permits displaying clearly all blocks on the screen. Furthermore, both the sequential and the pipeline version were implemented on a FPGA during a master thesis. This proves the validity of the concept to the students.

SIMULATOR

As shown in Figure 1, the simulator displays three windows. The main window is on the left and is devoted to the elements of the processor. At the right top, we find the program memory window containing the code written in assembly format and assembled in binary machine code. The last one, at the right bottom, is the data memory window. This window shows the data that are produced by the program, but also allows at the user to write his own data anywhere in memory.

All instructions are executed in one machine cycle. This cycle is divided in four steps. These are successively:

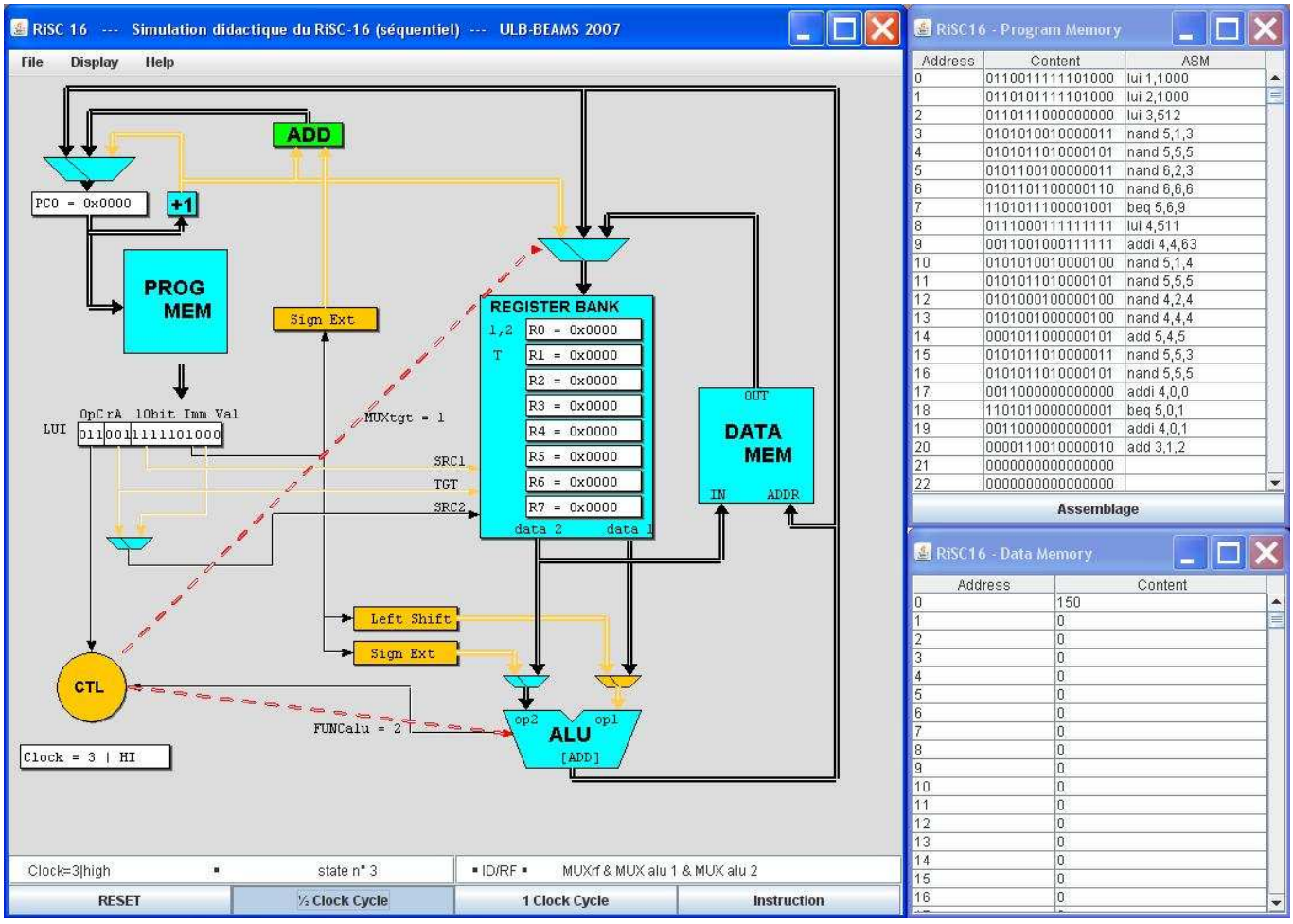

FIGURE 1
PRINT SCREEN OF THE SIMULATOR

- Instruction Fetch (IF): During this step, the program memory receives the address of the instruction from the program counter, and sends the instruction to the instruction register.
- Instruction Decode and Register Fetch (ID/RF): The control unit decodes the instruction, sends the operation code to the arithmetical-logical units and the address bits required to select source and target registers.
- Execution (EX): The arithmetical-logical unit receives the operands and executes the operation.
- Write Back (WB): The result is stored in its target and the program counter receives the new address from its input multiplexer.

A relevant functionality of the simulator is the fact that the processor behaviour can be simulated with three time scales:

- by half clock cycle, which is the smaller interval. This simulation mode enables the description of the internal processor mechanism in its lower details.
- by clock cycle. This mode enables seeing the state of processor at the end of each clock (=micro) cycle. The simulation is less detailed, but faster.
- by instruction. This corresponds to traditional software simulators. It shows the state of the different registers and the data memory at the end of each instruction. It is useful for the study of the assembly language.

The majority of elements run in an asynchronous manner. It means that between elements, there is no intermediate register controlled by a clock or a control signal. On the other hand, the control unit is synchronized with the clock, and, the program counter, the register bank and the data memory are synchronized with control signals for the write-back operations. Globally, there is synchronization at the instruction level because of the Write control signal of the program counter.

In order to highlight the elements that are working, the simulator uses a colour code. For all elements (except buses), three colours are coding the different states:

- *Green* means that the block is busy: data or address (depending on the block) is stable at the inputs and the block is processing it. Hence this state is transitional. A good example is the access time of the Program Memory. Such state does not exist for buses.
- *Orange*: this state always follows the previous one. The block has finished its processing and data is available at the outputs. This state can be described as stable.
- *Default* colour: this colour is used for two states, "default" state and "has been used" state. "Default" state means that the element did not yet receive relevant data for the current instruction. "Has been used" state means that the data produced by this block has been taken into account by the next block and will play no more role for the current instruction.

Figure 2 illustrates the different states and the corresponding colour of an element. The continuous lines represent the bus at the output of the considered element.

Buses are somewhat different because they just convey the data and do not process it. They can be coloured in orange and black. Black is the default colour. Orange is used for buses that convey information until the next element begins processing the data. Control signals are just displayed when they are active, in red dotted lines. A last colour is used for highlighting a register (of bank) the value of which has been changed.

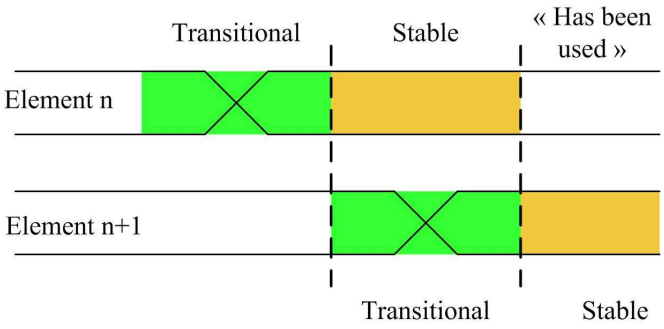

FIGURE 2
THE DIFFERENT STATES AND ITS COLOUR OF AN ELEMENT

In half clock cycle scale, the successive asynchronous events within this interval are represented like a video in which the elements change their colour according to their state. The user can choose between the three play speeds, the one that is more adapted to his level.

The user has the possibility to change the format of value being displayed in the program counter and register bank. Options are decimal, signed decimal, hexadecimal and binary format. Another function of the simulator is the possibility to save and to import the contents of data and program memory. The contents are saved in text files that can be easily modified.

The Java language has been chosen to implement the simulator. The choice of an object oriented language was justified by its modularity that enables adapting the simulator for other processors or a pipeline version. Each element of the processor was implemented in a class. There is only one class called “SequentialDesign” for this simulator, that contains an object of each element, and that dispatches to it the clock signal at each click on one of buttons. This means that to add a new element (like a shifter, for instance), we have just to implement it and connect it to the buses. As this simulator was in major part implemented by students (during a project in Master 1), Java has been preferred because it is known by the majority of students of different options. Furthermore, it can use a large library of graphical objects.

This first version of the simulator has been used in our lab since the last academic year. Two four hours’ labs have been dedicated to the simulator. For the first lab, the students begin by observing the detailed mechanism of all instructions. After, they have to program an addition of unsigned numbers with carry detection. Writing this small program, they can realize that the more we reduce the instruction set, the more we have to combine them to produce higher level instructions and they are confronted to assembly programming. An example of this code is displayed in the program memory window of Figure 1. In the second lab, guided by some programming exercises, they

have to propose instruction set and hardware improvements to accelerate the processor (e.g. logical and arithmetical shift). They have also to think about the way to transform this version of RISC16 into a synchronous processor, and what are the consequences in terms of silicon surface and clocking of the processor.

PROSPECT

A second version of the sequential simulator is in preparation to make it more interactive and to improve the visualization of the elements that are actually working within the instruction. The first stage is to allow the user to choose if he wants to see the time interval during which asynchronous events occur like a video or like a slide-show with intermediate clicks. This involves adding a temporal axis to situate the time within the clock cycle. Figure 3 gives an example of such axis.

FIGURE 3
EXAMPLE OF TEMPORAL AXIS

Since some elements of the processor are asynchronous, the elements that have got stable data at their input go into the busy state and are displayed in green, even if their output data are not used for the instruction. To resolve this problem, two new states and colours should be introduced, one for the busy state without useful data and one for the stable state without useful data.

Other versions of this processor are also planned to allow the students to compare different architectures of microprocessors.

- an advanced version with more registers in the bank, an enrichment of the instruction set, more logical function for ALU...
- a synchronous version.

A last considered version is the pipeline implementation. The students will thus have the occasion:

- to understand the pipeline principle that is not obvious,
- to compare it with the sequential implementation,
- to observe different pipeline hazards and how to solve them.

CONCLUSIONS

After one academic year of use, the simulator has been proven to be very helpful for students to learn MPU internal mechanism. Compared to an ex-cathedra course and textbooks, they appreciate the ability to see events at their own rate and make their own experiments. The actual impact on student marks should still be demonstrated by a broad statistical analyse. The feedback from the students has also underlined some improvement to bring, and has raised new questions leading to alternatives versions of the processor. These variants will be easily implemented thanks to the modularity of the program.

ACKNOWLEDGMENT

We would like to thank to Q. Monneaux and D. Cross for their contribution to the project.

REFERENCES

[1] Jacob, B., "The RiSC-16 Instruction-Set Architecture", *ENEE 446: Digital Computer Design*, Fall 2000. Available : http://www.ece.umd.edu/~blj/RiSC/RiSC-isa.pdf

[2] Jacob, B., "RiSC-16 Sequential Implementation", *ENEE 446: Digital Computer Design*, Fall 2000. Available : http://www.ece.umd.edu/~blj/RiSC/RiSC-seq.pdf

[3] Jacob, B., "The Pipelined RiSC-16", *ENEE 446: Digital Computer Design*, Fall 2000. Available : http://www.ece.umd.edu/~blj/RiSC/RiSC-pipe.pdf