RISC16 SIMULATORS - HELP

SOMMAIRE

GENERAL INFORMATION

INTRODUCTION ON THE SIMULATOR

Three simulators have been developed:

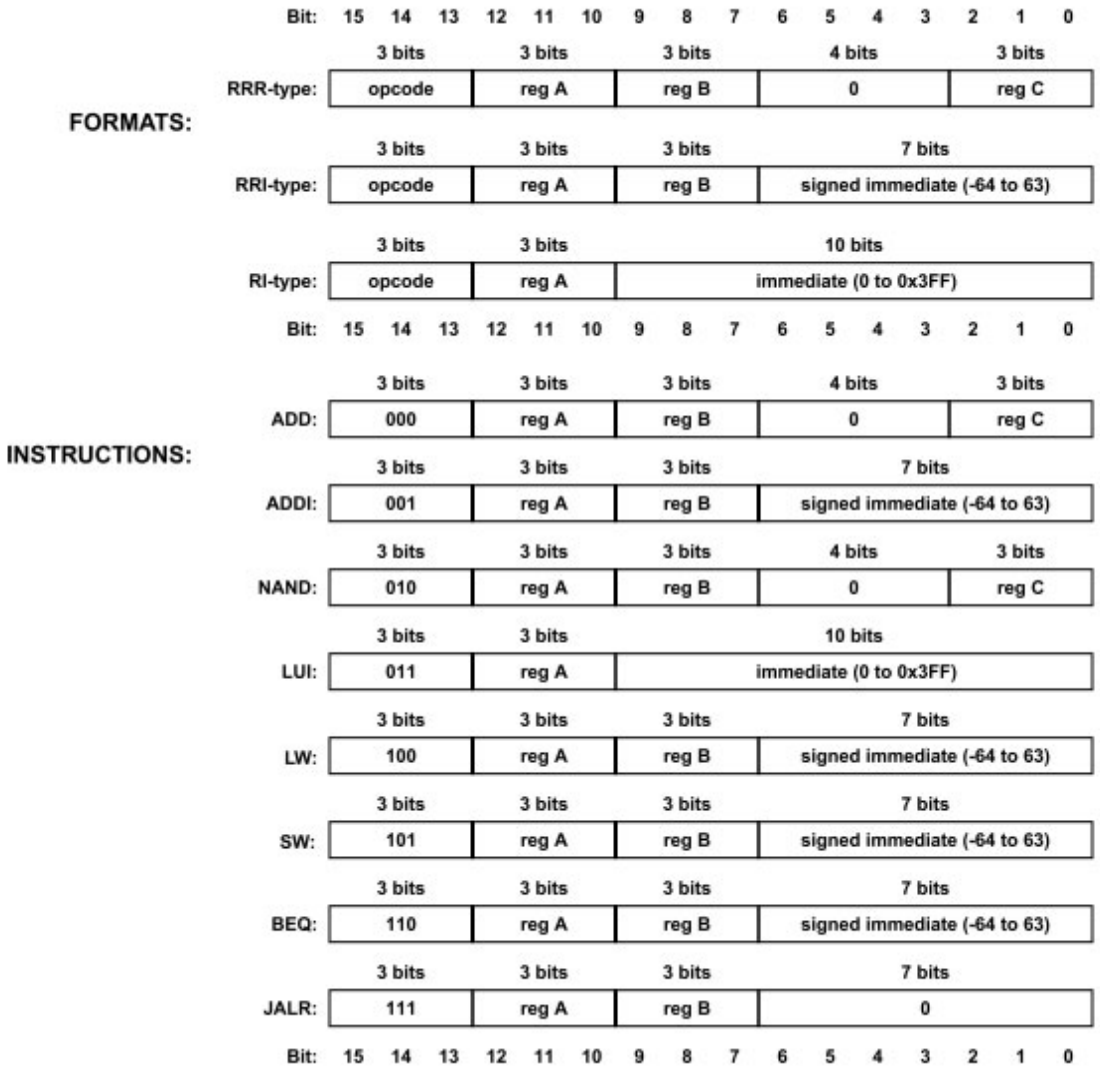
- Sequential Simulator: shows the internal dynamic behavior of the sequential version.
- Pipeline Simulator: presents the concept and mechanism of the pipeline version and highlights the pipeline hazards.
- Instruction Set Simulator: the aim of this simulator is to allow a comparison of different instructions sets and their relative performance.

These simulators are based on the RiSC16 ("Ridiculously Simple Computer") processor which has been developed by Prof. Bruce Jacob at the University of Maryland with an educational aim.

Before presenting each simulator in details, the next sections explain general considerations.

THE ORIGINAL INSTRUCTION SET

There are three machine-code instruction formats and a total of 8 instructions, illustrated in the next figure.



The next table describes the operation of the 8 instructions.

Mnemonic	Assembly Format	Action	
add	add regA, regB, regC	R[regA]← R[regB] + R[regC]	Add contents of regB with regC, store result in regA.
addi	addi regA, regB, Imm	R[regA]← R[regB] + Imm	Add contents of regB with Imm, store result in regA.
nand	and regA, regB, regC	R[regA]←!(R[regB] & R[regC])	Nand contents of regB with regC, store results in regA.
lui	lui regA, Imm	R[regA]← Imm & 0xFFC0	Place the 10 most significant bits of the 16-bit Imm into the 10 most significant bits of regA, resetting the bottom 6 least significant bits of regA.
sw	sw regA, regB, Imm	R[regA]→Mem[R[regB] + Imm]	Store value from regA into memory. Memory address is formed by adding Imm with contents of regB.
lw	lw regA, regB, Imm	R[regA]←Mem[R[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(R[regA] == R[regB]){
		PC←PC + 1 + Imm
		(or PC← label)
		}
			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.

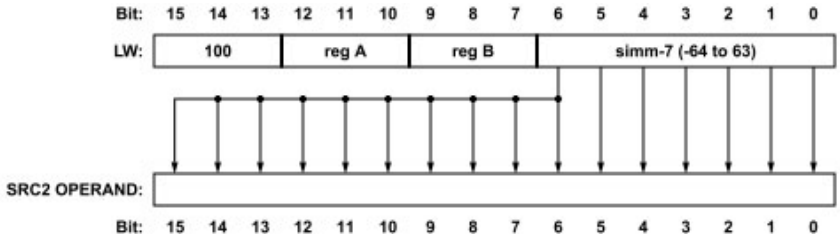
jalr	jalr regA, regB	PC← R[regB], R[regA] ← PC + 1	Branch to the address in regB. Store PC+1 into regA, where PC is the address of the jalr instruction.

Other instructions called pseudo-instructions may be used:

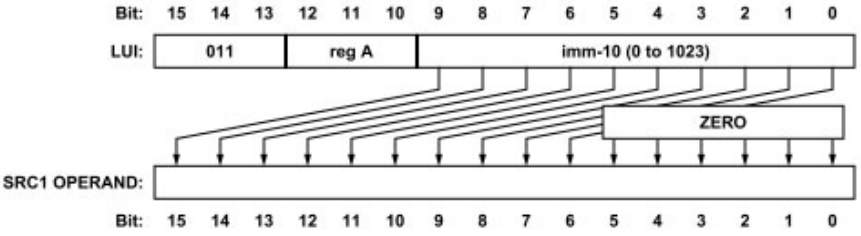
- **NOP** (= ADD 0,0,0) : this pseudo-instruction does nothing
- **RESET** (= JALR 0,0) : this pseudo-instruction reset the Program-Counter
- **MOVI rx, Imm(16bits)** (= LUI rx ImmH(10bits) + ADDI rx,rx,ImmL(6bits)) : This pseudo-instruction is a combination of a LUI and an ADDI instructions. It allows loading a 16bits constant in a register.

According to the number of bit, the immediate values are extended to 16 bits in different manners:

- 7 bits signed immediate value: the sign bit is copied on the 9 most significant bits so as to keep a two's complement number.



- 10 bits unsigned immediate value: this constant is left-shifted by 6 bits.



Each instruction is presented in the link below. Further information can be found in the "RiSC-isa" document of Prof Jacob.

ASSEMBLY LANGUAGE SYNTAX

There are two ways to write assembly code for the simulator:

- writing the instructions in the "ASM" column of the "Program Memory" window. To transform the instructions in binary code, it is necessary to click on the "Assembly" button. The pseudo-instruction MOVI will be translate in two instructions, so the next line must be empty.
- importing from a text file via the file menu. In this case, the code is assembled on-the-fly while it is imported. Several text files can be imported.

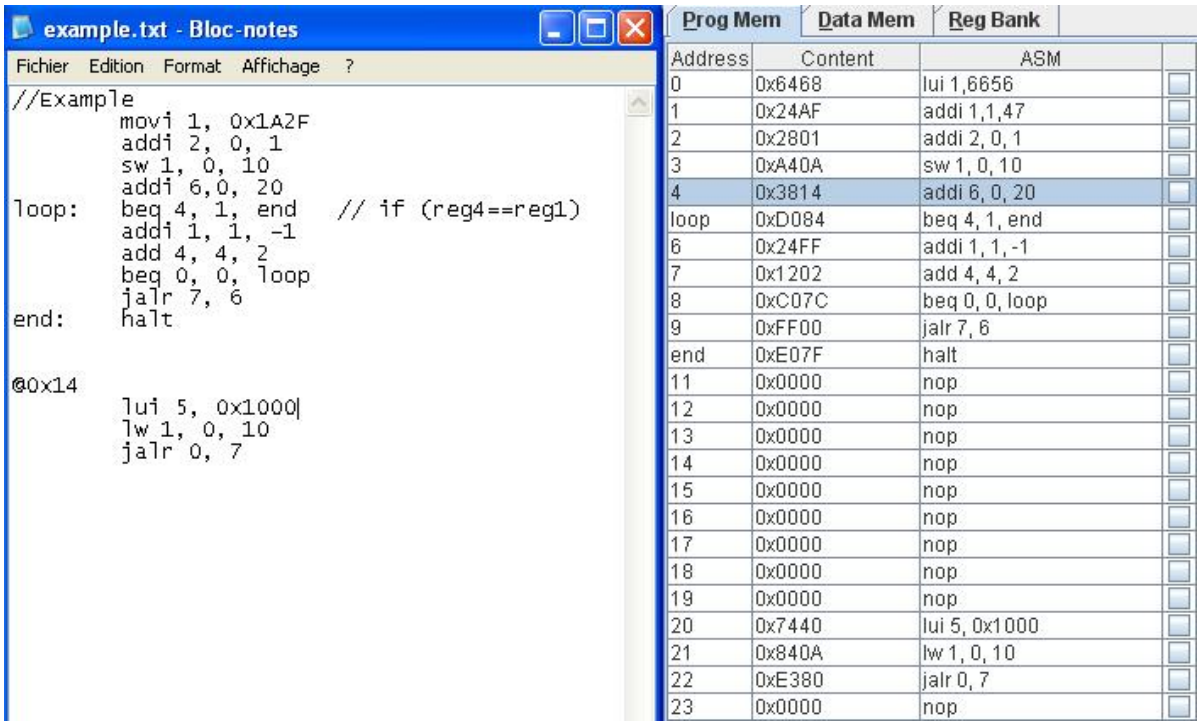
In text file, some functionalities are available:

- Comment : they must be preceded by a "//" or "#"
- Placement of instructions at specified address : the address must be prefixed by "@" and placed above instructions
- Hexadecimal value: the prefix "0x" indicate that the value is in hexadecimal format. This notation can also be used in the "Program Memory" window.
- Label: the labels are used to point the branch target address. The label must be placed before the instruction and followed by ":" (without any whitespace between label and ":").

The instruction syntax can be resumed by:

label:<tab>**opcode**<tab>**field0**, field1, field2<tab>// comments

An example of text file is shown in the next figure.



DYNAMIC BEHAVIOR OF THE SIMULATION

The majority of the blocks run in an asynchronous manner. It means that between blocks, 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, synchronization is achieved at the instruction level because the read control signal of the program memory is produced by the clock-driven control unit.

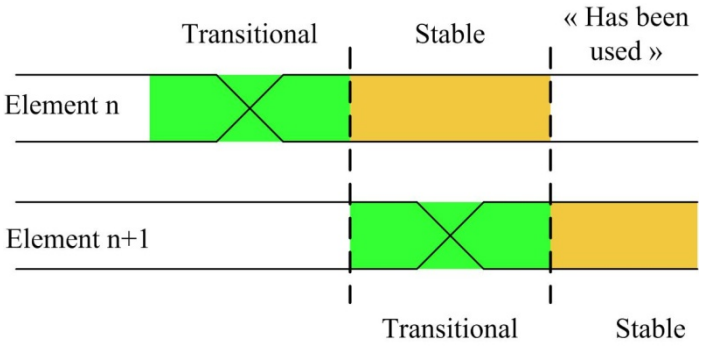
In the pipeline version, the pipeline registers introduce synchronization at the stage level.

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 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.

To easily differentiate which blocks are actually used for a given instruction, the blocks that are not going to process useful data for the current instruction are painted in grey after the *instruction* decode phase.

The next picture 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 only 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.

All blocks have a propagation delay of one clock cycle excepted for:

- The memories (ROM and RAM) and the bank register which need three half clock cycles to read or write data;
- The multiplexors take one clock cycle to select the proper input, then the data appears instantaneously at the output;
- The control unit of the sequential implementation needs three half clock cycles to decode the information. For the pipeline version, the control units show a propagation delay of one clock cycle.

RISC16 VISUAL SIMULATOR (SEQUENTIAL IMPLEMENTATION)

SEQUENCE OF EVENTS

	IF					ID/RF						
ADD 000	ROM		IR			CTL+RF(src1)		CTL+Mux_RF+RF(src1)	Mux_RF		RF(src2)	
ADDI 001	ROM		IR			CTL+RF(src1)+Sign_Ext		CTL+RF(src1)				
NAND 010	ROM		IR			CTL+RF(src1)		CTL+Mux_RF+RF(src1)	Mux_RF		RF(src2)	
LUI 011	ROM		IR			CTL+Left_Shift		CTL				
SW 101	ROM		IR			CTL+RF(src1)+Sign_Ext		CTL+Mux_RF+RF(src1)	Mux_RF		RF(src2)	
LW 100	ROM		IR			CTL+RF(src1)+Sign_Ext		CTL+RF(src1)				
BEQ 110	ROM		IR			CTL+RF(src1)+Sign_Ext		CTL+Mux_RF+RF(src1)+ADD	Mux_RF+ADD		RF(src2)	
JALR 111	ROM		IR			CTL+RF(src1)		CTL+RF(src1)				
	1	2	3	4	5	6	7	8	9	10	11	12

	EX			WB				
ADD 000	ALU			Mux_PC		RF+Mux_PC	RF+PC	RF+PC
ADDI 001	ALU			Mux_PC		RF+Mux_PC	RF+PC	RF+PC
NAND 010	ALU			Mux_PC		RF+Mux_PC	RF+PC	RF+PC
LUI 011	ALU			Mux_PC		RF+Mux_PC	RF+PC	RF+PC
SW 101	ALU		datam	datam+Mux_PC		Mux_PC	PC	PC
LW 100	ALU		datam	datam+Mux_PC		RF+Mux_PC	RF+PC	RF+PC
BEQ 110	ALU		CTL	CTL+Mux_PC		Mux_PC	PC	PC
JALR 111	ALU			Mux_PC		RF+Mux_PC	RF+PC	RF+PC
	13	14	15	16	17	18	19	20

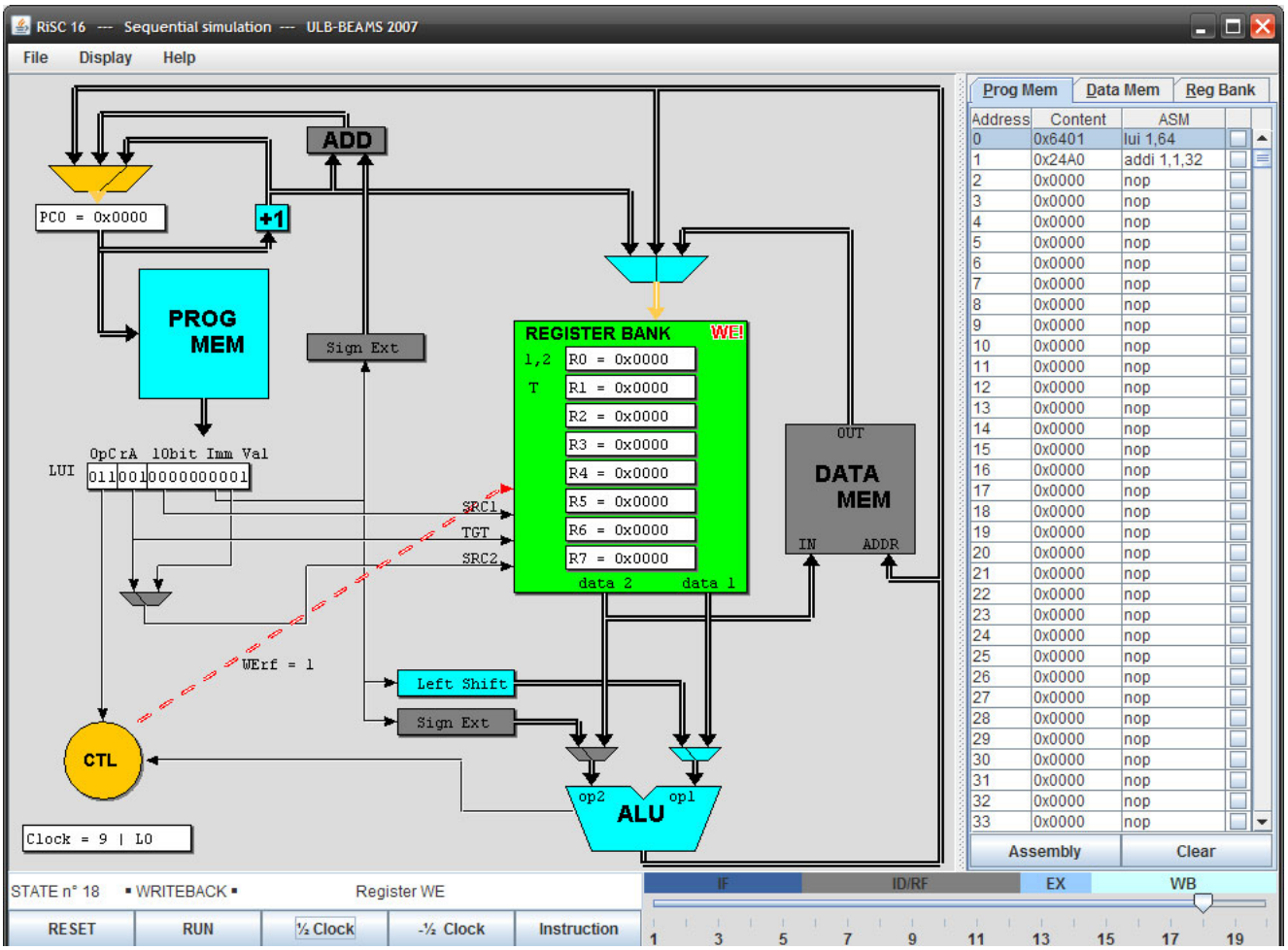
All instructions are executed in one machine cycle of 20 half clock cycles. This machine cycle is divided in four steps. These are successively:

- 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 register and the program counter receives the new address from its input multiplexer.

INTERFACE DESCRIPTION

The interface can be divided into three areas:

- The bottom area contains controls to drive the simulator
- The main area shows the elements of the RiSC16. Further information about these elements can be found in the "RiSC-seq" document of Prof Jacob.
- The right area allows viewing and editing the memories and the bank register. The last column of the program memory tab is used to place breakpoint. After editing the program memory, it is necessary to click the "Assembly" button before the changes are taken into account.



CONTROL PANEL

The processor behavior can be simulated with two time scales:

- **by half clock cycle** (*"½ Clock" button*), which is the smaller interval. This simulation mode enables the description of the internal processor mechanism in its lower details.
- **by instruction** (*"+1 Instruction" button*). 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 *"RUN" button* allows executing a program continuously. To stop program execution before a specific instruction, toggle a breakpoint by clicking the button in the right column of *"Program Memory Window"*. It is good practice to finish an assembly-program by the pseudo-instruction *halt*. The halt pseudo-instruction means “stop executing instructions” and is replaced by jalr 0,0 with a non-zero immediate field.

The *"-½ Clock" button* allow to move backward.

The *"RESET" button* reset the processor's state. After a reset, PC=0 and the register bank is empty but the data memory is preserved.

The right bottom area shows a *slider which indicates the actual* progression within the machine cycle. Below the slider, the time is shown in half-cycle units. Above the slider, you can see in which step the current instruction is. The slider can be moved forward and backward with the mouse.

MENU

- File menu:
    - Import ROM: Import the contents of program memory from a text file.
    - Import RAM: Import the contents of data memory from a text file.
    - Export ROM: Save the contents of program memory into a text file.
    - Export RAM: Save the contents of data memory into a text file.
    - Exit: Close the simulator.
- Display Menu: Change the format of values displayed in the program counter, register bank and data memory. Options are decimal, signed decimal, hexadecimal and binary format.
- Help Menu:
    - Help: Show this documentation.
    - About: Show a window with credits and contributors.

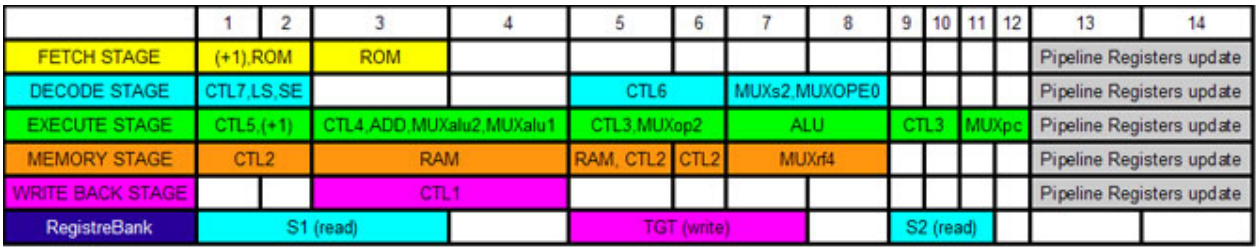
RISC16 VISUAL SIMULATOR (PIPELINED IMPLEMENTATION)

SEQUENCE OF EVENTS

The pipeline is composed of 5 stages:

- Instruction Fetch: The program memory receives the address of the instruction from the program counter, and the program counter is incremented.
- Instruction Decode: The opcode is decoded and the operands are fetched from the registers bank.
- Instruction Execute: The arithmetical-logical unit executes the operation.
- Memory Access: This stage is used for the data memory access.
- Write-back: The result is stored in the target register.

Each machine cycle, which is the same as the time to execute one stage of the pipeline, is made of 14 half clock cycles.



A description of the RiSC16 pipeline version can be found in the "RiSC-pipe" document of Prof Jacob.

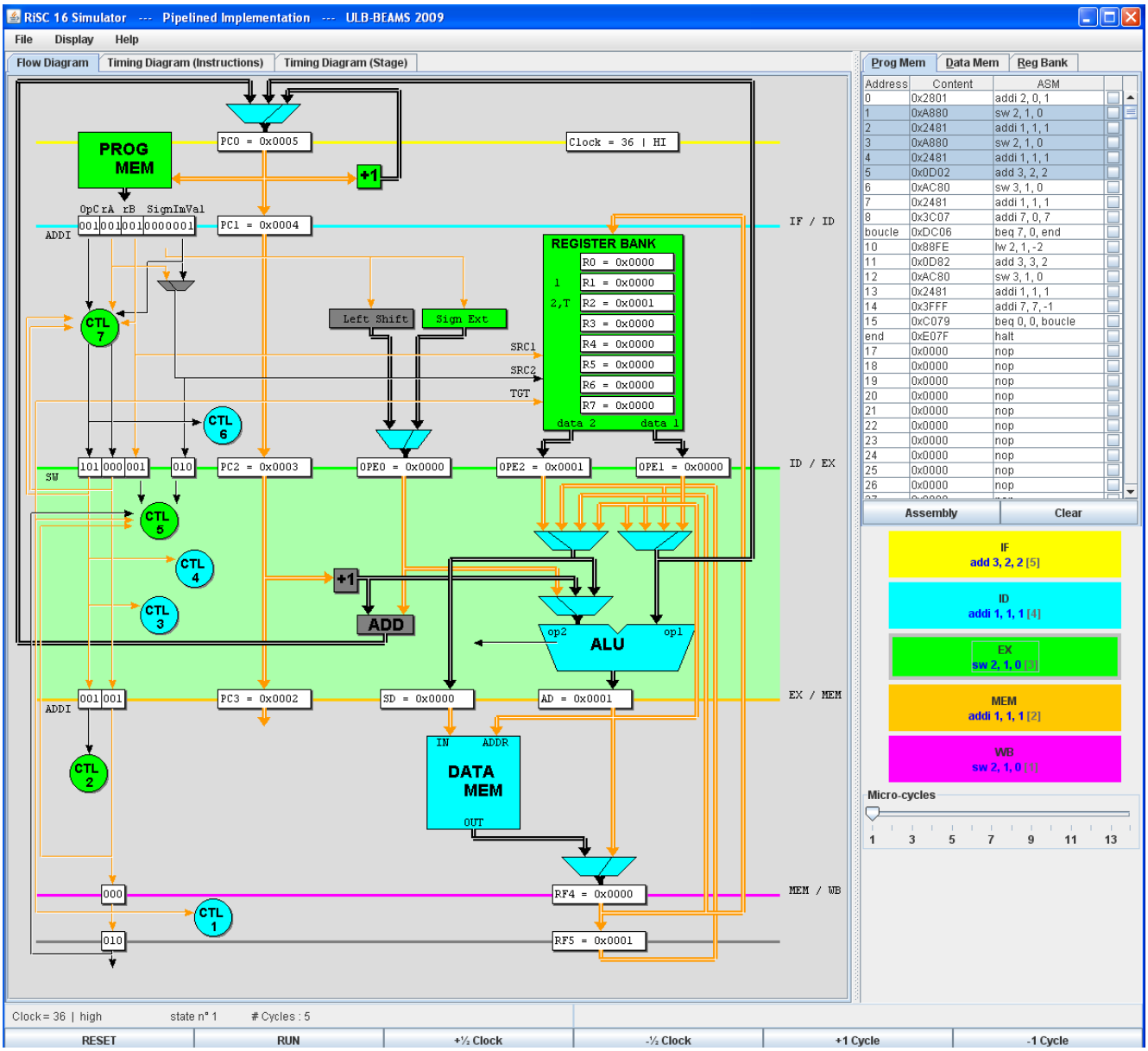
INTERFACE DESCRIPTION

The interface of this simulator is similar to the sequential one. The *slider* allows only moving one cycle forward and backward.

The colored stack on the right side of the screen shows at every moment which instruction is in each pipeline stage. The main window has got three tabs:

- Flow Diagram: it shows the block diagram of the RiSC16 pipeline implementation. The elements of this diagram are designed to distinguish the different stage of the pipeline. Click on a stage of the stages stack diagram at the right of the screen to highlight all the elements of this stage in the block diagram.
- Instructions-Timing Diagram: this diagram shows each stage of each instruction in function of cycles. The number of cycle per instruction (CPI) provides a performance index of the pipeline.
- Stage-Timing Diagram: it shows for each stage which instruction is executed in function of cycles. The number in the box indicates the address of the instruction.

The last two diagrams are helpful to emphasize the effects of the hazards on the pipeline performance.



PIPELINE HAZARDS

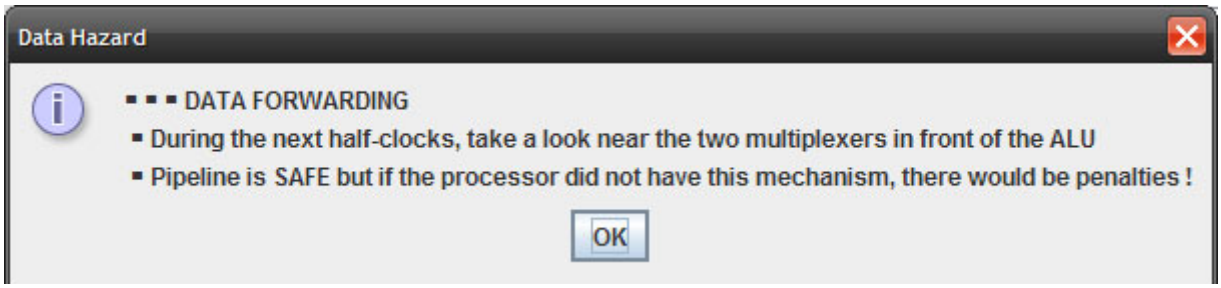
A hazard is a potential problem that can happen in a pipelined processor. There are three fundamental types of hazard: data hazards, branching (control) hazards, and structural hazards. Data hazards can be further divided into Write After Read, Write After Write, and Read After Write (RAW) hazards.

In this RiSC16 implementation, only control hazards and data hazards (RAW) may occur.

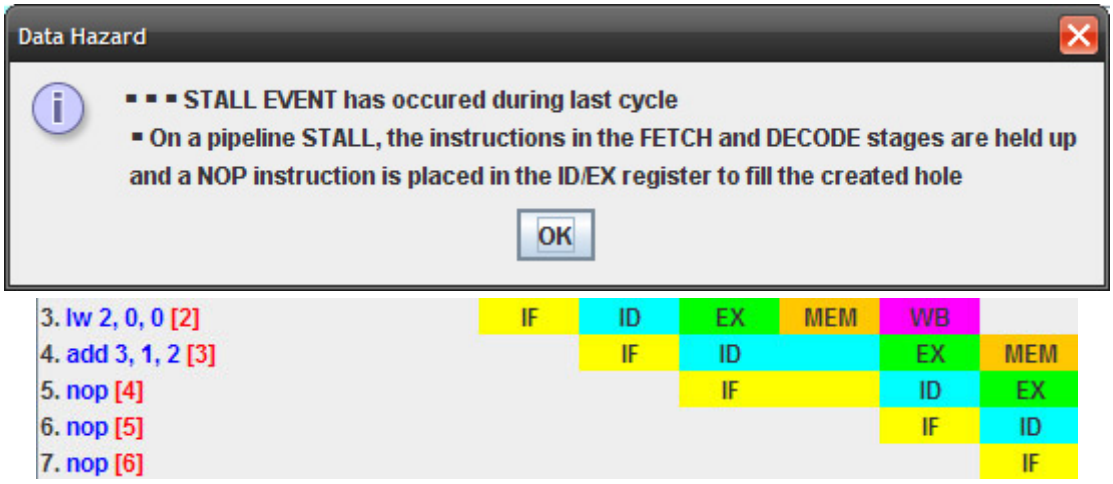
As you can see on the figures below, a dialog informs when a hazard has occurred. You can deactivate these dialogs in " *Display > Alerts* ".

Data Hazards

Most RAW data hazards are resolved using data forwarding. When it happens, the following message pops up to inform the user.

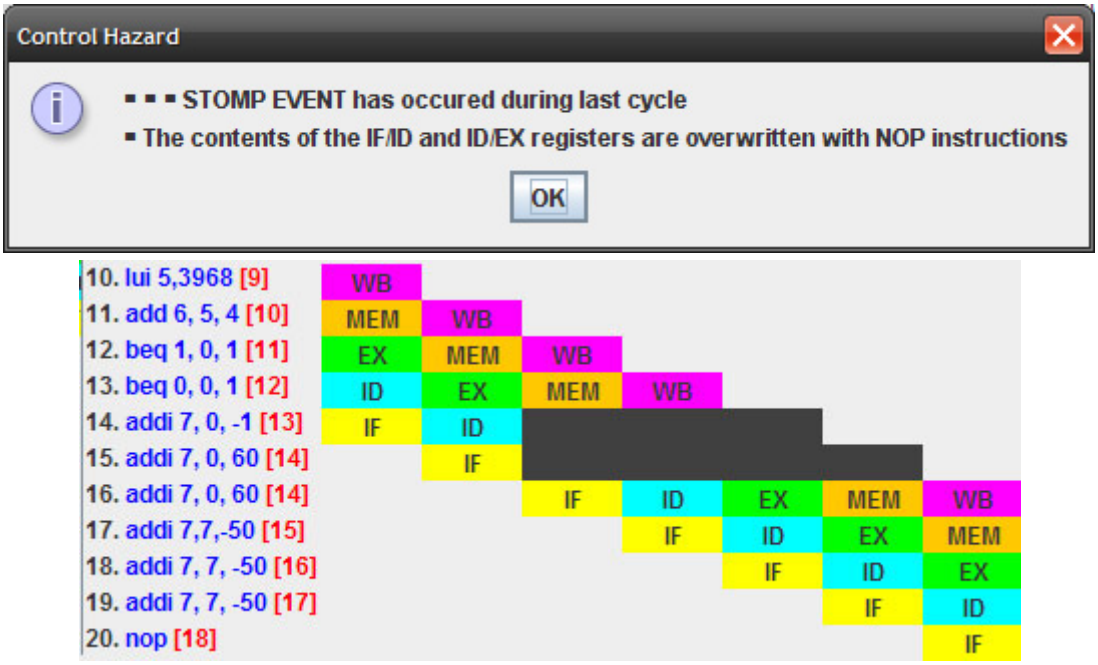


If the instruction currently in the execute stage (held in the ID/EX register) is a LW and targets any register that the instruction in decode stage uses as a source register, a STALL event happens. This event causes a penalty of one cycle.



Control Hazards

If the instruction in the execute stage is a BEQ and the condition is true or if the instruction is a JALR,a STOMP event happens. This event causes a penalty of two cycles.



RISC16 INSTRUCTION SET SIMULATOR

This simulator is a tool that emphasizes the limits of the very reduced RiSC16 instructions set. It allows to define others architectures that have news instructions and a different number of registers.

ENHANCED ARCHITECTURES

There are four preset architectures. The first is the original RiSC16 and has got the 8 instructions already presented. The instructions set of the architecture "Special IS[1]" is made up of the 8 original instructions, plus 8 new instructions. It is presented in the next table.

	Bit:	16 15 14 13	12 11 10	9 8 7	6 5 4 3	2 1 0
INSTRUCTIONS:	ADD	0000	reg A	reg B	(-8 to 7)	reg C
	SUB	0001	reg A	reg B	(-8 to 7)	reg C
	NAND	0010	reg A	reg B	0000	reg C
	LUI	0011	reg A	immediate (0 to 0x3FF)		
	SHL	0100	reg A	reg B	(-8 to 7)	reg C
	SHA	0101	reg A	reg B	(-8 to 7)	reg C
	NOR	0110	reg A	reg B	0000	reg C
	XOR	0111	reg A	reg B	0000	reg C
	ADDI	1000	reg A	reg B	signed immediate (-64 to 63)	
	SHIFTI	1001	reg A	reg B	signed immediate (-64 to 63)	
	BL	1010	reg A	reg B	signed immediate (-64 to 63)	
	BG	1011	reg A	reg B	signed immediate (-64 to 63)	
	LW	1100	reg A	reg B	signed immediate (-64 to 63)	
	SW	1101	reg A	reg B	signed immediate (-64 to 63)	
	BEQ	1110	reg A	reg B	signed immediate (-64 to 63)	
	JALR	1111	reg A	reg B	0000000	

An additional bit for the opcode is required to code the 16 instructions. So, the instruction size grows from 16 to 17 bits. The architecture "Special IS[1] – 16 reg – Instruction 24 bits" has got the same instruction set but it has a bank of 16 registers and the instruction size is increased to 24 bits. It allows lager immediate value as it shows in the next table.

	Bit:	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4	3 2 1 0
INSTRUCTIONS:	ADD	0000	reg A	reg B	(-128 to 127)	reg C
	SUB	0001	reg A	reg B	(-128 to 127)	reg C
	NAND	0010	reg A	reg B	00000000	reg C
	LUI	0011	reg A	immediate (0 to 0xFFFF)		
	SHL	0100	reg A	reg B	(-128 to 127)	reg C
	SHA	0101	reg A	reg B	(-128 to 127)	reg C
	NOR	0110	reg A	reg B	00000000	reg C
	XOR	0111	reg A	reg B	00000000	reg C
	ADDI	1000	reg A	reg B	signed immediate (-2048 to 2047)	
	SHIFTI	1001	reg A	reg B	signed immediate (-2048 to 2047)	
	BL	1010	reg A	reg B	signed immediate (-2048 to 2047)	
	BG	1011	reg A	reg B	signed immediate (-2048 to 2047)	
	LW	1100	reg A	reg B	signed immediate (-2048 to 2047)	
	SW	1101	reg A	reg B	signed immediate (-2048 to 2047)	
	BEQ	1110	reg A	reg B	signed immediate (-2048 to 2047)	
	JALR	1111	reg A	reg B	000000000000	

These news instructions are presented below.

ARITHMETICAL INSTRUCTION

SUB: R[regA] ← R[regB] – R[regC]

The contents of the regB and the regC are subtracted. The result is stored in regA.

LOGICAL INSTRUCTIONS

NOR: R[regA] ←NOT(R[regB] | R[regC])

The result of the NOR operation between regB and regC is stored in regA.

XOR: R[regA] ←(R[regB] ^ R[regC])

The result of the XOR (exclusive OR) operation between regB and regC is stored in regA.

The next three instructions aren't included in the architecture "Special IS[1]" but can be added in a new architecture as it will be describe later.

OR: R[regA] ← (R[regB] | R[regC])

The result of the OR operation between regB and regC is stored in regA.

XNOR: R[regA] ←NOT(R[regB] ^ R[regC])

The result of the XNOR operation between regB and regC is stored in regA.

AND: R[regA] ← (R[regB] & R[regC])

The result of the AND operation between regB and regC is stored in regA.

BRANCH INSTRUCTIONS

BL (Branch if Lower): if(R[regA]< R[regB]) {PC ← PC_{BL}+1+immed} else PC←PC_{BL}+1}

The instruction compares the contents of the regA and the regB. If the content of regA is lower to regB, the PC will be loaded with PC_{BL}+1+imm(extend), else, it will be loaded with PC_{BL}+1.

BG (Branch if Greater): if(R[regA]> R[regB]) {PC ← PC_{BG}+1+immed} else PC←PC_{BG}+1}

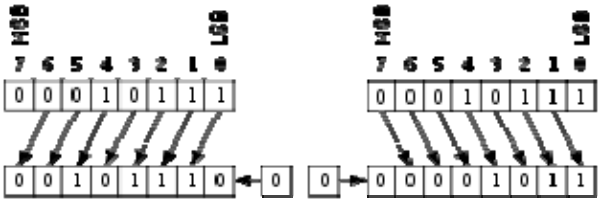
The instruction compares the contents of the regA and the regB. If the content of regA is greater to regB, the PC will be loaded with PC_{BG}+1+imm(extend), else, it will be loaded with PC_{BG}+1.

SHIFT INSTRUCTIONS

The shift operations allow to multiply (right shift) or to divide (left shift) a number by a power of two. These require the implementation of a barrel shifter in the ALU.

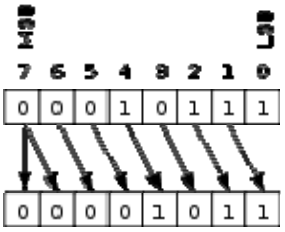
SHL (Logical SHIFT): R[regA] ← R[regB] << R[regC] or R[regB] >> R[regC] according to the sign of regC

This instruction shift regB by the number of bits specified in regC. The result is stored in regA. If regC is positive, the shift will execute to the left, otherwise, it will execute to the right.



SHA (Arithmetical SHIFT): R[regA] ← R[regB] << R[regC] or R[regB] >> R[regC] according to the sign of regC

This instruction shift regB by the number of bits specified in regC. The result is stored in regA. If regC is positive, the shift will execute to the left, otherwise, it will execute to the right with a preservation of the sign bit (see next picture).



SHIFTI (Immediate SHIFT): R[regA] ← R[regB] << immed or R[regB] >> immed

This instruction shift regB by the immediate constant. The result is stored in regA. The 7 bits immediate value (for the 17 bits size instruction) is interpreted as follows: the 5 least significant bits give the signed shift amplitude (5 bits: -16 to +15). The sixth bit specifies the mode (1 for arithmetical, 0 for logical). The seventh bit isn't used.

SPECIAL INSTRUCTIONS SET 2

This "Special IS[2]" differ from the precedent instruction set by one instruction. The BG instruction is replaced by a multiplication instruction.

MUL regA, regB, regC: R[regA-1] ← (R[regB] * R[regC]) >>16, R[regA] ← (R[regB] * R[regC])%16

This instruction multiplies the content of regB and regC. The 16 low significant bits of the result are stored in regA and the 16 most significant bits are stored in the register previous regA. This representation is called "big-endian": the most significant bits are at the smaller address.

CARRY AND OVERFLOW MANAGEMENT – CONDITIONAL INSTRUCTIONS

In the original architecture, there is no overflow management. The overflow management mechanism in the new architectures takes advantage of the unused bits in RRR type instructions to realize a branch in the case of an overflow (bits from 6 to 3 for the add, sub, sha and shl instructions). These four bits allow a relative jump in program memory from -8 to 7 (instruction size of 17 bits).

In assembler code, the relative jump must be placed at the end of the instruction. Like the branch instructions, labels can be used.

Example: add 3,1,2, [immed]

add 3,1,2, -8 In the case of overflow, the PC will be loaded with PC_{add}+1-8

add 3,1,2, label In the case of overflow, the PC will be loaded with the address pointed by the label

The "Architecture – Signed or Unsigned" menu allows to define if the branchs take place in case of an overflow (signed arithmetic) or in case of a carry (unsigned arithmetic).

USER DEFINED ARCHITECTURE

The user can configure architecture by:

- Selecting the instructions set ("Architecture – Instructions set – Other" menu)
- Choosing the number of registers ("Architecture – Registers" menu)
- Changing the size of the immediate values ("Architecture – Imm & Instru Sizes" menu)
- Defining the type of arithmetic: Signed or Unsigned ("Architecture – Signed or Unsigned" menu). This involves not only the overflow management, but also the branch operation.

INTERFACE DESCRIPTION

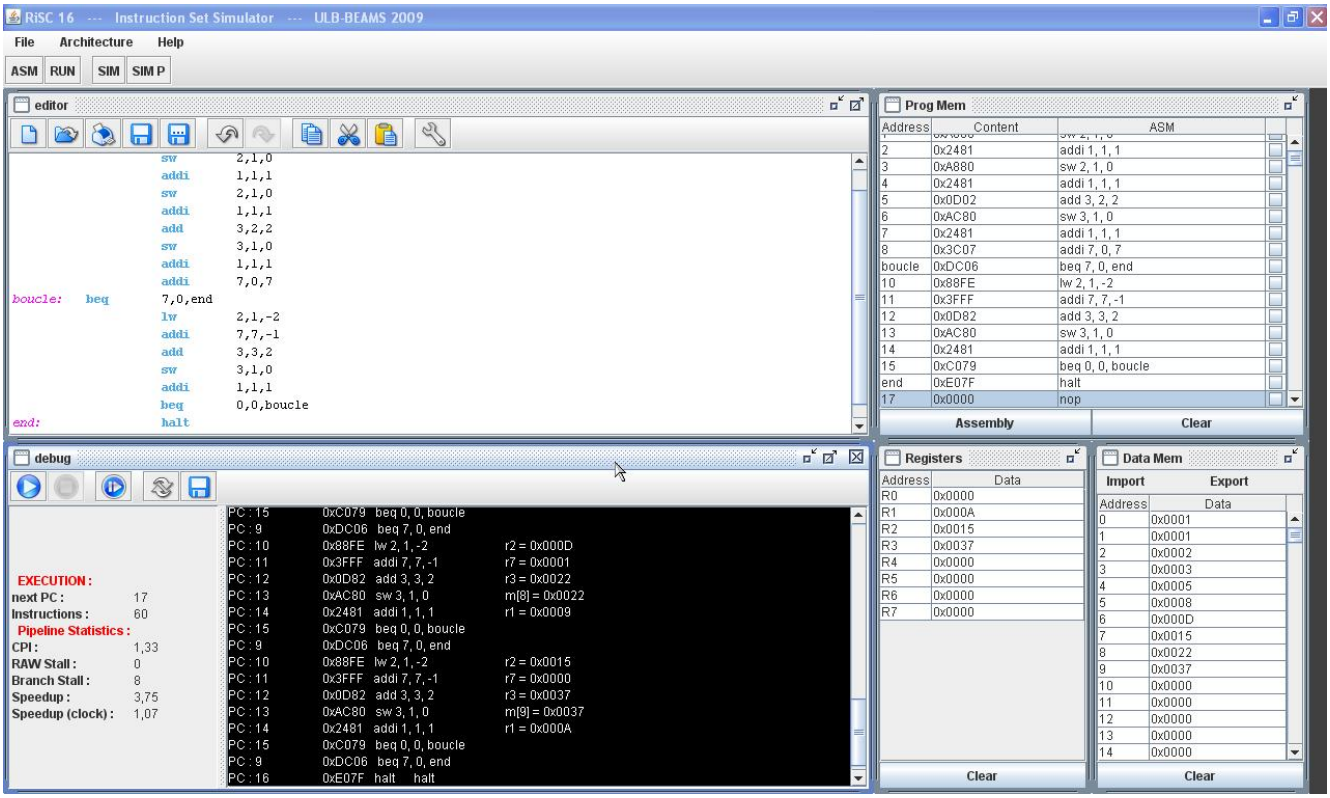
As shown in the next figure, the simulator is made up of:

- A syntax-aware editor with different colors for labels, opcodes and comments,
- A program memory window,
- A data memory window,
- A registers window and
- A debug window.

The last four windows appear only after a click on the run button. This action also assembles the code in the editor. After a change of the architecture, it is necessary to click on this button so that the modification will be taken into account.

The other buttons are:

- ASM: a click on it assembles the code in the editor.
- SIM: it starts the sequential simulator with the code in the editor.
- SIM P: it starts the pipeline simulator with the code in the editor.



The menu of the debug window is composed of:

- Play: execute the program until a breakpoint or a halt instruction.
- Stop: stop the execution of the program.
- Next: instruction by instruction mode.
- Reset: reset the PC but it don't reset the contents of the registers and the data memory.
- Save: allows saving the trace of the execution in a file.

This widow contains some statistics on the execution:

- CPI: cycles per Instruction
- RAW Stall: number of resource conflict hazards
- Branch Stall: number of branch executed.
- Speedup: pipeline speedup index which indicates by which factor the pipeline version is faster compared to a hypothetical sequential version using the same sequencing of each stage as the pipelined RiSC16, that is to say 70 half clock cycles (5 stages x 14 half clock cycle) per instruction.

$$Seepdup = \frac{pipeline\ depth}{CPI}$$

- Speedup (clock): this index is the real gain taking the actual cycle length of the sequential RiSC16 (10 clock cycles) into account.

$$Speedup\ (clock) = \frac{T_{SEQ}}{T_{PIPE} \cdot CPI} = \frac{10}{7 \cdot CPI}$$

ACKNOWLEDGMENT

We would like to thank for their contribution to the project to:

- David Cross
- Laurent Englebin
- Michael Huysman
- Marc Jaumain
- Prof. Pierre Mathys
- Quentin Monneaux
- Michel Osée
- Aliénor Richard