Labo n° 2
Microprocessor Architectures [ELEC-H-473]
RiSC16: internal processor behaviour 2/3

2014–2015

Introduction

This lab is about the internal behaviour of a RISC processor and goals are to:

- understand the internal behaviour of a simple pipelined processor, especially pipeline hazards.
- write and test some programs in assembly code for this specific RISC processor and compare performances and implementation with version from Lab 1.
- watch this code running in simulation .

1 RiSC16 – pipelined version

To reach these goals, you will use another simulator for the pipelined version of the RiSC16 initially designed by Bruce Jacob. The processor has exactly the same features that the sequential version but is implemented using a 5-stage pipeline. Several blocks were added, as shown on Figure 1.

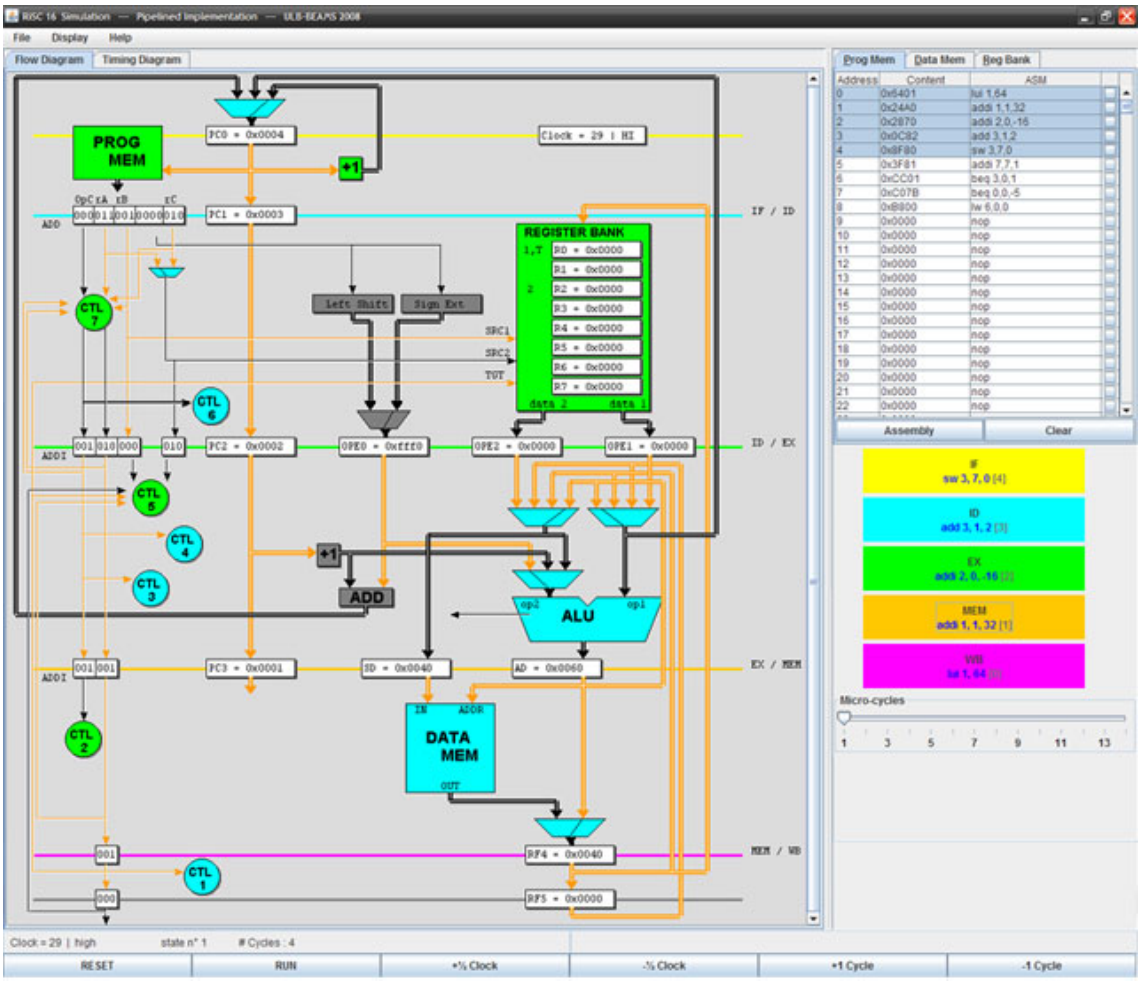

Figure 1: Pipelined RiSC-16 processor

The pipeline has 5 stages and instructions are split into:

1. **IF** (Instruction Fetch): instruction is read from program memory. `PC` is also incremented.
2. **ID** (Instruction Decode): opcode is decoded and operands for ALU are read in the register bank
3. **EX** (EXecute): ALU processes the operation
4. **MEM** (MEMory): access to Data Memory
5. **WB** (Write-Back): result is written in destination register

All stages are completed in one clock cycle. The interface presented in the simulator is designed in the same way that the sequential version to ease switching between the two implementations.

There are several zones in the simulator window:

- The execution zone has two tabs:
  - The first one shows the steps of a cycle
  - The second one shows the cycle-instruction diagram (see Figure 2). The number of cycles per instruction (CPI) is also shown.

Stage:	1	2	3	4	5
lui 1, 64 [0]	IF	ID	EX	MEM	WB
addi 1, 1, 32 [1]		IF	ID	EX	MEM
addi 2, 0, -16 [2]			IF	ID	EX
add 3, 1, 2 [3]				IF	ID
sw 3, 7, 0 [4]					IF

Figure 2: Pipeline execution

- The memory zone on the right has 3 tabs to show and edit content of program memory, data memory and file register.
- Another zone on the right details the content of the pipeline for each stage and the cursor shows the current step of the cycle.
- The lower part groups the interaction possibilities: buttons provide reinitialisation (reset), execution (run), instruction execution and undo last clock-cycle capabilities.
- The central zone shows the blocks for this architecture.

Internal blocks of the processor have two particularities:

1. The control unit is split into several smaller units. Each one has a specific role either for a flawless execution or for managing pipeline hazards.
2. Between each stages, Pipeline registers were added to store states variables related to the current stage. Among these variables, we can find the opcode, address of registers, state of the `PC`...

Thanks to these registers the pipeline can work as expected because they are the link between stages. Only pipeline registers and control unit are synchronous. Other blocks are asynchronous. The Figure 3 shows the execution sequence of instructions.

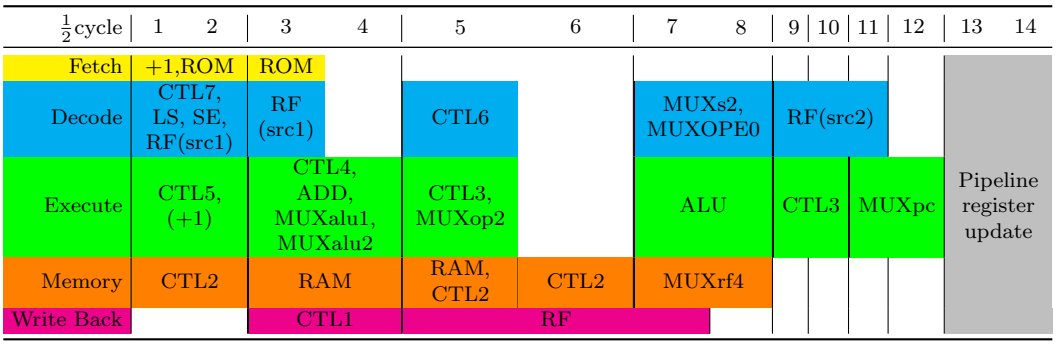

Figure 3: Instruction sequencing

The architecture performance can be measured using the cycles per instruction (CPI). An ideal CPI for a pipeline should be 1 after each cycle, that means that an instruction ends at each cycle.

But actually, the pipeline is subject to hazards. In this RiSC16 architecture, only dependency conflicts (Data Hazards) and control problems (Control Hazards) can occur.

1.1 Dependency hazards

Dependency conflicts happens when an instruction needs a result from a previous instruction which is not completed yet. Only incidents of type "Read After Write" can happen in the RiSC16. To solve this problem, several possibilities exist:

- Data forwarding: a value is taken from a previous stage before its written in the register bank and is used as input to the ALU. Thanks to this mechanism, there is no penalty. Data forwarding is signalled by a message in the simulator.
- When the `LW` instruction is in stage EX and an instruction in ID stage uses an operand at the same address than the destination register for the `LW` instruction, a Data hazard occurs. This hazard cannot be solved using Data forwarding, the only way to solve the issue is to stall the pipeline and so add a "bubble". This will add a penalty of one cycle. This is signalled by a *stall event* in the simulator.

1.2 Control conflicts

Control conflicts happens because of branch and call procedures. The address of the next instruction is computed in the execute stage an so, when a jump occurs, instructions loaded in the 2 previous stages are irrelevant and are discarded. This accident adds a penalty of 2 cycles and is named *stomp event* in the simulator.

Various information messages are displayed when an accident occurs. They can be disabled using the "Display > Alerts" menu.

2 Manipulation

Question 1. Reuse the example of the first lab and draw the cycle-instruction diagram for the 25 first cycles. Compare the result with the simulation. Beware, accidents happen in simulation.

Question 2. What is the function of each control unit? Which are used during accidents solving?

Question 3. Correct the program to suppress data hazards.

Question 4. Observe the CPI at the end of program execution and compare with the result of first lab.

Question 5. Find a way to compute the CPI using:

- pipeline *depth* (here: 5)
- instruction fully executed: *instruction*
- number of stall events: *stall*
- number of stomp events: *stomp*

Question 6. How would you modify the hardware to improve the design?

Question 7. Complete the program of first lab (multiplication of two registers).