Simulation of a 5-stage pipelined Nios II

Learning Goal: Processor pipeline.

Requirements: Nios2Sim.

1 Introduction

During this lab, you will observe the mechanisms of a pipelined processor. For that, you will use the pipeline simulator of Nios2Sim.

2 The Pipeline Simulator

Nios2Sim provides a 5-stage pipeline simulator which can be enabled during the simulation by selecting Nios II >Pipeline Simulator >5-stage pipeline. Nios2Sim models a relatively simple pipeline:

- Harvard's architecture.
- 5 stages (*Fetch*, *Decode*, *Execute*, *Memory* and *Writeback*).
- All instructions go through these 5 stages, even if some of them are not used.
- The stalls, flushes and forwarding paths can be enabled/disabled through the options.
- If a stage stalls, all its preceding stages stall as well.

3 Exercise 1

- Download the latest version of Nios2Sim.
- Copy the following assembly code in a new file:

```
.equ  RAM, 0x1000
main:
    addi t0, zero, 1    ; t0 = 1
    addi t1, zero, 2    ; t1 = 2
    add  t2, t0, t1     ; t2 = t0 + t1
    sub  t3, t2, t0     ; t3 = t2 - t0
    stw  t0, RAM(zero) ; RAM[0] = t0
    ldw  t4, RAM(zero) ; t4 = RAM[0]
    add  t5, t4, t0     ; t5 = t4 + t0
end:
    br   end
    nop
    nop
```

- Switch to the simulation view (Nios II >Start Simulation).
- Enable the simulation of the pipeline (Nios II >Pipeline Simulator >5-stage pipeline).
- Enable the W→D forwarding path (Nios II >Pipeline >Enable forwarding paths >W→D), and make sure that all the remaining options are **disabled** (i.e., stalls, flushes, E→E and M→E forwarding paths).
- Execute the code step by step.

3.1 Simulation

- What should the values of registers `t2`, `t3`, `t4` and `t5` be at the end of a normal execution?
- Why don't we get the correct values in the simulation?
  - Draw the pipeline execution diagram.
  - Show all the dependencies.

3.2 Inserting NOPs

- Insert `nop` instructions to modify the program and get the correct result.
- Draw the new pipeline execution diagram.
- Verify that you get the correct results in the simulation.

3.3 Stalling

- Enable the stalls in the pipeline simulation (Nios II >Pipeline Simulator >Enable stalls).
- Remove or comment the `nop` instructions that you added to the program during the previous exercise.
- Draw the pipeline execution diagram. This time you must insert bubbles instead of executing `nop` instructions.
- Verify that you get a correct behavior in the simulation.
- Do you know of a technique that can avoid the use of bubbles, and improve the performance of the pipeline?

3.4 Forwarding

- Enable all the forwarding paths of the pipeline (Nios II >Pipeline Simulator >Enable forwarding paths >...).
- Restart the simulation.
- Execute the program step by step and observe the evolution of the pipeline.
- How are the data hazards solved? Give a description for each different case.
  - Draw the new pipeline execution diagram.
  - For each instruction, show the forwarding paths that are used.
- A stall occurs during the execution of the program. This is due to the dependence between the `ldw` and `add` instructions. Is there any way for the compiler to avoid this?

4 Exercise 2

- Copy the following assembly code in a new file:

```
main:
    addi  a0, zero, data       ; a0: Array memory address
    addi  a1, zero, 2          ; a1: Number of elements
    call  proc
    nop
    nop
end:
    br    end
    nop
    nop

proc:
    add   v0, zero, zero       ; v0 = 0
    add   t0, zero, zero       ; t0 = 0
proc_outer:
    bge   t0, a1, proc_return ; if (t0 >= a1) goto return
    nop
    nop
    ldw   t3, 0(a0)            ; t3 = mem[a0]
    addi  t4, zero, 32         ; t4 = 32
proc_inner:
    beq   t4, zero, proc_next ; if (!t4) goto next
    nop
    nop
    andi  t1, t3, 1            ; t1 = t3 & 1
    add   v0, v0, t1           ; v0 = v0 + t1
    srli  t3, t3, 1            ; t3 = t3 >> 1
    addi  t4, t4, -1           ; t4 = t4 - 1
    br    proc_inner           ; goto inner
    nop
    nop
proc_next:
    addi  t0, t0, 1            ; t0 = t0 + 1
    addi  a0, a0, 4            ; a0 = a0 + 4
    br    proc_outer           ; goto outer
    nop
    nop
proc_return:
    ret                        ; return to caller
    nop
    nop

data:
    .word 0x080A0103
    .word 0x0F0F0F0F
```

4.1 Simulation

- Describe the program.
- What value should the procedure return in the `v0` register?
- How many cycles are necessary to complete the execution?
  - During the simulation, in the **Cycles Count** field of the **Control** tab, you can see how many cycles have been spent so far.
  - Before executing the simulation, make sure that the stalls and all the forwarding paths are enabled. The flushes must be disabled.
  - Remember that, in the simulation view, you can add/remove a breakpoint on a line by double clicking on it.

4.2 Branch

- Why are there `nop` instructions after the branches and jump instructions?
- What is the branch penalty?
- Are the `nop` instructions really executed? What happens if we remove them?
- Is there any way to hide the branch penalty (by only modifying the assembly code)?

4.3 Performance

- Modify the `proc` procedure by removing `nop` instructions whenever it is possible. You can switch the positions of the instructions but do not modify the instructions themselves. The functionality of the procedure must be the same. Justify each of your modifications.
- How many cycles are necessary to complete the execution now? Compare your result with the first version.

5 Submission

Nothing needs to be submitted for this lab.