Lab. 2: "PIPELINED INSTRUCTION UNIT (I)"

Computer Architecture and Engineering
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Lab goals:

- Get familiar with a simulator of a pipelined computer
- Analyze the impact of data and control hazards on the performance of a pipelined instruction unit
- Write programs in RISC V assembler

Lab work:

The simulator of a 5-stage pipeline RISC V computer

The simulator that will be used in this lab session can carry out a cycle-wise simulation of the execution of RISC V instructions. It supports the rv64i set of the RISC V instructions. It has separate instructions and data caches (*Harvard* architecture). Its register file has two read and one write ports. Control hazards are solved using *stalls* and *predict-not-taken*, with three possible sizes for the *branch latency* (one, two and three cycles). Data hazards are solved by inserting stalls or using *data forwarding*. It is worth mentioning that the simulated datapath changes according to the technique used for solving the hazards.

The simulator can be executed from the command line:

```
riscv-m -s <results> -d <data-hazards> -c <control-hazards> -f <archive.s>
```

where:

- <results> defines how the simulation result will be provided. Several alternatives are available:
  - **tiempo**: Displays in the terminal the execution time.
  - **final**: Displays in the terminal the execution time, registers and memory content after the execution of the provided archive.
  - **html**(*): Generates several html archives illustrating both the state of the execution cycle-wise and the final execution results. These results can be checked by opening the file **index.html** with the web browser. This is the default option.

    Adding the `-j` option generates a single file `index.htm` that includes all the results.
  - **html-final**: Generates the **final.html** html archive, with the final result of the execution.

- <data-hazards> defines how data hazards are solved. There are three alternatives:
  - **n**: There is no logic to solve data hazards.
  - **p**: Data hazards are solved by inserting stalls.
  - **c**(*): Data hazards are solved by using the forwarding technique and inserting stalls whenever necessary.
- <control-hazards> determines how control hazards are solved. There are six alternatives:
  - **s3**: Control hazards are solved by inserting 3 stalls.
  - **s2**: Control hazards are solved by inserting 2 stalls.
  - **s1**: Control hazards are solved by inserting 1 stall.
  - **pnt3**(*): Control hazards are solved by using *predict-not-taken* and inserting 3 stalls whenever the branch is taken. This is the default option.
  - **pnt2**: Control hazards are solved by using *predict-not-taken* and inserting 2 stalls whenever the branch is taken.
  - **pnt1**: Control hazards are solved by using *predict-not-taken* and inserting 1 stall whenever the branch is taken.
- <archive.s> is the name of the file holding the assembler code.

(*): Default option.

Example of a RISC V program

The following assembler code comprises a loop to compute the vector operation $\vec{Z} = a + \vec{X} + \vec{Y}$:

```
    .data
    # Vector x
x:  .dword 0,1,2,3,4,5,6,7,8,9
    .dword 10,11,12,13,14,15

    # Vector y
y:  .dword 100,100,100,100,100,100,100,100,100,100
    .dword 100,100,100,100,100,100

    # Vector z
    # 16 elements are 16*8=128 bytes
z:  .space 128

    # Scalar a
a:  .dword -10

    # Code
    .text

start:
```

```
    addi t0,gp,x
    addi t3, gp, y
    addi t1,gp,y
    addi t2,gp,z
    ld t4,a(gp)

loop:
    ld a0,0(t0)
    add a0,t4,a0
    ld a1,0(t2)
    add a1,a0,a1
    sd a1,0(t2)
    addi t0,t0,#8
    addi t1,t1,#8
    addi t2,t2,#8
    slt t5,t0,t3
    bnez t5,loop     # bne t5, zero, loop

    ori a7, zero, 10 # Program exit system call
    ecall
```

1. The program is available in the file `apxpy.s`. The following command shows how to execute it, showing the results and solving the data hazards using stalls, and the control hazards using 3 stalls:

   ```
   riscv-m -d p -c s3 -f apxpy.s
   ```

   Open now the file **index.html** using a web browser. This will show the processor configuration, the memory contents, and a set of links that can be used to navigate through the results:

   - INICIO: Displays the processor configuration and the initial memory state.
   - FINAL: Displays the performance results from the execution, the processor configuration, and the final memory contents.
   - Estado: Displays the instructions–time diagram for the program execution and the state of the execution unit at each cycle. It also reports which instruction is in each stage of the processor pipeline. Each instruction is colored differently. Finally, it also shows the content of the registers and memory at the end of each cycle. Read and write operations are represented using a background color in the register or memory location targeted by the corresponding instruction. In this page there are links to the pages reporting the state of the processor 1, 5 and 10 cycles before as well as after the current cycle.

   The evolution of the program execution can be studied by navigating through these state files. They show which instructions are under execution during each cycle and the stalls inserted when hazards are detected.

   The archive with the final results(accessible by clicking on the link FINAL) provides performance results and it can be used to check the final content of registers and memory and to verify the correct execution of a program.

⇒ Check that the result vector of the program is stored in memory at the adress defined by label *z*.

⇒ Analyze the results provided by the simulator, which include all the executed instructions and the filling of the pipeline.

- Cycles= 347 .
- Instructions= 167 .
- CPI= 2.08 .

Next, we will obtain an estimate of the execution time considering only the loop instructions.

⇒ Consider to chronogram of the first loop iteration and answer the following questions:

- The contribution of first loop iteration to the execution time comprises from cycle 6 to cycle 26 .
- The number of clock cycles consumed by a loop iteration is of 21 cycles when the branch is taken.
- The total stall cycles are 11 cycles, of which the stall cycles due to data hazards are 8 cycles and the clock cycles due to control hazards are 3 cycles.
- The loop executes 10 instructions.
- The CPI reached is 2.1 .
- Execution time (cycles) ≈ Nr of loop iterations: 16 × Execution time one iteration (cycles) = 336 cycles.

2. Next, execute the program again, keeping the strategy for solving data hazards butthis time changing the technique for solving control hazards to *predict-not-taken* with 3 penalty cycles:

```
riscv-m -d p -c pnt3 -f apxpy.s
```

⇒ Analyze the results provided by the simulator, comparing them with those from the previous section.

- Cycles= 344 .
- Instructions= 167 .
- CPI= 2.06 .

The analysis of the execution cycle-wise of the first loop iteration shows how incorrectly fetched instructions are aborted after the branch since it is taken.

⇒ Consider to chronogram of the first loop iteration and answer the following questions:

- The number of clock cycles consumed by a loop iteration is of 21 cycles when the branch is taken.
- The total stall cycles are 9 (but penalty cycles are 11) cycles, of which the stall cycles due to data hazards are 8 cycles and the clock cycles due to control hazards are 1 cycles.
- The loop executes 10 instructions.
- The CPI reached is 2.1.
- Execution time (cycles) $\approx$ Nr of loop iterations: 16 $\times$ Execution time one iteration (cycles) = 336 cycles.

$\Rightarrow$ Consider the last loop iteration and answer the following question:

- The number of clock cycles spent by a loop iteration is of 18 when the branch is not taken.

3. Maintaining the strategy to tackle control hazard as *predict-not-taken*, modify the simulator configuration in order to solve the data hazards by using data forwarding (option c from the spanish translation of shortcircuit, which is *cortocircuito*):

   ```
   riscv-m -d c -c pnt3 -f apxpy.s
   ```

   ⇒ Analyze the results provided by the simulator, comparing them with those from the previous section.

   - Cycles= 248.
   - Instructions= 167.
   - CPI= 1.49.

   The cycle-wise performance analysis of the first loop iteration shows how the shortcircuits are used where nedeed.

   ⇒ Consider to chronogram of the first loop iteration and answer the following questions:

   - The number of clock cycles consumed by a loop iteration is of 15 cycles when the branch is taken.
   - The total stall cycles are 2 (but 5 penalty cycles) cycles, of which the stall cycles due to data hazards are 2 cycles and the clock cycles due to control hazards are 0 cycles.
   - The loop executes 20 instructions.
   - The CPI reached is 1.5.
   - Execution time (cycles) ≈ Nr of loop iterations: 16 × Execution time one iteration (cycles) = 240 cycles.

Modifications of the provided code

The goal of this second part of the lab session is to change the assembler code in order to reduce the number of stalls.

1. Copy the code to a second file (named apxpy-p3.s for instance), and modify it to reduce the penalty due to data hazards. For the execution of the simulator assume the use of *pnt3* and shortcircuits. Take into consideration that, when shortcircuits are used, only the load operations will insert stalls due to data hazards. Execute the program with the command line:

   ```
   riscv-m -d c -c pnt3 -f apxpy-p3.s
   ```

   ⇒ Check the correctness of the program result and verify that vector z contains the expected values.

   ⇒ Analyze the results provided by the simulator, comparing them with those from the previous section.

- Cycles= 216 .
- Instructions= 167 .
- CPI= 1.29 .

⇒ Consider to chronogram of the first loop iteration and answer the following questions:

- The number of clock cycles consumed by a loop iteration is of 13 cycles when the branch is taken.
- The total stall cycles are 0 cycles, of which the stall cycles due to data hazards are 0 cycles and the clock cycles due to control hazards are 0 cycles. (but 3 penalty cycles)
- The loop executes 10 instructions.
- The CPI reached is 1.3 .
- Execution time (cycles) ≈ Nr of loop iterations: 16 × Execution time one iteration (cycles) = 208 cycles.

2. Maintaining the use of shortcircuits for solving the data hazards, select now a *pnt1* to solve control hazards. In this case, (conditional) jump instructions may cause stall cycles for solving data hazards. Modify apxpy-p3 (rename it to apxpy-p1.s for instance) and modify it to reduce penalty due to data hazards.

   Execute the program using the command:

   ```
   riscv-m -d c -c pnt1 -f apxpy-p1.s
   ```

   ⇒ Check the correctness of the program result .

   ⇒ Analyze the results provided by the simulator, comparing them with those from the previous section.

   - Cycles= 186 .
   - Instructions= 167 .
   - CPI= 1.11 .

   ⇒ Consider to chronogram of the first loop iteration and answer the following questions:

   - The number of clock cycles consumed by a loop iteration is of 11 cycles when the branch is taken.
   - The total stall cycles are 0 cycles, of which the stall cycles due to data hazards are 0 cycles and the clock cycles due to control hazards are 0 cycles. (but 1 penalty cycle)
   - The loop executes 10 instructions.
   - The CPI reached is 1.1 .
   - Execution time (cycles) ≈ Nr of loop iterations: 16 × Execution time one iteration (cycles) = 176 cycles.

Development of a new program

The following high-level code counts the number of null components in a vector:

```
...
cont = 0;
for (i = 0; i < n; i++) {
  if (a[i] == 0) {
    cont = cont + 1;
  }
}
...
```

Below, we offer the skeleton of an assembler code to perform this task, included in the file `search.s`. The vector is stored at the memory address represented by the label `a` and its size is defined at the label `tam`. The number of null components of the vector will be stored at `cont`.

```
        .data
a:      .dword  9,8,0,1,0,5,3,1,2,0
tam:    .dword 10       # Vector size
cont:   .dword 0        # Number of components == 0

        .text
start:  addi t0,gp,a      # Pointer
        ld t1,tam(gp)     # Vector size
        dadd t2,zero,zero # Zero counter

loop:
        ...

        ori x17, x0, 10  # Program exit system call
        ecall
```

⇒ Answer the following questions:

1. Complete the provided code. Once the program is executed, memory address `cont` should contain the number of null components in the considered input vector.

2. Analyze the code and check its correctness with the simulator. For this, use short-circuits and *pnt1*:

   ```
   riscv-m -d c -c pnt1 -f search.s
   ```

   Analyze its execution time and CPI.

   - Cycles= 111 .
   - Instructions= 61 .
   - CPI= 1.82 .

3. Identify if there are any stall cycles and their causes. Modify the code to reduce the penalties.

   Analyze its execution time and CPI.

   - Cycles= 81.
   - Instructions= 61.
   - CPI= 1.33.