LAB 2a: "PIPELINED INSTRUCTION UNIT (I)"

Computer Architecture and Engineering (3^{rd} course)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Lab goals:

- Getting familiar with a simulator of a pipelined computer
- Analyzing the impact of data and control hazards on a pipelined instruction unit performance
- Writing programs in MIPS assembler

Lab work:

A simulator of the 5-stage MIPS computer

The simulator that will be used in this lab is able to simulate the execution of MIPS instructions cycle by cycle. It supports a subset of the MIPS instructions working on the integer register file. It has separate instructions and data caches (*Harvard* architecture). Registers are written and read in the first and second clock semicycle, respectively. Control hazards can be solved using several strategies, such as *stalls*), *predict-not-taken* and delayed load, with three possible sizes for the *delay-slot* (one, two and three, respectively). Data hazards can be solved by inserting stalls or by using the *forwarding* technique. It is worth mentioning that the simulated datapath changes according to the technique used for solving hazards.

The simulator can be executed from the command line:

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

where:

- <results> defines how the result will be obtained from the simulation. Several alternatives are available:
  - **tiempo**: Shows the execution time in the terminal.
  - **final**: Shows in the terminal the execution time, registers and memory content after the execution.
  - **html**(*): Generates several html archives showing both the state of the execution cycle by cycle and the final execution results. These results can be checked by opening the file **index.html** in a web browser.

    Adding the `-j` option generates a single `index.htm` file that includes all the results.

  - **html-final**: Generates the **final.html** html archive, which contains the final result of the exectuion.

- <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 nine alternatives:

  - **s3**: Control hazards are resolved by inserting three stalls.
  - **s2**: Control hazards are resolved by inserting two stalls.
  - **s1**: Control hazards are resolved by inserting one stall.
  - **pnt3**(*): Control hazards are solved by using the *predict-not-taken* technique and inserting three stalls whenever the branch is taken.
  - **pnt2**: Control hazards are solved by using the *predict-not-taken* technique and inserting two stalls whenever the branch is taken.
  - **pnt1**: Control hazards are solved by using the *predict-not-taken* technique and inserting one stall whenever the branch is taken.
  - **ds3**: Control hazards are solved by using the delayed branch technique, with a *delay-slot* of 3.
  - **ds2**: Control hazards are solved by using the delayed branch technique, with a *delay-slot* of 2.
  - **ds1**: Control hazards are solved by using the delayed branch technique, with a *delay-slot* of 1.

- <archive.s> is the name of the file holding the assembler code.

(*): Default option.

Example of MIPS program

Next, it is shown the assembler code of a loop carrying out 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:
    dadd r1,$gp,x
    dadd r4,r1,#128 ; 16*8
    dadd r2,$gp,y
    dadd r3,$gp,z
    ld r10,a($gp)

loop:
    ld r12,0(r1)
    dadd r12,r10,r12
    ld r14,0(r2)
    dadd r14,r12,r14
    sd r14,0(r3)
    dadd r1,r1,#8
    dadd r2,r2,#8
    dadd r3,r3,#8
    seq r5,r4,r1
    beqz r5,loop
    trap #0         ; End of the program
```

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

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

   Now, let's open the file **index.html** using a web browser. It will be shown the processor configuration, the memory content and a set of links that can be used to navigate through results:

   - INICIO. It shows the processor configuration and the initial memory state.
   - FINAL. It shows the performance results issued from the execution, the processor configuration and the final memory content.
   - Estado. It shows the instructions–time diagram of the program execution and the state of the execution unit in each cycle. It also reports which instruction is in each stage of the processor pipeline. Each instructions is represented in a different color. 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 and 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 final results archive provides performance results and it can be used to check the final content of registers and memory and verify the correct execution of programs. In this case, the result vector must be stored in the memory location defined by the *z* label.

Regarding the performance results, to do the analysis, consider the first loop iteration and answer the following questions:

- The contribution of the first iteration to the execution time comprises from the cycle _____ to the cycle _____.
- The number of clock cycles consumed by one iteration of the loop is _____ cycles if the jump is taken and _____ cycles if the jump is not taken.
- The total stall cycles are _____ cycles, of which the stall cycles due to data hazards are _____ and the stall cycles due to control hazards are _____ cycles.
- The loop executes _____ instructions.
- The CPI reached is: _____.

2. Then, execute again the program changing the technique for solving control hazards to *predict-not-taken* with 3 penalty cycles:

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

The analysis of the execution cycle by cycle of the first loop iteration shows how incorrectly fetched instructions are aborted after the branch, since it is taken. Answer the following questions:

- The contribution of the first iteration to the execution time comprises from the cycle _____ to the cycle _____.
- The number of clock cycles consumed by one iteration of the loop is _____ cycles if the jump is taken and _____ cycles if the jump is not taken.
- The total stall cycles are _____ cycles, of which the stall cycles due to data hazards are _____ and the stall cycles due to control hazards are _____ cycles.
- The loop executes _____ instructions.
- The CPI reached is: _____.

3. Keeping the control hazard strategy to *predict-not-taken*, modify the simulator configuration in order to solve data hazards by using shortcircuits:

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

   The cycle by cycle performance analysis of the first loop iteration shows how shortcircuits are used whenever necessary. Answer the following question:

   - The contribution of the first iteration to the execution time comprises from the cycle _____ to the cycle _____.
   - The number of clock cycles consumed by one iteration of the loop is _____ cycles if the jump is taken and _____ cycles if the jump is not taken.
   - The total stall cycles are _____ cycles, of which the stall cycles due to data hazards are _____ and the stall cycles due to control hazards are _____ cycles.
   - The loop executes _____ instructions.
   - The CPI reached is: _____.

Code modifications.

The goal of this part of the lab is to change the assembler code in order to reduce as much as possible the number of stalls.

1. Copy the code to another file (apxpy-p3.s for instance) and modify it to reduce the data hazard penalty. Assume the use of *pnt3* and shortcircuits. It must be taken into consideration that when shortcircuits are used, only loads insert stalls to solve data hazards. Execute the program with the command line:

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

   Take a look at the final results and check that the result is correct. In particular, confirm that the vector $\vec{Z}$ stores the expected values.

   Consider the first loop iteration and answer the following questions:

   - The contribution of the first iteration to the execution time comprises from the cycle _____ to the cycle _____.
   - The number of clock cycles consumed by one iteration of the loop is _____ cycles if the jump is taken and _____ cycles if the jump is not taken.
   - The total stall cycles are _____ cycles, of which the stall cycles due to data hazards are _____ and the stall cycles due to control hazards are _____ cycles.
   - The loop executes _____ instructions.
   - The CPI reached is: _____.

2. Keeping the use of shortcircuits for solving data hazards, select now a *pnt1* in order to solve control hazards. In this case, the jump instructions could cause stall cycles to solve data hazards. Modify apxpy-p3 (rename it to apxpy-p1.s for instance) and modify it to reduce data hazard penalties.

   Execute the program using the command:

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

   Take a look at the final results and check that the result is correct.

   Consider the first loop iteration and answer the following questions:

   - The contribution of the first iteration to the execution time comprises from the cycle _____ to the cycle _____.
   - The number of clock cycles consumed by one iteration of the loop is _____ cycles if the jump is taken and _____ cycles if the jump is not taken.
   - The total stall cycles are _____ cycles, of which the stall cycles due to data hazards are _____ and the stall cycles due to control hazards are _____ cycles.
   - The loop executes _____ instructions.
   - The CPI reached is: _____.

3. Keeping shortcircuits to solve data hazards, now choose *ds1 (delay slot 1)* to solve control hazards. Modify apxpy-p1.s (for instance, rename it to apxpy-d1.s) to execute correctly, trying to fill the delay slot with useful instructions.

   Execute the program using the command:

   ```
   mips-m -d c -c ds1 -f apxpy-d1.s
   ```

   Take a look at the final results and check that the result is correct.

   Consider the first loop iteration and answer the following questions:

   - The contribution of the first iteration to the execution time comprises from the cycle _____ to the cycle _____.
   - The number of clock cycles consumed by one iteration of the loop is _____ cycles if the jump is taken and _____ cycles if the jump is not taken.
   - The total stall cycles are _____ cycles, of which the stall cycles due to data hazards are _____ and the stall cycles due to control hazards are _____ cycles.
   - The loop executes _____ instructions.
   - The CPI reached is: _____.

Developing 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, it is shown part of the assembler code to perform this task, included in the file `search.s`. The vector is stored at the memory addresss represented by the label `a`, its size is defined at the label `tam`, and the number of null components will be stored at `cont`.

```
        .data
a:      .dword  9,8,0,1,0,5,3,1,2,0
tam:    .dword 10       ; Tamaño del vector
cont:   .dword 0        ; Número de componentes == 0

        .text
start:  dadd r1,$gp,a   ; Puntero
        ld r4,tam($gp)  ; Tamaño lista
        dadd r10,r0,r0  ; Contador de ceros

loop:
        ...

        trap #0
```

1. Complete the code. The number of null componets must be stored at `cont` when the program is finished.

2. Analyze the code and check that it is correct by executing it in the simulator. We will use shortcircuits and *pnt1*:

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

   Analyze its execution time and CPI.

3. Identify if there are stall cycles and their causes. Then, modify the code to reduce such penalty.

   Analyze its execution time and CPI.

Subset of MIPS instructions supported by the simulator

- Load/Store

ld Rx, desp(Ry)
sd Rz, desp(Ry)

- Arithmetic, logic and shift

dadd Rx, Ry, Rz	daddi Rx, Ry, Imm
dsub Rx, Ry, Rz	dsubi Rx, Ry, Imm
and Rx, Ry, Rz	andi Rx, Ry, Imm
or Rx, Ry, Rz	ori Rx, Ry, Imm
xor Rx, Ry, Rz	xori Rx, Ry, Imm
dsra Rx, Ry, Rz	dsra Rx, Ry, Imm
dsll Rx, Ry, Rz	dsll Rx, Ry, Imm
dsrl Rx, Ry, Rz	dsrl Rx, Ry, Imm

- Comparison

seq Rx, Ry, Rz	seq Rx, Ry, Imm
sne Rx, Ry, Rz	sne Rx, Ry, Imm
sgt Rx, Ry, Rz	sgt Rx, Ry, Imm
slt Rx, Ry, Rz	slt Rx, Ry, Imm
sge Rx, Ry, Rz	sge Rx, Ry, Imm
sle Rx, Ry, Rz	sle Rx, Ry, Imm

- Conditional branch

bnez Ry, Desp
beqz Ry, Desp

- Other

nop
trap

Anex A

More common errors are:

- The file has been edited in Windows, thus including carriage returns '\r' which can be deleted using the command tr -d "\r"$< original\_file > file\_without\_r$
- .data is missing in the code
- A label is duplicated.
- In order to add *dword* integers, the adequate instruction is *dadd* (but the one used is *add*, which is incorrect in this case).
- In order to substract *dword* integers, the adequate instruction is *dsub* (but the one used is *sub*, which is incorrect in this case).