Lab. 3:
"PIPELINED INTRUCTION UNIT (II)"

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

Goals:

- Implement the logic to solve data and control hazards in a pipelined processor.

Development:

This lab uses a RISC V simulator able to interpret RISC V assembler code, including both integer and floating-poiint instructions. However, programs considered in the lab will only make use of integer instructions. These instructions simulate in a pipelined instructions unit integrating 5-stage (IF, ID, EX, MEM y WB) datapath. The code required to solve data and control hazards is not included in the version of the simulator available and its development defines the goal of this lab.

The lab assignment is structured as follows. It first describes the files included in the simulator. Then, it defines the implemented instructions, data structures under use, and it specifies the internal structure of the RISC V simulator pipeline. Finally, it proposes various exercises.

Structure of the simulator

The RISC V simulator is written in C. It includes various modules, being the following ones the most important for this lab:

main.c Main simulator program. It is in charge of reading the assembler and executing the different stages of the pipelined instruction unit.

main.h It contains all simulator shared variables: instruction and data memory, general purpose registers, inter-stage registers, control signals, etc.

tipos.h It contains definitions of all data instructions used in the simulator: instruction and data cache, register files, inter-stage registers, instruction formats, etc.

instrucciones.h It contains the code of supported instructions and some utility macros.

riscv.c It contains the implementation of all the stages of the instruction unit. *This file will be modified along this lab.*

riscv_int.c It contains the implementation of the ALU, the logic required for data hazard detection and the activation of short-circuits for integer instructions. *This file will be modified along this lab.*

Implemented instructions

The simulator supports the basic integer instruction set (rv64i) of the RISC V ISA specification.

Data structures

The following paragraphs describe several simulator data structures (defined at file `tipos.h`) and their use.

Basic types

The basic types are:

```
typedef int8_t          byte;   /* One byte: 8 bits */
typedef int16_t         half;   /* Half-word : 16 bits */
typedef int32_t         word;   /* One word: 32 bits */
typedef int64_t         dword;  /* A double word: 64 bits */
typedef enum {NO=0, SI=1} boolean; /* Logic value */
```

Instruction format

Instruction formats are represented with an enumerated type:

```
/* Instruction formats */
typedef enum {FormatoR, FormatoI, FormatoS,
              FormatoB, FormatoU, FormatoJ} formato_t;
```

Instructions are represented with the following data structure:

```
typedef struct {
  codop_t       codop;  /* Operation code */
  formato_t     tipo;   /* Format */
  byte          rs1,    /* Source register  1 */
                rs2,    /* Source register  2 */
                rs3;    /* Source register  3 */
  byte          rd;     /* Destination register */
  half          imm;    /* Immediate Value */
} instruccion_t;
```

Register file

The register file is a vector including elements of type `reg_int_t`, with a single field, named *valor*.

```
typedef struct {
  valor_t       valor;    /* Register value */
} reg_int_t;
```

Inter-stage registers

The inter-stage registers are represented with an struct containing the following fields:

- IF/ID register:

```
typedef struct {
  instruccion_t  IR;            /* IR */
  dword          PC;            /* PC */
} IF_ID_t;
```

- ID/EX Register:

```
typedef struct {
  instruccion_t  IR;            /* IR */
  dword          PC;            /* PC */
  dword          Ra,            /* Registers' value*/
                 Rb;
  dword          Imm;           /* Immediate value with extended sign */
} ID_EX_t;
```

- EX/MEM Register:

```
typedef struct {
  instruccion_t  IR;             /* IR */
  dword          ALUout;         /* ALU Result */
  dword          data;           /* Data to be written */
  boolean        cond;           /* Result defining a branch condition */
} EX_MEM_t;
```

- MEM/WB Register:

```
typedef struct {
  instruccion_t  IR;             /* IR */
  dword          ALUout;         /* ALU Result */
  dword          MEMout;         /* Memory Result */

} MEM_WB_t;
```

Other data structures

- The following structure defines how the simulator solves data hazards:

```
typedef enum {
  parada,               /* stalls */
  cortocircuito,        /* forwarding + stalls */
  ninguno
} riesgos_d_t;
```

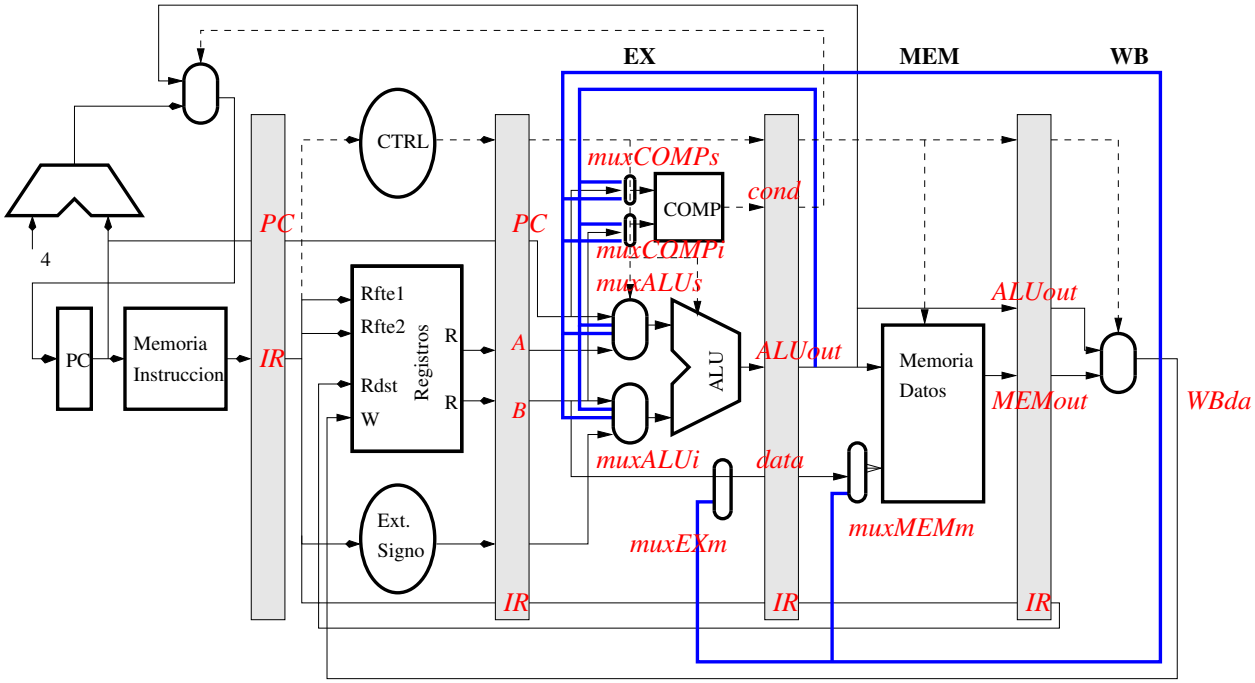

Figura 1: Pipelined RISC-V processor.

- The following structure defines how the simulator solves control hazards:

```
typedef enum {
  stall3,                /* Insert 3 stalls */
  stall2,                /* Insert 2 stalls */
  stall1,                /* Insert 1 stall  */
  pnt3,                  /* Predict-not-taken, 3 cycles */
  pnt2,                  /* Predict-not-taken, 2 cycles */
  pnt1                   /* Predict-not-taken, 1 cycle  */
} riesgos_c_t;
```

Structure of a pipelined instuction unit

The pipelined instruction unit integrates the following elements (see figure 1):

Instructions memory. Stores the program to be executed. It is a byte addressable memory.[1] The variable `MI` (`main.h`), of type `memoria_instruc_t` (`tipos.h`), represents this memory. The memory size is defined by constant `TAM_MEMO_INSTRUC` (`tipos.h`).

Integer register file. It contains integer registers. It is represented by variable `Rint` (`main.h`), of type `reg_int_t []` (`tipos.h`). The number of registers is defined by the constant `TAM_REGISTROS` (`main.h`).

[1]Successive bytes are assigned to consecutive memory addresses and succesive words are assigned to memory addresses differing in 4.

Arithmetic operator. It carries out arithmetic operations during EX. It is represented by the function `operacion_ALU (codop, in1, in2)` (`riscv_int.c`), where `codop`, `in1` and `in2` denote the operation to be carried out and the two operands to use, respectively. The function returns the result of the corresponding operation.

Evaluation of branch conditions. Evalutes a branch condition. It is represented by function textttoperacion_COMP (codop,i n1, in2) (`riscv_int.c`), where `codop`, `in1` and `in2` refer to the branch instruction to execute and the two operands to check, respectively. This function returns whether the branch must be taken or not.

Multiplexor of the upper input of the arithmetic operator. Represented by the function `mux_ALUsup (pc,ra,mem,wb)` (`riscv_int.c`), where `pc,ra,mem` and `wb` represent the inputs to the multiplexor. This function returns the selected input.

Multiplexor of the lower input of the arithmetic operator. Represented by the function `mux_ALUinf (rb,imm,mem,wb)` (`riscv_int.c`), where `rb,imm,mem` and `wb` represent the inputs to the multiplexor. This function returns the selected input.

Multiplexor of the upper input of the comparator (branches). Represented by the function `mux_COMPsup (ra,mem,wb)` (`riscv_int.c`), where `ra,mem` and `wb` represent the inputs to the comparator. The function returns one of the inputs, as applicable.

Multiplexor of the lower input of the comparator (branches). Represented by the function `mux_COMPinf (rb,mem,wb)` (`riscv_int.c`), where `rb,mem` and `wb` represent the inputs to the comparator. The function returns one of the inputs, as applicable.

Data multiplexor to be written in memory (EX). Represented by the function `mux_EXmem (rb,wb)` (`riscv_int.c`), where `rb` and `wb` represent the multiplexor inputs. The function returns the selected input, as applicable.

Data memory. It stores the data to be used by the program. It is a byte addressable memory. It is represented by variable `MD`, (`main.h`) of type `memoria_datos_t` (`tipos.h`). The memory size is provided by constant `TAM_MEMO_DATOS` (`tipos.h`).

Multiplexor of data to be written in memory (MEM). Represented by the function `mux_MEMmem (rb,wb)` (`riscv_int.c`), where `rb` and `wb` represent the multiplexor inputs. The function returns one of the inputs, as applicable.

Output of the multiplexor at WB. Represents the data to be written in the register file during WB. It is represented by variable `WBdata` (`main.h`), of type `dword`.

Inter–stage registers. Their names specify the stages they interconnect:

- `IF_ID`, of type `IF_ID_t` (`tipos.h`).
- `ID_EX`, of type `ID_EX_t` (`tipos.h`).
- `EX_MEM`, of type `EX_MEM_t` (`tipos.h`).
- `MEM_WB`, of type `MEM_WB_t` (`tipos.h`).

For instance, the access to `IF_ID.IR.rs1` enables the program to know the register source1 used by an instruction located in statge ID, which is the stage fed by the inter–register `IF_ID`.

Current PC and new PC value Represented by the variables `PC` and `PCn`, of type `dword`. The following instruction will be fetched from the location in `PCn`.

Control signals The simulator has the following control signals, all of them are `booleans`:

- `IFstall`: When it becomes active (`IFstall=SI`), it stalls the instruction in the IF stage during the following clock cycle. It also sends a `nop` to the ID stage.
- `IDstall`: When it becomes active, it stalls the instruction in the ID stage during the following clock cycle. It also sends a `nop` to the EX stage.
- `IFnop`: When it becomes active, it will send the next cycle a `nop` instruction to the ID stage.
- `IDnop`: When it becomes active, it will send the next cycle a `nop` instruction to the EX stage.
- `EXnop`: When it becomes active , it will send the next cycle a `nop` instruction, to the MEM stage.

Instruction check The following macros or functions can be invoked to check the content of certain instruction fields:

- `es_load(inst)` queries whether instruction `inst` is a load.
- `rs1_valido(inst)` queries whether `inst` uses a valid source register 1.
- `rs2_valido(inst)(inst)` queries whether `inst` uses a valid source register 2.
- `rd_valido(inst` queries whether `inst` uses a valid target register.

For example, the implementation of one of them is as follows:

```
boolean rs1_valido(instruccion_t inst) {
    // Simplified code
    // For arithmetic, logic, comparison, branch instr
    // There is Rfte1 if it is not x0
    return (inst.rs1!=0);
```

For instance:

- To know whether instruciton in ID is goint to read the source register field1, use `rs1_valido(IF_ID.IR)`.
- To know if the instruction in EX is going to write the target register field, use `rd_valido(ID_EX.IR)`.
- To know if the instruction in the ID stage is a load, use `es_load(IF_ID.IR)`.

Example of control logic. The following sequence of instructions requires two stalls to solve the existing data hazard:

```
add x1,x2,x3  IF ID EX ME WB
sub x4,x1,x5     IF id id ID EX ME WB
```

For the first stall, the control logic must check that:

- The destination register of the instruction that is in EX (inter–stage pipeline register ID_EX) equals the source1 register of the instruction that is in ID (inter–stage pipeline register IF_ID).
- The instruction in EX produces a result (in its destination register).
- The instruction in ID consumes a result (in its source1 register).

and using the processor data structures, macros and control signals:

```
if ( rd_valido(ID_EX.IR) &&
     ( (rs1_valido(IF_ID.IR)) && (ID_EX.IR.rd == IF_ID.IR.rs1) )
   )
{
    IDstall = SI;
    IFstall = SI;
}
```

RISC V simulator pseudo-code

Once the data structures has been initialized, the simulator loads the file containing the program to execute. Then, the simulator assembles and executes the program by running the simulator main loop, whose pseudo-code looks as follows:

```
/* Main loop of the RISCV simulator */

  /*** Fase: WB **************/
  writing_stage(): [fase_escritura ()]
    - write into the register

  /*** Fase: MEM *************/
  memory_stage(): [ fase_memoria ()]
    - control hazard detection
    - use of shortcirtuits
    - memory access, if necessary

  /*** Fase: EX *************/
  execution_stage(): [fase_ejecucion ()]
    - control hazard detection
    - use of shortcirtuits
    - operation in ALU/COMP
```

```
      /*** Fase: ID **************/
      decoding_stage(): [fase_decodificacion()]
        - data hazard detection
        - control hazard detection
        - register reading

      /*** Fase: IF ***********/
      fetching_stage(): [fase_busqueda()]
        - instruction fetch
        - PC updating

      cycle++;
      print_state;
      next_clock_cycle(); [impulso_reloj()]
```

Exercises

First of all, check how the simulator works using a simple fragment of code without data dependencies. Use the file `ejemplo.s`:

```
        # adds the components of vector y until it finds
        # a component equal to 0
        # stores the result in a

        # the result must be a=6

        .data
a:      .dword  0
y:      .dword  1,2,3,0,4,5,6,7,8
        .text
        add t1,x0,x0  # t1=0
        add t3,x0,x0  # t3=0
        addi t2,gp,y  # t2 traverses y
        nop
loop:   add t1,t3,t1
        ld t3,0(t2)   # t3 is y[i]
        addi t2,t2,8
        nop
        bnez t3,loop  # if t3<>0
        sd t1,a(gp)
end:    ori a7,x0,10
        ecall
```

To execute the simulator, use the command `riscv-m`. Remember that the simulator accepts several parameters. Check the assignment of lab 2 to see which are those parameters. An alternative approach is to execute the command `riscv-m -?`.

In this case, the simulator will run withouth logic for detecting data hazards and solving control hazards by inserting 3 stalls:

```
riscv-m -d n -c s3 -f ejemplo.s
```

This command will generate an **html** file for each cycle with all the information related to the state of the machine, plus an initial file `index.html`. These files can be visualized using any web browser (such as `firefox or konqueror`). By default, the simulator deletes these html files before each new simulation, except when the "-n" parameter is specified.

⇒ Check the correct behavior of the simulator, and the expected result of the execution.

The simulator only includes the logic (code) that is necessary to solve the control hazards by inserting three stalls. However, **it is not able to detect or solve data hazards**.

In this lab session the simulator must be enhanced with some new strategies to handle data hazards. To do so, the files `riscv.c` and `riscv_int.c` need to be modified by adding the necessary logic to perform the necessary actions.

In order to edit the files you can use any one of the available editors, such as `vi`, `emacs`, `[gk]edit` or `kate`.

After each modification of its code, the simulator must be recompiled. The compilation of the `riscv-m` simulator must be performed by executing the command `make` in the directory where simulator sources are located:

```
make
```

In order to check the correct behavior of applied modifications, use the testing programs provided. Follow the process described hereafter.

1. Modify the RISC V simulator to detect and solve the data hazards by inserting stalls. More precisely, the data hazards must be solved for the following sequence of instructions (saved in the file `datos1.s`):

```
        # the result must be t3=30, t4=20 y t5=40

        .ireg t1=10,t2=20
        .text
        add t3,t1,t2
        addi t4,t3,-10
        add t5,t1,t3
end:    ori a7,x0,10
        ecall
```

   ⇒ Draw the instructions–time diagram related to the execution of the provided sequence of code. Insert stall wherever necessary.

   ⇒ Taking as reference the control logic shown in page 7, modify the function detecting data hazards during instruction decoding (function `detectar_riesgos_datos` in file `riscv_int.c`). Write in that function the required code to activate the adequate control signals (`IFstall`, `IDstall`).

   Check the modifications by executing:

```
riscv-m -d p -c s3 -f datos1.s
```

2. Modify the RISC V simulator in order to detect and solve the data hazards through short-circuits.

   a) First, solve the data hazards in the aforementioned sequence of code (file `datos1.s`). Such sequence of instructions does not require the insertion of stalls.

   ⇒ Draw the instructions–time diagram related to the execution of the provided sequence of code. Insert short-circuits wherever necessary.

   ⇒ Modify the function implementing the upper (operand fuente1) and lower (operand fuente2) multiplexors located at the input of the ALU operator. These functions are `mux_ALUsup` and `mux_ALUinf` and can be found in file `riscv_int.c`.

   ⇒ Once the modified simulator has been successfully compiled, check its correct behavior by executing the command:

   ```
   riscv-m -d c -c s3 -f datos1.s
   ```

   b) Now solve data hazards provoked by **a load followed by an arithmetic instruction**. The code is the one saved in file `datos2.s`:

   ```
            # the result must be t3=30, t4=20 y t5=40

            .ireg 0,0,0,0,0,10,20 # t1=10, t2=20
            .data
   a:       .dword 30
            .text
            ld t3,a(gp)
            addi t4,t3,-10
            add t5,t1,t3
   end:     ori a7,x0,10
            ecall
   ```

   In this case, in addition to the activation of the corresponding short-circuit, the insertion of a stall in ID is necessary.

   ⇒ Draw the instructions–time diagram related to the execution of the provided sequence of code. Insert stalls and short-circuits wherever necessary.

   ⇒ As the diagram will show, in addition to the modification already carried out in the function implementing the multiplexor (function `mux_ALUsup` in `riscv_int.c` as requected in section 2*a*), the function detecting data hazards in ID must be also modified (function `detectar_riesgos_datos` in `riscv_int.c`).

   ⇒ Once the modified simulator has been successfully compiled, check its correct behavior by executing the command:

   ```
   riscv-m -d c -c s3 -f datos2.s
   ```

3. Modify the RISC V simulator in order to solve control hazards using *predict-not-taken*.

⇒ To do so modify the IF stage (function `fase_búsqueda` in `riscv.c`). Look for inspiration at the code implementing the insertion of stalls (stall3).

In order to check this modification, use the code in file `suma.s`.

```
        # adds the components of vector y until it finds
        # a component equal to 0
        # stores the result in a

        # the result must be a=6

        .data
a:      .dword  0
y:      .dword  1,2,3,0,4,5,6,7,8
        .text
        add t1,x0,x0  # t1=0
        add t3,x0,x0  # t3=0
        addi t2,gp,y  # t2 traverses y
loop:   add t1,t3,t1
        ld t3,0(t2)   # t3 is y[i]
        addi t2,t2,8
        bnez t3,loop  # if t3<>0
        sd t1,a(gp)
end:    ori a7,x0,10
        ecall
```

⇒ Check your code using the following command, where data hazards are solved using stalls:

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

Observe now that the instructions are cancelled only when the branch is taken. Verify the memory position storing `a` and check it holds a correct value.