LAB 5:
"TOMASULO'S ALGORITHM: IMPLEMENTATION"

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

Goals:

- Implement and evaluate data dependency management in the *Issue*, *Writeback*, and *Commit* stages of the dynamic instruction scheduling algorithm known as Tomasulo's Algorithm.

Development:

For the development of this lab, a RISC-V simulator (RISC-V/OOO) will be used as a starting point. This simulator is capable of handling dynamic instruction scheduling by applying Tomasulo's algorithm. The simulator takes an assembly language file as input and still requires the implementation of parts of the *Issue*, *Writeback*, and *Commit* stages of the Tomasulo's algorithm.

This bulletin is organized as follows: pseudocode of Tomasulo's algorithm, exercises to be completed, and appendices explaining the structure of the simulator, the data structures used, and the structure of the dynamic instruction management unit.

Tomasulo's Algorithm Simulator

In the appendices from A to C, at the end of this document, you will find all the details about the data structures and variables required for the implementation to be carried out in this assignment.

In this session, only the C files whose names follow the structure `FICHERO_alum.c` will be modified, and only in those sections where the following comment appears:

```
/* INSERTAR CÓDIGO */
```

The following section lists the instructions implemented by the simulator and provides the pseudocode for the stages of Tomasulo's algorithm to assist the student in the implementation.

Implemented instructions

The simulator implements the RISC-V architecture instructions corresponding to the `RV64IMFD` modules, that is, the 64-bit integer instructions, integer multiplication and division, and support for single- and double-precision floating-point operations

Description of Tomasulo's Algorithm

The following is a description of Tomasulo's algorithm for the *Issue*, *Execution*, *Write-back*, and *Commit* stages.

- *Issue*

```
// Data from decoding:
-ALU: I_OP, I_D, I_S1, I_S2 ; OPCODE, Dest reg, Src reg 1, Src reg 2
-LOAD: I_OP, I_D, I_S1, I_INM ; OPCODE, Dest reg. Src reg 1, Imm
-STORE: I_OP, I_S1, I_S2, I_INM ; OPCODE, Src reg 1, Src reg 2, Imm
-BRANCH: I_OP, I_S1, I_S2, dir, pred ; OPCODE, Src reg 1, Src reg 2, Addr, Pred

If {RS/LB/SB available} and {entry in the RB available} then
    // RS/LB/SB
    Reserve RS/LB/SB
    Write I_OP in RS/LB/SB
    Write RB entry in RS/LB

    // Operand 1
    If {I_OP is ALU, LOAD, STORE or BRANCH} then
        If {mark of register I_S1 is NULA} then
            Copy value from register into RS/LB/SB
        Else
            If {future value of register I_S1 is in the RB} then
                Copy value from RB into RS/LB/SB
            Else
                Copy mark (entry in RB) into RS/LB/SB

    // Operand 2
    If {I_OP is ALU, STORE or BRANCH} then
        If {mark of register I_S2 is NULA} then
            Copy value from register into RS/SB
        Else
            If {future value of register I_S2 is in the RB} then
                Copy value from RB into RS/SB
            Else
                Copy mark (entry in RB) into RS/SB

    // Displacement, if applicable: LOAD and STORE
    If {instr is LOAD or STORE} then
        Write displacement into LB/SB

    // Initialize confirmation in case of STORE
    If {instr is STORE} then
        Mark as not confirmed in SB

    // Reorder Buffer
    Reserve RB
    Mark as not completed in RB
    Write I_OP in RB
    If {I_OP is ALU or LOAD} then
        Write register I_D in RB
    If {I_OP is STORE} then
        Write SB entry in RB
    If {I_OP is BRANCH} then
        Write dir
        Write pred

    // Reserve the destination register
    If {I_OP is ALU or LOAD} then
        Write RB entry in register I_D
```

- *Execution*

```
For each {operator} do
  If {there is RS/LB/SB with operands ready} then
    Select_one()
    Operation():
        -ALU: Operation in the ALU
        -BRANCH: Branch condition calculation
        -LOAD/STORE: Address calculation
        -LOAD/STORE: Memory read / Memory write if SB confirmed
    Release_Operator()
```

- *Writeback*

```
If {there is an RS/LB with available results} then

// Dump results to the bus
Dump result on the bus (value)
Dump RB entry on the bus (code)

// If ALU or LOAD
Release the reservation station (RS or LB)

// Reorder Buffer
Write result in RB
Mark as completed in RB

// Read results
For each {RS/LB/SB} do
    // Operand 1
    If {mark of operand 1 from RS/LB/SB
        matches the code present on the bus}
        Copy bus value into RS/LB/SB
        Remove mark

    // Operand 2
    If {mark of operand 2 from RS/SB
        matches the code present on the bus}
        Copy bus value into RS/SB
        Remove mark
```

- *Commit*

```
If {instruction at the head of the RB has finished} then

    If {instruction is BRANCH} and {incorrect prediction}
        Release registers
        Remove RS, LB y SB, except confirmed SB
        Release operators (except memory if writing)
        Write the correct address in the program counter
        Remove RB

    Else If {instruction is STORE}
        Confirm write

    Else // Write to register file
        Write destination register
        If {no other later instruction writes to this register}
            Release register

    Release entry in the RB
```

Implementation of Tomasulo's Algorithm

After familiarizing yourself with the simulator structure (Appendix A) and the data structures (Appendices B and C), we will proceed with the implementation of data dependency management throughout the *Issue*, *Writeback*, and *Commit* stages.

To test the implementation of each phase, we will use the example program `basico.s`, which is very similar to the one presented in theory when explaining the dependency graph.

```
    # Initialization of FP registers
    .fpreg f1=1.0, f2=2.0

    .data
a:  .double 2.0

    .text
    fadd.d f4, f2, f1
    fmul.d f6, f4, f1
    fld f4, a(gp)
    fmul.d f6, f6, f4
final:
    li a7, 10               # syscall exit
    ecall
```

After finishing the implementation of each stage, you must compile using the `make` command in the directory where the source files are located. Once there are no **compilation errors**, the simulator should be tested with the command:

```
$> ./riscv-ooo -t basico.sign -f basico.s
```

Note: Remember that the executable file `riscv-ooo` **will not** exist if compilation errors have occurred.

Preliminary test

Before starting the implementation, we will proceed to compile the simulator (incomplete) with the aforementioned `make` command.

The simulator includes a system for detecting execution errors. In the previous command, the file `basico.sign` contains the summary of processor states corresponding to the correct execution of the file `basico.s`. If there is any difference with this file, the simulator will indicate the cycle in which the error occurred. If we access the state of the data path (by opening the file `estadoNNN.html`) corresponding to that cycle (`NNN`), we can observe (*in red and italics*) which fields are incorrect. If any tags are missing, the symbols "*??*" are shown.

Thus, if we run the simulator before beginning the implementation using the previous command, it will indicate that there are errors in the execution (in cycle 2) and that the details can be found in the file `estado002.html`.

```
ERROR. Existen diferencias entre la ejecución actual y el fichero de firmas.
Analice el fichero de estado 'estado002.html'
%ERROR. There are differences between the current execution and the
%signature file. Analyze the state file 'estado002.html'
```

Cycle 2 corresponds to the *Issue* stage of the instruction `fadd.d f4, f2, f1`, which has not yet been implemented. In that state file, we can observe how it indicates that neither the operands nor the destination register have been properly processed.

Implementation of the algorithm stages

We will address the errors encountered by proceeding with the implementation.

1. Implementation of the **Issue** stage
   This can be found in the function `fase_ISS_alum()` in the file `f_lanzamiento_alum.c`. You must implement the dependency management for operands 1 and 2, as well as the reservation of the destination register, for floating-point arithmetic instructions. You can find the description of Tomasulo's algorithm on page 2.

   Note: After running the simulator, it will notify you that an error has occurred in cycle 7, which corresponds to the moment when the instruction `fadd.d f4, f2, f1` attempts to do WB, since we have not implemented it yet.

   To verify that the Issue stage has been implemented correctly, check the following:

   a) Make sure that the first instruction `fadd.d f4, f2, f1` correctly reads operands `f2` and `f1`, since there are no pending writes to them, and reserves the destination register `f4` with the identifier of the entry it occupies in the ROB.

   b) Make sure that the second instruction `fmul.d f6, f4, f1` correctly reads the second operand `f1` and is chained to the first operation when attempting to read the first operand `f4`. Also check that the destination register `f6` is reserved with the identifier of the entry it occupies in the ROB.

   c) Finally, verify that the instruction `fmul.d f6, f6, f4` is chained with the instructions `fmul.d f6, ...` and `fld f4 ...`. Additionally, check that this instruction has overwritten the existing tag on register `f6`.

2. Implementation of the **Writeback** stage
   If you look at the file `estado007.html` from the previous run, you will see that it indicates that the multiplication operation from station `m1` has not received the expected data. Let us now complete the *WB* stage.

   The current implementation can be found in the function `fase_WB_alum()` in the file `f_transferencia_alum.c`. The transfer of results from the bus to the reservation stations must be completed, i.e., the resolution of dependencies for operands 1 and 2.

   After implementing and running the simulator again, you should verify that the instruction `fmul.d f6, f4, f1` correctly receives the missing operand in cycle 7. However, in cycle 8 the instruction `fadd.d f4, f2, f1` attempts to do *Commit* (not yet implemented) and the simulator will once again indicate an error.

3. Implementation of the **Commit** stage
   In the previous run, the instruction `fadd.d f4, f2, f1` did not correctly complete the *Commit* stage, since the register write for this phase has not yet been implemented.

The incomplete implementation can be found in the function `fase_COM_alum()` in the file `f_confirmacion_alum.c`. You must implement the write to the floating-point register file for the destination register, and perform the release as appropriate.

After finishing the implementation of all stages, the simulator should terminate without errors (in the cycle when the instruction `ecall` reaches *Commit*). Check in the final state (file `final.html` or `FINAL` link in the browser) that the values of registers `f4` and `f6` are correct.

Verifying the operation of Tomasulo's Algorithm

Once Tomasulo's algorithm has been implemented and compiled, its operation will be verified using the following examples:

1. Code contained in the file `ejemplo.s`.

```
    .data                   # Start of memory data segment
a:  .double 10.5
b:  .double 2
c:  .double 20

x:  .space 8
y:  .space 8

    .text                   # Start of code segment
inicio:
    fld ft0, a(gp)          # Load a
    fld ft1, b(gp)          # Load b
    fld ft2, c(gp)          # Load c
    fadd.d ft4, ft0, ft1    # t1 = a + b
    fmul.d ft5, ft2, ft4    # t2 = c * t1
    fsd ft4, x(gp)          # Save t1
    fsd ft5, y(gp)          # Save t2
final:
    li a7, 10               # syscall exit
    ecall
```

To invoke the execution of the simulator, use the syntax in the following example:

```
$> ./riscv-ooo -t ejemplo.sign -f ejemplo.s
```

This command will generate an **html** file for each cycle containing information about the state of the machine, which can be viewed using a web browser.

Its proper functionality—both logical and temporal—must be verified. For this, the latencies of each operator must be considered (by default, 2 cycles for load/store, 4 cycles for add/subtract, and 7 cycles for multiply/divide).

Observe how store instructions are managed. In cycles 6 and 7 (*Issue* stages of the `fsd`) the index of the corresponding write buffer is recorded in the `dest` field. This index is used in cycles 23 and 24 (*Commit* stages) to confirm memory writes in the

write buffer, thereby authorizing the memory operator to proceed with the operation. The memory write process concludes in cycles 25 and 27, respectively.

Check the changes made in the write buffers and memory during the specified cycles.

2. Verify the operation of the DAXPY loop ($a\vec{x} + \vec{y}$). The file `daxpy.s` contains the assembly code.

   Its correct operation should be checked using the initial configuration of the operators. The summary file used in this case will be `daxpy.sign`:

   ```
   $> ./riscv-ooo -t daxpy.sign -f daxpy.s
   ```

   Observe how branch instructions are managed. In cycle 31 (*Commit* stage) it is checked whether the branch prediction was correct. In case of incorrect prediction, all subsequent instructions in execution are discarded, except for confirmed memory store operations.

   Check the status of the data structures after executing the *Commit* stage of the branch instruction in the first iteration.

A. Simulator Structure

The RISC-V/OOO simulator is composed of the following source files in C:

main.c Main program of the simulator. Responsible for reading the assembly code, executing the different stages of the algorithm, and printing the results.

main.h Contains all shared variables of the simulator: operators, reservation stations, load and store buffers, *Reorder Buffer (ROB)*, data memory, etc.

tipos.h Contains the definitions of all data structures used in the simulator: operators, reservation stations, load and store buffers, ROB, data memory, etc.

input.lex.l Contains the lexical description of the assembly language used.

input.yacc.y Contains the grammatical rules for the syntactic analysis of the assembly language.

etiquetas.c, etiquetas.h Contains the handling of labels in the assembler.

presentacion*.c, presentacion*.h Contains the functions for printing results.

prediccion.c Contains the functions for branch prediction.

f_busqueda.c Contains the implementation of the instruction fetch stage (IF).

f_lanzamiento_alum.c Contains the implementation of the multicyle instruction issue stage (Issue) of Tomasulo's algorithm with speculation. *This file must be modified.*

f_ejecucion.c Contains the implementation of the instruction execution stage.

f_transferencia_alum.c Contains the implementation of the common data bus transfer stage and write-back to the ROB (WB) based on Tomasulo's algorithm with speculation. *This file must be modified.*

f_confirmacion.c Contains the implementation of the commit stage (Commit) of Tomasulo's algorithm with speculation. *This file must be modified.*

instrucciones..{c,h} Contains the operation codes of the implemented instructions, some utility macros, and the code to display these instructions.

codop.{c,h} Operation codes of the implemented instructions.

Other less relevant files `simbolos.h`, `registros.h`, etc.

B. Data Structures

The following describes the data structures used (found in the file `tipos.h`) and their usage.

B.1. Basic Types

The basic types used are:

```
typedef int8_t byte;       /* One byte: 8 bits */
typedef int16_t half;      /* Halfword: 16 bits */
typedef int32_t word;      /* One word: 32 bits */
typedef int64_t dword;     /* Double word: 64 bits */

typedef uint32_t ciclo_t;

typedef enum {NO=0, SI=1} boolean; /* Logical value */

typedef int codop_t;    /* Operation code */
typedef int marca_t;    /* Tag/code type */
```

NOTE: The constant `MARCA_NULA`, defined in the file `main.h`, is used as a null tag for the tag fields of the reservation stations.

```
typedef union {
    xword int_d;     /* 32 or 64 bits */
    float fp_s;
    float fp_ps[2];
    double fp_d;
} valor_t;    /* Data used */
```

B.2. Register Files

Register files are vectors composed of elements of type `valor_t`. Each register has the following fields: `valor` and `rob`.

```
typedef struct {
    valor_t valor;         /* Register value */
    marca_t rob;           /* Register tag */
} reg_t;
```

B.3. Reservation Stations

A reservation station is composed of elements of type `estacion_t`. Each entry has the following fields: occupied bit, operation to perform, tag for the first operand, value of the first operand, tag and value for the second operand, memory address, store confirmation bit, and entry in the destination instruction's *reorder buffer*.

Additionally, the reservation station has a `resultado` field that contains the value obtained after performing the operation. The existence of this field allows the operator to

be released immediately after finishing the operation, rather than at the end of the transfer stage in Tomasulo's algorithm with speculation.

Finally, a `orden` field is added, allowing the age of the instruction that launched the operation to be determined, plus a `PC` field for visualization functions only.

```
typedef struct {
    boolean ocupado;         /* Occupied bit */
    codop_t OP;              /* Operation code to perform */

    marca_t Q1;              /* Tag for the first operand. ALU */
    valor_t V1;              /* Value of the first operand. ALU */

    marca_t Q2;              /* Tag for the second operand. ALU and SB */
    valor_t V2;              /* Value of the second operand. ALU and SB */

    marca_t Q3;              /* Tag for the third operand. ALU (MADD) */
    valor_t V3;              /* Value of the third operand. ALU (MADD) */

    dword direccion;         /* Memory access address. LB and SB */
    dword desplazamiento;    /* Memory access offset. LB and SB */
    boolean confirm;         /* Indicates whether the store operation
                                has been confirmed (commit). SB */
    marca_t rob;             /* Indicates who the operation is for. */
    valor_t resultado;       /* Operation result */
    ciclo_t orden;           /* Instruction order */

    ...                      /* Other less relevant fields */
} estacion_t;
```

All reservation stations for all operators, as well as the load and store buffers, will use the same reservation station type (`estacion_t`) to simplify the simulator programming.

B.4. Reorder Buffer

The *reorder buffer* is a vector composed of elements of type `reorder_t`. Each entry has the following fields: occupied bit, operation to perform, operation status, operation destination, operation result, and exceptions produced by the instruction.

Additionally, an `orden` field is added, allowing the age of the instruction that issued the operation to be determined, for visualization functions only, and a `PC` field, which contains the instruction address.

```
typedef struct {
    boolean ocupado;          /* Occupied bit */
    codop_t OP;               /* Operation code to perform */

    boolean completado;       /* Operation status */
    dword dest;               /* Destination register or SB
                                 or destination address */
    valor_t valor;            /* Operation result */

    dword direccion;          /* Branch address */
```

```
    boolean condicion;        /* Condition result */
    boolean prediccion;       /* Indicates whether it was predicted
                                 taken or not */
    int excepcion;            /* Indicates whether any exception
                                 occurred when executing this
                                 instruction */
    dword PC;                 /* Instruction memory address */
    ciclo_t orden;            /* Instruction order */

    ...                       /* Other less relevant fields */
} reorder_t;
```

B.5. Additional Structures

Below are the structures used for the implementation of the arithmetic and load/store operators, as well as the common data transfer bus.

The data bus consists of a structure of type `bus_comun_t`, which is made up of two fields: lines for the transfer of tags/codes, and lines for data transfer.

```
typedef struct {
    boolean ocupado;       /* Occupied line */
    marca_t codigo;        /* Lines for codes */
    valor_t valor;         /* Data lines: result, address or exception */
    boolean condicion;
    codigo_bus_t control; /* Indicates what is being transferred */
} bus_comun_t;
```

Each operator consists of a structure of type `operador_t`, with the following fields: occupied bit, code of the active station, entry of the *reorder buffer*, number of cycles executed for the active operation, and operator evaluation time.

```
typedef struct {
    boolean ocupado;          /* Occupied bit */
    tipo_operador_t tipo;     /* Operator type */
    int estacion;             /* Reservation station in use */
    marca_t codigo;           /* Reorder buffer code */
    ciclo_t ciclo;            /* Current cycle of the operation */
    ciclo_t Teval;            /* Evaluation time */

    ciclo_t orden;            /* Instruction order */
} operador_t;
```

C. Structure of the Dynamic Instruction Management Unit

The dynamic management unit is composed of the following elements:

Structure	Variable	Type	Size
FP Regs.	`Rfp`	`reg_t []`	`TAM_REGISTROS`
Integer Regs.	`Rint`	`reg_t []`	`TAM_REGISTROS`
Reorder Buffer	`RB`	`reorder_t []`	`TAM_REORDER`
Res. Station Add/Sub	`RS`	`estacion_t []`	`TAM_RS_SUMREST`: `INICIO_RS_SUMREST` .. `FIN_RS_SUMREST`
Res. Station Mul/Div	`RS`	`estacion_t []`	`TAM_RS_MULTDIV`: `INICIO_RS_MULTDIV` .. `FIN_RS_MULTDIV`
Res. Station Integers	`RS`	`estacion_t []`	`TAM_RS_ENTEROS`: `INICIO_RS_ENTEROS` .. `FIN_RS_ENTEROS`
Load Buffer	`LB`	`estacion_t []`	`TAM_BUFFER_CARGA`: `INICIO_BUFFER_CARGA` .. `FIN_BUFFER_CARGA`
Store Buffer	`SB`	`estacion_t []`	`TAM_BUFFER_ALMACEN`: `INICIO_BUFFER_ALMACEN` .. `FIN_BUFFER_ALMACEN`
Operators	`Op`	`operador_t []`	`INICIO_OP_ENTEROS` .. `FIN_OP_ENTEROS` `INICIO_OP_SUMREST` .. `FIN_OP_SUMREST` `INICIO_OP_MULTDIV` .. `FIN_OP_MULTDIV` `INICIO_OP_DIRECCIONES` .. `FIN_OP_DIRECCIONES` `INICIO_OP_MEMDATOS` .. `FIN_OP_MEMDATOS`
Branch Target Buffer	`BTB`	`entrada_btb_t []`	`TAM_BUFFER_PREDIC`
Common Bus	`BUS`	`bus_comun_t`	