2nd Lab: "PIPELINED INTRUCTION UNIT"

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

Goals:

- Complete logic required to solve data and control hazards in a pipelined processor.

Development:

The development of this lab is based on the use of a MIPS simulator working with integer instructions. The simulator allows direct interpretation of MIPS assembler code. It simulates a 5-stage pipelined (IF, ID, EX, MEM y WB). The code required to solve data and control hazards is not included in the simulator.

This document structures as follows. It first describes which are the files included in the simulator, then the implemented instructions, data structures under use, the internal structure of the pipelined MIPS simulator and, finally, the exercises to be carried out.

Structure of the simulator

The MIPS simulator includes the following file written in C:

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 cache, 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.

input.lex.l It contains the lexical description of the supported assembler language.

input.yacc.y It contains the grammatical rules for the synthactic analysis of the considered assembler code.

etiquetas.c, etiquetas.h They contain code related to the management of assembler labels.

presentacion-html.c,presentacion-txt.c They contain all functions required for presenting results.

mips.c It contains the implementation of all the stages of the instruction unit: IF, ID, EX, MEM and WB. *This is the file to be modified along this lab.*

Implemented instructions

The simulator executes all MIPS integer instructions, including versions register–register and register–immediate. Unconditional branch instructions are not supported. More precisely, it supports the following instruction set:

```
ADD Rx, Ry, Rz    ADDI Rx, Ry, #Inm    LW Rx, desp(Ry)
SUB Rx, Ry, Rz    SUBI Rx, Ry, #Inm    SW Rx, desp(Ry)
AND Rx, Ry, Rz    ANDI Rx, Ry, #Inm    BEQZ Rx, desp
OR Rx, Ry, Rz     ORI Rx, Ry, #Inm     BNEZ Rx, desp
XOR Rx, Ry, Rz    XORI Rx, Ry, #Inm    NOP
SRA Rx, Ry, Rz    SRAI Rx, Ry, #Inm    TRAP
SRL Rx, Ry, Rz    SRLI Rx, Ry, #Inm
SLL Rx, Ry, Rz    SLLI Rx, Ry, #Inm
SEQ Rx, Ry, Rz    SEQI Rx, Ry, #Inm
SNE Rx, Ry, Rz    SNEI Rx, Ry, #Inm
SGT Rx, Ry, Rz    SGTI Rx, Ry, #Inm
SGE Rx, Ry, Rz    SGEI Rx, Ry, #Inm
SLT Rx, Ry, Rz    SLTI Rx, Ry, #Inm
SLE Rx, Ry, Rz    SLEI Rx, Ry, #Inm
```

Data structures

Hereafter paragraphs describe existing data structures (defined at file `tipos.h`) their use.

Basic types

Existing basic types are:

```
typedef unsigned char   byte;   /* One byte: 8 bits */
typedef short           half;   /* Half-word : 16 bits */
typedef int             word;   /* One word: 32 bits */

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

Instruction format

Instruction formats are represented through an enumerated type:

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

Instructions are represented through the following data structure:

```
typedef struct {
  codop_t       codop;          /* Operation code */
  formato_t     tipo;           /* Format */

  byte          Rfuente1,       /* Source register 1 */
```

```
                 Rfuente2;       /* Source register 2 */
  byte           Rdestino;       /* Destination register */
  half           inmediato;      /* 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 {
  word           valor;              /* Register value */
} reg_int_t;
```

Inter-stage registers

Inter-stage registers are represented through an struct containing all the following fields:

- IF/ID register:

```
    typedef struct {
      instruccion_t IR;              /* IR */
      word          NPC;             /* PC+4 */

      word          iPC;
      long          orden;           /* Display information */
    } IF_ID_t;
```

- ID/EX Register:

```
    typedef struct {
      instruccion_t IR;              /* IR */
      word          NPC;             /* PC+4 */
      word          Ra,              /* Registers' value*/
                    Rb;
      word          Imm;             /* Immediate value with extended sign */

      word          iPC;
      long          orden;           /* Display information */
    } ID_EX_t;
```

- EX/MEM Register:

```
    typedef struct {
      instruccion_t IR;              /* IR */
      word          ALUout;          /* Result */
      word          data;            /* Data to be written */
      boolean       cond;            /* Result defining a branch condition */
      word          iPC;
      long          orden;          /* Display information */
    } EX_MEM_t;
```

- MEM/WB Register:

```
  typedef struct {
    instruccion_t IR;             /* IR */
    word          ALUout;         /* Result */
    word          MEMout;         /* Result */

    word          iPC;
    long          orden;          /* Display information */
  } MEM_WB_t;
```

Other data structures

- It defines how are data hazards solved:

```
  typedef enum {
    parada,               /* by stall insertion */
    cortocircuito,        /* by forwarding + stalls */
    ninguno               /* nothing */
  } riesgos_d_t;
```

- It defines how are control hazards solved:

```
  typedef enum {
    stall,                /* By stall insertion*/
    pnt,                  /* Predict-not-taken */
    ds3,                  /* Delayed branch, with delay slot(ds) =3 */
    ds2,                  /* Delayed branch, with ds=2 */
    ds1,                  /* Delayed branch, with ds=1 */
  } riesgos_c_t;
```

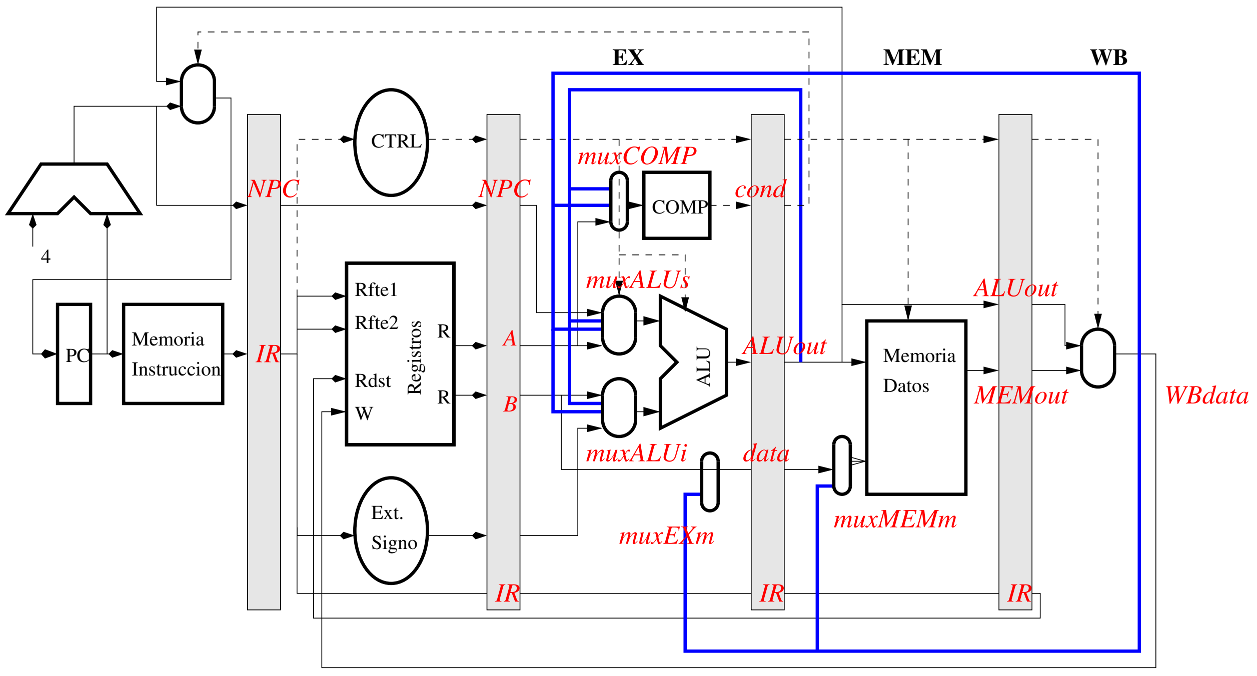
Structure of a pipelined instuction unit

The pipelined instruction unit is defined using the following elements (see figure):

Instructions memory. Stores the program to be executed. It can be addressed by word [1]. `MI` variable (of type `memoria_instruc_t`) represents such memory. Memory size is provided by constant `TAM_MEMO_INSTRUC` (see `tipos.h` file).

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

[1]successive words are assigned to consecutive memory addresses



Arithmetic operator. It performs the arithmetic operations in stage EX. It is represented by function `operacion_ALU (codop,in1,in2)` (see `mips.c` file), where `codop`, `in1` and `in2` respectively denote the operation to be carried out and and the operands to use.

Evaluation of branch conditions. Compute the condition of a branch. It is represented by the textttoperacion_COMP (codop,in1) function (see `mips.c` file), where `codop` and `in1` respectively represents the operation to be carried out and the data to be evaluated. This function returns whether the branch must be taken or not.

Multiplexor of the upper input of the arithmetic operator. It is represented by function `mux_ALUsup (npc,ra,mem,wb)` (see`mips.c` file), where `npc,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. It is represented by function `mux_ALUinf (rb,imm,mem,wb)` (see`mips.c` file), where `rb,imm,mem` y `wb`represent the inputs to the multiplexor. This function returns the selected input.

Multiplexor of the comparator input (branches). It is represented by function `mux_COMP (ra,mem,wb)` (see `mips.c` file), where `ra,mem` and `wb` represent the input comparators. The function returns the selected input.

Data multiplexor to be written in memory (EX stage). It is represented by function `mux_EXmem (rb,wb)` (see `mips.c` file), where `rb` and `wb` represent the multiplexor inputs. The function returns the selected input.

Data memory. It stores the data to be used by the program. It is addressed on a byte basis [2]. It is represented by variable textttMD, of type `memoria_datos_t`. Memory size is provided by constant `TAM_MEMO_DATOS` (see `tipos.h` file).

[2]successive words are located at addresses differing in 4 bytes

Multiplexor of data to be written in memory (MEM stage). It is represented by function `mux_MEMmem (rb,wb)` (see file`mips.c`), where `rb` and `wb` represent the multiplexor inputs. The function returns the selected input.

Output of the multiplexor of WB stage. It represents the data to be written in the register file during WB. It is represent by variable `WBdata` (see `main.h` file), of type `word`.

Inter–stage registers. Their names come from the stages they interconnect. They are the following ones:

- `IF_ID`, of type `IF_ID_t`.
- `ID_EX`, of type `ID_EX_t`.
- `EX_MEM`, of type `EX_MEM_t`.
- `MEM_WB`, of type `MEM_WB_t`.

Current and new PC value They are representes by `PC` and `PCn` variables, of type `word`. The following instruction will be fetched from the location in `PCn`.

Signals generated according to the type of instruction They are related to each stage. They detect certain features of the instruction located at each corresponding stage. Signals are represented by the following functions:

- `usaIR_fuente1_ID()`. The instruction in stage ID uses source register fuente1.
- `usaIR_fuente2_ID()`. The instruction in stage ID uses source register fuente2.
- `usaIR_fuente1_EX()`. The instruction in stage EX uses source register fuente1.
- `usaIR_fuente2_EX()`. The instruction in stage EX uses source register fuente2.
- `usaIR_destino_EX()`. The instruction in stage EX uses destination register.
- `usaIR_destino_MEM()`. The instruction in stage MEM uses destination register.
- `usaIR_destino_WB()`. The instruction in stage WB uses destination register.

Control signals The simulador has the following control signals, all of them of type `boolean`:

- `IFstall`: When it becomes active (`IFstall=SI`), it stalls the instruction in the IF stage during the following clock cycle.
- `IDstall`: When it becomes active instruction in ID stage is stalled during the following clock cycle.
- `IFnop`: When it becomes active , it transforms the instruction in the IF stage into a `nop` instruction, which will be sent to ID during the following clock cycle.
- `IDnop`: When it becomes active , it transforms the instruction in the ID stage into a `nop` instruction, which will be sent to EX during the following clock cycle.

- EXnop: When it becomes active , it transforms the instruction in the EX stage into a nop instruction, which will be sent to MEM during the following clock cycle.

MIPS simulator pseudo-code

After initializing data structures, the simulator main program loads the file containing the program to execute. Then it assembles and executes the main loop in the simulator, whose pseudo-code looks as follows:

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

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

  /*** Fase: MEM **************/
  memory_stage(): [ fase_memoria ()]
    -control hazard detection %detectar riesgos de control
    -use of shortcirtuits %aplicar cortocircuitos
    -memory access, if necessary %acceso a memoria, en su caso

  /*** Fase: EX *************/
  execution_stage(): [fase_ejecucion ()]
    -control hazard detection %detectar riesgos de control
    -use of shortcirtuits %aplicar cortociruitos
    -operation in ALU/COMP

  /*** Fase: ID **************/
  decoding_stage(): [fase_decodificacion()]
    -data hazard detection %detectar riesgos de datos
    -control hazard detection %detectar riesgos de control
    -register reading %leer registros

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

  Cycle++; %Ciclo++;
  print_state; %imprimir_estado;
  next_clock_cycle(); [impulso_reloj()]
```

Exercicies

First of all, check the simulator by using a simple fragment of code without data dependencies. This codes is in the file ejemplo.s

```
    .ireg 15,60 ; r1=15, r2=60
```

```
        .data
a:      .word  100
b:      .word  0
        .text
start: bnez r5,end
        add r3,r1,r2
        sub r4,r1,r2
        and r5,r1,r2
        or r6,r1,r2
        xor r7,r1,r2
        lw r1,a(r0)
        sw b(r0),r3
        sgt r2,r3,r4
        beqz r0,inicio ; unconditional branch
end:
```

In order to execute the simulator use the command `mips`. Simulator accepts several parameters:

```
mips -s salida -d riesgos_datos -c riesgos_control -f fichero.s
```

Parameter `-s salida` signals how we want the results. Options are `ciclo, final, html`, which respectively correspond to the execution step by step, the final results and the generation of results in *html*.

Parameter `-d riesgos_datos` defines the strategy that is used to solve data hazards. `n,p,c` are options corresponding to, respectively, no-strategy, stall insertion and forwarding.

Parameter `-c riesgos_control` denotes the control strategy to use. `p,t,3,2,1` are options corresponding to, respectively, stall insertion, *predict-not-taken* and delayed branch with *branch delay slot=3,2,1*.

By default, output is *html*, there is no data hazard detection and stalls are inserted in order to solve control hazards.

More precisely, we write:

```
mips -f ejemplo.s
```

This command will generate an **html** file for each cycle with all the information related to the state of the machine. Such files can be visualized using any web client (such as `firefox or konqueror`). **You are advised to delete these files before each new simulation:**

```
rm *.html
mips -f ejemplo.s
```

Check the correct behavior of the simulator, and the expected result of the execution. Obtained results must conform the following ones:

r0	r1	r2	r3	r4	r5	r6	r7	a	b
0	100	1	75	-45	12	63	51	100	75

However, the provided simulator only includes the necessary code to solve control hazards by inserting stalls or using the delayed branch technique with a *delay slot=3*. However, it is no able to detect or solve data hazards.

In this lab, the simulator must be enriched with new strategies for hazard detection and solving. In all cases, file `mips.c` much be modified by adding the necessary code in order perform the required actions.

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

The compilation of the `mips` simulator must be performed by executing the command `make` in the directory where the sources and the `Makefile` file are located.

In order to check the correct behavior of applied modifications, use the proof files provided.

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

   ```
   .ireg 15,60,0,65 ; r1=15, r2=60, r3=0, r4=65
   .text
   add r3,r1,r2
   sub r3,r3,r4
   ; r3 value must be 10
   ```

   To do so, the function decoding instructions (function `fase_decodificacion`) must be modified by writing the code activating necessary control signals.

   Check modifications by executing:

   ```
   mips -d p -f datos1.s
   ```

2. Modify the MIPS simulator in order to detect and solve data hazards by applying short-circuits.

   - First, data hazards provoked by the aforementioned sequence of code (file `datos1.s`) must be solved. Such sequence of instructions do not require the insertion of stalls. To do so, modify the function implementing the multiplexors located at the entry of the arithmetic logic operator (functions `mux_ALUsup` and `mux_ALUinf`).
     Check the modification by executing:

     ```
     mips -d c -f datos1.s
     ```

   - In this exercise the data hazard provoked by a load instruction followed by an arithmetic instruction (saved in file `datos2.s`):

     ```
             .ireg 0,0,0,65 ; r1=0, r2=0, r3=0, r4=65
             .data
     a:      .word 100
             .text
             lw r3,a(r0)
             sub r3,r3,r4
             ; Result must be r3=35
     ```

In this case, in addition to the activation of the corresponding short-circuit, the insertion of a stall is necessary in the decoding stage. Consequently, it must be modified both the function implementing the multiplexors located at the input of the arithmetic-logic operator (functions `mux_ALUsup` and `mux_ALUinf`), and the function (`fase_decodificacion`) implementing the decoding of instructions.

Check your code using the following command:

```
mips -d c -f datos2.s
```

3. Modify the MIPS simulator in order to solve control hazards using a *predict-not-taken* strategy.

   In order to do so, the function `fase_búsqueda` (the one carrying out with instruction fetching) must be modified.

   In order to check this modification, the following code (saved in `suma.s`) can be used.

```
        ; add vector componente until it is found
        ; a component equal to 0
        ; the result is stored in a
        .data
a:      .word  0
y:      .word  1,2,3,0,4,5,6,7,8
        .text
        add r2,r0,y; r2 traverse y
        add r1,r0,r0; r1=0
        nop ; avoid unforeseen data hazards
bucle:  lw r3,0(r2); r3 es y[i]
        add r1,r3,r1; this hazard is already solved
        add r2,r2,#4
        bnez r3,bucle
        sw a(r0),r1
final:
```

   Check your code using the following command:

```
mips -d c -c t -f suma.s
```