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

Computer architecture and engineering ($3th$ course)
E.T.S. de Ingeniería Informática (ETSINF)
Dept. of Computer Engineering (DISCA)

Goals:

- Knowing the use of a pipelined processor simulator.
- Analyzing the impact of control and data hazards in the performance of a pipelined instruction unit.
- Performing DLX assembler programs

Content:

The "DLX" (*DeLuXe*) was introduced as an example computer in the first edition (1990 year) of the famous *Computer Architecture: A Quantitative Approach* book written by J. Hennessy and D. Patterson. It is a 32-bits computer implementing a MIPS-like instruction set. At the end of this assignment a brief description of this instruction set is included.

The following piece of DLX code executes the following vector operation: $\vec{Z} = a + \vec{X} + \vec{Y}$:

```
; z = a + x + y
; Size of vectors: 16 words
; Vector x
        .data
x:      .word 0,1,2,3,4,5,6,7,8,9
        .word 10,11,12,13,14,15
; Vector y
y:      .word 100,100,100,100,100,100,100,100,100,100
        .word 100,100,100,100,100,100

; Vector z
; 16 elements are 64 bytes
z:      .space 64

; Scalar a
a:      .word -10

; The code
        .text

start:
        add r1,r0,x
```

```
        add r4,r1,#64 ; 16*4
        add r2,r0,y
        add r3,r0,z
        lw r10,a(r0)

loop:
        lw r12,0(r1)
        add r12,r10,r12
        lw r14,0(r2)
        add r14,r12,r14
        sw 0(r3),r14
        add r1,r1,#4
        add r2,r2,#4
        add r3,r3,#4
        seq r5,r4,r1
        beqz r5,loop

        trap #0         ; Program end
```

This program is saved in file `APXPY.S`.

Use of the DLX computer simulator

1. Execute the simulator of the DLX pipelined instruction unit (**DLXide**). It is able to simulate both the execution cycle of DLX instructions and the flow of such instructions through the computer datapath. All DLX instructions operating on the integer register file are supported. Instructions and data cache are separated (*Harvard* architecture). Registers are written and read in the first and second half of the clock cycle, respectively. Control hazards can be solved by using various strategies: using *stalls*, *predict-not-taken* y delayed branch, with two alternative *delay-slot*s of one or three instructions respectively. Data hazards can be solved by inserting stall or applying the *forwarding* technique. It must be noted that the simulated datapath changes according to the strategy employed to solve the hazards.

   In order to execute the simulator go to the directory created when sources (downloaded from PoliformaT) are decompressed, and executed the program **dlxide** from directory **DLXide**. In order to configure the simulated, go to menu **Simulador**, option *Configuración DLX*. The program will open a dialog windows showing the various available strategies that can be used to solve data and control hazards. By default, stalls are inserted in both cases.

2. Load the program `APXPY.S`. In order to do so, go to menu **Archivos**, option *Abrir*, and select the corresponding file. Once the program loaded, it must be assembled (use menu **Simulador**, option *Ensamblar*). If the program contains errors, they will be shown in the low part of the program window. After error correction, the program must be re-assemble. In case of containing no error, the program is stored in the memory of the simulated computer, and the action is reported to the user.

   Check that the loaded code corresponds to the one showed before.

3. Create the simulation window to executed the program (**Simulador** menu, *Ejecutar* option). A new window will be opened. The instructions–time diagram will be shown in such window. A window with multiple tabs enabling the inspection of the computer state will be also provided.

   The *Configuración* tab enables remembering which are the selected strategies for solving hazards. The *Registros* tab let us visualizing general use registers' content. By double clicking the "value"field you can alter its value. By clicking the left button the base used for coding numbers can be changed. PC Indexado and *Memoria* tabs enable visualizing the content of instruction and data cache, respectively. Finally, the *Estadísticas* tab reports the number of taken cycles, the set of executed instructions, the inserted stalls and the applied short-circuits.

   The simulator enables the execution of programs step (cycle) by step (cycle), forward various cycles (*Ejecutar Ciclos* button) or complete its execution until the end (*Ejecutar* button). After each clock cycles, the instructions–time diagram is updated. When an stall is inserted, stages keeping the same instructions are shown in italic. The computer data path is also updated. Each new fetched instruction has a new assigned color, which is used for signaling the path followed by the instruction within the instruction unit. When in one stage there is no instruction, it is shown the text `-nop-`[1]. In the low part of the data path window it also appears some control signals that are activated to insert stalls, abort instructions under execution or apply short-circuits:

   - *IF.stall*: Keeps the instruction in the IF stage in the same stage during the following clock cycle. It also sends to the ID stage the equivalent to a `nop` instruction.
   - *ID.stall*: Keeps the instruction in the ID stage in the same stage during the following clock cycle. It also sends to the EX stage the equivalent to a `nop` instruction.
   - *ID.nop, EX.nop, MEM.nop*: Converts the instruction in the corresponding stage to a `nop` instruction.
   - *WBtoMEM,WBtoEX,WBtoID,MEMtoEX,MEMtoID*: Applies a short-circuit between involved stages, by providing the adequate control signals to multiplexors.

   Using the default configuration (control hazards: *stalls* and data hazards: *stalls*), execute the program step by step for the first loop iteration. Observe how instructions evolve inside the pipelined instruction unit and the insertion of stalls when hazards are detected. Execute the whole program and check the results stored in memory at the address labelled as `z`, where the result vector is allocated. Note the number of executed instruction, the execution time and the number of inserted stalls. Compute the resulting CPI.

4. Modify the configuration of the simulator to solve control hazards using the *predict-not-taken* technique. Execute the program step by step for the first loop iteration.

[1] Avoid any confusion with the real `nop` operation

Observe how instructions evolve inside the pipelined instruction unit and the insertion of stalls when hazards are detected. Execute the whole program and check the results stored in memory at the address labelled as *z*, where the result vector is allocated. Note the number of executed instruction, the execution time and the number of inserted stalls. Compute the resulting CPI.

5. Keep the strategy for solving control hazards to *predict-not-taken* and modify the configuration of the simulator for solving data hazards relying on the *forwarding* technique. Execute the program step by step for the first loop iteration. Observe how instructions evolve inside the pipelined instruction unit and the insertion of stalls when hazards are detected. Execute the whole program and check the results stored in memory at the address labelled as *z*, where the result vector is allocated. Note the number of executed instruction, the execution time and the number of inserted stalls. Compute the resulting CPI.

Modification of the provided code.

The goal of this part of the lab is to carry out some code modification in order to reduce as much as possible the number of stalls.

1. Choose the *predict-not-taken* and *forwarding* techniques and modify the code to reduce the penalty relating to data hazards. Take into account that the only instructions that can introduce stall to solve data hazards are load instructions. Execute the program until the end and note the number of executed instructions, the execution time and the number of stalls. Compute the resulting CPI.

2. Keeping the *forwarding* technique to solve data hazards, select the *delay-slot 3* as a strategy to solve control hazards. Using the code produced for the previous section, modify such code for its execution trying to fill the existing *delay-slot* with useful instructions. Execute the program until the end and note the number of executed instructions, the execution time and the number of stalls. Compute the resulting CPI.

3. Keeping the *forwarding* technique to solve data hazards, select the *delay-slot 1* as a strategy to solve control hazards. Using the code produced for section 1, modify such code for its execution trying to fill the existing *delay-slot* with useful instructions. Execute the program until the end and note the number of executed instructions, the execution time and the number of stalls. Compute the resulting CPI.

Optimisation of a program.

The high level code shows how to sort a vector using the selection sort method:

```
...
FOR i := 0 TO n-2 DO
  FOR j := i+1 TO n-1 DO
    IF a(i)>a(j) THEN
       temp := a(i);
```

```
        a(i) := a(j);
        a(j) := temp;
      END;
    END;
END;
...
```

The equivalent DLX assembler program can be found in file `ORDENA.S`.

1. Take a little time to study and to understand the assembler version. Then configure the simulator in order to use the *forwarding* and the *predict-not-taken* techniques. Execute the program using the simulator and check its correct behavior. Analyze its execution time and its CPI.

2. Modify the produced program in order to use the delayed brach technique with a *delay-slot* set to 1. Configure the simulator and execute the program checking its correct behavior. Optimize the program in order to reduce the number of inserted stalls and filling the *delay-slot* with useful instructions. Analyze its execution time and its CPI.

El juego de instrucciones del DLX.

Instruction type/opcode	Instruction meaning
Data transfers	**Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR**
LB,LBU,SB	Load byte, load byte unsigned, store byte
LH,LHU,SH	Load half word, load half word unsigned, store half word
LW,SW	Load word, store word (to/from integer registers)
LF,LD,SF,SD	Load SP float, load DP float, store SP float, store DP float
MOVI2S, MOVS2I	Move from/to GPR to/from a special register
MOVF, MOVD	Copy one FP register or a DP pair to another register or pair
MOVFP2I,MOVI2FP	Move 32 bits from/to FP registers to/from integer registers
Arithmetic/logical	**Operations on integer or logical data in GPRs; signed arithmetic trap on overflow**
ADD,ADDI,ADDU, ADDUI	Add, add immediate (all immediates are 16 bits); signed and unsigned
SUB,SUBI,SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
MULT,MULTU,DIV,DIVU	Multiply and divide, signed and unsigned; operands must be FP registers; all operations take and yield 32-bit values
AND,ANDI	And, and immediate
OR,ORI,XOR,XORI	Or, or immediate, exclusive or, exclusive or immediate
LHI	Load high immediate—loads upper half of register with immediate
SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifts: both immediate (S__I) and variable form (S__); shifts are shift left logical, right logical, right arithmetic
S__,S__I	Set conditional: "__" may be LT,GT,LE,GE,EQ,NE
Control	**Conditional branches and jumps; PC-relative or through register**
BEQZ,BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC+4
BFPT,BFPF	Test comparison bit in the FP status register and branch; 16-bit offset from PC+4
J, JR	Jumps: 26-bit offset from PC+4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC+4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
RFE	Return to user code from an exception; restore user mode
Floating point	**FP operations on DP and SP formats**
ADDD,ADDF	Add DP, SP numbers
SUBD,SUBF	Subtract DP, SP numbers
MULTD,MULTF	Multiply DP, SP floating point
DIVD,DIVF	Divide DP, SP floating point
CVTF2D, CVTF2I, CVTD2F, CTD2I, CVTI2F, CVTI2D	Convert instructions: CVTx2y converts from type x to type y, where x and y are I (integer), D (double precision), or F (single precision). Both operands are FPRs.
__D,__F	DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register