Computer Architecture and Engineering - 3rd Year - ETSINF - UPV

Exercises of Unit 2. "Pipelined computers" (1)

Basic pipelining

MIPS data path

Ejercicio 2.1. A MIPS64-compatible processor integrates a pipeline of 5 stages:

IF: Instructions fetch.

ID: Instructions decoding and reading of register operands.

EX: ALU operation and PC updating in branch instructions.

M: Memory access in load and store instructions.

WB: Writing results back to registers

The machine solves control hazards using *predict-not-taken*. It also integrates two separated caches, one for instructions and another for data, and a multiport register file providing two read and one write ports.

n iterations of the following loop are executed on the processor:

```
        ...
L :     ld $t1,X($t2)
        dadd $t1,$t1,$t3
        sd $t1,X($t2)
        daddi $t2,$t2,8
        daddi $t4,$t4,-1
        bnez $t4,L
```

1. Draw a diagram indicating, for each instruction the clock cycle and the stage it has under completion. Consider only the first loop iteration. Compute the CPI and the execution time obtained according to the number of iterations n. Consider the following cases:

   a) Data hazards are solved using stalls.

   b) Data hazards are solved using short circuits.

Ejercicio 2.2. The following high-level code is available:

```
typedef struct elem
            int x[1..10];
            *elem next;
          ;

  int cont;
  *elem p;

...

do
```

```
  cont = cont + 1;
  p = (*p).next;
 while (p != NULL);
...
```

The type `int` and pointers take 32 bits. Constant `NULL` is represented with a 0. The compiler produces a loop as the following one:

```
; cont is located in R2
; p is located in R1
...
eti: add r2, r2, #1
     lw r1, 40(r1)
     bnez r1, eti
     nop          ; as many nops as necessary
```

This code executes on different versions of a processor pipelined in the following 5 stages:

IF: Instruction fetch.

ID: Instruction decoding and reading of register operands (during the 2^{nd} part of the clock cycle).

EX: ALU operation.

ME: Memory access in loads and stores.

WB: Result is written back to the destination register (during the 1^{st} part of the clock cycle).

The processor works at 100 MHz. It solves data hazards using the short-circuit technique and control hazards using *predict-not-taken*. It use a *Harvard* architecture (separated data and instruction caches).

Compute, for each one of the following cases, the *branch latency*, CPI, MIPS and execution time of the processor. Assume in your computation that the loop is executed 10.000 times:

1. The computation of the effective address and the branch condition, and the update of the PC are performed during the ID stage.
2. The computation of the effective address and the branch condition is performed during the EX stage, and the update of the PC is carried out during the MEM stage.

Ejercicio 2.3. A 2000$ computer incorporates a MIPS/LC processor. This processor has a MIPS-like pipelined instruction unit with 5 stages (IF : instruction fetch, ID : instruction decode, EX : execution, MEM : memory access, and WB : write into the register file). It incorporates a single cache for data and instructions, its *branch latency* is of 1 instruction, it works at 80MHz and it solves data hazards using the short-circuiting technique and control hazas using *predict-not-taken*. A public domain compiler is installed in the considered computer. It compiles the following code:

```
do {
  if (v[i] != 0) {
    temp = v[i];
    v[i] = w[i];
    w[i] = temp;
  }
  i = i-1;
} while (i != 0);
```

and generates the following one:

```
eti1:   ld r2,v(r1)
        beqz r2,eti2
        nop
        ld r3,w(r1)
        sd r3,v(r1)
        sd r2,w(r1)
eti2:   dadd r1,r1,-8
        bnez r1,eti1
        nop
```

The loop is typically applied on vectors containing a 80 % of components whose value is 0.

1. Compute the average CPI for handling vectors with big sizes (with n elements).
2. Assume that the original loop represents a normal workload of this computer. In order to improve the performance of the system, two options are considered:

   - Replace the existing processor with the new version MIPS/ST, which incorporates separate data and instruction caches. The resulting computer increases its price in 200$.
   - Buy a new commercial compiler, whose cost is of 200$, able to optimize the generated code by reducing in 3 clock cycles each loop iteration in which $v[i] \neq 0$ and in 2 cycles each iteration in which $v[i] = 0$. The number of instructions is not modified.

   Is any one of these options interesting? If it is, which one should be applied? Reason your answers from a cost/performance perspective, and assume that only 200$ are available for investment.

Ejercicio 2.4. The instruction cycle of an non-pipelined *load/store* processor is defined in terms of the following phases, whose duration is provided within a parenthesis:

- LI (10 ns): instruction fetch.
- DI (5 ns): instruction decoding and source register read.
- EXE (10 ns): computation of effective addresses in L/S instructions, operations in ALU instructions, condition and new PC value in branch instructions.
- EPC (5 ns): PC update in a branch instructions.
- MEM (10 ns): Memory access in L/S instructions
- ER (5 ns): Write the value of destination registers in store and ALU instructions.

The control circuitry manages the aforementioned phases according to the operation code of each instruction. The clock works at 200 MHz, so certain phases executes in 1 cycle and others in 2 cycles. All instruction cycles start with phases LI and DI but depending on the type of instruction, the other phases are (the frequency of each type of instruction is indicated in parenthesis):

- Load instructions (20 %): EXE, MEM and ER
- Store instructions (10 %): EXE and MEM

- ALU instructions (50 %): EXE and ER
- Branch instructions (20 %): EXE and EPC

This processor is expected to be pipelined, by using registers with 2 ns of delay and a clock with a null skew. The WPC phase disappears and all the branch-related logic is moved to the DI stage, whose duration is increased to 10 ns. So, the pipelined version of the processor integrates 5 stages LI, DI, EXE, MEM y ERB. Real measures are obtained from this new processor and it is observed that (i) 5 % of load instructions generate 1 stall due to a data hazard and, (ii) the compiler fills 10 % of the *branch delay slot* cases with a `nop` instruction.

Taking this information into account, compute:

1. CPI of the non-pipelined processor.
2. Clock frequency of the pipelined processor.
3. Number of instructions executed by the pipelined processor with respect to the non-pipelined one.
4. CPI of the pipelined processor.
5. Speed-up obtained through pipelining.

Ejercicio 2.5.

A 5-stage pipelined processor is available (IF: instruction fetch; ID: instruction decoding and register file read; EX: ALU operation; MEM: memory access and WB: write to the register file). The processor integrates the MIPS instruction set and it has a separate instruction and data cache. Its clock frequency is of 200 MHz. Data and control hazards are solved using respectively the *forwarding* technique and inserting 2 stalls each time a branch instruction appears.

Programs execute, in average, 18 % of branch, 39 % of load/store and 43 % of arithmetic instructions. Loads are 2 times more frequent than stores. *bytes* and *halfwords* accesses are 20 % of all memory accesses. The frequency of data hazards between a LOAD instruction and another following one consuming the data loaded from memory is the following one:

Frequency: 25 %	Frequency: 15 %
`LOAD R1, ...` Instructions reading `R1` ...	`LOAD R1, ...` Instruction not reading `R1` Instruction reading `R1`

In order to improve the performance, the following modifications are proposed:

- Remove from the instruction set those instructions providing access to *bytes* and *halfwords*. As a result, programs requiring such functionality must use other processor instructions:

`LB R1,x`
or
`LH R1,x`
	`LW R1,x'`
	`SRL R1,R1,#pos`
	`AND R1,R1,#masc`

`SB R1,x`
or
`SH R1,x`
	`LW R2,x'`
	`SLL R1,R1,#pos`
	5 more ALU instrutcions
	`SW x',R1`

- Since the processor design has been simplified, increase its clock frequency.

Answer the following questions:

1. CPI of the original processor.

2. Number of instructions of the modified processor.
3. CPI of the modified processor.
4. The minimum clock frequency that should be attained in order to justify the inclusion of the proposed modifications.

Ejercicio 2.6. A computer with memory-register architecture has a 6-stage pipelined instruction cycle:

- F: Instruction fetch and PC increment.
- RF: Decoding and register file read (2^{nd} part of the clock cycle).
- ALU1: Computation of the effective address in memory accesses and branches.
- MEM: Memory access.
- ALU2: Arithmetic operation, evaluation of branch conditions and new PC updating.
- WB: Write to the register file (1^{st} part of the clock cycle)

All instructions are executed in 6 stages. There are two types of arithmetic instructions:

Type R: ALUop Rd,Rs,Rt

Type M: ALUop Rd,Rs,desp(Rt)

1. In order to avoid structural hazards, how many adders are (at least) required for pipelining purposes? Determine the minimum number of read/write ports in the file register and in the memory, in order to avoid structural hazards.
2. If the machine uses the delayed branch technique, which is the value of its delay- slot?
3. Is it interesting to apply a *predict-taken* strategy towards positions in the code preceding the branch? If it is, how many clock cycles of penalty are induced by correctly predicted branches?

Ejercicio 2.7.

The following instruction-time diagrams correspond to the execution of various code fragments from various computers. Determine for each of them, which technique is used for solving data hazards (stall insertion or short-circuit) and control hazards (stall insertion, *predict-not-taken* or delayed branch), and in which stage is the PC updated.

1.
```
L     LW r2,a(r1)    IF ID EX M  WB
L+4   ADD r3,r2,r3       IF ID ID EX M  WB
L+8   ADD r3,r4,r3          IF IF ID EX M  WB
L+12  SUB r1,r1,#4                IF ID EX M  WB
L+16  BNEZ r1, L                     IF ID EX M  WB
L+20  SW z(r0), r3                      IF ID
L+24  ADD r3,r0,r0                         IF
L     LW r2,a(r1)                             IF ID EX M  WB
```

2.
```
L     LW r2,a(r1)    IF ID EX M  WB
L+4   ADD r3,r2,r3      IF ID ID EX M  WB
L+8   ADD r3,r4,r3         IF IF ID EX M  WB
L+12  SUB r1,r1,#4            IF ID EX M  WB
L+16  BNEZ r1, L                 IF ID EX M  WB
L+20  SW z(r0), r3                  IF IF IF
L     LW r2,a(r1)                             IF ID EX M  WB
```

3.
```
L     LW r2,a(r1)    IF ID EX M  WB
L+4   SUB r1,r1,#4      IF ID EX M  WB
L+8   ADD r3,r2,r3         IF ID ID EX M  WB
L+12  BNEZ r1, L              IF IF ID EX M  WB
L+16  ADD r3,r4,r3                IF ID ID EX M  WB
L     LW r2,a(r1)                    IF IF ID EX M  WB
```

Multicycle operators

Ejercicio 2.8. A MIPS processor is available. It supports the following multi-cycle instructions:

- Pipelined Adder / Subtracter (Tev=2, IR=1)
- Pipelined Multiplier (Tev=3, IR=1)
- Conventional Divider (Tev=4, IR=1/4)

In this processor, structural and data hazards are detected during the ID stage, inserting stall cycles when necessary. Short-circuits are also used.

Show the execution diagram of the following fragment of code:

```
L.D    F1, 0(R1)
DIV.D  F4, F0, F1
ADD.D  F2, F3, F4
L.D    F4, 4(R1)
MULT.D F3, F4, F2
L.D    F5, 8(R1)
```

The supplied diagram must represent the stages crossed by each instruction using the following notation: IF (fetching stage), ID (decoding stage), EX (monocyle execution stage), A1, A2 (adder/subtracter execution stages), M1, M2, M3 (multiplier execution stages), D1, D2, D3, D4 (divider execution stages), ME (memory access stage) and WB (register writing stage).

Ejercicio 2.9.

A processor executes the following loop computing $\vec{z} = A\vec{x} + B\vec{y}$:

```
loop: l.d F0,x(r10)
      l.d F1,y(r11)
      mult.d F4,F2,F0;  F2 contains A.
      mult.d F5,F3,F1;  F3 contains B.
      add.d F6,F4,F5
      daddi r14,r14,-1
      daddi r10,r10,8
      daddi r11,r11,8
```

```
s.d F6,z(r12)
daddi r12,r12,8
bnez r14,loop
nop
```

The processor provides two register files to store integer and floating point data. In addition, the following multi-cycle operators are available for floating point operations:

- A pipelined multiplier with $T_{ev} = 5$ and $IR = 1$, with stages M1, M2, etc.
- A non-pipelined adder with $T_{ev} = 3$ and $IR = 1/3$, with stages A1, A2, etc.

Other instructions are executed using a classic pipeline of 5 stages (IF,ID,EX,ME,WB). Data hazards are solved using short-circuits and inserting stalls whenever necessary. Conditional branches use the *predict-not-taken* technique. Branch condition and destination address are computed along ID stage. If the branch is taken, then the PC is updated with the new destination address at the end of such stage.

It is required:

1. The instructions-time diagram of the first loop integration and the first instruction of second loop iteration.
2. Assuming the equality of all loop iterations, compute the average loop CPI for n iterations.
3. In order to speed up the loop execution, two alternatives are under study: a) replace the non-pipelined adder by a pipelined one with $Tev = 3$ and $IR = 1$, of b) replace the pipelined multiplier by a non-pipelined one with $Tev = 2$ and $IR = 1/2$. Which option is the most suitable one for reducing the loop execution time? Justify your answer.

Static instruction scheduling

Ejercicio 2.10. A computer providing an MIPS-compatible instruction ser is available. The following sequence of code is executed in this computer:

```
i1       L.D F0,X(R1)
i2       MULT.D F0,F0,F4
i3       L.D F2,Y(R1)
i4       ADD.D F0,F0,F2
i5       S.D F0,Y(R1)
i6       DSUB R1,R1,#8
i7       BNEZ R1,L
i8       L.D F0,X(R1)
i9       MULT.D F0,F0,F4
i10      L.D F2,Y(R1)
i11 L:   DADD R1,R0,#dir
```

Identify at least two dependencies of each type in the above fragment of code.

Ejercicio 2.11. Consider the following MIPs assembler code:

```
loop:   L.D F0, 0(R1)
        MULT.D F0, F0, F10
        ADD.D F0, F0, F11
        S.D F0, 0(R1)
        DADD R1, R1, #8
        BNE R1, R3, loop
```

This code is executed on a MIPS where integer instructions traverse the following pipeline: IF (instruction fetch), ID (instruction decoding, source register reading and hazard detection) EX (execution), M (memory access) y WB (writeback), while floating point instructions follow these stages: IF, ID, En (execution in the corresponding multicycle operator) and WB. Data hazards can be solved using short-circuits, inserting the necessary stalls in ID, and control hazards through *predict-not taken*, updating the PC in ID.

The processor works at 4 GHz and the CPI for ALU instructions is 1.

Multi-cycle functional units have the following features:

Operator	Number	Latency	Type
Multiplication	1	3 cycles	Pipelined
Add/Sub	1	3 cycles	Pipelined

1. Identify a data dependency, an antidepency, and output dependency and a control dependency in the original code.

2. Provide the instructions–time diagram for the first loop iteration. Compute the execution time for n iterations.

3. Show the code resulting from the use of the *loop-unrolling* technique. Without providing the instructions–time diagram, compute the execution time of the resulting code and the speed-up (if any) wrt the original one.

Dynamic branch prediction

Ejercicio 2.12.

A processor has a dynamic branch predictor of type BTB (*Branch Target Buffer*) that delivers its prediction during the instruction fetch stage. The target address and branch condition are computed at the 3rd stage of the instruction cycle. The probability that a branch is found in the table is 80 % and the predictor accuracy is 90 %. The branches are effective in 60 % of the cases. Questions:

1. Show the instructions that will be fetched after the branch and their i–t diagram for the following options:

   - There is no entry in the prediction table and the branch is **not taken**.
   - There is no entry in the prediction table and the branch is **taken**.
   - The predictor predicts that the branch is **not taken** and the branch is finally **not taken**.
   - The predictor predicts that the branch is **not taken** and the branch is finally **taken**.
   - The predictor predicts that the branch is **taken** and the branch is finally **not taken**.
   - The predictor predicts that the branch is **taken** and the branch is finally **taken**.

   The instructions should be represented as **Branch**, the branch instruction, **PC+i** (i= 1, 2, ...), the subsequent instructions after the branch, **Dest**, the branch target instruction, and **Dest+i** (i= 1, 2, ...), the subsequent instructions after the branch target instruction. The instruction stages should be represented as **F1**, **F2**, etc.

2. Compute the average CPI for branch instructions.

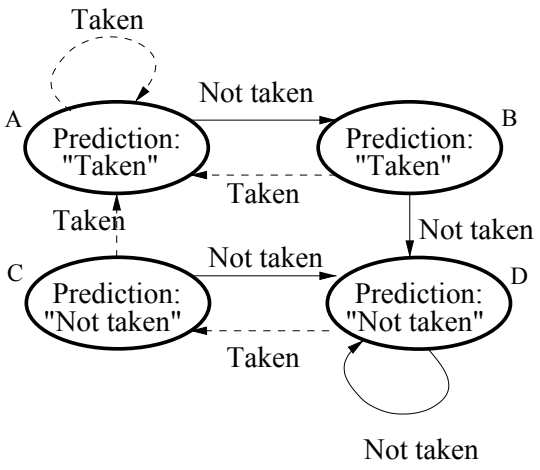
3. Suppose we can modify the BTB design in two ways. The first one increases the number of table entries, so that it stores 90 % of the executed branches. The second one uses a 2-bit predictor so that the prediction accuracy is raised to 95 %. Which modification is able to reduce the CPI of branch instructions?

Ejercicio 2.13.

Consider a processor with an instruction set similar to MIPS and a pipelined unit consisting of the following stages:

- **IF** Instruction fetch.
- **ID** Instruction decode and register read.
- **ALU** Computation of the branch target and memory access addresses.
- **MEM** Memory access.
- **EX1** First execution stage and computation of branch condition.
- **EX2** Second execution stage.
- **WB** Register write.

Two branch prediction schemes are considered for implementation in the processor. The predictors that have to be evaluated are: A *Branch Prediction Buffer* and a *Branch Target Buffer*, both of them delivering their prediction at the end of the ID stage. Both mechanisms are implemented with 4 entries in the *buffer* and use a 2-bit predictor whose operation is illustrated in the figure:



For evaluating the prediction mechanisms a test program is used, from which a fragment is shown below:

```
Address       Instructions
...
0x03          add r1, r0, r0
      lfor:
...
0x05          beqz r1, lendif
...
      lendif:
...
      ldo:
0x09          sub r8, r8, r2
0x0A          slt r3, r2, #2
0x0B          seq r4, r1, r0
0x0C          and r5, r3, r4
0x0D          beqz r5, lbreak
```

```
Address       Instructions

...
0x10          add r2, r2, #1
0x11          slt r4, r2, #3
0x12          bnez r4, ldo
      lbreak:
...
0x15          add r1, r1, #1
0x16          slt r6, r1, #2
0x17          bnez r6, lfor
0x18          sw z(r0), r8
```

Initially the *Branch Prediction Buffer* contains all entries in state "D", and all the *Branch Target Buffer* entries are empty. When a new entry is added to the *Branch Target Buffer*, its status will be "A" if the branch has been taken, and "D" if the branch has not been taken.

The final statistics of the execution of the test program are: 15 % of the instructions are conditional branches, 60 % of the branches are taken, the *Branch Prediction Buffer* succeeds in predicting 75 % of the cases, and the *Branch Target Buffer* succeeds in predicting 90 %, including cases in which there is no entry in the table (which are predicted as "not taken").

It is requested:

1. Obtain a trace of the execution of the code fragment until the "`sw z(r0), r8`" instruction is completely executed. It should show the contents of the entries of both predictors after each branch ("`beqz r1, lendif`", "`beqz r5, lbreak`", "`bnez r4, ldo`" and "`bnez r6, lfor`"). Labels should be used to record the target addresses.

Branch `beqz r1, lendif`. (r1 = 0)

BPB

Index	State
00	
01	
10	
11	

BTB

Index	Target address	State

Branch `beqz r5, lbreak`. (r5 = 1)

BPB

Index	State
00	
01	
10	
11	

BTB

Index	Target address	State

Branch `bnez r4, ldo`. (r4 = 1)

BPB

Index	State
00	
01	
10	
11	

BTB

Index	Target address	State

Branch `beqz r5, lbreak`. (r5 = 1)

BPB

Index	State
00	
01	
10	
11	

BTB

Index	Target address	State

Branch `bnez r4, ldo`. (r4 = 1)

BPB

Index	State
00	
01	
10	
11	

BTB

Index	Target address	State

Branch `beqz r5, lbreak`. (r5 = 0)

BPB

Index	State
00	
01	
10	
11	

BTB

Index	Target address	State

Branch `bnez r6, lfor`. (r6 = 1)

BPB

Index	State
00	
01	
10	
11	

BTB

Index	Target address	State

Branch `beqz r1, lendif`. (r1 = 1)

BPB

Index	State
00	
01	
10	
11	

BTB

Index	Target address	State

Branch `beqz r5, lbreak`. (r5 = 0)

BPB

Index	State
00	
01	
10	
11	

BTB

Index	Target address	State

Branch `bnez r6, lfor`. (r6 = 0)

BPB

Index	State
00	
01	
10	
11	

BTB

Index	Target address	State

2. Analyze the behavior of the BTB predictor when it mispredicts, indicating which instructions are executed after the branch. The number of lost execution cycles should be indicated. Canceled instructions should be represented as **X** in the corresponding cycle. The instructions following the branch will be represented as **pc+1**, **pc+2**, ... and the branch target instructions as **dest**, **dest+1**, etc.

3. Compute the average number of cycles per instruction (CPI) for the test program, using the BTB prediction scheme. Assume that instructions that are not branches are executed with CPI=1.

Ejercicio 2.14.

Assume the code of function `ones`, which returns (in `$v0`) the number of bits equal to 1 in *a*rgument register (`$a0`). The way to proceed is to obtain the LSB of the argument (with `andi $t1,$a0,1`), to increase the count in case that LSB==1 (with `daddi $v0,$v0,1`), and then to shift right 1 bit (using `dsrl $a0,$a0,1`). These operations are repeated 64 times to complete the job.

```
ones:  li $t0,64        # Number of iterations
       li $v0,0         # Initial count = 0
loop:  andi $t1,$a0,1   # $t1 = LSB of register $a0
       beqz $t1,next    # if (LSB!=0 )
       daddi $v0,$v0,1  #    $v0++ /* Increase count - LSB=1 */
next:  dsrl $a0,$a0,1   # Shift right $a0 1 bit
       daddi $t0,$t0,-1 # Pending iterations
       bgtz $t0,loop    # Next iteration
       jr $ra           # Exit of the function
```

Note that the code contains two conditional branches and one jump instruction:

`beqz $t1,next` taken each time that LSB==0

`bgtz $t0,loop` closes the loop

`jr $ra` unconditional jump.

The processor pipeline consists of the typical 5 stages, and implements a branch predictor. PC is written at stage *EX* (i.e., the branch latency is 2 cycles long). Note that no stall cycle appears due to structural or data hazards.

Obtain the number of executed instructions, the number of stall cycles, and the execution time (in processor cycles) of the function in the following cases:

1. Assuming a *predict-not taken* predictor and `$a0`$=-1$ (0xFFF...FFF).

2. The processor implements a BTB for (conditional) branches with a 1-bit predictor, which predicts the target address at *IF* stage and applies *predict-not taken* for return of function `jr $ra`. Function `ones` is called the first time with `$a0`$=-1$ (0xFFF...FFF)

3. The previous BTB uses a 2-bit hysteresis branch predictor. Function `ones` is called the first time with `$a0`=0x8080 8080 8080 8080