Universidade de Aveiro

Mestrado em Engenharia de Computadores e Telemática

Arquitecturas de Alto Desempenho

DLX – Pipelining 1

Academic year 2022/2023 Adaptation of exercise guide by Nuno Lau/José Luís Azevedo

2. Consider the programs `ex_1.s` and `ex2.s` located in the directory `./DLX/apps/dlx_apps` in turn. Read them carefully, trying to understand what they do.

```
                                ; ex1.s
        .text
        .global   main
main:    addi     r2,r0,2       ; r2 = 2
         addi     r3,r0,3       ; r3 = 3
         addi     r5,r0,5       ; r5 = 5
         addi     r7,r0,7       ; r7 = 7
         addi     r9,r0,9       ; r9 = 9
         addi     r11,r0,11     ; r11 = 11
         add      r1,r2,r3
         sub      r4,r1,r5
         and      r6,r1,r7
         or       r8,r1,r9
         xor      r10,r1,r11
         trap     0             ; end of program
```

2.1. Determine the values that should be stored in the registers at the end of the execution.

```
r1 = 5                r4 = 0            r6 = 5
r8 = 13 = 0xD         r10= 14 = 0xE
```

2.2. Sketch the clock cycle diagram of the execution in a processor with the five-stage pipeline architecture depicted below.

	Clock cycle															
Instruction	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
`addi $2,$0,2`	IF	ID	EX	MEM	WB											
`addi $3,$0,3`		IF	ID	EX	MEM	WB										
`addi $5,$0,5`			IF	ID	EX	MEM	WB									
`addi $7,$0,7`				IF	ID	EX	MEM	WB								
`addi $9,$0,9`					IF	ID	EX	MEM	WB							
`addi $11,$0,11`						IF	ID	EX	MEM	WB						
`add $1,$2,$3`							IF	ID	EX	MEM	**WB**					
`sub $4,$1,$5`								IF	**ID**	EX	MEM	WB				
`and $6,$1,$7`									IF	**ID**	EX	MEM	WB			
`or $8,$1,$9`										IF	ID	EX	MEM	WB		
`xor $10,$1,$11`											IF	ID	EX	MEM	WB	
`trap 0`												IF	ID	EX	MEM	WB

2.3. Run the code in the DLX simulator with the *forwarding* option turned off. Check if the execution is correct. If not, explain what has happened?

```
r1 = 5          r4 = -5 = 0xFFFFFFFB     r6 = 0
r8 = 13 = 0xD   r10= 14 = 0xE
```

Register 1 is written in clock cycle 10 and read in clock cycles 8 and 9 (see diagram above).

2.4. Intersperse `nop` instructions in the code so that the program now behaves as expected. What is the new clock cycle count for running the program?

```
         .text
         .global   main
main:     addi      r2,r0,2      ; r2 = 2
          addi      r3,r0,3      ; r3 = 3
          addi      r5,r0,5      ; r5 = 5
          addi      r7,r0,7      ; r7 = 7
          addi      r9,r0,9      ; r9 = 9
          addi      r11,r0,11    ; r11 = 11
          add       r1,r2,r3
          nop
          nop
          sub       r4,r1,r5
          and       r6,r1,r7
          or        r8,r1,r9
          xor       r10,r1,r11
          trap      0            ; end of program
```

18 clock cycles.

2.5. Run the original code in the DLX simulator with the *forwarding* option turned on. Compare the results with the previous runs.

The program behaves as expected, the `NOP` instructions are not required.

2. Consider the programs `ex_1.s` and `ex2.s` located in the directory `./DLX/apps/dlx_apps` in turn. Read them carefully, trying to understand what they do.

```
                                     ; ex2.s
        .text
        .global   main
main:   addi      r2,r0,0x4000       ; r2 = 0x4000
        addi      r5,r0,5            ; r5 = 5
        addi      r7,r0,7            ; r7 = 7
        addi      r9,r0,9            ; r9 = 9
        lw        r1,0(r2)
        sub       r4,r1,r5
        and       r6,r1,r7
        or        r8,r1,r9
        trap      0                  ; end of program
```

2.1. Determine the values that should be stored in the registers at the end of the execution.

```
r1 = 0x20024000        r4 = 0x20023FFB
r6 = 0                 r8 = 0x20024009
```

2.2. Sketch the clock cycle diagram of the execution in a processor with the five-stage pipeline architecture discussed at the lectures.

	Clock cycle												
Instruction	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
`addi r2,r0,0x4000`	IF	ID	EX	MEM	WB								
`addi r5,r0,5`		IF	ID	EX	MEM	WB							
`addi r7,r0,7`			IF	ID	EX	MEM	WB						
`addi r9,r0,9`				IF	ID	EX	MEM	WB					
`lw r1,0(r2)`					IF	ID	EX	MEM	**WB**				
`sub r4,r1,r5`						IF	**ID**	EX	MEM	WB			
`and r6,r1,r7`							IF	**ID**	EX	MEM	WB		
`or r8,r1,r9`								IF	ID	EX	MEM	WB	
`trap 0`									IF	ID	EX	MEM	WB

2.3. Run the code in the DLX simulator with the *forwarding* option turned off. Check if the execution is correct. If not, explain what has happened?

```
r1 = 0x20024000        r4 = 0xFFFFFFFB
r6 = 0                 r8 = 0x20024009
```

`r1` is written in clock cycle 8 and read in clock cycles 6 and 7 (see diagram above).

2.4. Intersperse `nop` instructions in the code so that the program now behaves as expected. What is the new clock cycle count for running the program?

```
        .text
        .globl    main
main:   addi      r2,r0,0x4000       ; r2 = 0x4000
        addi      r5,r0,5            ; r5 = 5
        addi      r7,r0,7            ; r7 = 7
        addi      r9,r0,9            ; r9 = 9
        lw        r1,0(r2)
        nop
        nop
        sub       $4,$1,$5
        and       $6,$1,$7
        or        $8,$1,$9
        trap      0                  ; end of program
```

15 clock cycles.

2.5. Run the original code in the DLX simulator with the *forwarding* option turned on. Compare the results with the previous runs.

The program behaves as expected, but one `NOP` instruction is always required because the value read from memory is only available at the end of clock 7 (see diagram above).

```
         .text
         .globl    main
main:     addi     r2,r0,0x4000  ; r2 = 0x4000
          addi     r5,r0,5       ; r5 = 5
          addi     r7,r0,7       ; r7 = 7
          addi     r9,r0,9       ; r9 = 9
          lw       r1,0(r2)
          nop
          sub      $4,$1,$5
          and      $6,$1,$7
          or       $8,$1,$9
          trap     0             ; end of program
```

3. Write a program that adds up the values of an integer array stored in memory. Run the code in the DLX simulator with the *forwarding* option turned off and then on. Take also into account in your runs the *branch predictor* alternatives: *none*, *static – predict always not taken* and *static – predict always taken*, and *initial predictor state* equal to *no initial state*. Discuss the results.

   Algorithm

```
int values[] = {1,2,3,4,5,6,7,8,9,10};
int nelem = 10;
int sum;
int i;
sum = 0;
for (i = 0; i < nelem; i++)
  sum += values[i];
```

   Program architecture

```
r1 -> add(nelem), val(nelem)
r2 -> add(values[0])
r3 -> val(sum)
r4 -> val(i)
r5 -> val(values[i])
r6 -> add(sum)
r7 -> val(nelem) - i
```

   Original code

```
            .data
values:     .word   1,2,3,4,5,6,7,8,9,10         ; values to be added
nelem:      .word   10                                   ; array size
sum:        .space  4                       ; sum of the array elements
            .text
            .global  main
main:        addi    r1,r0,nelem     ; r1 = add(nelem)
             lw      r1,0(r1)        ; r1 = val(nelem)
             addi    r2,r0,values    ; r2 = add(values[0])
             addi    r3,r0,0         ; r3 = sum = 0 (partial sum of
                                     ;          the array elements)
             addi    r4,r0,0         ; r4 = i = 0 (counting variable)
loop:        lw      r5,0(r2)        ; r5 = val(values[i])
             add     r3,r3,r5        ; r3 = sum + val(values[i])
             addi    r4,r4,1         ; r4 = i = i + 1
             addi    r2,r2,4         ; r2 = add(values[i])
             sub     r7,r1,r4        ; r7 = val(nelem) - i
             bnez    r7,loop         ; are all array elements summed?
             addi    r6,r0,sum       ; r6 = add(sum)
             sw      0(r6),r3        ; save sum of the array elements
             trap 0                  ; end of program
```

   Due to data hazards and a control hazard, the program will not run correctly. To see why the clock cycle diagram of the execution in a processor with the five-stage pipeline architecture depicted in the guide.

	Clock cycle																	
Instruction	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
`addi r1,r0,nelem`	IF	ID	EX	MEM	**WB**													
`lw   r1,0(r1)`		IF	**ID**	EX	MEM	WB												
`addi r2,r0,values`			IF	ID	EX	MEM	WB											
`addi r3,r0,0`				IF	ID	EX	MEM	WB										
`addi r4,r0,0`					IF	ID	EX	MEM	WB									
`lw   r5,0(r2)`						IF	ID	EX	MEM	**WB**								
`add  r3,r3,r5`							IF	**ID**	EX	MEM	WB							
`addi r4,r4,1`								IF	ID	EX	MEM	**WB**						
`addi r2,r2,4`									IF	ID	EX	MEM	WB					
`sub  r7,r1,r4`										IF	**ID**	EX	MEM	**WB**				
`bnez r7,loop`											IF	**ID**	EX	MEM	WB			
`addi r6,r0,sum`												IF	ID	EX	MEM	**WB**		
`sw   0(r6),r3`													IF	**ID**	EX	MEM	WB	
`trap 0`														IF	ID	EX	MEM	WB

By interspersing nop instructions in the code, the program can be made to work correctly. One should notice that the DLX processor asserts the branching condition and computes the target address only at the *execution / effective address* stage (EX) and not at the *instruction decode / register fetch* stage (ID) as in the example discussed at the lectures. Therefore, if the branch is taken, the PC is updated at the clock cycle corresponding to the *memory access* stage (MEM) of the branch instruction (three time delays are required).

Code with `nop` instructions interspersed to make it run as intended

```
             .data
values:      .word   1,2,3,4,5,6,7,8,9,10         ; values to be added
nelem:       .word   10                                   ; array size
sum:         .space  4                     ; sum of the array elements
             .text
             .global  main
main:         addi    r1,r0,nelem     ; r1 = add(nelem)
              nop
              nop
              lw      r1,0(r1)        ; r1 = val(nelem)
              addi    r2,r0,values    ; r2 = add(values[0])
              addi    r3,r0,0         ; r3 = sum = 0 (partial sum of
                                      ;         the array elements)
              addi    r4,r0,0         ; r4 = i = 0 (counting variable)
loop:         lw      r5,0(r2)        ; r5 = val(values[i])
              nop
              nop
              add     r3,r3,r5        ; r3 = sum + val(values[i])
              addi    r4,r4,1         ; r4 = i = i + 1
              addi    r2,r2,4         ; r2 = add(values[i])
              nop
              sub     r7,r1,r4        ; r7 = val(nelem) - i
              nop
              nop
              bnez    r7,loop         ; are all array elements summed?
              nop
              nop
              nop
              addi    r6,r0,sum       ; r6 = add(sum)
              nop
              nop
              sw      0(r6),r3        ; save sum of the array elements
              trap 0                  ; end of program
```

It takes 156 clock cycles to run the program.

The program can be run faster if some instruction reordering is carried out. In fact, the number of interspersed nop instructions can be reduced from 12 to 5 and the program then runs in 122 clock cycles. Try to find out how to do that.

With the *forwarding* option turned on, most of the `nop` instructions can be removed.

```
             .data
values:      .word   1,2,3,4,5,6,7,8,9,10          ; values to be added
nelem:       .word   10                                    ; array size
sum:         .space  4                       ; sum of the array elements
             .text
             .global  main
main:         addi    r1,r0,nelem     ; r1 = add(nelem)
              lw      r1,0(r1)        ; r1 = val(nelem)
              addi    r2,r0,values    ; r2 = add(values[0])
              addi    r3,r0,0         ; r3 = sum = 0 (partial sum of
                                      ;         the array elements)
              addi    r4,r0,0         ; r4 = i = 0 (counting variable)
loop:         lw      r5,0(r2)        ; r5 = val(values[i])
              nop
              add     r3,r3,r5        ; r3 = sum + val(values[i])
              addi    r4,r4,1         ; r4 = i = i + 1
              addi    r2,r2,4         ; r2 = add(values[i])
              sub     r7,r1,r4        ; r7 = val(nelem) - i
              bnez    r7,loop         ; are all array elements summed up?
              nop
              nop
              addi    r6,r0,sum       ; r6 = add(sum)
              sw      0(r6),r3        ; save sum of the array elements
              trap 0                  ; end of program
```

It takes 102 clock cycles to run the program. The version where instruction reordering is carried out, takes 92 clock cycles.

Taking into account now branch prediction, one notices that the DLX simulator considers the *branch predictor* alternatives *none* and *static – predict always not taken* as the same. So the alternative *static – predict always not taken* is the default.

Changing the *branch predictor* alternative to *static – predict always taken*, it takes 95 clock cycles to run the program; 85 clock cycles for the instruction reordered version.

One should look carefully to the clock cycle diagram of the execution to understand what is the difference between the two alternatives.