Universidade de Aveiro

Mestrado Integrado em Engenharia de Computadores e Telemática

Arquitectura de Computadores Avançada

DLX – Pipelining 2

Academic year 2020/2021 Adaptation of exercise guide by Nuno Lau/José Luís Azevedo

4. Write a program that orders the values of an integer array stored in memory from the largest value to the smallest one.

   Original code

```
              .data
values:       .word   1,2,3,4,5,6,7,8,9,10          ; values to be ordered
nelem:        .word   10                                      ; array size
              .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])
               add     r3,r0,r0          ; r3 = i = 0 (counting variable)
               addi    r8,r1,-1          ; r8 = nelem - 1
loop1:         slt     r9,r3,r8          ; r9 = (i < nelem - 1)
               beqz    r9,end            ; is the end of operations been
                                         ; reached?
               addi    r6,r2,4           ; r6 = add(values[j])
               lw      r4,0(r2)          ; r4 = val(values[i])
               addi    r5,r3,1           ; r5 = j = i+1 (counting variable)
loop2:         lw      r7,0(r6)          ; r7 = val(values[j])
               slt     r9,r4,r7          ; r9 =
                                         ;(val(values[i]) < val(values[j]))
               beqz    r9,goon           ; no element swap is required?
               add     r9,r4,r0          ; r9 = tmp = val(values[i])
               add     r4,r7,r0          ; val(values[i]) = val(values[j])
               add     r7,r9,r0          ; val(values[j]) = tmp
               sw      0(r2),r4          ; val(values[i]) = r4
               sw      0(r6),r7          ; val(values[j]) = r7
goon:          addi    r5,r5,1           ; j = j + 1
               addi    r6,r6,4           ; r6 = add(values[j])
               slt     r9,r5,r1          ; r9 = (j < nelem)
               bnez    r9,loop2          ; are still elements to be
                                         ; compared?
               addi    r3,r3,1           ; i = i + 1
               addi    r2,r2,4           ; r2 = add(values[i])
               j       loop1             ; check for end of operations
end:           trap    0                 ; end of program
```

   4.1. Run the code in the DLX simulator with the *forwarding* option turned off and, through the interspersing of `nop` instructions to take care of the hazards, make it execute correctly. What is the number of clock cycles for the full execution? What is the speed up attained relatively to the original code being run in a non-pipelined processor with a clock cycle time five times longer? Take into consideration that in the latter case not all instructions take the same time to execute.

   For comparing purposes, one assumes that the non-pipelined processor executes `store` and `branch/jump` instructions in 4 equivalent clock cycles and all other instructions in 5 equivalent clock cycles of the pipelined processor.

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 ordered
nelem:      .word   10                                    ; array size
            .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])
             add     r3,r0,r0        ; r3 = i = 0 (counting variable)
             addi    r8,r1,-1        ; r8 = nelem - 1
             nop
             nop
loop1:       slt     r9,r3,r8        ; r9 = (i < nelem - 1)
             nop
             nop
             beqz    r9,end          ; is the end of operations been
                                     ; reached?
             nop
             nop
             nop
             addi    r6,r2,4         ; r6 = add(values[j])
             lw      r4,0(r2)        ; r4 = val(values[i])
             addi    r5,r3,1         ; r5 = j = i+1 (counting variable)
loop2:       lw      r7,0(r6)        ; r7 = val(values[j])
             nop
             nop
             slt     r9,r4,r7        ; r9 =
                                     ;(val(values[i]) < val(values[j]))
             nop
             nop
             beqz    r9,goon         ; no element swap is required?
             nop
             nop
             nop
             add     r9,r4,r0        ; r9 = tmp = val(values[i])
             add     r4,r7,r0        ; val(values[i]) = val(values[j])
             add     r7,r9,r0        ; val(values[j]) = tmp
             nop
             sw      0(r2),r4        ; val(values[i]) = r4
             nop
             sw      0(r6),r7        ; val(values[j]) = r7
goon:        addi    r5,r5,1         ; j = j + 1
             nop
             addi    r6,r6,4         ; r6 = add(values[j])
             slt     r9,r5,r1        ; r9 = (j < nelem)
             nop
             nop
             bnez    r9,loop2        ; are still elements to be
                                     ; compared?
             nop
             nop
             nop
             addi    r3,r3,1         ; i = i + 1
             addi    r2,r2,4         ; r2 = add(values[i])
             j       loop1           ; check for end of operations
             nop
             nop
             nop
end:         trap    0               ; end of program
```

Putting aside the `nop` instructions, 620 instructions are executed in 1380 clock cycles

$$\text{instruction count} = 5 + 8 \cdot 9 + 12 \cdot \sum_{i=1}^{9} i + 3 =$$

$$= 5 + 72 + 540 + 3 = 620$$

$$\text{instruction count}_{store} = 2 \cdot \sum_{i=1}^{9} i = 90$$

$$\text{instruction count}_{br\,/\,jmp} = 2 \cdot 9 + 2 \cdot \sum_{i=1}^{9} i + 1 =$$

$$= 18 + 90 + 1 = 109 \quad .$$

Thus, the equivalent number of clock cycles for the non-pipelined processor is

$$\text{clock cycles}_{nonpipe} = \left(\text{instruction count}_{store} + \text{instruction count}_{br\,/\,jmp}\right) \cdot 4 +$$

$$+ \text{instruction count}_{other} \cdot 5 =$$

$$= 199 \cdot 4 + 421 \cdot 5 = 2901$$

and the speed up is given by

$$\text{speed up} = \frac{\text{clock cycles}_{nonpipe}}{\text{clock cycles}_{pipe}} = \frac{2901}{1380} = 2{,}10 \quad .$$

4.2. Turn the *forwarding* option on and discard the `nop` instructions that are no longer required. What was the speed up now attained?

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 ordered
nelem:       .word   10                                     ; array size
             .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])
              add     r3,r0,r0         ; r3 = i = 0 (counting variable)
              addi    r8,r1,-1         ; r8 = nelem - 1
loop1:        slt     r9,r3,r8         ; r9 = (i < nelem - 1)
              beqz    r9,end           ; is the end of operations been
                                       ; reached?
              nop
              nop
              addi    r6,r2,4          ; r6 = add(values[j])
              lw      r4,0(r2)         ; r4 = val(values[i])
              addi    r5,r3,1          ; r5 = j = i+1 (counting variable)
loop2:        lw      r7,0(r6)         ; r7 = val(values[j])
              nop
              slt     r9,r4,r7         ; r9 =
                                       ;(val(values[i]) < val(values[j]))
              beqz    r9,goon          ; no element swap is required?
              nop
              nop
              add     r9,r4,r0         ; r9 = tmp = val(values[i])
              add     r4,r7,r0         ; val(values[i]) = val(values[j])
              add     r7,r9,r0         ; val(values[j]) = tmp
              sw      0(r2),r4         ; val(values[i]) = r4
              sw      0(r6),r7         ; val(values[j]) = r7
goon:         addi    r5,r5,1          ; j = j + 1
              addi    r6,r6,4          ; r6 = add(values[j])
              slt     r9,r5,r1         ; r9 = (j < nelem)
              bnez    r9,loop2         ; are still elements to be
                                       ; compared?
              nop
              nop
              addi    r3,r3,1          ; i = i + 1
              addi    r2,r2,4          ; r2 = add(values[i])
              j       loop1            ; check for end of operations
              nop
              nop
end:          trap    0                ; end of program
```

Taking into account now branch prediction, one has for *branch predictor* alternatives *none* or *static – predict always not taken*, which are the same for the DLX simulator, 620 instructions executed in 887 clock cycles yielding a speed up of

$$\text{speed up} = \frac{\text{clock cycles}_{nonpipe}}{\text{clock cycles}_{pipe\text{-}nottaken}} = \frac{2901}{887} = 3{,}27 \ .$$

On the other hand, for the *branch predictor* alternative *static – predict always taken*, there is no difference in this case.

4.3. Turn on the MIPS compatibility mode, in order to enable *pipeline interlocking*, and discard the remaining `nop` instructions. Check that your initial code now runs correctly. Go through the clock cycle diagram to understand why. Explain what has happened in your own words. What is the speed up now attained?

With the MIPS compatibility mode now turned on, not only all the `nop` instructions can be removed, but also one can take advantage of the branch delay slot feature.

```
            .data
values:     .word   1,2,3,4,5,6,7,8,9,10        ; values to be ordered
nelem:      .word   10                                    ; array size
            .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])
             add     r3,r0,r0         ; r3 = i = 0 (counting variable)
             addi    r8,r1,-1         ; r8 = nelem - 1
loop1:       slt     r9,r3,r8         ; r9 = (i < nelem - 1)
             beqz    r9,end           ; is the end of operations been
                                      ; reached?
             addi    r6,r2,4          ; r6 = add(values[j])
             lw      r4,0(r2)         ; r4 = val(values[i])
             addi    r5,r3,1          ; r5 = j = i+1 (counting variable)
             lw      r7,0(r6)         ; r7 = val(values[j])
loop2:       slt     r9,r4,r7         ; r9 =
                                      ;(val(values[i]) < val(values[j]))
             beqz    r9,goon          ; no element swap is required?
             add     r9,r4,r0         ; r9 = tmp = val(values[i])
             add     r4,r7,r0         ; val(values[i]) = val(values[j])
             add     r7,r9,r0         ; val(values[j]) = tmp
             sw      0(r2),r4         ; val(values[i]) = r4
             sw      0(r6),r7         ; val(values[j]) = r7
goon:        addi    r5,r5,1          ; j = j + 1
             addi    r6,r6,4          ; r6 = add(values[j])
             slt     r9,r5,r1         ; r9 = (j < nelem)
             bnez    r9,loop2         ; are still elements to be
                                      ; compared?
             lw      r7,0(r6)         ; r7 = val(values[j])
             addi    r3,r3,1          ; i = i + 1
             j       loop1            ; check for end of operations
             addi    r2,r2,4          ; r2 = add(values[i])
end:         trap 0                   ; end of program
```

4.4. Consider the *branch predictor* alternatives: *none*, *static – predict always not taken* and *static – predict always taken*. Which one produces the most efficient run? Why? What is the speed up now attained?

In this case, both alternatives *static – predict always not taken* and *static – predict always taken* produce the same result: 620 instructions are executed in 689 clock cycles, yielding a speed up of

$$\text{speed up} = \frac{\text{clock cycles}_{nonpipe}}{\text{clock cycles}_{pipe}} = \frac{2901}{689} = 4{,}21 \ .$$