CLIL Pre-reading: Before reading the lecture, try to give a definition of the following terms:

1. Instruction reordering
2. Loop unrolling

06 Performance optimization

06.01 Static code optimization

Static code optimization aims at reducing the execution time of a code segment by applying functionally-equivalent transformations that make the code more suitable to be executed on the target architecture.

The key idea behind static code optimization is to reduce dependencies and conflicts among adjacent instructions that may cause pipeline stalls.

There are two main types of transformations: *instruction reordering* and *loop unrolling*. A third type of transformation, called *register renaming*, is used to increase the degrees of freedom available for instruction reordering. The three transformations will be explained in the following section by means of a simple example simulated by WinDLX.

Static code optimization can be manually performed for teaching purposely, but it is usually performed at compile time.

06.02 Example of static code optimization supported by WinDLX

WindDLX can be used to highlight pipeline stalls, to drive static code optimization, and to evaluate the impact of transformations made to optimize a given code segment. For explanation purposes in this section we assume the following configuration of WinDLX:

FP Add	1 unit with latency 3
Data forwarding	enabled

A. Example

We consider the following algorithm, that adds a constant (inc) to all the elements of an array (vett):

```
for (i=8; i>0; i--)
    vett[i] = vett[i] + inc;
```

To execute the algorithm on WinDLX write the following code in a text file called Loop0.s

```
          .global     main
main:
Loop:
          LD    f0, 0(r1)
          ADDD  f0, f0, f2
          SD    0(r1), f0
          SUBI  r1, r1, #8
          BNEZ  r1, Loop
          trap  0
```

r1 is the register used as loop index, f2 is the double-precision floating point register used to store the constant to be added to the array elements, f0 is the internal register used to load the array element to be incremented.

Notice that r1 and f2 are not initialized. This would be an error in a real-world program, but it is possible in this context since WindLX allows us to manually set the value of the internal registers before starting execution. In particular, r1 needs to be initialized to decimal value 64 in order for the algorithm to visit an array of 8 elements (of 8 bytes each).

Running the algorithm we observe that execution takes 84 clock cycles and that several clock cycles are lost because of stalls and branches.

To solve the same problem of adding a constant to all the elements of an array we can use the following code, which is more general and contains the specification of the data (in main memory) to be used as inputs. The manual initialization of registers in WinDLX is no longer required. Let's call the new algorithm Loop1.s.

```
    .data
vett:    .double 4.3, 2.1, 3.3, 5.2, 4.8, 9.3, 2.3, 0.5
n: .word 8
inc:     .double 3.2

    .text
    .global     main
main:
    LW    r1, n(r0)          ; number of elements
    LD    f2, inc(r0)        ; value of the increment
    ADDI  r2, r0, vett       ; pointer to the first array element array
loop:
    LD    f0, 0(r2)          ; read an element
    ADDD  f0, f0, f2         ; increment it
    SD    0(r2), f0          ; write it back in memory
    ADDI  r2, r2, #8         ; move to next element
    SUBI  r1, r1, #1         ; decrement the counter
    BNEZ  r1, loop           ; jump if not equal zero
end:
    trap  0
```

The code segment corresponds to the following pseudo-code and is executed in 95 clock cycles:

```
for (i=0; i<8; i++)
    vett[i] = vett[i] + inc;
```

A. Instruction reordering

Now change the code as follows and save it in a new file called Loop2.s. For the sake of conciseness only the main loop is reported, since the remaining rows are unchanged:

```
loop:
    LD    f0, 0(r2)          ; read an element
    ADDD  f0, f0, f2         ; increment the element
    SUBI  r1, r1, #1         ; decrement the counter
    SD    0(r2), f0          ; write it back in memory
    ADDI  r2, r2, #8         ; move to next element
    BNEZ  r1, loop           ; jump if not equal zero
end:
    trap  0
```

The only change from Loop1.s to Loop2.s is that SUBI instruction that decrements the loop counter has been placed between ADDD and SD. By executing Loop2.s we obtain CPUC=79, which is much lower than 95. In fact, by changing the position of SUBI we obtained a two-fold benefit (separating ADDD from SD and separating SUBI from BNEZ), thus avoiding 2 stall cycles per loop iteration (2x8=16 clock cycles).

This is an example of instruction reordering, that can be further applied by anticipating the increment of r2, which is a pointer to the next element of the array. This is done in Loop3.s:

```
loop:
    LD    f0, 0(r2)          ; read an element
    ADDI  r2, r2, #8         ; move to next element
    ADDD  f0, f0, f2         ; increment the element
    SUBI  r1, r1, #1         ; decrement the counter
    SD    -8(r2), f0         ; write it back in memory
    BNEZ  r1, loop           ; jump if not equal zero
end:
    trap  0
```

There are only two changes from Loop2.s to Loop3.s. First, the ADDI instruction that increments the pointer has been placed between LD and ADDD. Second, the SD instruction has been modified adding an offset (-8) to compensate for the anticipated increment of r2.

Execution time reduces to 71 clock cycles, since the increment of r2 is now computed at each iteration in place of a stall cycle. Hence, one clock cycle is saved for each iteration.

C. Loop unrolling

CLIL While-reading: Read part C and write a definition of *loop unrolling*. Then, check whether the definition of *loop unrolling* you provided prior to reading the lecture was correct.

Assume we have a loop to be executed 4 times.

```
for (i=0; i<4; i++)
    vett[i] = vett[i] + inc;
```

Loop unrolling consists of unrolling the loop by repeating the loop body as many times as needed:

```
    i = 0;
    vett[i] = vett[i] + inc;
    i++;
    vett[i] = vett[i] + inc;
    i++;
    vett[i] = vett[i] + inc;
    i++;
    vett[i] = vett[i] + inc;
```

Needless to say, this is possible only if we know at compile-time the number of times the loop has to be executed. In assembly, the unrolled loop appears as follows (Loop4.s):

```
element0:
    LD    f0, 0(r2)         ; read an element
    ADDI  r2, r2, #8        ; move to next element
    ADDD  f0, f0, f2        ; increment it
    SUBI  r1, r1, #1        ; decrement the counter
    SD    -8(r2), f0        ; write it back in memory
element1:
    LD    f0, 0(r2)         ; read an element
    ADDI  r2, r2, #8        ; move to next element
    ADDD  f0, f0, f2        ; increment it
    SUBI  r1, r1, #1        ; decrement the counter
    SD    -8(r2), f0        ; write it back in memory
element2:
    LD    f0, 0(r2)         ; read an element
    ADDI  r2, r2, #8        ; move to next element
    ADDD  f0, f0, f2        ; increment it
    SUBI  r1, r1, #1        ; decrement the counter
    SD    -8(r2), f0        ; write it back in memory
element3:
    LD    f0, 0(r2)         ; read an element
    ADDI  r2, r2, #8        ; move to next element
    ADDD  f0, f0, f2        ; increment it
    SUBI  r1, r1, #1        ; decrement the counter
    SD    -8(r2), f0        ; write it back in memory
end:
    trap  0
```

If we compare the execution of the Loop3.s (with n = 4) and Loop4.s, we find a small advantage (from 39 to 32 clock cycles) due to the removal of all branches. However, further advantages can be achieved by observing that the r1 is not required any more and that r2 doesn't need to be explicitly incremented 4 times. A simpler equivalent implementation is provided by the following code (Loop5.s), which corresponds to

```
    vett[0] = vett[0] + inc;
    vett[1] = vett[1] + inc;
    vett[2] = vett[2] + inc;
    vett[3] = vett[3] + inc;
```

```
    .data
vett:    .double 4.3, 2.1, 3.3, 5.2
n:       .word 4
inc:     .double 3.2

    .text
    .global     main
main:
    LD    f2, inc(r0)       ; value of the increment
    ADDI  r2, r0, vett      ; pointer to the first array element array
element0:
    LD    f0, 0(r2)         ; read an element
    ADDD  f0, f0, f2        ; increment it
    SD    0(r2), f0         ; write it back in memory
element1:
    LD    f0, 8(r2)         ; read an element
    ADDD  f0, f0, f2        ; increment it
    SD    8(r2), f0         ; write it back in memory
element2:
    LD    f0, 16(r2)        ; read an element
    ADDD  f0, f0, f2        ; increment it
    SD    16(r2), f0        ; write it back in memory
element3:
    LD    f0, 24(r2)        ; read an element
    ADDD  f0, f0, f2        ; increment it
    SD    24(r2), f0        ; write it back in memory
end:
    trap  0
```

In spite of the great simplification of the code, Loop5.s saves only 1 clock cycle with respect to Loop4.s. This is due to the lack of integer independent instructions to be placed between data-dependent floating point instructions.

However, loop unrolling provides also new opportunities for instruction reordering. In fact, the unrolled replicas of the loop body are usually almost independent from each other, thus enabling instruction reordering.

In the previous code segment (Loop5.s), however, instruction reordering is limited by write-after-write problems caused by the reuse of register f0 to load all elements of the array. Let's rewrite Loop4.s using different registers for the elements of the array (Loop6.s):

```
element0:
    LD    f0, 0(r2)         ; read an element
    ADDD  f0, f0, f2        ; incremet it
    SD    0(r2), f0         ; write it back in memory
element1:
    LD    f4, 8(r2)         ; read an element
    ADDD  f4, f4, f2        ; incremet it
    SD    8(r2), f4         ; write it back in memory
element2:
    LD    f6, 16(r2)        ; read an element
    ADDD  f6, f6, f2        ; incremet it
    SD    16(r2), f6        ; write it back in memory
element3:
    LD    f8, 24(r2)        ; read an element
    ADDD  f8, f8, f2        ; incremet it
    SD    24(r2), f8        ; write it back in memory
end:
    trap  0
```

Now the replicas of the loop body are completely independent from each other (thanks to **register renaming**) and can be interleaved to reduce the number of data hazards (Loop7.s):

```
elements:
      LD    f0, 0(r2)         ; read an element
      LD    f4, 8(r2)         ; read an element
      LD    f6, 16(r2)        ; read an element
      LD    f8, 24(r2)        ; read an element
      ADDD  f0, f0, f2        ; incremet it
      ADDD  f4, f4, f2        ; incremet it
      ADDD  f6, f6, f2        ; incremet it
      ADDD  f8, f8, f2        ; incremet it
      SD    0(r2), f0         ; write it back in memory
      SD    8(r2), f4         ; write it back in memory
      SD    16(r2), f6        ; write it back in memory
      SD    24(r2), f8        ; write it back in memory
end:
      trap  0
```

The new code executes in 26 clock cycles. The main residual drawback of the new implementation is the structural hazard caused by the violation of the repetition time of ADDD. To solve the problem a different reordering may be conceived (***students are invited to find such a solution***), or additional FP adders can be added to the architecture. If we change the configuration assuming there are three FP adders, the execution time reduces to 21 clock cycles.

D. Partial loop unrolling

CLIL While-reading: Read part D and write a definition of *partial loop unrolling*

In many cases, the number of iterations of a loop is data dependent and unknown at compile time, making it impossible to completely unroll the loop. The same problem occurs when the number of iterations is too large to fully unroll the loop.

In both cases, partial loop unrolling can be suitably applied. Consider for instance our original loop which spans an array of N=8 elements:

```
for (i=0; i<N; i++)
  vett[i] = vett[i] + inc;
```

In general, however, any loop can be at least partially unrolled. Assume, for instance, we know that the number of elements (N) is a multiple of 4. In this case we can unroll the loop as foolows:

```
for (i=0; i<N; i+=4) {
  vett[i] = vett[i] + inc;
  vett[i+1] = vett[i+1] + inc;
  vett[i+2] = vett[i+2] + inc;
  vett[i+3] = vett[i+3] + inc;
}
```

This solution is implemented in Loop8.s (reported below) which executes in 59 clock cycles to be compared with the 71 of version Loop3.s.

```
   .data
vett:   .double 4.3, 2.1, 3.3, 5.2, 4.8, 9.3, 2.3, 0.5
n: .word 8
inc:    .double 3.2

   .text
   .global     main
main:
   LW    r1, n(r0)         ; number of elements
   LD    f2, inc(r0)       ; value of the increment
   ADDI  r2, r0, vett      ; pointer to the first array element array
loop:
   LD    f0, 0(r2)         ; read an element
   ADDI  r2, r2, #8        ; move to next element
   ADDD  f0, f0, f2        ; incremet it
   SUBI  r1, r1, #1        ; decrement the counter
   SD    -8(r2), f0        ; write it back in memory
element1:
   LD    f0, 0(r2)         ; read an element
   ADDI  r2, r2, #8        ; move to next element
```

```
   ADDD  f0, f0, f2        ; incremet it
   SUBI  r1, r1, #1        ; decrement the counter
   SD    -8(r2), f0        ; write it back in memory
element2:
   LD    f0, 0(r2)         ; read an element
   ADDI  r2, r2, #8        ; move to next element
   ADDD  f0, f0, f2        ; incremet it
   SUBI  r1, r1, #1        ; decrement the counter
   SD    -8(r2), f0        ; write it back in memory
element3:
   LD    f0, 0(r2)         ; read an element
   ADDI  r2, r2, #8        ; move to next element
   ADDD  f0, f0, f2        ; incremet it
   SUBI  r1, r1, #1        ; decrement the counter
   SD    -8(r2), f0        ; write it back in memory
   BNEZ r1, loop
end:
   trap  0
```

Further improvements can be achieved by reducing the number of increments and decrements within each loop (Loop9.s):

```
loop:
   LD    f0, 0(r2)         ; read an element
   ADDD  f0, f0, f2        ; incremet it
   SD    0(r2), f0         ; write it back in memory
element1:
   LD    f0, 8(r2)         ; read an element
   ADDD  f0, f0, f2        ; incremet it
   SD    8(r2), f0         ; write it back in memory
element2:
   LD    f0, 16(r2)        ; read an element
   ADDD  f0, f0, f2        ; incremet it
   SD    16(r2), f0        ; write it back in memory
element3:
   LD    f0, 24(r2)        ; read an element
   ADDD  f0, f0, f2        ; incremet it
   SD    24(r2), f0        ; write it back in memory
   SUBI  r1, r1, #4        ; decrement the counter by 4
   ADDI  r2, r2, #32       ; move to next 4 elements
   BNEZ r1, loop
end:
   trap  0
```

and by reordering the instructions within the loop (Loop10.s):

```
loop:
   LD    f0, 0(r2)         ; read an element
   LD    f4, 8(r2)         ; read an element
   LD    f6, 16(r2)        ; read an element
   LD    f8, 24(r2)        ; read an element
   ADDD  f0, f0, f2        ; increment it
   ADDD  f4, f4, f2        ; increment it
   ADDD  f6, f6, f2        ; increment it
   ADDD  f8, f8, f2        ; increment it
   SD    0(r2), f0         ; write it back in memory
   SD    8(r2), f4         ; write it back in memory
   SD    16(r2), f6        ; write it back in memory
   SD    24(r2), f8        ; write it back in memory
   SUBI  r1, r1, #4        ; decrement the counter by 4
   ADDI  r2, r2, #32       ; move to next 4 elements
   BNEZ r1, loop
end:
   trap  0
```

Version Loop9.s doesn't provide any benefit in terms of CPUC, since the benefits of the removed instructions are masked by pipeline stalls. Loop10.s reduced the execution time to 43 clock cycles (in case of 3 FP units).

Unfortunately, the partial loop unrolling described so far is not general enough. In fact, it fails when the number of elements is not a multiple of 4. To fully generalize the approach we need to adopt the following solution:

```
for (i=0; i<=N-4; i+=4) {
  x[i]=x[i] + s;
  x[i+1]=x[i+1] + s;
  x[i+2]=x[i+2] + s;
  x[i+3]=x[i+3] + s;
}
for (j=i; j<N; j++) {
  x[j] = x[j] + s;
}
```

Which is implemented in Loop11.s:

```
loop:
   LD    f0, 0(r2)          ; read an element
   LD    f4, 8(r2)          ; read an element
   LD    f6, 16(r2)         ; read an element
   LD    f8, 24(r2)         ; read an element
   ADDD  f0, f0, f2         ; increment it
   ADDD  f4, f4, f2         ; increment it
   ADDD  f6, f6, f2         ; increment it
   ADDD  f8, f8, f2         ; increment it
   SD    0(r2), f0          ; write it back in memory
   SD    8(r2), f4          ; write it back in memory
   SD    16(r2), f6         ; write it back in memory
   SD    24(r2), f8         ; write it back in memory
   SUBI  r1, r1, #4         ; decrement the counter by 4
   ADDI  r2, r2, #32        ; move to next 4 elements
   SGEI  r3, r1, #4         ; check if r1 >= 4 and set flag r3
   BNEZ  r3, loop           ; if it is (r3=1) stay in the loop
   BEQZ  r1, end            ; if no more elements go to end
loop1:
   LD    f0, 0(r2)
   SUBI  r1, r1, #1
   ADDD  f0, f0, f2
   ADDI  r2, r2, #8
   SD    -8(r2), f0
   BNEZ  r1, loop1
end:
   trap  0
```

Loop11.s takes 48 clock cycles to execute when n=8, but it works properly also for n=9 or for any other value of n. Notice that for n=7 it is less efficient than for n=8, taking 51 rather than 48 clock cycles!

CLIL After reading: After reading the whole lecture, look at the **reference devices** (personal pronouns, demonstratives, etc.) highlighted in the text and find the word/words they refer to. For example:
"Now change the code as follows and save it in a new file called Loop2.c" → IT refers back to the word CODE

CLIL After reading – Lexis:

to optimize = to make something as good as possible
a given code = an already decided, arranged or agreed code
the following = the one or the ones about to be mentioned
to increment = to increase
to decrement = to decrease
a two-fold benefit = a double benefit
removal = taking something away from the place occupied
further (advantages) = additional (advantages)
to interleave = to place at intervals in or among / to intersperse one after the other / to distribute among other things at intervals
to iterate = to repeat a process
iterations = repeated processes

CLIL After reading - Conjunctions and adverbs in the text:

among = surrounded by / in the middle of / in the group / in a mass
behind = to the rear of
manually = which is done with hands
as follows = in the way which is about to be mentioned

CLIL After reading - Grammar notes:

1. to - in order to – so as to – in order that – so that - in order for something/somebody to

- The infinitive with 'to', that is a to-infinitive clause, can be used to express a purpose:
  One needs to decode the instruction to perceive (and handle) the conflict.

- One can use also 'in order to' to express a purpose:
  One needs to decode the instruction in order to perceive (and handle) the conflict.

- 'So as to' can also be used to express a purpose:
  One needs to decode the instruction so as to perceive (and handle) the conflict.

- 'In order not to' and 'so as not to' are used in negative sentences.

- 'So that' and 'in order that' (+ present tense with a future meaning) can be used to express a purpose:
  The execution stages of pipelined instructions are misaligned so that the first one starts executing before the second one.

  The execution stages of pipelined instructions are misaligned in order that the first one starts executing before the second one.

  'in order that' (more formal) and 'so that' followed by 'should' or 'could' or 'would' can be used to express a purpose in past sentences:
  The execution stages of pipelined instructions were misaligned in order that/so that the first one would start executing before the second one.

- 'in order for something/somebody to' can also be used to express a purpose:
  In particular, r1 needs to be initialized to decimal value 64 in order for the algorithm to visit an array of 8 elements (of 8 bytes each).

I. Exercise – Match column A with column B
(keys are provided below)

Column A	**Column B**
1. The limited number of instructions in the instruction set is exploited to simplify the encoding as much as possible in order to	a. avoid executing the wrong instruction.
2. Due to the high number of different instructions, CISC architectures make use of different sizes and formats to encode the instructions, so as to	b. capture the effects of pipeline hazards, we need a performance model that associates to each instruction of the instruction set at least three parameters.
3. As soon as the branch is detected, the fetched instruction is aborted in order to	c. minimize the average size of the encoding at the cost of making it more complex to fetch and decode the instructions.
4. To	d. speed up decoding and simplify the control logic.

Answer keys:

1d 2c 3a 4b

1. The limited number of instructions in the instruction set is exploited to simplify the encoding as much as possible in order to speed up decoding and simplify the control logic.

2. Due to the high number of different instructions, CISC architectures make use of different sizes and formats to encode the instructions, so as to minimize the average size of the encoding at the cost of making it more complex to fetch and decode the instructions.

3. As soon as the branch is detected, the fetched instruction is aborted in order to avoid executing the wrong instruction.

4. To capture the effects of pipeline hazards, we need a performance model that associates to each instruction of the instruction set at least three parameters