Lab. 4: “STATIC INSTRUCTIONS SCHEDULING”

Computer Architecture and Engineering (3^{rd} year)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Goals:

- Know, understand and apply some static scheduling techniques.

Assignment:

The simulator of the RISC-V computer with multicycle instructions

The simulator **riscv-m** enables the execution of programs written using RISC-V assembler. It supports a the RISC-V rv64imfd instruction set.

The simulated processor does not integrate dynamic instruction scheduling. Data hazards are solved by inserting stalls or applying forwarding, with stalls introduced when necessary. Control hazards can be solved by inserting stalls or using *predict-not-taken*, with various branch latencies. For multicycle instructions, there is a load/store unit, a multiplier, an adder and a comparison unit. All of them are pipelined and their latency can be configured.

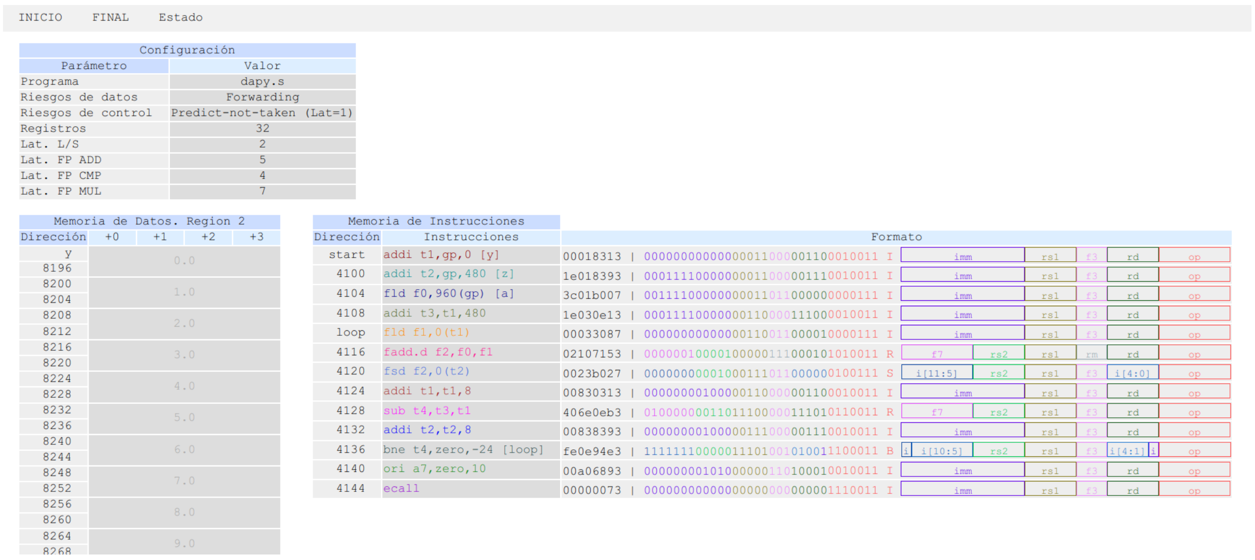
The simulator accepts various parameters. You can check the complete list of parameters by executing:

```
riscv-m -?
```

Example of a RISC-V program

Consider the assembler code of a program implementing a loop that adds a scalar (double) value to all the entries of an array stored in memory ($\vec{Z} = a + \vec{Y}$, DAPY).

```
# z = a + y
    .data
# vector y
# vector size: 60 elements
y:  .double  0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0
    .double 10.0,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0
    .double 20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0,28.0,29.0
    .double 30.0,31.0,32.0,33.0,34.0,35.0,36.0,37.0,38.0,39.0
    .double 40.0,41.0,42.0,43.0,44.0,45.0,46.0,47.0,48.0,49.0
    .double 50.0,51.0,52.0,53.0,54.0,55.0,56.0,57.0,58.0,59.0

# vector z
# 60 elements are 480 bytes
z:  .space 480

# scalar a
```



Configuración	
Parámetro	Valor
Programa	dapy.s
Riesgos de datos	Forwarding
Riesgos de control	Predict-not-taken (Lat=1)
Registros	32
Lat. L/S	2
Lat. FP ADD	5
Lat. FP CMP	4
Lat. FP MUL	7

Memoria de Datos. Region 2				
Dirección	+0	+1	+2	+3
y	0.0			
8196				
8200	1.0			
8204				
8208	2.0			
8212				
8216	3.0			
8220				
8224	4.0			
8228				
8232	5.0			
8236				
8240	6.0			
8244				
8248	7.0			
8252				
8256	8.0			
8260				
8264	9.0			
8268				

Memoria de Instrucciones			
Dirección	Instrucciones	Formato	
start	addi t1,gp,0 [y]	00018313 | 00000000000000011000001100010011 I	imm rs1 f3 rd op
4100	addi t2,gp,480 [z]	1e018393 | 00011110000000011000001110010011 I	imm rs1 f3 rd op
4104	fld f0,960(gp) [a]	3c01b007 | 00111100000000011011000000000111 I	imm rs1 f3 rd op
4108	addi t3,t1,480	1e030e13 | 00011110000000110000111000010011 I	imm rs1 f3 rd op
loop	fld f1,0(t1)	00033087 | 00000000000000110011000010000111 I	imm rs1 f3 rd op
4116	fadd.d f2,f0,f1	02107153 | 00000010000100000111000101010011 R	f7 rs2 rs1 rm rd op
4120	fsd f2,0(t2)	0023b027 | 00000000001000111011000000100111 S	i[11:5] rs2 rs1 f3 i[4:0] op
4124	addi t1,t1,8	00830313 | 00000000100000110000001100010011 I	imm rs1 f3 rd op
4128	sub t4,t3,t1	406e0eb3 | 01000000011011100000111010110011 R	f7 rs2 rs1 f3 rd op
4132	addi t2,t2,8	00838393 | 00000000100000111000001110010011 I	imm rs1 f3 rd op
4136	bne t4,zero,-24 [loop]	fe0e94e3 | 11111110000011101001010011100011 B	i i[10:5] rs2 rs1 f3 i[4:1] i op
4140	ori a7,zero,10	00a06893 | 00000000101000000110100010010011 I	imm rs1 f3 rd op
4144	ecall	00000073 | 00000000000000000000000001110011 I	imm rs1 f3 rd op

Figura 1: Content of the *index.html* file for `dapy.s`

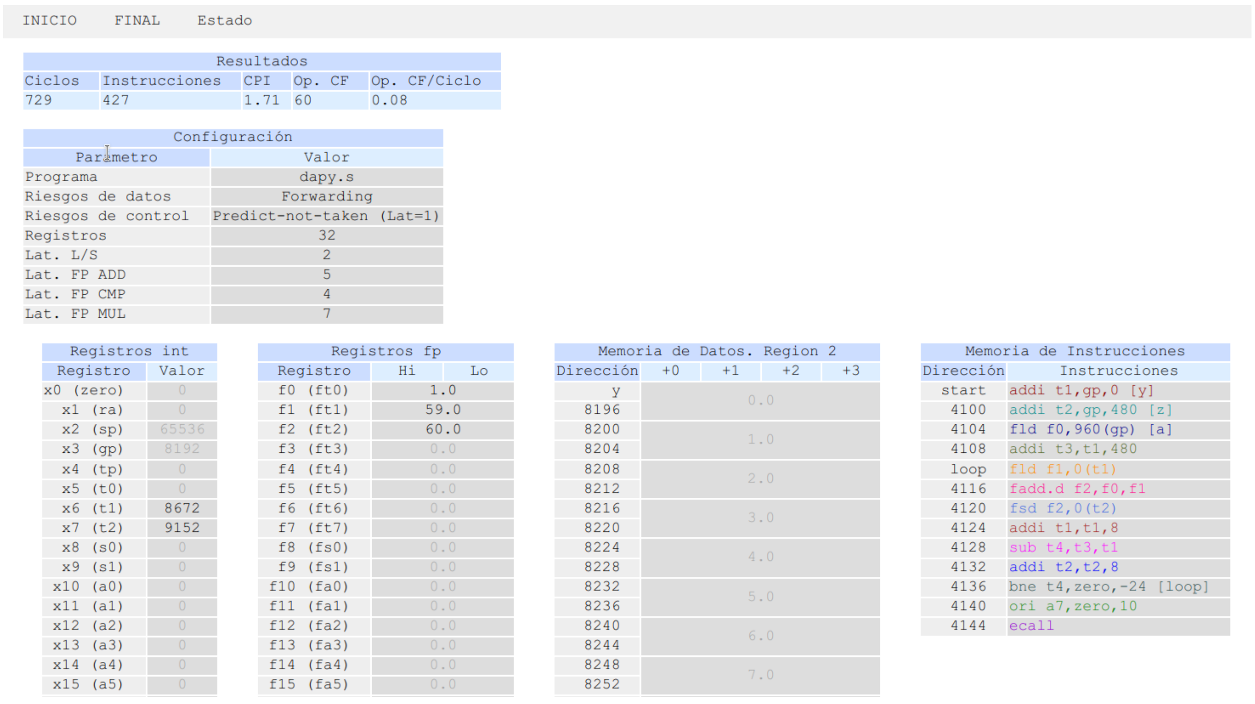
```
a:  .double 1.0

    .text

start:
    addi t1, gp, y        # t1 points to y
    addi t2, gp, z        # t2 points to z
    fld f0, a(gp)         # f0 holds a
    addi t3, t1, 480      # 60 elements are 480 bytes
loop:
    fld f1, 0(t1)
    fadd.d f2, f0, f1
    fsd f2, 0(t2)
    addi t1, t1, 8
    sub t4, t3, t1
    addi t2, t2, 8
    bnez t4, loop

    ori a7, zero, 10      # end
    ecall
```

This program is stored in file `dapy.s`. It can be run to return the result in html files, and handle the data and control hazards via forwarding and *predict-not-taken* respectively, with the following command:

```
riscv-m -d c -c pnt1 -a 5 -f dapy.s
```

Next, inspecting the file index.html with a web browser, we can analyze the configuration of the processor, the initial memory contents, and several links to navigate through the results:

- INICIO. Shows the processor configuration and the initial memory content.


INICIO FINAL Estado

Resultados				
Ciclos	Instrucciones	CPI	Op. CF	Op. CF/Ciclo
729	427	1.71	60	0.08

Configuración	
Parámetro	Valor
Programa	dapy.s
Riesgos de datos	Forwarding
Riesgos de control	Predict-not-taken (Lat=1)
Registros	32
Lat. L/S	2
Lat. FP ADD	5
Lat. FP CMP	4
Lat. FP MUL	7

Registros int	
Registro	Valor
x0 (zero)	0
x1 (ra)	0
x2 (sp)	65536
x3 (gp)	8192
x4 (tp)	0
x5 (t0)	0
x6 (t1)	8672
x7 (t2)	9152
x8 (s0)	0
x9 (s1)	0
x10 (a0)	0
x11 (a1)	0
x12 (a2)	0
x13 (a3)	0
x14 (a4)	0
x15 (a5)	0

Registros fp		
Registro	Hi	Lo
f0 (ft0)	1.0	
f1 (ft1)	59.0	
f2 (ft2)	60.0	
f3 (ft3)	0.0	
f4 (ft4)	0.0	
f5 (ft5)	0.0	
f6 (ft6)	0.0	
f7 (ft7)	0.0	
f8 (fs0)	0.0	
f9 (fs1)	0.0	
f10 (fa0)	0.0	
f11 (fa1)	0.0	
f12 (fa2)	0.0	
f13 (fa3)	0.0	
f14 (fa4)	0.0	
f15 (fa5)	0.0	

Memoria de Datos. Region 2				
Dirección	+0	+1	+2	+3
y	0.0			
8196				
8200	1.0			
8204				
8208	2.0			
8212				
8216	3.0			
8220				
8224	4.0			
8228				
8232	5.0			
8236				
8240	6.0			
8244				
8248	7.0			
8252				

Memoria de Instrucciones	
Dirección	Instrucciones
start	addi t1,gp,0 [y]
4100	addi t2,gp,480 [z]
4104	fld f0,960(gp) [a]
4108	addi t3,t1,480
loop	fld f1,0(t1)
4116	fadd.d f2,f0,f1
4120	fsd f2,0(t2)
4124	addi t1,t1,8
4128	sub t4,t3,t1
4132	addi t2,t2,8
4136	bne t4,zero,-24 [loop]
4140	ori a7,zero,10
4144	ecall

Figura 2: Content of the *final.html* file of `dapy.s`

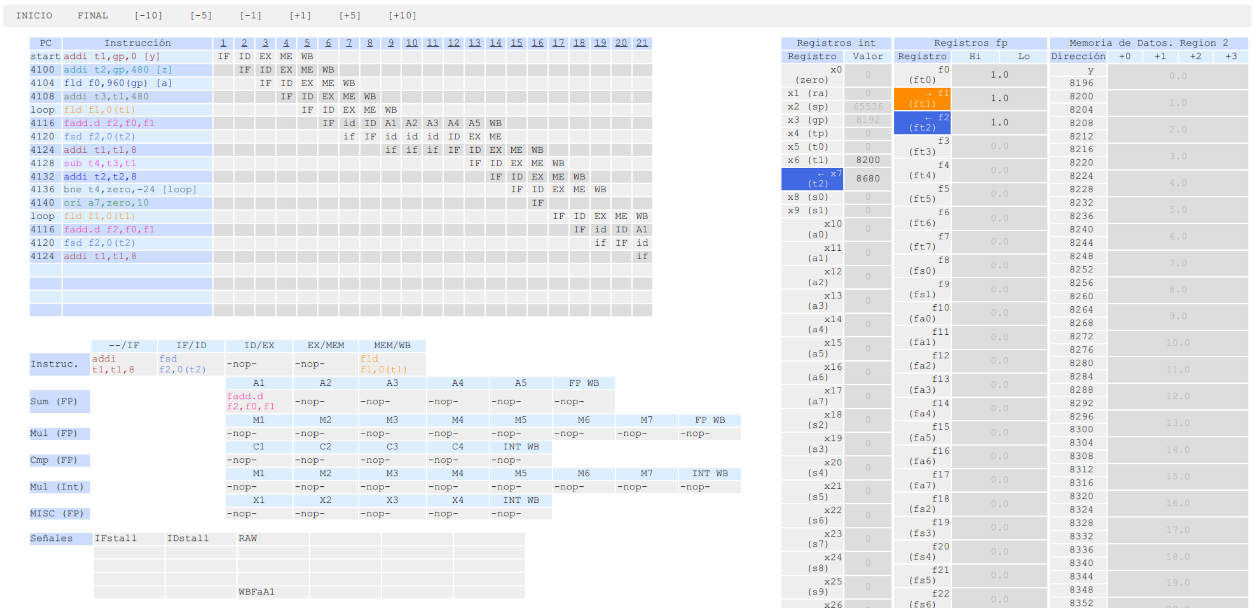
- FINAL. Shows performance results after executing programs, the processor configuration and the final contents of memory. Checking the final memory contents enables verifying the proper execution of the program.

- Estado. Shows the instructions–time diagram that belongs to the program execution and the state of the execution unit in a given cycle, indicating which instruction is hold by each processor stage. Each instruction is shown in a different color. Finally, it shows the contents of the registers and memory at the end of the analyzed cycle. In the case of read or write operations, the corresponding instruction involved is used as a background color in the register or memory position accessed. On this page we have links to the status pages corresponding to 1, 5 or 10 cycles before or after the current one.

Figure 1 displays the contents of the *index.html* file. It contains the size of the register file and the latencies related to the multicycle units. The size of the register file and the latencies of the multicycle units are given first. It also illustrates the initial contents of the data and instruction memories.

Following the link FINAL will open the file *final.html*. Figure 2 shows the content of this file in the case of our example. First, it offers a performance summary for the execution: execution time, number of instructions executed, CPI, floating point operations, and floating point operations per cycle. The configuration of the processor is also given as well as the final contents of the data and instruction memories. The right side of the figure indicates that the resulting array starts at memory location z, thus enabling the verification of the program correctness.

If the link Estado is followed, the simulator will open the file *estadoXXX.html* opened, where XXX represents the execution cycle, starting at “001”. Figure 3 shows its content



INICIO FINAL [-10] [-5] [-1] [+1] [+5] [+10]

PC	Instrucción	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
start	addi t1,gp,0 [y]	IF	ID	EX	ME	WB																
4100	addi t2,gp,480 [z]		IF	ID	EX	ME	WB															
4104	fld f0,960(gp) [a]			IF	ID	EX	ME	WB														
4108	addi t3,t1,480				IF	ID	EX	ME	WB													
loop	fld f1,0(t1)					IF	ID	EX	ME	WB												
4116	fadd.d f2,f0,f1						IF	id	ID	A1	A2	A3	A4	A5	WB							
4120	fsd f2,0(t2)							if	IF	id	id	id	ID	EX	ME							
4124	addi t1,t1,8									if	if	if	IF	ID	EX	ME	WB					
4128	sub t4,t3,t1													IF	ID	EX	ME	WB				
4132	addi t2,t2,8														IF	ID	EX	ME	WB			
4136	bne t4,zero,-24 [loop]															IF	ID	EX	ME	WB		
4140	ori a7,zero,10																IF					
loop	fld f1,0(t1)																	IF	ID	EX	ME	WB
4116	fadd.d f2,f0,f1																		IF	id	ID	A1
4120	fsd f2,0(t2)																			if	IF	id
4124	addi t1,t1,8																					if

	--/IF	IF/ID	ID/EX	EX/MEM	MEM/WB			
Instruc.	addi t1,t1,8	fsd f2,0(t2)	-nop-	-nop-	fld f1,0(t1)			
			A1	A2	A3	A4	A5	FP WB
Sum (FP)			fadd.d f2,f0,f1	-nop-	-nop-	-nop-	-nop-	-nop-

	M1	M2	M3	M4	M5	M6	M7	FP WB
Mul (FP)	-nop-	-nop-	-nop-	-nop-	-nop-	-nop-	-nop-	-nop-
	C1	C2	C3	C4	INT WB			
Cmp (FP)	-nop-	-nop-	-nop-	-nop-	-nop-			
	M1	M2	M3	M4	M5	M6	M7	INT WB
Mul (Int)	-nop-	-nop-	-nop-	-nop-	-nop-	-nop-	-nop-	-nop-
	X1	X2	X3	X4	INT WB			
MISC (FP)	-nop-	-nop-	-nop-	-nop-	-nop-			

Señales	IFstall	IDstall	RAW			
			WBFaA1			

Registros int	
Registro	Valor
x0 (zero)	0
x1 (ra)	0
x2 (sp)	65536
x3 (gp)	8192
x4 (tp)	0
x5 (t0)	0
x6 (t1)	8200
← x7 (t2)	8680
x8 (s0)	0
x9 (s1)	0
x10 (a0)	0
x11 (a1)	0
x12 (a2)	0
x13 (a3)	0
x14 (a4)	0
x15 (a5)	0
x16 (a6)	0
x17 (a7)	0
x18 (s2)	0
x19 (s3)	0
x20 (s4)	0
x21 (s5)	0
x22 (s6)	0
x23 (s7)	0
x24 (s8)	0
x25 (s9)	0
x26	

Registros fp		
Registro	Hi	Lo
f0 (ft0)	1.0	
→ f1 (ft1)	1.0	
← f2 (ft2)	1.0	
f3 (ft3)	0.0	
f4 (ft4)	0.0	
f5 (ft5)	0.0	
f6 (ft6)	0.0	
f7 (ft7)	0.0	
f8 (fs0)	0.0	
f9 (fs1)	0.0	
f10 (fa0)	0.0	
f11 (fa1)	0.0	
f12 (fa2)	0.0	
f13 (fa3)	0.0	
f14 (fa4)	0.0	
f15 (fa5)	0.0	
f16 (fa6)	0.0	
f17 (fa7)	0.0	
f18 (fs2)	0.0	
f19 (fs3)	0.0	
f20 (fs4)	0.0	
f21 (fs5)	0.0	
f22 (fs6)	0.0	

Memoria de Datos. Region 2				
Dirección	+0	+1	+2	+3
y	0.0			
8196				
8200	1.0			
8204				
8208	2.0			
8212				
8216	3.0			
8220				
8224	4.0			
8228				
8232	5.0			
8236				
8240	6.0			
8244				
8248	7.0			
8252				
8256	8.0			
8260				
8264	9.0			
8268				
8272	10.0			
8276				
8280	11.0			
8284				
8288	12.0			
8292				
8296	13.0			
8300				
8304	14.0			
8308				
8312	15.0			
8316				
8320	16.0			
8324				
8328	17.0			
8332				
8336	18.0			
8340				
8344	19.0			
8348				
8352				

Figura 3: *result021.html* file contents

for cycle 23 of our example. First, it offers links to pages *index.html* and *final.html* plus links to the files representing the state of the computer 5 cycles earlier ([-5]), one cycle earlier ([-1]), one cycle later ([+1]) and 5 cycles later ([+5]). It also offers a link to the instructions–time diagram till the current cycle (Crono). The execution unit stages are depicted, indicating which instruction is in each stage. Empty stages held the equivalent to a nop instruction. Since integer and floating point registers files are kept separated, it is possible that there is up to one integer and one floating point instruction in the WB stage. Control signals activated when a hazard is detected are also displayed. The applied forwardings are also visible. One can also inspect the contents of the integer registers (R0 to R31), the floating point registers (F0 to F31), and the floating point state register (FPSR). Finally, the contents of the data memory is also given. The files containing the state of the processor enable a step-wise execution of the program, allowing a closer analysis of the program.

After executing the program, check that in the address labelled as z there is a 60-element array with the expected content. Annotate the execution time and the resulting CPI.

⇒ **Fill the corresponding row of Table 1 located at the end of this document.**

Program modification using static instructions scheduling

1. *Loop unrolling*

   This technique basically replicates the base code of a loop several times, decreasing the total number of iterations.

   In our example, since the maximum number of stalls required to solve the RAW hazard is 3 cycles, the loop body of the program for $\vec{Z} = a + \vec{Y}$ must be replicated 4 times (3+1), as shown in the following. Note that some registers have been renamed to eliminate name dependencies:

```
# z = a + y
    .data
# vector y
# vector size: 60 elements
y:  .double  0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0
    .double 10.0,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0
    .double 20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0,28.0,29.0
    .double 30.0,31.0,32.0,33.0,34.0,35.0,36.0,37.0,38.0,39.0
    .double 40.0,41.0,42.0,43.0,44.0,45.0,46.0,47.0,48.0,49.0
    .double 50.0,51.0,52.0,53.0,54.0,55.0,56.0,57.0,58.0,59.0

# vector z
# 60 elements are 480 bytes
z:  .space 480

# scalar a
a:  .double 1.0

    .text

start:
    addi t1, gp, y       # t1 points to y
    addi t2, gp, z       # t2 points to z
    fld f0, a(gp)        # f0 holds a
    addi t3, t1, 480     # 60 elements are 480 bytes
loop:
    fld f1, 0(t1)
    fadd.d f2, f0, f1
    fsd f2, 0(t2)
    fld f3, 8(t1)
    fadd.d f4, f0, f3
    fsd f4, 8(t2)
    fld f5, 16(t1)
    fadd.d f6, f0, f5
    fsd f6, 16(t2)
    fld f7, 24(t1)
    fadd.d f8, f0, f7
    fsd f8, 24(t2)
    addi t1, t1, 32
    sub t4, t3, t1
    addi t2, t2, 32
    bnez t4, loop

    ori a7, zero, 10     # end
    ecall
```

This program is stored in file `dapyu1.s`. Execute this new program:

```
riscv-m -d c -c pnt1 -a 5 -f dapyu1.s
```

Check the correctness of the result and annotate the execution time. Calculate the resulting CPI. Quantify the speedup with respect to the original program.

⇒ **Fill the corresponding row of Table 1 located at the end of this document.**

The previous code can be easily modified to eliminate all the data hazards:

```
# z = a + y
    .data
# vector y
# vector size: 60 elements
y:  .double  0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0
    .double 10.0,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0
    .double 20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0,28.0,29.0
    .double 30.0,31.0,32.0,33.0,34.0,35.0,36.0,37.0,38.0,39.0
    .double 40.0,41.0,42.0,43.0,44.0,45.0,46.0,47.0,48.0,49.0
    .double 50.0,51.0,52.0,53.0,54.0,55.0,56.0,57.0,58.0,59.0

# vector z
# 60 elements are 480 bytes
z:  .space 480

# scalar a
a:  .double 1.0

    .text

start:
    addi t1, gp, y       # t1 points to y
    addi t2, gp, z       # t2 points to z
    fld f0, a(gp)        # f0 holds a
    addi t3, t1, 480     # 60 elements are 480 bytes
loop:
    fld f1, 0(t1)
    fld f3, 8(t1)
    fld f5, 16(t1)
    fld f7, 24(t1)
    fadd.d f2, f0, f1
    fadd.d f4, f0, f3
    fadd.d f6, f0, f5
    fadd.d f8, f0, f7
    fsd f2, 0(t2)
    fsd f4, 8(t2)
    fsd f6, 16(t2)
    fsd f8, 24(t2)
    addi t1, t1, 32
    sub t4, t3, t1
    addi t2, t2, 32
    bnez t4, loop

    ori a7, zero, 10     # end
    ecall
```

This new version of the program is stored in file `dapyu.s`. Execute the program:

```
riscv-m -d c -c pnt1 -a 5 -f dapyu.s
```

Check the correctness of the result and annotate the execution time. Calculate the resulting CPI. Quantify the speedup with respect to the original program.

⇒ **Fill the corresponding row of Table 1 located at the end of this document.**

2. *Software pipelining.*

   This technique replaces the original loop body of the program with a variant that consists of instructions belonging to different iterations of the original loop to eliminate data hazards.

   The code for the operation $\vec{Z} = a + \vec{Y}$ can be modified as follows to include software pipelining:

```
# z = a + y
    .data
# vector y
# vector size: 60 elements
y:  .double  0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0
    .double 10.0,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0
    .double 20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0,28.0,29.0
    .double 30.0,31.0,32.0,33.0,34.0,35.0,36.0,37.0,38.0,39.0
    .double 40.0,41.0,42.0,43.0,44.0,45.0,46.0,47.0,48.0,49.0
    .double 50.0,51.0,52.0,53.0,54.0,55.0,56.0,57.0,58.0,59.0

# vector z
# 60 elements are 480 bytes
z:  .space 480

# scalar a
a:  .double 1.0

    .text

start:
    addi t1, gp, y        # t1 points to y
    addi t2, gp, z        # t2 points to z
    fld f0, a(gp)         # f0 holds a
    addi t3, t1, 480      # 60 elements are 480 bytes
prepara:
    fld f2, 0(t1)
    fadd.d f4, f0, f2
    fld f2, 8(t1)
    addi t1, t1, 16
loop:
    fsd f4, 0(t2)
    fadd.d f4, f0, f2
    fld f2, 0(t1)
```

```
        addi t1, t1, 8
        sub t4, t3, t1
        addi t2, t2, 8
        bnez t4, loop
    resto:
        fsd f4, 0(t2)
        fadd.d f4, f0, f2
        fsd f4, 8(t2)

        ori a7, zero, 10      # end
        ecall
```

This program is stored in file `dapysp.s`. Execute the following command:

```
riscv-m -d c -c pnt1 -a 5 -f dapysp.s
```

Check the correctness of the result and annotate the execution time. Calculate the resulting CPI. Quantify the speedup with respect to the original program.

⇒ **Fill the corresponding row of Table 1.**

	1ª iteration				# iterations	Total execution			
	From cycle	To cycle	# cycles	# inst.		# cycles	# inst.	CPI	Speedup
Original (dapy.s)	5	16	12	7	60	729	426	1.71	1
dapyu1.s	5	37	33	16	15	504	246	2.05	1.427
dapyu.s	5	21	17	16	15	264	246	1.07	2.761
dapysp.s	10	18	9	7	58	485	419	1.16	1.503

Cuadro 1: Table of results

⇒ Answer the following questions about the results obtained:

- Why does loop unrolling improve software pipelining significantly?

- Why should the speedup be obtained by dividing the total cycles while a different result is obtained if the CPI is divided?

  Because there is a different number of instructions in every version of the program.
  As every version does the same in a different way, we have to compare the cycles it takes to do that, the number of instructions is not important.

Development of a new program

In this section we will assume that latencies for the adder and the multiplier are 2 and 4 cycles, respectively (options `-a 2 -m 4` for the simulator).

1. Write the RISC-V code for the execution of the operation $\vec{Z} = a * \vec{X} + \vec{Y}$ (DAXPY loop). Assume that all the arrays contain 60 floating point numbers.

   Take as reference the program in file `daxpy.s`.

   IMPORTANT: In case of detecting an error of type "undefined label (etiqueta indefinida) o syntax error", check whether the error is one of those listed in the Annex A of this document.

   Execute the program in the simulator.

   ```
   riscv-m -d c -c pnt1 -a 2 -m 4 -f daxpy.s
   ```

   Annotate the performance results:

   - Instructions= 607
   - Cycles= 850
   - CPI= 1.4

2. Apply *loop unrolling* to the developed code, reorganizing the code when necessary in order to reduce the number of stalls.

   Use as reference the program developed in the previous exercise, copying it to a different file (e.g., `daxpyu.s`). Write the new code and execute it.

   ```
   riscv-m -d c -c pnt1 -a 2 -m 4 -f daxpyu.s
   ```

   Evaluate the performance and compare it with the reference version.

   - Instructions= 407
   - Cycles= 470
   - CPI= 1.15
   - S= 1.809

   Were all stalls eliminated? If that was not the case, explain the reasons.

No, the stalls between separate loop iterations could not be fixed. There are 2 stalls inserted between the mul.d instructions of the previous iteration and the loads of the following iteration. This is because there is a structural hazard, and 2 stalls need to be inserted so the WB of the load doesn't happen at the same time as the WB of a mul.d operation.