
Lab. 4: “STATIC INSTRUCTIONS SCHEDULING”

Computer Architecture and Engineering (3rd 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 all 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
```

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

Dirección	Instrucciones	Formato
start	addi t1, gp, 0 [y]	00018313 0000000000000000110000110000010011 I imm rsl f1 rd op
4100	addi t2, gp, 480 [z]	1e018393 000111000000001100000110000010010011 I imm rsl f1 rd op
4104	fld f0, 960(gp) [a]	3c01b007 001110000000001101000000000000111 I imm rsl f1 rd op
4108	addi t3, t1, 480	1e030e13 0011110000000011000011100000100111 I imm rsl f1 rd op
loop	fld f1, 0(t1)	00033087 00000000000000001100100001000111 R imm rsl f1 rd op
4116	fadd.d f2, f0, f1	02107153 000000100001000011100010100010100011 S i[11:5] rs2 rsl f1 i[4:0] op
4120	fsd f2, 0(t2)	0023b027 000000000010000110001000001000111 R imm rsl f1 i[4:0] op
4124	addi t1, t1, 8	00830313 00000000100000110000001100000011000111 I imm rsl f1 rd op
4128	sub t4, t3, t1	406e0eb3 010000001101110000111010110011 R imm rs2 rsl f1 rd op
4132	addi t2, t2, 8	00838393 000000001000001100001100001000111 I imm rsl f1 rd op
4136	bne t4, zero, -24 [loop]	fe0e94e3 111111000001101001010011100011 B i[10:5] rs2 rsl f1 i[4:1] B op
4140	ori a7, zero, 10	00a06893 00000000101000001101000100100111 I imm rsl f1 rd op
4144	ecall	00000073 0000000000000000000001110011 I imm rsl f1 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																																																																									
<table border="1"> <thead> <tr> <th>Registros int</th> <th>Registros fp</th> <th>Memoria de Datos. Región 2</th> <th>Memoria de Instrucciones</th> </tr> <tr> <th>Registro</th> <th>Registro</th> <th>Dirección</th> <th>Dirección</th> </tr> </thead> <tbody> <tr> <td>x0 (zero)</td> <td>f0 (ft0)</td> <td>+0</td> <td>start</td> </tr> <tr> <td>x1 (ra)</td> <td>f1 (ft1)</td> <td>+1</td> <td>addi t1, gp, 0 [y]</td> </tr> <tr> <td>x2 (sp)</td> <td>f2 (ft2)</td> <td>+2</td> <td>4100 addi t2, gp, 480 [z]</td> </tr> <tr> <td>x3 (gp)</td> <td>f3 (ft3)</td> <td>+3</td> <td>4104 f1d f0, 960(gp) [a]</td> </tr> <tr> <td>x4 (tp)</td> <td>f4 (ft4)</td> <td></td> <td>4108 addi t3, t1, 480</td> </tr> <tr> <td>x5 (t0)</td> <td>f5 (ft5)</td> <td></td> <td>loop f1d f1, 0(t1)</td> </tr> <tr> <td>x6 (t1)</td> <td>f6 (ft6)</td> <td></td> <td>4116 fadd.d f2, f0, f1</td> </tr> <tr> <td>x7 (t2)</td> <td>f7 (ft7)</td> <td></td> <td>4120 fsd f2, 0(t2)</td> </tr> <tr> <td>x8 (s0)</td> <td>f8 (fs0)</td> <td></td> <td>4124 addi t1, t1, 8</td> </tr> <tr> <td>x9 (s1)</td> <td>f9 (fs1)</td> <td></td> <td>4128 sub t4, t3, t1</td> </tr> <tr> <td>x10 (a0)</td> <td>f10 (fa0)</td> <td></td> <td>4132 addi t2, t2, 8</td> </tr> <tr> <td>x11 (a1)</td> <td>f11 (fa1)</td> <td></td> <td>4136 bne t4, zero, -24 [loop]</td> </tr> <tr> <td>x12 (a2)</td> <td>f12 (fa2)</td> <td></td> <td>4140 ori a7, zero, 10</td> </tr> <tr> <td>x13 (a3)</td> <td>f13 (fa3)</td> <td></td> <td>4144 ecall</td> </tr> <tr> <td>x14 (a4)</td> <td>f14 (fa4)</td> <td></td> <td></td> </tr> <tr> <td>x15 (a5)</td> <td>f15 (fa5)</td> <td></td> <td></td> </tr> </tbody> </table>			Registros int	Registros fp	Memoria de Datos. Región 2	Memoria de Instrucciones	Registro	Registro	Dirección	Dirección	x0 (zero)	f0 (ft0)	+0	start	x1 (ra)	f1 (ft1)	+1	addi t1, gp, 0 [y]	x2 (sp)	f2 (ft2)	+2	4100 addi t2, gp, 480 [z]	x3 (gp)	f3 (ft3)	+3	4104 f1d f0, 960(gp) [a]	x4 (tp)	f4 (ft4)		4108 addi t3, t1, 480	x5 (t0)	f5 (ft5)		loop f1d f1, 0(t1)	x6 (t1)	f6 (ft6)		4116 fadd.d f2, f0, f1	x7 (t2)	f7 (ft7)		4120 fsd f2, 0(t2)	x8 (s0)	f8 (fs0)		4124 addi t1, t1, 8	x9 (s1)	f9 (fs1)		4128 sub t4, t3, t1	x10 (a0)	f10 (fa0)		4132 addi t2, t2, 8	x11 (a1)	f11 (fa1)		4136 bne t4, zero, -24 [loop]	x12 (a2)	f12 (fa2)		4140 ori a7, zero, 10	x13 (a3)	f13 (fa3)		4144 ecall	x14 (a4)	f14 (fa4)			x15 (a5)	f15 (fa5)		
Registros int	Registros fp	Memoria de Datos. Región 2	Memoria de Instrucciones																																																																							
Registro	Registro	Dirección	Dirección																																																																							
x0 (zero)	f0 (ft0)	+0	start																																																																							
x1 (ra)	f1 (ft1)	+1	addi t1, gp, 0 [y]																																																																							
x2 (sp)	f2 (ft2)	+2	4100 addi t2, gp, 480 [z]																																																																							
x3 (gp)	f3 (ft3)	+3	4104 f1d f0, 960(gp) [a]																																																																							
x4 (tp)	f4 (ft4)		4108 addi t3, t1, 480																																																																							
x5 (t0)	f5 (ft5)		loop f1d f1, 0(t1)																																																																							
x6 (t1)	f6 (ft6)		4116 fadd.d f2, f0, f1																																																																							
x7 (t2)	f7 (ft7)		4120 fsd f2, 0(t2)																																																																							
x8 (s0)	f8 (fs0)		4124 addi t1, t1, 8																																																																							
x9 (s1)	f9 (fs1)		4128 sub t4, t3, t1																																																																							
x10 (a0)	f10 (fa0)		4132 addi t2, t2, 8																																																																							
x11 (a1)	f11 (fa1)		4136 bne t4, zero, -24 [loop]																																																																							
x12 (a2)	f12 (fa2)		4140 ori a7, zero, 10																																																																							
x13 (a3)	f13 (fa3)		4144 ecall																																																																							
x14 (a4)	f14 (fa4)																																																																									
x15 (a5)	f15 (fa5)																																																																									

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 held 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

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 dapyul.s. Execute this new program:

```
riscv-m -d c -c pnt1 -a 5 -f dapyul.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 $\vec{Z} = \vec{a} + \vec{Y}$ 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.

Podemos restarle a la cantidad de ciclos totales los ciclos que hemos gastado en el principio y en el final, y el resultado lo hemos dividido por los ciclos gastados en cada iteración. CUIDADO; el primer loop tiene un stall que el resto no tienen, de manera que los ciclos gastados en cada iteración no son 9, sino 8, y por tanto hay que dividir 464/8

	1 ^a iteration				# iterations	Total execution			
	From cycle	To cycle	# cycles	# inst.		# cycles	# inst.	CPI	Speedup
Original (dapysp.s)	5	16	12	7	60	729	426	1.71	x
dapyu1.s	5	37	33	16	60/4 = 15	504	245	2.05	1.446
dapyu.s	5	21	17	16	60/4 = 15	264	246	1.07	729/264 = 2.76
dapysp.s	10	18	9	7	58	485	419	1.16	1.5

Cuadro 1: Table of results

⇒ Answer the following questions about the results obtained:

- Why does loop unrolling improve software pipelining significantly?

Because we manage to perform 4 iterations in just one, reducing the number of loop repetitions and avoiding stalls

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

Because by using the CPI in some cases we obtain a fairer speedup

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= 307
- Cycles= 355
- CPI= 1.16
- S= 2.394

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

No, not every stall was eliminated. The only stalls that exist are due to the attempt of performing 2 WBs in floating point registers at the same time.