Exercise sheet 7 – Processor architecture

Goals:

- Pipelining
- Instruction Scheduling

Exercise 7.1: Pipelining

(a) Given is a sequence of instructions and a five-stage-pipeline. State the procedure and be careful, the instructions are not ordered perfectly to fully utilise the pipeline.

Instruction sequence:

Nr.	Instruction	Comment
(1)	ADD R2, R1, R1	*; R1 = R1 + R2*
(2)	ADD R1, R3, R3	*; R3 = R1 + R3*
(3)	ADD R4, R5, R5	*; R5 = R4 + R5*
(4)	CMP R5, #0, R6	*; R6 = cmp(R5, 0)*
(5)	BNE R6, M	*; Jump to M if result != 0 BNE = branch not equal*
(6)	INST1	*; some random instruction 1*
(7)	INST2	*; some random instruction 2*
	...	
(8)	M: INST100	*; some random instruction 100*
	...	

Five-stage-pipeline:

Stage	Operation
1.	Fetch instruction
2.	Load operands
3.	Execute instruction
4.	Memory access
5.	Save result into register

Proposal for solution:

	Fetch instruction	Load operands	Execute instruction	Memory access	Save result into register	
	(1) ADD R2,R1,R1					1
	(2) ADD R1,R3,R3	(1) ADD R2,R1,R1				2
	(2) ADD R1,R3,R3	(*1)	(1) ADD R2,R1,R1			3
	(2) ADD R1,R3,R3			(1) ADD R2,R1,R1		4
	(2) ADD R1,R3,R3				(1) ADD R2,R1,R1 //write R1	5
	(3) ADD R4,R5,R5	(2) ADD R1,R3,R3				6
	(4) CMP R5, #0,R6	(3) ADD R4,R5,R5	(2) ADD R1,R3,R3			7
	(4) CMP R5, #0,R6	(*1)	(3) ADD R4,R5,R5	(2) ADD R1,R3,R3		8
	(4) CMP R5, #0,R6			(3) ADD R4,R5,R5	(2) ADD R1,R3,R3 //write R3	9
	(4) CMP R5, #0,R6				(3) ADD R4,R5,R5 //write R5	10
	(5) BNE R6, M	(4) CMP R5, #0,R6				11
	(5) BNE R6, M	(*1)	(4) CMP R5, #0,R6			12
	(5) BNE R6, M			(4) CMP R5, #0,R6		13
	(5) BNE R6, M				(4) CMP R5, #0,R6 //write R6	14
	(6) INST 1 (*2)	(5) BNE R6, M				15
	(7) INST 2 (*2)		(5) BNE R6, M			16
				(5) BNE R6, M		17
				(*3)	(5) BNE R6, M //jump to INST100	18
M	(8) INST 100					19
	INST 101	(8) INST 100				20
			(8) INST 100			21
				(8) INST 100		22
					(8) INST 100	23

time

(*1) Data dependency
(*2) Assumption: The next instructions (INST 1, INST 2) are already loaded into the pipeline
→ speculative loading of instructions
→ if the jump is not taken, execution can be proceeded with less empty cycles
(*3) Assumption: The jump is taken
→ INST 1 and INST 2, which are already loaded into the pipeline have to be dropped
→ the pipeline has to rebuild completely, beginning at INST 100 (jump marker M)

(b) Suggest ideas for a better pipeline utilisation of *exercise 7.1a*.
Assume there are additional instructions before the given program excerpt.

Proposal for solution: It is not possible to avoid data-dependency itself, but you can rearrange the execution order of the instructions that do not depend on each other. For example while the program waits for *ADD R2,R1,R1* to finish and to load *ADD R1,R3,R3* it could already load *ADD R4,R5,R5*. This is called *instruction scheduling*.

	Fetch instruction	Load operands	Execute instruction	Memory access	Save result into register	
	(1) ADD R2,R1,R1					1
	(3) ADD R4,R5,R5	(1) ADD R2,R1,R1				2
	(2) ADD R1,R3,R3	(3) ADD R4,R5,R5	(1) ADD R2,R1,R1			3
	(2) ADD R1,R3,R3	(*1)	(3) ADD R4,R5,R5	(1) ADD R2,R1,R1		4
	(2) ADD R1,R3,R3			(3) ADD R4,R5,R5	(1) ADD R2,R1,R1 //write R1	5
	(4) CMP R5, #0,R6	(2) ADD R1,R3,R3			(3) ADD R4,R5,R5 //write R5	6
	(5) BNE R6, M	(4) CMP R5, #0,R6	(2) ADD R1,R3,R3			7
	(5) BNE R6, M	(*1)	(4) CMP R5, #0,R6	(2) ADD R1,R3,R3		8
	(5) BNE R6, M			(4) CMP R5, #0,R6	(2) ADD R1,R3,R3 //write R1	9
	(5) BNE R6, M				(4) CMP R5, #0,R6 //write R6	10
	(6) INST 1 (*2)	(5) BNE R6, M				11
	(7) INST 2 (*2)		(5) BNE R6, M			12
				(5) BNE R6, M		13
				(*3)	(5) BNE R6, M	14
M	(8) INST 100					15
	INST 101	(8) INST 100				16
			(8) INST 100			17
				(8) INST 100		18
					(8) INST 100	19
						20
						21
						22
						23

(*1) Data dependency
(*2) Assumption: The next instructions (INST 1, INST 2) are already loaded into the pipeline
→ speculative loading of instructions
→ if the jump is not taken, execution can be proceeded with less empty cycles
(*3) Assumption: The jump is taken
→ INST 1 and INST 2, which are already loaded into the pipeline have to be dropped
→ the pipeline has to rebuild completely, beginning at INST 100 (jump marker M)

Another order which additionally can reduce the overall time by one time step:

	Fetch instruction	Load operands	Execute instruction	Memory access	Save result into register	
	(3) ADD R4,R5,R5					1
	(1) ADD R2,R1,R1	(3) ADD R4,R5,R5				2
	(4) CMP R5, #0,R6	(1) ADD R2,R1,R1	(3) ADD R4,R5,R5			3
	(4) CMP R5, #0,R6	(*1)	(1) ADD R2,R1,R1	(3) ADD R4,R5,R5		4
	(4) CMP R5, #0,R6			(1) ADD R2,R1,R1	(3) ADD R4,R5,R5 //write R5	5
	(2) ADD R1,R3,R3	(4) CMP R5, #0,R6			(1) ADD R2,R1,R1 //write R1	6
	(5) BNE R6, M	(2) ADD R1,R3,R3	(4) CMP R5, #0,R6			7
	(5) BNE R6, M	(*1)	(2) ADD R1,R3,R3	(4) CMP R5, #0,R6		8
	(5) BNE R6, M			(2) ADD R1,R3,R3	(4) CMP R5, #0,R6 //write R6	9
	(6) INST 1 (*2)	(5) BNE R6, M			(2) ADD R1,R3,R3 //write R1	10
	(7) INST 2 (*2)		(5) BNE R6, M			11
				(5) BNE R6, M		12
				(*3)	(5) BNE R6, M //jump to INST100	13
M	(8) INST 100					14
	INST 101	(8) INST 100				15
			(8) INST 100			16
				(8) INST 100		17
					(8) INST 100	18
						19
						20
						21
						22
						23

(*1) Data dependency
(*2) Assumption: The next instructions (INST 1, INST 2) are already loaded into the pipeline
→ speculative loading of instructions
→ if the jump is not taken, execution can be proceeded with less empty cycles
(*3) Assumption: The jump is taken
→ INST 1 and INST 2, which are already loaded into the pipeline have to be dropped
→ the pipeline has to rebuild completely, beginning at INST 100 (jump marker M)

Exercise 7.2: Pipeline-Simulator

The pipeline simulator is taken from *http://euler.vcsu.edu/curt.hill/mips.html*.

(a) Update the `RA_exercises` repository with `git pull`.

(b) Change into the `RA_exercises/sheet_07/PipelineSimulator` directory.

(c) To compile the pipeline-simulator run `javac GUI.java` from a terminal, or use the `Makefile`. *Hint: Make sure that you have properly installed the JDK and JRE. Within the VM it is already installed.*

(d) Start the simulator with: `java GUI`

(e) In *mips.html* you can find some information on how to work with the simulator.

(f) Create a little assembly program, based on *exercise 7.1a* and run the simulator. *Hint: There is no one-to-one mapping possible, because not all required instructions are available.*

Proposal for solution: Currently there is no solution to load a program into the simulator than loading the instructions one by one.

```
1  0: ADDI $1 $0 1   //can be used as an initialisation step for the registers
2  0: ADDI $5 $0 3   //can be used as an initialisation step for the registers
3
4  0: ADD $2 $1 $1
5  1: ADD $3 $2 $2
6  2: ADD $4 $3 $3
7  3: ADD $5 $4 $4
8  4: ADD $6 $5 $5
9  5: ADD $7 $6 $6
10 6: ADDI $1 $1 0
11 7: BNE $10 $1 0
```

This programs run infinite, as the statement in line 7 never becomes true. You can check out the current values in the register-view in the right upper corner of the simulator window.