Laboratory on processor design
Course Computer Technology

The target of this practical is that the student gets a feeling for the evolution of a digital circuit towards an actual processor.
To reach this target, the student designs a circuit that represent a working processor and simulates them using binary instructions.

The simulation software used is Logisim. With this tool we build, test and simulate a digital circuit. Although the software can be downloaded from various sources, we recommend using the version of DAC that can be downloaded from the CV. It contains several features that facilitate to fill the instruction and data memory.

In the first laboratory session, we get familiar with Logisim. At home, you can already follow the steps of the tutorial to get a feeling of the editor.

Exercise 1
Create the circuit following the steps of the tutorial of Logisim.

Exercise 2
Create a sequential circuit that generates the Fibonacci numbers.

$$f_0=0; \quad f_1=1; \ldots\ldots\ldots; f_n = f_{n-1} + f_{n-2} \text{ para } n = 2,3,4 \ldots.$$

To be more specific, the band width of the registers, adders etc should be 6 bits. After getting the essential working, you should convert the system in a real circuit able to take off and land safely. Therefore, the input should consist of

- A “reset” signal, that makes the circuit take the starting values
- A “start signal” that takes the input *n* and starts calculating the corresponding Fibonacci number
- The number *n* in 4 bits.

The circuit should automatically stop when having reached the Fibonacci number and generate

- an output “Fn” which is this number represented by 6 bits
- An “halt” signal that indicates that the calculation is ready.

Test the circuit using for *n* the most significant digit of your DNI. Make a screenshot of the simulator and upload it with the circuit in the assignment in guac.

Exercise 3
To generate the MIPS circuit of the book, we can start considering the program counter as a register with a content that on every clock cycle is increased by 4. Implement the corresponding circuit.

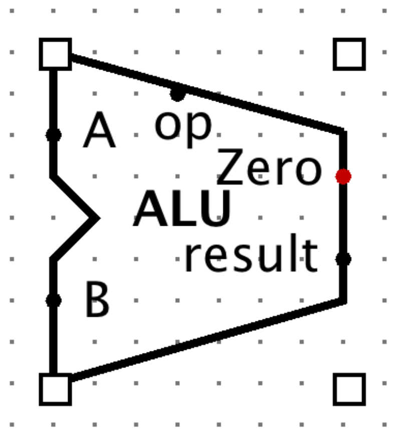
Exercise 4
With the instruction memory, we can extend the updating of the PC with the Jump instruction. Extend the circuit of Exercise 3 to include the instruction memory and PC such that on a J instruction, the PC goes to the indicated address. Test the circuit by putting several J instructions in the instruction memory. For that we need to get acquainted with the instruction memory ingredient that is described below and can be found in the file called **TG00MIPS**. Upload your working jump circuit in the assignment in guac.

In the large assignment we are going to extend to basic steps to build your own MIPS processor that can handle a limited set of instructions. Hopefully, your Jump instruction is already comfortably working. To facilitate the construction, several ingredients called sub-circuits are provided in a file called **TG00MIPS**. To start working with it, rename it to “NameOfTeam”MIPS.

To deal with this module, we should be aware that Logisim can handle sub-circuits in a hierarchical modular way. I.e. one can build a circuit that can be used in another circuit. The main circuit used for simulation purposes is called TG00MIPS. It uses directly a module called MIPS (can be replaced by another circuit) that is to be built by the student team.

The pre-built modules that can be used are described now.

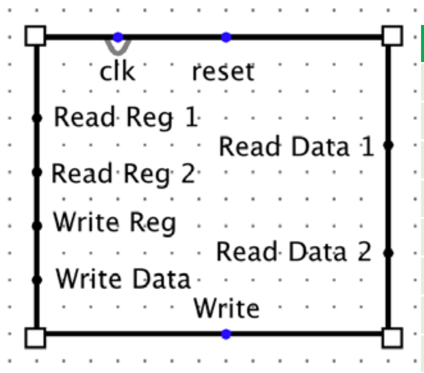
ALU



Signals	
A[31:0]	Data in
B[31:0]	Data in
op[3:0]	ALU Op function code
result[31:0]	Data out
Zero	Is (1) if the result is zero

op[3:0]	ALU operation	
0000	result = A & B	And (bitwise)
0001	result = A | B	Or (bitwise)
0010	result = A + B	add
0110	result = A – B	subtract
0111	If (A<B) result = 1 else result = 0	Assert if A is smaller than B
1100	result = $\overline{A \mid B}$	Nor (bitwise)

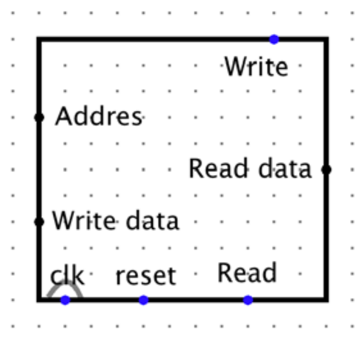
REGISTER FILE



signals	
clk	Clock tick to synchronize writing
Write	Write signal (1)
Read Data 1[31:0]	data read from address in “Read Reg 1”
Read Data 2[31:0]	data read from address in “Read Reg 2”
Write Data[31:0]	Data to write in address in “Write Reg”
Read Reg 1[5:0]	Register address for Reading 1
Read Reg 2[5:0]	Register address for Reading 2
Write Reg[5:0]	Register address for writing

Write	Register operation
0	Reading: Read Data 1[31:0] = Content stored into register of address Read Reg 1[5:0] Read Data 2[31:0] = Content stored into register of address Read Reg 2[5:0]
1	Write on falling CLK edge: Register with address Write Reg[5:0] is loaded with Write Data[31:0] The two reads are also done independent of the Write signal

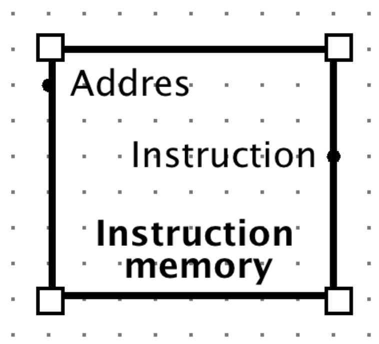
Data memory (32)



Signal	
clk	Clock tick to synchronize writing
Write	Write signal (1)
Read	read signal (1)
Read data[31:0]	Data bus output (read)
Write data[31:0]	Data bus input (writing)
Addres[31:0]	Address bus

Write	Read	Memory operation
0	1	Read: Read data[31:0] = M(Addres[31:0])
1	0	Write at falling clk edge: M(Addres[31:0]) ← Read data[31:0]

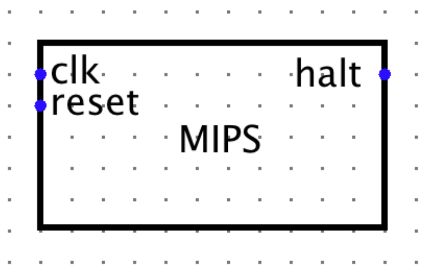
Instruction memory



Signals	
Instruction[31:0]	Output bus (read)
Addres[31:0]	Address bus

Memory operation
Continuously: Instruction[31:0] = M(Addres[31:0])

The interface (input output) of the processor to design is:



Input:

- clk: system clock to synchronize instructions
- reset: puts all memory elements to zero

Output:

- halt: final signal activated when running specific instruction “halt”.
  It replaces in the simulation the end of a process (running program) as we do not return to the operating system with a syscall.

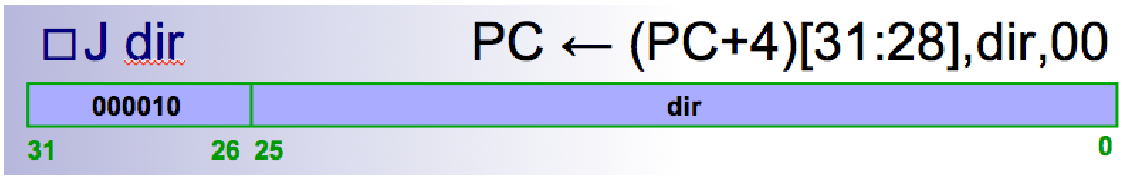
The modules help us to implement the typical characteristics of the processor TG00MIPS to be designed:

- 32 bits word
- 32 register file of 32 bits. Register $0 is read-only and has the value 0
- Arithmetic logical unit (ALU) of 32 bits with the operations:
    - Add and subtract of integer operands in two's complement (2C)
    - Logical AND, logical OR and NOR
    - comparison "less than"
    - State flag Z: indicates the ALU result is 0
- Instruction and data memory
    - 32 bits address bus
    - Addresses on level of byte

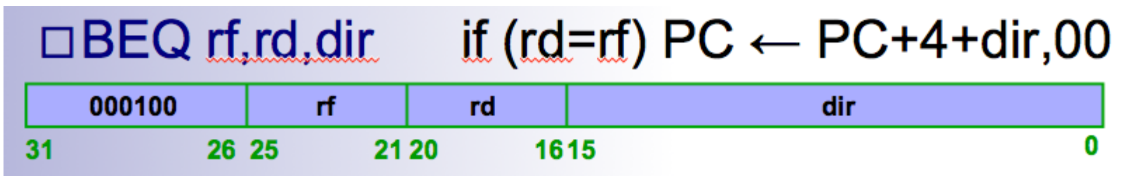
The processor should be designed such that it "understands" (at least) the following set of instructions.

- **<u>Branch instructions</u>:**

  Unconditional branch: **J dir**

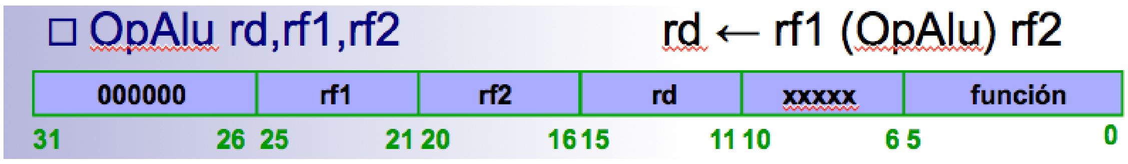


  Conditional branch: **BEQ rf, rd, dir**



- **Arithmetic-logical instructions:**

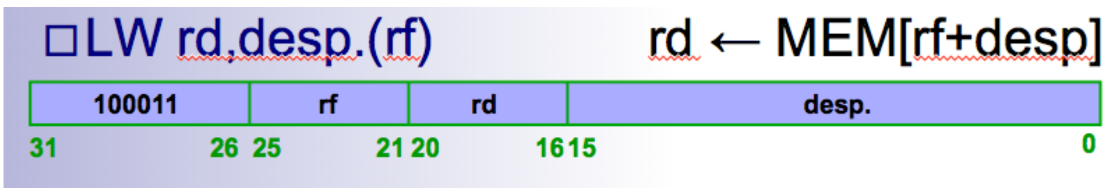
OpAlu rd, rf1, rf2



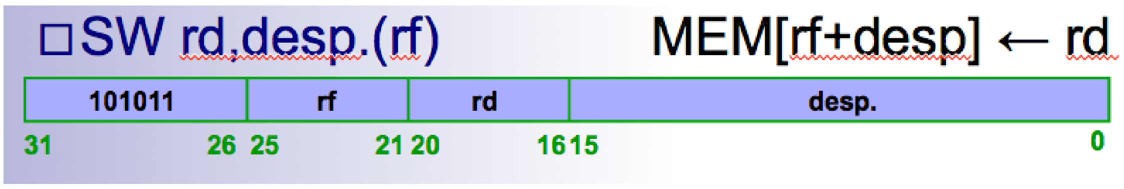
OpAlu	func	OpAlu	func
ADD	100000	OR	100101
SUB	100010	SLT	101010
AND	100100		

- **Transfer instructions:**

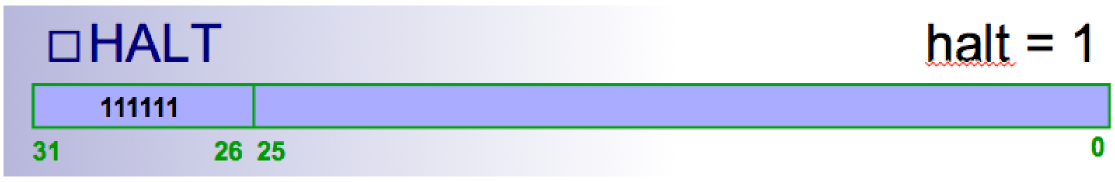
Load: **LW rd, desp.(rf)**



Store: **SW rd, desp.(rf)**



- **finalizing instruction:**
    - **HALT**



Large assigment : single-cycle processor

On successful finishing the exercises, we can now take the big jump of designing the full single cycle MIPS processor according to the specifications. The following steps are essential.

1.- Designing and implementing the datapath in Logisim.
Use the provided modules and other available elements: ALU, register file, adder, flip-flop's, registers, multiplexors, splitters, logical gates, bit extenders, shifters etc. Take care of choosing the right bit width.

2.- Implementing the control unit.
Design the alu control en control unit such that the correct signals are activated (asserted) within the instruction cycle.

3.- Clock cycle time.
Determine the clock cycle time in "ticks" for you designed MIPS processor given the delays of the used modules. We have to set that time in the main module of the processor, i.e. circuit TG00MIPS.

Clock cycle time:	ticks

The delay in all other modules is considered zero, but for the pre-defined modules we have the delay:

- **Instruction memory: 4**
  From the moment a valid address is given at the address gate up to the appearance of the corresponding data takes 4 ticks.
- **Data memory: 4**
  As well for Reading as well for writing: reading takes 4 ticks from the moment the address is given and for writing it also takes 4 ticks to get the data written in the given address
- **ALU: 4**
  From the change of an input up to the appearance of the correct result
- **Register file: 2**
  For as well Reading as writing

4.- Programming the system
We use the program that determines the sum of a vector introduced in the first lab to test the designed processor. Some of the instructions are adapted, because the instruction set is much smaller. Moreover, we end with a halt instruction instead of calling the operating system in the end.

MARS can help us to create part of the binary machine code, but not all. In principle each instruction should correspond to one in the set described earlier.
We can fill the instruction memory (.text) directly by editing the memory or alternatively use the file TG00rom.txt which in this version of Logisim is linked to the memory. On a reset, the content of this file is loaded into the instruction

memory. Note that the first instruction (label main) should correspond to position 0 in the memory.

The values in the data segment (.data) can be written into the file TG00ram.txt associated with the data memory. One can also edit the memory and save in the file. Notice that the data labeled as “vector” are in memory position 0, “size” in position 32, “res” in 36, etc.

After having loaded the instructions and data, we can test step by step whether the processor is doing what we intended. The probes are extremely useful for that. After we are convinced that the system does what it should, we can use the options of the “simulate” options of Logisim.
In “logging” a table can be constructed with the relevant elements such as the PC to be followed. Simulation then generates a table where you can follow the development of the program.

After examination of the table by the teacher and everyone is convinced that the processor may actually be working (and not before that) you can bother the tester of GUAC and go for green.

Determine by hand the execution time of the program in you processor. You have the cycle time you set before and the number of instructions to do so.

Total running time	ticks

Compare this time with the one required by the simulation.

Test program

```
.data
vector: .word 2, 4, 6, 8, -2 -4, -6 -7
size:    .word 8
res:     .word 0
one:   .word 1
four: .word 4

.text
main:  lw $8, size($0)         # $8 counter
       add $9, $0, $0          # $9 reading address of data
       lw $1, one($0)          # $1 value 1
       lw $4, four($0)         # $4 value 4
       sub $11, $11, $11       # $11 sum accumulator
loop:  lw $10, 0($9)
       add $11, $11, $10
       add $9, $9, $4
       sub $8, $8, $1
       beq $8, $0, exit
       j loop
exit:  sw $11, res($0)
       halt
```

Possible extension: Pipelined processor

The target is to convert the single cycle processor into a pipelined version with 5 stages that can handle exactly the same set of instructions. We follow the stages of the book IF, ID, EX, MEM, WB. Tasks to get this working:

1.- Designing and implementing the datapath in Logisim.
Don't forget to copy and save the single cycle design project. We can then start editing this design mainly re-arranging the units into stages.

2.- Implementing the control unit.
In principle the control unit should be the same as that of the single cycle processor. The most important is to get the pipeline registers implemented and each signal of an instruction activated in the right stage.

3.- Clock cycle time.
Here we have rethink well the clock cycle time in "ticks" for the pipeline version. You can change this cycle time in the main module of the processor, i.e. circuit TG00MIPS.

Clock cycle time:	ticks

The delays of the modules are specified before in assignment 1.

4.- Programming the system
We use the same program that determines the sum of a vector. However, we now have to study dependences and introduce the appropriate number of NOPS to avoid the hazards and take care the program works adequately.

We have to fill the instruction memory with the new program and the data memory with the same data as before.

Determine by hand the execution time of the program in you processor. You have the cycle time you set before and the number of instructions to do so.

Total running time	ticks

Compare this time with the one required by the simulation.

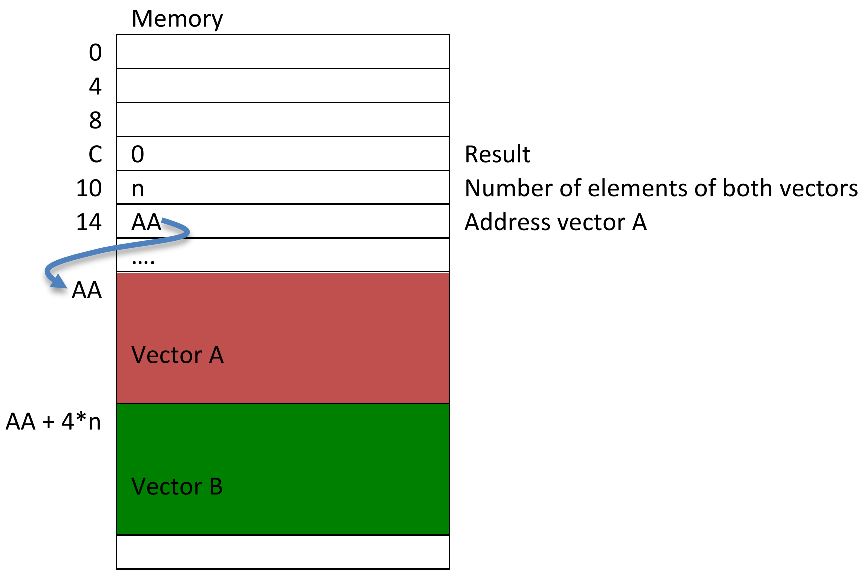
Getting efficient: Programming both processors

The target of this assignment is to create a program from a specification that can be run in the designed processors.

The specification of the program is to calculate the sum of absolute differences (SAD) of the elements of two given integer valued (32 bits in 2C) vectors A and B.

$$SAD(A,B) = \sum_i |A[i] - B[i]|$$

Memory position 10_{hex} contains the number *n* of elements of the vectors. Memory position 14_{hex} contains the address *AA* of the first element of vector A, whereas the first element of vector B can be found in position *AA*+4*n*.
The result of the SAD(A,B) calculation should be stored in memory position C_{hex} .
Memory position 0_{hex}, 4_{hex}, and 8_{hex} can be used for other variables your program needs. So the data memory looks as follows:



Tasks to get this working

1.- Make a program for your single cycle processor
Use the available instructions to create a program. Put the code of the program into the instruction memory starting at position 0; file TG00rom.txt.

Fill the data memory (TG00ram.txt) starting at position C_{hex} with the following integer (32 bits) values corresponding to two vectors with 4 elements:
0, 4, 24, 25, -3, -6, 1, 23, -1, -4,-1.

2.- Test the single cycle processor with your program
Simulate the execution step by step and check the result. Measure the running time of the program and write down

Running time	ticks

3.- .- Make a program for your pipeline processor
Change the program such that it works in a pipeline processor taking care of the possible hazards. Put the code of the program into the instruction memory starting at position 0; file TG00rom.txt.

Use the same data for the vectors in TG00ram.txt as before.

4.- Test the pipeline processor with your program
Simulate the execution step by step and check the result. Measure the running time of the program and write down

Running time	ticks

5.- Competition of pipeline processor running time
This optional task is for honour and glory. The idea is that you can reduce the running time of the program in number of ticks by

- Changing the program, reorder instructions reducing NOPS, efficient use of registers etc. to reduce the number of instructions
- Change the processor hardware to eliminate partly the hazards.
- Invent and implement new instructions in the hardware, such that the program is more efficient.

Write down in a manuscript which changes you made and what has been the effect on the total running time of the program.