Universidade do Vale do Itajaí
Computer Engineering
Processor Organization and Architecture

Test 06 – Organization of the MIPS monocycle in Quartus

Student: Lucas Mateus Gonçalves
Teacher Advisor: Douglas Rossi de Melo

June
2021

Universidade do Vale do Itajaí
Computer Engineering
Processor Organization and Architecture

Report

Sixth Organization Report of the MIPS Monocycle in Quartus presented to the Processor Architecture and Organization Program of the Computer Engineering Course at the University of Vale do Itajaí, as a partial requirement for Test 06.

Student: Lucas Mateus Gonçalves

Teacher Advisor: Douglas Rossi de Melo

June
2021

Conteúdo

1 Summary

This paper has the intention of explaining one `MIPS Mono-cycle` organization made in `VHDL` with *Quartus Software*.

Something of note to inform before the components are described and explained is, not all instructions have been integrated into this processor, it only has enough instructions to demonstrate and implement simple algorithms.

This paper will describe this processor's most important and complex components but will omit it's simpler parts for the sake of brevity.

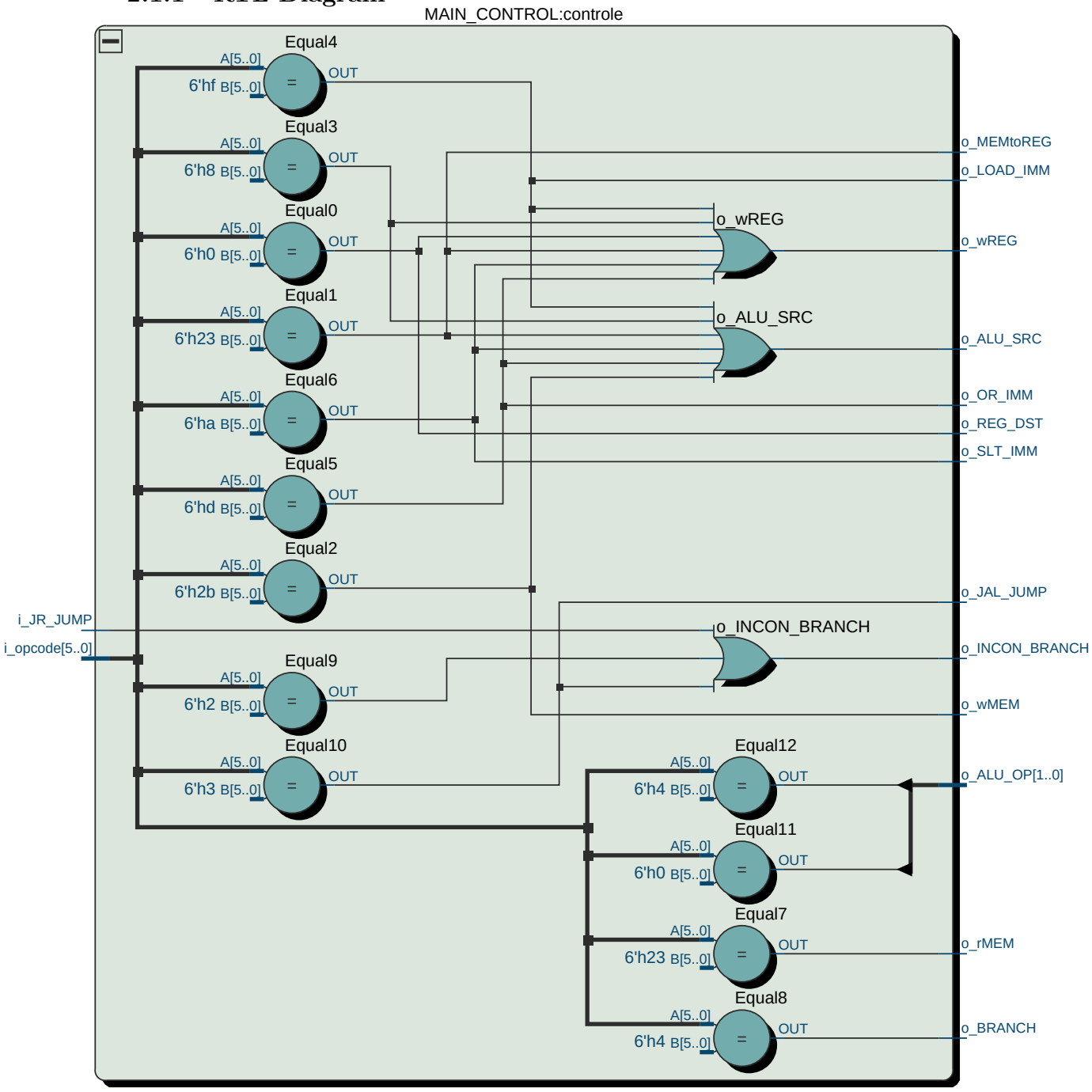
This paper will not explain the `MIPS` architecture in detail for it's the reader responsibility to be familiar with it.

The full project can be accessed in it's GitHub Repository.

2 Components of the Organization

2.1 Main Control

2.1.1 RTL Diagram



2.1.2 Description

The `Main Control Unit` controls the read and write signals of other components, the output of multiplexers and the control code of the other components.

It reads the opcode to decide what signals to turn, the opcode comes directly from the `instruction memory` and the signal it sends go directly to the components they regulate.

Of course, `Register` based instruction are not regulated by their opcodes, them and their respective signals are regulated by other components.

Although not all instructions were implemented in this processor, it is easy to understand and implement them by looking at the instructions already implemented.

```
o_REG_DST <= '1' when i_opcode = "000000" else '0'; --R
o_ALU_SRC <= '1' when (i_opcode = "100011" or i_opcode = "101011"
or i_opcode = "001000" or i_opcode = "001111"
or i_opcode = "001101" or i_opcode = "001010") else '0'; --(lw or sw
or addi or lui or ori or slti)
o_MEMtoREG <= '1' when i_opcode = "100011" else '0'; --lw
o_ALU_OP(1)     <=      '1' when i_opcode = "000000" else '0';  --R
```

The lines of code above are good examples of how the simplicity of the architecture makes for an easier implementation of its functions, and simplifies the organization.

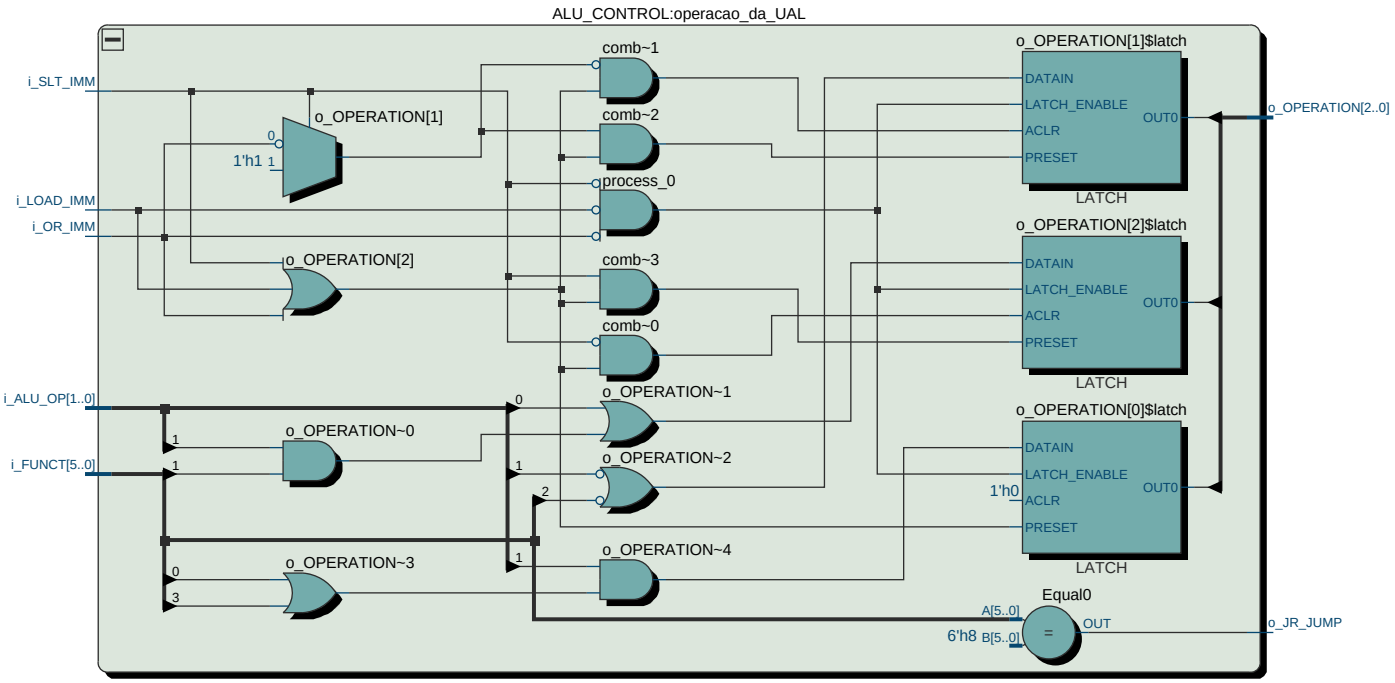
By knowing what every signal controls and what the opcode should return it is simple to add that instruction to the organization, as shown above.

For example, `o_REG_DST` controls a multiplexer that decides which part of the instruction vector the `Register File` will accept as the write address. `Register` based instructions have a different position of the write address in it's vector as opposed to `Immediate` and `Jump` based instructions.

So by simply turning on the respective signal, or signals the organization will behave as expected, there is almost never any need to change other components to implement new instructions.

2.2 Arithmetic and Logic Unit Control

2.2.1 RTL Diagram



2.2.2 Description

The `ALU Control` is a separate control unit used to choose which arithmetic or logic module of the unit should be outputted.

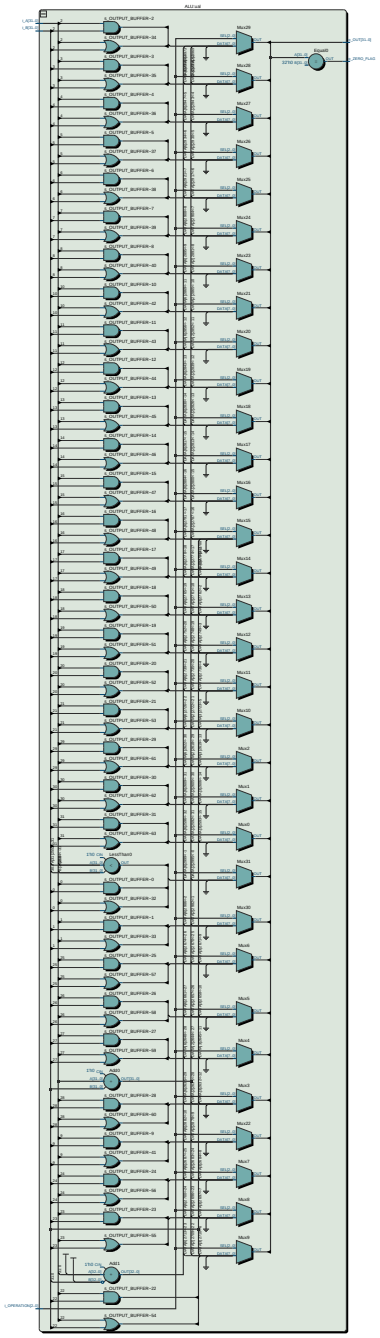
It's input signals come from the `Main Control` and the instruction vector. As the `Main Control` cannot interpret `Register` based instructions this Control Unit has to parse it and control the `ALU` directly as opposed `Immediate` and `Jump` based instructions that can be regulated by the `Main Control`.

```
if i_LOAD_IMM = '0' and i_OR_IMM = '0' and i_SLT_IMM = '0' then --R
        o_OPERATION(2) <= (i_ALU_OP(0) or (i_ALU_OP(1) and i_FUNCT(1)));
        o_OPERATION(1) <= (not i_ALU_OP(1) or not i_FUNCT(2));
        o_OPERATION(0) <= (i_ALU_OP(1) and (i_FUNCT(3) or i_FUNCT(0)));
end if;

if i_LOAD_IMM = '1' then
        o_OPERATION <= "011";   ---Forcing shift by 16
end if;

if i_OR_IMM = '1' then
        o_OPERATION <= "001";   ---Forcing OR
end if;

if i_SLT_IMM = '1' then
        o_OPERATION <= "111";   ---Forcing SLTI
end if;
```

This can be understood by the control logic, for if none of the `Immediate` control flags that come from the `Main Control` are set, the `Register` based control flags to the `ALU` are triggered based on the `'Function'` part of the instruction vector.

2.3 Arithmetic and Logic Unit

2.3.1 RTL Diagram



2.3.2 Description

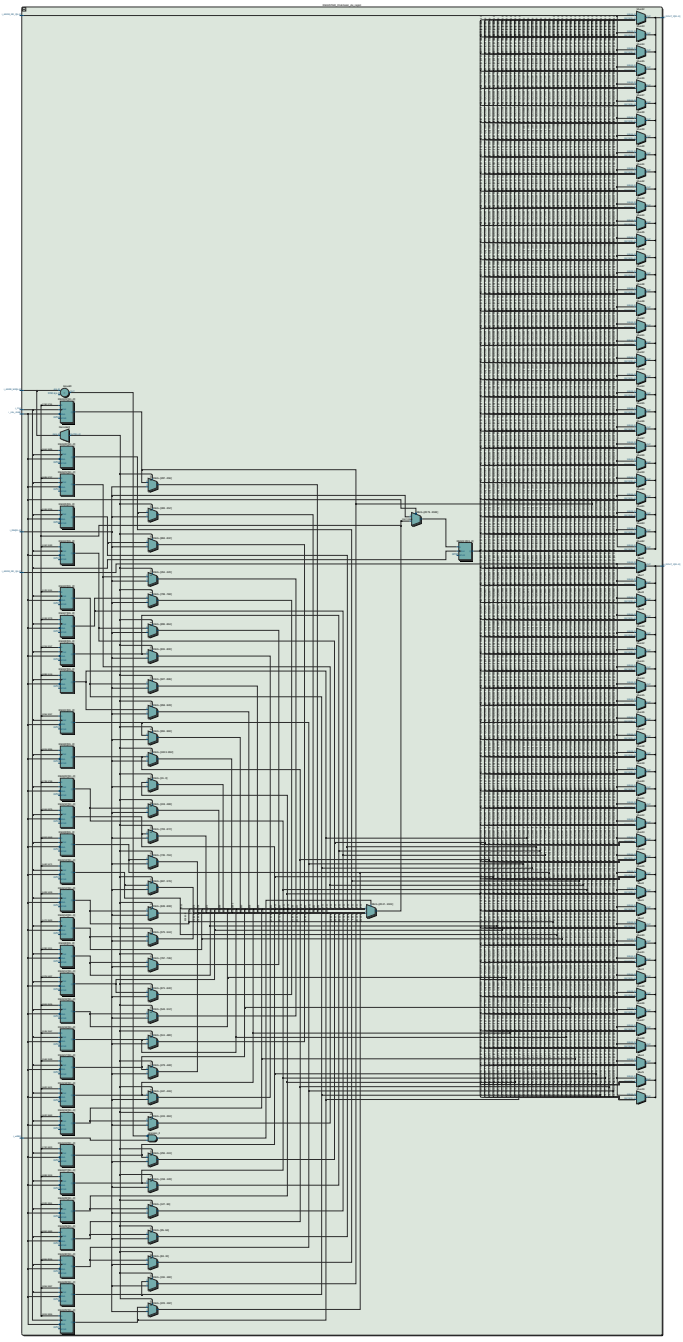
The `ALU` has been simplified to fit only the instructions that have been implemented, the most notable options of the *switch case* conditional are the `Load Upper Immediate` for `Array Loading` and `Set If Less Than`.

```
when "000" => s_OUTPUT_BUFFER <= i_A and i_B;
when "001" => s_OUTPUT_BUFFER <= i_A or i_B;
when "010" => s_OUTPUT_BUFFER <= std_logic_vector(signed(i_A) + signed(i_B));
when "011" => s_OUTPUT_BUFFER <= i_B(15 downto 0) & "0000000000000000"; --lui
when "110" => s_OUTPUT_BUFFER <= std_logic_vector(signed(i_A) - signed(i_B));
when "111" =>   --SLT
        if(signed(i_A) < signed(i_B)) then
                s_OUTPUT_BUFFER <= (0 => '1', others => '0'); -- 000...0001
        else
                s_OUTPUT_BUFFER <= (others => '0'); -- 000...0000
        end if;
when others => NULL;
```

The output is connected to a multiplexer and the `Data Memory`, so it can be used for data addressing.

2.4 Register File

2.4.1 RTL Diagram



2.4.2 Description

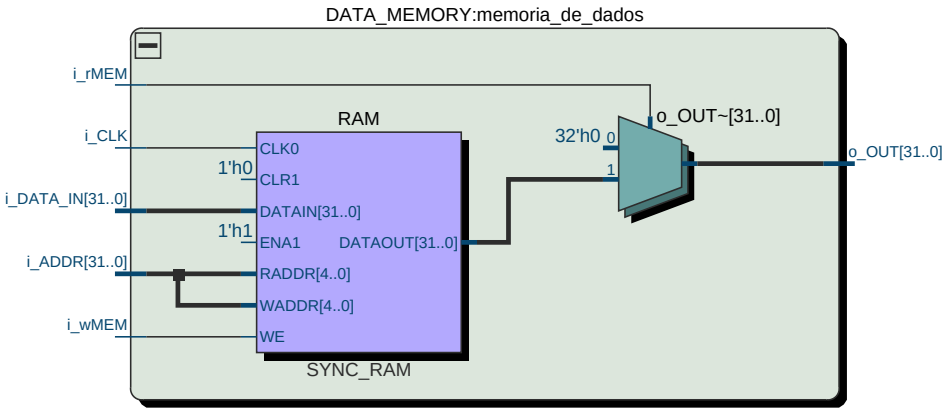
The `Register File` simply reads the three addresses that come from the instruction vector do determine which of its register is being written to, and which are outputted to other components.

If the processor is undergoing a `'jal'` instruction, the signal `'i_JAL_JUMP'` will force a write of the vector in `'i_DIN'` to the register *31*. The data in that vector will be the next instruction address.

```
if i_JAL_JUMP = '0' then

    if(i_wREG = '1' and not (i_ADDR_WR = "00000")) then
        REG(to_integer(unsigned(i_ADDR_WR)))    <=  i_DIN;
    end if;

else    ---if jal, the inferred register is $31, or $ra
    REG(31)    <=  i_DIN;
end if;
```

2.5 Data Memory

2.5.1 RTL Diagram



2.5.2 Description

The `Data Memory` simply holds some data that can be accessed and modified by the address values coming from the `ALU`.

Though the entire memory addressable by the `MIPS` architecture is 2^{32}, it has been cut down to *32* addresses for the sake of simulation and debugging.

The data it can hold is initialized by hand in the design, as in, initial arrays written in the algorithm have to be added by hand on the initializer list of the design.

Though all addresses can be accessed and modified later by the algorithm.

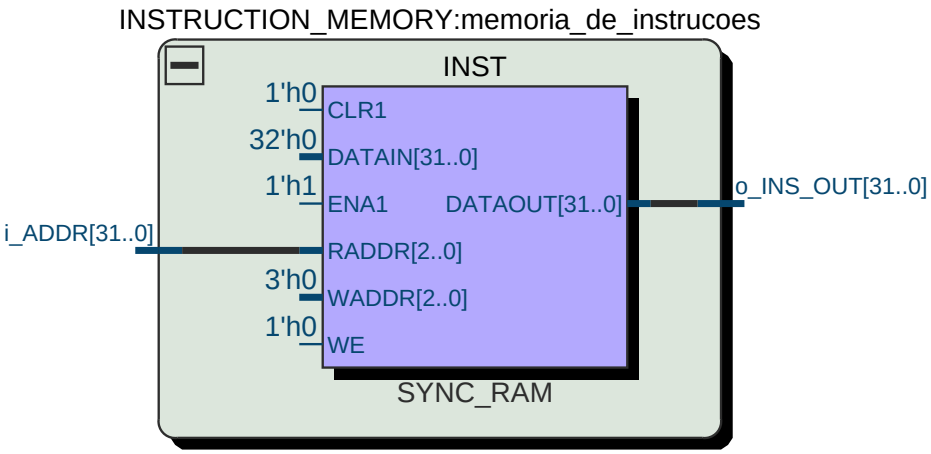
```
        type t_RAM is array ( 0 to 31 ) of std_logic_vector( 31 downto 0 );
signal RAM      :       t_RAM := (
    x"00000000",    ---0x10010000
    x"0000000a",    ---0x10010004
    x"00000014",    ---0x10010008
    x"0000001e",    ---0x1001000c
    x"00000028",    ---0x10010010
    x"00000032",    ---0x10010014
    x"0000003c",    ---0x10010018
    x"00000046",    ---0x1001001c
    x"00000050",    ---0x10010020
    x"0000005a",    ---0x10010024
    x"00000064",    ---0x10010028
    x"0000006e",    ---0x1001002c
    x"00000078",    ---0x10010030
    x"00000082",    ---0x10010034
    x"0000008c",    ---0x10010038
    x"00000096",    ---0x1001003c
    x"00000000",    ---0x10010040
    x"00000000",
    x"00000000",
    ...
    x"00000000",
    x"00000000"
);
```

The address is offset by two bits as the data is a *32* bit vector, and should not be accessed partially.

```
if(i_wMEM = '1') then
    RAM(( to_integer(unsigned(i_ADDR(13 downto 2))))) <= i_DATA_IN;
    ---             address           - offset     / 4
end if;
```

2.6 Instruction Memory

2.6.1 RTL Diagram



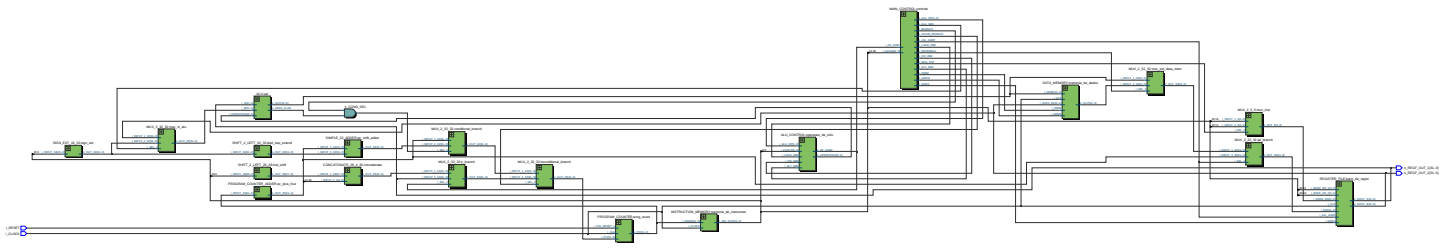
2.6.2 Description

It's a simple *Read Only Memory* that holds the instructions to execute. It's addressed by the `Program Counter`.

```
    --- First Algorithm
type t_RAM is array ( 0 to 6 ) of std_logic_vector( 31 downto 0 );
signal INST     :       t_RAM := (
    x"20120001",    ---main:addi $s2, $zero, 1
    x"3c011001",    ---     lui  $1, 0x000010001
    x"34330000",    ---     ori  $19, $1, 0x0
    x"8e680020",    ---     lw   $t0, 32($s3)     # $t0 = A[8]
    x"02484020",    ---     add  $t0, $s2, $t0    # $t0 = $t0 + h
    x"ae680030",    ---     sw   $t0, 48($s3)     # A[12] = $t
    x"00000000"     ---     nop
);

--- Second Algorithm
type t_RAM is array ( 0 to 8 ) of std_logic_vector( 31 downto 0 );
signal INST     :       t_RAM := (
     x"20100000",    ---main:addi $s0, $zero, 0
     x"20110001",    ---     addi $s1, $zero, 1
     x"20120002",    ---     addi $s2, $zero, 2
     x"20130003",    ---     addi $s3, $zero, 3
     x"20140004",    ---     addi $s4, $zero, 4
     x"12740001",    ---     beq $s3, $s4, L1     # if i==j goto L1
     x"02328020",    ---     add $s0, $s1, $s2    # f = g + h = 1 + 2 = 3
     x"02138022",    ---L1:     sub $s0, $s0, $s3    # f = f - i = 3 - 3 = 0
     x"00000000"
 );

--- Third Algorithm
type t_RAM is array ( 0 to 2 ) of std_logic_vector( 31 downto 0 );
signal INST     :       t_RAM := (
    x"20100001",    ---main:addi $s0, $zero, 1
    x"22100002",    ---loop:addi $s0, $s0, 2
    x"08100001"     ---     j    loop

);
```

3 Top Component

3.1 RTL Diagram



3.2 Description

The `Top` Component is where all other components connect to each other to create the organization. The components before-mentioned will be connected together directly or through multiplexers controlled by *Control Units*.

The `Instruction Memory ROM` has it's output connected to the `Main Control Unit`, the `ALU Control Unit` and the `Register Bank` in several inputs.

The `Register Bank's` inputs are the two *read* addresses for it's outputs, the latter of which is multiplexed to account for the `Register` and `Immediate` instructions that have the destination register located in different parts of the vector.

This multiplexer is controlled by the signal `'s_REG_DST'` that comes from the `Main Control`.

The instruction vector also feeds a sign extender that later connects to the multiplexer connected to the secondary input of the `ALU`, the same is controlled by the signal `'s_ALU_SRC'` from the `Main Control`.

The final five bits of the instruction vector also feed the `ALU Control Unit` to determine wich `Register` based instruction will be executed.

The before-mentioned sign extended vector feeds a shifter to later allow for conditional and unconditional jumps, and feeds the multiplexer to the `ALU` that allows the *immediate* from the instruction to be used in arithmetic, or used as an address for the `Data Memory` the `ALU's` output feeds.

The `Data Memory` has it's input data coming from the secondary output from the `Register File` and it's address from the `ALU`, the output is multiplexed by the `Main Control` with the signal `'s_MEMtoREG'`, the output of the multiplexer feeds the data vector of the `Register File`, which can be a calculation from the `ALU` or data from the memory.

All that's left is some simple arithmetic that controls and calculates the next `Program Counter`.

A simple *32* bit adder calculates the next possible branched address by adding the next address with whatever is in the immediate vector that has been sign extended, if there is a branch and the corresponding signal from the `ALU` will trigger the multiplexer to output this calculation, otherwise it will output the next address.

If however there is an unconditional jump, the part of the vector that holds the new address will be concatenated with part of the first four bits of the current address. If however there is no unconditional branch, the output of the before-mentioned multiplexer will be outputted.

The value of this multiplexer will be given to the `Program Counter` if the reset signal is high.

3.3 Test with Algorithms

3.3.1 Accessing the Data Memory

```
.data
Array_A: .word 0,10,20,30,40,50,60,70,80,90,100,110,120,130,140,150
.text
main:
    addi $s2, $zero, 1    # Inicializa $s2 em 1
    #la   $s3, Array_A
    lui  $1,  0x00001001
    ori  $19, 0x00000000

    lw   $t0, 32($s3)     # $t0 = A[8]
    add  $t0, $s2, $t0    # $t0 = $t0 + h
    sw   $t0, 48($s3)     # A[12] = $t
```

Instruction Memory Rom:

```
type t_RAM is array ( 0 to 6 ) of std_logic_vector( 31 downto 0 );
signal INST     :       t_RAM := (
    x"20120001",    ---main:addi $s2, $zero, 1    # Inicializa $s2 em 1
    x"3c011001",    ---     lui  $1, 0x000010001
    x"34330000",    ---     ori  $19, $1, 0x00000000
    x"8e680020",    ---     lw   $t0, 32($s3)     # $t0 = A[8]
    x"02484020",    ---     add  $t0, $s2, $t0    # $t0 = $t0 + h
    x"ae680030",    ---     sw   $t0, 48($s3)     # A[12] = $t
    x"00000000"     ---     nop
);
```

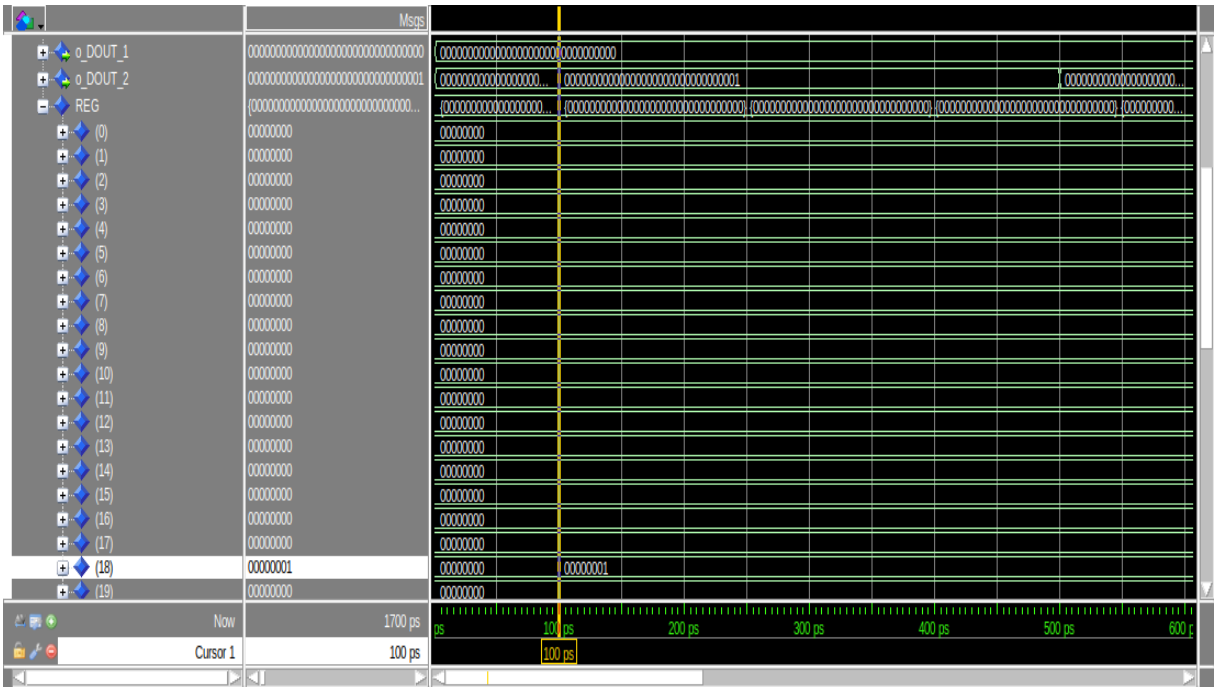
Data Memory initialization list:

```
type t_RAM is array ( 0 to 31 ) of std_logic_vector( 31 downto 0 );
--signal RAM    :       t_RAM := (others=>(others=>'0'));
signal RAM      :       t_RAM := (
    x"00000000",    ---0x10010000
    x"0000000a",    ---0x10010004
    x"00000014",    ---0x10010008
    x"0000001e",    ---0x1001000c
    x"00000028",    ---0x10010010
    x"00000032",    ---0x10010014
    x"0000003c",    ---0x10010018
    x"00000046",    ---0x1001001c
    x"00000050",    ---0x10010020
    x"0000005a",    ---0x10010024
    x"00000064",    ---0x10010028
    x"0000006e",    ---0x1001002c
    x"00000078",    ---0x10010030
    x"00000082",    ---0x10010034
```

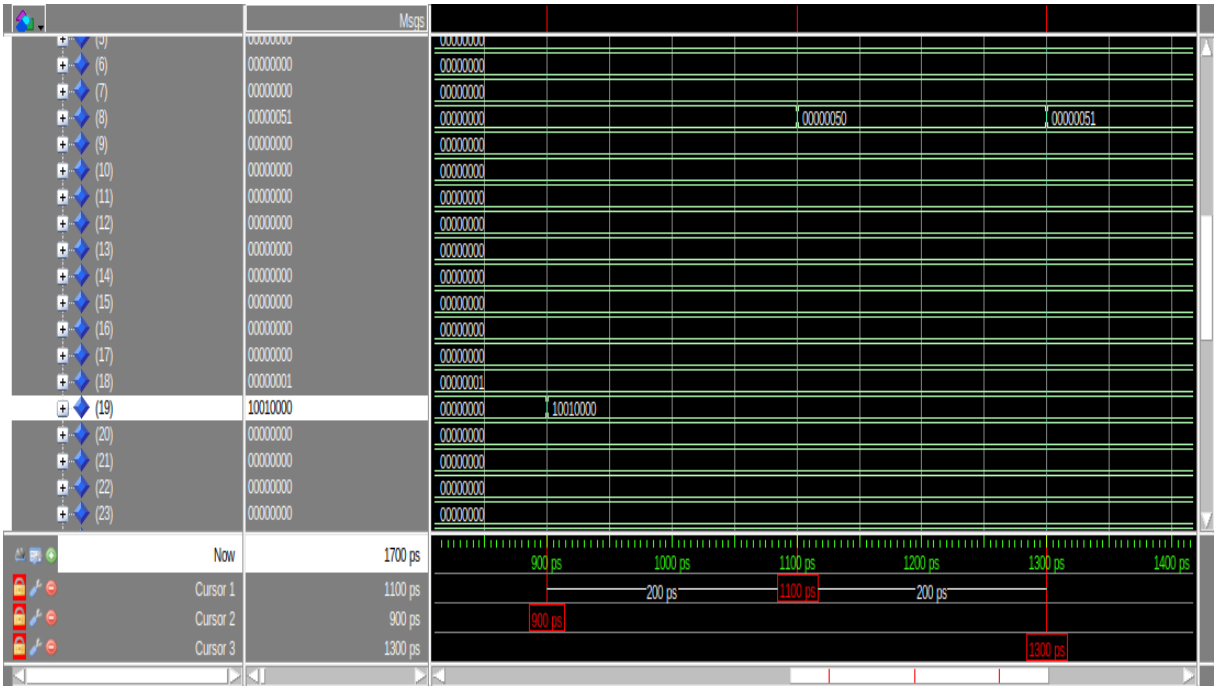
```
    x"0000008c",     ---0x10010038
    x"00000096",     ---0x1001003c
    x"00000000",     ---0x10010040
    ...
    x"00000000",
    x"00000000"
);
```

The algorithm is simple, it loads the eight element of an array, changes it and stores it on another element.

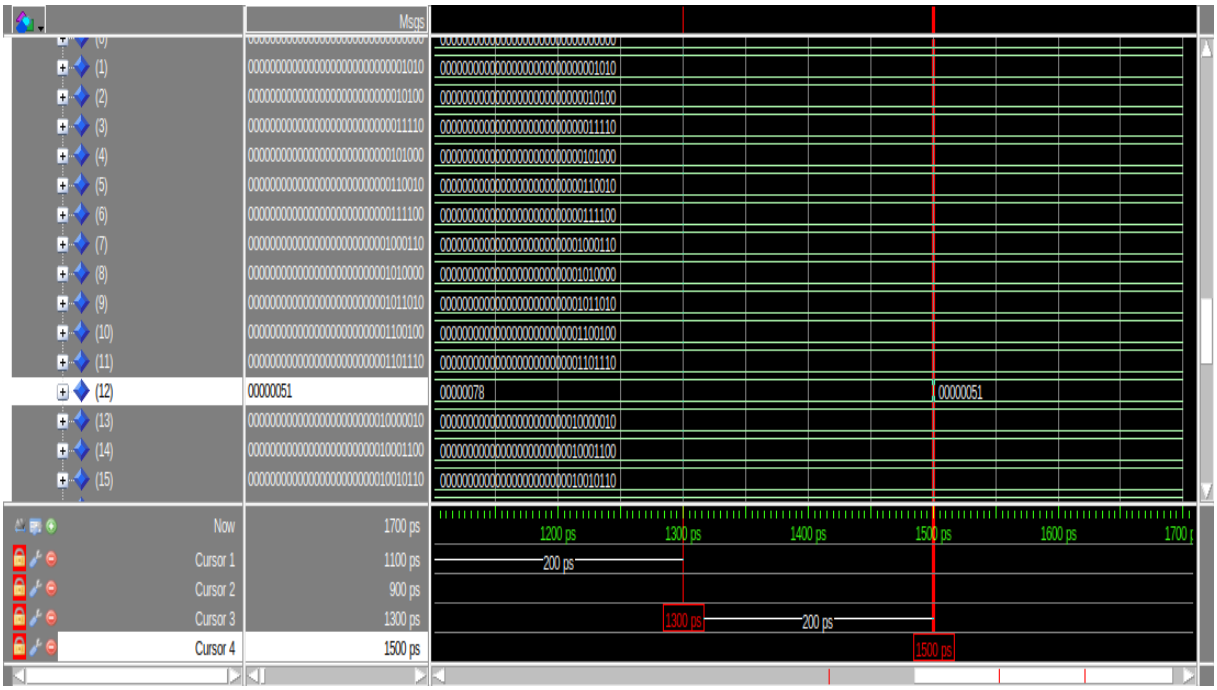
The following images will describe the instructions affecting the complex components only.



Register `$s2`, or `$18` receives the value 1.



Here the other registers receive and change their values, `$19` receives the base address of the array, `$8` receives the data from memory and is changed.



And here the twelfth element is changed.

3.3.2 Conditional Branching

```
 .text
main:
        addi $s0, $zero, 0
        addi $s1, $zero, 1
        addi $s2, $zero, 2
        addi $s3, $zero, 3
        addi $s4, $zero, 4
        beq $s3, $s4, L1     # if i==j goto L1
        add $s0, $s1, $s2    # f = g + h = 1 + 2 = 3
L1:     sub $s0, $s0, $s3    # f = f - i = 3 - 3 = 0
```

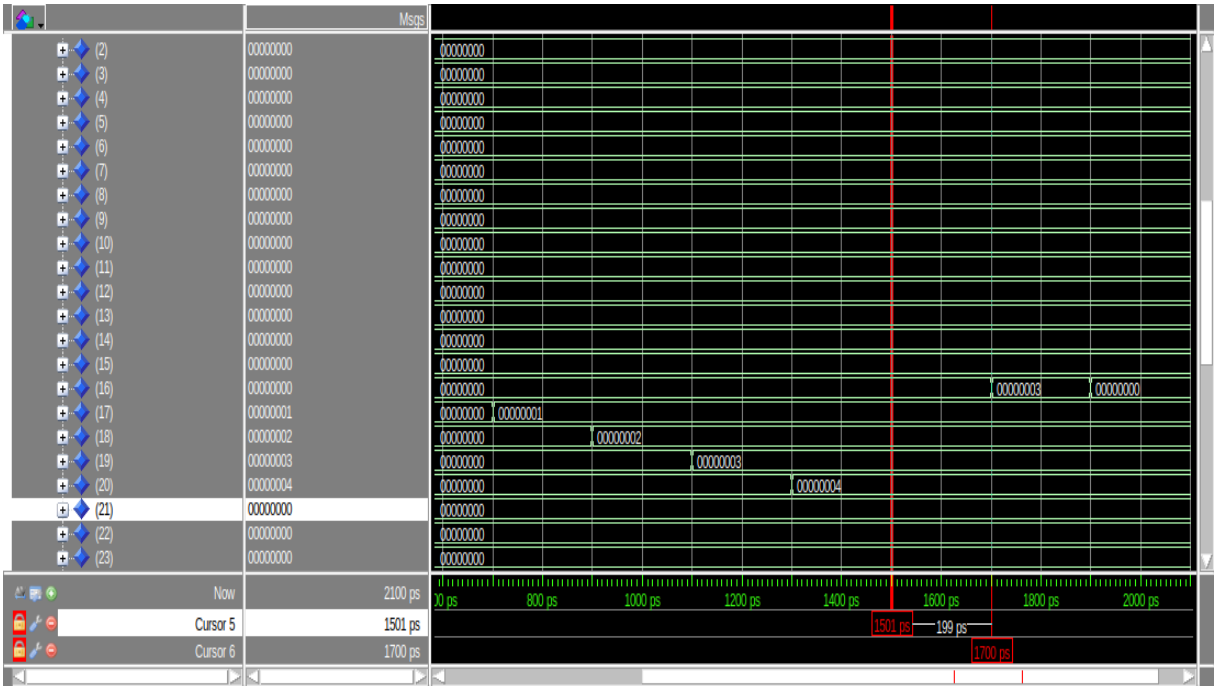
Instruction Memory Rom:

```
type t_RAM is array ( 0 to 8 ) of std_logic_vector( 31 downto 0 );
signal INST     :       t_RAM := (
    x"20100000",    ---main:addi $s0, $zero, 0
    x"20110001",    ---     addi $s1, $zero, 1
    x"20120002",    ---     addi $s2, $zero, 2
    x"20130003",    ---     addi $s3, $zero, 3
    x"20140004",    ---     addi $s4, $zero, 4
    x"12740001",    ---     beq $s3, $s4, L1     # if i==j goto L1
    x"02328020",    ---     add $s0, $s1, $s2    # f = g + h = 1 + 2 = 3
    x"02138022",    ---L1:      sub $s0, $s0, $s3    # f = f - i = 3 - 3 = 0
    x"00000000"
);
```

Data Memory initialization list:

```
type t_RAM is array ( 0 to 31 ) of std_logic_vector( 31 downto 0 );
signal RAM      :       t_RAM := (others=>(others=>'0'));
```

This algorithm simply initializes the registers and checks for a conditional to branch. The branch will not happen however, the actual branches will happen at later algorithms.



The registers are updated and based on the gap between register updates it can be inferred that the jump was not taken, as well by the final value of `$s0` being 0.

3.3.3 Unconditional Branching

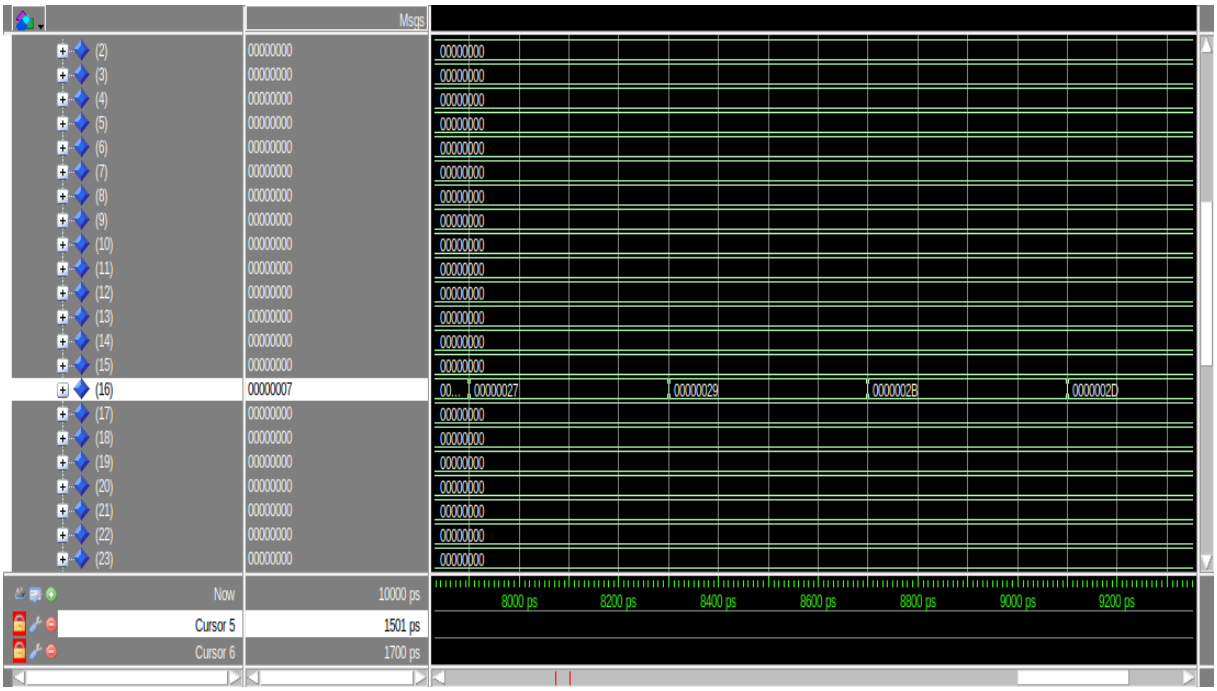
```
.text
main:
        addi $s0, $zero, 1
loop:   addi $s0, $s0, 2
        j    loop
```

Instruction Memory Rom:

```
type t_RAM is array ( 0 to 2 ) of std_logic_vector( 31 downto 0 );
signal INST     :       t_RAM := (
    x"20100001",    ---main:addi $s0, $zero, 1
    x"22100002",    ---loop:addi $s0, $s0, 2
    x"08100001"     ---     j    loop

);
```

Data Memory initialization list:

```
type t_RAM is array ( 0 to 31 ) of std_logic_vector( 31 downto 0 );
signal RAM      :       t_RAM := (others=>(others=>'0'));
```



The algorithm simply adds two to the `$16` register.

3.3.4 Unconditional Branching, Procedures

```
.text   # segmento de c digo (programa)
        j   main
leaf_example:
        add $t0, $a0, $a1   # $t0 = g + h
        add $t1, $a2, $a3   # $t1 = i + j
        sub $v0, $t0, $t1   # f = $t0 - $t1
        jr  $ra             # retorna do procedimento

main:

        addi $a0, $zero, 4  # inicializa 1   par metro (g)
        addi $a1, $zero, 3  # inicializa 2   par metro (h)
        addi $a2, $zero, 2  # inicializa 3   par metro (i)
        addi $a3, $zero, 1  # inicializa 4   par metro (j)
        jal leaf_example    # chama o procedimento
        nop             # n  o faz nada. $v0 tem o resultado do procedimento
```

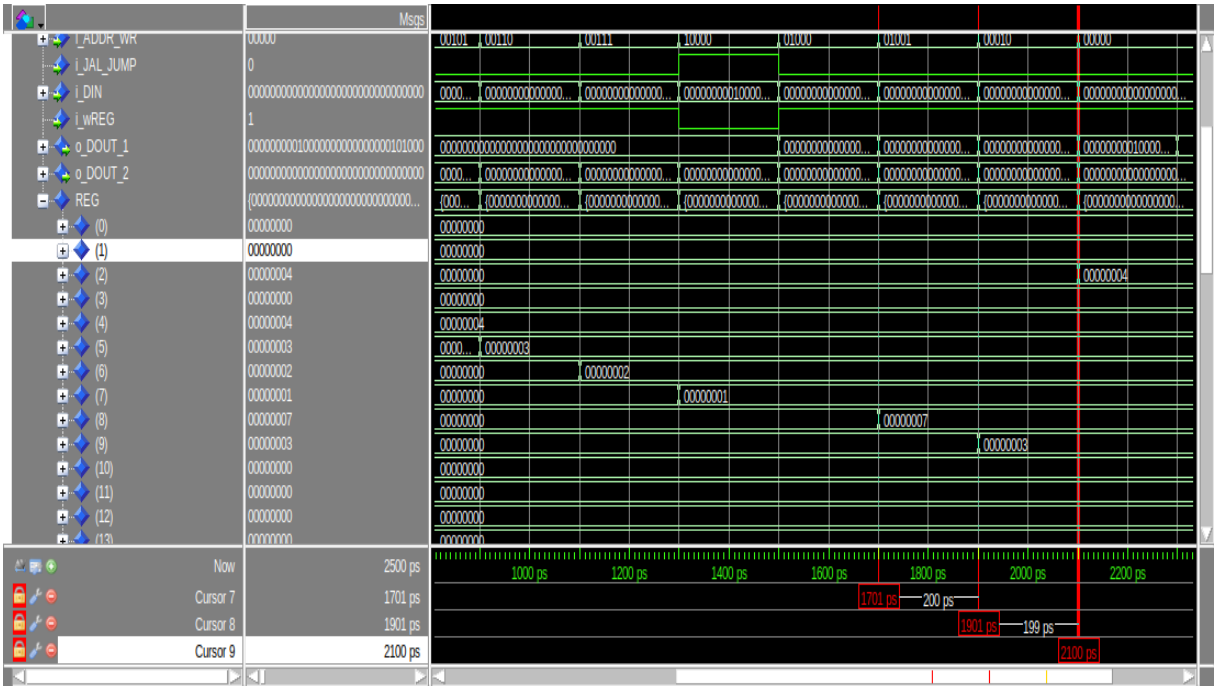
Instruction Memory Rom:

```
type t_RAM is array ( 0 to 7 ) of std_logic_vector( 31 downto 0 );
signal INST     :       t_RAM := (
    x"08100005",        ---             j   main
                    ---leaf_example:
    x"00854020",        ---     add $t0, $a0, $a1   # $t0 = g + h
    x"00c74820",        ---     add $t1, $a2, $a3   # $t1 = i + j
    x"01091022",        ---     sub $v0, $t0, $t1   # f = $t0 - $t1
    x"03e00008",        ---     jr  $ra             # retorna do procedimento
    x"20040004",        ---main:addi $a0, $zero, 4  # inicializa 1 par metro
    x"20050003",        ---     addi $a1, $zero, 3  # inicializa 2 par metro
    x"20060002",        ---     addi $a2, $zero, 2  # inicializa 3 par metro
    x"20070001",        ---     addi $a3, $zero, 1  # inicializa 4 par metro
    x"0c100001",        ---     jal leaf_example    # chama o procedimento
    x"00000000"         ---     nop                 # n  o faz nada.
);
```

Data Memory initialization list:

```
type t_RAM is array ( 0 to 31 ) of std_logic_vector( 31 downto 0 );
signal RAM      :       t_RAM := (others=>(others=>'0'));
```

The algorithm initializes some values in the `S` registers and jumps to a procedure, where a bit of arithmetic is done and the final values is given to the `$v0` register.



The registers pointed by the red cursors subtracted give the final value of 4 as is expected at `$v0`.

4 Final Considerations

Though this processor can be considered unfinished by not having all the instructions implemented, not having a complete `ALU` or having a segmented, unified and complete memory, this processor can be used to understand the real *MIPS* architecture almost to completion.

And although there are other new and superior architectures in the market, there is still good reason to learn *MIPS* as it still holds much of the market and will continue to do so as it's simplicity and time proven design still holds very well to most use cases.

Bibliography

PATTERSON, David A; HENNESSY, John L. Organização e projeto de computadores: a interface(1)hardware/software. Rio de Janeiro, RJ: Elsevier, c2014.