Control Unit

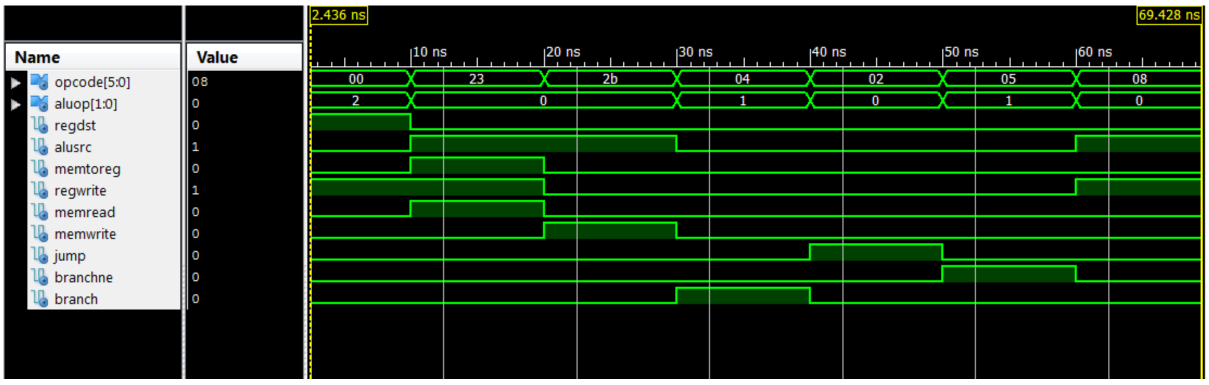
One of the basic modules for a CPU is the control unit. Our task was to design the control unit with the following functions: **sw, lw, beq, bne, j, add, sub, and, or, slt, addi.** The implementation provided by lecturer had almost all the functionality implemented except: **j, bne, addi.** From the MIPS32 instruction sheet we found out the corresponding OP codes for the missing instructions.

Instruction	OP Code (Hex)
J	0x02
Bne	0x05
addi	0x08

J and **addi** was very easy to integrate in the current design. We were able to use the existing hardware design, however for the **bne** instruction we had to add some more components to the design. Basically we added another branch data path, called "BranchNE" to the control unit.

Simulation

After we had described the behaviour of the control unit we simulated it with the OP codes for the previously mentioned instructions.



Result

- 0-10ns is R-type instruction
- 10-20ns is load word instruction
- 20-30ns is store word instruction
- 30-40ns is branch if equals instruction
- 40-50ns is jump instruction
- 50-60ns is branch not equals instruction
- 60-70ns is addi instruction

Conclusion

The simulation of the control unit worked just like expected. All the corresponding bits regarding the OP code were set correctly as you can see in the simulation waveform above.

Arithmetic logic unit (ALU)

After we had a working control unit next task was to implement ALU for all the logical operations regarding registers, like adding, subtracting, anding, oring etc. Again the lecturer provided us with a sample ALU VHDL code. This time the provided code had all the functionality necessary for ALU, so we didn't need to alter it at all. However, we did split the ALU in 3 parts – Sign extender, ALU control unit and ALU, so our final implementation would look like the provided schematic for MIPS32 processor.

ALU control unit

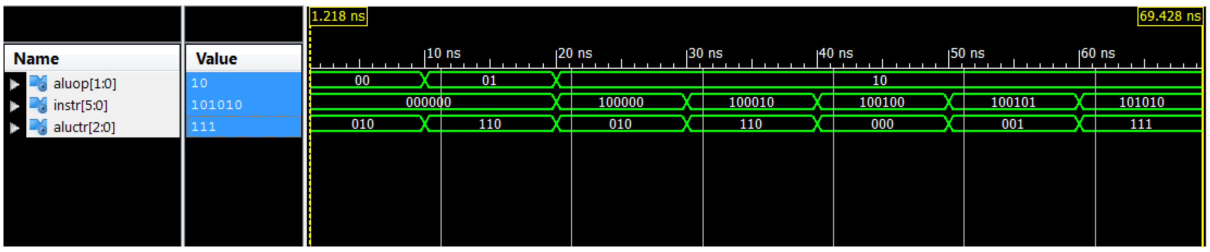
We implemented ALU control unit by following "ControlUnit.docx" document provided by our lecturer. Basically the task of the ALU control unit is to specify what type of arithmetic operation the main ALU has to carry out.

Logical /Arithmetic Operation	ALUCtr[2:0]
and	000
or	001
add	010
sub	110
slt	111

Depending on the ALUOp bus provided by Control Unit and the instruction bus we had to implement the following truth table in ALU control unit

opcode	ALUOp	Operation	instr	ALU function	ALUCtr[2:0]
lw	00	Load word	xxxxxx	Add	010
sw	00	Store word	xxxxxx	Add	010
beq	01	Branch if equal	xxxxxx	subtract	110
R_type	10	Add	100000	Add	010
		subtract	100010	Subtract	110
		AND	100100	AND	000
		OR	100101	OR	001
		SLT	101010	Set on less than	111

Simulation



Results

- 0-10ns load/store word instruction
- 10-20ns branch instruction
- 20-30ns add instruction
- 30-40ns subtract instruction
- 40-50ns AND instruction
- 50-60ns OR instruction
- 60-70ns SLT instruction

Conclusion

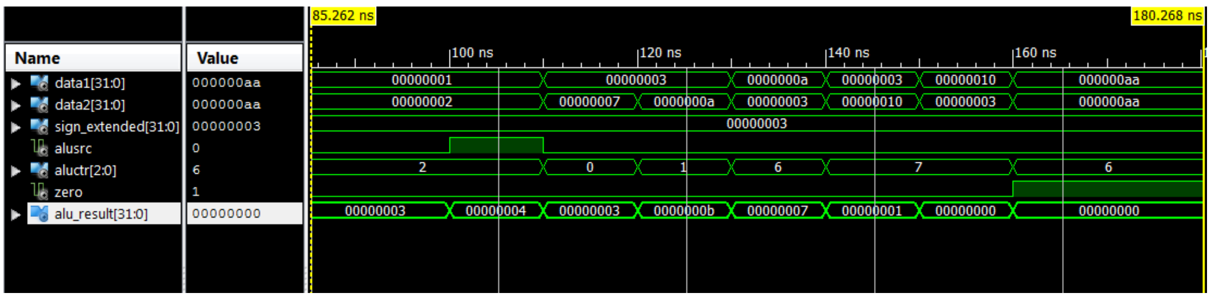
We can confirm that the simulation worked just like expected. The ALU Control unit generated correct output for every input just like described in the truth table.

Main ALU

The actual arithmetic operations are carried out in the main ALU. For our convenience in this module we integrated one of the MUX provided in the schematic. The one which selects between register files “Read Data 2” output and sign extender output. Once again, the code was more or less provided by the lecturer.

Simulation

We simulated whether the integrated MUX and all the arithmetic functions are working in the ALU.



Result

- 85 – 100ns select data2 as input and perform addition
- 100 – 110ns select sign extend as input and perform addition
- 110 – 120ns perform AND operation
- 120 – 130ns perform OR operation
- 130 – 140ns perform subtraction
- 140 – 150ns perform STL operation

- 150-160ns perform STL operation with reversed input
- 160-170ns perform subtraction with identical inputs

Conclusion

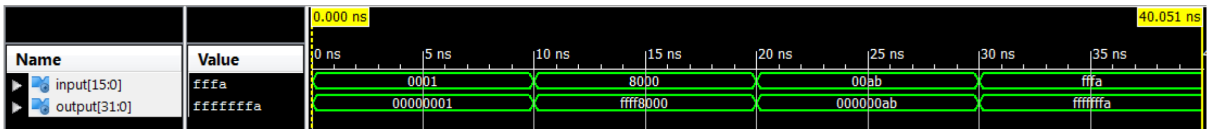
Everything worked just like expected. The ALU carried out all the arithmetic operations correctly, and the integrated MUX was performing correctly as well.

Sign Extender

One last component we have to design is sign extender before we can combine the actual CPU. Out lecturer provided with an example sign extender for 8 bit data bus. Only thing we had to change is the BUS width to 32 bits. For our sign extender we used the IEEE numeric_std library.

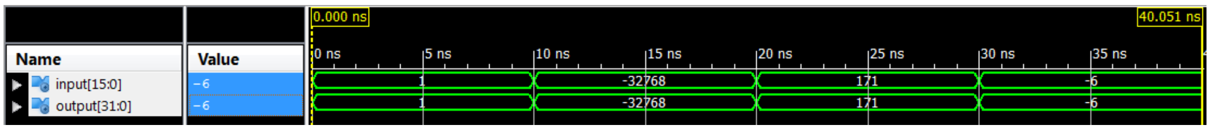
Simulation

Sign extension HEX result



Name	Value	0–10 ns	10–20 ns	20–30 ns	30–40 ns
input[15:0]	fffa	0001	8000	00ab	fffa
output[31:0]	fffffffa	00000001	ffff8000	000000ab	fffffffa

Sign extension Decimal result



Name	Value	0–10 ns	10–20 ns	20–30 ns	30–40 ns
input[15:0]	-6	1	-32768	171	-6
output[31:0]	-6	1	-32768	171	-6

Result

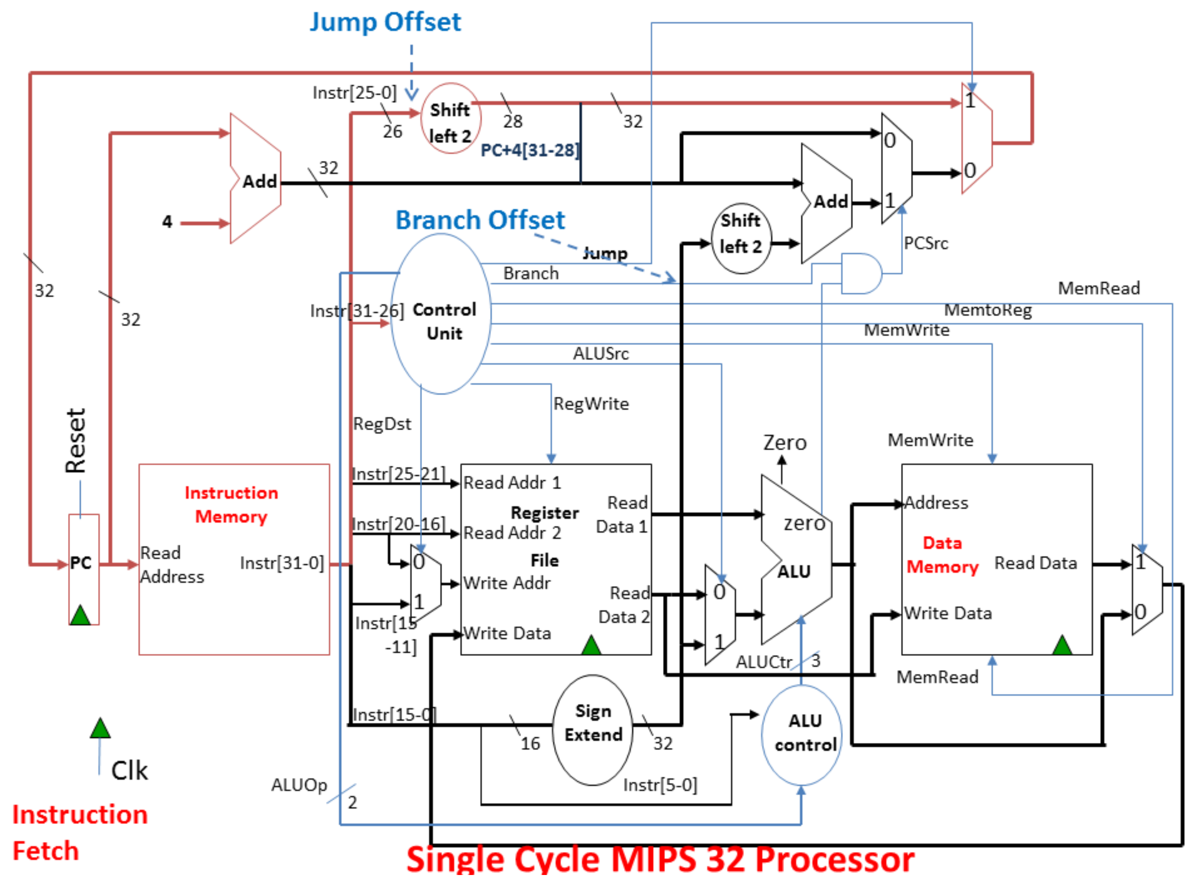
- 0 – 10ns extended number 0x0001 to 0x00000001
- 10 – 20ns extended number 0x8000 to 0xffff8000
- 20 – 30ns extended number 0x00ab to 0x000000ab
- 30 – 40ns extended number 0xfffa to 0xfffffffa

Conclusion

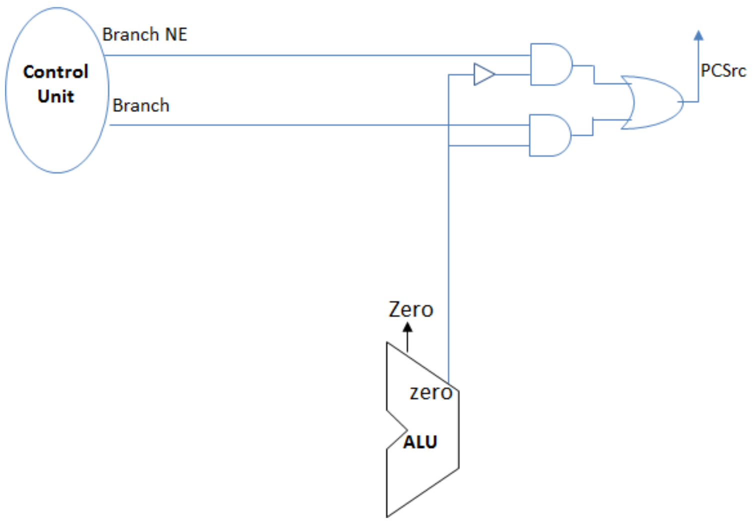
As you can see in the two waveforms shown above, the sign extensions between 2 byte number and 8 byte number is working correctly.

Single Cycle Processor implementation

At this stage we have designed all the necessary components for single cycle processor. Our next and final task is to merge everything together and simulate the CPU. We used the following schematic for our implementation:



However, as previously mentioned we slightly altered the branching to implement the BNE instruction:



Simulation

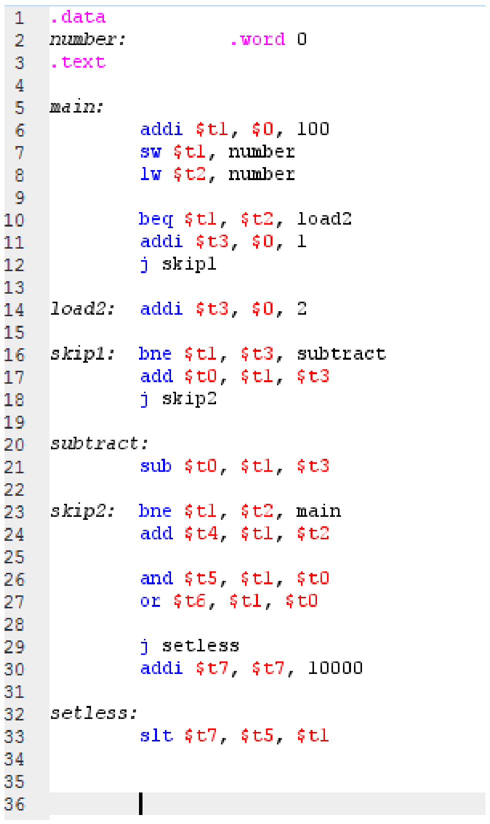
After we assembled the CPU with structural VHDL, we then needed a machine code to test whether our implementation and all the components together are working correctly. We designed a test code on MARS, and then just copied the machine code over to our instruction memory.

Only thing we have to alter is the jump address when executing J instruction. MARS starts the code from address 0x00400000. We start the code from the 0th address. BNE or BEQ works fine because they are not jumping to a specific address, but it alters the PC value by the difference between desired address and current address.

1st test code

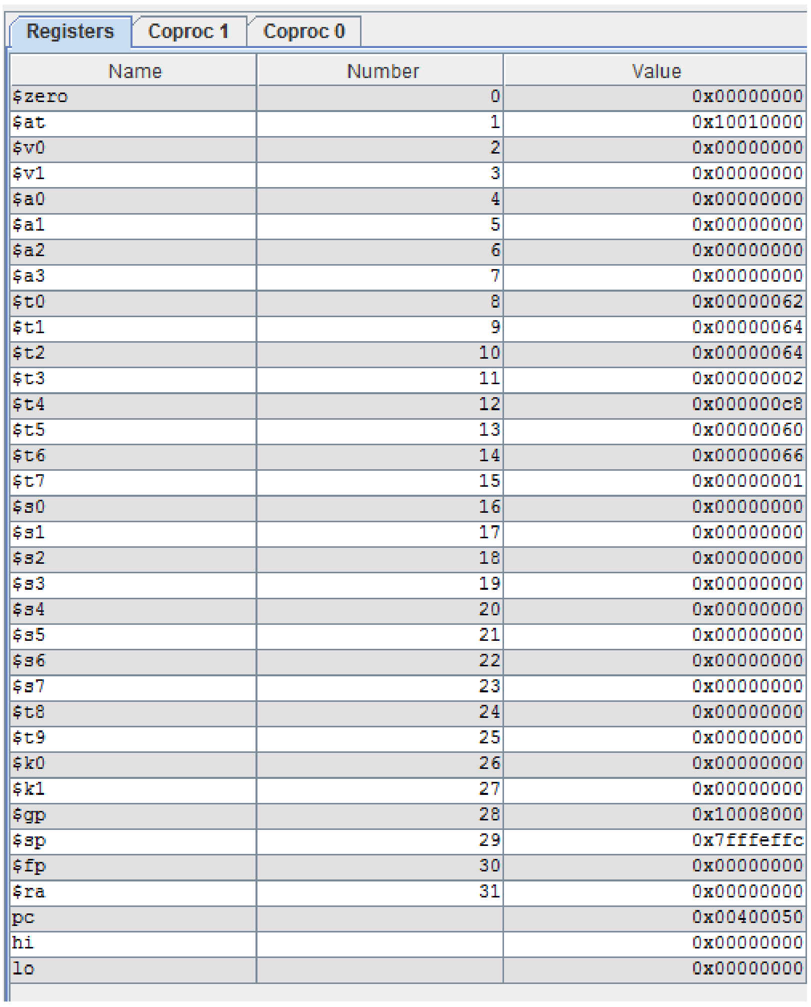
We wrote a small test code which would test all the instructions our implementation can handle.

Code in MARS



```
.data
number:         .word 0
.text

main:
        addi $t1, $0, 100
        sw $t1, number
        lw $t2, number

        beq $t1, $t2, load2
        addi $t3, $0, 1
        j skip1

load2:  addi $t3, $0, 2

skip1:  bne $t1, $t3, subtract
        add $t0, $t1, $t3
        j skip2

subtract:
        sub $t0, $t1, $t3

skip2:  bne $t1, $t2, main
        add $t4, $t1, $t2

        and $t5, $t1, $t0
        or $t6, $t1, $t0

        j setless
        addi $t7, $t7, 10000

setless:
        slt $t7, $t5, $t1


        
```

Register values in MARS after execution

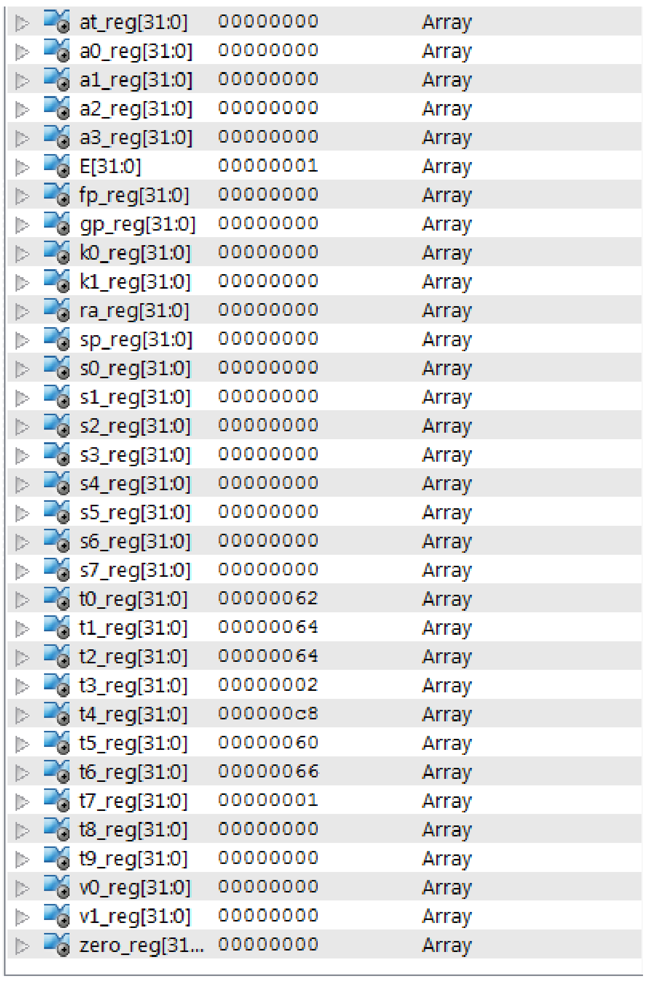


Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000062
$t1	9	0x00000064
$t2	10	0x00000064
$t3	11	0x00000002
$t4	12	0x000000c8
$t5	13	0x00000060
$t6	14	0x00000066
$t7	15	0x00000001
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400050
hi		0x00000000
lo		0x00000000

Machine code in Xilinx

```
X"20090064"   X"ac290028"   X"8c2a0028"   X"112a0002"   X"200b0001"
X"08100009"   X"200b0002"   X"152b0002"   X"012b4020"   X"0810000d"
X"012b4022"   X"152afff2"   X"012a6020"   X"01286824"   X"01287025"
X"08000011"   X"21ef2710"   X"01a9782a"
```

Register values in Xilinx after execution



at_reg[31:0]	00000000	Array
a0_reg[31:0]	00000000	Array
a1_reg[31:0]	00000000	Array
a2_reg[31:0]	00000000	Array
a3_reg[31:0]	00000000	Array
E[31:0]	00000001	Array
fp_reg[31:0]	00000000	Array
gp_reg[31:0]	00000000	Array
k0_reg[31:0]	00000000	Array
k1_reg[31:0]	00000000	Array
ra_reg[31:0]	00000000	Array
sp_reg[31:0]	00000000	Array
s0_reg[31:0]	00000000	Array
s1_reg[31:0]	00000000	Array
s2_reg[31:0]	00000000	Array
s3_reg[31:0]	00000000	Array
s4_reg[31:0]	00000000	Array
s5_reg[31:0]	00000000	Array
s6_reg[31:0]	00000000	Array
s7_reg[31:0]	00000000	Array
t0_reg[31:0]	00000062	Array
t1_reg[31:0]	00000064	Array
t2_reg[31:0]	00000064	Array
t3_reg[31:0]	00000002	Array
t4_reg[31:0]	000000c8	Array
t5_reg[31:0]	00000060	Array
t6_reg[31:0]	00000066	Array
t7_reg[31:0]	00000001	Array
t8_reg[31:0]	00000000	Array
t9_reg[31:0]	00000000	Array
v0_reg[31:0]	00000000	Array
v1_reg[31:0]	00000000	Array
zero_reg[31...	00000000	Array

Register value comparison between MARS and Xilinx

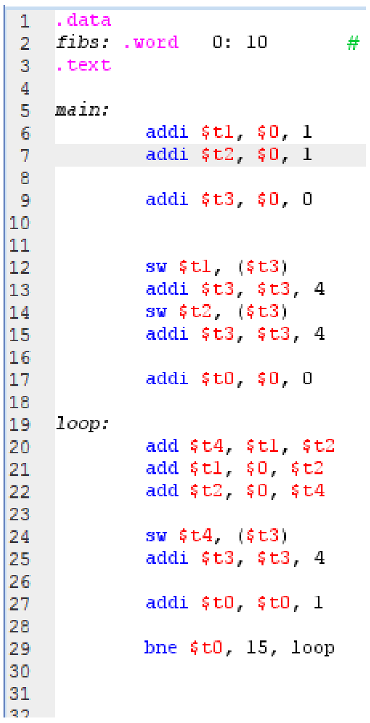
Register	MARS	Xilinx
$t0	0x62	0x62
$t1	0x64	0x64
$t2	0x64	0x64
$t3	0x02	0x02
$t4	0xc8	0xc8
$t5	0x60	0x60
$t6	0x66	0x66
$t7	0x01	0x01

From the first test code we can confirm that all the necessary instructions have been implemented correctly in our CPU design. We weren't confident that jumping and branching instruction will work. Despite our pessimism those two instructions were executed correctly.

2nd test code

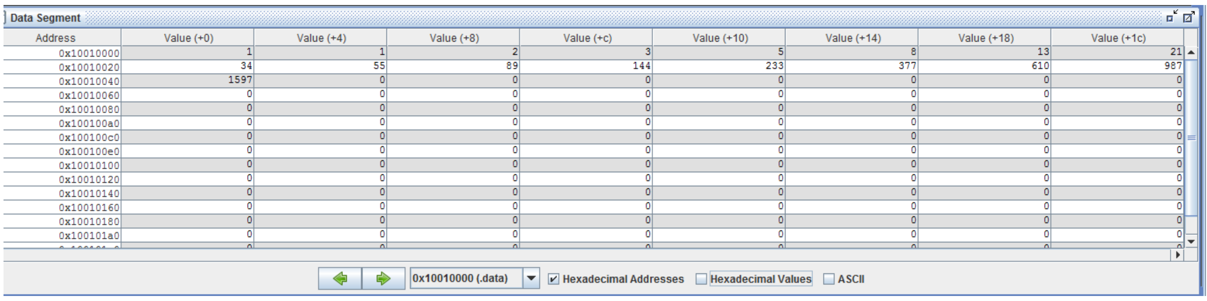
After we confirmed that all of the instructions are executed correctly, we thought of making another test code for calculating Fibannoci sequence and store it into the memory.

Code in MARS



```
.data
fibs: .word   0: 10       #
.text

main:
        addi $t1, $0, 1
        addi $t2, $0, 1

        addi $t3, $0, 0


        sw $t1, ($t3)
        addi $t3, $t3, 4
        sw $t2, ($t3)
        addi $t3, $t3, 4

        addi $t0, $0, 0

loop:
        add $t4, $t1, $t2
        add $t1, $0, $t2
        add $t2, $0, $t4

        sw $t4, ($t3)
        addi $t3, $t3, 4

        addi $t0, $t0, 1

        bne $t0, 15, loop
```

Memory contents in MARS after execution

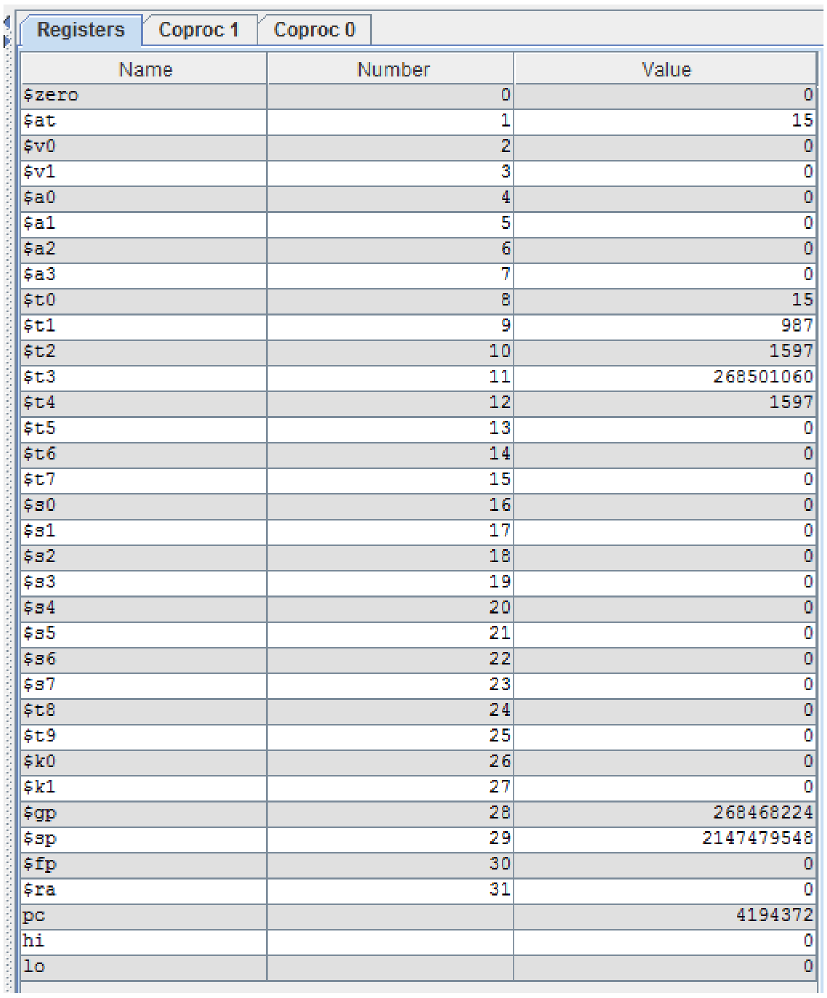


Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	1	1	2	3	5	8	13	21
0x10010020	34	55	89	144	233	377	610	987
0x10010040	1597	0	0	0	0	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0
0x100100e0	0	0	0	0	0	0	0	0
0x10010100	0	0	0	0	0	0	0	0
0x10010120	0	0	0	0	0	0	0	0
0x10010140	0	0	0	0	0	0	0	0
0x10010160	0	0	0	0	0	0	0	0
0x10010180	0	0	0	0	0	0	0	0
0x100101a0	0	0	0	0	0	0	0	0

0x10010000 (.data) [✓ Hexadecimal Addresses] Hexadecimal Values ASCII

Register values in MARS after execution



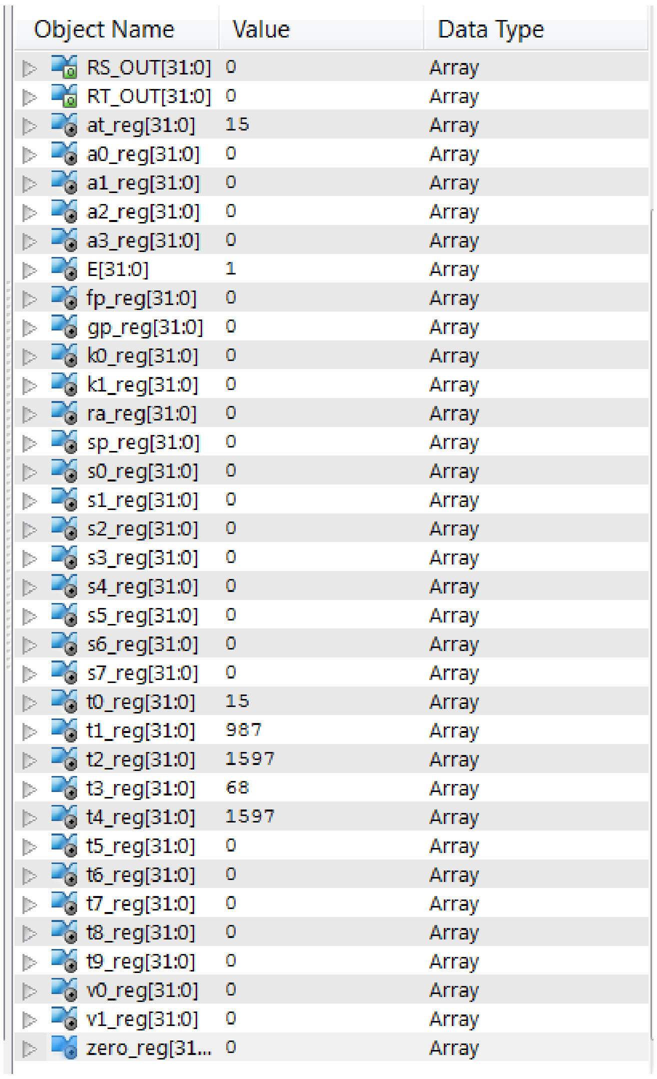
Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	15
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	15
$t1	9	987
$t2	10	1597
$t3	11	268501060
$t4	12	1597
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194372
hi		0
lo		0

Machine code in Xilinx

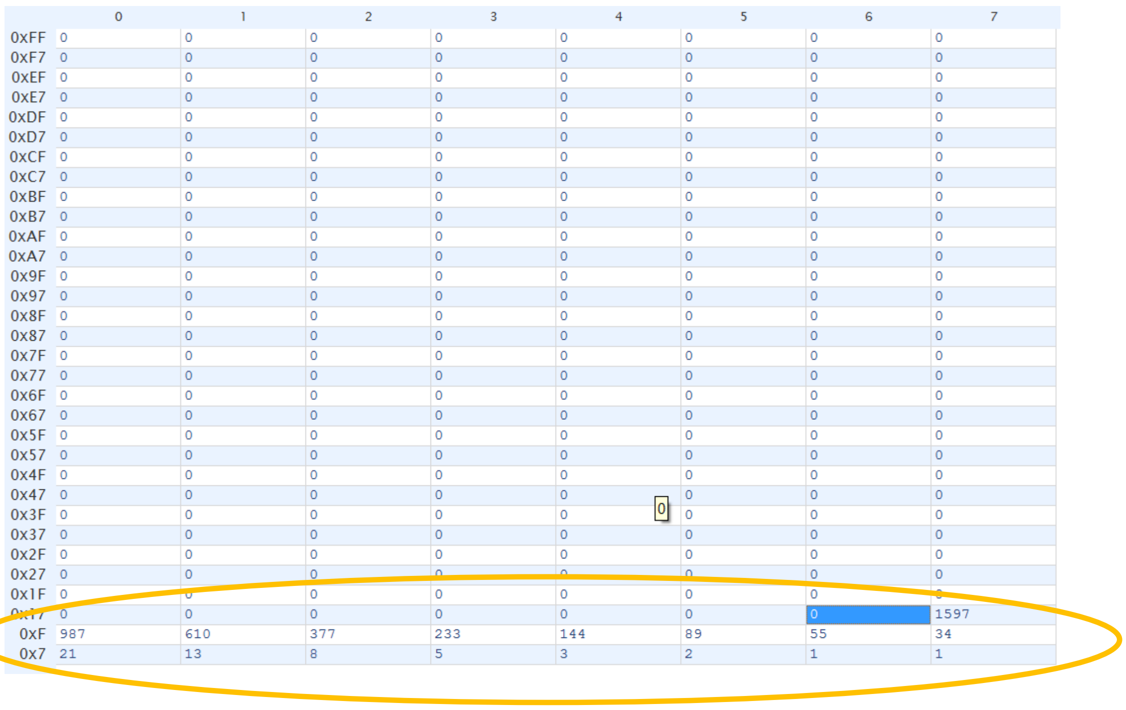
```
X"20090001"   X"200a0001"   X"200b0000"   X"ad690000"   X"216b0004"
X"ad6a0000"   X"216b0004"   X"20080000"   X"012a6020"   X"000a4820"
X"000c5020"   X"ad6c0000"   X"216b0004"   X"21080001"   X"2001000f"
X"1428fff8"
```

Register values in Xilinx after execution



Object Name	Value	Data Type
RS_OUT[31:0]	0	Array
RT_OUT[31:0]	0	Array
at_reg[31:0]	15	Array
a0_reg[31:0]	0	Array
a1_reg[31:0]	0	Array
a2_reg[31:0]	0	Array
a3_reg[31:0]	0	Array
E[31:0]	1	Array
fp_reg[31:0]	0	Array
gp_reg[31:0]	0	Array
k0_reg[31:0]	0	Array
k1_reg[31:0]	0	Array
ra_reg[31:0]	0	Array
sp_reg[31:0]	0	Array
s0_reg[31:0]	0	Array
s1_reg[31:0]	0	Array
s2_reg[31:0]	0	Array
s3_reg[31:0]	0	Array
s4_reg[31:0]	0	Array
s5_reg[31:0]	0	Array
s6_reg[31:0]	0	Array
s7_reg[31:0]	0	Array
t0_reg[31:0]	15	Array
t1_reg[31:0]	987	Array
t2_reg[31:0]	1597	Array
t3_reg[31:0]	68	Array
t4_reg[31:0]	1597	Array
t5_reg[31:0]	0	Array
t6_reg[31:0]	0	Array
t7_reg[31:0]	0	Array
t8_reg[31:0]	0	Array
t9_reg[31:0]	0	Array
v0_reg[31:0]	0	Array
v1_reg[31:0]	0	Array
zero_reg[31...	0	Array

Memory contents in Xilinx after execution



	0	1	2	3	4	5	6	7
0xFF	0	0	0	0	0	0	0	0
0xF7	0	0	0	0	0	0	0	0
0xEF	0	0	0	0	0	0	0	0
0xE7	0	0	0	0	0	0	0	0
0xDF	0	0	0	0	0	0	0	0
0xD7	0	0	0	0	0	0	0	0
0xCF	0	0	0	0	0	0	0	0
0xC7	0	0	0	0	0	0	0	0
0xBF	0	0	0	0	0	0	0	0
0xB7	0	0	0	0	0	0	0	0
0xAF	0	0	0	0	0	0	0	0
0xA7	0	0	0	0	0	0	0	0
0x9F	0	0	0	0	0	0	0	0
0x97	0	0	0	0	0	0	0	0
0x8F	0	0	0	0	0	0	0	0
0x87	0	0	0	0	0	0	0	0
0x7F	0	0	0	0	0	0	0	0
0x77	0	0	0	0	0	0	0	0
0x6F	0	0	0	0	0	0	0	0
0x67	0	0	0	0	0	0	0	0
0x5F	0	0	0	0	0	0	0	0
0x57	0	0	0	0	0	0	0	0
0x4F	0	0	0	0	0	0	0	0
0x47	0	0	0	0	0	0	0	0
0x3F	0	0	0	0	0	0	0	0
0x37	0	0	0	0	0	0	0	0
0x2F	0	0	0	0	0	0	0	0
0x27	0	0	0	0	0	0	0	0
0x1F	0	0	0	0	0	0	0	0
0x17	0	0	0	0	0	0	0	1597
0xF	987	610	377	233	144	89	55	34
0x7	21	13	8	5	3	2	1	1

Summary

As you can see in our results, both MARS and Xilinx generated the same Fibannoci sequence up to element 17. That means iterative algorithms are working in our single cycle processor implementation. Only problem we encountered was, that the simulation by default runs for 1 µS, but to generate 17 elements it needs 1.375 µS. Other than that we didn't have any problems at all.

Conclusion

We successfully managed to implement a working Single cycle processor. For us to be able to accomplish this task we learned a lot of very useful information about how the CPU works. Now we have much better understanding how the CPU works which includes: register file, instruction fetcher, memory, control unit and ALU. We still can further improve this implementation with pipelining and multiple cycles. Current design nowadays is considered to be very slow, since the clock cycle can't be faster than the slowest instruction, which is SW instruction for our case. Overall, we learned a lot from this coursework and we feel satisfied with what we can now do, and we hope to further improve our understanding how the CPU works with multiple cores.