The American University in Cairo

Computer Architecture

Project 1 (MS3)

Pipeline scalar Implementation of RISC-V

Ahmed Osama Hassan

900171850

Mohamed Ashraf Hemdan

900171923

Monday , July 6, 2020

Submitted to:

Dr. Cherif Salama

In this mile stone we try to improve the execution time of the processor, the execution time is given as the product of three important factors, namely IC, CPI and Tc (instruction count, cycles per instructions and time of clock cycle respectively). The instruction count is not dependent on the internal design of the processor as much as it depends on the nature of the program the runs on the processor itself hence in our pursuit of a smaller execution time the focus will mainly be on CPI and Tc. Therefore the value of CPI and Tc has to be reduced to the possible minimum to maintain this small execution time through implementing different microarchitectures other than the single cycle. The other three options would be:

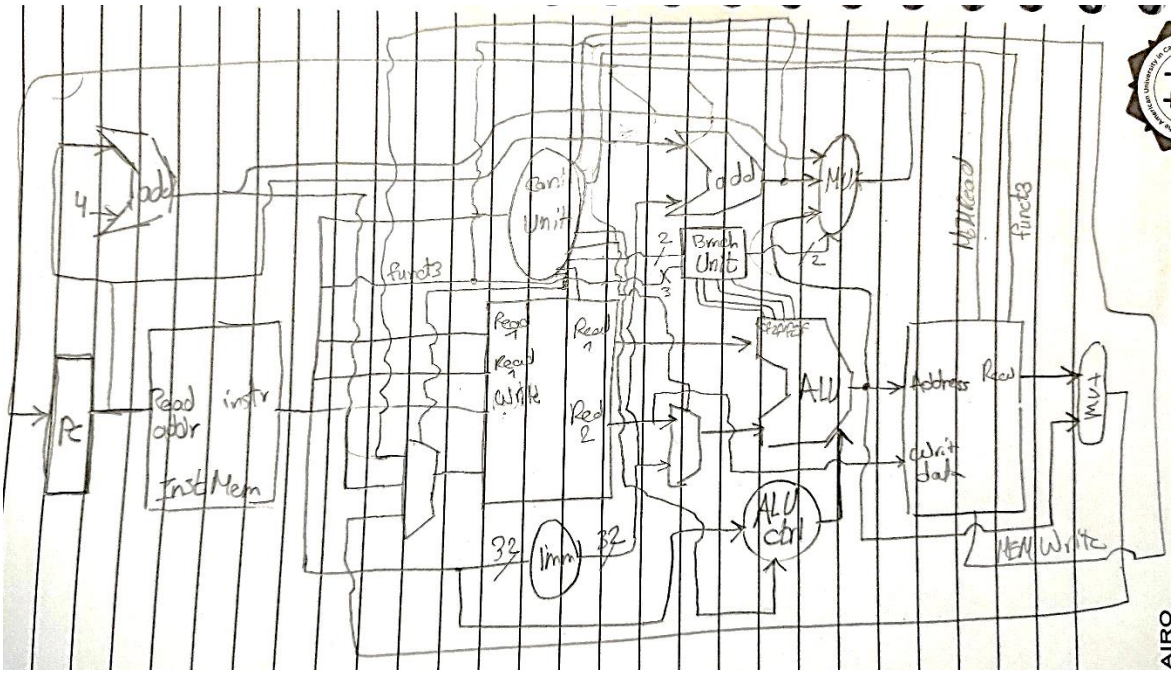
1) Multi-cycle MA: this will reduce the already long Tc of the single cycle (since Tc in single cycle encompasses all stages of instruction); however, it will increase the CPI which will greatly affect the execution time and will not enhance the performance to full potential.
2) Pipelined (scalar) MA: this will reduce the long Tc of the single cycle since the Tc will transform from the longest instruction delay to the longest stage delay which is bound to reduce Tc to nearly one fifth of its value in case of risc v, The CPI in this MA option will not change since after a period of heating up the pipeline will be finishing one instruction per cycle so the CPI is nearly one which shall enhance the performance greatly. Minor hardware resources have to be added though to deal with some implications that using this option may incur.
3) Pipelined (super scalar): this will reduce both the Tc , like scalar, and CPI. This option seems the most effective since it reduces the CPI to less than 1 as well as have a small Tc. However the techniques used to implement such processor needs more concepts and deeper knowledge of processor designs.

From the previous three solutions, the solution to be implemented in this mile stone to decrease execution time is going to be the pipelined (scalar) MA since it has an excellent execution time, as compared to the single and multi-cycle MAs; in addition to its ease of implementation, as compared to the super scalar which requires more knowledge about other concepts.

This design also uses a signle-single ported memory for both instructions and data since this is more close to the actually implementation of physical memory, not cache, according to the stored-program concept.

Previous design:

This design implements the 40 instructions of the risc v instruction set under the single cycle micro architecture:



Design Challenges and solutions

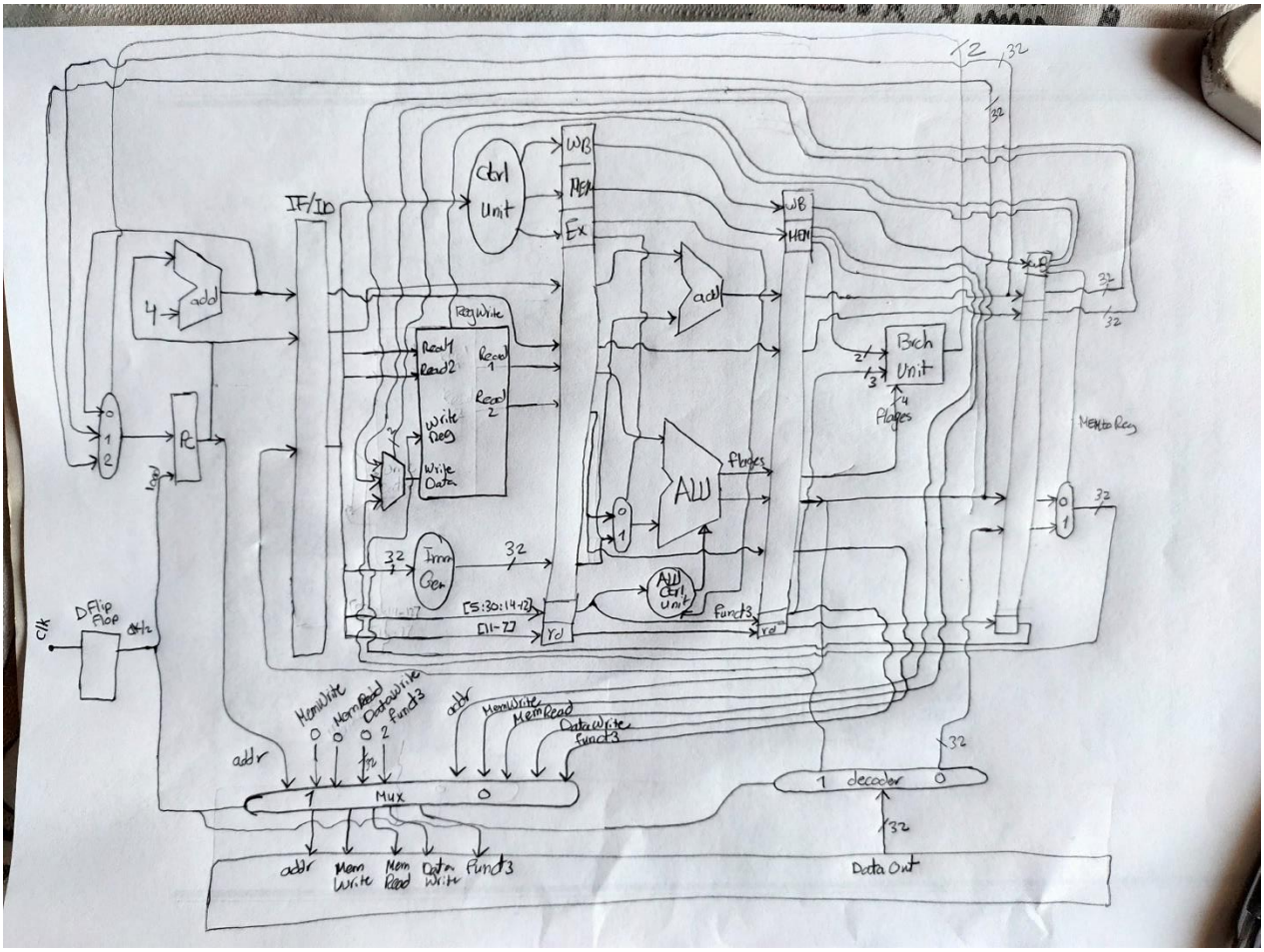
1- Pipelined data path
   The new design has to reduce Tc by making each cycle corresponds to a stage finished in the pipeline of each instruction, to do so the data from the previous stage has to be preserved to the next stage by some means type of memory.

2- Single-Single ported memory
   The new memory module has to accept both instructions and data which might cause some problems in the pipepline since the same hardware resource, memory, might be used twice either in fetching or in memory stage of instruction execution giving rise to a structural hazard that has to be handled.

Design Changes:

1- Addition of a four pipeline registers.
   Four different registers were implemented to divide the stages of the processor, to keep the signals necessary for continuing the execution alive while other signals from other instructions are being fetched, decoded, executed, etc. The different modules have to be divided into different stages in the sense of taking inputs from a certain register and providing output to another, The ALU for example takes input from ID_EX register and produces output to EX_MEM thus is in the execution stage of the pipeline.
2- New Memory unit.
   The new memory unit needs to be able to handle different modes of addressing as well as being able to distinguish the data input from the instruction input as well as allowing only the data input to have these different addressing modes, also the design of the processor has to take into consideration the memory structural hazard by applying different clocks to different pipeline registers.

New Data Path:



Testing Procedures:

We used several test programs to test our processor. All of them have NOP instruction since we are implementing a pipeline processor with no forwarding or hazard units implemented.

Tests:

1) First test: checks the different branching instructions as well as the jal and jalr

First Test: Simple test over (add, lw, beq instructions only)

addi x1, x0, 4
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
	0x00400093
	0x00000033
	0x00000033
	0x00000033

addi x3, x1, 4
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
	0x00408193
	0x00000033
	0x00000033
	0x00000033

addi x4, x0, 20
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
	0x01400213
	0x00000033
	0x00000033
	0x00000033

addi x5, x0, -5
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
	0xffb00293
	0x00000033
	0x00000033
	0x00000033

addi x8, x0, 26
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
	0x01a00413
	0x00000033
	0x00000033
	0x00000033

jal x7, function #checking jal add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	0x024003ef 0x00000033 0x00000033 0x00000033

addi x1, x1, 1
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
	0x00108093
	0x00000033
	0x00000033
	0x00000033

addi x1, x1, 2 # checking offset of jalr # end of program	0x00208093

function:
addi x2, x0, 1
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
bigger_than:
	0x00100113
	0x00000033
	0x00000033
	0x00000033

addi x3, x3, 3
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
	0x00318193
	0x00000033
	0x00000033
	0x00000033

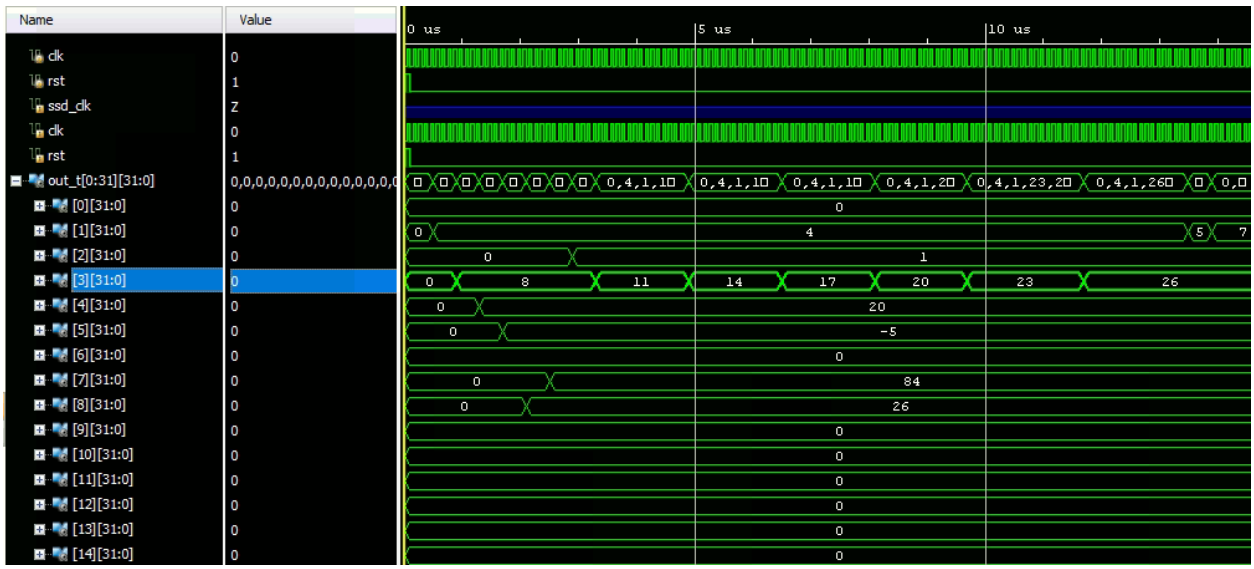
blt x4, x3, less_than #checking blt works correctly add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	0x02324863 0x00000033 0x00000033 0x00000033
bge x5, x3, bigger_than # checking BGE vs BGEU add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	0xfe32d0e3 0x00000033 0x00000033 0x00000033
bgeu x5, x3, bigger_than add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	0xfc32f8e3 0x00000033 0x00000033 0x00000033
less_than: beq x3, x8, end #check beq add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	0x02818063 0x00000033 0x00000033 0x00000033
bne x1, x2, function #checking bne and backward jumps add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	0xfa2090e3 0x00000033 0x00000033 0x00000033
end: jalr x0, x7, 4 #cheking jalr add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	0x00438067 0x00000033 0x00000033 0x00000033

Mem[0]	0
Mem[1]	0
Mem[2]	0
Mem[3]	0
Mem[4]	0
Mem[5]	0
Mem[6]	0
Mem[7]	0
Mem[8]	0
Mem[9]	0
Mem[10]	0
Mem[11]	0

This is a test that doesn't use the memory at data at all hence all data memories are initialized to zeros

Note all of these tests have been conducted to the pipelined processor before implementing the single processor.

Here are some screenshots from the simulation of the first test:



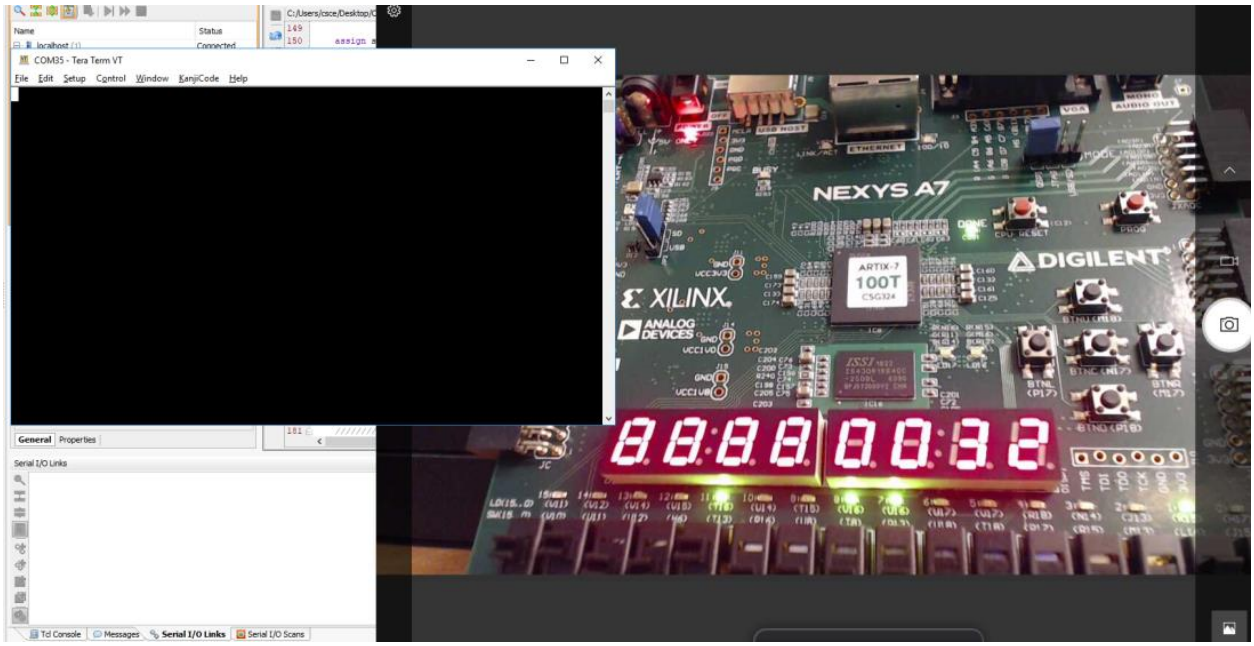
This shows the values resulting from the execution of the previous program note that the value of x3 reaches 26 then it finally equals x8 and the instruction beq get it out of the loop to return from the function (using the jalr instruction), before that the pc is trapped in the loop by different branch instructions and all of them are tested to prove that the whole spectrum of branches is working correctly under pipeline implementations, the jal is also tested since it was used to jump to the function.

Here is some screenshots from the processor doing some instructions

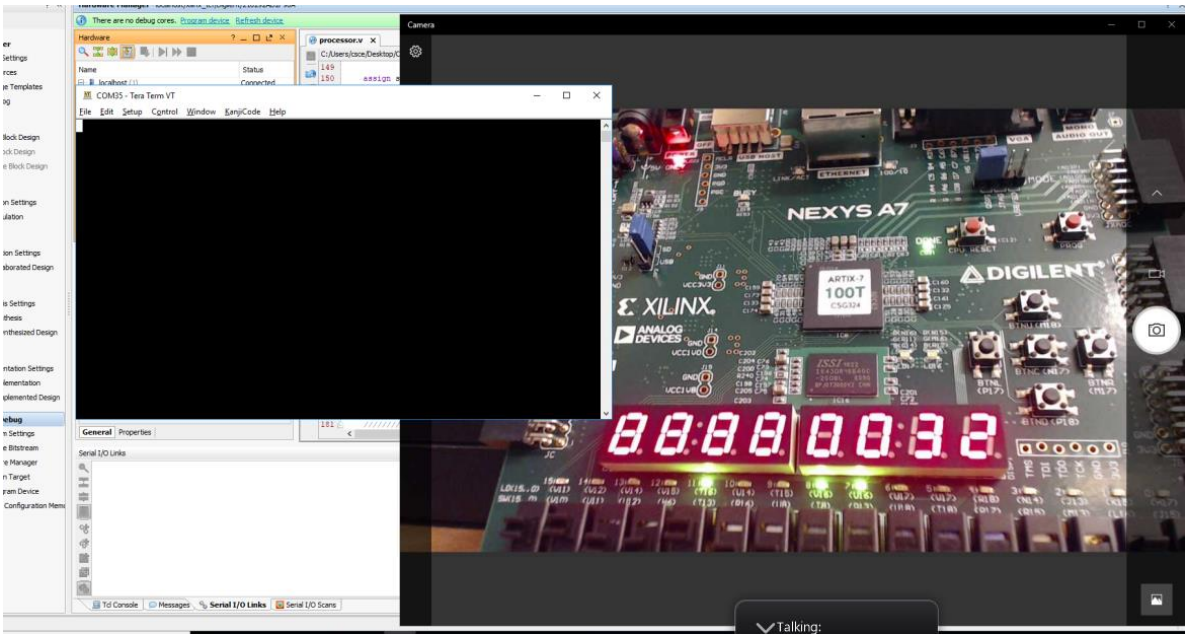
JAL instruction: `jal x7, function`

This instruction is from test 1

The ssd-sel is 0010 which shows the pc_branch output of the jal instruction or in other words the address of the instruction it is going to jump into



The pc has jumped in the next cycle as you can see below, the ssd-sel is 0000 which shows the value of the current pc, so the jal has executed successfully.



2) Second test:

The following test was mainly designed to test the R-format instructions as well as LUI and AUIPC instructions, plus the different addressing modes of the memory (Lw, lh, …etc) , also shift right logical and arithmetic are checked as well as the logical instructions like stl

`lui x1, 1256`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
	004e80b7
	00000033
	00000033
	00000033

`ori x1, x1, 48`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
	0300e093
	00000033
	00000033
	00000033

`auipc x2, 7934`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
	01efe117
	00000033
	00000033
	00000033

`lb x3, 0(x0)`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
	00000183
	00000033
	00000033
	00000033

`lh x4, 0(x0)`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
	00001203
	00000033
	00000033
	00000033

`lw x5, 0(x0)`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
	00002283
	00000033
	00000033
	00000033

`sb x5, 4(x0)`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
	00500223
	00000033
	00000033
	00000033

`sh x5, 8(x0)`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
	00501423
	00000033
	00000033
	00000033

sw x5, 12(x0)
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
	00502623
	00000033
	00000033
	00000033

add x2, x1, x0
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
	00008133
	00000033
	00000033
	00000033

sub x2, x2, x1
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
	40110133
	00000033
	00000033
	00000033

sll x6, x5, x3 add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	00329333 00000033 00000033 00000033
srl x7, x5, x3 add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	0032d3b3 00000033 00000033 00000033

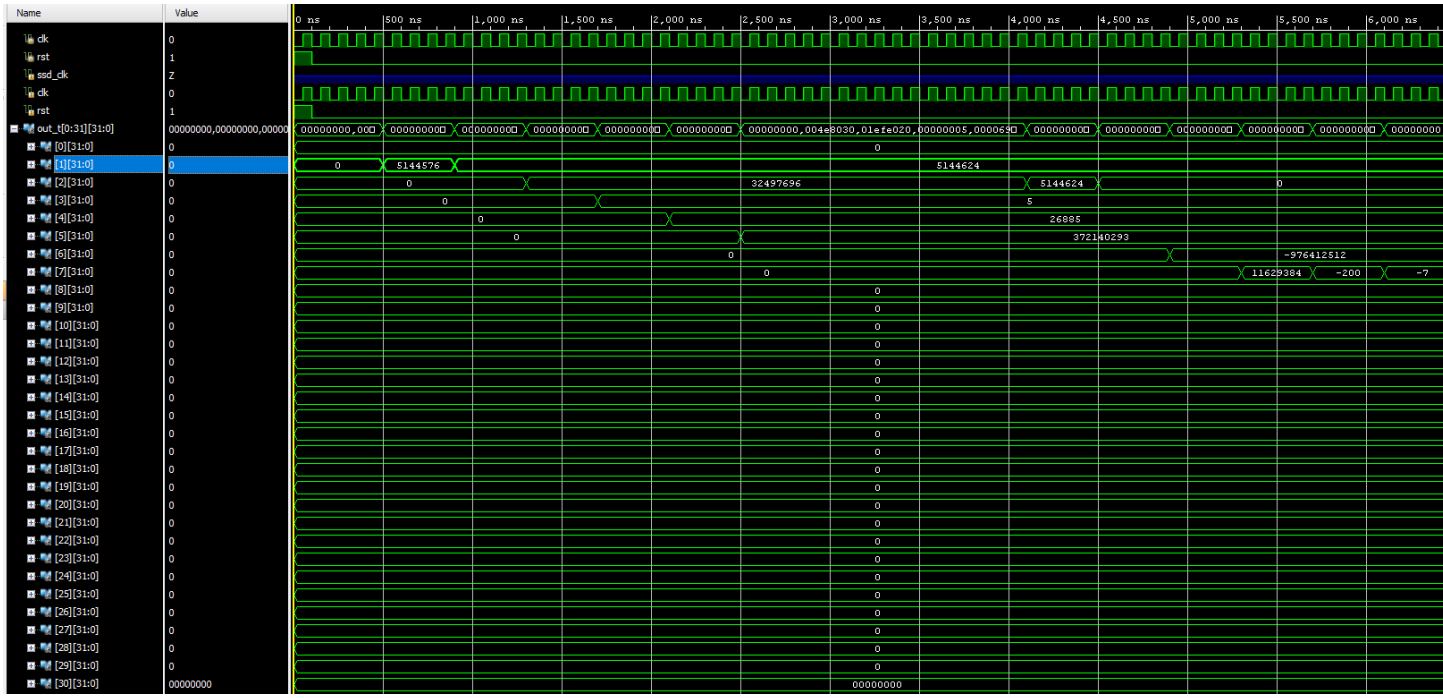
addi x7, x0, -200
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
	f3800393
	00000033
	00000033
	00000033

sra x7, x7, x3 add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	4033d3b3 00000033 00000033 00000033
slt x8, x2, x7 add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	00712433 00000033 00000033 00000033
sltu x9, x6, x7 add x0, x0, x0 add x0, x0, x0 add x0, x0, x0	007334b3 00000033 00000033 00000033

`xor x9, x6, x6`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
 	
`addi x6, x0, -1`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
 	
`or x9, x1, x6`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
 	
`and x1 ,x1 , x0`
`add x0, x0, x0`
`add x0, x0, x0`
`add x0, x0, x0`
	006344b3
	00000033
	00000033
	00000033
	
	fff00313
	00000033
	00000033
	00000033
	
	0060e4b3
	00000033
	00000033
	00000033
	
	0000f0b3
	00000033
	00000033
	00000033

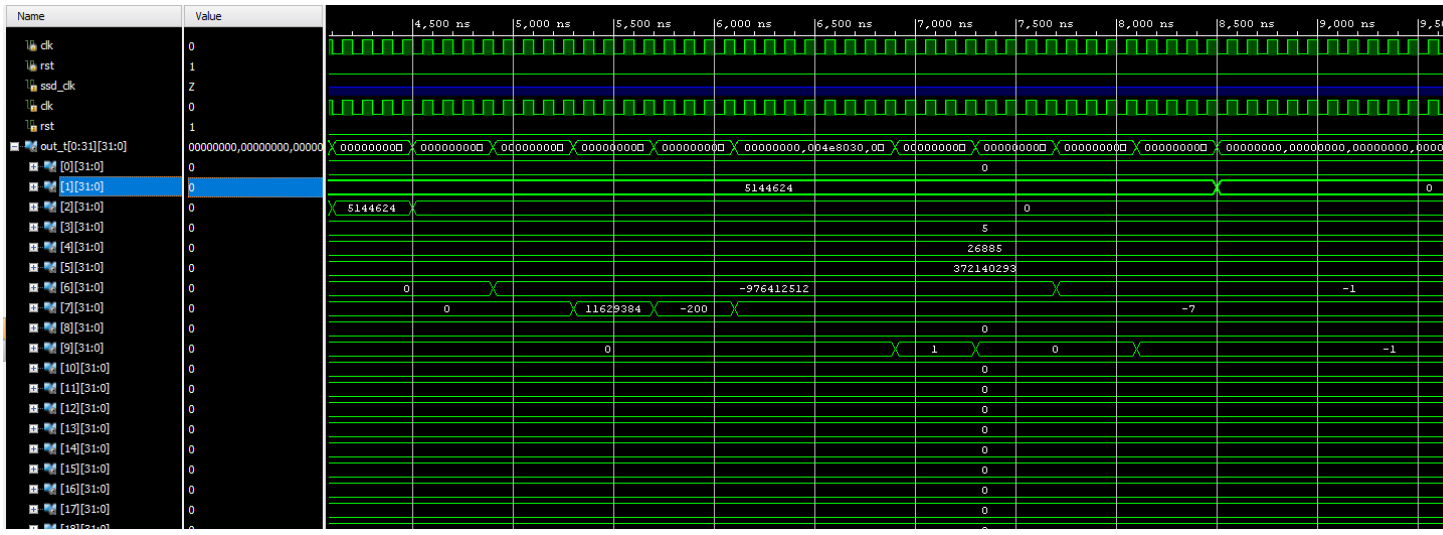
Mem[0]	8’d5
Mem[1]	8’d105
Mem[2]	8’d46
Mem[3]	8’d22
Mem[4]	0
Mem[5]	0
Mem[6]	0
Mem[7]	0
Mem[8]	0
Mem[9]	0
Mem[10]	0
Mem[11]	0

Here are some screenshots from the simulation of the second test:



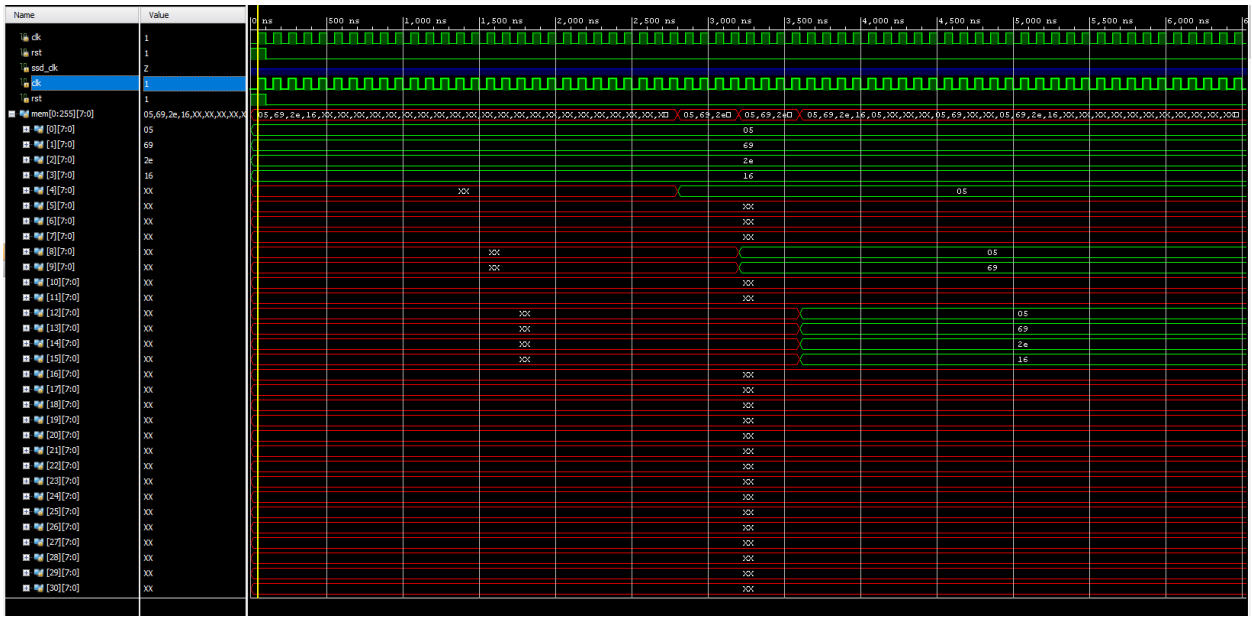
In the first set of instructions one can see that the LUI, ori and AUIPC instructions are working fine

One can also see that the sll and srl are working okay, the output of the rest of the instuctions is seen next



Here the logical instructions like slt and sltu are tested to prove they are working fine, finally some the rest of the arithmetic operations instructions are tested.

This is the memory (data) values after the program has finished executing so as to prove the L/S instructions are working correctly

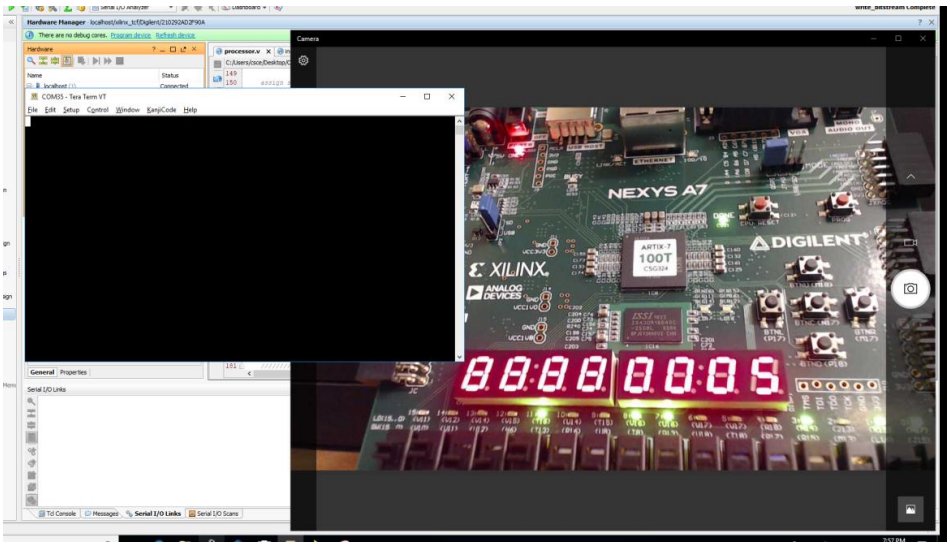


Here is some screenshots from the processor doing some instructions

LB instruction: `lb x3, 0(x0)`

Here the ssd-sel is set to 1011 which shows the value of the data read from the data memory which equals to 5 in this instant

Note that all these tests have been done before the single-single memory is implemented



There should have been a third test, for the immediate arithmetic instructions in specially however some of these instructions were tested here successfully also immediate arithmetic instructions were test during the previous milestone so since some of them are still working the rest is assumed to be working.