The American University in Cairo

Computer Architecture

Project 1 (MS2)

Single Cycle Implementation of RISC-V

Ahmed Osama Hassan

900171850

Mohamed Ashraf Hemdan

900171923

Wednesday, July 1, 2020

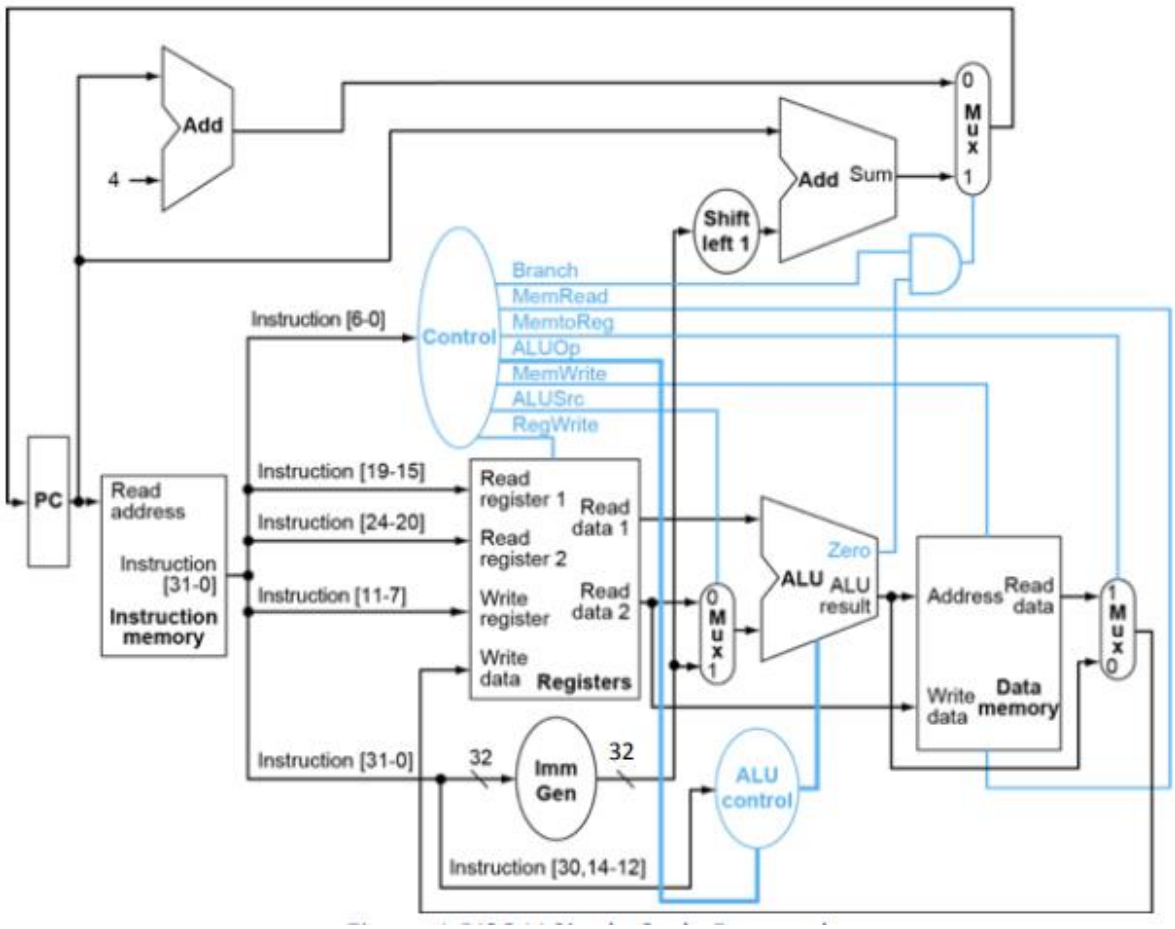
Submitted to:

Dr. Cherif Salama

We are trying to upgrade the simple single cycle implementation which supports only 7 instructions to support the whole RV32I Base Integer Instruction Set specified in the RISC-V specifications.

Previous design:

The design implemented before supports only 7 instructions which are (OR, AND, SW, LW, ADD, SUB, BEQ). Here is the path diagram for it.



Design Challenges and solutions

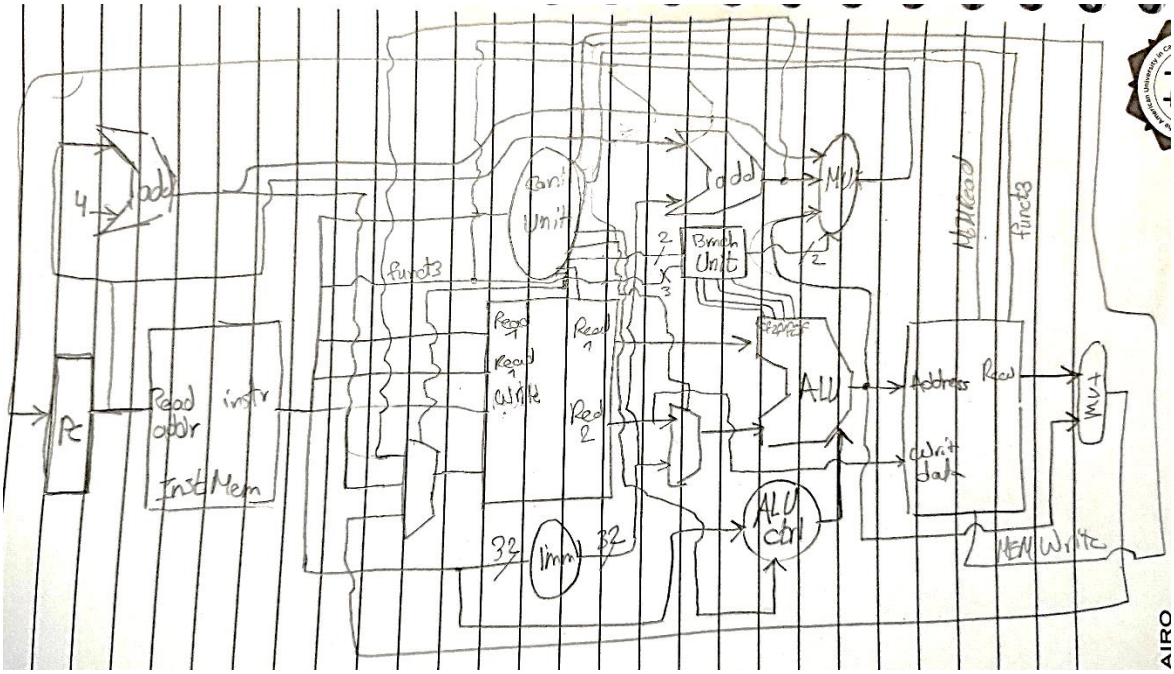
1- Data Memory only deals with words
   The data memory mentioned above only stores or read words. It cannot address bytes or half words. Also, the inner implementation depends on declaring an array of registers of size 32 bits. Therefore, it's required to change the inner implementation and add more control signals to it.
2- ALU limitations
   It's a very simple ALU that can only add, subtract, or, and and. It only supports the zero flag to indicate equality between input values. Therefore, we don't have any indication of how one of the input values is compared to the other like greater than or equal, less than, less than unsigned.

3- Immediate generator only supports limited instructions
   The immediate generator here only supports the above-mentioned instructions so it cannot deal with LUI, AUIPC, JAL … etc.
4- ALU control unit
   As there are new instructions with new ALU operations, there is also a need for the ALU control unit to support these new instructions
5- Different sources of writing to the reg file like (AUIPC instruction, and JALR instruction which requires writing to the reg file)
6- Branching limitations
   The old version controls branching be a single wire which is an ANDing between branch control signal and the zero flag from the ALU. Hence, we cannot differentiate between different branch instructions. For example, we don't know whether to branch or not based on the zero flag. We need to look at the instruction and see how the branch signal is related to the ALU flags to decide the output branch signal. Also, as there is new branching possibilities like the JALR which is taken from the output of the ALU, we need to have a 4*1 mux.

Design Changes:

1- Addition of a new unit for branching.
   This unit is responsible for choosing the pc_next from an array of 3 addresses: PC+4, PC_branch, ALU output (for JALR instruction). The unit receives a 2-bit signal from the control unit indicating the type of the new pc whether it's the ALU output, the normal pc added by or the PC immediate. If the unit receives a signal indicating that you should choose the PC+4 or the ALU output, it just sends the corresponding signal to the mux, responsible for choosing the pc-next, to choose it. However, if the signal indicates it's a branch instruction, it uses the ALU flags as input signals to decide the corresponding logical operation to be done to choose whether to branch or not. This unit was design as a separate module.
2- New Multiplexer before the Reg file Write Data port.
   This mux is responsible for choosing between the output of the pc-branch adder (for the AUIPC instruction), the normal write value got from the write back mux or the ALU output (for the JAL instruction).
3- ALU, ALU control unit, immediate generator Upgrade
   These modules have been edited to support the new instructions. The ALU for example supports SLT, SLTI, SLTIU
4- Control Unit
   The control unit has been changed from inside to support the new branch selection signal, write data mux selection line.
5- Data Memory Update
   The new data memory supports addressing by byte, half byte and word. It declares an array of 255 bytes. It receives the funct3 bits from the instruction and decide what to copy.

New Data Path:



Testing Procedures:

We used several test programs to test our processor.

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

lw x1, 0(x0)	0x00002083
lw x2, 4(x0)	0x00402103
lw x3, 8(x0)	0x00802183
L: add x1, x1, x2	0x002080b3
beq x2, x3, L	0xfe310ee3

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

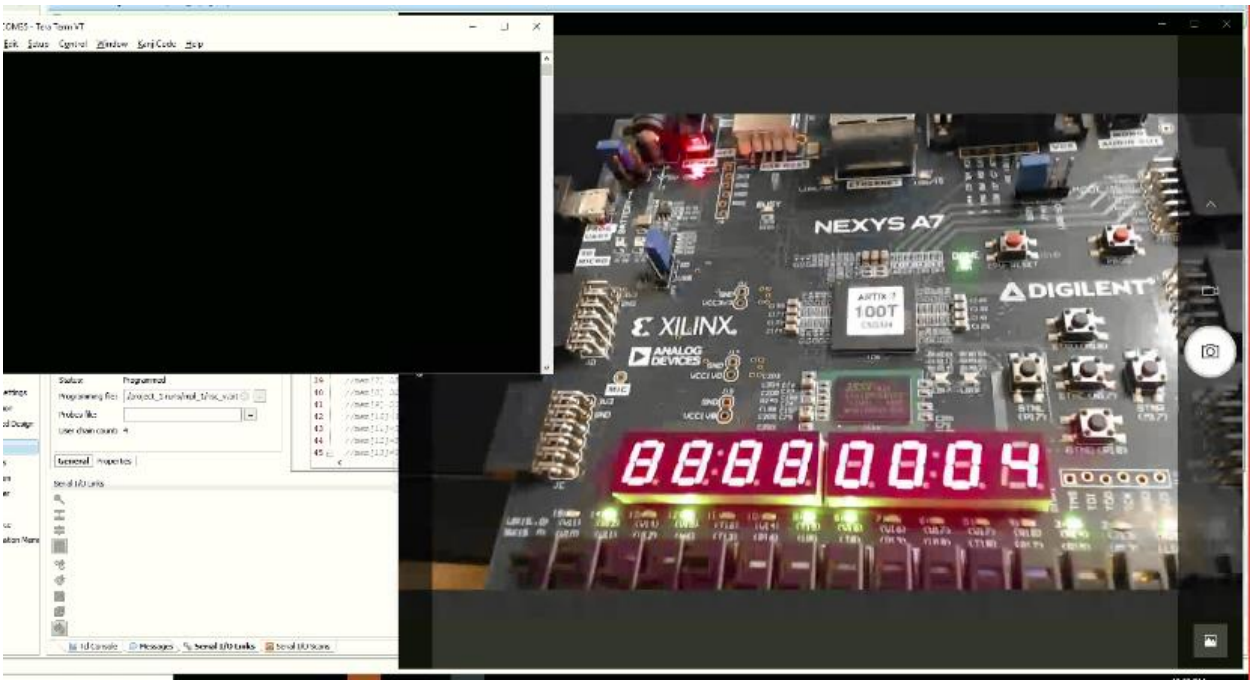
Here is some screenshots from the processor doing some instructinos

Lw instruction: lw x1, 0(x0)

This instruction is from experiment 4 (especially cycle 1)

All relevant information about selection lines are included in the excel sheet

This output on the seven segment display is the value getting out of the memory which indicates that the instruction loads the 4 successfully from memory

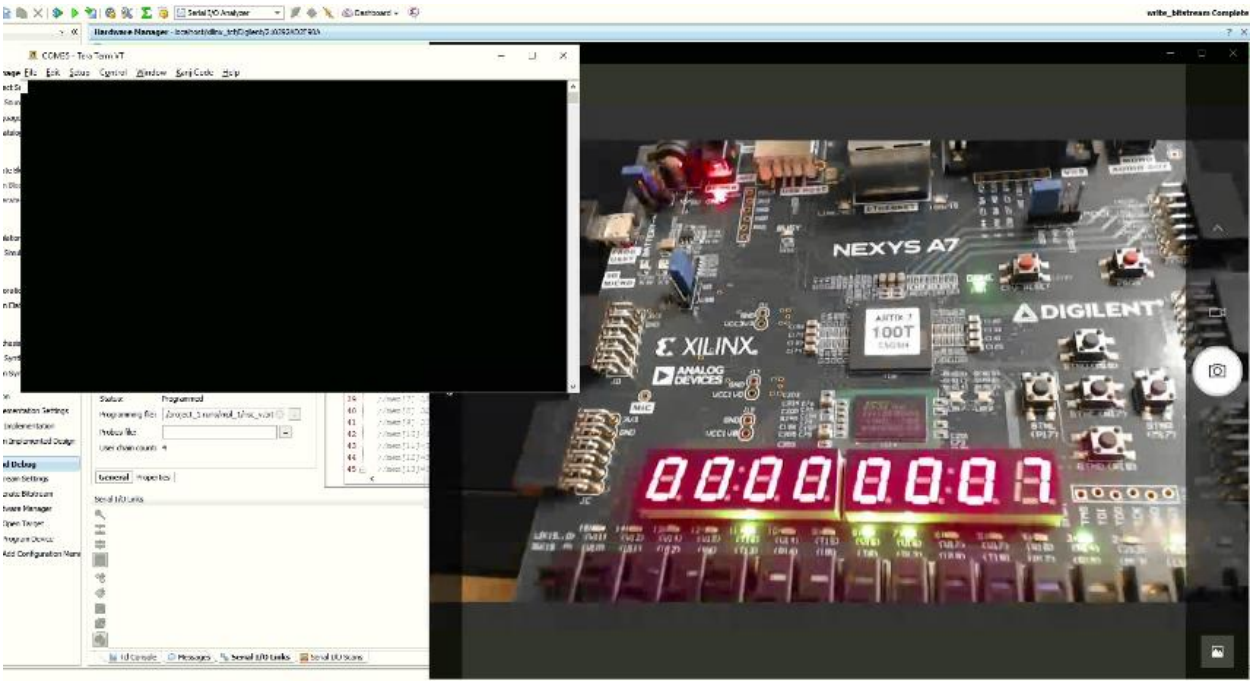


Add instruction: L: add x1, x1, x2

This instruction is from experiment 4 (especially cycle 4)

All relevant information about selection lines are included in the excel sheet

Here, the program should add the 4 getting out from memory and added to 3 which gives 7 as shown in the figure



Data collected during the testing of this program is the same as the one tested for the previous version.

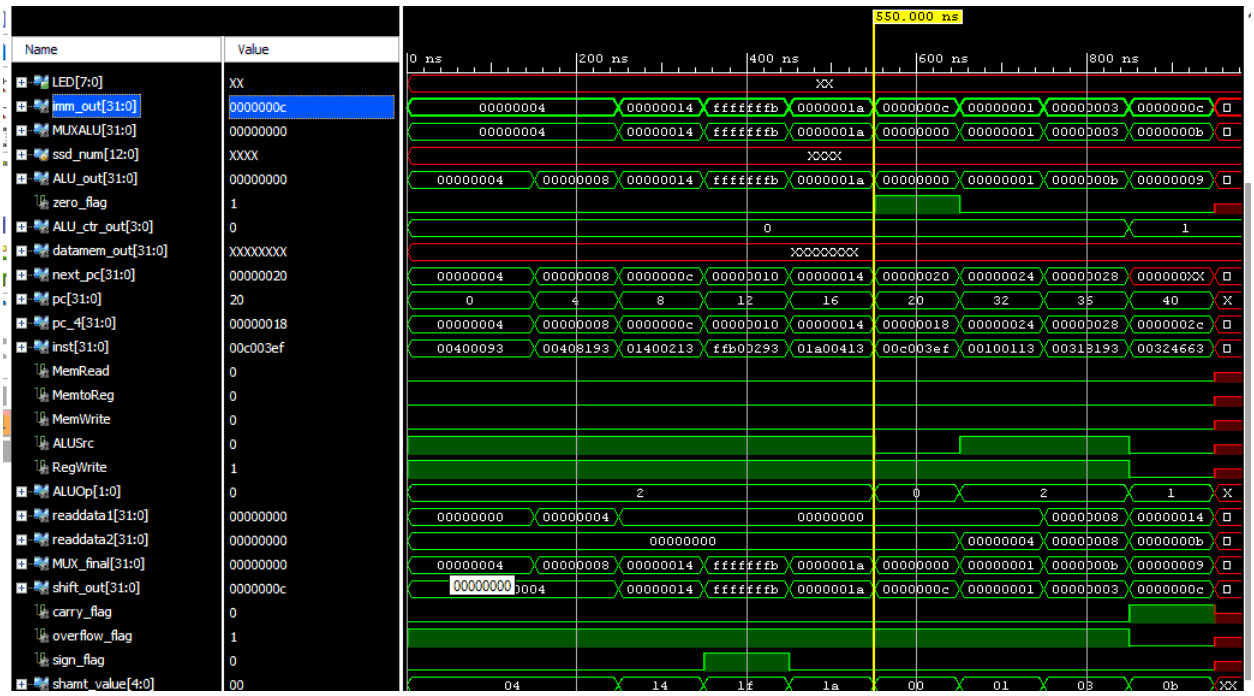
Attached with this report a data sheet indicating the values displayed on the SSD and LEDs. They are attached for reference

Second test:

We tested the new instructions like JAL, immediate instructions along with other branch instructions

addi x1, x0, 4
 	
addi x3, x1, 4
 	
addi x4, x0, 20
 	
addi x5, x0, -5
 	
addi x8, x0, 26
 	
jal x7, function #checking jal
 	
addi x1, x1, 1
 	
addi x1, x1, 2 # checking offset of jalr
 	
function:
 	
addi x2, x0, 1
 	
bigger_than:
addi x3, x3, 3
 	
blt x4, x3, less_than #checking blt works correctly
 	
bge x5, x3, bigger_than # checking BGE vs BGEU
 	
bgeu x5, x3, bigger_than
 	
less_than:
beq x3, x8, end #check beq
 	
bne x1, x2, function # checking bne and backward jumps
 	
end:
 	
jalr x0, x7, 4 #cheking jalr
	00000000010000000000000010010011
	00000000010000001000000110010011
	00000001010000000000001000010011
	11111111101100000000001010010011
	00000001101000000000010000010011
	00000000110000000000001111101111
	00000000000100001000000010010011
	00000000001000010000000010010011
	00000000000100000000000100010011
	00000000001100011000000110010011
	00000000001100100100011001100011
	11111110001100101101110011100011
	11111110001100101111101011100011
	00000000100000011000010001100011
	11111110001000001001010011100011
	00000000010000111000000001100111

Screenshots



The program here is executing addi instruction. It adds the value in imm_out to the value in readdata1 and output this value in ALU_out which is done successfully

Here is a JAL instruction got executed. The program jumps from pc 20 to 32.

Note:

There was another test but due to shortage in time we didn't do it. It will be done on the next mile stone for sure.

add x1, x0, x0 #unneccessary for immediates
 	
addi x1, x0, 13 #checks if the controls of the immediate arithmatic are working fine
 	
ori x1, x1, 5 #check different instruction of arithmatic
 	
xori x2, x1, 5 # check different arithmatic instructions
 	
addi x2, x2, -9 #checks the possibility of adding a negative immediate and still getting a right answer
 	
slti x3, x2, 6 # checks whether or not the slti is working
 	
slti x4, x2, 9
 	
sltiu x6, x2, 6 # check whether the signed vs unsigned cases are working
 	
slli x7, x2, 2 # checking the shift instructions
 	
srli x8, x2, 2 # comparing the logical vs arithmatic instructions
 	
srai x9, x2, 2
	00000000000000000000000010110011
	00000000110100000000000010010011
	00000000010100001110000010010011
	00000000010100001100000100010011
	11111111011100010000000100010011
	00000000011000010010000110010011
	00000000100100010010001000010011
	00000000011000010011001100010011
	00000000000100001000100111010011
	00000000000100001010101000010011
	01000000000100001010101010010011