Computer Structure Lab 2 Report

Computer Science and Engineering

Computer Structure

Group 89

Eduardo Alarcón Navarro NIA 100472175

100472175@alumnos.uc3m.es

Pol Gómez Morales NIA 100462234

100462234@ alumnos.uc3m.es

1.Introduction:

This lab practice consists in creating a set of instructions that replaces the original set of instructions using µprogramming (micro programming), reducing to ten the available instructions (lui, sw, lw, add, mul_add, beq jal, jr_ra, halt, xchb). This practice/lab has been conducted with the help of the online tool provided by the university known as wepsim, which can be found on http://wepsim.github.io. The architecture used has been RISK-V with a 32 bit processor.

2. Exercise 1:

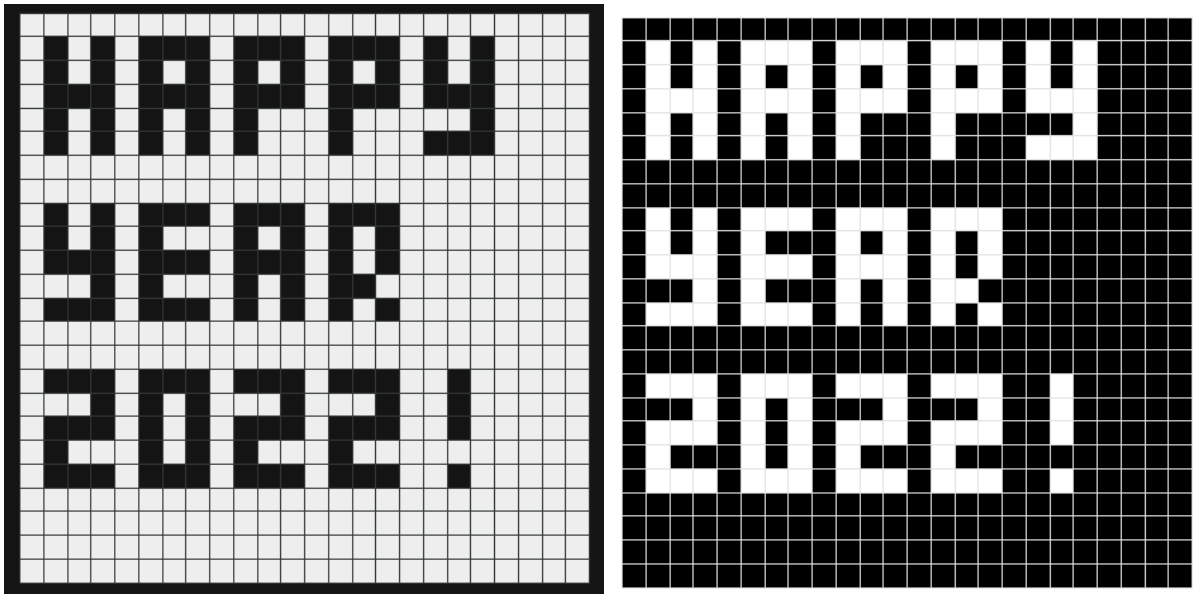
This exercise consisted of creating the 10 instructions using micro programming (µprogramming). The following table contains the necessary information requested in the statement:

Name of instruction	Design of instruction	Control signals activated	Design Decisions
lui (load unsigned integer)	1st cycle: MAR <- PC 2nd cycle: PC <- PC+4, MBR<- MM[MAR] 3rd cycle: reg <- MBR, fetch	(T2, C0), (M2, C2, TA, R, BW=11, M1, C1), (T1, SELC=10101, LC, A0, B, C=0)	We have decided to do a fetch in the instruction as the instruction is encoded in 64 bits, to allow to store words of length 32 bits
sw (store word)	1st cycle: MBR <- reg2 2nd cycle: MAR <- reg1 3rd cycle: MM[MAR] <- MBR & fetch	(MR=0, SELB=10000, T10=1, C1=1) (MR=0, SELA=10101, T9=1, C0=1) (Ta=1, BW=11, SE=1, TD=1, W=1, A0=1, B=1, C=0)	To store a word, we save the data we want to store into the MBR, which is the register that is dedicated to storing elements in the address of the MAR. Also, as we are storing a word, we selected BW=11, but if we wanted to store a bit, it would be BW=0, 2 bits BW=1 and half a word BW=10
lw (load word)	1st cycle: MAR <- reg2 2nd cycle: MBR <- MM[MAR] 3rd cycle: reg1 <- MBR & fetch	(MR=0, SELA=10000, T9=1, C0=1), (Ta=1, R=1, BW=11, SE=1, M1=1, C1=1), (T1=1, MR=0, SELC=10101, LC=1, A0=1, B=1, C=0)	This is the same as the previous one, but changing the order of the operations
add (add two registers)	1st cycle: reg3(SELA) + reg2(SLEB) _via ALU_ => RT3 -> reg1 & fetch	(MC=1, MR=0, SELA=01011, SELB=10000, MA=0, MB=0, SELCOP=1010, T6=1, SELC=10101, LC=1, SELP=11, M7, C7, A0=1, B=1, C=0),	Everything can be done in the same clock cycle, that is why it is really fast.
mul_add (multiply then add)	1st cycle: multiply rt1<- (reg2*reg3) 2nd cycle: add reg1 <- (rt1+reg4) & fetch	(MC=1, MR=0, SELA=01011, SELB=10000, MA=0, MB=0, SELCOP=1100, T6=1, C4), (MC=1, MR=0, SELB=00110, MA=1, MB=0, SELCOP=1010, T6=1, SELC=10101, LC=1, SELP=11, M7, C7, A0=1, B=1, C=0)	We only need two clock cycles, and the simple mathematical operations can be done in a single cycle. Also, as a remark: # SELP=11, M7, C7, update the Status register when using the ALU (Arithmetic Logic Unit)

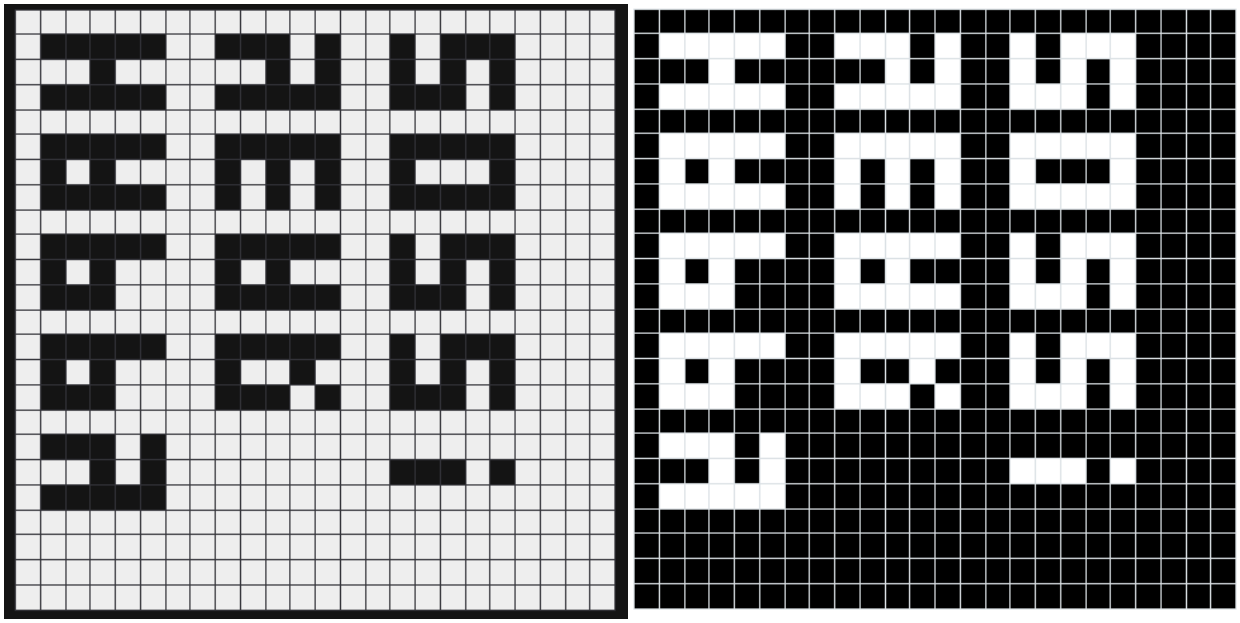
Name of instruction	Design of instruction	Control signals activated	Design Decisions
beq (branch if equal)	1st cycle: Save the content of the status register 2nd cycle: Subtract the two values, SLECOP=1011(-) and with SELP=11 we get the auxiliar values from the ALU 3rd cycle: Compare if Z=0, if so, go to the next μinstruction, else, go to bck2ftch (back to fetch) 4th cycle: Restore the Status Register (SR) 5th cycle: Save in RT1 <- PC 6th cycle: From the instruction get the offset and store it in RT2 7th cycle: PC <- RT1(pc) + RT2(value) & fetch 8th cycle (if Z≠0): Restore SR 9th cycle: Fetch	(T8, C5), (SELA=10101, SELB=10000, MC=1, SELCOP=1011, SELP=11, M7, C7), (A0=0, B=1, C=110, MADDR=bck2ftch), (T5, M7=0, C7), (T2, C4), (SE=1, OFFSET=0, SIZE=1010, T3, C5), (MA=1, MB=1, MC=1, SELCOP=1010, T6, C2, A0=1, B=1, C=0), bck2ftch: (T5, M7=0, C7), (A0=1, B=1, C=0)	
jal (jump at line)	1st cycle: ra <- PC 2nd cycle: PC <- offset & fetch	(T2, MR=1, SelC=00001, LC), (SE=1, OFFSET=0, SIZE=10100, T3, M2=0, C2=1, A0=1, B=1, C=0)	We have to store it in the register at the address of the PC at the moment, so the value that was stored there is erased from existence. Then, it adds the offset
jr_ra (jump register)	1st cycle: PC ← BR[ra] & fetch	(MC=1, MR=1, SELA=00001, T9=1, M2=0, C2=1, A0=1, B=1, C=0)	This instruction is really ad-hoc, as it always stores the content of pc into ra
halt	1st cycle: register0 is a hard wired 0, so we can store that 0 in PC and SR at the same time & fetch	(MC=1, SELA=00000, T9, M2=0, C2=1, M7=0, C7=1, A0=1, B=1, C=0)	As the register 0 is a hard wired 0, we don't need to compute much, just store that 0 in the PC and the SR, it would be like a hard stop
xchb (exchange bit)	1st cycle: MAR <- reg1 2nd cycle: MBR <- MM[MAR] 3rd cycle: RT1 <- MBR ###### 4th cycle: MAR <- reg2 5th cycle: MBR <- MM[MAR] 6th cycle: MAR <- reg1 7th cycle: MM[MAR] <- MBR ####### 8th cycle: MAR <- reg2 9th cycle: MBR <- RT1 10th cycle: MM[MAR] <- MBR & fetch	(MR=0, SELA=10101, T9=1, C0=1) (Ta=1, BW=0, SE=1, R=1, M1=1, C1=1) (T1=1, C4=1) (MR=0, SELB=10000, T10=1, C0= (Ta=1, BW=0, SE=1, R=1, M1=1, C1=1) (MR=0, SELA=10101, T9=1, C0=1) (TA=1, BW=0, SE=1, W=1, Td=1) (MR=0, SELB=10000, T10=1, C0=1) (T4=1, M1=0, C1=1) (Ta=1, BW=0, SE=1, TD=1, W=1, A0=1, B=1, C=0)	As you can see from the second column, we have 3 blocks. These blocks correspond to the **three main operations** our instructions have to compute: RT1 <- MM[reg1] & MM[reg1] <- MM[reg2] & MM[reg2] <- RT1. We get the information of the first position of the register1, which is an address, and we store it in a temporal register (RT1), then, on the address of the first register we store the value of the memory of the second register, previously obtained. Lastly, we store the value stored in the temporal register into the memory which address is stored on the second register.

3. Exercise 2:

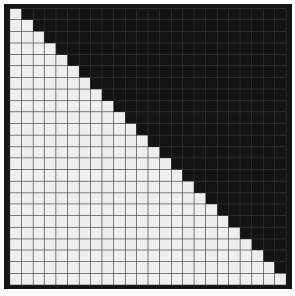
The second exercise consisted of applying the instructions developed on the previous exercise to test their functionality on a rather complex program, which had as a goal the transpose of a matrix of 24 bits x 24 bits. The original matrix, represented in the LED-matrix viewer from the wepsim was:



And, after the transformation, the matrix ended looking like this:



To test the functionality of the matrix, we did not use this matrix to test the instructions, but rather a simpler matrix, a lower diagonal matrix, like this one, to exactly know the result of our instructions.



The main instruction that attracts the most attention is the last one, the xchb (reg1), (reg2), which exchanges the bit stored on the memory address of the first register with the memory on the position which address is stored in the second register, storing temporally the first value on the internal temporal register.

We also have some different instructions from the original set that RISK-V comes with by default. For example, “mul_add”, which is the combination of “mul” and “add”. Also, the halt instruction is very interesting. There are also some other changes, for example, the creation of an ad-hoc “jr_ra”, instead of having an instruction that receives a parameter, being the target register, it will always overwrite the register “ra” (also known as x1 or 1 in the wepsim simulator). This led to us having to store in another register, t0, the value of the original ra, having an extra instruction, rather than storing the value of the program counter in a specific register.

Lastly, we have the “lui” (load unsigned integer), which is encoded in 64 bits instead of 32 bits.

The advantage of having more instructions is that the code in assembler is easier to do, generating less spaghetti code, as the instructions can be specific and efficient if you know what you are doing. The best example of this would be the xchb instruction.

The disadvantages of using custom instructions and this specific set of instructions are that programmers who are accustomed to programming with the default set of instructions won’t be productive in this environment. Also, these instructions are not available world-wide, which reduces the attractiveness of creating a program that is based on them. On top of this, creating efficient instructions is a really hard work, and generally, a user creating their own instructions will result in slower procedures, as they don’t usually have the knowledge of optimizing μprogramming.

The possible improvements we see on the instruction set we created for this exercise are, in general, simpler instructions that allow the user to jump from one place to another, as with this one, we had to jump using a beq (branch if equal) if we wanted to use a relative offset, which results in useless cycles. Also, we are missing elementary operations such as substructions, which forces us using a negative number stored in a register, even divisions can’t be performed.

I would add the basic arithmetic operations, such as subtraction and multiplication, as well as the ability of jumping to tags, instead of having to look for the instruction position.

4. Conclusions:

We found some problems when creating the instructions on wepsim, specially before we understood how the processor worked. We would have liked a simplified explanation of how the processor works with the cycles, ALL of them, not just saying, well we should add the fetch.

All in all, we think we dedicated around 25 hours to this delivery