ASSIGNMENT 2

Degree:
Applied Mathematics &
Computer Science

Subject:
Computer Structure

Sheila Gallardo Redondo
NIA: 100472913
Mail: 100472913@alumnos.uc3m.es
Group: 121

Alexandra Perruchot-Triboulet Rodríguez
NIA: 100453024
Mail: 100453024@alumnos.uc3m.es
Group: 121

Contents

EXERCISE 1

Design and Implementation of Microcode

Instruction	Elementary Operations	Control Signals	Design Decisions
lui reg1, U32	BR[reg1] ← U32 PC ← PC + 4	T2, C0, Ta, R, BW=11, M1, C1, T1, SelC=10101, LC, M2, C2, A0, B	Go to the MAR twice to update it properly and add another 4 to the PC because of the length of 2 words of the instruction.
sw reg1, (reg2)	MAR ← BR[reg2] MBR ← BR[reg1] Memory[MAR] ← MBR	SelA=10000, T9, C0, SelB=10101, T10, C1, Ta, Td, W, BW=11, A0, B	One cycle to take the address to the MAR, other to take the value to the MBR and other to save the information in the memory.
lw reg1, (reg2)	MAR ← BR[reg2] MBR ← Memory[MAR] BR[reg1] ← MBR	SelA=10000, T9, C0, T2, Ta, R, BW=11, M1, C1, SelC=10101, LC, T1, A0, B	We separate the cycles we the instruction reach the internal bus and the buses connected to the memory, otherwise the instruction does not work.
add reg1, reg2, reg3	RA ← reg2 RB ← reg3 RC ← reg1 RC ← RA + RB	SelCop=1010, MC, SelP=11, M7, C7, T6, SelA=01011, SelB=10000, SelC=10101, LC, A0, B	We use only one cycle because the register file has direct access to the ALU.
mul_add reg1, reg2, reg3, reg4	RA ← reg2 RB ← reg3 RT2 ← RA * RB RA ← reg4 RC ← reg1 RC ← RA + RT2	SelCop=1100, MC, SelA=10000, SelB=01011, T6, C5, SelCop=1010, SelA=00110, MB, SelC=10101, LC, SelP=11, M7, C7, A0, B	We have used the temporal register 2 to save the value of the multiplication and operate with it in the ALU again.

beq reg1, reg2, S10	RA ← reg1 RB ← reg2 SR ← RA - RB If Z==0 → go to fetch Else: BR[RT1] ← PC BR[RT2] ← S10 PC ← RT1 + RT2	SelCop=1011, MC, SelA=10101, SelB=10000, SelP=11, M7, C7, MADDR=backtofetch, C=0110, B, T2, C4, SIZE=01010, SE, T3, C5, SelCop=1010, MA, MB, T6, C2, A0	We have used a branch to a micro address inside of the instruction to make the conditional work. Moreover, to check if the 2 values are equals we subtract them and use the value of Z
jal U16	RC = ra RC ← PC PC ← U16	SelC=00001, MR, T2, LC, SIZE=10000, T3, C2, A0, B	We have selected the ra register in RC every time that the instruction is executed.
jr_ra	RA = ra PC ← RA	SelA=00001, MR, T9, C2, A0, B	We have selected the ra register in RA every time that the instruction is executed.
halt	RA = zero PC ← RA SR ← RA	SelA=00000, MR, T9, C7, C2, A0, B	To implement the instruction, since we need the zero value, we have saved the zero register in RA.
xchb (reg1), (reg2)	MAR ← BR[reg1] MBR ← Memory [MAR] RT1 ← MBR MAR ← BR[reg2] MBR ← Memory [MAR] MAR ← BR[reg1] Memory[MAR] ← MBR MBR ← BR[RT1] MAR ← BR[reg2] Memory [MAR] ← MBR	SelA=10101, T9, C0, Ta, R, BW=00, C1, M1, T1, C4, T10, SelB=10000, W, Td, T4, A0, B	We have saved the information of the memory address of reg1 in a temporal register so that we do not lose the information when we save first the information of the second memory address in the first one.

EXERCISE 2

Differences in the types of instructions

We have used the same set of instructions as in exercise 1 because they worked as we wanted to implement the assembly code. The only additional instructions we needed to develop exercise 2 were the ones we were provided (“*in*” and “*out*” instructions), that work with the input/output devices.

Advantages and Disadvantages

The first advantage of using the extended instruction set, is that we can use the input and output modules to be able to work with data such as images (like we have done in exercise 2). Apart from this difference, as we have said before the original instruction set and the extended one, are pretty much the same with the only difference of the two new instructions, therefore, the advantages and disadvantages are the same.

One of the main disadvantages of the original set of instructions, is that we cannot store negative numbers in any register because the *lui instruction* only considers unsigned numbers, and by default this ones are positive. One main consequence of this, is that we cannot perform any substraction operation since the only instructions to operate are *add* and *mul_add*.

Furthermore, we found problems in order to add a constant number to a register, since the *add* instruction requires that the numbers are in registers. To solve that, we stored the value we wanted to add in a register using *lui* and then performed the *add* operation with the two registers.

Finally, we also found problems in order to store the address when we jumped to a label in the ra register (to preserve the value of PC) since the address is stored in a register and the *lui* instruction only allows to save integers and no registers.

Possible Improvements

To improve the instruction set of the program we have some ideas. Firstly, we would include an instruction as *addi* that allow us to add a register's content with a constant included in the intruction. If the constant is a negative value, then we would be able to substract and have negative values too.

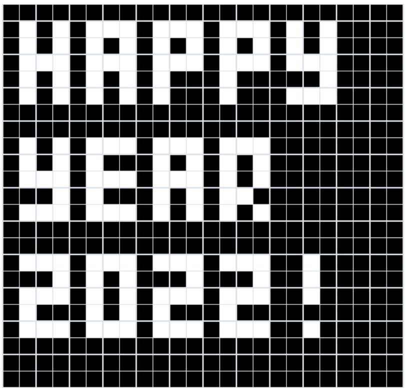
Moreover, we could incorporate a *li* instruction to be able to store positive and negative values. This would solve the problem of substraction and get negative values, but it does not solve the adding a constant problem.

To finish, we would add a *mv* instruction to move the values between registers and solve the problem of preserving the PC when we jump to a label.

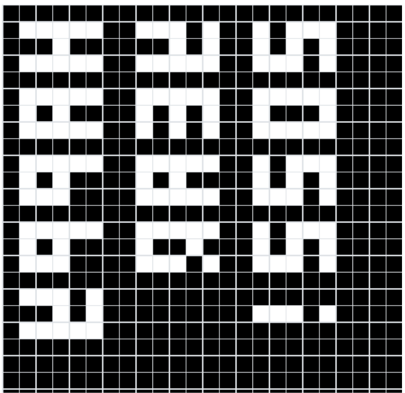
Result

The program takes an image as a matrix (where each byte of the image is in a different position of the matrix) and traspose this matrix. The result of this is a rotation of the image as we can see in the pictures:

Original:



Result Image:



Conclusions and Errors encountered

During this exercise we experienced several complications. One of them was that every time we executed an instruction, when it reached the MAR, this had a memory address that didn´t correspond to the actual instruction but the previous one. To solve that problem, we decided to add a second loop through the memory so that the MAR could update.

Moreover, another problem was that in the *lui instruction*, as it occupied two words, in one of them we had the immediate value U32, but we didn´t know if we had the address of that number or the number itself. We then found out that the second word contained the 32-bit number itself and not the address.

Furthermore, in the *beq instruction* we used a branch to jump to a micro address. To do that, we used the conditional jump for Z==0, that is, if the numbers weren´t equal, we jumped to this micro address. The problem was with the meaning of Z==0, we thought it meant that the two numbers were equal, and so it didn´t jump to the micro direction, but the truth is that it means the opposite.

We counted the number of times we use the bus per micro instruction, so that we used it once per cycle, trying to minimize the number of cycles as much as possible.

For the second exercise, the biggest problem we encountered was how to convert the indexes of the matrix into addresses for each byte. To solve that, we used our *mul_add* instruction to calculate each position of each byte and then add it to the initial address of the image to obtain each address corresponding to each byte. Since the matrix is 24*24 we have realised that with the mul_add instruction we consider reg3 and reg4 are de indexes of the matrix (i,j) and reg2 is the constant 24: **position = 24*i + j .** This allow us to interchange the indexes to transpose the matrix.

To conclude this report, the number of hours allocated to this practice were around 20 hours.