

Degree in Applied Mathematics and Computing

Computer Structure

2022-2023

Assignment 2

Introduction to Microprogramming

Ilan Halioua Molinero | 100472908
100472908@alumnos.uc3m.es

Alejandro Fernández - Paniagua | 100473072
100473072@alumnos.uc3m.es

Group 121

TABLE OF CONTENTS

1. EXERCISE 1	3
1.1. Design of the microcode for the extended instruction set	3
2. EXERCISE 2	6
2.1. Comparison of original and extended instruction sets	6
3. CONCLUSION.....	9
3.1. Problems encountered.....	9
3.2. Conclusions	9

1. EXERCISE 1

1.1. Design of the microcode for the extended instruction set.

Name of instruction	Elementary operations	Control signals	Design decisions
lui R _{RE1} , U32	MAR \leftarrow PC	T2, C0	Join 3 elementary operations in the same clock cycle as they don't interfere on internal bus. Increment PC to have address of next instruction and not of second word.
	MBR \leftarrow Memory [MAR]	Ta, R, BW=11, M1=1, C1	
	BR[R _{RE1}] \leftarrow MBR PC \leftarrow PC + 4 Jump to fetch	SelC=10101, T1, LC, M2=1, C2, A0, B, C=0	
sw R _{RE1} , (R _{RE2})	MAR \leftarrow BR[R _{RE2}]	SelA=10000, MR=0, T9, C0	Join 2 elementary operations in the same clock cycle.
	MBR \leftarrow BR[R _{RE1}]	SelB=10101, MR=0, T10, M1=0, C1	
	Memory [MAR] \leftarrow MBR Jump to fetch	Ta, Td, W, BW=11, A0, B, C=0	
lw R _{RE1} , (R _{RE2})	MAR \leftarrow BR[R _{RE2}]	SelA=10000, MR = 0, T9, C0	Join 2 elementary operations in the same clock cycle.
	MBR \leftarrow Memory [MAR]	Ta, R, BW=11, M1, C1	
	BR[R _{RE1}] \leftarrow MBR Jump to fetch	T1, SelC=10101, LC=1, MR=0, A0, B, C=0	
add R _{RE1} , R _{RE2} , R _{RE3}	BR[R _{RE1}] \leftarrow BR[R _{RE2}] + BR[R _{RE3}], SR	MR=0, SelA=10000, SelB=01011, MC=1, MA=0, MB=00, SelCop=1010, T6, SelC=10101, LC=1, SelP=11, M7=1, C7	Join 2 elementary operations in the same clock cycle.
	Jump to fetch	A0, B, C=0	
mul_add R _{RE1} , R _{RE2} , R _{RE3} , R _{RE4}	RT1 \leftarrow BR[R _{RE2}] * BR[R _{RE3}]	MR=0, SelA=10000, SelB=01011, MC=1, MA=0, MB=00, SelCop=1100, T6, C4	Join 2 elementary operations in the same clock cycle. Use RT1(temporal register) to use the result of the multiplication as an operand to the sum.
	BR[R _{RE1}] \leftarrow RT1 + BR[R _{RE4}], SR	MR=0, SelB=00110, MA=1, MB=00, MC=1, SelCop=1010, T6, SelC=10101, LC=1, SelP=11, M7=1, C7	

	Jump to fetch	A0, B, C=0	
beq R _{RE1} , R _{RE2} , S10	RT2 ← SR	T8, C5	
	SR ← BR[R _{RE1}] - BR[R _{RE2}]	SelA=10101, SelB=10000, MC=1, SelCop=1011, SelP=11, M7, C7	
	Conditional Jump to MADDR	A0=0, B=1, C=110, MADDR=bck2fch	If Z=0 , go to 'bck2fetch'
	SR ← RT2	T5, M7=0, C7	Join 2 elementary operations in the same clock cycle.
	RT1 ← PC	T2, C4	
	RT2 ← IR(S10)	SE=1, OFFSET=0, SIZE=01010, T3, C5	
	PC ← RT1+ RT2 Jump to fetch	MA=1, MB=1, MC=1, SelCop=1010, T6, M2=0, C2, A0=1, B=1, C=0	
	bck2fetch	T5, M7=0, C7	
		A0=1, B=1, C=0	
jal U16	BR[R1] ← PC	T2, MR=1, SelC=00001, LC	Join 2 elementary operations in the same clock cycle.
	PC ← IR(U16) Jump to fetch	Size=10000, Offset=00000, T3, M2=0, C2, A0, B, C=0	
jr_ra	PC ← BR[R1] Jump to fetch	MR=1, SelA = 00001, T9, M2=0, C2, A0, B, C=0	Unique cycle for both processes.

halt	$PC \leftarrow BR[R_0]$	SelA=00000, MR=1, T9, M2=0, C2	Use register x0 whose value is zero to update PC and SR. No jump to fetch as we want to terminate program execution and no further instructions will be processed.
	$SR \leftarrow BR[R_0]$	SelA=00000, MR=1, T9, M7=0, C7	
xchb (R_{RE1}), (R_{RE2})	$MAR \leftarrow BR[R_{RE1}]$	SelA=10101, T9, C0	Join 2 elementary operations in the same clock cycle. BW = 0 to exchange bytes. (This allows also to exchange half and full words)
	$MBR \leftarrow \text{Memory}[MAR]$	Ta, R, BW=0, M1=1, C1	
	$RT1 \leftarrow MBR$	T1, C4	
	$MAR \leftarrow BR[R_{RE2}]$	SelB=10000, T10, C0	
	$MBR \leftarrow \text{Memory}[MAR]$	Ta, R, BW=0, M1=1, C1	
	$MAR \leftarrow BR[R_{RE1}]$	SelA=10101, T9, C0	
	$\text{Memory}[MAR] \leftarrow MBR$	Ta, Td, W, BW=0	
	$MBR \leftarrow RT1$	T4, M1=0, C1	
	$MAR \leftarrow BR[R_{RE2}]$	SELB=10000, T10, C0	
	$\text{Memory}[MAR] \leftarrow MBR$ Jump to fetch	Ta, W, Td, BW=0, A0, B, C=0	

2. EXERCISE 2

2.1. Comparison of original and extended instruction sets.

Original Instruction Set	Extended Instructions of Ex. 1	Differences in the types of instructions	Advantages & Disadvantages	Possible Improvements
li R _{RE1} , n	lui R _{RE1} , U32	The extended instruction loads immediate value of 32 bits (more than what can be loaded with li instruction. On the other hand, the extended one has the restriction that the immediate value must be unsigned not as the one in li.		Make it available to use for loading signed integer values into a register.
sw R _{RE1} , (R _{RE2})	sw R _{RE1} , (R _{RE2})	SAME		none
lw R _{RE1} , (R _{RE2})	lw R _{RE1} , (R _{RE2})	SAME		none
add R _{RE1} , R _{RE2} , R _{RE3}	add R _{RE1} , R _{RE2} , R _{RE3}	SAME		none
mul R _{RE1} , R _{RE2} , R _{RE3} add R _{RE1} , R _{RE1} , R _{RE4}	mul_add R _{RE1} , R _{RE2} , R _{RE3} , R _{RE4}	The extended instruction multiplies and adds the content of several registers in one word, not in 2 as the RISC V original one.	The extended instruction mul_add is more efficient than the pair of instructions in the original set (add and mul) since you can pretty much perform any arithmetic operation with it. Addition, Subtraction and multiplication.	none
beq R _{RE1} , R _{RE2} , etiq	beq R _{RE1} , R _{RE2} , S10	Same functionality but in this case, the original one is more useful since it allows the user to make backward jumps to tags (apart from forward ones)	Opposite to the original jal instruction in RiscV, the extended one is limited in use, based on its design nature. Since it adds a given 16b value, the programmer must have counted the amount of words that it has to be added to pc in order to point to the	Make it able to jump to the back as well

			desired address. In contrary, the original one is much more user friendly and versatile since it simply requires to specify the desired address in the form of a 16b tag that the pc will be assigned.	
jal rd, address	jal U16	Original RiscV instruction lets user choose the register in which the return address will be stored, not as the extended one as it automatically assigns it to ra register.		Let user choose register in which store return address to function call.
jr ra	jr_ra	SAME		none
li a7 10 ecall	halt	Sort of like an instantaneous system call instruction, the halt extended instruction performs better than the pair of instructions needed to exit the program execution through the original instruction set. It basically serves on the same way as the sequence of original instructions but faster and more efficiently, taking up only one word in memory to end up the life of any program.		Let the user know everything has been successfully terminated by returning a one in the screen.
lw t1(RRE1) lw t2 (RRE2) sw t1 (RRE2) sw t2 (RRE1)	xchb (R _{RE1}), (R _{RE2})	The extended instruction does a fantastic job at exchanging data elements inside two addresses in the Main Memory. It is way faster than the corresponding alternative provided by Risc v's original set not only by simplifying the amount of coding lines but also being optimized since it reduces the size in memory taken by its functionality purpose from 4 whole words down to a single one.		none

OUTPUT – Dev Matrix:

⏻
Reiniciar

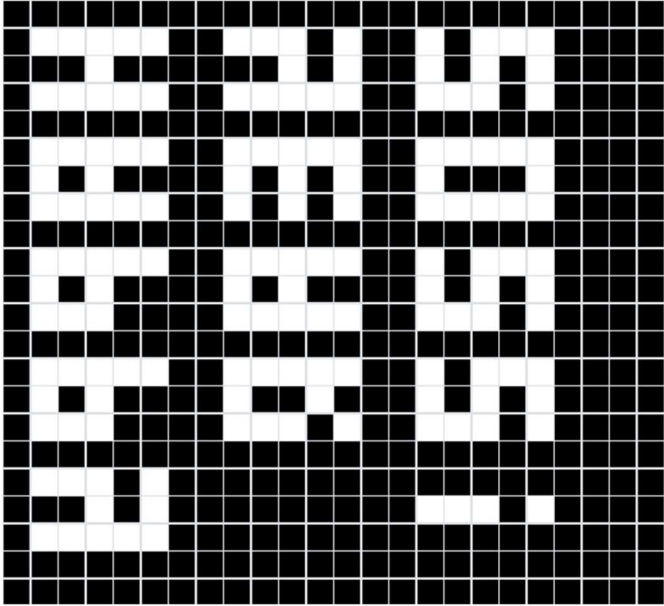
⏮
μInstrucción

⏭
Instrucción

▶
Ejecutar

📷 Estados

Dev Led-Matrix ▼



IR_DECO halt

IR 8C000000

PC 00000000

MAR 0000800C

MBR 8C000000

RT1 00008030

RT2 0000004C

RT3 00000000

SR 00000000

μADDR 00000088

3. CONCLUSION

5.1. Problems encountered.

During this assignment (microcode and assembly parts) several problems were encountered. Among them, a crucial one was that before starting this project, we had not realized the utility of signals *MR* and *MC*, two that became essential for the correct functioning of the program. For the assembly part, we realized that our *xchb* worked only to exchange words as we had in byte selector *BW=11*, so we had to change it to *BW=00*, realizing that this made not only the exchange of bytes work but also half and full words. Other problems during the assembly part were that as the *beq* instruction asked for an integer we did not realize at a first glance that we could use labels for the loops so at first we counted the exact bytes to add to pc to have the address of the instruction we wanted to go when the conditions were satisfied.

5.2. Conclusions.

Throughout the development of the course, we have encountered several theoretical doubts whose answer did not seem clear to us at first hand. However, after the thought process and hard work that the project has demanded us to deliver, we have totally wrapped our minds around it and feel confident about our perspective and knowledge of how a modern day processor works. Since the start of the lab assignment, we have been constantly cooperating with each other to get to the places we needed to step into in order to work out the challenges presented in the exercises successfully. In addition, this has indirectly enhanced our comfort level when interacting with the creator environment, which has overall played a huge role in the learning process of all the low-level methodologies and approaches taken in the design of the processor components in order to make up its complete and sophisticated structure. To sum up, even though the project has been a reasonable challenge, which has asked us for lots of time inversion, we have experienced a great level of satisfaction after learning how to get our hands on, with a computer in considerable detail depth.