Computer Structure
Lab 2 Report

Alex Zaharia & Skyler Edwards

100500012@alumnos.uc3m.es & 100500024@alumnos.uc3m.es

Group 87/89M

Universidad Carlos III de Madrid

Table of Contents

Exercise 1 Table

Instruction Name	Elementary Operations	Control Signals	Design Decisions

lui R_{RE1} U32	MAR ← PC
	
	MBR ← MP[MAR]
	
	R_{RE1} ← MBR
		T2, C0
		
		Ta, R, BW = 11, M1 = 1, C1, M2 = 1, C2
		
		T1, SELC = 10101, MR = 0 LC, A0 = 1, B = 1, C = 0
			Store address of U32 into MAR. Store data at address stored in MAR(U32) into MBR. This data is then moved into R_{RE1} FETCH

sw R_{RE2} (R_{RE1})	MAR ← RI(R_{RE1})
	
	MBR ← R_{RE2}
	
	MP[MAR] ← MBR
		SELA=10000, MR=0, T9, C0
		
		SELA=10101, MR=0, T9, M1=0, C1
		
		TA, W, BW=11, TD, A0=1, B=1, C=0
			Store address from R_{RE1} into MAR. Store data from R_{RE2} into MBR. Store Data from MBR into the memory location of the address stored in MAR. FETCH

lw R_{RE1} (R_{RE2})	MAR ← RI(R_{RE2})
	
	MBR ← MP[MAR]
	
	R_{RE1} ← MBR
		SELA=10000, MR=0, T9, C0
		
		TA, R, BW=11, M1=1, C1
		
		T1, SELC=10101, MR=0, LC , A0=1, B=1, C=0
			Store address from R_{RE2} into MAR. Store data at address in MAR into MBR. Store data in MBR into R_{RE1} FETCH

add R_{RE1}, R_{RE2}, R_{RE3}	R_{RE3} ← R_{RE1} + R_{RE2}	SELA=10000, SELB=01011, MR=0, MA=0, MB=00, SELCOP=1010, MC=1, T6, SELC=10101, LC, SELP=11, M7=1, C7, A0=1, B=1, C=0	Perform add operation in ALU using R_{RE2} and R_{RE3} Store result into R_{RE1} FETCH
mul_add R_{RE1}, R_{RE2}, R_{RE3}, R_{RE4}	R_{RE2} ← R_{RE2} * R_{RE3} R_{RE1} ← R_{RE2} + R_{RE4}	SELA=10000, SELB=01011, MR=0, MA=0, MB=00, SELCOP=1100, MC=1, T6,	Perform multiplication operation in ALU using R_{RE2} and R_{RE3}

		C5 SELA=00110, MR=0, MA=0, MB=01, SELCOP=1010, MC=1, T6, SELC=10101, LC, SELP=11, M7=1, C7, A0=1, B=1, C=0	Add result with R_{RE4} Store result into R_{RE1} FETCH

beq R_{RE1} R_{RE2} S10	$R_{T2} \leftarrow$ SR
	
	If $R_{RE1} == R_{RE2}$ or $R_{RE1} - R_{RE2} == 0$
	
	SR $\leftarrow R_{T2}$
	
	$R_{T1} \leftarrow$ PC
	
	$R_{RT2} \leftarrow$ IR(S10)
	
	PC $\leftarrow R_{T1} + R_{T2}$
		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=10000, 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
			Store SR into R_{T2}. IF $R_{RE1} == R_{RE2}$ or $R_{RE1} - R_{RE2} == 0$. Store R_{T2} into SR. Store PC into R_{T1}. S10 into R_{T2}. Perform ALU operation $R_{T1} + R_{T2}$ and store into PC

jal R_{RE1} U16	RC $\leftarrow$ PC PC $\leftarrow$ RI(U16)	T2, SELC=00001, MR=1, LC SIZE=10000, OFFSET=0, SE=0, T3, M2=0, C2, A0=1, B=1, C=0	Store PC value into RC (used to jump back to main program or PC once jr ra is called). Store U16 into PC, jumping to function.
jr R_{RE1}	PC $\leftarrow R_{RE1}$	SELA=00001, MR=1, M2=0, C2, T9, A0=1, B=1,	Store address in R_{RE1} into the PC

		C=0	(like returning to main)

halt	PC ← RA/zero
	
	SR ← RA/zero
		SELA=00000, MR=1, M2=0, C2, M7=0, C7, A0=1, B=1, C=0	Load zero into PC and zero into SR

xchb R_{RE1} R_{RE2}	R_{T1} ← Memory [R_{RE1}] Memory [R_{RE1}] ← Memory [R_{RE2}] Memory [R_{RE2}] ← R_{T1}	SELA=10101, MR=0, T9, C0 TA, R, BW=0, SE=0, M1=1, C1 T1, C4 SELA=10000, MR=0, T9, C0 TA, R, BW=0, SE=0, M1=1, C1 T1, C5 T4, M1=0, C1 TA, BW=0, SE=0, TD, W SELA=10101, MR=0, T9, C0 T5, M1=0, C1 TA, BW=0, SE=0, TD, W, A0=1, B=1, C=0	Store from memory of R_{RE1} into R_{T1}. Store from memory of R_{RE2} into the memory of R_{RE1}. Store from R_{T1} into the memory of R_{RE2}

Exercise 2 Description

We have found that the "j" or jump instruction is what distinguishes the two instruction sets the most. This can be seen in the instruction "beq a3 a3 while1." Though it isn't technically necessary to use the jump instruction, it is much more direct (less cycles) and easier to understand than having to use the "beq" instruction to do the same task. Another example of this is with the "add t6 a2 zero." This instruction has the same issue. It acts as a load instruction, loading a2 into t6. It just uses more clock cycles than needed. Though these aren't the best examples, we have learned that creating your own instruction set, as we did in WepSIM using microcode, has the advantage of being shorter and less verbose. However, in our case, it did the opposite: it was a disadvantage. This is assuming the instructions are specific towards the program being created. Ultimately, doing so allows us to minimize the amount of clock cycles, allowing the program to run as efficiently and effectively as possible. Besides this, the two different instruction sets do in the end achieve the same goal. Our instruction set can be further improved by looking over the control signals once more and optimizing the microcode to limit the amount of clock cycles being run. This may be an issue in our project. Another improvement that could be made in our project, but couldn't be made because we ran out of time, is in the assembly code with the output LED Matrix. Though our returning image is rotated 90%, we need to adjust the assembly code to reflect the image vertically. Input: image[i][j]. Our project's output: image[j][-i]. Improved project: image[j][i].

Conclusion

Although this lab only had 2 exercises, the content was much more challenging than the previous lab. It required a lot more effort in order to gain enough of an understanding to be able to complete the assignment. Overall, the lab took around 30-40 cumulative hours to complete. One area we had difficulties with was testing the microcode and making sure it all worked. Another issue we had was in figuring out what the instruction x jn was designed to do considering it isn’t an ordinary instruction instead one we are in a way supposed to design. However, we were eventually able to figure out how to resolve these issues after re-examining the slides and project description enough times. However, despite being time-consuming, the lab did aid in reinforcing the understanding of microprogramming. It was helpful to see how the concepts we learn in class/lab time can be applied and put to use to create something. We encountered an issue in the second part of the lab. We could not figure out how to simply rotate the image 90 degrees. Instead, the picture would rotate 90 degrees and also flip/mirror itself. However, it ultimately is flipped 90 degrees.