



Universidad Carlos III
Curso Estructura de computadores 2022-23
Práctica 2
Curso 2022-23

Ingeniería Informática, Segundo Curso

Paula Subías Serrano(NIA: 100472119, e-mail: 100472119@alumnos.uc3m.es)

César López Mantecón (NIA: 100472092, e-mail: 100472092@alumnos.uc3m.es)

Prof. Felix García Caballeira

Grupo: 81

Índice

1. Ejercicio 1:	3
2. Ejercicio 2:	6
2.1. Comparación de instrucciones	6
3. Horas invertidas	7
4. Conclusiones	7

1. Ejercicio 1:

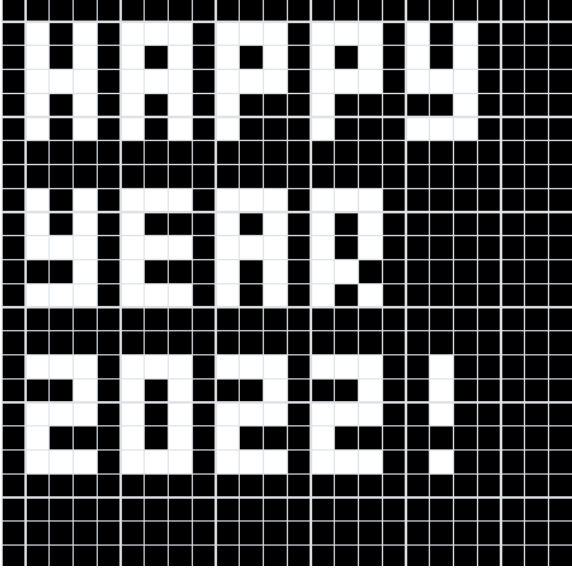
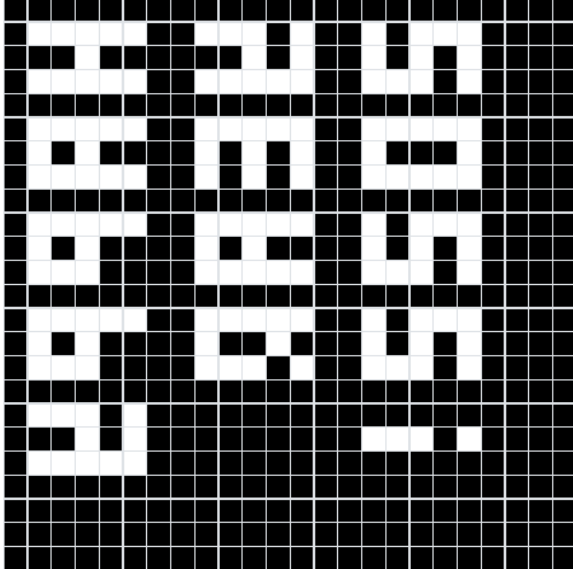
Nombre Instrucción	Diseño instrucción	Señales de control	Decisiones de diseño
lui R _{RE1} , U32	C1: MAR <- PC C2: MBR <- MP(MAR), PC <- PC + 4 C3: R <- MBR, salto a fetch	C1: T2, C0 C2: M1, C1, TA, BW=11, R, M2, C2 C3: T1, LC, SelC = 10101, MR=0, A0=1, B=1, C=0	<p>Se ha incluido el salto a fetch en el último ciclo para ahorrar un ciclo.</p> <p>PC se actualiza a PC + 4 para que apunte a la siguiente instrucción (ya que lui usa 2 palabras, con un único fetch apuntaría al dato). Esto se hace en el mismo ciclo en el que MBR almacena el dato ya que son circuitos independientes.</p>
sw R _{RE1} , (R _{RE2})	C1: MBR<-R1 C2: MAR<-R2 C3: MP <- MBR, salto a fetch	C1: SelA=10101, MR=0, T9, C1, M1=0 C2: T9, SelA=10000, MR=0, C0 C3: TD, W, TA, BW=11, A0=1, B=1, C=0	<p>Se ha incluido el salto a fetch en el último ciclo para ahorrar un ciclo</p>
lw R _{RE1} , (R _{RE2})	C1: MAR<-R2 C2: MBR <- MP(MAR) C3: R1 <- MBR, salto a fetch	C1: T9, SelA=10000, MR=0, C0 C2: M1, C1, TA, BW=11, R C3: T1, LC, SelC = 10101, MR=0, A0=1, B=1, C=0	<p>Se ha incluido el salto a fetch en el último ciclo para ahorrar un ciclo</p>
add R _{RE1} , R _{RE2} , R _{RE3}	C1: R1 <- R3 + R2, salto a fetch	C1: SelA=01011, SelB=10000, SelC=10101, MR=0, SelCop=1010, MC, T6, SelP=11, C7, M7, LC, A0, B, C=0	<p>Un solo ciclo al solo necesitar el bus de datos para cargar el resultado de la suma en R1</p> <p>Se ha incluido el salto a fetch en el último ciclo para ahorrar un ciclo</p>
mul_add R _{RE1} , R _{RE2} , R _{RE3} , R _{RE4}	C1: RT2 <- R2*R3 C2: R1 <- RT2 + R4, salto a fetch	C1: SelA=10000, SelB=01011, MR=0, SelCop=1100, MC, T6, C5 C2: SelA=00110, MB=01, SelCop=1010, MC, T6, LC, SelC=10101, M7, C7, SelP=11, A0, B, C=0	<p>Se ha incluido el salto a fetch en el último ciclo para ahorrar un ciclo.</p> <p>Se usa un registro temporal para almacenar el valor de la primera operación.</p>
beq R _{RE1} , R _{RE2} , S10	C1: RT2 <- SR, R1 - R2, se actualiza SR C2: SR <- RT2,	C1: T8, C5, SelA=10101, SelB=10000, MR=0, MA=0, MB=00, SelCop=1011, MC,	<p>SR es almacenado en un registro temporal para poder recuperar su</p>

	<p>if R1 - R2 != 0 (Z = 0) => salto a fetch C3: RT1 <- S10 C4: RT2 <- PC C5: PC <- RT2 + RT1, salto a fetch</p>	<p>SelP=11, M7, C7 C2: T5, C7, C=0110, B=1, A0=0, MADDR=fetch C3: SE, Size=01010, C4, T3 C4: T2, C5 C5: MA, MB=01, SelCop=1010, MC, T6, C2, M2=0, A0, B, C=0</p>	<p>valor más adelante. Para verificar si ambos valores son iguales se comprueba si su resta es igual a 0 a través del bit Z de SR (al que podemos acceder con C=6 en la unidad de control). Después, se recupera el valor de SR.</p> <p>Se vuelca el valor de SR a un registro temporal a la vez que se actualiza ya que su valor no cambiará hasta el final de ciclo. Permitiéndonos compactar las operaciones en un sólo ciclo.</p> <p>Empleamos las señales A0 = 0 y A1 = NOT(Z) para realizar un salto a fetch, en caso de que no sean iguales vía MDDR = fetch; o siguiente microinstrucción.</p> <p>SE se activa ya que S10 es una dirección <i>relativa</i>, por lo que puede ser negativa.</p>
jal U16	<p>C1: BR(RA) <- PC C2: PC <- U16, salto a fetch</p>	<p>C1: T2, LC, SelC=00001, MR C2: SE=0, Size=10000, Offset=0, T3, C2, M2=0, A0, B, C=0</p>	<p>Se ha incluido el salto a fetch en el último ciclo para ahorrar un ciclo.</p>
jr_ra	<p>C1: PC <- RA, salto a fetch</p>	<p>C1: SelA=00001, MR, T9, C2, M2=0, A0, B, C=0</p>	<p>Se ha incluido el salto a fetch en el último ciclo para ahorrar un ciclo</p>
halt	<p>C1: PC <- 0, SR <- 0, salto a fetch</p>	<p>C1: XCODE=0, T11, M2=0, C2, M7=0, C7, A0, B, C=0</p>	<p>Uso de EXCODE = 0 en vez de R0 para asegurarnos de que siempre se va a sustituir por un 0.</p>
xchb (R _{RE1}), (R _{RE2})	<p>C1: MAR <- R1 C2: MBR <- MP(R1), MAR <- R2 C3: RT1 <- MBR, MBR <- MP(R2) C4: MAR <- R1</p>	<p>C1: C0, T9, SelA=10101, MR=0 C2: R, TA, BW=00, M1, C1, SelA=10000, MR=0, T9, C0 C3: T1,TA, C4, R, BW=00, C1, M1</p>	<p>Se ha incluido el salto a fetch en el último ciclo para ahorrar un ciclo</p> <p>Cargamos un dato en MBR a la vez que</p>

	C5: MP(R1) <- MBR (== MP(R2)), MBR <- RT1 (== MP(R1)) C6: MAR <- R2 C7: MP(R2) <- MBR(==MP(R1 previo)), salto a fetch	C4: C0, T9, SelA=10101, MR=0 C5: TA, W, BW=00, TD, T4, C1, M1=0 C6: SelA=10000, MR=0, T9, C0 C7: TA, W, BW=00, TD, A0, B, C=0	actualizamos MAR ya que MAR mantendrá su valor anterior hasta el final del ciclo, permitiéndonos aunar operaciones en un sólo ciclo. Aplicamos el mismo concepto a la hora de cargar y volcar nuevos datos en MBR.
--	--	--	--

2. Ejercicio 2:

La función deberá transformar una imagen dada de tal manera que se obtenga su traspuesta. Después deberá mostrar la imagen transformada en el dispositivo LED-MATRIX.

Imagen antes de la transformación	Imagen después de la transformación
	

Para aplicar esta transformación hemos aplicado el siguiente algoritmo:

Pseudocódigo
<pre>t1 = 0 while (t1 < 24) { t2 = t1*24 + msg; t2 += t1; //lleva t2 al elemento de la diagonal msg[t1][t1] t3 = t2; while (t2 < row.size) { t2 += 1; t3 += 24; xchb t2, t3 } t1 += 1; } out(msg)</pre>

Este algoritmo se traduce a código ensamblador:

Código en ensamblador

```
demo:
#   t0 contador de diagonal
#   t1 msg
#   t2 index in_row, in_colum: al ser cuadrada podemos usar el mismo
#   t3 1, incremento de contador
#   t4 posición de memoria del elemento ij
#   t5 posición de memoria del elemento ji
#   t6 24, tamaño de fila

# set up, conjunto de valores y constantes necesarios para la
# implementación
lui t0, 0
add t1, zero, a0
lui t3, 1
lui t6, 24
# bucle 1: recorre el triángulo superior de la matriz
loop1:
    beq t0, t6, end_loop1 #condición de salida, 24 vueltas (tamaño de
                           la matriz)
    add t2, t3, t0 # t2 <- t0 + 1
    mul_add t4, t0, t6, t1 # lleva t4 al elemento en la diagonal que
                           toca
    add t4, t4, t0 # t4 <- &msg[t0][t0]
    add t5, zero, t4 # t5 <- &msg[t0][t0]
    # bucle 2: dos registros recorren la fila y columna desde la
    # diagonal, transponiendo los datos.
    loop2:
        beq t2, t6, end_loop2
        add t2, t2, t3
        add t4, t4, t3 # t4 <- &msg[i + 1][t0] iteramos en la fila
        add t5, t5, t6 # t5 <- &msg[t0][j + 1] iteramos en la columna
        xchb (t4) (t5)
        beq zero, zero, loop2 # salto relativo al bucle 2
    end_loop2:
        add t0, t0, t3
        beq zero, zero, loop1 # salto relativo al bucle 1
    end_loop1:
        # mostrar la imagen transformada
        # send address to IO.data
        add t5, zero, t1
        out t5 0x3108

        # send show to IO.control
        lui t5 0x20
        out t5 0x3104
        lui a0 1 # se devuelve un 1 si todo ha sido correcto
        jr_ra
```

2.1. Comparación del juego de instrucciones

La principal diferencia entre el juego de instrucciones de Risc-V y el desarrollado durante esta práctica es la extensión. Risc-V cuenta con 47 instrucciones mientras que nuestro juego solo cuenta con 12 instrucciones. Además, las instrucciones *mul_add* y *xchb* no existen en Risc-V.

Contar con un juego de instrucciones tan reducido ha presentado varias desventajas. Un claro ejemplo es la ausencia de instrucciones como *addi*, que nos permitirían incrementar los contadores mucho más cómodamente y harían el código ligeramente más eficiente y legible; *j_etiqueta* que pudiera funcionar con saltos relativos, ya que la instrucción que hemos microprogramado sólo funciona con direcciones absolutas; y una instrucción como *mv* que nos permitiera copiar el valor de un registro a otro, ya que haría el código, de nuevo, más legible y eficiente. Además, saliendo de los enunciados de la práctica, el juego de instrucciones no cuenta con ninguna operación para números en coma flotante, u otras operaciones que serían necesarias en programas de mayor magnitud o de otra índole.

En cuanto a las ventajas, las instrucciones *mul_add* y *xchb* fueron realmente útiles a la hora de realizar el último ejercicio. La primera para poder hallar rápidamente la dirección de memoria del primer elemento de cualquier fila de la matriz, y la segunda por ser justo la operación requerida para aplicar la transformación a la imagen.

Para poder solventar algunas de las desventajas, consideraríamos oportuno implementar algunas de las instrucciones que echamos en falta. Especialmente la instrucción *mv*, ya que no creemos que fuera una carga extra demasiado grande y presentaría notables beneficios para la legibilidad y desarrollo del código en ensamblador.

3. Horas invertidas

NOMBRE	INDIVIDUAL	EQUIPO	TOTAL
Paula Subías Serrano	2	6.5	8.5
César López	2.5	6.5	9
TOTAL	4.5	13	17.5

4. Conclusiones

La microprogramación es una disciplina interesante y muy útil. Tener un mínimo de conocimiento en esta área permite desarrollar programas más eficientes y conocer más profundamente los computadores. Además, con los conocimientos adquiridos sobre ensamblador, la microprogramación resultó ser algo muy relacionado que ayudó a afianzar estos conocimientos.

No encontramos grandes problemas a la hora de realizar la práctica más allá de superar la barrera de conocimiento necesaria para empezar. No obstante esto no fue tarea difícil y se pudo solucionar tras un par de tardes de estudio y algo de colaboración.

De nuevo, volvemos a destacar la importancia de la planificación *previa al código* para evitar errores. También nos ha parecido muy productivo la división del trabajo y revisión posterior por el otro miembro del grupo. Así como las discusiones sobre distintas implementaciones para llegar a la solución más óptima que fuimos capaces de idear.