Integrantes: Victoria Stefania Schenone Fernández & Juana Arslanian

**Trabajo Práctico 1**

**Microarquitectura**

**Tecnología Digital II**

1)

a) ¿Cuál es el tamaño de la memoria en cantidad de bytes?

a. La memoria tiene 256 bytes de memoria que equivale a 2048 bits, es decir 8 bits por palabra.

b) ¿Cuántas instrucciones sin operandos se podrían agregar al formato de instrucción?

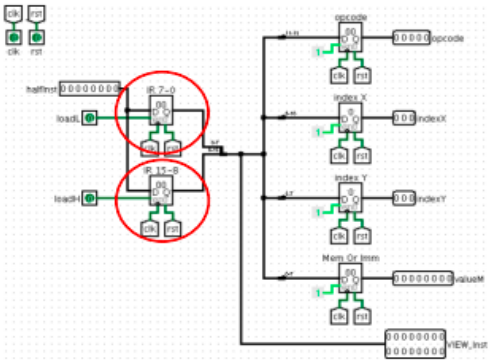
b. Solo se podrían agregar 3 instrucciones nuevas utilizando las instrucciones de opcode 14 (0x0E), 15 (0x0F) y 31 (0x1F), puesto que están reservadas para codificar nuevas instrucciones.

c) ¿Qué tamaño tiene el PC?

c. El tamaño del PC siempre dependerá de las dimensiones de la memoria direccionable, puesto que la memoria es de 256 bytes, el PC necesita poder contar hasta 256 bytes y, para ello, el PC requiere 8 bits, que es la mínima cantidad de bits para poder representarlo ya que es direccionable a 1 byte, y por ende los registros son de 8 bits.

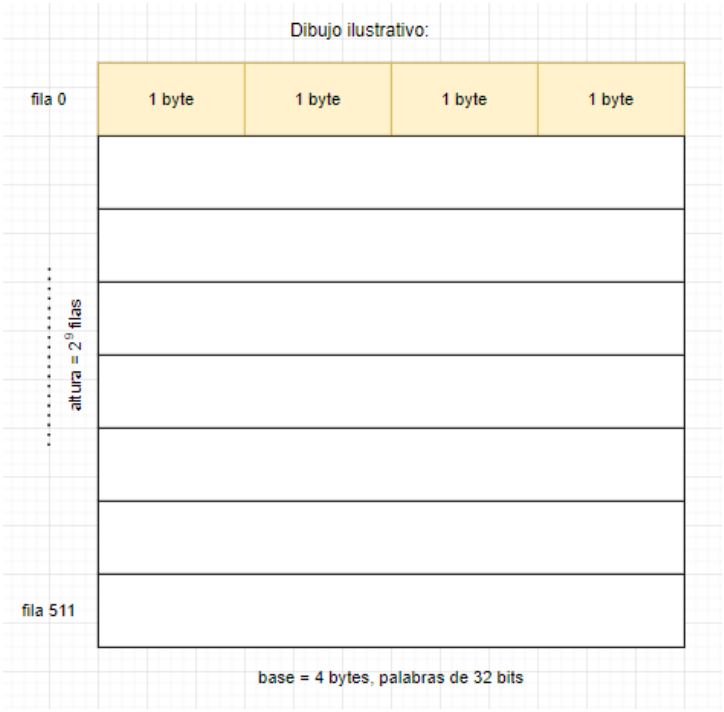
d) ¿Dónde se encuentra y qué tamaño tiene el IR?

d. Se encuentra en la unidad de decodificación de instrucciones (Decode) y el tamaño depende de cuantas instrucciones el procesador pueda manejar, como sabemos que las instrucciones ocupan 2 bytes, serán 16 bits. De este modo, el IR es un puntero de las instrucciones conformado de dos flip-flops de 8 bits cada uno, IR-L para la parte baja y, IR-H para la parte alta.



e) ¿Cuál es el tamaño de la memoria de microinstrucciones? ¿Cuál es su unidad direccionable?

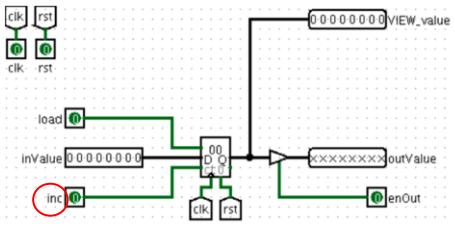
e. El tamaño de la memoria de microinstrucciones es de 2KB porque contiene direcciones de 9 bits, por lo que tendremos $2^9$ filas que es lo que entra en el microPC codificado en hexadecimal con 3 dígitos en la unidad de control, y palabras de 32 bits, direccionables a 1 byte. Entonces, calculamos base por altura: $2^9 * 2^2 = 2^{11} = 2048$ bytes.




2)

a) PC (Contador de Programa): ¿Qué función cumple la señal inc?

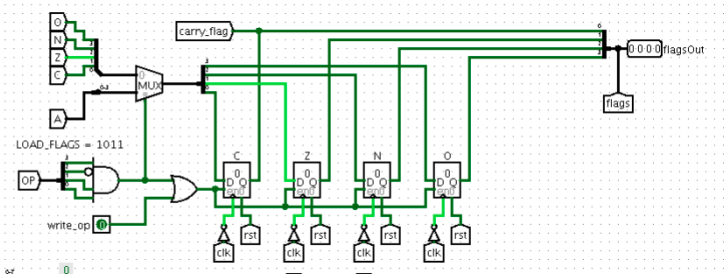
a. La señal inc tiene el papel de aumentar en 1 el valor actual PC, para así ir leyendo las direcciones de memoria y el contenido de cada registro para movernos a la siguiente instrucción, las cuales ocupan 2 bytes.




b) ALU (Unidad Aritmético Lógica): ¿Qué función cumple la señal opW?

b. En nuestra ALU, opW considera si se deben escribir los flips flops, que guardan el valor de los flags. Cuando "opW" se habilita, ya que está conectado a un OR

por lo que en 1 saldrá 1, la ALU modifica los registros de flags, (como C= Carry, Z = Zero, O= Overflow, N = Negative) con la información relevante en función de la operación realizada.




c) Control Unit (Unidad de control): ¿Cómo se resuelven los saltos condicionales? Describir detalladamente el mecanismo, incluyendo la forma en que interactúan las señales jc microOp, jz microOp, jn microOp y jo microOp, con los flags.

c. Cuando se trata de resolver saltos condicionales, es esencial entender **¿qué ocurre con el microPC?** Ponemos atención en el valor del registro microPC (Contador de Microinstrucciones) que se modifica para dirigir la ejecución del programa a una dirección de micro-instrucción específica en lugar de la siguiente microinstrucción secuencial (actuando distinto al PC, que mediante inc +1, guarda la dirección sucesiva).

Justamente el microPC es el puntero que va recorriendo el programa, y si se prende la señal del salto, se carga allí la dirección a la microinstrucción (**con load-microp**) correspondiente al destino del salto condicional.

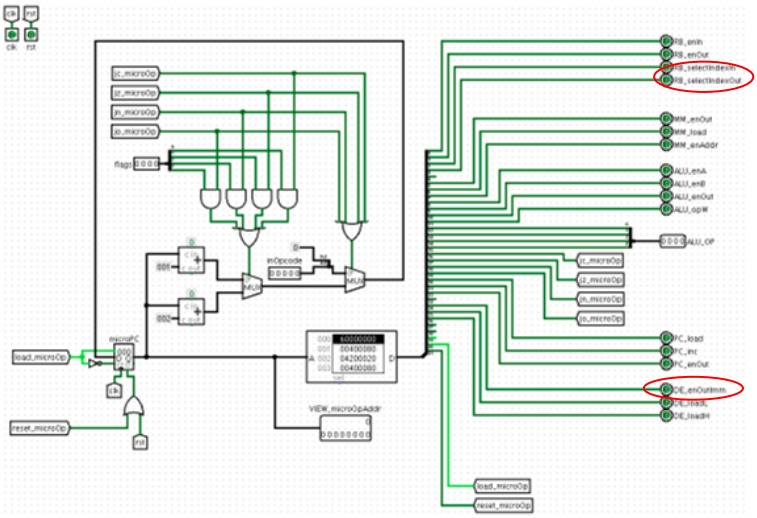
Es importante considerar las compuertas que componen al Control Unit, ya que gracias a ellas interactúan los flags, dado que la señal del flag usa una compuerta AND para habilitar. Si el resultado del AND te da 1, te lleva al microprocesador, mediante el **microp decide que salto deberá hacer (porque el resultado puede darte carry, cero, overflow o negativo)**. También es importante decir que resetea el valor al finalizar la operación, porque puede tener distintos usos dentro del mismo programa.

d) microOrgaSmall (DataPath): ¿Para qué sirve la señal DE enOutImm? ¿Qué parte del circuito indica cuál índice del registro a leer y escribir?

d. **La clave de la señal DE enOutImm es que habilita la entrada al bus de un valor inmediato (o sea una constante)**, mediante el componente de 3 estados. Se utiliza cuando una instrucción requiere un valor inmediato como parte de sus operandos (por ejemplo STRPOP que la implementamos para el 5a) para hacer operaciones aritméticas o lógicas.

**Si está en 0**, no se habilita la salida por lo que se hace presente el hi-z (no es nulo, ni cero, te habilita que lo conectes en un circuito más complejo y hace

que no pase), es decir que está desconectado y por lo tanto no pasa el valor del DECODE al bus.
**Si están ambas en 1** la entrada C y E, el valor inmediato se expone en el bus, lo que permite que otras partes del procesador puedan acceder a él.
La parte del circuito que indica cuál es el índice del registro a leer y escribir se encuentra en la Unidad de control (UC), precisamente son las señales **"RB selectIndexIn" y "RB selectIndexOut"** seleccionan los registros específicos que se deben utilizar para operaciones de lectura y escritura.
Visualmente:



3)

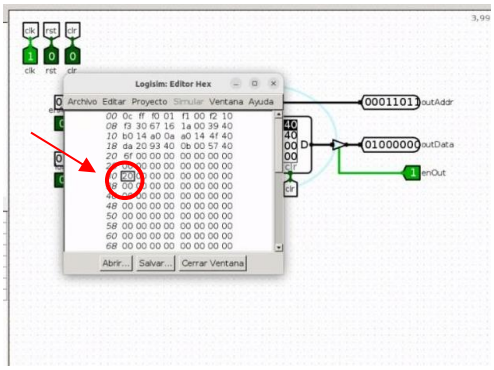
```
start:
        SET R7, 0xFF
        SET R0, 0x01
        SET R1, 0x00
        SET R2, 0x10
        SET R3, 0x30

loop:
        CALL |R7|, shift
        SUB R2, R0
        CMP R1, R2
        JZ end
JMP loop
end:
JMP end
```

```
shift:
        PUSH |R7|, R2
        SHL R2, 1
        STR [R3], R2
        ADD R3, R0
        POP |R7|, R2
        RET |R7|
```

a) Lo que hace el programa es lo siguiente:

- Inicializa registros en ciertos valores: R0 es una constante cuya función es incrementar y decrementar, R7 es el stack pointer, R2 y R3.

- Llama a la función shift mediante el CALL.
- Preserva el valor del registro R2, lo multiplica por dos, ya que se corre 1 lugar a la izquierda, y guarda el resultado de la multiplicación en la dirección de memoria del registro R3.
- Recupera el valor de R2 para así después seguir con su ejecución.

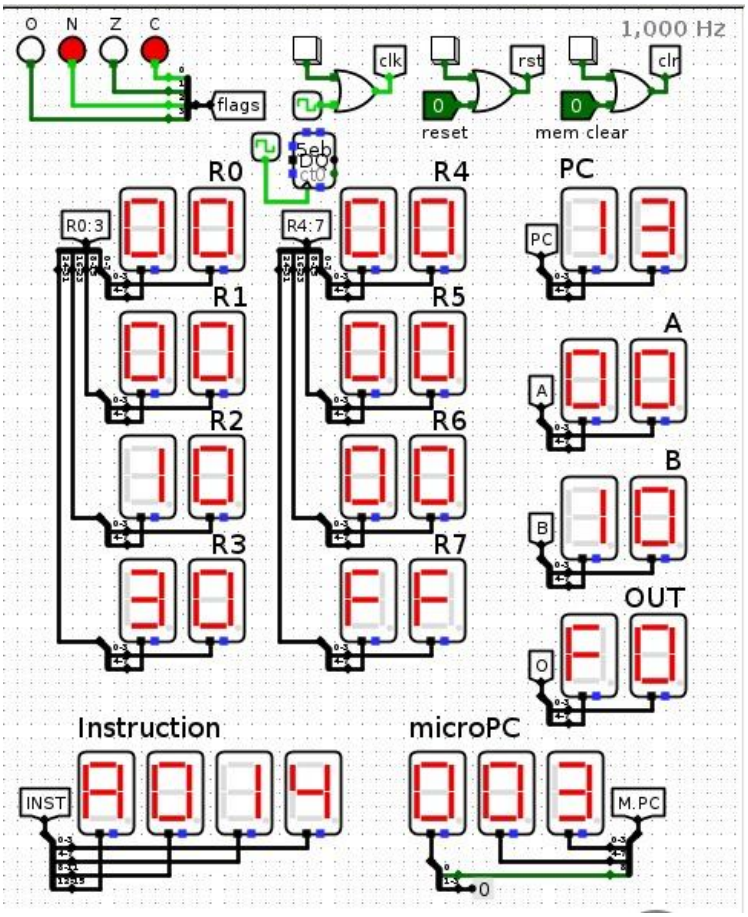



- A partir de entonces, irá guardando en la memoria; desde la dirección 0x30 en adelante, los valores del resultado de multiplicar los números desde el 0x10 (16 en decimal) hasta el 0x00.

b) Las etiquetas pueden ser cualquier cadena de caracteres finalizada en ":", y en este caso son:

- Start: |0x00|
- Shift: |0x14|
- Loop: |0x0a|
- End: |0x20|

c) El programa necesita 1515 (0x5eb) ciclos de clock para llegar a la instrucción de JMP end por primera vez porque cada instrucción está compuesta de varias microinstrucciones que ocupan más de un ciclo de clock, dado que se debe realizar el fetch, decodificar y enviar las señales correspondientes. Por ejemplo, ir a buscar una dirección de memoria y almacenarla puede tardar bastante, además, teniendo en cuenta que esa instrucción es ejecutada muchas veces. Así, el programa no finaliza con su ejecución sino que queda en un ciclo, esto es porque es una máquina simple. Para calcular los ciclos de clocks colocamos un contador de 9 bits con máximo valor 0x5ff.




d) Para ejecutar las microinstrucciones necesitamos para cada caso:

- ADD: 5 microinstrucciones.
- JZ: 3 si se realiza el salto, si no solamente son 2.
- JMP: 2 microinstrucciones, ya que el salto es incondicional.

  Nos guiamos mediante las filas de código del archivo .ops ya que cada una equivale a un ciclo del clock.

e) Podemos notar que las cuatro instrucciones trabajan en conjunto siendo las únicas que operan con la pila, con el fin de poner el stack pointer en su valor original antes de volver a mi programa. Veamos el funcionamiento de cada instrucción en particular:

   ★ PUSH: Almacena un valor de un registro en la dirección de memoria en la pila que señala el valor de otro registro y lo decrementa en 1 para ajustar el puntero.

   ★ POP: Quita un dato de la pila, para recuperar el valor original del stack pointer y si se debe utilizar en combinación con el PUSH, estrictamente. Con la idea de que “el último registro pusheado” será mi primer POP porque

funciona mediante el algoritmo de planificación LIFO, y así liberar ese espacio de la pila.

★ CALL: Salta a una subrutina que en este caso es “shift” y guarda la dirección de retorno en la pila. Previamente de ir a la etiqueta “shift” guarda el código que tenía antes para no olvidarselo, se actualiza el Program Counter, guardando el valor en algún lugar de la memoria y lo reemplaza por la dirección de shift.

Pasos importantes del CALL:

★ PC ← shift

★ R7 ← R7 -1 = 0XFF - 0X01 = 1111 1111 - 0000 0001 = 255 - 1 = 254 decimal

★ Memoria[R7] sería el contenido de R7 que es → 1111 1110 = 0xFE

★ RET: Funciona de forma similar a un POP, pero la diferencia es el destino de la instrucción que es el PC. Logra recuperar la dirección almacenada en la pila original, es decir transfiere el dato al registro inicial, cargando en el PC la dirección, regresando así al programa principal.

4)

a) **STRPOP:** Modificamos el common.py para poder compilarlo correctamente, y aclaramos los cambios realizados en el archivo, el cual se encuentra tanto en *micro* y *tools* para que se compile correctamente.

- Lo agregamos en TYPE_SR, porque funciona similarmente al CALL:

```
elif i[0]=="STRPOP":
    n=buidInst({"O": opcodes[i[0]], "X": reg2num(i[2]),
"M":mem2num(i[5], labels)})
```

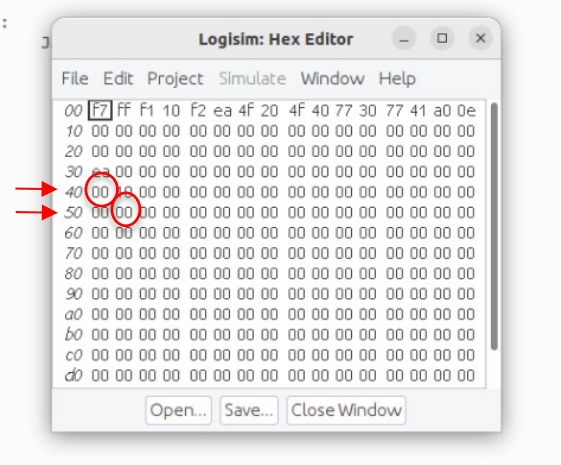
<u>Microinstrucción:</u>

```
01110: ; STRPOP |Rx|, M
   ALU_enA                  RB_enOut RB_selectIndexOut=0          ; A <- Rx
   ALU_enB                  ALU_enOut ALU_OP=cte0x01              ; B <- 1
   RB_enIn RB_selectIndexIn=0     ALU_enOut ALU_OP=ADD          ; Rx <- Rx
+ 1
   MM_enAddr                RB_enOut RB_selectIndexOut=0          ; addr <-
Rx
   ALU_enA                  MM_enOut                              ; A <- [Rx]
   ALU_enB                  ALU_enOut ALU_OP=cte0x00               ; B <- 0
   ALU_OP=ADD                  MM_enAddr DE_enOutImm         ; alu_out <-
mem_out y  addr <- [Rx]
```

MM_load ALU_enOut ; [M] <- alu_out
reset_microOp

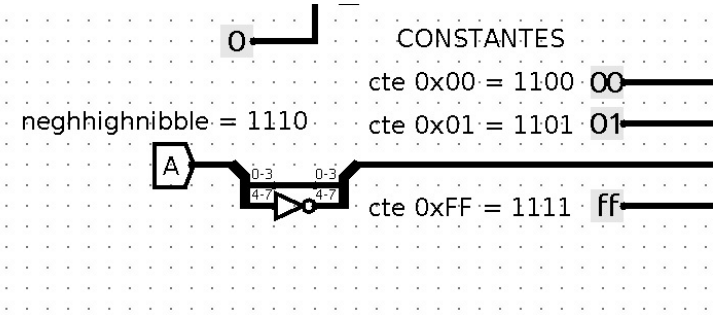
Testing:
main:
SET R7, 0xFF ;stack
SET R1, 0x10
SET R2, 0xEA
PUSH |R7|, R1
PUSH |R7|, R2
STRPOP |R7|, 0x30
STRPOP |R7|, 0x41

halt:
JMP halt
Resultado esperado:
[0x30] ← 0xEA
[0x41] ← 0x10




b) **NEGHIGHNIBBLE:**
Microinstrucción:
ALU_enA RB_enOut RB_selectIndexOut=0 ; A <- Rx
RB_enIn RB_selectIndexIn=0 ALU_enOut ALU_OP=14 ; Rx <- Rx inverso
reset_microOp

El circuito del inverso de bits se encuentra en la operación 14:




- Negamos los 4 bits más significativos con la compuerta NOT GATE.

Testing:
main:
SET R1, 0xF0

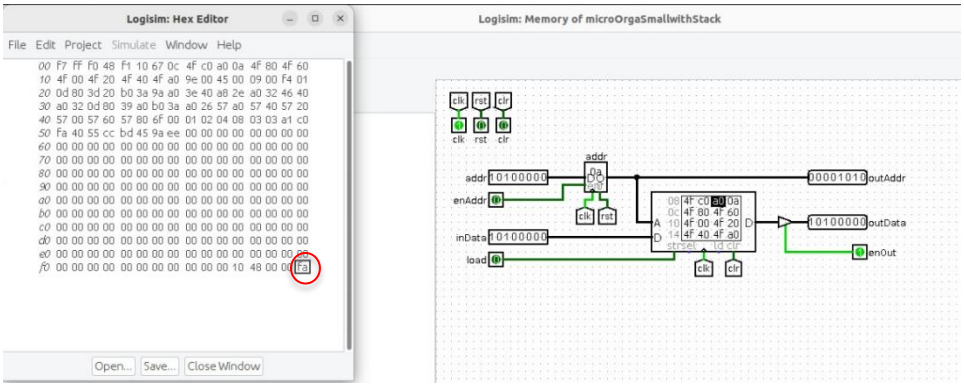
```
    SET R2, 0x0A
    SET R3, 0xE1
    SET R4, 0xA3
    ; TENGO R1 EN LOS PRIMEROS DB → “es como hacer NEGHIGHNIBBLE DE R1”
    DB 0X79
    DB 0X00 ; Completamos con ceros ya que necesitamos 16 bits para formar la instrucción
    ;TENGO R2 EN LOS SEGUNDOS DB → “es como hacer NEGHIGHNIBBLE DE R2”
    DB 0X7A
    DB 0X00 ;Idem a lo explicado anteriormente
    ;Tengo R3 EN LOS TERCEROS DB → → “es como hacer NEGHIGHNIBBLE DE R1”
    DB 0X7B
    DB 0X00 ; Idem a lo explicado anteriormente
    ; Tengo R4 EN LOS CUARTOS DB
    DB 0X7C
    DB 0X00

halt:
        JMP halt
```

**Resultado esperado:**

R1 = 0XF0 → 1111 0000 → 0X00
R2 = 0X0A → 1111 1010 → 0XFA
R3 = 0XE1 → 0001 0001 → 0X11
R4 = 0XA3 → 0101 0011 → 0X53

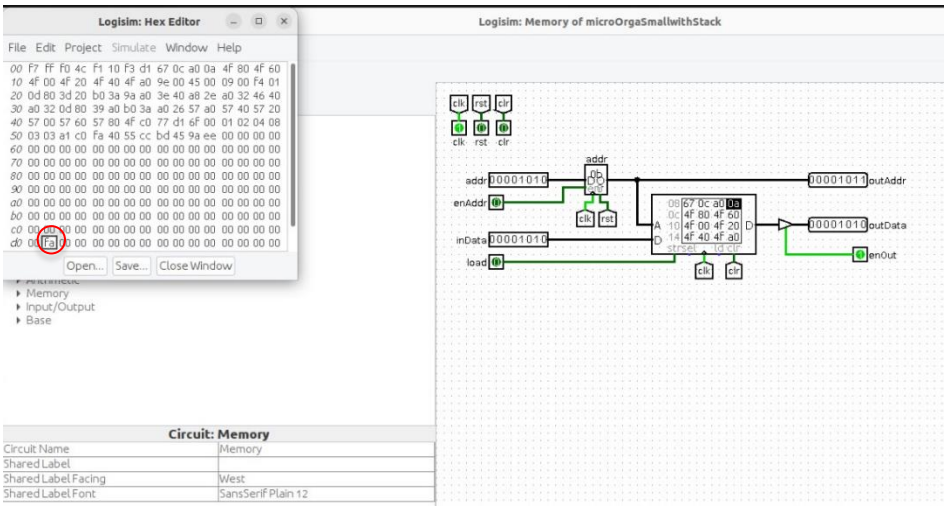
5)

a) Ver archivo llamado ej5a.asm
Imagen de cómo se guarda el máximo en la pila al haber recorrido todo el array.



Otra opción sería considerar que la ALU opera en complemento a 2 por lo que su rango está limitado de -128 a 127, por lo que en ese caso podría modificarse el código y usar JN en vez de JC, y tomar como valor máximo aceptado a 0x7E.

b) Ver archivo llamado ej5b.asm
Imagen de cómo el máximo se guarda en la dirección pasada, en este caso 0xD1.




c) Ver archivo llamado ej5c.asm