Trabajo practico numero 1

Microarquitectura

integrantes: Matias Espinel, Ivo Etcheverry y Santino Barontini

1) **. Trabajo previo - Estudiar la documentación**

- **Leer la hoja de datos del procesador.**
- **Estudiar los circuitos que implementan la máquina.**
- **Identificar todas las instrucciones y su funcionamiento.**
- **Determinar los tamaños de las instrucciones, memoria y memoria de microinstrucciones.**
    - **Tamaños de todas las instrucciones y las microinstrucciones:**
    - Las instrucciones están todas codificadas en 16 bits con los primeros 5 bits de identificación de upcode y el resto identifican los parámetros. Estas trabajan con bancos de memoria (0 a 7 en índice)
    - Estas son de 16 bits.
    - **Memoria de Microinstrucciones:** 2 kilobytes (4 bytes x 2^9 bits)
    - **Memoria: 256 bytes**
- **Analizar y estudiar el código python provisto junto a la máquina.**

2) **Ensamblar y ejecutar - Escribir el siguiente archivo, compilarlo y cargarlo en la memoria de la máquina:**
    a) **Previamente a ejecutar el programa, describir con palabras el comportamiento esperado del mismo. No se debe explicar instrucción por instrucción, la idea es entender que hace el programa y que resultado genera.**

Al inicio del programa se setean valores a diferentes bancos de registros(5 bancos), dentro de la función se declaran 3 etiquetas, Loop, shift y end.
Se salta a la etiqueta Loop mediante JMP, este por dentro salta a la etiqueta shift:

- Shiftea el número en R2 (básicamente multiplica por 2 el número), lo expone?? en R3 y se le suma 1 a R3 para que el próximo valor generado no se sobreescriba
- se le resta 1 a R2 que es 16 y se lo compara a 0, como este no es cero no se dispara la etiqueta de end
- Esto se repite hasta que R2 sea 0 y se dispare la la rutina de end, dejando como resultado:
    - 2x16
    - 2x15
    - 2x14
    - 2x13

- 2x12
- 2x10
- 2x9
- 2x8
- 2x7
- 2x6
- 2x5
- 2x4
- 2x3
- 2x3
- 2x2
- 2x1

- y ahí entra en el bucle infinito de end.

b) Identificar la dirección de memoria de cada una de las etiquetas del programa.

Direcciones de byte, por ende cada instrucción ocupa 2 direcciones de memoria.

```
start |00| SET R7 , 0xFF
    |02| SET R0 , 0x01
    |04| SET R1 , 0x00
    |06| SET R2 , 0x10
    |08| SET R3 , 0x30
 loop |0a| CALL | R7 | , shift
    |0c| SUB R2 , R0
    |0e| CMP R1 , R2
    |10| JZ end
shift |12| PUSH | R7 | , R2
    |14| SHL R2 , 1
    |16| STR [ R3 ] , R2
    |18| ADD R3 , R0
    |1a| POP | R7 | , R2
    |1c| RET | R7 |
    |1e| JMP loop
  end |20| JMP end
```

- start: en 0x00
- loop: en 0x0A (ahí justo arranca el CALL)
- end: en 0x20 (el JMP end)
- shift: en 0x12 (la rutina con PUSH)

c) Ejecutar e identificar cuántos ciclos de clock son necesarios para que el programa llegue a la instrucción JMP halt.

el programa llega JMP end (mismo que halt) cuando hay flag de 0. (JZ end) Este flag de 0 se produce cuando R2 llega a 0, que es 16 y se le está restando 1.
348? 5 del start, 15 veces el CALL, 16 veces el resto de las instrucciones de loop, y 15 veces el shift.

Cada instrucción tarda la cantidad de ciclos de clock que su cantidad de microinstrucciones, por ende para llegar al JMP halt se necesitan (2 micro x set)+ (7x call)x15 + (5x sub)x16 + (4xCMP) x 16 + (4xJZ) x 16 + 15x(6 +3+ 3+ 5 + 6 +6 + 2) + 2= . **790 ciclos de clock (contando el reset y sumando los últimos 2 de la ejecución del end.** .

d) ¿Cuántas microinstrucciones son necesarias para ejecutar la instrucción ADD? ¿Cuántas para la instrucción JZ? ¿Cuántas para la instrucción JMP?

- ADD (**5 MICRO**):

```
ALU_enA  RB_enOut RB_selectIndexOut=0 ; A <- Rx
ALU_enB  RB_enOut RB_selectIndexOut=1 ; B <- Ry
ALU_OP=ADD  ALU_opW
RB_enIn  RB_selectIndexIn=0  ALU_enOut ; Rx <- Rx + Ry
reset_microOp
```

- JZ (**4 MICRO**):

```
JZ_microOp  load_microOp ; if Z then microOp+2 else microOp+1
reset_microOp
PC_load     DE_enOutImm  ; PC <- M
reset_microOp
```

- JMP (**2 MICRO**):

```
PC_load    DE_enOutImm   ; PC <- M
reset_microOp
```

e) Describir detalladamente el funcionamiento de las instrucciones PUSH, POP, CALL y RET.

- **PUSH:**Función: Guardar un valor en la pila.
  1)Se escribe en la dirección de la memoria apuntada por SP
  2)La pila crece hacia abajo
  3) En este caso se utiliza para “salvar” el valor original de R2 y al mismo tiempo poder trabajar con este
- **POP:**Función: Recupera (desapila) el valor de la cima de la pila y lo coloca en el operando.
  1)Se toma el valor almacenado en la dirección de memoria apuntada por SP.
  2)Ese valor se copia en el operando destino.
  3)El puntero de pila SP se incrementa en el tamaño del operando.
  4) En este caso estamos das apilando R2 para que vuelva a su valor original
- **CALL:** Llama a una subrutina, es decir transfiere el control a otra parte del programa guardando la dirección de retorno en la pila.
  1) hace un PUSH implícito del PC en la pila y luego salta a la subrutina (por registro o inmediato).
  2) En este caso pasamos a la subrutina de shifteo
- **RET:** Vuelve desde la subrutina usando la dirección guardada en la pila.
  1)El procesador va a la pila

2)Recupera la dirección que se había guardado con el CALL
3)Salta/ Va a esa posición, siguiendo el programa desde donde se había quedado.

Programar - Escribir en ASM las siguientes funciones:

a) Escribir la funcion cantPares que toma un array de enteros positivos en memoria y cuenta cuantos elementos pares tiene.

```
cantPares(*p,size)
        for f=0; f<size; f++
                if (p[f] es par) count++
        return count
```

```
main:
        SET R7, 0xFF ;stack

        SET R0, p       ;p
        SET R1, 0x10 ;size
        SET R4, 0x00
        CALL |R7|, cantPares

halt:
        JMP halt

cantPares:
   PUSH |R7|, R0   ; apilo reg
   PUSH |R7|, R1
   PUSH |R7|, R2
   PUSH |R7|, R3
   PUSH |R7|, R6

   SET R4, 0x00    ; contador = 0
   SET R2, 0x01    ; constante 1
   SET R6, 0x00    ; constante 0 para comparación

loop:
   CMP R1, R6      ; size == 0?
   JZ fin          ; Si es cero, terminar.

   LOAD R3, [R0]   ; R3 = *p
   AND R3, R2      ; R3 = R3 & 1 (chequeo de paridad)
   JZ par          ; Si es cero (par), saltar a 'par'

; impar :
```

```
   JMP update ; Si es impar, solo actualiza

par:
   ADD R4, R2      ; contador++

update:
   ADD R0, R2      ; p++
   SUB R1, R2      ; size--
   JMP loop

fin:
   POP |R7|, R6    ; Restaurar registros
   POP |R7|, R3
   POP |R7|, R2
   POP |R7|, R1
   POP |R7|, R0

         RET |R7|

p:
   DB 0x01
   DB 0x02
   DB 0x04
   DB 0x08
   DB 0x03
   DB 0x03
   DB 0xA1
   DB 0xC0
   DB 0xFF
   DB 0x40
   DB 0x55
   DB 0xCC
   DB 0xBD
   DB 0x45
   DB 0x9A
   DB 0xEE
```

b) Escribir otra funci´on modArray que toma el mismo array del punto anterior y modifica sus valores, dividiendo por 4 y restando uno a los multiplos de 4 y multiplicando por 5 y restando uno al resto.

```
modArray(*p,size)
        for f=0; f<size; f++
                if (p[f] es mult4) p[f]=p[f]/4
```

else p[f]=p[f]*5-1

```
main:
        SET R7, 0xFF   ;stack
        SET R0, p      ;p
        SET R1, 0x10 ;size
        CALL |R7|, modArray

halt:
        JMP halt

modArray:
   PUSH |R7|, R0
   PUSH |R7|, R1
   PUSH |R7|, R2
   PUSH |R7|, R3
   PUSH |R7|, R5
   PUSH |R7|, R6

   SET R2, 0x01       ; constante 1
   SET R5, 0x00       ; constante 0
   SET R6, 0x03       ; constante 3

   loop:
      CMP R1, R5         ; size == 0
      JZ fin

      LOAD R3, [R0]      ; R3 = *p

      AND R3, R6         ; R3 AND 3 (11)
      JZ mult
      JMP resto
   resto:
      LOAD R3, [R0]   ; volver a cargar valor original
      MOV R4, R3         ; copia valor original a R4
      SHL R3, 1          ; ×4
      SHL R3, 1
      ADD R3, R4         ; + original → ×5
      SUB R3, R2         ; -1
      STR [R0], R3       ; guardar en memoria
      ADD R0, R2
      SUB R1, R2
```

```
    JMP loop

mult:
    LOAD R3, [R0]   ; volver a cargar valor original
    SHR R3, 1        ; /4
    SHR R3, 1
    SUB R3, R2       ; -1
    STR [R0], R3     ; guardar en memoria
    ADD R0, R2
    SUB R1, R2
    JMP loop

fin:
    POP |R7|, R6
    POP |R7|, R5
    POP |R7|, R3
    POP |R7|, R2
    POP |R7|, R1
    POP |R7|, R0

        RET |R7|

p:
DB 0x01
DB 0x02
DB 0x04
DB 0x08
DB 0x03
DB 0x03
DB 0xA1
DB 0xC0
DB 0xFF
DB 0x40
DB 0x55
DB 0xCC
DB 0xBD
DB 0x45
DB 0x9A
DB 0xEE
```

Considerar en todos los casos que los parámetros llegan en R0 y R1. El resultado debe ser guardado en R4. Considerar que ningún registro debe ser modificado, excepto el R4.

Para este ejercicio se proporciona un conjunto de archivos base donde pueden completar la implementación de sus funciones. Estos archivos pueden ser modificados para complementar su entrega
con otros ejemplos de datos de entrada.

4)

a) **Sin agregar circuitos nuevos, agregar la instrucción STRPOP que almacena en la dirección pasada como parámetro el número almacenado en la pila. Esta instrucción debe dejar la pila consistente, es decir, quitando el dato de la pila. Se recomienda utilizar como código de operación el 0x0E**

```
01110: ; STRPOP |Rx|, M   ; Rx es SP (crece hacia arriba: POP = Rx <- Rx + 1)

  ; Rx <- Rx + 1
  RB_selectIndexOut=0   RB_enOut     ALU_enA
  ALU_OP=cte0x01        ALU_enOut    ALU_enB
  ALU_OP=ADD            ALU_enOut    RB_selectIndexIn=0   RB_enIn

  ; A <- Mem[Rx]        (primero fijó la dirección, luego leo)
  RB_selectIndexOut=0   RB_enOut     MM_enAddr
  MM_enOut              ALU_enA

  ; Addr <- M           (fijo dirección destino inmediata)
  DE_enOutImm           MM_enAddr

  ; Bus <- A (passthrough con +0)  y escribo en Mem[M]
  ALU_OP=cte0x00        ALU_enOut    ALU_enB
  ALU_OP=ADD            ALU_enOut    MM_load

  reset_microOp
```

b) **Agregar la instrucci´on NEGHIGHNIBBLE, que hace el inverso de los 4 bits más significativos de un registro sin modificar los otros 4. El resultado se almacena en el mismo registro. Para implementar esta instrucción se debe modificar el circuito de la ALU para la operación 14. Bajo este código se debe agregar la operación de la ALU que realiza el intercambio de bits. Se recomienda utilizar como código de operación el 0x0F**

```
01111: ; NEGHIGHNIBBLE
RB_enOut ALU_enA RB_selectIndexOut=0
ALU_OP=NEGHIGHNIBBLE ALU_opW
RB_enIn ALU_enOut RB_selectIndexIn=0
reset_microOp
```

Para esta consigna modificamos el common.py para que la ALU reconozca la instrucción ya que sino no compilaba el.mem, en la parte del código de ALUops sume NEGHIGHNIBBLE con valor 14, también en la sección de opcodes sume STRPOP y NEGHIGHNIBBLE con sus respectivos valores de operación, y por último sume STRPOP a la sección de type_sr sumado a su propio elif en python para reconocer los caracteres.