Trabajo Práctico 3: Microarquitectura

Tecnología Digital II - Segundo Semestre
Maico Grimaldi, Mateo Flynn y Luz Alba Posse

Ejercicio 1

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

Como la memoria tiene 256 palabras, y cada palabra tiene 8 bits, entonces la misma tiene 256 bytes.

b) ¿Cuántas instrucciones sin operandos y con OPCODEs distintos se podrían agregar al formato de instrucción?

Las instrucciones sin operandos y con OPCODEs diferentes que se pueden agregar son 3, ya que son las instrucciones que están libres.

c) ¿Cuántas instrucciones sin operandos y con los OPCODEs existentes se podrían agregar al formato de instrucción?

La cantidad de instrucciones a agregar al formato de instrucción es:

$$16 \times ((2^5 - 1) \times 2^6) + 3 \times ((2^8 - 1) \times 2^3) + 5 \times ((2^3 - 1) \times 2^{11}$$

d) ¿Es posible agregar la instrucción “SUB Rx, M” que haga “Rx=Rx-M” que se codifique como 01101 XXXMMMMMMMM?

Agregar la instrucción no es viable, ya que se requeriría un bit extra de OPCODE para distinguir entre las dos instrucciones que comparten el código 01101, lo que conllevaría a la incapacidad de leer el último bit de la dirección de memoria pasada como parámetro.

e) ¿Qué tamaño tiene el PC?

Tiene 8 bits (1 byte).

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

Está en el decoder, ingresan 8 bits por high y 8 bits por low, teniendo así un total de 16 bits (2 bytes).

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

Tiene palabras de 32 bits, siendo la unidad direccionable, y direcciones de 9 bits, lo que implica una memoria de 2^9 posibles direcciones. Si cada dirección es de 32 bits, la memoria es de 2 KB direccionable a 4 bytes.

Ejercicio 2

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

Su función es incrementar el valor almacenado en el PC, está directamente conectada a la unidad de control, lo que resulta en el aumento del valor actual del registro seleccionado. Este incremento permite realizar la búsqueda (fetch) de la instrucción siguiente en la secuencia de ejecución del programa.

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

Habilita la escritura de los flip flops que almacenan el valor de los flags de la ALU.

c) ControlUnit (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.

Se resuelven en la UC. Cuando se llama a las instrucciones, se tiene que evaluar con las compuertas AND si el flag en cuestión está encendido o no, si está entonces devuelve un 1 y se activa la microinstrucción que genera el salto.

Dependiendo del tipo de salto condicional, se evaluará una condición específica utilizando los flags. Las señales de micro-operación jc, jz, jn y jo se activarán si se cumple la condición correspondiente.

- JC: Se activa si el bit de carry (C) es igual a 1.
- JZ: Se activa si el bit de cero (Z) es igual a 1.
- JN: Se activa si el bit de negativo (N) es igual a 1.
- JO: Se activa si el bit de overflow (O) es igual a 1

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

DE_enOutImm habilita el componente de tres estados que habilita la salida del Decoder al BUS de los datos inmediatos, entonces las señales que salen de la Unidad de control van a indicar qué registro leer y cuál escribir (si es 0 va a ser RX y si es 1 será RY).

Ejercicio 3

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.

Empieza inicializando los registros con valores específicos. Después, entra en un ciclo, donde por cada iteración se llama a *shift*. Entonces, se resta R0 con R2. Luego, se compara el valor en R1 con el valor en R3. Si son iguales, va a saltar a *end*. Sino, va a volver a empezar el ciclo. Este ciclo va a iterar 16 veces.

Shift hace una serie de operaciones. Primero, ya que R7 apunta al primera dirección libre de la pila, tras el PUSH, se guarda el valor de R3 en el stack, así se preserva R3 con su valor original y posibilitando el volver a esté después. Después, realiza un shift right en R3, moviendo esos bits dos posiciones hacia la derecha (como dividir por 4), se almacena en una dirección de memoria. Luego, recupera el valor que está en el stack de R3 y restaura el valor guardado de R7. Seguidamente, resta el valor de R0 con R3 y lo guarda en R3. Finalmente, retorna la subrutina usando el valor en R7, lo que hace que retorne a la línea en que se llamó.

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

Las direcciones de memoria que se encuentran al ejecutar el código son las siguientes:

- start: `0x00`
- loop: `0x0A`
- shift: `0x16`
- end: `0x14`

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

Se necesitan 1513 ciclos de clock para que el programa llegue a la instrucción JMP end. Esto se puede identificar al ejecutar el programa, en el momento en el que la instrucción es A014 y el R3 es 00, ya que A0 es 10100 en binario, siendo este el opcode de la instrucción JMP y la dirección de end previamente hallada es que es 0x14.

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

Por cada instrucción se deben contar las microinstrucciones de fetch(6), que no son representadas aqui debajo, por esto es que en el resultado se verá la suma de las microinstrucciones de cada operación más las del fetch. Como cada línea corresponde a una microinstrucción, ADD tiene 12 microinstrucciones.

```
00001: ; ADD
    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
```

De la misma forma, JZ tendrá 2 o 3 microinstrucciones dependiendo de a qué parte del ciclo salte.

```
10110: ; JZ
    JZ_microOp load_microOp ; if Z then microOp+2 else microOp+1
    reset_microOp
    PC_load    DE_enOutImm  ; PC <- M
    reset_microOp
```

Por último, JMP tiene 3 microinstrucciones.

```
10100: ; JMP
    PC_load    DE_enOutImm   ; PC <- M
    reset_microOp
```

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

PUSH se encarga de cargar un valor desde el registro RX a la memoria y luego cargar un valor desde el registro RY a la misma dirección de memoria. Primero, se carga la dirección de memoria contenida en el registro RX en el módulo de memoria (MM_enAddr). Luego, se carga el valor almacenado en el registro Ry en la dirección de memoria especificada (MM_load). A continuación, se realiza una operación aritmética donde se resta 1 al valor contenido en el registro Rx (ALU_enA, ALU_enB, RB_enIn con ALU_OP=SUB). En la úlima línea, se reinicia el micro-operador (reset_microOp).

```
01001: ; PUSH |Rx|, Ry
    MM_enAddr                       RB_enOut RB_selectIndexOut=0
    MM_load                         RB_enOut RB_selectIndexOut=1
    ALU_enA                         RB_enOut RB_selectIndexOut=0
    ALU_enB                         ALU_enOut ALU_OP=cte0x01
    RB_enIn RB_selectIndexIn=0      ALU_enOut ALU_OP=SUB
    reset_microOp
```

Por otro lado, POP toma un valor desde la memoria y lo carga en el registro RY, luego incrementa el valor del registro RX en 1. Primero, se carga el valor contenido en el registro RX en el módulo de la ALU como A (ALU_enA). Luego, se carga el valor constante 1 en la ALU como B (ALU_enB, ALU_OP=cte0x01). A continuación, se realiza una operación aritmética donde se suma 1 al valor contenido en el registro Rx (RB_enIn con ALU_enOut y ALU_OP=ADD). Después, se carga la dirección de memoria contenida en el registro RX en el módulo de

memoria (MM_enAddr). Al final, se carga el valor almacenado en esa dirección de memoria en el registro Ry (RB_enIn con MM_enOut).

```
01010: ; POP  |Rx|, Ry
    ALU_enA                         RB_enOut RB_selectIndexOut=0
    ALU_enB                         ALU_enOut ALU_OP=cte0x01
    RB_enIn RB_selectIndexIn=0      ALU_enOut ALU_OP=ADD
    MM_enAddr                       RB_enOut RB_selectIndexOut=0
    RB_enIn RB_selectIndexIn=1      MM_enOut
    reset_microOp
```

Así, CALL realiza una llamada a una dirección de memoria M almacenando la dirección de retorno en el registro RX. Primero, se carga la dirección de memoria M en el módulo de memoria (MM_enAddr). Luego, se carga el valor actual del contador de programa (PC) en la dirección de memoria especificada (MM_load). Después, se realiza una operación aritmética donde se resta 1 al valor contenido en el registro Rx (ALU_enA, ALU_enB, RB_enIn con ALU_OP=SUB). Finalmente, se carga la dirección de memoria M en el contador de programa (PC_load con DE_enOutImm).

```
01011: ; CALL |Rx|, Ry
    MM_enAddr                       RB_enOut RB_selectIndexOut=0
    MM_load                         PC_enOut
    ALU_enA                         RB_enOut RB_selectIndexOut=0
    ALU_enB                         ALU_enOut ALU_OP=cte0x01
    RB_enIn RB_selectIndexIn=0      ALU_enOut ALU_OP=SUB
    PC_load                         RB_enOut RB_selectIndexOut=1
    reset_microOp

01100: ; CALL |Rx|, M
    MM_enAddr                       RB_enOut RB_selectIndexOut=0
    MM_load                         PC_enOut
    ALU_enA                         RB_enOut RB_selectIndexOut=0
    ALU_enB                         ALU_enOut ALU_OP=cte0x01
    RB_enIn RB_selectIndexIn=0      ALU_enOut ALU_OP=SUB
    PC_load                         DE_enOutImm
    reset_microOp
```

Por último, RET retorna a una dirección de memoria contenida en el registro Rx. Primero, se carga el valor contenido en el registro Rx en el módulo de la ALU como A (ALU_enA). Luego, se carga el valor constante 1 en la ALU como B (ALU_enB, ALU_OP=cte0x01). A continuación, se realiza una operación aritmética donde se suma 1 al valor contenido en el registro Rx (RB_enIn con ALU_enOut y ALU_OP=ADD). Después, se carga la dirección de memoria contenida en el registro Rx en el módulo de memoria (MM_enAddr). Para terminar, se carga el valor almacenado en esa dirección de memoria en el contador de programa (PC_load con MM_enOut).

```
01101: ; RET  |Rx|
    ALU_enA                         RB_enOut RB_selectIndexOut=0
    ALU_enB                         ALU_enOut ALU_OP=cte0x01
    RB_enIn RB_selectIndexIn=0      ALU_enOut ALU_OP=ADD
```

```
    MM_enAddr                       RB_enOut RB_selectIndexOut=0
    PC_load                         MM_enOut
    reset_microOp
```

Ejercicio 4

a) Escribir la función cantImpares que toma un array de enteros positivos en memoria y cuenta cuantos elementos impares tiene.

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

       CALL |R7|, cantImpares

halt:
       JMP halt

cantImpares:
       PUSH |R7|, R0
       PUSH |R7|, R1
       PUSH |R7|, R3
       PUSH |R7|, R5
       PUSH |R7|, R6
       SET R4, 0x00 ; count
       SET R3, 0x01
       SET R5, 0x00 ; indice
      JMP loop

       loop:
              CALL |R7|, shift
              SUB R1, R3
              CMP R1, R5
              JZ ret
              JMP loop

       shift:
              LOAD R6, [R0]
              ADD R0, R3
              SHL R6, 7
              CMP R6, R5
              JZ par
              ADD R4, R3
              RET |R7|
       par:
              RET |R7|

        ret:
              POP |R7|, R6
              POP |R7|, R5
              POP |R7|, R3
              POP |R7|, R1
              POP |R7|, R0
```

```
            RET|R7|
```

b) Escribir otra funcíon `modArray` que toma el mismo array del punto anterior y modifica sus valores, dividiendo por 2 y restando 2 a los múltiplos de 4 y multiplicando por 7 y sumando uno al resto.

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

       PUSH |R7|, R0
       PUSH |R7|, R1
       PUSH |R7|, R2
       PUSH |R7|, R3
       PUSH |R7|, R4
       PUSH |R7|, R5
       PUSH |R7|, R6
       SET R2, 0x00
       SET R3, 0x01
       SET R6, 0x03

       CALL |R7|, modarray

       POP [R7], R6
       POP [R7], R5
       POP [R7], R4
       POP [R7], R3
       POP [R7], R2
       POP [R7], R1
       POP [R7], R0

halt:
       JMP halt

modarray:
       loop:
              LOAD R4,[R0];    cargo el primer número del array en R4
              SUB R1, R3
              CMP R1, R2
              JN fin
              PUSH [R7], R4
              AND R4, R6;      máscara entre 000011 y R4
              CMP R4, R2
              JZ es_multiplo ; si es 0, es porque los últimos dos bits eran 0, osea múltiplos de 4

       no_multiplo:
               POP [R7], R4
               MOV R5, R4       ; Carga el número actual en R5, para no perder su valor
               SHL R4, 3        ; Multiplica R4 por 8 (desplazamiento a la izquierda)
                SUB R4, R5       ; Resta R5 a R4
               ADD R4, R3       ; Suma 1 a R4
               JMP actualizar_numero
```

```
       es_multiplo:
              POP [R7], R4
              SHRA R4, 1        ; Realiza un desplazamiento a la derecha para verificar si
los últimos dos bits son cero
              SUB R4, R3        ; Resta 1 a R4
              SUB R4, R3        ; Resta 1 nuevamente a R4
              JMP actualizar_numero

       actualizar_numero:
              STR [R0], R4   ; Almacena el valor actualizado de R4 en el arreglo
              ADD R0, R3        ; Avanza al siguiente número en el arreglo
              JMP loop

       fin:
              RET|R7|
```

Ejercicio 5

a) Sin agregar circuitos nuevos, agregar la instrucción STRPOP que almacena en la dirección pasada como parámetro el número almacenado en la pila incrementando previamente en uno. 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
    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
```

b) Agregar la instrucción ANDOR, que toma de dos en dos los bits del registro pasado como parámetro y hace por un lado una AND entre ellos y los almacena en el nibble inferior, y también hace un OR de dos en dos y almacena los resultados en el nibble superior. 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 realice el intercambio de bits. Se recomienda utilizar como código de operación el 0x0F.

```
01111: ; ANDOR
    ALU_enA                         RB_enOut RB_selectIndexOut=0   ; A <- Rx
```

```
    RB_enIn RB_selectIndexIn=0      ALU_enOut ALU_OP=ANDOR
    reset_microOp
```