Checkpoint 1:

Introducción:

a) El tamaño de la memoria es de 256 palabras de 8 bits.
b) Como el bus tiene 8 bits el PC también tiene tamaño de 8 bits porque están conectados.
c) El IR tiene en total 16 bits (8 bits para IR-H y 8 bits para IR-L) se encuentra en el decode.
d) No se pueden por las instrucciones reservadas(respetando la arquitectura actual)
e)

Checkpoint 2:

Analizar - Estudiar el funcionamiento de los circuitos indicados y responder las siguientes preguntas:

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

b) ALU (Unidad Aritmético Lógica): ¿Qué función cumple la señal opW?
Permite escribir los flags en la unidad de control.

c) Control Unit (Unidad de control):
¿Cómo se resuelven los saltos condicionales? Describir el mecanismo.
Para resolver un salto condicional **la unidad de control habilita las señales para registros y direcciones de memoria (**PC_load y DE_enOutlmm) para que el PC sepa a qué posición de memoria saltar.

d) microOrgaSmall (Data Path):
¿Para qué sirve la señal DE_enOutImm?
Habilita la entrada al bus de un valor inmediato.

¿Qué parte del circuito indica que índice del registro a leer y escribir?
X e Y

Checkpoint 3:

a) Antes de correr el programa, identificar el comportamiento esperado.

primero hay un salto a la etiqueta seguir

luego se le asigna 0xFF a R0

luego se le asigna 0x11 a R1

luego en la etiqueta siguiente se suman los valores de R0 y R1 = 0xFF + 0x11 y se asigna ese valor a R0 y se espera que se prendan los flags de C y Z

como el flag C está prendido se va a realizar un salto a la etiqueta siguiente

en la etiqueta siguiente se vuelve a sumar al registro R0 + R1 y se actualiza el valor a R0

no se prende ningun flag

luego en el salto JC como el flag C está apagado el programa sigue la ejecución secuencial

en la etiqueta halt haces un salto JMP a la etiqueta halt y entras en un ciclo

b) ¿Qué lugar ocupará cada instrucción en la memoria? Detallar por qué valor se reemplazarán las etiquetas.

JMP seguir 0x0000

seguir: 0x0002, 0x0004

siguiente: 0x0006, 0x0008

halt: 0x00010

c) Ejecutar y controlar ¿cuántos ciclos de clock son necesarios para que este código llegue a la instrucción JMP halt?

```
JMP seguir 5 (fetch) + 2 = 7
seguir:
SET R0, 0xFF 5 (fetch) + 2 = 7
SET R1, 0x11 5 (fetch) + 2 = 7
siguiente:
ADD R0, R1 5 (fetch) + 5 = 10
JC siguiente 5 (fetch) + 3 = 8
siguiente:
ADD R0, R1 5 (fetch) + 5 = 10
JC siguiente 5 (fetch) + 2 = 8
halt:
JMP halt
```

57

d)

ADD, 5(fetch) + 5 = 10

JMP, 5(fetch) + 2 = 7

e)

32 Hz = (32 vueltas) / (1 seg) = (32 vueltas) / (1000 ms) = 32 clock

1 clock = (1000 ms) / 32 = 31.25 ms

```
JMP seguir 5 (fetch) + 2 = 7 clocks = 31.25 ms * 7

seguir:
SET R0, 0xFF 5 (fetch) + 2 = 7 clocks 31.25 ms * 7

SET R1, 0x11 5 (fetch) + 2 = 7 clocks 31.25 ms * 7

siguiente:
ADD R0, R1 5 (fetch) + 5 = 10 clocks 31.25 ms * 10

JC siguiente 5 (fetch) + 3 = 8 clocks 31.25 ms * 8

siguiente:
ADD R0, R1 5 (fetch) + 5 = 10 clocks 31.25 ms * 10

JC siguiente 5 (fetch) + 2 = 8 clocks 31.25 ms * 8

halt:
JMP halt
```

Checkpoint 4:

Analizar -
Cada componente tiene una X sobre un cable. Para cada uno de ellos indicar qué sucedería si ese cable:
a) Se corta:

- **MicroOrgaSmall**: Se corta la conexión que hay entre la ALU y la unidad de control, es decir las flags de la ALU no pueden ser leídas por la unidad de control. Por ende la unidad de control no va a poder prender ciertas señales nunca ya que no tiene las flags necesarias.
- **ALU**: En la ALU se corta la conexión que hay entre el código de operación y los multiplexores. Uno de esos multiplexores es el encargado de elegir la operación que se tiene que realizar y el otro multiplexor es el encargado de almacenar el carry o borrow en las operaciones que correspondan. Por ende si se corta ese cable no sabríamos que operación estaríamos haciendo y no podríamos obtener los resultados que queremos ni tampoco el flag de carry.
- **Unidad de control**: Afecta a la retroalimentación del Flip Flop

/* afecta a la retroalimentación del microPC por lo que dejaría de actualizarse */

- **PC**: No se podría escribir en el componente. O sea no se podría acceder a cierta posición de memoria.
- **Registros**: No se podría leer el registro elegido.
- **Memoria**: No se podría acceder a cierta posición en la memoria.
- **Decode**: No se podría guardar la posición de memoria a la que se quisiera acceder en caso de que haya un salto o algo así.

/*

STR [M], Rx
LOAD Rx, [M]
STR [Rx], Ry
LOAD Rx, [Ry]
JMP M
JC M
JZ M
JN M
INC Rx
DEC Rx
SHR Rx, t
SHL Rx, t
SET Rx, M

las operaciones str, load y set también usan valores de memoria por lo que también, ¿no se podría guardar la posición de memoria de estas operaciones?
*/

b) Vale siempre 0 (la cantidad de bits necesarios):

- **MicroOrgaSmall**: Todos los flags siempre estarian apagados y hay ciertas señales que no se prenderían nunca en ciertas condiciones.
- **ALU**: La operación siempre sería la 000 que es la suma, sin embargo los flags si se podrían prender. **/*en este caso, el flag de carry/borrow sería el de carry y funcionaría bien*/**

- **Unidad de control**: Si siempre el dato que le entra al microPC es 000 entonces siempre estaría en esa posición lo que significa que no podría leer todo el microprograma. Solo podría acceder a la primera línea.
- **PC**: No se podría acceder a cierta posición de memoria. Nunca se podría cargar un valor en el PC.
- **Registros**: Nunca podriamos saber el valor del registro.
- **Memoria**: Siempre estaríamos en la posición 0 de la memoria.
- **Decode**: La posición de memoria que siempre va tener el valor de M va a ser 0.

c) Vale siempre 1 (la cantidad de bits necesarios)

- **MicroOrgaSmall**: Todos los flags siempre estarian prendidos y se prenderían siempre las señales.
- **ALU**: La operación siempre sería la 001 que es la resta, sin embargo los flags si se podrían prender. **/*en este caso, el flag de carry/borrow sería el de borrow y funcionaría bien*/**
- **Unidad de control**: Si siempre el dato que le entra al microPC es 001 entonces siempre estaría en esa posición lo que significa que no podría leer todo el microprograma. Solo podría acceder a las dos primeras líneas.
- **PC**: Siempre se podría cargar un valor en el PC.
- **Registros**: Siempre podríamos saber el valor del registro.
- **Memoria**: Siempre estaríamos en la posición 001 de la memoria.
- **Decode**: La posición de memoria que siempre va tener el valor de M va a ser 001.

Checkpoint 5

Programar -
Escribir en ASM un programa que calcule la suma de los primeros n números naturales. El valor resultante debe guardarse en R1, y se espera que el valor de n sea leído de R0.

```
subrut:    CMP R0, 0x00
           JZ halt
           ADD R1, R0
           SUB R0, 0X01
           JMP subrut

halt:      JMP halt
```

Checkpoint 6

Ampliando la máquina - Agregar las siguientes instrucciones nuevas:

Nota 1: Los siguientes ítems deben ser presentados mediante un código de ejemplo que pruebe la funcionalidad agregada.

Nota 2: Tener en cuenta que si se agrega una operación, será necesario agregar el nombre mnemotécnico y el opcode en el archivo "**`assembler.py`**".

> Para generar un nuevo *set* de micro-instrucciones, generar un archivo `.ops` y traducirlo con el comando:
> `python buildMicroOps.py NombreDeArchivo.ops`
> Esto generará un archivo `.mem` que puede ser cargado en la memoria ROM de la unidad de control.

a) Sin agregar circuitos nuevos, agregar la instrucción `SIG`, que dado un registro aumenta su valor en 1. Esta operación no modifica los flags. Utilizar como código de operación el `0x09.`

```
01001: ; SIG
      RB_enOut  ALU_enA  RB_selectIndexOut=0
      ALU_OP=cte0x01
      ALU_enOut ALU_enB
      ALU_OP=ADD
      RB_enIn   ALU_enOut RB_selectIndexIn=0
      reset_microOp
```

b) Sin agregar circuitos nuevos, agregar la instrucción `NEG` que obtenga el inverso aditivo de un número sin modificar los flags.

Nota: el inverso aditivo de un número se puede obtener como `xor(XX, 0xFF)+0x01`.

Utilizar como código de operación el `0x0A.`

```
01010: ; NEG
      RB_enOut  ALU_enA  RB_selectIndexOut=0
      ALU_OP=cte0x0FF
      ALU_enOut ALU_enB
      ALU_OP=XOR
      RB_enIn   ALU_enOut RB_selectIndexIn=0

      RB_enOut  ALU_enA  RB_selectIndexOut=0
      ALU_OP=cte0x01
      ALU_enOut ALU_enB
      ALU_OP=ADD
      RB_enIn   ALU_enOut RB_selectIndexIn=0
      reset_microOp
```

c) Implementar un circuito que dados dos números A_{7-0} y B_{7-0} los combine de forma tal que el resultado sea $B1\ A6\ B3\ A4\ B5\ A2\ B7\ A0$. Agregar la instrucción MIX que aplique dicha operación entre dos registros, asignándole un código de operación a elección.

```
01011: ; MIX
     RB_enOut  ALU_enA  RB_selectIndexOut=0
     RB_enOut  ALU_enB  RB_selectIndexOut=1
     ALU_OP=MIX ALU_opW
     RB_enIn   ALU_enOut RB_selectIndexIn=0
     reset_microOp
```