Universidad Rey Juan Carlos

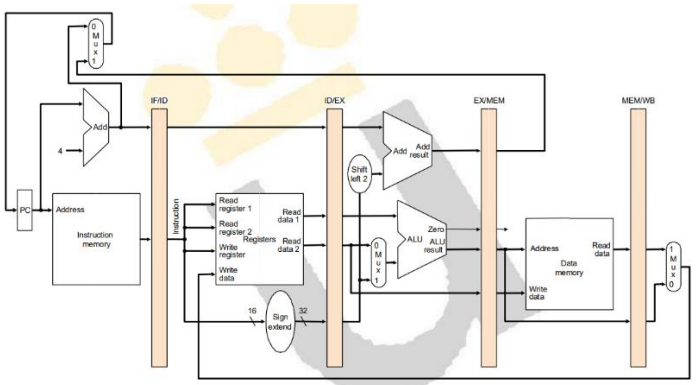
Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Grado en Ingeniería Informática

Arquitecturas Avanzadas de Computadores

Práctica 2 – Segmentación



Curso Académico 2021/2022

Milagros Mouriño Ursul

50366324G

Índice

Índice de imágenes

1. Introducción

La simulación del camino de datos correspondiente al procesador MIPS segmentado.

El objetivo de la práctica es la profundización en la técnica de implementación de la segmentación destinada al aumento de la productividad de las tareas realizadas por un procesador. Asimismo, se persigue la utilización del paradigma de la orientación a objetos para implementar los diferentes componentes del camino de datos en el lenguaje de programación Python.

La memoria consta de un apartado de propuesta en el que se explica el grueso del trabajo con el fichero de instrucciones que se ejecutan en el cauce de segmentación simulado, el diagrama UML con la jerarquía de clases correspondientes a los componentes del camino de datos, la codificación principal de la práctica y la explicación de la salida obtenida en algunas etapas de la segmentación. Por otro lado, se exponen las conclusiones extraídas y las referencias consultadas.

2. Propuesta

2.1. Instrucciones

En la figura 1, se observan las instrucciones en lenguaje ensamblador ejecutadas por la simulación del camino de datos. Se trata de un código que calcula la potencia de un número *a* elevado a *b*, cuyos valores son 3 y 2 respectivamente. Luego, computa la suma de estos tres valores y el múltiplo de cuatro más cercano a este último valor obtenido. Suponemos que en el registro $t0 se ha cargado la dirección de memoria de la variable *a* con el cual se baja en el registro $s0 este valor de la memoria de datos a través de la primera instrucción de carga de memoria. Posteriormente, se hallan otros dos instrucciones de tipo load para obtener el valor de la variable *b* (situada a 4 bytes de distancia respecto a la primera) en los registros $s1 y $t6. Se carga en dos registros, ya que se necesita conservar el valor de la *b* para el cálculo de la suma. Entre las líneas 5 y 9, se produce el cómputo de la potencia. Este se basa en un sencillo bucle que multiplica *b* veces el valor de *a* con la instrucción mul. Se decrementa el valor de *b* con subi y se comprueba la condición de salida del bucle con beq. Mediante la instrucción j, se consigue iterar en el bucle. Su fin se indica mediante la etiqueta FinBucle donde se halla una instrucción de escritura en memoria sw que almacena la potencia de los números. Más abajo se encuentra la etiqueta Suma que indica el comienzo del cómputo de la suma de *a*, *b* y la potencia obtenida. Esta se realiza mediante dos instrucciones aritméticas add que presentan riesgos de datos controlados por un cortocircuito de tipo Ex que se explicará a continuación. La instrucción sub se utiliza para almacenar una copia de la suma en $t6 de la cual se hace su correspondiente sw al final del programa. Además, hay una instrucción addi que guarda en $s2 el número cuatro que nos permite averiguar el múltiplo de este número. La instrucción rem averigua el módulo de la suma obtenida anteriormente en $t2 y el 4 que se guardó en $s2 y guarda el resultado en $t3. Si es igual a 0, se sale del bucle y se computa el múltiplo, sino se sigue restando uno con subi hasta encontrar el valor deseado. Finalmente, se ejecutan dos instrucciones sw para guardar el valor de la suma y el múltiplo a continuación de la *a*, la *b* y la potencia en memoria de datos.

2.2. Diagrama UML

La ilustración 2 refleja la jerarquía de clases para la implementación del circuito. La mayoría de las clases redefinen el método __str__ para poder ser impresos por consola y proporcionar feedback sobre el estado de los componentes en cada ciclo, implementan el método __init__ para la creación de instancias, así como métodos get y set con la anotación @property y @nombreatributo.setter respectivamente. Por un lado, se encuentra la clase ProgramCounter, la cual posee un atributo entero denominado address. Por el otro, se halla la clase Instruction de la cual heredan las clases InstructionJ, InstructionI e InstructionR correspondientes a cada uno de los tipos de instrucciones del repertorio de instrucciones MIPS. Existe un atributo de tipo protected llamado op_code referido al código de operación de cada instrucción dado que se presenta en todas las instrucciones. Por otro lado, la clase InstructionJ añade un atributo entero denominado target que indica la posición de memoria a la que se salta en las instrucciones de este tipo.

```
addi $t1, $zero, 1
lw $s0, 0($t0)
lw $s1, 4($t0)
lw $t6, 4($t0)
Potencia: beq $t6, $zero, FinBucle
mul $t1, $t1, $s0
subi $t6, $t6, 1
j Potencia
FinBucle: sw $t1, 8($t0)
Suma: add $t2, $s0, $s1
add $t2, $t2, $t1
sub $t6, $t2, $zero
addi $s2, $s2, 4
MulCuatro: rem $t3, $t2, $s2
beq $t3, $zero, Fin
subi $t2, $t2, 1
j MulCuatro
Fin: sw $t6, 12($t0)
sw $t2, 16($t0)
```

***Ilustración 1**: Fichero de instrucciones*

La clase InstructionR almacena los campos rs, rt y rd correspondientes a los dos registros fuente y el registro destino utilizados en las operaciones aritmético-lógicas. Además, InstructionI contiene un registro destino (rt), un único registro fuente (rs) y un offset de tipo int que se utiliza para realizar operaciones aritméticas con inmediatos como en las instrucciones addi o para indicar una posición de memoria en las instrucciones de tipo b. Estas dos últimas clases están asociadas a la clase Register (dado que presentan atributos de este tipo) que almacena el nombre del registro y su valor. Se ha añadido a esta clase el método __eq__ para la comprobación de riesgos de datos en la fase de decodicación. También, se diseñó la clase abstracta PipelineRegister que representa un registro de acoplamiento. Tiene un atributo protegido llamado instruction que almacenará el contenido de la instrucción que se haya procesado en una fase previa de la segmentación. A partir de la clase PipelineRegister heredan IfIdPipelineRegister, IdExPipelineRegister, ExMemPipelineRegister y MemWbPipelineRegister correspondientes a los cuatro registros que existen para el procesador MIPS. En concreto, la clase ExMemPipelineRegister contiene un valor referido a la dirección a la cual tienen que acceder las operaciones de memoria para cargar o colocar un dato, ya que este valor no se almacena en ningún registro. En cuanto a los componentes del camino de datos, se implementó la clase InstructionMemory que almacena en una lista de cadenas de texto todas las instrucciones del fichero visto en el apartado anterior en el constructor. También, posee un diccionario como atributo denominado labels que contiene como clave una etiqueta del programa y un entero como valor que representa la posición en memoria donde se encuentra la misma. Se empleó para la eficiencia a la hora de realizar búsquedas. Entre sus funciones están place_labels que rellena la información del diccionario labels, get_last_instruction_address utilizada para realizar el control del pipeline dentro del programa principal y la más importante: instruction_fetch.

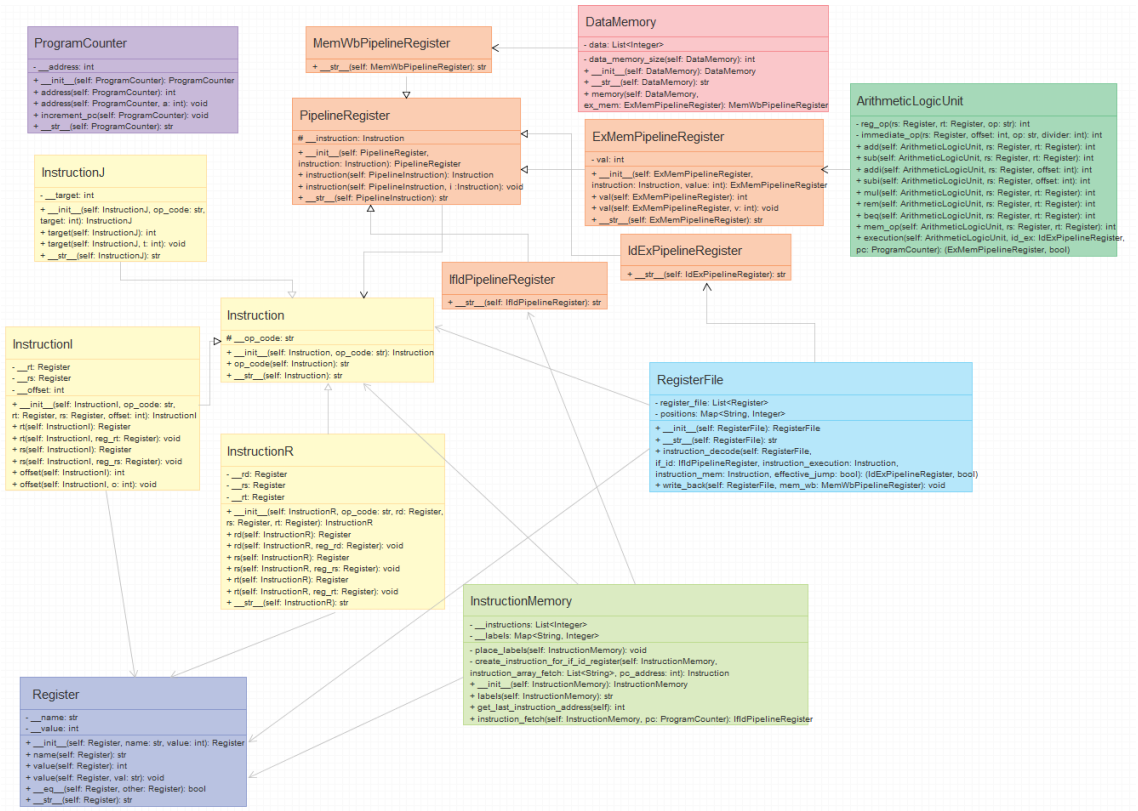


***Ilustración 2**: Diagrama UML*

Esta función construye un elemento de tipo IfIdPipelineRegister que necesita la fase de decodificación y se basa en el método create_instruction_for_id_register que, a partir del string almacenado en la lista instructions y la dirección apuntada por el pc, crea el objeto de tipo Instruction que se corresponde con el atributo de la instancia devuelta. Debido a esto, presenta relaciones de asociación con la clase Register, Instruction e IfIdPipelineRegister. Por otro lado, se visualiza la clase RegisterFile cuyos atributos son register_file, el cual es una lista de objetos de tipo Register que representa el banco de registros y un diccionario llamado positions que establece una correspondencia entre un nombre de registro y su número. Por ejemplo, relaciona el registro $zero con el 0 y el $at con el 1. Esto también permite la eficiencia de algunas operaciones. Esta clase consta de dos métodos importantes: instruction_decode y write_back correspondientes a las dos fases de la segmentación. La primera tiene como parámetros un objeto de tipo IfIdPipelineRegister resultado de la fase explicada anteriormente, dos instrucciones correspondientes al contenido de los registros de acoplamiento ex_mem y mem_wb para comprobar si se deben hacer cortocircuitos de Ex o Mem y un booleano denominado effective_jump que controla si se produce un salto tanto condicional como incondicional. Se relaciona con la clase IdExPipelineRegister, ya que el parámetro de salida de la función se corresponde a una tupla compuesta por un objeto de ese tipo y un booleano que indica si se debe insertar una burbuja en el caso de que exista un riesgo RAW entre una instrucción lw y otra. Por otro lado, el método write_back recibe un elemento de tipo MemWbPipelineRegister y realiza una escritura en el banco de registros en caso de procesarse una instrucción aritmético-lógica o lw.

La clase ArithmeticLogicUnit posee dos métodos estáticos reg_op e immediate_op reutilizados por los métodos add, sub, addi, subi, mul, rem, beq y mem_op (operación de memoria) para la realización de cálculos. Asimismo, consta de un método denominado execution que recibe como parámetro una instancia de IdExPipelineRegister extraída de la fase de decodificación y el pc, ya que en caso de que se produzca un salto se debe actualizar la dirección a la que apunta el mismo. Se relaciona con la clase ExMemPipelineRegister porque la función devuelve una tupla compuesta de un objeto de esa clase y un booleano para el control de riesgos de control. La instancia de ExMemPipelineRegister es el parámetro de entrada del método memory localizado en la clase DataMemory, el cual se refiere a la memoria de datos. Este método escribe o lee de la memoria de datos cuando se está procesando en dicha etapa una instrucción de memoria (lw o sw) y está relacionado con la clase MemWbPipelineRegister, ya que retorna un objeto de este tipo.

2.3. Explicación del código importante de la simulación del camino de datos

2.3.1. InstructionMemory

El método __init__ se encarga de inicializar la lista de instrucciones y el diccionario que contiene las etiquetas junto a la dirección de memoria donde se almacenan. Luego, se procede a abrir el fichero de instrucciones explicado en el apartado 1 y se va metiendo cada línea del mismo (tras haber quitado con la operación replace el carácter '\n') dentro de la lista instructions con el método append. Abajo, se invoca al método place_labels. En este procedimiento, se itera sobre la lista de instrucciones obteniéndose cada instrucción en la variable instruction_string. Luego, se llama al método Split que separa strings según un carácter delimitador (en este caso, el espacio) y los devuelve en un array de strings denominado instruction_array_string. Si la cadena que se halla en instruction_array_string[0] no es igual a alguno de los códigos de operaciones de las instrucciones, entonces se ha encontrado una etiqueta guardada bajo el nombre label, se quita con el método sub de la clase re, el carácter ":" y se añade una entrada al diccionario donde la clave es la etiqueta y el valor la posición en memoria de instrucciones. Por otra parte, hay dos métodos que se utilizan dentro de Pipeline.py denominados labels correspondiente al getter del diccionario de etiquetas, el cual devuelve en un string los ítems del mismo y get_last_instruction_address, que informa acerca de la posición en memoria de la última instrucción. En la ilustración 3, se visualiza el método más importante instruction_fetch encargado de la primera fase de la segmentación. Recibe como parámetro de entrada el pc, ya que se necesita saber la posición de memoria de la instrucción a la que se hará el fetch. Por otra parte, tiene un parámetro de salida de tipo IfIdPipelineRegister, dado que se tiene que almacenar la información obtenida tras la ejecución de esta fase en el correspondiente registro de acoplamiento. Si la dirección a la que apunta el pc es mayor a la de la última instrucción en memoria, entonces se devuelve None, ya que no hay ninguna otra instrucción de la cual poder hacer un fetch. En caso contrario, se realiza un split sobre el string con la instrucción a procesar y se guarda en la variable instruction_array_fetch. Es decir, si por ejemplo se tiene el string 'addi $t1, $zero, 1' con este método se obtiene ['addi', '$t1,', '$zero,', '1']. Esta variable junto con la dirección del pc se pasa como parámetro a la función create_instruction_for_if_id_register, que es la que verdaderamente realiza todo el trabajo. Este método devuelve un objeto de tipo Instruction asociado al atributo del objeto IfIdPipelineRegister de salida. En primer lugar, se obtiene el código de operación de la instrucción dentro de una variable operation_code accediendo a la posición 0 del array obtenido anteriormente. En el ejemplo anterior, operation_code es 'addi'. Se presentan cuatro ramas. En concreto, una por cada tipo de instrucción (I, J y R) y otra que contempla el caso de que encontrar una etiqueta. En la primera, se realiza la construcción de la instrucción de tipo I.

```
def instruction_fetch(self, pc):
    if pc.address > self.get_last_instruction_address():
        return None
    instruction_array_fetch = str(self.__instructions[pc.address]).split()
    instruction_fetch = self.create_instruction_for_if_id_register(instruction_array_fetch, pc.address)
    return IfIdPipelineRegister(instruction_fetch)

def create_instruction_for_if_id_register(self, instruction_array_fetch, pc_address):
    operation_code = instruction_array_fetch[0]
    if operation_code in ["lw", "sw", "beq", "addi", "subi"]:
        rt_name_register = re.sub(",", "", instruction_array_fetch[1])
        if operation_code in ["lw", "sw"]:
            offset_register = instruction_array_fetch[2]
            offset_register = offset_register.split("(")
            offset = int(offset_register[0])
            rs_name_register = offset_register[1]
            rs_name_register = re.sub("\\)", "", rs_name_register)
        elif operation_code == "beq":
            rs_name_register = re.sub(",", "", instruction_array_fetch[2])
            offset = self.__labels[instruction_array_fetch[3]]
        else:
            rs_name_register = re.sub(",", "", instruction_array_fetch[2])
            offset = int(instruction_array_fetch[3])
        return InstructionI(operation_code, Register(rt_name_register, 0), offset,
                            Register(rs_name_register, 0))
    elif operation_code in ["add", "sub", "rem", "mul"]:  # Type R
        rd_name_register = re.sub(",", "", instruction_array_fetch[1])
        rs_name_register = re.sub(",", "", instruction_array_fetch[2])
        rt_name_register = instruction_array_fetch[3]
        return InstructionR(operation_code, Register(rd_name_register, 0), Register(rs_name_register, 0),
                            Register(rt_name_register, 0))
    elif operation_code == "j":
        target = self.__labels[instruction_array_fetch[1]]
        return InstructionJ(operation_code, target)
    else:
        return self.create_instruction_for_if_id_register(instruction_array_fetch[1:], pc_address)
```

***Ilustración 3**: Método instruction_fetch*

Primeramente, se elimina una coma extra presente en el nombre del registro rt. El método split anterior separaba los campos del string por el carácter del espacio en blanco. Sin embargo, hay unas comas que separan los registros por lo que estas se deben eliminar con la operación sub de la clase re. Esta información se almacena en la variable rt_name_register común a todas las instrucciones de tipo I. Luego, aparecen tres ramificaciones que separan las instrucciones de memoria, el beq y las aritméticas con inmediato. En la primera, tenemos un array del siguiente estilo: ['lw', '$s0,', '0($t0)']. Previamente se trató el nombre del registro rt. Ahora falta por procesar el offset y el registro rs, que están juntos en la variable offset_register. Primero, se realizó un split según el carácter ‘(‘. En el ejemplo anterior, offset_register pasaría de valer '0($t0)' a ['0', '$t0)']. Se declaran dos variables para obtener el offset y el registro rs por separado. En concreto, se obtendría offset = 0 (se realiza un casting a int) y rs_name_register = '$t0)'. Falta por suprimir el paréntesis extra del nombre del registro rs que se realiza con la operación re.sub("\\)", "", rs_name_register). En cuanto a la instrucción beq, únicamente se realiza la eliminación de la última coma añadida en la variable rs_name_register y se obtiene el offset buscando dentro del diccionario la posición de memoria a la cual pertenece la etiqueta. Para las instrucciones addi y subi se realiza la misma operación para eliminar la coma en el registro rs y se hace un casting a int del offset, ya que se tienen que hacer operaciones con ese inmediato en la fase Ex. Tras haber conseguido toda la información, se hace un return de un objeto InstructionI con los campos obtenidos anteriormente. Se utiliza el constructor de la clase Register para crear los registros rt y rs inicializados a valor 0, ya que su contenido se averigua en la fase decode. En cuanto a la segunda rama, que devuelve una instancia de la clase InstructionR se realiza el mismo procesado anterior para suprimir las comas extras en los registros rd y rs, y se extrae el nombre rt sin ningún filtrado.

Por otra parte, se trata el caso de las instrucciones de tipo j. Se obtiene en una variable target la dirección de memoria a la que se salta a través del diccionario labels y se retorna la función construyendo un objeto de tipo InstructionJ. Finalmente, se considera la ocasión de que operation_code sea una etiqueta. Ante esta situación, se invoca de nuevo a la función eliminando el primer campo de la variable instruction_array_fetch para poder procesar el tipo de instrucción correspondiente.

2.3.2. RegisterFile

Esta clase consta de un método __init__ con una variable local register_names con los distintos nombres de los registros y dos atributos. Uno de ellos se denomina register_file y es una lista que representa el banco de registros. Por otro lado, se declaró un diccionario cuya clave es un nombre de registro y su valor es el número del mismo. Dentro del método constructor, se inicializan las variables y se rellena el contenido del banco de registros con registros cuyo valor es 0. Además, para cada uno de los nombres de registros se va guardando su correspondiente número en el diccionario positions. Posteriormente, se utiliza el diccionario para guardar el número del registro $t0 dentro de la variable index_reg. Y se guarda en el banco de registros para dicho registro el contenido 4. Esto se hace para evitar implementar una instrucción la (load address) que, dada una variable en memoria de datos, obtenga su correspondiente dirección en memoria. Es decir, se presupone que antes de iniciar el programa ya se ha cargado este valor en el banco de registros y se pueden ejecutar directamente las instrucciones load. A continuación, se observa el método __str__ que devuelve un string con el contenido almacenado en el banco de registros. Para ello, se utiliza un bucle for que lo recorre. Posteriormente, se muestran los dos métodos más relevantes de la clase: instruction_decode y write_back correspondientes a la lectura y escritura en el banco de registros. En la figura 4, se observa que en la primera rama del método instruction_decode contempla el caso de que el registro de acomplamiento IF/ID sea None. Esto se da en la primera iteración del programa principal, ya que no se ha leído de memoria nada previamente por lo cual se retorna None al no poder decodificarse nada. Por otro lado, la segunda ramificación comprueba si se produjo un salto en la fase de Ex. Esto pretende controlar los riesgos de control ocasionados por las instrucciones b y j. Se adopta la técnica del salto retardado que consiste en que, a pesar de que el salto se tome o no, se ejecuta la instrucción posterior. Si el salto se toma, la instrucción que está siendo decodificada debe ser eliminada, ya que realmente no se debe ejecutar en el siguiente ciclo. La instrucción que debe leerse de memoria de instrucciones y entrar en la fase IF es la indicada en el offset de las instrucciones b o el target de las j. Si no se entra en ninguna de las ramas anteriores, se procede a decodificar la instrucción del registro IF/ID. Al igual que en el método instruction_fetch de la clase InstructionMemory, hay tres sentencias if para los tres tipos de instrucciones que existen en el repertorio de instrucciones de MIPS.

```python
def instruction_decode(self, if_id, instruction_execution, instruction_mem, effective_jump):
    if if_id is None:
        return None
    elif effective_jump:
        return None
    else:
        instruction_fetch = if_id.instruction
        if isinstance(instruction_fetch, InstructionI):
            index_rt = self.positions[instruction_fetch.rt.name]
            index_rs = self.positions[instruction_fetch.rs.name]
            rt = Register(self.register_file[index_rt].name,
                          self.register_file[index_rt].value)
            rs = Register(self.register_file[index_rs].name,
                          self.register_file[index_rs].value)
            instruction_decode = InstructionI(instruction_fetch.op_code, rt, instruction_fetch.offset, rs)
        elif isinstance(instruction_fetch, InstructionR):
            index_rd = self.positions[instruction_fetch.rd.name]
            index_rt = self.positions[instruction_fetch.rt.name]
            index_rs = self.positions[instruction_fetch.rs.name]
            rd = Register(self.register_file[index_rd].name,
                          self.register_file[index_rd].value)
            rt = Register(self.register_file[index_rt].name,
                          self.register_file[index_rt].value)
            rs = Register(self.register_file[index_rs].name,
                          self.register_file[index_rs].value)
            instruction_decode = InstructionR(instruction_fetch.op_code, rd, rs, rt)
        elif isinstance(instruction_fetch, InstructionJ):
            instruction_decode = InstructionJ(instruction_fetch.op_code, instruction_fetch.target)
```

***Ilustración 4**: Primera parte del método instruction_decode*

```python
if isinstance(instruction_decode, InstructionI) or isinstance(instruction_decode, InstructionR):
    if instruction_execution is not None:
        if isinstance(instruction_execution, InstructionR):
            if instruction_execution.rd.__eq__(instruction_decode.rs):
                instruction_decode.rs.value = instruction_execution.rd.value
            elif instruction_execution.rd.__eq__(instruction_decode.rt):
                instruction_decode.rt.value = instruction_execution.rd.value
        elif instruction_execution.op_code in ["addi", "subi"]:  # Revise condition
            if instruction_execution.rt.__eq__(instruction_decode.rs):
                instruction_decode.rs.value = instruction_execution.rt.value
            elif instruction_execution.rt.__eq__(instruction_decode.rt):
                instruction_decode.rt.value = instruction_execution.rt.value
        if instruction_execution.op_code == "lw" and (instruction_execution.rt.__eq__(instruction_decode.rs)
                                                      or instruction_execution.rt.__eq__(instruction_decode.rt)):
            return None, True
    if instruction_mem is not None and instruction_mem.op_code == "lw":
        if instruction_mem.rt.__eq__(instruction_decode.rs):
            instruction_decode.rs.value = instruction_mem.rt.value
        elif instruction_mem.rt.__eq__(instruction_decode.rt):
            instruction_decode.rt.value = instruction_mem.rt.value
return IdExPipelineRegister(instruction_decode), False
```

***Ilustración 5**: Segunda parte del método instruction_decode*

En la primera ramificación, se contempla el caso de que la instrucción a decodificar sea una instrucción de tipo I. A través del diccionario positions, dado el nombre de los registros rt y rs se obtiene su correspondiente índice en el register_file. Esto permite conseguir eficientemente la información del banco de registros con la que se construye el objeto instruction_decode. En cuanto a las instrucciones de tipo R, ocurre igual salvo que además se obtiene el índice del registro rd. Respecto a las instrucciones de tipo j, se obtiene el target directamente de la variable instruction_fetch. En la figura 5, se observa el tratamiento de los riesgos de datos. Primero, se comprueba si instruction_decode es instancia de InstructionI o InstructionR, ya que en caso de ser una instrucción j no se produciría un riesgo de este estilo. Por otro lado, se comprueba que el parámetro de entrada instruction_execution no es None, ya que sino no se produce ningún riesgo de datos. Luego, se verifica si es una instrucción de tipo R. En caso de serlo, el riesgo se puede producir si el registro destino (rd) es igual a alguno de los registros fuente (rs o rt) de la instrucción que está siendo decodificada. Si se cumple alguno de los dos casos, entonces se hace el cortocircuito de Ex. Este se basa en colocar el valor del registro destino de la instrucción ejecutada (instruction_ex) en alguno de los registros fuentes de la instrucción decodificada (instruction_decode). Es análogo para las instrucciones aritmético-lógicas con inmediatos como addi y subi.

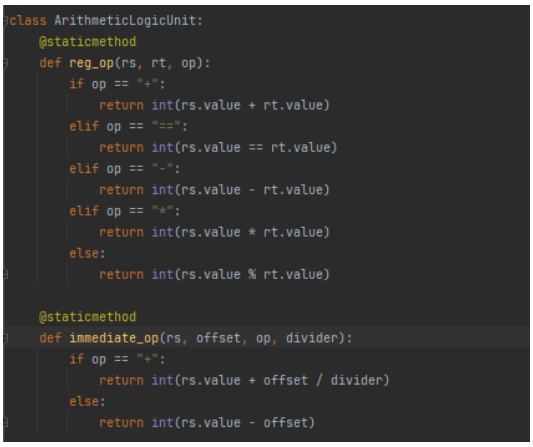
```python
def write_back(self, mem_wb):
    if mem_wb is not None:
        instruction_mem_wb = mem_wb.instruction
        operation_code = instruction_mem_wb.op_code
        index = 0
        if operation_code in ["lw", "addi", "subi"]:
            rt = instruction_mem_wb.rt
            index = self.positions[rt.name]
            self.register_file[index].value = rt.value
        elif operation_code in ["add", "sub", "mul", "rem"]:
            rd = instruction_mem_wb.rd
            index = self.positions[rd.name]
            self.register_file[index].value = rd.value
```

***Ilustración 6**: Método write_back*

El riesgo se presenta si el registro destino (rt) de instruction_ex es el mismo que el de alguno de los registros fuente (rs o rt) de instruction_decode. Se procede igual que en las instrucciones de tipo R donde se coloca el valor del registro rt de instruction_ex en alguno de los registros fuente de instruction_decode. Si la instrucción ejecutada es un lw y se cumplen las condiciones de riesgo RAW, entonces no hay cortocircuito que pueda realizarse. Se ha de insertar obligatoriamente una burbuja. El parámetro de salida es la tupla compuesta por el registro IdExPipelineRegister y un booleano. En el caso indicado, se devuelve la tupla (None, True) porque la instrucción que está siendo decodificada no debe pasar a la fase Ex, sino que tiene que volver a decodificarse, es decir, se debe insertar una burbuja (esto se controla con el booleano de la tupla). Por otro lado, se observa una ramificación que controla el riesgo de datos con cortocircuito de Mem. Primero, se verifica que instruction_mem no sea None, ya que al igual que el caso anterior no se presenta un cortocircuito. Además, se comprueba que la instrucción que ha ejecutado la fase de Mem (instruction_mem) sea de tipo lw ya que es la única que desencadena este tipo de riesgos. Se verifica si alguno de los registros fuentes de instruction_decode precisa del resultado bajado de memoria de la instrucción lw y, de ser así, se produce el cortocircuito. Se retorna el objeto (ExMemRegister(instruction_decode), False), dado que si se llega a este punto es porque la instrucción se ha podido decodificar adecuadamente y no se necesita insertar ninguna burbuja. En cuanto al método write_back de la ilustración 6, se comprueba que el parámetro mem_wb no sea None para evitar que se lancen excepciones. La variable instruction_mem_wb recoge la instrucción almacenada en mem_wb y en operation_code se obtiene el código de operación de la instrucción. Hay dos posibles ramificaciones. Si la instrucción es un lw, addi o subi, entonces se guarda en register_file el valor almacenado del registro destino rt. Por otro lado, si se trata de una instrucción de tipo R, se registra el valor del registro destino rd. Este método no tiene parámetro de salida.

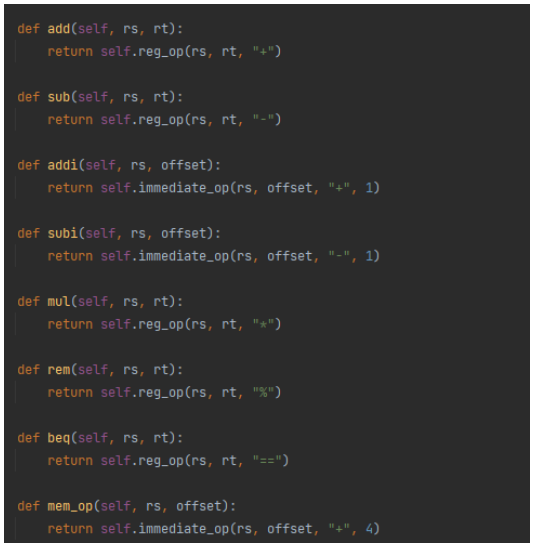
2.3.3. ArithmeticLogicUnit

En la figura 7, se observan los dos métodos estáticos reg_op y immediate_op. El primero de ellos realiza operaciones entre los registros rs y rt para las instrucciones tipo R y el beq. En cuanto al segundo, se utiliza para operaciones con instrucciones de tipo I (addi, subi, lw y sw). Tiene cuatro parámetros que se corresponden al registro fuente rs, el offset, un string op y un entero divider.



```python
class ArithmeticLogicUnit:
    @staticmethod
    def reg_op(rs, rt, op):
        if op == "+":
            return int(rs.value + rt.value)
        elif op == "==":
            return int(rs.value == rt.value)
        elif op == "-":
            return int(rs.value - rt.value)
        elif op == "*":
            return int(rs.value * rt.value)
        else:
            return int(rs.value % rt.value)

    @staticmethod
    def immediate_op(rs, offset, op, divider):
        if op == "+":
            return int(rs.value + offset / divider)
        else:
            return int(rs.value - offset)
```

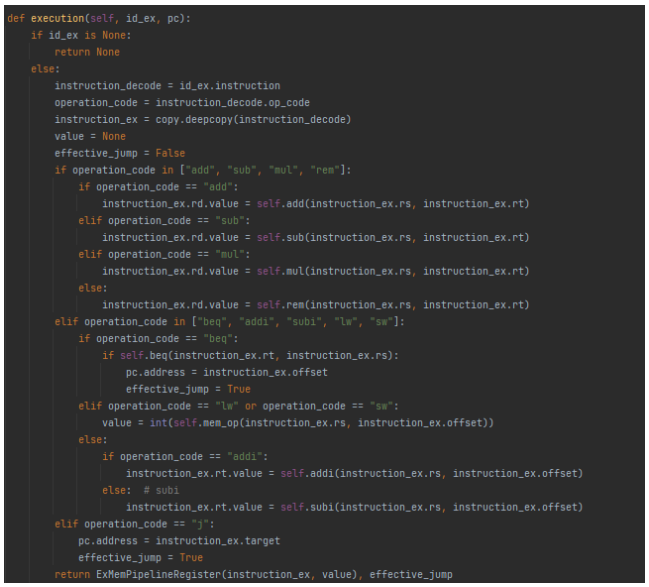
***Ilustración 7**: Métodos estáticos reg_op e immediate_op*



```python
def add(self, rs, rt):
    return self.reg_op(rs, rt, "+")

def sub(self, rs, rt):
    return self.reg_op(rs, rt, "-")

def addi(self, rs, offset):
    return self.immediate_op(rs, offset, "+", 1)

def subi(self, rs, offset):
    return self.immediate_op(rs, offset, "-", 1)

def mul(self, rs, rt):
    return self.reg_op(rs, rt, "*")

def rem(self, rs, rt):
    return self.reg_op(rs, rt, "%")

def beq(self, rs, rt):
    return self.reg_op(rs, rt, "==")

def mem_op(self, rs, offset):
    return self.immediate_op(rs, offset, "+", 4)
```

***Ilustración 8**: Métodos de la ALU*

La cadena op se utiliza para especificar el tipo de operación (suma o resta) y divider para distinguir entre instrucciones aritméticas (addi y subi) y de las de memoria (lw y sw), ya que el offset de estas últimas instrucciones indica un número de Bytes. Es decir, si se tiene como operación lw $t0, 4($s0) el offset es 4 y especifica un desplazamiento de 4 Bytes que se corresponden a una palabra. La dirección de memoria de datos es el valor indicado por el registro $s0 + 4/4 = $s0 + 1. Por otro lado, se muestran en la figura 8 las operaciones de la ALU que reutilizan el código de los métodos estáticos pasándoles los parámetros correspondientes. La ilustración 9 presenta el método execution encargada de realizar la fase Ex de la segmentación. Primero, se verifica si id_ex es None, en cuyo caso se retorna None dado que no hay instrucción que ejecutar. De lo contrario, se inicializan las variables instruction_decode (con la instrucción decodificada en la fase previa y extraída del parámetro de entrada id_ex), operation_code con el código de operación de la instrucción a ejecutar, instruction_ex que es una copia profunda de instruction_decode, value y effective_jump. Estas dos últimas son None y False respectivamente. Al igual que en las fases previas de la segmentación, hay un if por cada tipo de instrucción.



```
def execution(self, id_ex, pc):
    if id_ex is None:
        return None
    else:
        instruction_decode = id_ex.instruction
        operation_code = instruction_decode.op_code
        instruction_ex = copy.deepcopy(instruction_decode)
        value = None
        effective_jump = False
        if operation_code in ["add", "sub", "mul", "rem"]:
            if operation_code == "add":
                instruction_ex.rd.value = self.add(instruction_ex.rs, instruction_ex.rt)
            elif operation_code == "sub":
                instruction_ex.rd.value = self.sub(instruction_ex.rs, instruction_ex.rt)
            elif operation_code == "mul":
                instruction_ex.rd.value = self.mul(instruction_ex.rs, instruction_ex.rt)
            else:
                instruction_ex.rd.value = self.rem(instruction_ex.rs, instruction_ex.rt)
        elif operation_code in ["beq", "addi", "subi", "lw", "sw"]:
            if operation_code == "beq":
                if self.beq(instruction_ex.rt, instruction_ex.rs):
                    pc.address = instruction_ex.offset
                    effective_jump = True
            elif operation_code == "lw" or operation_code == "sw":
                value = int(self.mem_op(instruction_ex.rs, instruction_ex.offset))
            else:
                if operation_code == "addi":
                    instruction_ex.rt.value = self.addi(instruction_ex.rs, instruction_ex.offset)
                else:  # subi
                    instruction_ex.rt.value = self.subi(instruction_ex.rs, instruction_ex.offset)
        elif operation_code == "j":
            pc.address = instruction_ex.target
            effective_jump = True
        return ExMemPipelineRegister(instruction_ex, value), effective_jump
```

***Ilustración 9**: Método execution*

Dentro de la primera ramificación correspondiente a las instrucciones de tipo R, hay otras cuatro ramificaciones referidas a las cuatro operaciones aritméticas utilizadas en el programa: add, sub, mul y rem. El valor resultante se guarda en el valor del registro rd perteneciente a instruction_ex. La segunda rama trata las instrucciones I. Si la instrucción es un beq, se cambia la dirección del pc (razón por la cual se ha pasado como parámetro a la función) y pasa a contener el valor del offset, el cual es la dirección en memoria de la siguiente instrucción a leer. Además, el booleano effective_jump pasa a tener valor True, ya que se ha producido un salto. Si por el contrario es una operación de memoria (lw o sw), se actualiza la variable local value al valor obtenido tras la ejecución del método mem_op. Si estábamos ante una instrucción aritmética con inmediato, se llama a su correspondiente método (addi o subi) y se actualiza el valor del registro rt del instruction_ex. En cuanto a las instrucciones j, se cambia la dirección del pc a la apuntada con el target y se pone a True el booleano effective_jump. La función retorna una tupla con un objeto ExMemPipelineRegister compuesto de instruction_ex y el valor, así como la variable local effective_jump, la cual se necesita en la fase de decodificación.

2.3.4. DataMemory

Esta clase presenta un método constructor __init__ donde inicializa un array de enteros denominado data. Cabe mencionar que en la posición 4 y 5 están los valores de las variables a y b, los cuales son 3 y 2 respectivamente. Por otro lado, se ha definido un método data_memory_size para obtener el tamaño de memoria, así como el método __str__ que devuelve en un string el contenido del atributo data. Se observa el método memory dentro de la ilustración 10, que recibe un objeto de tipo ExMemPipelineRegister de entrada. Si este es None, entonces se devuelve None dado que no hay operación de memoria que pueda ejecutarse.

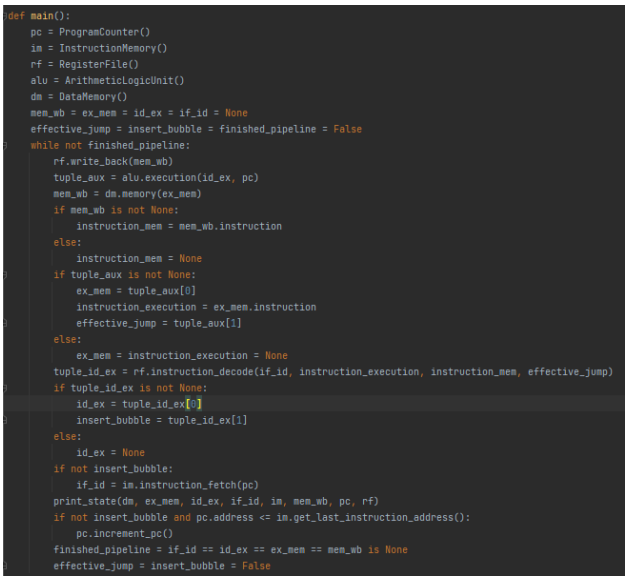
```python
def memory(self, ex_mem):
    if ex_mem is None:
        return None
    else:
        instruction_ex = ex_mem.instruction
        operation_code = instruction_ex.op_code
        instruction_mem_wb = copy.deepcopy(instruction_ex)
        if operation_code == "lw" or operation_code == "sw":
            if operation_code == "lw":
                instruction_mem_wb.rt.value = self.data[ex_mem.val % self.data_memory_size()]
            else:
                self.data[ex_mem.val % self.data_memory_size()] = instruction_mem_wb.rt.value
        return MemWbPipelineRegister(instruction_mem_wb)
```

***Ilustración 10**: Método memory*

De lo contrario, se definen variables locales: instruction_ex con la instrucción que guarda el parámetro ex_mem, operation code con el código de operación e instruction_mem_wb, que se corresponde con una copia profunda de instruction_ex. Si resulta que la instrucción ejecutada es de tipo lw o sw, entonces se lee o escribe en memoria de datos. En concreto, se accede al valor apuntado por el atributo value de ex_mem que se corresponde a la suma del registro fuente rs y el offset. Se realiza el módulo con el tamaño de la memoria para no acceder a posiciones del array incorrectas. Por último, se devuelve el objeto MemWbPipelineRegister compuesto del instruction_mem_wb.

2.3.5. Pipeline

En la ilustración 11, se muestra el método main donde se realiza toda la simulación del camino de datos. En primer lugar, se inicializan los componentes del camino de datos: el pc, la memoria de instrucciones, el banco de registros, la unidad aritmético-lógica, la memoria de datos, así como los diferentes registros de acoplamiento (mem_wb, ex_mem, id_ex, if_id). Asimismo, se declaran variables para el control (effective_jump, insert_bubble y finished_pipeline). A continuación, se observa el bucle encargado de controlar el cauce de segmentación. Mientras que no se haya acabado el proceso, se ejecutan las cinco etapas de la segmentación. Se invoca al método write_back del banco de registros pasándole el registro mem_wb. Luego, se llama al método execution cuyo resultado se almacena en la variable tuple_aux. Esta variable presenta una tupla con el valor del registro ex_mem y el booleano effective_jump, que controla los riesgos de control. Más adelante, se invoca la función memory de la memoria de datos a la cual se le pasa el ex_mem del ciclo anterior y lo devuelve en la variable mem_wb. Luego, hay un if que en función de si la variable mem_wb es None o no, guarda en instruction_mem la instrucción que contiene. Dado que ya se ha ejecutado la fase Mem, ya se puede actualizar el contenido de ex_mem. Hay una ramificación que comprueba si tuple_aux no es None. Si es así, en ex_mem se coloca el primer objeto de la tupla y en effective_jump el segundo. Además, se declara una variable instruction_execution con la instrucción almacenada en ex_mem. De lo contrario, se pone todo a None. Se invoca a la función instruction_decode cuyos parámetros son if_id, instruction_execution, instruction_mem y effective_jump. El primero es el registro de acoplamiento el cual tiene la información de la instrucción que se quiere decodificar, los otros dos (instruction_execution e instruction_mem) se pasan como parámetro para el control de riesgos de datos con cortocircuitos de Ex y Mem respectivamente.



```
def main():
    pc = ProgramCounter()
    im = InstructionMemory()
    rf = RegisterFile()
    alu = ArithmeticLogicUnit()
    dm = DataMemory()
    mem_wb = ex_mem = id_ex = if_id = None
    effective_jump = insert_bubble = finished_pipeline = False
    while not finished_pipeline:
        rf.write_back(mem_wb)
        tuple_aux = alu.execution(id_ex, pc)
        mem_wb = dm.memory(ex_mem)
        if mem_wb is not None:
            instruction_mem = mem_wb.instruction
        else:
            instruction_mem = None
        if tuple_aux is not None:
            ex_mem = tuple_aux[0]
            instruction_execution = ex_mem.instruction
            effective_jump = tuple_aux[1]
        else:
            ex_mem = instruction_execution = None
        tuple_id_ex = rf.instruction_decode(if_id, instruction_execution, instruction_mem, effective_jump)
        if tuple_id_ex is not None:
            id_ex = tuple_id_ex[0]
            insert_bubble = tuple_id_ex[1]
        else:
            id_ex = None
        if not insert_bubble:
            if_id = im.instruction_fetch(pc)
        print_state(dm, ex_mem, id_ex, if_id, im, mem_wb, pc, rf)
        if not insert_bubble and pc.address <= im.get_last_instruction_address():
            pc.increment_pc()
        finished_pipeline = if_id == id_ex == ex_mem == mem_wb is None
        effective_jump = insert_bubble = False
```

***Ilustración 11**: Función main*

Por último, la variable booleana effective_jump permite controlar los riesgos de control. La función retorna una tupla con un objeto IdExPipelineRegister y un booleano en una variable denominada tuple_id_ex. Si esta no es None, entonces id_ex pasa a ser el primer elemento de la tupla y la variable insert_bubble el de la segunda. De lo contrario, id_ex es None. Si no se ha de insertar una burbuja porque no se da el caso de que exista un riesgo RAW entre una instrucción lw que ha sido recientemente ejecutada y otra que está siendo decodificada, entonces se llama a la función instruction_fetch de la memoria de instrucciones y se actualiza el registro IF/ID. De lo contrario, se ha tenido que insertar una burbuja y no se actualiza ni el pc ni el registro IF/ID. A continuación, se invoca a la función print_state que refleja el estado de los registros en cada ciclo. Se aumenta o no el pc en función de si se inserta una burbuja y el valor del pc es inferior a la dirección de la última instrucción en memoria y se evalúa si el pipeline ha finalizado. Esto se produce cuando todos los registros son None. Se ponen a False de nuevo las variables effective_jump e insert_bubble para la siguiente iteración.

2.4. Análisis de la salida obtenida en algunas fases de la segmentación

La función print_states nos indica en cada iteración el estado de los registros de acoplamiento, el pc, la memoria de datos y el banco de registros. Asimismo, se incorpora el diccionario labels para ver la correspondencia entre las posiciones de memoria y sus etiquetas en instrucciones salto condicional e incondicional. Se analizará el output obtenido para cada uno de los riesgos de datos y control. En primer lugar, se planteó una dependencia RAW entre las instrucciones lw $t6, 4($t0) y Potencia: beq $t6, $zero, FinBucle. En la fase de Ex, se ejecuta el lw y en la decodificación del beq se deduce que existe el riesgo. Ante esto, se inserta una burbuja, es decir, el beq pasa a volver a decodificarse hasta que se haya ejecutado la fase Mem para el lw y se pueda realizar el cortocircuito de Mem. Se observa que se cumple exitosamente en las figuras 12 y 13. Por otro lado, se observa que el riesgo de control para las instrucciones de tipo j también está controlado, ya que en la figura 14 se muestra que en el registro de acoplamiento ID/EX se encuentra la instrucción j y en IF/ID una instrucción sw. Sin embargo, como el salto es efectivo en la figura 15 se observa que el sw se elimina (técnica de salto retardado) y se realiza el fetch de la instrucción a la que apunta el target de la instrucción j (se puede apreciar que el pc no se incrementa, sino que cambia bruscamente de valor). Cabe destacar que dentro del output del registro Ex/mem se observa un guión y un valor. Por ejemplo, en la ilustración 12 se observa lw $s1, 4($t0) – 5 y en la 15 j 4 – None. El valor que se sitúa a la derecha del guión se corresponde con el atributo value del registro de acoplamiento ExMemPipelineRegister que recoge el resultado obtenido en la ALU. Asimismo, es importante comentar que en las instrucciones beq y j no se guarda el string de la etiqueta, sino la posición de memoria donde se encuentra la misma en memoria de instrucciones (por ello se imprime el diccionario labels con la correspondencia entre la etiqueta y su posición en memoria). Además, se observa un salto efectivo de la instrucción beq en las figuras 16 y 17, ya que se elimina mul que se había decodificado y se carga de memoria la instrucción sw localizada con la etiqueta FinBucle. Por otra parte, se muestra una instrucción b con salto no efectivo donde se siguen ejecutando las instrucciones que vienen por detrás en las ilustraciones 18 y 19. Esto se percibe, ya que el pc únicamente se incrementa y la instrucción mul que había completado su fase de IF en el siguiente ciclo completa ID y se guarda su contenido en el registro ID/EX. En cuanto a los riesgos de datos con cortocircuito de Ex, no se observa por consola que se produce la estrategia de forwarding. Sin embargo, al finalizar el programa se muestra en la figura 20 que en memoria de datos los cinco valores esperados: variable *a* con valor 3, variable *b* con valor 2, potencia de *a* elevado a *b* con valor 9, suma de a, b y potencia con valor 14 y múltiplo de 4 más cercano 12, por lo que se deduce que se han realizado adecuadamente las correspondientes conexiones.

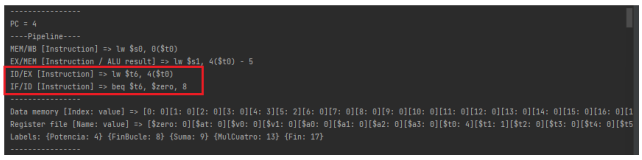


***Ilustración 12**: Riesgo de datos con inserción de burbuja y cortocircuito de Mem 1*

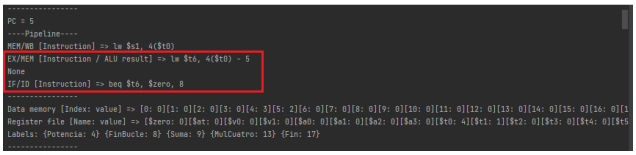


***Ilustración 13**: Riesgo de datos con inserción de burbuja y cortocircuito de Mem*

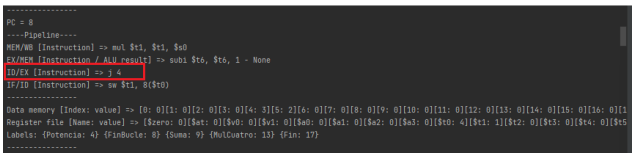


***Ilustración 14**: Riesgo de control de instrucción j 1*

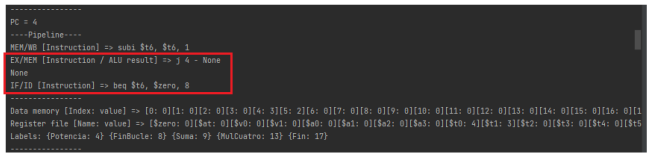


***Ilustración 15**: Riesgo de control de instrucción j 2*

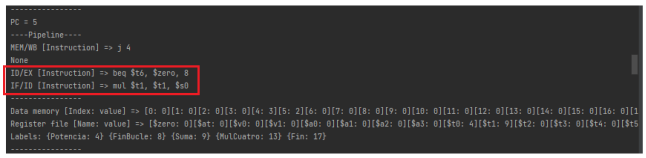


***Ilustración 16**: Riesgo de control de instrucción beq con salto efectivo 1*

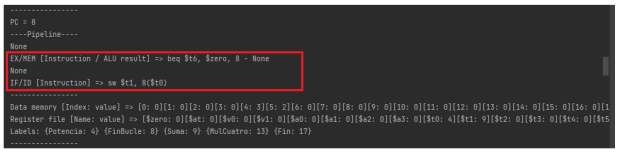


***Ilustración 17**: Riesgo de control de instrucción beq con salto efectivo 2*

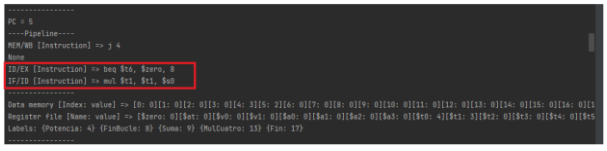


***Ilustración 18**: Riesgo de control de instrucción beq con salto no efectivo 1*

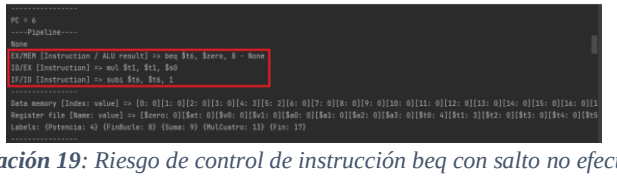


***Ilustración 19**: Riesgo de control de instrucción beq con salto no efectivo 2*

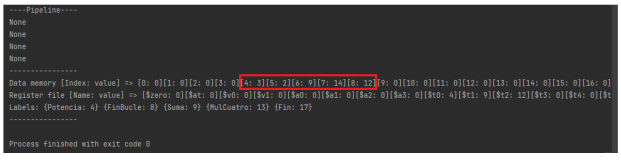


***Ilustración 20**: Memoria de datos*

3. Conclusiones

La presente práctica ha permitido profundizar en los conceptos de segmentación del camino de datos uniciclo del procesador MIPS. Se han interiorizado los diferentes riesgos de datos, así como la eficiente estrategia del forwarding para poder solventarlos. Además, el trabajo ha ayudado a aprender el paradigma de la orientación a objetos dentro del lenguaje de programación Python. Ha resultado ser una práctica de elevada complejidad, ya que se requería una constante dedicación. Se precisaba entender bien los conceptos para poder implementar la simulación del camino de datos. Asimismo, ha sido dura la elaboración de la memoria, ya que se requería la explicación y síntesis de la larga elaboración del proyecto. El trabajo terminó resultándome agradable al desarrollarse el código en el lenguaje de programación Python que no es un lenguaje tan verboso y presenta una gran flexibilidad para implementar código. De cara a las futuras entregas, me gustaría poder realizar un trabajo de calidad en menor tiempo.

4. Referencias

Apuntes de la asignatura: Tema 2 - Segmentación

Bibliografía de la asignatura: Tutorial Python 3

Google académico: Rodó, D. M. El lenguaje Python.