Práctica 2

ARQUITECTURAS AVANZADAS DE COMPUTADORES

Eduardo José Barrameda Portieles

Índice de contenidos

Índice de figuras

1. Introducción

A continuación, se van a representar las ideas que han sido plasmadas en el diseño e implementación de la práctica “Simulación de segmentación de instrucciones mediante Python”.

Comentaré también, como he resuelto problemas que han surgido durante el desarrollo de la práctica.

Destacar que hablaré de cómo he implementado el circuito con el uso de programación orientada a objetos, en primer lugar, el programa representará la cadena de montaje de la segmentación vacía, posteriormente, al leer una instrucción, el proceso no terminará hasta realizar el WB de la última instrucción de la memoria de instrucciones.

Además, se adjunta un fichero .txt con un programa en ensamblador en el que se representan riesgos, así como la implementación de la burbuja a la hora de ejecutar instrucciones que necesiten de una instrucción “lw” que aún no ha accedido a memoria.

Por último, comentaré una breve conclusión en la que remarcaré los conceptos aprendidos en clase, las dificultades generales y futuras mejoras.

2. Semejanza del circuito con el programa en Python

A continuación, explicaré las diversas clases que tiene el proyecto realizado, así como el porqué de las distintas variables que tenga la clase, métodos, objetivos de esta y su funcionamiento grosso modo.

Como podemos observar en Ilustración 1, un circuito está compuesto por varios componentes, a continuación, se explica la idea general de estos en las diversas clases que conforman el proyecto.

2.1. Instrucción

La clase instrucción define una instrucción, dicha instrucción tendrá campos para 3 registros, el tipo de instrucción, la operación, un inmediato, una etiqueta, un desplazamiento y la etapa en la que se encuentra.

En la clase se encuentra el método “decodificarInstruccion(self,instruccion)”, con dicho método podremos hacer frente al primer desafío de la práctica, decodificar la instrucción del txt.

Para lograrlo hago un Split de la cadena de caracteres, en la primera posición, generalmente tendremos “operación Rd”, en la segunda posición “Rsrc1” y en la segunda “Rsrc2”, para asignar el valor de cada variable realizamos las diferentes operaciones que correspondan, dependiendo estas de la instrucción se vaya a realizar. Destacar que en la clase se encuentran los métodos getters y setters.

En el código está explicado con más detalle.

2.2. Control

La clase Control define grosso modo la unidad de control del circuito representado en Ilustración 1.

En ella hay un constructor que inicializa la variable “señales” a un string vacío, también consta de un método “getSeñales(self,instruccion)”, dicho método devolverá una señal, señal que corresponde a la instrucción que recibe el método.

La señal tiene el siguiente formato: xxxxxxxx, 8 bits que pueden tomar los valores 0,1 y “–“ (si no la utiliza):

Señal[0]: Señal de RedDest.

Señal[1]: Representa AluSrc.

Señal[2]: Señal de MemToReg.

Señal[3]: Señal de RegWrite.

Señal[4]: Señal de MemRead.

Señal[5]: Señal de MemWrite.

Señal[6]: Señal de Branch.

Señal[7-8]: Señal de AluOp, tiene 2 bits.

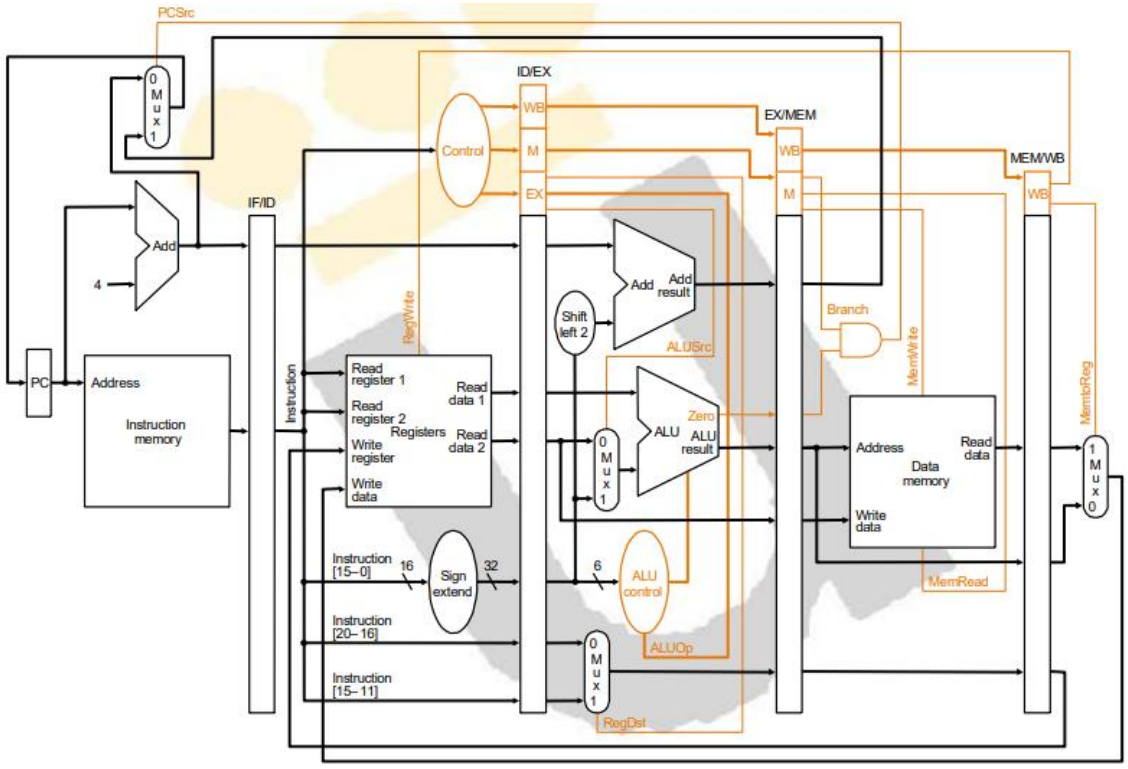

Ilustración 1: Circuito con segmentación, como podemos observar el circuito no tiene la unidad de forwarding junto a los multiplexores que controla, la comentaré más adelante.

2.3. Memoria de Instrucciones

En la clase he definido dos variables, una de ellas representará el banco de instrucciones y cada posición será a la que apunte el PC, en la segunda variable se almacenarán las etiquetas a medida que se carguen las instrucciones en el banco de instrucciones.

Primeramente, dividimos la instrucción haciendo un Split(sep=“,”), de esa manera en la primera posición tendremos la posible etiqueta, la operación y el primer registro. Si la longitud es 3, significa que hay una etiqueta que almacenar en la variable “etiquetas”, si no, se almacena la instrucción sin hacer ninguna otra operación.

En la clase también están definidos los métodos getters y setters.

2.4. Banco de Registros

El banco de registros tendrá 32 posiciones, cada una de ellas representará un registro del banco de registros del circuito de la Ilustración 1.

En la clase se encuentran los getters y setters.

2.5. Memoria de datos

En la memoria de datos implementada hay 32 posiciones, representando cada una 1 palabra de 32 bits, de modo que los desplazamientos implementados son sólo múltiplos de 4, es por eso por lo que los desplazamientos que se podrán utilizar serán sólo de 0, 4, 8 etc.

A la hora de instanciar la memoria de datos se inicializarán todas las posiciones a 0.

2.6. Unidad de control de la ALU

La clase tiene una variable, dicha variable representa la operación a realizar, estando sólo implementadas las instrucciones en las que se realizan sumas y restas, siendo muy simple añadir alguna más.

He considerado que para realizar una simple simulación no es necesario más, la simulación sería muy parecida.

El método que devuelve la operación a realizar “getOperacion”, recibe una instrucción y la señal de AluOp, como comentamos antes, serán los últimos 2 bits de las señales de la unidad Control.

Como podemos observar en Ilustración 2, sabiendo la operación de la instrucción a realizar y los bits de AluOp bastará para saber que operación realizar.

2.7. Unidad aritmética lógica(ALU)

He de destacar que tiene un método, el método “operar(a,b,operacion)”, dicho método devuelve el resultado de la operación y el valor de la señal “zero”.

En ella se realizan las diversas operaciones, generalmente sumas y restas, en la clase se ha definido junto a 5 variables:

Resultado: Representa el resultado de la operación.

Zero: Señal que se activa cuando Resultado==0, es para la operación instrucción beq.

A: Representará el primer valor que se pasa al método operar de la clase

B: Representará el segundo valor que se pasa al método operar de la clase

Operación: Representa la operación a realizar, es el valor que devuelve la unidad de control de la ALU.

2.8. CPU

Es la clase, en mi opinión, más importante, definirá el funcionamiento del circuito, se encargará de manejar el funcionamiento de todos los pasos en la segmentación, será la clase que “controle la orquesta” de los distintos elementos del circuito de Ilustración 1.

En primer lugar, comentaré las variables que la forman, estas se inicializan en el constructor de la clase, posteriormente explicaré los métodos de la clase.

El constructor recibe el nombre del .txt con el programa en ensamblador e inicializa la memoria de instrucciones.

También inicializa el banco de registros, la memoria de datos(en la posición 0 el valor 10), la ALU, la unidad de control de la ALU, la unidad de control, la variable “ciclo”(nos dirá en que ciclo se encuentra la CPU), la variable PC y los registros de acoplamiento.

A continuación, comentaré cada registro de acoplamiento, destacar que excepto el primer registro, son listas cuyas posiciones tienen datos que son de interés:

Control de la ALU

op	funct	ALUop	ALUctr
100011 (lw)	XXXXXX	00	010
101011 (sw)		00	010
000100 (beq)		01	110
000000 (tipo-R)	100000 (add)	11	010
	100010 (sub)	11	110
	100100 (and)	11	000
	100101 (or)	11	001
	101010 (slt)	11	111

Ilustración 2: Representación de las señales en la unidad de control de la ALU, como podemos observar las instrucciones de tipo R coinciden con el valor de los bits 11, en el resto es un valor independiente para cada instrucción

RegistroD1: Es un string que representa la instrucción obtenida de la memoria de instrucciones.

RegistroD2: Representa el registro de acoplamiento ID/EX, es una lista cuyo contenido en cada posición está detallado en el código.

RegistroD3: Representa el registro EX/MEM, es una lista cuyo contenido en cada posición está detallado en el código.

RegistroD4: Representa el registro MEM/WB, es una lista cuyo contenido en cada posición está detallado en el código.

Por último, hay que comentar que se inicializa “señalBurbuja”(indicará cuando hay que realizar una burbuja), “señalAux”(esta señal me ayuda a controlar que no se ejecuten instrucciones en etapas erróneas a la hora de realizar una burbuja), señal “ciclosPrograma”(avisa a la CPU cuando termine el WB de la última instrucción).

A continuación, pasaré a comentar los distintos métodos de la clase.

2.8.1. Método IF

Este método devuelve la instrucción leída de la memoria de instrucciones, en el código está explicado de manera más detallada. Representa la fase Instruction Fetch.

2.8.2. ID

Este método devuelve un tipo instrucción ya decodificada junto a los valores de los registros, desplazamiento, valor inmediato, la etiqueta y las señales de la unidad de control, el funcionamiento es bastante simple, está detallado en el código.

Representa la fase Instruction Decode.

2.8.3. EX

El método EX simulará la ejecución de la etapa EX, devolverá el valor de la ALU y el valor de la señal “zero”(véase Ilustración 3).

```
def EX(self, instruccionD, srC1, srC2, inmediato, desplazamiento, alu, señalesControl,
       pc):  # primero obtenemos la operación a realizar con aluControl y por último la reealizamos

    print("Realizando fase EX de instrucción %d" % pc)
    if señalesControl[
        1] == "0":  # si AluSrc vale 0 entonces hacemos operación de alu con el valor de los dos registros
        operacion = self.aluControl.getOperacion(señalesControl[7] + señalesControl[8], instruccionD)
        resultado, cero = alu.operar(srC1, srC2, operacion)
        print("Operacion a realizar es: " + str(srC1) + operacion + str(srC2))
        print("El resultado es: %d" % resultado)
        print(" AluSrc=" + señalesControl[1])
```

Ilustración 3: Muestra el funcionamiento del método EX, comentar que los otros casos del método funcionan de manera parecida

```
def MEM(self, memoriaDatos, direccion, dato, señalesControl,
        PC):  # si mem write es 1 escribiremos el dato en la direccion calculada por la Alu, en cas

    print("Realizando fase MEM de instruccion %d" % PC)

    if señalesControl[4] + señalesControl[5] == "00":
        print("La instruccion %d" % PC + " No tiene fase MEM")
        print("fase MEM terminada de instruccion %d" % PC)
        return ""
    elif señalesControl[4] == "1":
        print(" MemRead= 1 y MemWrite= 0")
        dato = memoriaDatos.getDato(direccion)
        print("el dato obtenido de la dirección " + str(direccion) + " de memoria es: %d" % dato)
        print("fase MEM terminada de instruccion %d" % PC)
```

Ilustración 4: Muestra del funcionamiento del método MEM, esta imagen muestra un funcionamiento resumido, en el código se explica más en profundidad

El funcionamiento es simple, en primer lugar, necesitamos saber la operación a realizar, para ello, utilizamos el método que comentamos anteriormente “getOperacion” del objeto de tipo aluControl.

Posteriormente se realiza la operación haciendo uso del método “operar” del objeto de tipo ALU.

Por último, imprimimos por pantalla los resultados, recalcar que los otros casos son bastante parecidos.

2.8.4. MEM

El método realizará lo mismo que la fase MEM de la segmentación, recibirá por parámetros la memoria de datos, la dirección en la que se va a escribir, el dato a escribir y las señales de control de la instrucción(véase Ilustración 4).

Dependiendo de la señal de control el método devolverá o no un dato leído, escribirá en la memoria o directamente no realizará nada, esto es debido a que no todas las instrucciones en ensamblador usan la memoria de datos.

2.8.5. WB

El método simulará el funcionamiento de la fase “write back” de la segmentación. Recibirá el dato leído por la memoria, el resultado de la ALU, el banco de registros y el PC, el pc lo recibe para poder imprimir por pantalla la instrucción que se está realizando etc., es a modo de claridad en la salida por pantalla.

```
def WB(self, datoMem, resultado, bancoRegistro, señalesControl, instruccionD, PC):
    print("Realizando fase WB de instruccion %d" % PC)

    if señalesControl[2] == "1" and señalesControl[3] == "1":
        print(" MemToReg= 1 y RegWrite= 1")
        if señalesControl[0] == "1":
            print(" RegDest= 1")
            print("Escribiendo dato de la memoria: " + str(datoMem) + " en registro " + instruccionD.getRd())
            bancoRegistro.setRegistro(instruccionD.getRd(), datoMem)

        elif señalesControl[0] == "0":
            print(" RegDest= 0")
            print(
                "Escribiendo dato de la memoria %d " % datoMem + " en registro " + instruccionD.getRd())  # en te
            bancoRegistro.setRegistro(instruccionD.getRd(), datoMem)
```

Ilustración 5:Muestra del funcionamiento del método WB, se muestra un resumen de su funcionamiento, con dicho método se simulará la fase "WB"

Dependiendo de la señal de control el método escribirá un inmediato en el registro destino, el valor obtenido en la memoria de datos, el valor calculado en la ALU o directamente no hará nada, esto es debido a que no todas las instrucciones tienen fase “write back”. (Véase Ilustración 5).

2.8.6. ForwardingUnit

El método simulará de una manera aproximada la ”unidad de forwarding” del camino segmentado.

En mi implementación se le pasa el registro2(registro ID/EX), el registro3(registro EX/MEM) y el registro4(registro MEM/WB), al pasarle los registros podré detectar cuando hay riesgo de datos(véase Ilustración 6) e implementar los cortocircuitos, además realizaré el cambio de los valores de los registros de acoplamiento.

Vamos a distinguir una serie de casos, enumerados a continuación:

1. Cuando la instrucción que se está ejecutando es distinta a una instrucción “lw”, “li” y “sw”, en esta parte del condicional comprobaremos si los registros Src1 o Src2 coinciden con el Rd de la instrucción almacenada en el registro de acoplamiento MEM/WB, si la instrucción con la que coincide es
   de tipo “lw”, se actualiza el valor leído por la memoria, si es una instrucción “li” se actualiza por el valor inmediato y si no se actualiza con el valor de la ALU almacenado en el registro de acoplamiento.
   Si los registros Src1 o Src2 no coinciden con el Rd de la instrucción almacenada en el registro de acoplamiento MEM/WB, se pasa a realizar un proceso parecido, pero con el registro de acoplamiento EX/MEM. La principal diferencia es la aparición de la señal “burbuja”, que es devuelta si la instrucción con la que coincide es una “lw”, realizando una burbuja en los casos en los que sea necesario.
2. Cuando la instrucción a ejecutar es de tipo sw: Este caso debería sobrar, la cuestión es que realicé de manera equivocada la decodificación de la instrucción “sw”, confundiendo el registro de destino con el registro src1, un fallo que no cometeré en un futuro. El funcionamiento es muy parecido al del caso anterior.

2.8.7 EjecutarInstrucciones

Dividiré el funcionamiento en 5 partes que comentaré de manera enumerada a continuación, antes de nada, destacar que una vez comienza la ejecución del método,

```
def forwardingUnit(self, registro2, registro3, registro4):
    if registro2[5].getOperacion() != "sw" and registro2[5].getOperacion() != "lw" and registro2[
        5].getOperacion() != "li":
        if registro2[5].getSrc1() == registro4[3].getRd() and registro2[5].getSrc1() != "zero":
            if registro4[3].getOperacion() == "lw":
                registro2[0] = registro4[0]
                print("Se actualiza valor de src1 con valor de memoria de instruccion LW")
            elif registro4[3].getOperacion() == "li":
                registro2[0] = registro4[3].getInmediato()
                print("Se actualiza valor de src1 con valor inmediato de li")
            else:
                registro2[0] = registro4[1]
                print("Se actualiza valor de src1 con valor en registro de acoplamiento MEM/WB")

        if registro2[5].getSrc2() == registro4[3].getRd() and registro2[
            5].getSrc2() != "zero":  # and registro4[3].getOperacion()=="sw":
```

Ilustración 6: Funcionamiento de la unidad de forwarding, es un fragmento del código, en él, se puede observar la principal funcionalidad del método

éste no termina hasta la realización de la fase WB de la última instrucción.

Para una correcta implementación de la simulación, las fases a realizar en orden son: WB, MEM, EX, ID e IF, esto es debido a la manera en la que está implementada la máquina segmentada. En un principio se ejecuta la parte 5, posteriormente la parte 4, después la 3, la 2 y la 1, de esa manera las siguientes instrucciones podrán ejecutarse en el orden que corresponde con el modelo segmentado de uniciclo.

1. Parte 1(WB): Para comprobar si a la instrucción le corresponde esa parte comprobamos la etapa en la que se encuentra, si es así, realizamos la ejecución del método WB(con los atributos que correspondan del registro MEM/WB), además comprobaremos si es la ejecución del WB de la última instrucción a ejecutar, si es así, termina la ejecución del método.
2. Parte 2(MEM): En primer lugar, comprobaremos si la fase que le corresponde es la fase MEM, si es así, ejecutamos el método MEM(con los atributos que correspondan del registro EX/MEM), además, actualizaremos los valores del registro de acople MEM/WB. Si ha habido una burbuja actualizo el valor del PC, de esa manera se “repiten” las fases de las instrucciones donde se realizó la burbuja, también vacío el registro EX/MEM
3. Parte 3(EX): En primer lugar, comprobaremos si la fase que le corresponde es la fase EX, si es así, ejecutamos el método EX(con los atributos que correspondan del registro ID/EX), además, actualizaremos los valores del registro EX/MEM. Por último, realizaremos las comprobaciones necesarias de saltos.
4. Parte 4(ID): En primer lugar, comprobaremos si la fase que le corresponde es la fase ID, si es así, ejecutamos el método ID (con los atributos que correspondan del registro IF/ID), además haremos uso del método “forwardingUnit”, este método nos devolverá los registros ya actualizados con los nuevos valores(por causa del uso de un cortocircuito, por ejemplo) y la señal “burbuja”, que nos ayudará a saber cuándo hay una burbuja.
5. Parte 5(IF): Si ninguna fase corresponde a las otras 4 entonces entra en este condicional, ejecutamos el método IF (con la dirección del PC).

4. Formato de la entrada

En este apartado comentaré el formato que deberá tener la entrada del programa para que no ocurra ningún error en tiempo de ejecución. Enumeraré varias cosas para tener en cuenta:

1. Se deberán usar registros con el formato: $(letra)(número), por ejemplo, $t1 o $s1
2. Los desplazamientos deberán ser múltiplos de 4, siendo desplazamiento 4 un desplazamiento de 1 palabra.
3. A la hora de poner una etiqueta el formato deberá ser el siguiente: “etiqueta: instrucción”, nótese que hay un espacio entre los dos puntos y la instrucción.
4. A la hora de realizar un salto se pondrá: “j etiqueta”
5. A las instrucciones sw y lw habrá que ponerles siempre un desplazamiento(puede ser cero): lw $s1,0($s2), por ejemplo.
6. Las instrucciones implementadas son: addi, sub, lw, li, sw, j, beq.

NOTA: El fichero txt entregado en la primera entrega ha sido modificado, tenía fallos en el formato y le he añadido algún caso peculiar.

5. Conclusión

La programación del cauce segmentado de datos en un principio parece ser bastante compleja, esto es, en mi opinión, debido a que es difícil abstraer todos los elementos del circuito, más que abstraerlos, abstraer el funcionamiento del circuito.

El principal problema que me encontré fue como hacer que se ejecutasen todas las fases de una instrucción en concreto nada más entrar al primer registro de acople y que el programa no terminase de ejecutar hasta realizar la fase write back de la última instrucción, claro está, sin fallos. Ahora lo veo más sencillo, es cuestión de dedicarle tiempo.

Aún así, hay muchas cosas que me gustaría cambiar, el fallo en la decodificación de la instrucción sw y añadir la posibilidad de un cortocircuito en instrucciones sw cuando necesita de una lw son un ejemplo. Refactorizar el código está en una de mis futuras tareas, así como realizar una implementación más orientada a objetos, tomando como ejemplo el uso de variables para los registros de acoplamiento, pudiendo usar objetos definidos por alguna clase de tipo registro de acoplamiento.