Trabajo práctico #2: datapath

Santiago Fernandez, *Padrón Nro. 94.489*
fernandezsantid@gmail.com

Pablo Rodrigo Ciruzzi, *Padrón Nro. 95.748*
p.ciruzzi@hotmail.com

Horacio Martinez, *Padrón Nro. 94.926*
hmk142@hotmail.com

2do. Cuatrimestre de 2015
66.20 Organización de Computadoras – Práctica Jueves
Facultad de Ingeniería, Universidad de Buenos Aires

26 de Noviembre, 2015

Resumen

En este trabajo práctico se verán modificaciones a distintos datapaths de una arquitectura MIPS, con el fin de agregar algunas instrucciones que no han sido implementadas en el mismo, y así poder familiarizarse con dicho concepto. La herramienta utilizada fue el DrMIPS version 2.0 [1][2].

Índice

1. Introducción

DrMIPS nos permite modificar el datapath de la arquietectura MIPS32 y el conjunto de instrucciones. Lo usaremos para agregar las siguientes instrucciones: tres de ellas (*sll*, *srl* y *jr*) al datapath monociclo y las otras dos (*j* y *blt*) al datapath pipeline.

Para lograrlo, agregaremos nuevos componentes al datapath y modificaremos el set de instrucciones, según sea necesario.

2. Desarrollo

Para el desarrollo del TP se utilizaron 2 *datapaths* distintos como base: uno monociclo y otro multiciclo, los cuales se muestran en las figuras 1 y 2 respectivamente. Cada uno de ellos se usó con su respectivo set de instrucciones. Ambas cosas eran simplemente modificables mediante un archivo JSON.

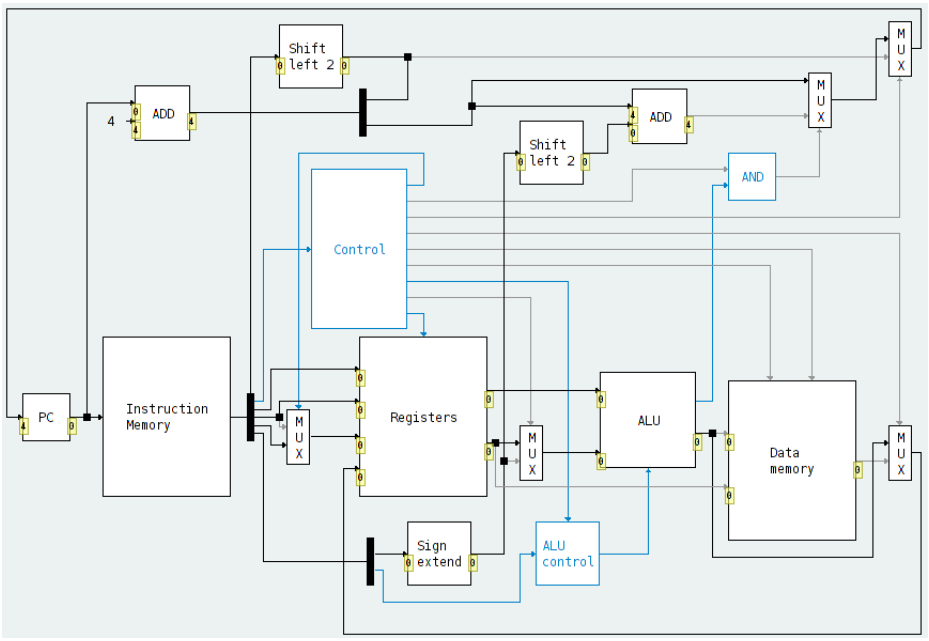

Figura 1: Datapath monociclo.

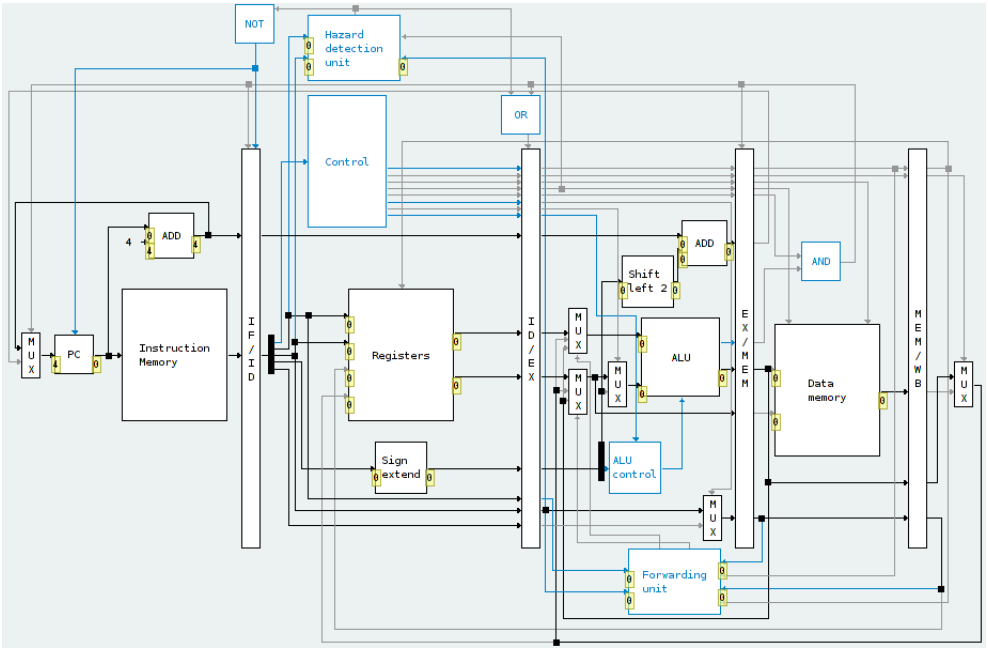

Figura 2: Datapath pipeline.

2.1. Punto 1

En este ítem, se agregaron las instrucciones *sll* y *srl* al datapath monociclo. Para ello, no fue necesario modificar el datapath, simplemente bastó con agregar las instrucciones al set de instrucciones.

2.1.1. Modificación al set de instrucciones

Se agregaron las siguientes líneas al campo *instructions* del archivo *default.set*:

```
"sll": {
        "type": "R", "args": ["reg", "reg", "reg"],
        "fields": {"op": 0, "rs": "#2", "rt": "#3",
                   "rd": "#1", "shamt": "#3", "func": 0},
        "desc": "$t1 = $t2 << $t3 = $t2 * 2^$t3"
},
"srl": {
        "type": "R", "args": ["reg", "reg", "reg"],
        "fields": {"op": 0, "rs": "#2", "rt": "#3",
                   "rd": "#1", "shamt": "#3", "func": 2},
        "desc": "$t1 = $t2 >> $t3 = $t2 / 2^$t3"
},
```

Lo que esto hace, es definir dos instrucciones nuevas del tipo R, que reciben como argumento tres registros. Luego, el campo *func*, junto con el *aluop*, será el que referencie a estas instrucciones en la sección de control de la ALU, por lo tanto, al ejecutarlas, la entrada de la ALU será la especificada. Por último,

debemos asociar esta entrada con la operación a realizarse, esto lo hacemos en la configuración de la ALU de la siguiente manera:

```
"alu": {
    ...
    "control": [
        ...
        {"aluop": 2, "func": 0, "out": {"Operation": 8}},
        {"aluop": 2, "func": 2, "out": {"Operation": 9}},
        ...
    ],
    "operations": {
        ...
        "8": "sll",
        "9": "srl",
        ...
    }
}
```

Esto le indica a la sección de control de la ALU, que ante el *aluop*=2 y *func*=0, debe ejecutar la operación 8, en este caso definida como *sll*, lo cual ocurre análogamente con *srl*.

2.1.2. Pruebas de sll y srl

Estas fueron las pruebas que se corrieron para verificar el correcto funcionamiento de las nuevas instrucciones:

Prueba 1:

```
li $t1,2
li $t2,6
sll $t3,$t1,$t2
li $t2,3
srl $t3,$t3,$t2
# 2 << 6 = 128, luego 128 >> 3 = 16
```

Prueba 2:

```
li $t1,3
li $t2,4
sll $t3,$t1,$t2
li $t2,5
srl $t3,$t3,$t2
# 3 << 4 = 48, luego 48 >> 5 = 1
```

Prueba 3:

```
li $t1,9
li $t2,1
srl $t3,$t1,$t2
```

```
li $t2,0
sll $t3,$t3,$t2
# 9 >> 1 = 4, luego 4 << 0 = 4
```

Prueba 4:

```
li $t1,16385
li $t2,18
sll $t3,$t1,$t2
li $t2,19
srl $t3,$t3,$t2
# 16385 << 17 = 262144, luego 262144 >> 19 = 0
```

Prueba 5:

```
li $t1,32767
li $t2,17
sll $t3,$t1,$t2
li $t2,17
srl $t3,$t3,$t2
# 16385 << 17 = -131072 (Recordar complemento a la base),
# luego -131072 >> 17 = 16385 (No hace extension de signo)
```

2.2. Punto 2

En este punto se pedía implementar la instrucción *j* (jump) en el datapath del pipeline. La principal diferencia de esta instrucción con el *branch* es que en este caso el salto es relativo a la posición actual, mientras que el de la instrucción *jump* es absoluto.

Para llevar a cabo dicha instrucción, lo que se quiso implementar fue algo similar a lo que ocurre entre el datapath monociclo original y el datapath monociclo sin jump (ver figura 3).

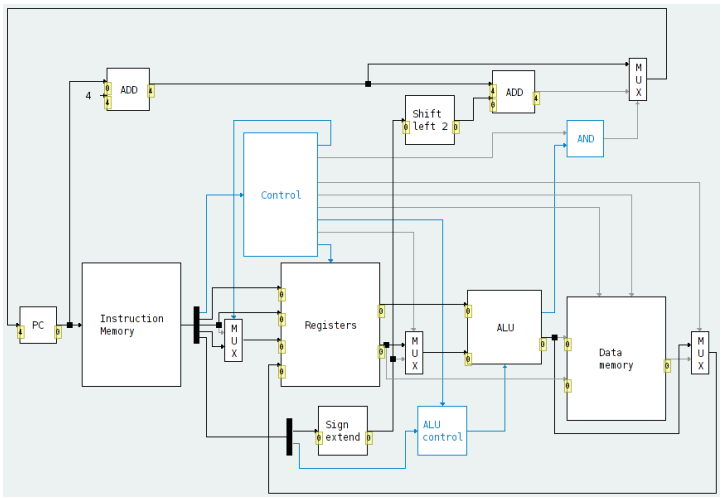

Figura 3: Datapath monociclo sin instrucción jump.

Podemos notar en contraste con la figura 1 que hay un multiplexor y un *"shifter"* a izquierda de más. Siguiendo esta idea, pero un poco más complejo por ser multiciclo, podemos ver la diferencia entre la figura 2 y 4, donde se aplica este mismo concepto para la resolución de este punto.

2.2.1. Modificación al set de instrucciones

Para llevar a cabo este punto, primero que nada se agregó tanto el tipo de instrucción J, así como también la instrucción propiamente dicha al set de instrucciones *default-no-jump.set*.

```
"types": {
    ...
    "J": [{"id": "op", "size": 6}, {"id": "target", "size": 26}]
},
"instructions": {
    ...
    "j": { "type": "J", "args": ["target"], "fields": {"op": 2,
           "target": "#1"}, "desc": "PC = target"
         },
    ...
}
```

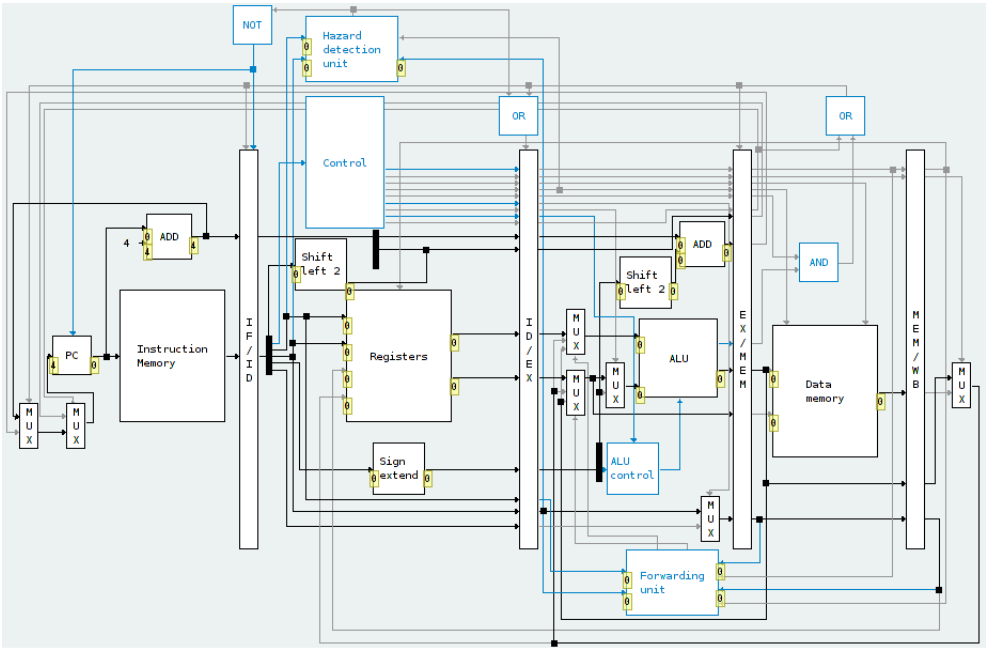

Figura 4: Datapath pipeline con instrucción *jump*.

Por otro lado, se agregó una salida de *jump* en la unidad de Control, que indique si la instrucción es de jump:

```
"control": {
    ...
    "2": {"Jump": 1},
    ...
}
```

2.2.2. Modificación al datapath

Por el lado del datapath propiamente dicho, lo que se hizo fue agregar un multiplexor (Que está justo debajo del *PC*) que decida entre *PC+4/branch* y *Jump*. Además, como ya se dijo, éste elige en base a la línea *jump* de control que se agregó. Por otro lado, se agregó el *"shifter"* a izquierda, tal como se había mencionado anteriormente.

En un principio se había llevado a cabo que tanto el cable de *target* como la línea de control del *jump* siguieran guardándose en los registros intermedios del pipeline hasta luego del *EX/MEM*. Luego, nos percatamos de que esto podía no ser así, ya que si al decodificarla ya se daba cuenta de que era un *jump*, al ser completamente incondicional, se podía enviar directamente desde esta etapa, evitando así que se ejecuten 2 operaciones innecesarias luego del *jump*. Esta "mejora" fue llevada a cabo, pero surgió un detalle: en una situación particular, que se ve plasmada en algunas de las pruebas realizadas (La 3 y la 6), donde el código contenía un *jump* con una instrucción de por medio luego de un *branch*, había problemas (Se ejecutaban instrucciones que no debían). Es por ello que se descartó esta "mejora", ya que no era del todo confiable.

Más allá de todo, había algo que no estaba solucionado, lo cual representaba un ***hazard*** de control en ambas situaciones: las 3 instrucciones siguientes al *jump* se seguían ejecutando (Para disminuir esta cantidad a una instrucción es que se había planteado la "mejora"). Para evitar dicho comportamiento, se hizo uso de la línea *jump* de la unidad de Control para que con ella se haga un *flush* del contenido de los registros del pipeline. Allí es donde se agregó la compuerta *or* que se encuentra arriba a la derecha, donde al haber un *jump* o un *branch taken*, se hace un *flush* de los registros del pipeline.

2.2.3. Pruebas de jump

Estas fueron las pruebas que se corrieron para verificar el correcto funcionamiento de la nueva instrucción:

Prueba 1:

```
    li $t1,1
    j hola
    li $t2,2
    li $t5,5
    li $t6,6
hola:
    li $t3,3
    li $t4,4
    # al finalizar deberian quedar
    # - t1=1, t3=3, t4=4
    # - t2, t5 y t6 no deberian ser modificados
```

Prueba 2:

```
    li $t0, 1
inicio:
    beq $t0, $0, fin
    li $t1, 1
    li $t2, 2
    li $t3, 3
    subi $t0, $t0, 1
    j inicio
    li $t4, 4
    li $t5, 5
fin:
    li $t6, 6
    # al finalizar deberian quedar
    # - t0=0, t1=1, t2=2, t3=3, t6=6
    # - t4 y t5 no deberian ser modificados
```

Prueba 3:

```
    li $t1, 1
    beq $0, $0, fin
```

```
    li $t5, 5
    j fin2
    li $t2, 2
fin:
    li $t3, 3
fin2:
    li $t4, 4
    # al finalizar deberian quedar
    # - t1=1, t3=3, t4=4
    # - t2 y t5 no deberian ser modificados
```

Prueba 4:

```
    li $t1, 1
    beq $0, $0, fin
    j fin2
    li $t2, 2
fin:
    li $t3, 3
fin2:
    li $t4, 4
    # al finalizar deberian quedar
    # - t1=1, t3=3, t4=4
    # - t2 no deberia ser modificado
```

Prueba 5:

```
    li $t1, 1
    beq $0, $0, fin2
    j fin
    li $t2, 2
fin:
    li $t3, 3
fin2:
    li $t4, 4
    # al finalizar deberian quedar
    # - t1=1, t4=4
    # - t2 y t3 no deberian ser modificados
```

Prueba 6:

```
    li $t1, 1
    beq $0, $0, fin2
    li $t5, 5
    j fin
    li $t2, 2
fin:
    li $t3, 3
fin2:
    li $t4, 4
```

```
    # al finalizar deberian quedar
    # - t1=1, t4=4
    # - t2, t3 y t5 no deberian ser modificados
```

2.3. Punto 3

En este punto se implementó la instrucción *jr* en el datapath monociclo; para ello, se modificó el datapath y el set de instrucciones:

2.3.1. Modificación al set de instrucciones

Se agregó un nuevo tipo de instrucción, *JR*, con la siguiente configuración (Aunque se podría haber utilizado el tipo "R" para implementarla):

```
"types": {
    ...
    "JR": [ {"id": "op","size": 6},{"id": "rs","size": 5},
            {"id": "ceros", "size": 21} ]
}
```

Los 6 bits más significativos corresponden al opcode, los siguientes 5 al registro que contiene la dirección a la cual se saltará y el resto, al no ser necesarios, son llenados con 0s.

Luego, se agrega la instrucción *jr* en particular, siguiendo el formato previamente descripto. Se elige un opcode igual a 1 simplemente porque este numero no es utilizado en ninguna de las otras instrucciones, pero podría haberse elegido cualquier otro numero de 6 bits que no sea utilizado.

```
"instructions": {
    ...
    "jr": { "type": "JR","args": ["reg"] ,
            "fields": {"op": 1,"rs": "#1","ceros": 0 }
          }
},
```

Finalmente se agrega una nueva salida a la unidad de control, *JR* y se la configura para que al recibir un opcode igual a 1 esta salida sea activada y el resto permanezca sin activarse:

```
"control": {
    ...
    "1": { "JR":1 }
},
```

2.3.2. Modificación al datapath

A continuación se presenta una imagen del datapath con los cambios realizados; los nuevos componentes agregados están resaltados con un círculo rojo:

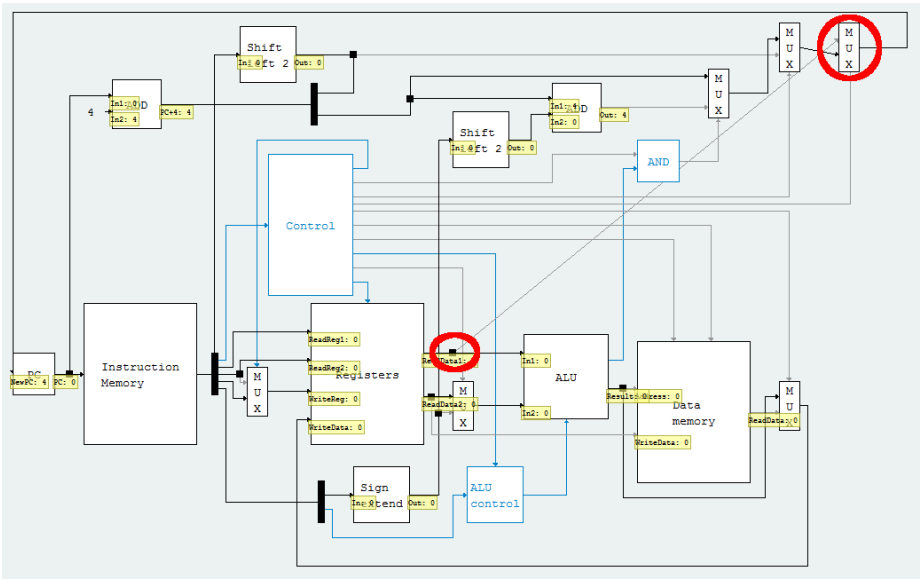

Figura 5: Datapath monociclo con la instrucción *jr*

Como puede observarse, se agregó un *fork* a la salida de los registros, el cual se dirige hacia una entrada de un nuevo multiplexor. La manera en la que esto funciona es la siguiente:

Cuando la instrucción a ejecutarse es *jr* ingresa un opcode igual a 1 a la unidad de control; esto setea todas las salidas a 0 menos la salida *JR*, la cual es seteada en 1. Esta salida se dirige hacia el selector del nuevo multiplexor agregado. Si este bit es 1, entonces el multiplexor elige la entrada proveniente del registro y se lo pasa al *PC* para que se salte a esa dirección.

Si el bit selector es 0, como ocurre en el resto de las instrucciones, el datapath funciona normalmente, ya que el multiplexor utiliza la entrada que anteriormente tenía salida al *PC*, por lo que no realiza ningún cambio.

2.3.3. Pruebas de jr

Estas fueron las pruebas que se corrieron para verificar el correcto funcionamiento de la nueva instrucción:

Prueba 1:

```
li $t0,20 #direccion en bytes de la sexta instruccion
li $t1,1
jr $t0
li $t2,2
li $t3,3
li $t4,4
addi $t5,$t4,3
# al finalizar el programa deberian quedar t1=1, t4=4,
# t5=7 y t2 y t3 no deberian ser modificados
```

Prueba 2:

```
    li $t8, 8 #direccion en bytes de la
          #instruccion de inicio (beq...)
    li $t0, 1
inicio:
    beq $t0, $0, fin
    li $t1, 1
    li $t2, 2
    li $t3, 3
    subi $t0, $t0, 1
    jr $t8
    li $t4, 4
    li $t5, 5
fin:
    li $t6, 6
# al finalizar el programa deberian quedar t0=0, t1=1,
# t2=2, t3=3, t6=6 y t4 y t5 no deberian ser modificados
```

2.4. Punto 4

En este punto se implementó la instrucción *blt* (Branch if Less Than) en el datapath pipeline. En un comienzo se intentó resolverlo únicamente con pseudoinstrucciones, lo cual resultó insatisfactorio. Por ello, se buscó y se llegó a la conclusión que no bastaba sólo con eso, sino que además se tenía que implementar la instrucción de *bne* (Branch if Not Equal).

2.4.1. Modificación al set de instrucciones

Para llevar a cabo este punto, como ya se dijo, se agregó la instrucción *bne* y la pseudoinstrucción *blt*.

Para el caso de *bne* se tuvo que, además de agregar la instrucción, agregar una línea más de la unidad de control que indique si es un *bne*. Así, el archivo *default_no_jump.set* sufrió las siguientes modificaciones:

```
"instructions": {
    ...
    "bne": { "type": "I", "args": ["reg", "reg", "offset"],
             "fields": {"op": 5, "rs": "#1", "rt": "#2", "imm": "#3"},
             "desc": "PC += ($t1 != $t2) ? (offset * 4 + 4) : 4"
           },
    ...
},
"pseudo": {
    ...
    "blt": { "args": ["reg", "reg", "offset"],
             "to": ["slt $1, #1, #2", "bne $1, $0, #3"],
             "desc": "PC += ($t1 < $t2) ? (offset * 4 + 4) : 4"
           },
    ...
},
"control": {
    ...
    "4": {"ALUOp": 1, "ALUSrc": 0, "BranchEQ": 1},
    "5": {"ALUOp": 1, "ALUSrc": 0, "BranchNE": 1},
    ...
}
```

2.4.2. Modificación al datapath

La idea para este punto fue desarrollar más que nada la instrucción *bne*, que luego utilizará la pseudoinstrucción *blt*. Para ello, se hizo uso de las líneas *BranchEQ* y *BranchNE* de la unidad de control, junto con la salida *zero* de la ALU. El concepto general recae en que se hará un branch si se cumplen algunas de estas 2 condiciones:

- Es un *bne* (que será indicado por la línea de control *BranchNE*) y el resultado de restar los registros intervinientes no dio cero. Esto último indicaría que no son iguales, mediante la salida *zero* con valor igual a *0* de la ALU.

- Es un *beq* (que será indicado por la línea de control *BranchEQ*) y la salida *zero* de la ALU tiene un valor 1 (Es decir, que se restaron los registros y el resultado dio cero, por lo que son iguales).

En cualquiera de los dos casos, el *PC* no tomará el valor de la siguiente instrucción sino que se le asignará el valor del *target*, además de hacer un *flush* de los registros del pipeline.

Esto es lo que se quiere llevar a cabo con el circuito de 2 ANDs, 1 NOT y 1 OR que se encuentra arriba del *Data Memory*, tal como se puede ver en la figura 6.

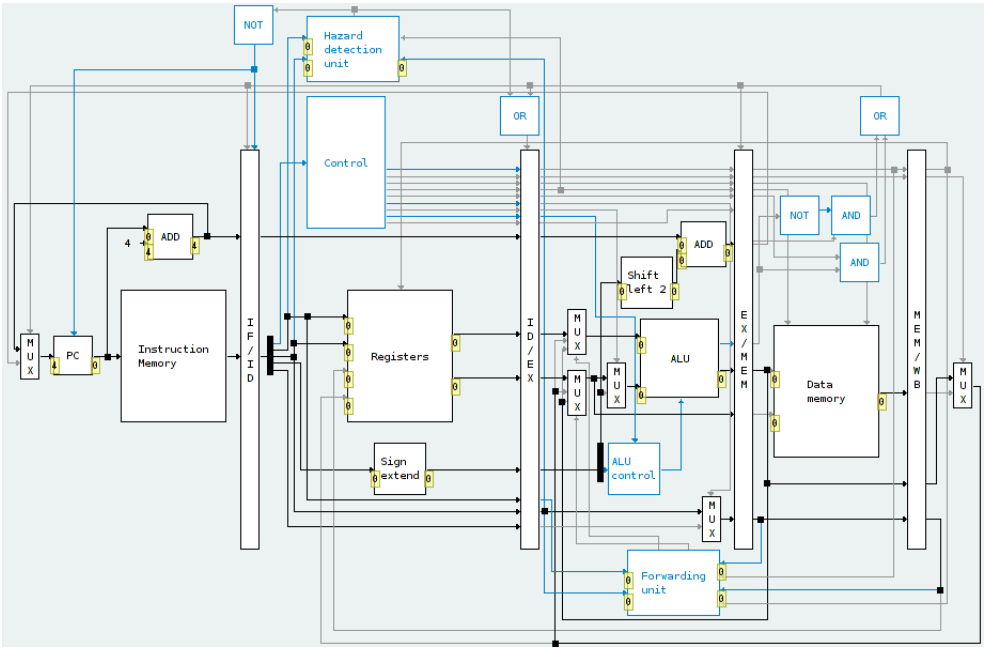

Figura 6: Datapath pipeline con instrucción *bne* y pseudoinstrucción *blt*.

2.4.3. Pruebas de blt

Estas fueron las pruebas que se corrieron para verificar el correcto funcionamiento de las nuevas instrucciones:

Prueba 1:

```
    li $t0, 1
    blt $t0, $0, fin
    li $t1, 1
    li $t2, 2
fin:
    li $t3, 3
    # al finalizar deberian quedar
    # - t0=1, t1=1, t2=2, t3=3
```

Prueba 2:

```
    li $t6, 6
    subi $t5, $t6, 1
    blt $t5, $t6, fin
    li $t1, 1
    li $t2, 2
fin:
    li $t3, 3
    # al finalizar deberian quedar
    # - t5=5, t6=6, t3=3
    # - t1 y t2 no deberian ser modificados
```

3. Conclusiones

Realizar este trabajo práctico nos permitió interiorizarnos aún más con la arquitectura de CPU MIPS, más en particular con el datapath de la misma, y con el simulador DrMIPS. Nos sirvió para comprender como funciona el datapath monociclo y el pipeline, así como también para darnos cuenta que hay distintas formas de implementar una instrucción.

En algunos casos es posible agregar instrucciones sin modificar el datapath, como en el caso de las instrucciones *sll* y *srl*, pero generalmente requiere el agregado de nuevos componentes, lo que implica más costos, un aumento en la complejidad del datapath y posiblemente una disminución en la velocidad de ejecución. Por lo cual, a la hora de elegir si agregar una nueva instrucción, deben considerarse estas desventajas.

Por último, pudimos observar que en determinados casos, como en el de *blt*, no es posible agregar una nueva instrucción al datapath, sino que es necesario implementarla como una combinación de instrucciones ya existentes, o bien primero deben implementarse nuevas instrucciones (como el caso de *bne*), para luego sí poder llevar a cabo la pseudoinstrucción *blt*.

4. Referencias

[1] DrMIPS, `https://bitbucket.org/brunonova/drmips/wiki/Home`.
[2] DrMIPS, `https://https://github.com/brunonova/drmips`.

66:20 Organización de computadoras
Trabajo práctico 2: Data Path.

1. Objetivos

El objetivo de este trabajo es familiarizarse con la arquitectura de una CPU MIPS, específicamente con el datapath y la implementación de instrucciones. Para ello, se deberán agregar instrucciones a diversas configuraciones de CPU provistas por el simulador DrMIPS [1]

2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá ser entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes.

Además, es necesario que el trabajo práctico incluya (entre otras cosas, ver sección 8), la presentación de los resultados obtenidos, explicando, cuando corresponda, con fundamentos reales, las causas o razones de cada resultado obtenido.

El informe deberá respetar el modelo de referencia que se encuentra en el grupo[1], y se valorarán aquellos escritos usando la herramienta TEX / LATEX.

4. Recursos

Usaremos el programa DrMIPS [1] para configurar y simular el data path de un procesador MIPS [4], tanto uniciclo como multiciclo.

5. Descripción.

5.1. Introducción

El programa DrMIPS nos permite evaluar distintos diseños de datapath para procesadores MIPS32, al darnos la posibilidad de organizarlo como queramos. Si bien sólo puede haber uno de algunos de los componentes del DP (como el registro de PC o la unidad de control), podemos poner sumadores, multiplexores, extensores de signo y conexiones arbitrariamente. También es

[1] `http://groups.yahoo.com/group/orga6620`

posible modificar el conjunto de instrucciones. Además de la estructura lógica del DP, DrMips nos permite escribir programas simples y simular su ejecución en el DP, mostrando los valores que toman las diversas entradas y salidas de cada elemento. El programa se puede conseguir en `https://bitbucket.org/brunonova/drmips/wiki/Home`, o se puede descargar para Ubuntu, ya sea desde el repositorio de Ubuntu (aunque la versión está desactualizada) o autorizando un repositorio externo (ver [2]).

5.2. Datapaths

El programa viene con algunos DP ya implementados, a saber:
Uniciclo:

- `unycicle.cpu`: El DP uniciclo por defecto.
- `unycicle-no-jump.cpu`: Variante más simple del DP uniciclo que no soporta la instrucción `j`.
- `unycicle-no-jump-branch.cpu`: Una variante aún más simple que no soporta `jump` ni `branch`.
- `unycicle-extended.cpu`: Una variante que soporta instrucciones adicionales, como multiplicación y división.

Multiciclo:

- `pipeline.cpu`: El DP de pipeline por defecto, implementa detección de hazards. Los DP de pipeline no soportan la instrucción `j` (salto).
- `pipeline-only-forwarding.cpu`: Variante del DP de pipeline que implementa forwarding pero no genera stalls (genera resultados incorrectos).
- `pipeline-no-hazard-detection.cpu`: Otra variante que no hace hazard detection de ninguna manera (genera resultados incorrectos).
- `pipeline-extended.cpu`: Una variante que soporta instrucciones adicionales, como multiplicación y división, como `unycicle-extended.cpu`.

5.3. Instrucciones a implementar

1. Implementar las instrucciones `sll` y `srl` (Shift Left Logical y Shift Right Logical) en el DP `unicycle.cpu`.
2. Implementar la instrucción `j` en el DP `pipeline.cpu`.
3. Implementar la instrucción `jr` (Jump Register) en el DP `unicycle.cpu`.
4. Implementar la instrucción `blt` (Branch if Less Than) en el DP `pipeline.cpu`. ¿Se puede hacer nativamente, o hay que agregar pseudoinstrucciones? ¿Por qué?

Para todos los casos, verificar que no se produzcan hazards de control o de datos.

6. Implementación.

Los archivos antes mencionados, así como los archivos `.set` que contienen los datos del conjunto de instrucciones, están en formato JSON [3], y se pueden modificar con un editor de texto. Se sugiere uno que pueda hacer *color syntax highlighting*, como el `gedit` que viene con el Ubuntu. La explicación de los formatos se encuentra en el archivo `configuration-en.pdf` que se distribuye con el programa.

7. Pruebas

En todos los casos debe verificarse que la instrucción se ejecute correctamente. Esto implica que el PC tome el valor deseado, y además que en el caso del DP multiciclo no se produzcan hazards, como ser la ejecución de la instrucción siguiente al salto, o en el caso de utilizar el valor de un registro, que éste tenga el valor correcto.

8. Informe.

Se debe entregar:

- Informe describiendo el desarrollo del trabajo práctico.
- Capturas de pantalla de los DP modificados.
- Programas de prueba.
- CD conteniendo los DP, los programas de prueba y los conjuntos de instrucciones usados en cada caso.
- Este enunciado.

9. Fechas de entrega.

- Primera entrega: Jueves 19 de Noviembre.
- Revisión: Jueves 26 de Noviembre.
- Vencimiento: Jueves 3 de Diciembre.

Referencias

[1] DrMIPS, `https://bitbucket.org/brunonova/drmips/wiki/Home`.

[2] PPA de Bruno Nova, `https://launchpad.net/~brunonova/+archive/ubuntu/ppa`.

[3] ECMA-404 The JSON Data Interchange Standard, `http://www.json.org/`.

[4] "Computer organization and design: the hardware-software interface", John Hennessy, David Patterson. Capítulo 5.