Trabajo Práctico 3
Datapath y Pipeline

Ignacio Velasco, *Padrón 99796*
igvelasco@fi.uba.ar
Slack: Ignacio Velasco

Carolina Rocío Pistillo, *Padrón 99177*
caropistillo@gmail.com
Slack: caropistillo

Juan Pablo Donato, *Padrón 100839*
judonato@fi.uba.ar
Slack: DonatoJP

1er. Cuatrimestre de 2019
86.37 / 66.20 Organización de Computadoras − Práctica Jueves
Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

Presentación de la resolución del trabajo práctico 3 de la materia 86.37/66.20 Organización de Computadoras de la Facultad de Ingenieria, el cuál tiene como tema principal el Datapath y Pipeline de un microprocesador MIPS32, junto con la implementación de algunas instrucciones para dicho dispositivo mediante la utilización del programa DrMIPS que nos permite evaluar distintos diseños para procesadores MIPS32.

Índice

1. Introducción

El presente informe tiene como objetivo describir el desarrollo de la resolución del tercer trabajo práctico de la materia 86.37/66.20 Organización de Computadoras, el cuál presenta como desafío la implementación de algunas instrucciones para un microprocesador MIPS32, haciendo uso del programa DrMIPS que nos permite evaluar distintos diseños de dicho procesador agregando componentes de control y modificando el Datapath.

En la sección *Análisis del programa DrMIPS* analizaremos brevemente las funcionalidades del programa, junto con los archivos que necesita para utilizarse, y así poder explicar como utilizamos dicho programa y donde debíamos modificarlo para cumplir con los objetivos.

Luego, en la sección *Diseño e Implementación* explicaremos cuales eran los objetivos de este trabajo práctico, y como fue el diseño de nuestra resolución a esos problemas. También mostraremos casos iniciales que no dieron resultado.

Más adelante, presentaremos en la sección *Casos de prueba* algunos ejemplos para probar las soluciones adoptadas en la sección anterior, y así mostrar como es que funciona realmente el trabajo realizado.

Por último, en la sección *Conclusiones* daremos conclusiones sobre los aspectos mas importantes del trabajo realizado.

2. Análisis del programa DrMIPS

DrMIPS es un simulador gráfico de un procesador MIPS32, utilizado para propósitos educativos y de desarrollo. Si bien está disponible en varios formatos y versiones, nosotros utilizamos la versión 2.0.3 compatible en Linux. Al iniciarse el programa, se puede ver una interfaz como se ve en la figura 1.

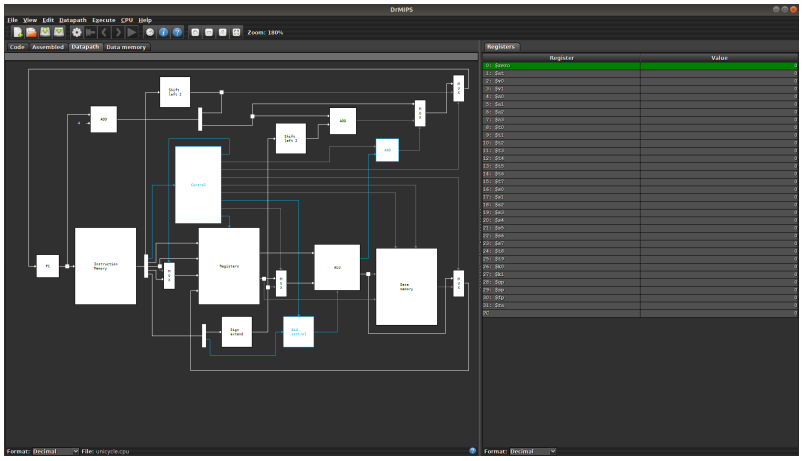


Figura 1: Pantalla inicial DrMIPS

En la pestaña *Datapath* podemos ver la imagen del camino de datos actual del procesador, con sus componentes y conexiones. En simultaneo, en la pestaña *Registers* podemos ver el estado actual de los registros, actualizando sus valores a medida que se ejecuta nuestro programa.

Para escribir nuestro programa en código Assembly, podemos hacerlo desde la pestaña `Code`, la cuál es un editor de texto que luego podemos compilar para generar las instrucciones a ejecutar. Una vez compiladas, podemos ver las instrucciones generadas, junto con su posición en la memoria, como se muestra en la figura 2

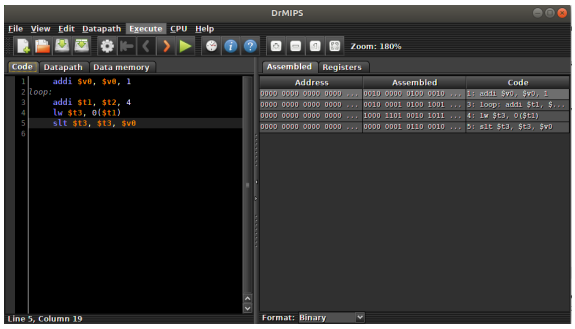
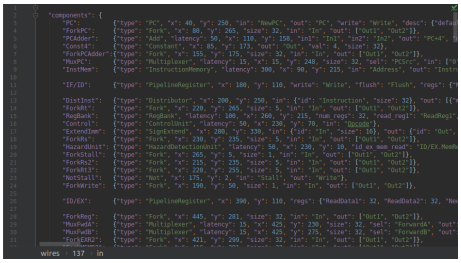
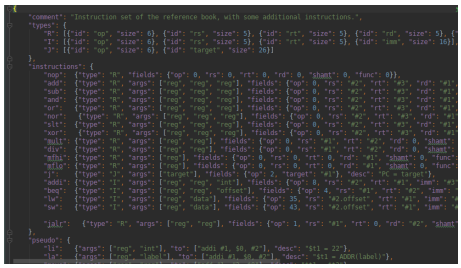


Figura 2: Editor de texto Code y las instrucciones generadas

Además, el programa cuenta con una serie de archivos necesarios para establecer la configuración del microprocesador. Aquellos archivos con formato **.cpu**, como por ejemplo *unicycle.cpu*, son los que establecen cuáles son los componentes a utilizar, junto con sus posiciones en la pantalla, sus conexiones entre sí, etcétera. Un ejemplo de este tipo de archivo se puede visualizar en la imagen 3a. Por otro lado, el conjunto de instrucciones posibles a utilizar en el programa se encuentran en los archivos de formato **.set**, como *default.set*. En este tipo de archivos, se detalla las instrucciones, estableciendo su formato específico, el tipo de operación a realizar en la unidad aritmético lógica, las señales de control a utilizar, etcétera. En la figura 3b podemos ver un ejemplo de este tipo de archivos. Ambos tipos de archivos tienen formato JSON.



(a) Formato de archivo .cpu



(b) Formato de archivo .set

Entonces, para poder implementar las soluciones a los problemas planteados, debemos modificar estos archivos en conjunto para:

- Generar las instrucciones a implementar, con su respectiva firma, tipo, formato y operaciones involucradas.
- Colocar dispositivos necesarios en el Datapath para lograr la tarea.
- Agregar y/o modificar conexiones entre componentes del datapath para lograr el efecto deseado.

Finalmente, una vez modificados los archivos .cpu y .set, y luego de haber generado nuestro código assembly, el programa nos permite ejecutarlo paso a paso para seguir el desarrollo de la secuencia de instrucciones y así detectar errores, o simplemente llegar al resultado final. Además, el programa nos permite visualizar los valores en las entradas de los componentes instante a instante y así detectar *hazards*.

3. Diseño e Implementación

En esta sección del informe explicaremos los objetivos del trabajo práctico, y luego la implementación de nuestra solución a cada uno de esos problemas por separado.

Algunas de las soluciones adoptadas resultaron exitosas en la primer prueba, pero otras no lo hicieron en la primera vez. En cualquiera de los casos, se presentarán ambas soluciones explicando la modificación que solucionó los errores de la versión anterior.

3.1. Instrucciones a implementar

Para este trabajo, se pidieron implementar 5 tipos de instrucciones en distintas condiciones. Estas son:

- Implementar la instrucción j en el Datapath *pipeline.cpu.*
- Implementar la instrucción sll (Shift Left Logical) en el DP unicycle.cpu.
- Implementar la instrucción srl (Shift Right Logical)en el DP pipeline.cpu.
- Implementar la instrucción jalr (Jump and Link Register) en el DP unicycle.cpu.
- Implementar la instrucción jalr en el DP pipeline.cpu.

Para hacerlo de forma organizada y prolija, hemos decidido realizar cada una de las implementaciones por separado, dividiendo cada respuesta en dos archivos .cpu y .set distintos. Es decir, no se verán las soluciones a otros problemas dentro de la respuesta a uno de ellos, así mejoramos la prolijidad y el entendimiento de los gráficos que se presentarán en la sección que sigue, donde mostraremos el diseño y la implementación a los problemas presentados.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Figura 4: Formato de instrucciones extraido de [5]

3.2. Implementación de las soluciones

3.2.1. Instrucción j en pipeline.cpu

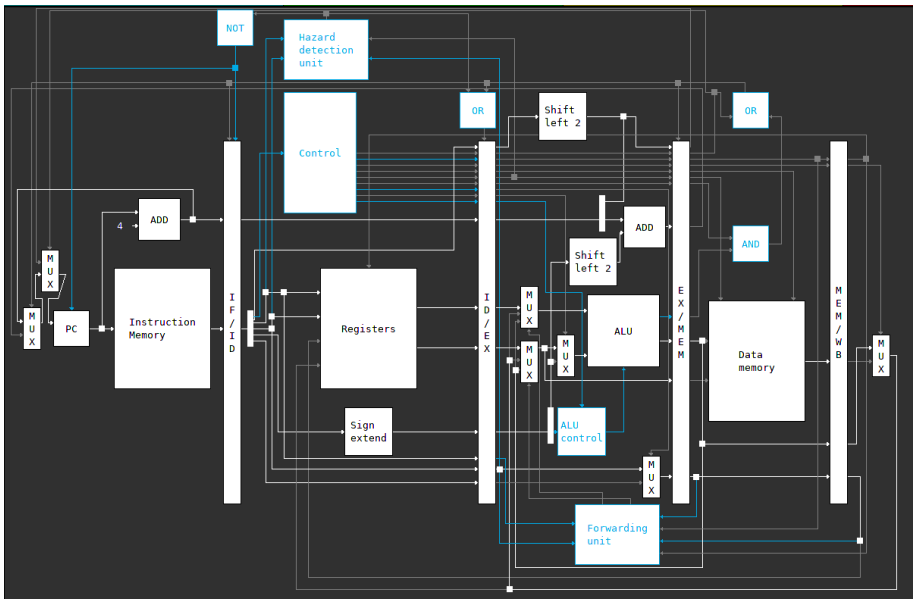


Figura 5: Datapath con la implementación de j en pipeline.cpu

La figura 5 muestra el esquema final de la solución a este problema. Además de los componentes que ya estaban presentes en *pipeline.cpu*, se agregaron:

- Un multiplexor '*MuxJump*' en la entrada del PC que decide si a la entrada del mismo tendremos el valor PC+4 o la dirección del target.
- Una señal de control '*Jump*' que comandará el multiplexor anterior, y que además 'flusheará' los registros del pipeline cuando una instrucción jump se ejecute.

- Una compuerta OR para, junto con la señal *jump*, vaciar los registros del pipeline.
- Otros componentes para lograr el conexionado, como bifurcaciones, un concatenador y un distribuidor de bits.

Básicamente, los componentes mas importantes son los primeros tres. De esta forma, cuando el microprocesador detecte la instrucción Jump a ejecutarse, determinará la dirección del target desde los bits [25-0] de la instrucción (por tener formato del tipo **J-format**, como se muestra en la figura 4), la cuál llegará al PC por medio del multiplexor mencionado en la lista anterior, junto con la señal de control 'Jump'. Para evitar *hazards*, es decir, que no se ejecuten las instrucciones que siguen a continuación del salto, utilizando la compuerta OR y la señal de control, optamos por vaciar los registros de control.

Sin embargo, esta versión no fue la primera en realizarse. La versión anterior a ella no contaba con el sistema de flush de los registros de pipeline. Entonces, lo que sucedía es que las instrucciones posteriores al jump se seguían ejecutando, y podían llegar a modificar los valores en los registros. La imagen 6 muestra el gráfico de esta primer versión, donde se ve que no estaba la compuerta OR mencionada y por lo tanto no se vaciaban los registros.

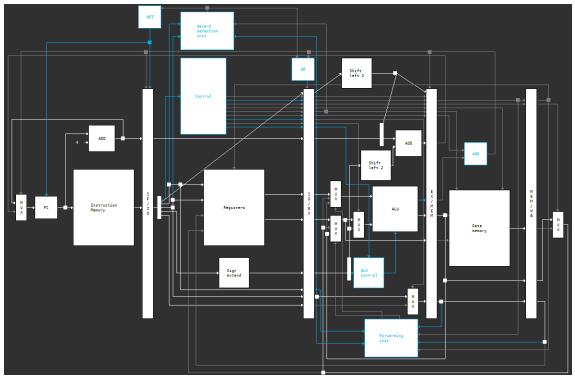


Figura 6: Primera versión que contenía hazards

3.2.2. Instrucción sll en unicycle.cpu

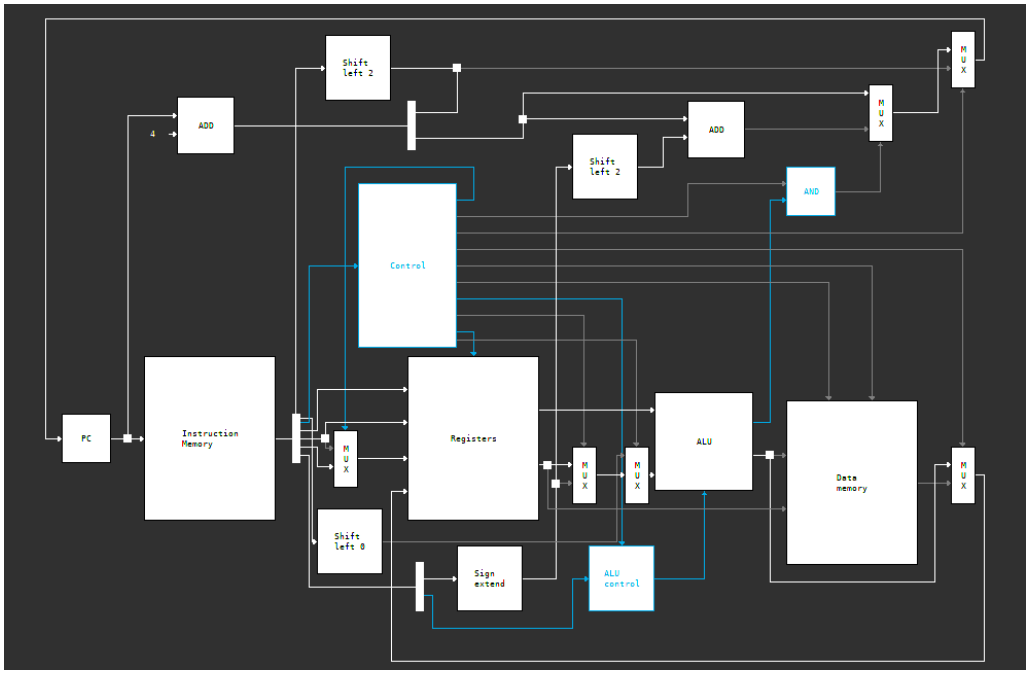


Figura 7: Datapath con la implementación de sll en unicycle.cpu

La instrucción **sll** es una instrucción del tipo R (figura 4), la cuál hará uso del campo *shamt* de la misma para desplazar lógicamente el registro rs tantos bits como *shamt* indique hacia la izquierda. Para comenzar, debimos definir dicha instrucción en el archivo .set correspondiente, configurando correctamente los campos (por ejemplo, el campo rt igual a 0 ya que no se utiliza), o el opcode correspondiente). Además, se agregó la operación "sll.[en] la ALU, y se definió una nueva secuencia de señales de control.

En la imagen 7 vemos el datapath modificado para soportar la instrucción Shift Left Logical en un procesador MIPS uniciclo. La principales modificaciones que se realizaron fueron:

- El multiplexor en la entrada a la ALU, que seleccionará entre un registro o el campo *shamt* de la instrucción.
- La señal de control *SHAMTSrc* para comandar el multiplexor anteriormente mencionado y poder seleccionar como fuente el campo shamt.
- Un *Left Shifter* de 0 unidades. Utilizamos este componente básicamente como un .^{ex}tensor sin signo"debido a que el campo shamt posee 5 bits, y la entrada a la ALU requiere 32.

De esta forma, cuando se deba ejecutar la instrucción, la unidad de control seleccionará, en la entrada a la ALU, el registro a modificar y el campo *shamt*, haciendo uso de la señal *SHAMTSrc*.

3.2.3. Instrucción srl en pipeline.cpu

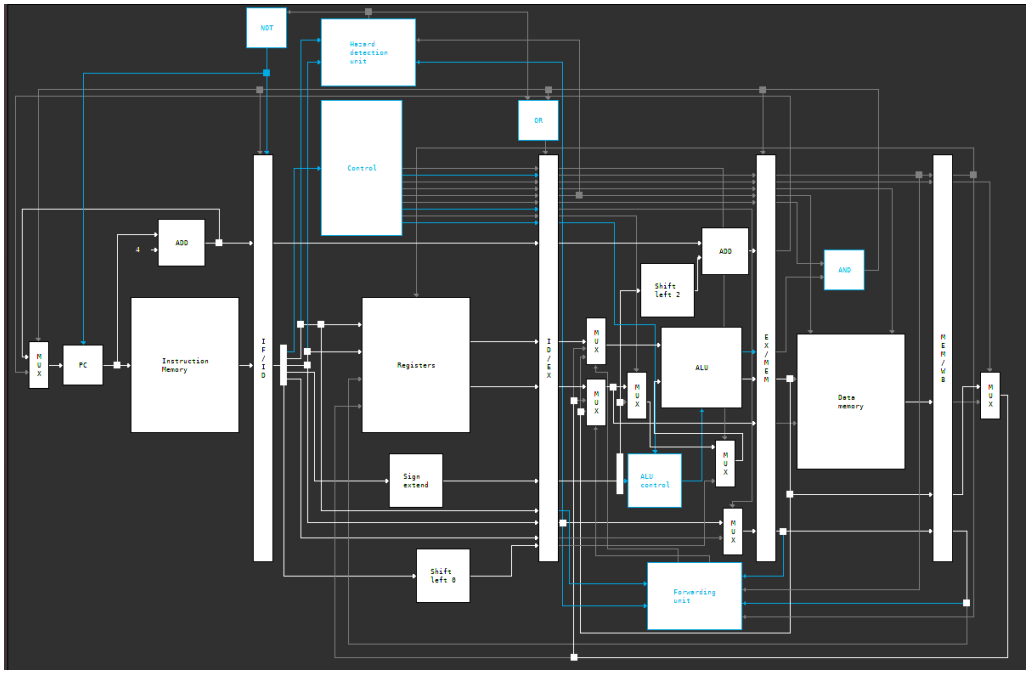


Figura 8: Datapath con la implementación de srl en pipeline.cpu

Para resolver este caso, nos basamos en el problema anterior: necesitamos algún componente que decida que una de las entradas a la ALU sea el campo *shamt* de la instrucción, ya que es el que nos indica la cantidad de bits a desplazar en el registro. Por lo tanto, las modificaciones en esta versión son exactamente las mismas que en el problema anterior, pero siempre cuidando que esta **no** es la versión de ciclo único. Es decir, la señal *SHAMTSrc* debe hacerse pasar de la etapa ID a la etapa EX.

3.2.4. Instrucción jalr en unicycle.cpu

En este caso, debiamos resolver dos problemas:

a) Llevar el valor de PC+4 hacia la entrada *WriteData* del *Register Bank* para escribir la dirección de la instrucción que sigue en un determinado registro

b) Llevar el dato de la dirección a ejecutarse desde el registro indicado en la instrucción hasta el PC.

Para lograrlo, y como se puede ver en la figura 9, agregamos los siguientes componentes:

- Una señal de control "Jalr"proveniente de la unidad de control, para comandar los siguientes multiplexores.
- Un multiplexor en la entrada a *WriteData* del *Register Bank* para decidir entre PC+4 o la salida *Result* de la unidad aritmético lógica.

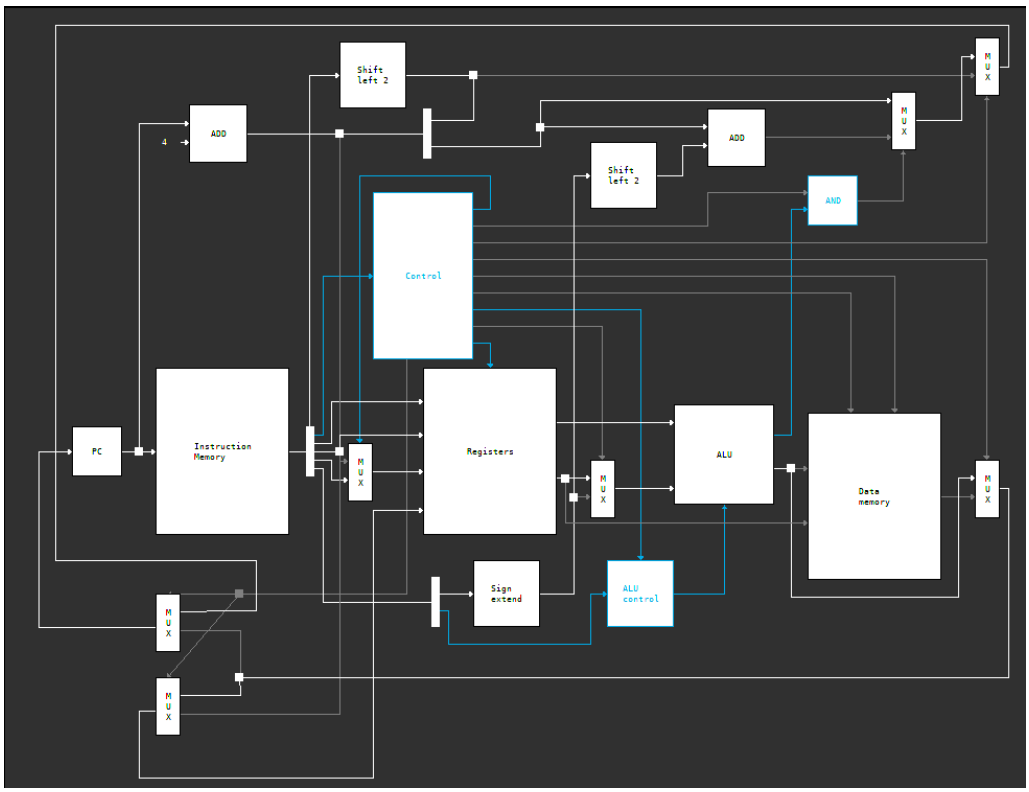


Figura 9: Datapath con la implementación de jalr en unicycle.cpu

- Un multiplexor en la entrada al Program Counter para conectarlo con la dirección de la siguiente instrucción, o con la dirección del target que se indica en la instrucción jalr.

De esta forma, cuando la señal *Jalr* de la unidad de control se encienda, el PC tendrá el valor de la dirección del target, mientras que en el registro destino se almacenará la dirección PC+4.

3.2.5. Instrucción jalr en pipeline.cpu

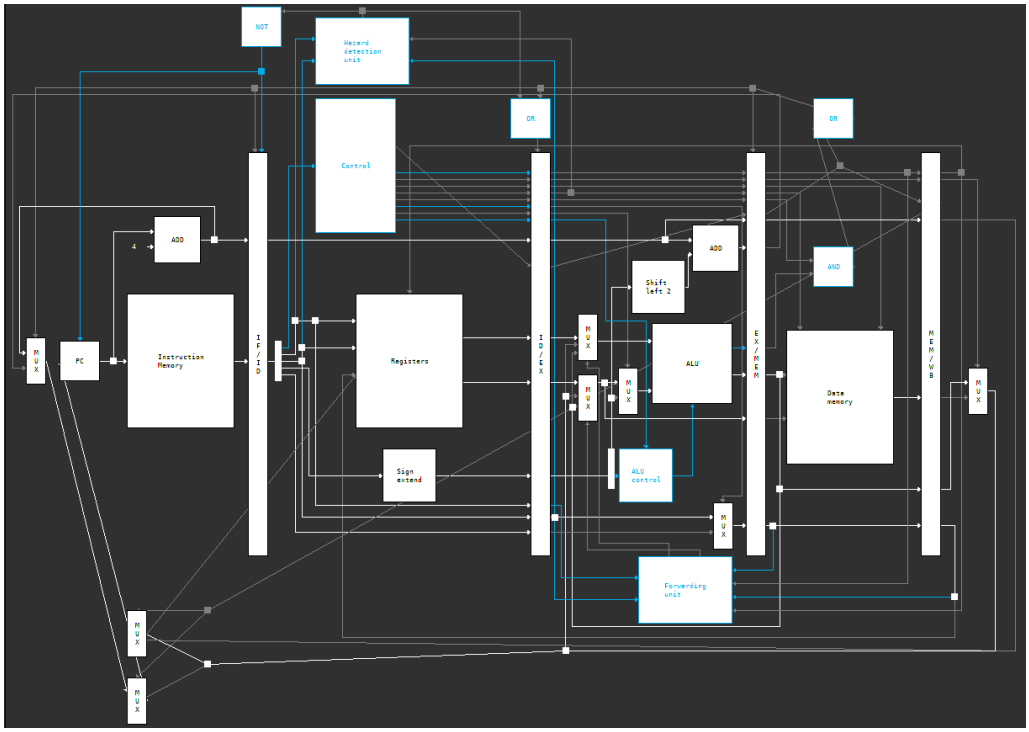


Figura 10: Datapath con la implementación de jalr en pipeline.cpu

Análogamente al caso anterior, en este ejemplo ponemos los mismos multiplexores para comandar la entrada al program counter y la entrada al *RegisterBank* para almacenar PC+4. Pero esta vez hay que considerar los siguientes problemas:

- El procesador es multi-ciclo. Esto significa que las señales los valores y las señales deben atravesar las distintas etapas de un pipeline para no tener hazards.
- Otra fuente de hazards es la ejecución de instrucciones posteriores al salto. Entonces, es necesaria alguna solución para evitar esto.

La imagen 10 muestra la solución adoptada para estos problemas. Primero, tanto los valores de PC+4 como los de la señal *Jalr* proveniente de la unidad de control se extendieron hasta la etapa WB del pipeline. Luego de esto, se colocaron los multiplexores para comandar las entradas que fueron mencionadas anteriormente. Finalmente, desde la señal *Jalr* se 'flushearon' los registros de pipeline para evitar hazards.

4. Casos de prueba

Ahora, procederemos a mostrar algunos ejemplos en código sobre las implementaciones detalladas en la sección anterior.

4.1. Primer caso: j en pipeline.cpu

Para este primer caso, tomamos el siguiente código de ejemplo:

```
    j target

no_target:
    addi $v1, $v1, 1

target:
    addi $v0, $v0, 1
    addi $v0, $v0, 1
    addi $v0, $v0, 1
    addi $v0, $v0, 1
```

La ejecución es sencilla: al llegar al salto, deberían ejecutarse las instrucciones a partir de *target* y **no así la instrucción de *no_target***. Es decir, el resultado final debería ser

$$\$V0 = 4$$

$$\$V1 = 0$$

Cómo se puede ver en la figura 12 el resultado es el esperado. Además, no hay *hazards* debido a que el registro $V1 mantiene su valor original 0.

Register	Value
0: $zero	0
1: $at	0
2: $v0	4
3: $v1	0
4: $a0	0
5: $a1	0
6: $a2	0
7: $a3	0
8: $t0	0
9: $t1	0
10: $t2	0
11: $t3	0
12: $t4	0
13: $t5	0
14: $t6	0
15: $t7	0
16: $s0	0
17: $s1	0
18: $s2	0
19: $s3	0
20: $s4	0
21: $s5	0
22: $s6	0
23: $s7	0
24: $t8	0
25: $t9	0
26: $k0	0
27: $k1	0
28: $gp	0
29: $sp	0
30: $fp	0
31: $ra	0
PC	40

Figura 11: Resultados del primer caso de prueba

Por otro lado, para verificar que el flush a los registros de pipeline no influye a las instrucciones anteriores al salto, ejecutando el siguiente código y vemos que el resultado es correcto.

```
1      addi $v0, $v0, 1
2      j yes
3  no:
4      addi $v0, $v0, -1
5  yes:
6      addi $v1, $v1, 1
7
```

Register	Value
0: $zero	0
1: $at	0
2: $v0	1
3: $v1	1
4: $a0	0
5: $a1	0
6: $a2	0
7: $a3	0
8: $t0	0
9: $t1	0
10: $t2	0
11: $t3	0
12: $t4	0
13: $t5	0
14: $t6	0
15: $t7	0
16: $s0	0
17: $s1	0
18: $s2	0
19: $s3	0
20: $s4	0
21: $s5	0
22: $s6	0
23: $s7	0
24: $t8	0
25: $t9	0
26: $k0	0
27: $k1	0
28: $gp	0
29: $sp	0
30: $fp	0
31: $ra	0
PC	32

Figura 12: Resultados del segundo caso de prueba

4.2. Segundo caso: sll en unicycle.cpu

Ahora, para probar la instrucción sll, simplemente la ejecutaremos con distintos parámetros y evaluaremos los resultados.

```
1 addi $v0, $v0, 1
2 sll $v0, $v0, 2
3
4
```

Register	Value
0: $zero	0
1: $at	0
2: $v0	4

Figura 13: Primer caso

Como se puede ver, todos los resultados son los esperados.

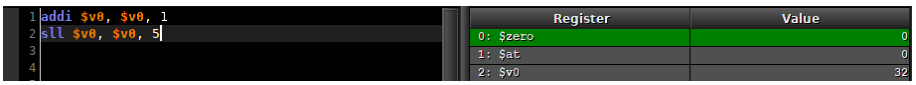


Figura 14: Segundo caso

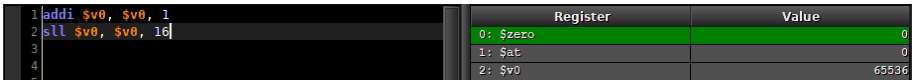


Figura 15: Tercer caso

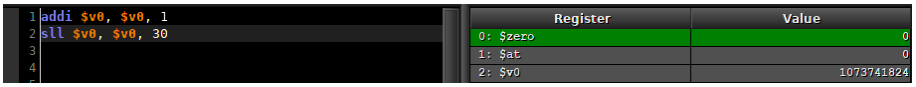


Figura 16: Cuarto caso

4.3. Tercer caso: srl en pipeline.cpu

Al igual que el caso anterior, presentaremos algunos resultados de instrucciones srl con distintos parametros.

Figura 17: Primer caso

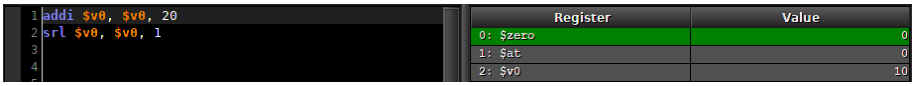


Figura 18: Segundo caso

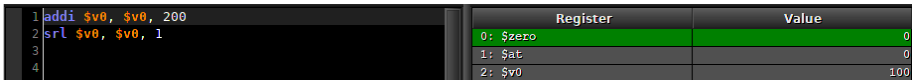


Figura 19: Tercer caso

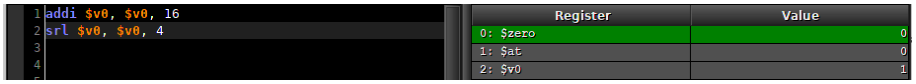


Figura 20: Cuarto caso

Aqui también todos los resultados son los esperados.

4.4. Cuarto caso: jalr en unicycle.cpu

Para este caso, intentaremos, al igual que en el primer caso, que se produzcan los saltos, verificando:

- El valor PC+4 se almacene en el registro deseado
- El registro PC tome su nuevo valor

Luego de ejecutar el código de prueba, de la imagen 21 podemos destacar:

- El registro $V0 contiene el valor de la dirección a saltar
- Una vez producido el salto, se guarda exitosamente el valor de PC+4 (el cúal es 8) en el registro $V1
- Nunca se ejecuta la instrucción que sigue al *jalr*. El valor de $a1 nunca cambia.

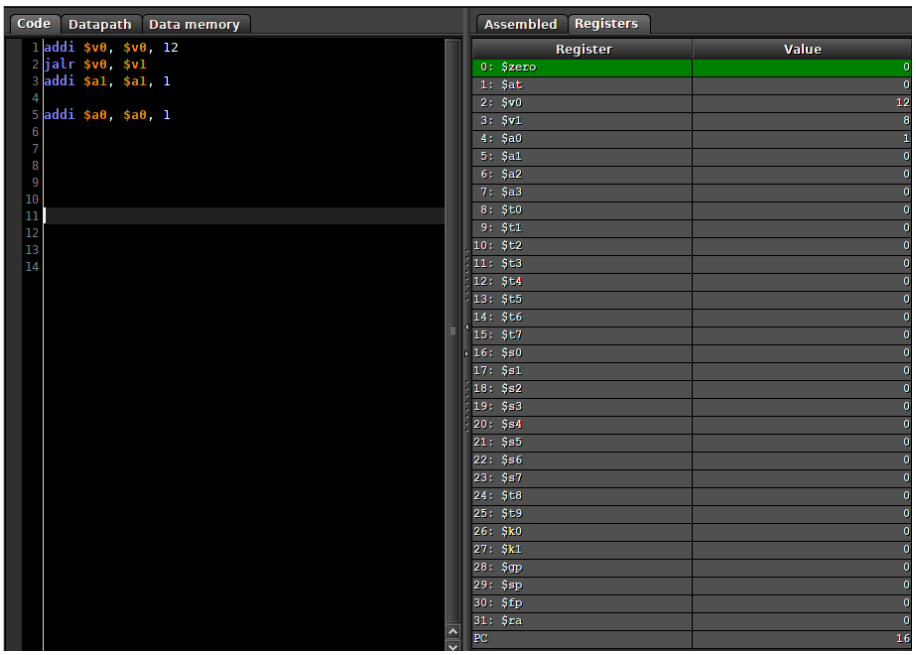


Figura 21: Caso de prueba para jalr en unicycle.cpu

- La ejecución continua por la instrucción que incrementa $a0 en 1.

El resultado, entonces, es el esperado. No se ven errores en cuanto a los valores de los registros, y tampoco hubo fallas en el salto.

4.5. Quinto caso: jalr en pipeline.cpu

Para el último caso, dado que es análogo al caso anterior, utilizaremos la misma prueba pero le agregaremos algunas instrucciones mas luego del salto, para verificar que no se produzcan hazards. Efectuamos la prueba y obtenemos:

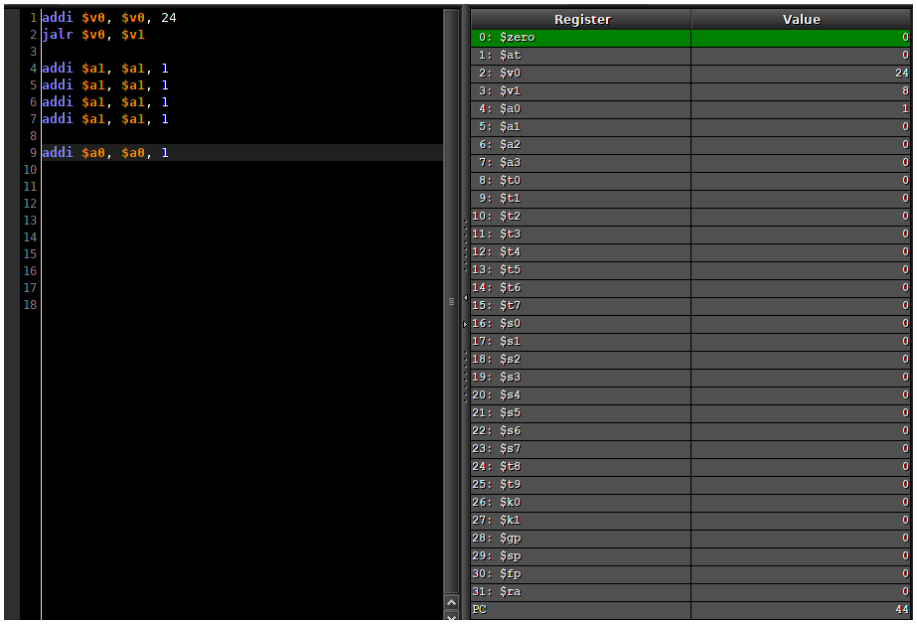


Figura 22: Caso de prueba para jalr en pipeline.cpu

De igual forma que antes:

- El registro $V0 contiene el valor de la dirección a saltar, en este caso 24
- Una vez producido el salto, se guarda exitosamente el valor de PC+4 (el cúal es 8) en el registro $V1

- Nunca se ejecuta la instrucción que sigue al *jalr*. El valor de $a1 nunca cambia.
- La ejecución continua por la instrucción que incrementa $a0 en 1.

Es decir, que la ejecución fue exitosa.

5. Conclusión

Este trabajo práctico fue útil para el aprendizaje en profundidad del funcionamiento del datapath de un microprocesador MIPS32, gracias a una herramienta interactiva de aprendizaje como lo es DrMIPS. Como conclusiones a este trabajo práctico podemos afirmar que hemos cumplido con los objetivos propuestos en él, y estar familiarizados con el funcionamiento de este procesador, tanto uniciclo como su versión pipeline. También pudimos resolver rápidamente los problemas surgidos en la resolución del mismo, como por ejemplo los casos de *hazards* en el primer ejemplo. Puede quedar como objetivo a futuro la implementación de otras instrucciones, o bién la utilización de las ya existentes para implementar otras pseudo-instrucciones.

Por último, destacaremos el hecho que, si bien las implementaciones propuestas cumplen con los requerimientos del trabajo, algunas de ellas no son muy eficientes. Por ejemplo, el hecho de tener que perder ciclos de trabajo en vaciar los registros de pipeline para evitar *hazards* en el desarrollo del programa no es la manera mas eficiente de hacerlo. Aún asi, es útil para este trabajo por tener la ventaja de ser simple en cuanto implementación, y es por este motivo que se decidió por este tipo de soluciones.

Referencias

[1] DrMIPS, https://brunonova.github.io/drmips/.

[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] MIPS Arquitecture https://en.wikipedia.org/wiki/MIPS_architecture

[5] “Computer organization and design: the hardware-software interface”, John Hennessy, David Patterson. Capitulo 5.

66:20 Organización de computadoras
Trabajo práctico 3: Datapath y pipeline.

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 [1], o se puede descargar para Ubuntu, ya sea desde el repositorio de Ubuntu (aunque la versión a veces 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 la instrucción `j` en el DP `pipeline.cpu`. Verificar que no se produzcan hazards.
2. Implementar la instrucción `sll` (Shift Left Logical) en el DP `unicycle.cpu`.
3. Implementar la instrucción `srl` (Shift Right Logical)en el DP `pipeline.cpu`.
4. Implementar la instrucción `jalr` (Jump and Link Register) en el DP `unicycle.cpu`.
5. Implementar la instrucción `jalr` en el DP `pipeline.cpu`.

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 20 de Junio.
- Vencimiento: Jueves 27 de Junio.

Referencias

[1] DrMIPS, `https://brunonova.github.io/drmips/`.

[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.