Trabajo Práctico 3
Datapath y pipeline

Franco Liberali, *Padrón 99491*
franco.liberali@gmail.com

Matías Iglesias, *Padrón 99635*
matiaseiglesias@yahoo.com

Edson Justo, *Padrón 97775*
justo.edson@gmail.com

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

09/05/2019

1. Introducción

El objetivo del trabajo practico es implementar las funciones:

- Jump (j) en DP pipeline
- Shift left logical (sll) en DP uniciclo
- Shift right logical (srl) en DP pipeline
- Jump and link register (jalr) en ambos DP's

Como vemos, se deben implementar tanto sobre el datapath(DP) pipeline de 5 etapas como en el DP unicycle. Para ello haremos uso de la aplicación DrMIPS la cual nos permite evaluar distintos diseños de datapath para procesadores MIPS32.

2. Diseño e implementación

A continuación se mostraran los DP originales y seguido los distintos diseños realizados con las modificaciones necesarias para la implementación de las distintas funciones antes mencionadas:

- DP unicycle.cpu original.

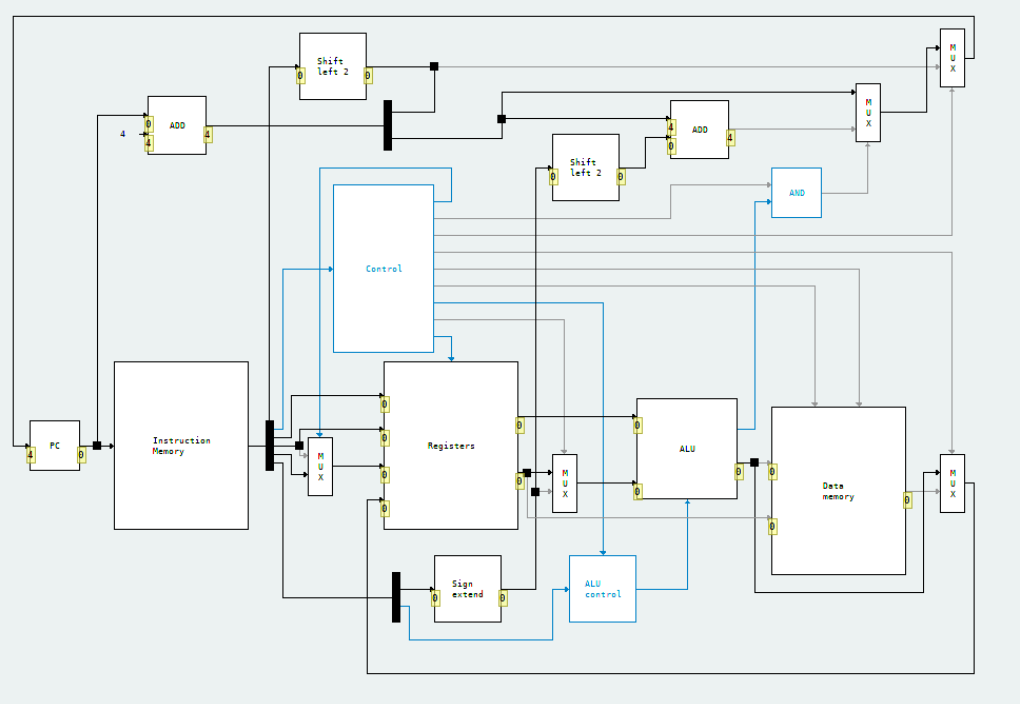

Figura 1: DP unicycle.cpu original

- Implementación la instrucción sll en el DP unicycle.cpu.

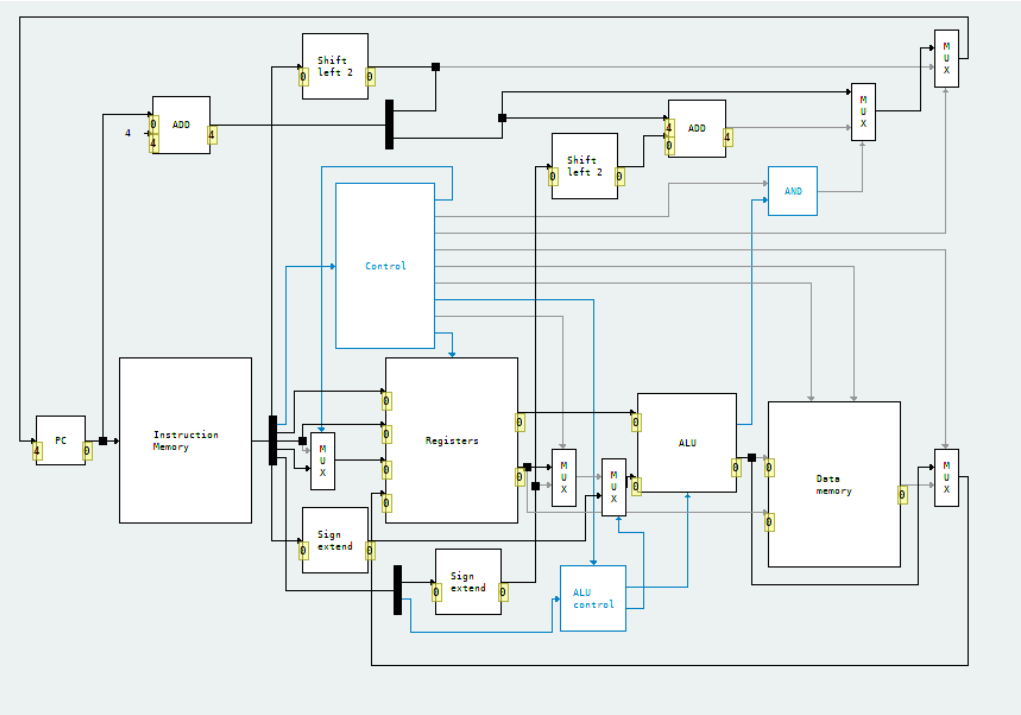

Figura 2: Implementación la instrucción sll

En este caso, para la implementación de la función sll se agrego al distribuidor la instrucción una nueva salida con los bits de shamt(10-6), la cual es a continuación extendida de 5 a 32 bits en el componente ExtendShamt y luego dirigir los 32 bits a un multiplexor ubicado antes de la segunda entrada a la ALU junto a la salida del MuxReg que era anteriormente la segunda entrada de la ALU.

De esta forma se configura el ALUControl para que cuando el código de operación de la instrucción se 2 y el func 0 (sll) active dicho multiplexor vía useShamt de manera que la segunda entrada a la ALU sea el shamt de la instrucción. A su vez, al encontrar ese código de operación y ese func enviará a la ALU Operation en 13, numero que fue configurado para que la ALU lleve a cabo sll.

- Implementación la instrucción jalr en el DP unicycle.cpu.

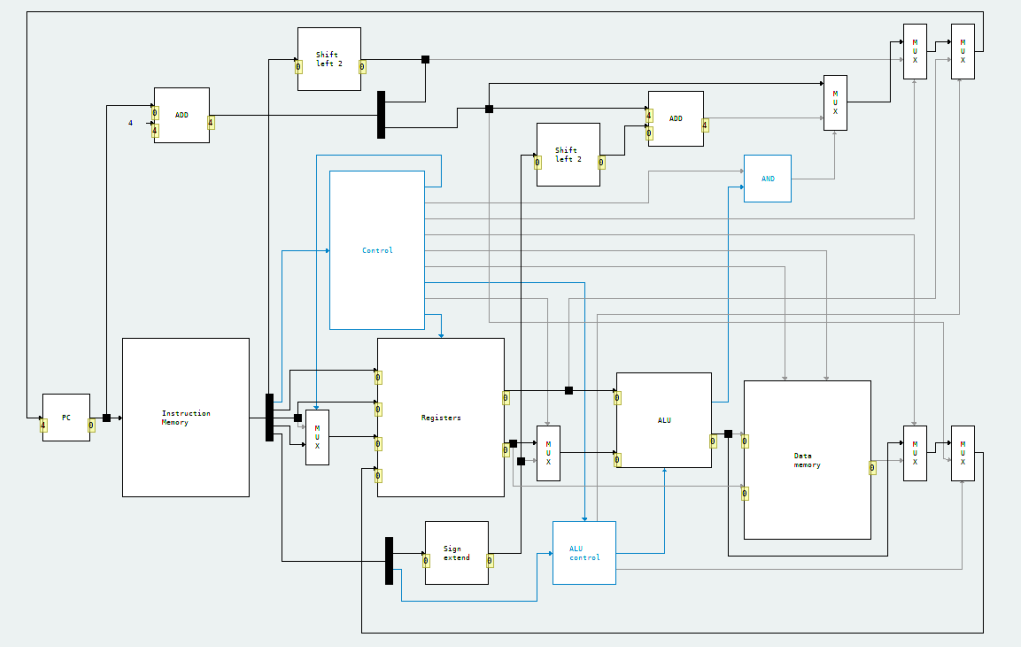

Figura 3: Implementación la instrucción j

A la hora de implementar la instrucción jalr, tuvimos que tener en cuenta que la misma debe realizar dos operaciones en el mismo ciclo, por un lado guardar el próximo PC en el registro $ra y por otro actualizar el PC con el valor contenido en el registro puesto en la instrucción. Esta dificultad se podría haber resulto con el uso de una pseudo instrucción que en realidad realice estas dos tareas por separado, pero se optó por implementar realmente esta instrucción ya que es posible.

Para guardar el siguiente PC a la instrucción de salto, se sacó otra salida del componente ForkBranch y se dirigió la mismo a la entrada de un nuevo multiplexor MuxJarlOrMem, cuya otra entrada es la salida del multiplexor MuxMem que seleccionaba si la entrada WriteData del RegBank debía venir de memoria o de la ALU. De esta manera, a estas dos opciones se le suma utilizar el valor del próximo PC, ya que la salida de este multiplexor está conectada a la entrada WriteData de la unidad encargada de administrar los registros.

En caso de ejecutar una instrucción jalr, los valores de código de operación y func serán 2 y 9 respectivamente. En caso de encontrar esta combinación, el ALUControl enviará 1 en la salida JalrReg para que los bits correspondientes al PC estarán disponibles en la entrada WriteData. En ese caso, la entrada RegWrite del banco de registros será el numero de registro de $ra ya que el mismo se encontrará siempre en el binario en la posición rd para la instrucción jalr.

Por otro lado, para actualizar correctamente el PC, los bits correspondientes a la nueva dirección, los cuales se encuentran en el registro especificado en la instrucción, serán llevados desde ForkReadData1, que bifurca el contenido del registro rs, a la entrada de un nuevo multiplexor MuxJumpOrJumpReg ubicado luego del multiplexor de saltos y multiplexor de branch para así poder determinar de que fuente tomar el próximo valor del PC.

En este caso, el selector de este multiplexor es la salida JumpReg del ALUControl, que se igual manera se pondrá en 1 al encontrar la combinación de op y func de jalr. Que estas dos salidas sean independientes también nos permitió implementar jr, que este caso debe poner JumpReg en 1, para saltar a un registro, pero JalrReg en 0 para no guardar el siguiente PC.

- DP pipeline.cpu original.

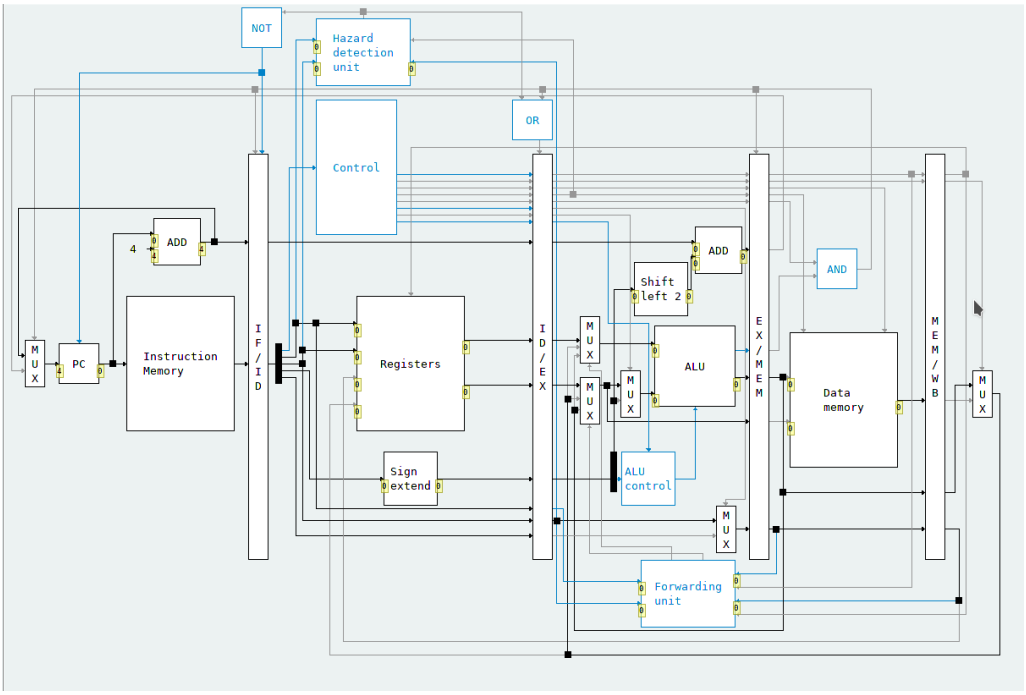

Figura 4: DP pipeline.cpu original

- Implementación la instrucción j en el DP pipeline.cpu.

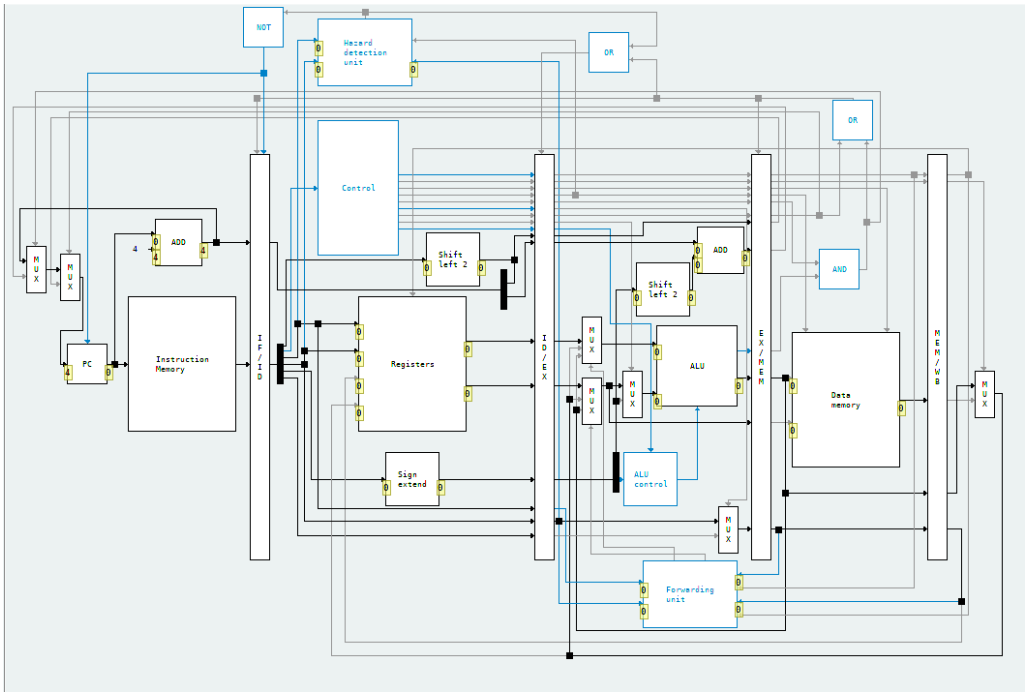

Figura 5: Implementación la instrucción j

Para poder implementar la función j en este DP tuvimos que tener en cuenta que era posible introducir hazards dentro del flujo de ejecución del pipeline, por lo tanto para evitar dichos posibles conflictos se opto por usar de guía la estrategia ya usada e implementada para la ejecución de un branch. Por lo tanto para no agregar posibles errores, por mas que la información necesaria para efectuar el salto ya esta disponible en la segunda etapa (luego del decodificación), se prolonga el salto a la misma etapa en la que se efectuaría un branch (Memory access (MEM)).

Para implementar la instrucción j, se uso como guía el DP monociclo, que ya tiene implementada esta instrucción. Para ello, se tomó de la instrucción el campo target de las instrucciones de tipo j(25-0), los cuales son movidos dos bits a izquierda (ya que los saltos se deben corresponder a direcciones alineadas) y por ultimo se completa la dirección a la cual saltar concatenando estos 28 bits con los 4 bits más significativos del PC. Estos 32 bits de la dirección del salto son dirigidos a la entrada de un multiplexor MuxJump que selecciona entre ellos y la dirección que se calculaba anteriormente. El selector es simplemente una salida desde la unidad de control que indica de la instrucción es de tipo J o no.

Al igual que la instrucción b cuando el salto es tomado, la instrucción j debe limpiar las instrucciones que sean han comenzado a ejecutar el pipeline luego de ella, por lo que mediante se agrega una compuerta OR entre si hay un salto por jump o por branch que será la que flushee los registros pipeline.

- Implementación la instrucción srl en el DP pipeline.cpu.

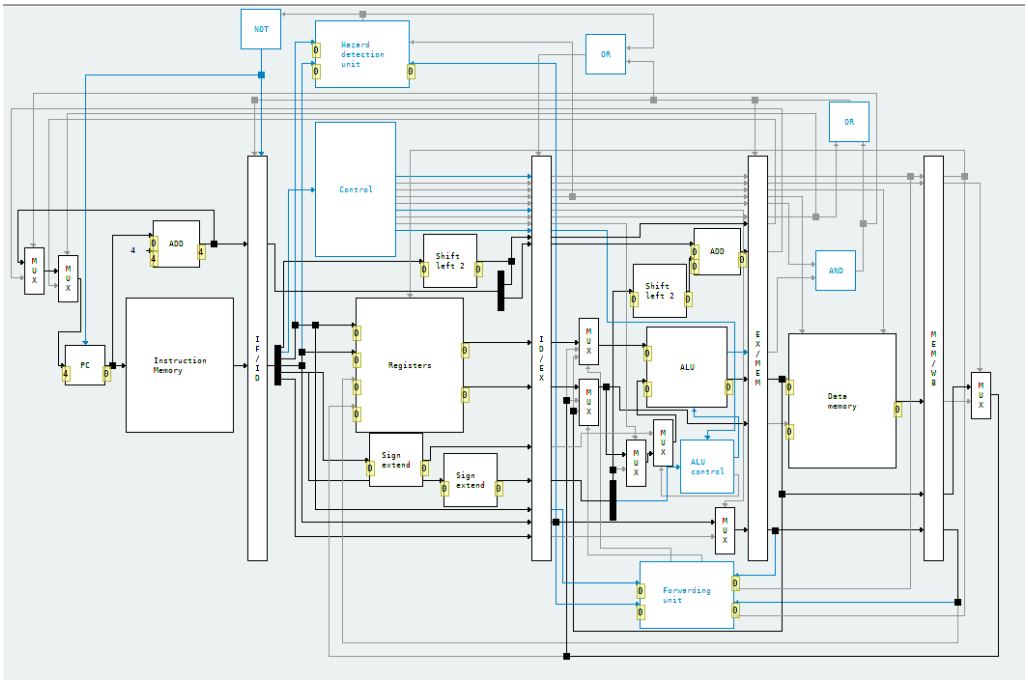

Figura 6: Implementación la instrucción srl

La implementación de srl es muy similar a la implementación de sll para el DP unicycle con diferencia que al ser un DP pipeline nos vemos obligados a usar los pipeline registers. Las modificaciones efectuadas son las mismas que en el caso anterior (sll), con diferencia que la instrucción es tomada desde el registro IF/ID para luego ser extendido el campo shamt y dichos 32 bits son almacenados en el registro ID/EX para ser usados por la ALU en caso de que la ALU Control detecte una instrucción srl. Luego el flujo del pipeline se mantiene igual a la de cualquier instrucción tipo R.

- Implementación la instrucción jalr en el DP pipeline.cpu.

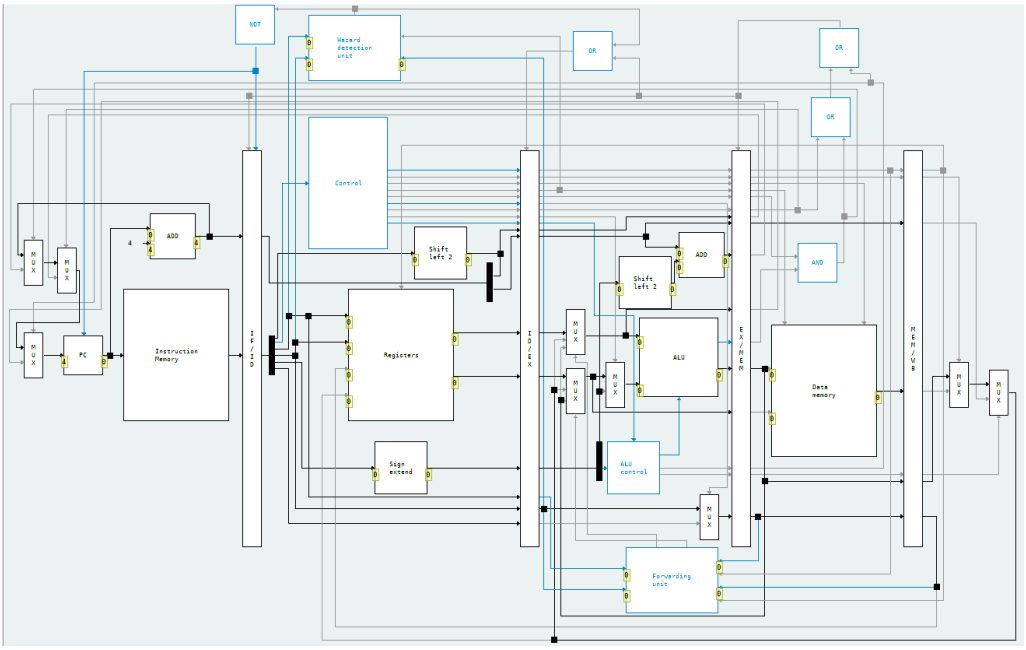

Figura 7: Implementación la instrucción jalr pipeline

Para implementar la función jalr se hizo uso del diseño de la instrucción j descripta anteriormente y de la instrucción jalr del caso unicycle, es decir, se respetaron los tiempos utilizados para un branch, como en el caso de j y para lograr la actualización del registro $ra se hicieron

los mismos cambios usados en el DP unicycle, solo que en este caso se integraron los pipelines registers. Para respectar las etapas del pipeline dicha actualización se realiza en la etapa WB.

Y al igual que en uncycle, se hizo uso de un nuevo multiplexor para elegir la nueva entrada del PC con la nueva opción del cotenido del registro rs. Para la limpieza del pipeline al realizar el salto, se puso otra compuerta OR para verificar si hay salto por jump, branch o jalr (esto se debe a que la instrucción jalr es de tipo r y no de tipo j, por lo que no puede ser considerada como un salto por jump).

3. Instrucciones de compilación

En la elaboración del trabajo practico no se hizo uso de ningún lenguaje de programación, aunque si se modificaron archivos con formato JSON los cuales son usados por DrMIPS para la configuración de los DP, los archivos modificados pueden verse accediendo al siguiente repositorio de github https://github.com/FrancoLiberali/orgacompustp3. En el mismo se encontrará la solución de cada una de la instrucciones pedidas en las ramas ex1, ex2, ex3, ex4 y ex5 respectivamente según el orden de aparición en la consigna.

4. Casos de Prueba

A la hora de evaluar los distintos diseños, dado la dificultad encontrada para hacer el seguimiento paso a paso, optamos por hacer pruebas cortas pero que pongan a prueba por completo a la implementación hecha, es decir, intentamos cubrir todos los posibles puntos flojos o críticos del diseño. Las mismas se encuentran en el repositorio antes mencionado dentro de la rama master.

- ex1

```
#caso trivial, MASDOS,no se tendría que terminar de ejecutar
j MASUNO
MASDOS:
addi $t2, $t2, 2
MASUNO:
addi $t1, $t1, 1
#chequeo que el j no ocurra antes del branch
addi $t1, $t1, -1
beq $t1, $t2,  MASDOS
j SUMA:
addi $t1, $t1, -1 #esta linea no debería ejecutarse

#loop infinito sumador
SUMA:
 addi $t1, $t1, 1
 j SUMA
```

- ex2

```
#caso trivial
addi $t3, $t4, 8
srl $t3, $t3, 2

#chequeamos que pasa cuando el numero del
#campo shamp tiene le bit mas significativo
#en 1, ya que el extensor de signo lo tomara
#como valor negativo
addi $t1, $t2, 0x0000ffff #Nos permitirá ver el value del registro $t1 todo en 1
srl $t1, $t1, 31

#caso con overflow
```

```
srl $t1, $t1, 1
```

- ex3

```
#vemos como no surguen problemas en el pipeline
#al usar al mismo tiempo la ALU con instrucciones
#que requieren que los datos provengan de distintos lugares

#caso trivial
addi $t3, $t4, 8
srl $t3, $t3, 2

#chequeamos que pasa cuando el numero del
#campo shamp tiene le bit mas significativo
#en 1, ya que el extensor de signo lo tomara
#como valor negativo
addi $t1, $t2, 0x0000ffff #Nos permitirá ver el value del registro $t1 todo
srl $t1, $t1, 31

#caso con overflow
srl $t1, $t1, 1
```

- ex4/5

```
#caso trivial, t2 debe terminar en 3
# y ra con la misma dirección que el pc
addi $t1, $t1, SUMA

beq $t2, $t3, SALTO

SALTO:
 jalr $t1
 addi $t2, $t2, 1


SUMA:
 addi $t2, $t2, 1
 jalr $ra
```

5. Conclusión

Luego de realizar los distintos ejercicios propuestos en este trabajo practico pudimos nota que mejorar el rendimiento del procesador no es una tarea sencilla ya que la complejidad encontrada en el DP unicycle es ordenes de magnitud menor que la del DP pipeline, esto se debe, en parte, a que en este ultimo caso, no solo se complejiza el hardware, sino que aumenta en cantidad cada vez que queremos agregar alguna funcionalidad. Además de que hay que tener en cuenta los posibles hazards que se pueden introducir en las distintas etapas del pipeline. También destacamos que los DP diseñados a lo largo del trabajo practico se encuentran muy lejos de un DP actual los cuales soportan múltiples operaciones.

Por otra parte, también es importante destacar que pudimos observar la efectividad del uso del pipeline y como el mismo logra ejecutar mas instrucciones en un mismo periodo de tiempo (comparado en este caso con el DP unicycle). Dicha mejora se logra apreciar luego de ejecutar al menos una cantidad de instrucciones cercanas al doble de la cantidad de etapas del pipeline.

6. Enunciado

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.