

Trabajo Práctico 3

Data Path

[8637/6620] Organización de Computadoras
Curso 2
Segundo Cuatrimestre de 2020

| Alumno: | Padrón |
|------------------------|--------|
| BIANCARDI, Julián | 103945 |
| CZOP, Santiago Nicolás | 104057 |

Índice

| | |
|--------------------------------|-----------|
| 1. Introducción | 2 |
| 2. Desarrollo | 2 |
| 2.1. Unicycle | 2 |
| 2.2. Pipeline | 7 |
| 3. Ejecución de Pruebas | 10 |
| 4. Código Fuente | 11 |
| 5. Conclusiones | 11 |

1. Introducción

En el siguiente informe se detallarán todos los pasos seguidos para la resolución del trabajo práctico 3 de la materia Organización de Computadoras (66.20). El mismo consiste en 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. Desarrollo

En la siguiente sección se detallarán los pasos realizados para las implementaciones de las instrucciones propuestas por el enunciado.

Para el desarrollo del trabajo primero se desarrollaron las implementaciones en los DP uniciclos, para luego saber fácilmente los cambios a realizar en los DP pipeline. Para el caso de DP unicolor optamos por realizarlo en un unico DP. En cambio, para el caso de los DP pipeline optamos por realizarlo en DP separados.

Adicionalmente, se implementaron códigos de prueba genéricos que buscan probar estas implementaciones (ver la sección 3).

2.1. Unicycle

El simulador DrMIPS [1] nos brinda una serie de data paths uniciclos de los cuales usaremos el "unicycle.cpu". Este mismo es el DP unicolor por defecto y no cuenta con el soporte tanto para la instrucción jr ni para la instrucción jalr pedidas por el enunciado. A continuación se muestra una imagen del mismo:

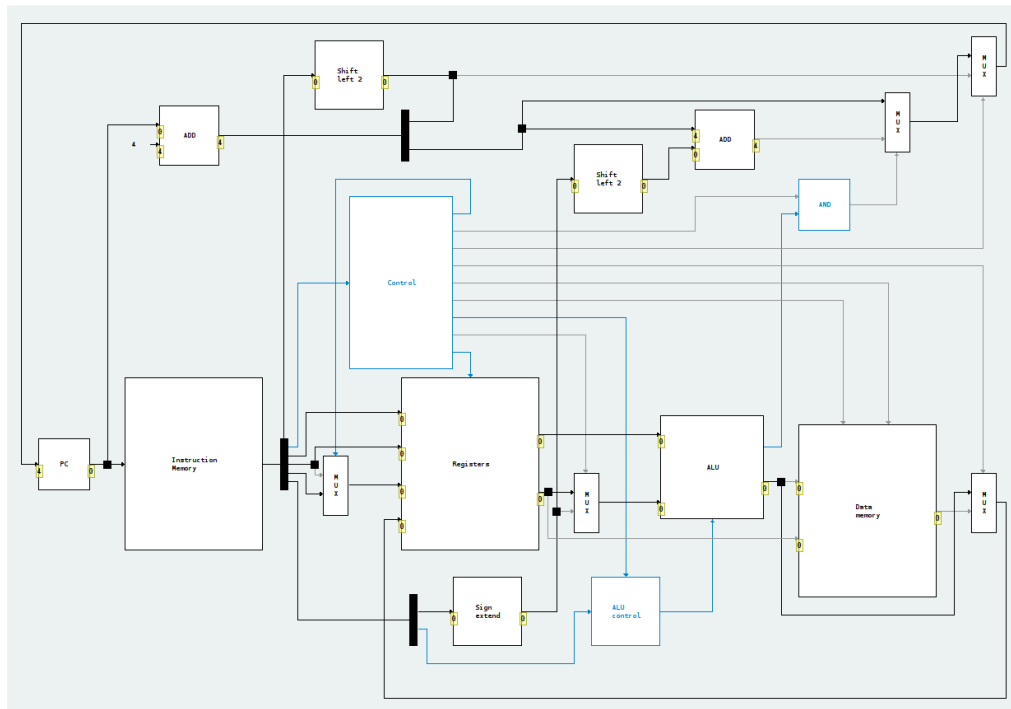


Figura 1: DP unicolor por defecto

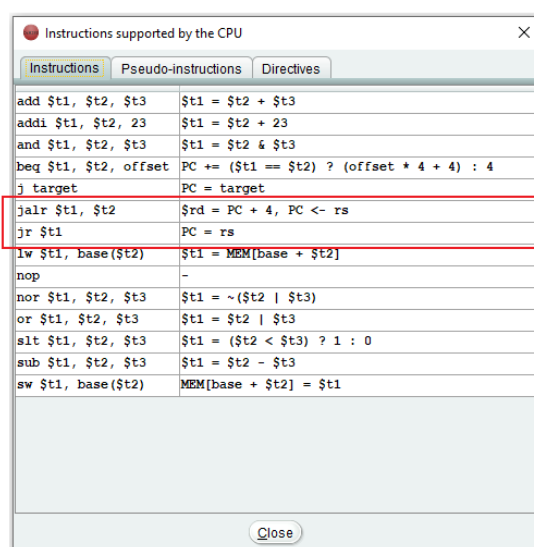
2.1.1. Instrucción jr, jalr

Para el caso del DP unicolor se decidió implementar ambas instrucciones propuestas por el enunciado en un único DP.

Primero debemos agregar las instrucciones al archivo .set que esté usando nuestro unicycle.cpu", en nuestro caso usaremos el que viene por default "default.set". Para esto agregaremos las siguientes líneas de código en la sección instructions:

```
1 "jr": {
2   "type": "R",
3   "args": ["reg"],
4   "fields": {"op": 1, "rs": "#1", "rt": 0, "rd": 0, "shamt": 0, "func": 8},
5   "desc": "PC = rs"
6 },
7
8 "jalr": {
9   "type": "R",
10  "args": ["reg", "reg"],
11  "fields": {"op": 9, "rs": "#2", "rt": 0, "rd": "#1", "shamt": 0, "func": 9},
12  "desc": "$rd = PC + 4, PC <- rs"
13 },
```

Con esto ya tendremos soporte para la detección de las instrucciones a ser utilizadas en los programas de assembly pero aún no tendremos su funcionamiento, es decir que debemos agregar los cableados necesarios en el DP para que se ejecuten correctamente.



| Instructions supported by the CPU | |
|-----------------------------------|---|
| Instructions | Pseudo-instructions |
| add \$t1, \$t2, \$t3 | \$t1 = \$t2 + \$t3 |
| addi \$t1, \$t2, 23 | \$t1 = \$t2 + 23 |
| and \$t1, \$t2, \$t3 | \$t1 = \$t2 & \$t3 |
| beq \$t1, \$t2, offset | PC += (\$t1 == \$t2) ? (offset * 4 + 4) : 4 |
| j target | PC = target |
| jalr \$t1, \$t2 | \$rd = PC + 4, PC <- rs |
| jr \$t1 | PC = rs |
| lw \$t1, base(\$t2) | \$t1 = MEM[base + \$t2] |
| nop | - |
| nor \$t1, \$t2, \$t3 | \$t1 = ~(\$t2 \$t3) |
| or \$t1, \$t2, \$t3 | \$t1 = \$t2 \$t3 |
| slt \$t1, \$t2, \$t3 | \$t1 = (\$t2 < \$t3) ? 1 : 0 |
| sub \$t1, \$t2, \$t3 | \$t1 = \$t2 - \$t3 |
| sw \$t1, base(\$t2) | MEM[base + \$t2] = \$t1 |

Figura 2: Instrucciones para DP unicolor

Para la implementación de ambas instrucciones reutilizamos el multiplexor MuxJump y Mux-Mem que venían en el DP unicolor por defecto.

Para el caso del multiplexor MuxJump basta con agregar una nueva entrada proveniente del valor del registro rs (salida RegData1 del RegisterFile) para la implementación de ambas instrucciones ya que requieren colocar el PC con el valor almacenado en el registro rs. Entonces nos quedaría algo como lo siguiente:

| | |
|----------|---------------|
| Jump = 0 | → PC = PC + 4 |
| Jump = 1 | → PC = target |
| Jump = 2 | → PC = rs |

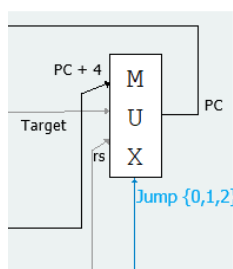


Figura 3: Modificación de MuxJump

Para el caso del multiplexor MuxMem basta con agregar una nueva entrada proveniente del PC + 4. Este agregado es únicamente para la instrucción jalr debido a que $rd = PC + 4$ (seleccionado el registro rd para escribir mediante la señal de control $RegDst = 1$), quedando algo como:

| | |
|--------------|-----------------------|
| MemToReg = 0 | → WriteData = ALU |
| MemToReg = 1 | → WriteData = Memoria |
| MemToReg = 2 | → WriteData = PC + 4 |

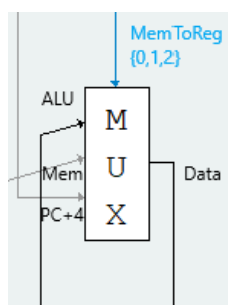


Figura 4: Modificación de MuxMem

Con la Unidad de Control indicaremos, mediante las señales de salida, que se deberá ejecutar la instrucción jr cuando el campo op de la instrucción sea igual a 1. Agregaremos la siguiente línea de código en la sección control:

```
1 "1": {"Jump": 2, "RegWrite": 0, "MemRead": 0},
```

Con la señal Jump = 2 indicamos que el PC tomará el valor proveniente del registro rs. Aquellas señales que no se les define un estado lógico significa que no importa su valor para la ejecución de la instrucción, por lo tanto toman el estado lógico X.

Finalizado estos dos pasos deberemos agregar los cableados necesarios en el DP para que la instrucción se ejecute correctamente:

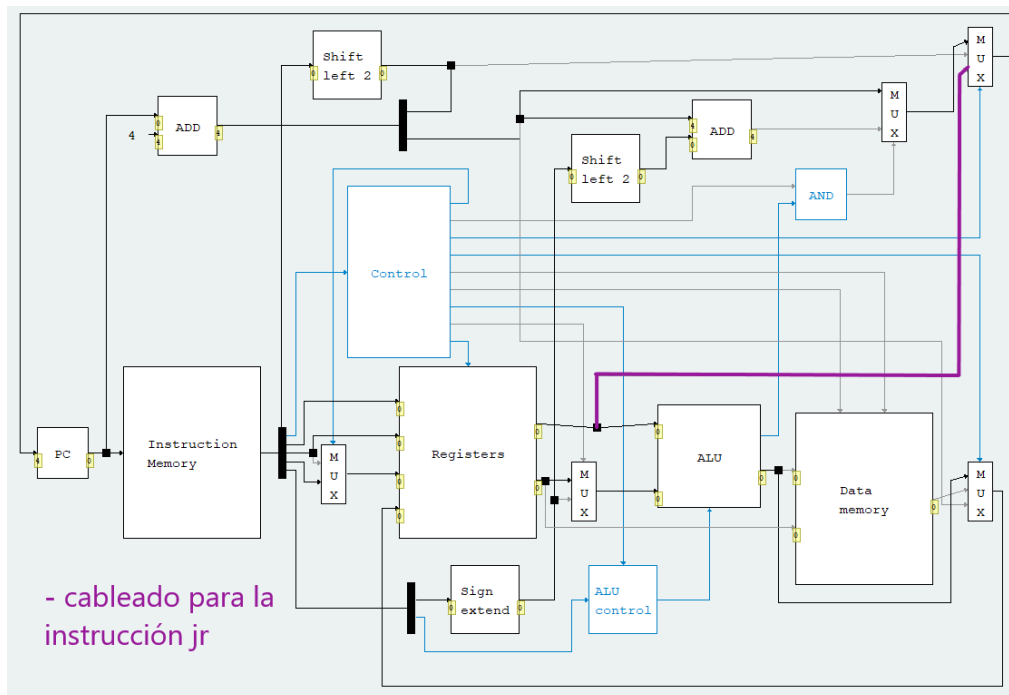


Figura 5: Cableado para la instrucción jr

Por el contrario, para la instrucción jalr la señales de la unidad de control son las siguientes: .

```
1 "9": {"Jump": 2, "RegDst": 1, "RegWrite": 1, "MemWrite": 0, "MemToReg": 2},
```

Con la señal $\text{Jump} = 2$ indicamos que el PC tomará el valor proveniente del registro rs. Con las señal $\text{RegDst} = 1$ y $\text{MemToReg} = 2$ indicamos que el registro rd tomará el valor del $\text{PC} + 4$. Los cableados necesarios para que la instrucción se ejecute correctamente son entonces:

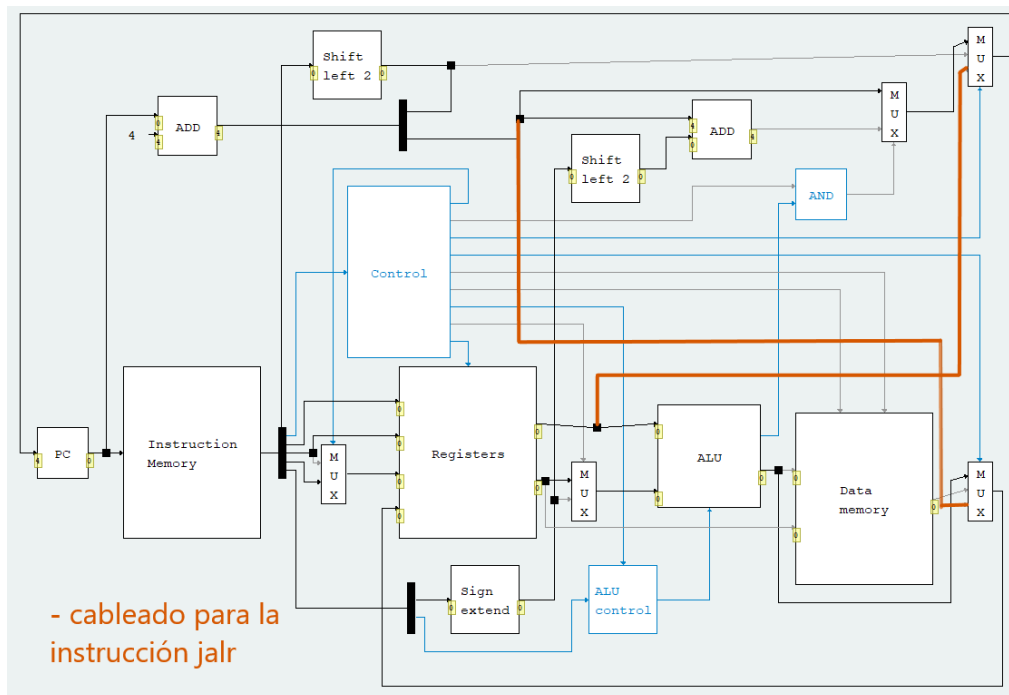


Figura 6: Cableado para la instrucción jalr

El resultado final resulta ser un DP uniciclo que soporta las instrucciones jr y jalr más las instrucciones por defecto:

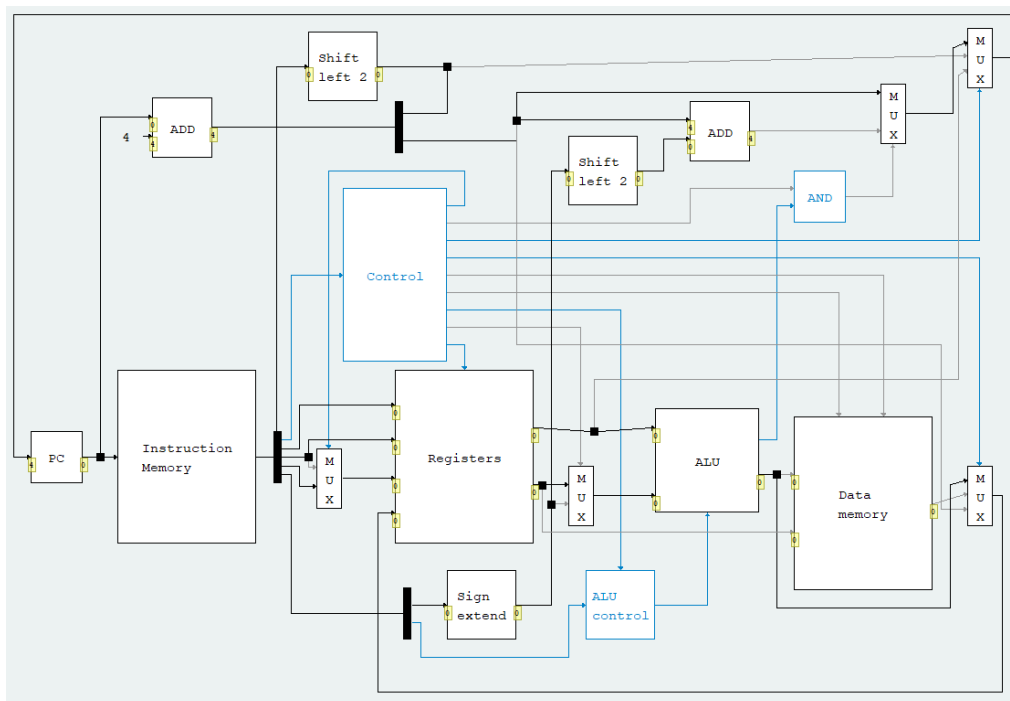


Figura 7: DP uniciclo final

2.2. Pipeline

El simulador DrMIPS [1] nos brinda una serie de data paths pipelines de los cuales usaremos el "pipeline.cpu". Este mismo es el DP pipeline por defecto, el cual no cuenta con el soporte tanto para la instrucción j,jr ni para la instrucción jalr.

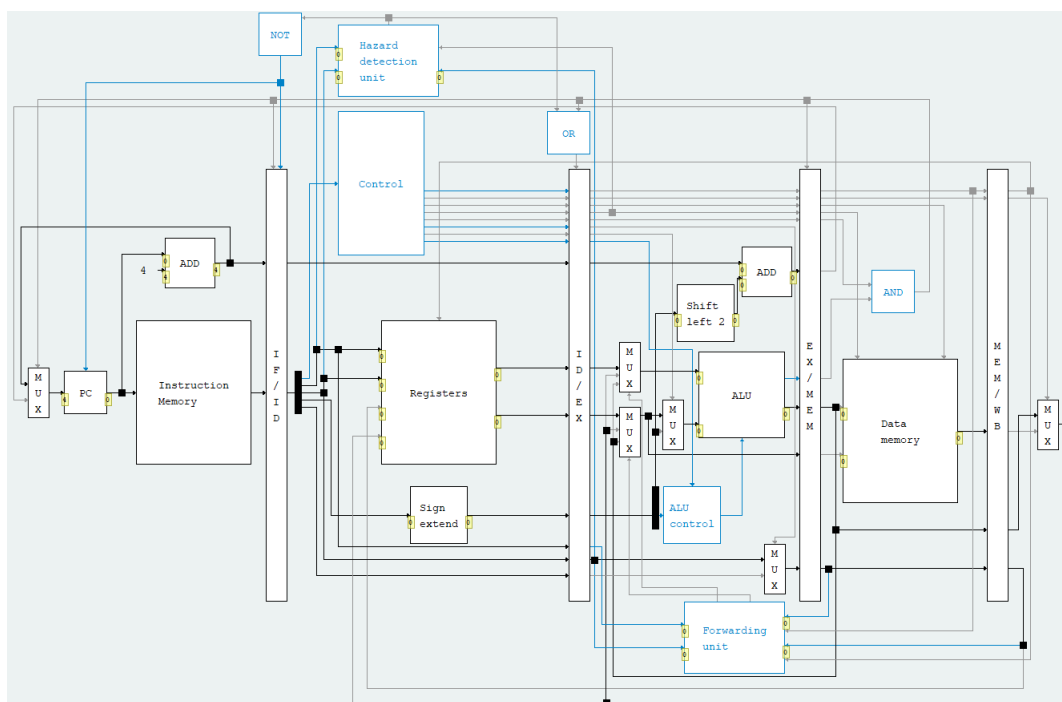


Figura 8: DP pipeline por defecto

Se optó por agregar las implementaciones de las instrucciones propuestas por el enunciado por separado. Así contamos con los siguientes archivos:

- pipeline-j.cpu con default-pipeline-j.set
- pipeline-jr.cpu con default-pipeline-jr.set
- pipeline-jalr.cpu con default-pipeline-jalr.set

2.2.1. Instrucción j

Primero añadimos la instrucción para que pueda ser utilizada por código assembly. Agregamos la siguiente línea de código en el archivo default-pipeline-j.set en la sección instructions (como veníamos haciendo para el caso de uniclo):

```

1 "j": {
2   "type": "J",
3   "args": ["target"],
4   "fields": {"op": 2, "target": "#1"},
5   "desc": "PC = target"
6 },

```

Y la siguiente línea en la sección control:

```

1 "2": {"Jump": 1},

```


Para incorporar la implementación de la instrucción al DP pipeline primero agregamos el componente ShiftLeft2 el cual mediante un desplazamiento a izquierda (o multiplicación lógica) deja, a los 22 bits que componen a la dirección de salto, en un estado válido, es decir, múltiplo de 4. Esto se debe a que el PC se incrementa de a 4 bytes, siempre contiene 0 en sus 2 bits menos significativos.

Una vez realizado esto debemos acordarnos de aplicar las correspondientes nop instruction (No operar) via hardware o software. En nuestro caso decidimos implementarlas via hardware aplicando el correcto flush a los registros de las etapas posteriores.

Detectando la señal Jump = 1 desde la unidad de control, aplicamos el flush al registro de la etapa ID y garantizamos así el correcto funcionamiento de la instrucción j.

Con estos pasos nos quedaría el siguiente DP pipeline que soporta la instrucción j:

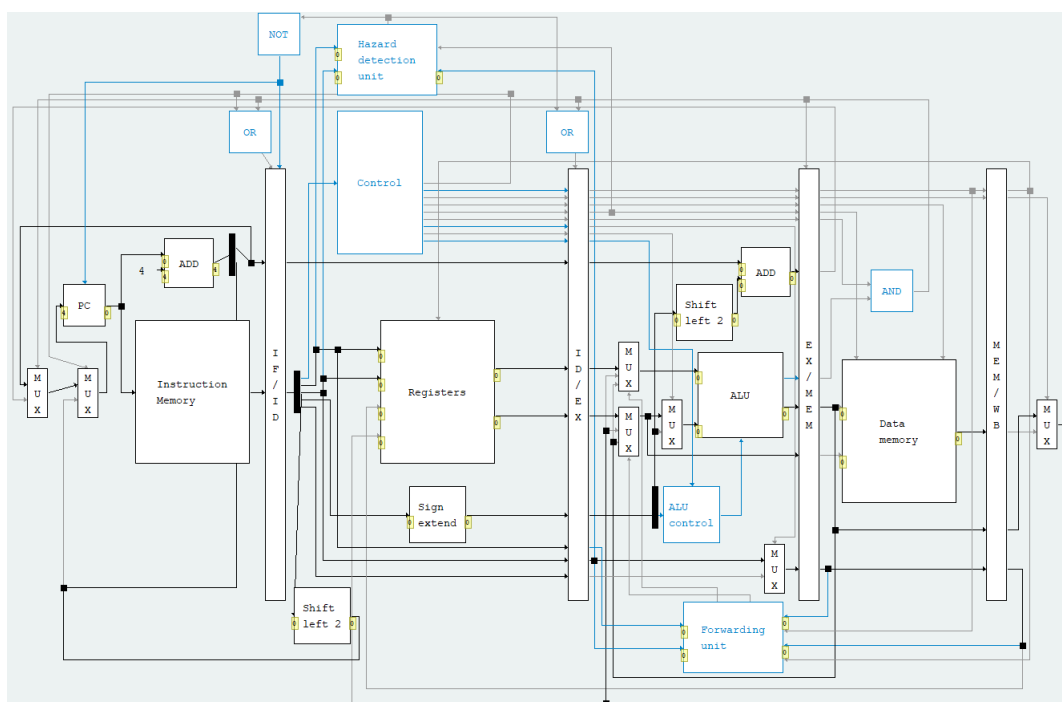


Figura 9: DP pipeline con instrucción j

2.2.2. Instrucción jr

Nuevamente realizamos los mismos pasos que en el caso anterior. Añadimos la instrucción para que pueda ser utilizada por código assembly incorporando la siguiente línea de código en el archivo default-pipeline-jr.set en la sección instructions:

```
1 "jr": {
2   "type": "R",
3   "args": ["reg"],
4   "fields": {"op": 1, "rs": "#1", "rt": 0, "rd": 0, "shamt": 0, "func": 8},
5   "desc": "PC = rs"
6 },
```

Y la siguiente línea en la sección control:

```
1 "1": {"Jump": 1, "RegWrite": 0, "MemRead": 0},
```

Para incorporar la instrucción al DP pipeline y que funcione correctamente debemos usar con cuidado los valores correspondientes en las etapas correctas para evitar así los posibles hazards. En el caso de la instrucción jr debemos usar el valor del registro rs y colocarlo en el PC, es decir, $PC \leftarrow rs$.

Para obtener el valor correcto del registro rs lo haremos desde la etapa de ejecución EX, más en específico desde el MuxForwardA el cual evita los posibles hazards de datos. Además de este agregado debemos parar el pipeline para evitar los hazards que puedan producirse de una instrucción y sus siguientes. Agregaremos instrucciones nop (No operar) ya sea desde hardware o de software, en nuestro caso lo haremos desde hardware incorporando dos nuevas compuertas lógicas OR activando los correspondientes Flush de los registros de las etapas anteriores IF, ID a la de ejecución EX. Para esto detectamos la señal $\text{Jump} = 1$ desde la unidad de control y aplicamos los flush antes mencionados.

Con estos pasos nos quedaría el siguiente DP pipeline que soporta la instrucción jr:

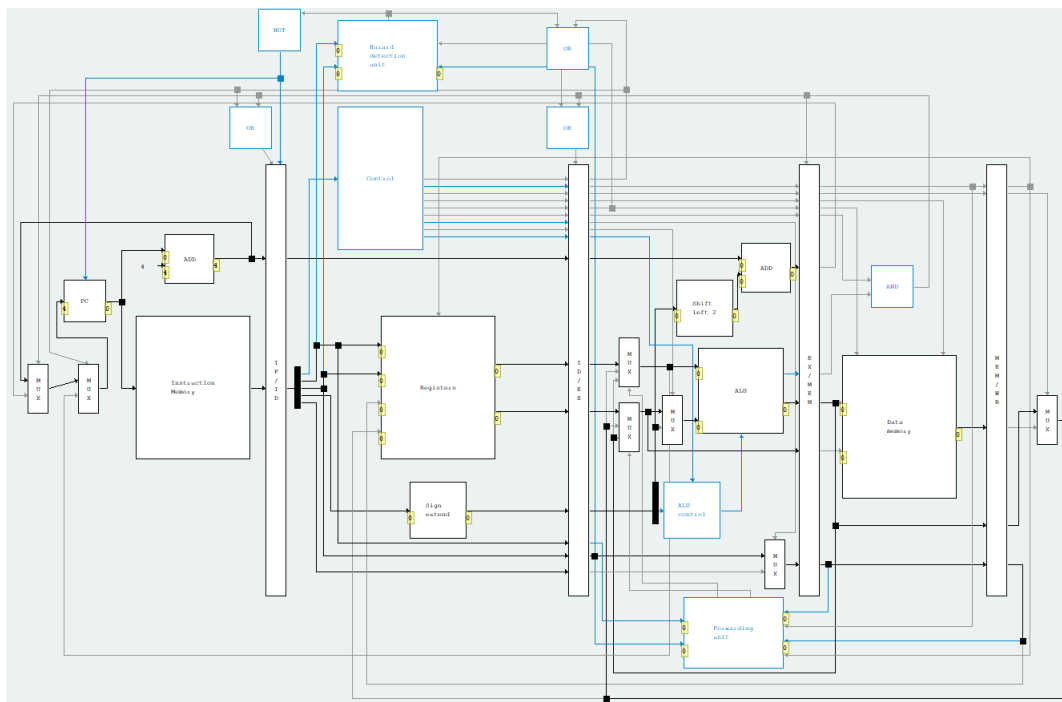


Figura 10: DP pipeline con instrucción jr

2.2.3. Instrucción jalr

Añadimos, como en los casos anteriores, las siguientes líneas de código en el archivo default-pipeline-jalr.set:

```
1 "jalr": {
2   "type": "R",
3   "args": ["reg", "reg"],
4   "fields": {"op": 9, "rs": "#2", "rt": 0, "rd": "#1", "shamt": 0, "func": 9},
5   "desc": "PC = rs, rd <- PC + 4"
6 },
7
8
9 "9": {"Jump": 1, "RegDst": 1, "RegWrite": 1, "MemRead": 0, "MemToReg": 2},
```

Con la señal $\text{MemToReg} = 2$ indicaremos que el valor a escribir en el registro indicado por la señal RegDst sea proveniente del $\text{PC} + 4$. Esto se debe a que la instrucción jalr debe realizar la siguiente operación: $\text{rd} \leftarrow \text{PC} + 4$.

Para incorporar la instrucción al DP pipeline vemos que la instrucción jalr es similar a la instrucción jr con el agregado de $\text{rd} \leftarrow \text{PC} + 4$. Sabiendo esto procedemos a realizar los mismos cableados para la implementación de $\text{PC} \leftarrow \text{rs}$. En cambio, para la implementación de $\text{rd} \leftarrow \text{PC} + 4$ basta con agregar una nueva entrada al MuxMem el cual determina el valor que será escrito en el registro una vez finalizada la ejecución de la instrucción.

Con estos pasos nos quedaría el siguiente DP pipeline que soporta la instrucción jalr:

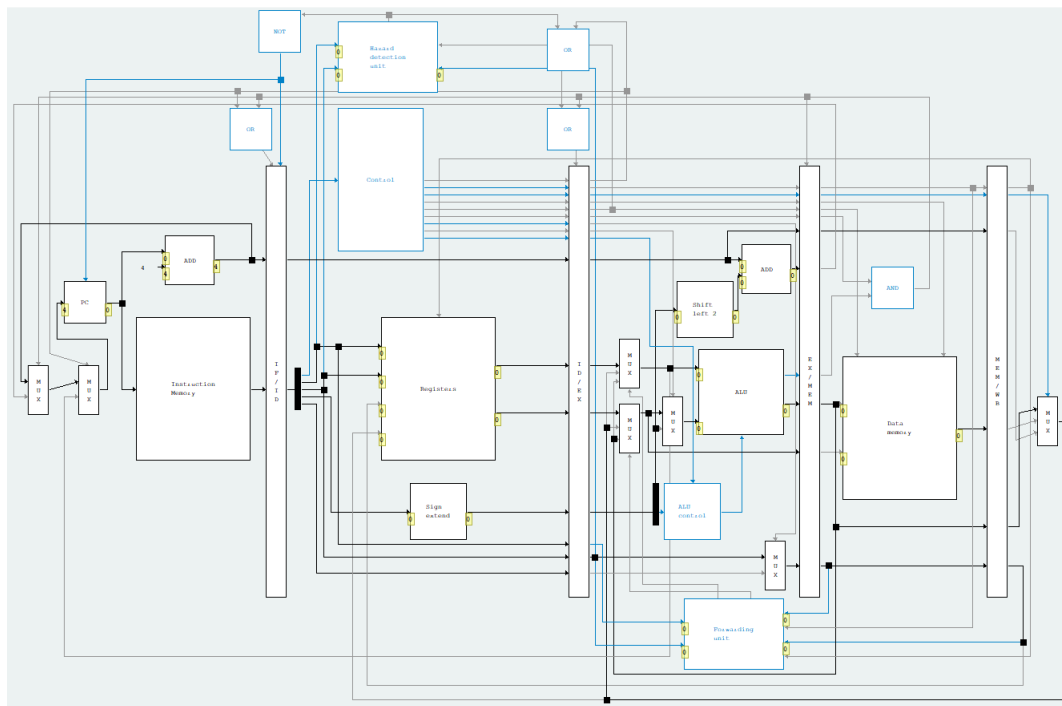


Figura 11: DP pipeline con instrucción jalr

3. Ejecución de Pruebas

En todo proyecto resulta necesario realizar pruebas que permitan comprobar el funcionamiento correcto el producto, así como pruebas que evalúen el desenvolvimiento en casos bordes tal que puedan descubrirse potenciales errores. Para lograr esto se escribió código assembly que trabaja con las instrucciones implementadas así como con otra variedad de instrucciones para intentar cubrir una amplia variedad de casos. En total se cuenta con 20 pruebas, que hacen uso de las siguientes instrucciones:

- addi
- add
- and
- beq
- j
- jr
- jalr
- lw
- neg
- nop
- nor

- or
- subi
- sw

Durante el armado y ejecución de las pruebas se logró descubrir un hazard. El siguiente ocurre en situaciones como la siguiente:

```
addi $t0, $t0, 20
sw    $t0, 0($t0)
jr    $t0
```

En este caso, cualquier instrucción de salto que trabaje con el registro t0 fallará. No es el caso si la instrucción sw guarda otro registro. Se logró replicar el problema en el pipeline por default de DrMIPS a partir de la instrucción beq, indicándonos que el problema no fue causado por nuestra implementación. En este caso, para solucionar el problema, es necesario agregar una instrucción de software *nop* posterior a *sw*.

Finalmente, se logró concluir que las pruebas demuestran el correcto funcionamiento de las distintas implementaciones realizadas, dado que no se presentaron más anomalías ya sea en memoria, en registros o en el flujo de las pruebas. Para replicar el análisis realizado a partir de las pruebas, se puede recurrir al repositorio del proyecto y abrirlas manualmente en DrMIPS, con el DataPath que les corresponda.

4. Código Fuente

Se puede observar el código implementado en la siguiente dirección, en la carpeta TP3.

<https://github.com/JulianBiancardi/Organizacion-de-Computadoras-66.20>

El código fuente del presente informe se encuentra disponible para lectura en:

<https://www.overleaf.com/read/cjdvgvmmwhbh>

5. Conclusiones

En conclusión, se considera que se ha logrado desarrollar una solución completa que cumple con todas las especificaciones indicadas en el enunciado. A lo largo del proceso de diseño, programación y testeo se incorporó mucho conocimiento tanto sobre DP unicycle como el DP pipeline al adentrarse en las implementaciones de los mismos. El amplio proceso de testeo permitió comprobar experimentalmente el correcto funcionamiento de nuestros DP y descubrir el hazard en el DP de pipeline default.

Referencias

- [1] DrMIPS [online] Disponible en: <https://bitbucket.org/brunonova/drmips/wiki/Home>.
- [2] 66.20, O., 2021. Camino de Datos Monociclo y Multiciclo Cableado. [online] Available at: <https://campus.fi.uba.ar/pluginfile.php/422980/mod_resource/content/1/caminos%20de%20datos%20monociclo%20y%20multiciclo.pdf>[Accessed February 2021].
- [3] 66.20, O., 2021. Pipeline Parte 1. [online] Available at: <https://campus.fi.uba.ar/pluginfile.php/422982/mod_resource/content/1/pipeline%20parte%202.pdf>[Accessed February 2021]
- [4] 66.20, O., 2021. Pipeline Parte 2. [online] Available at:<https://campus.fi.uba.ar/pluginfile.php/422981/mod_resource/content/1/pipeline%20parte%201.pdf>[Accessed February 2021]