

U.B.A. FACULTAD DE INGENIERÍA

DEPARTAMENTO DE ELECTRÓNICA
ORGANIZACIÓN DE COMPUTADORAS 86.37/66.20
1ER CUATRIMESTRE DE 2021

Trabajo práctico N°3: Datapath y pipeline

Apellido y Nombre	Padrón	Correo electrónico
Castillo, Julio	80.039	jecastillo@fi.uba.ar
La Penna, Mariano	98.432	mlapenna79@hotmail.com
Minino, Nahuel	99.599	nminino@fi.uba.ar

Grupo N°3

 Github:

<https://github.com/TPs-Orga-De-Compus/TP3>

Fecha de Entrega : 22/07/2021

Índice

1. Introducción	2
2. Enunciado	2
3. Instrucciones implementadas	2
4. andi rs, rt, imm	3
4.1. Modificaciones en archivos .set	3
4.1.1. Instrucción	3
4.1.2. Unidad de control	3
4.1.3. ALU	3
4.2. Caso de prueba	4
4.3. Diagramas .cpu	4
4.3.1. unicycle.cpu	4
4.3.2. pipeline.cpu	5
5. j rs, rt	6
5.1. Modificaciones en archivos .set	6
5.1.1. Tipo de instrucción	6
5.1.2. Instrucción	6
5.1.3. Unidad de control	6
5.2. Modificaciones en archivos .cpu	7
5.2.1. unicycle.cpu	7
5.2.2. pipeline.cpu	7
5.3. Caso de prueba	8
5.4. Diagramas .cpu	8
5.4.1. unicycle.cpu	8
5.4.2. pipeline.cpu	9
6. lw rs, rd, rt	12
6.1. Modificaciones en archivos .set	12
6.1.1. Instrucción	12
6.1.2. Unidad de control	12
6.2. Caso de prueba	13
6.3. Diagrama .cpu	13
7. Problemas encontrados	14
8. Conclusiones	14
9. Referencias	14

1. Introducción

El objetivo de este trabajo práctico es implementar varias instrucciones en un datapath monociclo y otro pipeline. Esto se hará mediante un software de simulación DrMIPS que simula los componentes principales de arquitectura de CPUs Mips, la conexión entre ellos, el almacenamiento en memoria principal y la ejecución de instrucciones. De esta forma nos interiorizaremos más con dichos datapaths.

2. Enunciado

Los detalles de la consigna están descriptos en el archivo “enunciado.pdf”.

3. Instrucciones implementadas

Se describirá en detalle cada una de las instrucciones implementadas en un ítem aparte, con su respectivo código, diagramas cuando sea necesario, detalles e instrucciones de prueba

- **andi rs, rt, imm** en los datapaths unicycle y pipeline.
- **j rs, rt** en los datapaths unicycle y pipeline.
- **lw rd, rs, rt** en el datapath pipeline.

4. `andi` `rs`, `rt`, `imm`

En los archivos de instrucciones **default.set** y **default-no-jump.set** (que son utilizados por el `cpu` `unicycle` y `pipeline` respectivamente) se hicieron las mismas modificaciones para implementar esta nueva instrucción. Los archivos `.cpu` no necesitaron ser alterados.

4.1. Modificaciones en archivos `.set`

4.1.1. Instrucción

En la sección **instructions** se agregó **andi**.

```
"andi": {"type": "I", "args": ["reg", "reg", "int"],
         "fields": {"op": 9, "rs": "#2", "rt": "#1", "imm": "#3"},
         "desc": "$t1 = $t2 & 24"}
```

El tipo es **I** (immediate) ya que sus parámetros son dos registros (5 bits cada uno) y un campo inmediato (16 bits). Los 6 bits restantes que son los de mayor peso representan la operación. El número de operación es 9, una nueva, explicada más adelante. El primer parámetro es el registro de destino y los otros dos representan los datos fuente.

4.1.2. Unidad de control

En la sección **control** agregamos una nueva operación para la unidad de control, arbitrariamente numerada 9.

```
"9": {"RegDst": 0, "RegWrite": 1, "ALUOp": 3, "ALUSrc": 1, "MemToReg": 0}
```

- `RegDst` es 0 porque los bits que representan el número de registro de destino son del 16 al 20 de la instrucción.
- `RegWrite` 1 ya que el resultado de la ALU se escribirá en registro.
- `AluOp` 3 es una nueva operación.
- `ALUSrc` 1 es porque el segundo operando de la ALU debe ser el valor inmediato, y no la segunda salida del banco de registros.
- `MemToReg` 0 ya que ningún registro se escribirá desde memoria principal.

4.1.3. ALU

En la sección **alu** se agrega una nueva llamada a la operación **and**, que es la número 0, pero sin valor en el campo `func`.

```
{"aluop": 3, "out": {"Operation": 0}}
```

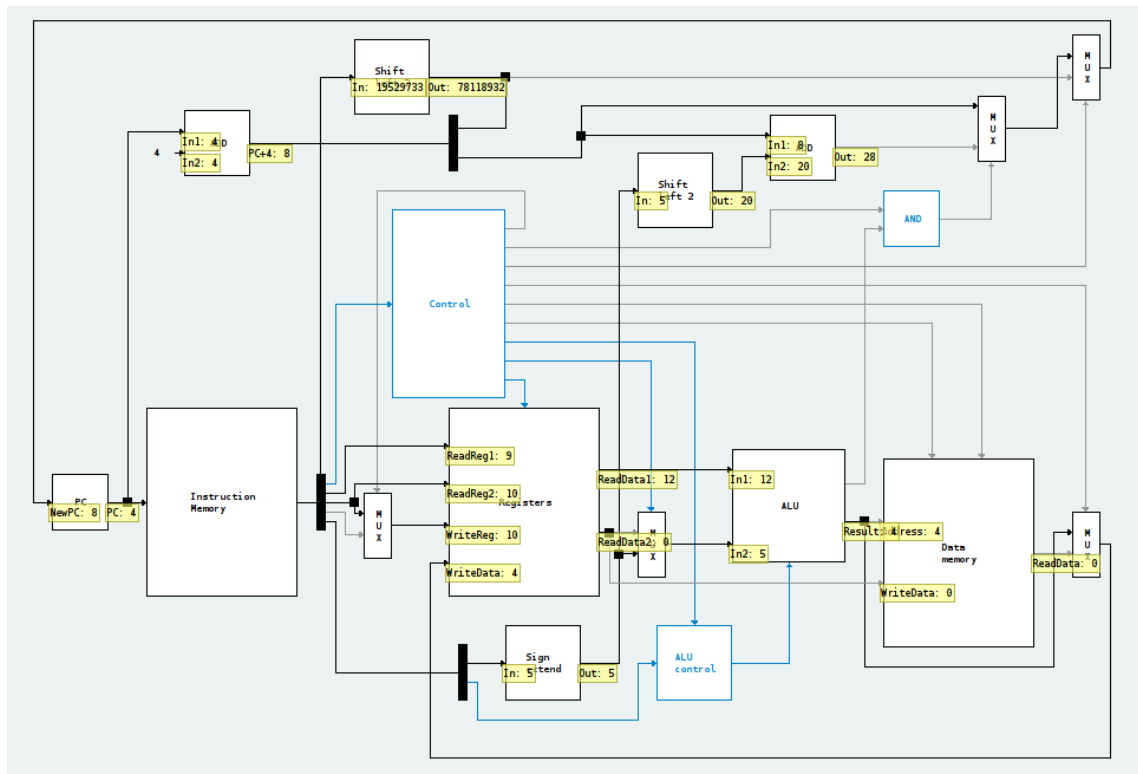
4.2. Caso de prueba

```
addi $t1, $zero, 12
andi $t2, $t1, 5
sw $t2, 4($zero)
```

Se carga el valor 12 en el registro temporal 1, luego a ese valor 12 se le hace un AND binario con el valor 5. Escrito en binario esto es 1100 & 0101, lo que da 0100 que es 4 en decimal. Posteriormente se guardó en memoria dicho valor.

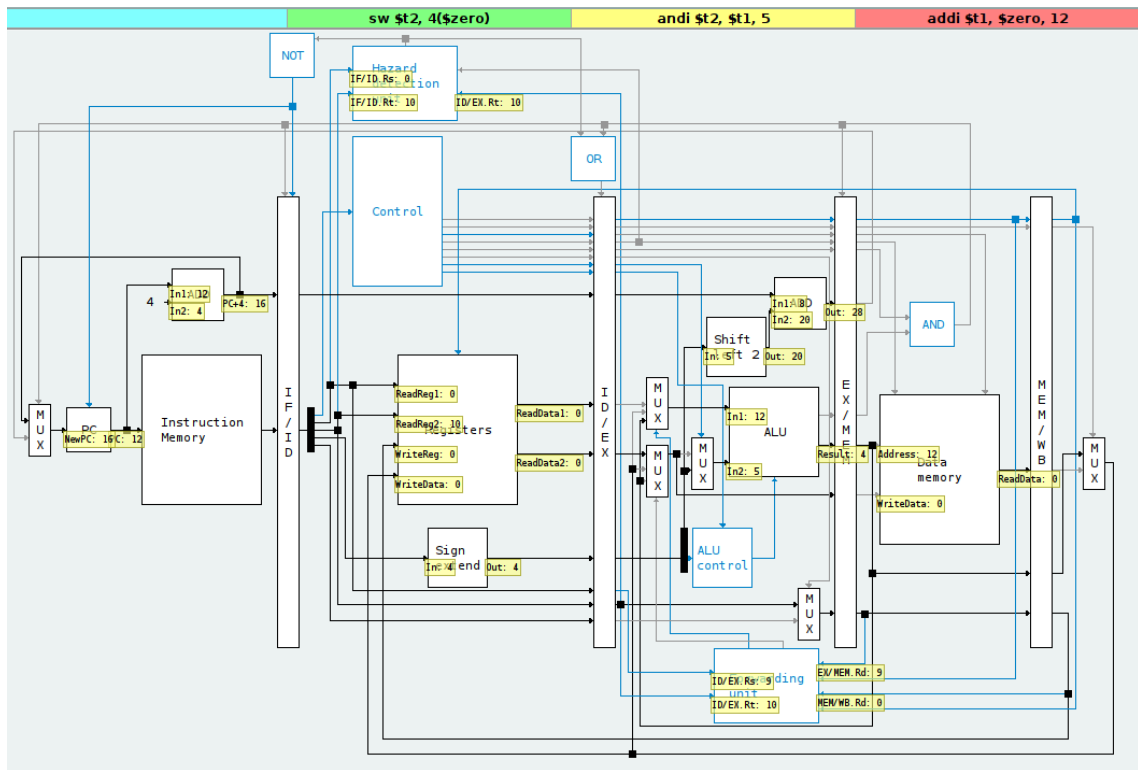
4.3. Diagramas .cpu

4.3.1. unicycle.cpu



Aquí se puede ver el recorrido de los valores durante la ejecución del **andi** en si, en unicycle.cpu.

4.3.2. pipeline.cpu



En este caso el recorrido de valores durante la ejecución del código en pipeline.cpu. Este es el cuarto ciclo de ejecución del programa, con la instrucción `andi` en la etapa de ejecución. Se ve como se soluciona el riesgo RAW respecto al registro `$t1` ya que el registro interetapa EX / MEM le envía el valor 12 a la primer entrada de la ALU, ya que en el ciclo anterior (al igual que en este) `$t1` todavía no contiene el valor 12 por lo que no se pudo leer desde allí.

5. j rs, rt

En los dos archivos .set se hicieron las mismas modificaciones para implementar esta instrucción. En cambio, para los archivos .cpu unicycle y pipeline, tuvimos que encarar el problema de formas distintas.

5.1. Modificaciones en archivos .set

5.1.1. Tipo de instrucción

Como necesitábamos que la instrucción contuviera los bits de los registros operandos rs y rt, creamos una variante del tipo de instrucción J en la sección **types** llamada JR.

```
"JR": [{ "id": "op", "size": 6}, {"id": "rs", "size": 5},
        {"id": "rt", "size": 5}, {"id": "nothing", "size": 16}]
```

Cabe destacar que también se podría realizar usando el tipo de instrucciones I sin hacer uso del valor inmediato.

5.1.2. Instrucción

En la sección **instructions** se agregó **jr** (no usamos j como nombre porque ya existía en el archivo default.set).

```
"jr": { "type": "JR", "args": ["reg", "reg"],
        "fields": {"op": 32, "rs": "#2", "rt": "#1", "nothing": 0},
        "desc": "PC = $t1 + $t2"}
```

En el tipo **JR** almacenamos los bits que seleccionan el registro rs del bit 25 al 21 y los que seleccionan al registro rt del bit 20 al 16. Los 16 bits menos significativos los dejamos en 0 ya que nunca los vamos a usar. Los 6 bits más significativos representan la operación. El número de operación es 32, una nueva, explicada a continuación.

5.1.3. Unidad de control

En la sección **control** agregamos una nueva operación para la unidad de control, numerada 32 debido a que, como se verá más adelante, se usará el bit 31 para distinguir entre saltos a la dirección indicada en la instrucción o a la dirección obtenida como resultado de la ALU.

```
"32": {"RegWrite": 0, "ALUOp": 0, "ALUSrc": 0, "MemToReg": 0, "Jump": 1}
```

- RegWrite es 0 ya que el resultado de la ALU no se escribirá a un registro.
- AluOp es 0 ya que vamos a usar la operación add desde una instrucción sin campo func.

- ALUSrc es 0 porque el segundo operando de la ALU va a ser la segunda salida del banco de registros y no un valor inmediato.
- MemToReg es 0 ya que ningún registro se escribirá desde memoria principal.
- Jump es 1 ya que seleccionará la dirección de salto como próxima dirección apuntada por el PC en lugar de la dirección siguiente a la actual (PC+4).

5.2. Modificaciones en archivos .cpu

5.2.1. unicycle.cpu

Para el caso del datapath uniciclo, se agregó un multiplexor que recibe como entrada “1” los 26 bits menos significativos del resultado de la ALU y en la entrada “0” los 26 bits menos significativos de la instrucción. A la salida del multiplexor se le aplica un shift a izquierda de 2 bits (se multiplica la dirección por 4) y se concatenan los 4 bits más significativos de la instrucción. El resultado de esto va al multiplexor del jump que decide si sumar 4 al PC actual o elegir la dirección indicada por la instrucción de jump como nuevo PC (dependiendo de si la unidad de control marca el bit de jump con 1 o 0).

El selector del multiplexor agregado es el bit más significativo de la instrucción. De esta forma, cuando se use “jr” el bit más significativo es 1 (porque el op de la instrucción es 32) y se usa el resultado de la ALU. Cuando se use “j” el bit más significativo es 0 (porque el op de la instrucción es 2) y el proceso es el mismo que ya estaba definido usando los 26 bits del campo “target”.

5.2.2. pipeline.cpu

Para el datapath multiciclo, se agrego una salida de la unidad de control que lleve el bit de jump a traves de los registros interfase.

De manera similar que con el datapath uniciclo, se agregó un Fork de la salida de la ALU a la cual se le hizo un shift a izquierda de 2 bits y se le concatenaron los 4 bits más significativos de la instrucción. Esta nueva dirección llega a la entrada “1” de un multiplexor en donde el selector es el bit de Jump que llega de la unidad de control. En la entrada “0” llega el resultado de “AddBranch”. La salida de este multiplexor llega al registro “EX/MEM” en la entrada de “target” y pasa a la entrada “1” del multiplexor de PC en la última etapa.

De esta forma, al multiplexor de PC le llega la dirección de salto o branch por un lado, y el valor de $PC + 4$ por otro. Para el selector de este multiplexor, antes llegaba la salida de AndBrach a la cual llegaba el bit de branch y un bit que indica si el resultado de la ALU es 0. Ahora, se agregó un OR al cual llegan la salida del AndBranch y el bit de Jump.

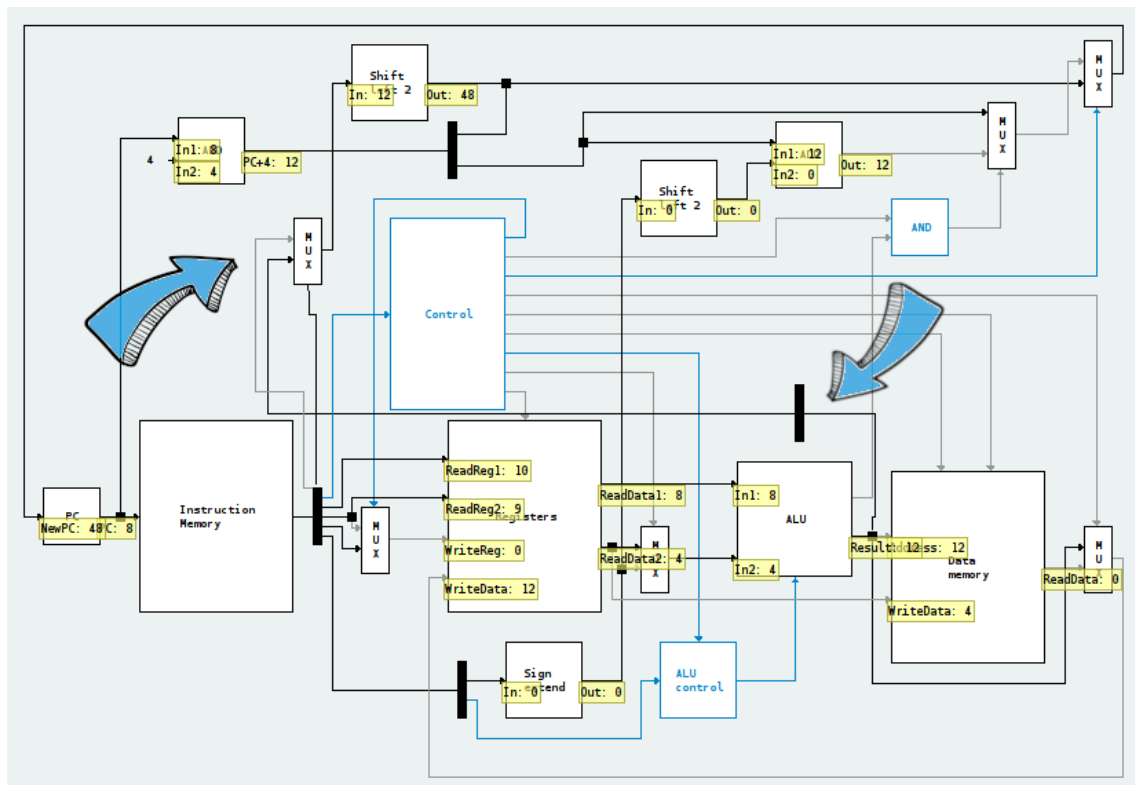
De esta forma, cuando Jump es 1 se usa el resultado de la ALU como nuevo PC y cuando Jump es 0, el comportamiento del datapath no cambia.

5.3. Caso de prueba

```
addi $t1, $zero, 4
addi $t2, $zero, 8
jr $t1, $t2
```

5.4. Diagramas .cpu

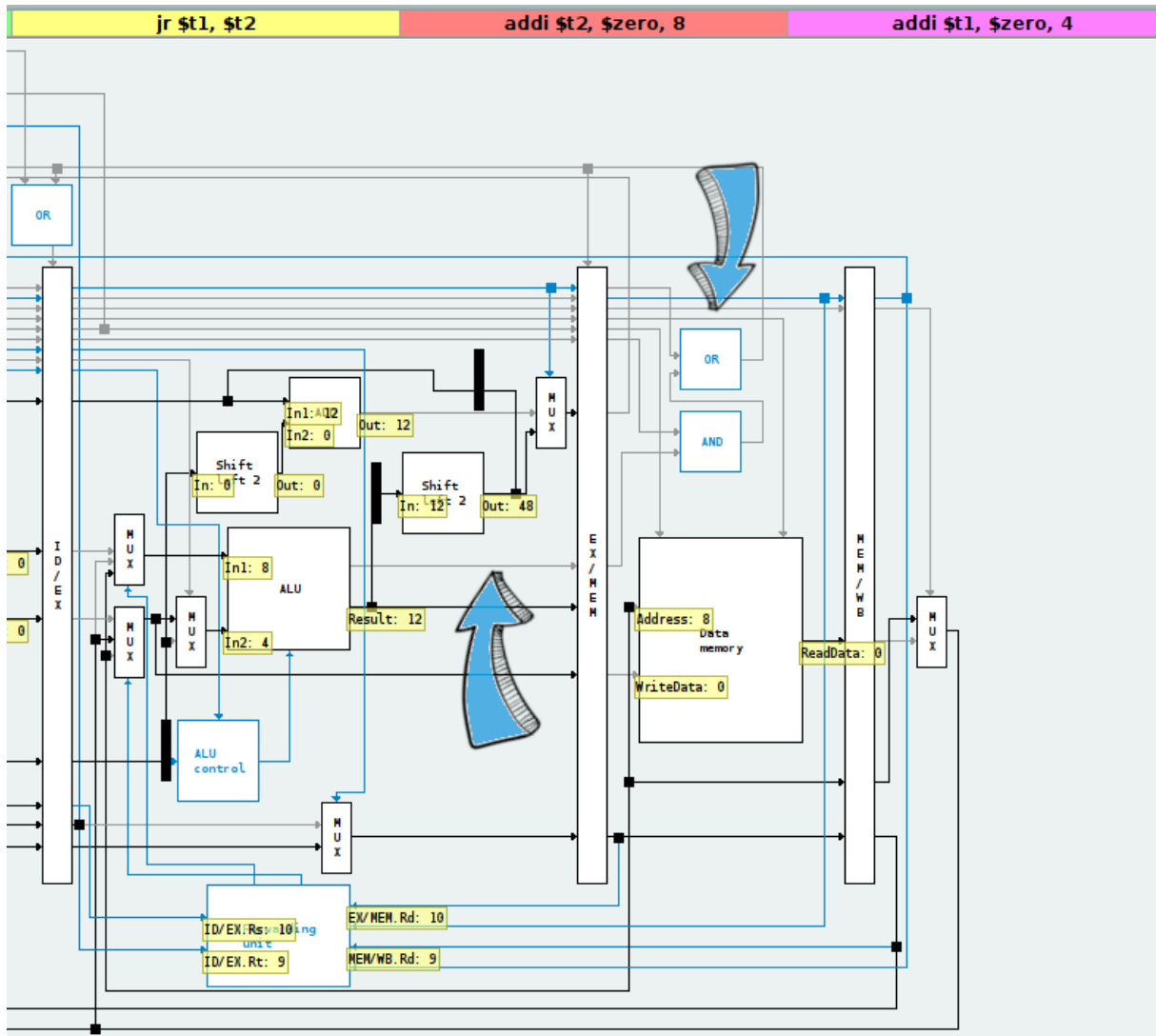
5.4.1. unicycle.cpu



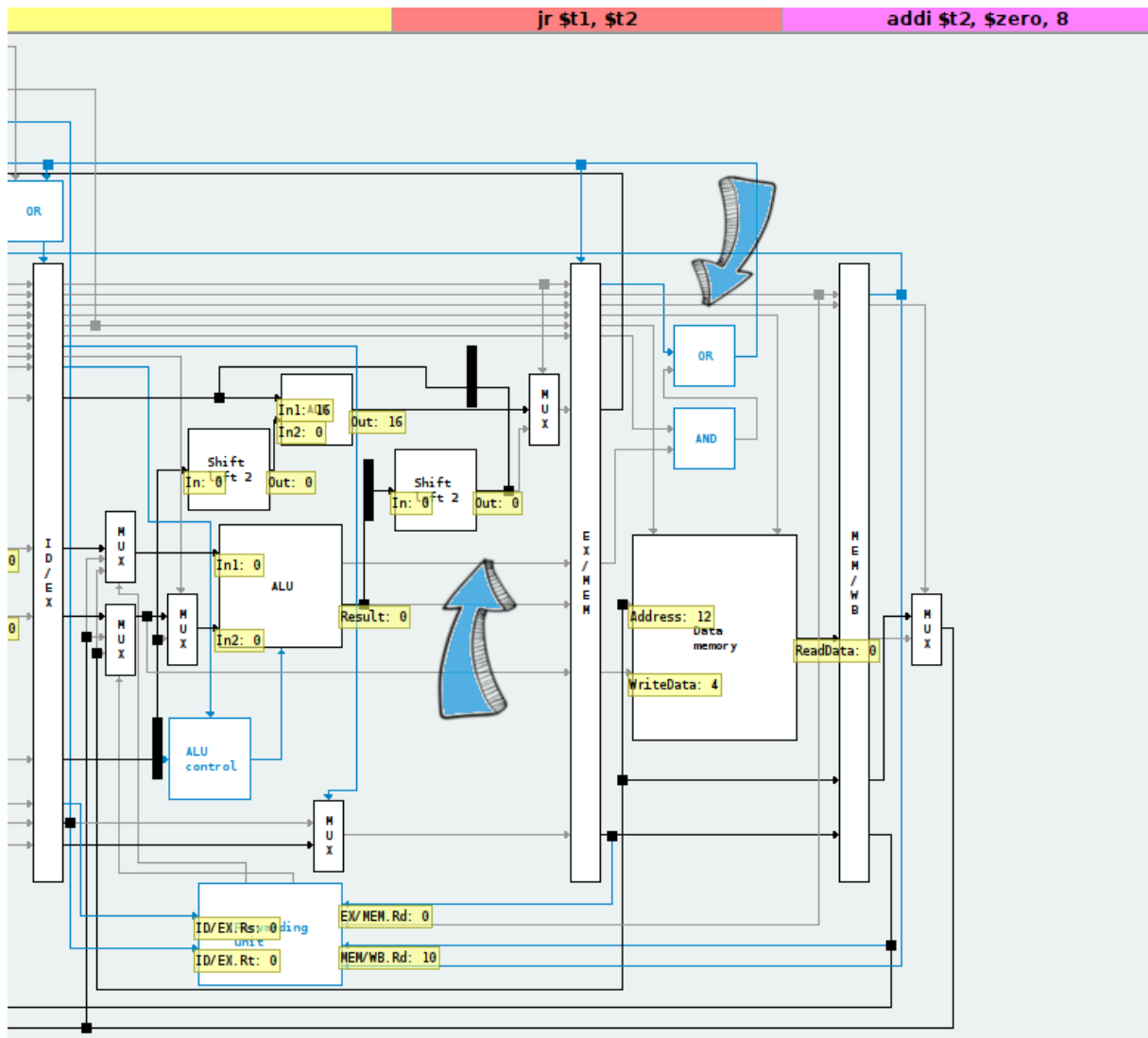
Mediante el Fork a la salida del resultado del cálculo de la ALU va ese dato por un cableado nuevo hasta el **nuevo distribuidor** señalado por la flecha derecha, el cuál deja pasar los 26 bits menos significativos del resultado que van al **nuevo multiplexor** señalado por la flecha izquierda como entrada 1. Luego, el bit de mayor peso de la instrucción, que para esta instrucción será 1 intencionalmente, hará que el multiplexor haga pasar ese resultado de la ALU mencionado, el cuál sufre luego un shift left 2 en el llamado ShiftJump y se le concatenan los 4 bits más significativos de la instrucción. En este ejemplo, el 12 se convierte en 48 y esa será la nueva dirección que llegará al multiplexor MuxJump, que recibiendo el 1 de unidad de control (Jump = 1) dejará pasar ese 48 al PC.

5.4.2. pipeline.cpu

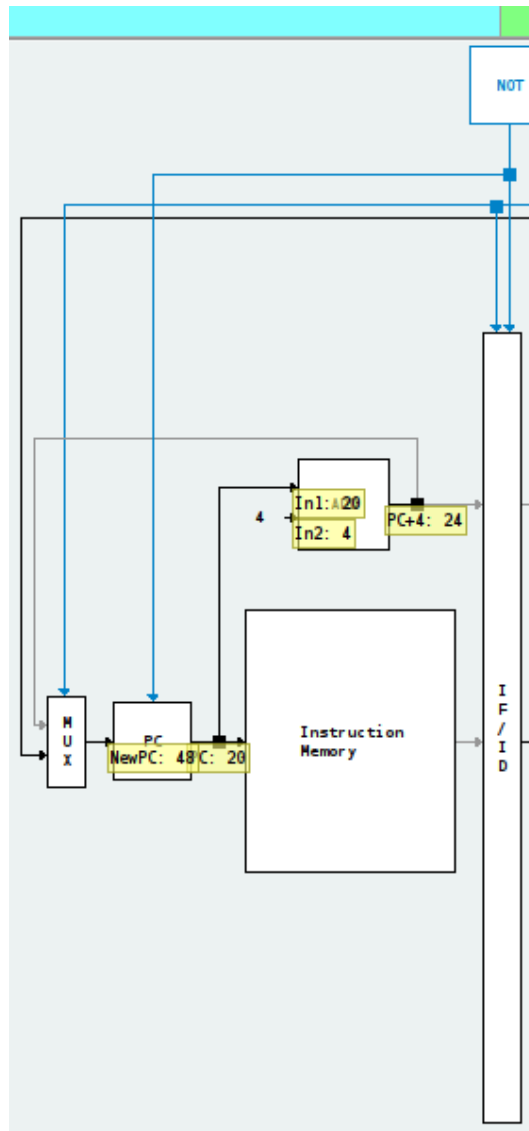
Se muestra a continuación el código de prueba con el **jr \$t1, \$t2** en etapa de ejecución:



Luego el código de prueba con el **jr \$t1, \$t2** en etapa de memoria, donde al PC viaja el valor 48:



Finalmente el PC con el valor deseado:



6. lw rs, rd, rt

Para la implementación de esta función en el datapath sólo fue necesario modificar el archivo `.set`, ya que en el `.cpu` los elementos presentes fueron suficientes para el correcto funcionamiento de la misma. Las entradas, salidas y cables empleados por las instrucciones ya existentes `add` y `lw` fueron los más aprovechados.

6.1. Modificaciones en archivos `.set`

6.1.1. Instrucción

En la sección **instructions** se agregó **lwr**.

```
"lwr": {"type": "R", "args": ["reg", "reg", "reg"],
      "fields": {"op": 33, "rs": "#2", "rt": "#3", "rd": "#1", "shamt": 0,
      "func": 32},
      "desc": "$t1 = MEM[$t2 + $t3]"}
```

El tipo es **R** (register) ya que sus parámetros son tres registros (5 bits cada uno). Los 6 bits restantes, los de mayor peso, corresponden al código de operación, en este caso se asignó el número de operación 33. El primer parámetro es el registro de destino y los otros dos representan los datos fuente.

6.1.2. Unidad de control

En la sección **control** agregamos una nueva operación para la unidad de control, arbitrariamente numerada 33.

```
"33": {"RegDst": 1, "RegWrite": 1, "ALUOp": 2, "ALUSrc": 0, "MemToReg": 1,
      "MemRead": 1, "MemWrite": 0}
```

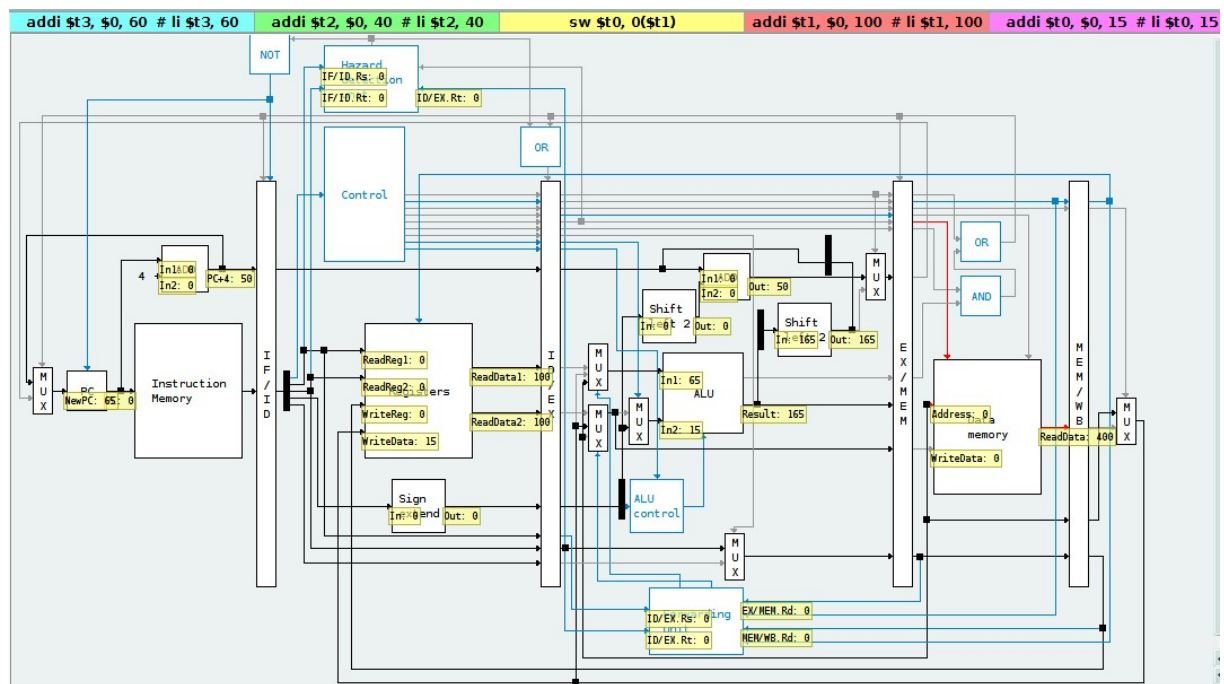
- `RegDst` es 1 porque los bits que representan el número de registro de destino son del 21 al 25 de la instrucción.
- `RegWrite` 1 ya que el resultado de la ALU se escribirá en registro.
- `AluOp` 2 es una operación existente.
- `ALUSrc` 0 es porque el segundo operando de la ALU debe ser la segunda salida del banco de registros.
- `MemToReg` 1 ya que el registro de destino se escribirá desde memoria principal.

6.2. Caso de prueba

```
li $t0, 15
li $t1, 100
sw $t0, 0($t1)
li $t2, 40
li $t3, 60
lwr $t1, $t2, $t3
```

Para este caso de prueba, en primer lugar se carga en la posición 100 de la MP el valor 15, mediante el registro t0. Luego cargando en los registro t2 y t3 respectivamente los valores 40 y 60, se genera mediante la suma de los mismos la dirección del dato a leer, el cual se verifica que se encuentra en el registro t1 un vez ejecutada la instrucción.

6.3. Diagrama .cpu



En la imagen se observa un paso intermedio en la ejecución de la instrucción lwr.

7. Problemas encontrados

No es un problema pero si algo que parece ralentizar la tarea: la interfaz de edición de los componentes y cableado de los archivos .cpu. Al ser sólo texto con coordenadas, las cuales hay que pasar a imágenes, nos llevó a probar el resultado una y otra vez hasta que quedara visualmente entendible. Fue difícil familiarizarse con los archivos .set y .cpu en general pero no lo llamaríamos **problema**.

8. Conclusiones

Mediante la realización del presente trabajo práctico pudimos comprender mejor la constitución del datapath, los elementos presentes en el mismo y la coordinación y movimiento de datos tanto en la configuración uniciclo como pipeline. Pudimos observar como en ciertas ocasiones, al agregar una instrucción que no se encontraba originalmente en el datapath, pueden aprovecharse los elementos, conexiones y salidas existentes para otras funciones, sin necesidad de realizar modificaciones en el archivo .cpu.

9. Referencias

- Hennessy, John L. and Patterson, David A., Computer Architecture: A Quantitative Approach, Third Edition, 2002.
- DrMIPS, <https://brunonova.github.io/drmips/>.