# Trabajo Práctico 3
# Datapath y Pipeline

Sebastian Ignacio Penna, *Padrón 98752*
sebapenna19@gmail.com

Constanza M Frutos Ramos, *Padrón 96728*
constanza.frutos.ramos@gmail.com

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

**Resumen**

En este trabajo práctico se busca entender el funcionamiento del datapath con y sin pipeline.

# Índice

# 1. Introducción

El objetivo principal de trabajo es poder entender el funcionamiento del datapath. Para esto se usa el simulador DrMIPS. En este hay archivos de unicilo y de pipeline. Estos deben modificarse, agregandoles instrucciones que no poseen, y logrando que dichas instrucciones se ejecuten de manera correcta.

# 2. Análisis del problema

Se piden agregar instrucciones a pipeline.cpu y a unycicle.cpu, siempre teniendo en cuenta que se ejecuten de manera correcta y no se produzcan hazards. Para esto se busca en el anexo del libro Hennesy-Patterson cada instrucción a implementar.

# 3. Tipos de instrucciones

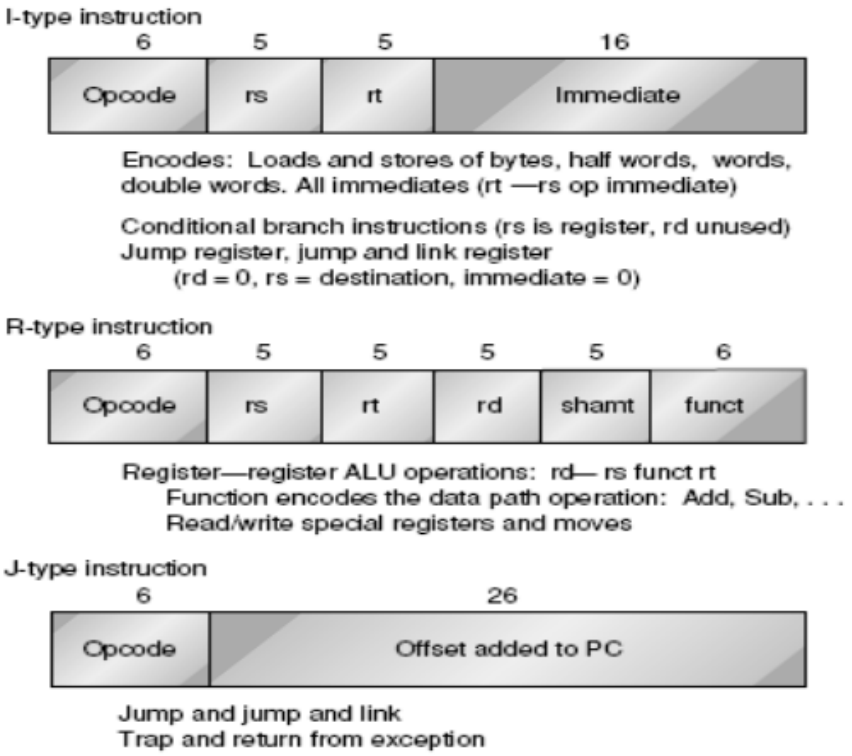


Figura 1: Tipos de instrucciones en MIPS.

Hay tres tipos de instrucciones en MIPS, estas son las de tipo I, R y J. Todas miden 32 bits, pero cada una de esta los interpreta de manera diferente. En este trábajo práctico se pide implementar instrucciones de tipo J: j, e instrucciones de tipo R: sll, srl, jr. Dado que jalr es una pseudoinstrucción, es necesario implementar jr para esta.

# 4. Desarrollo

## 4.1. Funcionamiento de la aplicación

Para agregar una instrucción se debe modificar el archivo .set que se vaya a usar junto al archivo .cpu (contiene el datapath) y .asm (donde se encuentra el caso de prueba). Este archivo esta en formato JSON. Sus campos son:

- comment: contiene un comentario del archivo.
- types: acá se declaran los tipos de instrucciones que se van a implementar. Se indica cada uno de sus campos, con nombre y cantidad de bits.
- instructions: se declaran las instrucciones, indicando su nombre, tipo, argumentos (si tiene), campos, y una descripción (opcional).
- pseudo: se declaran las pseudoinstrucciones y como se relacionan con las instrucciones anteriores.
- control: para cada clave indica como se cambian los componentes del datapath. En caso de ser una operación que se resuelve en la alu se indica el ALUOp, con el cual luego se obtiene la operación.
- alu: determina la salida de la operación.

Para agregar una instrucción deben modificarse estos campos de manera que el programa entienda la instrucción que se esta utilizando.

## 4.2. Implementación

## 4.3. Instrucciones a agregar

Las instrucciones se agregan en archivos especificos a cada ejercicios basados en default.set.

### 4.3.1. j en pipeline.cpu

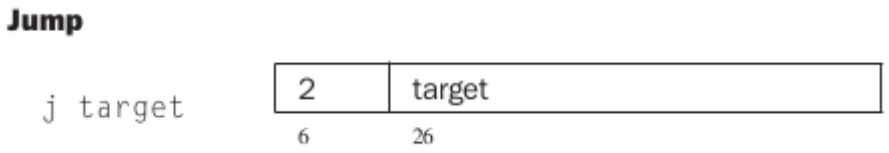


Figura 2: Instrucción j.

En el archivo ejercicio-1.set:
Primero se agrega el tipo de instrucción en "types":

```
"J": [{"id": "op", "size": 6}, {"id": "target", "size": 26}]
```

Luego se completan el resto de los campos del JSON:

```
En "instructions":
"j":    {"type": "J",
         "args": ["target"],
         "fields": {"op": 2, "target": "#1"},
         "desc": "PC = target"},
En "control":
"2": {"Jump": 1}
```

### 4.3.2. sll en unycicle.cpu


Figura 3: Instrucción sll.

En el archivo ejercicio2.set:
La instrucción sll es de tipo R. Este tipo ya esta definido. Se agrega la instrucción:

```
En "instructions":
"sll":  {"type": "R",
         "args": ["reg", "reg", "int"],
         "fields": {"op": 0, "rs": "#3", "rt": "#2", "rd": "#1",
         "shamt": "#3", "func": 0}}
En "alu":
{"aluop": 2, "func": 0,  "out": {"Operation": 13}}
y su código de operación:
"13": "sll"
```

### 4.3.3. srl en pipeline.cpu

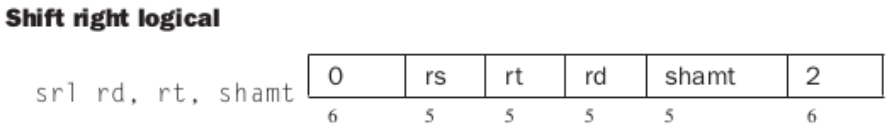


Figura 4: Instrucción srl.

La instrucción srl también es de tipo R. Se agrega la instrucción:

```
En "instructions":
"srl":  {"type": "R",
         "args": ["reg", "reg", "int"],
         "fields": {"op": 0, "rs": 0, "rt": "#2", "rd": "#1",
         "shamt": "#3", "func": 2}}
En "alu":
{"aluop": 2, "func":  2, "out": {"Operation": 13}}
y su código de operación:
"13": "srl"
```

### 4.3.4. jalr en unycicle.cpu y pipeline.cpu


Figura 5: Instrucción jalr.

La instrucción jalr es de tipo R. Se agrega la instrucción:

```
En "instructions":
"jalr": {"type": "R",
         "args": ["reg", "reg"],
         "fields": {"op": 1, "rs": "#1", "rt": 0, "rd": "#2",
         "shamt": 0, "func": 9}, "desc": "PC = rs, rd = PC+4"}
En "control":
"1": {"RegDst": 1, "RegWrite": 1, "ALUOp": 2, "ALUSrc": 0, "MemToReg": 0,
"JumpReg": 1, "Link": 1}
En "alu":
{"aluop": 2, "func":  9, "out": {"Operation": 15}}
y su código de operación:
"15": "jalr"
```

Esta instrucción recibe dos parámetros. El primer registro contiene el valor del nuevo PC, y en el segundo se guarda el PC+4 actual. Esta instrucción admite que se reciba un único parámetro, por lo cual se coloca el PC+4 en ra. Dado que el formato JSON no permite que haya dos claves iguales, y la instrucción es única, no se supo como armar la señal de control para determinar si hay un segundo registro o va el default.

# 5. Casos de Prueba

### 5.0.1. test1.asm

```
j finish
nop
nop
nop
nop
addi $t1, $zero, 4

finish:
 addi $t1, $zero, 8
```

Cuando se ejecuta la instrucción de jump salta al PC correspondiente a la etiqueta finish. Dado que es pipeline se agregan nops para evitar que se ejecute la instrucción addi, esto puede evitarse flusheando. La instrucción que suma cero con 4 en t1 nunca se ejecuta puesto que salta a la otra.

### 5.0.2. test2.asm

```
li $t0, 10
li $t1, 2
li $t2, 4

sll $t0, $t0, 2
sll $t1, $t1, 2
sll $t2, $t2, 3
```

Se cargan valores en tres registros y luego se llama a sll.

### 5.0.3. test3.asm

```
li $t0, 10
nop
nop
```

```
nop
nop
srl $t0, $t0, 2
```

Se carga el valor 10 en t0. Se meten ciclos de stall con nop. Y luego se ejecuta srl, quedando un 2 en t0.

#### 5.0.4. test4.asm

```
li $t0, 2048
li $t1, 1024

jalr $t0, $t1
```

Se carga 2048 en t0, 1024 en t1. Luego se llama a jalr con estos dos registros. Esto resulta en que el PC cambia al valor contenido en t0. Y el PC actual mas 4 se coloca en t1.

#### 5.0.5. test5.asm

```
li $t0, 2048
nop
nop
nop
nop
li $t1, 1024
nop
nop
nop
nop

jalr $t0, $t1
```

Se hace lo mismo que en anterior solo que se agregan los ciclos de stall para que no haya problemas en la ejecución.

# 6. Datapath

## 6.1. Modificaciones en el datapath

- Ej1 Jump en pipeline.cpu:
  Dado que no esta implementado el salto en este archivo, se agregan los componentes necesarios para esta instrucción. Para esto se agrega en el control un bit de Jump, el cual se coloca en 1 cuando la instrucción es la indicada. Este bit se arrastra a través de las distintas etapas sin modificarlo. Dado que necesitamos el target, en el distribuidor "DistInst"de la etapa ID, agregamos un cable que saque los bits correspondientes a este (25-0). En la etapa EX se agrega un shifter, en el cual ingresan estos bits y se shiftean a izquierda en 2, esto es para que la dirección sea multiplo de 4. Luego se le debe concatenar los 4 bits mas significativos del PC+4. Para esto se agrega un concatenador ÇoncatJump". El resultado de 32 bits se arrastra a las siguientes etapas. En la última etapa, WB, se mandan el bit de Jump y el Target completo a un multiplexor "MuxJump". El bit de Jump actúa como selector y el Target como la entrada asociada al 1. La salida de "MuxBranch"pasa a ser la entrada correspondiente al 0 de "MuxJump". Entonces, si es instrucción de salto se pone el target en el PC, y si no lo es se pone el resultado que se calcularía en otra ocasión (este puede ser PC+4 o dirección de Branch).

- Ej2 y Ej3 sll y srl:
  Para esta instrucción se necesita obtener los bits correspondientes al shamt, los cuales son los bits 10-6 de la instrucción. El distibutor separa entre los bits: 31-26: op; 25-21: rs; 20-16: rt; 15-11: rd; 15-0: imm en las de tipo I. Por lo tanto, para agregar esta función se deben obtener estos bits. Luego debe haber un multiplexor que elija entre rs o shamt si la función es de shift o no. La función es de shift cuando op es igual a 0 y func tiene sus tres bits mas significativos en 0. Entonces, Conociendo func (bits 5-0) y op se obtiene la señal de control al MUX, si es función de shift pasa el shamt y sino pasa rs.
  Si pongo el valor de shamt en rs no necesito hacer esto. Dado que estas instrucciones ignoran rs no hay problema. Entonces a Registers entran shamt, rt y rd. Luego a la ALU pasan rt y shamt, puesto que no es instrucción I por lo cual el MUX no elige al inmediato. En la ALU se elige la operación correspondiente con func y op. Esta parte se hizo de dos maneras. En el caso del Unycicle se cambio el op de la instrucción para que sea mas simple, en este caso directamente se tiene un bit shamt de control que se usa en el mux. En el caso del pipeline se calcula si es instrucción de shift o no usando tres or y dos mux.
  Una vez que se obtiene el shamt este ingresa a la ALU, donde se realiza la operación de shift que ya esta implementada en dicha unidad.
  En el caso del pipeline se tiene en cuenta que los datos sean los correspondientes a cada etapa.

- Ej4 y Ej5 jalr:
  Para implementar esta instrucción se decide cambiarle el op para que sea mas fácil la verificación. Se agregan dos bits de control, JumpReg y Link. Ambos bits se encienden con la instrucción jalr (se indica en el .set).
  Se agrega un multiplexor para elegir si el nuevo PC va a ser el calculo anterior (salida del MuxPC entre PC+4 o target del branch) o el valor en el registro rs del jalr. Y se agrega otro multiplexor para decidir cual sera el WriteData. En el caso de esta instrucción va el PC+4.
  En ambos .cpu se sigue la misma idea. En el caso del pipeline hay que respetar las etapas de ejecución. Se arrastran los datos necesarios de cada etapa hasta la última (WB) y ahí se finalizan las conexiones. De ID se obtiene el valor de rs, y los dos bits de control. De EX se obtiene el nuevo PC.

## 6.2. Capturas de pantalla del datapath

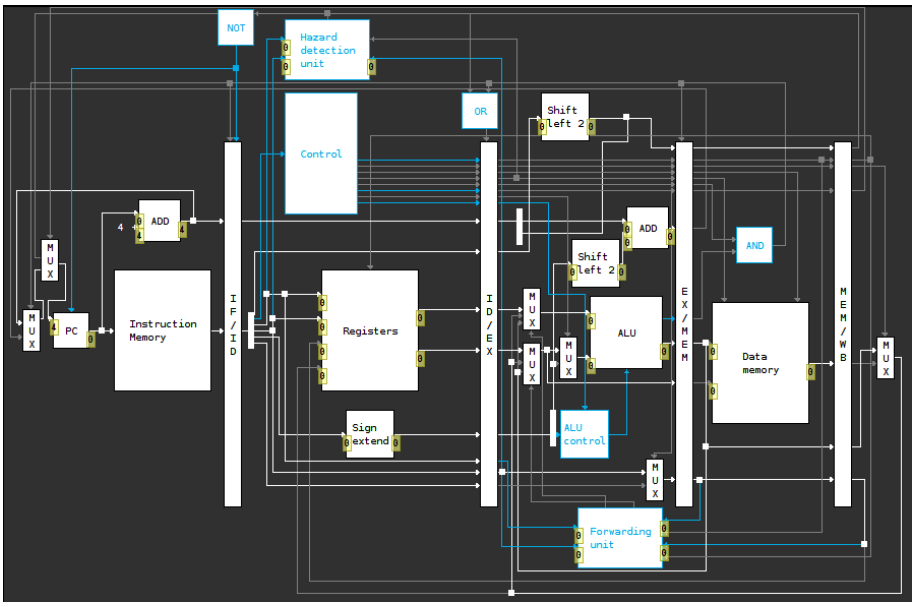


Figura 6: Captura de pantalla del ej1.

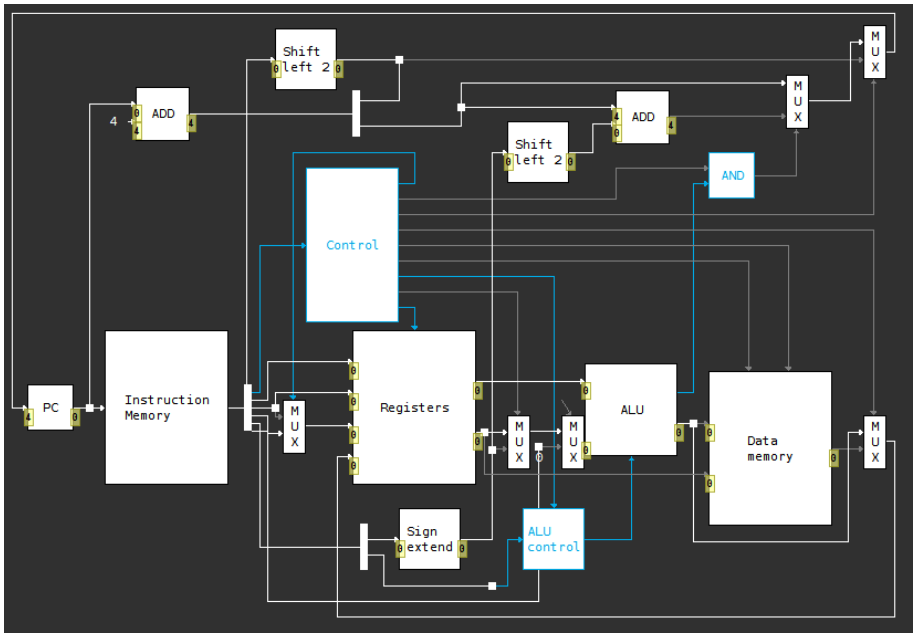


Figura 7: Captura de pantalla del ej2.

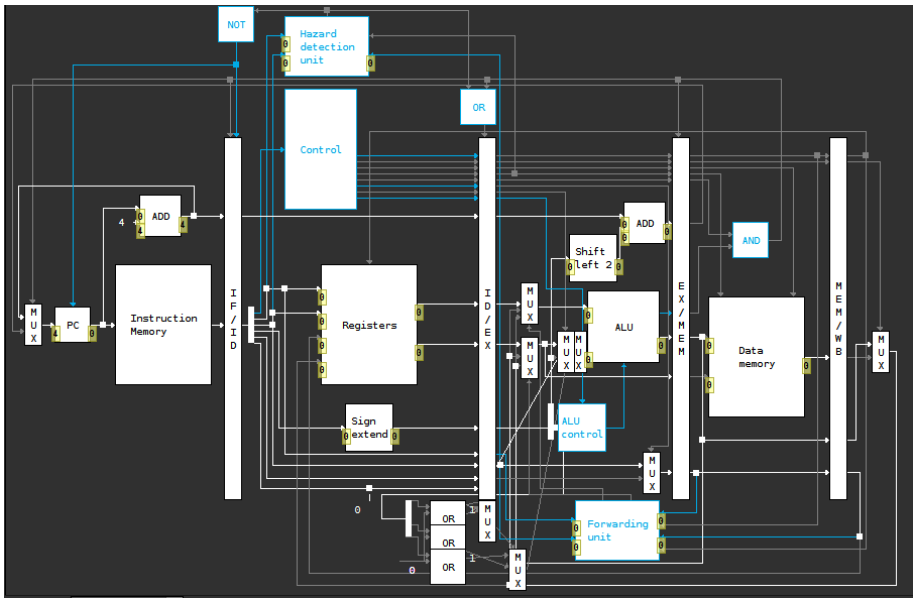


Figura 8: Captura de pantalla del ej3.

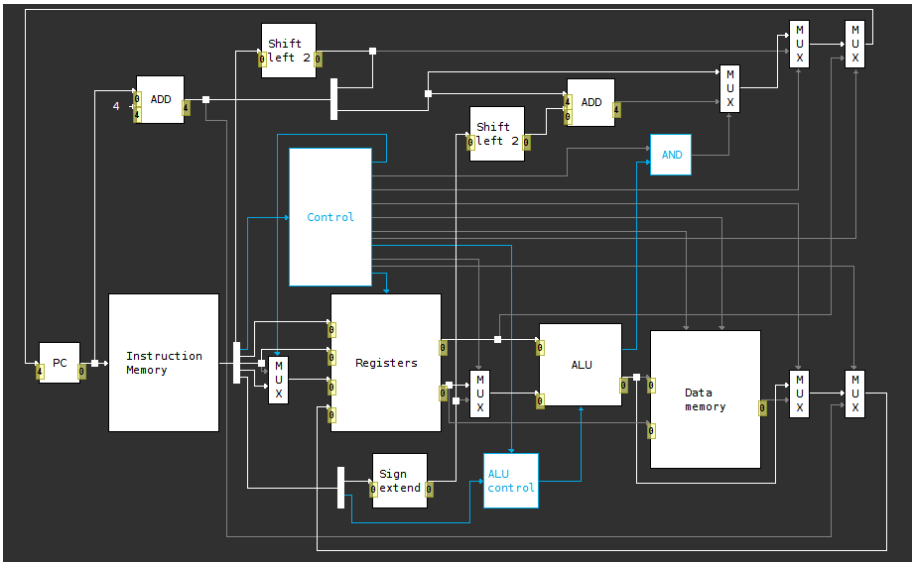


Figura 9: Captura de pantalla del ej4.

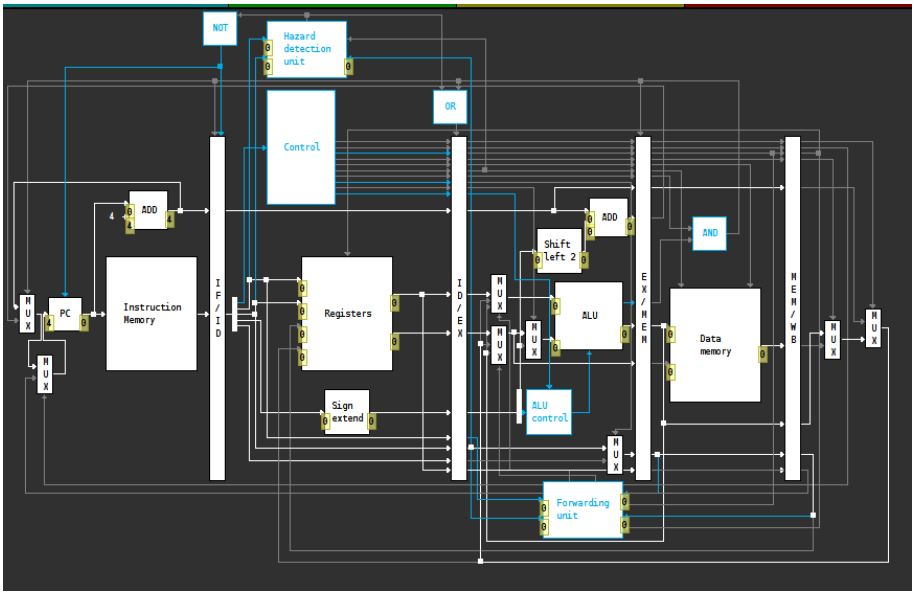


Figura 10: Captura de pantalla del ej5.

# 7. Conclusión

Al agregar instrucciones no soportadas por DrMIPS se tuvo que pensar el problema desde el lado del cableado del Datapath. Se investigaron los diferentes componentes soportados por este y se pensó cuales debían ser agregados para resolver cada problema.
Dado que es muy complejo resolver el problema haciendo las verificaciones correspondientes, se decidió cambiar los op para poder obtener una señal de control con mayor facilidad y así evitar agregar muchos componentes que entorpecen la vista del datapath.
Al resolver los problemas mediante el cableado del datapath se pueden entender conceptos teóricos de manera práctica. Se logran integrar conocimientos de esta materia y de estructura donde se usan los mismos componentes.

# Referencias

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

[2] Anexo del libro Patterson-Hennesy.