
# Asignatura:
# 66.20 Organización de computadoras

## Profesor: Hamkalo, José Luis

# Trabajo Práctico:
# Conjunto de instrucciones MIPS


| Nombre y apellido | Padrón | Correo electrónico | Slack |
|---|---|---|---|
| Julian Mejliker | 100866 | jmejliker@gmail.com | Julian Mejliker |
| Tomas Nocceti | 100853 | nocetti.tomas@gmail.com | Tomas Nocetti |


**Primer cuatrimestre 2020**

# Índice

# 1. Introducción

El objetivo del trabajo práctico es interiorizarse con las arquitecturas de computadoras seg- mentadas, utilizando de forma práctica una arquitectura tipo MIPS32 simulada. Durante el desarrollo del trabajo se realizarán modificaciones a implemetaciones ya provistas de la arquitectura MIPS en versión uniciclo y multiciclo. Las modificaciones a realizar consisten en el agregado de ciertas instrucciones a la arquitectura, tales como saltos y shifts teniendo en consideración los riesgos que pueden producirse en el caso multiciclo. La arquitectura uniciclo ejecuta las instrucciones de a una por vez, tomando varios ciclos de reloj para completar una instrucción; mientras que la arquitectura multiciclo está segmentada en cinco etapas, manteniendo la cantidad de ciclos de reloj requeridos por instrucción pero ejecutando más de una instrucción al mismo tiempo, y por consiguiente, obteniendo un CPI más bajo.

# 2. Implementaciones

## 2.1. Instruccion ANDI en uniciclo / multiciclo

Para la implementacion de la instruccion ANDI en uniciclo/multiciclo, no se realizó ninguna modificación en los componentes, sino que se creo una nuestra instrucción directamente en el archivo con el set de instrucciones. La decisión anterior fue tomada analizando los componentes ya existentes y viendo la similitud con otras instrucciones.

Para la implementación de la instrucción también se genero un nuevo opcode, el cual nos permite indicarle a la alu la operación a realizar y al register Bank que source utilizar.

### 2.1.1. Código

La instrucción ANDI agregada es la siguiente:

```
"andi": {
    "type": "I",
    "args": [
        "reg",
        "reg",
        "int"
    ],
    "fields": {
        "op": 3,
        "rs": "#2",
        "rt": "#1",
        "imm": "#3"
    },
    "desc": "rs = rt + imm"
}
```

La codificación del nuevo opcode es el siguiente:

```
"3": {
    "RegDst": 0,
    "RegWrite": 1,
    "ALUOp": 2,
    "ALUSrc": 1,
    "MemToReg": 0
},
```

### 2.1.2. Casos de prueba

Se utilizó el siguiente código de pruebas:

```
addi $a0, $zero, 4
andi $a3, $a0, 12
```

Se comprobó que los registros quedan con los resultados esperados.

## 2.2. Instrucción J en uniciclo

Para la implementación de esta instrucción se realizaron cambios en los componentes del datapath. El cambio fue simple, se cambió la entrada del ShiftJump, el cual antes recibia la dirección del salto desde la instruccion luego de ser decodificada pero ahora recibe la direccion del salto desde la ALU. En la siguiente imagen se puede ver como el resultado de la ALU es enviado al ShiftJump. Por otro lado, se modifico el Opcode utilizado por la funcion para poder definir la operacion a realizar por la ALU al momento de ejecutar la funcion. Tambien se elimino la concatenación de los 4 bits mas significativos del PC con la salida del ShiftJump.

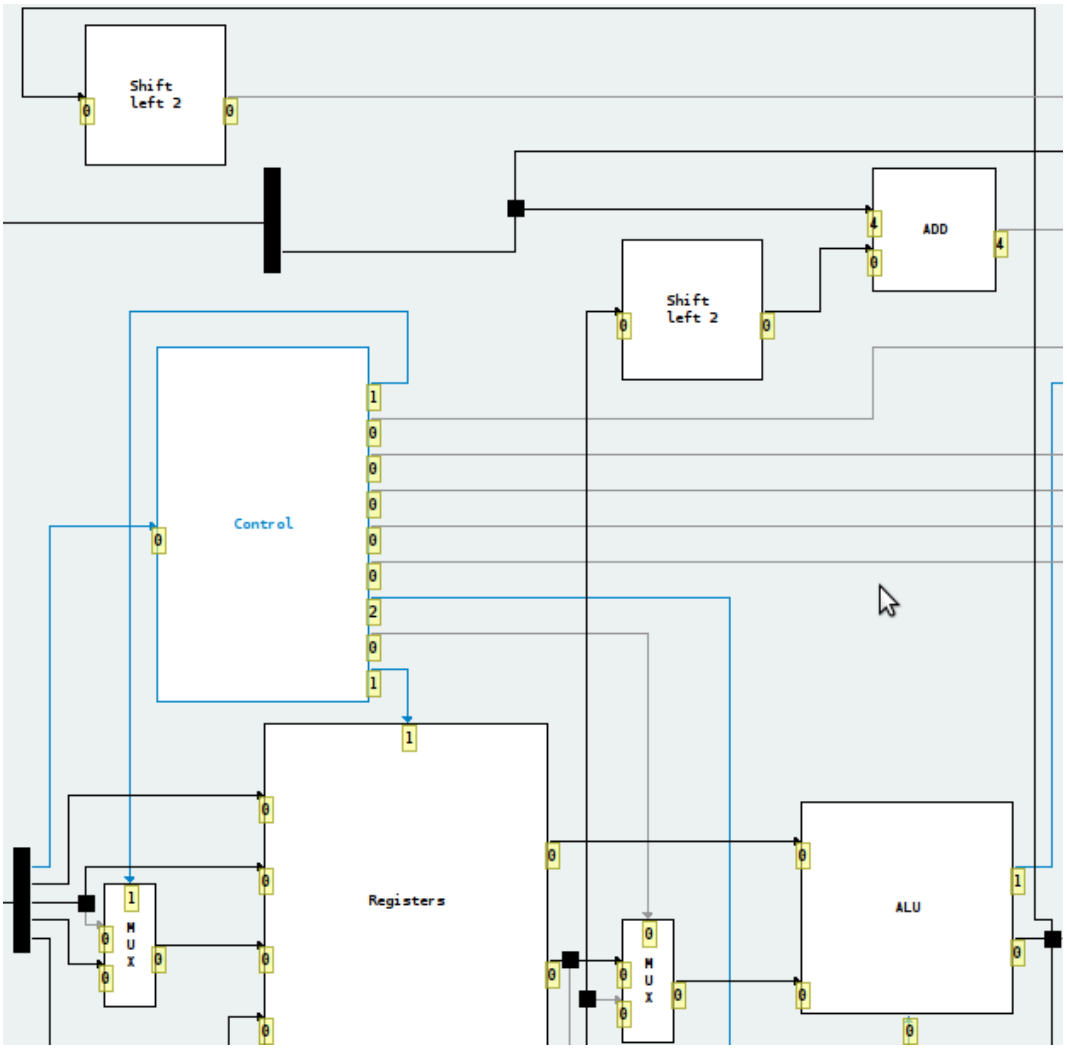


Figura 1: Nueva distribucion de componentes y cables

### 2.2.1. Codigo

Implmentacion de la instruccion J:

```
"j": {
    "type": "I",
    "args": [
        "reg",
        "reg"
    ],
    "fields": {
        "op": 2,
        "rs": "#1",
        "rt": "#2",
        "imm": 0
    },
    "desc": "PC = rs + rt"
```

```
},
```

Implementacion del OPCode:

```
"2": {
    "Jump": 1,
    "ALUSrc": 0,
    "ALUOp": 3
},
```

Cable que envia el resultado de la ALU al ShiftJump y del ShiftJump al MUX:

```
{"from": "ForkMem", "out": "Out3", "to": "ShiftJump", "in": "In", "start": {"x": 470, "y": 275}, "p
{"from": "ShiftJump", "out": "Out", "to": "MuxJump", "in": "1"}
```

#### 2.2.2. Casos de prueba

Se utilizo el siguiente codigo de prueba:

```
addi $a0, $zero, 1
addi $a1, $zero, 5
j $a0,$a1
and $a0, $zero, $a1
and $a0, $zero, $a1
and $a0, $zero, $a1
addi $a3, $zero, 100
and $a0, $zero, $a1
```

### 2.3. Instruccion J en Multiciclo

En este ejercicio se pide implementar la instrucción J en la arquitectura multiciclo. Para esto se decidió utilizar la estructura ya implementada para las instrucciones de tipo branch, aprovechando que las instrucciones de tipo jump cumplen el mismo objetivo: alterar el valor del PC. La diferencia entre las mismas radica en el direccionamiento. Las instrucciones branch direccionan de forma relativa a la posición actual mientras que las de tipo jump lo hacen de forma absoluta. Analizando el comportamiento de las instrucciones de tipo branch se puede observar que durante la etapa de IF se toma el valor inmediato de la instrucción, se le extiende el signo y se lo propaga a la siguiente etapa. A sı mismo se propaga directamente el valor del PC que tomará la siguiente instrucción a la próxima etapa para poder hacer el direccionamiento relativo al mismo. La problemática principal era como lograr que la etapa de pipie EX/MEM reciba en su registro "Target"la dirección de salto. Entonces lo primero que se hizo fue agregar una nueva señal de control para detectar que nos encontrábamos en la secuencia de salto, se la llamo "Jump". Este valor de señal se propago en los pipes ID/EX y EX/MEM. Luego se adopto el siguiente circuito para definir si el valor del "Target"se obtenía del branch o se obtenía de la salida de la ALU para realizar el Jump.

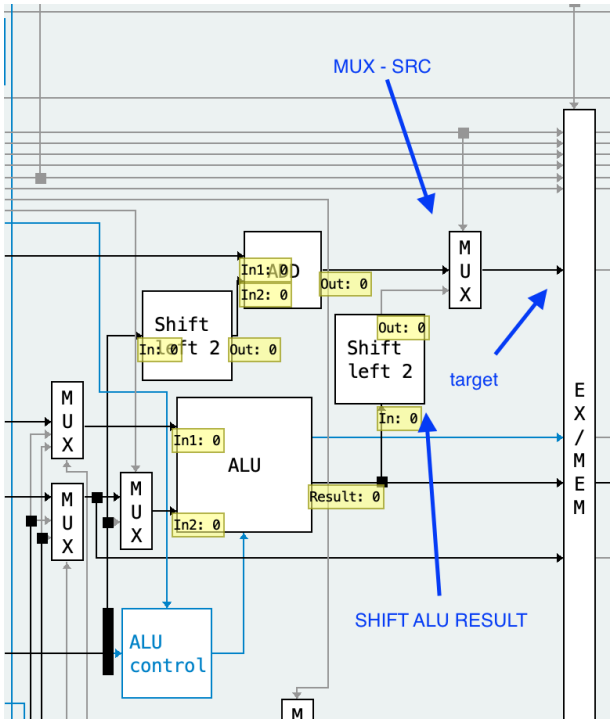


Figura 2: Circuito lógico en etapa EX

De aquí la señal de Jump llega del pipe ID/EX hacia el MUX - SRC donde decide si la dirección del registro Target es:

- La dirección de branch obtenida como se menciono anteriormente.
- La dirección de jump obtenida como salida de la ALU + un shift left 2 para convertir el valor a múltiplo de 4.

Ya en esta instancia y con el valor fijado en el registro para la próxima etapa se debió realizar la incorporación de un nuevo componente OR que nos permita ejecutar la lógica de salto en caso de una instrucción J.

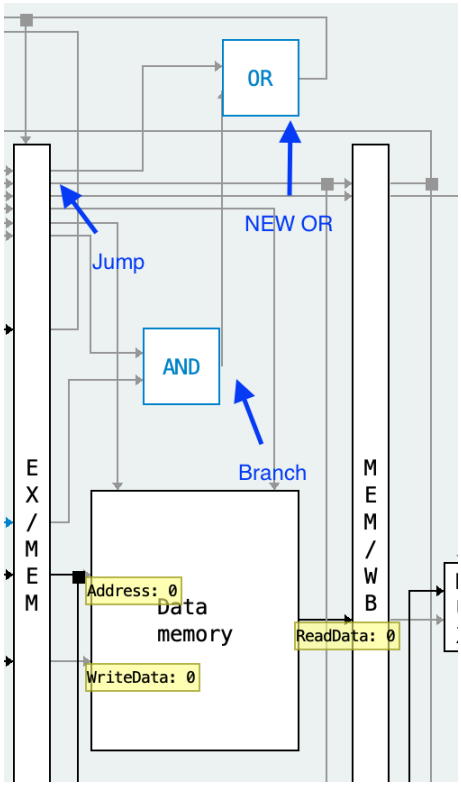


Figura 3: Circuito lógico en etapa MEM

Dado que el control del flujo de ejecución no fue modificado, no hay riesgos posibles, ya que la implementación provista ya se encarga de vaciar el pipeline al ejecutar el salto.

### 2.3.1. Código

Se agrego la nueva instrucción con el siguiente formato. Utilizando el mismo Control Op que en la función J Uniciclo.

```
"j": {
  "type": "I",
  "args": [
    "reg",
    "reg"
  ],
  "fields": {
    "op": 2,
    "rs": "#1",
    "rt": "#2",
    "imm": "0"
  },
  "desc": "PC[$t1 + $t2]"
}
```

### 2.3.2. Casos de prueba

El primer caso de prueba se busco ver el correcto funcionamiento de la instrucción con el pipeline limpio después de la etapa MEM. Este código genera un bucle infinito.

```
li  $a0, 0
li  $a1, 0
j $a0, $a1
li  $a2, 0
```

El segundo caso de prueba buscaba ver que la protección contra Hazards no se haya modificado.

```
sw $zero, 10
lw $a1, 10
add $a2, $a1, $zero
```

El tercer caso de prueba buscaba ver que la instrucción branch no se haya modificado.

```
li  $a0, 0
li  $a1, 0
beq $a0, $a1, -2
li  $a2, 0
```

## 2.4. Instrucción LW en multiciclo

Para el desarrollo de esta instrucción se opto por una pseudo instrucción. En primer lugar cambiamos el nombre de la instrucción debido a que debíamos hacer uso de la instrucción lw original y no pueden haber 2 instrucciones con el mismo nombre. El segundo cambio fue modificar la firma que pedía el trabajo debido a que el formato pseudo no soporta el tipo #data. La firma final es:

```
lwd rs, rd, imm, rt
```

Es importante destacar que no era posible realizar cambios en el circuito para poder ejecutar la instrucción en una sola pasada de pipeline.

### 2.4.1. Código

Implementación de la pseudo-instrucción desarrollada:

```
"lwd":{
     "args": [
         "reg",
         "reg",
         "int",
         "reg"
     ],
     "to":[
         "addi $1, $zero, #3",
         "mult #2, $1",
         "mflo $1",
         "add $1, $1, #4",
         "lw #1, 0($1)"
     ],
     "desc": "$t1 = $t2*Imm($t3)"
     }
```

### 2.4.2. Casos de prueba

Se utilizo el siguiente codigo:

```
li $t0, 10
sw $t0, 100
lwd $a1, $t0, 10, $a0
```

Se comprobó que los registros quedan con los resultados esperados.
El segundo caso de prueba buscaba ver que la protección contra Hazards no se haya modificado.

```
sw $zero, 10
lwd $a1, $t0, 10, $a0
add $a2, $a1, $zero
```

# 3. Conclusión

Luego de analizar todas las instrucciones pedidas se vio que el trabajo requería un análisis profundo de los distintos data-path (uniciclo y multicilo) para lograr la mejor optimización posible para cada una de ellas. También se entendió que muchas de estas instrucciones, en la realidad no merecen una alteración del circuito físico debido a que esto involucra un gasto de espacio que generalmente no se posee, por lo que la solución lógica es el uso de pseudoinstrucciones. En todo momento del trabajo se pensó qué existía para esa dada instrucción y qué se podía reutilizar del circuito logrando de esta manera minimizar las alteraciones en todo momento y reutilizar lógica de protección o forwarding que ya estaba resuelta.

# 4. Anexo

Agrego link al github: Codigo Fuente