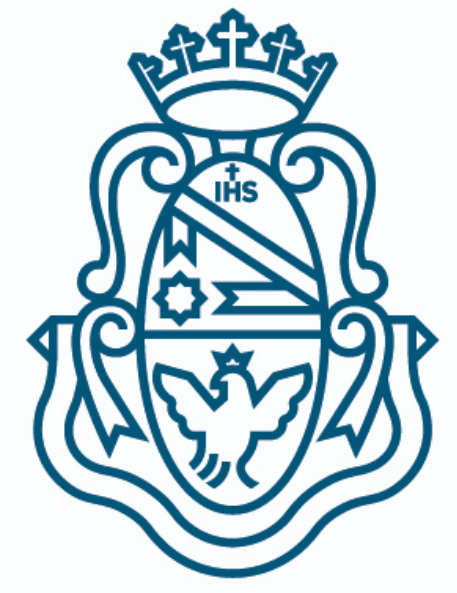
Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales



Arquitectura de Computadoras

Trabajo Práctico Final

Pipeline procesador MIPS simplificado

COLLANTE, Gerardo
QUINTEROS CASTILLA, Nicolás

Profesor:
Ing. Rodríguez, Martín

3 de agosto de 2020

Índice

1. Introducción

Con el nombre de MIPS (siglas de *Microprocessor without Interlocked Pipeline Stages*) se conoce a toda una familia de microprocesadores de arquitectura RISC (*Reduced Instruction Set Computer*) desarrollados por *MIPS Technologies*.

En este trabajo práctico se solicita implementar en el lenguaje de descripción de hardware `Verilog` (usado para modelar sistemas electrónicos usualmente) el *pipeline* del procesador *MIPS*.

2. Requerimientos

2.1. Etapas

Se solicita implementar las siguientes etapas del procesador:

1. `IF` *(Instruction Fetch)*: búsqueda de la instrucción en la memoria del programa.
2. `ID` *(Instruction Decode)*: decodificación de la instrucción y lectura de registros.
3. `EX` *(Execute)*: ejecución de la instrucción propiamente dicha.
4. `MEM` *(Memory Access)*: lectura o escritura desde/hacía la memoria de datos.
5. `WB` *(Write Back)*: escritura de resultados en los registros.

2.2. Instrucciones

Se solicita implementar las siguientes instrucciones:

- `R-Type`: `SLL, SRL, SRA, SLLV, SRLV, SRAV, ADDU, SUBU, AND, OR, XOR, NOR, SLT`
- `I-Type`: `LB, LH, LW, LWU, LBU, LHU, SB, SH, SW, ADDI, ANDI, ORI, XORI, LUI, SLQTI, BEQ, BNE, J, JAL`
- `J-Type`: `JR, JALR`

2.3. Riesgos

El procesador debe tener soporte para los siguientos tipos:

- **Estructurales**: se producen cuando dos instrucciones tratan de utilizar el mismo recurso en el mismo ciclo.
- **Datos**: se intenta utilizar un dato antes de que esté preparado. Mantenimiento del orden estricto de lecturas y escrituras.
- **Control**: intentar tomar una decisión sobre una condición todavía no evaluada.

Por tanto se deberá implementar una **unidad de cortocircuitos** y una **unidad de detección de riesgos**.

2.4. Otros

El programa a ejecutar debe ser cargado en la memoria de programa mediante un archivo ensamblado, por tanto debe crearse un ensamblador.

3. Ensamblador

4. Etapas o módulos principales

Se realizará una breve descripción de cada una de las etapas del procesador.

4.1. `IF`

La etapa de *Instruction Fetch* corresponde a la búsqueda de la instrucción a la memoria para posteriormente ejecutarla.

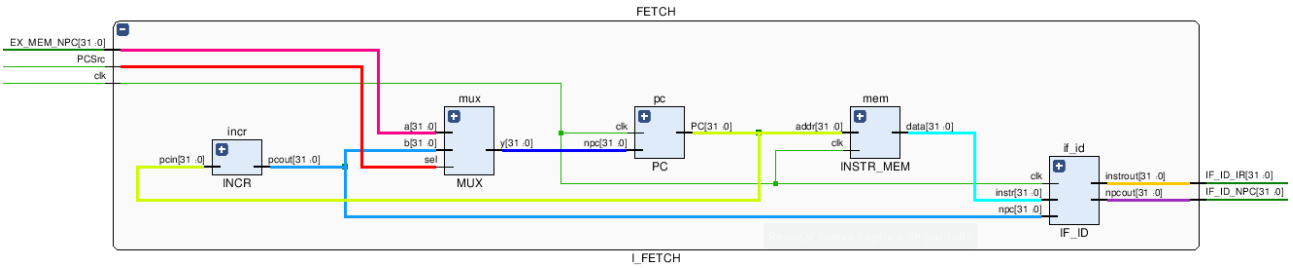


Imagen 1: Esquemático del *stage* `IF`.

4.1.1. `INCR`

Incrementa el valor del `PC`.

4.1.2. `MUX`

Este módulo a través del selector decide cual será el próximo valor del `PC`, en función a la operación realizada en el

4.1.3. `PC`

Asigna el valor al `PC`, además lo envía a `INCR` y `MEM`.

4.1.4. `MEM`

El módulo posee un arreglo de instrucciones que fueron cargadas previamente, entonces el `PC` sirve como índice para obtener la instrucción.

4.1.5. `IF_ID`

Es el *buffer* de salida, por tanto en el próximo flanco positivo de `clk`, se cargaran los valores haciendo avanzar la instrucción al siguiente *stage*. Recordemos que en el momento que la instrucción se encuentra disponible en algún *buffer* de salida también lo está para el siguiente *stage*.

4.1.6. Esquemático general e I/O

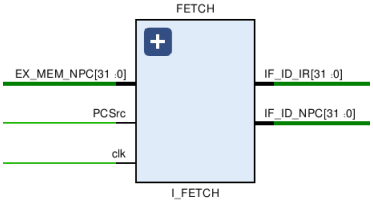


Imagen 2: Módulo `IF`.

En función al contador del programa (`PC`) se decide cual será la próxima instrucción leída desde memoria.

Entradas

- `EX_MEM_NPC`: valor de `PC` obtenido desde `ADDER` (ver sección 4.3.2) en `EX`.
- `PCSrc`: selector de `PC`.

Salidas

- `IF_ID_IR`: instrucción.
- `IF_ID_NPC`: valor del `PC`.

4.2. `ID`

La etapa de *Instruction Decode* corresponde a la decodificación de la instrucción.

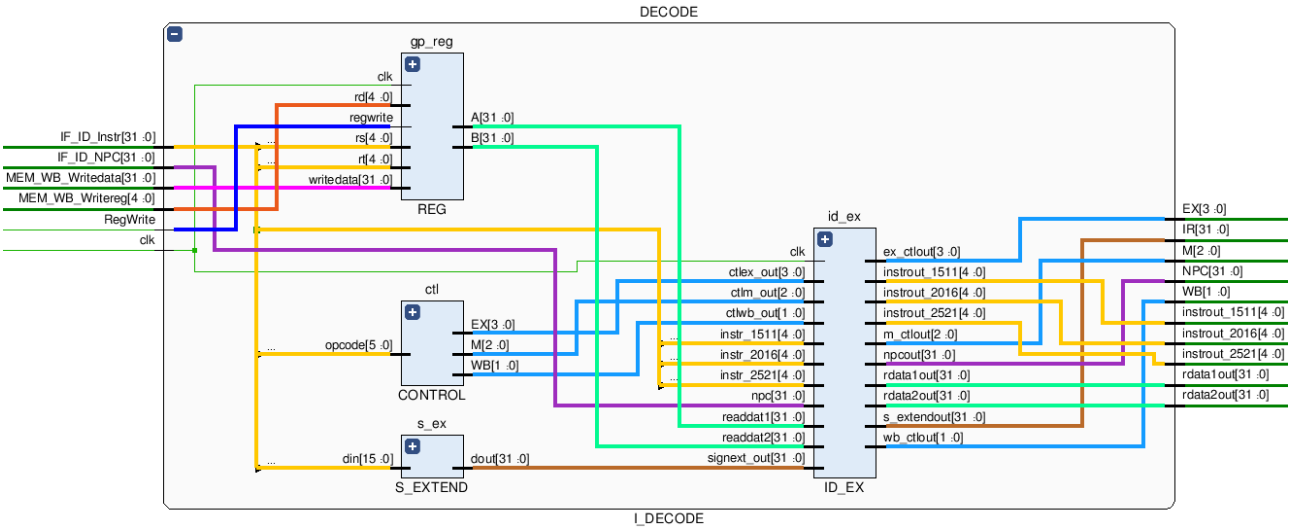


Imagen 3: Esquemático del *stage* `ID`.

4.2.1. `REG`

Es el módulo encargado de almacenar los valores en lo que sería equivalente a la memoria RAM. Para ello se vale de un arreglo de 32 posiciones de 32 bits cada uno.

Para la lectura de valores, *i.e.*, `A` y `B` se indexa con `rs` y `rt` respectivamente.

Sin embargo para escribir en la memoria es necesario que `regwrite` esté en alto, y así en el *rise edge* del `clk` se escribirán los datos de `writedata` en la pocisión indexada por `rd`.

4.2.2. `CONTROL`

Nos permite controlar los futuros *stages* a través de líneas de control que serán `EX`, `M` y `WB` respectivamente, en función del *opcode* de la instrucción.

Cada línea de control tiene una función especifica que se define en la siguiente Tabla 1.

Señal	0	1
RegDst	El número de registro destino para el registro Write viene del campo rd.	El número de registro destino para el registro Write viene del campo rt.
RegWrite	Nada.	El registro Write es escrito con el valor del registro WriteData.
ALUSrc	El segundo operando de la ALU viene de Read data 2.	El segundo operando de la ALU es el sign-extended, los 16 bits más bajos de la instrucción.
PCSrc	El PC es reemplazado por la salida del sumador que calcula PC+4.	El PC es reemplazado por la salida del sumador que calcula la rama objetivo.
MemRead	Nada.	El contenido de datos de memoria indexado por la dirección de entrada es puesto en la salida de Read Data.
MemWrite	Nada.	El contenido de datos de memoria indexado por la dirección de entrada es reemplazado por el valor de Write Data.
MemtoReg	El valor que alimenta al registro Write Data viene de la ALU.	El valor que alimenta al registro Write Data viene desde la memoria de datos.

Tabla 1: Función de cada señal de control.

Básicamente consiste en la implementación de un circuito combinacional de la Tabla 2.

Instruction		Líneas de control de la etapa de cálculo EX				Líneas de control de etapa de acceso a MEM			Líneas de control de la etapaWB	
		RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	000000	1	1	0	0	0	0	0	1	0
lw	100011	0	0	0	1	0	1	0	1	1
sw	101011	x	0	0	1	0	0	1	0	x
beq	000100	x	0	1	0	1	0	0	0	x

Tabla 2: Control de líneas

4.2.3. `S-EXTEND`

Incrementa el tamaño de la instrucción de 16bits a 32bits, replicando el bit de mayor orden en `[31:16]`.

4.2.4. `ID_EX`

Buffer de salida encargado de cargar los valores para la siguiente etapa, recordemos que se activa por *rise edge* de `clk`.

4.2.5. Esquemático general e I/O

El módulo `DECODE` nos provee funcionalidades para decodificar una instrucción, leer y escribir en RAM y finalmente controlar otras etapas a través de líneas de control.

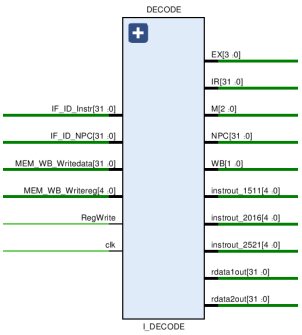


Imagen 4: Módulo `ID`.

Entradas

- `IF_ID_Instr`: instrucción proveniente de `IF`.
- `IF_ID_NPC`: valor del `PC`.
- `MEM_WB_Writedata`: dato a escribir en memoria.
- `MEM_WB_Writereg`: posición donde se escribirá el dato.
- `RegWrite`: habilitación de escritura.

Salidas

- `EX`, `M`, `WB`: líneas de control.
- `IR`: instrucción.
- `NPC`: valor del contador de programa.
- `instruot_1511`: `rd`.
- `instruot_2016`: `rt`.
- `instruot_2521`: `rs`.
- `rdata1out`: `A`.
- `rdata2out`: `B`.

4.3. EX

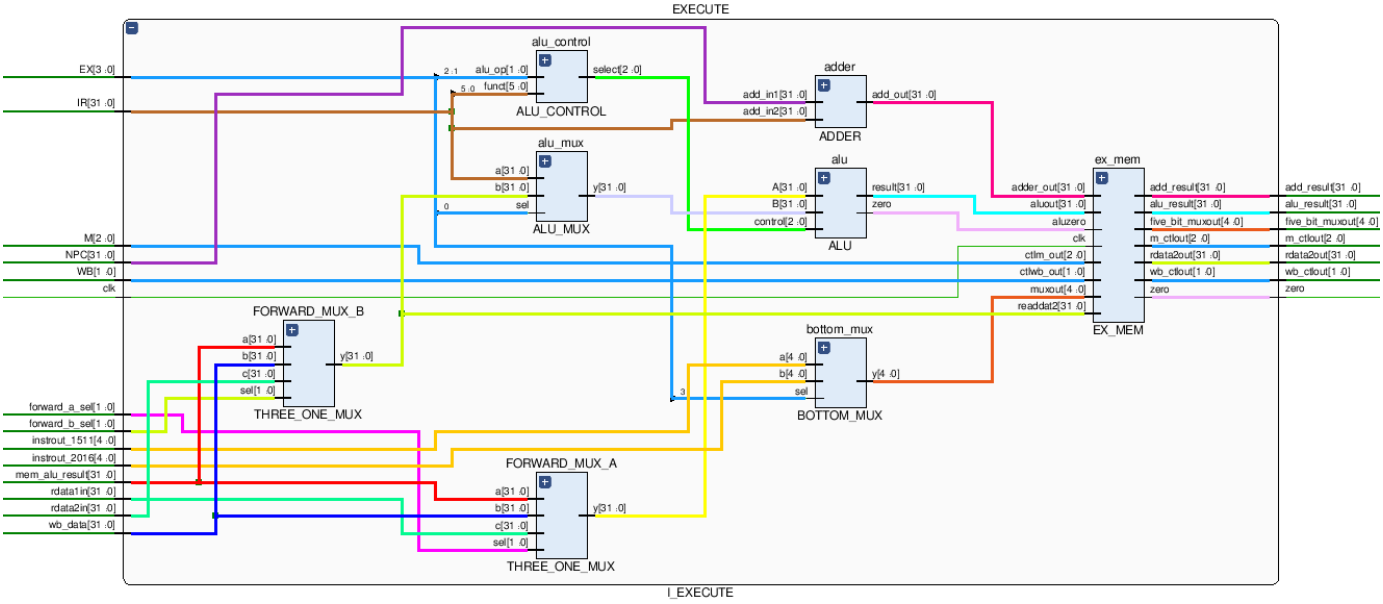


Imagen 5: Esquemático del *stage* `EX`.

4.3.1. ALU_CONTROL

Este módulo nos es útil para controlar la operación ejecutada por la ALU. Para esta labor utiliza `EX[2:1]` y el *opcode* que equivale a `IR[31:26]`, entradas de la Tabla 3 a implementar.

Opcode	ALUOp	Operación de la instrucción	Campo funct	Acción deseada de ALU	Entrada de control ALU
LW	00	load word	xxxxxx	add	0010
SW	00	store word	xxxxxx	add	0010
Branch equal	01	branch equal	xxxxxx	substract	0110
R-type	10	add	100000	add	0010
R-type	10	substract	100010	substract	0110
R-type	10	and	100100	and	0000
R-type	10	or	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Tabla 3: Tabla de control de ALU

4.3.2. ADDER

Retorna como salida el valor actual del `PC` al de la instrucción, lo que posiblemente si se dan las condiciones sea el nuevo valor del `PC`.

4.3.3. ALU_MUX

La `FU` (*Forwarding Unit*) es la encargada de los controles de peligros así como también la encargada de decidir los valores que operará la `ALU` ya que `A` está definida por `FORWARD_MUX_A` y `B` si bien se define por `sel` (`EX[0]`), `b` es provisto por `FORWARD_MUX_B`.

Esto se explica en mayor detalle en la sección 4.4.

4.3.4. ALU

La `ALU` es capaz de ejecutar las operaciones de suma, resta, `and`, `or` y si `A<B` entonces `1`, sino `0`.

Además provee un cable `zero` para las operaciones de rama.

4.3.5. BOTTOM_MUX

Este módulo puede consultarse en la Sección 4.4.2.

4.3.6. EX_MEM

Buffer de salida del módulo.

4.3.7. FORWARD_MUX_A

Consultar Tabla 4.

4.3.8. FORWARD_MUX_B

Consultar Tabla 5.

4.3.9. Esquemático general e I/O

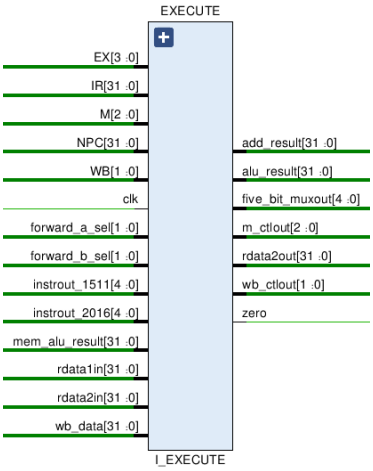


Imagen 6: Módulo `EX`.

Entradas

- `EX, M, W`: líneas de control.
- `NPC`: valor del `PC`.
- `IR`: instrucción.
- `forward`: entradas a los multiplexores de *forwarding*.
- `instrout_1511`: `rd`.
- `instrout_2016`: `rt`.
- `mem_alu_result`: resultado inmediato de la `ALU`.
- `rdata`: valores obtenidos desde `ID`.
- `rdata`: valor de `WB`.

Salidas

- `add_result`: resultado de `adder`.
- `alu_result`: resultado inmediato de la `ALU`.
- `five_bit_muxout`: si se dan las condiciones, `rd`.
- `m_cltout` y `wb_ctlout`: líneas de control.
- `zero`: útil para rama.
- `rdata2out`: valor desde `forward_mux_a`.

4.4. FORWARDING UNIT

En determinadas ocasiones en una etapa necesitamos un dato que aún no se ha terminado de procesar en otra, o quizás si pero aún el dato no ha sido guardado. Esto puede llevar a lo que se denomina parada o *stall* del *pipeline*, ya que necesitamos uno o más ciclos de reloj para obtener el dato. Esto acomete contra nuestro objetivo de rendimiento, por tanto necesitamos una manera de combatirlo.

Esto significa *e.g.* que cuando una instrucción intenta usar un registro en su etapa `EX`, una instrucción anterior intenta escribir en su etapa `WB`, pero en realidad necesitamos los valores como entradas a la `ALU` más que en memoria.

4.4.1. Condiciones de peligro

Peligro EX

```
if(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
if(EX/MEM.RegWrite) and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

Peligro MEM

```
if (MEM/WB.RegWrite && (MEM/WB.RegisterRd != 0)
& not(EX/MEM.RegWrite & (EX/MEM.RegisterRd != 0)
& (EX/MEM.RegisterRd != ID/EX.RegisterRs))
& (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
if (MEM/WB.RegWrite & (MEM/WB.RegisterRd != 0)
& not(EX/MEM.RegWrite & (EX/MEM.RegisterRd != 0)
& (EX/MEM.RegisterRd != ID/EX.RegisterRt))
& (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Estas condiciones serán implementadas a través de dos multiplexores, `ForwardA` y `ForwardB`.

`ForwardA`	**Source**	**Primer operando de la `ALU`**
00	`ID/EX`	Viene del archivo registro.
10	`EX/MEM`	Es adelantado desde el resultado previo de la `ALU`.
01	`MEM/WB`	Es adelantado desde la memoria de datos o un resultado anterior de la `ALU` (`MemToReg` decide en `WB`).

Tabla 4: Tabla del mux `ForwardA`.

`ForwardB`	**Source**	**Segundo operando de la `ALU`**
00	`ID/EX`	Viene del archivo registro.
10	`EX/MEM`	Es adelantado desde el resultado previo de la `ALU`.
01	`MEM/WB`	Es adelantado desde la memoria de datos o un resultado anterior de la `ALU` (`MemToReg` decide en `WB`).

Tabla 5: Tabla del mux `ForwardB`.

4.4.2. BOTTOM_MUX

Este módulo es útil para obtener `rd` en el *buffer* `EX_MEM`, debido a que utiliza como selector a `EX[3]` que equivale a `RegDst`. Cuando este valor es 1, entonces selecciona `rd` como salida del multiplexor.

Esto también es beneficioso porque esta condición conlleva que `Reg-Write` este habilitado por tanto se escribirá, como observamos en la Tabla 2.

4.4.3. Esquemático general e I/O

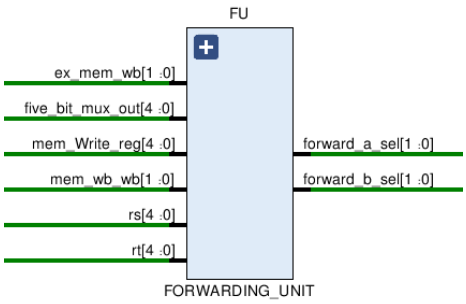


Imagen 7: Esquemático del módulo `FU`.

Entradas

- `ex_mem_wb[1] = EX/MEM.RegWrite`
- `five_bit_mux_out = EX/MEM.RegisterRd`
- `mem_Write_reg = MEM/WB.RegisterRd`
- `mem_wb_wb[1] = MEM/WB.RegWrite`
- `rs = ID/EX.RegisterRs`
- `rt = ID/EX.RegisterRt`

Salidas: las respectivas entradas a cada uno de los multiplexores ubicados en el *stage* `EX`.

- `forward_a_sel`
- `forward_b_sel`

4.5. MEM

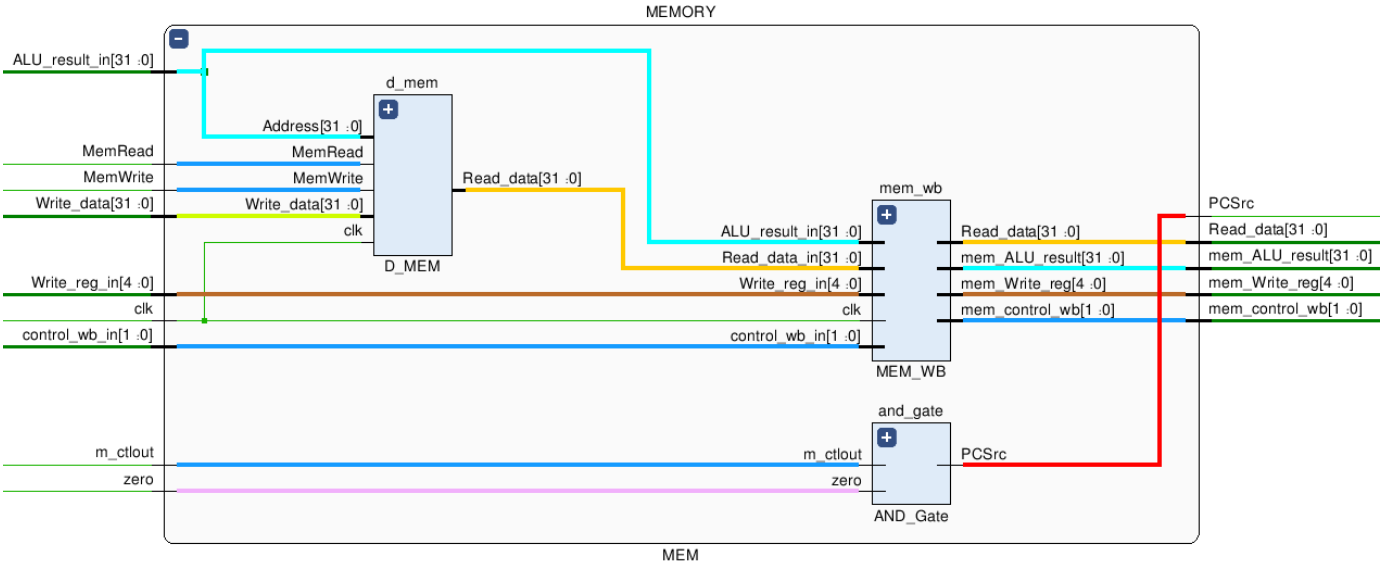


Imagen 8: Esquemático del *stage* `MEM`.

4.5.1. D_MEM

Equivale a la memoria ROM, posee un arreglo de 128 posiciones de 32 bits cada una.

Su funcionamiento es sencillo, `Address` funciona como índice del arreglo y tanto para leer como para escribir debemos setear los valores `MemWrite` como `MemRead` respectivamente como vimos en la Tabla 1.

4.5.2. MEM_WB

Buffer de salida del *stage*.

4.5.3. AND_Gate

Realiza la operación `M[2]&&zero`, (recordemos que `M[2]` es *branch* en la Tabla 2) obteniendo la salida `PCSrc` que luego será útil para decidir en el mux de `IF` cúal será el próximo `PC`.

4.5.4. Esquemático general e I/O

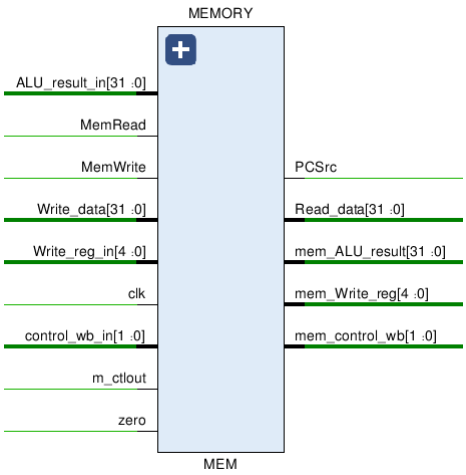


Imagen 9: Módulo MEM.

Entradas

- ALU_result_in: resultado de la ALU.
- MemRead: M[1].
- MemWrite: M[0].
- Write_Data: rdata2out.
- Write_reg_in: rd (EX[3]=1) o rt.
- control_wb_in: línea de control.
- m_ctlout: M[2].
- zero: proviene desde la ALU.

Salidas

- PCSrc: selector de mux en IF.
- Read_data: datos leídos.
- mem_ALU_results = ALU_result_in
- mem_Write_reg = Write_reg_in
- mem_control_wb = control_wb_in

4.6. WB

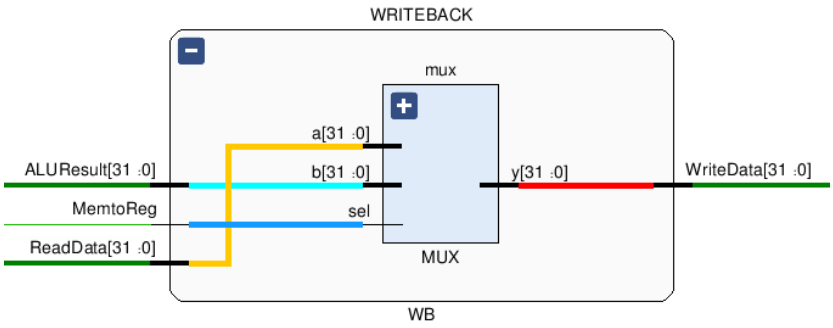


Imagen 10: Esquemático del *stage* `WB`.

El funcionamiento de este módulo es muy simple, básicamente decide a través del selector `MemtoReg` (Tabla 1) si `WriteData` será el valor leído desde memoria o el resultado de la `ALU`.

4.6.1. Esquemático general e I/O

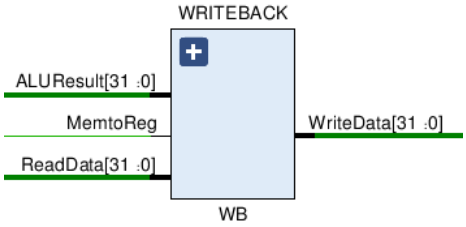


Imagen 11: Módulo `WB`.

Entradas

- `ALUResult`: resultado de la `ALU`.
- `MemtoReg`: `mem_control_wb[0]`.
- `ReadData`: datos desde la memoria.

Salidas

- `WriteData`: datos para `ID` y `EX`.

5. Conclusiones

`NAQV`

El trabajo en si mismo no presentaba una dificultad demasiado alta, sin embargo su desarrollo fue retrasado debía a que determinados pasos requerían modificaciones del *linker* o agregar determinadas librerías que no estaban en el proyecto en un primer momento. Pero una vez que se sorteados estos obstáculos el desarrollo fue mucho más veloz.

Fue una grata experiencia para adquirir roce con el desarrollo de sistemas operativos de tiempo real, se comprobó su funcionamiento y su potencial para las áreas para la cual ha sido desarrollada esta tecnología.

6. Apéndices

6.1. Modo de uso

6.1.1. Compilación

Para compilar el proyecto es necesario abrir la consola en el directorio `so2-tp4-rtos-GeraCollante` y ejecutar el comando `make`.

6.1.2. Clean

```
make clean
```

6.1.3. Ejecución

```
qemu-system-arm -machine lm3s811evb -cpu cortex-m3 -kernel
gcc/RTOSDemo.axf -serial stdio
```