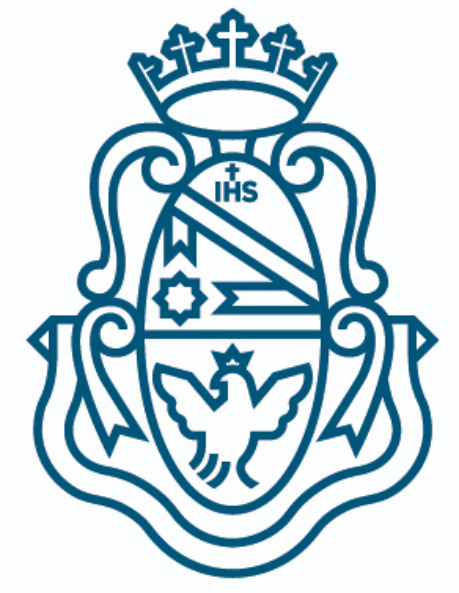
Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales



Arquitectura de Computadoras

Trabajo Práctico Final

Pipeline procesador MIPS simplificado

COLLANTE, Gerardo
QUINTEROS CASTILLA, Nicolás

Profesor:
Ing. Rodríguez, Martín

1 de octubre 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 siguientes 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.

Esquemático general e I/O

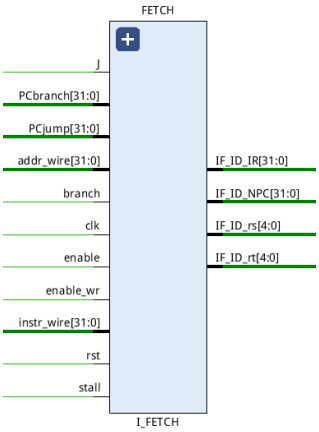


Imagen 1: Módulo `IF`.

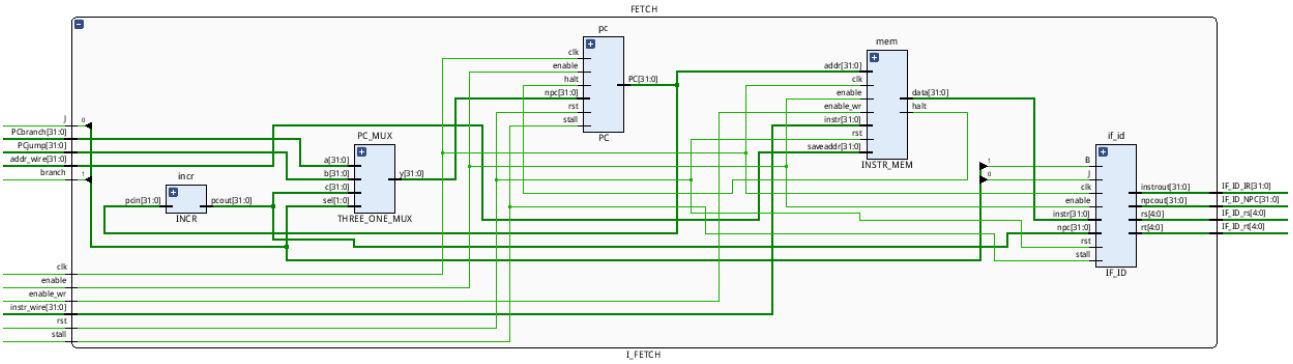


Imagen 2: Esquemático del *stage* `IF`.

4.1.1. `INCR`

Incrementa el valor del `PC`.

4.1.2. `PC_MUX`

Este módulo a través del selector `{J,branch}` decide cual será el próximo valor del `PC`, ya sea el siguiente `PC` incrementado, el valor siguiente de PC o el PC del salto o la ramificación.

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*.

Además posee las entradas `J` y `B`, para el caso donde sea necesario *flushear* una instrucción y evitar que esta entre al pipeline ya sea por un salto o ramificación.

También vale mencionar la entrada `stall`, que nos sirve para conservar los valores en el *buffer* aunque éste se encuentre habilitado realizando una parada del *pipeline*. Esta operación es necesaria para las instrucciones `lw` y `sw`.

4.2. `ID`

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

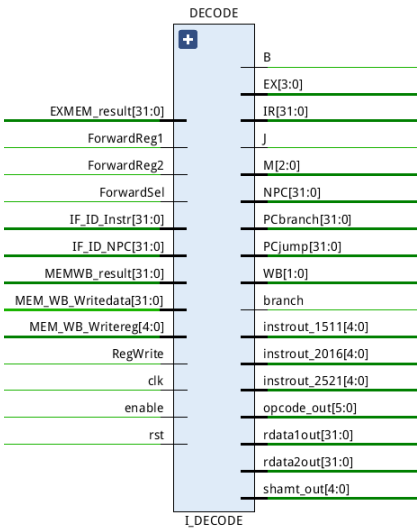


Imagen 3: Módulo `IF`.

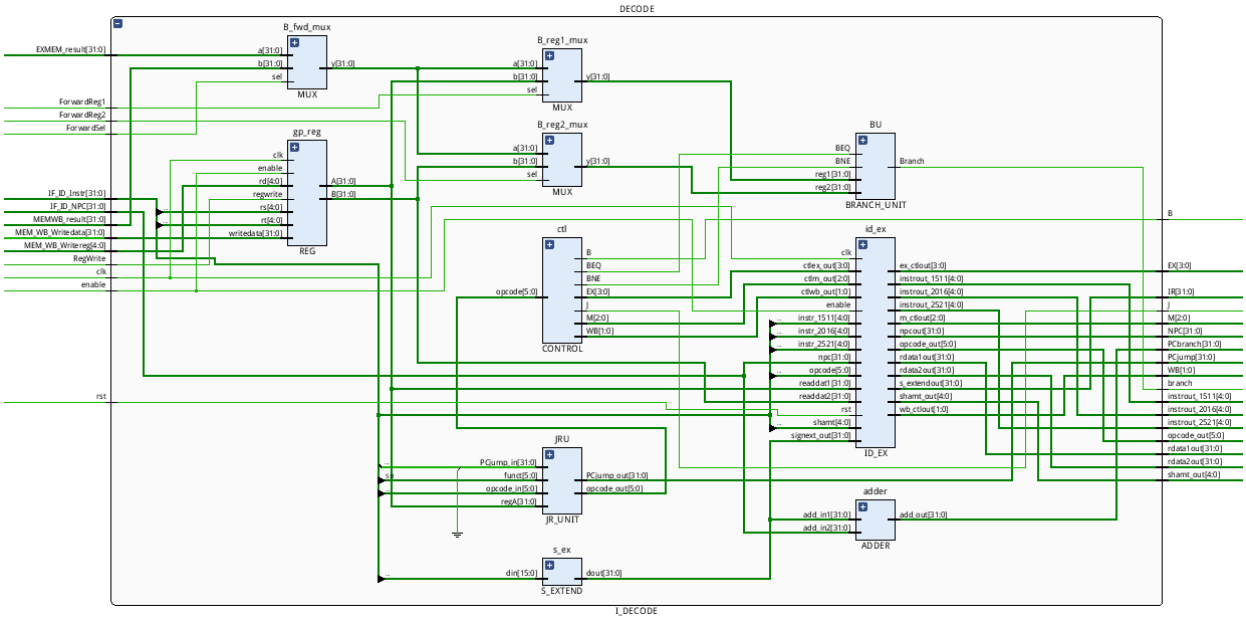


Imagen 4: Esquemático del *stage* `ID`.

4.2.1. REG

Es el módulo encargado de almacenar valores en registros de propósito genereal. 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.
`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.
`J`	No es una instrucción de salto.	Es una instrucción de salto, ya sea `J`, `JR` o `JAL`.
`B`	No es una instrucción de rama.	Es una instrucción de rama, ya sea `BNE` o `BEQ`.
`BNE`	No es una instrucción de rama.	Es la instrucción de rama `BNE`.
`BEQ`	No es una instrucción de rama.	Es la instrucción de rama `BEQ`.

Tabla 1: Función de cada señal de control.

4.2.3. JR_UNIT

Unidad encargada de verificar si la instrucción es `JR`, en caso que lo sea se encarga de *switchear* a través de un `mux` los 26 bits LSB por el valor indicado en el operando. Además cambia el `opcode` al de una operación `J` (recordemos que la instrucción `JR` posee `opcode = b000000`, por tanto podría ser malinterpretada como una operación `NOP`).

4.2.4. BRANCH_UNIT

Unidad encargada de realizar los cálculos para ver si una rama es tomada o no. Se apoya en multiplexores controlados a través de la `BRANCH FORWARDING UNIT` con el próposito de evitar peligros de datos.

4.2.5. S-EXTEND

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

4.2.6. ID_EX

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

4.3. EX

Este módulo es el encargado de realizar las diferentes operaciones con los operandos en la ALU.

Esquemático general e I/O

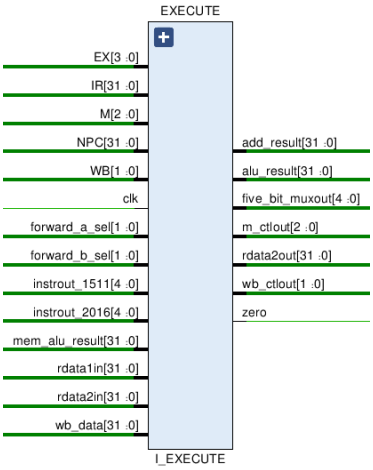


Imagen 5: Módulo `EX`.

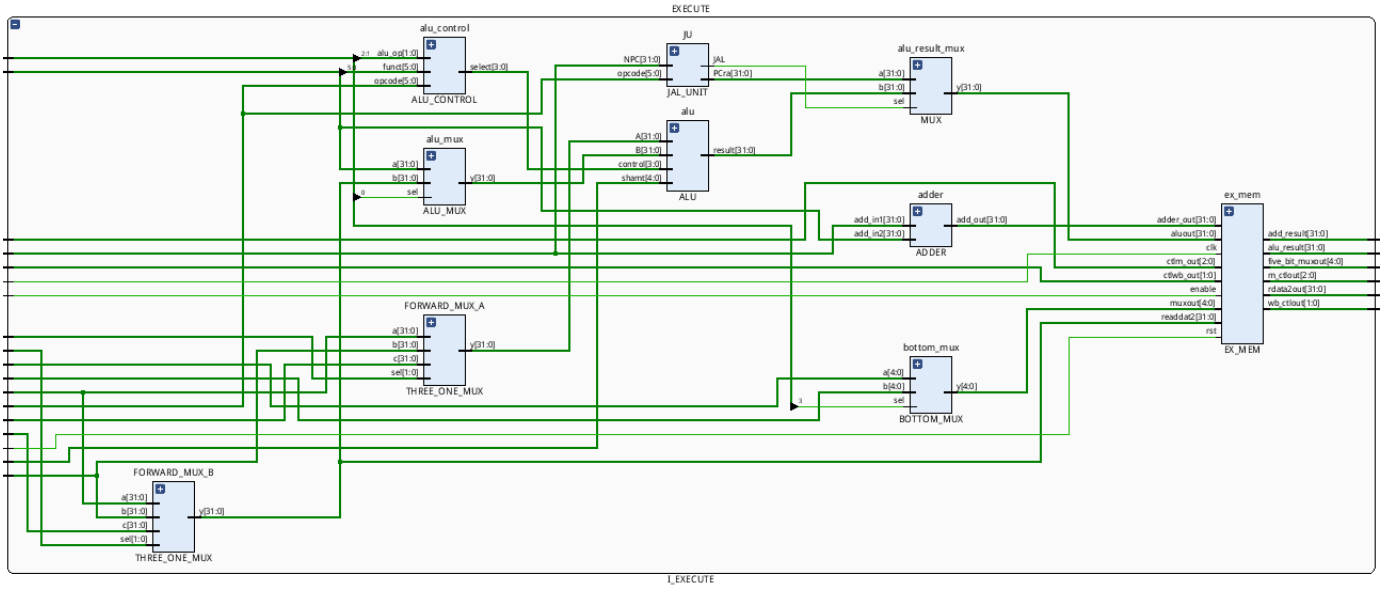


Imagen 6: Esquemático del *stage* `EX`.

4.3.1. `ALU_CONTROL`

Este módulo nos es útil para controlar la operación ejecutada por la ALU. Para esta labor se sirve del `opcode` y `funct` de la instrucción.

4.3.2. `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.

4.3.3. `ALU`

La `ALU` es capaz de ejecutar todas las operaciones indicadas en la sección 2.2, ya sean de *R-type* como de *I-type*.

4.3.4. `BOTTOM_MUX`

Este módulo puede consultarse en la Sección 4.4.2.

4.3.5. `JAL UNIT`

Unidad especial encargada de detectar una instrucción `JAL` con el objetivo de cargar el `PC` en el registro `$ra` (recordar que esta instrucción es complementaria a la instrucción `JR $ra`), apoyandose en el modulo `alu_result_mux` que es un multiplexor entre la salida de la ALU y el valor del `PC`. En caso que no sea detectado todo sigue su curso normal.

4.3.6. EX_MEM

Buffer de salida del módulo.

4.3.7. FORWARD_MUX_A

Consultar Tabla 2.

4.3.8. FORWARD_MUX_B

Consultar Tabla 3.

4.4. FORWARDING UNIT

Esquemático general e I/O

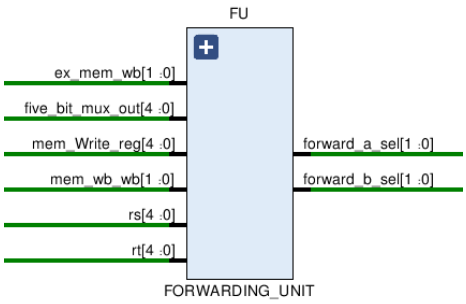


Imagen 7: Esquemático del módulo `FU`.

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 2: 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 3: 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.

4.5. MEM

Esquemático general e I/O

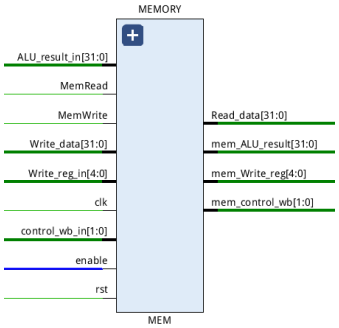


Imagen 8: Esquemático del módulo `MEM`.

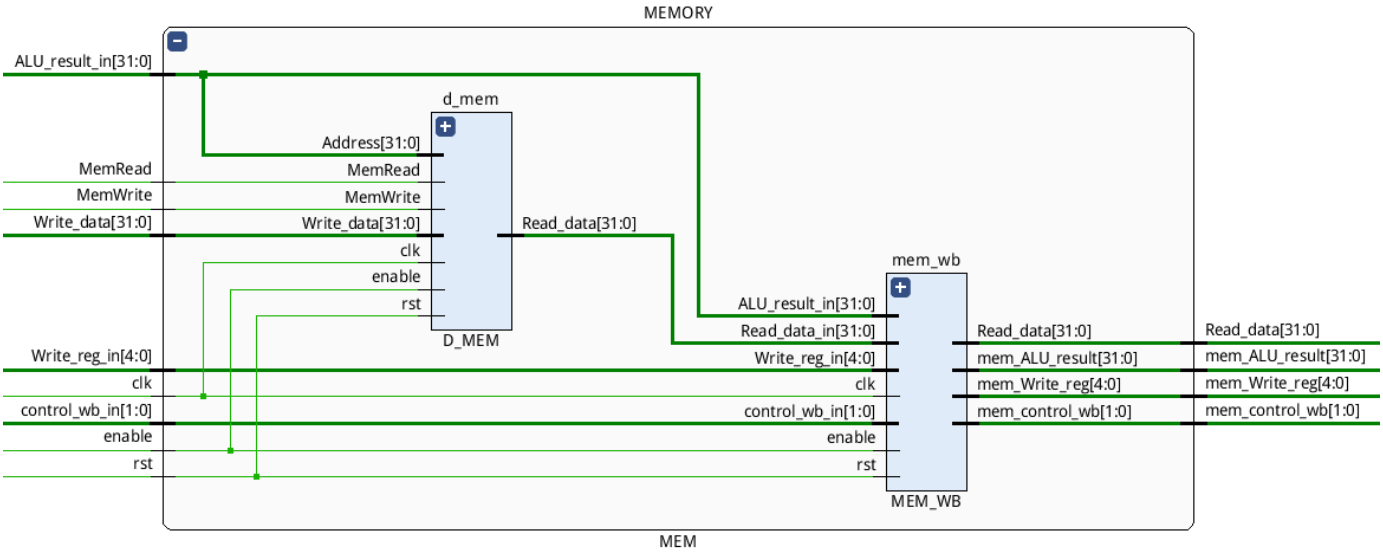


Imagen 9: Esquemático del *stage* `MEM`.

4.5.1. `D_MEM`

Equivale a la memoria RAM, 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*.

4.6. `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`.

Esquemático general e I/O

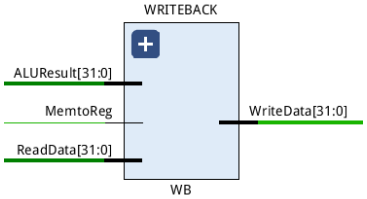


Imagen 10: Módulo `WB`.

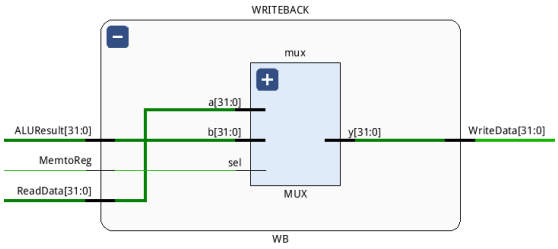


Imagen 11: Esquemático del *stage* `WB`.

4.7. DEBUG UNIT

Debido a la imposibilidad de utilizar la placa físicamente se solicitó la creación de una unidad de *debugging* cuyo propósito era el de emular el paso a paso del *pipeline* siendo cada paso accionado a través de un botón.

Esto generaba un pulso `enable` que es entrada de cada uno de los *buffers* de salida de cada etapa. Así para avanzar en el *pipeline* era condición necesaria que la señal `enable` estuviera en alto además de la señal del `clk`.

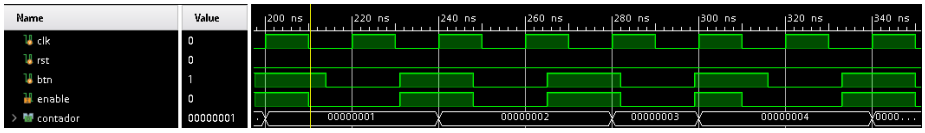


Imagen 12: Comportamiento de las señales.

4.8. HAZARD DETECTION UNIT

Su tarea es detectar un peligro de datos que sucedería en el caso que una función `lw` o `sw` necesite un dato que aún no se encuentra disponible.

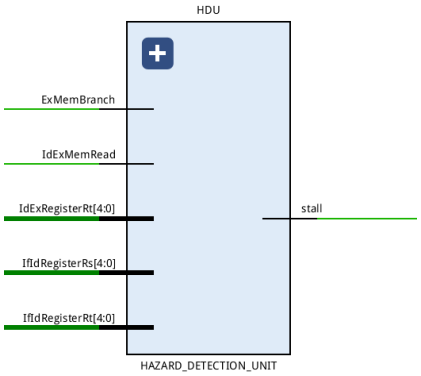


Imagen 13: Comportamiento de las señales.

En el caso que eso ocurra se realiza una parada congelando el *buffer* `IF_ID` para que no avance permitiendo así avanzar a las demás etapas del *pipeline* y se comprueban varias condiciones:

```
if(IdExMemRead &&
(IdExRegRt == IfIdRegRs) || (IdExRegRt == IfIdRegRt))
stall the pipeline
```

4.9. BRANCH FORWARDING UNIT

Se tomó la decisión de ejecutar las operaciones de rama en el stage `ID` debido a que esto sólo necesitaba realizar en una ocasión *flush*, en vez de tres que es lo que hubiera sucedido si realizabamos estas operaciones en el stage `MEM` como se propone en el libro *Arquitectura de Computadoras* de *Hennesy & Patterson*.

El inconveniente con realizar estas operaciones en `ID` es que esto puede llevar a riesgos de datos que debemos solucionar.

Básicamente los riesgos que podemos sufrir son desde los *buffers* `EX_MEM`, `MEM_WB` e incluso `ID_EX` (pero este caso no será analizado porque es salvado por la `HDU`).

4.9.1. *Forwarding* desde EX_MEM

Podemos observar que la operación 2 (`or`) realiza una operación y guarda el resultado en el operando `$v0`, pero el inconveniente es que aún ese resultado no se guardó y `beq` lo necesita, por tanto hacemos *forwarding*.

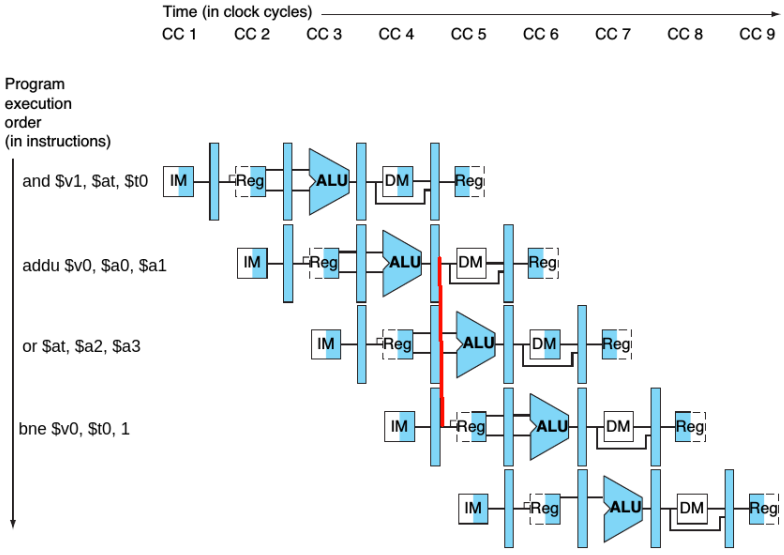


Imagen 14: Forwarding desde `EX_MEM`.

En este caso, en el momento que el resultado es guardado es solicitado por `bne`, por tanto debemos realizar un *forwarding* desde `MEM_WB`.

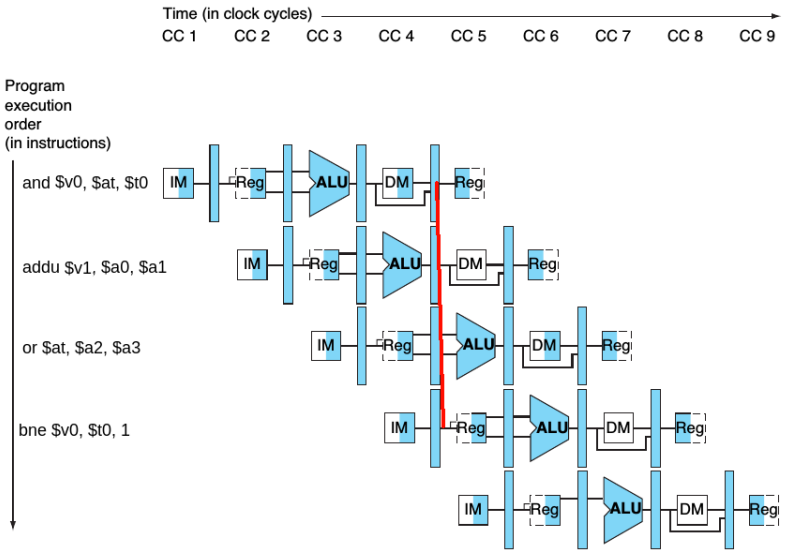


Imagen 15: Forwarding desde `MEM_WB`.

4.9.2. Funcionamiento

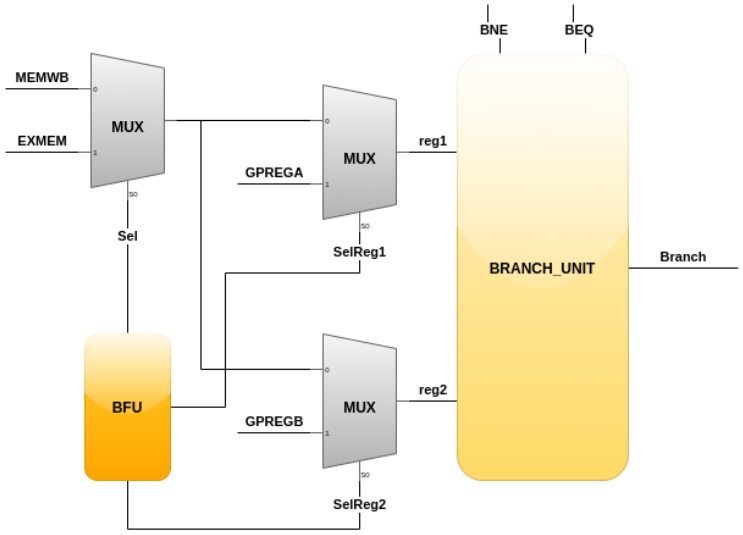


Imagen 16: Forwarding desde `MEM_WB`.

El funcionamiento de la `BFU` se resume en el siguiente gráfico donde observamos que es la encargada a través de los selectores de los multiplexores de elegir los operandos `reg1` y `reg2` que serán necesarios en el cálculo para decidir si la rama será tomada o no.

5. Conclusiones

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
```