MIPS SEGMENTADO

por Emmanuel Luján

Autor: Emmanuel Luján

Fecha:	*Diciembre del 2008*
Tema:	*MIPS Segmentado*
Materia:	*Técnicas Avanzadas de Diseño de Sistemas Digitales*
Universidad:	*Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina.*
Docente:	*Dr. Elias Todorivich*
Autor:	*Emmanuel Luján*
Correo electrónico:	*info@emmanuellujan.com.ar*
Sitio web:	*www.emmanuellujan.com.ar*

LEGALES

OBJETIVOS

Objetivos principales:

MIPS Segmentado: El objetivo es implementar un microprocesador MIPS en VHDL. En concreto, se va a realizar la versión segmentada del microprocesador. El resultado debe ser un micro capaz de realizar en el caso ideal (sin riesgos) una instrucción por ciclo de reloj. Para el ejercicio básico, no es necesario que el modelo soporte riesgos (salvo el estructural de acceso a memorias separadas de instrucciones y datos).
No se pide que el microprocesador soporte todo el juego de instrucciones completo, sino las siguientes instrucciones: add, sub, and, or, lw, sw, slt, lui y beq. En cualquier caso, la instrucción beq, que implica riesgos de control por ser un salto, funcionará "anómalamente" en la versión básica del ejercicio obligatorio. Todos los registros deben resetearse asíncronamente y funcionar por flanco de subida del reloj.

Calidad del código: Se deberán usar diversas técnicas que demuestren la calidad del código vhdl y se deberá presentar una enumeración de ellas.

Objetivos secundarios:

Traductor: Con fin de lograr mayor dinamicidad a la hora de hacer pruebas sobre el procesador se pide la construcción de un traductor básico. El mismo debe pasar el código assembler de las instrucciones planteadas a un archivo vhdl que pueda sintetizarse como una memoria rom. Particularmente dicha memoria se usará como memoria de instrucción del MIPS.

Algoritmo de prueba: Con motivo de probar la funcionalidad del procesador se pide implementar con el juego de instrucciones planteado el algoritmo de división Restoring. Dicho código debe poder ser traducido con la aplicación del inciso anterior.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

INTRODUCCIÓN

1.1.- SEGMENTACIÓN

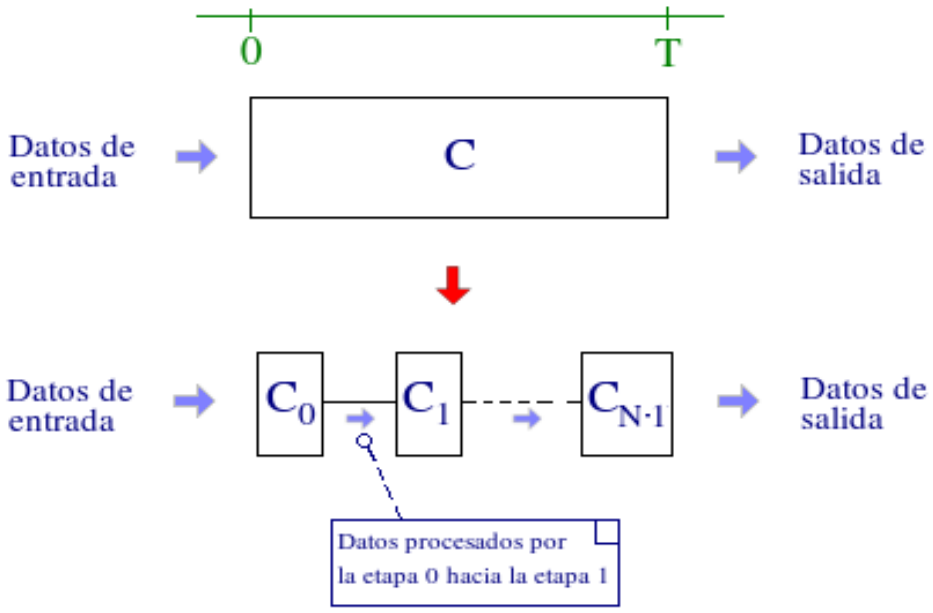
Existen dos maneras de acelerar un circuito combinacional: "Paralelismo espacial (replicación de hardware)" y "Pipelines (Cañerías)" . Nos concentraremos en esta última.

Comenzaremos con una definición muy general: la implementación de un pipeline a un circuito combinacional ocurre al segmentar o dividir dicho circuito en N etapas de procesamiento. Dónde los resultados procesados por una etapa son la entrada de la siguiente. Los resultados finales deberán ser los mismos que en el circuito original, pero éstos se arrojarán a una mayor velocidad.

La velocidad de procesamiento, V = cantidad de datos procesados por unidad de tiempo

(*otros nombres: Throghput, Ancho de banda, Producción.)

En base a esto definiremos: *Período de un circuito, P = 1 / V*



La velocidad Figura 1.1 – Concepto de segmentación

Analicemos un poco dónde se encuentra la mejora de la segmentación: el circuito C tarda un tiempo T en procesar un dato:

$$V(C) = 1 \text{ dato} / T$$

Dados M datos,

*el tiempo total de procesamiento de C, Tp(C) = M * T*

Ahora bien, la segmentación deberá proveer un tiempo más bajo para mejorar esto.

Debido a que cada etapa del circuito segmentado es un subconjunto del circuito original (más lógica adicional potencialmente) cada una tarda un tiempo Ti < T en procesar un dato.

En términos generales, cuando una dato ingresa a la primer etapa éste es procesado por la misma. Luego, dicho dato ahora pre-procesado por la primer etapa, pasa a la segunda. Pero mientras que el primer dato esta siendo procesado por la segunda etapa, el segundo dato esta siendo procesado por la primera. Esta situación se repite hasta que todos los datos son procesados por la última etapa.

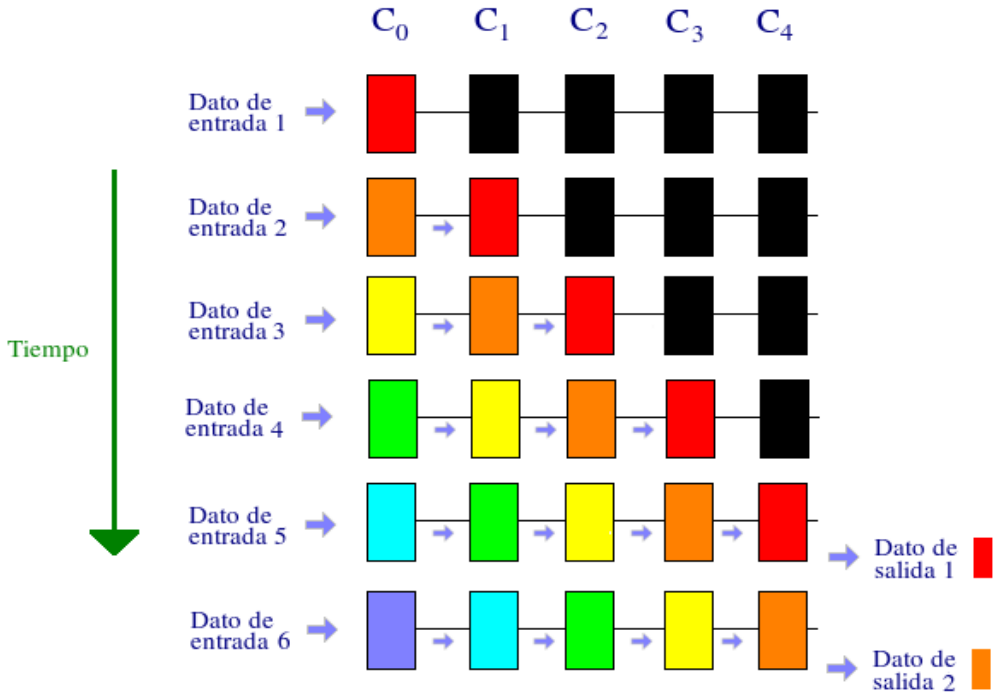


Figura 1.2 – Etapas de la segmentación

La situación anterior plantea un problema. Debido a que los Ti no son necesariamente iguales, las etapas están desincrinozadas. Veamos un ejemplo de mal funcionamiento, si la etapa C0 tarda un tiempo T0 y la etapa C1 tarda un tiempo T1 y T0<T1, siguiendo lo expuesto en el párrafo anterior el segundo dato estará listo para ser procesado por la segunda etapa antes de que se haya terminado de procesar el primer dato en esta segunda etapa. La segunda etapa dejaría de calcular el primer dato y comenzaría a calcular el segundo. Obviamente esto es una efecto indeseable ya que el primer dato nunca se calcularía. Para solucionar este tipo de inconvenientes se hará que las etapas que tardan menos esperen a las que tardan más, de hecho toda etapa tardará, mediante la incorporación retrasos, un tiempo igual a la etapa más lenta, o sea al Ti máximo. En otras palabras se sincronizará el circuito.

Ahora estamos en condiciones de replantearnos este problema. El primer dato será procesado por la última etapa luego de un tiempo igual a N veces la duración de una etapa sincronizada:

Tiempo de procesamiento de una etapa sincronizada, Te = max(Ti)

*Tiempo de llenado del pipe, Tl = N * Te*

Eso significa que inmediatamente después de transcurrido el tiempo *Tl* se obtiene en la salida el primer dato completamente procesado. Y luego, inmediatamente después del tiempo *Tl+Te* se obtiene en la salida el segundo dato completamente procesado. El tercero se obtendrá inmediatamente después de *Tl+2*Te*. En un sentido más general la velocidad de procesamiento del circuito segmentado luego del tiempo de llenado es:

V(Cseg) = 1 dato / Te

**notamos que V(Cseg) > V(C).*

Luego, dados M datos

*el tiempo total de procesamiento del circuito segmentado, Tp(Cseg) = Tl + Te * (M-1)*

ya que *Tl* es lo que tarda el primer dato, y con una velocidad de 1 dato cada un tiempo Te luego del priemero, los M-1 datos restantes tardarán Te *(M-1).

Comparemos con el tiempo de procesamiento del circuito original:

Sabiendo que,

N > 1, de otro modo carece de sentido segmentar.

M > 0, de otro modo carece de sentido procesar datos.

M es muy grande.

Ti,Te,T > 0 por ser tiempos

Ti < T como se explicó anteriormente
=> max(Ti) < T => Te < T

Entonces,

- *Te < T*
  => 1 < T/Te
  => 0 < T/Te – 1

  Ya que, lim (N/M + 1/M) cuando M->inf = 0

  para M suficientemente grande (debido a que el volumen de datos es muy grande): N/M + 1/M < T/Te – 1

  => (N + 1)/M < (T - Te) /Te
  => Te (N + 1)< M (T - Te)
  => 0< M (T - Te) - Te (N + 1)
  *=> 0< M (T - Te) - Te * N + Te*
  *=> 0< T * M - Te * N - Te *M + Te*
  *=> 0 < T * M - Te * (N +M-1)*
  *=> Te * (N +M-1) < T * M*
  *=> Te * N +Te * (M-1) < T * M*
  *=> Tl +Te * (M-1) < T * M*

  => T(Cseg) < T(C)

1.2.- ACERCA DE LOS MIPS

Con el nombre de MIPS (siglas de *M*icroprocessor without *I*nterlocked *P*ipeline *S*tages) se conoce a toda una familia de microprocesadores de arquitectura RISC desarrollados por MIPS Technologies.

Los diseños del MIPS son utilizados en la línea de productos informáticos de SGI; en muchos sistemas integrados; en dispositivos para Windows CE; routers Cisco; y videoconsolas como la Nintendo 64 o las Sony PlayStation, PlayStation 2 y PlayStation Portable.

Las primeras arquitecturas MIPS fueron implementadas en 32 bits, si bien versiones posteriores fueron implementadas en 64 bits. Existen cinco revisiones compatibles hacia atrás del conjunto de instrucciones del MIPS, llamadas `MIPS I`, `MIPS II`, `MIPS III`, `MIPS IV` y `MIPS 32/64`. Asimismo están disponibles varias "extensiones", tales como la `MIPS-3D` consistente en un simple conjunto de instrucciones SIMD en punto flotante dedicadas a tareas 3D comunes, la `MDMX(MaDMaX)` compuesta por un conjunto más extenso de instrucciones SIMD enteras que utilizan los registros de punto flotante de 64 bits, la `MIPS16` que añade compresión al flujo de instrucciones para hacer que los programas ocupen menos espacio (presuntamente como respuesta a la tecnología de compresión Thumb de la arquitectura ARM) o la reciente `MIPS MT` que añade funcionalidades multithreading similares a la tecnología HyperThreading de los procesadores Intel Pentium 4.

Debido a que los diseñadores crearon un conjunto de instrucciones tan claro, los cursos sobre arquitectura de computadores en universidades y escuelas técnicas a menudo se basan en la arquitectura MIPS. El diseño de la familia de CPU's MIPS influiría de manera importante en otras arquitecturas RISC posteriores como los DEC Alpha.

1.2.- MIPS SEGMENTADO

La versión de MIPS que se dará tratamiento en este informe corresponde al MIPS Segmentado. Su diseño es expuesto en la siguiente figura:

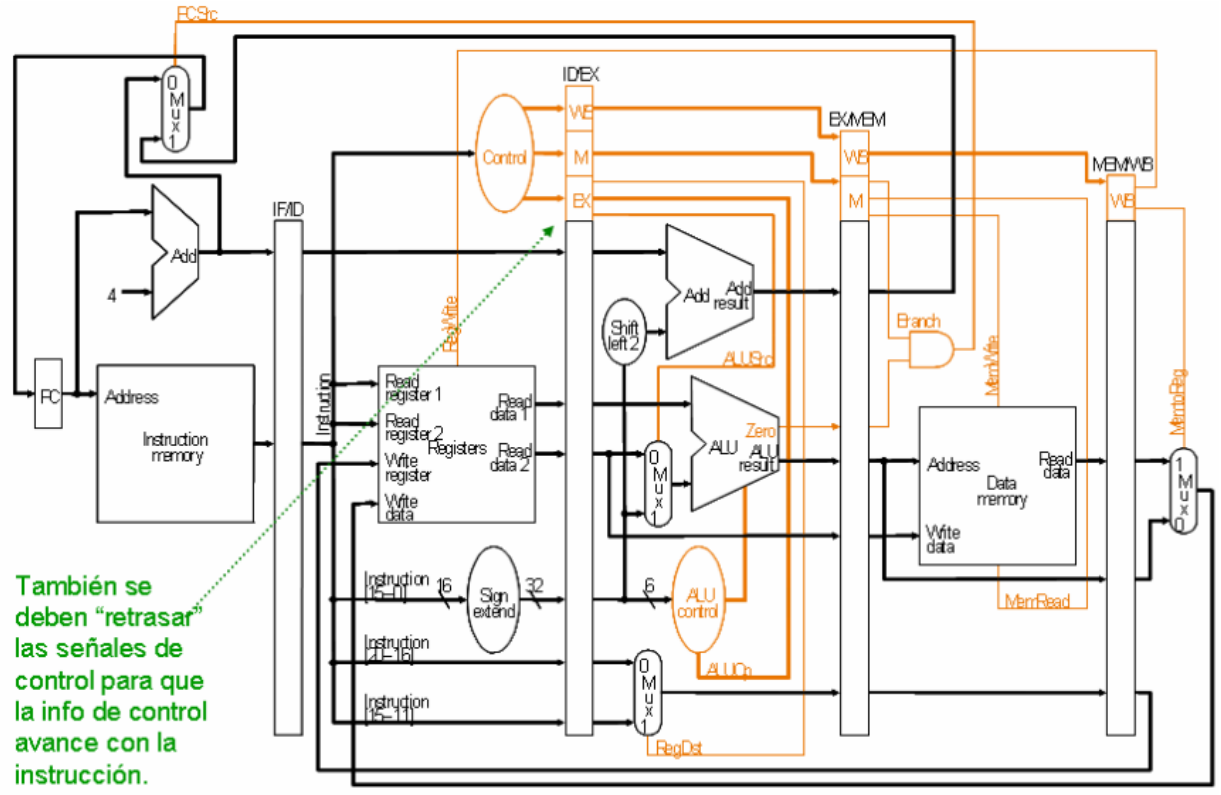


Figura 1.3 – MIPS Segmentado

IMPLEMENTACIÓN VHDL

2.1.- DISEÑO GENERAL

Se escoge una visión top-down para explicar el diseño en vhdl. La entidad top-level fue el test bench, quien usa sólo a la entidad SEGMENTED_MIPS estimulando sus dos únicas señales de entrada: Clock y Reset. El test bench no tiene entradas.

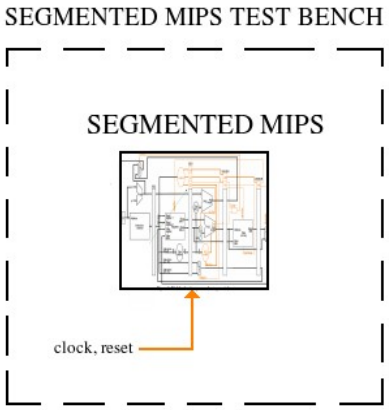


Figura 1.4 – Entidad Test bench

La entidad Segmented MIPS, hace uso de cinco entidades: Instruction Fetching, Instruction Decoding, Execution, Memory Access y Write Back; conectando sus entradas y sus salidas según el diseño preestablecido del procesador MIPS. No se agrega lógica adicional, sólo las conexiones entre las etapas y las conexiones de cada etapa con el Reset y el Clock.

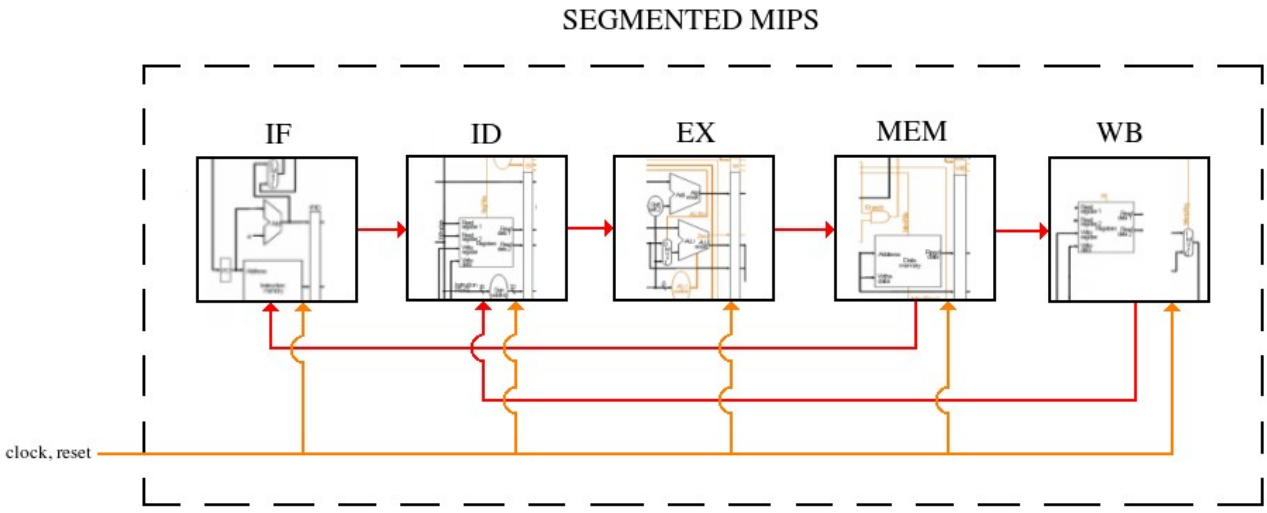


Figura 1.5 – Entidad Segmented Mips

Cada etapa se compone de los componentes necesarios para su funcionamiento, que son: otras entidades, lógica adicional y sus registros de sincronización.

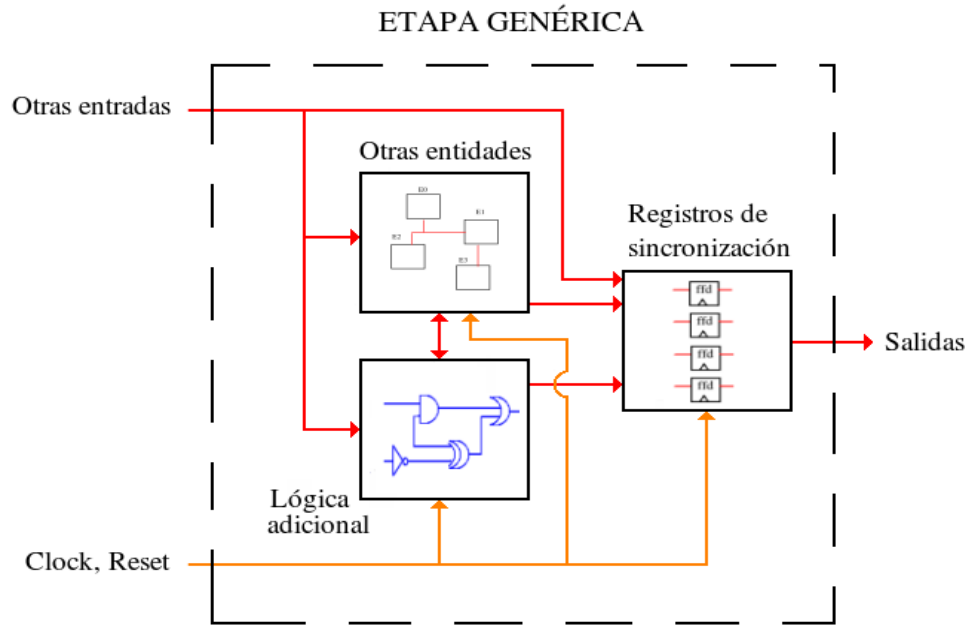


Figura 1.6 – Entidad de una etapa general

Esta etapa genérica contempla todos los casos de diseño de las etapas. Las mismas generalmente son más restrictivas

- Los tres componentes usan las entradas de la etapa. Los registro de sincronización sólo el Clock y el Reset, y las demás pueden usar todas las entradas.
- La lógica adicional, en caso de que existiere :
  - Se relaciona con las otras entidades, realizando algún cálculo para proveer una entrada a las mismas (por ejemplo el shift a izquierda de la etapa de execution).
  - Se relaciona con los registros de sincronización, realizando algún cálculo en base a las entradas de la etapa y proveyendo entradas para los mismos (por ejemplo la extensión de signo de instruction decoding).
  - Otros casos son posibles pero no se dan en el MIPS tratado.
- Las otras entidades, existen en todas las etapas, y :
  - Se relacionan con los registros, generando entradas a los mismos en base a las entradas de la etapa,
  - Se relacionan con la lógica adicional de la manera antes expuesta.
- Los registros de sincronización existen en todas las etapas excepto la última.
  - Se relacionan con los demás componentes de la manera antes expuesta.

2.2.- ANÁLISIS POR ETAPA

A continuación se mencionará el una descripción general del funcionamiento de cada etapa y la implementación de sus componentes. Antes de comenzar debemos notar que todas las entidades de sincronización de datos responden a la misma forma de diseño, por lo que se evaluarán al finalizar los análisis para no redundar en explicaciones.

Otro aspecto en común es que el procesamiento de todas las etapas se produce antes del próximo flanco ascendente de reloj.

2.2.1. - Instruction Fetching (Búsqueda de Instrucción)

Descripción general: al levantarse el flanco de reloj (rising edge) PC dejará salir la dirección de la instrucción actual. Por un lado la misma será la entrada de la Memoria de Instrucción, quien emitirá como salida la próxima instrucción.

Por otro lado dicha dirección será la entrada de un sumador, el cual arrojará como salida esa dirección más cuatro. Esta es la potencial (debido a que puede haber un salto) dirección de la siguiente instrucción. Se cuenta de a cuatro debido a la forma en que está implementada la Memoria de Instrucción, cada instrucción ocupa cuatro bytes y la memoria provee acceso de a un byte.

Luego la potencial próxima dirección es filtrada por un mux en conjunto con una entrada asincrónica y en base a un selector, PCSrc, que también es una entrada asincrónica. La otra dirección es una dirección de salto, y el selector es una orden de la unidad de control frente al evento de saltar. Si el selector esta desactivado quien pasa por el mux y entra a PC es la dirección calculada por el sumador. Si esta activado, hubo un salto, y quien pasa el la dirección de salto.

La salida del sumador y de la Memoria de Instrucción pasan a la etapa siguiente.

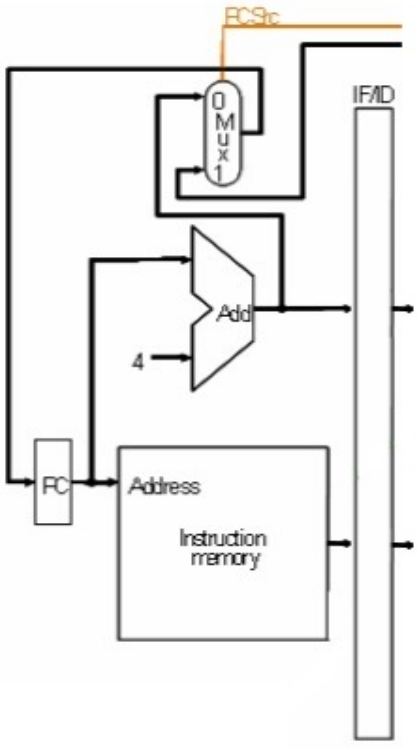


Figura 2.1 – Instruction fetching

Descripción de componentes:

Lógica adicional:

- Mux convencional. Implementación vhdl:

```
MUX_PC: process(PCSrc_IN,PC_ADDR_AUX2,NEW_PC_ADDR_IN)
begin
        if( PCSrc_IN = '0') then
                PC_ADDR_AUX3 <= PC_ADDR_AUX2(N-1 downto 0);
        else
                PC_ADDR_AUX3 <= NEW_PC_ADDR_IN;
        end if;
end process MUX_PC;
```

Otras entidades:

- Sumador: sumador de 32 bits construido en base a full adder's convencionales. Implementación vhdl: en base a generate de full adders.

```
BEGIN_FA:FULL_ADDER port map (
        X       => X(0),
        Y       => Y(0),
        CIN     => CIN,
        COUT    => CAUX(0),
        R       => R(0)
);
GEN_ADDER:  for i in 1 to N-1 generate
                NEXT_FA: FULL_ADDER port map (
                        X       => X(i),
                        Y       => Y(i),
                        CIN     => CAUX(i-1),
                        COUT    => CAUX(i),
                        R       => R(i)
                );
            end generate;
COUT <= CAUX(N-1);
```

- PC. Implementación vhdl: registro genérico instanciado a 32 bits con reset asincrónico.

```
REG:    process(CLK_IN,RESET_IN,DATA_IN)
        begin
                if(RESET_IN = '1') then
                        DATA_OUT <= (others => '0');
                elsif rising_edge(CLK_IN) then
                        DATA_OUT <= DATA_IN;
                end if;
        end process;
```

- Memoria de Instrucción (o de Programa): Memoria rom con el contenido de las instrucciones que ejecutará el procesador. El traductor explicado posteriormente se encarga de pasar código assembler a código vhdl. Dicho código vhdl es anexado a la rom aquí expuesta.

  Según el diseño del MIPS tiene un tamaño de 2^{32} instrucciones de 32 bits cada una. Debido a limitaciones de implementación relacionadas con la herramienta usada se restringe a la memoria a 1024 instrucciones de 32 bits cada una.

Implementación vhdl, se implementa un case dentro de un proceso:

```
case READ_ADDR is
        when "00000000000000000000000000000000" =>
                INST <= "00000001010000000010000000100000";
        when "00000000000000000000000000000100" =>
                INST <= "00000000000000000000000000100010";
        when "00000000000000000000000000001000" =>
                INST <= "00000000000000000000000000100010";
        when "00000000000000000000000000001100" =>
        ...
        when others =>
                INST <= "11111111111111111111111111111111";
end case;
```

2.2.2. -Instruction Decoding (Decodificación de Instrucción)

Descripción general: al producirse un flanco ascendente los registros de sincronización de la etapa anterior actualizan las señales a ser procesadas. La próxima instrucción (la salida del sumador) pasa directamente a los registros de sincronización. Esta será procesada por otra etapa en caso de que haya un salto.

La instrucción propiamente dicha (la salida de la Memoria de Instrucción) es separada en sus campos. Los tipos de instrucciones a los cuales daremos tratamiento son:

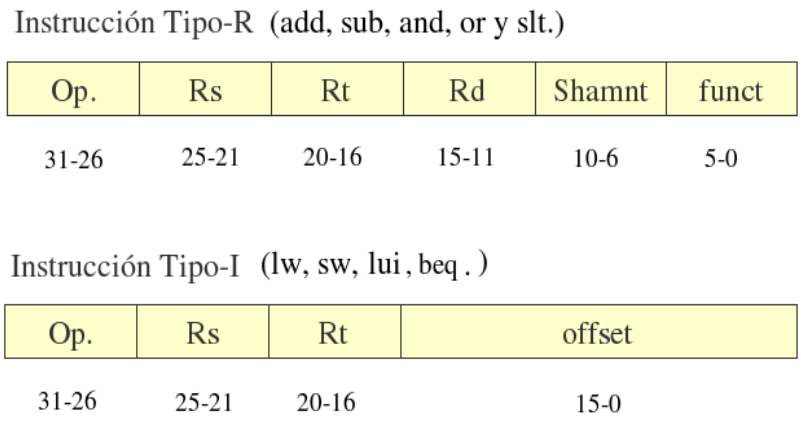


Figura 2.2 – Tipos de instrucciones

Dichos campos son las entradas de tres partes de procesamiento dentro de la etapa.

El primero es la Unidad de Control, quien se encargará de calcular las órdenes a las cuales responderá la Ruta de datos (Data Path) para ejecutar la instrucción.

El segundo es el Banco de Registros. Rs y Rt, quienes son direcciones de registros, se conectan a Read Register 1 y 2 respectivamente. En caso de producirse una lectura, cuando RegWrite esta desactivado, Read Data 1 y 2 actualizan sus salidas con el contenido de los registros direccionados por Rs y Rt. En caso de una escritura, hecho que ocurre en la etapa WB, cuando RegWrite esta activado, las dos entradas asincrónicas de la etapa dicen dónde se va a guardar el nuevo dato, Write Register, y cual es el dato guardarse, Write Data.

Por último se hace una extensión de signo para el campo Offset (Instrucción[15-0]).

Las salidas de la Unidad de Control, la próxima instrucción, las salidas del Banco de Registros, el Offset extendido y los componentes de la instrucción Rt (Instrucción[20-16]) y Rd (Instrucción[15-11]) son enviados a los registros de sincronización.

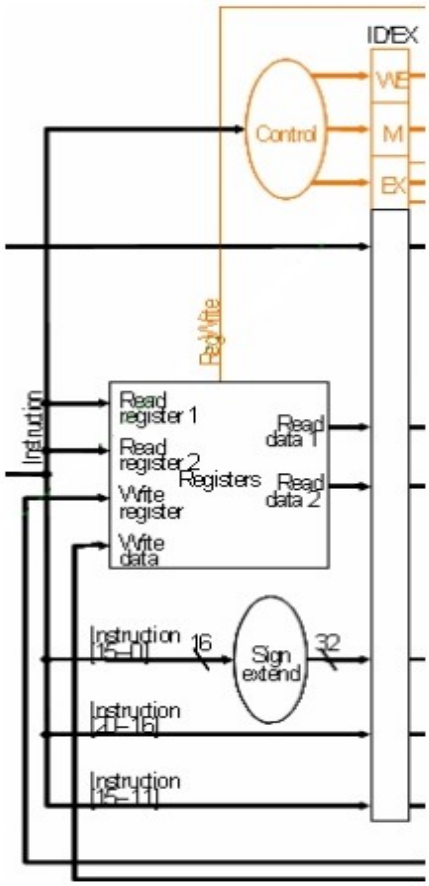


Figura 2.3 – Instruction decoding

Descripción de componentes:

Lógica adicional:

- Extensión de signo. Se debe tener en cuenta si el número es positivo o negativo para hacer la extensión. Y que el número esta representado en complemento a dos, por tanto el bit de más alto peso es indicador del signo a extender.

```
OFFSET_AUX    <= ZERO16b & OFFSET_A
                    when OFFSET_A(15) = '0'
                        else ONE16b & OFFSET_A;
```

Nota: se hace uso de constantes y alias.

Otras entidades:

- Unidad de Control: controla los componentes en cada etapa para ejecutar la instrucciones correctamente. Sus señales de control están sincronizadas, o sea que a medida que una instrucción se ejecuta etapa a etapa, dichas señales acompañan a la instrucción controlando a los componentes de la etapa. Es puramente combinacional.

La Unidad de Control toma como entrada el campo Op (Instrucción[31-26]) = (Op5, Op4, Op3, Op2, Op1, Op0)

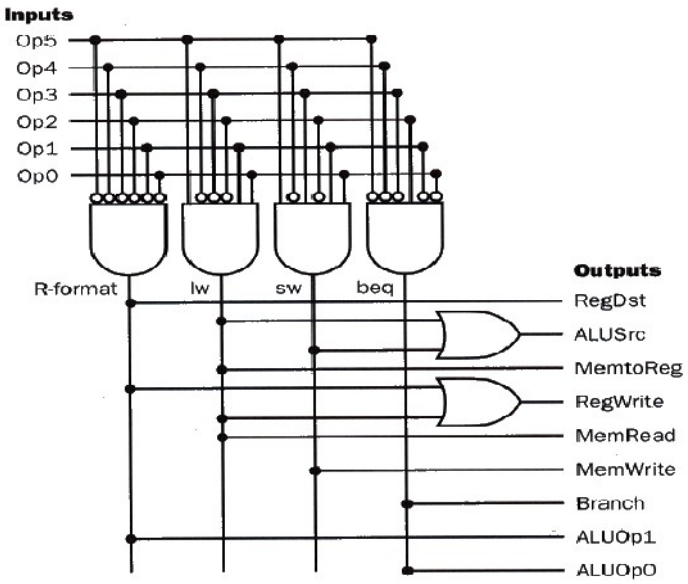


Figura 2.4 – Unidad de Control

El código VHDL es un mapeo directo y resulta trivial:

```
R_TYPE      <=   not OP(5) and not OP(4) and not OP(3) and
                 not OP(2) and not OP(1) and not OP(0);

LW          <=   OP(5) and not OP(4) and not OP(3) and
                 not OP(2) and   OP(1) and    OP(0);

SW          <=   OP(5) and not OP(4) and OP(3) and
                 not OP(2) and    OP(1) and    OP(0);

BEQ         <=   not OP(5) and not OP(4) and not OP(3) and
                   OP(2) and not OP(1) and not OP(0);

RegWrite    <=   R_TYPE or LW;
MemtoReg    <=   LW;
Brach       <=   BEQ;
MemRead     <=   LW;
MemWrite    <=   SW;
RegDst      <=   R_TYPE;
ALUSrc      <=   LW or SW;
ALUOp0      <=   BEQ;
ALUOp1      <=   R_TYPE;
```

La siguiente tabla muestra los valores de las señales de control dependiendo la instrucción que se ejecuta.

instrucción	RegDst	ALU Src	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALU OP1	ALU OP0
R-type	1	0	0	1	0	0	0	1	0
LW	0	1	1	1	1	0	0	0	0
SW	X	1	X	0	0	1	0	0	0
BEQ	X	0	X	0	0	0	1	0	1

Figura 2.5 – Salidas de la Unidad de Control

En pos de agrupar las señales de control de cada etapa se hace uso de los siguientes registros:

```
type WB_CTRL_REG is
record
        RegWrite        :       STD_LOGIC;
        MemtoReg        :       STD_LOGIC;
end record;

type MEM_CTRL_REG is
record
        Branch          :       STD_LOGIC;
        MemRead         :       STD_LOGIC;
        MemWrite        :       STD_LOGIC;
end record;

type ALU_OP_INPUT is
record
        Op0             :       STD_LOGIC;
        Op1             :       STD_LOGIC;
end record;

type EX_CTRL_REG is
record
        RegDst          :       STD_LOGIC;
        ALUOp           :       ALU_OP_INPUT;
        ALUSrc          :       STD_LOGIC;
end record;
```

- Banco de Registros: el funcionamiento fue explicado en la descripción general. Está definido de acuerdo al siguiente tipo:

```
type REGS_T is array (NUM_REG-1 downto 0) of STD_LOGIC_VECTOR(INST_SIZE-1 downto 0);
```

Posee un reset asincrónico implementado con el proceso:

```
REG_ASIG:
process(CLK,RESET,RW,WRITE_DATA,RD_ADDR)
begin
        if  RESET='1' then
                for i in 0 to NUM_REG-1 loop
                        REGISTROS(i) <= (others => '0');
                end loop;
        elsif  rising_edge(CLK) then
                if RW='1' then
                        REGISTROS(to_integer(unsigned(RD_ADDR)))
                                                        <=  WRITE_DATA;
                end if;
        end if;
end process  REG_ASIG;
```

Y su lógica responde al siguiente codigo:

```
RS <= (others=>'0') when RS_ADDR= "00000"
        else REGISTROS(to_integer(unsigned(RS_ADDR)));
RT <= (others=>'0') when RT_ADDR= "00000"
        else REGISTROS(to_integer(unsigned(RT_ADDR)));
```

2.2.3. - Execution (ejecución)

Descripción general: al producirse una flanco de subida los registros de sincronización de la etapa anterior actualizarán las señales de entrada la etapa actual. Algunas señales de control, WB y M, se postergarán hacia etapas siguientes, yendo directamente a los registros de sincronización.
El Offset extendido de la etapa anterior es multiplicado por cuatro (con el doble shift a izquierda) y sumado a la dirección de la próxima instrucción (calculada en IF). Recordemos que la multiplicación por cuatro se debe a la implementación de la Memoria de Instrucción. El resultado es una potencial dirección de salto, efecto que se decidirá en la etapa siguiente.

Por su parte la señal de control ALUOp y el campo Func de la instrucción serán las entradas del ALU Control, quien procesará estos datos y emitirá como salida la acción que debe tomar la ALU (ADD, SUB, AND, OR o SLT).

La ALU realizará entonces la operación comandada por ALU Control en base a los parámetros de entrada Rs y Rt o bien, en base a Rs y el Offset extendido. Esta decisión es tomada por un mux gobernado por ALUSrc, y depende de si lo que se quiere hacer es un ADD, SUB, AND, OR, SLT o BEQ o bien un LW o

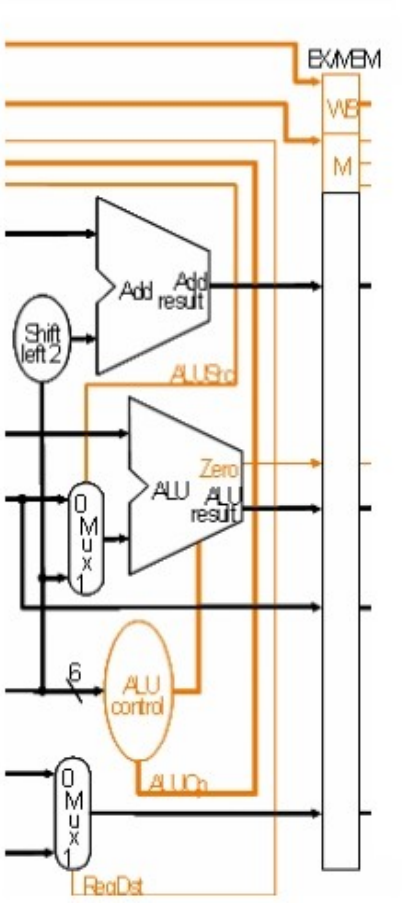


Figura 2.6 – Execution

SW, ya que estos últimos hacen la suma de "Base+Offset".

Por último existe un mux controlado por RegDst que filtra a Rt y Rd. Este se debe que dos registros destino dependiendo de la instrucción. El registro destino de la mayoría de las instrucciones es Rd, pero para el LW el registro destino es Rt.

Descripción de componentes:

Lógica adicional:

- Mux convencionales.
- Doble Shift a izquierda:

```
OFFSET_SHIFT2 <= OFFSET(29 downto 0) & "00";
```

Otras entidades:

- Sumador. Componente reusado de la etapa de IF.
- Control de la ALU. Responde al siguiente diagrama:

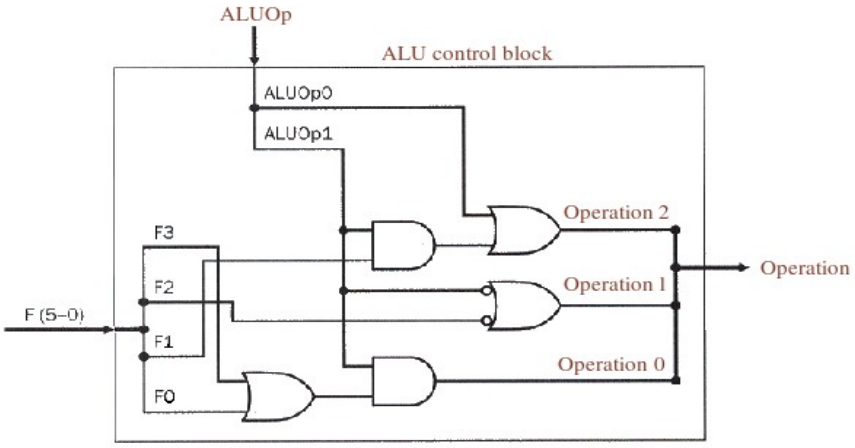


Figura 2.7 – Control de la ALU

Nuevamente el código VHDL resulta un mapeo directo:

```
ALU_IN.Op0 <= ALU_OP_IN.Op1 and ( FUNCT(0) or FUNCT(3) );
ALU_IN.Op1 <= (not ALU_OP_IN.Op1) or (not FUNCT(2));
ALU_IN.Op2 <= ALU_OP_IN.Op0 or ( ALU_OP_IN.Op1 and FUNCT(1) );
```

El resultado obtenido, ALU INPUT, es almacenado en un registro:

```
type ALU_INPUT is
record
        Op0     :       STD_LOGIC;
        Op1     :       STD_LOGIC;
        Op2     :       STD_LOGIC;
end record;
```

La siguiente tabla muestra los valores de las señales de control de la ALU dependiendo la instrucción que se ejecuta:

INSTR	ALUOp		FUNCT		ALU input
load word	0 0 (LW)		x x x x x x		0 1 0 (ADD)
store word	0 0 (SW)		x x x x x x		0 1 0 (ADD)
beq	0 1 (BEQ)		x x x x x x		1 1 0 (SUB)
add	1 0 (Tipo-R)	+	1 0 0 0 0 0	➡	0 1 0 (ADD)
sub	1 0 (Tipo-R)		1 0 0 0 1 0		1 1 0 (SUB)
and	1 0 (Tipo-R)		1 0 0 1 0 0		0 0 0 (AND)
or	1 0 (Tipo-R)		1 0 0 1 0 1		0 0 1 (OR)
slt	1 0 (Tipo-R)		1 0 1 0 1 0		1 1 1 (SLT)

Figura 2.8 – Entradas y Salidas del Control de la ALU

- Unidad Aritmético Lógica (ALU).
  - ALU de un bit. Este circuito es la célula básica de la ALU, y realiza un ADD, SUB, AND, OR o SLT dependiendo de tres entradas : OpCode2(bit-invert) ,OpCode1 y Opcode0, tal como se ve en la columna ALU Input de la figura anterior. Bit-Invert cambia la operación de ADD a SUB con un inversor, debido a que los números están representados en complemento a dos. Cin sólo interviene en la suma. Less y Set intervienen en el SLT, serán explicados luego. OpCode1 y OpCode0 son los selectores del mux que elige el resultado de salida.

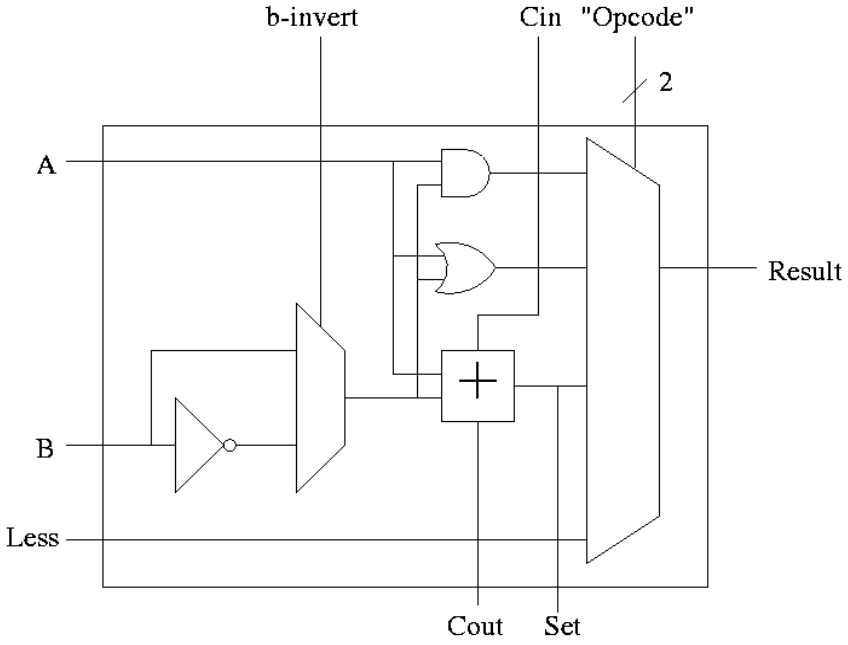


Figura 2.9 – ALU de 1 bit

Dicho diseño llevado a código VHDL:

```
MUX_BINV: process(BINVERT,Y) is
        begin
                if BINVERT='0' then
                        NEW_Y <= Y;
                else
                        NEW_Y <= not Y;
                end if;
        end process MUX_BINV;

R0 <= X and NEW_Y;
R1 <= X or NEW_Y;

FULLADDER_ALU: FULL_ADDER port map(
                        X       => X,
                        Y       => NEW_Y,
                        CIN     => CIN,
                        COUT    => COUT,
                        R       => R2
        );


R3 <= LESS;

MUX_RES_ALU: process(OP1,OP0,RES_AUX,R0,R1,R2,R3) is
        begin
                if (OP1 = '0' and OP0 = '0') then
                        RES_AUX <= R0;
                elsif (OP1 = '0' and OP0 = '1') then
                        RES_AUX <= R1;
                elsif (OP1 = '1' and OP0 = '0') then
                        RES_AUX <= R2;
                elsif (OP1 = '1' and OP0 = '1') then
                        RES_AUX <= R3;
                end if;
        end process MUX_OP;

RES <= RES_AUX;
SET <= R2;
```

- ALU: para implementar la ALU de 32 bits se hará uso de una concatenación de ALU de 1 bit. La primera llevará carry nulo. Todas serán controladas por los OpCode 2, 1 y 0.

  Para implementar el SLT todos los Less excepto el primero serán cero y todos los Set excepto el último también serán cero. El último Set estará conectado al primer Less, de esa manera si la resta dió negativa (A-B<0 => A<B), el último Set será 1 (por complemento a dos) y por tanto el primer Less será tabién uno. O sea que el resultado final será "00000000000000000000000000000001", que son todos los Less = 0 excepto el primero que es uno.

El siguiente diagrama muestra el diseño de la ALU:

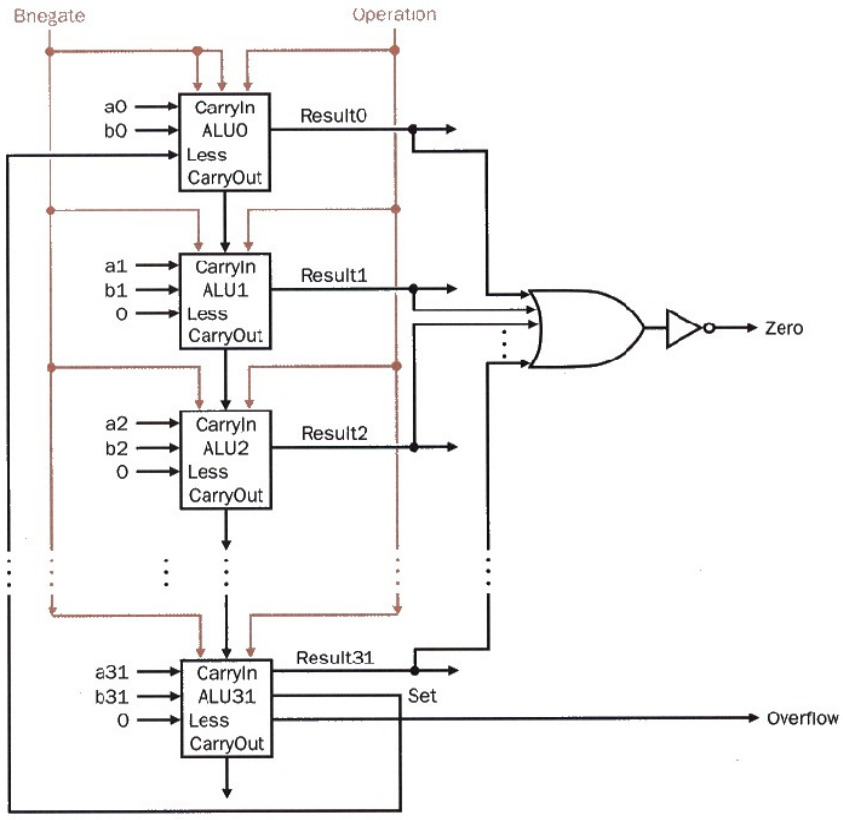


Figura 2.10 – ALU

Para el cálculo de las Flags se utiliza el siguiente registro:

```
type ALU_FLAGS is
record
        Carry           :       STD_LOGIC;
        Overflow        :       STD_LOGIC;
        Zero            :       STD_LOGIC;
        Negative        :       STD_LOGIC;
end record;
```

Recordemos que para calcular las flags:

Carry = Cn-1
Overflow = Cn-1 xor Cn-2
Zero = not (Rn-1 or .... or R0)
Negative = SETn-1 = Rn-1

El siguiente código VHDL muestra la implementación de la ALU:

```
BEGIN_ALU1B: ALU_1BIT port map (
                X       => X(0),
                Y       => Y(0),
                LESS    => LESS_AUX,
                BINVERT => ALU_IN.Op2,
                CIN     => ALU_IN.Op2,
                OP1     => ALU_IN.Op1,
                OP0     => ALU_IN.Op0,

                RES     => R_AUX(0),
                COUT    => COUT_AUX(0)
        );

GEN_ALU: for i in 1 to N-2 generate
                NEXT_ALU1B: ALU_1BIT port map (
                        X       => X(i),
                        Y       => Y(i),
                        LESS    => '0',
                        BINVERT => ALU_IN.Op2,
                        CIN     => COUT_AUX(i-1),
                        OP1     => ALU_IN.Op1,
                        OP0     => ALU_IN.Op0,
                        RES     => R_AUX(i),
                        COUT    => COUT_AUX(i)
                );
        end generate;

LAST_ALU1B: ALU_1BIT port map (
                X       => X(N-1),
                Y       => Y(N-1),
                LESS    => '0',
                BINVERT => ALU_IN.Op2,
                CIN     => COUT_AUX(N-2),
                OP1     => ALU_IN.Op1,
                OP0     => ALU_IN.Op0,
                RES     => R_AUX(N-1),
                COUT    => COUT_AUX(N-1),
                SET     => LESS_AUX
        );


FLAGS.Carry <= COUT_AUX(N-1);
FLAGS.Overflow <= COUT_AUX(N-1) xor COUT_AUX(N-2) ;
FLAGS.Negative <= '1' when R_AUX(N-1)='1' else '0';
FLAGS.Zero <= '1'
        when R_AUX=ZERO32b else '0';
R <= R_AUX;
```

2.2.4. - Memory access (acceso a memoria)

Descripción general: al producirse una flanco de subida los registros de sincronización de la etapa anterior actualizarán las señales de entrada la etapa actual. Las señales de control WB se postergarán hacia la siguiente etapa, yendo directamente a los registros de sincronización.

Dentro de las señales de control de la etapa actual se tiene la señal Branch. Para realizar la instrucción BEQ (salto por igual) se deben cuplir dos condiciones, que se desee hacer un salto (Branch=1) y que la resta entre los dos registros dados en la instrucción sea cero, o sea que los registros sean iguales. Es por eso que se realiza un AND entre Branch y el flag Zero de la ALU. El resultado es un control asincrónico hacia la etapa IF llamado PCSrc. Por otro lado en la etapa anterios se calculo la potencial dirección de salto (Dirección + Offset extendido por cuatro), dicha dirección será la entrada al mux controlado por PCSrc.

Por su parte la Memoria de Datos es controlada por las señales MemRead y MemWrite. Si la instrucción que se esta ejecutando es un LW sólo la señal MemRead estará activada, por tanto en base a la dirección calculada por la ALU, en el campo Address se emitirá como salida el contenido de dicha dirección en Read data. Si la instrucción que se esta ejecutando es un SW sólo la señal MemWrite estará activada, esto producirá que en la dirección del campo Address se grabe el contenido de Write data. Este último es el campo Rt, y fue provisto por la etapa anterior.

Por último, la dirección calculada por la ALU, el campo Address se redirige a los registros de sincronización para la siguiente etapa, así como también la dirección del mux controlado por RegDst de la etapa anterior.

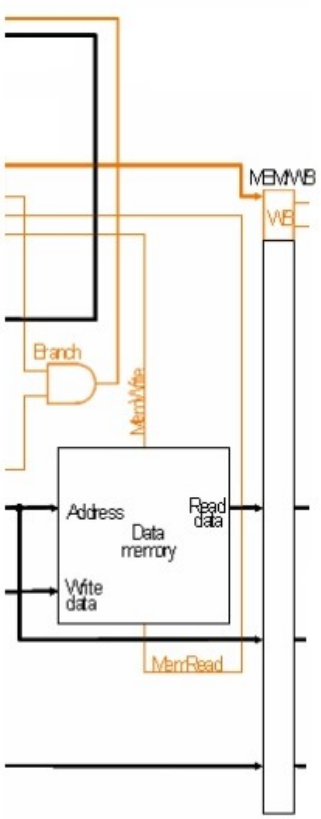


Figura 2.11 – Memory access

Descripción de componentes:

Lógica adicional:

- AND convencional.
- Memoria de Datos: la implementación VHDL resulta bastante sencilla. Es una entidad genérica, donde M es la cantidad de datos de la memoria y N es el tamaño de un dato en bits. Se define el siguiente tipo:

```
type MEM_T is array (NUM_REG-1 downto 0) of STD_LOGIC_VECTOR (INST_SIZE-1 downto 0);
signal MEM : MEM_T;
```

Posee un reset asincrónico.

```
MEM_PROC:
process(RESET,MemWrite,MemRead,WRITE_DATA,MEM,ADDR)
        begin
                if (RESET = '1') then -- Reset Asincrónico
                        for i in 0 to M-1 loop
                                MEM(i) <= (others => '1');
                        end loop;
                -- Ejecuto las ordenes de la unidad de control:
                elsif MemWrite='1' then  --O bien  escribo en la memoria
                        MEM(to_integer(unsigned( ADDR(9 downto 0) ))) <= WRITE_DATA;
                elsif MemRead='1' then -- O bien leo de ella
                        READ_DATA <= MEM(to_integer(unsigned( ADDR(9 downto 0) )));
                end if;

        end process MEM_PROC;
```

2.2.4. - Write back (post escritura)

Descripción general: al producirse una flanco de subida los registros de sincronización de la etapa anterior actualizarán las señales de entrada la etapa actual. El objetivo de esta etapa es actualizar el Banco de Resgistros de la etapa ID. La señal de control RegWrite estará activa en caso de que se deba hacer la escritura. Dicha escritura sólo es válida para las instrucciones de Tipo-R, por ejemplo un ADD que necesita que se guarde el resultado en el registro Rd. O bien para la instrucción LW, ya que esta trae un dato de la Memoria de Datos y lo guarda en un registro.

Write register es la dirección del registro a guardar y puede ser Rd si es una Tipo-R o Rt si es un LW (recordemos que esto se decidió en la etapa de EX con el mux controlado por RegDst). Y finalmente MemToReg decidirá que es lo que se guardará en Write data. Si esta activado lo que se quiere hacer es un LW y los datos serán los extraidos de la Memoria de Datos en la etapa anterior. Si no lo está, es una Tipo-R y el dato será el resultado procesado por la ALU que se viene trayendo desde la etapa de EX.

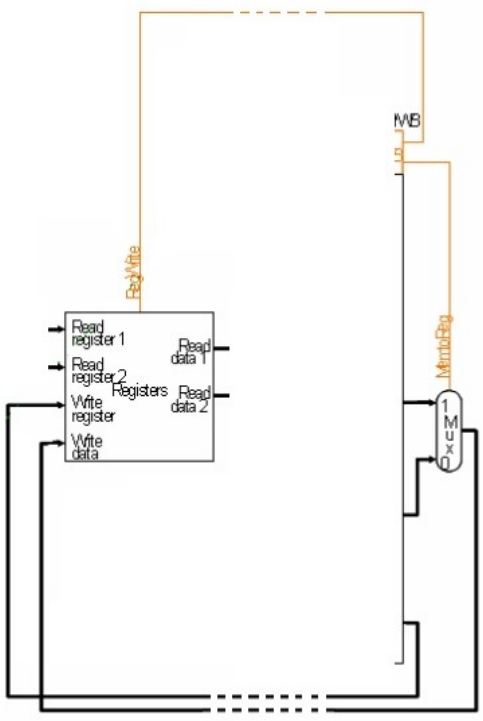


Figura 2.12 – Write back

Descripción de componentes:

Lógica adicional:

- Mux convencional

Otros componentes:

- Banco de Registros. Explicado en sección 2.2.2

2.2.5. - Registros de sincronización

Estos registros se encuentran en todas las etapas excepto WB. Y se encargan de retrasar el procesamiento de cada etapa al de la etapa más lenta, de manera de sincronizar todo el procesador (ver introducción).

Se implementan de manera muy sencilla en VHDL. Como ilustración se muestra la implementación de los registros de la etapa IF:

```
IF_ID_REGS:  process(CLK,RESET)
             begin
                  if RESET = '1' then
                       NEW_PC_ADDR_OUT
                            <= ZERO32b;
                       INST_REG_OUT
                            <= "ZERO32b;
                  elsif rising_edge(CLK) then
                       NEW_PC_ADDR_OUT    <= NEW_PC_ADDR_IN;
                       INST_REG_OUT<= INST_REG_IN;
                  end if;
             end process;
```

2.2.6. - Agregado de la instrucción LUI

En las descripciones que hemos hecho hasta ahora no se ha considerado la implementación de la instrucción LUI. Para ello deberemos modificar parcialmente algunas entidades.

LUI (Load Upper Immediate) es una instrucción tipo I, que tiene como propósito la carga alta de un registro (R[31..16]). La idea es la siguiente, el procesamiento de LUI es muy similar al LW, ya que ambos cargan un valor a un registro del Banco de Registros.

```
LW      Rt, Offset(Base)
LUI     Rt, immediate
```

La diferencia radica en la ALU. Mientras que en LW se suma la Base (Rs) con el Offset, en LUI se desprecia la Base y se hace un shift a derecha de 16 bits con Immediate (Offset), ya que el objetivo es cargar la parte alta. Dicho resultado, o sea la salida de la ALU, se guarda en Rt en ambos casos.

Para lograr esto primero se deberá hacer una modificación a la ALU. Se le agregará una señal de control más, que indique cuando se debe hacer el shift y que dicho resultado debe asignarse a la salida en vez del calculado habitualmente (R_AUX).

```
ALU_RES: process(ALU_IN.Op3,R_AUX)
        begin
                if  ALU_IN.Op3='1' then
                        R <= Y( ((N/2)-1) downto 0) && ZERO16b;
                else
                        R <= R_AUX;
                end if;
        end process;
```

Este cambio se propaga al Control de la ALU, debido a la nueva señal de control.

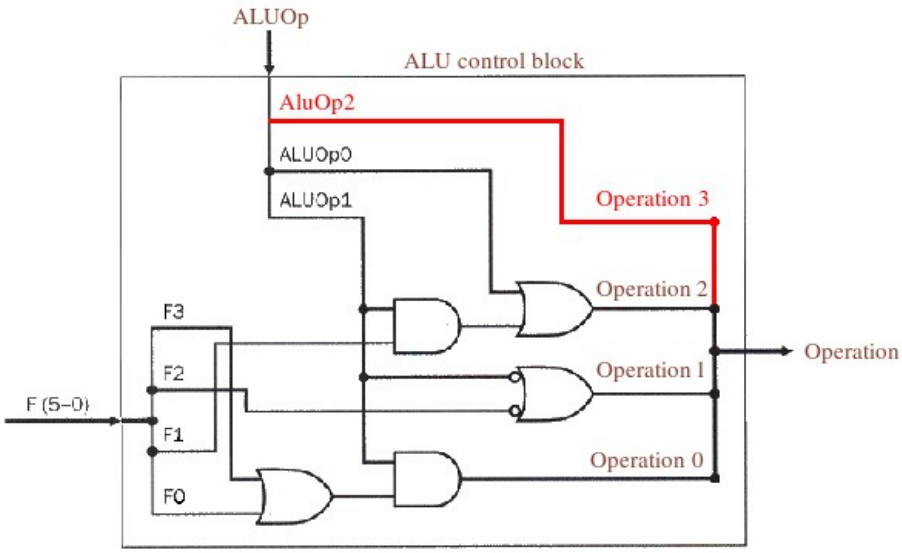


Figura 2.13 Modificación en Control de la ALU para LUI

De aquí se desprenden más propagaciones de cambios. En los registros ALU_INPUT y ALU_OP_INPUT, y por supuesto en la Unidad de Control.

Cambios en los registros:

```
type ALU_OP_INPUT is
  record
     Op0   : STD_LOGIC;
     Op1   : STD_LOGIC;
     Op2   : STD_LOGIC;
  end record;
```

```
type ALU_INPUT is
  record
     Op0   : STD_LOGIC;
     Op1   : STD_LOGIC;
     Op2   : STD_LOGIC;
     Op3   : STD_LOGIC;
  end record;
```

Cambios en la Unidad de Control:

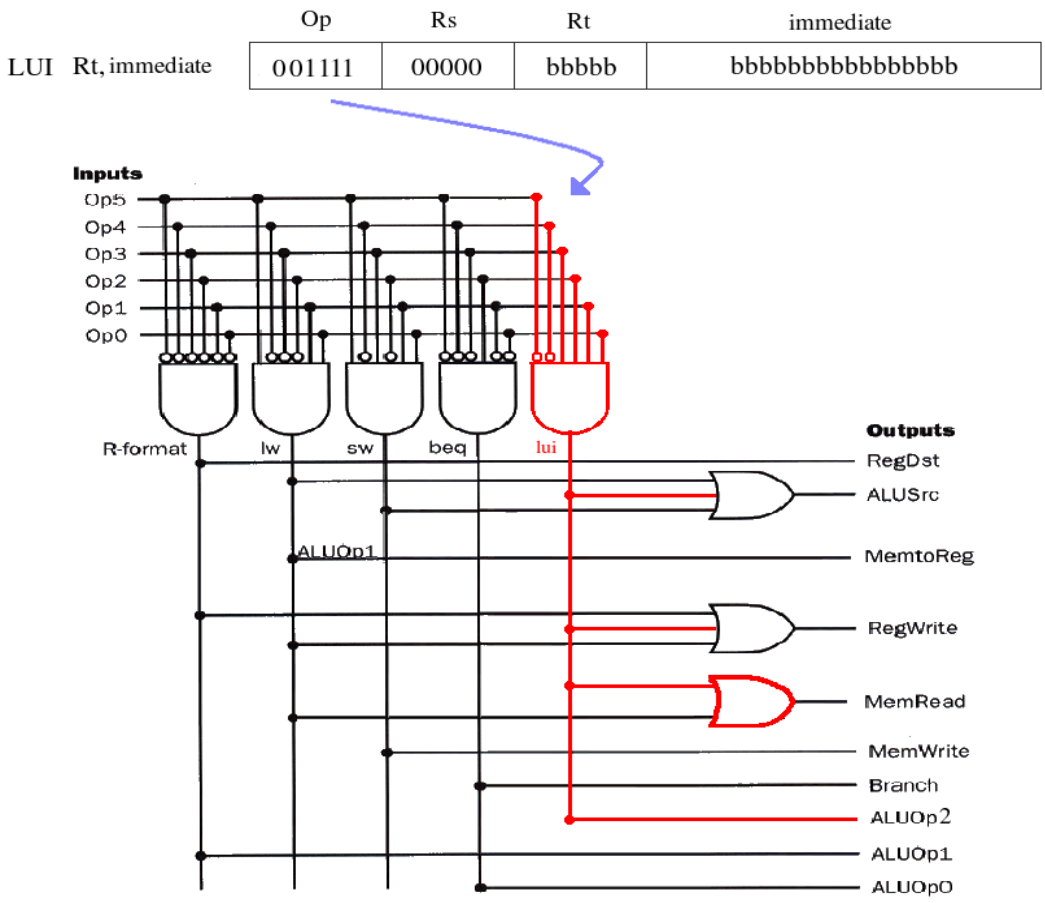


Figura 2.14 - Modificación en la Unidad de Control para LUI

Como vemos no sólo se agrega ALUOp2, sino que también se conservan los valores de las señales igual que en LW, excepto en MemToReg, la cual debe ser cero para que en la etapa de WB pase el resultado de la ALU y no la salida de la Memoria de Datos como contenido a escribirse en el registro Rt.

2.3. - CALIDAD DEL CÓDIGO

Convención de nombres:

- Uso de letras minúsculas para todos los nombres de las señales, nombres de variables (no se usaron en este proyecto) y puertos. Y mayúsculas para los nombres de las constantes y los tipos.
- Uso de nombres significativos.
- Uso de un nombre consistente para el clock y el reset: “clk” y “reset”.
- Uso consistente del orden de los buses multibit: (X downto 0)
- Uso de nombres similares en los puertos y las señales que se les conectan.
- Los nuevos tipos (tipos creados por el usuario) finalizan en “_T”

Encabezados y comentarios:

```
--
-- Descripción general. Ej: Memoria de datos del procesador MIPS Segmentado
--
-- Licencia: Copyright 2008 Emmanuel Luján
--
--      This program is free software; you can redistribute it and/or
--      modify it under the terms of the GNU General Public License as
--      published by the Free Software Foundation; either version 2 of
--      the License, or (at your option) any later version. This program
--      is distributed in the hope that it will be useful, but WITHOUT
--      ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
--      or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public
--      License for more details. You should have received a copy of the
--      GNU General Public License along with this program; if not, write
--      to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor,
--      Boston, MA 02110-1301 USA.
--
-- Autor:       Emmanuel Luján
-- Email:       info@emmanuellujan.com.ar
-- Versión:     1.0
--
```

Estilo de Escritura:

- Uso de un tab (en vez de dos espacios) de identación para mejorar la legibilidad.
- Líneas con menos de 72 caracteres excepto en los comentarios de las entidades, que de otra forma alargan mucho el código.

Puertos y genéricos:

- Uso de comentarios para describir puertos.
- Uso de mapeos explícitos para puertos y genéricos, en vez de mapeo posicional.

Uso de Arrays y Loops

Desaciertos en simulación y síntesis:

- Especificación de listas de sensibilidad completas.
- Enunciados de retardo removidos. Sólo se usan en el Test Bench.

Portabilidad del código:

- Uso de las librerías "STD" e "IEEE".
- Uso conservativo en la cantidad de subtipos. Sólo se usan dos tipos creado por el usuario en todo el proyecto.
- Uso solamente de STD_LOGIC y STD_LOGIC_VECTOR para señales.
- Uso de constantes en vez de valores numéricos fijos (hard-coded). Las constantes se encuentran en un archivo separado que es incluido por las las entidades que las usan.

Reloj:

- Activado sólo por flanco de subida.
- No se usan relojes internos, el reloj es único.

Uso de entidades genéricas:

- En el registro usado para implementar el PC, en IF.
- En el sumador usado en IF y EX.
- En la ALU de EX.
- En la Memoria de Datos de MEM

Uso de generates iterativos:

- Para la generación de la ALU en base a las ALU's de 1 bit.
- Para la generación del sumador en base a los sumadores completos.

Uso de Alias:

- Constantes
- Alias propiamente dichos, en ID, para clarificar las distintas partes de la instrucción que esta siendo decodificada.

Uso de Registros: Para aunar las señales de control de cada etapa

2.4. - HERRAMIENTAS USADAS

2.3.1. - GHDL

GHDL es un simulador de VHDL, basado en GCC.
Implementa al lenguaje VHDL según el estándar IEEE 1076-1987 o IEEE 1076-1993.
Compila archivos VHDL y crea un binario que simula (o ejecuta) nuestro diseño.
No hace síntesis: no puede traducir nuestro diseño a una netlist.
GHDL es Software Libre.

2.3.2. - GTKWAVE

Cuando se desea debuguear un diseño es muy útil mirar sos ondas digitales. GHDL puede generar un archivo de forma de onda, que soporta dos formatos.

El primer formato VCD (Value Change Dump), es un formato abierto definido para Verilog. La especificación del formato esta definida por Verilog LRM. VDC es un formato ACII, así que los archivos de este tipo crecen rápidamente. La mayoría (o todos) de los visores de ondas digitales soportan VCD.

El segundo es el formato de formas de inda de GHDL. Es un formato binario, y sus especificaciones no están aún completamente terminadas.

Versiones recientes de GTK Wave pueden leer este formato.

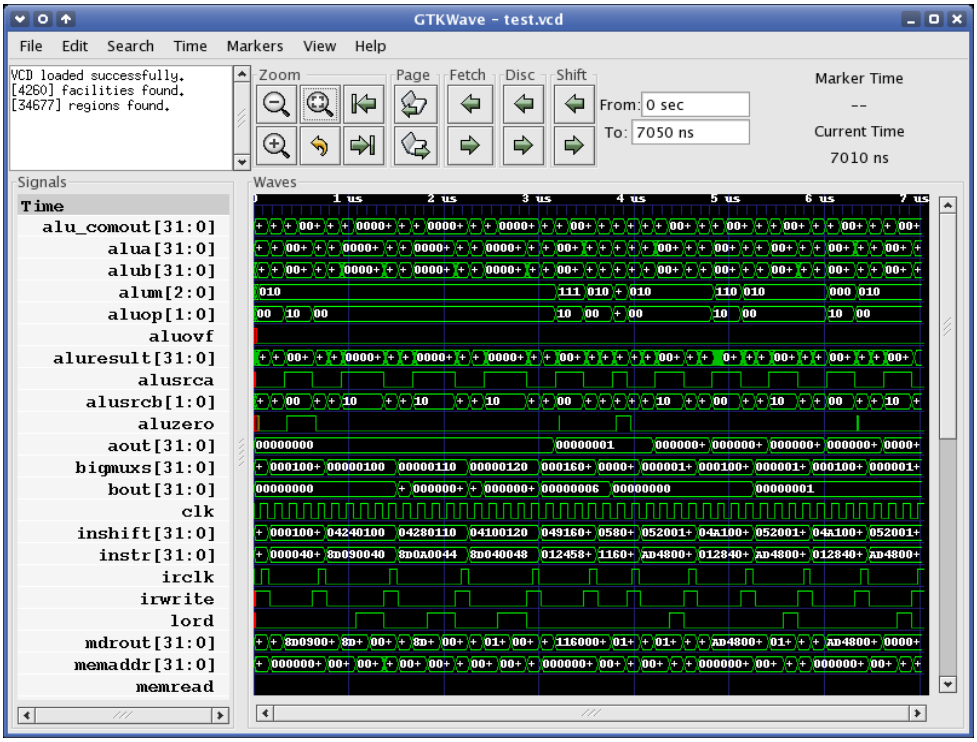


Figura 2.15 - GTKWave

TRADUCTOR

3.1.- Funcionalidad

Como se mencionó en los objetivos, se necesita un traductor básico para poder probar ejemplos en el procesador con mayor dinamicidad. Este traductor es un objetivo secundario al proyecto, por lo que su implementación se realiza sin mayor profundidad.

Instrucción Tipo-R (add, sub, and, or y slt.)

	Op.	Rs	Rt	Rd	Shamnt	funct
	31-26	25-21	20-16	15-11	10-6	5-0
ADD Rd, Rs, Rt	000000	bbbbb	bbbbb	bbbbb	00000	100000
SUB Rd, Rs, Rt	000000	bbbbb	bbbbb	bbbbb	00000	100010
AND Rd, Rs, Rt	000000	bbbbb	bbbbb	bbbbb	00000	100100
OR Rd, Rs, Rt	000000	bbbbb	bbbbb	bbbbb	00000	100101
SLT Rd, Rs, Rt	000000	bbbbb	bbbbb	bbbbb	00000	101010

Figura 3.1 – Formato binario de instrucciones Tipo-R

Instrucción Tipo-I (lw, sw, lui, beq .)

	Op.	Rs	Rt	offset
	31-26	25-21	20-16	15-0
LW Rt, offset (Rs)	100011	bbbbb	bbbbb	bbbbbbbbbbbbbbbb
SW Rt, offset (Rs)	101011	bbbbb	bbbbb	bbbbbbbbbbbbbbbb
LUI Rt, immediate	001111	00000	bbbbb	bbbbbbbbbbbbbbbb
BEQ Rs, Rt, offset	000100	bbbbb	bbbbb	bbbbbbbbbbbbbbbb

Figura 3.2 – Formato binario de instrucciones Tipo-I

En las figuras anteriores podemos ver el formato de las instrucciones implementadas y su representación en binario.

Los registros se representan con un signo "$", y el número que lo identifica.

Por ejemplo: ADD $1,$2,$3 .

Las "b" representan los bits que codifican a Rs, Rt, Rd y Offset. Dicha codificación representa el número que codifica al registro. Tomando como base el ejemplo anterior su codificación sería:

000000 00010 00011 00001 00000 100000

*Nota: la codificación real se escribe sin espacios.

El traductor:

Permite comentarios del tipo:

ADD $10,$10,$1 // Incremento de la variable 10 en 1

No es case sensitive

3.2.- Tratamiento de errores

- Detección de caracteres inválidos dentro del operador
- Eliminación de caracteres inválidos dentro del operador
- Detección de operador válido
- Detección de caracteres inválidos dentro de un registro
- Eliminación de caracteres inválidos dentro de un registro
- Detección de caracteres inválidos dentro del Offset
- Eliminación de caracteres inválidos dentro del Offset

ALGORITMO DE PRUEBA

4.1.- Restoring

Se implementará el algoritmo de división por el método de "Restoring".
La división entre el número X e Y da como cociente Q y resto R.
Condiciones:

X e Y deben estar representados en binario.
X e Y deben ser números naturales. Para lograrlo se puede usar normalización.
X < Y . Nuevamente se puede usar normalización.

Dicho algoritmo responde al siguiente pseudocódigo:

```
R0 := X
for i=1 to M loop
        R(i) := 2 * Ri-i  – Y
        if Ri >= 0 then
                Qi = 1
        else
                Qi = 0
                Ri := 2 * Ri-1
        end if
end for
```

Para pasar dicho programa al assembler acotado por las instrucciones disponibles se se hace uso de las siguientes técnicas:

- El shift a derecha lo hacemos sumando dos veces el mismo número.

```
shr $1= add $1,$1,$1
```

- El mayor igual llo implementamos con SLT:

```
slt $8,$4,$0      // if (r(i)<0 ) then aux := 1 else aux:=0
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
beq $8,$1,7       // if (aux = 1) then goto PC+7 (dónde se quiera saltar)
                  // else goto PC (continuar con la ejecución)
```

Los nops son para evitar problemas de dependencias.

- Para setear el Qi usamos una máscara y luego hago un shift a derecha:

```
ldi $5,1          // orMask = 1 = 00000000000000000000000000000001
...
or $12,$12,$5     // Q or orMask => q(i) = 1
...
add $5,$5,$5      // orMask := shifRight(orMask)     EJ:  con 4 bits: 0001 => 0010
```

- Debido a que el algoritmo es iterativo necesitamos alguna instrucción que permita volver hacia atrás, para ello usaremos BEQ con un Offset negativo. El compilador entenderá que debe pasar el mismo a complemento a dos. De esa manera conseguirá que se reste en vez de sumar en la etapa EX, por que lo se obtendrá un salto hacia atrás.

```
Beq $8,$1,-41     // Última instrucción del algoritmo, retorna al principio del loop.
```

Luego el código resultante es:

```
add $4,$10,$0     // r(0) := X
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
add $3,$4,$0      // Rant_aux := r(i-1)
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
add $4,$3,$3      // r(i) := r(i-1) + r(i-1) = 2 * r(i-1) = shifRight(r(i-1))
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $4,$4,$11     // r(i) := r(i) - Y = shifRight(r(i-1)) -Y
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
slt $8,$4,$0      // if (r(i)<0 ) then aux:=1 else aux:=0
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
beq $8,$1,8       // if (aux = 1) then goto NEW_PC+7(goto else)
                  // else goto NEW_PC(continue with the execution)
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
or $12,$12,$5     // Q or orMask => q(i) = 1
beq $0,$0,4       // goto NEW_PC+3 (go to the end of the loop)
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
```

```
sub $0,$0,$0      // nop
add $4,$3,$3      // r(i) := shifRight(r(i-1)); q(i)=0 because it was initializated all in zero.
add $6,$6,$1      // i++
add $5,$5,$5      // orMask := shifRight(orMask)  e.g. with 4 bits: 0001 => 0010
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
slt $8,$6,$7      // if (i<N) then aux:=1 else aux:=0
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
sub $0,$0,$0      // nop
beq $8,$1,-41     // if (aux = 1) then goto begining_of_the_loop
                  // else goto NEW_PC (end)
```

BIBLIOGRAFÍA

Apuntes teóricos:

- "Arquitectura de Computadoras I" ,
  Universidad Nacional del Centro de la Provincia de Buenos Aires.
  http://www.exa.unicen.edu.ar/catedras/arqui1/

- "Arquitectura de Computadoras y Técnicas Digitales" ,
  Universidad Nacional del Centro de la Provincia de Buenos Aires.
  http://www.exa.unicen.edu.ar/catedras/arqui2/

- "Lenguaje de descripción de hardware VHDL " ,
  Ing. Martín Vazquez.
  Universidad Nacional del Centro de la Provincia de Buenos Aires.
  http://www.exa.unicen.edu.ar/catedras/arqui1/

- "VHDL Avanzado" ,
  Dr. Elias Todorivich.
  Universidad Autónoma de Madrid, España.

- New York University, Computer Science.
  http://www.cs.nyu.edu/courses/fall99/V22.0436-001/class-notes.html

Herramientas:

- FPGA Libre
  http://fpgalibre.sourceforge.net/

- GHDL
  http://ghdl.free.fr/

- GTKWave
  http://home.nc.rr.com/gtkwave/