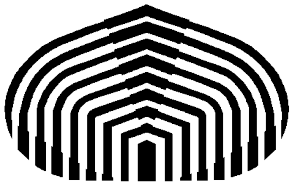

Universidad Simón Bolívar
EC3731-Arquitectura del Computador II
Prof. Ángel Terrones (aterrones@usb.ve / ELE-328)
Prof. Marta Pérez (mperezo@usb.ve / ELE-216)

Diseño y Síntesis del Procesador:

ELBETH

Estudiantes:
Emanuel Sanchez 12-11509
Ninisbeth Segovia 11-10960

Sartenejas, 02 de abril de 2016

Contenido

Pág.

1. Introducción.

1.1 Características del Procesador:

El procesador fue implementado de forma modular, y para ello se utilizó el *Hardware Description Language* (HDL), o Lenguaje de Descripción de Hardware, Verilog. Las características principales del procesador son las siguientes:

- Segmentado con un Pipeline de 3 etapas. (Ver sección 2)
- Unidad para detección de riesgos, en esta se realiza el reenvío de datos (Fordwarding) en caso de dependencia de datos. (Ver sección 3.2.4)
- Soporte para el ISA (Instruction Set Architecture) RISCV 32 I. (Ver sección 1.3)
- Soporte para el modo User y Machine. (Ver sección 1.3)
- 32 Registros de propósito general más uno para el *Program Counter*.
- Sistema *Little Endian*.
- Soporte para 256 Kb de memoria RAM. (Ver sección 3.1.1)
- Sistema para la detección de excepciones.

1.2 Plataforma y herramientas.

Como ya se había señalado anteriormente, el Procesador ELBETH fue implementado en Verilog HDL.

En un principio, se utilizó la potente herramienta Xilinx ISE (Integrated Synthesis Environment o Entorno Integrado de Síntesis), que es un software para diseño de hardware entre otras cosas. Luego, debido a problemas al momento de cargar archivos hexadecimales para realizar pruebas sobre la memoria en el ISE de Xilinx se decidió utilizar Icarus Verilog [1] para la compilación de los archivos en verilog, y poder continuar así la implementación del procesador. Cabe destacar que la gran mayoría del trabajo se realizó haciendo uso de esta segunda herramienta.

A medida que se escribía el código de los módulos, se utilizó la herramienta GTKWave [2], la cual consiste en una interfaz gráfica que muestra las ondas o señales en función del tiempo, de cada una de las entradas y salidas de dichos módulos.

Luego de realizar tests bench en verilog a los módulos hechos se decidió hacer uso del lenguaje de programación interpretado Python junto con la librería MyHDL [9] para realizar pruebas mucho más eficientes y fáciles de implementar, esto también se hizo con el propósito de posteriormente probar los tests diseñados en la universidad de Berkeley [3]. La librería mencionada anteriormente permite realizar la co-simulación del procesador descrito en Verilog con tests hechos en Python, lo cual abre la puerta a muchas otras herramientas que fueron luego implentadas y que son descritas más

adelante (Ver sección 4).

1.3 Instrucciones soportadas por el procesador:

Las instrucciones soportadas por ELBETH estan codificadas en 4 formatos y estas operan en dos niveles:

- ***User Mode:*** el Base Integer Instruction Set, específicamente el ISA RISCV32I, el cual incluye un total de 47 instrucciones, de las cuales fueron implementadas 45. *FENCE y FENCE.I* no fueron implementadas en el diseño debido a que tiene que ver con el manejo de hilos lo cual no está incluido en esta versión del procesador.
- ***Machine Mode:*** incluye 14 instrucciones de las cuales fueron implementadas sólo 9, debido a que las demás tiene relación con la transferencia del control desde el *Machine* a otros niveles de usuario, el manejo de interrupciones, y el manejo de hilos, características que no están dentro de esta versión de procesador.

Instrucción	Tipo	Operación
ADDI	**I**	rd= rs1+imm (con signo)
SLTI	**I**	si rs1<imm, rd=1 sino rd=0
SLTIU	**I**	si rs1<imm, rd=1 sino rd=0
ANDI	**I**	rd= rs1 && imm (12)
ORI	**I**	rd= rs1 || imm (12) (1ORI1=1)
XORI	**I**	como ORI pero 1 XORI 1=0
SLLI	**I***	shamt bits a la izq. 0 la der (*2^shamt)
SRLI	**I***	shamt bits a la der. 0 la izq (/2^shamt)
SRAI	**I***	Como SRLI pero conserva bit de signo
LUI	**U**	rd=[imm(20)12 ceros]
AUIPC	**U**	rd=[imm(20)12 ceros]+PC
ADD	**R**	rd=rs1+rs2
SUB	**R**	rd=rs1-rs2
SLT	**R**	si rs1<rs2 rd=1 sino rd=0
SLTU	**R**	si rs1<rs2 rd=1 sino rd=0
AND	**R**	rd= rs1 && rs2
OR	**R**	rd= rs1 || rs2 (1 OR 1= 1)
XORI	**R**	como OR pero 1 XOR 1= 0
SLL	**R**	Cant de 5 bits inf de r2 a la izq (*2^n)

SRL	**R**	Cant de 5 bits inf de r2 a la der (/2^n)
SRA	**R**	Como SRL pero conserva bit de sign (extendiendolo a la derecha)
JAL	**UJ**	rd=PC+4 (rd=x1).PCnew=PC+4+imm
JALR	**I**	PCnew=PC+4+imm(12)+rs1
BEQ	**SB**	si rs1=rs2 PCnew=PC+4+brch
BNE	**SB**	si rs1!=rs2 PCnew=PC+4+brch
BLT	**SB**	si rs1<rs2 PCnew=PC+4+brch
BLTU	**SB**	si rs1<rs2 PCnew=PC+4+brch
BGE	**SB**	si rs1>rs2 PCnew=PC+4+brch
BGEU	**SB**	si rs1>rs2 PCnew=PC+4+brch
LB	**I**	rd=24veces el bit 7 de los 8bits inf de MEM(rs1+imm con sig)][8 bits]
LH	**I**	rd=16veces el bit 15 de los 16bits inf de MEM(rs1+imm con sig)][16 bits]
LW	**I**	rd=32 bits de MEM(rs1+imm con sig)]
LBU	**I**	rd=24veces el bit 7 de los 8bits inf de MEM(rs1+imm sin sig)][8 bits]
LHU	**I**	rd=24veces el bit 7 de los 8bits inf de MEM(rs1+imm sin sig)][8 bits]
SB	**S**	MEM[rs2+imm(con sig)]=24veces el bit 7 de los 8bits inf de rs1[8 bits]
SH	**S**	MEM[rs2+imm(con sig)]=16veces el bit 15 de los 8bits inf de rs1[16 bits]
SW	**S**	MEM[rs2+imm(con sig)]= 32bits de rs1
SCALL	**I**	realiza un llamado al sistema operativo desde el modo *user*
SBREAK	**I**	retorna el control al sistema
RDCYCLE	**I**	retorna la cantidad de ciclos ejecutados
RDTIME	**I**	retorna el tiempo de ejecución desde un cierto punto (32 bits menos significativos).
RDTIMEH	**I**	igual a la anterior, pero retorno los 32 bits más significativos
RDINSTRET	**I**	retorna el número de instrucciones ejecutadas con éxito
CSRRW	**I**	realiza un swap entre los valores del registro rs1 y un registro del CSR

CSRRS	**I**	si rs1 =! 1, escribe rd con un valor del registro del CSR y los bit de rs1 que estén en alto son colocados en alto el registro del CSR, de lo contrario solo lee
CSRRC	**I**	si rs1 =! 1, escribe rd con un valor del registro del CSR y los bit de rs1 que estén en alto son colocados en cero en el registro del CSR, de lo contrario solo lee
CSRWI	**I**	igual a CSRRW con la diferencia de que escribe un valor inmediato si este es diferente de cero
CSRRSI	**I**	igual a CSRRS con la diferencia de que escribe un valor inmediato si este es diferente de cero
CSRRCI	**I**	igual a CSRC con la diferencia de que escribe un valor inmediato si este es diferente de cero
ECALL	**I**	realiza un llamado al sistema operativo desde e modo *machine*
EBREAK	**I**	retorna el control al sistema
ERET	**I**	instrucción para regresar de una excepción

Cuadro 1. Instrucciones soportadas por el Procesador ELBETH

2. Descripción del diseño.

En la Figura 1, puede observarse un diagrama de bloques para describir la comunicación entre el Procesador y la Memoria RAM, por medio del *Memory Bridge* el cual es el puente o modulo intermedio entre la memoria y el procesador. Este módulo llamado *elbeth_memory_bridge*, toma los 32 bits del PC, y los lleva a 16 bits de dirección por lo cual se dice que el procesador soporta una memoria RAM de 256 kBytes, lo que es equivalente a tener 2^16=65536 posibles direcciones.

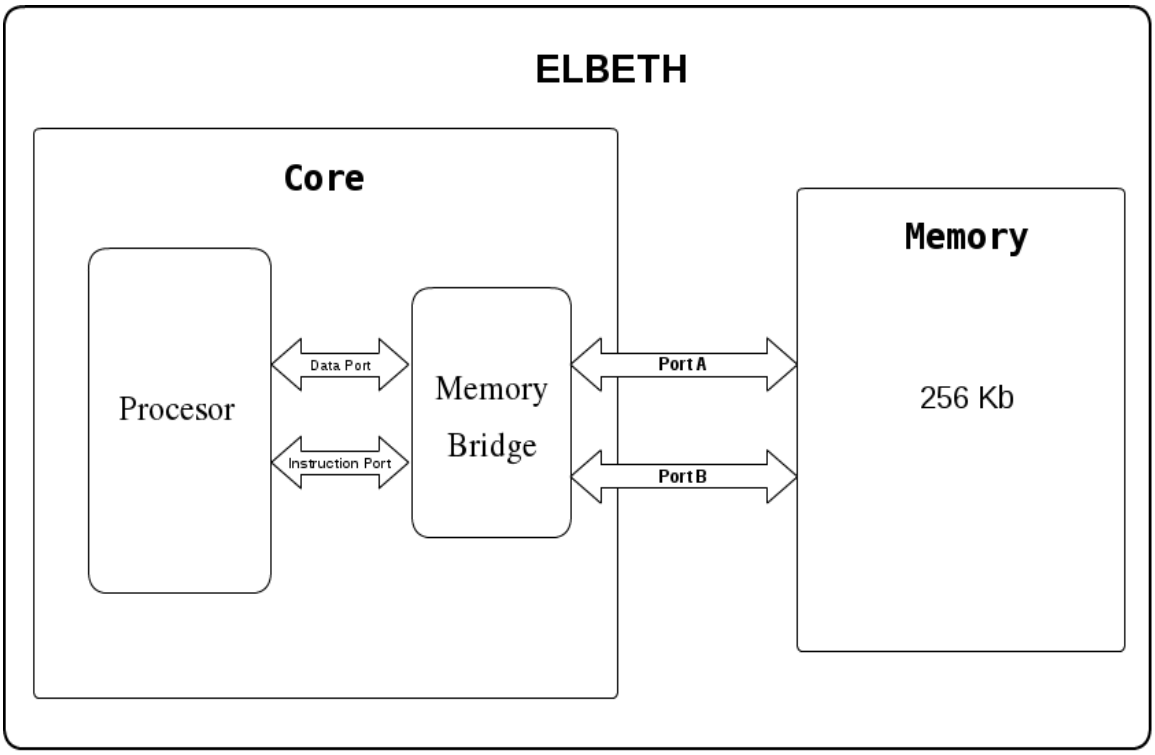

Figura 1. Diagrama de Bloques del Procesador-Memory Bridge-Memory RAM

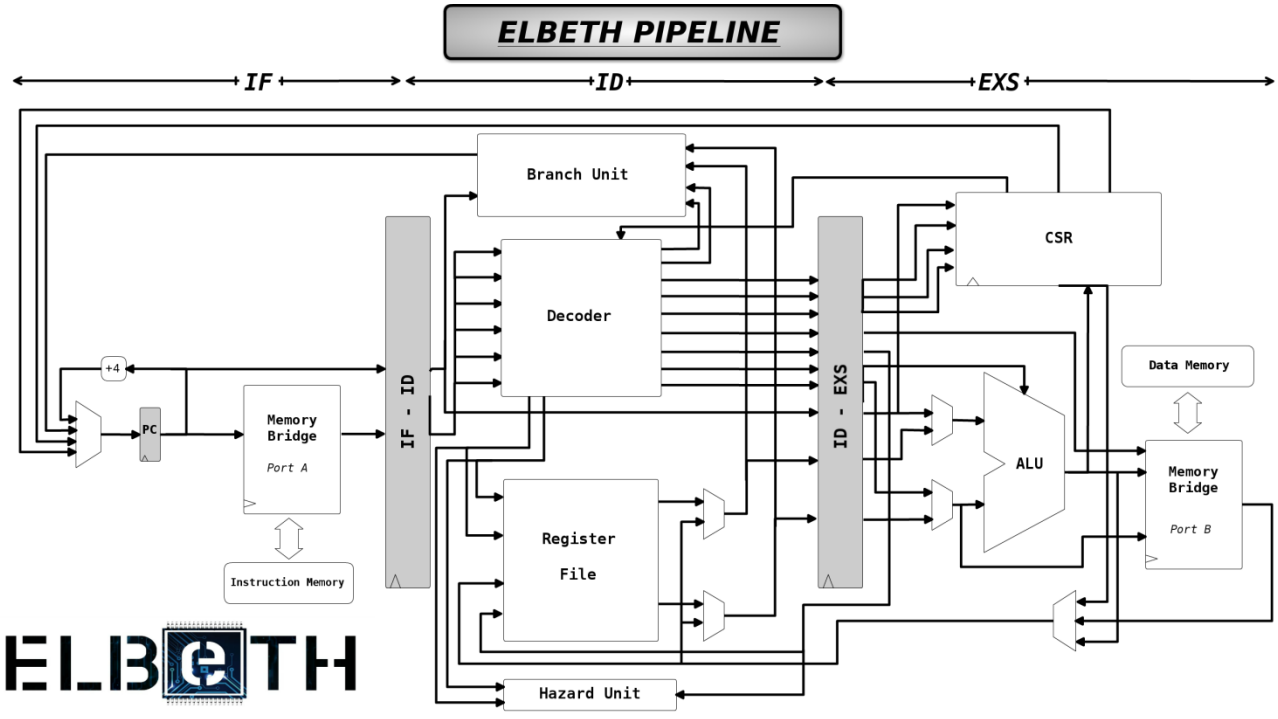

Figura 2. Pipeline de Procesador ELBETH.

3. Detalles del diseño del Procesador.

Procesador segmentado con un Pipeline de 3 etapas, más un módulo de Control, que como su nombre lo indica, controla cada una de las etapas y paredes del Pipeline. Las 3 etapas son: Instruction fetch, Instruction decode y Execution-Store.

Antes de describir estas etapas, es necesario destacar que la separación entre cada una de ellas, es debida a las paredes o registros, que van propagando los datos necesarios y las señales de control a lo largo de todo el pipeline. Debido a que el cauce es de 3 etapas solo requiere de 2 paredes o registros, llamados IF-ID y ID-EXS.

Además, es importante nombrar las señales de reloj (*CLK)* y reset (*RST*), que son indispensables para el funcionamiento del procesador y de cada una de sus etapas ya que, siguiendo el caso de los registros o paredes, tenemos que, para cada flanco de subida del *CLK* almacenan cada uno de los valores o parámetros en las entradas de las paredes, y al ocurrir el siguiente flanco, estos valores se habrán propagado a la próxima etapa por medio de la siguiente pared o registro.

Así mismo, la señal de *RST* se encuentra como entrada de las dos paredes, y de la mayoría de los módulos del procesador, ya que se debe estar disponible la posibilidad de reset.

Ahora, a continuación una breve descripción de las 3 etapas del pipeline del Procesador ELBETH, y de cada uno de los módulos que conforman dichas etapas:

3.1 Instruction Fetch Stage: Esta etapa involucra el cálculo y selección del PC junto con el *Memory Bridge* el cuál se comunica con memoria para cargar las instrucciones.

El primer elemento importante de esta etapa es el PC register el cual contiene la dirección de la instrucción que se va a ejecutar, la cual además es propagada a lo largo de todos los registros o paredes del pipeline (IF-ID register y ID-EXS register).

3.1.1 Memory Bridge – Memoria de instrucciones: Este módulo se encarga de realizar un mapeo de las direcciones (PC) de 32 bits a de 16 bits. Los datos son almacenados usando el formato Little Endian. El modelo de memoria usado en la comunicación con este módulo y en general con el procesador fue basado en la memoria del procesador MIPS ANTARES [4] en la cual cada dirección de memoria corresponde a un Word (4 bytes) y tanto la escritura como la lectura de un dato se da en dos ciclos de reloj tal como se puede apreciar en el siguiente diagrama:

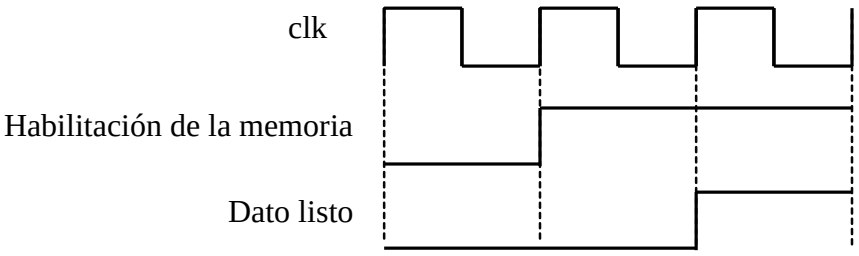

Figura 3: diagrama de lectura y escritura de memoria.

La pared o registro de separación entre esta etapa y la siguiente es IF-ID, contiene lo siguiente:

Entradas: la instrucción actual (extraída de la memoria de instrucciones), el PC asociado a esta instrucción (es decir, su dirección en memoria), la bandera y la fuente de excepción causada en la comunicación con memoria.

Salidas: las mismas entradas, si no se activan las banderas de reset o flush (propagación de parámetros).

Control: CLK, RST, flush, stall.

Además, cabe señalar que el valor del PC que se PROPAGA a través del Pipeline, es el PC actual, lo cual es un poco más ventajoso que propagar PC + 4, por lo que evitamos tener que restar luego este valor 4, cuando se retorne de un salto a subrutina.

3.2 Instruction Decoder Stage: Esta etapa se encarga de la decodificación de las instrucciones *(Decoder)*, como su nombre lo indica, la lectura y escritura en los 32 registros de propósito general (*Registers file*), la unidad que se encarga de verificar y calcular los saltos *(Branch Unit)* además de la unidad de detección de riesgos *(Hazard Unit)*.

3.2.1 Decoder: que como su nombre lo indica, decodifica la instrucción segmentada (Ver Figura 4) de le forma como se muestra a continuación:

- Número de registro rs1.
- Número de registro rs2.
- Número de registro de destino rd
- Los inmediatos, como datos, o como offset para saltos.

- Tipo de operación de la ALU.
- Tipo de comparación para el salto.
- Información del dato:
    - Byte, Halfword, Word.
    - Signed, Unsigned
- Instrucción de retorno de excepción.

Cabe destacar que este módulo genera una excepción al detectar una instrucción inválida.

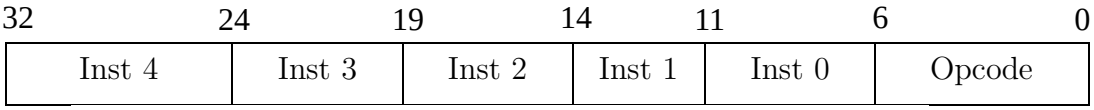

Figura 4. Segmentación de la instrucción que entra a Decoder.

3.2.2 Registers File: permite cargar valores de los 32 registros, para que sus datos puedan ser propagados hacia la pared de ID-EXS, la cual separa a esta etapa de la siguiente, y así una vez propagados a esta nueva etapa se realicen las operaciones correspondientes. Por otro lado, también se encarga de la escritura en los mismos registros antes mencionados.

3.2.3 Branch Unit: se encarga de hacer el cálculo de la nueva dirección (PC) al ejecutarse uno de los dos saltos incondicionales o al verificarse como verdadera cierta condición (“mayor que”, “menor que”, “igual a” o “diferente a”).

3.2.4 Hazard Unit: detecta si hay o no dependencia de datos comparando los registros de los cuales se van a cargar datos (ID) con el registro de destino al que se le escribirá un valor que está siendo calculado en la etapa EXS. Al detetectar dependencia esta unidad levanta una flag con el cual control procede a realizar *fordwardig* (redireccionamiento de datos).

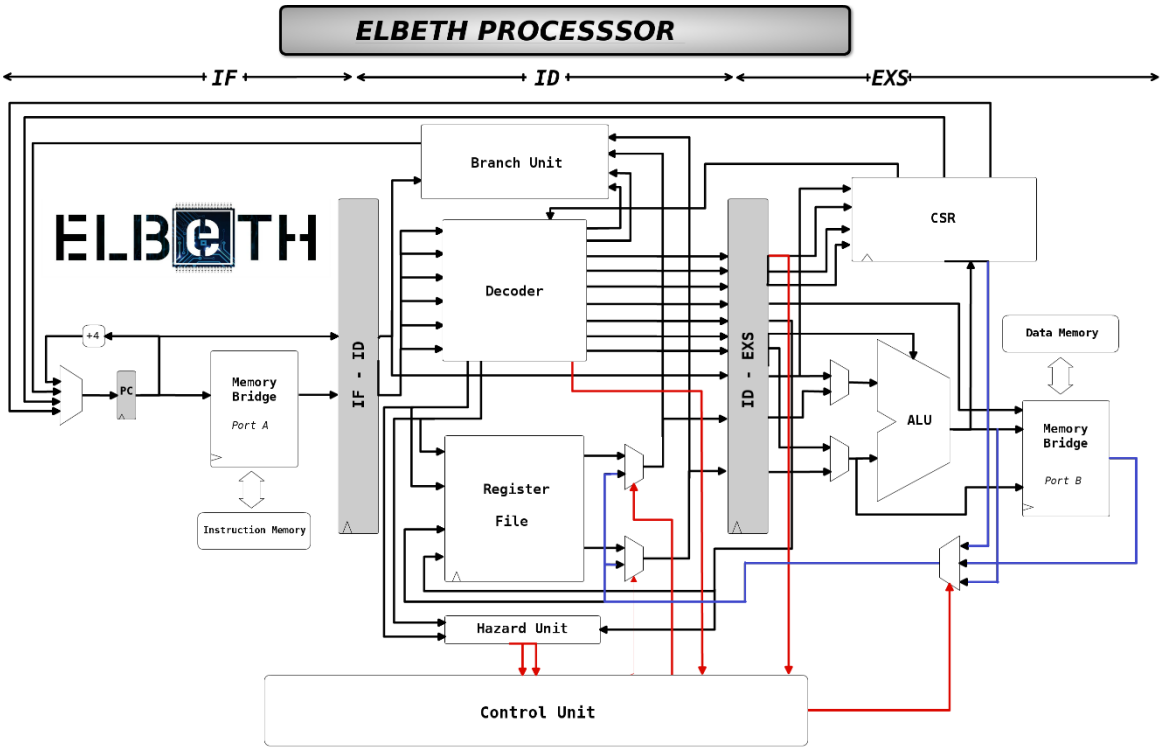

Figura 5. *Fordwarding* para procesador ELBETH

Es necesario resaltar además, la importancia que tiene la segunda y última pared o registro del Pipeline (ID-EXS). Esta etapa por ser la última, contiene toda la información de datos y parámetros importantes que han sido propagados desde la primera etapa. Pueden nombrarse resumidamente las entradas y salidas

más importantespara de esta pared *ID_EXS*:

Entradas: el PC, la operación de la ALU, el dato de rs1, el dato de rs2, el registro destino rd, el inmediato extendido a 32 bits, como dato o para calulo de saltos, los selectores para los multiplexores de rs1 y rs2, la señal de *enable* para la Memoria de datos por medio del *Bridge memory* tanto para lectura como para escritura, la informacion del tamaño y tipo de dato, las excepcionesde *Sourse* y de *CSR* y la señal de retorno de la excepcion *eret*.

Salidas: pueden ser las mismas que las entradas, dependiendo de la activación

Control: CLK, RST, flush, stall.

Para finalizar, es necesario señalar que los multiplexores de 2 a1 para rs1 y rs2, seleccionan los valores de rs1 y rs2 que van a entrar en la última pared (*ID-EXS*), dichos valores pueden venir del *Register file* o *forwarding*.

3.3 Execute Store Stage: Esta última etapa, está integrada por dos multiplexores de 2 a 1 para los dos puertos de la ALU (Unidad Aritmética Lógica), la ALU, el *Memory Bridge* conectado a la memoria de datos, un multiplexor de 3 a 1 para seleccionar el valor hacia el *Register File* y el *Control Status Register (CSR).*

3.3.1 ALU (Unidad Aritmética lógica): Esta unidad realiza las operaciones lógicas y aritméticas de los dos operandos que llegan a los puertos A y B. A la entrada de estos puertos se encuentran dos multiplexores de 2 a 1, uno para cada Puerto, de manera que se seleccione según el tipo y código de instrucción, entre el valor de un registro o el Inmediato extendido a 32 bits en el caso del primer puerto y un valor de un registro y el PC (para de las instrucciones AIUPC y LUI).

3.3.2 Memory Bridge – Memoria de datos: conectado a la memoria de datos, como se había señalado antes, es el modulo interfaz entre el Procesador y la memoria RAM. En este caso, la comunicación es hacia la memoria de datos.

3.3.3 Contol Status Register (CSR): este unidad se encarga del manejo de los niveles de privilegio así como también de las excepciones que puedan producirse en las etapas del procesador, está compuesta por varios registros dentro de los cuales se puede encontrar por ejemplo información de la cantidad de ciclos o instrucciones ejecutadas. En caso de una excepción este módulo carga una dirección (PC) a través de la cual se ejecutará un programa (ciertas instrucciones) para manejar dicha excepción, esto debe estar implementado en el archivo de memoria que se cargue al procesador.

Las excepciones soportadas por el procesador ELBETH son resumidas en el siguiente cuadro:

Código	**Causa**
0000	Dirección (PC) de la memoria de instrucción desalineada.
0001	Fallo en el acceso a un dato de la memoria de instrucción (timeout).
0010	Instrucción ilegal, mal codificada o no soportada.
0100	Dirección para cargar de la memoria de datos desalineada.
0101	Fallo en la lectura de la memoria de datos (timeout).
0110	Dirección para almacenar en la memoria de datos desalineada.
0111	Fallo en el almacenamiento de un dato (timeout).
1000	Llamada a sistema desde el *User Mode*
1011	Llamada a sistema desde el *Machine Mode*

Cuadro 2. Excepciones soportadas por ELBETH.

3.3.4 Selector de datos al *Register File* (Multiplexor de 3 a 1): selecciona el dato que será cargado al *Register File* o al *Forwarding*. Las 3 posibles salidas son: un dato proveniente de la ALU, un dato cargado de la memoria de datos o un registro leído del CSR.

3.1 Control Unit: Esta unidad, realiza la supervisión, control y selección de cada una de las señales y salidas de las etapas y cada uno de los módulos que las componen.

Las características y funcionalidades más importantes de control son las siguientes:

3.1.1 *Datapath:* Es un vector que básicamente maneja los selectores de cada uno de los multiplexores, que como habíamos señalado anteriormente, se encuentran en las etapas del pipeline, ya que la función de ellos es seleccionar una salida dependiendo del tipo de instrucción y del tipo de operación de dicha instrucción. Estos vectores también tiene información acerca de los datos que se cargan o almacenan en memoria además de las señales de habilitación de la memoria y el *Register File.*

Entre los selectores que maneja el *Datapath* están:

- El selector del multiplexor del PC, el cual puede seleccionar entre 4 opciones para la salida del *PC Register,* dichas opciones son: PC+4 (es decir la dirección de la instrucción siguiente), la direccion de un *Branch,* la dirección de *exception handler* (en la que se encuentra guardada un programa para manejar la excepción) y como ultima opción, la dirección de retorno de la excepción.
- Los selectores de los multiplexores de rs1 y rs2, para elegir dependiendo el caso, si el valor de cada uno de estos registros venia del *Register file* o del *fordwarding* de la unidad de riesgo (*Hazard unit*).
- El selector del multiplexor del Puerto A, para el cual se tienen dos opciones para salida. La primera opción, es que la entrada al Puerto A sea el valor del registro rs1, y la segunda, es que sea el contenido del registro PC. Esta última opción, es válida para cuando se ejecutan instrucciones del tipo *Jump* o *Branch*, con lo cual se puede calcular en la ALU la nueva dirección del PC, sumándole el *offset* que corresponda.

- El selector del multiplexor del Puerto B, el cual también tiene dos opciones. Una es el valor del registro rs2, y la otra es el valor inmediato que se haya propagado hasta la pared o registro de *ID_EXS*.
- El selector para el resultado final del pipeline, el cual puede venir de tres salidas: el primero seria del resultado de la ALU, el segundo de la salida del CSR el tercero de la salida del *Memory Bridge* conectado a la memoria de datos.

3.1.2 *Los **Stall** y **Flush** en IF-ID y ID-EXS:* Tanto los que pueden generarse en cualquiera de las etapas.

Una señal de *Flush* se da cuando se activa cualquiera de las siguientes señales: *Reset*, excepción o *eret* (señal que indica el retorno de una excepción), *branch taken* (señal que indica que un salto ha sido tomado y sólo genera *flush* en IF).

Un *Stall* se genera por las mismas razones para todas las etapas. Estas razones son: cuando la memoria de instrucción o de datos está cargando o escribiendo un dato (cuando el *flag* de *ready* está abajo). También puede darse cuando se levanta el *flag* de retorno de una excepción (*eret).*

Así mismo, "detener una etapa del pipeline" (*Stall)* puede entenderse como simplemente no dejar pasar una instrucción de una etapa a la siguiente, hasta que no se cumplan las condiciones para dejar fluir nuevamente el pipeline. Mientras que en términos de codificacion, se trata de un ciclo finito, dentro del cual se asignan los mismos valores a las paredes o registros del pipeline, y sólo las condiciones de parada del ciclo, permitirán actualizar dichos valores.

Por otro lado, una señal de *Flush* básicamente "limpia" las paredes o registros del pipeline. En términos de Harware puede decirse que "limpiar una pared o registro de pipeline" implica vaciar el contenido que hasta el momento tenían esos registros. Y en términos de codificación, una señal de *Flush* activa es generar a partir de esta pared un *NOP* el cual es implementado como una escritura en el registro *cero* (r0) la suma de r0 con r0.

Así mismo, el Control realiza elección de los *flags* de excepción y sus fuentes tomando en cuenta las señales que se van propagando por cada etapa, y que son almacenadas en las paredes del pipeline.

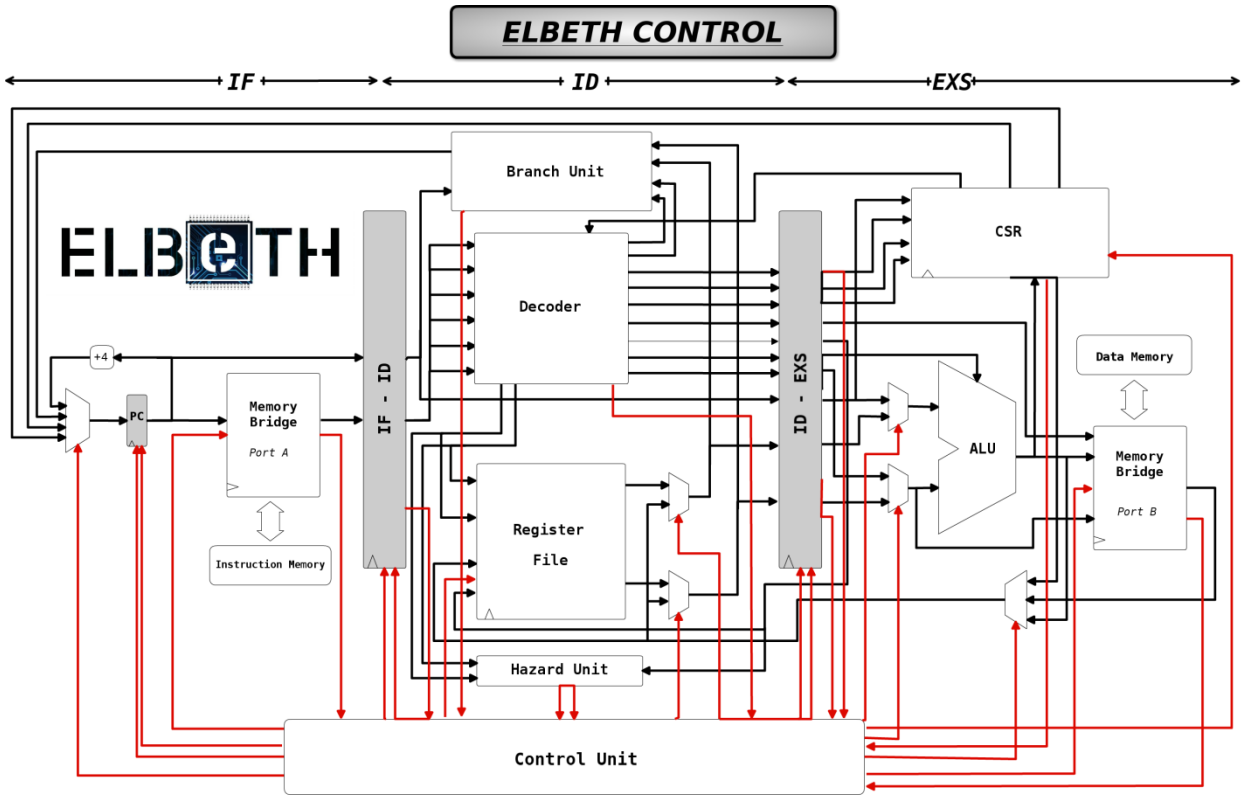

Figura 6. *Control* en el procesador ELBETH.

4. Proceso de Simulación

Al inicio de las pruebas comenzaron realizándose *tests bench* en Verilog de este modo se probaron las operaciones básicas de la ALU y algunos otros módulos, sin embargo, debido a su tediosa escritura y grandes cantidades de líneas en código, se hizo la integración de Python y Verilog para realizar pruebas más eficientes, profundas y fáciles de escribir en código.

Esta integración se realizó a través de la cosimulación que ofrece la librería MyHDL de Python. El procedimiento para ejecutar este proceso es descrito de forma detallada en el archivo README de la carpeta *simulation* de la fuente del procesador.

A continuación es presentado un esquema facilita la compresión de este proceso. Además describe los módulos necesarios para realizarlo junto sus correspondientes lenguajes, las herramientas implementadas y especifica todo esto aplicado al caso del presente proyecto.

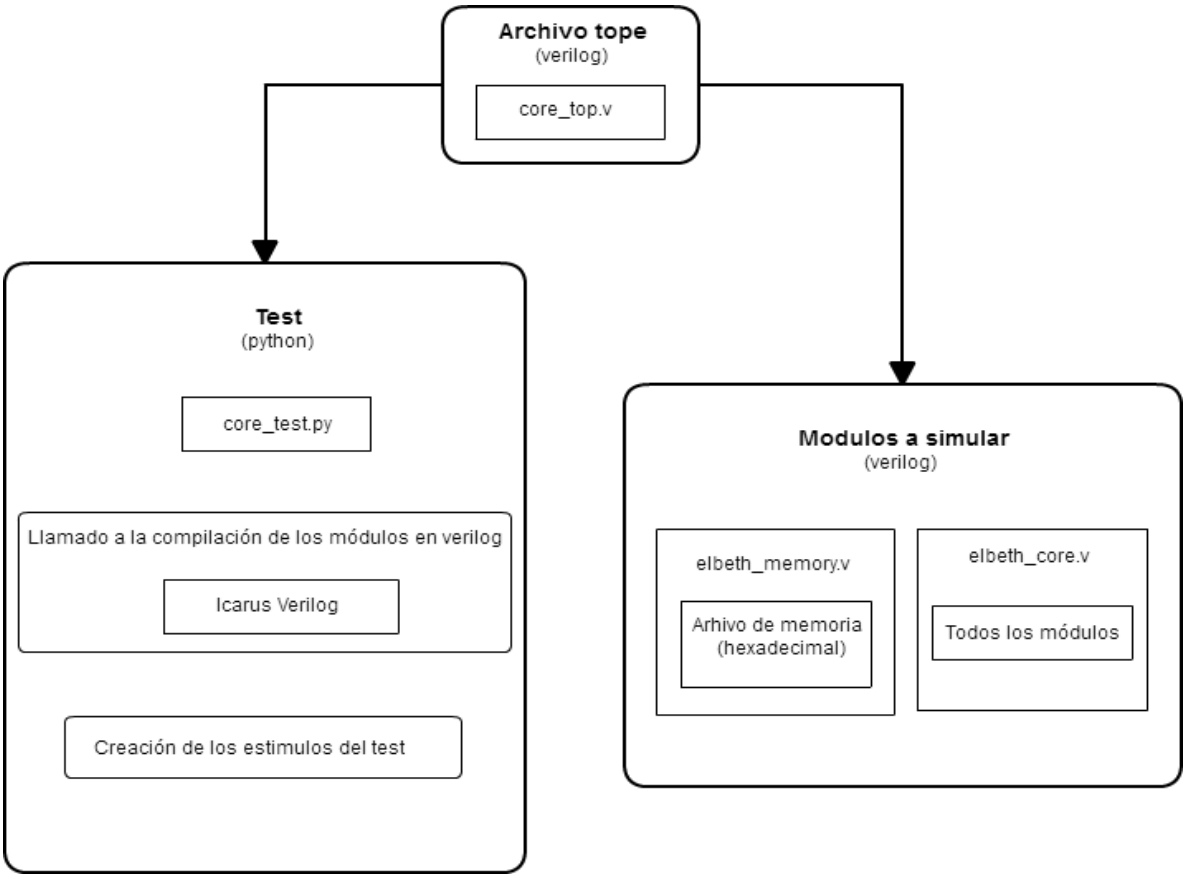

Figura 7. Esquema de Co-simulación.

Los archivos mencionados en el diagrama anterior se encuentran en la carpeta: */simulation/cosimulation/core* de la fuente de ELBETH. El proceso desarrollado es el siguiente: se sintetizan los módulos en Verilog a través de Icarus Verilog [1] y se generan los estímulos en lenguaje Python, los resultados de estas tarea y esta instancia son pasados como argumentos a la clase *simulation* de MyHDL la cuál construye (traza) y retorna las señales para ser visualizadas a través de un archivo VCD en GTKWave [2]. Del lado de Verilog se tiene que en el archivo tope se escriben los siguientes comandos $to_mydl() $from_myhdl(), los cuales reciben a la vez los correspondientes argumentos. En el primero aquellas señales que van de los módulos Verilog al test en Python (En el presente proyecto: *toHost* que es una conexión al registro que lleva el mismo nombre en el *CSR*, es un registro para fines de simulación) y en el segundo las señales que van en sentido contrario (clk, rst).

Otro modulo que fue probado a través de este mismo proceso fue la ALU (carpeta: */simulation/cosimulation/alu).* En Python (con base en un el test del Algol [5]) se seleccionaban datos de forma aleatoria y se "enviaban al procesador" para luego verificar los resultados con los pre-calculados en el mismo test (esta es quizás una de

las pruebas más vistosas de la cosimulación, luego del test del core-completo ya que se realizó la verificación de los datos entregados por el procesador, durante esta prueba se realizaban 1000 tests para cada operación de la ALU.

Durante todo el proceso de simulación tanto en Verilog como en Python se realizaron diferentes scripts para facilitar el proceso, estos junto a su funcionamiento se encuentran en la carpeta: */simulation/scripts*.

La prueba final que se realizó al procesador fue en base a los tests del procesador VSCALE [3]. Estos resultados son presentados en la siguiente sección.

5. Resultados al probar los Test

Puede verse en las Figuras 8, 9 y 10, el correcto funcionamiento del procesador ELBET:

Figura 8. Resultados del test final. (parte 1).

```
rv32ui-pt-sra.hex       OK
rv32ui-pt-addi.hex      OK
rv32ui-pt-sub.hex       OK
rv32ui-p-jalr.hex       OK
rv32ui-p-bne.hex        OK
rv32ui-p-bltu.hex       OK
rv32ui-p-divu.hex       FAILED
rv32ui-p-srai.hex       OK
rv32ui-p-j.hex          OK
rv32ui-pt-fence_i.hex   FAILED
rv32ui-pt-sh.hex        OK
rv32ui-p-xori.hex       OK
rv32ui-pt-and.hex       OK
rv32ui-p-amoand_w.hex   FAILED
rv32ui-pt-lh.hex        OK
rv32ui-pt-sb.hex        OK
rv32ui-p-sb.hex         OK
rv32ui-pt-amominu_w.hex FAILED
rv32ui-pt-rem.hex       FAILED
rv32ui-pt-lhu.hex       OK
rv32ui-p-mulhu.hex      FAILED
rv32ui-pt-amoor_w.hex   FAILED
rv32ui-pt-amoand_w.hex  FAILED
rv32ui-p-amomin_w.hex   FAILED
rv32ui-pt-lb.hex        OK
rv32ui-pt-simple.hex    OK
rv32ui-pt-andi.hex      OK
rv32ui-pt-sll.hex       OK
rv32ui-pt-blt.hex       OK
rv32ui-pt-slt.hex       OK
rv32ui-pt-add.hex       OK
rv32ui-p-mulhsu.hex     FAILED
rv32ui-pt-divu.hex      FAILED
rv32ui-p-or.hex         OK
rv32ui-p-xor.hex        OK
rv32ui-pt-mulhu.hex     FAILED
rv32ui-pt-beq.hex       OK
rv32ui-pt-amomin_w.hex  FAILED
rv32ui-pt-bgeu.hex      OK
rv32ui-pt-auipc.hex     OK
rv32ui-p-mulh.hex       FAILED
```

Figura 9. Resultados del test final. (parte 2).

```
rv32ui-p-amoadd_w.hex   FAILED
rv32ui-pt-jal.hex       OK
rv32ui-p-lh.hex         OK
rv32ui-p-srli.hex       OK
rv32ui-pt-mulh.hex      FAILED
rv32ui-p-andi.hex       OK
rv32ui-p-bge.hex        OK
rv32ui-p-srl.hex        OK
rv32ui-pt-amoswap_w.hex FAILED
rv32ui-pt-slti.hex      OK
rv32ui-pt-or.hex        OK
rv32ui-p-mul.hex        FAILED
rv32ui-p-add.hex        OK
rv32ui-p-lb.hex         OK
rv32ui-pt-lw.hex        OK
rv32ui-pt-j.hex         OK
rv32ui-p-blt.hex        OK
rv32ui-p-amominu_w.hex  FAILED
rv32ui-pt-lui.hex       OK
rv32ui-p-jal.hex        OK
rv32ui-p-div.hex        FAILED
rv32ui-p-slli.hex       OK
rv32ui-p-slti.hex       OK
rv32ui-pt-mul.hex       FAILED
rv32ui-p-lw.hex         OK
rv32ui-pt-jalr.hex      OK
rv32ui-p-remu.hex       FAILED
rv32ui-p-amomax_w.hex   FAILED
rv32ui-pt-remu.hex      FAILED
rv32ui-p-slt.hex        OK
rv32ui-pt-srli.hex      OK
------------------------------------------------------------
        RESUMEN:
        Total tests: 109.
        Failed tests: 35.
        Succes tests: 74.
------------------------------------------------------------

Done.
```

Figura 8. Resultados del test final. (parte 3).

Puede verificarse, que los tests que no fueron exitosos son aquellos que prueban las Unidades de multiplicación y división, las operaciones atómicas y las instrucciones fence, que no han sido aún diseñadas ni implementadas para el procesador ELBETH,

pero que próximas versiones pueden ser incluidas.

6. Comentarios Adicionales

Es necesario indicar que hay muchas optimizaciones que pueden hacerse al código del procesador, además, se puede realizar la implementación de otros módulos como una memoria cache, un predictor de saltos, una unidad de multiplicación y otra de división, entre otras mejoras.

Para información más detallada acerca del *ISA* puede revisarse los manuales [7, 8] correspondientes.

Para más información acerca de cómo simular el procesador ELBETH revisar los archivos README incluidos en el siguiente repositorio en el que se encuentra el mismo:

https://bitbucket.org/Emanuel84/elbeth

Referencias

- [1] http://gtkwave.sourceforge.net/gtkwave.pdf
- [2] http://iverilog.icarus.com/
- [3] https://github.com/ucb-bar/vscale/tree/master/src/test/inputs
- [5] https://github.com/AngelTerrones/Antares
- [6] https://github.com/AngelTerrones/Algol
- [7] http://www.eecs.berkeley.edu/~krste/papers/riscv-priv-spec-1.7.pdf
- [8] http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf
- [9] http://docs.myhdl.org/en/stable/manual/index.html