Forma

Descripción generada automáticamente

REPORTE PRÁCTICA FINAL: Creación de CPU dentro de software de simulación Vivado - Simulink

Autores:

Victor Abraham

Sebastian Joel Rivett Macias

Profesor:

Dr. Rene Cumplido

Materia:

Arquitectura en Computadoras

Estructura:

1) Recursos De Entrada Utilizados.

2) Descripción General Del Funcionamiento Del Circuito Completo.

3) Descripción Específica Del Funcionamiento De Cada Bloque.

4) Resultados De Funcionamiento.

5) Discusión y Conclusiones.

1) Recursos De Entrada Utilizados.

Para la parte de los recursos de entrada o mejor dicho recursos adicionales que fueron de ayuda a la hora de construir ciertas partes de la CPU simulada.

Tabla de instrucciones seleccionadas y sus correspondientes

Programa de Python para ensamblar código ensamblador: En esta parte fue necesario hacer uso de Python como lenguaje intermediario para mediante la obtención de un archivo de texto el cual cuente con instrucción.

2) Descripción General Del Funcionamiento Del Circuito Completo.

Se trata de una CPU que es capaz de ejecutar el siguiente conjunto de instrucciones:

3) Descripción Específica Del Funcionamiento De Cada Bloque.

BLOQUE MOV:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

**ENTRADAS:**

*PARTITION:* Tamaño de 2 bits, parámetro que permite seleccionar la región que se modificara correspondiente a dividir el valor del número de 64 bits en 4 regiones de 16 bits.

*Z\_K:* Tamaño de 2 bits, parámetro que permite seleccionar que tipo de instrucción se va a ejecutar, instrucción tipo MOVZ, o MOVK.

*IMM:* Tamaño de 16 bits, parámetro que considera la constante inmediata con la cual se trabajara en una de las 4 regiones seleccionadas.

*DATA\_REGISTER:* Tamaño de 64 bits, registro de entrada que correspondería al valor que se desea aplicar la instrucción de MOV correspondiente seleccionada.

**SALIDAS:**

MOV\_OUT: Tamaño de 64 bits, registro de salida al cual se le aplico la operación correspondiente seleccionada de MOV, ya sea MOVZ o MOVK.

Diagrama

Descripción generada automáticamente

Se creo un bloque que fuera multipropósito adyacente a la ALU, el cual permitiera hacer uso de las operaciones MOVZ y MOVK, referentes al conjunto de instrucciones LEGv8, este bloque tiene la capacidad de considerar las 4 regiones referentes a estos comandos en ensamblador ( 0 , 1 , 2 , 3 ), y en base a considerar si se cuenta con una instrucción tipo MOVZ entonces todas las regiones que no sean seleccionadas se consideraran como cero menos la región seleccionada de 16 bits y a esta misma región se le escribira el valor que exista en la entrada IMM, en el caso de MOVK en lugar de que todas las regiones no seleccionadas se consideren como cero se consideran con el valor del registro o no se modifican salvo la región seleccionada, se hace uso de celdas de tipo convert o cast, ya que en algunas ocasiones en algunas de las pruebas realizadas se generaban errores que no permitían hacer uso de las funciones correspondientes, este tipo de bloques permitiría corregir los errores justo antes de realizar la concatenación pertinente, generando así el número que interesa como el resultado final correspondiente a las instrucciones seleccionadas.

Cabe destacar que este bloque de operaciones cuenta con una señal de control que permite seleccionar si se elige operar con este bloque cual de las 2 operaciones se va a realizar, esta señal de control agregada tiene el nombre de Z\_K, siendo 0 el caso donde se aplicaría la instrucción MOVZ, mientras que el caso 1 donde se ejecuta la instrucción MOVK.

BLOQUE FORWARDING CONTROL:

Imagen que contiene Diagrama

Descripción generada automáticamente

**ENTRADAS:**

*Rn*: Registro de 64 bits que corresponde al registro que se esta ejecutando en la instrucción de manera actual en la tercera etapa del pipeline.

*Rm*: Registro de 64 bits que corresponde al registro que se está ejecutando en la instrucción de manera actual en la tercera etapa del pipeline. (De ser necesario).

*EX\_MEM\_Rd*: Registro de 64 bits que corresponde al registro que se está ejecutando en la etapa del pipeline correspondiente a EX\_MEM o ejecución a memoria.

*MEM\_WB\_Rd*: Registro de 64 bits que corresponde al registro que se está ejecutando en la etapa del pipeline correspondiente a MEM\_WB o memoria a escritura de memoria.

**SALIDAS:**

*Forward\_A*: Señal de control de 2 bits que permite identificar el estatus de cuál sería el registro que coincide con que etapa de cambio del pipeline, considerando como base el registro Rn.

*Forward\_B*: Señal de control de 2 bits que permite identificar el estatus de cuál sería el registro que coincide con que etapa de cambio del pipeline, considerando como base el registro Rm.

Texto

Descripción generada automáticamente

Se declaran las variables ForwardA y ForwardB como variables a utilizar en vivado de 2 bits cada una e inicializadas cada una en cero, según que caso se tenga se activaran cada una de las salidas con su correspondiente señal de 2 bits.

Si se tiene el caso de que Rn sea igual a EX\_MEM\_Rd, entonces la variable ForwardA actualizara su valor a 1. Por otra parte si se cuenta con el caso de que Rm sea igual a EX\_MEM\_Rd , entonces la variable ForwardB actualizara su valor a 1.

Los mismos casos ocurren pero ahora considerando la parte de MEM\_WB\_Rd, si se obtiene que el registro que se tiene en este lugar de transición del pipeline corresponde ya sea a Rn o Rm, los valores correspondientes a ForwardA y ForwardB cambiaran a 2 respectivamente según que caso se cumpla.

Si no se tiene que ninguno de los registros correspondientes a las transiciones del pipeline de 5 etapas EX\_MEM\_Rd y MEM\_WB\_Rd corresponda a que concuerde o sea idéntico a Rn o Rm las señales de control de ForwardA y ForwardB se mantendrán como cero o sin cambios.

BLOQUE ALU:

Diagrama, Esquemático

Descripción generada automáticamente

**ENTRADAS:**

*A\_INPUT:* Entrada de un registro de 64 bits, cabe destacar si se considera la parte de un valor inmediato se hará uso del bloque de extensión de signo.

*B\_INPUT:* Entrada de un registro de 64 bits, cabe destacar si se considera la parte de un valor inmediato se hará uso del bloque de extensión de signo.

*OPERATION:* Entrada de 2 bits que permite ejecutar una de las siguientes 4 operaciones según que número se tenga (0 – AND, 1 – OR, 2 – SUMA, 3 – XOR).

*A\_INVERT:* Entrada correspondiente a 1 bit que permite determinar si la entrada correspondiente a A\_INPUT será invertida dentro de la ALU, con este control se pueden considerar las operaciones inversas de las operaciones que se tienen dentro de la entrada OPERATION (0 – NOR, 1 – NAND, 2 – RESTA, 3 – XNOR).

*B\_INVERT:* Entrada correspondiente a 1 bit que permite determinar si la entrada correspondiente a A\_INPUT será invertida dentro de la ALU, con este control se pueden considerar las operaciones inversas de las operaciones que se tienen dentro de la entrada OPERATION (0 – NOR, 1 – NAND, 2 – SUMA NEGATIVA, 3 – XNOR).

**SALIDAS:**

*ALU\_OUT*: Salida de la ALU de 64 bits, corresponderá a un número de 64 bits el cual corresponderá a la operación seleccionada de la ALU, controlada por las entradas de OPERATION, A\_INVERT, B\_INVERT.

*ZERO:* Salida de la ALU de 1 bit, que permite determinar si se cuenta con un valor de cero al final de la operación realizada de la ALU, salida que se utiliza para considerar la parte del salto condicional si se tiene cero o como “bandera” para algunas operaciones en concreto.

*CARRY\_OUT*: Salida de la ALU de 1 bit que permite determinar si se cuenta con un bit acarreado en el último bit más significativo, para las cuestiones de la CPU creada en la práctica se despreciara.

Diagrama

Descripción generada automáticamente

El proceso para la creación de la ALU se origino de varias maneras, fue necesario considerar 2 modelos grandes de ALUS’s de 64 bits, ya que detectamos algunos errores correspondientes a los datos que no hacían sentido con lo que se debería de tener, correspondiendo más que nada en su mayoría a errores de la clase tipo de dato y conversiones automáticas. Esta parte de abordo dentro de las discusiones de una manera más amplia.

Para la parte de la construcción de la ALU de 64 bits se construyo haciendo uso de 4 ALU’s de 16 bits, de las cuales las mismas se construyeron haciendo uso de 4 ALU’s de 4 bits, las cuales a su vez están compuestas de 4 ALU’s de 1 bit cada una, con esta construcción pudimos notar que los tiempos de compilación del programa se reducían de manera significativa.

BLOQUE DE MEMORIA:

Gráfico, Gráfico de rectángulos

Descripción generada automáticamente

**ENTRADAS:**

*Write Data*: Entrada de 64 bits que indica cual será el valor que se escribira en la dirección de memoria dictaminado por la entrada Address.

*Address*: Entrada de dirección de memoria RAM en la cual se realizará la escritura de ser necesario.

*MemRead*: Entrada de carácter booleano que permite decidir si se realizara lectura en la memoria o no, si no se realiza lectura es necesario saber ya que se pueden eficientizar de cierta forma las operaciones.

*MemWrite*: Entrada de carácter booleano que permite decidir si se realizara escritura en la memoria o no, si no se realiza lectura es necesario saber ya que se pueden eficientizar de cierta forma las operaciones.

**SALIDAS:**

*Read data*: Si se cuenta con la entrada de MemRead en verdadero o 1, cual será la dirección que considerando Address se leerá, esta salida será la lectura realizada en memoria devolviendo el valor almacenado en memoria.

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

Para la parte del funcionamiento de la memoria se hizo uso de un bloque con el cual cuenta el programa utilizado para realizar la simulación, en este caso se simula el comportamiento de una memoria RAM, considerando esta como la “memoria” a la que se busca escribir o leer considerando el ejemplo o aplicación de los arreglos en ensamblador.

En este caso fue por mera simplicidad, ya que los tiempos de ejecución en promedio se elevaban bastante a la hora de considerar diferentes estructuras más complejas directas, haciendo uso de la mayor cantidad de subsistemas posibles detectamos que el problema se puede ejecutar de mejor manera.

El bloque de RAM simulado cuenta con 512 Kb en total de memoria disponible para hacer uso de ella.

BLOQUE INSTRUCTION MEMORY:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

En el bloque correspondiente a la memoria de instrucciones es el lugar donde se decodifica la instrucción en base a su opcode y los argumentos necesarios para considerar que registros se leen, si se hace uso de la memoria de alguna manera o adicional si se cuenta con algún valor inmmediate de 16 bits que se utiliza para considerar una instrucción de tipo I. Dentro de este mismo bloque se tiene conexión de forma directa con la etapa de control la cual fungiría como el discriminante para verificar cual es la instrucción que se va a verificar y según qué orden activaría las señales correspondientes.

Este bloque se encuentra localizado en la segunda etapa del pipeline de 5 etapas.

BLOQUE DE SEÑALES DE CONTROL:

Diagrama

Descripción generada automáticamente

**ENTRADAS:**

*OPCODE*: Se trata de una cadena de bits variable, según de que formato se trate se puede obtener los bits que permiten discernir de que instrucción se trata.

**SALIDAS:**

*REG2LOC*: Se trata de una variable tipo booleana, se coloca un cast para realizar la conversión adecuada ya que en algunas ocasiones según que ejecución de prueba se generaba un error, la variable sirve para considerar si un registro ira a una localización de memoria.

*ALUOP*: Se trata de una variable de un bitstream de 2 bits, esta como tal nos permite controlar cual será la operación que realizará la ALU, esto considerando si se trata de una operación tipo AND, OR, SUMA, XOR.

*ALUSRC*: Se trata de una variable tipo booleana, se coloca un cast para realizar la conversión adecuada ya que en algunas ocasiones según que ejecución de prueba se generaba un error, la variable sirve para considerar las fuentes de las operaciones de la ALU, esto sirve según el tipo de operación e instrucción más que nada es aplicado en instrucciones de tipo inmediato.

*BRANCH*: Variable tipo booleana, se coloca un cast para realizar la conversión adecuada ya que en algunas ocasiones según que ejecución de prueba se generaba un error, la variable sirve para considerar si se cuenta con un salto condicional con cero, esta misma va acompañada del resultado inmediato de la ALU el cual deberá o no de ser cero para realizar el salto correspondiente.

*MEMREAD*: Variable tipo booleana, se coloca un cast para realizar la conversión adecuada ya que en algunas ocasiones según que ejecución de prueba se generaba un error, la variable sirve para considerar si se debe de realizar algún tipo de lectura de memoria y en cual dirección considerando el bloque de memoria correspondiente.

*MEMWRITE*: Variable tipo booleana, se coloca un cast para realizar la conversión adecuada ya que en algunas ocasiones según que ejecución de prueba se generaba un error, la variable sirve para considerar si realizara una escritura sobre la memoria y en cual dirección considerando el bloque de memoria correspondiente.

*REGWRITE*: Variable tipo booleana, se coloca un cast para realizar la conversión adecuada ya que en algunas ocasiones según que ejecución de prueba se generaba un error, la variable sirve para considerar si realizara algún tipo de escritura en la memoria desde algún registro correspondiente al bloque de registros.

*MEMTOREG*: Variable de 2 bits de largo que permite la selección del tipo de registro y de que parte es considerado, si se trata un registro que es leído de forma directa de la memoria, directamente desde la salida de la ALU, o directamente desde la salida del bloque de MOV que es descrito más adelante.

*SEXT*: Variable que consta de 2 bits de largo, esta misma sirve como la variable control descrita como entrada para el bloque de extensión de signo el cual considera 4 formatos de instrucciones para realizar una normalización a una cadena de 64 bits para considerar el valor inmediato correspondiente según la localización de este mismo en que formato de instrucción.

*UBRANCH*: Variable tipo booleana, se coloca un cast para realizar la conversión adecuada ya que en algunas ocasiones según que ejecución de prueba se generaba un error, la variable sirve para considerar si realizara un salto sin condiciones el cual afecta de manera directa al program counter, hay que tener demasiado cuidad con esta parte ya que puede afectar el funcionamiento de este, y a su vez la cantidad de instrucciones que se puedan considerar.

Texto

Descripción generada automáticamente

Considerando cada uno de los formatos de instrucciones que se tienen según su opcode, es posible tomar en cuenta los estados correspondientes a cada una de las salidas de control, esto nos permitiría mantener un control riguroso sobre el comportamiento de la CPU bajo las condiciones adecuadas, según que variable de control se este trabajando en este caso existirán algunas activaciones o desactivaciones a nivel de bit, lo cual permitirá dictaminar el comportamiento según que se considere en que señal de control, estas mismas cabe destacar que serán consideradas como inputs para los bloques descritos posteriormente.

BLOQUE REGISTROS:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**ENTRADAS:**

*READ\_REGISTER\_01*: Entrada que corresponde a los 5 bits de la dirección de registro que se busca leer, o en este caso correspondería a un número del 0 al 31 para realizar la lectura de dicho registro siendo en este caso el registro de tipo 1 para la instrucción correspondiente.

*READ\_REGISTER\_02*: Entrada que corresponde a los 5 bits de la dirección de registro que se busca leer, o en este caso correspondería a un número del 0 al 31 para realizar la lectura de dicho registro siendo en este caso el registro de tipo 2 para la instrucción correspondiente. En esta parte esta lectura se realiza de forma directa paralela a la lectura del registro tipo 1.

*WRITE\_DATA*: Corresponde al valor en 64 bits que se escribiría en el registro que se busca actualizar según cual será el registro para seleccionar o escribir con ayuda de la entrada WRITE\_REGISTER.

*WRITE\_REGISTER*: Corresponde a un valor de 5 bits o que es lo mismo de 0 a 31 para seleccionar el registro al cual se busca realizar una modificación o actualización del valor.

*WRITE*: Entrada booleana que correspondería a saber si se busca escribir o actualizar el registro seleccionado por las 2 entradas adicionales de tipo write en la parte del archivo de registros, esto nos permite controlar de una manera más rigurosa la escritura en los registros comportándose como una variable de control.

**SALIDAS:**

*READ\_DATA\_01*: Valor asociado a la lectura del registro seleccionado en READ\_REGISTER\_01, corresponde a un valor de 64 bits.

*READ\_DATA\_02*: Valor asociado a la lectura del registro seleccionado en READ\_REGISTER\_02, corresponde a un valor de 64 bits.

Imagen que contiene Diagrama

Descripción generada automáticamente

Para la parte de escritura se cuenta con un bloque en específico que funciona como un selector el cual dependiendo de la señal booleana WRITE, se propagara a través de una de las 32 salidas que van de 0 a 31 para considerar, esta selección vendrá dada por la entrada WRITE\_REGISTER, y la data que se busque escribir en el registro seleccionado vendrá dada a su vez por la entrada de WRITE\_DATA.

Diagrama, Esquemático

Descripción generada automáticamente

Para la parte de la lectura simultanea de 2 valores del archivo de registros simplemente se tiene un multiplexor para cada una de las 2 lecturas de valores de 0 a 31 en las cuales se consideraría el registro del valor correspondiente. Se cuenta con otro bloque de forma idéntica conectado de tal manera a las salidas de los valores de los registros, cabe destacar que para este subsistema fue necesario manejar múltiples salidas para validar que se tengan los datos actualizados en los registros correspondientes.

BLOQUE ALU CONTROL:

Diagrama

Descripción generada automáticamente

**ENTRADAS:**

*ALUOP*: Entrada de 2 bits que corresponde a decir cuál será la operación que se ejecutará dentro de la ALU, permite ejecutar una de las siguientes 4 operaciones según qué número se tenga (0 – AND, 1 – OR, 2 – SUMA, 3 – XOR).

*OPCODE*: Permite determinar el tipo de instrucción, y en algunos casos hacer caso omiso de las operaciones que se pudieran llegar a requerir dentro de la ALU, ya que por ejemplo para las operaciones que involucran cargar o lectura desde memoria no es del todo necesario considerar la ALU, para la parte del MOVZ o MOVK igual. Funge principalmente como discriminante en si se utilizara la ALU o no y que salidas son necesarias de activar o no.

**SALIDAS:**

*OP0*: Variable Booleana de 1 Bit, correspondiente a la parte menos significativa de la operación que se debe de realizar en la ALU, sirve para construir de nuevo el valor correspondiente a la operación que deba de realizar la ALU la cual requiere de 2 bits para tomar en cuenta la operación seleccionada.

*OP1*: Variable Booleana de 1 Bit, correspondiente a la parte más significativa de la operación que se debe de realizar en la ALU, sirve para construir de nuevo el valor correspondiente a la operación que deba de realizar la ALU la cual requiere de 2 bits para tomar en cuenta la operación seleccionada.

*AINV*: Variable Booleana de 1 Bit, la cual servirá como entrada en la ALU, esta misma permite saber si la cadena de 64 bits que entra en la parte de la entrada de la ALU será invertida o no, esto sirve para considerar las operaciones complementarias de las lógicas NOR, NAND, XNOR, también la parte de la Resta y Suma Negada. (Considerando la entrada de la ALU A)

*BINV*: Variable Booleana de 1 Bit, la cual servirá como entrada en la ALU, esta misma permite saber si la cadena de 64 bits que entra en la parte de la entrada de la ALU será invertida o no, esto sirve para considerar las operaciones complementarias de las lógicas NOR, NAND, XNOR, también la parte de la Resta y Suma Negada. (Considerando la entrada de la ALU B)

Texto

Descripción generada automáticamente

La parte del bloque de control de la ALU es una pieza fundamental para dictaminar su comportamiento en base a condiciones booleanas conocidas según la instrucción que se busque considerar ejecutar dentro de la misma, ayuda a mantener un control en el orden de las instrucciones y esto es necesario ya que para la parte de la consideración del pipeline es fundamental mantener un orden para evitar ciclaciones u operaciones que no sirven como tal en la parte del código que se pudiera llegar a considerar.

BLOQUE SIGN EXTEND:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**ENTRADAS:**

*SELECT*: Entrada de 4 posibles combinaciones o 2 bits que permiten determinar cual de los formatos de inmediate se considerarían, esto es considerar un slices correspondiente al tipo de instrucción y cuales serian sus correspondientes coordenadas del bitstream relacionado al tipo de instrucción, en este caso existen 4 tipos de instrucción que hacen uso de este número el cual se describe con ayuda de IMM.

*IMM*: Valor correspondiente a la instrucción de 32 bits, en este caso corresponde al bit stream de la instrucción a la cual se busca obtener la constante inmediata de 16 bits.

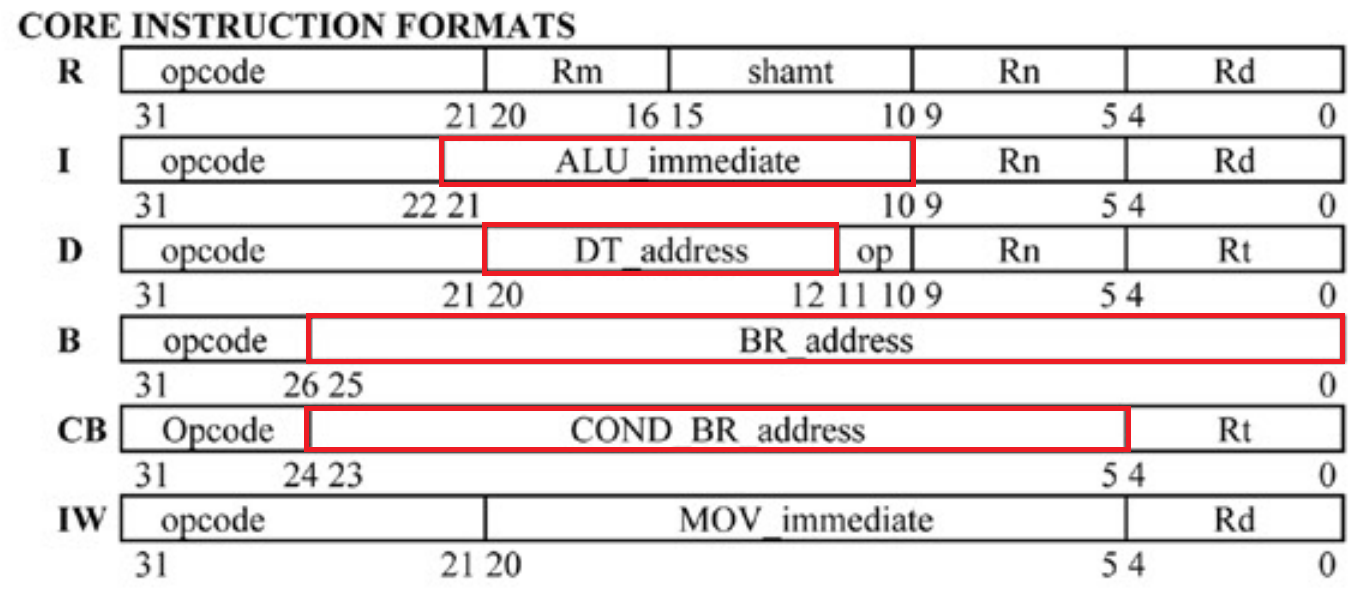
**SALIDAS:**

*RES*: Resultado de 16 bits de la constante inmediata que se busca recibir de esta parte y esta misma siendo normalizada a una constante o bitstream de 64 bits, lo que permitiría realizar operaciones de manera directa dentro de la ALU de 64 bits o considerar direcciones de memoria según la instrucción se requiera.

Diagrama

Descripción generada automáticamente

Para describir más a detalle la estructura interna del bloque de extensión de signo a 64 bits, cabe destacar que se hicieron uso de las siguientes instrucciones para considerar su constante inmediata, esto nos permitiría manejar un bitslice asociado y personalizado según que formato de instrucción se busque considerar con ayuda del multiplexor correspondiente, cabe destacar que como nos interesaba manejar constantes de 64 bits de bitstream en este caso se hizo uso de una operación OR aplicada a cada uno de los slices recuperados de cada instrucción, seguida de un cero de 64 bits de cadena de bits, esto nos permitiría obtener la cantidad de bits adecuadas y nuestra constante en este formato.



Cabe destacar que se añadió el cast correspondiente para validar que se obtenga la constante en el formato adecuado de cadena de bits, ya que en algunas ocasiones se producían errores de no existir dicho cast a 64 bits de cadenas de bits.

BLOQUE PROGRAM COUNTER:

Diagrama, Escala de tiempo

Descripción generada automáticamente

**ENTRADAS:**

*SIGN\_EXTEND*: Valor asociado a la constante inmediata considerada y normalizada en formato de 64 bits.

*U\_BRANCH*: Entrada de carácter booleano que permite determinar si se cuenta con un salto incondicional.

*BRANCH*: Entrada de carácter booleano que permite determinar si se cuenta con un salto con condicional a cero referente a la ALU o no.

*ZERO*: Entrada de carácter booleano que permite determinar si se cuenta con un zero a la ALU o no.

**SALIDAS:**

*PC*: Salida correspondiente al valor actual de salida de longitud de 64 bits referente al program counter.

Diagrama

Descripción generada automáticamente

El funcionamiento correspondiente al program counter es considerar las 4 entradas correspondientes bajo su correspondiente desfase en cada una de las entradas relacionadas a ello, esto se hizo considerando pruebas para ser capaces de operar bajo la misma cantidad de retraso en cada una de las instrucciones correspondientes, considerando esta parte es una de las condiciones para hacer que el program counter avance de manera correcta.

Cabe destacar que en esta parte fue considerar un cast para la retroalimentación del program counter, en esta parte en concreto se uso de esta manera ya que en diversas pruebas realizadas se obtenían divergencias de 4096 bits o alguna cadena de bits de diferentes longitudes a las deseadas por el mismo ciclo.

4) Resultados De Funcionamiento.

Para la parte de los resultados se hizo uso de 2 partes, primero validar todas las conexiones y que la simulación compilara, esta fue al principio la parte que más costo de realizar debido a todos los errores que se mencionan en la parte de discusión. Considerando los promedios de tiempos de compilación girando en torno a 10 minutos por run de compilación.

Para la parte del debug y realizar las validaciones de cada una de las etapas se ejecutara de manera correcta se hizo uso de una función en Matlab que permitiera generar archivos para realizar el proceso de captura de datos en base al avance de tiempo correspondiente, esto eficientizo el flujo de trabajo a la hora de capturar errores dentro de las etapas de pipeline y avanzar en la parte de la ejecución de instrucciones, así como la validación de la parte del program counter y sus correspondientes tanto saltos condicionales como incondicionales.

5) Discusión y Conclusiones.

Detección de bugs por parte del simulador Vivado.

En la creación de la CPU y realizando las pruebas de campo correspondientes, pudimos detectar un conjunto de errores, principalmente eran 2 los que predominaban con respecto al circuito:

* El tipo de datos que no era el correcto cuando se trataban cadenas de bits ya que estos mismos bitstream los convertía de forma “default” a booleano o no según que datapath se conectara y manejara.
* El segundo error que más predomino a la hora de realizar las pruebas dentro de Vivado fue un error que no permitía hacer uso de la ALU bajo determinadas condiciones, cabe destacar que el circuito de la ALU correspondiente a 64 bits fue generado considerando las pruebas correspondientes, básicamente el error consistía en que en las pruebas para forzar el error no ocurriera si no hasta generar las conexiones correspondientes en el circuito completo, se puede deber a diversos factores, ( 1 – se trabajaron los circuitos haciendo uso de las versiones de Vivado disponibles para Linux y Windows correspondientes, 2 – Se probaron diferentes arquitecturas o maneras de construir la ALU correspondiente de 64 bits y en ambas se generaba el mismo error a la hora de cablear todo, la solución fue realizar prueba a prueba del modelo correspondiente de la ALU, pruebas con ALU’s de 1 bit, 4 bits, 16 bits, 64 bits, hasta validar el comportamiento de la misma)