UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Grado En Ingeniería De Sistemas Y Tecnologías En Telecomunicación
Escuela Técnica Superior De Ingeniería De Telecomunicación

Integración de Sistemas Digitales
Tarea 3: Microcontrolador RISC-V

Integrantes:

Mario Rico Ibáñez
Laura Rivero Miró
Luis Garrido García
Sergio Moreno Suay
Carolina García Martínez

València, España Enero 2023

Índice

1. Introducción:

Este proyecto tiene como finalidad principal el aprendizaje y aplicación de los conceptos teóricos relacionados con la arquitectura RISC-V y la programación en ensamblador. El proyecto se ha desarrollado en cuatro etapas, incluyendo la implementación de instrucciones básicas, el desarrollo de una versión de single_cycle, así como una tercera fase en la que se ha hecho una versión segmentada del RISC-V y una última fase en la que se pretende implementar un programa para bajarlo a placa.

2. Fase 1

En esta primera fase se llevo a cabo el desarrollo de los elementos funcionales del microprocesador: ALU, banco de registros, y memorias (RAM y ROM) así como sus verificaciones correspondientes mediantes los respectivos testbenches básicos. Además, se programó en ensamblador bubbles y fibonacci. Vamos a destacar algunos aspectos importantes de cada módulo.

2.1. ALU

La ALU es un bloque puramente combinacional que es capaz de realizar un abanico de operaciones aritmetico-lógicas sobre dos operandos entrantes. El tamaño de los operandos es parametrizable y el tamaño del resultado es el mismo que el de los operandos.

Para que el código sea más legible se ha empleado un enum que define las operaciones posibles. Este enum es usado por todos los bloques que tienen que establecer la operación que debe realizar una ALU.

2.2. Banco de registros

El banco de registros es un módulo de treinta y dos registros de treinta y dos bits cada uno (32x32). La longitud de la palabra, es decir el tamaño de cada registro, se ha hecho parametrizable aunque para nuestro proyecto, el banco de registros va a ser en todo momento de un tamaño fijo. Se incluye en el módulo dos entradas con las direcciones de lectura, read_reg1 y read_reg2, un puerto de entrada con la dirección de escritura, write_reg, una entrada con el dato a escribir, writeData, y las dos salidas, Data1 y Data2. Por último hay una señal de enable para controlar la escritura en el banco de registros, RegWrite.

Para la versión de single-cycle el banco de registros incluye lectura asíncrona y escritura síncrona. Mientras que en la última versión de segmentado era necesario registrar las salidas así como implementar tanto lectura como escritura síncronas.

Para la parte de la verificación se hicieron comprobaciones de lectura y escritura cargando distintos datos en los registros.

2.3. Memorias

Se diseñaron dos memorias: una RAM (lectura asíncrona y escritura síncrona) y una ROM (lectura asíncrona). La RAM cuenta con enable y señal de reloj mientras que la ROM carece de reloj y enable pero cuenta con el comando de $readmemh que permite leer archivos de texto cuyo contenido está en hexadecimal. La salidas de dichas memoria en single-cycle no están registradas.

En cuanto a sus respectivos testbenchs contienen comprobaciones sencillas que nos permiten discernir si el diseño funciona correctamente o no. Para la RAM se probó lectura y escritura así como escrituras y lecturas seguidas. En el diseño de la RAM se añadieron aserciones que comprueban su correcto funcionamiento. Para la ROM se elaboró un test que facilita la lectura de dicha memoria.

2.4. Programas Fibonacci y Bubbles

Durante la fase 1 debíamos implementar dos programas en ensamblador, Fibonacci y Bubbles.

- Fibonacci: este programa constaba de un código en ensamblador que obtuviese y almacenase en memoria los N primeros términos de la sucesión de Fibonacci. Explicación del código en ensamblador: primero guardamos el valor de N en memoria, que será el número de términos que obtendremos de la secuencia. Además de inicializar dos registros al valor 1, uno de ellos será un contador para ir comparando con N, mientras que el otro es el primer término de la sucesión y del que partiremos para obtener todos los demás términos. A continuación guardamos en el sp el primer valor y desplazamos el valor de sp cuatro posiciones en memoria. Lo que ocurre a continuación es que entramos en el bucle. En este bucle lo que hacemos es primero, comparamos si ya hemos llegado al número de términos pedido, si es así saltamos al final, por el contrario si aún no hemos llegado al número de términos pedido lo que hacemos es sumar el término actual con el término anterior y el resultado lo guardamos en el sp, además de desplazar el sp cuatro posiciones de memoria y aumentar en uno el contador que regula el número de términos que ya hemos calculado de la sucesión.
- Bubbles: la función de este programa es ordenar una serie de números que están guardados en memoria de manera desordenada. Para ello, nuestro programa va realizando comparaciones entre estos números de forma iterativa y va intercambiando sus posiciones en memoria hasta que finalmente quedan ordenados. Explicación del código en ensamblador: al inicio de este código lo que tenemos es la inicialización de la secuencia además, se carga dicha secuencia en memoria de la posición 1000 hacia posiciones menores de 4 en 4. Una vez cargada toda la secuencia en memoria se entra en un bucle que procede a ordenar los valores. Para salir del Loop se ha cargado un 1 en el registro x8 y un 0 en el 7. El registro 7 será 1 cuando la ordenación sea correcta de forma que al hacer un and con el registro 7 y 8 dará 1 y salga del Loop, por lo contrario, cargará el valor de memoria en un registro e irá realizando swaps y posteriormente lo cargará en memoria con el orden correcto.

3. Fase 2

En esta segunda fase se incorporó la implementación de un modelo single-cycle.

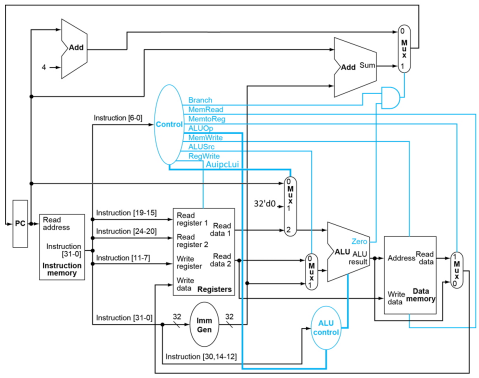

Figura 1: Esquema single-cycle

3.1. Diseño

Nuestro propuesta ha sido la siguiente, siguiendo el esquema propuesto en la tarea (Figura 1), hemos diseñado un módulo principal llamado main en el cual hemos instanciado los diferentes bloques necesarios. A continuación se incluyen los nombres de dichos módulos incluidos así como una breve descripción de cada uno de ellos. En este módulo no se ha incluido las instancias de la ram y la rom. Estas han sido incluidas en el testbench de single-cycle realizando las conexiones para su correcto funcionamiento.

- Pc_alu: sumador para incrementar el PC de 4 en 4. Las entradas son el valor del PC y un valor fijo de 10'd4.
- Control: dependiendo del tipo de instrucción, se generan las señales de control tales como BRANCH, MEM_READ, MEM_TO_REG... que gestionan los submódulos diseñados anteriormente.
- Registros: instancia del banco de registros.
- Imm_gen: genera inmediato cogiendo una sección del código en hexadecimal de la instrucción. Dicha sección depende del tipo de instrucción exceptuando a las R_FORMAT que no tienen inmediatos.
- PC_alu: Alu que suma 4 al PC.
- Alu_src_1_mux: multiplexor de 3 entradas cuya señal de selección es Auipc. La entradas son el pc, 0 y data 1 leído del banco de registros. La salida del multiplexor irá al primer operando de la ALU principal.
- Alu_src_2_mux: multiplexor de 2 entradas cuya señal de selección es alu src. Las entradas son data 2 leído del banco de registros y la salida del generador de inmediatos. La salida irá al segundo operando de la ALU principal.
- Alu_control: asigna la señal ALUselection con el tipo de operación que tiene que hacer la ALU. Esto se realiza dependiendo del tipo así como la función que desempeña la instrucción.
- Address_alu: este módulo es la ALU principal que cuenta con la señal zero, en la fase 2 una comparación exitosa significa zero = 0 pero posteriormente se cambió a cuando la comparación es exitosa zero = 1.
- Data_mux: multiplexor de 3 entradas cuya entrada de selección es MemToReg. Este multiplexor está modificado con respecto al esquema dado por la asignatura. A parte de las señales de el resultado de la ALU y del dato leído de la RAM, se ha añadido una señal que contiene el PC+4. Esta señal se ha añadido puesto que para la instrucción jal se necesitaba que se almacene en el banco de registros el PC+4. Por ello la señal de selección será de 2 bits.
- Jump_alu: ALU que suma el PC+inmediato de las instrucciones de salto.
- Pc_mux: multiplexor que selecciona entre el pc y el pc+inmediato. Escogerá esta última entrada cuando la señal zero sea 1 y quieras actualizar el pc a pc+ salto.

3.2. Verificación

Para aleatorizar la rom hemos seguido la explicación proporcionada en el blog de la asignatura. Lo primero a realizar fue el paquete en el que se encuentran las clases que generan los valores aleatorios de las instrucciones, más adelante explicaremos en que momento se produce esta generación de los estímulos. Además, dentro de la clase también se encuentran los covergroups para conocer si estamos generando todas las instrucciones con el fin de una comprobación completa del microprocesador. Para esto segundo, se han introducido aserciones dentro del diseño single_cycle para así ver si se están realizando de manera correcta. Hemos definido interfaces, para la ram y la rom, cada una presenta sus diferentes modports que serán usados dentro del testbench que vamos a generar. El program llamado estimulos, es principalmente usado con el fin de generar las señales aleatorias. Hemos creado dos principales tasks dentro de este archivo. Una task encargada de activar/desactivar las constraints que queremos o no comprobar si se están produciendo. Y otra encargada de generar las instrucciones aleatorias, y enviarlas al mailbox, que se encuentra dentro del archivo rom_aleatoria_tb. Este mailbox es un método de comunicación entre los diferentes módulos, el que genera la señal aleatoria y el que se encarga de almacenar los valores con el fin de no sobreescribir instrucciones ya generadas previamente en instrucciones de memoria. Es decir, esta rom_aleatoria va ser coherente, si el microprocesador le solicita una instrucción en la dirección 10, si por alguna casualidad, a la hora de generar estas señales aleatorias, el microprocesador vuelve a la dirección 10, le devolverá la señal previamente generada. Para finalizar, hemos creado el archivo rom_aleatoria_tb en el cual se encuentran instanciados todos los archivos previos con las conexiones pertinentes. Con el fin de que el microprocesador le solicite direcciones a la rom aleatoria y esta le responda con la instrucción generada.

Se ha comprobado el funcionamiento del single-cycle mediante el uso de los programas de fibonacci y bubbles. Se ha creado un testbench donde cargamos los archivos txt y comprobamos mediante questasim su correcto funcionamiento. En esta fase los programas incluyen instrucciones nop porque todavía no se ha implementado ningún tipo de control de riesgo y es por tanto necesario su uso.

4. Fase 3

4.1. Diseño

4.1.1. Pipelined

Se llevar a cabo la segmentación del datapath y el controlpath para implementar una pipeline de 5 etapas: búsqueda de instrucción (IF), decodificación (ID), ejecución (EX), acceso a memoria (MEM) y post-escritura (WB) para que distintos segmentos de diferentes instrucciones puedan estar trabajando simultáneamente. Para llevar a cabo la segmentación se ha trabajado a partir del modulo main de la fase dos. Tanto el banco de registros como las memorias se registraron de forma independiente en cada módulo. Para registrar las señales de cada fase se han creado cuatro módulos, uno para cada fase, llamados IF_ID_REG, ID_EX_REG, EX_MEM_REG, MEM_WB_REG, donde se han registrado las señales correspondientes de cada etapa.
Por último se han instanciado cada uno de los módulos listados anteriormente, en un único módulo juntándolo con el resto de elementos como control, alu, banco de registros y multiplexores. La ram y la rom son externas y se unen en el testbench para comprobar el funcionamiento correcto del conjunto, lo que se explicará en el apartado de verificación.

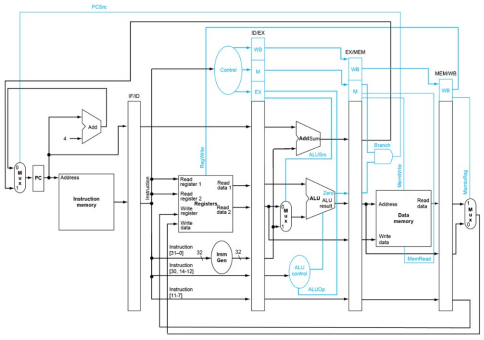

Figura 2: Esquema pipelined

4.1.2. Ampliación del juego de instrucciones

Se han añadido a la fase anterior y comprobado las siguientes instrucciones: SLLI, SRLI, SRAI, SLL, SRL, SRA, JAL, JALR, BLT, BLTU, BGE, BGEU. Para ello se han incluido las B_format.

4.1.3. Riesgo de datos.

La idea principal de este tipo de riesgo es que es necesario esperar a que una instrucción previa lleve a cabo su operación de lectura/escritura. Para ello la forma de implementarlo es identificar correctamente el tipo de forward, es decir adelanto, que necesitamos para cada instrucción, dependiendo de los registros fuente que usa la instrucción y dónde se están usando estos registros fuente anteriormente en el código ensamblador. Para ello creamos un módulo llamado data_forwarding que identifica si el registro destino de instrucciones anteriores es igual a alguno de los registros fuente de la instrucción actual, y en función de qué tipo de adelanto sea, obtendremos una señal de enable distinta. Además creamos una serie de mux, para así llevar el contenido del registro fuente que necesitamos a la entrada de la ALU dependiendo del enable del módulo data_forwarding. En nuestro diseño tuvimos que salirnos un poco de lo indicado en las diapositivas de la asignatura debido a que obtuvimos un problema porque a la vez que el valor estaba siendo guardado en un registro, se leía de ese mismo por lo que no tomaba el valor correcto. Para ello lo que hicimos fue crear un posible forward adicional, a la salida de una etapa de registros auxiliar a la que llamamos 'aux'. En esta etapa registrábamos de nuevo la salida del último mux, para así guardar un ciclo más su valor y poder así asegurar que funcionase el riesgo de datos correctamente para todos los casos.

4.1.4. Riesgo estructural

Este riesgo se presenta cuando hay conflicto en el uso de un recurso. En concreto, al usar las instrucciones load/store que requieren acceso a memoria.
Para solucionar dicho problema, se ha implementado un nuevo módulo llamado hazard_detection. En dicho módulo, cuando detecta una instrucción de carga (lw) congelará el PC y el registro de la etapa IF/ID. De esta forma cuando se quiera cargar algo en memoria esperará para seguir con la ejecución del programa. Además, se ha añadido un multiplexor tal y como indica el esquema. En este multiplexor de 2 entradas, si no se detecta una señal de carga concatenará todas las señales de control y a la salida del multiplexor se desconcatenarán. Por lo contrario, la salida del mux será 0 llegando así a la etapa de ID/EX.

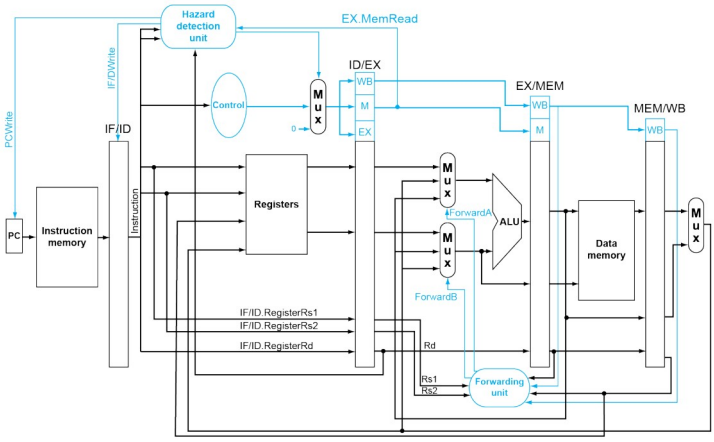

Figura 3: Esquema para solucionar el riesgo estructural

4.1.5. Riesgo de control

Para gestionar el riesgo de control, hemos creado el módulo llamada jump_predictor. La función fundamental de este módulo es intentar predecir si realizar un salto o no, tomar esta decisión y comprobar en el siguiente ciclo si la predicción ha sido correcta. De no serlo, el PC es corregido y se toman las medidas necesarias para que el error no suponga efectos secundarios sobre las memorias y registros.
El módulo implementa una predicción dinámica, es decir, es capaz de aprender para determinada dirección de salto si es conveniente saltar o no en función de los éxitos y fracasos anteriores. Esta información es guardada en un array donde el índice es la mencionada dirección de salto y el valor es un número que va del 0 al 3. Si el valor es 0 o 1 predecimos no salto, si es 2 o 3 predecimos salto.
Para actualizar el valor de array utilizamos el criterio ilustrado por este diagrama:

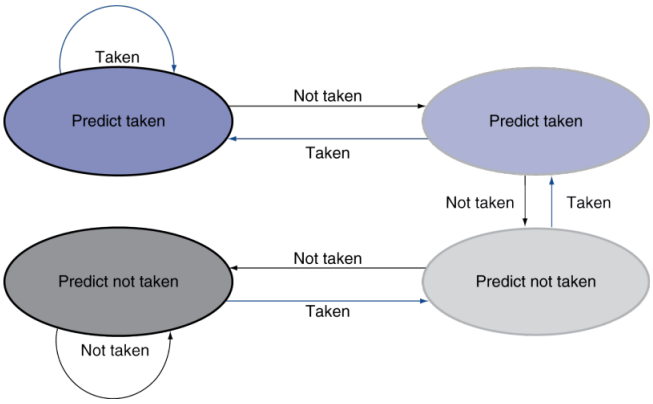

Figura 4: Algoritmo de actualización de predicción

Para realizar un salto, el módulo introduce en la ROM y en pc_id la dirección de la siguiente instrucción a ejecutar y en el PC esa misma dirección + 4. De esta manera el cambio a la instrucción tras el salto se produce inmediatamente. Averiguamos si nos hemos equivocado en el ciclo siguiente y de hacerlo debemos introducir una burbuja y saltar al PC correcto.

4.2. Verificación

4.2.1. Pipelined

Para la verificación del pipelined se ha empleado el single-cycle de la primera fase como "golden model". Para ello se ha creado un testbench en el que se han juntado el testbench del single-cycle y del pipelined.
En un único testbench se ha instanciado el módulo en el cual esta todo segmentado (pipelined), junto con la ram y la rom registradas,el main del single-cycle y con la ram y la rom de la fase 1, es decir, ram y rom sin registrar. En la rom registrada del pipelined se ha cargado el archivo txt fibonacci_pipelined que incluye nops y en la rom sin segmentar se ha cargado el programa fibonacci sin nops.
De esta forma hemos podido comparar que el funcionamiento de ambos es el mismo. La única diferencia es que cuando se comprueba en el banco de registros la secuencia de fibonacci cargada, en el caso de segmentación se puede observar como el número de ciclos que tarda es mayor que el de single-cycle, ya que se han introducido nops en el código del programa.
Esto no es lo esperado puesto que el comportamiento debería ser el contrario, sin embargo en la última fase del proyecto se alcanza el objetivo marcado, que es la reducción de ciclos al ejecutar un programa mediante la ejecución simultánea de distintas secciones de cada instrucción.

5. Fase 4

Para esta última fase hemos decido diseñar un contador binario. En primer lugar hemos tenido que reducir el tamaño de la ram y la rom para poder conseguir un tiempo de compilación menor y poder trabajar de forma más efectiva. Se ha creado un archivo llamado cochefantastico, donde se ha instanciado la versión del single-cycle junto con la RAM y la ROM sin registrar. Se ha creado una RAM directamente en el diseño para poder extraer las salidas, que será la RAM del GPIO. Se va a trabajar con la parte baja de cada RAM. Para activar los enables de las respectivas RAM, se ha diseñado un inversor que tiene como entrada el bit 6 de la dirección del single-cycle. Cuando este bit esté a uno significará que debemos escribir en la RAM del GPIO y cuando esté a cero indicará direcciones pertenecientes a la parte baja de la RAM interna. Para leer los datos que provienen de la placa se ha incluido un multiplexor con la finalidad de poder leer los valores escritos en la RAM del GPIO (como por ejemplo los valores de las señales CLK y RESET_N).

6. Comentarios finales

En primer lugar se ha conseguido desarrollar todas las fases de la tarea con sus respectivas comprobaciones (fibonacci, bubbles e implementación en placa).
En segundo lugar, cabe destacar que el proyecto ha sido planteado con ancho de bus de direcciones de memoria de 10 bits. A pesar de que todos los módulos son parametrizables, surgen ciertos problemas al cambiar este valor a 32. Esperamos que esto no suponga un inconveniente ya que todas las fases (fibonacci, bubbles y el contador binario que hemos a placa) funcionan sin problema.

Finalmente, en la fase 4, a la hora de bajar a placa, el CLK funciona con un pulsador en vez de con el reloj de la placa. No hemos conseguido cambiar la frecuencia del reloj para poder visualizar el contador binario sin el uso de un botón.