Escuela de Ingeniería Electrónica
EL3310 Diseño de Sistemas Digitales
I Semestre 2024

TEC | Tecnológico de Costa Rica

Diseño de Sistemas Digitales

Solución Proyecto 01: Procesador Monociclo

Andrés José Bonilla Blanco #2019064612, Oscar Mario Gonzalez Cambronero #2021075110

Resumen

Este documento presenta la solución del proyecto 01 del curso de Diseño de Sistemas Digitales, dividiendo el documento en cada uno de los submódulos utilizados para generar el *RTL* de la microarquitectura completa, presentando los módulos *testbench* utilizados para garantizar el funcionamiento de cada sección del procesador y realizando una breve descripción del programa implementado para demostrar el funcionamiento del procesador.

1. Repositorio del Proyecto

Todos los archivos de *RTL* y de inicialización de las memorias creados en proyecto se encuentran en la carpeta **Proyecto_1** del repositorio del curso. Este se encuentra público y se puede acceder **haciendo click aquí**.

2. Instrucciones del ISA Implementadas

La microarquitectura diseñada ejecuta un *subset* de 7 instrucciones provenientes del estándar *RISCV32i/RISCV64i* obtenido del libro [1], estas se describen a continuación:

- **lw (Cargar Palabra)**:
  - **Descripción**: Carga un valor de 64 bits desde la memoria en un registro.
  - **Sintaxis**: (lw rd, imm12(rs1))

- **Ejemplo**: (lw x3, 100(x1)) carga el valor en la dirección de memoria (x1 + 100) en el registro (x3).

- **sw (Almacenar Palabra)**:
  - **Descripción**: Almacena un valor de 64 bits desde un registro en la memoria.
  - **Sintaxis**: (sw rs2, imm12(rs1))
  - **Ejemplo**: (sw x5, 200(x4)) almacena el valor del registro (x5) en la dirección de memoria (x4 + 200).

- **beq (Salto si Igual)**:
  - **Descripción**: Realiza un salto a una dirección de destino si dos registros son iguales.
  - **Sintaxis**: (beq rs1, rs2, imm12)
  - **Ejemplo**: (beq x6, x7, 50) salta a la dirección de destino si (x6) es igual a (x7).

- **add (Sumar)**:
  - **Descripción**: Suma los valores de dos registros y almacena el resultado en un registro de destino.
  - **Sintaxis**: (add rd, rs1, rs2)
  - **Ejemplo**: (add x8, x9, x10) calcula (x8 = x9 + x10).

- **sub (Restar)**:
  - **Descripción**: Resta el valor de un registro de otro y almacena el resultado en un registro de destino.
  - **Sintaxis**: (sub rd, rs1, rs2)
  - **Ejemplo**: (sub x11, x12, x13) calcula (x11 = x12 - x13).

- **and (and a nivel de bits)**:
  - **Descripción**: Realiza una operación AND a nivel de bits entre dos registros y almacena el resultado en un registro de destino.
  - **Sintaxis**: (and rd, rs1, rs2)
  - **Ejemplo**: (and x14, x15, x16) calcula (x14 = x15 & x16).

- **or (or a nivel de bits)**:
  - **Descripción**: Realiza una operación OR a nivel de bits entre dos registros y almacena el resultado en un registro de destino.
  - **Sintaxis**: (or rd, rs1, rs2)
  - **Ejemplo**: (or x17, x18, x19) calcula (x17 = x18 | x19).

Instrucción						
lw	imm[11:0]		rs1	010	rd	0000011
sw	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
beq	imm[12 | 10:5]	rs2	rs1	000	imm[4:1 | 11]	1100011
add	0000000	rs2	rs1	000	rd	0110011
sub	0100000	rs2	rs1	000	rd	0110011
and	0000000	rs2	rs1	111	rd	0110011
or	0000000	rs2	rs1	110	rd	0110011

Cuadro 1: Tabla de Resumen de las Instrucciones RISC-V a utilizar en el procesador

3. Módulos del Datapath

3.1. Contador de Programa

El contador del programa es únicamente un registro de 64 bits que almacena el dato generado por los sumadores, posee señales se reinicio y reloj utilizadas para sincronizarse, este almacena el dato en el flanco positivo del reloj y genera la dirección para la memoria de instrcciones.

3.1.1. Prueba del Contador de Programa

Se generó una señal de reloj con la que el contador de programa almacena los datos en el flanco positivo y se verificó que la salida del registro sea acorde al dato almacenado en el flanco positivo anterior.

3.2. Módulos de Memoria

3.2.1. Memoria de Instrucciones

La memoria de instrucciones se implementó como un array de palabras de 32 bits, y de 80 palabras de largo, para que se puedan correr hasta 80 instrucciones. Esta memoria *no* es accesible por byte, es accesible por palabra, entonces se debe tener cuidado al realizar el cálculo de las direcciones a las que se quiere saltar por ejemplo con la instrucción beq. Si se quisiera que la memoria de instrucciones lograra emular el comportamiento de ser accesible por byte, para mayor compatibilidad con cualquier ensamblador, se debería hacer un bitshift a la derecha de 2 posiciones, sin extender signo.

3.2.2. Memoria de Datos

La memoria de datos se implementó como un array de palabras de 64 bits de 64 palabras de largo. Este bloque de memoria sí es accesible por byte, pero cada módulo del sistema tiene como menor

unidad de datos la palabra de 64 bits, entonces para acceder la memoria con unidades menores a 64 bits se debe realizar teniendo en cuenta que cada palabra corresponde a un dato completo en específico. En pocas palabras, la dirección de cada dato es 8*[índice de la palabra].

3.2.3. Registros

El bloque de registros se implementó haciendo un array de palabras de 64 bits, y de 32 palabras de largo. El bloque de registros *no* es accesible por byte. El que se encarga de darle las direcciones es la decodificación de la instrucción que viene desde la memoria de instrucciones.

3.3. Prueba de los Módulos de Memoria

3.3.1. Prueba de la Memoria de Instrucciones

1. **Parámetros del módulo**: El módulo `z_inst_mem_tb` tiene varios parámetros, incluyendo `width`, `depth`, `adr_in`, y `mem_shown`. Estos parámetros se utilizan para definir las características de la memoria que se está probando, y la cantidad de datos que se le muestran al usuario.

2. **Inicialización**: Se definen varias señales: incluyendo `clk`, `rst`, `read_adr`. Además, se instancia el módulo `inst_mem`.

3. **Generación de la señal de reloj**: Se genera una señal de reloj después de 5 unidades de tiempo.

4. **Simulación**: Se realiza una simulación en la que se inicializa la memoria con valores crecientes y luego se leen estos valores. Los resultados de la simulación se guardan en un archivo VCD.

5. **Finalización**: Una vez que se han leído todos los valores de la memoria, la simulación se detiene.

3.3.2. Prueba de la Memoria de Datos

1. **Parámetros del módulo**: El módulo `z_data_mem_tb` no tiene parámetros. Sin embargo, se definen varias señales, incluyendo `clk`, `rst`, `wrt_en`, `address`, `write_data` y `read_data`. Además, se instancia el módulo `data_mem`.

2. **Inicialización**: Se inicializan las señales de reloj, reset y habilitación de escritura a 0. También se establece un valor inicial para los datos de escritura.

3. **Generación de la señal de reloj**: Se genera una señal de reloj después de 5 unidades de tiempo del initial.

4. **Simulación**: Se realiza una simulación en la que se leen dos valores en la memoria en las direcciones 0 y 8 respectivamente. Luego, se habilita la escritura y se escribe el valor inicial

de `write_data` en la dirección 16. Después, se deshabilita la escritura y se leen los valores de las direcciones 0 y 16.

5. **Finalización**: Una vez que se han leído todos los valores de la memoria, la simulación se detiene.

3.3.3. Prueba de los Registros

1. **Parámetros del módulo**: El módulo `z_register_tb` tiene parámetros de ancho y profundidad, con valores predeterminados de 64 y 32 respectivamente.
2. **Señales de control**: Se definen varias señales de control, incluyendo `rst`, `clk`, y `regwrite`.
3. **Direcciones de registros**: Se definen las direcciones de los registros `adr_reg1`, `adr_reg2`, y `adr_wr_reg`.
4. **Datos de escritura**: Se define `wr_data` para los datos de escritura.
5. **Salidas de registros**: Se definen `reg_data1` y `reg_data2` como salidas de los registros.
6. **Instancia del módulo**: Se instancia el módulo `register` con los parámetros y señales definidos anteriormente.
7. **Generación de la señal de reloj**: Se genera una señal de reloj que empieza a cambiar de estado después de 5 unidades de tiempo.
8. **Inicialización**: Se inicializan las señales de control y se establecen valores iniciales para las direcciones de los registros y los datos de escritura.
9. **Simulación**: Se realiza una simulación en la que se habilita la escritura y se escribe el valor inicial de `wr_data` en la dirección `adr_wr_reg`. Luego, se cambia la dirección de lectura del registro 1 a la de `adr_wr_reg` para poder leer el dato.
10. **Finalización**: Una vez que se han realizado todas las operaciones, la simulación se detiene.

3.4. Generación de Inmediatos

El módulo de generación de inmediatos se implementó por medio de un decodificador con una estructura de *case* para poder extraer el inmediato acorde a en qué parte de la instrucción se encuentre. Para encontrar en qué lugar se encuentra el inmediato fue suficiente con ver los primeros 7 bits de la instrucción, una vez obtenido el inmediato, se le realiza la extensión a 64 bits copiando el bit más significativo. En caso de que la instrucción no requiera inmediatos el módulo genera un valor de 0 en los 64 bits.

3.4.1. Prueba Generación de inmediatos

1. **Parámetros del módulo**: El módulo `z_signextend_tb` no tiene parámetros. Las señales `IN` y `OUT` se utilizan para la entrada y salida del módulo `signextend`.
2. **Inicialización**: Se definen dos señales: `IN` y `OUT`. Además, se instancia el módulo `signextend` con `IN` y `OUT` como sus puertos.
3. **Generación de la señal de reloj**: No se genera ninguna señal de reloj en este testbench.
4. **Simulación**: Se realiza una simulación en la que se asignan diferentes valores a la señal `IN` para probar el módulo `signextend`. Los valores asignados a `IN` representan diferentes tipos de instrucciones en el código de máquina RISC-V, cada una con su inmediato en una ubicación específica. Los resultados de la simulación se guardan en un archivo VCD llamado "prueba_extend.vcd". Los valores asignados son:
   a) Un valor que no requiere inmediatos.
   b) Un valor que representa la instrucción 'lw', que tiene el inmediato en la parte superior de la instrucción.
   c) Un valor que representa la instrucción 'sw', que tiene el inmediato dividido entre la parte superior e inferior de la instrucción.
   d) Un valor que representa la instrucción 'beq', que tiene el inmediato dividido entre varias partes de la instrucción.
5. **Finalización**: Después de asignar todos los valores a la señal `IN`, la simulación se detiene.

3.5. ALU

La unidad aritmética lógica implementada realiza las operaciones de suma, resta, and (bit a bit), y or (bit a bit) que son las disponibles con las instrucciones implementadas. La codificación para las operaciones es un half-byte que puede tomar los valores 0000 ,0001 ,0010 y 0110 según la operación a realizar. El módulo fue implementado utilizando una estructura *case* y generando una bandera de cero en caso de que la operación produzca un resultado igual a cero.

3.5.1. Prueba ALU

Se generaron diferentes estímulos con los valores A y B de entrada de la ALU mientras que se colocaban diferentes bits de operación, además se verificó el funcionamiento de la bandera de cero por medio de una operación que generara un valor de cero. En caso de que se seleccionara un bit de operación que no correspondiera con ninguno de los mencionados anteriormente, la salida del resultado generada tendría todos sus bits en alto.

3.6. Sumadores

Se generaron sumadores de 64 bits que tienen una única saliendo siendo el resultado de la suma de dos entradas A y B.

3.6.1. Prueba Sumadores

Para probar el funcionamiento del sumador, se generaron 4 valores aleatorios y se verificó que el resultado de la suma fuera acorde a la suma de los dos datos utilizando la interfaz de *GTK Wave*

4. Módulo de Control

El módulo de control que proporcionaste toma una instrucción de 11 bits. Los primeros 7 bits son 'Instruction[6:0]', los siguientes tres son 'Instruction[14:12]' y el bit más significativo es 'Instruction[30]'.

Este módulo genera varias señales de control ('Branch', 'MemtoReg', 'MemWrite', 'ALUSrc', 'RegWrite') y una operación ALU ('ALUOp'). La señal 'Branch' se dirige directamente al mux21 superior.

El módulo define cuatro operaciones: 'beq_op', 'arit_op', 'sw_op' y 'lw_op'. Cada operación corresponde a un conjunto específico de señales de control y operaciones ALU.

En cada ciclo, el módulo decodifica la instrucción y establece las señales de control y las operaciones ALU correspondientes. Si la instrucción no coincide con ninguna operación definida, todas las señales de control se establecen en '1' y 'ALUOp' se establece en '1111'.

- **beq_op**: Cuando la instrucción es una operación 'beq', las señales de control se establecen para realizar una operación de bifurcación y la operación ALU se establece en '0110'.
- **sw_op**: Para una operación 'sw', las señales de control se establecen para realizar una escritura en memoria y la operación ALU se establece en '0010'.
- **lw_op**: En el caso de una operación 'lw', las señales de control se establecen para realizar una lectura de memoria y la operación ALU también se establece en '0010'.
- **arit_op**: Para las operaciones aritméticas, hay un 'case' interno que decodifica los bits 'Instruction[10:7]' de la instrucción para determinar la operación específica ('suma', 'resta', 'or', 'and'). Las señales de control se establecen para permitir la escritura en el banco de registros y la operación ALU se establece en función de la operación aritmética específica.
- **default**: Si la instrucción de entrada no coincide con ninguna de las operaciones definidas, todas las señales de control se establecen en '1' y 'ALUOp' se establece en '1111'.

4.1. Prueba del módulo de control

1. **Parámetros del módulo**: El módulo `z_control_tb` tiene un parámetro, `instruction`, que es un arreglo de 11 bits donde entra el opcode de la función, el código 'funct3', y el bit 30 de la instrucción para diferenciar entre add y sub. También se definen varias señales de control y una señal `zero`. Además, se instancia el módulo `control`.
2. **Inicialización**: Se inicializa el archivo de volcado y se volcarán todas las variables del módulo de prueba. Luego, se aplican varios vectores de prueba a la instrucción y a la señal `zero`.

3. **Generación de la señal de reloj**: No se genera una señal de reloj en este módulo de prueba.

4. **Simulación**: Se realiza una simulación en la que se aplican diferentes vectores de prueba para diferentes tipos de instrucciones (LW, SW, ADD, SUB, AND, OR, BEQ). Después de aplicar cada vector de prueba, se espera un cierto tiempo antes de aplicar el siguiente vector de prueba.

5. **Finalización**: Una vez que se han aplicado todos los vectores de prueba, la simulación se detiene.

z	inst	RegWrite	ALUsrc	ALUop	MemWrite	MemToReg	PCsrc	MemRead
0 1	beq	0	1	resta	0	X	0 1	0
0 1	ld	1	1	suma	0	0	0	1
0 1	sd	0	1	suma	1	X	0	0
0 1	add	1	0	suma	0	1	0	0
0 1	sub	1	0	resta	0	1	0	0
0 1	and	1	0	and	0	1	0	0
0 1	or	1	0	or	0	1	0	0

Cuadro 2: Tabla de Verdad del Módulo de Control

5. Programa Implementado

Para generar el código máquina del procesador se utilizó un programa llamado RARS. El código carga un valor decimal 1 en el registro ‘x1‘ con la instrucción ‘lw x5, 32(x0)‘. Luego, realiza una serie de operaciones ‘add‘ para calcular la secuencia de Fibonacci. En cada paso, suma los dos últimos números de la secuencia para obtener el siguiente número y lo almacena en los registros ‘x1‘ y ‘x2‘ alternativamente. Después de calcular la secuencia de Fibonacci, el código almacena el último número de la secuencia en la dirección de memoria ‘0+48‘ con la instrucción ‘sw x2, 48(x0)‘. Luego, el código carga algunos valores en los registros ‘x1‘, ‘x3‘ y ‘x5‘ con las instrucciones ‘lw x1, 32(x0)‘, ‘lw x3, 40(x0)‘ y ‘lw x5, 16(x0)‘ respectivamente. Después de eso, el código realiza una serie de operaciones ‘sub‘ y ‘and‘ en el registro ‘x2‘ y almacena el resultado en el registro ‘x4‘. Luego, realiza una operación ‘or‘ en los registros ‘x3‘ y ‘x5‘ y almacena el resultado en el registro ‘x3‘. Finalmente, el código almacena el valor del registro ‘x2‘ en la dirección de memoria ‘0+40‘ con la instrucción ‘sw x2, 40(x0)‘ y luego entra en un bucle infinito con la instrucción ‘inicio: beq x0, x0 inicio‘.

6. Banco de Pruebas de Módulo Top

Para verificar el funcionamiento del procesador monociclo se creó el módulo de pruebas `Z_monocicle_tb` en el documento del mismo nombre, en este se creó una instancia del módulo *top* con las dos únicas entradas que posee el procesador monociclo (RST y CLK). Para estimular el sistema se generó una señal de reloj y se generó un reinicio del sistema, restableciendo así todos los elementos que contienen memoria. Se utilizaron los siguientes comandos para inicializar el contenido de la memoria de instrucciones y de datos:

```
$readmemh("data_initial", monocicle_prueba.data_mono.memory);
$readmemb("inst_initial", monocicle_prueba.inst_mono.memory);
```

El archivo `inst_initial` contiene el lenguaje máquina producto del ensamblado del programa descrito para el cálculo de números de Fibonacci que es cargado a la memoria de instrucciones, mientras que `data_initial` contiene los datos iniciales de la memoria de datos.

Para visualizar el contenido de los registros y las memorias se crearon *tasks* de *System Verilog* que permiten hacer la impresión de valores en la consola sin tener que utilizar el *GUI* de la herramienta *VCS*:

- **mostrar_inst_mem**(input integer b): Imprime en pantalla el contenido de las primeras “b” direcciones de la memoria de instrucciones, siendo cada datos de 32 bits.
- **mostrar_regs**(input integer b): Imprime en pantalla el contenido de las primeros “b” registros de 64 bits.
- **mostrar_data_mem**(input integer b): Imprime en pantalla el contenido de las primeras “$2^b + 1$” direcciones de memoria (esta memoria imprime dicha cantidad de direcciones debido a que se implementó para ser *byte-addresable*.

Se verificó que los registros y la memoria de datos generaran valores acorde a las instrucciones almacenadas en memoria y se imprimió el *Program Counter* en cada ciclo de reloj, de manera que en caso de que no fuera una instrucciones de salto, se pudiera visualizar de qué manera se incrementa este.

Ya que no se utilizó autoverificación, se utilizó *GTK Wave* para visualizar el archivo *.vcd* generado con los siguientes comandos:

```
$dumpfile("prueba_monocicle");
$dumpvars(3,z_monocicle_tb);
```

Para visualizar cómo afectan las instrucciones el contenido de los elementos de memoria, se utilizaron los 3 niveles de jerarquía, visualizando así los siguientes tres arreglos:

```
monocicle_prueba.data_mono.memory
monocicle_prueba.inst_mono.memory
monocicle_prueba.register_mono.registers
```

7. Anexos

El archivo de incialización de la memoria de instrucciones que contiene el programa para demostrar el funcionamiento del procesador fue ensamblado utilizando *RISC-V Assembly and Runtime Simulator (RARS)*, se puede acceder al repositorio del autor **haciendo click aquí**.

Referencias.

[1] John L. Hennessy David A. Patterson. «Computer Organization and Design RISC-V Edition». En: (2021).