ENTREGA INICIAL PROCESADOR

Projecte d'Enginyeria de Computadors

29/03/2023

Víctor Mena Doz
Rubén Juánez Jiménez

Índice

1. Instrucciones implementadas

- **MOVI, MOVHI y HALF**
  Fueron las tres instrucciones que debíamos crear en la primera etapa. Las tres han sido testeadas en todos los juegos de prueba de cada etapa y las han superado a la perfección.

- **LD, ST, LDB y STB**
  Requeridas en la etapa 2.1, tuvimos que empezar a modificar bastante nuestro código para introducirlas en nuestro procesador. Al igual que las mencionadas en el apartado anterior, se han puesto a prueba al finalizar la implementación de cada etapa y su funcionamiento es correcto.

- **Operaciones lógicas y aritméticas: AND, OR, XOR, NOT, ADD, SUB, SHA y SHL**
  En la etapa 3 se creó una ALU con un número de instrucciones considerable. Después de varias modificaciones y decisiones propias pudimos implementar estas instrucciones sin ningún problema ni complejidad. Se ha comprobado su funcionamiento en diversas ocasiones y han dado los resultados esperados.

- **ADDI**
  Mismo caso que las comentadas en el apartado anterior, ningún detalle más a resaltar.

- **Comparación con o sin signo: CMPLT, CMPLE, CMPEQ, CMPLTU y CMPLEU**
  Las comparaciones también se añadieron en la etapa 3. Tras entender el uso de las conversiones *signed()* y *unsigned()*, además de crear dos constantes *true* y *false* con los dos posibles resultados de las operaciones, la implementación fue sencilla. Utilizamos una selección de juegos de prueba para afirmar la validez de sus resultados.

- **Extensión aritmètica: MUL, MULH, MULHU, DIV, DIVU**
  Como en los últimos tres apartados, estas multiplicaciones y divisiones también fueron implementadas en la etapa 3. Fueron las más complejas debido a varias conversiones de tipo de datos que se requieren para su implementación. Después de necesitar hacer debug en varias ocasiones, llegamos a usar una implementación sencilla y correcta respecto a los resultados a obtener.

- **Saltos condicionales modo relativo al pc: BZ y BNZ**
  Su implementación se creó en la etapa 4. Tuvimos que modificar varias señales de nuestra implementación previa e incluso meter nuevas. Tras pasar varios juegos de prueba creados, hemos podido verificar su funcionamiento.

- **Ruptura de secuencia modo registro: JZ, JNZ, JMP y JALL**
  Forman parte de las modificaciones que se comentan en el apartado anterior. También las hemos puesto en duda con varios tests creados con casos límite.

2. Controlador de memoria

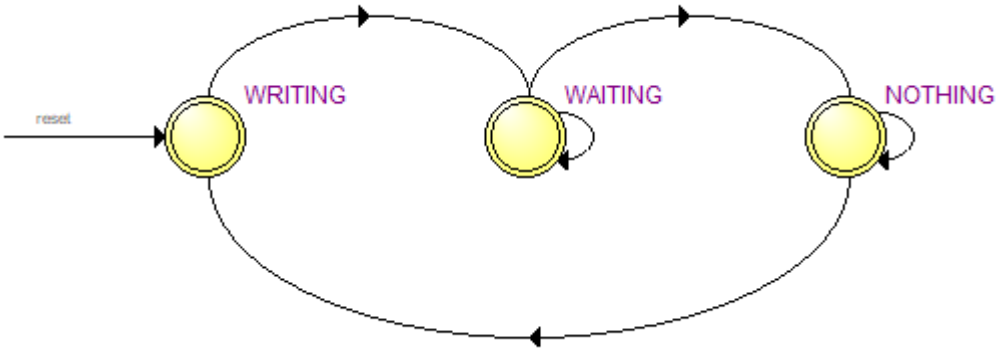
Para la creación de nuestro controlador de memoria que usamos en nuestra placa DE1 fue necesario recabar información sobre los protocolos de escritura y lectura especificados por el fabricante de la placa en cuestión. Una vez adquiridos los conocimientos requeridos, tomamos las siguientes decisiones de implementación:

- Nuestro grafo de estados estará formado por tres estados: NOTHING, WRITING y WAITING.

  En NOTHING nos encontramos cuando se realiza una lectura o no se requiere de ninguna operación de entrada o salida de la memoria.

  Para pasar a WRITING la señal WR tiene que estar a uno, en otras palabras, se debe solicitar la escritura. En este estado solo nos encontraremos un solo ciclo de memoria y saltaremos rápidamente al estado WAITING. En éste último estaremos hasta que la señal de WR vuelva a valer 0.

  Inicialmente no usábamos el estado WAITING, pero lo añadimos para así conseguir hacer la escritura de una forma mucho más rápida de tal forma que solo necesitaríamos una cuarta parte de un ciclo para realizarla. Esta ventaja será usada seguramente para la parte final del proyecto, pudiendo mejorar la velocidad de ejecución de instrucciones o escribir más seguido.



- Para hacer las escrituras se usará de media un ciclo, aunque realmente solo se requiera una cuarta parte de este y para las lecturas un ciclo también, aunque solo se requiera una octava parte. Esto es debido a que la memoria tiene un clock que va a una frecuencia 8 veces superior:

  → CLOCK PROCESADOR: 6,25 MHz

  → CLOCk MEMORIA: 50 MHz

3. Decisiones de implementación

Generalmente, nuestros archivos contienen en gran medida comentarios que especifican la utilidad de cada parte del código y en su defecto las señales auxiliares que hemos usado para nuestra implementación. De todas formas, existen diferencias entre la implementación que se nos ha pedido y lo que hemos acabado utilizando. Es por ello que en este apartado se intentan explicar todas estas modificaciones.

Para empezar, en la etapa 3 nos encontramos con muchas instrucciones que codificar y diferenciar en la ALU. Es por esto que hemos decidido crear un archivo auxiliar *const_alu.vhd* donde se hace una especificación de todos los *opcodes* que hemos requerido para diferencias las operaciones dentro de la ALU, así podemos estructurar de una forma más legible el código. En el mismo archivo se han añadido los resultados que se pueden obtener de las operaciones de comparación. En los anexos se adjunta una captura de estas constantes.

En la misma etapa que el apartado anterior, nuestra señal *z* sale de la comparación de *0* con el bus *y*. Lo hemos implementado de esta forma porque nos hace reducir el tiempo que se tarda en tomar la decisión de hacer un salto.

En la etapa 4, se requería la implementación de saltos. Para ello se usarían dos nuevas señales: *tknbranch* y *in_d*. En nuestra implementación no se han usado y en su lugar hemos creado dos funciones, implementadas en el archivo *func_ayuda_control_pkg* que son las que dictaminan si se requiere alterar la secuencia de ejecución de instrucciones y de qué forma. Debido a esto, los respectivos cálculos de las direcciones de salto se hacen en el archivo *unidad_control.vhd* justo después de saber los valores de retorno de una de las dos funciones. Además, no hemos requerido la ALU para hacer el paso de la dirección absoluta de salto que usa cada instrucción de salto, sino que directamente hemos añadido una señal *jump_dir* que viene directamente del banco de registros a la unidad de control para modificar el PC. Así el dato llega de una forma más directa y se invierte menos tiempo. En el anexo se muestra un trozo del esquema RTL donde vemos el uso de esta señal.

4. Anexos

Figura 1. Contenido archivo const_alu.vhd

```
package const_alu is

CONSTANT MOVI   : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00000";
CONSTANT MOVHI  : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00001";
CONSTANT AND_OP: STD_LOGIC_VECTOR(4 DOWNTO 0) := "00010";
CONSTANT OR_OP  : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00011";
CONSTANT XOR_OP: STD_LOGIC_VECTOR(4 DOWNTO 0) := "00100";
CONSTANT NOT_OP: STD_LOGIC_VECTOR(4 DOWNTO 0) := "00101";
CONSTANT ADD_OP: STD_LOGIC_VECTOR(4 DOWNTO 0) := "00110";
CONSTANT SUB_OP: STD_LOGIC_VECTOR(4 DOWNTO 0) := "00111";
CONSTANT SHA_OP: STD_LOGIC_VECTOR(4 DOWNTO 0) := "01000";
CONSTANT SHL_OP: STD_LOGIC_VECTOR(4 DOWNTO 0) := "01001";
CONSTANT CMPLT  : STD_LOGIC_VECTOR(4 DOWNTO 0) := "01010";
CONSTANT CMPLE  : STD_LOGIC_VECTOR(4 DOWNTO 0) := "01011";
CONSTANT CMPEQ  : STD_LOGIC_VECTOR(4 DOWNTO 0) := "01100";
CONSTANT CMPLTU: STD_LOGIC_VECTOR(4 DOWNTO 0) := "01101";
CONSTANT CMPLEU: STD_LOGIC_VECTOR(4 DOWNTO 0) := "01110";
CONSTANT ADDI   : STD_LOGIC_VECTOR(4 DOWNTO 0) := "01111";
CONSTANT MUL    : STD_LOGIC_VECTOR(4 DOWNTO 0) := "10000";
CONSTANT MULH   : STD_LOGIC_VECTOR(4 DOWNTO 0) := "10001";
CONSTANT MULHU  : STD_LOGIC_VECTOR(4 DOWNTO 0) := "10010";
CONSTANT DIV    : STD_LOGIC_VECTOR(4 DOWNTO 0) := "10011";
CONSTANT DIVU   : STD_LOGIC_VECTOR(4 DOWNTO 0) := "10100";
CONSTANT LD     : STD_LOGIC_VECTOR(4 DOWNTO 0) := "10101";
CONSTANT LDB    : STD_LOGIC_VECTOR(4 DOWNTO 0) := "10110";
CONSTANT ST     : STD_LOGIC_VECTOR(4 DOWNTO 0) := "10111";
CONSTANT STB    : STD_LOGIC_VECTOR(4 DOWNTO 0) := "11000";
CONSTANT Jxx    : STD_LOGIC_VECTOR(4 DOWNTO 0) := "11001";

CONSTANT true_a : STD_LOGIC_VECTOR(15 DOWNTO 0) := "0000000000000001";
CONSTANT false_a: STD_LOGIC_VECTOR(15 DOWNTO 0) := "0000000000000000";


end package const_alu;
```

Figura 2. Esquema RTL de la unidad de control

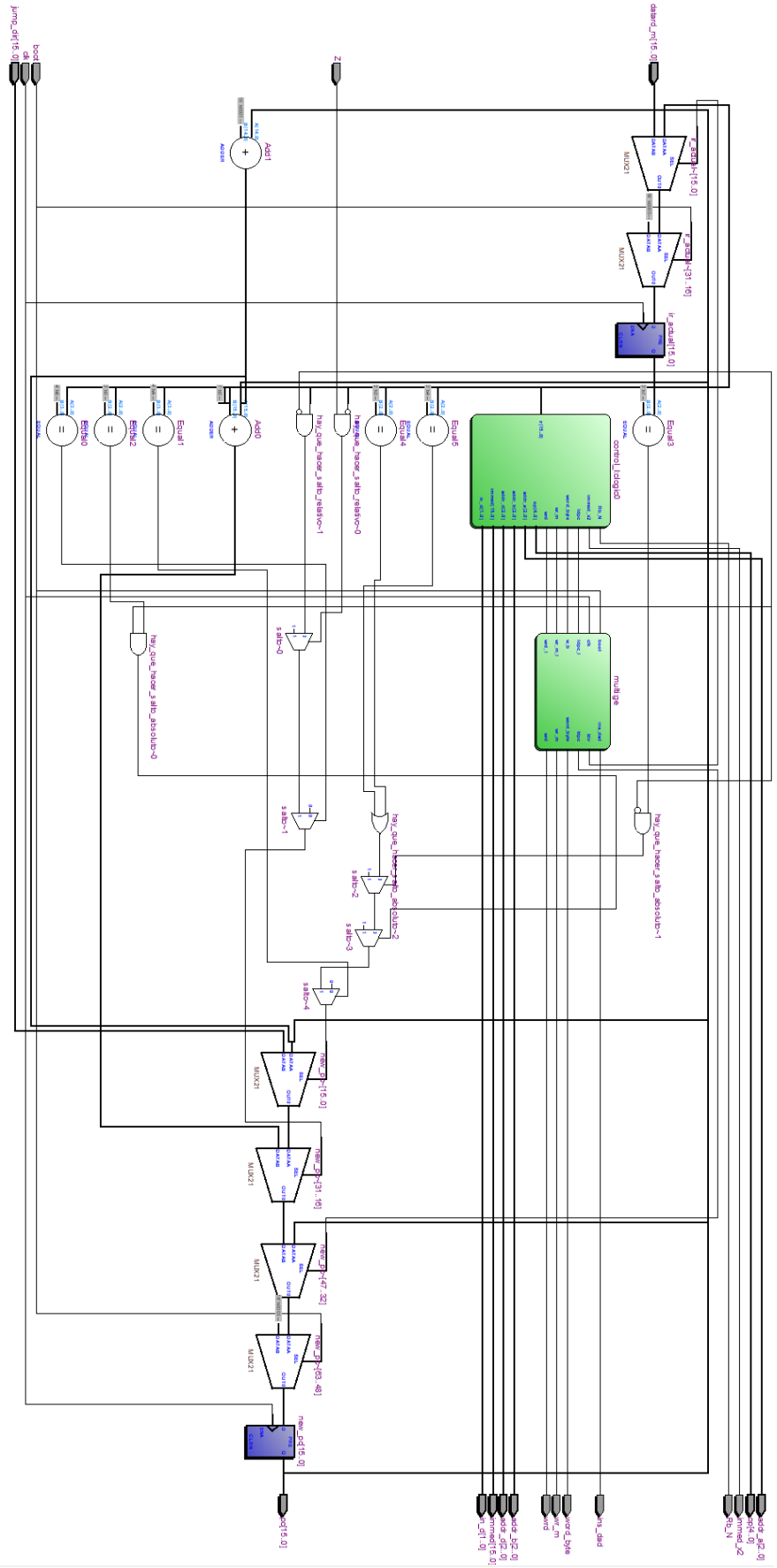


Figura 3. Fragmento del esquema RTL del datapath