**Proyecto final:** Procesador RISC con control de riesgos por dependencia de datos

Laboratorio de Organización y Arquitectura de Computadoras

**Grupo:** 2

**Alumnos:**

* Guzmán Sánchez José Emmanuel
* Mejía Ortiz Aarón Enrique
* Sáenz Barragán Ricardo

Objetivo

Diseñar el hardware de una arquitectura RISC, en VHDL, que permita el control de riesgos por dependencia de datos, implementando las unidades de detención y anticipación.

**Dispositivo**: MAX10 DE-Lite 10M50DAF484C7G

1. Desarrollo

El control de riesgos dentro de una arquitectura pipeline permite ejecutar instrucciones dependientes entre sí, de manera que no haya retrasos en la ejecución del programa y que el flujo de los datos no se vea afectado, es decir, que siempre se tomen los valores actualizados cuando existe dependencia.

La Fig. 1 muestra la arquitectura de un procesador RISC que implementa el control de riesgos por dependencia de datos y por acceso múltiple a memoria. Lo anterior se logra agregando la unidad de detención en la etapa 2, que insertará una burbuja cuando se intente acceder a la memoria para lectura y escritura al mismo tiempo, y la unidad de anticipación en la etapa 3, que adelanta el dato a la salida de esta etapa cuando se ha leído ese mismo dato en la etapa anterior.

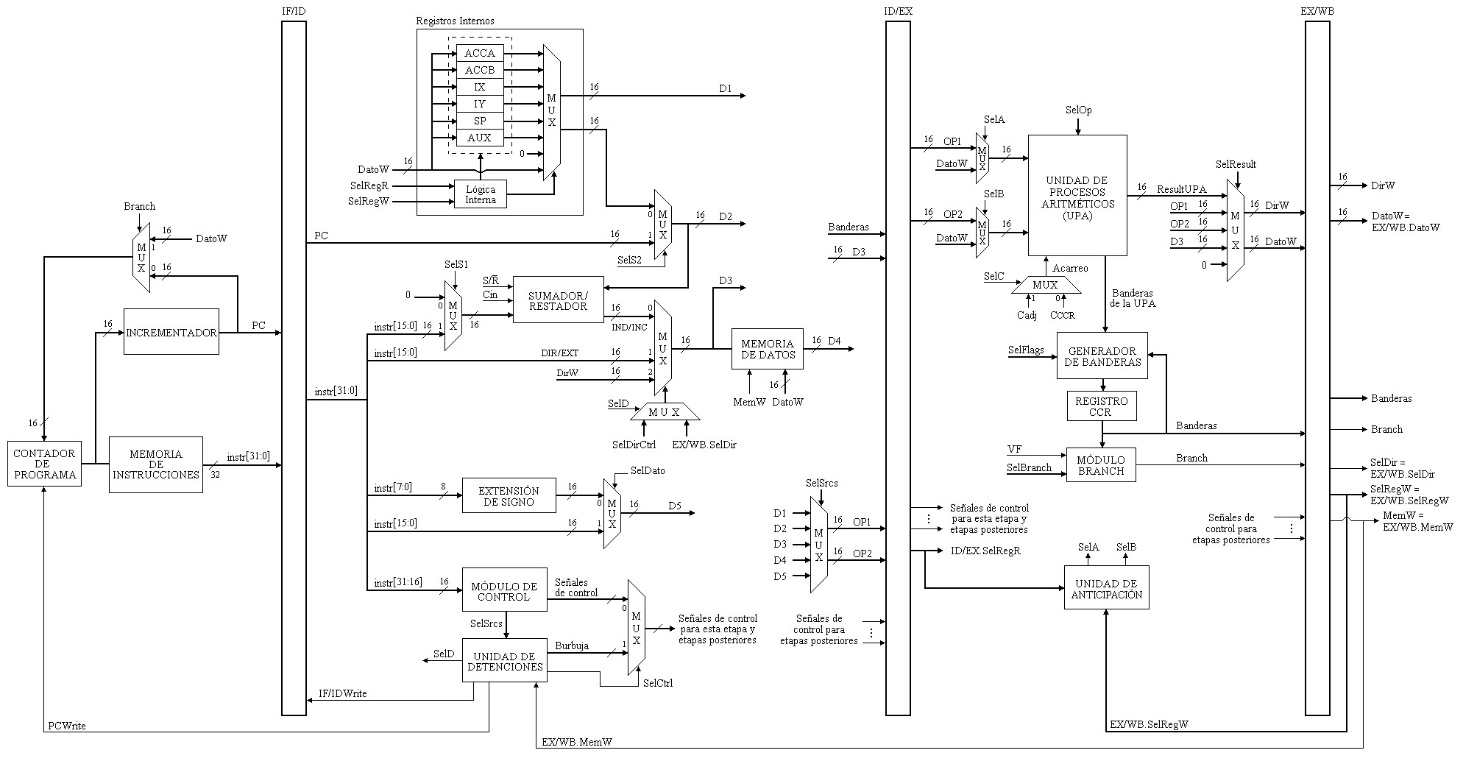


Figura 1. Diagrama del microprocesador 68HC11 en su versión pipeline con unidades de detención y anticipación

* Unidad de detención

La unidad de anticipación permite detener el flujo de la ejecución del programa cuando se intenta acceder a la memoria para lectura al mismo tiempo que para escritura. Para lograr lo anterior, esta unidad revisa el valor de las señales *SelSrc* y *memW*, ya que estas señales indican si hay un acceso múltiple a memoria, basándose en la tabla de la Fig. 2. Si se intenta acceder al mismo tiempo entonces ésta unidad insertará una burbuja en la arquitectura, es decir, en vez de mandar las señales de control provenientes de la unidad de control, manda las señales correspondientes a una instrucción *NOP* y detiene la carga en los registros *PC* e *IF/ID,* para que se mantenga la instrucción actual mientras se resuelve el conflicto de la memoria.

Una vez identificado el riesgo, entonces la unidad de detención indica de dónde tomar *SelDir* para el multiplexor que va a la memoria, por medio de la señal *SelD.* Con esto da prioridad a la escritura en memoria.

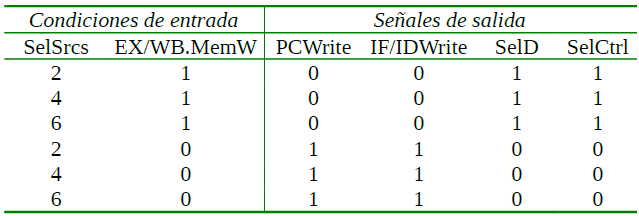
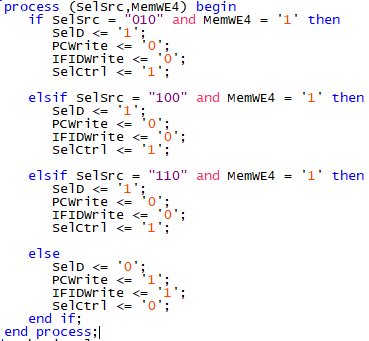
 

Figura 2. Lógica interna de la unidad de detención Figura 3. Lógica de la unidad de detención en VHDL

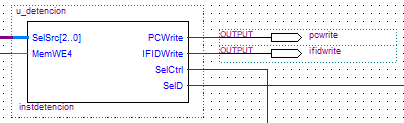
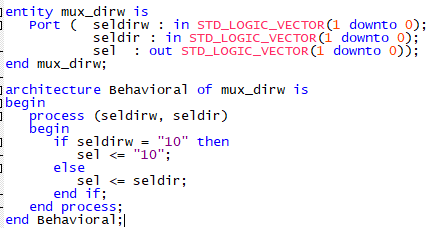
 

Figura 4. Bloque de la unidad de detención Figura 5. Multiplexor para la dirección de la memoria

Las señales *PCwrite* e *IF/IDWrite* salen de la unidad de detención de la etapa dos hacia la etapa 1, en donde se agrega un multiplexor en cada registro para indicarle si el valor a cargar será un nuevo valor o mantiene el valor que ya tiene previamente cargado

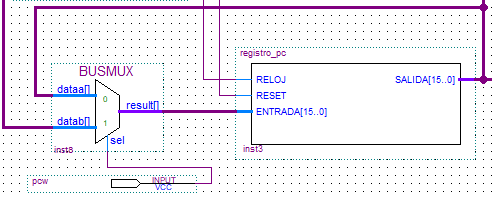


Figura 6. Modificación del registro PC para controlar la carga

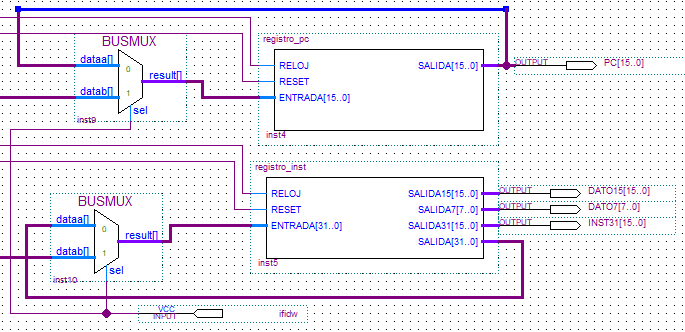


Figura 7. Modificación del registro IF/ID para controlar la carga

La señal *SelCtrl* entra a un multiplexor en la etapa 2, que controla si el valor de las señales de control para esa etapa y las etapas posteriores será de la unidad de control o una burbuja (señales de una instrucción *NOP*).

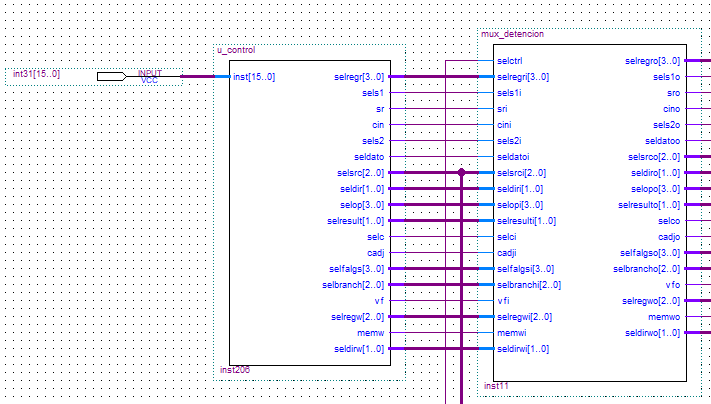
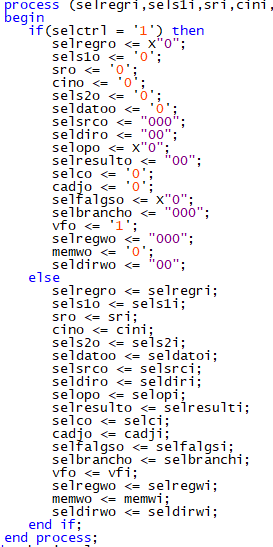
 

Figura 8. Bloque del multiplexor de la señal burbuja Figura 9. Lógica interna del multiplexor de la señal burbuja

Con lo anterior, la arquitectura es capaz de detectar acceso múltiple a la memoria de datos e insertar una burbuja en el flujo del programa para resolver el conflicto y permitir que primero se escriban los datos y después leerlos, reteniendo el valor del *PC* y del registro *IF/ID* para evitar perder el flujo del programa.

* Unidad de anticipación

La unidad de anticipación nos permite adelantar el valor de un registro cuando se hizo una lectura sobre éste, pero la instrucción previa lo modifico y aún no se ha actualizado. En estos casos, le lectura se hace sobre el registro aún no actualizado.

Para resolver este conflicto es necesario guardar el valor de *SelRegR* en el registro *ID/EX* para poder leer en la etapa 3, de esta forma, si se detecta que el registro que le leyó para la instrucción actual se va a modificar por la instrucción anterior, entonces adelanta el dato a las entras de la UPA, ya sea *OP1* u *OP2*. Lo anterior se logra comparando el valor *SelRegR* de la instrucción actual con el valor de *SelRegW* de la instrucción anterior, tal como se muestra en la Fig. 10.

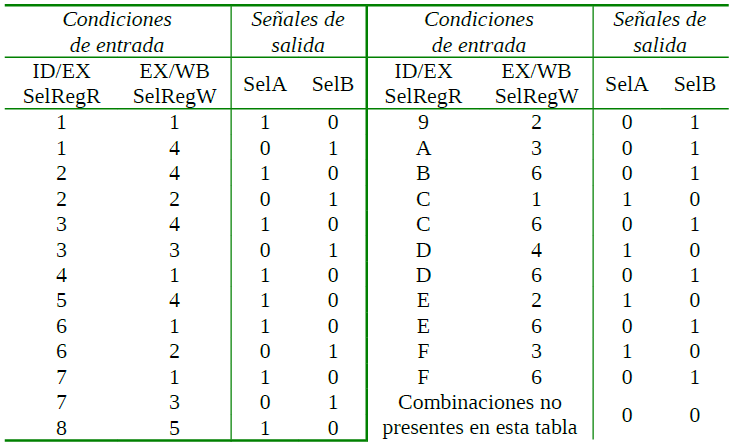


Figura 10. Lógica interna de la unidad de anticipación.

Las señales *SelA* y *SelB* permiten seleccionar, mediante multiplexores, si los valores de entrada a la UPA son las que provienen de la etapa 2 o se adelanta el valor de la señal *datoW*.

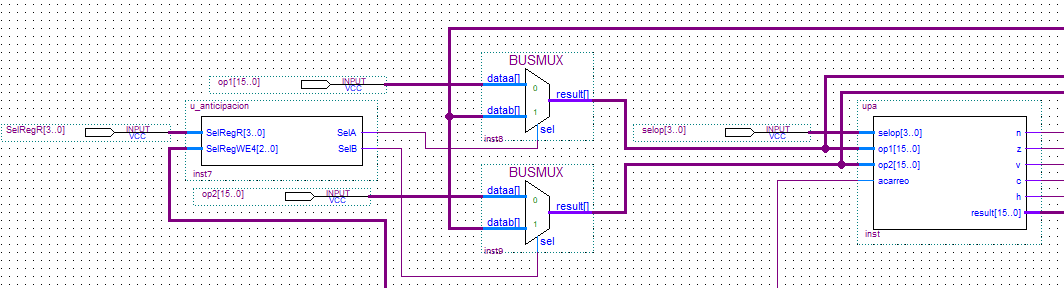


Figura 11. Multiplexores para las señales OP1 y OP2

El bloque anterior, permite que el valor de un registro se adelante antes de ser escrito, manteniendo una coherencia en el flujo de los datos y sin haber retrasos en la ejecución, ya que no se insertan burbujas.

El módulo anterior sólo nos permite anticipar el dato en la etapa 3, cuándo se va a ocupar en la UPA, sin embargo, es posible anticipar el *datoW* en la etapa 2 cuando éste se va a escribir en los registros. Con ayuda de la tabla de la figura 12, es posible saber cuándo es necesario anticipar un dato y lo manda a las salidas *D1 y D2* para poder leer los valores de los registro actualizados, en caso de que se quiere leer una registro que apenas se va a escribir.

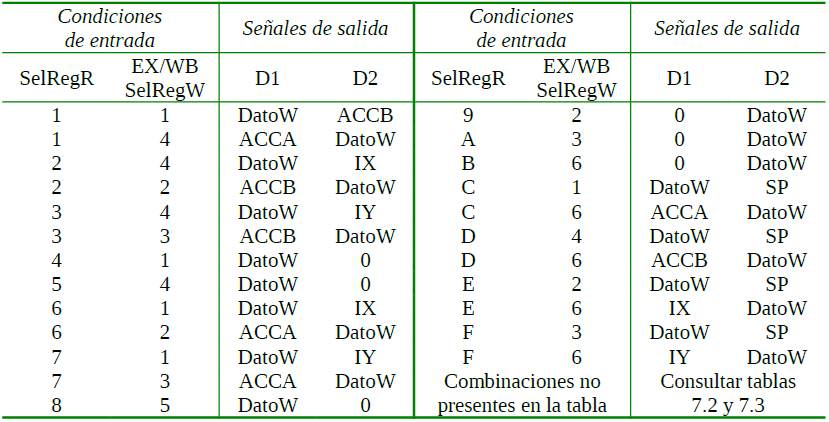
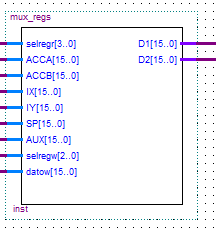
 

Figura 12. Lógica interna de los registros Figura 13. Bloque de los registros

Es importante destacar que para poder anticipar el dato que se va a escribir en los registros, es necesario agregar al multiplexor que selecciona *D1* y *D2* las señales *selRegW* y *datoW*

Una vez que se agregaron las unidades para el control por dependencia de datos, cargamos en la unidad de control las instrucciones para ejecutar el algoritmo de la suma y así poder probar el funcionamiento de nuestra arquitectura. Cabe destacar que es necesario agregar NOPs después de los saltos para que todos los registros se actualicen, ya que esta arquitectura no controla la dependencia de datos por saltos.

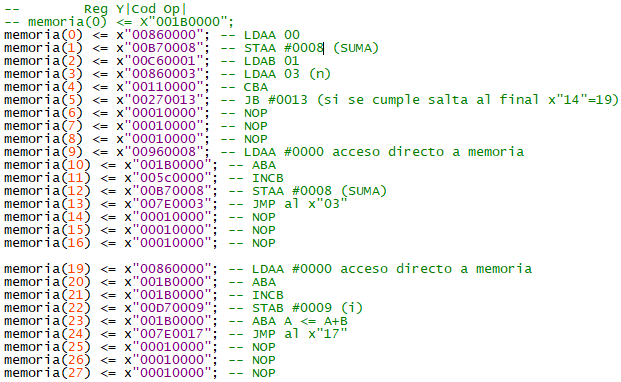


Figura 14. Instrucciones en memoria

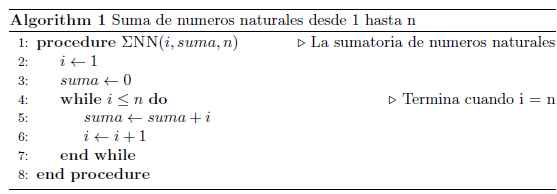


Figura 15. Algoritmo de la suma

1. Simulaciones

En la simulación es posible observar que las instrucciones se ejecutan una tras otra, es decir, no se insertan burbujas cuando existe una dependencia de datos, y los valores mantienen coherencia conforme el flujo del programa. En el acumulador B se guarda el contador *i*, y podemos apreciar que éste aumento cada determinado número de ciclos.

Al acumulador A carga la variable el valor de *n* para hacer la comparación y terminar el ciclo *while* y también carga el valor de *suma* que se guarda en la memoria. Podemos ver que ciertos ciclos en A se carga un 3, que es el valor de *n,*  y después se cargan los valores de la suma.

Finalmente en la variable suma, que es el contenido de la memoria, se puede observar cómo se va actualizando el valor conforme el flujo del programa. Al final es posible apreciar que se activan las banderas de *N,* cuando el valor de *i* sobrepasa el de *n,*  es entonces cuando se rompe el ciclo y termina la ejecución del programa.

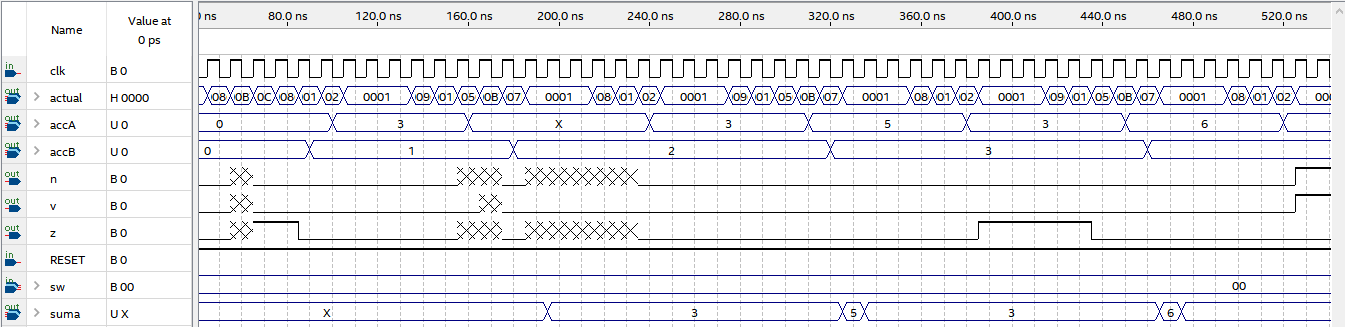


Figura 16. Simulación del programa

1. Conclusiones

Guzmán Sánchez José Emmanuel

La dependencia de datos es un factor muy importante que afectan el rendimiento y la eficiencia en una arquitectura, ya que puede causar retrasos en la ejecución de instrucciones dentro de la misma. Por esta razón, los módulos que nos permite controlar esta dependencia son de suma importancia para evitar retrasos e incongruencias cuando se ejecuta un programa. Para poder construir estos módulos fue necesario comprender cuándo se genera una dependencia de datos y en qué casos se presentan y una vez conseguido esto, saber cómo resolver los conflictos. En esta arquitectura los resolvimos mediante detenciones y anticipaciones, logrando con esto, mejor notablemente la eficiencia en la ejecución de instrucciones.

Mejia Ortiz Aarón Enrique

.

Sáenz Barragán Ricardo

.