TRABAJO PRÁCTICO FINAL

PIPELINE
PROCESADOR MIPS
SIMPLIFICADO

Alumnos:
Saul Muñoz
Cristian Velazquez
Arquitectura de Computadoras
2023

Objetivo

Implementar el pipeline del proceso Mips en una fpga, en nuestro caso una basys 3.

Implementación

Se usa en las siguientes etapas:

IF (Instruction Fetch): Búsqueda de la instrucción en la memoria de programa.
ID (Instruction Decode): Decodificación de la instrucción y lectura de registros.
EX (Excecute): Ejecución de la instrucción propiamente dicha.
MEM (Memory Access): Lectura o escritura desde/hacia la memoria de datos.
WB (Write back): Escritura de resultados en los registros.

El pipeline soporta las siguientes instrucciones:
Instrucciones tipo R
SLL, SRL, SRA, SLLV, SRLV, SRAV, ADDU, SUBU, AND, OR, XOR, NOR, SLT

Instrucciones tipo I

LB, LH, LW, LWU, LBU, LHU, SB, SH, SW, ADDI, ANDI, ORI, XORI, LUI, SLTI, BEQ, BNE, J, JAL

Instrucciones tipo J
JR, JALR

El procesador debe tener soporte para los siguientes tipos de riesgo:
Estructurales. Se producen cuando dos instrucciones tratan de utilizar el mismo recurso en el mismo ciclo.
De datos. Se intenta utilizar un dato antes de que esté preparado. Mantenimiento del orden estricto de lecturas y escrituras.
De control. Intentar tomar una decisión sobre una condición todavía no evaluada.

Esto se implementa en las siguientes unidades:
Unidad de Cortocircuitos.
Unidad de Detección de Riesgos

Otros requerimientos:

- El programa a ejecutar se carga en la memoria del programa mediante un archivo ensamblado.
- Archivo que contiene instrucciones en ensamblador.
- Transmite el ASM mediante interfaz UART antes de comenzar a ejecutar
- Tiene una unidad de Debug que envía información hacia y desde la PC mediante la UART. La información que se envía hacia PC es el contenido de los registros usados, contenido de la memoria de datos usada, PC y ciclos de reloj

Modos de operación

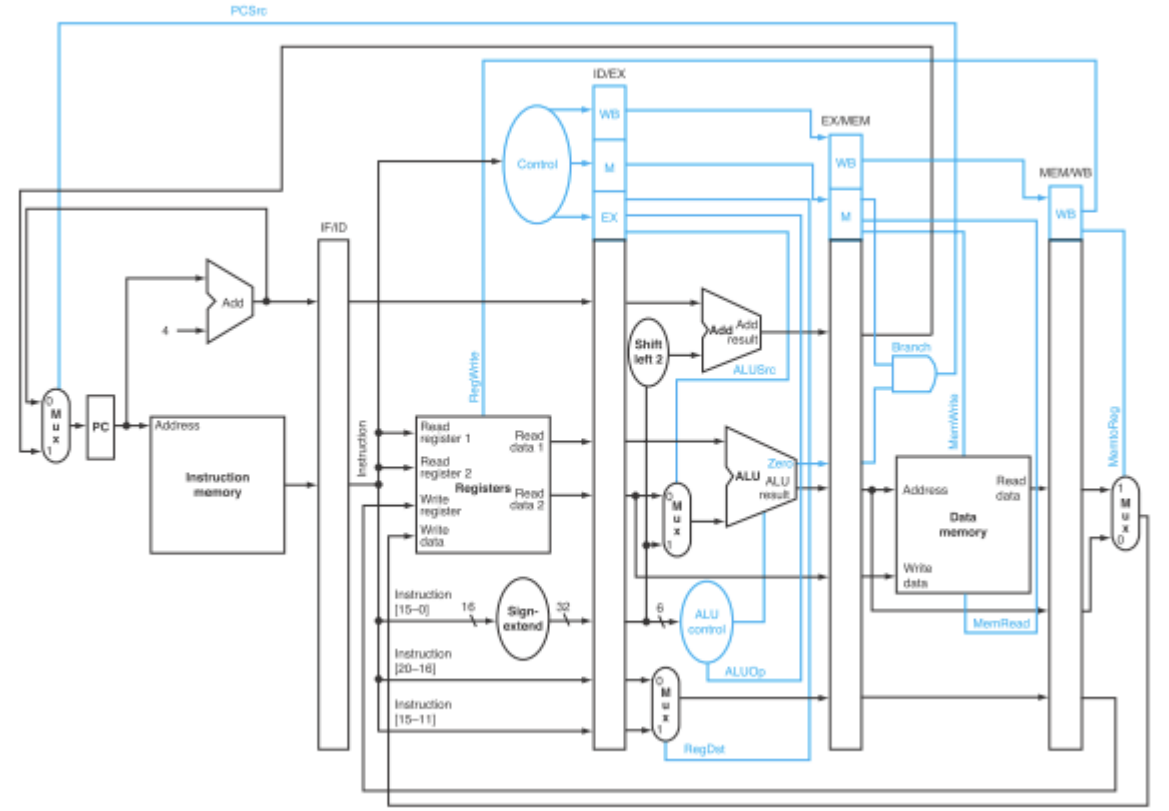
Antes de estar disponible para ejecutar, el procesador está a la espera para recibir un programa mediante la Debug Unit.

Una vez cargado el programa, debe permitir dos modos de operación:

Continuo, se envía un comando a la FPGA por la UART y esta inicia la ejecución del programa hasta llegar al final del mismo (Instrucción HALT). Llegado ese punto se muestran todos los valores indicados en pantalla.

Paso a paso: Enviando un comando por la UART se ejecuta un ciclo de Clock. Se debe mostrar a cada paso los valores indicados.

ETAPAS



IF (Instruction Fetch):

Durante la primera etapa se realiza la búsqueda de la instrucción en la dirección de la memoria de instrucciones que indica el PC y se incrementa el PC para la siguiente instrucción.

Módulos involucrados:

Memoria de Instrucciones: Contiene instrucciones que serán seleccionadas para ser ejecutadas.

Multiplexor PC: El PC se verá afectado por un incremento. Si ha habido una branch o un jump, el PC se incrementa a partir de esa instrucción.

PC: Aquí se genera una nueva dirección de instrucción, se puede hacer un incremento+4 y +8.

IF/ID: Este módulo se comporta como un latch, el cual recibe los datos de la etapa fetch y los envía a la etapa decode, cuando hay un posedge del clock. De esta forma no pierdo datos entre las etapas, los retengo en este módulo.

ID (Instruction Decode):

A continuación, se decodifica la instrucción separando sus diferentes campos. El código de operación de la instrucción indica qué tipo de instrucción es, y por tanto, qué tipo de operación se debe realizar en la ruta de datos. Si es necesario, se leen 1 o 2 operandos de los registros del banco de registro.

Módulos involucrados:

File register: Banco de registros de datos. Los 32 registros necesarios para almacenar valores se agrupan en un banco de registros porque en las instrucciones de tipo R es necesario acceder a dos registros simultáneamente. Este banco tiene por tanto dos salidas de datos de 32 bits, una entrada de datos de 32 bits y tres entradas de 5 bits para la identificación de los registros.

Extensor de signo: extiende el signo de los operandos inmediatos, que son de 16 bits, a 32 bits.

Sumador_PC_Jump: se utiliza para calcular nuevas direcciones de PC en el contexto de saltos condicionales en el procesador MIPS**.** Suma la dirección de Jump con PC+4. Así obtengo la dirección de Jump.

Control_Unidad: se reciben los 6 bits más altos de la instrucción y se generan las señales necesarias que viajarán junto con la instrucción a través del resto del pipeline para activar o desactivar en base a la instrucción decodificada.

ID_Unidad_Riesgos: se detecta cuando hay un riesgo y se agrega una burbuja.

ID_Mux_Unidad_Riesgos: Si hay una condición de riesgo se ponen a cero todas las signal del la unidad de control.

ID/EX: Módulo que recibe los datos de la etapa decode y los envía a la etapa execute , cuando hay un posedge del clock. De esta forma no pierdo datos entre las etapas, los retengo en este módulo. También envía las señales de control generadas por la Unidad de Control.

EX (Excecute):

En esta etapa se ejecuta la operación que indicaba el opcode, dependiendo de la instrucción puede ser utilizada la ALU (unidad funcional aritmético-lógica).

Principales módulos involucrados:

ALU: Una ALU capaz de realizar las operaciones incluidas en el repertorio de instrucciones.

Sumador_Branch: Suma dirección de signo extendido (x 4) con el PC+4. Así se obtiene la dirección de Branch.

Mux_ALU_Shamt: Envía el valor del operando A a la ALU dependiendo de la señal de control de la Unidad de Cortocircuito.

MEM (Memory Access):

Aca se lee o escribe un dato de la memoria de acuerdo con la instrucción ejecutada. Para determinar las ubicaciones de esta memoria que serán leídas o escritas se utiliza el resultado de la ALU de la unidad de ejecución.

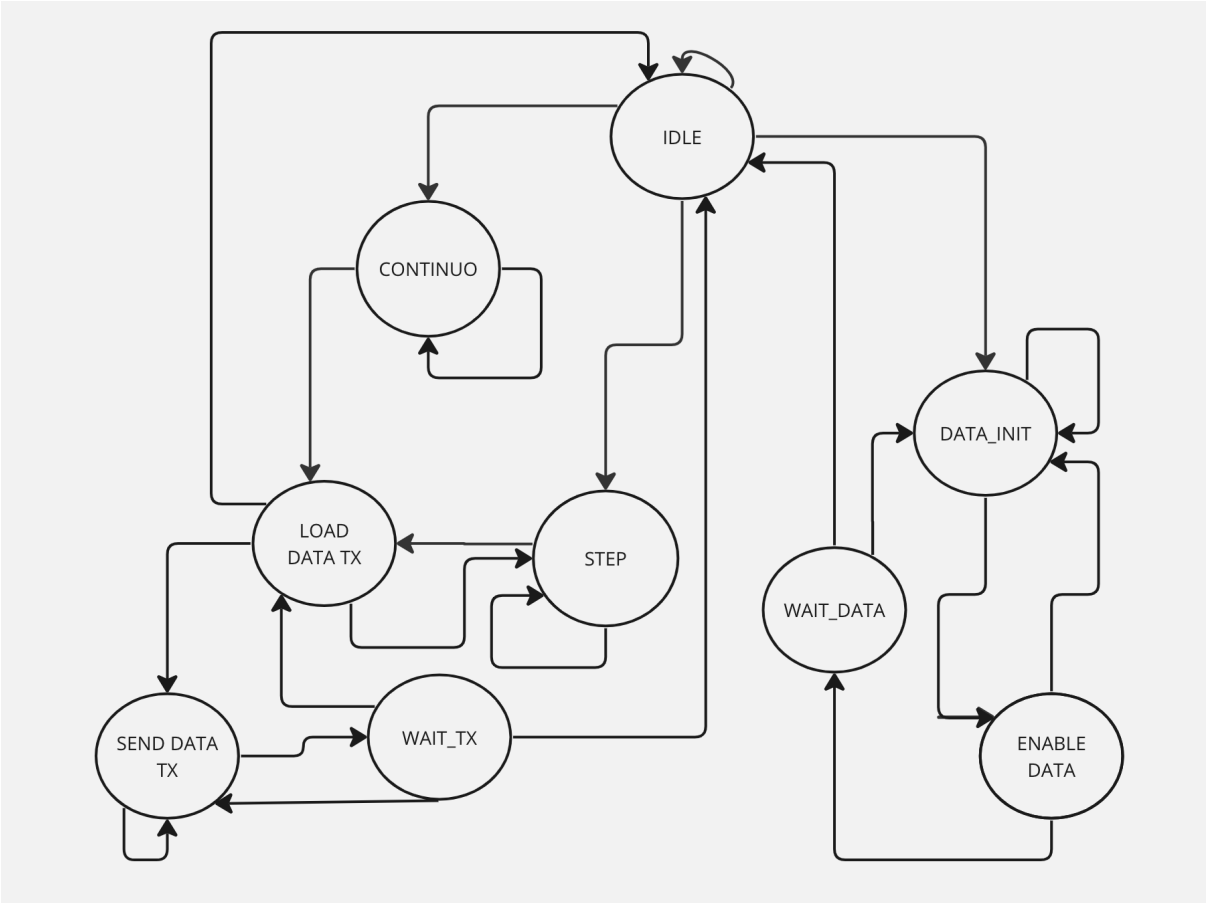
Módulos involucrados:

WB (Writeback):

Por último, si es necesario volcar algún resultado a un registro, se realiza esta escritura. Donde envía los valores nuevos de los registros al módulo Registros para proceder a actualizar los nuevos valores de los registros afectados por la instrucción que acaba de finalizar su ejecución. Por esta razón recibe el nombre de Write Back porque escriben sobre una etapa anterior del pipeline.

Módulo de debug

Este es el intermediario entre lo que envía el usuario desde el código python y el mips. Me permite centralizar toda la información para luego mandar el PC, ciclos de reloj y el contenido del register file y memory data a la computadora.
El envío de datos se hace con el módulo uart que se desarrolló en el tp2.
El módulo consta de 9 etapas, en el siguiente diagrama de estado se va a detallar cada estado. Estos estados se pueden ver en los leds de la fpga cuando se ejecuta.



IDLE: Estado inicial, desde este estado puede ir a 3 estados posibles según el carácter que se reciba por rx. Los caracteres son “c”,”s” y “d”. Estos valores se envían desde la consola por input del código python, el usuario según el modo que quiera usar va a ingresar un determinado carácter. Internamente antes de pasar a algún estado por el modo seleccionado, el primer carácter que se recibe es la “d”, ya que el estado DATA_INIT es que prepara todo para recibir las instrucciones a ejecutar.
d: Cuando se recibe este carácter se pasa al estado DATA_INIT.
s: Voy al estado STEP para realizar las ejecuciones de las instrucciones paso por paso.
c: Voy al estado CONTINUO para ejecutar todas las instrucciones.
default: Si recibo un carácter no esperado me quedo en el mismo estado.

DATA_INIT: Aquí armo las instrucción de 32 bits, como por el uart solo puedo recibir de a 8 bits. En este estado cada vez que recibo un dato de 8 bits se concatena con las demás palabras. Una vez hecho esto antes de pasar al siguiente estado, incremento un contador que permite al estado ENABLE_DATA saber si ya tengo lista la instrucción.

ENABLE_DATA: Si ya tengo completa la instrucción, habilito la escritura en la memoria de instrucciones y voy al estado WAIT_DATA. Pero si aun no está completa voy al estado DATA_INIT para seguir armando la instrucción.

WAIT_DATA: Primero pregunto si recibí un HALT, este me indica que ya termine de recibir instrucciones y voy al estado inicial. Si no recibí esta palabra aumento en 4 la dirección que se usa para guardar las instrucciones en le memoria y voy al estado DATA_INIT para armar una instrucción nueva.

CONTINUO: Este corresponde a uno de los modos de ejecución de las instrucciones. En este modo se ejecutan todas las instrucciones y cuando se recibe un halt envio el resultado final.

STEP: En este modo paso por paso, voy al estado LOAD_DATA_TX para enviar la información correspondiente a un ciclo de reloj. Para habilitar el clock y continuar se espera el ingreso de otro carácter “n” que indica el siguiente step. Para enviar información pare el clock para evitar perder información, solo lo habilito una vez que haya enviado la data correspondiente al step anterior.

LOAD_DATA_TX: Según el contador se carga la data para enviar (pc, registros y ciclos), aca se arma una variable de 32 bits (4 bytes). Osea que se envía una vez el pc, una vez los ciclos de reloj, 32 veces el register file y 16 veces el data memory. Estos últimos porque tengo definidos esa cantidad de registros para cada memoria.
Cuando terminó de enviar todos los datos vuelvo a estado inicial o en el caso que aún queden instrucciones a ejecutar porque se está usando el modo step voy al estado STEP.

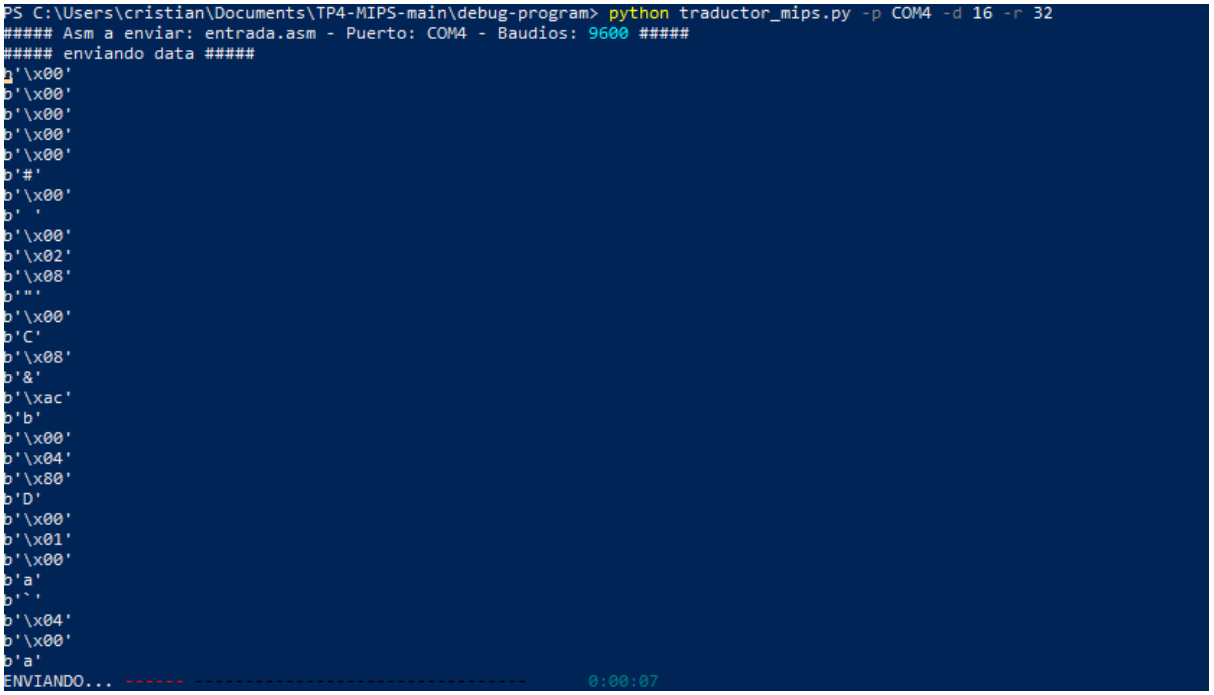
SEND_DATA_TX: Acá se envían los bits a enviar a la entrada de tx del uart. También tengo un contador, con este puedo enviar a de 8 bits, osea que separó los 32 bits en 4 partes. Cuando envió voy a estado WAIT_INIT. Si aún no termino de enviar el dato anterior sigo esperando la signal del tx para avisar que hay un dato listo para enviar.

WAIT_TX: Pregunto si se enviaron todas las palabras, ósea los 32 bits de la misma palabra. En caso que no sea se desplaza 8 bits y voy al estado SEND_DATA_TX para enviar la palabra de 8 bits que falta enviar o la que sigue.
Si ya se enviaron todas voy al estado DATA_TX para buscar la siguiente palabra de 32 bits que esta puede ser las mencionadas en el estado DATA_TX.

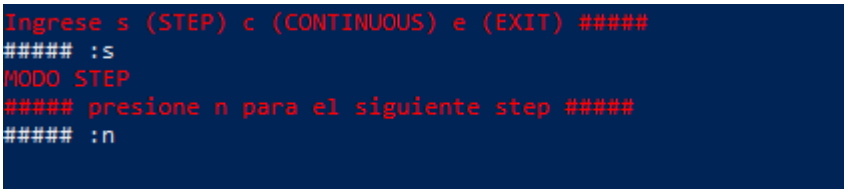
default: En el caso que hayas algún error o no se conozca a que esta ir, se vuelve al estado inicial (IDLE).

Funcionamiento de interfaz-debugger

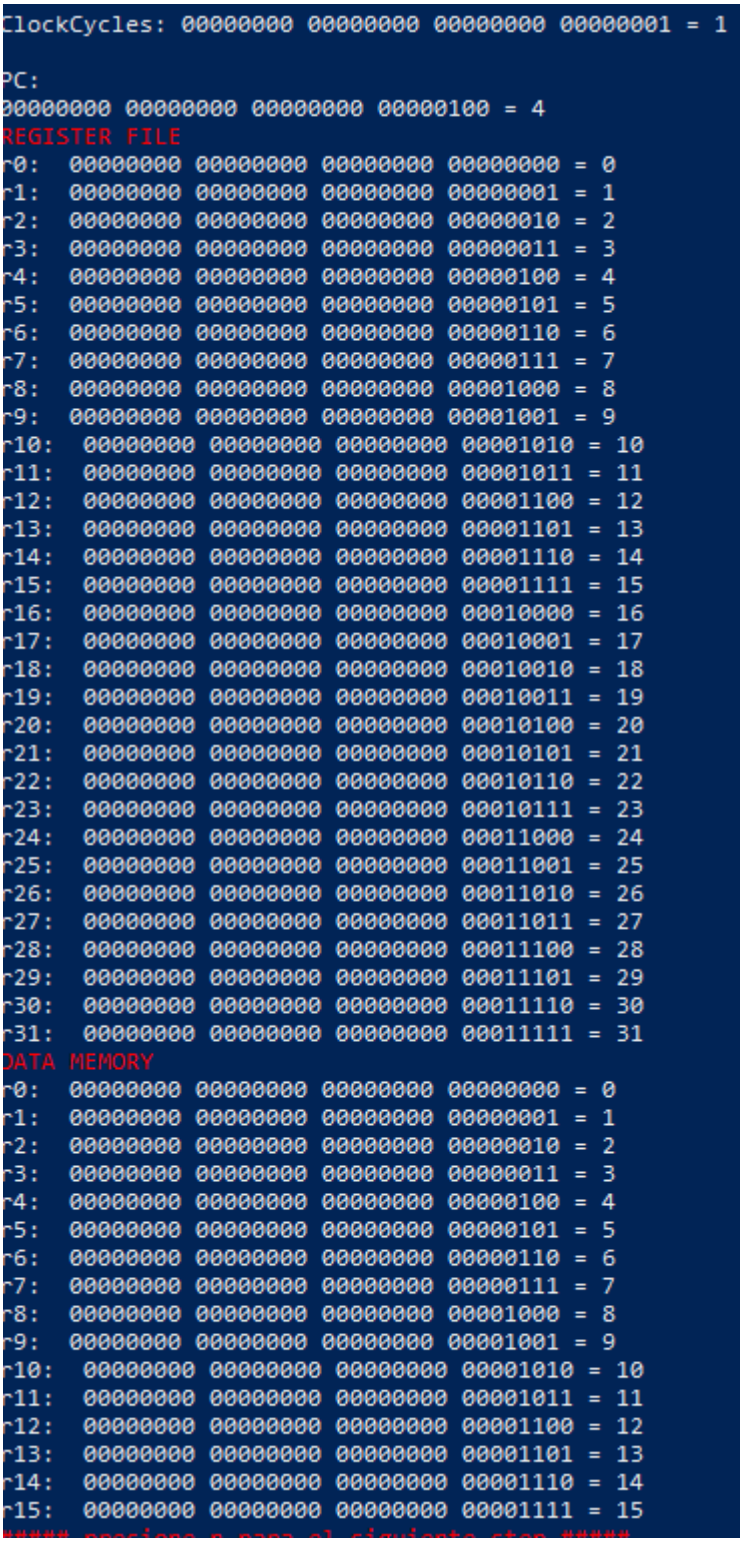
Se realizó un programa en python similar al realizado en el tp2, con la diferencia que este lee un archivo.asm donde están las instrucciones assembler a enviar. Las instrucciones se encuentran en un archivo llamada entrada.asm. Estas instrucciones
Este se puede ejecutar en linux o windows. Para el caso de linux el puerto debe ser /dev/puertoUsb y para windows es algun COM. El parámetro “d” representa la cantidad de registros de data memory que se desean ver en pantalla y “r” la cantidad de registers files que deseo ver.



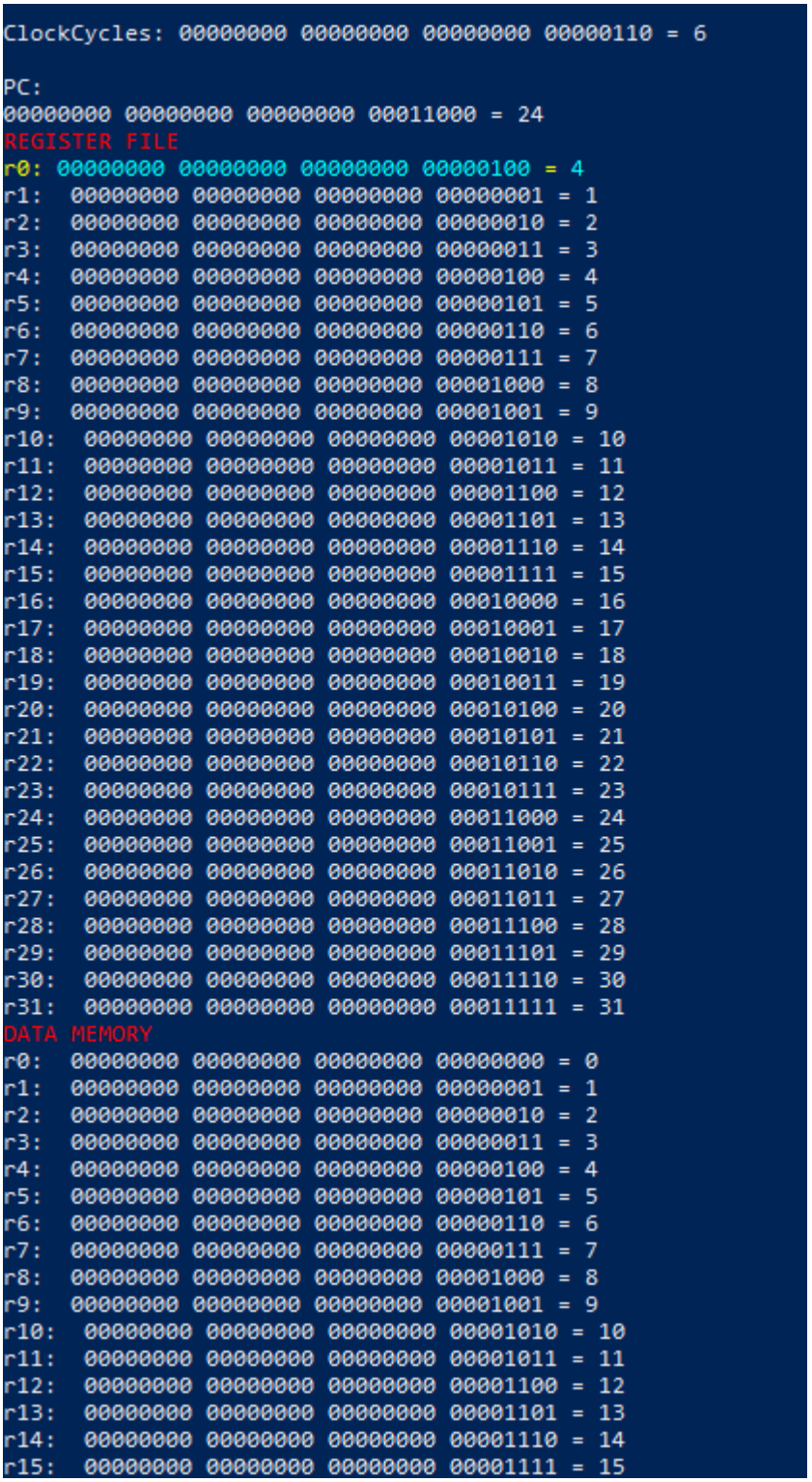
Cuando se ejecuta el programa se puede ver la impresión de los datos a enviar a la fpga. Una vez que se envían todas las instrucciones, se puede ingresar por teclado la acción a realizar. Para ejecutar paso por paso se usa la letra **s** y para seleccionar el modo continuo se usa la letra **c**. Si se quiere salir del programa se debe ingresar la letra **e**, la cual genera la salida del programa.



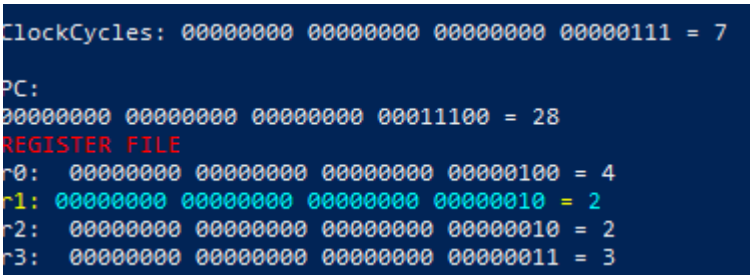
Para el primer caso se usó el modo paso por paso, con la letra n se pasa a la siguiente ciclo de reloj donde se puede ver que el pc se incrementa +4.



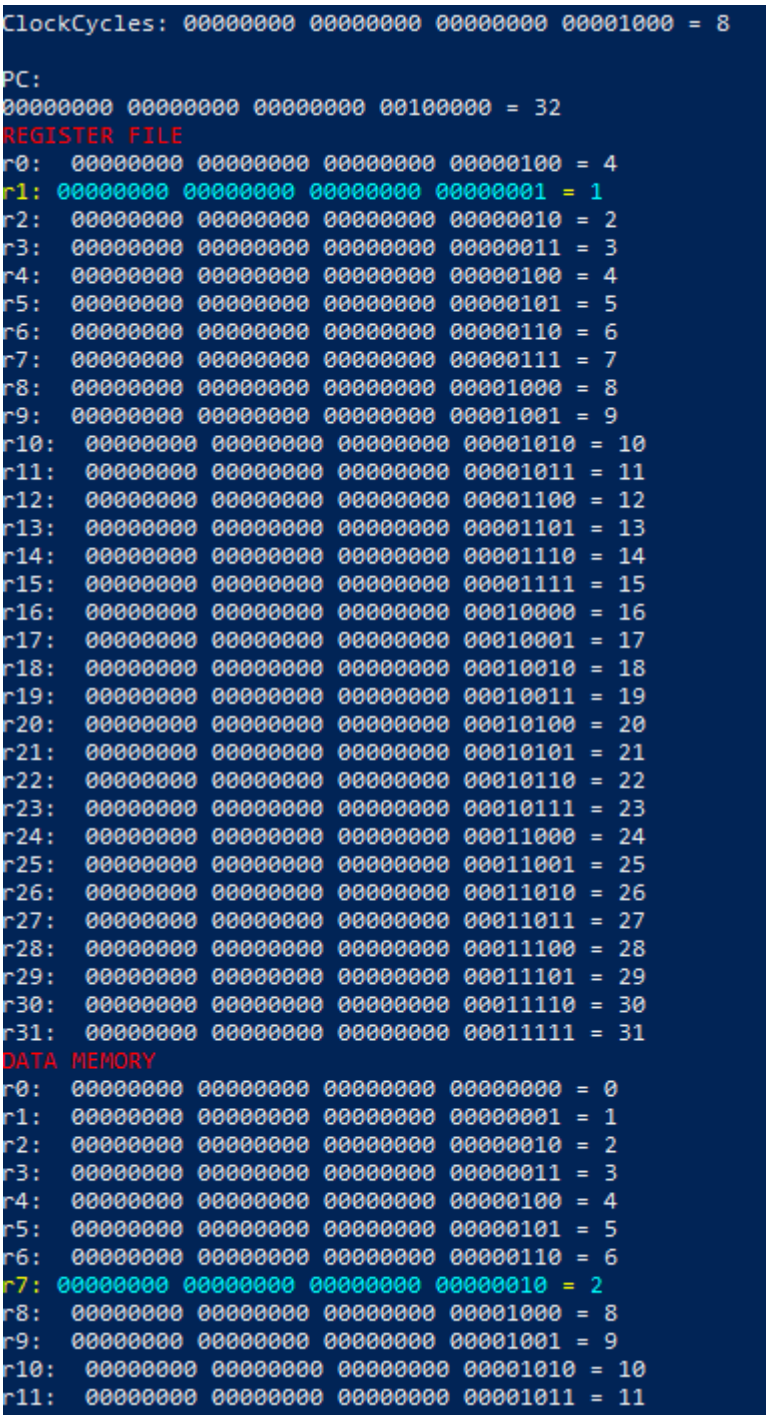
Como sabemos el clock se incrementa de a uno y el pc en 4. Como la primera operación es un nop y la siguiente instrucción es una suma que necesita de 4 etapas, el resultado recién se ve en el registro file en el pc=24, osea 6 clocks después.



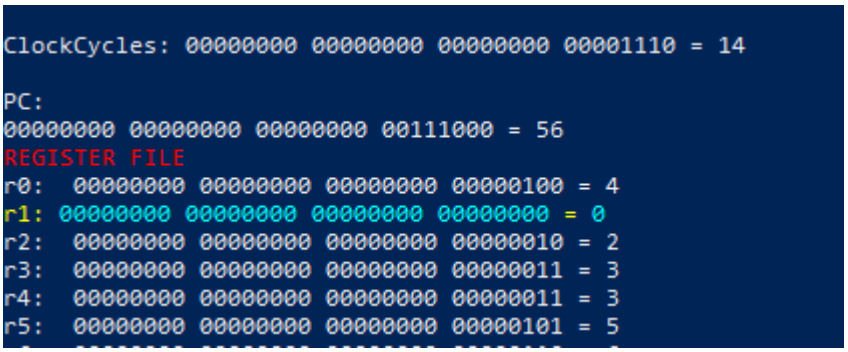
En la siguiente operación se realiza una resta (sub $1,$0,$2). Como se puede observar cada vez que se modifica un registro se pinta de color celeste con amarillo.

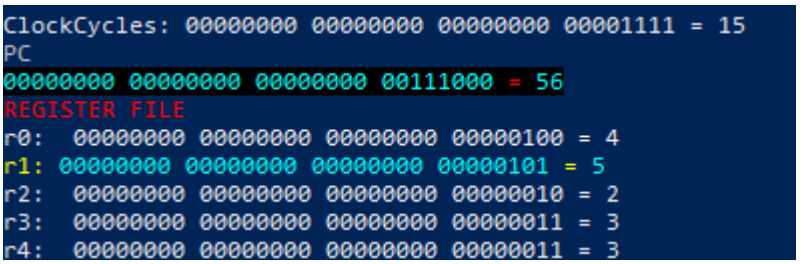


En la captura de abajo hubo una modificación en el register file y en la data memory.

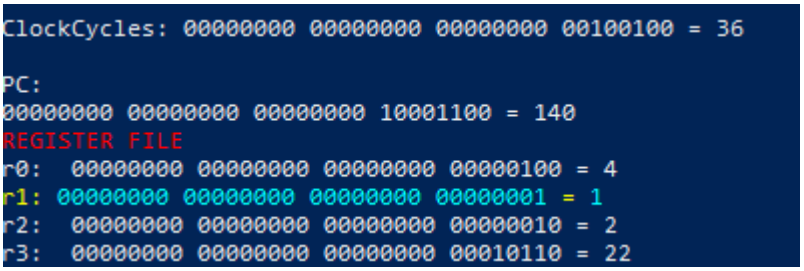


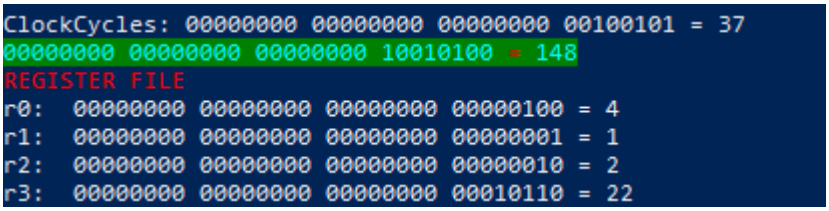
Aca se puede observar que hubo una burbuja, ya que el pc no cambio en dos clocks, cuando esto ocurre se pinta el pc con un fondo negro y simbolos rojos.





En este caso en el clock 37 hubo un salto, cuando esto pasa se lo detecta pitando con un fondo verde.





En caso para el modo continuo se ve el resultado final de la ejecución de las todas las instrucciones y la cantidad de clocks que se necesitaron.

```
##### MODO CONTINUO #####

ClockCycles: 00000000 00000000 00000000 00101011 = 43

PC: 00000000 00000000 00000000 10101100 = 172
REGISTER FILE
0:  00000000 00000000 00000000 00000100 = 4
1:  00000000 00000000 00000000 00000000 = 0
2:  00000000 00000000 00000000 00000010 = 2
3:  00000000 00000000 00000000 00010110 = 22
4:  00000000 00000000 00000000 00000000 = 0
5:  00000000 00000000 00000000 00000101 = 5
6:  00000000 00000000 00000000 00000011 = 3
7:  00000000 00000000 00000000 00000010 = 2
8:  00000000 00000000 00000000 00001000 = 8
9:  00000000 00000000 00000000 00001001 = 9
10:  00000000 00000000 00000000 00001010 = 10
11:  00000000 00000000 00000000 00001011 = 11
12:  00000000 00000000 00000000 00000000 = 0
13:  00000000 00000000 00000000 00001101 = 13
14:  00000000 00000000 00000000 00001110 = 14
15:  00000000 00000000 00000000 00001111 = 15
16:  00000000 00000000 00000000 00010000 = 16
17:  00000000 00000000 00000000 00010001 = 17
18:  00000000 00000000 00000000 00010010 = 18
19:  00000000 00000000 00000000 00010011 = 19
20:  00000000 00000000 00000000 00010100 = 20
21:  00000000 00000000 00000000 00010101 = 21
22:  00000000 00000000 00000000 00010110 = 22
23:  00000000 00000000 00000000 00010111 = 23
24:  00000000 00000000 00000000 00011000 = 24
25:  00000000 00000000 00000000 00011001 = 25
26:  00000000 00000000 00000000 00011010 = 26
27:  00000000 00000000 00000000 00011011 = 27
28:  00000000 00000000 00000000 00011100 = 28
29:  00000000 00000000 00000000 00011101 = 29
30:  00000000 00000000 00000000 00011110 = 30
31:  00000000 00000000 00000000 01110100 = 116
DATA MEMORY
0:  00000000 00000000 00000000 00000000 = 0
1:  00000000 00000000 00000000 00000001 = 1
2:  00000000 00000000 00000000 00000010 = 2
3:  00000000 00000000 00000000 00000011 = 3
4:  00000000 00000000 00000000 00000011 = 3
5:  00000000 00000000 00000000 00000101 = 5
6:  00000000 00000000 00000000 00000110 = 6
7:  00000000 00000000 00000000 00000010 = 2
8:  00000000 00000000 00000000 00001000 = 8
9:  00000000 00000000 00000000 00001001 = 9
10:  00000000 00000000 00000000 00001010 = 10
11:  00000000 00000000 00000000 00001011 = 11
12:  00000000 00000000 00000000 00001100 = 12
13:  00000000 00000000 00000000 00001101 = 13
14:  00000000 00000000 00000000 00001110 = 14
15:  00000000 00000000 00000000 00001111 = 15
```

Riesgos

Se identifican los siguiente tipos de riesgos:

Riesgos estructurales

Se producen cuando dos o más instrucciones necesitan utilizar el mismo recurso hardware al mismo tiempo. Un caso típico del riesgo estructural es el que se produce si no se separaran las memorias de instrucciones y datos, ya que diferentes instrucciones estarían intentando acceder al mismo recurso hardware al mismo tiempo (todas las instrucciones necesitan leer la instrucción de memoria en la fase IF y las de load y store necesitan leer o escribir un dato en memoria en la fase MEM).

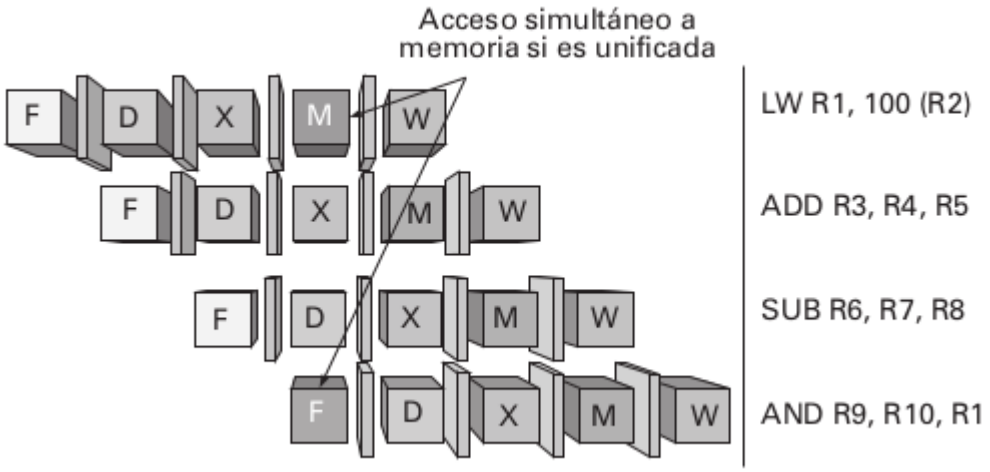


Figura. Ejemplo de riesgo estructural

Riesgos de datos

Se producen cuando dos o más instrucciones presentan dependencias de datos entre sí que, si no se resuelven correctamente, podrían llevar a la obtención de resultados erróneos en la ejecución del código por realizar operaciones de lectura y escritura en un orden diferente al indicado por la secuencia de instrucciones. Existen tres tipos de riesgos de datos, según el tipo de dependencia que los provoque:

- RAW o Read After Write (Lectura después de Escritura).
- WAR o Write After Read (Escritura después de Lectura).
- WAW o Write After Write (Escritura después de Escritura).

Las dependencias RAW son las reales, ya que las WAW y las WAR, aunque pueden provocar riesgos, se deben a la reutilización de los registros visibles para el programador, pero no existe un flujo real de información entre las instrucciones que provocan la dependencia.
En el caso del MIPS sólo pueden producirse riesgos RAW, ya que al pasar todas las

instrucciones por el mismo número de etapas y terminar en orden, las dependencias WAR y WAW nunca provocan riesgos. La dependencia RAW entre LW y ADD sí que es un riesgo, y si no se resuelve, el resultado de la ejecución sería erróneo, ya que uno de los operandos de la suma no tiene el valor correcto. Pero las dependencias por R3 (WAW) y por R5 (WAR), no suponen ningún riesgo ya que las lecturas y escrituras siempre quedarán en el orden adecuado.

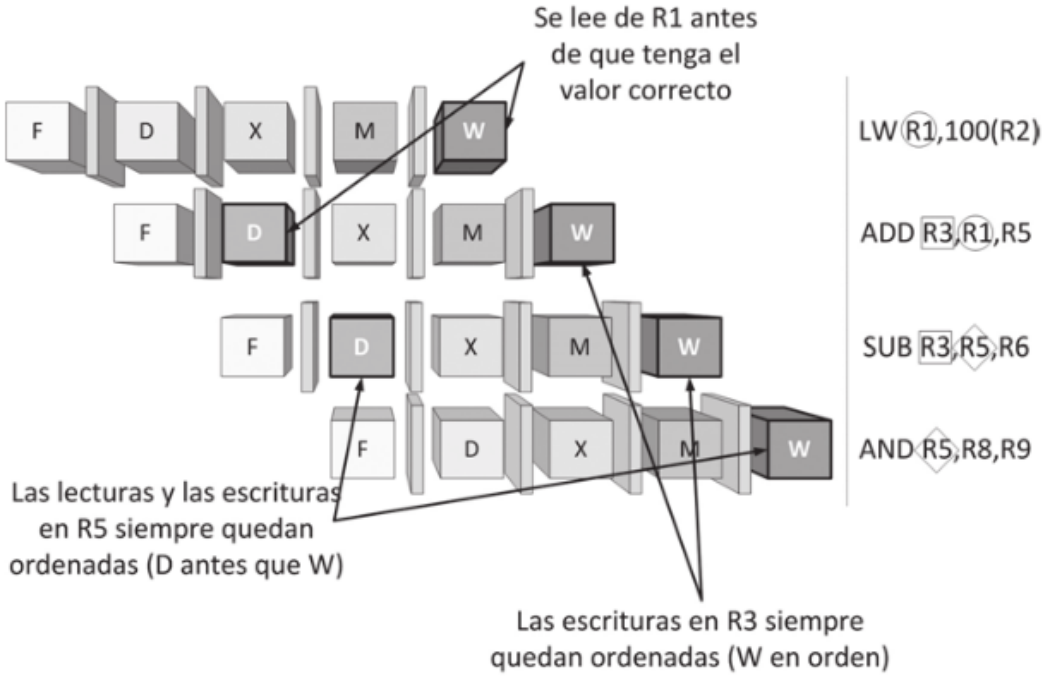


Figura. Ejemplo de dependencias y riesgos de datos

Riesgos de control

Se producen cuando una instrucción que modifica el valor del PC todavía no lo ha hecho cuando se tiene que comenzar la ejecución de la siguiente instrucción. En el caso del MIPS, la instrucción que provoca este tipo de riesgo es el BEQ, ya que hasta la fase MEM no carga el valor adecuado para el PC, y en los tres ciclos que tarda en llegar a esta fase el procesador no puede comenzar la ejecución de ninguna instrucción.

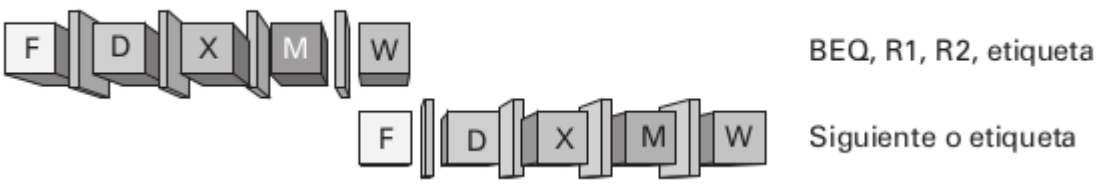


Figura. Ejemplo de riesgo de control

Para resolver estos riesgos, la introducción de paradas (penalizaciones de ciclo) en la ruta de datos. Con esta técnica se asegura la obtención de resultados correctos y se evitan los conflictos por recursos y por incertidumbres en cuanto al valor del PC. Es la unidad de control la encargada de realizar la detección de riesgos en la etapa D y de inhibir el avance de las instrucciones por la ruta de datos hasta que estos queden resueltos.

Resolución de riesgos estructurales

Para evitar los riesgos estructurales la memoria de datos y el banco de registros se encuentran separados y se realizan turnos para leer y escribir mediante el selector de operaciones.

Resolución de riesgos de datos

La técnica se denomina adelantamiento o cortocircuito y consiste en pasar directamente el resultado obtenido con una instrucción a las instrucciones que lo necesitan como operando. Es decir, estas instrucciones no tienen que esperar a que la que produce el resultado lo escriba en el registro destino, sino que reciben su valor en cuanto está disponible.

lo único que hay que hacer es identificar todos los posibles adelantamientos necesarios para el repertorio que ejecuta el procesador y comunicar los registros de segmentación involucrados.

tenemos las siguientes posibilidades para riesgos RAW:

LW seguido de SW: El operando está disponible a la salida de la etapa M de la instrucción LW y se necesita a la entrada de la etapa M de la instrucción SW.

LW seguido de aritmético-lógica o de BEQ: El operando está disponible a la salida de la etapa M de la instrucción LW y se necesita a la entrada de la etapa X de la instrucción aritmético-lógica o BEQ.

Aritmético-lógica seguida de SW: El operando está disponible a la salida de la etapa X de la instrucción aritmético-lógica y se necesita a la entrada de la etapa M de la instrucción SW.

Aritmético-lógica seguida de aritmético-lógica o BEQ: El operando está disponible a la salida de la etapa X de la instrucción aritmético-lógica y se necesita a la entrada de la etapa X de la instrucción BEQ.

LW seguido de acceso a memoria: El operando está disponible a la salida de la etapa M de la instrucción LW y se necesita a la entrada de la etapa X de la instrucción de acceso a memoria para calcular la dirección.

Aritmético-lógica seguida de acceso a memoria: El operando está a la salida de la etapa X de la instrucción aritmético lógica y se necesita a la entrada de la etapa X de la instrucción de acceso a memoria para calcular la dirección.

todas las instrucciones se comienzan y se terminan en orden, pasando todas ellas por el mismo número de etapas

El origen de un adelantamiento siempre estará en el registro X/M o en el registro M/W y el destino de un adelantamiento siempre estará en el registro D/X o en el X/M. Siempre se procurará que el operando se adelante justo a la entrada de la etapa que lo necesita.

la lógica de adelantamiento está compuesta por nuevos multiplexores para controlar las entradas de las etapas que pueden necesitar un adelantamiento, y unidades hardware sencillas que permitan comunicar las salidas de las etapas que producen los resultados con las entradas de estos multiplexores.

un nuevo control en la decodificación de la instrucción en la etapa D, detecta los adelantamientos y las paradas que van a ser necesarias para la ejecución de la instrucción. Ya que se observa que en algunos casos el adelantamiento no es suficiente para evitar las paradas en la ruta de datos, es típico el ejemplo de la combinación load y aritmético-lógica, que siempre obliga a realizar un ciclo de parada para resolver el riesgo.

para poder adelantar operandos a la etapa X los dos nuevos multiplexores a la entrada de la etapa y la unidad de adelantamiento en la propia etapa X que controla estos multiplexores, además de la unidad de detección de adelantamientos y paradas que se incluye en la unidad de control en la etapa D. El resto de la ruta de datos no se modifica, al igual que las señales de control.

Las mismas modificaciones se deberían realizar para adelantar operandos a la etapa M (a las instrucciones de store).

Resolución de riesgos de control

Como es en la etapa D en la que se decodifica la instrucción y se sabe que es un salto, en esta misma etapa se añade lo necesario para evaluar la condición del salto (un restador, el módulo Zero) y para calcular la dirección destino del salto (un sumador). La señal de control Branch pasa a generarse directamente para ser utilizada en la etapa D. Además, si se realiza esta modificación en la ruta de datos, aparece un nuevo adelantamiento a la etapa D (cuando hay un riesgo de datos RAW con uno de los registros necesarios para la evaluación de la condición del salto), lo que exige añadir el hardware necesario para realizar los adelantamientos también en esta etapa si se utiliza esta técnica para resolver los riesgos de datos.

Con la ruta de datos modificada del MIPS, sólo es necesario un ciclo de parada para resolver el riesgo de control.

Aún con esta modificación, cada vez que se ejecuta una instrucción de salto, el procesador tiene que parar un ciclo, y las instrucciones de salto son muy frecuentes, por lo que estas paradas empeoran significativamente el rendimiento de la segmentación.

La técnica hardware que intenta evitar esta parada se basa en realizar una predicción para

el salto. Las predicciones siempre consiguen una reducción de la penalización por salto cuando aciertan. Las predicciones más sencillas son las estáticas, en las que el procesador está diseñado para predecir en todos los casos que el salto se va a tomar o que el salto no se va a tomar. Es decir, la predicción es siempre la misma para cualquier salto.

Con la predicción de salto no tomado en el MIPS se consigue que la penalización sea de un ciclo para saltos tomados y de ningún ciclo para saltos que no se toman. Mientras se decodifica y se hace efectivo el salto, se va buscando la siguiente instrucción. Si finalmente el salto no se toma y la predicción se ha acertado, se continúa normalmente y no se ha perdido ningún ciclo. Si por el contrario el salto se toma y la predicción falla, se busca la instrucción destino y se ha perdido un ciclo haciendo un trabajo que no era necesario, pero que se hubiera perdido igualmente si no se hubiera realizado la predicción.

Es decir, gracias a la predicción de salto no tomado sólo se tiene penalización por salto cuando los saltos se toman (la predicción falla).

La implementación de esta técnica es sencilla, en la ruta de datos no hay que hacer prácticamente ninguna modificación y la unidad de control sólo tiene que añadir la lógica necesaria para continuar con el camino de salto no tomado si la predicción acierta, o para desechar la instrucción que se había buscado y comenzar el camino de salto tomado, buscando la instrucción destino de salto, si la predicción falla. Obviamente, si no se incorporan en la ruta de datos mecanismos para solucionarlo, no se debe dejar que las instrucciones predichas modifiquen el estado del procesador y/o la memoria, es decir, se debe parar su ejecución antes de que lleguen a las etapas en las que se escribe en memoria y en el banco de registros. Pero en el MIPS esto no supone ningún problema porque si el salto se resuelve durante la etapa D, la instrucción predicha sólo realiza la etapa F. Incluso si la ruta de datos no se modifica y el salto se resuelve en la etapa M, la instrucción predicha que más avanza sólo llega a la etapa X, por lo que no escribe en memoria o en registros.

En la arquitectura del MIPS y en otras similares no tiene sentido utilizar la predicción de salto tomado porque supone una penalización de un ciclo en todos los casos, independientemente de si se acierta la predicción o no. Esto se debe a que no se conoce la dirección destino de salto hasta la etapa D, por lo que no se puede buscar la instrucción que se predice que se va a ejecutar hasta que el salto no se resuelve realmente.

Para las instrucciones de salto condicional la dirección destino de salto se calculan en la etapa D y evalúan la condición y cargan el nuevo PC en la etapa X. Vamos a comparar el rendimiento de la predicción estática de salto no tomado con la de salto tomado

En el caso de la predicción estática, si ésta es de salto no tomado, siempre se buscan las instrucciones a continuación de la de salto hasta que el salto se resuelve:

Y como es un salto no tomado, cuando se resuelve el salto, simplemente se continúa con la

ejecución de las instrucciones que se habían buscado después del salto.

En el caso de que sea un salto que se toma:

Y la penalización es de 2 ciclos, la misma que se hubiera producido sin realizar la predicción con motivo del riesgo de control.

Si se hace predicción de salto tomado, tenemos que buscar siempre a continuación de la instrucción de salto la instrucción destino de salto y las siguientes. Pero para comenzar estas búsquedas, es necesario conocer la dirección destino de salto, que en este caso se calcula en la etapa D:

Es decir, hay una penalización de 2 ciclos en los saltos no tomados porque se falla la predicción y es como si no se hubiera hecho, y de 1 ciclo en el caso de los saltos tomados (acierto de la predicción) porque se tarda un ciclo en tener disponible la dirección destino de salto para poder hacer esta predicción.

En resumen:

Tipo de predicción	Penalización en acierto	Penalización en fallo
Salto no tomado	0	2
Salto tomado	1	2

Supongamos que en media, con los códigos ejecutados en este procesador el 75% de los saltos se toman y el 25% de los saltos no. Entonces, para la predicción de salto no tomado tenemos una penalización media por riesgo de control de $0.75 \cdot 2 = 1.5$ ciclos. Sin embargo, utilizando la predicción de salto tomado la penalización media es de $0.75 \cdot 1 + 0.25 \cdot 2 = 1.25$ ciclos, por lo que se consiguen mejores resultados con este tipo de predicción aunque haya 1 ciclo de penalización en los aciertos.

ANEXO

Prueba de instrucción halt

Se solicitó probar el cambio para que el halt no solo se envíe al final de las instrucciones sino que se lo lea del file de instrucciones assembler y a continuación de este haya otra instrucción como se puede observar en la siguiente captura:

```
hop
add $0,$1,$3
sub $1,$0,$2
halt
add $4,$7,$3
halt
```

Para poder realizar esta prueba en el file armadoinstruc.py se comento para que no se envíe como última instrucción el halt y se agregó una condición para que cuando se reciba el halt se envíen 32 bits. El problema de esto es que hay que cambiar toda la lógica para que tome el halt porque nosotros pensamos en la interacción del usuario con el programa y no consideramos al halt como una instrucción ya que no está la lista de instrucciones a implementar sino que se menciona en los modos de operación como una instrucción que está al final del programa por eso se pensó de esta manera.
Entonces para probar se modificó el código python para que no haya una interacción y supusimos que se usa el modo continuo. De esta forma se envían los caracteres necesarios para que el debug pueda ir cambiando de estado. Porque de lado del mips, se espera un carácter de inicialización el cual se usa para cargar las instrucciones en el memoria de instrucciones y luego se espera un carácter para indicar el modo de ejecución. Como se recibe un halt seguido de otra instrucción, en el módulo debug el halt hace que se pase al estado inicial pero aquí se espera el carácter para cargar instrucciones, por esa razón esto se modificó para que después se envíe una d. Así puedo ir a buscar la nueva instrucción y después al enviar otro carácter “c”, pasó al estado continuo para mostrar en pantalla los resultados correspondientes a la última instrucción..
Observamos que de esta forma no toma la última instrucción, esto se puede mejorar si modifica en el debug el estado de la recepción de la instrucción después de recibir un halt seguido de una instrucción para que se pase a la etapa donde se escribe la instrucción en el registro de instrucciones y limpiar lo anterior para evitar tener residuos. Otra posible mejora sería que siempre esté a la espera de instrucciones y que las instrucciones que se envían después del halt se guarde a continuación de la última recibida porque cuando hay un halt se comienza a escribir desde la posición 0 del registro de instrucciones.

Código binario de las instrucciones :

```
00000000000000000000000000000000
00000000001000110000000000100000
00000000000000010000010000100010
11111111111111111111111111111111
00000000111000110010000000100000
11111111111111111111111111111111
```

Instrucciones divididas en partes de 8 bits:

```
##### enviando data #####
1
b'\x00'
2
b'\x00'
3
b'\x00'
4
b'\x00'
5
b'\x00'
6
b'#'
7
b'\x00'
8
b' '
9
b'\x00'
10
b'\x02'
11
b'\x08'
12
b''
13
b'\xff'
14
b'\xff'
15
b'\xff'
16
b'\xff'
17
b'\x00'
18
b'\xe3'
19
b' '
20
b' '
21
b'\xff'
22
b'\xff'
23
b'\xff'
24
b'\xff'
```

Ejecución del programa:

```
ClockCycles: 00000000 00000000 00000000 00001000 = 8

PC: 00000000 00000000 00000000 00011100 = 28
REGISTER FILE
0:  00000000 00000000 00000000 00000100 = 4
1:  00000000 00000000 00000000 00000010 = 2
2:  00000000 00000000 00000000 00000010 = 2
3:  00000000 00000000 00000000 00000011 = 3
4:  00000000 00000000 00000000 00000100 = 4
5:  00000000 00000000 00000000 00000101 = 5
6:  00000000 00000000 00000000 00000110 = 6
7:  00000000 00000000 00000000 00000111 = 7
8:  00000000 00000000 00000000 00001000 = 8
9:  00000000 00000000 00000000 00001001 = 9
10:  00000000 00000000 00000000 00001010 = 10
11:  00000000 00000000 00000000 00001011 = 11
12:  00000000 00000000 00000000 00001100 = 12
13:  00000000 00000000 00000000 00001101 = 13
14:  00000000 00000000 00000000 00001110 = 14
15:  00000000 00000000 00000000 00001111 = 15
16:  00000000 00000000 00000000 00010000 = 16
17:  00000000 00000000 00000000 00010001 = 17
18:  00000000 00000000 00000000 00010010 = 18
19:  00000000 00000000 00000000 00010011 = 19
20:  00000000 00000000 00000000 00010100 = 20
21:  00000000 00000000 00000000 00010101 = 21
22:  00000000 00000000 00000000 00010110 = 22
23:  00000000 00000000 00000000 00010111 = 23
24:  00000000 00000000 00000000 00011000 = 24
25:  00000000 00000000 00000000 00011001 = 25
26:  00000000 00000000 00000000 00011010 = 26
27:  00000000 00000000 00000000 00011011 = 27
28:  00000000 00000000 00000000 00011100 = 28
29:  00000000 00000000 00000000 00011101 = 29
30:  00000000 00000000 00000000 00011110 = 30
31:  00000000 00000000 00000000 00011111 = 31
DATA MEMORY
0:  00000000 00000000 00000000 00000000 = 0
1:  00000000 00000000 00000000 00000001 = 1
2:  00000000 00000000 00000000 00000010 = 2
3:  00000000 00000000 00000000 00000011 = 3
4:  00000000 00000000 00000000 00000100 = 4
5:  00000000 00000000 00000000 00000101 = 5
6:  00000000 00000000 00000000 00000110 = 6
7:  00000000 00000000 00000000 00000111 = 7
8:  00000000 00000000 00000000 00001000 = 8
9:  00000000 00000000 00000000 00001001 = 9
```

Como se puede observar en la captura se ejecutaron las dos primeras instrucciones pero la última instrucción que está después del halt no se ejecutó.

Determinación de la frecuencia máxima de funcionamiento

La frecuencia de trabajo es 50Mhz, y se va determinar cuál es la máxima.

Frecuencias:

70 Mhz

Setup	
Worst Negative Slack (WNS):	1.416 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	4500

Timing constraints are not met.

80 Mhz

Setup	
Worst Negative Slack (WNS):	0.955 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	4500

Frecuencia máxima de funcionamiento: 84 Mhz

Setup	
Worst Negative Slack (WNS):	0,027 ns
Total Negative Slack (TNS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	4500

Timing constraints are not met.

Se observa fallos a partir de la frecuencia: 85 Mhz

Setup

Worst Negative Slack (WNS):	-0,058 ns
Total Negative Slack (TNS):	-0,058 ns
Number of Failing Endpoints:	1
Total Number of Endpoints:	4500

Timing constraints are not met.

La información que nos proporciona el análisis de falla para el path, se puede observar que la primera falla se origina entre la etapa de ejecución de las instrucciones y memoria, en el módulo de persistencia de datos (latch) y el módulo multiplexor utilizado para la comprobación de cero a la salida de la alu.

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay
Path 1	-0.058	11	12	132	u_MIPS/u_EX_MEM/RegWrite_reg_reg/C	u_MIPS/u_EX_MEM/Cero_reg_reg/D	11.610	2.279
Path 2	0.215	5	6	131	u_MIPS/u_IF/u_Memor...emory_reg/CLKARDCLK	u_MIPS/u_IF/u_PC/pc_reg[0]/CE	5.227	1.502
Path 3	0.215	5	6	131	u_MIPS/u_IF/u_Memor...emory_reg/CLKARDCLK	u_MIPS/u_IF/u_PC/pc_reg[10]/CE	5.227	1.502
Path 4	0.215	5	6	131	u_MIPS/u_IF/u_Memor...emory_reg/CLKARDCLK	u_MIPS/u_IF/u_PC/pc_reg[11]/CE	5.227	1.502
Path 5	0.215	5	6	131	u_MIPS/u_IF/u_Memor...emory_reg/CLKARDCLK	u_MIPS/u_IF/u_PC/pc_reg[12]/CE	5.227	1.502
Path 6	0.215	5	6	131	u_MIPS/u_IF/u_Memor...emory_reg/CLKARDCLK	u_MIPS/u_IF/u_PC/pc_reg[13]/CE	5.227	1.502
Path 7	0.215	5	6	131	u_MIPS/u_IF/u_Memor...emory_reg/CLKARDCLK	u_MIPS/u_IF/u_PC/pc_reg[14]/CE	5.227	1.502

Frecuencia de prueba:
100 Mhz

Setup

Worst Negative Slack (WNS):	-1.803 ns
Total Negative Slack (TNS):	-22.906 ns
Number of Failing Endpoints:	34
Total Number of Endpoints:	4500

Con esa frecuencia se puede observar que WNS es negativo porque lo que esta frecuencia no se puede implementar

Name	Slack ^1	Levels	Routes	High Fanout	From
Path 1	-1.803	11	12	132	u_MIPS/u_EX_MEM/RegWrite_reg_reg/C
Path 2	-0.648	5	6	131	u_MIPS/u_IF/u_Memoria_I...es/memory_reg/CLKARDCLK
Path 3	-0.648	5	6	131	u_MIPS/u_IF/u_Memoria_I...es/memory_reg/CLKARDCLK
Path 4	-0.648	5	6	131	u_MIPS/u_IF/u_Memoria_I...es/memory_reg/CLKARDCLK
Path 5	-0.648	5	6	131	u_MIPS/u_IF/u_Memoria_I...es/memory_reg/CLKARDCLK
Path 6	-0.648	5	6	131	u_MIPS/u_IF/u_Memoria_I...es/memory_reg/CLKARDCLK

En esa captura se puede observar que el problema estaría en la etapa el latch EX_MEM pero tambien se puede deducir que al subir la frecuencia comienzan a fallar mas etapas como sucede en la etapa IF. En el latch el problema sería el registro que se usa para habilitar la escritura en la memoria de registros y en la etapa IF el problema es también en la memoria de registros.

Bibliografía

1. Apuntes de las clases
2. Como crear un módulo clock, https://www.youtube.com/watch?v=ngkpvMaNapA
3. Mips, https://docplayer.net/225690-Mips-iv-instruction-set-revision-3-2-by-charles-price-september-1995.html
4. Arquitectura mips, https://www.fdi.ucm.es/profesor/jjruz/ec-is/temas/Tema%205%20-%20Repaso%20ruta%20de%20datos.pdf
5. decodificación, https://www.youtube.com/watch?v=ZryPweoBQFM
6. teoria de 5 etapas mips, https://www.youtube.com/watch?v=X0oSucJTwZU&list=PLqYtbHbPSNIgi2oZBwMfb3nJ4i3Cm-dmD
7. operaciones con inmediatos, https://www.youtube.com/watch?v=0SzpwGoRbuU
8. https://es.wikipedia.org/wiki/Extensi%C3%B3n_de_signo
9. https://la35.net/orga/mips-pipeline.html
10. Para creacion de uart sin ticks, https://forum.digilent.com/
11. instrucciones, https://www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf
12. mips reference sheet, https://uweb.engr.arizona.edu/~ece369/Resources/spim/MIPSReference.pdf