UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES



ARQUITECTURA DE COMPUTADORAS

IMPLEMENTACIÓN DE UN PIPELINE

Integrantes: Diez de Medina Quintar, Lucas Leandro

Lichtensztein, Leandro Federico

- 2013 -

Contenido

[Introducción 3](#_Toc357705076)

[Requerimientos 3](#_Toc357705077)

[Objetivos 5](#_Toc357705078)

[Alcances del trabajo 5](#_Toc357705079)

[Desarrollo 6](#_Toc357705080)

[Instrucciones 6](#_Toc357705081)

[Instrucciones Tipo-R 6](#_Toc357705082)

[Instrucciones Tipo-I 7](#_Toc357705083)

[Instrucciones Tipo-J 7](#_Toc357705084)

[Desarrollo del Pipeline 8](#_Toc357705085)

[Instruction Fetch (IF) 8](#_Toc357705086)

[Instruction decode 9](#_Toc357705087)

[eXECUTE 13](#_Toc357705088)

[memory acccess 14](#_Toc357705089)

[wRITE BACK 15](#_Toc357705090)

[fORWARDING UNIT 15](#_Toc357705091)

[HAZARD DETECTION UNIT 17](#_Toc357705092)

# Introducción

Como trabajo final de la materia Arquitectura de Computadoras, se nos solicitó implementar un pipeline para un procesador DLX, correspondiente a la familia de procesadores MIPS. El informe aquí presentado constituye una documentación del proceso de desarrollo realizado para obtener dicho Pipeline.

## Requerimientos

* El pipeline deberá ser capaz de ejecutar correctamente el siguiente conjunto de instrucciones:

|  |  |  |
| --- | --- | --- |
| **R-Type** | **I-Type** | **J-Type** |
| SLL | LB | JR |
| SRL | LH | JALR |
| SRA | LW |  |
| SRLV | LWU |  |
| SRAV | LBU |  |
| ADD | LHU |  |
| SLLV | SB |  |
| SUB | SH |  |
| AND | SW |  |
| OR | ADDI |  |
| XOR | ANDI |  |
| NOR | ORI |  |
| SLT | XORI |  |
|  | LUI |  |
|  | SLTI |  |
|  | BEQ |  |
|  | BNE |  |
|  | J |  |
|  | JAL |  |

En el archivo “Tabla de instrucciones.xlsx” entregado con este informe, puede encontrarse en detalle el funcionamiento, la sintaxis, y un ejemplo de cada instrucción.

* El pipeline implementado deberá constar de 5 etapas con un objetivo bien definido en cada una de ellas. Estas son:

**IF (Instruction Fetch):** El objetivo de esta etapa es obtener una instrucción de la memoria de instrucciones. Los elementos principales de esta etapa son el contador de programa, la memoria de instrucciones y un multiplexor que permite seleccionar la fuente del contador de programa.

**ID (Instruction Decode):** En esta etapa se decodifica la instrucción obtenida. Se determina cuáles serán las señales de control de acuerdo a cada instrucción, y se leen los registros involucrados en la instrucción. Los elementos principales de esta etapa son la unidad de control, y los registros del procesador.

**EX (Excecute):** Aquí se realiza la operación indicada por la instrucción, sobre los registros. Además se calcula aquí la dirección de los saltos (en caso de ser una instrucción de salto) y se determina si los mismos deben ser tomados o no. La unidad principal de esta etapa es la ALU, pero cobra particular importancia el sumador utilizado para el cálculo de la dirección de destino de los saltos.

**MEM (Memory Access):** Existen instrucciones que interactúan con una memoria de datos, ya sea para guardar en ella un valor, o para leer un valor desde la memoria hacia un registro. En esta etapa se realiza el acceso a la memoria de datos.

**WB (Write back):** Finalmente, obtenidos los resultados se escriben los registros nuevamente, actualizándose su valor.

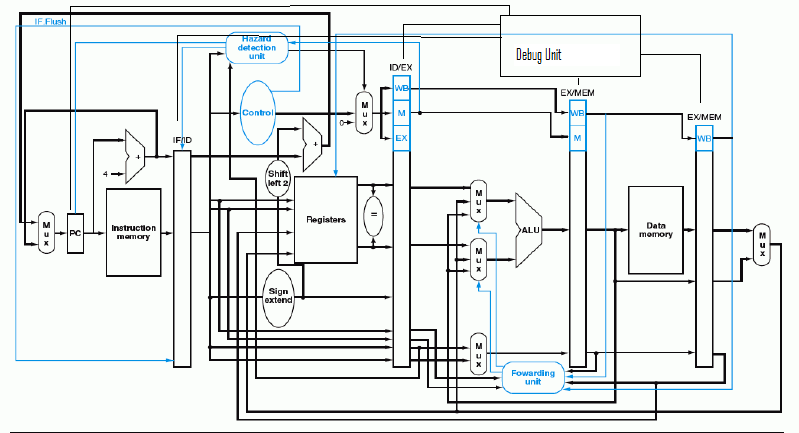
Cada etapa está separada por un registro que cumple la función de sincronizar el funcionamiento. El pipeline cuenta también con una unidad de cortocircuito y una unidad de detección de riesgos, necesarias para evitar riesgos de datos y de control durante la ejecución. A continuación se muestra un esquema del pipeline solicitado.

Figura 1: Esquema del pipeline completo.

* La memoria de datos deberá estar separada de la memoria de instrucciones, y se deberán utilizar IP Cores para su implementación.
* Deberá contarse con una unidad de Debuggeo utilizada para seguir la ejecución de las instrucciones. La misma deberá enviar por puerto serie, el estado de todos los registros del procesador, y de los registros de cada etapa del pipeline, como así también el PC actual y la instrucción que se ejecutará.
* Deberán existir dos modos de ejecución: Paso a paso, que permite ejecutar de a un único ciclo de clock; y Free Running, que permitirá ejecutar las instrucciones hasta que el programa finalice.

## Objetivos

El principal objetivo es desarrollar un pipeline de un procesador DLX que permita ejecutar correctamente el conjunto de instrucciones solicitado. Para ello, se plantearon los siguientes objetivos particulares:

* Desarrollar y testear las 5 etapas del pipeline por separado.
* Integrar todas las etapas incorporando los registros correspondientes a cada etapa.
* Desarrollar la unidad de control, la unidad de debugging, la unidad de cortocircuito y la unidad de detección de riesgos, y realizar una integración del sistema.
* Desarrollar un programa de ejemplo que permita probar el funcionamiento de todas las instrucciones.

## Alcances del trabajo

Si bien se agregará una unidad de detección de riesgos (para evitar riesgos de control), en este trabajo no se desarrollará un predictor de saltos, que nos permita realizar la ejecución perdiendo la menor cantidad de instrucciones. Cada vez que se tome un salto, se eliminarán las instrucciones que no deben ejecutarse, provocándose de esta forma la pérdida de dos instrucciones por cada salto tomado.

# Desarrollo

El desarrollo de este trabajo fue llevado a cabo utilizando el siguiente entorno:

* Lenguaje de desarrollo: Verilog
* Herramientas de desarrollo provistas por Xilinx (ISE design swite 14.1)
* Placa de desarrollo Nexys 3 de Digilent, con una FPGA Spartan 6.
* Repositorio GIT alojado en GitHub.

Se dividió el proceso de desarrollo en 3 etapas:

* Estudio de las instrucciones y su arquitectura.
* Desarrollo del pipeline realizando simulaciones con fines de prueba.
* Integración del desarrollo para ser utilizado en la placa, donde también se realizaron las pruebas de funcionamiento con la ayuda de una unidad de Debugueo.

A continuación presentamos una descripción de lo trabajado en cada etapa.

## Instrucciones

El pipeline implementado deberá ser capaz de ejecutar un conjunto de instrucciones que cumplen con ciertas características en común, a saber:

* 32 bits de longitud fija.
* Cada instrucción posee un código de operación (6 bits más significativos), y en algunos casos un código de función (6 bits menos significativos).
* Dentro de la instrucción se encuentran tanto los registros fuente (5 bits cada uno) como el registro destino (5 bits), si los hubiere.
* En caso de utilizar un valor inmediato, este siempre se encuentra en los 16 bits menos significativos de la instrucción.

El conjunto de instrucciones puede agruparse en tres categorías: Instrucciones Tipo R, instrucciones tipo I, e instrucciones Tipo J.

### Instrucciones Tipo-R

Este tipo de instrucciones realizan todo tipo de operaciones matemáticas y lógicas (suma, resta, and, or, shifts, etc). Se caracterizan por tener sus bits de operación (31:26) todos en 0, y utilizar como identificador de instrucción los bits de función (5:0). Estos le indicarán a la ALU qué operación deberá realizar.

Existen tres instrucciones (SLL, SRL, y SRA) que toman un único registro como fuente, y desplazan dicho registro *sa* veces (donde sa es obtenido de la instrucción, y significa Shift Ammount). Todas las demás instrucciones toman dos registros como fuente, y un registro destino, donde se almacenará el resultado de la operación. Todos aquellos campos que la instrucción no utilice, deberán ser iguales a 0.

La estructura de una instrucción Tipo R puede verse en la siguiente imagen.

### Instrucciones tipo RInstrucciones Tipo-I

Figura : Instrucciones tipo R

En este tipo de operaciones, uno de los operandos es un registro (**rs** o **base**), y el otro es un valor inmediato (**immediate**), incluido en los 16 bits menos significativos de la instrucción.

A su vez existen dos tipos de operaciones que utilizan valores inmediatos: operaciones desde/hacia la memoria de datos, y operaciones aritméticas o lógicas, con valores inmediatos.

Para interactuar con la memoria de datos pueden utilizarse instrucciones Load e instrucciones Store. Ambas utilizan el operando correspondiente al registro como una base, y el valor inmediato como un offset, para formar así la dirección efectiva a la que se desea acceder (tanto para leer como para escribir un dato en memoria). Las operaciones de Load toman un valor de memoria y lo guardan en el registro destino (**rt**), mientras que las operaciones de Store toman el valor del registro “destino” y lo guardan en una posición de memoria.

Existen además dos casos especiales, que son los saltos no condicionales. Para ello se utilizan los 26 bits menos significativos para indicar la dirección de destino del salto.

A continuación se muestra la estructura de las instrucciones tipo I.

Figura : Instrucciones tipo I

### Instrucciones Tipo-J

Finalmente, existe un último tipo de instrucciones que corresponden a aquellas en las que se realiza un salto incondicional, pero la dirección de destino se encuentra almacenada en un registro, en lugar de estar contenida en la misma instrucción.

Al igual que las instrucciones tipo R, el código de operación es 0, y se utilizan los bits de función (5:0) para indicar qué instrucción deberá ejecutarse. El operando **rs** (bits 25:21) es el registro que contiene la dirección efectiva a la que deberá saltarse, y en los restantes bits, en el caso de la instrucción JALR, puede indicarse un registro específico donde se guardará la dirección de retorno.

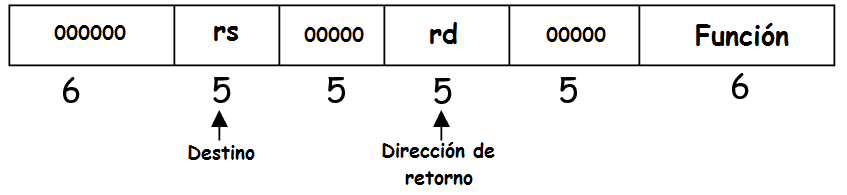
La estructura de una instrucción tipo J puede verse a continuación.

Figura : Instrucciones tipo J

Si se desea profundizar más a cerca de cada instrucción en particular, puede consultarse el archivo adjunto “Tabla de instrucciones.xlsx”. Más adelante, cuando analicemos cada una de las etapas del pipeline, volveremos sobre los distintos tipos de instrucciones para definir las señales de control que deberán utilizarse en cada caso.

## Desarrollo del Pipeline

Cuando procesamos una instrucción, la misma pasa por distintas etapas bien definidas, cada una con un objetivo en particular. Si deseamos procesar un conjunto de instrucciones, podemos dividir el procesamiento en estaciones (etapas), y colocar una instrucción en cada estación, logrando de esta manera paralelizar la ejecución.

Queda claro que el objetivo de diseño del pipeline es determinar cuáles son las tareas que deben realizarse en cada etapa. Para el caso del procesador DLX que se implementó en este trabajo, las etapas fueron mencionadas en la introducción. A modo de repaso, estas son:

* **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.

Analizaremos cada una de estas etapas en detalle.

### Instruction Fetch (IF)

Como dijimos, en esta etapa se realiza la búsqueda de la instrucción en la memoria de programa. La dirección de memoria de donde se tomará la instrucción está dada por un contador de programa (PC). En nuestro caso, utilizamos una memoria de 32 bits, por lo que cada entrada corresponde a una instrucción.

La memoria fue implementada utilizando un IP core provisto por la herramienta de desarrollo. Es una memoria ROM con 2048 palabras de 32 bits, por lo que se necesitan 11 bits de dirección (de allí que el PC tenga 11 bits). La siguiente figura muestra los módulos que componen la etapa de búsqueda de instrucción.

***Memoria***

***instrucciones***

ADDR

DR

***PC***

***Sumador***

0

1

***MUX***

*+1*

Destino de salto

PC Siguiente

Instrucción

El módulo PC es un arreglo de flip flops D activados por flanco ascendente. La memoria de instrucciones leerá el valor en el flanco descendente. El sumador es un circuito combinacional que incrementa en 1 el valor del PC. El multiplexor también es un circuito combinacional que selecciona la fuente del PC (para cuando se realizan operaciones de salto). La señal de control de este multiplexor será incorporada luego, cuando analicemos la unidad de detección de riesgos.

Esta etapa tiene como entrada la dirección de destino que se calcula en caso de estar ejecutando una operación de salto. Como salidas de la etapa se tienen la instrucción leída, y el valor del PC Siguiente (utilizado para calcular los destinos de los saltos). Cabe recordar que entre dos etapas del pipeline, siempre habrá un registro activado por flanco negativo, que será el que sincronice todas las señales de salida de la etapa anterior.

### Instruction decode

La etapa de decodificación consiste en interpretar la instrucción de salida de la etapa de búsqueda y según cual sea esta generar las señales de control, así como también proveer los registros necesarios para las operaciones a realizar.

El hardware involucrado en esta etapa es un banco de registros, donde se almacenan los registros del pipeline, un sumador para calcular la dirección de destino de los saltos, un módulo para la extensión de signo, la unidad de control, que genera las señales para el pipeline, y un registro de salida que sincroniza esta etapa con la siguiente. La siguiente figura muestra los elementos de la etapa Instruction Decode:

***Banco de***

***registros***

busA

busB

RA

RB

RW

busW

***Extensión***

***de signo***

***Sumador***

Dirección de destino del salto

PC base

Instrucción

Data A

Data B

Extensión

de signo

RegDst

ALUSrc

MemToReg

RegWrite

MemRead

MemWrite

Branch

[1:0] ALUOp

[2:0] trunk\_mode

Bne

Jump

[1:0] Jdes\_sel

Al ingresar la instrucción a esta etapa es descompuesta en varias partes y procesada por los diferentes módulos antes mencionados.

Los bits 31 a 26 , que son el código de la operación, y los bits 5 a 0, que son el formato, son ingresados a la unidad de control para generar las señales necesarias para que las operaciones se ejecuten correctamente.

Los bits 25 a 21 y 20 a 16 son las direcciones de los registros que serán requeridos para operar en la etapa siguiente, ingresan para ello al bloque de registros de la etapa.

Los bits 15 a 0 ingresan a la extensión de signo para ser convertidos a un valor de 32 bits.

Finalmente los bits 15 a 11 son la dirección donde se almacenará el resultado de la ejecución, y junto a las direcciones de los registros con los que se opera, bits 25 a 21 y 16 a 10, ingresan al registro de salida para pasar a la siguiente etapa.

El procesador cuyo pipeline se elabora posee 32 registros de 32 bits. Para su implementación se optó por instanciar 32 registros de 32 bits de tamaño, inicializando a todos con valor cero, excepto algunos a los cuales se les asignó un valor con propósito de testing. Estos valores serán mencionados más adelante.

Dicho bloque de registros permite realizar la búsqueda de dos de ellos simultáneamente, de manera que posee dos entradas de dirección y dos salidas de datos.

Por otro lado también tiene una entrada de dirección para escritura y una entrada para el dato a escribir. Para realizar la escritura existe una señal de control que habilita dicha función cuando se solicita.

En esta etapa se realiza también el cálculo de la dirección a la que se deberá saltar en caso de que la instrucción a ejecutarse así lo indique. Para ello se realiza la suma entre el pc actual y el valor contenido entre los bits 0 y 15 de la instrucción con su respectiva extensión de signo.

La extensión de signo consiste en el ingreso de los bits 15 a 0 de la instrucción en el módulo, en caso de que el bit 15 fuera 0, se completan los bits del 16 al 32 con 0. En caso contrario, los bits del 16 al 32 se completan con 1. De esta manera se obtiene a partir de un valor de 16 bits uno de 32, que permite ser utilizado por las operaciones de la etapa siguiente.

Para poder controlar el funcionamiento de todas las etapas del pipeline es necesario incorporar señales de control que garanticen que el hardware necesario para cada operación sea el que está activo en el momento preciso. Cada etapa requiere diferentes señales. Las etapas de instruction fetch e insctruction decode no requieren señales de control ya que su funcionamiento es igual independientemente de la instrucción actual. Sin embargo, existen instrucciones que estando en una etapa posterior tienen impacto sobre las etapas anteriores.

Las instrucciones de salto, por ejemplo, emiten una señal de control que es utilizada en la etapa IF para asignar al PC el valor correspondiente. De igual manera, aquellas instrucciones que guardan un valor en un registro emitirán una señal desde la etapa de WB, indicando que un valor debe guardarse en los registros, presentes en la etapa ID.

En el caso de la etapa de ejecución debo indicar de donde tomar la dirección del registro de destino donde guardaré el resultado (RegDst), la operación a realizar (ALUOp), y cuáles serán los operandos de la ALU (ALUSrc).

En el caso de memory Access, debo indicar si se produce un salto (Branch), si la memoria es leida (MemRead) y si la memoria es escrita (MemWrite).

Finalmente, en la etapa de writeback se debe indicar si el valor a guardar en el registro es la salida de la memoria o la de la etapa de ejecución (MemToReg), así como también habilitar la escritura del registro (RegWrite).

Resumiento, las señales de salida de la unidad de control son las siguientes:

* RegDst\_out (Indica el origen de la dirección del registro de destino)
* ALUSrc\_out (Entrada del multiplexor que elige la entrada de la ALU)
* MemToReg\_out (Indica que se debe cargar en un registro el valor leído en memoria)
* RegWrite\_out (Señal que permite la escritura en el registro de destino)
* MemRead\_out (Señal que habilita la lectura de memoria)
* MemWrite\_out (Señal que habilita la escritura en memoria)
* Branch\_out (Señala que la operación es un branch)
* ALUOp\_out (Indica a la ALU la operación que debe realizar)
* trunk\_mode\_out (Selecciona como se debe truncar el valor)
* Bne\_out (Señal que indica que la operación es Branch not equal)
* Jump (Señal que indica que la operación es un salto incondicional)
* Jdes\_sel (Es la señal que controla el multiplexor cuya entrada son las posibles direcciones para el salto)

La selección de los valores para las señales de control se realiza acorde a las siguientes tablas:



Además de las señales que aparecen en la Figura 5, se incorporaron señales de control que fueron necesarias para el manejo de otros elementos del pipeline cuyo diseño no estaba plenamente contemplado en el propuesto por la bibliografía consultada.

Figura 5: Señales de control para los distintos tipos de instrucciones.

Todas estas señales de control serán enviadas a las etapas siguientes a través de los registros para lograr la sincronización con las etapas por las que pase la instrucción que las generó. Como se ve en la siguiente figura, las señales de control se dividen por etapas de pipeline.



Finalmente, para detectar correctamente la instrucción que debe ejecutarse, se toman tres señales que dependen del código de la instrucción. Estos son:

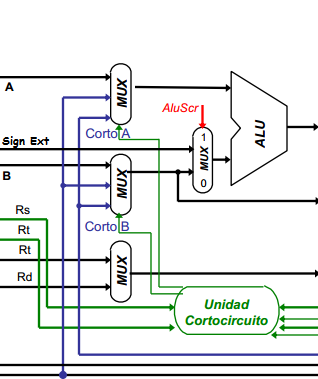
* op\_code (código de la operación)
* formato (el formato de la instrucción)
* stall\_mux (indica que se debe hacer un stall en el pipeline).

Con estos tres valores, se pretende conocer con exactitud la operación que se va a ejecutar, o la necesidad de anular una instrucción (Stall).

### eXECUTE

Durante la etapa de ejecución es cuando se realizan las operaciones indicadas en la instrucción, obteniendo a la salida el resultado de las mismas.

El hardware de esta etapa esta conformado por una unidad lógica aritmética (ALU), dos multiplexores de 3 entradas para seleccionar los operandos, un multiplexor para seleccionar la dirección de destino del resultado, y en nuestro diseño en particular se incorpora un multiplexor que selecciona la dirección del salto. Posee, al igual que las etapas anteriores, un registro de salida que sincroniza esta etapa con las siguientes.



*ALUOp y OpCode*

*PC Actual*

*Zero\_signal*

Las señales principales que ingresan a esta etapa son los dos registros, que serán los operandos de la ALU, y el código de la operación a realizar (ALUOp y OpCode).

Las señales originadas en los registros ingresan a dos multiplexores, a estos ingresan también el resultado de las dos instrucciones anteriores. El control de estos dos multiplexores es realizado por la forwarding unit, explicada más adelante.

La salida del primer multiplexor ingresa a la ALU directamente, mientras que la salida del segundo ingresa a un nuevo multiplexor que selecciona entre esta señal y la señal proveniente de la extensión de signo de la etapa anterior, esto permite la operación con literales. El control de este último multiplexor es realizado por la señal de control ALUSrc generada en la etapa anterior.

El código de la operación debería ingresar, de acuerdo a lo propuesto, al módulo de control de la ALU. En nuestro caso decidimos integrar este módulo a la ALU en sí misma, siendo su funcionamiento idéntico que si se los hubiese hecho por separado.

A la ALU ingresan los dos operandos de 32 bits, 6 bits del el código de la operación, la señal de control ALUop, generada en la etapa de decodificación, y el valor del pc actual. Sus salidas son el resultado de la operación y una señal que indica si el resultado es 0.

Dentro de la ALU, se utilizan dos multiplexores conectados uno luego del otro, al primero ingresa el ALUOp, esto selecciona entre suma (para operaciones de load y store), entre resta (saltos condicionales), operaciones tipo R y salto tipo JAL o JALR.

La salida de este multiplexor es el código de cada operación que se debe realizar, esto es el código de suma, resta, salto y el propio de la instrucción R que ingresó.

Esta señal de salida es la entrada al selector del multiplexor siguiente, cuyas entradas son las salidas de todos los circuitos combinacionales necesarios para ejecutar las instrucciones seleccionadas en los requerimientos. La salida de este multiplexor es el resultado de la operación.

Para una instrucción tipo R se ingresa el código de la misma y se realiza la operación correspondiente, poniendo el resultado a la salida.

Para una instrucción de load o store se realiza una suma entre los operandos para el cálculo de la dirección de memoria correspondiente, esto es sumar el offset a la base indicada.

Para las operaciones de salto condicional se realiza una resta, y luego mediante una compuerta nand se calcula la señal de cero que indica la igualdad o desigualdad de los operandos.

En el caso de una instrucción tipo JALR o JAL, se toma el pc actual como resultado de la operación, de manera que este sea almacenado en el registro correspondiente.

Por otro lado, se sintetiza también un multiplexor cuyas entradas son la dirección de los registros de destino dependiendo de si la operación a ejecutar es de tipo R o de tipo L. La selección de la dirección de destino dependerá de la señal de control RegDest.

Se agrega al diseño propuesto una compuerta AND de tres entradas, que son las salidas de circuitos combinacionales que indican si la fuente del PC debe ser la de un salto. Un caso es que la operación sea un salto si los operadores son iguales, en este caso se comprueba que la señal de Branch y la de cero estén en alto. En caso de un salto si los operadores son distintos se comprueba que esté en alto la señal de BranchNotEqual y en bajo la señal de cero. Finalmente si la señal de salto incondicional está en alto también se debe producir.

De esta manera el salto se “decide” en la etapa de ejecución en vez de en la etapa de acceso a memoria como se planteó inicialmente, esto evita la perdida de una instrucción adicional en caso de saltos en el programa.

Para poder realizar lo anterior se debe poseer la dirección a la que se realizará el salto, para ello se implementa un multiplexor cuya salida es el valor del próximo PC. Las entradas son el offset indicado por la instrucción branch, la dirección obtenida del literal de las instrucciones J y JAL, y el valor del registro que contiene la dirección de salto para JR y JALR.

Según el tipo de instrucción se selecciona como salida el valor correspondiente. De esta manera al finalizar la etapa de decodificación se obtiene combinacionalmente, o sea de manera inmediata, la dirección de destino y la decisión del salto.

En el diagrama propuesto en la consigna esta decisión y cálculo se realiza en la etapa siguiente (memory access) con la consecuente pérdida de una instrucción, adelantando estas operaciones a la ejecución se pierden sólo dos ciclos del pipeline.

Todas las salidas de esta etapa se conectan con la siguiente a través del registro de salida.

### memory acccess

La etapa de acceso a memoria realiza la lectura y escritura propiamente dicha de la memoria de datos. Ingresan a esta etapa las señales de control de lectura y escritura, el resultado de la ALU (para el caso en el que se deba guardar en memoria), la dirección a la que se desea acceder y la dirección del registro de destino para el caso de que se lea un valor.

El hardware que constituye esta etapa está compuesto por dos multiplexores, utilizados en caso de requerir truncamiento del valor que se lee o escribe, un bloque de memoria (que será la memoria de datos), un multiplexor para determinar si se debe escribir un registro desde memoria o no, y un registro de salida para sincronizar las etapas, que al igual que en las etapas anteriores no se muestra.

Memoria

de datos

ADDR

Write Data

Read Data

Trunk unit

trunk\_mode

Data B

ALU Result

Trunk unit

trunk\_mode

La memoria de datos está implementada utilizando un IP core, pero a diferencia de la memoria de programa esta es de tipo ram. Está compuesta por 2048 registros de 32 bits de ancho cada uno.

Para las operaciones de carga y almacenamiento de byte, half word, con o sin signo, se utilizan dos multiplexores, uno para load y otro para store, que se encargan de realizar el truncamiento y la extensión de signo necesaria en cada caso. Estas unidades de truncamiento son diseño propio y no parte del pipeline propuesto en la bibliografía.

El valor leído y el resultado de la ALU se pasan a la etapa siguiente a través del registro de sincronización antes mencionado.

### wRITE BACK

La etapa de write back está formada por un multiplexor que selecciona qué valor debe escribirse en el registro de salida.

Las entradas son la salida de la ALU y el valor leído de la memoria de datos. La señal MemToReg, generada en la unidad de control, define qué valor será el enviado a la etapa de decodificación.

MUX

MemToReg

Mem Data

ALU Result

Dato a guardarse

en el registro destino

### fORWARDING UNIT

#### DEPENDENCIAS DE REGISTROS

Con los elementos descritos anteriormente se construye un pipeline que permite la ejecución de todas las instrucciones solicitadas por la consigna, pero existen ciertos casos en los que una instrucción depende del resultado de la anterior, en este caso el pipeline hasta aquí descripto ejecutará erróneamente los cálculos. Para evitar este problema se incorpora la unidad de cortocircuito o forwarding unit.

Supongamos el caso en el que una operación genera un resultado en el registro r. Este valor no estará efectivamente en dicho registro hasta dentro de dos pulsos y medio de reloj. En el primer pulso se pondrá como salida de la etapa de ejecución, en el segundo como salida de la etapa de acceso a memoria y en el flanco siguiente (medio pulso) se graba en el registro.

Supongamos ahora que la instrucción siguiente utiliza este registro r como operando, si se ejecutara en el pipeline anterior el valor de r sería incorrecto, ya que aún no se ha grabado el resultado de la operación previa. En este caso, como dijimos el resultado está en el registro de salida de la etapa de execute.

Ahora la instrucción siguiente a esta, vuelve a utilizar el registro r como operando, en este caso sucede lo anterior, pero ahora el registro está en la etapa de memory access.

Esta relación entre el resultado de una instrucción y los operandos de la siguiente se denomina dependencia de registros.

Se puede deducir que existen cuatro tipos, que el registro de destino sea Rs o Rt de la instrucción antes ejecutada y que exista dependencia con la instrucción en el registro de ejecución o en el registro de acceso a memoria.

La solución a este problema es adelantar el valor del registro necesario para ser operado antes de que el registro en sí sea escrito.

La unidad de cortocircuito tiene por objetivo controlar los multiplexores que seleccionan la entrada a la ALU, tal como se muestra a continuación:



A estos multiplexores ingresan, como se dijo anteriormente, el valor de los registros (salida de ID), el resultado almacenado en el registro a la salida de execute y la salida del write back, que en pulsos es la salida del registro de memory access.

La forwarding unit recibe como entrada la dirección donde de escribe el resultado de la operación que actualmente está en la etapa de acceso a memoria y en la etapa de write back. Si También ingresan las dos direcciones de los operandos de la instrucción que está por ejecutarse.

Si alguno de los registros de los operandos es uno de los que deben escribirse en próximos pulsos, se genera la señal que realice la selección del valor “adelantado” como entrada a la ALU.

De esta manera cualquier instrucción puede utilizar cualquier registro sin riesgo a errores en el cálculo debido a las dependencias de registros.

### HAZARD DETECTION UNIT

#### Dependencias de datos

Existen casos en los que la dependencia no puede ser salvada con el uso de la unidad de cortocircuito. Uno de estos casos es cuando una instrucción trata de leer un registro inmediatamente después de una instrucción de load sobre el mismo registro.

Este tipo de dependencia se denomina dependencia de datos, y es solucionada mediante el uso de la hazard detection unit.

Para detectar este tipo de dependencia basta con controlar que la señal de MemRead (la que indica la escritura en un registro desde la memoria) esté en alto y que alguno de los operandos de la instrucción actual sea el registro a escribir. Este control se realiza durante la etapa de decodificación de la instrucción.

Si se da el caso anterior se soluciona retrasando la instrucción actual durante un pulso de clock extra, esto se logra haciendo un stall. Un stall es mantener el PC durante un pulso de clock extra y no escribir el registro de salida de la etapa de búsqueda durante el mismo tiempo. También se ponen el 0 todas las señales de control, de manera que funciona como si se generara una instrucción NOP desde la etapa de decodificación.

Para lograr lo explicado en el párrafo anterior, la hazard detection unit, tiene tres salidas, una es la señal que habilita la escritura del PC, otra la señal que habilita la escritura del registro IF/ID y una más que maneja un multiplexor cuya entrada son las señales de control y la salida pueden ser esas señales o el valor 0.



#### DEPENDENCIAS DE CONTROL

Existe un tercer tipo de riesgo que puede presentarse en el pipeline durante la ejecución de un programa, este sucede cuando una instrucción de salto (BEQ, BNE, J, JAL, JR, JALR) produce un cambio en el orden normal de ejecución de las instrucciones, o sea el PC cambia no al valor siguiente, sino a otro determinado por la instrucción.

El salto se decide (puede tomarse o no), en la etapa de ejecución (en el pipeline propuesto se decidía en la etapa de acceso a memoria), esto quiere decir que para cuando el salto se realice ya habrá una instrucción buscada y otra decodificada.

Al tomar el salto la instrucción inmediatamente siguiente a esta deberá ser la de la nueva dirección del PC, por lo tanto debo impedir que las instrucciones que fueron buscadas luego de la del salto se ejecuten y realicen modificaciones en los registros o la memoria.

Para evitar que se produzcan errores debo descartar la instrucción buscada, esto se realiza utilizando una señal de flush sobre el registro IF/ID.

También se debe evitar que las señales de control de la instrucción decodificada realicen modificaciones, por lo tanto se ponen todas en 0, de la misma manera que en las dependencias de datos.

Toda esta lógica es llevada a cabo en la hazard detection unit, que pone en alto la señal de IF/ID.flush y en 0 todas las de control, insertando en el pipeline dos operaciones NOP.

La lógica para detectar el salto es muy simple, basta con controlar la señal que define la entrada del PC, si esta está en alto quiere decir que el valor del PC no será el sucesivo sino el indicado por un salto.

### debug unit

FALTA LA UART

# Programa de prueba

Para testear el correcto funcionamiento del pipeline se decidió escribir un programa en el que se tuvieran en cuenta todas las instrucciones y que a su vez generara todas las dependencias posibles, esto es: de registros, de datos y de control.

Se dieron a los registros los siguientes valores iniciales:

Registro 1 = 10

Registro 3 = 15

Registro 5 = -5

Registro 6 = 12

Registro 15 = 5

Registro 16 = 32'b10101010\_10101010\_10101010\_10101010

A todos los demás registros se les dio el valor 0.

En la siguiente tabla se muestra el código ejecutado junto a los registros que son modificados y al valor que estos adoptan. En el caso de las instrucciones de salto tomados se muestra el valor del nuevo PC.

|  |  |  |
| --- | --- | --- |
| **PC** | **Instrucción** | **Registros modificados** |
| 0 | SUB r2, r1, r3 | r2 = -5 |
| 1 | AND r12, r2, r5 | r12 = -5 |
| 2 | OR r 13, r6, r2 | r13 = -1 |
| 3 | ADD r14, r2, r2 | r14 = -10 |
| 4 | SW r15, 100(r2) | Se almacena 5 en la posición 95. |
| 5 | BEQ r2, r12, 9 | Pc = 15 |
| 6 | AND r12, r2, r5 | Stall |
| 7 | OR r13, r6, r2 | Stall |
| 15 | XOR r2, r1, r3 | r2 = 5 |
| 16 | NOR r1, r2, r3 | r1 = -16 |
| 17 | LW r8, 100(r12) | r8 = 5 |
| 18 | SW r16, 100(r12) | Se almacena el registro 16 en la posición 95. |
| 19 | BNE r1, r3, 9 | PC = 29 |
| 20 | AND r12, r2, r5 | Stall |
| 21 | OR r13, r6, r2 | Stall |
| 29 | SLL r4, r2, 3 | r4 = 40 |
| 30 | SRL r7, r6, 1 | r7 = 6 |
| 31 | SRA r8, r5, 1 | r8 = -3 |
| 32 | SRLV r9, r14, r15 | r9 = 134217727 |
| 33 | SRAV r10, r14, r15 | r10 = -1 |
| 34 | SLLV r7, r2, r15 | r7 = 160 |
| 35 | ADDU r8, r7, r2 | r8 = 165 |
| 36 | J 46 | Pc = 46 |
| 37 | AND r12, r2, r5 | Stall |
| 38 | OR r13, r6, r2 | Stall |
| 46 | SUBU r9, r8, r7 | r9 = 5 |
| 47 | SLT r4, r2, r3 | r4 = 1 |
| 48 | SLT r4, r3, r2 | r4 = 0 |
| 49 | LB r8, 100(r12) | r8 = -86 |
| 50 | LH r9, 100(r12) | r9 = -21846 |
| 51 | JAL 61 | Pc = 61 r31 = 52 |
| 52 | AND r12, r2, r5 | Stall |
| 53 | OR r13, r6, r2 | Stall |
| 61 | LWU r10, 100(r12) | r10 = -1431655766 |
| 62 | LBU r11, 100(r12) | r11 = 170 |
| 63 | LHU r13, 100(r12) | r13 = 43690 |
| 64 | SB r10, 100(r12) | Memoria = 170 |
| 65 | SH r10, 100(r8) | Memoria = 43690 |
| 66 | LW r9, 100(r12) | r9 = 170 |
| 67 | LW r10, 100(r8) | r10 = 43690 |
| 68 | JR r7 | Pc = 160 |
| 69 | AND r12, r2, r5 | Stall |
| 70 | OR r13, r6, r2 | Stall |
| 160 | ADDI r11, r7+8 | r11 = 168 |
| 161 | ADDIU r12, r7+8 | r12 = 168 |
| 162 | ANDI r13, r11&33008 | r13 = 160 |
| 163 | ORI r14, r11|33008 | r14 = 33016 |
| 164 | XORI r15, r11^33008 | r15 = 32856 |
| 165 | JALR r21, r9 | Pc = 170 |
| 166 | AND r12, r2, r5 | Stall |
| 167 | OR r13, r6, r2 | Stall |
| 170 | LUI r16, 5 | r16 = 327680 |
| 171 | SLTI r11, r8, -5 | r11 = 1 |
| 172 | SLTI r12, r9, -5 | r12 = 0 |
| 173 | SLTIU r13, r8, 5 | r13 = 0 |
| 174 | SLTIU r14, r4, 5 | r14 = 1; |

# RESULTADOS

ACA IRIAN LAS IMÁGENES DEL SIMULADOR Y ALGUN TEXTO QUE EXPLIQUE QUE USAMOS EL SIMULADOR

# cONCLUSIÓN

Concluir

# Bibliografia (?)