UNC – FCEFyN – Arquitecturas de Computadoras **2015**

Trabajo Final: Procesador DLX

(Familia MIPS)



**Integrantes**:

|  |  |  |
| --- | --- | --- |
| Apellido | Nombre | Matricula |
| Reinoso | Ever Denis | 34.253.991 |
| Gonzalez | Mauricio Leonardo | 35.048.294 |

|  |  |
| --- | --- |
| **1** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

**Contenido**

Descripción General. 3

Características: 3

Desarrollo. 4

Construcción del Procesador Monociclo. 4

Construcción del Procesador Segmentado. 9

Implementación del Debugger. 20

Configuración y uso. 22

|  |  |
| --- | --- |
| **2** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

**Descripción General.**

Se presenta la creación de un Pipeline del procesador DLX, el cual pertenece a la familisa MIPS. Dicho procesador realiza la ejecución de instrucciones por medio de un pipeline en forma segmentada, donde cada instrucción se ejecuta en varios ciclos de reloj dividiendo así la instrucción en etapas, este tipo de arquitectura aumenta el rendimiento al tener que ejecutarse muchas instrucciones.

Se implementa también el control de riesgos para la ejecución segura de instrucciones Los riesgos estructurales que corresponden a los riesgos cuando más de una instrucción requiere el uso de un recurso simultáneamente, se soluciona por la forma intrínseca del sistema. Riegos de datos cuando una instrucción requiere leer un dato que otra instrucción no ha terminado de escribir, es decir que puede leer un valor no actualizado, se resuelve con una unidad de control y cortocircuito de modo que si se detecta este tipo de riesgos el dato es llevado de una etapa posterior a una anterior. Finalmente los riesgos de control ocurren cuando se requiere realizar un salto condicional y debe evaluarse la condición, como se verá en más detalle este se soluciona con la implementación de otra unidad que realiza el cálculo de la condición en etapas tempranas.

Por otro lado el procesador cuenta con una unidad de Debug la misma sirve para ejecutar el programa de manera paso a paso, viendo los valores del sistema que cambian con cada instrucción, o continuo donde el programa se ejecuta hasta el final y recién en ese momento se muestran los valores finales del sistema. Para la realización del debugger se configuró una unidad de UART para el envió de órdenes a la placa Nexyz 3 y el retorno de datos a la computadora. También requirió la implementación de una aplicación Python para poder visualizar los datos del sistema de una forma más prolija y ordenada como se detalle en las secciones finales.

**Características:**

* Memoria para datos separada de la otra memoria para instrucciones.
* Ejecución Segmentada.
* Set de instrucciones MIPS:

1. R-type
   * SLL, SRL, SRA, SLLV, SRLV, SRAV
   * ADDU, SUBU
   * AND, OR, XOR, NOR
   * SLT, SLTU
2. I-Type
   * LB, LH, LW, LWU, LBU, LHU, SB, SH, SW
   * ADDI, ADDIU, ANDI, ORI, XORI, LUI
   * SLTI, SLTIU, BEQ, BNE
   * J, JAL
   1. J-Type
      * JR, JALR

* Unidad de detección de Riesgos de Control y de Datos

|  |  |
| --- | --- |
| **3** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

* Unidad de cortocircuitos
* Módulo UART para Debugger
* Aplicación de Debugger

**Desarrollo.**

Para el desarrollo se comienza con una implementación más simple de un procesador monociclo, sin unidades de control de riesgo ya que no las requiere al no ejecutar las instrucciones de manera paralela. Luego se sigue con el desarrollo del procesador segmentado en el que se añade latchs intermedios para poder separar las etapas por ciclos de reloj y se agregan las unidades para el manejo de los riesgos.

Existe otra implementación intermedia que es la multiciclo en la que se separan por ciclos de reloj las etapas pero las instrucciones van ejecutándose individualmente este tipo de arquitectura no sigue la línea de desarrollo del pipe segmentado, ya que este continua por una adaptación diferente del monociclo por lo que el multiciclo se dejo de lado.

Finalmente se muestra la incorporación de la unidad de Debugger para el control del procesador.

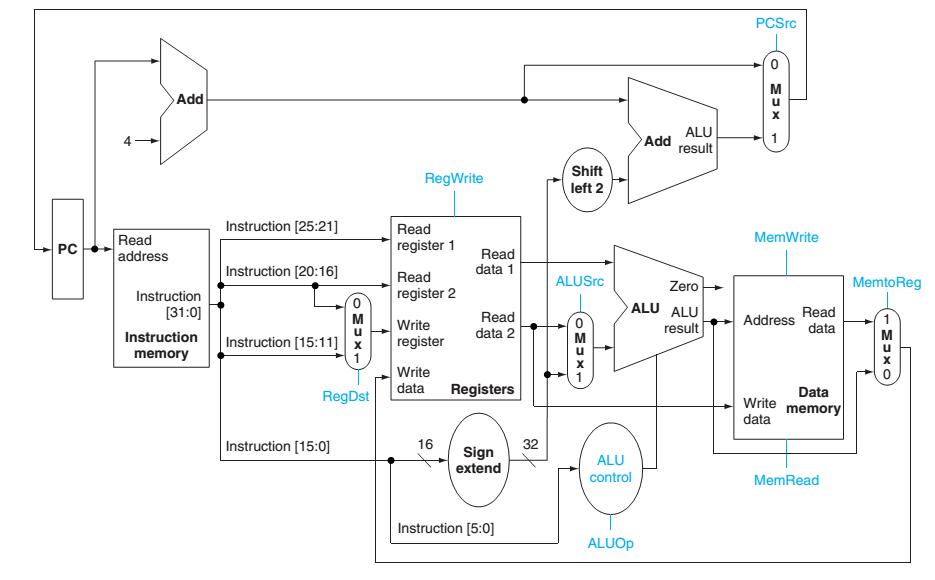
**Construcción del Procesador Monociclo.**

Se implementan para esta arquitectura únicamente las instrucciones esenciales R-type (ANS, OR, ADD, SUB, NOR, SLT), I-Type (LW, SW, BEQ) y J-Type (J).

Se muestra primero el diagrama en bloques de esta arquitectura sin la instrucción salto. Se pretende ver a grandes rasgos los cambios que se fueron efectuando en el procesador y dar una explicación detallada de la implementación final.

|  |  |
| --- | --- |
| **4** | Pipeline |

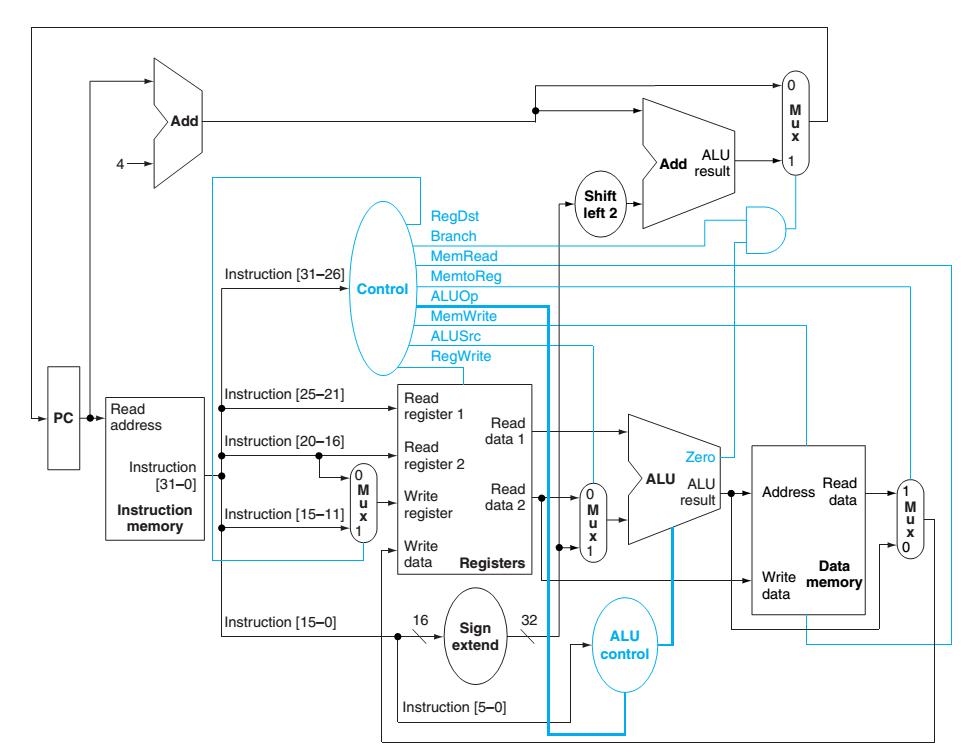
UNC – FCEFyN – Arquitecturas de Computadoras **2015**



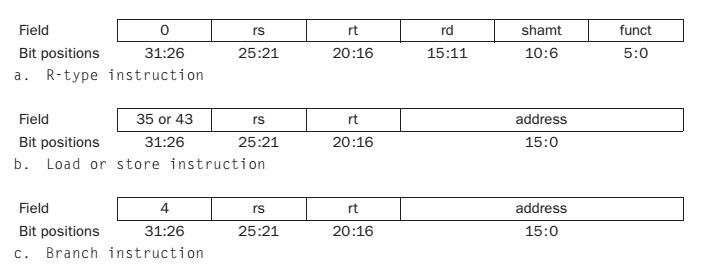
Se ven señales azules que maneja la unidad de control para determinar el comportamiento de los bloques y los multiplexores. La misma debe analizar la instrucción y determinar las correspondientes señales.

|  |  |
| --- | --- |
| **5** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**



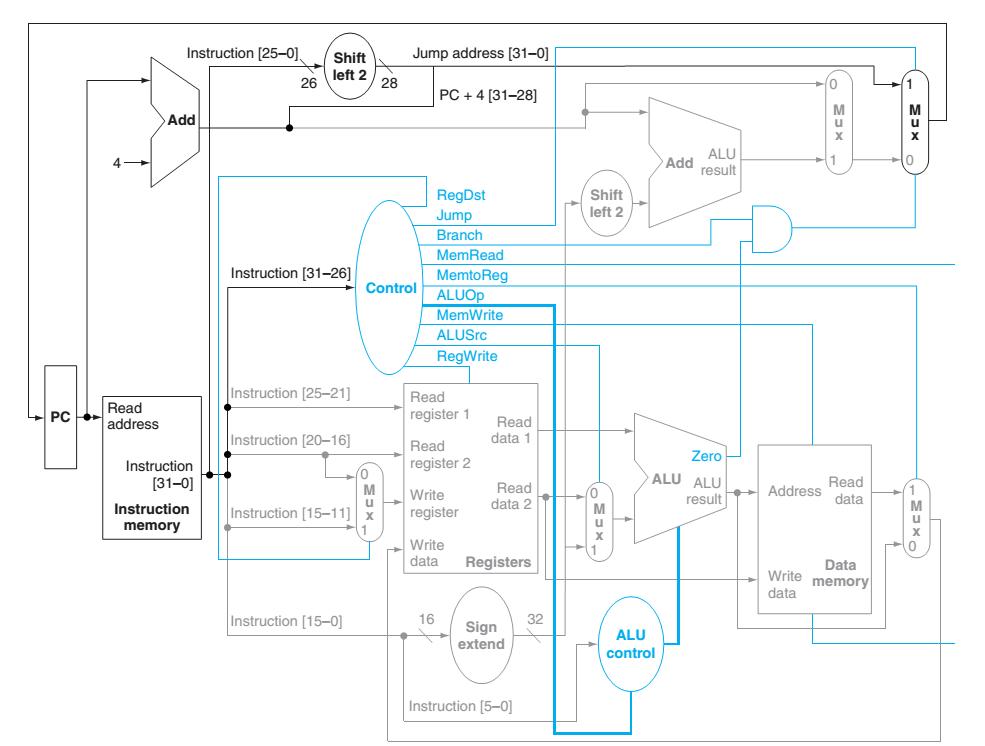
De acuerdo al tipo de instrucción que se tenga las señales cambiarán ya que el comportamiento de los bloques será diferente. Además al decodificar la instrucción los bits que se distribuyan de la instrucción dependerán del tipo de instrucción como se puede apreciar los bits de la instrucción representan valores diferentes para instrucciones diferentes.



Finalmente la versión final que se tiene del procesador monociclo con la instrucción Jump.

|  |  |
| --- | --- |
| **6** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**



Para esta implementación se explican brevemente los distintos módulos a medida que se ejecuta la instrucción. También debe notarse que los nombre para los componentes son los utilizados en la implementación del programa para mayor entendimiento del mismo.

PC es el contador del programa *ProgrammCounter*, es decir almacena la instrucción siguiente a ser ejecutada (su dirección) la misma debe ser de 32 bits. Solo cambiará su valor en cada ciclo de reloj. Su entrada va depender de la instrucción ejecutada anteriormente esta determinará la salida del *MuxPCSrc* ya sea la suma del PC en el *AdderInstructionFetch* (este suma 4 ya que cada dirección de instrucción ocupa 4 bytes) o la dirección calculada para un salto condicional esta se toma de *AdderInstructionDecode***.**

La salida de *ProgrammCounter* es el número de instrucción para tomar de *InstructionMemory* el valor de la instrucción. Este valor es entrada de muchos de los otros bloques y los bits que se tomen para cada bloque también varían como se aprecia en el diagrama. Ya que los distintos tipos de instrucciones se comportan de manera diferente es explicara el comportamiento para los distintintos tipos.

|  |  |
| --- | --- |
| **7** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

* **Instrucciones R-Type**: para este tipo de instrucción se deben tomar las direcciones de los 3registros dos fuentes y un destino. El *MuxRegDest* selecciona rd como destino. Luego con estas direcciones se obtienen del *RegisterBank* los valores de los registros fuente para que los lea la ALU. Es necesario seleccionar la salida del registro B como la entrada de la ALU en *MuxALUSrcB*. Por otro lado se transmite a *AluControlUnit* los primeros 4 bits de la instrucción que corresponden a la función que debe ejecutar la ALU ya que la ALU tomo como líneas de control las salidas de esta. Se calcula la operación que corresponda y el resultado debe ser la entrada de escritura de dato en el *RegisterBank*, para esto *MuxMemToReg* debe seleccionar como salida el resultado de la ALU. Entonces se guardael valor calculado en el registro especificado por rd.
* **Instrucciones I-type**:

1. Para una instrucción LW se seleccionan del registerBank el destino rt (se debe seleccionar en MuxRegDst) y la base rs para calcular la dirección junto con el offset address. La dirección debe sumarse junto con el registro base para obtener la dirección del valor a cargar. Para esto se debe pasar dicha dirección a un valor de 32 bits con SignExtensionUnit y elegirla como entrada de MuxALUSrcB. El resultado de la ALU es la dirección de DataMemory de donde se obtiene el valor de esa dirección de memoria y se selecciona en MuxMemToReg para cargar el registro. La ALU determina que debe hacer una operación de suma porque MainControlUnit indica a AluControlUnit que se trata de una operación de LW o SW.
2. Un SW es bastante similar solo se almacena un registro ya que la segunda salida del RegisterBank es la entrada de datos de DataMemory, se debe mandar la señal WriteData en este bloque y la escritura a registro no tiene relevancia.
   1. Si la instrucción es un salto condicional se deben leer 2 registros del *RegisterBank* que se compararan con una resta en la ALU determinada por la *MainControlUnit* al igual que con LW. La ALU además de la salida del resultado tiene una flag Z para indicar si el valor del resultado es 0. En este caso solo si es 0 los valores son iguales y debe efectuarse un salto ya que suponemos un BEQ. Entonces esta flag se utiliza como entrada de una compuerta AND *ANDBranchZero*, su otra entrada sale de la *MainControlUnit*, solo si esta indica que es un branch con 1 y la flag zero da 1 sedebe tomar el salto, ya que la salida de esta compuerta controla el *MuxPCSrc*. Así seleccionaría el offset del salto que debe hacer antes, una extensión a 32 bits en la SignExtensionUnit, un desplzamiento en ShiftInstructionDecode de 2 bits (que equivale a multiplicar por 4, así se estaría desplazando en relación a instrucciones) y sumar con el PC actual en AdderInstructionDecode para ser seleccionado como valor de salida en MuxPCSrc y MuxJump.

* **Instrucciones I-Type:** el formato para esta instrucción es bastante más simple que lasanteriores, se muestra a continuación

|  |  |
| --- | --- |
| **8** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**



Siendo address la dirección a donde se pretende saltar y el código de operación del salto. Se toma la dirección de InstructionMemory para realizársele una concatenación en ShiftJumpInstruction con el PC siguiente, es decir se toman los primeros 4 bits del PC (MSB) como los más significativos y la dirección como los 26 bits menos significativos. Posteriormente si se realiza un desplazamiento de 2 bits a la izquierda para que sea una dirección de 32 bits. El usar los 4 bits más significativos del PC nos proporciona a que el salto este en una posición cercana al programa. Esta nueva dirección es seleccionada en MuxJump como la entrada para el nuevo PC, a ese multiplexor lo controla la MainUnitControl al identificar que la instrucción es un salto incondicional.

Se realizan unas observaciones de MainControlUnit, esta unidad recibe el código de operación de la instrucción y a partir de esta se determinan todas las señales de control tanto para multiplexores como para la compuerta AND del branch, las señales de escritura/lectura en DataMemory, también la escritura para RegisterBank y el tipo de instrucción para ALUControlUnit. Como ejemplo se muestra la salida para una instrucción LW.

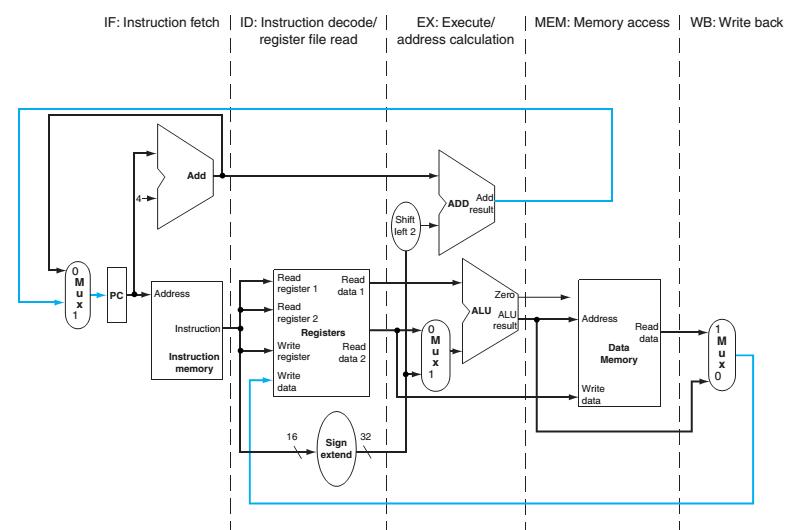
outputMuxRegDst = 1'b0; outputMuxJump = 1'b0; outputMuxALUSrcB = 1'b1; outputMuxMemToReg = 1'b1; outputRegWrite = 1'b1; outputMemRead = 1'b1; outputMemWrite = 1'b0; outputBranch = 1'b0; outputALUOp = 2'b00;

**Construcción del Procesador Segmentado.**

En este tipo de procesador las instrucciones se ejecutan por etapas como se ve son 5 etapas bien definidas, cada una de estas se ejecuta en un clock del reloj de manera simultánea. Se muestran los límites y los componentes que abarcan en relación a la implementación previa.

|  |  |
| --- | --- |
| **9** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**



Entonces ahora se cuenta con 5 etapas:

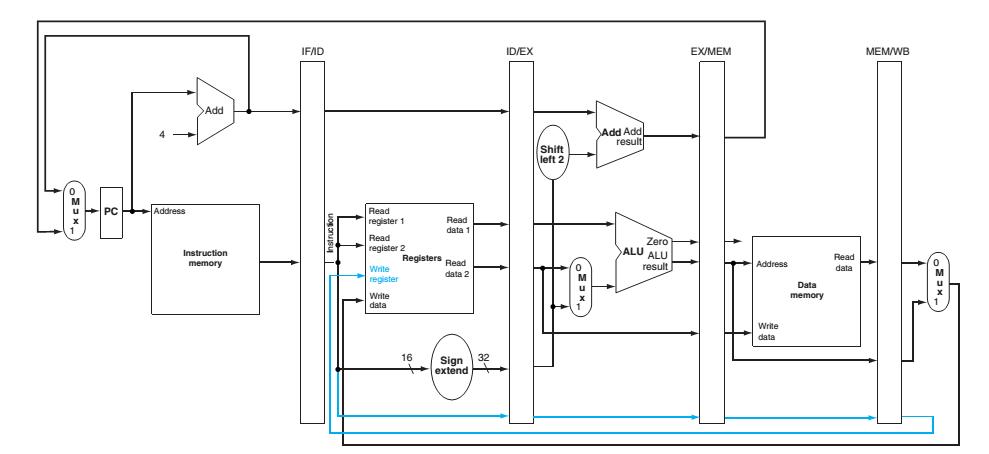
1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

Pero para poder separar la instrucción en etapas no basta una separación abstracta, se requiere de latchs intermedios entre etapas de modo que se guarden los valores de cada instrucción a medida que va pasando por cada etapa, así los datos obtenidos en un ciclo están disponibles para la siguiente etapa. Además de que permiten controlar las etapas, es decir el avance o detención de instrucciones. Debe notarse también en el siguiente gráfico que la etapa de WB no tiene un latch con IF ya que no se requiere mantener ningún valor entre estas etapas.

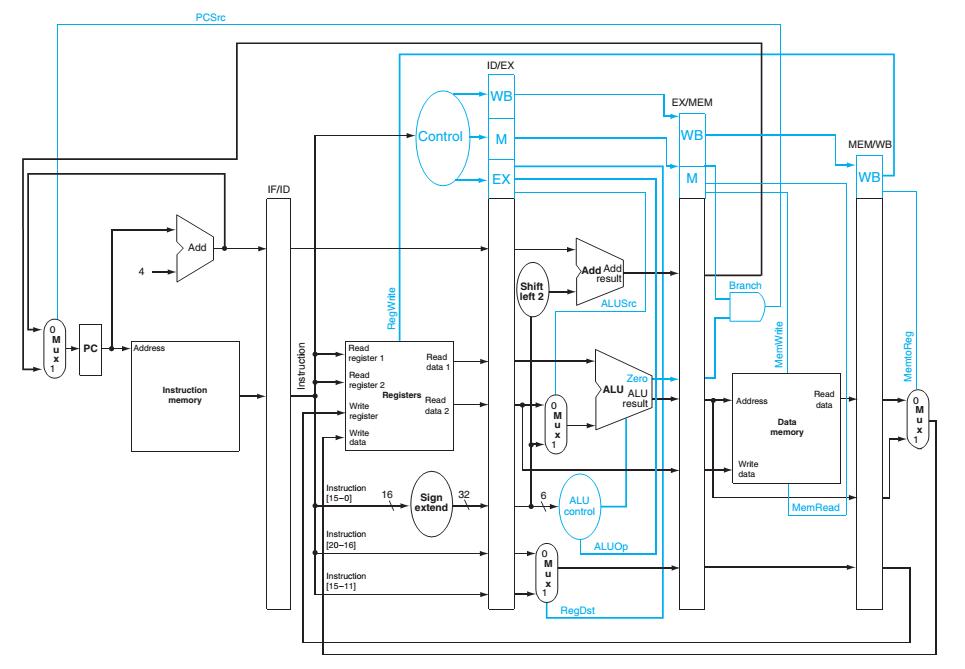
Con respecto al modelo anterior se hace una corrección ya que la dirección de escritura para registros en una instrucción Load ya no proviene de la etapa de Instruction Memory proviene de la etapa WB es decir del Latch MEME/WB de modo que el pipeline queda así.

|  |  |
| --- | --- |
| **10** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**



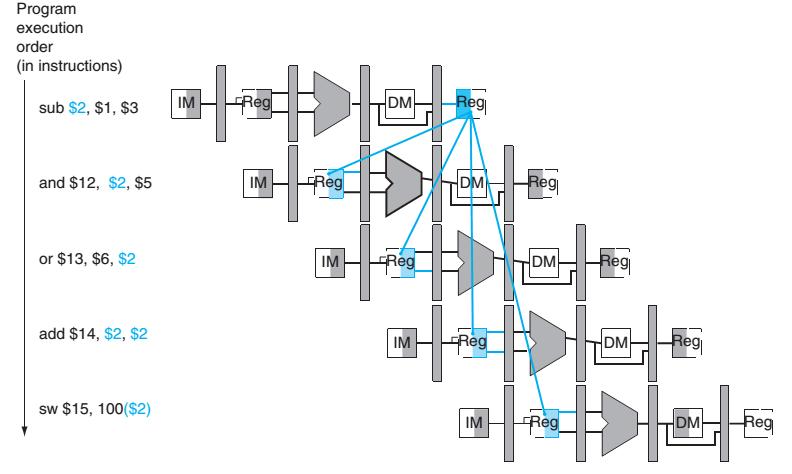
El paso siguiente es evidenciar las señales de control las mismas ahora deben transferirse por medio de los Latches junto con los datos ya que no hay otro modo de conservarlas a medida que cambian las instrucciones. Además las señales de control solo se utilizan para las últimas 3 etapas de la instrucción ya que el pipeline se comporta de la misma manera en las primeras 2 etapas. Como se puede ver la señal RegWrite viene de la etapa de WB.



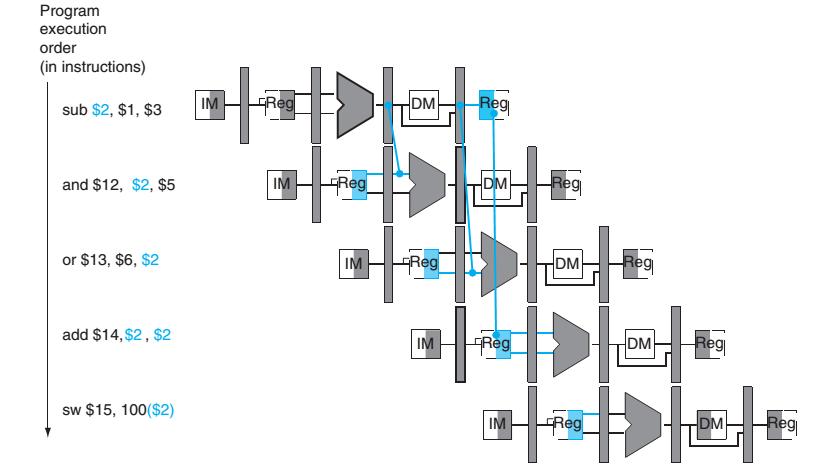
|  |  |
| --- | --- |
| **11** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

Los riesgos que se deben corregir se aprecian en el siguiente gráfico, es decir son riesgos de datos cuando las instrucciones posteriores requieren de un registro que se está cambiando en la instrucción actual.



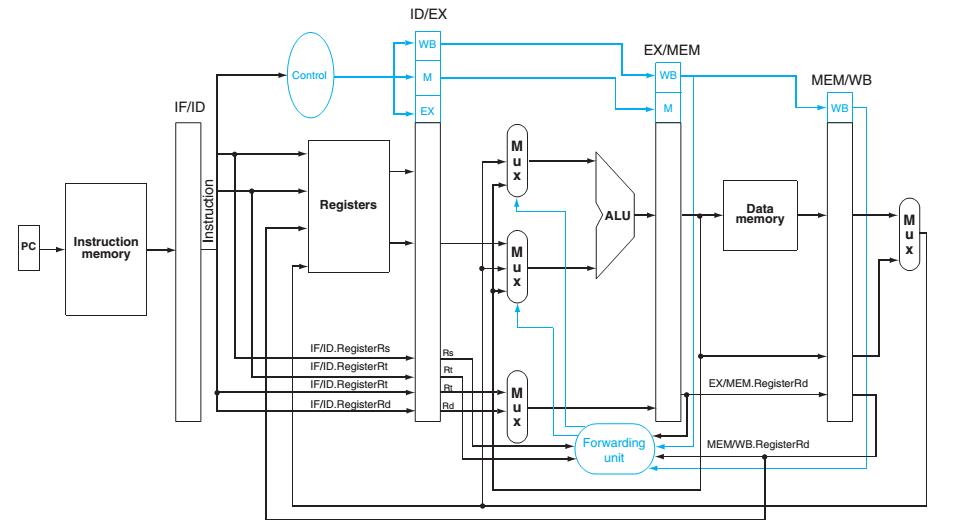
Este tipo de error se corrige con una unidad de cortocircuito, ForwardingUnit que permite transmitir el resultado de la ALU directamente a la entrada de los recursos que lo necesiten y no se tenga que esperar a que el dato se cargue en el banco de registro.



|  |  |
| --- | --- |
| **12** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

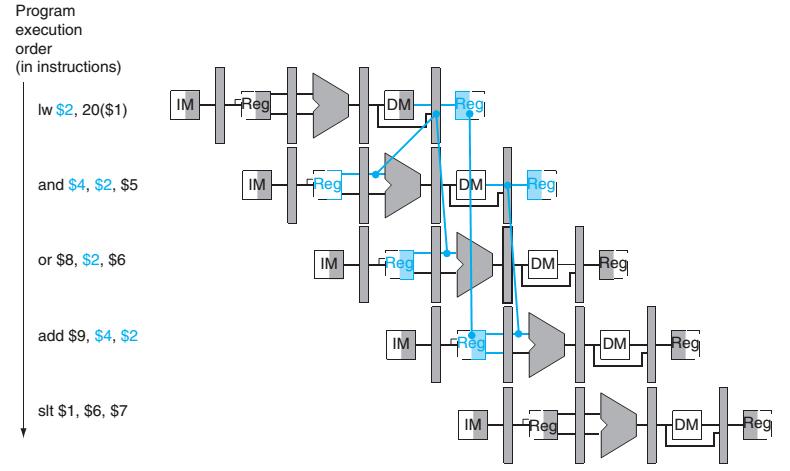
Para efectuar el control se deben añadir entradas a la ALU es decir debe contar con un multiplexor para el dato A, que selecciona el valor del registro o el resultado calculado previamente por la ALU. Al multiplexor que ya existía para el dato B se le añade otra entrada también proveniente del resultado calculado. Ambos multiplexores serán controlados por la ForwardingUnit en caso de ser necesario se habilita el cortocircuito se realiza comparando los registro fuente de la operación en el latch ID/EX con el registro destino de la operación en los latchs EX/MEM y MEM/WB.



Sin embargo surge un problema cuando la dependencia es en función de un LW en este caso solo se dispondrá del dato cuando se haya realizado el acceso a memoria, y como se ve en la imagen no es posible hacer un cortocircuito a una etapa anterior (es decir que ocurrió en un ciclo de reloj anterior) porque corresponde a un tiempo anterior.

|  |  |
| --- | --- |
| **13** | Pipeline |

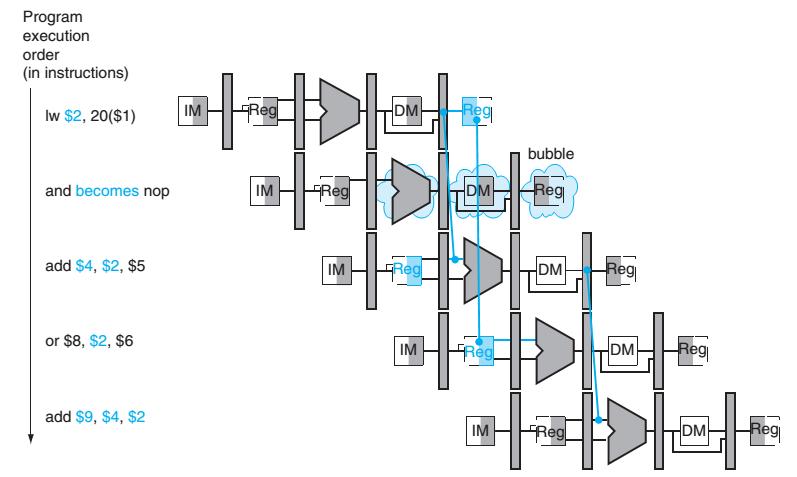
UNC – FCEFyN – Arquitecturas de Computadoras **2015**



Para solucionarlo se debe detener el pipeline de la instrucción con dependencia de datos en la etapa EX de modo que se pueda permitir el acceso a memoria de la instrucción anterior. Se realiza con un stall en el que el PC no cambia y los valores en el Latch ID/EX son mantenidos por lo que se pasan registros nulos a los siguientes Latchs que equivaldría al envió de una burbuja (NOP). Luego el dato es transmitido también por una unidad de cortocircuito de modo que solo se pierda un clock y no 2.

|  |  |
| --- | --- |
| **14** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

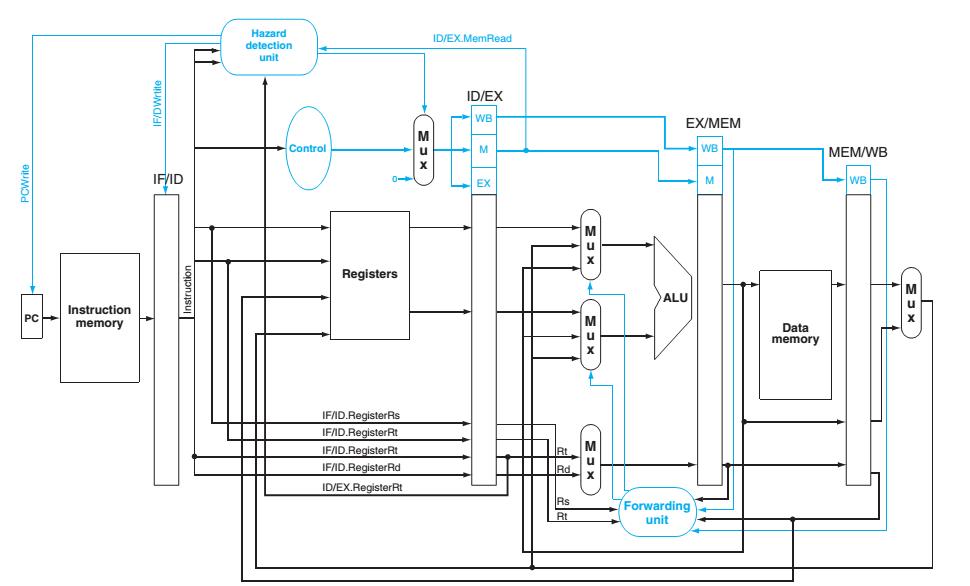


En el pipeline equivale a agregar una unidad para la detección de riesgo, HazardDetectionUnit. Se debe dar cuenta si se trata de una instrucción de salto con MemRead\_EX y comparar si el registro a cargar en el load es usado como fuente de alguna operación en este caso no habilita ni la escritura del PC ni del latch IF/ID, además se añade otro control de modo que al latch ID/EX solo se le pasen valores nulos en caso de un stall, por medio de un multiplexor controlado por la unidad de detección de riesgo.

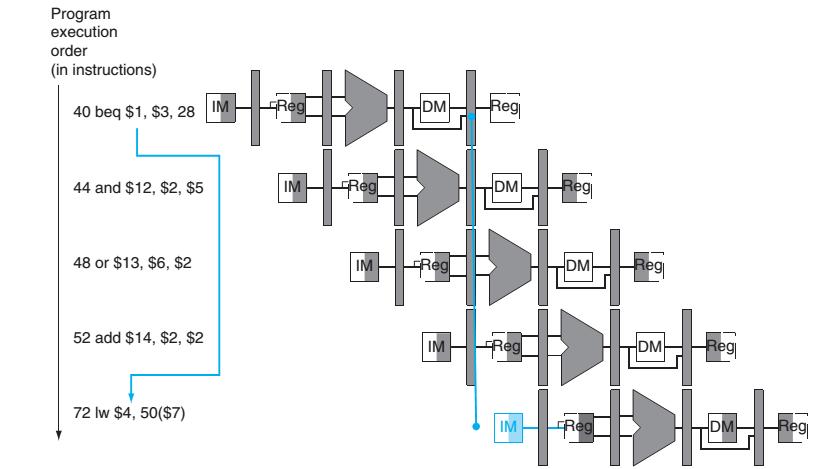
Para la entrada A de la ALU se incorpora otra entrada al multiplexor en caso de que se detecte este riesgo, este es el cortocircuito para no perder 2 ciclos de reloj, de su control se encarga la ForwardingUnit.

|  |  |
| --- | --- |
| **15** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**



Se tiene también que arreglar los riesgos de control, estos surgen de no saber si la condición será verdadera o no, por lo que no se puede determinar si se debe tomar el salto.



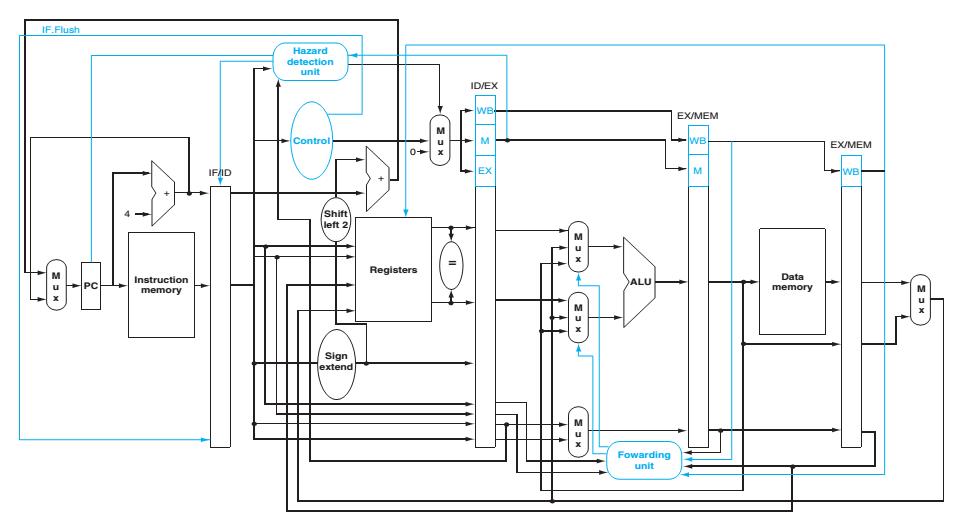
Existen varias técnicas que solucionan este riesgo pero se optó por elegir todos los saltos como NO tomados, es decir buscando la instrucción siguiente si el programa no tomara el salto. Y para ahorrar ciclos de reloj la comparación se realiza en una etapa anterior a EX es decir en ID con

|  |  |
| --- | --- |
| **16** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

una unidad de igualdad, de modo que solo se perderá un ciclo cuando el salto es tomado ya que se debe descartar la instrucción que se cargo en IF y pasar una burbuja (valores nulos) a ID.

La implementación final del procesador segmentado se muestra a continuación con la unidad de igualdad y los elementos faltantes de la etapa IF.



Se explica más en detalle los nuevos componentes en las etapas del procesador que no se encuentran en el procesador monociclo.

* **Latchs Intermedios**: son los encargados de mantener los datos y señales decontrol para las etapas de la instrucción. Los mismos se escriben (si están habilitados) y leen con cada clock del reloj. Hay una diferencia con los dos primeros latchs estos tienen una entrada de control latch flush que implica una burbuja es decir un NOP. Además el latch IF/ID tiene otra señal de control LatchWrite habilita escrituras para acciones distintas en caso de ser un branch, se escribe el latch y se descarta la instrucción ya que el salto se toma, mientras que con un load se tiene que mantener la instrucción, es decir solo se requiere un stall.
* **StageIF**: corresponde a la búsqueda de una instrucción por lo que abarca todos loscomponentes relacionados con el PC se mantienen los multiplexores anteriores MuxPCSrc, MuxJump y se añade otra para diferenciar los tipos de jump ya que se añadieron mas instrucciones y la dirección del jump puede provenir de un registro MuxJumpType. Se encuentra también en esta etapa el sumador AdderIF y la memoria de instrucciones InstructionMemory. Se destaca que el ProgramCounter solo se actualiza con una nueva señal de control añadida PCWrite que determina si debe producirse un stall o no.

|  |  |
| --- | --- |
| **17** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

* **StageID**: esta es la etapa de decodificación de instrucción, es la etapa máscompleja ya que contiene la unidad de detección de riesgo y de control, se detallan sus componentes.

o HazardDetectionUnit: pueda insertar un "Stall" o parada del Pipeline entre la instrucción de Load y las posteriores (que usan su resultado). Para chequear la instrucción de Load, el control para la "Hazard Detection Unit" consta de una sola condición. La primera comprobación es si el bit "MemRead\_EX" que proviene de "StageEX" esta en 1, dado que la única instrucción que lee la memoria de datos es un Load. Las siguientes dos comprobaciones sirven para ver si el registro destino "Rt" de la instrucción de Load que ya se encuentra en la etapa EX (StageEx), coincide con algunos de los registros fuentes (Rs o Rt) de la instrucción siguiente que ya ingreso a la etapa actual (StageID). Si se producen las condiciones anteriores, las instrucciones siguientes al Load pararan 1 ciclo de reloj, es decir que durante 1 ciclo el Pipeline no procesara instrucciones nuevas. Luego de esta parada, la "ForwardingUnit" puede manejar las dependencias de datos y continuar la ejecución. Para lograr un "Stall", mantenemos parada la instrucción en StageID (actual) y deshabilitamos el incremento del PC en StageIF para no perder las instrucciones siguientes. Finalmente, ponemos el selector del "MuxStall" en 1 para que establezca las 9 señales de control de "MainControlUnit" todas a cero e indicar al Pipeline "no hacer nada" (y esto equivaldría a un NOP, el cual actúa como una burbuja).

o MainControlUnit: al igual que en el monociclo dependiendo del código de operación de la instrucción determina las señales de control para la misma, es decir los controles de los mux, y otras señales para unidades de memoria o ejecución. Se muestra un ejemplo para una operación de

Jump.

outputRegDst = 2'b00; // x

outputJump = 1'b1; // Indicamos una operacion de Jump

outputJumpType = 1'b0;

outputMemToReg = 2'b00; // x

outputRegWrite = 1'b0;

outputMemRead = 1'b0;

outputMemWrite = 1'b0; outputBranch = 1'b0;

outputALUControlLines = 4'b1111; // None outputLoadControlLines = 3'b000; // x outputStoreControlLines = 3'b000; // x

Si se tratara de una instrucción R-type se debe hacer un control más detallado ya que ALUControlLines va depender de la función de la instrucción y debe indicarse ya que esta salida va directamente a la ALU, es decir no hay un control independiente como en el monociclo.

|  |  |
| --- | --- |
| **18** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

1. MuxStall: determina si al latch\_ID\_EX se le transimiten los valores de la instrucción o valores de una NOP en caso de un Stall.
2. ShiftJumpInstruction: para concatenar los 4 MSB del ProgrammCounter con la dirección de salto desplazada 2 bits a la izquierda como se hizo anteriormente.
3. SignExtensionUnit al igual que antes para la extensión a 32 bits de valores inmediatos de 16 bits para instrucciones de memoria y saltos condicionales. Estos últimos igual que antes requieren de ShiftID para el desplazamiento y AdderEX para el cálculo de la nueva dirección e

instrucción

o RegisterBank no cambia en relación a la implementación anterior.

* 1. EqualityUnit: se utiliza para comparar valores en caso de un branch, y determinar si se toma el salto, en relación a si es un BEQ o un BNE. Además los operandos a comparar pueden venir de etapas posteriores por lo que se cuentan con dos multiplexores MuxDataA, MuxDataB que pueden traer el dato de las otras etapas. Estos multiplexores por ser un riesgo de datos los controla la ForwardingUnit.
* **StageEX**:
  1. La ALU tiene más operaciones que en el monociclo, por otro lado sus líneas de control provienen de la MainControlUnit. Sus entradas corresponden a 2 multiplexores que seleccionan entre el registro, o en caso de haber dependencia de datos, de un valor a escribir en el registro inputMuxMemToReg, o el resultado de la operación anterior de ALU inputALUResult. Además la entrada B añade otro mux por si se quisiera

tener como entrada la salida de SignExtensionUnit para un load por ejemplo.

* 1. ForwardingUnit: es encargada de realizar los cortocircuitos cuando se tiene dependencia de datos, lo logra controlando las señales de los multiplexores asociados. Por ejemplo para un riesgo de datos con un branch en el que un registro implicado en alguna instruccion ($t8 en este ejemplo), se usa en la siguiente instruccion (un Branch). Es necesario enviar el registro $t8 desde "StageEX" (salida de la ALU) a la etapa anterior (StageID) para que el Branch compare correctamente con el valor actualizado de $t8:

ADD $t8, $t1, $t2 BEQ $t8, $t9, 1

* **StageMEM**: al entrar la dirección de memoria para un store primero se transfierea la StoreUnit de donde se determina por medio de las líneas de control que tipo de store es, para un SB solo se deben seleccionar los 8 bits menos significativos realizando una extensión a 8 bits y mantener el signo. La salida es la dirección para DataMemory que es equivalente al monociclo. Pero si la instrucción es un load

|  |  |
| --- | --- |
| **19** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

entonces la dirección debe pasar por LoadUnit que tiene instrucciones sin signo como LBU pero el análisis es similar.

* **StageWB**: simplemente se cuenta con un multiplexor que dependiendo si la señalde control está habilitada manda al registro el dato que corresponda según la instrucción.

**Implementación del Debugger.**

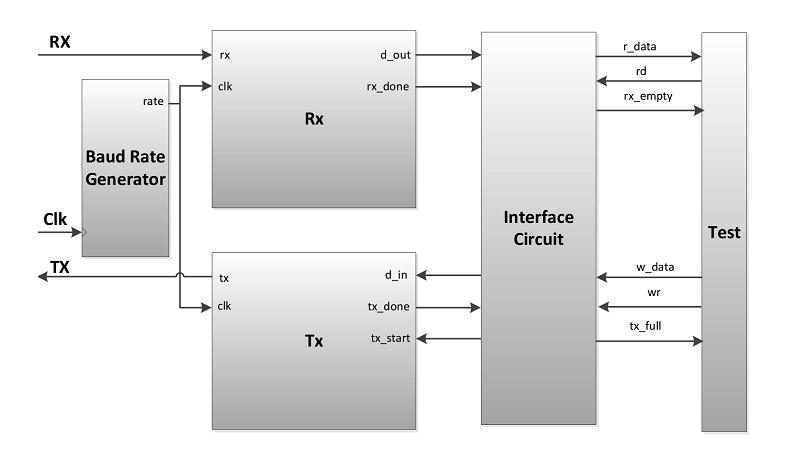
Se tiene el desarrollo de un modulo para debug que explota la comunicación por el puerto serie, UART que son las siglas de "Universal Asynchronous Receiver-Transmitter", es decir un transmisor-receptor asincrónico por lo que se deben tener ciertas precauciones ya que no se sincroniza por ningún clock como se explica más adelante con mayor detalle, se deben utilizar bits tanto para el inicio como para el final de la transmisión por parte del transmisor, mientras que el receptor debe sobre-muestrear los bits recibidos para acercarse a la zona ”más central del bit”.

La unidad de debug servirá para poder implementar los dos modos que se requieren para la realización del trabajo: el modo continuo y el modo paso a paso.

En el modo continuo se concluye la ejecución de todo el programa, es decir, que aparecerán los registros y señales de control con los últimos valores.

En el modo paso a paso se podrá ver cómo se van modificando los registros y las señales de control en cada ciclo de clock. Por lo tanto se podrá ir observando cómo avanzan las instrucciones etapa por etapa.

El diagrama de bloques de la UART será el siguiente:



|  |  |
| --- | --- |
| **20** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

Se detallan sus componentes.

**Baud Rate Generator**: Genera un Tick 16 veces por Baud Rate. Para un Baud Rate de 9600 setiene

50MHz / (9600 \* 16) = 326 ticks

En definitiva el Baud Rate Generator es un contador módulo 326. Es práctica común utilizar una velocidad de muestreo de 16 veces el baud rate deseado. Lo que significa que cada bit serial es muestreado 16 veces.

**Receptor Rx:** recibe el dato Rx y los ticks necesarios para muestrear, y envía a la FIFO el datocon la confirmación de que se completó la recepción. La secuencia de estados de la Recepción es la siguiente (Asumiendo N bits de datos, M bits de Stop).

1. Esperar a que la señal de entrada sea 0, momento en el que inicia el bit de Start. Iniciar el Tick Counter.
2. Cuando el contador llega a 7, la señal de entrada está en el punto medio del bit de Start. Reiniciar el contador.
3. Cuando el contador llega a 15, la señal de entrada avanza 1 bit, y alcanza la mitad del primer

bit de datos. Tomar este valor e ingresarlo en un shift register. Reiniciar el contador.

1. Repetir el paso 3 N-1 veces para tomar los bits restantes.
2. Si se usa bit de paridad, repetir el paso 3 una vez más.
3. Repetir el paso 3 M veces, para obtener los bits de Stop.

Para su implementación se utiliza una maquina de estados finitos ASM.

**FIFO**: buffer encargado de manejar los estados y sincronizar la transmisión, guardando yenviando los datos listos para que el transmisor los mande a la computadora personal de manera correcta.

**Transmisor Tx:** El transmisor será el encargado de recibir el dato del buffer FIFO y enviarloa la computadora. Es esencialmente un shift-register que funciona a una velocidad determinada. Dicha velocidad puede estar determinada por 1 ciclo de tick enable generado por el Baud Rate Generator dado que no se necesita sobremuestreo (por lo general se utiliza el mismo generador de ticks que el subsistema de Rx y se mantiene un contador interno para adaptar a las necesidades de Tx). Un bit es desplazado fuera del shif-register una vez cada 16 ticks del Baud Rate Generator. El diagrama ASM es similar al del caso Rx

Para que el pipeline funcione en modo Paso a Paso es necesario que sea inicializado con el envio de una tecla “o”, se debe analizar si la tecla apretada siguiente es una “S” (paso a paso) o una “C” (continuo) y de ello dependerá el comportamiento del sistema. Es por ello que tanto los Latches como el Program Counter pasaran a ser manejados exclusivamente con la unidad de

|  |  |
| --- | --- |
| **21** | Pipeline |

UNC – FCEFyN – Arquitecturas de Computadoras **2015**

debug y ya no con la señal de Clock. Es decir, la unidad de debug pondrá un ENABLE que permitirá la ejecución de las instrucciones durante un ciclo y también permitirá que se actualice el PC cuando este deba hacerlo.

La unidad que se encarga de este análisis es **StepUnit** que analiza la tecla que fue presionando comparando el byte recibido si es una “s” o “c” habilita latchs y ProgramCounter. Pero sos si es “s” habilita el siguiente módulo PipeToTx. Además existe otra verificación para determinar si hay o no instrucciones es decir si la siguiente instrucción es un Halt que equivale a un opcode con todos bits en 1.

**PipeToTx** toma como entrada todos los registros datos de memoria y valores para cadaetapa (datos y señales de control), para enviarlos uno a uno por Tx. Requiere para ello el uso de otra unidad denominada **SplitUnit** que se encarga de separar la palabra en 4 bytes para que pueda ser transmitida. Envía los bytes a la FIFO para que el transmisor TX obtenga de la cola los datos evitándose un overrun. Y enviándolos a la computadora personal.

Los datos enviados por UART son recibidos por una aplicación Python que los analiza y organiza para mostrarlos de forma correcta, para su uso lea la siguiente sección.

**Configuración y uso.**

La configuración de los pines se realiza con un archivo de extensión ucf, por usar la Nexys 3 el archivo debería estar configurado de la siguiente manera para su funcionamiento correcto.

NET "clk100Mhz" LOC = V10; NET "rx" LOC = N17;

NET "tx" LOC = N18;

NET "resetUART" LOC = B8;

Además al establecer la conexión serie en la computadora debe asegurarse que sea el Puerto COM correcto e iniciar la aplicación Debugger después de inciar el procesador en la Nexys 3. Los comandos que se cuentan son:

* Connect: se inicia la conexión.
* Step: se ejecuta una instrucción a la vez.
* Continues: se ejecuta el programa completo y se muestra el estado del procesador solo al final.

|  |  |
| --- | --- |
| **22** | Pipeline |