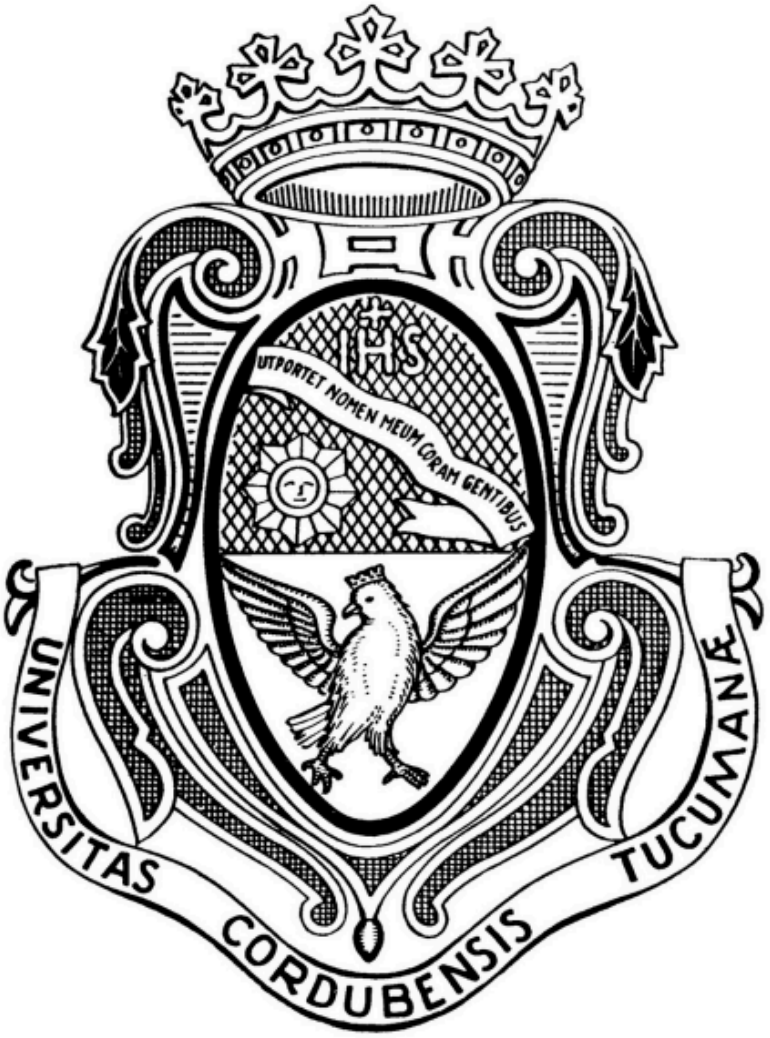
TRABAJO PRÁCTICO FINAL ARQUITECTURA DE COMPUTADORAS

MIPS



Profesores:
Pereyra Martin
Rodriguez Santiago

Alumnos:
Aichino Ignacio Daniel 40673680 (ignacio.aichino@mi.unc.edu.ar)
Vignolo Gabriel Enrique 39080905 (gabriel.vignolo@mi.unc.edu.ar)

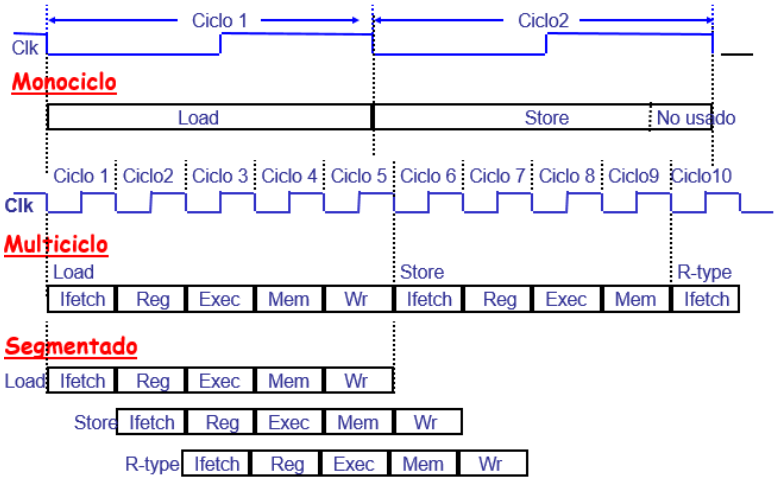
Objetivo

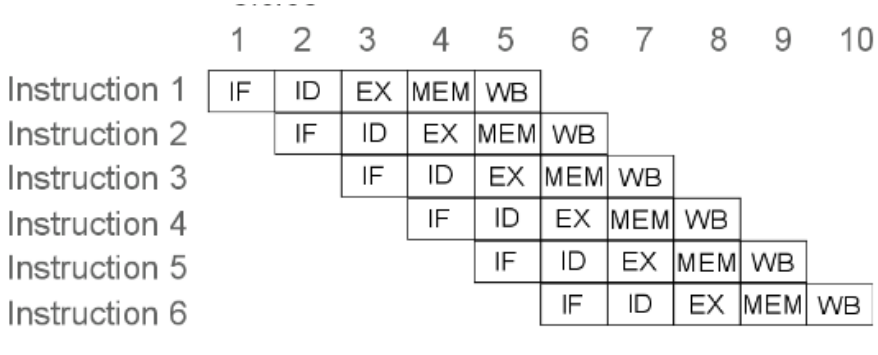
Implementar el desarrollo del pipeline de ejecución del procesador MIPS en nuestra placa adquirida Basys 3. Para ello se utiliza el lenguaje de programación de hardware Verilog.
MIPS es una familia de microprocesadores que utiliza RISC por lo que más adelante se especifica el set de instrucciones soportado de 32 bits siendo todas las instrucciones de tamaño fijo y solamente dos de ellas acceden a memoria.

PIPELINE MIPS y sus etapas

Implementación de etapas para ejecutar instrucciones es realizado mediante la técnica de **segmentación** que permite que múltiples instrucciones se pueden solapar en etapas duramente la ejecución y aprovechar el paralelismo, es así donde una instrucción empieza a recorrer el **pipeline** que consta de etapas independientes y secuenciales. Por lo tanto, en cada una de las etapas del pipeline se estará ejecutando tareas de diferentes instrucciones.

En resumen un pipeline segmentado nos facilita que todas las instrucciones tengan la misma duración y que los acceso a memoria se den en instrucciones Load y Store.
La dificultad es que aparecen ciertos riesgos de datos, estructurales y de control que se resuelven en detalle más adelante.





Etapas:

- **IF (Instruction Fetch):** Búsqueda de la instrucción en la memoria de programa mediante la dirección indicada del PC (Program Counter).
- **ID (Instruction Decode):** Decodificación de la instrucción y lectura de registros. También se generan las señales de control necesarias para las etapas del pipeline y la ejecución correcta de cada instrucción en cada etapa.
- **EX (Execute):** Ejecución de la instrucción propiamente dicha, se puede tratar de operaciones aritméticas lógicas como también el cálculo de direcciones para instrucciones específicas.
- **MEM (Memory Access):** Esta es la etapa en la cual se realiza lectura o escritura desde/hacia la memoria de datos.
- **WB (Write back):** Escritura de resultados en los registros. En esta etapa se actualizan los registros del pipeline para aquellas instrucciones que tienen como resultado un registro de destino.

Instrucciones permitidas:

- 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

Riesgos

Lo que se necesita es que cada ciclo inicie una nueva instrucción, por lo tanto, los riesgos van a ser situaciones que impiden que en cada ciclo se inicie la ejecución de una nueva instrucción.
Nuestro procesador cuenta con la implementación para atender los siguientes tipos de riesgos dada la segmentación:

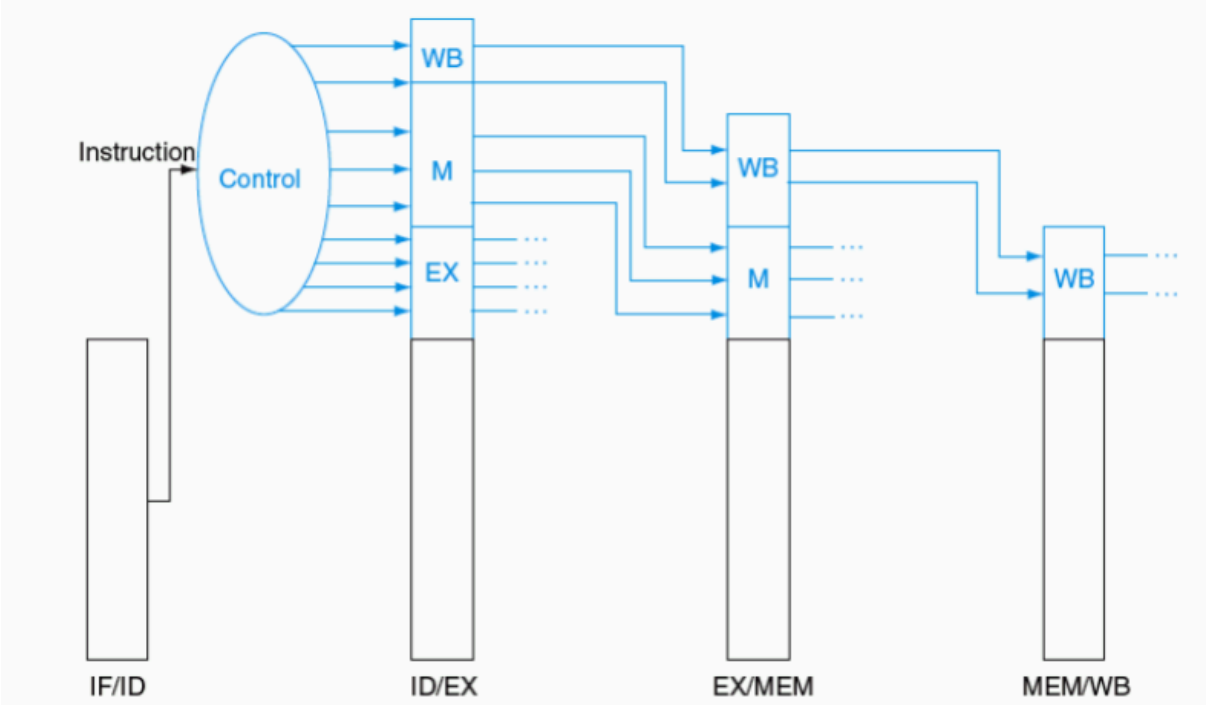
- **Estructurales.** Se producen cuando dos instrucciones tratan de utilizar el mismo recurso en el mismo ciclo. Si se tuviera una única memoria de instrucciones y datos no se podría leer una instrucción y leer un dato en el mismo ciclo.
- **De datos.** Proviene de dependencias de datos entre instrucciones. Se intenta utilizar un dato antes de que esté disponible, es decir que una instrucción necesita el resultado de una instrucción anterior.
- **De control.** Se intenta tomar una decisión sobre una condición todavía no evaluada.

Señales de Control

Para realizar un diseño correcto se tuvo que pensar el flujo de las señales de control necesarias, muchas de estas señales viajan a lo largo del pipeline para realizar un control en los distintos módulos del procesador y realizar la acción correcta durante la ejecución de cada etapa e instrucción. Estas señales de control se generan en la etapa de ID ya que es la etapa en la que se decodifica la instrucción actual y que serán necesarias para la ejecución de la misma instrucción como las que continúan. Para cada instrucción se necesitan distintas señales es por ello que se construyó una tabla la cual nos permite saber

y controlar los elementos que se manejan en la instrucción que se quiere ejecutar en cada etapa.

A continuación se deja una referencia a la Tabla de Señales de Control



Modos de operación:

El programa a ejecutar debe ser cargado en la memoria del programa mediante un archivo ensamblado. Para lograr esto se realizó un programa ejecutable desarrollado en el lenguaje de programación Python que llevará adelante la carga del programa en assembler y el control de la ejecución. Mediante una interfaz GUI se solicitará al usuario que seleccione el programa a cargar con extensión **.asm** y que indique el port para la comunicación serial con nuestra placa Basys3 y así poder enviar las instrucciones a la memoria de instrucciones y luego ejecutar el programa de manera continua o controlada.

El procesador estará a la espera para recibir el programa en assembler (previamente "traducido" a binario en Python) mediante el módulo de la **Debug Unit** (es decir mediante la interfaz de UART).

Una vez cargado el programa, se tiene dos **modos de operación:**

- **Continuo**, se envía un comando a la FPGA por la consola desplegada al ejecutar el programa en Python y hace el envío por 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. En este punto visualizará el último estado del programa al finalizar, los valores de los 32 registros del banco de Registros y los valores que se encuentran en la Memoria de Datos.
- **Paso a paso**: De igual manera que el caso anterior por la consola desplegada por el programa en Python se indicará enviando un comando que se enviará por UART y se incrementará un ciclo de Clock y avanzara el programa paso a paso. En cada **step** que se avance se visualiza el valor de los registros del banco de registros y los valores que se encuentran en la memoria de datos, teniendo además el valor actual del PC y clock actual. Podremos visualizar el PC en verde cuando se trate de un salto y en rojo cuando haya una burbuja. Por último se brinda información de los

valores de registros importantes en la los latches que separan cada segmento para llevar un control con más detalle.

PIPELINE

ETAPA IF

La etapa IF contiene una instancia del módulo PC para manejar el contador de programa que contendrá el address de la instrucción a ser ejecutada. También permite almacenar instrucciones dentro de la memoria que son cargados por la Unit Debug antes de comenzar la ejecución del programa.
Mediante un Multiplexor se obtiene el PC correspondiente ya que este multiplexor nos permite seleccionar el PC correcto frente a instrucciones del tipo J y Branches, caso contrario el valor nuevo del PC corresponde a PC+4.

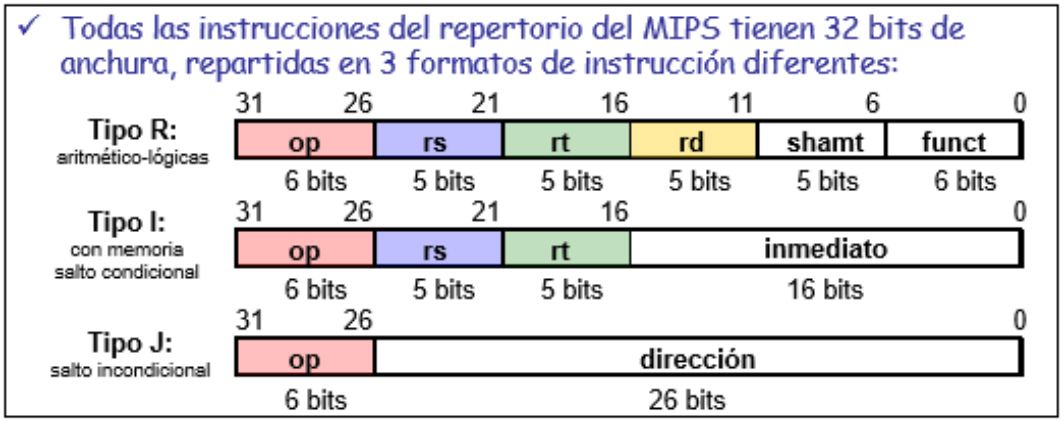
Módulos de etapa IF:

- **IF.v:** Inicializa la etapa.
- **pc.v:** Utiliza el PC obtenido desde la etapa ID dependiendo de la predicción realizada. Puede ser PC+4, el PC calculado por una instrucción Jump o Branch o en caso de JALR la unidad de riesgos indica si debe o no actualizar el PC.
- **memory_instruc.v:** Recibe el valor del PC que es la address para direccionar la memoria y obtener la instrucción que debe recuperar de los registros de la memoria de instrucciones. También escribe las instrucciones que son recibidas por la **Debug Unit** mediante el código Python y enviadas a este módulo.
- **IFID.v:** Es el latch que recibe la instrucción de la memoria de instrucciones y el nuevo valor del PC para la siguiente instrucción. Este latch envía estos datos a la etapa siguiente de decodificación. En caso de existir un riesgo y existir una burbuja mantendrá el valor que tenía en el clock anterior.

ETAPA ID

Como se mencionó, la instrucción se ejecuta dentro de un pipeline segmentado por etapas y luego de la primera etapa anterior continúa la instrucción en la etapa de decodificación.
En esta etapa se decodifica la instrucción separando sus diferentes campos. La decodificación sirve para identificar qué tipo de instrucción se trata, y hemos visto que pueden ser de distinto tipo y contener distintas partes que nos permitirán ejecutar esa instrucción. Además con la decodificación podremos generar la señales de control mencionadas anteriormente para el pipeline y las etapas siguientes y poder atender de manera adecuada los riesgos que puedan aparecer entre instrucciones y así garantizar una ejecución correcta y exitosa.
Como se ve en la siguiente imagen dependiendo la instrucción obtendremos las partes que me permitirán direccionar el banco de registro y obtener los registros operando como también obtener desplazamientos para instrucciones específicas como también el inmediato para calcular una nueva dirección a la cual se desea acceder o la dirección que deseo realizar el salto. El código de operación de la instrucción indica qué tipo de instrucción es, y por tanto, qué tipo de operación se debe realizar. Si es necesario, se leen 1 o 2 operandos de los registros del banco de registro como también se calcula la extensión de signo.

En esta etapa fundamental es cuando se analiza la instrucción para generar las señales correspondientes de control como también el desglose de lo necesario para luego poder ejecutar la instrucción en la siguiente etapa.
Se adelanta el cálculo de la dirección de destino en caso de tratarse una instrucción de salto y pasar a la etapa IF de la siguiente instrucción el valor correcto.. Esto se puede hacer porque el valor del contador del programa ya se lo tiene y debido a que ya se identificó la instrucción de salto entonces, se puede lograr la suma antes que se salte como también de tratarse de un branch comparar los registros de manera anticipada. Y entonces, no se necesita pasar al otro ciclo, porque se logró obtener la dirección y la condición de salto para lograrlo. Así, el **Delay slot = 1 ciclo de reloj**. La instrucción siguiente al salto se ejecuta siempre.



Módulos etapa ID:

- **BankRegisters.v**: Se inicializan los registros necesarios para almacenar valores y que se encuentran dentro de este banco de registro. Este banco se direcciona con la dirección RS, RT decodificados de la instrucción en la etapa anterior y lee los registros correspondientes para presentar los datos correspondientes a la salida. También se encarga de recibir la dirección RD del registro donde se va a guardar el dato a escribir dependiendo la instrucción ejecutada.
- **sign_extensor.v:** extiende el signo de los operandos inmediatos, que son de 16 bits, a 32 bits. Esto va dar una salida que puede ser:
    - Extender a 32 con los 16 más significativos respetando el signo.
    - Extender a 32 con los 16 más significativos usando un cero.
    - Colocar los 16 bits como más significativos y agregar ceros en los 16 menos significativos

  Dependiendo de la instrucción decodificada y la señal de control generada se tomara la decisión de cómo extender el dato
- **PC_Jump.v**: se utiliza para calcular nuevas address para el PC en los casos que se de saltos en una instrucción JUMP dentro del pipelines del MIPS. Suma la dirección de salto en la instrucción con PC+4. De esta manera se obtiene la dirección al cual tenga que hacer el jump. La decisión de utilizar esta dirección nueva del PC se toma en el mux_pc.v. La señal de control indicará en la etapa IF de la instrucción siguiente a la actual que debe tomar este nuevo PC.
- **UnitRisk.v:** se encarga de *detectar los posibles riesgos de datos y estructurales* que se puedan presentar en la ejecución del programa. En su bloque combinacional,

está comparando constantemente los registros y una señal de control que van a ingresar en la etapa EX y en la etapa ID, y de esta manera dispara flags que advierten al resto de los módulos para accionar de determinada manera**.** Este módulo se encarga de detectar cuando hay riesgos y agregar burbujas cuando como los casos de una lectura después de una escritura para una instrucción de Load, y JALR.

- **UnitControl.v** Este módulo se encarga de recibir el Opcode de la instrucción y de esta manera generar las señales de control necesarias para viajar junto con la instrucción a través del resto del pipeline que permite activar o desactivar funcionalidades en los distintos módulos en base a la instrucción decodificada. Mediante esta unidad podremos enviar la señales necesarias para la ejecución correcta de las instrucciones y evitar ciertos riesgos que pueden aparecer entre las instrucciones dentro del pipeline de una ejecución no controlada.
- **mux_unit_risk.v :** Solo hará un switch del valor de las señales de control. Si hay una condición de riesgo se ponen a cero todas las señales de la Unit Control ya que se da en los casos donde debemos frenar el avance del pipeline.
- **sum_branch.v:** Realiza constantemente la suma de la dirección del signo extendido (<<2 o multiplicado por 4) con el valor de PC+4 que viene viajando en el pipeline desde la etapa IF. El resultado de esta ALU anticipa y que lo único que hace es sumar, para obtener la nueva dirección para el PC en instrucciones branch.
- **and_branch.v:** Anticipa la comparación de los operando de tratarse de una instrucción branch a partir de la señal de control generada. Si la instrucción a ejecutar es BEQ o BNE y la comparación cumple la condición. De esta manera se podrá indicar al PC qué valor tomar en la etapa IF anticipando el valor calculado en el módulo sum_branch.v
- **mux_pc.v:** Proporciona el próximo PC para alimentar la etapa IF. A partir de las señales de control selecciona el nuevo PC correspondiente puede ser el valor de PC+4 o el generado por el calculo anticipado de una instrucción jump o branch. De esta manera el retardo Delay slot es de solo una instrucción en instrucciones de salto.
- **IDEX.v**: 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 Unit Control a las siguientes etapas.

ETAPA EX

Luego de la etapa anterior en la cual codificamos la instrucción obteniendo el valor de los registros necesarios para realizar la ejecución de la instrucción como también haber generado las señales de control que me permitirán manejar riesgos dentro del pipeline y una ejecución correcta, en esta etapa el objetivo principal es mediante el opcode de la instrucción ejecutar en la ALU la operación de la instrucción, así como también para casos de instrucciones branch calcular el nuevo PC.

Las operaciones aritméticas se llevan a cabo en la etapa EX. En la implementación, dicha etapa está constituida principalmente por la ALU, y varios multiplexores controlados por la Unidad de Control y la Unidad de Corto Circuito.

Módulos etapa EX:

- **alu.v:** Está ALU lo que hace es realizar las operaciones incluidas en el set de instrucciones permitidas.
  Puede resolver instrucciones que sean entre 2 registros, instrucciones que contengan un shmt (desplazamiento). La salida de este módulo es el resultado correspondiente a la operación y si la operación da cero, son salidas de la ALU.
  Así se presenta la codificación del **funct** para las operaciones aritméticas o lógicas.

```
ADD = 4'b0000;
SUB = 4'b0001;
AND = 4'b0010;
OR  = 4'b0011;
XOR = 4'b0101;
SRA = 4'b1011;
SRL = 4'b1001;
NOR = 4'b0100;
SLL = 4'b1000;
SLT = 4'b0111;
```

- **mux_alu_datoB.v**: Este módulo que es un multiplexor se encarga de otorgar el **operando B** a la ALU. El valor que otorgue depende de la instrucción y la señal de control que indica si debe usar el inmediato o no. En caso de no tratarse del inmediato tiene que ser el dato que se encuentra en EX/MEM o MEM/WR según si hay que hacer corto circuito o será siempre el valor del registro RT. Es decir que el módulo de la Unidad de CortoCircuito indicará esto último mediante una señal que genera el cortocircuito que es input para este módulo.
- **mux_alu_datoA.v:** Este módulo que es un multiplexor se encarga de otorgar el **operando A.** El valor que otorgue depende si la instrucción usará el dato del registro RS o si el dato tiene que se el dato que se encuentra en EX/MEM o MEM/WR según si hay que hacer corto circuito. Es decir que el módulo de la Unidad de CortoCircuito se indicará esto último.
- **UnitCortocircuito.v:** Este módulo resuelve uno de los riesgos que pueden aparecer en el pipeline del MIPS. Esta unidad viene a resolver los **riesgos de datos** en el cual podemos tener situación de **lectura después de escritura. Este módulo permite realizar el forwarding.** Esta unidad de cortocircuito lo que hace es validar valor de registro debe usarse para los operando de la alu. Realiza una comparación con los valores resultados de EX/MEM y MEM/WB cuando se trata de instrucciones que escriben en registro rd y el valor que tengo para rs y rt. La salida del cortocircuito me permitirá evitar dependencias de datos en las instrucciones en el pipeline, seleccionando correctamente el operando para la alu.
- **mux_register_rtrd.v:** Este multiplexor nos elige según la instrucción y la señal de control el registro de destino que corresponda. El registro de destino podrá ser RD o RT dependiendo el tipo de instrucción.
- **Control_ALU.v:** Este módulo es importante ya que sirve para interpretar la instrucción e indicar al módulo de la ALU que operación debe resolver. Como también enviar el flag para indicar si la operación será una instrucción con un desplazamiento.

Mediante una señal de control de 2 bits podremos saber de qué tipo de instrucción se trata la instrucción a ejecutar:

- **00:** La introducción **accede a memoria** se ejecutará una suma.
- **01:** La instrucción es un **salto condicional** se ejecuta una resta.
- **10:** La instrucción es **tipo R** y se evalúa el **funct [5:0]** de la instrucción y se selecciona la operación específica.
- **11:** La instrucción utiliza el valor **inmediato** se evalúa el **op[31:27]**

- **EXMEM.v:** Este módulo que representa el latch de la etapa recibe los datos generados en la etapa execute y enviarlos a la etapa de memory access cuando hay un posedge en el clock. Además continúa en viaje las señales de control generadas en la etapa ID a las etapas siguientes.

ETAPA MEM

Luego de la etapa anterior en la cual ejecutamos la instrucción obteniendo un resultado por parte de la alu, en esta etapa de acceso a memoria se realiza cualquier acceso a la memoria de datos requerido por la instrucción actual, por lo que, para instrucciones de cargas (LOAD), cargaría un operando de la memoria y para las instrucciones de almacenamiento (STORE), almacenaría un operando en la memoria. Para todas las demás instrucciones, no haría nada. Para determinar las ubicaciones de esta memoria que serán leídas o escritas se utiliza el resultado de la ALU de la unidad de ejecución para direccionar.
Además en esta etapa se encuentra el módulo AND que administra el flag de salto de Branch hacia el PC.

Módulos etapa MEM:

- **memory_data.v:** este módulo implementa una memoria de datos que lee un dato en memoria y guarda un dato en memoria en base a el pulso del clock y señales de control. Haremos una lectura en base a la señal generada por la Unit Control cuando sea una instrucción LW (Load Word), LWU (Load Word Unsigned), LB (Load Byte), LBU (Load Byte Unsigned), LH (Load Halfword), LHU (Load Halfword Unsigned), LUI (Load Upper Inmediate)
  Haremos una escritura en base a la señal que se generó por la Unit Control cuando sea una instrucción SW (Store Word), SB (Store Byte), SH (Store Halfword).
  Este módulo se encarga de implementar constantemente un filtro según una señal de control para los valores a escribir en la Memoria de Datos. En base a señales de control guarda el dato entero, el primer byte o la mitad del dato. Para las instrucciones JAL, JALR y las tipo R se aplica el filtro para el word entero. En caso de ser una instrucción SB será un byte y en caso de SH medio word.
  .
- **MEMWB.v**: Este módulo representa el latch de la etapa y se encarga de recibir y entregar los datos generados en la etapa de memory access a la etapa de write back cuando hay un pulso positivo de clock. Además envía las señales de control generadas por la Unidad de Control.

ETAPA WB

Esta es la última etapa del pipeline en el cual de ser necesario escribe el resultado de la instrucción en un registro, en esta etapa se lleva adelante la escritura. Es aquí el momento en el cual se envía el valor nuevo al banco de Registros al registro afectado por la instrucción que acaba de finalizar su ejecución.

Módulos etapa WB:

- **mux_register_rd.v:** Evalúa y procesa las señales de control para determinar el dato que debe escribirse en el banco de registros al finalizar la instrucción.
  La señal de control *i_ctl_dataload_size* indica el tamaño del dato cuando es cargado desde la memoria o para instrucción tipo R o JALR, y en función de esta, el módulo realiza las siguientes operaciones para convertir el dato a un formato de 32 bits correspondiente:
  - **2'b00 (Dato completo de 32 bits):**
    - El dato se toma tal cual desde la memoria sin modificaciones. (Instrucciones tipo R o JALR)
  - **2'b01 (Dato de 1 byte - 8 bits):**
    - Si se requiere extensión con signo, el valor se extiende rellenando los bits más significativos con el valor del bit más significativo del byte cargado (extensión de signo).
    - Si se requiere extensión sin signo, se aplica la máscara 32'hFF, dejando los bits superiores en cero.
  - **2'b10 (Dato de 2 bytes - 16 bits):**
    - Similar al caso de 1 byte, si se requiere extensión con signo, el valor se extiende usando el bit más significativo de los 16 bits cargados.
    - Si se requiere extensión sin signo, se aplica la máscara 32'hFFFF, dejando los bits superiores en cero.

  El resultado de este procesamiento se almacena en el registro interno *reg_filtered_data.*
  Una vez procesado el tamaño del dato, el módulo decide si el dato que se escribirá proviene de una instrucción LUI o de la memoria:
  - Si la instrucción es LUI (indicado por la señal de control *i_memwb_lui* = 1), el valor que se escribirá será el dato inmediato extendido (*i_memwb_extension*).
  - En cualquier otro caso, se utiliza el dato procesado previamente (*reg_filtered_data*).

  El resultado de esta selección se guarda en el registro interno *reg_data_to_reg*.
  Finalmente, el módulo determina si el dato a escribir proviene de la memoria o de la ALU, dependiendo de la señal de control *i_memwb_mem_to_reg*:
  - Si *i_memwb_mem_to_reg* = 1 (dato desde la memoria) el dato que se escribirá será *reg_data_to_reg* (que ya incluye el dato de memoria procesado o el inmediato de LUI).
  - Si *i_memwb_mem_to_reg* = 0 (dato desde la ALU) el dato que se escribirá será el resultado de la ALU (*i_memwb_alu*).

  El dato seleccionado se asigna a la salida *o_data_write*.

Salidas WB

o_wb_data_write: Aquí se selecciona el valor que se escribirá finalmente en el banco de registros utilizando un multiplexor basado en la señal i_memwb_jal. En caso de que sea un JAL el valor que se debe escribir en el banco de registros es el PC+8 y esto es así debido a que en JAL el procesador guarda el valor del PC+8 en el registro de retorno (registro 31) para que el programa pueda volver a la dirección siguiente al salto. En caso de que no sea un JAL el dato que se escribe es el obtenido en el módulo mux_register_rd. Este valor puede provenir de la memoria, de la ALU o de un inmediato extendido.

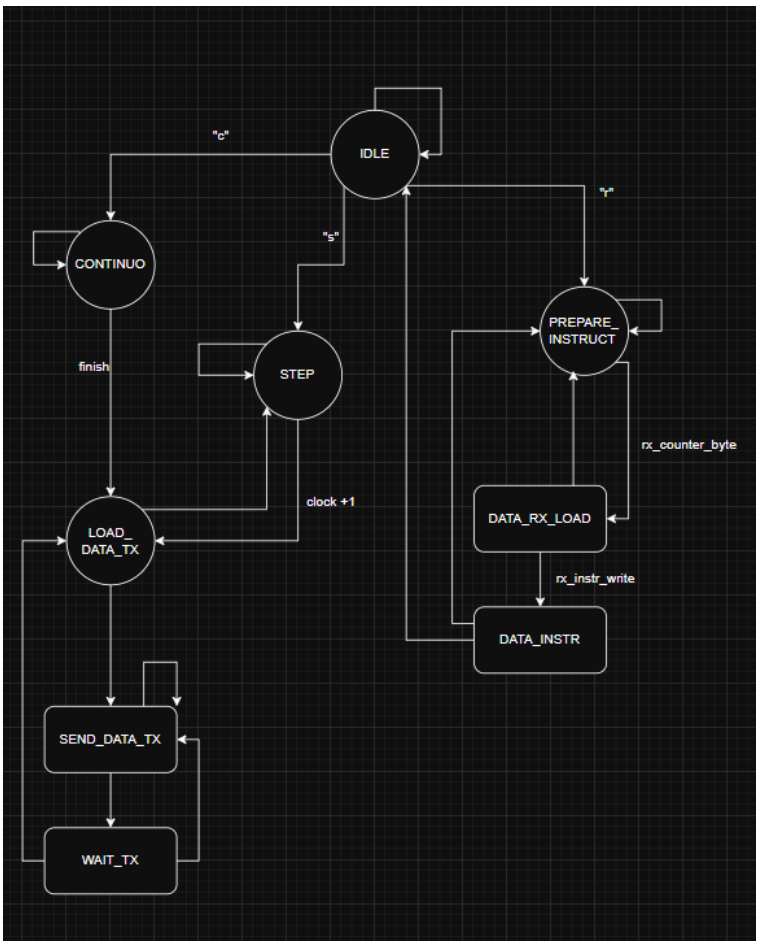
o_wb_register_adrr_result: Aquí se selecciona el registro de destino en el banco de registros utilizando un multiplexor basado en la señal i_memwb_jal. En caso de que sea un JAL el registro de destino es el registro 31 (5'b11111). Si no es un JAL, el registro de destino será el que se seleccionó en la etapa EX (RT o RD) dependiendo el tipo de instrucción.

o_wb_data_write_ex: Contiene el valor generado por el módulo mux_register_rd.

UNIT DEBUG

Este módulo es uno de los más importantes ya que nos permite enviar datos desde la pc a nuestro MIPS y obtener en nuestra consola información de la ejecución y registros. Desde la PC voy a poder enviar las instrucciones que se cargan inicialmente en la memoria de instrucciones.

Es por eso que la comunicación se realiza mediante UART como la aprendida en el trabajo práctico anterior. Mediante este módulo y el módulo UART establecemos una comunicación bidireccional que consta de diferentes etapas ya que se desarrolla siguiendo el concepto de máquinas de estados.



IDLE: Es el estado por default e inicia de nuestra máquina de estado para debuggear el funcionamiento del procesador y programas. Aquí la Unidad de Debug queda a la espera de recibir un dato que le indique a qué estado debe avanzar, dependiendo de ello es cómo va actuar el MIPS.
A **modo inicial** se comienza con la etapa de carga de instrucciones traducidas previamente de **asm** a **bin**. Para comenzar la carga de instrucciones en la memoria este estado debe recibir un carácter. Dicho carácter es **"r"** (RUN) como binario que indique la carga de instrucciones De suceder esta acción, se asigna un nuevo estado denominado
PREPARE_INSTRUCT que como su nombre indica habilita la preparación para la carga de instrucciones proveniente del código ejecutando en nuestra GUI.
Otras opciones de transición para este estado es recibir el carácter en binario **"s"** para dirigirme al estado **STEP** y realizar la ejecución del programa en modo step by step. La última opción de transición será recibir el carácter **"c"** para dirigirse al estado **CONTINUO** para realizar la ejecución completa hasta el final del programa.

Carga de instrucciones states:

PREPARE_INSTRUCT: Aquí voy a ir recibiendo desde la interfaz UART las instrucciones a cargar en la memoria. Mediante la recepción de datos de 8 bits se irán formando las instrucciones del tamaño de 32 bits. Este estado se complementa con los siguientes: ***DATA_RX_LOAD*** que se encarga de ir chequeando mediante un contador de bytes si ya se tienen los 4 bytes (32 bits de la instrucción) para poder enviarlo al módulo de memoria de

instrucciones del MIPS o continuar con la recepción de bits que conforman los bytes de la instrucción desde la UART.
Si ya se recibieron los 4 bytes de la instrucción que estoy intentado cargar desde la PC se realiza una transición al estado ***DATA_INSTR*** donde se verifica si la instrucción corresponde a un *halt* siendo la última instrucción de carga para un programa y en este caso volvemos a un estado IDLE a la espera de un modo de ejecución del programa ya cargado o se incrementa a la posición siguiente a la cual vamos a colocar la instrucción en la memoria y se regresa a este estado de preparación de instrucciones.

Modo de ejecuciones states:

CONTINUO: Este estado corresponde a uno de los modos de ejecución de nuestro MIPS y sus instrucciones de programa. En este modo se ejecutan todas las instrucciones y cuando se ejecuta la instrucción halt se envía el estado final por TX mediante la transición al estado ***LOAD_DATA_TX*** con la información final del último ciclo ejecutado del programa.

STEP: En este modo nos encontramos en la otra posibilidad de ejecución de programas de nuestro MIPS el cual corresponde a una ejecución paso por paso, es aquí cuando se hace una ejecución avanzando ciclo a ciclo de reloj de manera controlada por el usuario.
Aquí ocurre en cada ciclo que se avanza una transición al estado ***LOAD_DATA_TX*** para enviar la información correspondiente a nuestro cliente del ciclo de reloj ejecutado. Para continuar con la ejecución step by step se debe volver a habilitar el clock y continuar es por eso que se espera en este estado luego de envío de la información correspondiente la recepción por UART de otro carácter “n”(NEXT) que indica la habilitación para avanzar un nuevo ciclo para el siguiente step.
Cuando se ejecute un halt podremos recibir como entrada esta información en la Unit Debug y en este estado estaremos verificando si esto ocurre para que en el momento que suceda esta instrucción la transición sea al estado ***IDLE*** indicando la finalización de ejecución del programa en el MIPS.

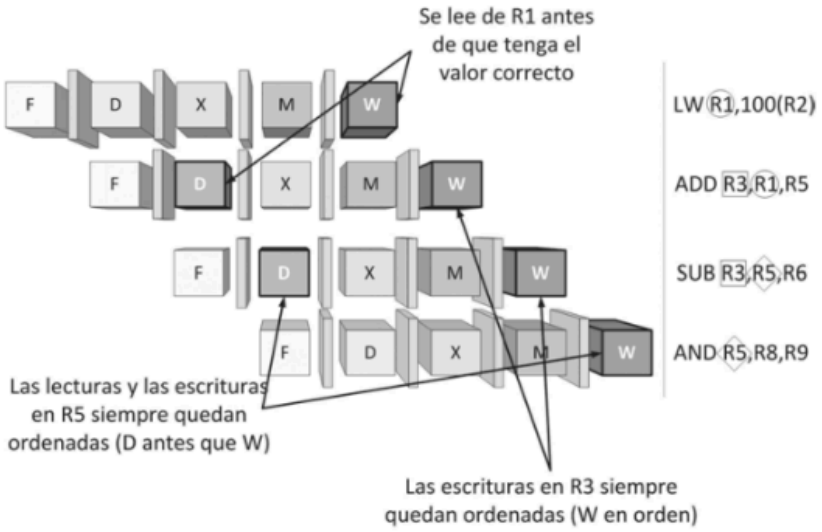
Transmisión de información a PC states:

LOAD_DATA_TX: Según el valor que se irá incrementando de un contador para indicar el case correspondiente para la data que deseo transmitir se hará la carga de la información para enviar (pc, registros, memoria ciclos, latches). Es en este estado donde se arma una variable de 32 bits y que contendrá información valiosa para el cliente en la PC. Durante este estado es que se envía información de todos los registros, la memoria de datos, el PC, clock, e información de registros importantes en los latches. Todos estos datos son de 32 bits y son recibos como entrada la Unit Debug brindado por el MIPS en sus distintas etapas.
En cada case definido en este estado ocurre la transición al estado ***SEND_DATA_TX*** que hará la transmisión de la información recibida por UART para el cliente. Como su nombre de estado lo indica es aquí donde se tiene los bits a enviar a la entrada del tx del UART. También se tiene un contador, que permitirá enviar de a 8 bits. Los datos de entrada brindados por MIPS de sus etapas son de 32 bits y son segmentados en 4 bytes con esta lógica de envío se tiene una transición posible a ***WAIT_TX*** que irá controlando el envío de los 32 bits del dato ya que se irá verificando si se enviaron todas las palabras, ósea los 32 bits recibidos en la Unidad de Debug por el MIPS y que sera una salida del módulo para la entrada de UART TX.

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.

Riesgos de Datos

Los riesgos de datos se producen cuando dos o más instrucciones presentan dependencias de datos entre sí, es decir utilizan los mismos recursos y la modificación de uno genera riesgos en las futuras instrucciones ya que, si no se resuelven correctamente, podrían llevar a la obtención de resultados erróneos en la ejecución del código.
Existen tres tipos de riesgos de datos, según el tipo de dependencia que los provoque:
- **LDE** (Lectura después de Escritura)
- **EDL** (Escritura después de Lectura).
- **EDE** (Escritura después de Escritura).



En MIPS sólo pueden producirse riesgos **LDE**, ya que al pasar todas las instrucciones por el mismo número de etapas y terminar en orden, las dependencias **EDL** y **EDE** nunca provocan riesgos por el solo hecho de que la etapa de escritura de cada instrucción es la última etapa por lo cual se darán un ciclo de clock después y no simultáneamente es por eso que son resueltos separando los momentos en que se leen y escriben los registros.
Las dependencias **LDE** son las reales y se dan cuando una instrucción anterior escribe en un registro que la siguiente instrucción lee por que existe dependencia. Este riesgo se resuelve con la Unidad de Cortocircuito en la etapa de EX, logrando la **anticipación de resultado (forwarding) o realimentación (bypassing)** para anticipar el dato con el control apropiado y detectar estas dependencias que me permite anticipar que valores otorgar a los operando para la alu.
El otro problema que se resuelve con la **Unidad de Riesgos** es cuando se tiene una dependencia de una instrucción tipo LOAD en la carga un valor de memoria en un registro que se utiliza en la una instrucción Tipo R. En ese caso la anticipación por la Unidad de Corto Circuito no alcanza por que sigue existiendo una dependencia atrás en el tiempo ya que la instrucción LOAD disponibiliza el dato en la etapa MEM. Por lo tanto se debe esperar un ciclo agregando una **burbuja** hasta que esté listo el resultado. Si la instrucción situada

en la etapa ID se bloquea, entonces se debe bloquear lo que esté en IF ya que de no hacerse así, se perdía la instrucción buscada en memoria. Sería evitar que se cambie el registro PC y el registro de segmentación IF/ID. Se va a usar una burbuja, es decir, un ciclo para esperar el Load y recién ahí hacer la operación del cortocircuito.

Riesgos de control

Se producen cuando una instrucción puede cambiar el flujo de control del programa. Surge la necesidad de tomar una decisión de realizar un salto o no. Esta decisión ocurre cuando hay una instrucción **branch (salto condicional).** Cuando ocurre esta instrucción se debe tomar una decisión de una condición y calcular una dirección de destino de salto. En el caso del MIPS, la instrucción que provoca este tipo de riesgo es el **BEQ y NBEQ ,** ya que hasta la fase MEM no carga el valor adecuado para el PC en el modelo teorico, y se tiene una penalización de tres ciclos hasta calcular la condiciones

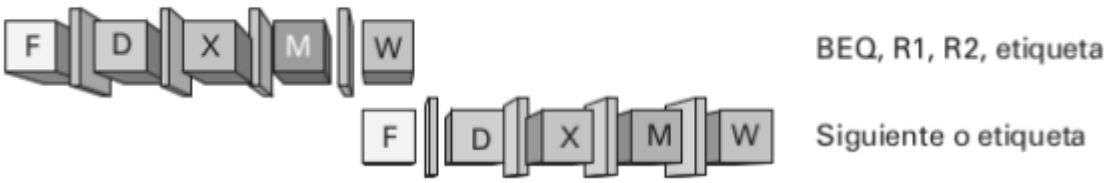


Figura Riesgo de Control

Solucion:

Mediante la **UnitControl** en la etapa ID podremos ver si la instrucción se trata de una instrucción BEQ o NBEQ y enviar una señal en la misma etapa y anticipar el cálculo y la validar la condición para indicar a la etapa IF de la siguiente instrucción que PC tomar. Lo único que se va a lograr es esperar un ciclo para la instrucción que siga al salto. Esto tiene una mejora del comportamiento de los saltos. Hay solo un ciclo de parada. Por lo que se tiene que analizar las instrucciones que están delante del salto y que no tengan dependencia de las operaciones después del salto

Riesgos de estructurales

Se producen cuando dos o más instrucciones necesitan utilizar el mismo recurso hardware al mismo tiempo. Por ejemplo sucede este tipo de riesgo si no se separaran las memorias de instrucciones y datos, ya que diferentes instrucciones intentarían 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)

La solución a este riesgo tiene un costo de hardware ahora se tienen 2 memorias, la de instrucciones y la de datos. Por lo tanto, en la etapa de búsqueda, se puede buscar las instrucciones mientras se está leyendo o escribiendo los datos de los resultados obtenidos de alguna otra operación referida a otra instrucción. Con estas unidades adicionales, es decir, agregando hardware se dio solución a los riesgos estructurales.

Analisis de Frecuencia

Frecuencia del clock de sistema

Para el sistema utilizamos un clock del clock wizard de IP-Core, que tiene una frecuencia de input de 100 MHz y alimenta al sistema con una frecuencia de salida de 69 MHz (periodo 14.49275 ns). Escogimos esta frecuencia de clock ya que según lo investigado (*https://support.xilinx.com/s/article/57304?language=en_US*) la forma de encontrar la maxima frecuencia posible es buscando un clock objetivo que en el reporte de timing de un WNS (*worst negative slack*) menor a 0. Una vez obtenido, en nuestro caso fue de 70 MHz, se realiza el siguiente calculo: 1T−WNS con T = 14.286ns y WNS = -0.086ns. Esto nos da como resultado una frecuencia maxima de 69.579738 MHz. Luego, fuimos afinando hasta obtener el clock que nos de WNS, WHS y WPWS mayores a 0 y TNS, THS y TPWS iguales a 0 que según el manual *UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs* deben tener dichos valores. Este clock obtenido fue el de 69 MHz.

50MHz

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2,406 ns	Worst Hold Slack (WHS):	0,039 ns	Worst Pulse Width Slack (WPWS):	3,000 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4492	Total Number of Endpoints:	4492	Total Number of Endpoints:	1930

All user specified timing constraints are met.

70MHz

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,733 ns	Worst Hold Slack (WHS):	0,007 ns	Worst Pulse Width Slack (WPWS):	2,633 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4492	Total Number of Endpoints:	4492	Total Number of Endpoints:	1930

All user specified timing constraints are met.

80MHz

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,152 ns	Worst Hold Slack (WHS):	0,020 ns	Worst Pulse Width Slack (WPWS):	2,633 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4555	Total Number of Endpoints:	4555	Total Number of Endpoints:	1962

All user specified timing constraints are met.

85MHz

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0,093 ns	Worst Hold Slack (WHS):	0,057 ns	Worst Pulse Width Slack (WPWS):	3,000 ns
Total Negative Slack (TNS):	-0,111 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	3	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4555	Total Number of Endpoints:	4555	Total Number of Endpoints:	1962

Timing constraints are not met.

Por el calculo ultimo da MAX 84MHz

Name	Slack ^1	Levels	High Fanout	From	To
Path 1	-0.093	1	1	module_TOP_MI...14_14/SP/CLK	module_TOP_MIPS/module_memwb/reg_dato_mem_reg[14]/D
Path 2	-0.009	1	1	module_TOP_MI...25_25/SP/CLK	module_TOP_MIPS/module_memwb/reg_dato_mem_reg[25]/D
Path 3	-0.008	1	1	module_TOP_MI...22_22/SP/CLK	module_TOP_MIPS/module_memwb/reg_dato_mem_reg[22]/D
Path 4	0.003	1	1	module_TOP_MI...26_26/SP/CLK	module_TOP_MIPS/module_memwb/reg_dato_mem_reg[26]/D
Path 5	0.024	2	1	module_TOP_MI...26_26/DP/CLK	module_Debug/tx_data_32_reg[26]/D
Path 6	0.029	2	1	module_TOP_MI...22_22/DP/CLK	module_Debug/tx_data_32_reg[22]/D
Path 7	0.069	1	1	module_TOP_MI...23_23/SP/CLK	module_TOP_MIPS/module_memwb/reg_dato_mem_reg[23]/D