Instituto Tecnológico de Buenos Aires

22.15 - Electrónica V

Trabajo Práctico N°2

EV20 - Unidad Central de Procesamiento

Grupo 1:

Matías Agustín Larroque
Leg. 56597

Tomás Agustín González Orlando
Leg. 57090

Lucero Guadalupe Fernandez
Leg. 57485

Manuel Fernando Mollón
Leg. 58023

Ezequiel Vijande
Leg. 58057

Profesores:

Andrés Carlos Rodríguez
Pablo Wündes

Entregado: 9 de Junio de 2020

1 Características Generales

En este informe se presenta la implementación en Quartus del procesador EV20 ilustrado en la siguiente figura:

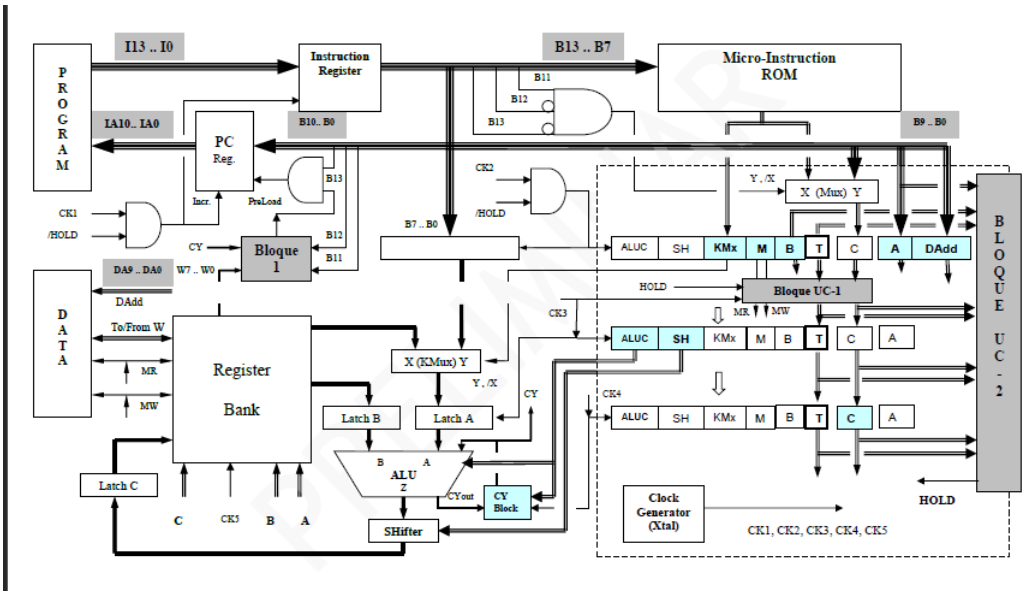


Figura 1: Esquema en bloques de la micro-arquitectura del Ev20

Nuestra implementación es una versión modificada del procesador presentado en la consigna. Las diferencias principales son:

- Registros y buses de datos de 16 bits
- Stack de 4 niveles para subrutinas
- Branch Prediction con memoria de los últimos 7 saltos
- Los saltos siempre se efectúan sin perder ciclos (Salvo que se haga una predicción incorrecta)

En las siguientes secciones del informe se detallaran los distintos módulos que integran al procesador en su conjunto

2 Interfaces

2.1 Set de Instrucciones

El primer modelo tomado de base para el diseño del EV20 cuenta con un set de 27 instrucciones de 14 bits que se pueden llamar "sencillas" ya que fueron ideadas para ser realizadas en un ciclo de máquina (o en un ciclo de pipeline). En el diseño final se mantuvo el número de instrucciones y la idea de su realización estilo "RISC", pero se modificó el tamaño de cada instrucción agregando 8 bits de manera de extender los operandos.

Para indicar de una mejor manera como fueron añadidos estos 8 bits, primero se muestra una imagen de algunas de las instrucciones del modelo de base:

1 0 0 x x x x x x x x x x x	JMP X
1 0 1 x x x x x x x x x x x	JZE X
1 1 0 x x x x x x x x x x x	JNE X
1 1 1 x x x x x x x x x x x	JCY X
0 1 0 0 y y y y y y y y y y	MOM Y,W
0 1 0 1 y y y y y y y y y y	MOM W,Y
0 1 1 0 i i i i i j j j j j	ADW Ri,Rj
0 1 1 1 s s s s s s s s s s	BSR S
0 0 1 0 i i i i i j j j j j	MOV Ri,Rj
0 0 1 0 1 1 1 1 i j j j j j	MOV POi,Rj
0 0 1 0 i i i i i 1 1 1 0 j	MOV Ri,PIj
0 0 1 0 1 1 1 1 i 1 1 1 0 j	MOV POi,PIj
0 0 1 1 i i i i i 0 0 0 0 0	MOV Ri,W
0 0 1 1 1 1 1 1 i 0 0 0 0 0	MOV POi,W

Figure 2: Fragmento de Set de Instrucciones - Modelo Base EV20

Se puede ver en la imagen, que el set de intrucciones presenta “expanding opcodes”. Por ejemplo, en las instrucciones de “JMP” (saltos) se interpreta un opcode de 3 bits con un operando de 11 bits, pero en las intrucciones de “MOV” (movimiento de registros) el opcode presenta 4 bits en lugar de 3. Aún más, si se mostrase el set de intrucciones completo podría notarse que el opcode llega a expandirse a 7 bits en algunas instrucciones.

Dicho esto, los 8 bits extra se añadieron como los 8 bits más significativos del operando, sin modificar el opcode. Este aumento de tamaño de las instrucciones permite trabajar con constantes de 16 bits en lugar de 8.

Se consideró que la solución planteada es la más eficiente teniendo en cuenta que se pretende mantener la implementación de las instrucciones con un estilo “RISC” (entonces se descarta la idea de tener que realizar más de un fetch para implementar una instrucción), y además pensando en un diseño escalable que permita expandir aún más los opcodes permitiendo agregar más instrucciones al set existente.

2.2 Memoria de datos y memoria de instrucciones

Para ingresar una instrucción al procesador y percibir sus resultados, son necesarias la memoria de instrucciones y la memoria de datos, implementadas por separado. A continuación se presenta una imagen de las memorias utilizadas para el diseño en “Quartus”:

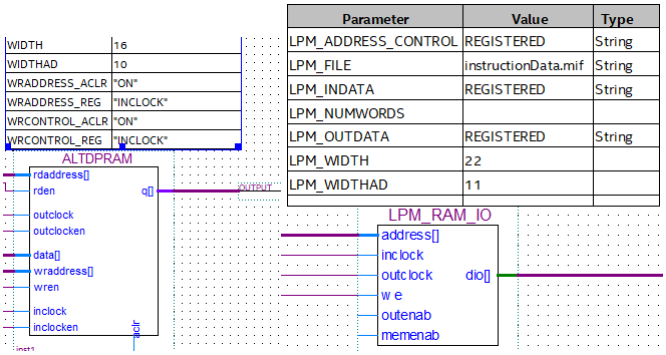


Figure 3: Memoria de Datos - Memoria de Instrucciones

Es importante notar que en los parámetros de Quartus para la memoria de instrucciones (ubicada a la derecha en la figura), se especifica un archivo “instructionData.mif”. Este archivo tiene extensión “.mif” que significa “Memory Initializer File”, y es utilizado para inicializar la memoria de instrucciones con un programa a ejecutar por el EV20, y así poder expresar resultados en la memoria de datos.

Para generar el archivo de instrucciones mencionado, se realizó un compilador que permite escribir programas de una manera relativamente “user friendly”, si es comparado con el desafío de escribir un progama en binario y respetando el formato un archivo “.mif”.

2.3 Compilador

El compilador permite interpretar un programa escrito en un editor de texto plano que permita guardar archivos con extensión “.txt”, utilizando los némonicos del set de instrucciones del EV20 (se mantuvieron los mismos némonicos del

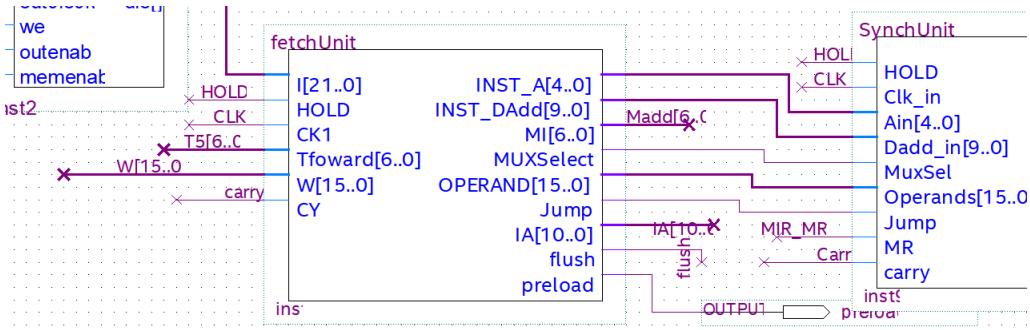


Figura 5: Micro Instruction ROM

set de instrucciones tomado como base para el diseño del EV20). La escritura del código debe ser secuencial, con una instrucción debajo de otra, sin identación, y se permiten ingresar comentarios luego de una doble barra "//".

A continuación se presenta una imagen de un código de input al compilador, y su respectiva salida como archivo ".mif", listo para ser compilado en Quartus:

```
MOK W,#10          //0
MOV R0,W
JMP 4
MOK W,#5
MOV R1,W   //el W debe finalizar en 10
BSR 10      //linea 5
MOK W,#1    //linea 6
CLR CY      //linea 7
CLR CY      //linea 8
JMP 8       //linea 9
MOK W,#15      //linea 10
RET
```

```
DEPTH = 13;
WIDTH = 22;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

0000 : 0001000000000000001010;
0001 : 0011000000000000000000;
0002 : 1000000000000000000100;
0003 : 0001000000000000000101;
0004 : 0011000000000000100000;
0005 : 0111000000000000001010;
0006 : 0001000000000000000001;
0007 : 0000001000000000000000;
0008 : 0000001000000000000000;
0009 : 1000000000000000001000;
000A : 0001000000000000001111;
000B : 0000011000000000000000;
000C : 1000000000000000001100;
END;
```

Figure 4: Input .txt - Output .mif - Compilador

Este compilador fue realizado en lenguaje "Python", lo cual lo hace capaz de incorporar funcionalidades extra las de permitir labels para los saltos o la declaración de subrutinas, sin la necesidad de un arduo esfuerzo en programación.

3 Unidades Funcionales

3.1 FETCH UNIT

3.1.1 Descripción General

Este bloque maneja la comunicacion entre el programa en ROM, la MIR y la Operand Unit. A continuación se puede ver el bloque en el main:

Y por dentro funciona de la siguiente manera:

TERMINAR, HABLAR SOBRE SYNCH UNIT.

3.1.2 Inputs

- I: Instruccion de 22 bits proveniente del programa alojado en la ROM.
- HOLD: Señal que suspende el avance del Fetch Unit si esta prendida.
- CK1: Clock de entrada.

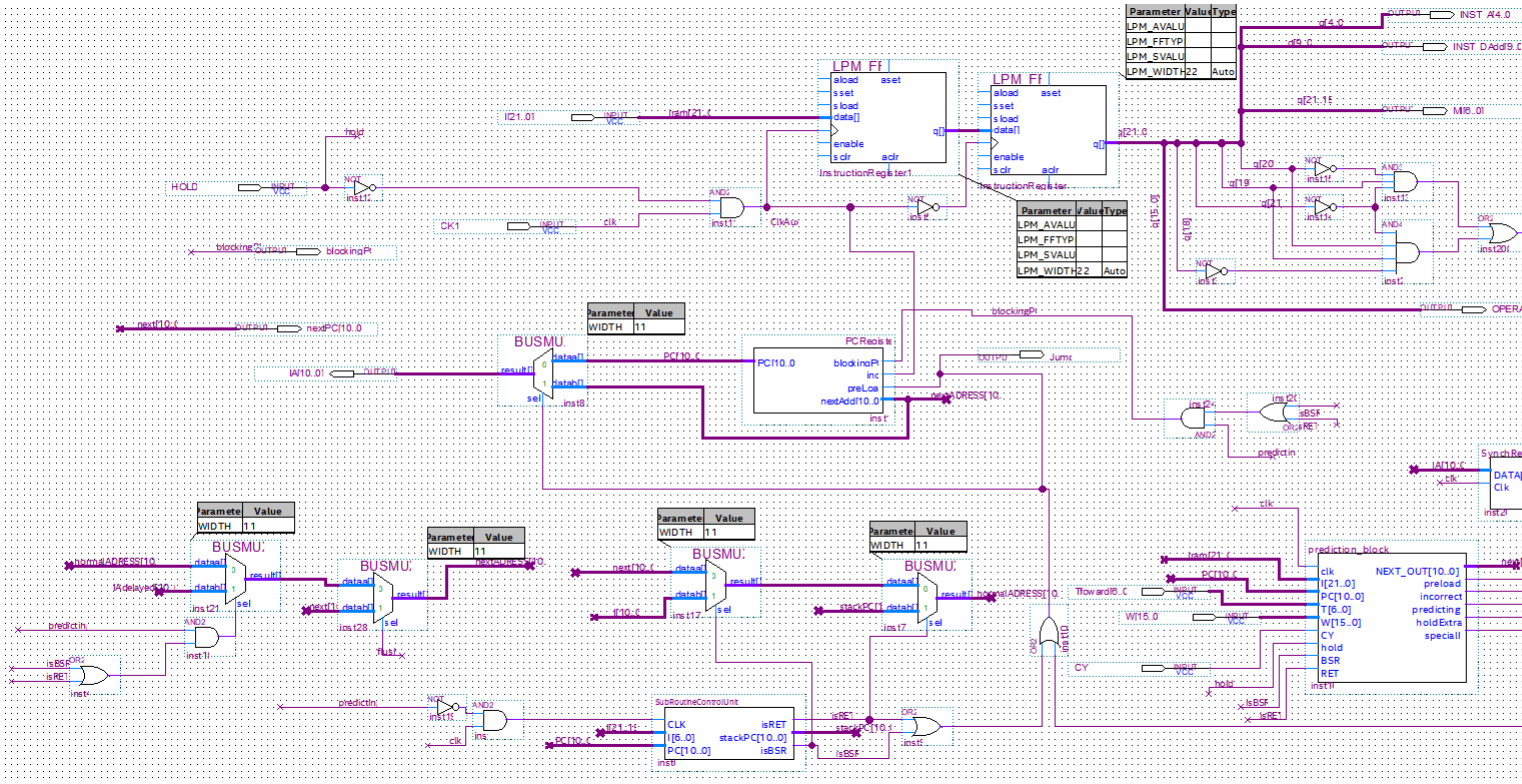


Figura 6: Micro Instruction ROM

- Tfoward: T del MIR de la etapa de Write back, utilizado para corroborar las predicciones de salto ya realizadas.
- W: Registro W de 16 bits provenientes del register bank.
- CY: Señal de carry proveniente del ALU unit.

3.1.3 Outputs

- INST_A: Parte de la instruccion que corresponde a la seleccion del bus A. (5 bits)
- INST_DAdd: Parte de la instruccion que corresponde a la direccion de memoria para instruciones de lectura y escritura de la misma. (10 bits)
- MI: Direccion de la micro instruccion para leerla de la ROM. (7 bits)
- MUXSelect: Selector de Mux para MIR, correspondiente a la seleccion del bus C.
- OPERAND: Operando que es leido por la Operand Unit. (16 bits)
- Jump: Señal que se prende cuando hay un salto en el programa
- IA: Direccion de la proxima instruccion del programa a ejecutar. (11 bits)
- flush: 1 en caso de que se necesite realizar el flush del pipeline (predicción de salto incorrecta). 0 en otro caso.
- preload: Señal de preload proveniente del prediction block dentro del fetch unit.

3.1.4 Funcionamiento detallado

La fetch Unit es la encargada de mantener internamente el STACK de saltos de subrutina, y de mantener el PC register actualizado con el valor de memoria a partir de la cual se obtendrá la próxima instrucción a ejecutar. Como internamente este módulo contiene al PC register, el fetch unit también contendrá al predictor de saltos, que actualizará el valor del

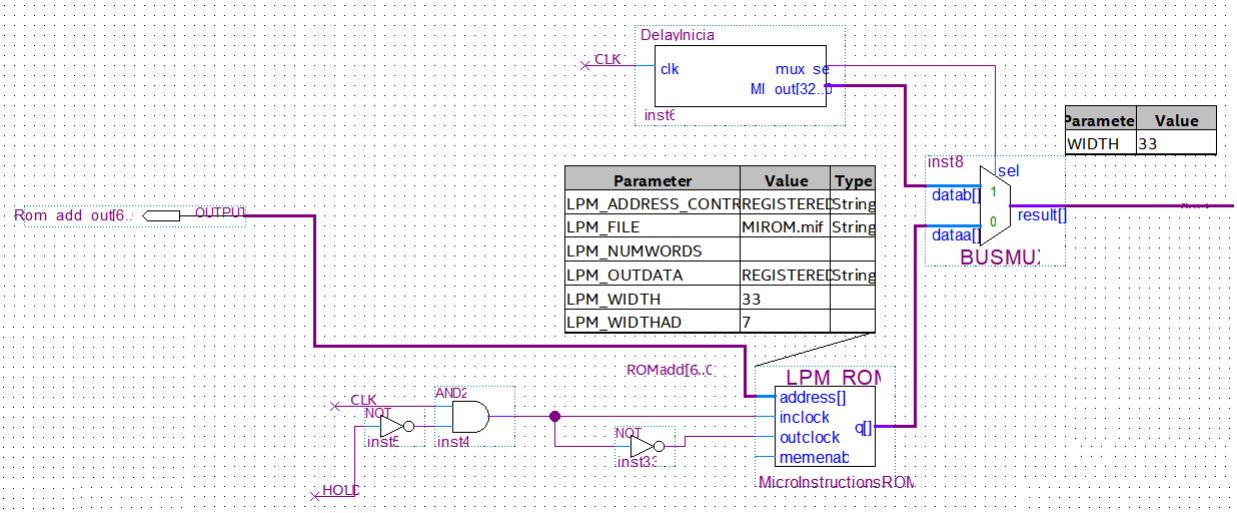


Figura 7: Micro Instruction ROM

PC en caso de que considere que debe realizarse un salto y en aquel caso en que el prediction_block detecte que se ha realizado una predicción de salto incorrecta.

Dado que el módulo de subrutinas impone el PC en caso de RET, el mismo deberá estar en comunicación con el módulo de branch prediction de alguna manera. Esta conexión se da a partir de la señal preload, que indicará que se deberá actualizar el PC con una nueva instrucción (proveniente del bloque predictor o del STACK) y unos muxes que eligen el valor de PC a actualizar.

El branch predictor deberá impedir que el módulo de subrutinas imponga PC en casos en que se haya realizado una predicción y se reciba un retorno. Esto se debe a que el módulo de branch_predictor no tiene control del stack suficiente como para reemplazar los contenidos del mismo en estos casos si hubo una predicción incorrecta.

El mux a la salida del PC register permite imponer una instrucción de fetch antes de que se actualice el PC register. Esto se debe a que la decisión de salto o no salto se debe tomar antes de un flanco positivo, para que el fetch se realice sobre esta nueva instrucción en dicho flanco. En flanco positivo también se actualizará el valor del PC. La señal de preload es la que se encargará de elegir la entrada del mux a elegir.

3.2 MICRO-INSTRUCTION ROM

3.2.1 Descripción General

Este bloque maneja las micro instrucciones que son procesadas por el pipeline. La dirección de cada micro instrucción esta dada por los 7 bits mas significativos del op code.

A continuación se muestra el bloque en el main:

3.2.2 Inputs

- ROM Address: Dirección de 7 bits a la cual buscar la micro instrucción. Para mantener la ortogonalidad, conjuntos de direcciones corresponden a la misma microinstrucción.
- Clk: Señal para, con Hold, hacer sincrónica la ROM.
- Hold: Señal para, con Clock, hacer sincrónica la ROM.

3.2.3 Outputs

- Microinstruction: A la salida, se obtiene la microinstruccion de 33 bits correspondiente a la dirección pedida. Esta misma en un principio corresponde a un NOP introducido por el DelayInicial, el cual es necesario para el correcto funcionamiento del pipeline en los primeros ciclos en que se arranca el programa.

```
DEPTH = 128;           % DEPTH is the number of addresses %
WIDTH = 33;            % WIDTH is the number of bits of data per word %

ADDRESS_RADIX = BIN;   % Address and value radixes are required %
DATA_RADIX = BIN;      % Enter BIN, DEC, HEX, OCT, or UNS; unless %
                       % otherwise specified, radixes = HEX %

-- Specify values for addresses, which can be single address or range
CONTENT
BEGIN
[1000000 .. 1001111]  :       0000000000000001000111000000000;      % JMP X %
[1010000 .. 1011111]  :       0000000000000001000111000000100000;      % JZE X %
[1100000 .. 1101111]  :       0000000000000001000111000000100000;      % JNE X %
[1110000 .. 1111111]  :       0000000000000001000111010000000000;      % JCY X %
[0100000 .. 0100111]  :       0000000010000001000111000000100000;      % MOM Y,W %
[0101000 .. 0101111]  :       0000000100000001000111000000010000;      % MOM W,Y %
[0110000 .. 0110111]  :       0101000001000100000000111010000000;      % ADW Ri,Rj %
[0111000 .. 0111111]  :       0000000000000001000111100000000000;      % BSR S %
[0010000 .. 0010111]  :       0000000000000000000000000110000000;      % MOV Ri,Rj - MOV POi,Rj - MOV Ri,POj %
[0011000 .. 0011111]  :       0001000001000100000000000100100000;      % MOV Ri,W - MOV POi,W %
[0001000 .. 0001001]  :       0000001000000001000100000001000000;      % MOK W,#K %
[0001010 .. 0001011]  :       0111001001000101000100000011100000;      % ANK W,#K %
[0001100 .. 0001101]  :       0110001001000101000100000011100000;      % ORK W,#K %
[0001110 .. 0001111]  :       0101001001000101000100111001100000;      % ADK W,#K %
0000100               :       0000000000000001000100000011000000;      % MOV W,Rj %
0000101               :       0111000001000101000100000011100000;      % ANR W,Rj %
0000110               :       0110000001000101000100000011100000;      % ORR W,Rj %
0000111               :       0101000001000101000100111011100000;      % ADR W,Rj %
0000000               :       0011000001000101000100000011100000;      % CPL W %
0000001               :       1011000000000001000111010000000000;      % CLR CY %
0000010               :       1100000000000001000111010000000000;      % SET CY %
0000011               :       0000000000000001000111000000000000;      % RET %
--
END;
```

Figura 8: MIROM.mif

3.2.4 MIROM.mif

Corresponde al archivo donde están plasmadas las micro instrucciones a continuación se pude apreciar la misma:
Template utilizado para la creación del mif: https://www.mil.ufl.edu/4712/docs/mif_help.pdf

3.3 ALU UNIT

3.3.1 Descripción General

Este bloque se encarga de llevar a cabo las operaciones de la ALU y se shifter, asi como tambien las de carry y el registro C:

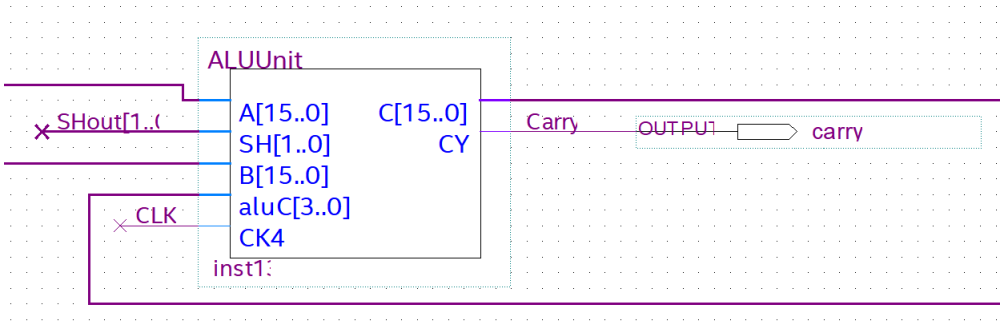


Figura 9: Micro Instruction ROM

Y por dentro funciona de la siguiente manera:

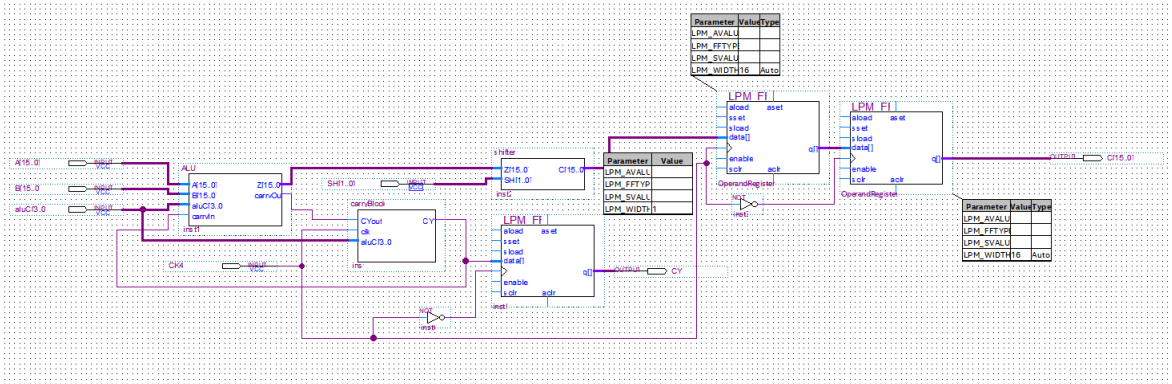


Figura 10: Micro Instruction ROM

Los datos provenientes de A y B son introducidos en la ALU junto con el selector de la operacion aluC. Se lleva a cabo asincronicametne la operacion de acuerdo a la siguietne tabla:

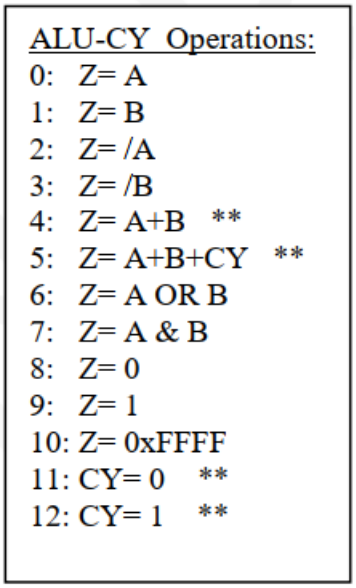


Figura 11: Tabla de operaciones de la ALU

De esta operacion sale el resutlado Z y el carry carryOut. El resultado es introducido en el shifter juunto al selector SH la cual indica que operacion de shift introducir en el resultado Z. La operacion de shifter corresponde a la siguiente tabla:

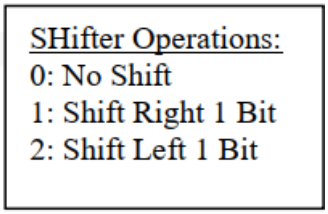


Figura 12: Tabla de operaciones del Shifter

La salida del shifter se almacena en el registro C. El cual usa dos flip flops para ser escrito en el flanco ascendente y leido en el descendente. Por otro lado el carry, el clock y el selector de la alu ingresna al bloque di carry, el cual decide si actualizar el carry out, el cual es guardado en un registro.

3.3.2 Inputs

- A: Operando de 16 bits que se introduce en la ALU.
- SH: Señal de 2 bits que indica que tipo de shift se introduce al resultado de la ALU.
- B: Operando de 16 bits que se introduce en la ALU.
- aluC: Seleccion de cual operacion realiza la ALU. (4 bits)
- Clk: Señal de clock

3.3.3 Outputs

- C: Salida del registro C de 16 bits.
- CY: Señal de Carry Out si esta es actualizada por la operacion realizada.

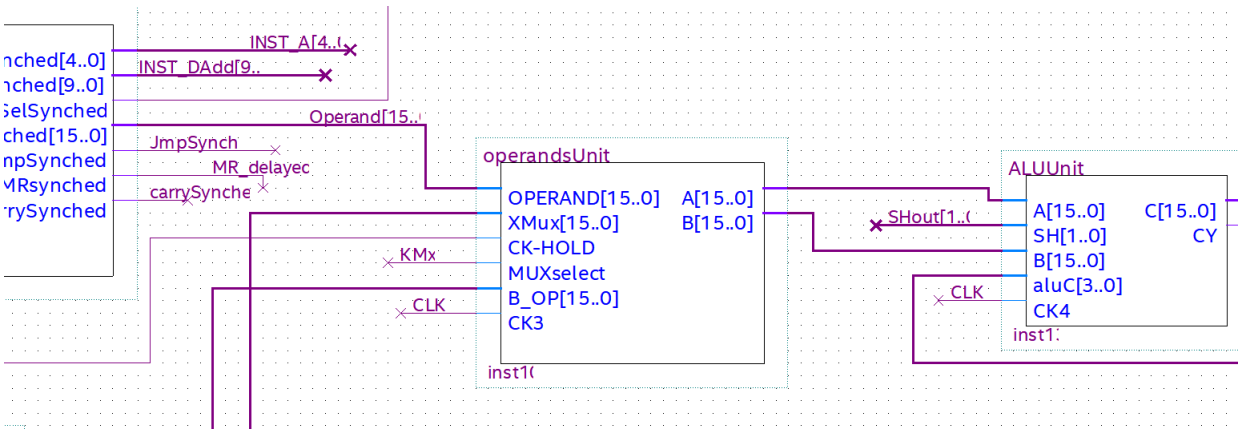


Figura 13: Micro Instruction ROM

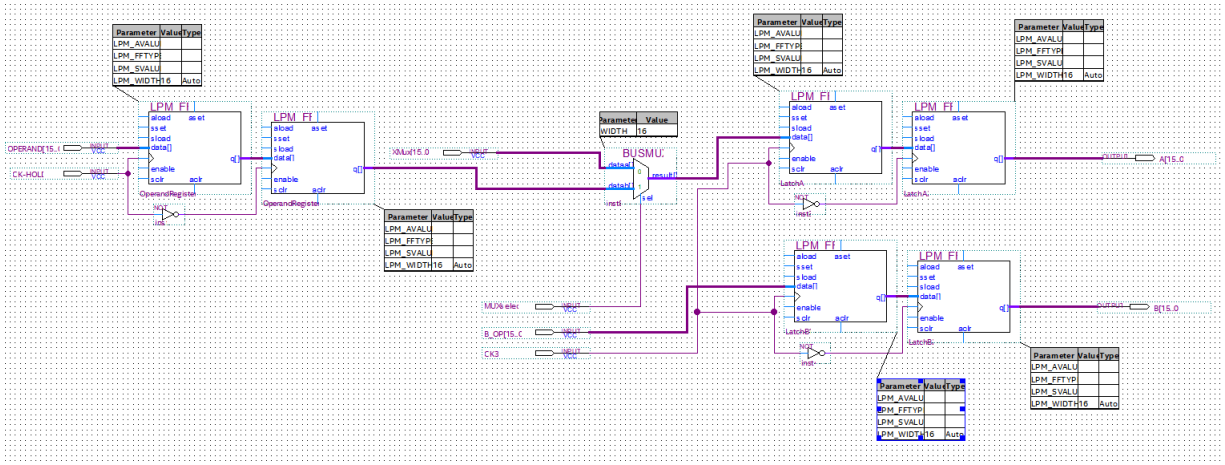


Figura 14: Micro Instruction ROM

3.4 OPERAND UNIT

3.4.1 Descripción General

Este bloque maneja la escritura y lectura de los registros A y B. A continuación se puede ver el bloque en el main:

Y por dentro funciona de la siguiente manera:

Donde se usan para los registros dos flip flops en cascada, de manera de ecribir el registro en el flanco ascendente y leerlo en el descendente. Lo que se escribe del registro A proviene de un MUX entre el dato proveniente de memoria y el opeando proveniente del register bank.

3.4.2 Inputs

- OPERAND: 16 bits de operando provenientes de la instruccion siendo fetcheada.
- XMux: 16 bits de operando provenientes del register bank
- CK-HOLD: Señal de clock con Hold.
- CK3: Señal de clock.
- MUXselect: Linea selectora del mux que escirbe el registro A.
- B_OP: 16 bits de operando provenientes del register bank.

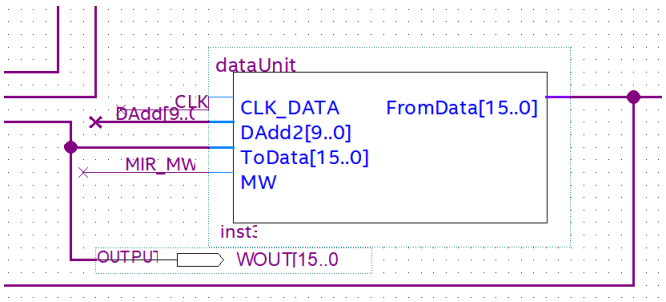


Figura 15: Micro Instruction ROM

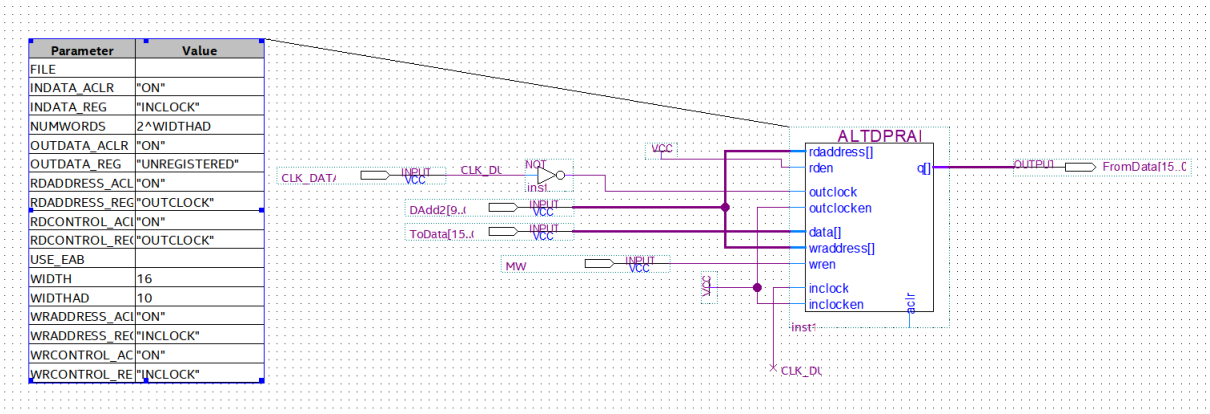


Figura 16: Micro Instruction ROM

3.4.3 Outputs

- A: Operando de 16 bits proveniente del Latch A.
- B: Operando de 16 bits proveniente del Latch B.

3.5 DATA UNIT

3.5.1 Descripción General

Este bloque maneja la escritura y lectura de datos en la RAM de datos. A continuación se puede ver el bloque en el main:

Y por dentro funciona de la siguiente manera:

Se puede apreciar que inlocken (input clock enable) y outclocken (out clock enable) ponen a Vcc habilitando siempre la actualización sincrónica de la RAM. El output clock proviene del Clk, y este mismo escribe lo que hay en memoria en el flanco ascendente. El input clock, es el Clk negado, de manera que se escribe en el flanco ascendente. Así se evitan problemas de lectura y escritura simultanea. Y wren (write enable) se prende solo cuando se quiere escribir (MW prendido) y escribe en memoria el dato introducido.

3.5.2 Inputs

- CLK_DATA: Clock con el que se sincrónica la lectura y escritura de la RAM.
- ToData: Data de 16 bits que se escribe en RAM cuando MW (Memory Write) esta habilitado.
- DAdd: Dirección de 10 bits en donde se leen o escriben los datos.
- MW: Señal que habilita la escritura de datos.

3.5.3 Outputs

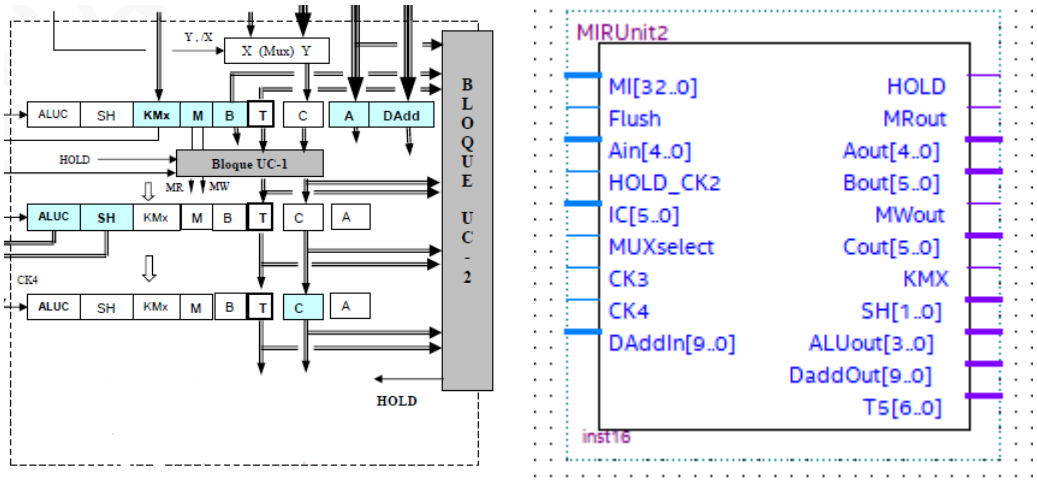
- FromData: Data de 16 bits que provienen de la memoria.

3.6 MIR UNIT

3.6.1 Introducción

Este bloque es el encargado de manejar el correcto avance del pipeline y la propagación de las señales de control al resto de los módulos. El termino MIR se usa como abreviación de "MicroInstruction Register" y el bloque tiene este nombre debido a los registros que utiliza para guardar las señales de control de cada etapa del pipeline.Sus tareas se pueden identificar como:

- Dividir las señales de control de cada etapa del pipeline desde decode hasta writeback.
- Detectar dependencias y generar la señal de Hold correspondiente
- Vaciar o hacer "flush" del pipeline en caso de ser necesario
- Propagar las señales de control en la etapa correspondiente a los demás módulos



(a) MIRUnit en la consigna del Ev20 (b) MirUnit en el esquemático de Quartus

Figura 17: Imagen del modulo presentado en la consigna y su equivalente en Quartus

3.6.2 Entradas y salidas

El modulo cuenta con 9 entradas y 11 salidas. Todas las entradas del modulo son leídas en los flancos positivos de clock (Salvo el flush que es asincronico) y todas las salidas son proporcionadas en los flancos descendentes de clock (salvo HOLD que es asincronica)

Las entradas son:

- MI (Microinstruccion de 32 bits proveniente de la ROM, su valor esta fijado por la instrucción que se desea ejecutar)
- Flush (Señal de 1 bit que se pone en alto cuando se desea "borrar" lo que se encuentra en el pipeline)
- Ain (Señal de 5 bits que indica que registro/puerto se desea mandar por el bus A)
- HOLD_CK2 (Señal de clock utilizada por el MIR correspondiente a la etapa de decode)
- IC (Señal de 6 bits que indica en que registro debe guardarse el resultado, solo se utiliza cuando MuxSelect esta en alto)
- MuxSelect (Señal de 1 bit que se pone en alto cuando debe utilizarse IC, si esta en bajo el registro destino se determina directamente de MI)
- CK3 (Señal de clock utilizada por el MIR correspondiente a la etapa de Operands)

- CK4 (Señal de clock utilizada por los MIR correspondientes a las etapas de Execute y de writeback)
- DAddIn (Señal de 11 bits que funciona como direccionamiento para acceder a la memoria de datos)

Mientras que las salidas son:

- HOLD (Señal de 1 bit que se pone en alto cuando se deben frenar las etapas de fetch y de decode)
- MRout (Señal de 1 bit que se pone en alto cuando se desea hacer una lectura de la memoria de datos)
- MWout (Igual que MRout pero para escritura)
- Aout (Señal de 5 bits que le indica al register bank que registro mandar al bus A)
- Bout (Señal de 6 bits que le indica al register bank que registro mandar al bus B)
- Cout (Señal de 6 bits que le indica al register bank en que registro guardar lo que llega por el bus C)
- KMX (Señal de 1 bit que se manda a un mux que decide si usar el operando del bus A o el que llega de la instrucción)
- SH (Señal de 2 bits que controla el shift register)
- ALUout (Señal de 4 bits que controla la ALU)
- DaddOut (Señal de 10 bits que indica la dirección a utilizar de la memoria de datos)
- T5 (Señal de 7 bits que indica el tipo de instrucción en la etapa de writeback)

3.6.3 Funcionamiento

- Descripción

El modulo funciona como un conjunto de 3 registros conectados en cascada, dichos registros sirven para separar las señales de control correspondientes a distintas etapas del pipeline. Ademas de estos registros, el modulo cuenta con un bloque de lógica combinacional que se encarga de detectar dependencias entre las etapas y en caso de encontrar una, frenar el avance del fetch y el decode mediante la señal de HOLD.

Finalmente, el modulo tiene una serie de bloques entre registros que se encarga de la mecánica de “Flush” al cambiar los valores de C a “35” y el T a “0” ya que esto equivale a convertir la instrucción en un NOP

- Módulos internos

El MIR unit esta compuesto por tres módulos internos. Estos son el MIR, el bloque UC-1 y el bloque UC-2.

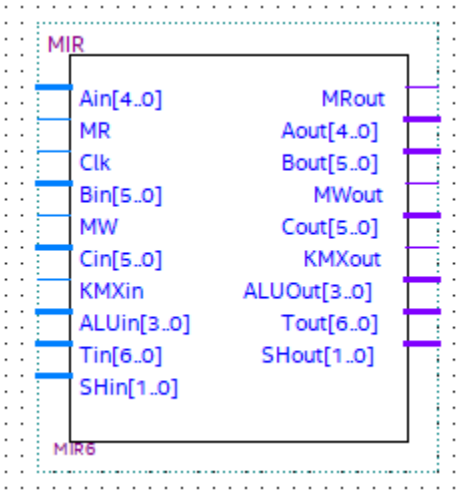


Figura 18: Esquema del MIR en Quartus

El MIR es simplemente un registro que lee sus entradas en los flancos positivos de clock y actualiza sus salidas en el flanco descendente. Internamente esta compuesto por flip flops para almacenar los valores de los distintos campos de la microinstruccion.

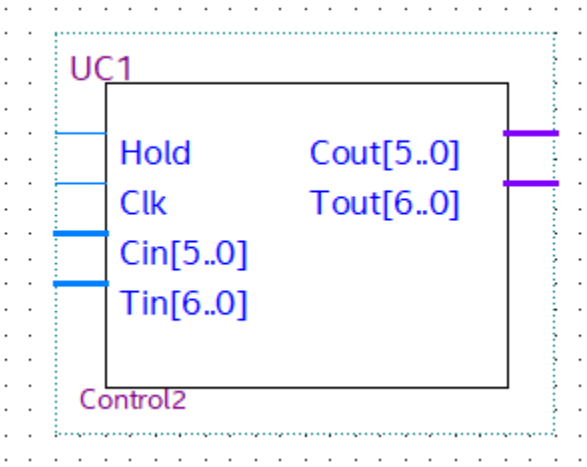


Figura 19: Esquema del bloque UC-1 en Quartus

El bloque UC-1 se comporta como un "cable" cuando la señal de HOLD esta en bajo.Esto quiere decir que cuando la señal de HOLD esta baja el bloque simplemente une con un cable Cout con Cin y Tout con Tin, esto se hace mediante un mux. Sin embargo, cuando la señal de Hold esta en alto, el bloque manda un NOP a la salida a partir del próximo flanco descendente de clock desde que se puso en alto el HOLD y permanece así hasta que el HOLD vuelva a bajar. Mandar un NOP a la salida quiere decir Poner el Cout en 35 (no se escribe en ningún registro) y poner el Tout en 0 (La instrucción no modifica nada).

Este bloque se utiliza para rellenar el pipelines con NOPS cuando se desea que siga avanzando el pipeline solo a desde la etapa de opeands en adelante. Asimismo, este bloque también se utilizo para implementar la mecánica de Flush, utilizando la señal de Flush que entra al MIR unit como señal de HOLD para los bloques UC-1 utilizados con dicho propósito.

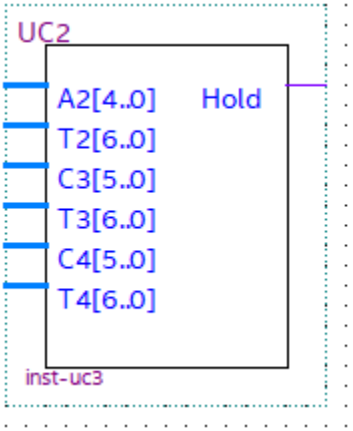


Figura 20: Esquema del bloque UC-2 en Quartus

El bloque UC-2 es el encargado de detectar dependencias entre las distintas etapas del pipeline y generar la señal de HOLD asincronicamente en caso de encontrar una. En principio existen tres tipos de dependencias posibles, RAW (Read After Write), WAR (Write After Read), WAW (Write After Write). Para nuestro procesador no es posible tener

dependencias del tipo WAR ya que la lectura se realiza en la etapa 3 del pipeline (Operands) y la única escritura que podría alterar la lectura seria un memory read (escribe el registro W) ya que se efectuá en la etapa 2.Sin embargo, no se genera una dependencia porque la lectura se realiza al comienzo del ciclo con el flanco positivo mientras que la escritura se realiza al final con el flanco descendente, por lo que no hay dependencia y no es necesario generar un HOLD.

Para el caso de las dependencias RAW, las mismas se pueden dividir en dos casos. El primer caso es cuando se desea leer un registro o puerto en la instrucción que se encuentra en la etapa 2 (Decode) y se desea escribir en ese mismo registro o puerto en una instrucción que se encuentra mas adelante del pipeline. Mientras que el segundo ocurre cuando se desea leer el registro W en la etapa 2 y una instrucción en una etapa posterior del pipeline desea escribir el mismo. Se distingue este caso del anterior ya que puede detectarse mas fácilmente utilizando el campo T.

Las dependencias WAW son un caso particular. Parecería que no hay dependencias del tipo WAW ya que las escrituras se realizan unicamente en la etapa 5 del pipeline.Sin embargo, esta el caso de Memory read el cual escribe en el registro W y se realiza en la etapa 2. Para asegurar que el valor leído en la etapa 2 no sea sobrescrito por un valor escrito en W en una de las etapas posteriores, es necesario generar un HOLD cuando la etapa 2 desea hacer un Write W y también la etapa 3 o la 4.

- Simulaciones

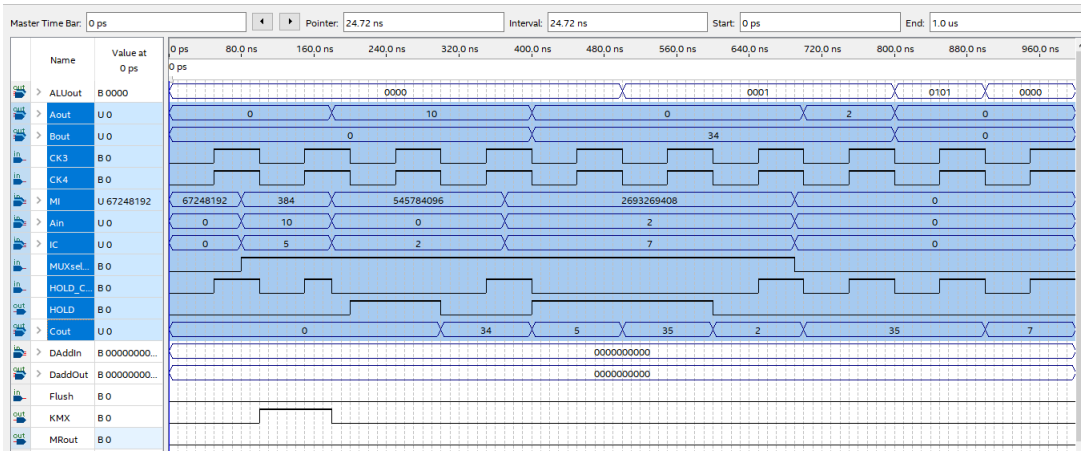


Figura 21: Simulación de MIR unit

La simulación anterior corresponde al siguiente código:

1. MOK W, #K
2. MOV R5, R10
3. MOV R2, W
4. ADW R7, R2

Como puede apreciarse correctamente del resultado de la simulación, el código anterior tiene dos dependencias.La primera dependencia es del tipo RAW entre la instrucción 1 y la instrucción 3, ya que el MOV necesita leer W pero el MOK todavía no escribió W.Cuando el MOV R2,W acaba de ser decodificado el MOK esta comenzando la etapa de execute, es por esto que se hace un HOLD por un ciclo hasta que el MOK llega al writeback y ya no hay mas una dependencia.

La segunda dependencia es entre la instrucción 3 y la 4, también del tipo RAW. La instrucción 4 necesita leer el registro 2 pero la instrucción 3 todavía no a escrito en el. En este caso las instrucciones están una inmediatamente después de la otra, por lo que se repite un caso similar al anterior pero la señal de HOLD permanece en alto por un ciclo adicional.

3.7 REGISTER BANK

En el banco de registros se implementaron los 28 registros de propósito general, los 2 puertos de entrada, 2 de salida, 2 registros auxiliares y el registro "W" o también llamado registro de trabajo. En total, en esta unidad se plasmaron 35

registros de 16 bits. Estos actualizan su entrada en cada flanco acendente del reloj de entrada, y actualizan su salida en cada flanco descendente, manteniendo consistencia con el resto del microprocesador.

El diseño de esta unidad contempla que solo puede ser escrito un registro por vez, y solo pueden ser leídos 2 registros por vez mediante dos buses de salida (“A” o “B”), pero se puede leer y escribir registros en el mismo ciclo de reloj.

Para escribir en alguno de los registros, no solo se cuenta con el input del nuevo contenido (bus “C”) del registro a escribir, si no que además se utiliza el bus de selección interpretado por un bloque decoder (efectivamente se implementó con un decoder de múltiples entradas y múltiples salidas). En cuanto a la lectura de un registro, también se cuenta con un bus de selección interpretado por un “bloque de salida”.

A continuación se presenta una imagen de una parte del esquemático para aportar a la descripción realizada:

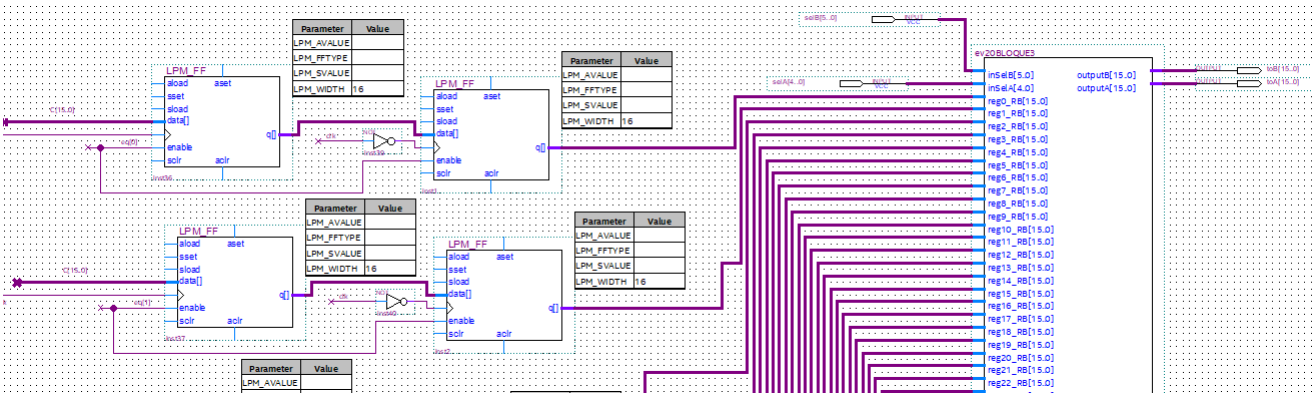


Figure 22: Fragmento de Esquemático Quartus - Banco de Registros

En cuanto al bloque de salida, también llamado en el proyecto como “BLOQUE 3”, este módulo presenta simpleza ya que fue implementado con compuertas tipo “MUX” para poder trasladar a los buses de salidas solo el contenido de los registros indicados por las líneas de selección:

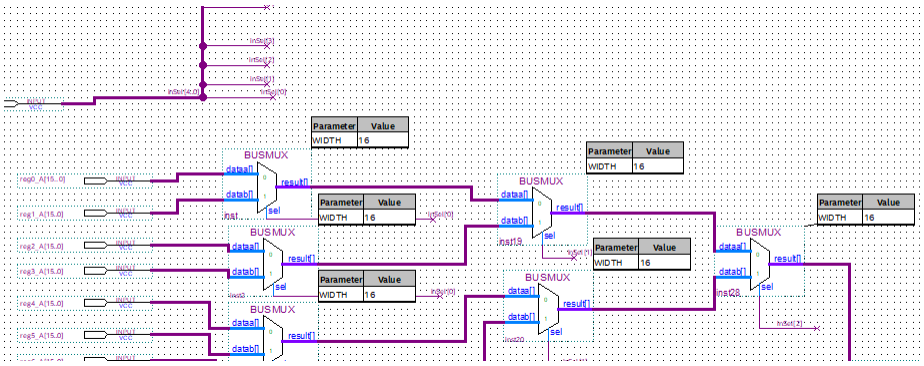


Figure 23: Fragmento de Esquemático- Muxes Banco de Registros

Finalmente, el banco de registros permite una escritura o lectura directa al registro de trabajo “W” con el fin de utilizar este registro para lecturas y escrituras de memoria. En cuanto a la lectura de memoria, indicada mediante el input “MR” (lectura de memoria y escritura en W), el bloque asegura que no entren en conflicto la señal que proviene de memoria con la señal de input convencional “C”.

A continuación se muestra una imagen con las interfaces de quartus que presenta el banco de registros diseñado:

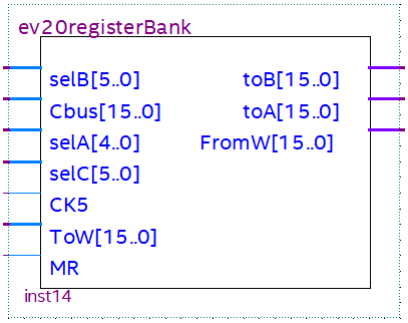


Figure 24: Interfaces del módulo- Banco de Registros

3.8 Stack

La base del módulo del stack se basa en un shift register bidireccional. La cantidad de niveles requerida define el tamaño del registro, en este caso 4, es decir que el MSB equivaldrá a 3.

El stack implementado con hardware específico tiene una estructura LIFO, *last in first out*. Los registros se inicializan en cero, cuando se pushea un valor al stack se shiftean los bits a la derecha, ingresando el valor por el bit más significativo (en la figura: MSB); mientras que al realizar la acción de pop, se shiftean a la izquierda, siendo la salida también por el bit más significativo.

Lo anterior mencionado se puede ver en la figura 25.

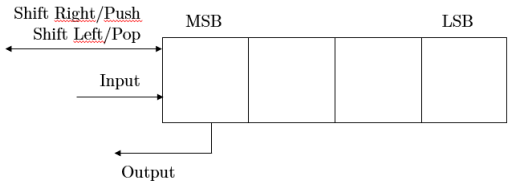


Figure 25: Shift Register Bidireccional.

La función del stack es guardar el valor del program counter register (PC), que es de 11 bits, por lo que necesitaríamos 11 shift registers para guardar cada uno de los bits y a su vez dar soporte para 4 subrutinas anidadas. El esquema del que se habla se puede ver en la figura 26, donde n equivale a 10, para un total de 11 bits.

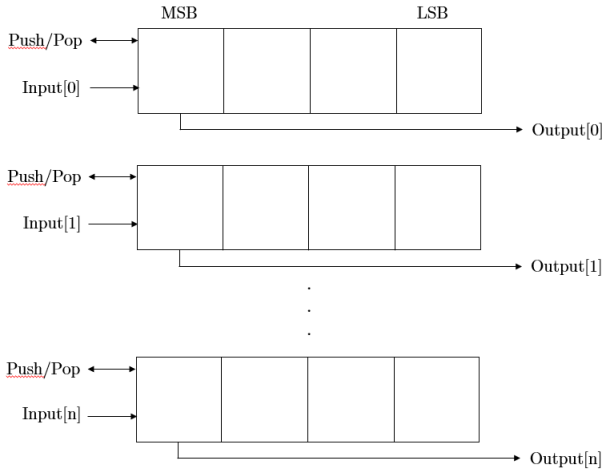


Figure 26: Estructura del stack.

La implementación del shift register se realizó en verilog, mientras que la estructura del stack fue con lógica combinacional. El módulo final se puede observar en la imagen 27. La idea es que el stack se actualiza de manera sincrónica, estando listo para ser leído el valor del TOS en cada flanco ascendente del clock, pero la salida, que es el TOS, se determina de manera asincrónica, entre un flanco descendente y uno ascendente del clock, ya que es cuando cambia la instrucción.

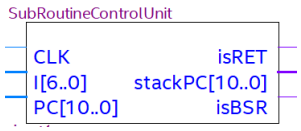


Figure 27: Entradas y salidas del módulo de control de subrutina.

En resumen entonces, el módulo de control de subrutina recibe los 7 bits más significativos del opcode para poder discernir si se trata de un BSR (*branch to subroutine*) o RET (*return to subroutine*), además recibe el PC, que es el valor a guardar, y el clock ya que funciona de manera sincrónica. Entre las salidas se encuentran las señales de isRET, e isBSR que están en *high* si se trata de un RET o de un BSR respectivamente, y estas señales se utilizan de manera externa al módulo para actualizar el valor del PC. También devuelve la variable stackPC que es el valor del TOS (*top of stack*).

3.9 BRANCH PREDICTION

3.9.1 Introducción

Un problema característico de las arquitecturas que cuentan con pipeline es la gran frecuencia de instrucciones de tipo salto/branch condicional que hay en los programas. Este tipo de instrucciones podría llegar a implicar un cambio en la dirección de memoria de la cual se extraerán las próximas instrucciones del programa en caso de que la condición a la que hacen referencia se cumpla, por lo que ante esta incertidumbre, el diseñador de la arquitectura deberá recurrir a alguna de las siguientes estrategias para poder continuar con el flujo del programa sin atorar el :

- Frenar el proceso de extracción de nuevas instrucciones de memoria hasta que se determine cuál es la nueva dirección de memoria de la próxima instrucción y luego continuar el flujo del programa. Se recurrirá en este caso a un atascamiento del pipeline cada vez que se reciba una instrucción de salto condicional, por lo que el procesador desaprovechará poder de cómputo que se aprovecharía en caso de que se utilizara alguna de las técnicas mencionadas a continuación.

- Asumir una política fija y determinada de antemano sobre cómo manejar los saltos, como por ejemplo, tomar los saltos hacia adelante, y luego verificar si se debió haber realizado el salto cuando se tenga información sobre el cumplimiento de la condición dada. En caso de que esta se cumpla, se preservarán los cambios y el flujo no será interrumpido, mientras que en el caso contrario se deberá volver a cargar el pipeline con las instrucciones que se debería haber llamado en primer lugar.

- Realizar una predicción sobre el salto basada en datos que obtienidos dinámicamente a lo largo de la ejecución de un programa particular, como por ejemplo, la frecuencia con la que se toman los saltos hacia adelante y hacia atrás, tomar el salto en caso de que se crea que se debería tomar y luego verificar si se ha realizado la predicción correcta, en cuyo caso.

- Permitir que el compilador agregue información sobre si el salto debería tomarse, y luego verificar si el mismo acertó en su predicción. En caso de que esto no sea así, se atascará el pipeline como mencionado en los casos anteriores.

El conjunto de técnicas que hacen referencia al hecho de predecir la próxima instrucción que se deberá ejecutar ante la presencia de un salto condicional son conocidas en inglés bajo el término de “branch prediction techniques”. En particular, para el diseño de la arquitectura del EV20, se decidió incorporar una estructura de branch prediction que prediga si un salto deberá tomarse o no a partir del resultado de las últimas N instrucciones de salto que han ocurrido durante la ejecución del programa, con N=7. Los detalles de implementación de dicho esquema de predicción serán presentados en las siguientes subsecciones.

3.9.2 Esquema de alto nivel del bloque

El bloque de predicción cuenta con las siguientes entradas:

- I[21..0]: Última instrucción extraída de memoria, sobre la cual se realizará un fetch de la micro-instruction ROM. Esta instrucción podrá ser de tipo salto condicional o no. De la misma se determinará el tipo de salto (sólo si la instrucción es de salto) que se podría llegar a realizar y la dirección de memoria de la próxima instrucción en caso de que el mismo sea tomado.
- PC[10..0]: Valor actual del PC.
- T[6..0]: Sección T del MIR de la etapa de Write Back del pipeline.
- W[15..0]: Valor del Working register en todo momento.

Las salidas del bloque de predicción son:

- nextOut[10..0]: Señal que indica la próxima dirección que deberá escribirse en el PC en caso de tomar el salto
- PreLoad: Señal que indica qeu deberá escribirse nextOut en el PC.
- incorrectPred: Señal que indica que se ha realizado una predicción incorrecta

El esquema permite manejar la predicción de instrucciones de salto anidadas sin tener que atorar el pipeline. El esquema interactúa a su vez con el STACK, pero deberá realizar HOLD del pipeline en casos de Saltos y retornos de subrutina.

3.9.3 Funcionamiento interno del módulo

El módulo de branch predictor cuenta con dos ramas principales:

1. Rama encargada de realizar la predicción de una instrucción de tipo salto.
2. Rama encarga de revisar si se ha realizado la predicción correcta de una instrucción de tipo salto una vez que ya se cuenta con el resultado de la condición de la misma.

El intercambio de información entre las dos ramas es realizado a través del uso de unas FIFOs comunes. En estas FIFOs se guardará:

1. El tipo de instruccion sobre la cual se ha realizado una predicción.
2. La predicción realizada sobre dicha instrucción.
3. La dirección de a la que se deberá retornar en caso de que la predicción realizada haya sido errónea.

Al realizarse una predicción incorrecta, las colas en cuestión serán vaciadas. En caso de que la predicción realizada haya sido correcta, se removerá un elemento de la cola. Es así como, para varias instrucciones de salto anidadas, la primer instrucción de salto sobre la cual se realizó una predicción es revisada primera, y en caso de que esta haya sido incorrecta, no se continúa revisando el resto de las instrucciones y directamente se realiza el flush del pipeline.

Los módulos internos del branch prediction están documentados dentro de los archivos .v correspondientes. Se considera que la documentación de los mismos y el esquema provisto en las subsecciones de este informe es lo suficientemente exhaustiva para que se pueda comprender el funcionamiento y la conexión de los mismos. Sin embargo, se adjunta la documentación de algunos de los módulos más importantes a modo de facilitar su comprensión:

Rama 1:

```
/*************************************
**************prediction_control*******
*************************************
```

- INPUT:

1. clk: clock signal
2. I: instruction (from RAM)
3. PC: Current value of PC register.

- OUTPUT:

1. enable: 1 if I is an instruction that requires prediction. 0 otherwise.

The instructions that require prediction are:

- JZE
- JNE
- JCY

1. unconditional: 1 if unconditional jump is required. 0 in other case.
2. next: resolves conflict for instructions that don t require prediction (except for BSR)

If I is an instruction that does not require prediction,

- a) next = PC if I is not JMP.
- b) next = I[10:0] if I is JMP.

If I is an instruction that requires prediction, next is undefined.

- TIME ANALYSIS (for input and for output):

1. Continuous: enable is continuously evaluated based on changes on I.

If changes on I happen with every negedge, then enable will also be updated after each negedge. */

Rama 2:

```
/************************************
 ************prediction_checker*********
 ************************************
```

- INPUT:

1. T: TYPE of the MIR that is currently on the execute step of the pipeline.
2. W: current working register.
3. pred_type: type of prediction to be checked (01 for JZE, 10 for JNE, XX for JCY)
4. CY: current value of carry.
5. last_pred: prediction to be checked (1 taken, 0 not taken).

- OUTPUT:

1. incorrect_pred: 1 if the prediction was incorrect. 0 Otherwise.
2. correct_pred: 1 if the prediction should have been "take the branch". 0 otherwise.
3. checked: 1 if the current instruction on the execute step of the pipeline involved previous prediction when fetched. 0 Otherwise.

3.9.4 Método de predicción

El algoritmo de predicción de salto elegido consiste en registrar las N últimas instrucciones de salto que han ocurrido durante la ejecución del programa, con $N = 7$. Para cada uno de estos N saltos, se registra si el salto debería haber sido tomado (T) o no (NT). Luego, se obtiene la cantidad de saltos de tipo T (C_T) y se los compara con la cantidad de saltos de tipo NT (C_{NT}). Si $C_T > C_{NT}$, entonces el próximo salto que se reciba será tomado, mientras que si $C_T < C_{NT}$, el próximo salto no deberá tomarse y se continuará con el normal flujo del programa. Nótese que $C_T \neq C_{NT}$ para $N = 2k + 1$,con $k \epsilon N$.

Se eligió $N = 7$ a para poder presentar el correcto funcionamiento del esquema en el contexto de una arquitectura con una capacidad máxima de stack de 4 llamados a subrutinas.

Esta técnica presenta dos ventajas que impulsaron a la decisión de la incorporación de la misma al diseño de la arquitectura:

1. La implementación de la misma resulta relativamente sencilla en comparación a otras técnicas de branch prediction.
2. La bibliografía de la cátedra presenta a este esquema como un esquema de predicción que obtuvo resultados empíricos exitosos, tal vez sorpresivamente por su algoritmo sencillo.

Para el caso en que el historial de saltos del momento tenga menos de N saltos registrados, el esquema asume que los datos faltantes fueron saltos de tipo T, de forma tal que al comienzo del programa se asume que todo salto encontrado debe ser tomado. Cuando se cuente con N saltos registrados y se reciba una nueva instrucción de tipo salto, se deberá eliminar del historial el registro más antiguo y reemplazar al mismo con el resultado (T o NT) del salto una vez que se haya verificado si la condición de salto se ha cumplido o no. Es decir, se prioriza información más reciente con respecto a información anterior a esta.

Nótese que para el caso de loops de muchas iteraciones y/o de secuencias de loops, el esquema mencionado debería tener un buen desempeño, salvo para algunos casos excepcionales como secuencias de loops de muy pocas iteraciones.

3.9.5 Predicción incorrecta

En caso de predicción incorrecta, se deberá volver al PC original. Esto se logra mediante una FIFO auxiliar que guarda los PCs a los que se debería volver si la predicción fue incorrecta.

En caso de tener una predicción incorrecta, se deberá realizar el flush del pipeline. Para esto, simplemente se reemplaza las instrucciones que están actualmente en el pipeline con un NOP.

Para el caso de escrituras y lecturas a memoria y BSR y RET, si se ha realizado una predicción, se deberá esperar a que la misma sea finalizada, por lo que se tendrá que realizar un HOLD sobre el pipeline hasta esperar a reconocer si se realizó una predicción correcta o no y luego decidir en función de lo mismo si se debe tomar el BSR o no.

3.9.6 Simulación

Se simuló el módulo por separado para distintos casos, con resultados satisfactorios. Los valores de los buses fueron elegidos manualmente para probar cada caso de interés

Descripción de las señales:

- clk: Señal de clock
- CY: Carry
- I: Instrucción recibida en etapa de Fetch.
- preload: 1 cuando el módulo indica que se ha recibido una instrucción de tipo salto.
- incorrect_pred: 1 cuando predicción incorrecta. Cero en cualquier otro caso.
- T: sección T del MIR en etapa de Write Back del pipeline.
- NEXT..: Señal que indica el próximo valor del PC.
- PC: Valor del PC en etapa de fetch.
- W: Working Register.

En la siguiente figura se muestra cómo se predice que se debe saltar para una instrucción de tipo JCY (preload=1) . Al recibir el T de la etapa de write back con valor 1010000 (JCY), el bloque reconoce que la predicción fue incorrecta y vuelve a cargar la dirección correcta en el PC (preload=1 e incorrect_pred =1 simultáneamente) . Esto se puede observar en el bus NEXT_ADDRESS, que recupera el valor que tenía el PC al momento de realizar la predicción.

El bloque reconoce que la predicción fue incorrecta porque CY = 0 en el momento en que se recibe el T previamente mencionado.

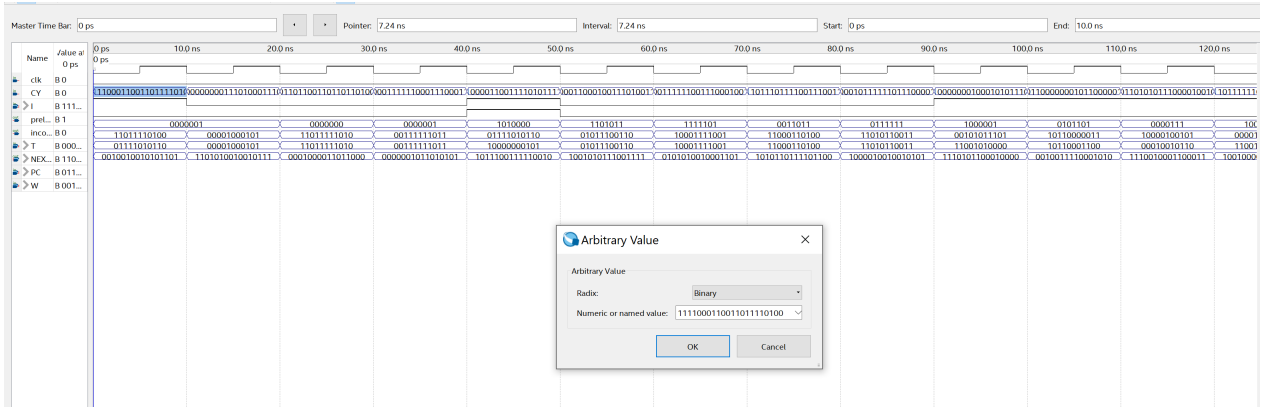


Figure 28: JCY simple. Predicción: T. Predijo mal.

En la siguiente figura se muestra cómo se predice que se debe saltar para una instrucción de tipo JNE (preload=1). Al recibir el T de la etapa de write back con valor 1000001 (JNE o JZE), el bloque reconoce que la predicción fue correcta y por lo tanto no modifica el valor de PC (preload=0 e incorrect_pred =0 simultáneamente) . Esto se puede observar en el bus NEXT_ADDRESS, que deja pasar el valor del PC actual sin hacer ninguna corrección.

El bloque reconoce que la predicción fue correcta porque W no vale 0 en el momento en que se recibe el T previamente mencionado (quinto ciclo de clock).

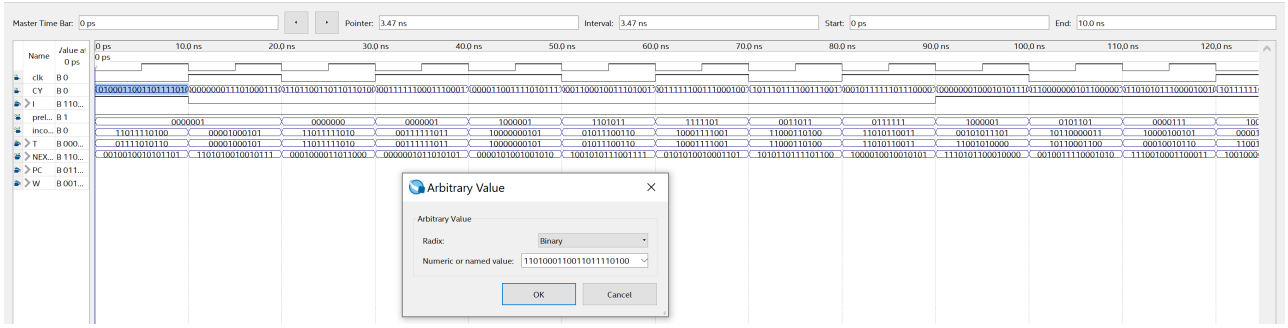


Figure 29: JNE simple. Predicción: T. Predijo bien.

En la siguiente figura se muestra cómo se predice que se debe saltar para una instrucción de tipo JZE seguida de una JCY(preload=1 para ambos ciclos consecutivos). Al recibir el T de la etapa de write back con valor 1000001 (JNE o JZE), el bloque reconoce que la predicción fue incorrecta y vuelve a cargar la dirección correcta en el PC (preload=1 e incorrect_pred =1 simultáneamente). Esto se puede observar en el bus NEXT_ADDRESS, que deja pasar el valor del PC actual sin hacer ninguna corrección. El bloque reconoce que la predicción fue correcta porque W[15] no vale 0 en el momento en que se recibe el T previamente mencionado (sexto ciclo de clock).

Se ve como, luego de reconocer una predicción previa incorrecta, ignora el siguiente T, pues este y los siguientes T del pipeline quedan invalidados, por lo que el JCY ya no es revisado.

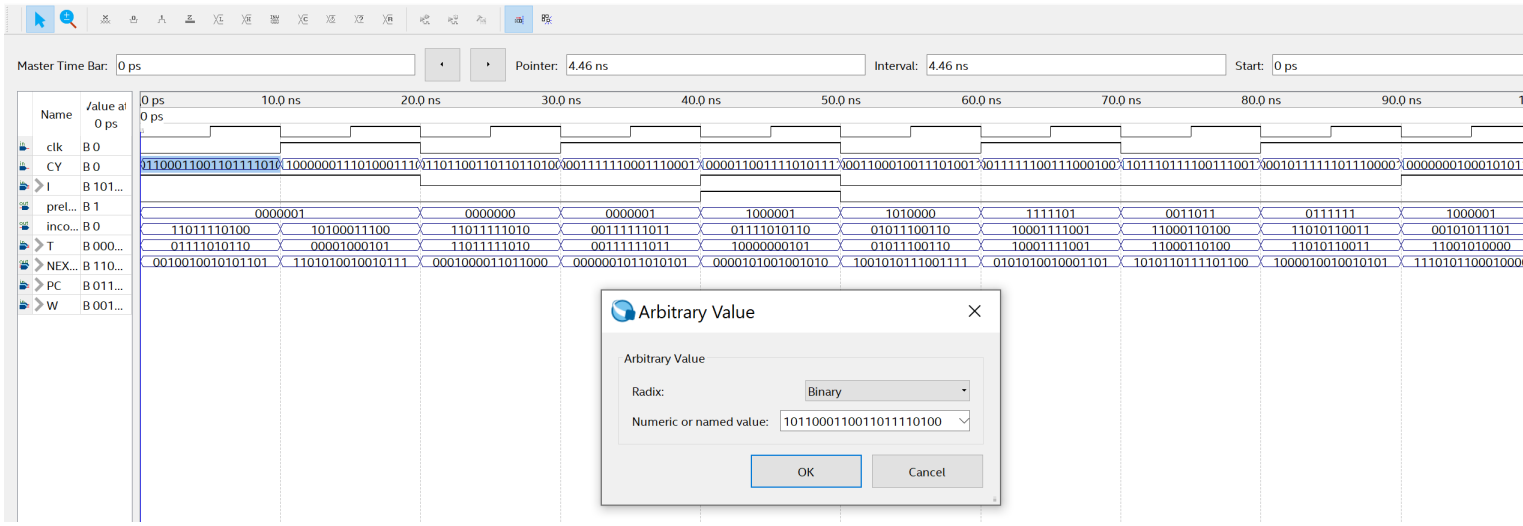


Figure 30: JZE + JCY, Predicción: T. Predijo mal JZE

Para la próxima figura, se realizan varias instrucciones de salto JCY intercaladas (instrucción dummy de por medio). El módulo tomará los saltos al principio, ya que el algoritmo de decisión tiene registrado que los últimos 7 saltos fueron tomados. El primer salto (T=1010000) encuentra el CY=0, por lo que incorrect_pred=1. Luego, el siguiente salto encuentra el CY = 1, por lo que incorrect_pred=0. El siguiente salto tendrá incorrect_pred=1 porque el algoritmo de predicción tiene memoria de los últimos N saltos, y no actualiza su algoritmo de decisión todavía.

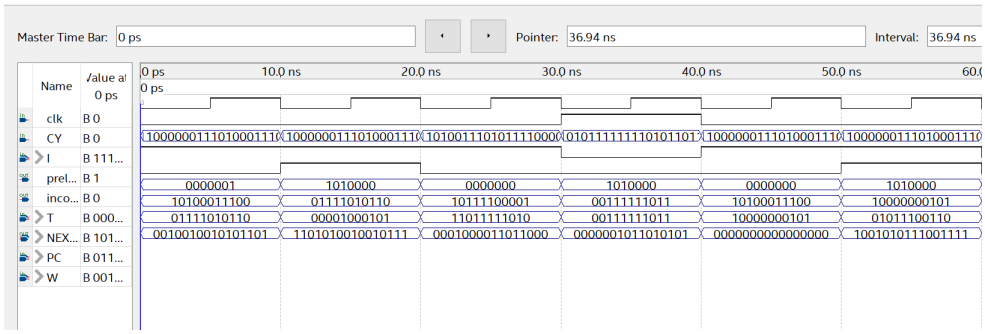


Figure 31: La predicción no se adapta porque el historial no tiene suficientes errores

4 Conclusión

La integración de los módulos pudo realizarse exitosamente y se verifico que el branch prediction funcione adecuadamente y que en caso de saltos incorrectos no se altere el estado de los registros o de la memoria de datos respecto al estado previo al salto. La mayor complejidad del procesador esta ubicada en el fetch unit, ya que es donde se encuentra el modulo de branch prediction y también el modulo encargado del stack para subrutinas. Dicho esto, el fetch unit fue el modulo que genero la mayor dificultad para integrar y lograr un correcto funcionamiento del mismo.

El resultado final fue acorde a lo esperado y se pudo garantizar el correcto avance del pipeline a su vez con la integración del branch prediction. Sin embargo, el diseño presenta mejoras posibles en como se implementaron los módulos asincronicos en la unidad de fetch para poder lograr utilizar un frecuencia de clock mayor todavía mayor.