INSTITUTO TECNOLÓGICO
DE
BUENOS AIRES

Ingeniería Electrónica

22.15 - Electrónica V

TRABAJO PRÁCTICO 2

EV21

Grupo 4:	*Legajos:*
Álvarez, Lisandro	57771
Fogg, Matías	56252
Dieguez, Manuel	56273
Martorell, Ariel	56209
Diaz, Ian Cruz	57515

Contenido

1. Introducción

1.1. Resumen

En el presente informe se detalla el desarrollo del procesador RISC EV21. cuya implementación está inspirada en el diseño de la Figura 1.1. El diseño se desarrollo en el software Quartus, para luego ser implementado en una FPGA Cyclone IV.

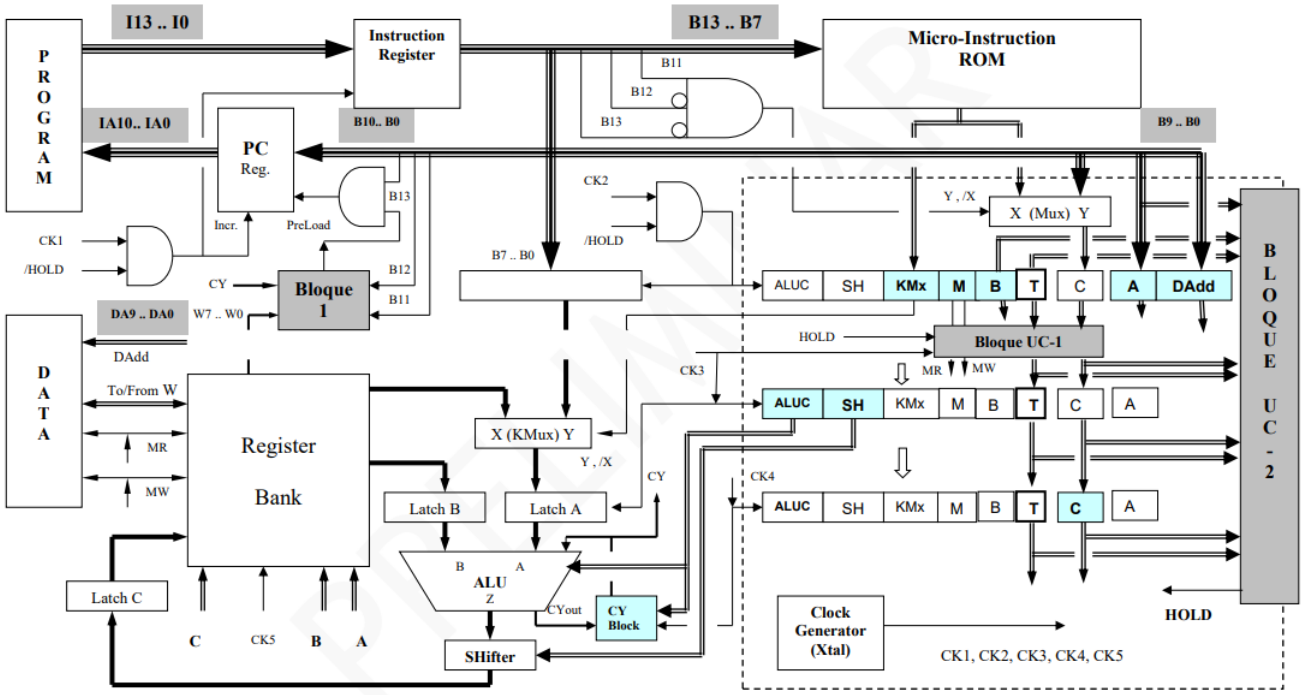


FIGURA 1.1: Arquitectura de referencia presentada por la cátedra

Como condición de diseño, se solicitó que la implementación incorpore registros de 16 bits, una salida VGA que maneje gráficos en baja o media resolución y manejo de subrutinas con un stack de direcciones de retorno de cuatro niveles.

1.2. Set de instrucciones y microinstrucciones

Usando como base el set de instrucciones presentado por la cátedra, se definieron e implementaron las instrucciones presentadas en la Tabla 1. A partir de las instrucciones elegidas, se desarrollaron las palabras de control de las microinstrucciones correspondientes, mostradas en la Tabla **??**.

1.3. Ensamblador

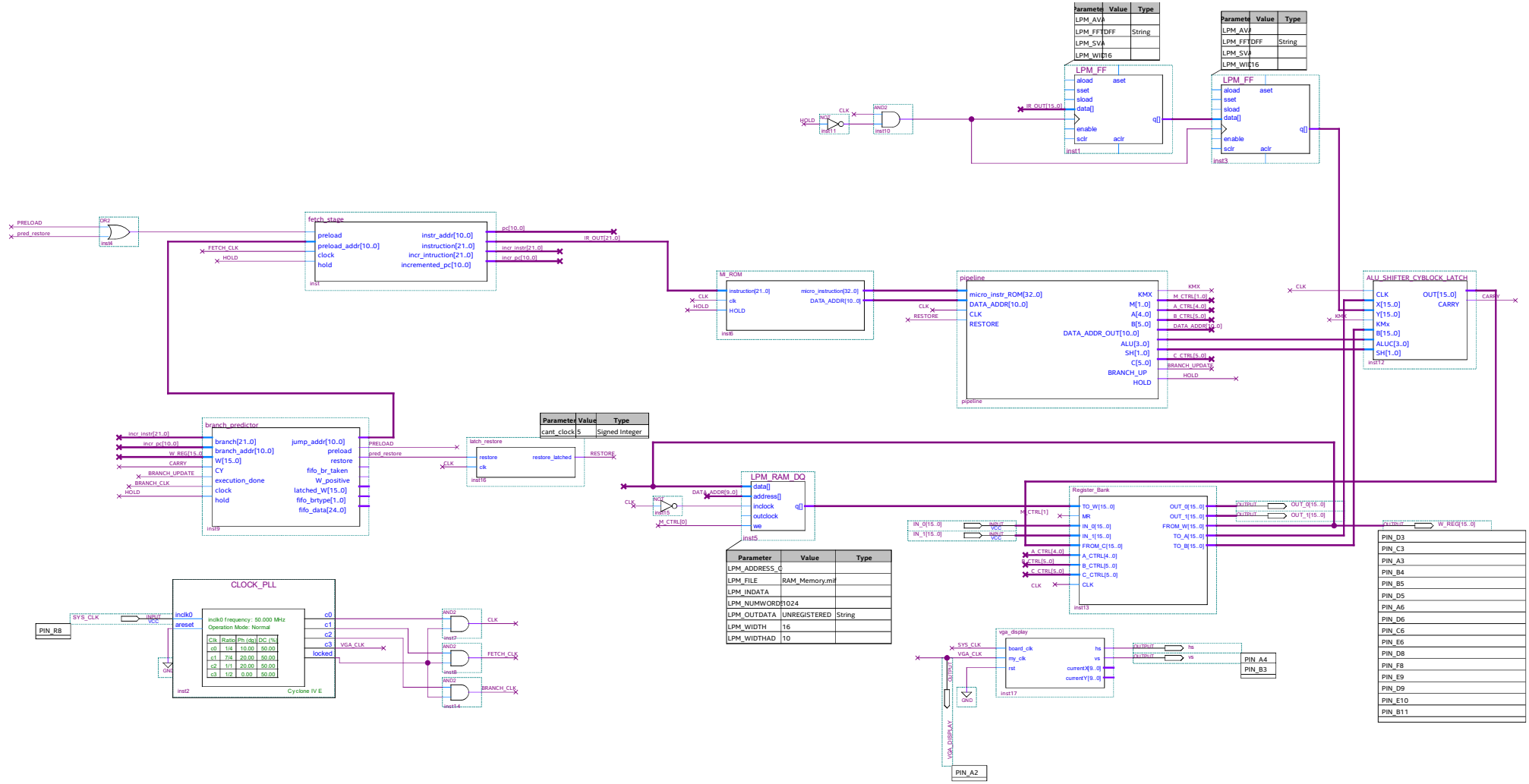
Se desarrollo un script en *Python* para traducir programas escritos en nemónicos a sus correspondientes palabras binarias de instrucción. El ensamblador convierte el código de texto con extensión .ev a un archivo de inicialización de memorias .mif de Quartus para así inicializar la memoria de programa con dicho archivo.

Código de Instrucción	Nemóico	Instrucción	Significado
10000000000xxxxxxxxxx	JMP X	Unconditional Jump	PC = X
10100000000xxxxxxxxxx	JZE X	Jumo if Working Register is Zero	IF W=0 THEN PC=X
11000000000xxxxxxxxxx	JPE X	Jump if Working Register is Positive	IF W15=0 THEN PC=X
11100000000xxxxxxxxxx	JCY X	Jump if Carry	IF CY THEN PC=X
01000000000yyyyyyyyyy	MOM Y,W	Move Working Register to Memory	M(Y) = W
01010000000yyyyyyyyyy	MOM W,Y	Move Memory to Working Register	W = M(Y)
01100000000iiiiijjjjj	ADW Ri,Rj	Add with Carry Reg. J with Working Reg. to Reg I	Ri = W + Rj + CY
01110000000sssssssss	BSR S	Unconditional Branch (Relative) to Subroutine S	Save PC; PC = PC + S
00100000000iiiiijjjjj	MOV Ri,Rj	Move Register J to Register I {Ri, Rj: 0 a 27}	Ri = Rj
00100000000001111ijjjjj	MOV POi,Rj	Move Register J to Output Port I	POi = Rj
00100000000iiiii1110j	MOV Ri,PIj	Move Input Port J to Register I	Ri = PIj
00100000000001111i1110j	MOV PO,PIj	Move Input Port J to Output Port I	POi = PIj
00110000000iiiii00000	MOV Ri,W	Move working Register to Register I	Ri = W
00110000000001111i00000	MOV POi,W	Move Working to Output Port I	POi = W
000100kkkkkkkkkkkkkkk	MOV W,#K	Move Constant to Working Register	W = K
000101kkkkkkkkkkkkkkk	ANK W,#K	AND Constant wit Working Register	W = W & K
000110kkkkkkkkkkkkkkk	ORK W,#K	OR Constant with Working Register	W = W OR K
000111kkkkkkkkkkkkkkk	ADK W,#K	ADD with Carry Constant with Working Register	W = W + K + CY
00001000000000000jjjjj	MOV W,Rj	Move Register J to Working Register	W = Rj
000010000000000001110j	MOV W,PIj	Move Input Port J to Working Register	W = PIj
00001010000000000jjjjj	ANR W,Rj	AND Register J with Working Register	W = W & Rj
00001100000000000jjjjj	ORR W,Rj	OR Register J with Working Register	W = W OR Rj
00001110000000000jjjjj	ADR W,Rj	ADD with Carry Register J with Working Register	W = W + Rj + CY
00000000000000000000	CPL W	Complement Working Register	W = /W
00000010000000000000	CLR CY	Clear Carry	CY = 0
00000100000000000000	SET CY	Set Carry	CY = 1
00000110000000000000	RET	Return From Subroutine	PC = Latest Stored PC {+1}
01111111111111111111	NOP	NOP Instruction	

Tabla 1: Set de instrucciones implementado

Nemónico	Significado	Funcion ALU	ALUC	SH	KMx	MR	MW	B Bus	C Bus	Type	A Bus
JMP X	PC = X	-	0000	00	0	0	0	100010	100011	1000000	00000
JZE X	IF W=0 THEN PC=X	-	0000	00	0	0	0	100010	100011	1000001	00000
JNE X	IF W15=0 THEN PC=X	-	0000	00	0	0	0	100010	100011	1000001	00000
JCY X	IF CY THEN PC=X	-	0000	00	0	0	0	100010	100011	1010000	00000
MOM Y,W	M(Y) = W	-	0000	00	0	0	1	100010	100011	0000001	00000
MOM W,Y	W = M(Y)	-	0000	00	0	1	0	100010	100011	0000010	00000
ADW Ri,Rj	Ri = W + Rj + CY	A + B + Cy	0101	00	0	0	0	100010	000000	0111101	00000
BSR S	Save PC; PC = PC + S	-	0000	00	0	0	0	100010	100011	1000000	00000
MOV Ri,Rj	Ri = Rj	A	0000	00	0	0	0	100010	000000	0001100	00000
MOV POi,Rj	POi = Rj	A	0000	00	0	0	0	100010	000000	0001100	00000
MOV Ri,PIj	Ri = PIj	A	0000	00	0	0	0	100010	000000	0001100	00000
MOV PO,PIj	POi = PIj	A	0000	00	0	0	0	100010	000000	0001100	00000
MOV Ri,W	Ri = W	B	0001	00	0	0	0	100010	000000	0001001	00000
MOV POi,W	POi = W	B	0001	00	0	0	0	100010	000000	0001001	00000
MOV W,#K	W = K	A	0000	00	1	0	0	100010	100010	0000010	00000
ORK W,#K	W = W OR K	A OR B	0110	00	1	0	0	100010	100010	0000011	00000
ANK W,#K	W = W & K	A & B	0111	00	1	0	0	100010	100010	0000011	00000
ADK W,#K	W = W + K + CY	A + B + Cy	0101	00	1	0	0	100010	100010	0110011	00000
MOV W,Rj	W = Rj	A	0000	00	0	0	0	100010	100010	0000110	00000
MOV W,PIj	W = PIj	A	0000	00	0	0	0	100010	100010	0000110	00000
ANR W,Rj	W = W & Rj	A & B	0111	00	0	0	0	100010	100010	0000111	00000
ORR W,Rj	W = W OR Rj	A OR B	0110	00	0	0	0	100010	100010	0000111	00000
ADR W,Rj	W = W + Rj + CY	A + B + Cy	0101	00	0	0	0	100010	100010	0110111	00000
CPL W	W = /W	/B	0011	00	0	0	0	100010	100010	0000011	00000
CLR CY	CY = 0	-	0000	00	0	0	0	100010	100011	0100000	00000
SET CY	CY = 1	-	0000	00	0	0	0	100010	100011	0100000	00000
RET	PC = Latest Stored PC {+1}	-	0000	00	0	0	0	100010	100011	1000000	00000
NOP	-	-	1111	00	0	0	0	100010	100011	0000000	00000

Tabla 2: Set de microinstrucciones



Parameter	Value	Type
cant_clock	5	Signed Integer

Parameter	Value	Type
LPM_ADDRESS_C		
LPM_FILE	RAM_Memory.mif	
LPM_INDATA		
LPM_NUMWORDS	1024	
LPM_OUTDATA	UNREGISTERED	String
LPM_WIDTH	16	
LPM_WIDTHAD	10	

Clk	Ratio	Ph (dg)	DC (%)
c0	1/4	10.00	50.00
c1	7/4	20.00	50.00
c2	1/1	20.00	50.00
c3	1/2	0.00	50.00

Pin
PIN_D3
PIN_C3
PIN_A3
PIN_B4
PIN_B5
PIN_D5
PIN_A6
PIN_D6
PIN_C6
PIN_E6
PIN_D8
PIN_F8
PIN_E9
PIN_D9
PIN_E10
PIN_B11

2. Pipeline

El microprocesador posee un pipeline de 5 etapa:

1. Fetch
2. Decode
3. Operand
4. Execute
5. Retire

La unidad de Fetch se encarga de enviar las instrucciones que se van a ejecutar en la etapa de *Type*, este proceso se detallará en una sección aparte. En la siguiente etapa se procede a obtener la microinstrucción que corresponde a partir de la instrucción obtenida en la etapa anterior. Este bloque se fija los bits con los que esta compuesta la instrucción y a partir de estos obtiene la microinstrucción que se utilizará Luego se procede a la etapa de *Operand*, en esta etapa se procede a buscar los operandos a utilizar, ya sea algún registro o alguna palabra en memoria, para esto se envían las palabras de control al *BUS A* y *BUS B* y se cargan los registros correspondientes a cada bus de datos. También se envía la señal de control correspondiente al *KMux* que permite seleccionar una constante externa para las instrucciones que lo requieren, se informa si se leerá o escribirá en memoria y a su vez se envía la dirección que se utilizará en este caso. A continuación en la etapa *Execute* se envían las señales de control a la ALU y al SHIFTER donde se ejecutará la operación correspondiente a la microinstrucción empleada. Finalmente en la etapa *Retire*, se guarda el resultado de la operación obtenida en la etapa anterior en el registro que indica el *BUS C*.

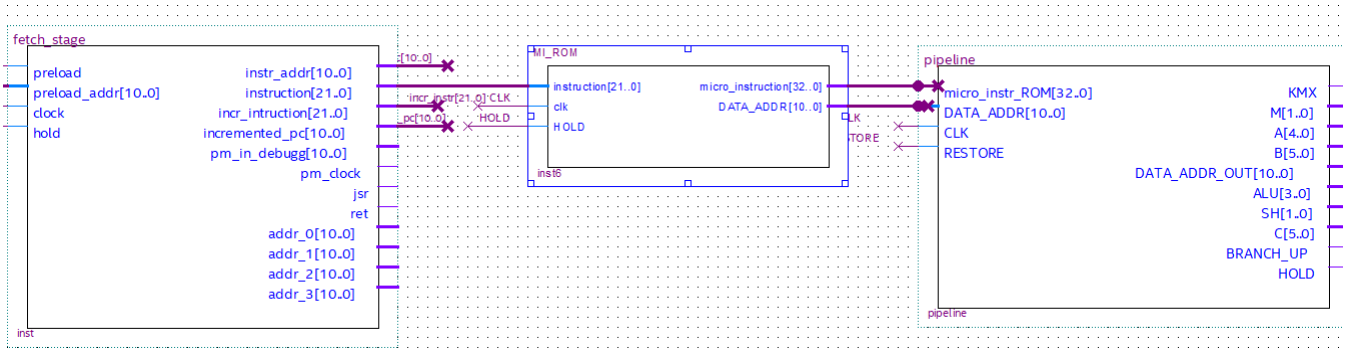


FIGURA 2.1: Bloque pipeline

2.1. Bloque de control de microinstrucciones

Estos bloques de control se encargan de manejar las dependencias entre las instrucciones dentro del pipeline

2.1.1. Unidad de control 1

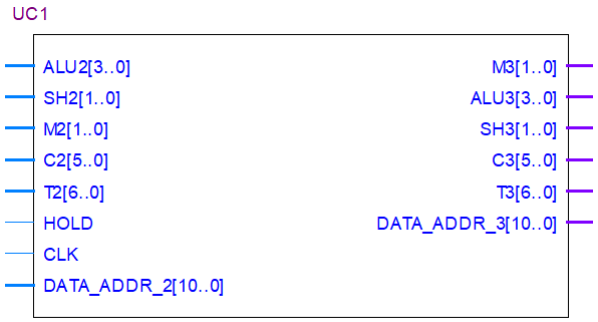


FIGURA 2.2: unidad de control 1

Este bloque recibe como entradas la microinstrucción proveniente de la etapa de decode del y la señal de HOLD proveniente de la salida del bloque de control 2. Cuando la señal de HOLD se encuentra en estado alto se le esta indicando a la unidad de control que se están por realizar varias operaciones en el mismo registro por lo tanto debe frenar la microinstrucción que recibió a la entrada hasta que la anterior termina de ejecutarse la microinstrucción que se encuentra en una etapa mas avanzada. Para realizar esto manda como salida a la ALU la operación de NOP y que a su vez cargue el en bus C el registro 35 que es el que se asegura que no se modifique ni lea ningún registro del banco.

2.1.2. Unidad de control 2

En el siguiente bloque se manejan las dependencias entre microinstrucciones analizando el sector de *Type* de cada una de estas. Este bloque se encarga de analizar el estado de las ultimas 4 etapas del pipeline para decidir si el flujo de este debe continuar o si se debe parar etapas anteriores hasta que se haya terminado de ejecutar microinstrucciones previas. Para realizar esto el bloque se fija en el *Type* de la microinstrucción que se encuentra en la etapa de *Decode* y analiza el *Type* de las microinstrucciones en etapas posteriores para ver si se esta por realizar alguna modificación o lectura en algún registro en el que todavía no se ha terminado de modificar, si es así se activa la señal de HOLD hasta que esta microinstrucción termine de modificar el registro y dar lugar a que las microinstrucciones demoradas continúen con su ejecución.

3. Unidad de fetch

Se implementó una unidad de *fetch* que se encarga de llevar registro del PC, recuperar las instrucciones de la memoria de programa, y gestionar el stack de PC para los saltos a subrutinas. El bloque implementado, con sus entradas y salidas, se muestra en la Figura 3.1,

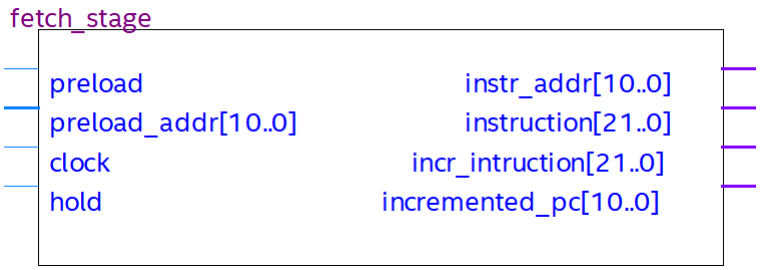


FIGURA 3.1: Bloque de unidad de fetch

El modulo recibe como entradas una señal de clock, una señal de preload y dirección de preload (provenientes del módulo predictor) y una señal de hold proveniente del módulo de control de pipeline. La señal de clock utilizada por la etapa de fetch tiene una frecuencia 7 veces mayor al clock del pipeline, pues por cada clock de pipeline, la etapa de fetch realiza las siguientes operaciones secuenciales:

1. Latchear entrada.
2. Actualizar PC
3. Leer de la memoria de programa la instrucción apuntada por el PC.
4. Decodificar la instrucción apuntada por el PC.
5. Leer de la memoria de programa la instrucción PC + 1.
6. Decodificar la instrucción apuntada por PC + 1.
7. Latchear salida.

3.1. Bloque PC

El bloque PC (Program Counter) se encarga de actualizar de forma correcta el PC. Tiene 4 comportamientos diferentes en 4 escenarios distintos. El primer escenario es aquel en el cuál no ocurren saltos a subrutinas, saltos condicionales, no retornos de subrutinas. En este caso, el PC simplemente se incrementa en una unidad. El segundo escenario es cuando la señal de *preload* esta encendida. En este caso, el PC se actualizará al valor que reciba en la entrada *preload_ addr*. Este escenario se da cuando se toma un salto condicional, o bien cuando se hace un failover luego de haber tomado una predicción equivocada. El tercer escenario es cuando llega una instrucción de salto a subrutina. En este caso, la dirección actual de PC se almacena en un stack de direcciones, y el PC se actualiza con el valor PC_{actual} + relative_addr. Por último, cuando ocurre un retorno de subrutina, se actualiza el PC con el valor almacenado en el stack.

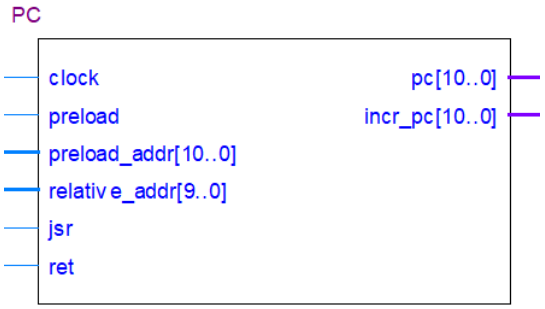


FIGURA 3.2: Bloque PC

3.2. Memoria de programa

Se utilizó un modelo de memoria ROM para implementar la memoria de programa. El módulo cuenta con capacidad para 2048 palabras de 22 bits, y su inicialización se realiza mediante un archivo .mif que se genera a partid de un archivo de texto con nemónicos utilizando un compilador especialmente desarrollado para el presente proyecto (que se detallará mas adelante).

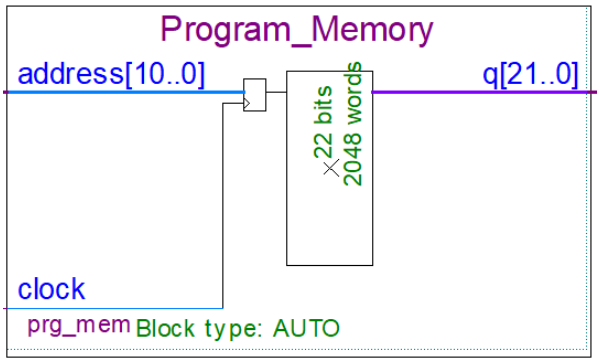


FIGURA 3.3: Memoria de programa

3.3. Registro de instrucciones

El bloque *IRegister* mostrado en la Figura 3.4, cumple la función de detectar si la instrucción actual que se esta procesando es una instrucción de salto a subrutina o de retorno de subrutina, para realimentar la información al bloque PC en el próximo ciclo de clock de pipeline.

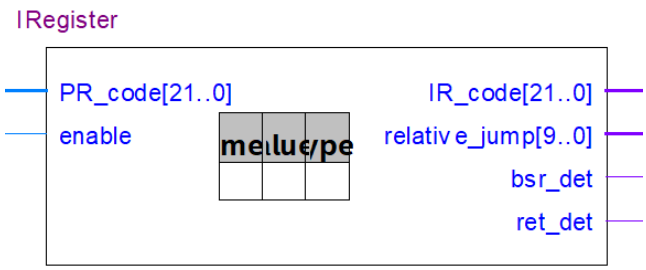
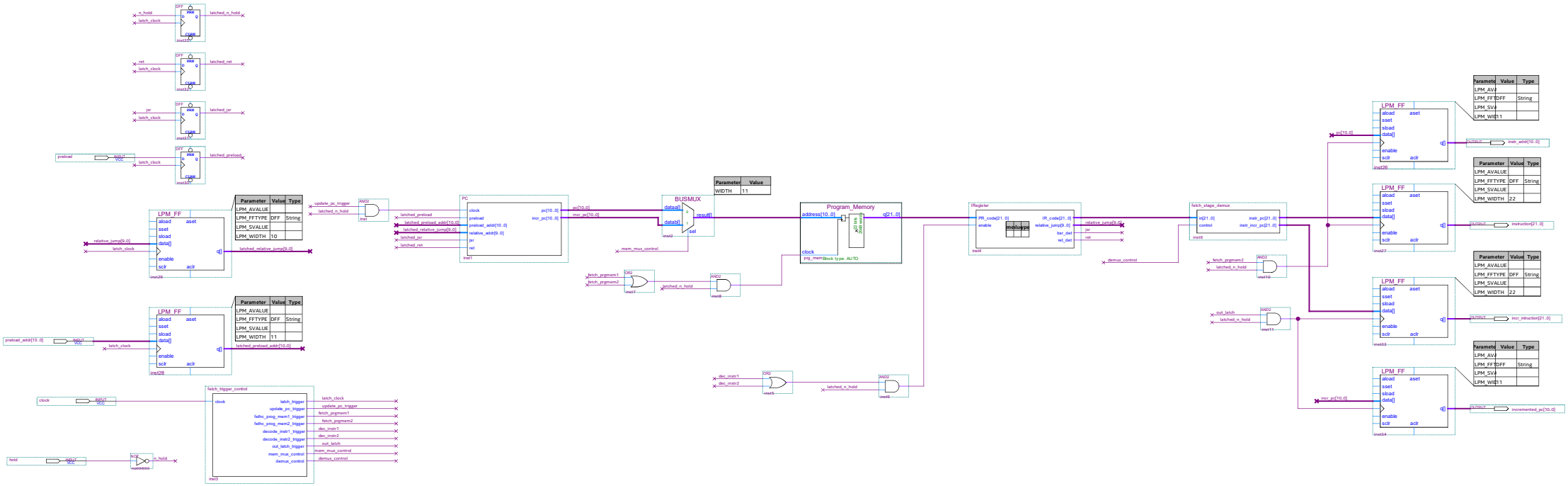


FIGURA 3.4: Bloque IRegister


n_hold
latch_clock
latched_n_hold
ret
latched_ret
jsr
latched_jsr
preload
latched_preload
LPM_FF
aload
sset
sload
data[]
enable
sclr
aclr
q[]
relative_jump[9..0]
latched_relative_jump[9..0]
preload_addr[10..0]
latched_preload_addr[10..0]
update_pc_trigger
latched_n_hold
AND2
latched_preload
latched_preload_addr[10..0]
latched_relative_jump[9..0]
latched_jsr
latched_ret
PC
clock
preload
preload_addr[10..0]
relative_addr[9..0]
jsr
ret
pc[10..0]
incr_pc[10..0]
BUSMUX
data[1]
data[0]
result[]
sel
Parameter Value
WIDTH 11
mem_mux_control
Program_Memory
address[10..0]
q[21..0]
prg_mem
Block type: AUTO
fetch_prgmem1
fetch_prgmem2
CMP2
dec_instr1
dec_instr2
IRegister
PR_code[21..0]
enable
IR_code[21..0]
relative_jump[9..0]
bsr_dst
ret_dst
demux_control
fetch_stage_demux
in[21..0]
control
instr_pc[21..0]
instr_incr_pc[21..0]
out_latch
LPM_FF
aset
aload
sset
sload
data[]
enable
sclr
aclr
q[]
pc[10..0]
instr_addr[10..0]
instruction[21..0]
incr_instruction[21..0]
incr_pc[10..0]
incremented_pc[10..0]
Parameter Value Type
LPM_AVALUE
LPM_FFTYPE DFF String
LPM_SVALUE
LPM_WIDTH 22
clock
hold
NOR2
n_hold
fetch_trigger_control
latch_trigger
update_pc_trigger
fetfhc_prog_mem1_trigger
fetfhc_prog_mem2_trigger
decode_instr1_trigger
decode_instr2_trigger
out_latch_trigger
mem_mux_control
demux_control
fetch_prgmem1
fetch_prgmem2
dec_instr1
dec_instr2
out_latch
mem_mux_control
demux_control

4. ALU

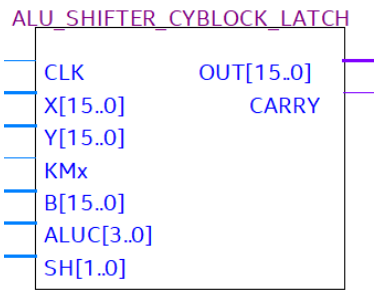


FIGURA 4.1: Bloque de la ALU

El bloque de la Figura 4.1 implementa la unidad arimético-lógica del procesador. En la tabla 3 se detallan las distintas operaciones disponibles, función de su señal de control $ALUC$.

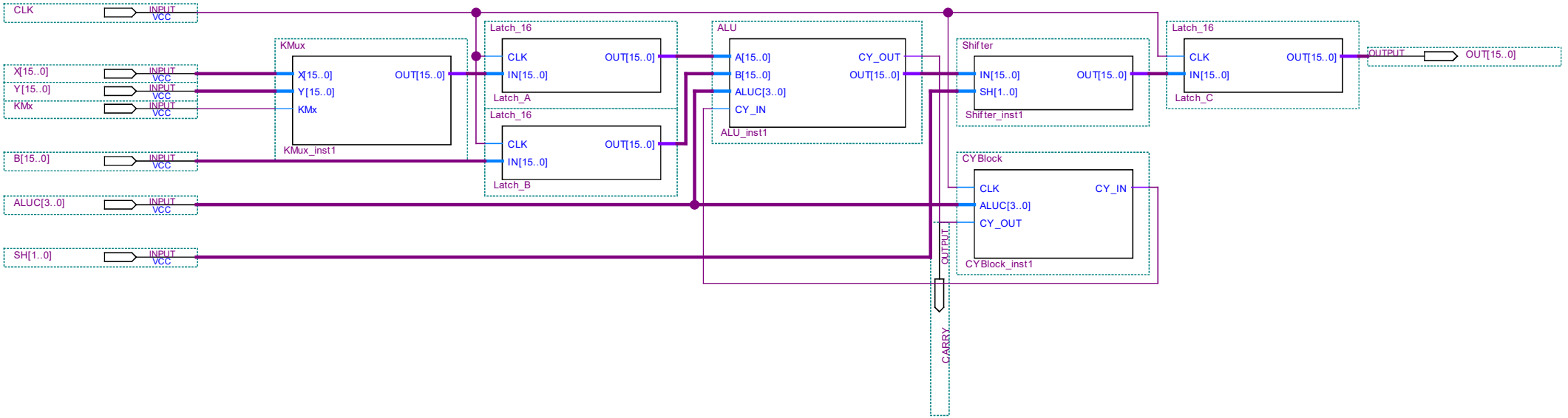
ALUC	Operacion ALU
0	$Z = A$
1	$Z = B$
2	$Z = /A$
3	$Z = /B$
4	$Z = A + B$
5	$Z = A + B + CY$
6	$Z = A$ OR B
7	$Z = A\&B$
8	$Z = 0$
9	$Z = 1$
10	$Z = 0xFFFF$
11	$CY = 0$
12	$CY = 1$

TABLA 3: Operaciones del bloque de ALU

5. Shifter

El bloque shifter implementa shifts del tipo lógico a la derecha o a la izquierda dependiendo de la señal de control que recibe en la etapa de operando.

SH	Operacion Shifter
0	No Shift
1	Shift Right 1 bit
2	Shift Left 1 bit


CLK
INPUT
VCC
X[15..0]
Y[15..0]
KMx
B[15..0]
ALUC[3..0]
SH[1..0]
KMux
X[15..0]
Y[15..0]
KMx
OUT[15..0]
KMux_inst1
Latch_16
CLK
IN[15..0]
OUT[15..0]
Latch_A
Latch_B
ALU
A[15..0]
B[15..0]
ALUC[3..0]
CY_IN
CY_OUT
OUT[15..0]
ALU_inst1
Shifter
IN[15..0]
SH[1..0]
OUT[15..0]
Shifter_inst1
CYBlock
CLK
ALUC[3..0]
CY_OUT
CY_IN
CYBlock_inst1
OUTPUT
CARRY
Latch_C
OUT[15..0]

6. Predictor de saltos dinámico

Uno de las característica adicional al diseño presentado por la cátedra que presenta la presente implementación es un predictor de saltos dinámico. El propósito del mismo es adelantarse a los saltos condicionales sin que se deba detener el flujo del pipeline. El diseño del predictor permite que el resultado de la predicción sea condicionado por los resultados de las ejecuciones anteriores, para una instrucción de salto en particular. El bloque realiza dos funciones basicas: predecir y actualizar.

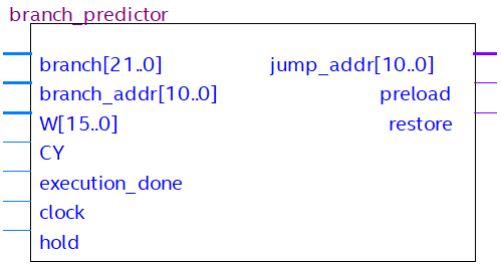


FIGURA 6.1: Bloque del predictor de saltos

Cuando se requiere que el bloque prediga el resultado de un salto condicional, se le debe proveer el binario de la instruccion de salto condicional *branch[21..0]* y la dirección de la instrucción de salto *branch_ addr[10..0]*. El bloque indicará en su salida *preload* si se debe hacer un salto a una dirección determinada en la salida *jump_ addr[10..0]*.
Por otro lado, la entrada *execution_ done* indica que la información de la ejecución de la condición del salto esta disponible para actualizar los datos del predictor. En ese caso, se leen las entradas de carry y el registro W, para actualizar la información del salto condicional correspondiente. Si sucediera que la predicción hecho no coincide con el resultado de la ejecución de la condición de salto, se le indica al control del pipeline que hubo un error de prediccion mediante la salida *restore*, y se indica la direccion de *failover* para retomar el flujo del programa en la instrucción que corresponda.
El predictor se alimenta de un clock que tiene una frecuencia 4 veces mayor a la del clock del pipeline, ya que internamente realiza 4 operaciones por cada ciclo del pipeline: latchea entrada, actualiza saltos, predice saltos, latchea salida.

6.1. Predict enable control

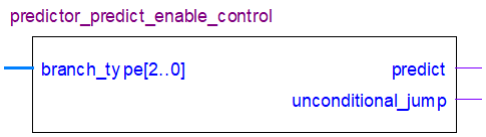


FIGURA 6.2: Bloque predictor_predict_enable_control

Este bloque recive los 3 bits mas significativos de la instrucción de salto condicional, que definien unívocamente si la instrucción se trata de un salto condicional, y de que tipo (JMP, PCY, JPO o JZE). Se indica en la salida *predict* si se debe realizar una predicción sobre la instrucción recibida. La salida *unconditional_ jump* indica si la instrucción es del tipo salto incondicional.

6.2. Branch history table

La tabla de historial de saltos (o BHT por sus siglas en inglés), es el módulo central del predictor. La tabla cuenta con 2^8 entradas, indexadas por los 8 bits menos significativos de la dirección de la instrucción de salto. Cada entrada almacena la información relevante a una instrucción de salto: la dirección entera de la instrucción y el estado del historial de saltos, codificado en 2 bits, que sirve para determinar el resultado de la predicción para salto registrado en la correspondiente entrada de la tabla.

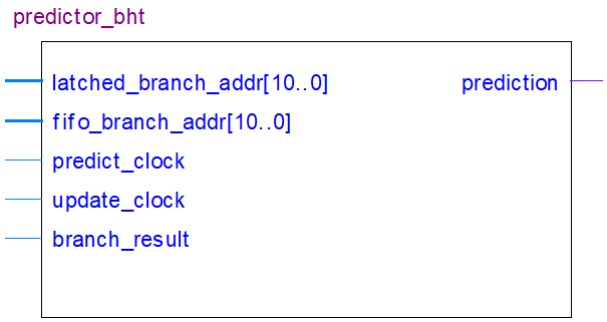
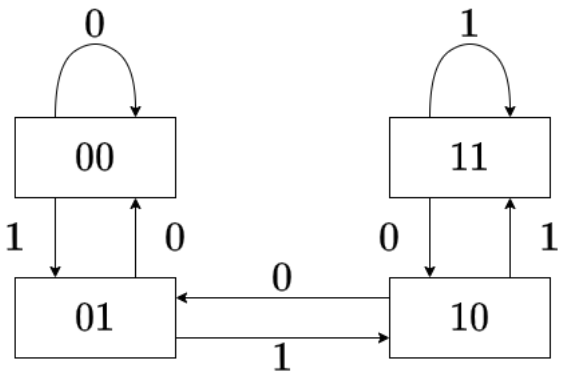


FIGURA 6.3: Bloque BHT

Entonces, cada vez que se solicita una predicción para un salto particular (identificado por la dirección de su instrucción), se indexa la tabla con los 8 bits menos significativos de la dirección de la instrucción de salto y se verifica que la dirección entera de la instrucción que se quiere predecir coincida con la almacenada en la tabla. De coincidir, se decodifica el estado del historial de saltos para determinar el valor de la predicción. Si la dirección almacenada en el registro correspondiente no coincidiera con la dirección del salto que se quiere predecir, se toma un valor de predicción por defecto.



00 : Fuertemente No-Tomado

01 : Débilmente No-Tomado

10 : Débilmente Tomado

11 : Fuertemente Tomado

FIGURA 6.4: FSM del historial de saltos

Por otro lado, cada vez que se cuenta con el resultado de la ejecución de la condición de salto de un salto condicional, se debe actualizar esta tabla. En este caso, se toma la dirección de la instrucción del salto que se quiere actualizar de una memoria FIFO (que se verá mas adelante) y el resultado de

la ejecución de la condición del salto para modificar el historial de salto del salto correspondiente, tal como se indica en la Figura 6.4.

6.3. FIFO y Prediction Check

Cada vez que el bloque le llega una solicitud de predicción, además de realizar la predicción a través de la BHT como se explicó anteriormente, también debe almacenarse la información relevante a la predicción temporalmente hasta que la ejecución de la condición de salto este resuelta. Para ello se utiliza una FIFO que para cada predicción almacena la dirección de la instrucción de salto, la dirección de salto, el tipo de salto y el resultado de la predicción.
De esta forma, cuando llega una solicitud de actualización, se contrasta el valor del resultado de la predicción almacenado en la FIFO para determinar el éxito de la predicción, y la dirección de la instrucción del salto para actualizar la BHT.

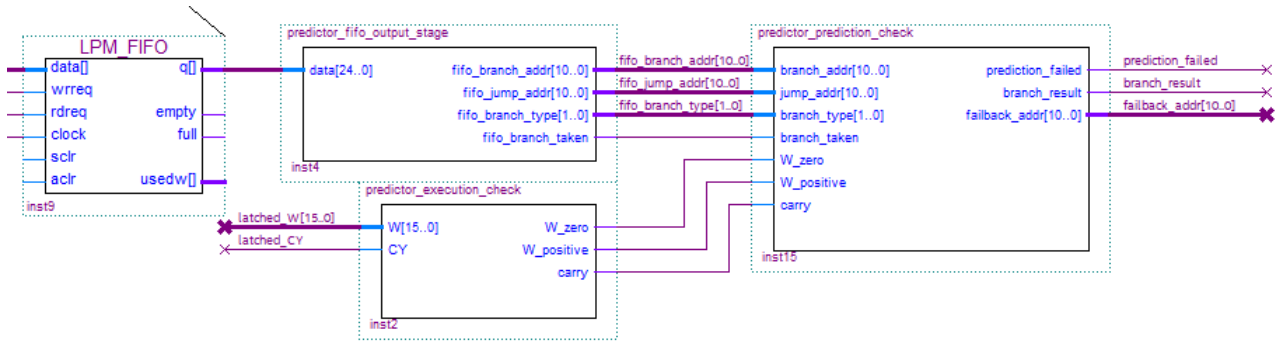
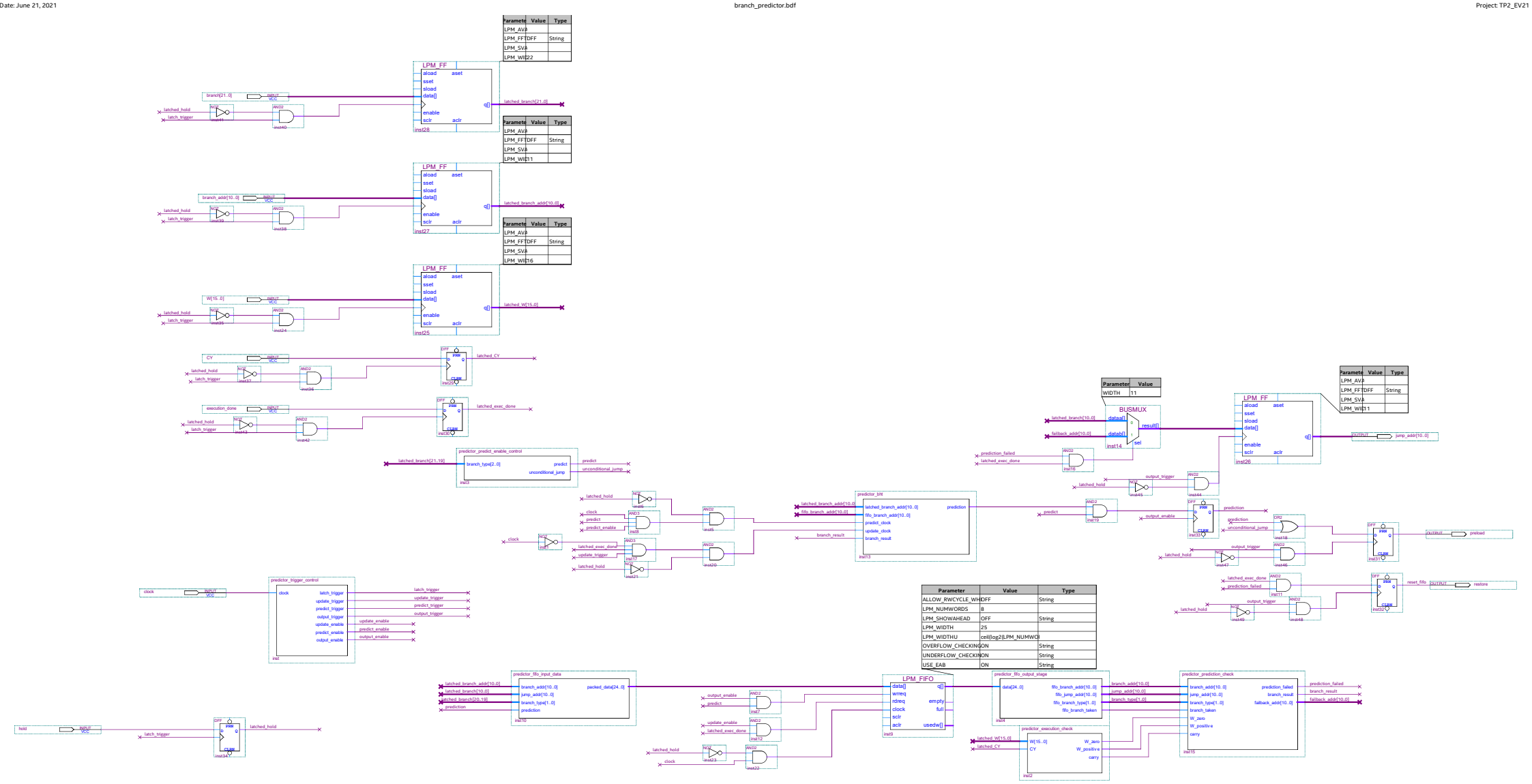


Figura 6.5: Bloques FIFO y Prediction Check



Parameter	Value	Type
LPM_AVA		
LPM_FFTDFF	String	
LPM_SVA		
LPM_WID	22	

Parameter	Value	Type
LPM_AVA		
LPM_FFTDFF	String	
LPM_SVA		
LPM_WID	11	

Parameter	Value	Type
LPM_AVA		
LPM_FFTDFF	String	
LPM_SVA		
LPM_WID	16	

Parameter	Value
WIDTH	11

Parameter	Value	Type
LPM_AVA		
LPM_FFTDFF	String	
LPM_SVA		
LPM_WID	11	

Parameter	Value	Type
ALLOW_RWCYCLE_WH	OFF	String
LPM_NUMWORDS	8	
LPM_SHOWAHEAD	OFF	String
LPM_WIDTH	25	
LPM_WIDTHU	ceil(log2(LPM_NUMWO	
OVERFLOW_CHECKIN	ON	String
UNDERFLOW_CHECKIN	ON	String
USE_EAB	ON	String

7. Banco de registros

El diseño cuenta con un banco de registros que se encarga del control de flujo del bus de salida de la ALU hacia los registros de uso general, los puertos de salida y la memoria de datos, así como el flujo de datos desde registros, puertos y memorias hacia los buses de entrada de la ALU. Como se mencionó anteriormente, se manejan datos de 16 bits.

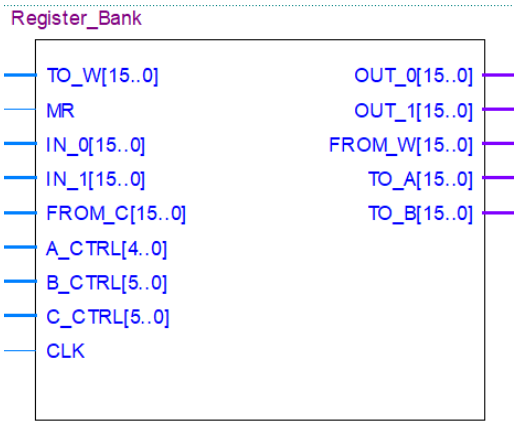


FIGURA 7.1: Bloque de banco de registros

8. Implementación de VGA

Se procedió a implementar un bloque compatible con un monitor *VGA* de resolución $640x480$ pixels. Para ello se utilizo un clock corriendo a 25 MHz, a su vez cuenta con una señal de reset en caso de querer reiniciar la pantalla. El bloque tiene como salida una señal de sincronización horizontal, una vertical y 2 señales de control que indican las coordenadas X e Y del pixel que se esta actualizando en ese momento.

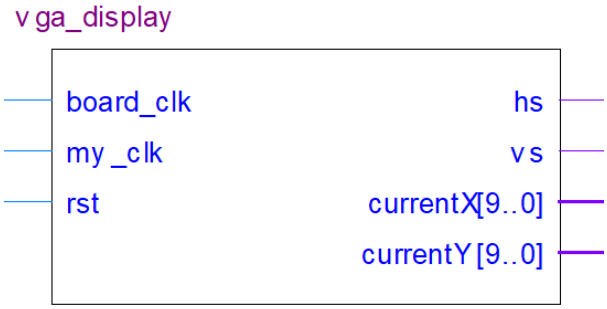


FIGURA 8.1: Modulo de protocolo VGA

9. Mediciones

9.1. Copiador de PI a PO

Se escribió un programa para leer datos desde el puerto de entrada y escribirlos al puerto de salida. La Figura 9.1 muestra los resultados obtenidos, midiendo los 8 bits menos significativos de ambos puertos. Con la ayuda de un Analog Discovery, se puso un contador ascendente en el puerto de entrada PI.

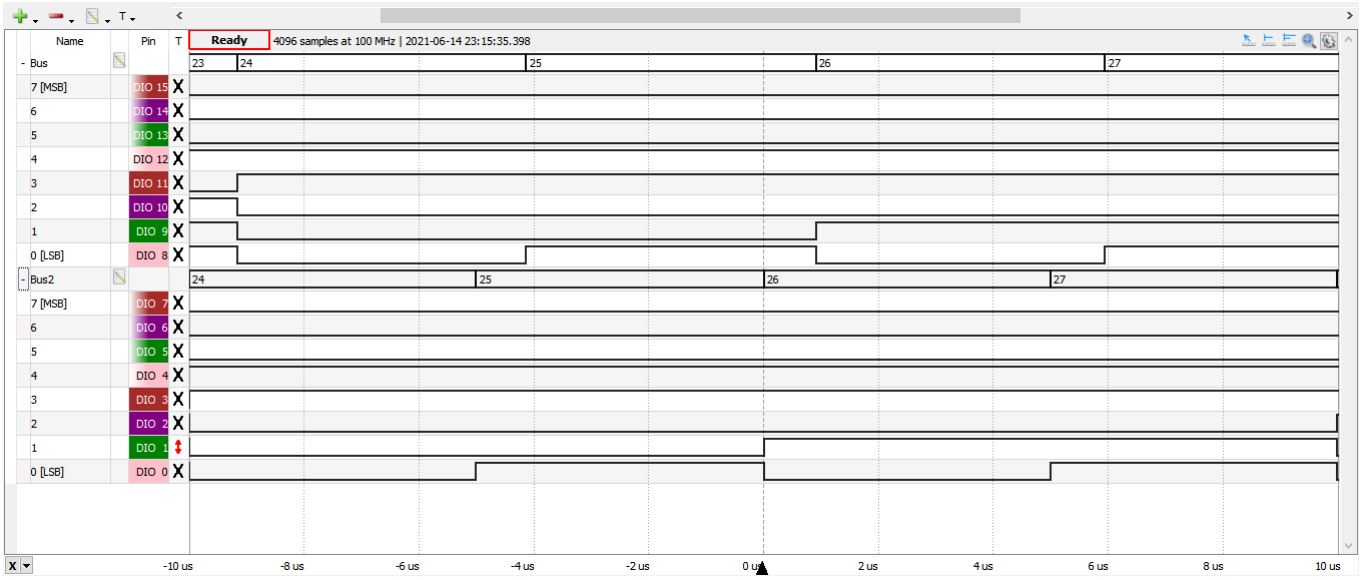


FIGURA 9.1: Señales registradas en el Analog Discovery

9.2. Contador en registro W

Se escribió un programa que implemente un contador en el registro W. Se carga un valor en memoria, se setea el registro W en cero, y luego se incrementa W con el valor almacenado en memoria. Luego se ejecuta un salto incondicional a la instrucción de incremento. Se midió el registro W con la ayuda de un Analog Discovery y los resultados se muestran en la Figura 9.2.

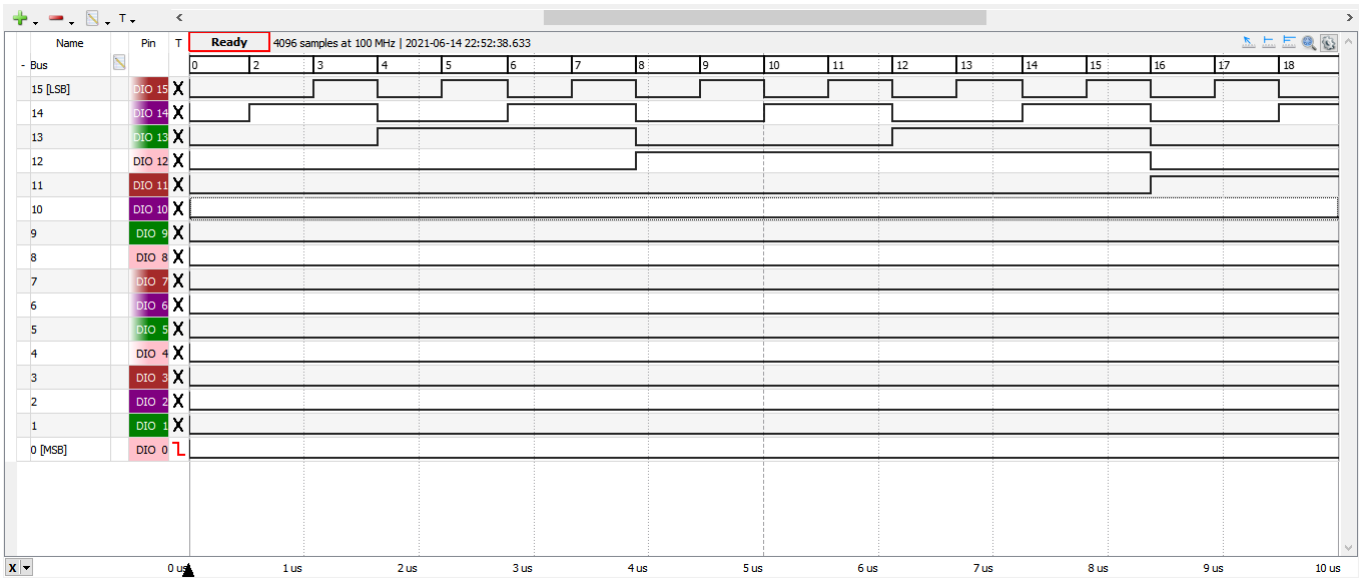


FIGURA 9.2: Señales registradas en el Analog Discovery