BUENOS AIRES INSTITUTE
OF
TECHNOLOGY

Electronics Engineering

22.15 - Electronics V

EV21 RISC PROCESSOR

PRACTICAL WORK 2

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

Contents

1 Introduction

1.1 Summary

This report details the development of the EV21 RISC processor, whose implementation is inspired by the design in Figure 1.1. The design was developed in Quartus software and then implemented on a Cyclone IV FPGA.

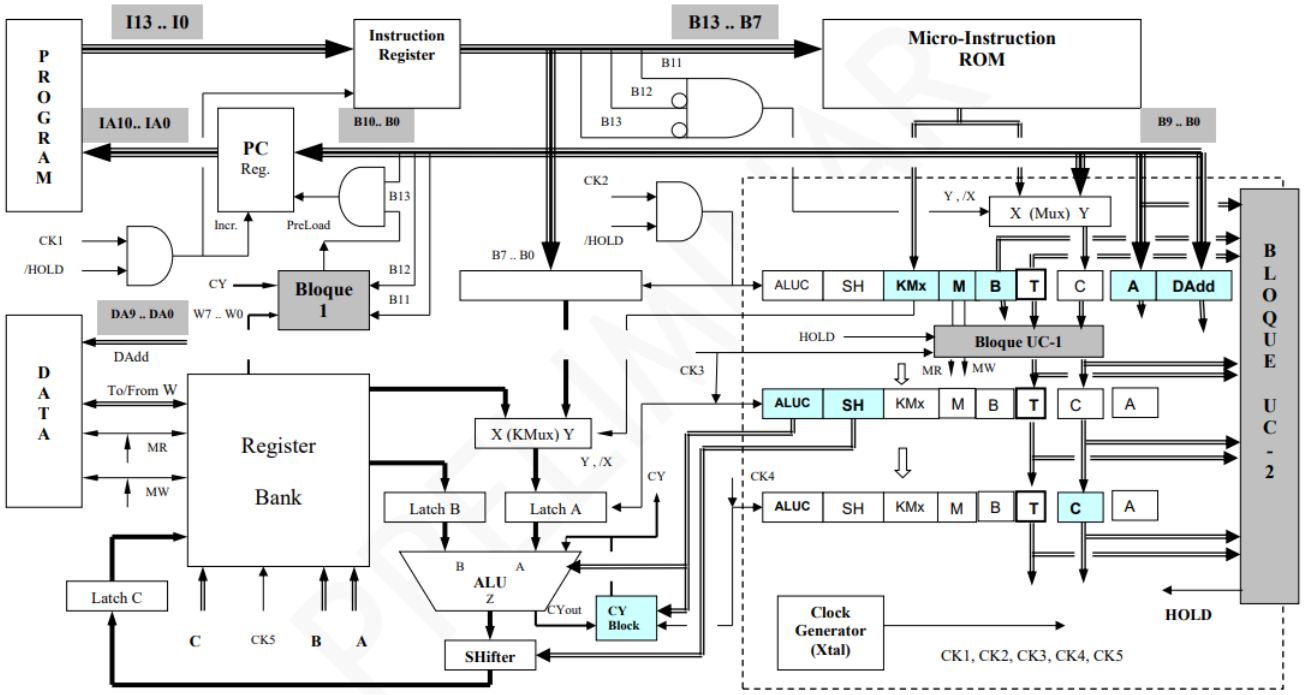


FIGURE 1.1: Reference architecture presented by the chair

As a design condition, it was requested that the implementation incorporate 16-bit registers, a VGA output that handles low to medium resolution graphics, and subroutine handling with a four-level return address stack.

1.2 Set of instructions and microinstructions

Using as a basis the set of instructions presented by the chair, the instructions presented in Table 1 were defined and implemented. From the chosen instructions, the control words of the corresponding microinstructions, shown in Table 2, were developed.

1.3 Ensamblador

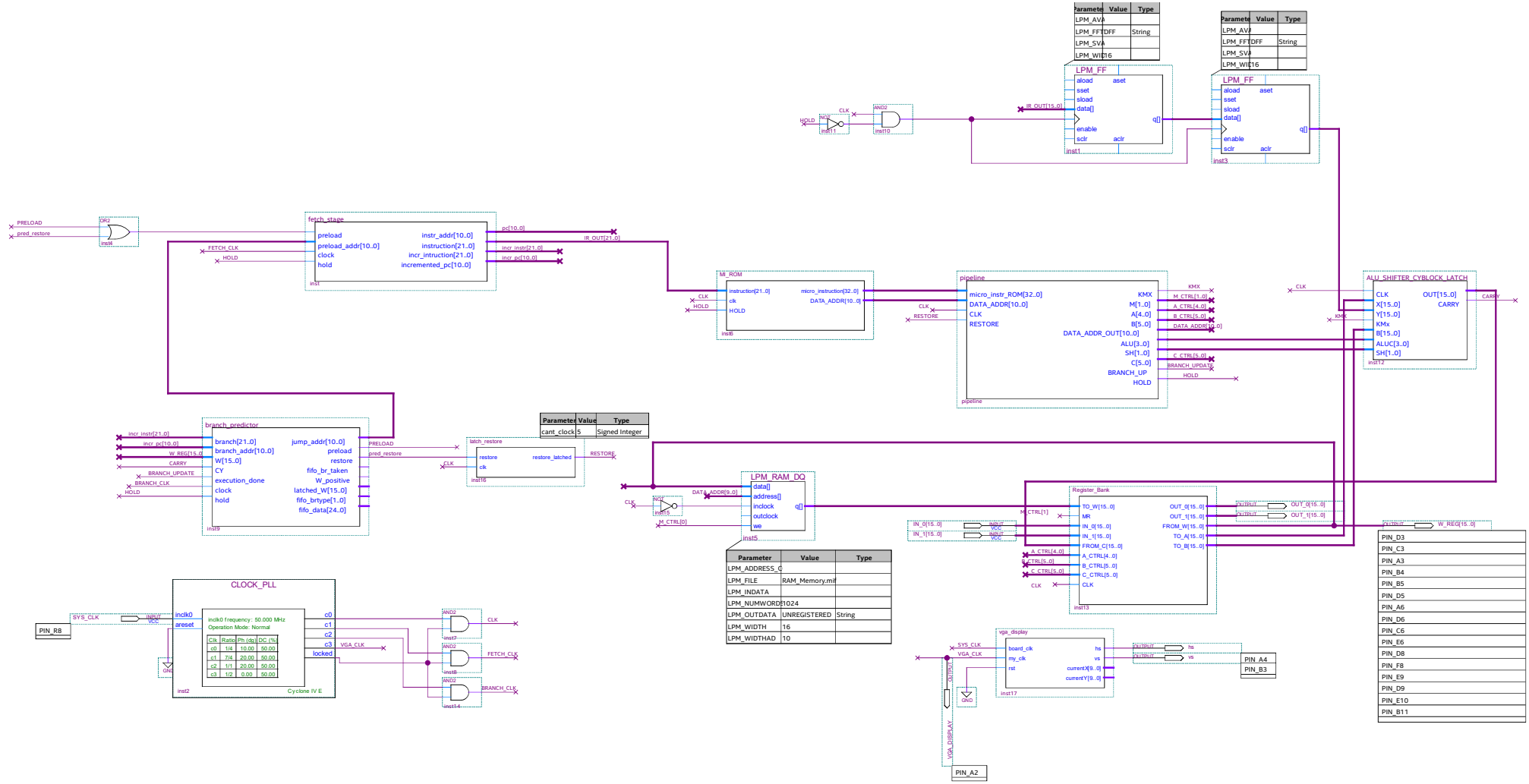
A script was developed in *Python* to translate programs written in mnemonics to their corresponding binary instruction words. The assembler converts text code with .ev extension to a Quartus .mif memory initialization file in order to initialize the program memory with that file.

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
00100000000001111jjjjj	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
00001000000000000001110j	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	

Table 1: Implemented instruction set

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

Table 2: Set of microinstructions



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

The microprocessor has a 5-stage pipeline:

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

The Fetch unit is in charge of sending the instructions to be executed in the Type stage, this process will be detailed in a separate section. The next stage proceeds to fetch the corresponding microinstruction from the instruction fetched in the previous stage. This block sets the bits with which the instruction is composed and obtains the microinstruction to be used from these bits. Then we proceed to the ÒtextitOperand stage, in this stage we proceed to search for the operands to be used, either a register or a word in memory, for this the control words are sent to the *BUS A* and *BUS B* and the registers corresponding to each data bus are loaded. The corresponding control signal is also sent to the *KMux* which allows selecting an external constant for the instructions that require it, it is informed whether it will be read or written to memory and in turn the address to be used in this case is sent. Next, in the *Execute* stage, the control signals are sent to the ALU and the SHIFTER where the operation corresponding to the microinstruction used will be executed. Finally, in the *Retire* stage, the result of the operation obtained in the previous stage is stored in the register indicated by the *BUS C*.

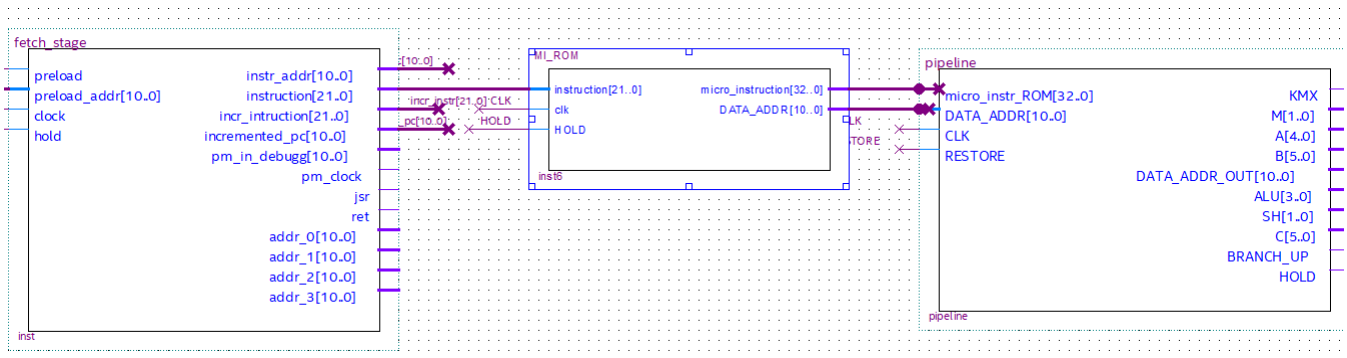


FIGURE 2.1: Pipeline block

2.1 Microinstruction control block

These control blocks are responsible for handling the dependencies between instructions in the pipeline.

2.1.1 Control unit 1

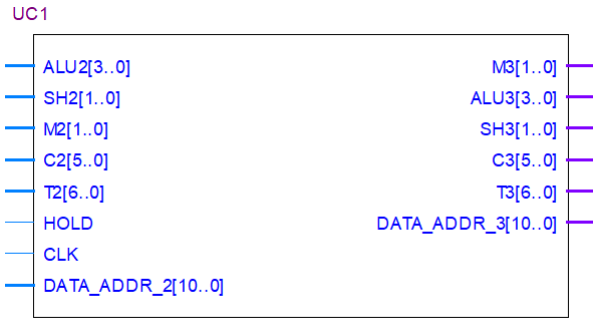


FIGURE 2.2: Control unit 1

This block receives as inputs the microinstruction coming from the decode stage and the HOLD signal coming from the output of control block 2. When the HOLD signal is high, it is indicating to the control unit that several operations are about to be performed in the same register, therefore it must stop the microinstruction received at the input until the previous one finishes executing the microinstruction that is in a more advanced stage. To do this, it sends as output to the ALU the NOP operation and in turn loads the C bus register 35 which is the one that ensures that no register of the bank is modified or read.

2.1.2 Control unit 2

The next block handles the dependencies between microinstructions by analyzing the *Type* sector of each one of them. This block is in charge of analyzing the status of the last 4 stages of the pipeline to decide if the pipeline flow should continue or if previous stages should be stopped until the execution of previous microinstructions is finished. To do this, the block looks at the *Type* of the microinstruction that is in the *Decode* stage and analyzes the *Type* of the microinstructions in later stages to see if it is about to make any modification or reading in any register in which it has not yet finished modifying, if so, the HOLD signal is activated until this microinstruction finishes modifying the register and gives rise to the delayed microinstructions to continue with their execution.

3 Fetch unit

A *fetch* unit was implemented, which is responsible for keeping PC register, fetching instructions from program memory, and managing the PC stack for jumps to subroutines. The implemented block, with its inputs and outputs, is shown in Figure 3.1,

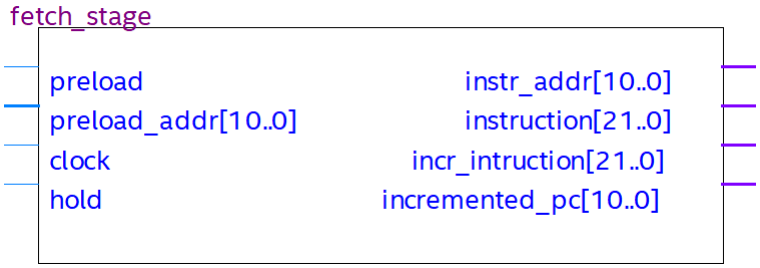


FIGURE 3.1: Bloque de unidad de fetch

The module receives as inputs a clock signal, a preload signal and preload direction (coming from the predictor module) and a hold signal coming from the pipeline control module. The clock signal used by the fetch stage has a frequency 7 times higher than the pipeline clock, because for each pipeline clock, the fetch stage performs the following sequential operations:

1. Latch Input.
2. Update PC
3. Read from the program memory the instruction pointed by the PC.
4. Decode the instruction pointed by the PC.
5. Read from the program memory the instruction PC + 1.
6. Decode the instruction pointed by PC + 1.
7. Latch output.

3.1 PC Block

The PC (Program Counter) block is responsible for correctly updating the PC. It has 4 different behaviors in 4 different scenarios. The first scenario is the one in which no subroutine jumps, conditional jumps, no subroutine returns occur. In this case, the PC is simply incremented by one unit. The second scenario is when the *preload* signal is on. In this case, the PC will update itself to the value it receives at the *preload_ addr* input. This scenario occurs when a conditional jump is taken, or when a failover is made after taking a wrong prediction. The third scenario is when a subroutine jump instruction arrives. In this case, the current PC address is stored in an address stack, and the PC is updated with the value $PC_{current}$ + relative_addr. Finally, when a subroutine return occurs, the PC is updated with the value stored in the stack.

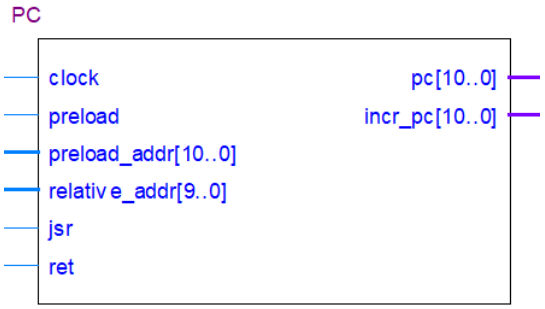


FIGURE 3.2: PC Block

3.2 Program memory

A ROM memory model was used to implement the program memory. The module has a capacity for 2048 words of 22 bits, and its initialization is done by means of a .mif file that is generated from a text file with mnemonics using a compiler specially developed for this project (to be detailed later).

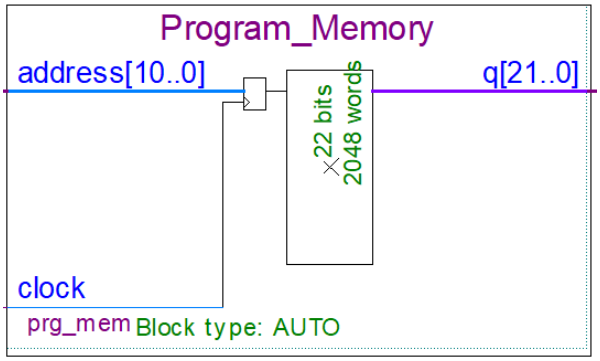


FIGURE 3.3: Program memory

3.3 Instruction Register

The *IRegister* block shown in Figure 3.4, performs the function of detecting whether the current instruction being processed is a subroutine jump or subroutine return instruction, in order to feed the information back to the PC block in the next pipeline clock cycle.

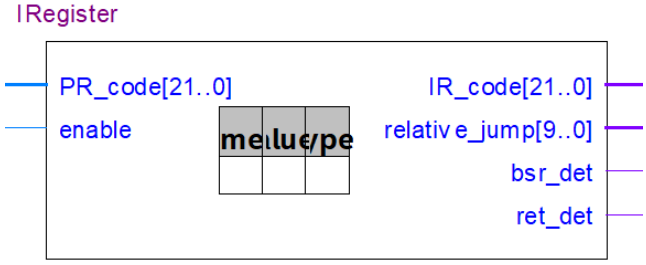
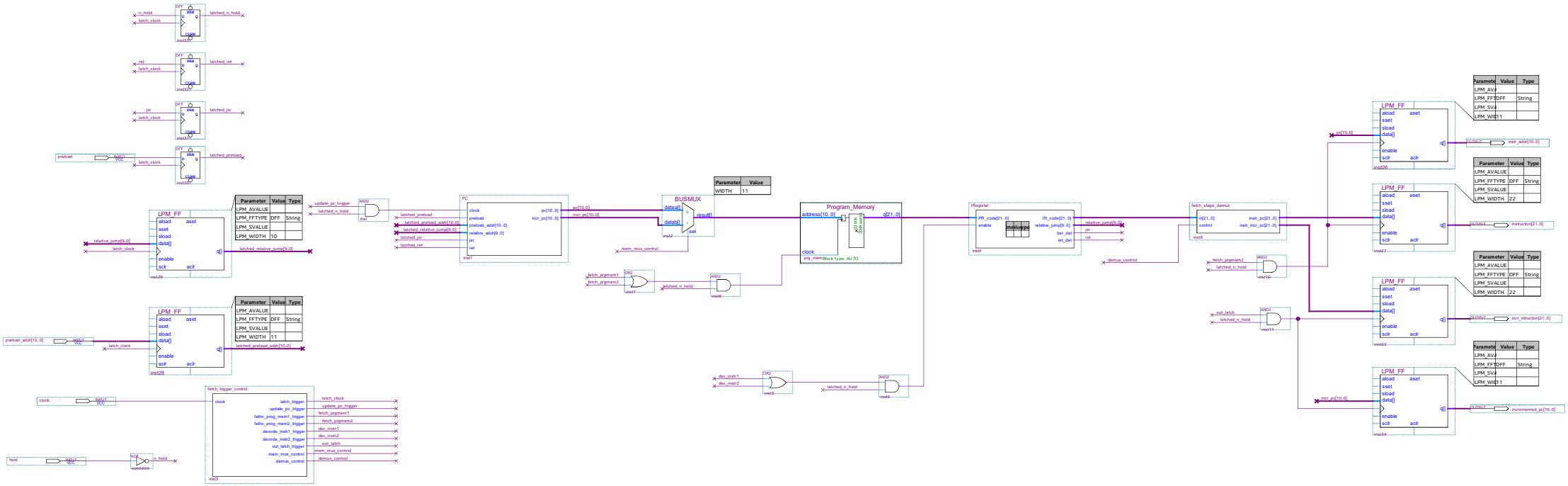


FIGURE 3.4: IRegister block


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
fethc_prog_mem1_trigger
fethc_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

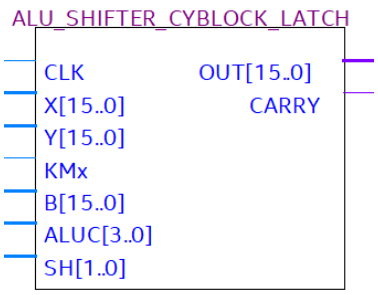


FIGURE 4.1: ALU Block

The block in Figure 4.1 implements the arithmetic-logic unit of the processor. Table 3 details the various operations available, depending on their control signal $ALUC$.

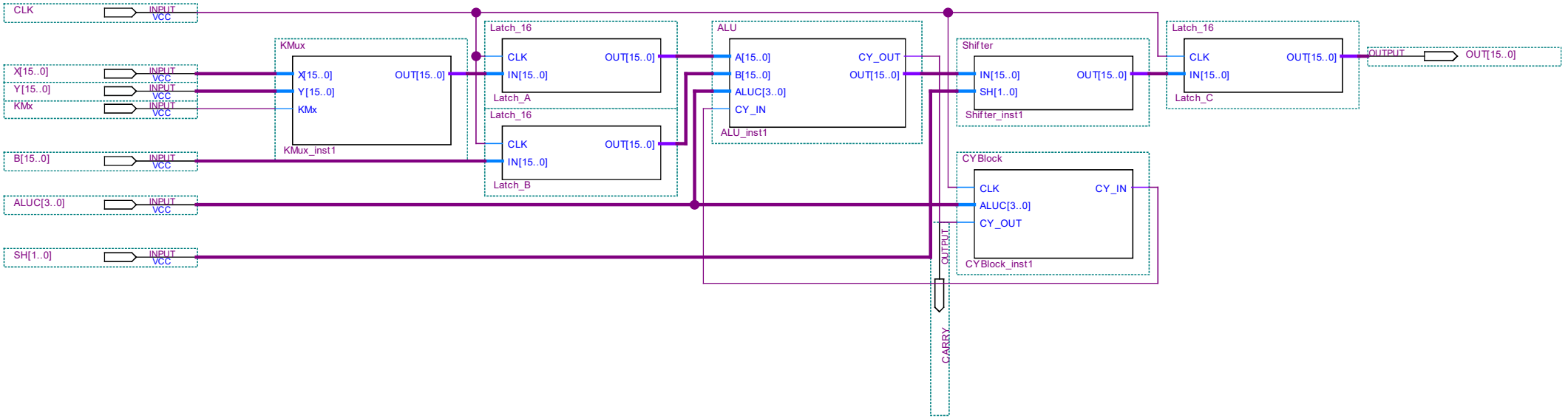
ALUC	Operation ALU
0	$Z = A$
1	$Z = B$
2	$Z = /A$
3	$Z = /B$
4	$Z = A + B$
5	$Z = A + B + CY$
6	$Z = A$ OR B
7	$Z = A\&B$
8	$Z = 0$
9	$Z = 1$
10	$Z = 0xFFFF$
11	$CY = 0$
12	$CY = 1$

TABLE 3: ALU block operations

5 Shifter

The shifter block implements shifts of the logical type to the right or left depending on the control signal it receives in the operand stage.

SH	Operation 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 Dynamic jump predictor

One of the additional features to the design presented by the chair that this implementation presents is a dynamic jump predictor. The purpose of the predictor is to anticipate conditional jumps without stopping the pipeline flow. The design of the predictor allows the prediction result to be conditional on the results of previous executions for a particular jump instruction. The block performs two basic functions: predict and update.

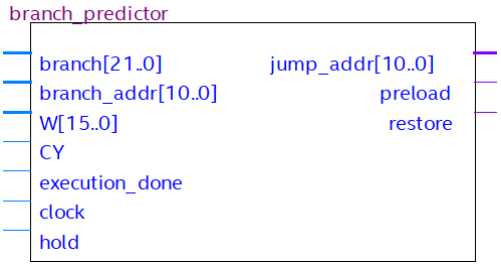


FIGURE 6.1: Jump predictor block

When the block is required to predict the result of a conditional jump, it must be provided with the binary of the conditional jump instruction *branch[21..0]* and the address of the jump instruction $branch_addr[10..0].The block will indicate in its output preload if a jump is to be made to a given address in the output$

This block receives the 3 most significant bits of the conditional jump instruction, which uniquely define whether the instruction is a conditional jump, and of what type (JMP, PCY, JPO or JZE). The output *predict* indicates whether a prediction is to be made on the instruction received. The output *unconditional jump* indicates whether the instruction is of the unconditional jump type.

6.1 Branch history table

The jump history table (or BHT) is the core module of the predictor. The table has 2^8 entries, indexed by the least significant 8 bits of the address of the jump instruction. Each entry stores the information relevant to a jump instruction: the integer address of the instruction and the state of the jump history, encoded in 2 bits, which is used to determine the result of the jump prediction recorded in the corresponding table entry.

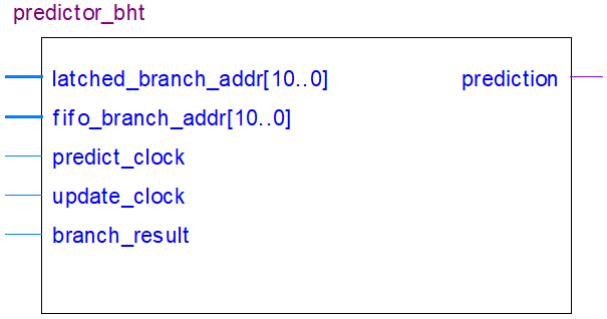
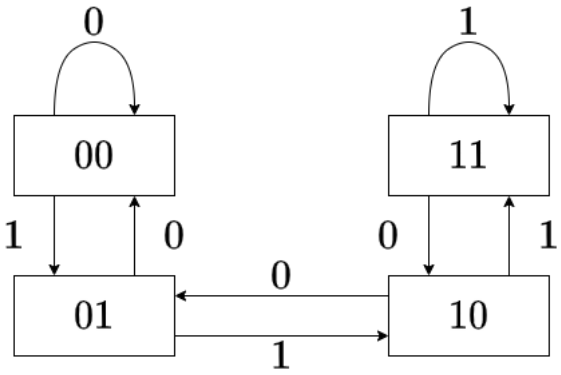


FIGURE 6.2: BHT Block

Then, each time a prediction is requested for a particular jump (identified by its instruction address), the table is indexed with the least significant 8 bits of the jump instruction address and it is verified that the integer address of the instruction to be predicted matches the one stored in the table. If it matches, the state of the jump history is decoded to determine the value of the prediction. If the address stored in the corresponding register does not match the address of the jump to be predicted, a default prediction value is taken.



00 : Strongly Not-Taken

01 : Weakly Not-Taken

10 : Weakly Taken

11 : Strongly Taken

FIGURE 6.3: FSM of the jump history

On the other hand, each time the result of the execution of the jump condition of a conditional jump is available, this table must be updated. In this case, the address of the instruction of the jump to be updated is taken from a FIFO memory (to be discussed later) and the result of the execution of the jump condition to modify the jump history of the corresponding jump, as shown in Figure 6.3.

6.2 FIFO and Prediction Check

Each time the block receives a prediction request, in addition to performing the prediction through the BHT as explained above, the information relevant to the prediction must also be stored temporarily until the execution of the jump condition is resolved. For this purpose, a FIFO is used which for each prediction stores the address of the jump instruction, the jump address, the jump type and the result of the prediction.

In this way, when an update request arrives, the value of the prediction result stored in the FIFO is contrasted to determine the success of the prediction, and the address of the jump instruction to update the BHT.

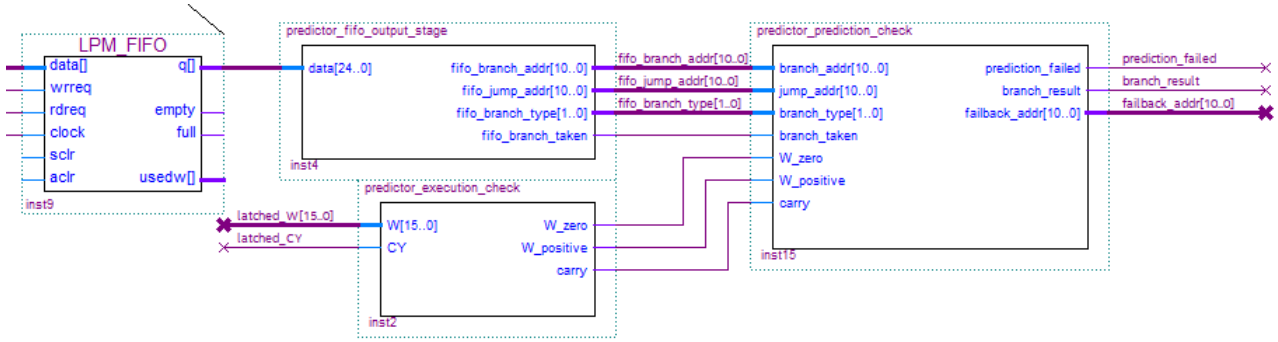
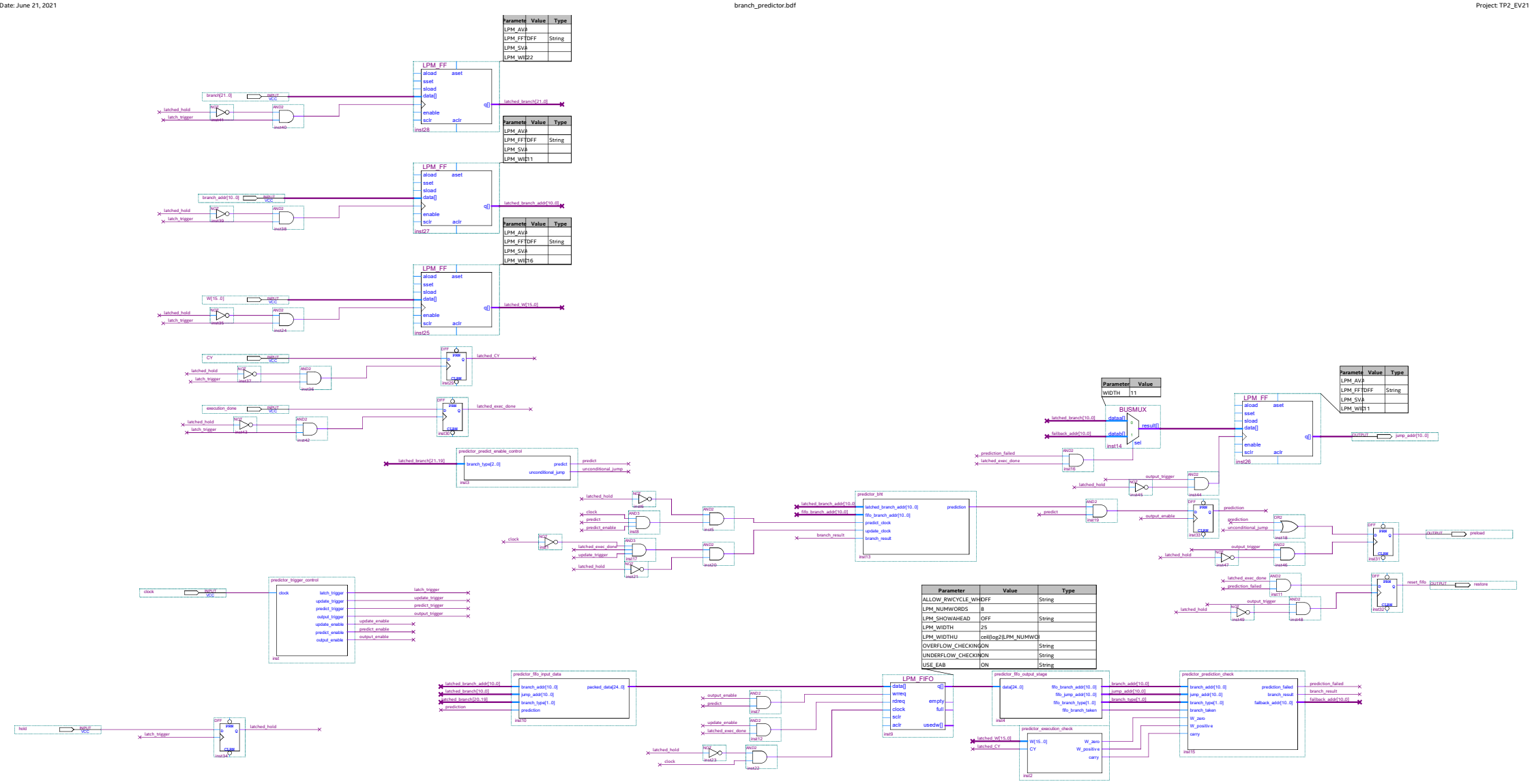


FIGURE 6.4: FIFO and Prediction Check blocks



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 Register Bank

The design has a register bank that handles the flow control from the ALU output bus to the general purpose registers, output ports and data memory, as well as the data flow from registers, ports and memories to the ALU input buses. As mentioned above, 16-bit data is handled.

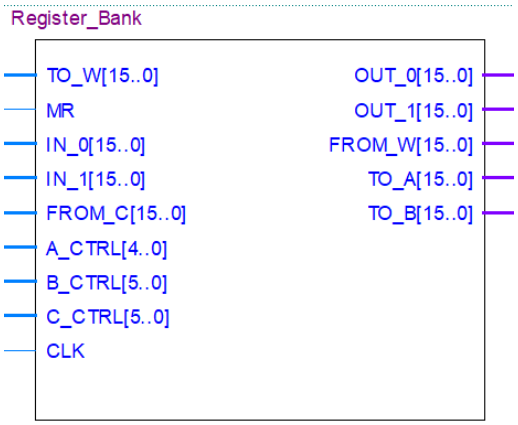


FIGURE 7.1: Register Bank Block

8 VGA implementation

We proceeded to implement a block compatible with a *VGA* monitor with a resolution of $640x480$ pixels. For this, a clock running at 25 MHz was used, and it also has a reset signal in case you want to restart the screen. The block has as output a horizontal synchronization signal, a vertical one and 2 control signals that indicate the X and Y coordinates of the pixel that is being updated at that moment.

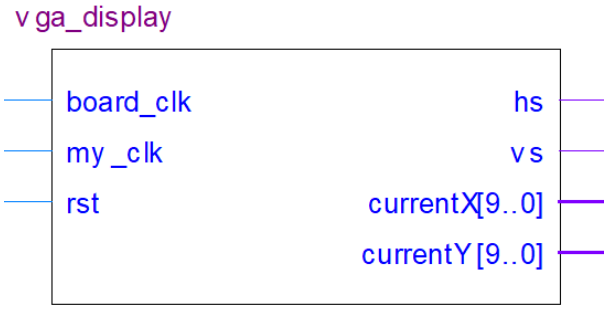


FIGURE 8.1: VGA protocol module

9 Measurements

9.1 IP to PO copier

A program was written to read data from the input port and write it to the output port. Figure 9.1 shows the results obtained, measuring the least significant 8 bits of both ports. With the help of an Analog Discovery, an up-counter was placed on the PI input port.

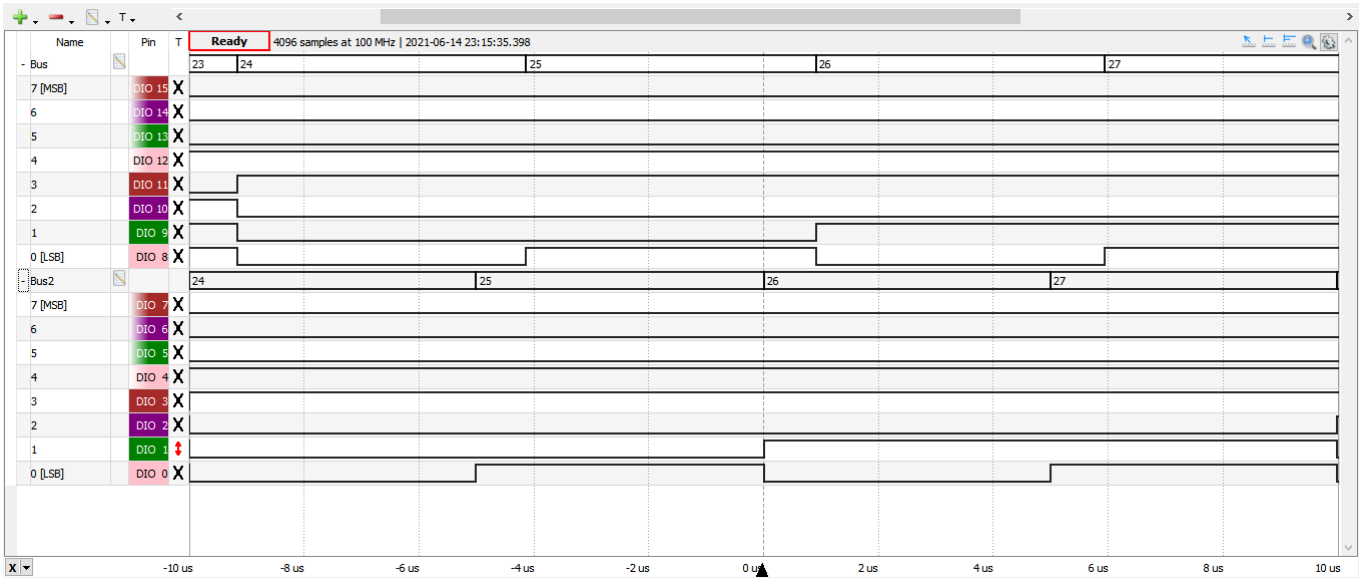


FIGURE 9.1: Signals recorded in Analog Discovery

9.2 Counter in W register

A program was written to implement a counter in register W. A value is loaded into memory, register W is set to zero, and then W is incremented with the value stored in memory. Then an unconditional jump to the increment instruction is executed. The W register was measured with the help of an Analog Discovery and the results are shown in Figure 9.2.

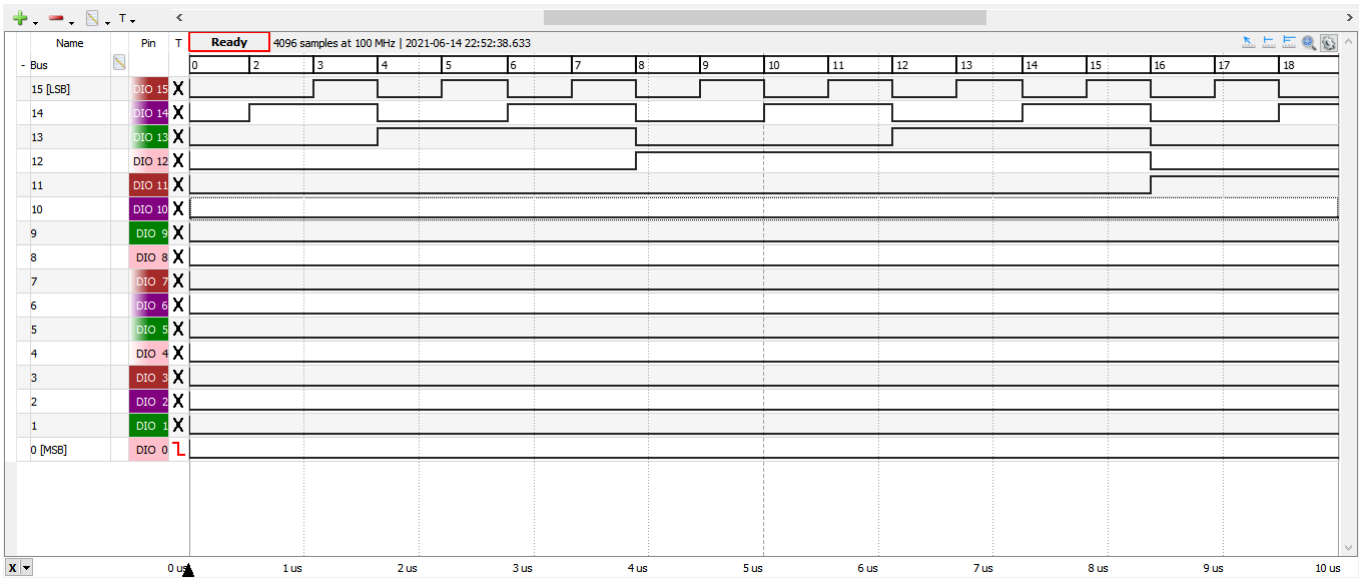


FIGURE 9.2: Signals recorded in Analog Discovery