Práctica Especial: Microprocesador MIPS Segmentado

Desarrollo

En primer lugar, aprovechamos la ALU desarrollada anteriormente para la materia Diseño Lógico, y la adaptamos a las necesidades del presente trabajo. Tomamos como referencia el diagrama de la figura 4.51 del libro *Estructura y diseño de computadores: la interfaz hardware/software* utilizado como bibliografía durante la cursada, y codificamos todo el procesador.

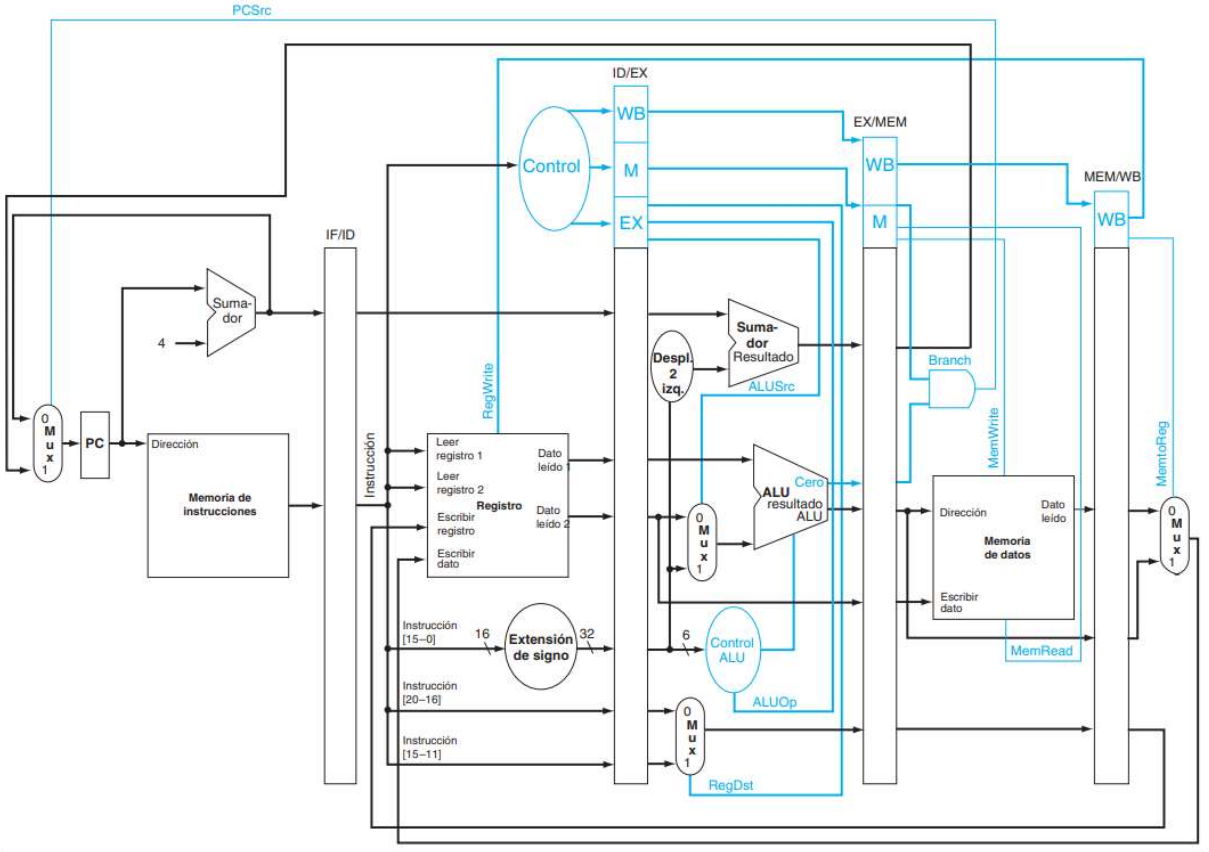


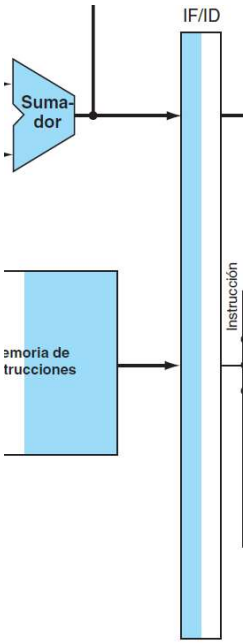
FIGURA 4.51 Camino de datos segmentado de la figura 4.40, con las señales de control conectadas a la parte de control de los registros de segmentación. Los valores de control de las tres últimas etapas se crean durante la descodificación de la instrucción y son escritos en el registro de segmentación ID/EX. En cada etapa de segmentación se usan ciertas líneas de control, y las líneas restantes se pasan a la etapa siguiente.

Consideraciones de diseño

Registros de Segmentación

Para modelar los registros de segmentación utilizamos procesos que responden al clock. Dichos procesos están compuestos por señales que permiten “persistir” los datos presentes en dichos registros. En cada ciclo de reloj efectuamos el pase entre los datos que poseen las señales de cada etapa.

A continuación se muestra el diagrama del registro de segmentación y su código asociado. Como vemos el registro maneja dos señales, el PC+4 proveniente de la etapa IF (IF_pc_4) que pasa a la etapa ID (ID_pc_4), y la instrucción que se obtiene de la memoria de instrucciones (I_DataIn) que pasa a la etapa ID (ID_instruction).




```
-------------------------------------------
-- REGISTRO DE SEGMENTACION IF/ID
-------------------------------------------
IF_ID : PROCESS (Clk)
BEGIN
    IF reset = '1' THEN
        ID_pc_4 <= (OTHERS => '0');
    ELSIF rising_edge(Clk) THEN
        ID_pc_4 <= IF_pc_4;
        ID_Instruction <= I_DataIn;
    END IF;
END PROCESS;
```

Unidad de control

Para la unidad de control recurrimos a una solución que permitió escribir código VHDL más compacto, favoreciendo la legibilidad. Para ello decidimos utilizar una señal auxiliar que contenga todos los valores de las señales de control bit a bit. Se utilizaron 2 procesos, uno que asigna el valor de todas las señales de control involucradas, y otro proceso que asigna el valor correspondiente a cada señal individual, accediendo a los bits (bit_index) correspondientes de la señal auxiliar mediante la sintaxis: aux_control(bit_index)
A continuación se muestra el código de la unidad:

```
196     -- CONTROL UNIT
197     PROCESS (ID_Instruction)
198     BEGIN
199         IF (ID_Instruction(31 DOWNTO 26) = "000000") THEN -- Tipo R
200             aux_control <= "1010000011";
201         ELSIF (ID_Instruction(31 DOWNTO 26) = "100011") THEN -- LW
202             aux_control <= "0000101010";
203         ELSIF (ID_Instruction(31 DOWNTO 26) = "101011") THEN -- SW
204             aux_control <= "0000100101";
205         ELSIF (ID_Instruction(31 DOWNTO 26) = "000100") THEN -- BEQ
206             aux_control <= "0001010001";
207         ELSIF (ID_Instruction(31 DOWNTO 26) = "001111") THEN -- LUI
208             -- Señales de control de LUI
209             aux_control <= "0100100011";
210         ELSIF (ID_Instruction(31 DOWNTO 26) = "001000") THEN -- ADDI
211             -- Señales de control de ADDI
212             aux_control <= "0101100011";
213         ELSIF (ID_Instruction(31 DOWNTO 26) = "001100") THEN -- ANDI
214             -- Señales de control de ANDI
215             aux_control <= "0110100011";
216         ELSIF (ID_Instruction(31 DOWNTO 26) = "001101") THEN -- ORI
217             -- Señales de control de ORI
218             aux_control <= "0111100011";
219         ELSE
220             aux_control <= "0000000000";
221         END IF;
222     END PROCESS;
223
224     PROCESS (aux_control)
225     BEGIN
226         -- Control etapa EX
227         ID_control_reg_dst <= aux_control(9);
228         ID_control_alu_op <= aux_control(8 DOWNTO 6);
229         ID_control_alu_src <= aux_control(5);
230         -- Control etapa MEM
231         ID_control_branch <= aux_control(4);
232         ID_control_mem_read <= aux_control(3);
233         ID_control_mem_write <= aux_control(2);
234         -- Control etapa WB
235         ID_control_reg_write <= aux_control(1);
236         ID_control_mem_to_reg <= aux_control(0);
237
238     END PROCESS;
239     ----------------------------------------------------------------
```

Una vez codificado todo el procesador iniciamos las pruebas utilizando el *testbench* provisto.

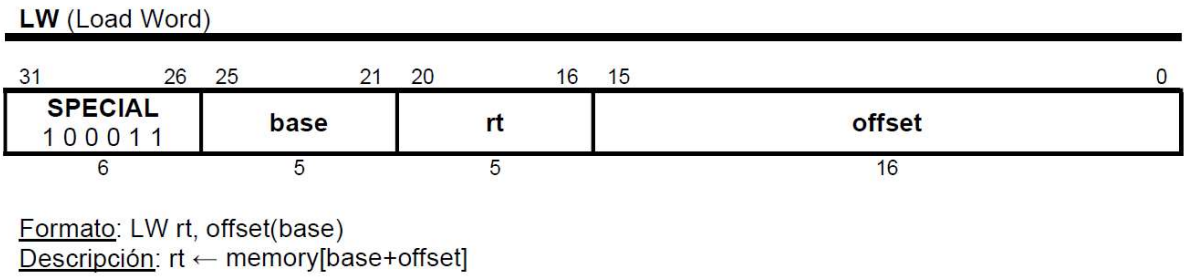
Pruebas

En primera instancia validamos que las instrucciones lleguen correctamente al registro de instrucciones. En la siguiente figura se puede ver cómo avanza el PC y el impacto en el ciclo siguiente de la etapa ID.

Luego procedimos a analizar la ejecución del *program2* instrucción por instrucción.

La primera instrucción es: 8c090000 → 100011 00000 01001 0000000000000000

Por los bits designados a la sección SPECIAL, sabemos que es una instrucción de tipo LW, y a partir de la estructura que se muestra en la figura debajo, podemos deducir que en el registro $9 se guarda el dato de la memoria en la posición $0 porque el offset = 0 y base = 0. Finalmente, la sintaxis es:
LW 9, 0(0).



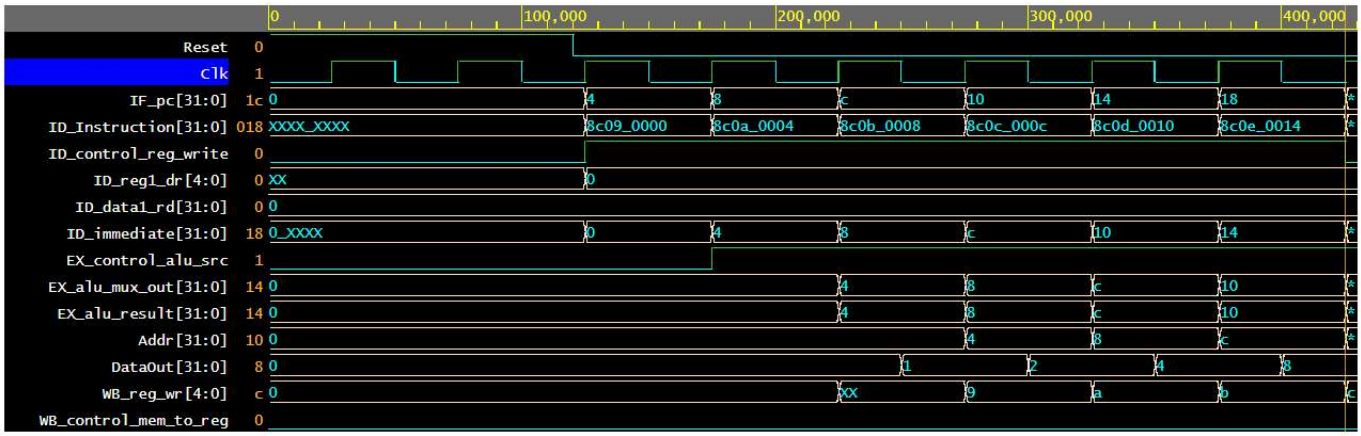
Formato: LW rt, offset(base)
Descripción: rt ← memory[base+offset]

De la misma manera analizamos el resto de las instrucciones, de las cuales se muestran las primeras 18 en la siguiente tabla.

	Instrucción		special	base	rt	offset	sintaxis
1	8c090000	100011 00000 01001 0000000000000000	LW	$0	9	0	LW 9, 0($0)
2	8c0a0004	100011 00000 01010 0000000000000100	LW	$0	10	4	LW 10, 4($0)
3	8c0b0008	100011 00000 01011 0000000000001000	LW	$0	11	8	LW 11, 8($0)
4	8c0c000c	100011 00000 01100 0000000000001100	LW	$0	12	12	LW 12, 12($0)
5	8c0d0010	100011 00000 01101 0000000000010000	LW	$0	13	16	LW 13, 16($0)
6	8c0e0014	100011 00000 01110 0000000000010100	LW	$0	14	20	LW 14, 20($0)
7	ac090018	101011 00000 01001 0000000000011000	SW	$0	9	24	SW 9, 24($0)
8	ac0a001c	101011 00000 01010 0000000000011100	SW	$0	10	28	SW 10, 28($0)
9	ac0b0020	101011 00000 01011 0000000000100000	SW	$0	11	32	SW 11, 32($0)
10	ac0c0024	101011 00000 01100 0000000000100100	SW	$0	12	36	SW 12, 36($0)
11	ac0d0028	101011 00000 01101 0000000000101000	SW	$0	13	40	SW 13, 40($0)

12	ac0e002c	101011 00000 01110 0000000000101100	SW	$0	14	44	SW 14, 44($0)
13	8c090018	100011 00000 01001 0000000000011000	LW	$0	9	24	LW 9, 24($0)
14	8c0a001c	100011 00000 01010 0000000000011100	LW	$0	10	28	LW 10, 28($0)
15	8c0b0020	100011 00000 01011 0000000000100000	LW	$0	11	32	LW 11, 32($0)
16	8c0c0024	100011 00000 01100 0000000000100100	LW	$0	12	36	LW 12, 36($0)
17	8c0d0028	100011 00000 01101 0000000000101000	LW	$0	13	40	LW 13, 40($0)
18	8c0e002c	100011 00000 01101 0000000000101000	LW	$0	13	40	LW 13, 40($0)

A continuación, se muestran las señales clave involucradas para las instrucciones del tipo LW.



Observamos por el código *special* que las primeras 6 instrucciones se corresponden con el tipo LW. Para validar el funcionamiento del procesador se realizó un análisis paso a paso de las señales.

Para la señal de control de escritura en el banco de registros (ID_control_reg_write), se observa que correctamente toma el valor ‘1’ para las 6 instrucciones que escriben en el banco de registros. Luego toma el valor ‘0’ porque las siguientes 6 instrucciones son del tipo SW, por lo tanto no escriben en el banco de registros.

Además, observamos que la dirección de entrada al banco de registros (en el diagrama: Leer registro 1) es $0, proveniente de la sección “base” de la instrucción. Esto se repite para las 6 instrucciones iniciales de LW. A su vez, la salida del banco de registros (ID_data1_rd), toma el valor de ‘0’, dado que es el dato almacenado en la dirección $0. Se entiende que inicialmente el banco de registros está vacío, por lo tanto es correcto que el dato sea cero.

En cuanto al offset, el mismo proviene de la sección inmediata de la instrucción (ID_immediate). Los valores que se muestran en el gráfico de la simulación coinciden con los datos provenientes de la instrucción que se muestran en la tabla.

La señal EX_control_alu_src, le indica al MUX de entrada de la ALU, que habilite la señal proveniente del offset a la entrada B de la ALU. Por lo tanto, es correcto que tome el valor ‘1’ para el bloque de instrucciones bajo análisis. También se puede observar que a la salida del MUX previo a la ALU, particularmente la señal “EX_alu_mux_out”, toma el valor correcto que es el mismo que el valor inmediato.

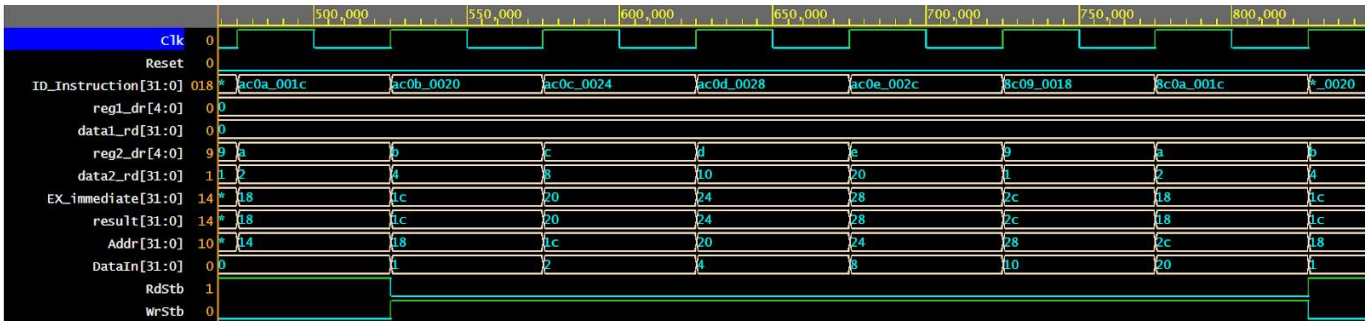
En cuanto al resultado de la ALU (EX_alu_result), también se observa un comportamiento correcto. Para el bloque bajo análisis, siempre se realiza la suma de offset + base, donde base siempre es 0, por lo tanto se da que: ALU result = offset. La dirección Addr, de lectura de la memoria, va tomando un ciclo más adelante el valor de ALU result, esto también se aprecia en la simulación.

“DataOut” contiene la salida de la memoria de datos, se observa que a medida que avanzan los ciclos los datos coinciden con los valores presentes en el archivo proporcionado.

En cuanto a la última etapa de WB, se observa que la señal de control mem_to_reg toma el valor de ‘0’ para todo el bloque bajo análisis, lo cuál es correcto porque al banco de registros debe llegar el dato proveniente de la memoria de datos.

Finalmente, la señal WB_reg_wr, contiene la dirección del banco de registros a donde se va a escribir el dato proveniente de la memoria de datos. Se observa que ciclo a ciclo va tomando los valores correspondientes a rt, presentes en la instrucción.

De manera similar se realizó el análisis de las instrucciones de SW, para lo cual las señales observadas son: reset, clock, instrucción (ID_Instruction), las direcciones de lectura del banco de registros (reg1_dr y reg2_dr), los datos leídos del banco de registros (data1_rd y data2_rd), el inmediato (EX_Immediate), la salida de la ALU (result) y por último las entradas y salidas de la memoria de datos (Addr, DataIn, RdStb y WrStb).



Se observa que el valor inmediato y el resultado de la ALU coinciden, lo cual es correcto porque en todas las instrucciones la base es 0. Este valor impacta en el siguiente ciclo en la dirección de memoria donde se guarda el dato presente en la señal DataIn, el cual es el mismo que el valor leído del banco de registro dos ciclos antes. Por último, se observa que las señales de control de lectura y escritura de la memoria de datos pasan a 0 y 1 respectivamente.

Las instrucciones número 13 a 18 son del tipo LW y las señales se comportan de la manera ya descrita anteriormente.

Las próximas 10 instrucciones son del tipo R, y su análisis se muestra en la siguiente tabla:

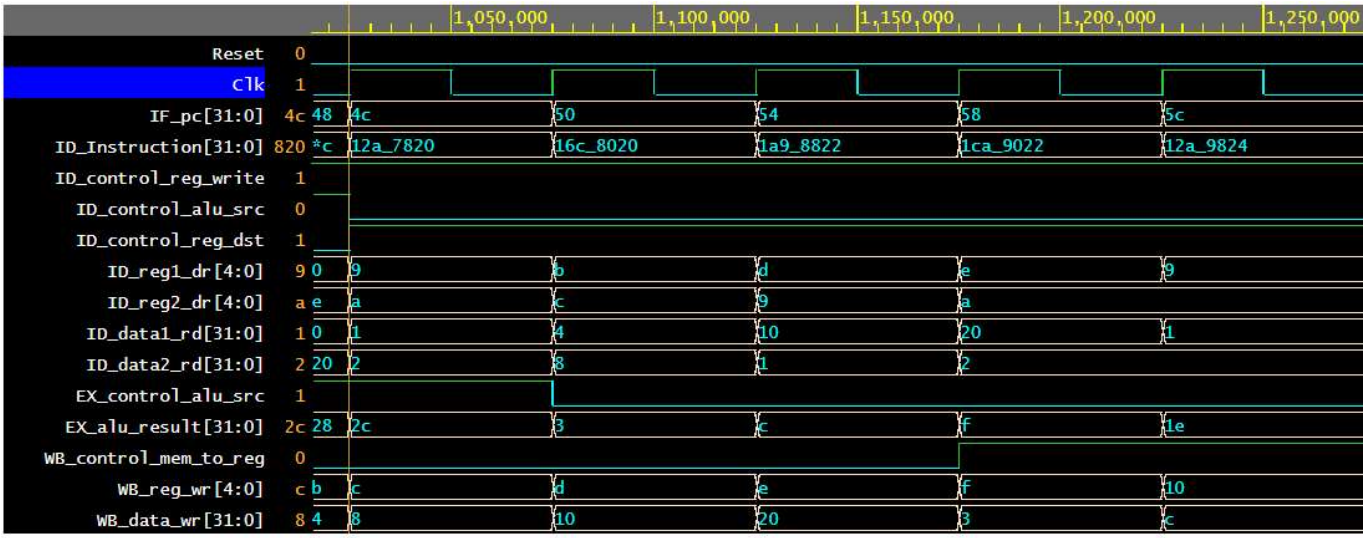
Instrucción			op	rs	rt	rd	shamt	funct	sintaxis
19	012a7820	000000 01001 01010 01111 00000 100000	0	9	10	15	0	ADD	ADD 15, 9, 10
20	016c8020	000000 01011 01100 10000 00000 100000	0	11	12	16	0	ADD	ADD 16, 11, 12
21	01a98822	000000 01101 01001 10001 00000 100010	0	13	9	17	0	SUB	SUB 17, 13, 9
22	01ca9022	000000 01110 01010 10010 00000 100010	0	14	10	18	0	SUB	SUB 18, 14, 10
23	012a9824	000000 01001 01010 10011 00000 100100	0	9	10	19	0	AND	AND 19, 9, 10
24	01eaa024	000000 01111 01010 10100 00000 100100	0	15	10	20	0	AND	AND 20, 15, 10
25	012aa825	000000 01001 01010 10101 00000 100101	0	9	10	21	0	OR	OR 21, 9, 10
26	020ab025	000000 10000 01010 10110 00000 100101	0	16	10	22	0	OR	OR 22, 16, 10
27	012ab82a	000000 01001 01010 10111 00000 101010	0	9	10	23	0	SLT	SLT 23, 9, 10
28	020ac02a	000000 10000 01010 11000 00000 101010	0	16	10	24	0	SLT	SLT 24, 16, 10

La instrucción 19 es de tipo ADD y realiza lo siguiente:

$15 = $9 + $10

En la dirección $15 del banco de registros, guarda la suma del contenido en la dirección $9 y del contenido en la dirección $10.

De instrucciones anteriores (1 y 2) de LW se debería haber cargado el valor 1 en la dirección $9 y el valor 2 en la posición $10, por lo tanto a la salida de la ALU en el ciclo siguiente debería haber un 3 como dato.



Nuevamente para validar el comportamiento correcto del procesador analizamos las señales:

ID_control_reg_write indica al banco de registros que se va a escribir, por lo tanto es correcto que tome el valor ‘1’

ID_control_alu_src toma el valor ‘0’ lo cual es correcto ya que el MUX de entrada a la ALU debe dejar pasar el contenido proveniente de ID_data2_rd, es decir la salida 2 del banco de registros.

ID_control_reg_dst es correcto que tome el valor de ‘1’, ya que para las instrucciones de tipo R, el registro destino se ubica en los bits 15 a 11.

Validamos correctamente que ID_reg1_dr y ID_reg2_dr se corresponden con $9 (x9) y $10 (xA) respectivamente.

A su vez, los datos leídos del banco de registros (ID_data1_rd, ID_data_2_rd) se corresponden con los datos esperados 1 y 2.

En el ciclo siguiente, y en la etapa EX, observamos correctamente que EX_control_alu_src toma el valor ‘0’ para dejar pasar el dato proveniente de la salida dos del banco de registros. También se valida correctamente que (EX_alu_result) en este mismo ciclo toma el valor 3, resultado de la suma.

Finalmente en la etapa WB, 3 ciclos más adelante respecto de la etapa ID, se observa que la señal WB_control_mem_to_reg toma el valor ‘1’, habilitando al MUX de la etapa WB para que deje pasar el dato proveniente de la salida de la ALU, es decir el resultado de la suma. Se observa que la señal WB_reg_wr toma el valor xF, correspondiente a $15 y WB_data_wr toma el valor de 3, es decir el resultado de la suma que se va a escribir en la posición $15.

De manera similar la instrucción 20, realiza:

$16 = $11 + $12

En $11 y $12, se cargaron los siguientes datos como resultado de las instrucciones 3 y 4.

$11 = memory[8] = 4

$12 = memory[12] = 8

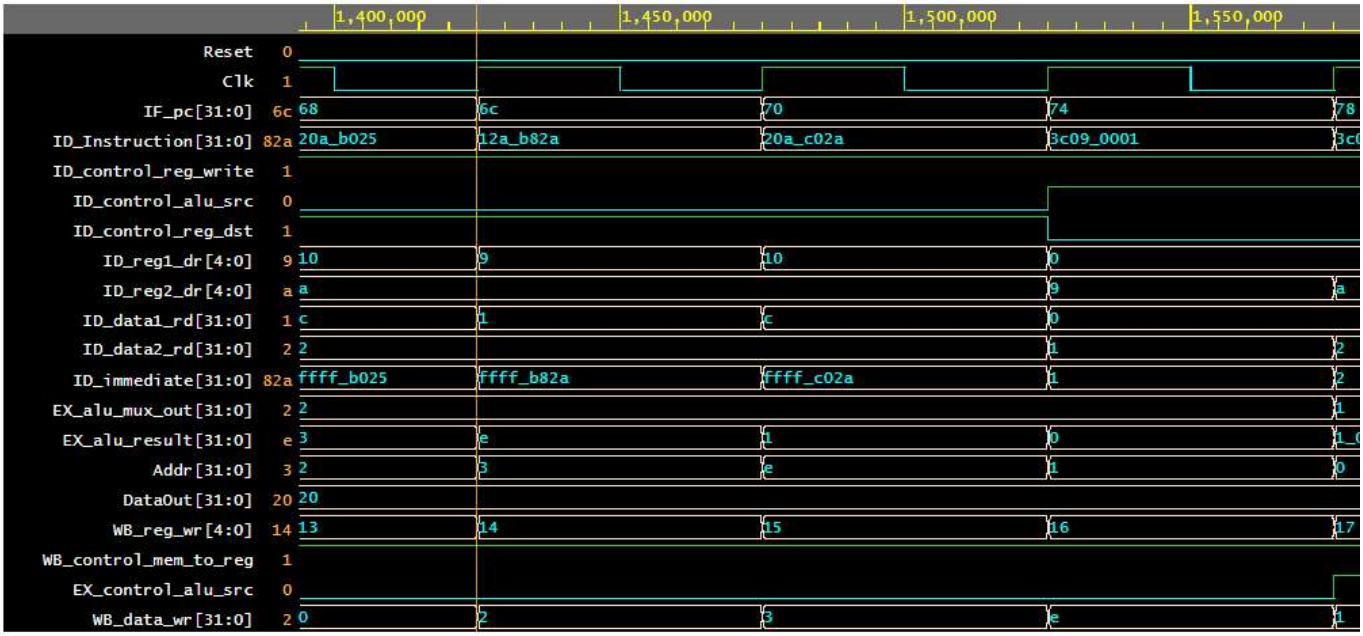
Guardando en $16, el valor 12. (4 + 8)

El análisis de las señales es idéntico al de la instrucción anterior por tratarse del mismo tipo, sin embargo se destaca que:

La señal WB_reg_wr toma el valor x10, correspondiente a $16 y WB_data_wr toma el valor de 12 (xC), es decir el resultado de la suma que se va a escribir en la posición $16.

Se analizaron el resto de las instrucciones del tipo R de manera similar, corroborando el correcto funcionamiento.

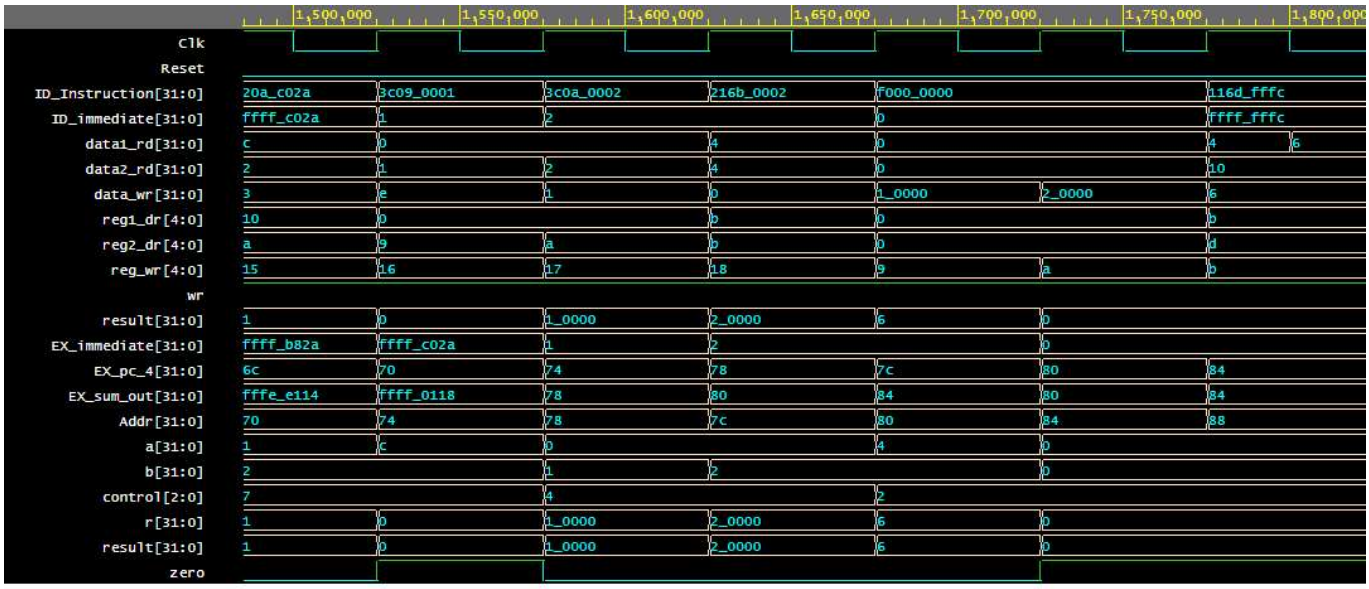
En el caso de la instrucción SLT (n° 27), se verificaron los siguientes valores:



La instrucción va a comparar si rs < rt, es decir si el contenido en $9 es menor que en $10. Los datos en esos registros son x1 y x2 respectivamente (ID_data_1_rd, ID_data_2_rd). En el ciclo siguiente se observa correctamente que EX_alu_result da 1, es decir true, ya que x1 es menor a x2, este resultado (WB_data_wr) se guarda dos ciclos más adelante en la etapa WB, en la dirección x17 (WB_reg_wr), equivalente a $23.

El análisis de las últimas 11 señales se muestra en la tabla a continuación:

Instrucción			special	rs	rt	immediate	sintaxis
29	3c090001	001111 00000 01001 0000000000000001	LUI	0	9	1	LUI 9, 1
30	3c0a0002	001111 00000 01010 0000000000000010	LUI	0	10	2	LUI 10, 2
31	216b0002	001000 01011 01011 0000000000000010	ADDI	11	11	2	ADDI 11, 11, 2
32	f0000000	111100 00000 00000 0000000000000000	-				
33	f0000000	111100 00000 00000 0000000000000000	-				
34	116dfffc	000100 01011 01101 1111111111111100	BEQ	11	13	65532	BEQ 11, 13, 65532
35	f0000000	111100 00000 00000 0000000000000000	-				
36	f0000000	111100 00000 00000 0000000000000000	-				
37	f0000000	111100 00000 00000 0000000000000000	-				
38	35ac0139	001101 01101 01100 0000000100111001	ORI	13	12	313	ORI 12, 13, 313
39	322d0139	001100 10001 01101 0000000100111001	ANDI	17	13	313	ANDI 13, 17, 313



En la instrucción 29, se guardará en el registro $9 el resultado del desplazamiento 1 << 16, el cual da x65536 (b10000000000000000). En el segundo ciclo se ve en la señal result que la salida de la ALU da el resultado correcto. Este impacta en el registro $9 (señal reg_wr) dos ciclos después en la señal data_wr, mientras que wr habilita la escritura. De manera similar trabaja la siguiente instrucción.

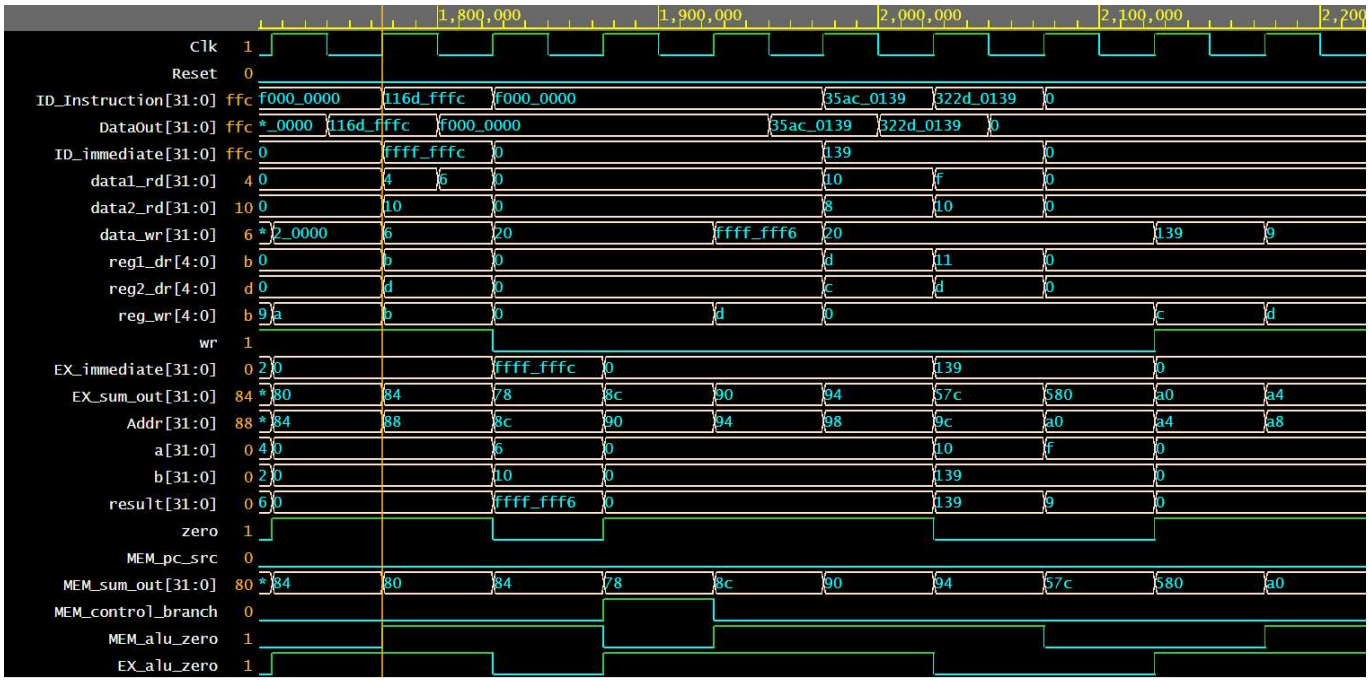
La instrucción ADDI realiza la siguiente sumatoria y la guarda en la dirección rt dos ciclos después:

rt ← rs + SIGN_EXTEND(immediate)

En este caso se observa que se realiza la suma del valor x0 leído en la dirección $11, y el valor inmediato x2. Se puede ver el resultado x6 a la salida de la ALU (result) en el quinto ciclo que se visualiza en la simulación (1 ciclo después de la decodificación de la instrucción ADDI). Y dos ciclos después, habiendo ya pasado por la etapa MEM, en WB se realiza la escritura en el banco de registros, con el control de escritura wr en 1, el dato x6 en data_wr, en la dirección $11 (xb) en reg_wr.

Resulta de mucha importancia aclarar que para evitar riesgos de datos, se agregaron al programa las instrucciones 32 y 33, ya que la 34 necesita leer el registro destino de la 31. Además, se incorporaron las instrucciones 35, 36 y 37 para que no existan riesgos de control luego del salto condicional.

A continuación se muestran las señales analizadas para la instrucción BEQ:



En el segundo ciclo en reg1_dr y reg2_dr se ven las direcciones de memoria que se quieren leer ($11 y $13). En el tercer ciclo los datos leídos son x6 y x10 (data1_rd y data2_rd respectivamente). A su vez, se guarda en EX_sum_out el resultado de la operación expresada debajo, sin embargo no se utiliza porque no se produce el salto ya que son distintos (x6 ≠ x10) .

(EX_immediate << 2) + PC + 4

(xFFFFFFFC << 2) + x88 = x78

El comportamiento de las señales para las dos instrucciones restantes (ORI Y ANDI) también se muestra en la última figura, y el análisis es similar al realizado anteriormente para la instrucción ADDI. Se comprobó que el funcionamiento es correcto.

A continuación dejamos el link a la simulación del *program2*: EDA Playground