

# Práctico N° 1: SV y diseño de un procesador de un ciclo

En el presente práctico se desarrollarán algunos módulos del procesador ARM de un ciclo presentado por Patterson y Hennessy en el libro “Computer organization and design - ARM Edition”, cuyo diagrama de bloques Top-level se muestra en la Fig. 1. Para esto, se pide crear un proyecto en Quartus, llamado **SingleCycleProcessor**, donde se guardarán todos los archivos correspondientes a la resolución de los ejercicios.

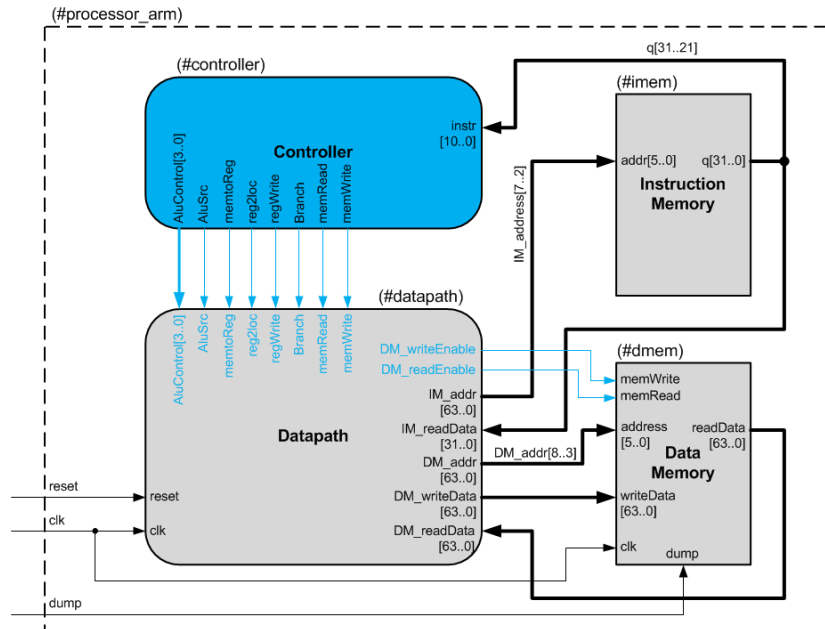


Fig.1. Top level entity: processor\_arm

Como se muestra en la Fig. 2, el módulo datapath se divide en cinco módulos, los cuales corresponden a las cinco etapas previstas para una futura implementación de pipeline: *fetch*, *decode*, *execute*, *memory* y *writeback*.

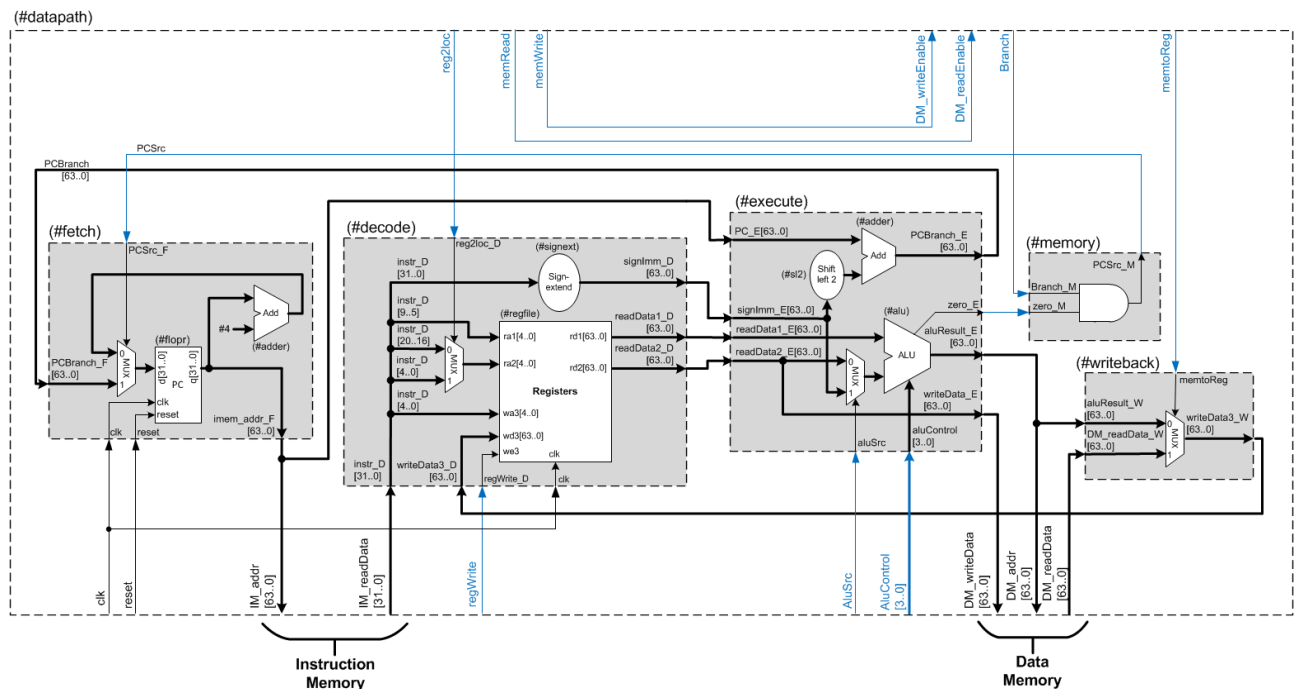


Fig.2. Module: datapath

Con fines de claridad y para unificar el nombre de las señales de todos los trabajos, la Fig. 3 muestra en detalle la implementación del módulo **controller**.

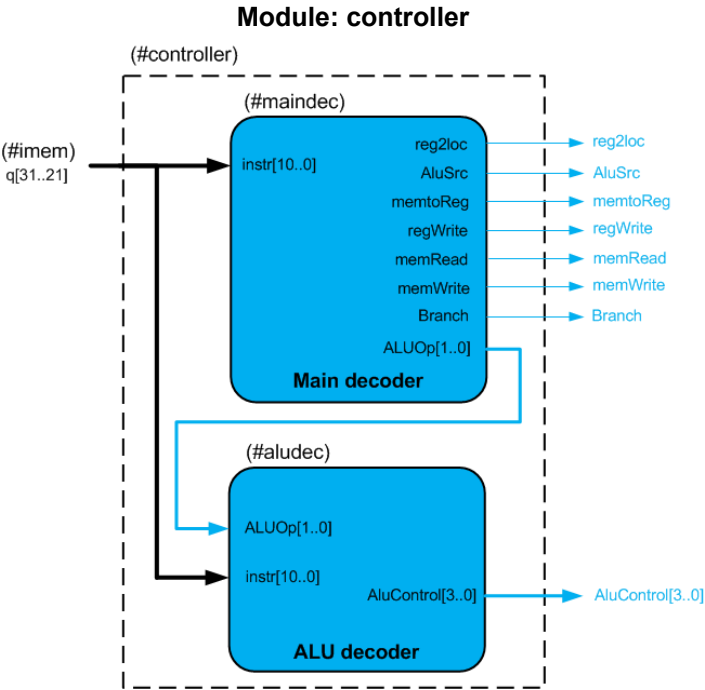


Fig.3. Module: controller

Instrucciones implementadas en el procesador:

Instruction	ALUOp	Instruction operation	Opcode field	Desired ALU action	ALU control input
LDUR	00	load register	XXXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXXX	pass input b	0111
R-type	10	ADD	10001011000	add	0010
R-type	10	SUB	11001011000	subtract	0110
R-type	10	AND	10001010000	AND	0000
R-type	10	ORR	10101010000	OR	0001

**IMPORTANTE:** Para todos los ejercicios se debe:

- Respetar los nombres de los módulos, y puertos de entradas y salidas de forma LITERAL (respetando mayúsculas y minúsculas).
- Los archivos de SystemVerilog (.sv) deben tener el mismo nombre que el módulo.
- Las señales de entrada y salida deben ser del tipo *logic* y se deben declarar en el orden en que están listadas.
- En todos los ejercicios el diseño comprende el desarrollo del módulo y la validación de su correcto funcionamiento mediante el uso de test bench.

**Ejercicio 1: flopr**

a) Implementar un Flip-Flop D con reset asíncrono, activo por alto, según el diagrama dado. Este módulo es utilizado para implementar el registro *Program Counter* (PC), determinando el comienzo de cada ciclo de instrucción. Utilizar código genérico para ampliar el diseño a registros de N bits (default N=64).

**Nombre del módulo:** flopr

**Puertos de entrada**

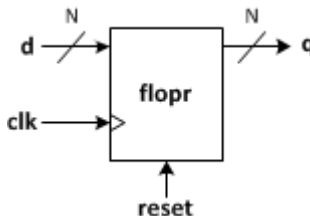
clk: 1 bit

reset: 1 bit

d: N bits (default: N=64)

**Puertos de salida**

q: N bits (default: N=64)



b) Realizar un test bench que:

- Instancie el módulo flopr con N=64.
- Genere una señal de reloj en el puerto **clk** del módulo, cuya frecuencia sea 100MHz (periodo de 10 ns).
- Por el puerto de entrada **d** ingrese 10 números distintos (de 64 bits c/u), uno por cada ciclo de reloj.
- Genere una señal de **reset** que permanezca en '1' durante el ingreso de los primeros 5 números y luego tome el valor '0' hasta finalizar la simulación.
- Verifique que durante los primeros 5 ciclos de clock la salida sea cero y en los 5 siguientes, se obtenga a la salida el valor ingresado. Note que la salida **q** debe actualizarse en el flanco de subida del **clk**.
- Finalmente, repetir el procedimiento anterior, pero instanciando el módulo *flopr* con N=32.

**Ejercicio 2: signext**

a) Diseñar el módulo de extensión de signo, que toma los 32 bits de instrucción (**a**), analiza el opcode y determina si la instrucción posee un valor inmediato. En caso afirmativo, toma los bits correspondientes a dicho inmediato y los convierte en un vector de 64 bits (**y**) signado o no, según corresponda al tipo de instrucción.

En el módulo diseñado se debe analizar el *Opcode* (11 bits más significativos) de las instrucciones implementadas en el procesador que poseen valores inmediatos (ver tabla a continuación), identificar cuáles son los bits correspondientes al campo de inmediato (ver Green card de LEGv8) y conectarlos a la salida del módulo, extendiendo el bit de signo hasta completar los 64 bits.

Para valores de entrada correspondientes a instrucciones sin valor inmediato, la salida del módulo signext debe ser **y** = "0".

Instruction	Instruction Operation	Opcode field
LDUR	Load Register	111_1100_0010
STUR	Store Register	111_1100_0000
CBZ	Compare and branch if zero	101_1010_0???

\* Considerar los "?" como condiciones sin cuidado.

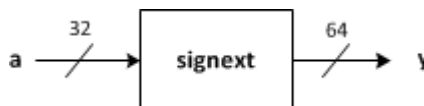
**Nombre del Módulo:** signext

**Puertos de entrada**

a: 32 bits

**Puertos de salida**

y: 64 bits



b) Realizar un testbench que:

- Ingrese por el puerto **a** todos los tipos de instrucciones detalladas en la tabla, con immediatos positivos y negativos, y verifique que la salida sea la correcta.
- Ingrese instrucciones que no estén en la tabla y verifique que la salida sea 0.

### Ejercicio 3: alu

a) Diseñar la Unidad Aritmética Lógica (ALU) según el diagrama, que realice las operaciones descritas en la tabla. Recordar que la señal de salida **zero** toma valor '1' cuando **result** es igual a "0".

Nombre del Módulo: alu

Puertos de entrada

a: 64 bits

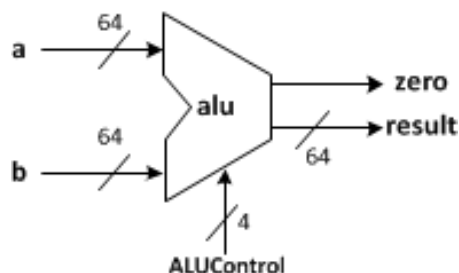
b: 64 bits

ALUControl: 4 bits

Puertos de salida

result: 64 bits

zero: 1 bit



ALU control lines	result
0000	a AND b
0001	a OR b
0010	add (a+b)
0110	sub (a-b)
0111	pass input b

b) Realizar un testbench que:

- Compruebe el funcionamiento de todas las operaciones para la totalidad de los siguientes casos: entre dos números positivos, entre dos números negativos y siendo uno positivo y uno negativo.
- Forzar una entrada que genere un overflow y analizar su comportamiento.
- Verifique que la bandera de **zero** tome valor '1', sólo cuando el resultado de cualquier operación sea igual a cero.

### Ejercicio 4: imem

a) Diseñar una memoria ROM que direcciona 64 palabras de N bits (default N=32). Inicializar la ROM para que contenga las instrucciones del programa dado y el espacio no utilizado en "0".

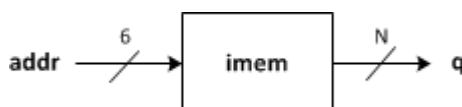
Nombre del Módulo: imem

Puertos de entrada

addr: 6 bits

Puertos de salida

q: N bits (default: N=32)



Programa:	Instrucciones ensamblada para inicializar la memoria:
<pre> STUR X1, [X0, #0] STUR X2, [X0, #8] STUR X3, [X16, #0] ADD X3, X4, X5 STUR X3, [X0, #24] SUB X3, X4, X5 STUR X3, [X0, #32] SUB X4, XZR, X10 STUR X4, [X0, #40] ADD X4, X3, X4 STUR X4, [X0, #48] SUB X5, X1, X3 STUR X5, [X0, #56] AND X5, X10, XZR STUR X5, [X0, #64] AND X5, X10, X3 STUR X5, [X0, #72] AND X20, X20, X20 STUR X20, [X0, #80] ORR X6, X11, XZR STUR X6, [X0, #88] ORR X6, X11, X3 STUR X6, [X0, #96] LDUR X12, [X0, #0] ADD X7, X12, XZR STUR X7, [X0, #104] STUR X12, [X0, #112] ADD XZR, X13, X14 STUR XZR, [X0, #120] CBZ X0, loop1 STUR X21, [X0, #128] loop1: STUR X21, [X0, #136]       ADD X2, XZR, X1 loop2: SUB X2, X2, X1       ADD X24, XZR, X1       STUR X24, [X0, #144]       ADD X0, X0, X8       CBZ X2, loop2       STUR X30, [X0, #144]       ADD X30, X30, X30       SUB X21, XZR, X21       ADD X30, X30, X20       LDUR X25, [X30, #-8]       ADD X30, X30, X30       ADD X30, X30, X16       STUR X25, [X30, #-8] finloop: CBZ XZR, finloop </pre>	<pre> ROM [0:46] = {32'hf8000001,               32'hf8008002,               32'hf8000203,               32'h8b050083,               32'hf8018003,               32'hcb050083,               32'hf8020003,               32'hcb0a03e4,               32'hf8028004,               32'h8b040064,               32'hf8030004,               32'hcb030025,               32'hf8038005,               32'h8a1f0145,               32'hf8040005,               32'h8a030145,               32'hf8048005,               32'h8a140294,               32'hf8050014,               32'haa1f0166,               32'hf8058006,               32'haa030166,               32'hf8060006,               32'hf840000c,               32'h8b1f0187,               32'hf8068007,               32'hf807000c,               32'h8b0e01bf,               32'hf807801f,               32'hb4000040,               32'hf8080015,               32'hf8088015,               32'h8b0103e2,               32'hcb010042,               32'h8b0103f8,               32'hf8090018,               32'h8b080000,               32'hb4ffff82,               32'hf809001e,               32'h8b1e03de,               32'hcb1503f5,               32'h8b1403de,               32'hf85f83d9,               32'h8b1e03de,               32'h8b1003de,               32'hf81f83d9,               32'hb400001f}; </pre>

## b) Realizar un testbench que:

- Ingrese a través del puerto **addr** las direcciones de las primeras 50 palabras en forma consecutiva y verifique que las primeras 47 palabras contienen el valor descrito en la columna "Instrucción ensamblada" y las 3 restantes contienen "0".

### Ejercicio 5: regfile

**a)** Diseñar el banco de 32 registros de 64 bits cada uno, con dos puertos de salida (**rd1** y **rd2**) y un puerto de escritura (**wd3**). Las señales de direccionamiento **ra1** y **ra2** determinan la posición de los datos de salida **rd1** y **rd2**, respectivamente. De forma análoga, el puerto **wa3** selecciona el registro en el que se almacenará el dato contenido de **wd3**.

**Nota 1:** El contenido del registro de dirección 31 (correspondiente a XZR) debe retornar siempre '0'. La escritura en este registro no debe estar permitida.

**Nota 2:** El dato contenido en **wd3** se guarda en la dirección determinada por **wa3** siempre que la señal **we3** tenga valor '1' y se detecte un flanco positivo de clock (escritura síncrona).

**Nota 3:** La lectura de los datos de salida **rd1** y **rd2** es asíncrona.

**Nota 4:** Inicializar los registros X0 a X30 con los valores 0 a 30 respectivamente.

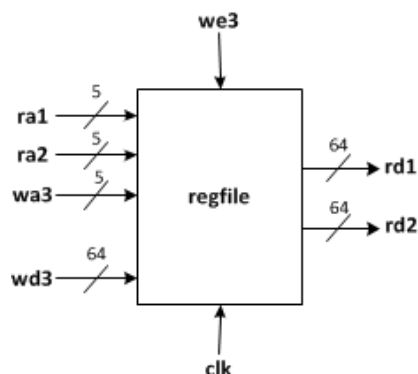
**Nombre del Módulo:** regfile

**Puertos de entrada**

clk: 1 bit  
 we3: 1 bit  
 ra1: 5 bits  
 ra2: 5 bits  
 wa3: 5 bits  
 wd3: 64 bits

**Puertos de salida**

rd1: 64 bits  
 rd2: 64 bits

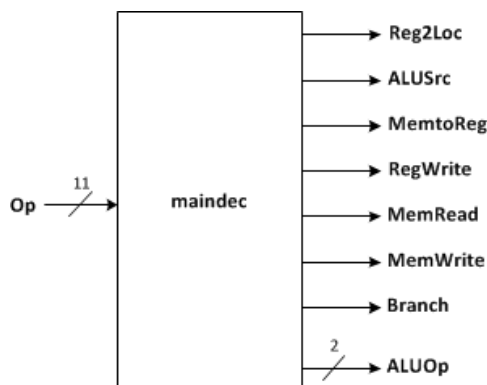


**b)** Realizar un testbench que:

- Genere una señal de reloj en el puerto **clk** del módulo, cuya frecuencia sea 100MHz (periodo de 10 ns).
- Direcciona los registros de 0 a 31 con **ra1** y **ra2** y verifique los valores de inicialización a la salida de los dos puertos luego del flanco descendente del **clk**.
- Escriba un valor en un registro (en el flanco positivo de clock y con **we3**= '1') y lo lea en el mismo ciclo de clock, verificando que se actualice correctamente la salida.
- Verifique que no se altere el contenido de un registro si **we3**= '0'.
- Escriba un valor distinto de cero en el registro X31 y verifique que la salida siempre permanezca en cero.

### Ejercicio 6: maindec

**a)** Diseñar el decodificador que genera las señales de control principal a partir del *Opcode* de la instrucción, según el diagrama, a fin que tomen los valores de salida reportados en la tabla. Considerar los caracteres "?" como condiciones sin cuidado.



<b>Nombre del Módulo:</b> maindec <b>Puertos de entrada</b> Op: 11 bits <b>Puertos de salida</b> Reg2Loc: 1 bit ALUSrc: 1 bit MemtoReg: 1 bit RegWrite: 1 bit MemRead: 1 bit MemWrite: 1 bit Branch: 1 bit ALUOp: 2 bits	Instruction	Opcode field
	LDUR	111_1100_0010
	STUR	111_1100_0000
	CBZ	101_1010_0???
	ADD	100_0101_1000
	SUB	110_0101_1000
	AND	100_0101_0000
	ORR	101_0101_0000

Instruction	reg2loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp
R-format	0	0	0	1	0	0	0	10
LDUR	0	1	1	1	1	0	0	00
STUR	1	1	0	0	0	1	0	00
CBZ	1	0	0	0	0	0	1	01
default	0	0	0	0	0	0	0	00

b) Realizar un testbench que ingrese todas las instrucciones implementadas y verifique que las señales de control a la salida correspondan a la tabla dada.

### Ejercicio 7: fetch

a) Diseñar el módulo fetch utilizando los módulos **flop**, **mux2** y **adder** según el diagrama.

**Nombre del Módulo:** fetch

**Puertos de entrada**

PCSrc\_F: 1 bit

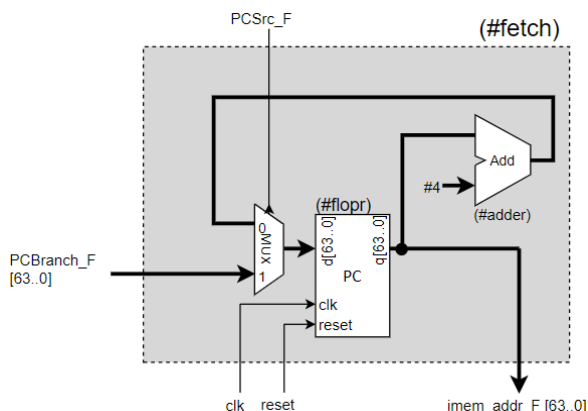
clk: 1 bit

reset: 1 bit

PCBranch\_F: 64 bits

**Puertos de salida**

imem\_addr\_F: 64 bits



b) Realizar un testbench que:

- Genere una señal de reloj en el puerto **clk** del módulo, cuya frecuencia sea 100MHz (periodo de 10 ns).
- Inicie con la señal **reset**= '1' durante 5 ciclos de clock y luego coloque **reset** = '0'.
- Inicialice **PCbrach\_F** con un valor fijo.
- Analice que después de colocar **reset** = '0' el PC inicia en "0" y que después de cada flanco positivo de clock se actualiza PC= PC+4.
- Coloque **PCSrc\_F**= '1' y en el siguiente flanco positivo de clock el PC tome el valor inicializado en **PCbrach\_F**.

### Ejercicio 8: execute

Diseñar el módulo execute utilizando los módulos **alu**, **sl2**, **adder** y **mux2** según el diagrama.

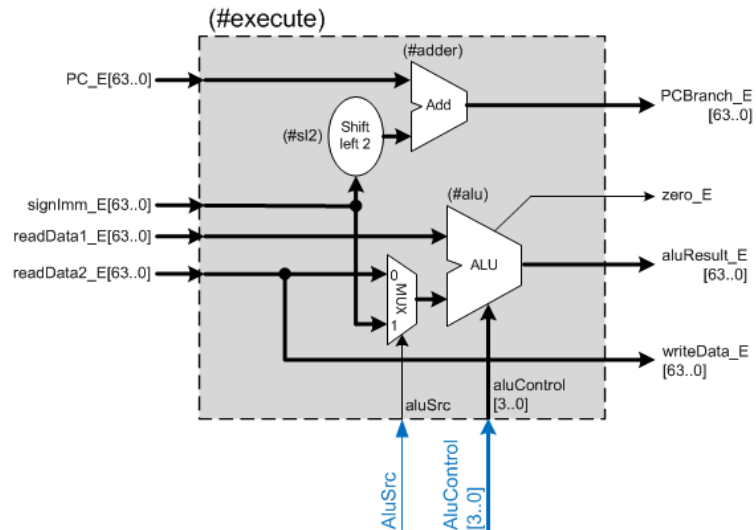
**Nombre del Módulo:** execute

#### Puertos de entrada

AluSrc: 1 bit  
 AluControl: 4 bits  
 PC\_E: 64 bits  
 signImm\_E: 64 bits  
 readData1\_E: 64 bits  
 readData2\_E: 64 bits

#### Puertos de salida

PCBranch\_E: 64 bits  
 aluResult\_E: 64 bits  
 writeData\_E: 64 bits  
 zero\_E: 1 bit



b) Realizar un testbench usando señales de entrada que permitan, a partir del análisis de las salidas resultantes, verificar la correcta instanciación y conexionado de todos los módulos y caminos de señal de la estructura interna del módulo execute.

## EJERCICIOS EXTRA

### Ejercicio 9:

Dibujar el circuito resultante de la síntesis del siguiente código en SystemVerilog:

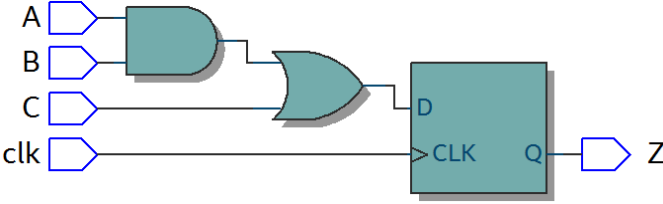
```
module ej1 (input logic A, B, C, clk,
            output logic Z);
    logic P;
    always_ff @ (posedge clk)
    begin
        P <= A & B;
        Z <= P | C;
    end
endmodule
```

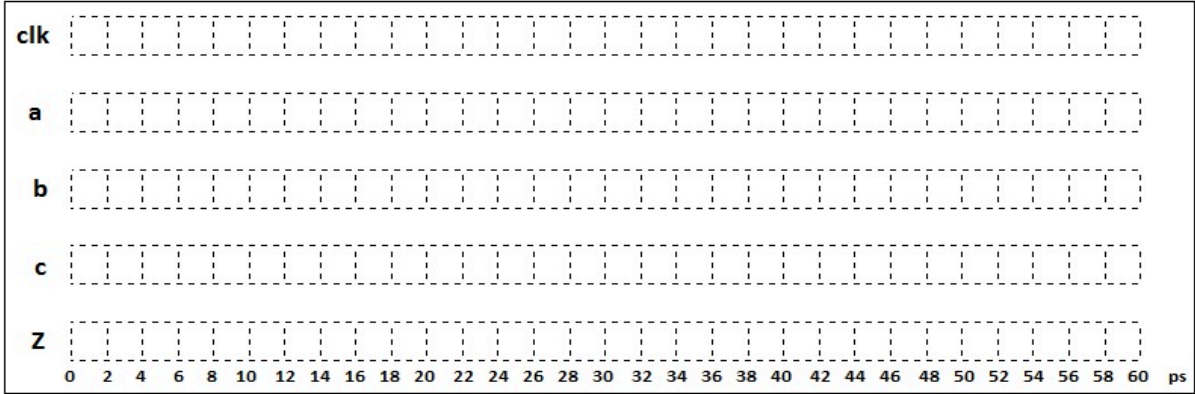
### Ejercicio 10:

La síntesis del siguiente código en SystemVerilog (**ej2**) da como resultado el circuito de la figura. Dibujar en el gráfico las señales de entrada y salida que resultan de correr el test bench (**ej\_tb**) descrito.

(CONTINÚA PÁGINA SIGUIENTE)



<pre>module ej2 (input logic A, B, C, clk,             output logic Z);     logic P;      assign P = A &amp; B;     always_ff @ (posedge clk)         begin             Z &lt;= P   C;         end endmodule</pre> 	<pre>module ej_tb();     logic a, b, c, clk, Z;      ej2 dut (a, b, c, clk, Z);      always         #4 clk = ~clk;     initial         begin             a=0; b=0; c=0; #10             a=1; #5             b=1; #5             c=1; #10             b=0; c=0; #15 \$stop;         end     initial         clk = 1; endmodule</pre>
--	---

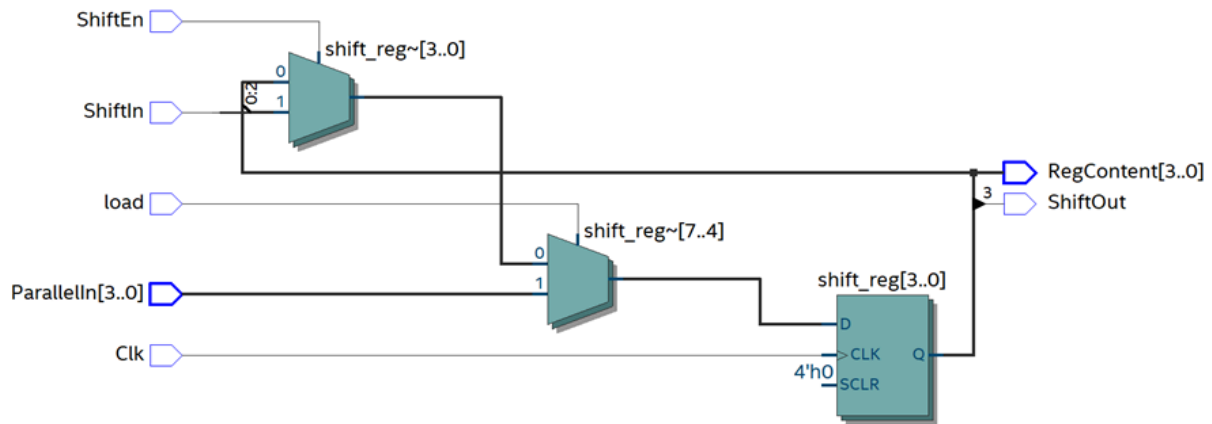


**Ejercicio 11:**  
Dibujar en el recuadro el circuito resultante de la síntesis del siguiente código en SystemVerilog:

<pre>module Recuperatorio     (input logic clk,a, b, c, d,      output logic x, y);     logic n1, n2;     logic areg, breg, creg, dreg;      always_ff @(posedge clk) begin         areg &lt;= a;         breg &lt;= b;         creg &lt;= c;         dreg &lt;= d;         x &lt;= n2;         y &lt;= ~(dreg   n2);     end     assign n1 = areg &amp; breg;     assign n2 = n1   creg; endmodule</pre>	
---	--

**Ejercicio 12:**

A continuación se describe en SystemVerilog el módulo *regPS*. Completar (en caso que corresponda) los cuadros vacíos para su correcta interpretación, considerando que el circuito resultante es el que se muestra en la figura.



```

module regPS (input _____ Clk, ShiftIn, load, ShiftEn,
               input logic [3:0] _____,
               output _____ ShiftOut,
               output logic [3:0] RegContent);

    _____ [3:0] shift_reg;

    _____ @(posedge Clk)
    if(_____ )
        shift_reg <= ParallelIn;
    else if (ShiftEn)
        _____ <= {shift_reg[2:0], ShiftIn};

    _____

begin
    _____ shift_reg[3];
    RegContent _____ shift_reg;
end

_____

```

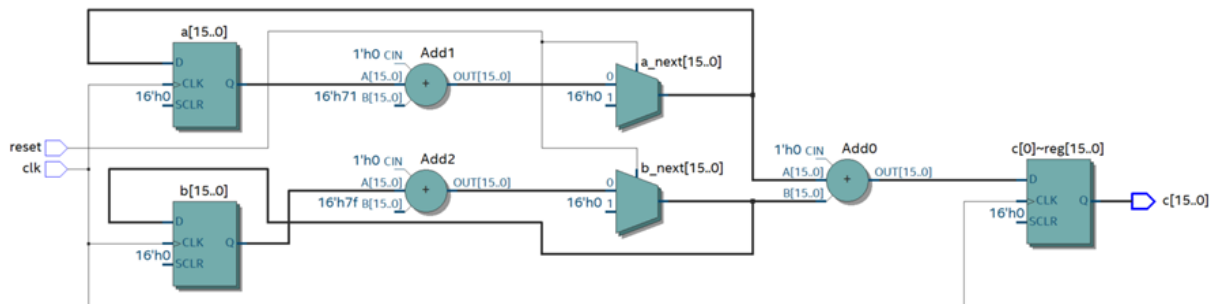
### Ejercicio 13:

Dado el siguiente módulo descrito en System Verilog:

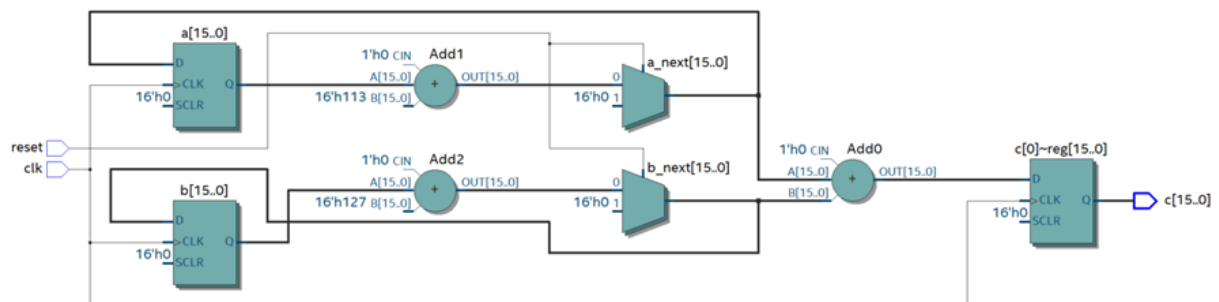
```
module mksum (output logic [15:0] c,  
             input logic reset, clk ) ;  
  
    logic [15:0] a, a_next, b, b_next ;  
    always_ff@(posedge clk)  
        begin  
            a <= a_next ;  
            b <= b_next ;  
            c <= a_next + b_next ;  
        end  
    always_comb  
        begin  
            if ( reset )  
                begin  
                    a_next = '0 ;  
                    b_next = '0 ;  
                end else begin  
                    a_next = a + 16'd113 ;  
                    b_next = b + 16'd127 ;  
                end  
        end  
endmodule
```

Seleccione cuál de los siguientes circuitos corresponde a la implementación del módulo **mksum**:

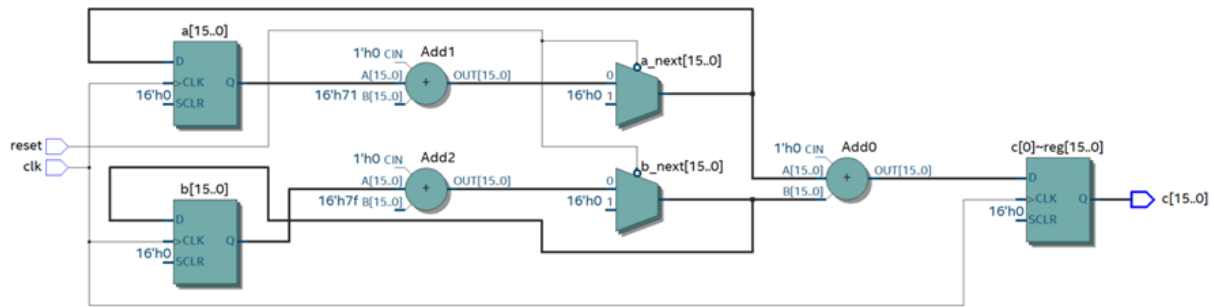
a.



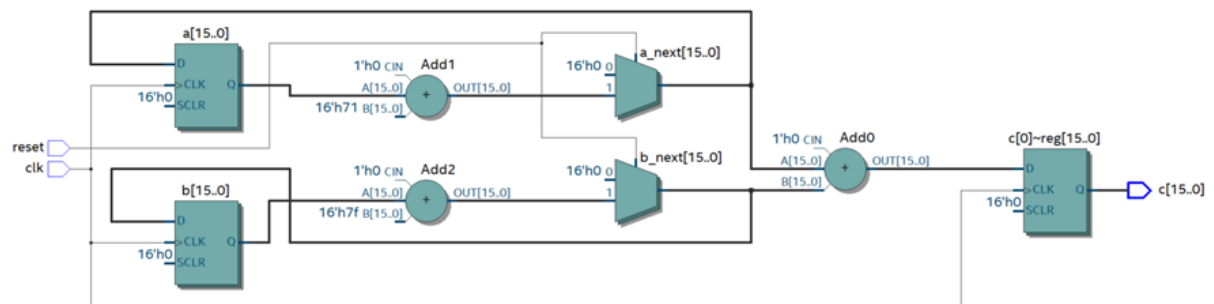
b.



C.



d.



#### Ejercicio 14:

Dado el siguiente segmento de código en SystemVerilog, con entradas "x\*", salidas "z\*" y señales internas "y\*", seleccione todas las respuestas que considere correctas.

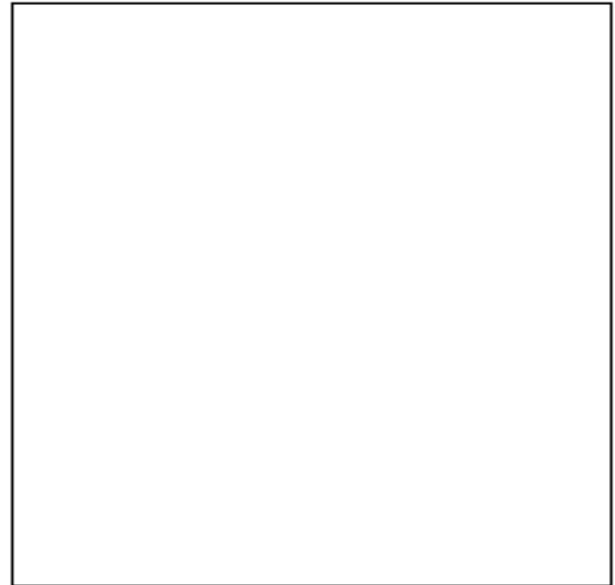
```
1> always_ff @ (posedge clk)
2>   begin
3>       z0 = y0; y0 = x0;
4>       y1= x1; z1 = y1;
5>       z2<= y2; y2 <= x2;
6>       y3<= x3; z3 <= y3;
7>   end
```

- a) La línea 3 implementa un registro paralelo.
- b) La línea 5 implementa un registro paralelo.
- c) La línea 6 implementa un registro de desplazamiento (shift register).
- d) La línea 4 implementa un registro de desplazamiento (shift register).

**Ejercicio 15:**

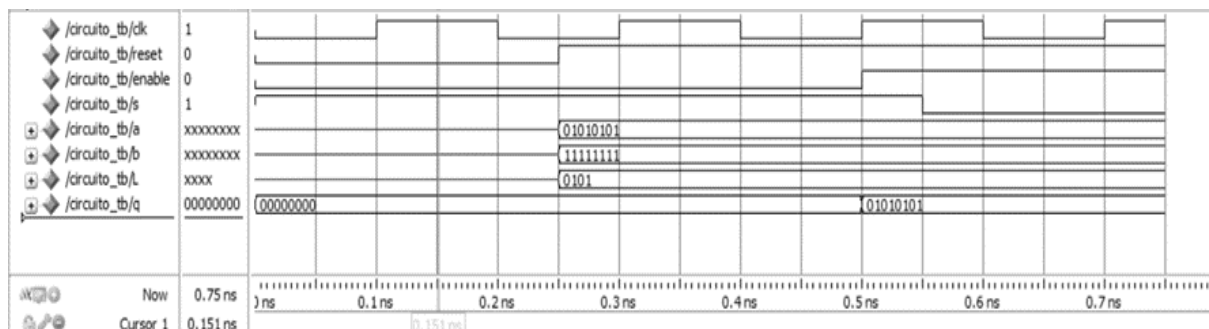
a) Dibujar el circuito resultante de la síntesis del siguiente código en SystemVerilog:

```
module circuito
    (input logic clk, r, enable, sel,
     input logic [7:0] A, B,
     output logic [7:0] Q,
     output logic [3:0] L);
    logic [7:0] Q_temp;
    always_ff @ (posedge clk, negedge r)
    begin
        if (~r)
            Q_temp <= 8'b0;
        else if (enable)
            if (sel)
                Q_temp <= Q_temp ^ A;
            else
                Q_temp <= Q_temp & B;
        end
        assign L = A[7:4];
        assign Q = Q_temp;
    endmodule
```



b) A continuación se describe en SystemVerilog el test bench del módulo **circuito**. Completar las líneas con los elementos faltantes para obtener las formas de onda de respuesta que se muestran en la figura.

```
module circuito_tb();
    logic      clk, reset, enable, s;
    logic      L;
    logic [7:0] a, b, q;
    dut (_____);
    always
    begin
        clk = _____; #100; clk = _____; #100;
    end
    initial
    begin
        reset = 0; enable = 0; s = 1; _____ ;
        _____ = 1;
        a = 8'b01010101;
        b = _____ ; _____ ;
        _____ = 1; _____ ;
        s = 0; _____ ;
        $stop;
    end
endmodule
```



**Ejercicio 16:**

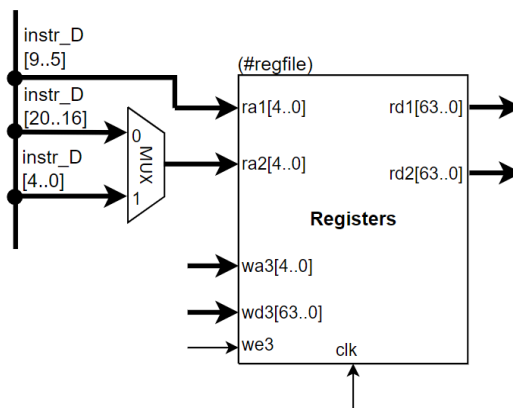
Como es sabido, las instrucciones del tipo *Store* utilizan un registro (**Rn**) como base y un valor inmediato (**DT\_address**) como desplazamiento para el cálculo de la dirección de destino de memoria.

Se propone una modificación a este esquema, de forma tal que permita a otro registro contener el valor de dicho desplazamiento, en lugar de utilizar el valor inmediato. Para esto, el campo **DT\_address** contendrá el valor del registro **Rm**, en la misma ubicación que las instrucciones del tipo “R” (bits 20 al 16).

En esta implementación no se permite la creación de un nuevo OpCode, ni de un nuevo tipo de instrucción, sino que se hará uso del bit **op[0]** del campo **op** de las instrucciones STUR existentes, de tal forma que:

op[0] = '0' → STUR Xt, [Rn, #offset] (existente)  
op[0] = '1' → STUR Xt, [Rn, Rm] (nueva)

Realizar todas las modificaciones que considere necesarias al bloque de registros del procesador LEGv8 **con pipeline** de la figura, para el correcto funcionamiento de esta nueva instrucción (sin alterar el funcionamiento de las instrucciones ya soportadas por el procesador). Luego diseñar en SystemVerilog el nuevo banco de 32 registros de 64 bits cada uno, con los puertos de entrada y salida dados y agregando los que considere necesarios.



Notas: - No es necesario inicializar los registros.

- Suponer que del bloque **#Control** sale una nueva señal **Str\_Reg** que indica el origen del offset de las instrucciones STUR (mismo valor que **op[0]**).

**Ejercicio 17:**

a) Diseñar en SystemVerilog un detector de paridad parametrizable (cantidad de bits de la entrada **in** = n, por defecto n=4) como el de la figura (**#detPar**). La salida **out** del circuito debe valer '0' cuando la cantidad de unos en el vector de entrada es par y '1' en caso contrario.

b) Diseñar en SystemVerilog un Flip-Flop D de 1 bit, con reset asíncrono, activo por bajo, según el diagrama dado (**#flipFlop**). Entradas= **d**, **clk**, **reset**. Salida= **q**.

c) Diseñar en SystemVerilog un módulo (**#RegdetPar**) que registre la salida del detector de paridad en cada flanco de subida del clock. Entradas= **input1**, **clk**, **reset**. Salida= **output1**.

