Universidad de La Laguna

Proyecto Libre

Diseño de procesadores

Autores: Melissa Díaz Arteaga

Rafael González de Chaves González

Índice:

Introducción:

En este informe vamos a explicar paso a paso como hemos elaborado una CPU monociclo, como más adelante añadimos algunas mejoras, y como usamos esas mejoras para crear un piano que posteriormente se introdujo en una FPGA.

Este proyecto fue desarrollado en Verilog, un lenguaje de descripción de hardware usado para modelar sistemas electrónicos. Éste permite la implementación de circuitos analógicos, digitales y mixtos. Su sintaxis es muy similar a la del lenguaje de programación C, por lo que resulta más fácil de usar.

Para poder desarrollar este proyecto necesitamos tres herramientas: Icarus Iverilog, GTKWave y Quartus II.

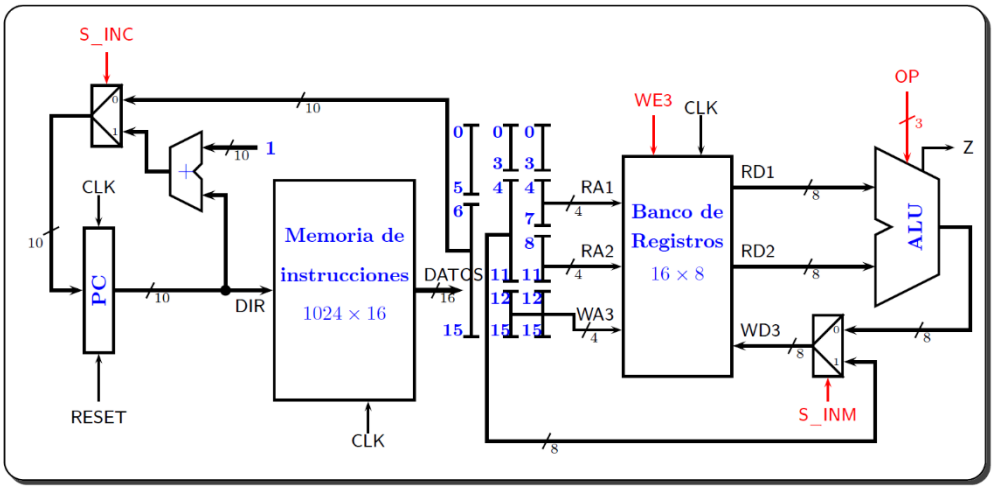
Icarus Iverilog es una herramienta de síntesis y simulación de Verilog, que funciona como compilador para traducir código fuente escrito en Verilog a cualquier formato. Para su simulación el compilador puede generar un fichero en un lenguaje intermedio (.vcp) el cual fue utilizado más adelante por el GTKWave con el fin de comprobar el funcionamiento del código.

GTKWave es una herramienta multiplataforma que permite la simulación de ficheros .vcd generados por Icarus Iverilog entre otros. Gracias a esta herramienta podemos predecir el comportamiento que va a tener el código en una maquina real.

Quartus II es una herramienta que nos permite compilar nuestros proyectos, realizar análisis del código, simular su funcionamiento, e incluso hacerlo funcionar en una maquina real como puede ser una FPGA. Un hardware pre-montado que nos permite ejecutar programas en él a muy bajo nivel, usando sus características para poder prever y verificar el rendimiento del mismo.

CPU Monociclo:

En este apartado explicaremos como hemos desarrollado una CPU simple capaz de ejecutar instrucciones de un repertorio básico en un solo ciclo sin recurrir al paralelismo. Para ello, se ha creado un camino de datos como el representado a continuación:



En él se pueden distinguir diferentes componentes:

- Una ALU (Unidad Aritmético Lógica) la cual se encargará de realizar cálculos aritméticos como la suma o la resta y cálculos lógicos como la negación, la OR o la AND.

- Un Banco de Registros, el cual se encarga de almacenar los datos para que puedan ser accesible por la ALU o de guardar los resultados de las operaciones, en este caso disponemos de 16 registros de tamaño 8 bits.

- Una Memoria de Instrucciones, la cual contiene las instrucciones que van a ser ejecutadas por nuestra CPU, esta memoria consta de 1024 palabras de 16 bits.

- Un registro especial llamado PC (Contador del Programa) que se encarga de almacenar la dirección de la siguiente instrucción que se debe ejecutar.

- Un sumador, el cual se encarga de sumar una unidad al valor del PC en cada ciclo de reloj.

- Varios multiplexores que nos permitirán usar diferentes partes del camino de datos según la instrucción que se esté ejecutando.

- Una UC (Unidad de Control) que se encargara de controlar los demás componentes y especificarles que deben hacer en cada momento según la instrucción actual.

Este camino de datos nos permite el uso de tres tipos de instrucciones: instrucciones de salto, instrucciones de carga inmediata e instrucciones de operación aritmética o lógica.

Disponemos de 16 bits para cada instrucción que se dividirán de forma diferente dependiendo del tipo de instrucción:

- Instrucción de salto: con los 6 bits menos significativos para el Opcode y los 10 bits restantes para direccionamiento.

- Instrucciones de carga inmediata: con 4 bits para el Opcode, los siguientes 8 bits para el valor inmediato a cargar, y los últimos 4 para indicar el registro destino.

- Instrucciones de operación aritmética o lógica: con los primeros 4 bits para el Opcode, los siguientes 4 bits para indicar el primer operando, los próximos 4 bits para el segundo operando, y los últimos 4 bits para el registro destino que albergara el resultado de la operación.

Implementación:

Para el desarrollo de este proyecto se nos proporcionó varios bloques de código que correspondían con la Memoria de Instrucciones, el Banco de Registros, la ALU, un Multiplexor 2 a 1, un registro usado para el PC y un sumador. Por lo que nos enfocamos en la conexión de todos los componentes de forma funcional y en la creación de la UC que los controlase.

Para conectar todos los componentes creamos un módulo llamado microc el cual contiene todo el camino de datos y recibe las señales de control de la UC

```
module microc(input wire clk, reset, s_inc, s_inm, we3, input wire [2:0] op, output wire z, output wire [5:0] opcode);

wire [9:0] SalMux10, SalSum, SalPc;
wire [15:0] SalMem;
wire [7:0] SalMux8, SalAlu, SalR1, SalR2;

registro #(10) pc(clk, reset, SalMux10, SalPc);
mux2 #(10) mux10(SalMem[9:0], SalSum, s_inc, SalMux10);
sum pc1(SalPc, 10'b000000001, SalSum);
memprog meminst(clk, SalPc, SalMem);

assign opcode = SalMem[15:10];

regfile banreg(clk, we3, SalMem[11:8], SalMem[7:4], SalMem[3:0], SalMux8, SalR1, SalR2);
mux2 #(8) mux8(SalAlu, SalMem[11:4], s_inm, SalMux8);
alu alu8(SalR1, SalR2, op, SalAlu, z);

endmodule
```

Para controlar el microc desarrollamos el módulo de la UC el cual, a partir del Opcode, detecta la instrucción a la que corresponde dentro de nuestro repertorio básico y cambia el valor de las señales de control en consecuencia.

El tipo de unidad de control implementada es cableada, tomamos esta decisión ya que se trata de una máquina sencilla con un repertorio de instrucciones básico.

```
module uc(input wire clk, z, input wire [5:0] opcode, output wire s_inc, s_inm, we3, output wire [2:0] op);

reg rz;
initial
begin
   rz = 1'b0;
end
always @(opcode[5], z)
if(opcode[5] == 1)
  rz = (s_inc & z);

assign s_inc = (((opcode[5] == 0)&(opcode[4] == 0)&(opcode[3] == 0))|((opcode[5] == 0)&(opcode[4] == 0)&
               (opcode[3] == 1)&(opcode[2] == rz)))? 0:1;
assign s_inm = ((opcode[5] == 0)&(opcode[4] == 1))? 1:0;
assign we3 = (((opcode[5] == 0)&(opcode[4] == 1))|((opcode[5] == 1)))? 1:0;
assign op = opcode[4:2];

endmodule
```

Repertorio de instrucciones:

La codificación elegida para nuestro repertorio de instrucciones es la siguiente:

Instrucciones de salto:

- Incondicional: Opcode: 0000XX (6 bits)

- Condicional:
  - Si no es cero: Opcode: 0010XX (6 bits)

  - Si es cero: Opcode: 0011XX (6 bits)

- Dirección destino: XXXXXXXXXX (10 bits)

Instrucción completa: - Opcode - Dirección destino -

Instrucciones de carga inmediata en registro:

- Opcode: 01XX (4 bits)
- Valor inmediato: XXXXXXXX (8 bits)
- Registro destino: XXXX (4 bits)

Instrucción completa: - Opcode - Valor inmediato - Registro destino –

Instrucciones de operaciones aritméticas y lógicas:

- Opcode: 1XXX (4 bits)
- Registro fuente A: XXXX (4 bits)
- Registro fuente B: XXXX (4 bits)
- Registro destino S: XXXX (4 bits)

Instrucción completa: - Opcode - Registro A - Registro B - Registro S -

Operaciones posibles en los 3 bits a X del Opcode:

- 000: s = a;
- 001: s = ~a;
- 010: s = a + b;
- 011: s = a - b;
- 100: s = a & b;
- 101: s = a | b;
- 110: s = -a;
- 111: s = -b;

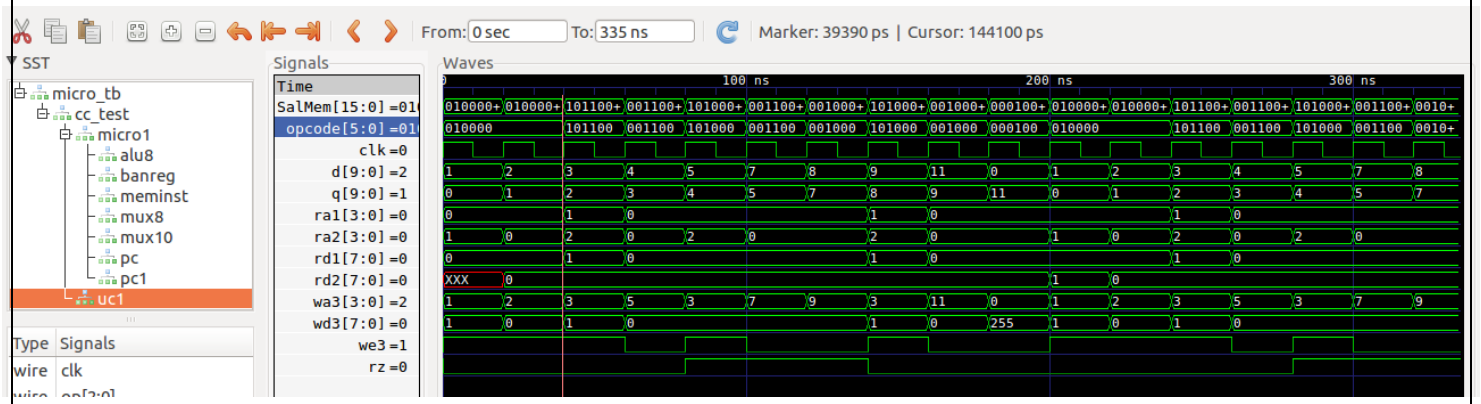
Para la comprobar el funcionamiento de nuestro hardware elaboramos un primer programa que se muestra a continuación:

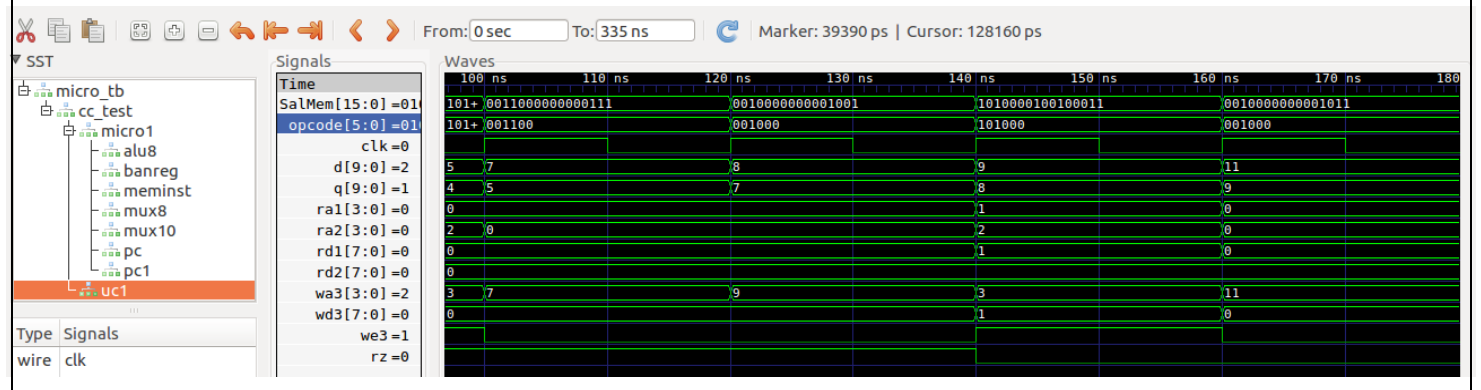
```
0100_0000_0100_0001  // Carga 4 en R1
0100_0000_0001_0100  // Carga 1 en R4
0100_0000_0010_0010  // Carga 2 en R2
0100_0000_0100_0011  // Carga 4 en R3
1010_0010_0011_0010  // Suma R2 con R3 y lo guarda en R2
1011_0001_0100_0001  // Resta R1 con R4 y lo guarda en R1
0010_0000_0000_0100  // Salta a la instrucción 4 si R1 es distinto de 0
0000_0000_0000_0010  // Salta a la instrucción 2
0000_0000_0000_0000  // Salta a la instrucción 0 (No se ejecuta porque el salto incondicional anterior se salta esta instrucción)
0100_0000_0010_0010  // Carga 2 en R2
0100_0000_0100_0011  // Carga 4 en R3
1010_0010_0011_0010  // Suma R2 con R3 y lo guarda en R2
```

Posteriormente elaboramos un segundo programa más complejo requerido por el profesor:

```
0100_0000_0001_0001     E0: LI $1, 1
0100_0000_0000_0010         LI $2, 0
1011_0001_0010_0011         SUB $3, $1, $2
0011_0000_0000_0101         BZ E1
1010_0000_0010_0011         ADD $3, $0, $2
0011_0000_0000_0111     E1: BZ E2
0000_0000_0000_0000         NOP
0010_0000_0000_1001     E2: BNZ E3
1010_0001_0010_0011         ADD $3, $1, $2
0010_0000_0000_1011     E3: BNZ E4
0000_0000_0000_0000         NOP
0001_0000_0000_0000     E4: J E0
```

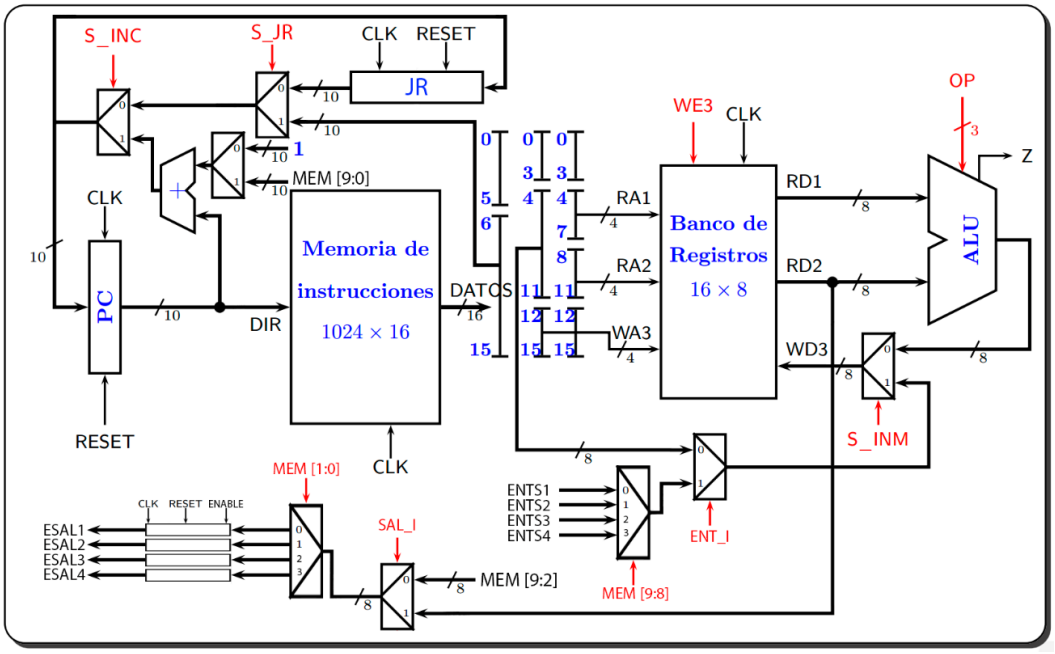
Para este último, obtuvimos los siguientes resultados:





Mejoras implementadas:

A partir de la CPU monociclo desarrollada hicimos ciertas mejoras para agregarle mayor funcionalidad a nuestro hardware y extender el repertorio de instrucciones para incluir capacidad de ejecución de subrutinas más fácilmente y la introducción de la entrada y salida para que nuestro hardware sea capaz de comunicarse con el exterior. Para ello fue necesario modificar ligeramente el camino de datos quedando como resultado el representado a continuación:



Para la inclusión de la entrada, se ha realizado una modificación en el camino entre la instrucción de memoria y el multiplexor que decide si el valor a escribir en el registro objetivo proviene de la memoria o de la ALU, hemos introducido un multiplexor intermedio que elige la proveniencia del valor que se llevara al multiplexor que decide entre la ALU y la memoria. Las entradas de este multiplexor son dos, la primera es el valor que se trae de memoria, y la segunda es la salida del multiplexor que decide una entre las cuatro posibles entradas del camino de datos.

Para la salida hemos contemplado dos posibilidades, se puede querer sacar por la salida un valor guardado en un registro o un valor albergado en la instrucción proveniente de memoria, para ello hemos incluido un nuevo fragmento al camino de datos, el primero de los elementos incluidos ha sido un multiplexor que se encargué de decidir la proveniencia del valor que se ha de sacar, de un registro o inmediato de memoria. El valor de la salida de este multiplexor va a un decodificador, el cual se encarga de seleccionar por qué salida se ha de sacar el valor.

Sin embargo, la salida ha de ser mantenida hasta que se vuelva a querer sacar otro valor por la misma, por ello hemos usado unos registros intermedios entre la salida y el decodificador, estos registros solo serán escritos en instrucciones de salida así que les hemos añadido una señal enable para evitar su escritura en otros casos. Por lo tanto la salida de estos registros es la que está conectada a su salida correspondiente del camino de datos.

Para facilitar el uso de subrutinas hemos añadido dos instrucciones de salto, el JAL (Jump And Link) y JR (Jump Return).

La primera se usa para hacer llamadas a subrutinas, cuando se ejecuta esta instrucción se hace el salto a la dirección especificada y a la par se guarda la dirección de la siguiente instrucción en un registro aparte. La segunda se usa para salir de las subrutinas, cuando se ejecuta salta fuera de la subrutina a la dirección siguiente de la que se llamó, esta dirección se encuentra en un registro aparte y fue guardada por la primera instrucción, el JAL.

Para añadir estas dos instrucciones fue necesario incluir un registro especial cuya entrada es la salida del sumador, y su salida se conecta un nuevo multiplexor que se encarga de decidir si la dirección del salto proviene de la instrucción en memoria o del registro especial.

También incluimos un salto relativo añadiendo un multiplexor a la entrada del sumador que decidiera si se suma uno o se suma la cantidad indicada en la instrucción de memoria.

Además, hicimos una reorganización del repertorio de instrucciones con respecto al original para diferenciar mejor unas instrucciones de otras y albergar todas las nuevas funcionalidades, el nuevo repertorio es el siguiente:

Instrucciones de salto:

- Incondicional: Opcode: 000000 (6 bits)
- Condicional:
  -Si no es cero: Opcode: 000010 (6 bits)
  -Si es cero: Opcode: 000011 (6 bits)
- JAL (Jump And Link): Opcode: 000100 (6 bits)
- JR (Jump Return): Opcode: 000101 (6 bits)
- Relativo: Opcode: 000110 (6 bits)

  Numero de instrucciones a saltar: XXXXXXXXXX (10 bits)

  Dirección destino: XXXXXXXXXX (10 bits)

  Instrucción completa: - Opcode - Dirección destino/Nº de direcciones a saltar -

Instrucciones de carga inmediata en registro:

- Opcode: 0100 (4 bits)
- Valor inmediato: XXXXXXXX (8 bits)
- Registro destino: XXXX (4 bits)

Instrucción completa: - Opcode - Valor inmediato - Registro destino -

Instrucciones de operaciones aritméticas y lógicas:

- Opcode: 1XXX (4 bits)
- Registro fuente A: XXXX (4 bits)
- Registro fuente B: XXXX (4 bits)
- Registro destino S: XXXX (4 bits)

Instrucción completa: - Opcode - Registro A - Registro B - Registro S -

Operaciones posibles en los 3 bits a X del Opcode:

000: s = a;

001: s = ~a;

010: s = a + b;

011: s = a - b;

100: s = a & b;

101: s = a | b;

110: s = -a;

111: s = -b;

Entrada:

- Opcode: 001000 (6 bits)
- Numero de entrada: XX (2 bits)
- Registro destino: XXXX (4 bits)

Instrucción completa: - Opcode - Nº de entrada - XXXX (4 bits sin uso) - Registro destino

Salida:

- Desde registro:

    - Opcode: 001010 (6 bits)

    - Numero de salida: XX (2 bits)

    - Registro origen: XXXX (4 bits)

Instrucción completa: - Opcode - XX (2 bits sin uso) - Registro origen - XX (2 bits sin uso) - Nº de salida

- Desde inmediato en la instrucción:

    - Opcode: 001011 (6 bits)

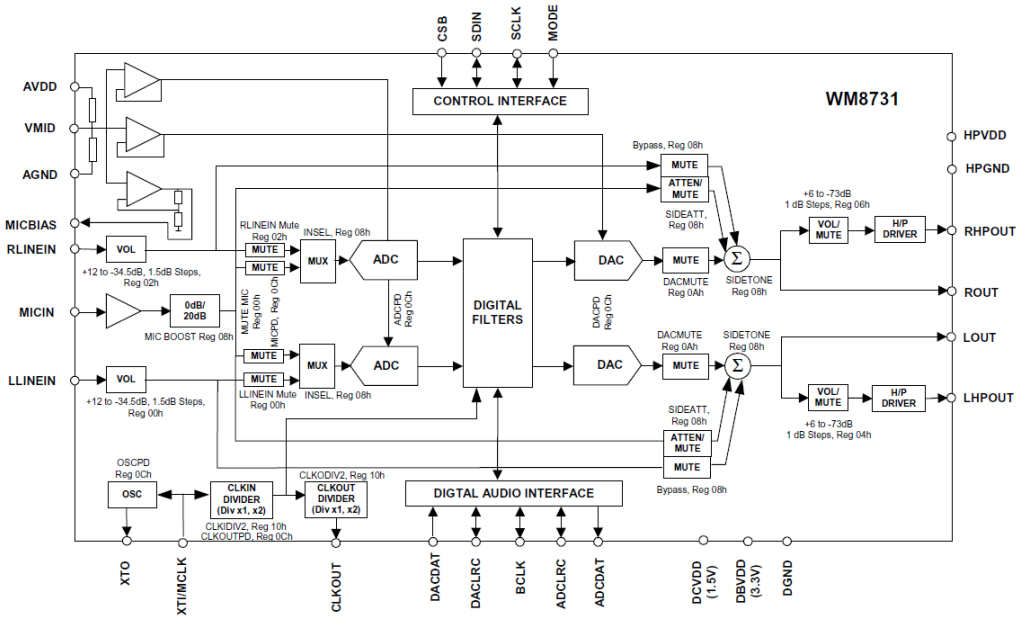
    - Inmediato: XXXXXXXX (8 bits)

    - Numero de salida: XX (2 bits)

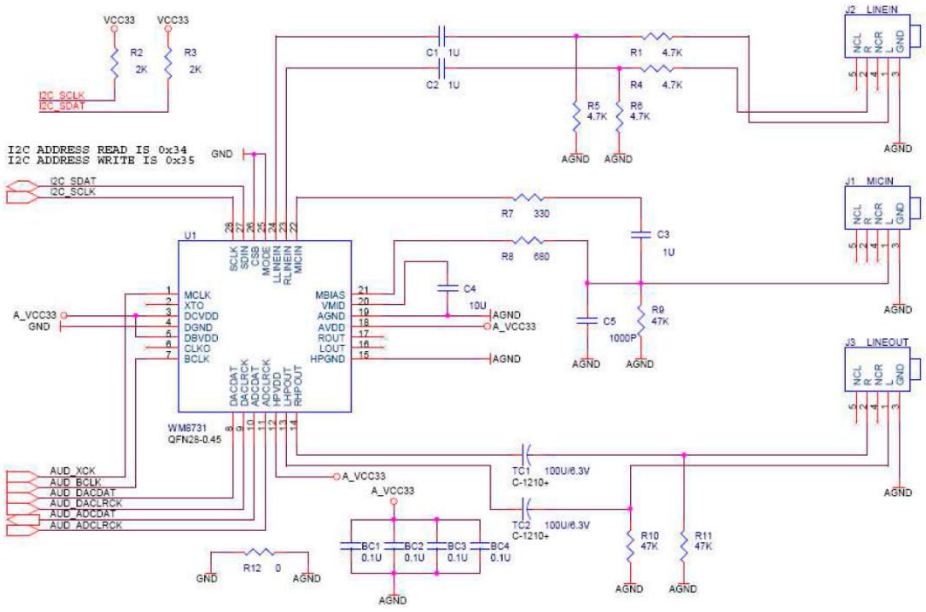
Instrucción completa: - Opcode - Inmediato - Nº de salida

Proyecto Libre:

Como proyecto hemos decidido realizar un piano partiendo del código del camino datos explicado en el apartado anterior. Para su implementación hemos usado el módulo de audio.



En él se pueden observar las señales de entrada y salida necesarias, como son las de la interfaz de control que provienen de las señales del bus i2c o también las señales de la interfaz digital de audio. Todas estas señales van al bloque de filtros digitales y de ahí salen a los conversores digital analógicos, los cuales están conectados el puerto de salida de audio izquierdo y derecho.



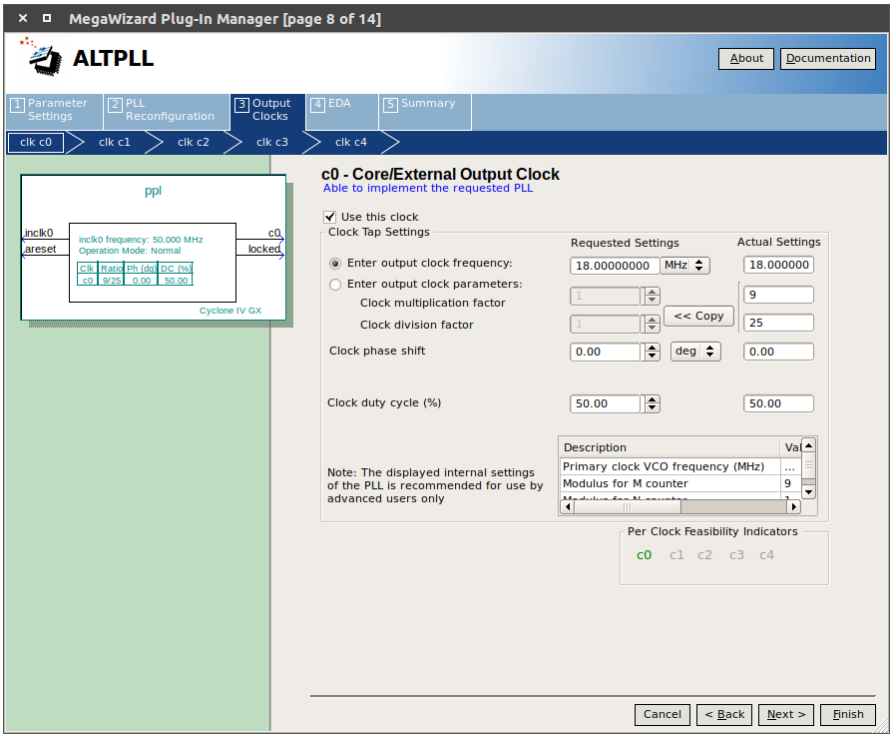
En esta imagen podemos observar mejor las señales de audio y del bus i2c que necesita el chip. Podemos ver también las salidas del mismo al Jack de 3.5.

Estas señales las hemos incluido en nuestro módulo del camino completo con tipos de conexiones, input, output e inout. Además, hemos inicializado las conexiones en triestado.

```
input wire AUD_ADCDAT, inout wire AUD_ADCLRCK, AUD_DACLRCK, AUD_BCLK, I2C_SDAT, output wire AUD_DACDAT, AUD_XCK, I2C_SCLK,
```

```
wire I2C_END;
wire AUD_CTRL_CLK;
assign  I2C_SDAT    =   1'bz;
assign  AUD_ADCLRCK =   AUD_DACLRCK;
assign  AUD_XCK     =   AUD_CTRL_CLK;
```

Para usar el módulo de audio hemos necesitado incluir un pll que transformase la señal de entrada del módulo de 50Mhz a 18Mhz.



Por último, hemos incluido el módulo de audio el nuestra cpu, este recibirá un selector de audio, un reloj de 18Mhz, una señal de reset, una frecuencia de la nota que se desea reproducir y la frecuencia de su armónico.

```
module audio_codec (
output          oAUD_DATA,
output          oAUD_LRCK,
output reg      oAUD_BCK,
input   [1:0]   iSrc_Select,
input           iCLK_18_4,
input           iRST_N,
input [15:0]    var_Rate,
input [15:0]    arm_Rate
                      );

parameter   REF_CLK         =   18432000;   //  18.432  MHz
parameter   DATA_WIDTH      =   16;         //  16      Bits
parameter   SAMPLE_RATE     =  48000;      // 48    KHz
parameter   CHANNEL_NUM     =   2;            //   Dual Channel
```

Además, definiremos parámetros como la velocidad del reloj de entrada, el número de bits que se usa para representar la señal, la frecuencia de muestreo (cada cuanto se envía un dato nuevo) y el número de canales, que nos serán útiles más adelante

Utilizando estos datos realizamos un reloj que calcula cada cuantos ticks del reloj de entrada debe enviar un bit, teniendo en cuenta la frecuencia del reloj que entrada, la frecuencia de muestreo, el tamaño de cada muestra y el número de canales de salida. Para estos valores se tomarán dos muestras con el fin de evitar un fenómeno de aliasing.

```
////////////    AUD_BCK Generator   //////////////
always@(posedge iCLK_18_4 or negedge iRST_N)
begin
    if(!iRST_N)
    begin
        BCK_DIV     <=  0;
        oAUD_BCK    <=  0;
    end
    else
    begin
        if(BCK_DIV >= REF_CLK/(SAMPLE_RATE*DATA_WIDTH*CHANNEL_NUM*2)-1 )
        begin
            BCK_DIV     <=  0;
            oAUD_BCK    <=  ~oAUD_BCK;
        end
        else
        BCK_DIV     <=  BCK_DIV+1;
    end
end
```

Además, debemos hacer otro reloj que calcule cada cuantos ticks del reloj de entrada debe cambiar la posición del puntero en la tabla de muestreo, es decir, pasar a la siguiente muestra según la frecuencia del reloj de entrada y la frecuencia de la nota a emitir por el número de muestras en la tabla de muestreo.

```
////////////    Select Generator    //////////////
always@(posedge iCLK_18_4 or negedge iRST_N)
begin
    if(!iRST_N)
    begin
        Sel_Clk  <= 0;
        select  <=  0;
    end
    else
    begin
        if(Sel_Clk >= REF_CLK/(var_Rate*64) )
        begin
            Sel_Clk     <=  0;
            select  <=  select+1;
        end
        else
        Sel_Clk     <=  Sel_Clk+1;
    end
end

sin_table Stable(select, sound_1);
```

Para saber la frecuencia de la nota hemos hecho un módulo que según el valor de la entrada en los botones reconoce que nota esta y le asigna a la variable de la frecuencia la correspondiente. En caso de que no haya coincidencia se le asigna un 0.

```
module switch(input wire clk, reset, input wire [7:0] select, output reg [15:0] sample_rate);

always@(select, reset)
begin
    if(reset)
    begin
        sample_rate <= 0;
    end
    else
    begin
        case(select)
            8'b00000111:
            begin
                sample_rate <= 1046;
            end
            8'b00001011:
            begin
                sample_rate <= 1174;
            end
            8'b00001101:
            begin
                sample_rate <= 1318;
            end
            8'b00001110:
            begin
                sample_rate <= 1396;
            end
            default:
            begin
                sample_rate <= 0;
            end
        endcase
    end
end

endmodule
```

Entonces con el puntero que calculamos en el módulo de audio, recorremos el módulo *sin_table* que tiene muestreada una onda sinusoidal con 64 valores.

```
module sin_table (input wire [5:0] select, output reg [15:0] wave);

always@(select[5:0])
begin
    case(select[5:0])
 0 :wave=0;
 1 :wave=4000;
 2 :wave=8000;
 3 :wave=12000;
 4 :wave=16000;
 5 :wave=20000;
 6 :wave=24000;
 7 :wave=28000;
 8 :wave=32000;
 9 :wave=36000;
 10 :wave=40000;
 11 :wave=44000;

    ....

 60 :wave=-8000;
 61 :wave=-4000;
 62 :wave=-1000;
 63 :wave=0;
default :wave=0;
        endcase
end

endmodule
```

Cuando un valor ha sido elegido tenemos que enviarlo, pero solo podemos enviarlo bit a bit por lo que es necesario serializarlo, antes habíamos calculado un reloj que marcaba cada cuanto tenemos que enviar un bit, así que, con ese reloj incrementamos un contador de 4bits, es decir, de 0 a 15 que es el tamaño de la muestra, y enviamos el bit.

```
////////////////////////////////////////////////////
////////////////     Sound Out   ////////////////////
assign sound_out = sound_1;

always@(negedge oAUD_BCK or negedge iRST_N)
begin
    if(!iRST_N)
        SEL_Cont    <=  0;
    else
        SEL_Cont    <=  SEL_Cont+1;
end

assign oAUD_DATA = sound_out[~SEL_Cont];
```

Además, hemos incluido un armónico para la nota, el cual necesita un puntero también recorriendo otra tabla de muestro. Este armónico tendrá su propia frecuencia por lo que calculamos un reloj para el armónico según la frecuencia del reloj de entrada, la frecuencia del armónico y el número de valores en la tabla, es este caso 48.

```
////////////    Armonic 1 Generator //////////////
always@(posedge iCLK_18_4 or negedge iRST_N)
begin
    if(!iRST_N)
    begin
        Sel_Clk_Arm  <= 0;
        select_Arm  <=  0;
    end
    else
    begin
        if(Sel_Clk_Arm >= REF_CLK/(arm_Rate*48) )
        begin
            Sel_Clk_Arm     <=  0;
            select_Arm  <=  select_Arm+1;
        end
        else
        Sel_Clk_Arm     <=  Sel_Clk_Arm+1;
    end
end

sin_table2 S1table(select_Arm, sound_2);
```

También debemos incluir en el selector de nota que según la nota se le asigne una frecuencia al armónico u otra. Entonces con este nuevo puntero recorremos una nueva tabla con una onda sinusoidal muestreada en 48 valores de menor tamaño.

```
if(reset)
begin
    sample_rate <= 0;
    arm_Rate <= 0;
end
else
begin
    case(select)
        8'b00000111:
        begin
            sample_rate <= 1046;
            arm_Rate <= 300;
        end
        8'b00001011:
        begin
            sample_rate <= 1174;
            arm_Rate <= 250;
        end
        8'b00001101:
        begin
            sample_rate <= 1318;
            arm_Rate <= 200;
        end
        8'b00001110:
        begin
            sample_rate <= 1396;
            arm_Rate <= 100;
        end
        default:
        begin
            sample_rate <= 0;
            arm_Rate <= 0;
        end
    endcase
end
```

```
module sin_table2 (input wire [5:0] select, output reg [15:0] wave);

always@(select[5:0])
begin
    case(select[5:0])
 0 :wave=0;
 1 :wave=4000;
 2 :wave=8000;
 3 :wave=12000;
 4 :wave=16000;
 5 :wave=20000;
 6 :wave=24000;
 7 :wave=28000;
 8 :wave=32000;
 9 :wave=36000;
 10 :wave=40000;
 11 :wave=44000;
 12 :wave=48000;

    ....

 46 :wave=-8000;
 47 :wave=-4000;
 48 :wave=0;
default :wave=0;
       endcase
end

endmodule
```

Ahora con el armónico, debemos calcular un nuevo valor final que sea la media entre la nota y el armónico y enviarlo serializado.

```
////////////////////////////////////////////////
////////////////    Sound Out   ////////////////////
assign sound_out = (sound_1 + sound_2)/2;

always@(negedge oAUD_BCK or negedge iRST_N)
begin
    if(!iRST_N)
        SEL_Cont    <=  0;
    else
        SEL_Cont    <=  SEL_Cont+1;
end

assign oAUD_DATA = sound_out[~SEL_Cont];
```

Por último, para que el usuario sepa que nota está tocando hemos hecho un programa que detecte la entrada y en consecuencia saque por la salida el valor que debe mostrar el visor de 7 segmentos, las posibilidades son do, re, mi y fa.

```
0100000000001111  // Inicializa a 0 el registro 15 (Acumulador)
0100000000000000  // Inicializa a 0 el registro 0
0100000011110101  // Inicializa a 15 el registro 5
0100000000001010  // Inicializa a 0 el registro 10
0100000011111011  // Inicializa a 15 el registro 11
0100000001110001  // Inicializa a 7 el registro 1 (DO)
0100000010110010  // Inicializa a 11 el registro 2 (RE)
0100000011010011  // Inicializa a 13 el registro 3 (MI)
0100000011100100  // Inicializa a 14 el registro 4 (FA)
0010100000000000  // Carga en la salida 1 el registro 0
0010100000000001  // Carga en la salida 2 el registro 0
1010101100001010  // Carga el valor del registro 11 en el registro 10
0010000100001011  // Carga el valor de la entrada 2 en el registro 11
1011101101011110  // Resta al registro 11 el registro 5 y lo guarda en el registro 14
0000110000001001  // Salta a la direccion 9 si la ultima operacion fue 0**
1011101001011110  // Resta al registro 10 el registro 5 y lo guarda en el registro 14
0000100000001011  // Salta a la direccion 11 si la ultima operacion no fue 0**
1011101100011110  // Resta al registro 11 el registro 1 y lo guarda en el registro 14
0000110000011010  // Salta a la direccion 26 si la ultima operacion fue 0 (DO)**
1011101100101110  // Resta al registro 11 el registro 2 y lo guarda en el registro 14
0000110000011101  // Salta a la direccion 29 si la ultima operacion fue 0 (RE)**
1011101100111110  // Resta al registro 11 el registro 3 y lo guarda en el registro 14
0000110000100000  // Salta a la direccion 32 si la ultima operacion fue 0 (MI)
1011101101001110  // Resta al registro 11 el registro 4 y lo guarda en el registro 14
0000110000100011  // Salta a la direccion 35 si la ultima operacion fue 0 (FA)**
0000000000001011  // Salta a la direccion 11**
0010110010001100  // Carga en la salida 1 el inmediato (o)
0010110010000101  // Carga en la salida 2 el inmediato (d)
0000000000001011  // Salta a la direccion 11**
0010110000011000  // Carga en la salida 1 el inmediato (e)
0010110010111101  // Carga en la salida 2 el inmediato (r)
0000000000001011  // Salta a la direccion 11**
0010110100111100  // Carga en la salida 1 el inmediato (i)
0010110000100101  // Carga en la salida 2 el inmediato (m)
0000000000001011  // Salta a la direccion 11**
0010110000100000  // Carga en la salida 1 el inmediato (a)
0010110000111001  // Carga en la salida 2 el inmediato (f)
0000000000001011  // Salta a la direccion 11**
```

```
assign HEX1 = ESal1;
assign HEX2 = ESal2;
assign HEX0 = 7'b0111111;
assign HEX3 = 7'b0111111;
```

Bibliografía:

http://www.cs.columbia.edu/~sedwards/classes/2008/4840/Wolfson-WM8731-audio-CODEC.pdf

https://www.altera.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-4904342209-de1-usermanual.pdf

http://latecladeescape.com/h/2015/08/frecuencia-de-las-notas-musicales

http://www.nch.com.au/wavepad/es/