Codiseño Hardware Software
Implementación de la FFT utilizando múltiples procesadores – Parte 2

Alonso Rodriguez Iglesias (alonso.rodriguez@udc.es)
Xabier Iglesias Pérez (xabier.iglesias.perez@udc.es)

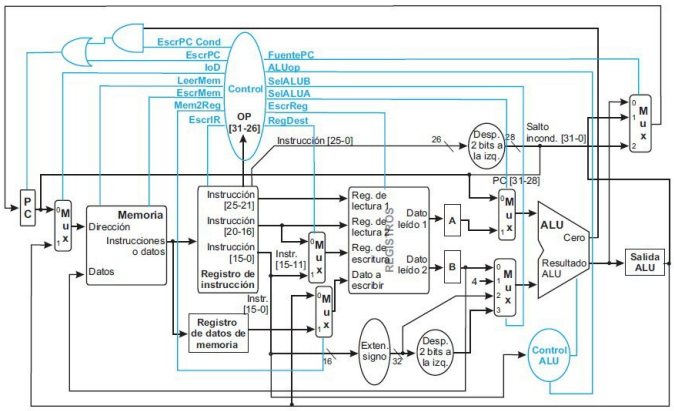
Introducción

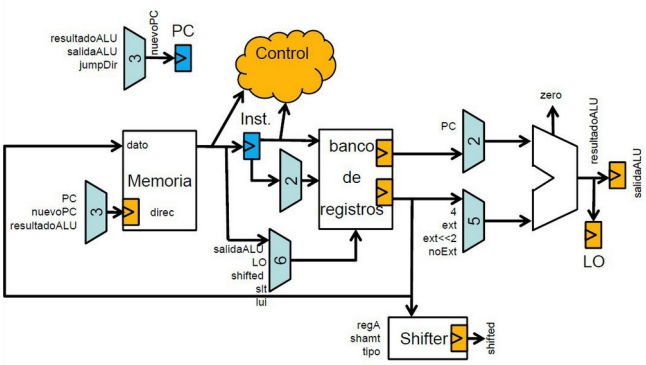
En esta parte, se implementarán en precisión de ciclo de reloj, diferentes instrucciones necesarias para ejecutar programas bajo nuestra arquitectura.

En concreto a nuestro equipo le ha sido asignada la instrucción jal

Descripción de la arquitectura

Los siguientes diagramas ilustran la arquitectura





La instrucción JAL

La instrucción JAL (Jump And Link) es el equivalente MIPS a la archiconocida CALL para arquitecturas basadas en el 8086.

En concreto la instrucción se ejecuta en tres pasos:

```
jal sub  # $ra <– PC+4  (the address 8 bytes away from the jal)
         # PC  <– sub   load the PC with the subroutine entry point
         # a branch delay slot follows this instruction
```

https://chortle.ccsu.edu/assemblytutorial/Chapter-26/ass26_4.html

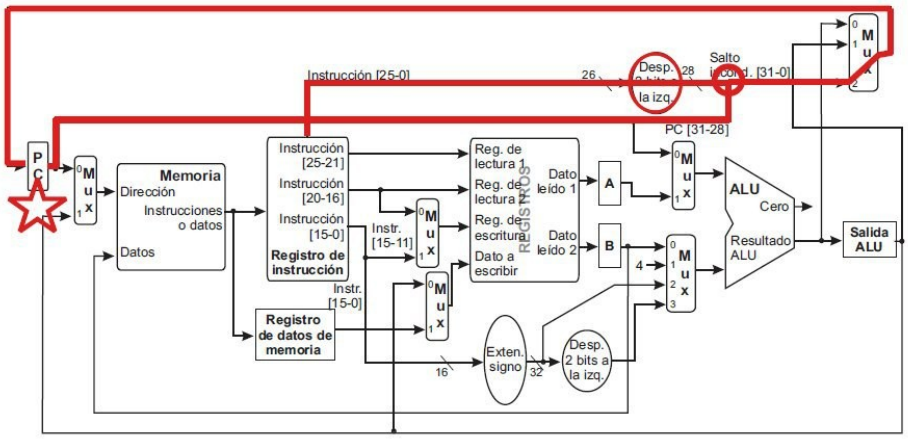
Implementación

La implementación que hemos seguido divide el problema en dos:

1. Saltar a la dirección

2. Guardar la dirección de retorno

Para el primero de estos seguiremos el siguiente camino de datos.



Analizando esto, obtenemos los siguientes pasos:

1. Obtener las señales Instrucción [25-0] y PC
2. Desplazar Instrucción [25-0] y concatenar (or) con PC
3. Poner el mux en la posición correcta (2)

La segunda parte es mas compleja. Para guardar un dato en el registro $ra, debemos introducir en el banco de registros las siguientes señales. En el registro de escritura especificar el registros $ra, en el dato a escribir debemos obtener PC+4. Para obtener este dato, usaremos la salida del registro PC, que ya está actualizado a PC+4 durante el salto.

Modificaciones en la arquitectura

El pipeline de salto se mantiene inalterado, sin embargo para obtener el PC+4, vamos a introducir una señal por el mux de datos (siguiendo la nomenclatura proporcionada).

Implementación en código

En muxes.cpp, hemos añadido lo siguiente:

case 5 => lee el valor de PC y pone rdValue al mismo.

```
void muxes::muxDato() {

    sc_int<32> slt = 0;

    switch (selDatoReg.read()) {
    case 0:     rdValue.write(salidaMem.read());            break;
    case 1:     rdValue.write(salidaALU.read());            break;
    case 2:     rdValue.write(LOreg.read());                break;
    case 3:     rdValue.write(shifted.read());              break;
    case 4:     slt.bit(0) = salidaALU.read().bit(31);
                rdValue.write(slt);                         break;
    case 5:     rdValue.write(PC.read().to_int());          break;
    default:    rdValue.write(inm.read() << 16);            // lui
    };
}
```

```
case jal:
    // Seleccionamos la salida del mux con una señal de control registrada
    selDatoReg = 5;

    state = jal2;
    break;
case jal2:
    // Escribe el valor de selDatoReg (ciclo anterior): $ra = PC
    EscrReg.write(1);
    rd.write((unsigned int) 31);

    // Realizamos el salto
    FuentePC.write(2);
    EscrPC.write(1);

    // Cargamos la siguiente instrucción
    LeerMem.write(1);
    IoD.write(3);

    state = inicial;
    break;
```

En el caso de control.cpp añadimos el siguiente código:

En el estado jal, simplemente decimos que el dato a escribir es el PC+4 (el case 5 comentado anteriormente).

Como la salida está registrada, debemos esperar un ciclo extra para poder disponer de nuestros datos.

En el segundo estado (jal2), activamos la escritura en el banco de registros, especificamos el registro $ra (31), ponemos el mux de Fuente en el modo correcto y especificamos que actualizaremos el pc. Finalmente cargamos la siguiente instrucción para ejecutar el salto.
Por supuesto todo esto se realiza simultáneamente al tick del reloj, pero lo tenemos por “fases” por mera conveniencia.

Para generar la dirección de salto programamos el caso 2 en gestionpc.cpp. (shifteamos la dirección de salto 2 bits a la izquierda y ponemos los 4 bits más significativos al valor actual del registro PC).

```
void gestionPC::mux(){

    switch (FuentePC.read()) {
    case 0:
        tmpPC = resultadoALU.read();    break;
    case 2:
        tmpPC = (jumpDir.read() << 2); // sobreescribimos tmpPC al completo
        tmpPC(31,28) = PC.read()(31,28); // y ponemos bien los cuatro primeros
        break;
    default:
        tmpPC = 0xdeaddead;
    };

    nuevoPC.write(tmpPC);
}
```

```
        state = jump;
    else if (opCode == 0x03)
        state = jal;
```

Por último añadimos el estado jal asociado al opcode 0x3.

```
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32) && !defined(__CYGWIN__)
    instMIPS = new mipsCA("instMIPS", "doc\\programaJAL.txt", "doc\\datos.txt");
#else
    instMIPS = new mipsCA("instMIPS", "doc/programaJAL.txt", "doc/datos.txt");
#endif
```

Para cargar el programa, hemos modificado el archivo Main_methods.cpp para seguir la organización de directorios de nuestro proyecto y asegurar compatibilidad con múltiples sistemas operativos (Windows vs “el resto”).

Asimismo, proveemos pipelines para su compilación y ejecución tanto en MSVC como en GCC, la primera mediante los archivos del proyecto de Visual Studio, y la segunda mediante un makefile.

Detalles con mmuMem.cpp

En nuestra práctica hemos encontrado múltiples problemas debido al ensamblador presente en mmuMem. Hemos añadido las referencias necesarias para codificar la instrucción jal, des-hardcodeado algunos valores para que aceptase la ampliación, y añadido el case al switch para generar la instrucción máquina correcta.

En concreto:

Línea 148: `char* nombres[] = { [...] "nop", "jal" };` <= Añadimos jal

Línea 151: `for (i = 0; i < sizeof(nombres)/sizeof(char*); ++i) {` <= Hacemos que el for se ejecute para el tamaño de nombres, eliminando el 28 hardcodeado.

Línea 274: `if ((inst == 17) || [...] || (inst == 24) || (inst == 28))` <= Añadimos la instucción (jal ahora es la 28)

Línea 278: `if (inst == 24 || inst == 28)` <= Añadimos de nuevo la jal a las instrucciones tipo j

Línea 325: `case 28: maq = (0x03 << 26) | (inm & 0x3ffffff); break; //jal` <= Añadimos la jal al case, y la formateamos.

Test de ejecución:
Salida esperada:

```
 $0 : 00000000    $t0: 00000004    $s0: 00000010    $t8: 00000018
 $at: 00000001    $t1: 0000000c    $s1: 00000011    $t9: 00000019
 $v0: 00000002    $t2: 00000014    $s2: 00000012    $k0: 0000001a
 $v1: 00000003    $t3: 0000001c    $s3: 00000013    $k1: 0000001b
 $a0: 00000004    $t4: 00000024    $s4: 00000014    $gp: 0000001c
 $a1: 00000005    $t5: 0000002c    $s5: 00000015    $sp: 0000001d
 $a2: 00000006    $t6: 00000034    $s6: 00000016    $fp: 0000001e
 $a3: 00000007    $t7: 0000003c    $s7: 00000017    $ra: 00000048
```

Compilación (Linux, Warnings suprimidos):

```
g++ -w -O0 -g -fno-operator-names  -c src/alu.cpp -o obj/alu.o
g++ -w -O0 -g -fno-operator-names  -c src/BRAM.cpp -o obj/BRAM.o
g++ -w -O0 -g -fno-operator-names  -c src/control.cpp -o obj/control.o
g++ -w -O0 -g -fno-operator-names  -c src/DSP48E1.cpp -o obj/DSP48E1.o
g++ -w -O0 -g -fno-operator-names  -c src/gestionPC.cpp -o obj/gestionPC.o
g++ -w -O0 -g -fno-operator-names  -c src/Main_methods.cpp -o obj/Main_methods.o
g++ -w -O0 -g -fno-operator-names  -c src/mmuMem.cpp -o obj/mmuMem.o
g++ -w -O0 -g -fno-operator-names  -c src/muxes.cpp -o obj/muxes.o
g++ -w -O0 -g -fno-operator-names  -c src/registersBank.cpp -o obj/registersBank.o
g++ -w -O0 -g -fno-operator-names  -c src/shifter.cpp -o obj/shifter.o
g++ -w -O0 -g -fno-operator-names  -o target/Main_methods obj/alu.o obj/BRAM.o obj/control.o
obj/DSP48E1.o obj/gestionPC.o obj/Main_methods.o obj/mmuMem.o obj/muxes.o obj/registersBank.o
obj/shifter.o -lsystemc
```

Salida de ejecución (`./target/Main_methods`):

```
        SystemC 2.3.3-Accellera --- Feb 21 2020 12:03:58
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
mipsCA: top.instMIPS
alu: top.instMIPS.instalu
DSP48E1: top.instMIPS.instalu.dsp48e1
control: top.instMIPS.instcontrol
gestionPC: top.instMIPS.instgestionPC
mmuMem: top.instMIPS.instmmuMem  doc/programaJAL.txt doc/datos.txt
BRAM: top.instMIPS.instmmuMem.code
BRAM: top.instMIPS.instmmuMem.data
BRAM: top.instMIPS.instmmuMem.interna0
BRAM: top.instMIPS.instmmuMem.interna1
muxes: top.instMIPS.instmuxes
shifter: top.instMIPS.instshifter
registersBank: top.instMIPS.instregistersBank

Info: /OSCI/SystemC: Simulation stopped by user.
$0 : 00000000    $t0: 00000004    $s0: 00000010    $t8: 00000018
$at: 00000001    $t1: 0000000c    $s1: 00000011    $t9: 00000019
$v0: 00000002    $t2: 00000014    $s2: 00000012    $k0: 0000001a
$v1: 00000003    $t3: 0000001c    $s3: 00000013    $k1: 0000001b
$a0: 00000004    $t4: 00000024    $s4: 00000014    $gp: 0000001c
$a1: 00000005    $t5: 0000002c    $s5: 00000015    $sp: 0000001d
$a2: 00000006    $t6: 00000034    $s6: 00000016    $fp: 0000001e
$a3: 00000007    $t7: 0000003c    $s7: 00000017    $ra: 00000048
```

Como podemos apreciar por la columna izquierda, por la ausencia de ‘dead’ al final de $s0, y por que los resultados, efectivamente coinciden con los que se piden, la implementación realiza lo que se pide.