Proyecto de Ingeniería de Computadores (PEC)

Objetivo del curso

El objetivo de esta asignatura es que el alumno aprenda a desarrollar un prototipo de un computador o un SoC (System on Chip) en un chip programable sobre una placa base para crear un mini-ordenador. Se pondrán en práctica algunos de los conocimientos adquiridos en asignaturas anteriores sobre el diseño de la microarquitectura de un procesador, sobre el diseño e implementación de software de sistema, y sobre el diseño de sistemas digitales.

Fases principales del proyecto.

1) Aprendizaje de las herramientas de desarrollo para los chips programables (FPGA) y práctica del lenguaje de descripción del hardware VHDL.
2) Implementar pequeños componentes o dispositivos en el chip programable de la placa base.
3) Implementar una primera versión simplificada del procesador en una FPGA (sin memoria externa, ni soporte para el sistema operativo o dispositivos externos)
4) Implementar una versión completa del procesador.
5) Programar un sistema de arranque (BIOS) para el Sistema Operativo en el procesador.
6) Evaluar el rendimiento de varias aplicaciones sobre la plataforma que se ha diseñado.

Objetivo de las sesiones

En estas sesiones vamos a implementar físicamente un procesador sencillo en la FPGA. Lo haremos por etapas. En cada etapa cogeremos el trabajo realizado en la anterior y le añadiremos o modificaremos algún componente.

Etapa 3: La ALU.

En esta etapa daremos soporte en la ALU para las instrucciones aritmético-lógicas (sumas con inmediato, multiplicaciones y divisiones de enteros) y comparaciones. No trataremos con el tema de las operaciones de punto flotante porque añaden complejidad al circuito y siempre se pueden emular con operaciones de números enteros de forma transparente con el compilador.

El diseño que se haga se podrá probar directamente sobre la placa de desarrollo ya que como ya puede realizar escrituras a memoria, podemos verificar que las operaciones de ALU funcionen correctamente.

Las instrucciones a implementar son las comparaciones: CMPLT, CMPLE, CMPEQ, CMPLTU y CMPLEU, y las aritmético-lógicas: ADDI, AND, OR, XOR, NOT, ADD, SUB, SHA, SHL, MUL, MULH, MULHU, DIV y DIVU.

La implementación de la ALU completa será libre, pudiendo cada uno escoger cómo la implementa y cuáles serán sus señales de control. Así que en este documento no daremos las cabeceras de la ALU. Simplemente recordad que para que siga encajando con el *datapath* ya diseñado, la ALU deberá tener dos entradas *x* e *y* de 16 bits (los dos operandos sobre los que se aplican las operaciones), y una salida *w* de 16 bits para el resultado de la operación que realiza la ALU. Además, ahora deberemos modificar el tamaño de la señal de control *op* para adecuarlo al número de operaciones distintas que debe codificar y finalmente añadir un bit *z* de salida que indica si el valor de la salida *w* de la ALU es igual o distinto del valor 0. Este bit será necesario para cuando implementemos las instrucciones de salto.

No debemos olvidar que la ALU debe seguir dando soporte al grupo de operaciones de movimiento (MOVI y MOVHI) y también debe de seguir realizando el cálculo de la dirección efectiva para las instrucciones de acceso a memoria (LD, ST, LDB y STB).

Como el número de operaciones que debe hacer la ALU es elevado, la señal *op* debería tener suficientes bits. Esto incrementará la complejidad de la lógica de la unidad de control que se encarga de generar esta señal. Si consultáis como se implementaba el procesador SISP-I-1 podréis ver que la estrategia que se utilizaba no era exactamente tener una señal

op que codificase inequívocamente a cada operación posible, si no que la señal *op* indicaba un tipo de operaciones y había otra señal de control, llamada *f*, que indicaba de que operación concreta se trataba dentro del grupo de operaciones que indicaba la señal *op*. Esta estrategia de implementación hace la lógica de control un poco más simple y fácil de entender, pero sois libres de implementarlo como queráis.

Operaciones de comparación

Las instrucciones de comparación se utilizan para implementar los bucles y sentencias condicionales. Recordemos las principales características de estas instrucciones (la información completa la tenéis en los anexos del SISA):

- CMPLT: compara dos operandos enteros *a* y *b*, y determina si *a* es menor que *b*. Es decir, calcula $a < b$. Si es así, entonces escribe en el registro destino el valor 1, de lo contrario escribe un 0.
- CMPLE: compara dos operandos enteros *a* y *b*, y determina si *a* es menor o igual que *b*. Es decir, calcula $a \leq b$. En este caso, escribe en el registro destino el valor 1, de lo contrario escribe un 0.
- CMPLEQ: escribe el valor 1 en el registro destino si $a = b$, de lo contrario escribe un 0.
- CMPLTU: igual que CMPLT pero para números naturales *a* y *b*.
- CMPLEU: igual que CMPLE pero para números naturales *a* y *b*.

Operaciones aritmético-lógicas

Hagamos un breve repaso de las operaciones que debe realizar la ALU:

- Hacer la and bit a bit entre las entradas *x* e *y* para poder ejecutar la instrucción AND.
- Hacer la or bit a bit entre las entradas *x* e *y* para poder ejecutar la instrucción OR.
- Hacer la xor bit a bit entre las entradas *x* e *y* para poder ejecutar la instrucción XOR.
- Hacer la not bit a bit de la entrada *x* para poder ejecutar la instrucción NOT.
- Hacer la suma de las entradas *x* e *y* para poder calcular la dirección efectiva para las instrucciones de acceso a memoria y poder ejecutar las instrucciones ADDI y ADD.
- Hacer la resta de las entradas *x* e *y* para poder ejecutar la instrucción SUB.
- Desplazar la entrada *x* tantos bits como indican los 5 bits de menor peso de *y* interpretados en Ca2 para poder ejecutar la instrucción SHA. Si *y* es positivo, desplaza hacia la izquierda, en caso contrario desplaza hacia la derecha extendiendo el signo.
- Desplazar la entrada *x* tantos bits como indican los 5 bits de menor peso de *y* interpretados en Ca2 para poder ejecutar la instrucción SHL, pero esta vez sin extender el signo.
- Hacer la multiplicación de las entradas *x* e *y* para ejecutar las instrucciones MUL, MULH y MULHU. La instrucción MUL devuelve la parte baja (16 bits) de la multiplicación. La instrucción MULH devuelve la parte alta (16 bits) de la multiplicación con signo y la MULHU devuelve la parte alta de la multiplicación pero sin signo.
- Hacer la división entera (con signo) de *x* entre *y* para poder ejecutar la instrucción DIV.
- Hacer la división natural (sin signo) de *x* entre *y* para poder ejecutar la instrucción DIVU.

Implementad un diseño en vhdl para esta ALU. Completad el contenido del fichero **alu.vhd** con vuestra implementación.

Cambios en el datapath

Los cambios que debemos hacer en el *datapath* para que estas instrucciones nuevas funcionen son mínimos. En primer lugar, ahora necesitamos que por el bus *y* no sólo pueda entrar un inmediato, sino el registro *b* leído del Banco de registros. Por lo tanto, añadiremos un nuevo multiplexor, con su respectiva señal de control (*Rb_N*), a la entrada *y* de la

ALU que seleccione si por el bus *y* entrará el valor procedente del registro *b* o bien el inmediato, ya sea directo (ADDI, LDB y STB) o desplazado un bit (LD y ST).

Implementad un diseño en vhdl para el nuevo *datapath*.

Cambios en la unidad de control

El único cambio añadido al módulo de control es la generación de nuevas las señales de control del multiplexor y de la ALU.

Además, debemos añadir la lógica para poder escribir en el registro destino que ahora incluyen las instrucciones nuevas (and, or, xor, not, add, sub, sha, shl, cmplt, cmple, cmpeq, cmpltu y cmpleu, addi, mul, mulh, mulhu, div y divu), aparte de las que ya teníamos en la versión anterior.

Para poder decodificar e implementar correctamente las instrucciones es necesario saber cómo se codifican. La siguiente tabla muestra un resumen la codificación de las nuevas instrucciones.

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0		
0 0 0 0	Rd	Ra	f f f	Rb	Op. Lógicas y Aritméticas	AND, OR, XOR, NOT ADD, SUB, SHA, SHL
0 0 0 1	Rd	Ra	f f f	Rb	Comparación con y sin signo	CMPLT, CMPLE, -, CMPEQ CMPLTU, CMPLEU, -, -
0 0 1 0	Rd	Ra	n n n n n n		Add inmediato	ADDI
1 0 0 0	Rd	Ra	f f f	Rb	Extensión aritmética	MUL, MULH, MULHU, - DIV, DIVU, -, -

Simulación y ejecución de programas

El procesador ya está listo para probar las nuevas instrucciones. Podemos probar el funcionamiento de estas nuevas instrucciones de varias formas

a) Simulando nada más la entidad **proc** con el *stub* de memoria de la etapa 2.1
b) Simulando la entidad **sisa** con el *stub* de memoria de la etapa 2.2 que emulaba a una SRAM comercial.
c) Programando la FPGA con nuestro diseño. Previamente habremos cargado el programa que deseemos ejecutar en la memoria SRAM mediante la aplicación del panel de control de las placas de desarrollo.

Juegos de prueba

Haremos un juego de pruebas para cada grupo de instrucciones y se intentarán probar todas las combinaciones de argumentos que evidencien alguna característica singular. Obviamente no se pueden probar todos los casos, pero sí los más significativos.

Instrucciones de comparación

Escribiremos un programa que pruebe todas las combinaciones de argumentos y podamos ver la salida. Por lo tanto, escribiremos en memoria el resultado de cada comparación. Un ejemplo de un programa de test podría ser el siguiente:

```
movi r0, 0
movi r1, 2
movi r2, 2
movi  r3, 3
```

```
movi  r4, -1
; cmplt
cmplt r5, r2, r3 ; 1
st    0(r0), r5
cmplt r5, r4, r2 ; 1
st    2(r0), r5
cmplt r5, r1, r2 ; 0
st    4(r0), r5
cmplt r5, r3, r0 ; 0
st    6(r0), r5
; cmple
cmple r5, r2, r3 ; 1
st    8(r0), r5
cmple r5, r4, r2 ; 1
st    10(r0), r5
cmple r5, r1, r2 ; 1
st    12(r0), r5
cmple r5, r3, r0 ; 0
st    14(r0), r5
; cmpeq
cmpeq r5, r2, r3 ; 0
st    16(r0), r5
cmpeq r5, r4, r2 ; 0
st    18(r0), r5
cmpeq r5, r1, r2 ; 1
st    20(r0), r5
cmpeq r5, r3, r0 ; 0
st    22(r0), r5
; cmpltu
cmpltu r5, r2, r3 ; 1
st     24(r0), r5
cmpltu r5, r4, r2 ; 0
st     26(r0), r5
cmpltu r5, r1, r2 ; 0
st     28(r0), r5
cmpltu r5, r3, r0 ; 0
st     30(r0), r5
; cmpleu
cmpleu r5, r2, r3 ; 1
st     32(r0), r5
cmpleu r5, r4, r2 ; 0
st     34(r0), r5
cmpleu r5, r1, r2 ; 1
st     36(r0), r5
cmpleu r5, r3, r0 ; 0
st     38(r0), r5
halt
```

Sumas con inmediato

Para probar la instrucción ADDI usaremos inmediatos negativos, positivos y cero. El registro destino será en un caso el mismo que el de lectura y en otro caso será diferente. Podemos observar el siguiente juego de pruebas:

```
movi r0, 0
movi r1, -1
addi r1, r1, -1
st   0(r0), r1 ; -2
addi r2, r1, -2
st   2(r0), r2 ; -4
addi r3, r1, 1
st   4(r0), r3 ; -1
addi r4, r1, 3
st   6(r0), r4 ; 1
addi r1, r1, 0
st   8(r0), r1 ; -2
addi r5, r1, 0
st   10(r0), r5 ; -2
halt
```

Después de ejecutar este código, la memoria debería quedar de la siguiente forma:

```
Mem[01:00] = fffe
Mem[03:02] = fffc
Mem[05:04] = ffff
Mem[07:06] = 0001
Mem[09:08] = fffe
Mem[0b:0a] = fffe
```

Instrucciones lógicas

Para probar el funcionamiento de las instrucciones lógicas, cogeremos valores aleatorios como argumentos y haremos las operaciones AND, OR, XOR y NOT. A continuación compararemos el valor que deberían dar con los resultados de la memoria. Un posible juego de pruebas sería el mostrado en el siguiente código.

```
movi  r0, 0
movi  r1, 0x7d
movhi r1, 0x33
movi  r2, 0x68
movhi r2, 0x43
and   r3, r1, r2
st    0(r0), r3 ; 0x0368
or    r4, r1, r2
st    2(r0), r4 ; 0x737d
not   r5, r2
st    4(r0), r5 ; 0xbc97
xor   r6, r1, r2
st    6(r0), r6 ; 0x7015
halt
```

Después de ejecutar este código, la memoria queda de la siguiente forma:

```
Mem[01:00] = 0368
Mem[03:02] = 737d
Mem[05:04] = bc97
Mem[07:06] = 7015
```

Suma y resta

Aunque en la parte de memoria ya hicimos el cálculo de la dirección efectiva utilizando la ALU, ahora la utilizaremos para comprobar que las sumas y las restas funcionan correctamente. Para probar la instrucción de sumar (ADD), haremos lo siguiente:

- Al mayor número representable le sumaremos 1 para ver cómo se transforma en el más negativo representable.
- Haremos -1 +1 = 0. Negativo más positivo cero.
- También calcularemos -1-1 = -2. Negativo más negativo es negativo.
- Tomaremos el número mínimo representable y le sumaremos -1. Debería convertirse en el máximo representable.

Podemos ver el juego de pruebas representado en el siguiente código:

```
movi  r0, 0
movi  r1, 1
movi  r2, 2
add   r3, r1, r2
st    0(r0), r3 ; 3
movi  r2, 0xff
movhi r2, 0x7f
add   r3, r2, r1
st    2(r0), r3 ; 0x8000
movhi r2, 0xff
add   r3, r1, r2
st    4(r0), r3 ; 0
movi  r1, 0xff
add   r3, r1, r2
st    6(r0), r3 ; 0xfffe
movi  r1, 0
movhi r1, 0x80
add   r3, r1, r2
```

```
st    8(r0), r3 ; 0x7fff
halt
```

El resultado debería ser el siguiente:

```
Mem[01:00] = 0003
Mem[03:02] = 8000
Mem[05:04] = 0000
Mem[07:06] = fffe
Mem[09:08] = 7fff
```

En el caso de la resta haremos unos cálculos similares, representados en el siguiente código.

```
movi  r0, 0
movi  r1, 1
movi  r2, 2
sub   r3, r1, r2
st    0(r0), r3 ; 0xffff
movi  r1, 0xff
movi  r2, 0xff
sub   r3, r1, r2
st    2(r0), r3 ; 0
movi  r2, 0xfe
sub   r3, r1, r2
st    4(r0), r3 ; 1
movi  r1, 0
movhi r1, 0x80
movi  r2, 1
sub   r3, r1, r2 ; 0x7fff
st    6(r0), r3
halt
```

El resultado de estas operaciones es el siguiente:

```
Mem[01:00] = ffff
Mem[03:02] = 0000
Mem[05:04] = 0001
Mem[07:06] = 7fff
```

Desplazamientos

Los desplazamientos en el procesador SISA se realizan con las instrucciones SHL y SHA. En el primer caso es lógico y en el segundo caso el desplazamiento es aritmético. En ambos casos el operando se puede desplazar a la derecha ya la izquierda. Sólo tenemos que hacer que el segundo registro sea positivo (izquierda) o negativo (derecha). Por lo tanto, probaremos las 4 combinaciones diferentes que nos resultan. Estas quedan representadas en el siguiente código.

```
movi r0, 0xbc
movhi r0, 0xa5
movi r1, 0x22
movhi r1, 0x40
movi r2, 3
movi r3, -2
sha r4, r0, r2 ; 2de0
sha r5, r0, r3 ; e96f
shl r6, r1, r2 ; 0110
shl r7, r1, r3 ; 1008
movi r0, 0
st    0(r0), r4
st    2(r0), r5
st    4(r0), r6
st    6(r0), r7
halt
```

Como antes, después de probar este juego de pruebas en la placa de desarrollo, el resultado en la memoria debía quedar de la siguiente manera:

```
Mem[01:00] = 2de0
Mem[03:02] = e96f
Mem[05:04] = 0110
Mem[07:06] = 1008
```

Multiplicaciones y divisiones

Para terminar la verificación de la ALU, probaremos las instrucciones de multiplicación y división. Las pruebas que haremos incluirán multiplicaciones de números positivo × positivo, positivo × negativo y negativo × negativo. También se probarán los valores máximos y las versiones enteras y naturales de la multiplicación. El siguiente código incluye una pequeña prueba de las instrucciones MUL, MULH y MULHU.

```
movi  r0, 0
movi  r1, 1
movi  r2, 2
mul   r3, r1, r2 ; 0002
mulh  r4, r1, r2 ; 0000
mulhu r5, r1, r2 ; 0000
st    0(r0), r3
st    2(r0), r4
st    4(r0), r5
movi  r2, 0xff
movhi r2, 0x7f
mul   r3, r1, r2 ; 7fff
mulh  r4, r1, r2 ; 0000
mulhu r5, r1, r2 ; 0000
st    6(r0), r3
st    8(r0), r4
st    10(r0), r5
movhi r2, 0xff
mul   r3, r1, r2 ; ffff
mulh  r4, r1, r2 ; ffff
mulhu r5, r1, r2 ; 0000
st    12(r0), r3
st    14(r0), r4
st    16(r0), r5
movi  r1, 0xff
mul   r3, r1, r2 ; 0001
mulh  r4, r1, r2 ; 0000
mulhu r5, r1, r2 ; fffe
st    18(r0), r3
st    20(r0), r4
st    22(r0), r5
movi  r1, 0
movhi r1, 0x80
mul   r3, r1, r2 ; 8000
mulh  r4, r1, r2 ; 0000
mulhu r5, r1, r2 ; 7fff
st    24(r0), r3
st    26(r0), r4
st    28(r0), r5
halt
```

Se realizan 5 multiplicaciones enteras y las mismas operaciones para naturales, a saber:

- 1 × 2 = 2
- Natural y entera: 1 × 0x7FFF = 0x7FFF
- Natural: 1 × 0xFFFF = 0xFFFF, entera: 1 × (-1) = -1
- Natural: 0xFFFF × 0xFFFF = 0xFFFE0001, entera: (-1) × (-1) = 1
- Natural: 0x8000 × 0xFFFF = 0x7FFF8000, entera: (-1) × (-32.878) = 0x00008000

El resultado de esta ejecución debería ser el siguiente:

```
Mem[01:00] = 0002
Mem[03:02] = 0000
Mem[05:04] = 0000
Mem[07:06] = 7fff
```

```
Mem[09:08] = 0000
Mem[0b:0a] = 0000
Mem[0d:0c] = ffff
Mem[0f:0e] = ffff
Mem[11:10] = 0000
Mem[13:12] = 0001
Mem[15:14] = 0000
Mem[17:16] = fffe
Mem[19:18] = 8000
Mem[1b:1a] = 0000
Mem[1d:1c] = 7fff
```

En cuanto a la división, también haremos las mismas divisiones, también una con overflow para comparar entre el resultado esperado y el contenido de la memoria:

- 2/1 = 2
- Natural y entera: 1 / 0x7fff = 0
- Natural: 1 / 0xFFFF = 0xFFFF, entera: 1 / (-1) = -1
- Natural: 0xFFFF / 0xFFFF = 1, entera: (-1) / (-1) = 1
- Natural: 0x8000 / 0xFFFF = 0, entera: (-32.768) / (-1) = overflow en 16 bits.

Este juego de pruebas lo tenemos representado en el siguiente código.

```
movi  r0, 0
movi  r1, 1
movi  r2, 2
div   r3, r2, r1 ; 0002
divu  r4, r2, r1 ; 0002
st    0(r0), r3
st    2(r0), r4
movi  r2, 0xff
movhi r2, 0x7f
div   r3, r1, r2 ; 0000
divu  r4, r1, r2 ; 0000
st    4(r0), r3
st    6(r0), r4
movhi r2, 0xff
div   r3, r1, r2 ; ffff
divu  r4, r1, r2 ; 0000
st    8(r0), r3
st    10(r0), r4
movi  r1, 0xff
div   r3, r1, r2 ; 0001
divu  r4, r1, r2 ; 0001
st    12(r0), r3
st    14(r0), r4
movi  r1, 0
movhi r1, 0x80
div   r3, r1, r2 ; 0000
divu  r4, r1, r2 ; 0000
st    16(r0), r3
st    18(r0), r4
halt
```

El resultado debería ser:

```
Mem[01:00] = 0002
Mem[03:02] = 0002
Mem[05:04] = 0000
Mem[07:06] = 0000
Mem[09:08] = ffff
Mem[0b:0a] = 0000
Mem[0d:0c] = 0001
Mem[0f:0e] = 0001
Mem[11:10] = xxxx (overflow)
Mem[13:12] = 0000
```

El resultado que genera la implementación de la división en la FPGA al ejecutar la operación (-32.768) / (-1) es 0x8000. No hagáis mucho caso a este valor hasta que no se implementes las excepciones en el procesador. En todo caso, el resultado no es representable. Además, se puede detectar la situación de *overflow* porqué el primer operando es negativo, el segundo también y el resultado debería dar positivo pero es negativo, por lo tanto en este caso se puede detectar fácilmente esta situación.

Finalmente, el siguiente código hace un último juego de prueba combinado para las instrucciones de multiplicación y división.

```
movi  r0, 64
movi  r1, 3
movi  r2, 4
movi  r3, -2
movi  r4, -7
mul   r5, r1, r2
st    0(r0), r5
mul   r5, r1, r3
st    2(r0), r5
mulh  r5, r1, r2
st    4(r0), r5
mulh  r5, r1, r3
st    6(r0), r5
mulhu r5, r1, r2
st    8(r0), r5
mulhu r5, r1, r3
st    10(r0), r5
div   r5, r2, r1
st    12(r0), r5
div   r5, r0, r3
st    14(r0), r5
divu  r5, r4, r2
st    16(r0), r5
divu  r5, r0, r3
st    18(r0), r5
halt
```

Su resultado debería ser el siguiente:

```
Mem[41:40] = 000c  parte_baja(3*4)
Mem[43:42] = fffa  3*(-2)
Mem[45:44] = 0000  parte_alta(3*4)
Mem[47:46] = ffff  parte_alta(3*(-2))
Mem[49:48] = 0000  parte_alta(3*4)
Mem[4b:4a] = 0002  parte_alta(3*fffe = 2fffa)
Mem[4d:4c] = 0001  4/3
Mem[4f:4e] = ffe0  64/(-2) = -32
Mem[51:50] = 3ffe  fff9/4 = 3ffe
Mem[53:52] = 0000  64/fffe = 0
```