Organización de Computadoras
Trabajo Práctico Integrador

Bargas, Santiago Darío. Grinovero, Jerónimo. Telli, Alejandro Román.
Universidad Nacional del Litoral, Facultad de Ingeniería y Ciencias Hídricas

***Resumen*— En el siguiente informe se presenta una explicación del proceso de desarrollo de un sistema integrado en Verilog y Assembly para realizar multiplicaciones entre números de dos dígitos para su posterior impactado en una FPGA.**

En la primera sección, que es la introducción, se explica la consigna presentada por la cátedra y se realiza una breve descripción del sistema integrado propuesto.

En las siguientes secciones, se explican las decisiones que se tomaron para definir las bases del diseño de los componentes del sistema, tanto en *hardware* como en *software*.

En la última sección se presentan las conclusiones obtenidas tras haber finalizado el trabajo.

I. Introducción

A. Consigna

La consigna del Trabajo Práctico Integrador consiste en el diseño de un sistema integrado (SI) que tiene la capacidad de realizar una operación de multiplicación entre números de 2 dígitos, utilizando un emulador de terminal en una PC como interfaz con el usuario.

El usuario realiza la operación introduciendo en el emulador el primer operando de 2 dígitos utilizando los números del teclado, el signo de multiplicación ("`*`"), el segundo operando con igual formato y la tecla `ENTER`.

El SI realizará la operación y devolverá la cadena "`=XX`", donde `XX` representa el resultado del producto.

B. Descripción del sistema integrado

Este sistema estará compuesto por un módulo de *hardware* y uno de *software*.

- **Módulo de *hardware*:** compuesto por el procesador RV32I diseñado durante el cursado, una UART y un módulo de memoria.
- **Módulo de *software:*** compuesto por el código escrito en lenguaje *Assembly* para implementar el algoritmo.

El hardware se engloba en un **módulo *top*** que integra todos los módulos del sistema. Su función es la de instanciar los módulos específicos y establecer las conexiones para lograr la comunicación y el flujo de datos entre ellos.

II. Diseño e implementación del procesador

El procesador propuesto por la cátedra es el RV32I, que fue construido durante el cursado y consta de dos módulos: el ***datapath*** (camino de datos) y la **unidad de control**.

A. El datapath

El módulo ***datapath*** se encarga de gestionar el camino donde fluyen los datos a medida que se ejecutan las instrucciones, realizando las operaciones aritméticas y/o lógicas necesarias para ejecutar las instrucciones del programa.

Este módulo debe tener la capacidad de implementar las siguientes instrucciones:

- **Carga de dato en registro**, en la forma `lw rd,offset(rs1)`, donde `rd` indica el registro que almacena la dirección donde se cargará el dato, `rs1` indica el registro que almacena la dirección de donde proviene el dato y `offset` es el campo inmediato que indica el desplazamiento en memoria a partir de la posición de `rs1`.
- **Almacenamiento de dato en memoria**, en la forma `sw rs2,offset(rs1)`, donde `rs2` indica el registro que almacena el valor a guardar, `rs1` indica el registro que almacena la dirección donde se almacenará el dato y `offset` es el campo inmediato que indica el desplazamiento en memoria a partir de la posición de `rs1`.
- **Suma con campo inmediato**, en la forma `addi rd,rs1,imm`, `rd` indica el registro que almacena la dirección donde se cargará el dato, `rs1` indica el registro que almacena la dirección de donde proviene el operando e `imm` es el campo inmediato que indica el número a sumar al valor de `rs1`.
- **Aritmético-lógicas entre registros**, en la forma `nombre rd,rs1,rs2`, donde `nombre` indica el nombre de la operación y `rs1` y `rs2` indican los registros que almacenan los valores de los operandos. Las

operaciones podrán ser de suma, resta, multiplicación, división, módulo, menor o igual que, AND lógico, y OR lógico.

- **Salto condicional si dos registros almacenan el mismo valor**, en la forma `beq rs1,rs2,imm`, donde `rs1` y `rs2` indican los registros que almacenan los valores a comparar e `imm` almacena la ubicación a realizar el salto.
- **Salto incondicional**, en la forma `jal rd,imm`, donde `rd` es el registro al que se le escribe la dirección de la siguiente instrucción y `imm` es la ubicación a realizar el salto.

Para implementar estas instrucciones, el *datapath* deberá contar con los siguientes sub-módulos:

- **Contador de programa**: registro de 16 bits que actúa como el contador de programa. La salida (`pc`) se actualiza con el valor de entrada (`pcNext`) en cada flanco positivo del reloj (`clk`).
- **Banco de registros**: banco de 32 registros de 32 bits. Las entradas (`a1, a2, a3, wd3`) determinan las operaciones de lectura y escritura, y `we` indica si se debe escribir. Las salidas (`rd1, rd2`) contienen los datos leídos.
- **Extensión de signo**: se encarga de extender el bit de signo de la entrada inmediata (`imm`) a 32 bits.
- **Unidad Aritmético Lógica**: realiza las operaciones aritméticas y lógicas necesarias para implementar las instrucciones mencionadas anteriormente. La salida `res` contiene el resultado de la operación.
- **Sumador**: realiza la suma de dos operandos (`op1` y `op2`), utilizada para operaciones específicas en el camino de datos
- **Multiplexores 2x1 y 3x1**: multiplexores de 2 a 1 y 3 a 1 de 32 bits. Este tipo de módulo selecciona una de las dos o tres entradas basándose en la señal de selección (`sel`).

B. La unidad de control

El módulo **unidad de control** se encarga de generar señales de control de los distintos módulos del flujo de datos para manejarlo. Cuenta con los siguientes sub-módulos:

- **Decodificador de operación principal**: toma el campo de operación (`op`) de una instrucción en el formato RV32I y genera señales de control para dirigir el camino de datos del procesador. Estas señales se definen de acuerdo a la siguiente tabla:

op	Ins	RW	IS	As	MW	RS	B	Ao	J
3	lw	1	00	1	0	01	0	00	0
19	addi	1	00	1	0	00	0	10	0
35	sw	0	01	1	1	xx	0	00	0
51	R-t	1	xx	0	0	00	0	10	0
99	beq	0	10	0	0	xx	1	01	0
111	jal	1	11	x	0	10	0	xx	1

  Las columnas representan los valores de: campo `op` de la instrucción, nombre de la instrucción, `RegWrite`, `ImmSrc`, `ALUsrc`, `MemWrite`, `ResultSrc`, `Branch`, `ALUop` y `Jump`.
- **Decodificador de ALU**: toma señales de control específicas y genera las señales de control necesarias para configurar la ALU de acuerdo con la instrucción que se está ejecutando. Estas señales se definen de acuerdo a la siguiente tabla:

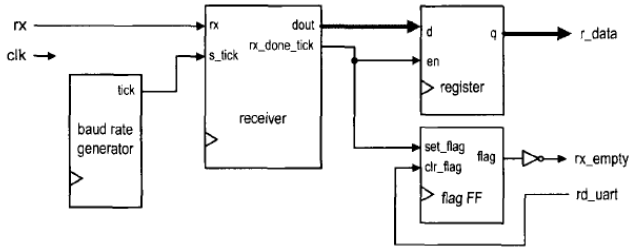
Instrucción	f3	op_5	$f7_5$	$f7_0$	$ALUctrl_{2:0}$
lw	xxx	x	x	x	000
addi	xxx	x	x	x	000
sw	xxx	x	x	x	000
add	000	1	0	0	000
mul	000	1	0	1	100
sub	000	1	1	0	001
slt	010	x	x	x	101
div	100	x	x	x	110
or	110	1	0	0	011
rem	110	1	0	1	111
and	111	x	x	x	010
beq	xxx	x	x	x	001
jal	xxx	x	x	x	000

- **Generador de señal de control** `pcSrc`: se utilizan las señales `zero` y `Branch` para decidir si la próxima dirección del contador de programa se toma del resultado de una operación de salto condicional.

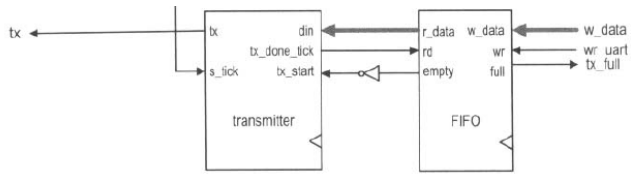
III. Diseño e implmentación de la UART

La **UART** ***(Universal Asynchronous Receiver/Transmitter)*** recibe desde el emulador el código ASCII correspondiente a la tecla presionada en el teclado en formato serie y lo convierte a paralelo. Este número se convierte a paralelo y se almacena en un *buffer* de 8 bits. Una vez que el código de entrada haya sido convertido, la UART informa de la recepción del dato al procesador mediante una bandera.

Una vez realizada la multiplicación, el procesador envía los resultados de los números en formato paralelo a la UART, que los recibe en una memoria FIFO, para luego convertirlos a formato



serie y enviarlos al emulador con una señal de salida.



El código pertinente en Verilog se encuentra anexado en la carpeta junto al informe.

IV. Diseño e implementación de la memoria

El **módulo de memoria** se encarga de la lectura de instrucciones y la carga y almacenamiento de datos.

Está compuesto por los siguientes sub-módulos:

- **Memoria de instrucciones:** memoria de sólo lectura de 64 registro de 32 bits. La salida (`rd`) se selecciona según la dirección de entrada (`address`). Aquí se cargará el programa antes de iniciar la simulación.
- **Memoria de datos**: memoria de acceso aleatorio de 32 registros de 32 bits. La salida (`rd`) se selecciona según la dirección de entrada (`address`), y la escritura se controla mediante el selector de lectura-escritura `we` y el dato a escribir `wd`.
- **MMU (*Memory Management Unit*):** se encarga de traducir las direcciones virtuales a direcciones físicas.
- **MMIO**: almacena los datos que le envía la UART tras haberlos leído. Compuesta de 4 registros de 32 bits.

La memoria se organiza virtualmente de forma lineal y consecutiva, pero se construye físicamente a partir de diferentes sub-módulos. Por lo tanto, se debe realizar un mapeo virtual para garantizar esa organización y así poder acceder mediante un mismo sistema a las memorias de datos y de entrada-salida.

El espacio virtual de la memoria de datos se define desde la posición virtual 0 (`0x0000`) hasta la posición virtual 127 (`0x007D`) inclusive, y el comienzo de la memoria de entrada-salida se define en la posición virtual 128 (`0x0080`) hasta la posición virtual 143 (`0x008F`).

V. Diseño e implementación del software

El módulo de software está compuesto por el código escrito en lenguaje *Assembly* para implementar el algoritmo de entrada-salida y multiplicación.

El manejo de la entrada se implementó con dos bucles. En el primer bucle se lee la posición de memoria en donde se encuentra almacenada la bandera de la UART que informa si se recibió un dato. Una vez que se recibe un dato, se entra al segundo bucle para verificar si ya fueron leídos los 5 datos correspondientes para realizar la multiplicación. En caso de no haber leído los 5 aún, se lee el dato que recibió la UART. En caso de haberse leído ya los 5 datos, se avanza a la multiplicación.

Se multiplican los números recibidos con la instrucción `mul` y se descompone el resultado en sus dígitos decimales, a los que se les suma 48 para convertirlos a código ASCII y se los almacena en un registro especial donde la UART los leerá al recibir una bandera.

El código Assembly resultante se carga en la memoria de instrucciones y el algoritmo queda listo para ser ejecutado una vez sea encendido el sistema el procesador.

VI. Conclusiones

El sistema integrado desarrollado permite al usuario calcular el producto entre dos números decimales de dos cifras de forma exitosa.

A pesar de que se logró el funcionamiento en el *testbench*, nos enfrentamos a un desafío al intentar implementarlo en una FPGA. La sintetización del código en Verilog implica consideraciones adicionales, como tiempos de reloj, recursos disponibles en la FPGA y posibles problemas de señales. Es necesario realizar modificaciones en el código y pruebas exhaustivas en el hardware para garantizar un rendimiento robusto y sin errores.

Este proyecto ha proporcionado una experiencia valiosa en el diseño de sistemas integrados, destacando la importancia de la planificación, la modularidad y la depuración para lograr un funcionamiento exitoso. A través de los desafíos encontrados, se han adquirido conocimientos cruciales que pueden aplicarse en proyectos futuros para mejorar la eficiencia y la efectividad de los sistemas integradores.