ESTRATEGIAS DE VERIFICACIÓN FUNCIONAL DE INSTRUCCIONES EN DESCRIPCIONES HARDWARE DE MICROPROCESADOR

FUNCTIONAL VERIFICATION STRATEGIES OF INSTRUCTIONS IN MICROPROCESSOR HARDWARE DESCRIPTIONS

Carlos Andrés Guerra Sandoval
CUCEI - Universidad de Guadalajara, México
carlos.guerra7893@alumnos.udg.mx

Luis García Magallón
CUCEI - Universidad de Guadalajara, México
luis.gmagallon@alumnos.udg.mx

Karen Estefanía González Hernández
CUCEI - Universidad de Guadalajara, México
karen.gonzalez0663@alumnos.udg.mx

Salatiel García Moreno
CUCEI - Universidad de Guadalajara, México
salatiel.garciam@gmail.com

Iván Rodrigo Padilla Cantoya
CUCEI - Universidad de Guadalajara, México
ivan.padilla@academicos.udg.mx

Marco Antonio Gurrola Navarro
CUCEI - Universidad de Guadalajara, México
marco.gurrola@academicos.udg.mx

Recepción: 11/julio/2023 **Aceptación:** 11/octubre/2023

Resumen

Se presenta una estrategia para verificar la correcta ejecución de las instrucciones en la descripción hardware de una microarquitectura de procesador. El proceso de diseño de un microprocesador implica realizar una descripción hardware (empleando VHDL o Verilog) capaz de decodificar y ejecutar todas las instrucciones de la arquitectura de procesador seleccionada (por ejemplo, ARM, RISC-V, etc). En este trabajo se presenta una estrategia para verificar que una

descripción hardware (microarquitectura) como la mencionada realmente ejecuta las instrucciones de la manera esperada ante un amplio conjunto de condiciones. Aunque lo ideal sería comprobar que una instrucción se ejecute correctamente ante todas las combinaciones de datos de entrada y de registros posibles, en el presente trabajo se muestra cómo establecer un compromiso entre la cantidad de combinaciones y el tiempo requerido para su simulación. Además, se explica cómo evitar que el tamaño de los archivos generados durante la simulación sature la memoria o el disco duro de nuestro equipo de cómputo.

Palabras Clave: microarquitectura, microprocesador, verificación funcional, VHDL.

Abstract

A strategy is presented to verify the correct execution of the instructions in the hardware description of a processor microarchitecture. The process of designing a microprocessor involves making a hardware description (using VHDL or Verilog) capable of decoding and executing all the instructions of the selected processor architecture (e.g., ARM, RISC-V, etc.). In this work, a strategy is presented to verify that a hardware description (microarchitecture) like the one mentioned really executes the instructions in the expected way under a wide set of conditions. Although the ideal would be to check that an instruction is executed correctly before all possible combinations of input data and registers, in this paper we show how to establish a compromise between the number of combinations and the time required for their simulation. In addition, it explains how to prevent the size of the files generated during the simulation from saturating the memory or hard drive of our computer equipment.

Keywords: *functional verification, microarchitecture, microprocessor, VHDL.*

1. Introducción

Los microprocesadores son un componente esencial en una cantidad, cada vez mayor de artículos de electrónica de consumo, sensores automotrices, equipos biomédicos, dispositivos de internet de las cosas, etc. Existen procesadores comerciales con una variada capacidad de procesamiento. Desde procesadores de

4 bits para aplicaciones muy simples, hasta procesadores de 64 bits con múltiples núcleos tan usuales en los equipos de cómputo y dispositivos móviles [1].

Debido a la gran cantidad de compuertas lógicas que se requiere para confeccionar un procesador, éstos deben fabricarse en alguna tecnología de Circuito Integrado (CI) de muy alta escala de integración (*Very Large Scale Integration* o VLSI). Por ejemplo, en alguna de las tecnologías ofrecidas por la empresa TSMC de 180 y 65 nm, que permiten escalas de integración de más de un millón de transistores por mm^2 [2].

Otra opción para la implementación de un procesador es en la forma de un *core* embebido en un FPGA. Donde el FPGA es en sí mismo un CI, pero con la capacidad de ser configurado de diferentes maneras.

Además de un procesador, los sistemas electrónicos pueden contener muchos otros componentes, como memoria de programa, memoria de datos, módulos de comunicación, regulador de voltaje, sensores, etc. Es usual contar con estos componentes como elementos independientes pero soldados sobre una tarjeta de circuito impreso (*Printed Circuit Board o PCB*). Opcionalmente, varios de estos componentes se pueden fabricar en un mismo CI, lo que resulta en sistemas de menor tamaño, menos propensos a fallas, que consumen menos energía, y cuya producción en grandes volúmenes resulta más económica. Un grupo de este tipo de CI son los microcontroladores.

Para la implementación de un procesador o de un microcontrolador, ya sea en CI o en FPGA, es necesario primero contar con el módulo de propiedad intelectual (*Intellectual Property o IP*) el cual consiste en la descripción de la microarquitectura realizada empleando algún lenguaje de descripción de hardware (*Hardware Description Language o HDL*) tal como VHDL o Verilog. Este tipo de módulos IP se pueden adquirir con el pago de una licencia a alguna empresa de diseño.

Desarrollo *in-house* de módulos IP

Actualmente es posible desarrollar módulos IP de microprocesador o microcontrolador en el ámbito académico o por aficionados entusiastas con suficientes conocimientos de circuitos digitales. Ejemplos de esto se listan en los

sitios [3] y [4]. Otro ejemplo se presenta en [5] donde se muestra el diseño de un procesador de propósito específico que implementa el filtro Kalman para aplicaciones inerciales, mientras que en [6] se describe el procedimiento para diseñar el módulo IP de un procesador que cumple con la Arquitectura de Conjunto de Instrucciones (*Instruction Set Architecture* o ISA) denominada RISC-V [7].

Los beneficios de contar con un módulo IP de procesador propio es que, además de no tener que pagar licencias para poder usarlo o explotarlo comercialmente, tendremos la posibilidad de modificarlo u optimizarlo según nuestras necesidades.

A continuación, se listan algunas condiciones actuales que favorecen el desarrollo *in-house* (con recursos propios) de cores de procesador:

- Se encuentran disponibles varias ISA libres de pago de regalías, tales como RISC-V [7], Power [8], SPARC [9], entre otras.
- También tenemos la posibilidad de crear desde cero nuestra propia ISA. Pero debido a que tal ISA no dispondría a su alrededor de un ecosistema de herramientas de desarrollo, en la práctica esto funciona más bien como un ejercicio puramente académico.
- Las licencias de software EDA (Electronic Design Automation) profesionales para el desarrollo de módulos IP y para el diseño de planos de circuito integrado son muy costosas, tal es el caso de Synopsys [10], Cadence [11], y Siemens EDA [12]. Son costosas aún en sus versiones con descuento para instituciones académicas. Sin embargo, tenemos disponibles herramientas EDA de acceso libre como Alliance [13], Alliance/Coriolis [14], OpenLane [15], y QFlow [16]. Estas herramientas, aún y cuando no nos ofrecen las mismas prestaciones que el software profesional, son suficientes para desarrollar sistemas de pequeño tamaño para aplicaciones de internet de las cosas y sistemas embebidos.
- Aunque la fabricación de un módulo IP en circuito integrado tiene un costo mínimo de USD 5,800 en [17], y de EUR 4,433 en [18], el prototipado también se puede realizar en una tarjeta de evaluación FPGA, por ejemplo, en la DE10-Lite [19] que tiene un costo de USD 82 para estudiantes y profesores. Esta tarjeta se basa en el FPGA MAX-10 que nos permite probar diseños que fabricados en CI podrían requerir de aproximadamente 500,000 transistores.

En el trabajo [6] se presenta el proceso de elaboración de una IP de procesador que ejecuta las instrucciones de la ISA RISC-V de 32 bits. Al final de este proceso se obtienen varios archivos en HDL con la descripción de la IP.Como complemento al trabajo mencionado, en el presente trabajo se presenta una estrategia para verificar si la descripción HDL de una microarquitectura realmente ejecuta de la manera esperada las instrucciones (código máquina en la memoria de programa) de la ISA seleccionada. A esto se le denomina verificación funcional.

Familia MCS-51

Para mostrar la estrategia de verificación propuesta, en este trabajo empleamos la IP denominada **core tlq31** desarrollada en lenguaje VHDL por Marco Gurrola, coautor del presente trabajo. La microarquitectura ejecuta las instrucciones del conocido microcontrolador 8031, el miembro más sencillo de la familia MCS-51.

La familia de microcontroladores MCS-51 fue lanzada por Intel en 1980 y su uso comercial continúa hasta la actualidad, principalmente en la forma de *cores* IP embebidos.

La arquitectura original maneja datos de 8 bits, e incluye instrucciones de 1, 2 y 3 bytes. Su conjunto de instrucciones es de tipo CISC (*Complex Instruction Set Computer*). Intel permitió a diversos fabricantes desarrollar sus propias versiones de este dispositivo, muchas de ellos con microarquitecturas diferentes pero capaces de ejecutar todas las instrucciones del microcontrolador original [20].

El MCS-51 posee arquitectura Harvard (memorias de programa y de datos separadas) y fue proyectado para usarse en sistemas embebidos. Inicialmente se creó mediante la tecnología NMOS, pero posteriormente Intel relanzó una versión de bajo consumo de energía fabricado con tecnología CMOS [21]. Los microcontroladores MCS-51 de mayores prestaciones poseen 256 bytes de RAM interna y 4 KB de memoria de programa integrada, tres temporizadores-contadores, un puerto serial y cuatro puertos de 8 bits cada uno configurables bit a bit. Por otra parte, la versión más simple, el 8031, sólo cuenta con 128 bytes de RAM interna, toda su memoria de programa debe ser externa, y sólo cuenta con dos temporizadores contadores [22].

Verificación de la ejecución de instrucciones

El diseño de una microarquitectura de procesador es un proceso de un alto nivel de complejidad debido a la cantidad de instrucciones y de condiciones en la que estas se pueden ejecutar. Es esencial contar con mecanismos de verificación que nos permitan detectar los errores de funcionamiento conforme se van diseñando y codificando en HDL las diferentes partes del sistema. Específicamente, queremos ser capaces de detectar si la ejecución de alguna instrucción no está produciendo los resultados esperados.

Es recomendable situar la verificación en las etapas más tempranas del proceso de desarrollo, idealmente después de la implementación de cualquier cambio en la microarquitectura. Esto con la finalidad de reducir el tiempo y esfuerzo invertidos en la detección de las fallas.

Al realizar cambios a la descripción hardware de una microarquitectura, ya sea porque se está tratando de corregir un *bug* detectado (error en la descripción HDL), o bien para añadir nuevas funcionalidades, es necesario verificar que el error ha sido corregido o que la nueva funcionalidad se ejecuta correctamente. Asimismo, hay que verificar que las demás operaciones siguen funcionando de manera correcta tras los cambios.

Verificación básica inicial

En el trabajo [6] se muestra cómo, una vez capturada la microarquitectura en HDL, se realizan pruebas iniciales del *core* mediante la ejecución de pequeños programas de no más de diez instrucciones cada uno. Para ello se emplea el simulador digital de uso libre Asimut [13]. La prueba tiene el propósito de verificar que la microarquitectura realiza correctamente acciones básicas como: la actualización del registro contador de programa después de cada ciclo de instrucción; la actualización del registro de instrucción tras la lectura de la memoria de programa; que el bus de datos hace llegar el resultado de la operación al registro destino, etc. Tras corregir los errores de código que normalmente se detectan en este punto, el *core* ejecutará correctamente el pequeño programa inicial y es entonces cuando decimos que nuestro procesador está “caminando”.

En el trabajo [6] se propone una segunda prueba que consiste en cargar la microarquitectura en una tarjeta de evaluación FPGA, para luego ejecutar un programa en tiempo real que contiene varias porciones de código de 5 o 6 instrucciones cada una. Cada porción de código tiene el propósito de probar, una por una, todas las instrucciones de la ISA. Al final de cada porción de código la ejecución hace una pausa para enviar el resultado a un conjunto de *displays* de 7 segmentos para su verificación visual. Una prueba como esta tiene la desventaja de no realizarse de manera automática y de verificar cada instrucción en una sola o en muy pocas condiciones de operación.

Combinaciones en la ejecución de una instrucción

El microcontrolador 8031 ejecuta 111 tipos de instrucciones diferentes [22] y cada tipo se puede ejecutar con diferentes condiciones de operación. Por ejemplo, para realizar una verificación completa de la instrucción **MUL AB**, que multiplica el contenido de 8 bits de los registros A y B, se deben comprobar un total de $2^8 x 2^8$ = 65,536 multiplicaciones diferentes. En cambio, la instrucción **ADDC A,direct** que suma el contenido del registro A con el contenido de alguno de los 148 registros con capacidad de direccionamiento directo, más el contenido de la bandera de acarreo C, posee en total $2^8 x 2^8 x 148 x 2$ = 19,398,656 de combinaciones diferentes posibles. En este trabajo se propone una estrategia para que, una vez elegido un conjunto de pruebas, generalmente menor que el universo total de combinaciones posibles, se realice de manera automática una simulación por cada combinación elegida.

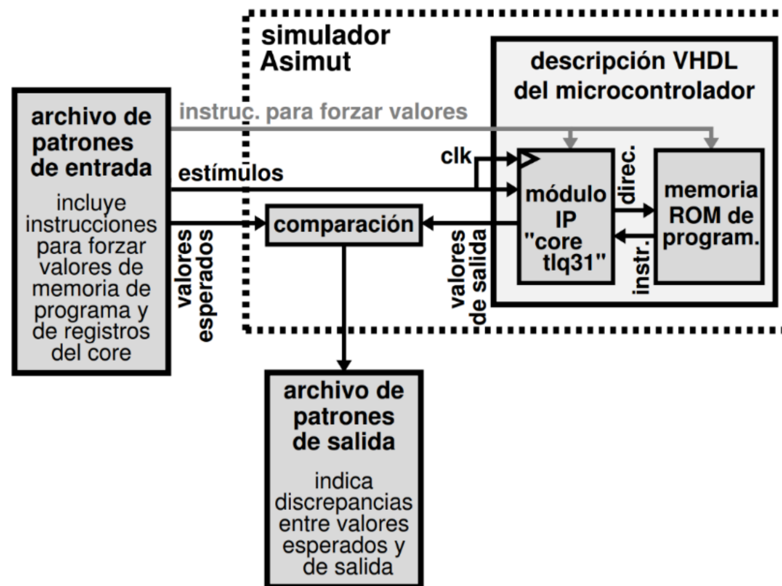
2. Métodos

Herramienta de verificación lógica Asimut

Para la simulación de la descripción HDL de la microarquitectura proponemos emplear el simulador lógico Asimut incluido en las herramientas de uso libre "*Alliance VLSI Cad System*" [13] desarrolladas en el LIP6 de la Universidad Sorbona. Asimut es un simulador lógico que se emplea para verificar que ante ciertos estímulos de entrada la descripción hardware en lenguaje VHDL del sistema entrega los resultados esperados.

Este software corre en el sistema operativo Linux, ejecutándose con un comando desde el emulador de terminal.

En la figura 1 se muestra el banco de pruebas a nivel simulación propuesto. Asimut primero carga la **descripción VHDL del microcontrolador** que consta del módulo IP que queremos probar, en este caso el **core tlq31**, y de la **memoria ROM de programa** de la cual obtiene las instrucciones a ejecutar.



Fuente: elaboración propia.

Figura 1 Banco de pruebas basado en el simulador Asimut.

Luego, Asimut abre un **archivo de patrones de entrada** que es un archivo de texto que contiene:

- **Estímulos** o valores de entrada (incluida la señal de reloj **clk**) que se conducen hasta las entradas del *core*.
- **Los valores esperados** que se espera que el *core* genere ante los valores de entrada aplicados.
- **Instrucciones para forzar valores** en algunas localidades de la ROM y en registros del *core* seleccionados.Las **instrucciones para forzar valores** nos permiten cambiar los valores de los elementos de memoria seleccionados. Esto tiene el propósito de llevar al sistema a un estado determinado a partir del cual Asimut puede continuar con la simulación.

Asimut compara si los **valores esperados** son iguales a los **valores de salida** producidos por el *core*. Los **valores de salida** resultantes en la simulación se guardan en el **archivo de patrones de salida** el cual contiene mensajes de error cuando los **valores de salida** discrepan de los **valores esperados**.

Nota: Para esta prueba la descripción VHDL de la memoria ROM se implementa con registros de *flip-flops*, ya que estos admiten el uso de instrucciones para forzar sus valores. Tras terminar la verificación de las instrucciones, esta memoria se puede substituir por otra más económica, por ejemplo, por una ROM auténtica.

Verificación para una sola instrucción

Para la verificación de la correcta ejecución de una instrucción por parte del *core* se debe preparar un archivo de patrones de entrada como el mostrado en la figura 2, el cual se preparó para verificar la instrucción **ADDC A,#data** de la familia MCS-51 [22]. Esta es una instrucción de 2 bytes de longitud que realiza una suma de 8 bits entre el contenido del registro acumulador **A**, más el dato inmediato **#data**, más el valor de la bandera de acarreo, bit **PSW(7)**.

En la parte superior del archivo se declaran los puertos de entrada y salida del sistema tal como aparecen en la descripción VHDL del **core tlq31**. Después se declaran algunos registros y/o algunas señales internas del sistema que se eligen, dependiendo de la instrucción a verificar, para facilitar la observación de los resultados de simulación sin la necesidad de usar los puertos de salida.

Los estímulos se incluyen en los renglones que cuentan con la indicación de incremento de tiempo **< +1ps>**. Estos renglones cuentan con una columna de valores por cada puerto de entrada. Aunque el incremento de tiempo de 1 ps entre un conjunto de patrones y otro es muy pequeño, éste no presenta problemas pues el código VHDL que se simula se encuentra descrito a nivel de expresiones booleanas que generan un resultado de manera instantánea (en cambio para descripciones VHDL a nivel de celdas lógicas fabricables el incremento de tiempo debe ser mayor pues éstas presentan un retardo de propagación). En cada renglón de patrones el orden de los valores en las columnas está definido por el orden en que se declararon los puertos, registros y señales internas en la parte superior del

archivo. El formato se puede declarar como **B** o como **X** dependiendo si los valores se expresan en binario o en hexadecimal, respectivamente.

En el primer par de renglones de estímulos (patrones) se activa la señal **rst**, mientras que en los renglones de patrones posteriores esta señal se mantiene inactiva. Después vienen dos instrucciones para forzar los valores en el primer par de localidades de la ROM con los códigos $\mathbf{34_{hex}}$ y $\mathbf{00_{hex}}$ (código que varía dependiendo de la instrucción). Con el primer byte se identifica la instrucción **ADDC A,#data**. El segundo byte $\mathbf{00_{hex}}$ representa el valor **#data**, y se forzará a un nuevo valor cada que se realice otra simulación de la instrucción. No se requiere código más allá del segundo byte pues en cuanto se termine la ejecución de esta única instrucción se fuerza el valor del contador de programa al valor de dirección $\mathbf{0000_{hex}}$, repitiéndose el proceso.

En la figura 2, se observa que después de las líneas que fuerzan el código máquina siguen varios pares de renglones donde la señal de reloj **clk** va de **0** a **1**, y estos se repiten hasta completar las funciones de inicialización interna del sistema (19 veces, para el caso específico de la microarquitectura **core tlq31**). Luego viene un ciclo de instrucción que inicia con tres instrucciones para forzar los valores que se van a sumar con la instrucción **ADDC A,#data**. Estos valores son el primer sumando que se carga en el registro **A** (declarado en la microarquitectura como registro **tlq31.a**), el segundo sumando guardado como dato inmediato (**#data**) en la dirección de memoria $\mathbf{0001_{hex}}$ (declarado en la microarquitectura como registro **prom31.r0001**), y finalmente el valor del acarreo de entrada (declarado en la microarquitectura como el bit de índice 7 del registro **tlq31.psw**). Posteriormente viene la instrucción que fuerza el valor del contador de programa a la dirección $\mathbf{0000_{hex}}$, pues cada vez que se termina de ejecutar la instrucción el sistema queda en la dirección $\mathbf{0002_{hex}}$ (se cambia sólo la parte baja declarada como **tlq31.pcl**). Después, se replica el par de renglones, donde la señal **clk** va de **0** a **1**, tantas veces como sea necesario hasta terminar el ciclo de instrucción (varía de una microarquitectura a otra), teniendo cuidado de que en el último ciclo se incluyan los valores esperados de la simulación. Los valores esperados se pueden colocar en las columnas de alguno de los puertos de salida, o en las columnas de las señales o registros internos declarados.

```
in       clk                     B;
in       rst                     B;
in       int0                    B;
in       int1                    B;
in       t0                      B;
in       t1                      B;
out      txd                     B;
in       rxd                     B;
in       p0          (7 downto 0) X;
in       p1          (7 downto 0) X;
out      p2          (7 downto 0) X;
out      p3          (7 downto 0) X;
in       vdd                     B;
in       vss                     B;

signal   pch         (7 downto 0) X;
signal   pcl         (7 downto 0) X;
register tlq31.a     (7 downto 0) X;
register tlq31.psw   (7 downto 0) B;

begin
--comentarios en vertical sobre significado de las columnas:
--            II
--          C RNN   TR               VV  P  P        P
--          L STTTT XX P   P   P  P  DS  C  C        S
--          K T0101 DD 0   1   2  3  DS  H  L    A   W
--Iniciar simulación y activar RST para inicializar sistema
<  0ps>     : 0 11111?*1 FF FF ?**?** 10 ?**?** ?** ?********;
< +1ps>     : 1 11111?*1 FF FF ?**?** 10 ?**?** ?** ?********;
--carga de programa forzando valores de la ROM
prom31.r0000(7 downto 0) <=X"34";  -- ADDC A,#data
prom31.r0001(7 downto 0) <=X"00";  -- #data
--repetir periodos de CLK hasta completar la RUTINA DE RESET
< +1ps>     : 0 01111?*1 FF FF ?**?** 10 ?**?** ?** ?********;
< +1ps>     : 1 01111?*1 FF FF ?**?** 10 ?**?** ?** ?********;
    :         :          :          :  :      :   :         :

--Inicia CICLO DE INSTRUCCION para valores 88hex + 87hex + 1
--cargar nuevos sumandos y acarreo de entrada
tlq31.a(7 downto 0)       <= X"88"; --primer sumando
prom31.r0001(7 downto 0)  <= X"87"; --segundo sumando
tlq31.psw(7)              <=  B'1'; --acarreo de entrada
--corregir dirección PC a 0000h (útil a partir del 2o ciclo)
tlq31.pcl(7 downto 0)     <= X"00"; --parte baja de dirección
--repetir periodos CLK hasta completar ciclo de instrucción
< +1ps>     : 0 01111?*1 FF FF ?**?** 10 ?**?** ?** ?********;
< +1ps>     : 1 01111?*1 FF FF ?**?** 10 ?**?** ?** ?********;
    :         :          :          :  :      :   :         :
--en el último periodo incluir valores esperados
< +1ps>     : 0 01111?*1 FF FF ?**?** 10 ?**?** ?** ?********;
< +1ps>     : 1 01111?*1 FF FF ?**?** 10 ?00?02 ?10 ?11***1*1;
--Finaliza CICLO DE INSTRUCCION

-Repetir bloque de CICLO DE INSTRUCCIÓN para otros valores
```


Fuente: elaboración propia.

Figura 2 Archivo de patrones de prueba para verificar la instrucción **ADDC A,#data**

Todas estas columnas inician con el signo "**?",** seguido de los valores esperados. Si alguna columna contiene asteriscos en lugar de valores numéricos, el simulador rellena la columna con los valores generados durante la simulación y coloca los resultados en el archivo de patrones de salida.

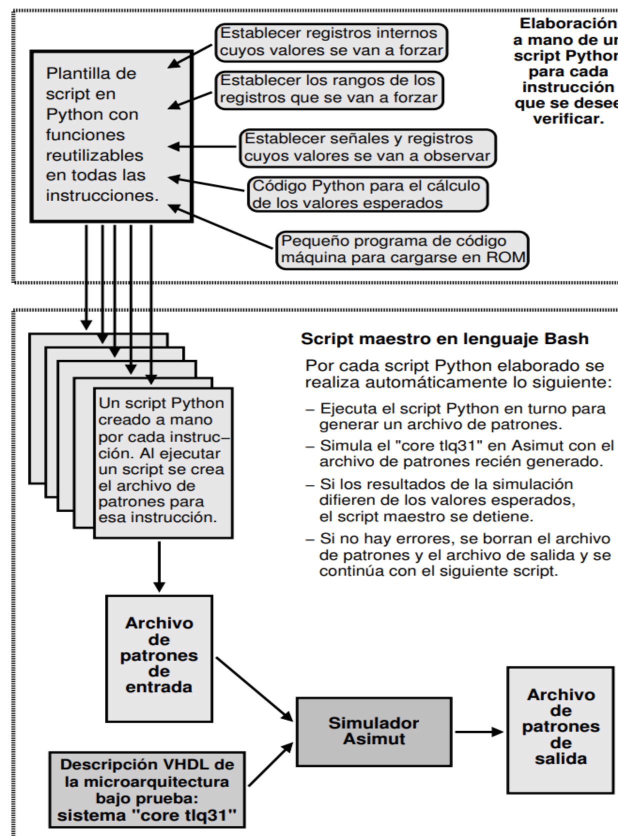
Para el caso de la instrucción verificada con el archivo de la figura 2, **ADDC**, se colocan valores esperados en las columnas del contador de programa, que deberá tener la dirección **0002**$_{\textbf{hex}}$ (bytes alto y bajo en las columnas **pch** y **pcl**, respectivamente), y el resultado esperado de la suma se coloca en la columna del

registro **A**, con el acarreo de salida en la columna del bit **PSW(7)**. También se colocan valores esperados en las columnas de los bits **PSW(6)**, **PSW(2)** y **PSW(0)** pues la instrucción ADDC los usa como bits de acarreo auxiliar, desborde y paridad, respectivamente, según se indica en las especificaciones [22].

El bloque de código CICLO DE INSTRUCCIÓN, en la figura 2, se repite para las diferentes combinaciones de valores de datos de entrada que se desee probar. Si existe algún error, i.e., discrepancia entre los valores esperados y los valores de salida, la simulación se detiene y los resultados de la simulación se pueden revisar en el archivo de salida como ayuda para depurar el error.

Estrategia de verificación para un conjunto de instrucciones

La figura 3 muestra la estrategia propuesta para la verificación de un conjunto de varias instrucciones.



Fuente: elaboración propia.

Figura 3 Marco de verificación ejecutando el banco de pruebas.

Es necesario preparar un archivo de patrones por cada instrucción que se va a probar. Sin embargo, estos archivos de patrones de entada no se implementan de manera manual, sino que cada uno se crea mediante un script en Python elaborado manualmente. Para facilitar el proceso se creó una plantilla que incluye el código en común y una pequeña librería de funciones que pueden emplearse en cualquiera de los scripts. En esa plantilla únicamente se modifican las siguientes partes de código que son específicas para cada instrucción:

- Instrucciones para forzar los valores de registros seleccionados del *core* (por ejemplo, contenido de registros específicos, dirección en el contador de programa, etc.) y/o de la memoria ROM (por ejemplo, una dirección de registro inmediato, un dato inmediato, un salto relativo, una dirección para salto absoluto, etc.).
- Rangos en que se van a variar los registros que se van a forzar. Se puede especificar un rango de variación completo (por ejemplo [0,255] para los datos de 8 bits) o puede ser un rango parcial. También puede ser cierta cantidad de valores generados aleatoriamente, o específicos contenidos en una lista.
- Señales y registros internos que se van a observar, cuyos valores de salida producidos durante la simulación se podrán comparar de manera automática con los valores esperados.
- Código para el cálculo de los valores esperados que se colocarán en el renglón de patrones al final del bloque de código CICLO DE INSTRUCCION (Figura 1). Estos valores se colocan en las posiciones correspondientes a señales o a registros internos, o bien en puertos de salida.
- El código máquina de un programa para probar la instrucción. Pueden ser unas cuantas instrucciones ya que para la verificación basta con ejecutar una sola instrucción y comparar los valores de salida con los resultados esperados.

Una vez que los scripts individuales están listos, se ejecuta un script “maestro” en lenguaje bash el cual contiene comandos que se envian directamente al emulador de terminal. Este script contiene un bucle que se repite por cada script Python elaborado, y en cada pasada realiza lo siguiente:

- Ejecuta el script Python para generar el archivo de patrones de entrada de la instrucción en turno.
- Ejecuta Asimut para simular la microarquitectura core tlq31 con el archivo de patrones recién creado.
- Si los valores de salida en la simulación difieren de los valores esperados la simulación se detiene y el usuario puede analizar el archivo de salida para depurar el error.
- Si no hay errores, se borran el archivo de patrones de entrada y el archivo de patrones de salida, y luego se continúa con el siguiente script Python.

3. Resultados

Se aplicó la metodología descrita en la sección anterior para el conjunto de 20 instrucciones que se muestra en la primera columna de la tabla 1.

Tabla 1 Tiempos de simulación por instrucción.

Instrucción	Registros internos que a forzar y sus rangos	Cantidad total de combinaciones	Tiempo de simulación total (horas)	Tiempo de simulación por combinación (ms)
ADD A,direct	A=[0,255] direct=[B] B=[0,255]	65,536	2.55	140
ADDC A,direct	A=[0,255] direct=[B] B=[0,255] CY=[0,1]	131,072	5.18	142
SUB A,direct	A=[0,255] direct=[B][0,255] CY=[0,1]	131,072	5.04	138
INC A	A=[0,255]	256	0.01	94
DEC A	A=[0,255]	256	0.01	94
MUL AB	A=[0,255] B=[0,255]	65,536	7.86	432
DIV AB	A=[0,255] B=[0,255]	62,720	10.91	626
DA A	A=[0,99] B=[0,99] CY=[0,1]	20,000	2.24	402
ANL A,direct	A=[0,255] direct=[B][0,255]	65,536	2.12	116
ORL A,direct	A=[0,255] direct=[B][0,255]	65,536	2.44	134
XRL A,direct	A=[0,255] direct=[B][0,255]	65,536	2.43	133
JC rel	CY=[0,1] rel=[0,255]	512	0.01	104
JNC rel	CY=[0,1] rel=[0,255]	512	0.01	102
JZ rel	A=[0,255] rel=[0,255]	65,536	2.38	131
JNZ rel	A=[0,255] rel=[0,255]	65,536	2.42	133
CJNE A,#data,rel	A=[0,255] #data=[0,255] rel=aleatorio	65,536	2.68	147
DJNZ direct,rel	direct=[0,127][0,255] #data=[0,255] rel=aleatorio	32,768	1.44	159
SJMP rel	rel=aleatorio	256	0.01	117
AJMP addr11	addr11=[0,2047]	2,048	0.07	125
LJMP addr16	addr16=[0,65535]	65,536	4.26	234

Fuente: elaboración propia.

En la segunda columna se muestran los registros internos seleccionados para forzar sus valores junto con los rangos en que éstos varían. En particular **direct=[B][0,255]** significa que **direct** toma únicamente el valor de la dirección del registro **B** y que el contenido de este registro varía en el rango de 0 a 255 (con esto se generan un total de 256 combinaciones). Por otro lado **direct=[0,127][0,255]** indica que **direct** toma el valor de las direcciones de registro que van de 0 a 127, y que el contenido de cada uno de tales registros varía en el rango de 0 a 255 (se generan 128 x 256 = 32,768 combinaciones). Por último, **rel=aleatorio** significa que se genera un valor para **rel** de manera aleatoria dentro del rango permitido.

En la tercera columna se muestran el total de combinaciones posibles con los rangos de valores elegidos. En la cuarta columna se muestra el tiempo de simulación requerido para el total de combinaciones. En la última columna se muestra el tiempo de simulación para una sola combinación.

A partir de los resultados de los 20 tipos de instrucción probados, se observa que el tiempo de simulación promedio por instrucción para una sola combinación es de 185 ms. Las instrucciones **MUL AB** y **DIV AB** se llevan un tiempo mucho mayor ya que la microarquitectura utilizada realiza la multiplicación mediante 8 sumas sucesivas, y la división mediante 8 restas. La instrucción **DA A** es la única instrucción que no se verificó simulando una sola instrucción para cada combinación. En cambio, se ejecuta la secuencia de instrucciones **ADD A,#data**; **DA A**; además de una instrucción de salto al final para regresar el contador de programa a la primera instrucción. Esto se decidió así porque la instrucción **DA A** comúnmente se emplea después de una instrucción de adición.

Es importante comentar que para 65,536 combinaciones se genera un archivo de patrones que supera 1 GB de tamaño. En específico, para la instrucción **ORL A,direct** el tamaño sobrepasó los 1.5 GB. Por esta razón, es importante ir eliminando los archivos de patrones de entrada y de salida cada vez que se termina sin errores una simulación con Asimut. Las simulaciones se corrieron en un equipo con procesador AMD Ryzen 5 4600H con 3.00 GHz de frecuencia de reloj.

Las herramientas Asimut y Python se instalaron en una máquina virtual Linux/Centos7. Esta máquina virtual corre con VirtualBox sobre un sistema

operativo anfitrión Windows 10. El equipo de cómputo posee 12 procesadores lógicos con 16 GB de memoria RAM, aunque la máquina virtual sólo usa 2 GB de RAM y un único procesador (pues las herramientas no están habilitadas para explotar procesamiento paralelo).

4. Discusión

Al realizar las pruebas de la sección anterior se identificó que los archivos de patrones generados pueden superar 1 GB de tamaño con tan solo 2^{16} = 65,536 combinaciones. Esto provoca el riesgo de sobrecargar la memoria RAM y de saturar el disco duro del equipo de cómputo. El riesgo es aún mayor si la herramienta Asimut corre sobre un sistema operativo Linux dentro de una máquina virtual. Debido a esto se procedió a modificar los scripts Python para limitar el tamaño de los archivos de patrones de entrada a un tamaño máximo de 100 MB, de manera que si una instrucción originalmente requería generar un archivo de 1.5 GB, ahora se deben generar 15 archivos de 100 MB cada uno. Más aún, el script bash se modificó para que, a excepción del primero, cada uno de estos archivos no se genere sino hasta haber simulado sin errores y haber eliminado el archivo de 100 MB previo.

Ahora bien, suponiendo que el promedio de 185 ms se cumple para la ejecución de las 111 diferentes instrucciones del microcontrolador MCS-51, se recomienda preparar al menos tres bancos de pruebas con las características siguientes:

- Banco de pruebas para verificación rápida de 5.5 min de duración. Este tiempo es suficiente para simular 16 combinaciones por cada tipo de instrucción.
- Banco de pruebas para verificación media de 1.5 horas de duración. Tiempo suficiente para simular 256 combinaciones por cada tipo de instrucción.
- Banco de pruebas para verificación larga de 23.4 horas de duración. Suficiente para cubrir 4,096 simulaciones por cada tipo de instrucción (puede haber instrucciones que no requieran de tantas combinaciones).

La flexibilidad que nos proporcionan estos tres bancos de prueba de diferentes duraciones nos ayuda a no detener en exceso el proceso de diseño. Si por ejemplo se realiza una modificación a la microarquitectura, la primera prueba que se debe

realizar es la de 5 minutos. Si tras esperar ese tiempo, se identifica un error, procedemos a buscarlo y corregirlo. Si la prueba se termina sin errores, continuamos con nuestro trabajo de diseño, pero antes arrancamos la verificación media y/o la verificación larga, que nos ayudarán a identificar errores que pudieron escaparse durante la verificación rápida.

Una variación del enfoque de verificación antes descrito es el incluir, dentro de las combinaciones para cada instrucción, un conjunto de valores críticos que se deben simular de manera obligatoria. Por ejemplo, en la multiplicación pueden elegirse como combinaciones obligatorias los productos $\mathbf{FF_{hex}}$ x $\mathbf{FF_{hex}}$, $\mathbf{01_{hex}}$ x $\mathbf{FF_{hex}}$, $\mathbf{FF_{hex}}$ x $\mathbf{01_{hex}}$, $\mathbf{00_{hex}}$ x $\mathbf{FF_{hex}}$, $\mathbf{FF_{hex}}$ x $\mathbf{00_{hex}}$, $\mathbf{7F_{hex}}$ x $\mathbf{02_{hex}}$, $\mathbf{3F_{hex}}$ x $\mathbf{04_{hex}}$, $\mathbf{1F_{hex}}$ x $\mathbf{08_{hex}}$. Es decir, elegir algún conjunto de valores que bajo nuestro propio criterio lleven al hardware a una alta exigencia y/o que impliquen un mayor riesgo de falla.

Cabe mencionar que el simulador Asimut, así como los scripts en lenguajes Python y bash pueden ejecutarse mediante instrucciones desde la línea de comandos de Linux. Pero las estrategias de verificación aquí descritas pueden realizarse con algún otro simulador que trabaje en lenguaje VHDL o en Verilog siempre y cuando su ejecución también pueda realizarse desde la línea de comandos.

5. Conclusiones

Una vez elaborada la descripción hardware de una microarquitectura de procesador es preciso realizar la verificación de la correcta ejecución de todas las instrucciones ante una gran cantidad de condiciones diferentes de datos y registros de entrada. Esto tiene la finalidad de detectar y depurar errores involuntarios introducidos durante la etapa de diseño. Para ello, en este trabajo proponemos emplear el simulador lógico Asimut, el cual es de uso libre y trabaja con descripciones en VHDL. Se explican a detalle las diferentes partes de los archivos de patrones de prueba requeridos por este simulador, y proponemos el empleo de un script Python por cada instrucción, para su elaboración automática. Debido a que los archivos de patrones pueden fácilmente superar 1 GB de tamaño, proponemos partirlos en archivos de hasta 100 MB y generarlos de uno en uno, teniendo cuidado de no generar un nuevo archivo de patrones hasta haber logrado simular sin errores

y haber eliminado el anterior. Con esto se disminuye el riesgo de saturar la memoria RAM y el disco duro de nuestro equipo de cómputo.

Para la propuesta de verificación aquí presentada se hicieron pruebas con una microarquitectura del microcontrolador 8031 (previamente elaborada por uno de los autores de este trabajo) resultando en un tiempo de simulación promedio de 185 ms por instrucción. Ante ello sugerimos la preparación de tres bancos de pruebas: uno que se ejecute en aproximadamente 5 minutos donde cada instrucción (de los 111 tipos que ejecuta este microcontrolador) se pruebe ante 16 combinaciones de datos diferentes; otra prueba de 1.5 horas donde cada instrucción se ejecuta para 256 combinaciones diferentes; y una más larga de cerca de 24 horas donde cada instrucción se puede simular ante 4096 condiciones diferentes.

6. Bibliografía y Referencias

[1] Parai, M. K., Das, B., & Das, G. (2013). An overview of microcontroller unit: from proper selection to specific application. International Journal of Soft Computing and Engineering (IJSCE), 2(6), 228-231.

[2] Muse Semiconductor, (2023). https://www.musesemi.com/.

[3] RISC-V International, (2021). RISC-V Exchange: https://riscv.org/exchange/.

[4] OpenCores, (2023). Projects: https://opencores.org/projects.

[5] Chávez Bracamontes, R., Gurrola Navarro, M. A., Jiménez Flores, H. J., Bandala Sánchez, M., IEICE Electronics Express, Volume 13, Issue 6, (2016). VLSI architecture of a Kalman filter optimized for real-time applications: https://doi.org/10.1587/elex.13.20160043.

[6] Gurrola, M. A., Quiroga, J. V., Avelar, Á. E., Bonilla, C. A., Padilla, I. R., & Medina, A. S., Pistas Educativas, Vol. 44, Núm. 143 (2022). Diseño de Procesador Risc-V de 32-Bits de Ciclo Único: https://pistaseducativas.celaya.tecnm.mx/index.php/pistas/article/view/2835

[7] OpenPOWER Foundation. (2023). Linux on POWER Architecture Reference: https://openpowerfoundation.org/specifications/linuxonpower/.

[8] Sparc International, Inc, (2023). The SPARC Architecture Manual, Version 9: https://sparc.org/technical-documents/#V8.

[9] Synopsys, Inc., (2023). Silicon Design: https://www.synopsys.com/silicon-design.html.

[10] The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213, Editors Andrew Waterman and Krste Asanovi´c, RISC-V Foundation, December 2019: https://riscv.org/technical/specifications/.

[11] Cadence Design Systems, Inc., (2023). Digital Design and Signoff: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff.html

[12] Siemens, (2023). Siemens EDA: https://eda.sw.siemens.com/en-US/

[13] Sorbonne Université. Laboratoire d'Informatique de Paris 6 (LIP6), (2017). Alliance. A Free VLSI/CAD System: https://www-soc.lip6.fr/equipe-cian/logiciels/alliance/.

[14] Sorbonne Université. Laboratoire d'Informatique de Paris 6 (LIP6), (2020). Alliance/Coriolis VLSI CAD Tools: https://coriolis.lip6.fr/.

[15] The-OpenROAD-Project/OpenLane, (2020). OpenLane: https://github.com/The-OpenROAD-Project/OpenLane.

[16] Open Circuit Design, (2019). Qflow 1.3 An Open-Source Digital Synthesis Flow: http://opencircuitdesign.com/qflow/index.html.

[17] Muse Semiconductor, (2023). TSMC MPW SHARED TAPEOUTS: https://www.musesemi.com/shared-block-tapeout-pricing.

[18] Europractice IC Service. GENERAL MPW & MINI@SIC SCHEDULES - 2023: https://europractice-ic.com/schedules-prices-2023/.

[19] Terasic Inc. DE10-Lite Board: https://www.terasic.com.tw/cgi-bin/page /archive.pl?No=1021.

[20] Ghoshal, S., (2010). 8051 Microcontroller. Internals, instructions, programming, and interfacing. Pearson.

[21] Singh, J., Raghavendra, S., & Kumar, S., (2020). A TextBook On Embedded System Design for Engineering Students. Nitya Publications.

[22] Intel Corporation, (1994). MCS-51 Microcontroller Family User’s Manual.