Lab05 - ARCH - 2019-II

{TA:ariana.villegas, Prof: jgonzalez} @utec.edu.pe

25 September 2019

1 Indicaciones

1. Fecha de **entrega**: `Miércoles 02 de Octubre del 2019 a las 17H en Canvas`

2 Objetivos

1. Escribir un Verilog testbench para verificar que el ALU realizado en el Lab04 funciona correctamente.
2. Les proveemos un ALU code que contiene bugs. Usando el mismo testbench del item anterior, van a simular el ALU con bugs para solucionar sus problemas.

3 Ejercicios

Hemos visto que es imposible de verificar la funcionalidad del ALU diseñado en el Lab04 usando observación directa como hicimos en ejercicios anteriores. Lo que necesitamos es un método que haga un chequeo automático para determinar si el circuito está funcionando bien. Este método se llama **functional verification.**

La idea principal de una verificación basada en software, no es tan diferente a lo que hemos hecho para verificar nuestros circuitos (por ejemplo, manualmente observa los outputs para diferentes inputs). En este laboratorio, haremos *funciontal verification* de forma automática. Implementaremos un módulo Verilog, específicamente un *testbench*, que permita: 1) usar inputs arbitrarios en el circuito que estamos probando (unit-under-test, UUT), y 2) chequee *correctness* del output. Los inputs a probar pueden ser generados dentro del módulo testbench usando *behavioral modeling* en Verilog o cargados de un archivo como test-vectors, lo cual representa un set de input data y los resutlados esperados. Si el output del UUT *matches* la salida esperada para todos los casos (para todas las posibles entradas), podemos decir que la implementación es correcta.

Al hacer *automatic functional verification*, podemos tener más *test cases* para chequear en una intervalo corto de tiempo comparado al esfuerzo manual.

Esto es especialmente importante cuando el espacio de inputs posibles es muy grande.

Hay ventajas adicionales en hacer *functional verification* en software. Por ejemplo, en un FPGA, nosostros solo podemos observar las salidas del top module. A diferencia de cuando usamos una herramienta de software que *simula nuestro circuito*, podemos tener acceso a todos los módulos internos. Esto permite rastrear el bug de forma fácil reduciendo el tiempo de pruebas de forma significativa.

En este ejercicio implementaremos un testbench en Verilog y simularemos nuestro circuito ALU del Lab04. Si hubiera algún error, podemos encontrar que está mal y corregirlo.

3.1 Creando un testbench para ALU

Usaremos nuestro ALU diseñado en el Lab04. Para poder realizar este ejercicio deberá estar finalizado antes de hacer el testbench.

Verificar que el archivo Lab05.tar contenga:

1. `ALU_test.v`
2. `bad_ALU.v`
3. `testvectors_hex.txt`
4. `Lab05.pdf`

Antes de comenzar debemos preparar un set de inputs de los cuales sepamos el output esperado. En proyectos grandes, podemos usar un *golden-model* para encontrar el resultado esperado dada un entrada específica. Un *golden-model* puede ser cualquier implementación que sepamos es correcta. Por ejemplo, puede ser otra implementación en Verilog previamente probada del circuito, o un programa escrito en un lenguaje alto nivel (Java, C++, Matlab, Python, etc.)

Podemos usar una forma sistemática de generar diferentes inputs de forma que verifiquemos el circuito entero.

En este ejercicio, les damos un set de inputs para el ALU diseñado en el Lab04. Determinar el resultado correcto para cada entrada, y usando un editor de texto, escribirlos en el archivo `testectors_hex.txt`. Este archivo contiene inputs para el ALU y su trabajo es llenar con el valor esperado de resultado en notación hexadecimal.

Notar que nuestro ALU tiene un output adicional de single-bit, llamado 'zero'. El valor esperado de este ouput puede ser facilmente determinado basdado en el resultado. Por lo tanto, podemos setearlo directamente en el test-

bench. Si el valor esperado del resultado es todos cero, el valor esperado de la señal 'zero' debe ser uno.

Adicionar el archivo `ALU_test.v` en su ambiente de trabajo (folder de sus archivos Verilog compilados con Icarus, o a su proyecto de Modelsim). Deben realizar las siguientes modificaciones al archivo, también indicadas en los comentarios del archivo:

1. Declarar un array que sea lo suficientemente grande para retener todos nuestros casos de test y que pueda almacenar los inputs *aluop, a, b* y el valor esperado de la salida *exp_result*. Por ejemplo, declarar un array de 4-elementos de 5-bit regs, se necesita escribir:

   ```
   reg[4:0] example_array [0:];
   ```

2. Adicionar un statment que lea el contenido del archivo `testvectors_hex.txt` en el array declarado antes. Hint: pueden usar la función `readmemh` (verificar la documentación online). Cuando lean el archivo, pueden espeficiar el path total del archivo o adicionar el archivo en su ambiente de trabajo de Quartus o Modelsim. Notar que la última línea del archivo es "X", la cual especifica el final del archivo para el testbench.

3. Generar el valor de `exp_zero` a partir de `exp_result`. Un statement de un assing simple será suficiente.

4. En el testbench, instanciar el ALU del Lab04. Asegurarse de conectar correctamente las señales al módulo instanciado.

3.2 Simular el ALU

Simular el ALU usando el testbench cread y obtener waveforms como resultado de la simulación.

3.3 Debugging de problemas

Usar el testbench ayuda a encontrar errores a través de simulación. Ahora su trabajo es realizar una simulación que les permita encontrar y corregir bugs en módulo Verilog. El archivo `bad_ALU.v`, se comporta de una forma similar al ALU que han desarrolado en Lab04, sin embargo contiene algunos errores intencionales que resultan en comportamiento incorrecto en algunos casos.

Como el nombre indica este código es 'bad'. No tomarlo como un ejemplo de como escribir en Verilog.

Modificar el testbench para instanciar `bag_ALU` en vez de su ALU. Simular el circuito y revisar los casos que resultan en error.

Deben encontrar por lo menos 7 errores (en 12 test-vectors). La salida de consola muestra todos los casos que resultan en error. Tambien deben revisar el waveform. Recordar que pueden cambiar el radix del waveform en GTKWave y Modelsim para facilitar su análisis.

Encontrar y resolver los errores. El código está muy próximo a funcionar por lo que no requiere un re-write extremo, solo unas pequeñas correcciones.

4 Indicaciones

1. Implementar los archivos indicados en Verilog.
2. Generar ***waveforms*** en archivos VCD para cada ejercicio según sea necesario.
3. Escribir un informe explicando. Incluir el código en el informe explicando los módulos (en Latex pueden usar lstlisting).
4. Subir en un .tar **solamente** archivos requeridos (verilog módulos y testbench, otros .txt, etc.), VCD files, Makefile e informe final en pdf.