Verificación de circuitos digitales con software libre

Eduardo Vázquez Díaz
lalohao@gmail.com

Contenido

***Resumen*—Se creó una maquina virtual con *Ubuntu Desktop 16.04.1* LTS en un contenedor utilizando el software de virtualizacion `Qemu`, donde posteriormente se instaló `verilator` y `gtkwave`; a partir de este sistema se exponen algunas técnicas para simular circuitos con verilog/C++, además de visualizar las ondas generadas de manera gráfica.**

I. Introducción

La importancia de probar los circuitos antes de ser llevados al silicio puede representar millones de dolares, sin contar el tiempo invertido en el diseño, y el que se necesitará volver a invertir para arreglarlo.

En 1990 el lenguaje de descripción de hardware mas usado era VHDL a pesar de que solo tenia constructores básicos para probar (TestBench) los circuitos. Los diseños empezaban a crecer y nuevo software comercial se creaba para compensar la falta de funciones. Algunas empresas invertían horas de trabajo para crear su propio sistema y no pagar los miles de dolares en licencias, una de ellas llevó a la creación de Accelera que fue la base de SystemVerilog [4].

De la misma manera surgió `Verilator`, un simulador potente de Verilog HDL que además es software libre, este compila el código y lo optimiza para ser simulado rápidamente [1], en algunos casos es incluso mas veloz que los simuladores comerciales [2].

I-A. Virtualización

La maquina virtual permite encapsular el sistema de verificación de circuitos en un contenedor que no será afectado (y que no afectará) la maquina utilizada, esto elimina errores que podrian ser causados al tener instalado software que utilice configuraciones con variables globales (PATHS) como ocurre con `HSPICE` y `Questa SIM` por ejemplo.

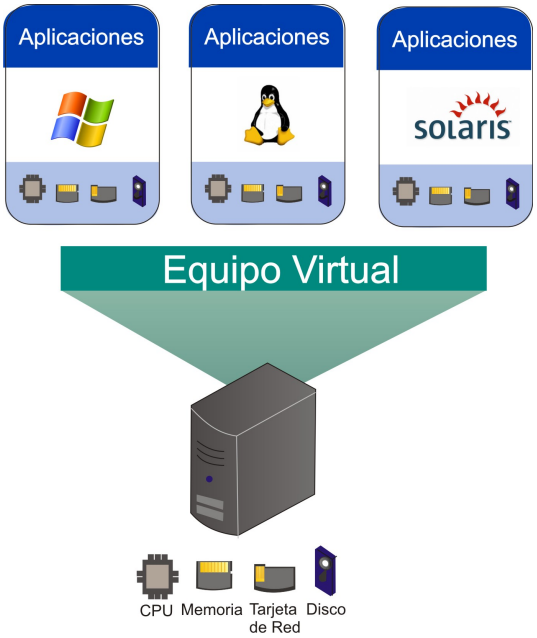

Figura 1. Las maquinas virtuales pueden o no conectarse entre ellas o hacia la red externa y pueden ser de diferentes arquitecturas y sistemas operativos independientemente del sistema anfitrion.

Se le dice anfitrión a la maquina donde se encuentran los contenedores virtuales, en este caso la anfitriona usa *Arch Linux*, pero esto no afecta a los contenedores ya que estan aislados, por lo que esto se puede aplicar de igual manera en Windows o Mac.

II. Desarrollo

Para instalar la maquina virtual se utiliza la linea de comandos, creando el disco duro virtual llamado **verif**, con un tamaño de 10Gb, posteriormente se carga el archivo `ISO` de Ubuntu en el disco duro creado, con 2G de memoria RAM y se procede a instalar el sistema operativo de forma normal.

```
qemu-img create -f raw verif 10G
qemu-system-x86_64 -cdrom ubuntu.iso \
                   --boot order=d -drive \
                   file=verif,format=raw \
                   -m 2G
```

Una vez instalado el sistema operativo se puede iniciar de la siguiente forma:

```
qemu-system-x86_64 --boot order=d \
   -drive file=verif,format=raw -m 2G
```

Posteriormente en la maquina virtual (Ubuntu) se instala el software necesario para simular [3], al igual que el editor de texto de su preferencia para modificar los archivos :

```
sudo apt-get install git make \
     autoconf g++ flex bison \
     verilator gtkwave
```

II-A. *Ejemplo 1: Compuerta AND*

Para entender el proceso de simulación es necesario entender la estructura jerárquica del espacio de trabajo.

Clona el siguiente repositorio:

```
git clone \
github.com/LaloHao/verilator_test
```

Dentro se encuentra una carpeta donde se simula la compuerta `and`, en ella se encuentran los siguientes archivos:

```
top->trace(tfp, 99);
tfp->open("cand.vcd");
top->a = 0;
top->b = 0;
```

El archivo `README` se puede ignorar; los archivos donde se describe el comportamiento y el testbench son **cand.v**, **cand-tb.cpp** respectivamente, después se encuentra el archivo **cand.vcd** que es utilizado por el programa `gtkwave` para visualizar las ondas al simularse, por ultimo el archivo **Makefile** donde contienen todos los comandos necesarios para ejecutar y abrir el visor de ondas.

Dentro de la carpeta *and* al ejecutar el comando `make` se simulará, ejecutará, y abrirá el visor de ondas automáticamente.

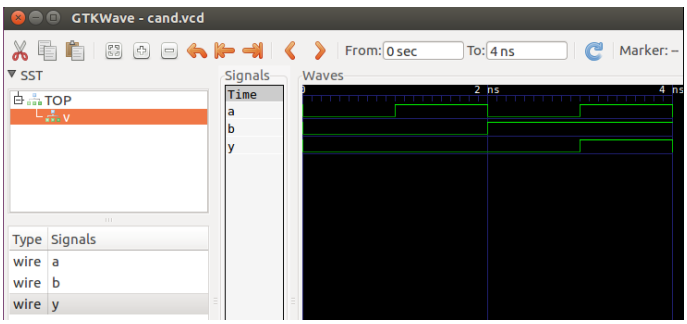

Figura 2. Visor de ondas mostrando la simulacion de la compuerta `and`.

El archivo con extensión **.v** describe el comportamiento del circuito como ya se mencionó, estas son sentencias Verilog pertinentes al lenguaje por lo que se omite la explicación de su contenido.

II-A1. Testbench: El archivo de testbench (extensión **.cpp**) contiene las instrucciones necesarias para simular.

Al ejecutarse el comando `make` Verilator genera los archivos **Vcand.cpp** y **Vcand.h** dentro de una carpeta llamada obj_{dir} donde se contienen funciones que son llamadas para controlar el flujo de señales y agregarlas al archivo **.vcd** donde se visualizaran, por ello es necesario incluirlo para referenciarlas desde el testbench al igual que los *headers* de Verilator:

```
#include "Vcand.h"
#include "verilated.h"
#include "verilated_vcd_c.h"
```

Dentro del testbench se crea un pulso de reloj que se utiliza como referencia para controlar el tiempo que correrá la simulación.

```
int clk;
```

Las siguientes lineas son para Verilator exceptuando las que contienen referencias a **Vcand** y **VcdC**; **Vcand** instancia el objeto **cand.v** el cual se incluyó anteriormente, **Vcdc** es la instancia del objeto donde se guardarán las ondas para visualizacion.

```
Verilated::commandArgs(argc, argv);
Vcand *top = new Vcand;
Verilated::traceEverOn(true);
VerilatedVcdC *tfp = new VerilatedVcdC;
```

El objeto llamado **top** es el archivo **cand.v**, desde ahí se pueden inicializar y modificar las señales, de igual el objeto **tfp** define en que archivo se guardarán las ondas, se enlazan por medio de `trace`:

```
while(i <= 15)
{
    gtkw->dump(clk);
    if(j%2)
        i++;
    ram->datain = rand() % 16;
    ram->addr = i;
    ram->clk = !ram->clk;
    ram->eval();
    clk++;
    j++;
}
```

Se usa un ciclo y envían todas las señales posibles usando `dump` para volcar las señales al archivo:

```
for(clk = 0; clk <= 4; clk++)
{
```

```
    tfp->dump(clk);
    top->a = !top->a;
    top->b = (clk >= 1);
    top->eval();
}
```

Finalmente se dan las instrucciones pertinentes a Verilator para terminar la simulación:

```
if(Verilated::gotFinish())
    exit(0);
tfp->close();
exit(0);
```

Los nombres descriptivos de variables facilitan el reconocimiento de cada objeto en proyectos mas grandes, como se verá en el siguiente ejemplo.

II-B. Ejemplo 2: Memoria RAM

Clona el repositorio:

```
git clone \
github.com/LaloHao/8bit-cpu
```

En el se encuentran un diseño de una memoria RAM y un CPU (Ejemplo 3: CPU).

```
cpu.v
cpu_tb.v
ram.v
ram_tb.v
```

II-B1. Testbench: El archivo **ram$_{tb.v}$** contiene el testbench, algunos nombres de variables se han cambiado para dar mas claridad, por ejemplo:

```
VerilatedVcdC * gtkw = new VerilatedVcdC;
Vram *ram = new Vram;
ram->trace(gtkw, 99);
gtkw->open("ram.vcd");
```

Recordando que al estar usando un lenguaje como C++ se tiene todos sus operadores:

```
#define WRITE 1
#define READ 0
```

Permitiendo definir la operación que realiza la RAM:

```
ram->rw = WRITE;
ram->rw = READ;
```

1. Escritura

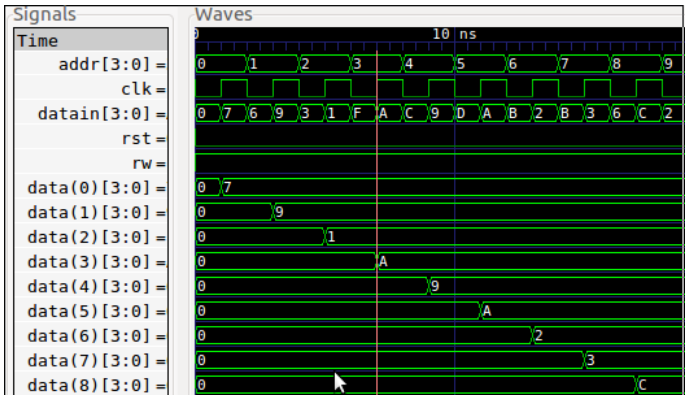

Figura 3. Escritura de RAM con datos aleatorios, es importante destacar que aunque `datain` cambie en cada pulso de reloj, la RAM solo guarda su valor en `risingedge` (cuando esta subiendo).

Se llenan todas las direcciones `addr` de memoria colocando en su entrada `datain` un valor aleatorio modulo 16 para limitarlo entre 0 y 15.

```
while(i <= 15)
{
    gtkw->dump(clk);
    if(j%2)
        i++;
    ram->datain = rand() % 16;
    ram->addr = i;
    ram->clk = !ram->clk;
    ram->eval();
    clk++;
    j++;
}
```

2. Lectura

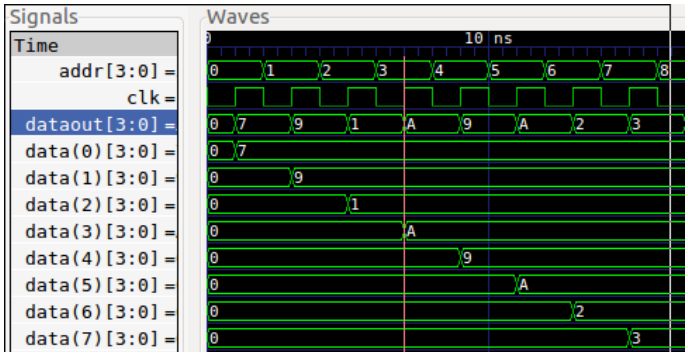

Figura 4. Lectura de RAM.

3. Reset

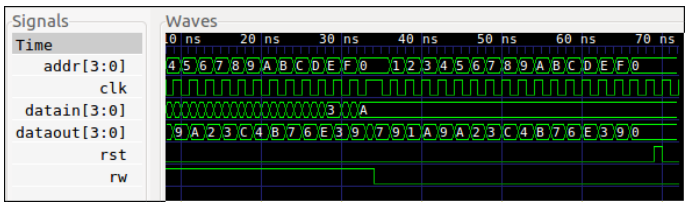

Figura 5. Verificacion de reset en la memoria RAM

II-C. ***TODO*** *Ejemplo 3: CPU*

II-D. ***TODO*** *Ejemplo 4: Receptor UART*

REFERENCIAS

[1] https://www.veripool.org/wiki/verilator Last accessed Sat Jan 28 12:01:25 2017.

[2] https://www.veripool.org/wiki/veripool/Verilog_Simulator_ Benchmarks Last accessed Wed Feb 1 20:41:50 2017.

[3] https://www.veripool.org/projects/verilator/wiki/Installing Last accessed Sat Jan 28 12:25:07 2017.

[4] Chris Spear. *SystemVerilog for verification a guide to learning the testbench language features*. Springer, 2008.

APÉNDICE A
EMACS

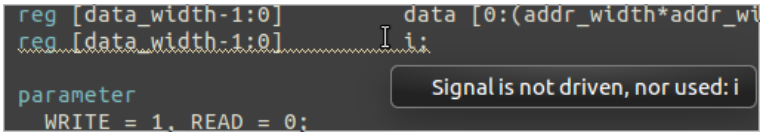

Figura 6. La maquina virtual contiene un editor de texto llamado `Emacs`, este interactua directamente con `Verilator` mostrando mensajes de error.

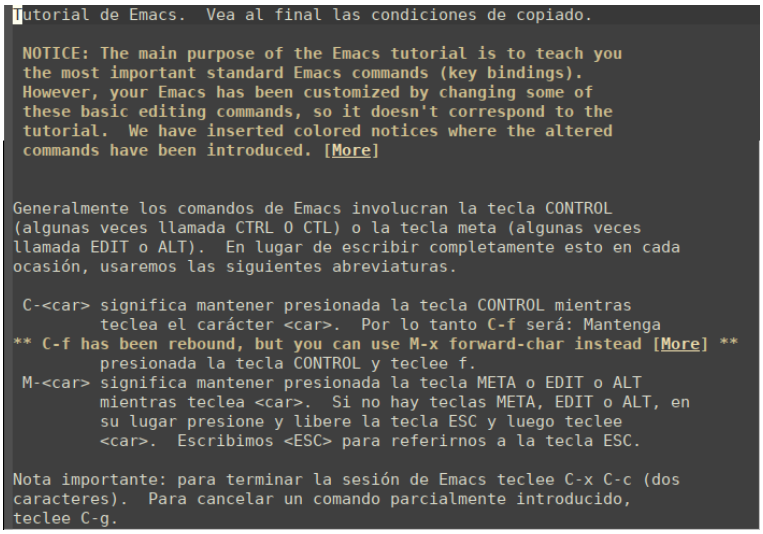

Figura 7. Se puede acceder al tutorial de emacs dentro de el presionando `Ctrl+h` y despues `t`.