Universidad de Costa Rica | Facultad de Ingeniería | Escuela de Ingeniería Eléctrica

Bitácora de Laboratorio 1

Roberto Sánchez Cárdenas — B77059
Gabriel Jiménez Amador — B73895

San José, 26 de abril de 2021
IE0424 — Laboratorio de Circuitos Digitales

Resumen

Este laboratorio presenta una serie de simulaciones por comportamiento de descripciones de hardware escritas en Verilog, en ellas se utilizó código escrito en el lenguaje C para analizar la ejecución de instrucciones en un microprocesador.

Índice

1. Introducción

Este proyecto tiene como fin obtener el conocimiento necesario para el correcto desarrollo de las actividades de laboratorio que serán propuestas más adelante en el curso de Laboratorio de Circuitos Digitales. En este caso se desarrolla una práctica introductoria sobre la síntesis de código HDL en un FPGA.

El estudiante debe luego de crear varios códigos, simularlos y justificar los resultados obtenidos de acuerdo con los conceptos de procesadores RISC.

2. Objetivos

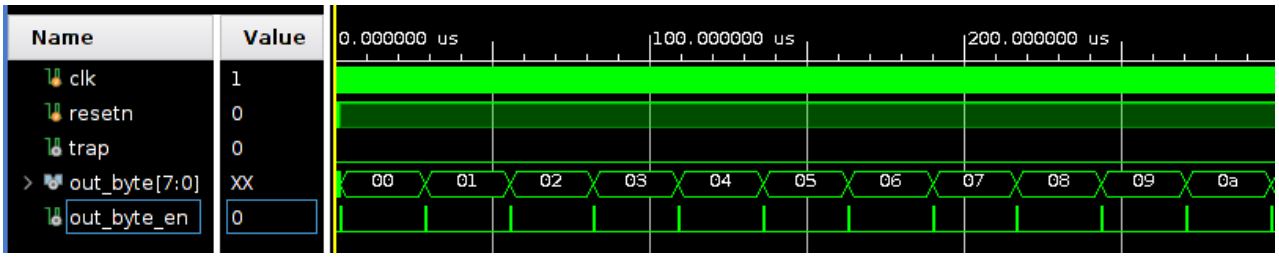
- Familiarizar al estudiante con el ambiente integrado de desarrollo Vivado de Xilinx.
- Presentar al estudiante una implementación en HDL de un microprocesador de la arquitectura RISC-V.
- Presentar al estudiante un ejemplo sencillo en código C para ser ejecutado en el microprocesador.

3. Resultados

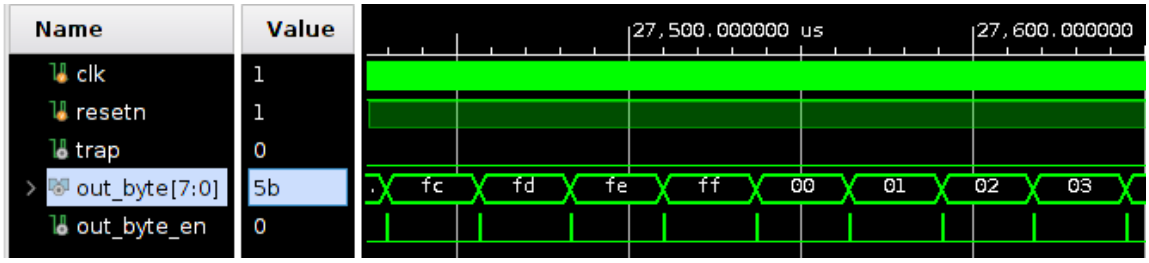
3.1. Ejercicio 1

- **¿Qué presenta la señal `out_byte`?**

  La señal `out_byte` corresponde a un número de 8 bits que funciona como contador de 0 a 255, cuando llega a 255 reinicia a 0.



(a) Simulación de señal `out_byte` entre 0 y 0.3 ms



(b) Simulación de señal `out_byte` entre 27.4 y 27.7 ms

Figura 1: Simulación de señal `out_byte` en diferentes periodos de tiempo

- **¿Cada cuántos ciclos de reloj cambia?**

  Cambia cada 2689 ciclos de reloj, donde cada ciclo de reloj tiene un periodo de 10 ns.

- **¿Cuál es el valor máximo que llega a tener la señal?**

  El valor máximo es de 255 o `0xFF`. Este es su valor máximo puesto que el contador `out_byte` es de 8 bits y sólo puede representar números sin signo entre $[0, 2^8 - 1] = [0, 255]$

- **Analice el código fuente de firmware y lo que se escribe en la dirección `0x10000000` y busque similitudes con lo que ve en `out_byte`.**

  Viendo el código fuente (`firmware.c`), en la dirección `0x10000000` se escribe la variable `number_to_display`, que es un contador que incrementa en 1 su valor cada cierto periodo de tiempo definido, así como lo hace `out_byte`.

  `out_byte` adquiere sus valores al tomar el valor de lo escrito en la posición de memoria `0x10000000` como se observa en la simulación de la Figura 2 en donde un ciclo de reloj después de que `mem_la_addr` cambia a `0x10000000` y se activa `mem_la_write`, se activa `out_byte_en` y adquiere el valor de `mem_la_wdata`.

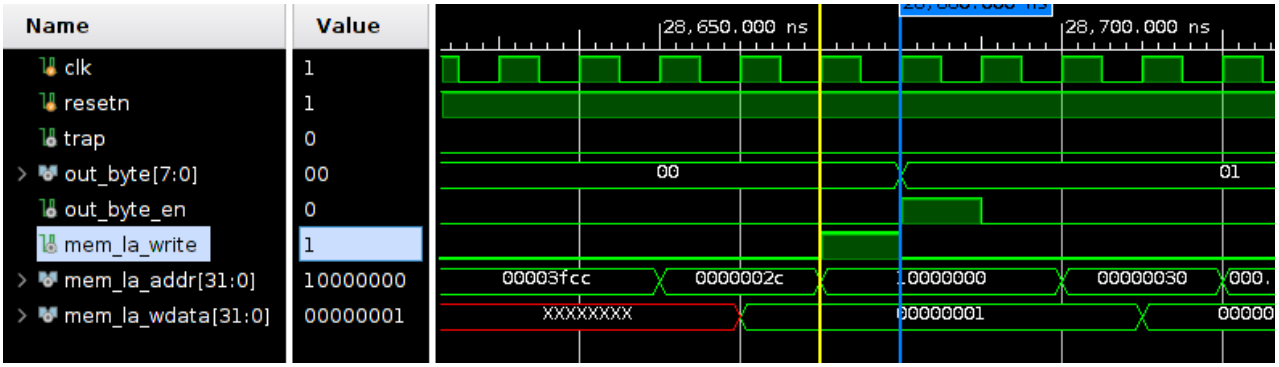


Figura 2: Simulación de la toma de datos de memoria hacia `out_byte`

3.2. Ejercicio 2

- **¿Qué hace terminar la simulación?**

  La señal `trap`; cuando esta se levanta pasa un ciclo de reloj y la simulación termina.

- **Relacione de acuerdo con lo que ve en la señal `out_byte`, la parte de firmware luego de la cual se termina la simulación.**

  La señal `out_byte` se mantiene en 0 siempre hasta al final de la simulación. Como ya no existe el *loop* infinito que incrementaba la señal que escribía a la posición de memoria `0x10000000`, solo puede escribir el 0 inicial.

- **Dé una posible razón por la cual se pueda explicar que la cause de la finalización de la simulación se haya dado.**

  Como se ha eliminado el ciclo del firmware, este finaliza con un código de máquina de:

  ```
  ac:    0000                    unimp
  ```

  El código `unimp` según el manual de RISC-V no es una instrucción válida e incita un trap al sistema o core implementado, como el PicoRV32 de Claire Wolf. La señal `trap` llega hasta el testbench que lo maneja como una señal para terminar la simulación.

3.3. Ejercicio 3

Se adjunta el código creado para este ejercicio a continuación. Este escribe los números pares de la serie de Fibonacci hasta 10946 en la dirección de memoria 0x10000000.

```
#include <stdint.h>

#define LED_REGISTERS_MEMORY_ADD 0x10000000
#define LOOP_WAIT_LIMIT 100

static void putuint(uint32_t i){
    *((volatile uint32_t *)LED_REGISTERS_MEMORY_ADD) = i;
}

static void delay(){
    uint32_t counter = 0;
    while (counter < LOOP_WAIT_LIMIT){
        counter++;
    }
}

void main(){

    while (1){
        uint32_t z = 0;
        uint32_t x = 0;
        uint32_t y = 1;
        putuint(z);
        delay();

        while (z < 10946){
            z = x + y;
            x = y;
            y = z;

            if (z % 2 == 0){
                putuint(z);
                delay();
            }
        }
    }
}
```

- **Compare el código obtenido con el que escribió en el anteproyecto y justifique las diferencias con base en los requerimientos de este firmware.**

  Realizando el objdump del código en C se obtiene el siguiente código ensamblador:

```
Disassembly of section .memory:

00000000 <putuint-0x10>:
   0:   00004137                lui     sp,0x4
   4:   00010113                mv      sp,sp
   8:   070000ef                jal     ra,78 <main>
   c:   00100073                ebreak

00000010 <putuint>:
  10:   fe010113                addi    sp,sp,-32 # 3fe0 <end+0x3ed8>
  14:   00812e23                sw      s0,28(sp)
  18:   02010413                addi    s0,sp,32
  1c:   fea42623                sw      a0,-20(s0)
  20:   100007b7                lui     a5,0x10000
  24:   fec42703                lw      a4,-20(s0)
  28:   00e7a023                sw      a4,0(a5) # 10000000 <end+0xffffef8>
  2c:   00000013                nop
  30:   01c12403                lw      s0,28(sp)
  34:   02010113                addi    sp,sp,32
  38:   00008067                ret

0000003c <delay>:
  3c:   fe010113                addi    sp,sp,-32
  40:   00812e23                sw      s0,28(sp)
  44:   02010413                addi    s0,sp,32
  48:   fe042623                sw      zero,-20(s0)
  4c:   0100006f                j       5c <delay+0x20>
  50:   fec42783                lw      a5,-20(s0)
  54:   00178793                addi    a5,a5,1
  58:   fef42623                sw      a5,-20(s0)
  5c:   fec42703                lw      a4,-20(s0)
  60:   06300793                li      a5,99
  64:   fee7f6e3                bgeu    a5,a4,50 <delay+0x14>
  68:   00000013                nop
  6c:   01c12403                lw      s0,28(sp)
  70:   02010113                addi    sp,sp,32
  74:   00008067                ret
```

```
00000078 <main>:
  78:   fe010113                addi    sp,sp,-32
  7c:   00112e23                sw      ra,28(sp)
  80:   00812c23                sw      s0,24(sp)
  84:   02010413                addi    s0,sp,32
  88:   fe042623                sw      zero,-20(s0)
  8c:   fe042423                sw      zero,-24(s0)
  90:   00100793                li      a5,1
  94:   fef42223                sw      a5,-28(s0)
  98:   fec42503                lw      a0,-20(s0)
  9c:   f75ff0ef                jal     ra,10 <putuint>
  a0:   f9dff0ef                jal     ra,3c <delay>
  a4:   03c0006f                j       e0 <main+0x68>
  a8:   fe842703                lw      a4,-24(s0)
  ac:   fe442783                lw      a5,-28(s0)
  b0:   00f707b3                add     a5,a4,a5
  b4:   fef42623                sw      a5,-20(s0)
  b8:   fe442783                lw      a5,-28(s0)
  bc:   fef42423                sw      a5,-24(s0)
  c0:   fec42783                lw      a5,-20(s0)
  c4:   fef42223                sw      a5,-28(s0)
  c8:   fec42783                lw      a5,-20(s0)
  cc:   0017f793                andi    a5,a5,1
  d0:   00079863                bnez    a5,e0 <main+0x68>
  d4:   fec42503                lw      a0,-20(s0)
  d8:   f39ff0ef                jal     ra,10 <putuint>
  dc:   f61ff0ef                jal     ra,3c <delay>
  e0:   fec42703                lw      a4,-20(s0)
  e4:   000037b7                lui     a5,0x3
  e8:   ac178793                addi    a5,a5,-1343 # 2ac1 <end+0x29b9>
  ec:   fae7fee3                bgeu    a5,a4,a8 <main+0x30>
  f0:   f99ff06f                j       88 <main+0x10>
  f4:   3a434347                fmsub.d ft6,ft6,ft4,ft7,rmm
  f8:   2820                    fld     fs0,80(s0)
  fa:   29554e47                fmsub.s ft8,fa0,fs5,ft5,rmm
  fe:   3820                    fld     fs0,112(s0)
 100:   322e                    fld     ft4,232(sp)
 102:   302e                    fld     ft0,232(sp)
 104:   0000                    unimp
        ...
```

Quitando la complejidad agregada por la funciones de `putuint` y `delay`, el código se ve mucho más complejo a lo creado a mano. El compilador parece preferir emplear el stack y no registros temporales (t0-t6). Algo que está idéntico entre ambos códigos es la forma en que se cargó la posición de memoria `0x10000000`, con `lui` pero retirando unos ceros del número, cómo se ve en la línea 20.

Además, a finales del objdump, se ven instrucciones de punto flotante como `fmsub.d`, `fld`, `fmsub.s`, pero se dan luego del salto repetitivo a `main` por lo que representan instrucciones no relacionadas directamente al código de Fibonacci. Finalmente para terminar el código se llama a la instrucción inválida de `unimp` en lugar de `ecall`. Gracias a que termina de esta forma es que se puede llamar a la señal de `trap` al simularlo en Vivado.

Siguiendo, el firmware además tiene el requerimiento de estar en un *loop* infinito para no llamar a la señal `trap` como se observa en el código fuente en C (`while(1)`). Esta diferencia se observa en que el llamando a `unimp` ni siquiera se da, puesto ocurre un salto incondicional en la instrucción `0xF0` al inicio de lo que sería en C el método `main`.

3.4. Ejercicio 4

A partir del código del ejercicio anterior, en el cual se desensambló el código por el cual se obtiene la secuencia de Fibonacci, se realizó una simulación con el firmware para observar el comportamiento de las distintas señales dentro del cpu picorv32. En esta sección se explica lo que presenta cada señal indicada.

reg_pc: Esta señal lleva un contador de programa, es decir, indica una referencia a donde se encuentra la instrucción dentro del código. si se observa el código desensamblado, se puede observar que en cada línea lo que aparece es un número hexadecimal, estos son los números que muestra el reg_pc.

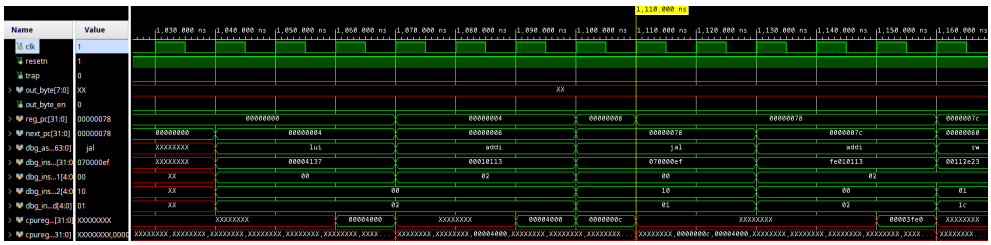


Figura 3: Simulación con todas las señales

Como se observa en la señal del reg_pc, esta va contando de 4 en 4 hasta que llega a un salto, después del contador 8 pasa 78.

next_pc: Este es un contador que va adelantado al reg_pc, es decir, indica el pc de la siguiente instrucción que se va a ejecutar.

dbg_ascii_instr: Esta señal indica el nombre de la instrucción en ascii.

dgb_insn_opcode: En esta señal se indica un código que contiene información para que cada instrucción pueda ser ejecutada de manera idónea.

dbg_insn_rs1: Registro fuente usado para la operación de la instrucción actual (rs1).

dbg_insn_rs2: Registro fuente usado para la operación de la instrucción actual (rs2).

cpuregs_wrdata: este registro es utilizado para traer valores inmediatos o almacenar temporalmente los valores a guardar.

cpuregs: Array de registros que están disponibles.

3.4.1. Momento en el que se obtiene el número 144 de la secuencia de Fibonacci

A nivel teórico, sabemos que en la serie de Fibonacci , el numero 144 se obtiene al sumar 89 con 55. Para hallar el punto en el que se obtiene este valor, buscamos en el registro cpuregs_wrdata, pues sabemos que debe aparecer en este punto pues el valor se debe guardar. En este punto se está ejecutando una instrucción add con pc_counter b0.

En el código desensamblado observamos que esta instrucción tiene la forma: **add a5,a4,a5**. Al analizar cpuregs se notó que estos valores se guardaron en las posiciones 14 y 15 del array, pues se nota que cuando se ejecuta la instrucción add, en estos se tiene 55 y 89 respectivamente, además que el 144 se guarda en la posición 15, la cual equivale a a5. Lo anterior se puede observar gráficamente en la siguiente imagen.

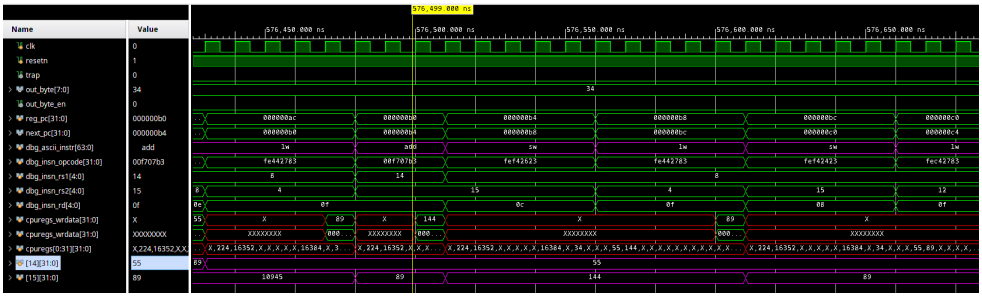


Figura 4: Momento en que se obtiene 144

3.4.2. Momento donde se escribe 144 en la dirección de memoria 0x10000000

Si se observa el código desensamblado, podemos notar en que la instrucción con pc_code 20 se carga la dirección 0x10000 en 15, además cargamos un dato en a4, y en la instrucción con op_code 24, se guarda el dato en de a5 en 14. Este es el punto en donde se guardan los datos en la dirección 0x10000000.

En la simulación se va a buscar en el cpuregs_wrdata el momento en que se carga 0x10000000.

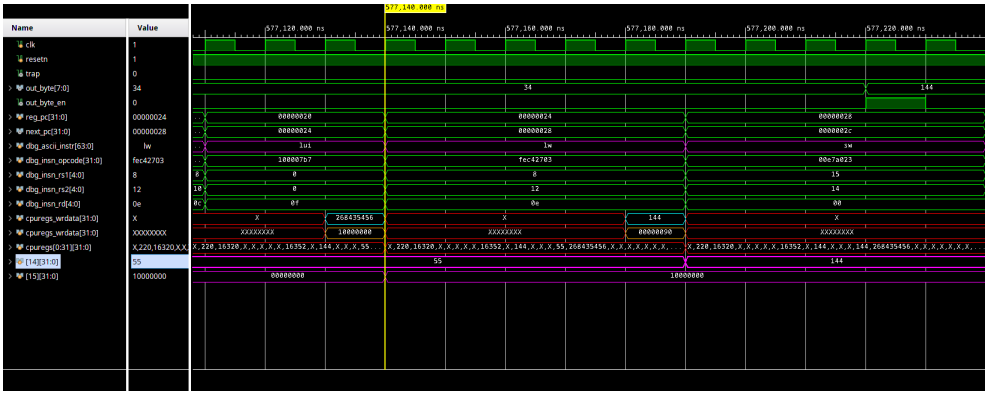


Figura 5: Momento en que se guarda el valor 144

En la figura previa observamos la ejecución de una instrucción lui que trae un 0x10000000 a cpuregs[14]. Seguidamente se ejecuta un lw que trae un 144 que se guarda en cpuregs[15]. Finalmente se observa un sw que toma cpuregs[14] y cpuregs[15] para guardar en memoria.

3.4.3. Justifique si el cpu utilizado utiliza el método de pipeline

Si se observa la figura 5 se puede evidenciar que el cpu no utiliza el método de pipelining, pues en el registro dbg_ascii_instr, las instrucciones están siendo llamadas con cada ciclo de reloj que pasa, si no que otra instrucción no se llama hasta que se termina de ejecutar la actual. En una metodología pipeline, siempre y cuando no hayan problemas de ejecución, cada ciclo de reloj llama a otra instrucción y las va ejecutando en paralelo.

Además, puesto que no tiene una estructura de pipeline, no debe evitar los riesgos de datos como read after write.

4. Repositorio

- El repositorio trabajado para el laboratorio es:

  https://github.com/ucr-ie-0424/laboratorio-1-jimenez-sanchez

- El ID del último commit es:

  363f497c77830f31a9e45932cd4f67e745a6d1ff

5. Conclusiones y recomendaciones

5.1. Conclusiones

- Se consiguió dar un primer vistazo a la herramienta de desarrollo Vivado de Xilinx mediante la síntesis e implementación de un contador básico en un lenguaje HDL como lo es Verilog.
- Se logró entender que la relación entre el *hardware* escrito en un lenguaje HDL y el *firmware* escrito en C es que el segundo dicta el comportamiento del primero.
- Se observó la concordancia entre lo escrito en el firmware y lo realizado por el hardware mediante simulaciones gráficas en Vivado.
- Se analizaron detalles propios del core PicoRV32 como el llamado a la señal `trap` según ciertas instrucciones del *firmware* o el hecho de que la propia implementación emplea pipelining.

5.2. Recomendaciones

- Cuando se escribe código en un lenguaje de alto nivel, para comprender a profundidad lo que un código realiza a nivel de registros resulta útil la funcionalidad del desensamblado, pues permite ver como se está manipulando los datos en el procesador, registros y memoria.
- Al interpretar una simulación de bajo nivel, es buena idea guiarse con el `program counter` pues permite navegar el código desensamblado. Además es importante analizar la pila de registros y analizar qué información se encuentra en cada posición.