
# Cátedra de Arquitectura de Computadoras

# Trabajo Práctico N° III
# BIP I

Carrizo, Aixa Mariel
Piñero, Tomás Santiago
10 de Diciembre de 2020

# Índice

# 1. Enunciado

El objetivo de este trabajo es implementar en FPGA, programando en Verilog, un procesador monociclo sin saltos. Este procesador se encuentra explicado detalladamente en [1] como "BIP I". Los requerimientos del trabajo son los siguientes:

- Se debe ejecutar una instrucción por ciclo de reloj;
- Se deben implementar las instrucciones de la Tabla **1**.
- Al ejecutar una instrucción `HLT`, se debe transmitir el valor del acumulador;
- Validar el desarrollo por medio de *Test Bench*.

## 1.1. Arquitectura del procesador

El BIP soporta direccionamientos directo e indirecto, con un bus de 16 bits tanto para datos (números enteros) como para instrucción.

Las instrucciones están conformadas por dos campos:

1. **Código de operación**: Los 5 bits más significativos. Indica qué instrucción se va a ejecutar;
2. **Operando**: Los 11 bits menos significativos. Indica el operando para la instrucción, que puede ser un dato inmediato (un número constante) o una dirección del dato en la memoria (una variable).

En la siguiente Tabla se muestran las instrucciones junto con su código, mnemónico y la actualización de la memoria de datos (DM) y el acumulador (ACC). El contador de programa se aumenta en 1 en cada ejecución de instrucción, con excepción de *'Halt'*.

| Instrucción | Codigo | Mnemonico | Actualización de ACC y DM |
|---|---|---|---|
| Halt | 00000 | HLT | - |
| Store Variable | 00001 | STO | DM[operando] ← ACC |
| Load Variable | 00010 | LD | ACC ← DM[operando] |
| Load Immediate | 00011 | LDI | ACC ← operando |
| Add Variable | 00100 | ADD | ACC ← ACC + DM[operando] |
| Add Immediate | 00101 | ADDI | ACC ← ACC + operando |
| Subtract Variable | 00110 | SUB | ACC ← ACC - DM[operando] |
| Subtract Immediate | 00111 | SUBI | ACC ← ACC - operando |

Tabla 1: Instrucciones del BIP I.

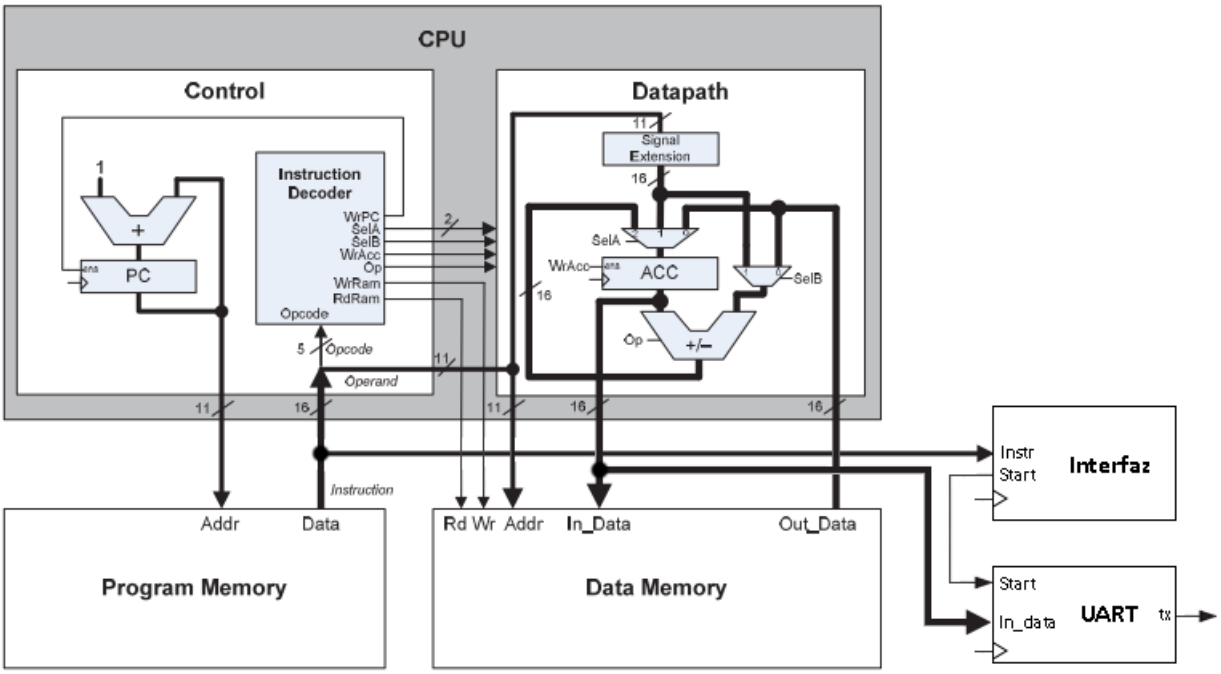


Figura 1: Esquema del proyecto.

# 2. Desarrollo

## 2.1. Diseño

Se siguió el diseño propuesto en la Figura **1**. La CPU está conformado por dos módulos:

- **Control**: toma las instrucciones de la memoria de programa, las decodifica y manda las operaciones al bloque *datapath*. Está compuesto por:
  - registro de contador de programa (PC);
  - sumador de 11 bits;
  - decodificador combinacional.
- ***Datapath***: procesa los datos bajo el comando del bloque de control. Tiene:
  - registro de ACC;
  - suma y resta de 16 bits;
  - extensor de bits;
  - dos multiplexores.

Este módulo interactúa con los módulos de memoria, uno para el programa y otro para los datos.

Para comunicar la CPU con el UART, se diseñó una interfaz que controla el contador de programa para indicar el envío del acumulador.

### 2.1.1. Bloque de control

El bloque de control trabaja en los flancos positivos del *clock*, instancia el módulo *'op_ decoder'* y modifica el contador de programa de acuerdo a la salida `o_write_pc` del mismo. Consta de una entrada (`i_instruction`) y 8 salidas:

1. `o_operand`: operando de la instrucción.
2. `o_sel_a`: selección del multiplexor A (memoria, operador o resultado de la alu)
3. `o_sel_b`: selección del multiplexor B (memoria u operador).
4. `o_write_acc`: escribir el acumulador.
5. `o_operacion`: operación a realizar (suma o resta).
6. `o_write_ram`: escribir la memoria de datos.
7. `o_read_ram`: leer la memoria de datos.
8. `o_addr`: leer la siguiente instrucción.

#### Decodificador de operación

Este módulo se encarga de tomar los 5 bits más significativos de la instrucción de entrada (`i_instruction`) y decodifica la operación a realizar junto con los datos a utilizar.

Los códigos de las operaciones son los que se muestran en la Tabla **1**.

### 2.1.2. *Datapath*

Este módulo lee/escribe la memoria de datos y realiza las operaciones según las instrucciones que recibe del bloque de control durante los flancos negativos del *clock*.

Como recibe un operando de 11 bits (`i_operando`) y el procesador trabaja con 16, lo primero que se realiza es una extensión del signo del operando ingresado.

Para facilitar el entendimiento de código se usaron parámetros locales que indican los valores que deben seleccionar los multiplexores:

- `MEMORIA`: se debe utilizar el valor desde la memoria de datos;
- `OPERANDO`: se debe utilizar el operando de la instrucción;
- `RESULTADO`: se debe utilizar el resultado de la operación realizada.

### 2.1.3. *bip_ i*

Instancia los dos módulos explicados anteriormente. Su objetivo es verificar el correcto funcionamiento del procesador (ver Sección **2.2**).

### 2.1.4. Memorias

Para los módulos de memoria se utilizó la plantilla provista por Vivado. Se trata de un bloque de memoria RAM *'No change mode'*, que es el recomendado para consumir poca potencia.

La plantilla se encuentra en:

`Templates → Synthesis Constructs → Example Modules → RAM → BlockRAM → Single Port`

La memoria de datos trabaja en el flanco positivo del *clock*, mientras que la de programa trabaja en el flaco negativo.

### 2.1.5. Interfaz

Este módulo se encarga de monitorear la instrucción leída por el procesador cada vez que se detecta un cambio en el *wire*. En caso de ser una instrucción `HLT`, le indica al transmisor UART que envíe el valor que se encuentra actualmente en el acumulador. Su código es el siguiente:

Código 1: Código fuente de "interfaz_uart".

```
1  `timescale 1ns / 1ps
2
3  module interfaz_uart
4  (
5      input  wire [15:0] i_instruccion,
6      input  wire        i_valid,
7      output wire        o_tx_start
8  );
9
10     localparam [15:0] HLT = 16'b0;
11
12     reg tx_start;
13
14     assign o_tx_start = tx_start;
15
16     always@(*)begin:check
17         tx_start = 1'b0;
18
19         if(i_valid)
20         begin
21             if(i_instruccion == HLT)
22                 tx_start = 1'b1;
23             else
24               tx_start = 1'b0;
25         end
26     end
27 endmodule
```

### 2.1.6. UART

Se utilizaron los módulos `tx_uart` y `baudrate_generator` del trabajo práctico anterior.

La única diferencia con el anterior es que esta vez el parámetro `DATA_BITS` vale 16 en vez de 8.

### 2.1.7. *top*

Es el módulo que instancia todos los módulos anteriores. El resultado de los *testbenches* realizados se muestran a continuación.

## 2.2. *Testbench*

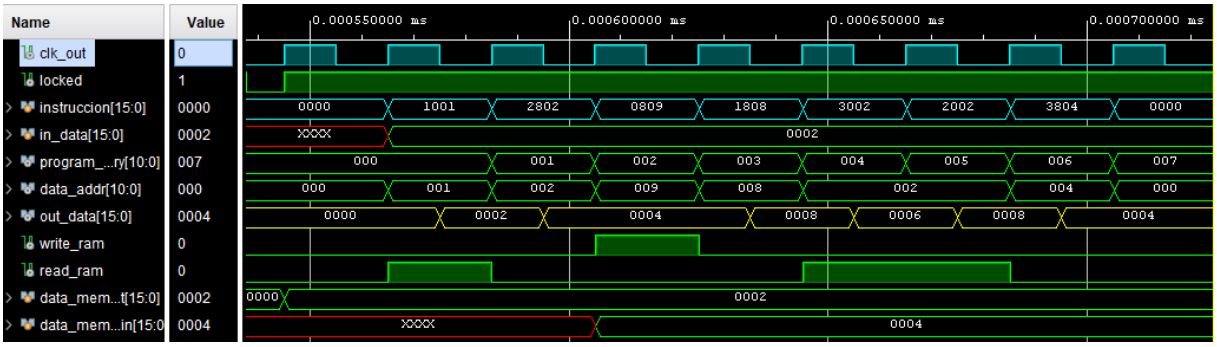


Figura 2: Resultado de *'tb_bip'*.

En la Figura **2** se muestra la ejecución del *testbench* procesador (bip_i) con estas instrucciones:

```
16'b00010_000_0000_0001; // Load variable 0x01 => ACC=DRAM[0x01]
16'b00101_000_0000_0010; // Add immediate +0x2 => ACC=DRAM[0x01]+0x02
16'b00001_000_0000_1001; // Store in 0x9 =>DRAM[0x09]=ACC
16'b00011_000_0000_1000; // Load immediate 0x08 => ACC=0x08
16'b00110_000_0000_0010; // Substract variable in 0x02 => ACC=0x08-DRAM[0
    x02]
16'b00100_000_0000_0010; // Add variable in 0x02 => ACC=0x08
16'b00111_000_0000_0100; // Substract immediate 0x04 => ACC = 0x04
16'b00000_000_0000_0000; // Halt
16'b00010_000_0000_0001; // Load variable 0x01 => ACC=DRAM[0x01]
```

Luego, la simulación con tiempo del módulo *top* dio el siguiente resultado:

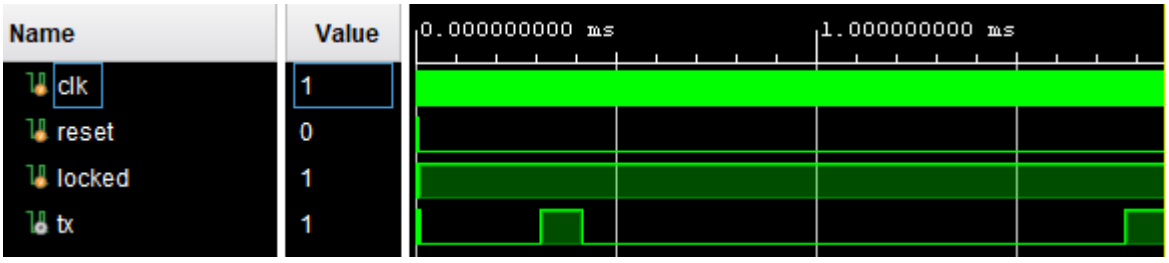


Figura 3: Resultado de *'tb_top'*.

# 3. Cálculo de frecuencia máxima

Para obtener el valor de la máxima frecuencia a la que puede trabajar el circuito se utilizó el reporte de tiempo que provee Vivado. Para obtener este reporte se utilizo la IP *clocking wizard*, con una entrada de 100 [MHz] y una salida de 50 [Mhz].

Una vez hecho esto, el reporte de tiempo realizado por Vivado muestra que el tiempo de *setup* es de `5.447 [ns]` y el tiempo de *hold* es de `0.232 [ns]`.

Sumando estos tiempos, se obtiene el tiempo del *clock*:

$$\begin{aligned} t_{clock} &= t_{setup} + t_{hold} \\ t_{clock} &= 5{,}447 + 0{,}232 \\ t_{clock} &= 5{,}679\,[ns] \end{aligned}$$

Finalmente, el periodo del *clock* es:

$$f_{clock} = 176\,[MHz] \tag{1}$$

# 4. Conclusiones

Se logró implementar un diseño modularizado del modelo propuesto, que permitió encontrar los errores con más facilidad cuando las simulaciones no mostraban los resultados esperados.

Se incorporó la utilización de herramientas de análisis de tiempo provistas por Vivado, lo que facilitó el cálculo de la frecuencia máxima de operación del circuito.

# Referencias

[1] Maicon Carlos Pereira, Paulo Viniccius Viera, André Luis Alice Raabe and Cesar Albenes Zeferino. *A Basic Processor for Teaching Digital Circuits and Systems Design with FPGA*. University of Vale do Itajai, 2012