Diseño de Microprocesadores
Práctica 1: Multiciclo

David Adrián Michel Torres
Eduardo Ethandrake Castillo Pulido

Realizar la implementación del procesador RISC-V multi-ciclo visto en clase. Este procesador debe ser capaz de ejecutar el programa que implementa el factorial de n . El valor de n se trasmitirá a través de una terminal serial, se recomienda el uso de docklight, pero puede cualquier otra terminal que sea capaz de transmitir valores hexadecimales directamente. n será recibida por el microprocesador por una UART la cual se tiene que integrar como periférico mapeado a memoria. Una vez procesado el factorial, el resultado será trasmitido por la UART hacia la terminal en la PC. Note que la UART solo puede transmitir 8 bits a la vez, por lo cual para poder transmitir los 32 bits resultantes se tiene que hacer en paquetes de 8 bits, comenzando por los 8 bits más significativos.

Entregables reporte:

- *La frecuencia de transmisión recepción debe ser 9600 baudios.*

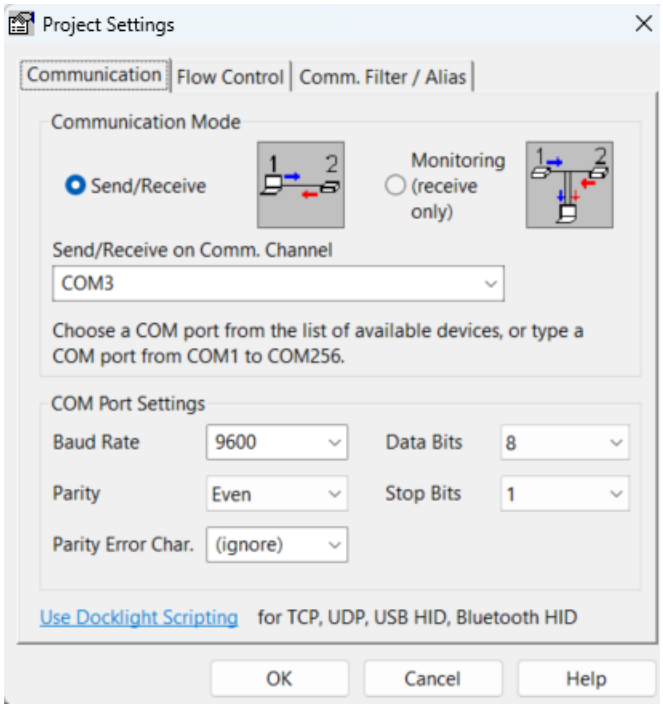


Imagen 1 – Configuración Serial Docklite

Se utilizó el programa de Docklight para esta práctica, el cual fue configurado a 9600 baudios con paridad par, 8 bits de datos y un bit de stop. Se configuró de manera send/receive para tener un UART full dúplex. Esta configuración se puede observar en la Imagen 1.

Imagen 1 – Configuración Serial Docklite

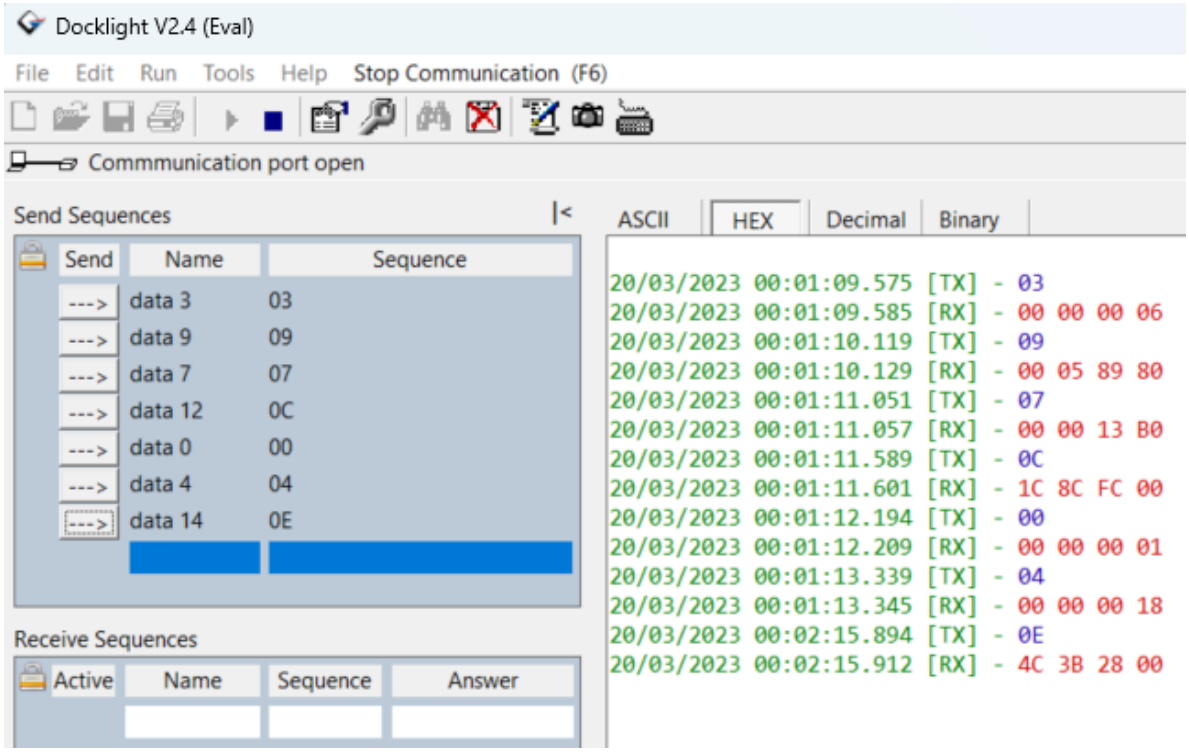


Imagen 2 – Ejecución programa

En la Imagen 2 podemos observar la ejecución del programa en el cual se le enviaron varios datos y se obtiene el factorial del dato recibido de manera exitosa. Es importante mencionar que debido a que el bus es de 32 bits. El máximo dato que se puede enviar y recibir el dato correcto es el 0xd, debido a que a partir del dato 0xe que se muestra en la imagen, se va a tener un desbordamiento porque el dato va a exceder los 32 bits. En este caso el factorial de 0xe es 14_4C_3B_28_00 y en nuestro programa se muestra solo los 32 bits [4C_3B_28_00].

- *Una captura de pantalla de la simulación de cada instrucción implementada donde señales los puntos clave de la ejecución en particular (seguir formato de simulación).*

Cada imagen cuenta con las instrucciones ejecutadas arriba en color amarillo

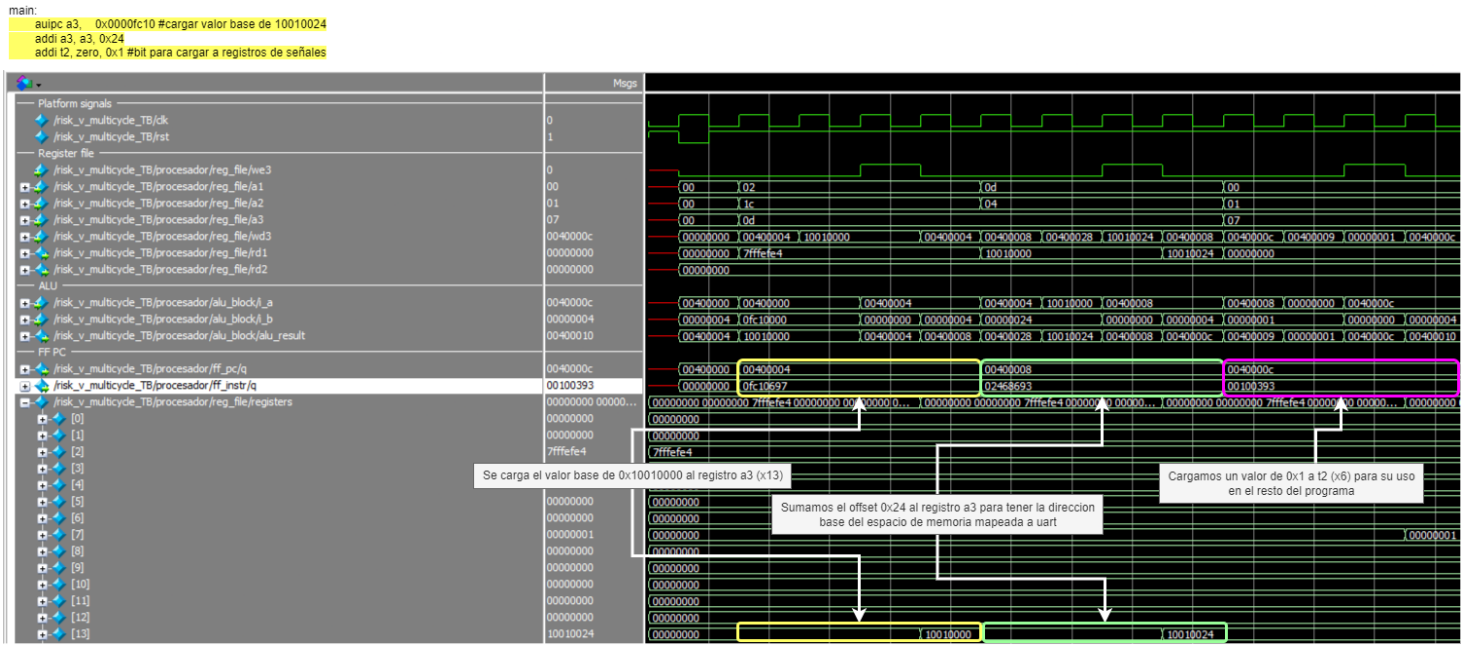


Imagen 3 – Simulación instrucciones bloque Main para inicialización

```
get_uart_data:
    lw t1, 0x10(a3) #obtener la señal de 0x10010034 para ver si ya recibimos un dato
    beq t1, zero, get_uart_data #checar si es un valor distinto de cero, sino seguir esperando
get_uart_data:
    lw t1, 0x10(a3) #obtener la señal de 0x10010034 para ver si ya recibimos un dato
    beq t1, zero, get_uart_data #checar si es un valor distinto de cero, sino seguir esperando
    lw a2, 0xC(a3) #loading number from address
    sw t2, 0x14(a3) #levantar señal para limpiar la bandera de rx
    sw zero, 0x14(a3) #bajar señal para limpiar la bandera de rx
```

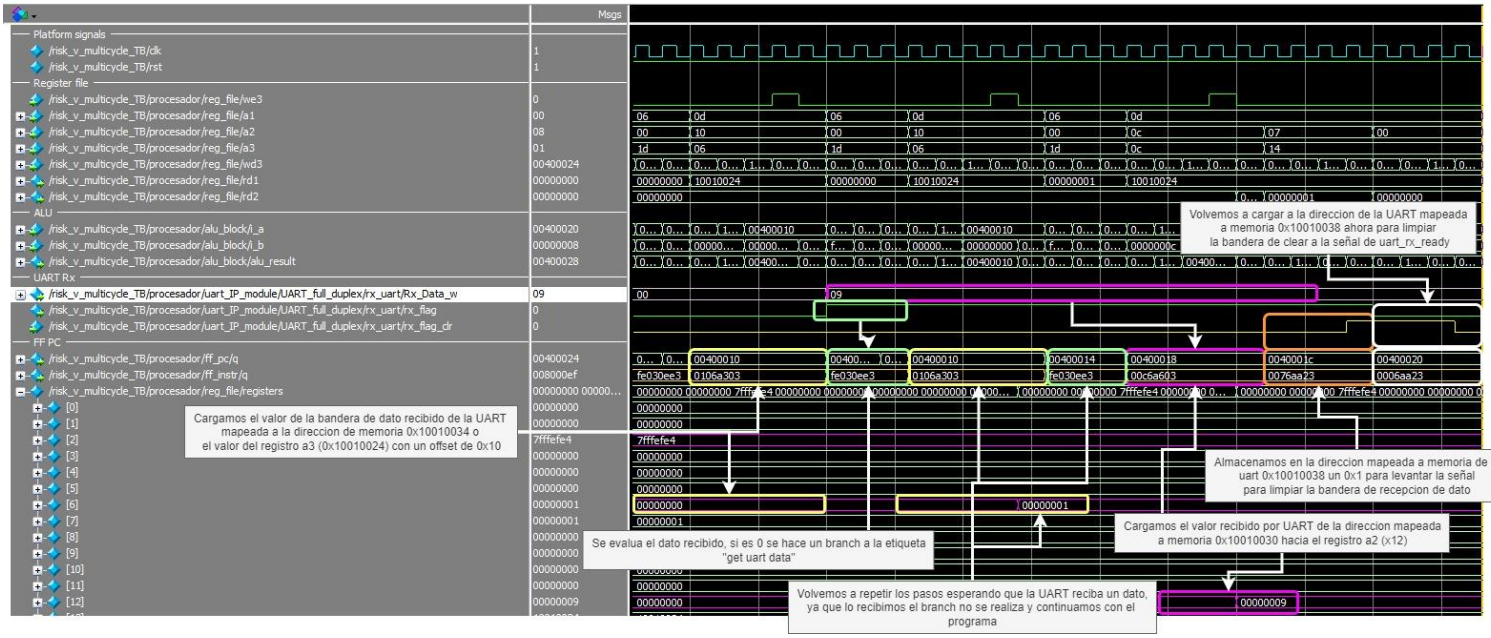


Imagen 4 – Simulación instrucciones cuando se recibe dato del UART Rx y se limpia la bandera uart_rx_ready

```
main_factorial:
    jal ra,factorial #calling procedure
    jal zero,send_uart_data # jump to uart data send

factorial:
    slti t0, a2, 1, #if n<1
    beq t0, zero, loop #branch to loop
    addi a0, zero, 1 #loading 1
    jalr zero,ra,0 #return to the caler
```

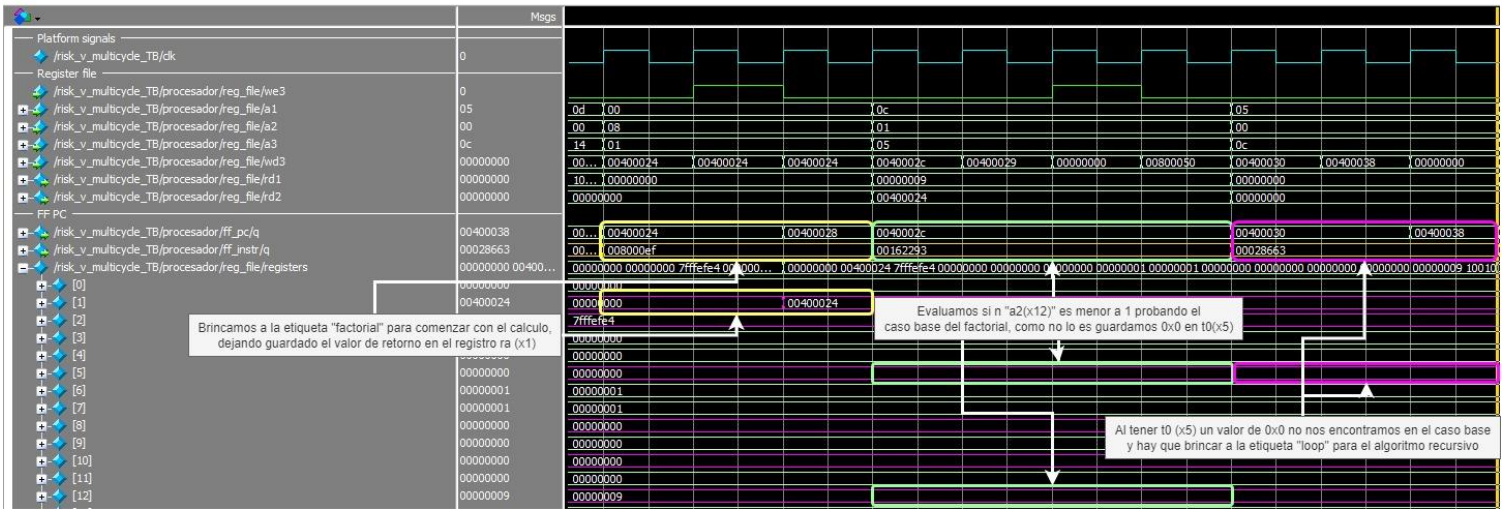


Imagen 5 – Simulación instrucciones cuando se ejecuta factorial de N

```
loop:
    addi sp, sp, -8 #decreasing stack pointer
    sw ra, 4(sp) #storing n
    sw a2, 0(sp) #storing return addres
    addi a2, a2, -1 #Decreasing n
    jal ra,factorial #recursive function
factorial:
    slti t0, a2, 1, #if n<1
    beq t0, zero, loop #branch to loop
lw a2, 0(sp) #load value from stack
lw ra, 4(sp) # load ra from stack
addi sp, sp, 8 #increase stack pointer
mul a0, a2, a0 # n*factorial(n-1)
jalr zero,ra,0 #return to caller
```

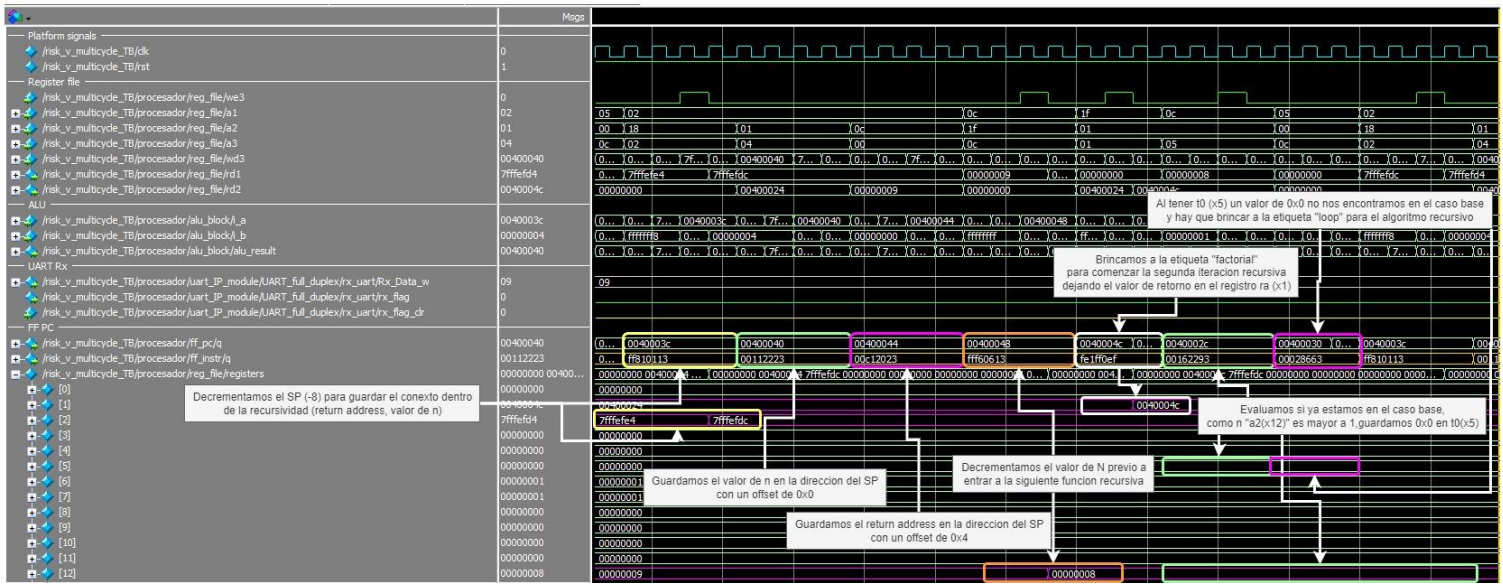


Imagen 6 – Simulación recursividad del factorial

```
loop:
    addi sp, sp, -8 #decreasing stack pointer
    sw ra, 4(sp) #storing n
    sw a2, 0(sp) #storing return addres
    addi a2, a2, -1 #Decreasing n
    jal ra,factorial #recursive function
factorial:
    slti t0, a2, 1, #if n<1
    beq t0, zero, loop #branch to loop
    addi a0, zero, 1 #loading 1
    jalr zero,ra,0 #return to the caller
loop:
    addi sp, sp, -8 #decreasing stack pointer
    sw ra, 4(sp) #storing n
    sw a2, 0(sp) #storing return addres
    addi a2, a2, -1 #Decreasing n
    jal ra,factorial #recursive function
    lw a2, 0(sp) #load value from stack
    lw ra, 4(sp) # load ra from stack
    addi sp, sp, 8 #increase stack pointer
    mul a0, a2, a0 # n*factorial(n-1)
```

Captura realizada una vez que a2 llega a 0, implicando que se llegó al caso base de la recursividad

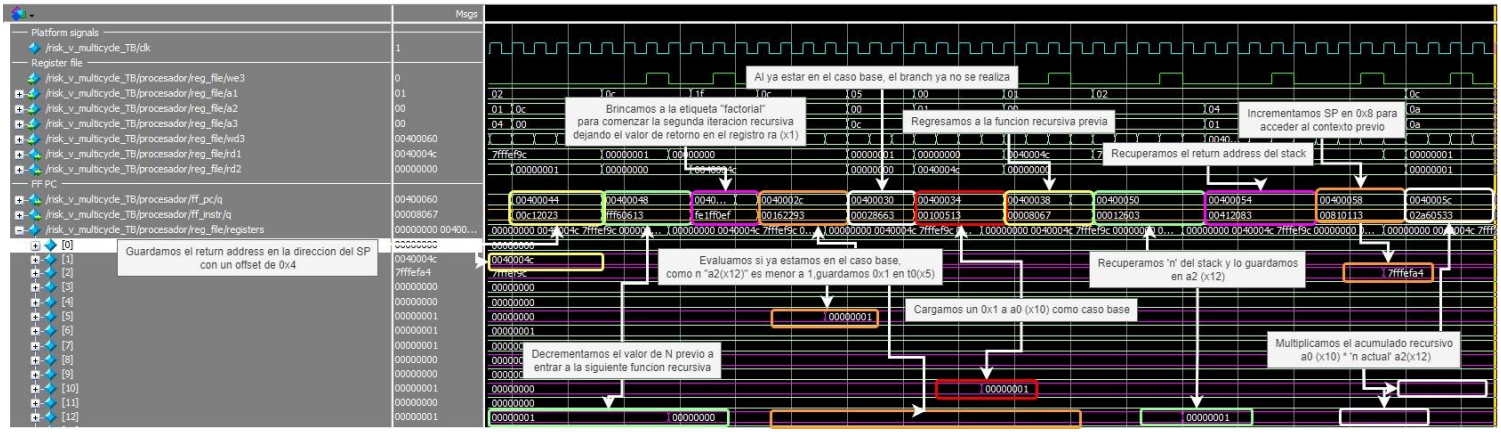


Imagen 7 – Simulación secuencia recursiva llegó a 0 para realizar multiplicación del acumulado recursivo.

```
loop:
    jalr zero,ra,0 #return to caller
loop:
    lw a2, 0(sp) #load value from stack
    lw ra, 4(sp) # load ra from stack
    addi sp, sp, 8 #increase stack pointer
    mul a0, a2, a0 # n*factorial(n-1)
    jalr zero,ra,0 #return to caller
loop:
    lw a2, 0(sp) #load value from stack
    lw ra, 4(sp) # load ra from stack
    addi sp, sp, 8 #increase stack pointer
    mul a0, a2, a0 # n*factorial(n-1)
    jalr zero,ra,0 #return to caller
```

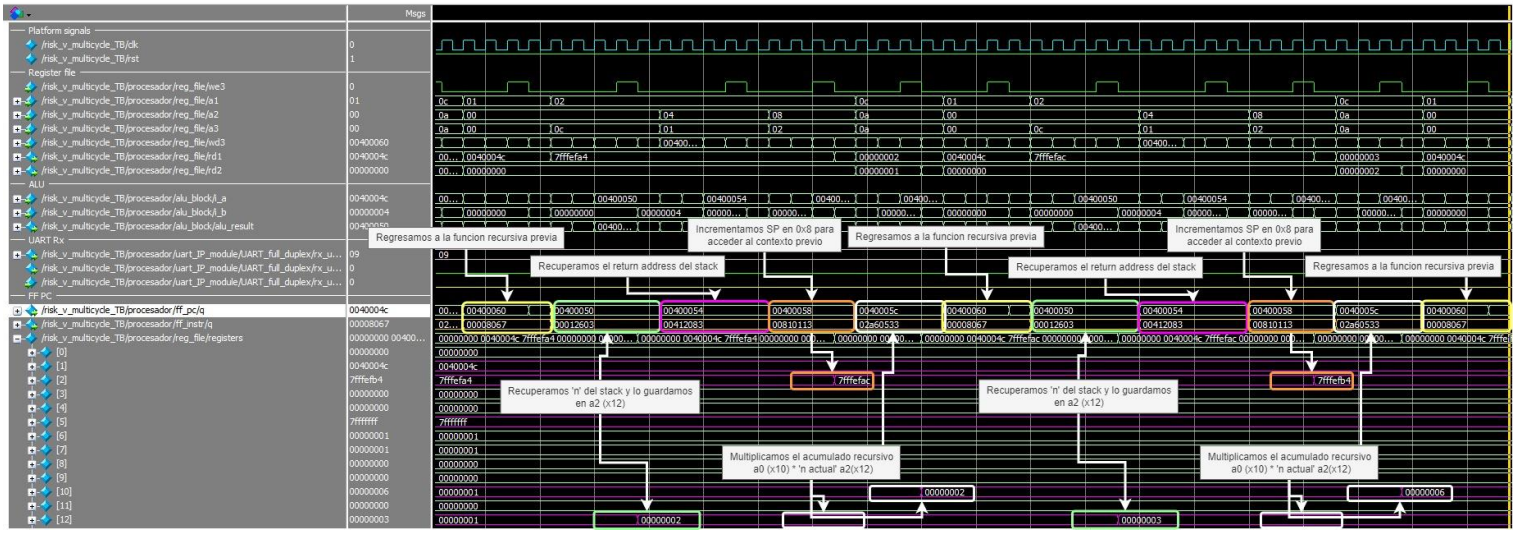


Imagen 8 – Simulación retorno y ejecución función previa

```
loop:
    jalr zero,ra,0 #return to caller
main_factorial:
    jal zero,send_uart_data # jump to uart data send

send_uart_data:
    addi s0, zero, 0x20 #contador i para el loop
send_loop:
    addi s0, s0, -0x8 #decrease counter
    srl t3, a0, s0 #shift right register factorial number to get 8 bits to send
    sw t3, 0(a3) #cargar el factorial en la direccion de memoria de dato UART
    sw t2, 4(a3) #cargar el bit 1 a la señal de enviar tx de uart
    sw zero, 4(a3) #cargar el bit 0 a la señal de enviar tx de uart tipo one shot
wait_to_send:
    lw t1, 8(a3) #obtener el valor de la bandera que termino de enviar la transmision
    beq t1, zero, wait_to_send #checar si es un valor distinto de cero, sino seguir esperando que termine de enviarse
```

Captura realizada una vez que el return address apunta al ciclo principal de main, fuera de la recursion

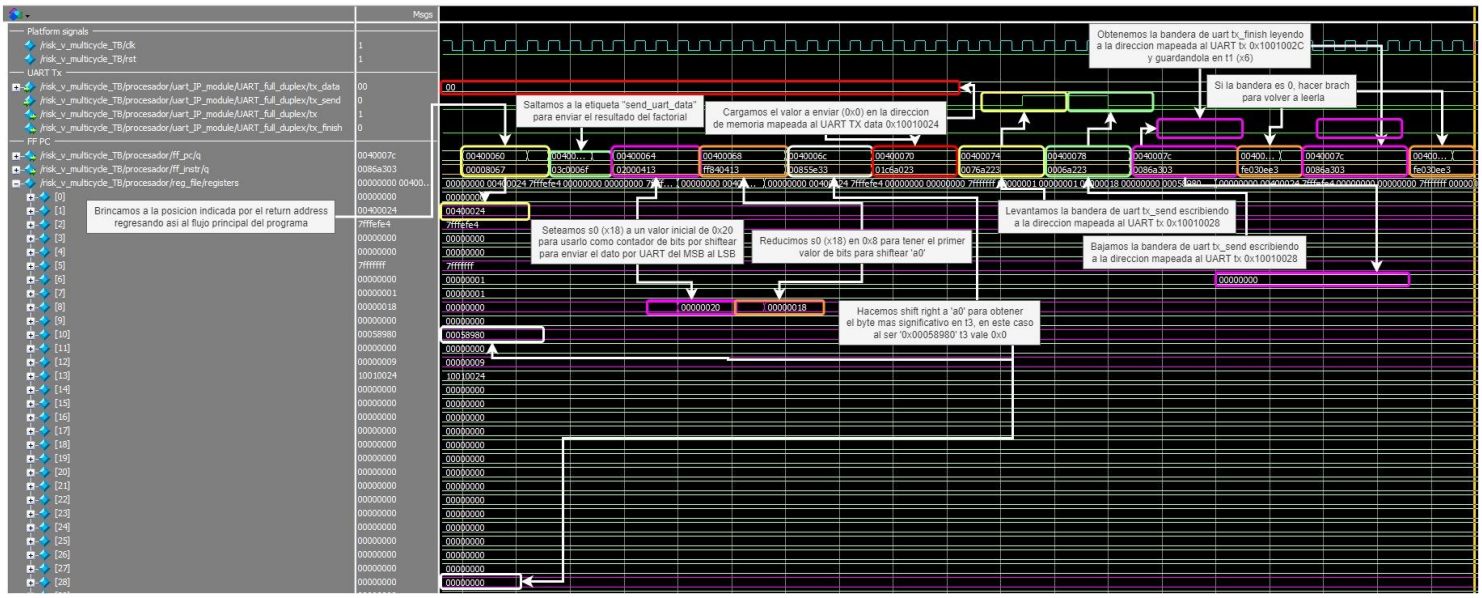


Imagen 9 – Simulación fuera recursión y envió de datos al UART Tx

```
wait_to_send:
    lw t1, 8(a3) #obtener el valor de la bandera que termino de enviar la transmision
    beq t1, zero, wait_to_send #checar si es un valor distinto de cero, sino seguir esperando que termine de enviarse
wait_to_send:
    lw t1, 8(a3) #obtener el valor de la bandera que termino de enviar la transmision
    beq t1, zero, wait_to_send #checar si es un valor distinto de cero, sino seguir esperando que termine de enviarse
    bne s0, zero, send_loop #check if al bytes have been sent
send_loop:
    addi s0, s0, -0x8 #decrease counter
    srl t3, a0, s0 #shift right register factorial number to get 8 bits to send
    sw t3, 0(a3) #cargar el factorial en la direccion de memoria de dato UART
    sw t2, 4(a3) #cargar el bit 1 a la señal de enviar tx de uart
    sw zero, 4(a3) #cargar el bit 0 a la señal de enviar tx de uart tipo one shot
wait_to_send:
    lw t1, 8(a3) #obtener el valor de la bandera que termino de enviar la transmision
```

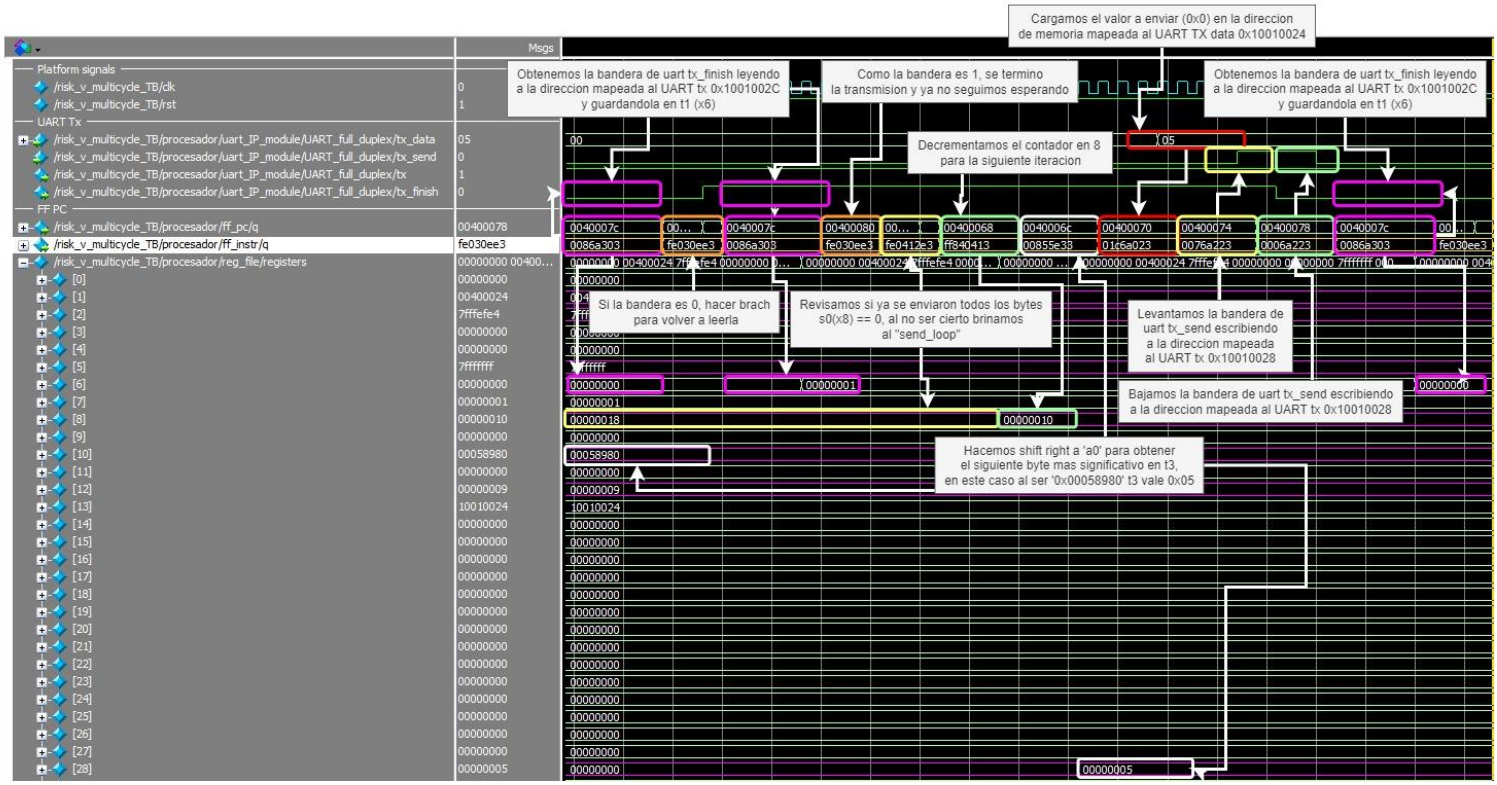


Imagen 10 – Simulación función recursiva para transmitir 32 bits en cadenas de 8bis al UART.

```
wait_to_send:
    lw t1, 8(a3) #obtener el valor de la bandera que termino de enviar la transmision
    beq t1, zero, wait_to_send #checar si es un valor distinto de cero, sino seguir esperando que termine de enviarse
wait_to_send:
    lw t1, 8(a3) #obtener el valor de la bandera que termino de enviar la transmision
    beq t1, zero, wait_to_send #checar si es un valor distinto de cero, sino seguir esperando que termine de enviarse
    bne s0, zero, send_loop #check if al bytes have been sent
    jal zero,get_uart_data # jump to wait next factorial number
get_uart_data:
    lw t1, 0x10(a3) #obtener la señal de 0x10010034 para ver si ya recibimos un dato
    beq t1, zero, get_uart_data #checar si es un valor distinto de cero, sino seguir esperando
get_uart_data:
    lw t1, 0x10(a3) #obtener la señal de 0x10010034 para ver si ya recibimos un dato
```

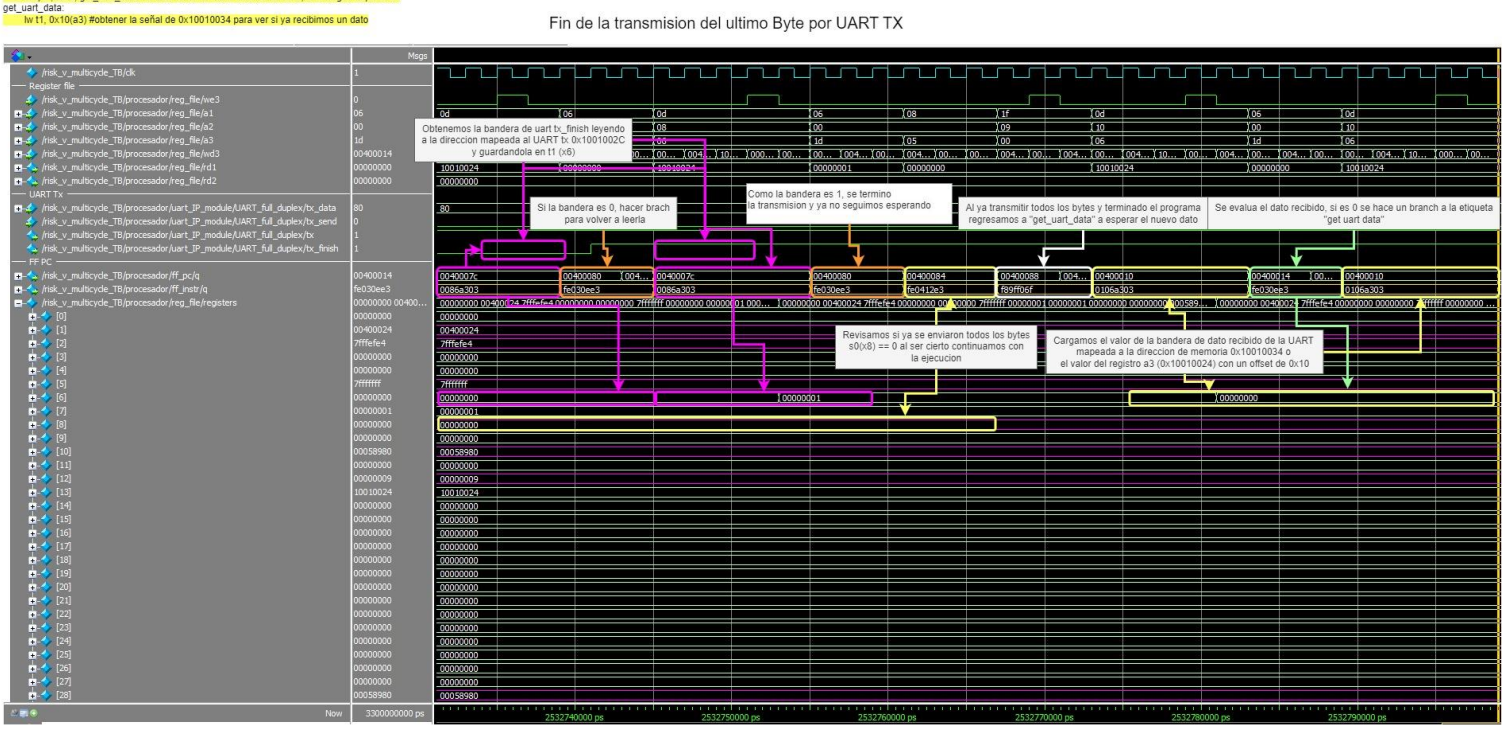


Imagen 11 – Fin de la transmisión del ultimo Byte por UART Tx

En la siguiente Imagen 12 podemos observar cómo se utilizó el stack para almacenar los contextos del programa conforme las funciones recursivas se fueron ejecutando.

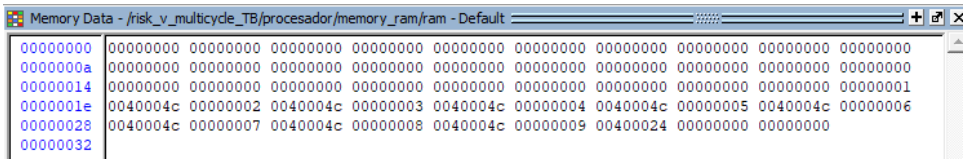


Imagen 12 – Uso del stack en memoria RAM

- *Una tabla donde se muestre el CPI de cada una de las instrucciones implementadas.*

En la Tabla 1 se puede observar cuantos ciclos toma por instrucción en base a nuestro diseño.

Instruction	Type	Clocks cycles
auipc	U-type	4
addi	I-type	4
lw	I-type	5
slti	I-type	4
jalr	I-type	4
beq	B-type	3
bne	B-type	3
sw	S-type	4
jal	J-type	3
mul	R-type	4
srl	R-type	4

Tabla 1 – Clocks cycles per instruction

Se saco un promedio de cuanto es el CPI por tipo de instrucción y los resultados están expresados en la Tabla 2.

Type	CPI
U-type	4
I-type	4.25
B-type	3
S-type	4
J-type	3
R-type	4

Tabla 2 – Promedio de CPI por tipo de instrucción

- *Grafica donde se muestre la relación n vs time CPU para n = 0 hasta n = 15.*

n	CPU time ns
0	128
1	224.852
2	320
3	416.009
4	512
5	608.203
6	705.914
7	802.882
8	896
9	992.331
10	1088
11	1184
12	1280
13	1378.869
14	1471.113
15	1565.564

Tabla 3 – CPU time en cada iteración de N

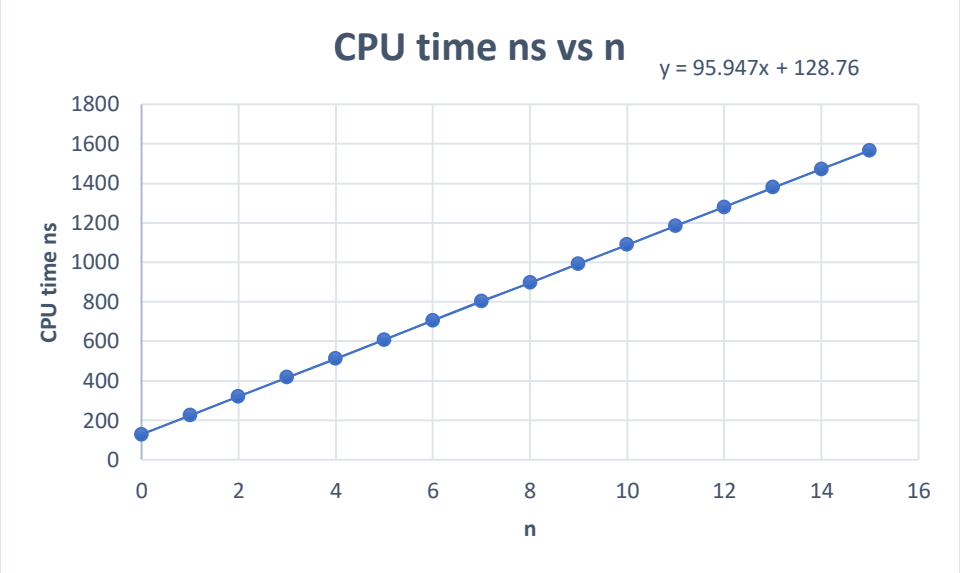


Imagen 13 Grafica de CPU time en nanosegundos vs el valor de n

En la Imagen 13 podemos observar como el comportamiento de tiempo de ejecución en comparación con el valor de N es lineal. Esto se puede observar también con la ecuación dentro de la imagen, siendo ‘y’ el CPU time y ‘x’ el valor de ‘n’.

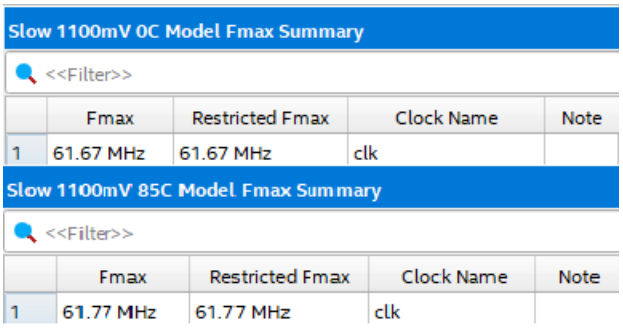
- *Resultados de síntesis en términos de logic elements (LEs) y la frecuencia máxima de operación de la implementación.*

Estos fueron los resultados de la compilación de la implementación realizada. Se puede observar en la Imagen 14 que se está utilizando un total de 2,616 ALM y un total de 4064 registros.

Flow Status	Successful - Mon Mar 20 02:02:43 2023
Quartus Prime Version	21.1.1 Build 850 06/23/2022 SJ Lite Edition
Revision Name	risk_v_multicycle
Top-level Entity Name	risc_v_top
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	2,616 / 41,910 (6 %)
Total registers	4064
Total pins	4 / 499 (< 1 %)
Total virtual pins	0
Total block memory bits	1,771,520 / 5,662,720 (31 %)
Total DSP Blocks	2 / 112 (2 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

Imagen 14 – Recursos utilizados en la FPGA

En base al análisis de tiempo, se puede determinar que a una temperatura de 0°C la frecuencia máxima de operación es de 61.67 MHz, mientras que a una temperatura de 85°C se tiene una frecuencia máxima de operación de 61.77 MHz. Esto nos da un promedio de frecuencia de operación de 61.72 MHz.



Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	61.67 MHz	61.67 MHz	clk	

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	61.77 MHz	61.77 MHz	clk	

Imagen 15 – Frecuencia máxima de operación

- *Microarquitectura en Visio o programa equivalente.*

Se diseñó la microarquitectura usando el programa DrawIO, en el diseño optamos por seguir el diseño de la Imagen 16, este archivo se puede encontrar en la carpeta docs/.

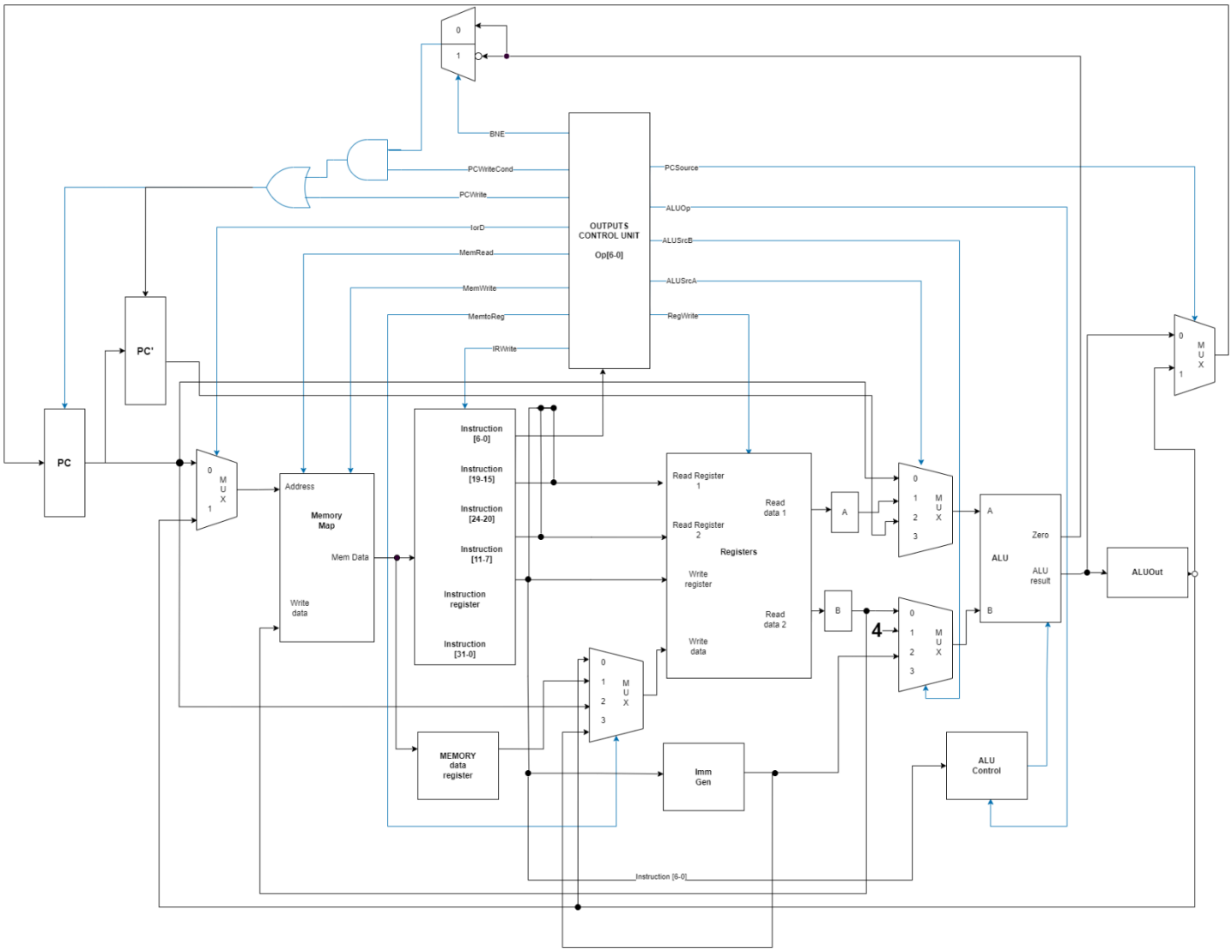


Imagen 16 – Arquitectura Risc-V implementada

En la Imagen 16 se puede observar cómo es el diagrama de la microarquitectura el cual tiene elementos claves como el memory map, register file, control unit, ALU, IMM generator, entre otros. Este diagrama fue adaptado para poder ejecutar instrucciones tipo J y B.

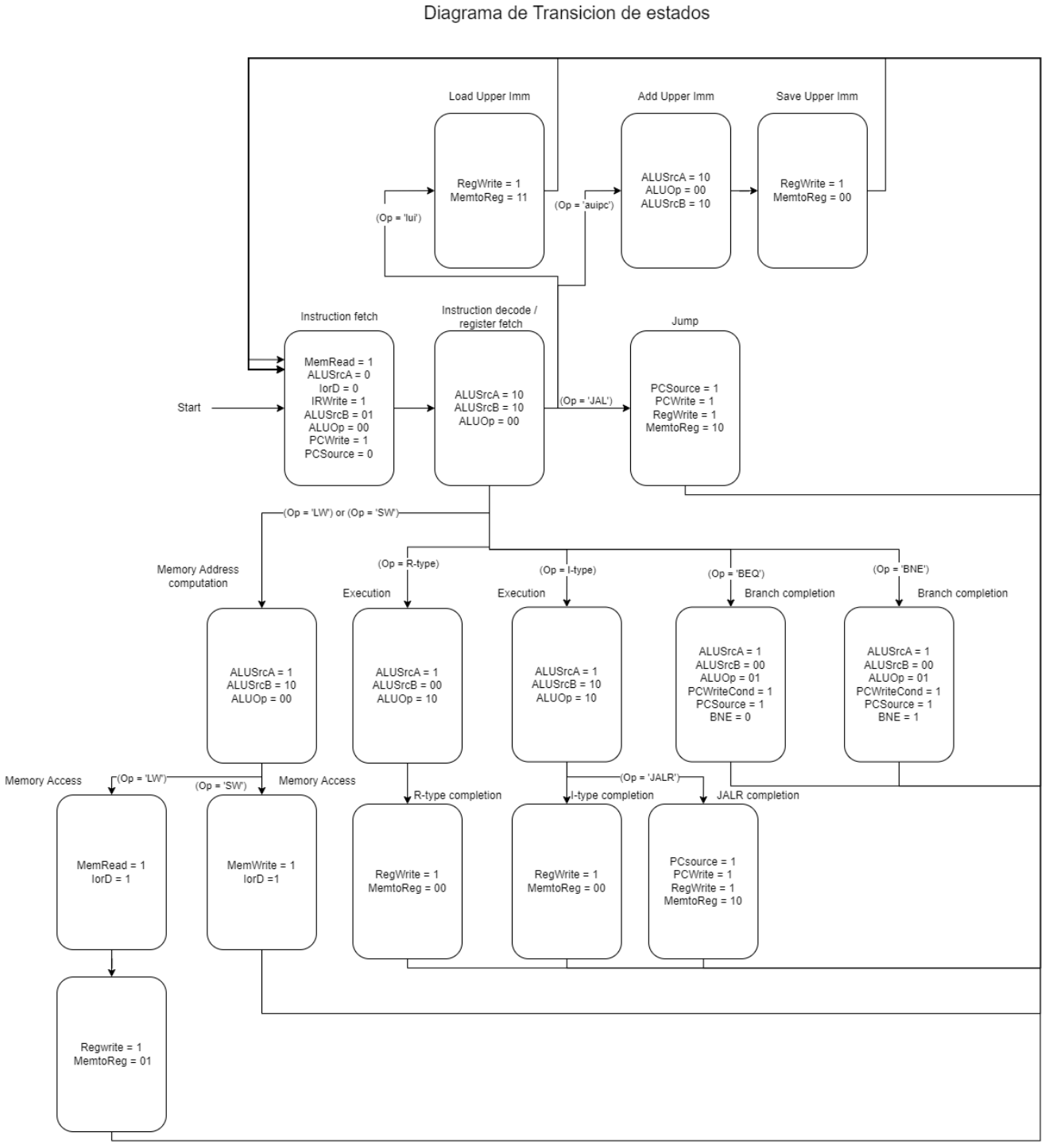


Imagen 17 – Diagrama de transición de estados

En la Imagen 17 tenemos el diagrama de estados que se implementó. En esta FSM se incluyo el soporte a los 6 tipos de instrucciones para que el procesador no tenga limitaciones con el modulo RV32I.

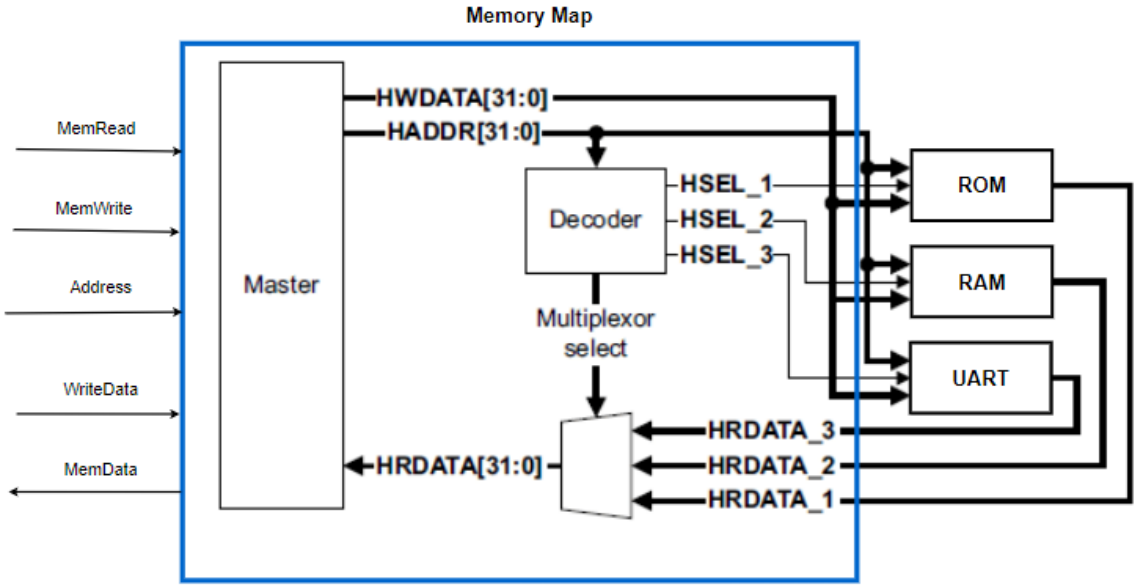


Imagen 18 – Arquitectura memory map

En la Imagen 18 se puede observar el diagrama que se siguió para el diseño del memory map, el cual tiene varias IPs conectadas (ROM, RAM, UART) y se elige a cuál de estas se le va leer o escribir un dato.

- *Captura de pantalla de las señales internas usando el Signal Tap donde se muestre la recepción de n.*

Se uso Signal tap para poder capturar la trama de Rx y obtener el dato que posteriormente será guardado en un registro del UART. Este registro será accedido por el procesador por medio del mapa de memoria para saber que dato se le hará factorial y leer la bandera de si ya se tiene dato o no.

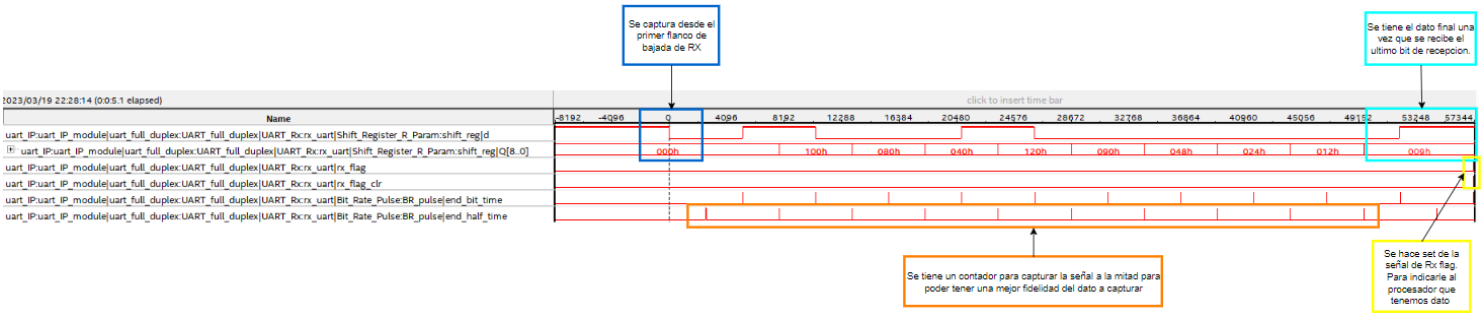


Imagen 19 – Captura de trama Rx usando signal tap

El trigger usado para capturar la trama Rx como lo muestra la Imagen 19 fue el flanco de bajada del dato en Rx, debido a que el que cambie a 0, indica el bit de inicio de una trama esta por recibirse. Se tienen contadores para poder capturar el dato a la mitad y poder tener una mejor fidelidad del dato a capturar. En este ejemplo se mando el numero 0x9 en la plataforma serial, por lo que si empezamos a contar la trama recibida del lsb al msb, se obtiene un 0000_1001 sin tomar en cuenta el bit de inicio, bit paridad ni bit final. Este numero equivale a un 0x9 que es justo el dato que ya tenemos almacenado al final de la trama en uno de los registros del módulo.

Haciendo un zoom en la parte final del módulo podemos observar como el UART declara la bandera de Rx flag, que es guardada en un registro que lo consume el procesador para saber que tenemos dato, y luego posteriormente se borra por la señal de Rx flag clear seteado por el procesador como se muestra en la Imagen 20.

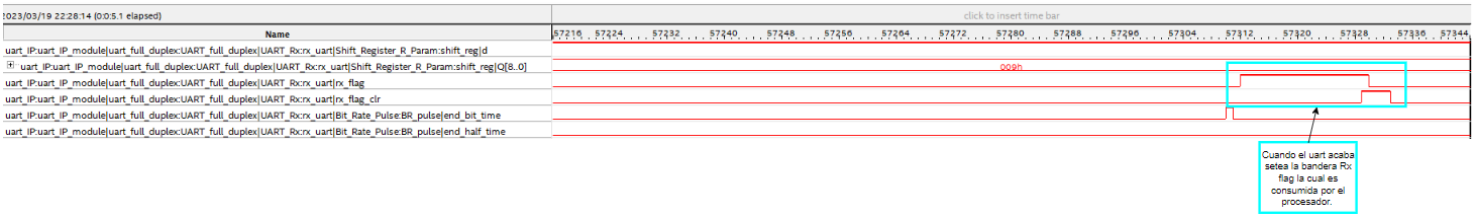


Imagen 20 – Set y clear de la bandera Rx flag consumida por el procesador para saber si ya hay dato en la UART.

También se utilizó signal tap para poder capturar la trama de Tx como se muestra en la Imagen 21. La cual como trigger inicio cuando se recibió un Tx send y se tenia un valor a enviar. Se puede observar en la imagen que el valor a enviar es de 0x5, se empieza con el bit de inicio para indicar que se va a empezar a transmitir la trama. Después se envía del lsb al msb el dato, en este caso se envió un 0000_0101 que es equivalente a 0x5. Una vez que se envía el dato completo se setea el flag de delayer para podernos mover de estado en la FSM.

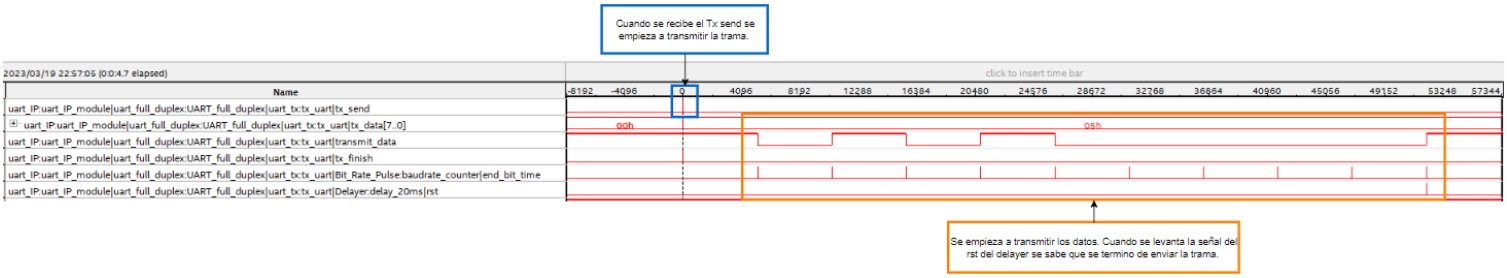


Imagen 21 – Captura de trama Tx

- *Todas las fuentes deben ser debidamente comentadas.*

Los archivos se comentaron en la parte superior de cada uno, especificando el nombre del módulo, función y fecha. Así como ciertos comentarios en los archivos. Estos archivos se encuentran en el repositorio en la carpeta /src.

- *Código fuente en Canvas y un repositorio en git hub, agregar al repositorio a iteso-jlpe. La estructura del repositorio debe serguir la misma estructura de la tarea del GPIO. Por cada instrucción agregada se espera al menos un commit. El formato de repositorio será DM_Apellido_1_Apellido_2_RISC_V. La implementación debe se debe encontrar en un branch con nombre Multiciclo.*

Todo el código, documentos, archivos de quartus se encuentran en el repositorio DM_Michel_Castillo_RISC_V.

- *Definición de las direcciones utilizadas para la comunicación con la UART (mapeo de direcciones utilizadas en relación con los registros internos).*

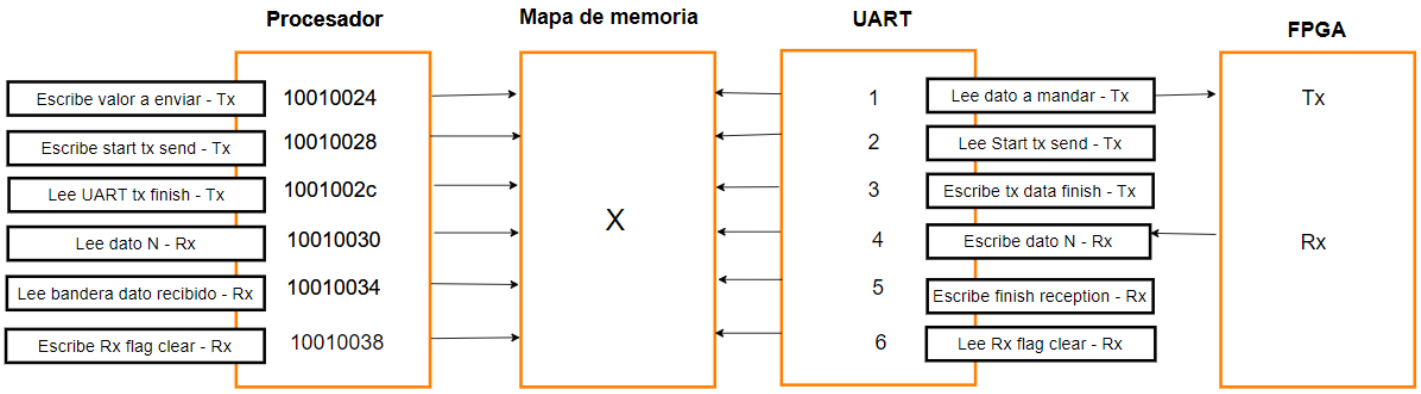


Imagen 22 – Mapeo de direcciones usadas entre el procesador y la UART.

En la Imagen 22 se muestra cómo se definió que direcciones del procesador iban a ser utilizadas para la comunicación de la UART, y en base a estas definiciones se realizó el programa del procesador. El mapa de memoria al saber cómo están mapeadas las IPs (procesador, ROM, RAM, UART) tiene reglas para determinar a quien hablarle y escribirle.

- *Programa en ensamblador de utilizado para la comunicación.*

El programa en ensamblador se encuentra en la carpeta /asm en el repositorio de GitHub. Dentro de esa carpeta también se encontrará el programa en binario el cual consume el procesador para ejecutar el programa.