Universidad Nacional de Córdoba.

Facultad de Ciencias Exactas, Físicas y Naturales.

ARQUITECTURA DE COMPUTADORAS

Trabajo Final

Procesador MIPS

Alumnos

- Carrizo, Facundo ... 39419552

Docentes

- Rodriguez, Santiago
- Pereyra, Martin

Índice

Introducción

Para el Trabajo Final de la materia, se realizó la implementación en Verilog del pipeline de cinco etapas del procesador MIPS. El mismo se trata de un procesador de 32 bits, con 32 registros, una memoria de instrucciones de 128 posiciones y una memoria de datos de 128 posiciones para simplificar el sintetizado y el envío de datos a la PC. El mismo fue desarrollado mediante lenguaje de especificación de hardware Verilog y haciendo uso del entorno de desarrollo Vivado.

Objetivos

Los requerimientos del trabajo son los siguientes:

- Se debe implementar el procesador MIPS segmentado en las siguientes etapas:
    - Instruction Fetch: busca la instrucción en la memoria de programa.
    - Instruction Decode: decodifica la instrucción y lee los registros.
    - Execute: ejecuta la instrucción.
    - Memory Access: lee o escribe la memoria de datos.
    - Write back: escribe los resultados en los registros.

- Las instrucciones a implementar del set de instrucciones del MIPS son las siguientes:
    - R-type: SLL, SRL, SRA, SLLV, SRLV, SRAV, ADDU, SUBU, AND,OR, XOR, NOR, SLT.
    - I-Type: LB, LH, LW, LWU, LBU, LHU, SB, SH, SW, ADDI, ANDI, ORI, XORI, LUI, SLTI, BEQ, BNE, J, JAL.
    - J-Type: JR, JALR.

  Su funcionamiento y descripción puede encontrarse en el manual de instrucciones

- El procesador debe tener soporte para los riesgos estructurales, de datos y de control mediante:
    - Unidad de cortocircuitos.
    - Unidad de detección de riesgos.
    - El programa a ejecutar debe ser cargado en la memoria de programa mediante un archivo ensamblado.

- El archivo de ensamblado debe convertirse a código de instrucción y transmitirse a través de una interfaz UART antes de comenzar su ejecución;

- Se debe simular una unidad de debug que envíe información hacia y desde el procesador mediante UART. La información es la siguientes:
    - Contador de programa (PC).
    - Contenido de los 32 registros.
    - Contenido de la memoria de datos usada.
    - Cantidad de ciclos de reloj ejecutados (Se añade esta información para verificar la cantidad de ciclos totales de ejecución del programa cargado)
- Una vez cargado el programa a ejecutar, el procesador debe permitir dos modos de operación:
    - Continuo: se envía un comando a la FPGA por la UART y se inicia la ejecución del programa hasta llegar al final del mismo (instrucción HALT). Luego, se muestra la información pedida en la pantalla.
    - Paso a paso: se envía un comando a la FPGA por la UART, se ejecuta un ciclo de clock y se muestra la información pedida. Seguido de recibir nuevamente un comando ya sea un ciclo más o en modo continuo hasta que termine de ejecutarse todo el programa.

- El clock del sistema debe crearse utilizando el ip-core correspondiente.
- Se debe mostrar el reporte de timing con la frecuencia máxima de reloj soportada.

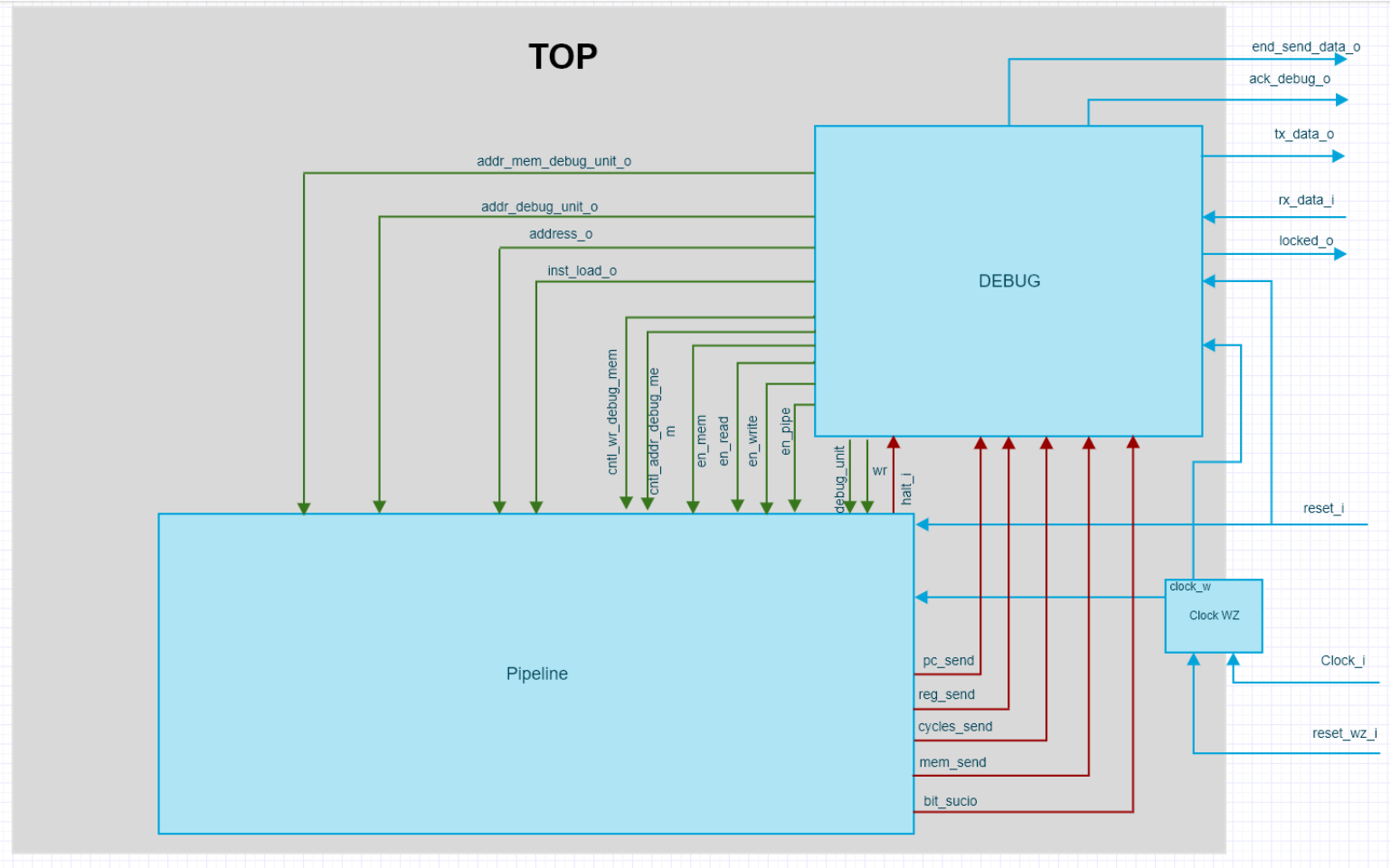
- Se debe validar el desarrollo por medio de Test Bench.

Desarrollo

A Continuación se detalla cada uno de los módulos implementados.

Modulo TOP

Este módulo se instancia el pipeline, el clock wizard y el debug unit.



Pipeline segmentado

En este módulo se instancian los siguientes módulos:

- IF (Etapa de búsqueda de la instrucción).
  - program_counter (Contador de Programa).
  - pc_adder: Sumador pc + 1
  - mem_inst (Memoria de instrucciones).

- ID (Etapa de decodificación).
  - hazard_detection (Módulo de generar una burbuja; Más adelante se va a detallar este módulo en concreto).
  - unit_branch: Se encarga de comparar los registros para ver si se toma el salto o no y además de generar la dirección de salto.
  - bank_register: Banco de registros.
  - unit_control: Unidad de control, se encarga de decodificar la instrucción y de generar las señales para las siguientes etapas.
  - sign_ext : Extiende el signo a 32 bits.

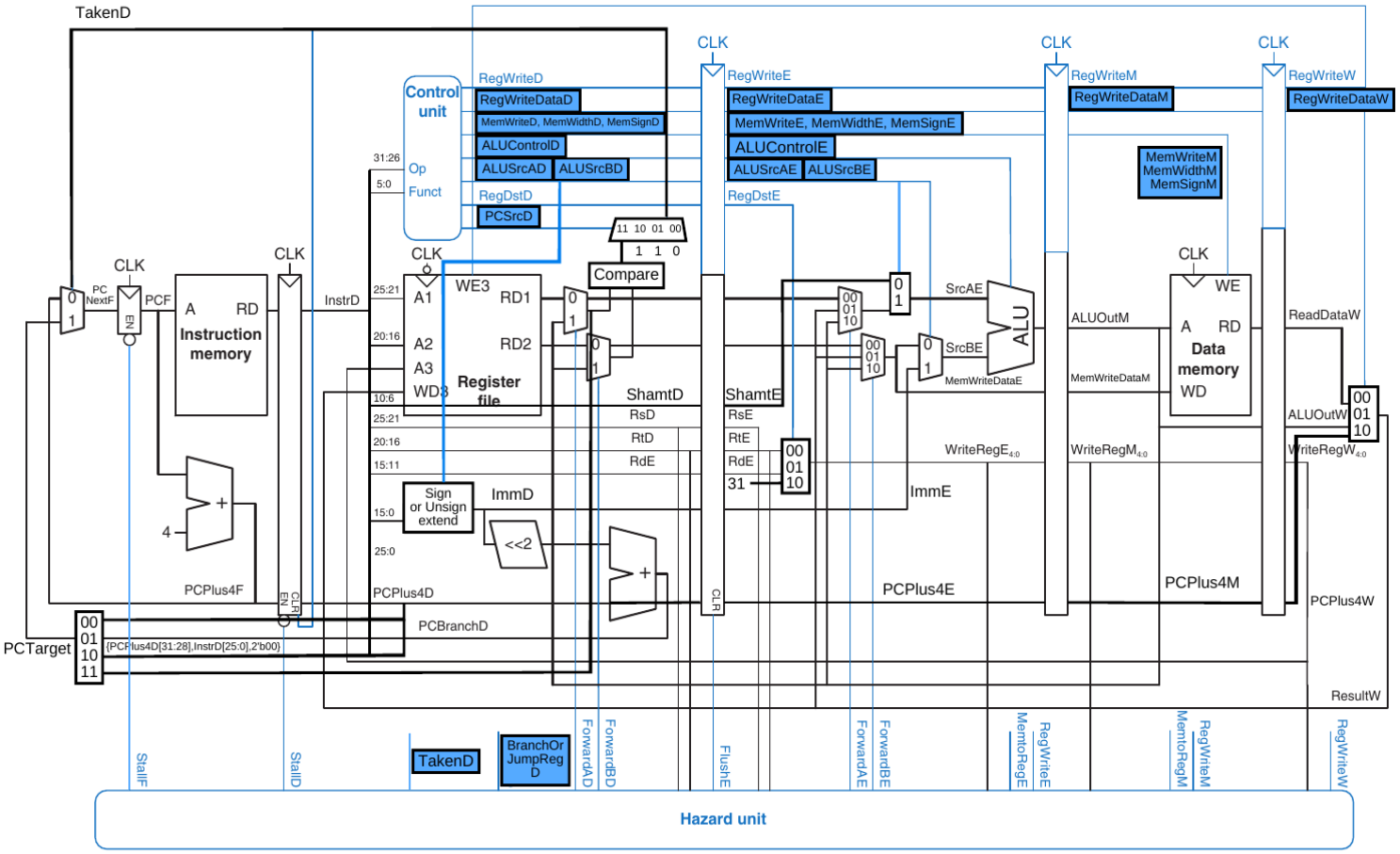
- EX (Etapa de ejecución).
  - Alu (Unidad aritmético lógica): Se reutilizo la alu implementada en los TPs anteriores, añadiendo algunas operaciones adicionales.
  - Alu_control (Unidad de control de la alu): Se encarga de generar el código de operación para la alu, en base a la señal de control proveniente de la etapa previa.
  - unit_forward (Unidad de cortocircuitos): Se encarga de redireccionar los datos de las etapas posteriores a las entradas de la alu siempre y cuando haya dependencias de datos (Más adelante se va a detallar este módulo en concreto).
- MEM (Etapa de acceso a memoria)

- ➢ cntl_bit_sucio: Se encarga de setear y devolver que parte de la memoria está sucia o no, para poder desde la debug unit enviar solo la información útil.
- ➢ controller_mem: Permite configurar los datos de un byte, half word o word signado tanto para insertar en memoria o devolver de la memoria con su correspondiente tamaño.
- ➢ mem_data (Memoria de datos).

- ❖ WB (Etapa de write-back): Este módulo sólo está compuesto de un multiplexor de las señales proveniente del registro MEM/WB.
- ❖ Registros intermedios
  - ❖ IF/ID
  - ❖ ID/EX
  - ❖ EX/MEM
  - ❖ MEM/WB

- ❖ Count_cycles: Se agrego este módulo para verificar la cantidad de ciclos ejecutados en el pipeline. Se habilita recién cuando se da inicio a la ejecución del programa.

A continuación se muestra el diagrama top del pipeline que se siguió de modelo para implementarlo.



Como se mencionó anteriormente, tanto el módulo Hazard detection y unit forward son fundamentalmente los encargados tanto de insertar una burbuja y de adelantar los datos cuando corresponda.

Ambos son bloques puramente combinacionales.

Hazard detection

Se van a insertar una burbuja bajo ciertas condiciones, que se detallan abajo:

1. Condición (LOAD + TIPO_R O INM)
    - ID/EX.memRead = 1 && ID/EX.RegisterRt = ID/RegisterRt OR ID/EX.RegisterRt = ID/RegisterRs

    Ejemplo:
    - lw $2, 4($0)
    - and $4,$2,$3

Como se puede observar el destino de la primera instrucción se lee en la etapa ID,y recién en la etapa mem se consigue el dato, por lo tanto se debe insertar una burbuja.

2. Condición (LOAD|R|I + BRANCH)
    - EX.RegWrite = 1 && EX.writeReg = ID/RegisterRt OR EX.writeReg = ID/RegisterRs

    Ejemplo:
    - lw $2, 4($0)
    - beq $4,$2, label

    or
    - and $2, $4, $4
    - beq $4,$2, label

En ambas condiciones se deben insertar burbujas. Se entiende por burbuja a detener la carga y actualización del reg IF_ID y del program counter durante un ciclo de clock.

Branch Forwarding

Este módulo tiene el fin de forwarding del resultado del latch EX/MEM cuando se detecte que en la condición de evaluación de los registros fuentes de las instrucciones de saltos condicionales tiene como destino el registro de la previa instrucción de salto.
En otras palabras en la siguiente condición:

- EX/MEM.RegisterRd = ID.RegisterRs
- EX/MEM.RegisterRd = ID.RegisterRt

Forwarding Unit

Las condiciones por el cual se produce un cortocircuito son las siguientes:

- EX/MEM.RegisterRd = ID/EX.RegisterRs
- EX/MEM.RegisterRd = ID/EX.RegisterRt
- MEM/WB.RegisterRd = ID/EX.RegisterRs
- MEM/WB.RegisterRd = ID/EX.RegisterRt

Debug unit

Este módulo consta de una máquina 12 estados en cual se van a detallar a continuación.
Además en este módulo se integra la uart implementada en el TP2 para recibir y enviar los datos a la PC.

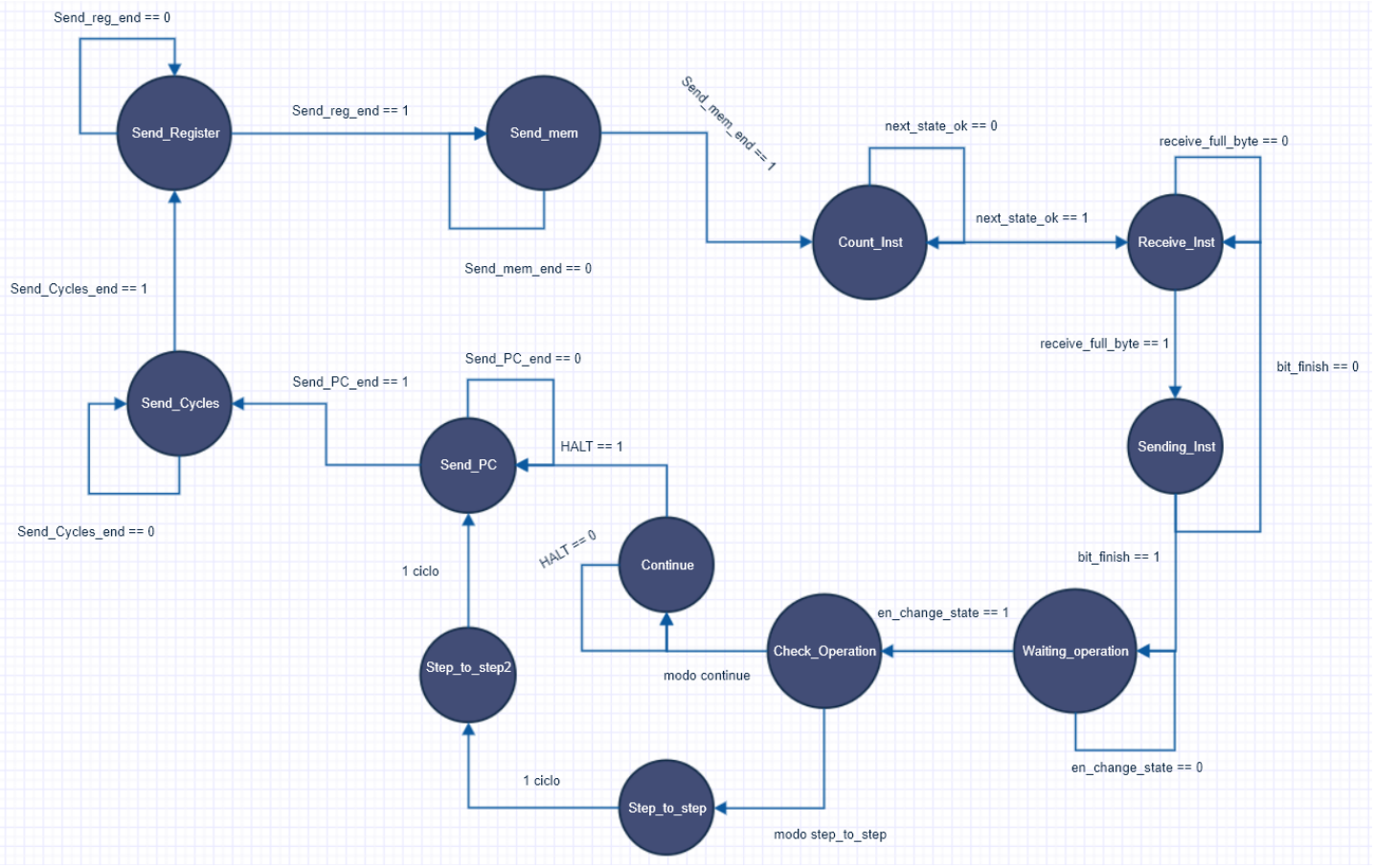
Estados:

- Count_Instr
- Receive_Instr
- Sending_Instr
- Waiting_operation
- Check_Operation

- Step_to_step
- Step_to_step_2
- Continue
- Sending_data_pc
- Sending_count_cyles
- Sending_data_registers
- Sending_data_mem

Count_inst: Espera por la cantidad de instrucciones a cargar en memoria.
Receive_Instr: Estado en el cual se están recibiendo los 4 bytes de la instrucción a cargar.
Sending_Instr: Carga la instrucción recibida en memoria.
Waiting_operation: Espera por un el modo de ejecución, continuo o paso a paso.
Check_Operation: Check que el modo de operación sea el correcto.
Step_to_step: Estado que se habilita un solo ciclo de ejecución.
Step_to_step_2: Estado en el que se deshabilita el pipeline y se pasa al envío de datos a la PC.
Continue: Estado en el que el pipeline ejecuta continuo hasta que reciba una instrucción HALT.
Sending_data_pc: Se envía a la PC el program counter
Sending_count_cyles: Se envía a la PC la cantidad de ciclos ejecutados en el pipeline.
Sending_data_registers: Se envían los 32 registros a la PC.
Sending_data_mem: Se envían la memoria de datos utilizada.

A continuación se muestra el diagrama de la Máquina de estado.


Send_reg_end == 0
Send_Register
Send_reg_end == 1
Send_mem
Send_mem_end == 1
Send_mem_end == 0
next_state_ok == 0
Count_Inst
next_state_ok == 1
receive_full_byte == 0
Receive_Inst
Send_Cycles_end == 1
receive_full_byte == 1
bit_finish == 0
Send_PC_end == 0
Send_Cycles
Send_PC_end == 1
Send_PC
HALT == 1
Sending_Inst
Send_Cycles_end == 0
HALT == 0
Continue
bit_finish == 1
1 ciclo
en_change_state == 1
Step_to_step2
Check_Operation
Waiting_operation
modo continue
en_change_state == 0
1 ciclo
Step_to_step
modo step_to_step

FSM

Ensamblador

Se implementó un programa en python que codifica las instrucciones en assembler a código binario, con el cual el test bench levanta esos datos y los envía uno por uno por la uart y recibidos por la debug unit para su posterior carga en la memoria de instrucción.

A continuación se muestra la salida del programa con su correspondiente codificación al código en assembler mostrado.

```
lb R5, 2(0)
lb R2, 0(0)
lb R3, 1(0)
subu R2, R2, R3
ori R4, R0, 1
beq R2, R0, 2
ori R7, R1, 4
xori R9, R1, 4
lui R6, 4
sw R4, 5(0)
slti R11, R1, 4
halt
10000000000001010000000000000010
10000000000000100000000000000000
10000000000000110000000000000001
00000000100001100010000000100011
00110100000001000000000000000001
00010000010000000000000000000010
00110100001001110000000000000100
00111000001010010000000000000100
00111100000001100000000000000100
10101100000001000000000000000101
00101000001010110000000000000100
11111100000000000000000000000000
```

Test Bench

Para la prueba del funcionamiento se probaron 3 códigos que comprueban todas las instrucciones.

Test1

En el siguiente código se prueban todas las instrucciones tipo R.

Code

```
lw R1, 0(0)
lw R2, 1(0)
sll R3, R1, R2
srl R4, R1, R2
sra R5, R1, R2
sllv R6, R1, R2
srlv R7, R1, R2
srav R8, R1, R2
addu R9, R1, R2
subu R10, R1, R2
and R11, R1, R2
or R12, R1, R2
xor R13, R1, R2
nor R14, R1, R2
slt R15, R1, R2
halt
```

Condiciones Iniciales:

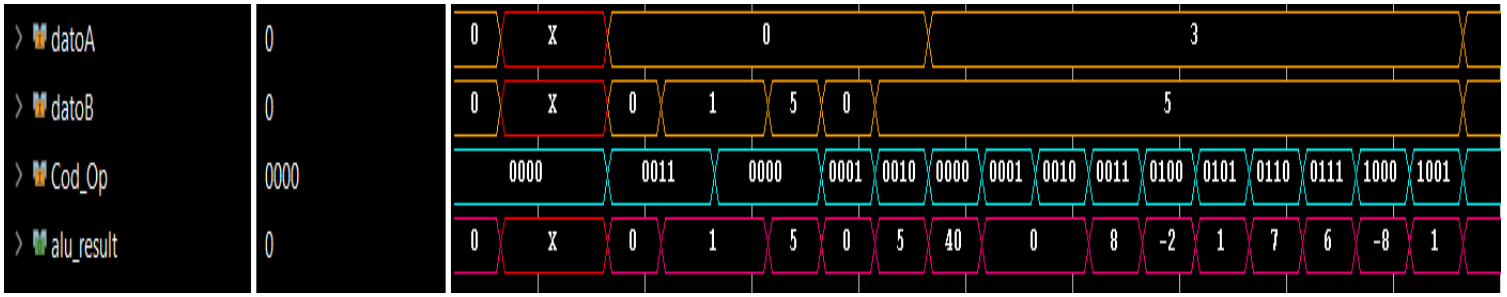
- R1: 3
- R2: 2

Resultados de Ejecución

La siguiente información son los datos de los registros enviados a través de la debug unit.

```
DATO RECBIDO R[ 0]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 1]:           3 00000000000000000000000000000011 00000003
DATO RECBIDO R[ 2]:           5 00000000000000000000000000000101 00000005
DATO RECBIDO R[ 3]:           5 00000000000000000000000000000101 00000005
DATO RECBIDO R[ 4]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 5]:           5 00000000000000000000000000000101 00000005
DATO RECBIDO R[ 6]:          40 00000000000000000000000000101000 00000028
DATO RECBIDO R[ 7]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 8]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 9]:           8 00000000000000000000000000001000 00000008
DATO RECBIDO R[10]:          -2 11111111111111111111111111111110 fffffffe
DATO RECBIDO R[11]:           1 00000000000000000000000000000001 00000001
DATO RECBIDO R[12]:           7 00000000000000000000000000000111 00000007
DATO RECBIDO R[13]:           6 00000000000000000000000000000110 00000006
DATO RECBIDO R[14]:          -8 11111111111111111111111111111000 fffffff8
DATO RECBIDO R[15]:           1 00000000000000000000000000000001 00000001
```

La siguiente imagen muestra el waveform con las entradas a la ALU, código de operación y resultado.



Se ve claramente el funcionamiento, por ejemplo la instrucción **addu R9, R1, R2**, dando como resultado un 8.

Test2

En el siguiente código se prueban todas las instrucciones tipo I.

Code

```
lw R1, 0(0)
lh R2, 0(0)
lb R3, 0(0)
lhu R4, 0(0)
lbu R5, 0(0)
sw R1, 0(0)
sh R1, 1(0)
sb R1, 2(0)
addi R6, R1, 4
andi R7, R1, 4
ori R8, R1, 4
xori R9, R1, 4
lui R10, 4
slti R11, R1, 4
halt
```

Condiciones Iniciales

Registros todos nulos.

Memoria

```
RAM[0] = 32'b00000000111100000000110000000011
RAM[1] = 32'b00000000111100000000001100000001
RAM[2] = 32'b00000000111100000001110000010001
RAM[3] = 32'b00000000111100000000000000000011
RAM[4] = 32'b00000000000000000000000000000100
RAM[5] = 32'b00000000000000000000000000000101
```

Resultado de ejecución

En la imagen de abajo se muestran los valores de cada registro hasta el 11 con su correspondiente valor de acuerdo a las instrucciones de arriba ejecutadas.

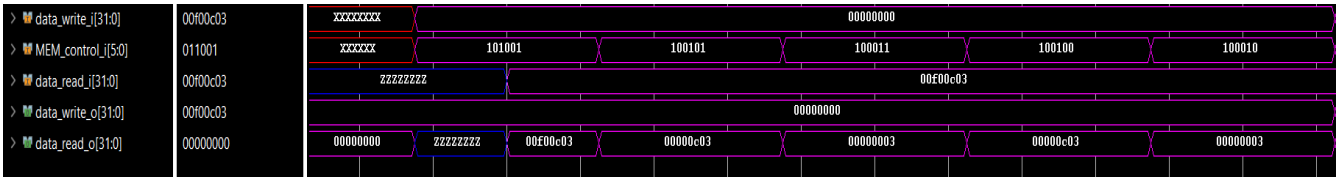
```
DATO RECBIDO R[ 0]:          0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 1]:   15731715 00000000111100000000110000000011 00f00c03
DATO RECBIDO R[ 2]:       3075 00000000000000000000110000000011 00000c03
DATO RECBIDO R[ 3]:          3 00000000000000000000000000000011 00000003
DATO RECBIDO R[ 4]:       3075 00000000000000000000110000000011 00000c03
DATO RECBIDO R[ 5]:          3 00000000000000000000000000000011 00000003
DATO RECBIDO R[ 6]:   15731719 00000000111100000000110000000111 00f00c07
DATO RECBIDO R[ 7]:          0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 8]:   15731719 00000000111100000000110000000111 00f00c07
DATO RECBIDO R[ 9]:   15731719 00000000111100000000110000000111 00f00c07
DATO RECBIDO R[10]:     262144 00000000000001000000000000000000 00040000
DATO RECBIDO R[11]:          0 00000000000000000000000000000000 00000000
```

Otro resultado importante es la memoria ya que en estos tipos de instrucciones se encuentran la carga y almacenamiento de datos.

```
MEMORIA[  0]: 00000000111100000000110000000011
MEMORIA[  1]: 00000000000000000000110000000011
MEMORIA[  2]: 00000000000000000000000000000011
MEMORIA[  3]: 00000000111100000000000000000011
MEMORIA[  4]: 00000000000000000000000000000100
```

En formato waveform

Instrucción Load con todos sus sabores.



Instrucción Store

Test3

En el siguiente código se prueban todas las instrucciones de saltos condicionales e incondicionales.

Code

```
lb R5, 2(0)
lb R2, 0(0)
lb R3, 1(0)
subu R2, R2, R3
ori R4, R0, 1
beq R2, R0, 2
bne R2, R0, -5
ori R4, R0, 1
andi R4, R0, 1
j 1
ori R4, R0, 1
ori R4, R0, 1
jalr R31, R5
jal 1
ori R4, R0, 15
halt
ori R4, R0, 15
ori R4, R0, 15
ori R4, R0, 15
ori R4, R0, 15
jr R31
ori R4, R0, 15
jr R1
```

Condiciones Iniciales

Registros

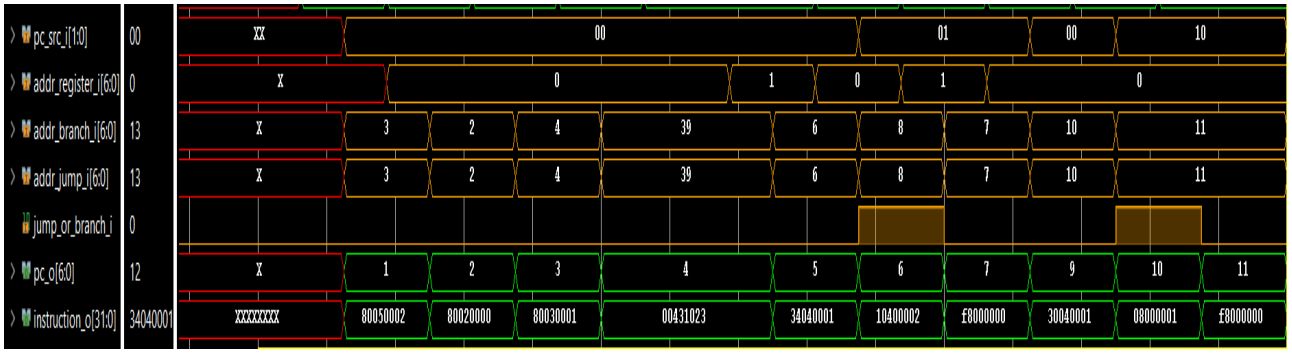
```
R2: 1
R3: 1
R5: 17
```

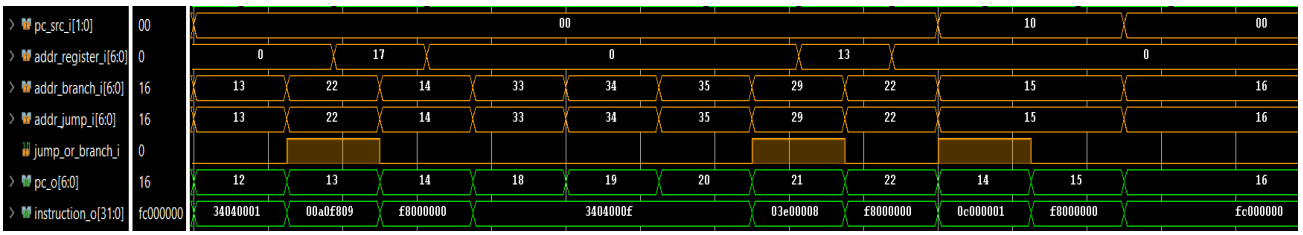
Resultado de ejecución

```
DATO RECBIDO R[ 0]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 1]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 2]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 3]:           1 00000000000000000000000000000001 00000001
DATO RECBIDO R[ 4]:          15 00000000000000000000000000001111 0000000f
DATO RECBIDO R[ 5]:          17 00000000000000000000000000010001 00000011
DATO RECBIDO R[ 6]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 7]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 8]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[ 9]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[10]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[11]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[12]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[13]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[14]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[15]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[16]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[17]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[18]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[19]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[20]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[21]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[22]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[23]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[24]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[25]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[26]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[27]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[28]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[29]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[30]:           0 00000000000000000000000000000000 00000000
DATO RECBIDO R[31]:          14 00000000000000000000000000001110 0000000e
```

```
MEMORIA[  0]: 00000000111100000000110000000001
MEMORIA[  1]: 00000000111100000000001100000001
MEMORIA[  2]: 00000000111100000001110000010001
MEMORIA[  3]: 00000000111100000000000000000011
MEMORIA[  4]: 00000000000000000000000000000100
MEMORIA[  5]: 00000000000000000000000000000101
```

Resultado en formato waveform





Análisis de timing

Para este apartado se utilizó la herramienta de reporte de timing que ofrece vivado. Se sintetizó el proyecto con varias frecuencias para ver cual es el límite máximo soportado por el pipeline.

A Continuación se presentan los reportes devueltos por la herramienta.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-1,455 ns	Worst Hold Slack (WHS):	0,137 ns	Worst Pulse Width Slack (WPWS):	3,000 ns
Total Negative Slack (TNS):	-27,509 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	31	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2113	Total Number of Endpoints:	2113	Total Number of Endpoints:	850

Timing constraints are not met.

Clock: 65 MHZ

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	-1.457	9	10	61	debug_unit/state_reg[3]/C	debug_unit/cont_byte_reg[0]/CE	8.673	1.743	6.930	7.7
Path 2	-1.457	9	10	61	debug_unit/state_reg[3]/C	debug_unit/cont_byte_reg[1]/CE	8.673	1.743	6.930	7.7
Path 3	-1.457	9	10	61	debug_unit/state_reg[3]/C	debug_unit/cont_byte_reg[2]/CE	8.673	1.743	6.930	7.7
Path 4	-1.457	9	10	61	debug_unit/state_reg[3]/C	debug_unit/cont_byte_reg[3]/CE	8.673	1.743	6.930	7.7
Path 5	-1.457	9	10	61	debug_unit/state_reg[3]/C	debug_unit/cont_byte_reg[4]/CE	8.673	1.743	6.930	7.7
Path 6	-1.457	9	10	61	debug_unit/state_reg[3]/C	debug_unit/cont_byte_reg[5]/CE	8.673	1.743	6.930	7.7
Path 7	-1.457	9	10	61	debug_unit/state_reg[3]/C	debug_unit/cont_byte_reg[6]/CE	8.673	1.743	6.930	7.7

Camino crítico

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0,816 ns	Worst Hold Slack (WHS):	0,137 ns	Worst Pulse Width Slack (WPWS):	3,000 ns
Total Negative Slack (TNS):	-11,249 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	24	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2113	Total Number of Endpoints:	2113	Total Number of Endpoints:	850

Timing constraints are not met.

Clock: 60 MHZ

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0,048 ns	Worst Hold Slack (WHS):	0,137 ns	Worst Pulse Width Slack (WPWS):	3,000 ns
Total Negative Slack (TNS):	-0,388 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	8	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2113	Total Number of Endpoints:	2113	Total Number of Endpoints:	850

Timing constraints are not met.

Clock: 55 MHZ

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,901 ns	Worst Hold Slack (WHS):	0,137 ns	Worst Pulse Width Slack (WPWS):	3,000 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2113	Total Number of Endpoints:	2113	Total Number of Endpoints:	850

All user specified timing constraints are met.

Clock: 50 MHZ

Dada esta información se obtuvo una frecuencia máxima de 50 MHZ.

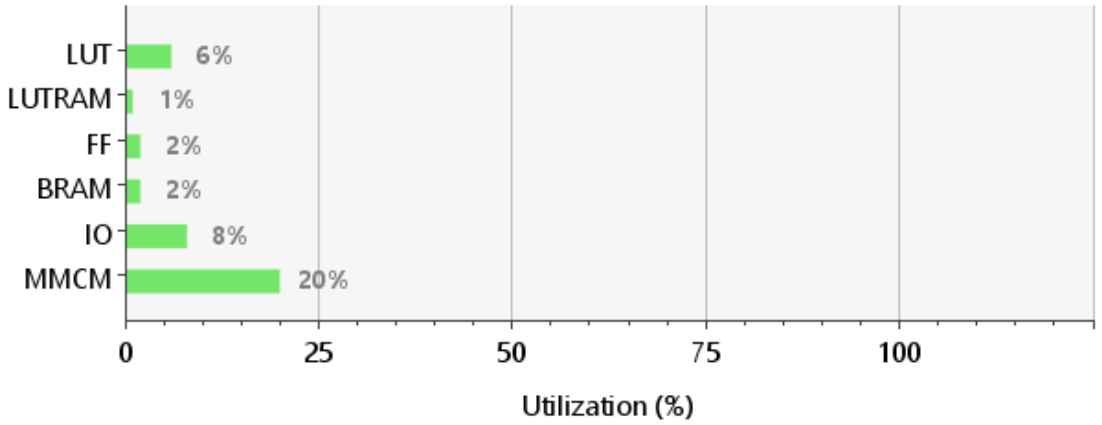
Conclusión

Una vez terminado el práctico, a través de la herramienta Vivado se obtuvieron las especificaciones en la utilización de los recursos de la FPGA y en la potencia que consume dicho circuito instanciado.

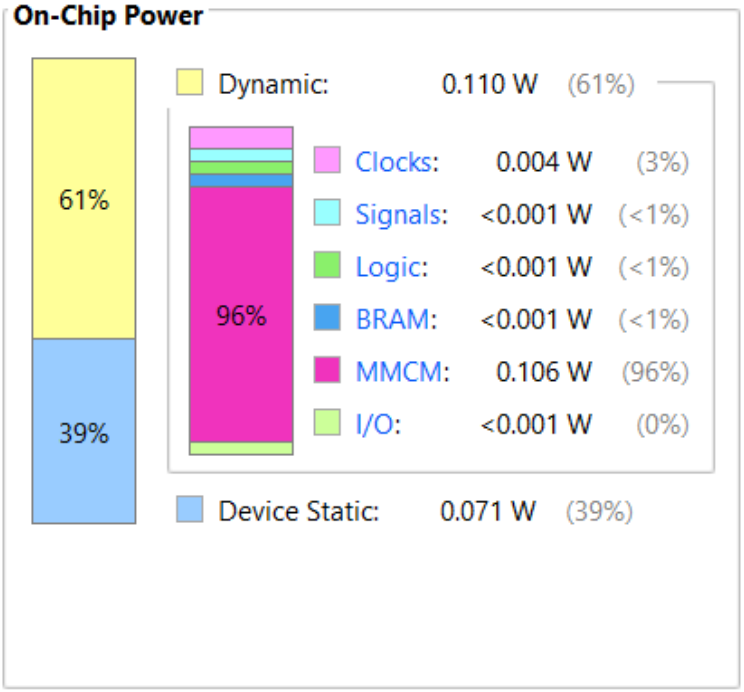
Se logró implementar y probar todas las instrucciones, verificando que no haya conflictos entre todas las posibles acciones que se pueden tomar. Además, la ejecución del MIPS a una frecuencia máxima de 50 MHz.

Recursos utilizados por el proyecto

Resource	Utilization	Available	Utilization %
LUT	1299	20800	6.25
LUTRAM	48	9600	0.50
FF	744	41600	1.79
BRAM	1	50	2.00
IO	8	106	7.55
MMCM	1	5	20.00



Consumo de potencia



Total On-Chip Power:	**0.18 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25,9°C**
Thermal Margin:	59,1°C (11,8 W)
Effective ϑJA:	5,0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium