Cátedra de Arquitectura de Computadoras

Trabajo Práctico Final
Procesador MIPS simplificado con pipeline

Carrizo, Aixa Mariel
Piñero, Tomás Santiago
13 de Agosto de 2021

Índice

Índice **1**

1. Enunciado **2**
1.1. Requerimientos 2

2. Desarrollo **5**
2.1. Diseño 5
2.1.1. MIPS 6
2.1.2. Interfaces 8
2.1.3. UART 10
2.2. Resultados 10
2.3. Compilador 12

3. Cálculo de frecuencia máxima **12**

4. Conclusiones **12**

Referencias **13**

1. Enunciado

El objetivo de este trabajo es implementar en FPGA, programando en Verilog, un procesador MIPS simplificado con *pipeline*.

1.1. Requerimientos

Los requerimientos del trabajo son los siguientes:

- Se debe implementar el procesador MIPS segmentado en las siguientes etapas:
  1. *Instruction Fetch*: busca la instrucción en la memoria de programa.
  2. *Instruction Decode*: decodifica la instrucción y lee los registros.
  3. *Execute*: ejecuta la instrucción.
  4. *Mempry Access*: lee o escribe la memoria de datos.
  5. *Write back*: escribe los resultados en los registros.
- Se deben implementar las instrucciones de la Tabla **1**;
- 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:
  1. Contenido de los 32 registros;
  2. Contador de programa (PC);
  3. Contenido de la memoria de datos usada;
  4. Cantidad de ciclos de reloj desde el inicio.
- Una vez cargado el programa a ejecutar, el procesador debe permitir dos modos de operación:
  1. 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;
  2. 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 hasta terminar el programa.

- El *clock* del sistema debe crearse utilizando el *ip-core* correspondiete.
- Se debe mostrar el reporte de *timing* con la frecuencia máxima de reloj soportada.
- Se debe validar el desarrollo por medio de *Test Bench*.

Tipo	Instrucción	Función
R	ADDU	rd = rs + rt
	AND	rd = rs & rt
	NOR	rd = rs ∼| rt
	OR	rd = rs | rt
	SLL	rd = rt << sa
	SLLV	rd = rt << rs
	SLT	rd = (rs < rt)
	SRA	rd = rt >> sa
	SRAV	rd = rt >> rs
	SRL	rd = u(rt >> sa)
	SRLV	rd = u(rt >> rs)
	SUBU	rd = rs - rt
	XOR	rd = rs ⊕ rt
I	ADDI	rt = rs + immediate
	ANDI	rt = rs & immediate
	BEQ	if(rs == rt) branch
	BNE	if(rs != rt) branch
	J	pc = instr_index
	JAL	ra = pc + 8 pc = instr_index
	LB	rt = mem[base+offset] & 0xFF
	LBU	rt = u(mem[base+offset]) & 0xFF
	LH	rt = mem[base+offset] & 0xFFFF
	LHU	rt = u(mem[base+offset]) & 0xFFFF
	LUI	rt = immediate << 16
	LW	rt = mem[base+offset]
	LWU	rt = u(mem[base+offset])
	ORI	rt = rs | immediate
	SB	mem[base+offset] = (reg[rt] & 0xFF)
	SH	mem[base+offset] = (reg[rt] & 0xFFFF)
	SLTI	rt = (rs < immediate)
	SW	mem[base+offset] = reg[rt]
	XORI	rt = rs ⊕ immediate
J	JALR	rd = pc + 8 pc = reg[rs]
	J	pc = reg[rs]

Tabla 1: Instrucciones a implementar.

2. Desarrollo

El desarrollo de este informe aborda la arquitectura del proyecto desde una perspectiva *top-down*, es decir, desde lo general a lo específico. Así, se facilita la lectura y entendimiento del proceso llevado a cabo.

2.1. Diseño

A continuación se muestra el módulo *top*, que está conformado por:

- El procesador MIPS;
- El UART;
- Interfaces entre los dos módulos anteriores.

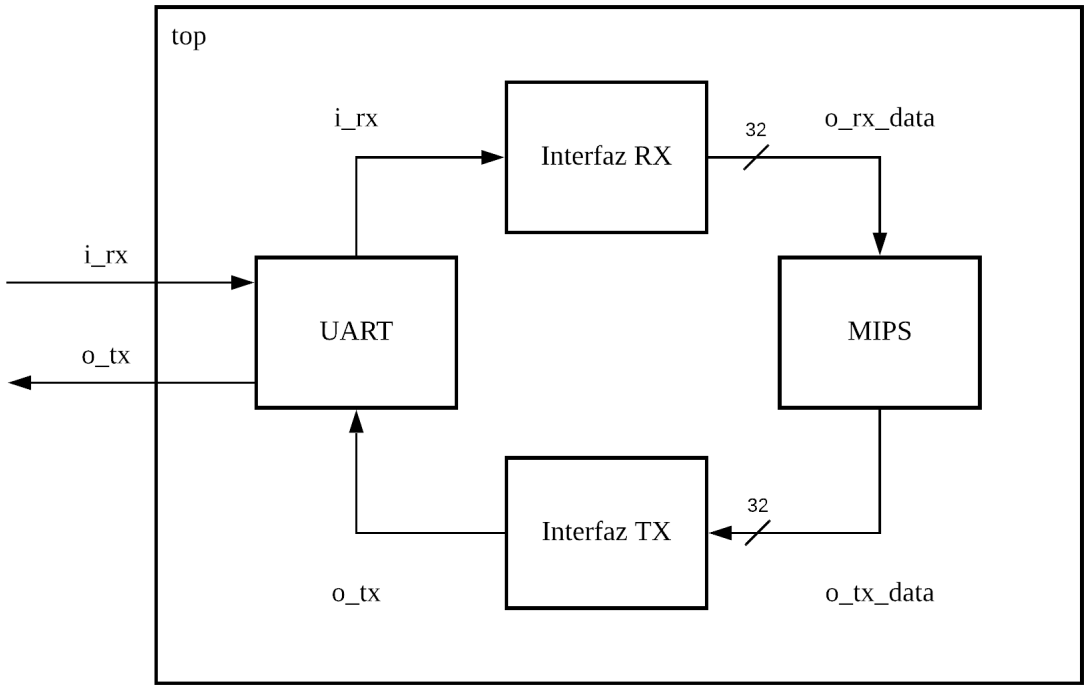

Figura 1: Arquitectura simplificada del módulo *top*.

En las siguientes secciones se explica la composición de estos componentes detalladamente.

2.1.1. MIPS

Para el diseño del procesador, se siguió el capítulo 4 del libro *Computer Organization and Design: The Hardware/Software Interface*[1]. Dicho capítulo comienza con el diseño de un MIPS monociclo y termina con un MIPS segmentado en 5 etapas junto con la unidad de detección de riesgos, la unidad de cortocircuito y el manejo de las excepciones.

Se siguió el libro paso a paso hasta tener el procesador funcionando correctamente sin el manejo de las excepciones.

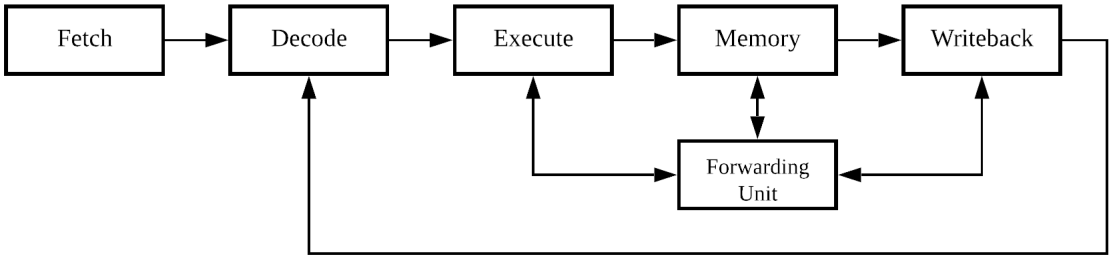

Figura 2: Arquitectura simplificada del módulo *top_pipeline*.

Como se puede observar en la Figura **2**, las etapas del *pipeline* implementado son las siguientes:

- *Fetch*: Se realiza la lectura de instrucción y actualización del *program counter* (PC);
- *Decode*: Se leen los registros mientras se decodifica la instrucción y se realiza la detección de los riesgos estructurales que puedan existir;
- *Execute*: Se realiza el cálculo de una operación o de una dirección, según corresponda;
- *Memory*: Se accede o escribe un operando en la memoria de datos;
- *Writeback*: Se escribe el resultado en un registro.

También se incluye la unidad de cortocircuito, que utiliza *buffers* internos en lugar de esperar que un registro determinado actualice su contenido, si existe un riesgo de datos (*data hazard*).

A continuación, se explica brevemente cómo está estructurada cada una de las etapas mencionadas.

Fetch

Dentro de este módulo se encuentra la memoria de programa, que utiliza 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 actualización del PC se realiza en función de las señales de control que indican si el destino es un salto (*i_pc_src*, si se debe realizar un *stall* o se terminó el programa con un *halt*. También se tiene en cuenta el modo de ejecución recibido.

Decode

En este módulo se realiza la decodificación de la instrucción recibida de la etapa anterior y la lectura/escritura de los registros.

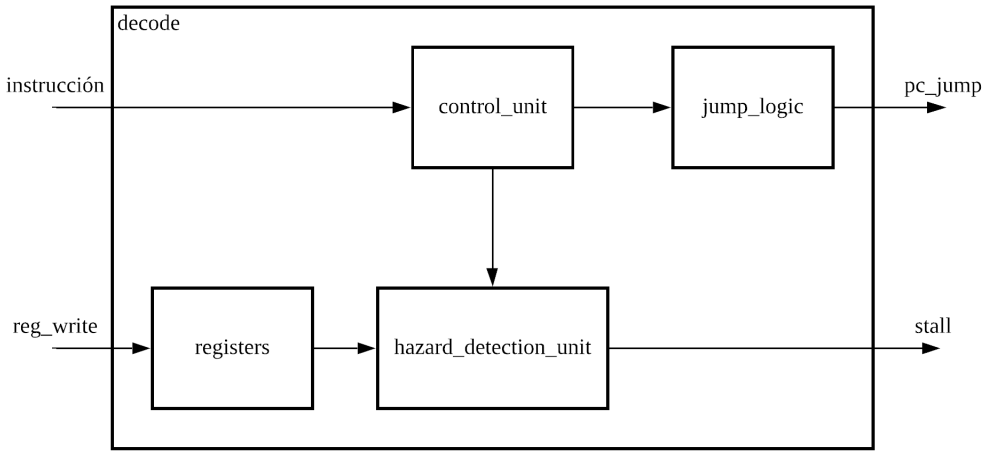

Figura 3: Arquitectura simplificada del módulo *decode*.

Como se puede ver en la Figura superior, el módulo *decode* está formado por:

- *registers*: módulo donde se encuentran los 32 registros del procesador;
- *control_unit*: módulo que se encarga de asignar los valores correspondientes de las señales de control según la instrucción recibida;
- *jump_logic*: módulo que establece la dirección de memoria de la siguiente instrucción en caso de recibir un salto no condicional;
- *hazard_detection_unit*: módulo que recibe la señal de control *mem_read* y los registros involucrados en la instrucción para detectar los riesgos estructurales.

Execute

En esta etapa se encuentra la ALU y el módulo "*alu_ctrl*". Para decidir los valores que se utilizan en la operación a realizar, se emplean 4 multiplexores:

1. *Dato A*: elige entre el valor dentro del registro, el resultado de la operación anterior y el valor leído de la memoria de datos;
2. *Dato B según la unidad de cortocircuito*: elige igual que el ítem anterior;
3. *Dato B a utilizar*: elige entre el valor *sa* o el valor elegido por el ítem anterior, dependiendo si la instrucción es un *shift*;
4. Valor del registro en el que se escribirá el resultado;

Memory

En esta etapa se encuentra el módulo que contiene la lógica para determinar los saltos condicionales (`branch_logic`)y la memoria de datos (`data_memory`). Para esta última se utilizó la misma plantilla que para la memoria de programa.

Respecto a la ejecución los saltos condicionales, se decidió suponer que el salto no se toma, ya que es menos costoso vaciar el *pipeline* que volver atrás el PC en caso de asumir que el salto debe tomarse. Teniendo esto en cuenta, el módulo que se encarga de lo lógica para los saltos condicionales toma la decisión de vaciar o no el *pipeline* basándose en las señales de control `i_zero` y `i_branch`.

Writeback

Durante esta etapa se decide el valor que se debe escribir, si el que proviene de la ALU o de la memoria de datos. También decide el valor que debe escribirse en el registro de retorno cuando la instrucción ejecutada es un `JAL` o `JALR` e indica al módulo de contador de ciclos que deje de contar.

2.1.2. Interfaces

La comunicación entre el UART y el procesador se realiza con dos interfaces: una para recepción y otra para transmisión. Cada una de estas interfaces es una máquina de estados que se encarga de concatenar o dividir los bits según el ancho de palabra soportado por el módulo destino (8 bits para UART, 32 bits para MIPS).

Recepción

La interfaz de recepción concatena los bits recibidos hasta llegar a los 32 bits para enviárselos al procesador.

Los estados que se pueden observar en la Figura **4** son:

1. `IDLE`: es el estado inicial. Cuando se encuentra en este estado se pasa directamente al estado siguiente.
2. `INSTRUCTIONS` (INSTR): todos los datos recibidos desde el inicio del sistema son interpretados como instrucciones, las cuales se guardan en `programa.mem` para su posterior lectura y ejecución. Al detectar la instrucción `HALT` (`0xFFFFFFFF`), se pasa al siguiente estado.

3. `EXEC_MODE` (EXMD): una vez recibidas las instrucciones a ejecutar, se recibe el modo de ejecución, que puede ser continuo (8'b0) o paso a paso (8'b1). Al terminar de recibir el dato, se pasa al siguiente estado.

4. `STEP`: es el estado final. Si es modo de ejecución continua, no se tiene en cuenta esta señal. En cambio, si es paso a paso, se ejecuta un ciclo y se espera que el usuario ingrese nuevamente la señal de step (8'b1).

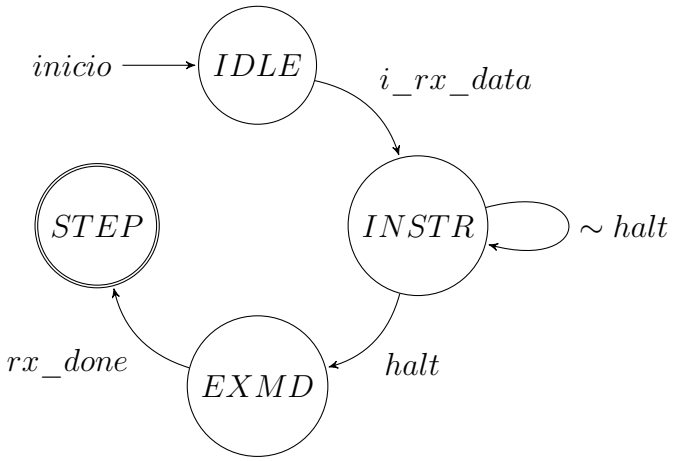

Figura 4: Diagrama de estados de la interfaz.

Transmisión

La interfaz de transmisión divide la información que se envía desde el procesador (32 bits) a 8 bits. La información que se envía es la pedida en el enunciado (ver **1**):

- El valor del *Program Counter* (PC);
- El valor de cada uno de los registros (32 en total);
- El último valor almacenado en memoria;
- La cantidad de ciclos ejecutados.

El envío de información depende del modo de ejecución seleccionado.

- *Modo continuo*: realiza el envío cuando se detecta que la instrucción `HALT` fue ejecutada.
- *Modo paso a paso*: realiza el envío al finalizar la ejecución de un paso (un ciclo de reloj).

Los estados de la interfaz se pueden ver en la Figura **5** y son los siguientes:

1. `IDLE`: mientras no se detecte la instrucción `HALT` o la señal de paso, se queda en este estado. Caso contrario, se pasa al siguiente.

2. `PC`: en este estado toma el contenido del PC y transmite su valor. Una vez que termina, pasa al siguiente estado.

3. `REG`: realiza el envío del contenido de los 32 registros de propósito general. En este estado se toma un arreglo de 1024 elementos (32 elementos de 32 bit cada uno) y se los envía uno por uno. Ya enviados los 32 registros, se pasa al estado siguiente.

4. `MEM`: se envía el último valor leído de la memoria de datos y se pasa al estado que sigue.

5. `CNTR`: se realiza el envío de la cantidad de ciclos de reloj ejecutados hasta el momento y vuelve al estado inicial.

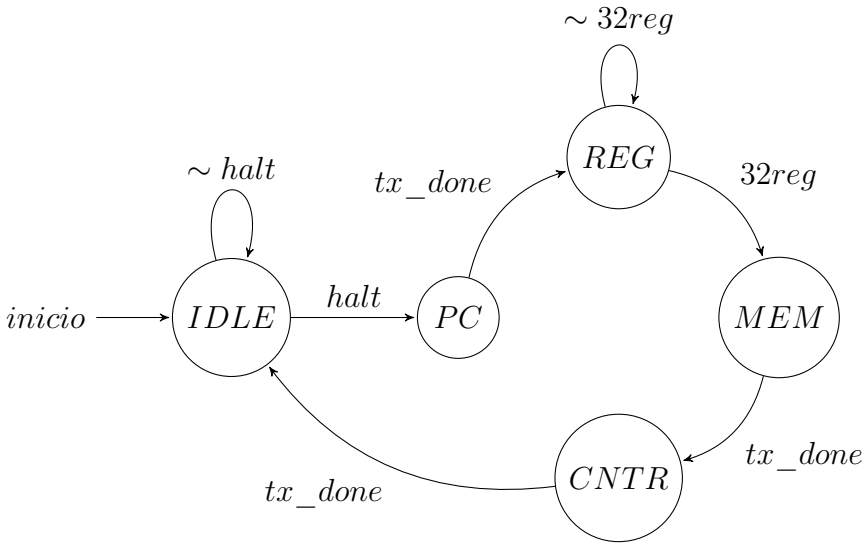

Figura 5: Diagrama de estados de la interfaz.

2.1.3. UART

El módulo UART utilizado para este trabajo es que se realizó en el Trabajo Práctico N° 2. Para saber más sobre el diseño de este módulo, dirigirse `aquí`.

2.2. Resultados

Para verificar el funcionamiento del MIPS, se escribieron 2 programas que engloban todas las instrucciones que soporta. Uno contiene los saltos no condicionales (Código **1**) y el otro los saltos condicionales (Código **2**). Cada uno de ellos permite evaluar el correcto funcionamiento de las unidades de cortocircuito y detección de riesgos.

A continuación se muestra el código de ambos programas. Cada línea tiene como comentario el valor que se debe guardar en el registro que se utiliza. También se muestran los resultados de la simulación post-síntesis de tiempo de cada uno de ellos. En el caso del primero, se elige el modo paso a paso, mientras que en el segundo, el modo continuo.

Como las imágenes de la simulación no se pueden observar en detalle, es posible consultarlas en los *links* que se encuentran debajo de cada código.

Código 1: Código fuente para saltos no condicionales.

```
1          addu $1,$5,$6        # 3
2          and  $2,$5,$6        # 0
3          nor  $3,$5,$6        # -4 fffffffc
4          j    SALTO
5  SALTO2: or   $4,$5,$6        # 3
6          sll  $7,$6,1         # 4
7          sllv $8,$6,$5        # 4
8          slt  $9,$5,$6        # 1
9          sra  $10,$6,1        # 1
10         srav $11,$6,$5       # 1
11         srl  $12,$6,1        # 1
12         jalr $16
13 SALTO:  srlv $13,$6,$5       # 1
14         subu $14,$6,$5       # 1
15         xor  $15,$5,$6       # 3
16         jal  SALTO2
17         addi $16,$5,1        # 2 (16)
18         andi $17,$5,1        # 1
19         lb   $18,10($5)      # 3
20         lbu  $19,11($5)      # 3
21         jr   $24
22         lh   $20,12($5)      # 3 (21)
23         lhu  $21,13($5)      # 3
24         lui  $22,5
25         lw   $23,14($5)      # 3 (25)
26         lui  $22,5
27         lw   $24,15($5)      # 3
28         lh   $20,12($5)      # 3 (21)
29         lhu  $21,13($5)      # 3 STALL
30         or   $25,$21,$5
31         halt
```

Instrucciones ejecutadas: `Link a simulación de Código 1`.

Contenido de los registros luego de la ejecución: `Link a simulación de Código 1`.

Código 2: Código fuente para saltos condicionales.

```
1          beq  $0,$0,SALTO
2          ori  $25,$6,1        # 3
3          sb   $26,10($5)      # 0
4          sh   $27,11($5)      # 0
5          slti $28,$5,2        # 1
6  SALTO:  sw   $29,12($5)      # 0
7          xori $30,$5,2        # 3
8          bne  $0,$0,SALTO2
9          subu $1,$6,$5        # FORWARDING
10         and  $12,$1,$5
11         sb   $26,10($5)      # 0
12         sh   $27,11($5)      # 0
13         slti $28,$5,2        # 1
14         halt
```

`Link a simulación de Código 2`.

2.3. Compilador

El compilador se realizo en *Perl*. Su código se puede ver en el repositorio de la materia (ver `aquí`).

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 `4.924 [ns]` y el tiempo de *hold* es de `0.088 [ns]`.

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

$$\begin{aligned} t_{clock} &= t_{setup} + t_{hold} \\ t_{clock} &= 4{,}924 + 0{,}088 \\ t_{clock} &= 5{,}012\,[ns] \end{aligned}$$

Finalmente, el periodo del *clock* es:

$$f_{clock} = 199{,}52\,[MHz] \tag{1}$$

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	4.924 ns	Worst Hold Slack (WHS):	0.088 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	7955	Total Number of Endpoints:	7955

All user specified timing constraints are met.

Figura 6: Resultados del análisis de tiempo.

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] David A. Patterson, John L. Hennessy, *Computer Organization and Design: The hardware/software interface*, Fifth Edition.

[2] Charles Price, *MIPS IV Instruction Set*, Revision 3.2, Septiembre 1995.