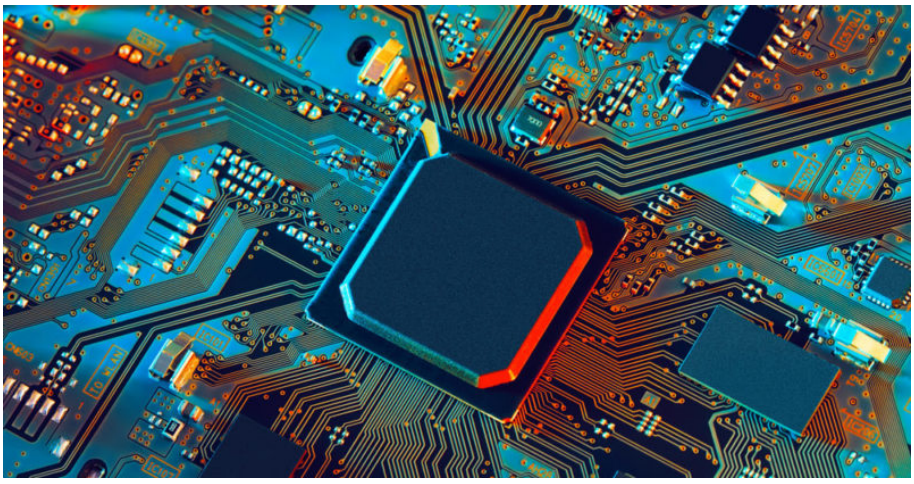
Universidad de La Habana

Facultad de Matemática y Computación

Arquitectura de Computadoras

S-MIPS

Proyecto Final



Curso 2024

Índice

1. Introducción

El proyecto consiste en diseñar en *Logisim* un procesador que implemente la arquitectura de juegos de instrucciones S-MIPS (Simplified-MIPS).

S-MIPS es una arquitectura de 32 bits. El procesador tiene 32 registros de propósito general de 32 bits, nombrados de R_0 a R_{31}. Por convenio, R_0 siempre tiene el valor constante 0 independientemente de las operaciones que se realicen sobre él y R_{31} (también denominado SP) actúa como puntero de la pila (*Stack Pointer*). Cuenta con dos registros adicionales Hi y Lo donde se almacena el resultado de la división y la multiplicación. También se debe disponer de un registro (denominado PC) para el *Program Counter*, el cual almacenará la dirección en la memoria de la próxima instrucción a leer de la *RAM*.

El procesador debe implementarse en el módulo *S-MIPS* del circuito *S-MIPS Board* que se brinda. No se debe modificar el circuito *RAM* ni el circuito *S-MIPS Board*. Durante la evaluación se utilizarán estos circuitos tal y como se entregó. Además queda prohibido utilizar las componentes *RAM* y *ROM* que proporciona *Logisim*.

2. Instrucciones y sus formatos

2.1. Notas generales.

- R_s , R_t y R_d especifican registros de propósito general.
- Un elemento entre corchetes ([ ]) indica *"el contenido de"*. Por ejemplo $[R_3] + [R_{22}]$ se refiere a la suma de los valores almacenados en los registros R_3 y R_{22}.
- $[PC]$ especifica la dirección de la instrucción en ejecución. Por ejemplo, saltar a la próxima instrucción es $[PC] \leftarrow [PC] + 4$
- I se refiere a los bits de la instrucción y el subíndice indica a cuáles de estos bits se refiere. $[I_{15,...,0}]$ se refiere al contenido de los 16 primeros bits de la instrucción, que en el caso de las instrucciones de tipo I es la constante.
- || indica la concatenación de bits.
- Los sobre-índices indican la repetición de un valor binario. 0^7 se refiere a: $\texttt{000 0000}_2$.
- $M\{I\}$ se refiere a los 32 bits almacenados en la memoria RAM en la dirección divisible por 4 más cercana al valor de I, es decir, en la dirección $I - I\,\%4$.

Las instrucciones de S-MIPS tienen una longitud constante de 32 bits. Hay 3 formatos de instrucciones distintos:

Tipo R:

Op-code	Rs	Rt	Rd	x	Func-code

Tipo I:

Op-code	Rs	Rt	Constante (complemento a2)

Tipo J:

Op-code	Destino-salto

Además se considera como el bit menos significativo (o bit 0) al bit más a la derecha, y como el bit más significativo (o bit 31) al bit más a la izquierda.

3. Instrucciones Aritméticas

3.1. Addition: add.

Tipo R:	00 0000	R_s	R_t	R_d	0 0000	10 0000

Efectos de la instrucción:

$R_d \leftarrow [R_s] + [R_t]$
$PC \leftarrow [PC] + 4$

Ensamblador:

`add` R_d, R_s, R_t

3.2. Substract: sub.

Tipo R:	00 0000	R_s	R_t	R_d	0 0000	10 0010

Efectos de la instrucción:

$R_d \leftarrow [R_s] - [R_t]$
$PC \leftarrow [PC] + 4$

Ensamblador:

`sub` R_d, R_s, R_t

3.3. Multiply: mult.

Tipo R:	00 0000	R_s	R_t	0 0000	0 0000	01 1000

Efectos de la instrucción:

$Hi \mid\mid Lo \leftarrow [R_s] * [R_t]$
$PC \leftarrow [PC] + 4$

Ensamblador:

`mult` R_s, R_t

3.4. Unsigned multiply: mulu.

Idéntica a la instrucción `mult` excepto:

- Func-code = 01 1001
- El contenido de R_s y R_t se considera como enteros sin signo.

3.5. Divide: div.

Tipo R:	00 0000	R_s	R_t	0 0000	0 0000	01 1010

Efectos de la instrucción:

$Lo \leftarrow [R_s]/[R_t]$
$Hi \leftarrow [R_s] mod [R_t]$
$PC \leftarrow [PC] + 4$

Ensamblador:

`div` R_s, R_t

3.6. Unsigned divide: divu.

Idéntica a la instrucción `div` excepto:

- Func-code = 01 1011
- El contenido de R_s y R_t se considera como enteros sin signo.

3.7. Addition inmediate: addi.

Tipo I:	00 1000	R_s	R_t	constante

Efectos de la instrucción:

$R_t \leftarrow [R_s] + ([I_{15}]^{16} \mid\mid [I_{15,\ldots,0}])$
$PC \leftarrow [PC] + 4$

Ensamblador:

`addi` R_t`,` R_s`, constante`

Ejemplo: `addi` R_3`,` R_8`, 34`

4. Instrucciones Lógicas

4.1. Logical and: and.

Tipo R:	00 0000	R_s	R_t	R_d	0 0000	10 0100

Efectos de la instrucción:

$R_d \leftarrow [R_s]\ and\ [R_t]$
$PC \leftarrow [PC] + 4$

Ensamblador:

`and` R_d`,` R_s`,` R_t

4.2. Logical or, nor, xor: or, nor, xor.

Idéntica a la instrucción `and` excepto:

- Func-code = 10 0101 para la instrucción `or`,
- Func-code = 10 0111 para la instrucción `nor`,
- Func-code = 10 1000 para la instrucción `xor`.

4.3. Logical and immediate: andi.

Tipo I:	00 1100	R_s	R_t	constante

Efectos de la instrucción:

$R_t \leftarrow [R_s]\ and\ (0^{16} \mid\mid [I_{15,\ldots,0}])$
$PC \leftarrow [PC] + 4$

Ensamblador:

`andi` R_t`,` R_s`, constante`

4.4. Logical or immediate & xor immediate: ori, xori.

Idéntica a la instrucción `andi` excepto:

- Func-code = 00 1101 para la instrucción `ori`.
- Func-code = 00 1110 para la instrucción `xori`.

5. Instrucciones de Comparación

5.1. Set less than: slt.

Tipo R:	00 0000	R_s	R_t	R_d	0 0000	10 1010

Efectos de la instrucción:

si $[R_s] < [R_t]$ entonces $R_d \leftarrow 0^{31} \,||\, 1$
sino $R_d \leftarrow 0^{32}$
$PC \leftarrow [PC] + 4$

Ensamblador:

`slt` R_d`,` R_s`,` R_t

5.2. Unsigned set less than: sltu.

Idéntica a la instrucción `slt` excepto:

- Func-code = 10 1011
- El contenido de R_s y R_t se considera como enteros sin signo.

5.3. Set less than immediate: slti.

Tipo I:	00 1010	R_s	R_t	constante

Efectos de la instrucción:

si $[R_s] < ([I_{15}]^{16} \,||\, [I_{15,\ldots,0}])$ entonces $R_t \leftarrow 0^{31} \,||\, 1$
sino $R_t \leftarrow 0^{32}$
$PC \leftarrow [PC] + 4$

Ensamblador:

`slti` R_t`,` R_s`, constante`

5.4. Set less than immediate unsigned: sltiu.

Idéntica a la instrucción `slti` excepto:

- Func-code = 00 1011.
- El contenido de R_s y R_t se considera como enteros sin signo.

6. Instrucciones de Copia de Registros

6.1. Move from Hi: mfhi.

Tipo R:	00 0000	0 0000	0 0000	R_d	0 0000	01 0000

Efectos de la instrucción:

$R_d \leftarrow [Hi]$
$PC \leftarrow [PC] + 4$

Ensamblador:

`mfhi` R$_d$

6.2. Move from Lo: mflo.

Tipo R:	00 0000	0 0000	0 0000	R_d	0 0000	01 0010

Efectos de la instrucción:

$R_d \leftarrow [Lo]$
$PC \leftarrow [PC] + 4$

Ensamblador:

`mflo` R$_d$

7. Instrucciones de Salto

7.1. No operation: nop.

Tipo R:	00 0000	0 0000	0 0000	0 0000	0 0000	00 0000

Efectos de la instrucción:

$PC \leftarrow [PC] + 4$

Ensamblador:

`nop`

7.2. Branch on equal: beq.

Tipo I:	00 0100	R_s	R_t	offset

Efectos de la instrucción:

si $[R_s] == [R_t]$ entonces $PC \leftarrow [PC] + 4 + ([I_{15}]^{14} \mid\mid [I_{15,\ldots,0}] \mid\mid 0^2)$,
o sea, $PC \leftarrow [PC] + 4 + 4 * offset$
sino $PC \leftarrow [PC] + 4$

Ensamblador:

`beq` R$_s$`,` R$_t$`, offset`

7.3. Branch on not equal: bne.

Tipo I:	00 0101	R_s	R_t	offset

Efectos de la instrucción:

si $[R_s] <> [R_t]$ entonces $PC \leftarrow [PC] + 4 + ([I_{15}]^{14} \mid\mid [I_{15,\ldots,0}] \mid\mid 0^2)$,
o sea, $PC \leftarrow [PC] + 4 + 4 * offset$
sino $PC \leftarrow [PC] + 4$

Ensamblador:

`bne` R$_s$`,` R$_t$`, offset`

7.4. Branch on less than or equal zero: blez.

Tipo I:	00 0110	R_s	R_t	offset

Efectos de la instrucción:

si $[R_s] \leq 0$ entonces $PC \leftarrow [PC] + 4 + ([I_{15}]^{14} \mid\mid [I_{15,\ldots,0}] \mid\mid 0^2)$,
o sea, $PC \leftarrow [PC] + 4 + 4 * offset$
sino $PC \leftarrow [PC] + 4$

Ensamblador:

`blez` R_s`, offset`

7.5. Branch on greater than zero: bgtz.

Tipo I:	00 0111	R_s	R_t	offset

Efectos de la instrucción:

si $[R_s] > 0$ entonces $PC \leftarrow [PC] + 4 + ([I_{15}]^{14} \mid\mid [I_{15,\ldots,0}] \mid\mid 0^2)$,
o sea, $PC \leftarrow [PC] + 4 + 4 * offset$
sino $PC \leftarrow [PC] + 4$

Ensamblador:

`bgtz` R_s`, offset`

7.6. Branch on less than zero: bltz.

Tipo I:	00 0001	R_s	R_t	offset

Efectos de la instrucción:

si $[R_s] < 0$ entonces $PC \leftarrow [PC] + 4 + ([I_{15}]^{14} \mid\mid [I_{15,\ldots,0}] \mid\mid 0^2)$,
o sea, $PC \leftarrow [PC] + 4 + 4 * offset$
sino $PC \leftarrow [PC] + 4$

Ensamblador:

`bltz` R_s`, offset`

7.7. Jump: j.

Tipo J:	00 0010	destino

Efectos de la instrucción:

$PC \leftarrow ([PC_{31,\ldots,28}] \mid\mid [I_{25,\ldots,0}] \mid\mid 0^2)$

Ensamblador:

`j destino`

7.8. Jump register: jr.

Tipo R:	00 0000	R_s	0 0000	0 0000	0 0000	00 1000

Efectos de la instrucción:

$PC \leftarrow [R_s]$

Ensamblador:

`jr` R_s

8. Instrucciones de Memoria

8.1. Load word: lw.

Tipo I:	10 0011	R_s	R_t	offset

Efectos de la instrucción:

$R_t \leftarrow M\{[R_s] + ([I_{15}]^{16} \mid\mid [I_{15,\ldots,0}])\}$
$PC \leftarrow [PC] + 4$

Ensamblador:

`lw` R_t`, offset(`R_s`)`

Ejemplo: `lw` R_3`, 16(`R_0`)`

8.2. Store word: sw.

Tipo I:	10 1011	R_s	R_t	offset

Efectos de la instrucción:

$M\{[R_s] + ([I_{15}]^{16} \mid\mid [I_{15,\ldots,0}])\} \leftarrow [R_t]$
$PC \leftarrow [PC] + 4$

Ensamblador:

`sw` R_t`, offset(`R_s`)`

8.3. Pop from stack: pop.

Tipo R:	11 1000	0 0000	0 0000	R_d	0 0000	00 0000

Efectos de la instrucción:

$R_d \leftarrow M\{[SP]\}$
$SP \leftarrow [SP] + 4$
$PC \leftarrow [PC] + 4$

Ensamblador:

`pop` R_d

8.4. Push to stack: push.

Tipo R:	11 1000	R_s	0 0000	0 0000	0 0000	00 0001

Efectos de la instrucción:

$SP \leftarrow [SP] - 4$
$M\{[SP]\} \leftarrow R_s$
$PC \leftarrow [PC] + 4$

Ensamblador:

`push` R_s

9. Instrucciones especiales

Las siguientes instrucciones no son realistas, pero son necesarias para simular una entrada de teclado y una terminal sin complicar la interfaz de simulación.

9.1. Halt program: halt.

Tipo R:	11 1111	0 0000	0 0000	0 0000	0 0000	11 1111

Efectos de la instrucción:

Detiene la simulación del programa actual.

Ensamblador:

`halt`

9.2. Write to terminal: tty.

Tipo R:	11 1111	R_s	0 0000	0 0000	0 0000	00 0001

Efectos de la instrucción:

Envía un caracter a la pantalla conectada al procesador (`TTY`). La pantalla es un circuito síncrono. Para enviar un caracter, se colocan los 7 bits menos significativos de R_s en la salida `TTY DATA` del procesador, se activa la salida `TTY ENABLE`, y en el próximo ciclo la pantalla mostrará el caracter *ASCII* correspondiente a los 7 bits de `TTY DATA`.

Ensamblador:

`tty` R_s

9.3. Set random: rnd.

Tipo R:	11 1111	0 0000	0 0000	R_d	0 0000	00 0010

Efectos de la instrucción:

Almacena en R_d un número aleatorio.

Ensamblador:

`rnd` R_d

9.4. Set key: kbd.

Tipo R:	11 1111	0 0000	0 0000	R_d	0 0000	00 0010

Efectos de la instrucción:

> Esta instrucción lee un caracter del teclado y lo almacena en R_d. La entrada `KBD AVAILABLE` del procesador indica si hay algún caracter esperando en el *buffer* del teclado y la entrada `KBD DATA`, es un número de 7 bits que corresponde al código *ASCII* del caracter que está en la punta del *buffer*. El *buffer* del teclado funciona como una cola. Los caracteres se añaden a la cola cuando se teclea. En cada ciclo de reloj, si `KBD ENABLE` está activa, el teclado elimina el caracter que está en la punta de la cola. Si en el momento que esta instrucción se ejecuta, `KBD AVAILABLE` está desactivada, en R_d se almacena el valor -1.

Ensamblador:

```
kbd Rd
```

10. Program Counter

El valor del registro *Program Counter* (PC) representa la dirección de memoria de la próxima instrucción que se va a ejecutar. Como en S-MIPS las instrucciones ocupan 4 bytes, cada vez que el procesador ejecuta una instrucción que no sea de salto el valor de PC aumenta en 4. Las instrucciones de salto (`beq`, `bne`, `blez`,...) suman su argumento (en complemento a 2) al de PC. El argumento indica el número de instrucciones a saltarse, por tanto, es necesario multiplicar el argumento por 4 antes de sumarlo al *Program Counter*. Como resultado de esto, si un programa quisiera saltarse una instrucción, debe hacer un salto con argumento 1:

```
add R3, R0, 46
add R4, R0, 46
beq R3, R4, 1
halt
```

En este ejemplo la instrucción `halt` siempre se salta. Otro resultado de este comportamiento es que "`beq R0, R0, 0`" es lo mismo que `nop`, y "`beq R0, R0, -1`" es un ciclo infinito.

11. Acceso a memoria

Las instrucciones de lectura y escritura en la *RAM* cuentan con 32 bits para direccionar la memoria. Sin embargo, la memoria para el programador de S-MIPS, es un *array* plano de 1 MB = 2^{20} bytes, direccionable a nivel de 1 byte, por lo que solo se usarán los 20 primeros bits para el direccionamiento. Además las transferencias entre la *RAM* y el procesador ocurren siempre en bloques de 16 bytes (65536 bloques).

Por simplicidad para este proyecto académico, cada bloque de la *RAM* cuenta con 4 palabras de 4 bytes, por lo que solo se podrá direccionar a nivel de 4 bytes. Sin embargo, se desea respetar el indizado original (de 1 byte) de S-MIPS. Para ello, el procesador debe ignorar cualquier dirección de memoria que no sea divisible entre 4, e interpretarla como si fuera el número divisible por 4 más cercano por debajo. Por ejemplo, las instrucciones:

- `lw R3, 0(R0)`,
- `lw R3, 1(R0)`,
- `lw R3, 2(R0)`,
- `lw R3, 3(R0)`,

hacen lo mismo: copiarán los bytes 0, 1, 2 y 3 de la *RAM* al registro `R3`. En el caso de:

```
sw R3, 0(R0),
```

se copiará a los bytes 0, 1, 2 y 3 de la *RAM* el valor guardado en el registro $\texttt{R}_3$.

S-MIPS es una arquitectura *Little-Endian*, lo que significa que, en cada palabra de 32 bits, el byte con dirección más pequeña es el menos significativo y el de dirección más grande es el más significativo. Esto luce *"al revés"* de cómo se escriben normalmente los números. Supongamos que los primeros bytes de la *RAM* tienen estos valores:

0	1	2	3	4	5	
1	2	0	0	1	2	···

Al hacer "`lw` $\texttt{R}_3$`, 0(`$\texttt{R}_0$`)`", el byte 0 de la *RAM*, que tiene el valor 1 ($\texttt{0000 0001}_2$) va hacia la parte menos significativa de $\texttt{R}_3$ y el byte 3, con valor 0 ($\texttt{0000 0000}_2$), va hacia la más significativa. El valor final de $\texttt{R}_3$ sería:

$$\texttt{0000 0000 0000 0000 0000 0010 0000 0001}_2 = 512_{10} + 1_{10} = 513_{10}.$$

Las escrituras se comportan de igual forma. Si se hace:

```
add R3, R0, 1027
sw R3, 0(R0)
```

se escribirán para el byte 0 de la *RAM* la parte menos significativa del valor 1027 y para el byte 3 la parte más significativa. Como 1027 tiene la siguiente representación binaria:

$$1027_{10} = 1024_{10} + 2_{10} + 1_{10} = \texttt{0000 0000 0000 0000 0000 0100 0000 0011}_2,$$

su byte menos significativo contiene $\texttt{0000 0011}_2 = 3_{10}$, y su byte más significativo contiene $\texttt{0000 0000}_2 = 0_{10}$, quedando en la *RAM*:

0	1	2	3	4	5	
3	4	0	0	1	2	···

12. Interfaz con la memoria

El módulo *RAM* del circuito *S-MIPS Board* implementa una memoria asíncrona de 1 MB. Esta memoria está organizada en $2^{16} = 65536$ bloques de 16 bytes. Esta *RAM* es más lenta que el procesador, y le toma varios ciclos leer y escribir datos. La cantidad de ciclos que toma hacer una lectura o una escritura están en las salidas `RT` (`Read Time`) y `WT` (`Write Time`). Estos valores no cambian una vez comienza la ejecución del procesador.

Aunque estas salidas sean constantes, no se deben usar sus valores *"a mano"* dentro del *CPU*. Hay que obtenerlos de la RAM. Los profesores vamos a probar distintas combinaciones de esos valores durante la verificación del procesador y este debe comportarse en concordancia.

La entrada `ADDR` (16 bits) indica el bloque de la *RAM* que se quiere indizar. La entrada `CS` (`Chip Select`) indica si se va a utilizar la *RAM* (`CS` $= 1$), o no (`CS` $= 0$). La entrada `¬R/W` indica, en el caso de que `CS` $= 1$, si se leerá de la *RAM* (`¬R/W` $= 0$), o si se escribirá en la RAM (`¬R/W` $= 1$). Cuando `CS` $= 0$, se ignora el valor de `¬R/W`.

En el momento que se desee acceder a la RAM, se debe activar `CS` y especificar el tipo de operación con `¬R/W`, manteniendo persistentes estos valores durante `RT` o `WT` ciclos de la CPU (si se lee o se escribe respectivamente). Cuando se finaliza una lectura se proporcionan los 16 bytes del bloque seleccionado, a través las 4 salidas de 32 bits $\texttt{O}_0$, $\texttt{O}_1$, $\texttt{O}_2$, y $\texttt{O}_3$. Análogamente, cuando se va a realizar una escritura se deben enviar los valores por las entradas de 32 bits $\texttt{I}_0$, $\texttt{I}_1$, $\texttt{I}_2$, e $\texttt{I}_3$. Sin embargo, la escritura puede hacerse parcialmente, usando la entrada `MASK`.

La *RAM* está dividida en 4 bancos que actúan como columnas o *slices*. Cada bloque de 16 bytes de la *RAM* está dividido en 4 palabras de 4 bytes (32 bits). Los primeros 64 bytes de la *RAM* lucen así:

Banco 0	Banco 1	Banco 2	Banco 3	
00 01 02 03	04 05 06 07	08 09 10 11	12 13 14 15	← Bloque 0
16 17 16 19	20 21 22 23	24 25 26 27	28 29 30 31	← Bloque 1
32 33 34 35	36 37 38 39	40 41 42 43	44 45 46 47	← Bloque 2
48 49 50 51	52 53 54 55	56 57 58 59	60 61 62 63	← Bloque 3
⋮	⋮	⋮	⋮	⋮

La entrada `MASK` es una entrada de 4 bits que selecciona cuál o cuáles bancos van a ser modificados por una operación de escritura. El bit menos significativo de `MASK` selecciona el Banco 0, el más significativo selecciona el Banco 3. Por ejemplo, si la entrada `MASK` tiene el valor 8_{10} (1000_2) y la entrada `ADDR` es 0, la escritura solo afectaría los bytes 12, 13, 14 y 15 de la *RAM*, pues estos están en el bloque 0 y en el banco 3. En esos 4 bytes se escribiría el valor de la entrada $\texttt{I}_3$ (`Data Input 3`). Si la entrada `MASK` fuera 12_{10} (1100_2) y `ADDR` fuera 2, la escritura modificaría los bytes $40, 41, 42, 43$ y $44, 45, 46, 47$ (bloque 2, bancos 2 y 3). En esos 16 bytes se escribirían los valores de las entradas $\texttt{I}_2$ e $\texttt{I}_3$.

La entrada `MASK` no afecta las operaciones de lectura. Las salidas $\texttt{O}_0$, $\texttt{O}_1$, $\texttt{O}_2$, y $\texttt{O}_3$ siempre contienen el bloque completo solicitado en `ADDR`. Las escrituras se realizan cuando `CS` es 1 y $\neg$R/W es 1 y toman la cantidad de ciclos de CPU que indica la salida WT.