



Escuela de Ingeniería y Ciencias  
Departamento de Ciencias Computacionales  
**Arquitectura de computadoras - TE2031**

## **Manual del laboratorio**

**Instructor:** Diego Fernando Valencia Martínez  
Agosto - Diciembre 2020

# Índice

<b>Práctica #1. Componentes generales del CPU</b>	<b>4</b>
Módulos Básicos	4
Desarrollo de la práctica	7
<b>Práctica #2. La Unidad Aritmético-Lógica</b>	<b>8</b>
Introducción	8
Diseño de la Práctica	8
Desarrollo de la Práctica	9
<b>Práctica #3. Memoria de instrucciones y memoria de datos</b>	<b>10</b>
Memoria de instrucciones	10
Memoria de datos	11
Nota sobre el direccionamiento de las memorias	12
Desarrollo de la práctica	14
<b>Práctica #4. El banco de registros ['Register File']</b>	<b>15</b>
Introducción	15
Diseño de la práctica	16
Desarrollo de la Práctica	17
<b>Práctica #5. MIPS IT Simulador – Instalación</b>	<b>18</b>
Introducción	18
Desarrollo de la práctica	18
Evidencia	24
Entregable	24
<b>Práctica #6. MIPS IT Simulador – Programación</b>	<b>25</b>
Introducción	25
Desarrollo de la práctica	25
Entregable	28
<b>Práctica #7. Unidad de Control y ALU control</b>	<b>29</b>
Introducción	29
Unidad de Control	29
ALU Control	30
<b>Práctica #8. Integración de Módulos</b>	<b>32</b>
Introducción	32
Prueba mediante un código de ensamblador	33
<b>Práctica #9. Puertos paralelos de E/S para el CPU</b>	<b>35</b>
Objetivos de la práctica	35

Entregables	35
<b>Práctica #10. Interrupciones</b>	<b>36</b>
Introducción	36
<b>Diagrama general del MIPS</b>	<b>37</b>

# Práctica #1. Componentes generales del CPU

## Módulos Básicos

La tarea para esta primera práctica será realizar seis distintos módulos pertenecientes al CPU MIPS. Se recomienda revisar constantemente el diagrama al final de este documento que incluye la arquitectura del procesador que se desarrollará. Esto, con el objetivo de ir reconociendo el rol y propósito de cada módulo.

### a. Módulo 1: Add

Este módulo deberá realizar una suma 'unsigned' entre dos números de 32 bits de modo que la salida tenga la misma dimensión i.e.  $A+B=C$ .

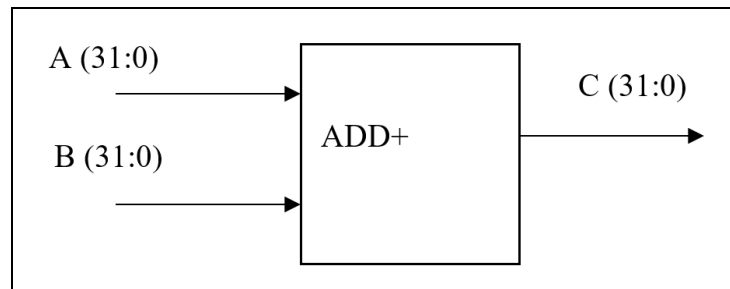


Figura 1.1. Módulo ADD para señales de 32 bits

### b. Módulo 2: Shift Left 2 bits

Este módulo recibe un vector de 26 bits para hacer un "corrimiento lógico" de 2 posiciones y obtener un vector final de 28 bits.

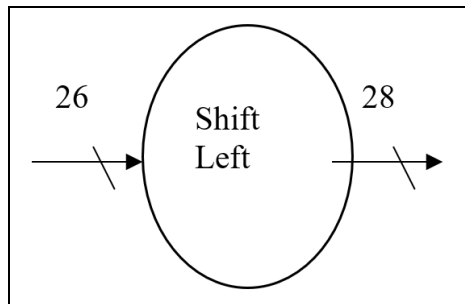


Figura 1.2. Módulo Shift Left con extensión para señales de 26 bits

### c. Módulo 3: Shift Left 2 bits para señales de 32 bits

A diferencia del módulo anterior, este corrimiento a la izquierda recibirá un vector de 32 bits y generará otro de 32 a la salida. Es decir, el corrimiento sigue el mismo concepto, pero el tamaño del vector de salida es el mismo que el de entrada.

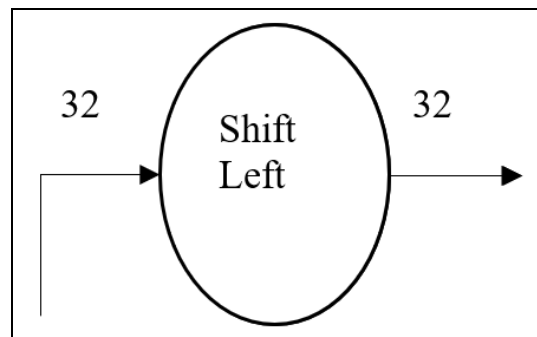


Figura 1.3. Módulo Shift Left para señales de 32 bits

### d. Módulo 4: Program Counter

Dada la siguiente entidad:

```
entity Program_counter is
  Port (D: in STD_LOGIC_VECTOR (31 downto 0);
        Q: out STD_LOGIC_VECTOR (31 downto 0);
        RESET: in STD_LOGIC;
        CLK: in STD_LOGIC);
end Program_counter;
```

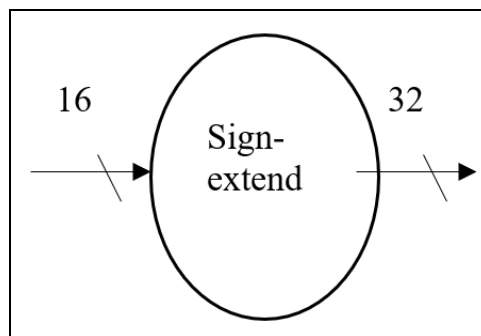
El 'program counter' no es más que un conjunto de Flip-Flop D's que, en cada transición negativa de reloj, Q obtiene lo que entre en D. La entrada **asíncrona** de RESET, pone en cero a todos los bits de la salida Q.

### e. Módulo 5: Sign extender

La extensión de signo es la operación en la cual se incrementa la cantidad de bits de un número preservando el signo y el valor del número original. Es llevada a cabo mediante agregación de dígitos del lado más significativo del número,

siguiendo ciertos lineamientos dependiendo de la representación particular utilizada.

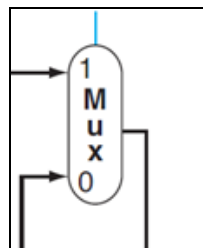
La función de este módulo es extender el tamaño de una variable al doble. Para esto, se recibe una señal de 16 bits que posteriormente pasa a 32 bits. La clave aquí es el bit más significativo del vector de entrada; si este bit está encendido (en '1'), entonces se agregan dieciséis 1's a la izquierda del vector de entrada para obtener uno de 32 a la salida. En caso contrario, si el MSB del vector de entrada se encuentra apagado (en '0'), ocurre lo mismo, pero con dieciséis 0's a la izquierda del vector de entrada.



*Figura 1.4. Módulo Sign Extender para señales de 16 bits*

#### f. Multiplexores 2 a 1

Ahora, deberás diseñar dos multiplexores en módulos independientes. Si bien ambos contarán con una entrada selectora de un bit y dos más para datos, las entradas y la salida serán de 5 bits para uno y de 32 para el otro.



*Figura 1.5. Módulo multiplexor*

## Desarrollo de la práctica

1. Dadas las descripciones anteriores, diseña en VHDL un código para cada componente que cumpla con los objetivos especificados.
2. Debes simular cada módulo con la herramienta del *ISim Simulator* del *ISE Design* mediante un código de 'test bench' que agregues al proyecto. Las simulaciones tendrán que ser demostradas al instructor. Es necesario que incluyas dentro de tu 'test bench' varios casos de prueba, especialmente casos extremos o 'edge cases'.

## Práctica #2. La Unidad Aritmético-Lógica

### Introducción

La Unidad Aritmético-Lógica, también conocida como **ALU** (por sus siglas en inglés), es un circuito digital que calcula operaciones aritméticas (suma, resta, multiplicación) y operaciones lógicas (and, or, not), entre dos números. Por mucho, los más complejos circuitos electrónicos son los que están contruidos dentro de los chips de microprocesadores modernos. Por lo tanto, estos procesadores tienen dentro de ellos un ALU muy complejo y potente. De hecho, un microprocesador moderno (y los mainframes) puede tener múltiples núcleos, cada núcleo con múltiples unidades de ejecución, cada una de ellas con múltiples ALU.

### Diseño de la Práctica

Crea un “VHDL Module” para construir y simular una **ALU de 32 bits** con las funciones que especifica la tabla 2.1. Para simular la ALU tendrás que hacerlo mediante un archivo de ‘test bench’ y demostrar que dicha ALU funciona correctamente. Observa bien que este componente que programarás cuenta con una bandera de cero o ‘zero flag’, la cual será habilitada según corresponda.

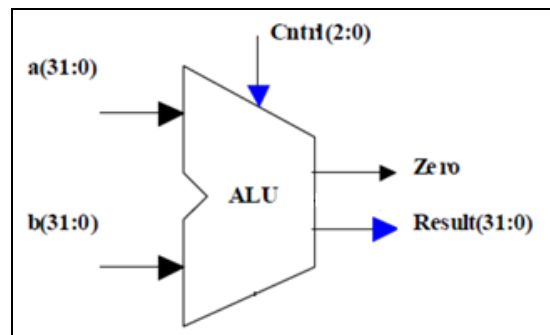


Figura 2.1. Esquema de la Unidad Aritmética Lógica



Tabla 2.1. Operaciones de la Unidad Aritmética Lógica		
ALU Control Lines	Función	Operación
000	And	Result = a and b
001	Or	Result = a or b
010	Add	Result = a + b
011	Mov	Result = a
100	B upper	Result = b[15:0] & x"0000"
110	Substract	Result = a - b
111	Set less than	Consultar más adelante

La bandera de Zero se habilita siempre y cuando ocurra una operación (cualquiera que sea) donde el resultado sea 0, por ejemplo,  $\text{Result} = a - b = 0$ ,  $\text{Zero} = '1'$ .

La operación *set less than* pregunta si  $a < b$ , entonces  $\text{Result} = \text{x}''00000001''$ ; en caso contrario,  $\text{Result} = \text{x}''00000000''$ .

## Desarrollo de la Práctica

1. Dada la descripción anterior, diseña en VHDL un código que cumpla con los objetivos especificados.
2. Verifica el correcto funcionamiento del módulo con la herramienta del *ISim Simulator* del *ISE Design* mediante un código de 'test bench' que agregues al proyecto.

## Práctica #3. Memoria de instrucciones y memoria de datos

### Memoria de instrucciones

La memoria de Instrucciones de un procesador tipo MIPS es un bloque fundamental. Esta almacena en lenguaje maquina (conjunto de 1's y 0's), las distintas instrucciones que el procesador irá ejecutando paso a paso. Esta memoria es tipo ROM, es decir, ya cuenta con un contenido específico en ella el cual no debe ser modificado durante el funcionamiento del procesador.

Deberás crear un nuevo módulo VHDL para este componente que cumpla con las siguientes características.

- Contar con 32 espacios de memoria donde cada uno alberga palabras de 32 bits
- Tener una entrada *READ\_ADDRESS[31:0]* y una salida *INSTRUCTION[31:0]*. De la salida *INSTRUCTION[31:0]* se obtiene la instrucción (el conjunto de 1's y 0's según el formato de la arquitectura MIPS) que se encuentra en la dirección *READ\_ADDRESS[31:0]*. En otras palabras, la variable *READ\_ADDRESS* se utiliza como el índice de la dirección a leer

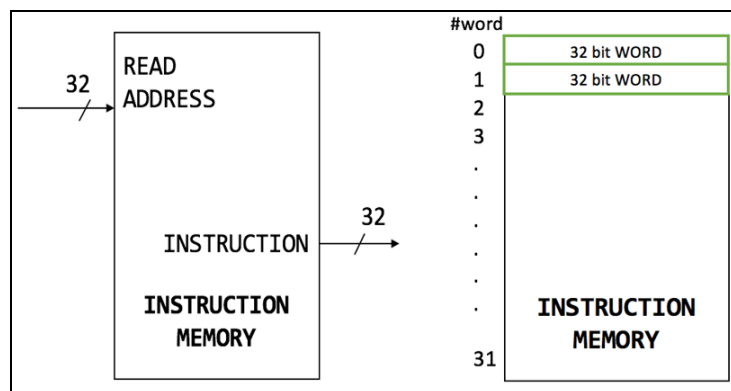


Figura 3.1. Memoria de instrucciones con palabras de 32 bits

A pesar de que la señal de entrada es de 32 bits y eso nos permite direccionar  $2^{32}$  posiciones, dado que nuestra memoria es de 32 localidades sólo necesitamos **5 bits** para la dirección “interna”. Más adelante se detalla este tema sobre el direccionamiento.

## Memoria de datos

La memoria de Datos, por lo general es una memoria tipo RAM (*Random Access Memory*) cuyas operaciones, ya sean de escritura o lectura de datos, son de acceso aleatorio. La ‘Data Memory’ contiene los datos que serán cargados a los registros o que son almacenados por alguna instrucción.

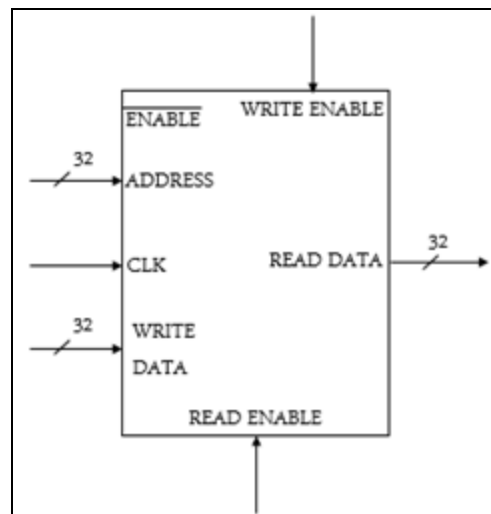
Como verás más adelante, las operaciones aritméticas y lógicas son efectuadas sobre los valores actuales de los registros del procesador y, dado que estos registros son limitados (32 para nuestro MIPS), el resto de los datos con los que trabajamos se quedan en memoria hasta ser necesitados.

La siguiente tabla contiene las entradas y salidas de la memoria.

Tabla 3.1. EE/SS de la memoria de datos	
Entradas: + ENABLE + WRITE_ENABLE + READ_ENABLE + CLK + RW_ADDRESS + WRITE_DATA	Salidas: + READ_DATA

Ahora, deberás crear un módulo VHDL distinto para el diseño de la memoria de datos que cumpla con lo siguiente.

- Las entradas de WRITE\_ENABLE y READ\_ENABLE son habilitadores para los procesos de lectura y escritura, nunca pueden estar los dos en '1' al mismo tiempo.
- La lectura consiste en poner en READ\_DATA el contenido de memoria de la dirección especificada por ADDRESS.
- El proceso de escritura almacena en memoria en la dirección dada por ADDRESS el dato que entra a WRITE DATA cada que ocurra una transición negativa de reloj por la entrada de CLK.
- En el proceso de escritura, READ\_DATA "saca" puros 0's puesto que no leyó ningún dato.
- ENABLE resulta un habilitador general de la memoria en sí (notar que está negado). Para efectos de esta práctica, dicha entrada no será requerida sino hasta las prácticas finales.
- Nota que la lectura no está asociada a un ciclo de reloj.



*Figura 3.2. Memoria de datos*

## **Nota sobre el direccionamiento de las memorias**

Como ya habrás notado en la práctica que realizaste el módulo 'program counter', el incremento de éste va de 4 en 4. Esto es debido a que las instrucciones para

la arquitectura MIPS se componen de 4 bytes cada una. Y es precisamente la salida del 'PC' la misma señal de entrada de *ADDRESS* de nuestra memoria de instrucciones. A continuación, se muestra un ejemplo de cómo se puede definir una memoria dentro de un 'process'.

```

PROCESS (Lista_sensitividad)
SUBTYPE REGISTRO IS STD_LOGIC_VECTOR (31 DOWNTO 0);
TYPE REG_BANK IS ARRAY (0 TO 31) OF REGISTRO;
VARIABLE ROM_MEMORY: REG_BANK: = (
                                X"0000",
                                X"0001",
                                X"0002",
                                X"0003",
                                OTHERS => (OTHERS => '0')
                                );
BEGIN
    << cuerpo del process >>
END PROCESS;

```

Como ya se introdujo anteriormente, solo necesitamos **5 bits** para direccionar nuestros módulos-memorias. De igual forma el **índice interno** de nuestras memorias, si se realiza como el ejemplo anterior, es un **número natural**, por lo que debemos dividir nuestro valor de *ADDRESS* entre 4 para acceder secuencialmente a las localidades (Recuerda que el PC aumenta de 4 en 4).

Dirección De Memoria					Número de palabra (índice de memoria interna)
0x00	BYTE 0	BYTE 1	BYTE 2	BYTE 3	palabra 1
0x04	BYTE 4	BYTE 5	BYTE 6	BYTE 7	palabra 2
0x08	BYTE 8	BYTE 9	BYTE A	BYTE B	palabra 3
0x0C	BYTE C	BYTE D	BYTE E	BYTE F	palabra 4

Figura 3.3. Acomodo de memoria en nuestro MIPS

De acuerdo con la figura anterior, podemos notar que si se quiere acceder a la **segunda** palabra de la memoria, la instrucción de ensamblador deberá contener la literal **0x04**, para la **tercera** palabra la **0x08** y así sucesivamente.

Para nuestras memorias internas sucede algo como lo siguiente:

b0000\_0000\_0000\_0(000\_01)00 => 0x04, ...1

b0000\_0000\_0000\_0(000\_10)00 => 0x08, ...2

b0000\_0000\_0000\_0(000\_11)00 => 0x0C, ...3

Donde los **5 bits** resaltados entre paréntesis nos dan el índice natural al que debemos acceder para que exista correspondencia entre el [PC + 4] y nuestras direcciones internas que van de 1 en 1.

## Desarrollo de la práctica

1. Dada las descripciones anteriores, diseña en VHDL un código para cada bloque que cumpla con los objetivos especificados. Es decir, un módulo para la memoria de instrucciones y otro para la memoria de datos.
2. Debes simular cada módulo con la herramienta del *ISim Simulator* del *ISE Design* mediante un código de 'test bench' que agregues al proyecto. Las simulaciones deberán mostrar lo siguiente:
  - a. Leer al menos tres localidades distintas de la memoria de instrucciones.
  - b. Realizar lectura en al menos dos localidades distintas de la memoria de datos, mostrar el contenido y posteriormente escribir datos nuevos. Al final volver a leer para verificar que el contenido cambió. Recuerda que la lectura no está asociada a un pulso de reloj.

## Práctica #4. El banco de registros ['Register File']

### Introducción

Un 'register file' es un arreglo de registros del procesador en un CPU. Resulta ser un módulo fundamental del procesador RISC diseñado e implementado en esta clase. El banco de registros es un conjunto de registros que son accedidos mediante un sólo puerto de escritura y dos puertos de lectura. Los puertos de lectura proveen información de datos al ALU, mientras que el puerto de escritura obtiene resultados de éste último y la memoria de datos. La arquitectura MIPS tiene un 'register file' de 32 registros de 32 bits c/u.

La figura 4.1 muestra los principales componentes de un 'register file': 32 registros de 32 bits cada uno, 2 multiplexores para leer simultáneamente 2 valores de 32 bits de dos diferentes registros usando *RegData1* y *RegData2*; además un decoder que selecciona el registro para en él escribir datos que vienen de las terminales de *WriteData*. También, posee una terminal de **enable** nombrada *RegWrite*.

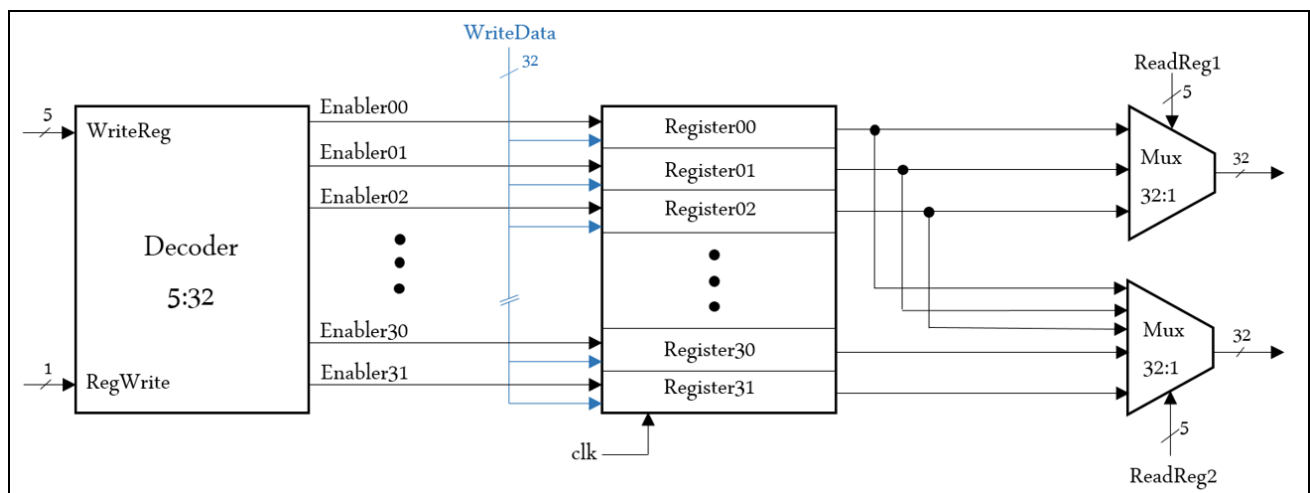


Figura 4.1. Esquema general del Banco de Registros

## Diseño de la práctica

Tendrás que realizar un módulo VHDL para cada uno de los siguientes componentes, que de manera conjunta forman al 'register file'.

### a. Multiplexor

Ya que necesitaremos dos multiplexores en nuestro 'register file', construye uno que acepte 32 cables de entrada de 32 bits cada uno y saque uno sólo también de 32 bits por *ReadData1* o *ReadData2*, según corresponda. Las entradas de *ReadReg1* y *ReadReg2* (de 5 bits de ancho cada una) serán las encargadas de escoger qué dato será el que se envíe al correspondiente *ReadData*.

### b. Decoder 5x32

El decoder tiene una entrada selectora de 5 bits de ancho (*WriteReg*) y, por supuesto, 32 cables de salida de un solo bit. El decodificador posee una entrada adicional de **enable** (*RegWrite*) para habilitar al selector. Si dicha entrada no está habilitada, las 32 salidas están en cero lógico o desactivadas.

### c. Registro

El registro posee 32 entradas habilitadoras, una señal de reloj y un bus de 32 bits con el dato a escribir (*WriteData*). Las salidas son el contenido de cada uno de los 32 registros de 32 bits. **El registro r0 nunca debe ser modificado y siempre deberá mantener un valor de cero.**

### d. Register File

Crea un módulo de 'register file' con la interfaz especificada anteriormente haciendo instancias de los componentes previos, es decir, utilizando el estatuto de 'PORT MAP' para realizar el "cableado" y lograr lo que se muestra en la figura anterior. Asegúrate de que los nombres de tus señales y componentes son los mismos que los nombres en el diagrama. Este banco de registros contendrá 32 registros, un decodificador y dos multiplexores que se conectarán entre sí según el diagrama visto al inicio de la práctica.



## Desarrollo de la Práctica

1. Dada las descripciones anteriores, diseña en VHDL los módulos indicados.
2. Debes simular cada componente con la herramienta del *ISim Simulator* del *ISE Design* mediante un código de 'test bench' que agregues al proyecto. Deberás simular cada uno de los componentes internos y finalmente el 'register file' completo compuesto por los sub componentes mencionados. Las simulaciones tendrán que ser demostradas al instructor.

## Práctica #5. MIPS IT Simulador – Instalación

### Introducción

Bajo el nombre MIPS (Microprocessor without Interlocked Pipeline Stages) se conoce a toda una familia de microprocesadores de arquitectura RISC que fueron desarrollados por MIPS Technologies. Los diseños del MIPS han sido utilizados en muchos sistemas embebidos, routers, videoconsolas e.g. Nintendo 64, entre otros. Debido a que los diseñadores crearon un conjunto de instrucciones entendible y simplificado, los cursos sobre arquitectura de computadores en universidades y escuelas a menudo se basan en la arquitectura MIPS.

Particularmente, para esta práctica el alumno utilizará el software de simulación de una computadora MIPS. En éste se identifican los distintos componentes de su arquitectura e.g. CPU y registros, memoria principal, puertos, periféricos y memoria caché. El simulador *MipsIt* cuenta con un ensamblador y desensamblador de instrucciones que permite entender la codificación/estructura de una instrucción/dato a nivel de bits dentro de la memoria principal. De igual manera, dentro de la práctica se revisará la estructura básica de un programa en ensamblador dentro del entorno del MIPS.

### Desarrollo de la práctica

Asegúrate de contar con los softwares *MipsIt Studio 2000* (entorno de programación) y *Mips* (simulador). Dentro del primero programaremos distintas rutinas para el MIPS. Un proyecto dentro del *MipsIt Studio* significa tener una serie de archivos fuente (source files) que son compilados y ensamblados para generar un archivo ejecutable que podrá simularse y verificar su funcionamiento gracias al simulador MIPS. En nuestro caso, un proyecto puede contener programas escritos en ensamblador o en lenguaje C. La imagen siguiente muestra un ejemplo de cómo se vería el ambiente que se usará para diseñar código.

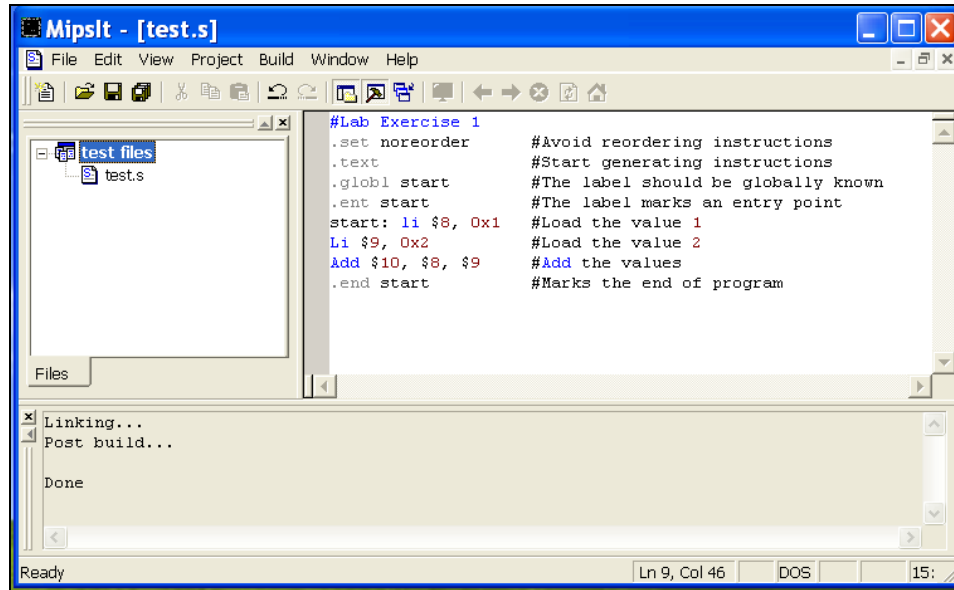
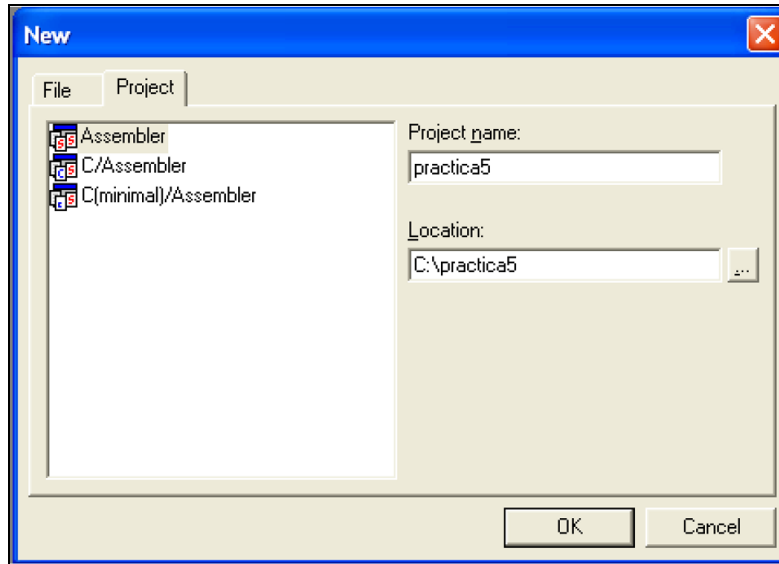


Figura 5.1. Entorno MipsIt

La **ventana de programas** (derecha) es donde el usuario escribe el código y puede ver el contenido del archivo creado, por otra parte, la **ventana de workspace** (izquierda) muestra la lista de 'source files' que contiene nuestro proyecto. Para abrir un archivo en específico basta con dar doble clic sobre éste. La **ventana de output** contiene los resultados de la compilación y ensamblaje del código escrito.

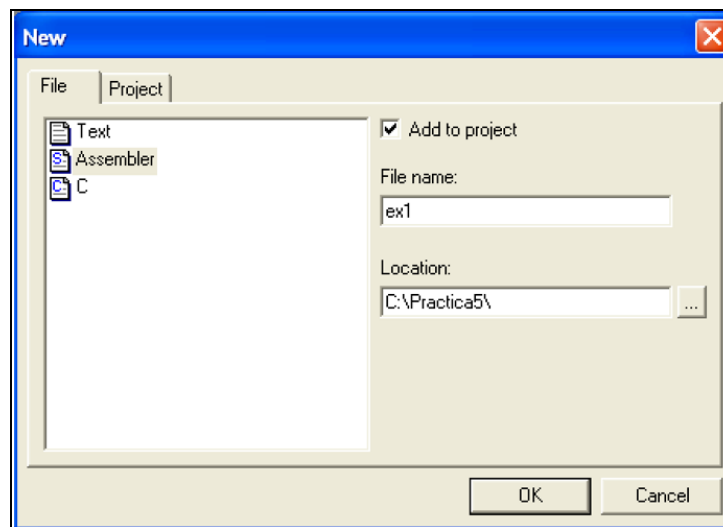
Para crear un nuevo proyecto:

1. Ve a **File -> New**. En la pestaña **Project** selecciona el tipo de proyecto, en nuestro caso tendrás que escoger la opción **Assembler**.
2. En **Project Name** nombra al proyecto **practica5**. En **Location**, especifica la carpeta que deberá contener nuestro proyecto recién creado. Es importante que la ruta de acceso a dicha carpeta no tenga espacios entre los nombres o caracteres inválidos. Por último, da clic en **OK**.



*Figura 5.2. Creación proyecto*

3. Ahora para asociar archivos fuente a nuestro proyecto, repite el paso 1 sólo que no olvides tener seleccionada la pestaña **File**. Aquí también tendrás que seleccionar **Assembler** como tipo archivo. Nombra a este archivo como **ex1**.



*Figura 5.3. Creación archivo*

Si se desea agregar un archivo existente al proyecto tendrás que situarte en el menú de **Project -> Add file...**

4. Situado dentro de ex1.s copia y pega el siguiente código en él.

```
#Lab Exercise 1
.set noreorder           #Avoid reordering instructions
.text                   #Start generating instructions
.global start            #The label should be globally known
.ent start               #The label marks an entry point
start:  li $8, 0x1        #Load the value 1
        li $9, 0x2        #Load the value 2
        add $10, $8, $9    #Add the values
        .end start        #Marks the end of program
```

5. En **File -> Save** (o Ctrl + S) puedes guardar los cambios realizados y en el menú **Build** compila tanto el ex1.s (Ctrl + F7) y el proyecto (F7). Si no hay errores, en la ventana **Output** aparecerá el mensaje **Done**.

En cuanto al simulador *MIPS*, es un entorno gráfico que permite visualizar en todo momento el estado de la memoria, los registros de la CPU, la salida de la consola, y así sucesivamente. Haciendo doble clic en el archivo Mips.exe, aparecerá una ventana como la siguiente:

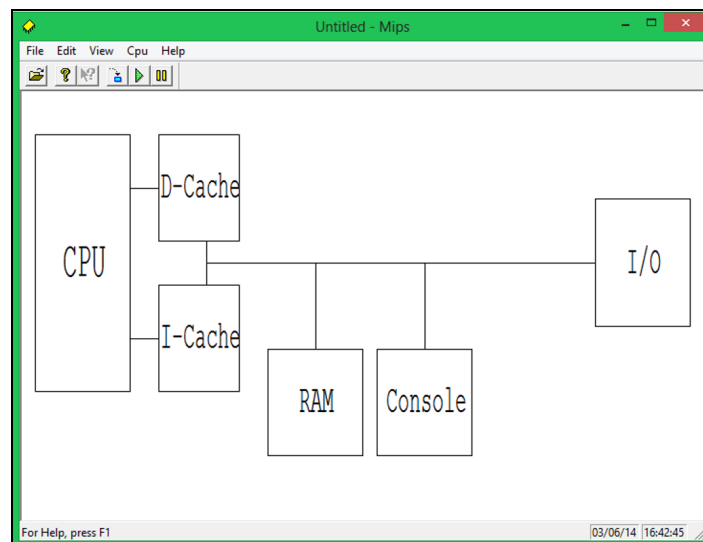


Figura 5.4. Simulador MIPS

En dicha ventana podrás observar que hay seis módulos, para abrir cada uno de estos módulos sólo hay que dar click sobre cualquiera de ellos.

- **CPU**

Dentro de este módulo se puede ver/modificar el contenido de los registros de la CPU incluyendo el 'program counter' (PC).

Registers			
r0/zero=00000000	r1/at =00000000	r2/v0 =00000000	r3/v1 =00000000
r4/a0 =00000000	r5/a1 =00000000	r6/a2 =00000000	r7/a3 =00000000
r8/t0 =00000000	r9/t1 =00000000	r10/t2 =00000000	r11/t3 =00000000
r12/t4 =00000000	r13/t5 =00000000	r14/t6 =00000000	r15/t7 =00000000
r16/s0 =00000000	r17/s1 =00000000	r18/s2 =00000000	r19/s3 =00000000
r20/s4 =00000000	r21/s5 =00000000	r22/s6 =00000000	r23/s7 =00000000
r24/t8 =00000000	r25/t9 =00000000	r26/k0 =00000000	r27/k1 =00000000
r28/gp =00000000	r29/sp =800bc000	r30/fp =00000000	r31/ra =bfc00088
pc =80020000	mdhi =00000000	mdlo =00000000	conf =00000000
bad va =00000000	status =00400000	cause =00000000	epc =00000000

Figura 5.5. Registros

- **RAM**

Recordemos que el MIPS tiene un espacio de direcciones de 32 bits, es decir un total de 232 posiciones de 1 byte de memoria cada una. Al hacer clic en **RAM**, verás una ventana con el contenido del reporte ordenado por filas. Cada una lleva una dirección especificada ('Address'), junto a la cual aparece la memoria en esa posición (ambos expresados en notación hexadecimal).

Memory		
8001FFC8	00 00 00 00	NOP
8001FFCC	00 00 00 00	NOP
8001FFD0	00 00 00 00	NOP
8001FFD4	00 00 00 00	NOP
8001FFD8	00 00 00 00	NOP
8001FFDC	00 00 00 00	NOP
8001FFE0	00 00 00 00	NOP
8001FFE4	00 00 00 00	NOP
8001FFE8	00 00 00 00	NOP
8001FFEC	00 00 00 00	NOP
8001FFF0	00 00 00 00	NOP
8001FFF4	00 00 00 00	NOP
8001FFF8	00 00 00 00	NOP
8001FFFC	00 00 00 00	NOP
80020000	00 00 00 00	NOP
80020004	00 00 00 00	NOP
80020008	00 00 00 00	NOP
8002000C	00 00 00 00	NOP
80020010	00 00 00 00	NOP
80020014	00 00 00 00	NOP
80020018	00 00 00 00	NOP
8002001C	00 00 00 00	NOP
80020020	00 00 00 00	NOP
80020024	00 00 00 00	NOP
80020028	00 00 00 00	NOP
8002002C	00 00 00 00	NOP
80020030	00 00 00 00	NOP
Address mode: Virtual View mode: Assembler		

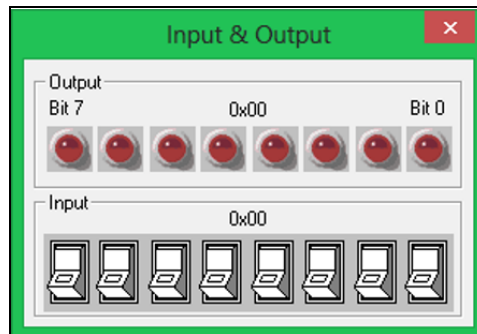
Figura 5.6. Memoria RAM

- **Console**

La consola no es más que una entrada/salida estándar para los programas.

- **I/O**

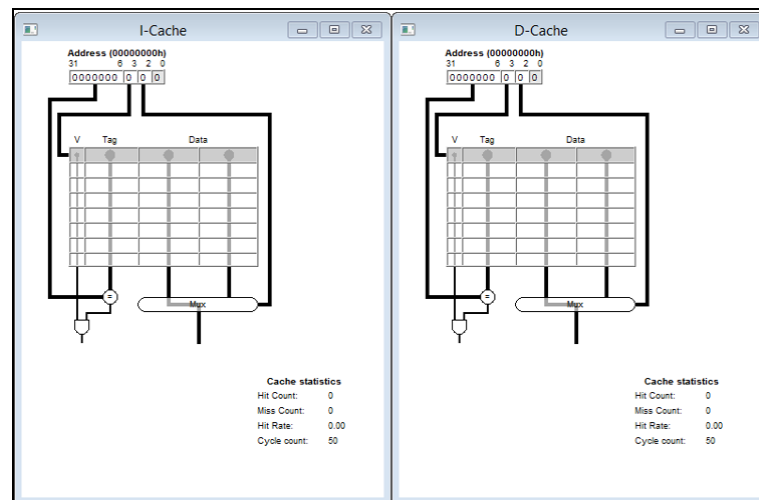
Aquí se pueden simular I/O mediante los 8 switches y los 8 LEDs.



*Figura 5.7. Inputs y Outputs*


- **Caché D/I**

Memorias caché para datos e instrucciones.



*Figura 5.8. Memorias caché*

Para cargar un programa ensamblador en el simulador, dentro de *MipsIt Studio 2000* haz clic en **Build->Upload->To Simulator**. Otra forma de hacer esto desde el simulador es seleccionar el menú **File->Open** para después cargar el archivo con extensión **.srec**.

Ya que hayas realizado cualquiera de las dos opciones anteriores, es hora de ejecutar la simulación. Podrás observar en la parte superior tres botones como los siguientes: . El botón de en medio (la flecha verde) hace que la simulación comience y siga de principio a fin, mientras que el último (las dos barras amarillas) pausan la misma. El primero de ellos (el azul) obliga a que la simulación ocurra paso a paso y se pueda ir viendo cómo es que cada ventana de cada módulo va cambiando conforme a la programación hecha.

## Evidencia

Muestra el proyecto corriendo al instructor. Considera que esta práctica tiene una extensión de dos semanas. La intención de la primera parte es familiarizarse con el simulador. También, toma en cuenta que se entregarán dos reportes. Sin embargo, para el segundo es suficiente con anexar lo realizado en la práctica 6 al primero que realices.

## Entregable

- Asegúrate de entregar tu carpeta del proyecto que contiene el archivo ex1.s.
- Un archivo PDF que además de los requisitos mínimos para todo reporte contenga lo siguiente:
  - Capturas de pantalla de la simulación con el Mips.exe de las ventanas de *CPU*, *I-Cache* y *RAM*, donde se pueda ver que el programa escrito en ensamblador se ha ejecutado paso a paso.
  - Explicación detallada donde se denote que el código en ensamblador se ha comprendido.



## Práctica #6. MIPS IT Simulador – Programación

### Introducción

Ahora es momento de estudiar y comprender a detalle la “instrucción de máquina” i.e. ver cómo es que se traducen instrucciones de lenguaje ensamblador a código máquina, cómo se guardan en memoria y cómo es que se ejecutan. El código máquina es el lenguaje directamente interpretable por un circuito microprogramable e.g. el microprocesador de una computadora o el microcontrolador de un autómata. Este lenguaje está compuesto por un conjunto de instrucciones que determinan acciones al ser tomadas por los diversos componentes de la arquitectura. Un programa en este lenguaje consiste en una cadena de estas instrucciones formadas únicamente por 1's y 0's. Estas instrucciones son normalmente ejecutadas en secuencia, con eventuales cambios de flujo causados por el propio programa o eventos externos. Es necesario considerar que el lenguaje de máquina es específico de la arquitectura, aunque el conjunto de instrucciones disponibles pueda ser similar entre distintas arquitecturas. Adicionalmente, los tipos de instrucción permitidos están definidos y determinados dentro de cada plataforma en el conjunto de instrucciones (en inglés ISA, ‘instruction set architecture’), que también determina los registros de origen y destino de la CPU, y en ocasiones un dato inmediato (aquellas literales que son especificados explícitamente en la instrucción).

### Desarrollo de la práctica

Considera la siguiente instrucción en ensamblador:

```
subi $8, $8, 0x2
```

Esta instrucción resta el valor de 2 al contenido del registro \$8 y coloca el resultado en el mismo registro (sobrescribiendo en él). La instrucción no puede ser

ejecutada por la computadora, por ello debe traducirse a código máquina. Entonces, contesta las siguientes preguntas:

1. Investiga y averigua cuántos bits son necesarios para representar la instrucción en código máquina.
2. Traduce la instrucción a código máquina. ¿Una instrucción es suficiente? Proporciona tus conclusiones y respuesta en formato hexadecimal y binario.
3. Ya que hayas averiguado cómo se representa la instrucción escrita en hexadecimal, en la ventana 'Memory' del módulo RAM de nuestro simulador Mips.exe escribe ese número que obtuviste en la dirección de memoria 0x80020000 y establece el Program Counter en 0x80020000 (en la ventana de los registros del módulo de CPU).

Hecho esto, verifica que en efecto tu respuesta en la pregunta 1 es correcta ya que deberá aparecer tal cual la instrucción en la ventana 'Memory' (a este proceso se le conoce como desensamblar o 'disassembling').

- a. ¿Qué es lo que en realidad se almacena en memoria? Anota las distintas representaciones que toma el contenido de memoria 0x80020000 (ASCII, integer, floating point).
  - b. ¿Cómo se interpreta cada representación?
  - c. ¿Cuántos bits observas?
  - d. ¿Cómo puede la computadora saber que el patrón de bits almacenado es una instrucción?
4. Ahora, resetea el CPU en el menú homónimo del Mips.exe **Cpu => reset**. Escribe en el registro **\$08** (en la ventana de CPU) el valor de **0xFF** y en la localidad de memoria 0x80020000 de la ventana de Memory vuelve a colocar las instrucciones del principio. Ejecuta paso a paso las instrucciones con el botón



- a. ¿Cuál es el nuevo valor del 'program counter'? ¿Por qué?

- b. ¿Cuál es el nuevo valor del registro \$8? ¿Por qué?
5. Dado el siguiente código en ensamblador:

```
#include <iregdef.h>
.set noreorder
.text
.globl start
.ent start

start: jal wait # Wait for button click
      nop
      lui s0, 0xbf90 #Load switch port address
      lb s1, 0x0(s0) #read first number from switches
      nop
      jal wait #Wait for button click
      nop
      lb s2, 0x0(s0) #Read second number from switches
      nop
      addu s3, s1, s2 #Perform an arithmetic operation
      sb s3, 0x0(s0) #Wirte the result to LEDs
      b start #Repeat all over again
      nop

####Add code for wait subroutine here! ####

.end start
```

- a. ¿Qué hace el programa?
- b. ¿Para qué funciona la librería incluida?
- c. ¿Qué notas sobre el puerto de I/O?

A continuación, escribe una subrutina 'wait' que haga que espere a que el botón de K2 sea presionado (de la ventana de **View->Interrupt**) y luego espere de nuevo a que el botón sea soltado y finalmente regrese. Para ello, se debe leer de la dirección **0xbfa00000** hasta que reciba un '1' en el **bit 0**. Luego, tendrá que esperar hasta que el bit se haga '0' de nuevo y por último regrese de la subrutina. Este método de recibir información del "mundo exterior" es conocido como **polling**. La subrutina debe ser llamada con la instrucción **jal wait** como se muestra en el código. La instrucción JAL

'jump and link' sirve para acceder a subrutinas debido a que coloca el PC en su propia dirección y además almacena la dirección de retorno en el registro **\$ra**.

De igual manera, simula el programa previo y utiliza los switches y LEDs del simulador (ventana de I/O) para investigar la suma entera definida por la instrucción **addu** (add unsigned). Esta operación suma enteros positivos. Prueba con las sumas 1+2, 0x0F + 2 y algunas otras que creas apropiadas para entender mejor la instrucción. Trata de cargar el valor de 0xFF con los switches usando la instrucción '**lb**'.

¿Qué sucede con los datos en el registro? ¿Por qué?

¿Cuándo es que aparecen resultados especiales o inusuales en la suma?

## Entregable

- Tu carpeta del proyecto que contiene los códigos en ensamblador hechos durante la parte 1 y 2.
- Un archivo PDF que además de los requisitos mínimos para todo reporte contenga lo siguiente:
  - Las respuestas a cada una de las preguntas
  - Capturas de pantalla de la simulación con el Mips.exe de las ventanas de CPU y RAM que justifiquen tus respuestas

## Práctica #7. Unidad de Control y ALU control

### Introducción

La unidad de control (UC) es uno de los tres bloques funcionales principales en los que se divide una CPU. Los otros dos bloques son la unidad de proceso y el bus de I/O. Su función es buscar las instrucciones en la memoria principal, decodificarlas (interpretarlas) y ejecutarlas, empleando para ello la unidad de proceso. Por otro lado, el bloque de ALU control tiene la función de especificarle a la ALU qué operación deberá ejecutar. Mediante ciertas combinaciones en las entradas de *ALUop* e *Instruc[5-0]*, la salida de *Cntrl* (que a su vez funge como entrada de control para la ALU) genera una señal de 3 bits que ordena a la unidad aritmética lógica a efectuar alguna operación, ya sea lógica o aritmética, entre dos valores.

### Unidad de Control

1. Diseña en VHDL una Unidad de Control (Control Unit) que sea capaz de manipular las distintas instrucciones designadas en el ISA de nuestro MIPS. Recuerda que el procesador que implementarás tendrá un set de instrucciones básico, evidentemente existen procesadores MIPS con muchas más instrucciones, nosotros consideraremos por el momento a las siguientes.

Instructions	Operations
Add rd, rs, rt	$Rd = rs + rt$
Sub rd, rs, rt	$Rd = rs - rt$
And rd, rs, rt	$Rd = rs \text{ and } rt$
Or rd, rs, rt	$Rd = rs \text{ or } rt$
Slt rd, rs, rt	If( $rs < rt$ ) $rd = 1$ ; else $rd = 0$ ;
Lw rt, offset(rs)	$Rt = \text{Mem}[\text{offset} + rs]$
Sw rt, offset(rs)	$\text{Mem}[\text{offset} + rs] = rt$
Beq rs, rt, offset	If( $rs \diamond rt$ ) $pc \leftarrow pc + \text{offset} + 4$
J addr	$PC \leftarrow \text{addr}$
Addi rt, rs, imm	$Rt = rs + \text{imm}$
Ori rt, rs, imm	$Rt = rs \text{ or } \text{imm}$
Lui rt, imm	$Rt = \text{imm} \ll 16$
Jr rs	$PC = rs$ PC jumps to the address in rs

Figura 7.1. ISA del MIPS del laboratorio

El circuito deberá tener una entrada de 6 bits *opCode* y nueve salidas. Ocho de ellas serán de un sólo bit y *ALUop* de tres. Las restantes ocho salidas: *RegDest*, *Jump*, *Branch*, *MemRead*, *MemtoReg*, *MemWrite*, *ALUSrc* y *RegWrite*, irán tomando el valor de '0' o '1' según la entrada *opCode*.

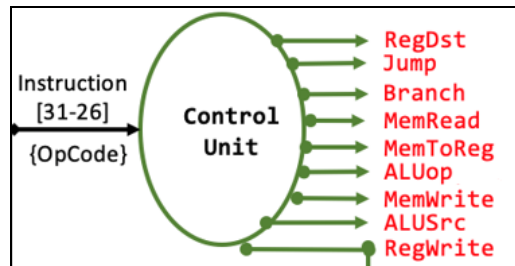


Figura 7.2. Unidad de control

Elabora una tabla de verdad para que te ayude a probar distintos 'opCodes' para operaciones tipo R, tipo I y tipo J. Se espera que las respuestas concuerden con dicha tabla. Simula tu código y verifica su funcionamiento con el *ISim Simulator*.

## ALU Control

Considerando la figura mostrada a continuación, diseña en VHDL una ALU control que reciba dos entradas i.e. una de 6 bits (los 6 bits menos significativos de la salida de *Instr* de la 'Instruction Memory'), otra de 3 bits *ALUop* que proviene de la unidad de control, y una salida de 3 bits que será la entrada de control *Cntrl[2:0]* a la ALU que ya se se programó e implementó en la práctica 1. Agrega también, una salida más de un bit llamada *jr* que sea la señal de control para activar el 'jump' de la función jr.

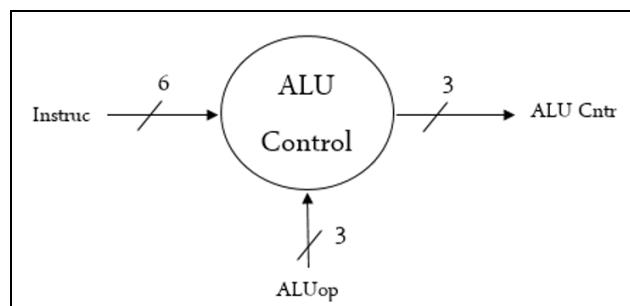


Figura 7.3. ALU Control

De igual manera, elabora una tabla de verdad para que te ayude a probar distintas 'functions' y 'ALUops' para operaciones tipo R, tipo I y tipo J. Se espera que las respuestas concuerden con dicha tabla. Simula tu código y verifica su funcionamiento con el ISim Simulator.

## Práctica #8. Integración de Módulos

### Introducción

En esta práctica crearás un proyecto nuevo y dentro de un archivo llamado 'MIPS Processor' se realizará la interfaz de unión entre todos los componentes que has realizado. Harás una instancia de cada uno de ellos en el archivo usando las sentencias 'COMPONENT' y 'PORT MAP'. Cabe señalar que es responsabilidad del alumno haber hecho a conciencia los componentes previos y cerciorarse de su funcionamiento individual adecuado. Deberás seguir el diagrama del MIPS al final de este manual para unir cada módulo y conjuntar tu procesador. Las únicas dos entradas que tu procesador MIPS tendrá son las de *RESET* y *CLK*. Posteriormente, incorporaremos una entrada y salida de puerto paralelo para que el procesador interactúe con la tarjeta Nexys.

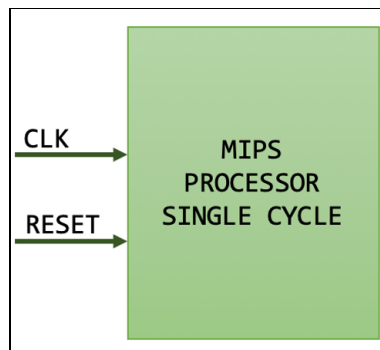


Figura 8.1. Entradas del procesador

Para probar el funcionamiento del procesador, modifica la 'Instruction Memory' para agregar el código incluido en la siguiente imagen:

```
variable memory: mem_array(0 to 5) := ("10001100000000010000000000000000", --LW r1,0(ZERO)
                                         "00000000000000000000000000000000", --NOP
                                         "100011000000000100000000000000100", --LW r2,4(ZERO)
                                         "00000000000000000000000000000000", --NOP
                                         "00000000010001000011000001000000", --ADD r3,r1,r2
                                         "00000000000000000000000000000000"); --NOP
```

Figura 8.2. Código prueba



Corre el simulador y haz oscilar al reloj al menos  $(n + 1)$  ciclos, donde  $n$  es el número de instrucciones relevantes que cargaste en el 'Instruction Memory'. Como podrás deducir, el código debe cargar las dos primeras palabras de la memoria de datos al registro 1 y 2 respectivamente y, posteriormente, sumarlas y guardar ese valor en el registro 3. Haz diversas pruebas de este pequeño código y asegúrate de su correcto funcionamiento antes de pasar a la siguiente sección.

## Prueba mediante un código de ensamblador

Ahora cargaremos un código que use el total de instrucciones que puede actualmente ejecutar nuestro procesador (¿Cuáles son?, ¿Dónde se definieron?). Deberás traducir a lenguaje maquina el siguiente programa en ensamblador, puedes ayudarte de algún recurso en línea para ello, pero verifica que la traducción quede de forma correcta y compatible con nuestro procesador.

```
add    r8, r0, r0
lw     r1, 0x0(r0)
lw     r2, 0x4(r0)
lw     r3, 0x8(r0)
add    r3, r2, r1
or     r4, r2, r1
sub    r5, r2, r1
and    r6, r2, r1
j      L1

L2:    addi r8, r8, 0x01
L1:    slt r7, r0, r8

beq    r7, r8, L2
lui    r9, 0x2000
ori    r9, r9, 0x14
sw     r8, 0x0(r9)
lw     r10, 0x0(r9)
jr     r0
```

*Figura 8.3. Programa ensamblador prueba*

Ahora deberás cargar el programa a la memoria de instrucciones. Antes de ejecutarlo en el simulador, sigue el programa “a mano” y determina cuál debe ser el valor final de los registros r1, r2, r3, r4, r5, r6, r7, r8, r9 y r10, asumiendo que:

```
mem[0] = 1
```

```
mem[1] = 2
```

```
mem[3] = 3
```

Es decir, precarga estos valores en memoria.

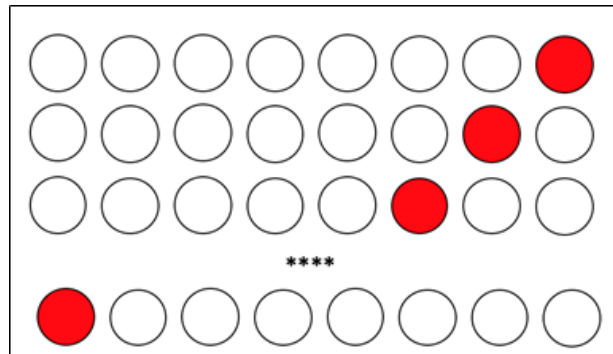
Elabora una tabla con la evolución de los valores de estos registros y verifica su valor final corriendo el programa. Incluye la tabla en tu reporte. Incluye también qué localidades de memoria fueron escritas/leídas. Y pon especial atención a los “store word” y “load word” de al final, ¿cómo se va comportar tu procesador con esos valores?

## Práctica #9. Puertos paralelos de E/S para el CPU

Para obtener puertos de E/S de nuestro procesador, tendrás que hacer algunas modificaciones a la 'Data Memory' para que el dato que sea escrito en una localidad que tu definas, salga con sus 8 bits menos significativos y pueda visualizarse en los LEDs de la tarjeta Nexys. Adicionalmente, deberás mapear el contenido de 8 push buttons.

### Objetivos de la práctica

1. Lograr reproducir la siguiente secuencia de ida y vuelta



2. Reflejar en los LEDs el estado de los switches. Debes hacer los mapeos desde y hacia la "Data Memory".
3. Implementar un **divisor de frecuencia** para que la secuencia se pueda ver en nuestra tarjeta.

### Entregables

- Un archivo PDF que además de los requisitos mínimos para todo reporte contenga lo siguiente:
  - Capturas de pantalla de la simulación utilizando la herramienta de *ISim Simulator*.
  - Una foto de la Nexys funcionando con el código.

## Práctica #10. Interrupciones

### Introducción

Las interrupciones permiten que un microprocesador -inclusive microcontrolador- puedan manejar un sinnúmero de casos, desde comunicación con dispositivos hasta temporizadores.

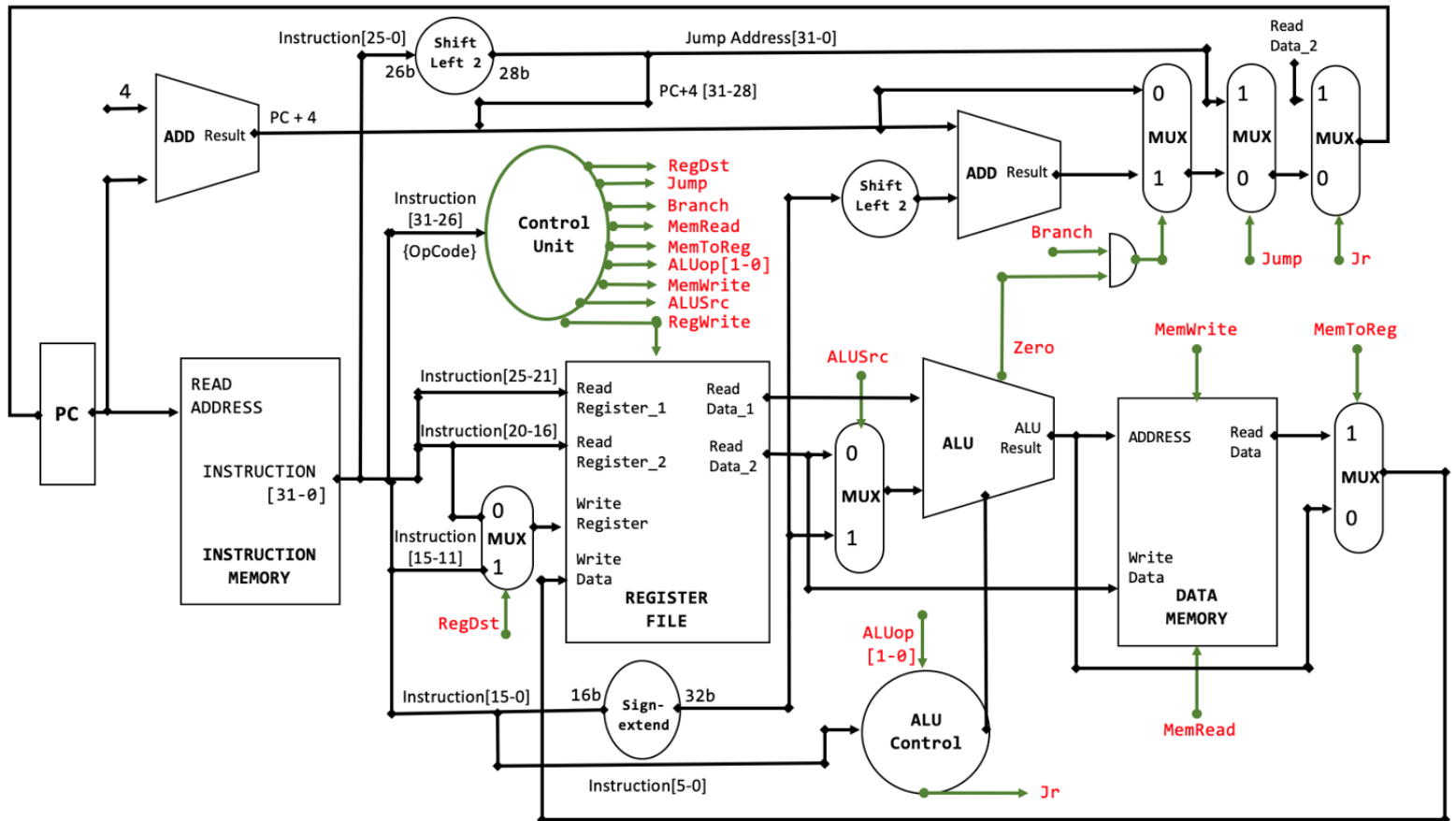
A través de esta práctica se buscará incluir que la tarjeta con MIPS tenga un puerto de interrupción que, al momento de activarse, cambie el contexto y el PC se mueva al vector de interrupciones.

Será necesario agregar diferentes señales.



***Práctica en construcción***

## Diagrama general del MIPS



Tecnológico  
de Monterrey