

Universidad de Guadalajara

Piensa y trabaja



Procesador MIPS

Equipo 6: “Fontaneros”

- Alan Emmanuel Marin Lemus
- Denice Estefania Rico Morones
- Santiago Rafael Gómez Brizuela
- Ferran Prado Roesner

Turno: Matutino

Github: <https://github.com/DeniceMorones/Proyecto-Final-.git>

Universidad de Guadalajara

Piensa y trabaja

Contenidos

- [Data path](#)
 - [Introducción](#)
 - [1era fase](#)
 - [Decodificador](#)
 - [Instrucciones](#)
 - [Verilog](#)
 - [Testbench](#)
 - [Conclusiones](#)
 - [Bibliografía](#)
- [Single Datapath](#)
 - [Introducción](#)
 - [Fase 1.2](#)
 - [Decodificador](#)
 - [Verilog](#)
 - [Testbench](#)
 - [Conclusiones](#)
 - [Bibliografía](#)
- [Ultima fase](#)
 - [Fase final](#)
 - [Decodificador](#)
 - [Verilog](#)
 - [Testbench](#)
 - [Ejecución del algoritmo vectorial](#)
 - [Conclusiones](#)
 - [Bibliografía](#)

Universidad de Guadalajara

Piensa y trabaja

Título: Data Path

Introducción:

Teniendo conocimiento sobre lo qué es un procesador tipo MIPS, comenzaremos con la construcción del “Data Path” el cual nos ayudará a decodificar un conjunto de instrucciones tipo **r**.

Esto estará conformador por:

- ALU (Unidad Aritmética Lógica).
- ALU Control.
- Memoria (Datos).
- Banco de Registros.
- Unidad de Control.
- Conjunto de instrucciones **32 bits**.

Ejecutaremos las siguientes instrucciones:

ADD, SUB, AND, OR Y SLT.

Un conjunto de instrucciones tipo **r** consta de 32 bits, las cuales son **Aritmético-Lógicas**; se dividen en las siguientes partes:

- Opcode (6 bits): código de operación.
- RS (5 bits): espacio de memoria donde se almacenará el primer operando.
- RT (5 bits): espacio de memoria donde se almacenará el segundo operando.
- RD (5 bits): indica el espacio de memoria donde se guardarán los datos.
- Shamt (5 bits): operaciones de desplazamiento.
- Funct (6 bits): identificador de la operación aritmética a realizar.

Universidad de Guadalajara

Piensa y trabaja

SLT como operación Aritmética-Lógica (Comparación):

(si (rs < rt) **entonces** (rd = 1)

en otro caso (rd = 0)).

Operador Condicional Ternario:

Evalúa una condición y retorna el valor de una función.

```
resultado = condición ? valor_si_cierto : valor_si_falso
```

Es equivalente a un **IF-ELSE**, esto evalúa un valor booleano lo que contiene:

- True o False.
- Variable Booleana.
- Llamada a una función que devuelve un booleano.

?: si es verdad devuelve el valor verdadero.

:: “si no” devuelve el valor falso.

Proceso de Compilación:

El proceso de compilación consiste en generar un código objeto que es equivalente a un programa fuente, esto solo ocurre cuando el archivo fuente esta libre de errores.

Cuando se habla de código objeto se trata de código en ensamblador o lenguaje máquina.

Esto se divide en distintas fases o pasos:

- 1) *Preprocesador: procesa el código fuente antes de su compilación.*
- 2) *Compilador: transforma el archivo fuente en lenguaje ensamblador.*
- 3) *Ensamblador: del código ensamblador lo traduce a binario.*
- 4) *Linker: ensambla los objetos para crear un ejecutable.*

Universidad de Guadalajara

Piensa y trabaja

Fase 1 llevada a cabo a partir del lunes 22 de abril (Solo instrucciones de tipo R):

Objetivo:

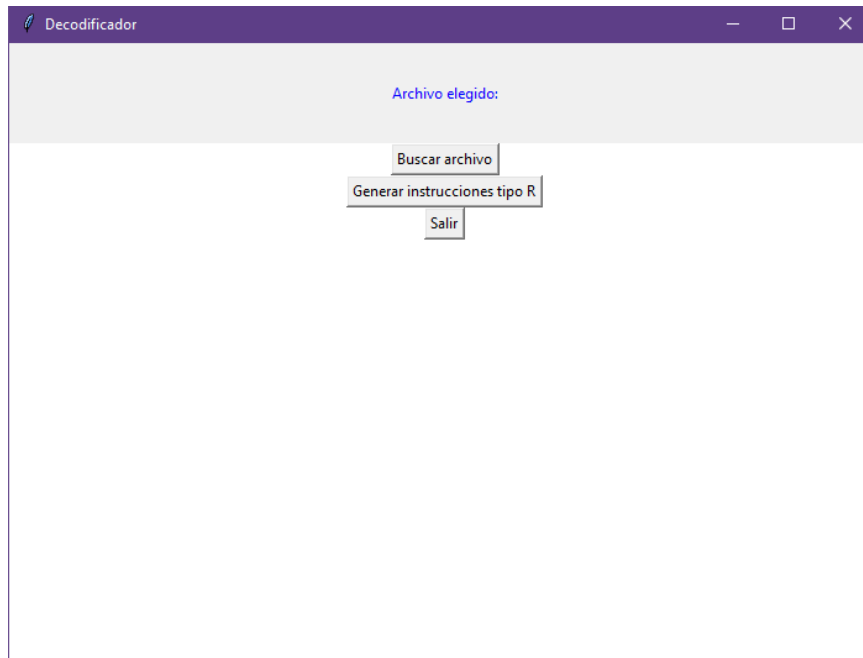
Crear un DataPath capaz de ejecutar instrucciones MIPS, en este caso las instrucciones ADD, SUB, AND, OR y SLT. Para esto, será necesario integrar algunos de los módulos ya usados en actividades previas con nuevos módulos que gestionaran el flujo de los bits en el sistema. Además se creará un decodificador, programado en Python, de lenguaje ensamblador a código binario para que las instrucciones puedan ser leídas y ejecutadas por el DataPath.

Decodificador:

Se creó el decodificador es un programa hecho con Python utilizando la librería tkinter para crear un interfaz intuitivo para el usuario. En este programa podemos seleccionar cualquier archivo .asm de nuestro PC para posteriormente decodificar las instrucciones tipo R que este contenga. Esto se hace mediante la lectura de un archivo guardando su ruta, la lectura del archivo se hace por líneas, posteriormente crea un archivo de texto instrucciones_r.txt en modo escritura para que se le permita añadir datos. Las instrucciones de las líneas leídas se separan por partes, las instrucciones cambian a binario y el diccionario decodifica la función, se terminan de procesar todas las instrucciones y se almacenan en el archivo de tipo texto.

Universidad de Guadalajara

Piensa y trabaja



Instrucciones en lenguaje ensamblador:

Estas instrucciones serán leídas por el decodificador Python al momento de insertar el archivo y generar las instrucciones de tipo R

```
instrucciones: Bloc de notas
Archivo  Edición  Formato  Ver  Ayuda
add $22 $0 $1
sub $23 $2 $3
and $24 $4 $5
or $25 $6 $7
slt $26 $8 $9
add $27 $10 $11
sub $28 $12 $13
and $29 $14 $15
or $30 $16 $17
slt $31 $18 $19
```

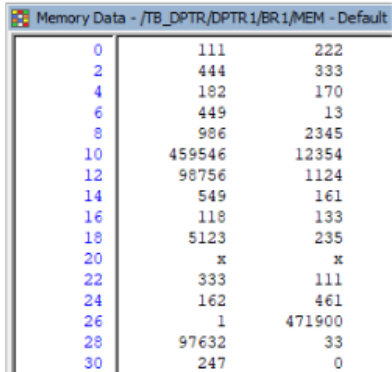
Las cuales, ordenadas quedan de la siguiente manera:

Instruccion ASM	rs	Dato 1	rt	Dato 2	rd	Resultado
add \$21 \$0 \$1	0	111	1	222	21	333
sub \$22 \$2 \$3	2	444	3	333	22	111
and \$23 \$4 \$5	4	182	5	170	23	162
or \$24 \$6 \$7	6	449	7	13	24	461
slt \$25 \$8 \$9	8	986	9	2345	25	1
add \$26 \$10 \$11	10	459546	11	12354	26	471900
sub \$27 \$12 \$13	12	98756	13	1124	27	97632
and \$28 \$14 \$15	14	549	15	161	28	33
or \$29 \$16 \$17	16	118	17	133	29	247
slt \$30 \$18 \$19	18	5123	19	235	30	0

Universidad de Guadalajara

Piensa y trabaja

Dividida en los datos de entrada y datos de salida en la simulación de la misma:



0	111	222
2	444	333
4	182	170
6	449	13
8	986	2345
10	459546	12354
12	98756	1124
14	549	161
16	118	133
18	5123	235
20	x	x
22	333	111
24	162	461
26	1	471900
28	97632	33
30	247	0

Verilog:

Banco de Registro:

El presente código define un módulo llamado "BancoDeRegistro" que incluye varios puertos de entrada y salida. Los puertos de entrada son dos registros de dirección de 5 bits (RA1 y RA2), un dato de entrada de 32 bits (Di), una dirección de registro de 5 bits (Dir) y una señal de escritura de registro (RegWrite). Los puertos de salida son dos datos de registro de 32 bits (DR1 y DR2).

Dentro del módulo se declara una memoria llamada "MEM" que consiste en un array de 32 registros de 32 bits cada uno.

En el bloque de procesamiento, se utiliza un proceso combinacional (`always @*) que se activa cada vez que cambia alguna de las entradas. Si la señal de escritura de registro (RegWrite) está activa, se escribe el dato de entrada (Di) en la dirección especificada por Dir en la memoria MEM. Luego, se asigna a DR1 y DR2 los datos almacenados en las direcciones especificadas por RA1 y RA2 respectivamente, permitiendo la lectura de datos desde la memoria en función de las direcciones proporcionadas por los registros RA1 y RA2.

ALU:

El presente código describe un módulo ALU (Unidad Aritmético Lógica) que realiza operaciones aritméticas y lógicas básicas entre dos operandos de 32 bits (operador1 y operador2) utilizando un selector para elegir la operación deseada. El módulo tiene una salida para el resultado de la operación y una salida para la bandera de cero (ZF).

Dentro del módulo se definen varios cables (AND, OR, add, subtract, setOnLessThan, NOR) que representan los resultados de diferentes operaciones entre los operandos.

En el bloque de procesamiento, se utilizan asignaciones para calcular los resultados de las operaciones lógicas y aritméticas (AND, OR, suma, resta, setOnLessThan, NOR) entre los operandos.

Universidad de Guadalajara

Piensa y trabaja

El bloque ``always @(*)`` se activa cada vez que cambia alguna de las entradas. Utiliza un caso (case) con el selector para determinar qué operación realizar. Dependiendo del valor del selector, se asigna el resultado correspondiente a la salida "resultado".

La asignación de ZF se utiliza para determinar la bandera de cero (ZF), que se activa cuando el resultado es cero.

MEM:

El código describe un módulo de memoria llamado "Mem" que contiene puertos de entrada para la escritura habilitada (Ewr), dirección de memoria (Dir), dato de entrada (Din) y un puerto de salida para el dato leído (Dout).

Dentro del módulo se declara un array llamado "A" que representa la memoria, con capacidad para almacenar 256 registros de 32 bits cada uno.

En el bloque de procesamiento, se utiliza un proceso combinacional (`always @(*)`) que se activa cada vez que cambia alguna de las entradas. Si la señal de escritura habilitada (Ewr) está activa, se escribe el dato de entrada (Din) en la dirección de memoria especificada por Dir en el array "A". Si la señal de escritura no está activa, se asigna a Dout el dato almacenado en la dirección especificada por Dir en el array "A", permitiendo la lectura de datos desde la memoria.

Unidad De Control:

El código describe un módulo de unidad de control que toma un campo de operación (op) de 6 bits como entrada y genera varias señales de control como salida para controlar el flujo de datos y operaciones en un sistema.

El módulo tiene como salidas regidas (output reg) las siguientes señales de control:

- MemToReg: Indica si se debe escribir en el registro desde la memoria.
- MemToWrite: Indica si se debe escribir en la memoria.
- ALUOp: Indica la operación a realizar en la ALU (Unidad Aritmético Lógica).
- RegWrite: Indica si se debe escribir en el registro.

Dentro del bloque de procesamiento (`always @(*)`), se utiliza un caso (case) para evaluar el campo de operación (op). Dependiendo del valor de op, se asignan valores a las señales de control según la operación especificada. En el caso por defecto, se asignan valores por defecto a las señales de control para operaciones no especificadas explícitamente en el código.

ALU Control:

El código describe un módulo de control de ALU (Unidad Aritmético Lógica) que toma dos entradas, Func y InOp, para determinar la operación de la ALU a realizar. La salida outOp especifica la operación que la ALU debe llevar a cabo.

Universidad de Guadalajara

Piensa y trabaja

Dentro del módulo, se utiliza un proceso combinacional (always @(*)) que se activa cuando cambia alguna de las entradas. Se utiliza un caso (`case`) para evaluar el valor de InOp y luego otro caso interno para evaluar el valor de Func.

Si InOp es 000, se determina la operación de la ALU basada en el valor de Func:

- 100000: outOp = 0010 (suma)
- 100010: outOp = 0110 (resta)
- 100100: outOp = 0000 (AND lógico)
- 100101: outOp = 0001 (OR lógico)
- 101010: outOp = 0111 (Establecer si Menor Que - STL)
- En cualquier otro caso, outOp se establece en 1111 como valor predeterminado.
- Si InOp no es 000, outOp se establece en 1111 como valor predeterminado.

Esta estructura permite controlar la operación de la ALU de manera selectiva dependiendo de las señales de entrada Func e InOp.

MUX 2_1 32bits:

El código describe un módulo Mux2_1_32 que implementa un multiplexor de 2 a 1 de 32 bits. Toma tres entradas: MemToReg, Op1 y Op2. Dependiendo del valor de MemToReg, el módulo selecciona la salida outOp entre Op1 y Op2.

Dentro del módulo, se utiliza un proceso combinacional (`always @(*)`) que se activa cuando cambia alguna de las entradas. Si MemToReg es verdadero (1), la salida outOp se asigna al valor de Op1; de lo contrario, outOp se asigna al valor de Op2. Esto implementa la funcionalidad básica de un multiplexor, seleccionando una de las dos entradas basada en una señal de control.

Data Path T-R:

Este código describe un módulo llamado "DataPathT_R" que implementa un camino de datos para una arquitectura de computadora. El módulo tiene una entrada InstruccionTR de 32 bits que representa una instrucción y una salida TR_ZF que indica el resultado de una operación en la ALU.

Dentro del módulo se definen varios cables (C1 a C6) que conectan los diferentes componentes del camino de datos. Se instancian varios módulos internos para realizar operaciones específicas:

1. Unidad_de_Control (UDC): Controla las señales de control para las operaciones en el camino de datos basadas en la instrucción recibida.
2. BancoDeRegistro (BR1): Implementa un banco de registros para almacenar y leer datos.

Universidad de Guadalajara

Piensa y trabaja

3. ALU_Control (AluC): Controla la operación de la ALU basada en la función especificada en la instrucción.
4. _ALU (Alu1): Realiza operaciones aritméticas o lógicas entre dos operandos.
5. Mem (Mem1): Simula una memoria para almacenamiento de datos.
6. Mux2_1_32 (mux1_2): Implementa un multiplexor para seleccionar entre dos entradas basado en una señal de control.

En esencia, este módulo representa la interconexión de diferentes componentes de hardware (ALU, registros, memoria) para ejecutar instrucciones de un programa en una arquitectura de computadora. Cada componente realiza su función específica y se conecta a otros componentes según las necesidades de la instrucción en curso.

Test Bench:

TB Unidad de Control:

Este código describe un módulo de simulación llamado "TB_Unidad_control" que prueba el funcionamiento de la unidad de control "Unidad_de_Control" (UC1) utilizando una serie de operaciones de control definidas por la señal TB_op.

1. Se declara una serie de señales cableadas (wires) que se conectarán a las salidas de la unidad de control UC1: TB_MemToReg, TB_MemToWrite, TB_ALUOp y TB_RegWrite.
2. Se instancia la unidad de control UC1, conectando las señales de entrada y salida correspondientes.

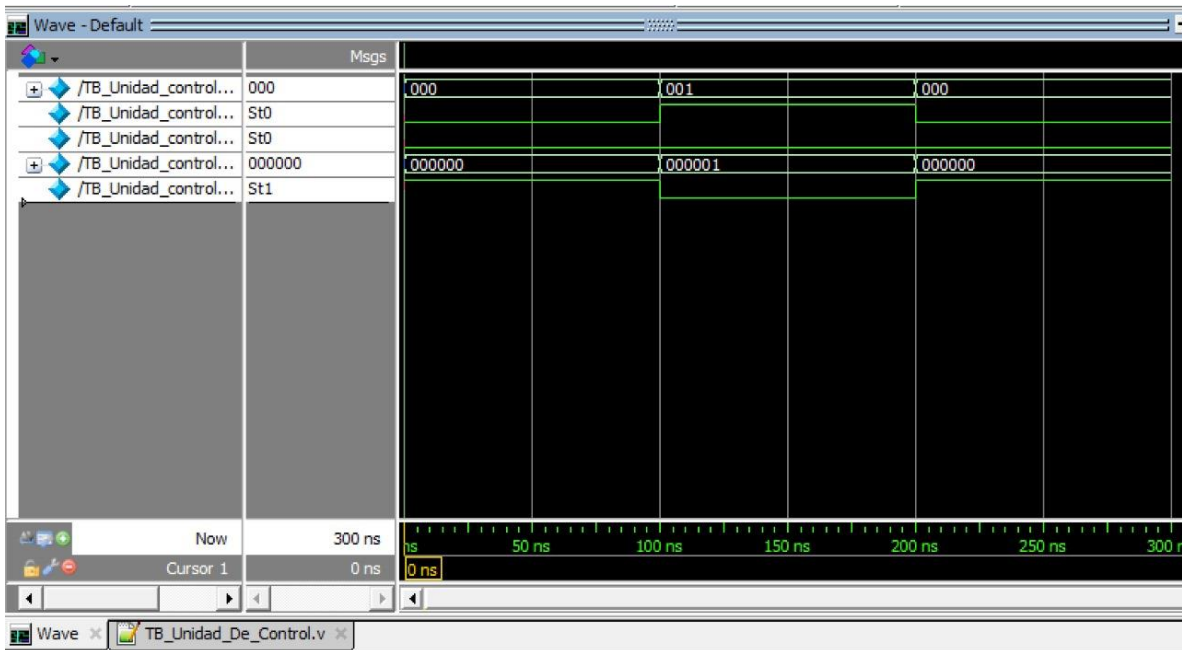
Dentro del bloque "initial", se inicializa la señal TB_op con diferentes valores para simular distintas operaciones de control:

- Se inicia con TB_op = 000000 (en binario), luego se espera 100 unidades de tiempo (#100).
- Se cambia TB_op a 000001 y se espera nuevamente 100 unidades de tiempo.
- Se vuelve a TB_op = 000000 y se espera otros 100 unidades de tiempo.
- Finalmente, se detiene la simulación con \$stop.
- Estas operaciones de cambio en TB_op simulan diferentes configuraciones de control en la unidad de control UC1 a lo largo del tiempo de simulación, lo que permite verificar su funcionamiento bajo diferentes condiciones de entrada.

Universidad de Guadalajara

Piensa y trabaja

Simulación:



TB ALU Control:

Este código describe un módulo de simulación llamado "TB_ALU_Control" que prueba el funcionamiento del módulo de control de ALU "ALU_Control" (ALU_C1) utilizando una serie de operaciones definidas por las señales TB_Func y TB_InOp.

1. Se declaran las señales TB_Func y TB_InOp como registros (reg) para simular las entradas a la ALU_Control, y se declara TB_outOP como un cable (wire) para capturar la salida de la ALU_Control ALU_C1.
2. Se instancia la ALU_Control ALU_C1, conectando las señales de entrada y salida correspondientes.

Dentro del bloque "initial", se definen diferentes valores para TB_Func y TB_InOp en secuencia para simular diferentes operaciones de la ALU_Control a lo largo del tiempo de simulación. Después de cambiar las señales de entrada, se espera 100 unidades de tiempo (#100) antes de realizar el siguiente cambio. Esto simula cambios en las operaciones de la ALU_Control y permite verificar su funcionamiento bajo diferentes configuraciones de entrada.

Las operaciones simuladas son:

- TB_Func = 100000, TB_InOp = 000 (suma)
- TB_Func = 100000, TB_InOp = 001 (resta)
- TB_Func = 100010, TB_InOp = 000 (AND lógico)
- TB_Func = 100100, TB_InOp = 000 (OR lógico)
- TB_Func = 100101, TB_InOp = 000 (STL - Establecer si Menor Que)
- TB_Func = 101010, TB_InOp = 000 (operación no especificada)

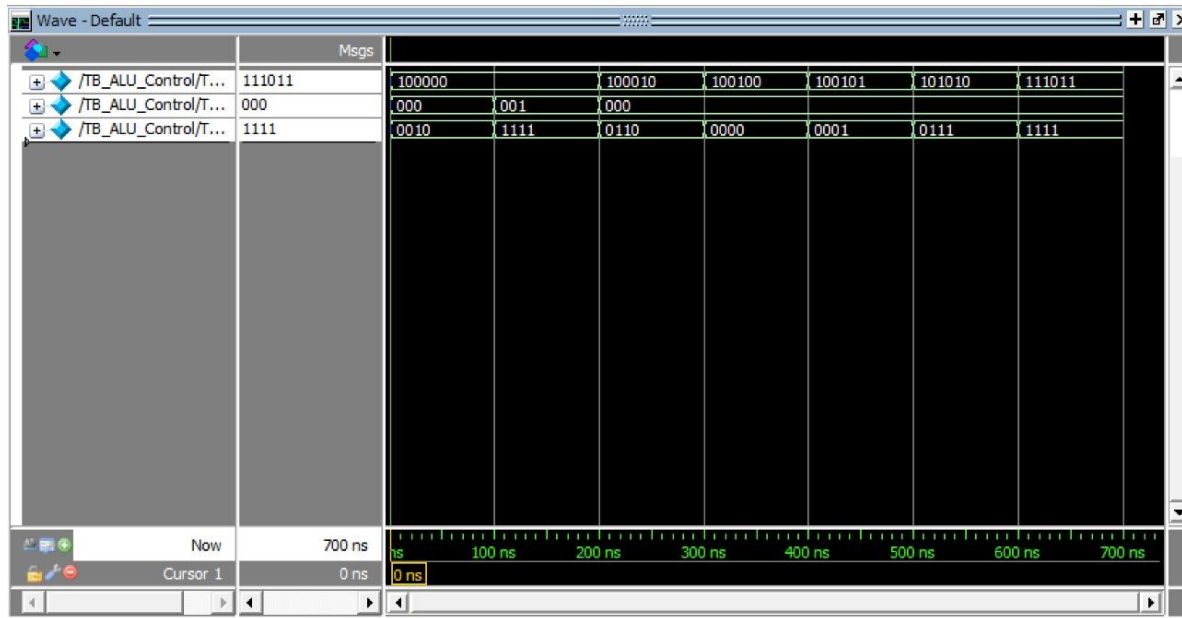
Universidad de Guadalajara

Piensa y trabaja

- TB_Func = 111011, TB_InOp = 000 (operación no especificada)

Esto te permite probar el comportamiento de la ALU_Control bajo diversas operaciones y verificar cómo responde a diferentes combinaciones de entradas.

Simulación:



TB_MUX 2_1 32bits:

Este código describe un módulo de simulación llamado "TB_Mux2_1_32" que prueba el funcionamiento de un multiplexor de 2 a 1 de 32 bits ("Mux2_1_32") utilizando diferentes configuraciones de entrada.

1. Se definen las señales TB_MemToReg, TB_Op1 y TB_Op2 como registros (reg) para simular las entradas al multiplexor, y se declara TB_outOp como un cable (wire) para capturar la salida del multiplexor.
2. Se instancia el multiplexor Mux2_1_32 (mux1_2), conectando las señales de entrada y salida correspondientes.

Dentro del bloque "initial", se realizan cambios en las señales de entrada del multiplexor en secuencia para simular diferentes configuraciones de entrada y verificar el comportamiento del multiplexor bajo estas condiciones. Después de cada cambio, se espera 100 unidades de tiempo (#100) antes de realizar el siguiente cambio.

Las operaciones simuladas son:

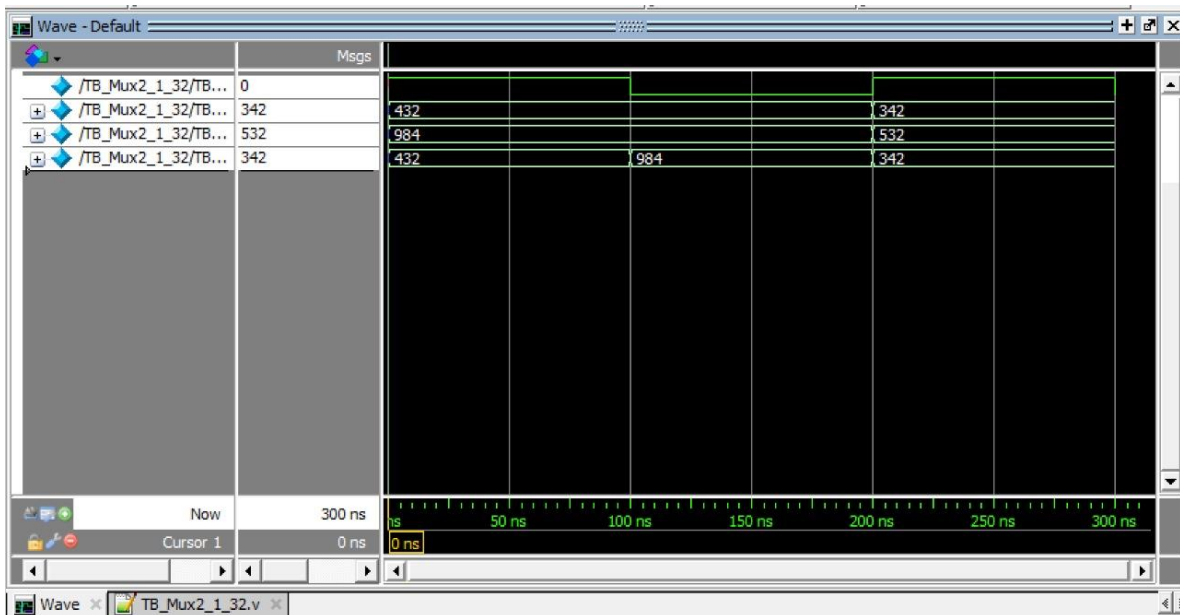
Universidad de Guadalajara

Piensa y trabaja

1. Seleccionar Op1 como salida (TB_MemToReg = 1, TB_Op1 = 432, TB_Op2 = 984).
2. Seleccionar Op2 como salida (TB_MemToReg = 0, TB_Op1 = 432, TB_Op2 = 984).
3. Seleccionar Op1 como salida (TB_MemToReg = 1, TB_Op1 = 342, TB_Op2 = 532).
4. Seleccionar Op2 como salida (TB_MemToReg = 0, TB_Op1 = 342, TB_Op2 = 532).

Después de realizar estas operaciones, se detiene la simulación con \$stop para finalizar la simulación después de probar las diferentes configuraciones de entrada y salida del multiplexor.

Simulación:



TB Data Path T-R:

Este código describe un módulo de simulación llamado "TB_DPTR" que prueba el funcionamiento de un camino de datos "DataPathT_R" (DPTR1) utilizando un archivo de instrucciones para cargar y ejecutar las instrucciones en el camino de datos.

1. Se declara la señal TB_InstruccionTR como un registro (reg) para simular las instrucciones que se cargarán en el camino de datos, y se declara TB_TR_ZF como un cable (wire) para capturar la salida de la instrucción ejecutada.
2. Se instancia el camino de datos DPTR1, conectando la señal de entrada InstruccionTR y la señal de salida TR_ZF.

Universidad de Guadalajara

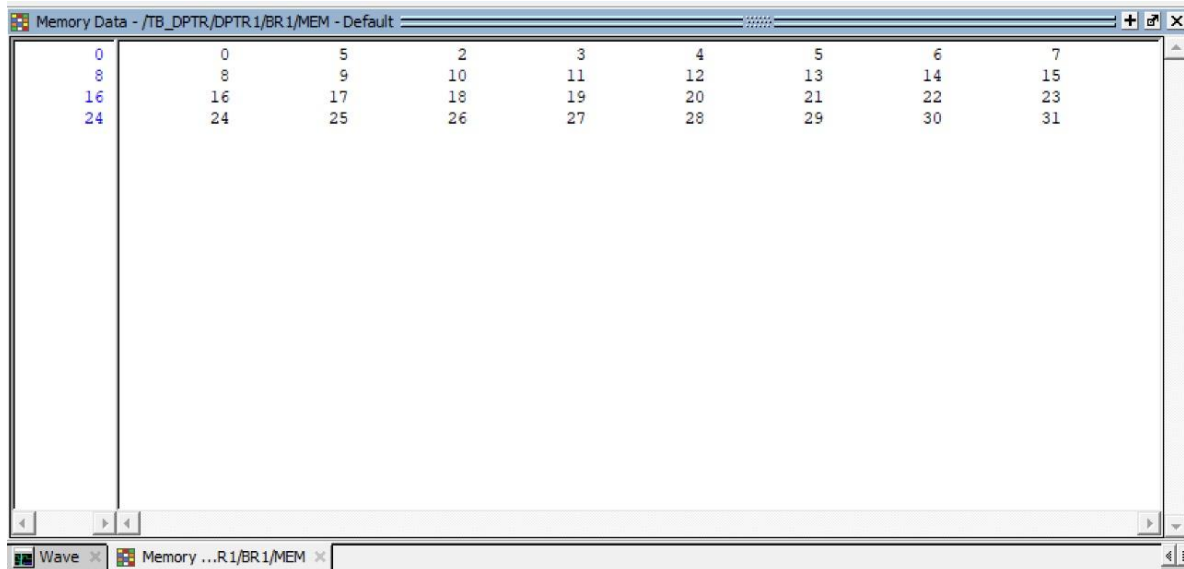
Piensa y trabaja

Dentro del bloque "initial":

- Se lee el contenido del archivo "DataTB.txt" en la memoria del banco de registros (MEM) del camino de datos DPTR1. Esto se realiza utilizando la función ``$readmemb``.
- Se espera 100 unidades de tiempo (#100) después de cargar la memoria con los datos del archivo.
- Se abre el archivo "instrucciones_r.txt" en modo lectura y se verifica si se ha abierto correctamente.
- Si el archivo se abre correctamente, se procede a leer cada instrucción binaria del archivo y se carga en la señal TB_InstruccionTR. Se espera 100 unidades de tiempo después de cada lectura.
- Se cierra el archivo después de leer todas las instrucciones.
- Si no se puede abrir el archivo de instrucciones, se muestra un mensaje de error en la consola.

Este código simula la carga de instrucciones desde un archivo y su ejecución en un camino de datos simulado, lo que permite probar el comportamiento del camino de datos bajo diferentes secuencias de instrucciones.

Simulación DPTR-1:



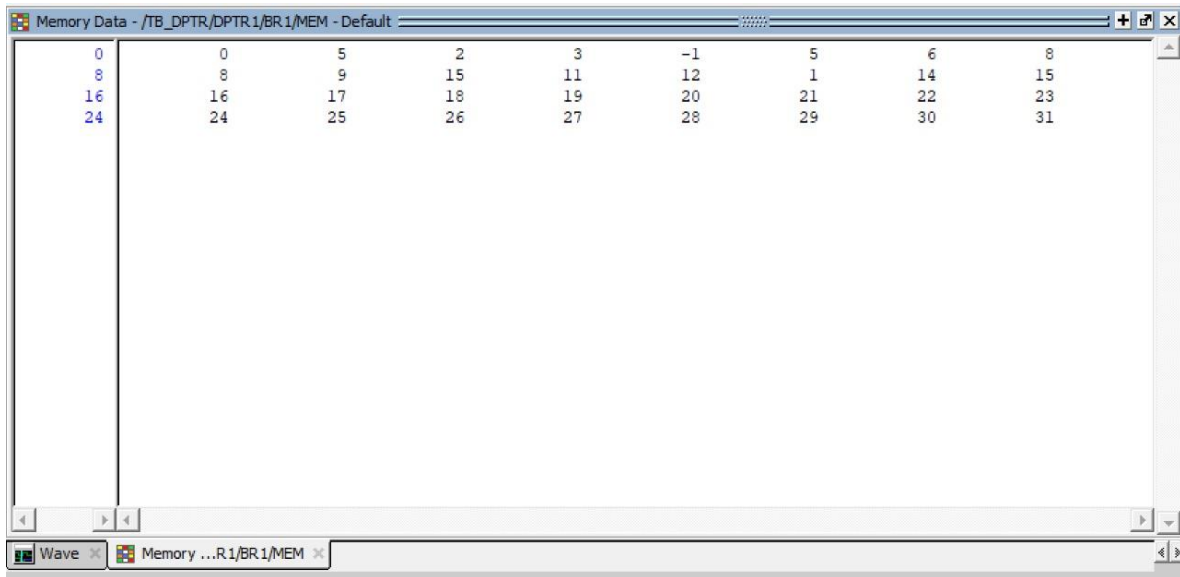
The screenshot shows a window titled "Memory Data - /TB_DPTR/DPTR1/BR1/MEM - Default". The window displays a memory dump with 4 rows and 8 columns. The first column contains addresses (0, 8, 16, 24) and the subsequent columns contain hexadecimal values. The values in the first row are 0, 5, 2, 3, 4, 5, 6, 7. The values in the second row are 8, 9, 10, 11, 12, 13, 14, 15. The values in the third row are 16, 17, 18, 19, 20, 21, 22, 23. The values in the fourth row are 24, 25, 26, 27, 28, 29, 30, 31.

0	0	5	2	3	4	5	6	7
8	8	9	10	11	12	13	14	15
16	16	17	18	19	20	21	22	23
24	24	25	26	27	28	29	30	31

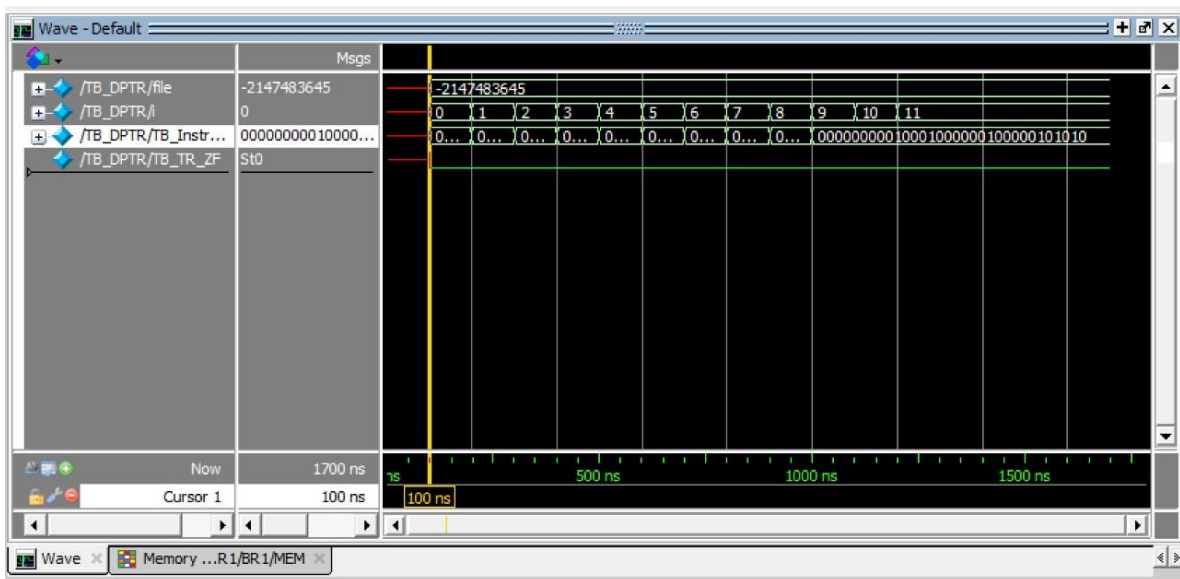
Simulación DPTR-2:

Universidad de Guadalajara

Piensa y trabaja



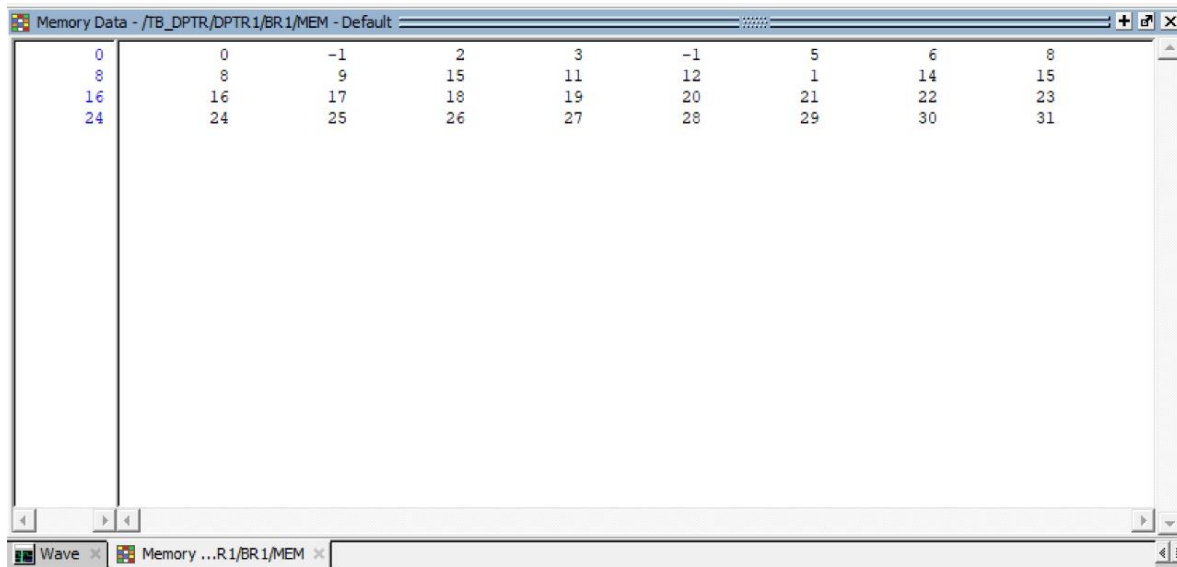
Simulación DPTR-3:



Simulación DPTR-4:

Universidad de Guadalajara

Piensa y trabaja

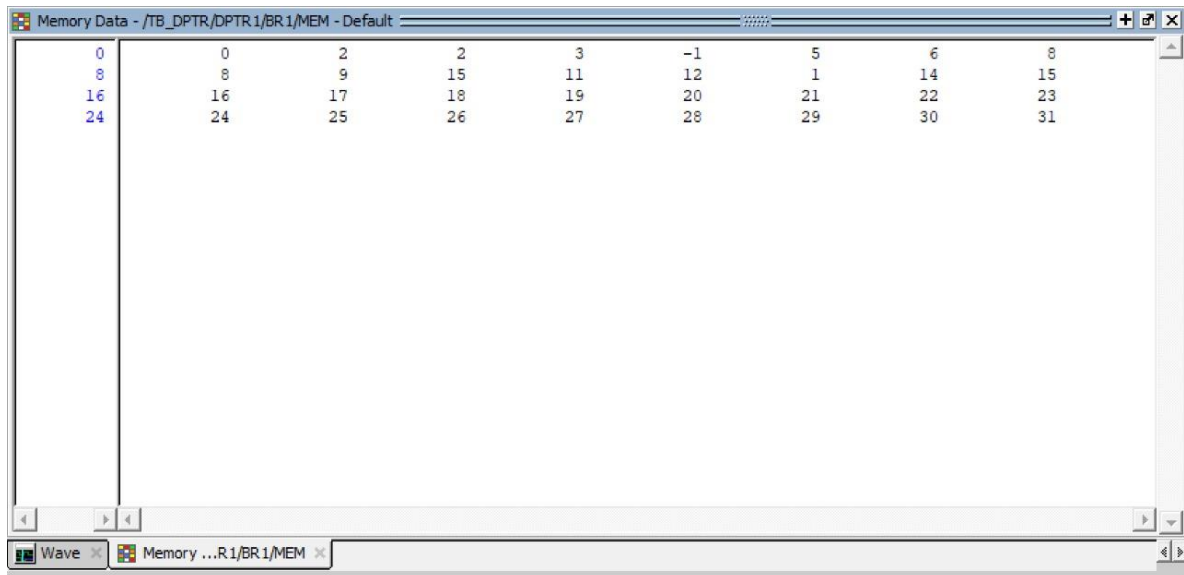


Memory Data - /TB_DPTR/DPTR1/BR1/MEM - Default

0	0	-1	2	3	-1	5	6	8
8	8	9	15	11	12	1	14	15
16	16	17	18	19	20	21	22	23
24	24	25	26	27	28	29	30	31

Wave Memory ...R1/BR1/MEM

Simulación DPTR-5:



Memory Data - /TB_DPTR/DPTR1/BR1/MEM - Default

0	0	2	2	3	-1	5	6	8
8	8	9	15	11	12	1	14	15
16	16	17	18	19	20	21	22	23
24	24	25	26	27	28	29	30	31

Wave Memory ...R1/BR1/MEM

Simulación DPTR-7:

Universidad de Guadalajara

Piensa y trabaja

0	0	3	2	3	-1	5	6	8
8	8	9	15	11	12	1	14	15
16	16	17	18	19	20	21	22	23
24	24	25	26	27	28	29	30	31

Simulación DPTR-8:

0	0	1	2	3	-1	5	6	8
8	8	9	15	11	12	1	14	15
16	16	17	18	19	20	21	22	23
24	24	25	26	27	28	29	30	31

Universidad de Guadalajara

Piensa y trabaja

Conclusiones:

Santiago Rafael Gómez Brizuela:

“El decodificador es muy importante para este proyecto ya que de aquí saldrán las instrucciones que le darán vida a nuestra simulación en Verilog, estás indicaran que datos serán leídos, como se deben procesar y como se deben de guardar, me gustó darle inicio al decodificador y usar Python nuevamente sin duda uno de mis lenguajes favoritos.”

Denice Estefania Rico Morones:

“Cómo conclusión puedo decir qué es de suma importancia contar con la documentación adecuada en un proyecto ya que nos ayuda a comprender qué es lo que estamos haciendo e ir documentando los cambios en este; además, la documentación facilita la colaboración y la comunicación entre los miembros del equipo. Cuando alguien necesita entender cómo funciona una parte específica del proyecto o cómo se realizó un cambio en el código, puede recurrir a la documentación.”

Ferran Prado Roesner:

“En esta primera fase del proyecto se logró la gestión de tiempo y recursos efectiva y un buen trabajo colaborativo. Cada miembro del equipo, compuesto solo por cuatro integrantes, tuvo roles muy importantes: desde la codificación en Python y Verilog hasta el trabajo de documentación y la gestión del proyecto. En esta actividad pude ver la importancia de una buena organización y un buen reparto de tareas. Este proyecto me ayudó a entender cómo cada componente y cada parte individual son importantes para el éxito del proyecto.”

Universidad de Guadalajara

Piensa y trabaja

Alan Emmanuel Marin Lemus:

“En Conclusión, el desarrollo de esta fase tomo varios desafíos que involucraron investigar huecos en nuestros conocimientos tanto de Python como el funcionamiento de las instrucciones tipo R, lo cual ayudo a terminar esta fase con éxito, cumpliendo con los objetivos planteados y logrando producir un módulo capaz de ejecutar instrucciones de tipo R de forma correcta.”

Bibliografía:

Práctica 13. La unidad aritmético lógica (ALU). (s. f.).
https://www.hpca.ual.es/~vruiz/docencia/laboratorio_arquitectura/practicas/practica_13/

Llamas, L. (2023, 1 junio). El operador ternario. Luis Llamas.
<https://www.luisllamas.es/programacion-operador-ternario/>

Prezi, M. C. O. (s. f.). PROCESO DE COMPILACION. prezi.com.
<https://prezi.com/qqn12utcx1yu/proceso-de-compilacion/>

Manual de Dev-C++ en español. (s. f.).
https://www.tel.uva.es/personales/josdie/fprog/Sesiones/manualDevCpp/compilacion_y_linkado.html

Universidad de Guadalajara

Piensa y trabaja

Título: single datapath

Introducción: teniendo en cuenta que ya contamos con un data path creado y más conocimiento en los procesadores MIPS, comenzaremos con la creación de un single data path, a diferencia de un data path sencillo, el single data path utiliza un solo camino para la transferencia de datos dentro del procesador, en otras palabras que consta de una ruta única.

Además de lo ya construido en el data path al single data path le agregamos:

- Memoria de instrucciones
- Un multiplexor de 32 bits
- Unidad de suma o ADD
- Y el contador o PC



Ljubisa Bajic

Ljubisa Bajic es el CEO y el arquitecto principal de Tenstorrent Inc. empresa diseñadora de procesadores de inteligencia artificial, anteriormente trabajaba en AMD y ATI

Tenstorrent desarrolla procesadores IP como el Tensix IA IP o el IP de CPU RISC-v, diferentes tipos de tarjetas de desarrollo y además son desarrolladores de 2 softwares llamados TT-Buda y TT-Metalio.

Jim Keller

Es un ingeniero estadounidense que a trabajado en AMD, Apple y Tesla, fue el arquitecto principal en la creación de la microarquitectura del procesador K8 de AMD así como también el Athlon 64, y participo en el K7 así como el K12 y el ZEN, con Apple participo en la creación de los procesadores A4 y A5, es coautor de las instrucciones x86-64 y la interconexión HyperTransport



Universidad de Guadalajara

Piensa y trabaja



Rajabali Makaradhwaja Koduri

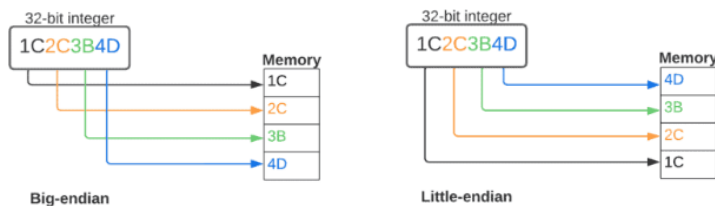
Rajabali Makaradhwaja Koduri es un ingeniero informático indio y ejecutivo de hardware de gráficos por computadora, fue el arquitecto y vicepresidente ejecutivo de IAGS de Intel también apporto en los procesadores de Radeon.

Big Endian, little endian (endianess)

En los sistemas informático el byte es la estructura principal lo que en diferentes formatos puede causarnos problemas generando así endianida, que no es más que el problema generado cuando un archivo almacena datos y otro lo quiere leer, la solución fue tomar un formato estándar y siempre usar un encabezado que define el formato de los datos, si este encabezado esta al revés el sistema deberá convertirlo.

Las 2 formas principales de formato son el Big-endian y Little-endian, el Big-endian manda el byte más significativo A la ubicación de memoria más pequeña, y el menos significativo a la más grande, en cambio el Little-endian manda la dirección menos significativa a la ubicación más pequeña.

Además de estos 2 existen más como el middle-endian o mix-endian pero los más comunes son el Big y el Little endian, dichos términos se los dio Danny Cohen quien es muy importante en la separación del TCP y IP.



El formato usado por nosotros fue Big endian

Universidad de Guadalajara

Piensa y trabaja

Modelos de arquitectura de computadoras

Existen muchas arquitecturas diferentes cada una de ellas con sus características principales, entre las más populares podemos encontrar

1. Arquitectura de Von Neumann: propuesta en 1940 por John Von Neumann es una arquitectura muy importante en el campo, fue usada en la creación de la computadora EDVAC, la cual es fundamento para muchos ordenadores modernos, para crear dicha su idea principal era la creación de una unidad central de procesamiento la cual tiene acceso a una memorias para almacenar datos y/o programas, las instrucciones son y datos se depositan en la memoria y posteriormente se recuperan a ttaves de un canal común.
2. Arquitectura Harvard: Parecida la arquitectura de Von Neumann con la diferencia de que la arquitectura Harvard utiliza memorias separadas para almacenar instrucciones y datos de forma independiente, esto mejora el rendimiento en ciertas aplicaciones ya que permite que la CPU acceda de manera simultánea a la memorias.
3. Arquitectura RISC: la arquitectura RISC o Reduced Instruction Set Computer es una arquitectura que a diferencia de otras utiliza un set de instrucciones reducido y muy optimizado, esta arquitectura utiliza un ciclo de reloj para ejecutar sus instrucciones volviéndose más eficiente al momento de ejecutar operaciones repetitivas y simples, suele usarse en aplicaciones que necesitan un procesamiento intensivo, es la base de muchos procesadores modernos
4. Arquitectura CISC: la arquitectura CISC o Complex Instruction Set Computer utiliza un set de instrucciones más complejo, por ende la arquitectura CISC es capaz de realizar tareas más difíciles en un solo ciclo, por ende es fácil de programas, mas no tan eficiente por dicha razón se han creado diversas técnicas para mejorar su eficiencia como la segmentación o la ejecución fuera de orden

Universidad de Guadalajara

Piensa y trabaja

5. Arquitectura Paralela: se basa en utilizar diversas unidades de procesamiento trabajando al mismo tiempo para realizar más tareas y operaciones al mismo tiempo, esto se logra con procesadores multinucleos o un sistema de muchos procesadores, la idea principal para su creación fue la de dividir tareas en secciones más pequeñas y repartirlas entre los procesadores
6. Arquitectura de la computación en la nube: esta arquitectura es una estructura tecnológica que permite la difusión a recursos informáticos por medio de internet, sin la necesidad de que los usuarios necesiten tener o administrar de manera física los equipos y servidores

Procesador MIPS

MIPS o microprocessro without interlocked pipeline stages es la forma en la que se les conoce a una familia procesadores de arquitectura RISC o Reduced Instruction Set Computer, los procesadores MIPS fueron creados por MIPS technologies

El primer modelo comercial fue el R2000 en 1985 en la siguiente tabla encontraras más modelos:

Modelo	Frecuencia [MHz]	Año	Proceso [µm]	Transistores [millones]	Tamaño del chip [mm²]	Pins E/S	Potencia [W]	Voltaje	Dcache [k]	Icache [k]	Scache [k]
R2000	8-16,7	1985	2	0,11	--	--	--	--	32	64	none
R3000	12-40	1988	1,2	0,11	66,12	145	4	--	64	64	none
R4000	100	1991	0,8	1,35	213	179	15	5	8	8	1024
R4400	100-250	1992	0,6	2,3	186	179	15	5	16	16	1024
R4600	100-133	1994	0,64	2,2	77	179	4,6	5	16	16	512
R5000	150-200	1996	0,35	3,7	84	223	10	3,3	32	32	1024
R8000	75-90	1994	0,5	2,6	299	591	30	3,3	16	16	1024
R10000	150-250	1995	0,35	6,8	299	599	30	3,3	32	32	512
R12000	270-400	1998	0,18–0,25	6,9	204	600	20	4	32	32	1024
R14000	500-600	2001	0,13	7,2	204	527	17	--	32	32	2048
R16000	700-800	2002	0,11	--	--	--	20	--	64	64	4096

Universidad de Guadalajara

Piensa y trabaja

La longitud de sus instrucciones son de 32 bits, y su largo de palabra siempre será de 4 bytes (4^8).

Las instrucciones más comunes para la arquitectura MIPS son las siguientes

- Instrucciones aritméticas: ADD, SUB, MUL, DIV
- Instrucciones lógicas: AND, OR, XOR, NOR
- Instrucciones de transferencia de datos: LW (Load Word), SW (Store Word)
- Instrucciones de salto: J (Jump), JAL (Jump and Link)
- Instrucciones de comparación: SLT (Set Less Than), BEQ (Branch if Equal)

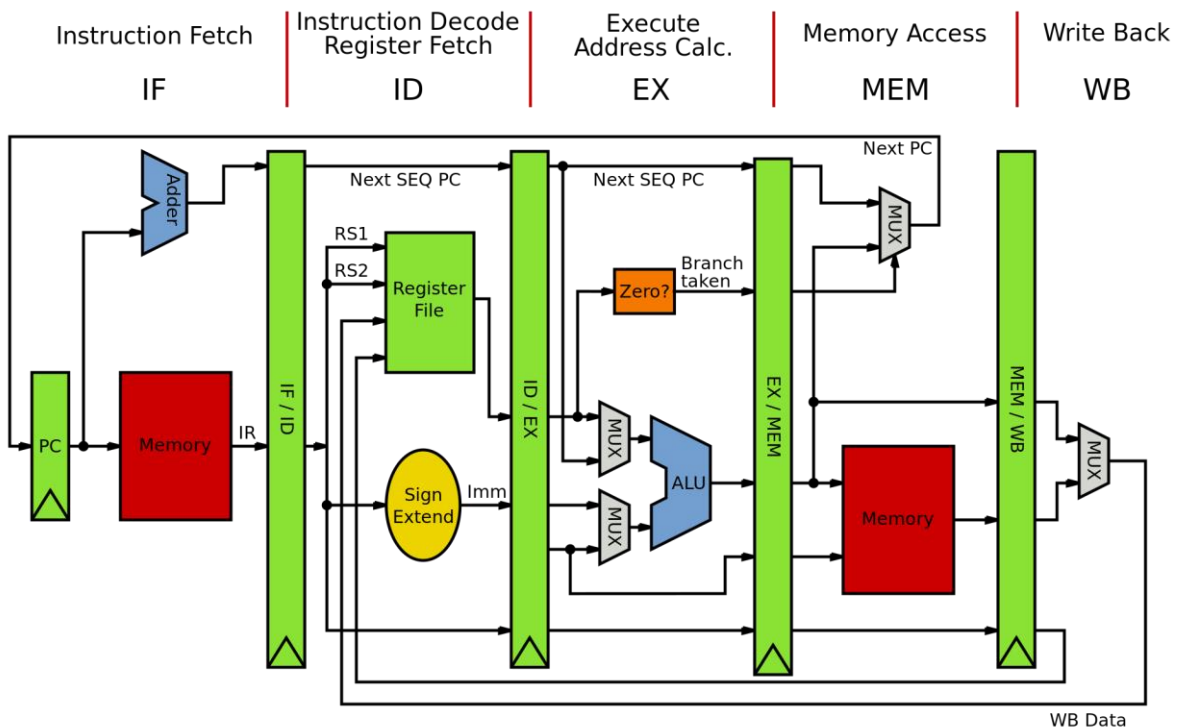
Características de los Tipos de instrucciones (R, I, J):

- R-Type: Son instrucciones que involucran tres registros y se utilizan para operaciones aritméticas y lógicas. Tienen un formato de 6 campos.
- I-Type: Estas instrucciones usan dos registros y un valor inmediato. Se utilizan principalmente para operaciones de transferencia de datos y comparación. Tienen un formato de 4 campos.
- J-Type: Son instrucciones de salto incondicional. Utilizan una dirección de destino y tienen un formato de 2 campos.

Universidad de Guadalajara

Piensa y trabaja

Instrucción	Sintaxis	Tipo	Código de operación	Código de ALU	Operación en ALU
ADD	ADD \$rd, \$rs, \$rt	R	000000	100000	Suma
SUB	SUB \$rd, \$rs, \$rt	R	000000	100010	Resta
AND	AND \$rd, \$rs, \$rt	R	000000	100100	AND lógico
OR	OR \$rd, \$rs, \$rt	R	000000	100101	OR lógico
LW	LW \$rt, offset(\$rs)	I	100011	-	Carga palabra
SW	SW \$rt, offset(\$rs)	I	101011	-	Almacena palabra
J	J target	J	000010	-	Salto incondicional
BEQ	BEQ \$rs, \$rt, offset	I	000100	-	Branch si igual
SLT	SLT \$rd, \$rs, \$rt	R	000000	101010	Set si menor
XOR	XOR \$rd, \$rs, \$rt	R	000000	100110	XOR lógico
ADDI	ADDI \$rt, \$rs, immediate	I	001000	100000	Suma inmediata
ORI	ORI \$rt, \$rs, immediate	I	001101	100101	OR inmediato
SLL	SLL \$rd, \$rt, shamt	R	000000	000000	Desplazamiento izq.
SRL	SRL \$rd, \$rt, shamt	R	000000	000010	Desplazamiento der.
SLTI	SLTI \$rt, \$rs, immediate	I	001010	101010	Set si menor inmediato
JAL	JAL target	J	000011	-	Salto y link
ANDI	ANDI \$rt, \$rs, immediate	I	001100	100100	AND inmediato
BEQZ	BEQZ \$rs, offset	I	000100	-	Branch si cero
BNE	BNE \$rs, \$rt, offset	I	000101	-	Branch si no igual
JR	JR \$rs	R	000000	001000	Salto registro
MUL	MUL \$rd, \$rs, \$rt	R	000000	000010	Multiplicación



Universidad de Guadalajara

Piensa y trabaja

Fase 1.2 llevada a cabo a partir del lunes 06 de mayo single datapath (Solo instrucciones de tipo R):

Objetivo:

El objetivo de esta fase es la creación de un single datapath basado en lo que teníamos construido anteriormente en el datapath, que igualmente será capaz de ejecutar instrucciones MIPS de tipo R, además en esta fase modificaremos el decodificador de Python para que permita una lectura mejor de las instrucciones así como el poder editarlas desde la interfaz.

Decodificador:

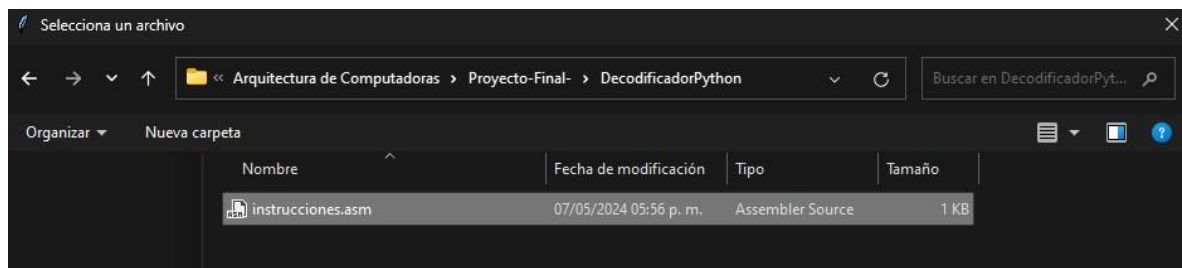
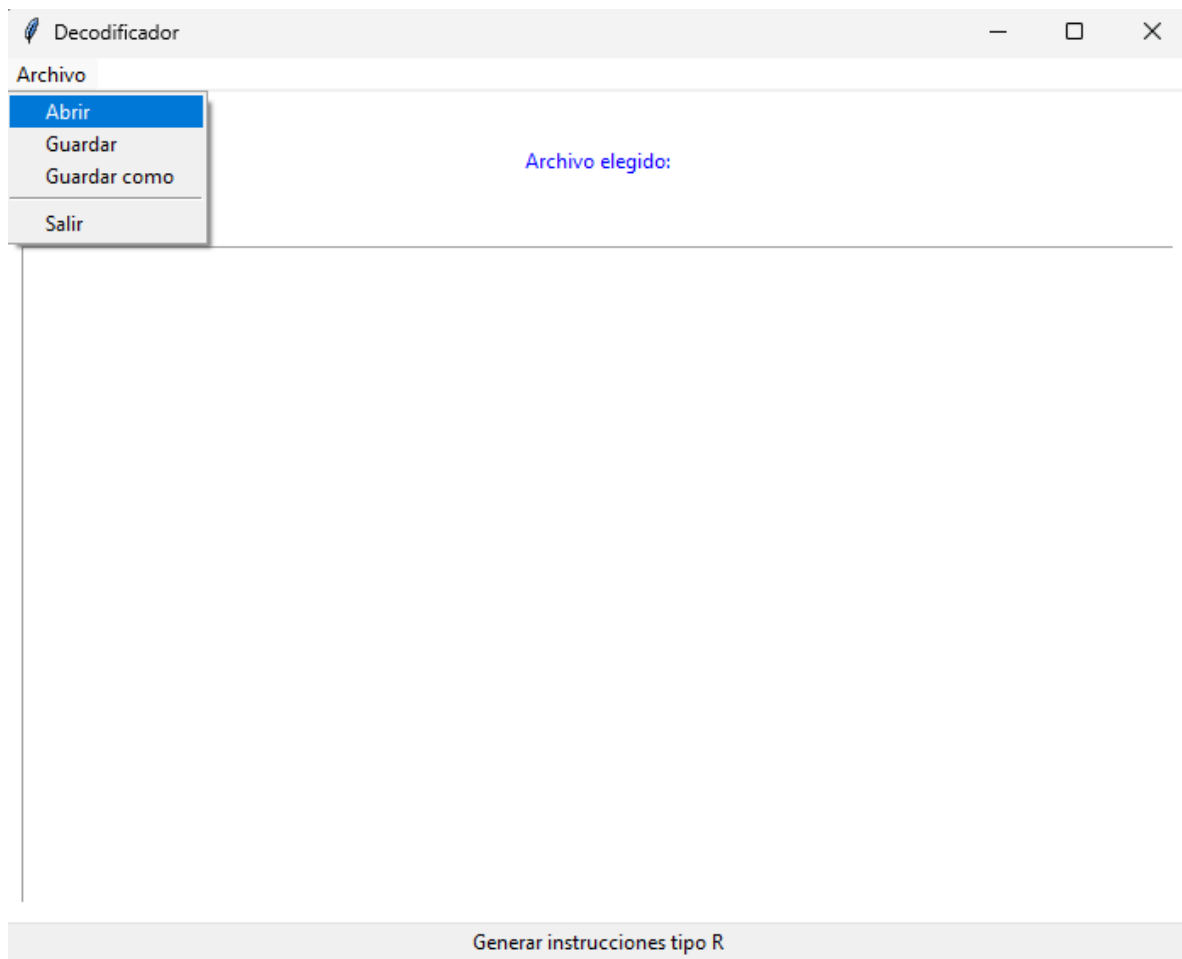
Nuestro decodificador es un programa hecho con Python utilizando la librería tkinter para crear un interfaz intuitivo para el usuario. En este programa podemos seleccionar cualquier archivo .asm o .txt de nuestro PC para posteriormente modificarlo y guardarlo. De igual manera podemos decodificar las instrucciones tipo R que el archivo seleccionado contenga.

Esto se hace mediante la lectura del archivo con la función `askopenfilename()` de tkinter, la cual abre el explorador de archivos para buscar el archivo deseado y guardar su ruta, posteriormente se lee el archivo por líneas con la función `open()` y se muestra en el editor de texto del programa.

Para leer/abrir un archivo debemos presionar Archivo > Abrir:

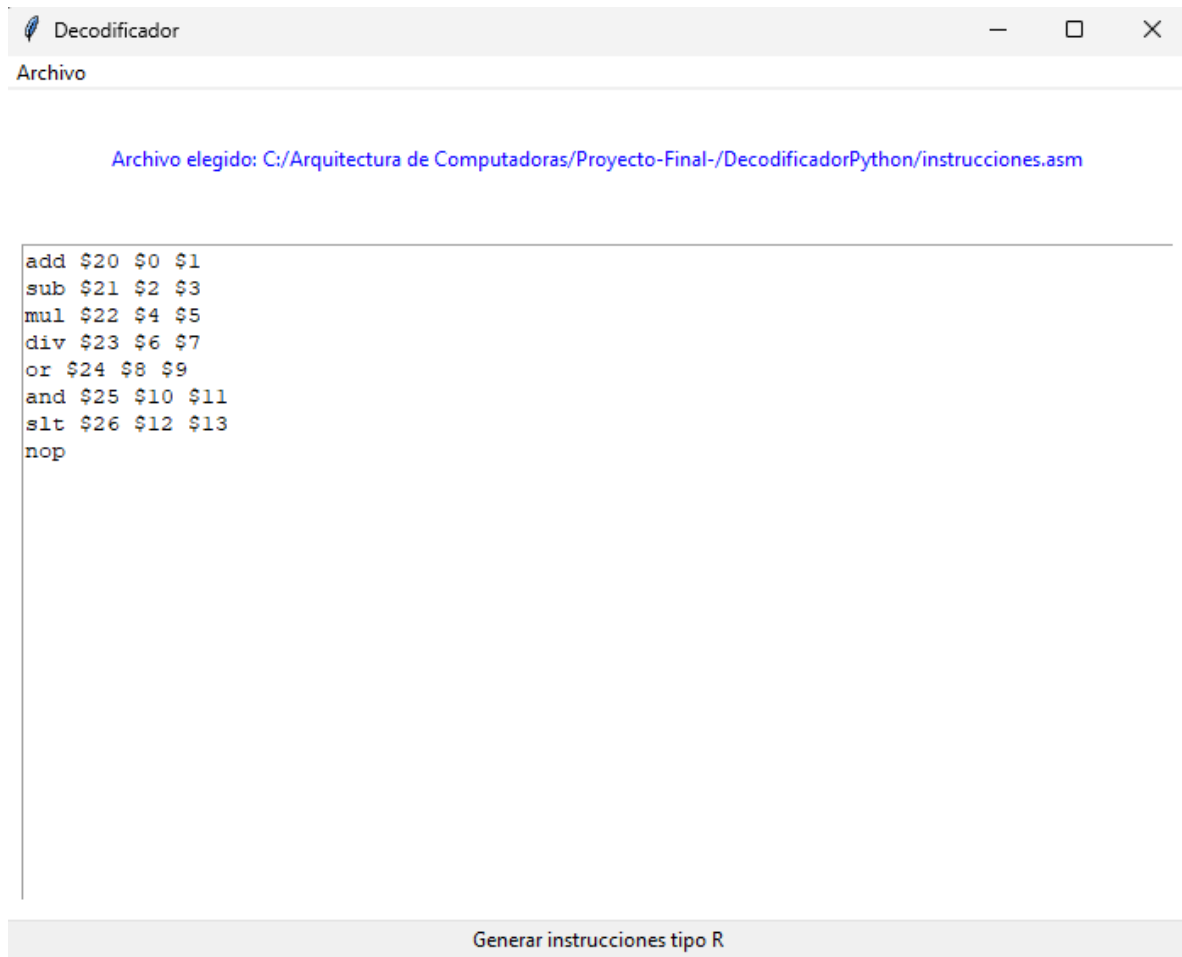
Universidad de Guadalajara

Piensa y trabaja



Universidad de Guadalajara

Piensa y trabaja



Si nuestro archivo contiene instrucciones de lenguaje ensamblador podemos modificarlo, guardarlo y decodificar las instrucciones tipo R que este contenga a instrucciones en formato MIPS. Esto se hace presionando el botón “*Generar instrucciones tipo R*”.

Universidad de Guadalajara

Piensa y trabaja

Decodificador

Archivo

Archivo elegido: C:/Arquitectura de Computadoras/Proyecto-Final-/DecodificadorPython/instrucciones.asm

```
add $20 $0 $1
sub $21 $2 $3
mul $22 $4 $5
div $23 $6 $7
or $24 $8 $9
and $25 $10 $11
slt $26 $12 $13
nop
```

Generar instrucciones tipo R

add \$20 \$0 \$1	0000000000000001101000000100000
sub \$21 \$2 \$3	00000000010000111010100000100010
mul \$22 \$4 \$5	0000000010000101101100000000010
div \$23 \$6 \$7	00000000110001111011100010011010
or \$24 \$8 \$9	00000001000010011100000000100101
and \$25 \$10 \$11	00000001010010111100100000100100
slt \$26 \$12 \$13	00000001100011011101000000101010
nop	00000000000000000000000000000000

Al presionar este botón en la parte inferior nos mostrará las instrucciones traducidas y se generará o se modificará automáticamente el archivo *“instrucciones_r.txt”* en la carpeta del DataPath de Verilog. En este archivo se guardan las instrucciones divididas en palabras de 8 bits, por lo que cada 4 líneas representan una sola instrucción.

Universidad de Guadalajara

Piensa y trabaja

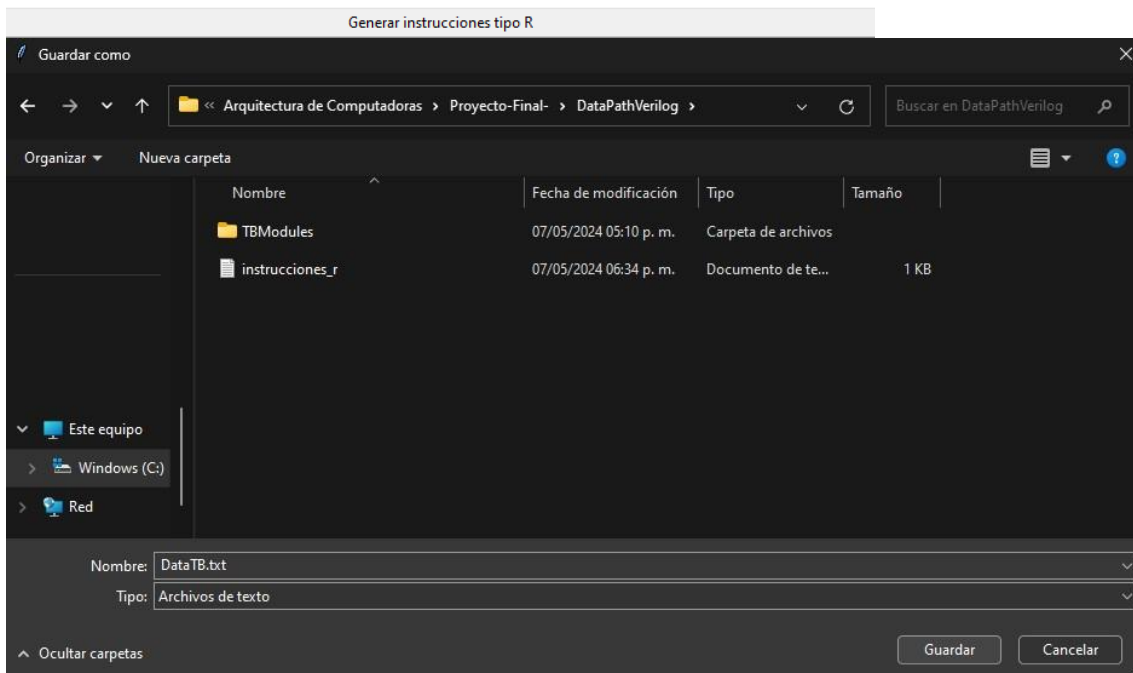
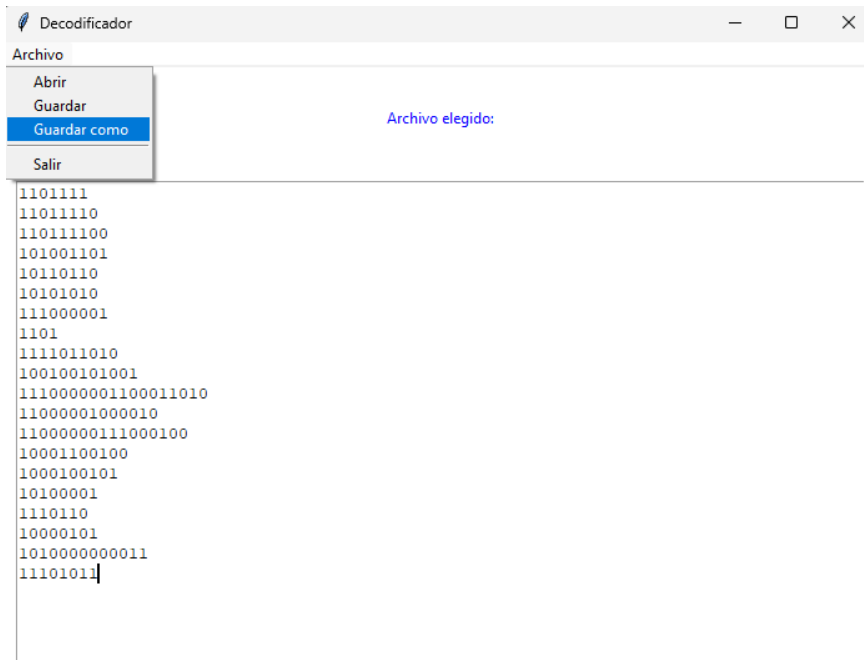
DataPathVerilog > instrucciones_r.txt

```
1 00000000
2 00000001
3 10100000
4 00100000
5 00000000
6 01000011
7 10101000
8 00100010
9 00000000
10 10000101
11 10110000
12 00000010
13 00000000
14 11000111
15 10111000
16 10011010
17 00000001
18 00001001
19 11000000
20 00100101
21 00000001
22 01001011
23 11001000
24 00100100
25 00000001
26 10001101
27 11010000
28 00101010
29 00000000
30 00000000
31 00000000
32 00000000
```

Universidad de Guadalajara

Piensa y trabaja

Otra opción que tenemos en este programa es únicamente usar el editor de texto sin necesidad de abrir otro archivo y guardar lo que hayamos escrito en un archivo nuevo utilizando la función “*Guardar como*”:



Universidad de Guadalajara

Piensa y trabaja

Verilog

Memoria de instrucciones

La memoria de instrucciones tiene una capacidad de 256 direcciones, cada una almacenando una instrucción de 32 bits, la inicialización carga las instrucciones desde un archivo de memoria (instrucciones_r.txt). La dirección de la memoria se multiplica por 4 (dirección * 4) para ajustarla al ancho de la palabra de la instrucción de 32 bits; en cada cambio de la dirección, se leen cuatro bytes de memoria y se almacenan en la instrucción de salida, de manera que cada instrucción ocupa cuatro ubicaciones consecutivas de memoria.

```
module InstructionMemory (  
    input wire [7:0] direccion,  
    output reg [31:0] instruccion  
);  
  
    // Contiene 256 direcciones  
    reg [7:0] memory [0:255];  
  
    // lectura de las instrucciones  
    initial begin  
        $readmemb("instrucciones_r.txt", memory);  
    end  
  
    assign direccion=direccion*4;  
  
    // lee la instruccion segun la direccion  
    always @(*) begin  
        instruccion[31:24] = memory[direccion];  
        instruccion[23:16] = memory[direccion+0'd1];  
        instruccion[15:8] = memory[direccion+0'd2];  
        instruccion[7:0] = memory[direccion+0'd3];  
    end  
endmodule
```


Universidad de Guadalajara

Piensa y trabaja

Multiplexor

Multiplexor de 32 bits con una entrada de selección ZF y dos entradas de datos de 32 bits (cero y uno), dependiendo del valor de la señal de selección (ZF), el módulo seleccionará una de las dos entradas de datos y la asignará a la salida salida. Si ZF es verdadero (1), la salida será igual a uno; si ZF es falso (0), la salida será igual a cero. El bloque always @(*) se encarga de actualizar continuamente la salida dependiendo del valor de ZF.

```
module multiplexor (  
    input wire ZF,  
    input wire [31:0] cero,  
    input wire [31:0] uno,  
    output reg [31:0] salida  
);  
  
    always @(*) begin  
        if (ZF) begin  
            salida = uno;  
        end else begin  
            salida = cero;  
        end  
    end  
  
endmodule
```

ADD

La unidad de suma de 32 bits o ADD toma dos operandos de entrada de 32 bits, operando1 y operando2, y produce un resultado de suma de 32 bits. El bloque always@(*) se encarga de sumar los dos operandos y asignar el resultado a la salida resultado. La suma se realiza de forma continua en cada cambio de los operandos de entrada.

Universidad de Guadalajara

Piensa y trabaja

```
module ADD(  
    input [31:0] operand1,    // Primer operando de 32 bits  
    input [31:0] operand2,    // Segundo operando de 32 bits  
    output reg [31:0] resultado // Resultado de la operación ADD  
);  
  
    always @(*) begin  
        resultado = operand1 + operand2; // Realiza la operación de suma ADD  
    end  
  
endmodule
```

PC

El contador de programa es una parte esencial de la arquitectura de un procesador, este mantiene la dirección de la próxima instrucción a ejecutar. Utiliza una señal de reloj clk para sincronizar su funcionamiento y una señal de reset para restablecer el contador a cero. El contador se incrementa en cada ciclo de reloj si no hay un reset activo. La dirección de la instrucción se asigna a la salida direccion, que es el valor actual del contador.

```
module PC (  
    input wire clk,          // para el reloj  
    input wire reset,        // un reset  
    output reg [7:0] direccion // Dirección de la memoria de instrucciones  
);  
  
    // contador para la dirección de la instrucción  
    reg [7:0] contador;  
  
    // actualizar la dirección  
    always @(clk, reset) begin  
        if (reset) begin  
            contador <= 8'd0; // restablecer el contador a cero en caso de reset  
        end else begin  
            if (clk) begin  
                contador <= contador + 1; // incrementar el contador en cada ciclo de reloj  
            end  
        end  
    end  
  
    // la dirección de la instrucción es la salida del contador  
    assign direccion = contador;  
  
endmodule
```

Universidad de Guadalajara

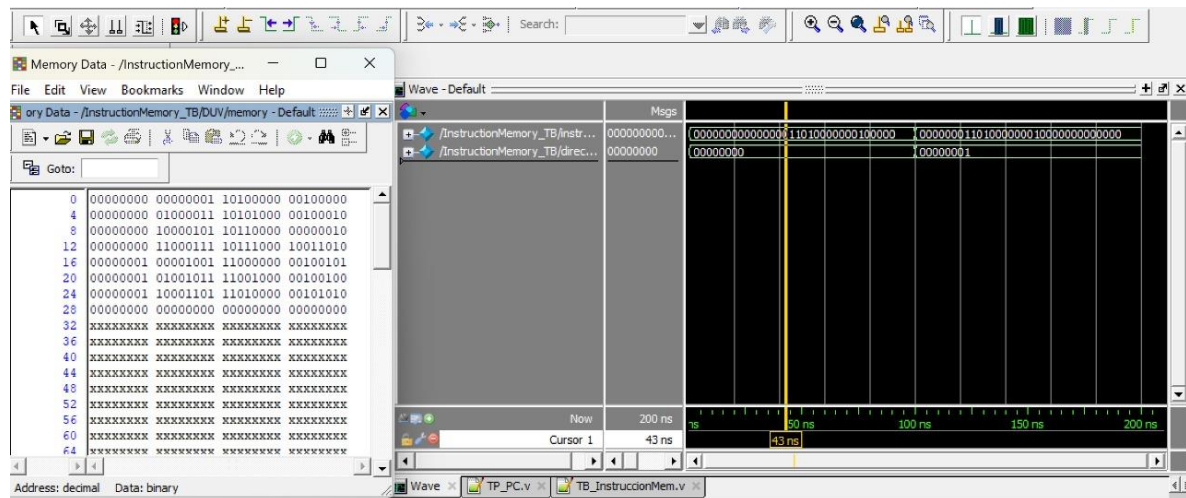
Piensa y trabaja

Test Bench:

Instructionmemory_TB

En el test bench la dirección de la memoria se establece en 8'h00. Después de un retraso de 100 unidades de tiempo, se cambia la dirección de la memoria a 8'h01. Se observará cómo la señal de salida instrucción refleja los valores almacenados en la memoria de instrucciones para las direcciones anteriormente proporcionadas.

Simulación:



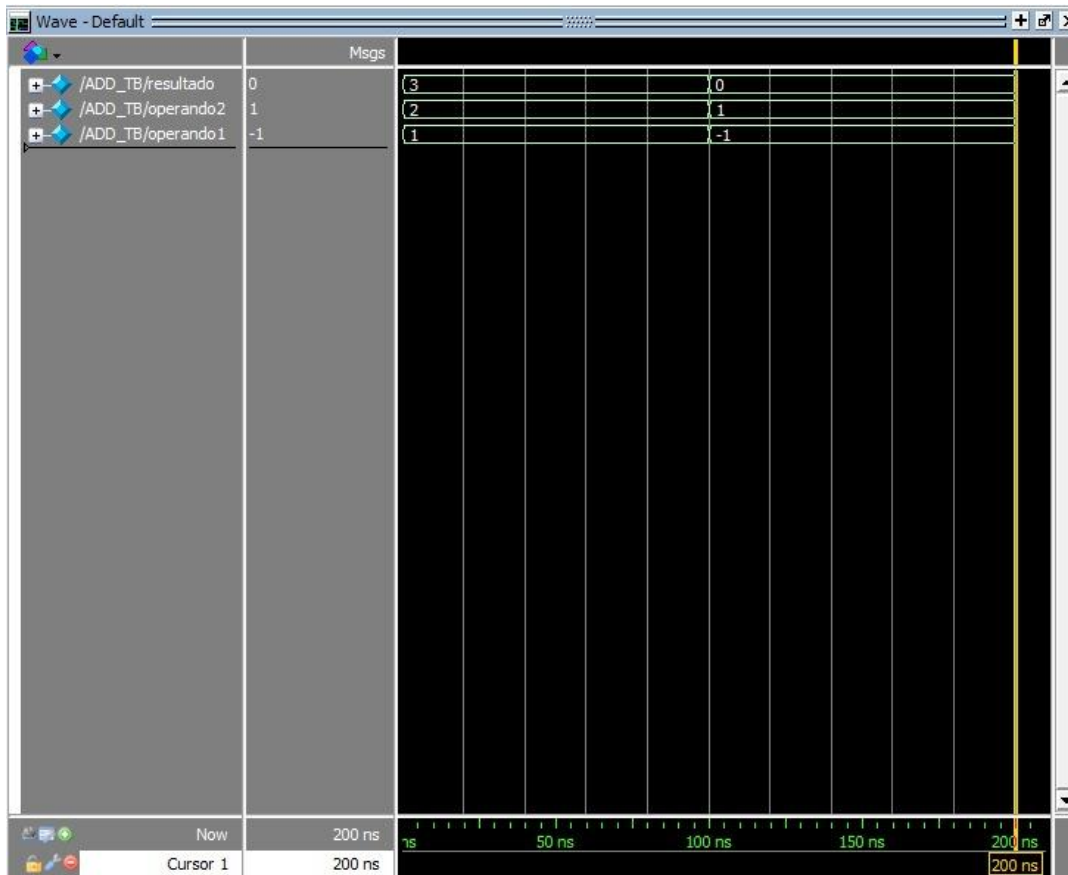
ADD_TB

Test bench para el módulo de suma llamado ADD, se definen dos registros operando1 y operando2 de 32 bits cada uno como entradas y un cable resultado de 32 bits como salida. Se instancia el módulo ADD, pasando las señales de entrada y salida correspondientes, en la sección inicial, se asignan valores iniciales a los operandos y se observa el comportamiento del módulo de suma. Primero, se asignan los valores 1 y 2 a los operandos y se espera un tiempo antes de cambiarlos a -1 y 1, respectivamente.

Universidad de Guadalajara

Piensa y trabaja

Simulación:



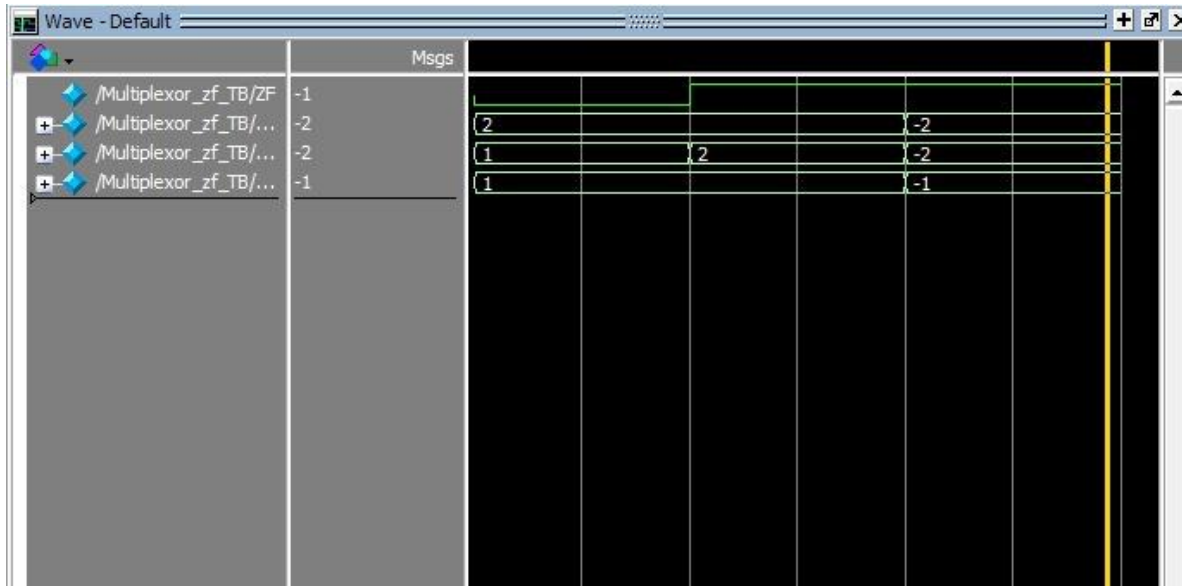
Multiplexor zf_TB

Selecciona entre dos entradas (cero y uno) basado en la señal ZF, la señal de bandera siempre comienza desactivada ($ZF = 0$), y se asignan valores iniciales a las entradas cero y uno, luego, se activa la señal de bandera ($ZF = 1$) y se observa cómo cambia la salida del multiplexor, después de un tiempo determinado, se cambian los valores de las entradas cero y uno.

Universidad de Guadalajara

Piensa y trabaja

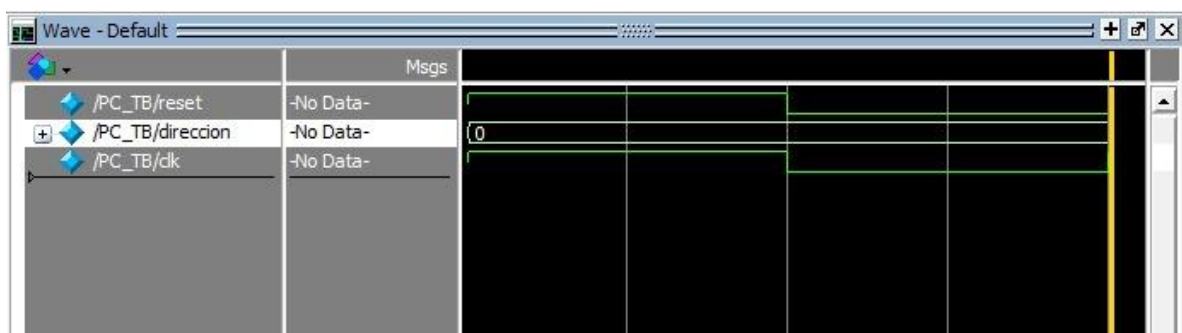
Simulación:



PC_TB

Test bench para un contador de programa (PC), utiliza una señal de reloj (clk) para generar pulsos de reloj cada 100 unidades de tiempo, inicializa y luego desactiva la señal de reset después de ciertos periodos de tiempo, lo que simula el proceso de inicialización del contador, la dirección de la memoria de instrucciones se asigna a la salida del contador.

Simulación:



Universidad de Guadalajara

Piensa y trabaja

Conclusiones:

Santiago Rafael Gómez Brizuela:

“En esta segunda etapa me pude dar cuenta que al momento de la creación de un proyecto siempre es tan importante como necesario la creación de una documentación, no solo para explicar y observar la evolución de nuestro proyecto sino también para mejorar la comprensión de esteya sea tanto para los integrantes de equipo como personas externas a el ”

Denice Estefania Rico Morones:

“En conclusion puedo decir que el proceso que hemos seguido implica la creación, implementación y prueba de varios módulos y sus correspondientes test benches, que son componentes esenciales en el diseño del procesador MIPS que estamos realizando. Comenzamos con la definición de módulos como la memoria de instrucciones, el multiplexor de ZF, la ALU suma, el contador del programa (PC) y los generados con anterioridad, cada uno desempeñando un papel crucial en la ejecución y control de operaciones dentro del procesador. Por ejemplo, la memoria de instrucciones almacena el código de operación que controla el comportamiento del procesador. El multiplexor interpreta estas instrucciones y coordina las operaciones del procesador en consecuencia. El PC rastrea la secuencia de instrucciones a ejecutar, y la ALU suma realiza la suma aritmética en los datos. Es importante entender estos conceptos y saber cómo implementarlos porque son las bases y cimientos de cualquier sistema informático. Un buen conocimiento de estos elementos es esencial para diseñar y desarrollar sistemas como en este caso, procesadores.”

Universidad de Guadalajara

Piensa y trabaja

Ferran Prado Roesner:

“En esta segunda fase de desarrollo del decodificador, noté las ventajas de utilizar Python para su creación, gracias a su simplicidad y capacidad para dar soluciones a problemas que no necesitan de aplicaciones extremadamente complejas. El uso de Python permitió enfocarse en la lógica del problema sin complicaciones adicionales. Además, utilizar la librería Tkinter simplificó la creación de la interfaz gráfica de usuario y con esto se logró implementar funciones como abrir, crear, editar y guardar archivos. Respecto a la decodificación, con Python fue bastante fácil hacer un algoritmo que traduzca las instrucciones a arquitectura MIPS, utilizando simples estructuras de control “if-else” y un diccionario con los códigos de cada instrucción.”

Alan Emmanuel Marin Lemus:

“En conclusión desarrollo de esta fase involucró una actualización en el decodificador de Python y la realización de nuevos módulos que se implementaron a los módulos anteriores para la creación del procesador MIPS. Aunque se tuvieron algunos imprevistos en la codificación en verilog todo el equipo tuvo un desempeño satisfactorio, teniendo como resultado un avance significativo en el desarrollo del proyecto final.”

Universidad de Guadalajara

Piensa y trabaja

Bibliografía:

Diferencias entre ruta de datos de ciclo único y ciclo múltiple

<https://www.geeksforgeeks.org/differences-between-single-cycle-and-multiple-cycle-datapath/>

Tenstorrent

<https://tenstorrent.com/>

Jim Keller

[https://en.wikipedia.org/wiki/Jim_Keller_\(engineer\)](https://en.wikipedia.org/wiki/Jim_Keller_(engineer))

Raja Koduri

https://en.wikipedia.org/wiki/Raja_Koduri

Big Endian contra Little Endian

<https://www.baeldung.com/cs/big-endian-vs-little-endian#:~:text=Big%2Dendian%20and%20little%2Dendian%20are%20the%20two%20main%20ways,at%20the%20smallest%20memory%20location.>

¿QUÉ ES ARQUITECTURA DE COMPUTADORAS?

<https://global.tiffin.edu/noticias/que-es-arquitectura-de-computadoras>

MIPS (procesador)

[https://es.wikipedia.org/wiki/MIPS_\(procesador\)#Instrucciones_reales](https://es.wikipedia.org/wiki/MIPS_(procesador)#Instrucciones_reales)

Universidad de Guadalajara

Piensa y trabaja

Última fase

Introducción: en esta fase tendremos la culminación de nuestro proyecto el cual será capaz de manejar instrucciones tipo R, tipo I y tipo J, con las cuales lograremos que nuestro datapath sea capaz de ejecutar un algoritmo factorial

Fase final lleva a cabo a partir del 13 de mayo

Objetivo:

El objetivo de esta fase es la culminación de nuestro proyecto el cual será capaz de ejecutar instrucciones tipo I y J , además claro de las R, las cuales ya era capaz de ejecutar, con las cuales seremos capaz de ejecutar un algoritmo factorial en nuestro datapath.

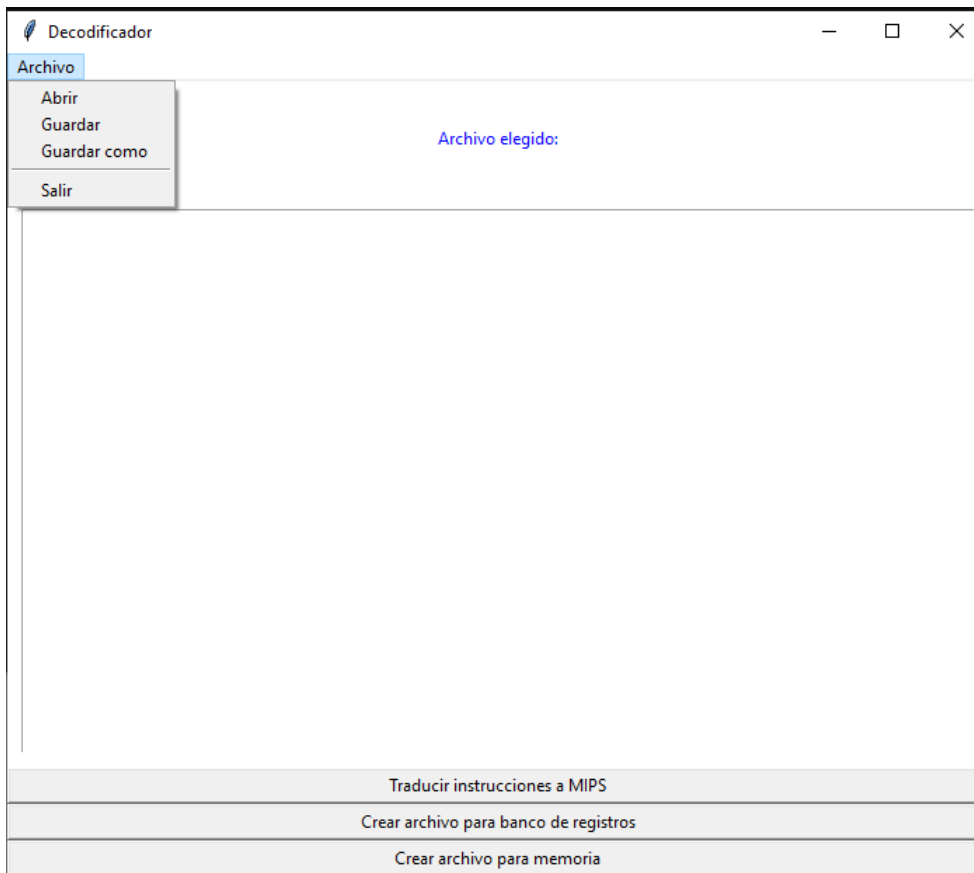
Además claro de manejar nuestro decodificador para que no solo decodifique instrucciones tipo R si no también instrucciones I y J, este nos ayudara para obtener su código binario el cual precargaremos en nuestro datapath para ejecutar nuestro algoritmo.

Decodificador:

Nuestro decodificador es un programa hecho con Python utilizando la librería tkinter para crear un interfaz intuitivo para el usuario. En este programa podemos seleccionar cualquier archivo .asm o .txt de nuestro PC para posteriormente modificarlo y guardarlo. De igual manera podemos decodificar las instrucciones tipo R, I, J que el archivo seleccionado contenga. Las otras dos opciones que tenemos son las de crear archivos .txt con datos en binario de 32 bits, para precargar el banco de registros o la memoria. Esto ultimo se hace a partir de un archivo .txt que contenga los números que se desean convertir a binario para precargarlos a las memorias.

Universidad de Guadalajara

Piensa y trabaja



En esta última fase el decodificador fue modificado para poder traducir las instrucciones tipo R, I y J que serán utilizadas en el programa. Se incluyeron las funciones:

- registerToBinary: Convierte un registro en su representación binaria de 5 bits.
- immediateToBinary: Convierte un valor inmediato en su representación binaria con un número específico de bits.
- instructionToBinary: Esta función toma una instrucción en ensamblador como entrada y la convierte en su formato binario correspondiente.
 - Las instrucciones se dividen en sus componentes (por ejemplo, opcode, registros, valores inmediatos).
 - Dependiendo del opcode, se genera el binario correspondiente para las instrucciones tipo R, I y J.

Ahora en la función “decodeInstruction” se lee cada línea del archivo y se ejecuta la función “instructionToBinary” con cada una de estas.

Piensa y trabaja

Decodificador

Archivo

Archivo elegido: C:/Arquitectura de Computadoras/Proyecto-Final-/DecodificadorPython/instrucciones.asm

```
nop
lw $7 5($15)
lw $8 6($15)
lw $9 7($15)
lw $10 8($15)
lw $11 9($15)
lw $12 10($15)
lw $13 11($15)
add $20 $0 $1
sub $21 $2 $3
or $22 $4 $5
and $23 $6 $7
slt $24 $8 $9
addi $25 $10 63
ori $26 $11 107
andi $27 $12 118
slti $28 $13 100
beq $0 $21 5
sw $20 15($15)
sw $21 16($15)
sw $22 17($15)
sw $23 18($15)
sw $24 19($15)
sw $25 20($15)
```

Traducir instrucciones a MIPS

Crear archivo para banco de registros

Crear archivo para memoria

nop	00000000000000000000000000000000
lw \$7 5(\$15)	10001101111001110000000000000101
lw \$8 6(\$15)	10001101111010000000000000001101
lw \$9 7(\$15)	10001101111010010000000000000111
lw \$10 8(\$15)	10001101111010100000000000001000
lw \$11 9(\$15)	10001101111010110000000000001001
lw \$12 10(\$15)	10001101111011000000000000001010
lw \$13 11(\$15)	10001101111011010000000000001011
add \$20 \$0 \$1	0000000000000001101000000100000
sub \$21 \$2 \$3	00000000010000111010100000100010
or \$22 \$4 \$5	00000000100001011011000000100101
and \$23 \$6 \$7	00000000110001111011100000100100
slt \$24 \$8 \$9	00000001000010011100000000101010
add \$25 \$10 \$3	00100001010110010000000000111111
ori \$26 \$11 107	00110101011110100000000011010101

Piensa y trabaja

Decodificador

Archivo

Archivo elegido: C:/Arquitectura de Computadoras/Proyecto-Final-/DecodificadorPython/registerBankData.txt

```
111
222
444
333
43
87
102
0
0
0
0
0
0
0
0
20
```

Traducir instrucciones a MIPS

Crear archivo para banco de registros

Crear archivo para memoria

[illegible]

Universidad de Guadalajara

Piensa y trabaja

NOTA: Para el correcto funcionamiento del decodificador es importante que los directorios estén organizados de la misma manera que están en github y con los mismos nombres, esto debido a que los archivos de precarga de memoria se deben guardar directamente en el directorio “DataPathVerilog”

Verilog

Mux2 1 5

Es un multiplexor de 2 a 1 que selecciona entre 2 opciones de 5 bits, esto por medio de un bloque always que por medio de un if-else que elige el dato que saldrá

- Si sel=1 sale el dato de Op2
- Si sel=0 sale el dato de op1

Y produce una salida dependiendo de la señal del selector dicha señal ha sido arrojada por el contolUnit o unidad de control, posteriormente el op seleccionado será enviado al banco de registros

Para instrucciones tipo R, RegDst será 1, indicando que el registro destino es el indicado en la ubicación de la instrucción.R.

Para instrucciones tipo I, RegDst será 0, indicando que el registro destino es el indicado en la instruccione I.

ShiftLeft2 26 28

Este módulo “alarga” una entrada de 26 bits desplazándola 2 posiciones hacia la derecha otorgándonos así una salida de 28 bits, esto por medio del uso de la concatenación agregándole 2 ceros, la entrada de 26 bits es la dirección de la instrucción J, la salida, ya de 28 bits forma la dirección final de la instrucción J.

ShiftLeft2 32

Este módulo desplaza 2 posiciones a la izquierda los 32 bits entrantes por medio del operador de desplazamiento “<<”, los nuevos 32 bits serán necesarios para la bifurcación final en instrucciones del tipo I

SignExtend

Universidad de Guadalajara

Piensa y trabaja

Este módulo tiene una entrada de 16 bits la cual se extiende hasta 32 bits por medio de una concatenación condicional fijándose en el bit más significativo de la entrada conservando el signo de valor, la señal entrante proviene de la instrucción tipo I

- Si la entrada de 16 bits tiene signo negativo se concatena una extensión de 16 bits más de unos lo que preserva el signo negativo
- Si la entrada de 16 bits tiene signo positivo se extiende con ceros hasta lograr los 32 bits, preservando así el signo positivo

Test Bench:

TB_ShiftLeft

El bloque ejecuta una vez y espera 100 unidades de tiempo para que el DUT procese el dato en la entrada, nos ayuda a verificar que el dato se haya desplazado a la izquierda correctamente

Simulación:

/TB_ShiftLeft/in	1	1	
/TB_ShiftLeft/out	4	4	

TB_SignExtend

El bloque asigna el valor decimal 12 a la entrada, se toma 100 unidades de tiempo para permitir que el DUT procese el dato, luego asigna el valor 20 a la entrada y espera 100 unidades de tiempo para que el DUT las procese y así sucesivamente aumentado 5 unidades decimales y esperando 100 unidades de tiempo, hasta que pasen 400 unidades de tiempo y se detenga con 30 unidades

Simulación:

	Msgs				
/TB_SignExtend/in	30	15	20	25	30
/TB_SignExtend/out	30	15	20	25	30

Universidad de Guadalajara

Piensa y trabaja

Funcionalidad de todas las instrucciones

```
1  nop
2  lw $7 5($15)
3  lw $8 6($15)
4  lw $9 7($15)
5  lw $10 8($15)
6  lw $11 9($15)
7  lw $12 10($15)
8  lw $13 11($15)
9  add $20 $0 $1
10 sub $21 $2 $3
11 or $22 $4 $5
12 and $23 $6 $7
13 slt $24 $8 $9
14 addi $25 $10 63
15 ori $26 $11 107
16 andi $27 $12 118
17 slti $28 $13 100
18 beq $1 $2 5
19 sw $20 15($15)
20 sw $21 16($15)
21 sw $22 17($15)
22 sw $23 18($15)
23 sw $24 19($15)
24 sw $25 20($15)
25 sw $26 21($15)
26 sw $27 22($15)
27 sw $28 23($15)
28 j 0
```

Demostraremos la correcta ejecución de todas las instrucciones de la imagen de al lado mostrando nuestro banco de registro y nuestra memoria

Universidad de Guadalajara

Piensa y trabaja

Banco de registros y memoria inicializada:

Memory Data - /TB_DataPath/DUT/registerBank/MEM				
0	111	222	444	
3	333	43	87	
6	102	0	0	
9	0	0	0	
12	0	0	0	
15	20	x	x	
18	x	x	x	
21	x	x	x	
24	x	x	x	
27	x	x	x	
30	x	x		
33				

Memory Data - /TB_DataPath/DUT/DataMem/MEM - Default									
0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0
16	0	0	0	0	5	1	1	0	0
24	0	82	999	555	37	118	77	22	
32	x	x	x	x	x	x	x	x	x
40	x	x	x	x	x	x	x	x	x
48	x	x	x	x	x	x	x	x	x
56	x	x	x	x	x	x	x	x	x
64	x	x	x	x	x	x	x	x	x
72	x	x	x	x	x	x	x	x	x
80	x	x	x	x	x	x	x	x	x

Banco de registros después de las instrucciones lw:

Instruccion ASM	Tipo	rt	offset	base	Dato
nop	R	-	-	-	-
lw \$7 5(\$15)	I	7	5	15	82
lw \$8 6(\$15)	I	8	6	15	999
lw \$9 7(\$15)	I	9	7	15	555
lw \$10 8(\$15)	I	10	8	15	37
lw \$11 9(\$15)	I	11	9	15	118
lw \$12 10(\$15)	I	12	10	15	77
lw \$13 11(\$15)	I	13	11	15	22

Memory Data - /TB_DataPath/DUT/registerBank/MEM				
0	111	222	444	
3	333	43	87	
6	102	82	999	
9	555	37	118	
12	77	22	0	
15	20	x	x	
18	x	x	x	
21	x	x	x	
24	x	x	x	
27	x	x	x	
30	x	x		

Universidad de Guadalajara

Piensa y trabaja

Banco de registros después de las instrucciones R:

Instrucción ASM	Tipo	rs	Dato 1	rt	Dato 2	rd	Resultado
add \$20 \$0 \$1	R	0	111	1	222	20	333
sub \$21 \$2 \$3	R	2	444	3	333	21	111
or \$22 \$4 \$5	R	4	43	5	87	22	127
and \$23 \$6 \$7	R	6	102	7	82	23	66
slt \$24 \$8 \$9	R	8	999	9	555	24	0

0	111	222	444
3	333	43	87
6	102	82	999
9	555	37	118
12	77	22	0
15	20	x	x
18	x	x	333
21	111	127	66
24	0	x	x
27	x	x	x
30	x	x	

Banco de registros después de las instrucciones I

Instrucción ASM	Tipo	rs	Dato	rt	Valor inmediato	Resultado
addi \$25 \$10 63	I	10	37	25	63	100
ori \$26 \$11 107	I	11	118	26	107	127
andi \$27 \$12 118	I	12	77	27	118	68
slti \$28 \$13 100	I	13	22	28	100	1

0	111	222	444
3	333	43	87
6	102	82	999
9	555	37	118
12	77	22	0
15	20	x	x
18	x	x	333
21	111	127	66
24	0	100	127
27	68	1	x
30	x	x	

Universidad de Guadalajara

Piensa y trabaja

Memoria después de las instrucciones sw(se salta 5 instrucciones porque se cumple la instrucción beq)

Instrucción ASM	Tipo	rs	Dato 1	rt	Dato 2
beq \$0 \$21 5	I	0	111	21	111

Las instrucciones en verde son las que si se ejecutan:

Instrucción ASM	Tipo	rt	Dato	offset	base
sw \$20 15(\$15)	I	20	333	15	15
sw \$21 16(\$15)	I	21	111	16	15
sw \$22 17(\$15)	I	22	127	17	15
sw \$23 18(\$15)	I	23	66	18	15
sw \$24 19(\$15)	I	24	0	19	15
sw \$25 20(\$15)	I	25	100	20	15
sw \$26 21(\$15)	I	26	127	21	15
sw \$27 22(\$15)	I	27	68	22	15
sw \$28 23(\$15)	I	28	1	23	15

Address	Value
0	0
8	0
16	0
24	82
32	x
40	100
48	x
56	x
64	x
72	x

La siguiente señal demuestra el correcto funcionamiento de la instrucción j saltando de dirección 108 en la memoria de instrucciones a la dirección 0 tras ejecutar "j 0".

Intruccion ASM	Tipo	target
j 0	J	0

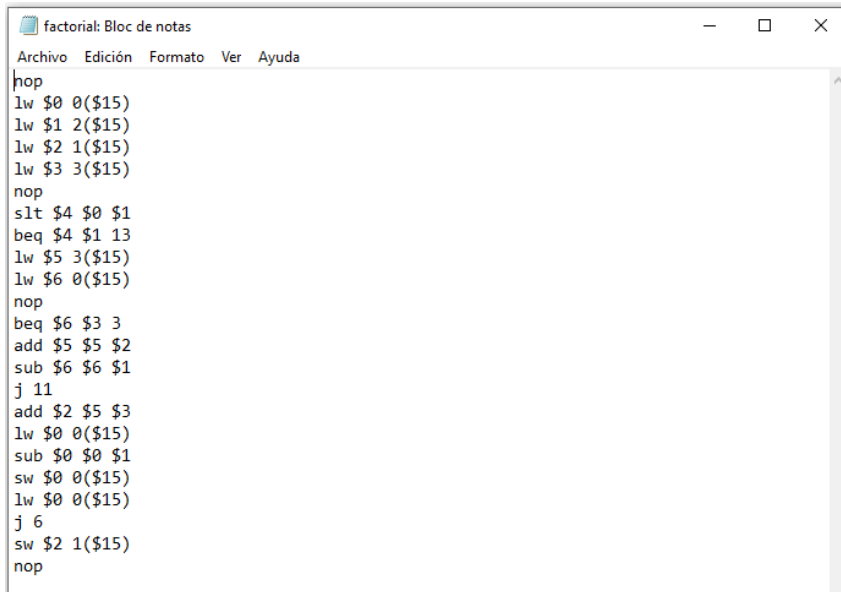
Signal	Value
/TB_DataPath/DUT/counter/dirOut	100
/TB_DataPath/DUT/instructionMem/address	104
/TB_DataPath/DUT/instructionMem/address	108
/TB_DataPath/DUT/instructionMem/address	0
/TB_DataPath/DUT/instructionMem/address	4

Universidad de Guadalajara

Piensa y trabaja

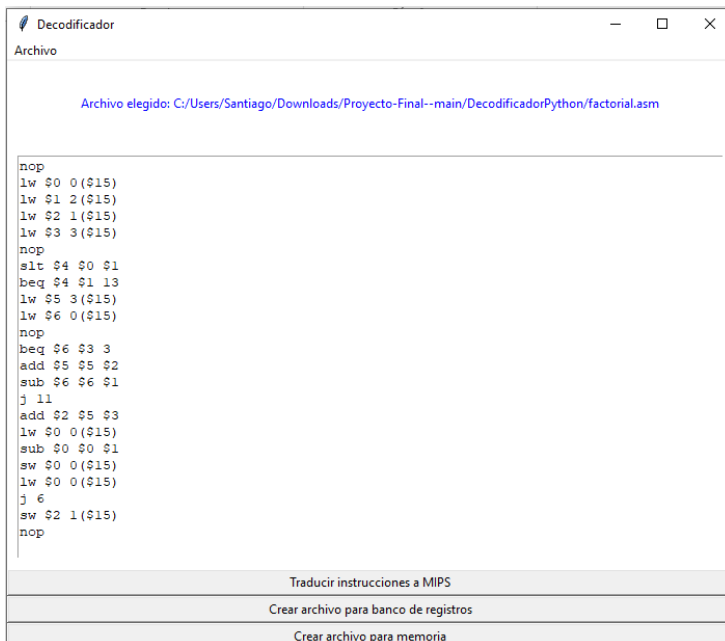
Ejecución del algoritmo vectorial

Con nuestro set de instrucciones en ensamblador listo



```
factorial: Bloc de notas
Archivo  Edición  Formato  Ver  Ayuda
nop
lw $0 0($15)
lw $1 2($15)
lw $2 1($15)
lw $3 3($15)
nop
slt $4 $0 $1
beq $4 $1 13
lw $5 3($15)
lw $6 0($15)
nop
beq $6 $3 3
add $5 $5 $2
sub $6 $6 $1
j 11
add $2 $5 $3
lw $0 0($15)
sub $0 $0 $1
sw $0 0($15)
lw $0 0($15)
j 6
sw $2 1($15)
nop
```

Procedemos a decodificarlo por medio de nuestro decodificador hecho en Python con una interfaz intuitiva creada con Tkinter, además de ser capaz de crear archivos tanto como para el banco de registros como la memoria.



```
Decodificador
Archivo

Archivo elegido: C:/Users/Santiago/Downloads/Proyecto-Final--main/DecodificadorPython/factorial.asm

nop
lw $0 0($15)
lw $1 2($15)
lw $2 1($15)
lw $3 3($15)
nop
slt $4 $0 $1
beq $4 $1 13
lw $5 3($15)
lw $6 0($15)
nop
beq $6 $3 3
add $5 $5 $2
sub $6 $6 $1
j 11
add $2 $5 $3
lw $0 0($15)
sub $0 $0 $1
sw $0 0($15)
lw $0 0($15)
j 6
sw $2 1($15)
nop

Traducir instrucciones a MIPS
Crear archivo para banco de registros
Crear archivo para memoria
```

Universidad de Guadalajara

Piensa y trabaja

Subido nuestro archivo al decodificador procedemos a decodificarlo en binario para la correcta ejecución de sus instrucciones en el datapath

nop	00000000000000000000000000000000
lw \$0 0(\$15)	10001101111000000000000000000000
lw \$1 2(\$15)	10001101111000010000000000000010
lw \$2 1(\$15)	10001101111000100000000000000001
lw \$3 3(\$15)	10001101111000110000000000000011
nop	00000000000000000000000000000000
slt \$4 \$0 \$1	0000000000000001001000000101010
beq \$4 \$1 13	00010000100000010000000000001101
lw \$5 3(\$15)	10001101111001010000000000000011
lw \$6 0(\$15)	10001101111001100000000000000000
nop	00000000000000000000000000000000
beq \$6 \$3 3	00010000110000110000000000000011
add \$5 \$5 \$2	00000000101000100010100000100000
sub \$6 \$6 \$1	00000000110000010011000000100010
i 11	00001000000000000000000000000011

Con las instrucciones precargadas, inicializamos el banco de registros y la memoria

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	5	1	1	0	x	x	x	x	x
29	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
42	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
56	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
70	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
84	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
98	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
112	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
126	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
140	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
154	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
168	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
182	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
196	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
210	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
224	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
238	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
252	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

0	0	0	0
3	0	0	0
6	0	0	0
9	0	0	0
12	0	0	0
15	20	x	x
18	x	x	x
21	x	x	x
24	x	x	x
27	x	x	x
30	x	x	x

Por medio de las instrucciones lw anteriormete precargadas cargamos datos de la memoria en el banco de registros esto tomado como base el dato en la ubicación 15 del banco de registros y sumándole diferentes offsets para obtener diferentes ubicaciones y mandando cada una a una dirección del banco empezando por la dirección 0

Universidad de Guadalajara

Piensa y trabaja

Memory Data - /TB_DataPath/DUT/registerBank/MEM			
0	5	1	1
3	0	0	0
6	0	0	0
9	0	0	0
12	0	0	0
15	20	x	x
18	x	x	x
21	x	x	x
24	x	x	x
27	x	x	x
30	x	x	

Se multiplica primero 5×4 (esto por medio de sumas y comparaciones, en este caso el 5 se suma 4 veces) y se guarda el resultado (20) en la dirección 2 del banco de registros

Memory Data - /TB_DataPath/DUT/registerBank/MEM - Default			
0	4	1	20
3	0	0	20
6	0	0	0
9	0	0	0
12	0	0	0
15	20	x	x
18	x	x	x
21	x	x	x
24	x	x	x
27	x	x	x
30	x	x	

Posteriormente el 20×3 (se suma el 20 3 veces hasta que la comparación sea válida) obteniendo 60 y guardándolo en la dirección 2 del banco de registros

Memory Data - /TB_DataPath/DUT/registerBank/MEM - Default			
0	3	1	60
3	0	0	60
6	0	0	0
9	0	0	0
12	0	0	0
15	20	x	x
18	x	x	x
21	x	x	x
24	x	x	x
27	x	x	x
30	x	x	

Piensa y trabaja

El 60x2 (suma el 60 2 veces hasta que la comparación lo avale) obtenido así 120 y guardándolo en el banco de registros

Memory Data - /TB_DataPath/DUT/registerBank/MEM - Default			
0	2	1	120
3	0	0	120
6	0	0	0
9	0	0	0
12	0	0	0
15	20	x	x
18	x	x	x
21	x	x	x
24	x	x	x
27	x	x	x
30	x	x	

Por ultimo suma el 120 una vez y guarda lo obtenido en el banco de registros

Memory Data - /TB_DataPath/DUT/registerBank/MEM - Default			
0	1	1	120
3	0	0	120
6	0	0	0
9	0	0	0
12	0	0	0
15	20	x	x
18	x	x	x
21	x	x	x
24	x	x	x
27	x	x	x
30	x	x	

Y para finalizar guardamos el factorial de 5 en nuestra memoria, esto por medio de la penúltima instrucción que manda el dato que está ubicado en la posición del registro 2(120) y la manda al valor de la ubicación 15 del banco de registros mas una posición (20)+1 siendo más específicos en la dirección 21.

[illegible]

Universidad de Guadalajara

Piensa y trabaja

Conclusiones:

Santiago Rafael Gómez Brizuela:

“En este proyecto, se diseñó y creó un DataPath en Verilog para ejecutar instrucciones MIPS de tipo R, I y J, abarcando operaciones como sumas, restas, lógicas, y manejo de memoria, además de control de flujo. Además, se desarrolló un decodificador en Python que traduce instrucciones en lenguaje ensamblador a MIPS binario, permitiendo la validación del procesador mediante varias pruebas. Asimismo, se implementó un algoritmo para calcular el factorial de cualquier número utilizando estas instrucciones, demostrando la capacidad del sistema. En resumen, el proyecto logró demostrar que el procesador MIPS puede ejecutar una amplia gama de instrucciones y realizar cálculos aritméticos complejos.”

Denice Estefania Rico Morones:

“En esta fase se finalizó el programa ensamblador para un procesador MIPS de 32 bits, abarcando las instrucciones tipo R, I y J. Las operaciones incluyeron sumas, restas, operaciones lógicas, así como cargas y almacenamientos de memoria, y control de flujo; aunque se identificaron problemas específicos con las instrucciones de carga y almacenamiento, el resto de las operaciones se ejecutaron a la perfección.

El proyecto incluyó la validación del código ensamblador a través de diversas pruebas. Este proceso evidenció tanto la capacidad del procesador para ejecutar tareas complejas como las áreas que necesitan mejoras.

En resumen, el proyecto cumplió con los objetivos de demostrar que el procesador MIPS puede manejar una amplia gama de instrucciones.”

Universidad de Guadalajara

Piensa y trabaja

Ferran Prado Roesner:

“La creación del DataPath, ha sido un proyecto educativo e interesante. Este trabajo ha requerido un buen entendimiento de la arquitectura MIPS y habilidades en programación Python. La documentación detallada de cada paso, desde el diseño del DataPath hasta la implementación del decodificador también fue muy importante para garantizar la claridad y la posibilidad de entender el proyecto. Además, se implementó un algoritmo para calcular la factorial de cualquier número utilizando estas instrucciones, demostrando la capacidad del sistema para ejecutar tareas complejas.”

Alan Emmanuel Marin Lemus:

“En este proyecto, se desarrolló un procesador MIPS de 32 bits capaz de ejecutar instrucciones tipo R, I y J, y procesar un programa en ensamblador para calcular el factorial de un número. Se implementó el "single datapath" en Verilog y se validó su funcionamiento con un archivo de prueba. Aunque hubo problemas con la instrucción sw, los módulos se desarrollaron con éxito. Se creó un decodificador en Python que traduce el ensamblador MIPS a código binario y genera archivos de memoria de instrucciones. En resumen, el proyecto demostró la capacidad del procesador para manejar instrucciones esenciales y realizar operaciones aritméticas, cumpliendo los objetivos iniciales y proporcionando una base sólida para futuras mejoras.”

Universidad de Guadalajara

Piensa y trabaja

Bibliografía

Patterson, D. A., & Hennessy, J. L. (2011). *Computer Organization and Design : The Hardware/Software Interface Revised 4th edition Ed. 4.*

https://bnt.execvox.com/book/88809585/?_locale=en