

Universidad de Guadalajara

Piensa y trabaja



Procesador MIPS

Fase 1

Equipo 6: “Fontaneros”

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

Turno: Matutino

Fecha: 29/04/24

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

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

Objetivo:

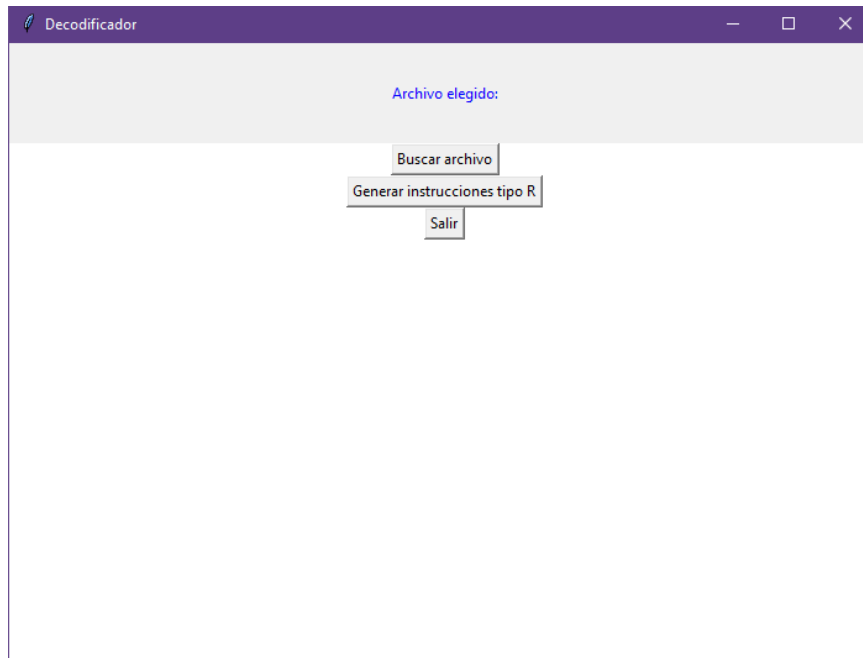
Crear un DataPath capaz de ejecutar instrucciones MIPS, en este caso las instrucciones ADD, SUB, AND, OR y SLT. Para esto, será necesario integrar algunos de los módulos ya usados en actividades previas con nuevos módulos que gestionaran el flujo de los bits en el sistema. Además se creará un decodificador, 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
```

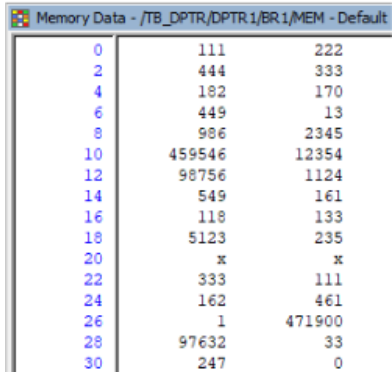
Las cuales, ordenadas quedan de la siguiente manera:

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.

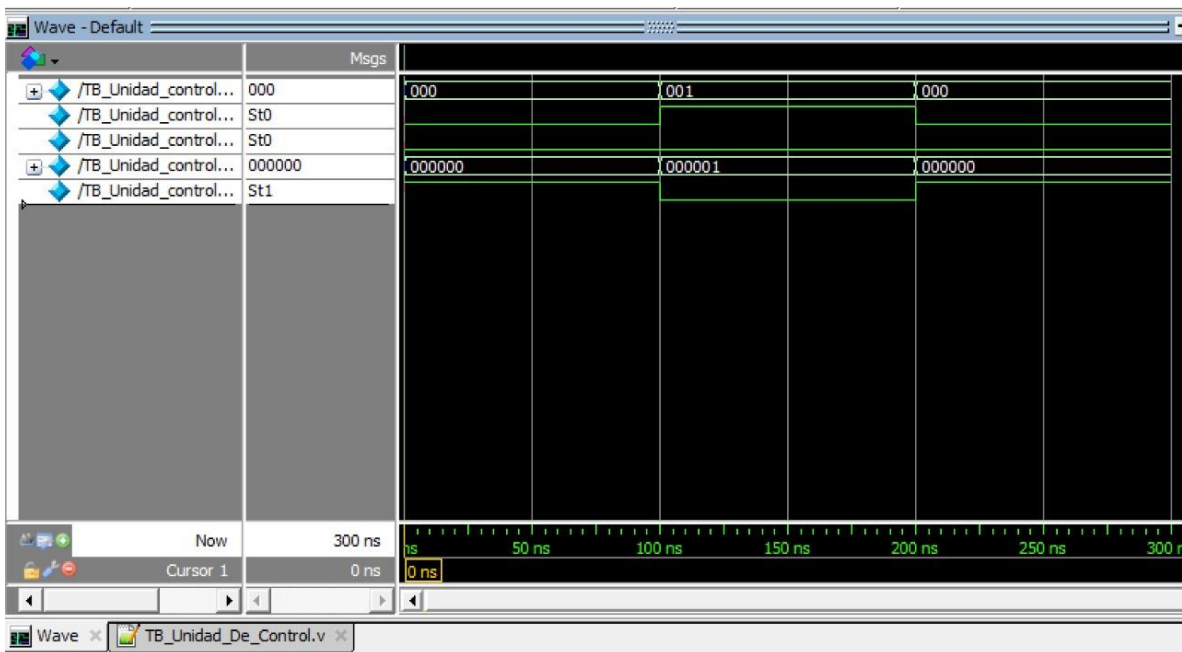
Dentro del bloque "initial", se inicializa la señal TB_op con diferentes valores para simular distintas operaciones de control:

- 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)

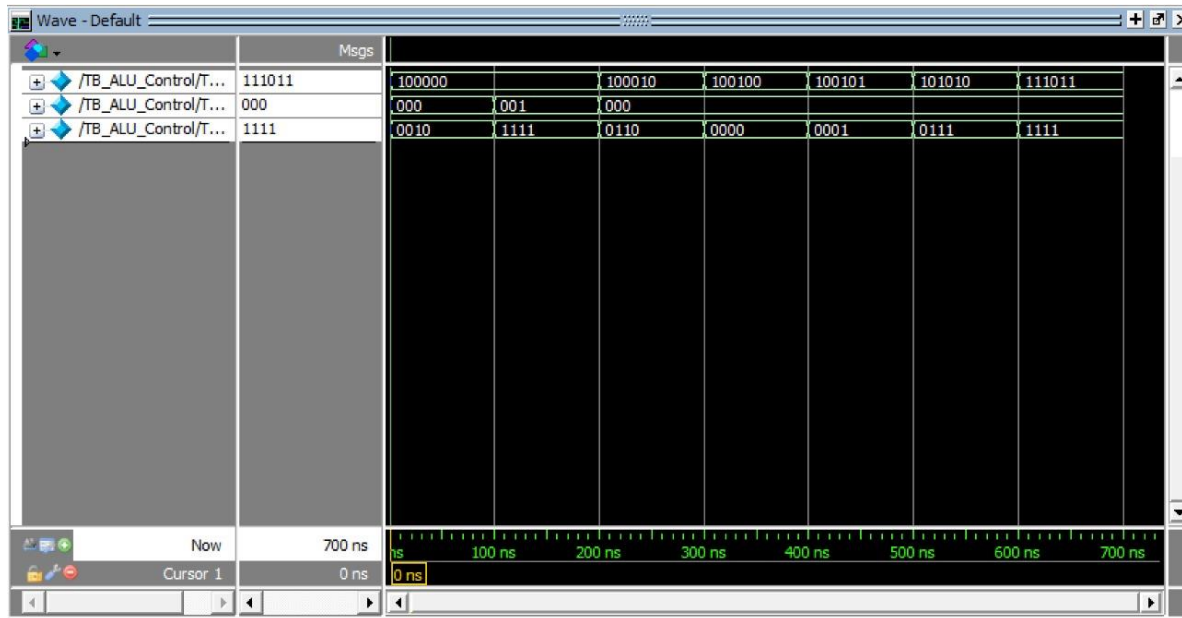
Universidad de Guadalajara

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.

Universidad de Guadalajara

Piensa y trabaja

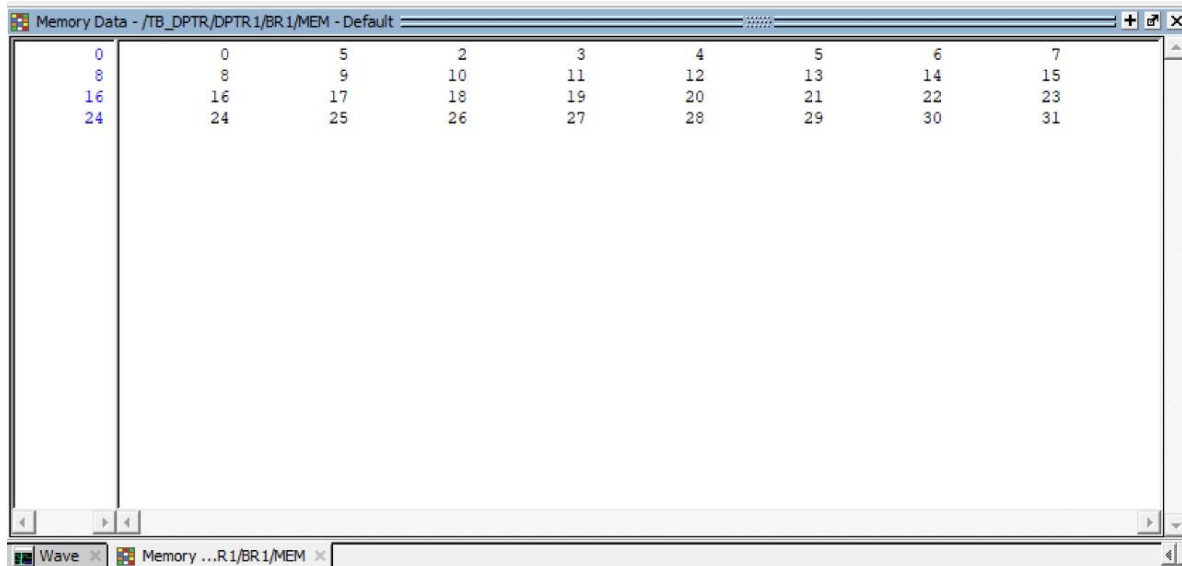
2. Se instancia el camino de datos DPTR1, conectando la señal de entrada InstruccionTR y la señal de salida TR_ZF.

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:



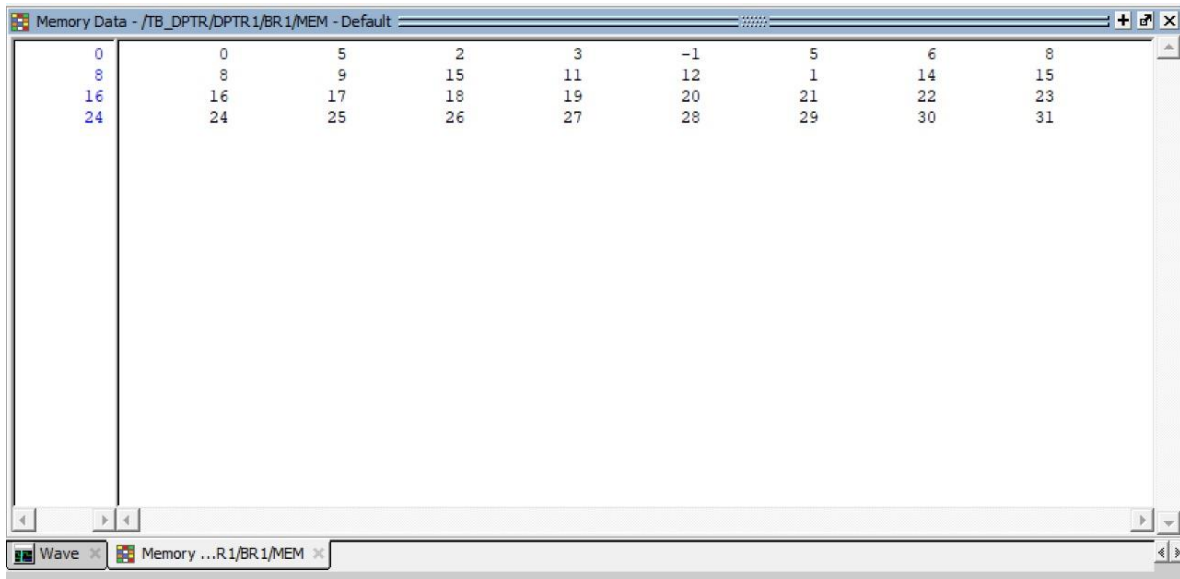
The screenshot shows a window titled "Memory Data - /TB_DPTR/DPTR1/BR1/MEM - Default". It displays a table with 8 columns and 4 rows of data. The first column contains addresses (0, 8, 16, 24) and the subsequent columns contain hexadecimal values. The table is as follows:

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

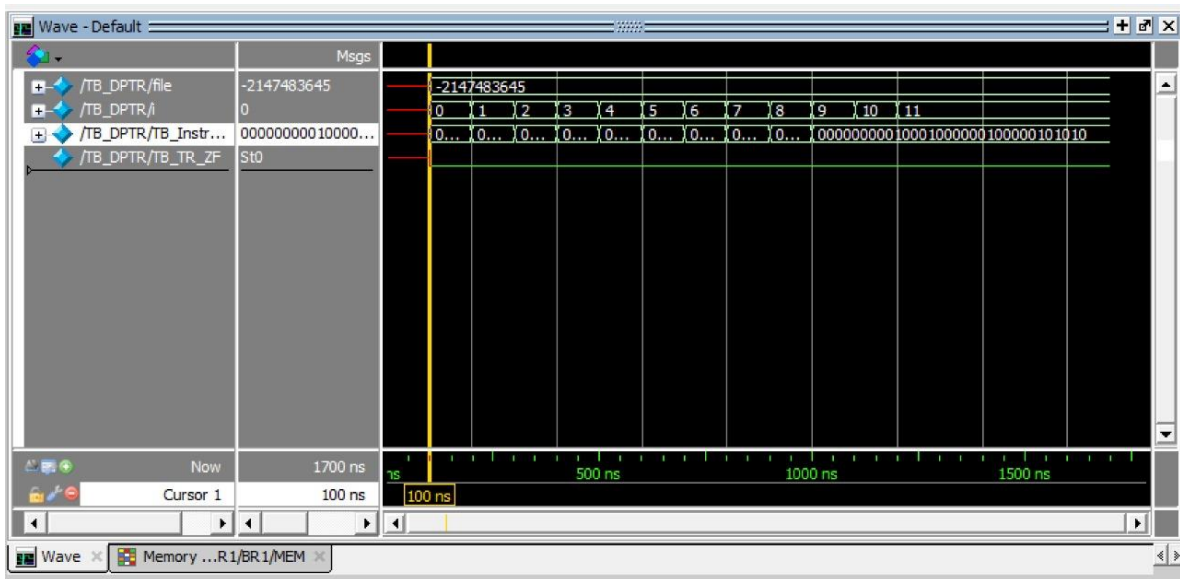
Simulación DPTR-2:

Universidad de Guadalajara

Piensa y trabaja



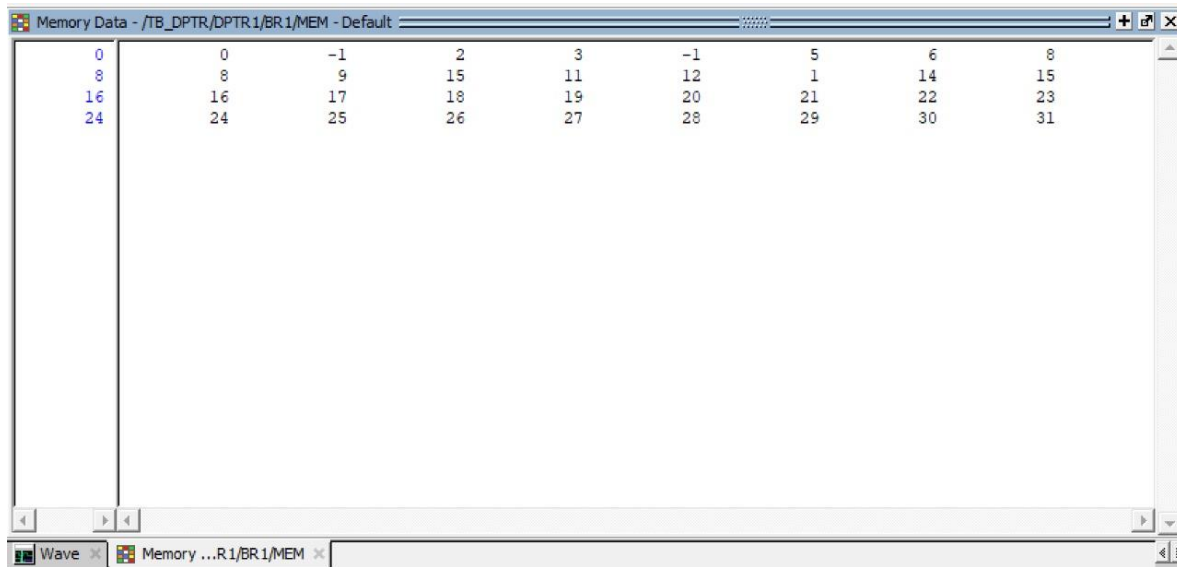
Simulación DPTR-3:



Simulación DPTR-4:

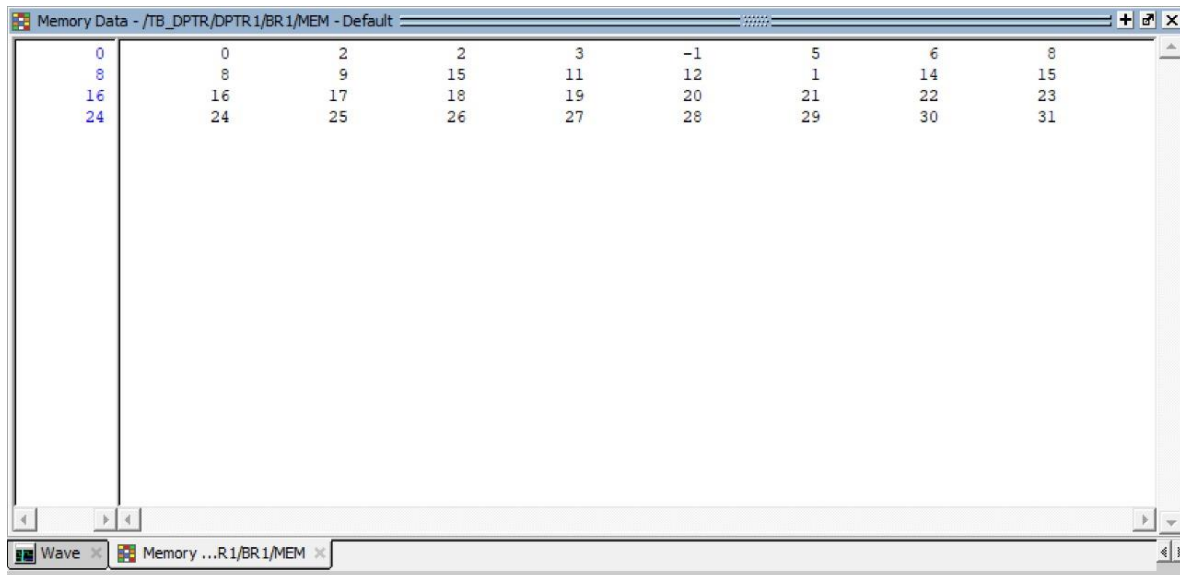
Universidad de Guadalajara

Piensa y trabaja



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

Simulación DPTR-5:



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

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 capas 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/practica13/

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