Arquitectura de Computadoras

MIPS32 CON MENÚ DE 3 ALGORITMOS

Equipo Tr3s: Maciel Vargas Oswaldo Daniel y García Saldivar Hugo Gabriel

Arquitectura MIPS32

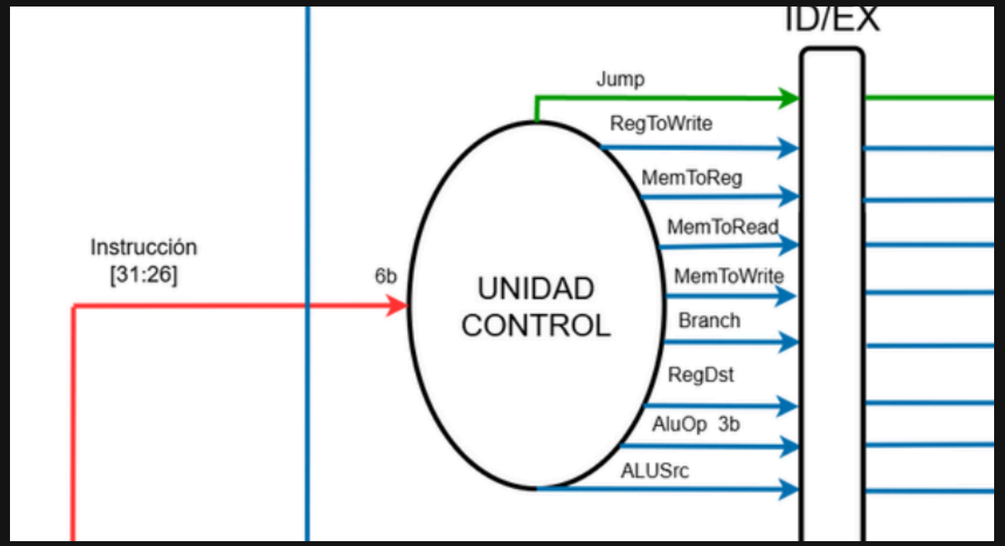
La arquitectura MIPS32 de tipo RISC (Reduced Instruction Set Computer) es decir de un enfoque que se aprovecha de instrucciones simples requiriendo de un menor gasto energético y un diseño simple en comparación con otros enfoques como el CISC (Complex Instruction Set Computer). La ventaja con el conjunto de instrucciones de tipo RISC en comparación al CISC es que estas estan altamente optimizadas y funcionan de manera muy eficiente en pipelines.

Pipeline

El pipeline son una serie de pasos secuenciales que se utilizan para llegar a un objetivo en particular similar a la definición de un algoritmo. La diferencia es que esta serie de pasos se dividen en distintas etapas en las que hasta terminar una de ellas con los datos obtenidos se operará la siguiente y asi secuencialmente hasta la última etapa.

Tipos de instrucciones

MIPS32 utiliza 3 tipos de instrucciones principales que son procesadas en la etapa “instruction decode” dependiendo del tipo de instrucción la unidad de control decidirá que módulos se activan o desactivan con el fin de que los datos pasen exitosamente y cumplan la función solicitada, a su vez encargándose de que no se guarde o lea ningún dato no requerido que pueda dificultar o dañar la lógica principal.



Tipo R

Nomenclatura

OP: Código de operación (generalmente en 0 para instrucciones tipo R).

Rs: Registro fuente.

Rt: Registro destino temporal.

RD: Registro destino final.

Shamt: Cantidad de desplazamiento (shift).

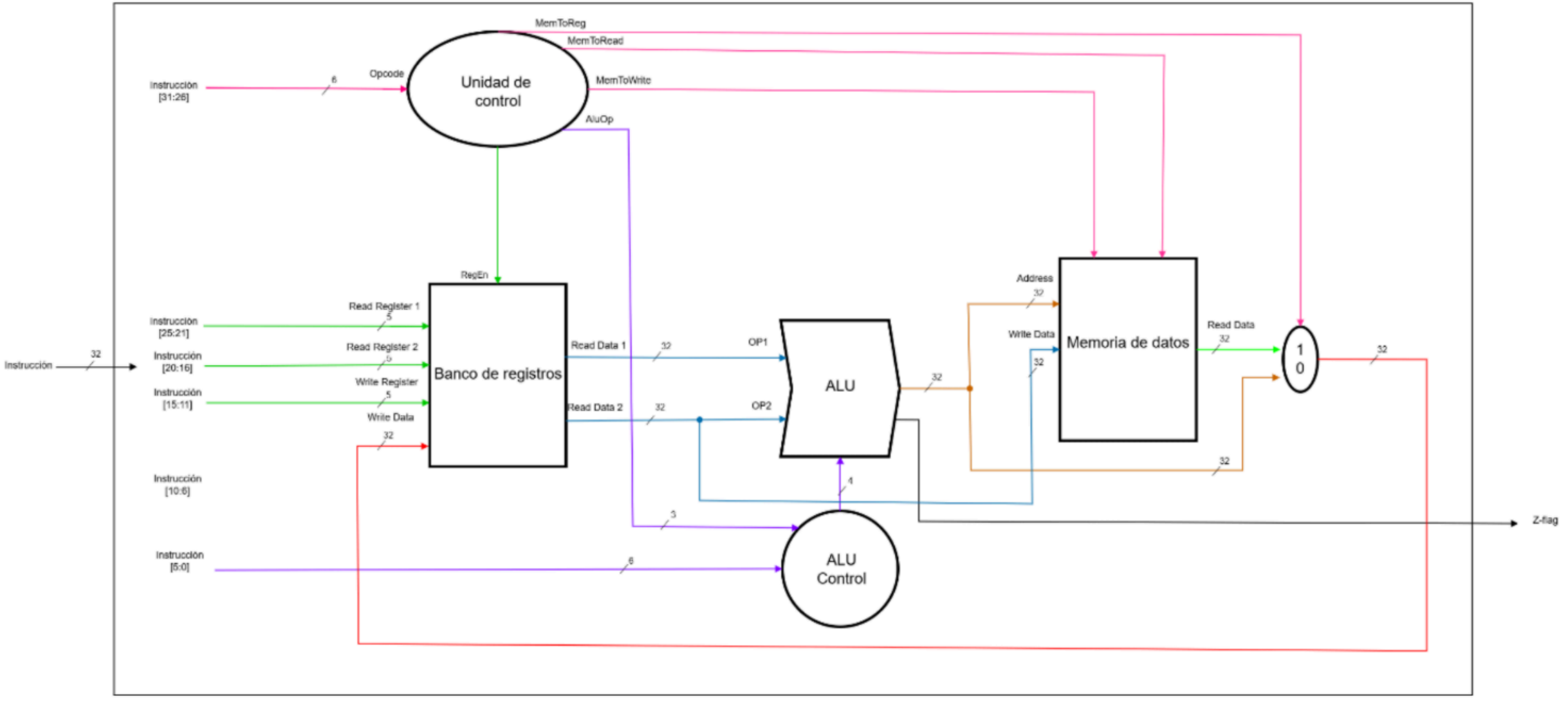
Funct: Subcódigo que especifica la operación exacta.

Instrucción

OP	Rs	Rt	RD	Shamt	Funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
32 bits					

Instrucciones implementadas

Instrucciones tipo R				
No.	**Mnemónico**	**Funct**	**Ensamblador**	**Descripción**
1	ADD	100000	ADD $RD, $Rs, $Rt	Realiza una operación suma de los registros $Rs y $Rt y el resultado lo guarda en $RD
2	SUB	100010	SUB $RD, $Rs, $Rt	Realiza una operación resta de los registros $Rs y $Rt y el resultado lo guarda en $RD
3	SLT	101010	SLT $RD, $Rs, $Rt	Si el registro $Rs es menor a $Rt, escribe un 1 en $RD, sino escribe un 0 en $RD
4	AND	100100	AND $RD, $Rs, $Rt	Realiza una operación AND de los registros $Rs y $Rt y el resultado lo guarda en $RD
5	OR	100101	OR $RD, $Rs, $Rt	Realiza una operación OR de los registros $Rs y $Rt y el resultado lo guarda en $RD
6	DIVU	011011	DIVU $RD, $Rs, $Rt	Realiza una operación módulo de los registros $Rs y $Rt y el resultado lo guarda en $RD
7	RSQRT	010110	RSQRT $RD, $Rs, $0	Realiza una operación raíz cuadrada de los registros $Rs y el resultado lo guarda en $RD
8	MUL	000010	MUL $RD, $Rs, $Rt	Realiza una operación multiplicación de los registros $Rs y $Rt y el resultado lo guarda en $RD


MemToReg
MemToRead
MemToWrite
AluOp
Opcode
6
Instrucción [31:26]
Unidad de control
RegEn
Read Register 1
5
Instrucción [25:21]
Read Register 2
5
Instrucción [20:16]
Write Register
5
Instrucción [15:11]
Write Data
32
Instrucción [10:6]
Instrucción [5:0]
6
Banco de registros
Read Data 1
32
Read Data 2
32
OP1
OP2
ALU
32
4
3
ALU Control
Address
32
Write Data
32
Memoria de datos
Read Data
32
1
0
32
32
Instrucción
32
Z-flag

Tipo I

Nomenclatura

OP: Código de operación.

Rs: Registro fuente.

Rt: Registro destino o fuente.

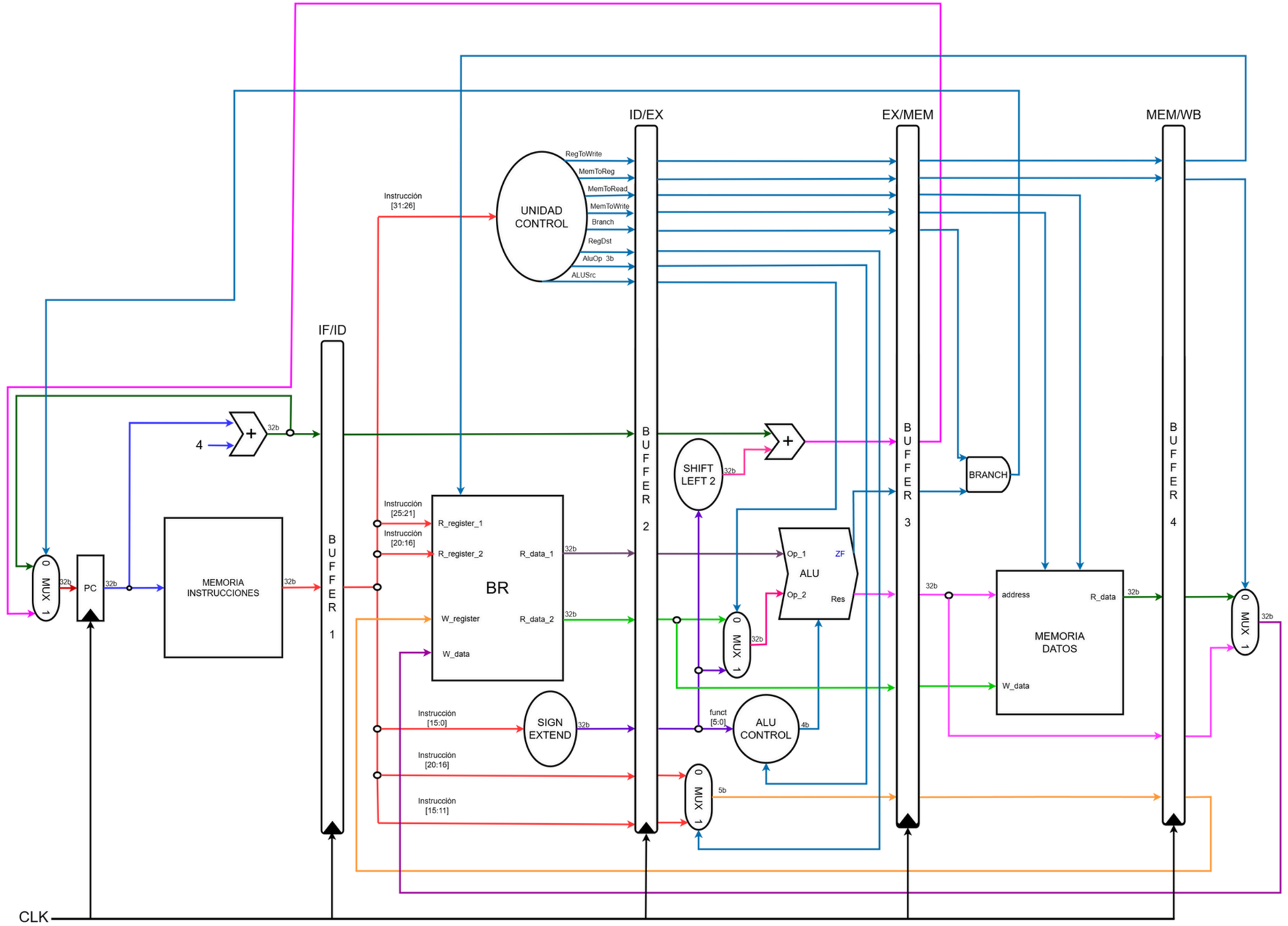
Immediate: Valor constante de 16 bits (se extiende a 32 bits por tipo I)

Instrucción

OP	RS	RT	Immediate
6 bits	5 bits	5 bits	16 bits
32 bits			

Instrucciones implementadas

Instrucciones tipo R				
No.	**Mnemónico**	**Opcode**	**Ensamblador**	**Descripción**
1	ADDI	001000	ADDI $Rt, $Rs, #Inmediate	Realiza una operación suma del registro $Rt con el valor inmediato, el resultado lo guarda en $Rs
2	ORI	001101	ORI $Rt, $Rs, #Inmedaite	Realiza una operación OR del registro $Rt con el valor inmediato, el resultado lo guarda en $Rs
3	ANDI	001100	ANDI $Rt, $Rs, #Inmediate	Realiza una operación AND del registro $Rt con el valor inmediato, el resultado lo guarda en $Rs
4	SW	101011	SW $Rt, $Rs, #Inmediate	Calcula la posición relativa [$Rs + #In] y guarda el dato $Rt (memoria de datos)
5	LW	100011	LW $Rt, $Rs, #Inmediate	Calcula la posición relativa [$Rs + #In] y extrae el dato $Rt (banco de registros)
6	BEQ	000100	BEQ $Rs, $Rt, #Inmediate	Compara los registros $Rs y $Rt, si son iguales se activa el salto (#in), si no el programa continúa con la siguiente instrucción.


ID/EX
EX/MEM
MEM/WB
IF/ID
UNIDAD CONTROL
RegToWrite
MemToReg
MemToRead
MemToWrite
Branch
RegDst
AluOp 3b
ALUSrc
Instrucción [31:26]
4
32b
MUX
0
1
PC
MEMORIA INSTRUCCIONES
BUFFER 1
BUFFER 2
BUFFER 3
BUFFER 4
Instrucción [25:21]
Instrucción [20:16]
R_register_1
R_register_2
W_register
W_data
R_data_1
R_data_2
BR
SHIFT LEFT 2
BRANCH
Op_1
Op_2
ALU
ZF
Res
address
R_data
MEMORIA DATOS
W_data
Instrucción [15:0]
SIGN EXTEND
funct [5:0]
ALU CONTROL
4b
Instrucción [20:16]
Instrucción [15:11]
5b
CLK

Tipo J

Nomenclatura

OP: Código de operación.

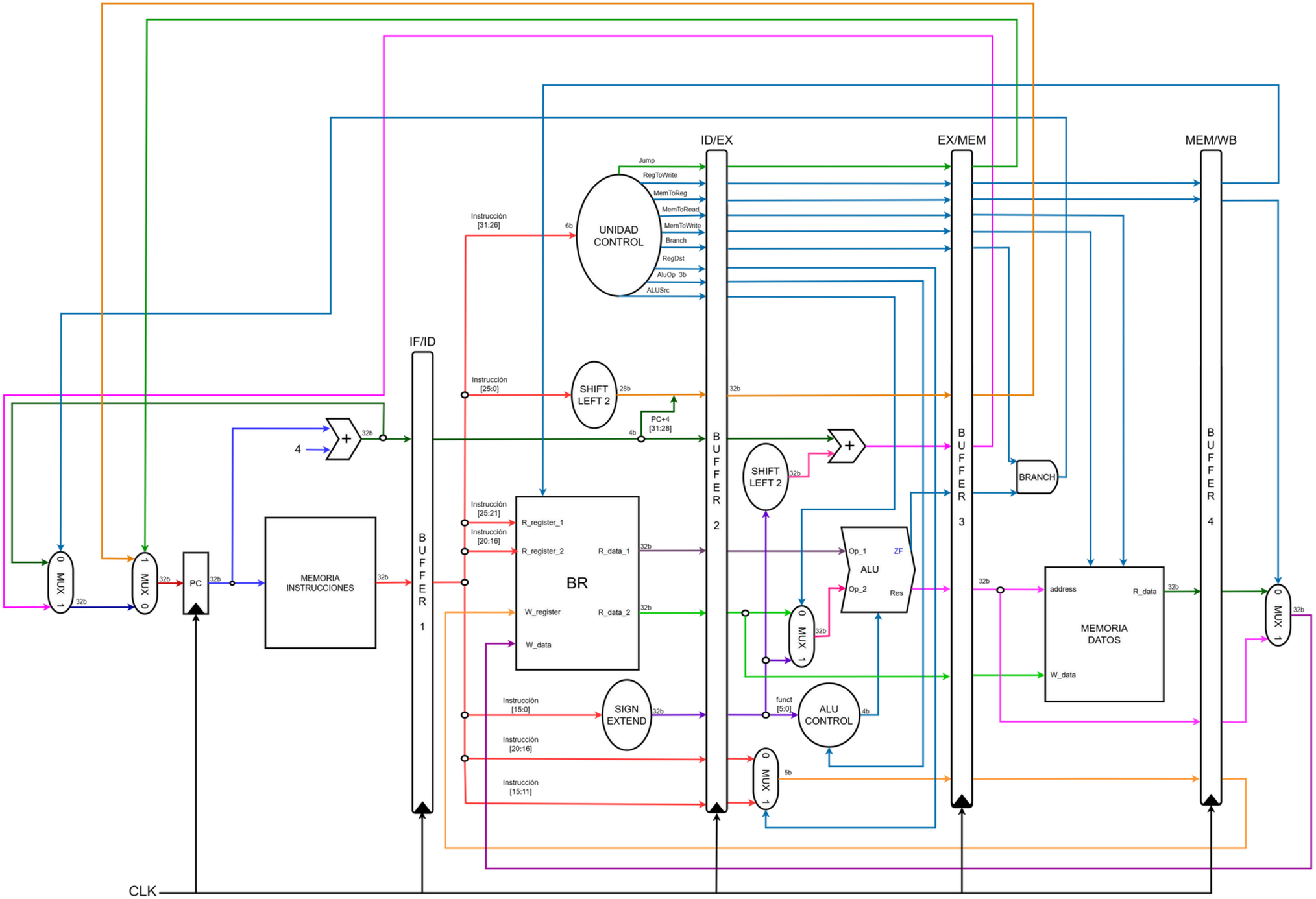
Target: Dirección de salto (se escala a 28 bits y se combina con PC).

Instrucción

OP	Target
6 bits	26 bits
32 bits	

Instrucciones implementadas

Instrucciones tipo J				
No.	**Mnemónico**	**Opcode**	**Ensamblador**	**Descripción**
1	J	000010	J #target	Realiza un salto hacia la instrucción definida en #target


ID/EX
EX/MEM
MEM/WB
IF/ID
Jump
RegToWrite
MemToReg
MemToRead
MemToWrite
Branch
RegDst
AluOp 3b
ALUSrc
UNIDAD CONTROL
6b
Instrucción [31:26]
Instrucción [25:0]
SHIFT LEFT 2
28b
PC+4 [31:28]
4b
32b
4
Instrucción [25:21]
Instrucción [20:16]
R_register_1
R_register_2
W_register
W_data
R_data_1
R_data_2
BR
SHIFT LEFT 2
Op_1
Op_2
ALU
ZF
Res
BRANCH
address
R_data
MEMORIA DATOS
W_data
MEMORIA INSTRUCCIONES
PC
MUX 0
MUX 1
Instrucción [15:0]
SIGN EXTEND
funct [5:0]
ALU CONTROL
4b
Instrucción [20:16]
Instrucción [15:11]
5b
BUFFER 1
BUFFER 2
BUFFER 3
BUFFER 4
CLK

Mención Honorífica

Gracias por todo NOP :’)

X	NOP	000000	NOP	No hacer nada

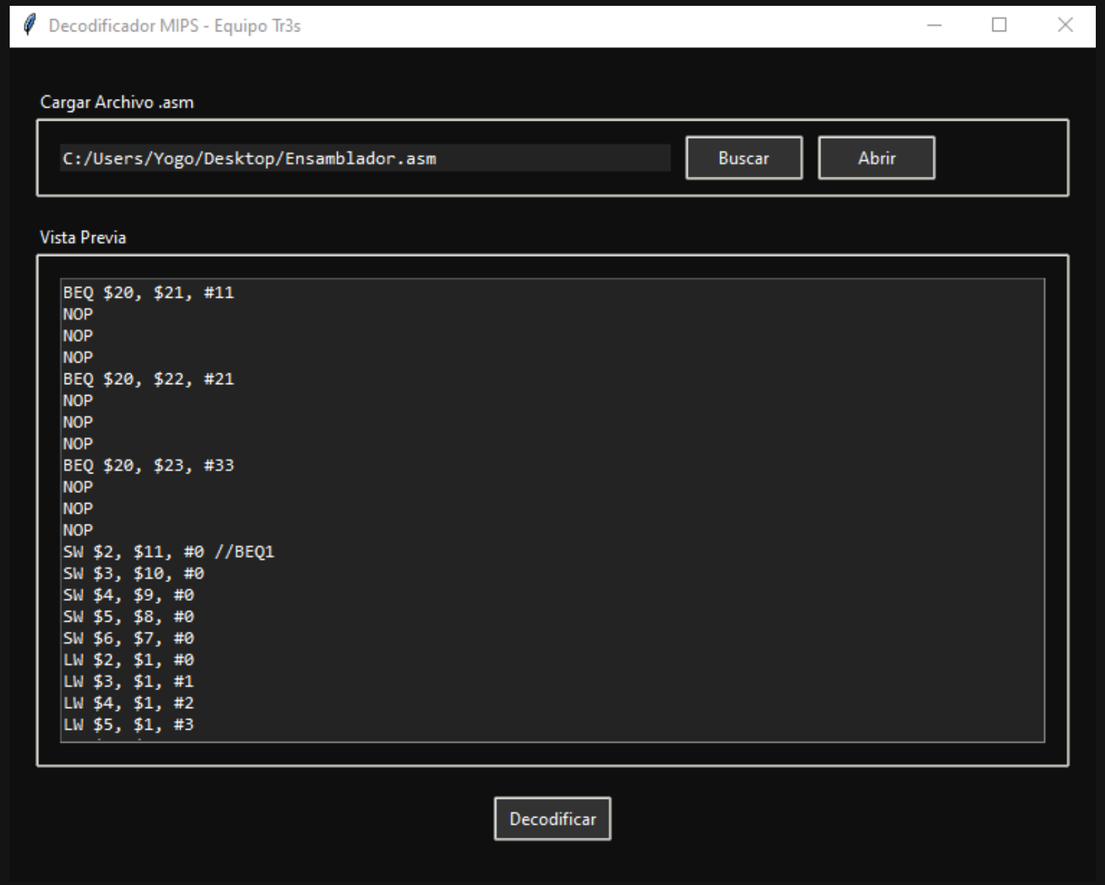
Decodificador

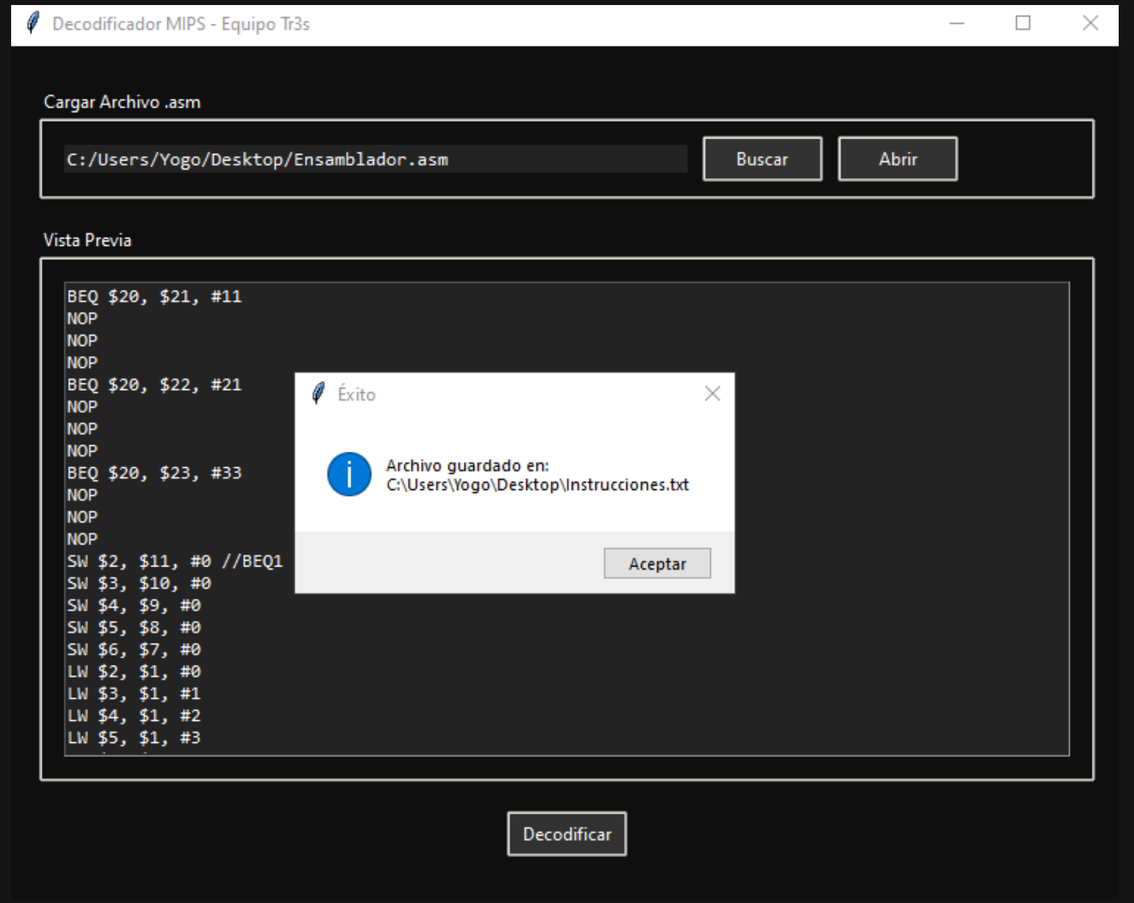
El decodificador nos sirve para ahorrarnos tiempo al decodificar instrucciones para que sean procesadas por nuestro MIPS32.

Con pocas instrucciones no hay problema pero al aumentar el riesgo de error aumenta significativamente.

Por lo que diseñar un decodificador nos ayuda a disminuir los errores y aumentar la productividad del proyecto.

Funcionamiento (capturas)





Código

```
R_TYPE = {
    "ADD": {"opcode": "000000", "funct": "100000", "format": "RD, RS, RT"},
    "SUB": {"opcode": "000000", "funct": "100010", "format": "RD, RS, RT"},
    "MUL": {"opcode": "000000", "funct": "000010", "format": "RD, RS, RT"},
    "SLT": {"opcode": "000000", "funct": "101010", "format": "RD, RS, RT"},
    "AND": {"opcode": "000000", "funct": "100100", "format": "RD, RS, RT"},
    "OR": {"opcode": "000000", "funct": "100101", "format": "RD, RS, RT"},
    "DIVU": {"opcode": "000000", "funct": "011011", "format": "RD, RS, RT"},
    "RSQRT": {"opcode": "000000", "funct": "010110", "format": "RD, RS, RT"},
    "NOP": {"opcode": "000000", "funct": "000000", "format": ""}
}

I_TYPE = {
    "ADDI": {"opcode": "001000", "format": "RT, RS, #inmediato"},
    "ORI": {"opcode": "001101", "format": "RT, RS, #inmediato"},
    "ANDI": {"opcode": "001100", "format": "RT, RS, #inmediato"},
    "SW": {"opcode": "101011", "format": "RT, RS, #inmediato"},
    "LW": {"opcode": "100011", "format": "RT, RS, #inmediato"},
    "BEQ": {"opcode": "000100", "format": "RT, RS, #inmediato"}
}

J_TYPE = {
    "J": {"opcode": "000010", "format": "#target"}
}
```

Algoritmo

Menú con 3 algoritmos

Objetivo

Para verificar la correcta implentación de nuestro datapath, vamos a realizar un menú de operaciones que funciona como un switch case en C. Este algoritmo usará las instrucciones de los 3 distintos tipos de instrucciones (R , I y J).

El propósito es que se ingrese en una dirección del banco de registrar mediante la lectura de un archivo, el número de algoritmo para realizar y después al cargar la simulación.

Código ensamblador

```
BEQ   $20,   $21,
#11
NOP
NOP
NOP
BEQ   $20,   $22,
#21
NOP
NOP
NOP
BEQ   $20,   $23,
#33
NOP
NOP
NOP
```

```
BEQ $20, $21, #11
NOP
NOP
NOP
BEQ $20, $22, #21
NOP
NOP
NOP
BEQ $20, $23, #33
NOP
NOP
NOP
SW $2, $11, #0 //BEQI
SW $3, $10, #0
SW $4, $9, #0
SW $5, $8, #0
SW $6, $7, #0
LW $2, $1, #0
LW $3, $1, #1
LW $4, $1, #2
LW $5, $1, #3
LW $6, $1, #4
J #100
NOP
NOP
NOP
MUL $10, $2, $6 //BEQ2
MUL $11, $3, $8
MUL $12, $2, $7
MUL $13, $3, $9
MUL $14, $4, $6
MUL $15, $5, $8
MUL $16, $4, $7
MUL $17, $5, $9
ADD $25, $10, $11
ADD $27, $12, $13
ADD $28, $14, $15
ADD $29, $16, $17
J #100
NOP
NOP
NOP
RSQRT $12, $2, $0 //BEQ3
ADDI $3, $1, #0 //Inicializar
NOP
NOP
ADDI $3, $3, #1
NOP
NOP
BEQ $2, $3, #18
NOP
NOP
NOP
DIVU $4, $2, $3
NOP
NOP
BEQ $0, $4, #11
NOP
NOP
NOP
BEQ $12, $3, #7
NOP
NOP
NOP
J #46 //salta a ADDI $3, $3, #1
NOP
NOP
NOP
SLT $10, $0, $4
J #100
NOP
NOP
NOP
```

Funcionamiento

Para hacer la estructura switch case utilizamos las instrucciones BEQ donde con un selector ubicado en la dirección $20 definimos el BEQ que tomará el flujo del programa.

Si el selector no es 1, 2 o 3 entra al default que es nuestro primer algoritmo.

Estructura del Banco de Registros

Case 1 invertir arreglo

Address	Dato	
$0	0	Zero
$1	1	Base pointer
$2	5	
$3	6	
$4	7	Arreglo a invertir (aquí se guarda ya invertido)
$5	8	
$6	2	
$7	1	
$8	2	
$9	3	Direcciones del arreglo a invertir
$10	4	
$11	5	
$12		
$13		
$14		
$15		
$16		
$17		
$18		
$19		
$20	1	Selector
$21	1	Case 1
$22	2	Case 2
$23	3	Case 3
$24		
$25		
$26		
$27		
$28		
$29		
$30		
$31		

Case 2 multiplicación matriz 2x2

Address	Dato		
$0	0	Zero	
$1	1	Base pointer	
$2	5	a1	
$3	6	b1	Matriz 1
$4	7	c1	
$5	8	d1	
$6	2	a2	
$7	1	b2	Matriz 2
$8	2	c2	
$9	3	d2	
$10	4	a1a2	
$11	5	b1c2	
$12		a1b2	
$13		b1d2	Resultado Multiplicaciones
$14		c1a2	
$15		d1c2	
$16		c1b2	
$17		d1d2	
$18			
$19			
$20	2	Selector	
$21	1	Case 1	
$22	2	Case 2	
$23	3	Case 3	
$24			
$25			
$26		a3	
$27		b3	Matriz 3 resultante
$28		c3	
$29		d3	
$30			
$31			

Case 3 saber si un número es primo

Address	Dato	
$0	0	Zero
$1	1	Base pointer
$2	5	Es primo?
$3	6	Contador i
$4	7	Módulo de $2 (resultado)
$5	8	
$6	2	
$7	1	
$8	2	
$9	3	
$10	4	1:Es primo, 0:No es primo
$11	5	
$12		
$13		
$14		
$15		
$16		
$17		
$18		
$19		
$20	3	Selector
$21	1	Case 1
$22	2	Case 2
$23	3	Case 3
$24		
$25		
$26		
$27		
$28		
$29		
$30		
$31		

1-Invertir arreglo

Descripción

Invierte un arreglo de 5 espacios y lo guarda en su dirección original.

Código ensamblador

```
SW $2, $11, #0 //BEQ1
SW $3, $10, #0
SW $4, $9, #0
SW $5, $8, #0
SW $6, $7, #0
LW $2, $1, #0
LW $3, $1, #1
LW $4, $1, #2
LW $5, $1, #3
LW $6, $1, #4
J #100
NOP
NOP
NOP
```

```
BEQ $20, $21, #11
NOP
NOP
NOP
BEQ $20, $22, #21
NOP
NOP
NOP
BEQ $20, $23, #33
NOP
NOP
NOP
SW $2, $11, #0 //BEQ1
SW $3, $10, #0
SW $4, $9, #0
SW $5, $8, #0
SW $6, $7, #0
LW $2, $1, #0
LW $3, $1, #1
LW $4, $1, #2
LW $5, $1, #3
LW $6, $1, #4
J #100
NOP
NOP
NOP
MUL $10, $2, $6 //BEQ2
MUL $11, $3, $8
MUL $12, $2, $7
MUL $13, $3, $9
MUL $14, $4, $6
MUL $15, $5, $8
MUL $16, $4, $7
MUL $17, $5, $9
ADD $26, $10, $11
ADD $27, $12, $13
ADD $28, $14, $15
ADD $29, $16, $17
J #100
NOP
NOP
NOP
RSQRT $12, $2, $0 //BEQ3
ADDI $3, $1, #0 //inicializar
NOP
NOP
ADDI $3, $3, #1
NOP
NOP
BEQ $2, $3, #18
NOP
NOP
NOP
DIVU $4, $2, $3
NOP
NOP
BEQ $0, $4, #11
NOP
NOP
NOP
BEQ $12, $3, #7
NOP
NOP
NOP
J #46 //salta a ADDI $3, $3, #1
NOP
NOP
NOP
SLT $10, $0, $4
J #100
NOP
NOP
NOP
```

Funcionamiento

Este algoritmo es estático y sirve para invertir un arreglo de 5 espacios que va de $2 hasta $6 definido en el banco de registros.

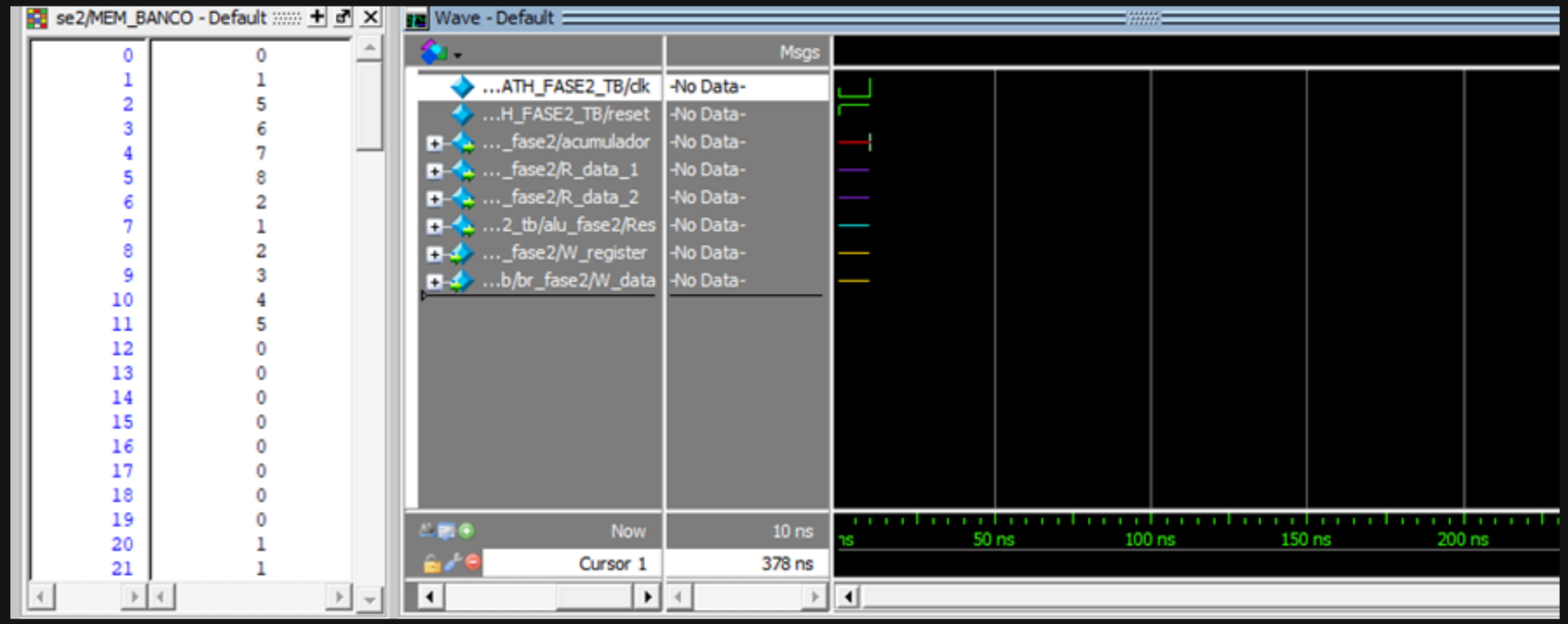
Pasamos el dato de estas direcciones a la memoria de datos pero intercambiados, esto con ayuda de las direcciones guardadas de $7 a $11.

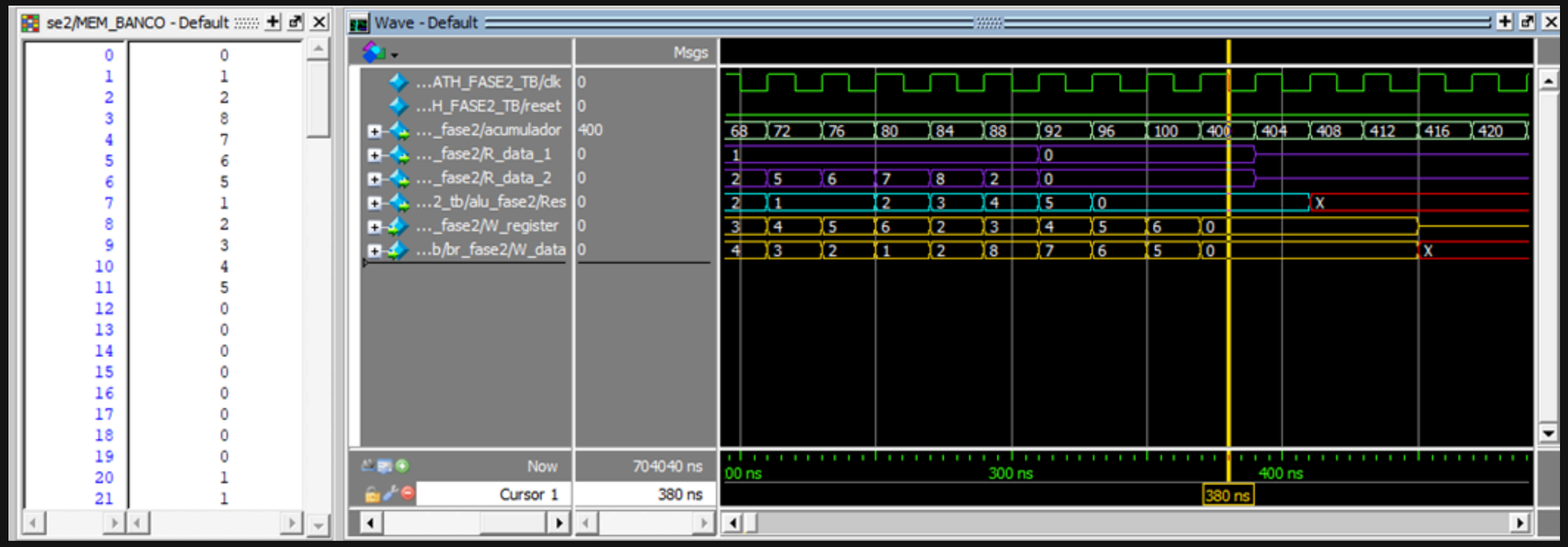
Finalmente regresamos el arreglo a su lugar original pero invertido y saltamos a la instrucción 100 (fin del programa).

Estructura del Banco de Registros

Case 1 invertir arreglo		
Address	Dato	
$0	0	Zero
$1	1	Base pointer
$2	5	
$3	6	
$4	7	Arreglo a invertir (aquí se guarda ya invertido)
$5	8	
$6	2	
$7	1	
$8	2	
$9	3	Direcciones del arreglo a invertir
$10	4	
$11	5	
$12		
$13		
$14		
$15		
$16		
$17		
$18		
$19		
$20	1	Selector
$21	1	Case 1
$22	2	Case 2
$23	3	Case 3
$24		
$25		
$26		
$27		
$28		
$29		
$30		
$31		

Simulaciones de comprobación





2-Multiplicación de matrices

Descripción

Multiplica una matriz A por una matriz B ambas de 2x2.

Guarda el resultado en una matriz C de 2x2.

Código ensamblador

```
MUL $10, $2, $6 //BEQ2
MUL $11, $3, $8
MUL $12, $2, $7
MUL $13, $3, $9
MUL $14, $4, $6
MUL $15, $5, $8
MUL $16, $4, $7
MUL $17, $5, $9
ADD $26, $10, $11
ADD $27, $12, $13
ADD $28, $14, $15
ADD $29, $16, $17
J #100
NOP
NOP
NOP
```

```
BEQ $20, $21, #11
NOP
NOP
NOP
BEQ $20, $22, #21
NOP
NOP
NOP
BEQ $20, $23, #33
NOP
NOP
NOP
SW $2, $11, #0 //BEQ1
SW $3, $10, #0
SW $4, $9, #0
SW $5, $8, #0
SW $6, $7, #0
LW $2, $1, #0
LW $3, $1, #1
LW $4, $1, #2
LW $5, $1, #3
LW $6, $1, #4
J #100
NOP
NOP
NOP
MUL $10, $2, $6 //BEQ2
MUL $11, $3, $8
MUL $12, $2, $7
MUL $13, $3, $9
MUL $14, $4, $6
MUL $15, $5, $8
MUL $16, $4, $7
MUL $17, $5, $9
ADD $26, $10, $11
ADD $27, $12, $13
ADD $28, $14, $15
ADD $29, $16, $17
J #100
NOP
NOP
NOP
RSQRT $12, $2, $0 //BEQ3
ADDI $3, $1, #0 //inicializar
NOP
NOP
ADDI $3, $3, #1
NOP
NOP
BEQ $2, $3, #18
NOP
NOP
NOP
DIVU $4, $2, $3
NOP
NOP
BEQ $0, $4, #11
NOP
NOP
NOP
BEQ $12, $3, #7
NOP
NOP
NOP
J #46 //salta a ADDI $3, $3, #1
NOP
NOP
NOP
SLT $10, $0, $4
J #100
NOP
NOP
NOP
```

Funcionamiento

Matriz A: $2 a $5
Matriz B: $6 a $9
Matriz C: $26 a $29

Usamos los registros $10 a $17 para guardar temporalmente el resultado de las multiplicaciones.

Para ello simplemente seguimos la fórmula para multiplicar matrices de 2x2.

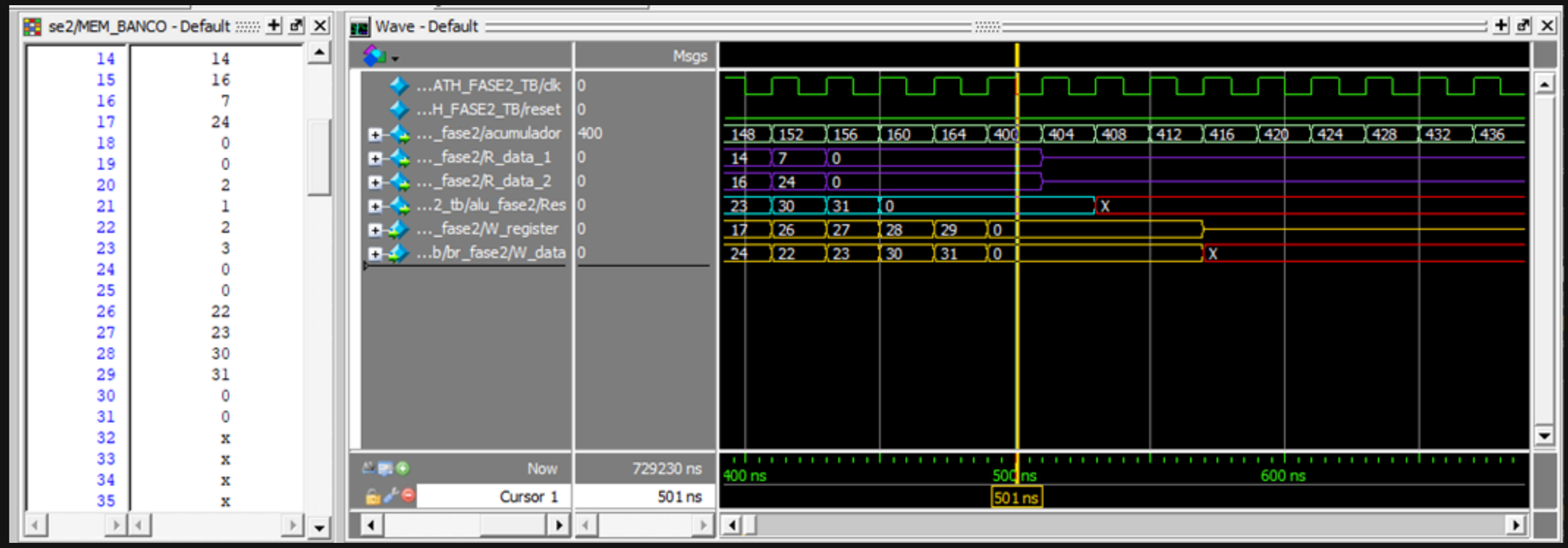
Guardamos el resultado en la matriz C.

$$\begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} x \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix} = \begin{pmatrix} a_1a_2 + b_1c_2 & a_1b_2 + b_1d_2 \\ c_1a_2 + d_1c_2 & c_1b_2 + d_1d_2 \end{pmatrix}$$

Estructura del Banco de Registros

Case 2 multiplicación matriz 2x2				
Address	Dato			
$0	0	Zero		
$1	1	Base pointer		
$2	5	a1		
$3	6	b1	Matriz 1	
$4	7	c1		
$5	8	d1		
$6	2	a2		
$7	1	b2	Matriz 2	
$8	2	c2		
$9	3	d2		
$10	4	a1a2		
$11	5	b1c2		
$12		a1b2		
$13		b1d2	Resultado Multiplicaciones	
$14		c1a2		
$15		d1c2		
$16		c1b2		
$17		d1d2		
$18				
$19				
$20	2	Selector		
$21	1	Case 1		
$22	2	Case 2		
$23	3	Case 3		
$24				
$25				
$26		a3		
$27		b3	Matriz 3 resultante	
$28		c3		
$29		d3		
$30				
$31				

Simulaciones de comprobación



3-Número es primo

Descripción

Devuelve un 1 si el número ingresado es primo, si no devuelve un 0.

Código ensamblador

```
RSQRT $12, $2, $0 //BEQ3
ADDI $3, $1, #0 //Inicialziar
NOP
NOP
ADDI $3, $3, #1
NOP
NOP
BEQ $2, $3, #18
NOP
NOP
NOP
DIVU $4, $2, $3
NOP
NOP
BEQ $0, $4, #11
NOP
NOP
NOP
```

```
BEQ $12, $3, #7
NOP
NOP
NOP
J  #46  //salta  a  ADDI
$3, $3, #1
NOP
NOP
NOP
SLT $10, $0, $4
J #100
NOP
NOP
NOP
```

```
BEQ $20, $21, #11
NOP
NOP
NOP
BEQ $20, $22, #21
NOP
NOP
NOP
BEQ $20, $23, #33
NOP
NOP
NOP
SW $2, $11, #0 //BEQ1
SW $3, $10, #0
SW $4, $9, #0
SW $5, $8, #0
SW $6, $7, #0
LW $2, $1, #0
LW $3, $1, #1
LW $4, $1, #2
LW $5, $1, #3
LW $6, $1, #4
J #100
NOP
NOP
NOP
MUL $10, $2, $6 //BEQ2
MUL $11, $3, $8
MUL $12, $2, $7
MUL $13, $3, $9
MUL $14, $4, $6
MUL $15, $5, $8
MUL $16, $4, $7
MUL $17, $5, $9
ADD $26, $10, $11
ADD $27, $12, $13
ADD $28, $14, $15
ADD $29, $16, $17
J #100
NOP
NOP
NOP
RSQRT $12, $2, $0 //BEQ3
ADDI $3, $1, #0 //Inicialziar
NOP
NOP
ADDI $3, $3, #1
NOP
NOP
NOP
BEQ $2, $3, #18
NOP
NOP
NOP
DIVU $4, $2, $3
NOP
NOP
BEQ $0, $4, #11
NOP
NOP
NOP
BEQ $12, $3, #7
NOP
NOP
NOP
J #46 //salta a ADDI $3, $3, #1
NOP
NOP
NOP
SLT $10, $0, $4
J #100
NOP
NOP
NOP
```

Funcionamiento

Ingresamos un número a evaluar en el registro $2

Inicializamos un contador i en el registro $3

Obtenemos la raíz cuadrada del número ingresado y la guardamos en el registro $12

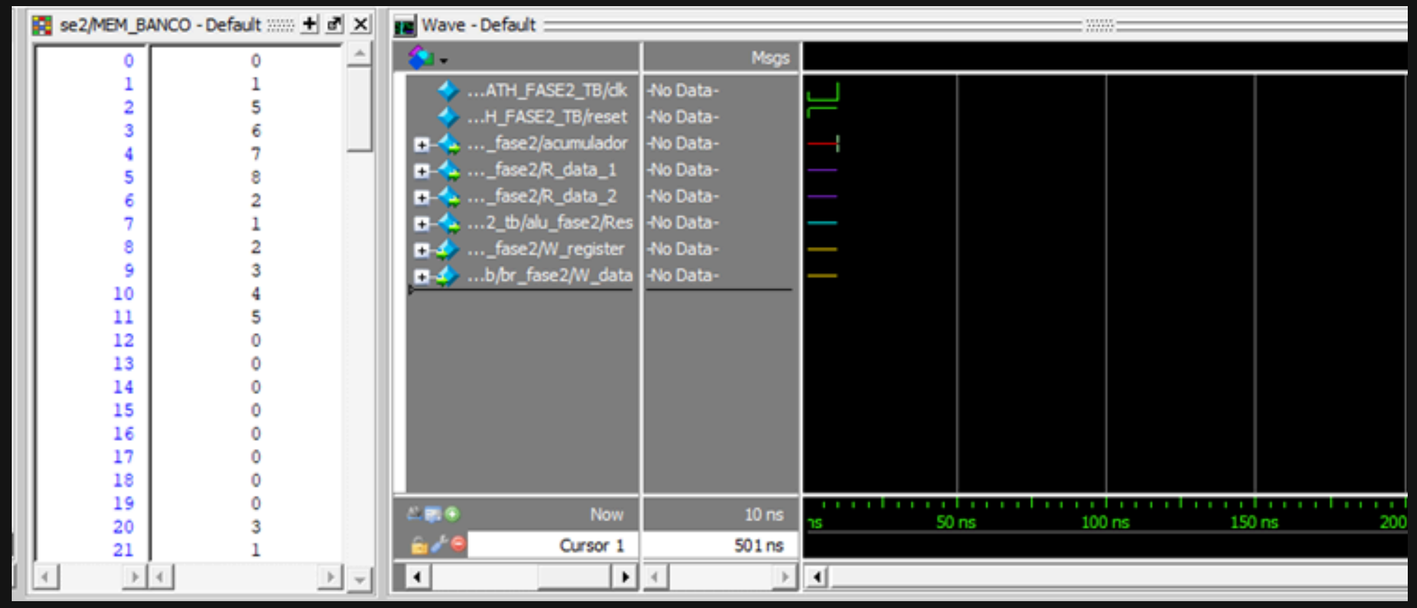
- Si el número es 2 salimos del programa con un BEQ (caso especial)
- Si el módulo de n%i=0 entonces se trata de un número no primo y salimos del programa con el segundo BEQ
- De no caer en ninguna de las condiciones anteriores cuando sqrt(n) sea igual que i salimos del programa, tercer BEQ (es primo)

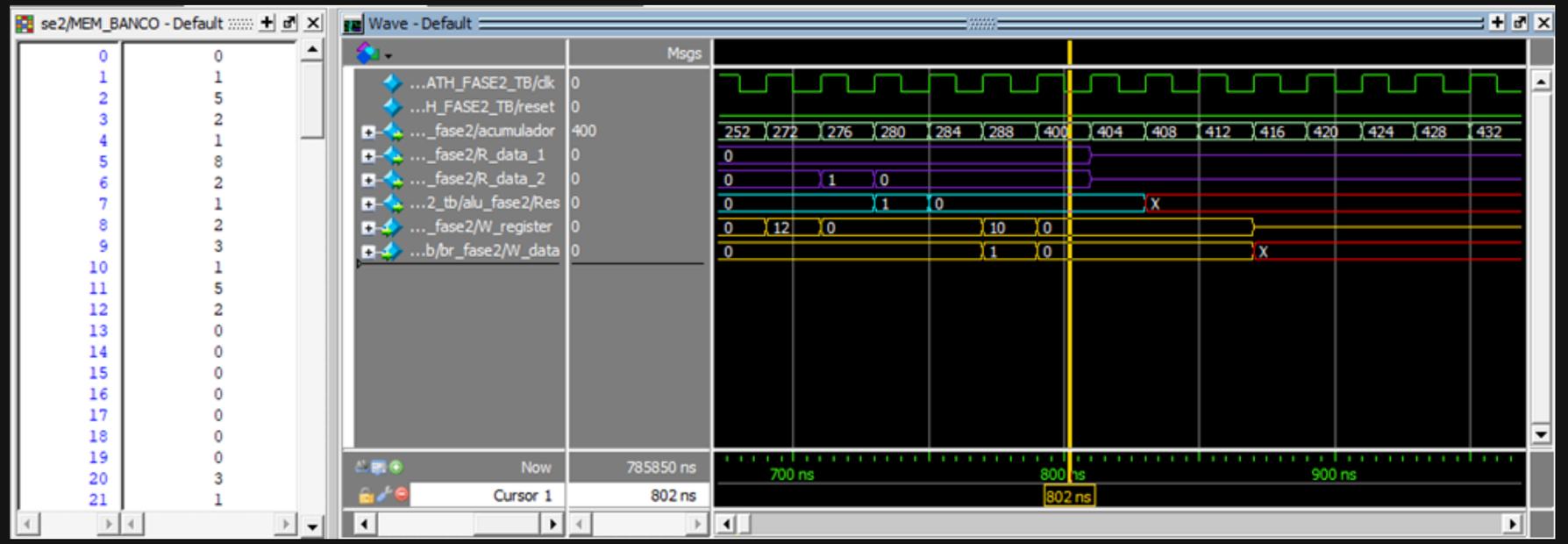
SLT en $10, si 0<módulo de n es primo: 1
si 0<0(MOD=0) no es primo:0

Estructura del Banco de Registros

Case 3 saber si un número es primo		
Address	Dato	
$0	0	Zero
$1	1	Base pointer
$2	5	Es primo?
$3	6	Contador i
$4	7	Módulo de $2 (resultado)
$5	8	
$6	2	
$7	1	
$8	2	
$9	3	
$10	4	1:Es primo, 0:No es primo
$11	5	
$12		
$13		
$14		
$15		
$16		
$17		
$18		
$19		
$20	3	Selector
$21	1	Case 1
$22	2	Case 2
$23	3	Case 3
$24		
$25		
$26		
$27		
$28		
$29		
$30		
$31		

Simulaciones de comprobación





Gracias por la atención

¿Preguntas?