Universidad de Guadalajara

CUCEI
Seminario de Solución de Problemas de Arquitectura de Computadoras
D14
Profesor: López Arce Delgado Jorge Ernesto

Proyecto final

Fase 3

Integrantes:

- Rubio Andrade Athziri Magdalena
- Casas Chavarría Diego Maximiliano
- García Ramírez José Manuel

Actividades a realizar

1. Código Verilog - Implementación del “single datapath” de MIPS de 32 bits para ejecutar instrucciones tipo J. (Debe ejecutar correctamente el archivo de validación adjunto)

2. Reporte – a. Redacción y descripción del desarrollo de los módulos que tienen el datapath y los elementos nuevos para instrucciones tipo J. b. Redacción de los elementos teóricos y diagrama de flujo del algoritmo a implementar en código ensamblador.

3. Programa ensamblador – a. Comenzar a trabajar con el decodificador de lenguaje ensamblador a código binario, la final debe crear el archivo a cargar en la memoria de instrucciones (extensión .mem o .txt), y en caso de ser necesario también el archivo de inicialización del Banco de Registro

4. Presentación (subirla a Moodle), esta debe ser un resumen de los tres puntos anteriores, se debe de mostrar que han investigado de la parte teórica del algoritmo a implementar en ensamblador, que han codificado en esta fase 3 así como la validación del datapath para dicha fase.

Bitácora (días fase 1):

- 18 de Junio del 2021
- 19 de Junio del 2021
- 20 de Junio del 2021
- 21 de Junio del 2021
- 22 de Junio del 2021
- 23 de Junio del 2021
- 24 de Junio del 2021

18 de Junio de 2021

- Horario:

  De 10:00 am a 1:00 pm

- Descripción:

  En este día lo único que acordamos fue dedicar los dos días siguientes, 19y 20 de Junio a leer del libro “Computer Organization and Design, the hardware/software interface, David A. Patterson; John L. Henessy, Fifth Ed.” También nos pusimos de acuerdo para conseguir información para implementar nuestro algoritmo en código ensamblador CRC.

19 de Junio de 2021

- Horario:

  12 pm - 6pm

- Descripción:

  Leer del libro “Computer Organization and Design, the hardware/software interface, David A. Patterson; John L. Henessy, Fifth Ed.” Específicamente en las páginas correspondientes a nuestra fase 3.

20 de Junio de 2021

- Horario:

  Todo el día

- Descripción:

  Leer del libro “Computer Organization and Design, the hardware/software interface, David A. Patterson; John L. Henessy, Fifth Ed.” páginas que describen la tercera fase del proyecto.Comienzo de investigacion del lenguaje ensamblador CRC.

21 de Junio de 2021

- Horario:

  De 9:00 am a 11:00 pm

- Descripción:

  En este transcurso de tiempo definimos, que aunque quisimos reunirnos más de una vez, nuestros horarios no terminaba de coincidir, así que cada uno trabajó por su cuenta en su parte correspondiente.

22 de Junio de 2021

- Horario:

  9:00 am a 12:00 am

  Descripción:

  El dia de hoy nos dedicamos al avance del reporte con ayuda de todos, dando aprobación a las opciones de información que se necesita para el reporte y asi todos poder tener acceso a información clara sobre nuestros módulos y su implementación.

  También se hizo seguimiento a la investigación para la implementación del algoritmo de CRC.

  A su vez la parte de la implementación de la instrucción J tuvo avance.

23 de Junio de 2021

- Horario:

  9:00 am a 10:00 pm

- Descripción:
- Se comenzó con la implementación del algoritmo del código ensamblador y máquina,avance del reporte y del código verilog.

24 de Junio de 2021

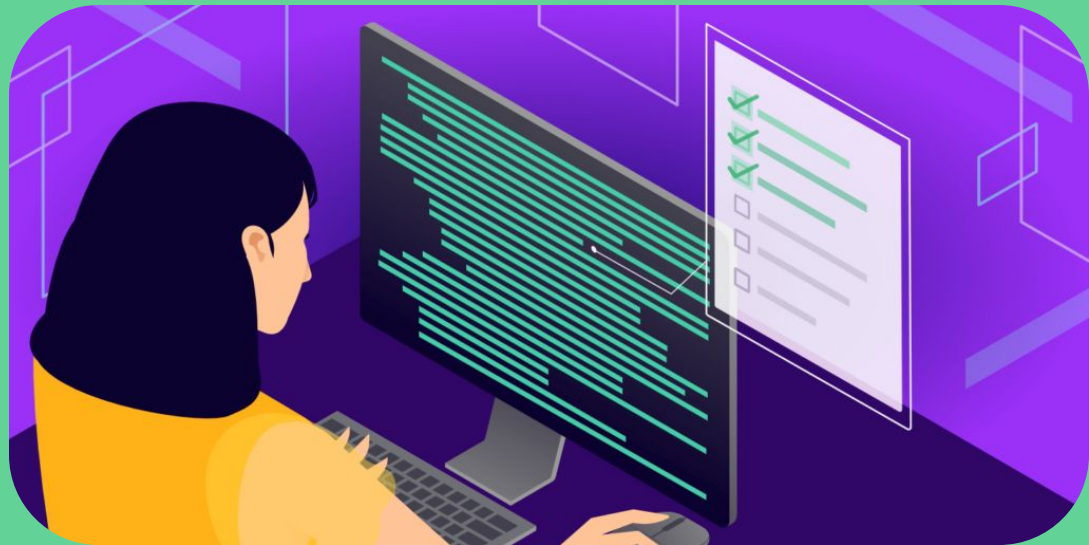
- Horario:

  8:00 am a 11:30 pm

- Descripción:

  Este día lo dedicamos a juntar todas las partes y subirlas a Github, también fue el día en que se terminó de realizar la presentación para poder describir todas sus partes y tener completa cada parte.

Código verilog



Unidad de control

- OPCODE (entrada de seis bits)
- MemREG (salida de 1 bit)
- RegWRITE (salida de 1 bit)
- MemWRITE(salida de 1 bit)
- Branch(salida de 1 bit)
- MemRead(salida de 1 bit)
- ALUSrc(salida de 1 bit)
- RegDst(salida de 1 bit)
- ALUOP(salida de 3 bits)

La entrada define por ahora el tipo de instrucción “R”.

- Jump(Salida 1 bit)

unit_control.v | DatapathX.v | buffer2.v

```
1  `timescale 1ns/1ns
2
3  module unit_control(
4      input[5:0] OPCODE,
5      output reg MemREG, RegWRITE, MemWRITE,
6      output reg Branch, MemRead, ALUSrc, RegDst, Jump,
7      output reg [2:0] ALUOP
8  );
9
10     always @* begin
11         case (OPCODE)
12             6'b000000: begin //TIPO R
13                 MemREG = 0;
14                 RegWRITE = 1;
15                 MemWRITE = 0;
16                 Branch = 0;
17                 MemRead = 0;
18                 ALUSrc = 0;
19                 RegDst = 1;
20                 Jump = 0;
21                 ALUOP = 3'b010;
22             end
```

```
100             6'b000010: begin //J
101                 MemREG = 1'bx;
102                 RegWRITE = 1'bx;
103                 MemWRITE = 1'bx;
104                 Branch = 1'bx;
105                 MemRead = 1'bx;
106                 ALUSrc = 1'bx;
107                 RegDst = 1'bx;
108                 Jump = 1; //Ejecuta el jump
109                 ALUOP = 3'bx;
110             end
```

Selección de la unidad de control para jump

Shift left 2

- X (entrada de 26 bits)
- Y (salida de 28 bits)

Se realiza un corrimiento a dos bits de “X” y se deposita en “Y”.

Shift_left_2_JUMP.v

```
1  `timescale 1ns/1ns
2
3  module SL2_J(
4      input [25:0] X,
5      output [27:0] Y
6  );
7
8
9  assign Y = {X, 1'b0, 1'b0};
10
11 endmodule
12
```

DatapathX.v

ALU Control

- FUNCTION(entrada de 6 bits)
- ALUOP(entrada de 3 bits)
- OP(salida de 4 bits)

Se evalúa con la ALUOP el tipo de instrucción y con FUNCTION la operación a realizar ya sea instrucciones de tipo R,I o J.

DatapathX.v | ALU.v | alu_ctrl.v

```
1  `timescale 1ns/1ns
2
3  module alu_ctrl(
4      input[5:0] FUNCTION,
5      input[2:0] ALUOP,
6      output reg [3:0] OP
7  );
8
9      always @(*) begin
10         case (ALUOP)
11             3'b010: begin
12                 case (FUNCTION)
13                 6'b100000: OP = 4'd0; //ADD
14                 6'b100010: OP = 4'd1; //SUB
15                 6'b100100: OP = 4'd2; //AND
16                 6'b100101: OP = 4'd3; //OR
17                 6'b101010: OP = 4'd4; //slt
18                 6'b011000: OP = 4'd5; //mul
19                 6'b011010: OP = 4'd6; //div
20                 6'b000000: OP = 4'd7; //nop
21                 6'b100110: OP = 4'd8; //XOR
22                 endcase
23             end
24             3'b000: OP = 4'd0; //addi...
25             3'b001: OP = 4'd4; //slti...
26             3'b100: OP = 4'd2; //andi...
27             3'b011: OP = 4'd3; //ori...
28         endcase
29     end
30 endmodule
```

ALU

- X(entrada de 32 bits)
- Y(entrada de 32 bits)
- SEL(entrada de 3 bits)
- R(salida de 32 bits)
- Z_flag(salida de 1 bit)

X y Y son los operandos y el SEL determina el operador, R es la salida y el Z_flag depende de lo anterior.

DatapathX.v | ALU.v | alu_ctrl.v

```
1  `timescale 1ns/1ns
2
3  module ALU(
4      input [31:0] X,
5      input [31:0] Y,
6      input [3:0] SEL,
7      output reg [31:0] R,
8      output reg Z_flag
9  );
10
11 always@* begin
12     case(SEL)
13     //R operando destino, X operando fuente 1,Y operando fuente 2
14         4'd0: R = X + Y; //ADD ADDI LW SW
15         4'd1: R = X - Y; //SUB BEQ
16         4'd2: R = X & Y; //AND ANDI
17         4'd3: R = X | Y; //OR ORI
18         4'd4: R = X < Y; //SLT SLTI
19         4'd5: R = X * Y; //Mul
20         4'd6: R = X / Y; //Div
21         4'd7: R = X << 0; //NOP
22         4'd8: R = X ^ Y; //XOR
23         default: R <= 32'bx;
24     endcase
25
26     Z_flag <=(R) ? 0:1;
27
28 end
29 endmodule
```

Buffer 2

- INB2DirJump(entrada de 32 bits)
- INB2Jump(entrada de 32 bits)

Estos son los cambios correspondientes a este modulo.

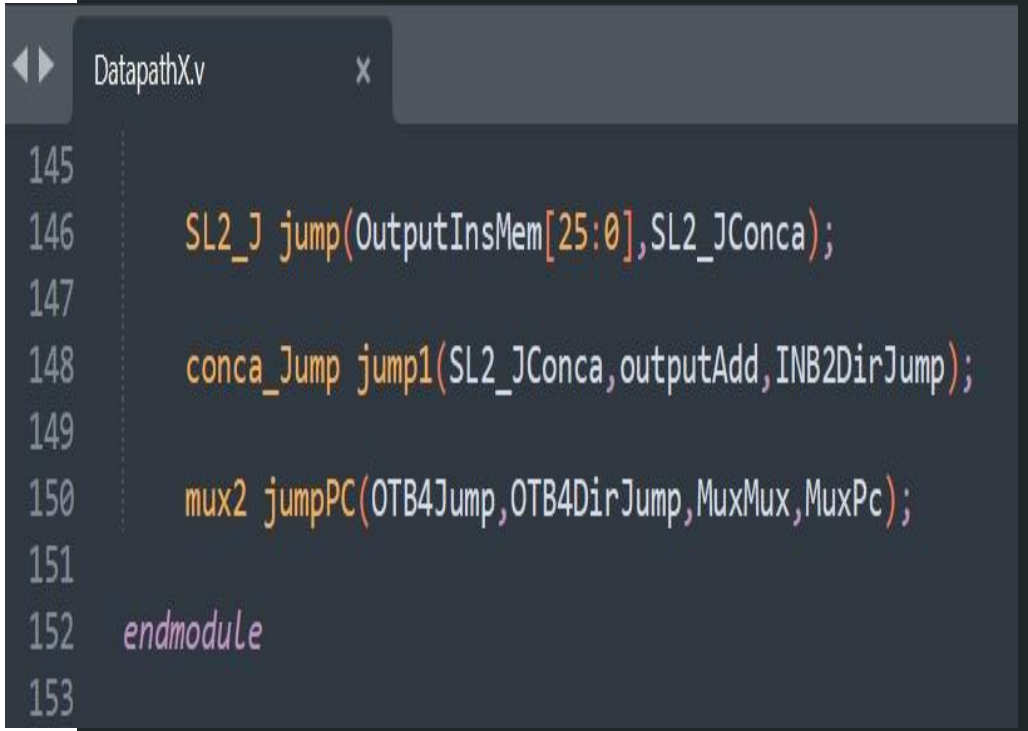
buffer2.v | DatapathX.v | buffer3.v

```
1  `timescale 1ns/1ns
2  //1.Creacion del modulo
3  //Declaracion del modulo con sus entradas y salidas
4  module buffer2
5      (
6      //UNIDAD DE CONTROL
7      //ENTRADAS
8      //WB
9      input INB2MemREG,
10     input INB2RegWRITE,
11     //M
12     input INB2Branch,
13     input INB2MemWRITE,
14     input INB2MemRead,
15     //EX
16     input INB2RegDst,
17     input [2:0]INB2ALUOP,
18     input INB2ALUSrc,
19     //J
20     input [31:0]INB2DirJump,
21     input INB2Jump,
22     //RESTO
23     input clk,
24     input [31:0] inputAddB2,
25     input [31:0] inputRD1,
26     input [31:0] inputRD2,
27     input [31:0] inputSinex,
28     input [4:0] inputRr1,
29     input [4:0] inputRr2,
30     //UNIDAD DE CONTROL
31     //SALIDAS
```

Single Datapath

- clk(entrada de 1 bit)

Se declaran cables de unión, con tamaños, y se realizan todas las instancias.
Adjunto nuestro diagrama a parte.
Datapath, o top level, solo se agregan los módulos nuevos y sus respectivas conexiones, también modificando los buffers ya que agregamos nuevas entradas y salidas.



Investigación

Instrucciones Tipo “J”:

- *Instrucciones Tipo “J”:*

- La instrucción Jump es tipo “J” y es la que hace un salto incondicional o una bifurcación es decir, la instrucción obliga a la máquina a seguir siempre el salto. Para distinguir entre saltos condicionales e incondicionales, el nombre MIPS para este tipo de instrucción es Jump.

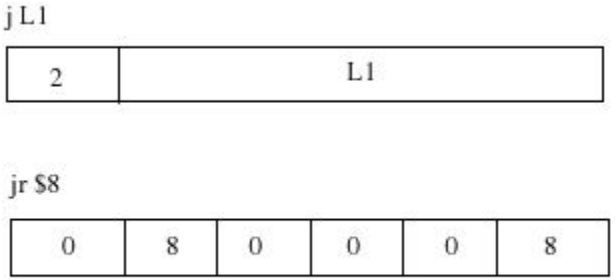
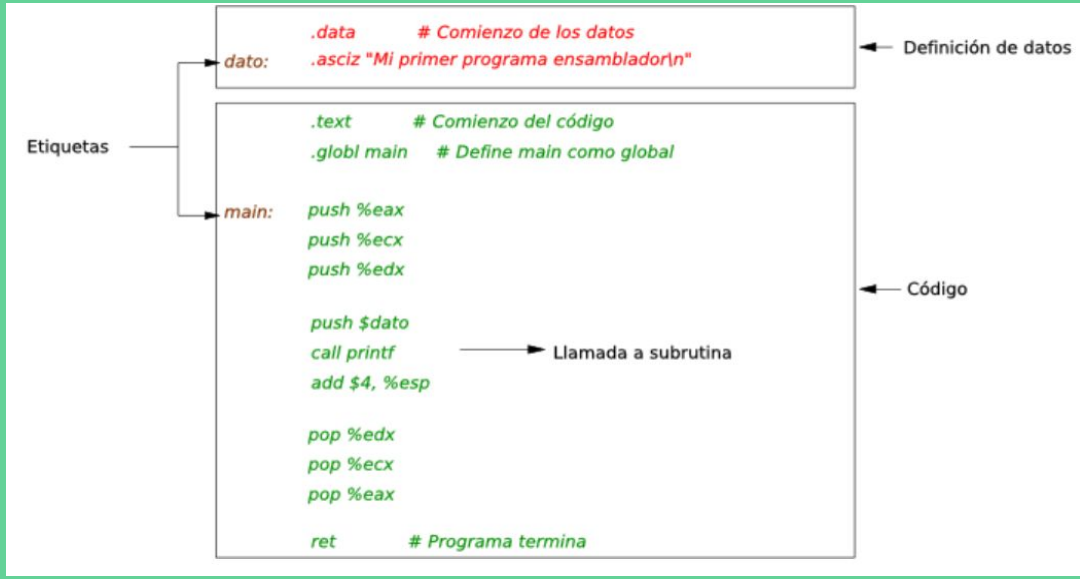


Figura 2.11: Estructura de las instrucciones *jump* y *jump register* en el MIPS

Set de instrucciones fase 3

J	J	j $target

Programa ensamblador



Lenguaje ensamblador

El lenguaje ensamblador trabaja con nemónicos, que son grupos de caracteres alfanuméricos que simbolizan las órdenes o tareas a realizar. La traducción de los nemónicos a código máquina entendible por el microcontrolador la lleva a cabo un programa ensamblador. El programa escrito en lenguaje ensamblador se denomina código fuente (*.asm). El programa ensamblador proporciona a partir de este fichero el correspondiente código máquina, que suele tener la extensión *.hex.

Algoritmo CRC

La **verificación por redundancia cíclica** (CRC) es un código de detección de errores usado frecuentemente en redes digitales y en dispositivos de

almacenamiento para detectar cambios accidentales en los datos.[1] Los bloques de datos ingresados en estos sistemas contiene un *valor de verificación adjunto*, basado en el residuo de una división de polinomios; el cálculo es repetido, y la acción de corrección puede tomarse en contra de los datos presuntamente corruptos en caso de que el valor de verificación no concuerde. Este código es un tipo de función que recibe un flujo de datos de cualquier longitud como entrada y devuelve un valor de longitud fija como salida. El término suele ser usado para designar tanto a la función como a su resultado. Pueden ser usadas como suma de verificación para detectar la alteración de datos durante su transmisión o almacenamiento. Las CRC son populares porque su implementación en *hardware* binario es simple, son fáciles de analizar matemáticamente y son particularmente efectivas para errores ocasionados por ruido en los canales de transmisión. La CRC fue inventada y propuesta por W. Wesley Peterson

Ejemplo

Sea r el grado de G(x).Agragrar r bits (ceros) al extremo derecho de la trama, de tal manera que ahora tengamos: trama + bits, y corresponda al polinomio x^y M(x)

El emisor que quiere enviar la trama: 1101

Siendo G(x)=x^3+1 donde r=3.(bits de redundancia)

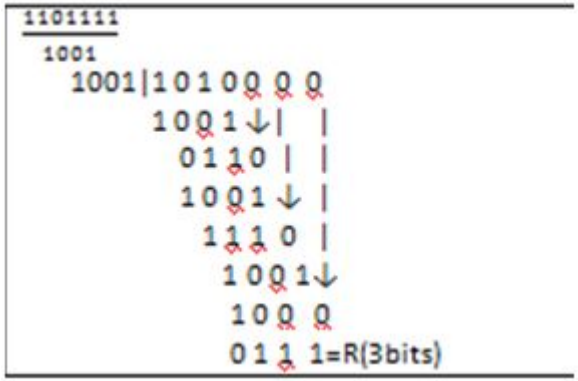
Entonces x^y M(x)=1101+000=11011000

Ahora bien dividiremos x^y M(x)/G(x)

G(x)=x^3+1(Polinomio divisor)

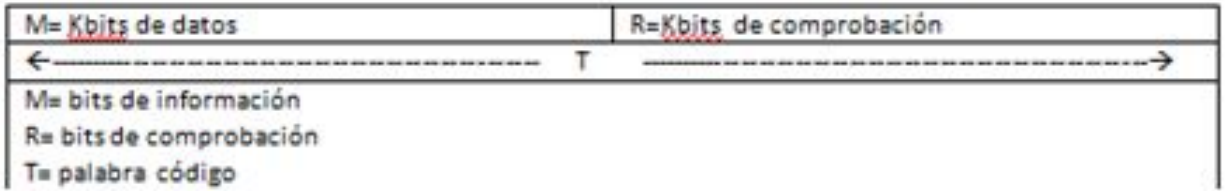
G(x)= 1x^3+0x^2+0x^1+1x^1

G(x)=1001 (Binario divisor)



Continuación

El destinatario recibirá:



Donde:

M=1101

R=111

T=1101111

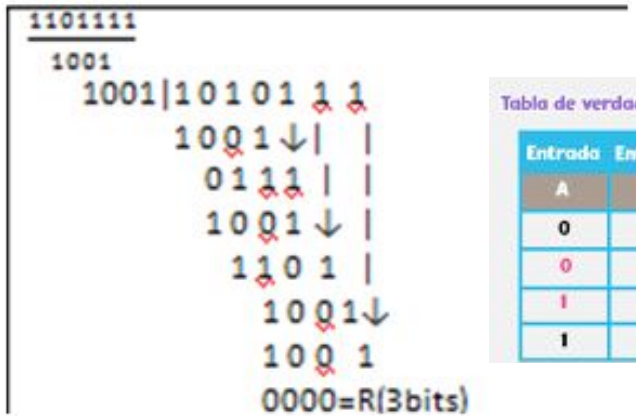


Tabla de verdad de la compuerta XOR

Entrada A	Entrada B	Salida A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Si el residuo arroja 0 quiere decir que no hubo errores.

Algoritmo ensamblador CRC

	BANK	
i	0	1
l	1	4
t	2	0
	3	0
	4	0
	5	0
	6	0
	7	1
	8	0
	9	1
	10	0
	11	0
	12	1
	13	0

DATA	
0	1
1	1
2	0
3	1
4	1
5	0
6	0
7	1

```
#i=$0
#l=límite=$1
#t=$2 (registro temporal)
#Cargamos el mensaje en la memoria:(1101)

lw $0 , $5 , $0  #El dato 0 se guarda en el registro 5 en el banco
lw $0 , $6 , $1  #El dato 1 se guarda en el registro 6 en el banco
lw $0 , $7 , $2  #El dato 2 se guarda en el registro 7 en el banco
lw $0 , $8 , $3  #El dato 3 se guarda en el registro 8 en el banco
```

```
#Ahora cargamos el siguiente mensaje en los siguientes 4 registros
lw $0 , $9 , $4  #El dato 4 se guarda en el registro 9 en el banco
lw $0 , $10 , $5  #El dato 5 se guarda en el registro 10 en el banco
lw $0 , $11 , $6  #El dato 6 se guarda en el registro 11 en el banco
lw $0 , $12 , $7  #El dato 7 se guarda en el registro 12 en el banco

#Una vez guardados los datos en los registros correspondientes, se implementan las
operaciones necesaria para nuestra primera división sintética

#contador:
addi $i , $0 , 0 #Inicio
addi $l , $0 , 4 #Límite

#for:

xor $14 , $18 , $5  #Se aplica  Xor a los registros 6 y 10  para almacenar el resultado en el
registro 5 del banco
xor $7 , $11 , $6  #Se aplica  Xor a los registros 7 y 11  para almacenar el resultado en el
registro 6 del banco
xor $8 , $12 , $7  #Se aplica  Xor a los registros 8 y 12  para almacenar el resultado en el
registro 7 del banco
sub $i , $i , $8 #
beq $i , $l , 2 #si nuestro registro (i) es igual al registro (l) hay que salir del ciclo for y saltar
dos operaciones j y addi
#suma de nuestro índice (++)
addi $i , $i , 1 #el registro (i)  lo guardas en el registro i y sumamos el 1 y si no son iguales
se hace i++  y pasamos a la siguiente instrucción j
j 13 #Regresamos al inicio del ciclo, brincamos a la dirección 13 donde se tiene el inicio
de ciclo
```

DATA

	BANK	
i	0	1
l	1	4
t	2	0
	13	1
	14	1
	15	0
	16	1
	17	1
	18	0
	19	0
	20	1

8	0
9	1
10	1
11	1
12	1
13	0
14	0
15	1

#Segunda división sintética con residuo obtenido de la anterior división sintética
#Cargamos el mensaje en la memoria:(0111)

lw $0 , $13 , $8 #El dato 8 se guarda en el registro 13 en el banco
lw $0 , $14 , $9 #El dato 9 se guarda en el registro 14 en el banco
lw $0 , $15 , $10 #El dato 10 se guarda en el registro 15 en el banco
lw $0 , $16 , $11 #El dato 11 se guarda en el registro 16 en el banco

#Ahora cargamos el siguiente mensaje en los siguientes 4 registros
#Se cargan los datos y donde se obtiene el CRC (polinomio x^3+1=1001)

```
lw $0 , $17 , $12  #El dato 12 se guarda en el registro 17 en el banco
lw $0 , $18 , $13  #El dato 13 se guarda en el registro 18 en el banco
lw $0 , $19 , $14  #El dato 14 se guarda en el registro 19 en el banco
lw $0 , $20 , $15#El dato 15 se guarda en el registro 20 en el banco
#Una vez guardados los datos en los registros correspondientes, se implementan las operaciones necesaria para nuestra primera división sintética

#contador:

addi $i , $0 , 0 #Inicio
addi $l , $0 , 4 #Límite

#for:

xor $14 , $18 , $13  #Se aplica  Xor a los registros 14 y 18  para almacenar el resultado en el registro 13 del banco
xor $15 , $19 , $14  #Se aplica  Xor a los registros 15 y 19  para almacenar el resultado en el registro 14 del banco
xor $16 , $20 , $15  #Se aplica  Xor a los registros 16 y 20  para almacenar el resultado en el registro 15 del banco
sub $i , $i , $12
beq $i , $l , 2 #si nuestro registro (i) es igual al registro (l) hay que salir del ciclo for y saltar dos operaciones j y addi
#suma de nuestro índice (++)
addi $i , $i , 1 #el registro (i)  lo guardas en el registro i y sumamos el 1 y si no son iguales se hace i++  y pasamos a la siguiente instrucción j
j 13 #Regresamos al inicio del ciclo, brincamos a la dirección 13 donde se tiene el inicio de ciclo
```