ARQUITECTURA DEL COMPUTADOR

Entrega 3

Presentado por:
Carlos Eduardo Jaramillo Franco
Dayan Fernando Fernández Pacho
Juan Sebastián Vargas Arévalo
Francisco Javier Tabares Arcila

Docente: Gonzales Antonio Ricardo

Universidad Politécnico Grancolombiano
Ingeniería diseño e innovación
Bogotá 2021

TABLA DE CONTENIDOS

INTRODUCCIÓN

Hacer mucho quedo atrás el tiempo en que el computador o el ambiente digital estaba reservado para algunos pocos, hoy por hoy es innegable la profunda influencia que dichos elementos juegan en nuestra cotidianidad, así bien queda claro que un profesional en software debe tener un conocimiento profundo de las herramientas que utiliza, tanto a nivel de software como de hardware, para que su trabajo pueda solucionar de la manera más adecuada los problemas que responden a la realidad digital en la que vivimos.

En el marco anteriormente dibujado podemos fácilmente ver la coherencia y la necesidad que aprender y dominar la arquitectura del computador tiene para un ingeniero de software, ya que es este conocimiento nos brindara las herramientas necesarias para poder afrontar los constantes cambios en el área, gestión de datos así como optimizar la arquitectura escogida para desarrollar aplicaciones.

OBJETIVOS

Desarrollar un aplicativo para calcular si un número es primo o no, usando como punto de partida el uso de seudocódigo así como del diagrama de flujo para aplicarlo usando un lenguaje de programación que permita ejecutar la tarea.

OBJETIVOS ESPECIFICOS

- Determinar los pasos o instrucciones necesarias para poder deducir si un número es primo o compuesto.
- Tomando lo anterior como referencia diseñar un algoritmo que desempeñe tal función (seudocódigo)
- Implementación de dicho algoritmo en código.

DESARROLLO

Primera entrega:

1) Documento con el diseño del algoritmo, en pseudocódigo o un lenguaje de programación en alto nivel, que dé solución al problema: "Determinar si un número es primo o no". Adicionalmente, se debe especificar qué tipos de instrucciones pueden ser requeridas para la solución del problema, de acuerdo con el algoritmo diseñado.

R://

Algoritmo:

```
INICIO
Variables
        número, divisor Entero
        esPrimo: Lógico
Fin Variables
esPrimo = Verdadero
Escribir "Ingrese un número"
Leer número
divisor = 2
mientras (divisor <= (número/2))
        Si(número mod divisor == 0) entonces
                esPrimo = Falso
        Fin Si
        divisor = divisor + 1
Fin mientras
Si (esPrimo) entonces
        Escribir "El número es primo"
Si no
        Escribir "El número no es primo"
Fin Si
FIN
```

Tipos de instrucciones requeridas:

Para el algoritmo anterior se van a necesitar las siguientes instrucciones básicas:

Instrucciones de memoria: se va a necesitar almacenar y consultar varios registros en memoria para poder comparar los números y determinar si es un número primo.

Instrucciones matemáticas y lógicas: acá está la base del algoritmo. Si no fuera por las operaciones matemáticas y lógicas, no se podría determinar si un número es primo o no.

Instrucciones de control: el algoritmo cuenta con una serie de condiciones que lo hacen saltar de un lado a otro. Hay varios condicionales y cuenta con un loop/while que se va a estar ejecutando cíclicamente hasta terminar la condición específica.

Manejo de registros: el algoritmo implica estar cambiando y asignando nuevos valores a registros ya definidos, por lo que estará realizando operaciones entre los registros.

Segunda entrega

1) Listado de instrucciones detallando por cada una:

Para este listado de instrucciones nos apoyamos en el código que se escribió en MARS y que colocamos en el punto 4º de esta segunda entrega. Están listadas todas las instrucciones con sus respectivos registros.

Nombre	Mnemónico	Parámetros
Load immediatly (Carga de inmediato)	Li	En cuál registro, qué carga
Load at (Carga en)	La	En cuál registro, qué carga
Move (Mover)	Move	Registro destino, Registro origen
Brancha (Ir a la rama-Saltar a)	B	Label destino
Branch Lower than (Si es menor saltar a)	Blt	Comparador 1, Comparador 2, label destino
Branch equal to (Si es igual saltar a)	Beq	Comparador 1, Comparador 2, label destino
División	Div	Dividendo, divisor
Move division reminder to (Mover residuo de la división a)	Mfhi	Registro destino
Branch if equals 0 (Saltar a si igual a 0)	Beqz	Comparador, label destino
Add immediatly (Sumar de inmediato)	Addi	Registro de almacenamiento, Sumando 1, Sumando 2

Descripción y ejemplo de la instrucción:

Mnemónico	Descripción	Ejemplo

Li	Asigna al registro el valor especificado	li $8, 5
La	Ingresa el valor al registro especificado	la $5, 8
Move	Mueve el valor de un registro a otro registro	move $4, $8
B	Salta al label especificado	b nombreLable
Blt	Salta al label especificado si el comparador 1 es menor que el comparador 2	blt $8, $9, nombreLable
Beq	Salta al label especificado si el comparador 1 es igual que el comparador 2	beq $9, $8, nombreLable
Div	Divide el valor del primer valor dado por el segundo valor dado	div $8, $9
Mfhi	Asigna a un registro el sobrante de la división	mfhi $10
Beqz	Salta al label especificado si el valor entregado es igual a 0	beqz $10, nombreLable
Addi	Suma y asigna de inmediato el valor de la suma al registro	addi $9, $9, 1

2) Definición de la cantidad de registros que se usarán y el tamaño de cada registro.

En total son 8 registros que se utilizaran y que la mayoría irá cambiando de valor por cada ciclo de la ejecución del código

Registro	Tamaño	Descripción
$zero	0	Es una constante
$v0	32 bits	Registro donde se almacena la entrada del número ingresado
$a0	32 bits	Registro donde se almacenan los mensajes
$8 ($t0)	32 bits	Registro donde se pasa el número ingresado. No cambia
$9 ($t1)	32 bits	Registro que se usa como contador. Irá incrementando para hacer el ciclo y poder evaluar si los números son primos
$10 ($t2)	32 bits	Registro usado para almacenar el residuo de la división en cada ciclo

3) Descripción breve de los modos de direccionamiento que serán posibles:

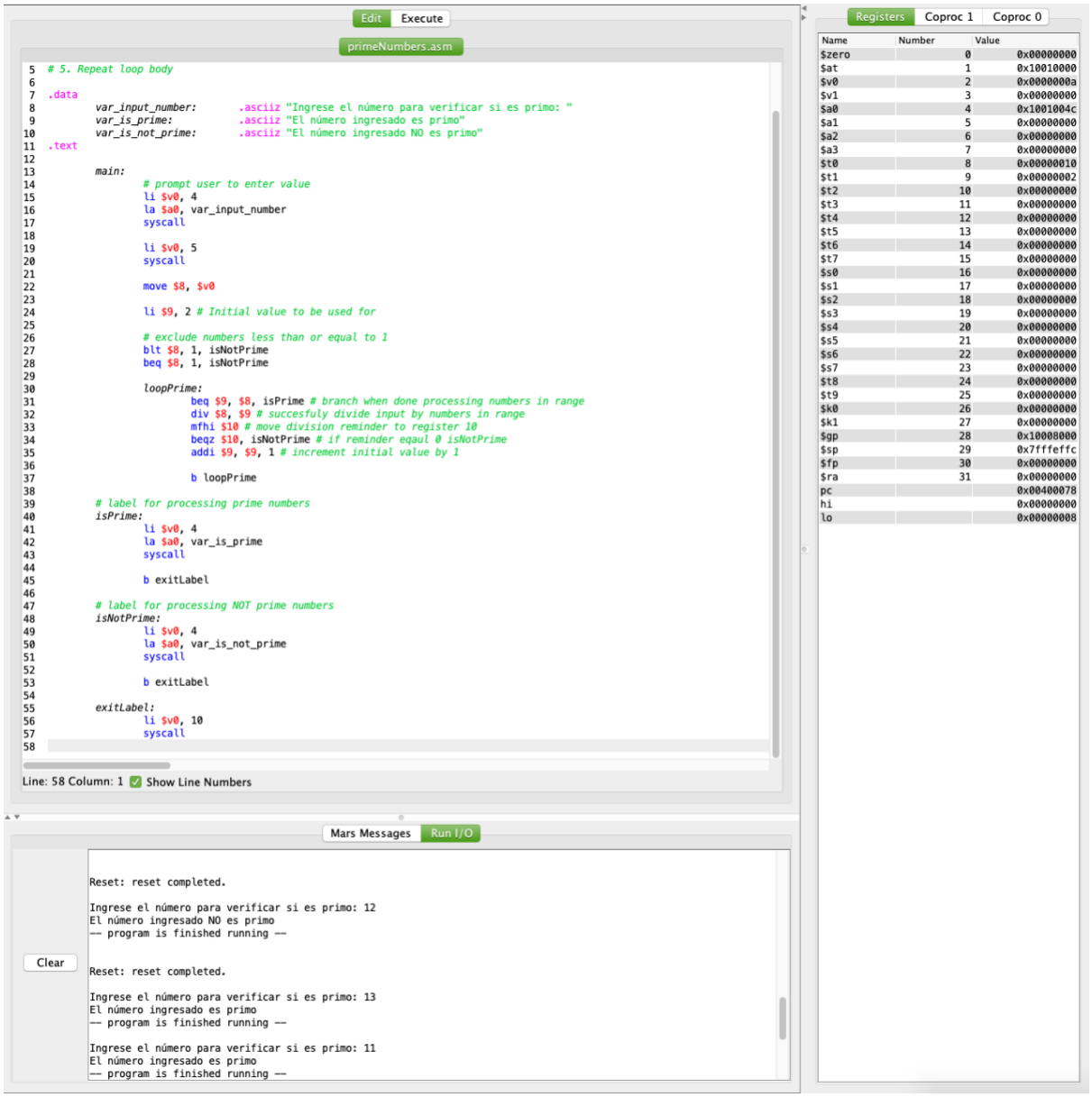
Se están agrupando las instrucciones utilizadas en MARS según el modo de direccionamiento que se vio en el módulo del escenario 3.

Modo de direccionamiento	Instrucción	Descripción
Directo a registro	div $8, $9, mfhi $10	Se incluye en la instrucción la dirección del registro, como destino o como fuente
Indirecto a registro	la $a0, move $8, $v0	Se incluye la dirección de un registro y un desplazamiento
Relativo a PC	beq $8, 1, lable; blt $8, 1, lable	El operando corresponde a una dirección de la memoria de programa y resulta de la suma entre el valor actual del contador de programa y una constante que viene en la instrucción
Inmediato	li, addi	Instrucciones en los que uno de los operandos es un valor constante que viene dentro de la instrucción.

4) Traducción del programa de alto nivel (entrega 1) al lenguaje ensamblador que han construido, usando su propio conjunto de instrucciones.

En esta sección pasamos el seudocódigo a lenguaje de ensamblador en MARS. Lo hicimos basados en este tutorial de Youtube: https://www.youtube.com/watch?v=DaD2kUl90nI

En este programa se pide en consola que se ingrese un número para posteriormente empieza el programa a



Codificación de las instrucciones:

Después de tener el código en lenguaje ensamblador procedemos a colocar la codificación de las instrucciones. Pensamos que el programa ensamblador nos muestra de una mejor manera cuáles son las instrucciones que se necesitan.

Instrucc.	Tipo	OpCode	Op.1	Op2.	Op3.	Resultado
li cargar	Aritmética Lógica	000	Registro destino	Valor	N/A	000 001 000 000
move	Aritmética Lógica	001	Reg. destino	Reg. origen	N/A	001 010 111 000

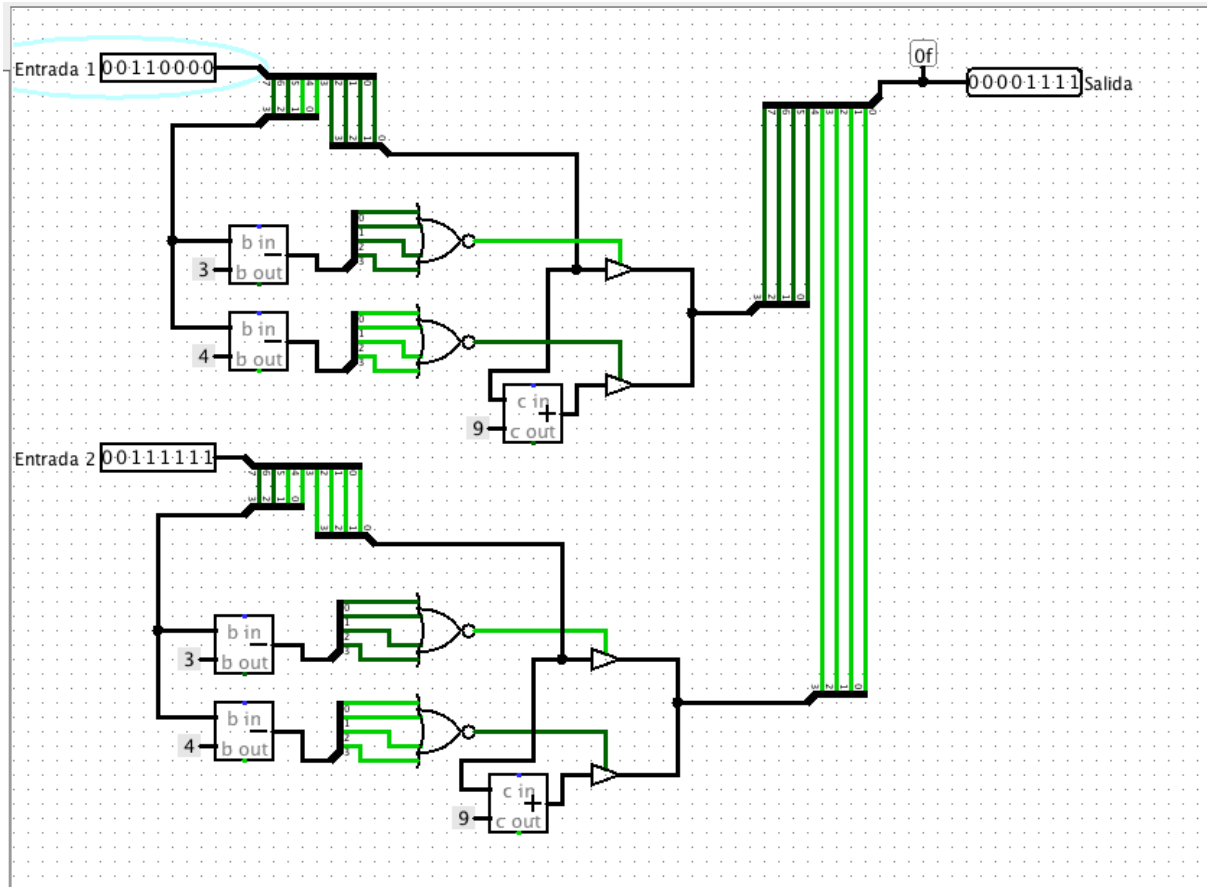
blt	Salto	010	Reg. base	Reg. comparar	Label	010 111 111 010
beq	Salto	011	Reg. base	Reg. comparar	Lable	011 110 110 001
div	Aritmética Lógica	100	Reg. Divide	Reg. Divider	N/A	100 100 110 000
mfhi	Aritmética Lógica	101	Reg. destino	N/A	N/A	101 010 000 000
beqz	Salto	110	$Zero	Reg. Comparar	Label	110 000 101 001
addi	Aritmética Lógica	111	Reg. destino	Reg1	Reg2	111 010 110 100

5) Listado en HEXA del binario que representa el programa en el punto anterior. Es tomar el resultado de los CODOPS en binario y pasarlo a HEXADECIMAL

#	Resultado		Hexa
0	000 001 000 000	000001000000	40
1	001 010 111 000	001010111000	2B8
2	010 111 111 010	010111111010	5FA
3	011 110 110 001	011110110001	7B1
4	100 100 110 000	100100110000	930
5	101 010 000 000	101010000000	A80
6	110 000 101 001	110000101001	C29
7	111 010 110 100	111010110100	EB4

6) ALU en Logisim, que pueda ejecutar las instrucciones requeridas por el programa: Este punto nos costó bastante pues no terminamos de entender muy bien cómo hacer la ALU para que nos sirviera para ejecutar las instrucciones requeridas. Buscamos varios tutoriales para construir una ALU:
https://www.youtube.com/watch?v=dYZ-Hwbcnq4&t=686s
https://www.youtube.com/watch?v=Ew7h_cOoKmY
https://www.youtube.com/watch?v=lvYCchzQTyE&t=482s
https://www.youtube.com/watch?v=oJtxKvqRTiQ&pp=qAMBugMGCgJlcxAB

Después de ver varios videos, creemos que la ALU que nos va a servir para nuestro propósito es la imagen que se muestra a continuación (así mismo se está adjuntando el archivo de Logisim):



Tercera entrega:

Link del video mostrando el proceso: https://youtu.be/NrJmt9tzq-A

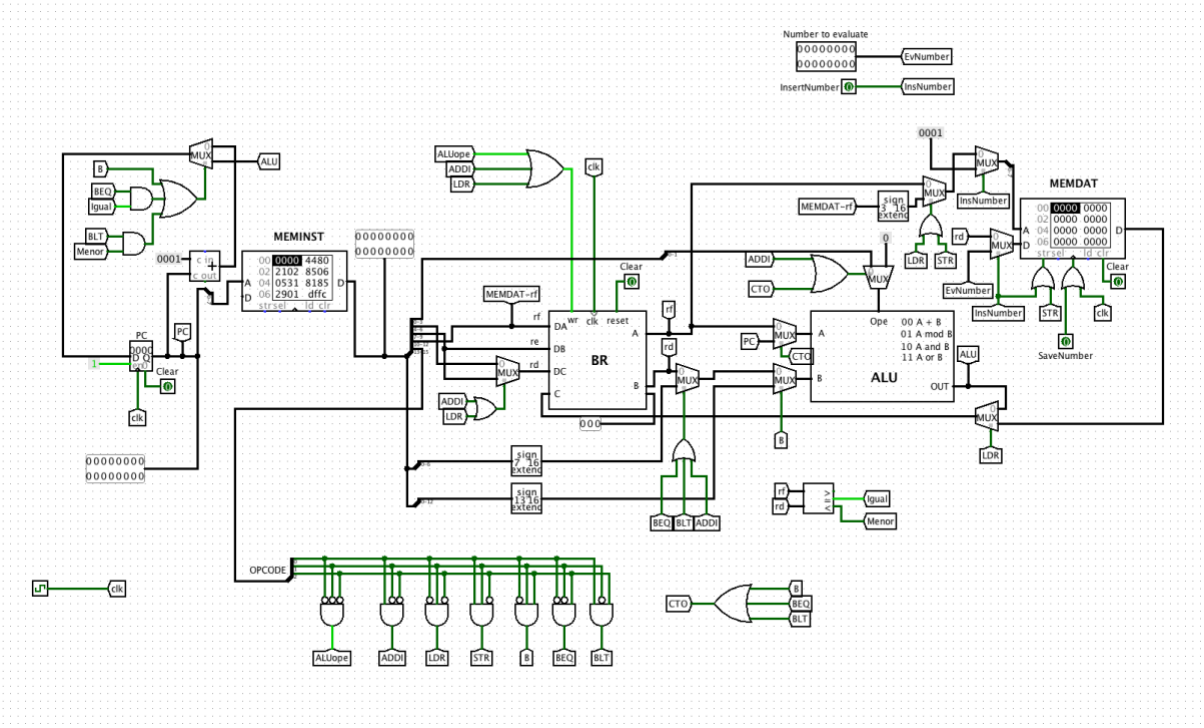
Implementación de un procesador simple de 16 bits. Utilizamos de referencia el link compartido en clase donde Corbera Francisco de la universidad de Málaga, explica paso a paso la implementación del procesador simple de 16 bit (http://www.ac.uma.es/~corbera/procesadorsimple/web/).

Solamente modificamos la ALU, pues la resta no la íbamos a necesitar, en lugar necesitábamos el módulo de la división, entonces hicimos esa modificación. También en lugar de tenar las memorias de datos e instrucciones en un circuito aparte las incluimos directas en el procesador para poder mirar el proceso que llevaba cada memoria.

No habíamos entendido muy bien la implementación de las instrucciones y todos los pasos previos. Por eso a modo de corrección de las entregas previas colocamos a continuación la programación y la implementación.

Vamos a explicar cómo está compuesto este procesador simple que implementamos y después pasaremos a explicar cómo hicimos el código para que corra en ese procesador.

Primeramente veremos cómo queda totalmente implementado el procesador en Logisim:

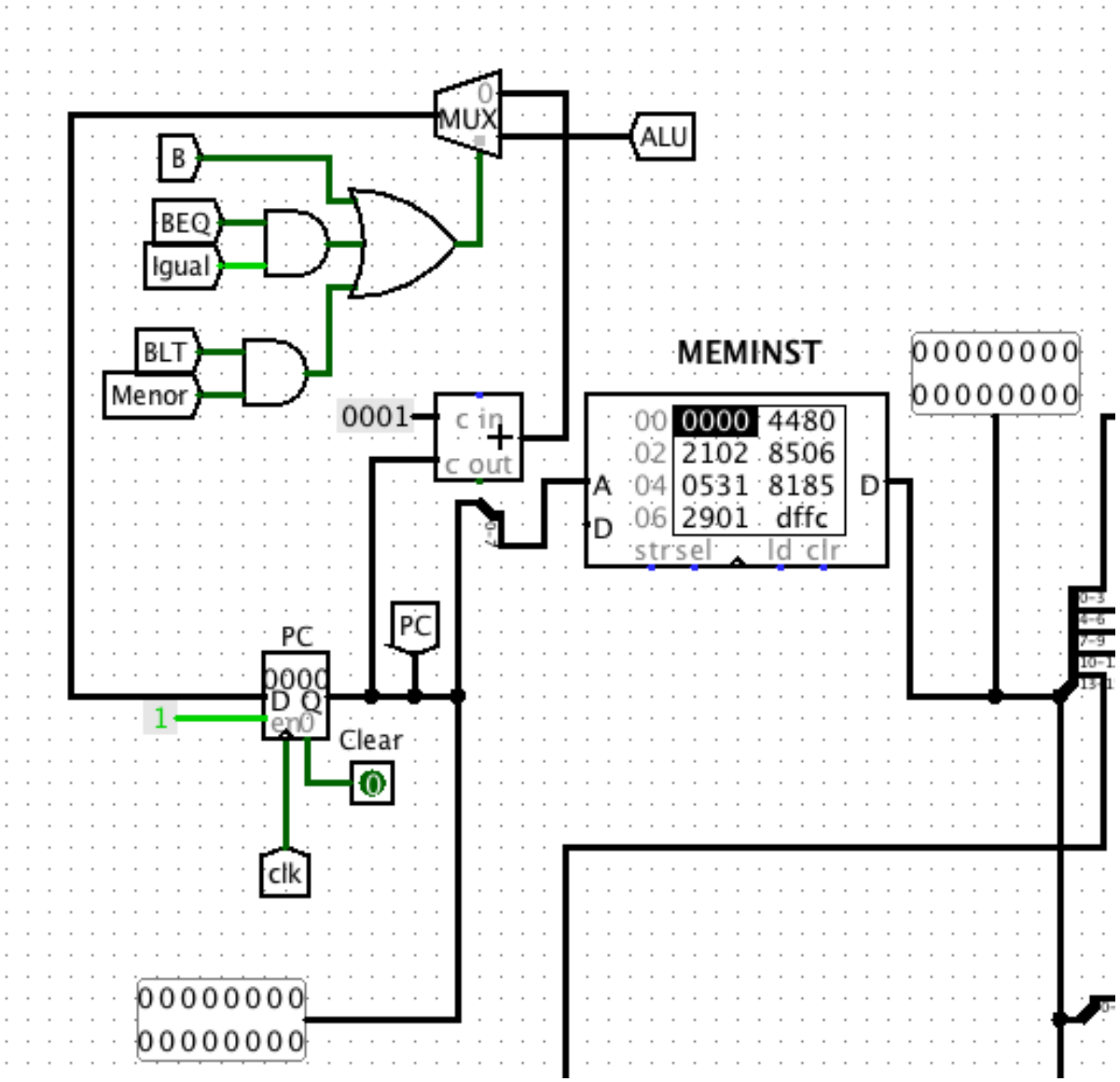


Esta compuesto por:

- MEMINST: Donde se almacenan los procesos a ejecutar por el procesador
- BR: Banco de registros. Donde se almacenan los registros que el procesador utilizará para ejecutar sus funciones.
- ALU: Unidad aritmético lógica, encargada de realizar los procesos aritméticos y lógicos del procesador. En nuestro caso suma y divide, pero solo devuelve el remanente.
- MEMDAT: Donde se almacenan los datos. Tanto el número que se va a evaluar como el resultado de la operación.

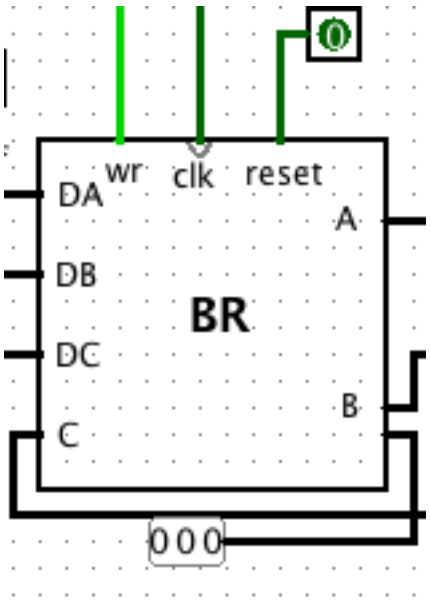
MEMINST:

La memoria de instrucciones. Contiene las instrucciones almacenadas y un contador que va aumentando cada vez que el reloj está ejecutándose.

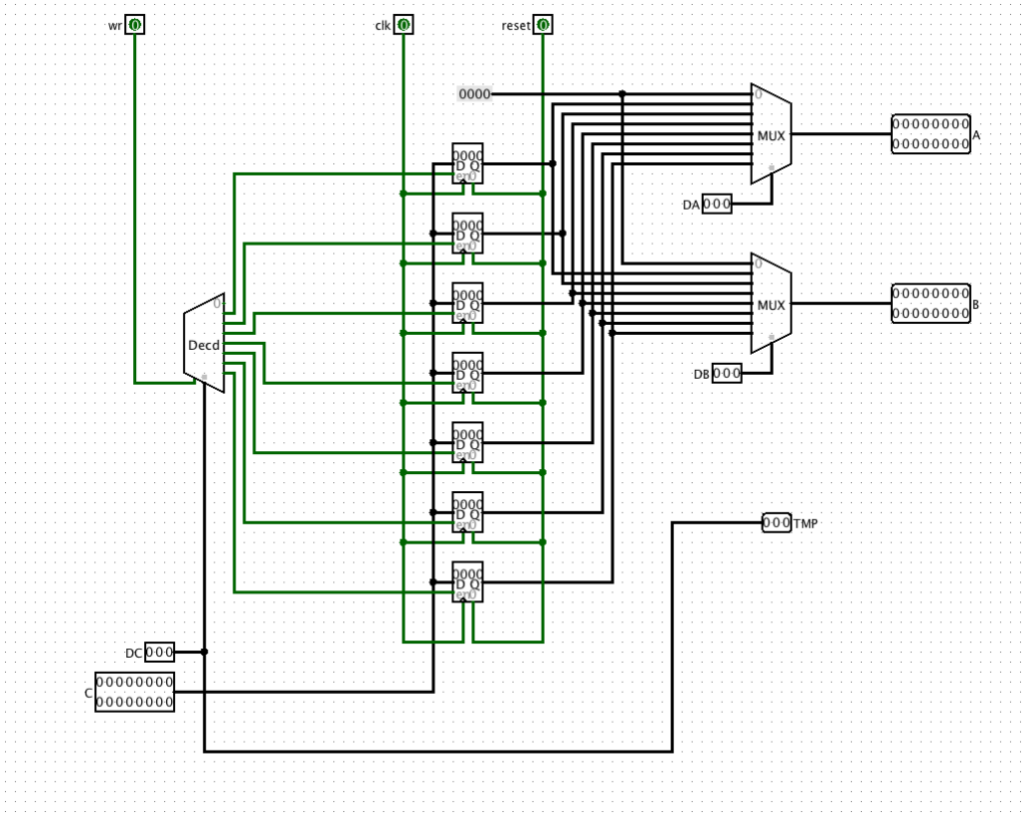


BR – Banco de Registros:

Es el lugar donde se almacenan los registros que el programa necesita utilizar. Este es el que se está trayendo a la implementación.



Acá se ven los 8 registros. El registro 0, siempre es 0 y por eso permanece una constante, pero los otros 7 son los registros de 16 bits donde se almacena la información que se necesita:

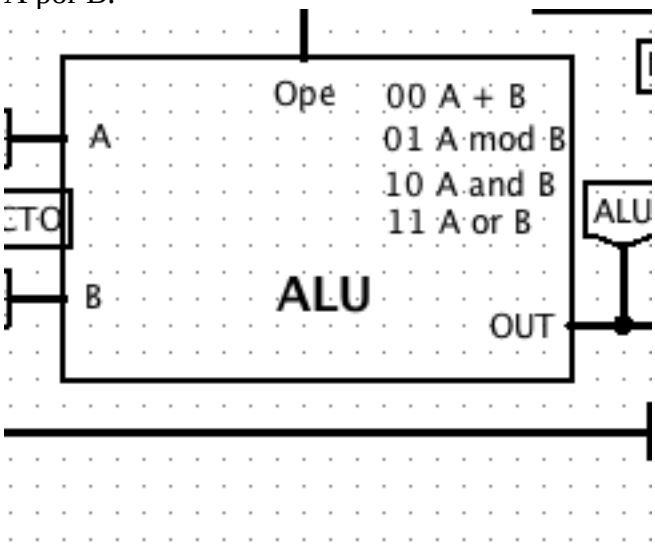


ALU-Aritmethic Logic Unit:

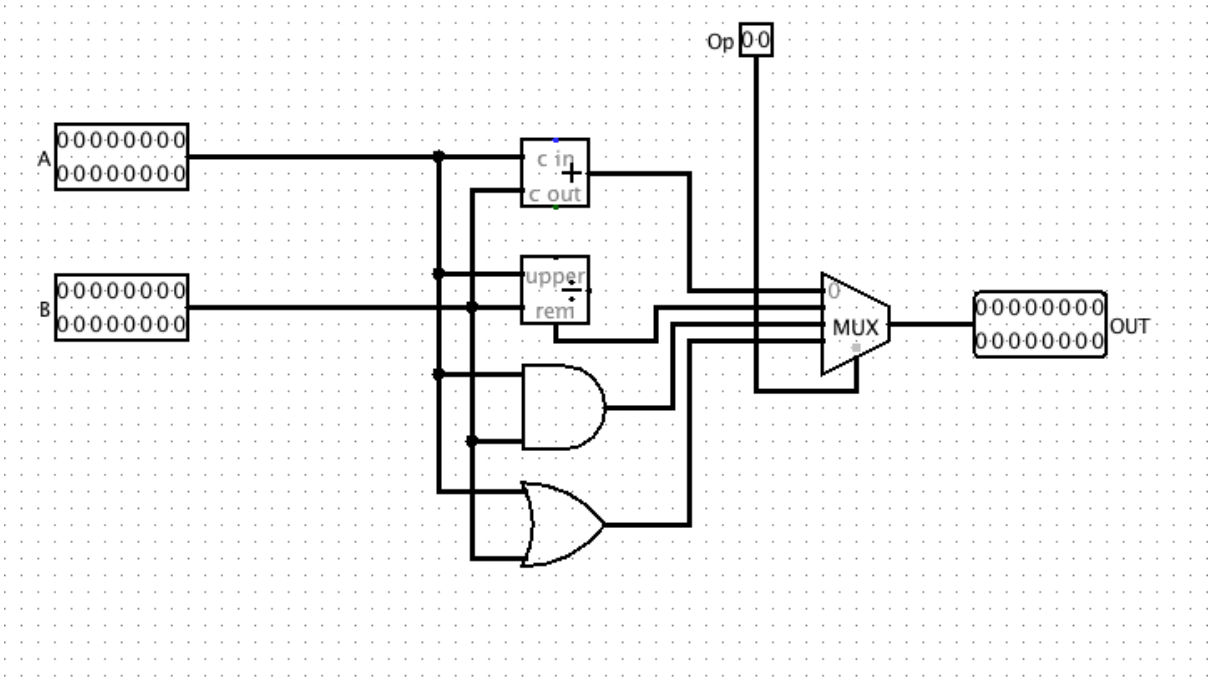
Unidad donde se realizan las operaciones lógicas o matemáticas. Esta unidad venía implementada con:

00 => Suma
01 => Resta
02 => AND
03 => OR

Nosotros modificamos la ALU para que el 01 no fuera resta sino el módulo de la división entre A por B.



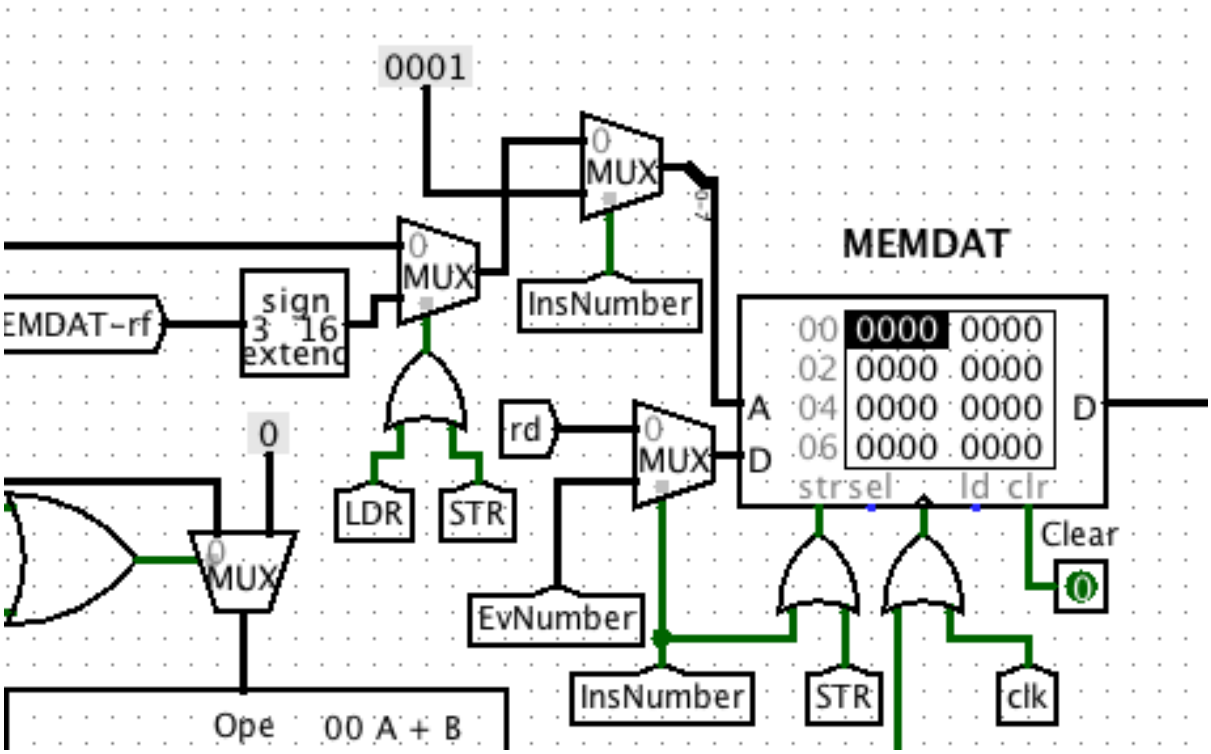
Esta es la ALU por dentro. Ya modificada a nuestro proyecto:



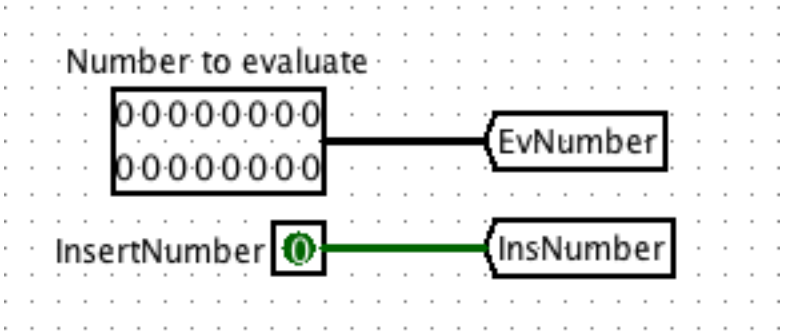
MEMDAT:

Por último está la memoria de datos. Para nuestro proyecto solo estamos usando los tres primeros registros.

00 => Se almacena el resultado si al terminar el proceso el número es primo.
01 => Se almacena el número que se va a evaluar.
02 => Es el estado del programa. Mientras esté en 0 indica que el programa está activo. Cuando termina el proceso se almacena el 1 en ese registro indicando que terminó.



Por comodidad se implementaron estos elementos para almacenar en el registro 01 el valor que se quiere evaluar:



PROGRAMACIÓN – Programa:

Para ejecutar en este procesador hicimos el siguiente programa para determinar si el número a evaluar es un número primo; es similar al presentado en las otras entregas, pero ahora está hecho específicamente adaptado para que este procesador ejecute las instrucciones:

#	Instrucción	Descripción
0	En blanco	
1	ldr r1, r1	// Almacena en BR r1, el valor de Memdata r1
2	addi r0, r2, 2	// Se inicia el contador en 2
3	beq r1, r2, jump(+7)	// Compara si son iguales r1 y r2-Si son iguales salta 7
4	mod, r1, r2, r3	// Almacena en r3 el módulo (remanente) de la división de r1 y r2
5	beq r0, r3, jump(+5)	// Compara si r3 es igual a 0. Si lo es salta y termina el programa. No es primo
6	addi r2, 1	// Se añade 1 al contador
7	b loop(-4)	// Si se llega a este punto es que no ha terminado ciclo y el remanente no ha dado 0
8	En blanco	// Espacio-solo por separar instrcciones
9	str r0, r1	// Se almacena en MEMDAT el valor del número buscado. Si llega a este punto el número es primo
10	addi r0, r4, 1	// Añade al registro 4 el valor de 1
11	str r2, r4	// Almacena el valor de 1 que estaba en BR r4 y lo almacena en MEMDATA r2
12	b r0 : end	// Se envía a sí mismo, entonces ya con esto se indica que el proceso terminó

Codificación:

En la siguiente tabla vamos a mostrar cómo quedaría la codificación definitiva. Por un lado están las instrucciones en lenguaje ensamblador, después su correspondencia en binario y por último su correspondencia en número hexadecimal:

#	Instrucción	Bin	Hex
0	En blanco	0000000000000000	0x0000
1	ldr r1, r1	0100010010000000	0x4480
2	addi r0, r2, 2	0010000100000010	0x2102
3	beq r1, r2, jump(+7)	1000010100000110	0x8506
4	mod, r1, r2, r3	0000010100110001	0x0531
5	beq r0, r3, jump(+5)	1000000110000101	0x8185
6	addi r2, 1	0010100100000001	0x2901
7	b loop(-4)	1101111111111100	0xDFFC
8	En blanco	0000000000000000	0x0000

9	str r0, r1	0110000010000000	0x6080
10	addi r0, r4, 1	0010001000000001	0x2201
11	str r2, r4	0110101000000000	0x6A00
12	b r0 : end	1100000000000000	0xC000

Las instrucciones 3 y 7, les pusimos un color para indicar que dentro de esas instrucciones se va a estar realizando el loop (ciclo) para determinar si un número es primo. Hasta que no se cumpla una de las condiciones, no se sale de ese ciclo.

Explicación de codificación:
En la tabla anterior ya vimos el programa codificado, pero de ¿dónde salieron esos códigos? A continuación explicamos el significado de esos códigos.

ADDI:

ADDI rd, rf,cte

BR[rd] <= BR[rf] + ExtSigno(cte,16)

15		13	12		10	9		7	6						0
OPCODE(001)			rf			rd			cte = constante						
0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0
			r0			r2			2						

Codificación: 0010000010000001 => 0x2081

LOAD – LDR:
Ingresa de la memoria MEMDAT al banco de registro el valor especificado.

LODE = LDR rd, [rf]

BR[rd] <= MEMDAT[BR[rf]]

15		13	12		10	9		7	6						0
OPCODE(010)			rf			rd			000000						
0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0
			BR - r1			MMEDAT r1									

Codificación: 0100010010000000 => 0x4480

BRANCH IF EQUAL-BEQ:
Si son iguales dos registros salta a la dirección dada. La dirección se da colocando el número que se le debe sumar a la ubicación actual para dirigirse a ese punto donde está ubicada la instrucción.

BRANCH equal = BEQ rf1, rf2,
dir

Si(rd == rf) -> PC <- PC + ExtSigno(dir, 16)

15 13	12 10	9 7	6 0
OPCODE(100)	rf	rd	dir
1 0 0	0 0 1	0 0 1	0 0 0 0 1 1 1
	r1	r2	7

Codificación: 1000010010000111 => 0x8487

ALUope:
Realiza una operación aritmética con los registros dados.

ALUope rd, rf, re

BR[rd] <= BR[rf]ALUopeBR[re]

ALUope:

(0000) ADD => suma (+)

(0001) MOD => modulo (%)

(0010) AND => and lógico bit a bit (&)

(0011) OR => or lógico bit a bit (|)

15 13	12 10	9 7	6 4	3 0
OPCODE(000)	rf	re	rd	ALUope
0 0 0	0 0 1	0 1 0	0 1 1	0 0 0 1
	r1	r2	r3	MOD(001)

Codificación: 0000010100110001 => 0x0531

BRANCH-b:
Salta a la dirección especificada. Es incondicional, no evalúa nada, solo brinca a la dirección especificada:

Branch = B dir

PC <= PC + ExtSingo(dir, 16)

15		13	12												0
OPCODE(110)			dir												
1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0
			-4												

Codificación: 1101111111111100 => 0xDFFC

STORE – str:
Almacena el valor del registro especificado BR rd en MEMDATA rf.

STORE = STR rd, [rf]

MEMDAT[BR[rf]] <= BR[rd]

15		13	12		10	9		7	6						0
OPCODE(011)			rf			rd			000000						
0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0
			MEMDAT r0			BR r1									

Codificación: 0110000010000000 => 0x6080

Con esto ya finalizamos la explicación de la codificación utilizada.

DATOS:
Tenemos dos tipos de datos. Las instrucciones que se tienen que cargar en MEMINST y los datos que se definen en MEMDAT.

En el caso de MEMINST las instrucciones están en un archivo aparte que hay que cargar en la memoria, para que el procesador pueda ejecutar los procesos.

En el caso de MEMDAT se especificaron los siguientes datos:

Posición	Valor	Descripción
000	0	Estado inicial: 0 Estado final: El número buscado si es primo
001	Número a buscar	En este registro se almacena el número que se va a buscar
010	0	Estado inicial: 0 Estado final: 1 (cuando termina el ciclo)

Con esto ya queda completo el conjunto de instrucciones y su respectiva implementación.

CONCLUSIONES

Con la construcción de esta entrega se realizó una investigación de los conceptos del problema y de la teoría en este módulo, en cada punto se desarrolló un algoritmo.

Al plantear un algoritmo como el propuesto en el problema pudimos abarcar y satisfacer la solución de forma tal que se estructuró un pseudocodigo para la solución del problema, sin embargo esta solución no da por sentado que sea la única forma de solución posible, por eso recibimos retroalimentación de las construcciones dadas para dar un refinamiento a nuestros métodos, siendo esto muy útil para el desarrollo de la práctica y el estado del arte