Universidad de la República - Facultad de Ingeniería

Curso 2022 : Arquitectura de Computadoras

Informe Obligatorio

Estudiante			
Cédula de identidad	Nombre	Correo Electrónico	Carrera
5.031.022-5	Guido Dinello	guido.dinello@fing.edu.uy	Ingeniería en Computación

Índice

I. Resumen

El presente trabajo obligatorio persigue mediante un enfoque práctico el objetivo de ser un medio introductorio hacia la arquitectura 8086. Para ello se plantea un problema, que será modelado en C y posteriormente implementado en lenguaje ensamblador buscando emular una "compilación manual" tanto de la interacción E/S, como de las estructuras de control y estructuras de datos utilizadas.

Para fomentar la comprensión de los elementos y mecanismos de hardware de la arquitectura 8086 se utiliza un ambiente simulado como plataforma de programación. El simulador (ArquiSim) [7] permite tanto el desarrollo de software en ensamblador, como el ensamblado y la simulación de la ejecución sobre una arquitectura 8086.

II. Descripción del problema

La problemática planteada es la de construir un programa sencillo que contemple las operaciones básicas de una calculadora, además esta calculadora interpretará los datos introducidos siguiendo la notación polaca inversa o *RPN* [6] .

III. Consideraciones de E/S

El programa hará uso de 3 puertos de Entrada/Salida de 16 bits. Cada uno de los cuales se inicializará con un valor por defecto definido mediante las constantes ENTRADA, PUERTO_SALIDA_DEFECTO y PUERTO_LOG_DEFECTO.

- Un puerto de solo lectura que estará reservado para la Entrada de los comandos y operandos los cuales seguirán el formato:

  Comando [*Parámetro*]

- Un puerto, reservado para la Salida donde se mostrarán los resultados de algunas de las operaciones.

- Un puerto que oficiará de bitácora, y donde se dejará registro de cada una de las operaciones realizadas junto con un código de error, siguiendo el formato:

0 Comando [*Parámetro*] *Código_Error*

Tanto los comandos como los parámetros son de 16 bits.

Código	Semántica
16	Operación realizada con éxito
8	Fallo por falta de operandos en la pila
4	Fallo por desbordamiento de la pila (ya hay 31 operandos)
2	Comando inválido (no reconocido)

Cuadro 1: Operaciones

IV. Operaciones Soportadas

Comando	Parámetro	Código	Descripción
Num	Número	1	Agrega Número a la pila
Port	Puerto	2	Setea el puerto de salida
Log	Puerto	3	Setea el puerto de la bitácora
Top		4	Muestra el tope de la pila en el puerto de salida (no retira de la pila)
Dump		5	Realiza un volcado de la pila en el puerto de salida (no retira de la pila)
DUP		6	Duplica el tope de la pila
SWAP		7	Intercambia tope de la pila con el elemento debajo
Neg		8	Calcula el opuesto
Fact		9	Calcula el factorial
Sum		10	Calcula la suma de todos los elementos de la pila (borrando la pila) y deja el resultado en el tope de la pila. Si no hay elementos deja 0 en el tope.
+,-,*,/, %,&,—,«,»		11 al 19	Realiza la operación binaria correspondiente
Clear		254	Borra todo el contenido de la pila
Halt		255	Detiene el procesamiento

Cuadro 2: Operaciones

Cuando se desea realizar una operación binaria pero solamente se encuentra 1 elemento en la pila (el operando derecho), además de dejar constancia en la bitácora se debe eliminar el elemento de la pila (quedará vacía).
El comando Dump realiza un volcado de la pila en el puerto de salida comenzando desde el tope de la misma y además esta operación no modifica la pila.

V. Consideraciones Generales

Se manejará una pila interna de trabajo de hasta 31 operandos enteros de 16 bits representados en complemento a 2. Todas las operaciones aritméticas (incluyendo la multiplicación) son de 16 bits, es decir, dejan como resultado en el tope de la pila los 16 bits menos significativos de la operación.

VI. Descripción de la solución incluyendo detalle de las estructuras de datos y constantes utilizadas

VI.I. Constantes

Con el fin de hacer el código parametrizable y evitar el *hardcodeo* de valores arbitrarios se decidió declararse una constante por código de error. Esto permite centralizar dicha información facilitando una posible posterior modificación de los **códigos numéricos asociados a cada error**, además aporta legibilidad dado que los nombre elegidos son descriptivos del tipo de error que declaran.
Por otro lado, tenemos las 3 constantes previamente mencionadas que servirán para inicializar los valores de los puertos.
Por último se declaran dos constantes, **STACK_SIZE** que refiere a la cantidad maxima de operandos que manejara nuestra pila, y **DOBLE_STACK_SIZE** que simplemente es el doble de la constante anterior y en particular servirá para chequear si la pila esta llena. Esta peculiaridad proviene del hecho de que la memoria se direcciona de a un byte pero nosotros estamos manejando operandos de dos bytes, luego con el fin de hacer cómodo el direccionamiento de nuestros operandos se incrementa el tope de a dos unidades por cada apilado y análogamente el mismo se decrementa de a dos unidades por desapilado, de aquí es fácil observar que cuando nuestra pila este llena es porque apilamos 31 operandos y por ende, nuestro tope tendrá el valor 62.

VI.II. Variables

Respecto a las variables declaradas, estas corresponden a los dos puertos modificables. La variable referida al puerto de salida bajo el identificador **puertoSalida** y la variable referida a la bitácora bajo el identificador de **puertoLog**.

VI.III. Estructura de datos

Finalmente, siguiendo la implementación usual de una calculadora RPN (y como fué menciondo previamente) se optó por una **estructura de datos LIFO** (Last In First Out), es decir, una pila o stack.

Para este fin, se reservaron STACK_SIZE cantidad de palabras accesibles dentro de la región del segmento de memoria DS asignado por el simulador mediante el identificador stack. Además se reserva una palabra para llevar el indicador de la cantidad de elementos que contiene actualmente nuestra pila, el cual corresponde a la variable tope.

VI.IV. Directivas utilizadas

La directiva utilizada para la declaración de constantes fue:

identificador EQU *valor*

La directiva utilizada para reservar palabras (2 bytes) fue la de:

identificador dw *valor_inicial*

, en el caso particular de stack también se hizo uso de la directiva DUP con el objetivo de repetir la instruccion dw STACK_SIZE cantidad de veces para reservar todas las palabras necesarias, además se lo acompaño del carácter *?* para indicar que no hay necesidad de inicializar los valores de dichas palabras reservadas con un valor específico.

identificador dw DUP(cant) *valor_a_repetir*

VI.V. Procedimientos y etiquetas

En general se optó por manejar la mayor parte del flujo del programa mediante etiquetas. En particular una para cada comando además de alguna internas de estos mismos y de la etiqueta main.

Por otro lado, con el fin de proporcionar una interfaz básica para el manejo de la pila se implementaron dos procedimientos, **pushStack** y **popStack**, los cuales hacen lo esperado. Es decir, pushStack recibe el operando a insertar en el stack en el registro AX, lo inserta en la posición adecuada e incrementa el tope, por su parte popStack hace lo inverso, decrementa el tope y retorna el valor del elemento que hay en el tope de la pila utilizando el registro BX.

El tercer y úlitmo procedimiento utilizado es el de **factorial**, que implementa la rutina recursiva para realizar dicho cálculo. Sigue la definición más directa de la operación

matemática y obtiene el resultado esperado multiplicando el valor del parámetro recibido (recibido en el registro AX) por el mismo menos uno reiteradamente. Luego de las llamadas a sí misma con dicho parámetro decrementado en una unidad, se alcanza el denominado paso base que es cuando este vale 0. Finalmemente guarda el resultado de las sucesivas multiplicaciones en el registro BX y retorna.

$$factorial(n) = \begin{cases} n * factorial(n-1), if n > 0 \\ 1, if n = 0 \end{cases} \quad (1)$$

VI.VI. Cuerpo y Flujo del Programa

Dado que el problema fue modelado como un switch case, parece bastante natural seguir un método de *branching* como las tablas de salto, es por esto, que el cuerpo principal del código tiene la estructura :

```
main:
; preprocesado e impresion de codigos
in ax, ENTRADA
...
cmp ax, CODIGO_NUMERICO_DE_COMANDO
je ETIQUETA_DE_COMANDO
cmp ax, CODIGO_NUMERICO_DE_COMANDO_2
je ETIQUETA_DE_COMANDO_2
...
jmp main
```

De esta forma se consigue el control de flujo deseado donde se leen los comandos de la entrada y dependiendo del código de operación leido, se salta hacia la etiqueta correspondiente.

A su vez, dentro de cada etiqueta la estructura es bastante similar. Algún chequeo sobre la cantidad de operandos en la pila, y el procesamiento del comando con sus respectivos operandos en caso de poder realizarse la operación o la impresión del código de error correspondiente.

Para el comando **NUM** se chequea si la pila está llena, de ser así se envía el código de error desborde de pila y se retorna al main, de lo contrario se realiza el push del parámetro leído, se imprime el código de éxito y se retorna al main.
Los comandos **PORT** y **LOG** son similares, pero aquí no es de interés si la pila está llena o no, simplemente modificamos el valor de la variable del puerto correspondiente y volvemos al main.
Para el comando **TOP** es necesario verificar que haya al menos un operando en la pila. Si lo hay, disminuimos en 2 unidades al tope -el tope lleva la cantidad de elementos, por ende, siempre está por encima del último valor de la pila-, accedemos a la última posición del stack, esto es, en el segmento de datos DS, en el desplazamiento asociado a la etiqueta stack y con un desplazamiento extra que nos indica que posición del stack queremos acceder (el valor del tope) envíamos dicho valor al puerto de salida y envíamos el código de éxito a la bitácora. De lo contrario se envía el error de falta de operandos. En cualquiera de los casos luego se retorna a la etiqueta main.
Para el **DUMP** se utiliza una estrategia similar copiando el valor del tope en un registro y accediendo con la estrategia de direccionamiento mencionada previamente hasta que el valor auxiliar del tope que teníamos en el registro llegue a 0.
En el comando **DUP**, se *popea* el valor del tope del stack y luego se *pushea* el mismo dos veces.
Para lograr el **SWAP**, se *popean* los dos últimos valores y se los apila en el orden inverso.
Luego tenemos las **operaciones binarias**, donde debemos asegurarnos de que hayan al menos dos operandos en nuestra pila (de haber 1 solo este es eliminado), y luego se *popean* los dos operandos, primero el derecho, luego el izquierdo y se realiza la operacion binaria respectiva para luego *pushear* su resultado.
Las operaciones que vale la pena entrar en detalle son **DIV**, **MOD**, **LSHIFT** y

RSHIFT.

En las primeras dos, se sigue la estructura mencionada previamente pero además se realiza el chequeo extra de si el operando izquierdo es negativo, de serlo, se carga -1 en el registro dx y de no serlo se carga 0. Esto es debido a que la operación idiv utilizada realiza las operaciones:

$$\text{ax} = \text{dx::ax} \ / \ \text{operando}$$
$$\text{dx} = \text{dx::ax} \ \% \ \text{operando}$$

, pero como estamos trabajando con operandos de 16 bits estos caben en ax, por lo que debemos completar en dx el bit del signo correspondiente para que la operación pueda considerar el signo de forma correcta.

En las últimas dos, los chequeos extras que se realizan es si el operando derecho es mayor o igual a 16. En estos casos, el resultado ya es conocido.

Si realizamos un shift a la izquierda de 16 posiciones o más a un número de 16 bits obtendremos 0x0000 y si realizamos un shift a la derecha de tantas posiciones obtendremos todos los bits iguales al bit del signo del número original, esto es, 0xFFFF si el número era negativo o 0x0000 si este era positivo.

Por último, el comando **CLEAR** realiza un borrado lógico de la pila llevando el valor del tope a 0 y el comando **HALT** deja al programa en un bucle para evitar que siga ejecutando instrucciones basura.

[1]

VII. Experimentación y problemas encontrados

- Un detalle a considerar fue la peculiaridad ya mencionada de tener que incrementar el tope de a dos valores debido al direccionamiento de a byte y el uso de operandos de 2 bytes.
- Contemplado de casos borde en las operaciones de DIV y MOD teniendo que cargar manualmente el registro dx. Contemplado de casos borde en las operaciones LSHIFT y RSHIFT para shifteos de 16 o mas posiciones.

[1] Para esta sección se consultaron los siguientes recursos: [8],[5],[2][6][4][9][1][3]

- Operaciones con pocas variantes de registros admitidas -como el IN y OUT- lo que significó tener que gastar instrucciones extra en hacer pasajes de valores entre registros por motivos meramente sintácticos.

- Una pequeña incomodidad fue la falta de flexibilidad del simulador para correr una batería de test de casos de prueba. Puesto que la única forma es cargarlos manualmente y verificarlos de la misma forma sin ninguna herramienta/método automatizable.

VIII. Conclusiones

Si bien la compilación es bastante directa de C a Assembler, como es de esperarse en este último hay que realizar ciertos trabajos extra que normalmente están ocultos a nuestra vista cuando programamos en lenguajes de mas alto nivel. Esto termina produciendo una **cantidad de líneas mayor**, y también requiere una inversión de **tiempo de programación también superior**.

Por otro lado, es interesante realizar un proyecto práctico en este tipo de lenguaje y arquitectura para tener en consideración aspectos que solemos no tener en consideración como el direccionamiento de memoria y sus diferentes métodos, asi como **implementación a bajo nivel de las estructuras de control típicas** del paradigma de programación imperativa, como el while, switch, if, etc.

IX. Mejoras a futuro

- Dado que los puertos están especificados como de 16 bits, sería más correcto remplazar la instrucción ***in*** *ax, ENTRADA*, por el par de instrucciones ***mov*** *registro, ENTRADA* ***in*** *ax, registro*. Puesto que en el Manual de Usuario de ArquiSim se especifica que la operación IN recibe como segundo parámetro un registro de 16 bits o un inmediato de 8 bits, por lo que con la versión actual daría problemas si definimos al puerto de entrada como una constante de más de 8 bits.
- Es sencillo darse cuenta que las operaciones aritméticas siguen todas una misma estructura. Es por esto que podría realizarse una **factorización** de dichas **instrucciones repetidas** y así disminuir la cantidad de líneas utilizadas. Además, esto facilitaría una posterior modificación del código gracias a una mayor centralización. Esto podría lograrse, por ejemplo, creando etiquetas que se encarguen del envío de los códigos de errores, una acción que realizamos prácticamente para todos los comandos.
- Se podría volver a la calculadora más **resiliente** realizando los **chequeos adecuados de casos borde** que actualmente no se contemplan. Ejemplo

de esto son los *shifteos* de cantidades de bits negativas, entre otros.

- Si bien **preservar el contexto de los registros** no fue necesario para el correcto funcionamiento del programa, sería una buena idea preservar los registros al entrar a cada etiqueta/procedimiento y luego restablecerlos antes de salir por cuestión de prolijidad y buenas prácticas. Más aún, quizá sería mejor que los procedimientos recibieran los parámetros necesarios por el stack de *hardware* y así prescindir de la utilización de algunos registros para dicho fin. Estos cambios, aportarían más flexibilidad y adaptibilidad al programa facilitando la incorporación de nuevas funcionalidades o la incrustación del programa en un entorno no dedicado disminuyendo las probabilidades de falla.
- Para hacer un **buen uso de memoria** sería más correcto reservar sólo un byte para la variable tope puesto que el máximo valor que alcanzará este es el doble de la cantidad máxima de operandos y con un byte es suficiente para representar del 0 al 62.
- Se podría argumentar que algunos **jumps** no son los más **correctos semánticamente**, por ejemplo en los comandos DUMP y SUM, en vez de comparar con 0 en cada iteración se podría usar jz dado que el sub modifica la flag zero, pero también cabe mencionar que en dicha situación habría que considerar especialmente el caso donde el stack tuviera 0 elementos, puesto que la bandera zf no estaría prendida dado que aún no se ha llamado a la operación sub.

X. Casos de Prueba

Entrada:

1,1, 1,2, 1,3, 1,4, 1,5, 1,1, 1,9, 1,8, 1,-1400, 1,10, 1,11, 1,12, 1,13, 11, 4, 12, 4, 13, 4, 14, 4, 15, 4, 16, 4, 17, 4, 18, 4, 7, 4, 19, 4, 10, 4, 8, 4, 6, 5, 254, 255

Salida:

Puerto 1: 25, -14, -140, 10, 8, 8, 9, 2560, 4, 160, 166, -166, -166, -166

Bitácora:

Puerto 2: 0, 1, 1, 16, 0, 1, 2, 16, 0, 1, 3, 16, 0, 1, 4, 16, 0, 1, 5, 16, 0, 1, 1, 16, 0, 1, 9, 16, 0, 1, 8, 16, 0, 1, -1400, 16, 0, 1, 10, 16, 0, 1, 11, 16, 0, 1, 12, 16, 0, 1, 13, 16, 0, 11, 16, 0, 4, 16, 0, 12, 16, 0, 4, 16, 0, 13, 16, 0, 4, 16, 0, 14, 16, 0, 4, 16, 0, 15, 16, 0, 4, 16, 0, 16, 16, 0, 4, 16, 0, 17, 16, 0, 4, 16, 0, 18, 16, 0, 4, 16, 0, 7, 16, 0, 4, 16, 0, 19, 16, 0, 4, 16, 0, 10, 16, 0, 4, 16, 0, 8, 16, 0, 4, 16, 0, 6, 16, 0, 5, 16, 0, 254, 16, 0, 255, 16

Entrada:

11, 6, 1,1234, 7, 1,4321, 5, 12, 5, 9, 5, 1,-5, 8, 16, 1,1, 1,2, 1,3, 1,4, 1,5, 1,6, 1,7, 1,8, 1,9, 1,10, 1,11, 1,12, 1,13, 1,14, 1,15, 1,16, 1,17, 1,18, 1,19, 1,20, 1,21, 1,22, 1,23, 1,24, 1,24, 1,26, 1,27, 1,28, 1,29, 1,30, 1,31, 1,32, 1,33, 5, 255

Salida:

Puerto 1: 4321, 31, 30, 29, 28, 27, 26, 24, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

Bitácora:

Puerto 2: 0, 11, 8, 0, 6, 8, 0, 1, 1234, 16, 0, 7, 8, 0, 1, 4321, 16, 0, 5, 16, 0, 12, 8, 0, 5, 16, 0, 9, 8, 0, 5, 16, 0, 1, -5, 16, 0, 8, 16, 0, 16, 8, 0, 1, 1, 16, 0, 1, 2, 16, 0, 1, 3, 16, 0, 1, 4, 16, 0, 1, 5, 16, 0, 1, 6, 16, 0, 1, 7, 16, 0, 1, 8, 16, 0, 1, 9, 16, 0, 1, 10, 16, 0, 1, 11, 16, 0, 1, 12, 16, 0, 1, 13, 16, 0, 1, 14, 16, 0, 1, 15, 16, 0, 1, 16, 16, 0, 1, 17, 16, 0, 1, 18, 16, 0, 1, 19, 16, 0, 1, 20, 16, 0, 1, 21, 16, 0, 1, 22, 16, 0, 1, 23, 16, 0, 1, 24, 16, 0, 1, 24, 16, 0, 1, 26, 16, 0, 1, 27, 16, 0, 1, 28, 16, 0, 1, 29, 16, 0, 1, 30, 16, 0, 1, 31, 16, 0, 1, 32, 4, 0, 1, 33, 4, 0, 5, 16, 0, 255, 16

Entrada:

1,1, 1,2, 1,3, 9, 5, 2,3, 9, 5, 3,4, 7, 9, 5, 999, -999, 2,5, 5, 254, 1,-6, 1,2, 14, 4, 1,-2, 13, 4, 255

Salida:

Puerto 1: 6, 2, 1

Puerto 3: 720, 2, 1, 2, 720, 1

Puerto 5: 2, 720, 1, -3, 6

Bitácora:

Puerto 2: 0, 1, 1, 16, 0, 1, 2, 16, 0, 1, 3, 16, 0, 9, 16, 0, 5, 16, 0, 2, 3, 16, 0, 9, 16, 0, 5, 16, 0, 3, 4

Puerto 4: 16, 0, 7, 16, 0, 9, 16, 0, 5, 16, 0, 999, 2, 0, -999, 2, 0, 2, 5, 16, 0, 5, 16, 0, 254, 16, 0, 1, -6, 16, 0, 1, 2, 16, 0, 14, 16, 0, 4, 16, 0, 1, -2, 16, 0, 13, 16, 0, 4, 16, 0, 255, 16

Carga de la pila hasta desborde

Entrada:

1, 1, 1, 2, 1, 3, 1, 4, 1, 5, 1, 6, 1, 7, 1, 8, 1, 9, 1, 10, 1, 11, 1, 12, 1, 13, 1, 14, 1, 15, 1, 16, 1, 17, 1, 18, 1, 19, 1, 20, 1, 21, 1, 22, 1, 23, 1, 24, 1, 25, 1, 26, 1, 27, 1, 28, 1, 29, 1, 30, 1, 31, 1, 32, 5, 255

Salida:

31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

Bitácora:

0, 1, 1, 16, 0, 1, 2, 16, 0, 1, 3, 16, 0, 1, 4, 16, 0, 1, 5, 16, 0, 1, 6, 16, 0, 1, 7, 16, 0, 1, 8, 16, 0, 1, 9, 16, 0, 1, 10, 16, 0, 1, 11, 16, 0, 1, 12, 16, 0, 1, 13, 16, 0, 1, 14, 16, 0, 1, 15, 16, 0, 1, 16, 16, 0, 1, 17, 16, 0, 1, 18, 16, 0, 1, 19, 16, 0, 1, 20, 16, 0, 1, 21, 16, 0, 1, 22, 16, 0, 1, 23, 16, 0, 1, 24, 16, 0, 1, 25, 16, 0, 1, 26, 16, 0, 1, 27, 16, 0, 1, 28, 16, 0, 1, 29, 16, 0, 1, 30, 16, 0, 1, 31, 16, 0, 1, 32, 4, 0, 5, 16, 0, 255, 16

División

---

12 / 4 = 3

Entrada:

1, 12, 1, 4, 14, 4, 255 Salida:

Puerto 1: 3 Bitácora:

Puerto 2: 0, 1, 12, 16, 0, 1, 4, 16, 0, 14, 16, 0, 4, 16, 0, 255, 16

-12 / 4 = -3

Entrada:

1, -12, 1, 4, 14, 4, 255 Salida:

Puerto 1: -3 Bitácora:

Puerto 2: 0, 1, -12, 16, 0, 1, 4, 16, 0, 14, 16, 0, 4, 16, 0, 255, 16

12 / -4 = -3

Entrada:

1, 12, 1, -4, 14, 4, 255 Salida:

Puerto 1: -3 Bitácora:

Puerto 2: 0, 1, 12, 16, 0, 1, -4, 16, 0, 14, 16, 0, 4, 16, 0, 255, 16

-12 / -4 = 3

Entrada:

1, -12, 1, -4, 14, 4, 255 Salida:

Puerto 1: 3 Bitácora:

Puerto 2: 0, 1, -12, 16, 0, 1, -4, 16, 0, 14, 16, 0, 4, 16, 0, 255, 16

Módulo

12 % 5 = 2

Entrada:

1, 12, 1, 5, 15, 4, 255 Salida:

Puerto 1: 2 Bitácora:

Puerto 2: 0, 1, 12, 16, 0, 1, 5, 16, 0, 15, 16, 0, 4, 16, 0, 255, 16

-12 % 5 = -2

Entrada:

1, -12, 1, 5, 15, 4, 255 Salida:

Puerto 1: -2 Bitácora:

Puerto 2: 0, 1, -12, 16, 0, 1, 5, 16, 0, 15, 16, 0, 4, 16, 0, 255, 16

12 % -5 = 2

Entrada:

1, 12, 1, -5, 15, 4, 255 Salida:

Puerto 1: 2 Bitácora:

Puerto 2: 0, 1, 12, 16, 0, 1, -5, 16, 0, 15, 16, 0, 4, 16, 0, 255, 16

-12 % -5 = -2

Entrada:

1, -12, 1, -5, 15, 4, 255 Salida:

Puerto 1: -2 Bitácora:

Puerto 2: 0, 1, -12, 16, 0, 1, -5, 16, 0, 15, 16, 0, 4, 16, 0, 255, 16

Shift

0111 1111 1111 1111 « 15 = 0x8000

Entrada:

1, 0x7FFF, 1, 15, 18, 4, 255 Salida:

Puerto 1: -32768 Bitácora:

Puerto 2: 0, 1, 32767, 16, 0, 1, 15, 16, 0, 18, 16, 0, 4, 16, 0, 255, 16

»0111 1111 1111 1111 “ 14 = 0xC000

Entrada:

1, 0x7FFF, 1, 14, 18, 4, 255 Salida:

Puerto 1: -16384 Bitácora:

Puerto 2: 0, 1, 32767, 16, 0, 1, 14, 16, 0, 18, 16, 0, 4, 16, 0, 255, 16

»”0111 1111 1111 1111 ” 15 = 0x0000

Entrada:

1, 0x7FFF, 1, 15, 19, 4, 255 Salida:

Puerto 1: 0 Bitácora:

Puerto 2: 0, 1, 32767, 16, 0, 1, 15, 16, 0, 19, 16, 0, 4, 16, 0, 255, 16

»0111 1111 1111 1111 » 14 = 0x0001

Entrada:

1, 0x7FFF, 1, 14, 19, 4, 255 Salida:

Puerto 1: 1 Bitácora:

Puerto 2: 0, 1, 32767, 16, 0, 1, 14, 16, 0, 19, 16, 0, 4, 16, 0, 255, 16

1000 0000 0000 0000 15 = 0xFFFF

Entrada:

1, 0x8000, 1, 15, 19, 4, 255 Salida:

Puerto 1: -1 Bitácora:

Puerto 2: 0, 1, -32768, 16, 0, 1, 15, 16, 0, 19, 16, 0, 4, 16, 0, 255, 16

1000 0000 0000 0000 14 = 0xFFFE

Entrada:

1, 0x8000, 1, 14, 19, 4, 255 Salida:

Puerto 1: -2 Bitácora:

Puerto 2: 0, 1, -32768, 16, 0, 1, 14, 16, 0, 19, 16, 0, 4, 16, 0, 255, 16

XI. Referencias

Referencias

[1] *A switch/case Implementation in Assembly*. 2022. URL: https://microchipdeveloper.com/tip:3.

[2] *Cartilla Reducida Intel 8086*. 2022. URL: https://eva.fing.edu.uy/pluginfile.php/28277/mod_resource/content/3/cartillas/cartillaIntel.pdf.

[3] *Ensamblador 8086*. 2022. URL: https://www.cs.buap.mx/~mgonzalez/asm_mododir2.pdf.

[4] *Jump Table*. 2022. URL: https://en.wikipedia.org/wiki/Branch_table#Typical_implementation.

[5] *Manual de Usuario - ArquiSim*. 2022. URL: https://eva.fing.edu.uy/pluginfile.php/144479/mod_resource/content/2/ManualDeUsuario.pdf.

[6] *Notación Polaca Inversa*. 2022. URL: https://es.wikipedia.org/wiki/Notaci%C3%B3n_polaca_inversa.

[7] *Simulador ArquiSim version 1.3.7.2*. 2022. URL: https://eva.fing.edu.uy/mod/resource/view.php?id=181524.

[8] *Sitio EVA - Arquitectura en Computadoras*. 2022. URL: https://eva.fing.edu.uy/course/view.php?id=195.

[9] *Switch Statements and Jump Tables*. 2022. URL: https://sites.google.com/site/arch1utep/jump-tables.