Diseño e implementación de un emulador de instrucciones MIPS en una arquitectura x86 de 64 bits.

Proyecto 1 de Laboratorio de Estructura de Microprocesadores

Danny Gabriel Mejías Anchía *(2014159999)*Javier Alonso Cordero Quirós*(2014115782)*Keylor Andrés Mena Venegas *(2014108164)*Luis Gerardo Leon Vega *(2014069639)*Luis Orlando Merayo Gatica *(2014049811)*

EL4313 – Lab. Estructura Microprocesadores

1er Semestre 2017

Tecnológico de Costa Rica

*Abstract*—This paper details the design, development and implementation process of a MIPS architecture (32 bits) emulator that runs in a x86 (64 bits) computer with Ubuntu 16.04 LTS as operative system (OS). Likewise, the techniques used during the project development process are analyzed, discussing and contrasting some of these techniques to finish successfully and with an acceptable throughput the emulator.

*Resumen*—En este documento, se detallan aspectos del proceso de diseño, desarrollo e implementación de un emulador de una arquitectura MIPS de 32 bits sobre un computador de arquitectura x86 de 64 bits con sistema operativo Ubuntu 16.04 LTS. Asimismo, se esbozará el análisis de las técnicas empleadas para el proceso de desarrollo del proyecto, discutiendo y discriminando algunas de estas para llegar de forma adecuada a una solución final.

# Introducción

El objetivo de este proyecto es desarrollar un emulador de una arquitectura MIPS capaz de ejecutar instrucciones en este set de instrucciones, ensamblado en un ordenador con una arquitectura x86 de 64 bits con un sistema operativo Ubuntu 16.04. El alcance de este documento se limitará a esbozar y realizar análisis del proceso de diseño, destacar el proceso de desarrollo y analizar los resultados obtenidos en pro del cumplimiento de los requisitos del sistema. Este emulador fue programado en ensamblador x86\_64 para cumplir con los requisitos mínimos del proyecto y, además, es capaz leer el programa mediante un archivo con instrucciones decodificadas en MIPS en formado de texto plano ASCII, satisfaciendo los requisitos del sistema.

# Marco Teórico

## Ambiente computacional x86\_64 y Ubuntu 16.04

El ambiente con arquitectura x86\_64 es una de las más comunes actualmente, ya que predominan en el mercado de los ordenadores personales portátiles y de escritorios, siendo muy utilizados para aplicaciones en el hogar, la oficina, la educación y la empresa.

Este ambiente requiere de un sistema operativo para poder extraer todo su potencial y facilitar su uso al usuario, facilitando comandos amigables y un entorno de interfaz gráfica. Uno de esos sistemas operativos es Ubuntu 16.04, proyecto GNU-Linux soportado la empresa Canonical [1]. Esta empresa se encarga de darle continuación al proyecto sacando versiones de este sistema operativo cada seis meses, sin embargo, solo los años pares y versiones de abril (XX.04) son las que tienen servicio de soporte especial (LTS – Long Term Support). Esto sugiere que el sistema operativo está en constante renovación, lo que permite que se mantenga a la vanguardia de la seguridad de los sistemas e innovación del software para optimizarlo a los sistemas modernos.

Otra de las ventajas de este sistema operativo es que es de código abierto y gratuito, es decir, puede copiarse, modificarse y derivar productos con base a este sistema operativo sin que haya problemas de derechos de autor. Asimismo, sigue la ideología GNU [2].

Por otro lado, algo que hace atractivo a los sistemas operativos Linux como Ubuntu, es su flexibilidad, su estabilidad y su manejo óptimo de recursos. Existen una gran variedad de programas y aplicaciones que corren sobre este sistema operativo y existen grandes comunidades que brindan soporte para el desarrollo y manejo de estos sistemas operativos, destacando los más populares como StackOverFlow y Ubuntu Forums. Eso facilita a los que se inicien en este sistema operativo o que quieran desarrollar para esta plataforma.

Asimismo, por ser Open Source, no requiere licencia para instalación, que a diferencia de sistemas operativos comerciales como Windows, se requiere el pago de una licencia para poder instalarlos y usarlos en equipos de arquitectura x86\_64.

## Ambiente de trabajo

.

* No posee una interfaz gráfica preinstalada.
* Se accede a él mediante consola SSH.
* Tiene un núcleo más ligero que el sistema operativo dirigido a escritorio, debido a que no posee aplicaciones preinstaladas e interfaz gráfica.

Ante esta situación, para preparar el sistema operativo se deben realizar las siguientes tareas:

1. *Instalar un entorno gráfico:*

Para el entorno gráfico, se ha seleccionado MATE [3]. Esta elección fue llevada a cabo debido a que es una de las más ligeras comparado a GNOME, KDE o Unity (default de Ubuntu Desktop). Eso facilitará la transferencia de interfaz gráfica mediante escritorio remoto.

Esta instalación garantiza instalar únicamente el núcleo, dejando por fuera aplicaciones adicionales, como OpenOffice.

1. *Instalar un servicio para escritorio remoto:*

Para poder acceder al servidor de forma gráfica y disfrutar del GUI (Graphical User Interface), es necesario instalar un servidor de compartición de escritorio. Se ha seleccionado el XRDP por su facilidad de instalación e implementación. Además, es posible conectarse usando el cliente de Escritorio Remoto de Windows [4].

1. *Crear un usuario para el Proyecto*

Agregar un usuario

| adduser tec |
| --- |

Después, se seguirán las instrucciones que implica ese comando, como configuración de la contraseña y credenciales.

1. *Instalar el NASM, el GCC y el SASM:*

Para desarrollar el emulador, es requerido un ensamblador (NASM), un debugger (GCC) y el entorno gráfico de desarrollo (SASM). Se ha elegido usar SASM[[1]](#footnote-1) como entorno IDE para poder usar el debugger de forma más amigable y tener mayor control sobre los “breakpoints”.

Ya con esto, queda preparado el ambiente de desarrollo. A partir de esto, ya se comenzó a desarrollar la solución.

## Otras consideraciones

.

*1. Nomenclatura y compatibilidad de los registros*

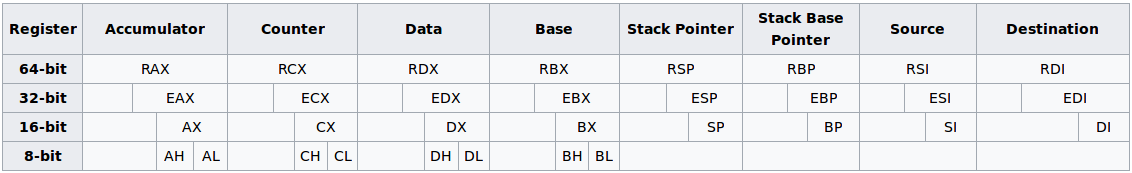
Para los registros, ambas arquitecturas tienen nomenclaturas diferentes para referirse a sus registros. Sin embargo, los registros de arquitecturas inferiores son compatibles con las nuevas arquitecturas, es decir, se pueden usar los registros de 32 bits en una arquitectura de 64 bits [5].

Figura 1 – Registros en las diversas arquitecturas x86 [6]

Como es posible observar mediante la figura 1, cuando se invoca un registro EAX en una arquitectura de 64 bits, el contenido se cargará en la parte baja del RAX (registro en 64 bits). Por otro lado, los registros de nombre “r8” - “r15” en formato de 64 bits, su parte baja tendrá como nomenclatura “r8d” - “r15d”.

*2. Sintaxis de las instrucciones*

Las instrucciones en el set x86 por lo general tienen dos operandos. El primer operando es, adicionalmente, el registro o campo de memoria destino del resultado de la operación. El segundo operando, por su parte, solo cumple un rol de operando, siendo posible usar registro, memoria o un valor inmediato [7]. Asimismo, este puede variar de acuerdo con la sintaxis empleada, siendo la sintaxis Intel la preferida para el desarrollo de este proyecto. Por ejemplo, para la instrucción:

| mov rax, rbx; mov dest, src |
| --- |

La instrucción dicta que el contenido del registro rbx se copie a rax.

# Solución Implementada

## Descripción de la solución

Para el proceso de desarrollo de la solución, se ha decidido usar un modelo comportamental de una arquitectura MIPS de 32 bits. Este modelo involucra el siguiente diagrama de bloques:

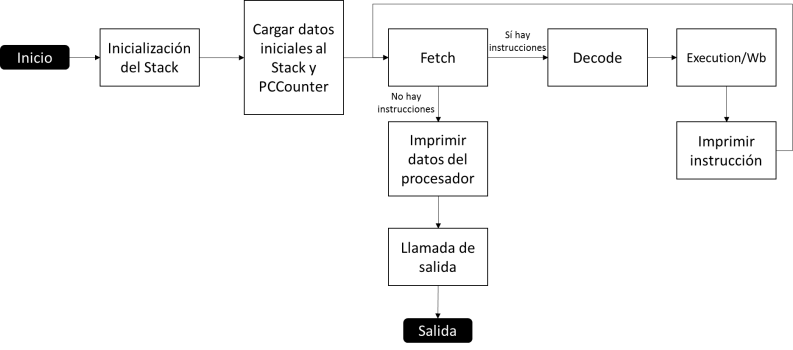


Figura 2 – Diagrama de bloques descriptivo de la solución

En el inicio de la solución, se inicializa el stack y el banco de registros, para poder cargar los argumentos del programa. Asimismo, se procede a cargar el archivo ROM.txt, que contiene toda la información de las instrucciones que deben ser ejecutadas y los datos que se almacenan en la memoria.

Realizado el proceso de preparación, se procede a inicializar el PC Counter en 00400000H para apuntar a la primera instrucción y se inicia el proceso de la arquitectura brincando al Fetch (proceso de adquisición de instrucción próxima). Posteriormente, al adquirir una instrucción, se inicia la decodificación de la misma, donde se determina el tipo de instrucción (sean R, J e I posibles) y que la instrucción sea válida. Luego se determina la localización de los registros que se requieren para ejecutar dicha instrucción y se llama a una subrutina que ejecuta la operación correspondiente al OpCode (Operation Code) obtenido.

La ejecución de dicha operación se realiza a cabo en el proceso de Execution/Wb, que se encarga de realizar la operación de forma comportamental en código x86\_64 y carga los datos en un stack de registros ya preconfigurado en el programa. Cuando se ejecuta alguna de las subrutinas que pertenecen a este proceso, realizan el “Write Back” al final de cada subrutina, siendo posible escribir en el registro contador de instrucciones (PC Counter), operaciones sobre memoria o, bien, escribir valores en los registros del procesador MIPS.

Asimismo, durante la ejecución, se llama al procedimiento para imprimir la instrucción que se está ejecutando en ese momento, los registros involucrados y la impresión de todos los registros empleables en la arquitectura emulada.

Al terminar la ejecución de la operación detectada, se procede a volver a la subrutina del “Fetch” para adquirir la próxima instrucción que se ejecutará. En caso de que haya instrucciones, el proceso de decodificación y ejecución se repiten de acuerdo con lo mencionado con anterioridad.

En caso contrario, se finalizan las estructuras del emulador de arquitectura y se procede a imprimir los datos de la arquitectura servidor (Host) y se realiza un llamado al sistema para finalizar la ejecución. Esto se resume en lo siguiente:

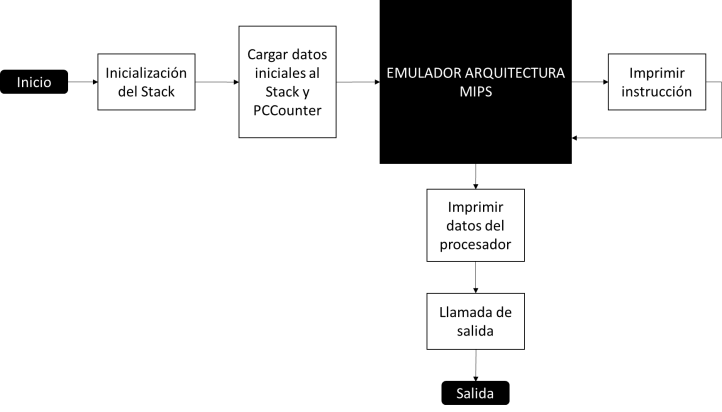


Figura 3 – Diagrama de bloques general de la solución

DESCRIPCION DE ETAPAS DEL PROCESO DEL DISEÑO

* + - 1. Inicialización del Stack

Para emular el banco de registros y el banco de memoria del programa, se hicieron dos arreglos de longitudes definidas previamente (150 palabras de 32 bits para las instrucciones y más de 100 palabras para le memoria de datos). Esto se realizó de esta forma para simplificar el proceso de desarrollo.

* + - 1. Carga de Datos

En la carga de datos, se lee el archivo de memoria “ROM.txt” línea por línea. Además, se tiene que discernir si el dato dado en hexadecimal-ASCII se encuentra entre corchetes (dirección), si está después del corchete (dato) y si está después del punto y coma (comentario). Para agregar cada instrucción, se mapea la dirección obtenida del archivo al arreglo de instrucciones y colocando el dato obtenido. Asimismo, se debe clasificar el dato si es una instrucción o si es un dato de memoria. Al finalizar la lectura del archivo, se inicializa el PC para proceder con la ejecución de la primera instrucción.

* + - 1. Obtención de Argumentos

Esta sección describe el proceso de obtención de los argumentos introducidos por el usuario. Para llamar al programa se debe abrir la terminal y abrir el directorio en el cual se encuentra el ejecutable. Luego se escribe el siguiente comando:

./emulador [arg0] [arg1] [arg2] [arg3]

donde “emulador” es el nombre del ejecutable y “[argx]” es uno de los argumentos. Es importante dejar un espacio entre estos para que el sistema pueda distinguir el principio y fin de cada uno. Para este proyecto los argumentos otras palabras, cada argumento debe ser de 8 son números de 32 bits en formato hexadecimal, en caracteres ya sea números o letras mayúsculas de A hasta F. Una vez que se llama al programa, el sistema guarda los argumentos en memoria en formato ASCII, y la dirección para acceder a ellos queda registrada en el Stack (parte amarilla) como se muestra en la figura 4:

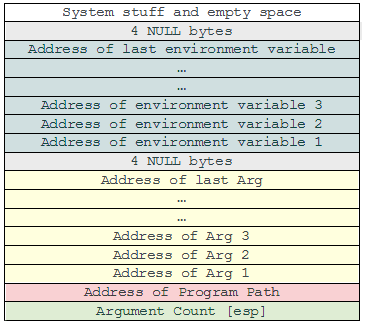


Figura 4. Diagrama de Stack del sistema

Además, el Stack cuenta con un registro de cuenta de argumentos (verde) que contiene la cantidad en binario de argumentos registrados. El mínimo de la cuenta va a ser 1 ya que siempre se cuenta la dirección del programa (rojo). Sabiendo esto, se puede programar una rutina para extraer los datos en memoria. El algoritmo consiste en extraer los datos de los argumentos haciendo uso correcto del stack. Se empuja el registro base (rbp) al Stack y se le asigna la dirección del registro puntero (rsp). Luego se revisa que el número de argumentos sea mayor a 1 para proceder a la extracción. Para obtener información del stack se utiliza la dirección de rbp y se le suma la posición del registro deseado multiplicado por 8. Se obtiene la dirección del argumento para luego copiar el contenido. Este último está en formato ASCII por lo que se llama al macro que lo convierte a binario. Finalmente, el resultado se copia en los registros de MIPS $a0, $a1, $a2, y $a3.

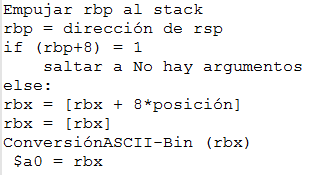


Figura 5. Pseudocódigo para obtención de argumentos

* + - 1. Fetch

Esta subrutina se encarga de localizar la siguiente instrucción que será ejecutada, asistida por el contador de programa (PC). Para ello, consulta el registro “r15” que es el PC y carga sobre “rdx” la instrucción a ejecutar. Asimismo, incrementa el PC cuando está por finalizar y verifica si todas las instrucciones fueron ejecutadas.

Por otro lado, asociado al Fetch, existe una etapa de “Pre-Decode”, que permite cargar sobre los registros de X86\_64 los operandos y el código de instrucción para facilitar la decodificación.

* + - 1. Decode

En esta sección se recibe la instrucción a ejecutar descompuesta en diferentes registros de la arquitectura x86\_64. Los pasos a seguir para implementar la decodificación son:

* 1. Identificación del código de operación

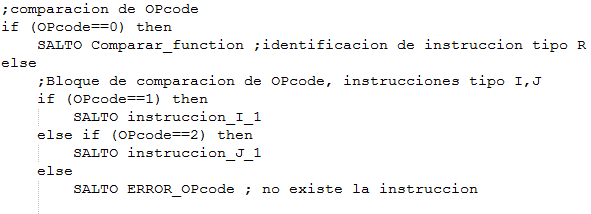
En este segmento del código, se compara el código de operación de la instrucción con los diferentes códigos de operación existentes para cada instrucción MIPS. Si su valor es “0” entonces es una instrucción tipo “R”, entonces es necesario determinar la función. Si es distinto de “0”, se salta a una pre-ejecución de la instrucción identificada. Si no se identifica un código de operación válido, el emulador anuncia un error en la ROM.

Figura 6. Pseudocódigo para identificación del código de operación

* 1. Identificación de la función

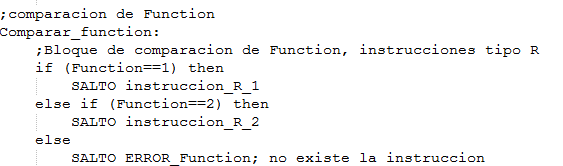
Todas las instrucciones tipo “R” tienen un código de operación con valor “0”. En este segmento del código determina cuál instrucción va a ejecutarse a partir de su valor de “*function*”, ubicado en el registro “r9” de x86, una vez identificada, hay un salto a la pre-ejecución de la misma. Si no se identifica una función válida, el emulador anuncia un error en la ROM.

Figura 7. Pseudocódigo para identificación del código de operación

* 1. Pre-Ejecución

Sobre este segmento de código, el emulador conoce qué instrucción debe ejecutarse, no obstante, desconoce que operandos utilizar y sobre cuál registro escribir el resultado. Como solución, se plantearon subrutinas que son llamadas antes de la ejecución de cada instrucción, donde el programa emula directamente la decodificación del banco de registros de MIPS.

* + 1. Obtención de operandos

Este conjunto de subrutinas accede al stack de registros creado en el inicio del programa y obtienen el valor del operando(s) a utilizar a partir de la dirección de los registros contemplados en la instrucción. Dado el stack creado, las direcciones deben ser alineadas para acceder al operando correcto.

* + 1. Definición de registro destino

Este conjunto de subrutinas prepara un registro de x86 que contiene un puntero a la dirección del registro en el stack donde se desea escribir el resultado luego de la ejecución. La dirección del destino deber ser alineada para acceder correctamente al stack.

El pseudocódigo de los pasos de acceso al stack se muestra a continuación:

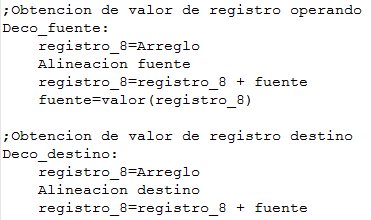


Figura 8. Pseudocódigo para pre-ejecución

1. Execution/WB

El programa tiene un bloque de código para la ejecución de cada instrucción MIPS. Primeramente, se despliega el nombre de la instrucción a ejecutar y los registros involucrados. Las instrucciones con signo se trabajan en 32 bits, según la compatibilidad de registros descrita anteriormente. Cuando el signo no es tomado en cuenta, las ejecuciones son en 64 bits. El programa emula un procesador MIPS uniciclo, pues las instrucciones se ejecutan una a una. Además, en este apartado, se puede detectar error de overflow para las instrucciones aritméticas con signo.

Luego de le ejecución, el programa escribe el resultado en el stack de registros, según el registro destino ubicado en la sección anterior.

El pseudocódigo de una ejecución se muestra a continuación:

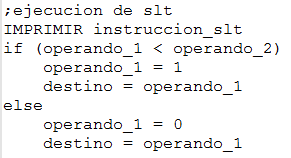


Figura 9. Pseudocódigo para ejecución de instrucción Slt

1. Impresión de registros

A través de un ciclo, se recorre el stack de registros y este se va imprimiendo en la consola y el archivo de resultados. El programa imprime el valor en hexadecimal; toman los valores en binario del registro, los traduce ASCII y hace un llamado de escritura. Una vez ejecutado este segmento de código, se salta al Fetch para buscar una nueva instrucción.

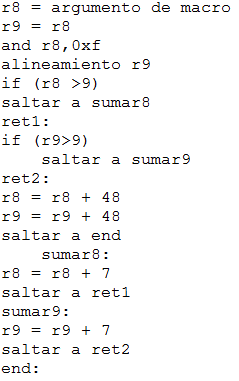


Figura 10. Pseudocódigo para ejecución de instrucción Slt

1. Impresión de Datos del procesador

En esta sección se obtiene el fabricante, modelo, familia y tipo del microprocesador de la computadora en que se ejecuta el programa. Para este proceso la instrucción CPUID es indispensable ya que dependiendo del valor de rax, el sistema entrega información del microprocesador en varios registros. El algoritmo para extraer esta información es el siguiente: Poner rax en 0 para obtener el fabricante con CPUID, la información se almacena en formato ascii y se almacena en rax, rbx, rcx y rdx. Entonces se crea una variable y se copia la información en forma concatenada. Para el modelo, familia y tipo se usa el rax en 1, e igualmente se llama CPUID. Esta vez la información solo estará contenida en rax pero en formato binario por ser números. Se extraen los datos y se convierten a ASCII para ser impresos en la terminal.

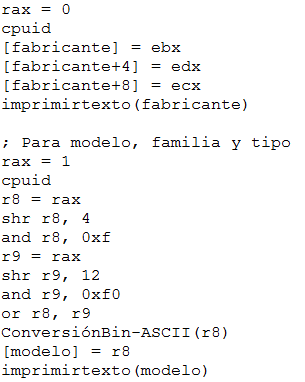


Figura 11. Pseudocódigo para ejecución de instrucción Slt

1. Impresión en archivo de resultados

La ejecución del programa, impresa en pantalla, se almacena en un archivo de texto, el cual se guarda si la ejecución resulta exitosa.

El archivo es abierto al principio de la ejecución del programa, para lograrlo, se utiliza el comando de llamada al sistema en x86 para abrir un archivo, y validarlo por medio de la escritura de los permisos y banderas necesarios.

La forma de escribir en el archivo es la misma que se utiliza para impresión de pantalla, no obstante, se escoge como destino el puntero que señala a la dirección del archivo.

1. Pantalla de Salida

Esta sección es alcanzada en caso de que se realice una ejecución exitosa, es decir, que el fetch haya leído la totalidad de las instrucciones, o si ocurre un error en la ejecución en el programa, en este caso se imprime en pantalla la causa. En ambos casos, se imprime en pantalla la información de los desarrolladores del programa.

## Limitaciones y recomendaciones

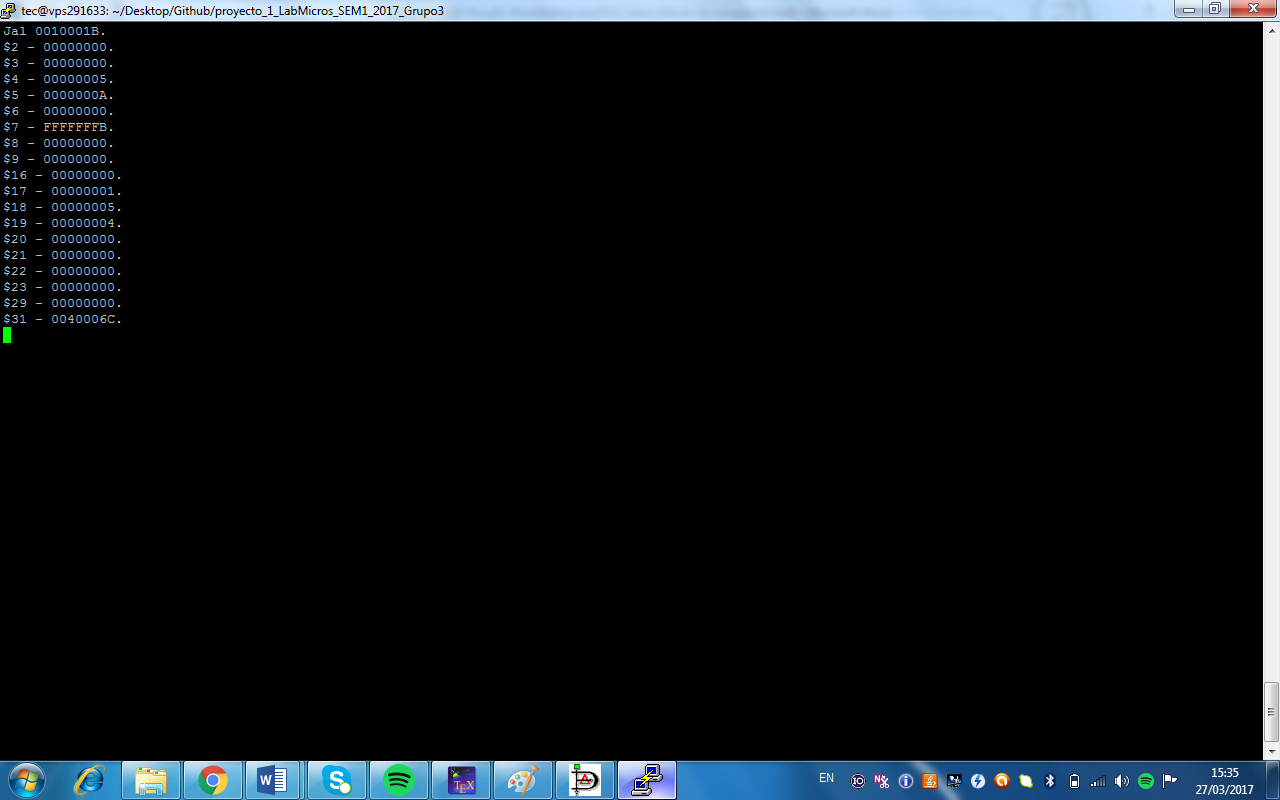
* + - 1. El programa no soporta un ROM de más de 150 instrucciones.
      2. El stack de datos no supera las 150 palabras.
      3. El programa reconoce una limitada cantidad de instrucciones MIPS. Entre los casos más críticos, no realiza almacenamiento de memoria.
      4. La instrucción de multiplicación no identifica error de overflow.
      5. Los argumentos de entrada del programa solo reconoce letras mayúsculas y cada argumento debe estar separado por un espacio
      6. El nombre del archivo a ejecutar debe ser “ROM.txt”
      7. El archivo “ROM.txt” debe de estar ubicado en la misma ruta del programa.

# Resultados

En esta sección se mostrara, por medio de imágenes la ejecución exitosa de un programa. En la figura 12 se muestra la pantalla de inicio, en donde el archivo ROM fue encontrado y leído de manera correcta. Luego de presionar enter, se corren las instrucciones en orden y se actualiza el banco de registros. La figura 13 muestra la ejecución de instrucciones de tipo R, I y J (Add, Ori y Jal). Una vez recorrida la totalidad de la ROM exitosamente, se despliega la pantalla de salida, figura 14. La figura 15 muestra la impresión de la ejecución un archivo de resultados.

##### 

Fig 12. Inicio del programa



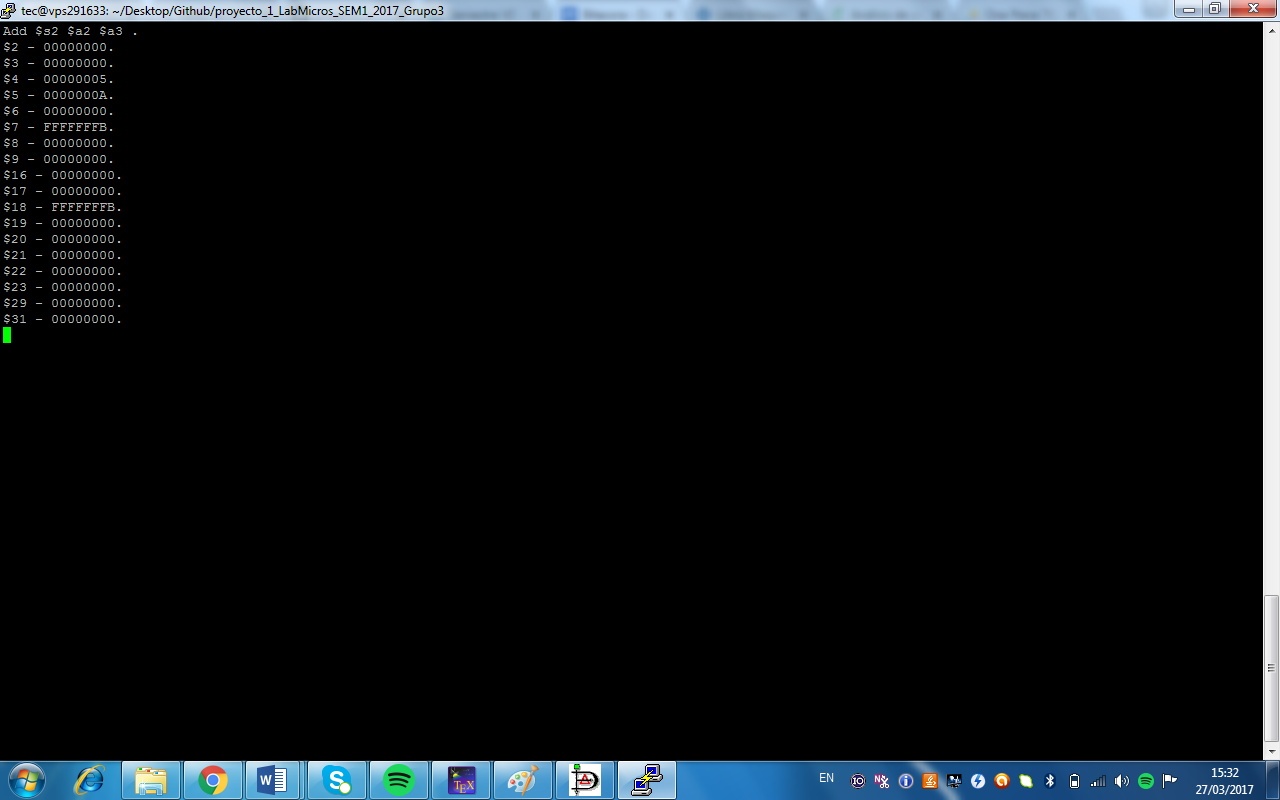
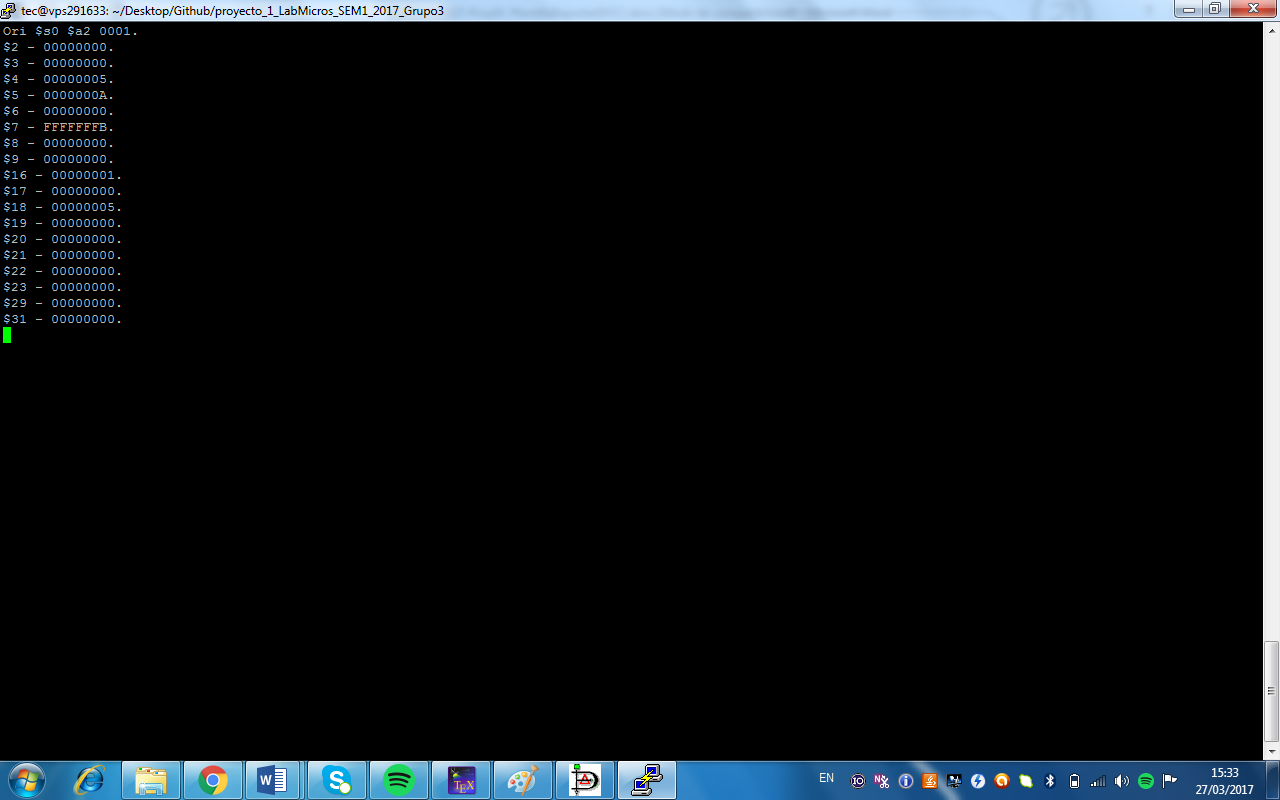


Fig 13. Ejecución de distintas instrucciones

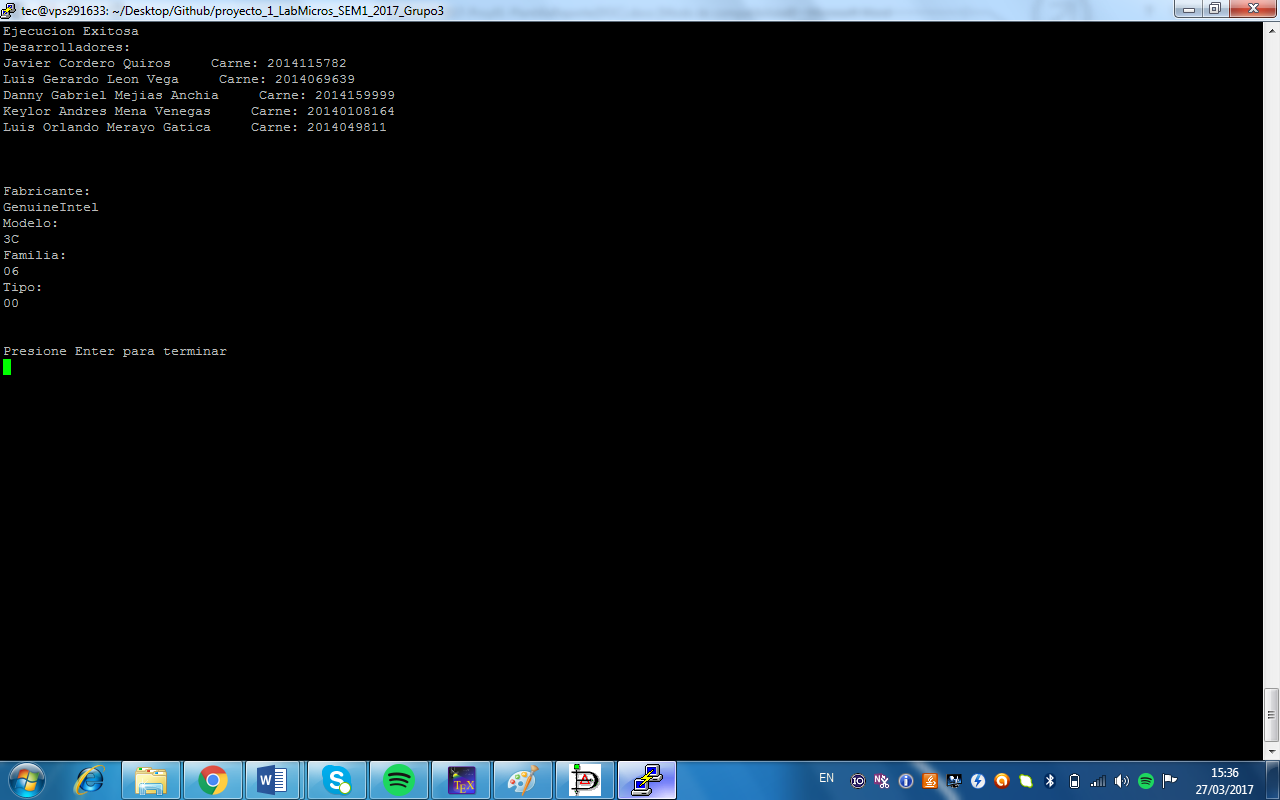


Fig 14. Finalización del programa

##### C:\Users\LuisOrlando\Downloads\fig6.png

##### 

Fig 15. Archivo de resultados

##### Conclusiones

* + - 1. Es importante reconocer los registros reservados para llamadas a subrutinas
      2. La utilización de macros disminuye la extensión del programa cuando se utilizan funcionas en más de una ocasión.
      3. Sólo se pueden operar sobre registros del mismo tamaño de bits.
      4. Al realizar el comando “syswrite” el dato debe de ser una cadena de caracteres o estar en formato ASCII para una correcta impresión.
      5. Es elemental conocer la estructuración de un programa en ensamblador

##### Referencias

Las referencias bibliográficas utilizan el siguiente formato:

1. B. Sánchez. “Cuaderno Práctico de Linux. Sistemas Operativos Monopuesto”, Segunda Edición, pp. 21, Septiembre 2015.
2. M. Overley. “The Open Source Handbook”, Pike & Fisher, USA, pp. 1-11, 2003.
3. "Install [wiki.mate-desktop.org]", Wiki.mate-desktop.org, 2017. [Online]. Available: http://wiki.mate-desktop.org/download. [Accessed: 13- Mar- 2017].
4. "xrdp", *Xrdp.org*, 2017. [Online]. Available: http://www.xrdp.org/. [Accessed: 13- Mar- 2017].
5. Lawlor, "x86\_64 NASM Assembly Quick Reference ("Cheat Sheet")", UAF Computer Science Department, 2007. [Online]. Available: https://www.cs.uaf.edu/2007/fall/cs301/support/x86\_64/index.html. [Accessed: 19- Feb- 2017].
6. "X86 Assembly/X86 Architecture - Wikibooks, open books for an open world", En.wikibooks.org, 2017. [Online]. Available: https://en.wikibooks.org/wiki/X86\_Assembly/X86\_Architecture. [Accessed: 19- Feb- 2017].
7. "X86-64 Architecture Guide", Cons.mit.edu. [Online]. Available: http://cons.mit.edu/sp14/x86-64-architecture-guide.html. [Accessed: 19- Feb- 2017].
8. N. Apellido1, N. Apellido2, and N. Apellido3, “Título” Casa Editora. Ciudad, vol. X, pp. 529–551, Mes Año.

1. Se deben instalar el NASM y GCC de primero antes del SASM, debido a que esto es únicamente un IDE. [↑](#footnote-ref-1)