DESARROLLO DE MATERIAL DIDÁCTICO PARA LA ASIGNATURA ARQUITECTURA DE COMPUTADORES BASADO EN LA PLATAFORMA SIE: ESTRUCTURA Y FUNCIONAMIENTO DEL PROCESADOR

PEDRO ABEL VARGAS PRIETO

Universidad Industrial de Santander

Facultad de Ingenierías Físico-Mecánicas

Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones

Bucaramanga

2012

DESARROLLO DE MATERIAL DIDÁCTICO PARA LA ASIGNATURA ARQUITECTURA DE COMPUTADORES BASADO EN LA PLATAFORMA SIE: ESTRUCTURA Y FUNCIONAMIENTO DEL PROCESADOR

PEDRO ABEL VARGAS PRIETO

Trabajo de grado presentado como requerimiento para optar al título de:

Ingeniero Electrónico

Tesis desarrollada para el grupo de investigación CPS

Director:
MSc. Jorge H. Ramón Suarez

Co-Director:
MSc. William A. Salamanca B.
ING. Carlos A. Angulo J.
MSc. Carlos A. Fajardo A.
MSc. Sergio A. Abreo C.

Universidad Industrial de Santander
Facultad de Ingenierías Físico-Mecánicas
Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones
Bucaramanga
2012

AGRADECIMIENTOS

Quisiera agradecer de manera muy especial a mi mentor y director Jorge H. Ramón Suarez, a mis codirectores Carlos A. Angulo, William A. Salamanca, Sergio A. Abreo, Carlos A. Fajardo a quienes les reconozco en especial sus enseñanzas, su apoyo y constante motivación. A mis compañeros de proyectos conjunto porque compartimos ratos de conocimiento y nos ayudamos mutuamente. A mis amigos por haber confiado y brindado apoyo en los momentos que mas lo necesitaba les agradezco de todo corazón y a todas las personas que de una u otra forma involucra este proyecto, en especial a aquellas a quienes va dirigido muchas gracias por sus aportes.

Este trabajo de grado está dedicado a mis padres que desde el cielo sé que me brindaron todo su amor y apoyo durante el transcurso de la carrera, quienes son el motivo de todos mis logros, a los que les debo todo lo bueno que tengo y con la ayuda de ellos seguiré forjando un futuro mejor.

Pedro Abel Vargas Prieto

Tabla de Contenido

PÁG:

Lista de Figuras

Lista de Anexos

Resumen

TÍTULO: DESARROLLO DE MATERIAL DIDÁCTICO PARA LA ASIGNATURA ARQUITECTURA DE COMPUTADORES BASADO EN LA PLATAFORMA SIE: ESTRUCTURA Y FUNCIONAMIENTO DEL PROCESADOR [*]

AUTOR: PEDRO ABEL VARGAS PRIETO [**]

PALABRAS CLAVE: Arquitectura de computadores, plataforma SIE, *software*, *hardware*, procesador, Linux, compilador GCC, depurador GDB.

El proyecto trata de renovar el material didáctico para el laboratorio de la asignatura Arquitectura de computadores. Se han tomado los temas principales y con cada uno de ellos se ha diseñado una actividad para usar como práctica de esta asignatura. Para este proceso se utilizará la plataforma SIE, la cual posee grandes ventajas tanto de *software* como de *hardware* para el aprendizaje, no sólo para esta asignatura sino para todo el área de sistemas digitales. Se construye un manual de guías donde se añaden una serie de prácticas que proporcionan al estudiante las habilidades necesarias para entender la estructura y el funcionamiento del procesador usando la metodología adecuada, en donde aplicará sus conocimientos y entenderá paso a paso el proceso de la actividad propuesta en cada guía hasta llegar al final, con la mejor comprensión del tema. El *software* que se utiliza para el progreso de las guías son: el sistema operativo Linux, el compilador GCC y el depurador GDB. Estas herramientas de desarrollo son gratuitas, lo que conlleva a que este material esté al alcance de cualquier estudiante, en especial para aquellos que no tienen los recursos para invertir en la adquisición de este *software*. También se elabora un manual de usuario que complementa el conjunto de prácticas donde se encuentran aspectos complementarios relacionados con el desarrollo de las guías en la plataforma SIE y otros detalles necesarios para dar claridad a cada una de las prácticas. Finalmente se hizo un proceso de validación por medio de un grupo de estudiantes, que arrojó falencias en cada una de las prácticas y permitió las correspondientes modificaciones para obtener un material académico de calidad.

[*] Trabajo de grado

[**] **Facultad**: Ingenierías Físico-Mecánicas.**Escuela:** Ingenierías Eléctrica, Electrónica y de Telecomunicaciones.**Grupo de Investigación:** CPS. **Director:** MSc. Jorge H. Ramón Suarez. **Codirectores:** MSc. William A. Salamanca B., ING. Carlos A. Angulo J., MSc. Carlos A. Fajardo A., MSc. Sergio A. Abreo C.

Abstract

TITLE: DEVELOPMENT OF A DIDACTIC MATERIAL TO COMPUTER ARCHITECTURE COURSES BASED ON THE SIE PLATFORM: STRUCTURE AND OPERATION OF THE PROCESSOR [*]

AUTHOR: PEDRO ABEL VARGAS PRIETO [**]

KEY WORDS: Computer architecture, SIE Board, software, hardware, Linux, Processor, GCC compiler, GDB debugger.

This project is intended to renew the didactic material of the computer architecture laboratory. The main topics of the computer architecture have been taken in order to design activities as a guide. The Platform SIE will be used for this purpose, it has many advantages in hardware and software to help students to learn concepts in all subjects belong the digital systems area. A set of guide is designed, which contains a set of activities intended to understand internal processor structure and its operation, by using an appropriate methodology where the student will apply his knowledge and will understand step by step the activity with the best understanding of the topic. The Software tools used in this project are: Linux operative system, GCC compiler and GDB debugger. This software tools are free, hence could be used by any student, specially those who don't have enough money to afford this kind of packages. A user guide is designed as complement to the set of activities having aspects related to developments in the platform SIE and other details needed to clarify the guide. Finally a validation process was performed with a group of students allowing detection of deficiencies in each guide and subsequently modifications and corrections in order to get a high quality material.

[*] Degree project

[**] **Faculty**: Physico-mechanical Engineering .**School:** Electrical, Electronics and Telecommunications Engineering.**Research group:** CPS. **Director:** MSc. Jorge H. Ramón Suarez. **Codirectors:** MSc. William A. Salamanca B., ING. Carlos A. Angulo J., MSc. Carlos A. Fajardo A., MSc. Sergio A. Abreo C.

INTRODUCCIÓN

Existen en el mercado algunas plataformas de desarrollo que pueden ser utilizadas en la enseñanza del área de sistemas digitales, estas proporcionan una gran variedad de recursos de *hardware* y *software*, con algunas ventajas como: posibilidad de modificación, disponibilidad de los diseños de los esquemáticos de las placas de circuitos impresos, en todos los módulos se suministra la información necesaria para su comprensión, se suministra el código de algunos trabajos diseñados en estas plataformas y utilizan componentes fáciles de conseguir.

Surge la necesidad de implantar una nueva plataforma con las ventajas mencionadas anteriormente, que sea común para toda el área de sistemas digitales, que no limite al estudiante en su proceso de formación y en la posibilidad de desarrollar nuevos proyectos conjuntos. Por esta razón, se propone un conjunto de proyectos donde se diseñan prácticas de laboratorio para las asignaturas del área de sistemas digitales, basados en la plataforma SIE y que sirvan cómo apoyo para la asignatura correspondiente.

El presente trabajo es un proyecto conjunto que consiste en desarrollar prácticas de laboratorio para la asignatura Arquitectura de computadores basado en la plataforma SIE con énfasis en la estructura y funcionamiento del procesador. Se presenta un manual de guías referentes a los principales temas de la asignatura, apoyadas por un manual de usuario que contiene algunos procesos de configuración de la plataforma, explicación de algunos comandos utilizados y conceptos necesarios para el desarrollo de las prácticas.

El primer capítulo del manual de guías es destinado a la programación básica del lenguaje ensamblador, con el propósito de brindarle al estudiante una herramienta con la cual se facilite programar en *assembler*. Este lenguaje está basado en la arquitectura

MIPS. Se mostrará en detalle el juego de instrucciones, así como las técnicas de programación en ensamblador en dicha arquitectura. Se presentan ejemplos que se refieren al compilador gcc *(GNU compiler collection)*. El segundo capítulo se dedica al estudio y al manejo de un depurador que facilite al estudiante detectar muchos de los problemas que se presentan al momento de la ejecución de un programa por medio del análisis y plantear la solución de forma sencilla. Se trabajará con el depurador GDB y se enfocará en la arquitectura MIPS.

El tercer capítulo está dedicado al análisis de los principales modos de direccionamiento de la arquitectura MIPS. El modo de direccionamiento especifica la forma de interpretar la información contenida en cada campo del formato de instrucciones (I, J o K), es así como el repertorio de instrucciones sirve cómo punto de partida para poder interpretar el funcionamiento de los modos de direccionamiento que se pueden presentar en la arquitectura MIPS.

En el cuarto y último capítulo de este manual de guías, se analizará uno de los temas más importantes de la asignatura como es la jerarquía de memoria. Se analizan los niveles básicos presentes en una jerarquía, teniendo como tema principal el rendimiento de la memoria caché, todo esto con el fin de entender de manera teórica y práctica las memorias, los tipos de memorias existen y cómo es su interacción con el procesador.

1. ESPECIFICACIONES DEL PROYECTO

1.1 Objetivo general

Desarrollar material didáctico de laboratorio para la asignatura Arquitectura de Computadores empleando la plataforma SIE, con énfasis en la estructura interna y funcional del procesador.

1.1.1. Objetivos específicos

- Crear un manual de usuario intermedio de la plataforma SIE orientado a la estructura interna y funcional del procesador, para uso en la asignatura Arquitectura de Computadores.
- Elaborar un conjunto de prácticas de manera que el alumno adquiera y aplique los conocimientos acerca de la arquitectura interna de un procesador.
- Validar cada una de las prácticas aplicadas a la asignatura.

1.2 Alcances

Este proyecto comprende los siguientes alcances:

- Suministrar la documentación adecuada, a modo de referencia, sobre los recursos que se utilizarán en el desarrollo de las prácticas, tales como la estructura interna del procesador, características, programación en C y *assembler*. Esto con el fin de que el estudiante obtenga lo necesario para el desarrollo de las prácticas y/o proyectos.
- Cada práctica estará diseñada con base en la estructura y funcionamiento del procesador, las cuales incluirán el set de instrucciones, el manejo de un depurador, modos de direccionamiento y jerarquías de memoria.

- Se validará cada una de las prácticas por medio de un grupo de estudiantes con el fin de evaluar la claridad de cada una de ellas. Estos resultados se debatirán con el grupo de prueba y un equipo asesor del proyecto, para unificar los conceptos, identificar falencias y proporcionar las correspondientes correcciones.

2. GENERALIDADES

2.1 PLATAFORMA SIE.

Esta plataforma fue diseñada con el objetivo de permitir de una manera didáctica el desarrollo de *hardware* de aplicaciones libres que permita la distribución y modificación del diseño, con el único requisito de que los productos derivados deben tener la misma licencia y deben dar crédito al autor del trabajo original, lo que constituye la base de los productos *hardware copyleft*.

El objetivo principal es difundir este conocimiento a quien esté interesado, con el único requisito de mejorarlo y mantenerlo. Por esta razón SIE está inspirada en el movimiento FOSS (*Free and Open Source Software*). Los dispositivos *hardware copyleft* comparten la misma filosofía, y son el complemento perfecto del *software* libre. Para que un dispositivo *hardware* sea reproducible y modificable es necesario suministrar los archivos necesarios para la fabricación.

La plataforma SIE fue diseñada por el profesor Carlos Ivan Camargo de la Universidad Nacional de Colombia como herramienta de implementación de los cursos del área de sistemas digitales para dicha universidad y está diseñada para que pueda ser manipulada fácilmente por el que esté interesado, brindándole toda la información necesaria para su correcta operación y de esta manera compartir información que pueda ser de ayuda a otros desarrolladores de *hardware* que lo necesiten [1]. En la Figura 2.1 se puede observar la tarjeta junto con los siguientes dispositivos que la conforman:

1. Procesador MIPS Ingenic JZ4725.
2. FPGA XC3S100E_VQ100 que proporciona 25 puertos de entrada/salida de propósito general con señales digitales (rango 0-3.3V).

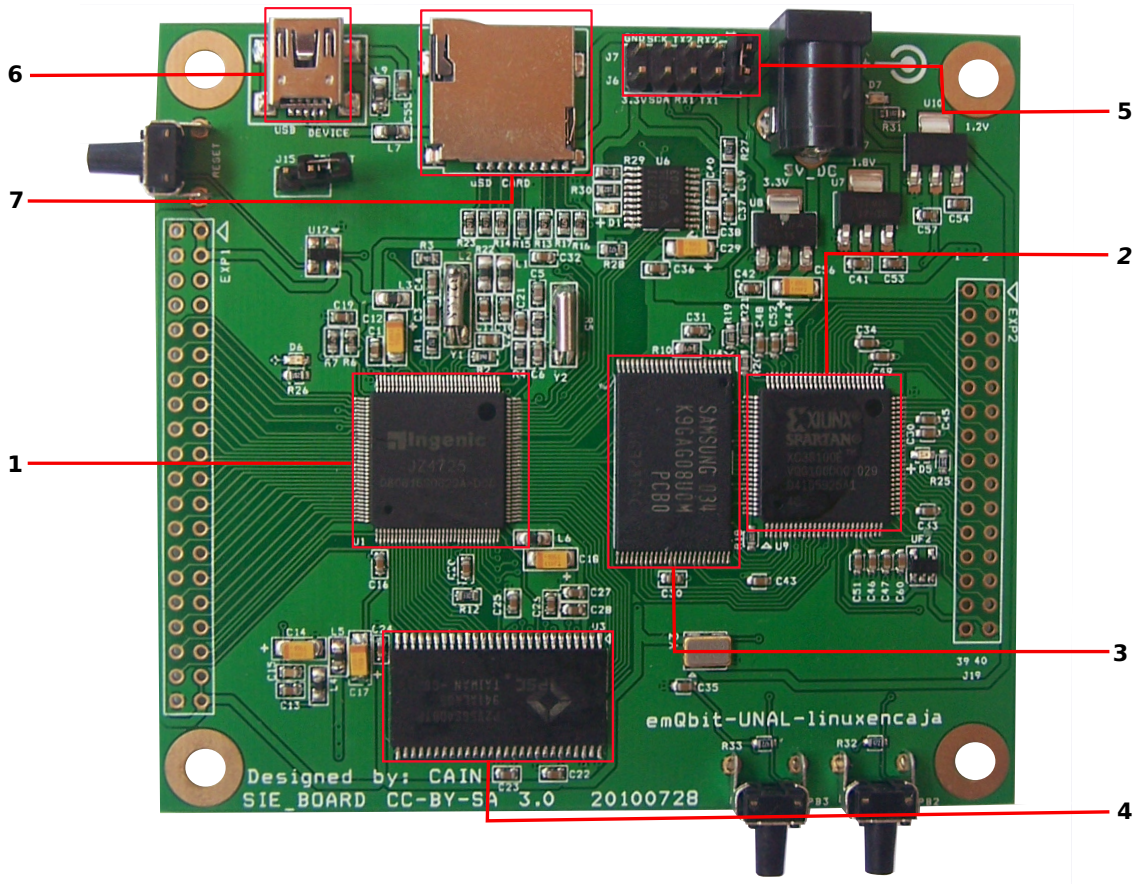

Figura 2.1: Plataforma de desarrollo SIE

FUENTE: [7].

3. Memoria NAND de 2GB.
4. 64MB de memoria SDRAM.
5. Puerto RS-232 Serial UART.
6. El Puerto USB puede ser usado como Ethernet, o dispositivo de consola serial.
7. Puerto Micro SD.

En la figura 2.2 se muestra el diagrama de bloques de la SIE.

El procesador de aplicaciones multimedia JZ4725 lleva incorporado un procesador Xburst de 32 bits, basado en la arquitectura MIPS, que contiene 32 registros, una caché

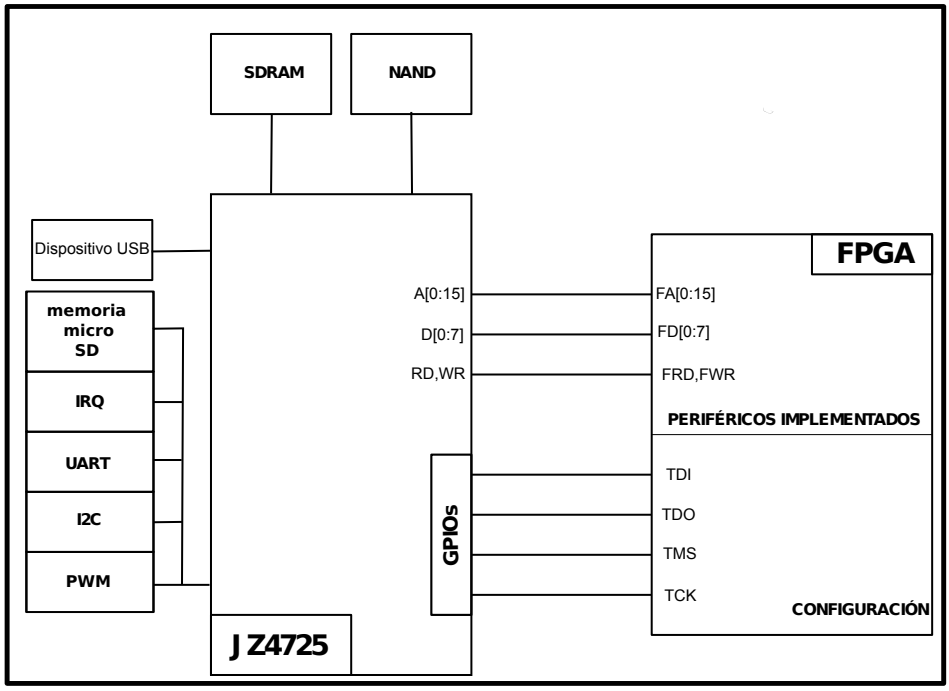

Figura 2.2: Diagrama de bloques de SIE.

FUENTE: [6].

de un tamaño de 16 KB para datos y 16 KB para instrucciones, llegando a manejar una velocidad de hasta 360 MHz. En la figura 2.3 se puede observar el diagrama de bloques de este procesador junto con todos los periféricos que lo conforman.

2.2 COMUNICACIÓN

La tarjeta SIE se comunica por medio del puerto USB el cual es configurado para poder ser utilizado como una interfaz de red (usb0). Por medio de esta comunicación es posible la transferencia de archivos y la ejecución de una consola remota utilizando el protocolo ssh. Este canal también puede ser utilizado como alimentación de la tarjeta por medio de un cable USB.

2.3 SISTEMA OPERATIVO

La tarjeta SIE tiene integrado un procesador JZ4725 que funciona bajo el sistema operativo *OpenWrt*, el cual es una distribución de *Linux* creada para dispositivos embebidos.

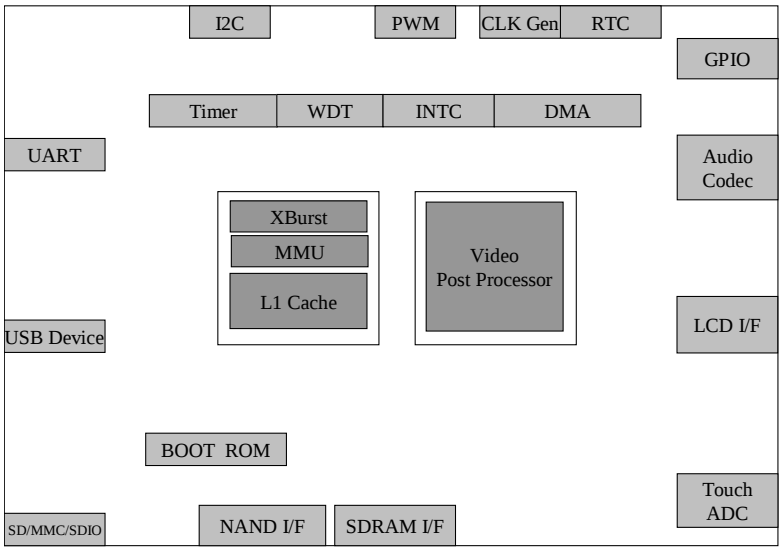

Figura 2.3: Diagrama del procesador

FUENTE: [3].

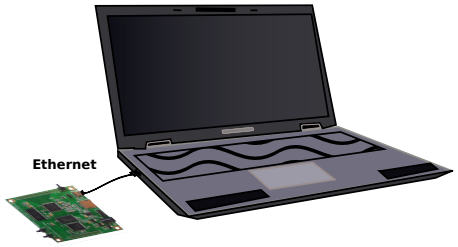

Figura 2.4: Comunicación tarjeta SIE con el PC

FUENTE: El autor.

OpenWrt se construye desde la base para ser una herramienta que permite personalizar el sistema mediante la inclusión de paquetes y utiliza principalmente una interfaz de línea de comando.

Para poder empezar a interactuar con la plataforma SIE, es necesario tener en nuestro computador una imagen del sistema operativo *OpenWrt* para posteriormente instalarla en la plataforma SIE por medio del proceso de *flasheo*, además de un conjunto de herramientas de compilación que permitirán diseñar aplicaciones de *software* necesarias para cumplir con éxito el desarrollo de cada una de las guías propuestas en este proyecto.

Entre las herramientas de compilación más importantes están incluidas: el compilador cruzado, programas de desarrollo cruzado, depuradores, etc.

Este proyecto está encaminado al desarrollo de prácticas de laboratorio para ayudar al estudiante a comprender la estructura interna y funcional del procesador, por este motivo se debe tener instalado *OpenWrt* en la plataforma SIE junto con las herramientas de compilación y así poder administrar los recursos del procesador, permitiendo que el usuario tenga acceso y esté en capacidad de manipularlo desde el sistema raíz.

Por último, cuando se tengan instaladas las herramientas de compilación de *OpenWrt* es probable que se haya instalado una versión del kernel distinta a la que está en la tarjeta SIE. Por esta razón se hace necesario realizar un procedimiento que permite instalar esta imagen del kernel para reemplazar la de la tarjeta y así evitar problemas de compatibilidad. Este procedimiento se conoce como *flasheo* y es necesario para el correcto funcionamiento de las herramientas que se han compilado.

Los procedimientos de instalación de herramientas de compilación de *OpenWrt* y *flasheo*, están descritos con más detalle en el manual de usuario anexo a este trabajo.

2.4 COMPILACIÓN DE PROGRAMAS

El programa escrito en un lenguaje de programación es llamado fuente y no se puede ejecutar directamente en una computadora. Lo que se debe hacer es compilar el programa obteniendo un archivo objeto y a partir de este se obtiene el ejecutable.

Normalmente la creación de un programa ejecutable conlleva dos pasos. El primer paso se llama compilación, que traduce el código fuente escrito en un lenguaje de programación a código en bajo nivel (normalmente en código objeto, no directamente a lenguaje máquina). El segundo paso se llama enlazado en el cual se une el código de bajo nivel generado de todos los ficheros y subprogramas que se han compilado con el código de

las funciones que hay en las bibliotecas del compilador, para que el ejecutable pueda comunicarse directamente con el sistema operativo, traduciendo así finalmente el código objeto a código máquina y generando un módulo ejecutable.

Estos dos pasos se pueden hacer por separado almacenando el resultado de la fase de compilación en archivos objetos (el típico *.o*) para enlazarlos en fases posteriores, o crear directamente el ejecutable con lo que la fase de compilación se almacena sólo temporalmente. Un programa podría tener partes escritas en varios lenguajes (por ejemplo *assembler*, C y C++), que se podrían compilar de forma independiente y luego enlazar para formar un único módulo ejecutable.

2.4.1. Compilador Cruzado

Un compilador cruzado es capaz de crear código ejecutable para otra arquitectura distinta de aquélla en la que él se ejecuta. Esta herramienta es útil cuando se quiere compilar código para una plataforma a la que no se tiene acceso o cuando es incómodo o imposible compilar en dicha plataforma. Para este caso se prefiere instalar dicho compilador en un equipo externo que puede ser un computador personal, debido a que si se hace en la plataforma SIE, gastaría demasiados recursos. El compilador cruzado que se utiliza es GCC, el cual es apto para crear código ejecutable para la arquitectura MIPS.

En el desarrollo de este proyecto se utilizan dos tipos de lenguaje de programación: C y *assembler*, el proceso de compilación para estos dos tipos de programas es similar. Los pasos para compilar y crear el ejecutable de un programa diseñado en lenguaje *assembler* para la plataforma SIE, son los siguientes:

1. Abrir una consola en el equipo y ubicarse en la carpeta donde está guardado el programa con extensión *.s*.
2. Digitar *mipsel-openwrt-linux-gcc -c -g ejemplo.s*, seguido de la tecla *enter*. El en-

samblador lee el archivo *.s* y genera un archivo objeto relocalizable el cual guarda con extensión *.o*, como se puede observar en la Figura 2.5.

Figura 2.5: Diagrama de generación del archivo objeto.

FUENTE: El autor.

3. Luego se digita *mipsel-openwrt-linux-gcc ejemplo.s -o ejemplo*, seguido de la tecla *enter*. El enlazador une el archivo objeto con la librería, produciendo en su salida el ejecutable del programa llamado *ejemplo*. Esto se puede visualizar en el diagrama de la Figura 2.6.

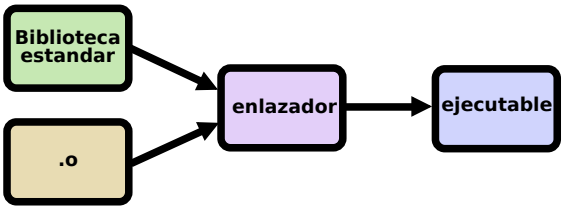

Figura 2.6: Diagrama de generación del ejecutable del programa.

FUENTE: El autor.

```
1 $ mipsel-openwrt-linux-gcc -c -g ejemplo.s
2 $ mipsel-openwrt-linux-gcc ejemplo.s -g -o ejemplo
```

Script 2.1: Compilación y generación del ejecutable

3. METODOLOGÍA

Estudio de la arquitectura del procesador y de la plataforma SIE.

Se estudió la arquitectura del procesador y la plataforma de desarrollo SIE basándose en aspectos como características de los elementos que los conforman y recursos que se utilizaron en el trascurso del proyecto. Lo anterior se hizo para conocer la tarjeta SIE, ya que era una herramienta novedosa y no se tenía experiencia en cuanto a su manejo. De todos los dispositivos que integran la plataforma SIE se hizo énfasis en el procesador, más específicamente en su funcionamiento.

Redacción del manual de usuario.

Teniendo en cuenta los recursos y características tanto de la plataforma SIE como de los elementos del procesador que se estudiaron en el paso anterior, se inició la elaboración de un manual práctico con la documentación requerida para el manejo de la plataforma SIE y las herramientas que se deben instalar y que son necesarias para llevar a cabo el desarrollo de las guías. Este manual se redactó a medida que se avanzó en la selección y construcción de cada una de las prácticas.

Selección de los temas para las prácticas.

Teniendo en cuenta los temas principales de la asignatura Arquitectura de Computadores, las características de la arquitectura del procesador JZ4725 y la viabilidad de implementar una práctica de cada tema en la plataforma SIE, se diseñaron las guías acorde a los siguientes temas: set de instrucciones, manejo de un depurador, modos de direccionamiento y jerarquía de memoria.

Diseño de las prácticas.

Con los temas definidos se procedió a investigar y documentar cada uno de ellos. A

partir de esta información se propuso una actividad en la plataforma SIE, de tal manera que el estudiante tenga una herramienta práctica con la que pueda comprobar la teoría. Cada práctica está compuesta por varias secciones como: introducción, fundamentos teóricos, procedimiento, resumen y preguntas. Por medio de estas secciones se busca que el estudiante aplique y refuerce conocimientos teóricos respecto al tema a tratar.

Estructura de las Prácticas.

La metodología que se utilizó para el diseño de las prácticas fue la siguiente:

- *Introducción.*

  Se hace una breve descripción del tema y la actividad que se va a realizar, dándole al estudiante una idea general de lo que está contenido dentro de la guía.

- *Objetivos.*

  Hace referencia a las metas que se deben llevar a cabo durante el proceso de la guía, de manera que al terminar de desarrollar la práctica se haya cumplido con cada uno de estos propósitos.

- *Marco teórico.*

  Se presentan los fundamentos básicos necesarios para llevar a cabo el desarrollo de la práctica. Esta información es la base para comprender la actividad propuesta en la guía.

- *Procedimiento general.*

  En esta sección se describe ordenadamente por medio de una serie de pasos el desarrollo de la actividad propuesta. Con el proceso descrito en esta parte se pretende cumplir con cada uno de los objetivos y ayudar al estudiante a comprender con detalle la actividad que se está desarrollando.

- *Resumen.*

  Se hace un breve resumen de lo que se hizo en la actividad, para que el estudiante

recuerde rápidamente el proceso que se llevó a cabo y así tener una idea clara de lo que se realizo.

- *Preguntas de prueba.*

  Esta sección es importante, ya que por medio de preguntas se puede comprobar si el estudiante cumplió con los objetivos requeridos en esta actividad.

- *Bibliografía.*

  En esta última sección se presentan las fuentes de referencia que se utilizaron para el diseño de la práctica. Es importante nombrarlas para que el estudiante pueda investigar más sobre el tema o despejar alguna duda que tenga sobre este.

Validación de las prácticas.

En este punto ya se tuvo una primera versión de las prácticas. Lo siguiente que se hizo fue someterlas a una validación con un grupo de estudiantes. Esto permitió obtener una valoración o juicio y así poder corregir posibles deficiencias de redacción y procedimentales. De acuerdo a los resultados arrojados en este paso se hicieron las modificaciones pertinentes.

Revisión.

Después de la validación se hizo una revisión de cada una de las prácticas con ayuda del grupo asesor del proyecto, donde se tienen en cuenta los procedimientos y la redacción que se utilizaron en cada una de ellas. Este paso arrojó las últimas modificaciones que se tuvieron en cuenta para la versión final de cada una de las guías.

Documentación final.

Esta última sección incluye la recopilación de toda la información, los resultados del proyecto, la presentación y edición final, para su respectiva entrega.

4. GUÍAS DE ARQUITECTURA DE COMPUTADORES. ESTRUCTURA Y FUNCIONAMIENTO DEL PROCESADOR

4.1 PROGRAMACIÓN BÁSICA EN LENGUAJE ENSAMBLADOR, ARQUITECTURA MIPS

El objetivo de esta guía es brindarle al estudiante una herramienta, con la cual se le facilite programar en lenguaje ensamblador. El lenguaje *assembler* usado en cada una de las prácticas está adaptado a la arquitectura MIPS. Con base en esta arquitectura se estudiará la estructura y el diseño de un programa en lenguaje ensamblador.

En esta guía se muestra en detalle el juego de instrucciones de la arquitectura MIPS, así como las técnicas de programación en lenguaje ensamblador. Aunque cada arquitectura tiene un juego de instrucciones propio, todas las arquitecturas incorporan una serie de instrucciones básicas con funcionalidades parecidas y estructura similar, pero esto no significa programar en otra arquitectura diferente.

El diseño de un programa en lenguaje ensamblador requiere algunos conocimientos previos entre ellos: la arquitectura del procesador con el que se va a trabajar, la estructura del programa, las secciones en que está dividida, el repertorio de instrucciones que maneja. La programación en lenguaje ensamblador ofrece diferentes ventajas, por ejemplo requiere considerablemente menos memoria y tiempo de ejecución que en un lenguaje de alto nivel.

4.2 DEPURACIÓN DE PROGRAMAS CON GDB

El diseño de un programa en cualquier lenguaje de programación es un proceso que debe terminar en la escritura de un código de programa que funcione correctamente.

Que un programa funcione correctamente significa que proporcione los resultados esperados, es decir, que su comportamiento real coincida con el comportamiento esperado. Un programa es correcto si no tiene errores de ejecución y cumple con su objetivo final.

En esta guía se utilizó el depurador GDB para ayudar en la solución de muchos problemas que se plantean al momento de verificar la correcta ejecución de un programa. Un depurador es una herramienta útil para todo programador, que permite observar cómo funciona un programa en desarrollo durante su ejecución. Muchas veces se presentan situaciones donde se tienen que analizar programas hechos por otras personas, sin tener el lenguaje fuente o la documentación adecuada. Aún si se tratase de un código propio, el uso de un depurador permite analizar programas que terminaron con un fallo e incluso procesos que ya se encuentran en ejecución, con la posibilidad de corregirlo, sin tener la necesidad de cambiar su código hasta que se tenga plena seguridad.

4.3 MODOS DE DIRECCIONAMIENTO

En esta práctica se trabajaron los principales modos de direccionamiento de la arquitectura MIPS. Recuerde que la arquitectura MIPS cuenta con un juego de instrucciones bastante amplio que abarca todo tipo de operaciones (aritmético, lógicas, salto, comparación, carga y almacenamiento), pero se distinguen por los diferentes tipos de formato. El modo de direccionamiento especifica la forma de interpretar la información contenida en cada campo del formato de instrucciones. Es así como el repertorio de instrucciones sirve como punto de partida para poder interpretar el funcionamiento de los modos de direccionamiento que se pueden presentar en una arquitectura MIPS.

Los modos de direccionamiento que ofrece la arquitectura MIPS son un mecanismo eficiente para conocer el funcionamiento interno del procesador. Al programar en *assembler* la técnica para acceder a los datos es tener en cuenta qué operaciones se deben realizar para obtener su dirección y si éstas pueden ser incluidas dentro una misma ins-

trucción como modo de direccionamiento. Por esto, los modos de direccionamiento son un recurso que el procesador ofrece para hacer más eficiente la ejecución de un programa.

Los modos de direccionamiento se refieren a la manera en que los operandos están especificados dentro del formato de instrucción. Dado que el formato de instrucción es de longitud limitada (32 bits), la codificación elegida impondrá unos límites en el número de direcciones de los operandos.

4.4 JERARQUÍA DE MEMORIA. RENDIMIENTO DE LA CACHÉ

En esta práctica se estudió uno de los temas más importantes que tiene la Arquitectura de computadores, la jerarquía de memoria, compuesta por: CPU, memoria caché, memoria principal y discos duros. Teniendo como objetivo comprender el rendimiento de una memoria de gran velocidad al costo de una memoria de baja velocidad.

Se analizó cada uno de los niveles básicos presentes en una jerarquía, enfatizando en el rendimiento de la caché. Todo esto con el fin de entender de manera teórica y práctica qué son las memorias, qué tipos de memorias hay y cómo es su interacción con el procesador.

La jerarquía de memoria es un sistema de administración de memorias. Es un concepto importante en el ámbito de la asignatura arquitectura de computadores. Se clarifican aspectos como la estructura y el funcionamiento de la jerarquía (CPU, memoria caché, memoria principal, discos duros), explicándose primero de manera teórica, y posteriormente se realiza un ejercicio práctico basado en programas que permiten observar el rendimiento de la memoria caché.

4.4.1. Análisis del Rendimiento de la Memoria Caché

Para analizar el rendimiento de la memoria caché y teniendo en cuenta su capacidad máxima de 4096 palabras, se ha diseñado un programa en C en el que por medio de un ciclo *while (b<N)* se llena el vector "vec[b]" por medio de la suma de las variables "a" y "b" (*vec[b] = a + b*), luego se guarda el valor de "vec[b]" en la variable "a" (*a = vec[b]*), después se suma el elemento del vector "vec[N-b]" con la variable "a" y se guarda en la variable "d" (*d = vec[N-b] + a)*, esta operación es la más importante en este programa ya que se está sumando un dato de las primeras posiciones con uno de las últimas posiciones del vector "vec". Por último, se aumenta la variable "b" en uno, repitiéndose este procedimiento "N" veces.

En el programa se utilizan las funciones *malloc()* y *gettimeofday()*. La primera es utilizada para reservar espacio en la memoria y la segunda se usa para medir el tiempo de ejecución del programa. Estas dos funciones se describen con mayor detalle en el manual de usuario.

El programa ejemplo que se utilizará para medir el rendimiento de la memoria caché se muestra en el *script* 4.2, se digita y se guarda en un archivo con extensión *.c*.

```
1  #include <stdio.h>              /*lib para la función printf*/
2  #include <stdlib.h>             /*lib para la función malloc()*/
3  #include <time.h>               /*lib para medir el tiempo*/
4  #include <sys/time.h>           /*lib que define el timeval para medir el tiempo*/
5  #define N 100                   /*tamaño del vector */
6  double timeval_diff(struct timeval *x, struct timeval *y)
7          {
8          return
9          (double)(x->tv_sec + (double)x->tv_usec/1000000) -
10         (double)(y->tv_sec + (double)y->tv_usec/1000000);   /*función para medir*/
11         }                                                   /*tiempo en [ms]*/
12 int main(int argc, char *argv[])
13         {                                        /*declaración de variables*/
14         struct timeval t_ini, t_fin;
```

```
15          double secs;
16          int d;
17          int a= 1;
18          int b = 1;
19          int *vec;                                   /*puntero para reservar memoria*/
20          vec=(int *) malloc(4*N*sizeof(int));        /*asignación de memoria*/
21                  gettimeofday(&t_ini, NULL);         /*inicio del cronometro*/
22                          while(b<=N)
23                                  {
24                                  vec[b]=b;
25                                  vec[b]=vec[N-b]+vec[b]; /*operación que exige la caché*/
26                                  b++;
27                                  }
28                  gettimeofday(&t_fin, NULL);             /*fin del cronometro*/
29          secs = timeval_diff(&t_fin, &t_ini);
30          printf("%.16g milliseconds\n", secs * 1000.0);  /*tiempo de ejecución*/
31          free(vec);                                      /*liberación de variables*/
32          return 0;
33          }
```

Script 4.2: Programa ejemplo

Por medio del programa del *script* 4.2 se analiza el rendimiento de la memoria caché a medida que se varía el tamaño de la variable “N”. Por medio de la función *gettimeofday* se mide y muestra el tiempo cada vez que se varia “N” con el fin de graficar los resultados.

4.4.1.1. Resultados

Cada vez que se cambia el valor de la variable “N” se obtiene un valor diferente de tiempo, obteniendo los resultados de la tabla 4.1, los cuales se grafican para observar como es el comportamiento del tiempo a medida que se aumenta el número de datos “N” en la caché.

La Figura 4.7 muestra como al aumentar “N” aumenta el tiempo hasta llegar a un valor donde ocurre un salto considerable. Este salto de tiempo se produce cuando “N” está en un intervalo entre 4000 a 4300. La razón por la cual se produce este salto de tiempo

Datos [N]	Tiempo [ms]
100	0.028
500	0.106
1000	0.203
1500	0.365
2000	0.485
2500	0.595
3000	0.706
3500	0.820
4000	0.936
4100	2.430
4200	2.470
4300	2.490
4500	2.510
5000	2.570
5500	2.725
6000	2.870
6500	2.985
7000	3.11

Tabla 4.1: Datos y tiempos

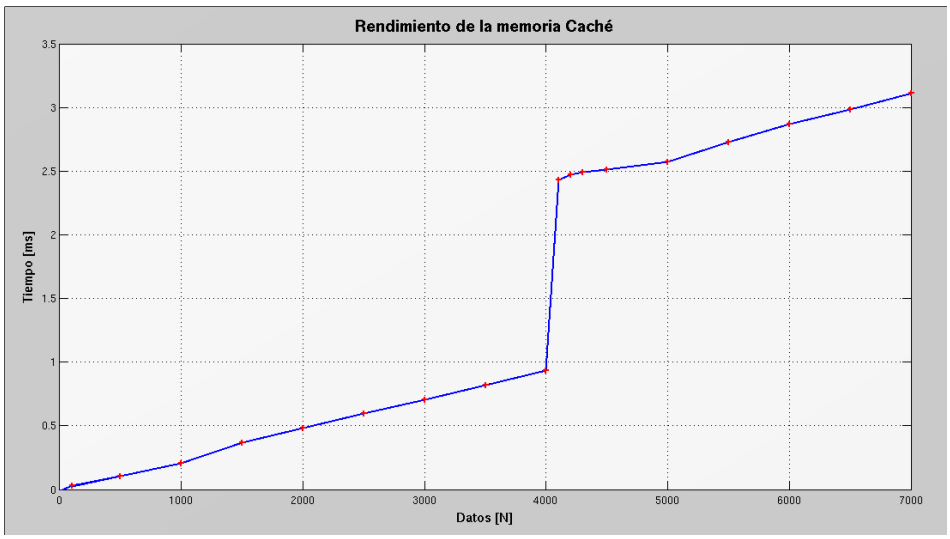

Figura 4.7: Eficiencia de la memoria caché.

FUENTE: El autor.

es debido a que la capacidad de datos de la caché es de 16 KB (4096 palabras) y cuando “N” supera este valor, quiere decir que se está desbordando la memoria caché y los datos que necesita la CPU se obtienen desde la memoria SDRAM, produciendo incremento de tiempo en la ejecución del programa.

5. VALIDACIÓN

Una de las fases más importantes para cumplir el objetivo en este trabajo es la validación. Este proyecto tiene como propósito general diseñar material de apoyo para la asignatura Arquitectura de Computadores. Debido a esto es imprescindible someterlo a un proceso de validación con lo que se busca obtener un material con una mejor estructura y nivel de entendimiento sin descuidar el ejercicio de lo aprendido, por lo cual también se someten a prueba las preguntas propuestas al final de la actividad. Con este proceso se quiere obtener un material de laboratorio que garantice al estudiante aplicar y reforzar los conocimientos adquiridos en la asignatura.

Una vez obtenida la primera versión de las prácticas, fue sometida a una validación con un grupo de estudiantes en la asignatura Sistemas Embebidos el segundo semestre de 2011. Se entregaron las guías para su respectivo desarrollo. En este procedimiento hubo varias inquietudes y sugerencias las cuales fueron la base para la retroalimentación de experiencias que dieron como resultado correcciones y mejoras en todos los aspectos de las guías, siendo el aporte más valioso para la consolidación de las mismas.

Durante el desarrollo del proceso anterior se observó un gran inconveniente que tuvo este grupo de estudiantes fue su poca experiencia con la plataforma SIE, debido a que es una nueva tarjeta que se empleará en los laboratorios del área de sistemas digitales. Lo anterior, no es un problema puesto que los estudiantes cuando lleguen a esta asignatura han tenido experiencia con esta plataforma en cursos anteriores de la rama. Otra de las dificultades presentadas fueron errores de tipo procedimental, debido a que en ocasiones el procedimiento no era claro. Estas últimas dificultades dejaron ver varios puntos donde habían falencias que sirvieron de aporte en cuanto a la versión final de las prácticas.

Otros de los inconvenientes que se presentó fue en la configuración de la plataforma SIE para trabajar en algunas de las guías que necesitaban la adición de librerías, por lo que fue necesario incluir en el manual de usuario el proceso de instalación y configuración de herramientas para trabajar con la tarjeta SIE. Por último, como resultado del proceso de validación se obtuvo una versión más amigable, entendible y robusta de cada una de las prácticas, las cuales fueron revisadas por los codirectores del proyecto.

6. CONCLUSIONES Y RECOMENDACIONES

La plataforma SIE es un sistema de desarrollo de *software* y *hardware* libre compuesto por varios dispositivos. Este se puede modificar permitiendo la creación de más módulos, ya que se dispone de información necesaria para esto, con el único requisito de dar crédito al autor original. También se puede compartir información que pueda ser de ayuda para otros desarrolladores. Debido a esto se implementa esta plataforma no solo en la asignatura Arquitectura de Computadores sino en todos los cursos del área de sistemas digitales como herramienta de aprendizaje.

La plataforma SIE trabaja con un procesador de aplicaciones multimedia JZ4725, basado en la arquitectura MIPS. En este trabajo se utiliza el procesador de manera didáctica donde se implementa un conjunto de prácticas relacionadas con los principales temas de la asignatura Arquitectura de Computadores. Con esto se proporciona una herramienta en donde los estudiantes aplican los conocimientos adquiridos en un sistema *hardware*.

Debido a que la tarjeta contiene un procesador que incluye componentes básicos de la arquitectura de un computador se puede utilizar para el diseño de programas en lenguaje ensamblador, conociendo la arquitectura, el repertorio de instrucciones y la estructura de un programa en *assembler*. Por medio de esta programación se llega a tener una idea de como funciona un procesador internamente y de como es su estructura básica. Por otro lado se presenta una guía de como diseñar programas en *assembler* de manera sencilla y desmentir la idea de que la programación en este lenguaje es difícil.

Una de las ventajas que tiene la plataforma SIE es que se pueden agregar librerías para trabajar con aplicaciones como por ejemplo: depuradores, compiladores, herramientas para desarrollar interfaces gráficas, etc. Para este caso se configuraron las librerías para

trabajar con el depurador GDB. Es así como por medio de esta plataforma se pueden analizar programas diseñados en *assembler*, C o C++. De esta manera el estudiante tiene una herramienta muy útil a la hora de diseñar y depurar programas.

Los modos de direccionamiento son un mecanismo eficiente para conocer el funcionamiento interno del procesador. Por medio de estos se pueden identificar los componentes que permiten las operaciones básicas entre registros, las operaciones de carga/almacenamiento y los saltos.

Por medio de este trabajo y utilizando la plataforma SIE el estudiante tendrá una idea clara respecto a la estructura y el funcionamiento interno del procesador, ya que se analizan temas como el repertorio de instrucciones, las instrucciones de máquina, el banco de registros, los modos de direccionamiento, el camino de datos y la jerarquía de memoria.

El proceso más importante en este proyecto fue el de validación llevado a cabo por un grupo de estudiantes. Este arrojó falencias en cada una de las prácticas, se observó que muchas de las dificultades presentadas fueron errores de tipo procedimental, debido a que en ocasiones el procedimiento no era claro. Estas dificultades se tuvieron en cuenta y sirvieron como punto de partida para hacer una retroalimentación para la respectiva modificación teniendo como prioridad los aportes que surgieron del proceso de validación.

Es importante diseñar una metodología clara, ya que es el camino que conduce al desarrollo ordenado del proyecto y esto conlleva a tener una mejor distribución de tiempo. Este procedimiento sirvió de instrumento para lograr cada uno de los objetivos propuestos de una forma estructurada.

Este trabajo queda atento a posibles modificaciones en pro del mejoramiento y ajuste

de la asignatura Arquitectura de Computadores en la medida que sea conveniente. Por este motivo se deja a disposición del interesado toda la documentación y referencias bibliográficas utilizadas en este proyecto, con el objetivo de que el usuario final obtenga el mejor material acorde a la asignatura y a los avances tecnológicos que la afecten.

Bibliografía

[1] Camargo, Carlos Ivan. *SIE: Plataforma Hardware copyleft para la ensenanza de sistemas digitales*. Universidad Nacional de Colombia. 2010. 6 p.

[2] Chu, Pong P. *FPGA PROTOTYPING BY VHDL EXAMPLES , 3rd Edition*. Jhon Wiley & Sons, INC, publication. 2008.

[3] Datasheet JZ4725, *multimedia Application processor*, Ingenic Semiconductor Co. Ltd, Revision: 1.0,May 2009

[4] Heinrich, Joe: *Mips R4000 Microprocessor User's Manual*, (2004).

[5] Página linux en caja. `http://linuxencaja.net/wiki/SIE/es`.

[6] Página qi-hardware. `http://en.qi-hardware.com/wiki/SIE`.

[7] Ochoa Blanco, Gustavo Adolfo. Mendez Florez, Neder Jair. *Desarrollo de material didáctico para el área de sistemas digitales basados en la plataforma SIE. Asignatura: arquitectura de computadores*. UNIVERSIDAD INDUSTRIAL DE SANTANDER. Bucaramanga, 2012.

[8] Patteeson, David A. Hennessy, John L. *Organizacion y Diseno de Computadores. La interfaz hardware/software*. McGRAW-HILL. 2003.

[9] Rodríguez Fabregues, Franco: *GDB GNU Debugger*, JCórdoba, ARGENTINA, Universidad Nacional de Córdoba, (2011).

[10] Stallings, Williams: *Computer Organization Architecture*, Pearson Education International, New Jersey, USA, (2003),0-13-049307-4.

[11] Xilinx Inc. *Xilinx ISE 10.1 Design Suite Software Manuals and Help - PDF Collection*. 2008.

ANEXO A. MANUAL DE GUIAS DE LABORATORIO ARQUITECTURA DE COMPUTADORES

LABORATORIO 1

PROGRAMACIÓN BÁSICA EN LENGUAJE ENSAMBLADOR, ARQUITECTURA MIPS

"La imaginación es más importante que el conocimiento. El conocimiento es limitado, mientras que la imaginación no"

ALBERT EINSTEIN

1 INTRODUCCIÓN

El objetivo de esta guía es brindarle al estudiante una herramienta, con la cual se le facilite programar en *assembler* o lenguaje ensamblador. Este lenguaje está basado en la arquitectura MIPS. Los ejemplos se refieren al compilador gcc *(GNU compiler collection)*, y se utilizan para ayudar al estudiante en la estructura y diseño de un programa en *assembler*.

Se mostrará en detalle el juego de instrucciones de la arquitectura MIPS, así como las técnicas de programación en ensamblador en dicha arquitectura. Aunque cada arquitectura tiene un juego de instrucciones propio, todas las arquitecturas incorporan una serie de instrucciones básicas con funcionalidades muy parecidas, no será fácil programar en otra arquitectura pero si ya se tiene experiencia en una, de ellas se facilitará el trabajo.

2 OBJETIVOS

- ⋆ Conocer las características básicas de la arquitectura del procesador MIPS.
- ⋆ Comprender la estructura y organización de un programa, como también el inicio y finalización del código.
- ⋆ Distinguir y aplicar apropiadamente la sintaxis e instrucciones básicas.

3 MARCO TEÓRICO

3.1 ESTRUCTURA INTERNA DEL COMPUTADOR

El computador se compone de varios bloques funcionales: el procesador, la memoria, los dispositivos de entrada/salida y los buses. El procesador se encarga del control y ejecución de los programas; los programas y los datos se guardan en memoria; la comunicación con el usuario se logra a través de los dispositivos de entrada/salida y los buses son los encargados de transferir la información entre todos los componentes del sistema antes mencionados. En la figura 1.1 se puede ver una ilustración de la conexión básica de estos bloques.

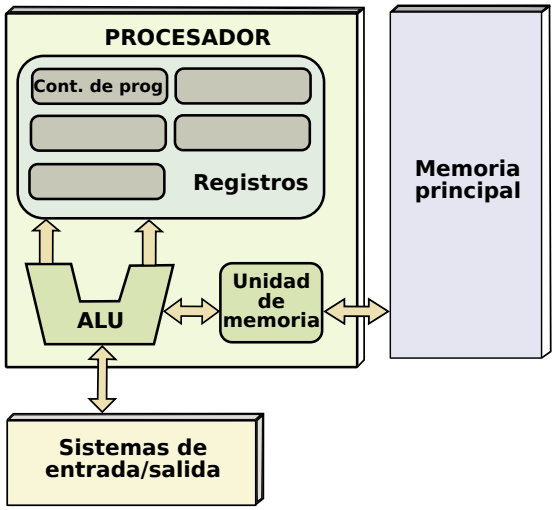

Figura 1.1: Diagrama de bloques de un computador.
FUENTE: El autor.

La programación en lenguaje ensamblador depende de la arquitectura del computador sobre el cual se está trabajando; ésta define el funcionamiento interno de la máquina, por ello al programar en ensamblador se llega a tener una mejor comprensión de cómo es dicho funcionamiento y la estructura básica de un computador [1].

3.2 SET DE INSTRUCCIONES

El repertorio de instrucciones (ISA o *Instruction Set Architecture*) de un procesador es el conjunto de instrucciones que éste puede ejecutar.

Mediante el ISA de un computador es posible conocer características de la máquina tales como el número y tipo de registros, modos de direccionamiento, uso de interrupciones y manejo de dispositivos de entrada/salida. En general, los repertorios de instrucciones de los distintos modelos de procesadores son bastante similares. Esto se debe a que su *hardware* se diseña usando principios fundamentales similares y por lo tanto, existe un repertorio básico de funciones que todos los computadores deben proporcionar. Asimismo, un ISA puede ser usado incluso por computadores diseñados por diferentes fabricantes. Por ejemplo, el ISA del IA-32 (también conocido como x86) es utilizado por procesadores fabricados por Intel y AMD, entre otros [2].

4 PROCEDIMIENTO GENERAL

El procedimiento a seguir con el fin de diseñar un programa en lenguaje ensamblador y ejecutarlo en la tarjeta SIE es el siguiente:

1. Estudiar las características de la máquina con la que se va a trabajar.
2. Analizar los registros de un procesador MIPS (JZ4725) y la función de cada uno de ellos dentro de un programa.
3. Descripción de la organización y estructura de un programa.
4. Descripción de las instrucciones básicas de la arquitectura MIPS.

4.1 CARACTERÍSTICAS DEL PROCESADOR

En el desarrollo de este curso se trabajará con el procesador de aplicaciones multimedia JZ4725. Éste lleva incorporado en su núcleo un procesador Xburst de 32 bits, basado en la arquitectura MIPS, contiene 32 registros, memoria caché de un tamaño de 16 KiB para datos y 16 KiB para instrucciones, llegando a manejar una velocidad de hasta 360 MHz.

4.2 REGISTROS MIPS

Se sabe que el procesador posee una arquitectura MIPS y tiene 32 registros de 32 bits, por lo tanto se necesitan 5 bits para especificar la dirección de un registro ($2^5 = 32$). La tabla 1.1 ilustra los nombres simbólicos y numéricos de los registros, además de una norma para usarlos, ya que se debe tener en cuenta la función de cada registro antes de intentar utilizarlo [3].

Registro	Número	Uso
zero	0	Constante 0
at	1	Reservado para el ensamblador
v0,v1	2,3	Evaluación de expresiones y resultados de funciones
a0-a3	4-7	Argumentos
t0-t7	8-16	Temporal (no se guarda el valor entre llamadas)
s0-s7	16-23	Temporal (el valor se guarda entre llamadas)
t8,t9	24,25	Temporal (no se guarda el valor entre llamadas)
k0,k1	26,27	Reservado para el kernel del sistema operativo
gp	28	Puntero del área global
sp	29	Puntero de pila
fp	30	Puntero de marco de pila
ra	31	Dirección de retorno, usadas por llamadas de función

Tabla 1.1: Nombre y uso de registros del procesador MIPS

- El registro *zero* contiene siempre la constante 0. Se puede utilizar para limpiar un registro.
- Los registros *$at* (1), *$k0* (26) y *$k1* (27) están reservados por el ensamblador y no deben ser usados.
- Los registros *$a0* - *$a3* (4-7) se usan para pasar los primeros cuatro argumentos de funciones.
- Los registros *$v0* y *$v1* (2 y 3) se usan para regresar valores de funciones después de un llamado a sistema.
- Los registros *$t0* - *$t7*, *$t8* y *$t9* (8-15, 24 y 25) se utilizan para almacenar valores temporales. Por tanto pueden ser usados en cualquier función sin preocuparse de guardar su valor anterior.
- Los registros *$s0* - *$s7* (16-23) se utilizan para almacenar variables que deben ser preservadas entre llamadas a funciones. Por tanto, si alguna función necesita usar alguno de estos registros, ha de guardar antes su valor (por ejemplo en la pila). De esta forma, la función que usa estos registros no ha de preocuparse de guardar sus valores antes de llamar a otra función.
- El registro *$gp* (28) es un apuntador global que apunta a la mitad de un bloque de memoria en el segmento de datos estáticos.
- El registro *$fp* (30) *frame pointer* contiene la dirección de la zona de la pila en la que están guardados los argumentos y las variables locales de la función que no caben en los registros (*$a0* - *$a4*). Esta zona se conoce como bloque de activación. Su utilidad es la de simplificar el acceso a estas variables cuando es necesario modificar la pila durante la ejecución de la función (por ejemplo durante la evaluación de expresiones).
- El registro *$sp* (29) *stack pointer* es el que apunta al tope de la pila.

- El registro *$ra* (31) es utilizado para almacenar la dirección de retorno de una llamada a procedimiento.

4.3 ORGANIZACIÓN Y ESTRUCTURA DEL PROGRAMA

Un programa diseñado en lenguaje ensamblador basado en MIPS lleva la extención [*.s*] y utiliza algunas palabras como *.data, .globl main, .asciiz, etc.* Este tipo de palabras que comienzan en punto (".") se llaman directivas del ensamblador (o seudocódigos de operación) y sirven para decirle al ensamblador como traducir un programa, pero no producen instrucciones de máquina.

Un programa está dividido en dos secciones: una de datos y otra de texto. Las directivas *.data* y *.text* marcan el inicio de cada sección respectivamente [4].

4.3.1 Segmento de datos

Se utiliza para comunicarle al ensamblador que todo lo que aparezca en él (mientras no se diga lo contrario) debe ser almacenado en la zona de datos. Cuando se utiliza la directiva *.data* se puede dar enseguida la dirección donde se quiere empezar a guardar los datos, si no se le asigna dirección el procesador toma una por defecto. Adicionalmente corresponde a aquellos datos cuyo valor puede cambiar, pero que en ningún caso cambian de ubicación en la memoria o son desalojados de ésta. El tamaño del segmento de datos, por tanto, permanece fijo durante toda la ejecución del programa. En la tabla 1.2 se encuentran las directivas para datos más utilizadas por la arquitectura MIPS, junto con su función y un ejemplo.

Directiva	**Función**	**Ejemplo**
.asciiz	Indica que lo que sigue es un dato de tipo cadena	cad: .asciiz "El resultado es: "
.ascii	Indica que lo que sigue es un caracter	cad: .ascii a, b, c, d
.byte	Es un conjunto de uno o más números enteros de 8 bits	ent: .byte 0,2,8,-5,199
.half	Media palabra, declara valores enteros de 2 Bytes	ent: .half 2,-5,961,-961
.word	Palabra, declara valores enteros de 4 Bytes	ent: .word 366010, 1, 23956
.space	Reserva como espacio de memoria en bytes	espacio: .space 4

Tabla 1.2: Tipos de datos usados por mips

4.3.2 Segmento de texto

También conocido como segmento de código, la directiva *.text* es la parte del programa correspondiente a instrucciones máquina. Indica el comienzo de la zona de memoria dedicada a las instrucciones. Su tamaño para cada programa es fijo durante todo el tiempo que dura la ejecución de éste. Al inicio

de este segmento se debe incluir la directiva *.globl main* para declarar este procedimiento como el principal y poder ser referenciado desde otros archivos. En el *script* 1.1 se muestra un ejemplo de esta directiva.

```
1          .text
2          .globl main
3  main:           li $t0, 4              #carga 4 en el registro $t0
4                  .
```

Script 1.1: Ejemplo del segmento .text

4.3.3 Etiquetas

Una *etiqueta* es una cadena de caracteres que acaba con dos puntos (":"). Las etiquetas pueden aplicarse tanto a instrucciones como a datos, pero en cualquier caso siempre hacen referencia a la dirección de memoria en la que comienza dicha instrucción o dato.

Una etiqueta es una dirección de memoria, pero no toma un valor hasta que el programa es compilado o ensamblado. Durante la ejecución de programa es posible que las etiquetas cambien de valor, si bien sus posiciones relativas en memoria seguirán siendo las mismas.

Las direcciones de memoria siempre son números en bytes, suponiendo que el primer byte de la memoria es el 0, el segundo es 1, el tercero es 2 y así sucesivamente. Esto se debe tener en cuenta a la hora de buscar una dirección.

4.3.4 Comentarios

Los comentarios sirven para dejar por escrito lo que se está haciendo en cada línea del programa y para mejorar su legibilidad remarcando las partes que lo forman. Comentar es importante para un programa escrito en un lenguaje de alto nivel y mucho más para programas diseñados en lenguaje ensamblador. El comienzo de un comentario se indica por medio del carácter #, el compilador ignora el resto de la línea a partir de este símbolo.

4.4 SET DE INSTRUCCIONES MIPS

El propósito de la práctica es estudiar el repertorio de instrucciones y las convenciones de programación de un procesador basado en la arquitectura MIPS. El set de instrucciones es usado en varios sistemas

reales y representativo de las arquitecturas tipo RISC (Reduced Instruction Set Computer). Como tal se caracteriza por ofrecer instrucciones sencillas, formatos de instrucción regulares, muchos registros de propósito general y modos de direccionamiento sencillos.

En la siguiente descripción se clasifican las instrucciones MIPS según el tipo de acción que realizan y se mostrarán ejemplos de cada tipo de instrucciones, posibilitando así comenzar a programar con el repertorio básico a disposición. Se presentarán las instrucciones básicas MIPS destacando la función de cada una de ellas.

4.4.1 Operaciones aritméticas

Los primeros tipos de operación y quizás los más usados, puesto que son los que realizan un verdadero procesamiento de los datos, son llevados a cabo por las instrucciones aritméticas. Los computadores han de ser capaces de realizar sumas, restas, multiplicaciones y divisiones con los datos. Las instrucciones aritméticas básicas son presentadas en la tabla 1.3 [5].

Instrucción	Ejemplo	Acción
add	add $t0, $t1, $t2	Suma $t1 y $t2, y lo coloca en $t0
addi	addi $t0, $t1, kte	Suma $t1 y el valor kte y lo coloca en $t0
sub	sub $t0, $t1, $t2	Resta $t1 y $t2, y lo coloca el resultado en $t0
mul	mul $t0, $t1, $t2	Coloca el producto de $t1 y $t2, en el registro $t0
div	div $t0, $t1, $t2	Divide $t1 entre $t2, y lo coloca el resultado en $t0

Tabla 1.3: Instrucciones aritméticas básicas

4.4.2 Operaciones lógicas

El set de instrucciones de la arquitectura MIPS permite realizar operaciones lógicas del tipo AND, OR, XOR, NOT. Se debe tener en cuenta que las operaciones lógicas, aunque toman como operandos los contenidos de dos registros (palabras), operan bit a bit. Por ejemplo, si se realiza una operación AND entre dos palabras genera como resultado una palabra en la que el bit 0 es la operación AND de los bits 0 de los operandos fuente, el bit 1 es la operación AND de los bits 1 de los operandos fuente, y así sucesivamente. En la tabla 1.4 se presentan instrucciones lógicas más usadas del set de instrucciones MIPS [5].

4.4.3 Operaciones de carga y almacenamiento

Como se ha visto, la mayoría de las instrucciones MIPS operan sólo con registros y constantes. Sin embargo, en el banco de registros caben muy pocos datos, mientras que los programas actuales nece-

Instrucción	Ejemplo	Acción
and	and $t0, $t1, $t2	Realiza un and lógico entre los contenidos de $t1 y $t2, y lo guarda en $t0
andi	andi $t0, $t1, kte	Realiza un and lógico entre los contenidos de $t1 y kte, y lo guarda en $t0
or	or $t0, $t1, $t2	Realiza un or lógico entre los contenidos de $t1 y $t2, y lo guarda en $t0
ori	ori $t0, $t1, kte	Realiza un or lógico entre los contenidos de $t1 y kte, y lo guarda en $t0
not	not $t0, $t1	Realiza un not lógico del contenido de $t1, y lo guarda en $t0
nor	nor $t0, $t1, $t2	Realiza un nor lógico entre los contenidos de $t1 y $t2, y lo guarda en $t0
xor	xor $t0, $t1, $t2	Realiza un xor lógico entre los contenidos de $t1 y $t2, y lo guarda en $t0
xori	xori $t0, $t1, kte	Realiza un xor lógico entre los contenidos de $t1 y kte, y lo guarda en $t0

Tabla 1.4: Instrucciones lógicas básicas

sitan manejar grandes cantidades de información, por lo tanto se debe almacenar en la memoria, la cual tiene más capacidad. Por ello, para trabajar con esa información es necesario mover primero los datos de la memoria a los registros y después volver a mover los datos de los registros a la memoria. Estas transferencias de información entre registros y memoria se realizan mediante las instrucciones de acceso a memoria. Estas instrucciones se encargan de traer y llevar datos [5].

Hay dos tipos de instrucciones de transferencia de datos en memoria: las de carga (*load*), que copia en un registro un dato que se encuentra en una determinada dirección de memoria; y las de almacenamiento (*store*), que copian en una determinada dirección de memoria un dato que se encuentra en un registro. Estas y otras instrucciones de carga y almacenamiento de datos son presentadas en la tabla 1.5.

También en esta sección se incluye una instrucción muy utilizada para transferir datos de un registro a otro. Esta es la instrucción *move* ubicada en la parte final de la tabla 1.5.

Instrucción	Ejemplo	Acción
la	la $t0, dir	Carga en $t0 la dirección dir (no su contenido)
li	li $t0, inm	Carga valor inm en el registro $t0.
lb	lb $t0, dir	Carga en $t0 el byte en memoria direccionado por dir
lw	lw $t0, dir	Carga en $t0 la palabra direccionada por dir
sb	sb $t0, dir	Almacena el byte menos significativo de $t0 en la posición dada por dir
sw	sw $t0, dir	Almacena la palabra del registro $t0 en la posición de memoria dada por dir
move	move $t0, $t1	Copia el contenido del registro $t1 y lo coloca en el registro $t0

Tabla 1.5: Instrucciones de carga/almacenamiento básicas

4.4.4 Operaciones de comparación

Son instrucciones que sirven para comparar los valores de dos registros fuente y, dependiendo del resultado de la comparación, se pone a 1 el registro destino. Las operaciones más comunes se pueden ver en la tabla 1.6 [6].

Instrucción	Ejemplo	Acción
seq	seq $t0, $t1, $t2	$t0 = 1 si $t1 = $t2 de lo contrario $t0 = 0
sge	sge $t0, $t1, $t2	$t0 = 1 si $t1 >= $t2 de lo contrario $t0 = 0
sgt	sgt $t0, $t1, $t2	$t0 = 1 si $t1 > $t2 de lo contrario $t0 = 0
sle	sle $t0, $t1, $t2	$t0 = 1 si $t1 <= $t2 de lo contrario $t0 = 0
slt	slt $t0, $t1, $t2	$t0 = 1 si $t1 < $t2 de lo contrario $t0 = 0
sne	sne $t0, $t1, $t2	$t0 = 1 si $t1 es diferente a $t2 de lo contrario $t0 = 0

Tabla 1.6: Instrucciones básicas de comparación

4.4.5 Operaciones de salto

Hasta ahora se han visto instrucciones donde el procesador realiza una ejecución secuencial, es decir, ejecuta una a una cada instrucción hasta llegar al final del programa. Sin embargo, esta forma de ejecución tiene bastantes limitaciones. Un programa de ejecución secuencial no podría, tomar diferentes acciones en función de los datos de entrada, de los resultados obtenidos o de la interacción con el usuario. Tampoco puede realizar un número n de veces ciertas operaciones, a no ser que se escriba n veces las mismas operaciones; y en este caso, n debería de ser un número predeterminado y no podrá variar en sucesivas ejecuciones.

La ventaja es que en un programa se puede tomar diferentes acciones y puede repetir un conjunto de instrucciones cierto número de veces, esto se puede conseguir con las instrucciones de salto condicionales y comparaciones. En la tabla 1.7 se puede observar las instrucciones más comúnmente utilizadas [6] [2].

Instrucción	Ejemplo	Acción
beq	beq $t0, $t1, salt	Salta a la etiqueta salt si $t0 = $t1 sino ejecuta la siguiente instrucción
beqz	beqz $t0, salt	Salta a la etiqueta salt si $t0 = 0 sino ejecuta la siguiente instrucción
bge	bge $t0, $t1, salt	Salta a la etiqueta salt si $t0 >= $t1 sino ejecuta la siguiente instrucción
bgez	bgez $t0, salt	Salta a la etiqueta salt si $t0 >= 0 sino ejecuta la siguiente instrucción
bgt	bgt $t0, $t1, salt	Salta a la etiqueta salt si $t0 > $t1 sino ejecuta la siguiente instrucción
bgtz	bgtz $t0, salt	Salta a la etiqueta salt si $t0 > 0 sino ejecuta la siguiente instrucción
ble	ble $t0, $t1, salt	Salta a la etiqueta salt si $t0 <= $t1 sino ejecuta la siguiente instrucción
blez	blez $t0, salt	Salta a la etiqueta salt si $t0 <= 0 sino ejecuta la siguiente instrucción
blt	blt $t0, $t1, salt	Salta a la etiqueta salt si $t0 < $t1 sino ejecuta la siguiente instrucción
bltz	bltz $t0, salt	Salta a la etiqueta salt si $t0 < 0 sino ejecuta la siguiente instrucción
bne	bne $t0, $t1, salt	Salta a la etiqueta salt si $t0 es $\neq$ a $t1 sino ejecuta la siguiente instrucción
j	j salt	Salto incondicional a la etiqueta salt
ja	ja $t0	Salta a la instrucción cuya dirección esta guardada en el registro $t0
jal	jal dir	Salta a la instrucción etiquetada dir y guarda la dirección siguiente en $ra

Tabla 1.7: Instrucciones básicas de salto

4.4.6 Proceso de impresión de datos

Para este fin se utilizará la función *printf*. Esta función permite escribir en consola datos como números, caracteres y cadenas. Recibiendo una cantidad de argumentos de diferente formato de impresión.

La función *printf* puede imprimir caracteres alfanuméricos, los cuales se mostrarán en pantalla exactamente como se encuentran en los datos a imprimir. Se debe utilizar un tipo de caracteres que hacen parte de la sintaxis y se conocen como especificadores de formato, los cuales están formados por un símbolo de porcentaje (%) seguido de un carácter. Los especificadores de formato compatibles con la función *printf* se describen en la tabla 1.8.

Código	Función
%c	Carácter simple
%d	Entero con signo en base decimal
%i	Entero con signo en base decimal (integer)
%e	Notación científica (letra 'e' minuscúla)
%E	Real en notación científica (letra 'E' mayuscúla)
%f	Real (punto flotante)
%g	Equivalente a la representación real más corta (%e o %f)
%G	Equivalente a la representación real más corta (%E o %f)
%o	Entero con signo en base octal
%s	Cadena de caracteres
%u	Entero sin signo en base decimal
%x	Entero con signo en base hexadecimal

Tabla 1.8: Códigos para impresión

Para imprimir un mensaje o resultado se deben utilizar los registros asignados para este fin. Los registros *$a0 - $a3* son utilizados para guardar los datos que se van a imprimir, en el registro *$a0* se guarda la dirección donde empieza el mensaje a imprimir y en el registro $a1 - $a3 se guarda los resultados numéricos que se quieren mostrar en consola. Los resultados numéricos se imprimen en orden, es decir, primero se imprime lo que esta en el registro *$a1* seguido de *$a2* y por último *$a3*, sin importar si están en desorden en el programa. Un ejemplo de un código para impresión se muestra en el *script* 1.2.

```
1 impr:   la      $a0, mensaje    # guarda la direccion del primer carácter en $a0
2         move    $a1, $t0        # guarda en $a1 el dato que desee imprimir
3         j       printf          # imprime mensaje y dato
4         nop                     # fin del programa
```

Script 1.2: Ejemplo del código para impresión

5 RESUMEN

El diseño de un programa en lenguaje ensamblador requiere algunos conocimientos previos como la arquitectura del procesador con el que se va a trabajar, la estructura del programa, las secciones en que está dividido, el repertorio de instrucciones que maneja, etc. La programación en lenguaje ensamblador ofrece diferentes ventajas: un programa escrito en lenguaje ensamblador requiere considerablemente menos memoria y tiempo de ejecución que un programa escrito en un lenguaje de alto nivel entre otros, siempre y cuando este bien diseñado. La programación en lenguaje ensamblador depende de la arquitectura del procesador sobre el cual se trabaja, esto es importante para entender el funcionamiento interno de la máquina, por ello al programar en ensamblador se llega a comprender cómo funciona el computador y cómo es su estructura básica. La capacidad de poder escribir programas en lenguaje ensamblador es muy importante para los profesionales del área de Sistemas Operativos debido a que los programas residentes y rutinas de servicio de interrupción casi siempre son desarrollados en lenguaje ensamblador.

La organización de esta guía corresponde al proceso de aprendizaje de la programación en lenguaje ensamblador. Primero se describen las características básicas de la arquitectura del procesador sobre la cual se va a trabajar, luego el entorno de programación, es decir, las secciones de los programas y herramientas necesarias para realizar la programación como set de instrucciones, directivas, etiquetas y comentarios. Esta guía es una introducción para que el estudiante use estas herramientas y este en la capacidad de diseñar un programa en lenguaje ensamblador básico y posteriormente se irá aprendiendo nuevos conceptos que proveerán de información necesaria para la detección y corrección de errores del programa escrito en lenguaje ensamblador. También se describe la estructura de un programa, cómo declarar datos y cómo utilizar las instrucciones básicas. Luego se explica la programación de estructuras condicionales y saltos. Hasta este punto se cubren los conceptos básicos de la programación en lenguaje ensamblador.

6 TRABAJO DE LABORATORIO

Una vez leído este documento el estudiante estará en la capacidad de diseñar un programa en lenguaje *assembler*. En este espacio y de manera progresiva se aplicarán los conocimientos adquiridos en esta guía.

1. Pensar en un programa a diseñar, como por ejemplo, sumar dos vectores, multiplicar dos números,

hacer una operación donde utilice varias operaciones, etc, el cual se ira armando progresivamente. El programa se deja abierto a la iniciativa de cada uno de los estudiantes, teniendo en cuenta el contenido de la guía.

2. Abrir un archivo de texto colocándole el nombre del programa y algo que no se puede olvidar es la extensión la cual debe ser *.s*, esto es muy importante a la hora de compilar.

3. Para empezar se diseñará primero la sección de datos, donde se declarará todas las variables que se necesiten según el programa. No olviden comentar cada linea de aquí en adelante.

4. Lo siguiente que se hará es diseñar la sección de texto, aquí es donde va el código del programa. En esta parte se debe tener en cuenta el encabezado de esta sección.

5. Dentro de la sección de texto es donde se utilizará el repertorio de instrucciones, se recomienda tener cuidado con la secuencia y la lógica de cada instrucción que se utilice. Otro punto importante es tener buen uso de los registros ya que algunos tienen diferentes funciones y otros están reservados por el sistema, ver tabla 1.1.

6. En la parte final es necesario imprimir en pantalla la respuesta del programa, no importa el tipo de respuesta, esto ayuda a comprobar si el programa cumple su objetivo a la hora de ejecutarlo. Para esto se debe tener muy claro el proceso de impresión de datos ya sea mensajes o resultados, ver tabla 1.8.

7. Por último se procederá a compilar el programa, crear el ejecutable, enviar el ejecutable a la tarjeta SIE y ejecutarlo. Este procedimiento está descrito en el manual de usuario.

Bibliografía

[1] John L. Hennessy, David A. Patteeson: *Organización y Diseño de Computadores. La interfaz hardware/software*, Madrid, ESPAÑA, 2 edición, McGRAW-HILL, (2003), 1-55860-281-X.

[2] Sergio Barrachina Mir, Maribel Castillo Catalán, José M. Claver Iborra y Juan Carlos Fernández Fernández: *Practicas de Introduccion a la Arquitectura de Computadores con el simulador SPIM*, (2007).

[3] José Daniel Muñoz Frías: *Estructura de Computadores*, Madrid, ESPAÑA, Universidad Pontificia Comillas, Madrid, (2003).

[4] Joe Heinrich: *Mips R4000 Microprocessor User's Manual*, (2004).

[5] Leopoldo Silva Bijit: *Estructuras de Computadores Digitales*, Santa Maria, CHILE, (2008).

[6] Juan A. Gómez Pulido: *Repertorio de Instrucciones MIPS*, Universidad de Extremadura, (2003).

LABORATORIO 2

DEPURACIÓN DE PROGRAMAS CON GDB

"Los programas deben ser escritos para que los lean las personas, y sólo incidentalmente, para que lo ejecuten las máquinas"

ABELSON AND SUSSMAN

1 INTRODUCCIÓN

El diseño de un programa en cualquier lenguaje de programación es un proceso más o menos largo que debe terminar en la escritura de un código del programa que funcione correctamente. Que un programa funcione correctamente significa que proporcione los resultados esperados, es decir, que su comportamiento real coincida con el comportamiento esperado. Un programa es correcto si no tiene errores de sintaxis y cumple con su objetivo final.

En esta guía se utilizará un depurador para que facilite la detección y solución de forma sencilla de muchos de los problemas que se plantean al momento de verificar la correcta ejecución de un programa. Se trabajará con el depurador GDB de GNU y se enfocará en la arquitectura MIPS.

2 OBJETIVOS

- ⋆ Conocer la función que realiza el depurador GDB.
- ⋆ Estudiar las diferentes herramientas de análisis que proporciona GDB.
- ⋆ Aplicar las herramientas de GDB en un programa ejemplo.
- ⋆ Solucionar posibles errores que se presentan en un programa diseñado en lenguaje ensamblador.

3 MARCO TEÓRICO

3.1 DEPURADOR GDB

Un depurador es una aplicación que permite correr un programa, posibilitando observar cómo es su funcionamiento y examinar el estado de algunas variables, registros y etiquetas utilizadas. El propósito final de un depurador consiste en permitir observar y comprender lo que está sucediendo dentro de un programa mientras es ejecutado, es así como se puede detectar los posibles errores junto con su ubicación dentro de un programa.

En los sistemas operativos UNIX/LINUX el depurador más utilizado es GDB, es decir, el depurador de GNU. Éste ofrece una cantidad muy extensa y especializada de herramientas. Es muy importante entender el hecho de que un depurador trabaja sobre archivos ejecutables. Esto quiere decir que el mismo funciona de forma independiente al lenguaje en que se escribió el programa original, sea éste lenguaje ensamblador o un lenguaje de medio o alto nivel como C [1].

3.2 CLASES DE ERRORES

En el proceso de escritura de un programa aparecen muchos errores que se deben ir corrigiendo hasta llegar a escribir el programa correctamente. Es común que se presenten errores, ya que no se tiene mucha experiencia en este tipo de programación y es importante saber de que trata cada tipo error [2].

Los tipos errores que se pueden encontrar son:

- **Errores de sintaxis**: Son errores producidos por el incumplimiento de las reglas sintácticas del lenguaje que se este utilizando. Hasta que no se corrijan no se puede obtener un código ejecutable del programa. Este tipo de errores se corrigen con la ayuda del compilador y por medio del compilador.
- **Errores de ejecución**: Son errores que se producen cuando se realiza una operación no válida que da lugar a la finalización anormal del programa (división por cero, abrir ficheros que no existen, acceder a zonas de memoria no permitidas, etc.).
- **Errores lógicos**: Son errores debidos a la traducción incorrecta de las especificaciones del programa. El programa termina normalmente pero no produce los resultados esperados. Esta

clase de errores se pueden analizar, observar y corregir con ayuda del depurador GDB que se estudiará en esta guía.

4 PROCEDIMIENTO GENERAL

Muchas veces se diseñan programas en lenguaje ensamblador, pero el programa que se diseña no siempre cumple con el objetivo y puede mostrar otros resultados inesperados, para situaciones como esta es útil un depurador. El procedimiento que se llevará a cabo para analizar programas en lenguaje ensamblador será el siguiente:

1. Estudiar las herramientas que proporciona el depurador GDB y aplicarlas en un programa en *assembler* propuesto.
2. Comprender y aplicar los comandos básicos de control del depurador.
3. Comprender y aplicar los comandos básicos de inspección de GDB.
4. Analizar y aplicar los comandos básicos para la alteración en la ejecución de un programa.
5. Resolver errores en un programa desde el depurador.

4.1 PROGRAMA EJEMPLO

Para un mejor entendimiento de cómo funciona GDB y sus comandos se utilizará un programa ejemplo diseñado para que se puedan aplicar y analizar cada uno de los comandos que se van a estudiar más adelante. Este programa se llama *sumavector.s* y contiene un arreglo de n elementos. El objetivo de este programa es hallar la suma de los elementos que se encuentran en las posiciones impares del vector V (V[1] + V[3] + V[5] + V[7]), además indica por medio de un mensaje si el resultado de la suma es positivo, negativo o cero.

```
1  .data
2  cero:           .ascii "El resultado de la suma es cero %d\n"
3  positivo:       .ascii "El resultado de la suma es positivo %d\n"
4  negativo:       .ascii "El resultado de la suma es negativo %d\n"
5  V:              .word 1, -2, 3, -4, 5, -6, 7, -8   # datos a sumar V[0]=1, V[1]=-2...
6  n:              .word 8
7
8  .text
9  .globl main
10 main:           li $s0, 0                  # s0 = suma = 0
11                 li $s1, 0                  # s1 = j = 0, subíndice
```

```
12                  li $s2, 4                   # s2 = 4, cuenta palabras
13                  lw $s3, n                   # s3 = 8, elementos del vector V
14
15  ciclo:          bge $s1, $s3, imprimir      # Mientras j<n, si no salta a imprimir
16                  mul $s5, $s2, $s1           # 4 bytes = 1 palabra = i = 4*j
17                  lw $s4, V($s5)              # s4 = carga V[i]
18                  add $s0, $s0, $s4           # s0 = suma los elementos y los acumula
19                  addi $s1, $s1, 1           # j = j + 1, aumenta subíndice
20                  j ciclo                     # Repite el ciclo
21
22  imprimir:       beqz $s0, resulcero         # si s0 = 0 vaya a resulcero, si no siga
23                  bltz $s0, resulneg          # si s0 = 0 vaya a resulneg, si no siga
24
25  resulpos:       la $a0, positivo            # carga mensaje positivo
26                  j fin                       # imprime resultado positivo
27
28  resulneg:       la $a0, negativo            # carga mensaje negativo
29                  j fin                       # imprime resultado negativo
30
31  resulcero:      la $a0, cero                # carga mensaje cero
32                  j fin                       # imprime resultado cero
33
34  fin:            move $a1, $s0               # carga s0 en a1 para imprimir suma
35                  j printf                    # Muestra el resultado en consola
36                  nop
```

Script 2.1: Programa ejemplo *sumavector.s* con errores para efectos pedagógicos

Este programa se compila y se envía a la tarjeta SIE, luego se ejecuta con el depurador. Estos procedimientos se encuentran en el manual de usuario.

Para una mejor comprensión del manejo del depurador y del desarrollo de la práctica, los comandos se han dividido en tres secciones: comandos de control, comandos de inspección y comandos de alteración en ejecución. Los comandos de cada sección serán aplicados en el programa ejemplo descrito anteriormente, el cual está diseñado para este propósito. La descripción de cada uno de los comandos de GDB utilizados en esta sección se pueden encontrar dentro del manual de usuario, junto con algunos ejemplos.

4.2 APLICACIÓN DE LOS COMANDOS DE CONTROL

Para aplicar estos comandos en el programa ejemplo. Primero se compilará el programa, se creará el ejecutable, posteriormente se envía a la tarjeta SIE y por último se ejecutará de la siguiente forma (*script* 2.2).

```
1 root@BenNanoNote:~# ./sumavector
2 El resultado de la suma es negativo -4
3 root@BenNanoNote:~#
```

Script 2.2: Ejecución del programa

Si se recuerda el objetivo de este programa es hallar la suma de los elementos que se encuentran en las posiciones impares del vector V, esto sería:

$$V[1] + V[3] + V[5] + V[7] = -2 + -4 + -6 + -8 = -20 \tag{2.1}$$

Sin embargo, al ejecutar el programa el resultado fue -4.

Si se analiza el programa, es posible que sea fácil descubrir los errores que produjeron este resultado no esperado, ya que los mismos podrían ser evidentes para algunos. Sin embargo, se recomienda no corregir el código fuente, sino seguir los pasos que se van a describir más adelante, donde se utilizará el depurador para determinar lo que está ocurriendo y ver como se procede en el caso donde los errores no son tan evidentes para detectarlos con solo inspeccionar el código fuente o en el caso que no se tenga acceso a él.

Ahora se aplicarán algunos de los comandos básicos de GDB que ayudarán a encontrar el por qué del resultado. Lo primero que se debe hacer es ejecutar el programa con GDB. A continuación se creará un punto de parada en la etiqueta ciclo, en este punto es donde se hace la sumatoria del vector y aquí se puede saber el número de veces que se ejecuta este ciclo.

```
1 (gdb) break ciclo
2 Breakpoint 1 at 0x4007a4: file sumavector.s, line 15.
3 (gdb)
```

Script 2.3: Punto de parada en la etiqueta ciclo

Al crear el punto de parada, GDB provee información correspondiente a: el número, la dirección, el archivo y la línea donde se creó el punto de parada. Esta información es importante ya que más adelante se puede necesitar.

Se tiene un vector de 8 elementos, pero sólo se van a sumar aquellos que están en las posiciones impares, siendo así, el ciclo debería ejecutarse solo 4 veces, esto se comprobará a continuación. Primero

se ejecutará el programa con el comando *run*, una vez se ejecuta este comando el programa se detendrá cuando llegue a la etiqueta ciclo como se puede observar a continuación (*script* 2.2).

```
1  (gdb) run
2  Starting program: /root/sumavector
3  GDB will be unable to debug shared library initializers
4  and track explicitly loaded dynamic code.
5  warning: no loadable sections found in added symbol-file /lib/libgcc_s.so.1
6  warning: no loadable sections found in added symbol-file /lib/libc.so.0
7  Breakpoint 1, ciclo () at sumavector.s:16
8  16      sumavector.s: No such file or directory.
9          in sumavector.s
10 Current language:  auto; currently asm
11 (gdb)
```

Script 2.4: Empezar la ejecución

Nota: cuando se este depurando un programa es importante tener en cuenta que el depurador en un punto de parada muestra la siguiente línea a ejecutar y no la línea donde se encuentra el punto, esto se debe a que la línea donde se encuentra el punto de parada ya ha sido depurada.

Después de que el programa se detiene por primera vez se puede utilizar el comando *continue* las veces que sea necesario para que siga ejecutándose hasta llegar al final. Si se cuenta el número de veces que se ejecutó el ciclo, se puede observar que fueron ocho, aunque el punto de parada es alcanzado 9 veces, pero hay que tener en cuenta que la última vez que es alcanzado hace la comparación y salta, por esta razón no se debe contar. Debido a todo esto se puede suponer que al parecer el programa está recorriendo y sumando todos los elementos.

Una vez que se ha identificado la parte en donde está el error, se verificará colocando un punto de parada tipo *watch* en el registro $s0, con este comando se observará cómo se está modificando la variable "suma", la cual se encuentra en el registro $s0. Ahora se puede eliminar el punto de parada anterior ya que no es de utilidad, para eliminar este punto de parada se usa el comando *delete* seguido del punto de parada a eliminar (1) y con el comando *info breakpoints* se puede comprobar que efectivamente se ha borrado.

Antes de hacer las anteriores modificaciones es necesario reiniciar la ejecución del programa con el comando *run*, de modo que se va a quedar ubicado en la etiqueta ciclo y aquí es donde se procede a hacer los cambios enunciados de la siguiente manera:

```
1  (gdb) watch $s0
2  Watchpoint 2: $s0
3  (gdb) delete 1
4  (gdb) info breakpoints
5  Num     Type           Disp Enb Address    What
6  2       watchpoint     keep y              $s0
7  (gdb)
```

Script 2.5: Punto de parada tipo *watch*

Ahora se hará un procedimiento similar al anterior, se usará el comando *continue* para avanzar en la ejecución cada vez que el programa se detenga por un cambio en el registro $s0. Si observa el programa notará que las paradas serán en el momento que suma cada elemento y los acumula en el registro $s0 (línea 18). Con este punto de parada se comprobará si se están sumando todos los elementos del vector, esto será fácil de comprobar, ya que con este tipo de parada el depurador muestra el antiguo y nuevo valor del registro, así que se puede ver paso a paso la sumatoria como se muestra a continuación:

```
1  (gdb) continue
2  Continuing.
3  Watchpoint 2: $s0
4  Old value = 0
5  New value = 1
6  ciclo () at sumavector.s:19
7  19      in sumavector.s
8  (gdb) continue
9  Continuing.
10 Watchpoint 2: $s0
11 Old value = 1
12 New value = -1
13 ciclo () at sumavector.s:19
14 19      in sumavector.s
15 (gdb)
```

Script 2.6: Punto de parada en la variable suma

Ejecutando lo anterior se comprueba que la sospecha era cierta, ya que después de ejecutar dos veces el comando *continue* puede verse que en las primeras dos iteraciones se sumaron los primeros dos elementos del vector. Si sigue ejecutando el comando *continue* observará que efectivamente se está sumando un elemento del vector por cada iteración sin importar si es par o impar, esta es la razón del resultado final (-4).

$$V[0]+V[1]+V[2]+V[3]+V[4]+V[5]+V[6]+V[7] = 1+-2+3+-4+5+-6+7+-8 = -4 \quad (2.2)$$

De acuerdo al análisis anterior es posible que el error tenga que ver con el subíndice "i", el cual no está iniciando ni se está actualizando bien. Hasta aquí se aplicarán los comandos de control y en la siguiente sección se seguirá con el análisis del programa pero aplicando los comandos básicos de inspección.

4.3 APLICACIÓN DE LOS COMANDOS DE INSPECCIÓN

En esta sección se utilizarán algunos de los comandos de inspección para seguir analizando el programa ejemplo. Con estos comandos será más fácil hacer un seguimiento detallado en los valores requeridos y en especial sobre el subíndice "i".

Lo primero que se hará es eliminar el *watchpoint* que se había creado e insertar un *breakpoint* en la misma posición, osea donde está el registro $s1, pero esta vez en la línea 19. La razón de este cambio es evitar que el programa se detenga en cualquier otra posición, como por ejemplo, una llamada al sistema donde se utilice este registro, estos pasos se harán así:

```
1 (gdb) delete 2
2 (gdb) break 19
3 Breakpoint 3 at 0x4007bc: file sumavector.s, line 19.
4 (gdb
```

Script 2.7: Punto de parada en la línea 19

Como el programa ya se estaba ejecutando es necesario reiniciarlo con el comando *run* y en seguida solicitará una confirmación, se le dice que si. El programa volverá a ejecutarse deteniéndose por primera vez en la línea 19 justo después de ejecutar la instrucción suma.

Parados en este punto se podrá verificar los valores de todos los registros mediante el comando *info registers*, en este paso se inspeccionarán todos los registros que se relacionen con el subíndice "i". Se utilizará este comando para hacer el recorrido en el vector durante cada ciclo. Después de ejecutar este comando se obtiene lo siguiente:

```
1 (gdb) info registers
2            zero       at       v0       v1       a0       a1       a2       a3
3  R0   00000000 00000001 2b013d50 2af7f2d0 00000001 7f9b0fa4 7f9b0fac 00000000
4              t0       t1       t2       t3       t4       t5       t6       t7
5  R8   2f2f2f2f aaff8d90 2af77014 00000b3b 7f1c0300 ffffffff 2af87144 2aff80a0
6              s0       s1       s2       s3       s4       s5       s6       s7
7  R16  00000001 00000000 00000004 00000008 00000001 00000000 00457e75 00457e95
```

```
8                t8        t9        k0        k1        gp        sp        s8        ra
9   R24  00000488 00400780 7f9b0b41 00000000 2b0173a0 7f9b0e60 00000000 2aff8128
10            status        lo        hi  badvaddr     cause        pc
11          00000413 00000000 00000000 2afeda80 00800024 004007c0
12              fcsr       fir   restart
13          00000000 00000000 00000000
14  (gdb)
```

Script 2.8: Contenido de registros

Si se observa bien el programa en *assembler* se ve que el subíndice "i" esta guardado en el registro $s5. Aquí es evidente el primer problema, el subíndice debería iniciar en cuatro (debido a que en cada posición de memoria contiene 4 bytes), ya que el programa solo hace la sumatoria de los elementos impares del vector. Sin embargo se ve que el primer elemento que se va a cargar es V[0] porque el dato contenido en el registro $s5 es cero. También se puede ver que en el registro $s1 fue donde se inicio previamente el subíndice, el cual después es multiplicado por 4 y guardado en el registro $s5. Antes de hacer cualquier modificación se comprobará los elementos del vector "V" por medio del comando *print* para verificar que estén correctos, esto sería de la siguiente manera:

```
1  (gdb) print *(&V+0)
2  $1 = 1
3  (gdb) print *(&V+1)
4  $2 = -2
5  (gdb) print *(&V+2)
6  $3 = 3
7  (gdb)
```

Script 2.9: Elementos del vector V

También se puede hacer algo similar para ver los valores de "i" y "V[i]" cada vez que la ejecución se detenga en la línea de la instrucción suma. Esto es posible por medio del comando *display* y se utiliza así:

```
1  (gdb) display $s5
2  1: $s5 = 0
3  (gdb) display $s0
4  2: $s0 = 1
5  (gdb)
```

Script 2.10: Contenido de "i" y "V(i)"

Se puede utilizar el comando *continue* varias veces para observar la forma en que varia el subíndice "i" y el resultado de la sumatoria en cada ciclo.

Esta experiencia permite ver el segundo error del programa, el subíndice se está incrementando solo una posición (4 bytes), mientras que en cada iteración debería moverse de a dos posiciones (8 bytes), ya que se quiere sumar solo los elementos de las posiciones impares. De esta forma se puede concluir que las instrucciones que se tienen que corregir serían las siguientes:

```
1 main:         li $s1, 0                   # s1 = j = 0, subíndice línea 10
2               addiu $s1, $s1, 1           # j = j + 1, aumenta subíndice línea 19
```

Script 2.11: Instrucciones a corregir

En la primera instrucción el subíndice "j" inicia en cero cuando debería iniciar en uno. En la segunda instrucción el subíndice "j" está aumentando un elemento por cada ciclo, donde debería aumentar dos por cada iteración. Si se modifica el subíndice "j" se corrige el subíndice "i" (i=j*4).

Una ventaja que tiene GDB es que no es necesario salir del modo de depuración para ir a corregir las instrucciones en el archivo original y volver a hacer el procedimiento hasta enviar de nuevo el ejecutable a la tarjeta, sino que maneja una serie de comandos que evitan hacer todo este procedimiento. En la siguiente sección se estudiarán los comandos que permiten hacer lo dicho anteriormente.

4.4 APLICACIÓN DE LOS COMANDOS PARA LA ALTERACIÓN

Una vez se identifican los posibles errores en el programa con ayuda de los anteriores comandos, se procederá a modificar los valores del subíndice "j" con el fin de comprobar si el resultado será correcto al cambiar las dos instrucciones determinadas en la sección anterior.

Es recomendable hacer los cambios uno a uno desde el principio hasta el final del programa. Lo primero que se hará es inicializar el subíndice "j" en uno, esto se realizará de forma manual utilizando el comando *set*. Para ello se debe insertar un nuevo punto de parada en la primera instrucción del programa, ubicada en la etiqueta *main*.

Una vez creado el punto de parada en *main* se ejecutará el programa y automáticamente se detendrá en el primer punto de parada que esté habilitado, en este caso será en la etiqueta *main* ubicada en la línea 10, a partir de este punto se ejecuta el programa paso a paso con el comando *stepi* hasta que se inicie el registro $s1 en cero (línea 11). En este momento se puede corregir este registro manualmente utilizando el comando *set* para inicializarlo en uno. Esto se hace de la siguiente forma:

```
1 (gdb) break main
2 Breakpoint 4 at 0x400784: file sumavector.s, line 10.
3 (gdb) run
4 Breakpoint 4, main () at sumavector.s:11
5 11      in sumavector.s
6 (gdb) stepi
7 12      in sumavector.s
8 (gdb) set $s1=1
```

Script 2.12: Inicializa el subíndice "j" en 1

De este modo se solucionará el primer problema.

La segunda modificación es incrementar el subíndice de dos en dos, esto se hace mediante el uso del comando *display*. Dado que existe un punto de parada dentro del ciclo en la instrucción de suma (línea 19), y se sabe que el índice actualmente se incrementa en uno con cada iteración, entonces se asignará al índice su valor más uno para que vaya aumentando de dos en dos ($s1 = $s1 + 1) cada vez que la ejecución se detenga en este punto de parada. En este punto se utilizará el comando *display* para verificar y visualizar el valor de subíndice (registro $s1). Luego se ejecuta el comando *continue* hasta que termine el programa verificando que la respuesta sea la correcta.

```
1 (gdb) continue
2 Breakpoint 2, ciclo () at sumavector.s:19
3 19      in sumavector.s
4 (gdb) display $s1=$s1+1
5 1: $s1 = $s1 + 1 = 2
6 (gdb) display $s1
7 2: $s1 = 2
8 (gdb) continue
9 Breakpoint 2, ciclo () at sumavector.s:19
10 19      in sumavector.s
11 2: $s1 = 3
12 1: $s1 = $s1 + 1 = 4
13 (gdb)
14 Breakpoint 2, ciclo () at sumavector.s:19
15 19      in sumavector.s
16 2: $s1 = 5
17 1: $s1 = $s1 + 1 = 6
18 (gdb)
19 Breakpoint 2, ciclo () at sumavector.s:19
20 19      in sumavector.s
21 2: $s1 = 7
22 1: $s1 = $s1 + 1 = 8
23 (gdb)
24 Continuing.
```

```
25 El resultado de la suma es negativo -20
```

Script 2.13: Incrementa el subíndice de dos en dos

Por medio del anterior procedimiento se comprueba que las posiciones de los elementos que en verdad se están sumando son las impares, esto es:

$$V[1] + V[3] + V[5] + V[7] = (-2) + (-4) + (-6) + (-8) = -20 \tag{2.3}$$

Una vez que se está seguro de los errores y de la efectividad de las soluciones que se han propuesto, se procede a modificar el programa original, no solo con la seguridad de que al compilarlo de nuevo el programa funcionará de forma adecuada, sino con el conocimiento de cual fue el problema y con una justificación clara de por qué las modificaciones hechas han sido exitosas.

5 RESUMEN

Un depurador es una herramienta útil para todo programador que permite observar cómo funciona un programa durante su ejecución. Muchas veces se presentan situaciones donde se tienen que analizar programas hechos por otras personas, sin tener el código fuente o la documentación adecuada, aún si se tratase de un código propio, el uso de un depurador permite analizar programas que terminaron con un fallo e incluso procesos que ya se encuentran en ejecución, con la posibilidad de corregirlo, sin tener la necesidad de cambiar su código, hasta que se tenga plena seguridad e información sólida para respaldar los cambios. Por esto es importante aprender a utilizar un depurador.

Aunque es necesario tener una idea lo suficientemente clara del lenguaje ensamblador y sus principios básicos de funcionamiento, no es necesario ser un experto en determinado lenguaje ensamblador o en un depurador específico, ya que cualquiera de estos puede cambiar. Lo importante es comprender el funcionamiento general del depurador con el fin de poder aprender a utilizarlo de forma efectiva.

6 PREGUNTAS DE PRUEBA

- ¿Por qué es importante usar un depurador en un programa?
- ¿Qué ventajas tiene el uso del depurador en el programa ejemplo?
- En términos generales. ¿Qué ventajas tiene la solución de errores con ayuda del depurador?
- Utilice el depurador para corregir los errores, si los tuvo, en el programa que diseño en el laboratorio 1.

- Diseñe un programa en *assembler* que ordene los elementos del vector *V[n]* y lo muestre en consola. Utilice la ayuda del depurador si es necesario.

Bibliografía

[1] Franco Rodríguez Fabregues: *GDB GNU Debugger*, JCórdoba, ARGENTINA, Universidad Nacional de Córdoba, (2011).

[2] Eduardo Domínguez Parra y Carlos Villarrubia Jiménez: *Prácticas de Sistemas Operotivos*, Huélamo, ESPAÑA, Universidad de Castilla, (2008).

LABORATORIO 3

MODOS DE DIRECCIONAMIENTO

"Pregúntate si lo que estás haciendo hoy te acerca al lugar en el que quieres estar mañana"

J.BROWN,E.U.

1 INTRODUCCIÓN

En esta sección se tratará de manera teórica y práctica los principales modos de direccionamiento de la arquitectura MIPS, se hace importante retomar cierta información previa del repertorio de instrucciones expuesto en la práctica 1.

Recuerde que la arquitectura MIPS cuenta con un juego de instrucciones bastante amplio que abarca todo tipo de operaciones (aritmético, lógicas, salto, comparación, carga y almacenamiento), pero hay que resaltar que se distinguen por los diferentes tipos de formato. El modo de direccionamiento especifica la forma de interpretar la información contenida en cada campo del formato de instrucciones (I, J, K), es así como el repertorio de instrucciones sirve como punto de partida para poder interpretar el funcionamiento de los modos de direccionamiento que se pueden presentar en una arquitectura MIPS.

2 OBJETIVOS

- Estudiar cada uno de los modos de direccionamiento disponibles de la arquitectura MIPS, para usarlos adecuadamente en la optimización de un programa.
- Observar el recorrido que se hace en cada modo de direccionamiento apoyándonos en el *datapath*, basado en la arquitectura MIPS.
- Implementar cada uno de los modos de direccionamiento que posee el procesador de la tarjeta SIE.

- Analizar cada tipo de direccionamiento dentro del procesador por medio del depurador GDB.

3 MARCO TEÓRICO

3.1 MODOS DE DIRECCIONAMIENTO

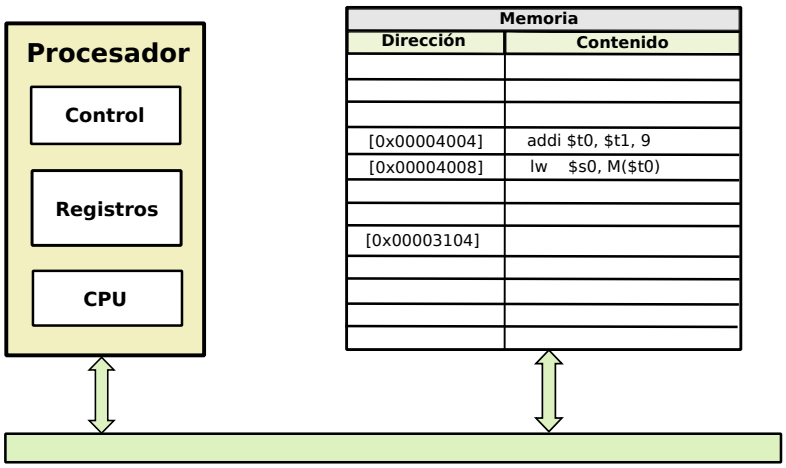

Figura 3.1: Elementos que intervienen en un direccionamiento.
FUENTE: El autor.

Un modo de direccionamiento es un procedimiento que permite determinar un operando, la dirección de un operando o una instrucción (datos a procesar). Lo más frecuente es especificar la dirección donde está almacenado el dato o la instrucción, empleando siempre el término modo de direccionamiento [1].

Se denomina objeto al operando, resultado o instrucción que se desea direccionar. Los operandos de la instrucción pueden venir especificados por un registro del procesador, en su propia instrucción (una constante) o en una posición de memoria.

Los modos de direccionamiento disponibles están determinados por la arquitectura interna de la máquina y condicionados por el repertorio de instrucciones. Es posible observar el camino de datos de cada modo por medio de un *datapath* de la arquitectura MIPS (figura 3.2) [2].

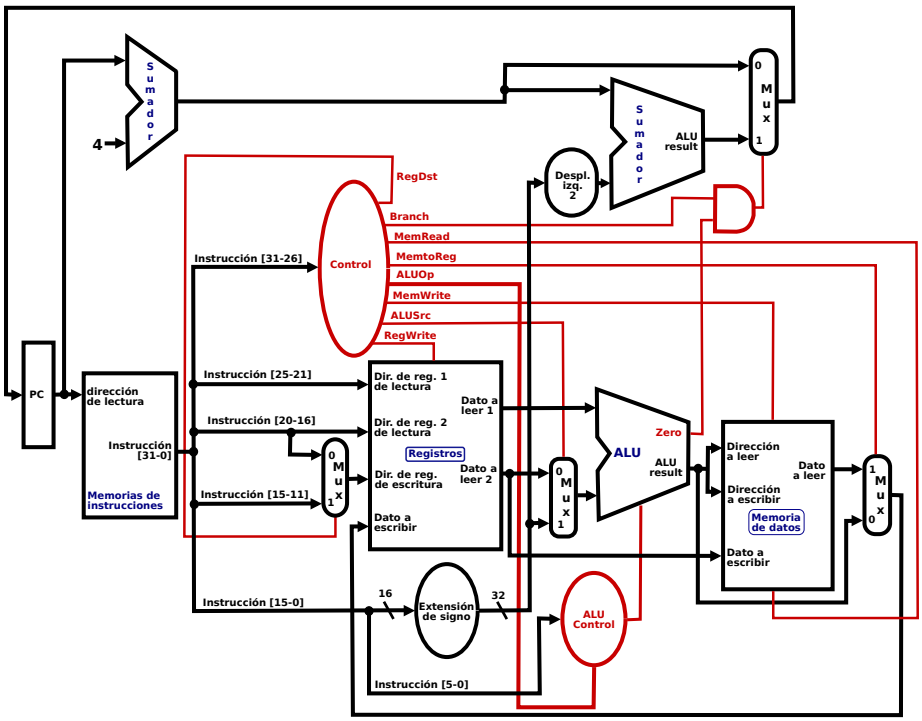

Figura 3.2: *Datapath* arquitectura MIPS. Figura tomada de [2]

FUENTE: [2].

4 PROCEDIMIENTO GENERAL

1. Conocer la función de cada uno de los tipos de direccionamiento presentes en la arquitectura MIPS.

2. Comprender de manera práctica cada uno de los modos de direccionamiento.

3. Analizar gráficamente el camino de datos de cada tipo de direccionamiento.

4. Implementar cada uno de los modos de direccionamiento por medio de un programa en lenguaje *assembler*.

5. Con el depurador GDB analizar cada uno de los modos de direccionamiento por medio del comando *dissamble*.

4.1 MODOS DE DIRECCIONAMIENTO EN LA ARQUITECTURA MIPS

Cada modo de direccionamiento utiliza diferente tipo de formato de instrucción, la descripción de los tipos de formato que se mencionan en esta práctica están descritos en el manual de usuario. Los principales modos de direccionamiento presentes en la arquitectura MIPS se describen a continuación.

4.1.1 Direccionamiento a Registro

El operando referenciado se encuentra especificado en un registro como se ve en la figura 3.3. El número de registro se especifica en un campo dentro de la instrucción. Las ventajas son: que solo es necesario un pequeño campo de direcciones en la instrucción y que no requiere referencias a memoria. El tiempo de acceso a un registro interno es mucho menor que el tiempo de acceso a la memoria principal. La desventaja es que el espacio de posiciones es muy limitado.

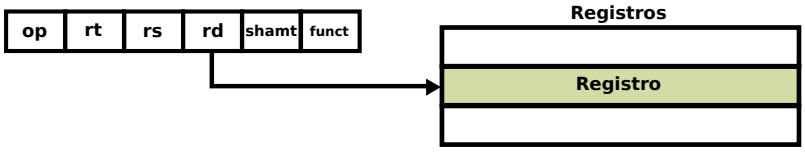

Figura 3.3: Direccionamiento a registro.

FUENTE: El autor.

Este tipo de direccionamiento utiliza el formato de instrucción tipo R. A continuación se presenta un ejemplo de una instrucción donde se aplica este modo (*Script*) 3.1, junto con su representación dentro de su formato en la tabla 3.1.

```
1           add     $s0 , $s1 , $s2          # $s0 = $s1 + $s2
```

Script 3.1: Ejemplo de direccionamiento a registro

op	rs	rt	rd	shamt	funct
0	$s1	$s2	$s0	0	32
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Tabla 3.1: Formato para direccionamiento a registro.

Esta instrucción suma el contenido del registro $s1 con el contenido del registro $s2 y lo guarda en el registro $s0.

Este modo de direccionamiento se puede observar paso a paso por medio del *datapath* de las figuras 3.4, 3.5, 3.6 y 3.7. De esta manera se comprenderá como funciona básicamente un procesador internamente [2].

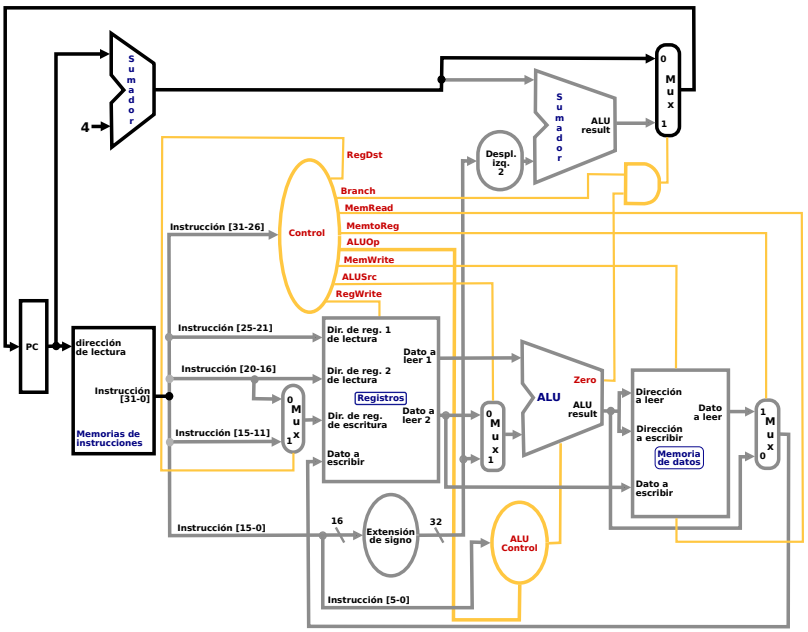

Figura 3.4: *Datapath* direccionamiento a registro paso 1.

FUENTE: [2].

En la figura 3.4 busca y después incrementa el contador de programa y examina la instrucciones que se debe ejecutar. El camino y las partes que se activan en este paso están resaltadas con color negro y rojo, mientras que las que no se activan están de color gris y naranja.

En el segundo paso se leen los dos registros fuente del archivo de registros ($s2 y $s1). La unidad de control principal utiliza el campo de código de operación para determinar la activación de las líneas de control y comunicarse con la ALU. Estas señales se activan como se muestra en la figura 3.5.

La operación de la ALU en el registro de operandos de datos se involucra en el tercer paso figura 3.6. Todos los valores de la línea de control se activan junto con el control de la ALU. En este momento es donde la ALU opera sobre el dato.

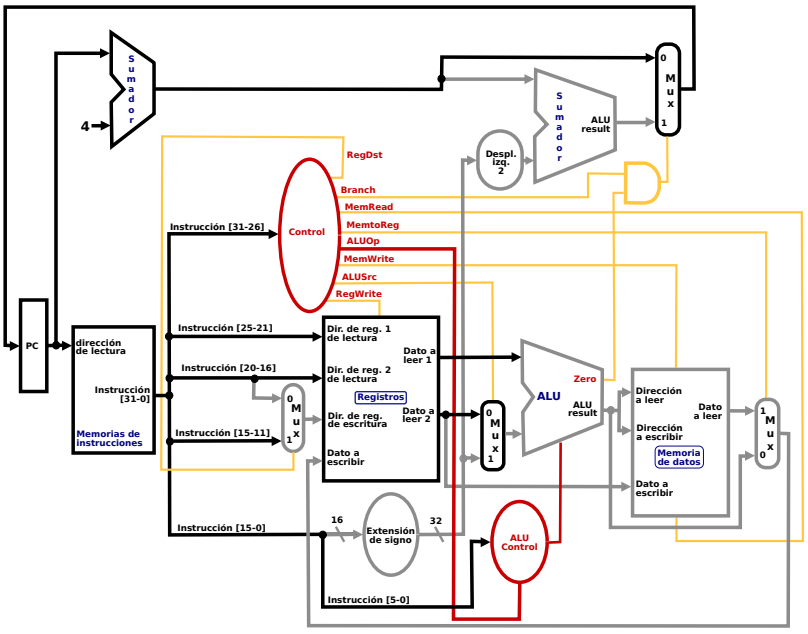

Figura 3.5: *Datapath* direccionamiento a registro paso 2.

FUENTE: [2].

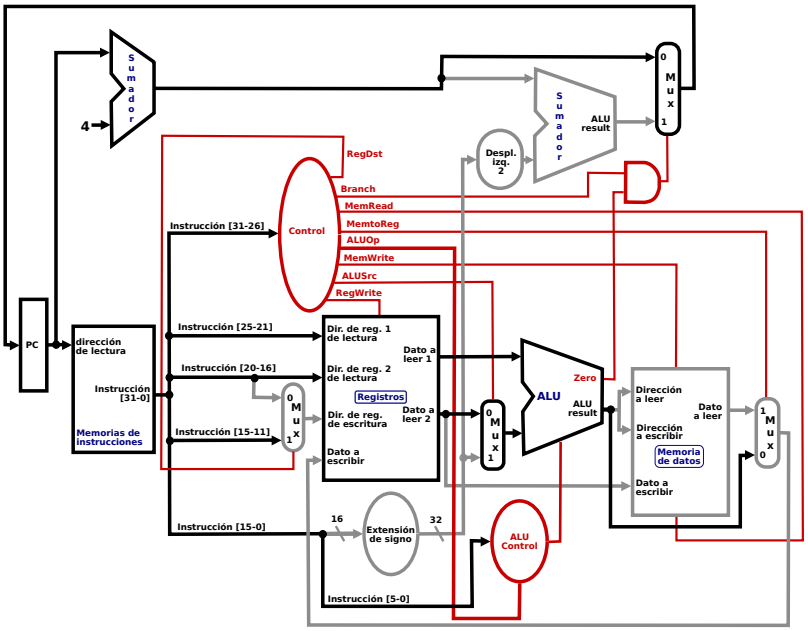

Figura 3.6: *Datapath* direccionamiento a registro paso 3.

FUENTE: [2].

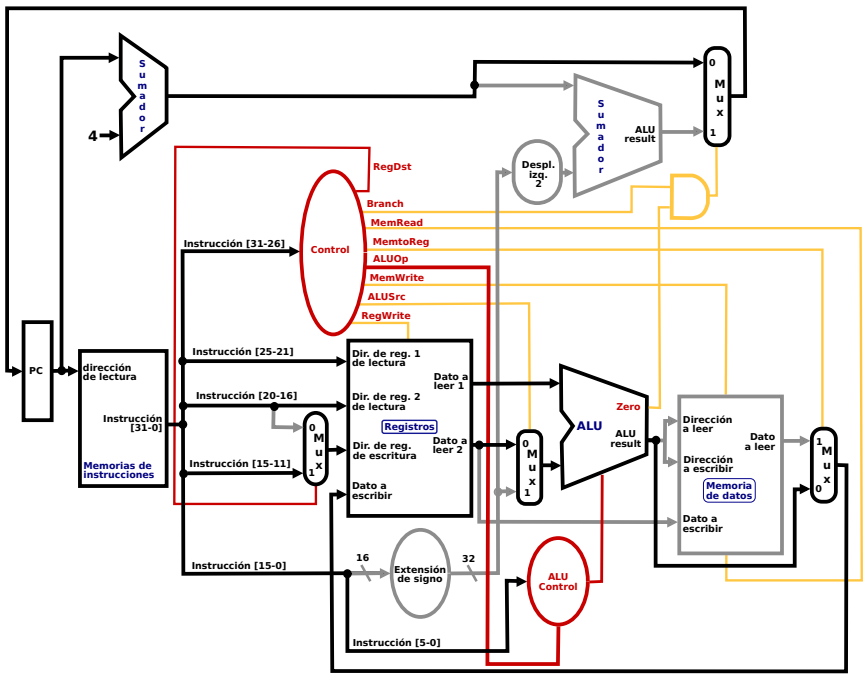

Figura 3.7: *Datapath* direccionamiento a registro paso final.

FUENTE: [2].

En la figura 3.7 se observa el último paso donde el control activa el *mux* para escribir el data en el registro *Rd*. El contador de programa (PC) también se actualiza en este paso final. Así es el camino de datos para un direccionamiento a registro [2].

Ahora ya teniendo una idea gráfica del funcionamiento interno del procesador se diseñará un programa en *assembler* donde se pueda simular este tipo de direccionamiento. El programa consiste en sumar dos números que se encuentran almacenados en dos registros, el resultado es almacenado en un tercer registro y mostrado en consola.

```
1  .data
2  RTA:     .ascii "la suma es: %d\n"
3  .text
4  .globl main
5  main:            li $s0, 0                  # s0= suma = 0
6                   li $s1, 8                  # carga 8 en el registro $s1
7                   li $s2, 6                  # carga 6 en el registro $s2
8                   add     $s0, $s1, $s2      # $s0= $s1 + $s2 (direccionamiento a registro)
9                   la      $a0, RTA           # a0= v0, carga mensaje RTA
```

```
10                  move    $a1, $s0            # carga el resultado de suma guardada en $s0
11  imprime:        j       printf              # imprime mensaje y resultado de la suma
12                  nop                         # fin
```

Script 3.2: Programa utilizando un direccionamiento a registro

Observe el *script* 3.2 en la línea 8 del programa se utiliza el modo de direccionamiento a registro, si se usa un depurador para ejecutar este programa se podría ver cuantos pasos o instrucciones de máquina gasta en ejecutar este tipo de direccionamiento. Para analizar este programa con GDB se pueden utilizar varios comandos, en esta ocasión se usará el comando *dissamble*, el cual dejará ver el programa en un nivel mas bajo como lo es el lenguaje de máquina. Los pasos para crear el ejecutable, enviarlo a la tarjeta y depurarlo estan descritos en el manual de usuario.

Usando el comando *disasemble* con la etiqueta *main* como argumento se mostrará el código de máquina en un rango del programa donde se encuentre la línea del modo de direccionamiento que se está analizando.

```
1  (gdb) disassemble main
2  Dump of assembler code for function main:
3  0x00400780 <main+0>:    li      s0,0
4  0x00400784 <main+4>:    li      s1,8
5  0x00400788 <main+8>:    li      s2,6
6  0x0040078c <main+12>:   add     s0,s1,s2
7  0x00400790 <main+16>:   lui     a0,0x40
8  0x00400794 <main+20>:   addiu   a0,a0,2112
9  0x00400798 <main+24>:   move    a1,s0
10 End of assembler dump.
11 (gdb)
```

Script 3.3: Lenguaje de máquina direccionamiento a registro

Una vez que se obtiene el código de máquina del *script* 3.3, se puede observar que para ejecutar este modo de direccionamiento a registro utiliza solo una instrucción de máquina (línea 6).

4.1.2 Direccionamiento inmediato

Figura 3.8: Direccionamiento inmediato.
FUENTE: El autor.

Uno de los operandos es una constante cuyo valor se almacena en el campo (inmediato) de 16 bits como se puede ver en la figura 3.8. La ventaja del direccionamiento inmediato es que una vez captada

la instrucción no se requiere una referencia a memoria para obtener el operando porque esta contenido dentro de la instrucción, por este motivo tarda menos tiempo que el anterior. La desventaja es que el tamaño de la constante está restringida a la longitud del campo de direcciones (16 bits). Este direccionamiento utiliza el formato de instrucción tipo I tabla 3.2. Un ejemplo donde se puede aplicar este tipo de direccionamiento se puede observar en la instrucción del *script* 3.4.

```
1  addi $s0, $s1, 4            # $s0 = $s1 + 4
```

Script 3.4: Ejemplo de direccionamiento inmediato

op	rs	rt	dato
0	$s1	$s0	4
6 bits	5 bits	5 bits	16 bits

Tabla 3.2: Formato para direccionamiento inmediato.

En esta instrucción se suma la constante 4 con el contenido del registro $s1 y se almacena en el registro $s0.

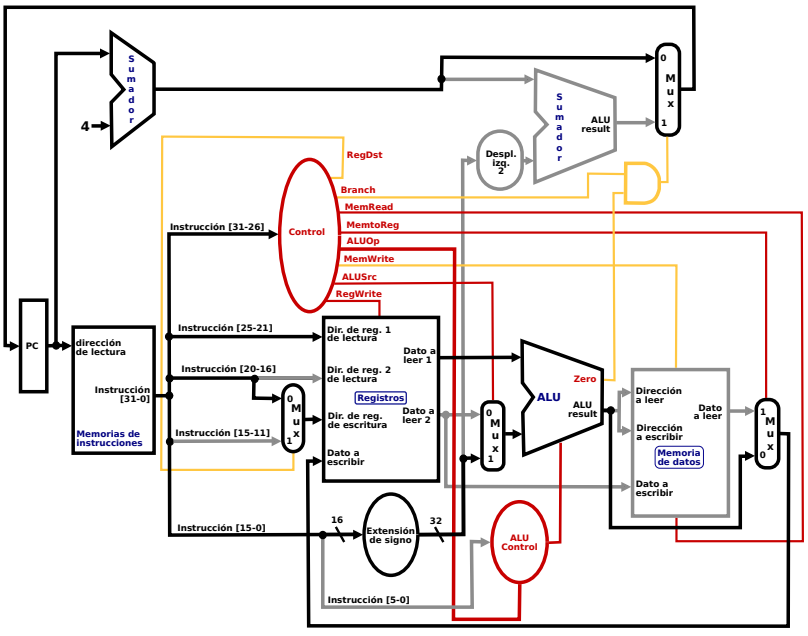

Figura 3.9: *Datapath* direccionamiento inmediato.

FUENTE: [2].

Este modo de direccionamiento también se puede analizar por medio del *datapath* de la figura 3.9, donde se puede observar de forma resaltada el camino de datos [2].

De acuerdo a la figura 3.9, se observa que el camino de datos es similar con el direccionamiento a registro, pero aquí el segundo dato que se suma se hace de forma directa sin necesidad de entrar al banco de registros, ya que el dato está dentro de la instrucción. Después de pasar por la ALU escribe el resultado directamente en el registro correspondiente.

Con el programa en *assembler* descrito en el *script* 3.5, se analizará este tipo de direccionamiento. Este programa es similar al del modo anterior la diferencia radica en que los operandos ya no son dos registros sino que uno de ellos es una constante que está implícita dentro de la instrucción.

```
1  .rdata
2  RTA:     .ascii "la suma es: %d\n"
3  .text
4  .globl main
5  main:            li $s0, 0                  # s0 = suma = 0
6                   li $s1, 8                  # carga 8 en el registro $s1
7                   addi $s0, $s1, 4           # $s0 = $s1 + 4 (direccionamiento inmediato)
8                   la $a0, RTA                # a0 = v0, carga mensaje RTA
9                   move    $a1, $s0           # carga el resultado de suma guardada en $s0
10 imprime:         j printf                   # imprime mensaje y resultado de la suma
11                  nop                        # fin
```

Script 3.5: Programa utilizando un direccionamiento base

En la línea 7 del programa se utiliza una instrucción con tipo de direccionamiento inmediato. Usando el depurador para ejecutar este programa se pueden ver cuantos pasos o instrucciones de máquina toma ejecutar este modo de direccionamiento.

Luego de realizar el procedimiento para compilar, enviar y ejecutar el programa en la tarjeta SIE y el procedimiento para ejecutar el programa con el depurador, los cuales se encuentran en el manual de usuario. Ejecutando el comando *disasemble* con la etiqueta *main* como argumento, se mostrará el código de máquina en un rango del programa donde se encuentre la línea del tipo de direccionamiento que se está analizando (*script* 3.6).

```
1  (gdb) disassemble main
2  Dump of assembler code for function main:
3  0x00400780 <main+0>:     li       s0,0
```

```
4  0x00400784 <main+4>:    li      s1,8
5  0x00400788 <main+8>:    addi    s0,s1,4
6  0x0040078c <main+12>:   lui     a0,0x40
7  0x00400790 <main+16>:   addiu   a0,a0,2112
8  0x00400794 <main+20>:   move    a1,s0
9  End of assembler dump.
10 (gdb)
```

Script 3.6: Lenguaje de máquina direccionamiento inmediato

De acuerdo con lo anterior, debe aparecer en consola el código que está en el *script* 3.6, observe que para este modo de direccionamiento se utiliza una sola instrucción de máquina (línea 5). Esto se debe a que el camino que se recorre es corto y similar al del modo anterior.

4.1.3 Direccionamiento base

Esta clase de direccionamiento también es conocido como direccionamiento indirecto con registros. La instrucción contiene en los bits del 0 - 15 el número del registro en el que se almacena la dirección de memoria del operando deseado. La ventaja es que este direccionamiento emplea solo una referencia a memoria, la desventaja es que tarda más tiempo que los modos anteriores. Esto se puede comprender más adelante por medio de la tabla 3.3, el *datapath* de la figura 3.11 y con ayuda del depurador.

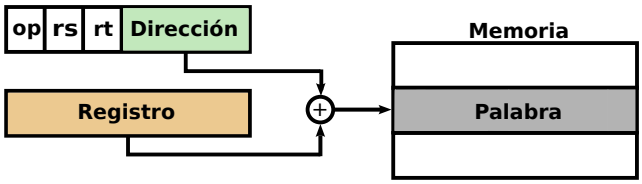

Figura 3.10: Direccionamiento base.

FUENTE: El autor.

Se tomará la instrucción de carga descrita en el *script* 3.7 como un ejemplo que ilustrará lo anterior. Este tipo de direccionamiento utiliza el formato tipo I, como se observa en la tabla 3.3.

```
1  lw $s0, V($s1)          # s0 = carga V[i]
```

Script 3.7: Ejemplo de direccionamiento base

La instrucción carga en el registro $s0 el elemento del vector V que está indicado por un subíndice contenido en el registro $s1.

op	rs	rt	dirección
35	$s1	$s0	V[i]
6 bits	5 bits	5 bits	16 bits

Tabla 3.3: Formato para direccionamiento base.

Analizando este modo de direccionamiento por medio del *datapath* de la figura 3.11 observe que su recurrido es diferente a los modos anteriores.

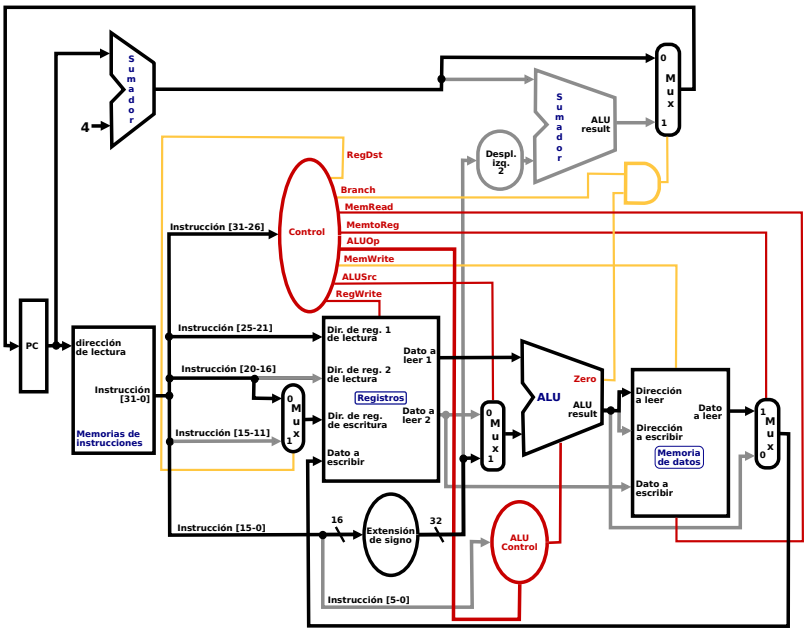

Figura 3.11: *Datapath* direccionamiento base.
FUENTE: [2].

En la instrucción de una operación de carga la principal diferencia radica en que en esta ocasión el camino de datos pasa por la memoria ya sea para cargar un dato a registro o para guardar un dato, esto le tomaría mas tiempo en la ejecución, debido a que tiene que hacer un mayor recorrido. Con un programa en *assembler* y para comprobar lo dicho anteriormente se observará el comportamiento de este direccionamiento. Lo importante para analizar de este programa es que se carga un dato de la memoria a un registro [2].

```
1  .data
2  RTA:            .ascii "El dato que se cargo es: %d\n"
3  V:              .word 1, -2, 3, -4, 5, -6, 7, -8
```

```
4  .text
5  .globl main
6  main:           li $s0, 0              # s0 = 0
7                  li $s1, 8              # $s1 subíndice i, donde i = $s1/4 = 2
8                  lw $s0, V($s1)         # s0 = carga V[i] (direccionamiento base)
9                  la $a0, RTA            # carga mensaje RTA
10                 move $a1, $s0          # carga el elemento de V[i] en $a1 para imprimir
11 imprime:        j printf               # imprime mensaje y el elemento V[i]
12                 nop                    # fin
```

Script 3.8: Programa utilizando un direccionamiento base

Observe que en la línea 8 del programa mostrado en el *script* 3.8, se usa este modo de direccionamiento por medio de una instrucción de carga, ahora se ejecutará este programa con GDB en la tarjeta SIE y se mostrará el código de máquina en un rango del programa donde se encuentre la línea de la instrucción del modo de direccionamiento que se está analizando.

```
1  (gdb) disassemble main
2  Dump of assembler code for function main:
3  0x00400780 <main+0>:    li      s0,0
4  0x00400784 <main+4>:    li      s1,8
5  0x00400788 <main+8>:    lui     s0,0x41
6  0x0040078c <main+12>:   addu    s0,s0,s1
7  0x00400790 <main+16>:   lw      s0,2188(s0)
8  0x00400794 <main+20>:   lui     a0,0x41
9  0x00400798 <main+24>:   addiu   a0,a0,2144
10 0x0040079c <main+28>:   move    a1,s0
11 End of assembler dump.
12 (gdb)
```

Script 3.9: Lenguaje de máquina direccionamiento base

De acuerdo con lo anterior observe que para este tipo de direccionamiento se utiliza tres instrucciones de máquina (líneas 5, 6 y 7) del *script* 3.9, esto es algo que se esperaba ya que se había comprobado de manera teórica y gráficamente. Este direccionamiento tiene que desplazarse hacia la memoria para cargar un dato en un registro, esto le implica ejecutar más instrucciones de máquina y consecuentemente toma más tiempo.

4.1.4 Direccionamiento relativo al PC

En la arquitectura MIPS se usa el direccionamiento relativo al PC para los saltos condicionales figura 3.12. El contador de programa (PC) se incrementa durante cada instrucción, por lo que cuando se

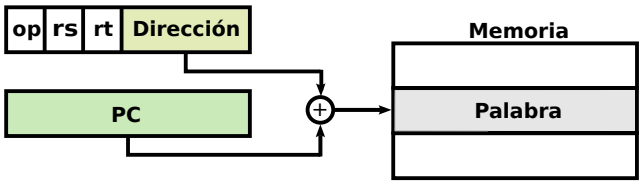

Figura 3.12: Direccionamiento relativo al PC.

FUENTE: El autor.

decide dar el salto, el PC ya apunta a la siguiente instrucción (PC+4).

Recuerde que las instrucciones MIPS tienen un tamaño fijo de 32 bits (4 bytes), por lo tanto los saltos serán siempre múltiplos de 4. Para conseguir una mayor distancia de saltos, esta multiplicación está implícita, por lo que un rango de salto puede ser de n*4 bytes. La dirección efectiva es un desplazamiento relativo a la dirección de la instrucción a donde se quiere saltar.

Por medio del ejemplo del *script* 3.10 se va a tener una visión más real del funcionamiento de este modo de direccionamiento.

```
1 bne $s0, $s1, cargue              # si $so es diferente a $s1 salta a la etiqueta cargue
```

Script 3.10: Ejemplo de direccionamiento relativo al PC

op	**rs**	**rt**	**dirección**
5	$s1	$s0	etiqueta
6 bits	5 bits	5 bits	16 bits

Tabla 3.4: Formato para direccionamiento relativo al PC.

La instrucción hace una comparación entre los registros $s0 y $s1, si su contenido es igual sigue ejecutando la siguiente instrucción, pero si su contenido es diferente salta a la etiqueta *cargue*. Este direccionamiento utiliza el formato tipo I y se observa en la tabla 3.4 ejemplificado con una instrucción.

Analizando este modo de direccionamiento por medio del *datapath* de la figura 3.13 se observa que su recorrido cambia respecto a los anteriores [2].

En la figura 3.13 se presenta el camino de datos para un direccionamiento relativo al PC. Después de leer los datos del banco de registros se activa la ALU y se verifica la salida *zero* para seleccionar el

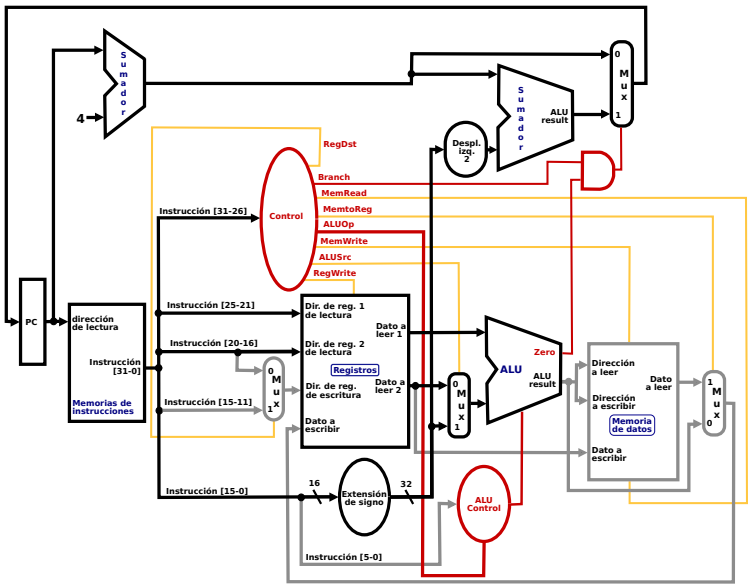

Figura 3.13: *Datapath* direccionamiento relativo al PC.

FUENTE: [2].

siguiente contador de programa (PC).

Con el programa en *assembler* del *script* 3.11, donde se utiliza una instrucción de salto se observará el comportamiento de este tipo de direccionamiento.

```
1  .data
2  RTA:     .ascii "El dato que se cargo en la memoria es: %d\n"
3  V:               .word 1, -2, 3, -4, 5, -6, 7, -8
4  .text
5  .globl main
6  main:            li $s0, 2              # carga 2 en el registro $s0, $s0 = 2
7                   li $s1, 5              # carga 5 en el registro $s1, $s1 = 5
8                   bne $s0, $s1, cargue   # si $s0 es diferente a 0 salta a la etiqueta
9                                          # cargue (direccionamiento relativo al PC)
10                  li $s2, 6              # carga 6 en el registro $s2, $s2 = 6
11                  nop                    # no opera
12 cargue:          la $a0, RTA            # carga mensaje RTA
13                  move $a1, $s0          # carga el dato de $s0 en $a1 para imprimir
14 imprime:         j printf               # imprime mensaje y el dato de $s0
15                  nop                    # fin
```

Script 3.11: Programa utiizando un direccionamiento relativo al PC

En el *script* 3.11 se utiliza el direccionamiento relativo al PC en la línea 7, ahora se ejecutará este programa con GDB en la tarjeta SIE y se mostrará el código de máquina en un rango del programa donde se encuentre la línea de la instrucción del modo de direccionamiento que se está analizando.

```
1  (gdb) disassemble main
2  Dump of assembler code for function main:
3  0x00400780 <main+0>:    li      s0,2
4  0x00400784 <main+4>:    li      s1,5
5  0x00400784 <main+4>:    bne     s0,s1,0x400794 <cargue>
6  0x00400788 <main+8>:    nop
7  0x0040078c <main+12>:   li      s2,6
8  0x00400790 <main+16>:   nop
9  End of assembler dump.
10 (gdb)
```

Script 3.12: Lenguaje de máquina direccionamiento relativo al PC

El *script* 3.12 se puede observar que este direccionamiento utiliza dos instrucciones de máquina (líneas 4 y 5) y la dirección de la etiqueta *cargue* es la que se muestra a continuación de la instrucción de salto (línea 4). Este tipo de direccionamiento es muy utilizado en ciclos donde se quiere comparar dos valores para luego tomar una decisión.

4.1.5 Direccionamiento pseudodirecto

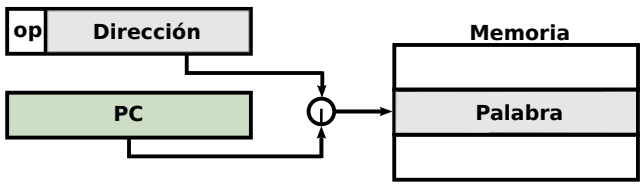

Figura 3.14: Direccionamiento pseudodirecto

FUENTE: El autor.

Es conveniente disponer de un modo de direccionamiento directo a memoria capaz de alcanzar todo el rango de direcciones para permitir que un programa pueda saltar a cualquier posición de memoria. Los MIPS contienen 32 bits para acceder al mapa de memoria. Como las instrucciones tienen un tamaño fijo de 32 bits, no se puede especificar una dirección completa dentro de una instrucción, una manera de extender el tamaño del salto es utilizar un nuevo formato de instrucción, denominado tipo J [3].

Para ver como funciona este tipo de direccionamiento dentro de la arquitectura de los MIPS se podría hacer mediante el ejemplo del *script* 3.13 y la tabla 3.5.

```
1  j  cargue                    # salto incondicional a la etiqueta cargue
```

Script 3.13: Ejemplo de direccionamiento pseudodirecto

op	**dirección**
35	etiqueta
6 bits	26 bits

Tabla 3.5: Representación del formato de instrucción tipo J.

Se puede hacer un salto para ir a cualquier dirección de la memoria sin exceder los 4 [GiB] que tiene como limite. En este ejemplo lo que se hace es saltar incondicionalmente a la posición de memoria en donde está la etiqueta *cargue*.

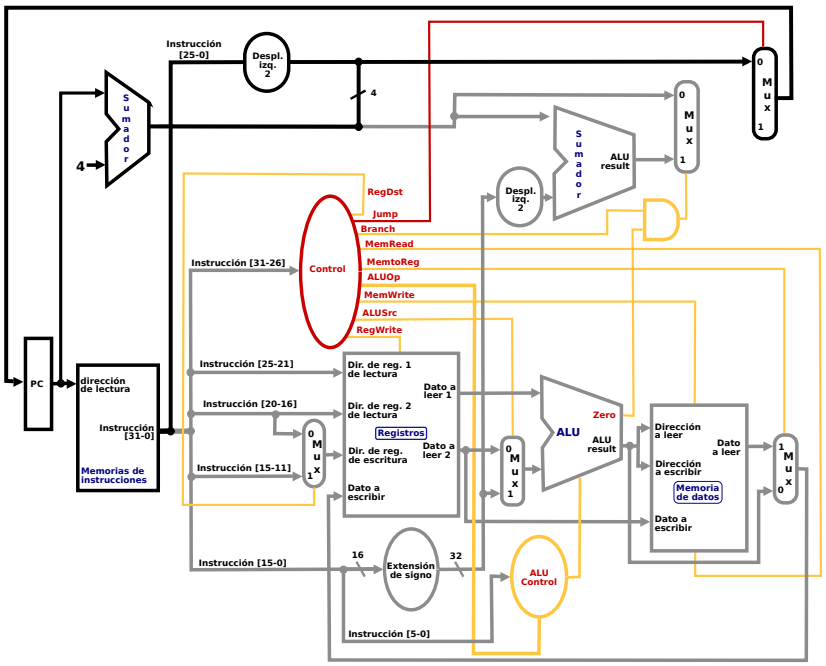

Figura 3.15: *Datapath* direccionamiento pseudadirecto.

FUENTE: [2].

Analizando este tipo de direccionamiento por medio del *datapath* de la figura 3.15 se observa el recorrido que hace dentro del procesador [2].

En la figura 3.15 se observa como el control y el camino de datos se extienden para llevar a cabo la instrucción de salto incondicional. El multiplexor que se ve en la parte superior derecha es el encargado de escoger entre el destino de salto o la instrucción secuencial que sigue a la actual. La señal de control de bifurcación *jump* es la encargada de controlar este multiplexor.

En un programa hecho en *assembler* en el *script* 3.14 se observará como se comporta este tipo de direccionamiento, utilizando la instrucción de salto incondicional.

```
1  .data
2  RTA:            .ascii "El dato que se cargo en la memoria es: %d\n"
3  .text
4  .globl main
5  main:           li $s0, 0       # carga 0 en el registro $s0, $s0 = 0
6                  li $s1, 6       # carga 6 en el registro $s1, $s1 = 6
7                  j cargue        # salto incondicional a la etiqueta
8                                  # cargue (direccionamiento pseudodirecto)
9                  li $s2, 2       # carga 2 en el registro $s2, $s2 = 2
10                 nop             # no opera
11 cargue:         la $a0, RTA     # carga mensaje RTA
12                 move $a1, $s0   # carga el dato guardado en $s0
13 imprime:        j printf        # imprime mensaje y el dato de $s0
14                 nop             # fin
```

Script 3.14: Programa utilizando un direccionamiento pseudodirecto

Observe que en la línea 7 del *script* 3.14 se utilizó el modo de direccionamiento pseudodirecto, ahora se ejecutará este programa con el depurador en la tarjeta SIE y se mostrará el código de máquina en un rango del programa donde se encuentre la línea de la instrucción del modo de direccionamiento que se está analizando.

```
1 (gdb) disassemble main
2 Dump of assembler code for function main:
3 0x00400780 <main+0>:    li      s0,0
4 0x00400784 <main+4>:    j       0x400794 <cargue>
5 0x00400788 <main+8>:    li      s1,6
6 0x0040078c <main+12>:   li      s2,2
7 0x00400790 <main+16>:   nop
8 End of assembler dump.
9 (gdb)
```

Script 3.15: Lenguaje de máquina direccionamiento pseudo

Con el código de máquina del *script* 3.15 se puede ver que usa una sola instrucción de máquina para ejecutar este modo de direccionamiento y la dirección de la etiqueta *cargue* es la que se muestra a continuación de la instrucción de salto incondicional (línea 4). Este modo de direccionamiento es muy utilizado para saltos largos donde no se necesite cumplir con alguna condición.

5 RESUMEN

Se han estudiado los modos de direccionamiento que ofrece la arquitectura MIPS como mecanismo eficiente para conocer el funcionamiento interno del procesador. No todos los cálculos de la dirección efectiva de un operando pueden realizarse en una sola instrucción de máquina, tan sólo aquellos que requieran cierto tipo de operaciones, como por ejemplo las operaciones aritméticas y lógicas.

Al programar en *assembler* la técnica para acceder a los datos es tener en cuenta qué operaciones se deben realizar para obtener su dirección y si éstas pueden ser incluidas en una misma instrucción como modo de direccionamiento. Por esto, los modos de direccionamiento son un recurso que el procesador ofrece para hacer más eficiente la ejecución de un programa.

Para hacer más eficiente la ejecución de un programa se deben tener en cuenta que los modos de direccionamiento se refieren a la manera en que los operandos están especificados dentro del formato de instrucción. Dado que el formato de instrucción es de longitud limitada (32 bits), la codificación elegida impondrá unos límites en el número de direcciones de los operandos.

6 TRABAJO DE LABORATORIO

1. Indique el tipo de direccionamiento y formato que se utiliza en cada una de las instrucciones contenidas en el programa del *script* 3.16. Este programa toma los n elementos del vector V, los suma y los guarda en memoria en la variable *SUMA*.

```
1  .data 0x00500000
2  resultado:      .ascii "La suma de los n elementos es: %d \n"
3  V:              .word 1, 5, 7, 3, 1, 4
4  SUMA:           .word 0
5  .text
6  .globl main
7  main:           li $s6, 6              # n elementos = 6
8                  li $s2, 0              # elemento i del vector suma = sum[0]
```

```
 9                  li $s3, 0               # carga el elemento i de suma = suma[i]
10                  li $s0, 0               # guarda la suma de los elementos de V[n]
11                  li $s7, -4              # inicia j en -4 de vector V = V[-4]
12
13  ciclo:          addi $s7, $s7, 4        # j=j+4
14                  lw $s4, V($s7)          # carga el elemento j en el reg $s4
15                  add $s0, $s0, $s4       # guarda el resultado de la suma en $s0
16                  sub $s6, $s6, 1         # n=n-1
17                  beqz $s6, almacena      # si n=0 salta a almacena, sino sigue
18                  j ciclo                 # salta a ciclo
19
20  almacena:       sw $s0, SUMA($s2)       # envía a memoria el resultado de la suma
21                  lw $s3, SUMA($s2)       # carga el resultado de la suma a $s3
22                  la $a0, resultado       # carga el mensaje a imprimir
23                  move $a1, $s3           # carga el resultado de la suma a imprimir
24                  jal printf              # imprime la suma de los elementos de V
25                  j nop                   # termina
```

Script 3.16: Programa ejemplo utiizando los modos de direccionamiento

2. Diseñe un programa donde utilice y señale cada uno de los modos de direccionamiento. Luego lo compila, lo envía a la tarjeta SIE y lo ejecuta.

7 PREGUNTAS DE PRUEBA

- ¿Qué relación hay entre los tipos de formato de instrucción y los modos de direccionamiento?
- ¿Por qué es importante conocer los modos de direccionamiento que maneja cierta arquitectura y qué ventajas otorgan?
- ¿Qué importancia tiene el *datapath* en los modos de direccionamiento?
- ¿Conociendo los tipos de formato de instrucción y los modos de direccionamiento, es más fácil diseñar un programa en *assembler*? Justifique.

Bibliografía

[1] Wladimir Rodríguez: *Arquitectura de Computadoras, Arquitectura del Conjunto de Instrucciones*, Escuela de Ingeniería de Sistema,Merida, VENEZUELA, (2004).

[2] John L. Hennessy, David A. Patteeson: *Organización y Diseño de Computadores. La interfaz hardware/software*, Madrid, ESPAÑA, 2 edición, McGRAW-HILL, (2003), 1-55860-281-X.

[3] José Daniel Muñoz Frías: *Estructura de Computadores, Direccionamiento y formatos*, Madrid, ESPAÑA, Universidad Pontificia Comillas. ETSI ICAI, (2003).

LABORATORIO 4

JERARQUÍA DE MEMORIA. RENDIMIENTO DE LA CACHÉ

"El hardware es lo que hace a una máquina rápida; el software es lo que hace que una máquina rápida se vuelva lenta"

Craig Bruce

1 INTRODUCCIÓN

En esta sección se estudiará uno de los temas más importantes que tiene la arquitectura de computadores como lo es la jerarquía de memoria. Esta jerarquía está compuesta por: registros en el nivel más alto, memoria caché nivel 3, memoria principal nivel 2 y discos duros en el nivel 1, teniendo como objetivo comprender el rendimiento de una memoria de gran velocidad al coste de una memoria de baja velocidad.

Se analizará cada uno de los niveles definiendo qué son, en qué se dividen y cómo están compuestos, teniendo como tema principal el rendimiento de la caché. Todo esto con el fin de entender de manera teórica y práctica qué son las memorias, qué tipos de memorias existen y cómo es su interacción con el procesador.

2 OBJETIVOS

- ⋆ Entender el propósito de la jerarquía de memoria en un procesador.
- ⋆ Conocer los principales niveles que componen la jerarquía de memoria.
- ⋆ Comprender los principios básicos de operación de la memoria caché.

⋆ Analizar el rendimiento de la memoria caché en el procesador JZ4725 por medio de la tarjeta SIE.

3 MARCO TEÓRICO

3.1 JERARQUÍA DE MEMORIA

Es la organización de los niveles de memoria que tienen los ordenadores, basándose en el principio de localidad, con el objetivo de conseguir un mejor rendimiento y posibilitar a la CPU el acceso ilimitado y rápido tanto al código como a los datos.

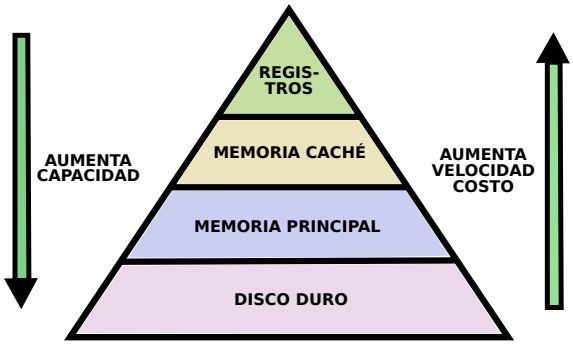

Figura 4.1: Estructura de la jerarquía de memoria.
FUENTE: El autor.

Se busca entonces contar con capacidad suficiente de memoria, con una velocidad que sirva para satisfacer el rendimiento deseado a un bajo costo. Gracias al principio de localidad, es factible utilizar una mezcla de los distintos tipos de memoria y lograr un rendimiento cercano al de la memoria más rápida [1].

Los principales niveles que componen la jerarquía de memoria son:

- *Nivel 0*: Registros.
- *Nivel 1*: Memoria caché.
- *Nivel 2*: Memoria principal.
- *Nivel 3*: Disco duro.

3.1.1 Principio de Localidad

Los programas tienden a reutilizar los datos e instrucciones que usaron recientemente, la localidad se presenta en dos dimensiones temporal y espacial [1].

- *Localidad temporal*: Si un dato es referenciado en determinado momento, es común que vuelva a ser referenciado poco tiempo después, por lo tanto mantiene los datos más recientemente accedidos cercanos al procesador.
- *Localidad espacial*: Cuando un dato es referenciado en determinado momento, es común que los datos con direcciones cercanas también sean accedidos poco tiempo después, moviendo las palabras próximas al nivel superior.

3.1.2 Memoria Caché

Es una memoria rápida y pequeña, situada en el mismo integrado junto al procesador a diferencia de la memoria principal y la CPU, especialmente diseñada para contener información que se utiliza con frecuencia en un proceso con el fin de evitar accesos a otras memorias, reduciendo considerablemente el tiempo de acceso al ser más rápida que la memoria principal.

La caché es una memoria en la que se almacena una serie de datos para su rápido acceso. La memoria caché es de tipo volátil (del tipo RAM), pero de una gran velocidad. Su objetivo es almacenar una serie de instrucciones y datos a los que la CPU accede continuamente, con el fin de que estos accesos sean lo más rápido posible [2].

3.1.3 Operación Básica de la Caché

La comunicación se lleva a cabo entre dos niveles adyacentes: la memoria caché y la memoria principal. La unidad de información que puede estar presente o no en el nivel más cercano a la CPU se denomina bloque como se muestra en la figura 4.2.

Cuando la CPU necesita acceder a la memoria principal, primero revisa la caché. Si encuentra el dato en caché, se lee rápidamente y esto es lo que se denomina como un *acierto (hit)*. Si el dato direccionado por la CPU no se encuentra en la caché se denomina como un *fallo (miss)* y esto implica que se debe acceder a la memoria principal para leerlo. Después se transfiere un bloque de la memoria principal a la memoria caché, para que las futuras referencias a memoria encuentren el dato requerido en ésta

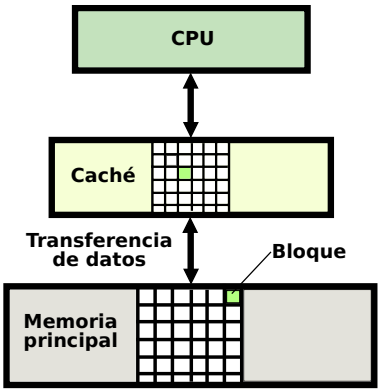

Figura 4.2: Transferencia de datos entre niveles.

FUENTE: El autor.

memoria rápida [3].

Como el objetivo de la jerarquía de memoria es brindar un mayor rendimiento a un sistema, la cantidad de aciertos y fallos juegan un papel importante y se debe tener claro los siguientes términos:

- *Tiempo de acierto*: es el tiempo necesario para acceder a un dato presente en el nivel superior de la jerarquía incluyendo el tiempo para saber si el acceso es un acierto o un fallo.
- *Tiempo de fallo*: es el tiempo de sustituir un bloque de nivel superior por uno de un nivel más bajo, más el tiempo necesario para proporcionar este nuevo bloque a la CPU.
- *Relación de aciertos*: es el porcentaje de accesos encontrados en la memoria superior (caché).
- *Relación de fallos*: es el porcentaje de accesos a la memoria no encontrados (1 - *relación de aciertos*).

4 PROCEDIMIENTO GENERAL

1. Comprender la jerarquía de memoria del procesador JZ4725 de la tarjeta SIE.
2. Estudiar los parámetros de la memoria caché del procesador JZ4725.
3. Analizar el rendimiento de la memoria caché por medio de un programa ejemplo, en el cual se aumenta el tamaño de los datos.
4. Medir el tiempo de ejecución del programa utilizando el comando *gettimeofday()*.

5. Observar mediante una gráfica de *cantidad de datos vs tiempo* el comportamiento de la memoria caché.

4.1 JERARQUÍA DE MEMORIA DEL PROCESADOR JZ4725

La jerarquía de memoria del JZ4725 es similar a la jerarquía de un ordenador, la diferencia radica en que el tamaño de las memorias es menor. El funcionamiento de la memoria caché es el mismo que se presenta en la sección 3.1.3. En la figura 4.3 se puede observar como está constituida esta jerarquía.

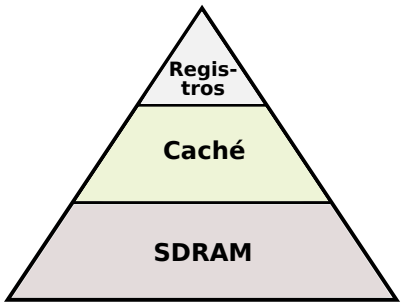

Figura 4.3: Jerarquia de memoria del JZ4725.
FUENTE: El autor.

En el primer nivel está la CPU *XBurst*, es el núcleo del procesador de alto rendimiento, bajo consumo de energía y trabaja con la arquitectura MIPS. En el segundo nivel está memoria caché y en el tercer nivel se encuentra la memoria SDRAM que cumple la misma función que la memoria principal en un ordenador.

4.2 PARÁMETROS DE CACHÉ

La memoria caché está ubicada entre el *XBurst* y la memoria SDRAM, tiene una capacidad de 16 KiB para datos y 16 KiB para instrucciones. Como la capacidad de datos es de 16 KiB, esto quiere decir que puede almacenar hasta 4096 palabras [4].

La capacidad en palabras es un parámetro muy importante en esta práctica, ya que sirve como punto de referencia en donde se tiene la certeza que es la máxima capacidad de la memoria caché, así que si está llena y se necesita acceder a un nuevo dato que no esté en la caché, necesariamente tendrá que ir a buscarlo a la memoria SDRAM, por supuesto esto le costará que se demore más tiempo.

4.3 ANÁLISIS DEL RENDIMIENTO DE LA MEMORIA CACHÉ

Para analizar el rendimiento de la memoria caché y teniendo en cuenta su capacidad máxima de 4096 palabras se ha diseñado un programa en C, en el que por medio de un ciclo *while (b<N)* se llena el vector "vec[b]" por medio de la suma de las variables "a" y "b" ($vec[b] = a + b$), luego se guarda el valor de "vec[b]" en la variable "a" ($a = vec[b]$), después se suma el elemento del vector "vec[N-b]" con la variable "a" y se guarda en la variable "d" ($d = vec[N\text{-}b] + a$), esta operación es la más importante en este programa ya que se está sumando un dato de las primeras posiciones con uno de las últimas posiciones del vector "vec". Por último, siguiendo con el programa se aumenta la variable "b" en uno, repitiéndose este procedimiento "N" veces.

En el programa se utilizan las funciones *malloc()* y *gettimeofday()*, la primera es utilizada para reservar espacio en la memoria y la segunda se usa para medir el tiempo de ejecución del programa. Estas dos funciones se describen con mayor detalle en el manual de usuario.

El programa ejemplo que se utilizará para medir el rendimiento de la memoria caché se muestra en el *script* 4.1, se digita y se guarda en un archivo con extensión *.c*.

```
1  #include <stdio.h>              /*libreria para la función printf*/
2  #include <stdlib.h>             /*libreria para la función malloc()*/
3  #include <time.h>               /*libreria para medir el tiempo*/
4  #include <sys/time.h>           /*libreria que define el timeval para medir el tiempo*/
5  #define N 100                   /*define el numero de iteraciones, tamaño del vector */
6  double timeval_diff(struct timeval *x, struct timeval *y)
7          {
8          return
9          (double)(x->tv_sec + (double)x->tv_usec/1000000) -
10         (double)(y->tv_sec + (double)y->tv_usec/1000000);       /*función para medir el*/
11         }                                                       /*tiempo en [ms]*/
12 int main(int argc, char *argv[])
13         {                                       /*declaración de variables*/
14         struct timeval t_ini, t_fin;
15         double secs;
16         int d;
17         int a= 1;
18         int b = 1;
19         int *vec;                               /*puntero para reservar memoria*/
20         vec=(int *) malloc(4*N*sizeof(int));    /*asignación de memoria*/
21                 gettimeofday(&t_ini, NULL);     /*inicio del cronometro*/
22                         while(b<=N)
23                                 {
```

```
24                                  vec[b]=b;
25                                  vec[b]=vec[N-b]+vec[b]; /*operación que exige la caché*/
26                                  b++;
27                                  }
28                  gettimeofday(&t_fin, NULL);             /*fin del cronometro*/
29          secs = timeval_diff(&t_fin, &t_ini);
30          printf("%.16g milliseconds\n", secs * 1000.0);  /*tiempo de ejecución en [ms]*/
31          free(vec);                                      /*liberación de variables*/
32          return 0;
33          }
```

Script 4.1: Programa ejemplo

Por medio del programa del *script* 4.1 se puede analizar el rendimiento de la memoria caché a medida que se varia el tamaño de la variable "N", por medio de la función *gettimeofday* se mide y muestra el tiempo cada vez que se varia "N" con el fin de graficar los resultados.

Teniendo el programa en el archivo *.c*, el paso a seguir es compilarlo y enviar el ejecutable a la tarjeta SIE. Los pasos para este procedimiento están descritos en el manual de usuario.

4.4 RESULTADOS

Cada vez que se cambia el valor de la variable "N" se obtiene un valor diferente de tiempo. Obtener estos valores y colocarlos en la tabla 4.1.

Datos [N]	**Tiempo [ms]**
100	
500	
1000	
1500	
2000	
2500	
3000	
3500	
4000	
4100	
4200	
4300	
4500	
5000	
5500	
6000	
6500	
7000	

Tabla 4.1: Datos y tiempos

A partir de los resultados obtenidos en la tabla 4.1, graficar para observar como es el comportamiento del tiempo a medida que se aumenta la cantidad de datos "N" en la caché.

Una vez se obtiene la grafica analizar y sacar las conclusiones respecto a esta actividad propuesta.

5 RESUMEN

La jerarquía de memoria es un sistema de administración de memorias. Es un concepto de los más importantes en el ámbito de la asignatura de arquitectura de computadores. Se clarifican distintos aspectos relacionados como la estructura y el funcionamiento de la jerarquía (CPU, memoria caché, memoria principal, discos duros), explicándose primero de manera teórica, y posteriormente se realizan un ejercicio práctico basado en programas que permiten observar el rendimiento de la memoria caché, todo esto implementado en la tarjeta SIE.

Por medio de una gráfica se analiza los resultados obtenidos, observando que la memoria caché aporta grandes beneficios al desempeño, ya que aprovecha la velocidad de la CPU cuando ejecutamos cualquier programa.

6 TRABAJO DE LABORATORIO

1. ¿Por qué es importante la jerarquía de memoria?
2. ¿Que pasaría si un sistema no tuviera memoria caché?
3. De acuerdo al programa del *script* 4.1. Explique. Qué pasaría si al ciclo *while* se le agrega nuevo vector y otra operación que exija la caché. Como la siguiente:

```
1        M[b]=b;
2        M[b]=M[N-b]+M[b];
```

Script 4.2: Vector a adicionar

4. Implemente el programa modificado en la tarjeta SIE, aumente el tamaño de la variable "N", mida los tiempos y grafique.
5. ¿En qué valor de "N" y por qué se desborda la caché?
6. ¿Qué se puede concluir de lo anterior?

Bibliografía

[1] John L. Hennessy, David A. Patteeson: *Organización y Diseño de Computadores. La interfaz hardware/software*, Madrid, ESPAÑA, 2 edición, McGRAW-HILL, (2003), 1-55860-281-X.

[2] Stallings, Williams: *Computer Organization Architecture*, Pearson Education International, New Jersey, USA, (2003),0-13-049307-4.

[3] Carballal Abarzúa Claudio: *Estudio, Análisis y Modelado de Memorias Caché Compartidas Bajo Administración Dinámica*,Facultad de ingeniería, Universidad de Buenos Aires, Buenos Aires, ARGENTINA, (2011).

[4] Datasheet JZ4725, *multimedia Application processor*, Ingenic Semiconductor Co. Ltd, Revision: 1.0,May 2009

ANEXO B. MANUAL DE USUARIO DE LA TARJETA SIE LABORATORIO ARQUITECTURA DE COMPUTADORES

CAPÍTULO 1

GUIA DE CONFIGURACIÓN DE LA PLATAFORMA SIE

1.1 INTRODUCCIÓN

Para poder empezar a interactuar con la tarjeta SIE se hace necesario tener instalado en nuestro computador un conjunto de herramientas que permitirán construir aplicaciones para la plataforma SIE, bien sea el caso de una fuente escrita en *assembler*, C o C++ el proceso de compilación se debe realizar para la arquitectura del procesador en el cual se ejecutará la aplicación, de lo contrario no funcionará. La plataforma SIE contiene un procesador MIPS el cual no permite que un archivo fuente sea compilado dentro de él, ya que no fue diseñado para ese fin, el código fuente se deberá compilar en nuestro computador con un conjunto de herramientas destinadas para tal fin y luego será pasado a la tarjeta para la ejecución en ella por medio de un compilador cruzado. [1] Este conjunto de herramientas se basa en la distribución de *Linux OpenWrt* [2] muy utilizada en sistemas embebidos.

1.2 INSTALACIÓN DE LAS HERRAMIENTAS DE COMPILACIÓN PARA LA PLATAFORMA SIE

1.2.1 Descarga e instalación del conjunto de herramientas

Antes de empezar a instalar es necesario que el sistema operativo disponga de ciertos paquetes. Se procede a descargarlos escribiendo el siguiente comando [1].

```
1 $ cd ~
2 $ sudo apt-get install sed wget cvs subversion git-core coreutils unzip texi2html
3 texinfo libsdl1.2-dev docbook-utils gawk python-pysqlite2 diffstat help2man make gcc
4 build-essential g++ desktop-file-utils chrpath flex libncurses5 libncurses5-dev
5 libxml-simple-perl zlib1g-dev pkg-config gettext libxml-simple-perl guile-1.8 cmake
```

Script 1.1: Descarga de paquetes necesarios

[1] Un compilador cruzado es capaz de crear código ejecutable para otra arquitectura distinta. Esta herramienta es útil para la plataforma SIE, ya que no se puede compilar código fuente en la tarjeta.

[2] OpentWrt es una distribución de *Linux* derivada de *Debian* diseñado para sistemas embebidos.

Descargar el directorio llamado *trunk* (éste directorio es una imagen del *software* que contiene las fuentes de *OpenWrt* y estará ubicado dentro de la carpeta de usuario).

```
1 $ svn checkout --revision=28395 svn://svn.openwrt.org/openwrt/trunk/
```

Script 1.2: Descarga de las fuentes de *OpenWrt*

Ingresar al directorio que se ha descargado.

```
1 $ cd trunk
```

Script 1.3: Entrar al directorio

Dentro del directorio se encuentra un archivo llamado *feeds.conf.default*, dentro de éste archivo hay una línea que dice:

```
1 src-svn packages svn://svn.openwrt.org/openwrt/packages
```

Script 1.4: línea a modificar

Ésta debe ser modificada para que quede de la siguiente forma:

```
1 src-svn packages svn://svn.openwrt.org/openwrt/packages@28395
```

Script 1.5: línea a modificar

El paso anterior permite compilar una versión de *OpenWrt* estable y no presenta ningún problema, hay que tener en cuenta que se está trabajando con una versión en desarrollo, así que *trunk* está en constante cambio, por esta razón puede que en ciertas ocasiones presente problemas.

OpenWrt utiliza *feeds* que proporcionan paquetes de software en el sistema, para incluirlos al sistema se ejecutan los siguientes comandos:

```
1 $ ./scripts/feeds update -a
2 $ ./scripts/feeds install -a
3 $ make defconfig
```

Script 1.6: Actualización de los *feeds*

1.2.2 Incluir paquetes de *software*

Para agregar o quitar paquetes de la imagen de *software*, se puede utilizar el menú interactivo de *OpenWrt*. Este menú permite de forma gráfica seleccionar los paquetes de *software* y librerías que se desee incluir dentro de la imagen de *software*. En este caso se mostrará como incluir los paquetes necesarios para poder trabajar en la asignatura arquitectura de computadores.

```
1 $ make menuconfig
```

Script 1.7: Configuración del menú interactivo de *OpenWrt* a compilar

Enseguida se desplegará la siguiente ventana (figura 1.1).

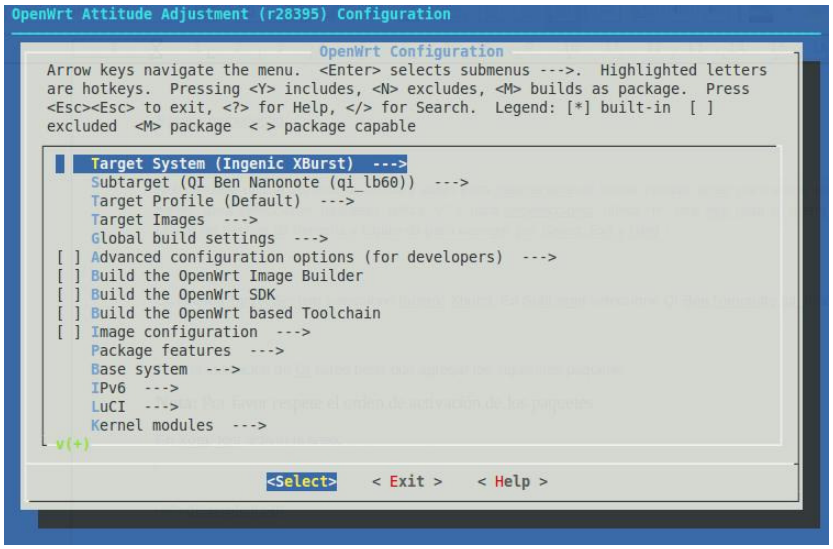

Figura 1.1: Menú de selección de paquetes.

Ingresar al menú ***Target System*** y seleccionar ***Ingenic Xburst***. En ***Subtarget*** seleccione ***QI Ben Nanonote (qi_lb60)***. Ingrese en el submenú ***Xorg***, después ingrese a ***font*** y active la línea ***dejavu-fonts-ttf***, Se desplegarán muchas mas opciones, active todas aquellas que tengan que ver con ***dejavu*** (figura 1.2).

Para el funcionamiento del depurador GDB active las siguientes librerías (figuras 1.3, 1.4 y 1.5).

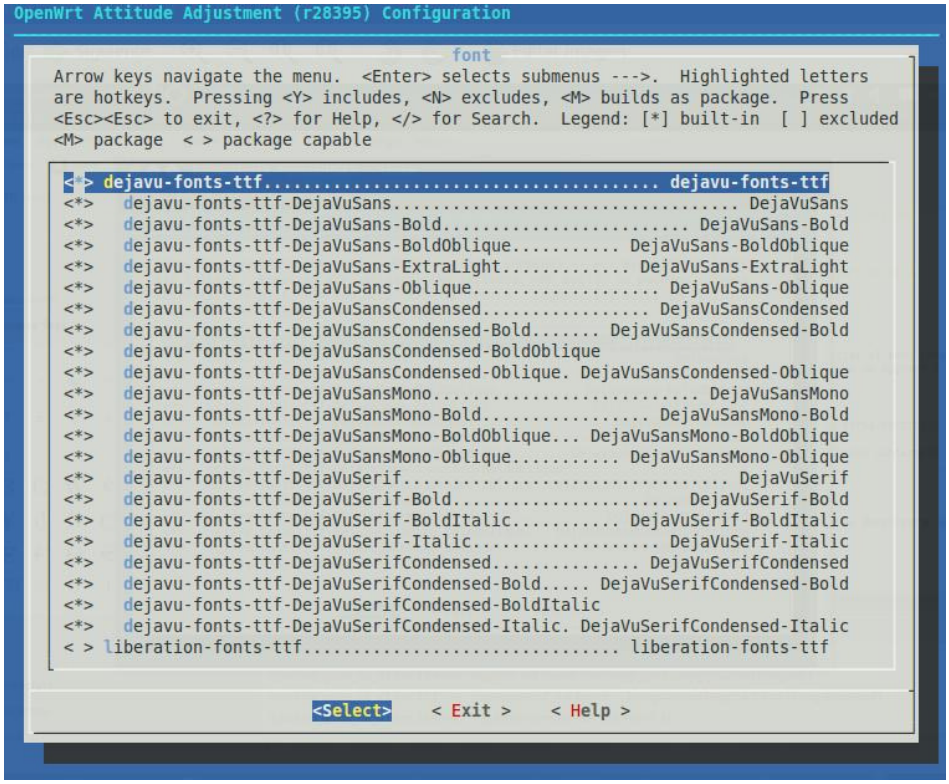

Figura 1.2: Activación de fuentes (Tipos de letras).

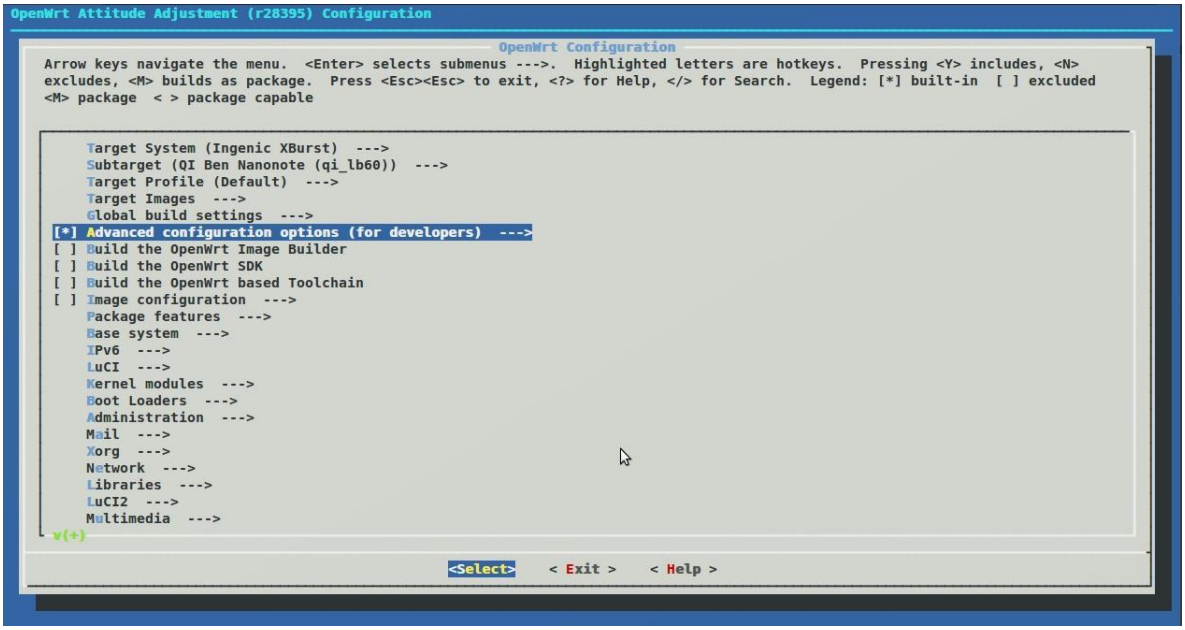

Figura 1.3: Activación de gdb paso 1.

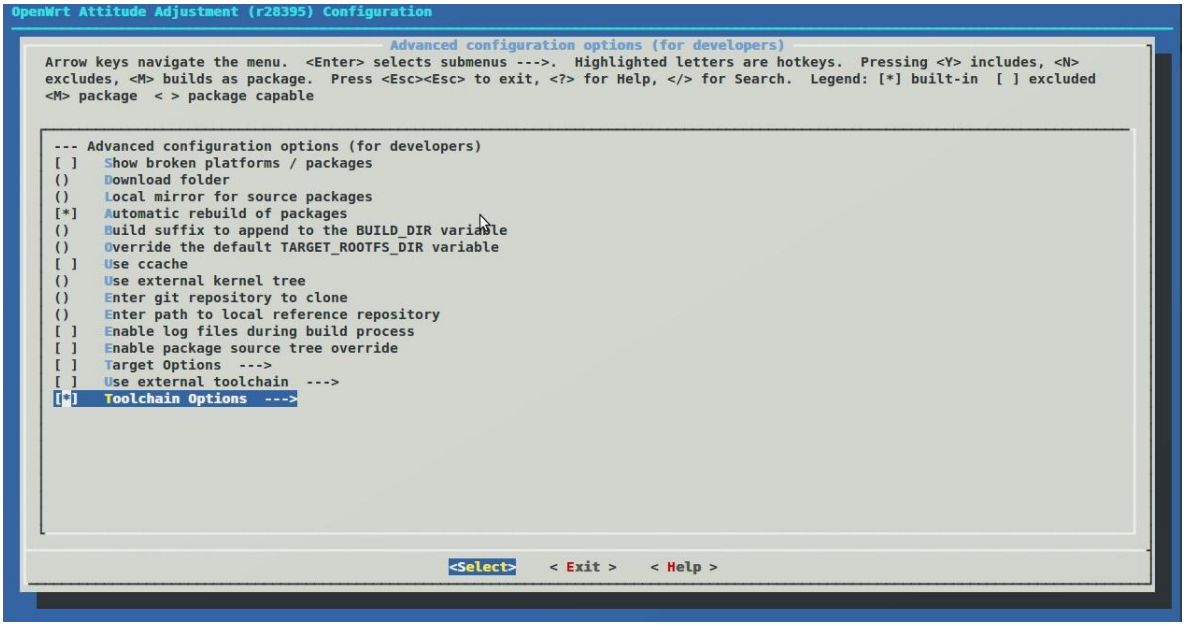

Figura 1.4: Activación de gdb paso 2.

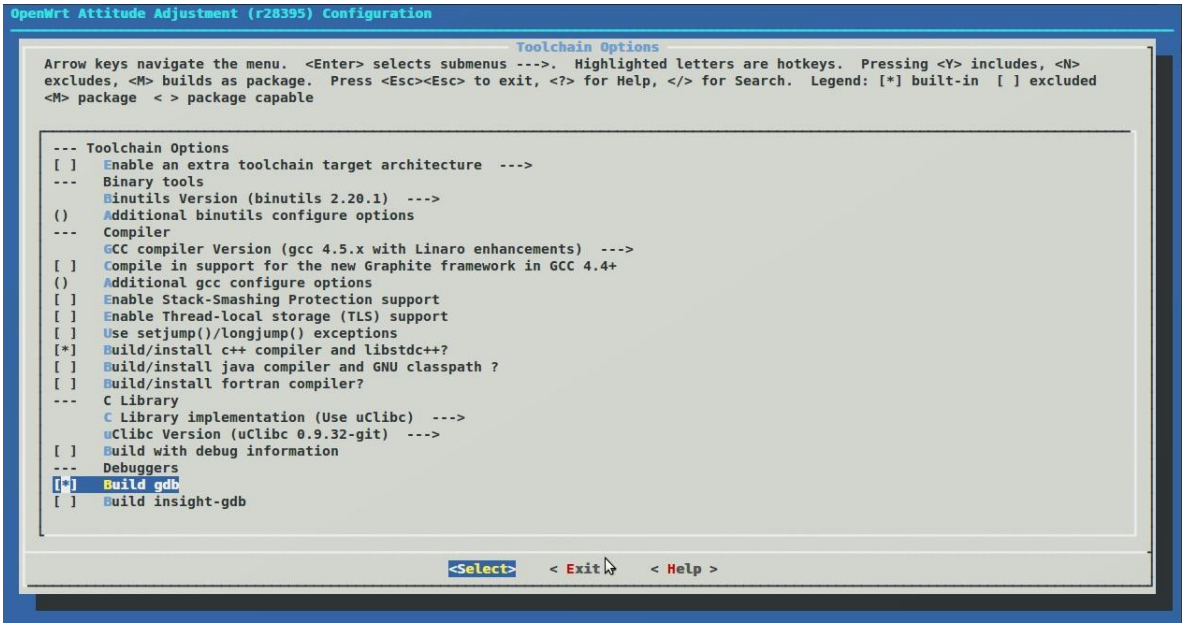

Figura 1.5: Activación de gdb paso 3.

Volver al menú principal entrar a ***Utilities*** seleccione ***gdb*** y ***gdbserver***.

Para compilar la imagen de *software*, es solo cuestión de dar la orden *make -j # V=99* aunque el argumento # es de decisión personal se recomienda usar el número de núcleos disponibles de su computador + 1, por ejemplo en un computador de un solo núcleo usted podría usar *-j2* y V=99 para mostrar todos los mensajes del proceso de compilación.

```
1 $ make -j2 V=99
```

Script 1.8: Compilación de herramientas de OpenWrt

El proceso de compilación de la imagen de *software* por primera vez toma de 2-4 horas, dependiendo de la cantidad de librerías que se van a compilar y de la velocidad del procesador del computador.

Cuando el proceso de compilación termina, en la ruta **/PATH/TO/trunk/bin/xburst** se pueden encontrar los archivos de *flasheo* de la plataforma SIE, específicamente ***openwrt-xburst-qi_lb60-root.ubi*** (imagen del Sistema de Archivos Raíz) y ***openwrt-xburst-qi_lb60-uImage.bin*** (imagen del kernel), estos archivos serán necesarios mas adelante. En la dirección **/PATH/TO/trunk/staging_dir/toolchain-mipsel_gcc-4.5-linaro_uClibc-0.9.32/bin** se encuentran los compiladores necesarios que permiten compilar código fuente para el plataforma SIE. Comprobar que estén estos archivos, en caso de no estar se debe repetir el proceso desde la inclusión de paquetes.

Nota: /PATH/TO hace referencia al directorio de usuario donde usted descargó el directorio *trunk*.

El siguiente paso es agregar la dirección donde se encuentran los comandos del compilador cruzado. Dentro de la carpeta de usuario se ejecuta lo siguiente:

```
1 $ cd ~ && sudo gedit .bashrc
```

Script 1.9: Modificación *.bashrc*.

Y al final del archivo agregue la siguiente línea:

```
1 export PATH=${PATH}:/PATH/TO/trunk/staging_dir/toolchain-mipsel_gcc-4.5-linaro_uClibc-0.
2                       9.32/bin
```

Script 1.10: Dirección de los comando propios al compilador cruzado.

1.3 FLASHEO DEL KERNEL Y DEL SISTEMA DE ARCHIVOS DE LA TARJETA SIE

Cuando se descarga el sistema de archivos basado en OpenWrt, por medio del procedimiento anterior se descargó la distribución con la versión 2.6.37.6 de las fuentes del kernel, es posible que la imagen del kernel de la tarjeta con la que se vaya a trabajar tenga una versión diferente a la de las fuentes del kernel del sistema de archivos descargado, en el momento de compilar en nuestro computador no habrá problema pero en el momento de utilizar el código en la tarjeta SIE no funcionará, ya que las versiones del kernel son diferentes. Además la imagen del sistema de archivos que actualmente tiene instalada la tarjeta puede que no tenga las librerías o paquetes que se seleccionó en el momento de compilar el sistema de archivos de *OpenWrt*, esto puede traer problemas al momento de ejecutar alguna aplicación en GDB específicamente, ya que no va poder encontrar las librerías necesarias para la aplicación. Para solucionar estos inconvenientes se debe *flashear* la imagen del kernel y la del sistema de archivos de la plataforma SIE [2].

1.3.1 Instalación de las herramientas necesarias para el *flasheo*

Primero que todo se debe conectar la tarjeta SIE al computador. Colocar un *jumper* en los pines del conector J15 y presionar *reset*. Para poder *flashear* se debe instalar las herramientas *xburst-tool* de la siguiente manera:

```
1  $ cd /tmp/
2  /* Para procesadores de 32 bits */
3  $ wget http://projects.qi-hardware.com/media/upload/xburst-tools/files/xburst-tools_0.
4    0+201004-0.1_i386.deb
5  /* Para procesadores de 64 bits */
6  $ wget http://projects.qi-hardware.com/media/upload/xburst-tools/files/xburst-tools_0.
7    0+201004-0.1_amd64.deb
8  $ sudo apt-get install --no-install-recommends libconfuse0
9  $ sudo dpkg -i xburst-tools_0.0+201004-0.1_xxxx.deb          /* (xxxx = i386 ó amd64) */
10 $ sudo dpkg -L xburt-tools | grep bin/
```

Script 1.11: Comandos para instalar las herramientas *xburst-tool*.

Al ejecutar la linea anterior debe salir en consola las líneas que se muestran en el *script* 1.13, en caso de no salir estas dos líneas se vuelve al inicio del proceso de *flasheo*.

```
1  /usr/bin/usbboot
```

```
2 /usr/bin/xbboot
```

Script 1.12: Imagen de *flasheo*

Esta herramienta permitirá programar la memoria NAND de la SIE, específicamente el programa *usbboot*.

1.3.2 Creación del directorio y copia de archivos para el *flasheo*

En el directorio de usuario crear una nueva carpeta (*flashear_SIE* por ejemplo), entrar a éste directorio y copiar los archivos necesarios para el *flasheo* de la tarjeta (*script* 1.13).

```
1 /* Imagen del kernel */
2 $ cp /PATH/TO/trunk/bin/xburst/openwrt-xburst-qi_lb60-uImage.bin /PATH2/TO/flashear_SIE/
3 /* Imagen del sistema de archivos */
4 $ cp /PATH/TO/trunk/bin/xburst/openwrt-xburst-qi_lb60-root.ubi /PATH2/TO/flashear_SIE/
```

Script 1.13: Imagen de *flasheo*

Descargar el archivo de configuración de la pagina http://projects.qi-hardware.com/index.php/p/nn-usb-fpga/source/tree/master/ y copiarlo en el directorio *flashear_SIE*.

Crear un archivo que se llame *flash_kernel.sh* y escribir las siguientes líneas:

```
1 #!/bin/bash
2 sudo usbboot -f ./usbboot_2gb_nand.cfg -c "boot"
3 sudo usbboot -f ./usbboot_2gb_nand.cfg -c "nprog 1024 openwrt-xburst-qi_lb60-uImage.
4 bin 0 0 -n"
```

Script 1.14: *Script* para la imagen del *kernel*

Crear otro archivo llamado *flash_rootfs.sh* y escribir lo siguiente:

```
1 #!/bin/bash
2 ROOTFS=openwrt-xburst-qi_lb60-root.ubi
3 sudo usbboot -f ./usbboot_2gb_nand.cfg -c "boot"
4 sudo usbboot -f ./usbboot_2gb_nand.cfg -c "nerase 16 512 0 0"
5 sudo usbboot -f ./usbboot_2gb_nand.cfg -c "nprog 2048 $ROOTFS 0 0 -n"
```

Script 1.15: *Script* para la imagen del sistema de archivos

La linea *sudo usbboot -f ./usbboot_2gb_nand.cfg -c "boot"* en ambos archivos carga el programa *usbboot* en la RAM de la tarjeta y la última línea indica cuales archivos debe programar en la RAM, en este

caso es la imagen del kernel y la imagen del sistema de archivos. A continuación se le cambian los permisos a los *scripts* para que puedan ser utilizados:

```
1 $ chmod 777  flash_kernel.sh flash_rootfs.sh
```

Script 1.16: Habilitación de permisos

1.3.3 *Flasheo* de la plataforma SIE

Luego se ejecutan los dos *scripts* de la siguiente manera:

```
1 $ ./flash_kernel.sh
2 $ ./flash_rootfs.sh
```

Script 1.17: Configuración las imágenes del kernel y del sistema de archivos.

En seguida se quita el *jumper* del conector J15 y se oprime *reset*.

Como se ha instalado un nuevo sistema de archivos se debe crear una contraseña para poder ingresar a la tarjeta SIE por medio de los siguientes comandos:

```
1 $ ifconfig
2 $ sudo ifconfig usb0 192.168.1.2 up
3 $ telnet 192.168.1.1
4 $ passwd
```

Script 1.18: Cambiar clave

Entrar a la tarjeta y colocar la clave, para evitar conflictos con otros cursos la clave debe ser *enter* (oprimir *enter*), luego salir y entrar a la tarjeta.

1.3.4 Comunicación con la tarjeta SIE

Para ejercer comunicación con la tarjeta SIE se deben seguir los siguientes pasos:

- Abrir una consola y digitar *sudo ifconfig usb0 192.168.1.2 up*, esto se hace para establecer comunicación con la tarjeta.
- Para comprobar si hay comunicación se digita *ping 192.168.1.1*, para detener el *ping* se hace con *Ctrl c*.
- Si hay comunicación, se procede entrar a la tarjeta digitando *ssh root@192.168.1.1* y *enter*.

Si no se puede entrar digitar lo siguiente.

```
1 telnet 192.168.1.1
2 ssh-keygen -R 192.168.1.1
3 ssh -l root 192.168.1.1
4 ssh root@192.168.1.1
```

Script 1.19: Proceso para entrar a la tarjeta SIE

Aquí termina el proceso de *flasheo*. Si todo ha salido bien la plataforma SIE está lista para el desarrollo de las prácticas de la asignatura arquitectura de computadores.

En algunas algunas tarjetas el proceso de *flasheo* no funciona. Para estos casos se utilizará una carpeta llamada SIE que será suministrada por el profesor a cargo de la asignatura. Ésta carpeta contiene los repositorios de la plataforma SIE.

Copiar el directorio SIE en la carpeta de usuario del computador, dentro de este directorio hay otro llamado *reflash*, entrar a este último y copiar los archivos *load_u-boot* y *openwrt-xburst-u-boot.bin* en la carpeta que se había creado para el *flasheo* (*flashear_SIE*). Estos dos archivos son utilizados para configurar el sistema de arranque de la tarjeta SIE.

```
1 $ cd /SIE/reflash/
2 $ cp load_u-boot openwrt-xburst-u-boot.bin flashear_SIE/
```

Script 1.20: Archivos para la configuración del sistema de arranque

luego se ejecuta el archivo *load_u-boot* para configurar el sistema de arranque de la tarjeta SIE.

```
1 $ ./load_u-boot
```

Script 1.21: Configuración el sistema de arranque.

Después de hacer esto se sigue el proceso de *flasheo* desde el paso del *script* 1.17.

1.4 COMPILACIÓN DE LA HERRAMIENTA XC3SPROG

La herramienta *xc3sprog* permite programar el FPGA *Spartan* 3 de *Xilinx*. Cuando se realiza el *flasheo* de la tarjeta se eliminan todos los programas antes instalados entre ellos *xc3sprog* y para volver a instalarlo se debe realizar los pasos descritos en el *script* 1.22 [3].

Ingresar al directorio de usuario y descargar el código fuente de *xcs3prog* (línea 1 y 2), entrar al directorio donde se encuentra el código fuente de *xc3sprog* (línea 3).

```
1 $ cd ~
2 $ git clone git://projects.qi-hardware.com/nn-usb-fpga.git
3 $ cd nn-usb-fpga/Software/xc3sprog
```

Script 1.22: Compilación de *xc3sprog*

Dentro del directorio *xc3sprog* existe un archivo llamado *devicedb.cpp* al que se debe agregar la librería que está en el *script* 1.23 y compilar *xc3sprog* con el comando *make*.

```
1 #include <stdio.h>
```

Script 1.23: Agregar librería al archivo *devicedb.cpp*

Obteniendo el siguiente mensaje:

```
1 $ mipsel-openwrt-linux-g++  -Wall -lstdc++ xc3sprog.o jtag.o iobase.o sakcXCProgrammer.
2   o iodebug.o bitfile.o devicedb.o progalgxcf.o progalgxc3s.o jz47xx_gpio.o -o xc3sprog
```

Script 1.24: Mensaje de instalación

Si no se obtiene el mensaje anterior se debe hacer el proceso de compilación de la herramienta *xcs3prog* desde el inicio.

Abrir una nueva consola, conectar e ingresar a la tarjeta por medio de los comandos necesarios para esto y crear el siguiente directorio:

```
1 $ mkdir /usr/share/xc3sprog
```

Script 1.25: Crear directorio en SIE

En la otra consola copiar el ejecutable que se creó llamado *xc3sprog* y el archivo *devlist.txt* en la tarjeta SIE:

```
1 $  scp xc3sprog root@192.168.254.101:/usr/bin
2 $  scp devlist.txt root@192.168.254.101:/usr/share/xc3sprog
```

Script 1.26: Envío de archivos a la plataforma SIE

Con el anterior procedimiento se puede programar el FPGA de la plataforma SIE.

Bibliografía

[1] Linux en caja: http://linuxencaja.net/wiki/Filesystem_and_reflashing/, *Filesystem and reflashing.*

[2] Qi Hardware: http://en.qi-hardware.com/, *Xburst-tools.*

[3] Qi Hardware: http://linuxencaja.net/, *Compilación de la herramienta xc3sprog.*

CAPÍTULO 2

PROGRAMACIÓN BÁSICA EN LENGUAJE ENSAMBLADOR, ARQUITECTURA MIPS

A continuación se describe una versión más completa de las instrucciones de la arquitectura MIPS. También se describe el procedimiento para compilar programas en *assembler* por medio del compilador cruzado, enviar a la tarjeta SIE el ejecutable y por último ejecutarlo.

2.1 MANUAL DE INSTRUCCIONES MIPS.

La siguiente descripción permite lograr una visión global del repertorio de instrucciones completo, brindándole al estudiante la posibilidad de comenzar a programar con todo el repertorio a disposición.

Para un mejor conocimiento del repertorio de instrucciones, cada instrucción se presentará mediante un ejemplo y su respectiva explicación, así se comprenderán y se le dará una mejor utilidad dentro de el diseño de un programa. A continuación se presentará una serie de tablas con una versión completa del repertorio de instrucciones para la arquitectura MIPS [1].

Instrucciones Aritméticas		
Instrucción	**Ejemplo**	**Acción**
add	add $t0, $t1, $t2	Suma $t1 con $t2 y lo coloca en $t0
addi	addi $t0, $t1, k	Suma $t1 con la constante k y lo coloca en $t0
addu	addu $t0, $t1, $t2	Suma $t1 con $t2 y lo coloca en $t0 (sin desbordamiento[1])
addiu	addiu $t0, $t1, k	Suma $t1 con la constante k y lo coloca en $t0, (sin desbordamiento)
clo	clo $t0, $t1	cuenta los unos de $t1 y lo coloca en $t0.
clz	clz $t0, $t1	cuenta los ceros de $t1 y lo coloca en $t0.
negu	negu $t0, $t1	Coloca en $t0 el negativo del entero contenido en el registro $t1
sub	sub $t0, $t1, $t2	Resta $t1 - $t2 y lo coloca en $t0
subu	subu $t0, $t1, $t2	Resta $t1 - $t2 y lo coloca en $t0
mul	mul $t0, $t1, $t2	Coloca el producto de $t1*$t2 en el registro $t0
mult	mult $t1, $t2	Coloca el producto de $t1*$t2 en el acumulador de 64 bit *acc ($lo y $hi)*
multu	multu $t1, $t2	Coloca el producto de $t1*$t2 en el acumulador de 64 bit *acc ($lo y $hi)*
div	div $t1, $t2	$lo=$t1/$t2; $hi=$t1 mod $t2
divu	divu $t0, $t1, $t2	$lo=$t1/$t2; $hi=$t1 mod $t2 (sin desbordamiento)

Tabla 2.1: Instrucciones aritméticas

[1]Se produce un desbordamiento cuando se supera el rango de representación de un número binario teniendo en cuenta la notación y el número de bits utilizados.

Instrucciones Lógicas		
Instrucción	**Ejemplo**	**Acción**
and	and $t0, $t1, $t2	Realiza un *and* lógico entre los contenidos de $t1 y $t2, y lo guarda en $t0
andi	andi $t0, $t1, k	Realiza un *and* lógico entre los contenidos de $t1 y k, y lo guarda en $t0
nop	nop	No hace ninguna acción
or	or $t0, $t1, $t2	Realiza un *or* lógico entre los contenidos de $t1 y $t2, y lo guarda en $t0
ori	ori $t0, $t1, k	Realiza un *or* lógico entre los contenidos de $t1 y k, y lo guarda en $t0
not	not $t0, $t1	Realiza un *not* lógico del contenido de $t1 y lo guarda en $t0
nor	nor $t0, $t1, $t2	Realiza un *nor* lógico entre los contenidos de $t1 y $t2, y lo guarda en $t0
xor	xor $t0, $t1, $t2	Realiza un *xor* lógico entre los contenidos de $t1 y $t2, y lo guarda en $t0
xori	xori $t0, $t1, k	Realiza un *xor* lógico entre los contenidos de $t1 y k, y lo guarda en $t0

Tabla 2.2: Instrucciones lógicas

Instrucciones Carga y Almacenamiento		
Instrucción	**Ejemplo**	**Acción**
la	la $t0, dir	Carga en $t0 la dirección dir (no su contenido)
li	li $t0, inm	Carga valor inm en el registro $t0.
lui	lui $t0, kte	Carga la mitad inferior de la kte en la mitad superior del registro $t0
lb	lb $t0, dir	Carga en $t0 el byte en memoria direccionado por dir
lbu	lbu $t0, dir	Carga en $t0 el byte en memoria direccionado por dir (extiende el signo)
lh	lh $t0, dir	Carga en $t0 la mitad inferior de la palabra en memoria direccionada por dir
lhu	lhu $t0, dir	Carga en $t0 la mitad inferior de la palabra en memoria direccionada por dir
lw	lw $t0, dir	Carga en $t0 la palabra en direccionada por dir
lwl	lwl $t0, dir	Carga en $t0 los bytes izq de la palabra direccionada por dir (no alineada)
lwr	lwr $t0, dir	Carga en $t0 los bytes der de la palabra direccionada por dir (no alineada)
sb	sb $t0, dir	Almacena el byte menos significativo de $t0 en la posición dada por dir
sh	sh $t0, dir	Almacena la mitad derecha de la palabra de $t0 en la posición dada por dir
sw	sw $t0, dir	Almacena la palabra del registro $t0 en la posición de memoria dada por dir
swl	swl $t0, dir	Almacena los bytes izquierdos de la palabra de $t0 en la posición dada por dir
swr	swr $t0, dir	Almacena los bytes derechos de la palabra de $t0 en la posición dada por dir
ulw	ulw $t0, dir	Carga en $t0 la palabra no alineada direccionada por dir
usw	usw $t0, dir	Almacena la palabra en $t0 en la posición posiblemente no alineada dada por dir
move	move $t0, $t1	Copia el contenido del registro $t1 y lo coloca en el registro $t0

Tabla 2.3: Instrucciones de carga/almacenamiento

Instrucciones de Comparación		
Instrucción	**Ejemplo**	**Acción**
seq	seq $t0, $t1, $t2	$t0 = 1 si $t1 = $t2 de lo contrario $t0 = 0
sge	sge $t0, $t1, $t2	$t0 = 1 si $t1 >= $t2 de lo contrario $t0 = 0
sgt	sgt $t0, $t1, $t2	$t0 = 1 si $t1 > $t2 de lo contrario $t0 = 0
sle	sle $t0, $t1, $t2	$t0 = 1 si $t1 <= $t2 de lo contrario $t0 = 0
slt	slt $t0, $t1, $t2	$t0 = 1 si $t1 < $t2 de lo contrario $t0 = 0
slti	slti $t0, $t1, kte	$t0 = 1 si $t1 < kte de lo contrario $t0 = 0
sltiu	sltiu $t0, $t1, kte	$t0 = 1 si $t1 < kte (entero sin sgno), de lo contrario $t0 = 0
sltu	sltu $t0, $t1, $t2	$t0 = 1 si $t1 < $t2 entero sin sgno), de lo contrario $t0 = 0
sne	sne $t0, $t1, $t2	$t0 = 1 si $t1 es $\neq$ a $t2 de lo contrario $t0 = 0
movn	movn $t0, $t1	Copia el contenido del registro $t1 y lo coloca en $t0, si $t2 es $\neq$ a 0
movn	movn $t0, $t1	Copia el contenido del registro $t1 y lo coloca en $t0, si $t2 = 0

Tabla 2.4: Instrucciones de comparación

Instrucciones de Salto		
Instrucción	**Ejemplo**	**Acción**
b	b dir	Salto incondicional a la instrucción etiquetada dir
bal	bal dir	Salta a la instrucción etiquetada dir y guarda la dirección siguiente en $ra
beq	beq $t0, $t1, salt	Salta a la etiqueta salt si $t0 = $t1 sino ejecuta la siguiente instrucción
beqz	beqz $t0, salt	Salta a la etiqueta salt si $t0 = 0 sino ejecuta la siguiente instrucción
bge	bge $t0, $t1, salt	Salta a la etiqueta salt si $t0 >= $t1 sino ejecuta la siguiente instrucción
bgez	bgez $t0, salt	Salta a la etiqueta salt si $t0 >= 0 sino ejecuta la siguiente instrucción
bgezal	bgezal $t0, salt	Salta a la etiqueta salt si $t0 >= 0 y guarda la direccion siguiente en $ra
bgt	bgt $t0, $t1, salt	Salta a la etiqueta salt si $t0 > $t1 sino ejecuta la siguiente instrucción
bgtz	bgtz $t0, salt	Salta a la etiqueta salt si $t0 > 0 sino ejecuta la siguiente instrucción
bgtzal	bgtzal $t0, salt	Salta a la etiqueta salt si $t0 > 0 y guarda la dirección siguiente en $ra
ble	ble $t0, $t1, salt	Salta a la etiqueta salt si $t0 <= $t1 sino ejecuta la siguiente instrucción
blez	blez $t0, salt	Salta a la etiqueta salt si $t0 <= 0 sino ejecuta la siguiente instrucción
blt	blt $t0, $t1, salt	Salta a la etiqueta salt si $t0 < $t1 sino ejecuta la siguiente instrucción
bltz	bltz $t0, salt	Salta a la etiqueta salt si $t0 < 0 sino ejecuta la siguiente instrucción
bne	bne $t0, $t1, salt	Salta a la etiqueta salt si $t0 es $\neq$ a $t1, sino ejecuta la siguiente instrucción
bnez	bnez $t0, salt	Salta a la etiqueta salt si $t0 es $\neq$ a 0 sino ejecuta la siguiente instrucción
j	j dir	Salto incondicional a la instrucción etiquetada por dir
jr	jr $t0	Salta a la dirección contenida en el registro $t0
jal	jal dir	Salta a la instrucción etiquetada dir y guarda la dirección siguiente en $ra
jalr	jalr $t0, $t1	Salta a la dirección contenida en $t1 y guarda la dirección siguiente en $t0

Tabla 2.5: Instrucciones de salto

Instrucciones de Rotación		
Instrucción	**Ejemplo**	**Acción**
rotvr	rotvr $t0, $t1, $t2	Rota el contenido de $t1 a la derecha según indica $t2 y lo guarda en $t0
rotvl	rotvl $t0, $t1, $t2	Rota el contenido de $t1 a la izquierda según indica $t2 y lo guarda en $t0
rotr	rotr $t0, $t1, kte	Rota el contenido de $t1 a la derecha según la kte y lo guarda en $t0
rotl	rotl $t0, $t1, kte	Rota el contenido de $t1 a la izquierda según la kte y lo guarda en $t0
sll	sll $t0, $t1, $t2	Desplaza $t1 a la izquierda según lo indicado por $t2 y lo guarda en $t0
sllv	sllv $t0, $t1, $t2	Desplazamiento variable de $t1 a la izquierda según $t2 y lo guarda en $t0
sra	sra $t0, $t1, $t2	Desplazamiento aritmético de $t1 a la derecha según $t2 y lo guarda en $t0
srav	srav $t0, $t1, $t2	Desplazamiento variable de $t1 a la derecha según $t2 y lo guarda en $t0
srl	sra $t0, $t1, $t2	Desplaza $t1 a la derecha según lo indicado por $t2 y lo guarda en $t0
srlv	srav $t0, $t1, $t2	Desplazamiento variable de $t1 a la derecha según $t2 y lo guarda en $t0

Tabla 2.6: Instrucciones de Rotación

2.2 COMPILACIÓN DE PROGRAMAS EN ASSEMBLER

El programa escrito en un lenguaje de programación es llamado programa fuente y no se puede ejecutar directamente en una computadora. La opción más común es compilar el programa obteniendo un archivo objeto y a partir de este se obtiene el ejecutable.

Normalmente la creación de un programa ejecutable conlleva dos pasos. El primer paso se llama compilación (propiamente dicho) y traduce el código fuente escrito en un lenguaje de programación almacenado en un archivo en código de bajo nivel (normalmente en código objeto, no directamente a lenguaje máquina). El segundo paso se llama enlazador en el cual se vincula el código de bajo nivel generado de todos los ficheros y subprogramas que se han mandado compilar y se añade el código de

las funciones que hay en las bibliotecas del compilador para que el ejecutable pueda comunicarse directamente con el sistema operativo, traduciendo así finalmente el objeto a código máquina, y generando un módulo ejecutable.

Estos dos pasos se pueden hacer por separado, almacenando el resultado de la fase de compilación en archivos objetos (el típico *.o*); para enlazarlos en fases posteriores, o crear directamente el ejecutable; el resultado de la fase de compilación se almacena sólo temporalmente. Un programa podría tener partes escritas en varios lenguajes (por ejemplo C, C++ y ensamblador), que se podrían compilar de forma independiente y luego enlazar juntas para formar un único módulo ejecutable [2].

2.2.1 Creación y envío del ejecutable a la tarjeta SIE

Teniendo en cuenta el uso de registros, las llamadas de funciones, las secciones del programa, los especificadores de formato y el uso de instrucciones; el estudiante está en la capacidad de crear un programa en *assembler*.

Una vez que se ha diseñado el programa se debe guardar con extensión *.s* y proceder a compilarlo con ayuda del compilador cruzado.

2.2.1.1 Compilador cruzado

Un compilador cruzado es capaz de crear código ejecutable para otra arquitectura distinta a aquélla en la que él se ejecuta. Esta herramienta es útil cuando se quiere compilar código para una plataforma a la que no se tiene acceso, o cuando es incómodo o imposible compilar en dicha plataforma. Para este caso, se compila en el computador un programa en *assembler* que está diseñado para otra arquitectura (MIPS).

Los pasos para compilar y crear el ejecutable son los siguientes:

- ▷ Abrir una consola y ubicarse en la carpeta donde está guardado el código fuente con extensión *.s*.
- ▷ Digite *mipsel-openwrt-linux-gcc -c -g ejemplo.s*, seguido de la tecla enter. El ensamblador lee el archivo *.s* y genera un archivo objeto relocalizable el cual lo guarda con extensión *.o* (figura 2.1).

Figura 2.1: Diagrama de generación del archivo objeto.

En este paso el compilador puede informar si existen errores en la sintaxis del programa, si los hay éste indica en qué línea está y cuál fue el posible error.

▷ Luego se digita *mipsel-openwrt-linux-gcc ejemplo.s -o ejemplo*, seguido de la tecla enter. El enlazador une el archivo objeto con la biblioteca, produciendo en su salida el ejecutable del programa llamado *ejemplo*. Esto se puede visualizar en el diagrama de la figura 2.2 [2].

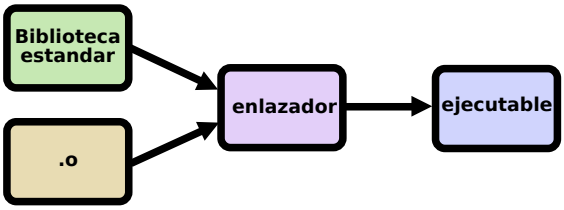

Figura 2.2: Diagrama de generación del ejecutable del programa.

```
1 $ mipsel-openwrt-linux-gcc -c -g ejemplo.s
2 $ mipsel-openwrt-linux-gcc ejemplo.s -g -o ejemplo
```

Script 2.1: Compilación y generación del ejecutable

Hasta aquí se ha compilado y creado el ejecutable del programa, solo faltaría enviar el ejecutable a la tarjeta SIE, pero antes de esto se debe conectar la tarjeta y establecer comunicación.

2.2.1.2 Comunicación con la tarjeta SIE

Para ejercer comunicación con la tarjeta SIE se deben seguir algunos pasos que quizás ya se han utilizado en cursos anteriores pero aquí se van a recordar.

- Abrir una consola nueva y digitar *sudo ifconfig usb0 192.168.1.2 up*, esto se hace para establecer comunicación con la tarjeta.
- Para comprobar si hay comunicación se digita *ping 192.168.1.1.*

- Si ya hay comunicación con la tarjeta, se procede a enviar el ejecutable digitando *scp ejemplo root@192.168.2.1:~* , así se envía el ejecutable del programa llamado *ejemplo* a la tarjeta. En este paso pide la clave de la tarjeta, se le da *enter*.

- Se ha enviado el ejecutable a la tarjeta, el paso siguiente es entrar a la tarjeta y eso se consigue digitando *ssh root@192.168.1.1*.

- Ya estando en la tarjeta se puede comprobar si el ejecutable está dentro de la tarjeta digitando el comando *ls -l*, este muestra todos los archivos contenidos en el directorio raíz de la tarjeta.

- Una vez comprobado que el archivo este dentro de la tarjeta, se procede a ejecutarlo digitando *./ejemplo*. Si todo ha salido bien, se ha diseñado y ejecutado el primer programa en *assembler* para la plataforma SIE.

Bibliografía

[1] MIPS Tecnologies: *MIPS32TM Architecture For Programing*, Volumen II, Mountains View, USA, MIPS Tecnologies, (2002).

[2] Amelia Ferreira, Vicente Robles: *Programación en Lenguaje Ensamblador bajo Linux*, Caracas, VENEZUELA, (2007).

CAPÍTULO 3

DEPURACIÓN DE PROGRAMAS CON GDB

3.1 EJECUCIÓN DE PROGRAMAS CON EL DEPURADOR

Es importante recordar que a la hora de compilar el programa ensamblador con extensión *.s*, se debe agregarle la bandera *-g*, esta incluye la tabla de símbolos ampliada en la cabecera del archivo ejecutable en la que se encuentran las equivalencias entre posiciones de memoria de variables, funciones y los nombres que se les asignó en el programa para que luego pueda ser depurado con GDB. El procedimiento para la compilación sería de la siguiente manera:

```
1 $ mipsel-openwrt-linux-gcc -c -g ejemplo.s
2 $ mipsel-openwrt-linux-gcc ejemplo.s -g -o ejemplo
```

Script 3.1: Compilación y generación del ejecutable

3.1.1 EJECUTAR EL PROGRAMA CON GDB

Una vez se esté parado en la consola de la tarjeta para ejecutar el programa con el depurador, se digita el comando "gdb" seguido del nombre del ejecutable del programa que se desee depurar y luego *enter*. Esto ocasionará que aparezca un nuevo *prompt* simbolizado por la cadena "(gdb)", lo que indica que el depurador está listo para recibir una orden, es similar al símbolo "$" o "#" en una consola de Linux.

```
1  root@BenNanoNote:~# gdb sumavector
2
3  dlopen failed on 'libthread_db.so.1' - File not found
4  GDB will not be able to debug pthreads.
5
6  GNU gdb 6.8
7  Copyright (C) 2008 Free Software Foundation, Inc.
8  License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
9  This is free software: you are free to change and redistribute it.
10 There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
11 and "show warranty" for details.
12 This GDB was configured as "mipsel-openwrt-linux".
13 (gdb)
```

Script 3.2: Iniciación del depurador GDB

Como se puede ver el programa ya se está depurando. Existen muchos comandos para analizar un programa, los cuales se explicarán más adelante. Es sencillo de manejar debido a la semejanza que tiene la consola de GDB con la consola que normalmente se maneja en Linux, se puede utilizar la tecla *tabular* para completar los comandos automáticamente en GDB. También es posible utilizar las teclas "Flecha Arriba" y "Flecha Abajo" para buscar los últimos comandos utilizados. El comando *help* proporciona información de cada uno de los comandos. Si se utiliza *help* sin argumentos, proporcionara una lista con las principales categorías sobre las cuales se pueden obtener ayuda. Sin embargo, es posible obtener ayuda más detallada utilizando el comando *help* más el comando que se desee obtener información [1] [2].

3.2 COMANDOS BÁSICOS DEL DEPURADOR GDB

GDB es el depurador de GNU que permite ver lo que está sucediendo dentro de programas escritos en C, C++ y *assembler*. A continuación se presentan los comandos básicos que maneja GDB separados en tres secciones y algunos ejemplos utilizando el programa *sumavector.s* que está incluido dentro de la guía respectiva.

3.2.1 Comandos básicos de control

El depurador permite controlar la ejecución del programa que se desea analizar, ejecutándolo y deteniéndolo en cualquier instante por medio de un indicador. El programa se puede analizar por partes o paso a paso para obtener información en cualquier instante del programa. A conti-nuación se muestra una breve descripción junto con algunos ejemplos de los comandos más utilizados en el depurador GDB para controlar y analizar la ejecución de un programa [3] [4].

- ***run***: Ejecuta el programa que se quiere depurar desde el inicio. Puede que la ejecución termine correctamente o que se encuentren resultados erróneos, si esto ocurre no hay que preocuparse por ahora, más adelante se verán otros comandos que ayudarán a solucionar esta clase de errores.

- ***break***: Permite detener el programa en cualquier parte. Generalmente cuando se coloca un *breakpoint* (punto de parada) se puede indicar una línea, una dirección (con * al inicio) o una etiqueta donde se desee parar. Es aconsejable colocar un *breakpoint* antes del punto del programa en donde se sospeche que pueden estar los errores, de tal forma que cuando se vaya a procesar se detenga la ejecución y así se podrá inspeccionar lo que sucede a partir de ese punto. Antes de correr el programa se debe establecer al menos un *breakpoint*.

▷ ***info breakpoints***: Muestra los puntos de parada que se han creado en el programa, así como la información referente a los mismos (número, tipo, dirección, estado, etc). En particular es importante resaltar que muestra cuales están desactivados (n) y cuales activados (y).

Para aplicar los comandos anteriores en el programa ejemplo, primero se ejecuta el programa con el depurador GDB y a continuación se digitan las siguientes líneas:

```
1  (gdb) break 12                                          # parada línea 12
2  Breakpoint 1 at 0x400788: file sumavector.s, line 12.
3  (gdb) break *0x400784                                   # parada dirección 0x400784
4  Breakpoint 2 at 0x400784: file sumavector.s, line 11.
5  (gdb) break main                                        # parada etiqueta main
6  Breakpoint 3 at 0x400784: file sumavector.s, line 10.
7  (gdb) info breakpoints                                  # información de paradas
8  Num     Type           Disp Enb Address    What
9  1       breakpoint     keep y   0x00400788 sumavector.s:12
10 2       breakpoint     keep y   0x00400784 sumavector.s:11
11 3       breakpoint     keep y   0x00400784 sumavector.s:10
12 (gdb)
```

Script 3.3: Puntos de parada

▷ ***delete***: Se utiliza para borrar un punto de parada. Para eliminar un punto de parada se utiliza el comando *delete* con el número del punto de parada como argumento y con *delete* solo o acompañado con el argumento *breakpoints*, se borran todos los puntos de parada que se han creado.

▷ ***disable* y *enable***: Se utilizan para desactivar y activar puntos de parada. Por medio de los comandos *disable* y *enable* e indicando su número del punto de parada como argumento, se desactivará o activará. Si no se especifica ninguno se desactivarán o activarán todos los puntos de parada creados.

▷ ***ignore***: Su función es la de ignorar puntos de parada. Se utiliza el comando *ignore* seguido del número del punto de parada que se va a ignorar y luego el número de las veces que se quiere ignorar. Este comando es muy utilizado en programas que contienen bucles.

```
1  (gdb) delete 2                          # borra punto de parada 2
2  (gdb) delete                            # borra todos los punto de parada
3  Delete all breakpoints? (y or n) y
4  (gdb) info breakpoints
```

```
5  No breakpoints or watchpoints.
6  (gdb) disable 1                          # desactiva parada 1
7  (gdb) enable 1                           # activa parada 1
8  (gdb) ignore n m                         # ignora la parada n, m veces
9  (gdb)
```

Script 3.4: Control de puntos de parada

- ***watch***: Los *watchpoints* permiten parar la ejecución del programa cuando un registro o variable cambia. No es necesario indicar un lugar determinado del programa para colocar los *watchpoints*. Es necesario ejecutar el programa con GDB, luego crear un *breakpoint*, después se ejecuta el programa y en este instante se pueden crear los *watchpoints* que se sean necesarios.

  Los *watchpoints* hacen que los programas se ejecuten más lentamente, pero permiten encontrar errores que no se saben donde se producen. La situación más común en la que se utiliza este comando es cuando se tiene alguna variable que no debía cambiar de valor y ésta cambia a un valor no deseado, lo que se hace es asignar un *watchpoint* al registro donde está contenida la variable y se ejecuta el programa.

  El comando para colocar un *whatchpoint* es *watch* seguido del registro o variable que se desee controlar, cada vez que este registro o variable sea modificada dentro del programa éste se va a detener indicando el antiguo y nuevo valor.

- ***info watchpoints***: Muestra los puntos de observación que se han creado durante la ejecución del programa, así como información referente a los mismos (número, tipo, dirección, estado, etc). En particular es importante resaltar que muestra las veces que el mismo ha sido alcanzado.

```
1  (gdb) break main                         # parada etiqueta main
2  Breakpoint 1 at 0x400784: file sumavector.s, line 10.
3  (gdb) watch $s5                          # parada cuando cambie $s5
4  Watchpoint 2: $s5
5  (gdb) continue
6  Watchpoint 2: $s5
7  Old value = 8
8  New value = 0
9  ciclo () at sumavector.s:17
10 17      in sumavector.s
11 (gdb) info watchpoints                   # información de puntos de observación
12 Num     Type           Disp Enb Address    What
13 1       breakpoint     keep y   0x00400784 sumavector.s:10
```

```
14          breakpoint already hit 1 time
15  2       watchpoint     keep y              $s5
16          breakpoint already hit 1 time
17  3       watchpoint     keep y              $s2
18          breakpoint already hit 1 time
19  (gdb)
```

Script 3.5: Puntos de observación

Cada vez que se crea un nuevo punto de parada sea *break* o *watch* el depurador le asigna un número ordenadamente empezando en 1, este número se conserva durante la depuración sin importar si se borran puntos anteriores a no ser que se borren todos los puntos de parada, si esto ocurre GDB vuelve a asignar el número desde 1 nuevamente.

- ***continue***: Este comando se utiliza para continuar la ejecución del programa después de que ha sido detenido por un *breakpoints* o un *watchpoints* hasta que se encuentre con otro punto de parada. También se puede ejecutar el comando *continue* más un número como argumento, donde este número indica las veces que se puede pasar por un punto de parada sin que se detenga el programa (se ignora n veces un punto de parada). Esta opción es muy utilizada cuando se tiene un punto de parada ubicado dentro de un bucle y se quiere que pasen n número de iteraciones antes de parar el programa.

- ***step***: Ejecuta el programa hasta alcanzar la siguiente línea de código fuente ejecutándose paso a paso. Básicamente permite ejecutar la próxima instrucción del programa en lenguaje ensamblador. Si se especifica un entero n como argumento se ejecutan n instrucciones o líneas.

  Este comando es muy utilizado a la hora de detectar errores, se recomienda una técnica que consiste en establecer un punto de parada al comienzo de una función donde posiblemente podría estar el problema y luego ejecutarla paso a paso con este comando. Se puede aprovechar el hecho que GDB repite el último comando con *enter* para ejecutar varias líneas seguidas.

- ***stepi***: Ejecuta la próxima instrucción de máquina del lenguaje ensamblador del programa. Como estas funciones cuentan con información de depuración, entra a ejecutar su código paso a paso de cada instrucción. Si se especifica un entero n como argumento se ejecutan n instrucciones.

- ***next***: Similar al comando *step*, pero en caso de encontrarse una llamada a un procedimiento (*printf* o *scanf*), el mismo es tratado como si fuera una instrucción, es decir, no se lleva a cabo el recorrido interno del procedimiento invocado sino que la ejecuta de forma completa y se detiene cuando termina la ejecución de la llamada.

- ***quit***: Se utiliza para salir de la consola de GDB y volver a la consola de SIE.

3.2.2 Comandos básicas de inspección

En los casos que la ejecución de un programa se encuentra detenida, sin importar el motivo de la parada, es posible inspeccionar el programa con algunos comandos utilizados por el depurador GDB que se describe a continuación [5] [3].

- ***info registers***: Su función es la de mostrar el contenido de todos los registros. Al ejecutar el comando *info registers* se despliega una lista con todos los registros indicando su nombre, su etiqueta y su contenido en hexagesimal. Este comando es muy útil para inspeccionar el contenido de varios registros a la vez.

- ***print***: Uno de los objetivos más importantes del depurador es la inspección de cualquier variable o registro del programa. De esta forma se puede comprobar si el programa esta funcionando con los datos de la forma deseada, para esto es utilizado el comando *print*, colocando como argumento la variable o el registro, así se podrá ver su contenido.

- ***display***: Su función es similar a la del comando *print*, la diferencia radica en que el comando *display* imprime la variable o registro cada vez que se detenga la ejecución del programa.

- ***disassemble***: Desensambla y muestra las instrucciones de máquina en un rango correspondiente a la etiqueta en donde esté ubicado. Este comando es muy útil en situaciones en las que no se tenga acceso al código ensamblador.

- ***info variables***: Imprime todas las variables globales definidas del programa. Este comando puede servir para verificar que el programa esté reconociendo todas las variables que se han definido al inicio del programa.

- ***info locals***: Es muy similar al comando anterior, la diferencia es que este muestra solo las variables locales definidas.

3.2.3 Comandos para la alteración en la ejecución

Un programa puede ser modificado mientras se está ejecutando con el depurador, cambiando el valor de la memoria o de los los registros, alterando en consecuencia de forma indirecta algunos parámetros en su ejecución.

Esta herramienta es muy útil cuando se encuentra un problema y se quiere probar la solución de forma rápida sin tener que volver a compilar y cargar el programa. Los comandos comúnmente utilizados para alterar de forma indirecta la ejecución de un programa son los siguientes [2].

- ***set***: Este comando permite evaluar una expresión y asignar su resultado a un registro o una posición de memoria, para luego continuar con la ejecución del programa con el nuevo valor. Este comando no imprime el resultado, solo cambia la variable.

- ***print***: Este comando además de inspeccionar el valor de cualquier variable, también puede asignarle un nuevo valor a un registro o una posición de memoria, colocando el nombre de la variable seguido del operador de asignación "=" y luego el nuevo valor que se le quiere dar.

- ***display***: Similar al comando *print*. La diferencia consiste en que la asignación será llevada a cabo cada vez que la ejecución se detenga.

- ***run***: Este comando permite continuar la ejecución en un punto del programa distinto a la instrucción en que se detuvo. La instrucción a la que se saltará puede ser indicada mediante una etiqueta o mediante una dirección.

Bibliografía

[1] Eduardo Domínguez Parra y Carlos Villarrubia Jiménez: *Prácticas de Sistemas Operotivos*, Huélamo, ESPAÑA, Universidad de Castilla, (2008).

[2] Francisco Romero Vargas: *Desarrollo de Funciones en el Sistema Informático*, Jerez de la Frontera, ESPAÑA, Instituto San Juan Bosco, (2006).

[3] Franco Rodríguez Fabregues: *GDB GNU Debugger*, JCórdoba, ARGENTINA, Universidad Nacional de Córdoba, (2011).

[4] GNU GDB: http://arco.esi.uclm.es/ david.villa/doc/repo/gdb/gdb.html, *Guía básica para el uso de GDB*.

[5] Juan Jose Moreno Moll: *Depurador GDB*, Madrid, ESPAÑA, Universidad de Valéncia, (2002).

CAPÍTULO 4

MODOS DE DIRECCIONAMIENTO

En esta sección se analizarán los tres tipos de formato de instrucción que maneja la arquitectura MIPS. Es importante conocerlos debido a que son utilizados en todos los modos de direccionamiento y ayudaría a una mejor comprensión en el desarrollo de las prácticas en general.

4.1 FORMATO DE INSTRUCCIÓN

El formato de instrucción indica los campos y el tamaño como está compuesto. En los MIPS todas las instrucciones tienen 4 bytes (32 bits). Los 32 bits se reparten en campos, algunos de los cuales son fijos. De esta forma, la interpretación de la instrucción es mas sencilla y uniforme [1] [2].

Para los MIPS el significado de cada campo en las instrucciones es el siguiente.

op	**rs**	**rt**	**rd**	**shamt**	**funct**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Tabla 4.1: Formato de instrucción. Nombre y tamaño de cada campo.

- ▷ *op:* es el código de operación también conocido como *opcode*.
- ▷ *rs:* es el registro del primer operando fuente.
- ▷ *rt:* es el registro del segundo operando fuente.
- ▷ *rd:* es el registro destino; donde se guarda el resultado de la operación.
- ▷ *shamt:* es utilizada en operaciones de rotación e indica el valor del desplazamiento.
- ▷ *funct:* es el que selecciona la variante de la operación especificada por el campo *op*.

Cada instrucción dentro de la arquitectura MIPS tiene asignado un código *op* y un código *funct*. Esto se puede observar en la tabla 4.2, donde se muestran los códigos de las instrucciones comúnmente utilizadas.

Instrucción	***op***	***funct***
add	000000	100000
addi	001000	na
sub	000000	100010
mul	011100	000010
div	000000	011010
and	000000	100100
andi	001100	na
or	000000	100101
ori	001101	na
nor	000000	100111
xor	000000	100110
xori	001110	na
li	001111	na
lb	100000	na
lw	100011	na
sb	101000	na
sw	101011	na
slt	000000	101010
beq	000100	na
bgtz	000111	na
blez	000110	na
bne	000101	na
j	000010	na
jal	000011	na

Tabla 4.2: Códigos *op* y *funct* de las instrucciones MIPS

La arquitectura MIPS maneja tres tipos de formato en donde se puede representar todo su repertorio de instrucciones, estos tipos de formato son:

- Formato tipo R
- Formato tipo I
- Formato tipo J

4.1.1 FORMATO TIPO R

Este tipo de formato es utilizado en las operaciones entre registros como las operaciones aritméticas. Por ejemplo si tenemos la siguiente instrucción:

```
1       add  $16, $20, $21    # $16 = $20 + $21, suma el contenido del registro 20 con el
2                             # contenido del registro 21 y lo almacena en el registro 16.
```

Script 4.1: Ejemplo del formato tipo R

Esta instrucción se representa en la tabla 4.3 donde se puede observar el campo, el valor en binario, el valor en decimal y por último el tamaño de cada campo en bits.

op	rs	rt	rd	shamt	funct
000000	10100	10101	10000	00000	100000
0	20	21	16	0	32
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Tabla 4.3: Formato de instrucción tipo R.

4.1.2 FORMATO TIPO I

Este tipo de formato es muy utilizado en operaciones que tengan que ver con la memoria, como las instrucciones de transferencia de datos o ramificación condicional. La diferencia con el formato anterior es que éste contiene un campo de 16 bits el cuál se utiliza para especificar la dirección de la memoria. Su estructura está en la tabla 4.4, junto con la siguiente instrucción que se toma como ejemplo:

```
1          lw  $16, V[$20]          # $16 = V[$20], carga el contenido del elemento V[i]
2                                   # contenido en memoria y lo almacena en el registro 16.
```

Script 4.2: Ejemplo del formato tipo I

Esta instrucción se representa en la tabla 4.4 donde se puede ver el nombre del campo, su valor en decimal y por último el tamaño de cada campo en bits. En el anterior ejemplo se puede ver que el significado del campo *rt* ha cambiado para esta instrucción: en una instrucción de carga el campo *rt* determina el registro que va a guardar el dato que se encuentra en la dirección especificada por V[$20].

op	rs	rt	dirección
35	20	16	V[i]
6 bits	5 bits	5 bits	16 bits

Tabla 4.4: Formato de instrucción tipo I.

4.1.3 FORMATO TIPO J

Este formato contiene el código de operación de 6 bits, seguido de un campo de 26 bits para especificar una dirección. Este tipo de formato se utiliza para operaciones de salto y bifurcación incondicional. A continuación se presenta un ejemplo donde se emplea este tipo de formato.

```
1          J main           # Ir a la etiqueta main, salto sin condición.
```

Script 4.3: Ejemplo del formato tipo J

En la tabla 4.5 se puede observar el nombre del campo, su valor en decimal y por último el tamaño de cada campo en bits.

op	**dirección**
35	[main]
6 bits	26 bits

Tabla 4.5: Representación del formato de instrucción tipo J.

MIPS es una arquitectura de carga/almacenamiento, lo que significa que sólo las instrucciones de carga y almacenamiento acceden a la memoria, la arquitectura proporciona un modo de direccionamiento para acceder a memoria llamado *direccionamiento base*. El resto de instrucciones operan sólo con valores almacenados en los registros.

Bibliografía

[1] John L. Hennessy, David A. Patteeson: *Organización y Diseño de Computadores. La interfaz hardware/software*, Madrid, ESPAÑA, 2 edición, McGRAW-HILL, (2003), 1-55860-281-X.

[2] José Daniel Muñoz Frías: *Estructura de Computadores, Direccionamiento y formatos*, Madrid, ESPAÑA, Universidad Pontificia Comillas. ETSI ICAI, (2003).

CAPÍTULO 5

JERARQUÍA DE MEMORIA. RENDIMIENTO DE LA CACHÉ

En esta sección se presenta la descripción de las funciones para reservar espacio en memoria y medir el tiempo de ejecución de un programa. También se describe el procedimiento para compilar un programa en lenguaje C con el compilador cruzado, enviar a la tarjeta SIE el ejecutable y por último ejecutarlo.

5.1 ASIGNACIÓN DE ESPACIO EN LA MEMORIA

En el programa ejemplo del *script* 5.1 se utiliza la función *malloc()*, esta función por medio de punteros reservan o liberan memoria según necesite, es decir, dinámicamente. El tamaño reservado se hace en *bytes*, por esta razón se multiplica por 4 (línea 20), para el caso del *script* 5.1 el tamaño es de 400 *Bytes*. Una vez que se utilizan las variables creadas por la función *malloc()* (línea 19), se deben liberar con la función *free()* (línea 32), con el nombre de la variable como parámetro [1].

```
1  #include <stdio.h>              /*libreria para la función printf*/
2  #include <stdlib.h>             /*libreria para la función malloc()*/
3  #include <time.h>               /*libreria para medir el tiempo*/
4  #include <sys/time.h>           /*libreria que define el timeval para medir el tiempo*/
5  #define N 100                   /*define el numero de iteraciones, tamaño del vector */
6  double timeval_diff(struct timeval *x, struct timeval *y)
7          {
8          return
9          (double)(x->tv_sec + (double)x->tv_usec/1000000) -
10         (double)(y->tv_sec + (double)y->tv_usec/1000000);       /*función para medir el*/
11         }                                                       /*tiempo en [ms]*/
12 int main(int argc, char *argv[])
13         {                                       /*declaración de variables*/
14         struct timeval t_ini, t_fin;
15         double secs;
16         static int d;
17         int a= 1;
18         int b = 1;
19         int *vec;                               /*puntero para reservar memoria*/
20         vec=(int *) malloc(4*N*sizeof(int));    /*asignación de memoria*/
21                 gettimeofday(&t_ini, NULL);     /*inicio del cronometro*/
22                         while(b<=N)
```

```
23                                  {
24                                  vec[b]=a+b;
25                                  a=vec[b];
26                                  d=vec[N-b]+a;           /*operación que exige la caché*/
27                                  b++;
28                                  }
29                  gettimeofday(&t_fin, NULL);             /*fin del cronometro*/
30          secs = timeval_diff(&t_fin, &t_ini);
31          printf("%.16g milliseconds\n", secs * 1000.0);  /*tiempo de ejecución en [ms]*/
32          free(vec);                                      /*liberación de variables*/
33          return 0;
34          }
```

Script 5.1: Programa ejemplo

5.2 TIEMPO DE EJECUCIÓN DEL PROGRAMA EJEMPLO

Para medir de forma precisa el tiempo que tarda en ejecutarse el programa ejemplo, se utiliza la función *gettimeofday()* que se encuentra en la librería *sys/time.h*. Esta función ofrece teóricamente una precisión de microsegundos (0,001 milisegundos). Su sintaxis y función de cada línea se muestran en el script 5.2 [2].

```
1  #include <time.h>                /*libreria para medir el tiempo*/
2  #include <sys/time.h>            /*libreria que define el timeval para medir el tiempo*/
3  double timeval_diff(struct timeval *x, struct timeval *y)       /*registro con 2 campos*/
4          {
5          return
6          (double)(x->tv_sec + (double)x->tv_usec/1000000) -
7          (double)(y->tv_sec + (double)y->tv_usec/1000000);       /*función para medir el*/
8          }                                                       /*tiempo en [ms]*/
9  int main(int argc, char *argv[])
10         {                                               /*declaración de variables*/
11         struct timeval t_ini, t_fin;
12                 gettimeofday(&t_ini, NULL);             /*inicio del cronometro*/
13                                                         /*cuerpo del programa*/
14                 gettimeofday(&t_fin, NULL);             /*fin del cronometro*/
15         secs = timeval_diff(&t_fin, &t_ini);
16         printf("%.16g milliseconds\n", secs * 1000.0);  /*imprimir tiempo de ejecución */
17         }                                               /*en [ms]*/
```

Script 5.2: Sintaxis de la función gettimeofday().

5.3 COMPILACIÓN Y ENVÍO A LA TARJETA SIE

El programa ejemplo completo se puede observar en el *script* 5.1. Teniendo el programa terminado los pasos a seguir son: compilarlo usando el compilador cruzado y enviar a la tarjeta el ejecutable.

1. Primero se conecta la tarjeta SIE al computador, abrir una consola y digitar *sudo ifconfig usb0 192.168.1.2 up*, esto se hace para establecer comunicación con la tarjeta.

2. Entrar a la tarjeta por medio del comando *ssh root@192.168.1.1*. En seguida pide la clave, oprimir la tecla *enter*.

3. Abrir una nueva consola y se ubican en la carpeta donde está el programa ejemplo.

4. Se ejecuta el comando *mipsel-openwrt-linux-gcc -o2 programa.c -o programa*. Con esta línea y por medio del compilador cruzado se crea el ejecutable *programa*.

5. Ahora se envía el ejecutable digitando *scp programa root@192.168.2.1:˜* , con esta línea se envía el ejecutable con nombre *programa* a la tarjeta.

6. Por último, se ejecuta el programa en la tarjeta digitando *./programa*.

Nota: cada vez que se modifique el programa se deben seguir los pasos del 4 al 6 y ejecutar el programa varias veces, esto se hace necesario para tener una medición de tiempo más precisa.

Bibliografía

[1] Héctor Tejeda Villela: http://www.fismat.umich.mx/mn1/manual/, *Manual de C*, Uso de malloc, sizeof y free.

[2] man.cx: http://man.cx/gettimeofday, *BSD System Calls Manual GETTIMEOFDAY*.