Universidad de Costa Rica

Facultad de Ingeniería
Escuela de Ingeniería Eléctrica
IE0521 – Estructuras de Computadoras Digitales II
II ciclo 2016

Quiz #9

Boris Altamirano Chinchilla B30255
Alonso Espinoza Barboza B22356
J.Johel Rodríguez Pineda B25706
Grupo 01

Profesor: Jose Daniel Hernández Vargas

01 de Diciembre de 2016

Índice

Índice de figuras

Resumen

Se investigó las generalidades de la arquitectura OpenRisc1200, además de llevar a cabo la instalación de varias herramientas como Docker y Vagrant, las cuales permitieron la simulación de un pequeño y simple programa generado en lenguaje C. Con esto se pudo observar algunas de las instrucciones utilizadas por esta arquitectura para llevar a cabo la ejecución del programa.

Palabras Clave: *Arquitectura de computadores*, *Compilación*, *Contenedor*

1. Introducción

En el siguiente proyecto se estudiara el procesador OpenRISC 1200 este es una inplementacion del OpenRISC 1000, el cual es un hardware libre lo que quiere decir que su codigo esta disponible para ser implementado en una FPGA ya que su licencia es publica y cualquier persona que lo desee lo puede implementar.
La arquitectura en la cual nos enfocaremos en el siguiente proyecto es la arquitectura Harvard esta es usada en computadores de propósito especifico como por ejemplo los DSP estos se usan en renderizado de vídeo y en procesamiento de sonido esto tiene ya que la mayor ventaja que tiene este es que puede acensar a los datos al mismo tiempo que acceso a las instrucciones.

Una de las mas grandes ventajas que tiene esta arquitectura es que no tiene que compartir caracteristicas las instrucciones de datos esto comparado con la arquitectura clásica que es la von newman que tiene como característica que las instrucciones y los datos comparten la misma memoria lo que hace que tenga que tener el mismo tamaño.

Además, para poder quedar más claros con la funcionalidad de los contenedores, se levantará un contenedor con este tipo de arquitectura, donde se creará un programa sencillo en lenguaje C, el cual mediante el toolchain del OR1200/GNU se podra, crear, compilar, y ademas desensamblar el código binario que este genere, lo cual llevará a responder nuevos conceptos tales como la compilación cruzada.

2. Arquitectura OpenRISC 1200

La arquitectura general con la que se implemento el procesador openRISC1200 es la arquitectura Harvard esta tiene como una de sus principales características es que físicamente se separan las memoria de instrucción y de datos esto quiere decir que los buses de ambos son completamente independientes esto nos permite tener distintos contenido en la misma dirección y también tener distinto numero de bits entre datos e instrucciones .

Los dos cache tanto datos como el de instrucciones usan mapeo directo , posee controlador de interrupciones programable e interrupciones por interfaz lo que permite comunicarse con puertos externos. MMUs son implementadas con 64 hasf based 1-way con mapeo directo a TLB tanto para datos como para instrucciones, posee una unidad de manejo de energía, un timer de alta resoluciones y facilidades con el debugging en tiempo real. Este sistema al ser libre a sido implementado en FPGA y ASIC pero no tiene una versión comercial, por lo tanto sus especificaciones en cuanto a rendimiento depende de como fue implementado y en que sistema.

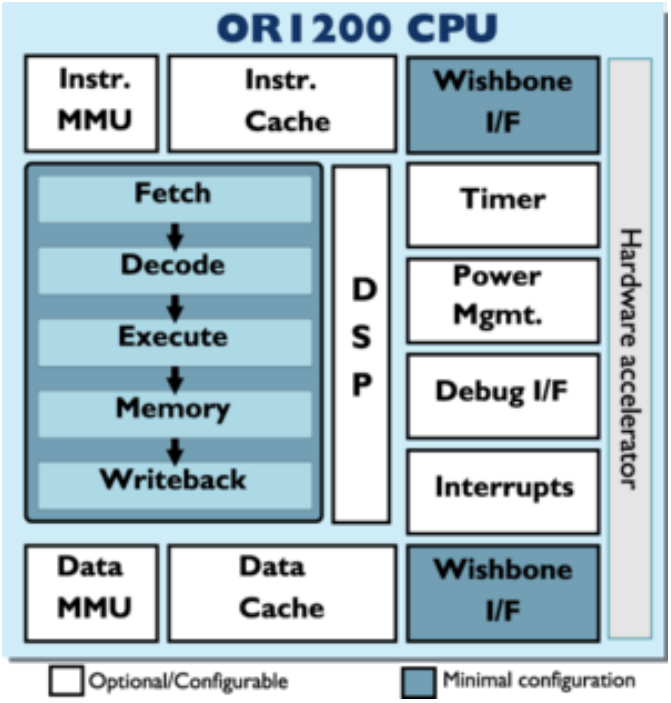

Figura 1: Diagrama general del CPU de un OpenRISC 1200

2.1. Pipeline Stages

Un pipeline es una técnica de implementación donde se busca aprovechar la independencia entre las acciones necesarias para completar una instrucción, de esta manera se traslapa la ejecución de instrucciones y así se reduce el tiempo de ejecución de un grupo de instrucciones. En la figura 2 se muestra un diagrama básico de un pipeline donde cada bloque representa una de las etapas necesarias para ejecutar una instrucción, estas etapas se conocen como *pipe stage*.

Figura 2: Esquema básico de un pipeline

El pipeline de en un procesador de este tipo cuenta con cinco etapas: Fetch (IF), Decode (ID), Execute (EX), Memory Access (MEM) y Write-back (WB).

1. **Instruction fetch cycle (IF):**
   Se envía el contador de programa a la memoria de instrucciones y se extrae la instrucción correspondiente a ejecutar. Además se actualiza el contador de programa sumándole cuatro (para instrucciones de 32 bits). En la figura 3 se observa este proceso.

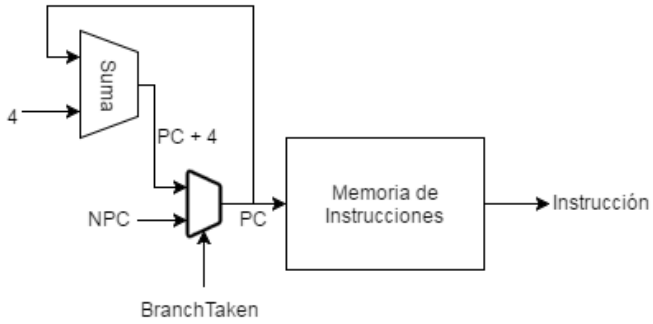

Figura 3: Etapa IF en un procesador RISC.

2. **Instruction decode/register fetch cycle (ID):**
   Decodifica la instrucción y lee los registros fuente, al mismo tiempo se hace una prueba de igualdad a los registros en caso de que se presente un salto condicional. Se realiza una extensión de signo al valor inmediato en caso de que se necesite y se calcula la posible dirección de salto. Es posible realizar la decodificación de la instrucción simultáneamente con la lectura de los registros por el formato de la instrucción. Observe el procedimiento de esta etapa en la figura 4.

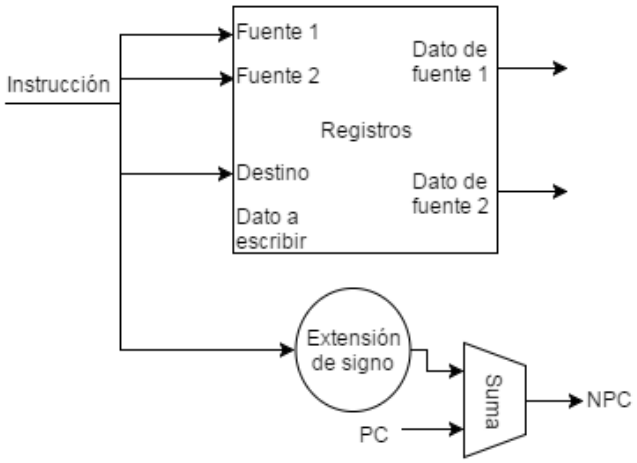

Figura 4: Etapa ID en un procesador RISC.

3. **Execution/effective address cycle (EX):**
   En esta etapa la ALU ejecuta una de tres posibles operaciones, estas pueden ser calcular la dirección efectiva al sumar el contenido de un registro con el offset, realizar una operación con los datos de los registroso realizar una operación con el dato de un registro y el valor inmediato, como se ilustra en la figura 5.

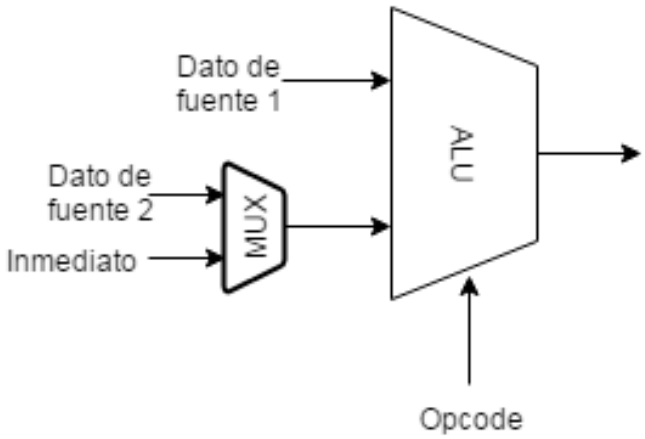

Figura 5: Etapa EX en un procesador RISC.

4. **Memory access (MEM):**
   Si la instrucción es un load se lee el dato en la posición de memoria calculada en la etapa previa, y si es un store se utiliza la dirección efectiva para indicar donde se escribe el dato de fuente 2. Para una descripción gráfica de este proceso refiérase a la figura 6.

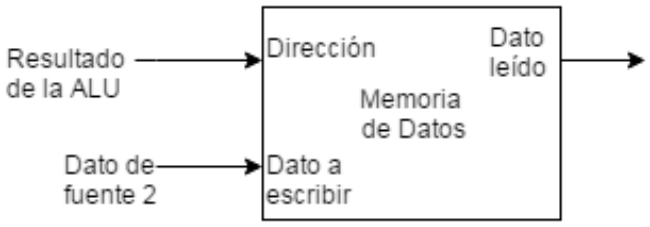

Figura 6: Etapa MEM en un procesador RISC.

5. **Write-back cycle (WB):**
   Si la instrucción es un load se lee el dato en la posición de memoria calculada en la etapa previa, y si es un store se utiliza la dirección efectiva para indicar donde se escribe el dato de fuente 2. Para una descripción gráfica de este proceso refiérase a la figura 6.

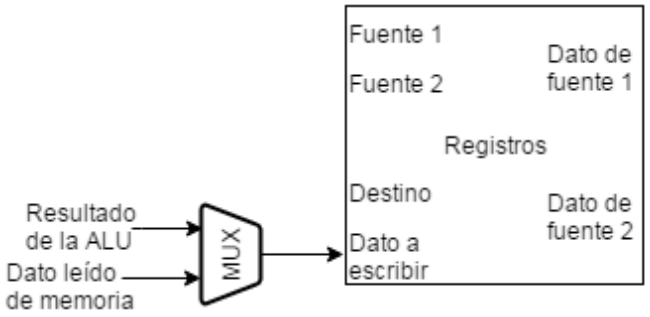

Figura 7: Etapa WB en un procesador RISC.

3. Resultados

3.1. Simulación de un programa sencillo en la arquitectura OpenRisc

Para realizar la simulación del programa en este tipo de arquitectura se requería instalar Docker y Vagrant para poder levantar un contenedor con la arquitectura deseada.
En la figura 8 se observa que en el primer comando que se ejecuta es para levantar el daemon del servicio del docker y seguidamente se ejecuta el ejemplo del hello-world utilizando siempre permisos de administrador(sudo) para poder ejecutarlo, ya que aún no se ha añadido el usuario al grupo.

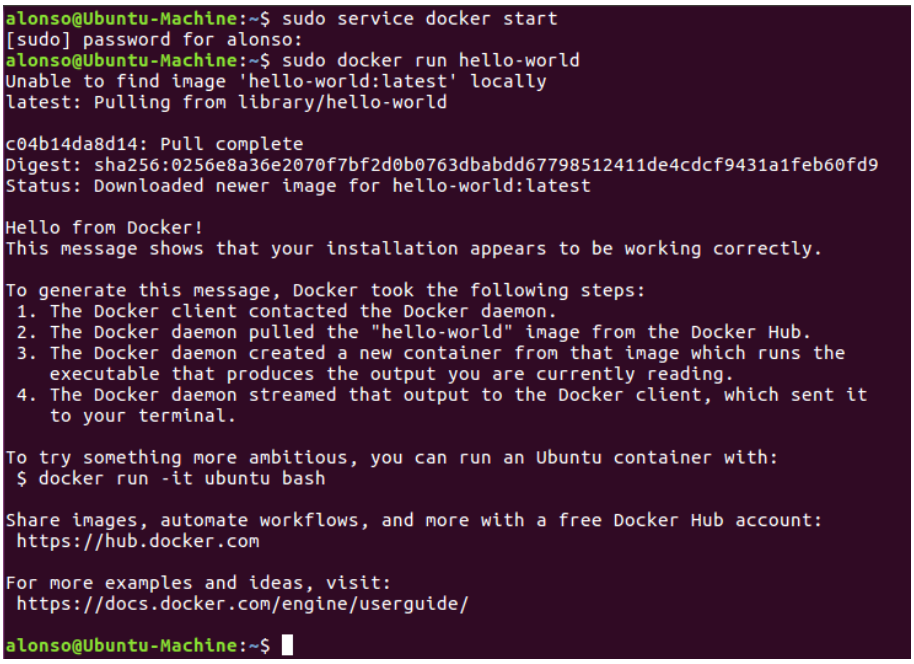

Figura 8: Comprobación de la ejecución del Daemon y el ejemplo Hello-world

Una vez que se ha creado el grupo Docker y que se ha añadido el usuario, se puede ejecutar de nuevo el mismo ejemplo del hello-world sin necesidad de otorgarle permisos de administrador, tal como se puede comprobar en la figura 9

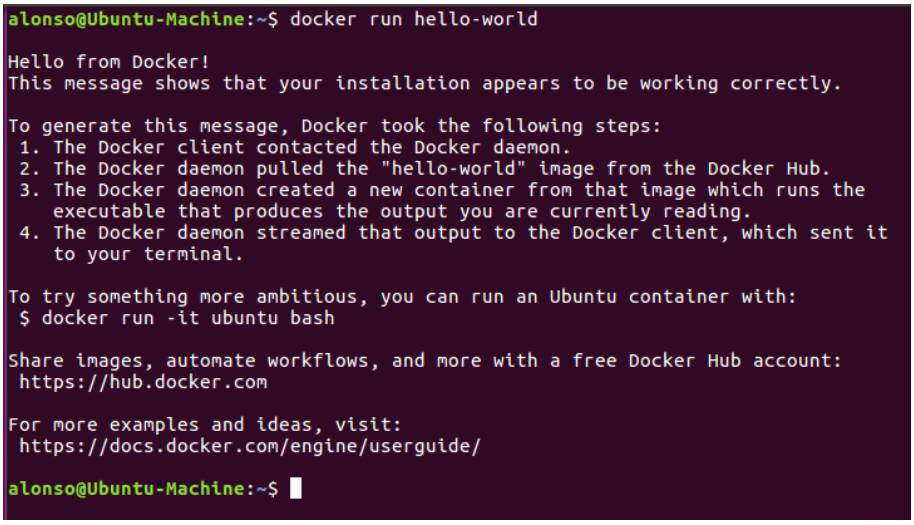

Figura 9: Comprobación del ejemplo luego de añadir el usuario al grupo Docker

Lo anterior corresponde a la documentación de instalación y ejecución del servicio de Docker, pero también se instaló el servicio de contenedor llamado Vagrant, el cual luego de ser adecuadamente instalado, se levanto y se ingreso por medio de una sesión de SSH. La figura 10 muestra que primero se utilizó el comando "vagrant up"para levantar el contenedor y el mensaje que muestra todo el proceso de levantamiento del mismo, además se muestra que con el comando "vagrant ssh"se logró ingresar al contenedor, lo cual se puede notar con el usuario que se encuentra en la linea de comando de la consola, la cual pasa de ser el usuario root para ingresar al directorio principal en el contenedor, el cual corresponde al directorio principal del repositorio asignado para la tarea.

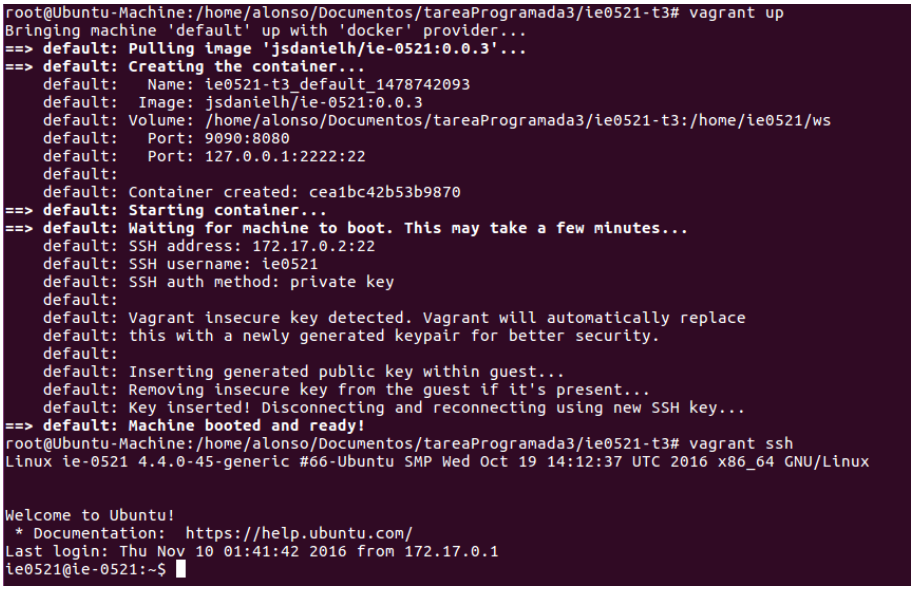

Figura 10: Levantamiento del servicio e ingreso al contenedor vía SSH

A continuación se debía corroborar que el toolchain para el OR1200 se encontraba adecuada-

mente instalado lo cual se pudo asegurar luego de ejecutar los comandos mostrados en la figura 11.

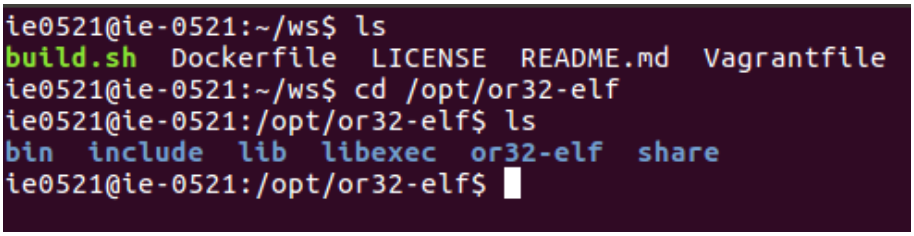

Figura 11: Toolchain debidamente instalado

Para poder entender mejor, se investigó que es un toolchain y cual es la aplicación específica para una aplicación hecha en un contenedor con la arquitectura OR1200, el cual utiliza el toolchain de GNU, el cual consta de un paquete con varias y diferentes herramientas de desarrollo de software, las cuales en este caso nos ayudaron a desarrollar la creación, compilación y simulación de un código en lenguaje C dentro del contenedor.

Este concepto del toolchain nos lleva también a un concepto sumamente importante como lo es el de la compilación cruzada(Çross compilation"), lo cual corresponde a la compilación de un código llevada a cabo en una computadora para que esta sea ejecutada más tarde en una arquitectura de hardware diferente. En este caso al sistema de la computadora se le denomina "host"(anfitrión), mientras que al sistema que hospedará más adelante la aplicación desarrollada se le conoce como "target"(objetivo). Como un ejemplo de cross compilation se puede mencionar el desarrollo de aplicaciones para teléfonos celulares, las cuales se desarrollan desde una computadora para que sean utilizadas en la arquitectura del celular.

Para la parte de la creación del programa en lenguaje C se utilizó el editor de texto "vi", el cual se encuentra incluido en el toolchain mencionado anteriormente y se creó el código mostrado en la figura 12

```c
#include <stdio.h>


//la siguiente función se encarga de devolver el valor esperado de acuerdo al valor de los argumentos
//variando la  operación de acuerdo al argumento 7 tal como se muestra:
int funcion(int arg1, int arg2, int arg3, int arg4, int arg5, int arg6, int arg7){
  if (arg7 > 4) {
    return(arg1 + arg2 + arg3 + arg4 - arg5 - arg6);
  }
  else if (arg7 <= 4) {
    return(arg1 + arg2 + arg3 + arg4 + arg5 - arg6);
  }

}
int main(int argc, char const *argv[]) {
  //declaración de las variables
  int arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg78;
  //inicialización de las variables
  arg1 = 3; arg2 = 8; arg3 = 5; arg4 = 1; arg5 = 9; arg6 = 0; arg7 = 5; arg78 = 3;

  //imprime los resultados
  printf("Los argumentos enviados a la función fueron:\n");
  printf("arg1 = %d, arg2 = %d, arg3 = %d, arg4 = %d, arg5 = %d, arg6 = %d, arg7 = %d\n",
    arg1, arg2, arg3, arg4, arg5, arg6, arg7 );
  printf("El resultado es: %d\n", funcion(arg1, arg2, arg3, arg4, arg5, arg6, arg7));
  printf("\nAhora se cambia el valor del sétimo argumento\n");
  printf("Los argumentos enviados a la función fueron:\n");
  printf("arg1 = %d, arg2 = %d, arg3 = %d, arg4 = %d, arg5 = %d, arg6 = %d, arg7 = %d\n",
    arg1, arg2, arg3, arg4, arg5, arg6, arg78 );
  printf("El resultado es: %d\n", funcion(arg1, arg2, arg3, arg4, arg5, arg6, arg78));
  return 0;
}
```

Figura 12: Programa en lenguaje C

Para dar por un hecho que el programa deba efectivamente el resultado esperado, se ejecutó en la consola de linux para observar el resultado, el cual se muestra en la figura 13

```
alonso@Ubuntu-Machine:~/Documentos/tareaProgramada3$ ./prueba
Los argumentos enviados a la función fueron:
arg1 = 3, arg2 = 8, arg3 = 5, arg4 = 1, arg5 = 9, arg6 = 0, arg7 = 5
El resultado es: 8

Ahora se cambia el valor del sétimo argumento
Los argumentos enviados a la función fueron:
arg1 = 3, arg2 = 8, arg3 = 5, arg4 = 1, arg5 = 9, arg6 = 0, arg7 = 3
El resultado es: 26
alonso@Ubuntu-Machine:~/Documentos/tareaProgramada3$
```

Figura 13: Ejecución del programa creado

En esta última figura se puede ver como en la primera ocasión se llama a la función con un séptimo argumento mayor que cuatro y se obtiene el resultado esperado, de igual forma al llamar a la función con un séptimo parámetro menor que 4, comprobando así su adecuado funcionamiento.

Luego de compilar este código y de desensamblar el binario generado por el mismo, se obtuvo el archivo "main.lst.[el] cual contiene todo el set de instrucciones utilizadas por la arquitectura OR1200 para la ejecución del programa. Estas instrucciones se analizan más a fondo a continuación.

3.1.1. Investigación sobre las instrucciones utilizadas por la arquitectura

Las diferentes instrucciones generadas para el programa y sus respectivas funciones fueron las siguientes:

Para la función main:

- l.sw: posee la forma: [l.sw offset(rA), rB] y lo que hace es guardar el contenido del registro rB en la posición dada por el offset dado hacia el registro rA.

- l.addi: posee la forma: [l.addi rD, rA, I] donde el valor inmediato se le agrega al valor del registro rA para formar el resultado que se guardará en el registro rD.
- l.movhi: posee la forma [l.movhi rD, k] y al valor k se correrá en 16 bits a la izquierda.
- l.ori: posee la forma [l.ori rD, rA, k] extiende en ceros el valor de k y se hace una operación bitwise OR, cuyo resultado se guardará en el registro rD.
- l.jal: es el típico "jump and link", donde se salta a una cierta dirección indicada por un registro y se guarda la dirección siguiente en otro registro especial para esta instrucción.
- l.nop: en esta instrucción no se hace ninguna acción, excepto para tomar un ciclo de reloj, por lo que se le llama instrucción de espera.
- l.lwz: posee una estructura de la forma [l.lwz rD, I(rA)] y se trae el contenido que alla en el offset del registro rA para que sea guardado en el registro rD.
- l.jr: tiene la forma: [l.jr rB], donde lo que se realiza es el salto a la posición a la que apunte el contenido del registro rB.

Además de las instrucciones anteriores, en la función creada para pasarle los argumentos se encontraron estas otras instrucciones:

- l.sflesi: la cual tiene una forma: [l.sflesi rA, I] y lo que hace esta instrucción es hacer una comparación de la forma $if(rA \leq I)\{puthightheflag\}else\{putinlowtheflag\}$
- l.j: posee la estructura: l.j N, donde la dirección N se multiplica por 4 para que sea una dirección conocida y se realiza el salto a esta dirección.
- l.sfgtsi: con un formato: l.sfgtsi rA, I y lo que hace es una comparación del registro rA y el valor I del tipo $if(rA < I)\{putinhightheflag\}else\{putinlowtheflag\}$
- l.bf: esta instrucción se llama de la forma: l.bf N y lo que hace es sumarle la dirección efectiva del valor de N a la dirección actual del programa y se da el salto en caso de que la bandera de la instrucción que se explico justo antes de esta, este en alto, en cuyo caso el salto se dará al siguiente pulso de reloj.
- l.add: posee la forma: l.add rD, rA, rB y lo que hace es sumar los registros A y B y guarda el resultado en el registro D.
- l.sub: posee una estructura igual a la de la instrucción l.add mostrada anteriormente, solo que en este caso se realiza una resta en lugar de una suma.

Todos los archivos generados en esta tarea se pueden encontrar en el siguiente repositorio `https://github.com/alonso193/ie0521-t3`

En este se encuentra el código en lenguaje de ensamblador generado por la arquitectura empleada y el código en lenguaje C que se creó para la tarea.

Si se comparan ambos códigos se puede observar que el compilador utiliza las instrucciones de l.sw para reservar el espacio en memoria de los diferentes argumentos, luego se realiza la asignación

de el valor de estos argumentos con esta misma instrucción.

En las figuras 14 y 15 se puede observar una comparación de los códigos que corresponden a las instrucciones equivalentes en cada uno de los lenguajes donde se reserva el espacio de los argumentos y luego se asignan los valores, que como se observa, son los mismos.

```c
int main(int argc, char const *argv[]) {
  //declaración de las variables
  int arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg78;
  //inicialización de las variables
  arg1 = 3; arg2 = 8; arg3 = 5; arg4 = 1; arg5 = 9; arg6 = 0; arg7 = 5; arg78 = 3;
```

Figura 14: Código en C correspondiente a la primera parte del main

```
00002310 <main>:
    2310:       d7 e1 17 f8     l.sw 0xfffffff8(r1),r2
    2314:       9c 41 00 00     l.addi r2,r1,0x0
    2318:       d7 e1 4f fc     l.sw 0xfffffffc(r1),r9
    231c:       9c 21 ff b4     l.addi r1,r1,0xffffffb4
    2320:       d7 e2 1f d4     l.sw 0xffffffd4(r2),r3
    2324:       d7 e2 27 d0     l.sw 0xffffffd0(r2),r4
    2328:       9c 60 00 03     l.addi r3,r0,0x3
    232c:       d7 e2 1f f4     l.sw 0xfffffff4(r2),r3
    2330:       9c 60 00 08     l.addi r3,r0,0x8
    2334:       d7 e2 1f f0     l.sw 0xfffffff0(r2),r3
    2338:       9c 60 00 05     l.addi r3,r0,0x5
    233c:       d7 e2 1f ec     l.sw 0xffffffec(r2),r3
    2340:       9c 60 00 01     l.addi r3,r0,0x1
    2344:       d7 e2 1f e8     l.sw 0xffffffe8(r2),r3
    2348:       9c 60 00 09     l.addi r3,r0,0x9
    234c:       d7 e2 1f e4     l.sw 0xffffffe4(r2),r3
    2350:       9c 60 00 00     l.addi r3,r0,0x0
    2354:       d7 e2 1f e0     l.sw 0xffffffe0(r2),r3
    2358:       9c 60 00 05     l.addi r3,r0,0x5
    235c:       d7 e2 1f dc     l.sw 0xffffffdc(r2),r3
    2360:       9c 60 00 03     l.addi r3,r0,0x3
    2364:       d7 e2 1f d8     l.sw 0xffffffd8(r2),r3
    2368:       18 60 00 01     l.movhi r3,0x1
```

Figura 15: Código en ensamblador correspondiente a la primera parte del main

Dentro del main en ambos códigos se realizan igualmente asignaciones a memoria de los diferentes caracteres utilizados para mostrar en pantalla los textos necesarios, aparte de esto se realiza un salto que llama a la función de la forma jump and link l.jal. Ya dentro de la función se puede observar como el compilador de la arquitectura OR1200 realiza un pequeño cambio en el orden en el que se realizan las comparaciones de los argumentos, probablemente para realizar una optimización de compilador.

Nuevamente, ahora en las figuras 16 y 17 se puede observar una comparación de la función creada en cada uno de los lenguajes. Se observa como en las direcciones 2270 y 22b4 del código de ensamblador se observan las dos comparaciones realizadas para determinar si el argumento 7

es mayor o menor-igual que 4 respectivamente, tal como se mencionó, lo realiza en orden diferente al establecido por el código en C.

```c
int funcion(int arg1, int arg2, int arg3, int arg4, int arg5, int arg6, int arg7){
  if (arg7 > 4) {
    return(arg1 + arg2 + arg3 + arg4 - arg5 - arg6);
  }
  else if (arg7 <= 4) {
    return(arg1 + arg2 + arg3 + arg4 + arg5 - arg6);
  }

}
```

Figura 16: Código en C de la funcion

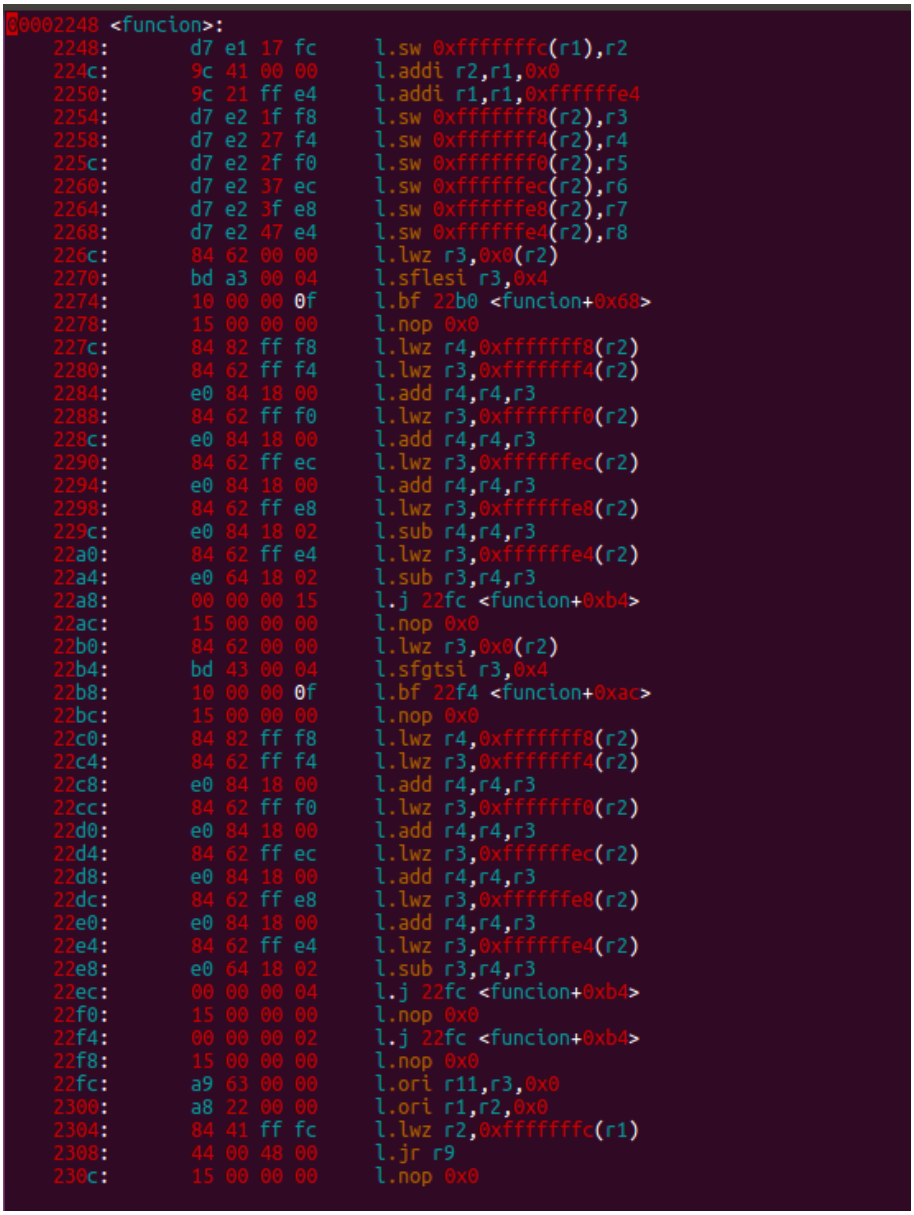

Figura 17: Código en ensamblador correspondiente a la función

4. Conclusiones

- Se logró la instalación adecuada de las herramientas docker y vagrant, las cuales fueron de utilidad para levantar el servicio del contenedor que se utilizó.
- En dicho contenedor se implemento una arquitectura del tipo OR1200, de forma que se pudiera apreciar las características de esta misma en cierta forma aislado de alguna manera del host que albergaba al contenedor.
- Para la creación y compilación del código así como el desensamblaje del binario generado por el mismo se utilizaron las herramientas del toolchain del OR1200/GNU.
- Lo realizado en esta tarea es un ejemplo de lo que es el çross compilation"donde una arquitectura "host", en este caso ubuntu, alberga un contenedor con una arquitectura de hardware "target", pudiendo simular un programa sencillo en la arquitectura del contenedor sin invadir el "host".

5. Referencias

1. Lambrant, D. & J. Baxter (2012). *OpenRISC 1200 IP Core Specification (Preliminary Draft)*. Disponible en https://github.com/openrisc/or1200/blob/master/doc/openrisc1200_spec.pdf

2. OPENCORES. (2012). *OR1200 OpenRISC processor*. Disponible en http://opencores.org/or1k/OR1200

3. OPENCORES *et al.* (2014). *OpenRISC 1000 Architecture Manual*. Disponible en https://github.com/ope arch-1.1-rev0.pdf?raw=true

4. Patterson, D. & J. Hennessy (2007). *Computer Architecture: A Quantitative Approach*. Cuarta Edición, USA: Morgan Kaufmann.