
Tarea 1

Simuladores ISA y Debug

Fecha de asignación: 29 febrero 2024	Fecha de entrega: 7 marzo 2024
Grupo: 1 persona	Profesor: Luis Chavarría Zamora Jason Leitón Jiménez

En esta tarea se explorarán 3 sets de instrucciones conocidos mundialmente en la industria, entre ellos: RISC-V, x86 y ARM. Las herramientas aquí presentadas son usadas profesionalmente.

Si usa otra herramienta diferente a las listadas en este enunciado, debe indicarle al profesor. Si no se cuenta con aval de ese cambio, este trabajo no será calificado y tendrá nota cero. Solo se considerarán otros simuladores que se ejecuten por línea de comandos. No es permitido el uso de herramientas como Ripes o Visual para la entrega final¹.

1. Instalación de herramientas

Nota importante: El profesor utilizó Ubuntu 18.04 LTS. Se recomienda usar como idioma para el distro de Linux el idioma Inglés, porque la comunidad y las soluciones en Inglés es más grande y al menos 20 GB no usados.

1.1. RISC-V

Para esta sección se basó en el material de [RV8](#), [RISC-V Simulator](#) y [Risc-V Assembly Language Hello World](#). Para instalar la herramienta siga los siguientes pasos:

1. Abra un terminal (puede usar el comando rápido `Ctrl + Alt + T`) y colóquese en su dirección de trabajo, se recomienda:

```
cd Documents
mkdir isa && cd isa && mkdir riscv && cd riscv
```

2. Instale los siguientes paquetes:

```
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk patchutils
sudo apt-get install build-essential bison flex texinfo gperf libtool bc zlib1g-dev device-tree-compiler
sudo apt-get install libexpat1-dev
```

¹estas herramientas manejan las direcciones de memoria sin considerar un esquema de memoria realista

3. Se clona el repositorio con el toolchain de RISCv (dependiendo de la velocidad del Internet y la máquina puede durar una hora clonando):

```
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
```

4. Se definen las variables de entorno:

```
export RISCv=$(pwd)
export PATH=$PATH:$RISCv/bin
```

5. Se define el toolchain a construir:

```
cd riscv-gnu-toolchain
mkdir build
cd build
../configure --prefix=$RISCv --with-arch=rv32i
```

6. Se ejecuta el proceso de construcción (dura como una hora):

```
sudo make
```

7. Se procede a construir el simulador RV8, para esto exporte la variable y clone el repositorio:

```
git clone https://github.com/rv8-io/rv8.git
```

8. Se actualizan los contenidos del repositorio (puede durar unos cuantos minutos):

```
cd rv8
git submodule update --init --recursive
```

9. Se procede a instalar la herramienta:

```
sudo make
sudo make install
```

Antes de comprobar el funcionamiento recuerde lo siguiente: **el toolchain es necesario para realizar la compilación del programa, entonces, cada vez que abra un nuevo terminal o reinicie el computador debe ejecutar el paso (4), solamente. Recuerde usar la dirección correcta.**

Para comprobar el funcionamiento realice lo siguiente:

1. Genere una carpeta de pruebas:

```
mkdir test && cd test
```

2. Escriba un archivo de ensamblador, para esto escriba lo siguiente:

```
gedit test.s &
```

- a) Escriba lo siguiente en el gedit (sección de código tomada de [Ripes](#)):

```
.global _start

.text
_start:

test_1:
    # Suma moviendo valores
    li a1, 0x00000003
    li a2, 0x00000007
    add a3, a1, a2
    # Suma cargando valores desde memoria
    addi sp, sp, -56
    sw a2, 0(sp)
    lw a5, 0(sp)
    add a6, a1, a5
    bne x30, x29, fail

pass:
    li a0, 42
    li a7, 93
    ecall
fail:
```

```
li a0, 0
li a7, 93
ecall
```

b) Guarde el archivo y cierre.

3. Debe compilarlo y enlazarlo:

```
riscv32-unknown-elf-as -march=rv32imafdc -o test.o test.s
riscv32-unknown-elf-ld -o test test.o
```

4. Ahora ejecútelo:

```
rv-jit test
```

Se nota que no muestra nada, esto es porque el programa no tiene ninguna impresión en pantalla.

Muchas veces pensamos que sólo podemos identificar problemas de código usando `print`, sin embargo, en este documento veremos cómo usar la capacidad de `debug` de RV8.

1. Primero, es necesario saber las direcciones del programa, como se programó en ensamblador es un mapeo uno a uno. Para observar esto ejecute:

```
riscv32-unknown-elf-objdump -d test &> test.txt && gedit test.txt &
```

2. Si se revisa el archivo `test.txt` se observa que cuenta con direcciones, iniciando con 00010054. Esta será la dirección inicial.

3. Para usar la funcionalidad de *debug* ejecute el siguiente comando:

```
rv-jit -d test
```

4. Si se desea revisar el estado de los registros ejecute:

```
reg
```

5. Si se desea ejecutar una instrucción del procesador ejecute ²:

```
run 1
```

Se observa que realiza la primera operación que es mover 3 al registro **a1** ³

6. Algunas veces deseamos ejecutar con puntos de parada o *breakpoints*, supongamos que deseamos revisar el resultado hasta la suma de **a1** y **a2**, en este caso, si revisamos **test.txt** se observa que es la dirección **1005c** ⁴.

```
break 0x1005c
```

7. Para ejecutar hasta el *breakpoint* y revisar el contenido de los registros ejecute:

```
run  
reg
```

Si se observa el contenido de los registros, se nota que **a1** y **a2** tienen los valores de 3 y 7 respectivamente, mientras que **a3** tiene el valor de **a** que es el resultado de la suma.

8. Si se desea eliminar el breakpoint, ejecute:

```
break 0
```

9. Para ejecutar hasta el final solo ejecute:

```
run
```

Se puede revisar el mapeo de memoria usando **map**, sin embargo, en este caso no se accedió a memoria para acceder a sus contenidos. Para salir del modo **debug** también puede solamente con **quit**.

²puede ser n número de instrucciones pues hay un mapeo 1 a 1 con respecto al código fuente

³en este caso realiza una suma con cero

⁴casi siempre se hace un mapeo 1 a 1 con respecto al código fuente

1.2. x86

Para esta sección se basó en el material de [NASM](#), [NASM Tutorial](#) y [NASM Assembly Language](#). Para instalar y comprobar el funcionamiento de la herramienta siga los siguientes pasos:

1. Abra un terminal (puede usar el comando rápido **Ctrl + Alt + T**) y colóquese en su dirección de trabajo, se recomienda:

```
cd Documents
cd isa && mkdir x86 && cd x86
```

2. Instale el paquete de NASM:

```
sudo apt update
sudo apt install nasm gdb
```

Para comprobar el funcionamiento y la capacidad de *debug* de [GDB](#) realice lo siguiente:

1. Realice el directorio para pruebas:

```
mkdir test && cd test
```

2. Escriba un archivo en ensamblador, para esto escriba lo siguiente:

```
gedit test.asm &
```

- a) Escriba lo siguiente en el gedit (código basado en la sección de código tomada de [Ripes](#)):

```
section .text
    global _start
_start:
    mov rax, 3
    mov rbx, 7
_presuma:
    add rax, rbx
_suma:
```

```
mov rax, 60
mov rdi, 0
syscall
```

b) Guarde el archivo y cierre.

3. Debe compilarlo y enlazarlo:

```
nasm -felf64 -o test.o test.asm
ld -o test test.o
```

4. Ahora ejecútelo:

```
./test
```

Al igual que en la subsección anterior, no se observa nada en la consola pues no hay impresión en consola. Para comprobarlo se usará la herramienta [GDB](#).

En **GDB**, a diferencia de RV8 y su herramienta de *debug* integrada, en esta se pueden poner *breakpoints* aprovechando las etiquetas y no las direcciones, en este caso `_presuma` y `_suma`.

1. Previamente, si se desea, se pueden explorar los elementos del código objeto, usando:

```
objdump -M intel -d test.o
```

Aquí se observa la separación usando etiquetas.

2. Para ejecutar con GDB use:

```
gdb test
```

3. Se plantean dos puntos de parada o *breakpoints*:

```
b _presuma
b _suma
```

Una nota importante es que si coloca una etiqueta antes de un punto ya ejecutado el programa le pregunta si volverá a iniciar.

4. Para ejecutar el código usando los *breakpoints* anteriores ejecute:

```
run
```

Se observa que se detiene en la etiqueta de `_presuma`.

5. Para revisar los contenidos de los registros ejecute:

```
i r
```

Se observa que `rax` y `rbx`, con valores de 3 y 7, respectivamente. Si se desea revisar registros con nombre específico ejecute:

```
i r rax rbx
```

Así se puede revisar sencillamente el contenido de los registros.

6. Para ir al siguiente *breakpoint* ejecute:

```
s
```

7. Ahora se pueden revisar los contenidos de los registros usando:

```
i r rax rbx
```

Aquí se observa que `rax` se actualizó a 10.

8. Si se ejecuta lo siguiente se termina el programa:

```
s
```

9. Adicionalmente, si se desea revisar los contenidos de memoria del programa puede usar:

```
print (int) ADDRESS
```

Sin embargo, en este caso, no se acceso a memoria directamente. Para salir del programa solo ejecute `q`.

1.3. ARM

Para esta sección se basó en el material de [Cross Tools](#). Para instalar y comprobar el funcionamiento de la herramienta siga los siguientes pasos:

1. Abra un terminal (puede usar el comando rápido Ctrl + Alt + T) y colóquese en sudirección de trabajo, se recomienda:

```
cd Documents  
cd isa && mkdir arm && cd arm
```

2. Instale el paquete para compilación cruzada en ARM:

```
sudo apt-get install gcc-arm-linux-gnueabi gcc-arm-none-eabi
```

Para comprobar el funcionamiento realice lo siguiente:

1. Realice el directorio para pruebas:

```
mkdir test && cd test
```

2. Escriba un archivo en ensamblador, para esto escriba lo siguiente:

```
gedit test.s &
```

- a) Escriba lo siguiente en el gedit (código basado en la sección de código tomada de [Ripes](#)):

```
.text  
.global _start  
_start:  
    mov r0, #4  
    mov r1, #7  
    mov r5, #0  
    mov r4, sp  
    strb r1, [r4, r5]!  
_presuma:  
    add r2, r0, r1
```

```
    ldrb r3, [r4, r5]!  
    add r4, r1, r3  
_suma:  
  
    mov r7, #4  
    swi 0  
  
    mov r7, #1  
    swi 0
```

b) Guarde el archivo y cierre.

3. Debe compilarlo y enlazarlo:

```
arm-linux-gnueabi-as test.s -o test.o  
arm-linux-gnueabi-ld test.o -o test
```

4. Ahora ejecútelo:

```
./test
```

Al igual que en la subsección anterior, no se observa nada en la consola pues no hay impresión en consola.

Si tiene problemas para ejecutar `./test` puede usar `qemu`, realizaría el proceso anterior solo que en el paso 4 realice lo siguiente:

```
sudo apt get install qemu  
qemu-arm test
```

El `qemu` funciona como un emulador de arquitectura para ARM en caso de que su computador no soporte esta ejecución.

1. Si desea puede explorar los elementos del código objeto usando:

```
arm-linux-gnueabi-objdump -d test
```

Para revisar la función de *debug* realice lo siguiente:

```
arm-none-eabi-gdb test
```

Una vez adentro se debe cargar el programa con el simulador usando:

```
target exec test  
target sim  
load test
```

Con esto ya realiza los mismos pasos de GDB vistos en x86.

2. Evaluación

Realice el mismo generador de números pseudo-aleatorios (LFSR) del taller 1 y guarde en 10 direcciones de memoria o 10 registros los primeros 10 números de 8 bits generados después de la semilla. Use el mismo valor semilla y colóquela en una posición de memoria o registro de su escogencia.

Realice este ejercicio para cada uno de los 3 ISAs mostrados en este documento, consulte los valores de cada operación usando el *debug* para RISC-V, ARM y x86.

Si usa otra herramienta diferente a las listadas en este enunciado, debe indicarle al profesor. Si no se cuenta con aval de ese cambio, este trabajo no será calificado y tendrá nota cero. Solo se considerarán otros simuladores que se ejecuten por línea de comandos.

2.1. Entregable

Se debe de subir en la sección de Evaluaciones un comprimido con los siguientes archivos:

1. Archivo de reporte en formato libre (Word, L^AT_EX). del ejercicio. Aquí muestra las capturas de como consulta los valores usando *debug* o consola (según sea el caso). **Si no muestra capturas concordantes con lo esperado en consola no se calificará.**
2. Archivos:
 - Ensamblador para ARM, RISC-V y x86.
 - Ejecutable para ARM, RISC-V y x86.

Si tienen dudas puede escribir al profesor al [correo electrónico](#). **Los documentos serán sometidos a control de plagios.** La entrega se debe realizar por medio del TEC-Digital en la pestaña de evaluación. Cualquier entrega tardía después de este punto recibirá una penalización de un punto por minuto.