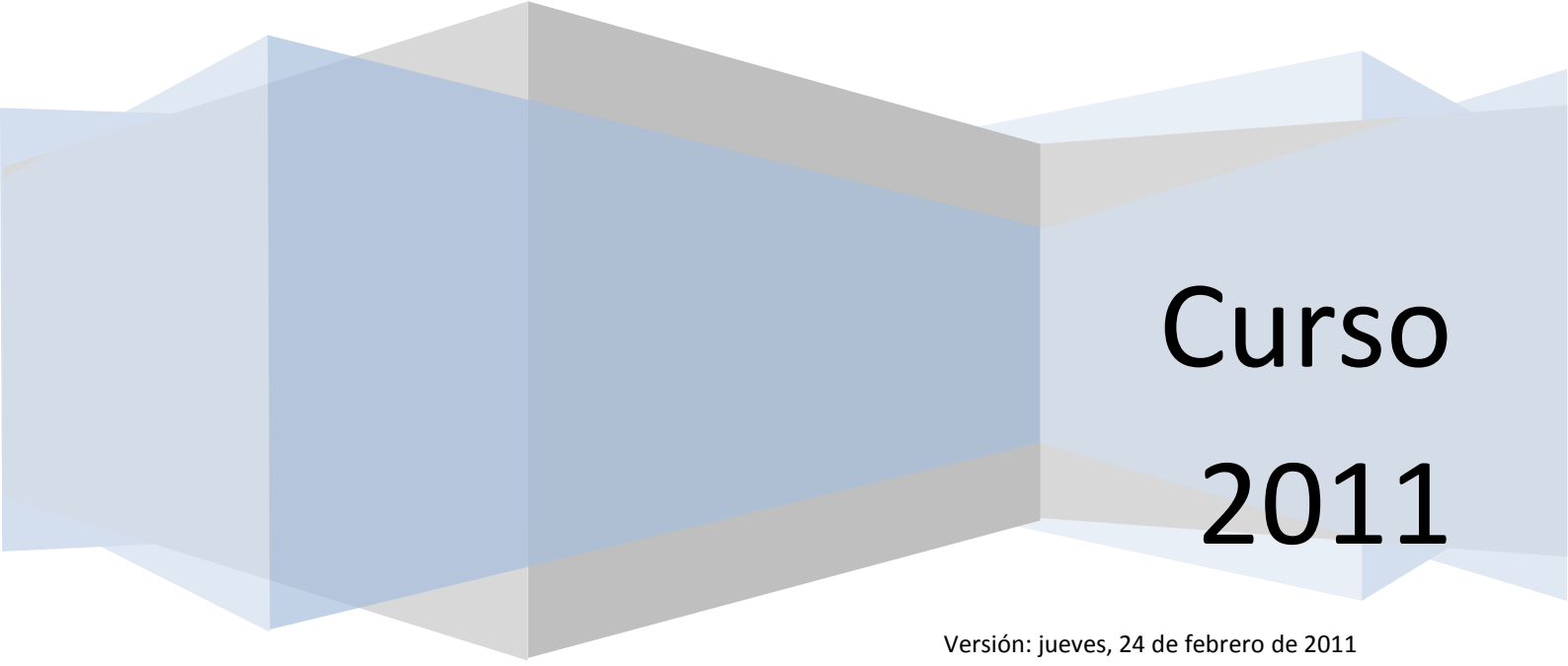


LAB1

# Introducción al Laboratorio de Arquitectura e Ingeniería de Computadores

Laboratorio de Arquitectura e Ingeniería de  
Computadores

Valentin Puente



Curso  
2011

Versión: jueves, 24 de febrero de 2011

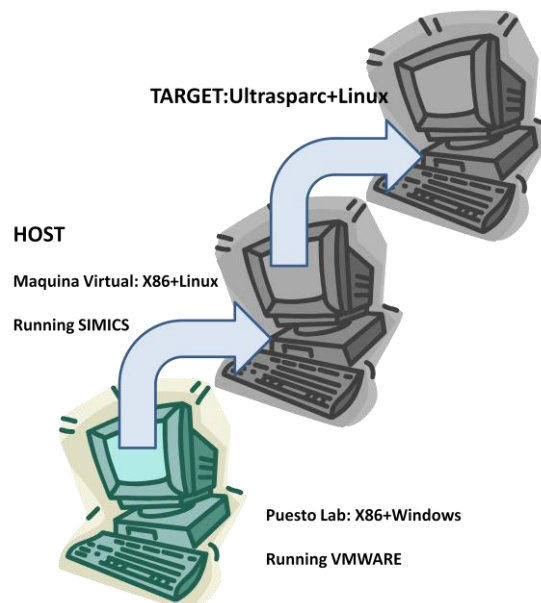
## 1 INTRODUCCIÓN Y OBJETIVOS

El objetivo fundamental de este laboratorio es familiarizarse con el entorno de simulación que emplearemos en el resto de la asignatura. Empezaremos usando **Simics** con algunos experimentos sencillos. Emplearemos un módulo “capturador” de instrucciones que nos permitirá determinar la mezcla de instrucciones de diversos *benchmarks* y hacer posibles recomendaciones arquitecturales a partir de los resultados.

El laboratorio tiene dos partes claramente diferenciadas. Una parte guiada y una parte abierta. La parte guiada de todas las prácticas será examinada al finalizar la asignatura mediante un examen escrito, hasta un máximo de 6 puntos. La parte abierta te permitirá realizar un trabajo mucho más creativo y en función de las tareas desarrolladas y los argumentos que las soporten, podrás obtener hasta 4 puntos adicionales. Las partes guiadas han de ser desarrolladas de forma individual.

### 1.1 TERMINOLOGÍA Y CONVENCIONES

El **Host** se refiere a una máquina virtual **Vmware** en el que estamos ejecutando el simulador. El **target** se refiere al computador que estamos simulando dentro de **Simics**. [workspace] se refiere al directorio donde has creado el entorno de simulación en el apartado 2.1.



Cuando debes escribir algo en la consola del host, aparecerá en el guión de la práctica como:

```
host$ Quiero_que_ejecutes_esto
```

Cuando debes escribir algo en la consola del target, aparecerá en el guión de la práctica cómo:

```
target# Quiero_que_hagas_que_ejecutas_esto
```

Cuando desees que **Simics** haga algo específico, aparecerá en el guión de la práctica como:

```
simics> Quiero_que_cambies_o_hagas_esto
```

## 2 PARTE DIRIGIDA

### 2.1 CONFIGURANDO EL ENTORNO DE TRABAJO

Para realizar esta práctica, se te facilitará una cuenta de acceso en el laboratorio de Arquitectura de Computadores. Los datos de esta cuenta son personales e intransferibles.

Una vez abierta la sesión, vamos a hacer una réplica personal de la maquina virtual. Sitúate en el directorio `/extra/aic_home/` y crea un directorio con tu usuario (`mkdir <username>`). Este directorio de trabajo se encuentra en el equipo local, por lo que deberás utilizar el mismo puesto de trabajo en sesiones sucesivas. Con el fin de asegurar la integridad de tus datos, puedes modificar los permisos al directorio creado, de forma que solamente tú tengas acceso. (`chmod -R og-rwx /extra/aic_home/<username>/`).

Necesitarás copiar dos ficheros al directorio creado. Copia `"/extra/AIC/VMMachine/Other Linux 2.6.x kernel.vmx"` y `"/extra/AIC/VMMachine/Other Linux 2.6.x kernel-000001.vmdk"` en ese directorio. También deberás crear un enlace (`ln -s`) de todos los ficheros `*.vmdk` del directorio `/extra/AIC/VMMachine/`, excepto el que ya te has copiado. Una vez hecho esto, debes arrancar la maquina virtual desde el menú open de *VMware* acceder a `"/extra/aic_home/<username>/Other Linux 2.6.x kernel.vmx"`. Una vez hecho arranca la maquina y responde a *VMware* que la máquina la has movido ("***I moved it***").

Dado que el directorio `/extra/aic_home/<username>` está alojado en el equipo local, se recomienda hacer copias de seguridad si quieres conservar el trabajo entre sesión y sesión. Estas mismas copias se pueden emplear para llevarse después el trabajo a casa y continuar trabajando<sup>1</sup>. Únicamente debes transportar los dos ficheros que copiaste en el párrafo anterior (el `.vmx` y `000001.vmdk`). Cuando lo abras te pedirá localizar los `vmdk` de la maquina base. Te los puedes descargar de [http://www.atc.unican.es/%7evpuente/AIC/AIC\\_1011.tar.bz2](http://www.atc.unican.es/%7evpuente/AIC/AIC_1011.tar.bz2) . Si te lo descargas, procura hacerlo desde la red de la Universidad; ocupa 1.5GB.

---

<sup>1</sup> Opcionalmente, puedes descargarte esta misma máquina virtual a tu equipo personal y reproducir en entorno para realizar las prácticas. Sin embargo, se recomienda emplear los equipos del laboratorio para el trabajo presencial.

La máquina virtual tiene 1 usuarios por defecto: *alumno*. Su password es el mismo que el nombre de usuario. Todas las herramientas necesarias para hacer todas las prácticas de la asignatura están incluidas, por lo que no es necesario tener acceso a la cuenta de root.

Posteriormente y ya dentro de la máquina virtual procedemos a llevar a cabo la configuración del entorno de *Simics* en el usuario alumno. Para ello debemos ejecutar:

```
host$ /net/aic/simics-3.0.31/bin/workspace-setup ~/simics-workspace
```

Por comodidad, en lo sucesivo en todas las prácticas llamaremos al directorio de instalación de *Simics* como `$(SIMICS_INSTALL)`, para ello ejecutaremos:

```
host$ export SIMICS_INSTALL=/net/aic/simics-3.0.31/
```

Podemos hacerlo permanente en el fichero de personalización de la Shell “`$HOME/.bashrc`”

## 2.2 PRIMEROS PASOS CON SIMICS

Ahora que tu *workspace* ha sido creado, estás listo para comenzar una simulación. Navega por él y carga una configuración denominada *Bagel* (que corresponde a una Sun Ultra III Sunfire).

```
host$ cd ~/simics-workspace
```

```
host$ ./simics targets/sunfire/bagle-common.simics
```

*Simics* iniciará y te presentará su interface por línea de comandos (`simics>`). Además aparecerá una nueva terminal gráfica (*xterm*). Si tienes algún problema con el servidor X11 desde el que has intentado iniciar el simulador, este dará un error debido a que no es capaz de presentar el *xterm*. Este *xterm* representa el estado de ejecución del *target*. La simulación arranca con el *target* en estado de pausa (su procesador está parado). Para iniciar la simulación ejecutamos:

```
simics> continue
```

En este momento el target se iniciará, y si dispones del disco de arranque del sistema simulador, veras en el *xterm* el proceso de arranque del target. Como ves, se trata de una estación de trabajo Sun UltraSparc, con un procesador que no tiene nada que ver con el del host (Intel X86). Transcurrido un tiempo aparecerá un *prompt* de login. Introduce como username “**root**” y sin password. El sistema simulado es un Linux convencional. Todo va un “poco” más lento. Ten en cuenta que estamos simulando un nuevo computador al completo! Se hace con tal detalle que el propio sistema operativo que se está ejecutando, lo hemos convencido que es un sistema real. Se están simulando, desde un punto de vista funcional, con tanto detalle todos los elementos hardware de las estación de trabajo, que Linux la considera como real. Nótese que esto es mucho más complejo que una máquina virtual o emulación.

Podemos interrumpir la simulación en cualquier momento pulsando [control-C] en la terminal de Simics. En ese momento el prompt “simics>” reaparecerá. A partir de ese momento no podrás interactuar de nuevo con el **target**. Para poder hacerlo debes volver a introducir “continue” en la interfaz de comandos de Simics.

### 2.2.1 CHECKPOINTING

Una vez arrancado el sistema, es recomendable salvar su estado en disco para poder reiniciar la simulación desde ese punto la siguiente vez que vengamos a trabajar. De este modo nos evitaremos tener que esperar otra “eternidad” para verla arrancar otra vez. Este proceso se denomina *Checkpointing* y es fundamental para trabajar con el simulador. Nos permitirá reproducir con precisión el mismo punto de partida cuando tengamos que hacer de forma repetitiva diferentes medidas o simplemente evitarnos esperas inútiles. Para crear un checkpoint llamado **golden-checkpoint.conf** ejecutaremos:

```
simics> write-configuration golden-checkpoint.conf
```

El checkpoint es salvado en tu workspace. La próxima vez que quieras recargar el estado del sistema simulado, simplemente ejecuta:

```
host$ ./simics -c golden-checkpoint.conf
```

Una cosa importante a tener en cuenta, es que los checkpoints son incrementales. Esto quiere decir que sólo guardan las diferencias entre el checkpoint actual y el generado previamente. Esto es bueno porque ahorra montones de espacio en disco pero puede ser problemático; si borramos el checkpoint intermedio no podremos recuperar el estado a partir del último. Ten en cuenta esto cuando estés borrando, creando u ordenando los checkpoints. Sé ordenado con ellos!!! (evitaras muchas pérdidas de tiempo).

### 2.2.2 MONTANDO EL SISTEMA DE FICHEROS DEL HOST

La próxima cosa a realizar es lograr añadir ficheros en el target desde el mundo “real” (el host). Para llevar esto a cabo **Simics** permite montar el sistema de ficheros del host dentro del target. Para montarlos, debes ejecutar:

```
target# mount /host
```

A partir de ese momento, en el directorio **/host** del target vemos el sistema de ficheros del host. Podemos ejecutar un **cp** de los ficheros (ejecutables por ejemplo) que vayamos a emplear en nuestra simulación dentro de los directorios del target (**/root** puede servir). Recuerda que si no hacemos un checkpoint del estado del target, los cambios son volátiles (los perderemos al cerrar **Simics**).

### 2.2.3 COMANDOS DE SIMULACIÓN

La simulación de *Simics* se puede controlar a través de una gran variedad de comandos sencillos. Ya sabes cómo parar y reiniciar la simulación (con **control+C** y **continue**). Algunos comandos útiles son los que permiten ejecutar un número prefijado de instrucciones (step-instruction) o ciclos de reloj (step-cycle). En esta práctica todas las instrucciones tienen un CPI 1 (es la configuración por defecto de Simics), luego los siguientes comandos son equivalentes.

```
simics> step-instruction 100_000
```

```
simics> step-cycle 100_000
```

Los números grandes pueden incluir el carácter “\_” para facilitar su legibilidad. La mayoría de instrucciones tienen un formato abreviado, siendo por ejemplo **step-instruction** equivalente a **si** o **continue** equivalente a **c**. El CLI (*Command Line Interface*) de *Simics* incorpora las funcionalidades básicas de edición y autocompletado de comandos típica de la shell **bash**.

Otros comandos de utilidad, muestran por pantalla multitud de parámetros y variables de estado del sistema simulado. Por ejemplo **pregs** y **fregs** muestran los valores de los registros de enteros y de punto flotante del procesador del sistema simulado. Como ves los procesadores Sparc no tienen los mismos registros que los procesadores MIPS. **read-regs** y **write-regs** puede ser utilizados para ver o alterar el valor de algún registro en particular. **get** y **set** hacen lo mismo con posiciones de memoria. **ptime** muestra cuanto tiempo simulado ha transcurrido. Para ver las estadísticas detalladas del procesador emplearemos **pstats**.

El comando **display** puede ser usado para ejecutar un comando cada vez que la simulación sea interrumpida. Por ejemplo:

```
simics> display ptime
```

Mostrara el resultado del comando **ptime** cada vez que la simulación es interrumpida. Para dejar de mostrar ese comando en cada interrupción, ejecutaremos:

```
simics> undisplay 1
```

Por último, el carácter “!” indicara que el comando debe ser ejecutado en el host (Es equivalente a **host\$ [comando]**). En cualquier caso es más sencillo mantener varios terminales abiertos y trabajar directamente en un shell para ejecutar comandos en el host.

Otros muchos comandos pueden ser encontrados en la Guía de Usuario de Simics. Disponible en la Web de la asignatura.

### 2.2.4 SEGUIMIENTO DE INSTRUCCIONES GENÉRICO

Para crear un seguimiento o **tracer** de instrucciones, ejecutaremos:

```
simics> new-tracer
```

Esto cargara un módulo estándar de Simics que mostrara instrucciones y accesos a datos en la consola de Simics cada vez que se ejecuten. El nombre asignado a este módulo cargado será trace0. Para iniciar y parar el seguimiento debemos ejecutar:

```
simics> trace0.start  
simics> trace0.stop
```

Cuando el tracer está activo, empezara a hacer un seguimiento de la ejecución tras continuar la simulación. Prueba a iniciar el tracer y ejecutar un número reducido de instrucciones (con **step-instruccion**) para comprobar la información facilitada. El módulo básico es muy sencillo, por lo que facilitaremos una versión modificada para hacerlos siguientes apartados de la práctica.

## 2.3 DETERMINACIÓN DE LA MEZCLA DE INSTRUCCIONES

Para esta parte emplearemos una versión modificada del módulo trace, que nos permitirá determinar la mezcla de instrucciones de diversos benchmarks. El primer paso es copiar el código fuente del tracer en tu directorio personal:

```
host$ cd ~/simics-workspace  
host$ $SIMICS_INSTALL/bin/workspace-setup --copy-device=trace
```

Ya tenemos copiado el código fuente del tracer original en **~/simics-workspace/modules/trace**.

Obtenemos el fichero lab1.tar.gz del la página de práctica. Para hacer esto puedes descargártelo desde un navegador de la maquina virtual, accediendo al aula virtual o puedes copiarlo desde el host de la máquina virtual. En el directorio **/mnt/hfgs/** tienes las carpetas compartidas del host.

```
host$ cd ~/simics-workspace  
host$ tar xjvf lab1.tar.gz  
host$ find ./lab1/ -exec touch {} \;
```

Reemplazamos los los ficheros **Makefile**, **gcommands.py**, **trace.c**, **trace.h**, y **mix.h** con los facilitados en WebCT en el directorio **lab1/2.5-code**.

```
host$ cp ./lab1/2.5-code/* modules/trace/
```

Compilar el nuevo módulo ejecutando **make** en tu workspace, asegurándote de que la variable de entorno **LANG** este puesta al valor por defecto. En otro caso las macros de compilación de Simics no funcionarían bien.

```
host$ cd ~/simics-workspace
```

```
host$ export LANG=""
```

```
host$ make
```

Debes descomprimir el fichero **benchmarks-1.tar** y asegúrate que el directorio resultante esta accesible en tu workspace.

Reinicia Simics a partir del checkpoint creado previamente. Si no los has creado, inicia con **targets/sunfire/bagel-common.simics** y asegúrate de crearlo una vez finalizado el proceso de arranque del target. Desde el target copia los binarios descomprimidos en el paso previo a un directorio del sistema simulado. Crea un nuevo checkpoint. ¡Recuerda que son incrementales!

Para cada benchmark:

Crear el tracer de instrucciones. Habilita puntos de ruptura mágicos<sup>2</sup>:

```
simics> new-tracer [elige un nombre tracer]
```

```
simics> magic-break-enable
```

```
simics> continue
```

Ejecuta cada uno de los benchmarks con los siguiente comandos (no lo hagas seguido, queremos monitorizar cada uno por separado).

```
target# ./bzip2_sparc -z input.jpg
```

```
target# ./mcf_sparc input.in
```

---

<sup>2</sup> Puedes buscar en el manual de usuario que significan. En cualquier caso de forma escueta, nos permiten desde el sistema simulado mandar comandos a Simics (dentro de la propia aplicación simulada).



```
target# ./soplex_sparc input.mps
```

Rápidamente, cada una de esas ejecuciones, alcanzarán un punto de ruptura y la simulación entrará en pausa, con algo así como esto:

```
[cpu0] v:0x000000000001abc4 p:0x000055f6bc4 magic (sethi 0x40000, %g0)
```

Ejecuta 100,000,000 instrucciones (ciclos), y así llegar a la parte de interés del benchmark (no nos interesa la inicialización y finalización).

```
simics> c 100_000_000
```

Inicia el tracer, con el fichero de salida en el directorio /tmp del host.

```
simics> [el_nombre_que_elegiste].start [fichero_salida]
```

Tracea 1,000,000 instrucciones

```
simics> c 1_000_000
```

Para el tracer. Al final del fichero de salida podremos encontrar una estadística pormenorizada de las instrucciones ejecutadas.

```
simics> [el_nombre_que_elegiste].stop
```

Como puede ver la mezcla de instrucciones diferentes varía considerablemente entre cada uno de los benchmarks. Anota los resultados obtenidos. ¿Qué benchmark crees que intenta resolver un problema numérico? ¿Qué benchmark crees que estará más limitado por memoria? ¿Qué benchmark parece ser bastante dependiente de la efectividad del predictor de saltos?

## 2.4 PROBLEMA DE ANÁLISIS

Supón que el sistema ha sido de tal modo que el CPI promedio de los *loads* y *stores* es de 2 ciclos, las instrucciones aritméticas en enteros es de 1 ciclo y el resto de instrucciones es de 1.5 ciclos en promedio. ¿Cuál será el CPI promedio de la máquina en cada benchmark? ¿Cuál será el *speedup* de cada benchmark si para los *loads* y *stores* se mejora el CPI en 1 ciclo? ¿Tiene sentido la mejora si el CPI de las instrucciones aritméticas cambia a 1.5 ciclos? ¿Cuál es el *speedup* promedio alcanzado?

## 2.5 PROBLEMA DE DISEÑO

Imagina que tu jefe quiere que evalúes una posible modificación del pipeline del sistema. Supón que el pipeline del procesador es muy similar al pipeline clásico de 5 etapas visto en el repaso

de clase, para un procesador MIPS. La modificación propuesta es fusionar las etapas EX y MEM en una sola. En la etapa resultante, la ALU y la memoria se acceden en paralelo. Las instrucciones aritméticas emplearán la ALU dejando la memoria sin usar. Las de acceso a memoria solo emplearán la memoria, dejando la ALU sin usar. Se considera que estos cambios son beneficiosos en términos de área y consumo energético y que no modifican el periodo de reloj del diseño inicial.

En el estándar SPARCV9 de *bagel*, la dirección efectiva de un *load* o *store* es calculada tanto sumando el contenido de dos registros o sumando el contenido de un registro con un valor inmediato (como ocurre en MIPS). El problema con el nuevo diseño es que no tenemos modo de calcular la dirección efectiva, puesto que las instrucciones de acceso a memoria no tienen a su disposición la ALU. Para superar esta limitación, nos planteamos modificar los modos de direccionamiento permitido, restringiéndolo al valor de un registro. Ni tan siquiera podemos calcularlo como la suma del registro con un *offset*.

Con el nuevo diseño cualquier *load* o *store* que emplee el contenido de dos registros o un *offset* no nulo para el cálculo de la dirección efectiva deberá ser descompuesto en dos instrucciones.

Tu trabajo es determinar el incremento en el número total de instrucciones que sufre cada uno de los benchmarks considerados y descubrir así si realmente es beneficioso el cambio o por el contrario degrada el rendimiento.

Modifica el tracer facilitado, en especial el fichero *mix.h*, para hacer un análisis exhaustivo de cada tipo de instrucción de acceso a memoria. Fíjate como es el fichero facilitado en la sección 2.3. Emplea la documentación del repertorio de instrucciones de SPARCV9 facilitado y fíjate cuales son los bits que corresponden a cada tipo de instrucción. ¡No te olvides recompilar el módulo antes de volver a ejecutar la simulación!

¿Cuál es el speedup promedio logrado con la mejora? ¿Qué le dirías a tu jefe?

## 3 PARTE ABIERTA

### 3.1 MEZCLA SINTÉTICA DE INSTRUCCIONES

El objetivo de esta sección es investigar cómo es de eficiente el compilador manejando código que ha sido creado de forma ofuscada por ti.

Usando no más de 15 líneas de C procura crear un código en ensamblador de SPARC que tenga el máximo número posible de instrucciones de salto condicional. Tu código de C puede emplear tantas técnicas desaconsejables de programación como consideres oportunas para llegar a ese objetivo, pero procura incluir en cada línea un sola sentencia del lenguaje y procura no llamar a funciones externas (ni llamadas al sistema!). El código debe finalizar correctamente.

Copia el fichero *mix\_manufacturing.c* en tu workspace, y añade tu código, verificando previamente que el código funciona en el host. Lo puedes hacer como:

```
host$ gcc -I $SIMICS_INSTALL/src/include/ mix_manufacturing.c -o mix
host$ ./mix
```

Una vez comprobado que esto funciona, lo repetiremos desde dentro de la máquina simulada.

```
target# gcc -I $SIMICS_INSTALL/src/include/ mix_manufacturing.c -o mix
target# ./mix
```

¿Te diste cuenta del /host/? Entonces sabrás como incorporar mix\_manufacturing.c. Afortunadamente podemos hacer esto porque la máquina simulada tiene un compilador de C instalado.

Determinar con el tracer que porcentaje de instrucciones son de salto condicional. El mejor lograra máxima nota en el apartado. El peor no obtendrá nada.

### 3.2 CAMBIOS EN EL ISA

Imagina que debes modificar el ISA de SPARCv9 para permitir dos tamaños de instrucción. Con esta modificación, será posible reducir el código máquina de tus programas ya que parte de las instrucciones requerirán menos espacio para ser codificadas. Elige una instrucción del repertorio de instrucciones y propón razonadamente por que debe ser más compacto el tamaño de su codificación. ¿Cómo afectaría esto a los benchmarks que se han ejecutado en los apartados previos? Puedes usar el manual completo de SPARCv9 disponible en <http://developers.sun.com/solaris/articles/sparcv9.html>