

# Programacion de BIOS

## TEORIA Y EJEMPLO PRACTICO

### Introduccion

Antes de comenzar a hablar sobre el código BIOS, se realizara una breve explicacion teorica de lo que es una computadora y como trabaja, esto para poder exponer la razon de ser del programa BIOS, posteriormente se mostrara un ejemplo de arranque usando un sistema Virtual y un BIOS de Código sencillo, se compilara como una prueba del concepto usando una raspberry pi, la cual se ha elegido para reforzar el ejemplo ya que la arquitectura de este sistema es ARM, diferente al sistema emulado x86.

### El modelo de Von Neuman

Uno de los aportes mas importantes en las ciencias computacionales fue propuesto por Jhon Von Neuman, y quedo expresado en su diagrama sobre un sistema de computo (figura 1).

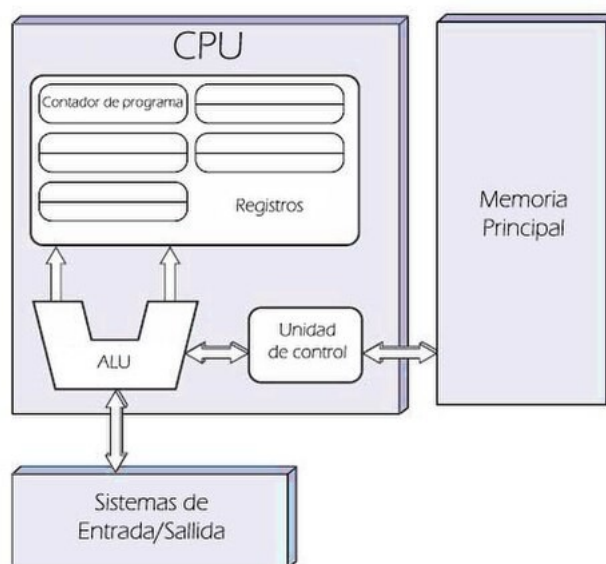


figura 1

En este modelo la memoria principal tomaba un papel muy importante al ser el lugar donde residia el *software*, que es como fue llamado al conjunto de intrucciones que la CPU del sistema de computo debia ejecutar, este tipo de sistema es conocido como computadora de programa almacenado. Y su forma de operar se basa en la interaccion que tiene la CPU con la memoria, mediante sus registros, lee y actua acorde a las instrucciones residentes en la memoria y los

resultados son guardados tambien en la memoria, por lo que pueden usarse para nuevos calculos, toma de decisiones o como instrucciones nuevas.

La forma de interacción de la CPU con la memoria es dirigida por la *Unidad de Control*, la cual es una maquina de estados (vista de una manera formal), que ejecuta el ciclo de instrucciones mostrado en la figura 2.

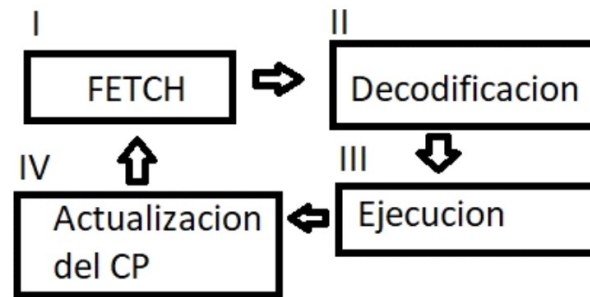


figura 2

Un recorrido por los Estados sería como el listado siguiente:

- 1) *Fetch* se busca la información en la memoria de la instrucción a ejecutar, la cual es indicada por el registro "Contador de programa" de la CPU (el como ese registro se cargo con la dirección sera explicado mas adelante).
- 2) la cpu *Decodifica* la informacion que obtuvo de la memoria para averiguar cual instruccion de las que puede realizar tiene que hacer (dicha lista de instrucciones puede variar entre computadoras diferentes).
- 3) La CPU "lleva a cabo" (*Ejecuta*) la tarea indicada por la instrucción, la cual puede ser mover un dato, o realizar una operacion.
- 4) tras finalizar la ejecución se "Actualiza el Registro Contador de Programa(CP)", el cual usualmente incrementa su valor para indicar que la celda siguiente contigua en memoria contiene la siguiente instruccion.

La CPU se mantiene moviendose en los estados del ciclo de *fetch* mientras este encendida y operando, similar a un motor de combustion interna de cuatro tiempos, el cual inclusive cuando esta esperando el vehiculo en un alto, este ciclo no se detiene.

Para que este modelo funcione, se deben tener ciertas consideraciones iniciales, primero el programa a ejecutar debe estar residente en la memoria de la computadora (el término coloquial es *cargado*). Los registros de la CPU deben tener valores que permitan la transicion de los estados del ciclo de *fetch*, el mas importante seria que el registro "Contador del Programa" contenga la dirección de inicio de la primera instrucción del programa a ejecutar. Surge la primera pregunta, ¿quien coloca el progama en la memoria? y la segunda pregunta, ¿Quien inicializa los registros para que la CPU inicie el ciclo de *fetch* y asi comienza la ejecución del programa?.

La respuesta a la primera pregunta sería que un programa ejecutado por la misma CPU podría realizar la tarea de colocar las instrucciones en la memoria del programa que se desea ejecutar y ese programa al terminar la carga de las instrucciones, sería encargado de actualizar los valores de los registros para dejar todo preparado para la ejecución del nuevo programa recién cargado en la memoria, es decir la misma CPU debería cargar los programas que va a ejecutar. La pregunta se vuelve al problema inicial ¿quién carga a este programa "cargador" de programas? , este problema donde la CPU intenta "iniciarse así misma", se le conoce como **BootStrap**, la solución es usar un "mecanismo externo" que permita preparar la CPU para "valerse por sí misma", a este proceso "externo" se le conoce como **Boot**.

La primera parte del proceso de Boot, inicializar la CPU al estado en el que pueda ejecutar el ciclo de *Fetch* se resuelve usando una máquina de estados externa cuyo único trabajo sea configurar los registros de la CPU (usualmente implementada en un circuito digital especializado en la CPU). Donde el valor del registro *contador del programa* (CP) es fijado a una dirección inicial acordada con antelación por el grupo de diseño de la CPU.

La segunda parte es "proveer de antemano" el primer programa a ejecutar ya "cargado" en celdas de memoria contiguas a la dirección inicial en el CP y reside en un tipo de memoria que es "persistente" para que siempre esté "listo" y no requiera ser "cargado" nuevamente. Este programa tiene como objetivo poder "cargar" otros programas e inicializar los registros de la CPU para que comience a ejecutar dichos programas así como otras tareas adicionales, dicho programa se le conoce como **BIOS**.

Tenemos que la "razón de ser" del programa BIOS es formar parte del proceso de Boot de una CPU, el cual permite tener un sistema listo para cargar programas en memoria y configura los registros de la CPU para que pueda comenzar el ciclo de *fetch* la ejecución de estos nuevos programas.

Una explicación teórica más detallada de este proceso se puede obtener del libro del Doctor Guillermo Levinne (información en la sección de Bibliografía de este documento), los siguientes párrafos se ocuparán de mostrar los aspectos técnicos de la implementación del proceso de BOOT en Hardware usando como ejemplo la implementación propuesta por la arquitectura x86.

## Proceso de Boot (Arranque) de arquitectura x86

Construimos un modelo que representa una arquitectura "ficticia" con el fin de poder explicar mejor el proceso técnico que es bastante complejo, así una mejora moderna del modelo de Von Neuman, consiste en la adición del módulo de acceso directo a memoria.

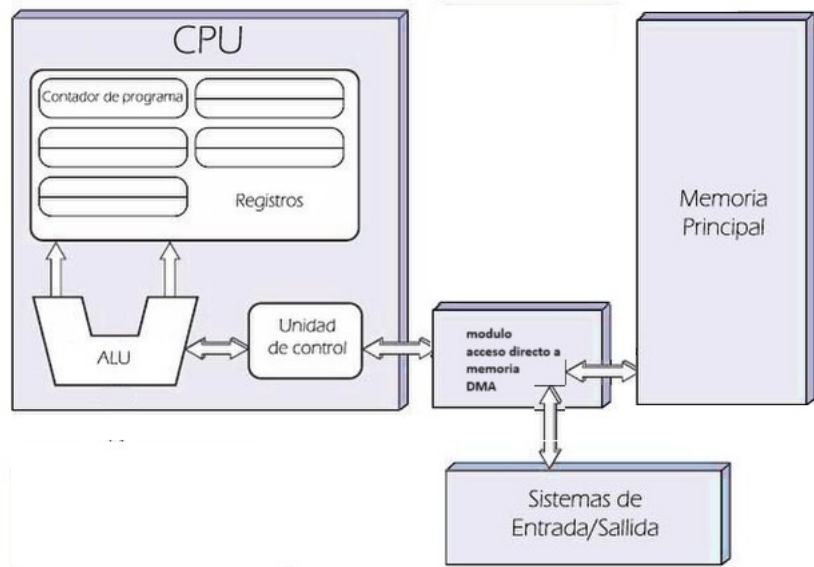


figura 3 (no es una arquitectura real)

De esta manera la interacción de periféricos y el CPU se realiza mediante regiones especiales de memoria (Esto es parcialmente cierto), Esto significa que el CPU solo puede acceder a regiones de memoria, las cuales pueden ser casillas en RAM, casillas en ROM, el teclado, el monitor o algún otro dispositivo periférico más complejo, en este esquema toma importancia en el concepto de mapa de la Memoria (Memory Map) para el proceso de arranque (Boot). Tomando el siguiente párrafo del manual *Minimal Boot* publicado por Intel:

## Power-Up (Reset Vector) Handling

When an IA bootstrap processor (BSP) powers on, the first address that is fetched and executed is at physical address **0xFFFFF0**, also known as the reset vector. This accesses the ROM / Flash device at the top of the **ROM - 0x10**. The boot loader must always contain a jump to the initialization code in these top 16 bytes.

La dirección marcada con verde, sería la cual acordaron los diseñadores que sería la primera dirección que se colocaría en el Registro Contador de Programa (CP), que como vimos anteriormente es requerida "a priori" para la ejecución del ciclo de Fetch. Como marca el documento Minimal Boot, la dirección 0x10 es colocada en nuestra flash por el programa Enlazador como lo muestra la línea 25 del script de "enlazado" (bios.ld en el anexo A parte 2):

```

20     } >ROM      /* Places this first section at the beginning of the ROM */
21     /* the --gap-fill option of objcopy will be used to fill the gap to .main */
22     .main main_address : {
23         *(main)
24     }
25     .reset 4096M - 0x10 : {      /* First instruction executed after reset */
26         *(reset)
27     }

```

El vector de reset está indicado por la etiqueta `.reset` en nuestro código del archivo `bios.S`

(Anexo A parte1) como se muestra en la linea 186:

```
183  /*****
184  /* reset: this section must reside at 0xfffffff0, and be exactly 16 bytes */
185  /*****
186  .section .reset, "ax"
187  /* Issue a manual jmp to work around a binutils bug. */
188  /* See coreboot's src/cpu/x86/16bit/reset16.inc */
189  .byte 0xe9
190  .int init - ( . + 2 )
191  .align 16, 0xff /* fills section to end of ROM (with 0xFF) */
192  /*****/
```

Otro acuerdo hecho por los diseñadores fue el siguiente:

## Initial Processor Mode

When the processor is first powered-on, it will be in a special mode similar to Real Mode, but with the top 12 address lines being asserted high, allowing boot code to be accessed directly from NVRAM (physical address 0xFFFxxxxx). Upon execution of the first long jump, these 12 address lines will be driven according to instructions by firmware. If one of the Protected Modes is not entered before the first long jump, the processor will enter Real Mode, with only 1MB of addressability. In order for Real Mode to work without memory, the chipset needs to be able to alias memory below 1MB to just below 4GB, to continue to access NVRAM. Some chipsets do not have this aliasing and a forcible switch to a normal operating mode will be required before performing the first long jump.

*figura documento minimal boot intel*

Como se explica ahí la memoria RAM no se encuentra disponible aun y como vimos anteriormente el ciclo de Fetch depende de que el programa almacenado se encuentre residente en la memoria, por lo que la única memoria disponible es la ROM y justamente contiene el código que necesitamos para inicializar la memoria RAM (y demás dispositivos), esto no es coincidencia si no producto del modelo (cita del Doctor Levinne), también debemos acomodar nuestro código en la ubicación indicada, lo cual es expresado al Enlazador en el script *bios.ld* en la línea 5:

```
1  OUTPUT_ARCH(i386)          /* i386 for 32 bit, i386 for 16 bit */
2
3  /* Set the variable below to the address you want the "main" section, from bios.S, */
4  /* to be located. The BIOS should be located at the area just below 4GB (4096 MB). */
5  main_address = 4096M - 4K; /* Use the last 4K block */
6
7  /* Set the BIOS size below (both locations) according to your target flash size */
8  MEMORY {
9      ROM (rx) : org = 4096M - 512K, len = 512K
10 }
```

También debemos asegurar que nuestro código tenga una longitud de 512 kb. En este punto nuestra CPU está lista para ejecutar el programa BIOS, ha salido de reset, se encuentra en Modo Real, el registro contador de programa (CP) ha sido inicializado apuntando a la dirección de la memoria donde está ubicado nuestro programa BIOS ya almacenado, por lo que puede comenzar a ejecutar el ciclo de fetch. En el código ejemplo el hardware que inicializaremos será el Super IO para mostrar un mensaje en pantalla y así indicar que la CPU está lista, se ha

resuelto el proceso **BootStrap**, mas no ha terminado todo el proceso de inicializacion **Boot**, aun que funcional nuestro sistema es aun muy "primitivo".

Damos por terminado el proceso de bootstrap de la arquitectura x86, se da a continuacion una explicacion somera de lo que hace el resto de codigo del archivo bios.S, remarcando que desde aqui deberia de comenzar el código propiamente del BIOS.

Para inicializar nuestro Chipset (SuperIO), se encuentra con el alias SUPERIO\_BASE (linea 76 archivo bios.S):

```
75  init_superio:
76      mov  dx, SUPERIO_BASE  /* The PC97338 datasheet says we are supposed  */
77      in   al, dx            /* to read this port twice on startup, but the  */
78      in   al, dx            /* VMware virtual chip doesn't seem to care... */
```

Cuya direccion de entrada esta indicada en la linea 30 del archivo bios.S:

```
29  /* The VMware platform uses an emulated NS PC97338 as SuperIO */
30  SUPERIO_BASE = 0x2e      /* Do NOT believe what you see in the BIOS bootblock: */
31  | | | /* the VMware SuperIO base is 0x2e and not 0x398. */
```

Esta direccion es la indicada por el fabricante del Chip en su hoja de datos :

**TABLE 8. INDEX and DATA Register Address Options and Configuration Register Accessibility**

BADDR Pin		INDEX Register Address	DATA Register Address	Accessible after Reset
1	0			
0	0	398h	399h	Yes
0	1	undefined	undefined	No <sup>a</sup>
1	0	15Ch	15Dh	Yes
1	1	2Eh	2Fh	Yes

a. Apply Plug and Play protocol.

Esto significa que para poder acceder al Super IO chip debemos acceder a su direccion en memoria:

```
64  /* 'init' doesn't have to be at the beginning, so you can move it around, as */
65  /* long as remains reachable, with a short jump, from the .reset section. */
66  .section main, "ax"
67  .globl init /* init must be declared global for the linker and must */
68  init:      /* point to the first instruction of your code section */
69      cli    /* NOTE: This sample BIOS runs with interrupts disabled */
70      cld    /* String direction lookup: forward */
71  (A) { mov  ax, cs /* A real BIOS would keep a copy of ax, dx as well as */
72      ds, ax /* initialize fs, gs and possibly a GDT for protected */
73      mov  ss, ax /* mode. We don't do any of this here. */
74
75  init_superio:
76  (B) { mov  dx, SUPERIO_BASE /* The PC97338 datasheet says we are supposed */
77      in   al, dx /* to read this port twice on startup, but the */
78      in   al, dx /* VMware virtual chip doesn't seem to care... */
79
80      /* Feed the SuperIO configuration values from a data section */
81      mov  si, offset superio_conf /* Don't forget the 'offset' here! */
82      mov  cx, (serial_conf - superio_conf)/2
```

Cargamos la direccion inicial del programa BIOS en el par de registros DS:SS (A) para despues cargar la direccion del Super IO en la seccion "baja" del registro (B), asi ahora el conjunto DS:SS contienen la direccion segmento desplazamiento para "ubicar" al SuperIO.

Con el proposito de enviar un mensaje (cadena de texto guardada en el segmento e datos con la etiqueta *hello string*) por el puerto serial, debemos seguir las instrucciones de la hoja de datos del Super IOS PC397338, la cual indica la direccion de memoria donde esta el dispositivo que maneja la salida serial, la cual usualmente esta conectada a un circuito interno dentro del Super IO que convierte señales seriales en RS232 (en ocasiones puede ser un IC externo) y las envia por un cabezal de Debug.

```
89  init_serial:      /* Init serial port                                */
90      mov si, offset serial_conf
91      mov cx, hello_string - serial_conf)/2
92  write_serial_conf:
93      mov ax, [si]
94      ROM_CALL serial_out
95      add si, 0x02
96      loop write_serial_conf
97
98  print_hello:      /* Print a string                                */
99      mov si, offset hello_string
100     ROM_CALL print_string
101
```

Como muestra el fragmento de codigo del archivo bios.S desde la linea 89 a la 100 realizamos un proceso de bajo nivel para mover un caractera la vez y enviarlo hacia el puerto de salida (linea 94), mediante un Loop que continua hasta que hallamos leído y enviado los caracteres del mensaje alojado en el segmento de datos(B):

```
164  /* Data:
165  *****/
166  superio_conf:
167  /* http://www.datasheetcatalog.org/datasheet/nationalsemiconductor/PC97338.pdf */
168  .byte PC97338_FER, 0x0f /* Enable COM, PAR and FDC */
169  .byte PC97338_FAR, 0x10 /* LPT=378, COM1=3F8, COM2=2F8 */
170  .byte PC97338_PTR, 0x00 /* Make sure COM1 test mode is cleared */
171  serial_conf: /* See http://www.versalogic.com/kb/KB.asp?KBID=1395 (A) */
172  .byte COM_MCR, 0x00 /* RTS/DTS off, disable loopback */
173  .byte COM_FCR, 0x07 /* Enable & reset FIFOs. DMA mode 0. */
174  .byte COM_LCR, 0x80 /* Set DLAB (access baudrate registers) */
175  .byte COM_BRD_LO, 0x01 /* Baud Rate 115200 = 0x0001 */
176  .byte COM_BRD_HI, 0x00
177  .byte COM_LCR, 0x03 /* Unset DLAB. Set 8N1 mode */
178  hello_string: (B)
179  .string "\r\nHello BIOS world!\r\n" /* .string adds a NUL terminator */
180  *****/
```

El inciso (A) muestra los datos que son necesarios para que el Super IO active a su circuito interno que realiza la conversion Serial a RS232.

Aqui concluye la explicacion somera de lo que hace el codigo de ejemplo del archivo bios.S al inicio de estos parrafos es donde deberia de comenzar el código de nuestro BIOS.

En conclusion, un circuito electronico se encarga de encender y "sacar" de Reset a nuestra CPU, apartir de ahi nuestro CPU comenzara a ejecutar instrucciones de una direccion de memoria especifica, como dicha direccion no cambia simplemente debemos asegurar que el incio de nuestro codigo (etiqueta .section main linea 66 del archivo bios.S) se encuentre en esa



dirección, para esto le indicamos al programa enlazador mediante un script el punto de entrada requerido, dicho script aunque tiene alguna complejidad tecnica, no cambiara mucho durante el proceso de diseño de nuestro BIOS, por lo que basta con preocuparse solo algunas veces, por lo que debemos centrarnos mas en como debemos inicializar nuestro hardware, lo cual requiere seguir las especificaciones de las hojas de datos de los fabricantes del hardware conectado a nuestro sistema, utilizando programacion bajo nivel en lenguaje ensamblador, el siguiente capitulo muestra como se implementan los archivos bios.S y bios.ld para generar un archivo bios.rom el cual puede probarse en maquina virtual y comprobar la efectividad de lo mostrado, mas adelante usaremos el ambiente de desarrollo del proyecto libre Coreboot el cual nos permitira abstraer mas los detalles tecnicos y enfocarnos en el desarrollo de un BIOS mas completo.

## Proceso de Compilacion

Debido a que estamos usando una Raspeberry Pi como estacion de diseño, necesitamos realizar una Cross-Compilacion de ARMx64 (microprocesador usado en la tarjeta) para generar código x64\_32/64, así como una maquina virtual para probar el programa ejecutable.

El proceso de compilacion debera hacerse por partes, debido a que debemos de configurar varios aspectos técnicos, la figura muestra los comandos requeridos para llevar acabo cada etapa.

```
rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ ls Inicio
bios_BackUP.rom bios.ld bios.rom bios.S Makefile PDF_Docs
rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ x86_64-w64-mingw32-gcc -c -o bios.o -m32 bios.S 1)
rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ x86_64-w64-mingw32-ld -nostartfile -Tbios.ld -o bios.out bios.o -Map xMemLayout.map 2)
rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ x86_64-w64-mingw32-objcopy -O binary -j .begin -j .main -j .reset --gap-fill=0x0ff bios.out bios.rom 3)
rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ ls Final
bios_BackUP.rom bios.ld bios.o bios.out bios.rom bios.S Makefile PDF_Docs xMemLayout.map
rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ ls
```

Lista de comandos requerida:

1)x86\_64-w64-mingw32-gcc -c -o bios.o -m32 bios.S

Creamos el archivo objeto **bios.o** a partir del archivo fuente (en ensamblador) **bio.S**

2)x86\_64-w64-mingw32-ld -nostartfile -Tbios.ld -o bios.out bios.o -Map xMemLayout.map

Enlazamos el archivo ajustando a direcciones fijas de memoria (a partir del inicio del programa), para cumplir con las especificaciones del Boot Strap para arquitectura X86, usando un listado (script) de enlazado **bios.ld** y se le aplica al archivo objeto **bios.o**, generando el archivo **bios.out**.

3)x86\_64-w64-mingw32-objcopy -O binary -j .begin -j .main -j .reset --gap-fill=0x0ff bios.out bios.rom

Vaciamos el contendio binario del archivo **bios.out** con la herramienta objcopy para construir el sistema de archivos que es compatible con las memorias SP( donde se alojara nuestro BIOS),



esto ayuda a llenar el espacio requerido para la particion con 0.

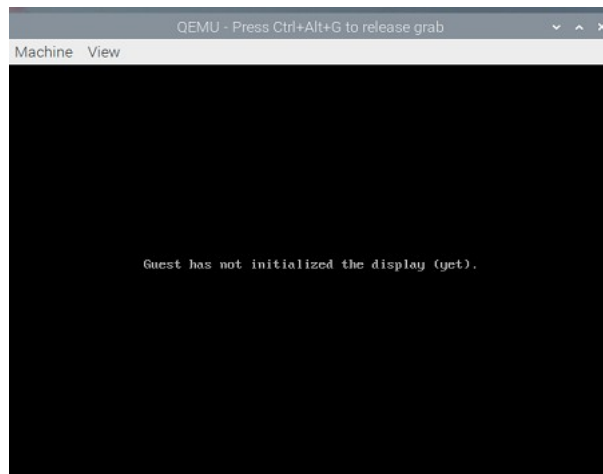
El resultado de la compilacion es un archivo **bios.rom**, el cual puede cargarse directamente en la memoria SPI fisica en la plataforma especifica para la cual construimos el BIOS (El cual no es un programa general). O podemos usarlo para inicializar nuestra maquina virtual.

## Inicializar la maquina virtual con un BIOS especifico.

Las maquinas virtuales tienen la opción de emular su funcionamiento con un bios específico, para ello debemos indicarle donde reside el archivo que contiene el programa de arranque que usara el mecanismo de Boot Strap de la aquitectura especifica a emular (En este caso x86\_32/64), se muestra el comando para inicializa una maquina virtual con Qemu y nuestro archivo **bios.rom**.

```
rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ qemu-system-i386 -bios bios.rom -serial ptty
char device redirected to /dev/pts/2 (label serial0)
```

Nos genera la ventana:

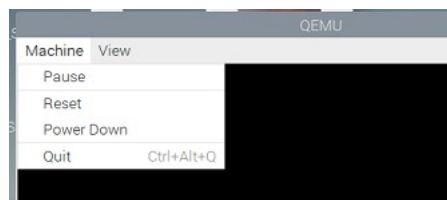


No veremos nada porque la salida esta saliendo por el puerto serial de diagnostico.

tendremos que abri otra ventana de terminal y usar el comando:

**`cat /dev/pts/[numero de puerto]`**

y aplicando reset en la ventana de la VM:



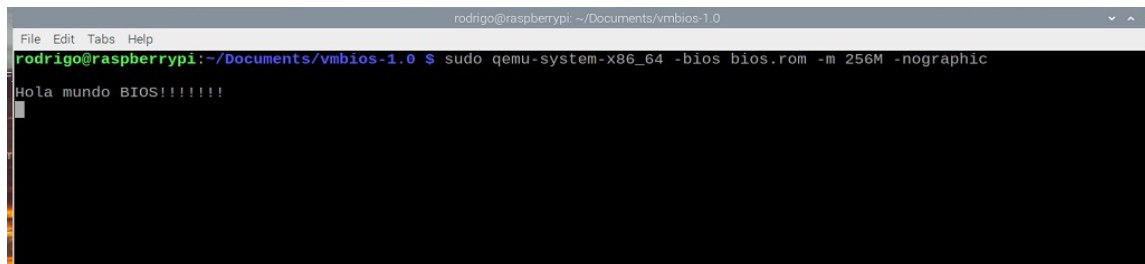
Veremos la salida:

```
rodrigo@raspberrypi:~ $ cat /dev/pts/3
Hola mundo BIOS!!!!!!
```

una manera mas corta seria usando el comando:

```
sudo qemu-system-x86_64 -bios bios.rom -m 256M -nographic
```

La salida sera la siguiente:

A screenshot of a terminal window titled 'rodrigo@raspberrypi: ~/Documents/vmbios-1.0'. The terminal shows the command 'rodrigo@raspberrypi:~/Documents/vmbios-1.0 \$ sudo qemu-system-x86\_64 -bios bios.rom -m 256M -nographic' and its output 'Hola mundo BIOS!!!!!!'. The terminal has a standard Linux prompt and a menu bar with 'File', 'Edit', 'Tabs', and 'Help'.

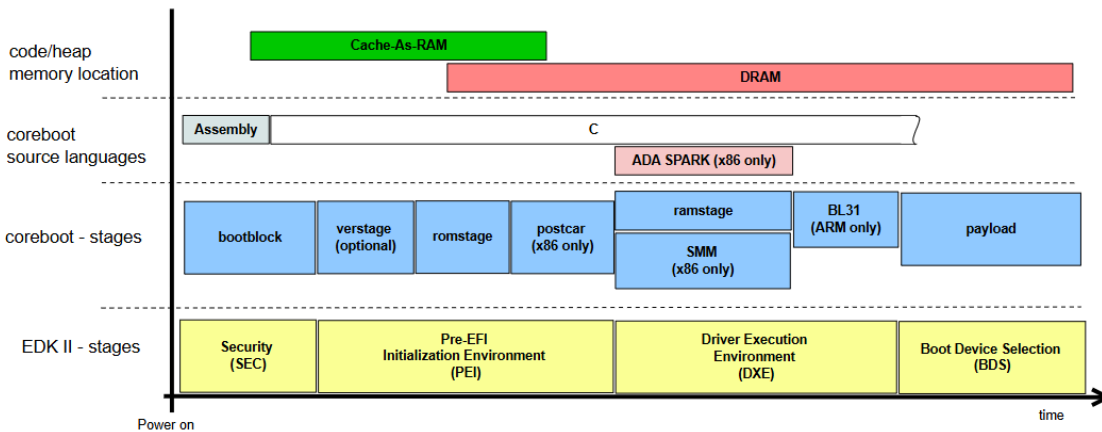
Automaticamente redirigimos la salida del puerto de debug de la VM a la terminal de trabajo.

## Recapitulando

En los parrafos anteriores fuimos desde una explicación teorica, pasando por un ejemplo real, el cual se describio a grandes razgos y se compilo, para probar en un ambiente virtual, aunque el código generaba un BIOS demasiado sencillo, sirve para intentar conectar la teoria y la practica, el siquiente tema trata sobre el uso de un proyecto libre mas completo y profesional para la creación de BIOS.

## INTRODUCCION A LAS HERRAMIENTAS DEL PROYECTO COREBOOT

El proyecto Coreboot busca generar código abierto para la inicializacion de sistemas de computo modernos, su filosofia es hacer una cosa y hacerla bien, por lo que solamente se encarga de la etpa basica de inicialización del sistema, para permitir la carga de un Sistema Operativo, se puede agregar como Payloads a la compilación del archivo Binario otro código de BIOS complementarios como SEABIOS o LibreBoot, tambien permite código propietario en formato de archivos BLOBS (archivos binarios crudos) de vendedores como Intel y AMD. La siguiente Figura muestra un diagrama de tiempos de los procesos de ejecución del software de base que inicializa un sistema de computo, podemos ver como se listan las etapas de CoreBoot con respecto a las de EDKII (UEFI) por ejemplo.



Las etapas de Coreboot se describen brevemente a continuación:

## Etapas

Coreboot esta armado con varias etapas, las cuales son compiladas en archivos Binarios separados y posteriormente "pegadas" usando el mecanismo de compresión personalizado CBFS (CoreBoot File System, el cual no es un sistema de archivos real). Excepto la etapa de BootBlock (escrita en Ensamblador).

## Bootblock

La primera etapa en ser ejecutada tras que la CPU sale de reset o sleep mode. Esta escrita en ensamblador y su principal función es preparar todo para poder "cargar" un ambiente para ejecutar C (CPU pasa a modo Real, se crea una Pila, etc).

Las tareas principales en esta etapa son las siguientes:

- Cache-As-RAM para el area de "heap" y la "stack" (pila).
- Ajustar el apuntador de Pila
- Limpiar la memoria para el BSS
- Descomprimir y cargar la siguiente etapa (Load the next estage)

En maquinas con arquitectura x86 se realiza adicional:

°Actualizacion del Microcode

°Iniciación del Timmer

°Cambiar del modo Real de 16-bit hacia el Modo Protegido 32-bit (o 64 segun el caso)

Bootblock cargara la etapa Verstage si esta habilitada (esta etapa es opcional) o pasara a la siguiente Roomstage.

## **Cache-As-Ram**

Cache como RAM, debido a que la memoria actual es bastante compleja de inicializar usando simplemente Ensamblador, se deja para una etapa posterior, por lo que en su defecto se utiliza una parte de la memoria Cache del Microprocesador para cargar el BIOS. La cual se activa usando la especificacion de cada vendedor de CPUs. Las siguientes etapas se ejecutan cuando la CAR (Cache as Ram) esta activa:

bootblock romstage verstage postcar

## **Verstage**

En esta etapa comienzan los mecanismos de seguridad, se ejecutan codigo de seguridad y se realizan las verificaciones correspondientes del BIOS y del Hardware, no puede ser modificado en campo (aqui es donde entra la TPM).

Los mecanismos de verificación de arranque seguro permiten confiar en actualizaciones de Firmware en campo combinadas con un sistema de recuperación "seguro".

## **Romstage**

La RAM es inicializada y se prepara todo para inicializar los dispositivos.

Tareas Comunes:

°Preinicializacion de Dispositivos

°Inicializacion de la DRAM

## **Postcar**

Para dejar la configuracion CAR y ejecutar código en la DRAM regular, esta etapa desactiva al CAR y carga el código en la RAM, la etapa es pequeña en comparación con las otras etapas.

## **Ramstage**

Comienza inicializacion principal de los dispositivos:

°Inicializacion de dispositivos PCI

°Inicalizacion de dispositivos "On-chip"

°Inicializacion de la TPM (sin no fue inicializada en la etapa Verstage)

°Inicializacion de Graphics es opcional.

°Inicializacion del CPU (Como ajustar el SMM)

Tras la inicializacion de las tablas de sistema correspondientes se le informa a la siguiente etapa Payload (la cual es opcional) o al Sistema Operativo. El inform incluye:

°ACPI tables (x86 specific)

°SMBIOS tables (x86 specific)

°coreboot tables

°devicetree updates (ARM specific)

°Tambien asegura (bloquea) el hardware and firmware :

- Protección contra escritura del boot media

- Bloque los registros relacionados (importantes)

- Asegura el modo SMM(x86 specific)

## Payload

Este código se ejecuta tras que Coreboot a terminado. Reside dentro del CBFS y no hay posibilidad de escogerlo durante el tiempo de ejecución.

Usualmente aqui se carga un programa BIOS mas completo como SEABIOS o LibreBoot, para complementar el proceso de BOOT y ofrecer mas servicios, en maquinas x86 es posible cargar la especificacion UEFI.

Para información sobre como configurar un ambiente de diseño de CoreBoot y el uso de las herramientas consulte los anexos correspondientes al final de este documento.

## CODIGO FUENTE DEL PROYECTO COREBOOT

Tras configurar nuestro ambiente de trabajo con el proyecto, se descargan varias carpetas y archivos, dentro de la carpeta *src* encontraremos todo el código fuente usado por las herramientas:

```
rodrigo@raspberrypi:~/coreboot/src $ pwd
/home/rodrigo/coreboot/src
rodrigo@raspberrypi:~/coreboot/src $ ls
acpi  commonlib  cpu  drivers  include  lib  northbridge  security  southbridge  vendorcode
arch  console  device  ec  Kconfig  mainboard  sbom  soc  superio
rodrigo@raspberrypi:~/coreboot/src $
```

Por ejemplo revisaremos los archivos fuente para la compilación de la ROM para la maquina virtual, úbicados en la direccion:

*/home/rodrigo/coreboot/src/cpu/qemu-x86*

```
rodrigo@raspberrypi:~/coreboot/src/cpu/qemu-x86 $ pwd
/home/rodrigo/coreboot/src/cpu/qemu-x86
rodrigo@raspberrypi:~/coreboot/src/cpu/qemu-x86 $ ls
bootblock.c  cache_as_ram_bootblock.S  Kconfig  Makefile.mk  qemu.c
rodrigo@raspberrypi:~/coreboot/src/cpu/qemu-x86 $
```

Los archivos con terminación *.c* contienen código fuente en lenguaje C, los archivos *.S* estaran

en Lenguaje ensamblador, los archivos con la terminación .mk contienen instrucciones para el traductor de scripts MAKE, estos script automatizan la compilación con gcc. podemos ver su contenido con ayuda del comando cat:

```
rodrigo@raspberrypi:~/coreboot/src/cpu/qemu-x86 $ cat Makefile.mk
## SPDX-License-Identifier: GPL-2.0-or-later

bootblock-y += cache_as_ram_bootblock.S
bootblock-y += bootblock.c

romstage-y += ../intel/car/romstage.c

ramstage-y += qemu.c
rodrigo@raspberrypi:~/coreboot/src/cpu/qemu-x86 $
```

De esta forma simplemente desplegaremos el contenido solamente para su lectura, si deseamos trabajar con los archivos o poder apreciarlos mejor, podemos usar nuestro editor de preferencia como en el caso de nuestra raspberry pi Geanny:

```
qemu.c x
1  /* SPDX-License-Identifier: GPL-2.0-or-later */
2
3  #include <cpu/cpu.h>
4  #include <device/device.h>
5
6  static void qemu_cpu_init(struct device *dev)
7  {
8  }
9
10 static struct device_operations cpu_dev_ops = {
11     .init = qemu_cpu_init,
12 };
13
14 static const struct cpu_device_id cpu_table[] = {
15     { X86_VENDOR_ANY, 0, 0 },
16     CPU_TABLE_END
17 };
18
19 static const struct cpu_driver driver __cpu_driver = {
20     .ops = &cpu_dev_ops,
21     .id_table = cpu_table,
22 };
23
24 struct chip_operations cpu_qemu_x86_ops = {
25     .name = "QEMU x86 CPU",
26 };
27
```

O ya que estamos en la terminal un editor de línea como Vim, Nano o Vi:

```
1  /* SPDX-License-Identifier: GPL-2.0-or-later */
2
3  #include <cpu/cpu.h>
4  #include <device/device.h>
5
6  static void qemu_cpu_init(struct device *dev)
7  {
8  }
9
10 static struct device_operations cpu_dev_ops = {
11     .init = qemu_cpu_init,
12 };
13
14 static const struct cpu_device_id cpu_table[] = {
15     { X86_VENDOR_ANY, 0, 0 },
16     CPU_TABLE_END
17 };
18
19 static const struct cpu_driver driver __cpu_driver = {
20     .ops = &cpu_dev_ops,
21     .id_table = cpu_table,
22 };
23
24 struct chip_operations cpu_qemu_x86_ops = {
25     .name = "QEMU x86 CPU",
26 }
```

Una exploración en las carpetas del código fuente nos dará una idea de cómo está "armado" todo el proyecto, por ejemplo la carpeta SRC en la que nos encontramos tiene el código de donde se toman las opciones de selección de ROM que hayamos elegido con la herramienta

MenuConfig de Coreboot (ver anexo correspondiente).

La idea es poder revisar nosotros mismos el código fuente y evitar la inclusion de software malicioso en nuestro BIOS, si deseamos tambien podemos compilar nosotros mismos los archivos manualmente como se mostro en el ejemplo del BIOS sencillo, si deseamos asegurar que las herramientas de compilación no inyectan código malicioso a nuestro BIOS. El proposito es tomar solamente los archivos necesarios para poder ejecutar la especificacion UEFI con la seguridad de que no hay código malicioso o "Back Doors" y reforzar la seguridad de nuestro sistema. Dicho sea de paso tener código de ejemplo y herramientas abiertas nos permite estudiar y aprender como se realiza el trabajo profesional de Software de Base.

## Bibliografía

Guillermo Levinne, Computación y programación moderna (una perspectiva integral de la informatica), editorial Addison Wesley.

Jenny M Peiner, James A Peiner, White Paper Minimal Intel Architecture Boot Loader, Intel Corporation.

<https://www.intel.com/content/www/us/en/intelligent-systems/intel-boot-loader-development-kit/minimal-intel-architecture-boot-loader-paper.html>

CoreBoot Wep page

<https://www.coreboot.org/>

<https://doc.coreboot.org/tutorial/part1.html>

Manual de Docker

<https://github.com/AngelSanchezT/books-1/blob/master/docker/the-docker-book.pdf>

Data sheet SuperIO chip PC97338:

<https://pdf.datasheetcatalog.com/datasheet/nationalsemiconductor/PC97338.pdf>

*Para información de las herramientas de compilación Libro:*

System Software an introduction to system programming by Leland L. Beck

Pagina web del manual oficial GNAT el enlazador (linker) del compilador GNU C

[https://gcc.gnu.org/onlinedocs/gcc-4.3.6/gnat\\_ugn\\_unw.pdf](https://gcc.gnu.org/onlinedocs/gcc-4.3.6/gnat_ugn_unw.pdf)

## Anexo A

### 1ra parte



# Código fuente BIOS Sencillo

Extraído del blog del autor:

<https://pete.akeo.ie/2011/06/crafting-bios-from-scratch.html>

```
1  /*****
2  /* VMware BIOS ROM example */
3  /* Copyright (c) 2011 Pete Batard (pete@akeo.ie) - Public Domain */
4  *****/
5
6
7  /*****
8  /* GNU Assembler Settings: */
9  *****/
10 /* intel_syntax noprefix /* Use Intel assembler syntax (same as IDA Pro) */
11 .code16 /* After reset, the x86 CPU is in real / 16 bit mode */
12 /*****
13
14
15 /*****
16 /* Macros: */
17 *****/
18 /* This macro allows stackless subroutine calls */
19 .macro ROM_CALL addr
20     mov sp, offset 1f /* Use a local label as we don't know the size */
21     jmp \addr /* of the jmp instruction (can be 2 or 3 bytes) */
22 1: /* see http://sourceware.org/binutils/docs-2.21/as/Symbol-Names.html */
23 .endm
24
25
26 /*****
27 /* Constants: */
28 *****/
29 /* The VMware platform uses an emulated NS PC97338 as SuperIO */
30 SUPERIO_BASE = 0x2e /* Do NOT believe what you see in the BIOS bootblock: */
31 /* the VMware SuperIO base is 0x2e and not 0x398. */
32 PC97338_FER = 0x00 /* PC97338 Function Enable Register */
33 PC97338_FAR = 0x01 /* PC97338 Function Address Register */
34 PC97338_PTR = 0x02 /* PC97338 Power and Test Register */
35
36 /* 16650 UART setup */
37 COM_BASE = 0x3f8 /* Our default COM1 base, after SuperIO init */
38 COM_RB = 0x00 /* Receive Buffer (R) */
39 COM_TB = 0x00 /* Transmit Buffer (W) */
40 COM_BRD_LO = 0x00 /* Baud Rate Divisor LSB (when bit 7 of LCR is set) */
41 COM_BRD_HI = 0x01 /* Baud Rate Divisor MSB (when bit 7 of LCR is set) */
42 COM_IER = 0x01 /* Interrupt Enable Register */
43 COM_FCR = 0x02 /* 16650 FIFO Control Register (W) */
44 COM_LCR = 0x03 /* Line Control Register */
45 COM_MCR = 0x04 /* Modem Control Register */
46 COM_LSR = 0x05 /* Line Status Register */
47 /*****
48
49
50 /*****
51 /* begin : Dummy section marking the very start of the BIOS. */
52 /* This allows the .rom binary to be filled to the right size with objcopy. */
53 *****/
54 .section begin, "a" /* The 'ALLOC' flag is needed for objcopy */
55     .ascii "VMBIOS v1.00" /* Dummy ID string */
56     .align 16
57 /*****/
```

```

60  /*****
61  /* main:
62  /* This section will be relocated according to the bios.ld script.
63  /*****
64  /* 'init' doesn't have to be at the beginning, so you can move it around, as
65  /* long as remains reachable, with a short jump, from the .reset section.
66  .section main, "ax"
67  .globl init /* init must be declared global for the linker and must
68  init: /* point to the first instruction of your code section
69  cli /* NOTE: This sample BIOS runs with interrupts disabled
70  cld /* String direction lookup: forward
71  mov ax, cs /* A real BIOS would keep a copy of ax, dx as well as
72  mov ds, ax /* initialize fs, gs and possibly a GDT for protected
73  mov ss, ax /* mode. We don't do any of this here.
74
75  init_superio:
76  mov dx, SUPERIO_BASE /* The PC97338 datasheet says we are supposed
77  in al, dx /* to read this port twice on startup, but the
78  in al, dx /* VMware virtual chip doesn't seem to care...
79
80  /* Feed the SuperIO configuration values from a data section
81  mov si, offset superio_conf /* Don't forget the 'offset' here!
82  mov cx, (serial_conf - superio_conf)/2
83  write_superio_conf:
84  mov ax, [si]
85  ROM_CALL superio_out
86  add si, 0x02
87  loop write_superio_conf
88
89  init_serial: /* Init serial port
90  mov si, offset serial_conf
91  mov cx, (hello_string - serial_conf)/2
92
93  write_serial_conf:
94  mov ax, [si]
95  ROM_CALL serial_out
96  add si, 0x02
97  loop write_serial_conf
98
99  print_hello: /* Print a string
100  mov si, offset hello_string
101  ROM_CALL print_string
102
103  serial_repeater: /* End the BIOS with a simple serial repeater
104  ROM_CALL readchar
105  ROM_CALL putchar
106  jmp serial_repeater
107
108  /*****
109  /* Subroutines:
110  /*****
111  superio_out: /* AL (IN): Register index, AH (IN): Data to write
112  mov dx, SUPERIO_BASE
113  out dx, al
114  inc dx
115  xchg al, ah
116  out dx, al
117  jmp sp

```

```

119 serial_out:      /* AL (IN): COM Register index, AH (IN): Data to Write */
120     mov dx, COM_BASE
121     add dl, al /* Unless something is wrong, we won't overflow to DH */
122     mov al, ah
123     out dx, al
124     jmp sp
125
126
127 putchar:        /* AL (IN): character to print */
128     mov dx, COM_BASE + COM_LSR
129     mov ah, al
130 tx_wait:
131     in al, dx
132     and al, 0x20 /* Check that transmit register is empty */
133     jz tx_wait
134     mov dx, COM_BASE + COM_TB
135     mov al, ah
136     out dx, al
137     jmp sp
138
139
140 readchar:       /* AL (OUT): character read from serial */
141     mov dx, COM_BASE + COM_LSR
142 rx_wait:
143     in al, dx
144     and al, 0x01
145     jz rx_wait
146     mov dx, COM_BASE + COM_RB
147     in al, dx
148     jmp sp
149
150
151 print_string:   /* SI (IN): offset to NUL terminated string */
152     lodsb
153     or al, al
154     jnz write_char
155     jmp sp
156 write_char:
157     shl esp, 0x10 /* We're calling a sub from a sub => preserve SP */
158     ROM_CALL putchar
159     shr esp, 0x10 /* Restore SP */
160     jmp print_string
161
162
163 /******
164 /* Data:
165 /******
166 superio_conf:
167 /* http://www.datasheetcatalog.org/datasheet/nationalsemiconductor/PC97338.pdf */
168     .byte PC97338_FER, 0x0f /* Enable COM, PAR and FDC */
169     .byte PC97338_FAR, 0x10 /* LPT=378, COM1=3F8, COM2=2F8 */
170     .byte PC97338_PTR, 0x00 /* Make sure COM1 test mode is cleared */
171 serial_conf: /* See http://www.versalogic.com/kb/KB.asp?KBID=1395 */
172     .byte COM_MCR, 0x00 /* RTS/DTS off, disable loopback */
173     .byte COM_FCR, 0x07 /* Enable & reset FIFOs. DMA mode 0. */
174     .byte COM_LCR, 0x80 /* Set DLAB (access baudrate registers) */
175     .byte COM_BRD_LO, 0x01 /* Baud Rate 115200 = 0x0001 */
176     .byte COM_BRD_HI, 0x00
177     .byte COM_LCR, 0x03 /* Unset DLAB. Set 8N1 mode */
178 hello_string:
179     .string "\r\nHello BIOS world!\r\n" /* .string adds a NUL terminator */
180 /******
181
182 /******
183 /* reset: this section must reside at 0xffffffff, and be exactly 16 bytes */
184 /******
185
186 .section reset, "ax"
187 /* Issue a manual jmp to work around a binutils bug.
188 /* See coreboot's src/cpu/x86/16bit/reset16.inc
189     .byte 0xe9
190     .int init - ( . + 2 )
191     .align 16, 0xff /* fills section to end of ROM (with 0xFF)
192 /******

```

## Anexo A

## 2da parte

### Código para el Enlazador del Bios Sencillo

Extraído del blog del autor:

<https://pete.akeo.ie/2011/06/crafting-bios-from-scratch.html>

```
1  OUTPUT_ARCH(i8086)          /* i386 for 32 bit, i8086 for 16 bit */
2
3  /* Set the variable below to the address you want the "main" section, from bios.S, */
4  /* to be located. The BIOS should be located at the area just below 4GB (4096 MB). */
5  main_address = 4096M - 4K;    /* Use the last 4K block */
6
7  /* Set the BIOS size below (both locations) according to your target flash size */
8  MEMORY {
9      ROM (rx) : org = 4096M - 512K, len = 512K
10 }
11
12 /* You shouldn't have to modify anything below this */
13 SECTIONS {
14     ENTRY(init)              /* To avoid antivirus false positives */
15     /* Sanity check on the init entrypoint */
16     _assert = ASSERT(init >= 4096M - 64K,
17         "'init' entrypoint too low - it needs to reside in the last 64K.");
18     .begin : { /* NB: ld section labels MUST be 6 letters or less */
19         *(begin)
20     } >ROM /* Places this first section at the beginning of the ROM */
21     /* the --gap-fill option of objcopy will be used to fill the gap to .main */
22     .main main_address : {
23         *(main)
24     }
25     .reset 4096M - 0x10 : { /* First instruction executed after reset */
26         *(reset)
27     }
28     .igot 0 : { /* Required on Linux */
29         *(.igot.plt)
30     }
31 }
32
```

## Anexo B

### Configuracion de un ambiente de Desarrollo para el proyecto Coreboot.

Este apartado describe los pasos requeridos para la instalacion de un ambiente de desarrollo de un BIOS usando Coreboot, las instrucciones originales se encuentran en la direccion:

<https://doc.coreboot.org/tutorial/part1.html>

Debido a que el sistema en el cual se "montara" la instalación es una raspberry pi modelo 3b+, sera necesario realizar algunos ajustes a las indicaciones originales,.

Actualice el sistema primero:

```
rodrigo@raspberrypi:~ $ sudo apt-get update
```

```
rodrigo@raspberrypi:~ $ sudo apt-get upgrade
```

Y posteriormente descargue el conjunto de herramientas necesarias para la configuración:

```
rodrigo@raspberrypi:~ $ sudo apt-get install -y bison build-essential curl flex git gnat libncurses5-dev libssl-dev m4 zlib1g-dev pkg-config
```

```
sudo apt-get install -y bison build-essential curl flex git gnat libncurses5-dev \
libssl-dev m4 zlib1g-dev pkg-config
```

La linea anterior descarga las herramientas que se requeriran durante las etapas del proceso de compilación del Bios. Lo siguiente es clonar el arbol de recursos:

```
git clone --recurse -submodules https://review.coreboot.org/coreboot.git
```

```
~ $ sudo git clone --recurse -submodules https://review.coreboot.org/coreboot.git
```

Acorde a las instrucciones de la pagina web del proyecto, nos pide que compilemos la "tool chain", los cuales son el grupo de complidores enlazadores y cargadores, para compilar el BIOS:

1) hay que ingresar a la carpeta de coreboot creada anteriormente:

```
cd coreboot
```

2) usar el comando make junto con el parametro crossgcc, elegiremos la arquitectra x86:

```
sudo make crossgcc-i386
```

**Este proceso es largo y propenso a errores, por lo que podrian requerirse varios intentos.**

Para evitar este inconveniente es mejor usar un contendor Docker con las herramientas ya compiladas de antemano, para eso primero debemos instalar el software en nuestra raspberry pi:

```
curl -sSL https://get.docker.com | sh
```

```
~ $ curl -sSL https://get.docker.com | sh
```

Terminada la instalación debemos otorgar permisos a nuestro usuario para que pueda ingresar a la ejecución del contenedor.

```
sudo usermod -aG docker <usuario>
```

```
rodrigo@raspberrypi:~ $ sudo usermod -aG docker rodrigo
```

Nota: en caso de que necesite desinstalar Docker realice lo siguiente:

-Liste los archivos instalados: `dpkg -l | grep -i docker`



-Elimine los archivos agregandolos en linea tras el siguiente comando: `sudo apt-get purge -y docker-buildx-plugin docker-ce ....`

-Por ultimo eliminelos directorios que quedan tras la instalación:

```
sudo rm -rf /var/lib/docker/etc/docker
```

```
sudo rm -rf /etc/apparmor.d/docker
```

```
sudo groupdel docker
```

```
sudo rm -rf /var/run/docker.sock
```

```
sudo rm -rf /var/lib/container
```

Si la instalación de Docker ocurrio sin problema, debemos descargar el contenedor con las herramientas ya compiladas:

```
sudo docker pull coreboot/coreboot-sdk
```

```
~ $ sudo docker pull coreboot/coreboot-sdk
```

Este contenedor es descargado del repositorio:

<https://hub.docker.com/r/coreboot/coreboot-sdk>

Este contenedor solo puede ejecutarse en un ambiente x86, debido a que nuestra Raspberry Pi usa ARM, necesitamos descargar un contenedor adicional:

```
docker pull tonistiigi/binfmt
```

```
~ $ sudo docker pull tonistiigi/binfmt
```

<https://hub.docker.com/r/tonistiigi/binfmt>

tras la descarga debemos ejecutar este contendor **Prior** a la ejecucion del contendor de Coreboot, **nota** asegure de contar con el software de virtualizacion Qemu

```
sudo apt-get install qemu
```

```
docker run --privileged --rm tonstigi/binfmt --install all
```

```
~/coreboot $ sudo docker run --privileged --rm tonistiigi/binfmt --install all
```

Ahora podemos ejecutar el contenedor con las herramientas de compilación del proyecto Coreboot:

```
cd coreboot #asegure de estar siempre en la carpeta del proyecto la cual clonamos #de git
```

```
sudo docker run -w /home/coreboot/coreboot -u root -it -v $PWD:/home/coreboot/coreboot --rm --platform linux/amd64 coreboot/coreboot-sdk /bin/bash
```

```

rodrigo@raspberrypi:~/coreboot $ sudo docker run -w /home/coreboot/coreboot -u root -it -v $PWD:/home/coreboot/coreboot/ --rm --platform linux/amd64 coreboot/coreboot-sdk /bin/bash
root@13d893afc647:/home/coreboot/coreboot#

```

Observe como en (A) el prompt ha cambiado, para terminar la ejecucion del contenedor y salir, teclee "exit".

El proceso de compilacion y creacion de nuestro BOS, se describe mas arriba en el documento

## Anexo C

### Compilando Coreboot BIOS para el software de virtualizacion QEMU.

Ejemplo practico del uso de las herramientas del ambiente de diseño de Coreboot, se compilará un archivo ROOM binario para inicializar el software de virtualización de Qemu, primero debemos ingresar a nuestro sesion Coreboot en el contenedor de Docker. Ejecutamos el comando *make clean*:

```

root@13d893afc647:/home/coreboot/coreboot# make clean
root@13d893afc647:/home/coreboot/coreboot#

```

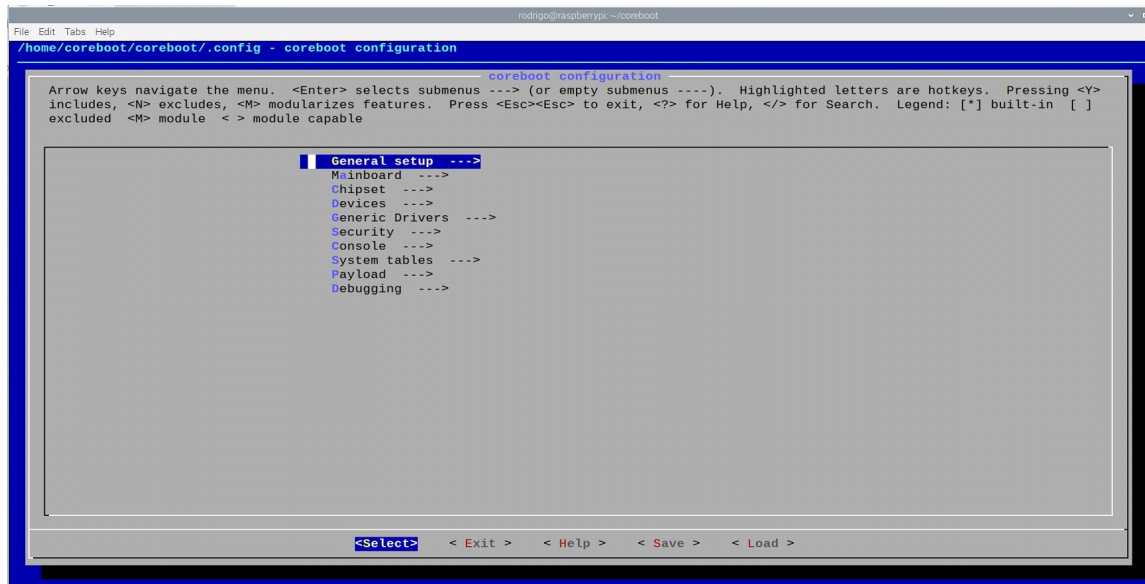
De esta forma nos aseguramos iniciar con una configuracion en blanco, posteriormente haremos uso de la herramienta del menu de configuración con el comando *make menuconfig*:

```

root@13d893afc647:/home/coreboot/coreboot# make clean
root@13d893afc647:/home/coreboot/coreboot# make menuconfig

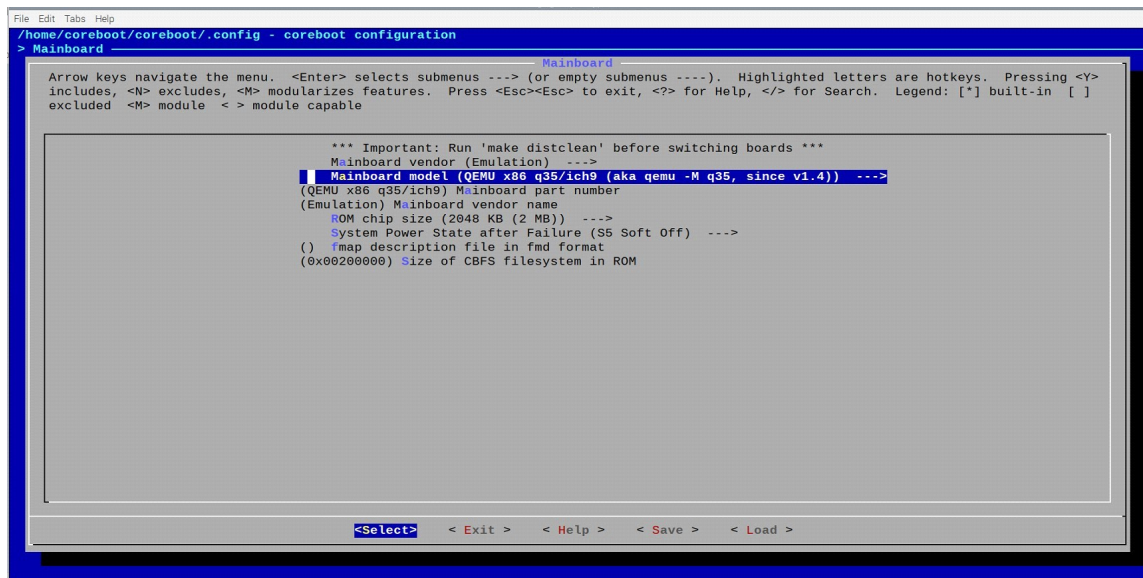
```

Nos presentara en pantalla un menu visual:

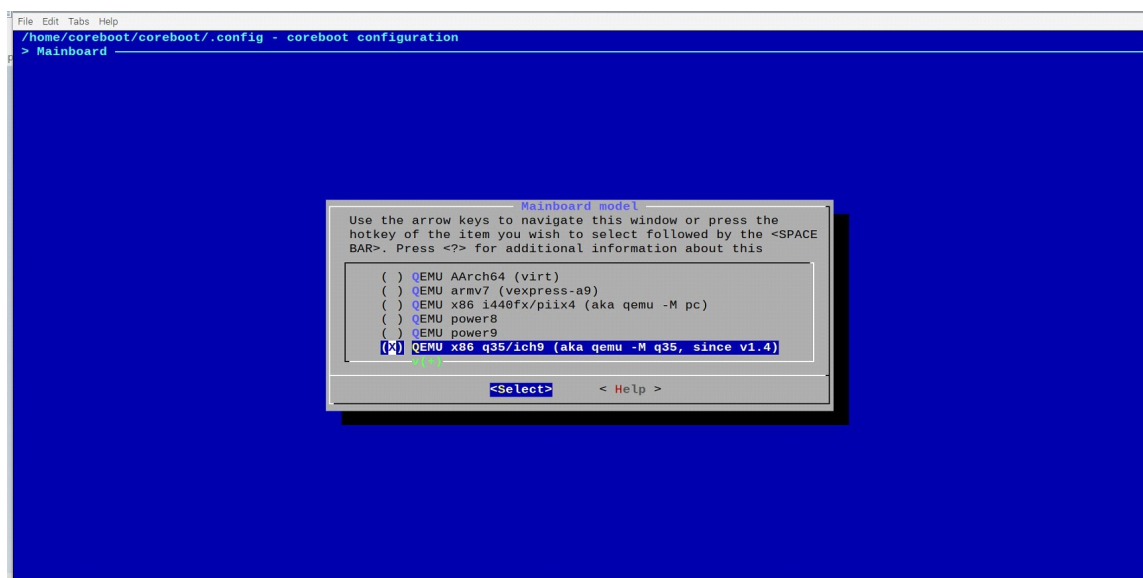


Elegiremos la opcion "Main Board":

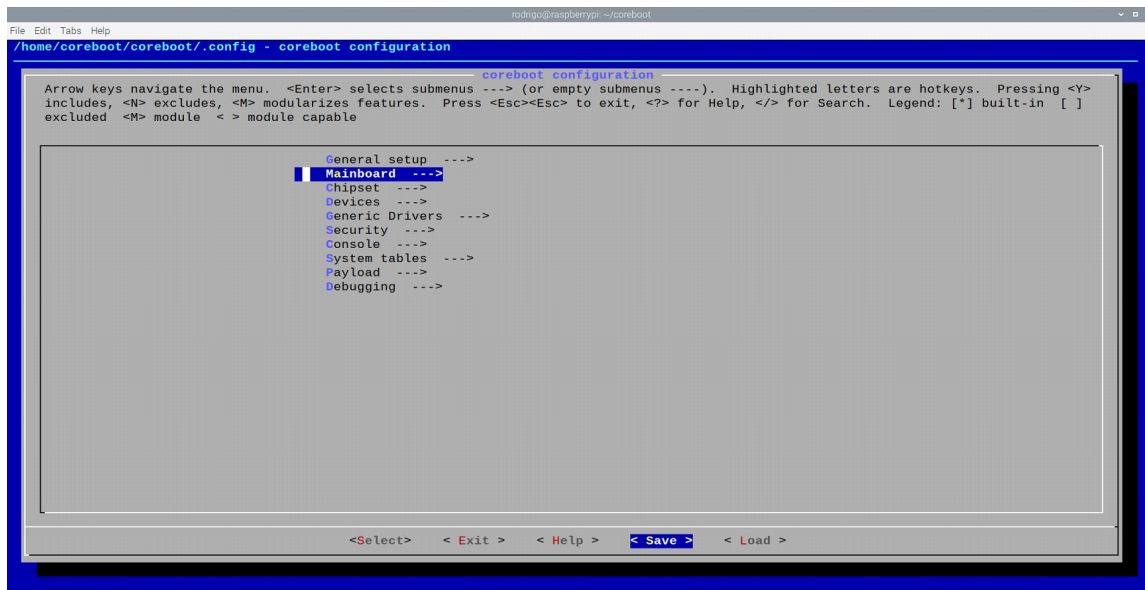




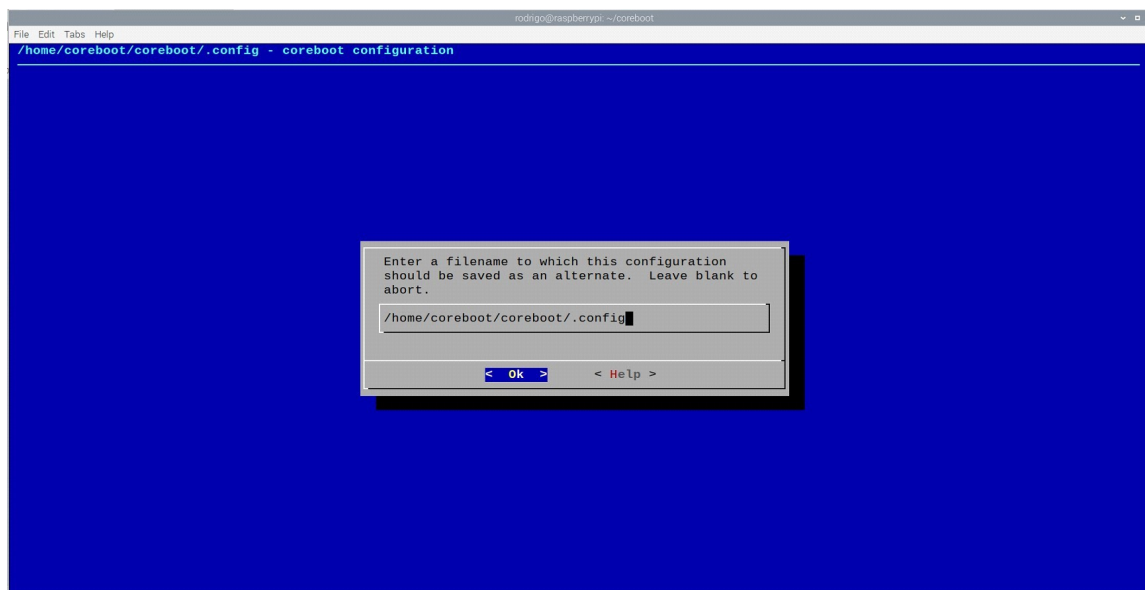
De ahí la opción *MainBoard mode (Qemu..* ", la cual nos despliega el siguiente listado:



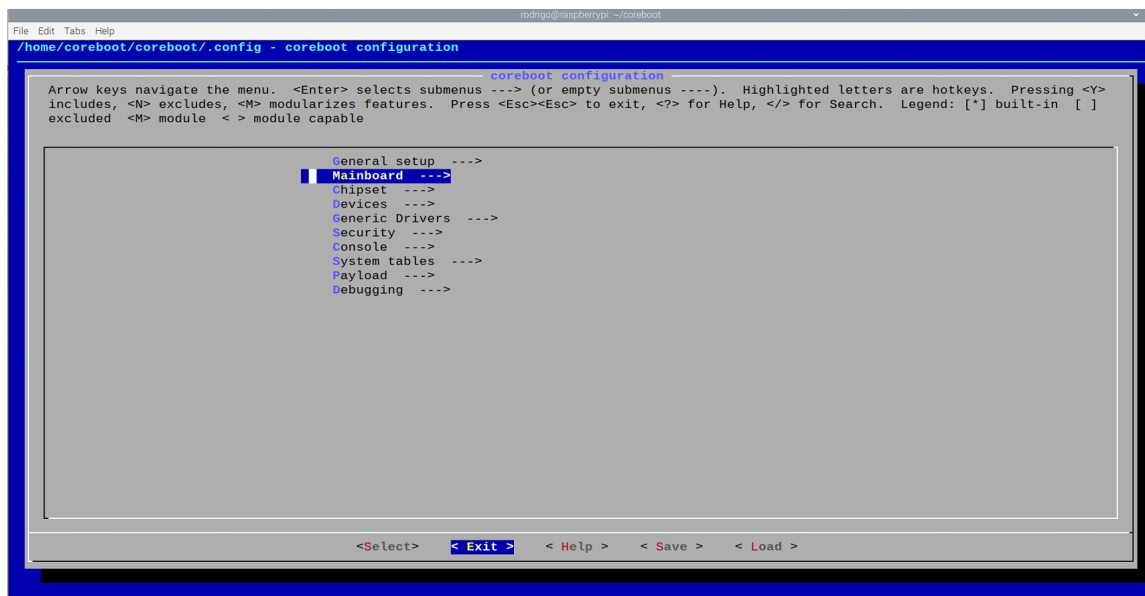
Marcamos la opción "QEMU x86 q35/ich9 ...", la cual corresponde al modelo de configuración de ROOM del software de la máquina Virtual.



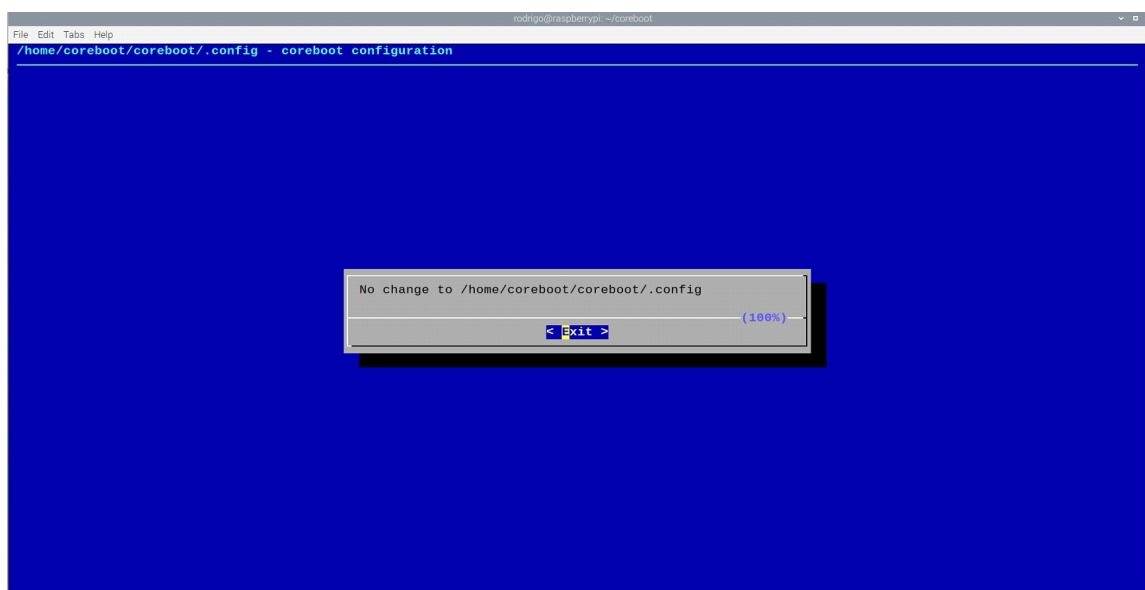
Tras seleccionarlo regresaremos al menú anterior y escogeremos la opción "<Save>", de la parte inferior del listado.



Nos mostrara la opción del nombre del archivo y de la dirección donde se guardara el archivo con esta configuración, lo dejamos tal como esta.



Terminamos la configuracion eligiendo la opcion "<Exit>" .



Tras elegir la opcion de "<Exit>" nos indicara si se realizaron cambios en el archivo, en este caso estamos usando las opciones por defecto así que puede que no veamos cambio alguno.

```
root@13d893afc647:/home/coreboot/coreboot# make clean
root@13d893afc647:/home/coreboot/coreboot# make menuconfig
root@13d893afc647:/home/coreboot/coreboot# make
HOSTCC      util/sconfig/lex.yy.o
```

Regresaremos al prompt del Terminal, donde usaremos el comando "*make*" , el cual iniciara el proceso de compilación acorde al archivo de configuración que construimos con la herramienta Menu (guardado en `:/home/coreboot/coreboot/.config`). podemos ver que iniciara el proceso de compilacion, el cual continuara durante un buen tiempo dependiendo de las capacidades de computo del sistema que se haya elegido para ejecutar la tarea (en este caso una Raspberry Pi

3, tomara una buena cantidad considerable de tiempo). Al finalizar el sistema creara un archivo ROM que podemos usar para iniciar nuestra maquina virtual.

```
HOSTCC      cbfstool/subpart_hdr_2.o
HOSTCC      cbfstool/subpart_entry_1.o
HOSTCC      cbfstool/cse_serger (link)
FMAP REGION: COREBOOT
Name      Offset      Type      Size  Comp
cbfs_master_header  0x0      cbfs header  32  none
fallback/ramstage  0x80      stage      17536 none
fallback/ramstage  0x4580    stage      71749 LZMA (149844 decompressed)
config      0x15e40    raw        2392 LZMA (7426 decompressed)
revision    0x16800    raw        709  none
build_info  0x16b00    raw        104  none
fallback/dsdt.aml  0x16bc0    raw        7175 none
cnos_layout.bin  0x18800    cnos_layout  640  none
fallback/postcar  0x18ac0    stage      21536 none
fallback/payload  0x1df40    simple elf  72645 none
payload_config  0x2fb40    raw        1621 none
payload_revision  0x301c0    raw        237  none
(empty)      0x30300    null       1886884 none
bootblock    0x1fcd0    bootblock   12288 none

Built emulation/qemu-q35 (QEMU x86 q35/ich9)
root@c46c68f0282d:/home/coreboot/coreboot# ls
```

La imagen anterior muestra el final de la compilacion, la cual se creo en el directorio `/build`, donde encontraremos un archivo denominado `coreboot.rom`:

```
root@c46c68f0282d:/home/coreboot/coreboot/build# ls
auto.conf      cb-config.rustcfg  config.h      cse_serger    dsdt.dsl      generated      postcar      rmodules_ppc64  smm          util
auto.conf.cmd  cbfs               coreboot.pre  decompressor  fnap.desc     ifuitool       ramstage     rmodules_riscv  smstub       verstage
bootblock      cbfstool           coreboot.rom  dsdt.aml     fnap.fmap     libgnat-x86_32  rmodtool     rmodules_x86_32  static.h     xcompile
build.h        cnos_layout.bin   cpu           dsdt.asl      fnap.fnd      mainboard      rmodules_arm  rmodules_x86_64  static_devices.h
build_info     config            cse_fpt       dsdt.d        fnap_config.h  option_table.h  rmodules_arm64  romstage        static_fu_config.h
root@c46c68f0282d:/home/coreboot/coreboot/build#
```

El cual usaremos para inicializar nuestra maquina virtual, con el siguiente comando:

`qemu-system-x86_64 -M q35 -bios ./build/coreboot.rom -serial stdio -display none`

```
root@c46c68f0282d:/home/coreboot/coreboot# qemu-system-x86_64 -M q35 -bios ./build/coreboot.rom -serial stdio -display none

[NOTE] coreboot-coreboot-unknown Sat Jun 08 01:51:33 UTC 2024 x86_32 bootblock starting (log level: 7)...
[DEBUG] FMAP: Found "FLASH" version 1.1 at 0x0.
[DEBUG] FMAP: base = 0xffe00000 size = 0x200000 #areas = 3
[DEBUG] FMAP: area COREBOOT found @ 200 (2096640 bytes)
[INFO] CBFs: mcache @0x00014e00 built for 13 files, used 0x2d4 of 0x4000 bytes
[INFO] CBFs: Found 'fallback/ramstage' @0x80 size 0x4480 in mcache @0x00014e2c
[DEBUG] BS: bootblock times (exec / console): total (unknown) / 117 ms

[NOTE] coreboot-coreboot-unknown Sat Jun 08 01:51:33 UTC 2024 x86_32 ramstage starting (log level: 7)...
[DEBUG] SMBus controller enabled
[INFO] QEMU: firmware config interface detected
[INFO] Firmware config version id: 3
[INFO] QEMU: firmware config: Found 'etc/s820'
[DEBUG] CBMEM:
[DEBUG] IMD: root @ 0x07ffff00 254 entries.
[DEBUG] IMD: root @ 0x07ffac00 62 entries.
[DEBUG] FMAP: area COREBOOT found @ 200 (2096640 bytes)
```

Esto iniciara la ejecucion de la Maquina Virtual con nuestro Bios.

**Nota** Este BIOS es para uso exclusivo de la maquina virtual con QEMU, no intente usarlo en un Hardware Fisico.

