UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

Sviluppo di un BIOS didattico per QEMU

Transizione in Protected Mode, configurazione della modalità testo 80x25 e inizializzazione del bus PCI

Relatore:

Prof. Giuseppe Lettieri

Prof. Leonardo Giovannoni

Candidato:

Michele Castrucci

ANNO ACCADEMICO 2024/2025

Indice

Elenco delle figure

Introduzione

In un periodo storico nel quale l'informatica si distanzia sempre più dal calcolatore fisico e la tendenza è quella di astrarre costantemente verso il linguaggio umano, al punto che alcune delle figure più eminenti affermano che la programmazione del futuro avverrà interamente in linguaggio naturale, paradossalmente è sempre più importante conoscere l'hardware. Avere delle basi solide permette di attuare migliori scelte progettuali nel processo di astrazione e, di conseguenza, garantire ottimizzazione e stabilità anche nella programmazione di alto livello. Per questo, sviluppare progetti come un sistema operativo didattico permette di entrare in contatto in maniera diretta e personale con le problematiche tipiche dello sviluppo di sistemi operativi e con le relative tecniche di risoluzione.

Il progetto descritto in questa relazione si propone di ampliare un sistema didattico esistente sviluppando un BIOS personalizzato che vada a sostituire SeaBIOS, il firmware standard dell'emulatore QEMU. Tale lavoro ci permetterà di analizzare a fondo le modalità operative del processore x86, esaminando nel dettaglio il passaggio dalla *Real Mode* alla *Protected Mode*. Approfondiremo inoltre l'interazione con l'hardware video, inizializzando la modalità testo 80x25 attraverso la configurazione dei registri e il caricamento dei glifi nella VGA. Assisteremo in maniera pratica alla fase di *Discovery* dei dispositivi connessi al bus PCI e alla fase di *Probing*, con la conseguente assegnazione degli indirizzi di memoria e I/O e la relativa gestione delle interruzioni. Infine, per realizzare tutto ciò in un ambiente *Freestanding*, svilupperemo da zero una libreria essenziale di funzioni di utilità.

Struttura modulare del codice e gestione del flusso di controllo

Il firmware è stato concepito con un'architettura rigorosamente modulare, strutturata in base alle fasi sequenziali di inizializzazione dell'hardware. Al termine della transizione in *Protected Mode* gestita dal file `entry.s`, il flusso di esecuzione viene trasferito alla funzione `main`, che agisce da orchestratore principale dell'intero processo di avvio.

Tale funzione si occupa di invocare le routine di inizializzazione (`init`) dei singoli componenti, ciascuna implementata e confinata nel proprio modulo di competenza. Dal punto di vista architetturale, ogni modulo fa affidamento su primitive di I/O

che estendono quelle definite nel modulo centrale di supporto. Questo modulo rappresenta l'unica entità condivisa trasversalmente da tutto il codice C: al suo interno sono definite le funzioni di utilità fondamentali, necessarie per sopperire alla totale assenza della libreria standard dovuta all'ambiente *Freestanding*.

Per garantire una narrazione coerente tra il design del software e la sua documentazione, anche la stesura dei capitoli del presente elaborato ricalca fedelmente questa medesima struttura modulare, analizzando ciascun componente in ordine cronologico di inizializzazione.

Capitolo 1

Dal Reset Vector alla Protected Mode

Al termine della sequenza di RESET il processore si trova in *Real Mode* con le interruzioni disabilitate. Il registro CS (selettore visibile) viene inizializzato al valore `0xF000`, mentre l'Instruction Pointer (IP) viene impostato a `0x0000FFF0`.

Sebbene in questa modalità a 16 bit la Base del segmento (CS.Base) sia normalmente calcolata seguendo la regola $CS \times 16$, lo stato di RESET costituisce un'eccezione hardware: i valori della parte nascosta del registro, o *shadow register* (CS.Base e CS.Limit), vengono forzati rispettivamente a `0xFFFF0000` e `0xFFFF`, ignorando il valore del selettore. Di conseguenza, l'indirizzo fisico della prima istruzione viene ottenuto sommando la Base forzata all'IP:

$$\texttt{0xFFFF0000} + \texttt{0x0000FFF0} = \texttt{0xFFFFFFF0}$$

Tale indirizzo è noto come *Reset Vector*. Questa configurazione garantisce che il processore acceda immediatamente all'area di memoria riservata alla ROM di sistema, la quale risiede nell'estremità superiore della memoria installata.

Per una disamina completa dei valori assunti da tutti i registri del processore nello stato di RESET, si rimanda alla documentazione ufficiale Intel, nello specifico alla sezione *Processor State After Reset* (Volume 3A) [5].

A causa dei valori assunti dalla parte nascosta del registro CS, in questa fase possiamo indirizzare solamente gli ultimi 64 KB di memoria; tale spazio di indirizzamento risulta insufficiente: non solo per la lettura integrale del codice contenuto nella ROM (che nel nostro caso è di 128 KB), ma soprattutto per l'inizializzazione dell'hardware. Per configurare correttamente la piattaforma è necessario accedere all'intero spazio di indirizzamento a 32 bit (4 GB), ad esempio per mappare le periferiche PCI o gestire le aree di memoria legacy (come il buffer VGA a `0xB8000`). Pertanto, è necessario modificare le impostazioni di segmentazione e commutare il processore dalla *Real Mode* alla *Protected Mode*, in cui abbiamo un indirizzamento a 32 bit.

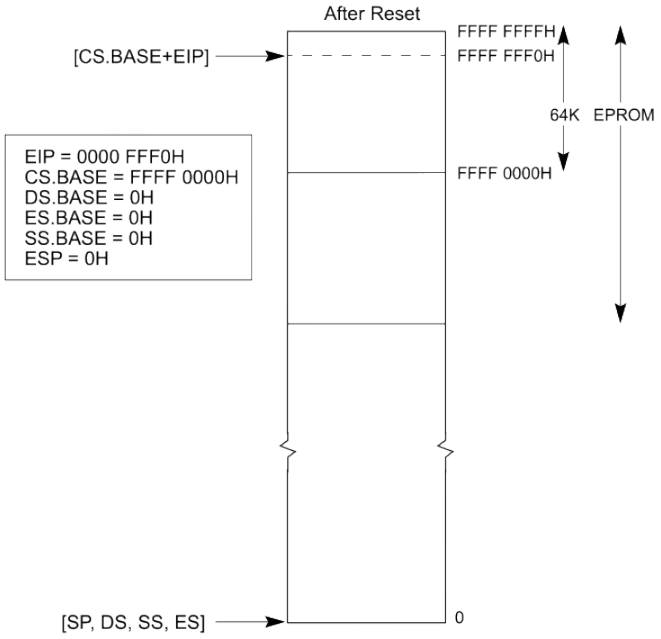

Figura 1.1: Stato della segmentazione dopo il RESET (Fonte: [5])

1.1 Configurazione della GDT e realizzazione del Flat Memory Model

Prima di commutare il processore in Protected Mode, è necessario definire la Global Descriptor Table (GDT). In questa modalità, infatti, il registro CS assume la funzione di selettore per i descrittori di segmento (Segment Descriptors), attraverso i quali vengono specificati i valori della parte nascosta del registro stesso (Base, Limit, Access Rights). Poiché l'obiettivo del BIOS è accedere liberamente all'intera memoria disponibile, la soluzione ideale prevederebbe di operare direttamente con indirizzi fisici lineari, bypassando il meccanismo di segmentazione. Tuttavia, l'architettura x86 impone l'uso della segmentazione quando si opera in Protected Mode, contrariamente alla paginazione che rimane un meccanismo opzionale.

Per nascondere il meccanismo di segmentazione imposto dall'architettura è possibile applicare il *Flat Memory Model*. Questo modello permette di mappare l'intero spazio di indirizzamento in maniera continua: configurando i segmenti con Base 0 e Limite massimo (4 GB, ottenuto abilitando la granularità a 4 KB), si fa in modo che l'indirizzo logico coincida con l'indirizzo lineare, simulando di fatto un accesso alla memoria non segmentato.

Per implementare il modello di memoria flat di base, devono essere creati almeno due descrittori di segmento: uno per il codice (Code Segment) e uno per i dati (Data Segment). Sebbene entrambi mappino lo stesso spazio di indirizzamento lineare (segmenti sovrapposti), essi differiscono nei permessi di accesso: Esecuzione/Lettura per il primo e Lettura/Scrittura per il secondo.

La struttura standard di un descrittore di segmento è la seguente:

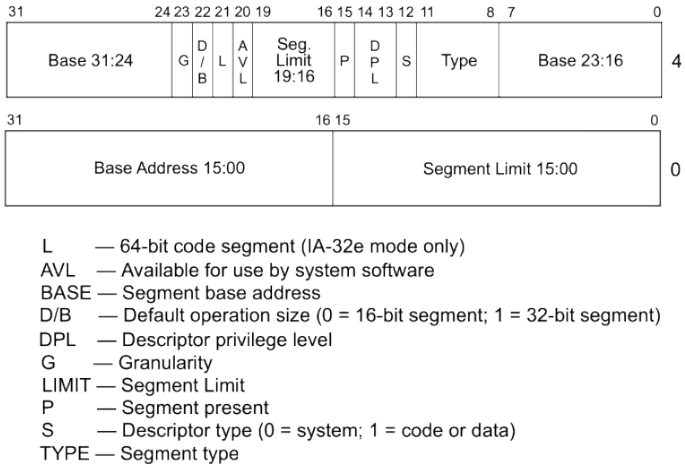

Figura 1.2: Descrittore di segmento (Fonte: [5])

Una volta creati i descrittori per i segmenti codice e dati, per completare la GDT è sufficiente inserire in testa un descrittore nullo (*Null Descriptor*), preceduto da un'etichetta che useremo per indirizzare la tabella. Per informare il processore della presenza della nuova tabella, è necessario creare il descrittore della GDT (*GDT Descriptor*). Questa struttura è composta da 48 bit: i primi 16 bit definiscono il *Limit* della tabella (la sua dimensione in byte meno uno), mentre i successivi 32 bit contengono l'indirizzo base lineare in cui la GDT è allocata in memoria.

Di seguito viene riportata l'implementazione della GDT presente nel file `entry.s`. È possibile osservare la definizione del *Null Descriptor*, seguito dai descrittori per il codice e per i dati (configurati secondo il *Flat Memory Model*) e infine la struttura del descrittore della tabella (`temp_gdt_descriptor`) utilizzato dall'istruzione di caricamento.

```
4  # ================ GDT ================
5
6  temp_gdt:
7    .quad 0 # NULL first descriptor
8
9  # Code segment descriptor (Index 1, Selector 0x08)
10 code_seg:
11   .word 0xFFFF                    # Limit (0-15)
12   .word 0x0000                    # Base (0-15)
13   .byte 0x00                      # Base (16-23)
14   .byte 0b10011010                # Access Byte: Present, Ring 0,
     ↪  Code, Executable, Readable
15   .byte 0b11001111                # Flags (4K granularity, 32-bit) +
     ↪  Limit (16-19)
16   .byte 0x00                      # Base (24-31)
17
18 # Data segment descriptor (Index 2, Selector 0x10)
19 data_seg:
20   .word 0xFFFF
21   .word 0x0000
22   .byte 0x00
23   .byte 0b10010010                # Access Byte: Present, Ring 0,
     ↪  Data, Writable
24   .byte 0b11001111
25   .byte 0x00
26 temp_gdt_end:
27
28 # ================ GDT DESCRIPTOR ================
29
30 temp_gdt_descriptor:
31   .word temp_gdt_end - temp_gdt - 1   # limit
32   .long temp_gdt                      # base
33
```

Listing 1: Definizione della GDT e del GDT Descriptor in entry.s

1.2 Ingresso in Protected Mode

```
40 # ================ ENTRY POINT 16 BIT ================
41
42 .section .text16
43 .code16
44
45 START:
46   cli            # disable interrupt
47   movw %cs, %ax
48   movw %ax, %ds # setup data segment
49
50   # Load IDT and GDT
51
52   lidtl (idt_descriptor)
53   lgdtl (temp_gdt_descriptor)
54
55   # enter protected code
56   movl %cr0, %ebx
57   orl $0x11, %ebx
58   movl %ebx, %cr0
59
60   # far jump to clear prefetch queue
61   ljmpl $0x08,$protected_mode
62
⋮
86
87 entry:
88   jmp START
```

Listing 2: Codice eseguito in *Real Mode* (Estratto da entry.s)

L'esecuzione del firmware ha inizio dall'etichetta `entry`. Come verrà approfondito nel capitolo dedicato al Linker Script, questa sezione viene posizionata esattamente in corrispondenza del *Reset Vector*, ovvero a soli 16 byte dalla fine dello spazio di indirizzamento fisico (`0xFFFFFFF0`). È evidente che tale spazio risulta insufficiente per contenere la logica di inizializzazione del sistema; di conseguenza, l'unica operazione possibile è eseguire un salto incondizionato (`jmp`) verso una locazione di memoria inferiore, dove è allocato il codice di ingresso alla *Protected Mode* (rappresentato dall'etichetta `START`).

La prima operazione da eseguire, dopo il salto, è la disattivazione delle interruzioni. Sebbene dopo un RESET questa azione possa sembrare ridondante (dato

che il flag IF dovrebbe essere già a 0), non avendo la certezza assoluta sul tipo di riavvio avvenuto, è più corretto eseguire esplicitamente l'istruzione `cli`. Successivamente, attraverso le istruzioni `lidtl` e `lgdtl`, forniamo al processore i descrittori della Tabella delle Interruzioni (IDT) e della GDT: per la prima utilizziamo un descrittore nullo (interamente a zero), mentre per la seconda passiamo la struttura definita in precedenza. Va precisato che queste operazioni non aggiornano ancora il registro CS, che cambierà solo con il successivo *Far Jump*. Il suffisso *-l*, aggiunto ai comandi `lidt` e `lgdt`, è necessario perché istruisce l'assemblatore ad aggiungere il byte `0x0x66` (prefisso di override) prima dell'opcode dell'istruzione. In questo modo, quando il processore decodifica l'istruzione in fase di fetch, sa di dover utilizzare momentaneamente operandi a 32 bit. Questo ci permette di caricare correttamente gli indirizzi lineari delle tabelle, anche se stiamo eseguendo il codice in un ambiente a 16 bit.

Il passaggio effettivo alla modalità protetta avviene modificando il registro di controllo `CR0`. Nello specifico, l'istruzione `orl $0x11, %cr0` imposta simultaneamente due flag. Il più importante è il bit PE (*Protection Enable*, bit 0) che, una volta asserito, attiva ufficialmente la Protected Mode. Il secondo è il bit ET (*Extension Type*, bit 4), abilitato per ragioni di retro-compatibilità con l'architettura i386. Storicamente, questo flag serviva a forzare l'uso del protocollo di comunicazione a 32 bit con il coprocessore matematico esterno (interfaccia 80387), sostituendo il vecchio protocollo a 16 bit. Sebbene dai processori 486 in poi la FPU sia stata integrata nella CPU e il bit ET sia fissato a 1 a livello hardware (comportamento replicato anche da QEMU), lo impostiamo comunque esplicitamente per rigore formale. Analizzando il codice presente in `entry.s`, si nota tuttavia che l'operazione viene eseguita sul registro `%ebx`. Questo passaggio intermedio è obbligatorio poiché l'architettura non permette di eseguire operazioni aritmetiche o logiche direttamente sui registri di controllo; l'unica modalità di accesso consentita per interagire con `%cr0` è infatti il trasferimento dati tramite l'istruzione `mov`.

L'ultima operazione necessaria per completare il cambio di modalità senza errori è l'esecuzione di un *far jump*. Questo passaggio è fondamentale per due motivi. In primo luogo, serve a rendere effettive le nuove impostazioni di segmentazione: il salto aggiorna il registro CS con il nuovo selettore, caricando così nella parte nascosta del registro i valori definiti nella GDT. In secondo luogo, un salto lontano costringe il processore a svuotare la *Prefetch Queue*. Senza questa operazione, la CPU rischierebbe di eseguire le istruzioni successive decodificandole in modo errato (come se fossero ancora a 16 bit) dato che si trovano già caricate nella pipeline. Infine, anche l'istruzione `ljmp` necessita del suffisso *-l* per la stessa ragione vista in precedenza: esso garantisce che il processore prelevi correttamente l'indirizzo a 32 bit completo, anziché limitarsi a leggere un offset di soli 16 bit.

1.3 Predisposizione dell'ambiente a 32-bit: registri dei segmenti e Stack

Sebbene il passaggio in *Protected Mode* sia ufficialmente avvenuto, rimangono ancora alcune operazioni preliminari per rendere il sistema stabile e pronto all'esecuzione di codice C a 32 bit. Attualmente ci troviamo in uno stato di segmentazione inconsistente: mentre il registro CS è già stato configurato secondo il *Flat Memory Model* (grazie al salto precedente), gli altri registri di segmento contengono ancora i vecchi valori della Real Mode. È quindi necessario aggiornare tutti i rimanenti selettori di segmento (DS, ES, FS, GS, SS) con il valore del descrittore Dati (`$0x10`). A livello implementativo, occorre ricordare un vincolo dell'architettura x86: non esistendo un opcode per caricare un valore immediato direttamente in un registro di segmento, è obbligatorio transitare attraverso un registro *general purpose*.

A completamento della configurazione dell'ambiente di esecuzione, rimane da inizializzare lo Stack Pointer. In questa specifica architettura didattica, la collocazione dello stack gode di una certa flessibilità: la memoria utilizzata dal BIOS, infatti, non necessita di essere preservata una volta ceduto il controllo al bootloader. La scelta è ricaduta sull'indirizzo `0x90000` per tre motivazioni strategiche:

1. Si trova a distanza di sicurezza dalla *Low Memory*, evitando conflitti con l'area che verrà utilizzata dal bootloader (`0x7C00`).

2. Si colloca nella regione tipicamente riservata alla *Extended BIOS Data Area* (EBDA).

3. Mantiene un ampio margine di sicurezza rispetto alla memoria video (che inizia a `0xA0000`), prevenendo corruzioni grafiche causate dalla crescita dello stack (che avviene verso il basso).

Ora che il sistema a 32 bit è pienamente operativo, è possibile iniziare a utilizzare il C. Per passare il controllo alla funzione principale del sistema, senza la necessità di riottenerlo successivamente, è sufficiente utilizzare l'istruzione `jmp` indirizzata all'etichetta della funzione desiderata (nel nostro caso: `main`). A differenza di una `call`, questa operazione trasferisce il flusso di esecuzione senza salvare alcun indirizzo di ritorno sullo stack (che rimarrebbe comunque inutilizzato), sancendo così il passaggio definitivo dal codice assembly di avvio al firmware scritto in C.

Capitolo 2

Libreria di supporto

Il sistema opera attualmente in un contesto definito *Bare Metal*: l'esecuzione avviene cioè in assenza di un sistema operativo sottostante e senza il supporto della Libreria Standard del C (`libc`). Di conseguenza, per poter proseguire nello sviluppo, è imperativo implementare manualmente una suite di funzioni di utilità.

2.1 Interfaccia I/O di base

La prima necessità riguarda l'interazione a basso livello con l'hardware. Il linguaggio C, per sua natura portabile, non prevede nativamente istruzioni per comunicare direttamente con lo spazio di indirizzamento I/O dell'architettura x86. Per colmare questa lacuna, è necessario definire delle funzioni specifiche che agiscano da "ponte" per invocare le istruzioni macchina di input e output.

Nel progetto si è scelto di adottare un sistema modulare tramite l'istruzione `extern`, che informa il Linker che la definizione della funzione risiede in un'altra unità di traduzione. In questo caso scriviamo la firma della funzione nel file `.h` in questo modo possiamo richiamarla ovunque includiamo l'header della nostra libreria di utilità, mentre il corpo della funzione lo implementiamo in un file assembly.

```
1    // Scrive un byte (data) sulla porta specificata (port)
2    extern void outb(uint8_t data, uint16_t port);
3
4    // Legge un byte dalla porta specificata (port)
5    extern uint8_t inb(uint16_t port);
```

Listing 3: Versione esemplificativa dei prototipi di I/O (File: utilities/utilities.h)

Di seguito vengono riportate come esempio le implementazioni per l'input/output di un byte verso le porte di I/O. Come si può notare, l'implementazione Assembly rispetta rigorosamente la convenzione di chiamata standard del C a 32 bit (`cdecl`),

allineandosi ai prototipi definiti nell'header per garantire la corretta gestione dello stack e il passaggio dei parametri.

```
1     .global outb
2     .type outb, @function
3
4     outb:
5         push %ebp           # Salva il Base Pointer del chiamante
6         mov %esp, %ebp      # Imposta il nuovo Base Pointer
7         mov 8(%ebp),  %al   # Carica il primo argomento (data) in AL
8         mov 12(%ebp), %dx   # Carica il secondo argomento (port) in DX
9         outb %al, %dx       # Scrive il valore di AL sulla porta in DX
10        pop %ebp            # Ripristina il Base Pointer
11        ret
12
13    .globl inb
14    .type inb, @function
15
16    inb:
17        push %ebp           # Salva il Base Pointer del chiamante
18        mov %esp, %ebp      # Imposta il nuovo Base Pointer
19        xor %eax, %eax      # Azzera EAX per pulire il valore di ritorno
20        mov 8(%ebp), %dx    # Carica il primo argomento (port) in DX
21        inb %dx, %al        # Legge dalla porta in DX e salva in AL
22        pop %ebp            # Ripristina il Base Pointer
23        ret
```

Listing 4: Implementazione Low-Level delle primitive I/O (File: utilities/utilities.s)

2.2 Definizione dei tipi primitivi

Data l'assenza della libreria standard del C, non è possibile utilizzare le macro e i tipi di dato con i quali si è soliti programmare (normalmente forniti da header come `stddef.h`, `stdbool.h` o `stdint.h`). Per rendere l'ambiente di sviluppo più familiare si procede alla loro definizione manuale.

- **La macro `NULL`**
  Tutti i programmatori la utilizzano per indicare un puntatore non valido. La sua definizione è molto semplice (`#define NULL ((void*)0)`).

- **Tipi booleani (`bool`, `true`, `false`)**
  Queste keyword non sono native del C storico (C89), ma sono state introdotte con lo standard C99 attraverso il tipo primitivo `_Bool` e l'header `stdbool.h`.

Possiamo replicare questo comportamento definendo il tipo tramite `typedef` e i valori di verità tramite un'enumerazione.

- **Interi a dimensione fissa (stile `stdint.h`)**
  In ambito embedded e bare-metal è fondamentale conoscere l'esatta dimensione delle variabili. Si adotta quindi la nomenclatura standard (es. `uint8_t`) per identificare istantaneamente la lunghezza in bit e il segno della variabile.

2.3 Funzioni di utilità: stampa a video e manipolazione della memoria

Una delle funzioni più utili definite nelle librerie standard è `memcpy`, anche essa è molto semplice da definire: si implementa attraverso un ciclo che copia ogni byte dall'indirizzo sorgente all'indirizzo di destinazione.

Un discorso a parte merita la funzione di stampa. Replicare fedelmente la `printf` della libreria standard rappresenta una sfida notevole, principalmente dovuta alla complessità legata alla gestione degli argomenti variabili (tramite `stdarg.h`). L'implementazione si concentra quindi su una funzione minimale per la stampa di stringhe grezze. L'architettura logica si basa su un ciclo che itera sui caratteri della stringa, copiandoli direttamente nella memoria video, associando a ciascuno l'attributo che ne definisce il colore, fino al raggiungimento del terminatore nullo '`\0`'. È stato inoltre integrato un meccanismo di gestione del carattere *newline* '(`\n`)' che interrompe la scrittura lineare e forza l'aggiornamento dell'offset di memoria all'inizio della riga successiva.

Tuttavia, esiste una differenza sostanziale rispetto all'output su `stdout`: scrivendo direttamente in memoria video, è necessario gestire un sistema di coordinate (x, y) per determinare il punto di inizio della scrittura. Poiché nella maggior parte dei casi si desidera semplicemente proseguire la stampa dal punto in cui si era rimasti o andare a capo, l'obbligo di specificare ogni volta le coordinate risulterebbe oneroso per il programmatore. Per risolvere questo problema, è stata creata una suite di funzioni *wrapper* attorno alla primitiva di stampa. Queste funzioni gestiscono il cursore interno della funzione, permettendo di stampare in sequenza senza dover scrivere manualmente gli offset. Per l'implementazione di questi wrapper si è fatto ricorso alla keyword `inline`. Questa istruzione suggerisce al compilatore di sostituire la chiamata alla funzione con il corpo della funzione stessa, eliminando l'overhead dovuto al salto (prologo ed epilogo dello stack frame). La disponibilità nativa di `inline` è, tra l'altro, uno dei motivi determinanti per la scelta dello standard C99 (rispetto al classico ANSI C), aspetto che verrà approfondito nel capitolo 5 dedicato alla compilazione.

Un'altra esigenza frequente è la visualizzazione di valori interi, spesso richiesti in base esadecimale (essenziale, ad esempio, per la verifica degli indirizzi di memoria). Tuttavia, come già discusso, l'implementazione di un supporto completo agli argomenti variabili risulta eccessivamente complessa. Si è pertanto optato per lo

sviluppo di una funzione dedicata alla conversione da intero a stringa ASCII. L'algoritmo implementato ricalca lo standard delle funzioni note come *itoa* (Integer to ASCII), basandosi sul metodo delle divisioni successive per estrarre le singole cifre del numero nella base desiderata. A livello operativo, costringere il programmatore a invocare esplicitamente la conversione prima di ogni stampa risulterebbe ridondante e poco pratico. Per ottimizzare il flusso di lavoro, sono stati introdotti due ulteriori *wrapper* che astraggono questo passaggio, permettendo di stampare il valore numerico in sequenza (accodandosi all'ultimo output) oppure su una nuova riga.

Le primitive di stampa richiedono, tra gli argomenti, un attributo che specifichi il colore del carattere e dello sfondo, il quale verrà copiato nella memoria video unitamente al carattere ASCII. Per rendere la stesura del codice più fluida e meno incline a errori, si è scelto di definire una serie di costanti predefinite (impostate su sfondo nero). Questo approccio astrae la complessità dell'hardware, esonerando il programmatore dalla necessità di consultare costantemente le tabelle della mappatura colori VGA. Inoltre, partendo da queste costanti base, sarà possibile in futuro ottenere qualsiasi combinazione cromatica applicando opportune operazioni logiche bit a bit per unire i colori di background a quelli di foreground. Per semplificare ulteriormente le fasi di sviluppo e debugging, si è deciso di implementare un sistema preliminare di *logging* semantico. Invece di riferirsi ai colori espliciti, sono state definite delle macro che descrivono la "tipologia" del messaggio (es. `ERROR`, `SUCCESS`), associandole visivamente a colori intuitivi.

Una funzione indispensabile per la gestione del terminale video è la routine `clearScreen`. L'implementazione si basa su un ciclo che sovrascrivendo l'area di memoria video con spazi con attributo bianco su nero permette di ripulire il terminale. Questa routine, come si vedrà nel capitolo successivo, è parte integrante della procedura di inizializzazione della VGA in modalità testo.

L'implementazione di questa libreria di supporto, seppur essenziale, fornisce una base solida per operare a un livello superiore rispetto al puro codice macchina. Grazie a queste primitive, le successive routine di inizializzazione dell'hardware potranno essere scritte con un approccio modulare e leggibile, offrendo un punto di partenza dal quale astrarre secondo le necessità. Nel prossimo capitolo sfrutteremo queste fondamenta per inizializzare il primo dispositivo: la VGA.

Capitolo 3

Inizializzazione della VGA in modalità testo 80x25

Inizializzare la VGA come primo passo è una scelta strategica: questo ci permette di effettuare il debugging stampando direttamente a schermo, senza dover dipendere esclusivamente dai log di QEMU. Data la particolare modalità di scrittura nei registri video, ci concentreremo inizialmente sull'astrarre le funzioni di I/O sviluppate nel capitolo precedente. L'obiettivo è creare una base che ci permetta di configurare la VGA senza doversi preoccupare ogni volta dei dettagli di basso livello dei singoli registri.

3.1 Astrazione dell'interfaccia I/O per la comunicazione con la VGA

Il controller VGA contiene al suo interno centinaia di registri di configurazione. Tuttavia, solo un sottoinsieme ridotto è accessibile direttamente dal programmatore e, per economizzare sullo spazio di indirizzamento I/O, molti di essi condividono gli stessi indirizzi fisici.

Per rendere l'accesso più intuitivo e meno soggetto a errori, è utile astrarre le funzioni originali creando un header di funzioni *wrapper*. Dato che la logica di accesso è breve e che deve essere estremamente performante, queste funzioni verranno definite come `inline`. Possiamo classificare i registri VGA in tre macro-categorie, distinte in base alla modalità di accesso hardware:

1. **Registri Generali (*General Registers*)**
   Si tratta di registri mappati direttamente su porte I/O dedicate o condivise in lettura/scrittura. Tra questi, il più rilevante è il *Miscellaneous Output* (Scrittura su `0x3C2`, Lettura su `0x3CC`). Il bit 0 di questo registro è cruciale, in quanto definisce la mappatura delle porte I/O per gli altri registri (0 per emulazione Monocromatica `0x3Bx`, 1 per Colore `0x3Dx`). Sebbene il sistema di indi-

rizzamento possa apparire frammentato, questi registri possono essere gestiti facilmente tramite le macro degli indirizzi e le funzioni `outb`/`inb` standard.

2. **Registri Indicizzati (Sequencer, CRTC, Graphics Controller)**
   Questi tre gruppi di registri condividono la medesima architettura di accesso basata su coppie di porte I/O contigue: un registro *Index* (a indirizzo N) e un registro *Data* (a indirizzo $N + 1$).

   Per ottimizzare e rendere più fluida la programmazione, implementeremo tre funzioni *wrapper* specifiche:

   (a) **Lettura:** Una funzione standard che scrive l'indice sulla porta dedicata e legge il valore dalla porta dati successiva.

   (b) **Scrittura Ottimizzata:** Sfruttando l'architettura *Little Endian* dei processori x86 e la contiguità delle porte, è possibile eseguire una singola scrittura a 16 bit (`outw`) sulla porta dell'Indice. Il bus indirizzerà automaticamente gli 8 bit meno significativi (LSB) al registro Indice e gli 8 bit più significativi (MSB) al registro Dati (indirizzo successivo). Questo riduce l'overhead dimezzando le operazioni di I/O.

   (c) **Modifica Bitwise:** Poiché ogni singolo bit nei registri VGA controlla funzionalità hardware specifiche, è frequente dover modificare solo una parte del registro senza alterare le altre. Questa funzione implementa il pattern *Read-Modify-Write*: legge il valore attuale, applica le modifiche richieste tramite operazioni logiche (maschere AND/OR) e riscrive il valore aggiornato sfruttando la scrittura ottimizzata.

3. **Attribute Controller Registers**
   Questo gruppo di registri ha una peculiarità rispetto agli altri: la porta `0x3C0` funziona sia da registro Indice che Dati. L'hardware lo gestisce tramite un Flip-Flop interno che commuta lo stato ad ogni accesso. Sebbene il registro inizi nello stato di "Indice", è facile perdere la sincronizzazione durante la programmazione, col rischio di inviare un indice quando l'hardware si aspetta un dato. L'architettura VGA prevede però un meccanismo di sicurezza: una lettura dal registro *Input Status 1* (`0x3DA`) resetta forzatamente il flip-flop. Per rendere il codice robusto, si crea una macro di reset da chiamare all'inizio delle funzioni di lettura e scrittura (non ottimizzata) dedicate a questo gruppo. Così facendo, ogni operazione sarà coerente, eliminando la necessità di tracciare via software lo stato del registro.

Adesso che si sono definite le Macro per ogni registro e che si è definito una piccola libreria di funzioni *wrapper* per interagire con essi, adesso possiamo procedere all'inizializzazione vera e propria.

3.2 Configurazione della modalità testo e caricamento della palette

La sequenza di inizializzazione implementata nel kernel è derivata dal lavoro di riferimento di Chris Giese [2], adattata per integrarsi con le strutture dati definite nel nostro sistema. A differenza del codice originale, progettato per funzionare in un ambiente dove il BIOS ha già predisposto l'hardware, la nostra implementazione *Bare Metal* richiede l'aggiunta di una fase esplicita per il caricamento della Palette (DAC). Il procedimento di inizializzazione segue uno schema rigido, di seguito riportato, atto a garantire la stabilità del segnale video.

1. **Caricamento della Palette (DAC)**
   L'operazione avviene tramite i registri del convertitore Digitale-Analogico (DAC). Si scrive l'indice di partenza sulla porta *Write Address* (`0x3C8`) e successivamente i valori RGB sulla porta *Data* (`0x3C9`). Sfruttiamo qui una particolarità hardware: il registro indice possiede un contatore interno con *auto-incremento*. È quindi sufficiente impostare l'indice iniziale a 0 e copiare sequenzialmente l'intero array di colori (3 byte per colore per 64 colori) sulla porta dati, senza dover riscrivere l'indirizzo a ogni iterazione.

2. **Disattivazione del segnale video (Screen Blanking)**
   Per prevenire artefatti visivi ("sfarfallio") e corruzione della memoria video durante la configurazione, è necessario disabilitare l'output. Questo si ottiene impostando a 1 il bit 5 del registro *Clocking Mode* (Indice `0x01`) all'interno del gruppo *Sequencer*.

3. **Scrittura dei Registri di Configurazione**
   Si procede al caricamento dei valori per la modalità 80x25, rispettando il seguente ordine al fine di evitare stati inconsistenti:

   - *Sequencer Registers*
   - *Miscellaneous Output Register*
   - *CRTC Registers (Cathode Ray Tube Controller)*
   - *Graphics Controller Registers*
   - *Attribute Controller Registers*

   Nota sulla protezione dei registri CRTC: i primi otto (da 0 a 7) sono protetti dalla scrittura diretta. Per potervi accedere, è necessario seguire una specifica procedura di sblocco: occorre dapprima azzerare il bit 7 del registro 3 per abilitare l'accesso ai successivi 10 e 11; dopodiché, operando sul bit 7 del registro 11, si rimuove definitivamente la protezione. Per un'analisi dettagliata di ogni singolo bit, si rimanda al manuale di R. F. Ferraro [1].

4. **Abilitazione della Palette**
   Il registro Indice dell'Attribute Controller utilizza il bit 5 (noto come *Palette Address Source*) per determinare chi ha accesso alla memoria dei colori: la CPU o il circuito video. Terminata la fase di configurazione, è necessario impostare questo bit a 1. Così facendo si cede il controllo all'hardware VGA, permettendogli di leggere correttamente i valori dalla palette e generare i colori a schermo.

5. **Pulizia della memoria (`clearScreen`)**
   Si azzera il buffer della memoria video (che in modalità testo inizia a `0xB8000`) per rimuovere eventuali dati spuri rimasti in memoria.

6. **Riattivazione dello schermo**
   Come passo finale, si riabilita l'output video azzerando il bit *Screen Off* nel Sequencer e, per sicurezza, si riattiva la protezione di scrittura sui registri CRTC per prevenire modifiche accidentali.

I dati relativi a font, palette e configurazione dei registri (contenuti nel file `dataVGA.c`) sono stati derivati dal codice sorgente di SeaBIOS [8], il firmware di riferimento per QEMU, al fine di garantire la massima compatibilità con l'hardware emulato.

3.3 Gestione dei piani di memoria e scrittura del font

In modalità testo, l'architettura VGA ripartisce le funzioni sui diversi piani di memoria: i Piani 0 e 1 contengono rispettivamente i codici ASCII e gli attributi colore, mentre il Piano 2 è destinato ai dati che definiscono la forma dei caratteri (glifi). Pertanto, per completare l'inizializzazione della modalità 80x25, è necessario trasferire la bitmap dei font nel piano dedicato.

L'accesso diretto al Piano 2 non è immediato, poiché la modalità testo attiva logiche di indirizzamento complesse. Prima di procedere alla scrittura, è dunque necessario disabilitare le impostazioni che regolano l'accesso ai piani di memoria. Nello specifico, si deve intervenire disabilitando il bit *Chain Four* (bit 3 del *Memory Mode register*), il bit *odd/even* (bit 4 del *Mode Register*) e il bit *Chain odd/even* (bit 1 del *Miscellaneous register* del Graphic Controller). Per abilitare la scrittura sul Piano 2, si impostano opportunamente il *Read Map Select Register* e il *Map Mask Register*. Durante la copia dei glifi, è fondamentale considerare che la VGA preleva i dati seguendo uno schema a passo fisso, descritto dalla formula:

$$\text{Offset} = (C \times 32) + R$$

dove:

C **(Codice ASCII):** È il valore numerico del carattere (da 0 a 255).

32 **(Fixed Stride):** Rappresenta la dimensione fissa allocata dalla VGA per ogni glifo. Indipendentemente dall'altezza effettiva del font, l'hardware riserva sempre un blocco di 32 byte per carattere.

R **(Scanline):** Indica l'indice della riga del carattere che si sta scrivendo (variabile da 0 a 15 per un font 8×16).

Di conseguenza, la routine di scrittura deve rispettare questo padding: ogni glifo deve essere allineato a 32 byte, anche se la dimensione effettiva dei dati del font (nel nostro caso: 16×8 bit, ovvero 16 byte) è inferiore allo spazio riservato. La logica implementativa del codice è stata derivata dal lavoro di riferimento di Chris Giese [2]. L'attivazione della modalità testo dota il nostro codice di un canale di output fondamentale. Disponendo ora di un riscontro visivo immediato, essenziale per le attività di debug, è possibile procedere con maggiore sicurezza alla fase successiva: l'enumerazione del bus PCI e l'assegnazione degli indirizzi ai dispositivi connessi, inclusa la stessa interfaccia VGA.

Capitolo 4

Configurazione delle periferiche PCI

In questo capitolo affronteremo la fase di *Discovery*, necessaria per individuare i dispositivi connessi al bus PCI e capire, tramite il *Probing*, di quali risorse hanno bisogno. Una volta raccolti questi dati, potremo assegnare a ciascun dispositivo il proprio spazio di indirizzamento in memoria e I/O. Prima di procedere con l'algoritmo vero e proprio, però, è necessario astrarsi dai dettagli hardware dello spazio di configurazione: definiremo quindi le primitive software utili a rendere la programmazione più fluida e indipendente dalla complessità del livello fisico.

4.1 Astrazione dell'interfaccia I/O per l'accesso allo Spazio di Configurazione PCI

Per interagire con lo spazio di configurazione PCI si utilizzano due registri a 32 bit mappati nello spazio di I/O: il *Configuration Address Port* (CAP) all'indirizzo `0xCF8` e il *Configuration Data Port* (CDP) all'indirizzo `0xCFC`.

Il primo registro (CAP) ha il compito di indirizzare univocamente la risorsa, specificando bus, dispositivo, funzione e offset del registro desiderato. Il secondo (CDP) è la porta attraverso cui avviene l'effettivo scambio dati (lettura o scrittura) una volta impostato l'indirizzo. Affinché l'indirizzamento sia valido, il CAP richiede una specifica formattazione dei dati nei suoi 32 bit, come illustrato in Figura 4.1, ed è fondamentale che il bit 31 (*Enable Bit*) sia impostato a 1.

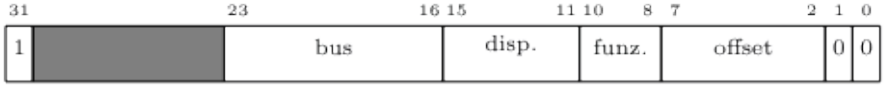

Figura 4.1: Struttura del *Configuration Address Port* (CAP) (Fonte: [6])

Dal punto di vista implementativo, è stata realizzata una funzione ausiliaria che, dati in ingresso i parametri del dispositivo (Bus, Device, Function, Offset), utilizza operazioni logiche (bitwise shift e OR) per comporre il valore a 32 bit da scrivere nel registro CAP. A valle di questa preparazione, l'accesso viene gestito da due funzioni `inline` dedicate rispettivamente alla lettura e alla scrittura nello spazio di configurazione.

4.2 Implementazione della fase di Discovery e Probing dei BAR

4.2.1 Fase di Discovery

La fase di *Discovery* è concettualmente lineare: essa consiste nell'iterare su tutti i possibili dispositivi connessi a un bus. In uno scenario completo, qualora venisse identificato un bridge PCI-PCI, sarebbe necessario estendere la ricerca ricorsivamente anche al bus secondario (effettuando una visita *Depth First*), al fine di calcolare correttamente la quantità di indirizzi richiesta dall'intero sotto-albero. Tuttavia, l'ambiente di emulazione QEMU x86 di riferimento presenta, nella sua configurazione standard, esclusivamente il Bus 0. Di conseguenza, si è scelto di limitare l'attuale implementazione alla scansione di tale BUS. Nonostante questa semplificazione operativa, la funzione è stata strutturata in modo da poter supportare la ricorsività: per rendere il BIOS compatibile con sistemi a bus annidati, in futuro sarà sufficiente integrare la logica specifica di gestione dei bridge nella sezione di codice indicata.

La procedura inizia con la verifica preliminare di esistenza del dispositivo: una lettura del *Vendor ID* della funzione 0 che restituisca un valore diverso da `0xFFFF` conferma che lo slot è occupato. Successivamente, si esamina il registro *Header Type*. Se il bit di stato "Multifunction" (Bit 7) è attivo, la scansione viene estesa a tutte le 8 possibili funzioni; in caso contrario, si interroga esclusivamente la prima. Per ogni funzione rilevata (ovvero con *Vendor ID* valido), si procede a discriminarne la tipologia mascherando il bit di multifunzione: se i restanti bit dell'*Header Type* sono a zero (indicando un dispositivo standard), si avvia la fase di *Probing* dei BAR.

Un'eccezione specifica è stata prevista per l'*Host Bridge*. Sebbene sottoporre questo dispositivo al probing non comporti errori critici (essendo esso privo di BAR configurabili), si è scelto di escluderlo esplicitamente dalla procedura per rigore formale e per evitare operazioni ridondanti.

4.2.2 Configurazione del PCI Routing

Contestualmente alla fase di *Discovery*, che opera a livello di dispositivo (prima di scendere nel dettaglio delle singole risorse BAR), è fondamentale configurare il registro *Interrupt Line*. Come specificato nella Sezione 2.3 (*Le interruzioni*) [6], è compito del BIOS informare il sistema operativo su quale ingresso del controller delle interruzioni (IO-APIC) è fisicamente collegato il dispositivo.

Nelle architetture x86, le quattro linee di interrupt del bus PCI (INTA#, INTB#, INTC#, INTD#) non sono collegate linearmente al controller, ma seguono uno schema di rotazione noto come *PCI Swizzling*, che varia in base allo slot occupato. Questo meccanismo serve a distribuire il carico delle interruzioni tra i quattro ingressi disponibili sul chipset (PIRQ[A-D]). Nel contesto dell'hardware emulato da QEMU (chipset PIIX3), la logica di routing per determinare l'indice della linea PIRQ (valore 0-3) è definita dalla seguente formula:

$$PIRQ = ((INTx + \text{Slot} - 2) \ \& \ 3) \tag{4.1}$$

dove $INTx$ rappresenta l'indice del pin del dispositivo (1 per INTA#, 2 per INTB#, ecc.) e & 3 applica un modulo 4 tramite maschera bitwise.

Per ottenere il numero effettivo del pin di input sull'IO-APIC, è necessario aggiungere un offset di 16 al risultato, poiché le linee PIRQ[A-D] sono tipicamente mappate sugli ingressi 16-19 del controller. Questa specifica implementazione software rispecchia fedelmente il comportamento dell'emulatore, come verificabile nella funzione `pc_pci_slot_get_pirq` presente nel codice sorgente di QEMU (file `hw/i386/pc_piix.c`, riga 86 [9]).

Dal punto di vista operativo, il BIOS esegue quindi i seguenti passaggi per ogni funzione rilevata:

1. Verifica il registro *Interrupt Pin*: se il valore è diverso da 0, il dispositivo richiede l'uso di interruzioni.

2. Calcola il vettore di interrupt corretto applicando la formula di routing sopra descritta con l'offset di base (16).

3. Scrive il risultato nel registro *Interrupt Line* (offset `0x3C`) del dispositivo.

Questa operazione garantisce che il driver del sistema operativo possa recuperare immediatamente l'informazione di routing.

4.2.3 Fase di Probing

La procedura di *Probing* dei BAR segue un protocollo standardizzato: si scrive un valore di tutti '1' (`0xFFFFFFFF`) nel registro e se ne effettua immediatamente la rilettura, seguita dal ripristino del valore iniziale. Se il valore restituito è conforme alle specifiche (descritte nella Sezione 2.2.3 *I Base Address Register* [6]), significa che il BAR è implementato e attivo. Dalla decodifica del valore letto è possibile estrarre informazioni cruciali:

- **Spazio di indirizzamento (Bit 0):** Determina se il dispositivo richiede la mappatura in spazio di I/O (Bit 0 = 1) o in memoria (Memory Mapped I/O, Bit 0 = 0).

- **Ampiezza del registro (Bit 2-1):** Nei BAR di memoria, se questi bit valgono `0x2`, il registro è a 64 bit. In tal caso, il software deve trattare il BAR corrente e quello successivo come un'unica entità logica a 64 bit.
- **Dimensione richiesta:** È il dato fondamentale per l'allocazione. La quantità di memoria necessaria si ottiene mascherando i bit di flag, invertendo il risultato e aggiungendo 1.

Per garantire la robustezza dell'algoritmo, sono state implementate due ulteriori verifiche. La prima riguarda i cosiddetti *Shadow BAR*: registri che, pur presentando bit di flag validi, restituiscono una dimensione nulla (non richiedono risorse) e devono quindi essere ignorati dall'allocatore. La seconda riguarda la gestione dell'*Expansion ROM Base Address Register* (offset `0x30`). Questo registro permette di mappare in memoria il codice di inizializzazione specifico della scheda. Tuttavia, poiché l'utilità di tale mappatura dipende dallo scenario d'uso, la relativa logica di probing è stata inclusa in un blocco di compilazione condizionale. L'attivazione è controllata dalla macro `MAP_EXP_ROM`, definita nel file di configurazione `pci/config.h`.

4.3 Allocazione e gestione dello spazio di indirizzamento

4.3.1 Preparazione e Sorting delle risorse

Durante la fase di *Probing*, sorge la necessità di memorizzare le caratteristiche di ogni BAR rilevato per poter procedere, in un secondo momento, all'assegnazione delle risorse. Idealmente, tali dati dovrebbero essere organizzati in due strutture dati dinamiche distinte: una per lo spazio di I/O e una per lo spazio di memoria (MMIO). Tuttavia, l'ambiente di esecuzione impone vincoli stringenti. Poiché il codice del BIOS risiede in ROM (*Read-Only Memory*), non è possibile scrivere dati all'interno di tale segmento. Inoltre, operando in modalità *Bare Metal*, il sistema è privo delle librerie standard e, di conseguenza, di un allocatore di memoria dinamica (come la funzione `malloc`). Per ovviare a queste limitazioni, si è optato per una strategia di allocazione manuale a indirizzi fissi. Le strutture dati contenenti gli array dei BAR I/O e MMIO sono state posizionate rispettivamente agli indirizzi fisici `0x10000` e `0x30000`. Tale scelta ricade all'interno della cosiddetta *Conventional Memory* (specificamente nella zona libera compresa tra `0x00007E00` e `0x0007FFFF`). Come già discusso in fase di inizializzazione dello stack, in questa fase preliminare di avvio il sistema impone vincoli minimi sull'occupazione della memoria fatta eccezione per poche aree riservate specifiche. Per un approfondimento dettagliato sulla mappa di memoria dell'architettura x86, si rimanda alla documentazione di OSDev [7].

Al termine della fase di *Probing*, i dati relativi a tutti i BAR rilevati risultano organizzati nei due array distinti. Prima di procedere all'allocazione effettiva, è imperativo sottoporre entrambe le liste a un'operazione di ordinamento (*sorting*) decrescente basato sulla dimensione richiesta.

Questa necessità nasce dal vincolo di *allineamento naturale* imposto dall'architettura PCI: l'indirizzo base di un BAR di dimensione S deve essere obbligatoriamente un multiplo di S. Adottando una strategia di allocazione "Largest-First" (dal blocco più grande al più piccolo), si garantisce il rispetto automatico dei vincoli di allineamento per ogni blocco successivo. Questo approccio ottimizza l'uso dello spazio, assicurando la contiguità delle assegnazioni ed eliminando la formazione di "gap" (frammentazione esterna) tra un dispositivo e l'altro.

4.3.2 Implementazione del processo di allocazione

Si procede, infine, all'assegnazione vera e propria degli indirizzi ai dispositivi. Per ogni BAR della lista ordinata, il sistema verifica preliminarmente il rispetto dell'allineamento naturale rispetto all'indirizzo base corrente. Qualora l'allineamento non fosse soddisfatto, l'indirizzo viene adeguato al multiplo corretto e il sistema notifica la generazione di un "gap" nello spazio di indirizzamento. Successivamente, l'indirizzo viene scritto nel registro del dispositivo, gestendo opportunamente i casi di BAR a 64 bit, i quali richiedono la scrittura su due registri a 32 bit adiacenti.

Un passaggio critico dell'algoritmo consiste nella verifica post-scrittura (*readback*). Immediatamente dopo l'assegnazione, il registro viene riletto per confermare che l'hardware abbia correttamente recepito il nuovo indirizzo. In caso di fallimento di questa verifica, viene eseguito un *rollback*, ripristinando il contenuto originale del BAR per impedire che il dispositivo resti in uno stato indefinito o pericoloso. Inoltre, per ragioni di sicurezza, la porzione di indirizzi originariamente destinata al BAR fallato non viene riutilizzata per i dispositivi successivi. Questa scelta previene situazioni inconsistenti in cui, a causa di errori hardware nel readback, due periferiche finiscano per condividere erroneamente il medesimo spazio di indirizzamento.

Per evitare i cosiddetti "*Magic Number*" e garantire la modularità del codice, le zone di memoria e di I/O su cui mappare i BAR sono state definite come macro nel file `pci/config.h`. L'algoritmo integra inoltre i controlli necessari affinché l'allocazione non esca dai limiti specificati (*boundary check*). In particolare, come spazio di memoria è stato reso disponibile l'ultimo Gigabyte di indirizzamento a 32 bit, mentre per lo spazio di I/O si è scelta la zona compresa tra `0xC000` e `0xFFFF`. Tale intervallo è stato selezionato in quanto risulta libero da conflitti, come confermato anche dall'implementazione di riferimento di SeaBIOS (`src/fw/pciinit.c`, riga 1012 [8]).

Capitolo 5

Compilazione

In questo capitolo discuteremo le scelte progettuali fatte nella fase di compilazione, e partendo dal codice del BIOS, arriveremo ad ottenere un file binario che rispecchia la struttura di una ROM. Per poi soffermarci sulle tecniche di debug utilizzate durante la stesura del codice, utili soprattutto nella prima parte di scrittura del codice in cui non si ha la possibilità di stampare a schermo.

5.1 Flusso di compilazione e Makefile

Data la complessità dei passaggi necessari per ottenere il file binario e l'elevato numero di parametri richiesti, si è reso indispensabile l'utilizzo del *Makefile*, lo standard de facto per la gestione di progetti di questo tipo. Il codice viene compilato tramite *GCC*; trattandosi di un progetto di programmazione *Bare Metal*, i flag di compilazione assumono un'importanza fondamentale, poiché permettono di istruire il compilatore sull'ambiente target, garantendo una generazione del codice coerente con i vincoli hardware. Di seguito si approfondiscono i flag utilizzati:

- **-fno-pie -no-pie**
  Questi flag sono strettamente necessari in ambito *Bare Metal*. Essi specificano che il file binario non deve essere *Position Independent* (rilocabile), a differenza di quanto avviene nei comuni sistemi operativi dove il codice può essere caricato in diversi punti della memoria. Il primo flag forza l'uso di indirizzamenti assoluti anziché relativi, mentre il secondo istruisce il linker a generare un eseguibile statico anziché un file ELF rilocabile. Come vedremo nella sezione dedicata al *Linker Script*, ciò permette di assegnare un indirizzo di base fisso al codice.

- **-nostdlib -ffreestanding -fno-builtin**
  Rappresentano i pilastri della programmazione *Bare Metal*. `-nostdlib` evita il collegamento automatico alla libreria standard C (*libc*), non disponibile nel nostro ambiente. `-ffreestanding` avverte il compilatore che il codice verrà eseguito in un contesto privo di sistema operativo (*Hosted* vs *Freestanding*). Infine, `-fno-builtin` impedisce al compilatore di sostituire frammenti di codice

con chiamate a funzioni standard ottimizzate (come la `memcpy`), che non sono presenti nel nostro binario.

- **-fno-stack-protector**
  Disabilita lo *stack protector*, una tecnica di sicurezza contro il *buffer overflow* che, per funzionare correttamente, richiede il supporto del sistema operativo.

- **-Wall -Wextra -O2 -g3**
  Parametri comuni anche in ambienti *Hosted*. I primi due attivano i messaggi di avvertimento (*warning*) per prevenire potenziali errori logici. `-O2` abilita il secondo livello di ottimizzazione del codice, mentre `-g3` include nel file oggetto le informazioni estese per il debug.

- **-mno-sse -mno-sse2 -mno-mmx -mno-80387**
  Queste opzioni inibiscono l'uso delle unità di calcolo in virgola mobile e vettoriali (FPU e SIMD). Sebbene non richieste esplicitamente, il compilatore potrebbe utilizzarle autonomamente per ottimizzare operazioni comuni (ad esempio, usando i registri XMM a 128 bit per una `memcpy`). La disabilitazione è necessaria per evitare l'uso di registri che richiederebbero una complessa inizializzazione dello stato del processore.

- **-m32**
  Questo flag obbliga GCC a generare un file ELF a 32 bit per architettura i386. Nonostante il progetto contenga parti di codice a 16 bit, questa è la scelta più corretta: la modalità a 16 bit di GCC, infatti, inserisce comunque il codice in un file ELF a 32 bit e compila l'assembly utilizzando la direttiva `.code16gcc`. Quest'ultima non genera vero codice a 16 bit nativo, ma istruzioni a 32 bit precedute da prefissi di override per renderle compatibili con la modalità reale. Di conseguenza, la strategia migliore è utilizzare stabilmente `-m32`, avendo l'accortezza di specificare esplicitamente nel codice assembly le direttive `.code16` e `.code32` nelle sezioni che l'assemblatore (*GAS*) deve tradurre rispettivamente a 16 o 32 bit.

- **-std=c99**
  Si è adottato lo standard C99 per l'ottimo bilanciamento tra compatibilità e funzionalità moderne. Una scelta progettuale rilevante è stata l'ampio uso dell'istruzione `inline`, utilizzata per ridurre l'overhead delle chiamate a funzione e aumentare l'efficienza. Tuttavia, dato che lo spazio in ROM è limitato, l'eventuale rimozione degli `inline` rappresenta una possibile strategia di ottimizzazione per recuperare spazio nel binario finale.

Una volta generato il file in formato *ELF*, il binario puro viene estratto tramite l'utility `objcopy` della suite *GNU Binutils*, utilizzando il flag `-O binary`.

5.2 Linker Script e struttura della ROM

Dopo aver reso il file *ELF* non rilocabile in fase di compilazione, utilizziamo il *Linker Script* per definire gli indirizzi e la struttura interna dell'eseguibile e, di conseguenza, del binario finale che costituirà la ROM. Attraverso le direttive globali, specifichiamo il *formato* del file generato (`elf32-i386`) e l'*entry point*, ovvero l'etichetta (`entry`) da cui inizierà l'esecuzione del codice.

Successivamente, definiamo l'area di memoria fisica dedicata alla ROM. Nel nostro caso, l'indirizzo di inizio è fissato a `0xFFFE0000` con una dimensione complessiva di 128 KB. Tale dimensione è stata scelta per soddisfare i requisiti di QEMU, che impone che la ROM sia un multiplo di 64 KB.

L'area di memoria è stata logicamente suddivisa in due blocchi:

- **Blocco superiore (ultimi 64 KB):** Contiene il codice assembly critico per la gestione del processore, incluso il passaggio dalla modalità reale alla modalità protetta. Questo codice deve obbligatoriamente risiedere nel segmento più alto, poiché la configurazione iniziale dei registri di segmento (CS) all'accensione permette l'esecuzione solo all'interno di questo intervallo, fino all'attivazione di una nuova GDT, come approfondito precedentemente nel Capitolo 1.

- **Blocco inferiore:** Viene utilizzato per mappare il codice sorgente scritto in C. Poiché allo stato attuale quest'ultimo non eccede i 64 KB, è stato inserito un *padding* di zeri per mantenere il corretto allineamento degli indirizzi. In futuro, qualora il codice C dovesse richiedere più spazio, sarà sufficiente rimuovere il padding per sfruttare appieno la capienza residua nel blocco superiore.

Ulteriori operazioni di padding sono state applicate tra il codice assembly e il *jump* iniziale, che deve essere posizionato tassativamente all'indirizzo del *Reset Vector* (`0xFFFFFFF0`), e infine tra quest'ultimo e il limite fisico della ROM (4 GB) per garantire che il file binario abbia la dimensione esatta richiesta. La generazione di questi spazi vuoti è stata ottenuta manipolando il *Location Counter* (`.`) all'interno dello script. Per una trattazione esaustiva della sintassi e delle potenzialità del linker GNU, si rimanda al manuale ufficiale della *Free Software Foundation* [3].

5.3 Strumenti di Debug

Sebbene il sistema sia ora sufficientemente inizializzato per utilizzare l'output a schermo come strumento di diagnostica, nelle prime fasi del progetto (precedenti all'inizializzazione della VGA) tale funzionalità non era disponibile. Si è reso quindi necessario l'impiego di strumenti di analisi e debug esterni per verificare la correttezza del codice.

Per quanto concerne la verifica del *Linker Script*, l'analisi è stata immediata: l'aggiunta del flag `-Map=`*`nome`*`.map` in fase di compilazione ha permesso di generare un file mappa contenente i dettagli esatti sull'allocazione dei simboli e delle sezioni in memoria.

Il debug del codice eseguito in ROM ha invece richiesto un approccio più elaborato, sfruttando le funzionalità di tracciamento nativamente offerte dall'emulatore QEMU su due diversi livelli operativi. Da un lato, si è operato sul tracciamento I/O (VGA e PCI): per monitorare le operazioni di lettura e scrittura in questo spazio di indirizzamento, fondamentali durante l'inizializzazione del bus PCI e della scheda video, QEMU è stato avviato con i flag `-trace enable=vga*, enable=pci*`.

Dall'altro lato, ci si è concentrati sul logging a basso livello dell'esecuzione della CPU. Per la primissima fase di avvio, in cui si effettua la delicata transizione dalla *Real Mode* alla *Protected Mode*, si è fatto ricorso al flag di logging avanzato `-d in_asm,cpu_reset,exec,unimp,int,cpu`. Questo comando registra ogni singola istruzione assembly eseguita e le relative eccezioni. Tuttavia, data la mole massiccia di dati generata, se ne sconsiglia l'uso nelle fasi successive all'avvio per non sovraccaricare l'analisi. Per un elenco completo degli eventi tracciabili, è possibile fare riferimento alla documentazione interna di QEMU, accessibile tramite il parametro `help`.

Infine, un'ulteriore metodologia di analisi si basa sull'integrazione con il GNU Debugger (GDB). Avviando QEMU con il flag `-s`, l'emulatore attiva un server GDB locale in ascolto sulla porta TCP 1234. Connettendosi a questa istanza da un terminale separato, è possibile arrestare l'esecuzione, ispezionare lo stato del processore e dei registri, ed eseguire il codice *step-by-step* sfruttando i simboli di debug incorporati precedentemente nel binario grazie al flag di compilazione `-g3`.

Conclusione

Il presente lavoro ha permesso di trasformare un sistema a 16 bit, limitato a soli 64 KB di memoria accessibile, in un sistema a 32 bit completamente operativo, dotato di output visivo e pieno accesso a tutte le periferiche hardware. Tale traguardo è stato raggiunto gestendo la delicata transizione dalla *Real Mode* alla *Protected Mode*, operazione che ha richiesto la configurazione di un nuovo modello di segmentazione, noto come *Flat Memory Model*.

Successivamente, si è proceduto all'inizializzazione della scheda video VGA in modalità testo 80x25, il che ha consentito di approfondirne la complessa struttura hardware e la logica di gestione dei glifi in memoria. Tramite l'implementazione di una routine di *Discovery*, è stato poi possibile identificare i dispositivi collegati al bus PCI e gestire il relativo *PCI Routing*. Su tali dispositivi è stata infine eseguita la fase di *Probing* dei registri BAR, allocando il corretto spazio di indirizzamento in memoria e I/O. Contestualmente, la gestione del processo di compilazione ha permesso di entrare in contatto con i flag di compilazione tipici della programmazione in ambiente *Freestanding* e di padroneggiare l'uso del *Linker Script*.

Allo stato attuale, ci troviamo di fronte a un sistema non ancora del tutto completo, ma sufficientemente stabile per garantire una programmazione efficace. Con poche ulteriori integrazioni, il firmware sviluppato sarà pienamente in grado di passare il controllo a un *Bootloader*, completando così la catena di avvio del calcolatore.

Bibliografia

[1] Richard F. Ferraro. Programmer's guide to the ega and vga cards, second edition. page 1040, 06 1991.

[2] Chris Giese. Sets vga-compatible video modes without using the bios. n.d. Codice di Pubblico Dominio. Mirror ospitato su OSDev.org. Consultato il 11/2025.

[3] GNU Project, Free Software Foundation. The gnu linker user guide. 2024. Consultato il: 11/2025.

[4] GNU Project, Free Software Foundation. Using the gnu compiler collection (gcc). 2026. Consultato il: 11/2025.

[5] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual. *Intel Documentation*, Combined Vols 1-4:5198, 06 2025.

[6] Giuseppe Lettieri. Il bus pci. page 14, 04 2024.

[7] OSDev Wiki. Memory map (x86). 2024. Consultato il 11/2025.

[8] SeaBIOS Project. Seabios: Open source implementation of a 16-bit x86 bios. n.d. Documentazione ufficiale e repository. Consultato il 11/2025.

[9] The QEMU Project. Qemu source code. 2024. Consultato il 11/2025.