# Progetto e realizzazione del supporto video per un nucleo multiprogrammato su architettura RISC-V

Tesi di Laurea Triennale - Ingegneria Informatica


UNIVERSITÀ DI PISA

Dipartimento di Ingegneria dell'Informazione

Relatore:
**Prof. Giuseppe Lettieri**

Candidato:
**Matteo Giugni**

Ottobre 2024

# Indice

**1 Introduzione** **4**
1.1 Caratteristiche RISC-V . . . . . 4
1.2 Obiettivi . . . . . 7
1.3 Nota . . . . . 7

**2 Prerequisiti e Toolchain di sviluppo** **8**
2.1 Debugger . . . . . 9
2.2 Makefile . . . . . 9

**3 Organizzazione delle cartelle del sistema** **10**

**4 Caricamento Modulo I/O** **11**
4.1 Modifiche al Makefile . . . . . 11
4.2 Gestione file ELF e Mapping in memoria virtuale . . . . . 14
4.3 Modifiche all'heap di sistema . . . . . 17

**5 Inizializzazione Modulo I/O** **19**
5.1 Creazione Processo Main_IO . . . . . 19
5.2 Finestra sulla Memoria per il Sistema . . . . . 22
5.3 Configurazione PCI . . . . . 22

**6 VGA** **27**
6.1 Struttura . . . . . 27
6.1.1 Registri Interni . . . . . 27
6.1.2 Memoria Video . . . . . 30
6.2 Configurazione VGA . . . . . 35
6.2.1 Modalità Testo . . . . . 35
6.2.2 Modalità Grafica . . . . . 40
6.3 Namespace vid . . . . . 41
6.4 Funzioni di Supporto alla Modalità Testo . . . . . 43
6.5 Primitive I/O . . . . . 47

**7 Conclusioni e indicazioni per la continuazione del progetto** **48**

**8 Ringraziamenti** **49**

**9 Bibliografia** **52**

# Introduzione

RISC-V è un'architettura di processore con set di istruzioni ridotto: reduced instruction set computer (RISC). A differenza di altre architetture Complex Instruction Set Computing (CISC), come quelle usate da intel, risulta essere più efficiente e veloce, con maggiore personalizzazione e flessibilità. Avendo infatti meno istruzioni nell'ISA (instruction set architecture), il processore riesce a gestire meglio le risorse, eseguendo più velocemente le operazioni e risparmiando energia.
Questi vantaggi risultano fondamentali in campi odierni in via di sviluppo, quali: *IoT* (Internet of Things), *Intelligenza Artificiale*, *Robotica*, *Telefonia*. Nasce nel 2010 nell'Università della California - Berkeley, come standard completamente open-source. L'obiettivo dei ricercatori era creare una architettura libera da restrizioni proprietarie. Questa caratteristica lo rende perfetto per lo sviluppo di progetti in sedi Universitarie, inoltre permette di non pagare costi di licenze legati al suo utilizzo.

## 1.1 Caratteristiche RISC-V

RISC-V mette a disposizione sia una versione con spazio di indirizzamento a 32 bit, sia una versione a 64 bit. Alcune istruzioni possono essere personalizzate. Sono disponibili diverse estensioni per semplificare lo sviluppo di software: esse permettono di eseguire istruzioni non disponibili nel set base, per esempio moltiplicazioni e divisioni.
La memoria definisce *word* come 32 bits(4 bytes), *halfword* come 16 bits(2 bytes), *doubleword* come 64 bits(8 bytes) e *byte* come 8 bit(1 byte). I prefissi corrispondenti sono nell'ordine: w, h, d, b. Essa è suddivisa in aree di memoria che possono essere vuote, occupate da dispositivi di I/O oppure dedite alla memoria principale. I/O risulta quindi essere *memory mapped*, pertanto non tutta la memoria fisica è direttamente accessibile o scrivibile dal kernel.
Le istruzioni RISC sono progettate per essere eseguite in un singolo ciclo di CPU. L'interfacciamento con la memoria si basa su due istruzioni: *load* per caricare dalla memoria in un registro e *store* per salvare da un registro in memoria.
I registri sono in totale 32 per l'estensione base *integer*. Eccezione fatta per

le istruzioni che accedono la memoria, tutte le altre indirizzano solo registri. Il primo registro è lo *zero register*: le scritture verso di esso vengono ignorate, le letture restituiscono sempre zero. Per eseguire una `MOV` è possibile utilizzare l'istruzione `ADD` con lo *zero register* come uno dei due addendi. Altri registri comprendono: *return address register, stack poitner register*, scratch registers o registri temporanei (t0-t6), registri di salvataggio (a0-a7, il loro valore non viene sporcato dalle chiamate di funzione, sono utilizzati come parametri per le funzioni). Esistono infine molti registri di stato e di controllo, chiamati **CSR**, che consentono di interagire con la configurazione e con lo stato di esecuzione della CPU, accessibili tramite le istruzioni *csrr* per leggere e *csrw* per scrivere[1].
L'organizzazione di paginazione, cache e memoria virtuale è simile a x86, con un controllo sui permessi di accesso di tipo rwx (read, write, execute).
Esistono tre livelli di privilegio:

- Macchina
- Supervisore
- Utente

Inizialmente il nucleo parte in modalità macchina per eseguire il boot, successivamente passa a modalità supervisore.
Le interruzioni di sistema vengono alzate tramite istruzione *ecall*: sia queste, sia quelle esterne, sia le eccezioni vengono tutte gestite dalla PLIC.
Lo spazio di indirizzamento della memoria offerto da QEMU [5] prevede la suddivisione degli indirizzi per diversi dispositivi, per esempio: PCI, dispositivo seriale UART, dispositivi virtuali VIRTIO, il PLIC ed altro.
Il formato è rappresentato dalla Tabella 1. I campi con TBD(To Be Defined) indicano una dimensione che viene calcolata runtime da QEMU. La dimensione della DRAM, nel nostro caso 128MiB, è invece specificata nel makefile con l'opzione `-m 128M`

[1]Sono presenti anche istruzioni per il *set* ed il *clear* di specifici bit: *csrs* e *csrc* rispettivamente.

| **NOME** | **INDIRIZZO BASE** | **DIMENSIONE** |
|---|---|---|
| VIRT_DEBUG | 0x0 | 0x100 |
| VIRT_MROM | 0x1000 | 0xf000 |
| VIRT_TEST | 0x100000 | 0x1000 |
| VIRT_RTC | 0x101000 | 0x1000 |
| VIRT_CLINT | 0x2000000 | 0x10000 |
| VIRT_ACLINT_SSWI | 0x2f00000 | 0x4000 |
| VIRT_PCIE_PIO | 0x3000000 | 0x10000 |
| VIRT_PLATFORM_ BUS | 0x4000000 | 0x2000000 |
| VIRT_PLIC | 0xc000000 | TBD |
| VIRT_APLIC_M | 0xc000000 | TBD |
| VIRT_APLIC_S | 0xd000000 | TBD |
| VIRT_UART0 | 0x10000000 | 0x100 |
| VIRT_VIRTIO | 0x10001000 | 0x1000 |
| VIRT_FW_CFG | 0x10100000 | 0x18 |
| VIRT_FLASH | 0x20000000 | 0x4000000 |
| VIRT_IMSIC_M | 0x24000000 | TBD |
| VIRT_IMSIC_S | 0x28000000 | TBD |
| VIRT_PCIE_ECAM | 0x30000000 | 0x10000000 |
| VIRT_PCIE_MMIO | 0x40000000 | 0x40000000 |
| VIRT_DRAM | 0x80000000 | 0x0 |

Tabella 1: Spazio di indirizzamento di memoria

## 1.2 Obiettivi

L'obiettivo di questa tesi è di continuare a migrare il nucleo didattico utilizzato nell'esame di Calcolatori Elettronici da x86 ad architettura RISC-V. Tale lavoro, che si basa sul precedente operato dei colleghi Edoardo Geraci, Andrea Bedini, Chiara Panattoni e Franscesco Barcherini, prevede inizialmente di realizzare il caricamento in memoria di un modulo I/O, collegato separatamente dal kernel ed eseguito a livello supervisore. Successivamente si inizializza il modulo I/O, in particolare lo spazio di configurazione PCI dei dispositivi: per esempio la VGA. Infine si configurano la modalità testo e la modalità grafica della VGA, abilitandola ad operare correttamente.

## 1.3 Nota

Durante questa esposizione, saranno fatti riferimenti ad indirizzi e/o valori definiti nel file header `include/costanti.h`
Molte funzioni che seguiranno utilizzano costanti definite in questo file, per assegnare valori alle variabili. Si consiglia dunque di leggere questo file quando si incontrano tali costanti.

# Prerequisiti e Toolchain di sviluppo

Per poter compilare ed eseguire il codice del nucleo è necessario installare una toolchain per RISC-V a 64 bit. In questa tesi si fa riferimento alla toolchain GNU[3]. Il pacchetto può essere installato tramite il package manager della propria distro Linux (apt, yum, aur, etc...) sotto uno dei seguenti nomi:

- riscv64-unknown-elf-gcc
- riscv-gnu-toolchain-bin
- gcc-riscv64-unknown-elf

Utilizzando, per esempio, l'ultima versione di Debian, è possibile installare la toolchain tramite il comando

```
sudo apt install gcc-riscv64-unknown-elf
```

Successivamente è necessario installare un ambiente adatto all'esecuzione di codice RISC-V a 64 bit: usiamo l'emulatore QEMU [2] installando l'architettura RISC-V 64 con il comando

```
sudo apt install qemu-system-riscv64
```

Per utilizzare la toolchain potrebbe essere necessario rendere visibili i binari alla shell, aggiungendo alla variabile di ambiente PATH la cartella d'installazione. Il percorso da impostare, di default, risulta in genere essere

```
export PATH="/opt/riscv/bin:$PATH"
```

## 2.1 Debugger

È possibile effettuare il debug del codice prodotto dalla toolchain tramite l'apposita versione di gdb, chiamata riscv64-unknown-elf-gdb. Come per x86, anche questa versione di gdb è compatibile con estensioni come gef. Per debuggare un programma, dunque, è sufficiente avviare QEMU e digitare il comando

```
riscv64-unknown-elf-gdb [executable-file]
```

## 2.2 Makefile

Nella cartella root del progetto [1] è presente il Makefile, in cui sono definite le regole di compilazione del sistema. Una volta installati gli strumenti necessari è possibile eseguire il comando make sui seguenti target:

- `make [compile]`
  compila il codice;
- `make run`
  compila ed esegue il codice nell'emulatore QEMU;
- `make debug`
  compila ed esegue il codice con il flag -S, esponendo l'emulatore per il debug tramite gdb sulla porta TCP 1234;[2]
- `make libce`
  compila la libreria libCE con funzioni di utilità per il codice;
- `make clean`
  rimuove i file oggetto e gli eseguibili.

[2]Per debuggare, dopo aver eseguito `make debug`, occorre aprire un nuovo terminale, eseguire gdb sullo stesso file eseguibile passato a QEMU e connettersi all'emulatore tramite il comando `target remote localhost:1234`

## Organizzazione delle cartelle del sistema

| Cartella | Descrizione |
|---|---|
| /doc | Contiene le relazioni e le slide inerenti al progetto. |
| /include | Contiene i file header. |
| /kernel | Contiene i file relativi al bootloader e al modulo sistema. |
| /libCE | Contiene la libreria utilizzata dal nucleo, completamente migrata a RISC-V. |
| /objs | Contiene i file oggetto generati durante la compilazione di libCE, modulo sistema, modulo I/O e modulo utente. |
| /io | Contiene i file relativi all'implementazione del modulo I/O. |
| /user | Contiene i file relativi all'implementazione del modulo utente. |
| /build | Contiene gli eseguibili. |

# Caricamento Modulo I/O

Il modulo I/O è un modulo di supporto al nucleo, distinto da quelli sistema ed utente: il suo scopo principale è di realizzare le primitive per l'accesso alle periferiche, gestendo le richieste di interruzione tramite processi detti processi "esterni", in quanto esterni al modulo sistema.
Il modulo si trova nei file all'interno della cartella */io* : `io.cpp` e `io.s`
Il makefile si occupa della loro compilazione e del loro collegamento, producendo il file `build/io.strip`, il quale verrà caricato inizialmente in memoria all'avvio del sistema e successivamente rimappato dal sistema nello spazio di indirizzamento di tutti i processi, all'interno dello spazio I/O condiviso.
Il modulo I/O è eseguito in modalità supervisore e gira ad interruzioni abilitate, permettendo così ai processi esterni di interrompersi. Esso può eseguire chiamate alle primitive di sistema, in particolare ha a disposizione primitive esclusive: il modulo utente non può invocarle perché necessario il livello supervisore.
Le primitive che il modulo I/O mette a disposizione invocano l'istruzione `ecall` dal file `io.s`
Tale istruzione permette di lanciare una interruzione software. Essa verrà gestita dal trap handler interno al modulo sistema. Dopo aver individuato la causa dell'interruzione mediante il registro di sistema *scause*, il trap handler invocherà il corpo della primitiva all'interno del file `io.cpp`
Il modulo utente usufruisce di queste primitive per accedere alle periferiche.
Il modulo I/O, dopo essere stato compilato e collegato, produce un file ELF che deve essere caricato in memoria. Il modulo sistema si occuperà poi di interpretare questo file ELF per trovare le sezioni .text, .data e .bss e caricarle nella memoria virtuale. Il file ELF specifica anche l'indirizzo a cui caricarle nella memoria virtuale.

## 4.1 Modifiche al Makefile

Il Makefile [6] era stato impostato, dai colleghi che precedentemente hanno lavorato a questo progetto, per compilare, collegare, caricare ed eseguire i moduli sistema ed utente. Per caricare correttamente anche il modulo I/O

sono sorti dei problemi. Inizialmente i moduli sistema ed utente venivano caricati in memoria con la seguente regola

```
1 run: $B/kernel $B/user.strip
2   $(RUN) -kernel $B/kernel -initrd $B/user.strip
```

Regola 1

Nella linea 1 si definisce il target per eseguire il nucleo e si dichiarano le dipendenze per creare il target, in questo caso il modulo sistema ed il modulo utente. Nella linea 2 è presente l'azione da svolgere, cioè il comando da eseguire.
`$(RUN)` è una variabile contenente il comando `qemu-system-riscv64` ed i relativi flags per eseguire il nucleo; $B è una variabile che viene espansa nella cartella build, dove sono presenti gli eseguibili; l'opzione `-kernel` carica l'immagine del kernel nel primo indirizzo di memoria fisica disponibile su QEMU: `0x80000000`; l'opzione `-initrd` carica invece il file ELF del modulo utente a metà della memoria fisica disponibile, nel nostro caso dunque viene caricato a `0x84000000`[3].
Per poter effettuare il caricamento di più moduli esterni al kernel è necessario operare in *multiboot*: l'ambiente di lavoro che utilizziamo però non mette a disposizione lo standard *multiboot*, non è quindi possibile caricare insieme più moduli tramite l'opzione `-initrd` .
Il problema è stato risolto creando un unico file da caricare con l'istruzione precedentemente citata: il modulo I/O ed il modulo utente sono stati concatenati in un unico file `moduli.strip`, aggiungendo in testa un piccolo header di riferimento, contenente dati essenziali per il sistema, quali la dimensione in byte di ciascun modulo.

```
1 $B/moduli.strip: $B/io.strip $B/user.strip
2     (printf "%d\n\%d\n" $$(stat -c%s $B/io.strip) $$(stat -c%s \
3     $B/user.strip); cat $B/io.strip $B/user.strip) > $@
4
5 run: build $B/kernel $B/moduli.strip
6     $(RUN) -kernel $B/kernel -initrd $B/moduli.strip
```

Regola 2

[3]In realtà il file potrebbe essere caricato una pagina antecedente questo indirizzo per eventuali header, quindi all'indirizzo `0x83FFF000`

Seguendo il formato precedentemente spiegato per le regole del makefile, si definiscono le regole per creare correttamente il file `moduli.strip` e per eseguire il nucleo.
Per quanto riguarda la prima regola:
Il comando `cat $B/io.strip $B/user.strip) > $@` concatena i due moduli I/O ed utente in un unico file, reindirizzando l'output nel file `$B/moduli.strip`, poiché `$@` è una variabile automatica che rappresenta il target: in questo caso `$B/moduli.strip`
Il comando `printf` stampa una stringa seguendo il formato specificato `"%d\n%d\n"` : numero decimale + line feed + numero decimale + line feed. Le specifiche di formato vengono sostituite dagli argomenti che seguono la stringa di formato: viene utilizzato il prefisso `$$` per ricorrere alla doppia espansione ed evitare che il makefile espanda `$` in una variabile interna ad esso, in questo caso NULL perché risulterebbe non definita.
Il comando `stat -c%s [file]` restituisce la dimensione in byte di `file`: si passano perciò alla printf le dimensioni dei moduli I/O ed utente, creando così una stringa che svolge la funzione di header per il file `moduli.strip`
I caratteri di line feed sono utilizzati dal sistema per facilitare il prelievo ed il calcolo della dimensione dei file.
La seconda regola invece richiama Regola 1: abbiamo ovviato al problema di non potere caricare separatamente i due file ELF dei moduli I/O ed utente concatenandoli in un unico file. La dipendenza `build` fa riferimento ad una regola che crea la cartella `build` nel caso questa fosse assente.

È di particolare interesse la regola che collega tutti i file del modulo I/O

```
1 START_IO = 0x0000010000001000
2
3 $(B)/io: objs/io_s.o objs/io_cpp.o $(HEADERS) $B/libce.a
4     $(LD) $(LDFLAGS) -Ttext $(START_IO) -o $@  objs/io_s.o \
      objs/io_cpp.o $(LDLIBS)
```

Regola 3

Il collegatore collega i file oggetto `io_s.o io_cpp.o` e le librerie, viene però specificato l'indirizzo in cui caricare la sezione .text mediante l'opzione `-Ttext`
Questo indirizzo di collegamento, definito alla linea 1, rappresenta la seconda pagina della zona di memoria virtuale assegnata allo spazio I/O condiviso:

un range di indirizzi virtuali con traduzione comune a tutti i processi per accedere al modulo I/O. Il file ELF avrà così questo indirizzo di partenza nella tabella delle sezioni come indirizzo virtuale e fisico[4] per la sezione .text Il modulo sistema potrà così utilizzare questi indirizzi virtuali per mappare nella memoria virtuale il modulo I/O.

## 4.2 Gestione file ELF e Mapping in memoria virtuale

Una volta caricati correttamente i file ELF dei moduli I/O ed utente all'interno della memoria, è necessario mapparli nella memoria virtuale del sistema. Per fare ciò viene utilizzata la funzione `carica_modulo` definita nel file `kernel/boot_main.cpp`
Il mapping nella memoria virtuale avviene sfruttando l'indirizzo della memoria virtuale dato al collegatore, tramite l'opzione `-Ttext` per ciascun modulo. Il sistema può infatti recuperare questi indirizzi virtuali ed utilizzarli come indirizzi di partenza nella funzione `map`. In questo modo il modulo I/O apparterrà alla zona I/O condiviso, comune a tutti i processi, come volevamo.
Questa funzione, oltre a mappare nella memoria virtuale, si occupa anche di rilocare i file ELF. Copia il contenuto delle sezioni da caricare in memoria all'interno dei frame liberi del sistema, così che il sistema sappia quali frame sono occupati dai moduli.
Carica modulo opera con un puntatore a file ELF, è perciò necessario che gli ELF siano allineati alla pagina. A causa dell'header inserito in testa al file `build/moduli.strip` i file ELF non risultano allineati.
Inoltre è presente un altro problema: come già anticipato nelle modifiche al makefile 4.1, il file `build/moduli.strip` viene caricato di default all'indirizzo `0x84000000`; questo indirizzo appartiene alla zona di memoria fisica libera, cui appartengono i frame liberi di M2[5]. Nel mentre che `carica_modulo` riloca le sezioni di interesse, potremmo rischiare che venga allocato un frame in cui risiedono sezioni che devono essere ancora rilocate.
Per risolvere entrambi i problemi si è scelto di effettuare uno spostamento dei soli file ELF all'interno di M1, partendo dal primo indirizzo libero, allineato alla pagina, dalla fine del modulo sistema: `end` è l'ultimo indirizzo fisico del modulo sistema ed è definito dal collegatore. Per fare ciò si utilizza la

[4]Solitamente gli indirizzi fisici e virtuali che vengono passati al caricatore sono equivalenti di default

[5]M1 è la zona di memoria dedicata al sistema, M2 è il resto della memoria fisica

funzione `sposta_ELF_moduli`.
Rimuovendo l'header che precedeva i file ELF e copiando interamente i file ELF, partendo da indirizzi di base allineati, risolviamo il primo problema.
Il secondo problema si risolve chiamando la funzione `sposta_ELF_moduli` prima di inizializzare i frame liberi di M2 e modificando l'indirizzo base di M2, il quale deve corrispondere al primo indirizzo libero, allineato alla pagina, che segue il secondo file ELF(vedi funzione `init_frame`). Così facendo non rischiamo che i frame liberi assegnati alla rilocazione di `carica_modulo` sovrascrivino i file ELF, che appartengono ora alla zona di memoria M1 e non più M2.

```
1  // FILE kernel/boot_main.cpp
2
3    char* puntatore_moduli = (char*) MOD_START;
4    natq lunghezza_header=0;
5    natq IO_size = 0;
6    natq user_size = 0;
7    natq exp = 1;
8    natq i=0;
9
10   //calcolo la dimensione in byte del modulo I/O
11   for(; *(puntatore_moduli+lunghezza_header) != '\n';
       lunghezza_header++){
12     exp *= 10;
13   }
14   exp/=10;
15
16   //converto la stringa in numero decimale
17   for(;i<lunghezza_header;i++){
18     //48 equivale all'ASCII code del carattere 0
19     natq conversione = *(puntatore_moduli+i)-48;
20     IO_size += conversione*exp;
21     exp/=10;
22   }
23   flog(LOG_DEBUG, "IO_size: %d byte", IO_size);
24
25   exp=1;
26   lunghezza_header++;
27   i=lunghezza_header;
28   //calcolo la dimensione in byte del modulo utente
29   for(; *(puntatore_moduli+lunghezza_header) != '\n';
       lunghezza_header++){
30     exp *= 10;
```

```
31  }
32  exp/=10;
33  //converto la stringa in numero decimale
34  for(;i<lunghezza_header;i++){
35    //48 equivale all'ASCII code del carattere 0
36    natq conversione = *(puntatore_moduli+i)-48;
37    user_size += conversione*exp;
38    exp/=10;
39  }
40  lunghezza_header++;
41
42  flog(LOG_DEBUG, "user_size: %d byte", user_size);
43
44  // Primo frame libero dopo il modulo sistema
45  natq moduli_base = allinea(reinterpret_cast<paddr>(&end),
     DIM_PAGINA);
46
47
48  start_io = moduli_base;
49  flog(LOG_DEBUG, "new start io %p",start_io);
50
51  start_user = allinea(reinterpret_cast<paddr>(moduli_base +
     IO_size),DIM_PAGINA);
52  flog(LOG_DEBUG, "new start user %p",start_user);
53
54  start_M2 = allinea(reinterpret_cast<paddr>(start_user +
     user_size),DIM_PAGINA);
55  flog(LOG_DEBUG, "new start m2 %p",start_M2);
56
57  // Allineo entrambi gli elf alla pagina durante la copia
58  memcpy((void*) start_io,(void*)(puntatore_moduli+
     lunghezza_header),IO_size);
59  memcpy((void*) start_user,(void*)(puntatore_moduli+
     lunghezza_header+IO_size),user_size);
60  return;
```

Codice 1: Corpo della funzione `sposta_ELF_moduli`

Alla riga `11` e alla riga `29` si conta la lunghezza della stringa che rappresenta la dimensione dei file ELF dei moduli I/O ed utente rispettivamente. Si utilizza il carattere di line feed `\n` per individuare la fine della stringa, così da facilitare il prelievo.
Alle righe `17` e `34` vengono prelevati i char dall'header per convertirli in numero decimale, così da ottenere la dimensione dei due ELF.

Alla riga 45 si definisce l'indirizzo di partenza da cui copiare gli ELF. La funzione `allinea` ritorna il primo indirizzo allineato alla pagina partendo da `end`.
Il primo ELF del file `build/moduli.strip` è quello del modulo I/O. Si copia dunque questo file a partire da `moduli_base`.
Il file ELF del modulo utente viene invece copiato a partire dal primo indirizzo, allineato alla pagina, dalla fine dell'ELF del modulo I/O. Importante per avere anche questo file ELF allineato.
La copia viene completata dalla funzione `memcpy` alle righe `58` e `59`, utilizzando le dimensioni dei due file, calcolate precedentemente alle righe `17` e `34`, come numero di byte da copiare.
Da notare la linea `54` dove viene definita la variabile `start_M2`: primo indirizzo libero allineato alla pagina, partendo dalla fine del file ELF modulo utente. Questo indirizzo viene utilizzato dalla funzione `init_frame` per inizializzare i frame liberi di M2.

```
1  // FILE kernel/vm.cpp
2
3    // primo frame di M2
4    paddr fine_M1 = start_M2;
5    // numero di frame in M1 e indice di f in vdf
6    N_M1 = (fine_M1 - START_DRAM) / DIM_PAGINA;
7    // numero di frame in M2
8    N_M2 = N_FRAME - N_M1;
9
10   if (!N_M2)
11     return;
12
13   primo_frame_libero = N_M1;
```

Codice 2: Parte della funzione `init_frame`

## 4.3 Modifiche all'heap di sistema

Dopo che le sezioni da caricare sono state mappate nella memoria virtuale del sistema e rilocate nei frame liberi, le due copie dei file ELF non servono più. Lo spazio occupato dalle copie dei file ELF viene quindi riutilizzato per l'heap di sistema

```
1 // FILE kernel/boot_main.cpp
2
3     heap_start = allinea(reinterpret_cast<void*>(&__heap_start),
      DIM_PAGINA);
4
5     heap_init(heap_start, HEAP_SIZE+(start_M2 - start_io));
```

Codice 3: Ridimensionamento dell'heap di sistema

Alla linea 5 si aggiunge alla costante `HEAP_SIZE` la dimensione della memoria occupata per la copia dei file ELF.
`__heap_start` viene fornito dal collegatore.

```
INF     ?       Running in S-Mode
INF     ?       Inizializzazione VGA in corso
INF     ?       VGA inizializzata
INF     ?       KBD inizializzata
INF     ?       PCI initialized
INF     ?       PLIC Initialized
INF     ?       Spostamento ed allineamento dei file ELF modulo I/O e utente
DBG     ?       IO_size: 17648 byte
DBG     ?       user_size: 10216 byte
DBG     ?       new start io 0000000080138000
DBG     ?       new start user 00000008013d000
DBG     ?       new start m2 0000000080140000
INF     ?       Numero di frame: 320 (M1) 32448 (M2)
INF     ?       Allocata tabella root
INF     ?       Creata finestra sulla memoria centrale:  [0000000000001000, 0000000088000000)
INF     ?       Attivata paginazione
INF     -       Nucleo di Calcolatori Elettronici - RISC-V
INF     -       Heap del modulo sistema: [0000000080038000, 0000000080140000)
INF     -       Suddivisione della memoria virtuale:
INF     -       - sis/cond [0000000000000000, 0000008000000000)
INF     -       - sis/priv [0000008000000000, 0000010000000000)
INF     -       - io /cond [0000010000000000, 0000018000000000)
INF     -       - usr/cond [ffff800000000000, ffffc00000000000)
INF     -       - usr/priv [ffffc00000000000, 0000000000000000)
INF     -       mappo il modulo I/O:
INF     -        - segmento sistema read-only  mappato a [0000010000000000, 0000010000004000)
INF     -        - segmento sistema read/write mappato a [0000010000004000, 0000010000005000)
INF     -        - heap:                                 [0000010000005000, 0000010000105000)
INF     -        - entry point: 0x0000010000001000
INF     -       mappo il modulo utente:
INF     -        - segmento utente  read-only  mappato a [ffff800000000000, ffff800000003000)
INF     -        - segmento utente  read/write mappato a [ffff800000003000, ffff800000004000)
INF     -        - heap:                                 [ffff800000004000, ffff800000104000)
INF     -        - entry point: 0xffff800000001000
```

Figura 1: Output da terminale per l'allocazione dell'heap e il caricamento dei moduli

# Inizializzazione Modulo I/O

Durante il boot ci si occupa anche dell'inizializzazione del modulo I/O: vengono inizializzati heap e periferiche. Per fare ciò, si crea il processo `main_IO`: si passa alla primitiva `activate_p` l'indirizzo di partenza del modulo I/O, dopo essere stato mappato nella memoria virtuale da `carica_modulo`. Quest'ultima funzione, infatti, restituisce l'indirizzo di partenza del modulo, dopo averlo rilocato.

```
1  // FILE kernel/boot_main.cpp
2
3      vaddr carica_IO(paddr root_tab){
4       flog(LOG_INFO, "mappo il modulo I/O:");
5       return carica_modulo(start_io, root_tab, 0, DIM_IO_HEAP);
6      }
7
8      bool crea_spazio_condiviso(paddr root_tab){
9       io_entry = ptr_cast<void(natq)>(carica_IO(root_tab));
10      if (!io_entry)
11         return false;
12      user_entry = ptr_cast<void(natq)>(carica_utente(root_tab));
13      if (!user_entry)
14         return false;
15
16      return true;
17     }
```

Codice 4: Corpo delle funzioni `carica_IO` e `crea_spazio_condiviso`

## 5.1 Creazione Processo Main_IO

Il processo `main_IO` viene attivato dalla primitiva `activate_p`. Per far sapere al sistema quando l'inizializzazione del modulo I/O è terminata, si utilizza un semaforo di sincronizzazione: il processo `main_sistema` si bloc-

ca in attesa della fine dell'inizializzazione.

```
1  // FILE kernel/boot_main.cpp
2
3  void main_sistema(natq){
4
5    ...
6
7    flog(LOG_INFO,"Creo il processo main I/O");
8    natl sync_io = sem_ini(0);
9    if (sync_io == 0xFFFFFFFF) {
10       flog(LOG_ERR, "impossibile allocare il semaforo di sincr
      per I/O");
11       goto error;
12   }
13
14   id = activate_p(io_entry, sync_io, MAX_EXT_PRIO, LIV_SISTEMA);
15   if (id == 0xFFFFFFFF) {
16       flog(LOG_ERR, "impossibile creare il processo main I/O");
17       goto error;
18   }
19   flog(LOG_INFO,"Attendo inizializzazione modulo I/O");
20
21   sem_wait(sync_io);
22
23   ...
24 }
```

Codice 5: creazione processo `main_IO`

Alla linea `8` si inizializza il semaforo di sincronizzazione.
Alla linea `14` si attiva il processo `main_IO`.
Alla linea `21` si attende la fine dell'inizializzazione, bloccandosi sul semaforo di sincronizzazione.

Quando il processo `main_IO` passerà in esecuzione, si occuperà di inizializzare heap e periferiche. Sbloccherà poi il processo `main_sistema`, chiamando la primitiva `sem_signal` sul semaforo di sincronizzazione, ed infine terminerà.

```
1  // FILE io/io.cpp
2
3  extern "C" void main(natq sem_io) {
4
```

```
5       //inizializzazione semaforo mutua esclusione per heap
6       ioheap_mutex = sem_ini(1);
7       if(ioheap_mutex == 0xFFFFFFFF){
8           panic("Impossibile creare semaforo ioheap_mutex");
9       }
10
11      //inizializzazione heap modulo I/O
12      natb* end_ = allinea(end,DIM_PAGINA);
13      heap_init(allinea_ptr(end_, DIM_PAGINA), DIM_IO_HEAP);
14
15      //inizializzazione periferiche
16      flog(LOG_INFO,"Inizializzo la console (kbd + video)");
17      if(!console_init()){
18          panic("Inizializzazione console fallita");
19      }
20
21      flog(LOG_INFO,"Inizializzazione modulo I/O completata");
22
23      sem_signal(sem_io);
24      terminate_p();
25  }
```

Codice 6: Processo `main_IO`

Alla linea `6` si inizializza il semaforo di mutua esclusione per accedere allo heap del modulo I/O: l'heap I/O è una risorsa condivisa tra più processi, le interruzioni sono però abilitate nel modulo I/O e perciò risulta necessario garantire la mutua esclusione.
Alla linea `12` si inizializza l'heap del modulo I/O. L'indirizzo di partenza è il primo indirizzo virtuale, allineato alla pagina, dalla fine del modulo I/O.
Alla linea `16` si inizializza la console: scheda video e tastiera. La scheda video usata dal sistema è la VGA. Essa viene inizializzata pulendo lo schermo, impostando il cursore sulla prima cella e scegliendo l'attributo per i caratteri: scritte bianche su sfondo nero di default.
Alla linea `23` si sblocca il processo `main_sistema` e infine si termina il processo `main_IO`.

## 5.2 Finestra sulla Memoria per il Sistema

La finestra sulla memoria non è altro che una traduzione identità da indirizzo virtuale a fisico: è necessaria al sistema per poter accedere allo spazio di indirizzamento di memoria, in particolare tutta la memoria fisica. Appartiene allo spazio virtuale sistema condiviso e perciò tutti i processi posseggono una copia della finestra nel loro albero di traduzione: il sistema può così utilizzare l'albero di traduzione del processo che era attualmente in esecuzione senza dover disattivare la MMU per accedere ad indirizzi in memoria.
Vengono quindi mappati in sé stessi tutti gli indirizzi, utili al sistema, da `0x0` a `0x800000000+[dimensione DRAM]`. Nel nostro caso sono di interesse per il sistema le zone di memoria dedicate a UART, PLIC e DRAM, già mappate dai colleghi, VIRT_TEST, PCIe-ECAM e PCIe-MMIO.
PCIe-ECAM ed il PCIe-MMIO sono indispensabili per poter interagire con le periferiche ed inizializzarle. Mappiamo dunque le aree di memoria dedicate a questi due dispositivi, aggiungendoli alla finestra sulla memoria.

```
1  // FILE kernel/vm.cpp
2
3    //Mappa PCIe-ECAM
4    if(map(root_tab, PCI_ECAM, PCI_ECAM+PCI_ECAM_SIZE, BIT_W |
       BIT_R | BIT_G, identity_map,2) != (PCI_ECAM+PCI_ECAM_SIZE)){
5      return false;
6    }
7    //Mappa PCIe-MMIO
8    if(map(root_tab, PCI_MMIO, PCI_MMIO+PCI_MMIO_SIZE, BIT_W |
       BIT_R | BIT_G, identity_map,2) != (PCI_MMIO+PCI_MMIO_SIZE)){
9      return false;
10   }
```

Codice 7: Mapping del PCI nella finestra sulla memoria

## 5.3 Configurazione PCI

Il nucleo fa uso dello standard PCI Express per comunicare con le periferiche. Per retrocompatibilità ci è possibile interfacciarsi a questo standard come faremmo con lo standard PCI [7]. Lo standard PCI mette a disposizione uno *spazio di configurazione* per poter accedere ai registri di configurazione delle periferiche. In particolare, poiché le periferiche sono Memory-Mapped nell'architettura RISC-V, lo spazio di configurazione si occupa anche di mappare

sia le risorse sia i registri interni delle periferiche nello spazio di indirizzamento di memoria.
QEMU mette a disposizione i dispositivi PCIe_ECAM[6] e PCIe_MMIO[7] nel suo *spazio di indirizzamento*: il primo è lo spazio di configurazione PCI; il secondo è dove devono essere rimappate le risorse e i registri interni delle periferiche.
Durante la fase di boot, viene chiamata la funzione `pci_init`, che si occupa di inizializzare il PCI.

```
// FILE kernel/pci.cpp

extern "C" void pci_init(){

    // qemu -machine virt puts PCIe config space here.
    natl  *ecam = (natl *) PCI_ECAM;

    // look at each device on bus 0
    for(int dev = 0; dev < 32; dev++){
        int bus = 0;
        int func = 0;
        int offset = 0;
        natl off = (bus << 16) | (dev << 11) | (func << 8) |
                (offset);

        volatile natl *base = ecam + off;
        natl id = base[0];
        // struttura per configurare pci
        PCI_config *pointer = (PCI_config *)(ecam + off);

        if(id == 0x11111234){
        // PCI device ID 1111:1234 is VGA
        // VGA is at 00:01.0, using extended control registers
        // (4096 bytes)

        // tell the VGA to reveal its framebuffer at
        // physical address 0x50000000
        pointer->bar0 = VGA_FRAMEBUFFER;

        // tell the VGA to set up I/O ports at 0x40000000
        pointer->bar2 = VGA_MMIO_PORTS;
```

[6]PCI-Express Enhanced Configuration Access Mechanism
[7]PCI-Express Memory-Mapped I/O

```
32
33          // command and status register.
34          // bit 0 : I/O access enable
35          // bit 1 : memory access enable
36          // bit 2 : enable mastering
37          //abilitiamo accessi in memoria per il dispositivo
38          //bit 1 nel command register
39          pointer->command |= 0x2;
40
41          flog(LOG_INFO,"Inizializzazione VGA in corso");
42          vga_init();
43          }
44      }
45  }
```

Codice 8: Corpo funzione `pci_init`

Nella funzione si utilizza una *struttura dati* che rappresenta lo spazio di configurazione PCI, così da facilitare gli accessi ai registri di configurazione.
Alla linea `8` si effettua la ricerca dei dispositivi: quelli che ci interessano sono tutti sul bus 0. La periferica che vorremmo configurare è la VGA [8]. Questa ha `VENDOR ID 1234` e `DEVICE ID 1111`.
Alla linea `20` configuriamo la VGA seguendo le specifiche [8] : assegniamo l'indirizzo `VGA_FRAMEBUFFER = 0x50000000` alla memoria video[8] e l'indirizzo `VGA_MMIO_PORTS = 0x40000000` ai registri interni[9]. Entrambi questi indirizzi fanno parte del dispositivo PCIe-MMIO di QEMU.
Infine abilitiamo gli accessi in memoria per il dispositivo, poiché Memory-Mapped.
`pci_init` chiama anche la funzione `vga_init`, la quale si occupa di configurare i registri interni della VGA per renderla operativa. Di norma questo compito sarebbe assegnato al PCI-BIOS, il quale non è supportato da QEMU RISC-V. Risulta quindi essere compito nostro configurare correttamente la VGA per abilitare la modalità testo e la modalità grafica. Questa funzione verrà approfondita nella *prossima sezione*, dedicata alla VGA.

[8] Il BAR0 deve contenere l'indirizzo di partenza della memoria video
[9] Il BAR2 deve contenere l'indirizzo di partenza dello spazio dedicato ai registri interni

| Register | Offset | Bits 31-24 | Bits 23-16 | Bits 15-8 | Bits 7-0 |
|---|---|---|---|---|---|
| 0x0 | 0x0 | Device ID | | Vendor ID | |
| 0x1 | 0x4 | Status | | Command | |
| 0x2 | 0x8 | Class code | Subclass | Prog IF | Revision ID |
| 0x3 | 0xC | BIST | Header type | Latency Timer | Cache Line Size |
| 0x4 | 0x10 | Base address #0 (BAR0) | | | |
| 0x5 | 0x14 | Base address #1 (BAR1) | | | |
| 0x6 | 0x18 | Base address #2 (BAR2) | | | |
| 0x7 | 0x1C | Base address #3 (BAR3) | | | |
| 0x8 | 0x20 | Base address #4 (BAR4) | | | |
| 0x9 | 0x24 | Base address #5 (BAR5) | | | |
| 0xA | 0x28 | Cardbus CIS Pointer | | | |
| 0xB | 0x2C | Subsystem ID | | Subsystem Vendor ID | |
| 0xC | 0x30 | Expansion ROM base address | | | |
| 0xD | 0x34 | Reserved | | | Capabilities Pointer |
| 0xE | 0x38 | Reserved | | | |
| 0xF | 0x3C | Max latency | Min Grant | Interrupt PIN | Interrupt Line |

Figura 2: Spazio di configurazione PCI

```
// FILE include/pci.h

    struct PCI_config {
        natw vendor_id;
        natw device_id;
        natw command;
        natw status;
        natb revision_id;
        natb prog_if;
        natb subclass;
        natb class_code;
        natb cache_line_size;
        natb latency_timer;
        natb header_type;
        natb bist;
        natl bar0;
        natl bar1;
        natl bar2;
        natl bar3;
        natl bar4;
        natl bar5;
        natl cardbus_cis_pointer;
        natw subsystem_vendor_id;
        natw subsystem_id;
        natl expansion_rom_base_address;
        natb capabilities_pointer;
        natb padding[7];
        natb interrupt_line;
        natb interrupt_pin;
        natb min_grant;
        natb max_latency;
    };
```

Struttura Dati 1: Spazio configurazione PCI

# VGA

La scheda video emulata da QEMU è la Video Graphics Array (VGA) [9]. Essa ha una memoria interna, che deve essere mappata nello spazio di PCIe_MMIO, ed offre una serie di registri interni per poter essere configurata.
Le funzioni di configurazione, inizializzazione, scrittura in memoria video e nei registri interni sono tutte definite nel file `kernel/vga.cpp` e nella cartella `libCE`
La VGA offre due modalità di nostro interesse: *modalità testo* e *modalità grafica*.
La prima si occupa di dividere il display in celle, ciascuna cella può poi contenere un carattere da mostrare a video. È possibile scegliere il colore sia del carattere in foreground che dello sfondo, dietro il carattere, in background. È presente anche un cursore, lampeggiante, che tiene traccia di dove sarà scritto il prossimo carattere. La modalità testo che viene configurata nel nucleo è la modalità testo `80x25`.
La seconda modalità divide invece il display in pixel, permettendo di colorare ciascun pixel a nostro piacimento: seguendo una palette di default caricata nella VGA. La modalità grafica che viene configurata nel nucleo è la modalità grafica `320x200`.
In entrambe le modalità il primo numero indica il numero di colonne ed il secondo numero indica il numero di righe.

## 6.1 Struttura

La struttura della VGA comprende i registri interni e la memoria video. Nella sezione di *Configurazione PCI* abbiamo assegnato ad entrambi un indirizzo base.[10]

### 6.1.1 Registri Interni

*I registri interni* [10] sono generalmente delle porte indirizzate nello spazio di I/O nel range: `0x3C0-0x3DF`. Poiché la nostra architettura prevede il

[10]Per maggiori informazioni sulla struttura della VGA vedere [11].

Memory-Mapped I/O, questi registri devono essere mappati nello spazio di indirizzamento di memoria. Una volta assegnato al BAR2 l'indirizzo di base, i registri vengono rimappati nella regione scelta in modo 1:1[11], con offset dalla base: `0x400-0x41F`.

```
// FILE include/vid.h

    #define AC           0x400  //attribute or palette registers
    #define AC_READ      0x401  //attribute read register
    #define MISC         0x402  //miscellaneous register
    #define MISC_READ    0x40c  //miscellaneous read register
    #define SEQ          0x404  //sequencer
    #define SEQ_DATA     0x405  //sequencer data
    #define GC           0x40e  //graphic address register
    #define GC_DATA      0x40f  //graphic address register data
    #define CRTC         0x414  //cathode ray tube controller
                                //address register
    #define CRTC_DATA    0x415  //CRTC data
    #define PC           0x408  //PEL write index
    #define PD           0x409  //PEL write data

    #define INPUT_STATUS_REGISTER 0x41a //reset AC index mode
```

Offset dei registri interni VGA

La VGA ha in realtà più di 300 registri interni. Quelli mostrati in *figura* sono solo una piccola parte, ma permettono di accedere a tutti gli altri. Per semplificare l'accesso a tutti i registri, molti di questi vengono infatti indicizzati da altri: AttributeController, Sequencer, GraphicsController, CathodeRayTubeController e DAC register (PC) sono tutti registri che permettono di indicizzarne altri. Questi sono divisi principalmente in 2 registri: uno per indicizzare il registro interno di interesse ed uno per leggere/scrivere i dati. In particolare, il registro dei dati segue sempre quello di indicizzazione. L'AttributeController (AC) è una eccezione: le letture vengono eseguite come per gli altri, però le scritture vengono fatte nello stesso registro di indicizzazione. AC ricorda se la prossima operazione è di indice o di dati. Queste si alternano ad ogni operazione sul registro. Per imporre una operazione di indicizzazione su AC, bisogna leggere il registro `INPUT_STATUS_REGISTER`.
PC e PD svolgono la funzione di caricamento della palette: per velocizza-

[11] La prima porta 0x3C0 viene rimappata in: BAR2+0x400. Le altre seguono, nello stesso ordine, incrementando l'offset.

re l'operazione è possibile scrivere l'indice di partenza in PC per caricare il primo colore e poi eseguire un ciclo di scritture in PD nel formato RGB(Red-Green-Blue), per caricare tutti i colori. Queste terne definiscono ciascuna un colore. La palette che viene caricata in modalità testo si trova nell'array `text_palette` all'interno del file `include/palette.h`
Per le scritture in tutti questi registri è necessario prima scrivere l'indice del registro a cui si vuole accedere e poi il dato da scrivere.
Per leggere i registri, bisogna prima indicizzarli e poi leggere il contenuto dal registro dei dati.
Gli altri registri hanno accesso diretto e non svolgono la funzione di registri di indicizzazione. Per poterli leggere bisogna accedere al registro apposito di lettura.
Nel file `kernel/vga.cpp` sono definite le funzioni per le operazioni di lettura e scrittura sui registri interni della VGA:

- `natb readport(natl port, natb index)`
- `void writeport(natl port, natb index, natb val)`

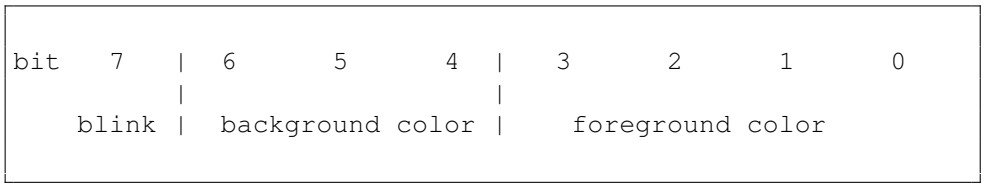
La prima permette di leggere il contenuto di un registro interno, specificando la porta `port` e l'indice `index` e ritornando il contenuto del registro. Se la porta non è un registro di indicizzazione, l'indice viene ignorato.
La seconda permette invece di scrivere il valore `val` nel registro interno di porta `port` ed indice `index`. Come per la lettura, anche qui l'indice viene ignorato se si sta scrivendo in un registro non di indicizzazione. Se si sta scrivendo nel registro PD, l'indice viene ignorato, per configurare l'indirizzo di partenza per la palette bisogna dunque scrivere anche nel registro PD: ciò permette di poter utilizzare questa funzione per scrivere le terne RGB nel registro PD senza dover modificare il registro PC.

### 6.1.2 Memoria Video

La memoria video della VGA è rimappata a partire dal BAR0 dello spazio di configurazione PCI. Chiamiamo questo indirizzo di base: `VGA_FRAMEBUFFER`.
In modalità testo, l'indirizzo di partenza del text buffer coincide con `VGA_FRAMEBUFFER`. Il font utilizzato è il *code page 737*.
Il text buffer è diviso in celle, ognuna formata da gruppi di 4 byte. In ordine abbiamo per ogni cella:

1 Character byte: contiene il codice del *code page 737* del carattere da mostrare a video nella cella.

2 Attribute byte: contiene il blink bit, il background color ed il foreground color. Il background color è il colore della cella. Il foreground color è il colore del carattere. Il byte è così organizzato:




Attribute byte

È possibile disattivare il blink bit ed avere 4 bit per il background color.

3 Byte ignorato.

4 Byte ignorato.

Figura 3: Font code page 737

Supponiamo ora di volere scrivere il carattere "A" nella prima cella ed il carattere "B" nella seconda cella. Entrambi in bianco su sfondo nero.
Sappiamo che nel *code page 737* il carattere "A" è codificato nel byte 0x65, mentre il carattere "B" è 0x66. L'attributo invece è 0x0F, da specifiche.
Per scrivere il carattere "A" in bianco su sfondo nero nella prima cella, sarà necessario scrivere 0x65 all'indirizzo `VGA_FRAMEBUFFER` e 0x0F all'indirizzo successivo `VGA_FRAMEBUFFER+1`, cioè l'Attribute Byte.
Per scrivere invece nella seconda cella il carattere "B" in bianco su sfondo nero, dovremo saltare i 2 byte che vengono ignorati e poi scrivere character e attribute byte. Rispettivamente 0x66 all'indirizzo `VGA_FRAMEBUFFER+4` e 0x0F all'indirizzo `VGA_FRAMEBUFFER+5`.
In sintesi, per accedere al character byte della cella numero N, sarà necessario scrivere il byte del carattere all'indirizzo `VGA_FRAMEBUFFER+4*N`. L'attribute byte sarà all'indirizzo successivo.

In modalità grafica, invece, lo schermo è diviso in pixel. Ogni byte della memoria video rappresenta un pixel. In ogni byte andrà inserito il codice della palette che identifica il colore di nostro interesse.
Supponiamo di voler colorare di verde le prime cinque colonne, le ultime cinque colonne e le due colonne centrali. Il resto del display invece di rosso. Ricordiamo di avere 320 colonne e 200 righe. Di seguito un esempio di codice per svolgere questa operazione ed il relativo output a schermo.

```
1  // Esempio
2
3      //indirizzo base del frame buffer
4      char* base = (char*) VGA_FRAMEBUFFER;
5
6      for(int i=0; i < 200;i++){
7
8        for(int j=0; j < 320;j++){
9
10         if(j<5 || j==159 || j==160 || j>314){
11             base[i*320 + j] = 0xfe;
12         }else{
13             base[i*320 + j] = 0xff;
14         }
15       }
16     }
```

Codice 9: Esempio modalità grafica


Figura 4: Output a video dell'*esempio* sull'utilizzo della modalità grafica

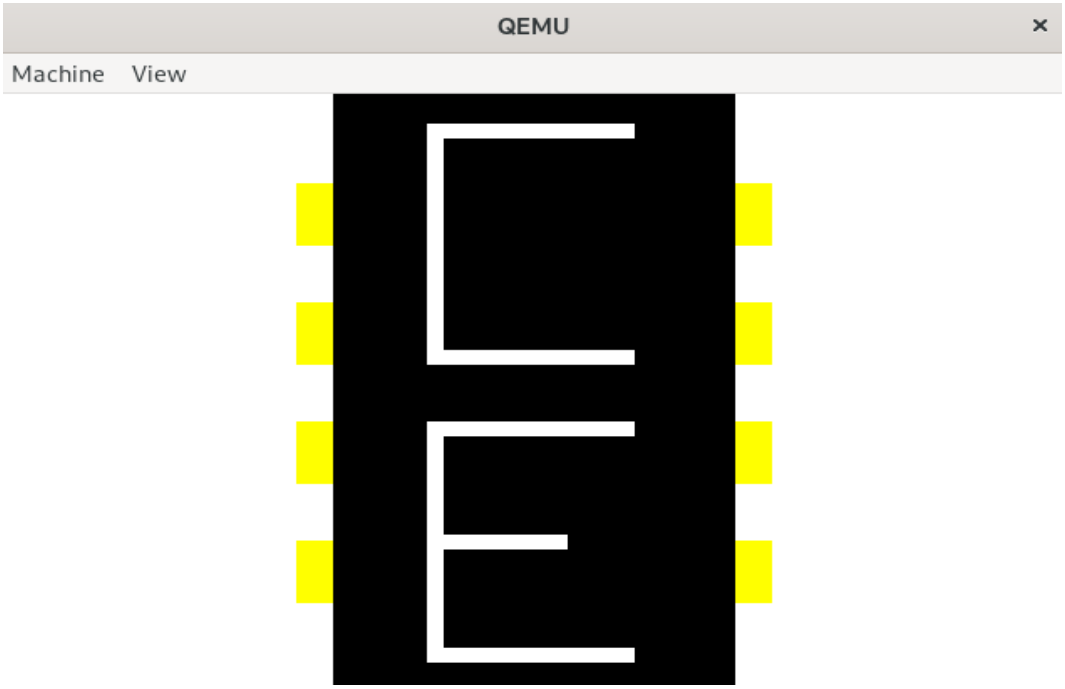



Figura 5: Esempio su un utilizzo più pratico della modalità grafica - logo del corso di Calcolatori Elettronici

## 6.2 Configurazione VGA

Per rendere operativa la VGA è indispensabile configurarla correttamente nella modalità di nostro interesse. Per fare ciò bisogna programmare i registri interni.
La funzione che si occupa della inizializzazione della VGA nella modalità voluta è `vga_init`, già introdotta nella sezione *Configurazione PCI*.

```
1  // FILE kernel/vga.cpp
2
3      extern "C" void vga_init() {
4          // text mode
5          init_textmode_80x25();
6
7
8          // graphic mode
9          // init_graphicmode_320x200();
10
11         flog(LOG_INFO,"VGA inizializzata");
12     }
```

Codice 10: Corpo funzione `vga_init`

La funzione inizializza di default la VGA in modalità testo `80x25`. Per inizializzare di default la modalità grafica `320x200`, basterà commentare la funzione `init_textmode_80x25` e procedere col togliere il commento alla funzione `init_graphicmode_320x200`.
Nelle prossime sezioni verranno entrambe analizzate nel dettaglio. Per i dati da utilizzare nella configurazione dei registri, vedere [10] e [12].

### 6.2.1 Modalità Testo

La funzione `init_textmode_80x25` programma tutti i registri interni, sfruttando una serie di chiamate di funzione per ciascun registro interno e/o di indicizzazione.
Queste funzioni sono tutte definite nel file `kernel/vga.cpp`
Questa configurazione permette di dividere lo schermo in celle: 80 celle per riga e 25 celle per colonna. Ogni cella è dedita a contenere un carattere ed è composta da 9 colonne di pixel e 16 righe di pixel: 9x16. In particolare, la nona colonna è utilizzata come colonna vuota, per creare separazione tra

due caratteri in celle contigue. Il font scelto è quindi un font 8x16, poiché la nona colonna è di default vuota. Tale font prevede un byte per ciascuna riga della cella: abbiamo quindi 16 byte per ogni carattere, uno per riga. I bit interni a ciascun byte indicano invece i pixel delle colonne della cella: un bit a 1 equivale al pixel colorato con foreground color, un bit a 0 equivale al pixel colorato con background color. Ogni carattere è dunque una serie di 16 byte.
Il font dei caratteri è salvato nel file `include/font.h` all'interno dell'array `font_16`[12]. Nello specifico, il font utilizzato è il *code page 737*.

```
1  // FILE kernel/vga.cpp
2
3      void init_textmode_80x25(){
4
5        writeport(MISC, 0x00, 0x67);
6
7        init_SEQ();
8        init_CRTC();
9        init_GC();
10       init_AC();
11       init_PD();
12
13       load_font(font_16);
14     }
```

Codice 11: Corpo funzione `init_textmode_80x25`

Alla linea 13 viene caricato il font nella memoria interna della VGA, nel seguente modo

```
1  // FILE kernel/vga.cpp
2
3  void load_font(unsigned char* font_16){
4
5      //...
6
7      //configurazione registri interni per caricare il font
8
9      //...
10
11     volatile void *vga_buf = (void*)(VGA_FRAMEBUFFER+
12                                      0xe000*4+2);
```

[12] 16 indica il numero di byte per carattere, dunque il numero di righe.

```
13      natl padding = 0;
14
15      for (natl i = 0; i < 256; i++) {
16          for(natl j = 0; j < 16; j++){
17              memcpy((void*)(vga_buf + 64*i + 64*padding + 4*j),
18                      (void*)(font_16+16 * i+j), 1);
19          }
20          padding++;
21      }
22
23      //...
24
25      //rigenerazione registri interni
26
27      //...
28  }
```

Codice 12: Parte della funzione `load_font`

Analizzando i file sorgenti QEMU [13] si ricava il modo in cui deve essere caricato il font nella memoria interna della VGA.
L'indirizzo base da cui deve essere caricato è dato da `VGA_FRAMEBUFFER + offset`, dove `offset` è indicato dal `sequencer register index 03h` moltiplicato per `4`: la regione da noi scelta nel registro parte da `E000h`.
Alla linea 11 è calcolato l'indirizzo di partenza: è presente anche un padding di 2 byte.
Il modo in cui deve essere caricato risulta poco intuitivo: si divide la memoria in *aree* da 64 byte, ciascun carattere occupa una di queste *aree*. Tra un carattere ed un altro deve esserci una *area* di 64 byte vuota. I 16 byte che identificano il carattere devono essere caricati dall'indirizzo di partenza dell'*area*, a 4 byte di distanza l'uno dall'altro. Alla linea 17 viene eseguita la copia nel modo spiegato.
Vedere *la spiegazione grafica* per chiarimenti.
Il font così caricato risulta utilizzabile. Un esempio dell'output a video di tutto il font caricato è descritto nella Figura 7.

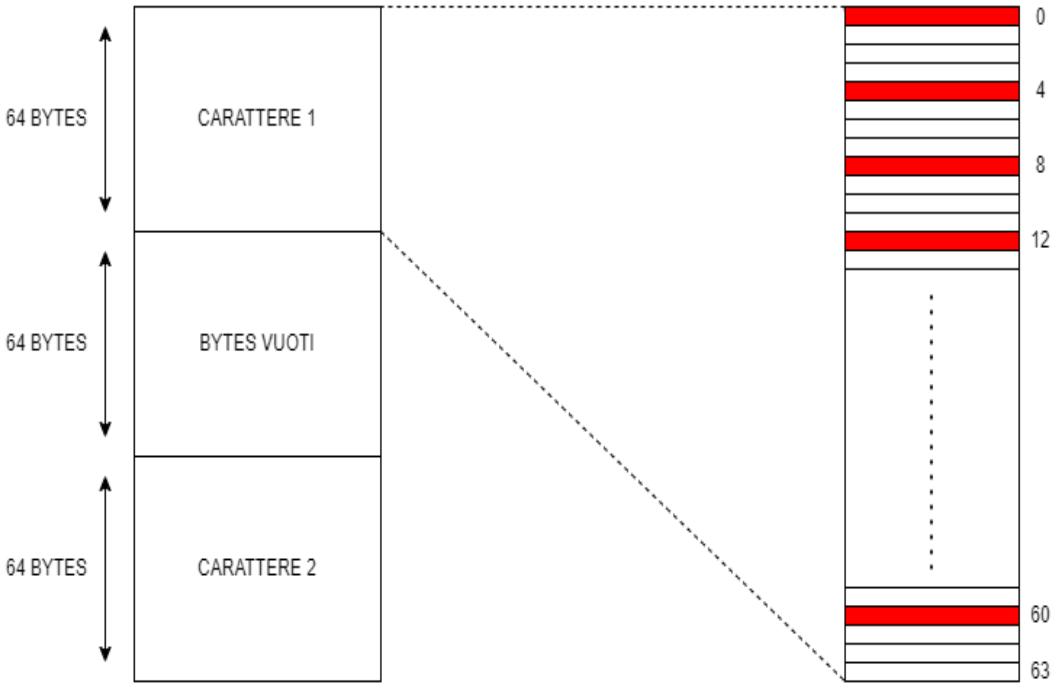



Figura 6: Caricamento font nella memoria interna della VGA
64 byte per carattere: all'interno di questi 64 byte, i 16 byte del font del carattere vengono caricati ogni 4 byte.

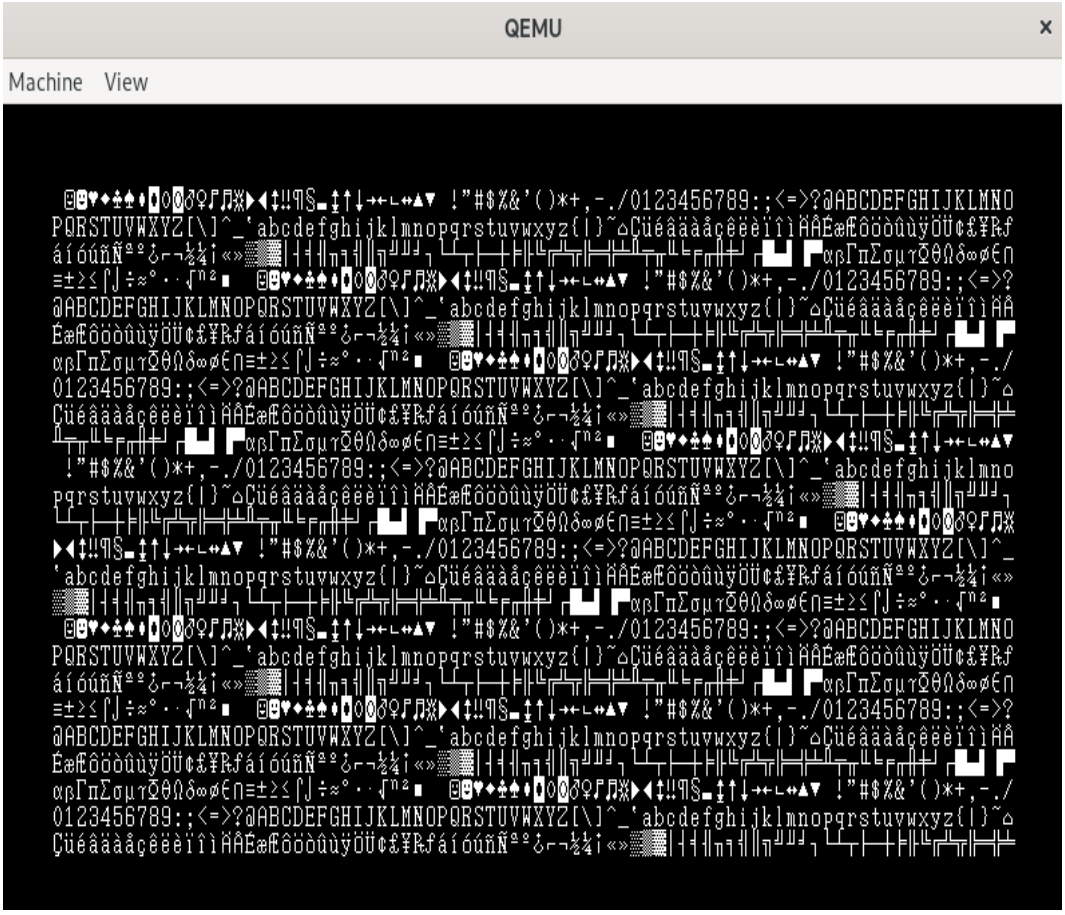



Figura 7: Visualizzazione a schermo del font caricato

### 6.2.2 Modalità Grafica

La funzione `init_graphicmode_320x200` inizializza invece la modalità grafica. Tale configurazione divide lo schermo in pixel: 320 colonne e 200 righe. Ogni pixel può essere colorato utilizzando il codice del corrispettivo colore all'interno della palette caricata. La palette di default è definita nel file `include/palette.h` e viene caricata nel formato RGB.

```
// FILE kernel/vga.cpp

void init_graphicmode_320x200(){

    //...

    //configurazione registri interni

    //...

    //reset index mode
    discard = VGA_BASE[INPUT_STATUS_REGISTER];
    //set bit PAS -> attribute controller a regime
    //abilita lo schermo
    VGA_BASE[AC] = 0x20;

    load_palette();
}

void load_palette(){
    // Set default VGA palette
    writeport(PC, 0xff, 0x00);

    // il primo colore deve essere caricato 2 volte
    writeport(PD, 0xff, (std_palette[0] & 0xff0000) >> 16);
    writeport(PD, 0xff, (std_palette[0] & 0x00ff00) >> 8);
    writeport(PD, 0xff, (std_palette[0] & 0x0000ff));

    for (int i = 0; i < 256; i++) {
        writeport(PD, 0xff, (std_palette[i] & 0xff0000) >> 16);
        writeport(PD, 0xff, (std_palette[i] & 0x00ff00) >> 8);
        writeport(PD, 0xff, (std_palette[i] & 0x0000ff));
    }
}
```

Codice 13: Parte della funzione `init_graphicmode_320x200` e funzione `load_palette`

Alla linea 15 viene attivato lo schermo.

## 6.3 Namespace vid

Viene definito un namespace per facilitare le operazioni con la VGA. Nel file `include/vid.h` sono presenti tutte le costanti utili ai fini di configurazione della scheda video, alcune costanti e funzioni di supporto alla modalità testo. Nel file `include/libce.h` sono invece dichiarate le rimanenti funzioni di supporto alla modalità testo.

```
1  // FILE include/vid.h
2
3  namespace vid {
4    //remapped VGA ports into qemu
5    #define AC            0x400  //attribute or palette registers
6    #define AC_READ       0x401  //attribute read register
7    #define MISC          0x402  //miscellaneous register
8    #define MISC_READ     0x40c  //miscellaneous read register
9    #define SEQ           0x404  //sequencer
10   #define SEQ_DATA      0x405  //sequencer data
11   #define GC            0x40e  //graphic address register
12   #define GC_DATA       0x40f  //graphic address register data
13   #define CRTC          0x414  //cathode ray tube controller
14                                //address register
15   #define CRTC_DATA     0x415  //CRTC data
16   #define PC            0x408  //PEL write index
17   #define PD            0x409  //PEL write data
18
19   #define INPUT_STATUS_REGISTER 0x41a //reset AC index mode
20
21   const natl COLS = 80;
22   const natl ROWS = 25;
23   const natl VIDEO_SIZE = COLS * ROWS;
24   const natb CUR_HIGH = 0x0e;
25   const natb CUR_LOW = 0x0f;
26
27   extern volatile natb* VGA_BASE;
```

```
28    extern volatile natb* video;
29    extern natb x, y;
30    extern natb attr;
31
32    void cursore();
33    void scroll();
34  }
```

Header 1: Namespace vid

`VGA_BASE` è il puntatore allo spazio dedicato ai registri interni della VGA e viene inizializzato nel file `libCE/vga_base.cpp`
`video` è il puntatore al buffer della memoria video ed è inizializzato nel file `libCE/video.cpp`
`x` e `y` indicano rispettivamente numero di colonna e numero di riga correnti del cursore e sono inizializzati nel file `libCE/xy.cpp`
`attr` indica l'attributo di default da utilizzare per la stampa dei caratteri e viene inizializzato dalla funzione `clear_screen`.

```
1  // FILE include/libce.h
2      namespace vid{
3          // Funzioni supporto video
4          void clear_screen(natb col);
5          void char_write(natb c);
6          void str_write(const char str[]);
7          natl cols();
8          natl rows();
9      }
```

Header 2: Supporto modalità testo

## 6.4 Funzioni di Supporto alla Modalità Testo

Il modulo I/O fa uso di funzioni di supporto per gestire lo schermo in modalità testo, le principali sono:

- `clear_screen(natb attribute)`
- `scroll()`
- `cursore()`
- `char_write(natb c)`
- `str_write(const char buf[])`
- `cols()`
- `rows()`

`clear_screen` permette di inizializzare lo schermo, selezionando il colore da utilizzare per i caratteri e lo sfondo. In particolare pulisce tutto lo schermo e posiziona il cursore nella prima cella.
`scroll` permette di scorrere lo schermo di una riga, inserendone una vuota in fondo e traslando il testo di una riga indietro.
`cursore` posiziona il cursore nella nuova cella dopo qualsiasi operazione a video.
`char_write` stampa un carattere a schermo.
`str_write` stampa una stringa di caratteri a schermo.
`cols` ritorna il numero di colonne della scheda video in modalità testo.
`rows` ritorna il numero di righe della scheda video in modalità testo.

```
1  // FILE libCE/clear_screen.cpp
2
3  namespace vid{
4
5    void clear_screen(natb attribute)
6    {
7
8      for(int i=0;i<VIDEO_SIZE*4;i+=4){
9        video[i] = ' ';
10       video[i+1] = attribute;
11     }
```

```
12      attr = attribute;
13    }
14  }
```

Codice 14: Corpo della funzione `clear_screen`

Alla linea `8` si inizializza tutto lo schermo con cella vuota e sfondo dell'attributo scelto.
Alla linea `12` si assegna l'attributo scelto all'attributo di default. `attr` è dichiarato nel file `include/vid.h` nel namespace `vid` ed indica l'attributo di default da utilizzare per scrivere nella memoria video.

```
1  // FILE libCE/scroll.cpp
2
3  namespace vid{
4    void scroll(){
5
6      for(unsigned int i = 0; i < VIDEO_SIZE*4 - COLS*4; i+=4){
7        video[i] = video[i+COLS*4];
8        video[i+1] = video[i+COLS*4+1];
9      }
10
11     for(unsigned int i = 0; i < COLS*4; i+=4){
12       video[VIDEO_SIZE*4 - COLS*4 + i] = ' ';
13       video[VIDEO_SIZE*4 - COLS*4 + i + 1] = attr;
14     }
15
16     if(y == 0){
17       x = 0;
18       cursore();
19       return;
20     }
21
22     y--;
23     cursore();
24   }
25 }
```

Codice 15: Corpo della funzione `scroll`

Alla linea `6` si fa la copia di ogni riga con la successiva, l'ultima riga viene invece inizializzata alla linea `8` vuota con attribute byte di default.
Successivamente si procede a riposizionare il cursore nella stessa colonna ma una riga di anticipo. Se la funzione viene chiamata mentre il cursore è posi-

zionato sulla prima riga, si procede spostando il cursore all'inizio della riga, senza decrementare.

```
// FILE libCE/str_write.cpp

    namespace vid{
        void cursore(){

            natl port = CRTC;
            natw pos = COLS * y + x;
            VGA_BASE[port] = CUR_HIGH;
            VGA_BASE[port + 1] = (natb)(pos >> 8);
            VGA_BASE[port] = CUR_LOW;
            VGA_BASE[port + 1] = (natb)pos;
        }
    }
```

Codice 16: Corpo della funzione `cursore`

Si calcola la posizione corrente del cursore e si inserisce nei registri interni della VGA.

```
// FILE libCE/char_write.cpp

namespace vid{
  void char_write(natb c) {

    switch (c) {
    case 0:
      break;
    case '\r':
      x = 0;
      break;
    case '\n':
      x = 0;
      y++;
      if (y >= ROWS)
        scroll();
      break;
    case '\b':
      if (x > 0 || y > 0) {
        if (x == 0) {
          x = COLS - 1;
          y--;
```

```
23          } else {
24            x--;
25          }
26        }
27        break;
28      default:
29        //character byte
30        video[y * COLS * 4 + x * 4] = c;
31        //attribute byte
32        video[y * COLS * 4 + x * 4 + 1] = attr;
33        x++;
34        if (x >= COLS) {
35          x = 0;
36          y++;
37        }
38        if (y >= ROWS)
39          scroll();
40        break;
41      }
42      cursore();
43    }
44  }
```

Codice 17: Corpo della funzione `char_write`

Per stampare il carattere a video si analizza prima se questo è un carattere speciale: il carattere speciale di `line feed` `\n` indica di andare su una nuova riga; il carattere speciale di `return carriage` `\r` indica di tornare alla prima colonna della riga attuale; il carattere speciale di `backspace` `\b` indica di spostare il cursore nella colonna precedente; il carattere speciale di fine stringa `\0` indica che non deve essere stampato niente.
Se il carattere non è speciale, si procede stampando il carattere scelto, usando l'attributo di default.
Alla linea `42` si calcola la nuova posizione del cursore, da salvare nella VGA.

```
1  // FILE libCE/str_write.cpp
2
3      namespace vid{
4          // scrive la sequenza di caratteri contenuta in buf[]
5          // (lo zero finale non viene scritto)
6          void str_write(const char buf[])
7          {
8              natl i = 0;
9              while (buf[i]) {
```

```
10                  char_write(buf[i]);
11                  i++;
12              }
13          }
14      }
```

Codice 18: Corpo della funzione `str_write`

Si itera la stringa stampando ogni carattere tramite la funzione `char_write`, fino al carattere di fine stringa che viene ignorato.

## 6.5 Primitive I/O

Come già anticipato, il modulo I/O mette a disposizione delle primitive per il modulo utente, così che questo possa interagire con le periferiche in modo controllato. La scheda video fa parte, insieme alla tastiera, della periferica *console*.
Le primitive offerte al modulo utente sono:

- `ini_console(natb attribute)`
- `read_console(char* buff, natq quanti)`
- `write_console(const char* buff, natq quanti)`

`ini_console` inizializza lo schermo tramite la funzione `clear_screen`, assegnando il valore `attribute` all'attributo di default `attr` del `namespace vid`.
`read_console` prende l'input da tastiera.
`write_console` scrive a video `quanti` caratteri da `buff`, in modalità testo. Viene fatto un controllo sugli indirizzi del buffer per assicurarsi che siano accessibili dal livello utente, così da evitare il problema del Cavallo di Troia, e sui permessi di accesso a tali indirizzi, così che non possano causare page fault nel modulo I/O.
La funzione `read_console` non riguarda questa tesi ed è stata implementata da chi si è occupato della tastiera: VIRTIO keyboard PCI.

# Conclusioni e indicazioni per la continuazione del progetto

Il lavoro svolto ha condotto ad una migrazione quasi completa del nucleo didattico in architettura RISC-V. Il primo approccio seguito è stato quello di realizzare e ampliare le funzionalità essenziali del nucleo separatamente, implementando la compilazione ed il caricamento del modulo io in memoria, modificando opportunamente le procedure di avvio. Durante questa fase si è dedicata attenzione anche alla pulizia ed alla omogeneizzazione del codice e dei sorgenti. Inoltre è stata configurata correttamente la scheda video VGA, la quale permette di operare sia in modalità testo sia in modalità grafica, e le funzioni per il suo corretto utilizzo. Sono state scritte le primitive I/O per accedere alla scheda video, non è però stato implementato il modo di accesso a tali primitive dal modulo utente.
Per terminare completamente la migrazione del nucleo didattico, si mettono in evidenza le seguenti mancanze interne da implementare:

- Ripristinare le primitive I/O per permettere le chiamate dai processi utente
- Integrare la periferica Hard-Disk
- Ripristinare le funzionalità di dump e di debug per aumentare la verbosità dei log sul terminale
- Realizzare le estensioni per il debugger in modo da ottenere informazioni e comandi aggiuntivi rispetto a quelli di base di gef e gdb
- Riorganizzare il file-system del progetto
- **Eliminare il solo ciclo infinito for(;;); nel file `libCE/reboot.cpp` per ripristinare il corretto shutdown**

Il progetto ha avuto un buon grado di complessità. Nonostante ciò, l'esperienza è stata molto formativa e ha permesso di entrare nel dettaglio su argomenti che non vengono trattati a lezione o che vengono trattati con generalità.

# Ringraziamenti

Ringrazio il Prof. Giuseppe Lettieri per la sua continua disponibilità e per avermi permesso di intraprendere questa esperienza istruttiva.
Un ringraziamento speciale va alla mia famiglia per il loro supporto fin da quando ero piccolo ed il loro sacrificio nel permettermi di studiare.

## Elenco delle Figure


1 Output da terminale per l'allocazione dell'heap e il caricamento dei moduli . . . . . 18
2 Spazio di configurazione PCI . . . . . 25
3 Font code page 737 . . . . . 31
4 Output a video dell'*esempio* sull'utilizzo della modalità grafica 33
5 Esempio su un utilizzo più pratico della modalità grafica - logo del corso di Calcolatori Elettronici . . . . . 34
6 Caricamento font nella memoria interna della VGA 64 byte per carattere: all'interno di questi 64 byte, i 16 byte del font del carattere vengono caricati ogni 4 byte. . . . . 38
7 Visualizzazione a schermo del font caricato . . . . . 39

# Elenco dei Codici, Comandi e Strutture


1 Regola 1 . . . . . 12
2 Regola 2 . . . . . 12
3 Regola 3 . . . . . 13
4 Codice 1: Corpo della funzione `sposta_ELF_moduli` . . . . 15
5 Codice 2: Parte della funzione `init_frame` . . . . . 17
6 Codice 3: Ridimensionamento dell'heap di sistema . . . . . 18
7 Codice 4: Corpo delle funzioni `carica_IO` e `crea_spazio_condiviso` . . . . . 19
8 Codice 5: creazione processo `main_IO` . . . . . 20
9 Codice 6: Processo `main_IO` . . . . . 20
10 Codice 7: Mapping del PCI nella finestra sulla memoria . . . . 22
11 Codice 8: Corpo funzione `pci_init` . . . . . 23
12 Struttura Dati 1: Spazio configurazione PCI . . . . . 26
13 Offset dei registri interni VGA . . . . . 28
14 Attribute byte . . . . . 30
15 Codice 9: Esempio modalità grafica . . . . . 32
16 Codice 10: Corpo funzione `vga_init` . . . . . 35
17 Codice 11: Corpo funzione `init_textmode_80x25` . . . . . 36
18 Codice 12: Parte della funzione `load_font` . . . . . 36
19 Codice 13: Parte della funzione `init_graphicmode_320x200` e funzione `load_palette` . . . . . 40
20 Header 1: Namespace vid . . . . . 41
21 Header 2: Supporto modalità testo . . . . . 42
22 Codice 14: Corpo della funzione `clear_screen` . . . . . 43
23 Codice 15: Corpo della funzione `scroll` . . . . . 44
24 Codice 16: Corpo della funzione `cursore` . . . . . 45
25 Codice 17: Corpo della funzione `char_write` . . . . . 45
26 Codice 18: Corpo della funzione `str_write` . . . . . 46

# Bibliografia


[1] Repository del progetto:
https://github.com/niccolopratesi/RISC-V-Calcolatori-Elettronici

[2] Official QEMU mirror:
https://github.com/qemu/qemu/tree/master/hw/riscv

[3] GNU toolchain for RISC-V, including GCC:
https://github.com/riscv-collab/riscv-gnu-toolchain

[4] RISC-V Specifications:
https://riscv.org/technical/specifications/

[5] QEMU virt board:
https://github.com/qemu/qemu/blob/master/hw/riscv/virt.c

[6] GNU Makefile manual:
https://www.gnu.org/software/make/manual/make.html

[7] Standard PCI: https://wiki.osdev.org/PCI

[8] QEMU Standard VGA: https://github.com/qemu/qemu/blob/v8.0.2/docs/specs/standard-vga.txt

[9] VGA hardware: https://wiki.osdev.org/VGA_Hardware

[10] VGA registers: http://www.osdever.net/FreeVGA/vga/portidx.htm

[11] VGA Specifications: http://www.techhelpmanual.com/70-video_graphics_array__vga_.html

[12] VGA Configuration Data: https://www.singlix.com/trdos/archive/vga/modes.c

[13] QEMU VGA source files: https://github.com/qemu/qemu/blob/v8.0.2/hw/display/vga.c#L1184