UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

Porting su architettura RISC-V di parte del nucleo didattico

Con particolare attenzione all'utilizzo di VGA

Relatore:
Prof: Giuseppe Lettieri

Candidato:
Edoardo Geraci

Anno accademico 2021/2022

Indice

1. Introduzione

RISC-V è un'architettura basata sul principio **RISC** (Reduced Instruction Set Computer) concepita nel 2010 dagli sviluppatori dell'università di Berkley in California. Ciò che ha di rivoluzionario ed unico RISC-V è un'**ISA** (Instruction Set Architecture) completamente open-source, su cui è possibile fare porting di software e per cui si può sviluppare hardware ed una struttura modulare, con microprocessori contenenti un numero estremamente ridotto di istruzioni rispetto ad x86 o ARM, rendendo il costo in dimensioni (e spese di produzione) di microprocessori per sistemi embedded molto più basso.
Le caratteristiche principali di **RISC-V** sono:

- Essere costruita attorno al principio RISC;
- Avere istruzioni che vengono eseguite tutte in un singolo ciclo di CPU;
- L'utilizzo di un'architettura load-store;
- Una forte modularità;
- Una versione a 32 bit e una versione a 64 bit, con estensioni per supportare le istruzioni con floating point;
- Un ampio supporto hardware.

Dalla sua nascita e fino ad oggi, le applicazioni di **RISC-V** sono state innumerevoli: Intelligenze artificiali, automotive, cloud servers, IoT, etc.
La sua natura open-source permette a realtà di sviluppo minori di innovare senza il peso del costo di licenza delle ISA proprietarie.

La seguente tesi tratta la scrittura di un rudimentale kernel eseguibile su architettura RISC-V, che da inizio ad un lavoro di porting del kernel didattico utilizzato per l'esame di Calcolatori Elettronici su architettura RISC-V.
Particolare enfasi è stata posta sull'analisi delle operazioni iniziali di setup delle modalità d'esecuzione, sull'utilizzo della UART e sulla configurazione di VGA in modalità grafica e testuale.
All'interno della tesi verrà analizzata inizialmente la struttura del progetto, per poi commentare il codice ed il flusso d'esecuzione, fino ad arrivare ad una stampa a video tramite VGA come Proof of Concept (PoC) di utilizzo della stessa.

2. Prerequisiti per l’utilizzo e lo sviluppo del sistema

Per poter compilare ed eseguire il codice del sistema è necessario innanzitutto installare una toolchain per RISC-V.
Per far ciò, su ubuntu è possibile utilizzare il comando:

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

Una volta installata la toolchain sarà necessario installare l’emulatore qemu, e l’architettura RISC-V a 64 bit presente all’interno del pacchetto di architetture qemu-system-misc.
All’interno del pacchetto è presente l’enviroment che utilizzeremo, ovvero qemu-system-riscv64.

Per debuggare utilizzando gdb, una volta installata la toolchain, basterà utilizzare il comando

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

dove file è l’elf che qemu utilizzerà come kernel.
È inoltre necessario configurare correttamente le variabili d’ambiente per utilizzare la toolchain, inserendo all’interno del config file della shell

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

Il progetto potrà poi essere compilato ed eseguito mediante il comando make, ed in particolare:

- **make run**: compila ed esegue il codice
- **make debug**: compila ed esegue il codice con la flag -S, che fa sì che qemu attenda il collegamento di gdb mediante porta TCP [in particolare viene utilizzata la porta standard ovvero 1234]
- **make clean**: elimina tutti i file non necessari
- **make compile:** compila il codice

Inoltre il makefile si occupa di individuare la toolchain per RISC-V e, se non presente, restituisce un errore.

3. Organizzazione del sistema

File	Descrizione
entry.s	Entry del sistema operativo, esegue le prime operazioni necessarie al boot
font.h	File contenente il font da fornire a VGA
kernel.ld	File di configurazione per il linker
main.c	Programma che chiama le funzioni di inizializzazione dei diversi sottosistemi
palette.h	File contenente la palette da fornire a VGA
pci.c	Inizializzazione della PCI al fine di utilizzare VGA in modalità grafica e testuale
start.s	Setup della modalità Supervisor di RISC-V e prime configurazioni
types.h	File contenente la definizione dei tipi utilizzati all'interno del progetto
uart.h	Piccola libreria di stampa sulla UART, utilizzata principalmente per inviare messaggi di debug
vga.c	Inizializzazione e utilizzo di VGA in modalità grafica e testuale

4. Analisi dettagliata del flusso di esecuzione

Affinché il sistema possa eseguire il nostro codice come kernel, è necessario, secondo specifica di RISC-V, all'indirizzo di memoria fisica **0x80000000**.
Affinché ciò sia possibile è necessario modificare il file di configurazione del linker utilizzato dalla toolchain di compilazione.
Tale modifica si trova appunto nel file **kernel.ld**:

```
/* START OF ADDITION */
MEMORY
{
  ram (rwxai) : ORIGIN = 0x80000000, LENGTH = 0x8000000
}
/* END OF ADDITION */
```

Attraverso questa aggiunta al file di configurazione standard del linker di qemu si ottiene la

creazione di un mapping di memoria a partire dall'indirizzo **0x80000000** grande 128 Mbyte con i permessi di lettura, scrittura ed esecuzione.

Fatto ciò possiamo compilare ed eseguire il nostro codice.

4.1 File entry.S

Il file **entry.S** contiene il codice che verrà eseguito all'avvio:

```
.section .init
.global _entry
_entry:
    # set up a stack for C.
    # stack0 is declared in main.c,
    # with a 4096-byte stack.
    # sp = stack0
    la sp, stack0
    li a0, 1024*4
    csrr a1, mhartid
    addi a1, a1, 1
    mul a0, a0, a1
    add sp, sp, a0
    # jump to start in start.s
    call start
```

La funzione **_entry** contiene il codice di inizializzazione dello stack su cui verrà eseguito il codice C. (Manuale dell'ASM di RISC-V)
Una volta inizializzato lo stack viene chiamata la funzione start, definita all'interno del file **start.S**.

4.2 File start.S

Il file **start.S** contiene le configurazioni necessarie all'utilizzo della Supervisor Mode (S-Mode).

4.2.1 Modalità di esecuzione di RISC-V

RISC-V ha tre modalità di esecuzione: Machine Mode (M-Mode), Supervisor Mode (S-Mode) e User-Mode (U-Mode).
La CPU si avvia in M-Mode; questa modalità è da utilizzare per effettuare tutte le configurazioni del computer utili a poter poi continuare a lavorare alternando tra S-Mode ed U-Mode. La M-Mode ha tutti i privilegi.
In S-Mode la CPU può eseguire le *istruzioni privilegiate*, come ad esempio l'abilitazione e la disabilitazione delle interruzioni, la gestione della paginazione, etc.
In U-Mode non si possono utilizzare le istruzioni privilegiate, e se un applicativo tenta di eseguirle viene attivata la S-Mode che ne causa la terminazione.
Se dell'applicativo che esegue in U-Mode necessita di eseguire delle istruzioni privilegiate può utilizzare l'istruzione *ecall* fornita dalla CPU che permette di eseguire in S-Mode del codice a partire da una entry definita dal kernel.

La funzione **start** effettua le configurazioni eseguibili solo in M-Mode, ed in particolare:

```
.global start
start:
  # set M Previous mode to Supervisor for mret
  csrr a0, mstatus
  li a1, 0x1800
  xori a1, a1, -1
  and a0, a0, a1
  li a1, 0x800
  or a0, a0, a1
  csrw mstatus, a0
```

La prima parte di codice configura il registro **mstatus** affinché si possa eseguire le chiamata alla funzione **mret** che cambierà la modalità di esecuzione da M-Mode ad S-Mode.

```
# set M Exception Program Counter to main
# main() defined in main.c
la a0, main
csrw mepc, a0
```

Successivamente viene configurato il Program Counter della M Exception, sollevata dalla chiamata ad **mret**, attraverso il registro **mepc** affinché il codice eseguito sia quello della funzione main definita nel file **main.c**.

```
# disable paging
li a0, 0x0
csrw satp, a0
```

Attraverso la scrittura nel registro **satp** viene disabilitato il paging.

```
# delegate all interrupts and exceptions to supervisor mode
li a0, 0xffff
csrw medeleg, a0
csrw mideleg, a0
csrr a0, sie
ori a0, a0, 0x200
ori a0, a0, 0x20
ori a0, a0, 0x2
csrw sie, a0
# configure physical memory protection to access to all
# physical Memory
li a0, 0x3fffffffffffULL
csrw pmpaddr0, a0
li a0, 0xf
csrw pmpcfg0, a0
mret
```

Successivamente viene delegata la gestione di tutte le eccezioni alla S-Mode e viene configurata la protezione della memoria affinché la S-Mode abbia accesso a tutta la memoria fisica disponibile.
Attraverso l'istruzione **mret** viene cambiata la modalità d'esecuzione da M-Mode ad S-Mode e viene eseguita la funzione **main**.

4.3 File main.c

```c
#include "types.h"
#include "uart.h"

__attribute__ ((aligned (16))) char stack0[4096];

int main(){

  char* message = "Running in S-Mode\n\r";
  printf(message);
  pci_init();
  message = "PCI initialized\n\r";
  printf(message);
  message = "VGA inizialized\n\r";
  printf(message);
  return 0;
}
```

All'interno del file **main.c** viene definito lo stack configurato durante l'esecuzione della funzione **_entry**.
La funzione **main** stampa dei messaggi di debug sulla UART mediante le funzioni definite nel file **uart.h** ed esegue l'inizializzazione della PCI mediante la funzione **pci_init** definita nel file **pci.c**.

4.4 File uart.h

Il file **uart.h** contiene la definizione di alcune funzioni di utilità per la stampa su UART, mappata in memoria all'indirizzo **0x10000000**:
Una funzione per la stampa degli interi con segno **print_int**:

```
static void print_int(int xx, int base, int sign)
{
  char buf[16];
  int i;
  uint x;

  if(sign && (sign = xx < 0))
      x = -xx;
  else
      x = xx;

  i = 0;
  do {
      buf[i++] = digits[x % base];
  } while((x /= base) != 0);

  if(sign)
      buf[i++] = '-';

  while(--i >= 0)
      WRITE_UART(buf[i]);
}
```

Una per la stampa dei numeri esadecimali, di particolare aiuto per la gestione del debug quando si lavora con indirizzi di memoria:

```
static void print_ptr(uint64 x)
{
  int i;
  WRITE_UART('0');
  WRITE_UART('x');
  for (i = 0; i < (sizeof(uint64) * 2); i++, x <<= 4)
      WRITE_UART(digits[x >> (sizeof(uint64) * 8 - 4)]);
}
```

Ed una piccola implementazione della funzione **printf**, che seppur non completa permette di effettuare la stampa molto più semplicemente:

```
static void printf(char *fmt, ...)
{
  va_list ap;
  int i, c;
  char *s;

  if (fmt == 0)
      panic("null fmt");
  va_start(ap, fmt);
  for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
      if(c != '%'){
      WRITE_UART(c);
      continue;
      }
      c = fmt[++i] & 0xff;
      if(c == 0)
      break;
      switch(c){
      case 'd':
      print_int(va_arg(ap, int), 10, 1);
      break;
      case 'x':
      print_int(va_arg(ap, int), 16, 1);
      break;
      case 'p':
      print_ptr(va_arg(ap, uint64));
      break;
      case 's':
      if((s = va_arg(ap, char*)) == 0)
      s = "(null)";
      for(; *s; s++)
      WRITE_UART(*s);
      break;
      case '%':
      WRITE_UART('%');
      break;
      default:
      // Print unknown % sequence to draw attention.
      WRITE_UART('%');
      WRITE_UART(c);
      break;
      }
  }
}
```

4.5 File pci.c

All'interno del file **pci.c** è definita la funzione **pci_init** che si occupa di inizializzare la PCI al fine di poter utilizzare la VGA.
Inizialmente occorre trovare il PCI device della VGA, che ha come ID 1111:1234 sul bus 0. Lo si identifica attraverso una ricerca lineare sul bus:

```
for(int dev = 0; dev < 32; dev++){
    int bus = 0;
    int func = 0;
    int offset = 0;
    uint32 off = (bus << 16) | (dev << 11) | (func << 8) | (offset);
    volatile uint32 *base = ecam + off;
    uint32 id = base[0];
```

Una volta identificato il dispositivo si procede a configurarlo:

```
    if(id == 0x11111234){
    // PCI device ID 1111:1234 is VGA
    // command and status register.
    // bit 0 : I/O access enable
    // bit 1 : memory access enable
    // bit 2 : enable mastering
    base[1] = 7;

    for(int i = 0; i < 6; i++){
    uint32 old = base[4+i];
    // writing all 1's to the BAR causes it to be
    // replaced with its size
    base[4+i] = 0xffffffff;
    base[4+i] = old;
    }

    // tell the VGA to reveal its framebuffer at
    // physical address 0x40000000.
    base[4+0] = vga_framebuffer;
    vga_init((char*)vga_framebuffer);
    }
```

In particolare abilitiamo il mastering, l'accesso all'I/O e l'accesso alla memoria della VGA, per poi far sì che il framebuffer della VGA, ovvero la zona di memoria dove andremo a scrivere per utilizzare la modalità grafica, sia localizzata all'indirizzo **0x40000000**.
Dopo aver effettuato le dovute configurazioni viene chiamata la funzione **vga_init**, che si occupa di inizializzare ed utilizzare la VGA.

4.6 File vga.c

Il file **vga.c** implementa la configurazione della VGA in modalità grafica 320x300 ed in modalità testuale 80x25(*).
All'interno del file vga.c vengono definite numerose funzioni atte a configurare ed utilizzare le due modalità.
In particolare abbiamo, per la modalità grafica:

- init_graphicmode_320x300
- load_palette

e per la modalità testuale:

- init_textmode_80x25
- init_MISC
- init_SEQ
- init_CRTC
- init_GC
- init_AC
- init_PD
- load_font
- post_font
- print_VGA

Abbiamo poi le funzioni **writeport** e **readport** che sono semplici funzioni di utilità per leggere e scrivere sulle IO-Ports di VGA, mappate nella memoria all'indirizzo fisico **0x3000000**.

Vediamo una breve analisi dei principali registri di VGA, con riferimento alle configurazioni inerenti alla modalità grafica:

4.6.1 MISCELLANEOUS REGISTER

```
writeport(MISC, 0xff, 0x63);
```

Miscellaneous Output Register (Read at 3CCh, Write at 3C2h)

7	6	5	4	3	2	1	0
VSYNCP	HSYNCP	O/E Page		Clock Select		RAM En.	I/OAS

Si effettua una scrittura nel *Miscellaneous Output Register* per configurare le polarità del sync pulse, la modalità Odd/Even, il clock etc.

4.6.2 SEQUENCER REGISTERS

```
writeport(SEQ, 0x00, 0x03);
writeport(SEQ, 0x01, 0x01);
writeport(SEQ, 0x02, 0x0f);
writeport(SEQ, 0x03, 0x00);
writeport(SEQ, 0x04, 0x0e);
```

Si effettua una scrittura nei *Sequencer Registers*, composto da Reset Register, Clocking Mode Register, Map Mask Register Character Map Register e Sequencer Memory Mode Register.
Attraverso quest'opera zione possiamo eseguire un reset sincrono dei registri, impostare la dimensione della memoria grafica (320x300) o testuale(80x25), abilitare la scrittura sui 4 plane di VGA, disabilitiare la modalità Odd/Even e impostare la memoria a 256KB.

4.6.3 CONTROL REGISTERS

Vertical Retrace End Register (Index 11h)

7	6	5	4	3	2	1	0
Protect	Bandwidth			Vertical Retrace End			

Attraverso la scrittura nel registro di Vertical Retrace End Register all'interno dei *Control Registers* abilitiamo la scrittura sui registri di VGA che andremo ad utilizzare nelle configurazioni successive.

```
writeport(CRTC, 0x11, 0x0e);
writeport(CRTC, 0x00, 0x5f);
writeport(CRTC, 0x01, 0x4f);
writeport(CRTC, 0x02, 0x50);
writeport(CRTC, 0x03, 0x82);
writeport(CRTC, 0x04, 0x54);
writeport(CRTC, 0x05, 0x82);
writeport(CRTC, 0x06, 0xbf);
writeport(CRTC, 0x07, 0x1f);

writeport(CRTC, 0x08, 0x00);
writeport(CRTC, 0x09, 0x41);

writeport(CRTC, 0x10, 0x9c);
writeport(CRTC, 0x12, 0x8f);
writeport(CRTC, 0x13, 0x28);
writeport(CRTC, 0x14, 0x40);
writeport(CRTC, 0x15, 0x96);
writeport(CRTC, 0x16, 0xb9);
writeport(CRTC, 0x17, 0xa3);
writeport(CRTC, 0x18, 0xff);
```

All’interno della documentazione sui Control Registers è presente un’analisi accurata di ogni singolo registro interessato da queste configurazioni.

4.6.4 GRAPHIC REGISTERS

Le configurazioni dei graphic registers sono standard sia per la modalità grafica che per quella testuale; la più notabile nella configurazione della modalità grafica è la scrittura al registro di indice **0x05**:

```
writeport(GC, 0x00, 0x00);
writeport(GC, 0x01, 0x00);
writeport(GC, 0x02, 0x00);
writeport(GC, 0x03, 0x00);
writeport(GC, 0x04, 0x00);
writeport(GC, 0x05, 0x40);
writeport(GC, 0x06, 0x05);
writeport(GC, 0x07, 0x0f);
writeport(GC, 0x08, 0xff);
```

Graphics Mode Register (Index 05h)

7	6	5	4	3	2	1	0
	Shift256	Shift Reg.	Host O/E	Read Mode		Write Mode	

La scrittura del valore **0x40**, ovvero **0b1000000** abilita l’utilizzo della modalità a 256 colori.

4.6.5 ATTRIBUTE CONTROL REGISTERS

```
writeport(AC, 0x00, 0x00);
writeport(AC, 0x01, 0x01);
writeport(AC, 0x02, 0x02);
writeport(AC, 0x03, 0x03);
writeport(AC, 0x04, 0x04);
writeport(AC, 0x05, 0x05);
writeport(AC, 0x06, 0x06);
writeport(AC, 0x07, 0x07);

writeport(AC, 0x08, 0x08);
writeport(AC, 0x09, 0x09);
writeport(AC, 0x0a, 0x0a);
writeport(AC, 0x0b, 0x0b);
writeport(AC, 0x0c, 0x0c);
writeport(AC, 0x0d, 0x0d);
writeport(AC, 0x0e, 0x0e);
writeport(AC, 0x0f, 0x0f);

writeport(AC, 0x10, 0x41);
writeport(AC, 0x11, 0x00);
writeport(AC, 0x12, 0x0f);
writeport(AC, 0x13, 0x00);
writeport(AC, 0x14, 0x00);

// Enable display
writeport(AC, 0xff, 0x20);
```

Configurazioni standard degli *Attribute Control Registers*.
Con l'ultima scrittura, attiviamo il display.

4.6.6 Funzioni di utilità

Effettuiamo una breve analisi delle funzioni più interessanti definite all'interno del file **vga.c**:

```
void load_font(unsigned char* font_16){
  VGA_BASE[PC] = 0x0;
  init_PD();
  writeport(MISC, 0x00, 0x67);
  VGA_BASE[AC] = 0x20;
  writeport(SEQ, 0x04, 0x06);
  writeport(SEQ, 0x01, 0x23);
  writeport(GC, 0x05, 0x00);
  writeport(GC, 0x06, 0x05);
  writeport(SEQ, 0x02, 0x04);

  volatile void *vga_buf = (void *)(0xa0000);
  for (uint32 i = 0; i < 256; ++i) {
      memcpy((void *)(vga_buf + 32 * i), (void*)(font_16+16 * i), 16);
  }
}
```

La funzione **load_font** carica il bitmap di un font 16*16 all'indirizzo **0xa0000**, specifica di VGA. Così facendo forniamo alla VGA il font da utilizzare in modalità testuale.

```
static void print_VGA(char *message, uint8 fg, uint8 bg)
{
  volatile uint8 *p = (void *)(0xb8000);
  int cursor = readport(CRTC, 0x0f);
  cursor = (cursor+1)*2 -2; // where to write
  while(*message){
      if(*message !=  0x0a){ //detect the '\n'
      p[cursor++] = *message;
      p[cursor++] = (bg<<4) | fg;
      }
      else{
      int line_offset = cursor%(VGA_TEXT_WIDTH*2);
      cursor = cursor + VGA_TEXT_WIDTH*2 - line_offset;
      printf("%d\n", cursor);
      }
      message++;
  }
  p[cursor++] = ' ';
  p[cursor++] = (bg<<4) | fg;
  // new cursor position
  writeport(CRTC, 0x0e, (cursor/2 -1) >> 8);
  writeport(CRTC, 0x0f, (cursor/2 -1) & 0xff);
}
```

La funzione **print_VGA** esegue una stampa su VGA in modalità testuale, gestendo contestualmente la posizione del cursore ed i newline.

5. Conclusioni e problemi

Ciò che abbiamo ottenuto è un punto di inizio per la creazione di un sistema operativo didattico su architettura RISC-V. La struttura del codice risulta essere diversa rispetto al nucleo didattico, prediligendo una maggior modularità. È importante sottolineare un problema riscontrato durante la stesura del progetto:
Il codice creato non funziona su una versione standard di qemu; è stato necessario attuare un cambiamento al codice che gestisce l'allocazione della memoria VGA di qemu affinché il codice potesse funzionare, ed alcuni indirizzi sono diversi nel codice funzionante rispetto a quello presentato in questa tesi.
In particolare, il file (**hw/display/vga.c**) ha la seguente modifica:

```
     memory_region_add_subregion_overlap(address_space,
-                                        0x000a0000,
+                                        0x50000000,
                                         vga_io_memory,
                                         1);
```

Questo fa sì che l'indirizzo di memoria che solitamente si troverebbe in **0xa0000** si trovi in **0x50000000**. Questa modifica si è resa necessaria poiché le configurazioni di qemu creano, su architettura RISC-V, un overlapping di indirizzi che rende la zona di memoria di VGA atta alla scrittura in modalità testuale inaccessibile.
Tuttora non è noto se esista un modo per accedervi comunque, o se sia un bug.

6. Ringraziamenti

In breve, ci tengo a ringraziare il mio relatore Giuseppe Lettieri, che mi ha concesso il suo tempo ogni volta che ne ho avuto necessità, i miei compagni di corso che mi hanno fatto svagare cantando “per dimenticare” degli Zero Assoluto in qualsiasi momento possibile, la mia famiglia che mi ha dato la possibilità non scontata di studiare senza dovermi preoccupare di altro, la signorina Eleonora Sara Bacci, che mi ha ascoltato parlare di kernel fino a notte fonda e mi ha supportato ogni volta che ho pensato che non sarei riuscito a fare ciò che volevo. Un ringraziamento particolare va a Filippo “AdaIlBarsa” Barsanti, senza il quale né io né questa tesi saremmo in piedi oggi.