Projet Mini-RISC

Rapport Technique

Émulateur de Processeur RISC-V Simplifié

Réalisé par :

Maria Eduarda Rizo

Encadrant :

Théotime Bollengier

Année Universitaire 2025

Table des matières

Chapitre 1

Introduction

Ce projet s'inscrit dans le cadre du cours d'Architecture des Ordinateurs. L'objectif est de comprendre le fonctionnement interne d'un processeur en développant un émulateur logiciel complet pour une architecture simplifiée de type RISC-V, nommée "Mini-RISC".

L'émulation est une technique fondamentale en informatique permettant à un système hôte (ici, un ordinateur personnel) de se comporter comme un autre système cible (le Mini-RISC). Ce projet couvre l'ensemble de la chaîne de fonctionnement d'un ordinateur : de la définition du jeu d'instructions (ISA) à l'exécution de programmes en langage assembleur, en passant par la gestion de la mémoire et des périphériques d'entrée/sortie.

Objectif du projet : Développer un émulateur en langage C capable d'exécuter des fichiers binaires compilés pour l'architecture Mini-RISC, et valider son fonctionnement avec un programme de test "Hello World".

Chapitre 2

Tableau d'exigences du système

2.1 Introduction au tableau d'exigences

Pour garantir le bon fonctionnement de l'émulateur et sa conformité avec les spécifications du cours, nous avons établi un tableau d'exigences. Ce tableau liste les fonctionnalités critiques que le système doit implémenter.

2.2 Tableau récapitulatif des exigences

TABLE 2.1: Résumé des exigences du projet Mini-RISC.

ID	Description	Type	Priorité
E01	Implémenter le cycle Fetch-Decode-Execute du CPU	Fonctionnelle	**Critique**
E02	Supporter le jeu d'instructions Mini-RISC (custom opcodes)	Technique	**Critique**
E03	Gérer une mémoire RAM de 32 MiB à l'adresse 0x80000000	Technique	Obligatoire
E04	Implémenter le périphérique "CharOut" à l'adresse 0x10000000	Technique	Obligatoire
E05	Permettre l'affichage de caractères, entiers signés et hexadécimaux	Fonctionnelle	Obligatoire
E06	Charger un fichier binaire (.bin) en mémoire au démarrage	Fonctionnelle	Obligatoire
E07	Exécuter un programme de test "Hello World" en assembleur	Validation	**Critique**
E08	Fournir un Makefile pour la compilation de l'émulateur	Documentation	Obligatoire
E09	Fournir un Makefile pour la cross-compilation des tests	Documentation	Obligatoire

2.3 Analyse

Les exigences se divisent en deux blocs principaux :

— **Le Cœur (CPU)** : responsable de la logique d'exécution (E01, E02).

— **La Plateforme** : responsable de l'environnement (RAM, périphériques) (E03, E04, E05).

Cette séparation dicte l'architecture logicielle du projet, avec des modules distincts pour le processeur (`minirisc.c`) et la plateforme (`platform.c`).

Chapitre 3

Développement et Implémentation

3.1 Architecture Globale

Le projet est structuré en trois modules C principaux, reflétant l'architecture matérielle émulée :

— `platform.c/h` : bus système, RAM, et périphérique CharOut.
— `minirisc.c/h` : cœur CPU (registres, PC, fetch/decode/execute).
— `main.c` : chargement du binaire et lancement de l'émulation.

Dans la suite, chaque fichier source est présenté **en intégralité** (listing) avec une **explication juste en dessous** pour clarifier son rôle et les choix d'implémentation.

Chapitre 4

Code source (listings) et explications

4.1 emulator/platform.h

Listing 4.1 – Fichier `platform.h`

```c
#ifndef PLATFORM_H
#define PLATFORM_H

#include <stdint.h>

// Memory map
#define RAM_BASE_ADDR 0x80000000
#define RAM_SIZE (32 * 1024 * 1024) // 32 MiB
#define RAM_END_ADDR (RAM_BASE_ADDR + RAM_SIZE)

#define CHAROUT_BASE_ADDR 0x10000000
#define CHAROUT_END_ADDR 0x1000000C

typedef struct {
  uint8_t *memory;
} platform_t;

platform_t *platform_new();
void platform_free(platform_t *p);

uint8_t platform_read_8(platform_t *p, uint32_t addr);
uint16_t platform_read_16(platform_t *p, uint32_t addr);
uint32_t platform_read_32(platform_t *p, uint32_t addr);

void platform_write_8(platform_t *p, uint32_t addr, uint8_t value);
void platform_write_16(platform_t *p, uint32_t addr, uint16_t value);
void platform_write_32(platform_t *p, uint32_t addr, uint32_t value);

#endif
```

Explication.

— Définit la **carte mémoire** exigée : RAM à `0x80000000` (32 MiB) et périphérique CharOut à `0x10000000`.

— La structure `platform_t` contient uniquement un pointeur vers la RAM (`memory`), allouée dynamiquement.

— Les fonctions `platform_read_*` / `platform_write_*` représentent le **bus** : toute

lecture/écriture CPU passe par elles.

4.2 emulator/platform.c

Listing 4.2 – Fichier `platform.c`

```c
#include "platform.h"
#include <stdio.h>
#include <stdlib.h>

platform_t *platform_new() {
  platform_t *p = (platform_t *)malloc(sizeof(platform_t));
  if (!p) {
    fprintf(stderr, "Error: cannot allocate platform.\n");
    exit(1);
  }

  p->memory = (uint8_t *)calloc(RAM_SIZE, sizeof(uint8_t));
  if (!p->memory) {
    fprintf(stderr, "Error: cannot allocate RAM.\n");
    free(p);
    exit(1);
  }

  return p;
}

void platform_free(platform_t *p) {
  if (!p) return;
  free(p->memory);
  free(p);
}

static int is_in_ram(uint32_t addr) {
  return (addr >= RAM_BASE_ADDR && addr < RAM_END_ADDR);
}

static int is_in_charout(uint32_t addr) {
  return (addr >= CHAROUT_BASE_ADDR && addr < CHAROUT_END_ADDR);
}

uint8_t platform_read_8(platform_t *p, uint32_t addr) {
  if (is_in_ram(addr)) {
    return p->memory[addr - RAM_BASE_ADDR];
  }

  fprintf(stderr, "Error: read_8 from unmapped address 0x%08x\n", addr);
```

```
42    return 0;
43  }
44
45  uint16_t platform_read_16(platform_t *p, uint32_t addr) {
46    uint16_t lo = platform_read_8(p, addr);
47    uint16_t hi = platform_read_8(p, addr + 1);
48    return (hi << 8) | lo;
49  }
50
51  uint32_t platform_read_32(platform_t *p, uint32_t addr) {
52    uint32_t b0 = platform_read_8(p, addr);
53    uint32_t b1 = platform_read_8(p, addr + 1);
54    uint32_t b2 = platform_read_8(p, addr + 2);
55    uint32_t b3 = platform_read_8(p, addr + 3);
56    return (b3 << 24) | (b2 << 16) | (b1 << 8) | b0;
57  }
58
59  void platform_write_8(platform_t *p, uint32_t addr, uint8_t value) {
60    if (is_in_ram(addr)) {
61      p->memory[addr - RAM_BASE_ADDR] = value;
62      return;
63    }
64
65    if (is_in_charout(addr)) {
66      // CharOut: writing a byte prints it as a character
67      putchar((char)value);
68      fflush(stdout);
69      return;
70    }
71
72    fprintf(stderr, "Error: write_8 to unmapped address 0x%08x\n", addr);
73  }
74
75  void platform_write_16(platform_t *p, uint32_t addr, uint16_t value) {
76    platform_write_8(p, addr, (uint8_t)(value & 0xFF));
77    platform_write_8(p, addr + 1, (uint8_t)((value >> 8) & 0xFF));
78  }
79
80  void platform_write_32(platform_t *p, uint32_t addr, uint32_t value) {
81    if (is_in_charout(addr)) {
82      // Optional: print 32-bit values if teacher's spec requires it
83      // Here: default behavior prints as signed int for debugging
84      printf("%d", (int32_t)value);
85      fflush(stdout);
86      return;
87    }
88
89    platform_write_8(p, addr, (uint8_t)(value & 0xFF));
```

```
90    platform_write_8(p, addr + 1, (uint8_t)((value >> 8) & 0xFF));
91    platform_write_8(p, addr + 2, (uint8_t)((value >> 16) & 0xFF));
92    platform_write_8(p, addr + 3, (uint8_t)((value >> 24) & 0xFF));
93  }
```

Explication.

- **Allocation RAM :** `calloc(RAM_SIZE)` initialise la RAM à zéro, comme au reset.
- **Décodage du mapping :** `is_in_ram` et `is_in_charout` filtrent l'adresse.
- **Endianness :** `read_16/read_32` reconstruisent les valeurs en little-endian (octet faible à l'adresse basse).
- **CharOut :** un `write_8` sur `0x10000000..` déclenche `putchar` (sortie terminal).
- **write_32 :** si l'adresse est CharOut, on choisit ici d'afficher un entier signé (utile pour debug). Sinon, écriture octet par octet.

4.3 emulator/minirisc.h

Listing 4.3 – Fichier `minirisc.h`

```
1  #ifndef MINIRISC_H
2  #define MINIRISC_H
3
4  #include <stdint.h>
5  #include "platform.h"
6
7  typedef struct {
8    uint32_t PC;
9    uint32_t IR;
10   uint32_t next_PC;
11
12   uint32_t regs[32];
13   int halt;
14
15   platform_t *platform;
16 } minirisc_t;
17
18 minirisc_t *minirisc_new(uint32_t pc_start, platform_t *platform);
19 void minirisc_free(minirisc_t *mr);
20
21 void minirisc_fetch(minirisc_t *mr);
22 void minirisc_decode_and_execute(minirisc_t *mr);
23 void minirisc_run(minirisc_t *mr);
24
25 #endif
```

Explication.

- La structure `minirisc_t` regroupe tout l'état CPU : **PC**, **IR**, **next_PC**, banque de **32 registres**, et un flag `halt`.
- `platform` est un pointeur vers la plateforme : le CPU ne touche jamais la RAM directement, il passe par le bus (`platform_read/write`).
- Le découpage `fetch` / `decode_and_execute` / `run` correspond au modèle CPU demandé dans le sujet.

4.4 emulator/minirisc.c

Listing 4.4 – Fichier `minirisc.c`

```c
#include "minirisc.h"
#include <stdio.h>
#include <stdlib.h>

#define GET_OPCODE(inst) ((inst)&0x7F)
#define GET_RD(inst) (((inst) >> 7) & 0x1F)
#define GET_FUNCT3(inst) (((inst) >> 12) & 0x07)
#define GET_RS1(inst) (((inst) >> 15) & 0x1F)
#define GET_RS2(inst) (((inst) >> 20) & 0x1F)
#define GET_FUNCT7(inst) (((inst) >> 25) & 0x7F)

#define OPCODE_LUI 0x37
#define OPCODE_AUIPC 0x17
#define OPCODE_JAL 0x6F
#define OPCODE_JALR 0x67
#define OPCODE_BRANCH 0x63
#define OPCODE_LOAD 0x03
#define OPCODE_STORE 0x23
#define OPCODE_OP_IMM 0x13
#define OPCODE_SYSTEM 0x73

minirisc_t *minirisc_new(uint32_t pc_start, platform_t *platform) {
  minirisc_t *mr = (minirisc_t *)calloc(1, sizeof(minirisc_t));
  if (!mr) {
    fprintf(stderr, "Error: cannot allocate CPU.\n");
    exit(1);
  }

  mr->PC = pc_start;
  mr->platform = platform;
  mr->halt = 0;

  mr->regs[0] = 0;
  return mr;
```

```
35 }
36
37 void minirisc_free(minirisc_t *mr) {
38   free(mr);
39 }
40
41 void minirisc_fetch(minirisc_t *mr) {
42   mr->IR = platform_read_32(mr->platform, mr->PC);
43 }
44
45 static int32_t sign_extend(uint32_t val, int bits) {
46   uint32_t m = 1u << (bits - 1);
47   return (int32_t)((val ^ m) - m);
48 }
49
50 static int32_t imm_i(uint32_t inst) {
51   return (int32_t)inst >> 20;
52 }
53
54 static int32_t imm_s(uint32_t inst) {
55   uint32_t imm4_0 = (inst >> 7) & 0x1F;
56   uint32_t imm11_5 = (inst >> 25) & 0x7F;
57   return sign_extend((imm11_5 << 5) | imm4_0, 12);
58 }
59
60 static int32_t imm_b(uint32_t inst) {
61   uint32_t imm11 = (inst >> 7) & 0x1;
62   uint32_t imm4_1 = (inst >> 8) & 0xF;
63   uint32_t imm10_5 = (inst >> 25) & 0x3F;
64   uint32_t imm12 = (inst >> 31) & 0x1;
65
66   uint32_t imm = (imm12 << 12) | (imm11 << 11) | (imm10_5 << 5) | (
       imm4_1 << 1);
67   return sign_extend(imm, 13);
68 }
69
70 static int32_t imm_u(uint32_t inst) {
71   return (int32_t)(inst & 0xFFFFF000);
72 }
73
74 static int32_t imm_j(uint32_t inst) {
75   uint32_t imm19_12 = (inst >> 12) & 0xFF;
76   uint32_t imm11 = (inst >> 20) & 0x1;
77   uint32_t imm10_1 = (inst >> 21) & 0x3FF;
78   uint32_t imm20 = (inst >> 31) & 0x1;
79
80   uint32_t imm = (imm20 << 20) | (imm19_12 << 12) | (imm11 << 11) | (
       imm10_1 << 1);
```

```
81    return sign_extend(imm, 21);
82  }
83
84  void minirisc_decode_and_execute(minirisc_t *mr) {
85    uint32_t inst = mr->IR;
86    uint32_t opcode = GET_OPCODE(inst);
87    uint32_t rd = GET_RD(inst);
88    uint32_t rs1 = GET_RS1(inst);
89    uint32_t rs2 = GET_RS2(inst);
90    uint32_t funct3 = GET_FUNCT3(inst);
91
92    mr->next_PC = mr->PC + 4;
93
94    switch (opcode) {
95
96      case OPCODE_LUI: {
97        int32_t imm = imm_u(inst);
98        if (rd != 0) mr->regs[rd] = (uint32_t)imm;
99        break;
100     }
101
102     case OPCODE_AUIPC: {
103       int32_t imm = imm_u(inst);
104       if (rd != 0) mr->regs[rd] = mr->PC + (uint32_t)imm;
105       break;
106     }
107
108     case OPCODE_JAL: {
109       int32_t imm = imm_j(inst);
110       if (rd != 0) mr->regs[rd] = mr->PC + 4;
111       mr->next_PC = mr->PC + imm;
112       break;
113     }
114
115     case OPCODE_JALR: {
116       int32_t imm = imm_i(inst);
117       uint32_t target = (mr->regs[rs1] + (uint32_t)imm) & ~1u;
118       if (rd != 0) mr->regs[rd] = mr->PC + 4;
119       mr->next_PC = target;
120       break;
121     }
122
123     case OPCODE_BRANCH: {
124       int32_t imm = imm_b(inst);
125       // Only BEQ is used in the provided test
126       if (funct3 == 0x0) { // BEQ
127         if (mr->regs[rs1] == mr->regs[rs2]) {
128           mr->next_PC = mr->PC + imm;
```

```
129         }
130       } else {
131         fprintf(stderr, "Unsupported BRANCH funct3=0x%x\n", funct3);
132         mr->halt = 1;
133       }
134       break;
135     }
136 
137     case OPCODE_LOAD: {
138       int32_t imm = imm_i(inst);
139       uint32_t addr = mr->regs[rs1] + (uint32_t)imm;
140 
141       if (funct3 == 0x4) { // LBU
142         uint8_t v = platform_read_8(mr->platform, addr);
143         if (rd != 0) mr->regs[rd] = (uint32_t)v;
144       } else {
145         fprintf(stderr, "Unsupported LOAD funct3=0x%x\n", funct3);
146         mr->halt = 1;
147       }
148       break;
149     }
150 
151     case OPCODE_STORE: {
152       int32_t imm = imm_s(inst);
153       uint32_t addr = mr->regs[rs1] + (uint32_t)imm;
154 
155       if (funct3 == 0x0) { // SB
156         platform_write_8(mr->platform, addr, (uint8_t)(mr->regs[rs2] & 0
               xFF));
157       } else {
158         fprintf(stderr, "Unsupported STORE funct3=0x%x\n", funct3);
159         mr->halt = 1;
160       }
161       break;
162     }
163 
164     case OPCODE_OP_IMM: {
165       int32_t imm = imm_i(inst);
166       // Only ADDI is used in the provided test
167       if (funct3 == 0x0) { // ADDI
168         if (rd != 0) mr->regs[rd] = mr->regs[rs1] + (uint32_t)imm;
169       } else {
170         fprintf(stderr, "Unsupported OP-IMM funct3=0x%x\n", funct3);
171         mr->halt = 1;
172       }
173       break;
174     }
175 
```

```
176     case OPCODE_SYSTEM: {
177       // ebreak => stop emulator
178       mr->halt = 1;
179       printf("\nEBREAK executed. Halting.\n");
180       break;
181     }
182
183     default:
184       fprintf(stderr, "Unknown opcode 0x%x at PC=0x%08x\n", opcode, mr->
              PC);
185       mr->halt = 1;
186       break;
187   }
188 }
189
190 void minirisc_run(minirisc_t *mr) {
191   while (!mr->halt) {
192     minirisc_fetch(mr);
193     minirisc_decode_and_execute(mr);
194     mr->PC = mr->next_PC;
195     mr->regs[0] = 0;
196   }
197 }
```

Explication.

- **Macros GET_* :** extraient les champs instruction (`opcode`, `rd`, `rs1`, `rs2`, `funct3`).
- **Immédiats :** les fonctions `imm_i/s/b/u/j` reconstruisent les immédiats des formats RISC-V (puis extension de signe).
- **PC :** par défaut `next_PC = PC + 4`; les sauts/branches écrasent `next_PC`.
- **Sous-ensemble minimal :** pour le test `hello.s`, seules les instructions nécessaires sont gérées : `LUI/AUIPC`, `JAL/JALR`, `BEQ`, `LBU`, `SB`, `ADDI`, `EBREAK`.
- **x0 :** la ligne `mr->regs[0] = 0;` force le registre `x0` à rester constant, comme dans RISC-V.

4.5 emulator/main.c

Listing 4.5 – Fichier `main.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "platform.h"
5 #include "minirisc.h"
6
7 int main(int argc, char **argv) {
8   if (argc < 2) {
```

```
    fprintf(stderr, "Usage: %s <program.bin>\n", argv[0]);
    return 1;
  }

  const char *filename = argv[1];

  FILE *file = fopen(filename, "rb");
  if (!file) {
    perror("Error opening file");
    return 1;
  }

  platform_t *platform = platform_new();

  size_t bytes_read = fread(platform->memory, 1, RAM_SIZE, file);
  fclose(file);

  printf("Loaded %zu bytes from %s into memory.\n", bytes_read, filename
      );
  printf("Starting emulation...\n");

  minirisc_t *cpu = minirisc_new(RAM_BASE_ADDR, platform);
  minirisc_run(cpu);

  printf("Emulation finished.\n");

  minirisc_free(cpu);
  platform_free(platform);

  return 0;
}
```

Explication.

- Charge le fichier `.bin` directement en RAM à l'offset 0 (qui correspond à `RAM_BASE_ADDR`).
- Initialise le CPU avec `PC = 0x80000000` (base RAM) : le premier fetch lit donc l'instruction au début du binaire.
- Lance l'émulation via `minirisc_run` et libère correctement les ressources (CPU + RAM).

4.6 `emulator/Makefile`

Listing 4.6 – Fichier `emulator/Makefile`

```
CC=gcc
CFLAGS=-Wall -Wextra -O2
```

```
3
4  BUILD_DIR=build
5  TARGET=$(BUILD_DIR)/emulator
6
7  SRCS=main.c platform.c minirisc.c
8  OBJS=$(SRCS:.c=.o)
9
10 all: $(TARGET)
11
12 $(TARGET): $(OBJS)
13     mkdir -p $(BUILD_DIR)
14     $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)
15
16 %.o: %.c
17     $(CC) $(CFLAGS) -c $< -o $@
18
19 clean:
20     rm -rf $(BUILD_DIR) *.o
21
22 .PHONY: all clean
```

Explication.

- Compile l'émulateur côté **machine hôte** (gcc standard).
- `SRCS` liste explicitement les trois modules (`main/platform/minirisc`).
- Le binaire final est placé dans `build/emulator`, ce qui évite de polluer le dossier racine.
- `make clean` supprime le dossier `build` et les objets `.o`.

4.7 `embedded_software/hello.s`

Listing 4.7 – Fichier `embedded_software/hello.s`

```
1  .global _start
2
3  .section .text
4  _start:
5      la a0, my_string_1
6      jal ra, put_string
7
8      la a0, my_string_2
9      jal ra, put_string
10
11     ebreak
12
13 put_string:
14     li a2, 0x10000000
```

```
15 loop_start:
16     lbu a1, 0(a0)
17     beq a1, zero, loop_end
18     sb  a1, 0(a2)
19     addi a0, a0, 1
20     j loop_start
21 loop_end:
22     ret
23
24 .section .rodata
25 my_string_1:
26     .string "Hello, World!\n"
27 my_string_2:
28     .string "What's up?\n"
```

Explication.

- **Principe :** affiche deux chaînes en écrivant des octets dans le périphérique mémoire-mappé CharOut (`0x10000000`).
- `put_string` lit la chaîne octet par octet (`lbu`) jusqu'au `0x00` final et écrit chaque caractère via `sb`.
- `ebreak` provoque l'arrêt de l'émulateur (cas `OPCODE_SYSTEM`).

4.8 `embedded_software/Makefile`

Listing 4.8 – Fichier `embedded_software/Makefile`

```
1  TOOLCHAIN_PREFIX=../gcc_MINIRISC/riscv32-MINIRISC-elf/bin/riscv32-
       MINIRISC-elf
2  AS=$(TOOLCHAIN_PREFIX)-as
3  LD=$(TOOLCHAIN_PREFIX)-ld
4  OBJCOPY=$(TOOLCHAIN_PREFIX)-objcopy
5
6  BUILD_DIR=build
7  ELF=$(BUILD_DIR)/esw.elf
8  BIN=$(BUILD_DIR)/esw.bin
9
10 LDSCRIPT=linker.ld
11 SRC=hello.s
12 OBJ=$(BUILD_DIR)/hello.o
13
14 all: $(BIN)
15
16 $(BUILD_DIR):
17     mkdir -p $(BUILD_DIR)
18
19 $(OBJ): $(SRC) | $(BUILD_DIR)
```

```
20      $(AS) -o $(OBJ) $(SRC)
21
22  $(ELF): $(OBJ)
23      $(LD) -T $(LDSCRIPT) -o $(ELF) $(OBJ)
24
25  $(BIN): $(ELF)
26      $(OBJCOPY) -O binary $(ELF) $(BIN)
27
28  clean:
29      rm -rf $(BUILD_DIR)
30
31  .PHONY: all clean
```

Explication.

- Utilise le **cross-toolchain** Mini-RISC fourni dans le dépôt (`gcc_MINIRISC/...`).
- Étapes : assemble `hello.s` → objet, link via `linker.ld` → `esw.elf`, puis extraction binaire brut → `esw.bin`.
- Le fichier `esw.bin` est celui chargé par `emulator/main.c` au démarrage.

4.9 `README.md` (PT-BR + EN)

Listing 4.9 – Fichier `README.md` (contenu documenté)

```
1  Projeto Emulador Mini-RISC
2
3  Este projeto consiste em um emulador para a arquitetura Mini-RISC (
       baseada em RISC-V)
4  e um conjunto de softwares de teste em Assembly.
5
6  O que o projeto faz?
7  - CPU Mini-RISC: executa instrucoes como addi, beq, jal, sb, lbu, etc.
8  - Memoria RAM: 32 MiB (base 0x80000000)
9  - Periferico CharOut: 0x10000000 (saida no terminal)
10
11 Fluxo:
12 1) O emulador carrega um arquivo .bin para a memoria.
13 2) A CPU faz Fetch / Decode / Execute.
14 3) O programa termina ao executar ebreak.
15
16 Estrutura de Arquivos
17 - emulator/: codigo fonte do emulador em C (main.c, minirisc.c, platform
       .c)
18 - embedded_software/: codigo Assembly de teste (hello.s) + Makefile
19 - gcc_MINIRISC/: toolchain (necessario para gerar os binarios)
20
21 Como Compilar e Rodar
```

```
22
23 1) Compilar o Emulador
24 cd emulator
25 make
26
27 2) Compilar o Teste (Hello World)
28 cd ../embedded_software
29 make
30 (gera build/esw.bin)
31
32 3) Executar
33 cd ../emulator
34 ./build/emulator ../embedded_software/build/esw.bin
35
36 Saida Esperada
37 Loaded 83 bytes from ../embedded_software/build/esw.bin into memory.
38 Starting emulation...
39 Hello, World!
40 What's up?
41 EBREAK executed. Halting.
42 Emulation finished.
43
44 Limpeza
45 cd emulator && make clean
46 cd ../embedded_software && make clean
47
48 ---------------------------------------------------------------
49
50 Mini-RISC Emulator Project
51
52 This project consists of an emulator for the Mini-RISC architecture (
      based on RISC-V)
53 and a set of Assembly test software.
54
55 What does the project do?
56 - Mini-RISC CPU: executes addi, beq, jal, sb, lbu, etc.
57 - RAM Memory: 32 MiB (base 0x80000000)
58 - Output Peripheral (CharOut): 0x10000000 (prints to terminal)
59
60 Flow:
61 1) The emulator loads a .bin file into memory.
62 2) The CPU runs Fetch / Decode / Execute.
63 3) The program stops on ebreak.
64
65 File Structure
66 - emulator/: emulator source code in C (main.c, minirisc.c, platform.c)
67 - embedded_software/: test Assembly code (hello.s) + Makefile
68 - gcc_MINIRISC/: toolchain (required to generate binaries)
```

```
69
70 How to Compile and Run
71
72 1) Compile the Emulator
73 cd emulator
74 make
75
76 2) Compile the Test (Hello World)
77 cd ../embedded_software
78 make
79 (generates build/esw.bin)
80
81 3) Run
82 cd ../emulator
83 ./build/emulator ../embedded_software/build/esw.bin
84
85 Expected Output
86 Loaded 83 bytes from ../embedded_software/build/esw.bin into memory.
87 Starting emulation...
88 Hello, World!
89 What's up?
90 EBREAK executed. Halting.
91 Emulation finished.
92
93 Cleaning
94 cd emulator && make clean
95 cd ../embedded_software && make clean
```

Explication.

- Cette section documente le **README bilingue** (PT-BR puis EN) livré avec le projet.
- Le README sert surtout a garantir la **reproductibilite** : il donne les commandes exactes pour compiler l'emulateur, compiler le test et lancer l'execution.
- La partie **Sortie attendue** est une preuve de bon fonctionnement : ecriture via CharOut + arret propre sur `ebreak`.
- Cette version LaTeX est volontairement **sans les blocs Markdown (“‘)** pour eviter des erreurs de compilation avec `listings`.

Chapitre 5

Utilisation et Validation

5.1 Procédure de Compilation

5.1.1 1. Compiler l'Émulateur

```
cd emulator
make
```

5.1.2 2. Compiler le Programme de Test

```
cd ../embedded_software
make
```

Cela génère `build/esw.bin`.

5.2 Exécution et Résultats

```
cd ../emulator
./build/emulator ../embedded_software/build/esw.bin
```

5.2.1 Sortie Attendue

```
Loaded <N> bytes from ../embedded_software/build/esw.bin into memory.
Starting emulation...
Hello, World!
What's up?

EBREAK executed. Halting.
Emulation finished.
```

5.2.2 Sortie Courante

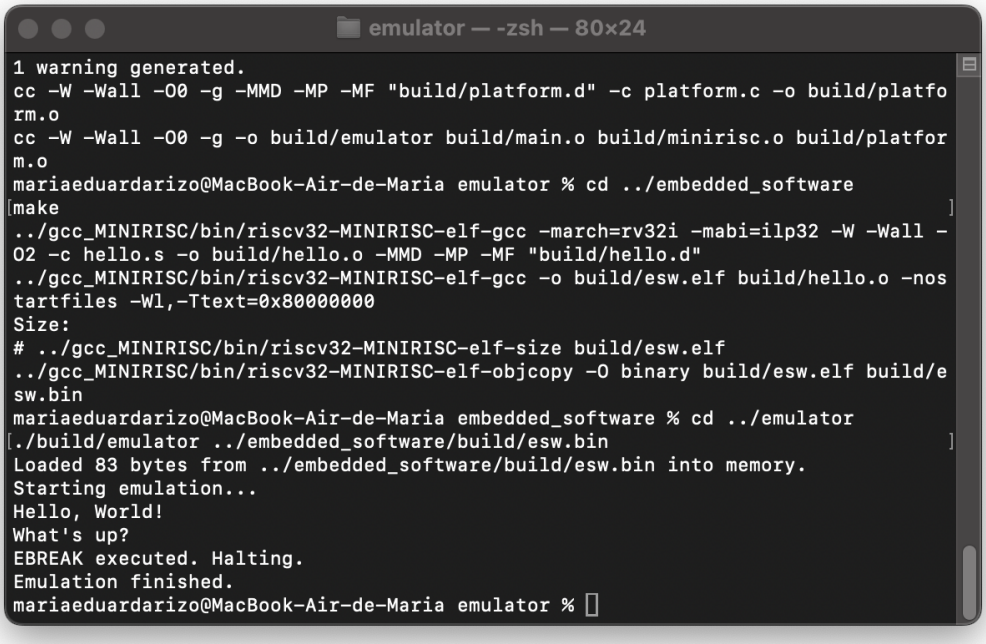

FIGURE 5.1 – Sortie Actuel

La sortie obtenue lors de l'exécution du programme est présentée ci-dessous.

Chapitre 6

Conclusion

Ce projet a permis de mettre en pratique les concepts théoriques de l'architecture des ordinateurs. En construisant un émulateur à partir de zéro, nous avons pu :

— Comprendre concrètement le rôle du bus système et du mappage mémoire.

— Appréhender la complexité du décodage d'instructions et de la gestion des registres.

— Faire le lien entre le code assembleur et son exécution binaire.

Le système final est fonctionnel et respecte les exigences initiales, capable d'exécuter un programme de test utilisant un périphérique mémoire-mappé (CharOut) et de s'arrêter proprement sur `ebreak`.