

Algorithmique et programmation C

Projet C - Émulateur MIPS

# Table des matières

---

<b>I Introduction</b>	<b>3</b>
I.1 mpType .....	3
I.2 mpError .....	3
<b>II Compilation</b>	<b>5</b>
II.1 mpString .....	5
II.2 mpToken .....	6
• La ponctuation .....	7
• Les nombres .....	7
• Les registres .....	8
• Les instructions (mnémonique) .....	8
• Autres .....	9
II.3 mpInstruction .....	9
II.4 mpTranspiler .....	11
<b>III Mémoire et Registres</b>	<b>13</b>
III.1 mpRegister .....	13
III.2 mpMemory .....	14
<b>IV Émulateur</b>	<b>16</b>
IV.1 mpEmulator .....	16
• Exécution d'une instruction .....	16
• Mode simple .....	16
• Mode pas à pas .....	16
• Mode interactif .....	17
IV.2 main .....	18
<b>V Conclusion</b>	<b>19</b>

# I. Introduction

---

Fichiers communs à tout le projet : (à réécrire)

## I.1 mpType

Pour éviter les inclusions circulaires, un fichier d'en-tête `mpType.h` est disponible pour y déclarer la grande majorité des types dont les différents modules auront besoin.

## I.2 mpError

L'émulateur gère une grande quantité d'erreur, ce module est composé d'une fonction `mpRaise` pour faire remonter les erreurs à l'utilisateur :

`mpError.h`

```
void mpRaise(mpError* error);
```

`mpError` est une structure composée en partie de : un type, un code, la ligne à afficher où occure l'erreur, etc. Tous ces membres sont utilisables seulement si le champ `success` est à true.

`mpType.h`

```
typedef struct
{
    bool          success;

    mpErrorType   type;
    mpErrorCode   code;

    mpString      context;
    mpString      line;
    mpChar*       col;
    size_t        row;
}
mpError;
```

Ci-dessous quelques erreurs implémentées par l'émulateur :

`mpType.h`

```
typedef int mpErrorCode;

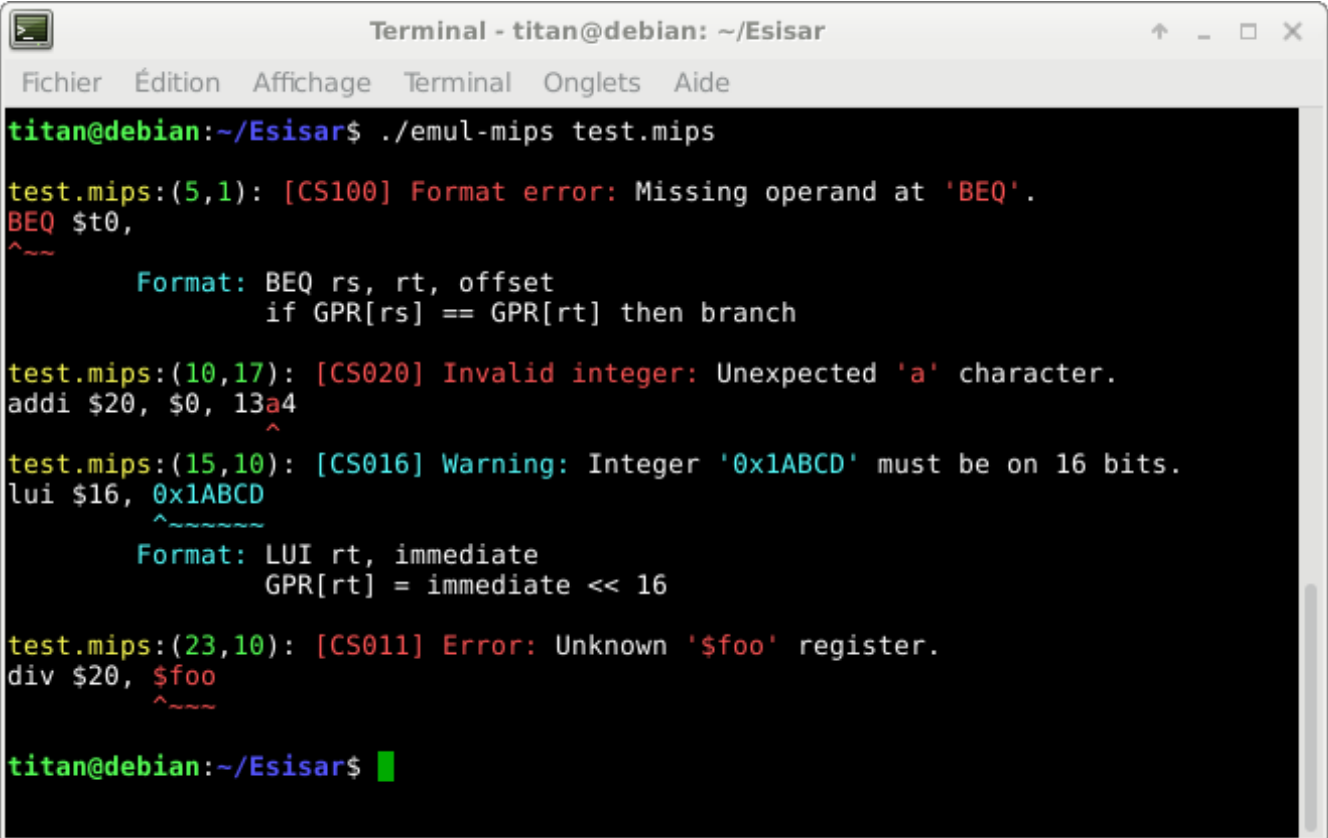
typedef enum
{
    mpErrorUNEXPECTED_CHAR ,
    mpErrorUNEXPECTED_WORD ,
    ...
}
```

```

    mpErrorINTEGER_FORMAT,
    mpErrorINTEGER_OVERFLOW,
    ...
    mpErrorUNKNOWN_REGISTER,
    mpErrorMISSING_OPERAND,
    ...
    mpErrorEMULATOR_LW_OUT_MEMORY,
    mpErrorEMULATOR_SW_READ_ONLY,
    mpErrorEMULATOR_SW_OUT_MEMORY,
}
mpErrorType;

```

**Exemple :** Affichage d'erreur sur la console.



The screenshot shows a terminal window titled "Terminal - titan@debian: ~/Esisar". The user has run the command `./emul-mips test.mips`. The terminal displays several error messages from the MIPS emulator, each with a line number and column number in parentheses, followed by a description of the error and the corresponding assembly instruction. The errors are:

- `test.mips:(5,1): [CS100] Format error: Missing operand at 'BEQ'.` followed by the instruction `BEQ $t0,` and a caret under the comma.
- `test.mips:(10,17): [CS020] Invalid integer: Unexpected 'a' character.` followed by the instruction `addi $20, $0, 13a4` and a caret under the 'a' in the immediate value.
- `test.mips:(15,10): [CS016] Warning: Integer '0x1ABCD' must be on 16 bits.` followed by the instruction `lui $16, 0x1ABCD` and a caret under the 'D' in the immediate value.
- `test.mips:(23,10): [CS011] Error: Unknown '$foo' register.` followed by the instruction `div $20, $foo` and a caret under the 'f' in the register name.

The terminal ends with the prompt `titan@debian:~/Esisar$` and a green cursor.

FIGURE I.1 –  
Exemple d'erreurs gérées par l'émulateur.

## II. Compilation

---

Cette partie transforme une ligne composée de mots en valeurs sémantique, pour ensuite en déduire si une instruction est présente et si elle est correctement écrite avec les bonnes opérandes.

### II.1 mpString

Ce petit module définit quelques fonctions usuelles sur les caractères et chaînes de caractères (Comme une fonction `mpStrCiCmp` pour comparer deux chaînes sans tenir compte de la casse, ou `mpGetFilename` qui retourne le nom du fichier pour un chemin donné). Mais aussi principalement une fonction `mpGetLine` qui retourne dans une `mpLine` la ligne courante d'un fichier sans tenir compte des commentaires.

#### mpString.h

```
bool
mpGetLine(
    FILE*    file,
    mpLine*  line,
    mpError* error
);
```

Cette fonction peut aussi retourner une erreur. En l'occurrence la seule erreur possible ici est `mpErrorLINE_OVERFLOW` quand la ligne courante du fichier est plus grande que le buffer de la ligne passée en paramètre. Pour pallier (entres autres) ce problème, une liste dynamique a été développée fonctionnant de la même manière qu'un `std::vector` en C++ et permettrait d'ajouter bien plus de caractères qu'actuellement. Cela est dû au fait que `mpLine` est une structure contenant un tableau fini de caractères de dimension `mpLINE_MAX` accompagné un entier non signé pour la taille :

#### mpType.h

```
#define mpLINE_MAX 81u

typedef struct
{
    mpChar text[mpLINE_MAX];
    size_t length;
}
mpLine;
```

La liste dynamique développée est disponible dans le code source et fonctionne aussi avec un type `template` comme en C++. Malheureusement le temps n'aura pas permis une implémentation pour `mpLine`.

Toutefois, la fonction `mpGetLine` possède un (autre) petit défaut : si le fichier a été enregistré sous Windows les retours à la ligne seront codés avec les caractères `"\r\n"` et la fonction prendra ces deux caractères comme **deux** retours à la ligne (au lieu d'un). Le problème est juste visuel car si une erreur occure sur une ligne l'affichage du numéro sur la console ne correspond pas.

## II.2 mpToken

Le module `mpToken` est important car il permet d'associer aux mots d'une ligne une étiquette et une valeur (respectivement `mpTag` et `mpValue`) dans un jeton (`mpToken`) :

`mpType.h`

```
typedef enum
{
    mpTagEND      = -1,
    mpTagNONE     = 0,

    mpTagLEFT_BRACKET ,
    mpTagRIGHT_BRACKET ,
    mpTagCOMMA ,
    mpTagCOLON ,

    mpTagINSTRUCTION ,
    mpTagREGISTER ,
    mpTagINTEGER ,
    mpTagLABEL
}
mpTag;
```

`mpType.h`

```
typedef union
{
    char    character;
    int     integer;
    void*   pointer;
}
mpValue;
```

`mpType.h`

```
typedef struct
{
    mpTag    tag;
    mpValue  value;
    mpChar*  word;
}
mpToken;
```

`mpTag` est une énumération des différents types possibles, par exemple une parenthèse, un nombre ou bien encore un registre. `mpValue` est la valeur associée (si elle existe) au `mpTag`. Le dernier membre `word` de `mpToken` est juste un pointeur sur le début du mot à la ligne `mpLine` correspondant, mais ne modifie pas cette dernière pour ajouter un caractère de fin `'\0'`.

Pour extraire un `mpToken` d'une `mpLine` on appelle la fonction `mpFetchToken` qui retourne `true` tant qu'il a des mots à trouver dans la ligne. Chaque token est placé dans une liste et cette liste sera interprétée par le `mpTranspiler` de la partie II.4 page 11.

`mpToken.h`

```
bool
mpFetchToken(
    mpFetch* fetch,
    mpError* error
);
```

`mpToken.h`

```
void
mpFetchInit(
    mpFetch* fetch,
    mpLine* line
);
```

`mpFetch` est une structure permettant de garder le contexte du parcours d'une ligne tout en retournant un `mpToken` à chaque fois qu'un mot est décodé, il faut impérativement initialiser cette structure avec `mpFetchInit` avant de l'utiliser.

Exemple d'utilisation :

```
mpFetchInit(&fetch, &line);
```

```
while (mpFetchToken(&fetch, &error))
{
    if (error.success)
    {
        // Faire quelque chose avec fetch.token
    }
}
```

- La ponctuation

Comme on peut le voir dans `mpTag`, la ponctuation de l'émulateur MIPS comprend les parenthèses ouvrantes '(' et fermantes ')', ainsi que les virgules ',' et les deux-points ':' (Pour les labels, bien qu'ils n'aient pas été implémentés). Si une autre ponctuation est trouvée alors une erreur sera remontée à l'utilisateur et l'émulation du programme demandée n'aura pas lieu.

Aucune valeur n'est placée dans le champ `mpValue` du `mpToken` car le `mpTag` se suffit à lui-même. Ci-dessous un exemple d'erreur affichée à l'utilisateur :

```
Terminal - titan@debian: ~/Esisar
Fichier  Édition  Affichage  Terminal  Onglets  Aide
titan@debian:~/Esisar$ ./emul-mips test.mips
test.mips:(19,7): [CS042] Error: Unexpected '!' character.
LW $17!, -4($sp)
titan@debian:~/Esisar$
```

- Les nombres

Les nombres peuvent être écrits en base binaire, octal, décimal et hexadécimal avec en option le caractère '-' ou '+' pour indiquer le signe.

- **Base 2 :**

Commence par le suffixe "0b" suivi des chiffres '0' et '1'. Le type d'erreur associé est `mpErrorBINARY_FORMAT`.

- **Base 8 :**

Commence obligatoirement par un zéro '0' (comme en C) suivi des chiffres entre '0' et '7'. Une erreur de type `mpErrorOCTAL_FORMAT` est remontée dans le cas contraire.

- **Base 10 :**

Contient les chiffres entre '0' et '9' mais ne doit pas commencer par un '0' sinon il sera décodé comme un nombre en base octal. Le type d'erreur associé est `mpErrorINTEGER_FORMAT`.

- **Base 16 :**

Commence par le suffixe "0x" avec les chiffres entre '0' et '9' et les caractères entre 'A' et 'F', le décodage est insensible à la casse. Le type d'erreur associé est `mpErrorHEX_FORMAT`.

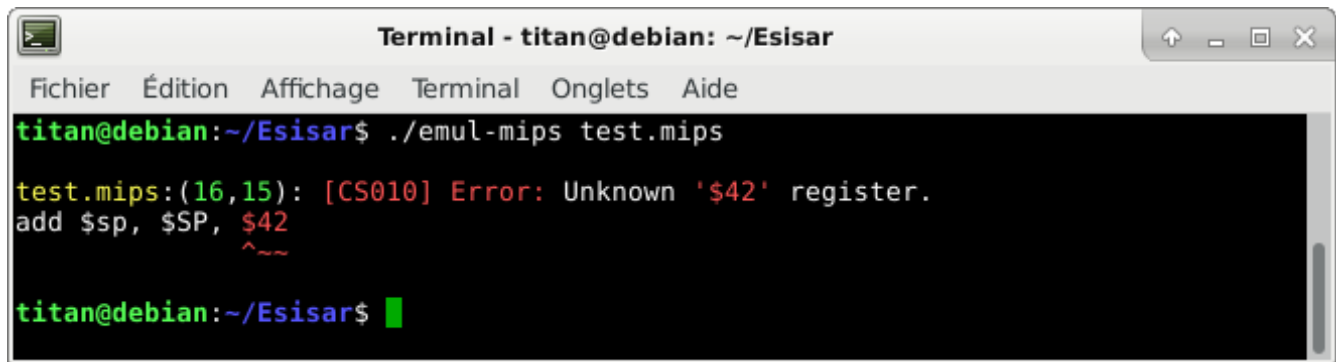
À ce stade de la compilation aucune vérification n'est faite sur le nombre maximum de bits autorisés pour représenter un nombre. Si aucune erreur de syntaxe n'est détectée on définit le `mpTag` comme un `mpTagINTEGER`, et on stocke la valeur décodée dans le membre `integer` de `mpValue`.

Exemple :

```
token.tag          = mpTagINTEGER;
token.value.integer = value; // Valeur décodée auparavant
```

- Les registres

Les registres commencent tous par un dollar '\$' et sont suivis soit d'un nombre en 0 et 31, soit de leur mnémonique associée (par exemple `a0`, `a1`, `sp`, `ra`, etc) sans tenir compte de la casse. Si le registre indiqué n'est pas valide une erreur sera affichée à l'utilisateur et l'émulation du programme ne se lancera pas. Exemple d'erreur :

A screenshot of a terminal window titled "Terminal - titan@debian: ~/Esisar". The window has a menu bar with "Fichier", "Édition", "Affichage", "Terminal", "Onglets", and "Aide". The terminal shows the command `./emul-mips test.mips` being executed. The output is `test.mips:(16,15): [CS010] Error: Unknown '$42' register.` followed by the assembly instruction `add $sp, $SP, $42` with a red squiggly line under the `$42`. The prompt `titan@debian:~/Esisar$` is visible at the bottom.

De la même façon que pour les nombres on définit les valeurs du `mpToken` comme suit :

Exemple :

```
token.tag          = mpTagREGISTER;
token.value.integer = theRegister; // numéro du registre décodée auparavant
```

- Les instructions (mnémonique)

Les instructions valides sont stockées dans une liste qui sera expliquée dans la partie II.4. Les instructions sont triées par ordre alphabétique de leur mnémonique et une fonction `mpGetInstruction` regarde si un mot donné est un mnémonique valide avec une recherche dichotomique (sans tenir compte de la casse).

À la différence des autres types, on place dans la valeur `mpValue` du `mpToken` un pointeur vers une structure `mpInstruction` qui sera expliqué dans la partie II.3, ce pointeur est retourné par la fonction `mpGetInstruction` en cas de succès. Si le pointeur est `NULL` et que le mot ne correspond à aucun autres types, et qu'il est composé uniquement de caractères alpha-numériques, alors il est considéré comme un label.

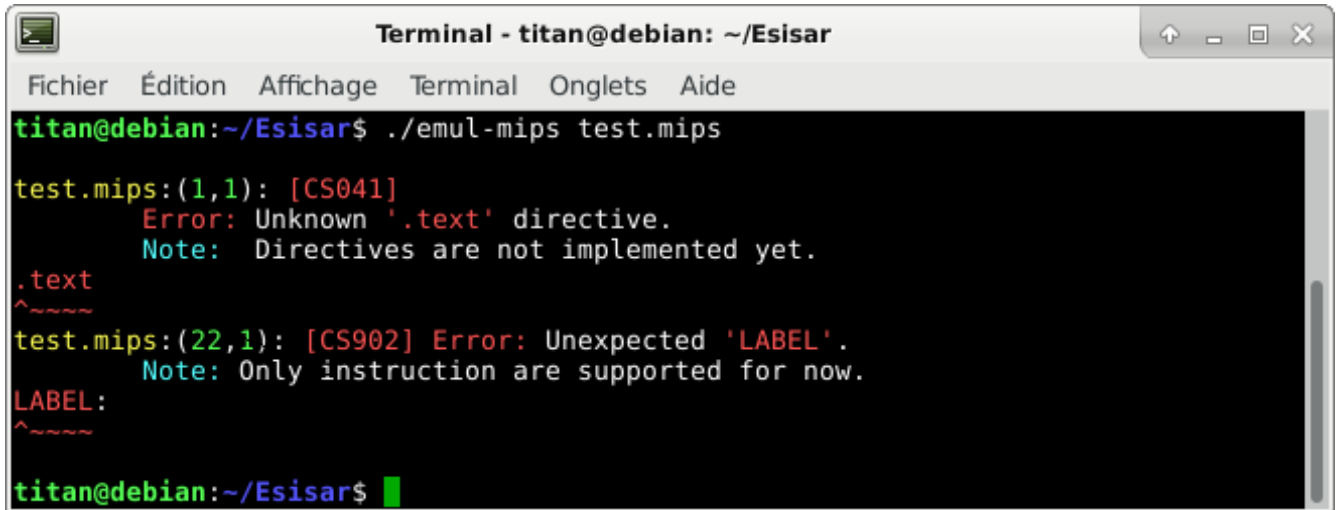


Exemple :

```
token.tag          = mpTagINSTRUCTION;
token.value.pointer = (void*) instruction;
```

- Autres

Si un mot commence par un point '.' il est alors considéré comme une directive, mais une erreur sera directement remontée à l'utilisateur car elles ne sont pas supportées. De même pour les labels :



```
Terminal - titan@debian: ~/Esisar
Fichier  Édition  Affichage  Terminal  Onglets  Aide
titan@debian:~/Esisar$ ./emul-mips test.mips
test.mips:(1,1): [CS041]
    Error: Unknown '.text' directive.
    Note: Directives are not implemented yet.
.text
^~~~~
test.mips:(22,1): [CS902] Error: Unexpected 'LABEL'.
    Note: Only instruction are supported for now.
LABEL:
^~~~~
titan@debian:~/Esisar$
```

## II.3 mpInstruction

Ce module s'occupe de convertir la liste de `mpToken` créée dans la partie II.2 en une instruction de 32 bits. Tout d'abord on définit deux types, une structure `mpInstruction` qui définit une instruction avec un `opcode`, une mnémonique, une description accompagnée d'un format (ces deux derniers servent uniquement à l'affichage) et une fonction `mpToMemory` qui va écrire dans l'instruction dans la mémoire.

### mpInstruction.h

```
typedef void
(*mpToMemory)(
    mpInstruction*    inst,
    mpVector_mpToken* tokens,
    mpMemory*         memory,
    mpError*          error
);
```

### mpInstruction.h

```
typedef struct
{
    int          opcode;
    mpCString    mnemonic;
    mpToMemory    ToMemory;

    mpCString    format;
    mpCString    description;
}
mpInstruction;
```

Ensuite une liste complète des 26 instructions supportées est définie :

mpInstruction.c

```
static
mpInstruction const s_instruction[mpINSTRUCTION_MAX] =
{
    {
        mpADD_OPCODE,
        "ADD",
        mpRtype_ORRR,
        "rd, rs, rt",
        "GPR[rd] = GPR[rs] + GPR[rt]"
    },
    {
        mpADDI_OPCODE,
        "ADDI",
        mpItype_ORRI,
        "rt, rs, immediate",
        "GPR[rt] = GPR[rs] + immediate"
    },
    ...
};
```

Avant d'écrire dans la mémoire, les fonctions `mpToMemory` vérifient si les étiquettes `mpTag` de la liste de `mpToken` correspondent bien à une syntaxe valide. Prenons l'exemple avec de l'instruction ADD décrite ci-dessus, on observe que la structure contient une fonction `mpRtype_ORRR` :

mpInstruction.c

```
static
mpTag const s_ORRR[] =
{
    mpTagINSTRUCTION,
    mpTagREGISTER,
    mpTagCOMMA,
    mpTagREGISTER,
    mpTagCOMMA,
    mpTagREGISTER,
    mpTagEND
};
```

mpInstruction.c

```
void
mpRtype_ORRR (
    mpInstruction* inst,
    mpVector_mpToken* tokens,
    mpMemory* memory,
    mpError* error
) {
    mpMatchTag(
        tokens, s_ORRR, error);

    if (error->success)
    {
        mpWriteRtype(memory, ...);
    }
}
```

Dans un premier temps cette fonction va comparer les `mpTag` de la liste de `mpToken` avec un tableau de `mpTag` prédéfini. Si cette comparaison est un succès au `mpTag` près sans un en trop ou en moins, alors une fonction s'occupe de récupérer les valeurs `mpValue` pour écrire correctement l'instruction sur 32 bits. Cette dernière fonction `mpWriteInstruction` sera expliquée dans le chapitre III page 13 sur la mémoire et les registres.

mpInstruction.c

```

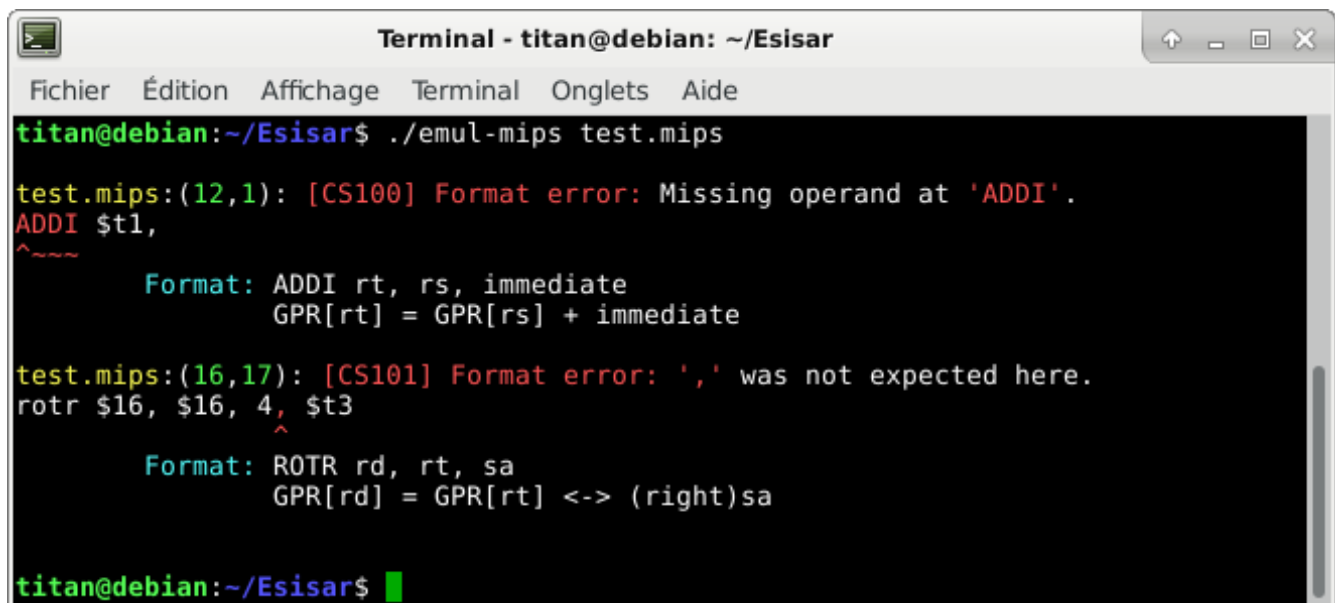
static
void
mpWriteRtype (
    mpMemory* memory,
    int op, int sa,
    int rd, int rs, int rt
) {
    uint32_t hex = 0;

    hex |= (uint32_t)((op & 0x3F));
    hex |= (uint32_t)((sa & 0x1F) << 6);
    hex |= (uint32_t)((rd & 0x1F) << 11);
    hex |= (uint32_t)((rt & 0x1F) << 16);
    hex |= (uint32_t)((rs & 0x1F) << 21);

    mpWriteInstruction(memory, hex);
}

```

Quelques exemples d'erreurs :



```

Terminal - titan@debian: ~/Esisar
Fichier  Édition  Affichage  Terminal  Onglets  Aide
titan@debian:~/Esisar$ ./emul-mips test.mips
test.mips:(12,1): [CS100] Format error: Missing operand at 'ADDI'.
ADDI $t1,
^~~~~
    Format: ADDI rt, rs, immediate
            GPR[rt] = GPR[rs] + immediate
test.mips:(16,17): [CS101] Format error: ',' was not expected here.
rotr $16, $16, 4, $t3
                ^
    Format: ROTR rd, rt, sa
            GPR[rd] = GPR[rt] <-> (right)sa
titan@debian:~/Esisar$

```

Cette méthode est répétée autant de fois pour chacune des 26 instructions avec des fonctions qui leur sont adaptées. Cette architecture permet une grande maintenabilité avec beaucoup de factorisations de code mais n'est cependant pas tant maléable que ça.

## II.4 mpTranspiler

Le mpTranspiler fait la jonction entre tous les modules de ce chapitre. La fonction mpTranspiler prend en paramètres le nom du fichier à émuler et une mémoire où écrire les instructions :

mpTranspiler.h

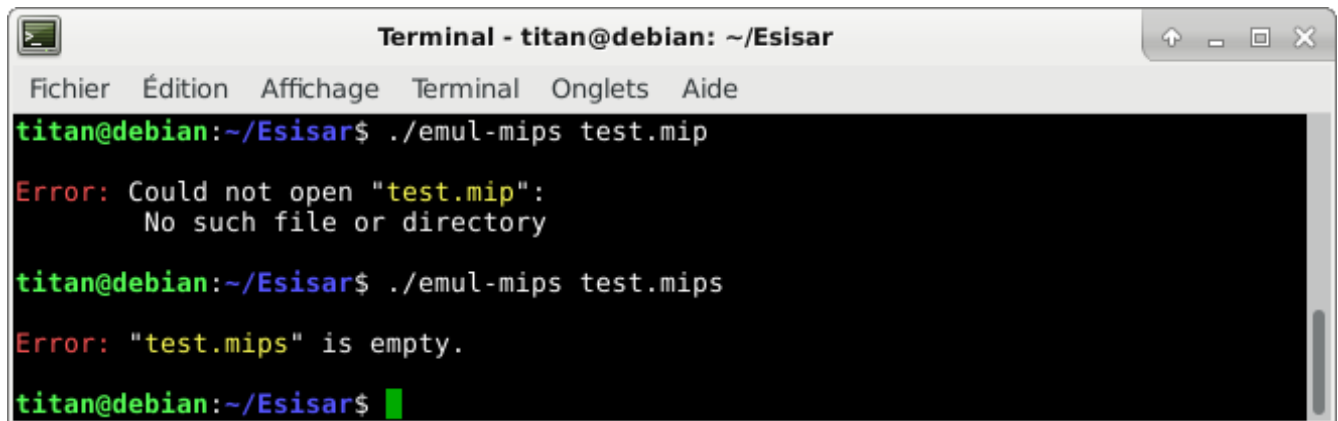
```
void
mpTranspiler(
    mpCString filename,
    mpMemory* memory,
    mpError* error
);
```

Des sous-routines existent, une pour ouvrir le fichier et traiter les erreurs, une autres pour récupérer proprement une ligne dans le fichier, et une dernière pour récupérer une liste de `mpToken` et l'envoyer à la mémoire. Cette dernière fonction présentée ci-dessous est importante car elle est réutilisée pour le mode interactif :

#### mpTranspiler.h

```
void
mpTranspiler_FetchToken(
    mpLine* line,
    mpMemory* memory,
    mpError* error
);
```

Ces fonction peuvent aussi faire remonter des erreurs, comme annoncer que le fichier envoyé n'existe pas ou encore que le fichier est vide :



The image shows a terminal window titled "Terminal - titan@debian: ~/Esisar". The window has a menu bar with "Fichier", "Édition", "Affichage", "Terminal", "Onglets", and "Aide". The terminal output shows two commands and their corresponding error messages:

```
titan@debian:~/Esisar$ ./emul-mips test.mip
Error: Could not open "test.mip":
       No such file or directory

titan@debian:~/Esisar$ ./emul-mips test.mips
Error: "test.mips" is empty.

titan@debian:~/Esisar$
```

## III. Mémoire et Registres

---

Bla bla

### III.1 mpRegister

La gestion de registres est très simple, il s'agit tout simplement d'un tableau de 32 registre + 4 registres spéciaux :

mpRegister.h

```
#define mpUSUAL_REGISTER    32u
#define mpSPECIAL_REGISTER 4u

typedef union
{
    uint32_t    GPR[mpUSUAL_REGISTER + mpSPECIAL_REGISTER];
    mpRegister_ all;
}
mpRegister;
```

Ce tableau est placé dans un `union` avec une structure composée de tous les registres et leur mnémonique. De cette façon, on peut accéder à un registre par deux moyens, soit utilisant le tableau `GPR`, soit par la structure `all`, cette fonctionnalité a pour seul but d'apporter une meilleur lisibilité du code quand on souhaite accéder à un registre en particulier, sinon on utilisera le tableau.

mpRegister.h

```
typedef struct
{
    /* Registres */
    uint32_t zero;
    uint32_t at;
    uint32_t v0, v1;
    uint32_t a0, a1, a2, a3;
    uint32_t t0, t1, t2, t3, t4, t5, t6, t7;
    uint32_t s0, s1, s2, s3, s4, s5, s6, s7;
    uint32_t t8, t9;
    uint32_t k0, k1;
    uint32_t gp, sp, fp, ra;
    /* Registres spéciaux */
    uint32_t pc, ir;
    uint32_t hi, lo;
}
mpRegister_;
```

On accompagne les registres d'une fonction pour les afficher proprement à l'utilisateur :

```

$0 00 [00000000] $1 at [00000000] $2 v0 [00000000] $3 v1 [00000000]
$4 a0 [00000000] $5 a1 [00000000] $6 a2 [00000000] $7 a3 [00000000]
$8 t0 [0000000A] $9 t1 [00000000] $10 t2 [0000002D] $11 t3 [00000000]
$12 t4 [00000000] $13 t5 [00000000] $14 t6 [00000000] $15 t7 [00000000]
$16 s0 [DABCD002] $17 s1 [00000000] $18 s2 [00000000] $19 s3 [00000000]
$20 s4 [0000000D] $21 s5 [00000005] $22 s6 [00000000] $23 s7 [00000000]
$24 t8 [00000000] $25 t9 [00000000] $26 k0 [00000000] $27 k1 [00000000]
$28 gp [00000000] $29 sp [000003AC] $30 fp [00000000] $31 ra [00000000]
$32 pc [00000044] $33 ir [0295001A] $34 hi [00000003] $35 lo [00000002]

```

### III.2 mpMemory

La mémoire se compose simplement d'un tableau d'octets avec un intervalle [ `minAddress`, `maxAddress` ] pour autoriser la lecture et l'écriture, le reste étant en lecture seule. Le choix d'avoir un tableau d'octet (8 bits) a été préféré par rapport à un tableau d'entier (32 bits) pour coller au plus à un processeur MIPS. `minAddress` est initialisée à 0 avant la partie compilation puis s'incrémente de 4 à chaque ajout d'instruction,

#### mpMemory.h

```

#define mpMEMORY_MAX 1024

typedef struct
{
    uint8_t  octet[mpMEMORY_MAX];
    uint32_t minAddress;
    uint32_t maxAddress;
}
mpMemory;

```

Sont fournis avec la mémoire deux fonctions, une pour la lecture et l'autre pour l'écriture, et dans lesquelles est testé la valeur `address` avec les bornes de la mémoire. Le cas échéant une erreur est remontée et l'émulation est stoppée. La fonction `mpWriteInstruction` correspondant à un `mpStoreWord` à l'adresse `minAddress` tout en l'incrémentant de 4 après.

#### mpMemory.h

```

uint32_t
mpLoadWord (
    mpMemory* memory,
    uint32_t  address,
    mpError*  error
);

```

#### mpMemory.h

```

void
mpStoreWord (
    mpMemory* memory,
    uint32_t  address,
    uint32_t  value,
    mpError*  error
);
mpInstruction;

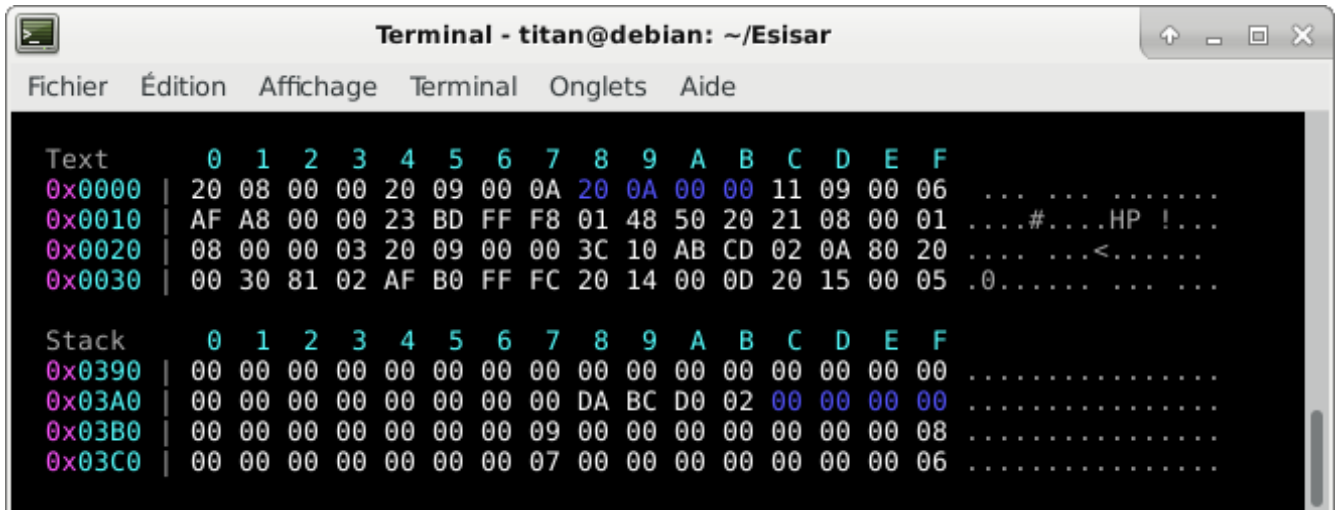
```

Il existe une constante `mpBIG_ENDIAN` qui permet d'indiquer dans quelle ordre on souhaite écrire nos données en mémoire. Si cette constante est à 1 alors on écrit les données en *big endian* sinon en *little endian*.

#### mpMemory.h

```
#define mpBIG_ENDIAN 1
```

La mémoire est aussi accompagnée d'une fonction pour l'afficher à l'utilisateur. Cette fonction prend en paramètre une borne d'adresse sur laquelle centrer le contenu. Ci-dessous, le premier bloc est centré sur le PC et le deuxième sur le SP.



The screenshot shows a terminal window titled "Terminal - titan@debian: ~/Esisar". It displays a memory dump with two sections: "Text" and "Stack". Each section shows a range of memory addresses and their corresponding hexadecimal values, with some values converted to ASCII characters.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
Text																	
0x0000	20	08	00	00	20	09	00	0A	20	0A	00	00	11	09	00	06	... ..
0x0010	AF	A8	00	00	23	BD	FF	F8	01	48	50	20	21	08	00	01	....#....HP !...
0x0020	08	00	00	03	20	09	00	00	3C	10	AB	CD	02	0A	80	20	....<.....
0x0030	00	30	81	02	AF	B0	FF	FC	20	14	00	0D	20	15	00	05	.0.....
Stack																	
0x0390	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0x03A0	00	00	00	00	00	00	00	00	DA	BC	D0	02	00	00	00	00	.....
0x03B0	00	00	00	00	00	00	00	09	00	00	00	00	00	00	00	08	.....
0x03C0	00	00	00	00	00	00	00	07	00	00	00	00	00	00	00	06	.....

## IV. Émulateur

### IV.1 mpEmulator

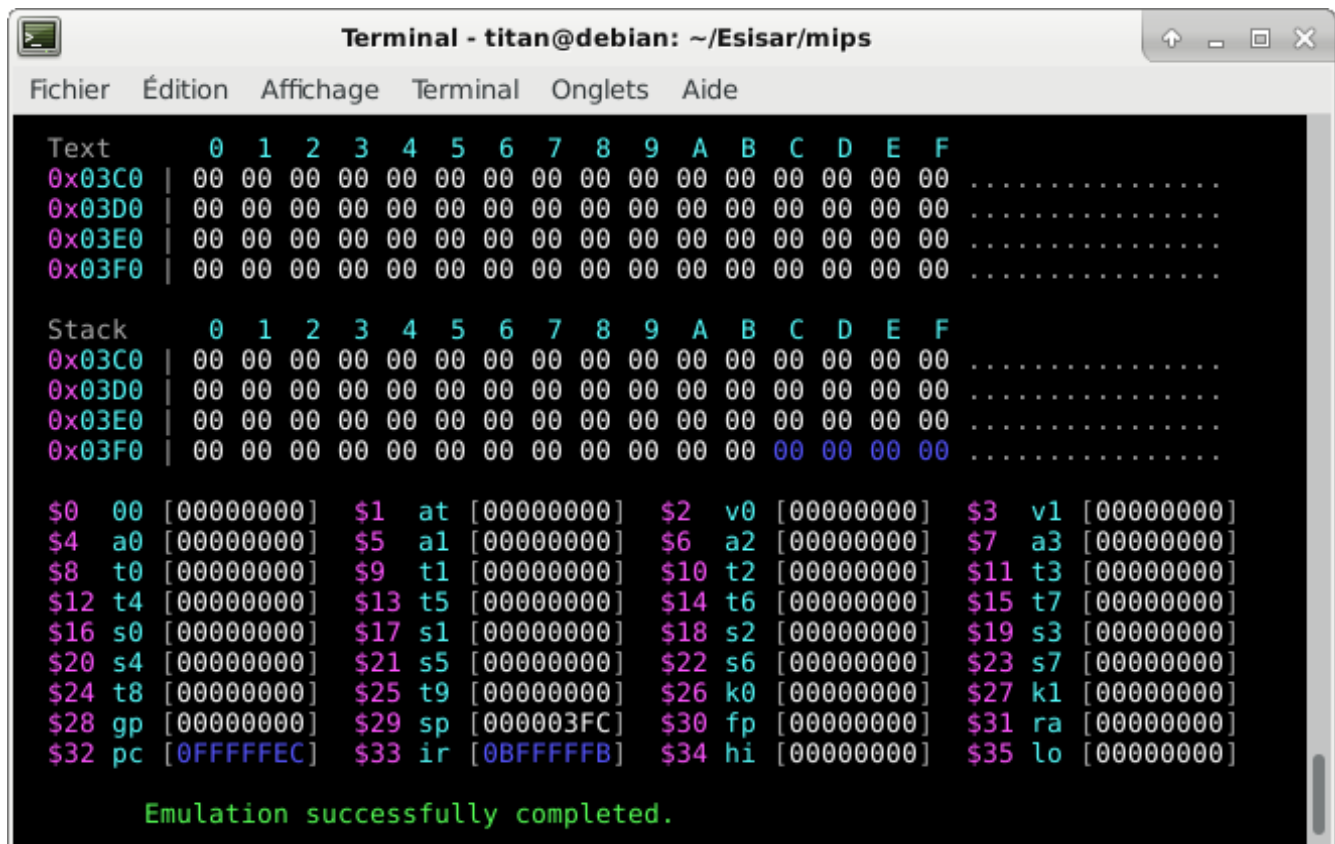
dzadza dza

- Exécution d'une instruction

...

- Mode simple

Ce mode permet d'émuler un programme écrit en MIPS et d'avoir directement le résultat à afficher sur la console.



```
Terminal - titan@debian: ~/Esisar/mips
Fichier  Édition  Affichage  Terminal  Onglets  Aide

Text
0x03C0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x03D0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x03E0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x03F0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

Stack
0x03C0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x03D0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x03E0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x03F0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

$0 00 [00000000] $1 at [00000000] $2 v0 [00000000] $3 v1 [00000000]
$4 a0 [00000000] $5 a1 [00000000] $6 a2 [00000000] $7 a3 [00000000]
$8 t0 [00000000] $9 t1 [00000000] $10 t2 [00000000] $11 t3 [00000000]
$12 t4 [00000000] $13 t5 [00000000] $14 t6 [00000000] $15 t7 [00000000]
$16 s0 [00000000] $17 s1 [00000000] $18 s2 [00000000] $19 s3 [00000000]
$20 s4 [00000000] $21 s5 [00000000] $22 s6 [00000000] $23 s7 [00000000]
$24 t8 [00000000] $25 t9 [00000000] $26 k0 [00000000] $27 k1 [00000000]
$28 gp [00000000] $29 sp [000003FC] $30 fp [00000000] $31 ra [00000000]
$32 pc [0FFFFFFC] $33 ir [0BFFFFFF] $34 hi [00000000] $35 lo [00000000]

Emulation successfully completed.
```

- Mode pas à pas

Ce mode reprend le mode simple mais avec la possibilité d'exécuter les instructions pas à pas, c'est-à-dire en appuyant sur la touche ENTRÉE pour passer à une instruction suivante.



```

Terminal - titan@debian: ~/Esisar/mips
Fichier  Édition  Affichage  Terminal  Onglets  Aide

Text      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0x0000 | 20 08 00 00 20 09 00 0A 20 0A 00 00 11 09 00 06 .....
0x0010 | AF A8 00 00 23 BD FF F8 01 48 50 20 21 08 00 01 ....#....HP !...
0x0020 | 08 00 00 03 20 09 00 00 3C 10 AB CD 02 0A 80 20 ....<.....
0x0030 | 00 30 81 02 AF B0 FF FC 8F B1 FF FC 20 14 00 0D .0.....

Stack     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0x03C0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x03D0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x03E0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 .....
0x03F0 | 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 .....

$0 00 [00000000] $1 at [00000000] $2 v0 [00000000] $3 v1 [00000000]
$4 a0 [00000000] $5 a1 [00000000] $6 a2 [00000000] $7 a3 [00000000]
$8 t0 [00000002] $9 t1 [0000000A] $10 t2 [00000001] $11 t3 [00000000]
$12 t4 [00000000] $13 t5 [00000000] $14 t6 [00000000] $15 t7 [00000000]
$16 s0 [00000000] $17 s1 [00000000] $18 s2 [00000000] $19 s3 [00000000]
$20 s4 [00000000] $21 s5 [00000000] $22 s6 [00000000] $23 s7 [00000000]
$24 t8 [00000000] $25 t9 [00000000] $26 k0 [00000000] $27 k1 [00000000]
$28 gp [00000000] $29 sp [000003EC] $30 fp [00000000] $31 ra [00000000]
$32 pc [00000014] $33 ir [AFA80000] $34 hi [00000000] $35 lo [00000000]

Text[pc] = ADDI $sp, $sp, -8
Press [ENTER] to continue...

```

- Mode interactif

Le mode interactif laisse la possibilité à l'utilisateur d'entrer ses instructions à souhait.

```

Terminal - titan@debian: ~/Esisar/mips
Fichier  Édition  Affichage  Terminal  Onglets  Aide

Text      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0x0000 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0010 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0020 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0030 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

Stack     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0x03C0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x03D0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x03E0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x03F0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

$0  00 [00000000]  $1  at [00000000]  $2  v0 [00000000]  $3  v1 [00000000]
$4  a0 [00000000]  $5  a1 [00000000]  $6  a2 [00000000]  $7  a3 [00000000]
$8  t0 [00000000]  $9  t1 [00000000]  $10 t2 [00000000]  $11 t3 [00000000]
$12 t4 [00000000]  $13 t5 [00000000]  $14 t6 [00000000]  $15 t7 [00000000]
$16 s0 [00000000]  $17 s1 [00000000]  $18 s2 [00000000]  $19 s3 [00000000]
$20 s4 [00000000]  $21 s5 [00000000]  $22 s6 [00000000]  $23 s7 [00000000]
$24 t8 [00000000]  $25 t9 [00000000]  $26 k0 [00000000]  $27 k1 [00000000]
$28 gp [00000000]  $29 sp [000003FC]  $30 fp [00000000]  $31 ra [00000000]
$32 pc [00000000]  $33 ir [00000000]  $34 hi [00000000]  $35 lo [00000000]

emul-mips > addi $a0, $0, 0x1A

```

## IV.2 main

...

## V. Conclusion

---

Ce projet était très intéressant à réaliser. Avec plus de temps il aurait été possible de faire un projet bien plus complet, c'est pourquoi je garde mes idées dans un coin pour, pourquoi pas, un jour retravailler dessus.