

Universidade de Brasília - UnB
Departamento de Ciência da Computação – CiC



TRABALHO 2 – SIMULADOR MIPS

Disciplina: Organização e Arquitetura de Computadores
Professor: Marcelo Grandi Mandelli
Aluno: Raphael Rodrigues
Matrícula: 11/0039530
Turma: C

Objetivos

Compreender o funcionamento do processador *MIPS* e implementar um simulador de uma determinada *ISA (Instruction Set Architecture)*.

Explicação do código implementado

O código implementado visa buscar completa similaridade com o simulador *MARS*. Para isso foram necessárias desenvolver algumas funções, que serão descritas a seguir.

Função main – É a função principal do programa. Define a inicialização da memória e a chamada da função *run*. Ela realiza a inicialização de variáveis importantes do código, como por exemplo:

```
uint32_t opcode, rs, rt, rd, shamt, funct = 0;
int16_t imm16 = 0;
int32_t imm26 = 0, pc = 0;
```

Caracterizando os seguintes campos de instrução do MIPS:

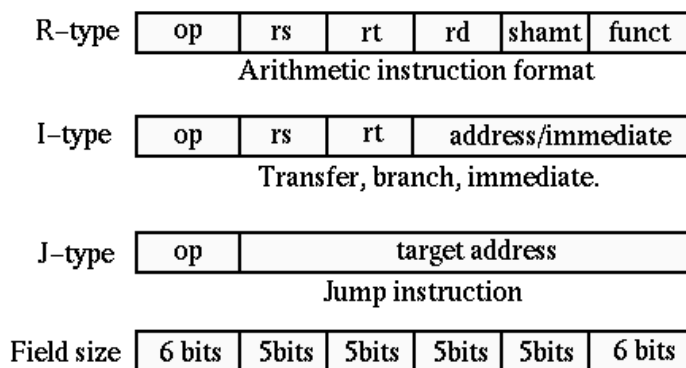


Imagem 1 – Campos de instruções do MIPS [1].

Inicialização da função:

```
int main(int argc, char *argv[]){
```

Função init_mem – Essa função, inicialmente, inicializa toda a memória de 4096 words com valores zerados. Após a primeira inicialização é feita uma leitura dos arquivos binários dados, de código e de dados, através dos argumentos 1 e 2 digitados ao executar o programa. Realizando assim o “preenchimento” da memória virtual do simulador.

Inicialização da função:

```
init_mem(argv[1], argv[2]);
```

Função run – É a segunda função mais importante do programa. É nela que estão as chamadas das funções *step*, *dump_Mem*, *dump_Reg* e *exit*. A sua principal função é rodar a simulação MIPS, fornecendo um menu de escolha de funções, como definido na especificação. Em que:

(1) Step	<Executa uma instrução>
(2) Run	<Executa todo o código e faz o dump dos registradores e da memória ao finalizar>
(3) dump_mem	<Realiza o dump de memória>
(4) dump_reg	<Realiza o dump de registradores>
(5) Exit	<Sai do programa>

Nela são utilizadas algumas variáveis chave, como por exemplo a `p_f`, o terceiro argumento da execução do programa, que indica qual será o método de execução do programa. Em que, caso seja `p` será mostrado um menu de instruções para o usuário seguir. Caso contrário, será feita toda a execução do código, mostrando somente o resultado final e a criação dos arquivos de *dump* de memória e de *dump* dos registradores. Essa função será executada enquanto não houver a chamada de encerramento forçado do programa, caso o usuário digite a opção 5, ou caso o programa encontre a chamada de programa do syscall ou caso o programa ultrapasse os 2048 endereços de *word* reservado ao campo *text*. Será utilizada também a variável chave `identificaFIM`, que é setada com valor 'y', caso seja visto que o valor de `v0` é igual a 10 e caso seja executada a instrução de syscall logo após a definição do valor de `v0`.

Inicialização da função:

```
void run(uint32_t ri, uint32_t opcode, uint32_t rs, uint32_t rt, uint32_t rd,
uint32_t shamt, uint32_t funct, int16_t imm16, int32_t imm26, char opcao, char
command[12], char identificaFIM, char p_f){
```

Função fetch – Essa função lê uma instrução da memória e coloca-a em `ri`, atualizando o `pc` para apontar para a próxima instrução.

Inicialização da função:

```
void fetch(uint32_t *ri){
```

Função step – Nela será executada cada instrução por vez, seguindo a sequência dada na memória, anteriormente inicializada através da função `init_mem`. Nessa função será feita a administração de cada instrução, verificando se é do tipo R, do tipo I ou do tipo J, conforme visto anteriormente. Seu funcionamento é simples, primeiramente ela chamará a função `fetch`. Após, será realizada a identificação da instrução retornada pela função `fetch`, através de uma condicional (do campo `opcode`), identificando todos os campos conforme a imagem 1. Com a identificação de tipo, será realizada agora a identificação da instrução a ser executada, através das chamadas de função `identificaInstrucaoTipoR` e `identificaInstrucaoTipoI_J`. E, finalmente, será realizada a função `execute`, efetivamente, executando aquela instrução.

Inicialização da função:

```
void step(uint32_t ri, uint32_t opcode, uint32_t rs, uint32_t rt, uint32_t rd,
uint32_t shamt, uint32_t funct, int16_t imm16, int32_t imm26, char opcao, char
command[12], char *identificaFIM){
```

Função decode_TIPO_R – Essa função tem como princípio e funcionalidade extrair os campos tipo R das instruções. Nela são definidos os valores `rs`, `rt`, `rd`, `shamt` e `funct`, de acordo com a imagem 1. Caso queira saber todos campos e como eles estão sendo definidos, basta tirar os comentários das linhas:

```
//    printf ("OPCODE: 0x%X\n", opcode);
//    printf ("RS: %u\n", *rs);
//    printf ("RT: %u\n", *rt);
//    printf ("RD: %u\n", *rd);
//    printf ("SHAMT: %u\n", *shamt);
//    printf ("FUNCT: 0x%X\n", *funct);
```

Que, deste modo, serão mostrados todos os valores de cada campo de cada instrução de ri.

Inicialização da função:

```
decode_TIPO_R(uint32_t opcode, uint32_t *rs, uint32_t *rt, uint32_t *rd, uint32_t
*shamt, uint32_t *funct, uint32_t ri){
```

Função decode_TIPO_I – Seu funcionamento é semelhante ao decode_TIPO_R, mudando somente os campos conforme o tipo da instrução, sendo, nesse caso, tipo I. Serão definidos os valores dos campos rs, rt e imm16 (imediato de 16 bits), conforme a imagem 1. Para que possam ser vistos os valores dos campos de cada instrução tipo I, pode-se retirar os comentários das linhas:

```
//    printf ("OPCODE: 0x%X\n", opcode);
//    printf ("RS: %u\n", *rs);
//    printf ("RT: %u\n", *rt);
//    printf ("IMM16: %d\n", *imm16);
```

Que, deste modo, serão mostrados todos os valores de cada campo de cada instrução de ri.

Inicialização da função:

```
decode_TIPO_I(uint32_t opcode, uint32_t *rs, uint32_t *rt, int16_t *imm16, uint32_t
ri){
```

Função decode_TIPO_J – Seu funcionamento é semelhante as funções de decode_TIPO_I e decode_TIPO_R, mudando somente os campos conforme o tipo da instrução, nesse caso, tipo J. Nela serão definidos os valores dos campos imm26 (imediato de 26 bits). Para que sejam vistos os valores dos campos da instrução, vindo através da variável ri, é necessário que se retire os comentários das seguintes linhas:

```
//    printf ("OPCODE: 0x%X\n", opcode);
//    printf ("IMM26: %d\n", *imm26);
```

Que, deste modo, serão mostrados todos os valores de cada campo de cada instrução de ri.

Inicialização da função:

```
decode_TIPO_J(uint32_t opcode, uint32_t *imm26, int32_t ri){
```

Função identificaInstrucaoTipoR – Essa função tem como funcionalidade apenas identificar a partir do campo funct qual será a instrução a ser executada. Ela possui como retorno a variável command, que armazena o nome da instrução. Por exemplo, caso a instrução lida seja ADD, será armazenada, então, a string ADD em command. Essa funcionalidade servirá para a função execute, que será feita a

execução conforme a variável `command`. Para que seja mostrada a instrução que está sendo executada no momento da chamada da função `step`, seja através do terceiro argumento ser `p` ou `f`, basta tirar os comentários das linhas:

```
//          printf ("ADD");
//          printf ("ADDU");
//          printf ("AND");
//          printf ("JR");
//          printf ("NOR");
//          printf ("OR");
//          printf ("SLT");
//          printf ("SLTU");
//          printf ("SLL");
//          printf ("SRL");
//          printf ("SUB");
//          printf ("SUBU");
//          printf ("SYSCALL");
//          printf ("SRA");
//          printf ("XOR");
//          printf ("DIV");
//          printf ("DIVU");
//          printf ("MFHI");
//          printf ("MFLO");
//          printf ("MULT");
//          printf ("MULTU");
```

E somente para melhor visualização, pode-se retirar, também, o comentário das linhas:

```
//      printf ("\nINSTRUÇÃO LIDA: \n");
//      printf("\n");
```

No começo e no término da função.

Inicialização da função:

```
char *identificaInstrucaoTipoR(uint32_t funct){
```

Função `identificaInstrucaoTipoI_J` – Sua funcionalidade é bastante semelhante com a função `identificaInstrucaoTipoR`, mudando somente de acordo com o tipo de instrução. Poderia ser feita juntamente com a função `identificaInstrucaoTipoR`, mas, para um entendimento mais fácil foi feita separadamente. Para que seja mostrada a instrução que está sendo executada no momento da chamada da função `step`, seja através do terceiro argumento ser `p` ou `f`, basta tirar os comentários das linhas:

```
//          printf ("ADDI");
//          printf ("ADDIU");
//          printf ("ANDI");
//          printf ("BEQ");
//          printf ("BNE");
//          printf ("LBU");
//          printf ("LHU");
//          printf ("LL");
//          printf ("LUI");
//          printf ("LW");
```

```
//      printf ("ORI");
//      printf ("XORI");
//      printf ("SLTI");
//      printf ("SLTIU");
//      printf ("SB");
//      printf ("SC");
//      printf ("SH");
//      printf ("SW");
//      printf ("LWC1");
//      printf ("LDC1");
//      printf ("SWC1");
//      printf ("SDC1");
//      printf ("J");
//      printf ("JAL");
//      printf ("LB");
//      printf ("LH");
//      printf ("BGTZ");
//      printf ("BLEZ");
```

E somente para melhor visualização, pode-se retirar, também, o comentário das linhas:

```
//      printf ("\nINSTRUÇÃO LIDA: \n");
//      printf ("\n");
```

No começo e no término da função.

Inicialização da função:

```
char *identificaInstrucaoTipoI_J(uint32_t opcode){
```

Função execute – Esta função executa a instrução lida pela função fetch e decodificada pelo função decode. Sendo assim, ela realiza cada operação conforme especificado no help do MARS:

MIPS
MARS
License
Bugs/Comments
Acknowledgements
Instruction Set Song

Operand Key for Example Instructions

label, target	any textual label
\$t1, \$t2, \$t3	any integer register
\$f2, \$f4, \$f6	even-numbered floating point register
\$f0, \$f1, \$f3	any floating point register
\$0	any Coprocessor 0 register

Basic Instructions

Extended (pseudo) Instructions

Directives

Syscalls

Exceptions

Macros

```

abs.d $f2,$f4      Floating point absolute value double precision : Set $f2 to absolute value of $f4, double precision
abs.s $f0,$f1      Floating point absolute value single precision : Set $f0 to absolute value of $f1, single precision
add $t1,$t2,$t3     Addition with overflow : set $t1 to ($t2 plus $t3)
add.d $f2,$f4,$f6   Floating point addition double precision : Set $f2 to double-precision floating point value of $f4 plus $f6
add.s $f0,$f1,$f3   Floating point addition single precision : Set $f0 to single-precision floating point value of $f1 plus $f3
addi $t1,$t2,-100   Addition immediate with overflow : set $t1 to ($t2 plus signed 16-bit immediate)
addiu $t1,$t2,-100  Addition immediate unsigned without overflow : set $t1 to ($t2 plus signed 16-bit immediate), no overflow
and $t1,$t2,$t3     Bitwise AND : Set $t1 to bitwise AND of $t2 and $t3
andi $t1,$t2,100    Bitwise AND immediate : Set $t1 to bitwise AND of $t2 and zero-extended 16-bit immediate
bcif l,label        Branch if specified FP condition flag false (BCIF, not BCLF) : If Coprocessor 1 condition flag specified by immediate is false (zero) then branch to statement at label's address
bcif l,label        Branch if FP condition flag 0 false (BCIF, not BCLF) : If Coprocessor 1 condition flag 0 is false (zero) then branch to statement at label's address
bcit l,label        Branch if specified FP condition flag true (BCIT, not BCLT) : If Coprocessor 1 condition flag specified by immediate is true (one) then branch to statement at label's address
bcit l,label        Branch if FP condition flag 0 true (BCIT, not BCLT) : If Coprocessor 1 condition flag 0 is true (one) then branch to statement at label's address
beq $t1,$t2,label   Branch if equal : Branch to statement at label's address if $t1 and $t2 are equal
bgez $t1,label      Branch if greater than or equal to zero : Branch to statement at label's address if $t1 is greater than or equal to zero
bgezal $t1,label    Branch if greater than or equal to zero and link : If $t1 is greater than or equal to zero, then set $ra to the Program Counter and branch to statement at label's address
bgtz $t1,label      Branch if greater than zero : Branch to statement at label's address if $t1 is greater than zero
blez $t1,label      Branch if less than or equal to zero : Branch to statement at label's address if $t1 is less than or equal to zero
bltz $t1,label      Branch if less than zero : Branch to statement at label's address if $t1 is less than zero
bltzal $t1,label    Branch if less than zero and link : If $t1 is less than or equal to zero, then set $ra to the Program Counter and branch to statement at label's address
bne $t1,$t2,label   Branch if not equal : Branch to statement at label's address if $t1 and $t2 are not equal
break              Break execution : Terminate program execution with exception
break 100          Break execution with code : Terminate program execution with specified exception code
c.eq.d $f2,$f4      Compare equal double precision : If $f2 is equal to $f4 (double-precision), set Coprocessor 1 condition flag 0 true else set it false
c.eq.l $f2,$f4      Compare equal double precision : If $f2 is equal to $f4 (double-precision), set Coprocessor 1 condition flag specified by immediate to true else set it to false
c.eq.s $f0,$f1      Compare equal single precision : If $f0 is equal to $f1, set Coprocessor 1 condition flag 0 true else set it false
c.eq.s l,$f0,$f1    Compare equal single precision : If $f0 is equal to $f1, set Coprocessor 1 condition flag specified by immediate to true else set it to false
c.le.d $f2,$f4      Compare less or equal double precision : If $f2 is less than or equal to $f4 (double-precision), set Coprocessor 1 condition flag 0 true else set it false
c.le.d l,$f2,$f4    Compare less or equal double precision : If $f2 is less than or equal to $f4 (double-precision), set Coprocessor 1 condition flag specified by immediate true else set it to false
c.le.s $f0,$f1      Compare less or equal single precision : If $f0 is less than or equal to $f1, set Coprocessor 1 condition flag 0 true else set it false
c.le.s l,$f0,$f1    Compare less or equal single precision : If $f0 is less than or equal to $f1, set Coprocessor 1 condition flag specified by immediate to true else set it to false
c.lt.d $f2,$f4      Compare less than double precision : If $f2 is less than $f4 (double-precision), set Coprocessor 1 condition flag 0 true else set it false
c.lt.d l,$f2,$f4    Compare less than double precision : If $f2 is less than $f4 (double-precision), set Coprocessor 1 condition flag specified by immediate to true else set it to false

```

Close

Imagem 2 – Página de ajuda da ferramenta MARS.

Ou também, de forma simplificada, através do guia de referência do MIPS:

CORE INSTRUCTION SET				OPCODE
NAME, MNEMONIC	FOR-	OPERATION (in Verilog)	/ FUNCT	
	MAT		(Hex)	
Add	add	R R[rd] = R[rs] + R[rt]	(1)	0 / 20 _{hex}
Add Immediate	addi	I R[rt] = R[rs] + SignExtImm	(1,2)	8 _{hex}
Add Imm. Unsigned	addiu	I R[rt] = R[rs] + SignExtImm	(2)	9 _{hex}
Add Unsigned	addu	R R[rd] = R[rs] + R[rt]		0 / 21 _{hex}
And	and	R R[rd] = R[rs] & R[rt]		0 / 24 _{hex}
And Immediate	andi	I R[rt] = R[rs] & ZeroExtImm	(3)	c _{hex}
Branch On Equal	beq	I if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4 _{hex}
Branch On Not Equal	bne	I if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5 _{hex}
Jump	j	J PC=JumpAddr	(5)	2 _{hex}
Jump And Link	jal	J R[31]=PC+4;PC=JumpAddr	(8) (5)	3 _{hex}
Jump Register	jr	R PC=R[rs]		0 / 08 _{hex}
Load Byte Unsigned	lbu	I R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24 _{hex}
Load Halfword Unsigned	lhu	I R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25 _{hex}
Load Linked	ll	I R[rt] = M[R[rs]+SignExtImm]	(2,7)	30 _{hex}
Load Upper Imm.	lui	I R[rt] = {imm, 16'b0}		f _{hex}
Load Word	lw	I R[rt] = M[R[rs]+SignExtImm]	(2)	23 _{hex}
Nor	nor	R R[rd] = ~(R[rs] R[rt])		0 / 27 _{hex}
Or	or	R R[rd] = R[rs] R[rt]		0 / 25 _{hex}
Or Immediate	ori	I R[rt] = R[rs] ZeroExtImm	(3)	d _{hex}
Set Less Than	slt	R R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / 2a _{hex}
Set Less Than Imm.	slti	I R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2)	a _{hex}
Set Less Than Imm. Unsigned	sltiu	I R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b _{hex}
Set Less Than Unsig.	sltu	R R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / 2b _{hex}
Shift Left Logical	sll	R R[rd] = R[rt] << shamt		0 / 00 _{hex}
Shift Right Logical	srl	R R[rd] = R[rt] >> shamt		0 / 02 _{hex}
Store Byte	sb	I M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28 _{hex}
Store Conditional	sc	I M[R[rs]+SignExtImm] = R[rt]; R[rt] = (atomic) ? 1 : 0	(2,7)	38 _{hex}
Store Halfword	sh	I M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29 _{hex}
Store Word	sw	I M[R[rs]+SignExtImm] = R[rt]	(2)	2b _{hex}
Subtract	sub	R R[rd] = R[rs] - R[rt]	(1)	0 / 22 _{hex}
Subtract Unsigned	subu	R R[rd] = R[rs] - R[rt]		0 / 23 _{hex}
Syscall	syscall	R PC = ExceptionAddr		0 / 0c _{hex}
Shift Right Arithmetic	sra	R R[rd] = R[rt] >> shamt		0 / 03 _{hex}
Exclusive Or	xor	R R[rd] = R[rs] ^ R[rt]		0 / 2b _{hex}
Move From C0	mfc0	R R[rt] = Rc0[rd]		10 / 00 _{hex}

Imagem 3 – Conjunto de instruções básicas [2].

ARITHMETIC CORE INSTRUCTION SET				② OPCODE
NAME, MNEMONIC	FOR-MAT	OPERATION	/ FMT / FT	/ FUNCT
			(Hex)	
Branch On FP True	bclt	FI if(FPcond)PC=PC+4+BranchAddr (4)	11/8/1/--	
Branch On FP False	bclf	FI if(!FPcond)PC=PC+4+BranchAddr(4)	11/8/0/--	
Divide	div	R Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	0/--/--/1a	
Divide Unsigned	divu	R Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6) 0/--/--/1b	
FP Add Single	add.s	FR F[fd]=F[fs]+F[ft]	11/10/--/0	
FP Add Double	add.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}	11/11/--/0	
FP Compare Single	c.x.s*	FR FPcond = (F[fs] op F[ft]) ? 1 : 0	11/10/--/y	
FP Compare Double	c.x.d*	FR FPcond = ({F[fs],F[fs+1]} op {F[ft],F[ft+1]}) ? 1 : 0	11/11/--/y	
* (x is eq, lt, or le) (op is ==, <, or <=) (y is 32, 3c, or 3e)				
FP Divide Single	div.s	FR F[fd]=F[fs]/F[ft]	11/10/--/3	
FP Divide Double	div.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}	11/11/--/3	
FP Multiply Single	mul.s	FR F[fd]=F[fs]*F[ft]	11/10/--/2	
FP Multiply Double	mul.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}	11/11/--/2	
FP Subtract Single	sub.s	FR F[fd]=F[fs]-F[ft]	11/10/--/1	
FP Subtract Double	sub.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}	11/11/--/1	
Load FP Single	lwc1	I F[rt]=M[R[rs]+SignExtImm]	(2) 31/--/--/--	
Load FP Double	ldc1	I F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2) 35/--/--/--	
Move From Hi	mfhi	R R[rd]=Hi	0/--/--/10	
Move From Lo	mflo	R R[rd]=Lo	0/--/--/12	
Move From Control	mfc0	R R[rd]=CR[rs]	10/0/--/0	
Multiply	mult	R {Hi,Lo} = R[rs] * R[rt]	0/--/--/18	
Multiply Unsigned	multu	R {Hi,Lo} = R[rs] * R[rt]	(6) 0/--/--/19	
Shift Right Arith.	sra	R R[rd]=R[rt]>>>shamt	0/--/--/3	
Store FP Single	swc1	I M[R[rs]+SignExtImm]=F[rt]	(2) 39/--/--/--	
Store FP Double	sdc1	I M[R[rs]+SignExtImm]=F[rt]; M[R[rs]+SignExtImm+4]=F[rt+1]	(2) 3d/--/--/--	
Move Single	mov.s	FR F[fd]=F[fs]	11/10/0/6	
Move Double	mov.d	FR F[fd]=F[fs]	11/11/0/6	
Move To C1	mtc1	FI* F[fs]=R[rt]	11/4/-/0	
Move From C1	mfc1	FI* R[rt]=F[fs]	11/0/-/0	
Convert from Y to X	cvt.x.y	FR F[fd] _x =F[fs] _y (x,y)={S,D,W}		
Square Root	sqrt.s	FR F[fd]=sqrt(F[fs])	11/10/0/4	

Imagem 4 – Conjunto de instruções aritméticas básicas [2].

Inicialização da função:

```
execute(char command[12], uint32_t rs, uint32_t rt, uint32_t rd, uint32_t shamt,
int16_t imm16, int32_t imm26){
```

Função dump_Mem – Essa função realiza o dump de acordo com o parâmetro 3, indicado na execução do programa. Caso seja p, será feito um dump de um intervalo desejado pelo usuário, caso contrário, ou seja, f, será realizado um dump no arquivo mem.txt. A variável hexa tem utilidade apenas para melhor visualização do resultado a ser apresentado. Por exemplo, caso queiramos mostrar, como saída, 0x0000FFFF será necessário que o valor de hexa seja igual a 4. Se não tivesse o tratamento do hexa seria mostrado somente o valor FFFF.

Inicialização da função:

```
void dump_Mem(int start, int end, char format, char p_f){
```

Função dump_Reg – Possui o mesmo princípio e funcionamento semelhante ao dump_Mem, mudando somente algumas características internas à função. Como por exemplo o que será mostrado, sendo os valores dos registradores.

Inicialização da função:

```
void dump_Reg(char format, char p_f){
```

Função lb – Essa função possui um funcionamento semelhante ao da função execute. Ela será executada no syscall, portanto não terá valor de imediato, mas sim, de um contador de byte. Como era necessário que o syscall utilizasse a instrução de lb para o print de caracteres, então foi-se criada a função lb separadamente e exclusivamente para a instrução syscall. Em que é feito um percorrimto dos bytes da memória enquanto não é achado o valor de 0. Pois, de acordo com o MARS, a cada final de dado na memória será criado um valor de \0, indicando o fim do dado. Portanto, seguindo esse princípio foi-se feita essa função.

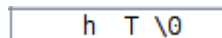


Imagem 5 – Término do caracter de espaço, \0, e início de outra string.

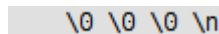


Imagem 6 – Término da string contendo o caracter '\n' indicando através do \0.

Inicialização da função:

```
lb(int *i){
```

Função exit – Realiza o encerramento do programa. A chamada da função exit(0), contida na biblioteca stdlib.h da linguagem de programação C, condiz com o encerramento bem sucedido do programa.

Compilação e execução

Para compilar o código é necessário o seguinte comando, considerando que o caminho indicado no terminal, antes de \$, esteja no mesmo caminho do código:

```
$ gcc SimuladorMIPS.c -o prog
```

Para sua execução basta utilizar o comando caso queira passo-a-passo:

```
$ ./prog <parametro1> <parametro2> p
```

Ou caso queira a execução completa, utilizar o comando:

```
$ ./prog <parametro1> <parametro2> f
```

Sendo que o campo destinado ao <parametro1> é o caminho do arquivo contendo o segmento de código do programa em assembly *.bin*, destinado ao arquivo de código. E o campo destinado ao <parametro2> é o caminho do arquivo contendo o segmento de código do programa em assembly *.bin*, destinado ao arquivo de dados.

Resultados da execução

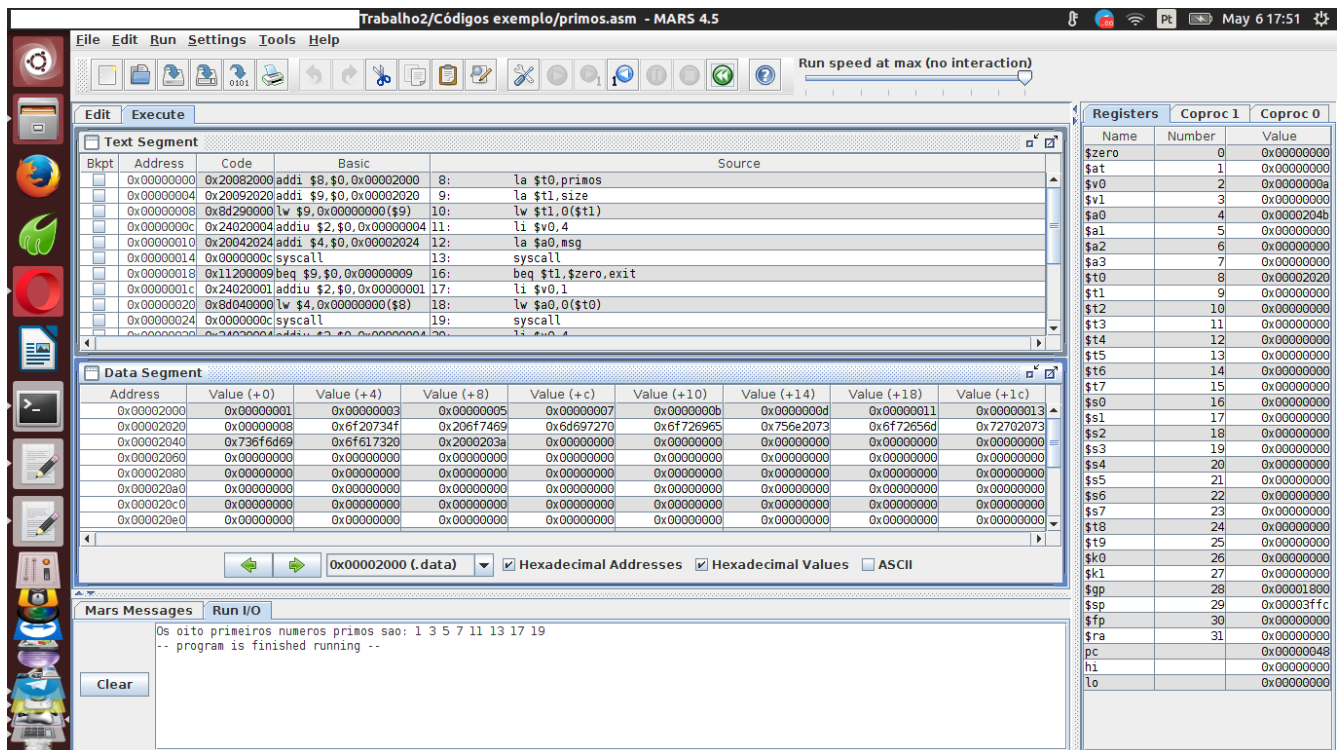


Imagem 7 – Compilação, execução e resultado no simulador MARS.

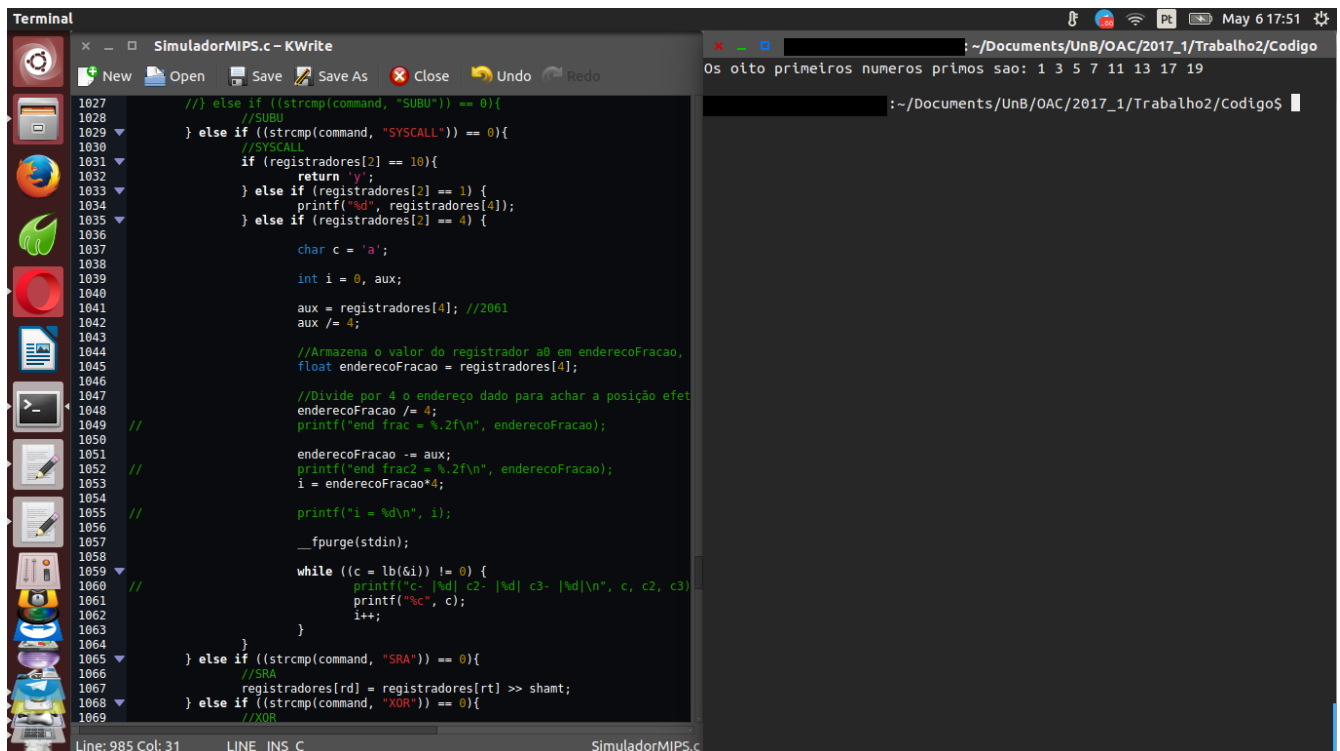


Imagem 8 – Compilação, execução e resultado do simulador implementado.

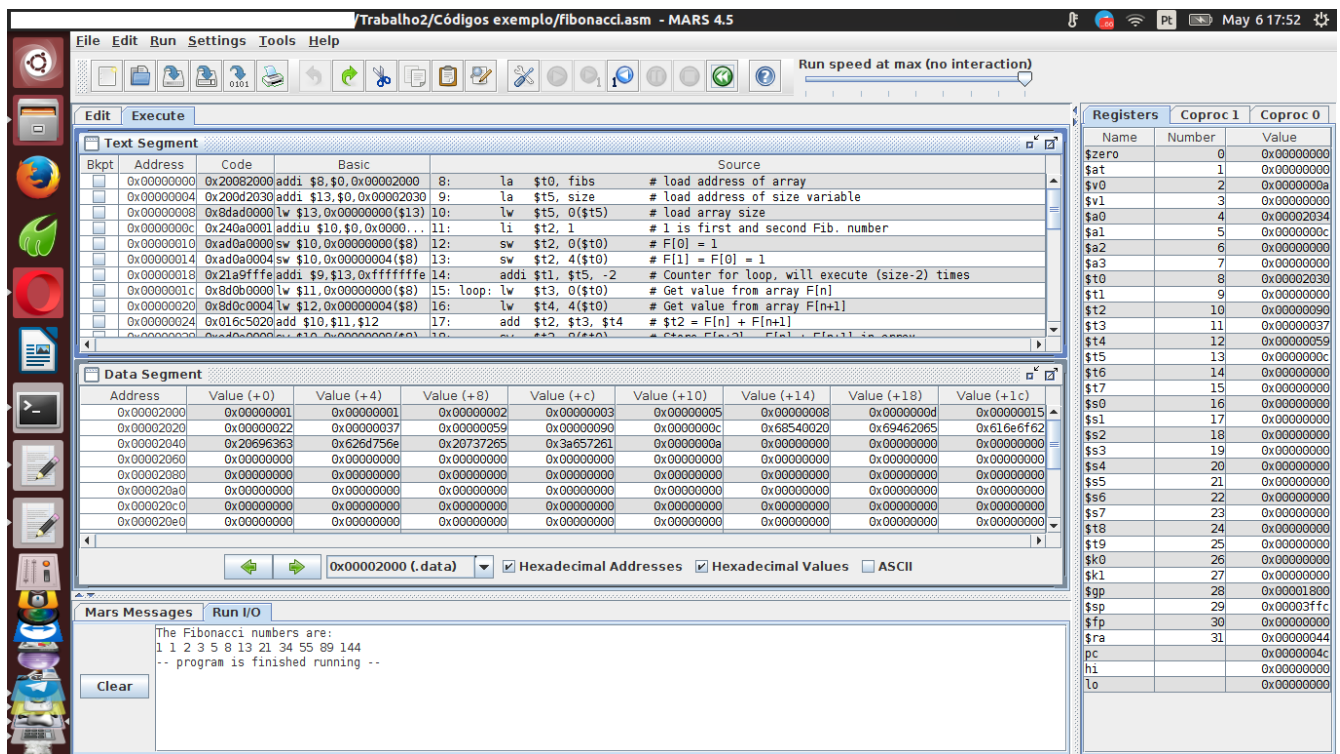


Imagem 9 – Compilação, execução e resultado do simulador MARS.

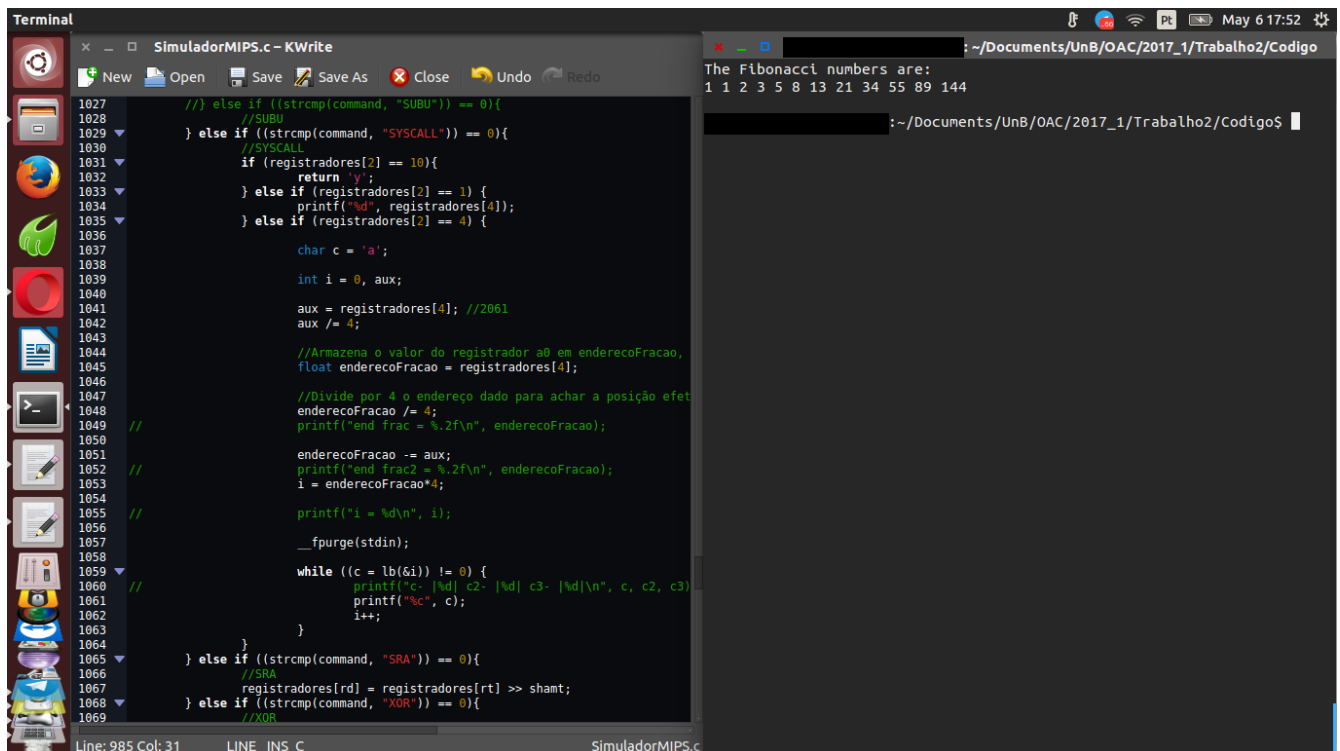


Imagem 10 – Compilação, execução e resultado do simulador implementado.

Referências bibliográficas

- [1] http://fourier.eng.hmc.edu/e85_old/lectures/instruction/node7.html → Instruction Set of MIPS Processor.
- [2] Computer Organization and Design, 4th Ed, D. A. Patterson and J. L. Hennessy.