



UNIVERSIDADE FEDERAL DE SANTA CATARINA

CENTRO TECNOLÓGICO - CTC

ORGANIZAÇÃO DE COMPUTADORES I - INE5411-03208B

PROFESSOR: MARCELO DANIEL BEREJUCK

ALUNOS: GUSTAVO BATISTELL (12101228) E LUCAS DE S. MORO (23201136)

RELATÓRIO - LABORATÓRIO 1

Florianópolis, Outubro de 2024.

Introdução

Este relatório tem como objetivo apresentar e explicar a solução de implementação da atividade proposta no Laboratório 1, onde a aplicação armazena e lê as variáveis **b**, **d** e **e** na memória de dados, realiza os cálculos aritméticos $a = b + 35$, $c = d^3 - (a + e)$ e armazena o resultado final numa variável **c**.

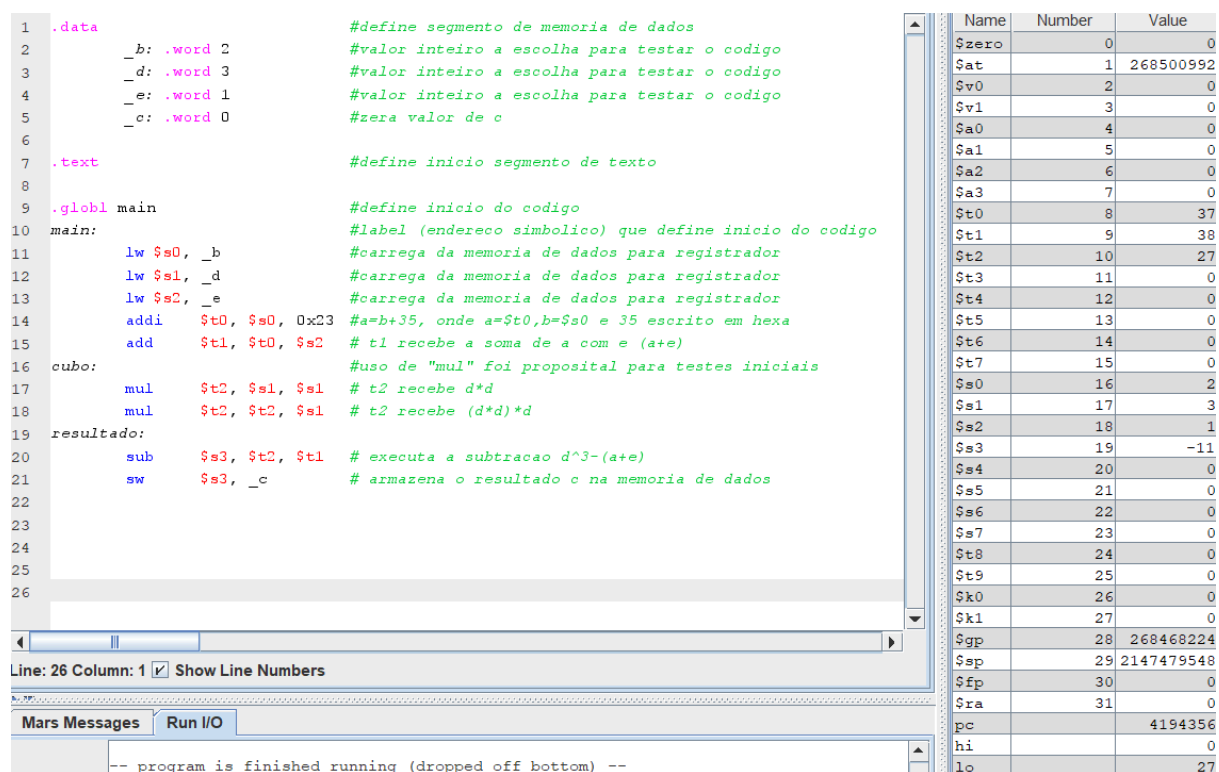
Como limitação imposta, não deveriam ser utilizadas instruções de multiplicação e desvios, o que faz sentido para um processador mais simples (com limitação de hardware) e simularia o trabalho que o compilador para o suposto processador teria em gerar as instruções em assembly.

No primeiro exercício, os valores das variáveis são diretamente inseridos no código, enquanto no segundo exercício, os valores são fornecidos pelo usuário através de entrada via **syscall**.

Exercício 1

A abordagem escolhida pela dupla foi de inicialmente utilizar as instruções não permitidas para testar leitura/escrita em memória e verificar os possíveis resultados de **c** conforme as entradas de **b**, **d** e **e**.

A imagem abaixo mostra o primeiro código de teste com instruções de multiplicação não permitidas funcionando corretamente e o valor de **c** apresentado no registrador \$s3.



```
1 .data                                #define segmento de memoria de dados
2     _b: .word 2                      #valor inteiro a escolha para testar o codigo
3     _d: .word 3                      #valor inteiro a escolha para testar o codigo
4     _e: .word 1                      #valor inteiro a escolha para testar o codigo
5     _c: .word 0                      #zera valor de c
6
7 .text                                #define inicio segmento de texto
8
9 .globl main                          #define inicio do codigo
10 main:                               #label (endereço simbolico) que define inicio do codigo
11     lw $s0, _b                      #carrega da memoria de dados para registrador
12     lw $s1, _d                      #carrega da memoria de dados para registrador
13     lw $s2, _e                      #carrega da memoria de dados para registrador
14     addi $t0, $s0, 0x23             #a=b+35, onde a=$t0,b=$s0 e 35 escrito em hexa
15     add $t1, $t0, $s2               # t1 recebe a soma de a com e (a+e)
16     #uso de "mul" foi proposital para testes iniciais
17     mul $t2, $s1, $s1               # t2 recebe d*d
18     mul $t2, $t2, $s1               # t2 recebe (d*d)*d
19     resultado:
20     sub $s3, $t2, $t1               # executa a subtracao d^3-(a+e)
21     sw $s3, _c                     # armazena o resultado c na memoria de dados
22
23
24
25
26
```

Name	Number	Value
\$zero	0	0
\$at	1	268500992
\$v0	2	0
\$v1	3	0
\$a0	4	0
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	37
\$t1	9	38
\$t2	10	27
\$t3	11	0
\$t4	12	0
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$s0	16	2
\$s1	17	3
\$s2	18	1
\$s3	19	-11
\$s4	20	0
\$s5	21	0
\$s6	22	0
\$s7	23	0
\$t8	24	0
\$t9	25	0
\$k0	26	0
\$k1	27	0
\$gp	28	268468224
\$sp	29	2147479548
\$fp	30	0
\$ra	31	0
pc		4194356
hi		0
lo		27

Line: 26 Column: 1 ☒ Show Line Numbers

Mars Messages Run I/O

-- program is finished running (dropped off bottom) --

Figura 1 - Teste inicial funcional do código, mas com instruções **mul** não permitidas.

Neste primeiro teste foi constatada a limitação para entrada de valores inteiros positivos e negativos onde a combinação resulta em um valor de **c** dentro do intervalo [-2147483648, +2147483647] sem ocorrer overflow. Esse intervalo é $[-2^{31}, 2^{31} - 1]$.

Em um segundo passo, para eliminar as instruções não permitidas de multiplicação necessárias para implementar d^3 , foi analisada a multiplicação de maneira mais analítica, ou seja, se para encontrar d^2 (como um valor intermediário já que $d^2 \cdot d = d^3$) o valor de **d** pode ser somado em **d** vezes, o resultado intermediário desta soma poderia ser reutilizado para uma segunda soma de **d** vezes.

Assim, comentando os trechos de código desnecessários para a multiplicação, foi testado d^2 com sucesso, conforme apresentado na Figura 2 abaixo.

1	.data		#define segmento de memoria de dados
2	#	_b: .word 2	#valor inteiro a escolha para testar o codigo
3		_d: .word 25	#valor inteiro a escolha para testar o codigo
4	#	_e: .word 1	#valor inteiro a escolha para testar o codigo
5		_c: .word 0	#zera valor de c
6			
7	.text		#define inicio segmento de texto
8			
9	.globl main		#define inicio do codigo
10	main:		#label (endereço simbolico) que define inicio do codigo
11	#	lw \$s0, _b	#carrega da memoria de dados para registrador
12		lw \$s1, _d	#carrega da memoria de dados para registrador
13	#	lw \$s2, _e	#carrega da memoria de dados para registrador
14	#	addi \$t0, \$s0, 0x23	#a=b+35, onde a=\$t0,b=\$s0 e 35 escrito em hexa
15	#	add \$t1, \$t0, \$s2	# t1 recebe a soma de a com e (a+e)
16			
17			#iniciando valores.
18		addi \$t2, \$s1, 0	#t2 recebe d pra usar no decrementador
19		addi \$t3, \$0, 0	#zera um temporario pra usar nas somas
20	loop_soma:		
21		add \$t3, \$t3, \$s1	# t3 recebe d+d
22		addi \$t2, \$t2, -1	# decrementa t2
23		bne \$t2, \$0, loop_soma	#compara e se acumulador nao chegou no valor d, desvia
24		sw \$t3, _c	

Name	Value
\$...	0
\$at	1 268500992
\$v0	2 0
\$v1	3 0
\$a0	4 0
\$a1	5 0
\$a2	6 0
\$a3	7 0
\$t0	8 0
\$t1	9 0
\$t2	10 0
\$t3	11 625
\$t4	12 0
\$t5	13 0
\$t6	14 0
\$t7	15 0
\$s0	16 0
\$s1	17 25
\$s2	18 0
\$s3	19 0
\$s4	20 0
\$s5	21 0
\$s6	22 0
\$s7	23 0
\$t8	24 0

Figura 2 - Segundo teste implementando d^2 sem utilizar instruções de multiplicação

O terceiro passo foi a implementação de $d^2 \cdot d$, ou seja, d^3 sem empregar instrução **mul**. Também houve sucesso neste passo, conforme apresentado no exemplo da Figura 3, apesar de limitado a valores de **d** inteiros positivos. Mesmo com o sucesso, ainda foram utilizados desvios **bne**, também não permitidos.

1	.data	#define segmento de memoria de dados			
2	_b: .word 4	#valor inteiro a escolha para testar o codigo			
3	_d: .word 7	#valor inteiro a escolha para testar o codigo			
4	_e: .word 2	#valor inteiro a escolha para testar o codigo			
5	_c: .word 0	#zera valor de c			
6	.text	#define inicio segmento de texto			
7	.globl main	#define inicio do codigo			
8	main:	#label (endereço simbolico) que define inicio do codigo			
9	lw \$s0, _b	#carrega da memoria de dados para registrador			
10	lw \$s1, _d	#carrega da memoria de dados para registrador			
11	lw \$s2, _e	#carrega da memoria de dados para registrador			
12	addi \$t0, \$s0, 0x23	#a=b+35, onde a=\$t0, b=\$s0 e 35 escrito em hexa			
13	add \$t1, \$t0, \$s2	# t1 recebe a soma de a com e (a+e)			
14	#iniciando valores.				
15	addi \$t2, \$s1, 0	#t2 recebe d pra usar no decrementador			
16	addi \$t3, \$0, 0	#zera um temporario pra usar nas somas			
17	loop_square:				
18	add \$t3, \$t3, \$s1	#t3 recebe d+d			
19	addi \$t2, \$t2, -1	#decrementa t2			
20	bne \$t2, \$0, loop_square	#compara e se acumulador nao chegou no valor d, desvia pro loop			
21	#inicio do segundo loop p d^3 com base em d^2				
22	addi \$t2, \$s1, 0	#reinicia t2 com d p decrementador			
23	addi \$t4, \$0, 0	#inicia t4 p realizar as somas			
24	loop_cube:				
25	add \$t4, \$t4, \$t3	#t4 = t4 + d^2			
26	addi \$t2, \$t2, -1	#decrementa t2			
27	bne \$t2, \$0, loop_cube	#enquanto t2 != 0 continua no loop			
28	#entrega de c				
29	sub \$s3, \$t4, \$t1				
30	sw \$s3, _c				

Figura 3 - Teste funcional do código sem instruções **mul** (resultado apresentado em \$s3)

Por fim, na implementação final do exercício, aplicando agora a resolução completa das expressões e armazenando o valor final na memória de dados

Valores de entrada e saída

Como mostrado abaixo na Figura 4, é realizada a operação $a = b + 35$ com $b = 2$, resultando em $a = 37$. Depois é executado $c = d^3 - (a + e)$ com $d = 3$ e $e = 1$, resultando em $c = -11$ e armazenando o resultado na memória.

\$t0	8	37
\$t1	9	38
\$t2	10	0
\$t3	11	9
\$t4	12	27
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$s0	16	2
\$s1	17	3
\$s2	18	1
\$s3	19	-11

Figura 4 - Registradores - Entrada de dados em verde e saída em vermelho.

O código possui 27 linhas de código Assembly, 21 linhas de instruções na coluna Basic, porém 17 linhas de instruções na coluna Source. O motivo dessa diferença de linhas é que a coluna Source contém as instruções fornecidas pelo programador, em um nível de

abstração mais alto e próximo do código fonte, já a coluna Basic representa as instruções reais geradas pelo assembler após a tradução do código original. Isso ocorre pois algumas instruções de alto nível podem ser traduzidas em mais de uma instrução de nível de máquina, incluindo operações adicionais. A Figura 5 abaixo apresenta o Text Segment do simulador Mars.

Address	Code	Basic	Source
0x00400000	0x3c011001	lui \$1,4097	9: lw \$s0, _b #carrega da memoria de dados para registrador
0x00400004	0x8c300000	lw \$16,0(\$1)	
0x00400008	0x3c011001	lui \$1,4097	10: lw \$s1, _d #carrega da memoria de dados para registrador
0x0040000c	0x8c310004	lw \$17,4(\$1)	
0x00400010	0x3c011001	lui \$1,4097	11: lw \$s2, _e #carrega da memoria de dados para registrador
0x00400014	0x8c320008	lw \$18,8(\$1)	
0x00400018	0x22080023	addi \$8,\$16,35	12: addi \$t0, \$s0, 0x23 #a=b+35, onde a=\$t0,b=\$s0 e 35 escrito em hexa
0x0040001c	0x01124820	add \$9,\$8,\$18	13: add \$t1, \$t0, \$s2 # t1 recebe a soma de a com e (a+e)
0x00400020	0x222a0000	addi \$10,\$17,0	15: addi \$t2, \$s1, 0 #t2 recebe d pra usar no decrementador
0x00400024	0x200b0000	addi \$11,\$0,0	16: addi \$t3, \$0, 0 #zera um temporario pra usar nas somas
0x00400028	0x01715820	add \$11,\$11,\$17	18: add \$t3, \$t3, \$s1 #t3 recebe d+d
0x0040002c	0x214affff	addi \$10,\$10,-1	19: addi \$t2, \$t2, -1 #decrementa t2
0x00400030	0x1540ffff	bne \$10,\$0,-3	20: bne \$t2, \$0, loop_square #compara e se acumulador nao chegou no valor d, desvia pro loop de soma
0x00400034	0x222a0000	addi \$10,\$17,0	22: addi \$t2, \$s1, 0 #reinicia t2 com d p decrementador
0x00400038	0x200c0000	addi \$12,\$0,0	23: addi \$t4, \$0, 0 #inicia t4 p realizar as somas
0x0040003c	0x018b6020	add \$12,\$12,\$11	25: add \$t4, \$t4, \$t3 #t4 = t4 + d^2
0x00400040	0x214affff	addi \$10,\$10,-1	26: addi \$t2, \$t2, -1 #decrementa t2
0x00400044	0x1540ffff	bne \$10,\$0,-3	27: bne \$t2, \$0, loop_cube #enquanto t2 != 0 continua no loop
0x00400048	0x01899822	sub \$19,\$12,\$9	29: sub \$s3, \$t4, \$t1
0x0040004c	0x3c011001	lui \$1,4097	30: sw \$s3, _c
0x00400050	0xac33000c	sw \$19,12(\$1)	

Figura 5 - Text Segment do simulador Mars

Exercício 2

No segundo exercício, a proposta foi avançar na implementação anterior, permitindo que os valores das variáveis **b**, **d** e **e** fossem fornecidos pelo usuário durante a execução do programa. Para realizar essa implementação, utilizamos o serviço de syscall disponível no MIPS, que permite a leitura de inteiros diretamente da entrada padrão. O fluxo de leitura é realizado em três etapas, cada uma correspondente a uma variável.

Na Figura 6 abaixo mostra o pedaço do segundo código com as implementações citadas acima

```

# Leitura de b
li $v0, 5          # Código de serviço para leitura de inteiro
syscall            # Chama o sistema para ler o inteiro
add $s0, $v0, $zero # Move o valor lido para $s0 (b)
sw $s0, b          # Armazena o valor de b na memória

# Leitura de d
li $v0, 5          # Código de serviço para leitura de inteiro
syscall            # Chama o sistema para ler o inteiro
add $s1, $v0, $zero # Move o valor lido para $s1 (d)
sw $s1, d          # Armazena o valor de d na memória

# Leitura de e
li $v0, 5          # Código de serviço para leitura de inteiro
syscall            # Chama o sistema para ler o inteiro
add $s2, $v0, $zero # Move o valor lido para $s2 (e)
sw $s2, e          # Armazena o valor de e na memória

```

Figura 6 - Leitura das variáveis **b**, **d** e **e** fornecidas pelo usuário

Para cada variável, são executados os seguintes passos:

Preparação para Leitura: Carrega o código de serviço 5 no registrador \$v0, que indica ao sistema que uma operação de leitura de inteiro será realizada.

Execução da Syscall: A instrução **syscall** é chamada para ler o inteiro da entrada padrão. O valor lido é retornado no registrador \$v0.

Armazenamento do Valor: O valor lido é movido para um registrador específico (\$s0 para b, \$s1 para d, \$s2 para e) usando a instrução **add**, que transfere o valor de \$v0 para o registrador correspondente. Em seguida, o valor é armazenado na memória na posição designada para cada variável.

Valores de entrada e saída

Como mostrado no exemplo da Figura 7 abaixo, é realizada novamente a operação $a = b + 35$ com $b = 2$, resultando em $a = 37$. Depois é executado $c = d^3 - (a + e)$ com $d = 3$ e $e = 1$, resultando em $c = -11$ e armazenando o resultado na memória.

2		
3		
1		
-- program is finished running (dropped off bottom) --		
\$v0	2	1
\$v1	3	0
\$a0	4	0
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	37
\$t1	9	38
\$t2	10	0
\$t3	11	9
\$t4	12	27
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$s0	16	2
\$s1	17	3
\$s2	18	1
\$s3	19	-11

Figura 7 - Registradores e terminal - Entrada de dados em verde e saída em vermelho.

Como mostra a Figura 8 abaixo do Text Segment no simulador Mars, o código possui 36 linhas de código Assembly, 30 linhas de instruções na coluna Basic e 26 linhas de instruções na coluna Source. Novamente isso ocorre pois a coluna Source contém as instruções fornecidas pelo programador, em um nível de abstração mais alto e a coluna Basic as instruções geradas pelo assembler após a tradução do código original.

Address	Code	Basic	Source
0x00400000	0x24020005	addiu \$2,\$0,5	12: li \$v0, 5 #leitura de inteiro
0x00400004	0x0000000c	syscall	13: syscall # Chama o sistema para ler o inteiro
0x00400008	0x00408020	add \$16,\$2,\$0	14: add \$s0, \$v0, \$zero # Move o valor lido para \$s0 (b)
0x0040000c	0x3c011001	lui \$1,4097	15: sw \$s0, b # Armazena o valor de b na memória
0x00400010	0xac300000	sw \$16,0(\$1)	
0x00400014	0x24020005	addiu \$2,\$0,5	18: li \$v0, 5 # Código de serviço para leitura de inteiro
0x00400018	0x0000000c	syscall	19: syscall # Chama o sistema para ler o inteiro
0x0040001c	0x00408820	add \$17,\$2,\$0	20: add \$s1, \$v0, \$zero # Move o valor lido para \$s1 (d)
0x00400020	0x3c011001	lui \$1,4097	21: sw \$s1, d # Armazena o valor de d na memória
0x00400024	0xac310004	sw \$17,4(\$1)	
0x00400028	0x24020005	addiu \$2,\$0,5	24: li \$v0, 5 # Código de serviço para leitura de inteiro
0x0040002c	0x0000000c	syscall	25: syscall # Chama o sistema para ler o inteiro
0x00400030	0x00409020	add \$18,\$2,\$0	26: add \$s2, \$v0, \$zero # Move o valor lido para \$s2 (e)
0x00400034	0x3c011001	lui \$1,4097	27: sw \$s2, e # Armazena o valor de e na memória
0x00400038	0xac320008	sw \$18,8(\$1)	
0x0040003c	0x22080023	addi \$8,\$16,35	29: addi \$t0, \$s0, 0x23 # a = b + 35, onde a = \$t0, b = \$s0 e 35 escrito em hexa
0x00400040	0x01124820	add \$9,\$8,\$18	30: add \$t1, \$t0, \$s2 # \$t1 recebe a soma de a com e (a + e)
0x00400044	0x222a0000	addi \$10,\$17,0	32: addi \$t2, \$s1, 0 # \$t2 recebe d para usar no decrementador
0x00400048	0x200b0000	addi \$11,\$0,0	33: addi \$t3, \$zero, 0 # Zera um temporário (\$t3) para usar nas somas (d^2)
0x0040004c	0x01715820	addi \$11,\$11,\$17	36: add \$t3, \$t3, \$s1 # \$t3 recebe d + d (soma de d)
0x00400050	0x214affff	addi \$10,\$10,-1	37: addi \$t2, \$t2, -1 # Decrementa \$t2
0x00400054	0x1540ffff	bne \$10,\$0,-3	38: bne \$t2, \$zero, loop_square # Compara e se acumulador não chegou no valor de d, desvia pro loop de soma
0x00400058	0x222a0000	addi \$10,\$17,0	41: addi \$t2, \$s1, 0 # Reinicia \$t2 com d para o decrementador
0x0040005c	0x200c0000	addi \$12,\$0,0	42: addi \$t4, \$zero, 0 # Inicia \$t4 para realizar as somas (d^3)
0x00400060	0x018b6020	add \$12,\$12,\$11	45: add \$t4, \$t4, \$t3 # \$t4 = \$t4 + d^2 (acumula d^2)
0x00400064	0x214affff	addi \$10,\$10,-1	46: addi \$t2, \$t2, -1 # Decrementa \$t2
0x00400068	0x1540ffff	bne \$10,\$0,-3	47: bne \$t2, \$zero, loop_cube # Enquanto \$t2 != 0, continua no loop
0x0040006c	0x01899822	sub \$19,\$12,\$9	49: sub \$s3, \$t4, \$t1 # c = d^3 - (a + e)
0x00400070	0x3c011001	lui \$1,4097	50: sw \$s3, c # Armazena o resultado final (c) na memória.
0x00400074	0xac33000c	sw \$19,12(\$1)	

Figura 8 - Text Segment do simulador Mars

Conclusão

As limitações de um processador podem inferir trabalho extra para um projetista e para o compilador. Nota-se que o uso de instruções mais poderosas facilitam o trabalho do programador e foram necessários diversos passos para chegar mais próximo às “exigências de projeto”.

Como as exigências não foram alcançadas por completo, um quarto passo seria necessário além dos 3 descritos na execução do Exercício 1. Esse quarto passo foi tentado pela dupla por diversas vezes sem sucesso, com a intenção de eliminar os desvios, visto a restrição dada. O caminho para tentar solucionar envolveu o uso de instruções xor, andi, sll, srl e or, mas com estas, houve sucesso apenas em sinalizar o fim da contagem do decrementador. Essa abordagem utilizada pela dupla acabava trazendo a necessidade do uso de instruções como **j** e **jr** que também eram restritas. Em última análise, a dupla cogitou continuar a utilizar as instruções de lógica e deslocamento para sinalizar o fim da contagem do decrementador e utilizar **sw** e **lw** para endereçamento, e assim emular o efeito dos desvios desejados, o que não foi efetivamente testado até o momento da entrega deste relatório.