

1º Trabalho- Organização de Computadores

DESMONTADOR/DESASSEMBLADOR

Lorenzo Sacchet Taschetto e Alan Henrique Jacobs

SUMÁRIO

INTRODUÇÃO-----	3
OBJETIVOS-----	4
REVISÃO BIBLIOGRÁFICA-----	5
METODOLOGIA-----	6-10
EXPERIMENTO-----	11-12
RESULTADOS-----	13-14
DISCUSSÃO-----	15
CONCLUSÕES E PERSPECTIVAS-----	16

INTRODUÇÃO

Esse relatório descreve o desenvolvimento do programa desassemblador/desmontador para o processador MIPS, conforme proposto pelo primeiro trabalho da disciplina Organização de Computadores. A implementação do programa foi realizada em assembly para o processador MIPS e compilação utilizou o programa MARS. O relatório contém os tópicos (além da introdução): Objetivos, Revisão bibliográfica, Metodologia, Experimento, Resultados, Discussão e Conclusões e Perspectivas.

Na seção ‘objetivos’ é relatado o que se esperava do trabalho, bem como o funcionamento desejado.

Na seção ‘revisão bibliográfica’ é exposto as fontes, materiais e pesquisas usados para realização do trabalho.

Na seção ‘metodologia’ é descrito o funcionamento de todo código realizado para alcançar o objetivo final.

Na seção ‘experimento’ é relatado o cenário de testes usados, os critérios de avaliação e as métricas que verificam o funcionamento do código.

Na seção ‘resultados’ é exposto o que foi obtido com o funcionamento do programa mediante os testes.

Na seção ‘discussão’ é descrito a análise dos resultados, além de citar desafios na implementação e possíveis complementações ao código.

Na seção ‘conclusões e perspectivas’ é apresentado o que se concluiu com o trabalho, se o programa cumpriu a proposta e perspectivas futuras.

OBJETIVOS

O trabalho tem como objetivo a criação de um desmontador/desassemblador em assembly MIPS que receberá um arquivo .bin como “input”, o qual contém instruções em linguagem de máquina, e realizará operações que transformam o código de máquina em instruções do assembly MIPS. As instruções serão gravadas em um arquivo “output”, precedidas do endereço em que estão e do código de máquina correspondente.

Exemplo genérico de como deve ser o arquivo “output”:

Endereço	Instrução em linguagem de máquina	Instrução em assembly MIPS
.....

Se repete para cada linha do arquivo “input”

Exemplo para arquivo “input” contendo:

F8 FF BD 27 --> 27BDFFF8

Resultado esperado no arquivo “output”:

0x00040000 0x27BDFFF8 addiu \$sp, \$sp, 0x0000FFF8

REVISÃO BIBLIOGRÁFICA

Para a realização do trabalho, foram utilizados conceitos e conhecimentos adquiridos no capítulo 2 (Instruções: A linguagem de Máquina) do livro "Organização e Projeto de Computadores 4º edição" de Patterson e Hennessy. Tabelas e trechos do livro fundamentaram o embasamento do trabalho, tais como:

Endereçamento absoluto – Tipo J

j longe

Supondo que esta instrução está no endereço de memória 0xE0AA0000 e que longe é uma etiqueta que corresponde a 0xE0000008 então teremos:

op	target
000010	0000 0000 0000 0000 0000 10

PC = 1110 0000 1010 1010 0000 0000 0000 10

Após substituir os 26 bits apropriados do PC teremos

PC = 1110 0000 0000 0000 0000 0000 0000 1000

AC1 – Formatos de Instruções 13

Formato Tipo I

opcode	rs	rt	endereço
6 bits	5 bits	5 bits	16 bits
código da operação	registrador	registrador	endereço de memória

Formato Tipo R

opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
código da operação	registrador fonte	registrador fonte	registrador destino	deslocamento	sub código da operação

Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	$s1 = s2 + s3$	Três operandos; dados nos registradores
	subtract	sub \$s1,\$s2,\$s3	$s1 = s2 - s3$	Três operandos; dados nos registradores
	add immediate	addi \$s1,\$s2,20	$s1 = s2 + 20$	Usada para somar constantes
Transferência de dados	load word	lw \$s1,20(\$s2)	$s1 = \text{Memória}[s2 + 20]$	Dados da memória para o registrador
	store word	sw \$s1,20(\$s2)	$\text{Memória}[s2 + 20] = s1$	Dados do registrador para a memória
	load half	lh \$s1,20(\$s2)	$s1 = \text{Memória}[s2 + 20]$	Halfword da memória para registrador
	load half unsigned	lhu \$s1,20(\$s2)	$s1 = \text{Memória}[s2 + 20]$	Halfword da memória para registrador
	store half	sh \$s1,20(\$s2)	$\text{Memória}[s2 + 20] = s1$	Halfword de um registrador para memória
	load byte	lb \$s1,20(\$s2)	$s1 = \text{Memória}[s2 + 20]$	Byte da memória para registrador
	load byte unsigned	lbu \$s1,20(\$s2)	$s1 = \text{Memória}[s2 + 20]$	Byte da memória para registrador
	store byte	sb \$s1,20(\$s2)	$\text{Memória}[s2 + 20] = s1$	Byte de um registrador para memória
	load linked word	ll \$s1,20(\$s2)	$s1 = \text{Memória}[s2 + 20]$	Carrega word como 1ª metade do swap atômico
	store condition, word	sc \$s1,20(\$s2)	$\text{Memória}[s2 + 20] = s1; s1 = 0 \text{ or } 1$	Armazena word como 2ª metade do swap atômico
Lógica	load upper immed.	lui \$s1,20	$s1 = 20 * 2^{16}$	Carrega constante nos 16 bits mais altos
	and	and \$s1,\$s2,\$s3	$s1 = s2 \& s3$	Três operadores em registrador; AND bit a bit
	or	or \$s1,\$s2,\$s3	$s1 = s2 s3$	Três operadores em registrador; OR bit a bit
	nor	nor \$s1,\$s2,\$s3	$s1 = \sim (s2 s3)$	Três operadores em registrador; NOR bit a bit
	and immediate	andi \$s1,\$s2,20	$s1 = s2 \& 20$	AND bit a bit registrador com constante
	or immediate	ori \$s1,\$s2,20	$s1 = s2 20$	OR bit a bit registrador com constante
	shift left logical	sll \$s1,\$s2,10	$s1 = s2 \ll 10$	Deslocamento à esquerda por constante
	shift right logical	srl \$s1,\$s2,10	$s1 = s2 \gg 10$	Deslocamento à direita por constante
Desvio condicional	branch on equal	beq \$s1,\$s2,25	if ($s1 == s2$) go to PC + 4 + 100	Testa igualdade; desvio relativo ao PC
	branch on not equal	bne \$s1,\$s2,25	if ($s1 \neq s2$) go to PC + 4 + 100	Testa desigualdade; relativo ao PC
	set on less than	slt \$s1,\$s2,\$s3	if ($s2 < s3$) $s1 = 1$; else $s1 = 0$	Compara menor que; usado com beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if ($s2 < s3$) $s1 = 1$; else $s1 = 0$	Compara menor que sem sinal
	set less than immediate	slti \$s1,\$s2,20	if ($s2 < 20$) $s1 = 1$; else $s1 = 0$	Compara menor que constante
	set less than immediate unsigned	sltiu \$s1,\$s2,20	if ($s2 < 20$) $s1 = 1$; else $s1 = 0$	Compara menor que constante sem sinal
	jump	j 2500	go to 10000	Desvia para endereço de destino
Desvio incondicional	jump register	jr \$ra	go to \$ra	Para switch e retorno de procedimento
	jump and link	jal 2500	$sra = PC + 4$; go to 10000	Para chamada de procedimento

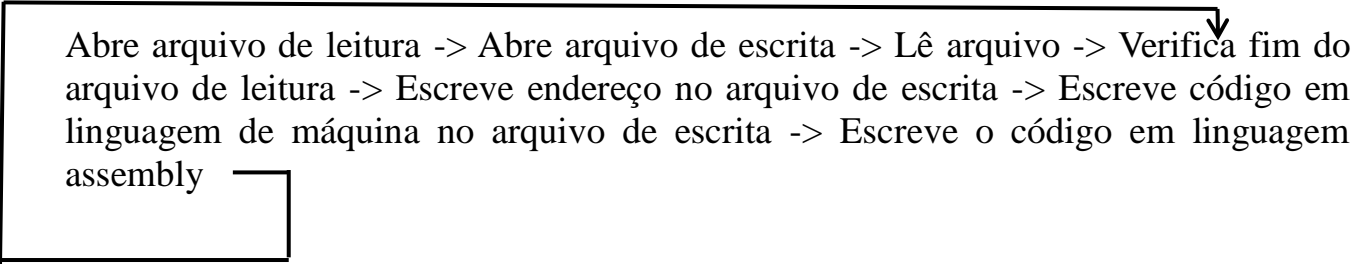
Página 683 a 700 também ofereceram informações acerca do formato das instruções.

METODOLOGIA

O programa possui comentários ao lado de cada linha que descrevem o que está sendo feito e possui comentários gerais de orientação dentro do código, optou-se por deixar o código o mais explicado possível, bem como organizado e legível. Além das instruções presentes no arquivo fornecido pelo professor, o programa também possui suporte para outras instruções consideradas mais usada na linguagem assembly MIPS.

Aqui está a explicação textual do código:

Fluxo do programa:



```
graph LR; A[Abre arquivo de leitura] --> B[Abre arquivo de escrita]; B --> C[Lê arquivo]; C --> D[Verifica fim do arquivo de leitura]; D --> E[Escreve endereço no arquivo de escrita]; E --> F[Escreve código em linguagem de máquina no arquivo de escrita]; F --> G[Escreve o código em linguagem assembly];
```

Abre arquivo de leitura -> Abre arquivo de escrita -> Lê arquivo -> Verifica fim do arquivo de leitura -> Escreve endereço no arquivo de escrita -> Escreve código em linguagem de máquina no arquivo de escrita -> Escreve o código em linguagem assembly

A seção .data contém todas declarações de strings, caracteres que serão impressos/escritos no arquivo de output ao longo do programa, bem como a declaração do buffer que guarda a string de dígitos hexadecimais.

O label "abre_arquivo_leitura" em Assembly MIPS realiza a abertura de um arquivo para leitura por meio de chamadas de sistema específicas. O registrador \$v0 é utilizado para armazenar o código da chamada do sistema relacionado à abertura de arquivo. Além disso, são definidos os parâmetros necessários para a abertura do arquivo, como o nome do arquivo de saída e o modo de leitura (0). O valor de retorno da chamada syscall do sistema representa o identificador do arquivo aberto e é armazenado em um registrador específico, permitindo o uso posterior desse identificador. Para garantir um tratamento adequado de erros, a função inclui um mecanismo de verificação. O valor de retorno é comparado com zero para determinar se ocorreu algum erro durante a abertura do arquivo. Se o valor for menor que zero, indica que houve um erro e o fluxo do código é direcionado para o label que lida com o erro ("label_erro_input").

O label "abre_arquivo_escrita" funciona analogamente ao "abre_arquivo_leitura", mudando o modo, o qual nesse caso é de escrita (1), e o label responsável por lidar com possíveis erros "label_erro_output".

O label "le_arquivo" inicia carregando o descritor do arquivo e define o endereço de memória onde os 4 bytes lidos serão armazenados (tamanho máximo de leitura = 4 bytes). Em seguida, é realizada a chamada do sistema para ler o arquivo, utilizando a

instrução "syscall" e o código da chamada do sistema adequado. Após a execução da chamada do sistema, a leitura é realizada e os quatro bytes do arquivo são armazenados na área de memória especificada anteriormente.

O label “verifica_fim_arquivo” começa com a instrução "slti \$t0, \$v0, 4", a qual compara o valor de retorno da chamada do sistema (\$v0) com o valor 4. Se o valor de retorno for menor que 4, o registrador \$t0 é definido como 1. Caso contrário, \$t0 é definido como 0. Em seguida, a instrução "bne \$t0, \$zero, fim_programa" é utilizada para desviar o fluxo do código para o label 'fim_programa' se o valor de \$t0 for diferente de zero. Isso significa que o valor de retorno da chamada do sistema indicou o fim do arquivo.

O label “endereço” começa carregando um valor localizado em um endereço de memória específico para o registrador \$a0, em seguida, o endereço da variável 'buffer' é carregado para o registrador \$a1. Após isso, é feita uma chamada de função utilizando a instrução "jal" para executar a função "hex_string", a qual tem como objetivo converter o valor hexadecimal presente no registrador \$a0 em uma string legível. Em seguida, o código realiza a impressão do conteúdo convertido em hexadecimal. A partir disso, é realizada a impressão de um espaço. Por fim, o código realiza o incremento do valor de um endereço. O valor localizado em 8(\$sp) é carregado em \$t0. Em seguida, o valor de \$t0 é incrementado em 4 e armazenado novamente em 8(\$sp).

O label “codigo_maquina” inicia carregando o valor localizado na pilha, especificamente em 4(\$sp), para o registrador \$a0. Esse valor representa o endereço do código de máquina hexadecimal que será convertido para uma string. Em seguida, o endereço da string de destino, denominada 'buffer', é carregado para o registrador \$a1. É realizada uma chamada de função utilizando a instrução "jal" para executar a função "hex_string". Essa função tem como objetivo converter o código de máquina hexadecimal presente no registrador \$a0 para uma string legível, armazenando o resultado na variável 'buffer'. Após a conversão, o código imprime o conteúdo convertido em hexadecimal e um espaço.

O label “codigo_assembly” começa carregando o valor da linha localizado na pilha para o registrador \$t4, em seguida começa a extração dos campos para posterior identificação.

Extração dos campos:

# Extrair campos da instrução		
# PARA O OPCODE FORMATO R	# PARA O OPCODE FORMATO I	# PARA O OPCODE FORMATO J
# \$s0 = opcode 6 bits	# \$s0 = opcode 6 bits	# \$s0 = opcode 6 bits
# \$s1 = rs 5 bits	# \$s1 = rs 5 bits-----	
# \$s2 = rt 5 bits	# \$s2 = rt 5 bits	
# \$s3 = rd 5 bits-----		-> # \$s1 or \$s2 or \$s3 or \$s4 or \$s5 26 bits
# \$s4 = shamt 5 bits	-> # \$s3 or \$s4 or \$s5 16 bits	
# \$s5 = funct 6 bits-----		

Por meio de deslocamentos aritméticos para a direita e operações AND com máscaras, conseguimos isolar todos os campos para o formato R, se após a identificação da instrução for percebido que se trata de uma instrução tipo I ou J, é realizado operações de deslocamento aritmético para a esquerda e operações OR para estruturar corretamente a separação de campos de acordo com o tipo da instrução.

Identificação do tipo da instrução:

São realizadas comparações, usando ‘beq’ entre o opcode, separado anteriormente (\$s0), da instrução analisada com opcodes característicos de cada tipo de instrução a fim de direcionar o fluxo do código para a seção correspondente.

Opcodes 0x02 e 0x03 direcionam para o label “opcode_J”

Opcodes 0x00 direcionam para o label “opcode_R”

Em caso de não ter acontecido saltos para esses loops, o programa desvia incondicionalmente para o label “opcode_I”, por conclusão a partir da eliminação das possibilidades tipo J e R.

OBS: Se o opcode for 0x1c o programa desvia diretamente para o label responsável por lidar com a instrução ‘mul’ (“local_mul”), uma vez que essa instrução tem a estrutura do tipo R, mas não segue o padrão de opcode 0x00.

Identificação da instrução:

Após o tipo da instrução ter sido definido, o fluxo do programa se encontra no label do opcode_ J || R || I, nesses label são feitas as concatenações, se necessários, dos campos, como já foi citado acima (parte de “Extrações dos campos”). Em seguida, é feito comparações para direcionar o fluxo do programa para o label específico da instrução.

Em “opcode_J” comparações são feitas entre o opcode da instrução que está sendo analisada com o opcode de instruções do tipo J.

Em “opcode_I” comparações são feitas entre o opcode da instrução que está sendo analisada com o opcode de instruções do tipo I.

Em “opcode_R” comparações são feitas entre o campo funct da instrução que está sendo analisada com o campo funct de instruções do tipo R.

Finalmente, com isso feito, o fluxo do programa se encontrará no label específico da instrução (local_nome_da_instrução).

Em caso da instrução não ser identificada, o programa imprime um mensagem de “Instrução não identificada”.

Label local instrução:

Dentro de cada label específico da instrução já identificada (local_nome_da_instrução), é feito a impressão/escrita dos registradores, espaços, vírgulas, parênteses (se for o caso) e até campos de endereço (IMM) (instruções J e I).

Identificação do registrador:

Cada vez que o programa quer imprimir um registrador, antes ele precisa identificá-lo, isso é feito no label “registrador”, o registrador \$s1 passa a conter o número do registrador que está sendo analisado e comparações são feitas com base nesse número e o valor de cada registrador. Em caso de compatibilidade, o fluxo do programa é desviado para label do tipo reg_número_do_registrador, e são esses labels que imprimem/escrevem o registrador no arquivo output.

Em caso de o registrador não ser identificado o programa imprime uma mensagem de “Registrador não identificado”.

Impressões/Escritas:

As ‘impressões/escritas no arquivo output’ realizadas ao longo programa obedecem a uma estrutura padronizada. Primeiro carrega-se o endereço da variável (espaço, string, vírgula, mensagens de erro, registradores...) a ser impresso/escrito, após isso se define o tamanho da variável que será escrita, finalmente é feito a chamada ao sistema (15) para ele imprimir/escrever no arquivo output.

Funcionamento hex_string:

Inicialmente, a função executa a exibição do prefixo "0x" para indicar que o valor está em formato hexadecimal. Isso é realizado carregando os códigos ASCII '0' e 'x' nos registradores \$t0 e armazenando-os nos endereços \$a1 e \$a1+1, respectivamente.

O endereço \$a1 é incrementado em 2 para avançar para o próximo caractere. Em seguida, o registrador \$t1 é carregado com o valor 0xF000, que será utilizado como máscara para extrair os dígitos hexadecimais do número, o registrador \$t2 é usado como contador com o valor inicial de 32, que será decrementado em 4 a cada iteração. O registrador \$t3 recebe o endereço da tabela hexadecimal. A função entra em um loop chamado "loop_hex_string" onde realiza as etapas de conversão para cada dígito hexadecimal. Primeiramente, é feito um deslocamento lógico para a

direita do número original (\$a0) pelo valor do contador (\$t2) e o resultado é armazenado no registrador \$t4. Em seguida, é realizada uma operação lógica AND entre \$t4 e 0x0F para manter apenas os 4 bits inferiores, correspondentes ao dígito hexadecimal. O registrador \$t4 recebe o endereço da tabela hexadecimal adicionado ao valor do dígito convertido. Posteriormente, é carregado um byte no endereço \$t4 e o resultado é armazenado novamente em \$t4. Esse byte representa o caractere correspondente ao dígito hexadecimal.

O valor de \$t4 é armazenado no endereço \$a1, incrementando \$a1 em 1 para avançar para o próximo caractere da string. O contador \$t2 é decrementado em 4 (cada dígito hexadecimal ocupa 4 bits). O registrador \$t1 é deslocado logicamente para a direita por 4 bits para selecionar o próximo dígito hexadecimal do número original. O loop continua enquanto \$t1 for diferente de zero, ou seja, enquanto houver dígitos hexadecimais a serem convertidos.

Quando o loop termina, é carregado um byte de memória no endereço \$a1 para adicionar o caractere nulo de terminação da string. Por fim, a função retorna pulando para o endereço armazenado em \$ra (retorno da função).

Fim do programa:

O programa carrega o valor 17 em \$v0, chamada ao sistema para encerrar o programa.

EXPERIMENTO

O programa foi testado a partir do arquivo .bin fornecido pelo professor, do qual, inicialmente, analisamos as instruções em linguagem de máquina e manualmente as convertimos para instruções em assembly MIPS a fim de ter uma referência para comparar com os resultados obtidos na execução do programa. Uma vez feito isso, pode-se comparar cada instrução gerada pelo programa com as feitas manualmente (referência). Além disso, foi feita depurações no código para garantir que todos os campos estavam sendo acessados corretamente e com os valores certos nos registradores.

Referência feita manualmente do arquivo .bin do professor:

F8 FF BD 27	27BDFFF8	addiu \$sp,\$sp,0x0000FFF8
05 00 08 24	24080005	addiu \$t0,\$zero,0x00000005
04 00 A8 AF	AFA80004	sw \$t0,0x00000004(\$sp)
20 20 08 00	00082020	add \$a0,\$zero,\$t0
0F 00 10 0C	0C10000F	jal 0x0010000F
04 00 A8 8F	8FA80004	lw \$t0,0x00000004(\$sp)
21 48 02 00	00024821	addu \$t1,\$zero,\$v0
00 00 A2 AF	AFA20000	sw \$v0,0x00000000(\$sp)
21 20 08 00	00082021	addu \$a0,\$zero,\$t0
21 28 09 00	00092821	addu \$a1,\$zero,\$t1
1C 00 10 0C	0C10001C	jal 0x0010001C
08 00 BD 27	27BD0008	addiu \$sp,\$sp,0x00000000
11 00 02 24	24020011	addiu \$v0,\$zero,0x00000011
00 00 04 24	24040000	addiu \$a0,\$zero,0x00000000
0C 00 00 00	0000000C	syscall
F8 FF BD 27	27BDFFF8	addiu \$sp,\$sp,0x0000FFF8
04 00 BF AF	AFBF0004	sw \$ra,0x00000004(\$sp)
00 00 A4 AF	AFA40000	sw \$a0,0x00000000(\$sp)
02 00 04 14	14040002	bne \$a0,\$zero,0x00000002
01 00 02 20	20020001	addi \$v0,\$zero,0x00000001
1A 00 10 08	0810001A	j 0x0010001A
FF FF 84 20	2084FFFF	addi \$a0,\$a0,0x0000FFFF
0F 00 10 0C	0C10000F	jal 0x0010000F
00 00 A4 8F	8FA40000	lw \$a0,0x00000000(\$sp)
02 10 82 70	70821002	mul \$v0,\$a0 \$v0
04 00 BF 8F	8FBF0004	lw \$ra,0x00000004(\$sp)
08 00 BD 23	23BD0008	addi \$sp,\$sp,0x00000008
08 00 E0 03	03E00008	jr \$ra

FC FF BD 27	27BDFFFC	addiu \$sp,\$sp,0x0000FFFC
00 00 A4 AF	AFA40000	sw \$a0,0x00000000(\$sp)
04 00 02 24	24020004	addiu \$v0,\$zero,0x00000004
01 10 01 3C	3C011001	lui \$at,0x00001001
00 00 24 34	34240000	ori \$a0,\$at,0x00000000
0C 00 00 00	0000000C	syscall
00 00 A4 8F	8FA40000	lw \$a0,0x00000000(\$sp)
01 00 02 24	24020001	addiu \$v0,\$zero,0x00000001
0C 00 00 00	0000000C	syscall
01 10 01 3C	3C011001	lui \$at,0x00001001
0F 00 24 34	3424000F	ori \$a0,\$at,0x0000000F
04 00 02 24	24020004	addiu \$v0,\$zero,0x00000004
0C 00 00 00	0000000C	syscall
21 20 05 00	00052021	addu \$a0,\$zero,\$a1
01 00 02 24	24020001	addiu \$v0,\$zero,0x00000001
0C 00 00 00	0000000C	syscall
0A 00 04 24	2404000A	addiu \$a0,\$zero,0x0000000A
0B 00 02 24	2402000B	addiu \$v0,\$zero,0x0000000B
0C 00 00 00	0000000C	syscall
04 00 BD 27	27BD0004	addiu \$sp,\$sp,0x00000004
08 00 E0 03	03E00008	jr \$ra

Orientação para o professor testar:

O zip enviado tem o arquivo “INPUT.bin”, esse possui os códigos em linguagem de máquina enviados pelo professor. Também contém o “CÓDIGO.asm”, o arquivo de saída, “OUTPUT.txt”, e o executável MARS.

O programa está configurado para ler do INPUT.bin e escrever no OUTPUT.txt.

RESULTADOS

Os resultados obtidos com a execução do programa foram conferidos com a referência e mostrou que o desassemblador foi capaz de ler corretamente as instruções em linguagem de máquina, realizar a desmontagem e gerar o resultado esperado. A saída contém o endereço da instrução, a instrução em linguagem de máquina e a instrução em linguagem assembly MIPS correspondente.

Resultado:

0x00040000	0x27BDFFF8	addiu \$sp, \$sp, 0x0000FFF8
0x00040004	0x24080005	addiu \$t0, \$zero, 0x00000005
0x00040008	0xAFA80004	sw \$t0, 0x00000004(\$sp)
0x0004000C	0x00082020	add \$a0, \$zero, \$t0
0x00040010	0x0C10000F	jal 0x0010000F
0x00040014	0x8FA80004	lw \$t0, 0x00000004(\$sp)
0x00040018	0x00024821	addu \$t1, \$zero, \$v0
0x0004001C	0xAFA20000	sw \$v0, 0x00000000(\$sp)
0x00040020	0x00082021	addu \$a0, \$zero, \$t0
0x00040024	0x00092821	addu \$a1, \$zero, \$t1
0x00040028	0x0C10001C	jal 0x0010001C
0x0004002C	0x27BD0008	addiu \$sp, \$sp, 0x00000008
0x00040030	0x24020011	addiu \$v0, \$zero, 0x00000011
0x00040034	0x24040000	addiu \$a0, \$zero, 0x00000000
0x00040038	0x0000000C	syscall
0x0004003C	0x27BDFFF8	addiu \$sp, \$sp, 0x0000FFF8
0x00040040	0xAFBF0004	sw \$ra, 0x00000004(\$sp)
0x00040044	0xAFA40000	sw \$a0, 0x00000000(\$sp)
0x00040048	0x14040002	bne \$zero, \$a0, 0x00000002
0x0004004C	0x20020001	addi \$v0, \$zero, 0x00000001
0x00040050	0x0810001A	j 0x0010001A
0x00040054	0x2084FFFF	addi \$a0, \$a0, 0x0000FFFF
0x00040058	0x0C10000F	jal 0x0010000F
0x0004005C	0x8FA40000	lw \$a0, 0x00000000(\$sp)
0x00040060	0x70821002	mul \$v0, \$a0, \$v0
0x00040064	0x8FBF0004	lw \$ra, 0x00000004(\$sp)
0x00040068	0x23BD0008	addi \$sp, \$sp, 0x00000008
0x0004006C	0x03E00008	jr \$ra
0x00040070	0x27BDFFFC	addiu \$sp, \$sp, 0x0000FFFC

0x00040074	0xAFA40000	sw \$a0, 0x00000000(\$sp)
0x00040078	0x24020004	addiu \$v0, \$zero, 0x00000004
0x0004007C	0x3C011001	lui \$at, 0x00001001
0x00040080	0x34240000	ori \$a0, \$at, 0x00000000
0x00040084	0x0000000C	syscall
0x00040088	0x8FA40000	lw \$a0, 0x00000000(\$sp)
0x0004008C	0x24020001	addiu \$v0, \$zero, 0x00000001
0x00040090	0x0000000C	syscall
0x00040094	0x3C011001	lui \$at, 0x00001001
0x00040098	0x3424000F	ori \$a0, \$at, 0x0000000F
0x0004009C	0x24020004	addiu \$v0, \$zero, 0x00000004
0x000400A0	0x0000000C	syscall
0x000400A4	0x00052021	addu \$a0, \$zero, \$a1
0x000400A8	0x24020001	addiu \$v0, \$zero, 0x00000001
0x000400AC	0x0000000C	syscall
0x000400B0	0x2404000A	addiu \$a0, \$zero, 0x0000000A
0x000400B4	0x2402000B	addiu \$v0, \$zero, 0x0000000B
0x000400B8	0x0000000C	syscall
0x000400BC	0x27BD0004	addiu \$sp, \$sp, 0x00000004
0x000400C0	0x03E00008	jr \$ra

DISCUSSÃO

O programa apresentou os resultados corretos, o código ficou organizado, inteiramente comentado e com otimizações de funções. As maiores dificuldades foram ao lidar com a leitura do arquivo .bin, exigiu maiores atenções e bastantes tentativas, ademais, algumas instruções com comportamento diferente (como a ‘mul’) dos padrões necessitaram de uma análise maior. O código poderia ter uma redução de linhas se não fosse optado pela separação de cada funcionalidade (com espaços e linhas de ‘#’) e explicações detalhadas, porém foi priorizado a legibilidade e o entendimento.

CONCLUSÕES E PERSPECTIVAS

O programa cumpriu bem o que foi proposto, leu o arquivo input .bin de forma certa, desmontou as instruções em linguagem de máquina e as escreveu corretamente em assembly MIPS no arquivo output. O trabalho exigiu bastantes habilidades e tempo, permitindo que assim adquiríssemos conhecimento abrangente em linguagem assembly e linguagem de máquina. O programa pode ser utilizado como ferramenta de estudo, abrindo portas para uma compreensão melhor de códigos em baixo nível e da arquitetura MIPS, impactando positivamente no meio acadêmico.