Trabalho 1 de OAC

Sobre

Simulador de MIPS

O trabalho consiste de montar um emulador de MIPS em C para acesso a memória. Esse trabalho tem o objetivo de auxiliar o entendimento do acesso a memória na arquitetura MIPS.

Participante

Aluno	Matrícula	Username
Victor André Gris Costa	16/0019311	@victoragcosta

Implementação

Descrição do Problema

Neste trabalho queremos montar um emulador de MIPS na questão de acesso a memória. Para tanto precisamos implementar funções para os *loads* e *saves* presentes na arquitetura. Nessa temos como carregar da memória para registradores *words* de 32 bits, *half words* de 16 bits e *bytes* de 8 bits, com sinal para *words* e com ou sem sinal para *half words* e *bytes*. Também podemos salvar de registradores para a memória *words*, *half words* e *bytes*. Sendo assim implementamos funções que salvam dados num vetor que representa a memória e funções que retornam o valor de um certo espaço na memória em forma de *words*. O vetor mem, que representa nossa memória, é formado de *words* e, no nosso caso, usamos uma memória de 4096 *words* ou 16364 *bytes*. Nesse pacote de funções incluímos também uma função *dump* que imprime múltiplas *words* da memória em sequência com o intuito de auxiliar no teste das funções. Vale a pena observar que esse simulador trabalha com uma arquitetura *little endian*, ou seja, o bit menos significativo fica no endereço menor.

Funções implementadas

- **MIPS:** sb $t1, 0($t2)
  C: void sb(uint32_t address, int16_t kte, int8_t dado);
  Esta instrução captura os 8 bits (*byte*) menos significativos armazenados em $t1 e o guarda no endereço de memória armazenado em $t2. Para simular isso criamos essa função em C em que address toma o lugar de $t2, kte toma o lugar do 0 (acesso indexado) e dado toma o lugar de $t1. Para essa função decidi escrever byte a byte fazendo um cast do vetor de *words* para um vetor de *bytes*. Essa função exige que o endereço de memória escrito exista de verdade.
- **MIPS:** sh $t1, 0($t2)
  C: void sh(uint32_t address, int16_t kte, int16_t dado);
  Esta instrução captura os 16 bits (*half word*) menos significativos armazenados em $t1 e o guarda no endereço de memória armazenado em $t2. Para simular isso criamos essa função em C em que address toma o lugar de $t2, kte toma o lugar do 0 (acesso indexado) e dado toma o lugar de $t1. Para essa função decidi escrever byte a byte fazendo um cast do vetor de *words* para um vetor de *bytes*. Essa função exige que o endereço de memória escrito exista de verdade e que ele somado de kte esteja alinhado com endereços pares.
- **MIPS:** sw $t1, 0($t2)
  C: void sh(uint32_t address, int16_t kte, int32_t dado);
  Esta instrução captura os 32 bits (*word*) menos significativos armazenados em $t1 e o guarda no endereço de memória armazenado em $t2. Para simular isso criamos essa função em C em que address toma o lugar de $t2, kte toma o lugar do 0 (acesso indexado) e dado toma o lugar de $t1. Para essa função decidi escrever byte a byte fazendo um cast do vetor de *words* para um vetor de *bytes*. Essa função exige que o endereço de memória escrito exista de verdade e que ele somado de kte esteja alinhado com endereços múltiplos de 4.
- **MIPS:** lbu $t1, 0($t2)
  C: uint32_t lbu(uint32_t address, int16_t kte);
  Esta instrução captura os 8 bits (*byte*) menos significativos armazenados no endereço de $t2 + 0 (index), extende o número até 32 bits com 0 e o guarda em $t1. Para simular isso criamos essa função em C em que address toma o lugar de $t2, kte toma o lugar do 0 (acesso indexado) e retorno da função toma o lugar de $t1. Para essa função decidi fazer o cast do vetor de *words* para um vetor de *bytes* sem sinal e acessar o endereço + index diretamente. Essa função exige que o endereço de memória escrito exista de verdade.
- **MIPS:** lb $t1, 0($t2)
  C: int32_t lb(uint32_t address, int16_t kte);
  Esta instrução captura os 8 bits (*byte*) menos significativos armazenados no endereço de $t2 +

0 (index), extende o número até 32 bits com o sinal (bit mais significativo) e o guarda em $t1. Para simular isso criamos essa função em C em que address toma o lugar de $t2, kte toma o lugar do 0 (acesso indexado) e retorno da função toma o lugar de $t1. Para essa função decidi fazer o cast do vetor de *words* para um vetor de *bytes* e acessar o endereço + index diretamente. Essa função exige que o endereço de memória escrito exista de verdade.

- **MIPS:** lhu $t1, 0($t2)
  C: uint32_t lbu(uint32_t address, int16_t kte);
  Esta instrução captura os 16 bits (*half word*) menos significativos armazenados no endereço de $t2 + 0 (index), extende o número até 32 bits com 0 e o guarda em $t1. Para simular isso criamos essa função em C em que address toma o lugar de $t2, kte toma o lugar do 0 (acesso indexado) e retorno da função toma o lugar de $t1. Para essa função decidi fazer o cast do vetor de *words* para um vetor de *half words* sem sinal e acessar o endereço + index divididos por 2. Essa função exige que o endereço de memória escrito exista de verdade e que ele somado de kte esteja alinhado com endereços pares.
- **MIPS:** lh $t1, 0($t2)
  C: int32_t lh(uint32_t address, int16_t kte);
  Esta instrução captura os 16 bits (*half word*) menos significativos armazenados no endereço de $t2 + 0 (index), extende o número até 32 bits com o sinal (bit mais significativo) e o guarda em $t1. Para simular isso criamos essa função em C em que address toma o lugar de $t2, kte toma o lugar do 0 (acesso indexado) e retorno da função toma o lugar de $t1. Para essa função decidi fazer o cast do vetor de *words* para um vetor de *half words* e acessar o endereço + index divididos por 2. Essa função exige que o endereço de memória escrito exista de verdade e que ele somado de kte esteja alinhado com endereços pares.
- **MIPS:** lw $t1, 0($t2)
  C: int32_t lw(uint32_t address, int16_t kte);
  Esta instrução captura os 32 bits (*word*) armazenados no endereço de $t2 + 0 (index) e o guarda em $t1. Para simular isso criamos essa função em C em que address toma o lugar de $t2, kte toma o lugar de 0 (acesso indexado) e retorno da função toma o lugar de $t1. Para essa função decidi acessar no vetor de memória o endereço + index divididos por 4. Essa função exige que o endereço de memória escrito exista de verdade e que ele somado de kte esteja alinhado com endereços múltiplos de 4.
- **C:** dump_mem(uint32_t add, uint32_t size);
  Essa função imprime na tela um pedaço da memória específico entre add e add + size, *word* por *word*. Ambos os argumentos de entrada precisam ser múltiplos de 4 para essa função funcionar sem emitir erros.

Testes e Resultados

As funções foram testadas com os testes sugeridos pelo professor e mais alguns feitos por mim.

Teste do professor

Todas as funções de escrita escreveram corretamente na memória como pode ser visto pelo *output* da função dump_mem. Após isso testamos a leitura de alguns desses mesmos *bytes* na memória e podemos conferir que estavam fazendo a leitura corretamente. A partir da impressão do decimal de *half words* e *bytes* de leituras com e sem sinal, podemos constatar de que as funções estavam fazendo a extensão de sinal corretamente.
Código de teste:

```
printf("Testes do Professor\n\n");
// Teste 1: Escrita (Save)
sb(0, 0, 0x04); sb(0, 1, 0x03); sb(0, 2, 0x02); sb(0, 3, 0x01);
sb(4, 0, 0xFF); sb(4, 1, 0xFE); sb(4, 2, 0xFD); sb(4, 3, 0xFC);
sh(8, 0, 0xFFF0); sh(8, 2, 0x8C);
sw(12, 0, 0xFF);
sw(16, 0, 0xFFFF);
sw(20, 0, 0xFFFFFFFF);
sw(24, 0, 0x80000000);

// Testar Estado da memória após escrita
dump_mem(0, 28);
/*
deve resultar em :
mem[0] = 01020304
mem[1] = fcfdfeff
mem[2] = 008cfff0
mem[3] = 000000FF
mem[4] = 0000FFFF
mem[5] = FFFFFFFF
mem[6] = 80000000
*/

// Testar leitura
printf("| lb(0,0):  0x%08X | lb(0,1):  0x%08X | lb(0,2):  0x%08X | lb(0,3):  0x%08X |\n",
 lb(0, 0), lb(0, 1), lb(0, 2), lb(0, 3));
printf("| lb(0,0):  %10d | lb(0,1):  %10d | lb(0,2):  %10d | lb(0,3):  %10d |\n",
 lb(0, 0), lb(0, 1), lb(0, 2), lb(0, 3));

printf("| lb(4,0):  0x%08X | lb(4,1):  0x%08X | lb(4,2):  0x%08X | lb(4,3):  0x%08X |\n",
 lb(4, 0), lb(4, 1), lb(4, 2), lb(4, 3));
```

```
printf("| lb(4,0):  %10d | lb(4,1):  %10d | lb(4,2):  %10d | lb(4,3):  %10d |\n",
 lb(4, 0), lb(4, 1), lb(4, 2), lb(4, 3));
printf("| lbu(4,0): %10d | lbu(4,1): %10d | lbu(4,2): %10d | lbu(4,3): %10d |\n",
 lbu(4, 0), lbu(4, 1), lbu(4, 2), lbu(4, 3));

printf("| lh(8,0):  0x%08X | lh(8,2):  0x%08X |\n", lh(8, 0), lh(8, 2));
printf("| lh(8,0):  %10d | lh(8,2):  %10d |\n", lh(8, 0), lh(8, 2));
printf("| lhu(8,0): %10d | lhu(8,2): %10d |\n", lhu(8, 0), lhu(8, 2));

printf("| lw(12,0): 0x%08X |\n", lw(12, 0));
printf("| lw(12,0): %10d |\n", lw(12, 0));
printf("| lw(16,0): 0x%08X |\n", lw(16, 0));
printf("| lw(16,0): %10d |\n", lw(16, 0));
printf("| lw(20,0): 0x%08X |\n", lw(20, 0));
printf("| lw(20,0): %10d |\n", lw(20, 0));

printf("-------------------------------------------------------------------------------\n");
// Fim testes do Professor
```

Resultado:

```
Testes do Professor

mem[0] = 01020304
mem[1] = FCFDFEFF
mem[2] = 008CFFF0
mem[3] = 000000FF
mem[4] = 0000FFFF
mem[5] = FFFFFFFF
mem[6] = 80000000
| lb(0,0):  0x00000004 | lb(0,1):  0x00000003 | lb(0,2):  0x00000002 | lb(0,3):  0x00000001 |
| lb(0,0):           4 | lb(0,1):           3 | lb(0,2):           2 | lb(0,3):           1 |
| lb(4,0):  0xFFFFFFFF | lb(4,1):  0xFFFFFFFE | lb(4,2):  0xFFFFFFFD | lb(4,3):  0xFFFFFFFC |
| lb(4,0):          -1 | lb(4,1):          -2 | lb(4,2):          -3 | lb(4,3):          -4 |
| lbu(4,0):        255 | lbu(4,1):        254 | lbu(4,2):        253 | lbu(4,3):        252 |
| lh(8,0):  0xFFFFFFF0 | lh(8,2):  0x0000008C |
| lh(8,0):         -16 | lh(8,2):         140 |
| lhu(8,0):      65520 | lhu(8,2):        140 |
| lw(12,0): 0x000000FF |
| lw(12,0):        255 |
| lw(16,0): 0x0000FFFF |
| lw(16,0):      65535 |
| lw(20,0): 0xFFFFFFFF |
| lw(20,0):         -1 |
-------------------------------------------------------------------------------
```

Testes de Reescrita

Eu decidi pegar o próximo endereço disponível sem uso na memória para testar a reescrita de valores. Utilizei funções de escrita e leitura de *byte* para conferir se a reescrita aconteceria. Pela saída do programa eu percebi que a reescrita ocorreu tranquilamente.

Código:

```
// Escrita e leitura de bytes
printf("Teste de Escrita e leitura de bytes\n\n");

sb(28, 0, 0xF0); sb(28, 1, 0xCA); sb(28, 2, 0xF0); sb(28, 3, 0xFA);
printf("| lb(28,0):  0x%08X | lb(28,1):  0x%08X | lb(28,2):  0x%08X | lb(28,3):  0x%08X |\n",
 lb(28, 0), lb(28, 1), lb(28, 2), lb(28, 3));
printf("| lb(28,0):  %10d | lb(28,1):  %10d | lb(28,2):  %10d | lb(28,3):  %10d |\n",
 lb(28, 0), lb(28, 1), lb(28, 2), lb(28, 3));
printf("| lbu(28,0): %10d | lbu(28,1): %10d | lbu(28,2): %10d | lbu(28,3): %10d |\n",
 lbu(28, 0), lbu(28, 1), lbu(28, 2), lbu(28, 3));

printf("-------------------------------------------------------------------------------\n");

// Reescrita e leitura de bytes
printf("Teste de Reescrita e leitura de bytes\n\n");

sb(28, 0, 0x0F); sb(28, 1, 0xAC); sb(28, 2, 0x0F); sb(28, 3, 0xAF);
printf("| lb(28,0):  0x%08X | lb(28,1):  0x%08X | lb(28,2):  0x%08X | lb(28,3):  0x%08X |\n",
 lb(28, 0), lb(28, 1), lb(28, 2), lb(28, 3));
printf("| lb(28,0):  %10d | lb(28,1):  %10d | lb(28,2):  %10d | lb(28,3):  %10d |\n",
 lb(28, 0), lb(28, 1), lb(28, 2), lb(28, 3));
printf("| lbu(28,0): %10d | lbu(28,1): %10d | lbu(28,2): %10d | lbu(28,3): %10d |\n",
 lbu(28, 0), lbu(28, 1), lbu(28, 2), lbu(28, 3));

printf("-------------------------------------------------------------------------------\n");
```

Resultado:

```
Teste de Escrita e leitura de bytes

| lb(28,0):  0xFFFFFFF0 | lb(28,1):  0xFFFFFFCA | lb(28,2):  0xFFFFFFF0 | lb(28,3):  0xFFFFFFFA |
| lb(28,0):         -16 | lb(28,1):         -54 | lb(28,2):         -16 | lb(28,3):          -6 |
| lbu(28,0):        240 | lbu(28,1):        202 | lbu(28,2):        240 | lbu(28,3):        250 |
--------------------------------------------------------------------------------
Teste de Reescrita e leitura de bytes

| lb(28,0):  0x0000000F | lb(28,1):  0xFFFFFFAC | lb(28,2):  0x0000000F | lb(28,3):  0xFFFFFFAF |
| lb(28,0):          15 | lb(28,1):         -84 | lb(28,2):          15 | lb(28,3):         -81 |
| lbu(28,0):         15 | lbu(28,1):        172 | lbu(28,2):         15 | lbu(28,3):        175 |
--------------------------------------------------------------------------------
```

Teste de Limites

Nesse teste decidi testar se as funções iriam emitir erros caso tentássemos escrever fora da memória de 4096 *words*. Eu testei todas as funções com limites superior e inferior e constatei de que elas funcionaram corretamente.

Código:

```
// Teste de Limites
printf("Teste de limites\n\n");
printf("> Deve imprimir 8 erros e depois 10 erros alternados com zeros\n\n");
dump_mem(-4, 4);                // Esse endereço não existe e deve imprimir um erro
dump_mem(MEM_SIZE * 4, 4);      // Esse endereço não existe e deve imprimir um erro
sw(-4, 0, 0xFFFFFFFF);          // Esse endereço não existe e deve imprimir um erro
sw(MEM_SIZE * 4, 0, 0xFFFFFFFF); // Esse endereço não existe e deve imprimir um erro
sh(-4, 0, 0xFFFF);              // Esse endereço não existe e deve imprimir um erro
sh(MEM_SIZE * 4, 0, 0xFFFF);    // Esse endereço não existe e deve imprimir um erro
sb(-4, 0, 0xFF);                // Esse endereço não existe e deve imprimir um erro
sb(MEM_SIZE * 4, 0, 0xFF);      // Esse endereço não existe e deve imprimir um erro
printf("%d\n", lw(-4, 0));          // Esse endereço não existe e deve imprimir um erro e retornar
printf("%d\n", lw(MEM_SIZE * 4, 0)); // Esse endereço não existe e deve imprimir um erro e retornar
printf("%d\n", lh(-4, 0));              // Esse endereço não existe e deve imprimir um erro e reto
printf("%d\n", lh(MEM_SIZE * 4, 0));    // Esse endereço não existe e deve imprimir um erro e reto
printf("%d\n", lhu(-4, 0));             // Esse endereço não existe e deve imprimir um erro e reto
printf("%d\n", lhu(MEM_SIZE * 4, 0));   // Esse endereço não existe e deve imprimir um erro e reto
printf("%d\n", lb(-4, 0));                // Esse endereço não existe e deve imprimir um erro e re
printf("%d\n", lb(MEM_SIZE * 4, 0));      // Esse endereço não existe e deve imprimir um erro e re
printf("%d\n", lbu(-4, 0));               // Esse endereço não existe e deve imprimir um erro e re
printf("%d\n", lbu(MEM_SIZE * 4, 0));     // Esse endereço não existe e deve imprimir um erro e re

printf("--------------------------------------------------------------------------------\n");
```

Resultado:

```
Teste de limites

> Deve imprimir 8 erros e depois 10 erros alternados com zeros

Erro: acesso a endereços inexistentes!
Erro: acesso a endereços inexistentes!
Erro: acesso a endereços inexistentes!
Erro: acesso a endereços inexistentes!
Erro: acesso a endereços inexistentes!
Erro: acesso a endereços inexistentes!
Erro: acesso a endereços inexistentes!
Erro: acesso a endereços inexistentes!
Erro: acesso a endereços inexistentes!
0
Erro: acesso a endereços inexistentes!
0
Erro: acesso a endereços inexistentes!
0
Erro: acesso a endereços inexistentes!
0
Erro: acesso a endereços inexistentes!
0
Erro: acesso a endereços inexistentes!
0
Erro: acesso a endereços inexistentes!
0
Erro: acesso a endereços inexistentes!
0
Erro: acesso a endereços inexistentes!
0
Erro: acesso a endereços inexistentes!
0
--------------------------------------------------------------------------------
```

Teste de desalinhamento

Nesse teste eu conferi se as funções seriam capazes de emitir um erro caso tentasse escrever na memória de forma desalinha e ler também. Nesse caso as funções que acessam *half words* só podem acessar endereços pares e funções que acessam *words* só podem acessar endereços múltiplos de 4. Após realizar o teste constatei que estavam emitindo erros corretamente e não executando a instrução.

Código:

```
// Teste de desalinhamento
printf("Teste de desalinhamento\n\n");

printf("> Devem ocorrer 7 erros de desalinhamento\n\n");

sh(3, 0, 0xFFFF); // Esse endereço não está alinhado e deve dar erro
sh(2, 1, 0xFFFF); // Esse offset não está alinhado e deve dar erro
sh(4, 3, 0xFFFF); // Esse offset não está alinhado e deve dar erro

sw(3, 0, 0xFFFFFFFF); // Esse endereço não está alinhado e deve dar erro
sw(4, 1, 0xFFFFFFFF); // Esse offset não está alinhado e deve dar erro
sw(4, 2, 0xFFFFFFFF); // Esse offset não está alinhado e deve dar erro
sw(4, 3, 0xFFFFFFFF); // Esse offset não está alinhado e deve dar erro

printf("\n> Devem ocorrer 10 erros alternados de 0s\n\n");

printf("%d\n", lh(3, 0)); // Esse endereço não está alinhado e deve dar erro
printf("%d\n", lh(4, 1)); // Esse offset não está alinhado e deve dar erro
printf("%d\n", lh(4, 3)); // Esse offset não está alinhado e deve dar erro
printf("%d\n", lhu(3, 0)); // Esse endereço não está alinhado e deve dar erro
printf("%d\n", lhu(4, 1)); // Esse offset não está alinhado e deve dar erro
printf("%d\n", lhu(4, 3)); // Esse offset não está alinhado e deve dar erro

printf("%d\n", lw(3, 0)); // Esse endereço não está alinhado e deve dar erro
printf("%d\n", lw(4, 1)); // Esse offset não está alinhado e deve dar erro
printf("%d\n", lw(4, 2)); // Esse offset não está alinhado e deve dar erro
printf("%d\n", lw(4, 3)); // Esse offset não está alinhado e deve dar erro

printf("--------------------------------------------------------------------------------\n");
```

Resultado:

```
Teste de desalinhamento

> Devem ocorrer 7 erros de desalinhamento

Erro: escrita não alinhada.
O endereço + offset de dados precisam ser múltiplos de 2.
Erro: escrita não alinhada.
O endereço + offset de dados precisam ser múltiplos de 2.
Erro: escrita não alinhada.
O endereço + offset de dados precisam ser múltiplos de 2.
Erro: escrita não alinhada.
O endereço + offset de dados precisam ser múltiplos de 4.
Erro: escrita não alinhada.
O endereço + offset de dados precisam ser múltiplos de 4.
Erro: escrita não alinhada.
```

```
O endereço + offset de dados precisam ser múltiplos de 4.
Erro: escrita não alinhada.
O endereço + offset de dados precisam ser múltiplos de 4.

> Devem ocorrer 10 erros alternados de 0s

Erro: acesso não alinhado.
O endereço + offset de dados precisam ser múltiplos de 2.
0
Erro: acesso não alinhado.
O endereço + offset de dados precisam ser múltiplos de 2.
0
Erro: acesso não alinhado.
O endereço + offset de dados precisam ser múltiplos de 2.
0
Erro: acesso não alinhado.
O endereço + offset de dados precisam ser múltiplos de 2.
0
Erro: acesso não alinhado.
O endereço + offset de dados precisam ser múltiplos de 2.
0
Erro: acesso não alinhado.
O endereço + offset de dados precisam ser múltiplos de 2.
0
Erro: acesso não alinhado.
O endereço + offset de dados precisam ser múltiplos de 4.
0
Erro: acesso não alinhado.
O endereço + offset de dados precisam ser múltiplos de 4.
0
Erro: acesso não alinhado.
O endereço + offset de dados precisam ser múltiplos de 4.
0
Erro: acesso não alinhado.
O endereço + offset de dados precisam ser múltiplos de 4.
0
--------------------------------------------------------------------------------
```

Teste de alinhamento

Nesse teste eu conferi se as funções iriam acessar endereços que não estariam alinhados sozinhos, porém somados do index (kte) estariam alinhados. Eu tive que ir ao MIPS testar se era esse o comportamento e realmente é. O MIPS irá acessar o endereço se ele for válido após somar endereço com index. Após realizar os testes eu pude perceber que as funções estavam funcionando corretamente.

Código:

```
// Teste de alinhamento
printf("Teste de alinhamento\n\n");
sh(31, 1, 0x4567);      // Grava meia palavra em 32
sh(35, -1, 0x0123);     // Grava meia palavra em 34
sw(37, -1, 0x89ABCDEF); // Grava uma palavra em 36
sw(39, 1, 0xAAAAAAAA);  // Grava uma palavra em 40

dump_mem(32, 12); // Escreve esse fragmento de memória que acabamos de escrever
/*
deve resultar em :
mem[8] = 01234567
mem[9] = 89ABCDEF
mem[10] = AAAAAAAA
*/

printf("| lh(39, 1):  0x%08X | lh(41, -1):  0x%08X |\n", lh(39, 1), lh(41, -1));
printf("| lh(39, 1):  %10d | lh(41, -1):  %10d |\n", lh(39, 1), lh(41, -1));
printf("| lhu(39, 1): %10d | lhu(41, -1): %10d |\n", lhu(39, 1), lhu(41, -1));

printf("| lw(37, -1): 0x%08X |\n", lw(37, -1));
printf("| lw(37, -1): %10d |\n", lw(37, -1));

printf("--------------------------------------------------------------------------------\n");
```

Resultado:

```
Teste de alinhamento

mem[8] = 01234567
mem[9] = 89ABCDEF
mem[10] = AAAAAAAA
| lh(39, 1):  0xFFFFAAAA | lh(41, -1):  0xFFFFAAAA |
| lh(39, 1):      -21846 | lh(41, -1):      -21846 |
| lhu(39, 1):      43690 | lhu(41, -1):      43690 |
| lw(37, -1): 0x89ABCDEF |
| lw(37, -1): -1985229329 |
--------------------------------------------------------------------------------
```

Sobre o código

Sistema Operacional

O programa foi desenvolvido em um *Windows Subsystem for Linux* de Ubuntu. Basicamente é um Ubuntu embutido no Windows 10, então o código é garantido de funcionar em Ubuntu, porém não se tem tanta certeza quanto a Windows.

Compilador

O programa é compilado utilizando o g++ (gcc version 4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04.4)).

IDE

O programa foi desenvolvido no editor de texto Visual Studio Code. Esse editor de texto é praticamente uma IDE com as extensões corretas.