DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

UNIVERSIDADE FEDERAL DE MINAS GERAIS, BRAZIL

(UFMG)

Máquina Simple86

Um emulador para a arquitetura exemplo Simple86

Belo Horizonte, 13 de Maio de 2016

Autores:

Breno Rodrigues Marques da Silva

Eugênio Pacceli Reis da Fonseca

1 - Introdução

Para o primeiro trabalho prático de Software Básico, matéria lecionada por Daniel Macedo pelo Departamento de Ciência da Computação, da Universidade Federal de Minas Gerais, foi proposta a implementação de um emulador de uma arquitetura exemplo chamada Simple86.

A arquitetura Simple86 contém um conjunto de instruções de tamanho variado, e, as palavras vindas da memória, assim como os registradores, contém 16 bits. As especificações completadas da arquitetura estão descritas no enunciado que esse trabalho responde, “TP1 – Software Básico.pdf”.

A implementação do emulador para a Simple86, foi feita na linguagem de programação C++, para aproveitamento dos conceitos do paradigma orientado por objetos.

2 – Escolha da linguagem e do paradigma

A linguagem C++, por implementar o paradigma orientado por objetos, permite uma modularização mais completa do código, sem perder os poderes que herdou da linguagem C.

Fazer o emulador dividido por módulos (descritos pelas classes), significou reduzir o problema em vários pedaços menores. O controle de acesso por meio das definições de público e privado de cada classe, permite à cada módulo garantir seu bom funcionamento, sem perder a capacidade de interagir com os outros. O conceito de instanciar objetos das classes (módulos) permite que condições iniciais sejam satisfeitas na criação de cada instância de um módulo, e o reaproveitamento de código torna a tarefa de programar mais produtiva.

A divisão por módulos também permitiu que a equipe trabalhasse em trechos de códigos diferentes, por integrante, e através da definição de interfaces e boas normas, foi possível produzir código válido sem precisar saber de todos os detalhes das outras

partes do programa, dado que essas também devem funcionar como unidades separadas e manter suas condições de validez.

3 - Apresentação da implementação

A implementação do emulador foi dividida em quatro arquivos diferentes, separados por classe, sendo que o arquivo “main.cpp”, é responsável por instanciar cada classe, montar o emulador, carregar o arquivo de entrada, e dar início à execução da máquina.

As instâncias dos módulos Memory, Execute e FetchAndDecode devem operar em conjunto, sendo FetchAndDecode o “maestro” da execução sequencial das instruções recebidas.

A palavra mínima da Simple86 é representada pelo tipo int16_t, da biblioteca do padrão C++ 2011, chamada “cstdint”, e todos os módulos trabalham com valores desse tipo. O int16_t representa um inteiro signed de 16 bits, em qualquer compilador com suporte ao padrão.

Todos os arquivos de código do programa estão amplamente comentados.

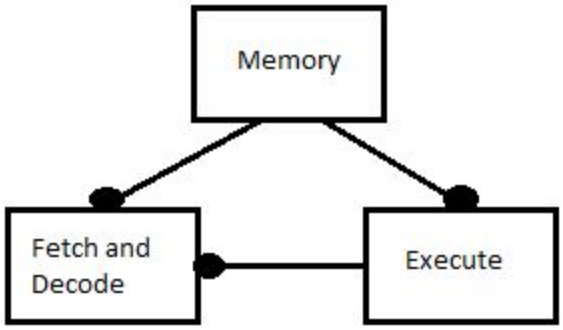

Figura 1: FetchAndDecode opera com uma instância de Memory e outra de Execute, essa instância de Execute, por sua vez, também deve operar nessa mesma instância de Memory.

Classe Memory (definida em Memory.h)

A classe Memory representa o módulo de memória da máquina Simple86. Ela implementa cada registrador como um atributo privado, assim como um vetor de 1000 palavras de 16 bits. O acesso a esses espaços e valores é feito por métodos públicos.

Os valores e métodos públicos da classe Memory são apresentados a seguir:

- `enum Register { AX, AL, AH, BX, BL, BH, CX, CL, CH, BP, SP, IP, ZF, SF }`

Um enumerador com os nomes de todos os registradores da máquina. Todos os objetos e códigos externos ao módulo devem passar um dos valores desse enumerador, para as funções de acesso aos registradores, afim de poder acessar os registradores de fato.

- `Memory()`

Toda instância de objetos em C++ representa uma chamada implícita à um método com o mesmo nome da classe, conhecido como construtor. O construtor garante que o módulo possa começar com um estado inicial definido, e, para o caso de Memory, significa ajeitar os valores de SP, BP e IP, de acordo com as especificações do enunciado.

- `Register getRegName(int16_t address)`

Através de um parâmetro de instrução, que representa um registrador, a função retorna o nome do registrador (valor do enumerador Register) equivalente. Permite a decodificação do valor vindo da instrução para o nome mapeado no enumerador Register.

- `int16_t getRegister(Register reg)`

Retorna o valor armazenado no registrador representado pelo valor de Register passado. Trata também o acesso às partes baixas e altas dos registradores AX, BX e CX, através de máscaras e shifts. Se o valor pedido for de um sub registrador de 8 bits, os 8 bits mais significativos do retorno são 0s, e o valor estará nos bits mais significativos.

- `int16_t setRegister(Register reg, int16_t newValue)`

Guarda uma palavra no registrador especificado. Pode também guardar meia palavra se os registradores alvos forem altos, ou baixos, por meio de shifts e máscaras.

- `int16_t readMemory(int16_t source)`

Lê uma palavra da posição de memória especificada por parâmetro. Existem 1000 posições de palavras de 16 bits na memória.

- `int16_t writeMemory(int16_t destination, int16_t newValue)`

Escreve na posição de memória especificada, uma palavra.

Classe Execute (definida em Execute.h)

Implementa a execução de cada instrução da Simple86, especificadas no enunciado. Os valores e o tipo de operação já devem estar decodificados. O resultado dessas funções é a modificação dos estados do módulo Memory, recebido na criação de uma instância de Execute.

Os parâmetros chamados destiny são registradores ou posições de memória que vão receber os resultados das operações (distinguíveis pelo parâmetro operandType, que informa à função quais são os tipos de parâmetros que a função recebe). Os parâmetros de nome source são, ou posições de memória, registradores, ou valores absolutos, usados para gerar o resultado das computações (distinguíveis também pelo operandType).

Exemplos de valores de operandType são opRM (memory to register), opRR (register to register), opRI (integer to register), entre outras combinações.

A lista das assinaturas dos métodos públicos principais de Execute (equivalentes às especificações do enunciado):

- `void mov(int16_t destiny, int16_t source, int16_t operandType)`
- `void add(int16_t destiny, int16_t source, int16_t operandType)`
- `void sub(int16_t destiny, int16_t source, int16_t operandType)`
- `void mul(int16_t source, int16_t operandType)`
- `void div(int16_t source, int16_t operandType)`

- `void binaryAnd(int16_t destiny, int16_t source, int16_t operandType)`
- `void binaryOr(int16_t destiny, int16_t source, int16_t operandType)`
- `void binaryNot(int16_t destiny, int16_t operandType)`
- `void cmp(int16_t source1, int16_t source2, int16_t operandType)`
- `void jmp(int16_t destiny)`
- `void jz(int16_t destiny)`
- `void js(int16_t destiny)`
- `void call(int16_t destiny)`
- `void ret()`
- `void push(int16_t source, int16_t operandType)`
- `void pop(int16_t destiny, int16_t operandType)`
- `void dump()`
- `void read(int16_t destiny, int16_t operandType)`
- `void write(int16_t source, int16_t operandType)`
- `void halt()`

Para passar valores menores (8 bits), é necessário apenas soma-los à uma variável do tipo int16_t equivalente a 0, ou simplesmente fazer um cast.

Classe FetchAndDecode (definida em FetchAndDecode.h)

A classe FetchAndDecode é responsável por sequenciar e coordernar a execução das instruções recebidas.

Inclui métodos para decodificar a instrução, e passar os parâmetros corretos para as funções de uma instância do módulo Execute.

- `bool is16bitsInstruction(int8_t opCode)`
- `bool is32bitsInstruction(int8_t opCode)`
- `bool is48bitsInstruction(int8_t opCode)`
- `void initMachine()`

initMachine inicializa a execução do programa recebido e coordena toda a execução, até um halt.

Main e montador (main.cpp)

O arquivo main.cpp é o ponto de entrada do emulador Simple86, contêm a função populateMemory(char* file), que implementa o montador: recebe um arquivo com um programa em binário Simple86, e povoa um módulo Memory; e a função main, que junta os módulos com suas devidas instâncias e hierarquia, e depois chama initMachine(), dando início à todo fluxo de execução.

4 – Compilação

O programa foi escrito no padrão C++11, e é necessário um compilador com suporte à tal padrão da linguagem C++. Tudo foi testado num ambiente Linux, com o g++ 5.3.1. O arquivo Makefile trata de chamar o compilador g++ com os parâmetros necessários para a compilação correta do projeto.

```
g++ -Wall -std=c++11 main.cpp -o Simula86_Emulator
```

Os arquivos testExecute.cpp e testMemory.cpp foram usados para testar a validade dos módulos, bem como os exemplos tp1ex1.sa e tp1ex2_fatorial.sa. Esses arquivos não são compilados junto com o projeto. Para compilar testExecute.cpp ou testMemory.cpp, basta pedir para o g++.