MC404 2008s2 - Projeto 2

Grupo 30
071294 - Jorge Augusto Hongo
072201 - Raphael Kubo da Costa

December 4, 2008

1 Introdução

Este relatório detalha algumas escolhas de design e implementação e problemas enfrentados durante a elaboração do segundo projeto de MC404 do segundo semestre de 2008.

Este projeto foi escrito e testado diretamente no **Linux**, utilizando registradores de 32 bits e as chamadas de sistema nativas do sistema operacional. Os comentários no código estão em inglês por simples convenção e costume.

2 Estrutura do código

As tarefas de modularizar o código e dividi-lo em pequenas unidades funcionais provou-se bem mais difícil que no primeiro projeto.

Ao contrário do projeto anterior, a maior parte das funções e macros encontra-se no arquivo *main.asm*. Arquivos auxiliares, como *util.h* ou *string.asm* contêm funções e macros agrupadas por semelhança de funcionalidades.

No final, o código foi dividido nos seguintes arquivos:

disasm.asm Subrotinas, definições e macros relaciondas à manipulação do arquivo de disassembly.

file.asm Subrotinas relaciondas à manipulação de arquivos em geral.

instr.h Definições estáticas da estrutura de cada opcode e listagem de registradores e modos de endereçamento.

main.asm Subrotina principal, que controla e chama as demais. Possui o *core* do programa.

math.h Macros relacionadas a operações matemáticas.

opcodes2asm.py Script para geração do arquivo *opcodes.h* com base em *opcodes.dat*.

opcodes.dat Arquivo de texto usado como base para a geração da listagem de opcodes em Assembly.

opcodes.h Gerado por *opcodes2asm.py*. Contém a lista de opcodes, seus atributos e os grupos usados pelo programa.

string.{asm,h} Subrotinas, definições e macros relaciondas à manipulação de strings de maneira simplificada.

syscalls.h Macros e definições para as chamadas de sistema do Linux (abertura e fechamento de arquivos, seeks e afins).

Em geral, em todas as macros e subrotinas, tentou-se preservar o conteúdo dos registradores passados, exceto em casos em que era necessário retornar algum valor no registrador *EAX*.

3 Funcionamento do programa

A estrutura do programa tem como base o Disasm, *disassembler* escrito por Marius Gedminas, disponível em **http://gemin.as/disasm**. Seu programa, com versões em Pascal e Assembly, desmonta arquivos .COM para o formato de Assembly utilizado pelo TASM.

Do Disasm foi utilizada em especial a idéia da criação de camadas de indireção para referenciar as propriedades e informações de cada instrução do 8086. Para alguns trechos e principalmente para consulta foi utilizada a biblioteca **libASM**, de Christian Fowelin (*http://www.fowelin.de/christian/computer/libasm*).

3.1 Algoritmo e estrutura utilizados

Basicamente, o programa realiza uma busca na tabela de opcodes de *opcodes.h*: cada byte lido possui uma entrada correspondente na tabela com informações como o tamanho dos operandos, seus tipos e se pertence a algum grupo de opcodes.

Todas as informações da entrada são processadas e guardadas em variáveis apropriadas. Essas informações são então utilizadas para escrever no arquivo de disassembly a instrução correspondente e seus operandos devidamente formatados.

Nomes de registradores são definidos em *instr.h*, e há estruturas como `ARRAY_16BITREGS` ou `ARRAY_CONSTARGS` que possuem o endereço de memória das instruções ou strings dos registradores desejados. Por sua vez, os opcodes disponíveis na tabela costumam conter a posição de determinado item dessas estruturas. Para a escrita no arquivo final, é necessário passar por várias camadas de indireção.

Grande parte da complexidade do projeto advém da estrutura dos opcodes do 8086: como este programa decodifica uma instrução por vez, sem informações sobre as anteriores ou as próximas, é necessário realizar muitas checagens sobre os dados disponíveis em tabela.

4 Conclusão

4.1 Complexidade de código

Devido à maneira como as instruções são representadas no 8086 e como os opcodes são estruturados, muitas macros ficaram bastante grandes e com uma quantidade maior que a desejável de checagens de parâmetros. Em uma linguagem de mais alto nível, essa quantidade de código e checagem ficaria grande, e em Assembly é possível entender o tamanho a que tenha chegado.

Fez-se bastante uso do preprocessador do nasm, tanto para definir macros simples quanto para realizar checagens de parâmetros em macros maiores ou mais complexas, como `StoreData`. Na medida do possível, tentou-se criar macros e procedimentos que facilitassem o desenvolvimento. Como exemplo, foram criados *wrappers* para a macro `sys_write`, por sua vez já uma abstração sobre a chamada de sistema `write`: `print_string`, `printnl` e `write_string` tornam a programação um pouco menos dolorosa. A macro `exec`, por sua vez, torna a chamada de uma função que recebe argumentos na pilha mais parecida com uma chamada comum de linguagem de alto nível.

O projeto e o algoritmo (não muito eficiente) são bastante simples. Em C ou em alguma linguagem qualquer de mais alto nível, seria possível implementar o projeto em algumas horas e com um número muito menor de linhas de código.

4.2 Problemas de remontagem com o nasm

Não foi possível realizar testes satisfatórios com o nasm após a desmontagem com o programa. Algumas instruções, como `mov bx, bx`, precisam ser representadas como um opcode mas o nasm utiliza outro equivalente. Por isso, programas como o quinto arquivo de teste (*05.COM*) não puderam ser remontados com sucesso.

4.3 Bugs

Durante os testes, percebeu-se que, caso o último (ou os últimos) bytes correspondam a instruções e não sejam completados (isso ocorre quando o final do arquivo não corresponde a código, mas a dados, e o último byte é algo como 0x00 (ADD), que necessitaria de outros bytes como argumentos). Mesmo depois do último byte, o programa procura mais bytes e os escreve (0x00). Na remontagem, não há grandes problemas, uma vez que os novos bytes não são referenciados.

Embora não se trate exatamente de um bug, não foi possível implementar a separação clara entre a seção de código e a seção de dados, como ocorre no **Disasm**.

4.4 Código aberto

Da mesma maneira que o projeto anterior, que encontra-se hospedado no **Launchpad** com seu código aberto, este projeto teve suas versões gerenciadas pelo **git**

e está hospedado no github em `‘http://github.com/rakuco/decompify‘`.

O programa foi batizado como **decompify** e foi liberado sob a licença GPLv3.