Trabalho Prático 1 - Montador

> Lucas Machado
> Yuri Niitsuma
> Nathalia Campos

Introdução

O trabalho prático consiste em criar um montador para o Wombat2, uma arquitetura simplificado.
Foi utilizado a linguagem *C++* para criar o montador e testado no **Eufrates** (computador do CRC).

Makefile

Utilizei o GenericMakefile como template com os seguintes comandos.

- **make**: Compila o montador. Reaproveita os arquivos "*.o*" que não forem modificado.
- **make clean**: Elimina todos os arquivos compilados.
- **make about**: Executa ./montador apenas para imprimir o *help*.

```
Montador Wombat2
(C) 2016 Yuri Niitsuma <ignitzhjfk@gmail.com>
         Lucas Machado <lucaspedro.machado@gmail.com>
         Nathalia Campos <nathalia.campos.braga@gmail.com>
    Uso: montador NOME_ARQUIVO [-o NOME_ARQUIVO_SAIDA] [-v]
```

- **make run**: Executa o montador normalmente nos arquivos de teste *all.a*, *media.a* e *pa.a*.
- **make v**: Executa em modo verbose utilizando o arquivo padrão da especificação "W2-1.a", imprime todo o processo da montagem incluindo a tabela de símbolos.

Executar diretamente

O montador pode ser executado diretamente no formato.
`./montador NOME_ARQUIVO [-o NOME_ARQUIVO_SAIDA] [-v]`

Se não for passado o nome do arquivo de saída (caso deseje executá-lo diretamente pela linha de comando) ele será gravado no arquivo padrão "*exec.mif*" dentro da pasta "*tst*".

Na compilação é necessária o uso do **c++11** pois utilizo uma funções para separar os tokens.

Implementação

A implementação do montador foi utilizado o processo de dois passos especificado no livro Organização Estruturada de Computadores do Andrew S. Tanenbaum.

A `main` se encontra no arquivo `montador.cpp` onde abre o arquivo de entrada e saída e chama as funções `pass_one` e `pass_two`. Em que na primeira passada ele localiza a futura posição na memória de todos os símbolos, *labels* e *variáveis*, para inserir na tabela de símbolos `class Symbol` implementada no arquivo `symbol_table`.

Na segunda passada `pass_two` é onde o arquivo de saída é gerado. Os mnemônico são convertidos diretamente

em strings binários pela `class TableOpcode` implementada no arquivo `table_opcodes` e extendida da classe `Symbol` (para utilizar a tabela de símbolo gerada na primeira passada).

A implementação da pseudo-instrução "`.data`" foi convencionado na seguinte forma.

`[VAR_NAME]: .data [TAM] [VALUE]`

- VAR_NAME: Nome da variável.
- TAM: Tamanho de bytes usados
- VALUE: Valor 8 bits inicial.

Caso o TAM seja maior ou igual a 2, o sinal de complemento de dois é extendido com a quantidade de blocos.

Exemplo:

- TAM = 2 com negativo:
  -1 → 11111111 11111111
- TAM = 3 com negativo:
  -2 → 11111111 11111111 11111110
- TAM = 2 com não-negativo:
  2 → 00000000 00000010

Escolhemos o formato de memória `.mif` para facilitar correção e leitura em bits.

Testes

Para testar o montador, foram escritos três programas:

- "media.a": lê dois inteiros na entrada padrão e e retorna o valor medio entre eles;
- "all.a": Ela não faz nada de específico. Contém todas as instruções só para verificar se o binário do Montador está correto com o binário do CPUSim.
- "pa.a": Função pra calcular uma PA de **n** termos tal que **n** é entrada de usuário.
  SOMA = 1 + 2 + ... + n

Retirando os comentários do all_cpusim.mif e comparando utilizando `diff` para encontrar as diferenças recebemos o resultado.

```
diff tst/all.mif tst/all_cpusim.mif
13,15c13
< 05        :  00100000;
< 06        :  00100000;
< 07        :  00100000;
---
> [05..07]:  00100000;
60,62c58
< 34        :  00000000;
< 35        :  00000000;
< [36..FF]:  00000000;
---
> [34..FF]:  00000000;
```

Que só indicam os blocos com bytes repetidos. Pelos testes concluímos que o montador funciona corretamente (excluíndo o .data).

Os dois programas podem ser encontrados no diretório **tst** incluindo as RAMs extraídas do CPUSim.

Referências

- cplusplus Reference C++11
- Organização Estruturada de Computadores - Andrew S. Tanenbaum