Como os dados e programas são armazenados

Podemos verificar o modo que um arquivo binário é armazenado usando a opção -l no montador. Exemplo: nasm -g -f elf64 exemplo1.asm ***-l exemplo1.lst***

Desse modo seria criado um arquivo de lista exemplo1.lst, contendo algumas informações de armazenamento e codigo binário. Mais detalhes sobre comando nasm, digite no terminal (shell): *nasm -h.*

Por exemplo, um fragmento da seção de dados do arquivo de lista, do programa de exemplo 1:

```
Linha    Endereço   Valor em hex
  36  00000009  40660301          dVar1     dd   17000000
  37  0000000D  40548900          dVar2     dd   9000000
  38  00000011  00000000          dResult   dd   0
```

Na primeira linha, o 36 é o número da linha. O próximo número, 0x00000009, é o endereço relativo na área de dados de onde essa variável será armazenada. Como *dVar1* é double word que requer quatro bytes, o endereço da próxima variável é 0x0000000D. A variável *dVar1* usa 4 bytes como endereços *0x00000009*, *0x0000000A*, *0x0000000B* e *0x0000000C*. O restante da linha é a declaração de dados digitada no arquivo de origem da linguagem de montagem original. O 0x40660301 é o valor, em hexadecimal, conforme colocado na memória. Os 17.000.000 são 0x01036640. Lembrando que **a arquitetura é little-endian**, o byte menos significativo (0x40) é colocado no endereço de memória mais baixo, com dois dígitos hexadecimais em cada byte. Como tal, o 0x40 é colocado no endereço 0x00000009, o próximo byte, 0x66, é colocado no endereço 0x0000000A e assim por diante.
Isso pode ser confuso, pois, à primeira vista, o número pode aparecer para trás ou distorcido (dependendo de como é visualizado). Separemos de dois em dois: 40.66.03.01, a seguir para interpretar o valor correto tomamos 01.03.66.40, ou seja, $0x01036640=17000000_d$.
De outro modo, vejamos a figura abaixo:

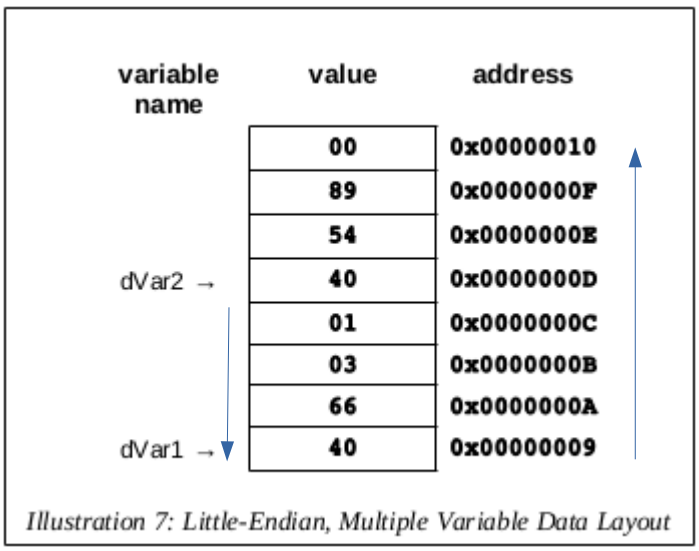

Illustration 7: Little-Endian, Multiple Variable Data Layout

Mais um exemplo, o fragmento da seção text do programa exemplo1.asm.

```
95                                  last:
96 0000005A 48C7C03C000000             mov     rax, SYS_exit
97 00000061 48C7C300000000             mov     rdi, EXIT_SUCCESS
98 00000068 0F05                       syscall
```

Novamente, os números à esquerda são os números das linhas. O próximo número, 0x0000005A, é o endereço relativo de onde a linha de código será colocada.
O próximo número, 0x48C7C03C000000, é a versão em linguagem de máquina da instrução, em hexadecimal, que a CPU lê e entende. O restante da linha é a instrução original da fonte de linguagem assembly.
O rótulo *last:*, não possui uma instrução em linguagem de máquina, pois o rótulo é usado para referenciar um endereço específico e não é uma instrução executável.

Montador de duas passagens

Vamos deixar claro a diferença entre *assembly*, a linguagem que usa mneumônicos relativos as instruções de máquina. Existem vários *assembly*, como o *nasm*, *yasm*, *gas*, etc. Já *assembler* é o montador, aquele que lê e converte para código binário.
O assembler (montador) lê o arquivo de origem e converte cada instrução de linguagem assembly, digitada pelo programador, em um conjunto de 0 e 1 que a que a CPU entende. Esses 0 e 1 são ditos linguagem de máquina. Há uma correspondência individual entre as instruções da linguagem de montagem (assembly) e a linguagem de máquina binária. Esse relacionamento significa que a linguagem de máquina, na forma de um arquivo executável, pode ser convertida novamente em linguagem assembly legível por humanos. Obviamente, os comentários, nomes de variáveis e nomes de rótulos estarão ausentes, portanto, o código resultante pode ser muito difícil de ler.
À medida que o assembler lê cada linha da linguagem assembly, ele gera código de máquina para essa instrução. Isso funcionará bem para instruções que não executam saltos (goto).
No entanto, para instruções que possam alterar o fluxo de controle (por exemplo, instruções IF, saltos incondicionais), o assembler não pode converter a instrução. Por exemplo, dado o seguinte fragmento de código:

```
    mov     rax, 0
    jmp     skipRest
    ...
    ...
skipRest:
```

O código acima é lido linha por linha e convertido para instrução de máquina. O montador lê o arquivo de montagem uma linha por vez, mas não lê a linha em que *skipRest* está definido. De fato, ele nem sabe ao certo se *skipRest* está definido.
Essa situação pode ser resolvida lendo o arquivo de origem do assembly duas vezes. Todo o processo é chamado de montador de duas passagens. As etapas necessárias para cada passe são detalhadas nas seções a seguir.

Primeiro passo

As etapas executadas na primeira passagem variam de acordo com o design do montador.

No entanto, algumas das operações básicas executadas na primeira passagem incluem o seguinte:
• Criar tabela de símbolos
• Expandir macros
• Avaliar expressões constantes

Uma macro é um elemento do programa que é expandido para um conjunto de instruções predefinidas do programador. *Veremos macros posteriormente.*
Uma expressão constante é uma expressão composta inteiramente de constantes. Como a expressão é apenas constante, ela pode ser totalmente avaliada no momento da montagem. Por exemplo, supondo que a constante *BUFF* esteja definida, a instrução a seguir contém uma expressão constante;

mov rax, BUFF+5

Esse tipo de expressão constante é comumente usado em programas grandes ou complexos.
Os endereços são atribuídos a todas as instruções no programa. A tabela de símbolos é uma lista ou tabela de todos os símbolos do programa, nomes de variáveis e etiquetas do programa, e seus respectivos endereços no programa.
Conforme apropriado, algumas diretivas do assembler são processadas na primeira passagem.

Segundo passo

As etapas realizadas na segunda passagem variam com base no design do montador.
No entanto, algumas das operações básicas executadas na segunda passagem incluem o seguinte:
• Geração final de código
• Criação de arquivo de lista (se solicitado)
• Criar arquivo de objeto
O termo geração de código refere-se à conversão da instrução de linguagem de montagem fornecida pelo programador na instrução de linguagem de máquina executável da CPU. Devido à correspondência individual, isso pode ser feito para obter instruções que não usam símbolos na primeira ou o segundo passagem.
Deve-se notar que, com base no design do montador, grande parte da geração de código pode ser feita na primeira passagem ou tudo na segunda passagem. De qualquer maneira, a geração final é realizada na segunda passagem. Isso exigirá o uso da tabela de símbolos para verifique os símbolos do programa e obtenha os endereços apropriados da tabela.
O arquivo de lista, enquanto opcional, pode ser útil para depuração. Se solicitado, ele será gerado na segunda passagem.
Se não houver erros, o arquivo final do objeto será criado na segunda passagem.

Diretivas Assembler

Diretivas do assembler são instruções para o assembler que orientam o assembler a fazer algo. Isso pode ser formatação ou layout. Essas diretivas não são traduzidas em instruções para a CPU.

Linkador ou linkeditor

O *linkador*, às vezes referido como editor de ligação, combinará um ou mais arquivos de objeto em um único arquivo executável. Além disso, todas as rotinas das bibliotecas do usuário ou do sistema

são incluídas conforme necessário. Geralmente usamos o GNU gold linker, *ld*. Em alguns momentos, podemos usar o *gcc* como *linkador* (veremos depois). O comando vinculador apropriado para o programa *exemplo1* é:

ld -g -o exemplo1 exemplo1.o

Observe que *-o* é uma letra minúscula da letra *O*, que pode ser confundida com o número 0.
A opção *-g* é usada para informar o *linkador* a incluir informações de depuração (*debug*) no arquivo executável final. Isso aumenta o tamanho do arquivo executável, mas é necessário para permitir a depuração efetiva. O exemplo *-o* especifica para criar o arquivo executável chamado *exemplo1* (sem extensão). Se a opção *-o <fileName>* for omitida, o arquivo de saída será nomeado *a.out* (por padrão). O *exemplo1.o* é o nome do arquivo do objeto de entrada lido pelo *linkador*. Deve-se observar que o arquivo executável pode ter qualquer nome e não precisa ter o mesmo nome que nenhum dos arquivos do objeto de entrada.

Processo de linkagem

Linkar é o processo fundamental de combinar as soluções menores em uma única unidade executável. Se forem usadas rotinas de usuário ou de biblioteca do sistema, o *linkador* incluirá as rotinas apropriadas. Os arquivos de objeto e as rotinas da biblioteca são combinados em um único módulo executável. O código do idioma da máquina é copiado de cada arquivo de objeto em um único executável.
Como parte da combinação dos arquivos de objeto, o *linkador* deve ajustar os endereços realocáveis conforme necessário. Supondo que existam dois arquivos de origem, o principal e um secundário, contendo algumas funções, as quais foram montadas nos arquivos de objeto ***main.o*** e ***funcs.o***. Quando cada arquivo é montado, as chamadas para rotinas fora do arquivo que está sendo montado são declaradas com a diretiva assembler *extern*. O código não está disponível para uma referência externa e essas referências são marcadas como externas no arquivo de objeto.
O arquivo de lista mostrará um R para esses endereços realocáveis. O vinculador deve satisfazer as referências externas. Além disso, o local final das referências externas deve ser colocado no código.
Por exemplo, se o arquivo de objeto ***main.o*** chamar uma função no arquivo ***funcs.o***, o *linkador* deverá atualizar a chamada com o endereço apropriado, conforme mostrado na ilustração a seguir.

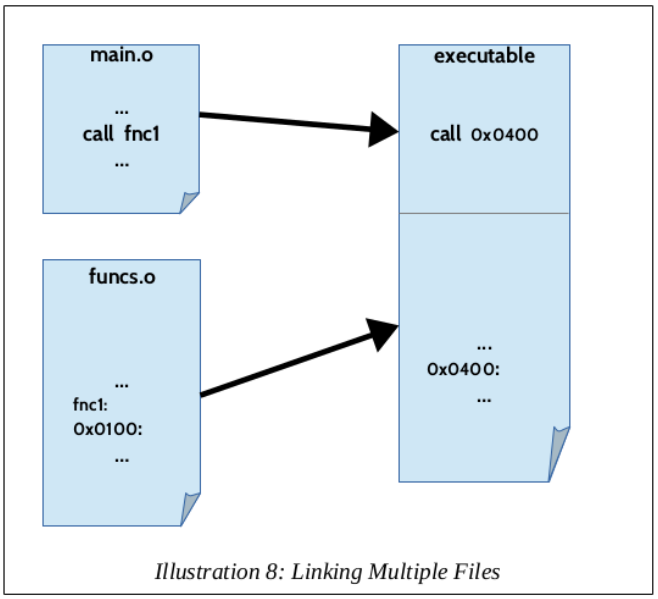

Illustration 8: Linking Multiple Files

Aqui, a função ***fnc1*** é externa ao arquivo de objeto ***main.o*** e é marcada com um R.
A função real ***fnc1*** está no arquivo ***funcs.o***, que inicia seu endereço relativo de 0x0 (na *section text*), pois não conhece o código principal. Quando os arquivos de objeto são combinados, o endereço relativo original de ***fnc1*** (mostrado como *0x0100* :) é alterado para seu endereço final no arquivo executável (mostrado como *0x0400 :*). O *linkador* deve inserir esse endereço final na instrução de chamada principal (mostrada como chamada *0x0400* :) para concluir o processo de *linkagem* (ligação) e garantir que a chamada de função funcione corretamente.
Isso ocorrerá com todos os endereços realocáveis para código e dados.