C para Drivers
Apostila da disciplina: INE5450

Curso de Ciências da Computação
Prof. Dr. João Dovicchi[1]

Florianópolis - 2013

[1]http://www.inf.ufsc.br/~dovicchi --- dovicchi@inf.ufsc.br

Conteúdo

Capítulo 1

Introdução

Porque um curso de C para programação de drivers de dispositivo? Bem, primeiro porque os programas que controlam dispositivos são escritos, preferencialmente, na mesma linguagem do kernel, ou seja, em C. Segundo porque é muito divertido, ou seria primeiro? Depois, porque é necessário entender algumas particularidades da linguagem C que facilitam muito a programação de interfaces com o hardware. Além disso, podemos dar mais razões para justificar a linguagem C:

- C é uma linguagem robusta. Robusta em diversos sentidos, não apenas na sua estrutura como linguagem, como também pela qualidade dos programas gerados pelo compilador C;
- C tem um conjunto de funções internas muito bem desenvolvidas e que ajudam na construção de programas e bibliotecas;
- C fornece uma boa base para programação de baixo nível com características de linguagem de alto nível;
- C gera programas altamente eficientes e rápidos com uma quantidade razoável de otimizações em tempo de compilação;
- C disponibiliza uma grande variedade de tipos de dados e operadores;
- C tem compiladores para quase todas as plataformas de sistemas operacionais e arquiteturas de processadores e, por isso é uma das linguagens mais portáveis.
- C é POSIX (*Portable Operating System Interface*) ou norma ISO/IEC 9945, cumprindo as especificações de portabilidade, escalabilidade e confiabilidade.

É importante ressaltar que a linguagem C foi feita para a programação do sistema operacional Unix e por isso é uma linguagem apropriada para construção de sistemas operacionais, compiladores, *software* aplicativo e outros tipos de *softwares*. Além disso se presta bem para a programação de bibliotecas de funções científicas, algébricas e integra-se facilmente com o Assembly e com o FORTRAN.

Neste curso, vamos abordar a linguagem C como ferramenta para desenvolvimento de drivers ou módulos que possam ser inseridos no kernel do Sistema Operacional Linux. Muito bem, eu disse a linguagem C? Bem, temos que considerar outros aspectos do kernel em relação ao hardware, principalmente quando precisamos de um controle direto e então, teremos que mostrar um pouco de Assembly, por conseguinte, mostrar como integrar códigos em Assembly e em C (linkedição).

Eu falei em diversão, não foi? É que nos dias de hoje, alguns hardwares interessantes estão sendo desenvolvidos, tipo: Raspberry Pi, Arduino, XBee/ZigBee, etc. portanto, vamos precisar de muita coragem para terminar as aulas no horário.

Mas vamos começar logo com isso, mas antes precisamos de C. C é uma linguagem clássica de programação, do tipo procedimental, estruturada e com uma grande facilidade para manipular endereços de memória e registros de processador. Além disso, a linguagem C permite a integração com um código mais de baixo nível como a própria linguagem de máquina. Nesta introdução vamos relembrar alguns conceitos. Primeiro um pouco de quando e onde surgiu a linguagem e depois umas "pinceladas"no assunto arquitetura de processadores, endereços e endereçamento e, ainda, tipos de dados e armazenamento.

1.1 Uma linguagem limpa

Em 12/10/11 ou `0xCAB` morreu Dennis Ritchie [1] exatamente 7 dias depois do Jobs da Apple e, por isso, sua morte nem foi muito comentada. Entretanto, este nome é um dos maiores nomes da ciência da computação que, junto com Brian Kernighan criou a linguagem C [2, 3].

A idéia era criar uma linguagem limpa para implementar o sistema operacional Unix e que pudesse lidar tanto com complexidades algorítmicas de alto nível como com aspectos de baixo nível como endereçamento de memória, endereços de hardware, interrupções e estruturas de dados tipados. Além disso, a linguagem deveria ter uma interface com a própria linguagem de máquina e instruções do processador. Como diz o próprio Dennis Ritchie em um histórico

artigo:

> "The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system. Derived from the typeless language BCPL, it evolved a type structure; created on a tiny machine as a tool to improve a meager programming environment, it has become one of the dominant languages of today." [4, 5]

Neste aspecto, a linguagem C cumpre o papel de linguagem de nível médio, possibilitando a interface entre o nível mais alto de algorítmos complexos e o nível mais baixo do Assembly. Assim, o C passou a ser uma linguagem de referência e foi além da própria linguagem em si, dando origem às linguagens C++ e Java, além muitas outras que seguiram o mesmo modelo. Neste ponto, sugiro a leitura dos dois prefácios do livro do Dennis Ritchie e do Brian Kernighan [2]

1.2 O processador

Como dispositivo principal, a compreensão de sua arquitetura é fundamental para podermos entender como o sistema operacional funciona e controla os outros dispositivos.

1.2.1 Arquitetura CISC

A arquitetura CISC (*Complex Instruction Set Computer*) é baseada em um conjunto de registros com funções específicas para armazenar dados ou endereços (o que, genericamente, chamamos de x86). Neste tipo de arquitetura, uma única instrução executa várias operações tais como carregamento de endereços, apontadores de instrução, e manipulação de dados na memória e, algumas vezes com multiplas operações e cálculos de endereços em uma só instrução de programa [6, 7].

Os 4 registradores `A`, `B`, `C` e `D` são registradores de uso geral e são denominados de Acumulador, Base, Contador e Dados, respetivamente. Estes registradores são usados com 32/64 bits ou com 8 bits (para compatibilidade). As instruções usam registradores específicos para operação de dados e endereços. Um exemplo é a instrução `mul` do set do x86 que usa o par `DX:AX` para armazenar o resultado da multiplicação de `AX` $\times$ `CX`, ou seja, depois da operação, o resultado é armazenado no par de registradores `DX:AX`. Por exemplo:

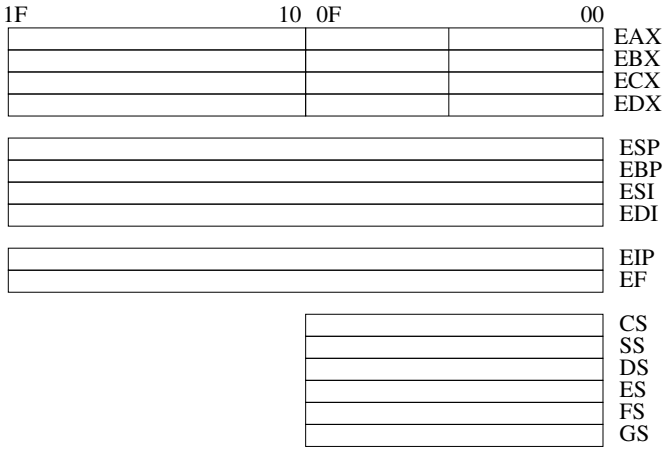


Figura 1.1: Esquema dos registradores x86 (sem as extensões)

```
section .text
global  _start

_start:
  mov  cx, 8          ; CX (contador) recebe o valor 8.
  mov  ax, 1          ; AX (acumulador) recebe o valor 1.
  mov  dx, 0          ; DX (data) recebe zero porque vai compor com AX.
  mul  cx             ; O par DX:AX recebe o valor de AX * CX
  ... etc.
```

Outros registradores têm utilização bem específica, por exemplo, o `SP` (*Stack Pointer*) é o apontador da pilha; o `BP` (*Base Pointer*) é o endereço base da pilha; o `SI` é o indexador da pilha; e o `DI` é o indexador de dados. O registrador `IP` (*Instruction Pointer*) serve para armazenar o endereço da instrução do programa e, por isto, algumas vezes é chamado de `PC` (*Program Counter*). Além disso, a arquitetura CISC possui um registrador de *flags* e vários registradores de segmento de dados e segmento de códigos. Estes registradores foram, posteriormente expandidos para 32 e 64 bits.

A AMD deu início a uma reformulação da arquitetura x86 e implementou novos registradores *Streaming SIMD Extension* (SSE e SSE2). Os 8 registradores originais do x86 foram ampliados de 32 para 64 bits e foram criados mais 8 registradores de 64 bits, ampliando em 4 vezes o espaço de armazenamento. Além disso, criaram mais 8 registradores para instruções SSE ou SSE2 que depois foram ampliados para 16 nos processadores de 64 bits. Estes registradores não foram expandidos pois o SSE utiliza registradores de 128 bits. [8]

1.2.2 Arquitetura RISC

A arquitetura RISC (*Reduced Instruction Set Computer*) tem um conjunto de registradores que possibilitam seu uso como registros específicos ou como registros genéricos. [9]

As vantagens da arquitetura RISC está em um conjunto de instruções pequeno e altamente optimizado. Cada instrução pode ser executada em um único ciclo de *clock*. Todas elas têm o mesmo tamanho e não têm microcódigo (executam mais rapidamente). Além disso, o compilador gera um código mais eficiente.

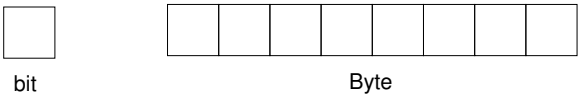
Na arquitetura RISC, o processador ARM (*Advanced Risc Machine*) é o representante do futuro nos sistemas móveis e já pode ser encontrado em mais de 90% de todos os *gadgets* móveis, embarcados e microcontrolados. [10]

O processador ARM possui 16 registradores (veja figura 7.1) que podem estar em diversos estados (*modes*):

- User: Estado de execução normal;
- FIQ: Fast Interrupt (transf. rápida de dados);
- IRQ: Usado para manipulação de interrupções;
- Supervisor: Modo protegido para o sistema operacional;
- Abort: Implementa mem. virtual / mem protection;
- Undefined: Emulação de hardware de coprocessamento por software;
- System: Roda *tasks* do SO em modo privilegiado.

1.3 A memória

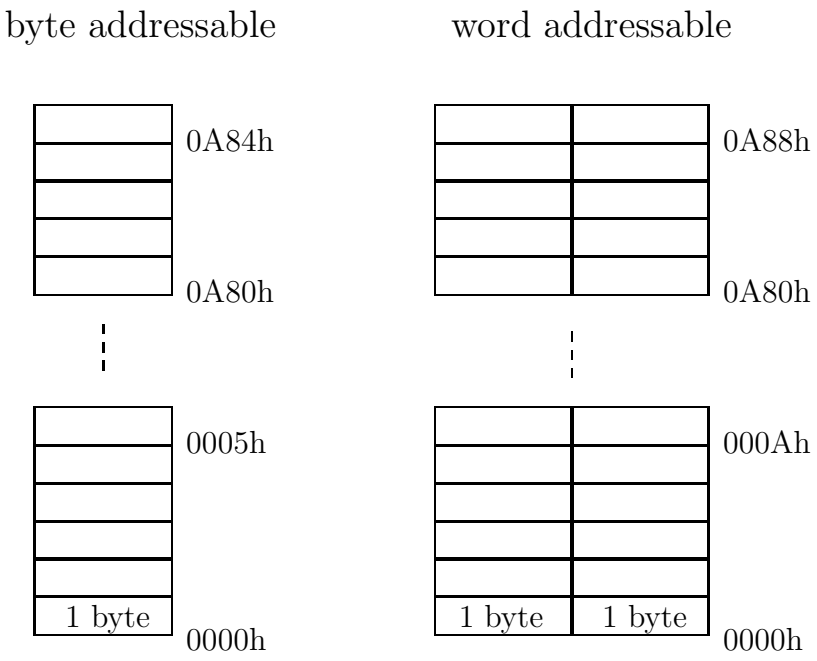
Desde os "imemoriáveis" tempos da memória ferromagnética [11] as posições de dígitos binários (*bits) são organizadas em grupos de oito bits (*byte) e referenciados por um endereço de 16, 32 ou 64 bits que conhecemos como barramento.



1.3.1 Organização da memória e endereçamento

A memória principal pode ser considerada como uma matriz de bits, onde cada linha representa uma posição da memória. O número de bits de cada posição é, em geral de 8, 16, 32 bits etc. Tudo depende da arquitetura do *hardware*. Cada posição da memória pode ser considerada como lida ou escrita por vez (por ciclo) e cada posição tem um endereço. [12, 13]

As memórias são geralmente manipuladas pelo endereçamento que pode ser *multi-byte* em tamanho. Por exemplo, pode-se endereçar por uma `word` (16 bits) chamada de *word addressable* ou simplesmente endereçar por um *byte* (8 bits) chamada de *byte-addressable*. Cada endereço em uma memória contém 8 bits e é manipulado dependendo do endereçamento. Ele será contado de um em um no caso do endereçamento por *byte* ou conterá apenas endereços pares (de dois em dois) no caso do endereçamento por *word*.



Os processadores ARM têm a capacidade de endereçamento de 26 bits ou seja, pode endereçar até 2^{26} posições ou blocos de 64 Megabytes. Embora se possa transferir bytes individuais entre memória e processador, o ARM tem um endereçamento do tipo *Word Addressable* de 32 bits real. Isto significa que todos os operandos de instruções são do tipo 32-bit-word-size e as instruções endereçam sempre múltiplos de 4 [14].

O diagrama da figura 1.2 mostra o esquema de endereçamento do processador ARM.

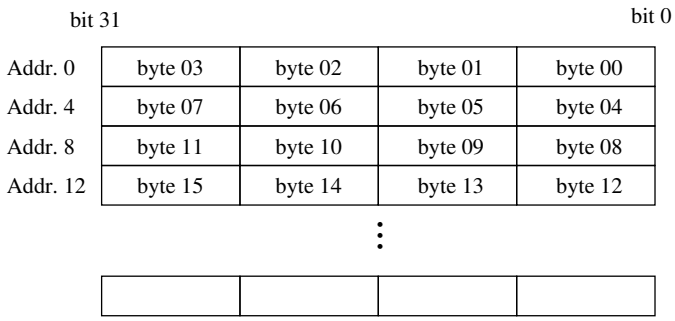
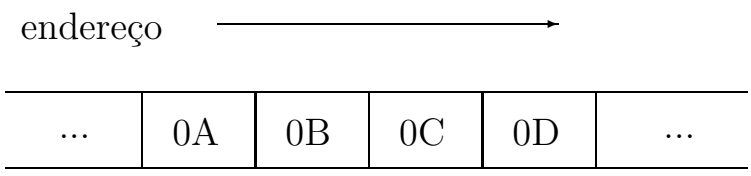


Figura 1.2: Endereçamento da arquitetura ARM

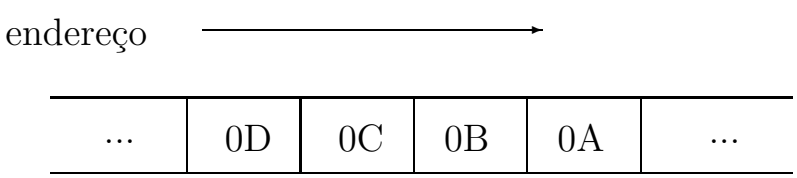
1.3.2 Endianness

O armazenamento de bits na memória pode ser feito de duas formas: em ordem direta ou em ordem inversa. O armazenamento em ordem direta é denominado **big endian** e o armazenhamento em ordem inversa é chamado de **little endian**. Estes modelos são denominados de ordem de bytes (*byte order*) ou *endianness* da arquitetura.

A ordem de bytes pode ser arranjada em grupos de 8, 16 ou 32 bits e resultam em uma ordem de armazenamento de bits nos endereços de memória de forma diferente. Por exemplo, o armazenamento de uma palavra de 32 bits na ordem de 8 bits, digamos, 0x0A0B0C0D em hexadecimal no formato big endian fica como:



e no formato little endian fica como:



No caso da ordem de bytes de 16 bits resulta em 0A0B0C0D em big endian e, no caso de little endian ficaria 0C0D seguido de 0A0B [15].

Capítulo 2

Tipos e Operadores

A linguagem C é bastante econômica com relação à tipagem de dados. Geralmente, um caractere pode ser representado por 8 bits um inteiro por 2 ou 4 bytes e um número de ponto flutuante pode ser representado por 32 ou 64 bits (IEEE 754) [16]

2.1 Tipos

Os tipos em C são indicados pelas palavras `char` para caracteres, `int` para inteiros, `float` para números de ponto flutuante de precisão simples ou representação IEEE 754 de 32 bits e `double` para números de ponto flutuante de precisão dupla ou representação IEEE 754 de 64 bits.

Os tipos podem ser qualificados, por ex.: `short int` ou `long int`

`int`	$\longrightarrow$	16 ou 32 bits
`short int`	$\longrightarrow$	16 bits
`long int`	$\longrightarrow$	32 bits

Qualificadores *signed* ou *unsigned* podem ser aplicados a `char` ou `int` e o qualificador `long double` especifica um número de ponto flutuante de precisão extendida (80 ou 128 bits). Os cabeçalhos `<limits.h>` e `<float.h>` contêm a definição para estes tamanhos[1].

[1]No Linux o `ieee754.h` é equivalente ao `/usr/include/float.h`.

Além das definições de variáveis, a linguagem C permite a definição de valores constantes e grupos de constantes enumeradas. Estes valores podem ser definidos diretamente ou podem ser definidos condicionalmente. Por exemplo:

```
/* define constantes diretamente */

#define MIN 0
#define MAX 255
#define NOVA_LINHA '\xb'

const double e = 2.7182818284590;

/* define constantes condicionalmente */

#ifndef O_RDONLY
#define O_RDONLY 0
#endif

#ifndef PI
  const double PI = 3.14159265359;
#endif

/* define grupos de constantes enumeradas */

enum resp (NAO,SIM);
enum semana (DOM=0,SEG,TER,QUA,QUI,SEX,SAB);
```

As enumerações (`enum`) são constantes numéricas associadas aos nomes do grupo enumerado. O qualificador `const` pode ser aplicado a qualquer variável, que assume um valor que não pode ser mudado durante o algoritmo.

2.2 Operadores

A linguagem C tem operadores aritméticos, relacionais, lógicos e bit-a-bit. Existe, também, operadores algébricos (parênteses, chaves etc.), operadores de atribuição, operadores de composição de membros e operadores de conversão de tipos.

2.2.1 Operadores aritméticos

Os operadores artméticos soma (`+`), subtração (`-`), divisão (`/`), multiplicação (`*`) e módulo (`%`) são associativos da esquerda para a direita, sendo que os operadores `* / %` têm igual precedência entre si e maior precedência que os operadores `+` e `-`.

Os operadores unários (`+ -`) de atribuição de sinal têm precedência maior que os operadores aritméticos e menor que os operadores de incremento e decremento.

Os operadores de incremento (`++`) e de decremento (`--`) existem em duas formas. A forma pré e pós-fixada. A diferença é que, em uma expressão, a forma pré-fixada é incrementada antes de ser avaliada e a forma pós-fixada é incrementada depois. Por exemplo:

```
x = 10;
y = x++; /* y recebe o valor 10 e x passa a valer 11 */

...

x = 10;
y = ++x; /* x passa a valer 11 antes da atribuição para y */
```

Estes operadores têm precedência maior que os demais, sendo que a forma pos-fixada (p. ex. `++var`) tem precedência maior que a forma pre-fixada (p. ex. `var++`).

2.2.2 Operadores relacionais

Os operadores relacionais são os que avaliam resultado de comparações de valores ou expressões, retornando os valores "verdadeiro" (1)[2] e "falso" (0).

Os operadores `<, <=, >, >=` possuem a mesma precedência e têm precedência maior que os operadores `==` e `!=`

2.2.3 Operadores lógicos

A linguagem C possui 3 operadores lógicos. O `&&` para `AND`, o `||` para `OR` e o `!` para `NOT`. O operador `!` tem a maior precedência, seguido do `&&` e depois pelo `||`.

Enquanto o operador lógico `NOT` tem precedência maior que os operadores artméticos e relacionais, os operadores `AND` e `OR` têm precedência menor que aqueles.

2.2.4 Operadores *bitwise*

O uso de operadores bit-a-bit não é uma coisa corriqueira na programação em geral. Na maioria das vezes tem-se que pensar em argumentos inteiros (`char` e `int`) e argumentos de ponto flutuante (`float` e `double`). No entanto,

[2] A linguagem C considera qualquer valor diferende de zero como "verdadeiro"

x	**y**	**x&&y**	**x‖y**	**!x**
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Tabela 2.1: Tabela verdade dos operadores lógicos em C

algoritmos de criptografia, de compressão, de tratamento de sinais (filtros) etc. podem demandar o uso de operações com bits.

Outra utilidade da operação com bits são alguns "truques" que se pode fazer para aumentar a velocidade de processamento, o que não costuma ser recomendado por diversas razões. Em algumas situações, as operações podem apresentar resultados falhos dependendo da especificação dos operandos (p. ex. `signed` ou `unsigned int`).

Finalmente as operações com bits podem ser usadas para ler estados de hardware, portas, registradores de *flags* etc. que é a parte que nos interessa.

Operador	**Significado**	**Nota**
&	AND	
|	OR	
^	XOR	ou exclusivo
~	NOT	complemento de 1
$\gg$	move bits right	Div. por potência de 2
$\ll$	move bits left	Mul. por potência de 2

Tabela 2.2: Operadores bit-a-bit

Os operadores bit-a-bit (veja tabela 2.2.4) comparam os bits individuais de dois conjuntos de bytes. Isto faz com que estas operações sejam frequentemente usadas em drivers de dispositivos.

O operador `&` pode ser usado para desligar um bit específico, ou seja, a operação `AND` de um conjunto de bits com qualquer bit zero faz com que o bit correspondente seja desligado. Por exemplo:

```
#include <stdlib.h>
#include <stdio.h>

#define FLAG 0x10AB // 0001000010101011
#define MASK 0xFFFE // 1111111111111110

int main(void){
  printf("\n%X & %X = %X\n", FLAG, MASK, (FLAG & MASK));
  return 0;

}
```

O operador |, funciona ao contrário do operador `&` e pode ser usado para ligar um bit, ou seja, a operação `OR` de um conjunto de bits com qualquer bit 1 faz com que o bit correspondente seja ligado. Por exemplo:

```
#include <stdlib.h>
#include <stdio.h>

#define FLAG 0x8000 // 1000000000000000
#define MASK 0x8005 // 1000000000000101

int main(void){
  printf("\n%X | %X = %X\n", FLAG, MASK, (FLAG | MASK));
  return 0;

}
```

O operador ^ ativa um bit se, e somente se, um dos bits operados forem diferentes. O operador `OR` entre um conjunto de bits e um conjunto igual de uns inverte todos os bits do operando, funcionando como se fosse um operador `NOT`. Por exemplo:

```
#include <stdlib.h>
#include <stdio.h>

int main(){
  unsigned int Flags = 0x5555; /* dec = 21845 / bin = 0101010101010101 */
  unsigned int Mask  = 0xFFFF;
  unsigned int Bits  = 0x0000;

  Bits = Flags ^ Mask; /* XOR operator */

  printf("\nFlags = %X\nBits  = %X \n\n", Flags, Bits);
  return 0;

}
```

Embora as aplicações destes operadores (`AND, OR` e `XOR`) sejam muito úteis para testar estados de portas e hardware em programas que requeiram estas operações em baixo nível (como drivers, por exemplo), elas deve ser evitadas em sentenças condicionais como os operadores lógicos e relacionais. Por exemplo:

```
#include <stdlib.h>
#include <stdio.h>

int main(){
  unsigned int var1 =7;
  unsigned int var2 =8;

  printf("var1 && var2 = %d\n", var1 && var2); /* 1 */
  printf("var1 & var2  = %d\n", var1 & var2);  /* 0 */
  return 0;

}
```

É bom lembrar que os operadores lógicos e relacionais sempre retornam resultados booleanos 1 ou 0 (`TRUE` ou `FALSE`) enquanto que os operadores *bitwise* retornam um valor de acordo com a operação realizada.

Os operadores `<<` e `>>`, movem os bits para a esquerda ou para a direita, respectivamente. A forma de uso do operador é:

```
operando >> num. de vezes
```

Conforme os bits são movidos para um lado, zeros são adicionados pelo outro lado e, portanto, embora algumas vezes se denomina a operação de "rotate", não ocorre uma rotação. Os bits que saem por um lado não retornam pelo outro.

Cada deslocamento para a esquerda corresponde à multiplicação por 2 e o inverso, ou seja, o deslocamento para a direita, corresponde à divisão por 2. Claro que se tem que respeitar o tipo do operando. Com exemplo, podemos mostrar:

```
#include <stdlib.h>
#include <stdio.h>

int main(){
  int num=5, pot_2=4;

  printf("5*(2^4) = %d\n", num<<pot_2);
  return 0;

}
```

Para referência, uma tabela de precedência de operadores pode ser encontrada em:

`http://www.difranco.net/compsci/C_Operator_Precedence_Table.htm`

Capítulo 3

Array e ponteiros

A linguagem C tem algumas estruturas de dados que facilitam muito a disposição deles e a sua organização na memória. Uma delas é o agrupamento de variáveis do mesmo tipo em "arranjos" ou *array* que as reúne em uma fila indexada de mesmo tipo.

3.1 Arrays

Arrays são coleções de variáveis do mesmo tipo ordenadas sequenciamente, tanto na abstração do algorítmo como na memória física do computador. Um *array* é declarado com o nome da variável e o valor do tamanho do arranjo. Por exemplo:

```
/* definicao de um Array */
...
int var_inteira[20] /* define 20 posicoes consecutivas de memoria */
                    /* para alocar um inteiro em cada uma delas   */
char var_frase[30]  /* define 30 posicoes consecutivas de memoria */
                    /* para alocar caracteres de uma frase.       */

var_frase = "Um texto de trinta caracteres"
                    /* Atribuindo 30 carcteres a variavel 29 char */
                    /* + 1 NULL (como fim do array de char).      */
```

No exemplo acima, cada caractere da variável `var_frase` fica disposto, na memória, em uma seqüência de *bytes* consecutivos a partir do endereço do primeiro caractere, o "U" (veja fig. 3.1)

A definição de um array é feita acrescentando-se um índice entre colchetes a uma variável, por exemplo: `int x[10]`. Considerando que uma variável é o "nome" de um endereço de memória, quando se indexa um tipo de variável

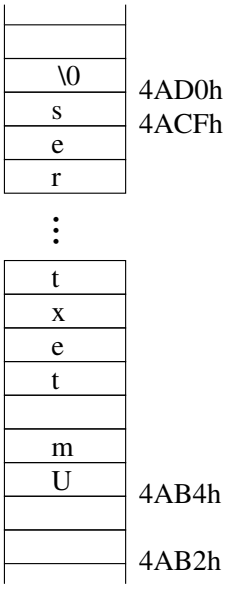


Figura 3.1: Disposição de uma *string* em posições da memória

fica alocada, para ela, um espaço de memória a partir do endereço do primeiro elemento (p. ex. `x[0]`) com 32 bits (ou do tamanho de um `int`[1] na respectiva arquitetura) seguido de todos os outros (veja fig. 3.2).

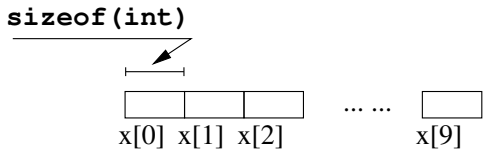


Figura 3.2: Disposição de um vetor (array) na memória

Os arrays podem representar quantidades vetoriais (unidimensionais), matriciais (bidimensionais) ou multidimensionais. Entretanto, estas representações são feitas como arrays de arrays e são organizadas, formalmente, como um conjunto linear de elementos na memória. Por exemplo:

```
/* definicao de um Arrays dimensionais */
...
float a[2][3]      /* define 6 posicoes 'float' de memoria     */
                   /* para os elementos de uma matriz.         */
double m[2][3][2] /* define 12 posicoes 'double' de memoria   */
                   /* para elem. de um array multidimensional. */
```

[1]Equivalente a `sizeof(int)`.

A imagem da figura 3.3 representa os dois arrays multidimensionais do exemplo acima.

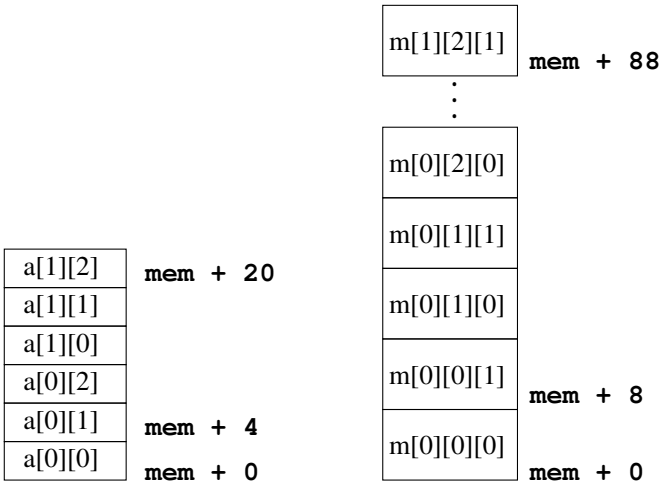


Figura 3.3: Disposição de vetores multidimensionais na memória

3.1.1 Usando array como parâmetro de funções

Os elementos de um *array* podem ser passados como parâmetro de uma função, neste caso, um outro *array* deve receber uma cópia dos elementos daquele da chamada da função. Por exemplo:

```
#include <stdio.h>
#include <stdlib.h>

#define NUMS 10

float med(float[],int n);

int main(int argc, char *argv[]){
  float num[NUMS]; /*vetor de numeros */
  float media;
  int i;

  for(i=0; i<NUMS; i++){
    scanf("%f", &num[i]);
  }
  media = med(num,NUMS);
  printf("Media = %f\n", media);
  return 0;
}

float med(float a[], int n){
  int i;
  float result, soma=0.0;

  for(i=0; i<n; i++){
    soma += a[i];
  }
  result = soma/(float) n;
  return result;
}
```

3.2 Apontadores (ponteiros)

Arrays e apontadores são conceitos muito interligados em C. O conceito de apontador facilita muito a manipulação de dados diretamente em seus endereços da memória. Os apontadores, em si, são variáveis que podem armazenar endereços de um determinado tipo e indicam o endereço de determinada variável[2]. Da mesma forma, um apontador para um array, aponta para o endereço do primeiro elemento do array.

Um apontador é declarado como o tipo que ele aponta, um asterisco e o nome do apontador (`<tipo> *<nome>`), por exemplo, `int *p;` cria uma variável "p" que é um apontador para um endereço de um inteiro na memória. Um apontador é associado a uma variável do mesmo tipo por um &, por exemplo, `p=&x;` associa o endereço de `x` à variável `p`. Suponhamos uma variável e um ponteiro do tipo da variável:

[2]O compilador resolve o problema de *endianness* para cada arquitetura.

```
...
int x;  /* x eh uma variavel do tipo int */
int *p; /* p eh uma variavel que pode conter o endereco de um int */

p = &x; /* p passa a conter o endereco da variavel x */
        /* equivale a declarar p = x; porque p eh um apontador.*/
```

A imagem da figura 3.4 representa como um apontador "p" referencia uma variável "x" como no código acima.

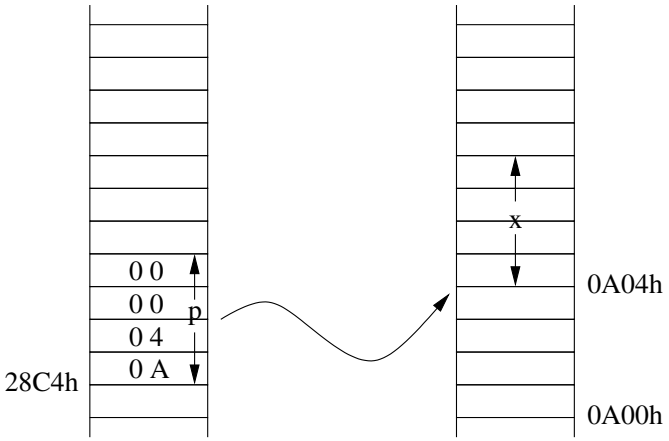


Figura 3.4: Um apontador para o endereço de uma variável tipo `int`

Existem dois operadores básicos para se lidar com operadores em C. O operador "&" e o operador "*" que representam o endereço de uma variável e o objeto apontado pela variável, respectivamente. Por exemplo:

```
...
    int x=1, y=2, v[10];
    int *p;     // define uma p como apontador p/ int
    ...
    p = &x;     // p aponta para o endereco de x
    ...
    y = *p;     // y recebe o valor de onde p aponta (y=x)
    ...
    *p = 0;     // atribui zero ao objeto apontado por p (x=0)
    ...
    p = &v[0]; // p aponta agora para o end. v[0]
   ... etc.
```

As operações com apontadores são contextuais. Por exemplo, se `*intp` aponta para um inteiro (`intp = &x;`), qualquer contexto onde se possa usar `x`, pode-se usar `*intp`. Assim:

```
...
int x, *intp;
...
intp = &x; /* intp aponta para o ender. do interio x */
...
*intp = *intp + 10; /* equivale a x = x + 10; */
...
```

Um apontador pode ser incrementado ou decrementado para apontar para outro endereço relativo ao seu tipo e pode ser operado como inteiro. A aritmética com apontadores é simples. pode-se incrementar o apontador de uma unidade e ele estará apontado para o próximo endereço relativo ao seu tipo. Por exemplo se um apontador para float (e.g. `float *p`;) apontar para uma variável (e.g. `float x;`), a operação `p+1` aponta para o próximo endereço `float`, ou seja, 4 bytes depois.

Deve-se tomar cuidado com as operações feitas durante a atribuição de valores. Por exemplo:

```
...
float x[10], *p;
p = &x;
*p = 10.0;   /* atribui 10.0 a x[0] */
*p++ = 0.5;  /* NOTE QUE: atribui 0.5 a x[0] e incrementa o p */
*p = 1.5;    /* x[1] = 1.5 */
*(++p)=2.5;  /* x[2] = 2.5 */
p = &x[3];   /* p aponta para x[3] */
*p = 3.5;    /* x[3] = 3.5 */
p = x+4;     /* p aponta para x[0+4] */
*p = 4.5;    /* x[4] = 4.5 */
p = x; /* p aponta para x[0] novamente */
*(p+5) = 5.5 /* x[5] = 5.5 */
...
```

No exemplo acima, inicialmente `p` aponta para `x[0]` que recebe o valor `10.0`. Na atribuição do segundo valor, `p` somente será incrementado depois da atribuição do valor, ou seja, primeiro atribui o valor `0.5` para `x[0]` e depois incrementa o apontador que então passa a apontar para `x[1]`.

3.2.1 Apontadores como argumentos de funções

Os apontadores podem ser passados como argumentos de funções. Nestes casos, são passados os endereços de variáveis para a função que tem, então, acesso às variáveis da chamada da função.

Quando se passa um valor para uma função, a variável da função contem uma cópia da variável usada na chamada da função. Assim, qualquer alteração da variável dentro do escopo da função não interfere na variável que é passada

na chamada da função. Este tipo de chamada é dita "chamada por valor". Por exemplo:

```
...
int main(){
    int a, b;
    ...
    troca(a,b); /* se for para swap(a,b)
    ...
}

void troca(int x, int y){    /* errado !!! */
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

No exemplo acima, se a intenção do programa é trocar os valores de `a` e `b`, a função `troca` não conseguirá realizá-la. A troca das variáveis `x` e `y` na função altera apenas a cópia das variáveis e não as variáveis em si.

Se, no entanto, a função receber os endereços de `a` e `b` em apontadores apropriados então a troca pode ser feita diretamente no endereço da variável que chamou a função. Este tipo de chamada é dita "chamada por referência". Por exemplo:

```
...
int main(){
    int a, b;
    ...
    troca(&a,&b); /* chama 'troca' c/ o ender. de 'a' e 'b' */
    ...
}

void troca(int *px, int *py){    /* px aponta para 'a' e py para 'b' */
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;  /* troca o conteúdo de px e de py */
}
```

Neste caso, agora, os apontadores permitem que a função acesse diretamente as variáveis quando a função é chamada, por meio de seus respectivos endereços de memória. As alterações dentro da função por meio dos apontadores opera, diretamente, nos endereços da memória onde estão armazenadas estas variáveis.

3.2.2 Ponteiros para funções

A programação de drivers usa muito o conceito de apontadores para funções que é uma maneira de chamar a função por referência. Uma vez que uma função tem uma posição física na memória, a linguagem C permite a criação de ponteiros para funções.

Embora isto seja um pouco confuso, é uma característica poderosa de C porque o endereço de uma função é o ponto de entrada da função e pode ser usado para chamá-la.

Quando uma função é compilada em C, o código fonte é convertido em um código objeto e um ponto de entrada para a função é criado. Quando a função é chamada, na execução do programa, é feito um *"call"* na linguagem de máquina para o ponto de entrada da função. Por exemplo:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void chk(char *a, char *b, int (*cmp)());

int main(){
  char c1[20], c2[20];
  int (*p)(); /* 'p' apontador p/ uma funcao */
  p = strcmp; /* 'p' aponta p/ a funcao strcmp */

  gets(c1);
  gets(c2);
  chk(c1, c2, p);
  exit(0);
}

void chk(char *a, char *b, int(*cmp)()){
  if(!(*cmp)(a,b)){
    printf("strings iguais\n");
  }else{
    printf("strings diferentes\n");
  }
}
```

Note que ao chamar a função `chk` são passados dois ponteiros para caracteres e um ponteiro `p`. Dentro da função `(*cmp)(a,b)` chama a função `strcmp` descrita em `string.h`. Note que os parênteses são necessários para que o compilador entenda como um apontador de função. Note, ainda, que no exemplo acima pode-se usar diretamente a função, por exemplo:

```
chk(c1, c2, strcmp);
```

A declaração `int (*p)()` é usada apenas como exemplo de como declarar um ponteiro para uma função dentro de um bloco do programa.

Além de ponteiros para funções, pode-se criar um *array* de apontadores para funções. Um exemplo do uso de array de apontadores para funções é em programas onde a função a ser chamada depende de uma escolha, um menu por exemplo. O livro do Deitel (cap. 7 figura 7.28) [17] traz um exemplo muito bom:

```
/* Demonstrating an array of pointers to functions */
#include <stdio.h>

/* prototypes */
void function1( int a );
void function2( int b );
void function3( int c );

int main( void )
{
   /* initialize array of 3 pointers to functions that each take an
      int argument and return void */
   void (*f[ 3 ])( int ) = { function1, function2, function3 };

   int choice; /* variable to hold user's choice */

   printf( "Enter a number between 0 and 2, 3 to end: " );
   scanf( "%d", &choice );

   /* process user's choice */
   while ( choice >= 0 && choice < 3 ) {

      /* invoke function at location choice in array f and pass
         choice as an argument */
      (*f[ choice ])( choice );

      printf( "Enter a number between 0 and 2, 3 to end: ");
      scanf( "%d", &choice );
   } /* end while */

   printf( "Program execution completed.\n" );
   return 0; /* indicates successful termination */
} /* end main */

void function1( int a )
{
   printf( "You entered %d so function1 was called\n\n", a );
} /* end function1 */

void function2( int b )
{
   printf( "You entered %d so function2 was called\n\n", b );
} /* end function2 */

void function3( int c )
{
   printf( "You entered %d so function3 was called\n\n", c );
} /* end function3 */
```

Capítulo 4

Estruturas

Estruturas "polimórficas" que na linguagem C reúnem elementos de tipos diferentes em cada um dos componentes são chamadas de `struct` e `union`.

A vantagem destas estruturas é que elas se encontram agrupadas sequencialmente na memória e podem ser facilmente referenciadas por endereços. O referenciamento delas por seus endereços é chamado de apontamento e as variáveis que trabalham com estes endereços são denominadas apontadores ou ponteiros.

A programação de drivers de dispositivos, sejam internos do kernel ou como módulos carregados posteriormente, depende fundamentalmente do conhecimento destas estruturas para poder controlar a alocação de memória e eficiência do código no *kernel space*.

4.1 Struct

Uma estrutura (`struct`) em C é um conjunto de variáveis de tipos diferentes que se encontram relacionadas entre si. Por exemplo, uma agenda de endereços e telefones de pessoas, um cadastro de clientes, uma folha de pagamentos etc. podem conter variáveis de diversos tipos que podem ficar agrupadas em uma estrutura. Por exemplo, uma folha de pagamentos, pode conter variáveis como:

```
int id;
char nome[90];
char cargo[10];
float renda;
```

Estas variáveis podem estar relacionadas em um grupo, de tal forma que seus membros fiquem vinculados a um determinado indivíduo. Esta estrutura poderia ser declarada da seguinte maneira:

```
struct funcionario{
        int id;
  char nome[20];
  char sobrenome[20]
  char cargo[10];
  int idade;
  char sexo;
  float renda;
};
```

Uma vez declarada a estrutura, pode-se criar qualquer variável com os mesmos tipos de membros dela. Assim,

```
struct funcionario empregado[1000]
```

declara uma variável chamada `empregado`, que contem 1000 elementos e, cada um contem os membros da estrutura `funcionario`.

4.1.1 Acesso aos membros da estrutura

Cada membro da estrutura pode ser referenciado por um operador ponto (`"."`), por exemplo:

```
empregado[9].sexo = "M";
```

atribui o caractere `M` ao membro `sexo` da décima estrutura `empregado` que é do tipo `funcionario`.

Além do operador ponto, existe o operador seta (`->`) que pode acessar o membro da estrutura por referência ao seu endereço. Por exemplo, uma variável do tipo `estrutura.membro` pode ser acessada por um ponteiro do tipo `(*estruturaPtr).membro` que é equivalente a usar o operador seta: `estruturaPtr->membro`

4.1.2 Estruturas e funções

As estruturas podem ser passadas para funções e acessadas por valor ou por referência. No exemplo abaixo, existe uma estrutura cujos membros podem

ser acessados por meio de uma cópia do valor da estrutura passada para uma função, ou por meio da referência ao endereço dos membros da estrutura. Por exemplo:

```
#include <stdio.h>
#include <stdlib.h>

struct cartas {
        int indice;
        char naipe;
};
void imprime_por_valor (struct cartas qual);
void imprime_por_refer (struct cartas *p);

int main(){
        struct cartas rei_paus = {13,'p'};
        struct cartas dama_copas = {12,'c'};
        imprime_por_valor (rei_paus);
        imprime_por_refer (&dama_copas);
        return 0;
}

void imprime_por_valor (struct cartas qual){
        printf("%i de %c\n", qual.indice, qual.naipe);
}

void imprime_por_refer (struct cartas *p){
        printf("%i de %c\n", p->indice, p->naipe);
}
```

A primeira função (`imprime_por_valor`) usa uma estrutura chamada "qual" para receber cópias dos membros de uma estrutura do tipo "cartas" e `qual.indice` ou `qual.naipe` são referências a estes valores na chamada da função.

A segunda função (`imprime_por_refer`) recebe um apontador para uma estrutura do tipo cartas ou `*p`. Neste caso, `p->indice` ou `p->naipe` são apontadores para os endereços dos membros da estrutura da chamada da função.

Assim, as estruturas podem ser passadas como parâmetro de funções:

```
...
struct{
   int i;
   double d;
}nums;

void funcao(struct nums ns){
   ns.i = ns.i + 10;
   ns.d = ns.d + 0.5;
}

int main(){
   struct nums k;
   k.i = 50;
   k.d = 0.3;
   funcao(k);

... // qual o valor de k.i e k.d?
```

Neste caso, os valores passados são apenas cópias das variáveis de onde a função foi chamada. Para alterar os valores, os dados devem ser passados como apontadores:

```
#include <stdio.h>

struct{int i; double d;} nums;

void funcao(struct nums *);

int main(){
   struct nums k;
   k.i = 50;
   k.d = 0.3;
   funcao(&k);
   printf("%d, %f\n", k.i, k.d);
   return 0;
}

void funcao(struct nums *pns){
   pns->i = pns->i + 10; // pns->i equivale a (*pns).i
   pns->d = pns->d + 0.5;
}
```

O uso da palavra `typedef` pode definir uma estrutura como um sinônimo de tipo de dado. Por exemplo:

```
typedef struct{int i; double d;} nums;

void funcao(nums *); // prototipo da funcao usando
                     // o tipo nums

int main(){
   nums k; // def. da var. do tipo nums
   k.i = 50;
   k.d = 0.3;
   funcao(&k);
   printf("%d, %f\n", k.i, k.d);
   return 0;
}
```

Obs.: compare este código com o anterior.

O uso de `typedef` permite que funções possam retornar valores como estruturas, por exemplo:

```
...
#include <math.h>

typedef struct{
   double r,i;
}complexo;

complexo compl_add(complexo a, complexo b){
   complexo c;
   c.r = a.r + b.r;
   c.i = a.i + b.i;
   return c;
}
```

4.2 Union

A estrutura de dados do tipo `union` é semelhante à do tipo `struct`. A diferença é que cada membro de um `union` ocupa a mesma posição na memória. Assim, o endereço da estrutura na memória fica reservado para ser ocupado pelo maior membro, possibilitando a criação de uma variável do tipo `union` que pode conter objetos de diferentes tipos e tamanhos mas não ao mesmo tempo. Por exemplo:

```
#include <stdlib.h>
#include <stdio.h>

union Num{
        int x;
        float y;
        double z;
};

int main(){
        union Num valor;

        valor.x = 10;
        printf("\tint: %i\n\tfloat: %f\n\tdouble: %f\n\n",\
      valor.x, valor.y, valor.z);

        valor.y = 10.25;
        printf("\tint: %i\n\tfloat: %f\n\tdouble: %f\n\n",\
      valor.x, valor.y, valor.z);

        valor.z = 10.12345678912345678912345678912345;
        printf("\tint: %i\n\tfloat: %f\n\tdouble: %f\n\n",\
      valor.x, valor.y, valor.z);

  exit(0);
}
```

Se formos comparar uma `struct` e uma `union`, veremos que, na memória, uma union ocupa bem menos espaço porque o seu tamanho é o do maior membro, enquanto `struct` tem o tamanho somado do tipo de todos os seus membros (veja figuras 4.1 e 4.2).

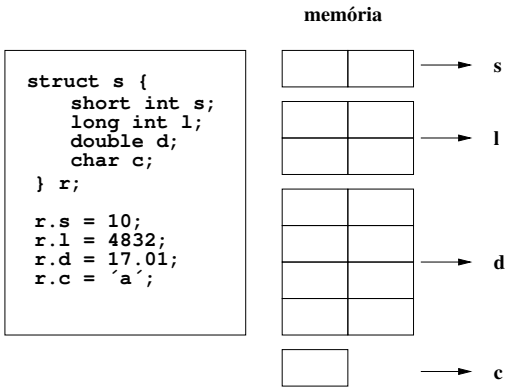


Figura 4.1: Alocação de uma estrutura na memória

Embora uma `union` seja mais econômica do que uma `struct` cabe ao programdor garantir a a lógica do algoritmo.

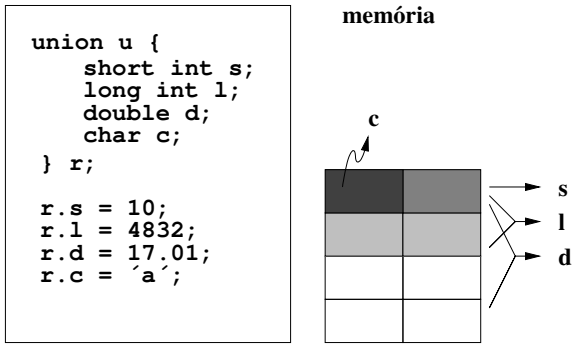


Figura 4.2: Alocação de uma união na memória

4.3 Campos de bits

A linguagem C ainda possui um método que permite o acesso a um bit dentro de byte e que podem ser úteis em diversas situações. No caso de uma memória limitada, pode-se armazenar diversas variáveis booleanas dentro de um byte. Dispositivos de rede podem transmitir informações codificadas nos bits dentro de um byte e, ainda, certas rotinas de criptografia necessitam de acesso aos bits dentro de um byte.

Embora as operações bit-a-bit que já citamos anteriormente possam ser usadas para essas tarefas, o uso de um campo de bits pode facilitar e dar mais eficiência ao código.

Um campo de bits pode ser criado por meio de uma estrutura em que cada um de seus membros tem um comprimento específico em bits. Por exemplo:

```
struct nome_do_campo{
      tipo nome1 : num_de_bits;
      tipo nome2 : num_de_bits;
      ...
      tipo nomeN : num_de_bits;
} variaveis;
```

O tipo deve ser declarado como `int`, `unsigned` ou `signed`. No caso de um campo de bit ter tamanho 1, ele deve ser declarado como unsigned pois não pode ser `signed` com um único bit.

Um exemplo é o retorno de uma porta serial, cujo estado é representado em um byte com as informações na tabela 4.1.

Bit	**Se ligado:**
0	Transmissão de dados
1	Recepção de dados
2	borda de transmissão da portadora
3	mudança em receive
4	clear-to-send
5	data-set-ready
6	sinal de chamada
7	sinal recebido

Tabela 4.1: Tabela de funções da porta serial

O código para representar estas funções pode ser descrito pelo seguinte campo de bits:

```
struct serial{
   unsigned dtr:  1;
   unsigned rts:  1;
   unsigned tr:   1;
   unsigned dr :  1;
   unsigned cts:  1;
   unsigned dsr:  1;
   unsigned ring: 1;
   unsigned line: 1;
}
```

4.4 Estruturas autoreferenciadas

O conceito de estruturas autoreferenciadas é fundamental para a implementação de listas, filas, pilhas, grafos e estruturas em árvores. Estes conceitos também são muito utilizados em compiladores e sistemas operacionais. Grande parte das estruturas utilizadas em kernel de sistemas operacionais e símbolos descritores de hardware para uso com módulos e drivers são implementadas por meio de estruturas autoreferenciadas.

Uma estrutura autoreferenciada contém um membro que é um ponteiro para uma estrutura do mesmo tipo da própria estrutura, por exemplo:

```
struct nodo{
  int dado;
  struct nodo *proxptr;
}
```

No caso acima, o ponteiro `proxptr` aponta para uma estrutura do tipo `struct nodo` e funciona como um link que pode vincular uma estrutura do mesmo tipo à estrutura referenciada. Por isso o nome de "autoreferenciada".

Dois ou mais objetos de estrutura autoreferenciada podem ser unidos para formar uma lista, uma fila, uma pilha ou uma árvore. O ultimo elemento da estrutura deve ter um apontador para `\NULL` como no caso de uma cadeia de caracteres (string). Este ponteiro para NULL indica o fim de uma estrutura de dados.

Capítulo 5

A Memória

A linguagem C permite a manipulação da memória de 3 maneiras. A primeira é a alocação estática da memória de acordo com a declaração das variáveis e das estruturas de dados utilizadas no programa. O segundo modo é a alocação automática da memória diretamente na pilha para utilização pelas variáveis e parâmetros de funções tanto na chamada como no retorno de valores. A terceira maneira de alocação de memória é a alocação dinâmica que pode ser gerenciada de forma mais flexível.

5.1 Alocação estática e automática

Nas alocações estáticas e automáticas de variáveis, o tamanho da memória tem que ser conhecido em tempo de compilação.

As declarações, no exemplo abaixo, alocam memória para diversas variáveis. A alocação é estática, uma vez que ocorre antes que o programa seja executado:

```
...
   char c; int i; float f[10];
...
```

Neste caso, o segmento de memória de dados deve ter tamanho adequado para um caractere (`char c`), um inteiro (`int i`) e um vetor de 10 posições para variáveis de ponto flutuante de 32 bits, ou IEEE 754 de precisão simples (`float f[10]`).

Na medida do possível, a memória é alocada sequencialmente, dependendo do tamanho do bloco de `word` da arquitetura e isto pode ser testado facilmente. Por exemplo:

```
/* Tipos de variaveis e enderecos */
#include <stdio.h>
#include <stdlib.h>

int main (){
   int x = 5;
   double y = 5.5;
   char a='A';
   char* b ="ab";

   printf ( "Val x: %d  Val y: %f  Val a: %c  Val b: %s\n",\
    x,y,a,b);
   printf ( "End x: %X  End y: %X  End a: %X  End b: %X\n",\
      &x,&y,&a,&b);
   exit(0);
}
```

Além disso, é bom lembrar que cada arquitetura pode ter tamanhos diferentes de tipos, ou seja, podemos ter `int` de 2 ou 4 bytes, por exemplo. O uso do operador `sizeoff()` é fundamental para sabermos quanto de memória será utilizada.

5.2 Alocação dinâmica da memória

Se o tamanho da memória não é conhecido antes, mas apenas durante a execução do programa, por exemplo, como um dado fornecido pelo usuário ou por uma leitura do programa, então a alocação estática não é adequada.

Para resolver este problema usa-se a alocação dinâmica da memória, onde a memória pode ser gerenciada mais diretamente e de forma mais flexível. Para isso deve-se incluir a biblioteca `stdlib.h`.

A alocação dinâmica de memória é fundamental para implementação de estruturas de dados dinâmicas, tais como as estruturas autoreferenciadas e é um recurso muito utilizado no desenvolvimento de drivers de dispositivos e módulos do kernel de sistemas operacionais.

5.2.1 malloc

A função malloc (abreviatura de *memory allocation*) aloca um bloco de bytes consecutivos na memória do computador e devolve o endereço desse bloco. O bloco é alocado na memória da pilha e a função `malloc` retorna um apontador para este endereço de memória. Usando este apontador, o programa tem acesso ao bloco de memória dinâmica. Quando a memória não for mais necessária, o apontador é passado para a função `free` que libera a pilha para outros propósitos [18].

O número de bytes é especificado no argumento da função. Por exemplo, no seguinte fragmento de código, `malloc` aloca 2 bytes para um tipo `short int`:

```
...

short int *iptr;
iptr = malloc(2);
scanf("%d", iptr);

...
```

O endereço devolvido por malloc é do tipo `void *` e um ponteiro do tipo referenciado armazena este endereço.

Quando se deseja alocar um tipo de dados de vários bytes, como por exemplo uma estrutura (`struct`) é necessário usar um operador que calcula o tamanho em bytes do tipo adequado. O operador `sizeoff()` calcula quantos bytes um determinado tipo tem. Por exemplo:

```
...
typedef struct {
  int dia, mes, ano;
} data;
data *d;
d = malloc( sizeof (data));
d->dia = 31; d->mes = 12; d->ano = 2008;
...
```

Convém lembrar que a memória é finita, assim é preciso testar para ver se há memória suficiente para ser alocada dinamicamente. Por exemplo:

```
void *malloc_teste(size_t nbytes){
  void *ptr;
  ptr = malloc( nbytes);
  if (ptr == NULL){
    printf( "ERRO! Memoria insuficiente!\n");
    exit( EXIT_FAILURE);
  }
  return ptr;
}
```

Assim, é possível verificar antes de alocar se há memória suficiente. Note que o parâmetro `size_t` é geralmente equivalente a `unsigned int`.

5.2.2 free

A memória utilizada por variáveis estáticas de uma função é liberada assim que a função termina. A memória usada pelas variáveis alocadas dinamicamente não é liberada e deve ser "limpa" ou liberada pela função `free()`.

A função `free` é chamada para desalocar e liberar a memória utilizada pela chamada da função `malloc`. A cada chamada `malloc` aloca um bloco de bytes consecutivos maior que o solicitado: os bytes adicionais são usados para guardar informações sobre o bloco (essas informações permitem que o bloco seja corretamente desalocado, mais tarde, pela função `free`).

O número de bytes adicionais pode ser grande, mas não depende do número de bytes solicitado no argumento de `malloc`. Assim, deve-se evitar chamar o `malloc` repetidas vezes com argumento muito pequeno. É preferível alocar um grande bloco de bytes e retirar pequenas porções desse bloco na medida do necessário.

A função `free` libera a memória alocada por `malloc`. O comando `free(ptr)` avisa que o bloco de bytes apontado por `ptr` está livre. A próxima chamada de `malloc` poderá usar esses bytes.

No exemplo abaixo, temos a alocação de uma variável como array de tamanho definido pelo usuário:

```
...
   float *vetor;
   int n, i;
   printf("Entre o tamanho do vetor: ");
   scanf("%d", &n);

   vetor = malloc(n * sizeof(float));

   for(i = 0; i<n; i++){
     printf("Entre o termo %d: ", i);
     scanf ("%f", &vetor[i]);
   }

printf("(");
   for(i=0; i<n-1; i++){
     printf("%f ", vetor[i]);
   }
   printf("%f)\n", vetor[i]);

   free(vetor);
   ...
```

5.2.3 kmalloc e vmalloc

A alocação dinâmica de memória no *kernel space* não é tão trivial quanto a alocação de memória em *user space*. Isto se deve a diversos fatores que têm que ser considerados.

Primeiramente, o kernel está limitado a 1GB de memória virtual e física. Depois, a memória do kernel não pode ser paginada sem se fragmentar e, por isso, deve-se levar em conta que o kernel utiliza memória física contígua. Ainda

é preciso lembrar que o kernel deve alocar memória sem passa para o estado de *sleep*. Além de tudo isso, lembre-se:

rros no *kernel space* pagam um preço muito maior do que erros em *user space*: KERNEL PANIC!"

Quando se trata de administrar a memória em códigos do núcleo (kernel) do sistema operacional as funções `malloc` e `free` não estão disponíveis porque os dispositivos de disco ainda não foram montados quando o kernel é carregado e, por isso, a `stdlib.h` não está disponível. Para estes casos, existemas funções próprias do kernel: o `kmalloc` e o `vmalloc`.

O `kmalloc` pode ser usado para alocação de memória menores que uma página, ou seja menor que a constante `PAGE_SIZE` (4096 Bytes na maioria dos casos). Esta alocação é contígua e como esta alocação de memória não usa o mapeamento da memória, ela toma a memória diretamente do mapamento físico do kernel.

A interface para allocação de memória no kernel (`kmalloc()`) é:

```
#include <linux/slab.h>

void * kmalloc(size_t size, int flags);
```

Do mesmo modo que `malloc()`, `kmalloc` pode falhar e deve ser testado para verificar se a memória foi alocada ou não:

```
struct allocat *p

p = kmalloc (sizeof(struct allocat), GFP_KERNEL);

if(!p){
  /* alocacao falhou... tratar o erro! */
  ... etc.
}
```

Existem muitos sinalizadores (*flags*) para tratar a alocação de memória pelo kernel. A sinal `GFP_KERNEL` é o mais comum e especifica uma alocação dinâmica normal. Usa-se este sinal em execução de códigos no contexto de processos sem rotinas de *lock*. Chamar o `kmalloc` com esta *flag* possibilita que o processo passe para o estado *sleep* assim, este sinal deve ser usado quando for seguro fazê-lo. Colocar o processo em *sleep* possibilita a liberação de memória se necessário.

Em situações onde os processos do kernel não podem "dormir" usa-se a *flag* `GFP_ATOMIC`. Neste caso o sinal instrui a alocação de memória para não bloquear nunca, sendo usado em situações onde o processo não pode ser posto em estado de *sleep* (veja tabela 5.1.

Flags	**Descrição**
`GFP_ATOMIC`	A alocação tem alta prioridade e não pode entrar em estado *sleep*. Usada em manipulação de interrupções, principalmente.
`GFP_DMA`	A alocação é do tipo de acesso direto (DMA). Tipicamente para drivers que necessitam acesso direto à memória.
`GFP_KERNEL`	Alocação normal de bloco da memória. Este sinal é típico de processos que podem ser postos para *sleep*.
`GFP_NOFS`	Esta alocação pode bloquear e iniciar I/O de disco. Geralmente usada em iniciação de sistemas de arquivos e operações similares.
`GFP_NOIO`	Alocação que bloqueia a memória mas não bloqueia iniciação de I/O. Usada quando é necessário mais blocos de I/O.
`GFP_USER`	Alocação normal e pode bloquear. Este sinal é usado para alocar memória para processos do *user space*.

Tabela 5.1: Tabela de sinais do kernel para alocação de memória

É importante lembrar que, em casos de alocação maior que `PAGE_SIZE` haverá risco de fragmentação e a chamada poderá falhar mesmo havendo memória suficiente. Se desejar máxima eficiência na alocação de memória em assuntos de kernel, crie seu próprio `kmem_cache` para cada estrutura necessária. A vantagem é que o uso da memória pode ser monitorado via `/proc`.

A alocação de mais de uma página de memória pode ser feita por mapeamento de memória no *kernelspace* usando-se a função `vmalloc`. A alocação é semelhante ao malloc do *userspace*, ou seja, o kernel aloca as páginas de memória por meio de mapeamento dos endereços virtuais delas. Este processo de alocação é mais lento que o `kmalloc` e pode acontecer pequenos atrasos no acesso por causa do *overhead* gerado pelo mapeamento dos endereços virtuais.

Capítulo 6

O Compilador C (GCC)

O GCC ou **Gnu C Compiler**[1] é o programa chamado para preprocessar, compilar, montar (*Assemble*) e "linkar" (*link*) o código programado em linguagem C ou C++. O GCC foi desenvolvido para ser 100% *free software* distribuído sob licença GPL [19] e mantido por um comitê de desenvolvedores.

Além de compilar código em C e C++, o GCC pode ser usado para compilar código Assembly, Ada, FORTRAN (f77 e f95), objective C/C++, java e go. O manual do Unix (`man gcc`) tem mais de 16.000 linhas e explica todas as milhares opções do programa. Estas opções se relacionam tanto com a linguagem e o tipo de compilação como, também, com a arquitetura e o sistema operacional. Desta forma, este capítulo não pode ter a pretensão de documentar todo o GCC mas, na medida do possível, apresentar aspectos gerais e mais comuns da compilação.

6.1 Usando o GCC

O comando mais comum do GCC é invocá-lo com o nome do código em C a ser compilado. Ou seja:

```
$ gcc prog.c
```

Ao ser invocado desta forma, o gcc gera um arquivo executável com o nome de saída: `a.out`.

[1]Na verdade o nome oficial é GNU project C and C++ Compiler.

Quando se deseja especificar um arquivo de saída (um nome para o executável) deve-se utilizar a opção `-o`, por exemplo:

```
$ gcc -o executavel fonte.c
```

O GCC tem um número muito grande de opções de compilação e de préprocessamento conforme pode ser encontrado em sua documentação oficial [20]. As opções de otimização do código (`-O`)[2] tem vários níveis: `-O -O0 -O1 -O2 -O3 -Os -Ofast` e, estas opções ainda disponibilizam *flags* de dependência da arquitetura da máquina.

Outra opção de compilação são os níveis de aviso (*Warning*) em tempo de compilação (`-W`). A opção `-Wall` liga todos os avisos.

A principal diretiva de preprocessamento do compilador C é a diretiva `#include`. Esta diretiva insere uma cópia do arquivo especificado no local onde ela se encontra. Uma diretiva `include` pode especificar um arquivo em um diretório padrão do ambiente do compilador ou um diretório indicado pelo usuário. Por exemplo:

```
#include <arq1.h>
#include "arq2.h"
```

No primeiro caso, o compilador procura no diretório padrão do ambiente do compilador, ou seja, no diretório `/usr/include` e, no caso do nome do arquivo entre aspas o compilador procura no diretório local ou no diretório indicado pela opção `-I` da linha de comando na chamada do compilador. Por exemplo:

```
$ gcc -I. -I../include -I/usr/local/share/include (...etc)
```

Outra diretiva importante em C é a diretiva `#define`. Esta diretiva é usada para criar símbolos ou constantes simbólicas e macros que definem operações destes símbolos. Por exemplo:

[2]Note que se trata da letra 'o' maiúscula e não o zero

```
#define PI 3.14159

#define AREA_CIRC(r)((PI)*(r)*(r))
```

Existem diretivas condicionais que são utilizadas para estabelecer a condição de compilação, por exemplo:

```
#if !defined(CONST_ZERO)
   #define CONST_ZERO 0
#endif

#ifdef __DEBUG__
   #define DEB_VAR 1
#else
   #define DEB_VAR 0
#endif

#ifndef SQRT2
   #define SQRT2 1.414213
#endif
```

As constantes simbólicas são especificadas em C por dois sublinhados antes do nome e dois sublinhados depois. Por exemplo, `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__` etc. e elas não podem ser utilizadas em diretivas `#define` ou `#undef`.

6.2 GCC e o Kernel

A constução do kernel depende das macros `-D__KERNEL__`, `-D__ASSEMBLY__` e `-DMODULE` que estão definidas no `Makefile` principal e nos arquivos de configuração do (`kconfig`).

```
KBUILD_CPPFLAGS := -D__KERNEL__

KBUILD_CFLAGS   := -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
                   -fno-strict-aliasing -fno-common \
                   -Werror-implicit-function-declaration \
                   -Wno-format-security \
                   -fno-delete-null-pointer-checks
KBUILD_AFLAGS_KERNEL :=
KBUILD_CFLAGS_KERNEL :=
KBUILD_AFLAGS   := -D__ASSEMBLY__
KBUILD_AFLAGS_MODULE  := -DMODULE
KBUILD_CFLAGS_MODULE  := -DMODULE
KBUILD_LDFLAGS_MODULE := -T $(srctree)/scripts/module-common.lds
```

6.3 O make

O `make` é um utilitário desenvolvido pela GNU que opera sobre *scripts* específicos chamados `Makefile` (exatamente assim). O propósito do `make` é analisar e determinar, automaticamente, quais as partes de um programa que devem ser compiladas, ou recompiladas, e invocar os comandos necessários para esta ação.

Um exemplo de `Makefile` pode ser:

```
CC = gcc
CFLAGS = -O2 -ansi -Wall
LIBDIR = -L/usr/local/lib
LIBS = -lm

INC =-I./ -I/usr/local/include

PROGS= fatorial

all: $(PROGS)

fatorial: fat.c main.c
  $(CC) $(CFLAGS) $(INC) $(LIBDIR) -o $@ $? $(LIBS)

clean:
  rm -f *~ *core *.o *.dat $(PROGS)
```

Neste exemplo, um programa chamado `fatorial` é criado a partir dos fontes `fat.c` e `main.c`, e o comando pode ser expandido para:

```
$ gcc -O2 -ansi -Wall -I./ -I/usr/local/include\
-L/usr/local/lib -o fatorial fat.c main.c -lm
```

O make é um utilitário muito poderoso e pode ser melhor conhecido a partir da sua `manpage` (use o comando `man make`) ou a documentação completa no site da GNU [21].

6.3.1 O make e o kernel

As regras de compilação do kernel do Linux são estabelecidas em arquivos de configuração denominados `Kconfig` ou `sysconfig` e são utilizados pelo utilitário `make`. Em cada nível de diretório de organização dos fontes do kernel se encontra um `Makefile` para orientar a compilação do kernel para cada arquitetura específica.

Estruturalmente, temos o Makefile principal no diretório *top* do kernel que é o responsável pela criação do **vmlinux** (imagem residente do kernel Linux) e os módulos.

A lista de subdiretórios a ser usados na compilação depende da configuração do kernel e da arquitetura para a qual o kernel será compilado. Assim, o `Makefile` principal inclui, textualmente, o `Makefile` em `arch/$(ARCH)/Makefile` que supre as informações específicas da arquitetura.

Cada subdiretório contém um `kbuild Makefile` específico que utiliza as informações dos arquivos `.config` onde consta a lista dos vários arquivos a ser usados para a construção dos alvos estáticos e modulares do kernel. As regras são definidas dentro dos arquivos dentro de `scripts/Makefile.*`.

Normalmente, os usuários não precisam saber detalhes sobre estes arquivos, uma vez que apenas compilam o kernel, usando os comandos `make menuconfig` e o `make`. Os desenvolvedores, por sua vez, são os trabalham em partes específicas tais como *drivers* de dispositivos, sistemas de arquivos, protocolos de rede etc. e, por isso, precisam manter os `kbuild Makefiles` para o subsistema no qual estão desenvolvendo. Assim, devem ter o conhecimento sobre os arquivos de construção do kernel e um conhecimento bem detalhado da *public interface* para o kbuild.

Diferentemente dos makefiles tradicionais, os makefile do kernel (kbuild) definem os arquivos a ser compilados, as opções de compilação e os diretórios onde se encontram. Um *kbuild makefile* bem simples pode conter apenas a linha:

```
obj-y += foo.o
```

Isto diz ao kbuild que deve ser contruído um objeto `foo.o` neste diretório que deve ser compilado a partir de `foo.c` ou `foo.S`. Caso `foo.o` seja um módulo, usa-se `obj-m`. O padrão mais utilizado é:

```
obj-$(CONFIG_FOO) += foo.o
```

onde, `$(CONFIG_FOO)` vai ser substituido por "y" (interno do kernel) ou "m" (módulo).

Exercícios:

1. Leia a documentação em:

```
/usr/src/linux.../Documentation/kbuild/modules.txt
```

Capítulo 7

O Processador ARM

O processador arm é um processador de 32 bits de alta performance, de arquitetura RISC e de baixo consumo. Tem sido utilizado em dispositivos móveis, tais como *Smartphones*, tablets etc. e, recentemente, em computadores de baixo custo e consumo ideais para controladores e sistemas embarcados.

Com uma estrutura de 31 registradores de 32 bits, sendo 16 deles, acessíveis em qualquer estado (*mode*), o ARM possue um conjunto bem enxuto e eficiente de instruções onde todas elas são condicionais. Por exemplo, pode-se testar o valor de um registrador e a condição é mantida até que novo teste da mesma condição é executado. Além disso, operações aritméticas e *shift operations* podem ser feitas enquanto os valores são carregados no registrador. [22]

O processador ARM possui 16 registradores (veja figura 7.1) que podem estar em diversos estados (*modes*):

Modo	**Mnem.**	**Descrição**
User	usr	Estado de execução normal
FIQ	fiq	Fast Interrupt (transf. rápida de dados)
IRQ	irq	Usado para manipulação de interrupções
Supervisor	svc	Modo protegido para o sistema operacional
Abort	abt	Implementa mem. virtual / mem protection
Undefined	und	Emul. de hardware de coprocess. via softw
System	sys	Roda tasks do SO em modo privilegiado

Tabela 7.1: Estados do processador ARM

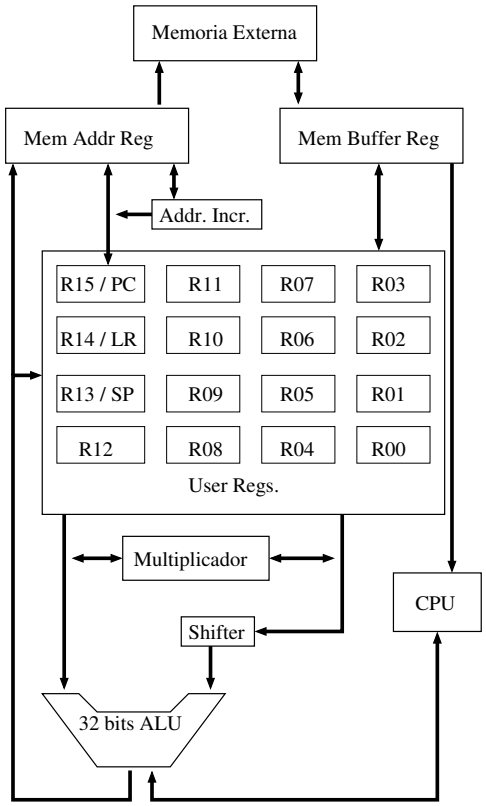


Figura 7.1: Esquema dos registradores do processador ARM

O modo *System* pode ser usado pelo usuário por meio de interrupções por software ou SWI (*software interrupt*).

O Raspberry Pi usa um processador ARMv7 (v6 *compatible* rev. v6l) que é um processador do tipo ARM Cortex A8. Que é denominado de *Application Processor*. Estes processadores são assim definidos pela habilidade que o processador tem de executar sistemas operacionais complexos tais como, Linux, Android etc.. Estes processadores do Raspberry Pi integram uma unidade de gerenciamento de memória ou MMU (*Memory Management Unit*) que facilitam os requisitos destes sistemas operacionais permitindo a execução de softwares de terceiros.

Processadores mono ou multicore, tais como o Cortex-A8, Cortex-A9, Cortex-A5, Cortex-A7, Cortex-A15, Cortex-A53 e Cortex-A57 possuem uma performance extendida e escalabilidade para habilitar até quatro núcleos implementados em sistemas simétricos ou assimétricos. Os Cortex-A50 podem executar instruções de 64 bits. [23]

7.1 ARM Assembly

Dentro da filosofia de "simplicidade e eficiência"seria de se esperar que o código Assembly da arquitetura ARM fosse extremamente econômico. E realmente é. Afinal, trata-se de um processador tipo *"Restrict Instruction System Computer"* (RISC). Existem apenas 5 classes de instruções no código Assembly do ARM [24].

1. **Operações com dados** ou *Data Operation* que é um conjunto de instruções mais frequentemente utilizadas. São 16 instruções e têm um formato muito similar entre si. Estas operações são efetuadas sobre os registradores ou valores imediatos localizados na instrução, nunca na memória diretamente. Exemplo, `ADD`, `MUL`, `CMP` etc..

2. **Transferência de dados** ou *Load and Save* são apenas duas instruções: carrega um valor no registrador ou salva o conteúdo de um registrador. Também indicam se os dados transferidos são bytes ou words e como os endereços usados são obtidos.

3. **Transferência múltipla** ou *Multiple load and save* que são instruções que permitem a transferência entre um e 16 registradores que serão movidos do processador para a memória. Apenas transferências de words são efetuadas neste grupo de instruções.

4. **Desvios** ou *Branching* que são utilizadas para desvios da sequência do programa. Embora o contador de programa (PC) ou apontador de instruções (IP) pode ser alterado diretamente, as instruções de desvios permitem redirecionar o programa para qualquer parte dos 64 MBytes do *address space* em uma única instrução de forma mais conveniente.

5. **Interrupções por software** ou *Software Interrupt* (SWI) que são instruções que permitem o acesso dos programas às facilidades do Sistema Operacional.

Nota: Os processadores ARM, originalmente, não têm suporte a instruções para ponto flutuante ou números reais. Esta facilidade está disponível em um co-processador adicional. As operações com ponto flutuante são implementadas via software.

Os mnemônicos do Assembly do ARM são formados por 3 letras, por exemplo, `ADD, MOV, CMP` etc. mas podem conter mais letras de condições. O Assembly do ARM permite operações condicionais, com as condições estabelecidas no próprio código de instrução. Cada instrução tem 4 de seus bits separado

para indicação de condição, podendo se usados para 16 condições possíveis. Se a condição for verdadeira, a instrução é executada, caso contrário será ignorada e a próxima instrução é carregada no processador.

Assim, por exemplo, `ADDAL` é uma operação de `ADD` executada sempre (*Always*), ou seja, incondicionalmente, enquanto que uma instrução `ADDEQ` só executada se os operandos forem iguais.

Estas extensões comparam os estados dos bits 28—31 que marcam os estados (flags) de condições. O bit 28 (V, se ligado, indica o estado de overflow; o bit 29 (C), se ligado, indica o estado de carry on; o bit 30 (Z), se ligado, indica zero; e o bit 31 (N), se ligado indica o estado negativo (ver tabela 7.2).

As extensões dos mnemônicos são:

Extensão	**significado**	**nota**
AL	Always	executa sempre
NV	Never	nunca executa
EQ	Equal	executa se operandos iguais
NEQ	Not Equal	executa se diferentes
VS	Overflow set	executa se flag V (overflow) estiver ligado
VC	Overflow clear	executa se flag V estiver desligado
MI	Minus	executa se flag N (negativo) estiver ligado
PL	Plus	executa se flag N estiver desligado
CS	Carry set	executa se flag C (carry) estiver ligado
CC	Carry clear	executa se flag C estiver desligado
HI	Higher	executa se flag C estiver ligado e Z é falso
LS	Lower or same	executa se forem menor ou igual (C desligado e Z ligado)
GE	Greater than or equal	executa se for maior ou igual (N e V desligado ou N e V ligado)
LT	Less than	executa se for menor que (N ligado e V desligado ou N desligado e V ligado)
GT	Greaer than	o mesmo que GE e flag Z desligado
LE	Less than or equal	o mesmo que LT e flag Z ligado

Tabela 7.2: Tabela de extensões de mnemônicos

7.2 ARM cross-compiling e o GCC

A compilação de software, desde aplicativos, sistemas operacionais e sistemas embarcados pode ser feita em outra arquitetura por meio de *cross-compiling*.

O GCC tem ferramentas para este tipo de compilação que pode ser instalado no Linux (Ubuntu/Debian), por exemplo, como:

```
$ sudo apt-get install binutils-arm-linux-gnueabi
```

A compilação pode ser feita para gerar um código ELF (linux ABI[1]) ou um código EABI (*Embeded ABI*).

A documentação do GNU binutils pode ser encotrada em:

```
http://sourceware.org/binutils/docs-2.23.1/
```

7.3 ARM e sistemas embarcados

O compilador C/C++ e o montador (GAS) suporta a arquitetura ARM. Existe muitas opções de compilação e é necessário conhecer todas elas para um ajuste fino para um projeto real desenvolvido para o ARM seja eficiente.

A arquitetura ARM tem algumas particularidades que devem ser compreendidas quando se trata de usar o processador em um sistema embarcado. Embora este assunto não seja escopo deste curso, alguns cuidados na compilação de códigos em C/C++ deve, ser tomados.

Em se tratando do Raspberry Pi os processadores ARM7 e ARM9 podem operar com instruções de 32 ou de 16 bits, além dos estados de operação da CPU. Os modos *USER, SYSTEM, SUPERVISOR, ABORT, UNDEFINED, IRQ* e *FIQ* diferem quanto à visibilidade dos registradores e execução de instruções. Além disso, controladores baseados em arquitetura ARM (ARM MCU) suportam o remapeamento de controladores de interrupções que permitem o aninhamento de IRQs.

Em primeiro lugar, o mapeamento dinâmico dos vetores de interrupção do ARM. Por exemplo o *reset* no endereço `0x0000` dos primeiros 32 bytes de memória devem ser mapeados na ROM. Alguns microcontroladores ARM possibilitam o remapeamento na RAM que pode ser dinamicamente alocado pelo software de controle ou pelo kernel do SO.

Em segundo lugar, a iniciação do clock da CPU, em diversos casos de configuração de hardware e outras interfaces, não necessita da programação em Assembly e pode ser realizada em código C/C++.

[1] ABI significa *Application Binary Interface*.

Samek, em seu artigo [25] abrange estes e outro problemas endereçados pelo *design* de sistemas embarcados em arquitetura ARM.

Alguns aspectos das opções de compilação conforme a documentação da própria GNU [20] pode ser encontrado em:

`http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html`

Bibliografia

[1] WIKIPEDIA. *Dennis Ritchie — Wikipedia, a enciclopédia livre*. 2013. [Online; accessed 5-agosto-2013]. Disponível em: <http://en.wikipedia.org/wiki/Dennis_Ritchie>.

[2] KERNIGHAN, B.; RITCHIE, D. *C, A Linguagem de Programação*. 4a.. ed. [S.l.]: Campus, 1988. ISBN 8570014104.

[3] KERNIGHAN, B.; RITCHIE, D. *C Programming Language*. 4th. ed. [S.l.]: Prentice Hall, 1988. ISBN 0-13-110362-8.

[4] RITCHIE, D. The development of the c language. *ACM*, Association for Computing Machinery, Inc., p. 1–16, 1993. Disponível em: <http://heim.ifi.uio.no/ inf2270/programmer/historien-om-C.pdf>.

[5] RITCHIE, D. M. The development of the c language. In: *The second ACM SIGPLAN conference on History of programming languages*. New York, NY, USA: ACM, 1993. (HOPL-II), p. 201–208. ISBN 0-89791-570-4. Disponível em: <http://doi.acm.org/10.1145/154766.155580>.

[6] TANENBAUM, A. S. *Structured Computer Organization*. [S.l.]: Englewood Cliffs, New Jersey: Prentice-Hall, 1979. ISBN 0-13-148521-0.

[7] TANENBAUM, A. S. *Organização estruturada de computadores*. Prentice-Hall, 1992. Disponível em: <http://books.google.com.br/books?id=Dj-9XwAACAAJ>.

[8] MANUAL. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*. [S.l.], 2012. Publ. nr. 24592, rev. 3.19. Disponível em: <http://support.amd.com/us/Processor_TechDocs/24592_APM_v1.pdf>.

[9] RISC Principles. In: GUIDE to RISC Processors. [S.l.]: Springer New York, 2005. p. 39–44. ISBN 978-0-387-21017-9.

[10] ARM. *ARM Annual Report & Accounts 2011*. [S.l.], 2011. Disponível em: <http://ir.arm.com/phoenix.zhtml?c=197211&p=irol-reportsannual>.

[11] WIKIPEDIA. *Magnetic-core memory — Wikipedia, a enciclopédia livre*. 2013. [Online; accessed 22-julho-2013]. Disponível em: <http://en.wikipedia.org/wiki/Magnetic-core_memory>.

[12] TANENBAUM, A. *Structured computer organization*. Upper Saddle River, N.J: Pearson Prentice Hall, 2006. ISBN 9780131485211.

[13] MONTEIRO, M. *Introdução à organização de computadores*. Livros Técnicos e Científicos, 2002. ISBN 9788521612919. Disponível em: <http://books.google.com.br/books?id=mY-NAQAACAAJ>.

[14] COCKERELL, P. J. *ARM Assembly Language Programming*. Computer Concepts Ltd., Hertfordshire, UK, 1987. ISBN 9780951257906. Disponível em: <http://books.google.com.br/books?id=pcpqAAAACAAJ>.

[15] WIKIPEDIA. *Endianness — Wikipedia, a enciclopédia livre*. 2013. [Online; accessed 7-agosto-2013]. Disponível em: <http://en.wikipedia.org/wiki/Endianness>.

[16] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, p. 1–58, 29.

[17] DEITEL, P. J.; DEITEL, A. M. *C/C++ : como programar*. 5a.. ed. São Paulo: Pearson Prentice Hall, 2006. ISBN 9788576050568.

[18] FEOFILOFF, P. *Algoritmos em linguagem C*. [S.l.]: Campus / Elsevier, 2008–2009. ISBN 978-85-352-3249-3.

[19] WIKIPEDIA. *GNU General Public License — Wikipédia, a enciclopédia livre*. 2013. [Online; accessed 7-agosto-2013]. Disponível em: <http://pt.wikipedia.org/wiki/GNU_General_Public_License>.

[20] STALLMAN, R. M. et al. *Using the GNU Compiler Collection - for GCC version 4.7.3*. Boston, MA, USA: Gnu Press - Free Software Foundation, 2010. [Online; accessed 5-agosto-1013]. Disponível em: <http://gcc.gnu.org/onlinedocs/gcc-4.7.3/gcc.pdf>.

[21] STALLMAN, R. M.; MCGRATH, R.; SMITH, P. D. *Gnu Make, A Program for Directing Recompilation*. Boston, MA, USA: Gnu Press - Free Software Foundation, 2010. [Online; accessed 5-agosto-1013]. Disponível em: <http://www.gnu.org/software/make/manual/make.pdf>.

[22] RUSLING, D. A. *The Linux Kernel*. [s.n.], 1999. Disponível em: <http://www.tldp.org/LDP/tlk/tlk.html>.

[23] ARM. *ARM Processors*. [S.l.], 2013. Disponível em: <http://www.arm.com/products/processors/index.php>.

[24] COCKERELL, P. *ARM Assembly Language Programming*. Hertfordshire, England, UK: Computer Concepts Ltd., 2003. Disponível em: <http://peter-cockerell.net/aalp/resources/pdf/all.zip>.

[25] SAMEK, M. *Building Bare Metal ARM Systems with GNU*. Quantum Leaps, LCC, 2007. [Online; accessed 5-agosto-1013]. Disponível em: <http://www.state-machine.com/arm/Building_bare-metal_ARM_with_GNU.pdf>.