

ELF52 - Sistemas Microcontrolados

Linguagem *Assembly*

Professor:

Prof. Marcos Eduardo

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Linguagem de Máquina

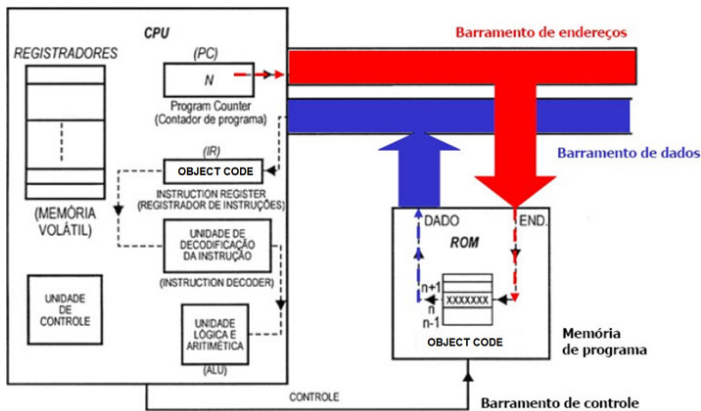
Linguagem de Máquina x Assembly

- Um microcontrolador executa comandos específicos, que são constituídos de números binários;
- Estes comandos ou **object codes** constituem a linguagem de máquina;
- As instruções *assembly* e os comandos de uma linguagem alto nível, são traduzidos em linguagem de máquina.



LOS VERDADEROS PROGRAMADORES
PROGRAMAN EN BINARIO

Linguagem de Máquina

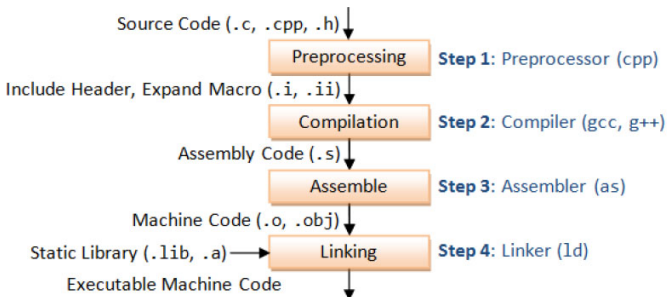


Linguagem Assembly

- Para facilitar a vida do programador, criou-se a Linguagem *Assembly*, que possui o mesmo conjunto de instruções, porém utiliza símbolos (**mnemônicos**) no lugar dos números;
- A conversão da linguagem **assembly** para a linguagem de máquina é feita pelo **assembler** (montador) que, dentre outras coisas, transforma os **mnemônicos** em códigos binários (**opcodes**). Não confunda!
- Entretanto, ainda é específico para cada tipo de CPU, sendo considerada uma linguagem de baixo nível.

Linguagem de Alto Nível

- Há algumas linguagens mais próximas à linguagem humana:
 - C, C++, Pascal, Java, etc.
- De maneira geral, a conversão é feita da seguinte forma:



Linguagem *Assembly ARM Cortex-M4*

Assembly ARM Cortex-M4

- Tecnologia Thumb-2:
 - Mistura de instruções de 16 e 32 bits;
 - Instruções ARM (32) + *Thumb* (16).
- Arquitetura *LOAD/STORE*.

Assembly ARM Cortex-M4

- O código fonte do *assembly* é um arquivo de texto (.s ou .asm);
- Por exemplo, uma função que recebe R0 como entrada:

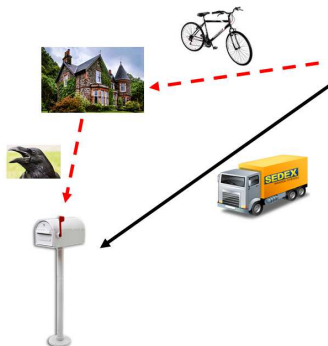
```
1 Func MOV    R1, #100      ; R1=100
2      MUL    R0, R0, R1    ; R0=100*input
3      ADD    R0, #10       ; R1=100*input+10
4      BX     LR           ; retorna 100*input+10
```

Modos de Endereçamento

- As instruções operam com **dados** e **endereços**;
- Formato que a instrução usa para especificar a localização da **memória** para ler ou escrever **dados**.

- Modos:

- Imediato;
- Registrador;
- Indexado;
- Relativo ao PC.



Endereçamento Imediato

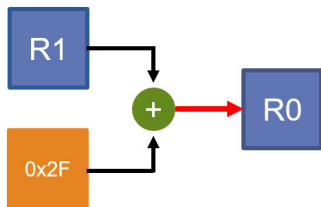
- Uma constante pode ser colocada dentro do código de instrução:
 - Definida por uma **hashtag** ('#') antes do operando;
- Exemplo:



```
1 MOV R0, #25      ;move const. dec 25 para reg R0
2 MOV R1, #0x2F    ;move const. hex 2Fh para reg R1
3 MOV R2, #2_1101  ;move const. bin 00001101 para R2
```

Endereçamento por Registrador

- Algumas instruções podem operar dados com registradores do microprocessador:
- Registrador com modo imediato. Exemplo:



ADD R0, R1, #0x2F

```

1 ADD R1, R2, #18    ;R1 <= R2 + 18
2 AND R0, R1, #0x0F  ;R0 <= R1 & 0x0F
3 MUL R0, R2, #8     ;R0 <= R2 * 8
  
```

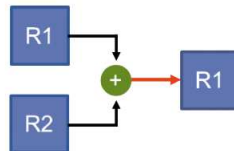
Endereçamento por Registrador

- Exemplos:

MOV R1, R2



ADD R1, R2



```
1 ADD R0, R1, R2 ;R0 <= R1 + R2
2 ADD R3, R4      ;R3 <= R3 + R4
3 MOV R1, R3      ;R1 <= R3
```

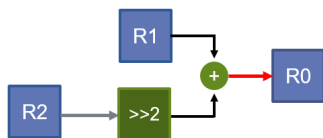
Endereçamento por Registrador

- Exemplo de registrador escalado:

MOV R0, R1, LSL #3



ADD R0, R1, R2, LSR #2



| | | | |
|---|-----|--------------------|----------------------------|
| 1 | MOV | R0, R1, LSL #3 | ;R0 <= (R1 << 3) |
| 2 | ADD | R0, R1, R2, LSR #2 | ;R0 <= R1 + (R2 >> 2) |
| 3 | MOV | R1, R3, ASR #7 | ;R1 <= R3 / 128 (signed) |
| 4 | MOV | R2, R4, LSR #7 | ;R2 <= R4 / 128 (unsigned) |

Endereçamento Indexado

- Instruções ARM não suportam operações de memória para memória (RAM ou ROM);
- Somente as instruções LDR/STR podem acessar a memória;
- Os registradores atuam como ponteiros para a memória:

```
1 LDR R0, [R1]      ;R0 <= [R1]  R0 recebe o dado  
2                  ;apontado por R1  
3 LDR R0, [R1, #0] ;R0 <= [R1+0] o mesmo que R0 =[R1]  
4 STR R2, [R0, #4] ;[R0+4] <= R2 guarda o dado R2  
5                  ;endereço apontado por R0 + 4
```

Endereçamento Relativo ao PC

- Se o ponteiro for o *Program Counter* (PC), é usado para:
 - Saltos (*Branches*);
 - Chamadas de funções;
 - Acesso à constantes salvas na ROM.

Endereçamento Relativo ao PC

- Exemplos:

```
1 B      label      ; pula para label
2 BL     subrotina ; salva PC no R14 (LR) e chama subrotina
3 BX     R14        ; return ou MOV PC, R14 (LR)
4
5 LDR     R1, =Count ; R1 aponta para variável Count
6 LDR     R0, [R1]    ; R0 <= valor apontado por R1
```

Tipos de Operandos

Operandos

- Nas instruções *assembly*, a seguinte lista de símbolos pode ser utilizada:

| Instrução | Descrição |
|----------------------|---|
| Ra Rd Rm Rn Rt e Rt2 | Registradores |
| {Rd,} | Registrador de destino opcional |
| #imm12 | Constante de 12 bits, 0 a 4095 |
| #imm16 | Constante de 16 bits, 0 a 65535 |
| operand2 | Segundo operando flexível * |
| {cond} | Condição lógica opcional * |
| {type} | Estabelece um tipo de dado opcional * |
| {S} | Opcional: seta os bits de condição do PSR |
| Rm {, shift} | Deslocamento opcional no Rm |
| Rn {, offset} | Offset opcional no Rn |

- * Descritos a seguir.

Operando2

| operand2 | |
|-------------|---|
| #constant | Valor imediato de 8 bits* |
| Rm , <opsh> | Registrador, deslocado opcionalmente como abaixo |
| Rm, LSL Rs | Reg. Rm com desloc. lógico (unsig.) para esquerda definido por Rs |
| Rm, LSR Rs | Reg. Rm com desloc. lógico (unsig.) para direita definido por Rs |
| Rm, ASR Rs | Reg. Rm com desloc. aritmético (sig.) para direita definido por Rs |
| Rm, ROR Rs | Reg. Rm com rotação lógica para direita definido por Rs |

- O operando 2 aceita os seguintes valores para a constante:
 - Constante produzida deslocando um valor de 8 bits para esquerda por qualquer número de bits;
 - Constante na forma 0x00XY00XY;
 - Constante na forma 0xXY00XY00;
 - Constante na forma 0xXYXYXYXY.

Tipos de Instruções

Operações de Acesso à Memória

- Acesso à memória de código ou de dados:
 - **LD** lê dados da memória;
 - **ST** escreve dados na memória;
 - Instruções de processamento de dados não acessam a memória;
 - Arquitetura **LOAD-STORE**.
- Para operações com dados em memória, é necessário:
 - Leitura da memória em registrador;
 - Operação;
 - Escrita em memória.

Operações de Acesso à Memória

- Para acessar a memória, **sempre** estabeleça um **registrador ponteiro** (ou base) para o objeto;
- Exemplos:

Estado Inicial

Registradores

| | |
|----|------------|
| R0 | 0x20000000 |
| R1 | 0x00 |
| R2 | 0x00 |
| R3 | 0x50003210 |
| R4 | 0x00 |

Memória

| | |
|------------|------------|
| 0x20000000 | 0x44332211 |
| 0x20000004 | 0x88776655 |
| 0x20000008 | 0x12345678 |
| 0x2000000C | 0xDDCCBBAA |
| 0x20000010 | 0x99999999 |

Operações de Acesso à Memória

- Exemplos:

LDR R4, [R0]



- O primeiro operando é obrigatoriamente por registrador e segundo operando é indexado (`[]`).

Operações de Acesso à Memória

- Exemplos:

STR R3, [R0]



- O primeiro operando é obrigatoriamente por registrador e segundo operando é Indexado ([]).

Tipos de Acesso à Memória

- O registrador que contém o endereço ou a localização dos dados é chamado de **registrador base**;
- Utiliza-se o modo de endereçamento indexado;
- Há três tipos: Com *offset*, pré-indexado, pós-indexado.

Tipos de Acesso à Memória

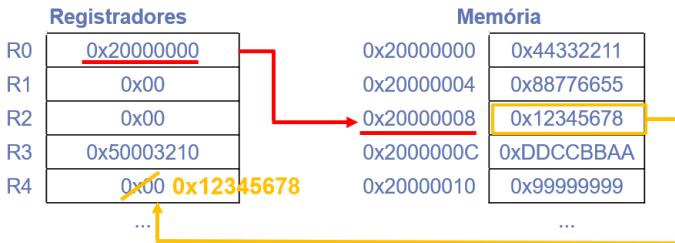
- Com *offset*:
 - O endereço é incrementado ANTES da operação, mas o valor do registrador base NÃO é alterado;
 - [XX]** → Conteúdo apontado por XX
 - Exemplos:

```
1 LDR R0, [R1]           ; R0 <= [R1]  R0 recebe o dado
2                         ; apontado por R1
3 LDR R0, [R1, #8]       ; R0 <= [R1+8] R0 recebe o
4                         ; dado apontado pelo
5                         ; R1 + 8
6 STR R2, [R0, #4]       ; [R0+4] <= R2 guarda o dado
7                         ; R2 ender. apontado por R0+4
```

Acesso à Memória: Com *Offset*

- Exemplo:

```
LDR    R4, [R0, #8]
```



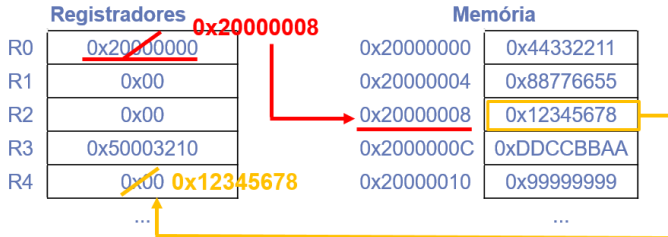
Tipos de Acesso à Memória

- Pré-Indexado → '!':
 - O endereço é incrementado ANTES da operação, e o valor do registrador base é salvo;
- Exemplos:

| | | |
|---|----------------------------------|-------------------------------|
| 1 | LDR R0, [R1, #2]! | ;R0 <= [R1 + 2], R1 <= R1 + 2 |
| 2 | STR R2, [R3, #4]! | ;[R3 + 4] <= R2, R3 = R3 + 4 |
| 3 | LDR R0, [R1, R2]! | ;R0 <= [R1 + R2], R1 <= R1+R2 |
| 4 | LDR R0, [R1, R2, LSR #2]! | ;R0 <= [R1 + (R2 << 2)] |
| 5 | | ;R1 <= R1 + (R2 << 2) |

Acesso à Memória: Pré-Indexado

- Exemplo:



Tipos de Acesso à Memória

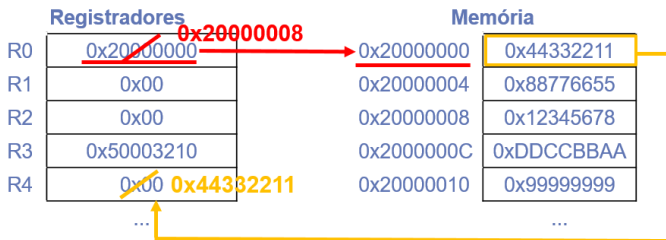
- Pós-Indexado:
 - O endereço é incrementado após a operação e o valor do registrador base é salvo.
- Exemplos:

```
1 LDR    R0, [R1], #8    ;R0 <= [R1]  R1 <= R1 + 8
2 STR    R2, [R3], R4    ;[R3] <= R2, R3 = R3 + R4
3 LDR    R0, [R1], R2, LSR #2 ;R0 <= [R1]
4                                ;R1 <= R1 + (R2 << 2)
```

Acesso à Memória: Pós-Indexado

- Exemplo:

LDR R4, [R0], #8



Tipos de Acesso à Memória

- Tabela Comparativa:

| Modo | Mnemônico <i>Assembly</i> | Endereço Acessado | Valor Final no Registrador Base |
|---|------------------------------|----------------------|------------------------------------|
| Com Offset, base não alterada | LDR R0, [R1, #d] | $R1 + d$ | R1 |
| Pré-indexado, base alterada | LDR R0, [R1, #d]! | $R1 + d$ | $R1 + d$ |
| Pós-indexado, base alterada | LDR R0, [R1], #d | R1 | $R1 + d$ |

Operações de Acesso à Memória

- Algumas das instruções *load/store*:

```
1 LDR{type}{cond} Rd, [Rn] ;load memory at [Rn] to Rd
2 STR{type}{cond} Rt, [Rn] ;store Rt to memory at[Rn]
3 LDR{type}{cond} Rd, [Rn, #n] ;load memory at[Rn+n]to Rd
4 STR{type}{cond} Rt, [Rn, #n] ;store Rt to memory [Rn+n]
5 LDR{type}{cond} Rd, [Rn, #n]! ;load memory at[Rn+n]to Rd
6 ;Rn := Rn + n
7 STR{type}{cond} Rt, [Rn, #n]! ;store Rt to memory [Rn+n]
8 ;Rn := Rn + n
9 LDR{type}{cond} Rd, [Rn], #n ;load memory at [Rn] to Rd
10 ;Rn := Rn + n
11 STR{type}{cond} Rt, [Rn], #n ;store Rt to memory [Rn]
12 ;Rn := Rn + n
```

Tipos de dados da memória

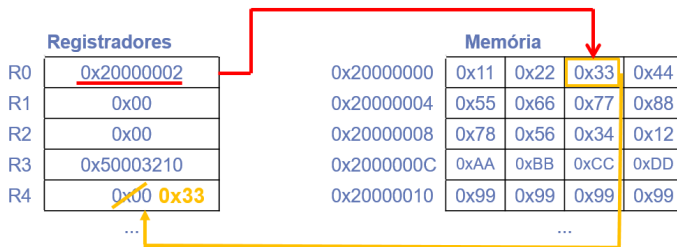
- Em relação a dados de memória, pode-se acessar dados de 8, 16, 32 ou 64 bits. Para 8 e 16 bits pode ser com sinal ou sem sinal;
- Ao colocar um valor de 8 bits ou 16 bits em um registrador:
 - Se for uma operação sem sinal, os bits mais significantes são preenchidos com 0;
 - Se for uma operação com sinal, os bits mais significantes serão iguais ao bit de sinal.
- É **dever** do programador saber como a memória será acessada.

| {type} | Tipo do dado | Significado |
|--------|-------------------------------|---|
| | Word de 32 bits | 0 a +4.294.967.295 ou -2.147.483.648 a +2.147.483.647 |
| B | Byte de 8 bits sem sinal | 0 a 255 |
| SB | Byte de 8 bits com sinal | -128 a +127 |
| H | Halfword de 16 bits sem sinal | 0 a 65535 |
| SH | Halfword de 16 bits com sinal | -32768 a +32767 |
| D | 64-bits | Usa dois registradores |

Operações de Acesso à Memória

Exemplo:

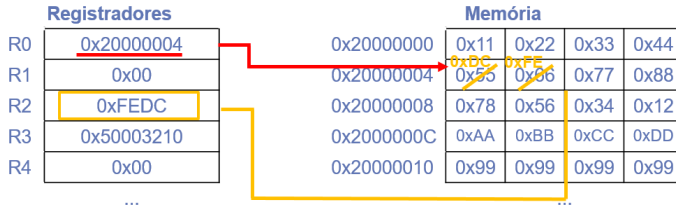
LDRB R4, [R0]



Operações de Acesso à Memória

Exemplo:

STRH R2, [R0]



Operações de Transferência

- Para mover valores entre registradores ou uma constante e um registrador;
- Só conseguimos transferir no máximo valores de até 16 bits:

```
1 MOV{S}{cond} Rd, <op2> ; set Rd equal to the value  
2                           ; specified by op2  
3 MOV{cond} Rd, #im16      ; set Rd equal to im16, im16  
4                           ; is  
5                           ; 0 to 65535  
6 MVN{S}{cond} Rd, <op2> ; set Rd equal to the !value  
                           ; specified by op2
```

- Note que **MVN** inverte os bits do op2 e salva em Rd (equivalente ao complemento de 1);
- Já o **NEG** inverte os bits do op2, adiciona 1 e salva em Rd (equivalente ao complemento de 2).

Operações de Transferência

- E quando desejamos carregar um registrador com uma constante não suportada pelo **MOV**, o que fazer?

- Usar o **MOV** e **MOVT** (Instrução):

```
1 MOV R1, #0x5678 ;R1[31:16]:=0 R1[15:0]:=0x5678
2 MOVT R1, #0x1234 ;R1[31:16]:=0x1234 R1[15:0]
3 ; não afetada
```

- No Keil (Diretiva):

```
1 LDR R6, Pi ; Definição da constante fora da
2 ; execução do código
3 Pi DCD 314159
```

- Ou:

```
1 LDR R6, =314159
```

- Não confundir com o **LDR** de acesso à memória.

Operações de Transferência

- E quando desejamos carregar um registrador com uma constante não suportada pelo **MOV**, o que fazer?
 - Usar o **MOV** e **MOVT** (Instrução):

```
1 MOV R1, #0x5678 ;R1[31:16]:=0 R1[15:0]:=0x5678
2 MOVT R1, #0x1234 ;R1[31:16]:=0x1234 R1[15:0]
3 ; não afetada
```

- No Keil (Diretiva):

```
1 LDR R6, Pi ; Definição da constante fora da
2 ; execução do código
3 Pi DCD 314159
```

- Ou:

```
1 LDR R6, =314159
```

- **Não confundir com o LDR de acesso à memória.**

Exercício: Instruções de Memória/Transfer

- 1 Abra o arquivo no moodle como criar um projeto novo e, em seguida, faça um código que realize os seguintes passos e depois depure no Keil:

(Acrescente ao final do arquivo a instrução **NOP** (do inglês, *No Operation*) para que seja possível depurar o código inteiro)

- a) Salve no registrador R0 o valor 65 decimal;
- b) Salve no registrador R1 o valor 0x1B00.1B00;
- c) Salve no registrador R2 o valor 0x1234.5678;
- d) Guarde na posição de memória 0x2000.0040 o valor de R0;
- e) Guarde na posição de memória 0x2000.0044 o valor de R1;
- f) Guarde na posição de memória 0x2000.0048 o valor de R2;
- g) Guarde na posição de memória 0x2000.004C o número 0xF0001;
- h) Guarde na posição de memória 0x2000.0046 o byte 0xCD, sem sobrescrever os outros bytes da WORD;
- i) Leia o conteúdo da memória cuja posição 0x2000.0040 e guarde no R7;
- j) Leia o conteúdo da memória cuja posição 0x2000.0048 o guarde no R8;
- k) Copie para o R9 o conteúdo de R7.

Operações Lógicas

- Combinar, extrair ou testar uma informação;
- Operações unárias (uma entrada):
 - **Negação;**
 - **Complementar.**
- Operações Binárias (duas entradas):
 - **AND;**
 - **OR;**
 - **XOR.**

Operações Lógicas

- Algumas instruções lógicas:

| | | | | | | |
|---|------------|-----|----------|---------|-----------|------------------|
| 1 | AND | {S} | { cond } | { Rd, } | Rn, <op2> | ;Rd=Rn&op2 |
| 2 | ORR | {S} | { cond } | { Rd, } | Rn, <op2> | ;Rd=Rn op2 |
| 3 | EOR | {S} | { cond } | { Rd, } | Rn, <op2> | ;Rd=Rn^op2 |
| 4 | BIC | {S} | { cond } | { Rd, } | Rn, <op2> | ;Rd=Rn&(~op2) |
| 5 | ORN | {S} | { cond } | { Rd, } | Rn, <op2> | ;Rd=Rn (~op2) |

- Adicionar o sufixo 'S' para a condição N ou Z ser atualizada no resultado da operação.

Exercício: Operações Lógicas

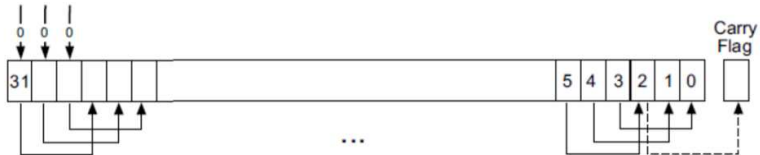
- 2) Faça um código que realize os seguintes passos e depois depure no Keil:

(Acrescente ao final do arquivo a instrução **NOP** (do inglês, *No Operation*) para que seja possível depurar o código inteiro)

- a) Realize a operação lógica AND do valor 0xF0 com o valor binário 01010101 e salve o resultado em R0. Utilize o sufixo 'S' para atualizar os bits;
- b) Realize a operação lógica AND do valor 11001100 binário com o valor binário 00110011 e salve o resultado em R1. Utilize o sufixo 'S' para atualizar os bits;
- c) Realize a operação lógica OR do valor 10000000 binário com o valor binário 00110111 e salve o resultado em R2. Utilize o sufixo 'S' para atualizar os bits;
- d) Realize a operação lógica BIC do valor 0xABCDABCD com o valor 0xFFFF0000 e salve o resultado em R3. Utilize o sufixo 'S' para atualizar os bits.

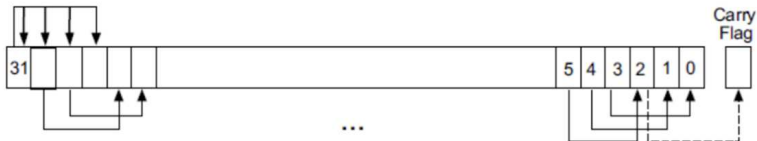
Operações de Deslocamento

- Tem dois parâmetros de entrada e uma saída;
- Deslocamento lógico para direita (LSR{S}):
 - Similar à divisão sem sinal por 2^n ;
 - Um zero é colocado na posição mais significativa.



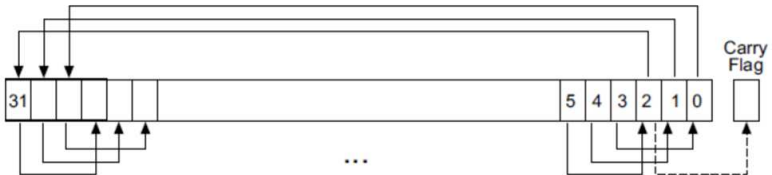
Operações de Deslocamento

- Deslocamento aritmético para direita (ASR{S}):
 - Similar à divisão com sinal por 2^n ;
 - O sinal é preservado.



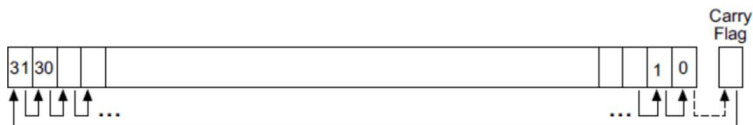
Operações de Deslocamento

- Rotação à direita ROR{S}:
 - Gira para a direita o valor dos bits dos registradores;
 - Não há rotação para a esquerda porque uma rotação para a esquerda de n equivale a uma rotação para a direita de $32-n$.



Operações de Deslocamento

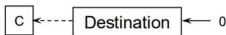
- Rotação à direita estendida $RRX\{S\}$:
 - Rotação de UM ÚNICO bit para a direita.



Operações de Deslocamento

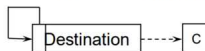
- Resumo:

LSL: Logical Shift Left



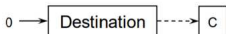
Multiplicação por uma potência de 2

ASR: Arithmetic Right Shift



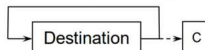
Divisão por uma potência de 2 preservando o sinal

LSR: Logical Shift Right



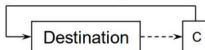
Divisão por uma potência de 2

ROR: Rotate Right



Rotação circular de n bits do LSB para o MSB

RRX: Rotate Right Extended



Rotação circular de 1 bit do flag C para o MSB

Operações de Deslocamento

```
1 LSR{S}{cond} Rd, Rm, Rs ; logical shift right Rd=Rm>>Rs  
2 ;(unsigned)  
3 LSR{S}{cond} Rd, Rm, #n ; logical shift right Rd=Rm>>n  
4 ; (unsigned)  
5 ASR{S}{cond} Rd, Rm, Rs ; arithmetic shift right  
6 ; Rd=Rm>>Rs (signed)  
7 ASR{S}{cond} Rd, Rm, #n ; arithmetic shift right  
8 ; Rd=Rm>>n (signed)  
9 LSL{S}{cond} Rd, Rm, Rs ; shift left Rd=Rm<<Rs (signed,  
10 ; unsigned)  
11 LSL{S}{cond} Rd, Rm, #n ; shift left Rd=Rm<<n (signed,  
12 ; unsigned)  
13 ROR{S}{cond} Rd, Rm, Rs ; rotate right  
14 ROR{S}{cond} Rd, Rm, #n ; rotate right  
15 RRX{S}{cond} Rd, Rm ; rotate right 1 bit with extension
```

Exercício: Operações de Deslocamento

- 3 Faça um código que realize os seguintes passos e depois depure no Keil verificando, na simulação, os valores dos registradores antes e depois:
- a) Realize o deslocamento lógico em 5 bits do número 701 para a direita com o *flag* 'S';
 - b) Realize o deslocamento lógico em 4 bits do número -32067 para a direita com o *flag* 'S' (Use o **MOV** para o número positivo e depois **NEG** para negativar com complemento de 2);
 - c) Realize o deslocamento aritmético em 3 bits do número 701 para a direita com o *flag* 'S';
 - d) Realize o deslocamento aritmético em 5 bits do número -32067 para a direita com o *flag* 'S';
 - e) Realize o deslocamento lógico em 8 bits do número 255 para a esquerda com o *flag* 'S';
 - f) Realize o deslocamento lógico em 18 bits do número -58982 para a esquerda com o *flag* 'S';
 - g) Rotacionar em 10 bits o número 0xFABC1234;
 - h) Rotacionar em 2 bits com o *carry* o número 0x00004321; (Realize duas vezes).

Operações Aritméticas

- Tipos de operações aritméticas:
 - Soma, subtração, multiplicação, divisão e comparação.
- Executados por meio de hardware digital;
- Carry:
 - Soma:
 - 0: soma coube nos 32 bits;
 - 1: soma não coube nos 32 bits.
 - Subtração:
 - 0: resultado incorreto;
 - 1: resultado correto.
- Overflow:
 - Operações com sinal;
 - O bit V é setado quando há uma passagem entre 0x8000.0000 e 0x7FFF.FFFF.

Operações Aritméticas

- Soma e Subtração:

- Nas operações abaixo, quando *Rd* não é especificado, o resultado é colocado em *Rn*.

| | | |
|---|--------------------------------------|--------------------|
| 1 | ADD {S}{cond} {Rd}, Rn, <op2> | ;Rd = Rn + op2 |
| 2 | ADD {cond} {Rd}, Rn, #im12 | ;Rd = Rn + im12 |
| 3 | ADC {S}{cond} {Rd}, Rn, <op2> | ;Rd = Rn + op2 + C |
| 4 | SUB {S}{cond} {Rd}, Rn, <op2> | ;Rd = Rn - op2 |
| 5 | SUB {cond} {Rd}, Rn, #im12 | ;Rd = Rn - im12 |
| 6 | RSB {S}{cond} {Rd}, Rn, <op2> | ;Rd = op2 - Rn |
| 7 | RSB {cond} {Rd}, Rn, #im12 | ;Rd = im12 - Rn |
| 8 | CMP {cond} Rn, <op2> | ;Rn - op2 |
| 9 | CMN {cond} Rn, <op2> | ;Rn - (-op2) |

- **CMP** e **CMN** apenas criam condições de comparação para *if-then* e *loops*.

Operações Aritméticas

- Multiplicação e divisão (Resultado em 32 bits):
 - Nas operações abaixo, quando Rd não é especificado, o resultado é colocado em Rn.

| | | |
|---|-----------------------------------|-----------------------------|
| 1 | MUL {S}{cond} {Rd,} Rn, Rm | <i>;Rd = Rn * Rm</i> |
| 2 | MLA {cond} Rd, Rn, Rm, Ra | <i>;Rd = Ra + Rn*Rm</i> |
| 3 | MLS {cond} Rd, Rn, Rm, Ra | <i>;Rd = Ra - Rn*Rm</i> |
| 4 | UDIV {cond} {Rd,} Rn, Rm | <i>;Rd = Rn/Rm unsigned</i> |
| 5 | SDIV {cond} {Rd,} Rn, Rm | <i>;Rd = Rn/Rm signed</i> |

- Multiplicação (Resultado em 64 bits):

| | | |
|---|--|-------------------------|
| 1 | UMULL {cond} RdLo, RdHi, Rn, Rm | <i>;Rd = Rn * Rm</i> |
| 2 | SMULL {cond} RdLo, RdHi, Rn, Rm | <i>;Rd = Rn * Rm</i> |
| 3 | UMLAL {cond} RdLo, RdHi, Rn, Rm | <i>;Rd = Rd + Rn*Rm</i> |
| 4 | SMLAL {cond} RdLo, RdHi, Rn, Rm | <i>;Rd = Rd + Rn*Rm</i> |

Exercício: Operações Aritméticas

4. Faça um código que realize os seguintes passos e depois depure no Keil verificando, na simulação, os valores dos registradores antes e depois:
- (Acrescente ao final do arquivo a instrução **NOP** (do inglês, *No Operation*) para que seja possível depurar o código inteiro)
- a) Adicione os números 101 e 253 atualizando os *flags*;
 - b) Adicione os números 1500 e 40543 sem atualizar os *flags*;
 - c) Subtraia o número 340 pelo número 123 atualizando os *flags*;
 - d) Subtraia o número 1000 pelo número 2000 atualizando os *flags*;
 - e) Multiplique o número 54378 por 4; (Essa operação é semelhante a qual?)
 - f) Multiplique com o resultado em 64 bits os números 0x11223344 e 0x44332211
 - g) Divida o número 0xFFFF7560 por 1000 com sinal;
 - h) Divida o número 0xFFFF7560 por 1000 sem sinal;

Bloco If-Then

- Um bloco IT consiste de uma a quatro instruções condicionais, as condições devem ser coerentes;
- Sintaxe:

```
1 IT {x{y{z}}} {cond}
```

- $x \rightarrow$ especifica a condição para a segunda instrução no bloco (T: executa se verdadeiro; E: executa caso contrário);
 - $y \rightarrow$ especifica a condição para a terceira instrução no bloco (T: executa se verdadeiro; E: executa caso contrário);
 - $z \rightarrow$ especifica a condição para a quarta instrução no bloco (T: executa se verdadeiro; E: executa caso contrário).
- Exemplo:

```
1 ITTE NE           ; começo do bloco
2   ANDNE   R0,R0,R1
3   ADDSNE  R2,R2,#1
4   MOVEQ   R2,R3
```

Códigos de Condição

| Sufixo | Descrição | Flags | Sufixo | Descrição | Flags |
|------------------------|---------------------------------|-------|-----------|--------------------------------------|--------------|
| EQ | <i>Equal</i> | Z=1 | HI | <i>Higher, unsigned</i> | C=1 && Z=0 |
| NE | <i>Not Equal</i> | Z=0 | LS | <i>Lower or same, unsigned</i> | C=0 Z=1 |
| CS ou HS | <i>Higher or same, unsigned</i> | C=1 | GE | <i>Greater than or equal, signed</i> | N = V |
| CC ou LO | <i>Lower, unsigned</i> | C=0 | LT | <i>Less than, signed</i> | N!=V |
| MI | <i>Negative</i> | N=1 | GT | <i>Greater than, signed</i> | Z=0 && N = V |
| PL | <i>Positive or zero</i> | N=0 | LE | <i>Less than or equal, signed</i> | Z=1 && N!=V |
| VS | <i>Overflow</i> | V=1 | AL | <i>Always. Default</i> | |
| VC | <i>No overflow</i> | V=0 | | | |

Bloco If-Then

- A instrução de salto condicional B{cond} label é a única que **não precisa** estar em bloco IT;
- As instruções IT, CBZ, CBNZ, CPSIE, CPSID **não podem** estar em bloco IT;
- Uma instrução que altera o PC, só pode estar em um bloco IT se for a última instrução do bloco.

Exercício: Bloco *If-Then*

- 5) Faça um código que realize os seguintes passos e depois depure no Keil verificando, na simulação, os valores dos registradores antes e depois:

(Acrescente ao final do arquivo a instrução **NOP** (do inglês, *No Operation*) para que seja possível depurar o código inteiro)

- a) Mova o valor 10 para o registrador R0;
- b) Teste se o registrador é maior ou igual que 9;
- c) Crie um bloco com *If-Then* com 3 execuções condicionais:
 - 1 Se sim, salve o número 50 no R1;
 - 2 Se sim, adicione 32 com o R1 e salve o resultado em R2;
 - 3 Se não, salve o número 75 no R3.
- d) Agora verifique se o registrador é maior ou igual a 11 e execute novamente o passo (c)

Pilha (*Stack*)

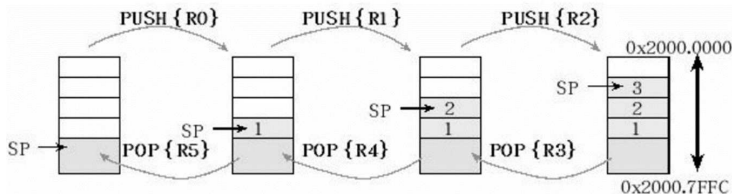
- Região da RAM para armazenamento temporário;
- *Last-in-First-out* (LIFO);
- Opera sempre em 32 bits;
- Para salvar um registrador na pilha **PUSH**;
- Para restaurar um registrador da pilha **POP**;
- Por padrão, SP (R13) aponta para o topo da pilha e é decrementado de 4 bytes a cada **PUSH** e incrementado de 4 bytes a cada **POP** (*Full Descending Stack*).

Pilha (*Stack*)

- Comandos:

```
1 PUSH <reglist>
2 POP <reglist>
```

- <reglist> → Lista de registradores separada entre ',' entre chaves.
Ex: {R0, R1, R4-R9};
- Exemplo de operação:



Exercício: Pilha (*Stack*)

- 6 Faça um código que realize os seguintes passos e depois depure no Keil verificando, na simulação, os valores dos registradores antes e depois:

(Acrescente ao final do arquivo a instrução **NOP** (do inglês, *No Operation*) para que seja possível depurar o código inteiro)

- a) Mova o valor 10 para o registrador R0;
- b) Mova o valor 0xFF11CC22 para o registrador R1;
- c) Mova o valor 1234 para o registrador R2;
- d) Mova o valor 0x300 para o registrador R3;
- e) Empurre para a pilha o R0;
- f) Empurre para a pilha os R1, R2 e R3;
- g) Visualize a pilha na memória (o topo da pilha está em 0x2000.0400);
- h) Mova o valor 60 para o registrador R1;
- i) Mova o valor 0x1234 para o registrador R2;
- j) Desempilhe corretamente os valores para os registradores R0, R1, R2 e R3.

Instruções de Salto

- Instruções para interromper o fluxo ou chamar subrotinas:

```
1 B{cond} label    ;branch to label
2 BX{cond} Rm      ;branch indirect to location
                   specified by Rm
3 BL{cond} label    ;branch to subroutine at label
```

- Instrução B*{cond} é a única que não precisa estar dentro de um bloco *if-then*;
- Outras instruções de salto **condicional** (não muda *flags* de condição, compara com zero e só pode pular **para frente** de 4 a 130 bytes):

```
1 CBZ   Rn, <label>    ;compare and branch if zero
2 CBNZ  Rn, <label>    ;compare and branch if non
                   zero
```


Exercício: Instruções de Salto

- 7 Faça um código que realize os seguintes passos e depois depure no Keil verificando, na simulação, os valores dos registradores antes e depois:
- a) Mover para o R0 o valor 10;
 - b) Somar R0 com 5 e colocar o resultado em R0;
 - c) Enquanto a resposta não for 50 somar mais 5;
 - d) Quando a resposta for 50 chamar uma função que:
 - 1 Copia o R0 para R1;
 - 2 Verifica se R1 é menor que 50;
 - 3 Se for menor que 50 incrementa, caso contrário modifica para -50.
 - e) Depois que retornar da função coloque uma instrução NOP;
 - f) Acrescente uma instrução para ficar travado na última linha de execução.

Diretivas do *Assembler* do Keil

- Auxiliam o processo de montagem (*assembly*);
- Não fazem parte do conjunto de instruções:
 - **CODE**: espaço para instruções de máquina (ROM);
 - **DATA**: espaço para variáveis globais (RAM);
 - **STACK**: espaço para pilha (RAM);
 - **ALIGN=n**: começa a área alinhada para 2^n bytes;
 - **|.text|**: seções de código produzidas pelo compilador C. Faz o código *assembly* poder ser chamado do C;
 - **NOINIT**: faz uma área da RAM ser não inicializada.

Diretivas do Assembler do Keil

```
1 AREA RESET, CODE, READONLY      ;reset vectors in flash ROM
2 AREA DATA                      ;places objects in data memory
   (RAM)
3 AREA |.text|, CODE, READONLY, ALIGN=2      ;code in
   flash ROM
4 AREA STACK, NOINIT, READWRITE, ALIGN=3      ;stack area
```

- Para linkar variáveis e funções entre dois arquivos:
 - **EXPORT** ou **GLOBAL**: no arquivo onde o objeto está definido;
 - **IMPORT** ou **EXTERN**: no arquivo que está tentando acessar.

```
1 IMPORT name                    ;imports function "name" from
   other file
2 EXPORT name                   ;exports public function "name"
   for use
3                               ;elsewhere
```

Diretivas do Assembler do Keil

- **ALIGN**: garante que o próximo objeto será alinhado propriamente. Recomendável colocar ao final do arquivo:

```
1 ALIGN      ;skips 0 to 3 bytes to make next word  
    aligned  
2 ALIGN 2    ;skips 0 or 1 byte to make next halfword  
    aligned  
3 ALIGN 4    ;skips 0 to 3 bytes to make next word  
    aligned
```

- **THUMB**: colocada no topo do arquivo para especificar que o código será gerado com instruções Thumb:

```
1 THUMB
```

- **END**: Diretiva no fim de cada arquivo.

Diretivas do Assembler do Keil

- Adicionando variáveis e constantes:

```

1 DCB expr{,expr}           ;places 8-bit byte(s) into
    memory
2 DCW expr{,expr}           ;places 16-bit halfword(s)
    into memory
3 DCD expr{,expr}           ;places 32-bit word(s) into
    memory
4 SPACE size                 ;reserves size bytes,
    uninitialized
  
```

- Como fazer um vetor de 5 bytes?

```

1 ;Declarar na região da RAM (abaixo do AREA DATA)
2 nome_do_vetor    SPACE    5    ;(5 bytes)
3
4 ;Na região do código quando for utilizar
5 LDR R0, =nome_do_vetor      ;(região inicial)
6 LDRB R1, [R0]               ;(vetor de bytes)
  
```

Diretivas do Assembler do Keil

- DCB, DCW e DCD podem ser colocados na área de dados (alocando um espaço no qual o nome da variável retorna o endereço constante dele):

```
1 ;Declarar na região da RAM (abaixo do AREA DATA)
2 variavel DCB 0x00 ; cria a variável (0x00 ignorado)
3 ;Na região do código quando for utilizar
4 LDR R1, =variavel ;pega o endereço na RAM
5 STR R2, [R1] ;salva R2 na variável
```

- Ou na memória de programa (criando constantes):

```
1 ;Na região do código quando for utilizar
2 LDR R5, =STR_1 ;pega o end. do prim. elemento
3 LDRB R6, [R5], #1 ;pega o valor do prim. elemento
4 ;Após o código, para não desalinhar
5 STR_1 DCB "UTFPR",0
6 VETOR_1 DCB 0xFF, 0xAA, 0x12, 0x14,0
```

Diretivas do *Assembler* do Keil

- **EQU**: define um nome simbólico à uma constante numérica:

```
1 GPIO_PORTD_DATA_R EQU 0x400073FC  
2 GPIO_PORTD_DIR_R EQU 0x40007400  
3 GPIO_PORTD_DEN_R EQU 0x4000751C
```

Diretivas do *Assembler* do Keil

- **EQU**: define um nome simbólico à uma constante numérica:

```
1 GPIO_PORTD_DATA_R EQU 0x400073FC
2 GPIO_PORTD_DIR_R   EQU 0x40007400
3 GPIO_PORTD_DEN_R   EQU 0x4000751C
```



Exercício: Diretivas do *Assembler* do Keil

- 8) Faça um código que realize os seguintes passos e depois depure no Keil verificando, na simulação, os valores dos registradores antes e depois:
 - a) Ler de uma posição da memória RAM um número e calcular o fatorial. Armazenar o resultado em R0.

Dúvidas?