

Sistemas Microcontrolados

Conjunto de Instruções
Assembly

Prof. Guilherme Peron

Linguagem de Máquina

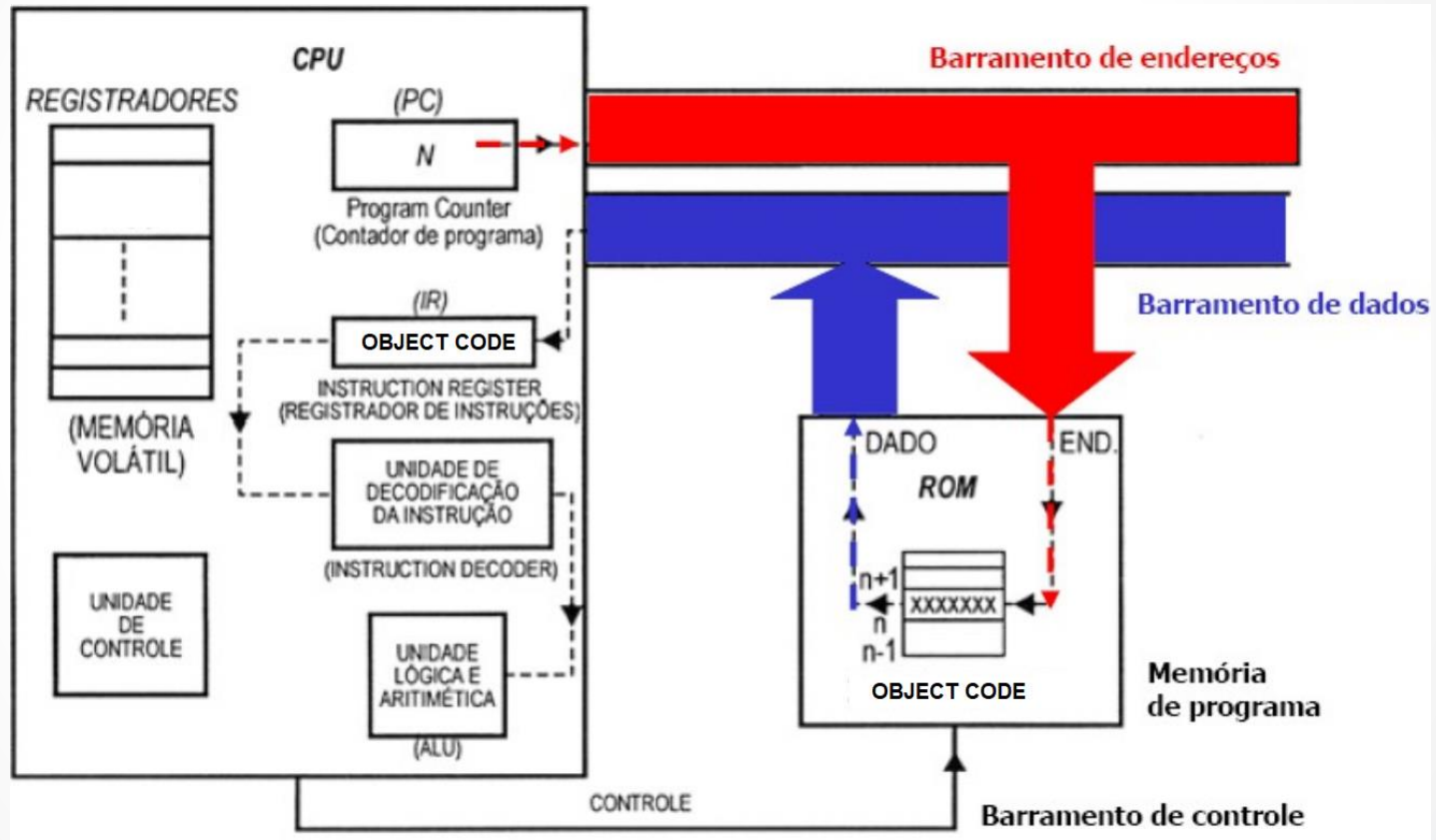
- O que é linguagem de máquina?

Linguagem de Máquina

- 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.



Linguagem de Máquina

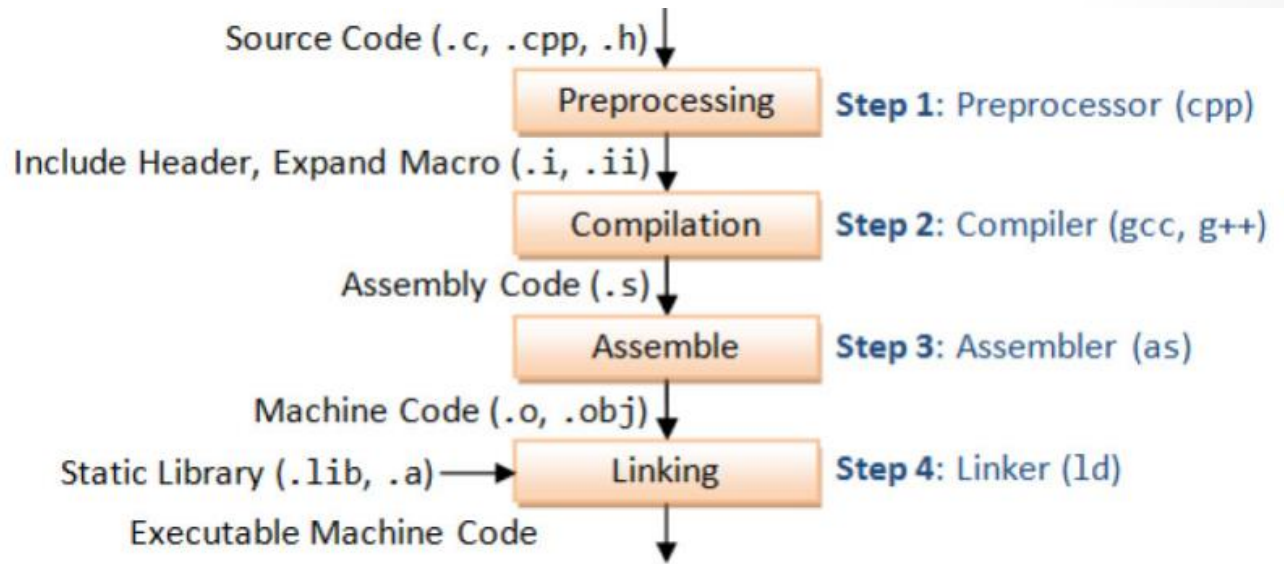


Linguagem *Assembly*

- Para facilitar a vida do programador, criou-se a Linguagem *Assembly*, que possui o mesmo conjunto de instruções, porém utiliza símbolos (**mnemônicos**) ou **opcodes** no lugar dos números.
- A conversão da linguagem **assembly** para a linguagem de máquina é feita pelo **assembler** (montador). NUNCA CONFUNDIR!
- 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
- A conversão destas linguagens para a linguagem de máquina é feita pelo compilador.



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)

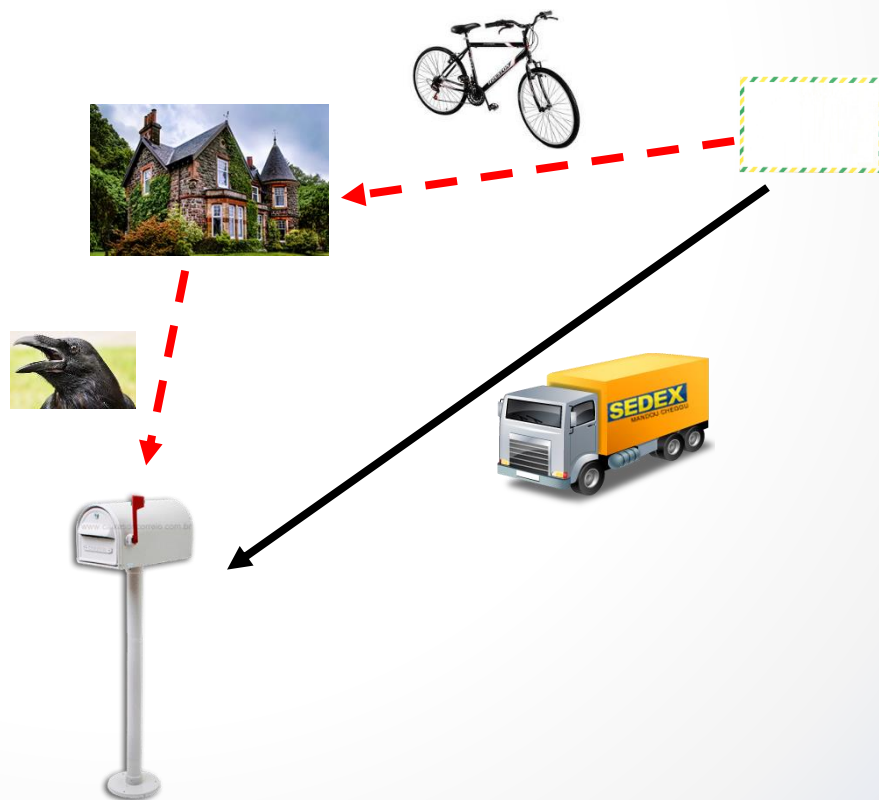
Exemplo:

Função que recebe R0 como entrada:

```
Func  MOV    R1, #100      ; R1=100;
      MUL    R0, R0, R1    ; R0=100*input;
      ADD    R0, #10       ; R1=100*input+10;
      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
- Definido por uma *hashtag* ('#') antes do operando.
- Exemplos:



MOV R0, #25

```
MOV    R0, #25          ;move a constante decimal 25 para o reg R0
MOV    R1, #0x2F        ;move a constante hexa 2Fh para o reg R1
MOV    R2, #2_1101      ;move a constante binária 00001101 para R2
```

Endereçamento por Registrador

- Algumas instruções podem operar dados com registradores do microprocessador.
- Registrador com Imediato

- Exemplos:



ADD R1, R2, #18 ; R1 <= R2 + 18

AND R0, R1, #0x0F ; R0 <= R1 & 0x0F

MUL R0, R2, #8 ; R0 <= R2 * 8

Endereçamento por Registrador

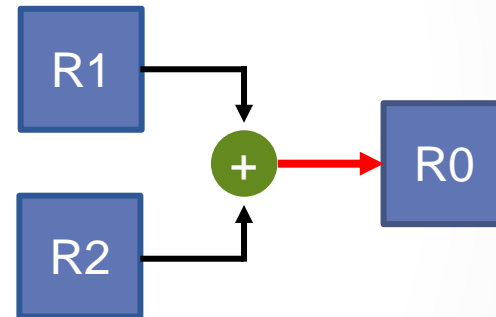
- Registrador com Registrador

- Exemplos:

MOV R1, R2



ADD R1, R2



ADD R0, R1, R2 ; R0 <= R1 + R2

ADD R3, R4 ; R3 <= R3 + R4

MOV R1, R3 ; R1 <= R3

Endereçamento por Registrador

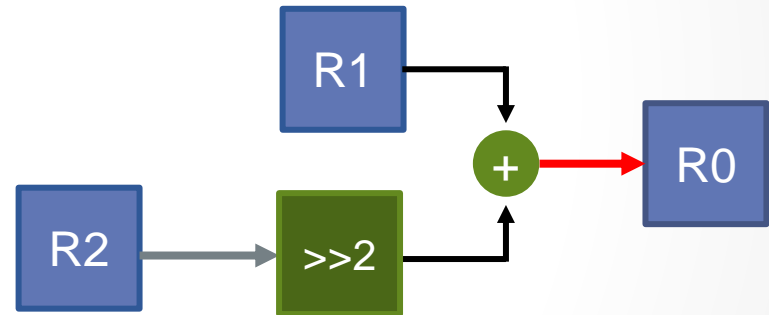
- Registrador escalado

- Exemplos:

MOV R0, R1, LSL #3



ADD R0, R1, R2, LSR #2



MOV R0, R1, LSL #3

;R0 <= (R1 << 3)

ADD R0, R1, R2, LSR #2

;R0 <= R1 + (R2 >> 2)

MOV R1, R3, ASR #7

;R1 <= R3 / 128 (signed)

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;

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

Endereçamento Relativo ao PC

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

Endereçamento Relativo ao PC

- Exemplos:

```
B    label    ;pula para label
```

```
BL   subrotina ;chama subrotina e salva PC no R14 (LR)
```

```
BX   R14      ;return ou  MOV PC, R14
```

```
LDR   R1, =Count ;R1 aponta para Count
```

```
LDR   R0, [R1]    ;R0 <= valor apontado por R1
```

Tipos de Operandos

Operandos

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

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 que seta os bits de condição
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	Registrador Rm com deslocamento lógico para esquerda definido por Rs
Rm, LSR Rs	Registrador Rm com deslocamento lógico para direita definido por Rs
Rm, ASR Rs	Registrador Rm com deslocamento aritmético para direita definido por Rs
Rm, ROR Rs	Registrador 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
 - **LDR** lê dados da memória;
 - **STR** escreve dados na memória;
 - Instruções de processamento de dados não acessam a memória.
 - Arquitetura **LOAD-STORE**
- Para operações com dados em memória:
 - Leitura da memória em registrador;
 - Operação;
 - Escrita em memória.

Operações de Acesso à Memória

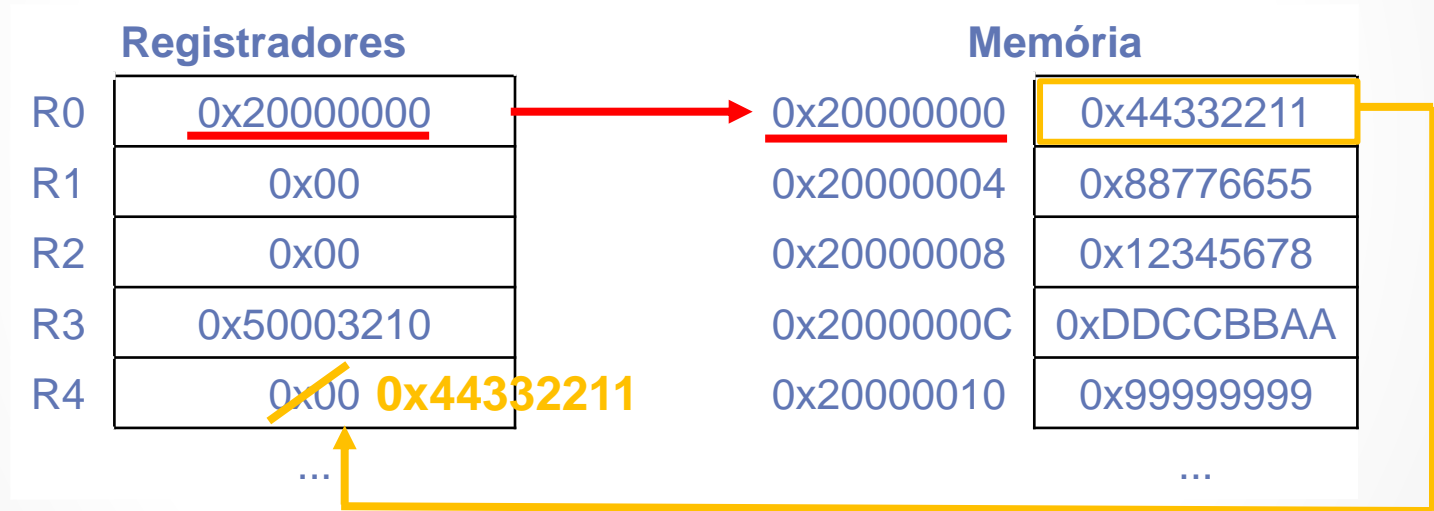
- Para acessar a memória SEMPRE estabelecer um **registrador ponteiro** (ou base) para o objeto;
- Exemplos:
Estado Inicial

Registradores		Memória	
R0	0x20000000	0x20000000	0x44332211
R1	0x00	0x20000004	0x88776655
R2	0x00	0x20000008	0x12345678
R3	0x50003210	0x2000000C	0xDDCCBBAA
R4	0x00	0x20000010	0x99999999
...		...	

Operações de Acesso à Memória

- Exemplo:

LDR R4, [R0]

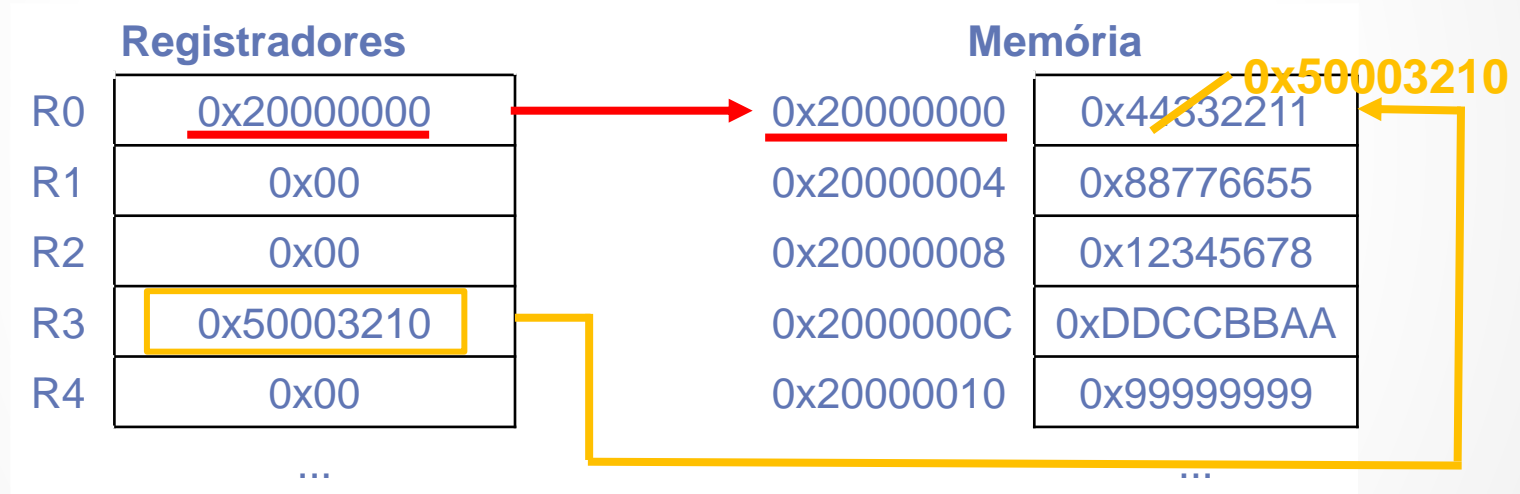


- O primeiro operando é por registrador e o segundo é indexado ([]).

Operações de Acesso à Memória

- Exemplo:

STR R3, [R0]



- O primeiro operando é por registrador e o segundo 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

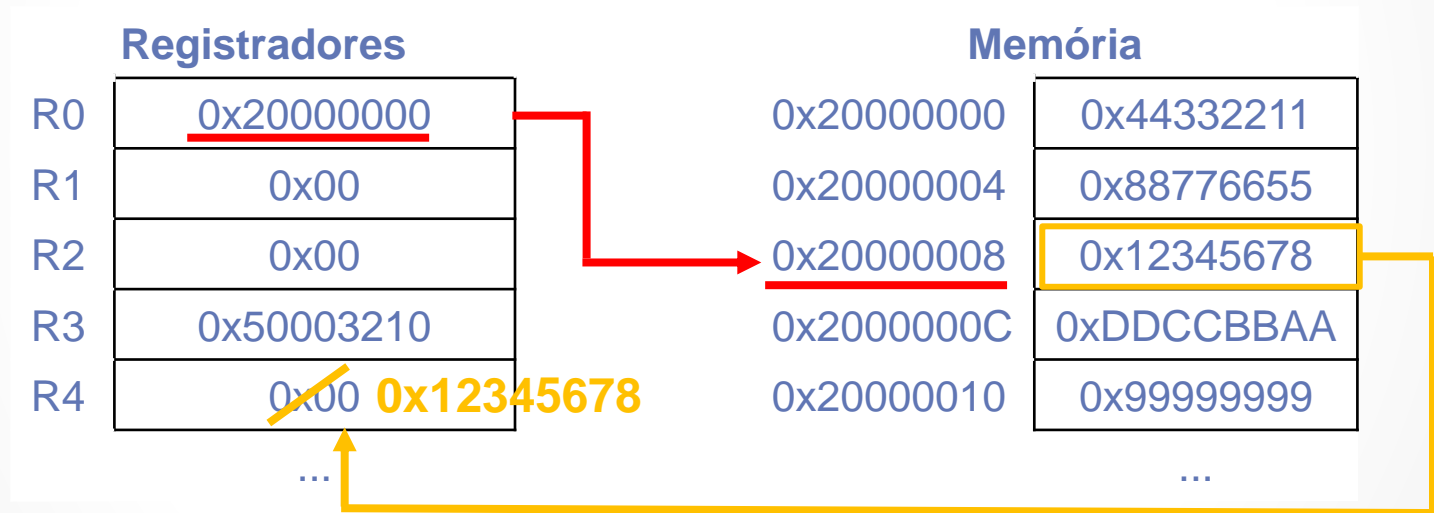
- Com offset:
 - O endereço é incrementado ANTES da operação, mas o valor do registrador base NÃO é alterado.
 - **[XX]** → Conteúdo apontado por XX
 - Exemplos:

```
LDR    R0, [R1]           ;R0 <= [R1]  R0 recebe o dado apontado por R1
LDR    R0, [R1, #8]       ;R0 <= [R1+8] R0 recebe o dado apontado pelo
                           ;R1 + 8
STR    R2, [R0, #4]       ;[R0+4] <= R2  guarda o dado R2 endereço
                           ; apontado por R0 + 4
```

Com Offset

- Exemplo:

LDR R4, [R0, #8]



Tipos de Acesso à Memória

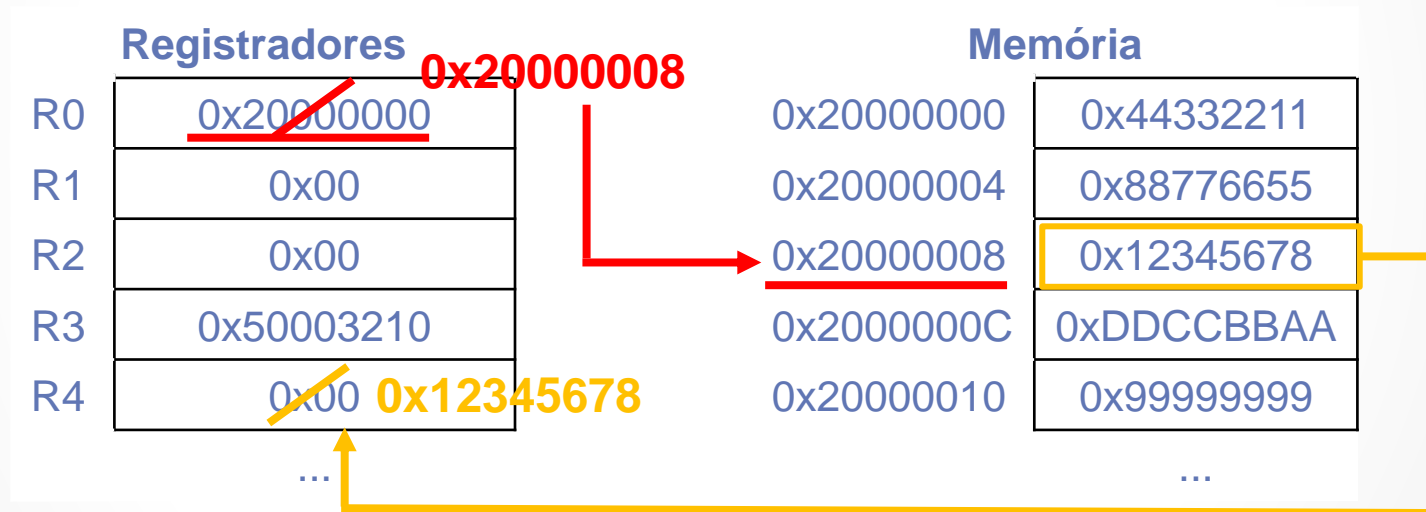
- Pré-Indexado → '!'
 - O endereço é incrementado ANTES da operação, mas o valor do registrador base é salvo.
 - Exemplos:

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

Pré-Indexado

- Exemplo:

LDR R4, [R0, #8]!



Tipos de Acesso à Memória

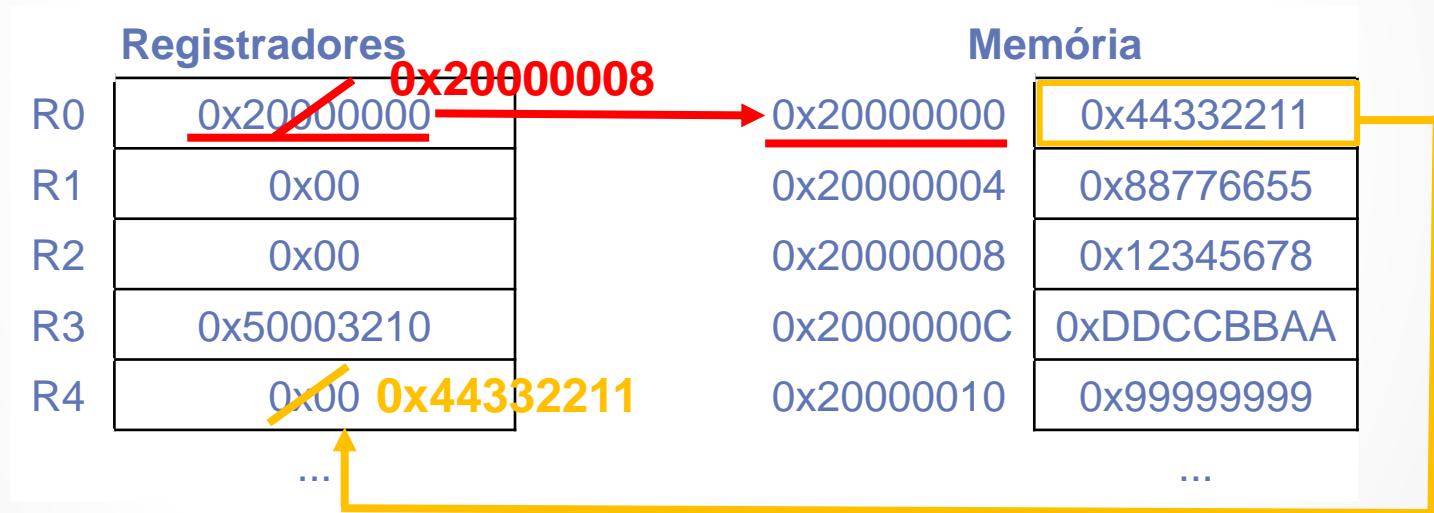
- Pós-Indexado:
 - O endereço é incrementado após a operação e o valor do registrador base é salvo.
 - Exemplos:

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

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 instruções load e store

```
LDR{type}{cond} Rd, [Rn] ; load memory at [Rn] to Rd
STR{type}{cond} Rt, [Rn] ; store Rt to memory at [Rn]
LDR{type}{cond} Rd, [Rn, #n] ; load memory at [Rn+n] to Rd
STR{type}{cond} Rt, [Rn, #n] ; store Rt to memory [Rn+n]
LDR{type}{cond} Rd, [Rn, #n]! ; load memory at [Rn+n] to Rd;
                                ; Rn := Rn + n;
STR{type}{cond} Rt, [Rn, #n]! ; store Rt to memory [Rn+n]
                                ; Rn := Rn + n;
LDR{type}{cond} Rd, [Rn], #n ; load memory at [Rn] to Rd;
                                ; Rn := Rn + n;
STR{type}{cond} Rt, [Rn], #n ; store Rt to memory [Rn]
                                ; 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, os bits mais significantes são preenchidos com 0.
- 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

`MOV{S}{cond} Rd, <op2> ; set Rd equal to the value specified
by op2`

`MOV{cond} Rd, #im16 ; set Rd equal to im16, im16 is 0 to 65535`

`MVN{S}{cond} Rd, <op2> ; set Rd equal to the value specified
by op2`

Operações de Transferência

- Quando desejar carregar um registrador com uma constante não suportada pelo MOV, o que fazer?

Operações de Transferência

- Quando desejar carregar um registrador com uma constante não suportada pelo MOV, o que fazer?
 - USAR o MOV e MOVT (Instrução):

```
MOV  R1, #0x5678 ;R1[31:16]:=0 R1[15:0]:=0x5678
MOVT R1, #0x1234 ;R1[31:16]:=0x1234 R1[15:0] não afetada
```
 - No Keil (Diretiva):

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

ou

```
LDR R6, =314159
```

Não confundir com o LDR de acesso à memória

Instruções de Memória/Transfer

- **Exercício** (Abra o arquivo no moodle como criar um projeto novo):
 1. Faça um código que realize os seguintes passos e depois depure no Keil:

(Acréscetar ao final do arquivo a instrução **NOP** para conseguir depurar o código inteiro. Esta instrução significa **No Operation.**)

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

`AND{S}{cond} {Rd,} Rn, <op2> ; Rd=Rn&op2`

`ORR{S}{cond} {Rd,} Rn, <op2> ; Rd=Rn|op2`

`EOR{S}{cond} {Rd,} Rn, <op2> ; Rd=Rn^op2`

`BIC{S}{cond} {Rd,} Rn, <op2> ; Rd=Rn&(~op2)`

`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.

Operações Lógicas

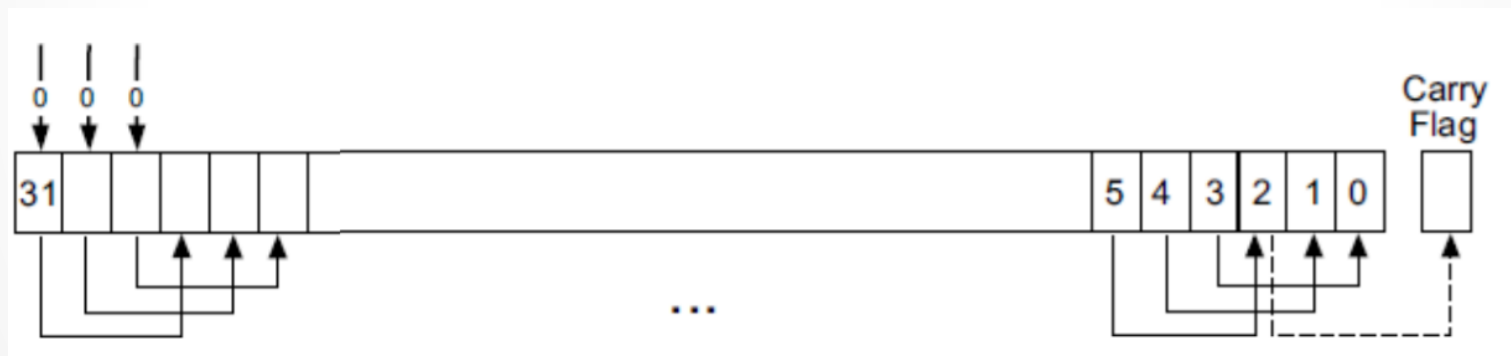
- **Exercício** (Abra o arquivo no moodle como criar um projeto novo):
 2. Faça um código que realize os seguintes passos e depois depure no Keil:

(Acrescentar ao final do arquivo a instrução **NOP** para conseguir depurar o código inteiro. Esta instrução significa **No Operation**.)

 - a) Realizar a operação lógica **AND** do valor 0xF0 com o valor binário 01010101 e salvar o resultado em R0. Utilizar o sufixo 'S' para atualizar os flags.
 - b) Realizar a operação lógica **AND** do valor 11001100 binário com o valor binário 00110011 e salvar o resultado em R1. Utilizar o sufixo 'S' para atualizar os flags.
 - c) Realizar a operação lógica **OR** do valor 10000000 binário com o valor binário 00110111 e salvar o resultado em R2. Utilizar o sufixo 'S' para atualizar os flags.
 - d) Realizar a operação lógica **AND** do valor 0xABCDABCD com o valor 0xFFFF0000 (sem usar LDR) e salvar o resultado em R3. Utilizar o sufixo 'S' para atualizar os flags. Utilizar a instrução **BIC**.

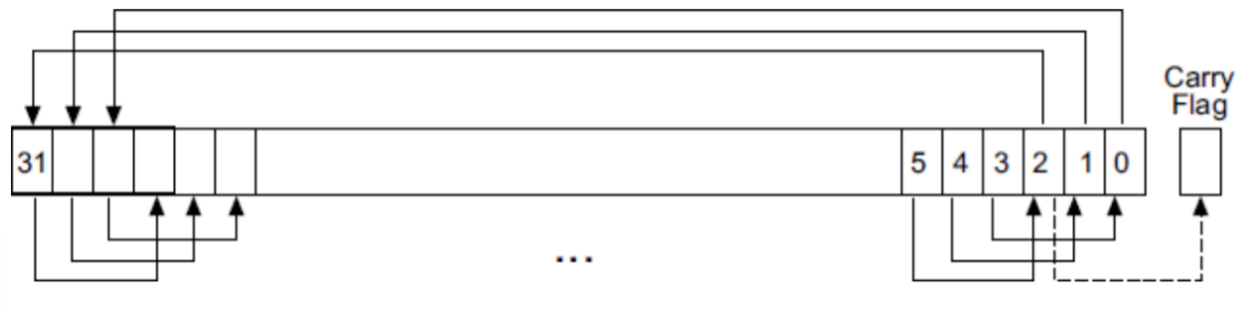
Operações de Deslocamento

- 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.



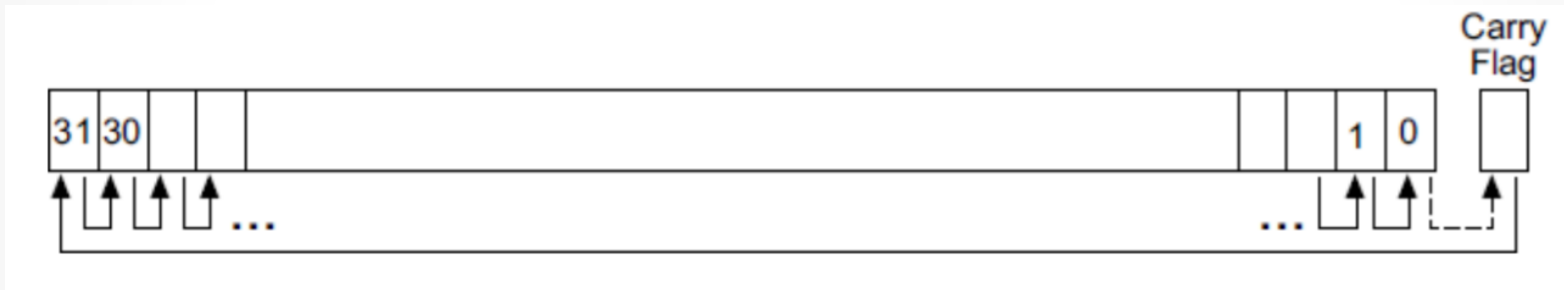
Operações de Deslocamento

- 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

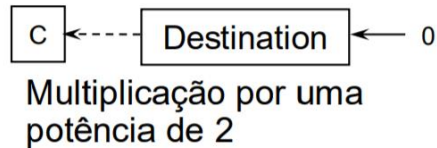
- Rotação à direita estendida RRX{S}:
 - Rotação de UM ÚNICO bit para a direita



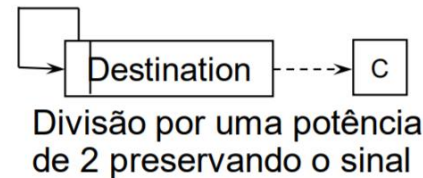
Operações de Deslocamento

- Resumo:

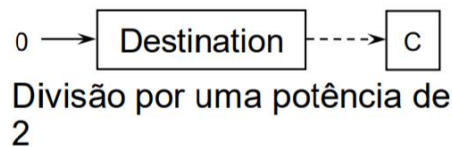
LSL: Logical Shift Left



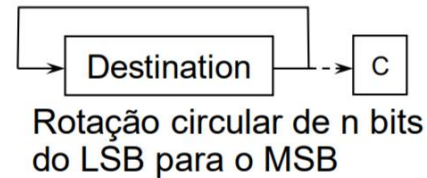
ASR: Arithmetic Right Shift



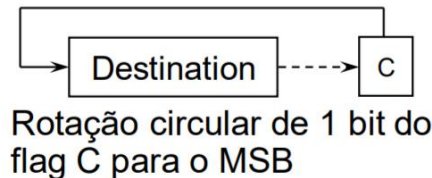
LSR: Logical Shift Right



ROR: Rotate Right



RRX: Rotate Right Extended



Operações de Deslocamento

```
LSR{S}{cond} Rd, Rm, Rs ; logical shift right Rd=Rm>>Rs  
                        ; (unsigned)  
LSR{S}{cond} Rd, Rm, #n ; logical shift right Rd=Rm>>n  
                        ; (unsigned)  
ASR{S}{cond} Rd, Rm, Rs ; arithmetic shift right Rd=Rm>>Rs  
                        ; (signed)  
ASR{S}{cond} Rd, Rm, #n ; arithmetic shift right Rd=Rm>>n  
                        ; (signed)  
LSL{S}{cond} Rd, Rm, Rs ; shift left Rd=Rm<<Rs (signed,  
                        ; unsigned)  
LSL{S}{cond} Rd, Rm, #n ; shift left Rd=Rm<<n (signed,  
                        ; unsigned)  
ROR{S}{cond} Rd, Rm, Rs ; rotate right  
ROR{S}{cond} Rd, Rm, #n ; rotate right  
RRX{S}{cond} Rd, Rm ; rotate right 1 bit with extension
```

Operações de Deslocamento

- Exercício

3. Faça um código que realize os seguintes passos e depois depure no Keil: (Acrescentar ao final do arquivo a instrução **NOP**)

Verifique na simulação, os valores dos registradores antes e depois.

- a) Realizar o deslocamento lógico em 5 bits do número 701 para a direita com o flag 'S';
- b) Realizar o deslocamento lógico em 4 bits do número -32067 para a direita com o flag 'S'; (Usar o **MOV** para o número positivo e depois **NEG** para negativar)
- c) Realizar o deslocamento aritmético em 3 bits do número 701 para a direita com o flag 'S';
- d) Realizar o deslocamento aritmético em 5 bits do número -32067 para a direita com o flag 'S';
- e) Realizar o deslocamento lógico em 8 bits do número 255 para a esquerda com o flag 'S';
- f) Realizar 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; (Realizar 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
 - 1: resultado positivo ou zero
 - 0: resultado negativo
- *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:

ADD{S}{cond} {Rd,} Rn, <op2>	;Rd = Rn + op2
ADD{cond} {Rd,} Rn, #im12	;Rd = Rn + im12
ADC{S}{cond} {Rd,} Rn, <op2>	;Rd = Rn + op2 + C
SUB{S}{cond} {Rd,} Rn, <op2>	;Rd = Rn - op2
SUB{cond} {Rd,} Rn, #im12	;Rd = Rn - im12
RSB{S}{cond} {Rd,} Rn, <op2>	;Rd = op2 - Rn
RSB{cond} {Rd,} Rn, #im12	;Rd = im12 - Rn
CMP{cond} Rn, <op2>	;Rn - op2
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:

MUL{S}{cond} {Rd,} Rn, Rm	;Rd = Rn * Rm
MLA{cond} Rd, Rn, Rm, Ra	;Rd = Ra + Rn*Rm
MLS{cond} Rd, Rn, Rm, Ra	;Rd = Ra - Rn*Rm
UDIV{cond} {Rd,} Rn, Rm	;Rd = Rn/Rm unsigned
SDIV{cond} {Rd,} Rn, Rm	;Rd = Rn/Rm signed

- Multiplicação (Resultado em 64 bits)

UMULL{cond} RdLo, RdHi, Rn, Rm	;Rd = Rn * Rm
SMULL{cond} RdLo, RdHi, Rn, Rm	;Rd = Rn * Rm
UMLAL{cond} RdLo, RdHi, Rn, Rm	;Rd = Rd + Rn*Rm
SMLAL{cond} RdLo, RdHi, Rn, Rm	;Rd = Rd + Rn*Rm

Operações Aritméticas

- Exercícios

4. Faça um código que realize os seguintes passos e depois depure no Keil: (Acrescentar ao final do arquivo a instrução **NOP**)

Verifique na simulação, os valores dos registradores antes e depois.

- a) Adicionar os números 101 e 253 atualizando os *flags*;
- b) Adicionar os números 1500 e 40543 sem atualizar os *flags*;
- c) Subtrair o número 340 pelo número 123 atualizando os *flags*;
- d) Subtrair o número 1000 pelo número 2000 atualizando os *flags*;
- e) Multiplicar o número 54378 por 4; (Essa operação é semelhante a qual?)
- f) Multiplicar com o resultado em 64 bits os números 0x11223344 e 0x44332211
- g) Dividir o número 0xFFFF7560 por 1000 com sinal;
- h) Dividir 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:

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

- x → especifica a condição para a segunda instrução no bloco (T ou E)
- y → especifica a condição para a terceira instrução no bloco (T ou E)
- z → especifica a condição para a quarta instrução no bloco (T ou E)

- Exemplo:

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

Códigos de Condição

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

Bloco If-Then

- Exercícios

5. Faça um código que realize os seguintes passos e depois depure no Keil: (Acrescentar ao final do arquivo a instrução **NOP**)

Verifique na simulação, os valores dos registradores antes e depois.

- 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
 - Se sim, salve o número 50 no R1
 - Se sim, adicione 32 com o R1 e salve o resultado em R2
 - 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*)

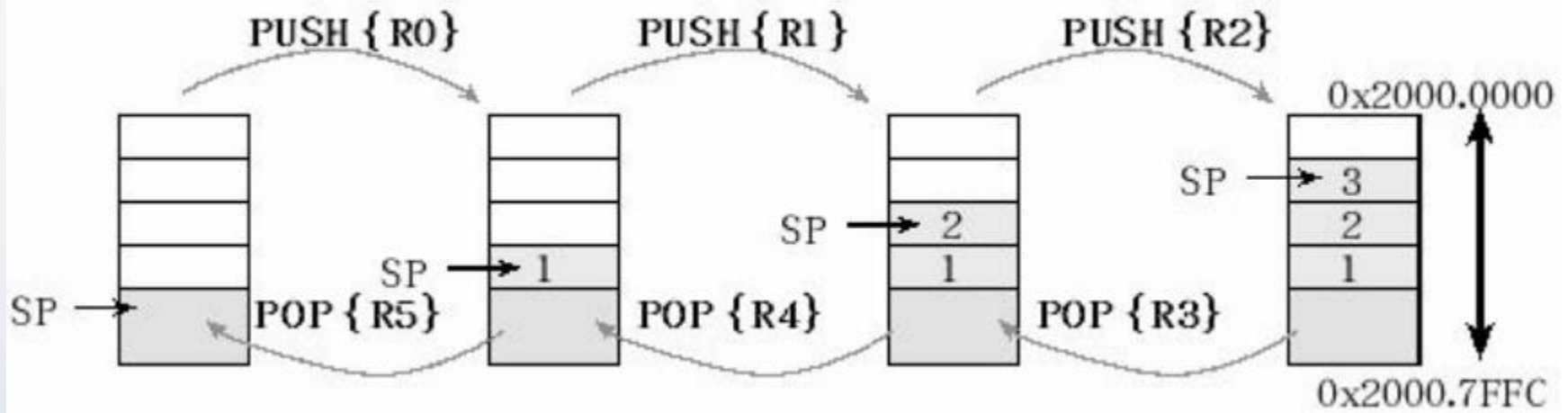
- Comandos

PUSH <reglist>

POP <reglist>

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

- Exemplo de operação:



Pilha (*Stack*)

- Exercícios

6. Faça um código que realize os seguintes passos e depois depure no Keil: (Acrescentar ao final do arquivo a instrução **NOP**)

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

Instruções de Salto

- Instruções para interromper o fluxo ou chamar subrotinas.

`B{cond} label ;salta para o label`

`BL{cond} label ;salta para a subrotina no label`

`BX{cond} Rm ;salta para a localização especificada por Rm`

- Instrução `B{cond}` é a única que não precisa estar dentro de um bloco if-then

- Outras instruções de salto **condicional**:

`CBZ Rn, <label> ;compara e salta se zero`

`CBNZ Rn, <label> ;compara e salta se não for zero`

B

Start

MOV R0, #4	rom: 0x08	PC=0x0C
ADD R6, R0, #10	rom: 0x0C	PC=0x10
LSL R2, R6, #2	rom: 0x10	PC=0x14
B pula1	rom: 0x14	PC=0x20
SUB R0, R1, #3	rom: 0x18	
ADD R0, R2, R5	rom: 0x1C	

pula1

MOV R3, #3	rom: 0x20	PC=0x24
MOV R2, #2	rom: 0x24	PC=0x28

B (cond)

Start

MOV R0, #4	rom: 0x08	PC=0x0C
ADD R6, R0, #10	rom: 0x0C	PC=0x10
CMP R6, #10	rom: 0x10	PC=0x14
BEQ pula1	rom: 0x14	PC=0x20
SUB R0, R1, #3	rom: 0x18	
ADD R0, R2, R5	rom: 0x1C	

pula1

MOV R3, #3	rom: 0x20	PC=0x24
MOV R2, #2	rom: 0x24	PC=0x28

BL

Start

MOV R0, #4	rom: 0x08	PC=0x0C	
ADD R6, R0, #10	rom: 0x0C	PC=0x10	
LSL R2, R6, #2	rom: 0x10	PC=0x14	
BL pula1	rom: 0x14	PC=0x20	LR=0x18
SUB R0, R1, #3	rom: 0x18		
ADD R0, R2, R5	rom: 0x1C		

pula1

MOV R3, #3	rom: 0x20	PC=0x24	
MOV R2, #2	rom: 0x24	PC=0x28	
BX LR	rom: 0x28	PC=LR=0x18	

BL (2)

Start

MOV R0, #4	rom: 0x08	PC=0x0C	
ADD R6, R0, #10	rom: 0x0C	PC=0x10	
LSL R2, R6, #2	rom: 0x10	PC=0x14	
BL pula1	rom: 0x14	PC=0x20	LR=0x18
SUB R0, R1, #3	rom: 0x18		
ADD R0, R2, R5	rom: 0x1C		

pula1

MOV R3, #3	rom: 0x20	PC=0x24	
BL func2	rom: 0x24	PC=0x30	LR=0x28
MOV R2, #2	rom: 0x28	PC=0x34	
BX LR	rom: 0x2C	PC=LR=0x28	

func2

SUB R1, #3	rom: 0x30	PC=0x34	
BX LR	rom: 0x34	PC=LR=0x28	

BL (2)

Start

MOV R0, #4	rom: 0x08	PC=0x0C	
ADD R6, R0, #10	rom: 0x0C	PC=0x10	
LSL R2, R6, #2	rom: 0x10	PC=0x14	
BL pula1	rom: 0x14	PC=0x20	LR=0x18
SUB R0, R1, #3	rom: 0x18		
ADD R0, R2, R5	rom: 0x1C		

pula1

MOV R3, #3	rom: 0x20	PC=0x24	
PUSH {LR}	rom: 0x24	PC=0x28	
BL func2	rom: 0x28	PC=0x38	LR=0x2C
POP {LR}	rom: 0x2C	PC=0x30	LR=0x18
MOV R2, #2	rom: 0x30	PC=0x34	
BX LR	rom: 0x34	PC=LR=0x18	

func2

SUB R1, #3	rom: 0x38	PC=0x34	
BX LR	rom: 0x3C	PC=LR=0x28	

Instruções de Salto

- Exercícios

7. Faça um código que realize os seguintes passos e depois depure no Keil:

- 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:
 - d.1) Copia o R0 para R1
 - d.2) Verifica se R1 é menor que 50
 - d.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

```
AREA RESET, CODE, READONLY      ;reset vectors in flash ROM
AREA DATA                      ;places objects in data memory (RAM)
AREA |.text|, CODE, READONLY, ALIGN=2      ;code in flash ROM
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

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


Diretivas do Assembler do Keil

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

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

```
THUMB
```

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

```
END
```

Diretivas do Assembler do Keil

- Adicionando variáveis e constantes:

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

- Como fazer um vetor?

Declarar na região da RAM (abaixo do AREA DATA)

```
nome_do_vetor    SPACE    5 (5 bytes)
```

Na região do código quando for utilizar

```
LDR R0, =nome_do_vetor    (região inicial)
```

```
LDRB R1, [R0]             (vetor de bytes)
```

Diretivas do Assembler do Keil

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

```
GPIO_PORTD_DATA_R EQU 0x400073FC  
GPIO_PORTD_DIR_R   EQU 0x40007400  
GPIO_PORTD_DEN_R   EQU 0x4000751C
```

Assembly ARM Cortex-M4



Exercícios

- Exercícios

8. Ler de uma posição da memória RAM um número e calcular o fatorial. Armazenar o resultado em R0.