ASSEMBLY PASSO A PASSO

WIE

OLÁ, EU SOU PAULO

Graduando em Engenharia de Computação pela Universidade Estadual de Feira de Santana, atualmente sou monitor da disciplina TEC499 Módulo Integrador Sistemas Digitais

OLÁ, EU SOU LARA

Graduando em Engenharia de Computação pela Universidade Estadual de Feira de Santana, auxiliar do monitor

ROTEIRO

- Oque é Assembly
- Primeiros passos
- Arquitetura do processador
- Primeiras Instruções
- Operações logicas
- Operações Aritméticas
- Desvio e loops
- Memoria
- Funções e macros

O QUE É ASSEMBLY?

ASSEMBLY

- Linguagem mais baixo nível
- Compilação gera o assembly
- Ação do processador
- Assembler

POR QUE SABER ASSEMBLY?

1. ENTENDER COMO FUNCIONA O COMPUTADOR
2. OTIMIZAÇÃO DE PROCESSOS
3. APRENDIZADO DE MAQUINA E GRÁFICOS 3D
4. SEGURANÇA OU QUEBRA DE SOFTWARE

ESTRUTURA DO ASSEMBLY

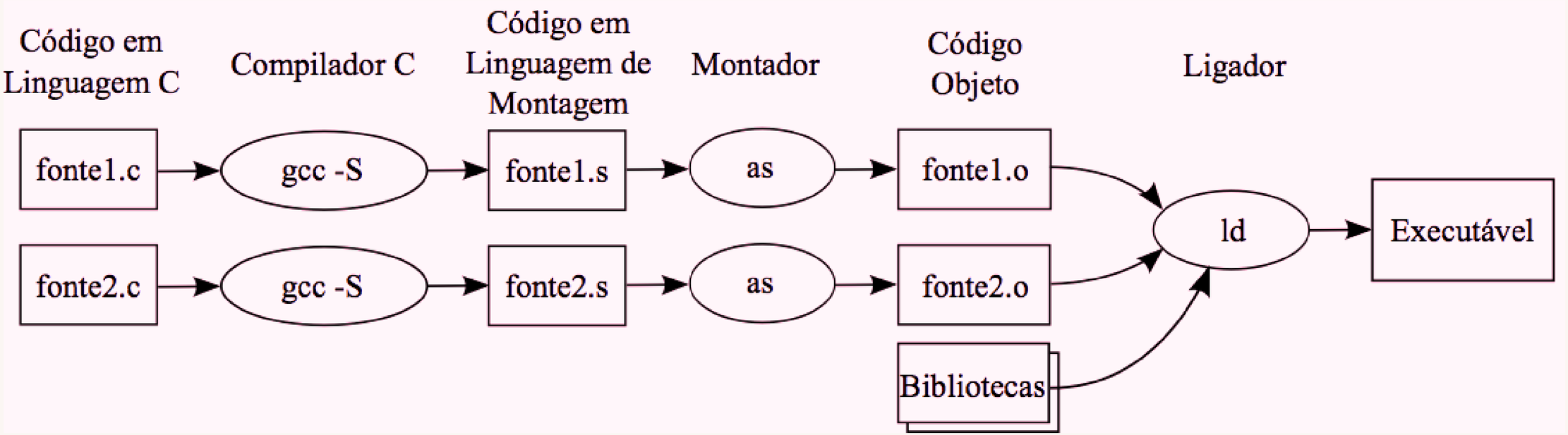
- Divisão do assembly
  - Diretivas
  - Labels
  - Instrução
  - Comentario
- Cada instrução esta em algum lugar da memoria
- Cada instrução tem 32 bits
- Operação e Operandos

```
13 @    r6: hexstr address
14 @
15 @    r8: current byte
16 @    r9: byte half
17 @
18 @    r11: inner loop counter
19
20
21 .global _start
22
23 _start:
24     ldr     r4, =input      @ load input array
25     ldr     r6, =hexstr     @ load hexstr address
26     ldrb    r8, [r4], #1    @ load first byte and incr
27 loop:
28     cmp     r8, #0          @ see if we have a null
29     beq     exit
30     ldr     r5, =outstr     @ set output address
31     mov     r11, #0         @ reset loop counter
32 inner:
33     cmp     r11, #3         @ check to see if end of pr
34     bge     inexit          @ jump to inner loop cleanu
35     @ process first letter/hex
36     and     r9, r8, #0xf0   @ mask byte by 0b11110000
37
38
39 exit:
40
41 .data
42 hexstr:     .ascii      "0123456789ABCDEF"
43 input:      .byte       212, 228, 188
main.s [+]
```

HELLO WORLD!

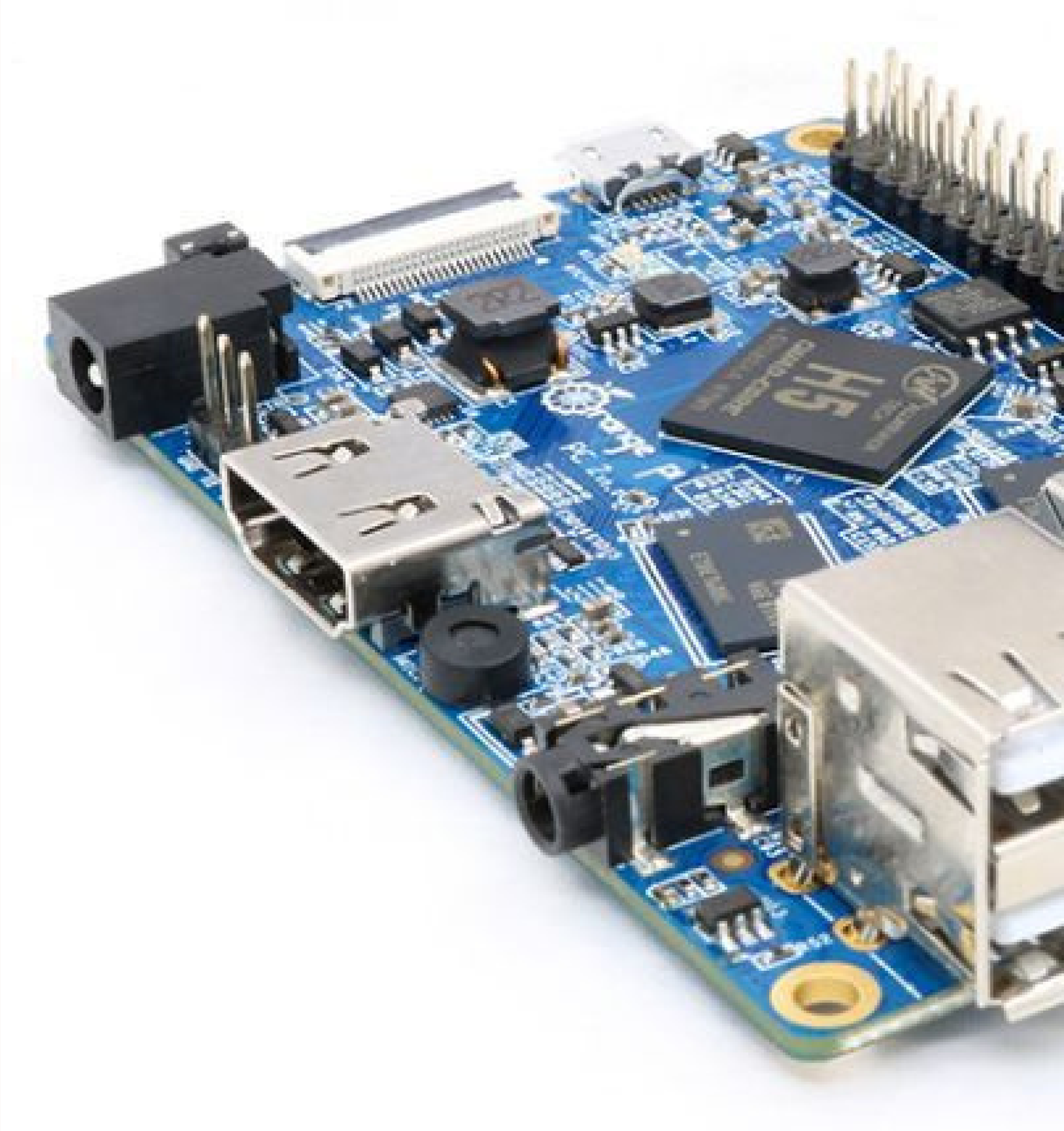
```
@ Assembler program to print "Hello World!"
@ to stdout.
@
@ R0-R2 - parameters to linux function services
@ R7 - linux function number
@

.global _start          @ Provide program starting
@ address to linker

@ Set up the parameters to print hello world
@ and then call Linux to do it.
_start: mov R0, #1      @ 1 = StdOut
      ldr   R1, =helloworld   @ string to print
      mov   R2, #13     @ length of our string
      mov   R7, #4      @ linux write system call
      svc   0           @ Call linux to print

@ Set up the parameters to exit the program
@ and then call Linux to do it.
      mov     R0, #0  @ Use 0 return code
      mov     R7, #1  @ Service command code 1
                      @ terminates this program
     svc     0        @ Call linux to terminate

.data
helloworld:      .ascii  "Hello World!\n"
```

TRADUÇÃO DE CÓDIGO



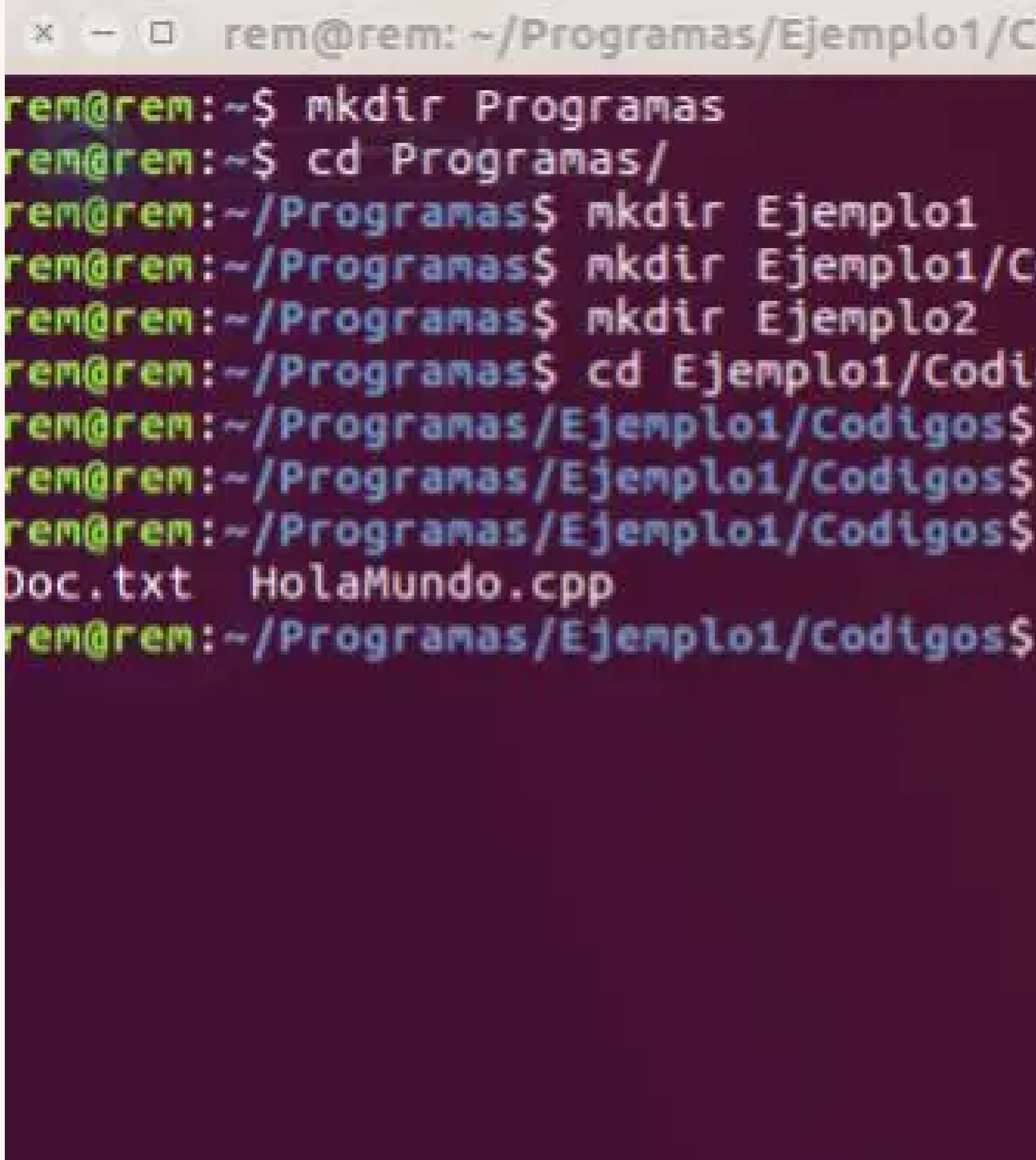
ACESSANDO A SBC

- Orange-pi PC PLUS
- Secure Shell (SSH)
- Entrando na rede:
  - U: INTELBRAS
  - P: Pbl-Sistemas-Digitais
- Abrir o terminal
  - ssh aluno@10.0.0.103
  - ssh aluno@10.0.0.104



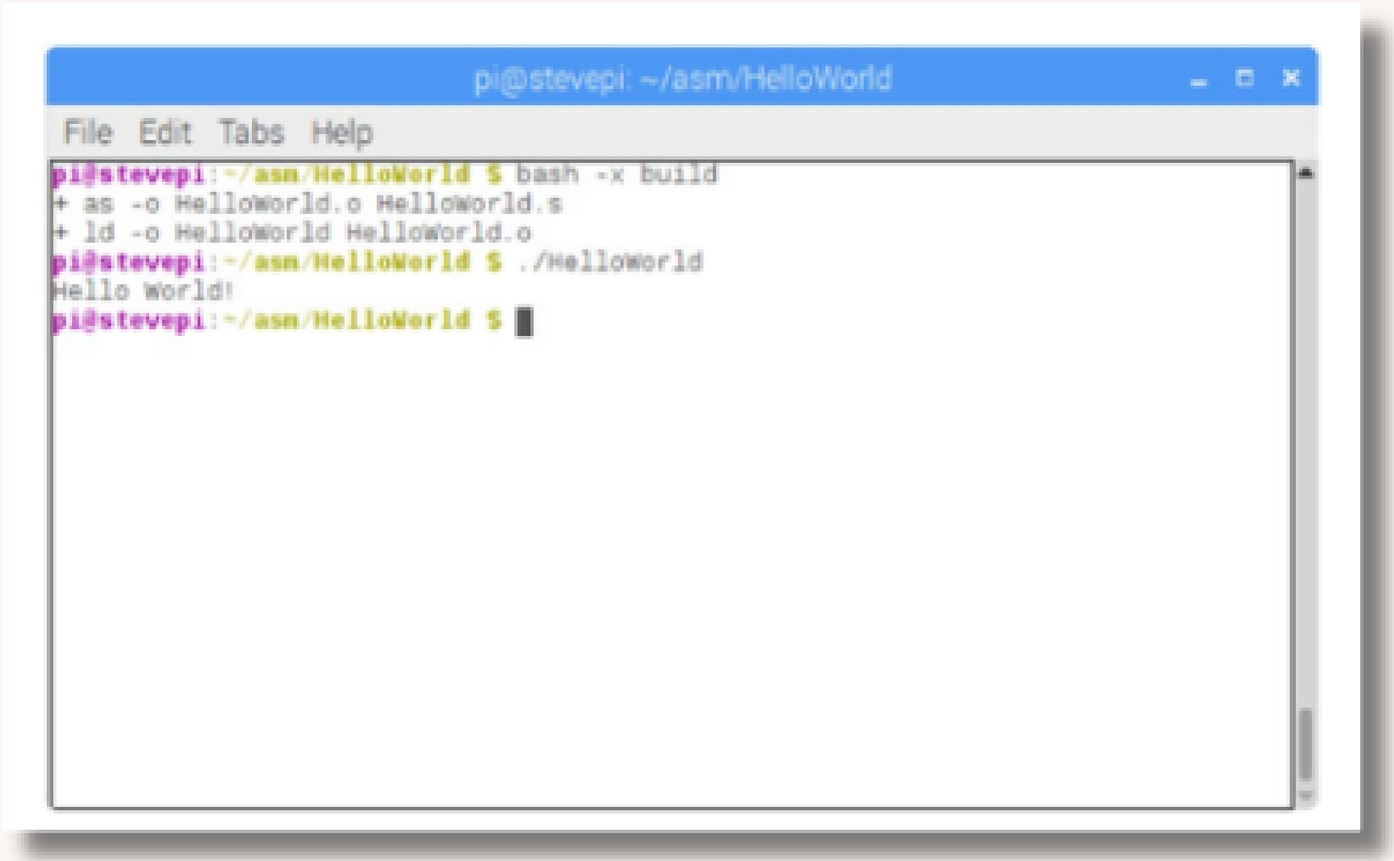
USANDO O LINUX

- **ls**
  - Listas os arquivos do diretório
- **cd**
  - vai para um diretório
  - cd OficinaAssembly
- **mkdir**
  - cria um novo diretório
  - mkdir <Nome da dupla>
- **nano**
  - Abre um arquivo como editor de texto
  - nano helloworld.s

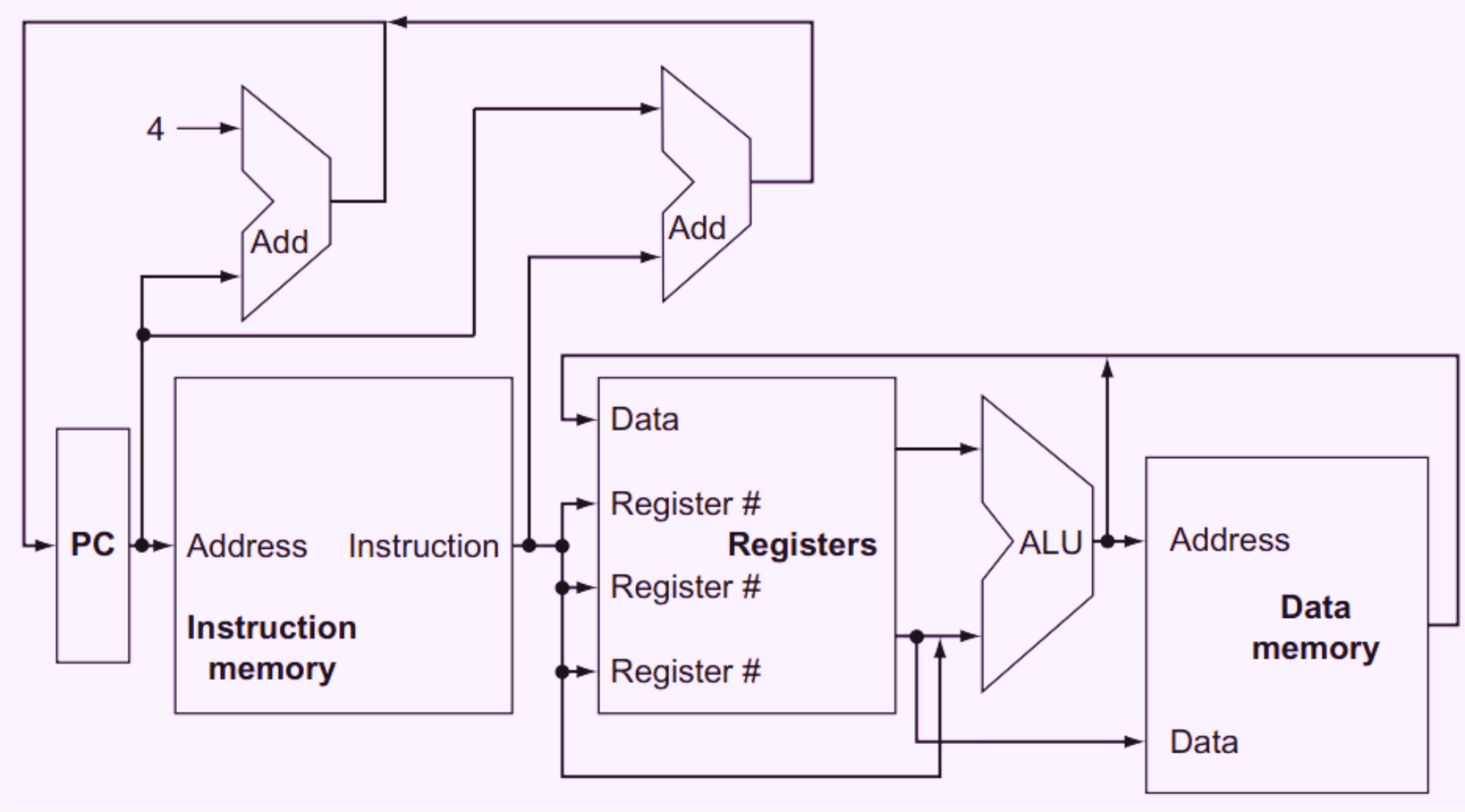


EXECUTANDO O CODIGO

```
as -o HelloWorld.o HelloWorld.s
ld -o HelloWorld HelloWorld.o
```



ARQUITETURA DO PROCESSADOR



BANCO DE REGISTRADOR

Memoria rápida e interna no processador.

- r0: 1st function argument, scratch, function return value
- r1: 2nd function argument, scratch
- r2: 3rd function argument, scratch
- r3: 4th function argument, scratch
- r4: callee-saved
- r5: callee-saved
- r6: callee-saved
- r7: callee-saved, system call number
- r8: callee-saved
- r9: callee-saved
- r10: callee-saved
- r11: callee-saved
- r12 (ip): scratch
- r13 (sp): stack pointer, callee-saved
- r14 (lr): link register, scratch
- r15 (pc): program counter

HELLO WORLD!

```
@ Assembler program to print "Hello World!"
@ to stdout.
@
@ R0-R2 - parameters to linux function services
@ R7 - linux function number
@

.global _start          @ Provide program starting
@ address to linker

@ Set up the parameters to print hello world
@ and then call Linux to do it.
_start: mov R0, #1      @ 1 = StdOut
      ldr   R1, =helloworld   @ string to print
      mov   R2, #13     @ length of our string
      mov   R7, #4      @ linux write system call
      svc   0           @ Call linux to print

@ Set up the parameters to exit the program
@ and then call Linux to do it.
      mov     R0, #0  @ Use 0 return code
      mov     R7, #1  @ Service command code 1
                      @ terminates this program
     svc     0         @ Call linux to terminate

.data
helloworld:      .ascii  "Hello World!\n"
```

PRIMEIRAS INSTRUÇÕES

MOV

1. Transfere um valor para um registrador
2. MOV r2, r1
3. MOV r2, #4

```
mov     R0, #0  @ Use 0 return code
mov     R7, #1  @ Service command code 1
```

EXERCICIO #1

- Utilize a instrução MOV para colocar nos registradores os valores:
  - **8** no **R3**
  - **5** no **R2**
  - O valor do registrador **R13** (sp) no **R1**

EXERCICIO #1

```
.global _start
_start:
        MOV     R3, #8
        MOV     R2, #5
        MOV     R1, SP

end:    MOV     R0, #0
        MOV     R7, #1
        SVC     0
```

DEBUGANDO NO GDB

- Método para entender passo a passo do seu código
- as **-g** -o codigo.o codigo.s
- gdb codigo
- Usando:
  - b _start
  - r
  - s
  - i r

OPERAÇÕES LOGICAS

Deslocamento de bits

1. Desloca os bits do registrador
2. LSL (Logical Shift Left)
   - LSL R2, R1, #3
3. LSR (Logical Shift Right)
   - LSR R2, R2, #2
4. ASR (Arithmetic Shift Right)
   - ASR R3, R4, #8

EXERCICIO #2

- Coloque o valor do registrador R13 (sp) no registrador R0 e faça:
  - Desloque o valor do R0 para a direita 3x e salve no R1
  - Desloque o valor do R0 em 6 para e direita e salve no R2
- Calcule o dobro do valor de R2 e salve em R3

EXERCICIO #2

```
.global _start
_start:
        MOV     R0, SP
        LSR     R0, #3
        MOV     R1, R0
        LSL     R0, #6
        MOV     R2, R0

        LSL     R3, R2, #1

end:    MOV     R0, #0
        MOV     R7, #1
        SVC     0
```

OPERAÇÕES LOGICAS

And, Or, Exclusive Or e Bit Clear

1. And
   - AND R2, R1, #0xf
2. Or
   - ORR R3, R1, R2
3. Exclusive Or
   - EOR R2, R2, #2
4. Bit Clear
   - BIC R1, R0, #0b1100

BIT CLEAR

Operação logica que junta a operação And e Not para criar uma operação de limpeza de bits.

1	0	0	1	1	1	1
0	0	1	1	1	0	0
1	0	0	0	0	1	1

EXERCICIO #3

- Considerando R1 = 0xf0, R2 = 0xf, R3 = 0xf0f e r4 = 0xff0 e calcule: (R1 ∪ R3) ∩ (R2 ∪ R4)
- Extra: Coloque no R0 o valor 0b110110, posteriormente substitua os bits 2 a 4 por 010

EXERCICIO #3

```
.global _start
_start:
        MOV     R1, #0xf0
        MOV     R2, #0xf
        MOV     R3, #0xf0f
        MOV     R4, #0xff0
        ORR     R1, R1, R3
        ORR     R2, R2, R4
        AND     R1, R1, R2

extra:  MOV     R0, #0b110110
        BIC     R0, #0b011100
        MOV     R5, #0b010
        LSL     R5, #2
        ORR     R0, R0, R5

end:    MOV     R7, #1
        SVC     0
```

OPERAÇÕES ARITMÉTICAS

ADD,SUB,MUL,SDIV/UDIV

1. **ADD**
   - ADD R2, R1, #0xf
2. SUB
   - SUB R3, R1, R2
3. MUL
   - MUL R2, R2, #2
4. SDIV/UDIV
   - SDIV R1, R0,R4

EXERCICIO #4

- Considerando g=6 , h=5, i=3,j=5

  calcule:

  - f = (g + h) - (i + j)
  - ((f²) + f)/3
- Extra: Calcule o resto da divisão de:
  - (g+h)/i

EXERCICIO #4

```
.global _start
_start:
        MOV     R1, #6  @g
        MOV     R2, #5  @h
        MOV     R3, #3  @i
        MOV     R4, #5  @j
        ADD     R5, R1, R2
        ADD     R6, R3, R4
        SUB     R0, R5, R6 @f

        MUL     R1, R0, R0
        ADD     R1, R0
        SDIV    R0, R1, R3

extra:  ADD     R5, R1, R2
        SDIV    R6, R5, R3
        MUL     R6, R6, R3
        SUB     R6, R5, R6

end:    MOV     R0, #0
        MOV     R7, #1
        SVC     0
```

REGISTRADOR PC

- Cada instrução esta em um local da memoria que e chamado pelo seu endereço
- O registrador PC guarda o endereço da próxima instrução
- Mas e se eu não quiser que a próxima instrução seja literalmente a proxima?

BRANCH OU DESVIO

Branch é uma instrução essencial nos processadores modernos que permite desviar o fluxo do programa para outros trechos de código, possibilitando a criação de estruturas condicionais e loops, tornando os programas mais versáteis e eficientes.

```
.global _start
_start:
        MOV     R1, #0xf0f0
        ADD     R1, R1, #0xf
        B       goto
        MOV     R3, R1
goto:   MOV     R2, R1
        BIC     R2, R2, R3

        MOV     R0, #0
        MOV     R7, #1
        SVC     0
```

CPSR E FLAGS

- Flags Condicionais
  - **N**
  - **Z**
  - C
  - V
- Interrupções
  - I
  - F
  - A
- Instruções
  - Thumb
  - Jazelle
- Outros
  - Q
  - GE
  - M

31	30	29	28	27	-	24	-	19 – 16	-	9	8	7	6	5	4 – 0
N	Z	C	V	Q		J		GE		E	A	I	F	T	M

Figure 4-1. *The bits in the CPSR*

BRANCH CONDICIONAIS E CPM

1. B {conditional} label
   - _start:

     MOV R0,1

     MOV R1, 2

     CMP R1,R2

     BEQ _start

Table 4-1. *Condition codes for the branch instruction*

{condition}	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or **HS**	C set	Higher or same (unsigned >=)
CC or **LO**	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear and Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always (same as no suffix)

A INSTRUÇÃO ADDS E SUBS

1. Não é necessário usar CPM

- _start:

  MOV R0,#10

  SUBS R0,#1

  BNE _start

Table 4-1. Condition codes for the branch instruction

{condition}	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or **HS**	C set	Higher or same (unsigned >=)
CC or **LO**	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear and Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always (same as no suffix)

IF/ELSE

```
int main () {
   int R0 = 5;
   if (numero > 0) {
       R0 = 1;
    } else if (numero < 0) {
       R0 = -1;
    } else {
       R0 = 0;
    }
    return R0;
}
```

```
.global _start

_start:
        MOV     R0, #5
        CMP     R0, #0
        BGT     if
        BLT     elseif
        MOV     R0, #0
        B       end
if:     MOV     R0, #1
        B       end
elseif: MOV     R0, #-1
end:    MOV     R7, #1
        SVC     0
```

LOOPS: FOR

```
int main () {
    int R0 = 1;
    for (int R1 = 0; R1 < 5; R1++){
       R0 = R0 * 3;
    }
    return R0
}
```

```
.global _start
_start:
        MOV     R0, #1
        MOV     R1, #0
        MOV     R2, #3
for:    CMP     R1, #5
        BGE     end
        MUL     R0, R2
        ADD     R1, #1
        B       for
end:    MOV     R7, #1
        SVC     0
```

LOOPS: WHILE

Algoritmo de Euclides para MDC

```
int main () {
    int R0 = 56;
    int R1 = 32;
    int R2 = 0;
    while (R1 != 0) {
        R2 = R0 % R1;
        R0 = R1;
        R1 = R2;
   }
   return R0;
}
```

Table 4-1. *Condition codes for the branch instruction*

{condition}	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or **HS**	C set	Higher or same (unsigned >=)
CC or **LO**	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear and Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always (same as no suffix)

LOOPS: WHILE

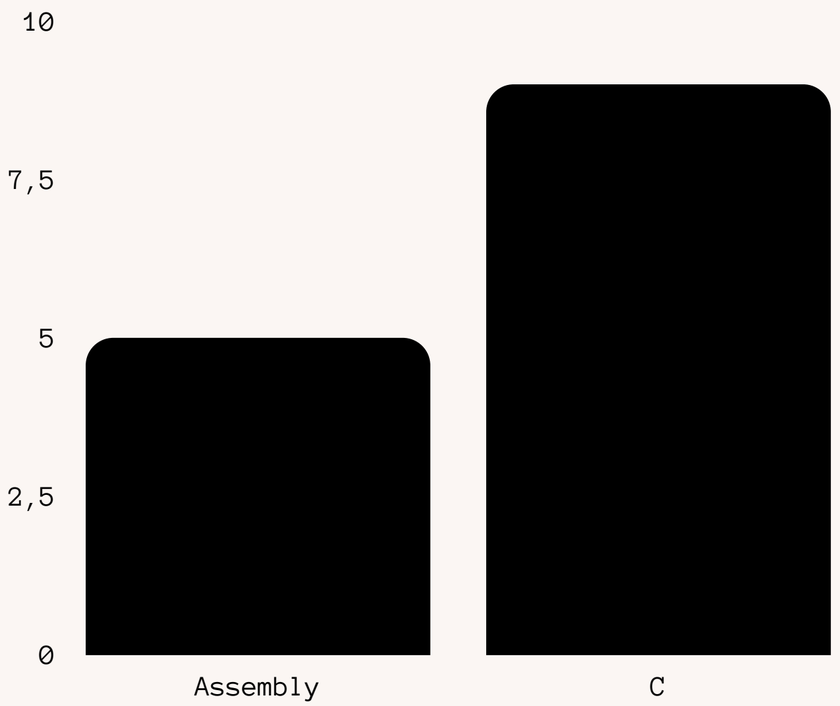
Algoritmo de Euclides para MDC

```
int main () {
    int R0 = 56;
    int R1 = 32;
    int R2 = 0;
    while (R1 != 0) {
        R2 = R0 % R1;
        R0 = R1;
        R1 = R2;
    }
    return R0;
}
```

```
.global _start
_start:
        MOV     R0, #56
        MOV     R1, #32
        MOV     R2, #0
while:  CMP     R1, #0
        BEQ     end
        SDIV    R3, R0, R1
        MUL     R4, R3, R1
        SUB     R2, R0, R4
        MOV     R0, R1
        MOV     R1, R2
        B       while
end:    MOV     R7, #1
        SVC     0
```

TEMPO DE EXECUÇÃO

Usando o comando no linux time
podemos calcular o tempo de execução
de um processo. É esse foi o resultado



DIRETIVAS DE MÉMORIA

- Decimais começam com números de 1-9, e contém dígitos de 0-9
- Octais começam com 0, e contém dígitos de 0-7
- Binários começam com ob ou 0B, contém 0 ou 1
- Hexadecimais começam com 0x ou 0X, contém dígitos de 0-F
- Flutuantes começam com 0f ou 0F, e contém um número flutuante

Table 5-1. *The list of memory definition Assembler directives*

Directive	Description
.ascii	A string contained in double quotes
.asciz	A zero-byte terminated ascii string
.byte	1-byte integers
.double	Double-precision floating-point values
.float	Floating-point values
.octa	16-byte integers
.quad	8-byte integers
.short	2-byte integers
.word	4-byte integers

```
label: .byte 74, 0112, 0b00101010, 0x4A, 0X4a, 'J', 'H' + 2
       .word  0x1234ABCD, -1434
       .ascii      "Hello World\n"
```

INSTRUÇÃO LDR

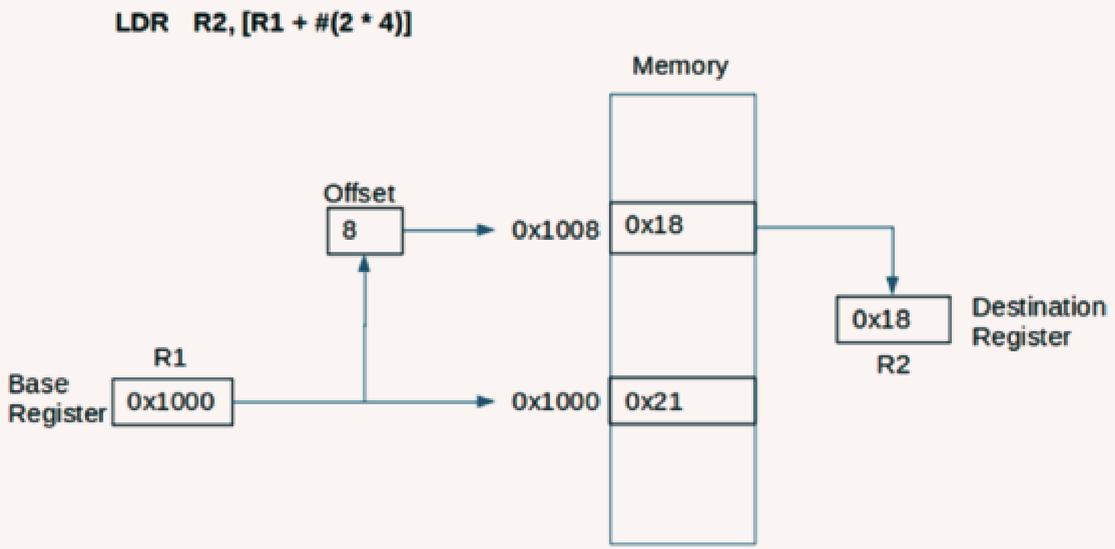
- Arquitetura Load-Store
- Load carrega um endereço de memória ou o valor que está dentro de uma memória

Listing 5-2. Loading an address and then the value

```
@ load the address of mynumber into R1
      LDR    R1, =mynumber
@ load the word stored at mynumber into R2
      LDR    R2, [R1]              .data

mynumber:    .WORD 0x1234ABCD
```

LDR: INDEXANDO UM ARRAY

```
@ Load the first element
LDR   R2, [R1]
@ Load element 3
@ The elements count from 0, so 2 is
@ the third one. Each word is 4 bytes,
@ so we need to multiply by 4
LDR   R2, [R1, #(2 * 4)]
```



Carregando o terceiro elemento de um array de word

INSTRUÇÃO STR E .DATA

- Instrução espelho de Load, pois guarda os valores na memória
- .data é utilizada para indicar o início da seção de dados, onde armazenamos dados que podem ser lidos e escritos durante a execução do programa

```
.data
inFile:  .asciz  "main.s"
outFile: .asciz    "upper.txt"
buffer:     .fill  BUFFERLEN + 1, 1, 0
outBuf:     .fill  BUFFERLEN + 1, 1, 0
inpErr: .asciz     "Failed to open input file.\n"
inpErrsz: .word  .-inpErr
outErr:     .asciz       "Failed to open output file.\n"
outErrsz: .word   .-outErr
```

INSTRUÇÃO STR E .DATA

```
.global _start

_start:
        LDR     r0, =array
        MOV     r1, #10

array_loop:
        LDR     r3, [r0]
        ADD     r3, #1
        ADD     r0, r0, #4
        STR     r3, [r0]
        LDR     r4, [r0]
        SUBS    r1, r1, #1
        BGE     array_loop

        MOV     r7, #1
        SVC     0

.section .data
array: .word    0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

EXERCICIO #5

Dado a área de dado abaixo:

```
.data
a:    .word    0xf
b:    .word    0
c:    .word     0
```

Crie um código que coloque no b e c o dobro e o quadraplo do valor em a respectivamente.

EXERCICIO #5

```
.global _start
_start:
        LDR     R0, =a
        LDR     R1, =b
        LDR     R2, =c

        LDR     R4, [R0]
        LSL     R4, #1
        STR     R4, [R1]
        LSL     R4, #1
        STR     R3, [R2]

end:    MOV     R0, #0
        MOV     R7, #1
        SVC     0

.data
a:      .word   0xf
b:      .word   0
c:      .word   0
```

PILHA

- Área da Memória onde ocorre duas operações: Push e Pop
- LIFO(Last in first out)
- LDM e STM

```
PUSH   {r0, r5-r12}
POP {r0-r4, r6, r9-r12}
```

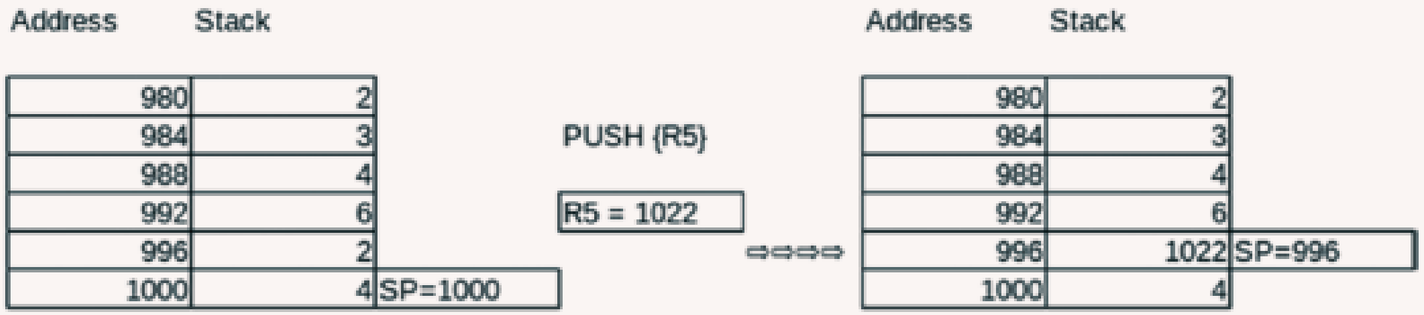


Figure 6-1. *Pushing R5 onto the stack*

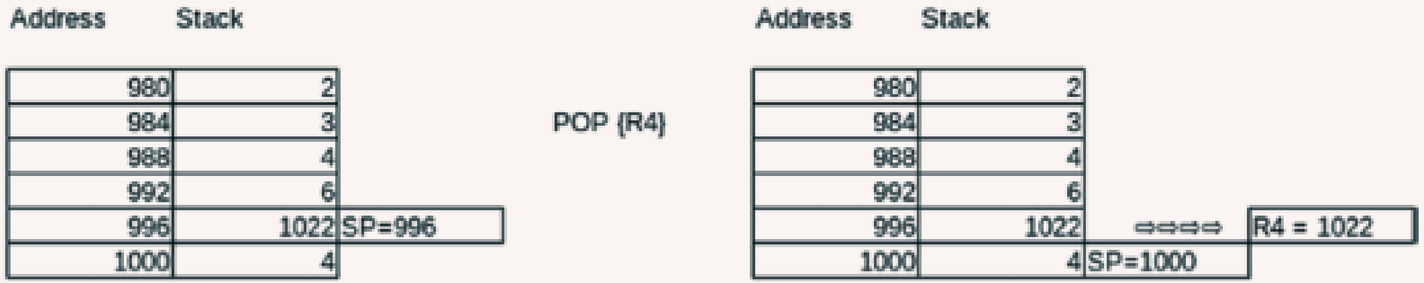


Figure 6-2. *Popping R4 from the stack*

BRANCH COM LINK(BL)

- Instrução usada para chamar função no ARM Assembly
- Copia o endereço da próxima instrução para LR
- Desvia a execução para o endereço especificado em BL

Listing 6-1. Skeleton code to call a function

```
        @ ... other code ...
        BL    myfunc
        MOV   R1, #4
        @ ... more code ...
-----------------------------
myfunc:       @ do some work
              BX LR
```

BL – FUNÇÕES DENTRO DE FUNÇÕES E PARÂMETROS

- Necessidade de uso da pilha pois o valor do LR será perdido
- Parâmetros de função devem ser passado do R1-R3, caso tenha mais que 4 parâmetros, os valores devem ser salvos na pilha. E o retorno tem que ser no R0.

Listing 6-2. Skeleton code for a function

```
        @ ... other code ...
        BL    myfunc
        MOV   R1, #4
        @ ... more code ...
----------------------------
myfunc:     PUSH {LR}
            @ do some work ...
            BL    myfunc2
            @ do some more work...
            POP {LR}
            BX LR
myfunc2:    @ do some work ....
            BX LR
```

EXERCICIO #6

Faça uma função que calcule a função quadrática $f(x) = x^2 + 2x - 3$ e coloque nos registradores R1, R2, R3 os valores de f(1), f(2), f(3) respectivamente.

EXERCICIO #6

```
.global _start
f:
        PUSH    {R2}
        MOV     R0, #0
        MUL     R2, R1, R1
        ADD     R0, R0, R2
        LSL     R2, R1, #1
        ADD     R0, R2
        SUB     R0, #3
        POP     {R2}
        BX      LR

_start:
        MOV     R1, #2
        BL      f
        MOV     R2, R0
        MOV     R1, #3
        BL      f
        MOV     R3, R0
        MOV     R1, #1
        BL      f
        MOV     R1, R0

end:    MOV     R0, #0
        MOV     R7, #1
        SVC     #0
```

MACROS

1. É uma diretiva para evitar a repetição de código

   - .MACRO macronome parametro1 , parametro2,...

```
.macro multiply x, y, result
    MOV \result, #0
    MOV R4, \x
    MOV R5, \y
loop:
    ADD \result, \result, R4
    SUBS R5, R5, #1
    BNE loop
.endm

multiply 3, 4, R0
```

EXERCICIO #7

Repita o Exercício anterior usando macros em vez de função e pense qual a diferença:

EXERCICIO #7

```
.global _start
.macro f, x, y
        PUSH    {R0}
        MOV     \y, #0
        MUL     R0, \x, \x
        ADD     \y, R0
        LSL     R0, \x, #1
        ADD     \y, R0
        SUB     \y, #3
        POP     {R0}
.endm

_start:
        MOV     R4, #1
        f       R4, R1
        MOV     R4, #2
        f       R4, R2
        MOV     R4, #3
        f       R4, R3

end:    MOV     R0, #0
        MOV     R7, #1
        SVC     #0
```

DESAFIO FINAL

Crie no Assembly um programa que calcule o numero na posição X da sequencia Fibonacci em uma função

1, 1, 2, 3, 5, 8, 13 ...

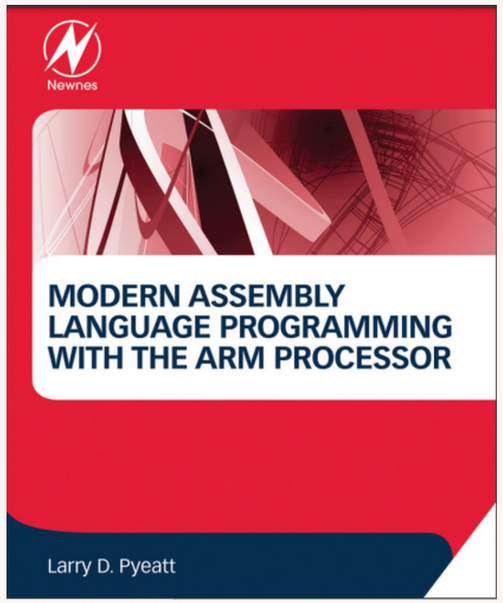
DESAFIOS

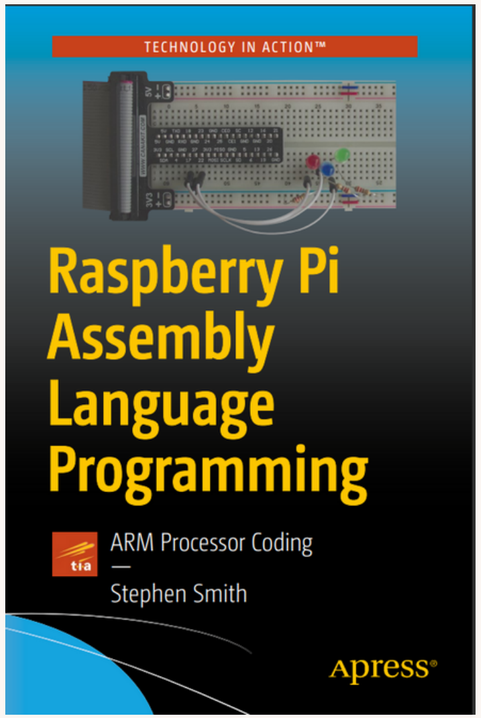
Crie no Assembly um programa que calcule o numero na posição X da sequencia Fibonacci em uma função

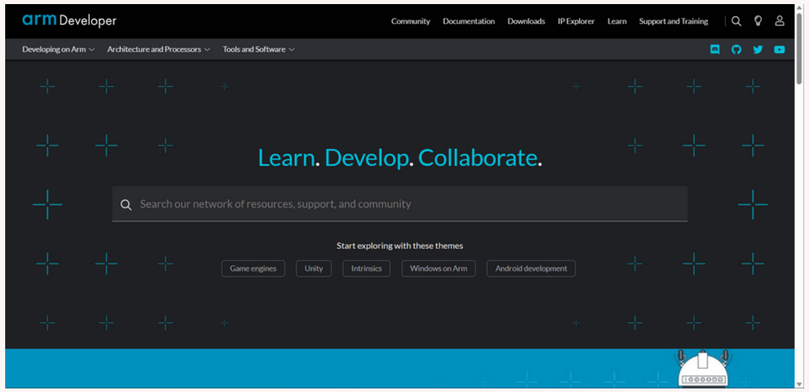
1, 1, 2, 3, 5, 8, 13 ...

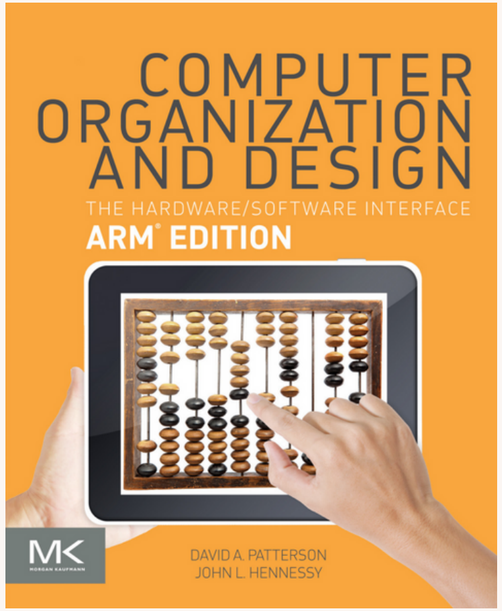
```
.global _start
fibonacci:
        PUSH    {R2, R3, LR}
        CMP     R1, #1
        BEQ     fib1
        CMP     R1, #0
        BEQ     fib0
        MOV     R2, R1
        SUB     R1, R2, #1
        BL      fibonacci
        MOV     R3, R0
        SUB     R1, R2, #2
        BL      fibonacci
        ADD     R0, R3
        B       endfib
fib1:   MOV     R0, #1
        B       endfib
fib0:   MOV     R0, #0
endfib: POP     {R2, R3, LR}
        BX      LR

_start:
        MOV     R1, #10
        BL      fibonacci
        MOV     R7, #1
        SVC     0
```

REFERÊNCIAS









Obrigado!