Projeto de Organização e Arquitetura de Computadores II PCS3422 - Grupo D

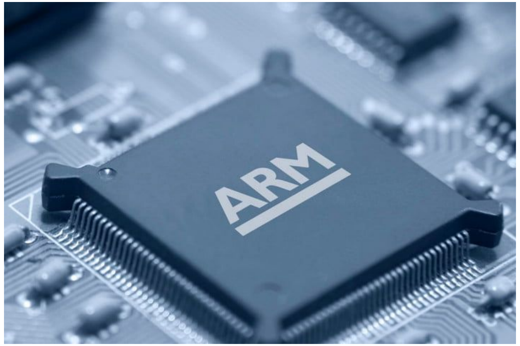
Toolchain - Compilador completo compatível com ISA LEGv8 com saída binária e .MIF

Toolchain

➜ Conjunto de ferramentas de compilação

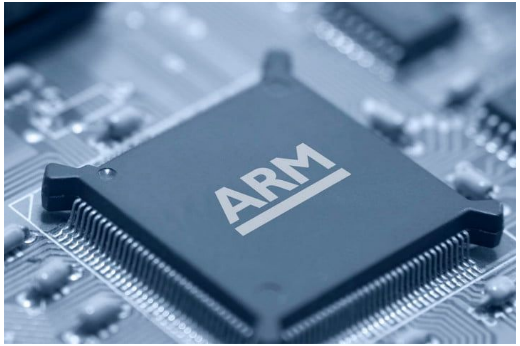
Ex: pré-processador (C intermediário), compilador (assembly), assembler (arquivo objeto) e linker (binário final).

Toolchain



- ➜ Conjunto de ferramentas de compilação
- ➜ Cross-compiling: arquitetura diferente da máquina de desenvolvimento

Toolchain



- ➜ Conjunto de ferramentas de compilação
- ➜ Cross-compiling: arquitetura diferente da máquina de desenvolvimento
- ➜ Uso de máquina virtual Linux

Problema

➜ Como compilar um programa em C que gere instruções para LEGv8?

Problema

➜ Como compilar um programa em C que gere instruções para LEGv8?

1. Uso de cross-compiling toolchain que gere instruções para ARM

Problema

➜ Como compilar um programa em C que gere instruções para LEGv8?

1. Uso de cross-compiling toolchain que gere instruções para ARM
2. Programa em python 3.7 que converte instruções ARM para LEGv8

Problema

➜ Como compilar um programa em C que gere instruções para LEGv8?

1. Uso de cross-compiling toolchain que gere instruções para ARM
2. Programa em python 3.7 que converte instruções ARM para LEGv8
3. Código assembly traduzido para arquivo objeto (.o)

Problema

➜ Como compilar um programa em C que gere instruções para LEGv8?

1. Uso de cross-compiling toolchain que gere instruções para ARM
2. Programa em python 3.7 que converte instruções ARM para LEGv8
3. Código assembly traduzido para arquivo objeto (.o)
4. Modificação do arquivo .o para .MIF (Memory Initialization File)

Cross-compiling Toolchain

➜ AArch64 GNU/Linux target (aarch64-linux-gnu)

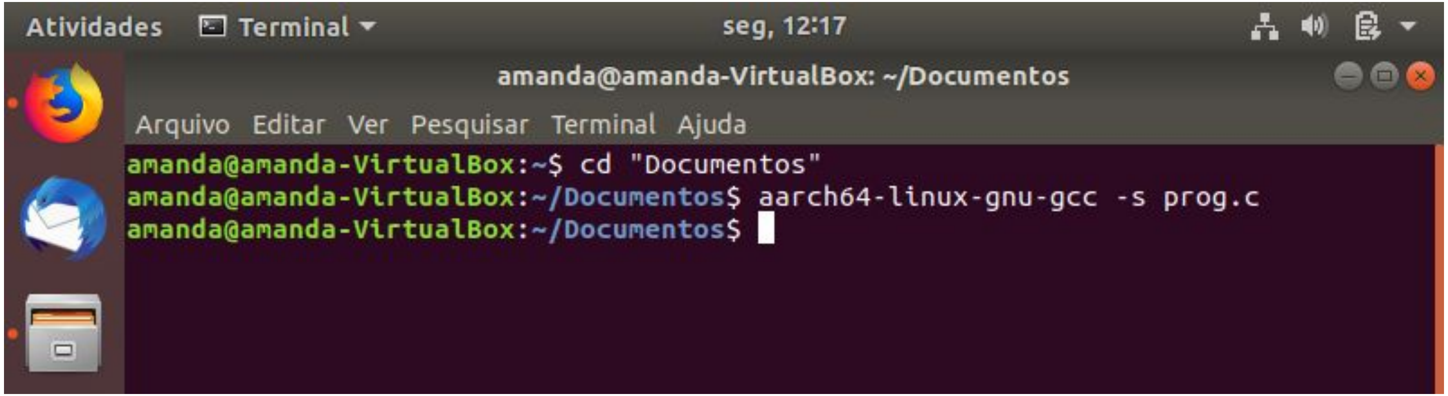
Cross-compiling Toolchain

➜ AArch64 GNU/Linux target (aarch64-linux-gnu)

➜ Programa em C

Cross-compiling Toolchain

➜ Gera o arquivo Assembly (.s)



Arquivo Assembly

- ➜ Arquivo Assembly (.s) com instruções para ARM


```
        .arch armv8-a
        .file   "prog.c"
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "Hello World"
        .text
        .align  2
        .global main
        .type   main, %function
main:
        stp     x29, x30, [sp, -16]!
        add     x29, sp, 0
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      puts
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 7.4.0-1ubuntu1~18.04.1) 7.4.0"
        .section        .note.GNU-stack,"",@progbits
```

Tradução ARM -> LEGv8

➜ Mapeamento da conversão de instruções

➜ Busca de operações LEGv8 equivalentes para cada operação ARM possível

➜ Ex: ADC Xd, Xn, Xm ADD Xd, Xn, HS

ADD Xd, Xm, Xd

LEGv8 assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	ADD X1, X2, X3	X1 = X2 + X3	Three register operands
	subtract	SUB X1, X2, X3	X1 = X2 – X3	Three register operands
	add immediate	ADDI X1, X2, 20	X1 = X2 + 20	Used to add constants
	subtract immediate	SUBI X1, X2, 20	X1 = X2 – 20	Used to subtract constants
	add and set flags	ADDS X1, X2, X3	X1 = X2 + X3	Add, set condition codes
	subtract and set flags	SUBS X1, X2, X3	X1 = X2 – X3	Subtract, set condition codes
	add immediate and set flags	ADDIS X1, X2, 20	X1 = X2 + 20	Add constant, set condition codes
	subtract immediate and set flags	SUBIS X1, X2, 20	X1 = X2 – 20	Subtract constant, set condition codes
	multiply	MUL X1, X2, X3	X1 = X2 × X3	Lower 64-bits of 128-bit product
	signed multiply high	SMULH X1, X2, X3	X1 = X2 × X3	Upper 64-bits of 128-bit signed product
	unsigned multiply high	UMULH X1, X2, X3	X1 = X2 × X3	Upper 64-bits of 128-bit unsigned product
	signed divide	SDIV X1, X2, X3	X1 = X2 / X3	Divide, treating operands as signed
	unsigned divide	UDIV X1, X2, X3	X1 = X2 / X3	Divide, treating operands as unsigned
Data transfer	load register	LDUR X1, [X2,40]	X1 = Memory[X2 + 40]	Doubleword from memory to register
	store register	STUR X1, [X2,40]	Memory[X2 + 40] = X1	Doubleword from register to memory
	load signed word	LDURSW X1, [X2,40]	X1 = Memory[X2 + 40]	Word from memory to register
	store word	STURW X1, [X2,40]	Memory[X2 + 40] = X1	Word from register to memory
	load half	LDURH X1, [X2,40]	X1 = Memory[X2 + 40]	Halfword memory to register
	store half	STURH X1, [X2,40]	Memory[X2 + 40] = X1	Halfword register to memory
	load byte	LDURB X1, [X2,40]	X1 = Memory[X2 + 40]	Byte from memory to register
	store byte	STURB X1, [X2,40]	Memory[X2 + 40] = X1	Byte from register to memory
	load exclusive register	LDXR X1, [X2,0]	X1 = Memory[X2]	Load; 1st half of atomic swap
	store exclusive register	STXR X1, X3, [X2]	Memory[X2]=X1;X3=0 or 1	Store; 2nd half of atomic swap
	move wide with zero	MOVZ X1,20	X1 = 20 or 20 * 2^{16} or 20 * 2^{32} or 20 * 2^{48}	Loads 16-bit constant, rest zeros
	move wide with keep	MOVK X1,20	X1 = 20 or 20 * 2^{16} or 20 * 2^{32} or 20 * 2^{48}	Loads 16-bit constant, rest unchanged
Logical	and	AND X1, X2, X3	X1 = X2 & X3	Three reg. operands; bit-by-bit AND
	inclusive or	ORR X1, X2, X3	X1 = X2 | X3	Three reg. operands; bit-by-bit OR
	exclusive or	EOR X1, X2, X3	X1 = X2 ^ X3	Three reg. operands; bit-by-bit XOR
	and immediate	ANDI X1, X2, 20	X1 = X2 & 20	Bit-by-bit AND reg with constant
	inclusive or immediate	ORRI X1, X2, 20	X1 = X2 | 20	Bit-by-bit OR reg with constant
	exclusive or immediate	EORI X1, X2, 20	X1 = X2 ^ 20	Bit-by-bit XOR reg with constant
	logical shift left	LSL X1, X2, 10	X1 = X2 << 10	Shift left by constant
	logical shift right	LSR X1, X2, 10	X1 = X2 >> 10	Shift right by constant
Conditional branch	compare and branch on equal 0	CBZ X1, 25	if (X1 == 0) go to PC + 4 + 100	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	CBNZ X1, 25	if (X1!= 0) go to PC + 4 + 100	Not equal 0 test; PC-relative
	branch conditionally	B.cond 25	if (condition true) go to PC + 4 + 100	Test condition codes; if true, branch
Unconditional jump	branch	B 2500	go to PC + 4 + 10000	Branch to target address; PC-relative
	branch to register	BR X30	go to X30	For switch, procedure return
	branch with link	BL 2500	X30 = PC + 4; PC + 4 + 10000	For procedure call PC-relative

Tradução ARM -> LEGv8

➜ Problema: nem todas as instruções são perfeitamente conversíveis

➜ Existência de instruções quais a arquitetura LEG é incapaz de replicar perfeitamente, mas cuja as diferenças são geralmente irrelevantes para o programa

➜ Ex: PRFM, PRFUM, LDNP

➜ Existência de instruções não adaptáveis pela arquitetura LEG

➜ Ex: Instruções SIMD, Instruções de ponto flutuante, Interrupções de sistema operacional, operações de shift variável

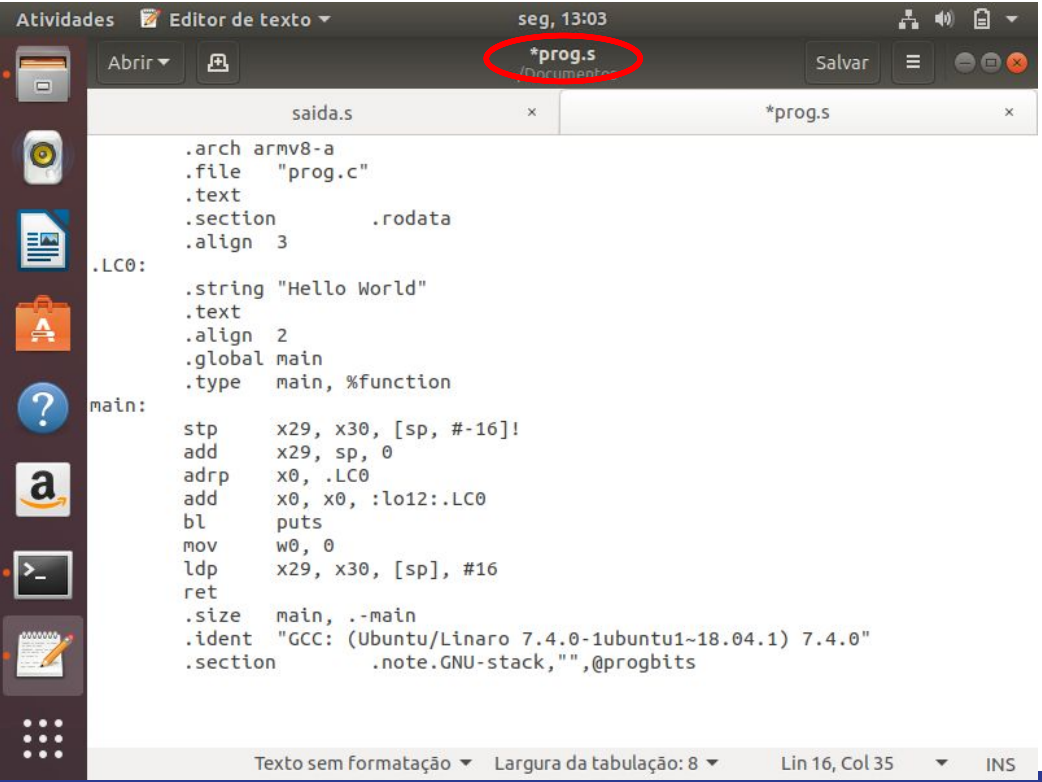
Tradução ARM -> LEGv8

➜ Conjunto das traduções obtidas implementada em um script de Python 3.7

➜ Programa lê palavra por palavra e quando encontra instrução a ser convertida a substitui pelo equivalente definido

```
def convert_1_to_3(x):
    if x[0] == 'sbc':
        x1 = ['add',x[1],x[2],'HS']
        x2 = ['sub',x[1],x[1],x[3]]
        x3 = ['subi',x[1],x[1],'#1']
    elif x[0] == 'sbcs':
        x1 = ['add',x[1],x[2],'HS']
        x2 = ['sub',x[1],x[1],x[3]]
        x3 = ['subi',x[1],x[1],'#1']
    elif x[0] == 'ngc':
        x1 = ['add',x[1],'WZR','HS']
        x2 = ['sub',x[1],x[1],x[2]]
        x3 = ['subi',x[1],x[1],'#1']
    elif x[0] == 'ngcs':
        x1 = ['add',x[1],'WZR','HS']
        x2 = ['subi',x[1],x[1],'#1']
        x3 = ['subs',x[1],x[1],x[2]]
```

Arquivo Assembly para LEGv8

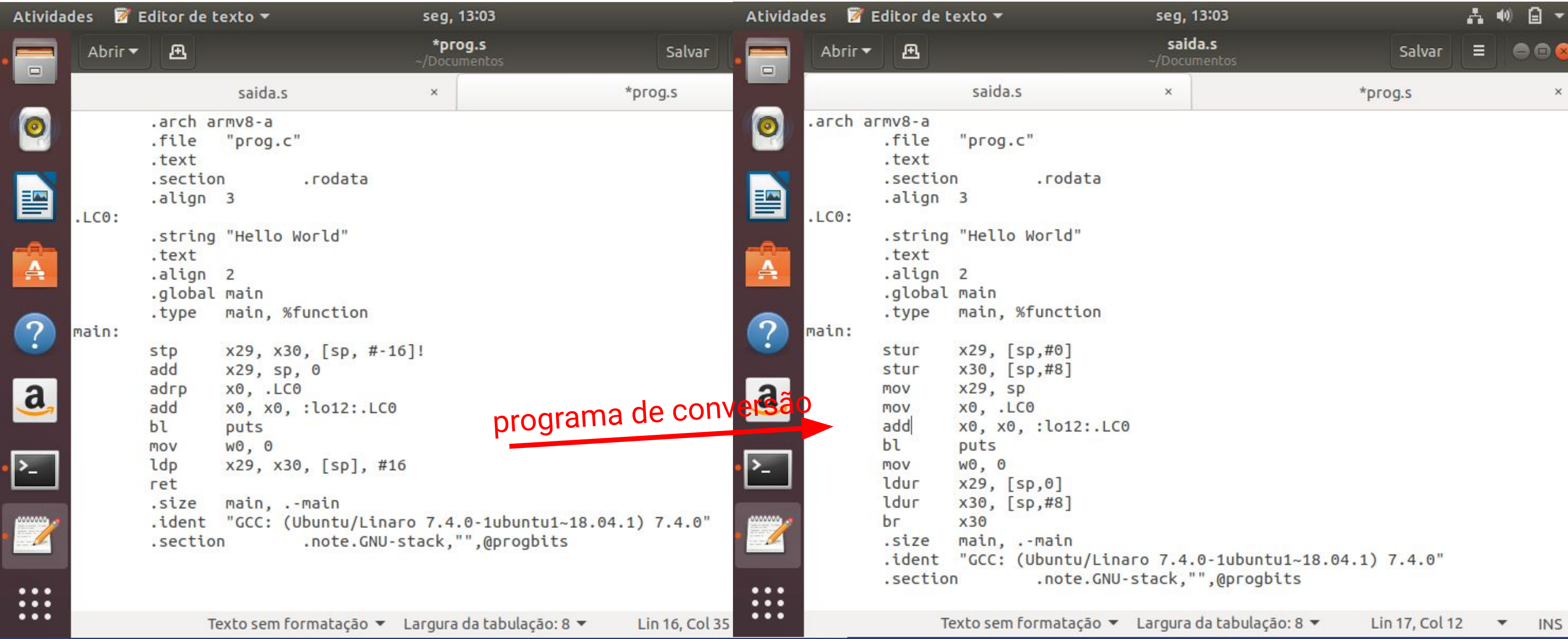


Arquivo Assembly para LEGv8

```
	.arch armv8-a
	.file	"prog.c"
	.text
	.section	.rodata
	.align	3
.LC0:
	.string "Hello World"
	.text
	.align	2
	.global main
	.type	main, %function
main:
	stp	x29, x30, [sp, #-16]!
	add	x29, sp, 0
	adrp	x0, .LC0
	add	x0, x0, :lo12:.LC0
	bl	puts
	mov	w0, 0
	ldp	x29, x30, [sp], #16
	ret
	.size	main, .-main
	.ident	"GCC: (Ubuntu/Linaro 7.4.0-1ubuntu1~18.04.1) 7.4.0"
	.section	.note.GNU-stack,"",@progbits
```

programa de conversão

Arquivo Assembly para LEGv8



Arquivo .MIF

➜ Memory Initialization File

Arquivo .MIF

- ➜ Memory Initialization File
- ➜ Input para inicialização da memória em compiladores e simuladores

Arquivo .MIF

➜ Memory Initialization File

➜ Input para inicialização da memória em compiladores e simuladores

➜ Contém todos os valores iniciais para cada endereço da memória a ser inicializado

Arquivo .MIF

```
DEPTH = 32;                % Memory depth and width are required   %
                           % DEPTH is the number of addresses      %
WIDTH = 14;                % WIDTH is the number of bits of data per word %
%  DEPTH and WIDTH should be entered as decimal numbers            %

ADDRESS_RADIX = HEX;       % Address and value radixes are required  %
DATA_RADIX = HEX;          % Enter BIN, DEC, HEX, OCT, or UNS; unless  %
                           % otherwise specified, radixes = HEX    %

-- Specify values for addresses, which can be single address or range
CONTENT
BEGIN
[0..F]:    3FFF;    % Range--Every address from 0 to F = 3FFF  %
6      :   F;       % Single address--Address 6 = F %
8      :   F E 5;   % Range starting from specific address %
--                  % Addr[8] = F, Addr[9] = E, Addr[A] = 5 %
END;
```

Geração do Arquivo .MIF

➜ Obtenção de arquivo em Assembly .s com instruções LEGv8

Geração do Arquivo .MIF

➜ Obtenção de arquivo em Assembly .s com instruções LEGv8

➜ Obtenção de arquivo objeto .o do mesmo usando o toolchain aarch64 no arquivo.s obtido

```
amanda@amanda-VirtualBox:~/Documentos$ aarch64-linux-gnu-gcc -c saida.s
amanda@amanda-VirtualBox:~/Documentos$ 
```

Geração do Arquivo .MIF

➜ Obtenção de arquivo em Assembly .s com instruções LEGv8

➜ Obtenção de arquivo objeto .o do mesmo usando o toolchain aarch64 no arquivo.s obtido

```
amanda@amanda-VirtualBox:~/Documentos$ aarch64-linux-gnu-gcc -c saida.s
amanda@amanda-VirtualBox:~/Documentos$
```

➜ Conversão para arquivo .MIF através de programa em Python

Entendendo o formato ELF

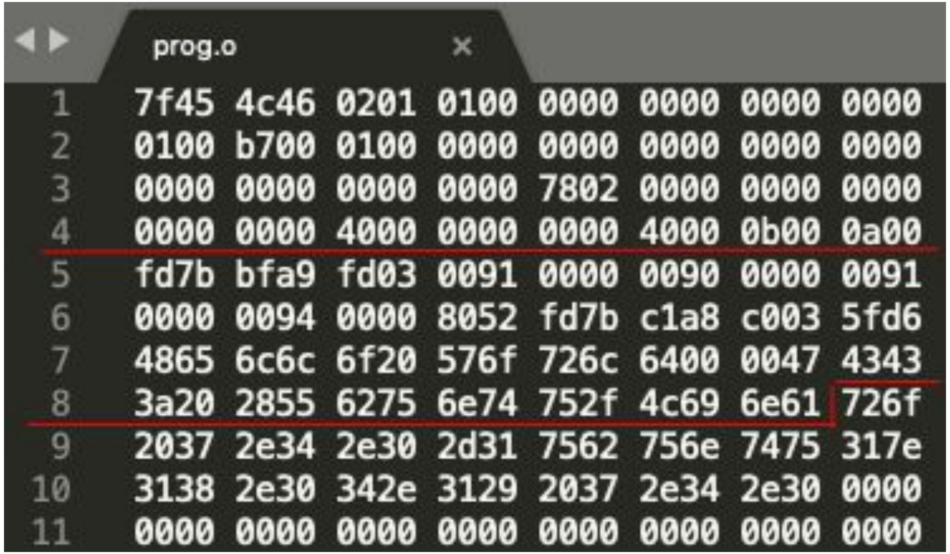
➜ Arquivo EOF possui 3 Headers diferentes, com várias configurações em cada um.

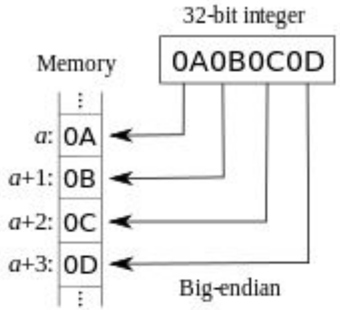
➜ Arquivos ELF são os *outputs* de praticamente todos os compiladores para Linux

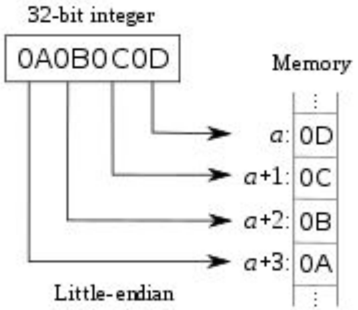
➜ Pode ser um arquivo sem extensão ou com extensão: *.axf, .bin, .elf, .o, .prx, .puff, .ko, .mod* ou *.so*

Formato ELF: Informações nos headers

➜ Dentre as configurações relevantes estão: Magic numbers, 32/64 bits, *big or little endianness*, SO, tipo de *instruction set*, entre muitas outras.







Geração do Arquivo .MIF - Código (1/6)

```python
def get_file_name():
    try:
        file_name = sys.argv[1]
    except IndexError as e:
        print("\nError: pass a file name as an argument to this program!\n")
        print("Try something like: python " + sys.argv[0] + " program.o\n")
        logging.exception("Error: pass a file name as an argument to this program!")
        raise e
    else:
        elf_extensions = ["o", "axf", "bin", "elf", "prx", "puff", "ko", "mod", "so"]
        if len(file_name.split(".")) == 1 or file_name.split(".")[-1].lower() not in elf_extensions:
            print("\nWarning: file given may not be an Executable and Linkable Format (ELF)!\n")
            logging.warning("file given may not be an Executable and Linkable Format (ELF)!")

    return file_name


def open_file(file_name):

    # Leitura do arquivo em Python 2
    if sys.version_info[0] < 3:
        with open(file_name, 'r') as f:
            hex_str_list = ["{:02x}".format(ord(c)) for c in f.read()]

    # Leitura do arquivo em Python 3
    else:
        with open(file_name, 'rb') as f:
            hex_str_list = ["{:02x}".format(c) for c in f.read()]

    hex_list = [hex(int(x, 16)) for x in hex_str_list]
    # logging.debug("hex_list = " + str(hex_list))

    return hex_str_list, hex_list
```

Geração do Arquivo .MIF - Código (2/6)

```python
file_header_dict_64bits = {

    # This dictionary contains the File Header specifications for 64 bits configurations
    # Contents ordered in this fashion:
    #   addr: (size_in_bytes, [functions], (arguments)),
    # or
    #   addr: (size_in_bytes, function, (name, {dictionary})),

    # 0x7F followed by ELF(45 4c 46) in ASCII; these four bytes constitute the magic number.
    0x00: (4, ["equals", "and_func"], (0x7f, 0x45, 0x4c, 0x46)),

    # This byte is set to either 1 or 2 to signify 32- or 64-bit format, respectively.
    0x04: (1, "add_value_to_dict", ("EI_CLASS", {1: "32", 2: "64"})),

    # This byte is set to either 1 or 2 to signify little or big endianness, respectively.
    # This affects interpretation of multi-byte fields starting with offset 0x10.
    0x05: (1, "add_value_to_dict", ("EI_DATA", {1: "little", 2: "big"})),

    # Set to 1 for the original and current version of ELF.
    0x06: (1, "add_value_to_dict", ("EI_VERSION", {0: "", 1: "original"})),

    # Identifies the target operating system ABI.
    # It is often set to 0 regardless of the target platform.
    0x07: (1, "add_value_to_dict", ("EI_OSABI", {
        0x00: "System V",
        0x01: "HP-UX",
        0x02: "NetBSD",
```

Geração do Arquivo .MIF - Código (3/6)

```
def parse_dict(hex_str_list, hex_list, CONFIGURATION_DICT, file_dict_32bits, file_dict_64bits, offset=0)
    PC = 0x00
    eof = False
    while not eof:
        if CONFIGURATION_DICT["EI_CLASS"] == '32':
            size, functions, values_tuple = file_dict_32bits[PC]
        else:
            size, functions, values_tuple = file_dict_64bits[PC]

        focused_bytes = hex_str_list[PC + offset:PC + offset + size]
        int_focused_bytes = [int(x, 16) for x in hex_list[PC + offset:PC + offset + size]]

        # EOF
        if size == 0:
            logging.info("Reached EOF.")
            break

        logging.info("PC = " + str(hex(PC)))
        logging.info("PC + offset = " + str(hex(PC + offset)))
        logging.info("focused_bytes = " + str(focused_bytes))
        logging.info("int_focused_bytes = " + str(int_focused_bytes))

        if type(functions) == list:
            map_function, reduce_function = functions

            logging.info("values_tuple = " + str(values_tuple))
            logging.debug("map_list = " + str(list(map(getattr(AssertingTools, map_function), int_focuse

            result = reduce(getattr(AssertingTools, reduce_function), map(getattr(AssertingTools, map_fu
            logging.info("result = " + str(result))

        else:
            logging.info("values_tuple = " + str(values_tuple))
```

Geração do Arquivo .MIF - Código (4/6)

```
def parse_dict(hex_str_list, hex_list, CONFIGURATION_DICT, file_dict_32bits, file_dict_64bits, offset=0)
    PC = 0x00
    eof = False
    while not eof:
        if CONFIGURATION_DICT["EI_CLASS"] == '32':
            size, functions, values_tuple = file_dict_32bits[PC]
        else:
            size, functions, values_tuple = file_dict_64bits[PC]

        focused_bytes = hex_str_list[PC + offset:PC + offset + size]
        int_focused_bytes = [int(x, 16) for x in hex_list[PC + offset:PC + offset + size]]

        # EOF
        if size == 0:
            logging.info("Reached EOF.")
            break

        logging.info("PC = " + str(hex(PC)))
        logging.info("PC + offset = " + str(hex(PC + offset)))
        logging.info("focused_bytes = " + str(focused_bytes))
        logging.info("int_focused_bytes = " + str(int_focused_bytes))

        if type(functions) == list:
            map_function, reduce_function = functions

            logging.info("values_tuple = " + str(values_tuple))
            logging.debug("map_list = " + str(list(map(getattr(AssertingTools, map_function), int_focuse

            result = reduce(getattr(AssertingTools, reduce_function), map(getattr(AssertingTools, map_fu
            logging.info("result = " + str(result))

        else:
            logging.info("values_tuple = " + str(values_tuple))
```

Geração do Arquivo .MIF - Código (5/6)

```
        else:
            logging.info("values_tuple = " + str(values_tuple))

            key, dictionary = values_tuple
            function = getattr(AssertingTools, functions)
            logging.info("key = " + str(key))
            logging.info("function: " + functions)

            multi_bytes = multi_byte_concat(CONFIGURATION_DICT, focused_bytes)
            logging.info("multi_bytes = " + str(multi_bytes))

            value = dictionary.get(int(multi_bytes, 16))
            logging.info("value = " + str(value))

            CONFIGURATION_DICT = function(config_dictionary=CONFIGURATION_DICT, key=key, value=value, in
            logging.debug("CONFIGURATION_DICT = " + str(CONFIGURATION_DICT))

            # print(CONFIGURATION_DICT)

        PC += size

        print("PC = " + str(PC) + " (" + str(hex(PC)) + "); with offset: " + str(PC + offset) + " (" + s

    return CONFIGURATION_DICT
```

Geração do Arquivo .MIF - Código (6/6)

```
# Section Header
section_header_offset = program_header_offset + program_header_size
section_header_size = int(CONFIGURATION_DICT["e_shentsize"], 16)

if section_header_size > 0:
    CONFIGURATION_DICT = parse_dict(hex_str_list, hex_list, CONFIGURATION_DICT, section_header_dict_
# /Section Header

# Reading Data
data_offset = section_header_offset + section_header_size
data_hex = hex_str_list[data_offset:]

word_size = int(int(CONFIGURATION_DICT['EI_CLASS']) / 8)

eof = False
parsed_instructions = []
while not eof:
    if data_offset + word_size < len(data_hex):
        multi_byte = multi_byte_concat(CONFIGURATION_DICT, data_hex[data_offset:data_offset + word_s
    else:
        multi_byte = multi_byte_concat(CONFIGURATION_DICT, data_hex[data_offset:])
        eof = True

    parsed_instructions.append(multi_byte)
    data_offset += word_size
# /Reading Data

# MIF Convertion
write_mif_file(CONFIGURATION_DICT, file_name, parsed_instructions)
```

Arquivo .MIF

```
DEPTH = 64;
WIDTH = 8;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;
CONTENT
BEGIN
0    :    0004000300000000;
1    :    0000000000000000;
2    :    0000000000000000;
3    :    0005000300000000;
4    :    0000000000000000;
5    :    0000000000000000;
6    :    0005000000000008;
7    :    0000000000000000;
8    :    0000000000000000;
9    :    000100000000000b;
a    :    0000000000000000;
b    :    0000000000000000;
c    :    0007000300000000;
d    :    0000000000000000;
e    :    0000000000000000;
f    :    0006000300000000;
10   :    0000000000000000;
11   :    0000000000000000;
12   :    000100120000000e;
13   :    0000000000000000;
14   :    0000000000000020;
15   :    0000001000000013;
16   :    0000000000000000;
17   :    0000000000000000;
18   :    00632e676f727000;
19   :    616d007824006424;
1a   :    0073747570006e69;
1b   :    0000000000000008;
1c   :    0000000500000113;
1d   :    0000000000000000;
```

Obrigado!

GRUPO D

Amanda Gales - 9853356

Daniel Lavedonio - 8992882

Luiz Victorio Martins - 9349311

Rodrigo Sakai - 9351183