ElfoViewer - Documentação

Integrantes:

Isabelle Ritter Vargas - NUSP 11806600

Matheus Rezende Pereira - NUSP 11261805

Repositório no GitHub aqui

Link do vídeo do youtube aqui

1. Sobre o projeto

O projeto, aqui denominado de *ElfoViewer*, consiste em uma ferramenta didática e interativa para visualizar o conteúdo da memória do processador. A motivação veio do interesse de entender como seria a distribuição das instruções dentro da memória enquanto o programa executava, igual nesse vídeo. Entretanto, por algumas dificuldades apontadas pelo professor em relação ao qemu, o grupo optou por desenvolver essa ferramenta de forma estática. Assim, com o ElfoViewer, é possível visualizar o conteúdo de um arquivo .elf de qualquer programa que você quiser! O objetivo é entender como os diferentes tipos de instruções estão distribuídos a depender do programa em execução.

2. Classificações de instruções

Um dos passos fundamentais do ElfoViewer é definir os grupos de instruções. Para isso, foram analisados mais de 700 instruções e classificadas de acordo com o que o grupo achou pertinente.

A relação atual está na tabela abaixo:

Grupo	Exemplos de instruções (**não exaustivo**)
Instruções de acesso à memória	str, ldr
Instruções para coprocessadores	ldc, mrc
Instruções de ULA	add, sub, eor
Instruções de acesso à memória múltiplo	ldm, stm
Instruções de operações com vetores	vadd, vstr
Instruções de desvio	b, beq, bl, bx

Instruções de ULA com PF	adf, dvf
Instruções para modo supervisor	svc
Instruções de acesso a memória com PF	stf, ldf
Pseudo instruções para pilha	push, pop
Pseudo instruções para rotações	ror, asr, lsr
Nop	nop

3. Renderização dos pixels

Para renderizar os pixels utilizou-se a biblioteca PyGame, que é comumente utilizada para manipulação de gráficos e interação com a tela, em conjunto com Python, linguagem na qual implementamos a nossa ferramenta didática.

Vamos seguir para olhar mais a fundo como é o processo de renderização dos pixels.

drawVirtPixel

Essa função é responsável pelo desenho de um pixel na tela, na superfície *surface*, de cor *color*, tamanho *newSize* e com as posições de início *xOrigin* e *yOrigin.*

```
def drawVirtPixel(surface, xOrigin, yOrigin, color, newSize):
    for x in range(newSize):
        for y in range(newSize):
            if (x==0 or y==0 or x==newSize-1 or y ==newSize-1)
                gfxdraw.pixel(surface, (newSize*xOrigin)+x, (newSize*yOrigin)+y, color)
            else:
                gfxdraw.pixel(surface, (newSize*xOrigin)+x, (newSize*yOrigin)+y, color)
```

get_instruction_by_memory_index e get_instruction_by_address

Essas duas funções são bastante parecidas, e basicamente buscam uma instrução com base em um index ou em um endereço. Ou seja, são úteis quando precisamos mapear qual é a próxima instrução que será desenhada na tela.

```
def get_instructio_by_memory_index(lst, index):
    for item in lst:
        if 'memory_index' in item and item['memory_index'] == index:
            return item
    return "No relative Index"  # Return None if the item with the specified index is not found

def get_instruction_by_address(lst, address):
    for item in lst:
        if 'address' in item and item['address'] == address:
            return item
    return "No address"  # Return None if the item with the specified index is not found
```

coord

Função que é responsável, com base no tamanho da tela (*screenX, screenY*) e no tamanho do pixel (*virtPixelSize*), por retornar qual é a coordenada (x,y) relacionada a aquela posição.

```
def coord(pos, screenX, screenY, virtPixelSize):
    y=pos//(screenY//virtPixelSize)
    x=pos%(screenY//virtPixelSize)
    return (x,y)
```

fillRegion e drawPixelRelative

Essas duas funções são responsáveis por desenhar um pixel, sendo que a *drawPixelRelative* utiliza a *fillRegion* para preencher esse espaço de um pixel (por meio da função básica *drawVirtPixel*.

```
def fillRegion(surface, start, end, color, screenX, screenY, virtPixelSize):
    size = end - start
    lineCount = size//(screenY//virtPixelSize)
    for pos in range(start, end):
        x, y = coord(pos, screenX, screenY, virtPixelSize)
        drawVirtPixel(surface, x, y, color, virtPixelSize)

def drawPixelRelative(surface, spot, color, screenX, screenY, virtPixelSize):
    fillRegion(surface, spot, spot+1, color, screenX, screenY, virtPixelSize)
```

display_dialogue_box

Função que renderiza as labels que surgem quando você clica em um pixel, indicando qual é a instrução que ela está simbolizando.

```
def display_dialogue_box(text, position, screen):
    font = pygame.font.Font(None, 24)
    dialogue_text = font.render(text, True, (0, 0, 0))
    screen.set_at(position, (255, 255, 255))
    ## Draws the box to the left of the cursor if beyond middle of screen
    if position[0] > screen.get_width() / 2:
        dialogue_box = dialogue_text.get_rect(topright=position)
    else:
        dialogue_box = dialogue_text.get_rect(topleft=position)


    # Draws the box background
    box_width = dialogue_text.get_width() + 10
    box_height = dialogue_text.get_height() + 10
    if position[0] > screen.get_width() / 2:
        dialogue_box_background = pygame.Rect((position[0]-box_width+10, position[1]+2), (box_width, bo
x_height))
    else:
        dialogue_box_background = pygame.Rect(position, (box_width, box_height))
    pygame.draw.rect(screen, (255,255,255), dialogue_box_background)

    # Draws the text
```

```
    screen.blit(dialogue_text, dialogue_box.move(5, 5))
    pygame.display.flip()
```

highlight_virtpixel_border

Função que desenha uma borda no pixel desejado.

```
def highlight_virtpixel_border(surface, virtPixelSize, color, index, screenX):
    line = (index//(screenX//virtPixelSize))
    column = (index%(screenX//virtPixelSize))
    xCorner = column * virtPixelSize
    yCorner = line * virtPixelSize
    # Draws a border around the pixel starting in xCorner, yCorner with virtPixelSize as a side and usi
ng pygame rect
    pygame.draw.rect(surface, color, (xCorner, yCorner, virtPixelSize, virtPixelSize), 2)
    return xCorner, yCorner, xCorner+virtPixelSize, yCorner+virtPixelSize
```

4. Leitura de arquivos `.elf`

A leitura dos arquivos `.elf` é feita principalmente utilizando-se expressões regulares. Isso pode ser notado ao ínicio da função de parsing `parse_objdump`, contida no arquivo `objDumpParser.py`:

```
def parse_objdump(file_name):
    with open(file_name, 'r') as file:
        lines = file.readlines()

    instruction_pattern = re.compile(r'\s*([0-9a-f]+):\s*([0-9a-f]+)\s*(\S+)\s+(.*)')

    instructions = []
    index = 0
```

A expressão utilizada busca criar um padrão de match que vai filtrar endereço, valor, instrução e conteúdo da linha, sendo que ele é aplicado a todas as linhas do código e antes de se iniciar o loop busca a primeira linha contendo instruções para encontrar o endereço de início da memória para organizar a renderização.

Em seguida, a expressão é aplicada a cada linha do arquivo, gerando um dicionário que contém todas as informações necessárias para a renderização. (Note que é retornado uma lista de dicionários e cada um é uma instrução com suas informações.)

```
for line in tqdm(lines, desc='Building dict', unit='line'):
        match = instruction_pattern.match(line)
        if match:
            address, value, instruction, content = match.groups()
            identifier_value = format(int(value[1], 16), '04b')
            whole_instruction = format(int(value, 16), '032b')
            group = nameChooser(instruction)
            memory_index = int(((int(address, 16) - starting_address))/4)
```

```
            instructions.append({
                'index': index,                         # General dict index
                'memory_index': memory_index,           # Memory index, relative to the processor's memory
                'address': address,                     # Instruction address in hex
                'instruction': instruction,             # Instruction mnemonic
                'value': value,                         # Instruction value in hex
                'whole_binary': whole_instruction,      # Instruction value in binary
                'group': group,                         # General instruction macro group
                'content': content,                     # Instruction content
            })
            index += 1
print("Done!\n")
filtered_instructions = [item for item in instructions if item.get("instruction") != ";"]
return instructions
```

Portanto, com este método é possível entregar aos métodos de renderização as informações necessárias para a disposição e diferenciação das instruções na tela de maneira escalável e programática.

5. Montagem final do visualizador de `.elf`

A montagem da vista final é realizada no arquivo `main.py`, que será abordado em duas grandes etapas nesta sessão por se tratar de um arquivo extenso e que chama boa parte dos métodos desenvolvidos.

Setup

Primeiro é realizado o setup dos parâmetros de renderização com base nos parâmetros fornecidos (ou omitidos) pelo usuário. As características definidas são:

Parâmetros de renderização

```
default_params = "average"
params_arg = sys.argv[1] if len(sys.argv) > 1 else default_params
try:
    # Load the selected theme module dynamically
    global virtPixelSize, screenX, screenY, fillColor, borderColor
    param_module = importlib.import_module(f"renderParams.{params_arg}")
    virtPixelSize = param_module.renderParams["virtPixelSize"]
    screenX = param_module.renderParams["screenX"]
    screenY = param_module.renderParams["screenY"]
    fillColor = param_module.renderParams["fillColor"]
    borderColor = param_module.renderParams["borderColor"]
except ImportError:
    print("Invalid render parameter selection.")
    sys.exit(1)
```

Definem os parâmetros da tela renderizada pelo pygame, como o tamanho virtual dos pixels (`virtualPixelSize`, ou vps), que define o **lado de cada instrução na imagem**. Além disso, também é

aqui que se define o tamanho da janela, que **não é dinâmico** devido a forma como a imagem é renderizada, portanto precisa ser dimensionado adequadamente para o dump analisado. (Os 4 tamanhos fornecidos no código tendem a ser adequados para qualquer dump, só é importante se atentar ao realizar a chamada do programa.)

Parâmetros de disposição

```
default_render = "relative"
render_arg = sys.argv[2] if len(sys.argv) > 2 else default_render
if render_arg == "relative":
    renderStyle = 'memory_index'
elif render_arg == "linear":
    renderStyle = 'index'
```

Referente ao segundo argumento do programa, ele define como as instruções são posicionadas na imagem, se relativa a sua posição na memória ou linear, sequencialmente na imagem.

Parâmetros de tema

```
default_theme = "DebuggersDream"
theme_arg = sys.argv[3] if len(sys.argv) > 3 else default_theme
try:
    # Load the selected theme module dynamically
    theme_module = importlib.import_module(f"themes.{theme_arg}")
    groupColor = theme_module.theme
    theme_name = theme_module.name
except ImportError:
    print("Invalid theme selection.")
    sys.exit(1)
```

Escolhe o tema da pasta de temas, apenas estético e é um parâmetro opcional.

Renderização

Em seguida, é impresso no terminal um display de texto e o código constrói uma superfície com o display das instruções, de fato renderizando (mas não dispondo) a imagem final:

```
count = 0
DISPLAYSURF.fill(groupColor['background'])
for instruction in tqdm(instructions, desc="Rendering and creating table",unit="instructions"):
    count += 1
    color = groupColor[instruction['group']]
    try:
        group_ammounts[group_types.index(instruction['group'])] += 1
    except Exception as e:
        pass
    dp.drawPixelRelative(DISPLAYSURF, instruction[renderStyle], color, screenX, screenY, virtPixelSize)
AUXSURF = DISPLAYSURF.copy()
print("\n")
```

```
group_ammounts[-1] = count
print("Group ammounts:", group_ammounts)
group_percentages = []
for i in group_ammounts[:13]:
    group_percentages.append(str("{:.4f}".format((i/group_ammounts[-1])*100)) + "%")
print(tabulate([group_ammounts, group_percentages], headers=group_types, tablefmt="fancy_grid"))
```

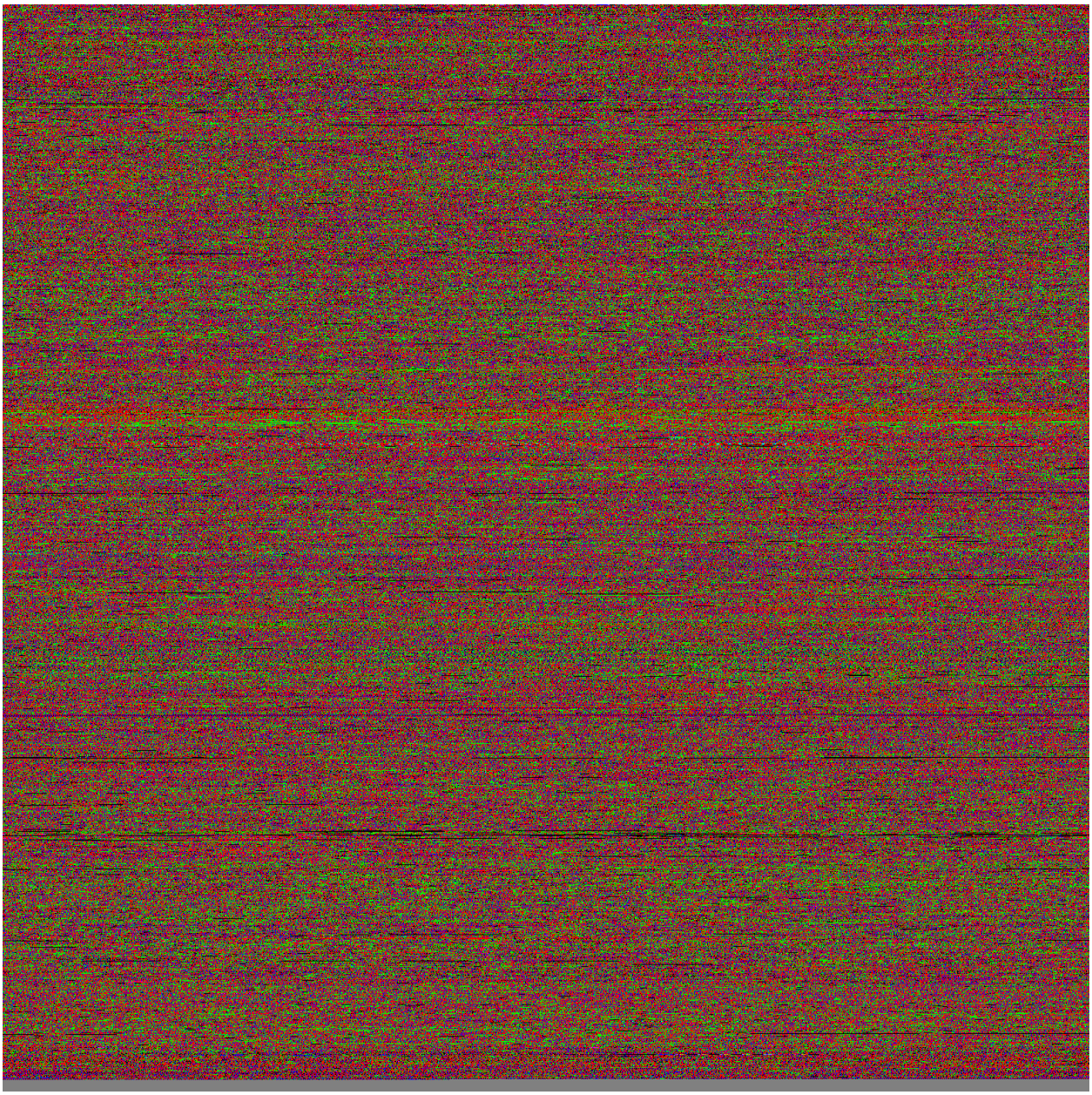
Vale notar também que aqui é construída a tabela para mostrar as estatísticas do programa.

Gerenciamento de janela e funções

Com a tela renderizada e armazenada em uma variável, o loop central do programa só é encarregado de dispor a imagem ao usuário e gerenciar as interações com a tela, mostrando os diálogos e atualizando a tela com as informações requisitadas, sejam elas informações da instrução selecionada, a legenda das cores, ou o o fluxo de branches.

6. Exemplos mapeados

Linux



Acesso a memoria	Desvios	ULA	Coprocessadores	Acesso a memoria multiplo	Acesso a memoria com PF	UNDEFINED	Pilha	ULA com PF	Modo supervisor	Operacoes com vetores	Nop	Rotacoes	Total
589710	492606	1020566	2777	7536	9427	224913	76326	0	1	2	228	13193	2437285
24.1954%	20.2113%	41.8731%	0.1139%	0.3092%	0.3868%	9.2280%	3.1310%	0.0000%	0.0000%	0.0001%	0.0094%	0.5413%	

Aqui mapeamos o .elf do kernel do linux. Olhando a imagem e as estatísticas percebe-se uma presença muito forte de instruções de ULA (41,87%), seguido por instruções de acesso à memória (24,19%) e de instruções de desvio (20,21%), como esperado. Percebe-se também que elas estão espalhadas de forma uniforme, ainda que em alguns áreas tenha a predominância de um tipo de instrução.

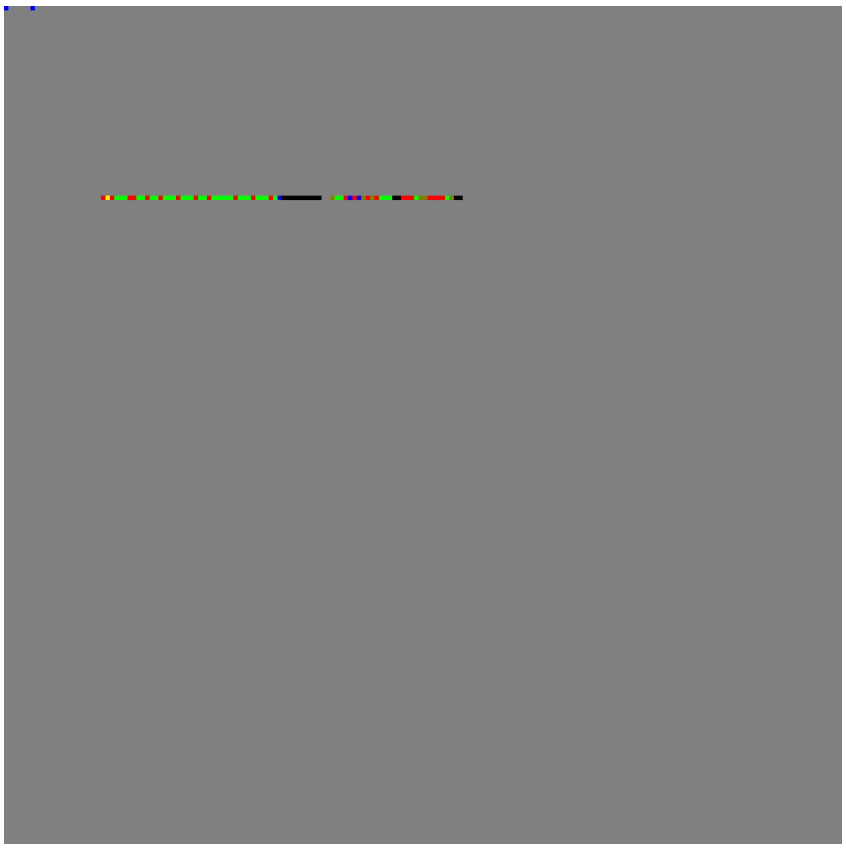
Esse exemplo é interessante para mostrar uma exemplo real, com uma magnitude muito maior do que os próximos exemplos.

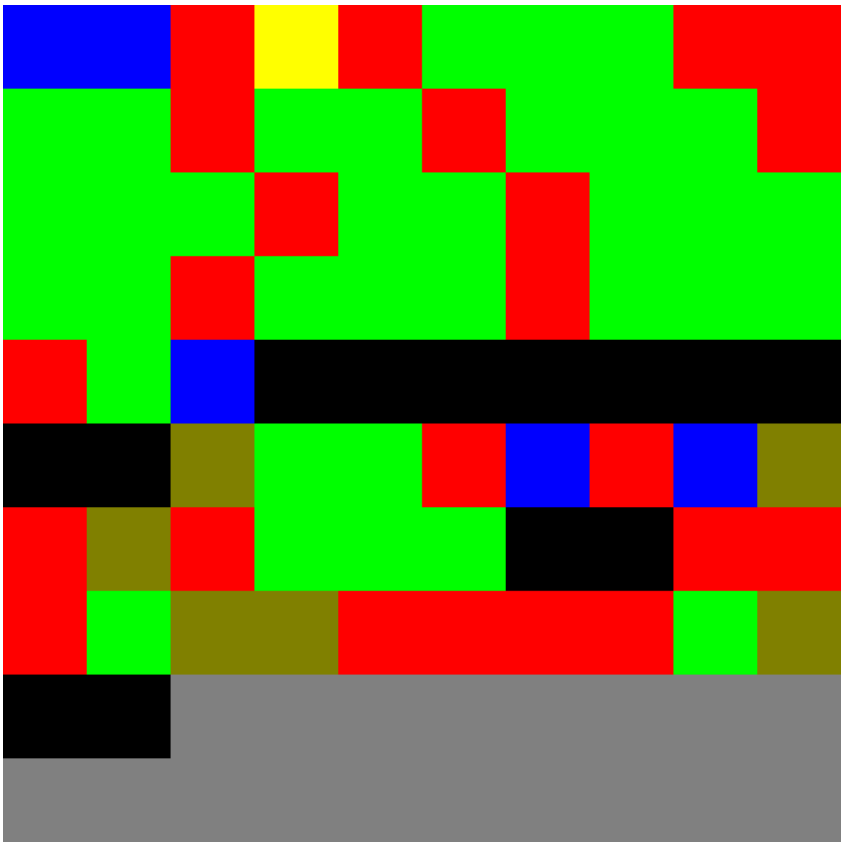
Rotina de controlar leds da placa Evaluator

Aqui utilizamos um dos códigos para controlar leds na placa Evaluator. Aqui temos a memória mapeada de duas formas: relativa e absoluta.

Na primeira imagem, conseguimos ver o vetor de interrupções lá no início, nos primeiros endereços, enquanto que o resto do código segue em uma outra região da memória.

Já na segunda imagem as posições estão mapeadas todas juntas, com o vetor de interrupções ao lado das instruções de código regulares.



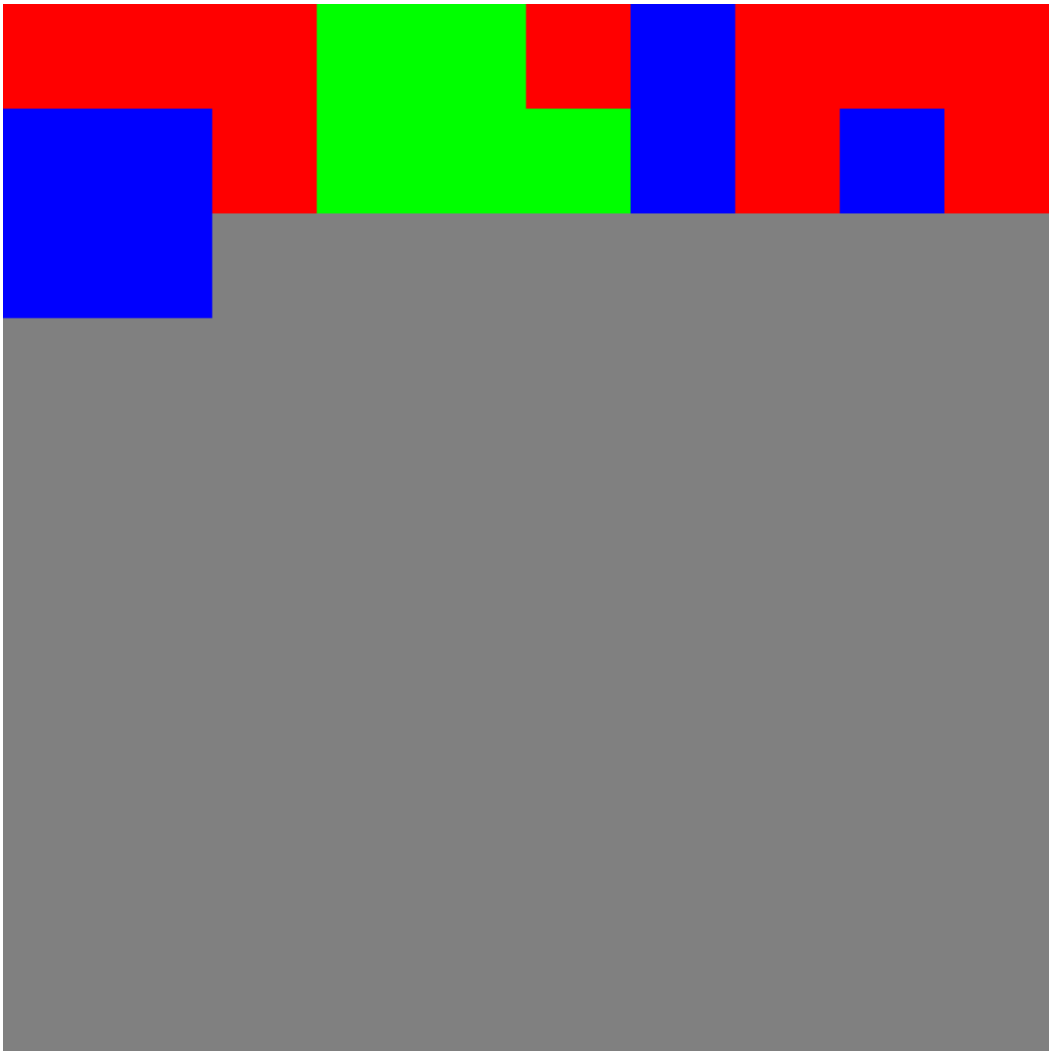


Acesso a memoria	Desvios	ULA	Coprocessadores	Acesso a memoria multiplo	Acesso a memoria com PF	UNDEFINED	Pilha	ULA com PF	Modo supervisor	Operacoes com vetores	Nop	Rotacoes	Total
34	5	23	1	0	0	13	6	0	0	0	0	0	82
41.4634%	6.0976%	28.0488%	1.2195%	0.0000%	0.0000%	15.8537%	7.3171%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	

Nesse exemplo, a maior parte das instruções se agrupa em instruções de acesso à memória (41,40%)

B-sort feito em sala

Por fim, nesse último exemplo, ainda bastante simples, temos um código B-sort feito em sala. Diferente do anterior, não foi usado o Evaluator. Por ser bem simples, percebe-se que o foco maior está em instruções de ULA (45,45%), seguido por desvios (31,81%) e acesso à memória (22,72%)



Acesso a memoria	Desvios	ULA	Coprocessadores	Acesso a memoria multiplo	Acesso a memoria com PF	UNDEFINED	Pilha	ULA com PF	Modo supervisor	Operacoes com vetores	Nop	Rotacoes	Total
5	7	10	0	0	0	0	0	0	0	0	0	0	22
22.7273%	31.8182%	45.4545%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	