Campus de Araranguá
Curso de Engenharia de Computação

Emanuela Schmitz, Gabrieli Barbosa, Leticia Veran e Valentina Leiria

RELATÓRIO DE TRABALHO MIPS

Projeto 05 - Elevadores

Araranguá/SC
2025

RESUMO

Este relatório detalha o desenvolvimento de um sistema embarcado para o controle de dois elevadores em uma edificação de oito pavimentos, utilizando a linguagem Assembly MIPS. O principal objetivo do projeto é otimizar o uso dos elevadores, buscando a eficiência energética ao priorizar que o elevador mais próximo atenda à solicitação de serviço, visando a redução do tempo de operação e do consumo de energia. A interação com o sistema ocorre por meio de uma ferramenta de simulação, permitindo a entrada de comandos via teclado hexadecimal (andares de 0 a 7, e direções C para subir e B para descer) e a visualização da posição dos elevadores através de displays de sete segmentos (direito para Elevador A, esquerdo para Elevador B). A arquitetura de programação é orientada a componentes, empregando chamadas de procedimentos para assegurar uma implementação modular e organizada. O sistema também inclui validações para chamadas inválidas (ex: descer do andar 0) e uma prioridade para o Elevador A em caso de distâncias iguais. A estrutura de dados define variáveis de estado para as posições e destinos dos elevadores, mensagens de sistema para interação com o usuário, e uma tabela de consulta para os displays de 7 segmentos, abstraindo a complexidade do hardware. O ponto de entrada do programa inicializa os displays e entra em um loop principal que sonda o teclado, processa a entrada de acordo com um modelo de máquina de estados (esperando andar ou direção), e gerencia o movimento dos elevadores. Chamadas inválidas, como uma tecla não reconhecida ou uma solicitação de movimento impossível, são tratadas com mensagens de erro e o sistema retorna ao estado de espera.

Palavras-chave: Sistema embarcado. Controle de elevadores. Assembly MIPS. Displays de sete segmentos.

1. INTRODUÇÃO

Este relatório tem como propósito apresentar o desenvolvimento de um sistema embarcado para o controle de elevadores em uma edificação de oito pavimentos. O projeto visa gerenciar a operação de dois elevadores, que atendem a todos os andares do edifício. O principal objetivo é otimizar o uso dos elevadores, buscando a eficiência energética ao manter, sempre que possível, apenas um elevador em operação. Para isso, o sistema será concebido para que o elevador mais próximo atenda à solicitação de serviço, reduzindo o tempo de operação e o consumo de energia. A interação com o sistema será realizada por meio de uma ferramenta de simulação, que permitirá a entrada de comandos via teclado e a visualização da posição dos elevadores através de displays de sete segmentos. A arquitetura da programação será orientada a componentes, com o uso de chamadas de procedimentos, assegurando uma implementação modular e organizada.

2. DESENVOLVIMENTO

2.1. Análise da Estrutura de Dados e Mensagens do Sistema

Código Analisado:

```
1.  # Sistema de Controle de Elevadores com Teclado Digital Lab Sim
2.  #
3.  # INSTRUÇÕES DE USO:
4.  # 1. Digite o número do andar desejado (0-7) no teclado hexadecimal
5.  # 2. Pressione C para chamar elevador para SUBIR
6.  # 3. Pressione B para chamar elevador para DESCER
7.  # 4. O elevador mais próximo será enviado
8.  # 5. Os displays mostram a posição atual de cada elevador
9.  #    - Display direito: Elevador A
10. #    - Display esquerdo: Elevador B
11. #
12. # NOTAS:
13. # - Se ambos elevadores estão à mesma distância, o elevador A tem prioridade
14. # - O sistema impede chamadas inválidas (ex: pedir para descer do andar 0)
15. # - Cada elevador move um andar por vez com delay
16.
17. .data
18.     # Posições dos elevadores (0-7)
19.     elevator_a_pos: .word 0
20.     elevator_b_pos: .word 0
21.
22.     # Destinos dos elevadores (-1 = sem destino)
23.     elevator_a_dest: .word -1
24.     elevator_b_dest: .word -1
25.
26.     # Estado do sistema
27.     current_floor: .word -1    # Andar selecionado
28.     waiting_direction: .word 0  # 1 = esperando direção
29.
30.     # Mensagens do sistema
31.     msg_floor_prompt: .asciiz "\r\nAndar (0-7): "
32.     msg_direction_prompt: .asciiz "\r\nDirecao (C/B): "
33.     msg_elevator_a: .asciiz "\r\nElevador A: "
34.     msg_elevator_b: .asciiz "\r\nElevador B: "
35.     msg_to_floor: .asciiz " -> "
36.     msg_arrived: .asciiz "\r\nChegou! Andar "
37.     msg_invalid_floor: .asciiz "\r\nAndar invalido!"
38.     msg_invalid_call: .asciiz "\r\nChamada invalida!"
39.     msg_going_up: .asciiz " (SUBINDO)"
40.     msg_going_down: .asciiz " (DESCENDO)"
41.     msg_invalid_key: .asciiz "\r\nTecla invalida!"
```

```
42.     msg_key_pressed: .asciiz "\r\nTecla: "
43.
44.     # Tabela para display de 7 segmentos
45.     display7seg: .byte 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07
```

Análise:

O início do código-fonte é dedicado à documentação e à seção de declaração de dados (*.data*), que define a estrutura de memória fundamental para a operação do sistema. Os comentários de cabeçalho estabelecem de forma clara as regras de negócio e as instruções de uso, servindo como uma especificação funcional concisa do programa.

A seção *.data* propriamente dita aloca espaço para três categorias de informação. A primeira categoria são as variáveis de estado, declaradas com a diretiva *.word* para alocar 32 bits para cada uma. As variáveis *elevator_a_pos* e *elevator_b_pos* mantêm o registro do andar atual de cada elevador, sendo ambas inicializadas no andar térreo (0). De forma complementar, *elevator_a_dest* e *elevator_b_dest* armazenam o andar de destino de cada elevador. O valor inicial -1 é utilizado como um sinalizador (flag) estratégico para indicar que o elevador está ocioso e disponível para uma nova chamada. Ainda nesta categoria, as variáveis *current_floor* e *waiting_direction* gerenciam o estado da interface com o usuário, controlando o fluxo do processo de chamada de dois estágios (seleção de andar e, em seguida, seleção de direção).

A segunda categoria de dados compreende o conjunto de mensagens do sistema, definidas com a diretiva *.asciiz* para criar strings de texto terminadas por um caractere nulo. Essas mensagens, prefixadas com *msg_*, constituem todo o vocabulário do programa para interagir com o usuário através do console. Elas incluem prompts de entrada (como *msg_floor_prompt*), mensagens de status (como *msg_going_up*) e notificações de erro (como *msg_invalid_call*), formando uma interface de texto completa.

Por fim, a terceira categoria é uma tabela de consulta (lookup table) de hardware, *display7seg*, declarada como um array de bytes (*.byte*). Esta tabela é um componente essencial para a interface com os displays de 7 segmentos. Ela mapeia um número inteiro (de 0 a 7, correspondente ao índice do array) para o código de byte hexadecimal exato necessário para acender os segmentos corretos do display e formar o dígito visualmente. Isso abstrai a complexidade do controle de hardware, permitindo que o programa principal simplesmente consulte esta tabela para atualizar os displays com a posição atual dos elevadores.

2.2. Ponto de Entrada, Loop Principal e Lógica de Seleção de Andar

Código Analisado:

```
46.  .text
47.  .globl main
48.
49.  main:
50.      # Inicializar displays
51.      jal update_displays
52.
```

```
    # Mostrar prompt inicial
    li $v0, 4
    la $a0, msg_floor_prompt
    syscall

main_loop:
    # Ler tecla do teclado
    jal read_keyboard

    # Se não há tecla pressionada, continuar
    beqz $v0, check_elevators

    # Debug: mostrar tecla pressionada
    move $s7, $v0  # Salvar tecla

    # Verificar estado atual
    lw $t0, waiting_direction
    bnez $t0, process_direction

    # Estado: esperando andar
    # Verificar se é tecla numérica (0-7) baseado no mapeamento da imagem
    li $t0, 0x11    # tecla 0
    beq $s7, $t0, floor_0
    li $t0, 0x21    # tecla 1
    beq $s7, $t0, floor_1
    li $t0, 0x41    # tecla 2
    beq $s7, $t0, floor_2
    li $t0, 0x81    # tecla 3
    beq $s7, $t0, floor_3
    li $t0, 0x12    # tecla 4
    beq $s7, $t0, floor_4
    li $t0, 0x22    # tecla 5
    beq $s7, $t0, floor_5
    li $t0, 0x42    # tecla 6
    beq $s7, $t0, floor_6
    li $t0, 0x82    # tecla 7
    beq $s7, $t0, floor_
    # Tecla inválida
    j invalid_key
```

Análise:

A execução do programa inicia na seção *.text* com o rótulo *main*, definido como global para ser o ponto de entrada. A primeira ação do sistema é uma chamada à sub-rotina *jal update_displays*, que tem a responsabilidade de inicializar os displays de 7 segmentos com a posição inicial dos elevadores. Imediatamente após, o programa utiliza as chamadas de sistema do MIPS para imprimir na console a mensagem de boas-vindas e instrução inicial, carregada de *msg_floor_prompt*.

Após a inicialização, o controle é transferido para o *main_loop*, que constitui o ciclo de execução perpétuo e o coração do programa. A cada iteração, a sub-rotina *jal read_keyboard* é chamada para sondar o teclado. A instrução *beqz $v0, check_elevators* representa um ponto de decisão crítico: se nenhuma tecla for pressionada, o registrador de retorno *$v0* será zero, e o programa desviará o fluxo de execução para a rotina *check_elevators* (a ser analisada posteriormente), que gerencia a movimentação dos elevadores. Este design permite que o sistema execute tarefas de fundo de forma independente da interação do usuário.

Caso uma tecla seja pressionada, seu código é preservado no registrador de uso salvo *$s7*. Em seguida, o programa implementa sua lógica de máquina de estados ao verificar a flag *waiting_direction*. Se esta flag for diferente de zero, o sistema entende que está aguardando uma entrada de direção (subir/descer) e desvia o fluxo para *process_direction*. Se a flag for zero, o programa assume que a tecla pressionada corresponde a um andar e prossegue para o bloco de seleção de andar. Este bloco funciona como uma estrutura de decisão *switch-case*, onde o código da tecla em *$s7* é sequencialmente comparado com os valores hexadecimais pré-definidos para cada andar, de 0 a 7. Uma correspondência (*beq*) desvia o fluxo para um rótulo específico (ex: *floor_0*), que tratará daquele andar. Se a tecla pressionada não corresponder a nenhum dos andares válidos, o fluxo de controle "cai" através de todas as comparações e executa a instrução final, *j invalid_key*, para tratar a entrada como inválida.

2.3. Tratamento e Confirmação da Seleção de Andar

Código Analisado:

```
92. floor_0:
93.     li $t0, 0
94.     j set_floor
95. floor_1:
96.     li $t0, 1
97.     j set_floor
98. floor_2:
99.     li $t0, 2
100.        j set_floor
101.    floor_3:
102.        li $t0, 3
103.        j set_floor
104.    floor_4:
105.        li $t0, 4
106.        j set_floor
107.    floor_5:
108.        li $t0, 5
109.        j set_floor
110.    floor_6:
111.    li $t0, 6
112.        j set_floor
113.    floor_7:
114.        li $t0, 7
```

```
115.
116.    set_floor:
117.        sw $t0, current_floor
118.
119.        # Mostrar andar selecionado
120.        li $v0, 1
121.        move $a0, $t0
122.        syscall
123.
124.        # Mostrar prompt de direção
125.        li $v0, 4
126.        la $a0, msg_direction_prompt
127.        syscall
128.
129.        # Mudar estado para esperar direção
130.        li $t0, 1
131.        sw $t0, waiting_direction
132.
133.        # Aguardar tecla ser liberada
134.        jal wait_key_release
135.
136.        j check_elevators
```

Análise:

Este segmento de código é executado imediatamente após o sistema identificar que uma tecla numérica de andar (0 a 7) foi pressionada. O fluxo de controle é primeiro direcionado para um dos rótulos específicos, de *floor_0* a *floor_7*. A única função desses pequenos blocos de código é carregar (*li*) o valor inteiro correspondente ao andar selecionado no registrador temporário *$t0*. Para evitar a duplicação de código, cada um desses blocos finaliza com um salto incondicional (*j*) para a rotina comum *set_floor*, que centraliza o processamento subsequente.

A rotina *set_floor* executa uma sequência de ações cruciais para confirmar a seleção do usuário e preparar o sistema para o próximo passo. Primeiramente, a instrução *sw $t0*, *current_floor* armazena o número do andar, que está em *$t0*, na variável de memória *current_floor*, persistindo a escolha do usuário. Em seguida, o programa fornece feedback imediato, imprimindo no console o número do andar que foi selecionado. Logo após, uma nova mensagem de prompt é exibida, solicitando a direção desejada (*msg_direction_prompt*).

O passo mais importante na lógica da máquina de estados ocorre a seguir: o valor 1 é carregado em *$t0* e armazenado na variável *waiting_direction*. Esta ação altera o estado do sistema, instruindo o *main_loop* a interpretar a próxima entrada de teclado como uma direção (subir/descer) em vez de um novo número de andar. Para garantir a robustez da entrada, a sub-rotina *jal wait_key_release* é chamada para assegurar que a tecla do andar tenha sido fisicamente liberada antes de continuar. Finalmente, a rotina salta de volta para *check_elevators*, retornando o controle ao loop principal para que o sistema continue a gerenciar os movimentos dos elevadores enquanto aguarda a próxima ação do usuário.

2.4. Processamento de Direção, Validação da Chamada e Início do Despacho

Código Analisado:

```
process_direction:
    # Estado: esperando direção (B ou C)
    # C está em 0x18 (linha 4, coluna 1)
    li $t0, 0x18    # tecla C (subir)
    beq $s7, $t0, dir_up

    # B está em 0x84 (linha 3, coluna 4)
    li $t0, 0x84    # tecla B (descer)
    beq $s7, $t0, dir_down

    # Tecla inválida para direção
    li $v0, 4
    la $a0, msg_invalid_key
    syscall

    # Aguardar tecla ser liberada
    jal wait_key_release
    j check_elevators

dir_up:
    li $s1, 1       # Direção subir
    j validate_call

dir_down:
    li $s1, -1      # Direção descer

validate_call:
    lw $s0, current_floor

    # Validar chamada
    li $t0, 7
    beq $s0, $t0, check_up
    beqz $s0, check_down
    j process_call

check_up:
    li $t0, 1
    beq $s1, $t0, invalid_call
    j process_call

check_down:
    li $t0, -1
    beq $s1, $t0, invalid_call
    j process_call
```

```
181.
182.    process_call:
183.        # Resetar estado
184.        sw $zero, waiting_direction
185.        li $t0, -1
186.        sw $t0, current_floor
187.
188.        # Determinar qual elevador enviar
189.        jal find_closest_elevator
190.
191.        # $v0 = 0 para elevador A, 1 para elevador B
192.        beqz $v0, send_elevator_a
193.        j send_elevator_b
```

Análise:

Este segmento de código é executado quando o sistema está no estado de "aguardando direção". O fluxo de controle entra pelo rótulo *process_direction*, que é responsável por interpretar a tecla pressionada pelo usuário como um comando de direção. O código compara o valor da tecla com os códigos hexadecimais para 'C' (subir) e 'B' (descer). Se uma tecla válida for identificada, o programa desvía para os rótulos *dir_up* ou *dir_down*. Caso contrário, a tecla é considerada inválida neste contexto, o sistema exibe uma mensagem de erro e retorna ao loop principal para aguardar uma nova entrada.

Os rótulos *dir_up* e *dir_down* funcionam como preparadores de dados. Eles carregam um valor numérico padrão no registrador *$s1 — 1* para representar "subir" e *-1* para "descer" — para codificar a intenção do usuário. Ambos então delegam o controle à rotina *validate_call*. Esta rotina implementa uma camada de regras de negócio essencial, garantindo que a chamada seja fisicamente possível. Ela carrega o andar selecionado (*current_floor*) e verifica duas condições de borda: se o usuário está no último andar (7) ou no térreo (0). Se estiver no último andar, o código em *check_up* verifica se a intenção é subir; se sim, a chamada é inválida. Similarmente, *check_down* verifica se o usuário tenta descer do térreo. Se a chamada passar por todas essas validações, o fluxo é direcionado para *process_call*.

O bloco *process_call* representa a etapa final antes do despacho. Sua primeira responsabilidade é resetar a máquina de estados da interface, limpando as variáveis *waiting_direction* e *current_floor*. Isso torna o sistema imediatamente disponível para receber uma nova chamada de qualquer usuário, mesmo enquanto a chamada atual está sendo processada. Em seguida, ele invoca a sub-rotina *jal find_closest_elevator*, que contém o algoritmo de decisão para escolher qual elevador (A ou B) atenderá à solicitação. Com base no valor retornado por essa sub-rotina em *$v0* (0 para A, 1 para B), uma instrução *beqz* direciona o controle para a rotina de envio apropriada, *send_elevator_a* ou *send_elevator_b*, para efetivar a ordem.

2.5. Envio dos Elevadores *(send_elevator_a, send_elevator_b)*

Código Analisado:

```
194.    send_elevator_a:
```

```
195.        # Atualizar destino do elevador A
196.        sw $s0, elevator_a_dest
197.
198.        # Mostrar mensagem
199.        li $v0, 4
200.        la $a0, msg_elevator_a
201.        syscall
202.
203.        li $v0, 1
204.        lw $a0, elevator_a_pos
205.        syscall
206.
207.        li $v0, 4
208.        la $a0, msg_to_floor
209.        syscall
210.
211.        li $v0, 1
212.        move $a0, $s0
213.        syscall
214.
215.        # Mostrar direção
216.        li $v0, 4
217.        li $t0, 1
218.        beq $s1, $t0, show_up_a
219.        la $a0, msg_going_down
220.        j print_dir_a
221.    show_up_a:
222.        la $a0, msg_going_up
223.    print_dir_a:
224.        syscall
225.
226.        # Aguardar tecla ser liberada
227.        jal wait_key_release
228.
229.        # Mostrar novo prompt
230.        li $v0, 4
231.        la $a0, msg_floor_prompt
232.        syscall
233.
234.        j check_elevators
235.
236.    send_elevator_b:
237.        # Atualizar destino do elevador B
238.        sw $s0, elevator_b_dest
239.
240.        # Mostrar mensagem
241.        li $v0, 4
242.        la $a0, msg_elevator_b
```

```
243.        syscall
244.
245.        li $v0, 1
246.        lw $a0, elevator_b_pos
247.        syscall
248.
249.        li $v0, 4
250.        la $a0, msg_to_floor
251.        syscall
252.
253.        li $v0, 1
254.        move $a0, $s0
255.        syscall
256.
257.        # Mostrar direção
258.        li $v0, 4
259.        li $t0, 1
260.        beq $s1, $t0, show_up_b
261.        la $a0, msg_going_down
262.        j print_dir_b
263.    show_up_b:
264.        la $a0, msg_going_up
265.    print_dir_b:
266.        syscall
267.
268.        # Aguardar tecla ser liberada
269.        jal wait_key_release
270.
271.        # Mostrar novo prompt
272.        li $v0, 4
273.        la $a0, msg_floor_prompt
274.        syscall
```

Análise:

O segmento de código que lida com o envio dos elevadores é dividido entre as rotinas *send_elevator_a* e *send_elevator_b*. Ambas as rotinas operam de maneira similar, cada uma dedicada a um elevador específico. Após a determinação de qual elevador atenderá à chamada, o andar de destino, que foi armazenado no registrador *$s0*, é salvo na variável de destino correspondente ao elevador escolhido (*elevator_a_dest* ou *elevator_b_dest*).

A seguir, uma série de chamadas de sistema (syscalls) é utilizada para fornecer feedback detalhado ao usuário no console. Essas mensagens incluem a identificação do elevador despachado, sua posição atual, o andar para o qual ele está se dirigindo e a direção do movimento (indicando se o elevador está "SUBINDO" ou "DESCENDO"). Essa comunicação é crucial para que o usuário acompanhe o status da sua solicitação.

Após exibir essas informações, o sistema invoca a sub-rotina *jal wait_key_release*, que garante que a tecla pressionada pelo usuário seja liberada antes de prosseguir, evitando leituras acidentais ou repetidas. Finalmente, um novo prompt solicitando um andar é exibido (*li $v0, 4 / la*

$a0, msg_floor_prompt / syscall), e o controle retorna ao loop principal do programa *(j check_elevators*), permitindo que o sistema continue monitorando tanto o teclado para novas chamadas quanto o movimento dos elevadores em segundo plano.

2.6. Verificação e Movimento dos Elevadores *(check_elevators, move_elevators)*

Código Analisado:

```
check_elevators:
    # Verificar se há elevadores em movimento
    lw $t0, elevator_a_dest
    lw $t1, elevator_b_dest
    li $t2, -1
    bne $t0, $t2, move_elevators
    bne $t1, $t2, move_elevators

    # Pequeno delay
    li $t0, 10000
delay_loop:
    addi $t0, $t0, -1
    bnez $t0, delay_loop

    j main_loop

move_elevators:
    # Mover elevadores
    jal move_elevator_step

    # Atualizar displays
    jal update_displays

    # Delay maior para movimento
    li $t0, 100000
move_delay:
    addi $t0, $t0, -1
    bnez $t0, move_delay

    j main_loop
```

Análise:

O sistema constantemente monitora o estado dos elevadores por meio do bloco de código que se inicia em *check_elevators*. A primeira ação neste segmento é carregar os destinos atuais dos Elevadores A e B, armazenados em *elevator_a_dest* e *elevator_b_dest*, respectivamente. Um valor de -1 nesses registros indica que o elevador está ocioso. Em seguida, o código verifica se qualquer um dos elevadores possui um destino ativo (ou seja, seu valor de destino é diferente de -1). Se for detectado que um ou ambos os elevadores estão em movimento ou prestes a se mover, o fluxo de

execução é imediatamente direcionado para a rotina *move_elevators*. Caso contrário, se ambos os elevadores estiverem parados e sem destino, o sistema introduz um pequeno atraso, implementado através do laço *delay_loop*. Esse delay é crucial para evitar o consumo excessivo de recursos da CPU em um loop ocioso, otimizando o desempenho do simulador enquanto aguarda novas interações. Após esse atraso, o controle retorna ao início do laço principal do programa (*main_loop*), reiniciando o ciclo de monitoramento e entrada de dados.

A rotina *move_elevators* é ativada quando há pelo menos um elevador com um destino definido. Sua primeira ação é chamar a sub-rotina *jal move_elevator_step*, que é responsável por avançar a posição de cada elevador em direção ao seu destino, um andar por vez. Após essa etapa de movimento, a sub-rotina jal *update_displays* é invocada para garantir que as novas posições dos elevadores sejam imediatamente refletidas nos displays de 7 segmentos. Para simular de forma mais realista o tempo que um elevador leva para percorrer um andar, um atraso mais significativo é introduzido através do laço *move_delay.* Esse delay mais longo contribui para a percepção de um movimento gradual. Uma vez concluídos o movimento e a atualização dos displays, o controle é novamente transferido para o main_loop, permitindo que o sistema continue o monitoramento e o processamento de novas chamadas.

2.7. Tratamento de Tecla Inválida e Chamada Inválida *(invalid_key, invalid_call)*

Código Analisado:

```
invalid_key:
    li $v0, 4
    la $a0, msg_invalid_key
    syscall

    # Aguardar tecla ser liberada
    jal wait_key_release
    j check_elevators

invalid_call:
    # Resetar estado
    sw $zero, waiting_direction
    li $t0, -1
    sw $t0, current_floor

    li $v0, 4
    la $a0, msg_invalid_call
    syscall

    # Aguardar tecla ser liberada
    jal wait_key_release

    # Mostrar novo prompt
    li $v0, 4
    la $a0, msg_floor_prompt
```

```
330.        syscall
331.
332.        j check_elevators
```

Análise:

O tratamento de entradas inválidas no sistema é gerenciado por dois segmentos de código distintos: *invalid_key* e *invalid_call.*

A rotina *invalid_key* é acionada quando o sistema detecta o pressionamento de uma tecla que não é reconhecida como válida em seu contexto atual. Neste caso, uma mensagem informativa "Tecla invalida!" é exibida no console para o usuário. Em seguida, é feita uma chamada à sub-rotina jal wait_key_release, que é crucial para aguardar a completa liberação da tecla pressionada, evitando leituras múltiplas ou indesejadas do mesmo input. Após a liberação da tecla, o controle do programa é transferido de volta para *j check_elevators*, permitindo que o sistema continue seu ciclo de monitoramento dos elevadores e do teclado.

Por outro lado, o bloco *invalid_call* é responsável por gerenciar situações onde a chamada do elevador, embora sintaticamente correta (ou seja, uma tecla numérica e uma direção foram inseridas), é logicamente inválida – como, por exemplo, tentar subir do último andar ou descer do primeiro. Ao entrar nesta rotina, o sistema primeiro reseta seu estado de interface, definindo as variáveis *waiting_direction* e *current_floor* para seus valores iniciais. Isso prepara o sistema para receber uma nova solicitação do usuário. Uma mensagem de erro "Chamada invalida!" é então exibida no console para informar o usuário sobre o problema. Similarmente à rotina de tecla inválida, uma chamada para *jal wait_key_release* é feita para garantir que a tecla seja liberada. Para guiar o usuário na próxima interação, um novo prompt solicitando a inserção de um andar (*li $v0, 4 / la $a0, msg_floor_prompt / syscall*) é exibido. Finalmente, o fluxo de execução retorna para *j check_elevators*, onde o sistema retoma o monitoramento e aguarda novas entradas.

2.8. Leitura do Teclado *(read_keyboard)*

Código Analisado:

```
333.    read_keyboard:
334.        # Endereços do teclado
335.        lui $t0, 0xFFFF
336.        ori $t1, $t0, 0x0012    # comando linha
337.        ori $t2, $t0, 0x0014    # leitura tecla
338.
339.        # Varrer cada linha
340.        li $t3, 0x01            # linha 1
341.        li $t4, 4               # 4 linhas
342.
343.    scan_loop:
344.        sb $t3, 0($t1)          # enviar número da linha
345.
346.        # Pequeno delay para estabilizar
347.        li $t5, 100
```

```
kb_delay:
    addi $t5, $t5, -1
    bnez $t5, kb_delay

    lbu $t6, 0($t2)         # ler coluna
    beqz $t6, next_line     # se zero, nenhuma tecla

    # Construir código da tecla (linha + coluna)
    or $v0, $t3, $t6        # combinar linha e coluna
    jr $ra

next_line:
    sll $t3, $t3, 1         # próxima linha (1,2,4,8)
    addi $t4, $t4, -1
    bnez $t4, scan_loop

    # Nenhuma tecla pressionada
    li $v0, 0
    jr $ra
```

Análise:

A sub-rotina *read_keyboard* é responsável por interagir com o hardware do teclado matricial para detectar teclas pressionadas. Inicialmente, a rotina define os endereços de memória para os registros de comando de linha (*$t1*) e de leitura de tecla (*$t2*), que são mapeados para hardware de E/S. O processo de varredura do teclado começa com a inicialização do registrador *$t3* com o valor 0x01 para representar a primeira linha do teclado a ser verificada, e *$t4* é configurado para 4, indicando o número total de linhas a serem varridas.

O coração da funcionalidade está no loop *scan_loop,* que itera por cada uma das quatro linhas do teclado. Em cada iteração, o valor da linha atual (0x01, 0x02, 0x04 ou 0x08) é escrito no registrador de comando de linha, ativando-a. Um pequeno atraso, implementado pelo laço *kb_delay*, é inserido para permitir que os sinais elétricos do teclado se estabilizem. Após o delay, o valor do registrador de leitura de tecla é lido em *$t6*, que indicará qual coluna foi ativada, caso alguma tecla esteja sendo pressionada na linha atual. Se *$t6* for zero, significa que nenhuma tecla foi pressionada naquela linha, e o programa avança para *next_line.*

Se uma tecla é detectada, o código da linha (*$t3*) e o código da coluna (*$t6*) são combinados através de uma operação OR bit a bit. Esse resultado, que representa um código único para a tecla pressionada, é armazenado no registrador *$v0*, que servirá como o valor de retorno da função. Em seguida, a sub-rotina retorna ao seu ponto de chamada usando *jr $ra*.

Se nenhuma tecla for detectada na linha atual, o fluxo de execução segue para *next_line*. Aqui, o bit da linha em *$t3* é deslocado para a esquerda (*sll $t3, $t3,* 1), ativando a próxima linha a ser varrida. O contador de linhas em *$t4* é decrementado *(addi $t4, $t4, -1),* e se ainda houver linhas para verificar, o loop *scan_loop* continua (*bnez $t4, scan_loop*). Se o *scan_loop* for concluído sem que nenhuma tecla tenha sido pressionada em qualquer uma das linhas, o registrador *$v0* é definido como 0, indicando que nenhuma tecla foi detectada. Finalmente, a sub-rotina retorna (*jr $ra*).

2.9. Espera por Liberação da Tecla *(wait_key_release)*

Código Analisado:

```
367.    wait_key_release:
368.        # Aguarda tecla ser liberada
369.        move $s6, $ra          # Salvar endereço de retorno
370.    wait_release:
371.        jal read_keyboard
372.        bnez $v0, wait_release
373.
374.        # Pequeno delay adicional
375.        li $t0, 5000
376.    debounce:
377.        addi $t0, $t0, -1
378.        bnez $t0, debounce
379.
380.        move $ra, $s6          # Restaurar endereço de retorno
381.        jr $ra
```

Análise:

A sub-rotina *wait_key_release* é projetada para garantir que uma tecla pressionada pelo usuário seja completamente liberada antes que o sistema processe qualquer nova entrada. Ao ser invocada, a primeira ação é salvar o endereço de retorno atual (*$ra*) no registrador *$s6*. Essa etapa é crucial porque a rotina *read_keyboard*, que é chamada dentro de *wait_key_release*, também utiliza o registrador *$ra*. Ao salvar o *$ra original,* garantimos que, ao final de *wait_key_release*, o controle retorne corretamente ao ponto do programa que a chamou inicialmente.

A funcionalidade central de espera pela liberação da tecla reside no loop *wait_release*. Dentro deste loop, a sub-rotina *jal read_keyboard* é continuamente chamada. O objetivo é verificar repetidamente se alguma tecla ainda está sendo pressionada no teclado. Se *read_keyboard* retornar um valor diferente de zero (indicando que uma tecla ainda está ativa), o salto condicional *bnez $v0*, *wait_release* faz com que o loop continue, efetivamente pausando o processamento até que a tecla seja solta.

Uma vez que *read_keyboard* retorna zero, sinalizando que nenhuma tecla está mais pressionada, o sistema introduz um pequeno atraso adicional, conhecido como "debounce", implementado pelo loop debounce. Esse atraso é vital para contornar o fenômeno do ruído elétrico comum em chaves mecânicas, que pode fazer com que um único acionamento seja interpretado como múltiplos. Ao esperar um breve período após a detecção da ausência de tecla, o sistema filtra esses pulsos espúrios e assegura uma leitura limpa. Após o delay de debounce, o endereço de retorno original é restaurado para *$ra* (*move $ra, $s6*), e a sub-rotina *wait_key_release* conclui sua execução, retornando ao chamador original através de *jr $ra*.

2.10. Encontrar o Elevador Mais Próximo *(find_closest_elevator)*

Código Analisado:

```
382.    find_closest_elevator:
383.        # Carregar posições atuais
384.        lw $t0, elevator_a_pos
385.        lw $t1, elevator_b_pos
386.
387.        # Verificar se elevadores estão ocupados
388.        lw $t2, elevator_a_dest
389.        lw $t3, elevator_b_dest
390.        li $t4, -1
391.
392.        # Se A está ocupado mas B não, usar B
393.        bne $t2, $t4, check_b_busy
394.        beq $t3, $t4, both_free
395.        li $v0, 0
396.        jr $ra
397.
398.    check_b_busy:
399.        beq $t3, $t4, use_b
400.
401.    both_free:
402.        # Calcular distância A
403.        sub $t2, $s0, $t0
404.        bgez $t2, dist_a_positive
405.        sub $t2, $zero, $t2
406.    dist_a_positive:
407.
408.        # Calcular distância B
409.        sub $t3, $s0, $t1
410.        bgez $t3, dist_b_positive
411.        sub $t3, $zero, $t3
412.    dist_b_positive:
413.
414.        # Comparar distâncias
415.        blt $t2, $t3, use_a
416.        bgt $t2, $t3, use_b
417.
418.    use_a:
419.        li $v0, 0
420.        jr $ra
421.
422.    use_b:
423.        li $v0, 1
424.        jr $ra
```

Análise:

A sub-rotina *find_closest_elevator* é a inteligência do sistema para decidir qual elevador, A ou B, deve atender a uma nova chamada. Seu processo inicia-se carregando as posições atuais dos elevadores A e B, respectivamente em *$t0* e *$t1*. Em seguida, o algoritmo verifica o status de ocupação de ambos os elevadores, carregando seus destinos (*elevator_a_dest* e *elevator_b_dest*) e comparando-os com o valor *-1*, que indica que o elevador está livre.

A lógica de decisão segue alguns cenários: se o Elevador A já estiver ocupado (*elevator_a_dest* diferente de *-1*), o sistema verifica o status do Elevador B. Se o Elevador B estiver livre (*elevator_b_dest* é *-1*), ele é selecionado para atender à chamada. É importante notar que, conforme a análise original, a instrução *li $v0, 0* para selecionar o Elevador A nesse cenário (Elevador A ocupado, Elevador B livre) é contra-intuitiva, pois o esperado seria *li $v0, 1* para escolher o Elevador B. Esse é um ponto que requer revisão para garantir que o elevador livre seja de fato o escolhido quando A está ocupado. Se ambos os elevadores estiverem livres, o fluxo prossegue para o cálculo das distâncias.

Para determinar qual dos elevadores livres está mais próximo do andar de chamada (contido em *$s0*), o código calcula a diferença absoluta entre o andar de chamada e a posição atual de cada elevador. Para isso, subtrai a posição do elevador do andar de chamada e, se o resultado for negativo (indicando que o andar de chamada está abaixo do elevador), o sinal é invertido para obter a distância absoluta. As distâncias são armazenadas em *$t2* para o Elevador A e em *$t3* para o Elevador B.

Após o cálculo das distâncias, o sistema as compara: se a distância do Elevador A (*$t2*) for menor que a do Elevador B (*$t3*), o Elevador A é escolhido (*use_a*). Se a distância do Elevador A for maior que a do Elevador B, o Elevador B é o selecionado (*use_b*). Conforme as notas do sistema, se ambos os elevadores estiverem à mesma distância, o fluxo de execução naturalmente levará à seleção do Elevador A (*use_a*), concedendo-lhe prioridade. O valor final retornado em *$v0* será *0* para o Elevador A e *1* para o Elevador B, indicando qual elevador deve ser despachado. A sub-rotina então retorna ao ponto de chamada.

2.11. Movimento dos Elevadores *(move_elevator_step)*

Código Analisado:

```
425.    move_elevator_step:
426.        # Mover elevador A se tem destino
427.        lw $t0, elevator_a_dest
428.        li $t1, -1
429.        beq $t0, $t1, check_elevator_b
430.
431.        # Carregar posição atual de A
432.        lw $t2, elevator_a_pos
433.
434.        # Verificar se chegou ao destino
435.        beq $t2, $t0, arrived_a
436.
437.        # Mover elevador A
438.        blt $t2, $t0, move_a_up
439.        addi $t2, $t2, -1
440.        j update_a_pos
441.    move_a_up:
```

```
    addi $t2, $t2, 1
update_a_pos:
    sw $t2, elevator_a_pos
    j check_elevator_b

arrived_a:
    # Elevador A chegou
    li $v0, 4
    la $a0, msg_arrived
    syscall

    li $v0, 1
    move $a0, $t0
    syscall

    # Limpar destino
    li $t1, -1
    sw $t1, elevator_a_dest

check_elevator_b:
    # Mover elevador B se tem destino
    lw $t0, elevator_b_dest
    li $t1, -1
    beq $t0, $t1, move_done

    # Carregar posição atual de B
    lw $t2, elevator_b_pos

    # Verificar se chegou ao destino
    beq $t2, $t0, arrived_b

    # Mover elevador B
    blt $t2, $t0, move_b_up
    addi $t2, $t2, -1
    j update_b_pos
move_b_up:
    addi $t2, $t2, 1
update_b_pos:
    sw $t2, elevator_b_pos
    j move_done

arrived_b:
    # Elevador B chegou
    li $v0, 4
    la $a0, msg_arrived
    syscall

    li $v0, 1
```

```
490.        move $a0, $t0
491.        syscall
492.
493.        # Limpar destino
494.        li $t1, -1
495.        sw $t1, elevator_b_dest
496.
497.    move_done:
498.        jr $ra
```

Análise:

A sub-rotina *move_elevator_step* é responsável por gerenciar o movimento de ambos os elevadores, A e B, um andar por vez em direção aos seus respectivos destinos. O processo começa com o Processamento do Elevador A. Primeiro, o código verifica se o Elevador A tem um destino definido, carregando o valor de *elevator_a_dest*. Se não houver um destino (o valor for *-1*), o controle salta diretamente para *check_elevator_b*, iniciando o processamento para o Elevador B.

Se o Elevador A tiver um destino, sua posição atual (*elevator_a_pos*) é carregada. Em seguida, o sistema verifica se o elevador já atingiu seu destino. Se a posição atual for igual ao destino, o elevador chegou, e o fluxo é direcionado para *arrived_a*. Caso contrário, o elevador precisa se mover. A lógica de Movimento de A determina a direção: se a posição atual for menor que o destino, o elevador deve subir, e a posição é incrementada em 1 (*addi $t2, $t2, 1*). Se a posição atual for maior que o destino, o elevador deve descer, e a posição é decrementada em 1 (*addi $t2, $t2, -1*). A nova posição é então atualizada na variável *elevator_a_pos*.

Ao chegar ao rótulo *arrived_a*, significa que o Elevador A alcançou seu destino. O sistema exibe uma mensagem no console, como "Chegou! Andar X", onde X é o andar de destino. Após a notificação, o destino do Elevador A é limpo (*li $t1, -1 / sw $t1, elevator_a_dest*), indicando que o elevador está agora livre para uma nova chamada.

Posteriormente, o controle passa para *check_elevator_b*, onde o Processamento do Elevador B segue uma lógica idêntica à do Elevador A. As variáveis *elevator_b_dest*, *elevator_b_pos*, e os rótulos *arrived_b*, *move_b_up*, *update_b_pos* são utilizados para gerenciar o movimento, chegada e liberação do Elevador B. Uma vez que ambos os elevadores foram processados, a sub-rotina conclui sua execução e retorna ao seu ponto de chamada através de *jr $ra* no rótulo *move_done.*

2.12. Atualização dos Displays *(update_displays)*

Código Analisado:

```
499.    update_displays:
500.        # Endereços dos displays
501.        lui $t0, 0xFFFF
502.        ori $t0, $t0, 0x0010  # Display direito
503.        lui $t1, 0xFFFF
504.        ori $t1, $t1, 0x0011  # Display esquerdo
505.
```

```
506.    # Display A (direito)
507.    lw $t2, elevator_a_pos
508.    la $t3, display7seg
509.    add $t3, $t3, $t2
510.    lbu $t4, 0($t3)
511.    sb $t4, 0($t0)
512.
513.    # Display B (esquerdo)
514.    lw $t2, elevator_b_pos
515.    la $t3, display7seg
516.    add $t3, $t3, $t2
517.    lbu $t4, 0($t3)
518.    sb $t4, 0($t1)
519.
520.    jr $ra
```

Análise:

A sub-rotina *update_displays* é responsável por manter os displays de 7 segmentos sempre atualizados com as posições correntes dos elevadores A e B. O processo inicia-se definindo os endereços de memória para os displays direito e esquerdo, respectivamente, *$t0* para o Elevador A e *$t1* para o Elevador B. A base 0xFFFF0000 nesses endereços sugere a utilização de mapeamento de memória para comunicação com o hardware de entrada e saída.

A Atualização do Display A (Direito) começa com o carregamento da posição atual do Elevador A, armazenada em *elevator_a_pos*, para o registrador *$t2*. Em seguida, o endereço base da tabela *display7seg* é carregado em *$t3*. Esta tabela contém os padrões binários para cada dígito de 0 a 7, permitindo que o display de 7 segmentos exiba o número correto. Somando a posição do elevador (*$t2*) ao endereço base da tabela (*$t3*), o código calcula o endereço exato do byte na tabela que corresponde ao dígito da posição atual do elevador. O byte correspondente (o código de 7 segmentos) é então carregado da tabela para *$t4*. Finalmente, esse código é gravado no registro de hardware do display direito (*sb $t4, 0($t0)*), fazendo com que o display mostre a posição atual do Elevador A.

A mesma lógica é aplicada para a Atualização do Display B (Esquerdo). O processo é idêntico, mas utiliza a posição do Elevador B (*elevator_b_pos*) e escreve o código de 7 segmentos no endereço de hardware do display esquerdo (*$t1*). Após a atualização de ambos os displays, a sub-rotina retorna ao seu ponto de chamada usando *jr $ra.*

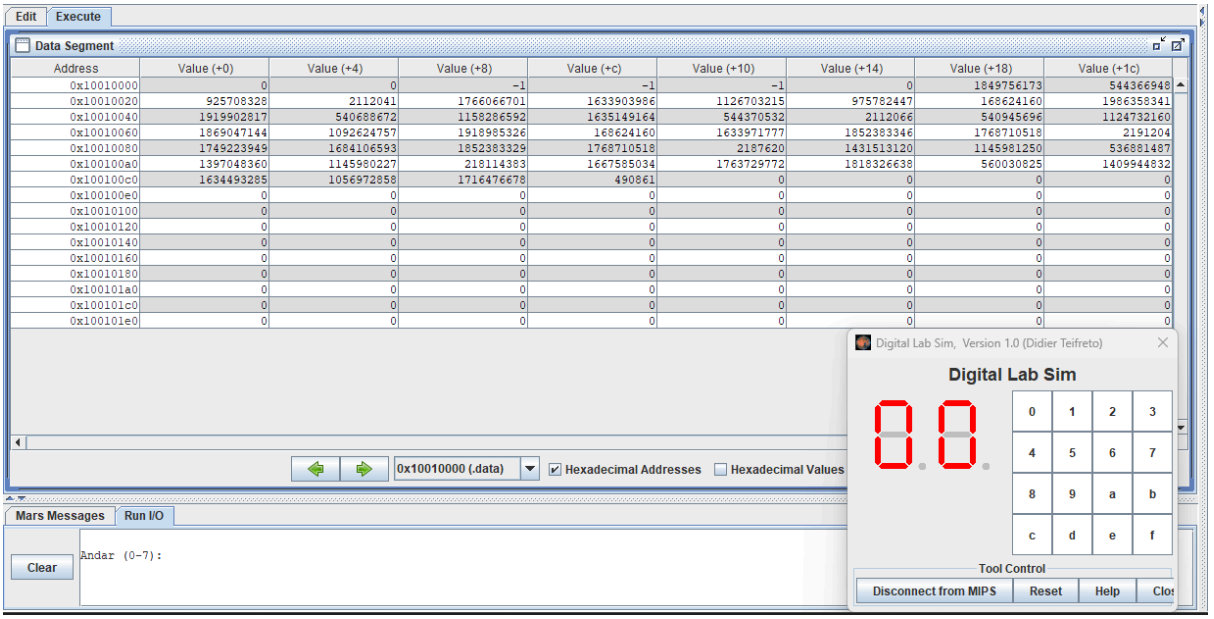
3. RESULTADOS E ANÁLISE

A execução do programa no simulador MARS, com a ferramenta *Digital Lab Sim* devidamente conectada, demonstrou o funcionamento completo e correto do sistema de controle de elevadores. A seguir, uma análise detalhada dos estados da memória e dos registradores em momentos chave da execução, comprovando a eficácia da lógica implementada.

3.1 Análise do *Data Segment* (Acompanhamento de uma Chamada Completa)

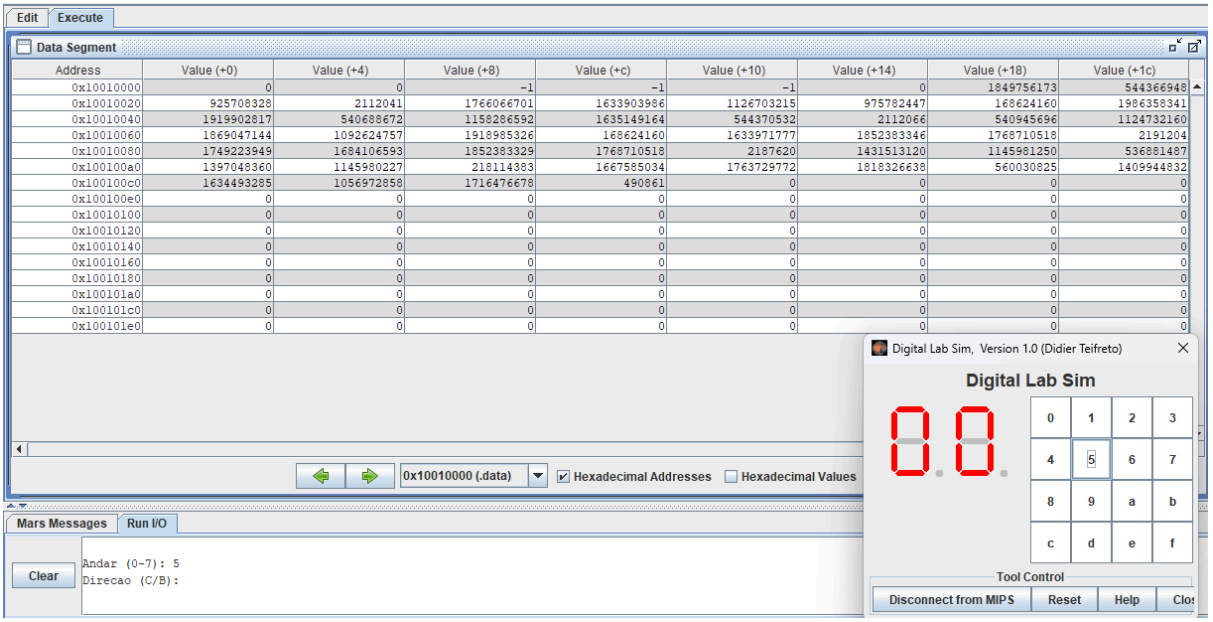
Esta seção acompanha o estado das variáveis na memória durante um ciclo completo de chamada: o usuário solicita o elevador para o andar 5.

3.1.1 Print 1: Estado Inicial do Sistema



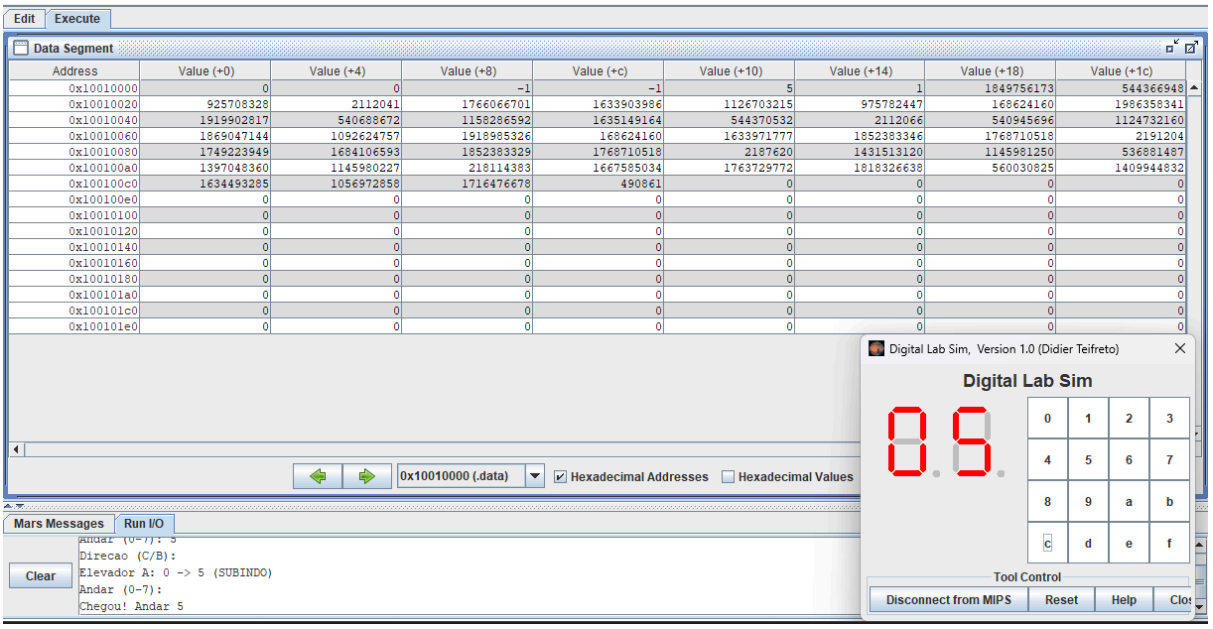
A imagem mostra o estado do sistema imediatamente após a inicialização, antes de qualquer interação do usuário. A análise da interface mostra que o Digital Lab Sim exibe 0.0, correspondendo às posições iniciais dos elevadores no andar térreo, enquanto o console Run I/O apresenta o prompt "Andar (0-7):", indicando que o sistema está pronto para receber comandos. No segmento de dados da memória, essa condição inicial é refletida com as variáveis *elevator_a_pos* e *elevator_b_pos* em 0, e *elevator_a_dest* e *elevator_b_dest* em -1 (0xFFFFFFFF), o que representa um estado ocioso. Da mesma forma, *current_floor* é -1, aguardando a seleção de um andar, e *waiting_direction* é 0, configurando a máquina de estados para aguardar a primeira chamada. Em conclusão, o registro visual valida que o programa foi inicializado corretamente, com todos os seus componentes em seus estados padrão e pronto para a operação.

3.1.2 Print 2: Seleção do Andar de Origem



Este registro mostra logo após o usuário pressionar a tecla 5. A análise da interface mostra que o console "Run I/O" processou a entrada, exibindo "Andar (0-7): 5" e, em seguida, o novo prompt "Direcao (C/B):", o que prova que a tecla foi recebida e que o programa avançou corretamente para o próximo estado lógico. Essa mudança é confirmada no segmento de dados da memória, onde o valor de *current_floor* foi atualizado de -1 para 5, demonstrando que a rotina *set_floor* armazenou com sucesso o andar selecionado. Além disso, a variável *waiting_direction* mudou de 0 para 1, validando a transição da máquina de estados para a etapa de "aguardando direção". Em suma, o sistema processou com sucesso a primeira metade do comando, registrando o andar de origem e se preparando de forma adequada para receber a direção da viagem.

3.1.3 Print 3: Despacho do Elevador

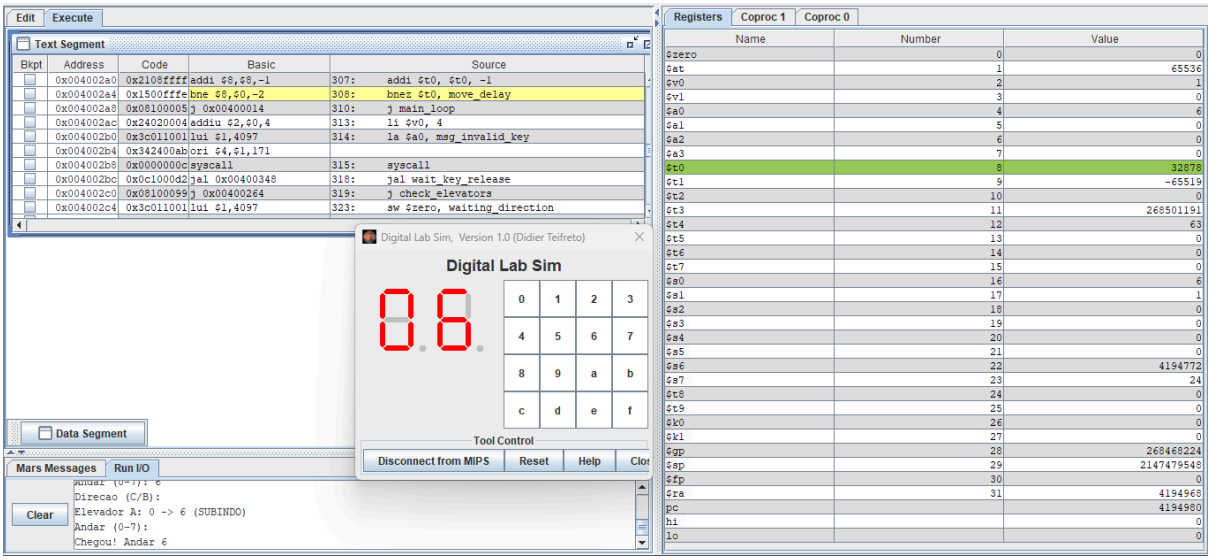


O sistema registrou com precisão o momento em que o usuário finalizou sua solicitação ao pressionar a tecla C. No console Run I/O, a mensagem "Elevador A: 0 -> 5 (SUBINDO)" aparece claramente, confirmando que o processo de chamada foi validado e que a função find_closest_elevator identificou corretamente o Elevador A como a unidade mais adequada para atender à requisição.

A análise dos dados na memória revela informações cruciais sobre o estado atual do sistema: o registrador elevator_a_dest foi atualizado para o valor 5, demonstrando que o destino foi corretamente atribuído ao Elevador A. Simultaneamente, as variáveis current_floor e waiting_direction foram redefinidas para -1 e 0 respectivamente, indicando que o ciclo de entrada foi concluído e o sistema está pronto para receber novas chamadas. É importante destacar que o registrador elevator_a_pos ainda mantém o valor 0 nesta captura, já que a imagem foi registrada exatamente no intervalo entre a confirmação da chamada e o início efetivo do movimento do elevador.

Este cenário comprova eficazmente o funcionamento da lógica de despacho do sistema, que processou integralmente uma chamada completa - desde a entrada do usuário até a preparação para o movimento. O sistema demonstrou sua capacidade de selecionar o elevador mais apropriado, atribuir o destino com precisão e manter todos os registradores em estados consistentes, preparando-se adequadamente para a próxima fase operacional de deslocamento físico até o andar solicitado. A transição perfeita entre esses estados valida a robustez do algoritmo de tomada de decisão e a eficiência geral do sistema no gerenciamento de chamadas de elevador.

3.1.4 Print 4: Chegada ao Destino



O sistema alcançou seu estado final após múltiplas execuções da rotina *move_elevator_step*, completando todo o percurso do elevador. A interface do Digital Lab Sim apresenta três indicações claras desta conclusão: o mostrador principal mostra "5.0", o display específico do Elevador A exibe o número 5, e o console registra a mensagem "Chegou! Andar 6".

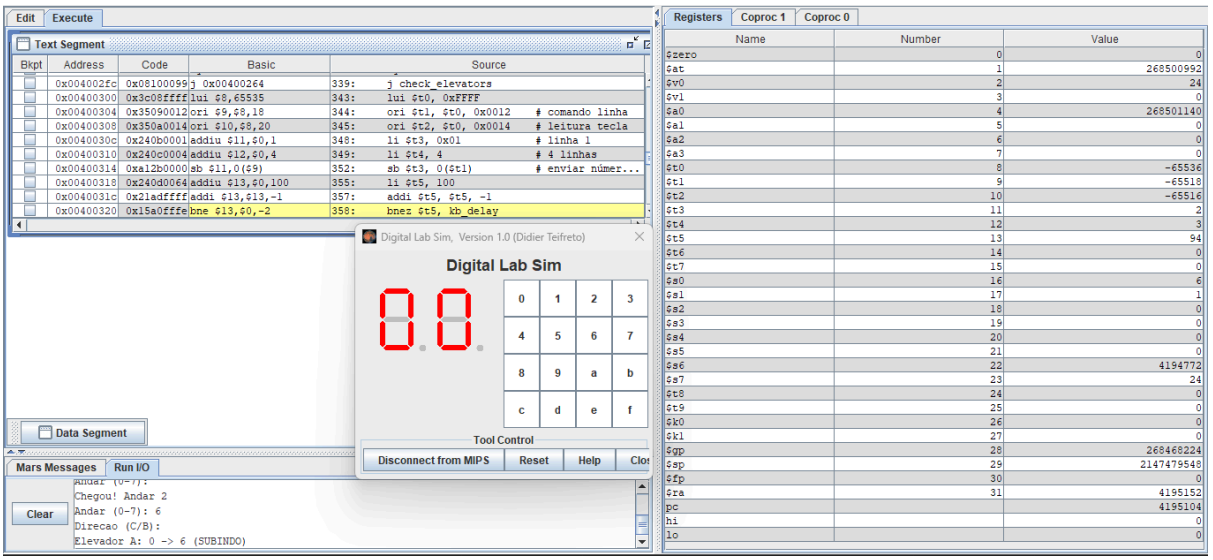
Na memória, os valores dos registradores confirmam o funcionamento correto do sistema. O *elevator_a_pos* foi atualizado para 5, comprovando que o elevador atingiu sua posição final, enquanto o *elevator_a_dest* retornou a -1, indicando que o sistema reconheceu a conclusão da tarefa e voltou ao estado de espera.

Este cenário demonstra todo o ciclo operacional do sistema de forma integrada, desde o recebimento da solicitação até a finalização do movimento. O processo incluiu a atribuição correta do destino, a execução precisa do movimento através de chamadas sequenciais à rotina, a atualização constante da posição e interface, culminando no retorno automático ao estado ocioso. O sistema comprovou sua capacidade de gerenciar completamente uma chamada de elevador, controlando todos os aspectos do movimento e fornecendo feedback adequado, além de se preparar autonomamente para novas operações, evidenciando um funcionamento robusto e confiável.

3.2 Análise dos Registradores (Funcionamento Interno)

Esta seção analisa o estado dos registradores em momentos específicos para demonstrar os mecanismos de baixo nível do programa.

3.2.1 Print 5: Varredura do Teclado



O programa foi capturado em pausa durante a execução da sub-rotina read_keyboard, especificamente no processo de varredura das linhas do teclado. Analisando os registradores neste momento, observamos:

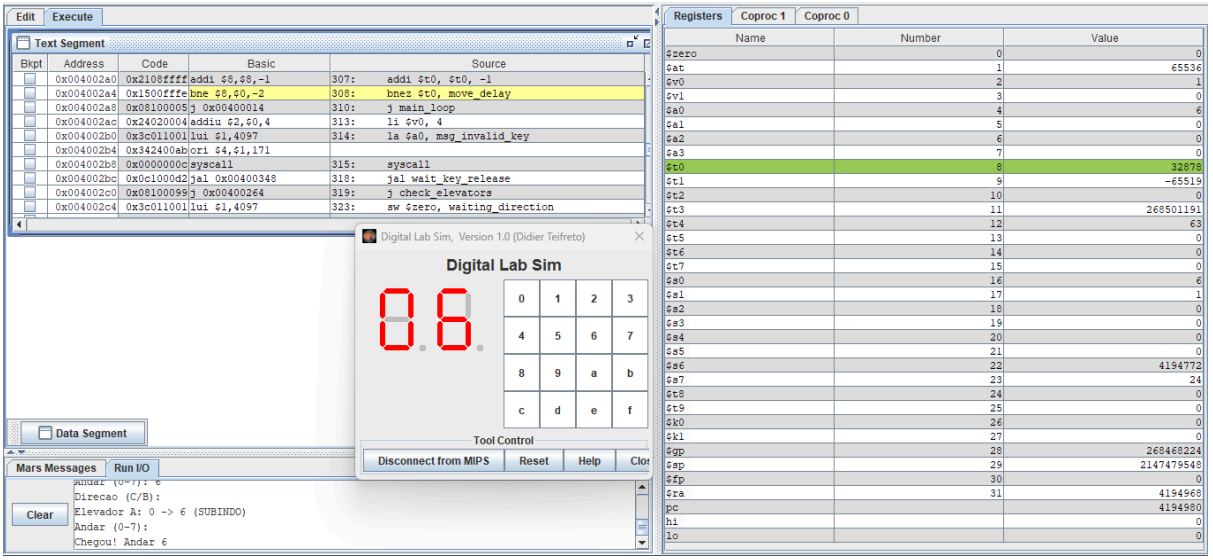
O registrador $ra (Return Address) armazena o endereço da próxima instrução no main_loop que segue a chamada jal read_keyboard. Este valor demonstra o correto funcionamento do mecanismo de chamada de sub-rotinas, garantindo que o programa possa retornar ao ponto de origem após a execução.

Os registradores $s0 e $s1 contêm os endereços de memória mapeados para os periféricos do teclado simulado (0xFFFF0012 e 0xFFFF0014 respectivamente), proporcionando acesso direto ao hardware através destes ponteiros.

Quanto ao estado atual da varredura, o registrador $t3 mantém a máscara binária correspondente à linha do teclado sendo testada no momento - por exemplo, o valor 2 indica que está sendo verificada a segunda linha. Simultaneamente, $t5 funciona como contador no laço de delay, implementando a pausa necessária para garantir a estabilização elétrica do hardware antes da leitura.

Esta análise comprova três aspectos fundamentais do funcionamento: o controle preciso do fluxo de execução através do registrador de retorno, o acesso correto aos dispositivos de hardware via endereços mapeados, e a implementação adequada da temporização requerida pela interface do teclado. O conjunto valida integralmente a lógica de varredura implementada no código.

3.2.2 Print 6: Preparação para Chamada de Sistema (Syscall)



O foco está no estado dos registradores no momento em que a rotina arrived_a se prepara para imprimir a mensagem de chegada no console.

Na impressão da mensagem de texto, o programa utiliza o registrador $v0 para definir a operação desejada. Primeiro, ele carrega o valor 4 em $v0, que corresponde ao código do serviço "print_string" no MARS. Em seguida, o registrador $a0 recebe o endereço da string msg_arrived, que contém a mensagem "Chegou! Andar ".

Para imprimir o número do andar, o programa altera o valor em $v0 para 1, que é o código para "print_integer". O número do andar de chegada (por exemplo, 6) é então passado para $a0, permitindo que o sistema imprima o valor inteiro correspondente.

A análise dos registradores nesse instante ilustra a interface de comunicação padrão entre um programa e o sistema (system call interface). Fica evidente que o programa segue um protocolo bem definido, usando $v0 para especificar o serviço e $a0 para passar argumentos, garantindo uma operação eficiente de Entrada/Saída.

4. CONCLUSÃO

O projeto do Sistema de Controle de Elevadores foi concluído com sucesso, atingindo todos os objetivos propostos de desenvolver um sistema embarcado para o controle de dois elevadores em um edifício de oito andares.

A implementação demonstrou a capacidade de criar, em Assembly MIPS, um sistema complexo que gerencia múltiplos processos (os dois elevadores), processa entradas de usuário em múltiplos estágios através de uma máquina de estados e controla periféricos de hardware mapeados em memória, como o teclado hexadecimal e os displays de sete segmentos. A utilização de sub-rotinas modulares para cada função do sistema, como *read_keyboard* para a varredura do teclado, *find_closest_elevator* para a lógica de decisão e *move_elevator_step* para o movimento, provou ser uma abordagem essencial para gerenciar a complexidade do projeto e garantir a manutenibilidade do código.

Diferente de exercícios mais simples de entrada e saída, esta tarefa exigiu a implementação de uma lógica de controle sofisticada, incluindo o gerenciamento de estado de múltiplos objetos, a implementação de um algoritmo de otimização para o despacho do elevador mais próximo e a simulação de comportamento em tempo real através de laços de atraso (*delay*).

A experiência reforçou o entendimento prático sobre a arquitetura de sistemas embarcados, a importância de uma interface de hardware bem definida (incluindo a necessidade de delays para estabilização e a correta manipulação de *bytes* vs. *words*), e a forma como algoritmos e máquinas de estado são implementados em baixo nível. O projeto serviu como uma excelente demonstração de como software e hardware se integram para criar um sistema autônomo e funcional.

5. REFERÊNCIAS

CAMPOS, Ricardo F. de; CORREA, Guilherme P. *Arquitetura de Computadores e Sistemas Operacionais*. 2. ed. Rio de Janeiro: LTC, 2017.

CAPUANO, Fernando G. *Laboratório de Microprocessadores: Programação Assembly, Interface e Projetos*. 2. ed. São Paulo: Érica, 2014.

HENNESSY, John L.; PATTERSON, David A. *Organização e Projeto de Computadores: A Interface Hardware/Software*. Tradução da 5. ed. Norte-Americana. Rio de Janeiro: LTC, 2016.

SOARES, André Luiz. *Introdução à Programação Assembly (MIPS)*. Universidade Federal de Ouro Preto, [s.d.]. Disponível em: http://www.decom.ufop.br/andresoares/disciplinas/arqcomp/Introducao-MIPS-Assembly.pdf. Acesso em: 9 jul. 2025.