

Universidade Federal de Pernambuco

Emmanuel Silva Nascimento  
Guilherme Vinicius Nigro Braga  
Ruan Alexandre Ferreira Da Silva  
João Gabriel Valentim Dias  
Erick Martins Gomes Vieira

Relatório da APS4  
Nível do Sistema de Computador

Recife  
2024

NandGame

# Memória

A memória é a parte onde guardamos as informações que serão utilizadas pela CPU no funcionamento do computador, ela é separada em 3 partes:

ROM → É a memória onde guardamos as instruções que serão enviadas à CPU.

RAM → É a memória temporária do computador, normalmente utilizada para guardar dados que serão usados pela CPU.

IO → É a memória que guarda as entradas e saídas do do computador, para que possam ser lidas pelo computador (no caso da entrada), e para ser lida por outros componentes fora do computador (no caso da saída).

Após utilizar os módulos fornecidos pelo arquiteto, fomos capazes de juntar todas em memory.v e criar o testbench para a compilação e a simulação em ondas.

Fotos dos testes no testbench:

```
// Reset
reset = 0;
#10;
reset = 1;
#10;

// Ler ROM
address = 8'd0;
#10;
address = 8'd1;
#10;

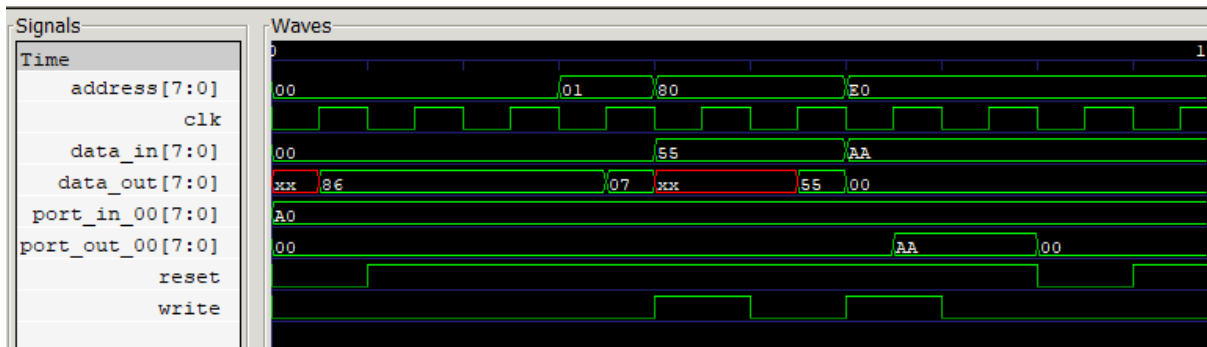
// Escrever e ler na RAM
write = 1;
address = 8'd128;
data_in = 8'h55;
#10;
write = 0;
address = 8'd128;
#10;

// Escrever e ler porta de saída
write = 1;
address = 8'hE0;
data_in = 8'hAA;
#10;
write = 0;
address = 8'hE0;
#10;

// Reset
reset = 0;
#10;
reset = 1;
#10;

$finish;
```

Fotos das simulações em onda pelo gtkwave:



Agora explicando cada teste:

Primeiro temos um reset para reiniciar a máquina. 10ns para os tempos de espera da máquina, o reset está em negedge então ele ficará como “1” o teste inteiro até ser atualizado para 0 pela borda negativa.

Segundo temos a leitura da ROM onde lemos o endereço 00 na ROM nos dando o valor 86, que significa load\_A imediato, e após 10ns lemos o endereço 01 da ROM, que armazena o valor 07, após isso temos mais 10ns.

Terceiro temos a escrita da RAM, o sinal write sobe para 1, o que significa que devemos armazenar o valor 55 no endereço 80, após isso temos 10ns de espera para poder ler o endereço 80 na RAM, que mostra exatamente o valor 55.

Após isso temos a escrita e leitura na porta de saída, write sobre para 1 e é nos dado o endereço da porta 0 (E0) e o valor a armazenar (AA), após isso temos mais 10ns de espera, write cai para 0 e nos é dado o endereço E0 novamente, só que dessa vez para ler a porta 0, que mostra o valor AA.

Por último temos mais um reset, ele começa com 0, e quando sobe para 1, todas as saídas caem para 0.

## Data Path

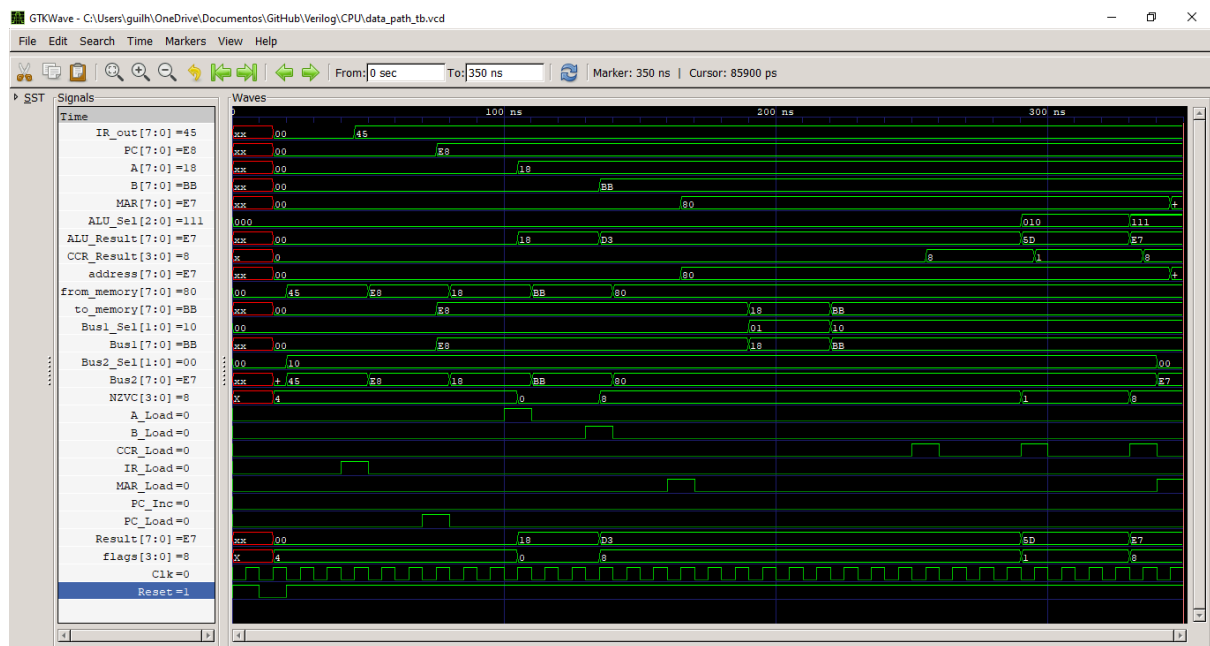
O data path é um conjunto de componentes que executam as operações de dados dentro da CPU. Nele estão presentes os elementos responsáveis pelo transporte e processamento dos dados, registradores, ALU, multiplexadores, etc.

Após concluir o módulo `data_path.v` fornecido pelo engenheiro, criamos um testbench para testar casos essenciais que o data path deve cobrir. Carregamento dos registradores A, B, PC e IR, enviar dados para a memória via barramento de dados, operações da ALU e endereçamento de dados via MAR.

Testes do testbench:

```
57 // Load IR, PC, A, B
58 from_memory = 8'h45;
59 Bus2_Sel = 2'b10;
60 #20 IR_Load = 1;
61 #10 IR_Load = 0;
62 from_memory = 8'hE8;
63 #20 PC_Load = 1;
64 #10 PC_Load = 0;
65 from_memory = 8'h18;
66 #20 A_Load = 1;
67 #10 A_Load = 0;
68 from_memory = 8'hBB;
69 #20 B_Load = 1;
70 #10 B_Load = 0;
71
72 // PC, A or B to memory
73 #20 Bus1_Sel = 01;
74 #30 Bus1_Sel = 10;
75
76 // ALU operations
77 #30 ALU_Sel = 3'b000; // A + B
78 CCR_Load = 1;
79 #10 CCR_Load = 0;
80 #30 ALU_Sel = 3'b010; // A - B
81 CCR_Load = 1;
82 #10 CCR_Load = 0;
83 #30 ALU_Sel = 3'b111; // ~A
84 CCR_Load = 1;
85 #10 CCR_Load = 0;
86
87 // Send Alu Result to address
88 Bus2_Sel = 2'b00;
89 MAR_Load = 1;
90 #10 MAR_Load = 0;
```

Simulando os dados obtidos no GTKWave obtivemos o seguinte resultado:



Com isso concluímos que ao receber os dados corretos da control unit, o data path consegue realizar suas operações corretamente.

# Control Unit

A Control Unit é responsável por dirigir as operações da CPU, gerando os sinais de controle necessários para coordenar e controlar o data path durante a execução das instruções. Ela interpreta as instruções decodificadas e gera sinais que determinam quais operações devem ser realizadas, quais dados devem ser utilizados e para onde os resultados devem ser direcionados.

Antes de concluirmos o módulo `control_unit.v` fornecido para o engenheiro, definimos as instruções que poderiam ser usadas pela nossa CPU no arquivo `instructions.v`, dessa forma as definições poderiam ser chamadas em outros arquivos posteriormente.

Instruções do `instructions.v`:

```
1 // Loads and Stores
2 `define LDA_IMM 8'h86 // Load Register A (Immediate Addressing)
3 `define LDA_DIR 8'h87 // Load Register A from memory (RAM or IO) (Direct Addressing)
4 `define LDB_IMM 8'h88 // Load Register B (Immediate Addressing)
5 `define LDB_DIR 8'h89 // Load Register B from memory (RAM or IO) (Direct Addressing)
6 `define STA_DIR 8'h96 // Store Register A to memory (RAM or IO)
7 `define STB_DIR 8'h97 // Store Register B to memory (RAM or IO)
8 `define STR_DIR 8'h98 // Store Result of Operation to memory (RAM or IO)
9
10 // Data Manipulations
11 `define ADD_AB 8'h42 // A <= A + B
12 `define SUB_AB 8'h43 // A <= A - B
13 `define AND_AB 8'h44 // A <= A & B
14 `define OR_AB 8'h45 // A <= A | B
15 `define INCA 8'h46 // A <= A + 1
16 `define DECA 8'h48 // A <= A - 1
17 `define XOR_AB 8'h4A // A <= A ^ B
18 `define NOTA 8'h4B // A <= ~A
19 `define INCB 8'h4C // B <= B + 1
20 `define DECB 8'h4D // B <= B - 1
21 `define NOTB 8'h4E // B <= ~B
22 `define SUB_BA 8'h4F // B <= B - A
23
24 // Branches
25 `define BRA 8'h20 // Branch Always to (ROM) Address
26 `define BMI 8'h21 // Branch if N == 1 to (ROM) Address
27 `define BPL 8'h22 // Branch if N == 0 to (ROM) Address
28 `define BEQ 8'h23 // Branch if Z == 1 to (ROM) Address
29 `define BNE 8'h24 // Branch if Z == 0 to (ROM) Address
30 `define BVS 8'h25 // Branch if V == 1 to (ROM) Address
31 `define BVC 8'h26 // Branch if V == 0 to (ROM) Address
32 `define BCS 8'h27 // Branch if C == 1 to (ROM) Address
33 `define BCC 8'h28 // Branch if C == 0 to (ROM) Address
34
```

Após isso, concluindo a `control_unit.v`, ela era capaz de realizar qualquer uma das instruções definidas.

Para testar a control unit, criamos um testbench que contempla todas as instruções possíveis. Não poderei mostrar todas as formas de onda aqui, porém elas estarão presentes junto aos arquivos do código fornecidos posteriormente.




Instruções do testbench:



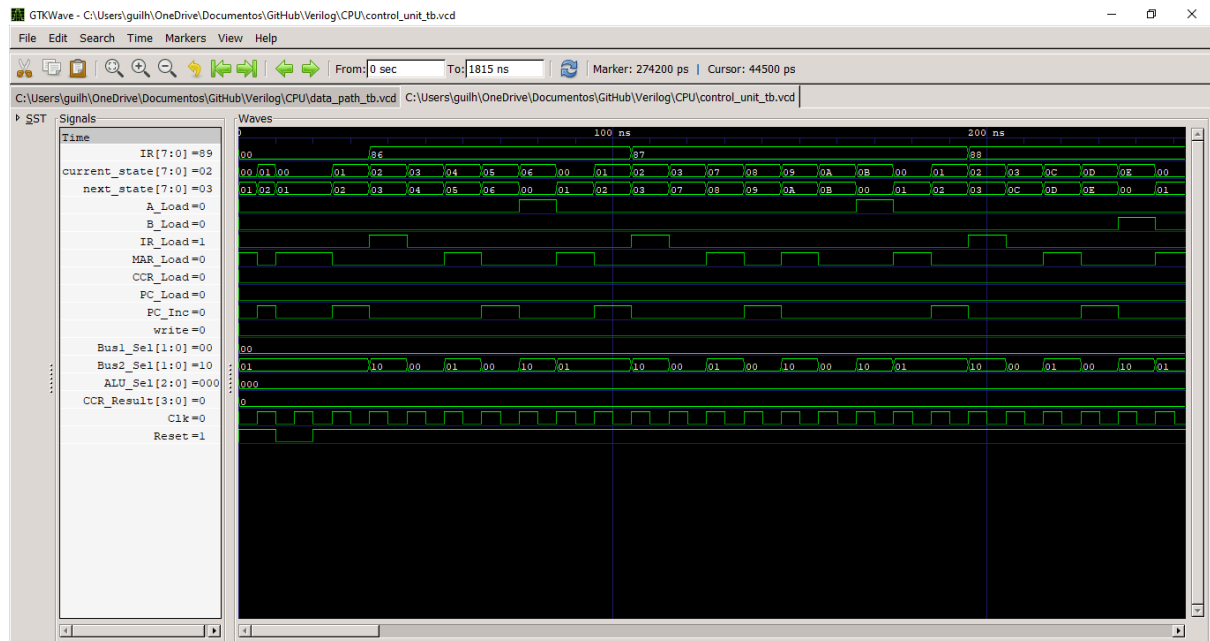
```
46 instruction_sequence[0] = 8'h86; // LDA_IMM
47 instruction_sequence[1] = 8'h87; // LDA_DIR
48 instruction_sequence[2] = 8'h88; // LDB_IMM
49 instruction_sequence[3] = 8'h89; // LDB_DIR
50 instruction_sequence[4] = 8'h96; // STA_DIR
51 instruction_sequence[5] = 8'h97; // STB_DIR
52 instruction_sequence[6] = 8'h98; // STR_DIR
53 instruction_sequence[7] = 8'h42; // ADD_AB
54 instruction_sequence[8] = 8'h43; // SUB_AB
55 instruction_sequence[9] = 8'h44; // AND_AB
56 instruction_sequence[10] = 8'h45; // OR_AB
57 instruction_sequence[11] = 8'h46; // INCA
58 instruction_sequence[12] = 8'h48; // DECA
59 instruction_sequence[13] = 8'h4A; // XOR_AB
60 instruction_sequence[14] = 8'h4B; // NOTA
61 instruction_sequence[15] = 8'h4C; // INCB
62 instruction_sequence[16] = 8'h4D; // DECB
63 instruction_sequence[17] = 8'h4E; // NOTB
64 instruction_sequence[18] = 8'h4F; // SUB_BA
65 instruction_sequence[19] = 8'h20; // BRA
66 instruction_sequence[20] = 8'h21; // BMI True
67 instruction_sequence[21] = 8'h21; // BMI False
68 instruction_sequence[22] = 8'h22; // BPL True
69 instruction_sequence[23] = 8'h22; // BPL False
70 instruction_sequence[24] = 8'h23; // BEQ True
71 instruction_sequence[25] = 8'h23; // BEQ False
72 instruction_sequence[26] = 8'h24; // BNE True
73 instruction_sequence[27] = 8'h24; // BNE False
74 instruction_sequence[28] = 8'h25; // BVS True
75 instruction_sequence[29] = 8'h25; // BVS False
76 instruction_sequence[30] = 8'h26; // BVC True
77 instruction_sequence[31] = 8'h26; // BVC False
78 instruction_sequence[32] = 8'h27; // BCS True
79 instruction_sequence[33] = 8'h27; // BCS False
80 instruction_sequence[34] = 8'h28; // BCC True
81 instruction_sequence[35] = 8'h28; // BCC False
```

Lógica para leitura das instruções:



```
97  for (i = 0; i < 30; i = i + 1) begin
98      @(posedge IR_Load);
99      IR = instruction_sequence[i];
100     case (i)
101         20: CCR_Result = 4'b1000; // BMI True
102         21: CCR_Result = 4'b0000; // BMI False
103         22: CCR_Result = 4'b0000; // BPL True
104         23: CCR_Result = 4'b1000; // BPL False
105         24: CCR_Result = 4'b0100; // BEQ True
106         25: CCR_Result = 4'b0000; // BEQ False
107         26: CCR_Result = 4'b0000; // BNE True
108         27: CCR_Result = 4'b0100; // BNE False
109         28: CCR_Result = 4'b0010; // BVS True
110         29: CCR_Result = 4'b0000; // BVS False
111         30: CCR_Result = 4'b0000; // BVC True
112         31: CCR_Result = 4'b0010; // BVC False
113         32: CCR_Result = 4'b0001; // BCS True
114         33: CCR_Result = 4'b0000; // BCS False
115         34: CCR_Result = 4'b0000; // BCC True
116         35: CCR_Result = 4'b0001; // BCC False
117     endcase
118 end
```

## Formas de onda das três primeiras instruções:



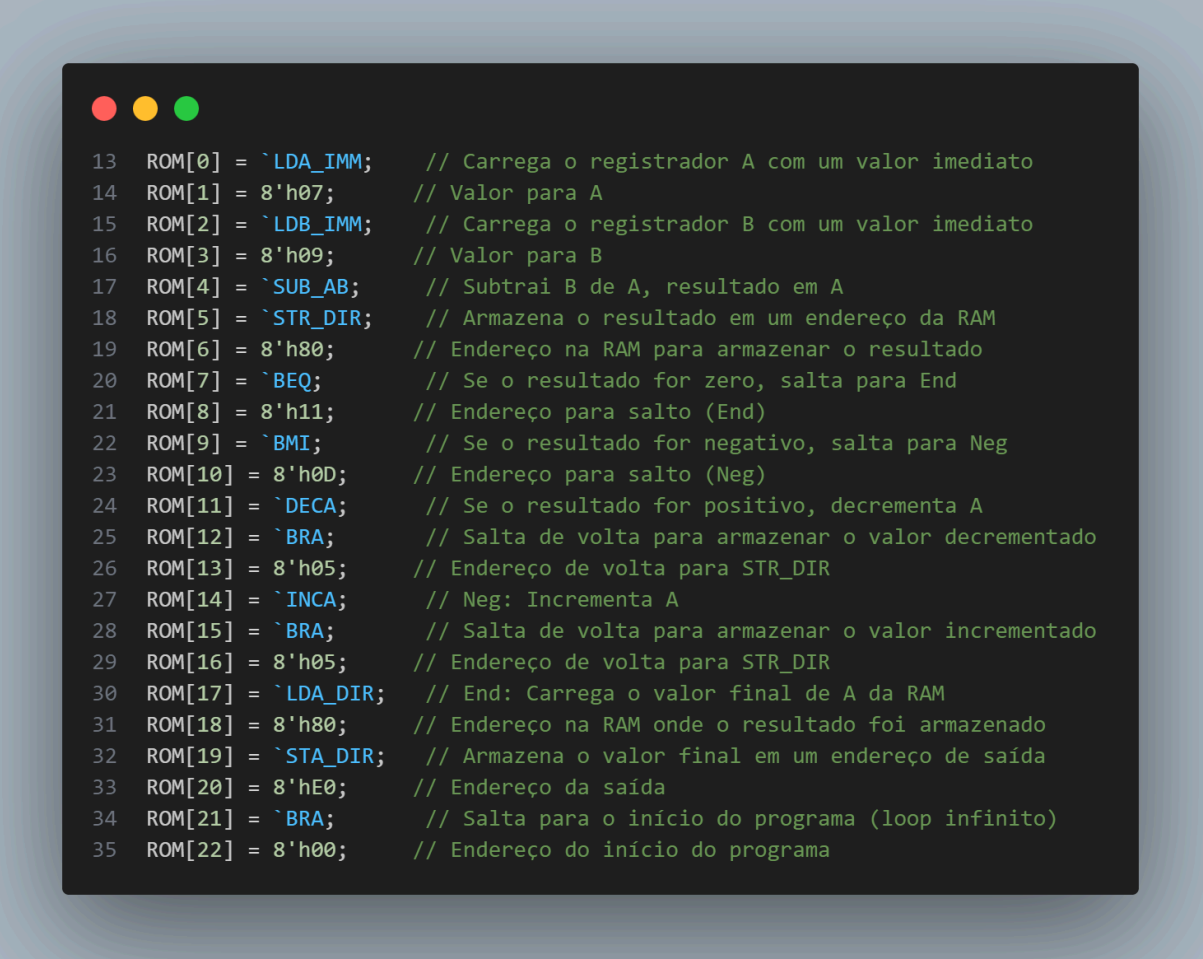
Analisando as formas de onda percebemos que cada instrução está sendo executada perfeitamente.

# Computer

Finalizando todos os módulos componentes, finalmente estávamos aptos para juntar cada módulo para formar o computador.

Para testar nosso computador, desenvolvemos um programa nulificador de contas. Esse programa usa como input um conta matemática qualquer  $A - B$  e é capaz de transformar o resultado em 0 executando um algoritmo que vai ser melhor entendido ao ver o programa. O programa foi armazenado na rom para leitura pela CPU.

Código do programa nulificador de contas:

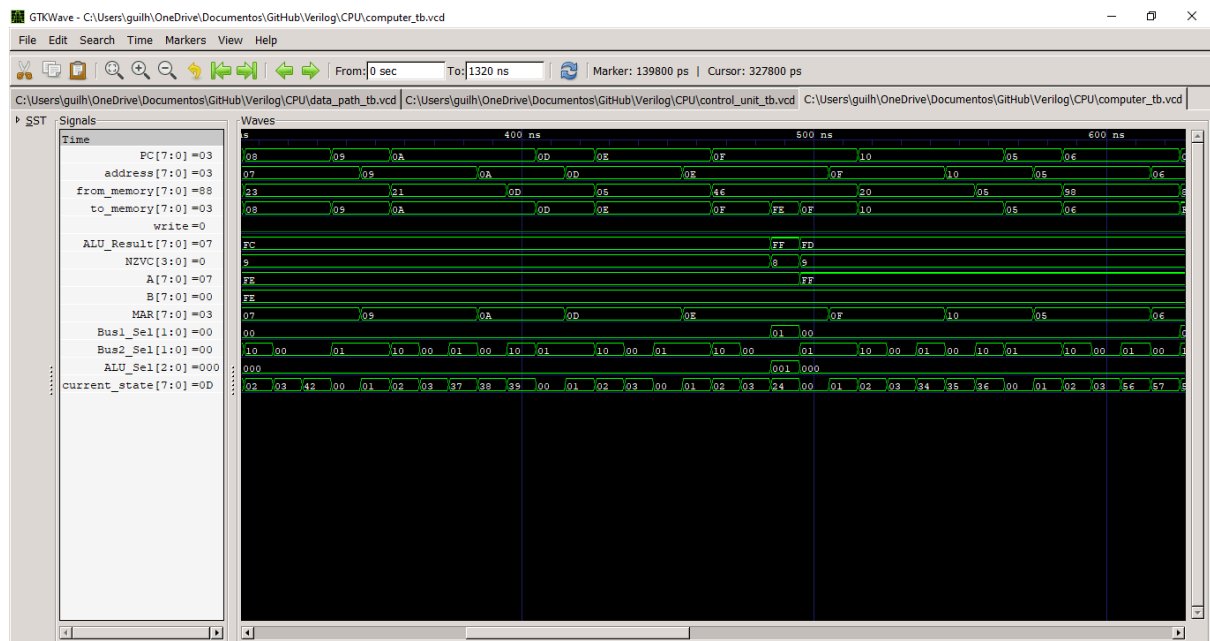


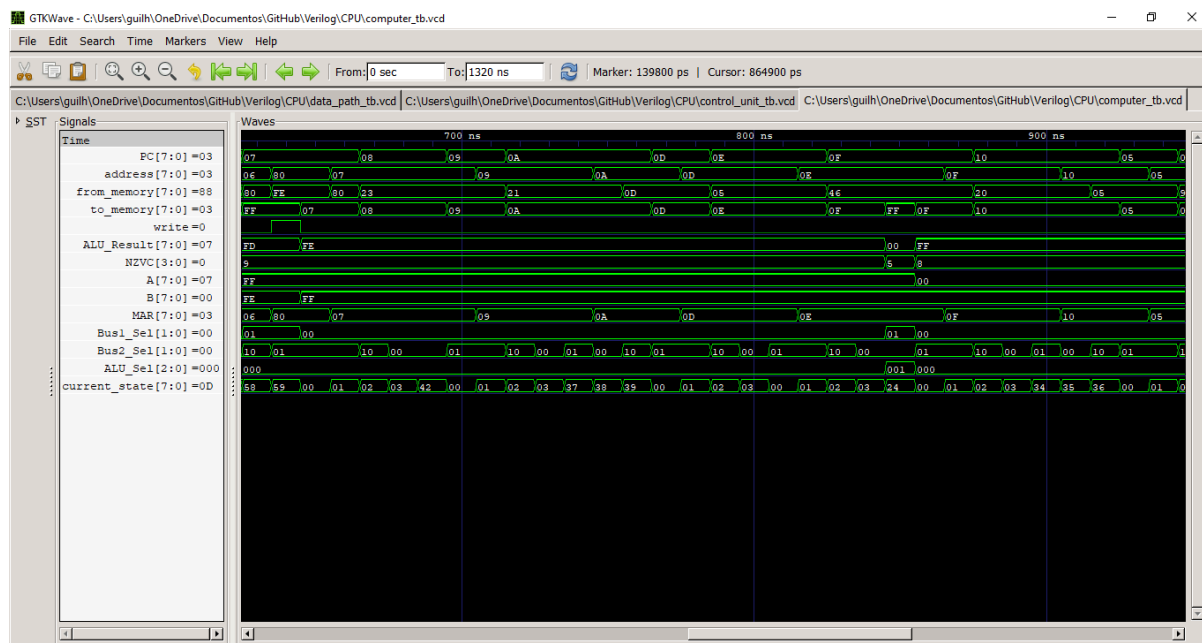
```
13 ROM[0] = `LDA_IMM;    // Carrega o registrador A com um valor imediato
14 ROM[1] = 8'h07;       // Valor para A
15 ROM[2] = `LDB_IMM;    // Carrega o registrador B com um valor imediato
16 ROM[3] = 8'h09;       // Valor para B
17 ROM[4] = `SUB_AB;     // Subtrai B de A, resultado em A
18 ROM[5] = `STR_DIR;    // Armazena o resultado em um endereço da RAM
19 ROM[6] = 8'h80;       // Endereço na RAM para armazenar o resultado
20 ROM[7] = `BEQ;        // Se o resultado for zero, salta para End
21 ROM[8] = 8'h11;       // Endereço para salto (End)
22 ROM[9] = `BMI;        // Se o resultado for negativo, salta para Neg
23 ROM[10] = 8'h0D;      // Endereço para salto (Neg)
24 ROM[11] = `DECA;      // Se o resultado for positivo, decrementa A
25 ROM[12] = `BRA;       // Salta de volta para armazenar o valor decrementado
26 ROM[13] = 8'h05;      // Endereço de volta para STR_DIR
27 ROM[14] = `INCA;      // Neg: Incrementa A
28 ROM[15] = `BRA;       // Salta de volta para armazenar o valor incrementado
29 ROM[16] = 8'h05;      // Endereço de volta para STR_DIR
30 ROM[17] = `LDA_DIR;   // End: Carrega o valor final de A da RAM
31 ROM[18] = 8'h80;      // Endereço na RAM onde o resultado foi armazenado
32 ROM[19] = `STA_DIR;   // Armazena o valor final em um endereço de saída
33 ROM[20] = 8'hE0;      // Endereço da saída
34 ROM[21] = `BRA;       // Salta para o início do programa (loop infinito)
35 ROM[22] = 8'h00;      // Endereço do início do programa
```

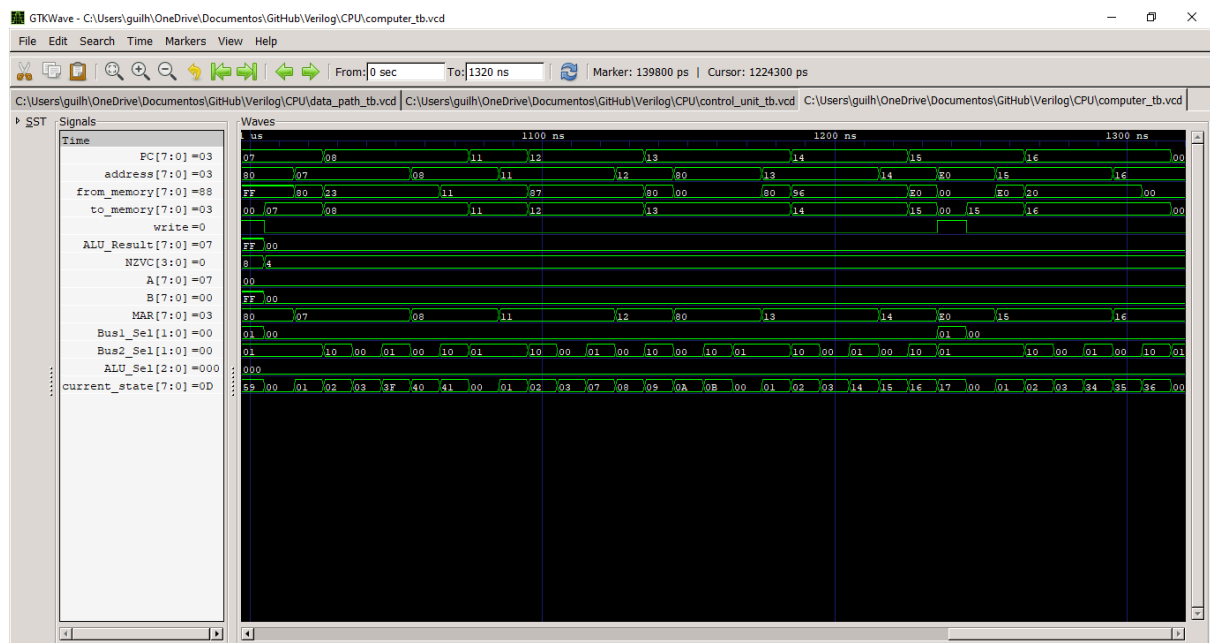
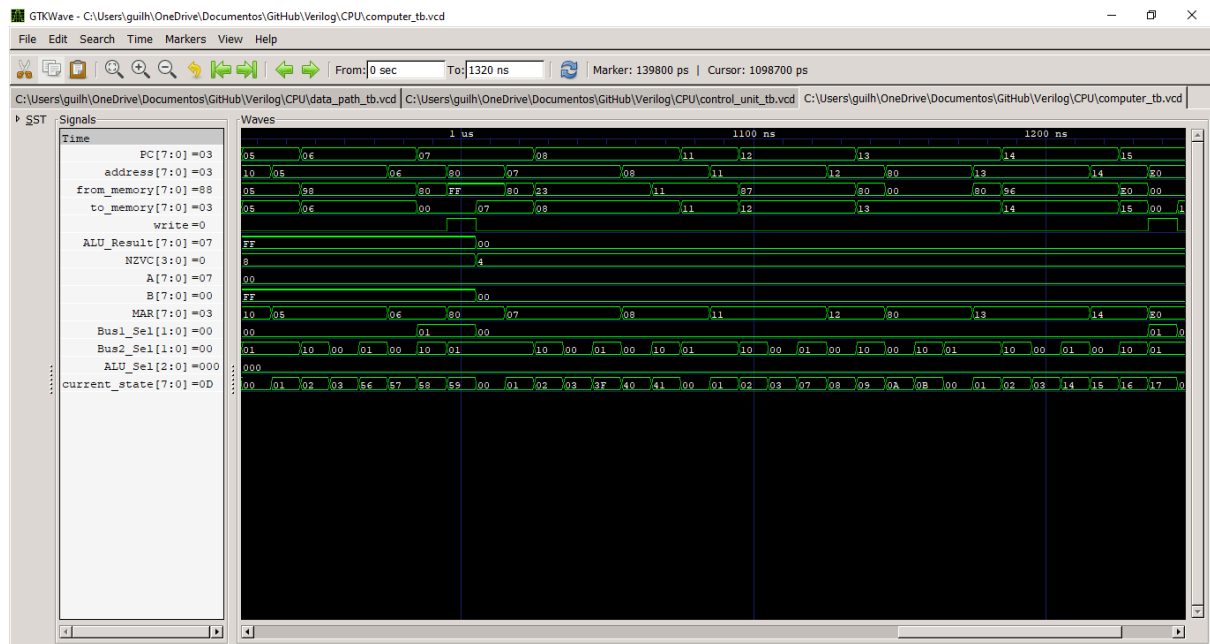
O testbench de execução do programa apenas resetava o computador para que funcionasse corretamente, pois não usamos portas IO para execução do programa, porém elas poderiam ser utilizadas sem quaisquer problemas.

As formas de onda de resultado do GTKWave mostraram que o computador conseguiu obter um resultado satisfatório, anulando a conta inicial e voltando ao loop do começo do programa. Em um caso de um computador real a subtração  $A - B$  seria alterada conforme os inputs do usuário na IO do computador e com isso o

Formas de onda de resultado:







Para melhor entendimento, as ações que o computador realiza são:

- 1 - Gravar um valor no registrador A, um valor no registrador B e realizar a subtração;
- 2 - Armazena o valor na RAM;
- 3 - Verifica que o resultado deu negativo;
- 4 - Faz um jump para o End (incrementar A);
- 5 - Faz um jump para armazenar o valor de A novamente;
- 6 - Faz os passos 3, 4 e 5 novamente, A agora é 0;
- 7 - Verifica que A é 0 e faz um jump para o End;
- 8 - Coloca A em um endereço de saída;
- 9 - Volta para o começo do loop.



## Auto Avaliação

O sucesso do projeto foi algo bem satisfatório para a equipe. Analisar um computador por dentro e desenvolvê-lo inicialmente parecia algo extremamente desafiador, mas com o desenvolvimento do projeto vimos que era possível. Quando finalmente conseguimos fazer o computador funcionar achamos muito interessante como cada coisa se relacionava para tudo dar certo no final. Acreditamos que talvez para melhorar o nosso próximo projeto talvez nós devêssemos inovar com criatividade, fazendo algo além do que foi pedido. Com base no sucesso que obtivemos, consideramos o nosso projeto nota 9.