UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ERICK TEIXEIRA PIRES

Uma implementação do CES

RIO DE JANEIRO
2019

ERICK TEIXEIRA PIRES

Uma implementação do CES

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Nelson Quilula Vasconcelos M.Sc

RIO DE JANEIRO
2019

CIP - Catalogação na Publicação

P667i

Pires, Erick Teixeira
Uma implementação do CES / Erick Teixeira Pires. - Rio de Janeiro, 2019.
120 f.

Orientador: Nelson Quilula Vasconcelos.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2019.

1. computadores. 2. circuitos lógicos. 3. arquitetura de computadores. 4. linguagens de programação de baixo nível. I. Vasconcelos, Nelson Quilula, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

ERICK TEIXEIRA PIRES

Uma implementação do CES

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 30 de Julho de 2019

BANCA EXAMINADORA:

Nelson Quilula Vasconcelos
M.Sc. (UFRJ)

Gabriel Pereira da Silva
D.Sc. (UFRJ)

Paulo Henrique de Aguiar Rodrigues
Ph.D. (UFRJ)

Dedico esse trabalho a todos os homens e mulheres que criaram e tornaram a Computação uma realidade. Nosso trabalho só é possível pois estamos sobre ombros de gigantes.

AGRADECIMENTOS

Agradeço à UFRJ e aos professores do Departamento de Ciência da Computação por todo o conhecimento que me foi passado durante a minha graduação. Ao professor Nelson Quilula Vasconcelos, que me orientou durante esse trabalho, sempre me ajudando a esclarecer dúvidas e a encontrar o modo certo de resolver os problemas encontrados.

Agradeço aos meus amigos Alexandre Pierre, Diego Tertuliano, Gustavo Monteiro, Iago Leal e Pedro Aragão pelas muitas conversas e discussões. Todas elas certamente me ensinaram bastante e me ajudaram a esclarecer muitos pensamentos que foram utilizados nesse trabalho.

Agradeço também ao meu amigo Wesley Alves pela ajuda revisando o texto deste trabalho.

"In this sense, an object is of
the highest degree of complexity if it
can do very difficult and involved things."

John von Neumann

RESUMO

Os computadores modernos são sistemas complexos, e isso cria uma distância entre os conceitos de Programação em Linguagens de Baixo Nível e os de Circuitos Digitais. Esse projeto tem como objetivo implementar um computador muito simples utilizando os elementos básicos da lógica combinacional e sequencial em uma tentativa de encurtar a distância cognitiva entre essas duas áreas da Computação. Para isso foram utilizados circuitos integrados derivados da família 74, que são comumente empregados para o ensino de Circuitos Lógicos. A montagem foi feita em *protoboards*, também familiares aos estudantes. O computador possui apenas um modo de endereçamento, um único acumulador, apenas 4 instruções, o que torna fácil o entendimento total do seu funcionamento. O computador implementa a arquitetura de Von Neumann, que é a base dos computadores modernos. Um simulador desse computador é aplicado no ensino de uma disciplina do curso de graduação em Ciência da Computação na Universidade Federal do Rio de Janeiro desde 2008.

Palavras-chave: computadores. circuitos lógicos. arquitetura de computadores. linguagens de programação de baixo nível.

ABSTRACT

Modern computers are complex systems, and this fact creates some distance between the concepts of Low-level Programming and Digital Circuits. The goal of this work is to implement a very simple computer using the basic elements of combinational and sequential logic in an attempt to bridge the cognitive gap between these two areas of Computing. To implement this computer integrated circuits derived from the 74 family were used. These components are commonly used to teach Logic Circuits. The circuit was built using breadboards, also familiar to students. The computer has only one addressing mode, only one accumulator and only 4 instructions, making it easy to fully understand its working. The computer implements the Von Neumann architecture, that is the base for the modern computers. A simulator for this computer has been used to teach a discipline on the undergraduate course on Computer Science of the Federal University of Rio de Janeiro since 2008.

Keywords: computers. logic circuits. computer architecture. low-level programming languages.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BASIC	*Beginner's All-purpose Symbolic Instruction Code*
BD	*Buffer* das chaves de Dados
BE	*Buffer* das chaves de Endereço
BS	*Buffer* da entrada do Somador
CES	Computador Extremamente Simples
CI	Circuito Integrado
CMOS	*Complementary Metal–Oxide–Semiconductor*
DIP	*Dual In-Line Package*
E/S	Entrada e Saída
GND	*Ground*
I_{OL}	*Low-level output current*
KiB	*KibiByte*
LED	*Light-Emitting Diode*
LL(1)	*Left-to-right, Leftmost derivation parser (1 token lookahead)*
MES	Montador Extremamente Simples
PCB	*Printed Circuit Board*
RAM	*Random Access Memory*
RC	Registrador de Condição
RD	Registrador de Dados
RE	Registrador de Endereço
RISC	*Reduced Instruction Set Computer*
RI	Registrador de Instrução
ROM	*Read-Only Memory*
RP	Registrador de Programa

RT	Registrador de Trabalho
SDL	*Simple DirectMedia Layer*
SMD	*Surface Mounting Device*
SRAM	*Static Random Access Memory*
TTL	*Transistor-Transistor Logic*
UC	Unidade de Controle
ULA	Unidade Lógica/Aritmática
V_{cc}	*Voltage Common Collector*
VGA	*Video Graphics Array*

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO E OBJETIVOS

A complexidade dos computadores atuais dificulta a associação entre os conceitos apresentados em Circuitos Lógicos e o funcionamento de um computador. Para demonstrar de uma forma mais compreensível o funcionamento de um computador a partir de elementos lógicos, foi construído um computador simples a partir de elementos básicos da lógica combinacional e sequencial, utilizando principalmente componentes com integração em pequena ou média escala, da família TTL ou derivados desta família. Os únicos componentes que usam integração em larga escala são empregados para implementar a memória. Para a construção, foi escolhido o Computador Extremamente Simples (CES) descrito pelo professor Nelson Quilula Vasconcelos.

A descrição formal do CES pode ser encontrada em (VASCONCELOS, 2008 (Acessado em 5 fev. 2019)) e é explicada no Capítulo 2 dessa monografia. Para que o operador possa ter acesso a dados do computador, assim como inserir dados, um painel foi acrescentado à descrição inicial, e modificações tiveram que ser feitas. Além disso, foi acrescentada a opção de parar a execução da máquina, e isso também resultou em modificações. Todas essas modificações são descritas no terceiro capítulo dessa monografia.

No desenrolar do trabalho, foram criados códigos de teste e ferramentas para facilitar o desenvolvimento do computador. Todo material desenvolvido pode ser encontrado no seguinte repositório *Git*: *https://github.com/erickpires/CES*

1.2 TRABALHOS RELACIONADOS

Vários computadores simples que utilizam lógica TTL já foram criados. Alguns desses computadores serviram como inspiração para esse projeto.

1.2.1 Gigatron

O Gigatron (KERVINCK; BELGERS, 2017 (Acessado em mar. 2018)) é um computador minimalista que busca o interesse da comunidade *retro*. Ele é um produto comercial que é vendido como um *kit* constituído de uma placa de circuito impresso e circuitos integrados, e deve ser montado pelo comprador. Após montado ele pode ser utilizado para jogar alguns dos jogos inclusos, visualizar algumas imagens, rodar um programa para desenhar o conjunto de Mandelbrot e ter acesso a uma versão de BASIC.

O Gigatron possui arquitetura RISC, 8 *bits* de barramento de dados, 32K *bytes* de memoria RAM e 64K palavras de memória de programa. A sua Unidade Aritmética e

Lógica (ULA) possui 8 operações e 8 modos de endereçamento, além disso, o barramento pode ser tratado de 4 modos diferentes. Ele gera som e vídeo VGA por software.

Apesar de ter uma Unidade de Controle (UC) *hardwired*, o Gigatron ainda é bem mais complexo do que o CES.

1.2.2 Ben Eater

Ben Eater possui um canal no YouTube no qual produz conteúdo com objetivo de ensinar assuntos relacionados à eletrônica. Em uma série (EATER, 2016 (Acessado em mar. 2018)) de vídeos, ele mostra o passo-a-passo de como criar um computador simples utilizando lógica TTL em uma *protoboard*. A arquitetura do seu computador é bem expansível, por possuir uma UC micro-controlada. O computador possui 8 *bits* de dados e 4 *bits* endereço.

Em comparação ao CES, esse computador possui uma UC mais complexa, mas um barramento de endereços pequeno demais para construir algo mais complexo.

1.2.3 dreamcatcher

Um computador de 8 *bits* com um barramento de 8 *bits* construído em *protoboard* com CIs da família **74HC**. Possui uma UC micro-controlada com aproximadamente 32 instruções. O programa a ser executado é carregado atráves de *DIP switches*. Um vídeo de demonstração do projeto pode ser encontrado em (CONSTANTINO, 2017 (Acessado em abr. 2018)), e os diagramas do projeto estão em (CONSTANTINO, 2018 (Acessado em abr. 2018)).

1.3 ESTRUTURA DA MONOGRAFIA

O capítulo 2 descreve a arquitetura do CES.

O capítulo 3 descreve as modificações feitas na arquitetura para acomodar o painel e o paramento do processador.

O capítulo 4 descreve como o computador foi dividido em 4 placas de acordo com suas funções lógicas.

O capítulo 5 descreve como os elementos de hardware foram escolhidos dadas as limitações encontradas.

O capítulo 6 descreve a montagem do computador, assim como problemas encontrados.

O capítulo 7 descreve os testes que foram feitos durante a construção do computador.

O capítulo 8 descreve a ferramenta que foi criada para facilitar a disposição dos Circuitos Integrados nas *protoboards*.

O capítulo 9 descreve a ferramenta para montagem do código de máquina (MES).

O capílulo 10 apresenta as considerações finais.

2 CES: COMPUTADOR EXTREMAMENTE SIMPLES

O CES possui apenas um modo de endereçamento: o endereçamento direto, ou seja, todas as instruções usam um único operando: um endereço. A arquitetura é baseada em acumulador, e há apenas um registrador de propósito geral: o Registrador de Trabalho (RT) e um registrador de condição de apenas um *bit* (RC). A descrição completa do CES pode ser encontrada em (VASCONCELOS, 2008 (Acessado em 5 fev. 2019)).

2.1 INSTRUÇÕES

O CES pode ser considerado uma máquina RISC de apenas 4 instruções. Essas instruções são:

- Lê um valor de um endereço de memória para RT (mnemônico LE);
- Escreve o valor atual de RT em um endereço de memória (mnemônico ESC);
- Subtrai T do valor presente em uma posição de memória, escreve o resultado em RT e atualiza o RC com o valor de 'pede emprestado' (mnemônico SUB);
- Desvia para um determinado endereço caso tenha havido pede emprestado na última subtração (mnemônico DNP).

Apesar de possuir apenas 4 instruções, o CES é uma Máquina de Turing completa.

2.2 REGISTRADORES

Como já foi visto, o processador possui um único indicador, que indica se houve 'pede emprestado' na última subtração efetuada. Esse indicador é armazenado em um registrador denominado RC. O valor que é armazenado em RC corresponde ao complemento do 'vai um' de um somador, pois a subtração é implementada como a soma do minuendo com o complemento a dois do subtraendo.

As palavras de dados possuem 16 *bits*, o que significa que o registrador RT também possui 16 *bits*. O endereço é formado por 14 *bits*, o que significa que o processador é capaz de endereçar 16K palavras ou 32KiB.

O apontador de programas é armazenado no registrador RP que possui 14 *bits*.

O processador possui apenas 4 instruções diferentes, logo é possível codificar uma instrução com apenas 2 *bits*. Como cada instrução possui um único modo de endereçamento e inclui um endereço como operando, as duplas (código da instrução, endereço) podem ser codificadas utilizando 16 *bits*, sendo os 2 *bits* mais significativos responsáveis por codificar

a instrução, e os demais 14 *bits* responsáveis por armazenar o endereço. Com isso, podemos perceber que cada instrução ocupa o espaço equivalente a exatamente uma palavra, ou seja, 16 *bits*. O CES usa arquitetura de Von Neumann básica, ou seja, ele armazena o programa e os dados no mesmo espaço de endereçamento.

O CES emprega também 3 registradores invisíveis:

- RI: Registrador de Instrução. Possui 2 *bits* e é responsável por armazenar a instrução que o processador está executando em um dado momento;
- RD: Registrador de Dados. Possui 16 *bits* e age como um buffer, armazenando o último dado lido da memória;
- RE: Registrador de Endereços. Possui 14 *bits* e armazena o endereço que será usado no próximo acesso à memória.

2.3 UNIDADE LÓGICA E ARITMÉTICA

A ULA é constituída de apenas um somador, que é responsável tanto por realizar a subtração (utilizando complemento a dois) e por incrementar o valor de RP a cada ciclo do processador. Uma das entradas do somador é a saída de um complementador, que é responsável por gerar, seletivamente, ou complemento do valor presente em RT ou o valor zero.

2.4 EXECUÇÃO DAS INSTRUÇÕES

Cada ciclo de instrução é quase sempre realizado em dois ciclos do relógio do processador. A única exceção ocorre quando é efetuado um desvio. Nesse caso a instrução é executada em apenas um ciclo do relógio. Dessa forma, é possível considerar que a execução de cada instrução é dividida em duas etapas, sendo que cada etapa é realizada em um ciclo de relógio. Para o controle dessas duas etapas, a Unidade de Controle possui um *flip-flop* interno denominado **Estado**.

No início da primeira etapa, o endereço do operando da instrução está em RE, o código da operação está em RI e a dupla (código da instrução, endereço) está em RD. Para decidir se o processador irá desviar ou não, a UC utiliza tanto o valor armazenado em RI como o valor do indicador (armazenado em RC). Vamos discutir essas duas etapas para cada uma das instruções do processador.

Primeira etapa:

- LE e SUB: O valor que está em RE é utilizado como endereço para uma operação de leitura na memória. O valor de RP é incrementado em uma unidade utilizando o somador. Ao final dessa etapa, o resultado obtido da memória é armazenado em RD e o valor do incremento de RP é armazenado em RP e em RE;

- ESC: O valor que está em RE é utilizado como endereço para uma operação de escrita na memória. O valor que está em RT é colocado no barramento de dados para ser escrito na memória. O valor de RP é incrementado em uma unidade utilizando o somador. Ao final dessa etapa, o valor do incremento de RP é armazenado em RP e em RE;

- DNP sem desvio: O valor de RP é incrementado em uma unidade utilizando o somador. Ao final dessa etapa, o valor do incremento de RP é armazenado em RP e em RE;

- DNP com desvio: O valor que está em RE é utilizado para buscar a próxima instrução na memória. Os 14 *bits* menos significativos do valor que está em RD são copiados para RP atráves do somador, fazendo uma soma entre o valor de RD e zero. Ao final dessa etapa, a condição inicial é obtida, ou seja, o valor do endereço do operando da próxima instrução está em RE, o código da próxima instrução está em RI e a dupla (código da instrução, endereço) está em RD.

Segunda etapa:

- LE: O valor que está em RE é utilizado para buscar a próxima instrução na memória. Ao final dessa etapa, o valor que está em RD é copiado para RT atrás de uma soma com o valor zero, e a condição inicial descrita anteriormente é obtida;

- SUB: O valor que está em RD é somado com o complemento a dois do valor que está em RT. O valor que está em RE é utilizado para buscar a próxima instrução na memória. Ao final dessa etapa o valor de RT é atualizado com o valor da subtração, e a condição inicial descrita anteriormente é obtida.

- ESC e DNP sem desvio: O valor que está em RE é utilizado para buscar a próxima instrução na memória. Ao final dessa etapa, a condição inicial descrita anteriormente é obtida;

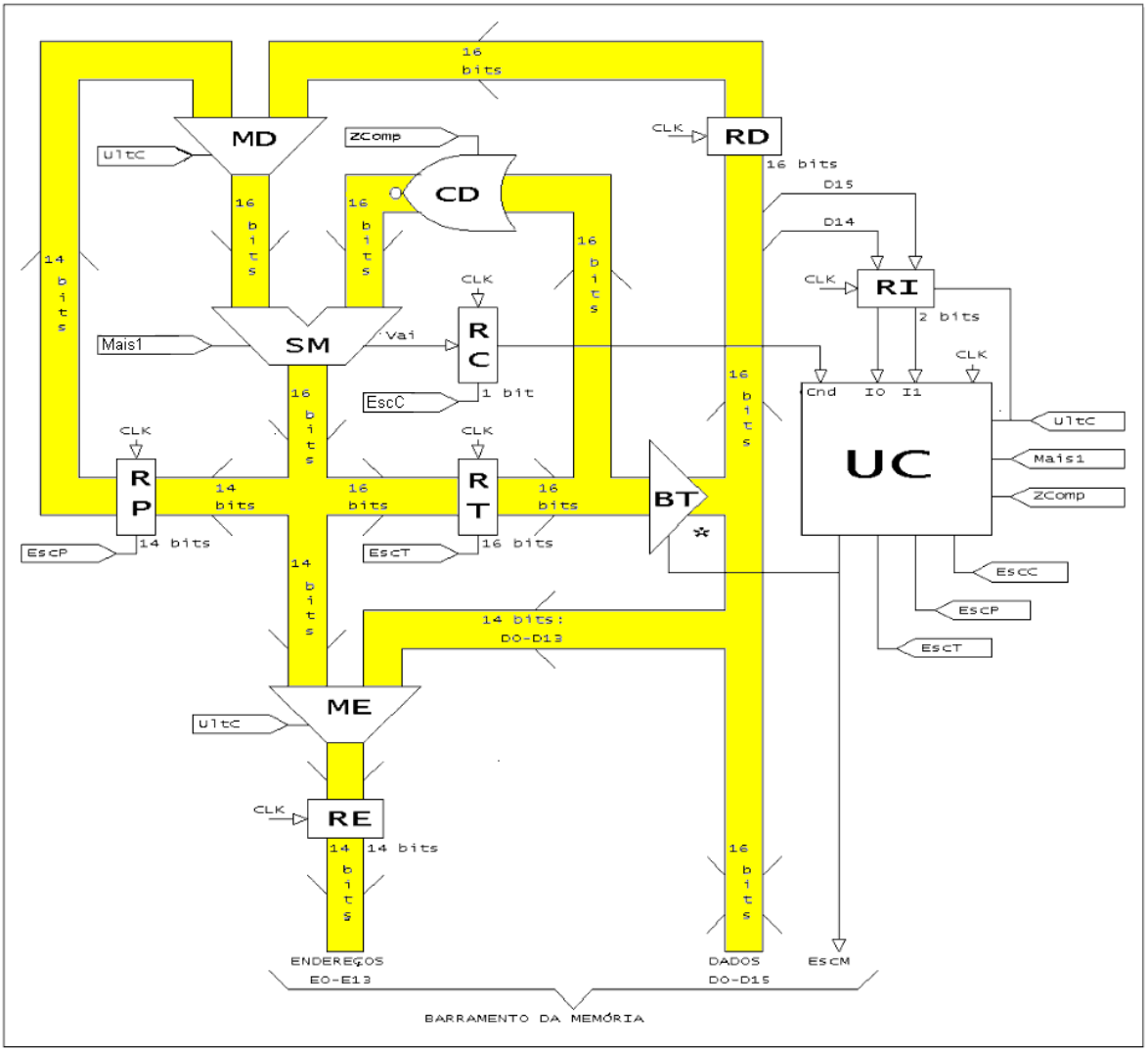
- DNP com desvio: Não ocorre.

2.5 ARQUITETURA DO CES

Um diagrama da arquitetura do CES pode ser encontrado na Figura 1.

Analisando o diagrama da Figura 1, podemos entender como certas operações são realizadas. Para incrementar o valor de RP, soma-se o valor atual de RP na entrada esquerda com o valor zero na entrada direita e ativa-se o sinal **Mais1**. Para salvar o valor lido da memória em RP ou RT, soma-se esse valor na entrada esquerda com o valor zero na entrada direita. Para realizar uma subtração, o minuendo é somado na entrada esquerda do somador com o complemento do valor de RT na entrada direita, e o sinal

Figura 1 – Diagrama da arquitetura do CES



Fonte: Vasconcelos, Nelson Quilula (2008)

Mais1 é ativado. Isso é necessário para realizar a computação do complemento-a-dois do valor RT, que é obtido invertendo os *bits* de RT e somando o valor 1.

2.6 UNIDADE DE CONTROLE

A UC recebe como entrada os sinais:

- *RC*: O valor do indicador de 'vai um';
- I_0 e I_1: Os dois *bits* que codificam a instrução atual;
- *Clk*: O relógio principal do processador.

E produz os seguintes sinais de controle:

- $UltC$: Último ciclo. Esse sinal tem valor 1 quando o atual ciclo de relógio corresponde à última etapa da instrução que está sendo executada;
- $Mais1$: Esse sinal é ligado diretamente ao sinal *carry-in* do somador e tem valor 1 quando o somador deve somar 1 ao resultado da soma atual;
- $ZComp$: Quando esse sinal é 1, o valor zero é sempre enviado para a entrada direita (entrada B) do somador. Quando esse sinal tem valor 0, o complemento do valor de RT é enviado à entrada direita do somador;
- $EscC$: O valor do sinal *carry-out* do somador só é copiado para o RC se esse sinal tiver valor 1;
- $EscP$: O valor obtido do somador pode ser salvo em RP seletivamente utilizando esse sinal;
- $EscT$: O valor obtido do somador pode ser salvo em RT seletivamente utilizando esse sinal;
- $EscM$: Esse sinal indica se a operação atual na memória é de leitura (valor 0) ou de escrita (valor 1);

Como já foi citado, a UC emprega um *flip-flop* denominado **Estado**, que é usado para indicar em qual etapa da execução da instrução atual o processador está. A saída não invertida desse *flip-flop* é chamada $\mathbf{Est_1}$.

Analisando quando cada sinal de saída precisa estar ativado ou desativado considerando todas as combinações dos sinais de entrada e o valor de $\mathbf{Est_1}$ podemos construir a Tabela 1.

Tabela 1 – Sinais de controle

RI	**RC**	$\mathbf{Est_1}$	$EscM$	$UltC$	$EscP$	$EscT$	$EscC$	$ZComp$	$Mais1$
00	**x**	**0**	0	0	1	0	0	1	1
00	**x**	**1**	0	1	0	1	0	1	0
01	**x**	**0**	1	0	1	0	0	1	1
01	**x**	**1**	0	1	0	0	0	X	X
10	**x**	**0**	0	0	1	0	X	1	1
10	**x**	**1**	0	1	0	1	1	0	1
11	**0**	**0**	0	1	1	0	0	1	0
11	**0**	**1**	X	X	X	X	X	X	X
11	**1**	**0**	0	0	1	0	0	1	1
11	**1**	**1**	0	1	0	0	0	X	X

Fonte: Vasconcelos, Nelson Quilula (2008)

Utilizando a Tabela 1 e mapas de Karnaugh, podemos derivar as expressões lógicas de cada um dos sinais gerados pela Unidade de Controle, como apresentado na Tabela 2.

Tabela 2 – Expressões dos sinais de controle

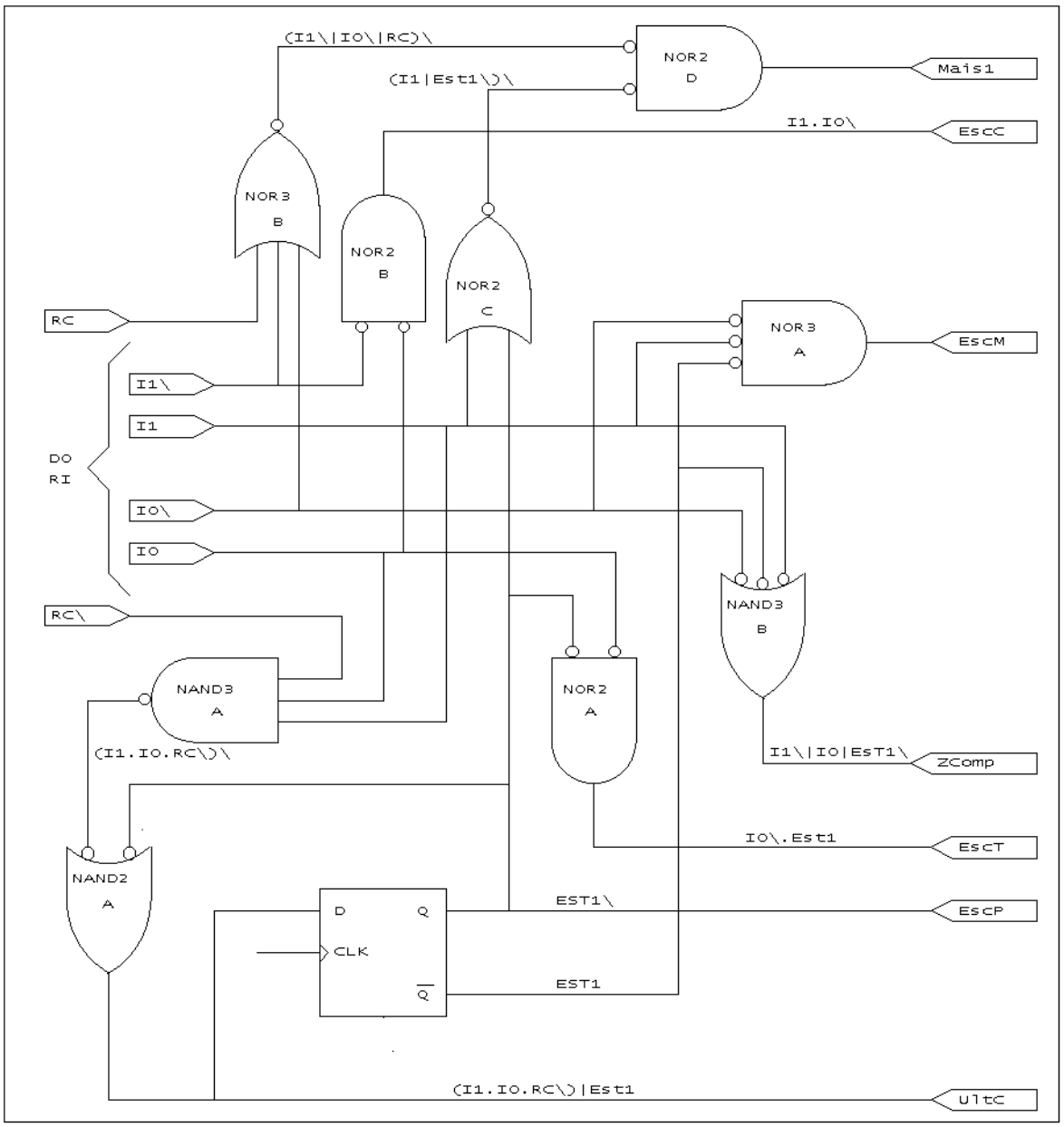
Sinal	**Expressão lógica**	**Implementação**
EscM	$\overline{I_1}.I_0.\overline{Est_1}$	$Nor3(I_1, \overline{I_0}, Est_1)$
UltC	$(I_1.I_0.\overline{RC})\|Est_1$	$Nand2(Nand3(I_1, I_0, \overline{RC}), \overline{Est_1})$
EscP	$\overline{Est_1}$	$\overline{Est_1}$
EscT	$\overline{I_0}.Est_1$	$Nor2(I_0, \overline{Est_1})$
EscC	$I_1.\overline{I_0}$	$Nor2(\overline{I_1}, I_0)$
ZComp	$\overline{I_1}\|I_0\|\overline{Est_1}$	$Nand3(I_1, \overline{I_0}, Est_1)$
Mais1	$(I_1\|\overline{Est_1}).(\overline{I_1}\|\overline{I_0}\|RC)$	$Nor2(Nor2(I_1, \overline{Est_1}), Nor3(\overline{I_1}, \overline{I_0}, RC))$

Fonte: Vasconcelos, Nelson Quilula (2008)

Utilizando essas expressões, podemos construir um diagrama para a Unidade de Controle, como o mostrado na Figura 2.

Observação: O único *flip-flop* empregado corresponde ao *flip-flop* de **Estado**, mas para simplificar a implementação esse *flip-flop* armazena realmente o complemento do estado.

Figura 2 – Diagrama da Unidade de Controle



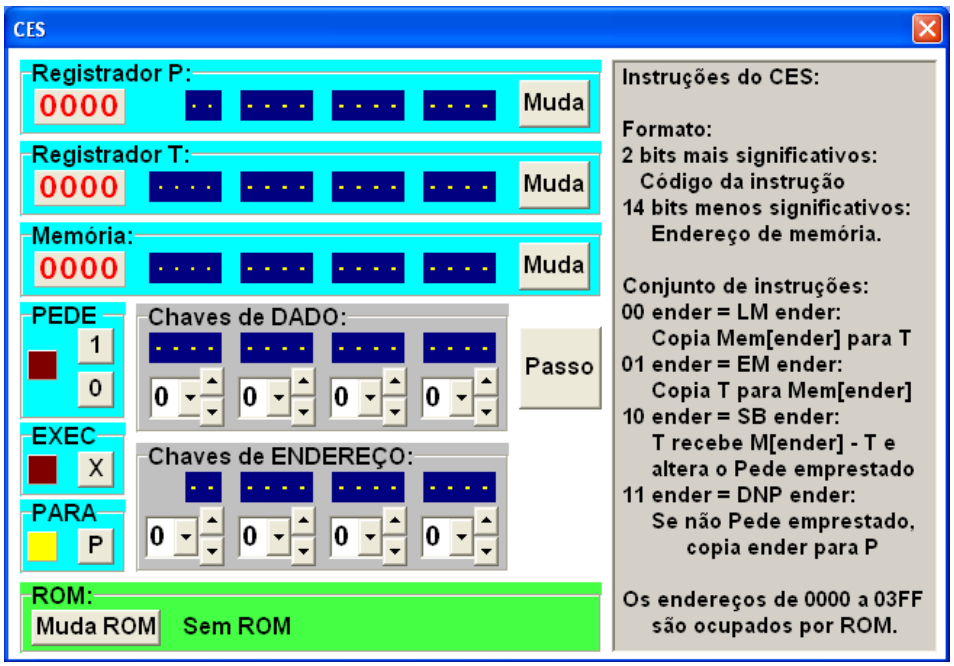
Fonte: Vasconcelos, Nelson Quilula (2008)

3 MODIFICAÇÕES PARA ACOMODAR O PAINEL E A PARADA DO PROCESSADOR

3.1 SOBRE O PAINEL DO SIMULADOR

O simulador do CES possui um painel através do qual é possível visualizar e controlar os valores dos registradores RP, RT, RC, assim como valores em qualquer endereço de memória correspondente à memória RAM. Uma imagem do simulador é mostrada na Figura 3.

Figura 3 – Simulador do CES



O registrador RC (Pede) muda de cor dependendo do seu estado e é controlado por dois botões (0 e 1) que, ao serem pressionados, mudam o seu estado.

Os registradores RT e RP possuem "lâmpadas" que mostram o valor presente no registrador em questão. Além disso, possuem o botão "Muda" que ao ser pressionado faz o valor presente nas "Chaves de Dados" ser copiado para o registrador em questão.

A memória emprega um botão "Muda" similar ao dos registradores RT e RP. Porém, quando este botão é pressionado, o valor presente nas "Chaves de Dados" é copiado para a posição de memória cujo endereço está presente nas "Chaves de Endereço". As "lâmpadas" da memória mostram o valor que está presente no endereço de memória correspondente às "Chaves de Endereço".

O simulador conta ainda com mais três botões, "Exec", "Para" e "Passo". O botão "Para" muda o computador do modo *executando* para o modo *parado*. Já o botão "Exec" muda o computador do modo *parado* para o modo *executando*.

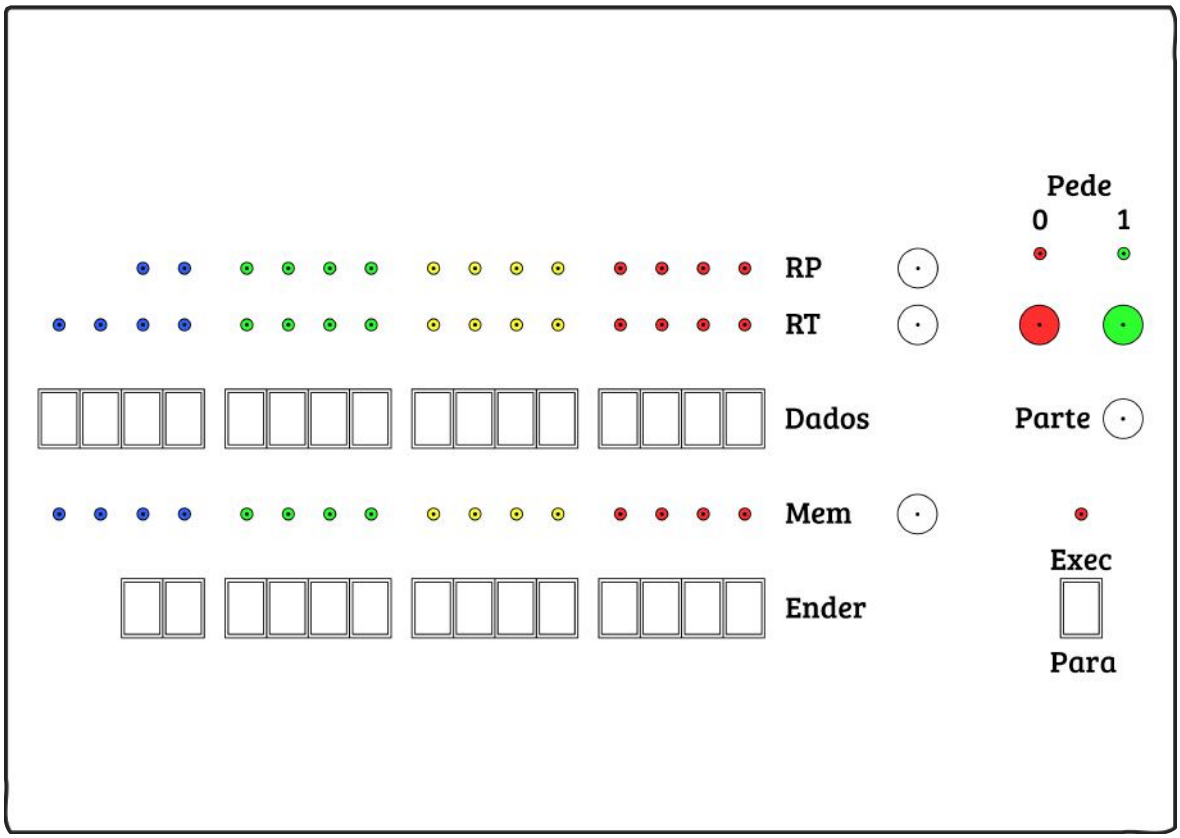
Quando está no modo *executando*, o computador executa instruções normalmente. Quando o computador muda para o modo *parado*, este termina de executar a instrução atual e para de executar. Ao mudar do modo *parado* para o modo *executando*, o computador volta a executar instruções normalmente.

O botão "Passo" só pode ser utilizado quando o computador está no modo *parado*. Quando pressionado, este botão faz com que o computador execute apenas uma instrução e volte a parar.

3.2 SOBRE O PAINEL PROJETADO PARA O CES

As funcionalidades oferecidas pelo painel do simulador são de grande importância para que um usuário possa operar o computador. Por este motivo, um painel similar ao presente no simulador foi projetado para a implementação física do CES. A Figura 4 mostra um desenho esquemático deste painel.

Figura 4 – Painel



As "lâmpadas" dos registradores e memória são implementadas com *LEDs*. Esses *LEDs* são separados em grupos de quatro cores para facilitar a leitura dos números. Ao lado dos registradores RT e RP e da memória, há um botão em formato circular que possui a mesma função dos botões "Muda" no simulador.

O registrador RC (Pede) é visualizado com dois *LEDs*, um vermelho e outro verde, que indicam qual é o estado do registrador, 0 e 1, respectivamente. Abaixo desses *LEDs*

há dois botões circulares, um vermelho e outro verde, que mudam o estado do registrador RC.

As “Chaves de Dados” e “Chaves de Endereço” são implementadas utilizando chaves retangulares, similares à da Figura 5. As chaves são separadas em grupos de quatro, também para facilitar a leitura.

Figura 5 – Chave utilizada no painel

O botão “Para” se tornou uma chave que agora altera o computador entre os modos *parado* e *executando*. O botão “Parte” substituiu o botão “Passo”. Esse novo botão tem uma função um pouco diferente que será descrita abaixo.

3.2.1 Diferenças entre os painéis

Devido à forma como o registrador RP é conectado, não é possível obter o seu valor sem que haja uma grande modificação. Seria necessário inserir um buffer de três estados na saída de RP, o que implica em no mínimo mais dois circuitos integrados e mais de 20 conexões. Para evitar essa modificação os *LEDs* de RP mostram na realidade a saída do somador. Quando o processador está na primeira etapa do seu ciclo, ou seja, quando o *flip-flop* **Estado** tem valor 0, o valor na saída do somador é sempre igual ao valor em RP. Quando o computador está no modo *parado* o *flip-flop* **Estado** tem sempre valor 0 e, portanto, o valor lido nos *LEDs* de RP corresponderam ao valor em RP. Quando o computador está no modo *executando* o valor nos *LEDs* não tem esse significado, mas isso não é um problema, visto que os valores mudarão rápido demais para serem lidos.

Outra modificação que foi realizada é em relação ao comportamento da chave “Para” e do botão “Parte”. Agora a chave “Para” muda o computador entre os modos *parado* (quando está na posição *desligada*) e *executando* (quando está na posição *ligada*). Quando o computador está no modo *executando* e muda para o modo *parado*, este termina de executar a instrução atual e para de executar. Mudar o modo do computador de *parado*

para *executando* não faz com que o computador volte a executar instruções, apenas o deixa preparado para isso.

O botão "Parte" possui duas funções a depender do modo do processador. Quando o computador está no modo *parado*, o acionamento deste faz com que o computador execute apenas uma instrução, da mesma forma que o botão "Passo" fazia no painel do simulador. Quando o computador está no modo *executando*, porém, não está executando nenhuma instrução (ou seja, a chave "Para" foi alterada da posição *desligada* para a posição *ligada*) o botão "Parte" faz com que o computador comece a executar efetivamente instruções. Pressionar o botão "Parte" quando o computador está no modo *executando* e está executando instruções não possui efeito algum.

3.3 ALTERAÇÕES NA ARQUITETURA DO CES

O painel adiciona a possibilidade de alterar valores de registradores e de posições de memória. Para acomodar essas novas funcionalidades, a arquitetura do CES vista na Figura 1 teve que ser alterada. Após aplicadas essas alterações, obteve-se a arquitetura da Figura 6. Omitidos na imagem estão os *clocks* dos registradores e a Unidade de Controle, pois esta será descrita com mais detalhe na seção 3.5.

3.4 A PARADA DO PROCESSADOR

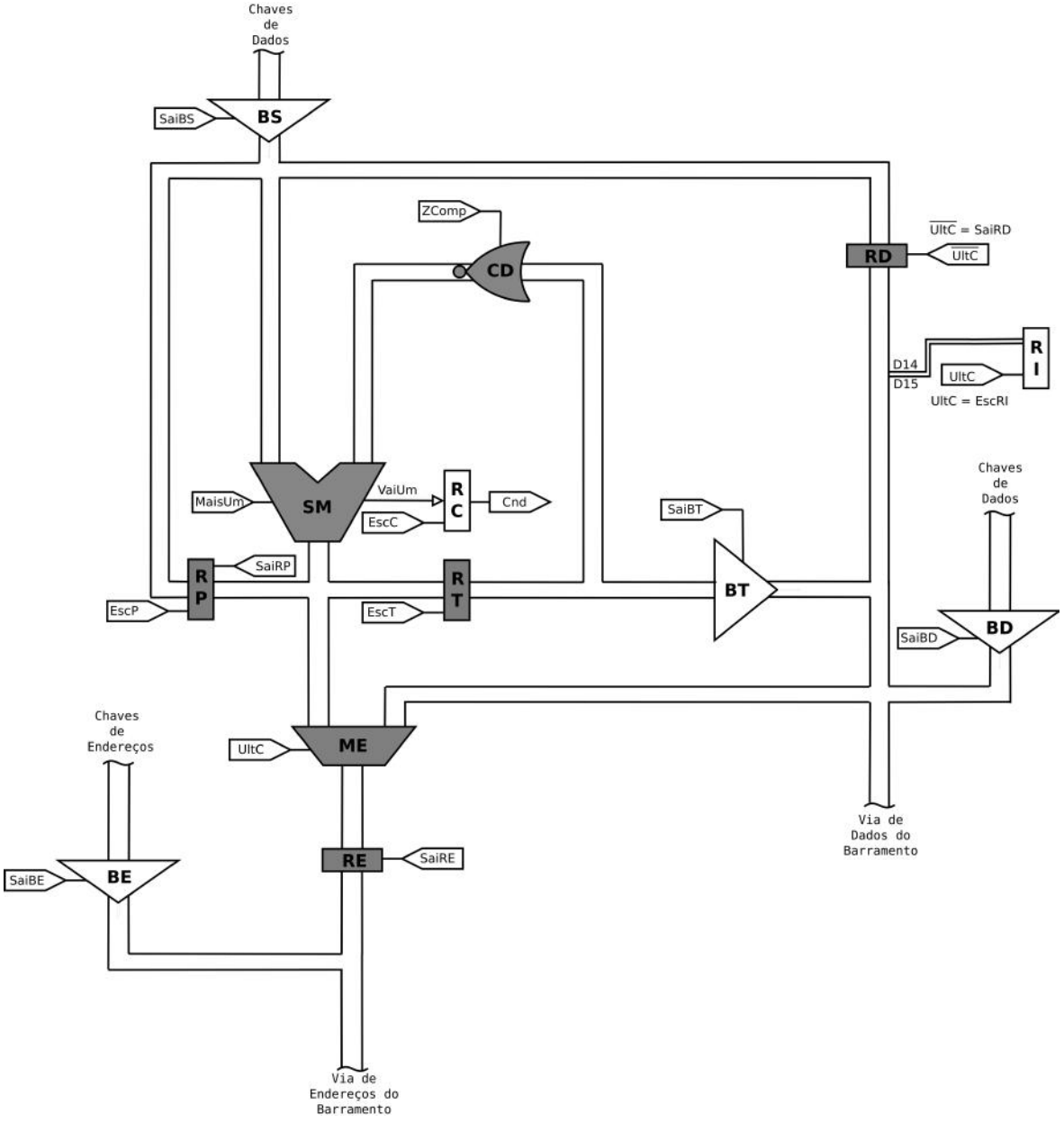
Um dos requisitos do painel é que a execução do computador possa ser parada. Um dos métodos que podemos pensar para atingir esse objetivo é interromper o *clock* do processador. Infelizmente essa não é uma estratégia possível para o CES por dois motivos principais.

O primeiro deles é que, como descrito anteriormente, cada instrução no CES leva, em geral, dois ciclos de *clock* para ser executada. Se simplesmente parássemos o *clock* do processador, poderíamos parar o processador enquanto uma instrução estivesse sendo executada. Logo, esta instrução nunca terminaria de executar e o processador estaria em um estado potencialmente instável. Um circuito especial precisaria ser desenvolvido para garantir que a parada fosse realizada somente quando o processador terminasse de executar uma instrução.

Outro fator menor que também precisaria ser levado em consideração é que ao interromper o *clock*, este teria o seu valor lógico fixado (ou em 0 ou em 1). Caso esse valor fosse oposto ao presente no momento da parada, o transição poderia ser interpretada como um novo ciclo de *clock*.

O segundo problema é o fato de que os valores de certos registradores podem ser alterados enquanto o computador está parado. Registradores precisam de um *clock* ativo para realizar leitura de um novo valor, portanto o *clock* do registrador cujo valor se

Figura 6 – Novo Arquitetura do CES



desejasse alterar precisaria ser gerado. Novamente, precisaríamos de um circuito especial, mas dessa vez para gerar *clocks* para os registradores a serem alterados.

Uma vez que a parada do processador acarretaria em desenvolvimento de algum circuito, e como a Unidade de Controle teria que ser modificada para acomodar a escrita em registradores, optou-se por alterar o comportamento da UC para acomodar também a parada do processador.

Três sinais de entrada foram adicionados à Unidade de Controle. O sinal *Para* indica que o computador está no estado *parado*. *Partiu* indica que o processador acabou de partir (e por isso não pode partir novamente). Por fim, *Parte* é um sinal que está ativo enquanto o processador está executando a pseudo instrução de partida descrita na Subseção 3.4.1.

3.4.1 Pseudo instrução de partida

Quando inicia sua partida, o processador não está necessariamente em um estado em que a execução possa começar imediatamente. Sabemos que o endereço pelo qual o processador deve começar a executar está em RE. Este fato é verdade pois enquanto o processador está parado, o valor da saída do somador, que nestas condições é igual ao valor de RP, é sempre selecionado pelo Multiplexador de Endereço (ME) e copiado para RE. No momento da partida, o processador precisa utilizar este valor para obter um estado válido em que a execução possa ocorrer normalmente. Este passo a mais pode ser interpretado como uma pseudo instrução que o processador executa durante a sua partida.

Primeiramente o valor em RE deve ser inserido no Barramento de Endereços para leitura da dupla (código da instrução, endereço). O código da instrução deve ser então copiado para RI, e o endereço do operando deve ser copiado para RE. Analisando a execução das instruções como descrito na Seção 2.4, podemos notar que esta pseudo instrução é equivalente à segunda etapa da execução das instruções *ESC* e *DNP* sem desvio.

3.5 ALTERAÇÕES NA UNIDADE DE CONTROLE

Observando a Figura 6, pode-se notar que novos sinais de controle foram acrescentados ao processador. Estes sinais são: $SaiBS$, $SaiRE$, $SaiRP$, $SaiBD$ e $SaiBE$.

Mais dois sinais também precisam ser gerados por uma restrição imposta pelos componentes utilizados. Os *flip-flops* utilizados para implementar os registradores RC e RI não possuem uma entrada para indicar se um determinado valor deve ser escrito no registrador em questão (no diagrama estes sinais são $EscC$ e $UltC$ para RC e RI respectivamente). Para obter escrita seletiva nestes registradores, é empregada a estratégia mencionada anteriormente de inibir o *clock* do registrador quando uma escrita não deve ocorrer. Para inibir o *clock* o sinal de controle de seleção para cada registrador é utilizado como entrada de uma porta $Nand$ de duas entradas. A outra entrada é *clock* do processador invertido. A saída dessas duas portas $Nand$ são nomeadas $RlgRC$ e $RlgRI$ e são ligadas às entradas *clock* de RC e RI respectivamente. Quando o sinal de seleção nessas portas tiverem valor lógico 1, a saída tem o mesmo valor que o *clock* do processador. Quando os sinais de seleção tiverem valor 0, as portas terão sempre o valor 1 na saída.

O antigo sinal $EscM$ não é mais utilizado para controlar diretamente a escrita na memória. Com a adição de um botão no painel que também controla a escrita na memória, o antigo sinal $EscM$ é combinado com um novo sinal gerado pelo botão no painel para gerar o novo sinal $EscMem$. Este sim é utilizado para controlar a escrita na memória.

Além dos sinais mencionados, a Unidade de Controle gera outros sinais que são utilizados internamente na geração dos demais sinais. Estes sinais são: $Desvia$, $BotParteAtivo$, $\overline{BotParteAtivo}$, $VaiPartir$ e $VaiParar$.

A adição do painel também resultou na adição de novas entradas para a lógica da Unidade de Controle. Estas novas entradas são: $MudaP$, $MudaT$, $BotEscMem$, $BotParte$, $IntPara$, $Parte$, $Partiu$ e $Para$. Estas novas entradas são obtidas da seguinte forma:

- $MudaP$: Saída complementar ($\overline{Q}$) de um *flip-flop* do tipo D cuja entrada é derivada do botão no painel que insere um valor em RP;
- $MudaT$: Saída complementar ($\overline{Q}$) de um *flip-flop* do tipo D cuja entrada é derivada do botão no painel que insere um valor em RT;
- $BotEscMem$: Saída complementar ($\overline{Q}$) de um *flip-flop* do tipo D cuja entrada é derivada do botão no painel que insere um valor na memória;
- $BotParte$: Este sinal entra na Unidade de Controle em sua forma complementar, e é obtido de um circuito de *debouncing* formado por um *Latch* ligado ao botão *Exec* e a resistores de *pull-up*;
- $IntPara$: Este sinal entra na Unidade de Controle em sua forma complementar e é obtido diretamente da chave (interruptor) que para o processador e um resistor de *pull-up*. A outra extremidade do interruptor é ligada ao *Terra* (GND) do circuito;
- $Parte$: Saída de um *flip-flop* do tipo D cuja entrada é o sinal $VaiPartir$;
- $Partiu$: Saída de um *flip-flop* do tipo JK cuja entrada J é o sinal $Parte$ e a entrada K é o sinal $\overline{BotParteAtivo}$;
- $Para$: Saída de um *flip-flop* do tipo JK cuja entrada J é o sinal $VaiParar$ e a entrada K é o sinal $Parte$.

Os *flip-flops* dos sinais $MudaP$, $MudaT$ e $BotEscMem$ são agrupados em um registrador denominado RS no diagrama das placas na Figura 13. A entrada D desses *flip-flops* é ligada ao botão do painel em questão e a um resistor de *pull-up*, a outra extremidade do botão é conectada ao *Terra* (GND) do circuito.

Na Tabela 3 encontra-se a nova forma de obter os antigos sinais de controle. Note que a adição do sinal interno $Desvia$.

Tabela 3 – Novas expressões dos sinais de controle

Sinal	Expressão lógica	Implementação
$Desvia$	$I_0.I_1.RC.\overline{Para}$	$And4(I_0, I_1, RC, \overline{Para})$
$\overline{EscM}$	$\overline{I_0.\overline{I_1}.\overline{Est_1}.\overline{Para}}$	$Nand4(I_0, \overline{I_1}, \overline{Est_1}, \overline{Para})$
$\overline{UltC}$	$\overline{Desvia\|Parte\|(Est_1.\overline{Para})}$	$Nor3(Desvia, Parte, Nor2(\overline{Est_1}, Para))$
$\overline{EscP}$	$\overline{MudaP\|(\overline{Est_1}, \overline{Para})}$	$Nor2(MudaP, Nor2(Est_1, Para))$
$\overline{EscT}$	$\overline{MudaT\|(\overline{I_0}.Est_1.\overline{Para})}$	$Nor2(MudaT, Nor3(I_0, \overline{Est_1}, Para))$
$EscC$	$I_1.\overline{I_0}.\overline{Para}$	$Nor3(\overline{I_1}, I_0, Para)$
$ZComp$	$\overline{I_1.\overline{I_0}.Est_1.\overline{Para}}$	$Nand4(I_1, \overline{I_0}, Est_1, \overline{Para})$
$Mais1$	$(I_1\|\overline{Est_1}).\overline{Desvia}.\overline{Para}$	$Nor3(Nor2(I_1, \overline{Est_1}), Desvia, Para))$

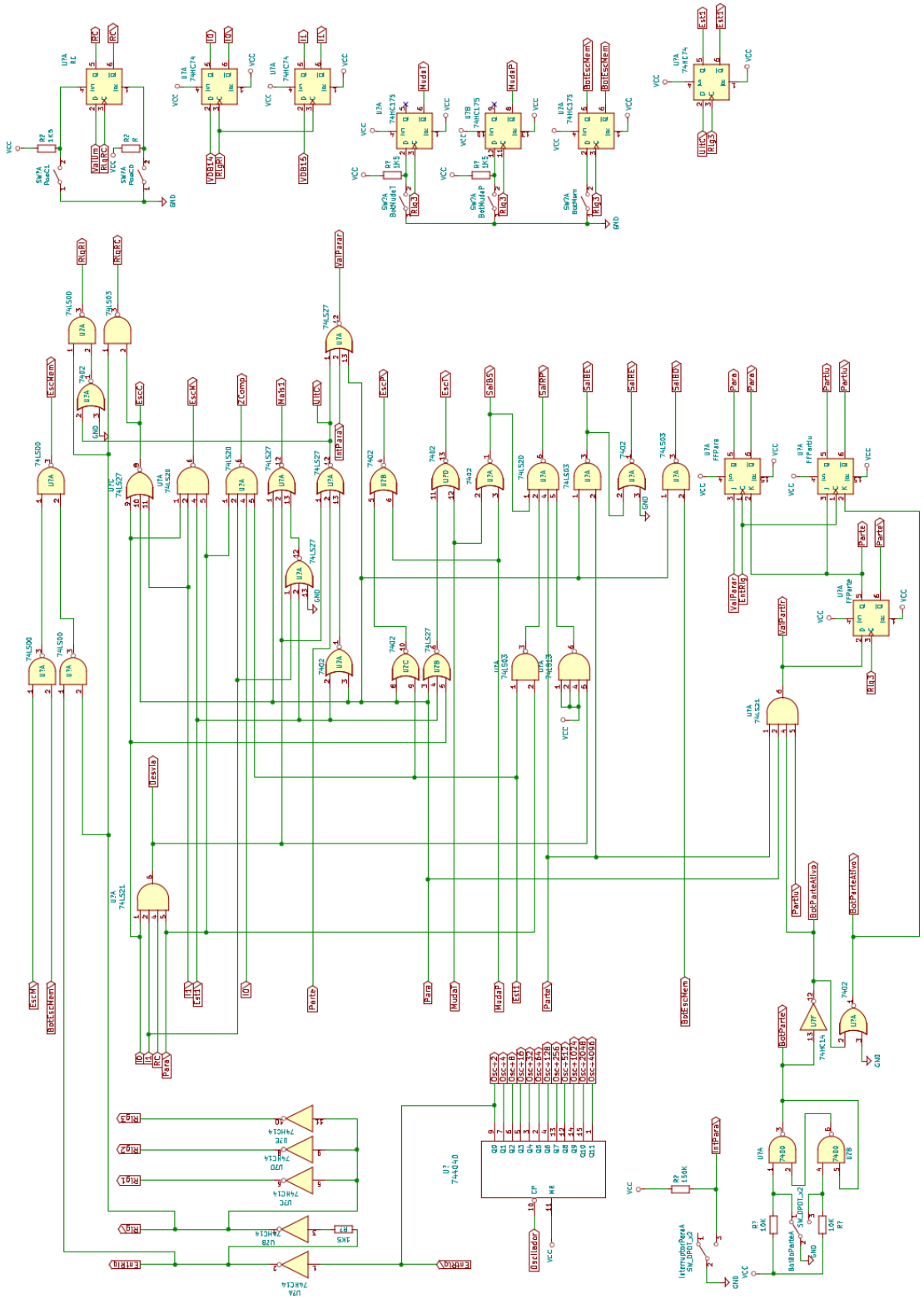
Agora, na Tabela 4 encontram-se os novos sinais de controle. Nesta tabela também estão descritas as expressões para geração do *clock* do processador como descrito na Subseção 3.6.2.

Tabela 4 – Novos sinais de controle

Sinal	Expressão lógica	Implementação
$EntRlg$	$\overline{\overline{EntRlg}}$	$NotST(\overline{EntRlg})$
$\overline{Rlg}$	$EntRlg + 18ns$	$NotST((EntRlg)15k\Omega)$
$Rlg_{1,2,3}$	$\overline{\overline{Rlg}}$	$NotST(\overline{Rlg})$
$BotParteAtivo$	$BotParte$	$NotST(\overline{BotParte})$
$\overline{BotParteAtivo}$	$\overline{BotParteAtivo}$	$Not(BotParteAtivo)$
$VaiPartir$	$BotParteAtivo.\overline{Parte}.$ $\overline{Partiu}.Para$	$And4(BotParteAtivo, \overline{Parte},$ $\overline{Partiu}, Para)$
$VaiParar$	$UltC.\overline{Para}.IntPara$	$Nor3(\overline{UltC}, Para, \overline{IntPara})$
$\overline{EscMem}$	$\overline{(EscM\|BotEscMem)\|}$ $\overline{(\overline{EntRlg}\|Rlg)}$	$Nand2(Nand2(EntRlg, \overline{Rlg}),$ $Nand2(\overline{EscM}, \overline{BotEscMem}))$
$\overline{SaiBD}$	$\overline{Para.BotEscMem}$	$Nand2(Para, BotEscMem)$
$\overline{SaiBE}$	$\overline{Para}\|Parte$	$Nand2(Para, \overline{Parte})$
$\overline{SaiBS}$	$\overline{MudaP\|MudaT}$	$Nor2(MudaP, MudaT)$
$\overline{SaiRE}$	$\overline{\overline{Para}\|Parte}$	$Nor2(\overline{Para}, Parte)$
$\overline{SaiRP}$	$(I_1.I_0.RC.\overline{Para})\|Parte\|$ $(Est_1.\overline{Para})\|SaiBS$	$Nand4(Nand2(Est_1, \overline{Para}), \overline{Parte},$ $Nand4(I_1, I_0, RC, \overline{Para}), \overline{SaiBS})$
$RlgRI$	$\overline{UltC.\overline{Rlg}}$	$Nand2(\overline{\overline{UltC}}, \overline{Rlg})$
$RlgRC$	$\overline{EscC.\overline{Rlg}}$	$Nand2(EscC, \overline{Rlg})$

Por fim, a Figura 7 mostra o novo diagrama esquemático da Unidade de Controle.

Figura 7 – Novo diagrama da Unidade de Controle



3.6 OUTRAS MODIFICAÇÕES REALIZADAS NO CES

3.6.1 O *buffer* dos *LEDs*

Como descrito anteriormente, as "lâmpadas" do painel foram implementadas com *LEDs*. Estes precisam mostrar os valores gerados pelos CIs do somador, de RT e da memória, sendo que a saída desses componentes são entradas de outros componentes no circuito. A corrente consumida por um *LED* pode ser ajustada modificando a resistência que é aplicada em série com o mesmo, mas, em geral, um valor próximo de 10 mA é utilizado.

Em um CI da família **74LS**, o valor máximo recomendado para corrente de saída no estado *low* (I_{OL}) é de 8 mA, como pode ser observado no *datasheet* para o **74LS04** em (INTRUMENTS, 2004 (Acessado em 23 jun. 2019)). O valor no estado *high* é muito menor devido à forma como a saída de um componente **74LS** é construída. Essa limitação cria um problema, pois os *LEDs* estariam utilizando a capacidade total de saída dos componentes, não deixando margem para os componentes que usam esses sinais como entrada. A solução foi utilizar algum componente entre cada *LED* e o sinal que este representa, servindo como um *buffer*. A saída de tal componente é dedicada a alimentar cada *LED*.

Como qualquer porta lógica poderia ser utilizada, por simplicidade optou-se por utilizar CIs **74LS04** (composto de 6 portas NOT) para implementar os *buffers*. Os *LEDs* têm os seus anodos conectados a um resistor de 1 $k\Omega$, que por sua vez tem a sua outra extremidade ligada em V_{cc}. O cátodo de cada *LED* é ligado à saída de um *buffer*. Quando o sinal na entrada do *buffer* tem valor 1, a saída do mesmo é levada para *Terra*, e o *LED* acende.

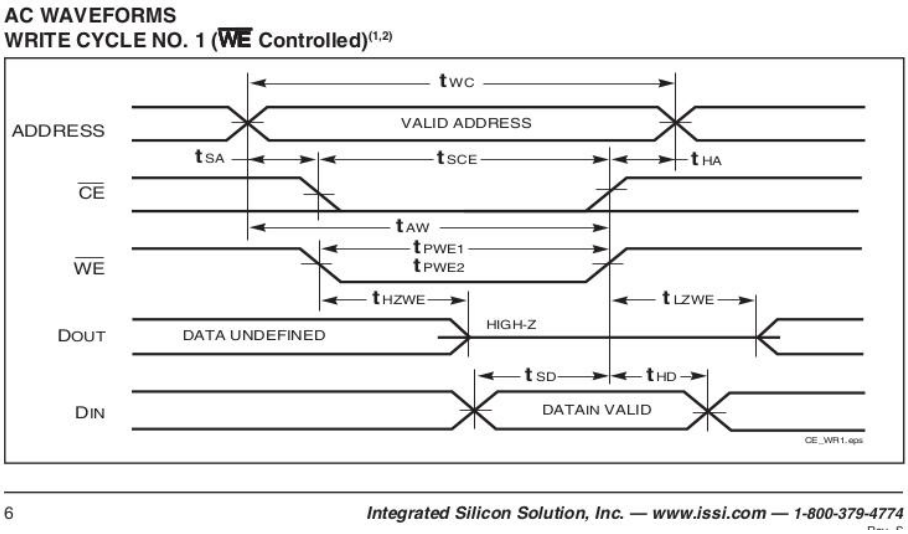
3.6.2 Sobre o *clock* do processador

A descrição inicial do CES não menciona o *clock* do processador. Porém, uma limitação do CI responsável pela memória RAM faz com que um circuito trivial para geração de *clock* não possa ser empregado.

A Figura 8 mostra o diagrama de tempo de uma operação de escrita no CI **61c256**. É possível observar que este componente exige que o endereço de escrita esteja válido antes que o sinal de escrita ($\overline{WE}$) seja ativado. O endereço para escrita é gerado, ou pelo Registrador de Endereços (RE), ou pelo Buffer de Endereços (BE), sendo que RE é o mais lento dos dois. Analisando o *datasheet* do CI utilizado para implementar RE, o **74LS173** (disponível em (INTRUMENTS, 1999 (Acessado em 6 jul. 2019))), observamos que o tempo de propagação do registrador é da ordem de 18ns.

Dada essa informação, sabemos que o sinal de escrita na memória só pode se tornar ativo após, no mínimo, 18ns. Para obter este resultado, foi criada uma "cópia" atrasada

Figura 8 – Formas de onda de um ciclo de escrita



Fonte: (SOLUTION, 2009 (Acessado em 6 jul. 2019))

do *clock* de entrada. Esse atraso é criado utilizando um circuito RC, porém, o capacitor do circuito é omitido e a capacitância inerente à *protoboard* é utilizada. Essa cópia atrasada é então combinada com o *clock* de entrada para gerar um atraso na ativação no sinal de escrita na memória.

Como um grande número de componentes da família **74LS** foi utilizado para implementar os registradores, a entrada de *clock* de todos os CIs não pode ser conectada à saída de uma única porta lógica, pois isso resultaria em uma carga maior do que o máximo recomendado. Para solucionar este problema são criados três *buffers* do *clock* do processador que são denominados Rlg_1, Rlg_2 e Rlg_3. As entradas de *clock* do circuito são então distribuídas entre esses três *buffers* de forma a obter uma carga semelhante em todos eles. Cada um desses sinais é gerado por um inversor de um CI **74HC14**, que garante subidas e descidas rápidas por possuir saídas do tipo *Schmitt-Trigger*.

Por conveniência foi acrescentado um contador binário de 12 *bits* (**74HC4040**) ao circuito de *clock*. O *clock* de entrada do processador é obtido de uma das saídas desse contador, e sua entrada é um oscilador de cristal de quartzo. Podemos então aumentar e diminuir a velocidade do processador em fatores que são potências de dois. Dependendo de qual saída do contador for escolhida, também estarão disponíveis *clocks* mais rápidos e mais lentos que o principal, sendo possivelmente úteis em dispositivos de Entrada e Saída.

3.6.3 Decodificação de Endereço

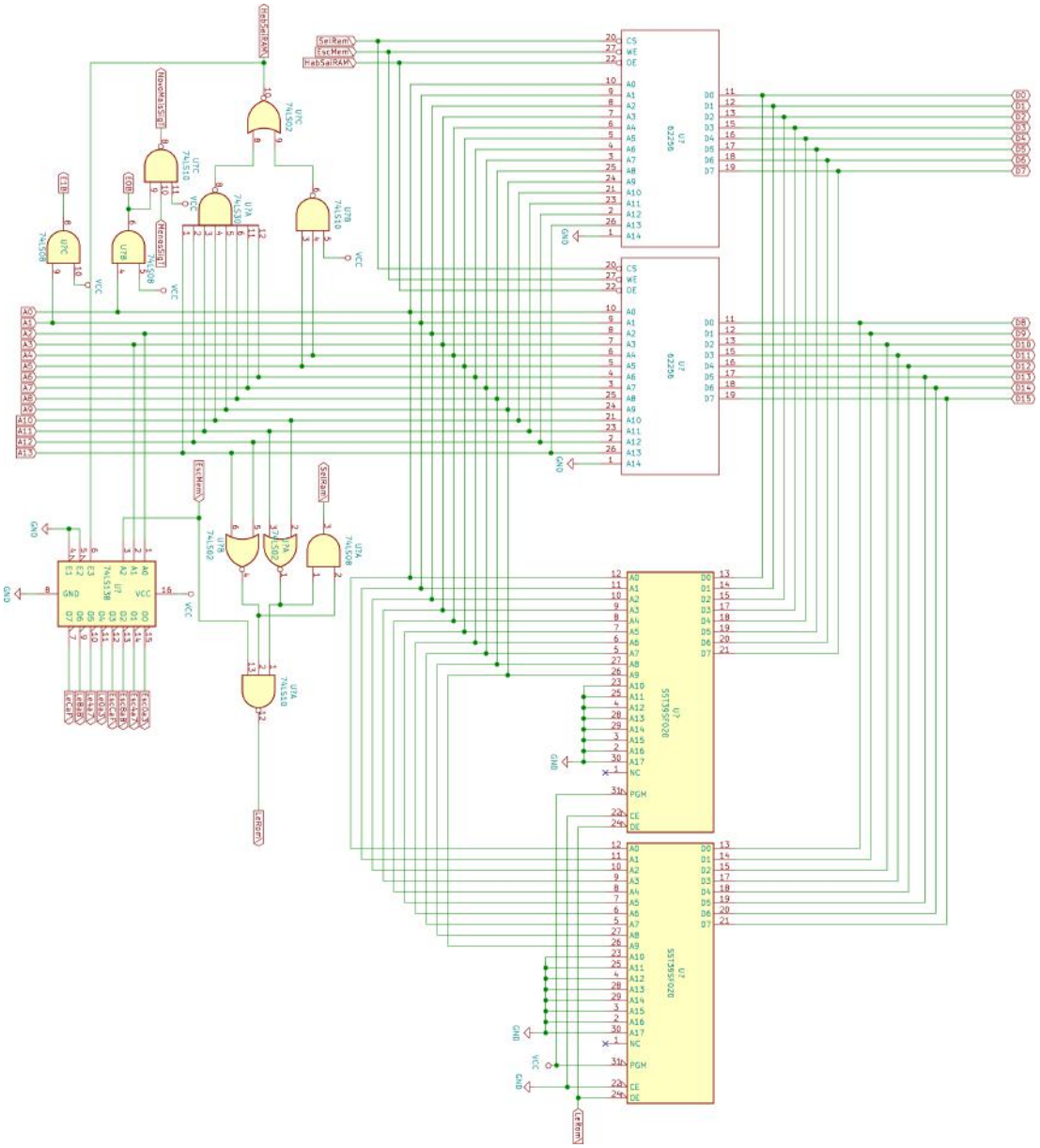
Inicialmente o espaço de endereçamento do CES era dividido em três regiões. A primeira região possui 1K palavras e é destinada a ROM. Os demais endereços com exceção dos últimos dois é destinada a RAM. Os últimos dois endereços de memória (ou seja,

0x3FFE e *0x3FFF*) são utilizados para endereçar interfaces de entrada e saída.

Neste trabalho a região da ROM foi mantida sem alteração, mas a região destinada a interfaces de entrada e saída foi expandida para 16 endereços. Essa modificação foi realizada por dois motivos: primeiramente, porque ela permitiu otimizar o número de componentes utilizados para decodificação de endereços e também porque mais endereços destinados a entrada e saída deixam o computador mais versátil.

Além dessa modificação, um decodificador foi adicionado para facilitar a seleção de possíveis dispositivos de E/S. Um diagrama do circuito de decodificação de endereços é mostrado na Figura 9.

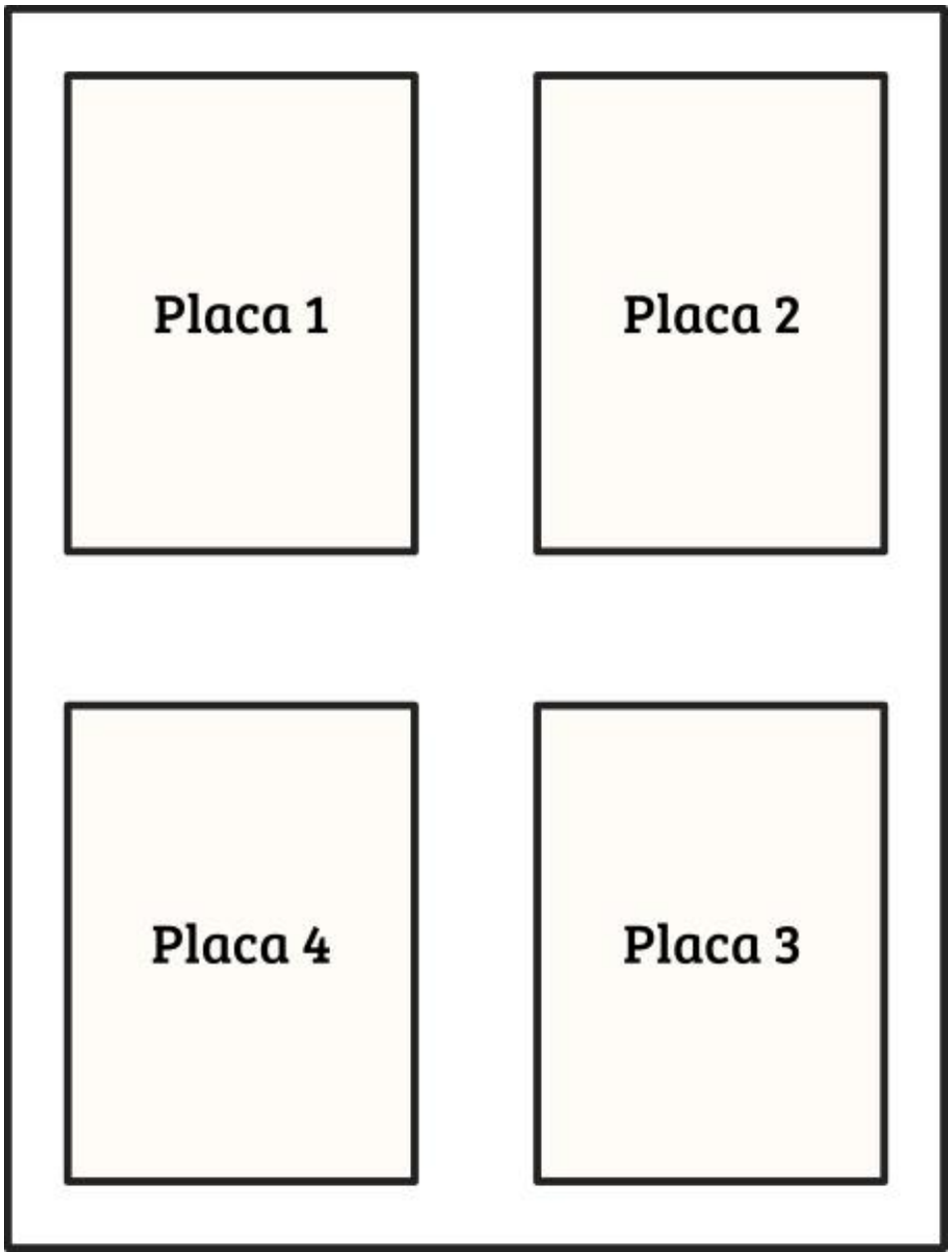
Figura 9 – Diagrama de Decodificação de Endereços



4 DIVISÃO EM PLACAS

A divisão em mais de uma placa(*protoboard*) foi necessária pois a quantidade de CIs para montar o computador é bem maior do que a quantidade de CIs que é possível dispor em uma das placas disponíveis no laboratório. Numa estimativa inicial foram contabilizados cerca de 70 CIs. A partir das dimensões das *protoboards* disponíveis, foi determinado que seriam necessárias 4 *protoboards* para a montagem. As 4 *protoboards* foram presas a um retângulo de madeira para dar sustentação e nomeadas Placa_1, Placa_2, Placa_3 e Placa_4 em sentido horário como mostrada a Figura 10.

Figura 10 – Distribuição das placas



Os componentes de cada placa foram colocados de forma que os de mesma função lógica estivessem na mesma placa, mas por limitação de espaço essa divisão lógica não foi totalmente possível. Dessa forma, os componentes foram divididos nas placas da seguinte forma:

- Placa_1: Registradores e somador. Nessa placa estão: Registrador de Trabalho (RT), Registrador de Dados (RD), Registrador de Programa (RP), Somador (SM), Complementador de Dados (CD) e Buffer da entrada esquerda do Somador (BS);
- Placa_2: Interface com o barramento. Nessa placa estão: Buffer de Dados (BD), Buffer de Endereços (BE), Buffer de RT (BT), Buffer dos LEDs (BL), Multiplexador de Endereços (ME) e Registrador de Endereços (RE);
- Placa_3: Unidade de Controle: Nessa placa estão: Divisor do Relógio, Registrador de Instrução (RI), Registrador de Carry (RC), *Flip-flop* de estado (Est_1), *Flip-flop* do sinal Partiu, *Flip-flop* do sinal Para, *Flip-flop* do sinal Parte, Registrador dos botões do painel, Unidade de Controle (constituída de 10 CIs);
- Placa_4: Decodificação de endereço e memórias. Nessa placa estão: Decodificador de endereço, Memória RAM e Memória ROM.

Os CIs de cada placa foram distribuídos utilizando a ferramenta descrita no Capítulo 8 e o diagrama de cada uma das placas pode ser encontrado nas Figuras 11, 12, 13 e 14.

Figura 11 – Placa 1

1	ESB0	CD1 74HC02	VCC	1	SS1	SM1 74LS283	VCC	1	UltC\	RD1 74LS173	VCC	1
2	SRT0		ESB1	2	ESB1		ESB2	2	GND(N)		GND(CLR)	2
3	ZComp		ZComp	3	ESA1		ESA2	3	ESA0		VDB0	3
4	ESB2		SRT1	4	SS0		SS2	4	ESA1		VDB1	4
5	SRT2		ESB3	5	ESA0		ESA3	5	ESA2		VDB2	5
6	ZComp		ZComp	6	ESB0		ESB3	6	ESA3		VDB3	6
7	GND		SRT3	7	MaisUm		SS3	7	Rlg2		GND(G2\)	7
8				8	GND		Vai3p4	8	GND		GND(G1\)	8
9	ESB4	CD2 74HC02	VCC	9	SS5	SM2 74LS283	VCC	9	UltC\	RD2 74LS173	VCC	9
10	SRT4		ESB5	10	ESB5		ESB6	10	GND(N)		GND(CLR)	10
11	ZComp		ZComp	11	ESA5		ESA6	11	ESA4		VDB4	11
12	ESB6		SRT5	12	SS4		SS6	12	ESA5		VDB5	12
13	SRT6		ESB7	13	ESA4		ESA7	13	ESA6		VDB6	13
14	ZComp		ZComp	14	ESB4		ESB7	14	ESA7		VDB7	14
15	GND		SRT7	15	Vai3p4		SS7	15	Rlg2		GND(G2\)	15
16				16	GND		Vai7p8	16	GND		GND(G1\)	16
17	ESB8	CD3 74HC02	VCC	17	SS9	SM3 74LS283	VCC	17	UltC\	RD3 74LS173	VCC	17
18	SRT8		ESB9	18	ESB9		ESB10	18	GND(N)		GND(CLR)	18
19	ZComp		ZComp	19	ESA9		ESA10	19	ESA8		VDB8	19
20	ESB10		SRT9	20	SS8		SS10	20	ESA9		VDB9	20
21	SRT10		ESB11	21	ESA8		ESA11	21	ESA10		VDB10	21
22	ZComp		ZComp	22	ESB8		ESB11	22	ESA11		VDB11	22
23	GND		SRT11	23	Vai7p8		SS11	23	Rlg2		GND(G2\)	23
24				24	GND		Vai11p12	24	GND		GND(G1\)	24
25	ESB12	CD4 74HC02	VCC	25	SS13	SM4 74LS283	VCC	25	UltC\	RD4 74LS173	VCC	25
26	SRT12		ESB13	26	ESB13		ESB14	26	GND(N)		GND(CLR)	26
27	ZComp		ZComp	27	ESA13		ESA14	27	ESA12		VDB12	27
28	ESB14		SRT13	28	SS12		SS14	28	ESA13		VDB13	28
29	SRT14		ESB15	29	ESA12		ESA15	29	ESA14		VDB14	29
30	ZComp		ZComp	30	ESB12		ESB15	30	ESA15		VDB15	30
31	GND		SRT15	31	Vai11p12		SS15	31	Rlg2		GND(G2\)	31
32				32	GND		VaiUm	32	GND		GND(G1\)	32
33	GND(M)	RT1 74LS173	VCC	33	SaiRP\	RP1 74LS173	VCC	33				33
34	GND(N)		GND(CLR)	34	GND(N)		GND(CLR)	34				34
35	SRT0		SS0	35	ESA0		SS0	35				35
36	SRT1		SS1	36	ESA1		SS1	36	SaiBS\	BS1 74HC244	VCC	36
37	SRT2		SS2	37	ESA2		SS2	37	CHD0		SaiBS\	37
38	SRT3		SS3	38	ESA3		SS3	38	ESA1		ESA0	38
39	Rlg1		GND(G2\)	39	Rlg1		GND(G2\)	39	CHD2		CHD1	39
40	GND		EscT\	40	GND		EscP\	40	ESA3		ESA2	40
41	GND(M)	RT2 74LS173	VCC	41	SaiRP\	RP2 74LS173	VCC	41	CHD4		CHD3	41
42	GND(N)		GND(CLR)	42	GND(N)		GND(CLR)	42	ESA5		ESA4	42
43	SRT4		SS4	43	ESA4		SS4	43	CHD6		CHD5	43
44	SRT5		SS5	44	ESA5		SS5	44	ESA7		ESA6	44
45	SRT6		SS6	45	ESA6		SS6	45	GND		CHD7	45
46	SRT7		SS7	46	ESA7		SS7	46				46
47	Rlg1		GND(G2\)	47	Rlg1		GND(G2\)	47				47
48	GND		EscT\	48	GND		EscP\	48				48
49	GND(M)	RT3 74LS173	VCC	49	SaiRP\	RP3 74LS173	VCC	49				49
50	GND(N)		GND(CLR)	50	GND(N)		GND(CLR)	50				50
51	SRT8		SS8	51	ESA8		SS8	51				51
52	SRT9		SS9	52	ESA9		SS9	52	SaiBS\	BS2 74HC244	VCC	52
53	SRT10		SS10	53	ESA10		SS10	53	CHD8		SaiBS\	53
54	SRT11		SS11	54	ESA11		SS11	54	ESA9		ESA8	54
55	Rlg1		GND(G2\)	55	Rlg1		GND(G2\)	55	CHD10		CHD9	55
56	GND		EscT\	56	GND		EscP\	56	ESA11		ESA10	56
57	GND(M)	RT4 74LS173	VCC	57	SaiRP\	RP4 74LS173	VCC	57	CHD12		CHD11	57
58	GND(N)		GND(CLR)	58	GND(N)		GND(CLR)	58	ESA13		ESA12	58
59	SRT12		SS12	59	ESA12		SS12	59	CHD14		CHD13	59
60	SRT13		SS13	60	ESA13		SS13	60	ESA15		ESA14	60
61	SRT14		SS14	61	ESA14		SS14	61	GND		CHD15	61
62	SRT15		SS15	62	ESA15		SS15	62				62
63	Rlg1		GND(G2\)	63	Rlg1		GND(G2\)	63				63
64	GND		EscT\	64	GND		EscP\	64				64

Figura 12 – Placa 2

1	SaiBD\	BD1 HC244	VCC	1	EscM\	BT1 HC244	VCC	1	SRT0	BL1 LS04	VCC	1
2	CHD0		SaiBD\	2	SRT0		EscM\	2	LedsRT0		SRT3	2
3	VDB1		VDB0	3	VDB1		VDB0	3	SRT1		LedsRT3	3
4	CHD2		CHD1	4	SRT2		SRT1	4	LedsRT1		SRT4	4
5	VDB3		VDB2	5	VDB3		VDB2	5	SRT2		LedsRT4	5
6	CHD4		CHD3	6	SRT4		SRT3	6	LedsRT2		SRT5	6
7	VDB5		VDB4	7	VDB5		VDB4	7	GND		LedsRT5	7
8	CHD6		CHD5	8	SRT6		SRT5	8				8
9	VDB7		VDB6	9	VDB7		VDB6	9	SRT6	BL2 LS04	VCC	9
10	GND		CHD7	10	GND		SRT7	10	LedsRT6		SRT9	10
11				11				11	SRT7		LedsRT9	11
12	SaiBD\	BD2 HC244	VCC	12	EscM\	BT2 HC244	VCC	12	LedsRT7		SRT10	12
13	CHD8		SaiBD\	13	SRT8		EscM\	13	SRT8		LedsRT10	13
14	VDB9		VDB8	14	VDB9		VDB8	14	LedsRT8		SRT11	14
15	CHD10		CHD9	15	SRT10		SRT9	15	GND		LedsRT11	15
16	VDB11		VDB10	16	VDB11		VDB10	16				16
17	CHD12		CHD11	17	SRT12		SRT11	17	SRT12	BL3 LS04	VCC	17
18	VDB13		VDB12	18	VDB13		VDB12	18	LedsRT12		SRT15	18
19	CHD14		CHD13	19	SRT14		SRT13	19	SRT13		LedsRT15	19
20	VDB15		VDB14	20	VDB15		VDB14	20	LedsRT13		VDB0	20
21	GND		CHD15	21	GND		SRT15	21	SRT14		LedsMem0	21
22	SaiRE\	RE1 74LS173	VCC	22	UltC\	ME1 LS257	VCC	22	LedsRT14		VDB1	22
23	GND(N)		GND(CLR)	23	VDB0		GND(G\)	23	GND		LedsMem1	23
24	VEB0		SM0	24	SS0		VDB3	24				24
25	VEB1		SM1	25	SM0		SS3	25	VDB2	BL4 LS04	VCC	25
26	VEB2		SM2	26	VDB1		SM3	26	LedsMem2		VDB5	26
27	VEB3		SM3	27	SS1		VDB2	27	VDB3		LedsMem5	27
28	Rlg2		GND(G2\)	28	SM1		SS2	28	LedsMem3		VDB6	28
29	GND		GND(G1\)	29	GND		SM2	29	VDB4		LedsMem6	29
30	SaiRE\	RE2 74LS173	VCC	30	UltC\	ME2 LS257	VCC	30	LedsMem4		VDB7	30
31	GND(N)		GND(CLR)	31	VDB4		GND(G\)	31	GND		LedsMem7	31
32	VEB4		SM4	32	SS4		VDB7	32				32
33	VEB5		SM5	33	SM4		SS7	33	VDB8	BL5 LS04	VCC	33
34	VEB6		SM6	34	VDB5		SM7	34	LedsMem8		VDB11	34
35	VEB7		SM7	35	SS5		VDB6	35	VDB9		LedsMem11	35
36	Rlg2		GND(G2\)	36	SM5		SS6	36	LedsMem9		VDB12	36
37	GND		GND(G1\)	37	GND		SM6	37	VDB10		LedsMem12	37
38	SaiRE\	RE3 74LS173	VCC	38	UltC\	ME3 LS257	VCC	38	LedsMem10		VDB13	38
39	GND(N)		GND(CLR)	39	VDB8		GND(G\)	39	GND		LedsMem13	39
40	VEB8		SM8	40	SS8		VDB11	40				40
41	VEB9		SM9	41	SM8		SS11	41	VDB14	BL6 LS04	VCC	41
42	VEB10		SM10	42	VDB9		SM11	42	LedsMem14		SS1	42
43	VEB11		SM11	43	SS9		VDB10	43	VDB15		LedsEnd1	43
44	Rlg2		GND(G2\)	44	SM9		SS10	44	LedsMem15		SS2	44
45	GND		GND(G1\)	45	GND		SM10	45	SS0		LedsEnd2	45
46	SaiRE\	RE4 74LS173	VCC	46	UltC\	ME4 LS257	VCC	46	LedsEnd0		SS3	46
47	GND(N)		GND(CLR)	47	VDB12		GND(G\)	47	GND		LedsEnd3	47
48	VEB12		SM12	48	SS12		GND(VDB15)	48				48
49	VEB13		SM13	49	SM12		GND(SS15)	49	SS4	BL7 LS04	VCC	49
50	N.C.		GND(SM15)	50	VDB13		N.C.	50	LedsEnd4		SS7	50
51	N.C.		GND(SM15)	51	SS13		GND(VDB14)	51	SS5		LedsEnd7	51
52	Rlg2		GND(G2\)	52	SM13		GND(SS14)	52	LedsEnd5		SS8	52
53	GND		GND(G1\)	53	GND		N.C.	53	SS6		LedsEnd8	53
54	SaiBE\	BE1 HC244	VCC	54	SaiBE\	BE2 HC244	VCC	54	LedsEnd6		SS9	54
55	CHE0		SaiBE\	55	CHE8		SaiBE\	55	GND		LedsEnd9	55
56	VEB1		VEB0	56	VEB9		VEB8	56				56
57	CHE2		CHE1	57	CHE10		CHE9	57	SS10	BL8 LS04	VCC	57
58	VEB3		VEB2	58	VEB11		VEB10	58	LedsEnd10		SS13	58
59	CHE4		CHE3	59	CHE12		CHE11	59	SS11		LedsEnd13	59
60	VEB5		VEB4	60	VEB13		VEB12	60	LedsEnd11		RC\	60
61	CHE6		CHE5	61	GND(A14)		CHE13	61	SS12		LedsPede	61
62	VEB7		VEB6	62	N.C.		N.C.	62	LedsEnd12		Para	62
63	GND		CHE7	63	GND		GND(A15)	63	GND		LedsPara	63
64				64				64				64

Figura 13 – Placa 3

Figura 14 – Placa 4

1	NOR2a	NAND3 74LS10	VCC	1	A13	NAND8 74LS30	VCC	1	GND(A14)	RAM1 61C256	VCC	1
2	NOR2b		EscMem\	2	A12		N.C.	2	A12		EscMem\	2
3	A5		LeRom\	3	A11		A6	3	A7		A13	3
4	A4		VCC(up)	4	A10		A7	4	A6		A8	4
5	VCC(up)		MenosSigT	5	A9		N.C.	5	A5		A9	5
6	E4eE5\		E0B	6	A8		N.C.	6	A4		A11	6
7	GND		NovoMaisSigT	7	GND		Ender<3FC0h	7	A3		HabSelRAM\	7
8				8				8	A2		A10	8
9	NOR2a	AND2 74LS08	VCC	9	NOR2a	NOR2 74LS02	VCC	9	A1		SelRam\	9
10	NOR2b		N.C.	10	A10		NOR2b	10	A0		D7	10
11	SelRam\		N.C.	11	A11		A12	11	D0		D6	11
12	VCC(up)		N.C.	12	N.C.		A13	12	D1		D5	12
13	A0		VCC(up)	13	N.C.		HabSaiRAM\	13	D2		D4	13
14	E0B		A1	14	N.C.		E4eE5\	14	GND		D3	14
15	GND		E1B	15	GND		Ender<3FC0h	15				15
16				16				16	GND(A14)	RAM2 61C256	VCC	16
17				17	A2	Decodificador 74LS138	VCC	17	A12		EscMem\	17
18				18	A3		Esc0a3\	18	A7		A13	18
19				19	EscMem\		Esc4a7\	19	A6		A8	19
20				20	GND(E1\)		Esc8aB\	20	A5		A9	20
21				21	GND(E2\)		EscCaF\	21	A4		A11	21
22				22	HabSaiRAM\		Le0a3\	22	A3		HabSelRAM\	22
23				23	LeCaF\		Le4a7\	23	A2		A10	23
24				24	GND		Le8aB\	24	A1		SelRam\	24
25				25				25	A0		D15	25
26				26				26	D8		D14	26
27				27				27	D9		D13	27
28				28				28	D10		D12	28
29				29				29	GND		D11	29
30				30				30				30
31				31				31	N.C.	ROM1 SST39SF020	VCC	31
32				32				32	GND(A16)		VCC(Prog)	32
33				33				33	GND(A15)		GND(A17)	33
34				34				34	GND(A12)		GND(A14)	34
35				35				35	A7		GND(A13)	35
36				36				36	A6		A8	36
37				37				37	A5		A9	37
38				38				38	A4		GND(A11)	38
39				39				39	A3		LeRom\	39
40				40				40	A2		GND(A10)	40
41				41				41	A1		GND(CS\)	41
42				42				42	A0		D7	42
43				43				43	D0		D6	43
44				44				44	D1		D5	44
45				45				45	D2		D4	45
46				46				46	GND		D3	46
47				47				47				47
48				48				48	N.C.	ROM2 SST39SF020	VCC	48
49				49				49	GND(A16)		VCC(Prog)	49
50				50				50	GND(A15)		GND(A17)	50
51				51				51	GND(A12)		GND(A14)	51
52				52				52	A7		GND(A13)	52
53				53				53	A6		A8	53
54				54				54	A5		A9	54
55				55				55	A4		GND(A11)	55
56				56				56	A3		LeRom\	56
57				57				57	A2		GND(A10)	57
58				58				58	A1		GND(CS\)	58
59				59				59	A0		D15	59
60				60				60	D8		D14	60
61				61				61	D9		D13	61
62				62				62	D10		D12	62
63				63				63	GND		D11	63
64				64				64				64

5 ESCOLHA DO HARDWARE

De maneira geral a escolha dos CIs foi limitada pelos componentes disponíveis no mercado.

Durante todo o desenvolvimento foi preferido trabalhar com CIs da família **74HC** pois estes são implementados com lógica CMOS o que de maneira geral implica em um consumo menor de energia e uma velocidade maior. Além disso, por ser uma implementação mais recente esses CIs são em geral mais fáceis de ser encontrados.

Porém, nem todos os CIs necessários estão disponíveis na família **74HC**, o que tornou necessário a utilização de CIs **74LS** em conjunto com os demais. Um problema que surge com essa necessidade é que os níveis de tensão dos CIs **74LS** respeitam os níveis TTL, mas os CIs **74HC** respeitam os níveis CMOS. Uma comparação entre esses níveis encontra-se na Tabela 5.

Tabela 5 – Níveis de tensão

	Entrada nível lógico 0	**Entrada nível lógico** 1	**Saída nível lógico** 0	**Saída nível lógico** 1
TTL	$0V$ até $0.8V$	$2.0V$ até $5.0V$	$0V$ até $0.5V$	$2.7V$ até $5.0V$
CMOS	$0V$ até $1.5V$	$3.5V$ até $5.0V$	$0V$ até $0.05V$	$4.95V$ até $5.0V$

Referência: (MEDIA, (Acessado em jan. 2019))

Como podemos notar ao analisar a Tabela 5, embora a saída de um circuito integrado de nível CMOS seja compatível com a entrada de um de nível TTL, o contrário não é verdade. Isso leva a duas possíveis soluções, ou utilizar resistores de *pull-up* sempre que houver uma saída de um CI TTL para um CMOS, ou utilizar apenas circuitos TTL ligados a saídas de outros circuitos TTL.

A solução com *pull-ups* certamente funcionaria, porém, os CIs responsáveis pelo somador e os responsáveis pelos registradores não estavam disponíveis na família **74HC** no mercado. Como o somador e os registradores possuem como saída um barramento de 16 *bits*, além do *bit* de *carry* do somador, o uso de no mínimo 16 resistores de *pull-up* em cada registrador e no somador poderia resultar de diversos problemas com mau contato. Com base nisso, optou-se pela segunda opção, utilizar somente CIs **74LS** nas saídas do somador e dos registradores.

Na Tabela 6 podemos ver quais CIs foram escolhidos para cada componente, assim como certas observações.

Tabela 6 – CIs dos componentes

Componente	CI	Placa	Total	Observação
Buffers	**2x74HC244**	**1 e 2**	**8**	*Buffers* de 8 *bits* com saída *tri-state*
Multiplexadores	**4x74LS257**	**2**	**4**	Mutiplexador de 4 *bits* com duas entradas
Registradores	**4x74LS173**	**1 e 2**	**16**	Registradores de 4 *bits*
Somador	**4x74LS283**	**1**	**4**	Somadores completos de 4 *bits*
Complementador	**4x74LS02**	**1**	**4**	4 NOR de duas entradas
RC, Est_1 e RI	**74HC74**	**3**	**2**	2 *Flip-flops* tipo D com saídas Q e $\overline{Q}$ e entradas de *set* e *reset*
Sincronizador dos botões e de *Parte*	**74HC175**	**3**	**1**	4 *Flip-flops* tipo D com saídas Q e $\overline{Q}$
Partiu e *Para*	**74HC112**	**3**	**1**	Dois *flip-flops* JK
Divisor de *clock*	**74HC4040**	**3**	**1**	Contador de 12 *bits*
Buffer do *clock*	**74HC14**	**3**	**1**	Inversor com entrada *Schmitt trigger*
Buffer dos LEDs	**74LS04**	**2**	**8**	Inversor
AND 2 entradas	**74LS08**	**4**	**1**	Decodificação de Endereços
AND 4 entradas	**74HC21**	**3**	**1**	Parte da Unidade de Controle
NAND 2 entradas	**74HC00**	**3**	**2**	Parte da Unidade de Controle
NAND 3 entradas	**74LS10**	**4**	**0**	Decodificação de endereços
NAND 4 entradas	**74HC20**	**3**	**2**	Parte da Unidade de Controle
NAND 8 entradas	**74HC30**	**4**	**1**	Decodificação de endereços
NOR 2 entradas	**74HC02**	**3**	**2**	Parte da Unidade de Controle
NOR 2 entradas	**74LS02**	**4**	**1**	Decodificação de endereços
NOR 3 entradas	**74HC27**	**3**	**2**	Parte da Unidade de Controle
Decodificador	**74LS138**	**4**	**1**	Decodificação de endereços
Memória PROM	**SST39SEF020**	**4**	**2**	Flash de 256K *bytes*
Memória RAM	**61c256**	**4**	**2**	SRAM de 32K *bytes*

6 MONTAGEM DO COMPUTADOR

A escolha por utilizar *protoboards* para a montagem de um computador tem pontos positivos e negativos. Pelo lado positivo, temos uma maior velocidade de modificação, caso seja necessário mover algumas conexões (ou até mesmo mover completamente um CI). Temos também uma base firme para trabalhar, e a ausência de solda torna mais fácil substituir CIs com defeito.

Por outro lado, *protoboards* não garantem uma conexão livre de maus contatos, o que pode se tornar um problema com o grande número de conexões que o CES exige. Além disso, *protoboards* podem não ser ideais para sinais de alta velocidade, uma vez que a capacitância entre duas trilhas adjacentes não é desprezível.

6.1 CONEXÕES INTERNAS

Para realizar as conexões internas (entre CIs de uma mesma *protoboard*), foram utilizados fios retirados de cabos Ethernet. Estes fios foram escolhidos por apresentarem um diâmetro ideal para as conexões das *protoboards*, além de serem amplamente disponíveis e facilmente cortados no comprimento certo para realizar cada conexão. Todavia, estes fios apresentam alguns problemas. Os fios são demasiadamente rígidos, o que torna sua modelagem mais difícil, deixando-os com maior propensão a partir. Remover o isolamento dos fios não é uma tarefa muito fácil, pois o isolamento não foi projetado para ser removido e por isso não é constituído de materiais que facilitem essa tarefa. Além disso, desencapadores de fios em geral não funcionam. Outro problema inerente a estes fios é a sua tendência a oxidar, e como óxido não é um bom condutor de eletricidade, os protótipos construídos empregando estes fios tendem a ter uma vida útil limitada. Por fim, há um limite muito pequeno para as cores dos fios, o que não permite uma correta distinção de conexões diferentes.

Uma alternativa melhor para as conexões seria a utilização de fios de Kynar com um diâmetro pouco maior do que o dos fios do cabo Ethernet. Os fios de Kynar são comumente utilizados na técnica de *Wire-wrapping*. Eles são mais flexíveis e estão disponíveis em diâmetros padrões, o que torna possível remover o isolamento utilizando um desencapador de fios. Fios de Kynar não foram utilizados pois possuem um custo muito elevado e um rolo de fios possui uma única cor, o que diminuiria ainda mais a distinção entre conexões diferentes.

6.2 CONEXÕES EXTERNAS

As conexões externas (que ligam uma *protoboard* a outra) mostraram-se um desafio. O primeiro ponto relevante é que essas conexões são em geral vias do barramento, o que significa que aproximadamente 16 fios estão sempre juntos. Utilizar fios de cabos Ethernet para realizar essas conexões poderia causar problemas, pois a rigidez destes fios tornaria essas conexões pouco maleáveis, aumentando a probabilidade de que conexões se romperem e de fios se soltarem da *protoboard*. Além disso, os barramentos ocupariam um volume considerável, o que cria também um possível problema estrutural.

Para realizar as conexões externas, foram utilizados cabos IDE retirados de computadores antigos. Um exemplo de cabo IDE encontra-se na Figura 15.

Figura 15 – Cabo IDE

Fonte: (COMPUTERHOPE, 2017 (Acessado em 10 mar. 2019))

Um problema com esse cabo é que os fios não possuem rigidez, o que torna impossível conectá-los a *protoboards*. Para resolver esse empecilho os fios de cada cabo foram soldados a pinos (comumente utilizados em placas de circuito impresso), como mostra a Figura 16.

6.3 FONTE DE ALIMENTAÇÃO

Todos os CIs utilizados possuem alimentação de 5 *volts*. Essa também é a voltagem padrão de uma porta USB. Portanto, a alimentação dos CIs se dá através de um cabo USB que é, em geral, conectado a um carregador de *smartphone*.

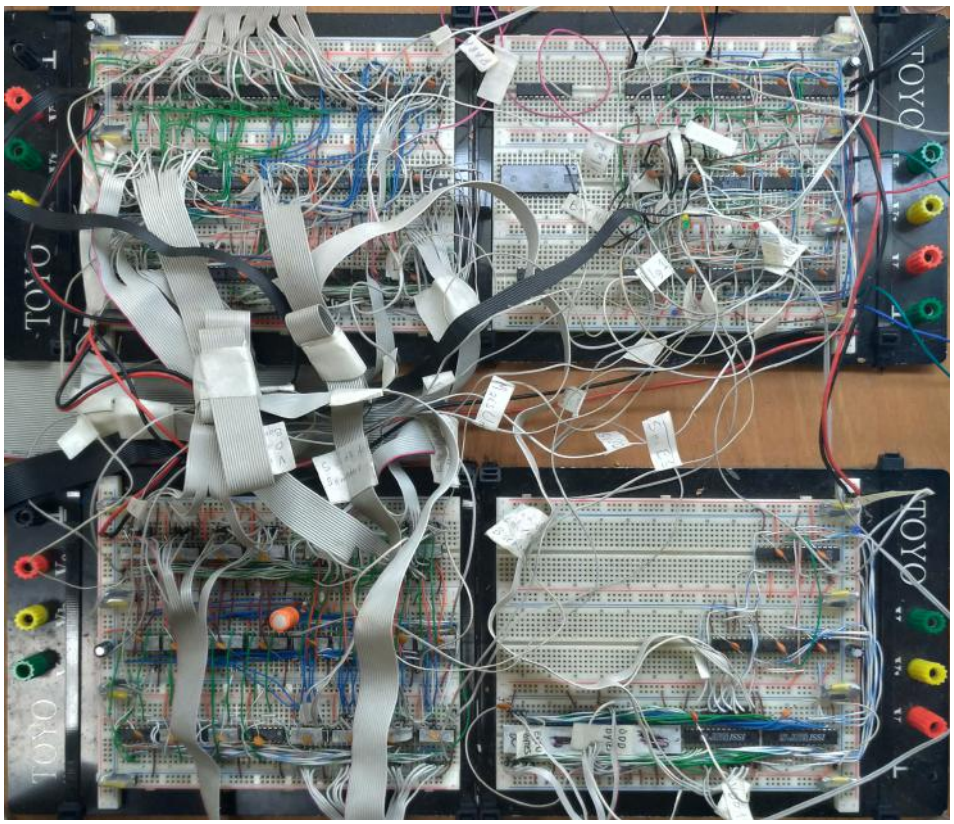
Além disso, capacitores de desacoplamento foram utilizados para garantir que os componentes não sofram com variações de tensão. Um capacitor de $100nF$ foi instalado diretamente entre os pinos V_{cc} e GND de cada CI.

Figura 16 – Pino e cabo IDE

6.4 RESULTADO FINAL

Na Figura 17 encontra-se uma imagem da montagem final do computador. O projeto consome em média $700mA$ quando o processador está parado e $610mA$ quando está executando. O *clock* máximo suportado é de $2MHz$.

Figura 17 – Montagem final

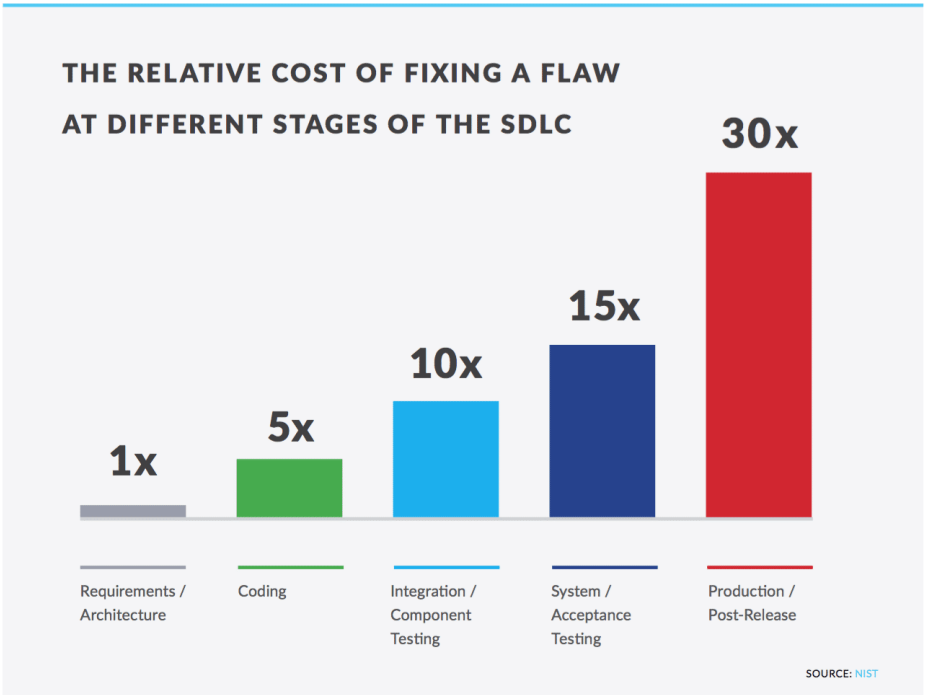


7 TESTES REALIZADOS

A montagem do CES em quatro *protobards* é um projeto consideravelmente grande. Cada conexão entre dois pinos está sujeita a erro humano. Além disso, outros problemas podem acontecer, como um mau contato nas vias de uma *protoboard* ou um fio partido. Fazer a montagem de todo o projeto para somente depois lidar com os possíveis problemas tornaria a solução destes mais complicada, visto que o problema estaria inserido em um sistema mais complexo, e encontrar a causa raiz do problema demandaria mais tempo.

Uma situação similar é encontrada em Desenvolvimento de *Software*, em que localizar e corrigir um *bug* em um sistema complexo leva mais tempo (e portanto é mais custoso) do que encontrar este problema durante o desenvolvimento dos componentes que formam esse sistema. A Figura 18 mostra o custo relativo para corrigir um problema nas diversas fases do ciclo de desenvolvimento de um *software*.

Figura 18 – Custo relativo para corrigir uma falha



Fonte: National Institute of Science and Technology (NIST)

Visando minimizar o custo de correção de falhas, aconselha-se cada vez mais a criação de testes que possam encontrar uma falha logo no início. Duas categorias importantes são:

- Teste unitário: são testes de baixo nível. Eles consistem em testar métodos e funções individualmente;

- Teste de integração: certificam-se de que módulos ou serviços usados pela aplicação funcionam bem juntos.

Fonte: Atlassian, (PITTET, (Acessado em 6 fev. 2019)).

De forma similar aos componentes de um *software*, os componentes do CES foram testados buscando-se encontrar falhas nas fases iniciais do desenvolvimento do projeto.

Para a realização dos testes, foi utilizado um Arduino Mega. Essa placa foi escolhida por dois motivos principais. Primeiro, as placas Arduino possuem uma vasta biblioteca que tornam a sua programação mais fácil do que a programação de um micro-controlador simplesmente. Essa característica permite uma iteração rápida do ciclo que envolve avaliar os requisitos de um teste, programar este teste e rodá-lo no circuito em questão. Segundo, o Arduino Mega utiliza como base o micro-controlador ATmega2560 e expõe 70 pinos de entrada/saída do micro-controlador. Esse grande número de pinos é essencial, pois o CES possui barramentos de 16 *bits*.

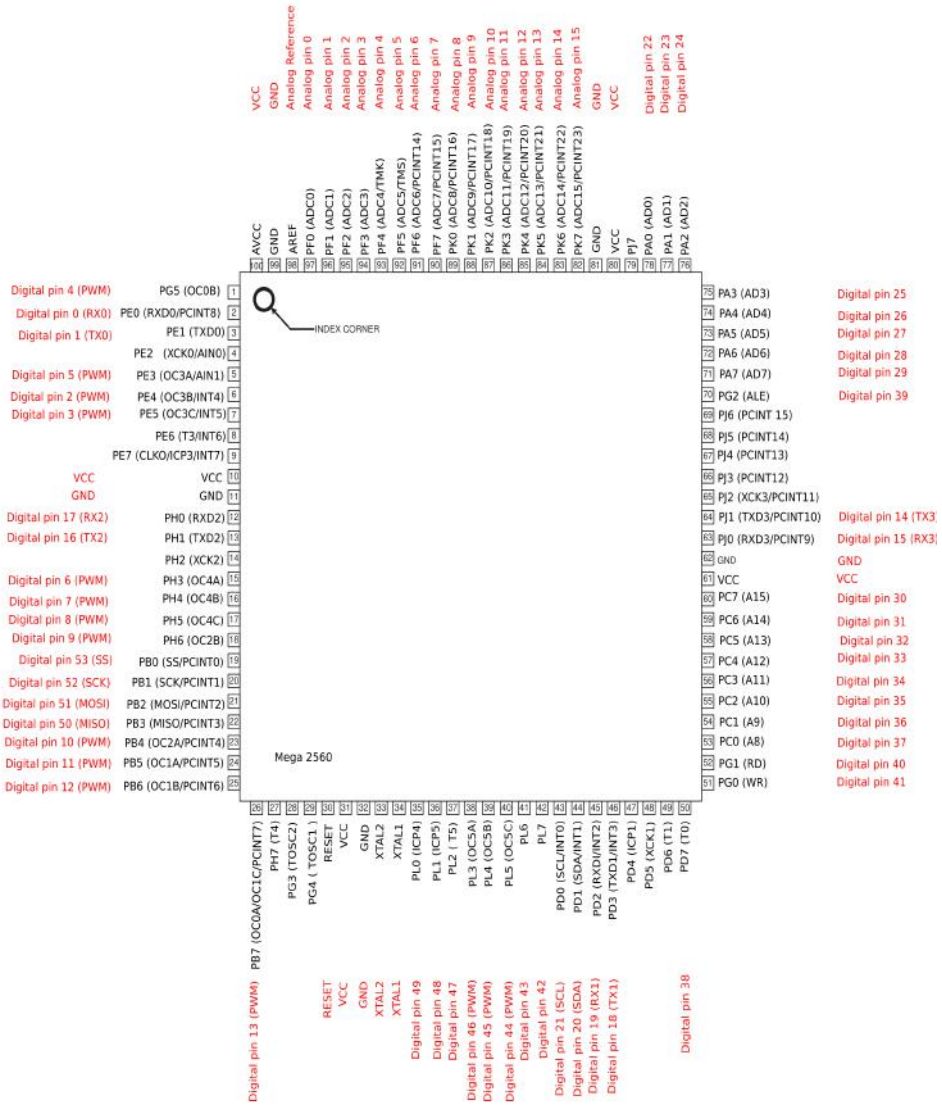
A escrita e leitura direta das portas do micro-controlador foram utilizadas nos testes para evitar dependência nas rotinas $digitalWrite$ e $digitalRead$ da biblioteca do Arduino, que são conhecidamente lentas, e para evitar manipulação direta de *bits* (e em vez disso utilizar manipulação de *bytes*). As informações sobre as portas do micro-controlador, assim como outros aspectos deste, foram encontradas no datasheet fornecido pela Atmel e que pode ser encontrado em (ATMEL, 2014 (Acessado em mai. 2018)).

Para relacionar as portas do ATmega2560 com os pinos no Arduino Mega, foi utilizada a Figura 19.

Os testes foram capazes de detectar diversos erros como maus contatos, fios partidos, conexões erradas e circuitos integrados com defeito. Realizar esses testes antes de ter o projeto completo foi muito importante, pois detectar alguns desses erros em um sistema complexo demandaria muito tempo.

Os Códigos A.1, A.2, A.3, A.4, A.5 e A.6 correspondem a cada teste realizado e podem ser encontrados no Apêndice A. A seguir será descrito quais foram os critérios para cada teste realizado.

Figura 19 – Pinos do Arduino Mega



Fonte: (ARDUINO, 2017 (Acessado em mai. 2018))

7.1 TESTE DA PLACA 1

Este teste encontra-se no Código A.1.

Em todos os testes realizados, buscou-se começar o teste dependendo do menor número possível de componentes, pois dessa forma seria fácil encontrar problemas que estivessem ocorrendo diretamente na saída da placa. Dessa forma, o primeiro teste a ser realizado na placa 1 é inserir 0 nas Chaves de Dados, passando esses dados diretamente pelo somador (somando com o valor 0) e lendo o valor na saída do somador. Em seguida são testados todos os demais valores possíveis nas Chaves de Dados, ou seja, de 0 até $2^{16} - 1$, também somando com o valor 0 e lendo o valor na saída do somador. Este é um teste parcial do somador e dos buffers de entrada das Chaves de Dados.

Em seguida, este teste é repetido, mas dessa vez **Mais1** é ativado. Dessa forma, espera-se encontrar na saída do somador o valor inserido acrescido de 1. Este é mais um teste parcial dos somadores.

Depois é realizado um teste similar aos dois primeiros, porém, agora o valor não é inserido nas Chaves de Dados, em vez disso o valor é inserido no barramento de dados, armazenado no registrador RD e direcionado para o somador para ser somado com o valor 0. Este é um teste completo de registrador RD. Note que o primeiro teste garante que se esse teste falhar o problema deve estar em RD.

Logo após, o registrador RP é testado utilizando uma lógica similar à que ocorre quando o CES está em funcionamento. Primeiro obtém-se o valor 0 na saída do somador utilizando o mesmo método do primeiro teste. Esse valor é então escrito no registrador RP. A entrada pelas Chaves de Dados é desabilitada, e a nova entrada esquerda do somador passa a ser o registrador RP. No lado direito do somador está o valor 0, e **Mais1** é ativado. Dessa forma, espera-se encontrar o valor 1 na saída do somador. Essa valor é então escrito em RP, e agora espera-se pelo valor 2 na saída do somador. Essa lógica é repetida para todos os valores possíveis, ou seja, até $2^{14} - 1$, dado que RP é um registrador de 14 *bits*. Este é um teste completo do registrador RP.

Um teste similar ao anterior é realizado em RT. Porém, o valor de RT só pode ser acessado após passar pelo complementador (ou seja, o valor de RT invertido). Para isso, escreve-se no barramento de dados o valor a ser testado, começando por 0. Este valor é armazenado em RD e depois é levado para a saída do somador. O valor é então lido para o registrador RT ao mesmo tempo que o valor 0 é lido para o registrador RD. É realizada uma soma com o valor 0 na entrada esquerda do somador, o inverso do valor sendo testado (valor em RT) na entrada direita e **Mais1** é ativado. O valor esperado na saída do somador é o resultado de $0 - RT$. Este teste é repetido para todos os valores possíveis, ou seja, de 0 até $2^{16} - 1$. Com isso testa-se RT, o complementador e parcialmente o somador. Esse não é um teste ideal, pois este depende de diversos componentes, o que dificulta a identificação da origem de uma possível falha. Porém, dado que o valor de RT

não pode ser acessado diretamente, esse é o melhor possível nessa situação.

Por fim, testou-se completamente o somador. O teste ideal do somador consiste em executar todas as somas possíveis e verificar o resultado, porém, esse não é um teste viável dado que o número total de iterações seria 2^{32}. Em vez disso testou-se individualmente os quatro somadores de 4 *bits* que formam o somador da seguinte forma. O primeiro somador é testado somando todos os números de 0 a 16 do lado esquerdo com todos os números de 0 a 16 do lado direito. O mesmo teste é repetido, mas dessa vez fazendo um *shift* de 4 *bits* para a esquerda em ambos os valores, assim testando o segundo dos quatro somadores. Essa lógica é repetida mais duas vezes para testar completamente todos os somadores.

7.2 TESTE DA PLACA 2

A segunda placa possui o maior número de conexões externas (conexões para outras placas). Por esse motivo, nem mesmo utilizando os 70 pinos disponíveis no Arduino Mega foi possível testar a placa por completo em um único teste. Foram realizados 3 testes diferentes na placa 2, cada um testando um conjunto lógico diferente da placa. Primeiro testou-se Buffer das Chaves de Dados (BD) e o Buffer do Registrador de Trabalho (BT). O segundo teste envolveu o Multiplexador de Endereços (ME) e o Registrador de Endereços (RE). Por fim, testou-se o Buffer das Chaves de Endereços. Vale notar que o Buffer dos LEDs (BL) não foi testado, pois o funcionamento deste não é essencial para o correto funcionamento do computador.

7.2.1 Teste de BD e BT

Este teste encontra-se no Código A.2.

Ambos os *buffers* tem como saída as Vias de Dados do Barramento (VDB), portanto esse teste sempre lê VDB para obter o resultado.

Buffers possuem uma função lógica relativamente simples, e por isso os seus testes também são simples. Para testar o *Buffer* de Dados, são escritos nas Chaves de Dados (entrada de BD) todos os valores de 0 a $2^{16} - 1$ e a resposta é lida em VDB.

Para o teste de BT a mesma lógica é repetida. Todos os valores de 0 a $2^{16} - 1$ são escritos na entrada de BT, e o resultado é lido em VDB.

7.2.2 Teste de ME e RE

Este teste encontra-se no Código A.3.

Esses dois componentes precisam ser testados ao mesmo tempo, pois a saída de ME não pode ser acessada diretamente, em vez disso, a saída de ME é utilizada como entrada de RE. O conjunto testado é formado por duas entradas (a Via de Dados do

Barramento (VDB) e a Saída do Somador (SS)) e uma saída (a Via de Endereços do Barramento(VEB)).

O teste é realizado da seguinte forma. Todos os valores de 0 a $2^{16} - 1$ são escritos na entrada A do multiplexador, ou seja, em VDB. Ao mesmo tempo o complemento desse valor (sendo o valor em uma dada iteração igual a i o complemento é calculado obtendo com o resultado de $0xFFFF - i$) é escrito na entrada B do multiplexador, ou seja, na Saída do Somador (SS). Esses valores são escolhidos para manter a maior distância de Hamming entre eles.

Em cada iteração, primeiro seleciona-se a entrada A do multiplexador, verificando se o valor correto é obtido em VEB. Em seguida, seleciona-se a entrada B do multiplexador e verifica-se o valor em VEB.

7.2.3 Teste de BE

Este teste encontra-se no Código A.4.

Como BE também é um *buffer*, este teste é similar ao teste em 7.2.1. Sua entrada são as Chaves de Endereço e sua saída é a Via de Endereço do Barramento (VEB). Todos os valores possíveis, isto é, de 0 a $2^{14} - 1$, são escritos nas Chaves de Endereços, e o resultado é lido em VEB.

7.3 TESTE DE INTEGRAÇÃO DAS PLACAS 1 E 2

É comum em uma arquitetura de Von Neumann a divisão do processador em Unidade de Controle, Unidade Lógica e Aritmética e Registradores. De forma geral, a UC é responsável por gerar sinais que controlam os Registradores e a ALU. A UC do CES está totalmente contida na placa 3, enquanto os Registradores e a ALU estão distribuídos nas placas 1 e 2. Dessa forma, faz sentido integrar essas duas placas e testar o funcionamento das duas em conjunto, simulando os sinais da UC.

Este teste encontra-se no Código A.5 e pode ser dividido em 4 partes.

7.3.1 Teste das chaves

Neste teste os *buffers* das Chaves de Dados e das Chaves de Endereço são testados. Além disso, a estabilidade do valor das chaves também é testada da seguinte forma: em cada iteração o valor do resultado é lido 255 vezes antes de avançar para a próxima iteração. Se pelo menos um desses 255 não corresponder ao valor esperado, o teste falha.

Para as Chaves de Endereço, o valor do resultado é lido nas Vias de Endereços do Barramento. Os valores testados variam de 0 a $2^{14} - 1$.

Para as Chaves de Dados o valor do resultado é lido em dois locais diferentes e em duas etapas. Primeiro, o resultado é lido diretamente das Vias de Dados do Barramento, utilizando como saída o *buffer* BD.

Em seguida, o resultado é lido também em VDB, mas dessa vez como uma forma de ler indiretamente o valor de RT. O valor testado é escrito nas Chaves de Dados, passa pelo *buffer* BS, é somado ao valor zero e é armazenado em RT. Finalmente, utiliza-se o *buffer* BT para emitir o valor de RT nas Vias de Dados do Barramento.

7.3.2 Teste de incremento de RP

Este teste é similar ao descrito na Seção 7.1. O valor 0 é carregado em RP. Em seguida, é executado um *loop* em que o valor de RP é somado ao valor 0, e com o sinal **Mais1** é ativado. O valor da soma é armazenado novamente em RP. Este teste itera sobre todos os valores possíveis de RP, isto é, de 0 a $2^{14} - 1$.

7.3.3 Teste de RD

Nesse teste o Registrador de Dados é testado com todos os valores possíveis, isto é, de 0 a $2^{16} - 1$. Cada valor é escrito nas Vias de Dados do Barramento e armazenado em RD. Em seguida, o valor é levado à entrada esquerda do somador. Na entrada direita está o valor 0. O sinal **Mais1** pode estar ativado ou desativado com probabilidade de 50%. O valor da soma é armazenado em RT e depois é escrito na Via de Dados do Barramento na qual é lido e comparado com o valor esperado.

7.3.4 Teste de RT

Este é um teste bem completo de todos os componentes das duas placas. Nele itera-se sobre todos os valores possíveis para o Registrador de Trabalho (RT), ou seja, de 0 a $2^{16} - 1$. Primeiro, o valor a ser testado é invertido (inversão *bit*-a-*bit*) e escrito nas Vias de Dados do Barramento. Esse valor passa por RD e é levado à entrada esquerda do somador. Na entrada direita do somador está o valor 0, e **Mais1** está desativado. O valor sai do somador sem alteração e é armazenado em RT. Agora o valor 0 é inserido em RD e levado à entrada esquerda do somador. Na entrada direita é inserido o valor de RT após passar pelo Complementador de Dados. O resultado (que deve ser o valor sendo testado, sem inversão) é novamente armazenado em RT, mas dessa vez utiliza-se o *buffer* BT para emitir esse valor na Via de Dados do Barramento, na qual ele é finalmente lido e comparado ao valor a ser testado.

7.4 TESTE DA PLACA 3

Este teste encontra-se no Código A.6.

Como mencionado na Seção 7.3, a placa 3 é composta primariamente pela Unidade de Controle. Pela Imagem 7 podemos notar que a lógica da UC é em sua maior parte combinacional, portanto o teste mais apropriado para esta placa é um teste de Tabela

Verdade, em que todas as possíveis entradas são exercitadas, e o valor de cada uma das saídas é verificado. A Tabela 4 mostra as expressões lógicas de todos os sinais gerados pela UC.

A UC possui 13 variáveis de entrada e 25 sinais de saída. As variáveis de entrada são: I_0, I_1, RC, $Para$, Est_1, $Parte$, $MudaP$, $MudaT$, $BotParte$, $Partiu$, $IntPara$, $BotEscMem$ e $\overline{EntRlg}$.

E os sinais de saída são: $Desvia$, $EscM$, $UltC$, $EscP$, $EscT$, $EscC$, $ZComp$, **Mais1**, $EntRlg$, $\overline{Rlg}$, $Rlg1$, $Rlg2$, $Rlg3$, $BotParteAtivo$, $\overline{BotParteAtivo}$, $VaiPartir$, $VaiParar$, $EscMem$, $SaiBD$, $SaiBE$, $SaiBS$, $SaiRE$, $SaiRP$, $SaiRI$ e $SaiRC$.

Alguns dos valores de entrada não possuem conexão externa e são, em vez disso, gerados por *flip-flops*. Nesses casos, fez-se com que os *flip-flops* emitissem os valores desejados da seguinte forma: para *flip-flops* do tipo D o valor desejado é inserido diretamente na sua entrada D, e um ciclo de *clock* é gerado para fazê-lo admitir o valor. *Flip-flops* do tipo JK foram momentaneamente conectados de forma a se comportarem como um *flip-flop* do tipo D, isto é, o valor desejado foi inserido na entrada J e inverso desse valor na entrada K.

Como a UC possui 13 variáveis de entrada, a sua Tabela Verdade possui 2^{13} linhas. Cada uma dessas linhas foi testada, e o valor das 25 variáveis foi lido e comparado com o esperado. O valor esperado foi calculado pelo micro-controlador utilizando as expressões lógicas da Tabela 4, pois dessa forma não foi necessário pré-computar e armazenar todas as 8096 linhas da tabela.

7.5 AUSÊNCIA DO TESTE DA PLACA 4

Os únicos componentes presentes na quarta placa e que são fundamentais para o funcionamento do computador são as memórias (RAM e ROM) e o Decodificador de Endereços. Não existe um bom método para testar as memórias, visto que cada pastilha de memória é um Circuito Integrado sem acesso ao seu funcionamento. Além disso, cada pastilha de memória pode ser simplesmente testada utilizando a função de teste de um gravador de CIs.

Quanto ao Decodificador de Endereços, embora possa ser criado um teste para avaliá-lo, o teste seria muito simples, e considerando o tempo necessário para escrever e preparar este teste com os possíveis benefícios deste, julgou-se desnecessário este teste.

8 FERRAMENTA DE DISPOSIÇÃO DOS CIRCUITOS INTEGRADOS

Para trabalhar com um número grande de circuitos integrados em 4 *protoboards*, viu-se necessário uma ferramenta para dispor os CIs nas placas, de forma a criar um diagrama que pudesse ser utilizado para posicionar os CIs nas placas e conectá-los.

De início um editor de texto foi utilizado, mas este se mostrou inadequado para a tarefa. Pequenas modificações, como mover um CI, significavam ter que refazer uma grande parte do documento, o que tornou impossível iterações rápidas do diagrama. Portanto, tornou-se necessário o uso de uma ferramenta especializada para tal tarefa.

Ferramentas para a criação de diagramas lógicos de circuitos e até mesmo de placas de circuito impresso estão disponíveis, mas uma ferramenta para a criação de diagramas de *protoboards* não foi encontrada. Por essa razão, uma ferramenta para a criação desse tipo de diagrama foi desenvolvida para o projeto. A ferramenta em questão foi nomeada *bread_placer* (pois em inglês *protoboards* são chamadas de *breadboards*) e o seu código pode ser encontrado em: (PIRES, 2018 (Acessado em fev. 2019)a). Os diagramas mostrados no capítulo 4 foram criados utilizando essa ferramenta.

Para o desenvolvimento da ferramenta, foi utilizada a biblioteca *SDL*, que provê abstrações para geração de imagens gráficas. Esta biblioteca é comumente utilizada para desenvolvimento de jogos e por isso está disponível em diversas plataformas. Com isso, a ferramenta *bread_placer* pode, em teoria, ser portada sem muito esforço para qualquer plataforma que disponibilize a biblioteca padrão da linguagem C e a biblioteca *SDL*. Para o desenvolvimento desse trabalho, a ferramenta foi utilizada somente em ambiente *Linux*.

A entrada da ferramenta se dá através de um arquivo de extensão *ics_list* que possui informação sobre todos os CIs que devem ser colocados na placa. Sempre que a ferramenta é fechada, um arquivo de extensão *icprj* com o mesmo nome do arquivo de entrada é gerado com informação sobre a disposição dos CIs na placa. Esse arquivo deve ser passado para a ferramenta nas execuções posteriores, pois o arquivo de extensão *ics_list* é lido automaticamente quando um arquivo *icprj* é utilizado. Mesmo após a primeira execução, mais CIs podem ser adicionados ao arquivo *ics_list*, e estes serão automaticamente incluídos no projeto como CIs fora da placa.

8.1 SINTAXE DO ARQUIVO *ICS_LIST*

Este arquivo deve conter um ou mais CIs. Componentes são descritos por grupos de linhas separados por uma linha em branco. Quando uma linha em branco ou o final do arquivo é encontrado, a ferramenta termina a descrição do CI atual. Um exemplo da sintaxe desse arquivo encontra-se no Código 8.1.

Código 8.1 – Exemplo *ics_list*

```
IC 16
Code 74LS173
Name RE1
Pins
*1 SaiRE\
*2 GND(N)
#3 VEB0
#4 VEB1
#5 VEB2
#6 VEB3
*7 Rlg2
*8 GND
*9 GND(G1\)
*10 GND(G2\)
*11 SM3
*12 SM2
*13 SM1
*14 SM0
*15 GND(CLR)
*16 VCC
```

A primeira linha de cada bloco começa com as letras *IC* seguidas do número de pinos que o CI possui. As duas próximas linhas contêm as palavras *Code* e *Name* e devem ser seguidas, respectivamente, do código do CI e do seu nome no diagrama. Logo em seguida há uma linha com apenas a palavra *Pins*, após ela os pinos do CI devem ser descritos. A ordem com que os pinos são descritos não é importante, mas todos os pinos devem ser descritos antes do final do bloco.

A descrição de um pino se dá da seguinte maneira:

- Primeiro um símbolo: # indica que esse pino se conecta a outra placa e $*$ indica que todas as conexões desse pino estão dentro da mesma placa. Essa informação não é verificada pela ferramenta no momento, porém, o símbolo # é utilizado para deixar a etiqueta do pino em negrito;
- Após o símbolo deve vir o número do pino dentro do CI. Essa numeração começa em 1;
- Por fim deve vir o nome, ou etiqueta, do pino. Existem valores especiais para esse campo: VCC(...) deixará a etiqueta vermelha, GND(...) deixará a etiqueta cinza e N.C. (ou seja, Não Conectado) deixará a etiqueta azul.

8.2 UTILIZANDO A FERRAMENTA

A ferramenta divide conceitualmente os CIs em dois grupos. Os que estão na placa e os que estão fora dela. Do lado superior direito da tela há um contador de quantos CIs estão fora da placa. Ao iniciar a ferramenta pela primeira vez, todos os CIs estarão fora da placa.

Além disso, para os CIs que estão na placa existe o conceito de selecionado ou não. Apenas um CI pode estar selecionado em dado momento, e somente um CI selecionado pode ser movido.

A ferramenta possui um "cursor" que é colorido de azul quando não há CI selecionado e de roxo quando há um CI selecionado.

Todos os comandos da ferramenta são executados com o teclado e são os seguintes.

- Q ou ESC: Fecha a ferramenta;
- Z: Entra ou sai do modo de zoom;
- W, A, S e D: Movem a janela quando em modo de zoom;
- *Setas direcionais*: Movem o cursor. Quando há um CI selecionado, este é movido junto do cursor;
- R: Se houver um CI selecionado, este é rotacionado 180^{o};
- $BACKSPACE$ ou $DELETE$: Se um CI estiver selecionado, leva este CI para fora da placa;
- I: Insere na placa um CI de fora dela. Uma janela é aberta para escolher o CI desejado;
- $SPACEBAR$ ou $ENTER$: Seleciona o CI que estiver sob o cursor;

Dois CIs não podem ocupar o mesmo espaço. Por isso, sempre que um CI estiver sendo movido, deve haver espaço para movê-lo. Da mesma forma, para inserir na placa um CI de fora da placa também deve haver espaço suficiente para o CI ser inserido.

8.3 FORMATOS DE SAÍDA

A ferramenta tem como saída uma imagem com o diagrama idêntico ao visualizado durante a sua utilização no formato *bmp*. Além disso, em ambiente *Linux*, foi testado gerar uma imagem vetorial em formato *svg* utilizando *scripts* em *bash* e em *python* e os comandos *convert* (parte do *ImageMagick*) e *potrace*.

9 UM MONTADOR EXTREMAMENTE SIMPLES

9.1 SOBRE O MONTADOR

O montador (do inglês *assembler*) é a ferramenta responsável por transformar código em linguagem de montagem (*assembly*) em código de máquina (codificação binária) que o computador pode de fato executar. Essa transformação é em geral um-para-um, ou seja, cada linha em um programa em linguagem de montagem corresponde a uma única instrução, e, de maneira geral, a uma quantidade fixa de *bytes*, no programa em linguagem de máquina. Podemos dizer que a linguagem de montagem representa o mais próximo possível a linguagem de máquina, mas garantindo que o programa possa ser facilmente entendido pelo programador.

Ser capaz de transformar linguagem de montagem em linguagem de máquina de forma automática é algo desejável para qualquer computador, incluindo o CES. Essa transformação pode ser feita de maneira manual, dada a simples correlação entre as duas linguagens, mas a transformação manual de um programa sem que ocorram erros, além de demandar muito tempo, é bastante inconveniente.

Existem montadores com muitas funcionalidades além de simplesmente transformar linguagem de montagem em linguagem de máquina, um exemplo é o montador da arquitetura *x86*. Uma possível abordagem para gerar código de máquina para o CES é a utilização funcionalidades de montadores mais complexos de outra arquitetura. Porém, essa solução alternativa é um pouco complicada e sujeita a erros. Por esse motivo um montador capaz de transformar a linguagem de montagem do CES em código de máquina foi desenvolvido e nomeado Montador Extremamente Simples (MES). O código do projeto pode ser encontrado em: (PIRES, 2018 (Acessado em fev. 2019)b).

O objetivo do MES é ser um montador simples para o CES, porém, é necessário ter o mínimo de funcionalidades que se espera em um montador. O Código 9.1 é um exemplo típico em linguagem de montagem que ajuda a entender quais são essas funcionalidades básicas.

O MES utiliza os seguintes mnemônicos para as instruções do CES:

- LM: Lê valor da memória;
- EM: Escreve valor na memória;
- SB: Subtrai;
- DNP: Desvia se não pede emprestado.

Contrário ao comportamento de alguns montadores, o MES é sensível à caixa das letras, ou seja, ele diferencia letras maiúsculas das minúsculas.

Código 9.1 – Exemplo de código de montagem

```
; Meu programa

#include basic.asm

NGT     equ     SB Zero ; T = 0 - T, nao pede emprestado se T for 0
INV     equ     SB HFFFF; T = T\, nao pede emprestado;

        ORG 0
Zero:   DW 0
Um:     DW 1
Dois:   DW 2
Tres:   DW 3
Sete:   DW 7
Quinze: DW 15
H8000:  DW 8000h
HFFFF:  DW 0FFFFh               ; menos 1

IniRam  equ 400h


Halt    MACRO
        LM Zero
        NGT
        DNP $
        ENDM

        ORG 15

        Var1 equ (IniRam + 1)
        LM Quinze
Loop:   EM Var1
        LM Um
        SB Var1
        DNP Loop
        Halt
```

Examinando o Código 9.1, podemos notar algumas funcionalidades esperadas de um montador que não estão diretamente relacionadas com transformação de código em linguagem de montagem em código de máquina. Algumas delas são:

- *equ*: Comando usado para criar substituições por um determinado valor. Uma substituição é declarada inserindo o nome a ser substituído, seguindo da palavra reservada *equ*, seguida do valor da substituição. Durante o processamento do código, qualquer ocorrência dessas palavras será substituída pelo seu respectivo valor. Vale notar que esta substituição é textual e ocorre em todo o arquivo (assim como $\#define$ na linguagem C, por exemplo), ou seja, não existe escopo para essas substituições;

- *Macros*: Macros são trechos de código que não fazem parte do programa por si só. Em vez disso, sempre que o nome da macro aparece no código original esse é substituído durante o processamento pelo trecho de código correspondente à macro. Macros são iniciadas com o nome da macro seguido pela palavra reservada $MACRO$. Ao lado direito pode vir a lista de argumentos. Durante a invocação de uma macro, uma lista de parâmetros com o mesmo comprimento da lista de argumentos deve ser passada. Os argumentos serão substituídos em ordem pelos seus respectivos valores (parâmetros). Um exemplo de uma macro que recebe parâmetros, assim como a sua invocação, pode ser encontrado no Código 9.3.

  Uma macro pode possuir rótulos locais e globais, como descrito no item *Labels* abaixo.

  Uma macro é terminada pela palavra reservada $ENDM$;

- DW: Alocação de constantes (ou variáveis). Instrui o montador a utilizar o endereço atual de memória, guardando nesta posição o valor que está à direita da palavra reservada DW. No código de exemplo, as oito linhas no início alocam constantes nos oito primeiros endereços do programa;

- *Rótulos*: Rótulos (*labels*) são palavras utilizadas para representar (e abstrair) o endereço ou de uma instrução ou de um dado alocado com DW. A declaração de um rótulo é realizada iniciando uma determinada linha com o nome desejado para o rótulo e inserindo dois-pontos (:) logo em seguida. O nome dos rótulos pode ser utilizado em qualquer lugar em que o valor de um endereço pode ser usado, ou seja, como operando para uma instrução ou como parâmetro para uma macro. Vale notar que embora as pseudo-instruções *equ* e $MACRO$ possuam um nome no início da linha, esses nomes não são rótulos e por isso não utilizam dois-pontos.

  Rótulos declarados dentro de uma macro são globais, isto é, pertencem ao escopo do programa inteiro. Na maioria dos casos deseja-se um escopo local para rótulos

declarados em uma macro, e por isso existe a palavra reservada $LOCAL$, que deve ser inserida na primeira linha após a declaração da macro. Seguindo esta palavra reservada, deve vir uma lista com o nome dos rótulos que serão declarados na macro e que terão escopo local. Um exemplo disso pode ser visto no Código 9.2;

- \$: O caracter \$ é um rótulo especial que é sempre substituído pelo endereço da posição atual de memória. No código de exemplo ele é utilizado para fazer com que uma instrução desvie para si mesma;

- *Valores em hexadecimal*: A habilidade de expressar um valor em hexadecimal utilizando a letra *h* como sufixo do número. Por exemplo, *8000h* representa o valor 32768. Uma limitação do montador é a de que valores numéricos precisam começar com caracteres entre 0 e 9. Por este motivo, é necessário iniciar um número com o dígito 0 se o seu dígito mais significativo for maior do que 9. No código de exemplo, podemos ver isso acontecer com o valor *0FFFFh*;

- *ORG*: Origem. Comando que instrui o montador sobre qual endereço deve ser utilizado como base para as alocações de endereço. No código de exemplo, ORG 15 significa que todas as instruções abaixo dessa linha devem começar no endereço 15;

- *Expressões matemáticas*: O valor dos operandos das instruções pode ser gerado a partir da avaliação de uma expressão matemática. As expressões podem conter as quatro operações básicas (soma, subtração, multiplicação e divisão), sendo que as operações de multiplicação e divisão têm precedência sobre as operações de soma e subtração (como esperado). Além disso, é possível utilizar parênteses para manipular a ordem em que as operações são avaliadas. As expressões podem envolver rótulos e substituições declaradas com o comando *equ*. No código de exemplo, podemos notar o uso de uma expressão na declaração da substituição $Var1$.

- *Inclusões*: Um arquivo pode incluir outros arquivos. Para incluir outro arquivo, é utilizado a diretiva *#include nome_do_arquivo*. O arquivo incluído é lido e inserido no local da inclusão.

Código 9.2 – Exemplo de macro

```
Espera  MACRO                ; // Duracao da espera: 7 * T + 3 ciclos de
    relogio
        LOCAL   Esp, FimEsp
Esp:    SB      Zero             ; Enquanto ( T )        // 0 - T, pede
    se T != 0
        DNP     FimEsp
        SB      HFFFF            ;   T = -1 - ( 0 - T ); // T = T - 1,
            nunca ha pede
        DNP     Esp
FimEsp:
        ENDM
```

Código 9.3 – Exemplo de macro

```
        ORG 0
Zero:   DW 0
Um:     DW 1


        IniRam equ 400 h

TROCA   MACRO       OP0,OP1,TMP      ; Troca os valores de OP0 e OP1 usando
    variavel TMP
        LM          OP0          ; T recebe o valor original de OP0, nao
            altera o pede
        EM          TMP          ; TMP = OP0
        LM          OP1
        EM          OP0          ; OP0 = OP1
        LM          TMP
        EM          OP1          ; OP1 = TMP
        ENDM

        ORG 15

        Var1 equ IniRam
        Var2 equ (Var1 + 1)
        Var3 equ (Var2 + 1)

        LM Zero
        EM Var1
        LM Um
        EM Var2
        TROCA Var1, Var2, Var3     ; Var1 = 1, Var2 = 0
```

9.2 INTERFACE DA LINHA DE COMANDO

O MES é um programa de linha de comando, e seu uso é bem simples. A seguir está a especificação de como o comando deve ser chamado.

$$mes < arquivo.asm > \ [-h]\ [-H]\ [-c]\ [-B]\ [-o < prefixo\_saida >]$$

Os parâmetros entre colchetes são opcionais. O MES também possui diferentes métodos de saída, que podem gerar mais de um arquivo. Todos esses arquivos possuem o mesmo prefixo, alterando apenas a extensão. Por padrão, esse prefixo é obtido a partir do arquivo de entrada, removendo a sua extensão.

Os parâmetros opcionais tem o seguinte significado:

- $-h$: Imprimir ajuda. Quando este parâmetro está present, o montador imprime uma mensagem de ajuda e termina a execução;
- $-H$: *Intel Hexadecimal*. Quando este parâmetro está presente, o montador emite um arquivo de extensão *.hex* com o padrão Intel (como descrito em (INTEL, 1988 (Acessado em 23 jun. 2019)));
- $-c$: Arquivo hexadecimal do CES. Quando este parâmetro está presente, o montador emite um arquivo de extensão *.ces.hex* que segue o padrão do arquivo aceito no simulador do CES ((VASCONCELOS, 2008 (Acessado em 6 fev. 2019)));
- $-B$: Binários separados. Quando este parâmetro está presente, o montador emite dois arquivos de extensões .1.*bin* e .2.*bin*. O primeiro arquivo contém os *bytes* menos significativos das palavras de 16 *bits* do programa, enquanto que o segundo arquivo contém os *bytes* mais significativos. Este modo de saída é útil para gerar arquivos que serão utilizados na implementação do CES, uma vez que foram utilizadas duas ROMs de 8 *bits* de largura cada;
- $-o < prefixo\_saida >$: Saída. Faz com que o prefixo dos arquivos de saída tenha o valor $prefixo\_saida$.

Além dos tipos de saída que podem ser habilitados com os parâmetros descritos acima, o MES sempre gera dois arquivos de saída.

- Arquivo binário: Um único arquivo binário (de extensão *.bin*) contendo o código de máquina do programa;
- Arquivo de listagem: É um arquivo de extensão *.lst* que contém o código de montagem final que foi utilizado para gerar o código de máquina, ou seja, com todas as macros e rótulos substituídos;

9.3 OPERAÇÃO DO MONTADOR

Para montar o programa recebido, o MES executa diversas passagens pela entrada.

A primeira passagem é responsável por dividir o arquivo em linhas, procurando por quebras de linha no arquivo passado e criando uma estrutura em forma de lista encadeada em que cada linha tem um ponteiro para a próxima linha.

A segunda passagem é responsável por expandir *include*s. A inclusão é realizada textualmente, ou seja, o arquivo que é incluído é lido, divido em linhas e o seu conteúdo é inserido no local da inclusão.

A terceira passagem é responsável por dividir todas as linhas em *tokens*. *Tokens* são conjuntos de caracteres que possuem algum significado, como por exemplo: o nome de rótulos, o nome das instruções, números, operadores matemáticos, etc. Os *tokens* também são armazenados em uma lista encadeada.

A quarta passagem é responsável por classificar cada linha de acordo com o seu tipo. Por exemplo, uma linha pode ser do tipo: alocação de variável, declaração de macro, declaração de *equ*, etc.

A quinta passagem é responsável por substituir todos os *equ*s.

A sexta passagem é responsável por fazer a substituição de macros.

A sétima passagem é responsável por calcular o endereço de cada linha.

A oitava passagem é responsável por substituir cada utilização de rótulo por seu devido endereço.

A nona passagem é responsável por resolver expressões matemáticas nos operandos de instruções. Para resolver, a expressão o montador utiliza um *parser* descendente recursivo que implementa uma gramática livre de contexto do tipo LL(1).

A décima passagem é responsável por gerar o código de máquina do programa e armazená-lo em um *buffer*.

Por fim, os arquivos de saída são gerados lendo o *buffer* com o código de máquina.

10 CONCLUSÃO

10.1 VISÃO GERAL

Foi apresentada uma implementação de hardware do CES, um computador antes apenas simulado. Para o desenvolvimento do computador foram utilizados conhecimentos das áreas de Circuitos Lógicos e Arquitetura de Computadores.

Alguns *softwares* auxiliares foram desenvolvidos: um programa para disposição de Circuitos Integrados em *protoboards*, programas para teste dos diversos módulos do hardware e um montador para a linguagem de montagem do CES. O desenvolvimento desses *softwares* envolveu conhecimentos nas áreas de Programação, Estruturas de Dados e conceitos básicos da área de Compiladores.

No caso dos softwares de teste, foi utilizado o microcontrolador ATmega 2560 para testar os componentes lógicos do computador. Estes programas de testes empregaram conhecimentos de Programação de Microcontroladores, que são abordados em Sistemas Embutidos, e conceitos-chave de Teste de *Software*, que foram adaptados para teste de *hardware*.

10.2 DIFICULDADES ENCONTRADAS

A maior dificuldade encontrada durante o projeto foi a tendência dos fios utilizados na montagem a oxidar e a apresentar mau contato. Este problema é amplificado pelo fato de que as condições climáticas e atmosféricas comuns da região do laboratório tendem a acelerar a oxidação de metais. Lidamos parcialmente com esta adversidade utilizando pinos de metal soldados aos fios com extremidades em duas *protoboards* diferentes. Como estes fios têm maior chance de serem movidos ao manusear o computador, é importante que estes tenham um contato firme e livre de maus contatos.

No início do projeto planejou-se implementar o computador com Circuitos Integrados da família **74HC**, pois estes empregam tecnologia CMOS. Ao buscar os componentes no mercado, nem todos estavam disponíveis nesta família, tendo que ser substituídos por CIs da família **74LS**. Como existe uma diferença nos níveis de tensão das duas famílias, muitos outros CIs tiveram que ser trocados por seus correspondentes na família **74LS**. Estas substituições resultaram em um consumo maior de energia do que o previsto inicialmente.

Outro problema relacionado com a disponibilidade de componentes no mercado ocorreu com as memórias RAM. Durante a compra dos materiais, as memórias foram encontradas somente em formato SMD. Para a montagem em *protoboards* seriam necessárias memórias em formato DIP, e por isso uma placa que adaptasse um componente SMD para DIP foi desenvolvida. Esta placa passou por três revisões, mas todas elas encontraram

problemas com mau contato e com falta de espaço na *protoboard*. Por fim, este problema foi resolvido ao realizar mais uma busca por componentes, mas desta vez incluindo memórias usadas, retiradas de computadores antigos. Havia um risco de componentes usados apresentarem defeitos, logo foram comprados mais componentes do que o necessário.

10.3 TRABALHOS FUTUROS

10.3.1 Interface de Entrada e Saída

A especificação do CES, em (VASCONCELOS, 2008 (Acessado em 5 fev. 2019)) inclui dois endereços no final do espaço de endereçamento (*0x3FFE* e *0x3FFF*), que são reservados para a comunicação com interfaces de entrada e saída.

Neste trabalho esse espaço reservado foi expandido para as últimas 16 posições de memória. Idealizou-se utilizar 4 dessas posições para realizar a comunicação com uma USART Intel 8251, e as demais posições ficariam livres para acrescentar mais dispositivos de entrada e saída. Porém, não foi possível adicionar a USART ao computador devido a restrições de tempo.

Outra possibilidade seria a implementação de uma interface de entrada e saída por *software*. Como as últimas posições de memória já estão reservadas e um decodificador já está disponível no circuito, o trabalho dessa implementação se concentraria em escrever rotinas para a implementação de algum protocolo de comunicação (por exemplo, o RS232).

10.3.2 Placa de Circuito Impresso

Embora tenhamos lidado com uma grande parte dos possíveis problemas com mau contato utilizando pinos de metal, ainda é possível que erros sejam causados por mau contato nos fios com as duas extremidades na mesma *protoboard*. O tempo de vida dessa implementação do CES não pode ser garantido, visto que a oxidação dos contatos tende a aumentar com o tempo. Uma possível solução para este problema seria a criação de uma Placa de Circuito Impresso (PCB) baseada na implementação atual.

A criação de uma PCB conta com outras vantagens além de uma durabilidade maior. Por exemplo, pode-se utilizar componentes no formato SMD, o que implica uma diminuição considerável das dimensões físicas do computador. Além disso, como componentes da família **74HC** são mais facilmente encontrados em formato SMD, uma implementação puramente CMOS do CES é possível.

Outra vantagem em comparação com construções em *protoboards* é que circuitos em PCBs possuem menos problemas com ruído, o que possibilitaria a utilização de *clocks* maiores.

Como os diagramas esquemáticos da Unidade de Controle e da Decodificação de Endereços (que corresponde às Placas 3 e 4) foram desenvolvidos durante a implementação em uma ferramenta de CAD para circuitos eletrônicos, parte do trabalho necessário para

a criação de uma PCB já foi realizada. Os diagramas das Placas 1 e 2 precisariam ser criados, assim como o *design* da Placa de Circuito Impresso.

10.4 OBSERVAÇÕES FINAIS

A implementação do CES utilizando os elementos mais simples da lógica combinacional e sequencial permite que estudantes de Arquitetura de Computadores tenham um entendimento mais claro de como um computador é construído a partir de elementos lógicos. Muitas vezes uma barreira mental de abstração é criada, e a ligação entre os componentes de um computador e circuitos lógicos é perdida. Esperamos que a existência de uma implementação física do computador ajude a quebrar essa barreira.

Durante o desenvolvimento do computador, uma disciplina rigorosa de testes foi aplicada. Estes testes garantiram que falhas, tanto na montagem do circuito quanto nos componentes, fossem descobertas logo após a montagem de partes do circuito, com seus elementos isolados. A descoberta de falhas o mais cedo possível é muito importante, pois ao longo do tempo elas tendem a se propagar, e a causa raiz se torna cada vez mais difícil de encontrar. Este trabalho mostra que uma disciplina de testes é valiosa mesmo quando não relacionada ao Desenvolvimento de *Software*.

REFERÊNCIAS

ARDUINO. **ATmega2560-Arduino Pin Mapping**. [S.l.], 2017 (Acessado em mai. 2018). <https://www.arduino.cc/en/Hacking/PinMapping2560>.

ATMEL. **Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V**. [S.l.], 2014 (Acessado em mai. 2018). <http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf>.

COMPUTERHOPE. **IDE**. [S.l.], 2017 (Acessado em 10 mar. 2019). <https://www.computerhope.com/jargon/i/ide.htm>.

CONSTANTINO, P. **Complete Home-Built 8-bit CPU + Computer with graphics and sound made from scratch using 74HC Logic**. [S.l.], 2017 (Acessado em abr. 2018). <https://www.youtube.com/watch?v=g_ZaioqF1B0>.

CONSTANTINO, P. **A CPU/Minicomputer built from 74HC series logic.** [S.l.], 2018 (Acessado em abr. 2018). <https://github.com/Pconst167/dreamcatcher>.

EATER, B. **Build an 8-bit computer from scratch**. [S.l.], 2016 (Acessado em mar. 2018). <https://eater.net/8bit/>.

INTEL. **Intel Hexadecimal Object File**. [S.l.], 1988 (Acessado em 23 jun. 2019). <https://ia601409.us.archive.org/33/items/IntelHEXStandard/Intel%20HEX%20Standard.pdf>.

INTRUMENTS, T. **SN54173, SN54LS173A, SN74173, SN74LS173A 4-BIT D-TYPE REGISTERS WITH 3-STATE OUTPUTS**. [S.l.], 1999 (Acessado em 6 jul. 2019). <http://www.ti.com/lit/ds/symlink/sn74ls173a.pdf>.

INTRUMENTS, T. **SN5404, SN54LS04, SN54S04, SN7404, SN74LS04, SN74S04 HEX INVERTERS**. [S.l.], 2004 (Acessado em 23 jun. 2019). <http://www.ti.com/lit/ds/symlink/sn54ls04-sp.pdf>.

KERVINCK, M. van; BELGERS, W. **Gigatron – TTL microcomputer**. [S.l.], 2017 (Acessado em mar. 2018). <https://gigatron.io/>.

MEDIA, E. **Logic Signal Voltage Levels**. [S.l.], (Acessado em jan. 2019). <https://www.allaboutcircuits.com/textbook/digital/chpt-3/logic-signal-voltage-levels/>.

PIRES, E. **A breadboard IC placement utility using SDL**. [S.l.], 2018 (Acessado em fev. 2019). <https://github.com/erickpires/bread_placer/>.

PIRES, E. **Montador Extremamente Simples**. [S.l.], 2018 (Acessado em fev. 2019). <https://github.com/erickpires/mes/>.

PITTET, S. **The different types of software testing**. [S.l.], (Acessado em 6 fev. 2019). <https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>.

SOLUTION, I. S. **32K x 8 HIGH-SPEED CMOS STATIC RAM**. [S.l.], 2009 (Acessado em 6 jul. 2019). <http://www.issi.com/WW/pdf/61C256AH.pdf>.

VASCONCELOS, N. Q. **Um Computador Extremamente Simples (CES)**. [S.l.], 2008 (Acessado em 5 fev. 2019). <http://www.dcc.ufrj.br/~cp/CES/0CES.htm>.

VASCONCELOS, N. Q. **SimCES**. [S.l.], 2008 (Acessado em 6 fev. 2019). <http://www.dcc.ufrj.br/~cp/NovoCES/SimCES.exe>.

APÊNDICES

APÊNDICE A – TESTE DAS PLACAS

Código A.1 – Placa 1

```
// --- Variable types defined by length ---

typedef char int8;
typedef byte uint8;
typedef short int16;
typedef unsigned short uint16;
typedef long int32;
typedef unsigned long uint32;
typedef uint8 bool;

//
// Distribuicao das portas
//
// * Chaves de Dados  = PORTF e PORTK = Pins[A0..A7] e Pins[A8..A15]
// * Via de Dados     = PORTA e PORTB = Pins[22..29] e Pins[53..50],Pins
    [10..13]
// * Saida do Somador = PORTC e PORTL = Pins[37..30] e Pins[49..42]

// * Clock            = PORTE4 = Pins[2]
// * SaiBufferChaves  = PORTE5 = Pins[3]
// * SaiRD            = PORTG5 = Pins[4]
// * SaiRT            = PORTE3 = Pins[5]
// * SaiRP            = PORTH3 = Pins[6]
// * ZComp            = PORTH4 = Pins[7]
// * MaisUm           = PORTH5 = Pins[8]
// * VaiUm            = PORTH6 = Pins[9]

#define nop() __asm__ volatile("nop\n\t")

static bool chaves_ativadas = false;
static bool rp_ativado      = false;
static bool rd_ativado      = false;

static volatile uint16 valor_lido;

void clock() {
    PORTE |= (1 << 4);
    nop();

    PORTE &= ~(1 << 4);
```

```
    nop();
}

void ativa_chave_de_dados() {
    if(rp_ativado || rd_ativado) {
        Serial.println("Mais de uma entrada ativa ao mesmo tempo.");
        while(true);
    }

    PORTE &= ~(1 << 5);
    chaves_ativadas = true;

    nop();
}

void desativa_chave_de_dados() {
    PORTE |= (1 << 5);
    chaves_ativadas = false;

    nop();
}

void escreve_chave_de_dados(uint16 valor) {
    uint8 lower_byte = valor;
    uint8 upper_byte = valor >> 8;

    PORTF = lower_byte;
    PORTK = upper_byte;

    nop();
}

void ativa_zcomp() {
    PORTH |= (1 << 4);

    nop();
}

void desativa_zcomp() {
    PORTH &= ~(1 << 4);

    nop();
}

void ativa_mais_um() {
    PORTH |= (1 << 5);
```

```
    nop();
}

void desativa_mais_um() {
    PORTH &= ~(1 << 5);

    nop();
}

void ativa_rd() {
    if(rp_ativado || chaves_ativadas) {
        Serial.println("Mais de uma entrada ativa ao mesmo tempo.");
        while(true);
    }

    PORTG &= ~(1 << 5);
    rd_ativado = true;

    nop();
}

void desativa_rd() {
    PORTG |= (1 << 5);
    rd_ativado = false;

    nop();
}

void escreve_em_rd(uint16 valor) {
    uint8 lower_byte = valor;
    uint8 upper_byte = valor >> 8;

    PORTA = lower_byte;
    PORTB = upper_byte;
    nop();

    clock();

    nop();
}

void ativa_rp() {
    if(chaves_ativadas || rd_ativado) {
        Serial.println("Mais de uma entrada ativa ao mesmo tempo.");
        while(true);
    }
```

```
    PORTH &= ~(1 << 3);
    rp_ativado = true;

    nop();
}

void desativa_rp() {
    PORTH |= (1 << 3);
    rp_ativado = false;

    nop();
}

void ativa_rt() {
    PORTE |= (1 << 3);

    nop();
}

uint16 le_somador() {
    nop();

    uint8 lower_byte = PINC;
    uint8 upper_byte = PINL;

    return (upper_byte << 8) | lower_byte;
}

uint8 le_vai_um() {
    return (PINH & (1 << 6));
}

bool checa_somador(uint16 valor) {
    valor_lido = le_somador();

    return valor == valor_lido;
}

void reporta_diff(uint16 expected, uint16 received) {
    char buff[8];
    uint16 diff = expected ^ received;
    sprintf(buff, "0x%04x", diff);

    uint8 popcnt = 0;
    uint8 mask = 0x01;
    for(uint8 _ = 0; _ < 16; _++) {
        if(diff & mask) { popcnt++; }
```

```
        mask = mask << 1;
    }

    Serial.print("\t\tDiff: ");
    Serial.print(buff);
    Serial.print(" PopCnt: ");
    Serial.println(popcnt);
}

bool testa_zero() {
    escreve_chave_de_dados(0);
    ativa_zcomp();
    desativa_mais_um();
    ativa_chave_de_dados();

    bool result = checa_somador(0);
    if(!result) {
        Serial.println("Falha ao testar zero pelas chaves de dados.");
        reporta_diff(0, valor_lido);
    }

    desativa_chave_de_dados();
    return result;
}

bool testa_chaves_de_dados() {
    ativa_zcomp();
    desativa_mais_um();
    ativa_chave_de_dados();

    for(uint16 i = 1; i != 0; i++) {
        escreve_chave_de_dados(i);

        bool result = checa_somador(i);
        if(!result) {
            Serial.print("Falha ao testar chaves de dados com: ");
            Serial.println(i);
            Serial.print("Valor lido: ");
            Serial.println(valor_lido);
            reporta_diff(i, valor_lido);

            desativa_chave_de_dados();

            return 0;
        }
    }
```

```
    desativa_chave_de_dados();
    return 1;
}

bool testa_mais_um() {
    ativa_zcomp();
    ativa_mais_um();
    ativa_chave_de_dados();

    uint16 i = 0;
    do {
        escreve_chave_de_dados(i);

        bool result = checa_somador(i + 1);
        if(!result) {
            Serial.print("Falha ao testar mais um com: ");
            Serial.println(i);
            Serial.print("Valor lido: ");
            Serial.println(valor_lido);
            reporta_diff(i, valor_lido);

            desativa_chave_de_dados();
            return 0;
        }
    } while(++i != 0);

    desativa_chave_de_dados();
    return 1;
}


bool testa_rd() {
    ativa_zcomp();
    desativa_mais_um();
    ativa_rd();

    uint16 i = 0;
    do {
        escreve_em_rd(i);
        bool result = checa_somador(i);
        if(!result) {
            Serial.print("Falha ao testar RD com: ");
            Serial.println(i);
            Serial.print("Valor lido: ");
            Serial.println(valor_lido);
            reporta_diff(i, valor_lido);
```

```
            desativa_rd();
            return false;
        }

    } while(++i != 0);

    desativa_rd();
    return true;
}

bool testa_rp() {
    ativa_zcomp();
    desativa_mais_um();

    escreve_chave_de_dados(0);
    ativa_chave_de_dados();
    clock();
    desativa_chave_de_dados();


    ativa_rp();
    bool result = checa_somador(0);
    if(!result) {
        Serial.print("Falha ao checar RP com: ");
        Serial.println(0);
        Serial.print("Valor lido: ");
        Serial.println(valor_lido);
        reporta_diff(0, valor_lido);

        //        while(true);

        desativa_rp();
        return false;
    }

    ativa_mais_um();
    result = checa_somador(1);
    if(!result) {
        Serial.println("Falha ao checar MaisUm");
        reporta_diff(1, valor_lido);

        desativa_rp();
        return false;
    }

    for(uint16 i = 1; i < 0x4000; i++) {
```

```
        nop();
        nop();
        bool result = checa_somador(i);
        if(!result) {
            Serial.print("Falha ao checar RP com: ");
            Serial.println(i);
            Serial.print("Valor lido: ");
            Serial.println(valor_lido);
            reporta_diff(i, valor_lido);

            // while(true);

            desativa_rp();
            return false;
        }

        clock();
    }

    desativa_rp();
    return true;
}

bool testa_rt() {
    desativa_mais_um();
    ativa_zcomp();
    escreve_chave_de_dados(0);
    ativa_chave_de_dados();

    clock();

    desativa_zcomp();
    ativa_mais_um();

    bool result = checa_somador(0);
    if(!result) {
        Serial.print("Falha ao checar RT com: ");
        Serial.println(0);
        Serial.print("Valor lido: ");
        Serial.println(valor_lido);
        reporta_diff(0, valor_lido);

        desativa_chave_de_dados();
        return 0;
    }

    for(uint16 i = 1; i != 0; i++) {
```

```
            desativa_mais_um();
            ativa_zcomp();
            escreve_chave_de_dados(i);

            clock();

            escreve_chave_de_dados(0);
            desativa_zcomp();
            ativa_mais_um();

            uint16 esperado = 0 - i;
            result = checa_somador(esperado);
            if(!result) {
                Serial.print("Falha ao testar RT com: ");
                Serial.println(i);
                Serial.print("Valor lido: ");
                Serial.println(valor_lido);
                reporta_diff(i, valor_lido);

                desativa_chave_de_dados();
                return 0;
            }
        }

        desativa_chave_de_dados();
        return 1;
    }

    bool testa_somadores() {
        desativa_mais_um();
        ativa_chave_de_dados();

        for(uint8 somador = 0; somador < 4; somador++) {
            for(uint16 direito = 0; direito < 16; direito++) {
                uint16 lado_direito = direito << (4 * somador);

                ativa_zcomp();
                escreve_chave_de_dados(~(lado_direito));
                clock();

                desativa_zcomp();

                for(uint16 esquerdo = 0; esquerdo < 16; esquerdo++) {
                    uint16 lado_esquerdo = esquerdo << (4 * somador);

                    escreve_chave_de_dados(lado_esquerdo);
```

```
                uint16 lido = le_somador();
                uint8 vai_um = le_vai_um() != 0;

                uint16 expected = (lado_esquerdo + lado_direito);
                uint8  expected_vai_um = lado_esquerdo > (0xffff -
                    lado_direito);

                if(lido != expected || vai_um != expected_vai_um) {
                    Serial.print(lado_esquerdo);
                    Serial.print(" + ");
                    Serial.print(lado_direito);
                    Serial.print(" = ");
                    Serial.print(lido);
                    Serial.print('(');
                    Serial.print(vai_um);
                    Serial.print(") == ");
                    Serial.print(expected);
                    Serial.print('(');
                    Serial.print(expected_vai_um);
                    Serial.println(')');
                    reporta_diff(expected, valor_lido);

                    desativa_chave_de_dados();
                    return false;
                }
            }
        }
    }

    desativa_chave_de_dados();
    return true;
}

void setup() {
    Serial.begin(9600);

    DDRF = 0xff;
    DDRK = 0xff;

    DDRA = 0xff;
    DDRB = 0xff;

    DDRC = 0x00;
    DDRL = 0x00;

    DDRE |= (1 << 4);
    DDRE |= (1 << 5);
```

```
    DDRG |= (1 << 5);
    DDRE |= (1 << 3);
    DDRH |= (1 << 3);
    DDRH |= (1 << 4);
    DDRH |= (1 << 5);
    DDRH &= ~(1 << 6);

    ativa_rt();
    desativa_zcomp();
    desativa_chave_de_dados();
    desativa_rp();
    desativa_rd();
    desativa_mais_um();
    clock();
}

void loop() {
    delay(2000);

    bool result;

    Serial.print("Testando Zero... ");
    result = testa_zero();
    if(!result) { return; }
    Serial.println("Ok");

    Serial.print("Testando Chaves de Dados... ");
    result = testa_chaves_de_dados();
    if(!result) { return; }
    Serial.println("Ok");

    Serial.print("Testando Mais Um... ");
    result = testa_mais_um();
    if(!result) { return; }
    Serial.println("Ok");


    Serial.print("Testando RD... ");
    result = testa_rd();
    if(!result) { return; }
    Serial.println("Ok");

    /*
    */
    Serial.print("Testando RP... ");
    result = testa_rp();
    if(!result) { return; }
```

```
    Serial.println("Ok");
    Serial.print("Testando RT... ");
    result = testa_rt();
    if(!result) { return; }
    Serial.println("Ok");

    /*
    */
    Serial.print("Testando Somadores...");
    result = testa_somadores();
    if(!result) { return; }
    Serial.println("Ok");

}
```

Código A.2 – Placa 2: BD e BT

```
// --- Variable types defined by length ---
typedef char int8;
typedef byte uint8;
typedef short int16;
typedef unsigned short uint16;
typedef long int32;
typedef unsigned long uint32;

#define nop() __asm__ volatile("nop\n\t")

// Saidas: VDB
// Entradas: SRT e CHD

// Distribuicao das portas
//
// * Chaves de Dados  = PORTF e PORTK = Pins[A0..A7] e Pins[A8..A15]
// * Via de Dados     = PORTA e PORTB = Pins[22..29] e Pins[53..50],Pins
   [10..13]
// * Saida de RT      = PORTC e PORTL = Pins[37..30] e Pins[49..42]

// * Clock = PORTE4 = Pins[2]
// * SaiBD = PORTE5 = Pins[3]
// * SaiRE = PORTG5 = Pins[4]
// * SaiBE = PORTE3 = Pins[5]
// * EscM  = PORTH3 = Pins[6]
// * UltC  = PORTH4 = Pins[7]

static uint16 valor_lido;

void reporta_diff(uint16 expected, uint16 received) {
    char buff[8];
    uint16 diff = expected ^ received;
    sprintf(buff, "0x%04x", diff);

    uint8 popcnt = 0;
    uint8 mask = 0x01;
    for(uint8 _ = 0; _ < 16; _++) {
        if(diff & mask) { popcnt++; }

        mask = mask << 1;
    }

    Serial.print("\t\tDiff: ");
    Serial.print(buff);
    Serial.print(" PopCnt: ");
    Serial.println(popcnt);
```

```
}

void desativa_sai_bd() {
    PORTE |= (1 << 5);
    nop();
}

void desativa_esc_m() {
    PORTH |= (1 << 3);
    nop();
}

void desativa_sai_re() {
    PORTG |= (1 << 5);
    nop();
}

void desativa_sai_be() {
    PORTE |= (1 << 3);
    nop();
}

void vdb_leitura() {
    DDRA = 0x00;
    DDRB = 0x00;
    nop();
}

void chd_escrita() {
    DDRF = 0xff;
    DDRK = 0xff;
    nop();
}

void srt_escrita() {
    DDRC = 0xff;
    DDRL = 0xff;
    nop();
}

void controle_escrita() {
    DDRE |= (1 << 4);
    DDRE |= (1 << 5);
    DDRG |= (1 << 5);
    DDRE |= (1 << 3);
    DDRH |= (1 << 3);
    DDRH |= (1 << 4);
```

```
}

void setup() {
    vdb_leitura();
    chd_escrita();
    srt_escrita();

    controle_escrita();

    desativa_sai_bd();
    desativa_esc_m();
    desativa_sai_re();
    desativa_sai_be();

    Serial.begin(9600);
}

void ativa_sai_bd() {
    PORTE &= ~(1 << 5);
    nop();
}

void ativa_esc_m() {
    PORTH &= ~(1 << 3);
    nop();
}

void escreve_chave_de_dados(uint16 valor) {
    uint8 lower_byte = valor;
    uint8 upper_byte = valor >> 8;

    PORTF = lower_byte;
    PORTK = upper_byte;

    nop();
}

void escreve_srt(uint16 valor) {
    uint8 lower_byte = valor;
    uint8 upper_byte = valor >> 8;

    PORTC = lower_byte;
    PORTL = upper_byte;

    nop();
}
```

```
uint16 le_vdb() {
    nop();

    uint8 lower_byte = PINA;
    uint8 upper_byte = PINB;

    return (upper_byte << 8) | lower_byte;
}

bool checa_vdb(uint16 valor) {
    valor_lido = le_vdb();

    return valor == valor_lido;
}

bool testa_bd() {
    ativa_sai_bd();

    escreve_chave_de_dados(0);

    bool result = checa_vdb(0);

    if(!result) {
        Serial.print("Falha ao checar BT com: ");
        Serial.println(0);
        Serial.print("Valor lido: ");
        Serial.println(valor_lido);
        reporta_diff(0, valor_lido);

        desativa_sai_bd();
        return 0;
    }

    for(uint16 i = 1; i != 0; i++) {
        escreve_chave_de_dados(i);

        result = checa_vdb(i);
        if(!result) {
            Serial.print("Falha ao checar BT com: ");
            Serial.println(i);
            Serial.print("Valor lido: ");
            Serial.println(valor_lido);
            reporta_diff(i, valor_lido);

            desativa_sai_bd();
            return 0;
        }
```

```
    }

    desativa_sai_bd();
    return 1;
}

bool testa_bt() {
    ativa_esc_m();

    escreve_srt(0);

    bool result = checa_vdb(0);

    if(!result) {
        Serial.print("Falha ao checar BT com: ");
        Serial.println(0);
        Serial.print("Valor lido: ");
        Serial.println(valor_lido);
        reporta_diff(0, valor_lido);

        desativa_esc_m();
        return 0;
    }

    for(uint16 i = 1; i != 0; i++) {
        escreve_srt(i);

        result = checa_vdb(i);
        if(!result) {
            Serial.print("Falha ao checar BT com: ");
            Serial.println(i);
            Serial.print("Valor lido: ");
            Serial.println(valor_lido);
            reporta_diff(i, valor_lido);

            desativa_esc_m();
            return 0;
        }
    }

    desativa_esc_m();
    return 1;
}

void loop() {
    delay(500);
```

```
    bool result;

    Serial.print("Testando BD... ");
    result = testa_bd();
    if(!result) { return; }
    Serial.println("Ok");

    Serial.print("Testando BT... ");
    result = testa_bt();
    if(!result) { return; }
    Serial.println("Ok");
}
```

Código A.3 – Placa 2: ME e RE

```
// --- Variable types defined by length ---
typedef char int8;
typedef byte uint8;
typedef short int16;
typedef unsigned short uint16;
typedef long int32;
typedef unsigned long uint32;

#define MAX_END 16383

#define nop() __asm__ volatile("nop\n\t")

// Saidas: VEB
// Entradas:VDB e SS

// Distribuicao das portas
//
// * Via de Enderecos = PORTF e PORTK = Pins[A0..A7] e Pins[A8..A15]
// * Via de Dados     = PORTA e PORTB = Pins[22..29] e Pins[53..50],Pins
   [10..13]
// * Saida do Somador = PORTC e PORTL = Pins[37..30] e Pins[49..42]

// * Clock = PORTE4 = Pins[2]
// * SaiBD = PORTE5 = Pins[3]
// * SaiRE = PORTG5 = Pins[4]
// * SaiBE = PORTE3 = Pins[5]
// * EscM  = PORTH3 = Pins[6]
// * UltC  = PORTH4 = Pins[7]

static uint16 valor_lido;

void reporta_diff(uint16 expected, uint16 received) {
    char buff[8];
    uint16 diff = expected ^ received;
    sprintf(buff, "0x%04x", diff);

    uint8 popcnt = 0;
    uint8 mask = 0x01;
    for(uint8 _ = 0; _ < 16; _++) {
        if(diff & mask) { popcnt++; }

        mask = mask << 1;
    }

    Serial.print("\t\tDiff: ");
    Serial.print(buff);
```

```
    Serial.print(" PopCnt: ");
    Serial.println(popcnt);
}

void desativa_sai_bd() {
    PORTE |= (1 << 5);
    nop();
}

void desativa_esc_m() {
    PORTH |= (1 << 3);
    nop();
}

void desativa_sai_re() {
    PORTG |= (1 << 5);
    nop();
}

void desativa_sai_be() {
    PORTE |= (1 << 3);
    nop();
}

void controle_escrita() {
    DDRE |= (1 << 4);
    DDRE |= (1 << 5);
    DDRG |= (1 << 5);
    DDRE |= (1 << 3);
    DDRH |= (1 << 3);
    DDRH |= (1 << 4);
}

void vdb_escrita() {
    DDRA = 0xff;
    DDRB = 0xff;
    nop();
}

void ss_escrita() {
    DDRC = 0xff;
    DDRL = 0xff;
}

void veb_leitura() {
    DDRF = 0x00;
    DDRK = 0x00;
```

```
}

void setup() {
    vdb_escrita();
    ss_escrita();
    veb_leitura();

    controle_escrita();

    desativa_sai_bd();
    desativa_esc_m();
    desativa_sai_re();
    desativa_sai_be();


    Serial.begin(9600);
}

void clock() {
    PORTE |= (1 << 4);
    nop();

    PORTE &= ~(1 << 4);
    nop();
}

void ativa_sai_re() {
    PORTG &= ~(1 << 5);
    nop();
}

void ult_c_zero() {
    PORTH &= ~(1 << 4);
    nop();
}

void ult_c_um() {
    PORTH |= (1 << 4);
    nop();
}

void escreve_vdb(uint16 valor) {
    uint8 lower_byte = valor;
    uint8 upper_byte = valor >> 8;

    PORTA = lower_byte;
    PORTB = upper_byte;
```

```
    nop();
}

void escreve_ss(uint16 valor) {
    uint8 lower_byte = valor;
    uint8 upper_byte = valor >> 8;

    PORTC = lower_byte;
    PORTL = upper_byte;

    nop();
}

uint16 le_veb() {
    nop();

    uint8 lower_byte = PINF;
    uint8 upper_byte = PINK;

    upper_byte &= 0x3f;

    return (upper_byte << 8) | lower_byte;
}

bool checa_veb(uint16 valor) {
    valor_lido = le_veb();

    return valor == valor_lido;
}

bool testa_me() {
    ativa_sai_re();

    for(uint16 i = 0; i <= MAX_END; i++) {
        uint16 complemento = MAX_END - i;

        escreve_vdb(i);
        escreve_ss(complemento);

        ult_c_zero();
        nop();
        clock();

        bool result = checa_veb(i);
        if(!result) {
            Serial.print("Falha ao checar ME com: ");
```

```
            Serial.println(i);
            Serial.print("Valor lido: ");
            Serial.println(valor_lido);
            Serial.println("UltC: 0");
            reporta_diff(i, valor_lido);

            Serial.println("\tEntrada por VDB");

            desativa_sai_re();
            return 0;
        }

        ult_c_um();
        nop();
        clock();

        result = checa_veb(complemento);
        if(!result) {
            Serial.print("Falha ao checar ME com: ");
            Serial.println(complemento);
            Serial.print("Valor lido: ");
            Serial.println(valor_lido);
            Serial.println("UltC: 1");
            reporta_diff(complemento, valor_lido);

            Serial.println("\tEntrada por SS");

            desativa_sai_re();
            return 0;
        }
    }

    desativa_sai_re();
    return 1;
}

void loop() {
    delay(500);

    bool result;

    Serial.print("Testando ME... ");
    result = testa_me();
    if(!result) { return; }
    Serial.println("Ok");

}
```

Código A.4 – Placa 2: BE

```
// --- Variable types defined by length ---
typedef char int8;
typedef byte uint8;
typedef short int16;
typedef unsigned short uint16;
typedef long int32;
typedef unsigned long uint32;

#define MAX_END 16383

#define nop() __asm__ volatile("nop\n\t")

// Saidas: VEB
// Entradas: CHE

// Distribuicao das portas
//
// * Via de Enderecos    = PORTF e PORTK = Pins[A0..A7] e Pins[A8..A15]
// * Chaves de Enderecos = PORTC e PORTL = Pins[37..30] e Pins[49..42]

// * Clock = PORTE4 = Pins[2]
// * SaiBD = PORTE5 = Pins[3]
// * SaiRE = PORTG5 = Pins[4]
// * SaiBE = PORTE3 = Pins[5]
// * EscM  = PORTH3 = Pins[6]
// * UltC  = PORTH4 = Pins[7]

static uint16 valor_lido;

void reporta_diff(uint16 expected, uint16 received) {
    char buff[8];
    uint16 diff = expected ^ received;
    sprintf(buff, "0x%04x", diff);

    uint8 popcnt = 0;
    uint8 mask = 0x01;
    for(uint8 _ = 0; _ < 16; _++) {
        if(diff & mask) { popcnt++; }

        mask = mask << 1;
    }

    Serial.print("\t\tDiff: ");
    Serial.print(buff);
    Serial.print(" PopCnt: ");
    Serial.println(popcnt);
```

```
}

void desativa_sai_bd() {
    PORTE |= (1 << 5);
    nop();
}

void desativa_esc_m() {
    PORTH |= (1 << 3);
    nop();
}

void desativa_sai_re() {
    PORTG |= (1 << 5);
    nop();
}

void desativa_sai_be() {
    PORTE |= (1 << 3);
    nop();
}

void controle_escrita() {
    DDRE |= (1 << 4);
    DDRE |= (1 << 5);
    DDRG |= (1 << 5);
    DDRE |= (1 << 3);
    DDRH |= (1 << 3);
    DDRH |= (1 << 4);
}

void che_escrita() {
    DDRC = 0xff;
    DDRL = 0xff;
}

void veb_leitura() {
    DDRF = 0x00;
    DDRK = 0x00;
}

void setup() {
    che_escrita();
    veb_leitura();

    controle_escrita();
```

```
    desativa_sai_bd();
    desativa_esc_m();
    desativa_sai_re();
    desativa_sai_be();

    Serial.begin(9600);
}

void ativa_sai_be() {
    PORTE &= ~(1 << 3);
    nop();
}

void escreve_chave_de_enderecos(uint16 valor) {
    uint8 lower_byte = valor;
    uint8 upper_byte = valor >> 8;

    PORTC = lower_byte;
    PORTL = upper_byte;

    nop();
}

uint16 le_veb() {
    nop();

    uint8 lower_byte = PINF;
    uint8 upper_byte = PINK;

    return (upper_byte << 8) | lower_byte;
}

bool checa_veb(uint16 valor) {
    valor_lido = le_veb();

    return valor == valor_lido;
}

bool testa_be() {
    ativa_sai_be();

    for(uint16 i = 0; i <= MAX_END; i++) {
        escreve_chave_de_enderecos(i);

        bool result = checa_veb(i);
        if(!result) {
            Serial.print("Falha ao checar BE com: ");
```

```
            Serial.println(i);
            Serial.print("Valor lido: ");
            Serial.println(valor_lido);
            reporta_diff(i, valor_lido);

            desativa_sai_be();
            return 0;
        }
    }

    desativa_sai_be();
    return 1;
}

void loop() {
    delay(500);

    bool result;

    Serial.print("Testando BE... ");
    result = testa_be();
    if(!result) { return; }
    Serial.println("Ok");
}
```

Código A.5 – Placa 1 e 2

```
// --- Variable types defined by length ---
typedef char int8;
typedef byte uint8;
typedef short int16;
typedef unsigned short uint16;
typedef long int32;
typedef unsigned long uint32;

#define nop() __asm__ volatile("nop\n\t")

// Distribuicao das portas
//
// * Via de Enderecos = PORTC e PORTL = Pins[37..30] e Pins[49..42]
// * Via de Dados     = PORTF e PORTK = Pins[A0..A7] e Pins[A8..A15]

// * Clock  = PORTE4 = Pins[2]  (saida)
// * SaiBD\ = PORTE5 = Pins[3]  (saida)
// * SaiRE\ = PORTG5 = Pins[4]  (saida)
// * SaiBE\ = PORTE3 = Pins[5]  (saida)
// * EscM\  = PORTH3 = Pins[6]  (saida)
// * UltC\  = PORTH4 = Pins[7]  (saida)
// * ZComp  = PORTH5 = Pins[8]  (saida)
// * SaiBS\ = PORTH6 = Pins[9]  (saida)
// * MaisUm = PORTB4 = Pins[10] (saida)
// * VaiUm  = PORTB5 = Pins[11] (entrada)
// * SaiRP\ = PORTB6 = Pins[12] (saida)
// * EscRP\ = PORTB7 = Pins[13] (saida)
// * EscT\  = PORTJ1 = Pins[14] (saida)


void clock() {
    PORTE |= (1 << 4);
    nop();

    PORTE &= ~(1 << 4);
    nop();
}

void desativa_sai_bd() {
    PORTE |= (1 << 5);
    nop();
}

void desativa_esc_m() {
    PORTH |= (1 << 3);
    nop();
```

```
}

void desativa_sai_bt() {
    desativa_esc_m();
}

void desativa_sai_re() {
    PORTG |= (1 << 5);
    nop();
}

void desativa_sai_be() {
    PORTE |= (1 << 3);
    nop();
}

void desativa_sai_bs() {
    PORTH |= (1 << 6);
    nop();
}

void desativa_sai_rp() {
    PORTB |= (1 << 6);
    nop();
}

void desativa_esc_rp() {
    PORTB |= (1 << 7);
    nop();
}

void desativa_esc_t() {
    PORTJ |= (1 << 1);
}

void ativa_sai_re() {
    PORTG &= ~(1 << 5);
    nop();
}

void ativa_sai_be() {
    PORTE &= ~(1 << 3);
    nop();
}

void ativa_sai_bd() {
    PORTE &= ~(1 << 5);
```

```
    nop();
}

void ativa_sai_bs() {
    PORTH &= ~(1 << 6);
    nop();
}

void desativa_z_comp() {
    PORTH &= ~(1 << 5);
}

void ativa_z_comp() {
    PORTH |= (1 << 5);
    nop();
}

void ativa_esc_m() {
    PORTH &= ~(1 << 3);
    nop();
}

void ativa_sai_bt() { // a.k.a. EscM
    ativa_esc_m();
}

void ativa_sai_rp() {
    PORTB &= ~(1 << 6);
}

void ativa_esc_rp() {
    PORTB &= ~(1 << 7);
}

void ativa_esc_t() {
    PORTJ &= ~(1 << 1);
    nop();
}

void ult_c_zero() {
    PORTH &= ~(1 << 4);
    nop();
}

void ult_c_um() {
    PORTH |= (1 << 4);
    nop();
```

```
}

void desativa_sai_rd() {
    return ult_c_um();
}

void mais_um(uint8 valor) {
    if(valor) {
        PORTB |= (1 << 4);
    } else {
        PORTB &= ~(1 << 4);
    }
}

void vdb_escrita() {
    DDRF = 0xff;
    DDRK = 0xff;
    nop();
}

void vdb_leitura() {
    DDRF = 0x00;
    DDRK = 0x00;
    nop();
}

void veb_leitura() {
    DDRC = 0x00;
    DDRL = 0x00;
}

void setup_controle() {
    DDRE |= (1 << 4);
    DDRE |= (1 << 5);

    DDRG |= (1 << 5);

    DDRE |= (1 << 3);

    DDRH |= (1 << 3);
    DDRH |= (1 << 4);
    DDRH |= (1 << 5);
    DDRH |= (1 << 6);

    DDRB |= (1 << 3);
    DDRB |= (1 << 4);
    DDRB &= ~(1 << 5);
```

```
    DDRB |= (1 << 6);
    DDRB |= (1 << 7);

    DDRJ |= (1 << 1);
}

void setup() {
    vdb_leitura();
    veb_leitura();

    setup_controle();

    desativa_sai_bd();
    desativa_sai_be();
    desativa_sai_re();
    desativa_sai_bs();
    desativa_esc_m();

    ult_c_zero();

    desativa_z_comp();
    mais_um(0);

    desativa_sai_rp();
    desativa_esc_rp();

    desativa_esc_t();

    Serial.begin(9600);
}

uint16 le_veb() {
    nop();

    uint8 lower_byte = PINC;
    uint8 upper_byte = PINL;

    upper_byte &= 0x3f;

    return (upper_byte << 8) | lower_byte;
}


bool checa_estabilidade_che(uint16 valor) {
    for(uint8 i = 0; i < 255; i++) {
        uint16 result = le_veb();
        if(result != valor) {
```

```
            return false;
        }

        nop();
        nop();
    }

    return true;
}

void testa_chave_enderecos() {
    ativa_sai_be();

    uint16 valor_esperado = 0x0001;

    for(uint8 i = 0; i <= 14; i++) {
        uint16 result;

        while((result = le_veb()) != valor_esperado) {
            char buff[8];
            sprintf(buff, "%04x", valor_esperado);
            Serial.print("Insira o valor 0x");
            Serial.print(buff);
            Serial.print(" nas chaves de enderecos (valor lido 0x");

            sprintf(buff, "%04x", result);
            Serial.print(buff);
            Serial.println(")");
        }

        if(!checa_estabilidade_che(valor_esperado)) {
            Serial.println("Falha de estabilidade");
            goto END;
        }

        valor_esperado = (valor_esperado << 1) & 0x3fff;
    }

END:
    desativa_sai_be();
}

uint16 le_vdb() {
    nop();

    uint8 lower_byte = PINF;
    uint8 upper_byte = PINK;
```

```
    return (upper_byte << 8) | lower_byte;
}

void escreve_vdb(uint16 valor) {
    uint8 lower_byte = valor;
    uint8 upper_byte = valor >> 8;

    PORTF = lower_byte;
    PORTK = upper_byte;

    nop();
}

bool checa_estabilidade_vdb(uint16 valor) {
    for(uint8 i = 0; i < 255; i++) {
        uint16 result = le_vdb();
        if(result != valor) {
            return false;
        }

        nop();
        nop();
    }

    return true;
}

void testa_chave_de_dados_via_bd() {
    vdb_leitura();
    ativa_sai_bd();

    uint16 valor_esperado = 0x0001;

    for(uint8 i = 0; i <= 16; i++) {
        uint16 result;

        while((result = le_vdb()) != valor_esperado) {
            char buff[8];
            sprintf(buff, "%04x", valor_esperado);
            Serial.print("Insira o valor 0x");
            Serial.print(buff);
            Serial.print(" nas chaves de dados (valor lido 0x");

            sprintf(buff, "%04x", result);
            Serial.print(buff);
            Serial.println(")");
```

```
        }

        if(!checa_estabilidade_vdb(valor_esperado)) {
            Serial.println("Falha de estabilidade");
            goto END;
        }

        valor_esperado = (valor_esperado << 1) & 0xffff;
    }

END:
    desativa_sai_bd();
}

uint16 le_rt() {
    nop();

    clock();

    return le_vdb();
}

bool checa_estabilidade_rt(uint16 valor) {
    for(uint8 i = 0; i < 255; i++) {
        uint16 result = le_rt();
        if(result != valor) {
            return false;
        }

        nop();
        nop();
    }

    return true;
}

void testa_chave_de_dados_via_bs() {
    vdb_leitura();
    desativa_sai_rd();

    ativa_sai_bs();
    ativa_esc_t();
    ativa_sai_bt();

    ativa_z_comp();
    mais_um(0);
```

```
    uint16 valor_esperado = 0x0001;

    for(uint8 i = 0; i <= 16; i++) {
        uint16 result;

        while((result = le_rt()) != valor_esperado) {
            char buff[8];
            sprintf(buff, "%04x", valor_esperado);
            Serial.print("Insira o valor 0x");
            Serial.print(buff);
            Serial.print(" nas chaves de dados (valor lido 0x");

            sprintf(buff, "%04x", result);
            Serial.print(buff);
            Serial.println(")");
        }

        if(!checa_estabilidade_rt(valor_esperado)) {
            Serial.println("Falha de estabilidade");
            goto END;
        }

        valor_esperado = (valor_esperado << 1) & 0xffff;
    }

END:
    desativa_sai_bs();
    desativa_sai_bs();
    desativa_esc_t();
    desativa_sai_bt();

    desativa_z_comp();

}

void testa_chaves() {
    testa_chave_enderecos();

    testa_chave_de_dados_via_bd();

    testa_chave_de_dados_via_bs();
}

void incrementa_rp() {
    vdb_leitura();

    ult_c_um();
```

```
    ativa_z_comp();
    mais_um(0);

    ativa_sai_rp();
    ativa_sai_re();

    ativa_esc_t();
    ativa_sai_bt();

    uint16 rp_atual = le_rt();

    ativa_esc_rp();

    mais_um(1);

    while(true) {
        mais_um(0);
        desativa_esc_rp();
        nop();
        clock();
        uint16 rp_anterior = le_vdb();
        ativa_esc_rp();
        mais_um(1);

        clock();
        uint16 result = le_vdb() & 0x3fff;
        uint16 veb_atual = le_veb();

        rp_atual = (rp_atual + 1) & 0x3fff;

        if(result != rp_atual) {
            char buff[8];
            sprintf(buff, "%04x", result);
            Serial.print("RP: 0x");
            Serial.print(buff);

            Serial.print(" (deveria ser 0x");
            sprintf(buff, "%04x", rp_atual);
            Serial.print(buff);

            Serial.print(" (vbe= 0x");
            sprintf(buff, "%04x", veb_atual);
            Serial.print(buff);

            Serial.print(" (rp_ant= 0x");
            sprintf(buff, "%04x", rp_anterior);
            Serial.print(buff);
```

```
            Serial.println("))");
            break;
        }

        if(rp_atual == 0) { break; }
    }

END:
    desativa_sai_rp();
    desativa_esc_rp();
    desativa_sai_re();

    desativa_esc_t();
    desativa_sai_bt();
}

void testa_rd() {
    ativa_z_comp();

    desativa_sai_bt();

    uint16 counter = 0;
    while(true) {
        uint8 coin = rand() % 2;

        if(coin) {
            mais_um(1);
        } else {
            mais_um(0);
        }

        ult_c_um();
        vdb_escrita();

        escreve_vdb(counter);
        clock();

        ult_c_zero();
        nop();

        ativa_esc_t();
        clock();

        desativa_esc_t();
        ult_c_um();
        vdb_leitura();
```

```
        ativa_sai_bt();
        nop();

        uint16 result = le_vdb();
        uint16 expected = counter + coin;

        desativa_sai_bt();


        if(result != expected) {
            Serial.print(expected);
            Serial.print(" != ");
            Serial.println(result);
        }

        counter++;

        if(counter == 0) { break; }
    }
}

void testa_rt() {
    uint8 error = 0;
    uint16 no_error_counter = 0;
    uint16 counter = 3;
    while(true) {
        ativa_z_comp();
        desativa_sai_bt();

        mais_um(0);

        ult_c_um();
        vdb_escrita();

        escreve_vdb(~counter);
        clock();

        ult_c_zero();
        nop();

        escreve_vdb(0);

        ativa_esc_t();

        clock();
```

```
            ativa_esc_t();
            desativa_z_comp();
            nop();

            clock();

            desativa_esc_t();
            ult_c_um();
            vdb_leitura();

            ativa_sai_bt();
            nop();

            uint16 result = le_vdb();
            uint16 expected = counter;

            desativa_sai_bt();


            if(result != expected) {
                char buff[8];
                sprintf(buff, "0x%04x", expected);
                Serial.print(buff);
                Serial.print(" != ");
                sprintf(buff, "0x%04x", result);
                Serial.print(buff);
                Serial.print(" (no error for: )");
                Serial.println(no_error_counter);

                no_error_counter = 0;
                error = 1;
            } else {
                no_error_counter++;
            }

            counter++;

            if(counter == 0) { break; }
        }
    }

    void loop() {
        testa_chaves();
        incrementa_rp();
        testa_rd();
        testa_rt();
    }
```

Código A.6 – Placa 3

```
// --- Variable types defined by length ---
typedef char int8;
typedef byte uint8;
typedef short int16;
typedef unsigned short uint16;
typedef long int32;
typedef unsigned long uint32;

// Vetor de leitura:
// (Desvia, EscM, UltC, EscP, EscT, EscC, ZComp, Mais1, EntRlg, Rlg\,
// Rlg1, Rlg2, Rlg3, BotParteAtivo, BotParteAtivo\, VaiPartir, VaiPartir
   ,
// VaiParar, EscMem, SaiBD, SaiBE, SaiBS, SaiRE, RlgRI, RlgRC).

// Vetor de escrita:
// (I0, I1, RC, Para, Est1, Parte, MudaP, MudaT, BotParte, Partiu,
// IntPara, BotEscMem, EntRlg\)

// Distribuicao das portas:
// * Vetor de leitura: PORTC e PORTL e PORTA e PORTB = (Pins[37..30] e
   Pins[49..42]) e (Pins[22..29] e Pins[53..50],Pins[10..13])
// * Vetor de escrita: PORTF e PORTK = Pins[A0..A7] e Pins[A8..A15]

// * Clock  = PORTE4 = Pins[2]  (saida)

#define nop() __asm__ volatile("nop\n\t")

void clock() {
    PORTE |= (1 << 4);
    nop();
    nop();

    PORTE &= ~(1 << 4);
    nop();
    nop();
}

bool read_desvia(uint32 values) {
    return (values & (1L << 0)) != 0;
}

bool read_esc_m(uint32 values) {
    return (values & (1L << 1)) != 0;
}

bool read_ult_c(uint32 values) {
```

```
    bool result = (values & (1L << 2)) != 0;
    return result;
}

bool read_esc_p(uint32 values) {
    return (values & (1L << 3)) != 0;
}

bool read_esc_t(uint32 values) {
    return (values & (1L << 4)) != 0;
}

bool read_esc_c(uint32 values) {
    return (values & (1L << 5)) != 0;
}

bool read_z_comp(uint32 values) {
    return (values & (1L << 6)) != 0;
}

bool read_mais_1(uint32 values) {
    return (values & (1L << 7)) != 0;
}

bool read_ent_rlg(uint32 values) {
    return (values & (1L << 8)) != 0;
}

bool read_rlg_(uint32 values) {
    return (values & (1L << 9)) != 0;
}

bool read_rlg1(uint32 values) {
    return (values & (1L << 10)) != 0;
}

bool read_rlg2(uint32 values) {
    return (values & (1L << 11)) != 0;
}

bool read_rlg3(uint32 values) {
    return (values & (1L << 12)) != 0;
}

bool read_bot_parte_ativo(uint32 values) {
    return (values & (1L << 13)) != 0;
}
```

```
bool read_bot_parte_ativo_(uint32 values) {
    return (values & (1L << 14)) != 0;
}

bool read_vai_partir(uint32 values) {
    return (values & (1L << 15)) != 0;
}

bool read_vai_parar(uint32 values) {
    return (values & (1L << 16)) != 0;
}

bool read_esc_mem(uint32 values) {
    return (values & (1L << 17)) != 0;
}

bool read_sai_bd(uint32 values) {
    bool result =  (values & (1L << 18)) != 0;
    return result;
}

bool read_sai_be(uint32 values) {
    return (values & (1L << 19)) != 0;
}

bool read_sai_bs(uint32 values) {
    return (values & (1L << 20)) != 0;
}

bool read_sai_re(uint32 values) {
    return (values & (1L << 21)) != 0;
}

bool read_sai_rp(uint32 values) {
    return (values & (1L << 22)) != 0;
}

bool read_rlg_ri(uint32 values) {
    return (values & (1L << 23)) != 0;
}

bool read_rlg_rc(uint32 values) {
    return (values & (1L << 24)) != 0;
}

bool get_i0(uint16 inputs) {
```

```
    return (inputs & (1 << 0)) != 0;
}

bool get_i1(uint16 inputs) {
    return (inputs & (1 << 1)) != 0;
}

bool get_rc(uint16 inputs) {
    return (inputs & (1 << 2)) != 0;
}

bool get_para(uint16 inputs) {
    return (inputs & (1 << 3)) != 0;
}

bool get_est1(uint16 inputs) {
    return (inputs & (1 << 4)) != 0;
}

bool get_parte(uint16 inputs) {
    return (inputs & (1 << 5)) != 0;
}

bool get_muda_p(uint16 inputs) {
    return (inputs & (1 << 6)) != 0;
}

bool get_muda_t(uint16 inputs) {
    return (inputs & (1 << 7)) != 0;
}

bool get_bot_parte(uint16 inputs) {
    return (inputs & (1 << 8)) != 0;
}

bool get_partiu(uint16 inputs) {
    return (inputs & (1 << 9)) != 0;
}

bool get_int_para(uint16 inputs) {
    return (inputs & (1 << 10)) != 0;
}

bool get_bot_esc_mem(uint16 inputs) {
    return (inputs & (1 << 11)) != 0;
}
```

```
bool get_ent_rlg_(uint16 inputs) {
    return (inputs & (1 << 12)) != 0;
}

bool calculate_desvia(uint16 inputs) {
    return (get_i1(inputs) &&
            get_i0(inputs) &&
            get_rc(inputs) &&
            (!get_para(inputs)));
}

bool calculate_esc_m(uint16 inputs) {
    return !((!get_i1(inputs)) &&
             get_i0(inputs) &&
             (!get_est1(inputs)) &&
             (!get_para(inputs)));
}

bool calculate_ult_c(uint16 inputs) {
    bool inner_nor = !((!get_est1(inputs)) || get_para(inputs));

    bool result =  !(calculate_desvia(inputs) || inner_nor || get_parte(
       inputs));

    return result;
}

bool calculate_esc_p(uint16 inputs) {
    return !(get_muda_p(inputs) ||
             (!(get_est1(inputs) ||
                get_para(inputs))));
}

bool calculate_esc_t(uint16 inputs) {
    return !(get_muda_t(inputs) ||
             (!(get_i0(inputs) ||
                (!get_est1(inputs)) ||
                get_para(inputs))));
}

bool calculate_esc_c(uint16 inputs) {
    return !((!get_i1(inputs)) ||
             get_i0(inputs) ||
             get_para(inputs));
}

bool calculate_z_comp(uint16 inputs) {
```

```
    return !(get_i1(inputs) &&
             (!get_i0(inputs)) &&
             get_est1(inputs) &&
             (!get_para(inputs)));
}

bool calculate_mais_1(uint16 inputs) {
    return !((!(get_i1(inputs) ||
                (!get_est1(inputs)))) ||
             calculate_desvia(inputs) ||
             get_para(inputs));
}

bool calculate_ent_rlg(uint16 inputs) {
    return !get_ent_rlg_(inputs);
}

bool calculate_rlg_(uint16 inputs) {
    return !calculate_ent_rlg(inputs);
}

bool calculate_rlg1(uint16 inputs) {
    return !calculate_rlg_(inputs);
}

bool calculate_rlg2(uint16 inputs) {
    return !calculate_rlg_(inputs);
}

bool calculate_rlg3(uint16 inputs) {
    return !calculate_rlg_(inputs);
}

bool calculate_bot_parte_ativo(uint16 inputs) {
    return !get_bot_parte(inputs);
}

bool calculate_bot_parte_ativo_(uint16 inputs) {
    return !calculate_bot_parte_ativo(inputs);
}

bool calculate_vai_partir(uint16 inputs) {
    return (calculate_bot_parte_ativo(inputs) &&
            (!get_parte(inputs)) &&
            (!get_partiu(inputs)) &&
            get_para(inputs));
}
```

```
bool calculate_vai_parar(uint16 inputs) {
    return !((calculate_ult_c(inputs)) ||
             get_para(inputs) ||
             (!get_int_para(inputs)));
}

bool calculate_esc_mem(uint16 inputs) {
    return !((!((calculate_esc_m(inputs)) &&
                (!(get_bot_esc_mem(inputs))))) &&
             (!(calculate_ent_rlg(inputs) &&
                calculate_rlg_(inputs))));
}

bool calculate_sai_bd(uint16 inputs) {
    return !(get_para(inputs) &&
             get_bot_esc_mem(inputs));
}

bool calculate_sai_be(uint16 inputs) {
    return !(get_para(inputs) &&
             (!get_parte(inputs)));
}

bool calculate_sai_bs(uint16 inputs) {
    bool result = !(get_muda_p(inputs) || get_muda_t(inputs));
    return result;
}

bool calculate_sai_re(uint16 inputs) {
    return !((!get_para(inputs)) ||
             get_parte(inputs));
}

bool calculate_sai_rp(uint16 inputs) {
    bool inner_nand2 = !(get_est1(inputs) && (!get_para(inputs)));

    bool result =  !((!calculate_desvia(inputs)) &&
                     inner_nand2 &&
                     (!get_parte(inputs)) &&
                     (calculate_sai_bs(inputs)));

    return result;
}

bool calculate_rlg_ri(uint16 inputs) {
    return !(!(calculate_ult_c(inputs)) &&
```

```
            calculate_rlg_(inputs));
}

bool calculate_rlg_rc(uint16 inputs) {
    return !(calculate_esc_c(inputs) &&
            calculate_rlg_(inputs));
}


void test(bool (*testing) (uint32), bool (*reference) (uint16), uint32
   read, uint16 input, char* name) {
    bool tested = testing(read);
    bool ref = reference(input);
    if(tested != ref) {
        Serial.print("Error: ");
        Serial.print(name);
        Serial.print(" => val: ");
        Serial.print(tested);
        Serial.print(" ref: ");
        Serial.println(ref);

        char buff[16];
        Serial.print("\tRead: ");
        sprintf(buff, "0x%08lx", read);
        Serial.print(buff);
        Serial.print(" Input: ");
        sprintf(buff, "0x%04x", input);
        Serial.println(buff);
    }
}

#define testa(to_test, input, read) test(read_##to_test, calculate_##
   to_test, read, input, (char*) #to_test)

void direciona_portas() {
    DDRF = 0xff;
    DDRK = 0xff;

    DDRC = 0x00;
    DDRL = 0x00;
    DDRA = 0x00;
    DDRB = 0x00;

    DDRE |= (1 << 4);
}

void escreve_input(uint16 valor) {
```

```
    uint16 inversion_mask = (1 << 6) | (1 << 7) | (1 << 10) | (1 << 11);
    valor = valor ^ inversion_mask;

    uint8 lower_byte = valor;
    uint8 upper_byte = valor >> 8;

    PORTF = lower_byte;
    PORTK = upper_byte;

    clock();
}

uint32 le_output() {
    nop();

    uint32 lower_byte = PINC;
    uint32 middle_lower_byte = PINL;
    uint32 middle_upper_byte = PINA;
    uint32 upper_byte = PINB & 0x01;

    uint32 result = (upper_byte << 24) | (middle_upper_byte << 16) | (
       middle_lower_byte << 8) | (lower_byte);
    return result;
}

void setup() {
    direciona_portas();
    Serial.begin(9600);
}

static uint16 n_testes = 0;
void loop() {
    Serial.print("Teste num:");
    Serial.println(n_testes++);

    for(uint16 input = 0; input < (1 << 13); input++) {
        escreve_input(input);
        uint32 saida = le_output();

        testa(desvia, input, saida);
        testa(esc_m, input, saida);
        testa(ult_c, input, saida);
        testa(esc_p, input, saida);
        testa(esc_t, input, saida);
        testa(esc_c, input, saida);
        testa(z_comp, input, saida);
        testa(mais_1, input, saida);
```

```
        testa(ent_rlg, input, saida);
        testa(rlg_, input, saida);
        testa(rlg1, input, saida);
        testa(rlg2, input, saida);
        testa(rlg3, input, saida);
        testa(bot_parte_ativo, input, saida);
        testa(bot_parte_ativo_, input, saida);
        testa(vai_partir, input, saida);
        testa(vai_parar, input, saida);
        testa(esc_mem, input, saida);
        testa(sai_bd, input, saida);
        testa(sai_be, input, saida);
        testa(sai_bs, input, saida);
        testa(sai_re, input, saida);
        testa(sai_rp, input, saida);
        testa(rlg_ri, input, saida);
        testa(rlg_rc, input, saida);
    }
}
```