UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

CENTRO DE TECNOLOGIA - CT

CIRCUITOS DIGITAIS

ELEMENTO DE MEMÓRIA FIFO

ELE2715 - Grupo 02 - Implementação - Problema 06

Igor Michael Araujo de Macedo

Isaac de Lyra Junior

João Matheus Bernardo Resende

Lucas Batista da Fonseca

Sthefania Fernandes Silva

Natal, 18 de abril de 2021

RESUMO

O aumento da complexidade dos circuitos exige formas diferentes de tratar os dados inseridos, como por exemplo, dar importância a ordem em que eles são inseridos e retirados do sistema projetado. Diante disso, o seguinte relatório tem como objetivo implementar um circuito digital que reproduz o comportamento de um elemento de memória *FIFO (First In, First Out)*. O circuito projetado tem como entradas 3 botões síncronos responsáveis por realizar as operações de escrita, leitura e limpeza dos dados; há também uma entrada (*w_data)* responsável pela inserção do dado que será armazenado, assim como uma saída (*r_data*) para remoção dos valores guardados; por fim, ele conta, também, com 2 saídas para representar se a memória está cheia ou vazia. Para o desenvolvimento do projeto, foram necessários conceitos sobre circuitos sequenciais, máquinas de estado finitos, projetos RTL, banco de registadores e elementos de memória. Foram utilizados, para a implementação e simulação, os *softwares LogiSim* e *Modelsim*. Os resultados das simulações foram bem sucedidas, evidenciado que o projeto possui coerência.

Palavras-chave: Máquinas de Estados Finitos. Projeto RTL. Banco de Registradores. FIFO. VHDL.

SUMÁRIO

1 INTRODUÇÃO

Uma das formas mais comuns de organizar pessoas que buscam acessar o mesmo serviço é o uso de filas. O arranjo é feito de forma que o primeiro a chegar assume a primeira posição e os demais são introduzidos na posição posterior, conforme mostra a Figura 1.

Figura 1- Exemplo de fila.

Fonte: FreePik.

De maneira análoga, em sistemas digitais há algumas situações em que se faz necessário um armazenamento ordenado, onde a leitura do dado armazenado o remove da lista. Quando o primeiro item inserido na memória é o primeiro a ser lido - e removido - trata-se uma fila FIFO (*First-In, First-Out*, que em tradução livre significa o primeiro que entra, é o primeiro a sair). A FIFO pode ser implementada usando uma memória, que dependendo do tamanho da fila, pode ser um banco de registradores ou uma RAM (*Random Access Memory* - Memória de acesso aleatório) (VAHID, 2008).

Um banco de registradores (*register-file*) consiste em um bloco operacional que permite o acesso a um conjunto de M registradores que possuem uma largura de N bits. Este componente permite o armazenamento de bits evitando os erros de congestionamento (acúmulo de fios em um espaço pequeno) e *fanout* (limite de ramificações, nos fios, que podem ser realizadas até que a corrente em cada fio seja pequena demais para o circuito funcionar de forma eficiente) (VAHID, 2008).

O componente funciona conforme mostrado na Figura 2, nela pode-se observar um banco de registradores 16x32 (16 registradores com 32 bits de largura cada). O ato de escrever em um banco de registradores traduz-se em inserir o valor em *w_data* e clicar em *w_en*, visando habilitar o registro. No entanto, o acesso ao banco é feito de maneira ordenada, logo, é preciso saber o endereço na qual o dado será armazenado, para isso há o *w_addr,* uma entrada de 4 bits que contém o endereço do registrador o qual terá o dado *w_data* armazenado (VAHID, 2008).

Figura 2 - Exemplo de um banco de registradores.

Fonte: Adaptado de VAHID (2008).

A memória RAM é, basicamente, um banco de registradores, isso se dá pois ambos possuem funcionalidades parecidas. No entanto, as principais diferenças entre eles são: 1) o tamanho M: circuito com memórias menores (entre 4 e 512 dados) utilizam banco de registradores, já as memórias maiores usam RAM; e 2) tamanho do circuito: as memórias RAM são mais compactas por não utilizar flip-flops, diferente do banco de registradores.

Para projetar circuitos com tais complexidades há a necessidade de arranjos lógicos que possam abranger tantas funções. Um desses arranjos são os blocos de controle, estes são úteis em implementações de sistemas que necessitam de entradas e saídas para controlar um determinado comportamento, como por exemplo, as mudanças de estados (VAHID, 2008).

Nesse sentido, o bloco de controle atua em junção com outro bloco construtivo que possui as entradas e saídas de dados do sistema, em particular, esse bloco deve conter registradores para armazenar e unidades funcionais para operar esses dados. Esse arranjo é conhecido como componente do nível de transferência entre registradores, do inglês *Register-Transfer-Level* (RTL). Um circuito composto por tais componentes é nomeado bloco operacional (VAHID, 2008).

A combinação de um bloco operacional com um bloco de controle gera um processador. O método mais comum de construir um é usando o projeto RTL. Nele são especificados os registradores do circuito, as possíveis transferências e operações que serão feitas com os dados de entrada; saída e dos registradores, além de definir o controle que coordena quando e como transferir e operar dados (VAHID, 2008).

Diante do que foi exposto, o presente relatório irá definir o projeto de uma FIFO utilizando a metodologia de projetos RTL.

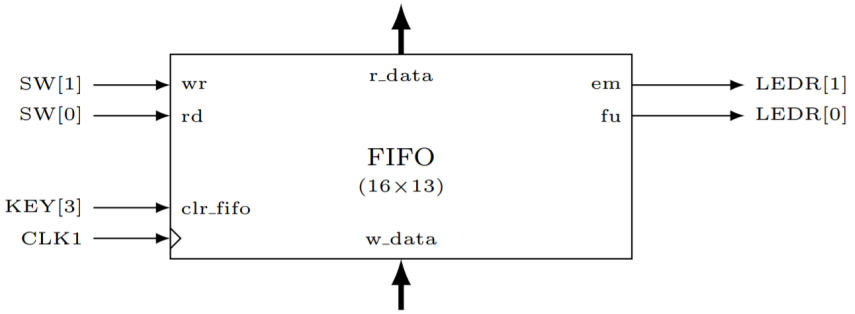
2 DESENVOLVIMENTO

O problema relatado consiste em criar o projeto de uma FIFO, a qual terá a capacidade de realizar o registro, a escrita e a leitura de dados inseridos pelo usuário.

O funcionamento desse banco de dados é determinado por 4 entradas: a entrada denominada *w_data* é voltada para inserir o valor do dado que será armazenado na FIFO, para permitir o armazenamento desse valor, *wr* (*write*) deve está em nível lógico alto e um pulso da entrada *clk* deve ser dado. Para ler o valor guardado há a entrada *rd* (*read*), após um pulso de *clk*, o valor lido será removido da máquina através da saída *r_data*.

A máquina também é dotada pela entrada *clr* (*clear*), a qual terá como funcionalidade fazer uma limpeza em todas as posições de memória da FIFO e reiniciar os valores dos contadores internos. Além disso, a FIFO possui duas saídas *em* e *fu*, onde a primeira significa vazio e a segunda cheio. Logo, quando em=1 não há nenhum dado armazenado na FIFO, já quando fu=1 significa que todas as posições de memória estão ocupadas. A Figura 3 mostra o diagrama de blocos do problema.

Figura 3 - Diagrama de blocos da FIFO.



Fonte: Dados do problema.

2.1 CONSIDERAÇÕES DO PROJETO

O projeto da FIFO foi elaborado com base em algumas condições determinadas pelos projetistas. Primeiramente, não será possível sobrescrever os dados armazenados ou ler a FIFO sem nenhum dado guardado. Logo, quando *fu* estiver em nível lógico alto, o usuário somente poderá ler os dados guardados (*rd*=1) ou apagar todos os dados armazenados (*clr*=1). E quando *em* estiver em nível lógico alto, o usuário somente poderá escrever novos dados (w*r*=1).

Optou-se por todos os botões de entrada (*rd, wr e clr*) serem síncronos, assim, foi necessário definir a precedência de cada um deles. Portanto, *clr* tem maior precedência,

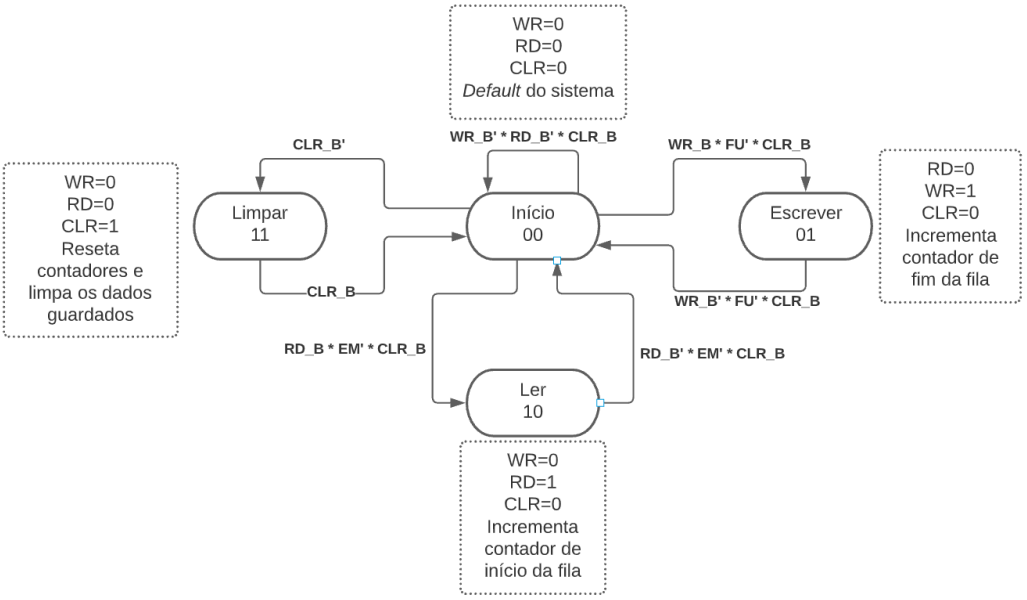
seguido de *rd* e *wr*, respectivamente. Além disso, o circuito foi projetado para receber altos valores de *clock (*32 *hertz)*.

Por fim, para projetar a FIFO, foi seguida a metodologia descrita por VAHID (2008), a qual divide o método de projeto RLT em 4 passos: 1º Descrever o comportamento do circuito (máquina de estados de alto nível); 2º criar bloco operacional; 3º conectar bloco operacional a um bloco de controle e 4º obter a máquina de estados finitos (FMS).

2.2 MÁQUINA DE ESTADOS DE ALTO NÍVEL

Para descrever o comportamento desse sistema foi montada a máquina de alto nível mostrada na Figura 4. Esta conta com 4 estados: Início, Escrever, Ler e Limpar. Por questão de praticidade foi determinado que as entradas serão: *WR_B, RD_B, CLR_B* (B de botão), *FU* e *EM* e as saídas são: *CLR, WR* e *RD*.

Figura 4 - Máquina de alto nível.



Fonte: Elaborado pelos autores.

Primeiramente, a máquina da FIFO foi pensada para que no estado inicial, o sistema fique aguardando o recebimento de um pulso de sinal síncrono vindo de um dos 3 botões de entrada (*CLR_B*, *RD_B* e *WR_B*). Diante disso, o estado "Início" determina o comportamento da máquina operando por *default*, ou seja, quando nenhuma de suas entradas sofreu alteração (nenhum dos botões foi acionado).

A máquina muda para o estado “Escrever” quando o botão *WR_B* é acionado. Nesse estado, a máquina deve fazer uma operação de escrita, ou seja, guardar o valor inserido pelo usuário, assim sua saída *WR* receberá valor lógico alto. Todavia, a restrição para escrita é que a máquina não poderá estar cheia (*FU* = 1 significa que todos os endereços de memória estão ocupados), logo, enquanto *FU*=0 a escrita poderá ser realizada, caso contrário não.

A máquina muda para o estado “Ler” quando o botão *RD_B* é acionado. Nesse estado a máquina deve fazer uma operação de leitura, ou seja, o valor lido será removido da máquina através da saída *R_data* e a saída *RD* da máquina de estados receberá nível lógico alto. A restrição para esse caso é que uma leitura só pode ser realizada quando a máquina já possui algo armazenado em sua memória, ou seja, caso a máquina esteja com sua memória vazia (*EM* = 1) a máquina não deve ir para o estado “Ler”.

Por último, a máquina muda para o estado “Limpar” quando o botão *CLR_B* for acionado. Na Figura 4, é possível perceber que esse estado possui prioridade sobre os outros, pois independente de qualquer outro botão acionado sincronicamente, quando *CLR_B* está em nível lógico baixo o estado setado é “Limpar”. Nesse estado, a máquina deve resetar todas as posições de memória e reiniciar os contadores.

Note que os estados “Ler”, "Escrever” e “Limpar” foram criados para atuar por um pulso de *clock* e depois retornar ao estado “Início”.

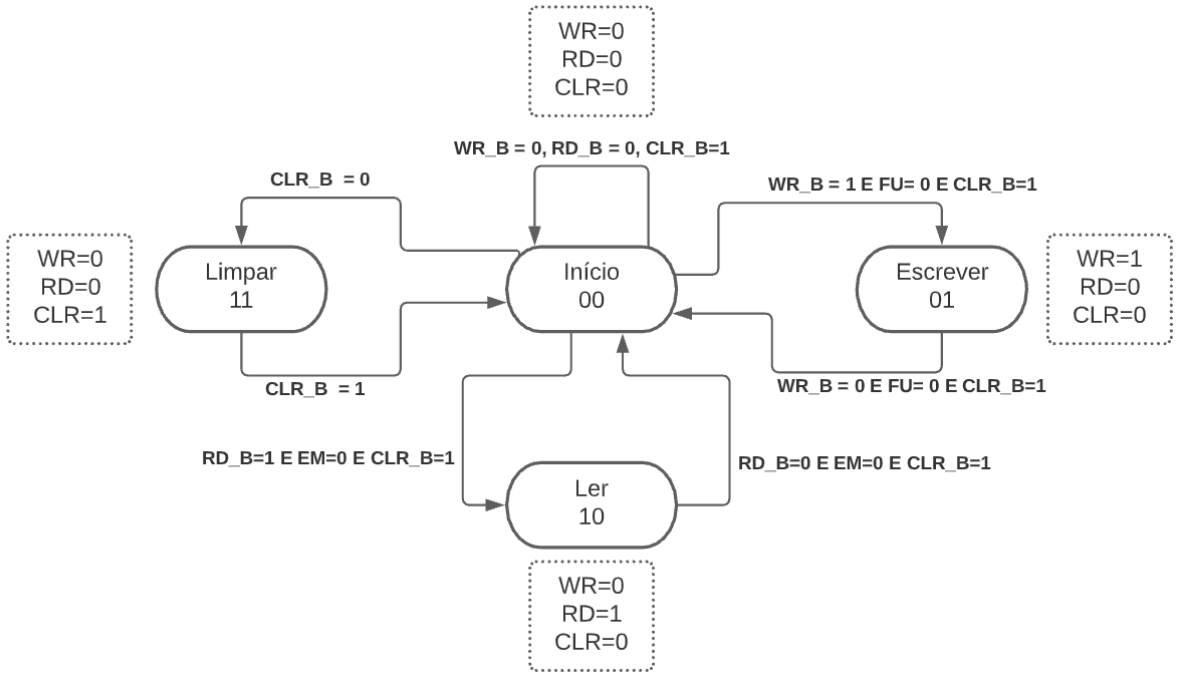
2.3 BLOCO DE CONTROLE E MÁQUINA DE ESTADOS FINITOS

Dando seguimento a construção da máquina, foi criado um bloco de controle; nele está presente a máquina de estados finitos e as operações que a constituem.

Para definir a máquina de estados finitos foi feita a representação do circuito em baixo nível. A máquina de baixo nível se assemelha à de alto nível, a principal diferença entre elas é que na máquina de baixo nível trataremos diretamente com as operações, já na máquina de alto nível foi feita para representar o funcionamento do circuito usando uma nomenclatura mais amigável para o entendimento humano.

Diante disso, foi montada a máquina de baixo nível mostrada na Figura 5.

Figura 5 - Máquina de baixo nível.



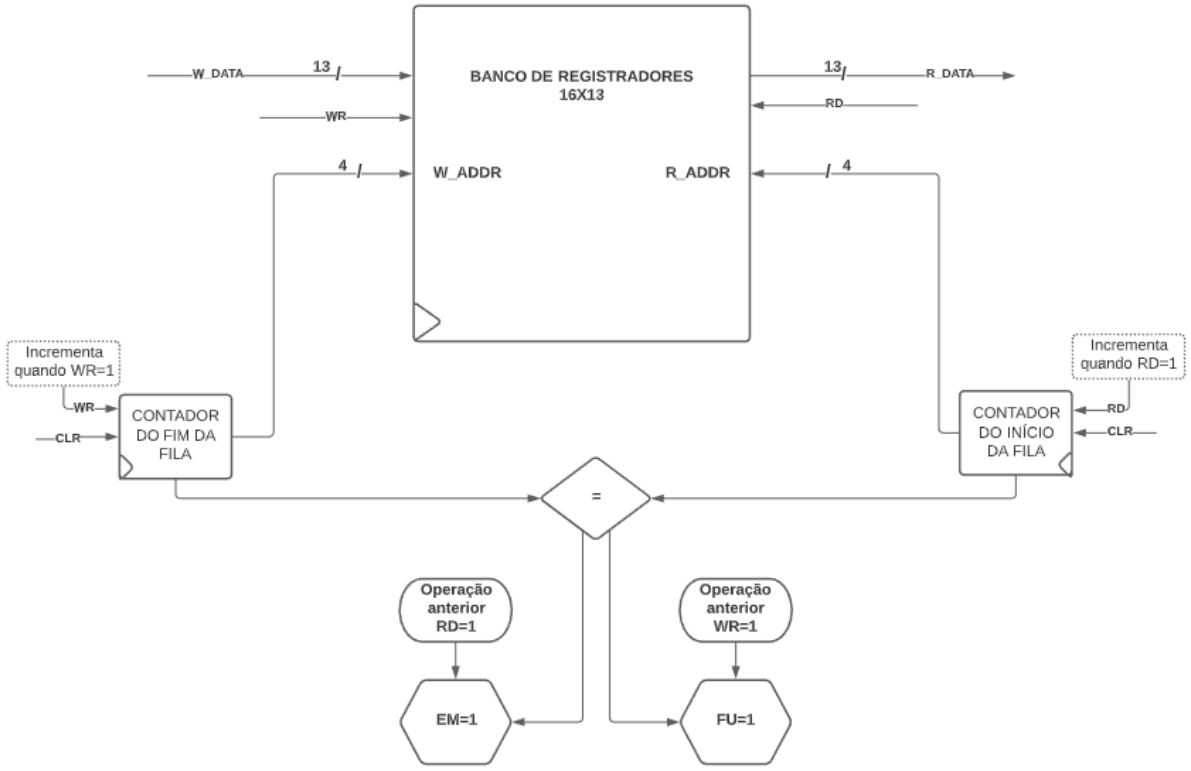
Fonte: Elaborado pelos autores.

A partir do fluxograma da máquina de baixo nível, foi construída a tabela de estados finitos que define todos os casos demonstrados na Figura 5. Essa tabela está presente para visualização no Anexo B.

2.4 BLOCO OPERACIONAL

Para que a máquina funcione da maneira que foi instruída, foi criado um bloco operacional que contém um banco de registro e a lógica de operações de escrita e leitura de dados, inseridos pelo usuário, conforme mostra a Figura 6.

Figura 6 - Bloco operacional



Fonte: Elaborado pelos Autores.

Com base no conceito de banco de registradores, este foi pensado na construção do componente de armazenamento da FIFO. Com o uso de 16 registradores de 13 bits, duas entradas habilitadoras de escrita (*wr*) e leitura (*rd)* do usuário de 1 bit e um acesso ao endereço dos registradores de 4 bits, o banco de registradores idealizado é o mostrado na parte superior da Figura 6. Assim, será possível fazer o armazenamento dos valores de entrada (*W_data*) em cada posição vazia no banco de registradores.

Após o registro dos dados inseridos, o usuário poderá usar a função ler, sendo esta ativada quando a entrada *rd* estiver em nível lógico alto. A leitura começa na primeira posição e remove o dado do banco através de *R_data,* tendo como condição de parada *EM* = 1, ou seja, quando o banco está vazio. Da mesma forma ocorre quando o banco está cheio (*FU* = 1), a diferença é que será mostrado que no banco não existem espaços para serem alocados, logo, nenhuma escrita poderá ser realizada.

Para que o banco compute os dados de entrada de forma ordenada, é acionado um contador que será alimentado a cada vez que *wr* for pressionado e houver um pulso de *clock*, o valor contado será utilizado na entrada *w_addr* do banco de registros. Essa entrada será

responsável por selecionar qual registrador deverá receber o dado introduzido, o contador de escrita foi nomeado “contador de fim da fila”.

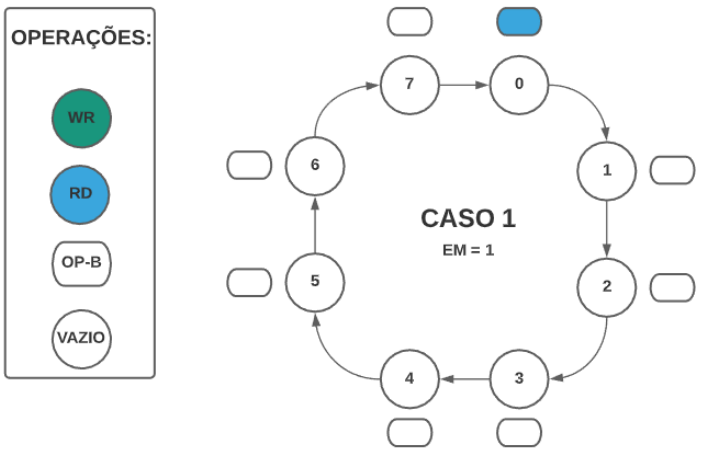
Para ser feita uma leitura é um acionado um novo contador que será alimentado a cada vez que *rd* for pressionado e houver um pulso de *clock*, o valor contado será utilizado na entrada *r_addr* do banco de registros. Essa entrada será responsável por selecionar de qual registrador precisamos ler o dado guardado, o contador de leitura foi nomeado “contador de início da fila”.

Cada liberação de espaço determina um novo fim da fila, logo, a fila criada é circular e o contador de escrita deve ir do valor mínimo ao máximo quantas vezes forem necessárias para preencher todos os espaços. Diante disso, para identificar quando a FIFO está cheia ou vazia é preciso realizar a comparação de ambos os contadores para verificar se o início e o fim da fila são iguais.

Como a fila é circular pode ocorrer do início e do fim serem iguais em duas situações: quando a FIFO está cheia e também quando está vazia. Para contornar isso, sugere-se que além de verificar a igualdade entre os contadores, verifique se a operação realizada anteriormente ao instante em que as filas se encontram iguais é de leitura ou escrita.

Para ilustrar melhor isso, foram explanadas as seguintes situações em que a FIFO pode se encontrar ao longo do seu uso. Na Figura 7, temos a situação em que a quantidade de escritas (*wr)* é igual a quantidade de leituras (*rd*), logo, tudo o que outrora estava armazenado foi removido. Dessa forma, temos que a FIFO está vazia (*em*=1). As operações mostradas na Figura 7 são definidas *wr* (escrita), *rd* (leitura) e op-b (operação anterior). Note que, o que permitiu comprovar que a FIFO está vazia, foi que a última operação realizada foi de leitura.

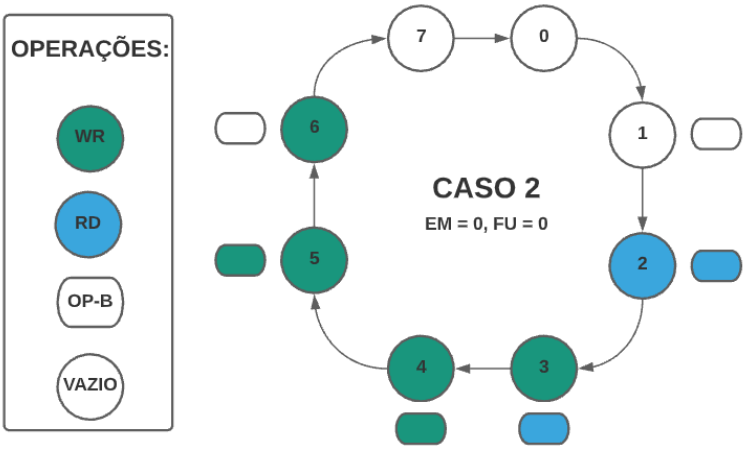
Figura 7 - Quando a quantidade de escritas e leituras são iguais e a operação anterior é uma leitura.



Fonte: Elaborado pelos autores.

No segundo caso, temos a situação quando temos várias escritas e várias leituras, conforme a Figura 8. Quando isso acontece, temos a situação em que nem se está vazio (*em*=0) e nem se está cheio (*fu*=0). Essa situação permite que o fim da fila seja alterado, isso porque o registrador da posição lida volta a ficar disponível para um novo registro, rearranjando o fim da fila. Nesse caso, novos valores podem ser escritos da posição 7 até a condição de parada, que no caso é a posição 2.

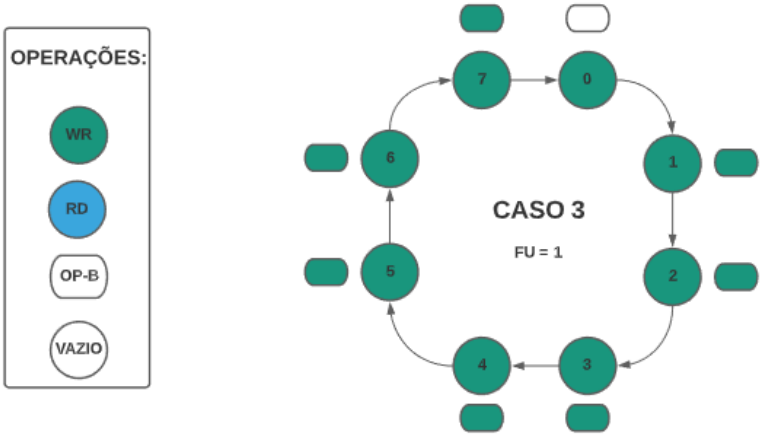
Figura 8 - Caso em que há várias escritas e várias leituras.



Fonte: Elaborado pelos autores.

Já no caso 3, temos a situação de quando é atingido o máximo de escritas e nenhuma leitura, conforme pode ser visto na Figura 9. Nesta configuração, temos que a FIFO está cheia (*fu*=1 e *em*=0), indicando que não pode ser mais escrito em nenhum registrador, enquanto não houver leitura do primeiro registrador da fila ou a limpeza de todos os registradores. Note que, o que permitiu comprovar que a FIFO está cheia, foi que a última operação realizada foi de escrita.

Figura 9 - Caso em que há o número máximo de escrita, mas nenhuma leitura.

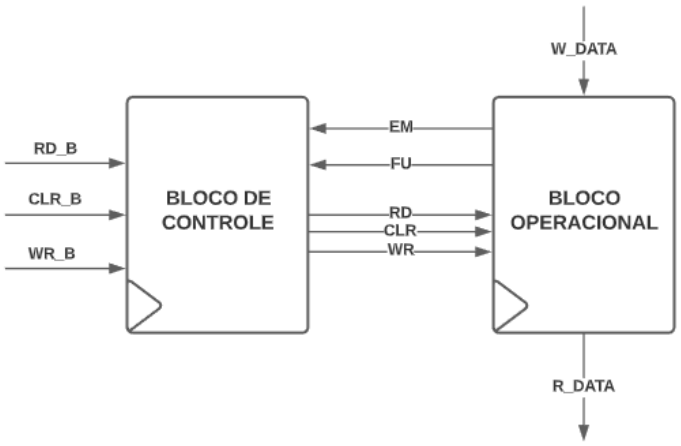


Fonte: Elaborado pelos autores.

Por padrão, o banco de registradores deve iniciar vazio, ou seja, *em*=1. Sendo assim, uma lógica que também preveja essa condição deve ser elaborada.

Por fim, a junção do bloco de controle com o bloco operacional pode ser observada na Figura 10.

Figura 10 - Bloco operacional e bloco de controle..



Fonte: Elaborado pelos autores.

2.5 CORREÇÕES NECESSÁRIAS

Apesar do projeto, no geral, explicitar grande parte do que deve ser implementado, foi necessário definir algumas operações. Logo, neste tópico é definido como será verificada a operação anterior, elemento fundamental para definição de cheio e vazio da fila, e também é definida a máquina de estados que descreve os botões síncronos (*WR_B, RD_B, CLR_B*).

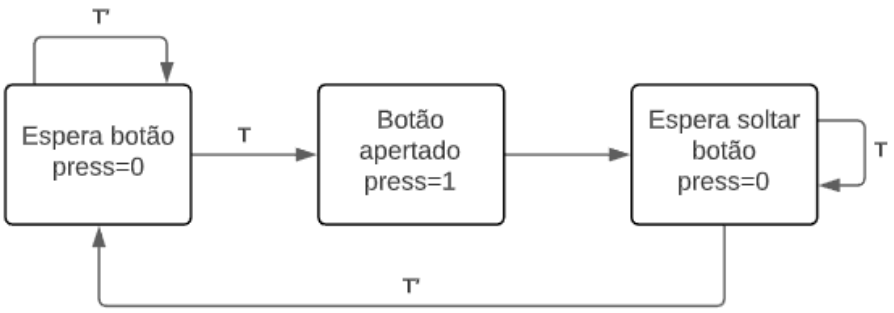
2.5.1 Circuito de verificação da operação anterior

Para verificar qual operação foi realizada antes do contador de início e fim possuírem valor contado igual, foi pensado em registrar quando o usuário da FIFO fez uma leitura ou escrita. Assim, caso uma leitura tenha sido feita o registrador guarda 0, caso contrário (quando uma escrita é feita) guarda 1.

2.5.2 Botão síncrono

Para a criação dos botões síncronos (*WR_B, RD_B, CLR_B*) foi feita a máquina de estados mostrada na Figura 11. Esta máquina irá garantir que, quando pressionado o botão, *press* terá nível lógico alto durante 1 *clock*, solucionando o problema de que, por exemplo, o usuário aperte o botão e fique pressionado por um certo tempo.

Figura 11 - Máquina de estados do botão síncrono.



Fonte: Elaborado pelos autores.

Em posse da máquina de estados de baixo nível, foi possível construir a tabela de estados que define o problema e identificar a lógica combinacional dos estados futuros e da saída. Nos Quadros 1 e 2 pode-se observar a tabela de estados e as expressões lógicas do botão síncrono, respectivamente.

Quadro 1 - Tabela de estados do botão síncrono.

ESTADOS	ESTADO ATUAL		BOTÃO	ESTADO FUTURO		SAÍDA
	Q1	Q0	T	D1	D0	PRESS
Espera pressionar botão	0	0	0	0	0	0
	0	0	1	0	1	0
Botão apertado	0	1	x	1	0	1
Espera soltar botão	1	0	0	0	0	0
	1	0	1	1	0	0

Fonte: Elaborado pelos autores.

Quadro 2 - Lógica combinacional do botão síncrono.

EXPRESSÕES LÓGICAS
D1 = Q1' Q0 + Q1 Q0' T
D0 = Q1' Q0' T
PRESS = Q1' Q0

Fonte: Elaborado pelos autores.

3 RESULTADOS

Após a realização das correções, tornou-se viável implementar o circuito nos programas de simulação. Para a simulação com visualização dos componentes foi utilizado o *LogiSim*, devido a sua praticidade para criar máquinas de estado. Já para simulação em VHDL foi utilizado o *ModelSim.*

3.1 IMPLEMENTAÇÃO NO LOGISIM

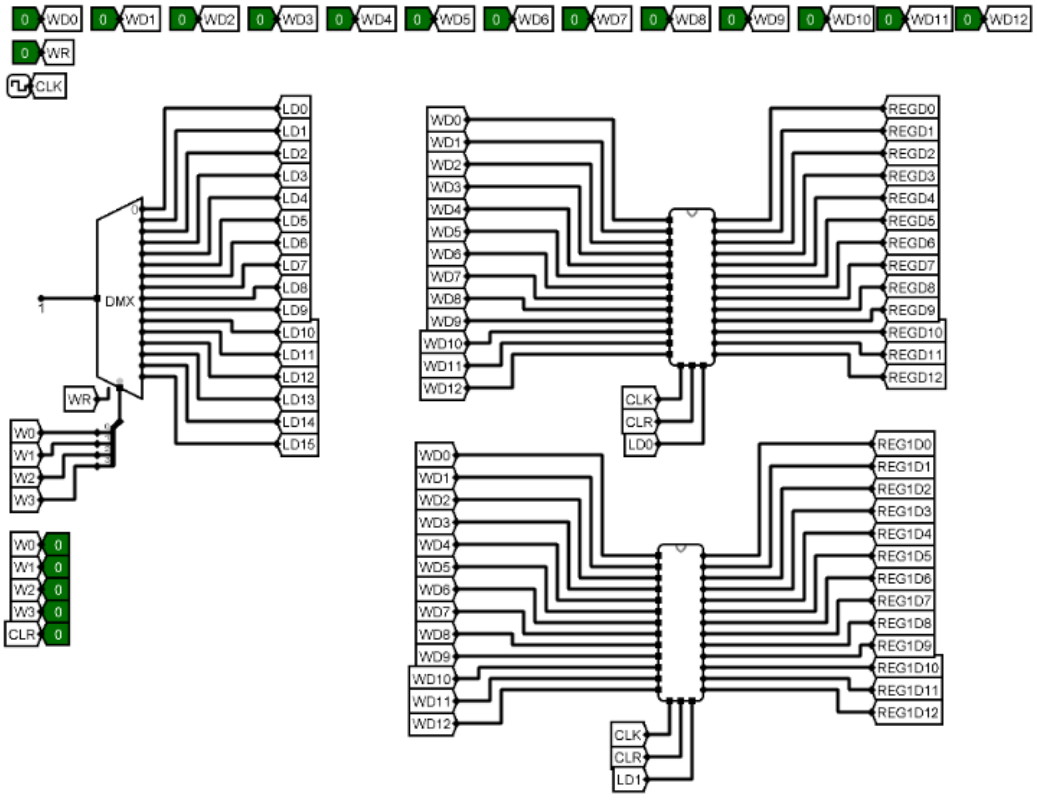
O software utilizado para criação do esquemático e simulações foi o Logisim.

3.1.1 Banco de registradores

A implementação do banco de registadores 16x13 baseou-se na utilização de um demultiplexador 1x16, junto com conjunto de 16 registradores de 13 bits e um multiplexador 16x1. O demux possui uma entrada enable que quando esta recebe 0 lógico, todas as saídas irão receber 0 lógico, independente dos valores da chave seletora, essa entrada enable será a variável WR que é o botão de permitir escrita. Esse DEMUX vai receber 1 lógica, constante, na entrada de dados, e os 4 bits da chave seletora são o valor do contador de 4 bits da escrita, por fim, as 16 saídas são as entradas das portas de permissão de registro dos 16 registradores.

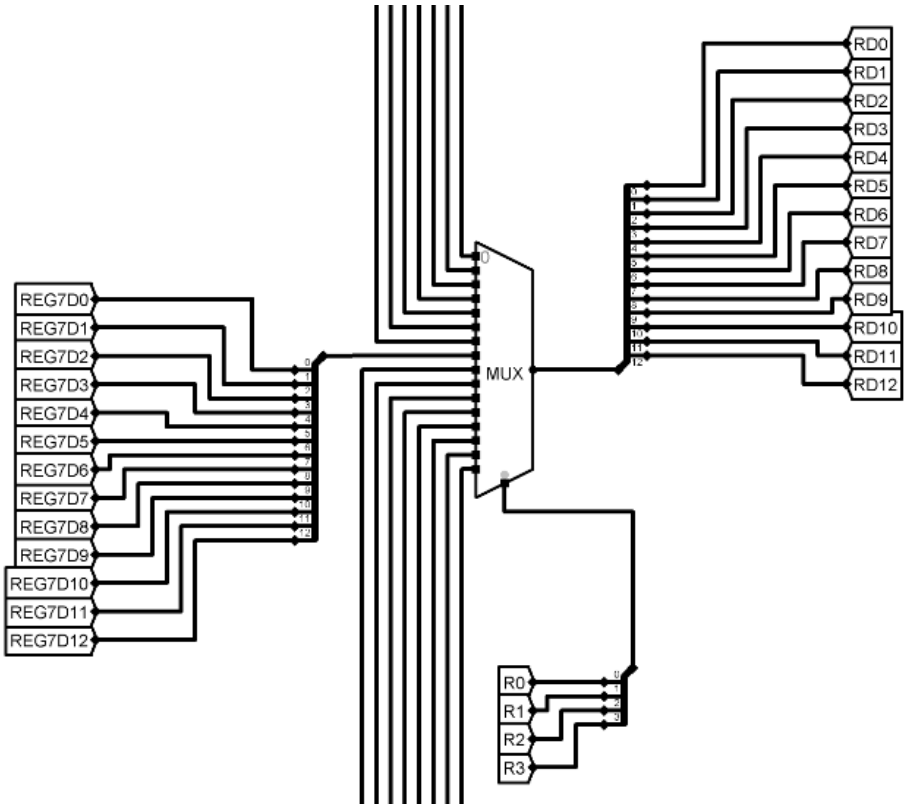
Os registradores, recebem os 13 bits da entrada *W_data*, a entrada CLR, o CLK e uma da saída do DEMUX. O MUX recebe todas as saídas dos registradores e os 4 bits de seleção são os 4 bits do contador e a saída do mux é a saída *R_data*.

Figura 12 - Lógica combinacional da FIFO.



Fonte: Elaborado pelos autores.

Figura 13 - Lógica combinacional da FIFO.



Fonte: Elaborado pelos autores.

3.1.2 Bloco de controle

Da tabela de estados, exibida no Anexo B, foi possível extrair as expressões lógicas mostradas no Quadro 3.

Quadro 3 - Lógica combinacional da FIFO.

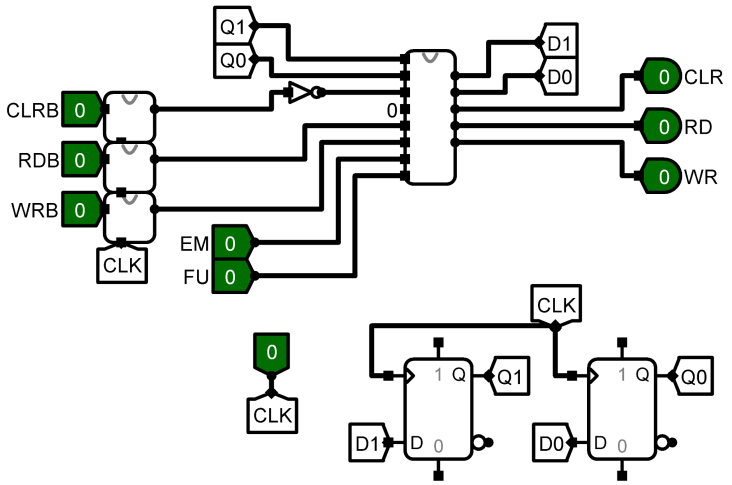
EXPRESSÕES LÓGICAS
D1 = Q1’ Q0’ CLR_B’ + Q1’ Q0’ RD_B WR_B’ EM’
D0 = Q1’ Q0’ CLR_B’ + Q1’ Q0’ RD_B WR_B’ EM’
CLR = Q1 Q0
RD = Q1 Q0’ FU + Q1 Q0’ EM + Q1 Q0’ WR_B + Q1 Q0’ RD_B + Q1 Q0’ CLR_B
WR= Q1’ Q0

Fonte: Elaborado pelos autores.

Em posse das expressões lógicas foi feita a lógica combinacional da máquina de estados da FIFO, esta pode ser visualizada no Anexo C. A lógica foi colocada em um

componente, o qual pode ser visualizado na Figura 14. Além disso, a figura conta com o registrador de estados (2 bits).

Figura 14 - Bloco de controle.



Fonte: Elaborado pelos autores.

3.1.3 Bloco operacional

No bloco operacional, além do banco de registradores (explicado no tópico 3.1.1), há o circuito que verifica a operação anterior, um comparador de igualdade e dois contadores para verificar o início e fim da fila.

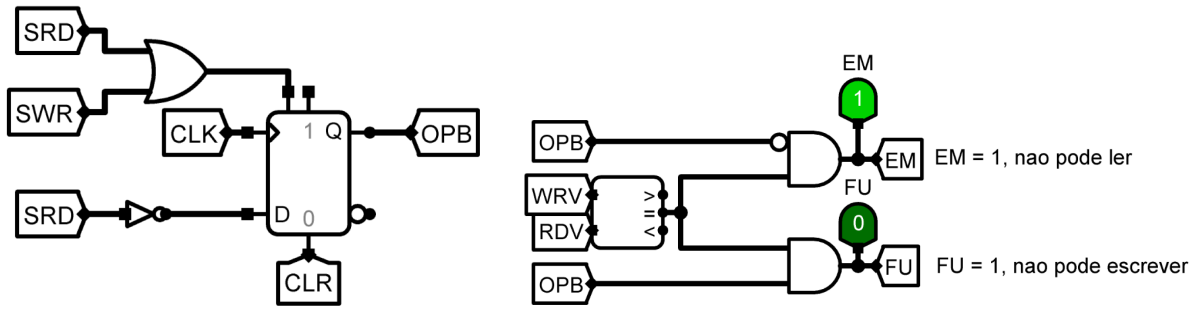
Os contadores definem o endereço do banco de registros, o contador de fim é o contador que é incrementado toda vez que uma escrita é solicitada, já o contador de início é o contador que é incrementado toda vez que uma leitura é solicitada. As saídas dos contadores de início e fim são colocadas nas entradas *W_addr* e *R_addr* do banco de registros, respectivamente.

O comparador verifica se os contadores de início e fim são iguais, isso é feito para averiguar se a FIFO está cheia ou vazia. No entanto, somente a comparação é insuficiente para definir quando o armazenamento está cheio ou vazio, isso porque, a fila é circular logo, o início e o fim são mutáveis. Visando detectar em quais momentos a FIFO está cheia ou vazia é visualizada a operação anterior ao momento em que os contadores ficam iguais. Nesse caso, se a operação anterior foi uma leitura, a FIFO está vazia e se for escrita a FIFO está cheia.

Para isso foi criado o registrador mostrado na Figura 15a, caso uma leitura tenha sido feita o registrador guarda 0, caso uma escrita tenha sido feita ele guarda 1. A saída deste registrador é usada como entrada de uma porta lógica AND (Figura 15b), a qual já possui

como entrada o resultado da comparação de igualdade. Dessa forma, foi determinado quando a FIFO está cheia ou vazia.

Figura 15 - a) Registrador da operação anterior b) Lógica para verificar quando a FIFO está vazia ou cheia.

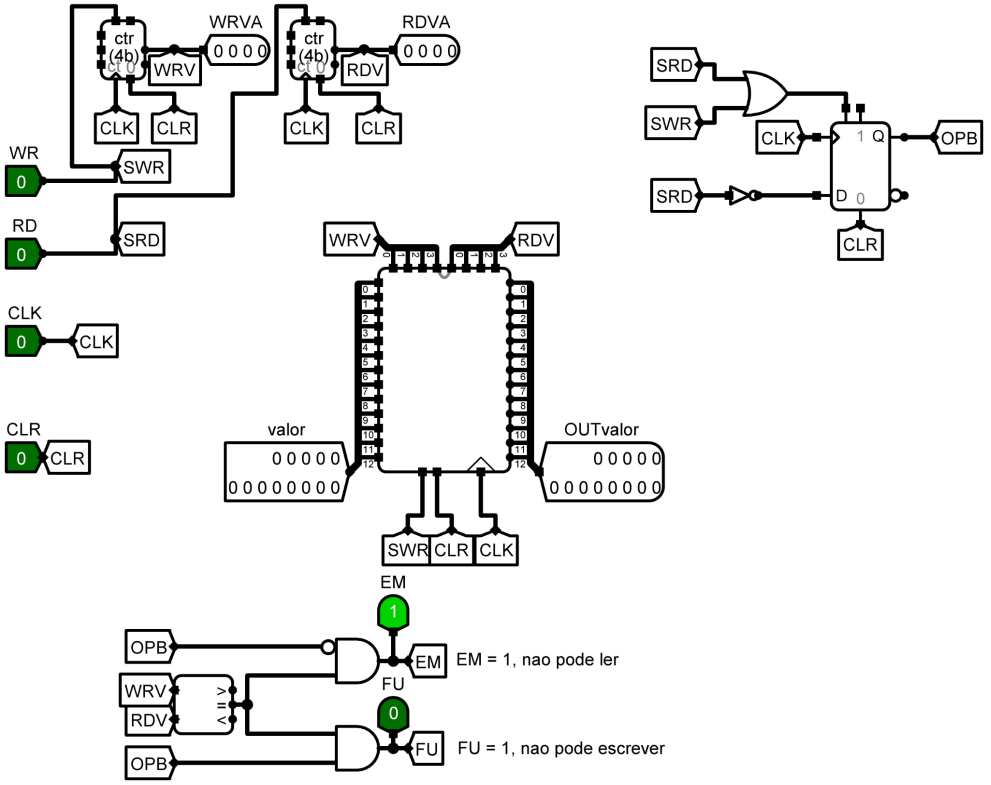


(a) (b)

Fonte: Elaborado pelos autores.

A combinação de todos esses blocos se configuram no bloco operacional.

Figura 16 - Bloco Operacional.

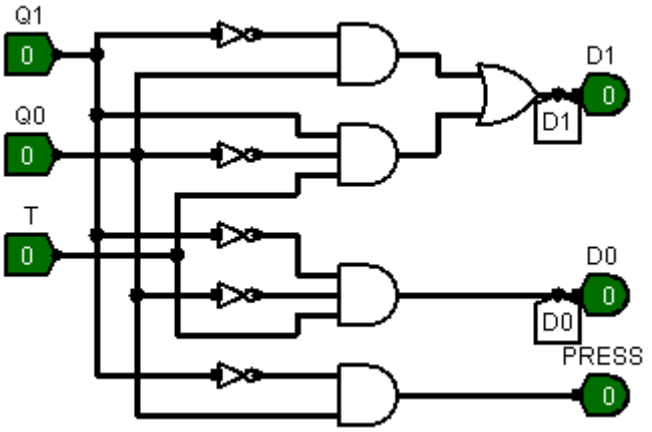


Fonte: Elaborado pelos autores.

3.1.4 Botão síncrono

Com as expressões lógicas Quadro 2, foi possível montar a lógica combinacional do no botão síncrono, a qual está exibida na Figura 17.

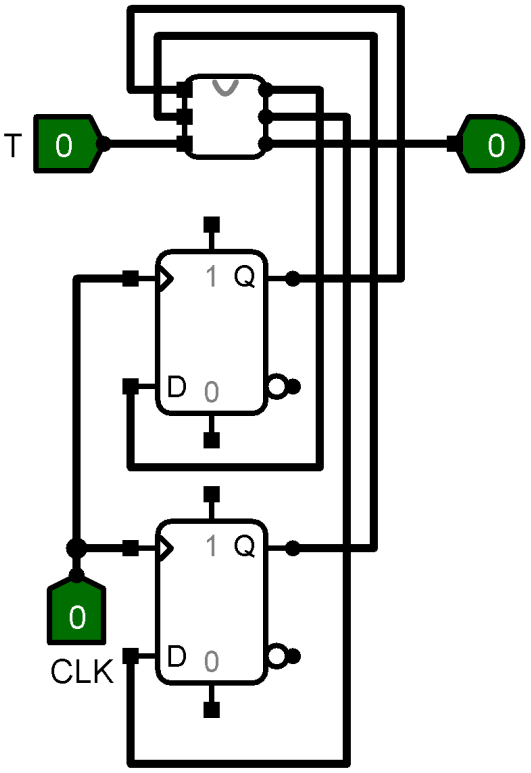
Figura 17 - Lógica do botão síncrono.



Fonte: Elaborado pelos autores.

Após isso, o restante da máquina de estados do botão síncrono foi montada, isto é, o registrador de estados foi conectado à lógica combinacional.

Figura 18 - Acionamento do botão síncrono.



Fonte: Elaborado pelos autores.

3.2 IMPLEMENTAÇÃO NO MODELSIM

O desenvolvimento do projeto em VHDL foi feito tomando como base o esquemático produzido no *Logisim* e foi composto por três componentes principais (Bloco operacional, Bloco de controle e FIFO), os quais utilizam blocos menores (como flip-flop do tipo D, somadores, comparadores, registradores, multiplexadores e demultiplexadores). Aqui no relatório será explicado com detalhes os três principais blocos mas, de toda forma, o código-fonte completo estará disponível no Anexo D.

O desenvolvimento do bloco operacional, denominado BLOCO_OPERACIONAL, foi construído tomando como base a Figura 6, onde é possível ver seu diagrama em blocos. Ele foi definido com uma entrada de dados w_data de 13 bits; os habilitadores de escrita e leitura, wr_en e rd_en; o *clear* clr (essas três últimas entradas, de 1 bit cada, são saídas do bloco de controle); e o *clock* clk (comum a todo o circuito). Como saída este componente tem os endereços de memória atuais de leitura e escrita WRaddr e RDaddr (saídas opcionais, optadas por serem utilizadas para acompanhamento de processo nas simulações); também tem há a saída de dados r_data de 13 bits; e, por fim, as *flags* empty e full, que serão utilizadas no bloco de controle para indicar se o banco de registradores está vazio ou cheio.

Figura 19 - Definição do bloco BLOCO_OPERACIONAL e chamada de componentes auxiliares.

```
504  --=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=--
505      -- B L O C O    O P E R A C I O N A L
506  --=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=--
507  entity BLOCO_OPERACIONAL is
508      port (w_data: in bit_vector(12 downto 0);
509            wr_en, rd_en, clr, clk: in bit;
510            r_data: out bit_vector(12 downto 0);
511            WRaddr, RDaddr: out bit_vector(3 downto 0); -- enderecos de memoria atuais
512            empty, full: out bit);
513  end BLOCO_OPERACIONAL;
514
515  architecture hardware of BLOCO_OPERACIONAL is
516
517  component CONT4 is
518  port (ld, clr, clk: in bit; -- clr barrado
519        O: out bit_vector(3 downto 0));
520  end component;
521
522  component COMP4 is
523  port (A, B: in bit_vector(3 downto 0);
524        AmenorB, AigualB, AmaiorB: out bit);
525  end component;
526
527  component FFD is
528  port (D, S, R, clk: in bit; -- set e reset barrados
529        Q, NQ: out bit);
530  end component;
531
532  component MUX2x1 is
533  port(A, B, S: in bit;
534       Y: out bit);
535  end component;
536
537  component BANCO_REG is
538      port(w_data: in bit_vector(12 downto 0); -- dado a ser escrito
539           wr_en: in bit; -- habilitador de leitura
540           waddr, raddr: in bit_vector(3 downto 0); -- endereços de memoria de escrita e leitura, respectivamente
541           clr, clk: in bit; -- clear barrado
542           r_data: out bit_vector(12 downto 0)); -- dado a ser lido
543  end component;
```

Fonte: Elaborado pelos autores.

Na Figura 19 além de poder ser vista a definição do bloco com suas respectivas entradas e saídas, é possível ver, também, a definição dos componentes auxiliares que serão utilizados. Como pôde ser visto no diagrama deste bloco (Figura 6), ele é composto por um banco de registradores 16x13; dois contadores de 4 bits, para definir os endereços de memória do início e fim da FIFO; um comparador de 4 bits para comparar esses dois endereços; e um flip-flop tipo D para armazenar a operação anterior (ver Figura 15).

Tendo cada um dos componentes definidos, o próximo passo é criar sinais auxiliares necessários, chamar cada componente e conectá-los, como pode ser visto na Figura 20.

Figura 20 - Arquitetura do BLOCO_OPERACIONAL.

```
545  signal waddr_lt_raddr, waddr_eq_raddr, waddr_gt_raddr: bit; -- variaveis do comparador
546  signal I_vc, ld_vc, D_vc, Q_vc, NQ_vc: bit; -- variaveis pras saidas vazio e cheio
547  signal waddr, raddr: bit_vector(3 downto 0); -- endereços dos comparadores de escrita e leitura
548
549  begin
550      -- contadores
551      CONTADOR_ESCRITA: CONT4 port map(wr_en, clr, clk, waddr);
552      CONTADOR_LEITURA: CONT4 port map(rd_en, clr, clk, raddr);
553
554      -- banco de registradores
555      BANCO_DE_REG: BANCO_REG port map(w_data, wr_en, waddr, raddr, clr, clk, r_data);
556
557      -- comparador
558      COMPARADOR: COMP4 port map(waddr, raddr, waddr_lt_raddr, waddr_eq_raddr, waddr_gt_raddr);
559
560      -- vazio ou cheio
561      ld_vc <= rd_en or wr_en;
562
563      LOAD_VC: MUX2X1 port map(Q_vc, I_vc, ld_vc, D_vc);
564      INP_VC: MUX2x1 port map('1', '0', rd_en, I_vc);
565      FF0: FFD port map(D_vc, '1', clr, clk, Q_vc, NQ_vc);
566
567      empty <= NQ_vc and waddr_eq_raddr;
568      full <= Q_vc and waddr_eq_raddr;
569
570      -- saidas
571      WRaddr<= waddr;
572      RDaddr <= raddr;
573
574  end hardware;
```

Fonte: Elaborado pelos autores.

Outro bloco muito importante é o bloco de controle. Neste, é onde a máquina de estados de baixo nível da Figura 5 está definida, assim como suas saídas. Foi adotado nesta etapa, assim como foi feito no esquemático da Figura 14, que as variáveis de entrada wr e rd, definidas pelo usuário, passaria por uma pequena MDE denominada botão síncrono, para assegurar que estas variáveis só teriam nível lógico durante um pulso de clock, independente de passar um certo tempo pressionadas. Dessa forma, este bloco foi denominado

BLOCO_DE_CONTROLE; ele possui como entradas wr, rd e clr_fifo, definidos pelo usuário; também empty e full, vindas do bloco operacional, e o clock; como saída há as variáveis wr_en, rd_en e clr, que serão entradas do bloco operacional; por fim, há as variáveis fu e em, que serão saídas do sistema, informando que o banco está cheio ou vazio. Como componentes a serem utilizados neste bloco, é utilizado a MDE do botão síncrono e um flip-flop D para os estados da FIFO. As definições de entradas, saídas e componentes utilizados podem ser vistos na Figura 21.

Figura 21 - Definição do bloco BLOCO_DE_CONTROLE e chamada de componentes auxiliares.

```vhdl
--=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
    -- B L O C O   D E   C O N T R O L E
--=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
entity BLOCO_DE_CONTROLE is
    port (wr, rd, clr_fifo, empty, full, clk: in bit;
            wr_en, rd_en, clr, fu, em: out bit);
end BLOCO_DE_CONTROLE;

architecture hardware of BLOCO_DE_CONTROLE is

component FFD is
port (D, S, R, clk: in bit; -- set e reset barrados
        Q, NQ: out bit);
end component;

component BS is
    port(t, clk: in bit;
        press: out bit);
end component;
```

Fonte: Elaborado pelos autores.

Após isso, foram definidos sinais auxiliares; aplicado os botões síncronos às entradas wr e rd; feita a lógica dos estados; e as saídas. As equações booleanas para as lógicas dos estados e saídas podem ser vistos no Quadro 3 e, o desenvolvimento disso, na Figura 22.

Figura 22 - Arquitetura do BLOCO_DE_CONTROLE.

```
471   signal Nem, Nfu: bit; -- barrados
472   signal wr_b, rd_b, clr_b, Nwr_b, Nrd_b, Nclr_b: bit; -- botoes sincronos
473   signal Dff, Q, NQ: bit_vector(1 downto 0); -- estados
474
475   begin
476       -- auxiliares
477       Nem <= not empty;
478       Nfu <= not full;
479       Nwr_b <= not wr_b;
480       Nrd_b <= not rd_b;
481
482       -- botao sincrono
483       BOTAO_WR:  BS port map(wr, clk, wr_b);
484       BOTAO_RD:  BS port map(rd, clk, rd_b);
485
486       -- estados
487       Dff(0) <= (NQ(1) and NQ(0) and clr_fifo) or (NQ(1) and NQ(0) and Nrd_b and wr_b and Nfu);
488       Dff(1) <= (NQ(1) and NQ(0) and clr_fifo) or (NQ(1) and NQ(0) and rd_b and Nem);
489
490       FF0: FFD port map(Dff(0), '1', '1', clk, Q(0), NQ(0));
491       FF1: FFD port map(Dff(1), '1', '1', clk, Q(1), NQ(1));
492
493       -- saidas
494       clr <= not(Q(1) and Q(0));
495       rd_en <= Q(1) and NQ(0);
496       wr_en <= NQ(1) and Q(0);
497       fu <= full;
498       em <= empty;
499
500   end hardware;
```

Fonte: Elaborado pelos autores.

Por fim, o bloco principal que chama todos os componentes anteriores é a FIFO propriamente dita. Ela tem como entradas apenas os dados inseridos pelo usuário (wr, rd, clr_fifo e w_data) e o *clock* do circuito. Como saída há o r_data; as *flags* de memória cheia ou vazia (fu e em); e, como saídas opcionais, há o waddr e o raddr, que indicam os endereços atuais de leitura e escrita. A ideia é, basicamente, fazer a conexão entre os blocos de controle e operacional, assim como na Figura 10. O código-fonte completo pode ser visto na Figura 23.

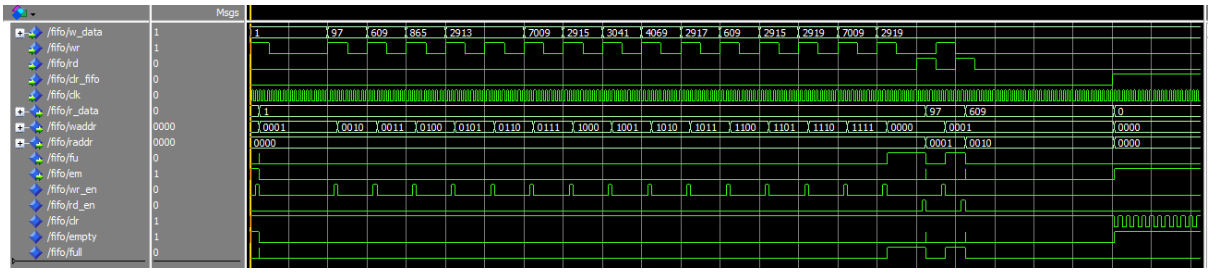
Foram feitos vários testes e, na Figura 24, é possível ver um deles. Como pode ser observado, antes de ser inserido qualquer valor, a saída em tem nível lógico alto e a saída r_data está zerada, uma vez que ela está lendo o último valor da fila e ele ainda não existe. Ao ser inserido o primeiro dado (1), a saída r_data, após o wr_en=1 dado pelo botão síncrono e a próxima borda de subida do *clock*, também exibe este valor. Após isso, são inseridos novos dados (97, 609, 865, …, 2919) até encher a memória da FIFO, indicada por fu=1. A próxima instrução que é dada é a de leitura, sendo assim assim, é exibido na saída r_data o segundo valor da fila (97); depois é feita mais uma leitura, exibindo o terceiro valor (609). Por fim, é dado um sinal de limpeza (clr_fifo=1); é possível perceber que após isso os endereços de início e final da fila são zerados (waddr=0000 e raddr=0000), há a indicação de memória vazia (em=1) e a saída não exibe mais nada (r_data=0).

Figura 23 - Código fonte do componente de memória FIFO, em VHDL.

```
574  ---=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=---
575                          -- F I F O
576  ---=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=---
577  entity FIFO is
578      port (w_data: in bit_vector(12 downto 0);
579            wr, rd, clr_fifo, clk: in bit;
580            r_data: out bit_vector(12 downto 0);
581            fu, em: out bit;
582            waddr, raddr: out bit_vector(3 downto 0));
583  end FIFO;
584
585  architecture hardware of FIFO is
586
587  component BLOCO_DE_CONTROLE is
588      port (wr, rd, clr_fifo, empty, full, clk: in bit;
589            wr_en, rd_en, clr, fu, em: out bit);
590  end component;
591
592  component BLOCO_OPERACIONAL is
593      port (w_data: in bit_vector(12 downto 0);
594            wr_en, rd_en, clr, clk: in bit;
595            r_data: out bit_vector(12 downto 0);
596            empty, full: out bit);
597  end component;
598
599  signal wr_en, rd_en, clr, empty, full: bit;
600
601  begin
602
603      BlocoDeControle: BLOCO_DE_CONTROLE port map(wr, rd, clr_fifo, empty, full, clk, wr_en, rd_en, clr, fu, em);
604      BlocoOperacional: BLOCO_OPERACIONAL port map(w_data, wr_en, rd_en, clr, clk, r_data, waddr, raddr, empty, full);
605
606  end hardware ; -- hardware
```

Fonte: Elaborado pelos autores.

Figura 24 - Simulação, em VHDL, da FIFO.



Fonte: Elaborado pelos autores.

Sendo assim, como pôde ser visto na Figura 24, as simulações feitas no projeto desenvolvido em VHDL mostram que está funcionando como esperado. E, por fim, mais uma vez, o código fonte completo pode ser visto no Anexo D.

4 CONCLUSÃO

Em sistemas digitais, há algumas situações em que se faz necessário um armazenamento ordenado de dados inseridos pelo usuário. Nesse contexto, existem algumas ferramentas que contemplam o armazenamento de informações na ordem em que esta é inserida, como por exemplo, as FIFOs. Diante disso, o presente relatório tem por finalidade apresentar a implementação de um projeto do circuito lógico de uma FIFO.

Em posse do projeto, foram realizadas algumas correções visando detalhar circuitos que o projetista não havia especificado. Após isso, o circuito foi implementado em um *software* que possibilita a visualização dos componentes (LogiSim) e em VHDL. Os resultados encontrados foram os esperados, evidenciando que o circuito foi projetado corretamente.

Por fim, a confecção do projeto da FIFO nos permitiu o entendimento de circuitos lógicos dos quais ainda não havíamos trabalhado, como: memória RAM, banco de registradores e dos circuitos que operam com dados introduzidos e retirados de forma ordenada, como as FIFOS.

REFERÊNCIAS

VAHID, Frank. **Sistemas digitais: projeto, otimização e HDLS**. Rio Grande do Sul: Artmed Bookman, 2008.

FREEPIK. **Ícone de pessoas na fila**. Disponível em: https://br.freepik.com/vetores-gratis/icone-de-pessoas-na-fila-definido-com-pessoas-diferentes-esperando-na-fila-para-a-ilustracao-de-caixa-eletronico_6870838.htm. Acesso em: 09 abr. 2021.

ANEXOS

ANEXO A - Relato semanal

Líder: Lucas Batista da Fonseca

A.1 Equipe

Tabela 1 - Identificação da equipe.

Função	Discente
Redator	Sthefania Fernandes Silva
Debatedor	João Matheus Bernardo Resende
Videomaker	Isaac de Lyra Junior
Auxiliar	Igor Michael Araujo de Macedo

Fonte: Elaboração própria.

A.2 Defina o problema

A necessidade de registrar dados e realizar a sua leitura a partir do primeiro dado que foi inserido, essa metodologia é conhecida como memória FIFO (do inglês *first-in first-out*, ou seja, o primeiro que entra é o primeiro que sai). Partindo dessa premissa, a problemática dessa semana é realizar a implementação de uma memória FIFO com 16 registradores de 13 bits, essa memória possui 5 entradas, sendo elas: entrada de dados (W_DATA); habilitador de leitura (WR); habilitados de escrita (RD); limpador de registros (CLR_FIFO); clock. E possuindo 3 saídas, sendo elas: saída de dados (R_DATA); memória cheia (FU); memória vazia (EM.)

A.3 Registro do *brainstorming*

Inicialmente como houve a concatenação de dois grupos, foi discutido qual grupo iremos realizar a implementação, como tínhamos integrantes do grupo 2 da semana passada e já possuíam uma base de implementação, decidimos utilizar o projeto do grupo 2 como referência. Além disso, nessa semana discutimos como iríamos realizar a implementação do banco de registros, tivemos inicialmente a ideia de usar barramento com buffer tri-state, entretanto, tivemos dificuldade na implementação, por fim decidimos utilizar multiplexador para selecionar a saída de um registrador. Por fim, realizamos debates de ideias para a implementação do registrador da operação anterior que verifica se o registrador está cheio ou vazio.

A.4 Pontos-Chaves

Os pontos-chaves para o desenvolvimento deste problema foi, além de ter uma boa base de conhecimentos em conceitos de circuitos sequenciais, máquina de estados finitos e projeto RTL, foi o estudo de banco de registradores e elementos de memória.

A.5 Questões de pesquisa

- Circuitos sequenciais;
- Máquina de estados finitos;
- Projeto RTL;
- Banco de registradores;
- Elementos de memória;
- FIFO.

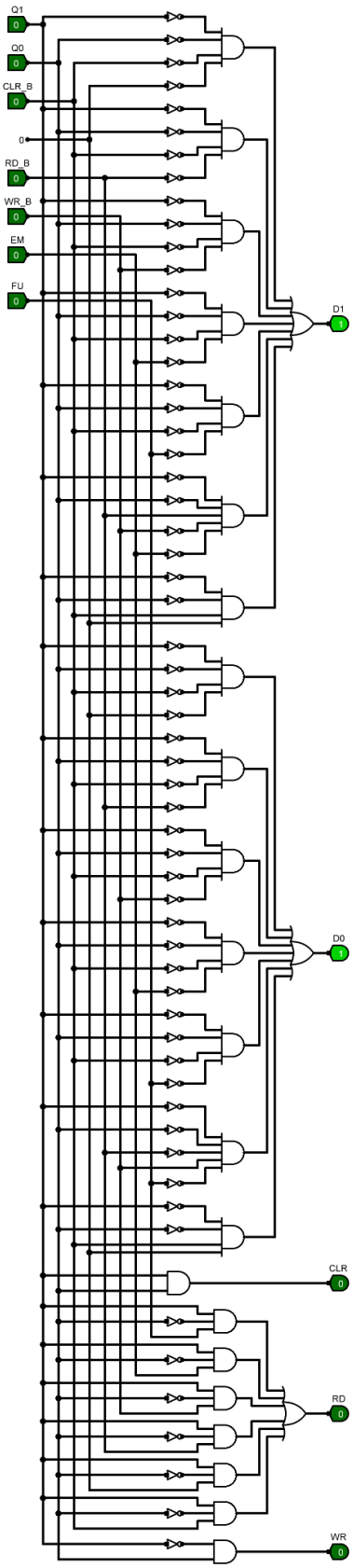
A.6 Planejamento da pesquisa

Foi combinado utilizar os primeiros dias para leitura e estudo mais aprofundado do relatório do grupo 2. Após essa etapa foi realizada pesquisa de implementação do banco de registro. A principal fonte de pesquisa foi o livro do Vahid, 2008,(componentes de blocos operacionais e projeto RTL, respectivamente), mais especificamente as seções sobre bancos de registradores e componentes de memória.

ANEXO B - Tabela de Estados da Máquina de Baixo Nível

ESTADO	PROCESSOS	INPUT							OUTPUT				
		ESTADO ATUAL		BOTÕES			EM	FU	ESTADO FUTURO		CLR	RD	WR
		Q1	Q0	CLR_B	RD_B	WR_B			D1	D0			
INÍCIO 00	LIMPAR	0	0	0	X	X	X	X	1	1	0	0	0
	INÍCIO	0	0	1	0	0	X	X	0	0	0	0	0
	PODE ESCREVER	0	0	1	0	1	X	0	0	1	0	0	0
	NÃO PODE ESCREVER	0	0	1	0	1	X	1	0	0	0	0	0
	PODE LER	0	0	1	1	X	0	X	1	0	0	0	0
	NÃO PODE LER	0	0	1	1	X	1	X	0	0	0	0	0
ESCREVER 01	ESCREVENDO	0	1	X	X	X	X	X	0	0	0	0	1
LER 10	LENDO	1	0	X	X	X	X	X	0	0	0	1	0
LIMPAR 11	LIMPANDO	1	1	X	X	X	X	X	0	0	1	0	0

ANEXO C - Lógica Combinacional da Máquina de Estados da FIFO.



ANEXO D - Código fonte, em VHDL, do elemento de memória FIFO

```
---=========-=========-=========-=========-=========-=========-=========-=========-=---
        -- F L I P - F L O P    T I P O     D
---=========-=========-=========-=========-=========-=========-=========-=========-=---

entity FFD is
    port (D, S, R, clk: in bit; -- set e reset barrados
            Q, NQ: out bit);
end FFD;

architecture hardware of FFD is

signal QS: bit;

begin
    process(CLK, S, R)
    begin
        if S = '0' then QS <= '1';
        elsif R = '0' then QS <= '0';
        elsif clk = '1' and clk'event then QS <= D;
        end if;
    end process;

    Q <= QS;
    NQ <= not(QS);

end hardware;

---=========-=========-=========-=========-=========-=========-=========-=========-=---
        -- S O M A D O R    C O M P L E T O
---=========-=========-=========-=========-=========-=========-=========-=========-=---
entity COMP_ADD is
    port(A, B, CI: in bit;
            S, CO: out bit);
```

```
end COMP_ADD;

architecture hardware of COMP_ADD is

begin
    S <= A xor B xor CI;
    CO <= (B and CI) or (A and CI) or (A and B);
end hardware;

---=========-=========-=========-=========-=========-=========-=========-=---
        -- S O M A D O R    D E    4   B I T S
---=========-=========-=========-=========-=========-=========-=========-=---
entity ADD4 is
    port(A, B: in bit_vector(3 downto 0);
            O: out bit_vector(3 downto 0);
            CO: out bit);
end ADD4;

architecture hardware of ADD4 is

component COMP_ADD
    port(A, B, CI: in bit;
            S, CO: out bit);
end component;

signal VAI_UM: bit_vector(2 downto 0);

begin
    S0: COMP_ADD port map(A(0), B(0), '0', O(0), VAI_UM(0));
    S1: COMP_ADD port map(A(1), B(1), VAI_UM(0), O(1), VAI_UM(1));
    S2: COMP_ADD port map(A(2), B(2), VAI_UM(1), O(2), VAI_UM(2));
    S3: COMP_ADD port map(A(3), B(3), VAI_UM(2), O(3), CO);
end hardware;

---=========-=========-=========-=========-=========-=========-=========-=---
```

```
    -- C O N T A D O R    D E    4 B I T S
----------------------------------------------------------------
entity CONT4 is
    port (ld, clr, clk: in bit; -- clr barrado
            O: out bit_vector(3 downto 0));
end CONT4;

architecture hardware of CONT4 is

component REGISTER4BITS is
    port (I: in bit_vector(3 downto 0);
            ld, clr, clk: in bit; -- clr barrado
            O: out bit_vector(3 downto 0));
end component;

component ADD4 is
    port(A, B: in bit_vector(3 downto 0);
            O: out bit_vector(3 downto 0);
            CO: out bit);
end component;

signal CO: bit;
signal D, Q: bit_vector(3 downto 0);

begin

    SOMADOR: ADD4 port map(Q, ('0', '0', '0', '1'), D, CO);
    REGISTER4: REGISTER4BITS port map(D, ld, clr, clk, Q);

    -- saida
    O <= Q;

end hardware;

----------------------------------------------------------------
```

```
    -- C O M P A R A D O R    D E     4 B I T S
---=========================================================-----
entity COMP4 is
    port (A, B: in bit_vector(3 downto 0);
            AmenorB, AigualB, AmaiorB: out bit);
end COMP4;

architecture hardware of COMP4 is

signal A0eqB0, A1eqB1, A2eqB2, A3eqB3: bit;
Signal A0gtB0, A1gtB1, A2gtB2, A3gtB3: bit;
signal AeqB, AgtB, AltB: bit;

begin
    -- A igual a B
    A0eqB0 <= not(A(0) xor B(0));
    A1eqB1 <= not(A(1) xor B(1));
    A2eqB2 <= not(A(2) xor B(2));
    A3eqB3 <= not(A(3) xor B(3));

    AeqB <= A0eqB0 and A1eqB1 and A2eqB2 and A3eqB3;

    -- A maior que B
    A0gtB0 <= A(0) and not(B(0)) and A3eqB3 and A2eqB2 and A1eqB1;
    A1gtB1 <= A(1) and not(B(1)) and A3eqB3 and A2eqB2;
    A2gtB2 <= A(2) and not(B(2)) and A3eqB3;
    A3gtB3 <= A(3) and not(B(3));

    AgtB <= A0gtB0 or A1gtB1 or A2gtB2 or A3gtB3;

    -- A menor que B
    AltB <= not(AeqB or AgtB);

    -- saidas
    AmenorB <= AltB;
```

```
    AigualB <= AeqB;
    AmaiorB <= AgtB;

end hardware;

---=========-----------=========-----------=========-----------=========-----
                    -- M U X    2 x 1
---=========-----------=========-----------=========-----------=========-----
entity MUX2x1 is
    port(A, B, S: in bit;
             Y: out bit);
end MUX2x1;

architecture hardware of MUX2x1 is

begin

    Y <= (A and (not S)) or (B and S);
end hardware;

---=========-----------=========-----------=========-----------=========-----
                    -- D E M U X    1x16
---=========-----------=========-----------=========-----------=========-----
entity DEMUX1x16 is
    port(I, en: in bit;
        S: in bit_vector(3 downto 0);
        O: out bit_vector(15 downto 0));
end DEMUX1x16;

architecture hardware of DEMUX1x16 is

begin

     O(0)  <= en and I and (not S(3) and not S(2) and not S(1) and not
S(0));
```

```
     O(1)   <= en and I and (not S(3) and not S(2) and not S(1) and
S(0));
    O(2)  <= en and I and (not S(3) and not S(2) and     S(1) and not
S(0));
     O(3)   <= en and I and (not S(3) and not S(2) and     S(1) and
S(0));
    O(4)  <= en and I and (not S(3) and     S(2) and not S(1) and not
S(0));
     O(5)   <= en and I and (not S(3) and     S(2) and not S(1) and
S(0));
    O(6)  <= en and I and (not S(3) and     S(2) and     S(1) and not
S(0));
     O(7)   <= en and I and (not S(3) and     S(2) and     S(1) and
S(0));
    O(8)  <= en and I and (    S(3) and not S(2) and not S(1) and not
S(0));
     O(9)   <= en and I and (    S(3) and not S(2) and not S(1) and
S(0));
    O(10) <= en and I and (    S(3) and not S(2) and     S(1) and not
S(0));
     O(11) <= en and I and (    S(3) and not S(2) and     S(1) and
S(0));
    O(12) <= en and I and (    S(3) and     S(2) and not S(1) and not
S(0));
     O(13) <= en and I and (    S(3) and     S(2) and not S(1) and
S(0));
    O(14) <= en and I and (    S(3) and     S(2) and     S(1) and not
S(0));
     O(15) <= en and I and (    S(3) and     S(2) and     S(1) and
S(0));

end hardware;

---=========-=========-=========-=========-=========-=========-=====

                    -- M U X    16x1

---=========-=========-=========-=========-=========-=========-=====
entity MUX16x1 is
```

```
     port(I0, I1, I2, I3, I4, I5, I6, I7, I8, I9, I10, I11, I12, I13,
I14, I15: in BIT;
        S: in bit_vector(3 downto 0);
        O: out bit);
end MUX16x1;

architecture hardware of MUX16x1 is

begin

    O <= (I0  and not S(3) and not S(2) and not S(1) and not S(0))
      or (I1  and not S(3) and not S(2) and not S(1) and     S(0))
      or (I2  and not S(3) and not S(2) and     S(1) and not S(0))
      or (I3  and not S(3) and not S(2) and     S(1) and     S(0))
      or (I4  and not S(3) and     S(2) and not S(1) and not S(0))
      or (I5  and not S(3) and     S(2) and not S(1) and     S(0))
      or (I6  and not S(3) and     S(2) and     S(1) and not S(0))
      or (I7  and not S(3) and     S(2) and     S(1) and     S(0))
      or (I8  and     S(3) and not S(2) and not S(1) and not S(0))
      or (I9  and     S(3) and not S(2) and not S(1) and     S(0))
      or (I10 and     S(3) and not S(2) and     S(1) and not S(0))
      or (I11 and     S(3) and not S(2) and     S(1) and     S(0))
      or (I12 and     S(3) and     S(2) and not S(1) and not S(0))
      or (I13 and     S(3) and     S(2) and not S(1) and     S(0))
      or (I14 and     S(3) and     S(2) and     S(1) and not S(0))
      or (I15 and     S(3) and     S(2) and     S(1) and     S(0));

end hardware;

---=========-=========-=========-=========-=========-=========-=====

                    -- M U X    16x1_13

---=========-=========-=========-=========-=========-=========-=====
entity MUX16x1_13 is
     port(I0, I1, I2, I3, I4, I5, I6, I7, I8, I9, I10, I11, I12, I13,
I14, I15: in BIT_VECTOR(12 DOWNTO 0);
```

```
        S: in bit_vector(3 downto 0);
        O: out BIT_VECTOR(12 DOWNTO 0));
end MUX16x1_13;

architecture hardware of MUX16x1_13 is

component MUX16x1 is
     port(I0, I1, I2, I3, I4, I5, I6, I7, I8, I9, I10, I11, I12, I13,
I14, I15: in BIT;
        S: in bit_vector(3 downto 0);
        O: out bit);
end component;

begin

     BIT0:  MUX16x1 port map (I0(0),  I1(0),  I2(0),  I3(0),  I4(0),
I5(0),  I6(0),  I7(0),  I8(0),  I9(0),  I10(0),  I11(0),  I12(0),
I13(0),  I14(0),  I15(0),  S, O(0));
     BIT1:  MUX16x1 port map (I0(1),  I1(1),  I2(1),  I3(1),  I4(1),
I5(1),  I6(1),  I7(1),  I8(1),  I9(1),  I10(1),  I11(1),  I12(1),
I13(1),  I14(1),  I15(1),  S, O(1));
     BIT2:  MUX16x1 port map (I0(2),  I1(2),  I2(2),  I3(2),  I4(2),
I5(2),  I6(2),  I7(2),  I8(2),  I9(2),  I10(2),  I11(2),  I12(2),
I13(2),  I14(2),  I15(2),  S, O(2));
     BIT3:  MUX16x1 port map (I0(3),  I1(3),  I2(3),  I3(3),  I4(3),
I5(3),  I6(3),  I7(3),  I8(3),  I9(3),  I10(3),  I11(3),  I12(3),
I13(3),  I14(3),  I15(3),  S, O(3));
     BIT4:  MUX16x1 port map (I0(4),  I1(4),  I2(4),  I3(4),  I4(4),
I5(4),  I6(4),  I7(4),  I8(4),  I9(4),  I10(4),  I11(4),  I12(4),
I13(4),  I14(4),  I15(4),  S, O(4));
     BIT5:  MUX16x1 port map (I0(5),  I1(5),  I2(5),  I3(5),  I4(5),
I5(5),  I6(5),  I7(5),  I8(5),  I9(5),  I10(5),  I11(5),  I12(5),
I13(5),  I14(5),  I15(5),  S, O(5));
     BIT6:  MUX16x1 port map (I0(6),  I1(6),  I2(6),  I3(6),  I4(6),
I5(6),  I6(6),  I7(6),  I8(6),  I9(6),  I10(6),  I11(6),  I12(6),
I13(6),  I14(6),  I15(6),  S, O(6));
```

```
    BIT7:  MUX16x1 port map (I0(7),  I1(7),  I2(7),  I3(7),  I4(7),
I5(7),  I6(7),  I7(7),  I8(7),  I9(7),  I10(7),  I11(7),  I12(7),
I13(7),  I14(7),  I15(7),  S, O(7));

    BIT8:  MUX16x1 port map (I0(8),  I1(8),  I2(8),  I3(8),  I4(8),
I5(8),  I6(8),  I7(8),  I8(8),  I9(8),  I10(8),  I11(8),  I12(8),
I13(8),  I14(8),  I15(8),  S, O(8));

    BIT9:  MUX16x1 port map (I0(9),  I1(9),  I2(9),  I3(9),  I4(9),
I5(9),  I6(9),  I7(9),  I8(9),  I9(9),  I10(9),  I11(9),  I12(9),
I13(9),  I14(9),  I15(9),  S, O(9));

    BIT10: MUX16x1 port map (I0(10), I1(10), I2(10), I3(10), I4(10),
I5(10),  I6(10),  I7(10),  I8(10),  I9(10),  I10(10),  I11(10),  I12(10),
I13(10), I14(10), I15(10), S, O(10));

    BIT11: MUX16x1 port map (I0(11), I1(11), I2(11), I3(11), I4(11),
I5(11),  I6(11),  I7(11),  I8(11),  I9(11),  I10(11),  I11(11),  I12(11),
I13(11), I14(11), I15(11), S, O(11));

    BIT12: MUX16x1 port map (I0(12), I1(12), I2(12), I3(12), I4(12),
I5(12),  I6(12),  I7(12),  I8(12),  I9(12),  I10(12),  I11(12),  I12(12),
I13(12), I14(12), I15(12), S, O(12));


end hardware;


--=============================================================
    -- R E G I S T R A D O R    D E    4 B I T S
--=============================================================
entity REGISTER4BITS is
    port (I: in bit_vector(3 downto 0);
            ld, clr, clk: in bit; -- clr barrado
            O: out bit_vector(3 downto 0));
end REGISTER4BITS;

architecture hardware of REGISTER4BITS is

component FFD is
    port (D, S, R, clk: in bit; -- set e reset barrados
            Q, NQ: out bit);
end component;
```

```
component MUX2x1 is
    port(A, B, S: in bit;
            Y: out bit);
end component;

signal D, Q, NQ: bit_vector(3 downto 0);

begin
    -- condicao de carregamento
    MUX1: MUX2x1 port map(Q(0), I(0), ld, D(0));
    MUX2: MUX2x1 port map(Q(1), I(1), ld, D(1));
    MUX3: MUX2x1 port map(Q(2), I(2), ld, D(2));
    MUX4: MUX2x1 port map(Q(3), I(3), ld, D(3));

    -- flip-flops:
    FF0: FFD port map(D(0), '1', clr, clk, Q(0), NQ(0));
    FF1: FFD port map(D(1), '1', clr, clk, Q(1), NQ(1));
    FF2: FFD port map(D(2), '1', clr, clk, Q(2), NQ(2));
    FF3: FFD port map(D(3), '1', clr, clk, Q(3), NQ(3));

    -- saída
    O <= Q;

end hardware;

---=========-=========-=========-=========-=========-=========-=========-=========-=====
    -- R E G I S T R A D O R    D E    13 B I T S
---=========-=========-=========-=========-=========-=========-=========-=========-=====
entity REGISTER13BITS is
    port( I: in bit_vector(12 downto 0);
            ld, clr, clk: in bit; -- clr barrado
            O: out bit_vector(12 downto 0));
end REGISTER13BITS;

architecture hardware of REGISTER13BITs is
```

```
component FFD is
    port (D, S, R, clk: in bit; -- set e reset barrados
            Q, NQ: out bit);
end component;

component MUX2x1 is
    port(A, B, S: in bit;
            Y: out bit);
end component;

signal D, Q, NQ: bit_vector(12 downto 0);

begin
    -- condicao de carregamento
    MUX0:  MUX2x1 port map(Q(0),  I(0),  ld, D(0));
    MUX1:  MUX2x1 port map(Q(1),  I(1),  ld, D(1));
    MUX2:  MUX2x1 port map(Q(2),  I(2),  ld, D(2));
    MUX3:  MUX2x1 port map(Q(3),  I(3),  ld, D(3));
    MUX4:  MUX2x1 port map(Q(4),  I(4),  ld, D(4));
    MUX5:  MUX2x1 port map(Q(5),  I(5),  ld, D(5));
    MUX6:  MUX2x1 port map(Q(6),  I(6),  ld, D(6));
    MUX7:  MUX2x1 port map(Q(7),  I(7),  ld, D(7));
    MUX8:  MUX2x1 port map(Q(8),  I(8),  ld, D(8));
    MUX9:  MUX2x1 port map(Q(9),  I(9),  ld, D(9));
    MUX10: MUX2x1 port map(Q(10), I(10), ld, D(10));
    MUX11: MUX2x1 port map(Q(11), I(11), ld, D(11));
    MUX12: MUX2x1 port map(Q(12), I(12), ld, D(12));

    -- flip-flops:
    FF0:  FFD port map(D(0),  '1', clr, clk, Q(0),  NQ(0));
    FF1:  FFD port map(D(1),  '1', clr, clk, Q(1),  NQ(1));
    FF2:  FFD port map(D(2),  '1', clr, clk, Q(2),  NQ(2));
    FF3:  FFD port map(D(3),  '1', clr, clk, Q(3),  NQ(3));
    FF4:  FFD port map(D(4),  '1', clr, clk, Q(4),  NQ(4));
```

```
    FF5:  FFD port map(D(5),  '1', clr, clk, Q(5),  NQ(5));
    FF6:  FFD port map(D(6),  '1', clr, clk, Q(6),  NQ(6));
    FF7:  FFD port map(D(7),  '1', clr, clk, Q(7),  NQ(7));
    FF8:  FFD port map(D(8),  '1', clr, clk, Q(8),  NQ(8));
    FF9:  FFD port map(D(9),  '1', clr, clk, Q(9),  NQ(9));
    FF10: FFD port map(D(10), '1', clr, clk, Q(10), NQ(10));
    FF11: FFD port map(D(11), '1', clr, clk, Q(11), NQ(11));
    FF12: FFD port map(D(12), '1', clr, clk, Q(12), NQ(12));

    -- saída
    O <= Q;

end hardware;

---=========-=========-=========-=========-=========-=========-=========-=---
    -- B A N C O    D E    R E G I S T R A D O R E S
---=========-=========-=========-=========-=========-=========-=========-=---
entity BANCO_REG is
    port(w_data: in bit_vector(12 downto 0); -- dado a ser escrito
            wr_en: in bit; -- habilitador de leitura
              waddr, raddr: in bit_vector(3 downto 0); -- endereços de
memoria de escrita e leitura, respectivamente
            clr, clk: in bit; -- clear barrado
            r_data: out bit_vector(12 downto 0)); -- dado a ser lido
end BANCO_REG;

architecture hardware of BANCO_REG is

component REGISTER13BITS is
    port( I: in bit_vector(12 downto 0);
        ld, clr, clk: in bit; -- clr barrado
        O: out bit_vector(12 downto 0));
end component;

component MUX16x1_13 is
```

```
     port(I0, I1, I2, I3, I4, I5, I6, I7, I8, I9, I10, I11, I12, I13,
I14, I15: in BIT_VECTOR(12 DOWNTO 0);
        S: in bit_vector(3 downto 0);
        O: out BIT_VECTOR(12 DOWNTO 0));
end component;

component DEMUX1x16 is
    port(I, en: in bit;
        S: in bit_vector(3 downto 0);
        O: out bit_vector(15 downto 0));
end component;

signal ld: bit_vector (15 downto 0);
signal r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14,
r15: bit_vector(12 downto 0);

begin
       -- demux para selecionar qual registrador deve ser carregado
(endereco waddr)
    DEFINE_LOAD: DEMUX1x16 port map('1', wr_en, waddr, ld);

     -- registradores, so ira carregar no registrador correspondente a
saida do demux anterior
      DEFINE_REG0:   REGISTER13BITS port map(w_data, ld(0),   clr, clk,
r0);
      DEFINE_REG1:   REGISTER13BITS port map(w_data, ld(1),   clr, clk,
r1);
      DEFINE_REG2:   REGISTER13BITS port map(w_data, ld(2),   clr, clk,
r2);
      DEFINE_REG3:   REGISTER13BITS port map(w_data, ld(3),   clr, clk,
r3);
      DEFINE_REG4:   REGISTER13BITS port map(w_data, ld(4),   clr, clk,
r4);
      DEFINE_REG5:   REGISTER13BITS port map(w_data, ld(5),   clr, clk,
r5);
      DEFINE_REG6:   REGISTER13BITS port map(w_data, ld(6),   clr, clk,
r6);
```

```
      DEFINE_REG7:   REGISTER13BITS port map(w_data, ld(7),   clr, clk,
r7);
      DEFINE_REG8:   REGISTER13BITS port map(w_data, ld(8),   clr, clk,
r8);
      DEFINE_REG9:   REGISTER13BITS port map(w_data, ld(9),   clr, clk,
r9);
      DEFINE_REG10:  REGISTER13BITS port map(w_data, ld(10), clr, clk,
r10);
      DEFINE_REG11:  REGISTER13BITS port map(w_data, ld(11), clr, clk,
r11);
      DEFINE_REG12:  REGISTER13BITS port map(w_data, ld(12), clr, clk,
r12);
      DEFINE_REG13:  REGISTER13BITS port map(w_data, ld(13), clr, clk,
r13);
      DEFINE_REG14:  REGISTER13BITS port map(w_data, ld(14), clr, clk,
r14);
      DEFINE_REG15:  REGISTER13BITS port map(w_data, ld(15), clr, clk,
r15);

    --seleciona qual registrador deve ser lido (endereco raddr)
     DEFINE_READ: MUX16x1_13 port map(r0, r1, r2, r3, r4, r5, r6, r7,
r8, r9, r10, r11, r12, r13, r14, r15, raddr, r_data);

end hardware;

---=========-=========-=========-=========-=========-=========-=========-=====
            -- B O T A O    S I N C R O N O
---=========-=========-=========-=========-=========-=========-=========-=====
entity BS is
    port(t, clk: in bit;
        press: out bit);
end BS;

architecture hardware of BS is

component FFD is
    port (D, S, R, clk: in bit; -- set e reset barrados
```

```
            Q, NQ: out bit);
end component;

signal D, Q, NQ: bit_vector (1 downto 0);

begin
    -- logica combinacional
    D(1) <= (NQ(1) and Q(0)) or (Q(1) and NQ(0) and t);
    D(0) <= NQ(1) and NQ(0) and t;

    -- estados
    FF0: FFD port map(D(0), '1', '1', clk, Q(0), NQ(0));
    FF1: FFD port map(D(1), '1', '1', clk, Q(1), NQ(1));

    --saida
    press <= NQ(1) and Q(0);

end hardware;


---=========-=========-=========-=========-=========-=========-=========-=---
    -- B L O C O    D E    C O N T R O L E
---=========-=========-=========-=========-=========-=========-=========-=---
entity BLOCO_DE_CONTROLE is
    port (wr, rd, clr_fifo, empty, full, clk: in bit;
            wr_en, rd_en, clr, fu, em: out bit);
end BLOCO_DE_CONTROLE;

architecture hardware of BLOCO_DE_CONTROLE is

component FFD is
port (D, S, R, clk: in bit; -- set e reset barrados
        Q, NQ: out bit);
end component;
```

```
component BS is
    port(t, clk: in bit;
        press: out bit);
end component;

signal Nem, Nfu: bit; -- barrados
signal wr_b, rd_b, clr_b, Nwr_b, Nrd_b, Nclr_b: bit; -- botoes sincronos
signal Dff, Q, NQ: bit_vector(1 downto 0); -- estados

begin
    -- auxiliares
    Nem <= not empty;
    Nfu <= not full;
    Nwr_b <= not wr_b;
    Nrd_b <= not rd_b;

    -- botao sincrono
    BOTAO_WR:  BS port map(wr, clk, wr_b);
    BOTAO_RD:  BS port map(rd, clk, rd_b);

    -- estados
     Dff(0) <= (NQ(1) and NQ(0) and clr_fifo) or (NQ(1) and NQ(0) and Nrd_b and wr_b and Nfu);
     Dff(1) <= (NQ(1) and NQ(0) and clr_fifo) or (NQ(1) and NQ(0) and rd_b and Nem);

    FF0: FFD port map(Dff(0), '1', '1', clk, Q(0), NQ(0));
    FF1: FFD port map(Dff(1), '1', '1', clk, Q(1), NQ(1));

    -- saidas
    clr <= not(Q(1) and Q(0));
    rd_en <= Q(1) and NQ(0);
    wr_en <= NQ(1) and Q(0);
    fu <= full;
```

```
    em <= empty;

end hardware;

---=========-----------------------------------------------
    -- B L O C O    O P E R A C I O N A L
---=========-----------------------------------------------
entity BLOCO_OPERACIONAL is
    port (w_data: in bit_vector(12 downto 0);
            wr_en, rd_en, clr, clk: in bit;
            r_data: out bit_vector(12 downto 0);
             WRaddr, RDaddr: out bit_vector(3 downto 0); -- enderecos de
memoria atuais
            empty, full: out bit);
end BLOCO_OPERACIONAL;

architecture hardware of BLOCO_OPERACIONAL is

component CONT4 is
port (ld, clr, clk: in bit; -- clr barrado
        O: out bit_vector(3 downto 0));
end component;

component COMP4 is
port (A, B: in bit_vector(3 downto 0);
        AmenorB, AigualB, AmaiorB: out bit);
end component;

component FFD is
port (D, S, R, clk: in bit; -- set e reset barrados
        Q, NQ: out bit);
end component;

component MUX2x1 is
port(A, B, S: in bit;
```

```
        Y: out bit);
end component;

component BANCO_REG is
    port(w_data: in bit_vector(12 downto 0); -- dado a ser escrito
            wr_en: in bit; -- habilitador de leitura
              waddr, raddr: in bit_vector(3 downto 0); -- endereços de
memoria de escrita e leitura, respectivamente
            clr, clk: in bit; -- clear barrado
            r_data: out bit_vector(12 downto 0)); -- dado a ser lido
end component;

signal   waddr_lt_raddr,   waddr_eq_raddr,   waddr_gt_raddr:   bit;   --
variaveis do comparador
signal I_vc, ld_vc, D_vc, Q_vc, NQ_vc: bit; -- variaveis pras saidas
vazio e cheio
signal   waddr,   raddr:   bit_vector(3   downto   0);   --   endereços   dos
comparadores de escrita e leitura

begin
    -- contadores
    CONTADOR_ESCRITA: CONT4 port map(wr_en, clr, clk, waddr);
    CONTADOR_LEITURA: CONT4 port map(rd_en, clr, clk, raddr);

    -- banco de registradores
     BANCO_DE_REG: BANCO_REG port map(w_data, wr_en, waddr, raddr, clr,
clk, r_data);

    -- comparador
         COMPARADOR:   COMP4   port   map(waddr,   raddr,   waddr_lt_raddr,
waddr_eq_raddr, waddr_gt_raddr);

    -- vazio ou cheio
    ld_vc <= rd_en or wr_en;

    LOAD_VC: MUX2X1 port map(Q_vc, I_vc, ld_vc, D_vc);
```

```
    INP_VC: MUX2x1 port map('1', '0', rd_en, I_vc);
    FF0: FFD port map(D_vc, '1', clr, clk, Q_vc, NQ_vc);

    empty <= NQ_vc and waddr_eq_raddr;
    full <= Q_vc and waddr_eq_raddr;

    -- saidas
    WRaddr<= waddr;
    RDaddr <= raddr;

end hardware;

--=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
                        -- F I F O
--=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
entity FIFO is
    port (w_data: in bit_vector(12 downto 0);
            wr, rd, clr_fifo, clk: in bit;
            r_data: out bit_vector(12 downto 0);
            fu, em: out bit;
            waddr, raddr: out bit_vector(3 downto 0));
end FIFO;

architecture hardware of FIFO is

component BLOCO_DE_CONTROLE is
    port (wr, rd, clr_fifo, empty, full, clk: in bit;
            wr_en, rd_en, clr, fu, em: out bit);
end component;

component BLOCO_OPERACIONAL is
    port (w_data: in bit_vector(12 downto 0);
            wr_en, rd_en, clr, clk: in bit;
            r_data: out bit_vector(12 downto 0);
            empty, full: out bit);
```

```
end component;

signal wr_en, rd_en, clr, empty, full: bit;

begin

       BlocoDeControle: BLOCO_DE_CONTROLE port map(wr, rd, clr_fifo,
empty, full, clk, wr_en, rd_en, clr, fu, em);
     BlocoOperacional: BLOCO_OPERACIONAL port map(w_data, wr_en, rd_en,
clr, clk, r_data, waddr, raddr, empty, full);

end hardware ; -- hardware
```