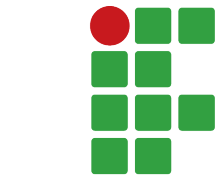


INSTITUTO FEDERAL
Santa Catarina

Câmpus
São José

AE2

Estudando uma FIFO

Curso: Engenharia de Telecomunicações
Disciplina: ELD129003 - Eletrônica Digital 2
Professor: Marcos Moecke

Aluna
Luiza Kuze Gomes

2 de agosto de 2024

Sumário

1 Introdução

Este relatório tem como objetivo analisar três circuitos: o Register File, o controlador FIFO (First-In-First-Out) e uma FIFO completa. A FIFO completa é composta pelos dois circuitos mencionados, ou seja, o Register File e o controlador FIFO. Em seguida, será abordada a função de cada um desses circuitos na construção da FIFO completa.

O foco da análise será o desempenho (Fmax), a área ocupada pelo circuito em diferentes versões e a simulação funcional dos componentes individuais e da FIFO completa.

Para as análises realizadas, utilizei inicialmente o dispositivo da família *Cyclone IV E*, modelo *EP4CE6E22A7*, tanto para o Register File quanto para o controlador FIFO. Para a FIFO completa, foi utilizado o modelo *EP4CE115F29I8L*, que oferece mais recursos de hardware.

2 Código do Register File

O Register File é um componente organizado como uma matriz de registradores capazes de armazenar e manipular dado. Cada registrador na matriz é acessado por um endereço único de *W bits* e pode armazenar um conjunto de dados com tamanho fixo de *N bits*.

Durante as aulas, foram estudadas duas versões do Register File: uma versão parametrizada, que possui um código mais limpo e pode ser facilmente adaptada, e uma versão não parametrizada, que não segue essas boas práticas e se torna quase impossível de simular em larga escala. Inicialmente, serão discutidas as características individuais de cada versão, examinando seu código.

Por fim, a versão parametrizada será explorada em maior profundidade, considerando valores de 4 e 16 endereços no registrador, destacando como essas mudanças influenciam a eficiência em termos de número de elementos lógicos, frequência máxima de operação do circuito e visualização no RTL Viewer.

2.1 Comparação entre Versão Parametrizada e Não Parametrizada

A versão parametrizada é o objetivo trabalhado desde o início da disciplina, desenvolvendo códigos genéricos que permitem alterações na quantidade de bits facilmente. Além disso, esse tipo de código é mais legível e fácil de entender.

Por outro lado, na versão não parametrizada, surgem desafios significativos ao tentar adaptar o código para diferentes valores de bits de endereço. A seguir, apresentarei uma comparação detalhada entre os trechos de código parametrizados e não parametrizados.

2.1.1 Inicialização dos Registradores

Duas importantes melhorias foram implementadas nesta parte do código. A primeira foi a otimização da inicialização dos registradores durante o reset, e a segunda foi a mudança do valor do sinal `array_reg` para `array_next` a cada borda de subida do clock.

No código não otimizado, a inicialização dos registradores era feita posição por posição durante o reset, conforme exemplificado a seguir:

```
1  -- Inicialização dos Registradores Ruim
2  process(clk, reset)
3  begin
4      if (reset = '1') then
5          array_reg(3) <= (others => '0');
6          array_reg(2) <= (others => '0');
7          array_reg(1) <= (others => '0');
8          array_reg(0) <= (others => '0');
9      elsif (clk'event and clk = '1') then
10         array_reg(3) <= array_next(3);
```

```
        array_reg(2) <= array_next(2);
        array_reg(1) <= array_next(1);
        array_reg(0) <= array_next(0);
    end if;
end process;
```

No código otimizado, a expressão `(others => (others => '0'))` é utilizada para zerar todas as posições de endereço de uma só vez. Esse método é mais eficiente, pois elimina a necessidade de zerar cada posição individualmente. A sintaxe diferente de `(others=>'0')`, que já havia sido utilizada em aula, é necessária porque `array_reg` é definido como uma matriz, então é preciso zerar todas as linhas e colunas da matriz simultaneamente:

```
-- Inicialização dos Registradores Boa
process(clk, reset)
begin
    if (reset = '1') then
        array_reg <= (others => (others => '0'));
    elsif (clk'event and clk = '1') then
        array_reg <= array_next;
    end if;
end process;
```

2.1.2 Lógica de Escrita nos Registradores

No código, a lógica de escrita nos registradores foi aprimorada para simplificar e tornar o processo mais eficiente. No código não parametrizado, a escrita nos registradores é feita individualmente para cada posição, como mostrado no exemplo abaixo:

```
-- Lógica de Escrita nos Registradores Ruim
process(array_reg, en, w_data)
begin
    array_next(3) <= array_reg(3);
    array_next(2) <= array_reg(2);
    array_next(1) <= array_reg(1);
    array_next(0) <= array_reg(0);
    if en(3) = '1' then
        array_next(3) <= w_data;
    end if;
    if en(2) = '1' then
        array_next(2) <= w_data;
    end if;
    if en(1) = '1' then
        array_next(1) <= w_data;
    end if;
    if en(0) = '1' then
        array_next(0) <= w_data;
    end if;
end process;
```

Neste código, cada posição do registrador é atualizada separadamente com base no sinal de `en`. Isso resulta em um código repetitivo, pois cada linha de código precisa ser ajustada para cada posição individual do registrador.

Em contraste, o código parametrizado utiliza uma abordagem mais compacta. A lógica de escrita é realizada de forma genérica, atualizando todos os registradores de uma vez, conforme o endereço de escrita. O exemplo a seguir ilustra essa abordagem:

```
-- Lógica de Escrita nos Registradores Boa
process(array_reg, wr_en, w_addr, w_data)
begin
```

```
    array_next <= array_reg;
    if wr_en = '1' then
        array_next(to_integer(unsigned(w_addr))) <= w_data;
    end if;
end process;
```

Neste código otimizado, `array_next` é atualizado com o valor de `array_reg`, e a escrita é realizada diretamente no endereço especificado por `w_addr`. O trecho `to_integer(unsigned(w_addr))` é utilizado para converter o endereço de escrita para um valor inteiro, permitindo acessar e atualizar a posição correta do registrador. Esta abordagem é mais escalável, facilitando a manutenção.

2.1.3 Decodificação para Endereço de Escrita

Na versão parametrizada do código, a decodificação do endereço de escrita é incorporada diretamente na lógica de escrita, eliminando a necessidade de um processo adicional para habilitar os sinais de escrita. Isso ocorre porque a lógica de escrita parametrizada usa uma abordagem mais direta e genérica para atualizar o conteúdo dos registradores com base no endereço fornecido.

Por outro lado, na versão não parametrizada, um processo adicional é necessário para realizar a decodificação do endereço de escrita e habilitar o sinal de escrita correspondente. O código a seguir ilustra essa abordagem não parametrizada:

```
-- Decodificação para Endereço de Escrita Ruim
process(wr_en, w_addr)
begin
    if (wr_en = '0') then
        en <= (others => '0');
    else
        case w_addr is
            when "00" => en <= "0001";
            when "01" => en <= "0010";
            when "10" => en <= "0100";
            when others => en <= "1000";
        end case;
    end if;
end process;
```

Neste código, a decodificação é realizada por meio de uma estrutura `case` que ativa o sinal de escrita apropriado com base no valor de `w_addr`. Isso implica que o código precisa verificar explicitamente cada valor possível de `w_addr` para determinar qual sinal de escrita deve ser habilitado. Essa abordagem pode ser propensa a erros e difícil de manter, especialmente à medida que o número de endereços aumenta.

2.1.4 Leitura dos Registradores

A leitura dos registradores é significativamente mais direta na versão parametrizada. Nessa abordagem, a leitura é realizada diretamente com base no endereço de leitura fornecido, como ilustrado no código a seguir:

```
-- Leitura dos Registradores Boa
r_data <= array_reg(to_integer(unsigned(r_addr)));
```

Neste código parametrizado, o endereço de leitura `r_addr` é convertido para um valor inteiro usando o trecho de código `to_integer(unsigned(r_addr))`, e esse valor é utilizado para acessar diretamente o registrador correspondente no array `array_reg`. Esse método é eficiente porque evita a necessidade de lógica adicional para selecionar o registrador a ser lido, e proporciona uma maneira clara e compacta de realizar a leitura.

Por outro lado, no código não parametrizado, a leitura dos registradores é realizada por meio de multiplexadores, resultando em um código mais complexo e menos flexível. O código a seguir exemplifica essa abordagem:

```
-- Leitura dos Registradores Ruim
with r_addr0 select
    r_data0 <= array_reg(0) when "00",
               array_reg(1) when "01",
               array_reg(2) when "10",
               array_reg(3) when others;
with r_addr1 select
    r_data1 <= array_reg(0) when "00",
               array_reg(1) when "01",
               array_reg(2) when "10",
               array_reg(3) when others;
```

Neste código, um multiplexador é usado para selecionar o valor de `r_data0` e `r_data1` com base nos endereços `r_addr0` e `r_addr1`, respectivamente. Essa abordagem não parametrizada exige uma estrutura `with select` para cada linha do array de registradores, tornando o código mais verboso e difícil de escalar.

Portanto, a versão parametrizada proporciona uma solução mais simples para a leitura de registradores, facilitando tanto a implementação quanto a manutenção do código.

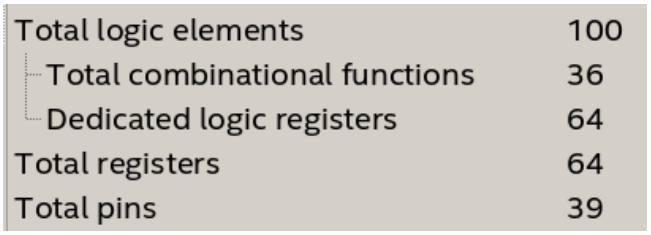
2.2 Análise Parametrizada

Nesta seção, o foco é na análise da versão parametrizada do sistema, alterando a quantidade de bits de endereçamento de dados de 2 para 4, obtendo assim 4 e 16 endereços, respectivamente. O objetivo é examinar como essa mudança impacta diversos aspectos do sistema, incluindo o número de elementos lógicos, a estrutura do *RTL* Viewer, a frequência máxima e a simulação funcional.

2.2.1 Elementos Lógicos

O aumento no número de bits de endereçamento leva a um incremento no uso dos recursos da FPGA. Esse aumento inclui o número de elementos lógicos e funções combinacionais. O número de pinos também aumenta, mas de forma mais moderada, indicando que a maior parte da complexidade adicional é gerenciada internamente pelo circuito, conforme mostrado nas figuras 1 e 2.

Figura 1: Análise e Síntese - Register File com 4 Endereços



Fonte: Elaborada pela autora

Figura 2: Análise e Síntese - Register File com 16 Endereços

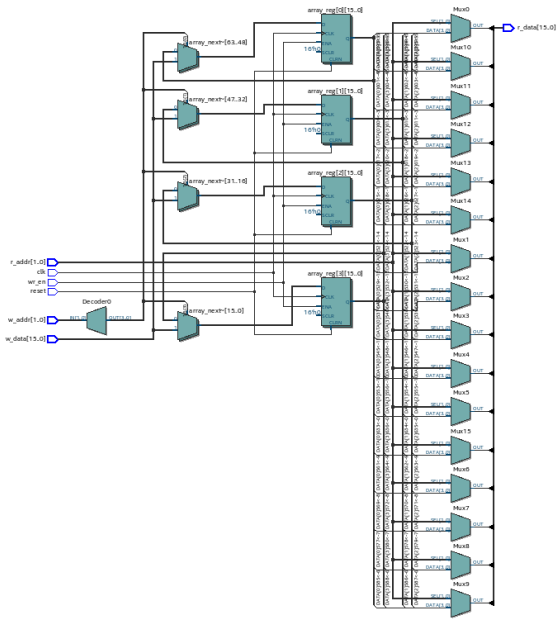
Total logic elements	440
Total combinational functions	184
Dedicated logic registers	256
Total registers	256
Total pins	43

Fonte: Elaborada pela autora

2.2.2 RTL Viewer

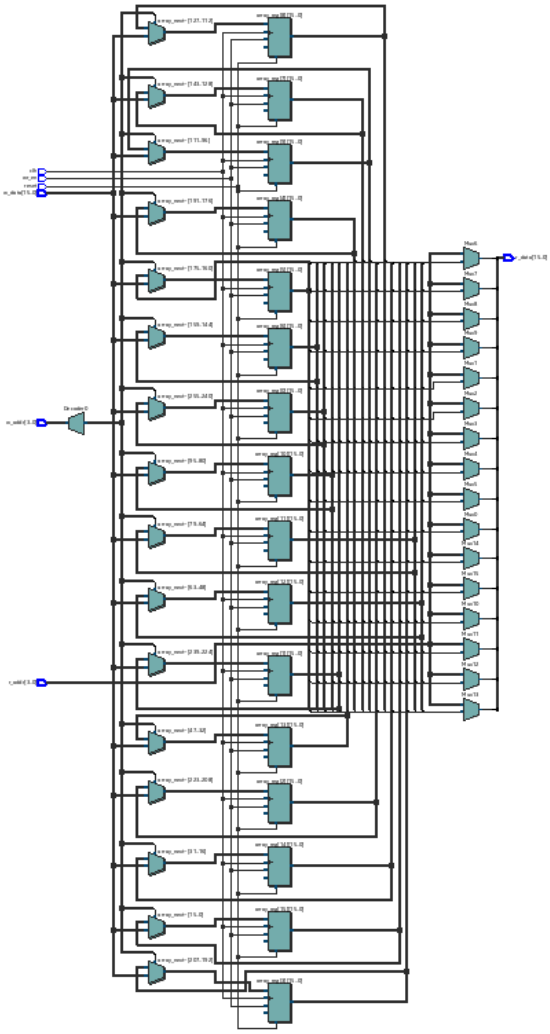
No RTL Viewer, observamos que a quantidade de novos elementos é proporcional ao aumento no número de bits inseridos no sistema. O número de barramentos cresce significativamente, dificultando a interpretação das conexões, como ilustrado na figura 4 em comparação a figura 3. Entre os elementos que surgem em quantidade proporcional, estão novos registradores e multiplexadores.

Figura 3: RTL Viewer - Register File com 4 Endereços



Fonte: Elaborada pela autora

Figura 4: RTL Viewer - Register File com 16 Endereços



Fonte: Elaborada pela autora

2.2.3 Frequência Máxima

Não foi possível obter a frequência máxima. Ao abrir a ferramenta *Timing Analyzer*, há somente a mensagem "No paths to report."

2.2.4 Simulação Funcional

Na simulação, podemos verificar efetivamente as diferenças entre os dados de endereço e os dados armazenados. No código, definimos dois valores genéricos: W e N. O valor W define o número de bits de endereço, que nos dá uma quantidade de 2^W endereços no total. O valor N define o número de bits no valor armazenado em cada posição de endereço. Esta configuração gera uma matriz no código, onde o encadeamento de registradores é definido da seguinte forma:

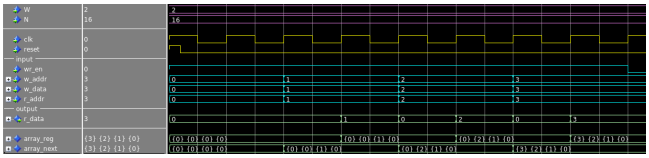
```
1 type reg_file_type is array (2**W-1 downto 0) of std_logic_vector(N-1 downto 0);
```

Como até o momento nunca havia sido utilizado esse tipo, a maior dificuldade foi entender seu funcionamento. Somente após a simulação a compreensão se tornou mais clara com a ideia da matriz.

Analisando a simulação, a lógica adotada foi escrever valores em uma sequência crescente no vetor de posições de escrita. Com N posições, os valores escritos variam de 0 até $N - 1$. Mantive o endereço de leitura sincronizado com esses valores.

Verificando a saída, observamos na figura 5 que a ordem apresentada é "0 1 0 2 0 3 0 4". Isso ocorre porque a posição anterior ainda não havia registrado o número correspondente. No entanto, no próximo ciclo de clock, a atualização é refletida, uma vez que o código é sensível à borda de subida.

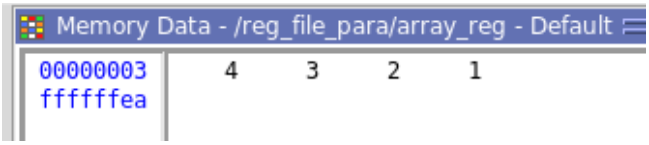
Figura 5: Resultado da Simulação - Register File com 4 Endereços



Fonte: Elaborada pela autora

Outra maneira de visualizar os dados armazenados é através da *Memory List* no *ModelSim*, uma ferramenta que permite observar o registro de cada valor nas posições definidas. Ao final da execução do testbench desenvolvido para o Register File com 4 posições de endereço, a memória é exibida da seguinte forma:

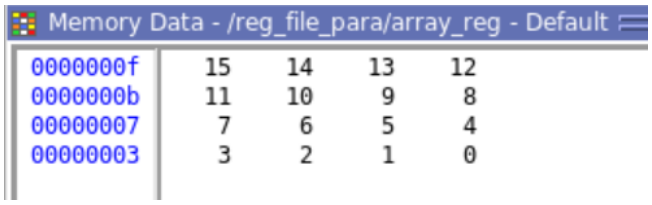
Figura 6: Visualização da Memória - Register File com 4 Endereços



Fonte: Elaborada pela autora

A simulação para 16 endereços segue uma abordagem semelhante, com a única diferença na alteração na quantidade de bits de endereço. Isso resulta em uma matriz com outro tamanho de armazenamento, essa alteração na matriz pode ser verificada no *Memory List* para uma simulação de 16 endereços na figura 7. A lógica e o comportamento permanecem consistentes.

Figura 7: Visualização da Memória - Register File com 16 Endereços



Fonte: Elaborada pela autora

3 Código do Controlador de FIFO

O controlador de FIFO é essencial na gestão de uma estrutura FIFO, organizando os elementos e coordenando os ponteiros de escrita e leitura. Diferente do Register File (seção 2), as posições de dados na FIFO não são acessíveis de forma aleatória, garantindo que o primeiro dado escrito será o primeiro lido.

No estudo atual, três implementações do controlador foram desenvolvidas e utilizam diferentes tipos de contadores:

- **Contador Binário**: Um registrador que incrementa seu valor a cada pulso de clock, proporcionando uma contagem sequencial direta.
- **LFSR (Linear Feedback Shift Register)**: Um registrador de deslocamento que gera sequências pseudoaleatórias.

As três arquiteturas implementadas são:

1. **Enlarged Binary Counter**: Utiliza um contador binário.
2. **Lookahead Binary Counter**: Utiliza lógica de previsão para atualizar os ponteiros.
3. **LFSR (Linear Feedback Shift Register)**: Utiliza lógica de previsão e um contador LFSR.

Nesta seção, são documentados a simulação funcional, o RTL Viewer, o número de elementos lógicos e a frequência máxima para 16 e 1024 endereços de memória, registrando as mudanças nos ponteiros de escrita e leitura.

3.1 Enlarged Binary Counter

A arquitetura `enlarged_bin_arch` utiliza um contador binário aumentado, com um bit extra para gerenciar os ponteiros de escrita e leitura. Nesta arquitetura, o bit mais significativo (MSB) dos ponteiros é usado para determinar se a FIFO está cheia ou vazia, enquanto os bits menos significativos (LSBs) são utilizados como endereço de registro.

O ponteiro de escrita é incrementado se uma operação de escrita estiver ativa e a FIFO não estiver cheio. A flag de *"full"* é ativada quando os ponteiros de leitura e escrita têm os mesmos LSBs, mas bits mais significativos diferentes:

```
-- Lógica de próximo estado do ponteiro de escrita
w_ptr_next <=
   w_ptr_reg + 1 when wr = '1' and full_flag = '0' else
   w_ptr_reg;
full_flag <= '1' when r_ptr_reg(W) /= w_ptr_reg(W) and
              r_ptr_reg(W-1 downto 0) = w_ptr_reg(W-1 downto 0) else
              '0';
```

O ponteiro de leitura é incrementado quando uma operação de leitura está ativa e a FIFO não está vazia. A flag de *empty* é ativada se os ponteiros de leitura e escrita são iguais:

```
-- Lógica de próximo estado do ponteiro de leitura
r_ptr_next <=
   r_ptr_reg + 1 when rd = '1' and empty_flag = '0' else
   r_ptr_reg;
empty_flag <= '1' when r_ptr_reg = w_ptr_reg else
              '0';
```

A arquitetura é a mais simples, pois o contador binário apenas incrementa seu valor anterior sem grandes mudanças. Mais adiante, será discutido como essa simplicidade pode afetar o desempenho, sugerindo que outras arquiteturas podem ser mais eficientes.

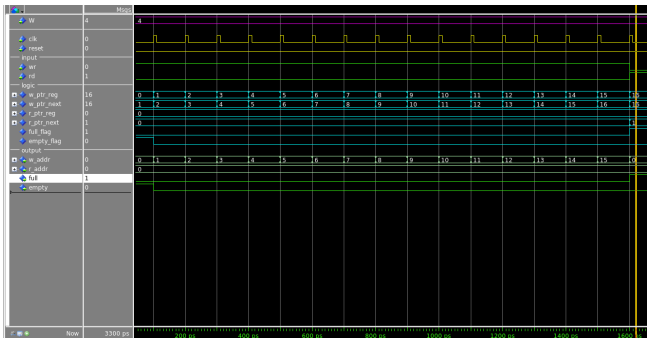
3.1.1 Simulação Funcional

A simulação foi realizada utilizando 16 endereços, com o objetivo de visualizar a mudança dos valores das saídas *full* e *empty*, que são alteradas a cada operação de escrita e leitura.

A simulação no *ModelSim*, apresentada na Figura 8, mostra as entradas de escrita e leitura definidas como "1"e "0", respectivamente. Todas as posições do controlador de FIFO são ocupadas, fazendo com que o MSB do ponteiro de escrita se torne 1. Note que o MSB do ponteiro de escrita é

diferente do MSB do ponteiro de leitura. Quando todas as posições estão ocupadas, a saída *full* é alterada para "1".

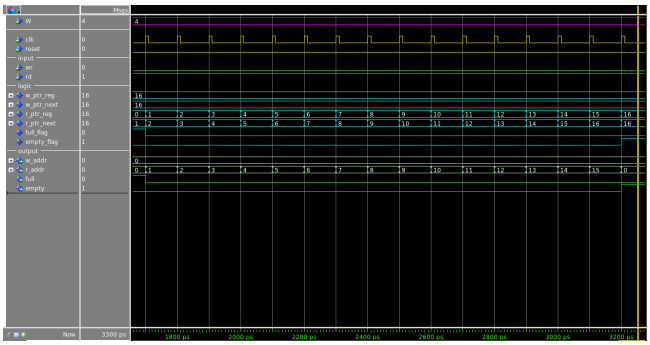
Figura 8: Simulação Funcional do FIFO Controller - Enlarged Bin Counter (FIFO Cheia)



Fonte: Elaborada pela autora

Na operação reversa, apresentada na Figura 9, com as entradas de escrita e leitura definidas como "0"e "1", respectivamente, todos os dados armazenados são removidos (lidos). Note que o MSB do ponteiro de escrita se iguala ao MSB do ponteiro de leitura. Quando todas as posições estão vazias, a saída *empty* é alterada para "1".

Figura 9: Simulação Funcional do FIFO Controller - Enlarged Bin Counter (FIFO Vazia)

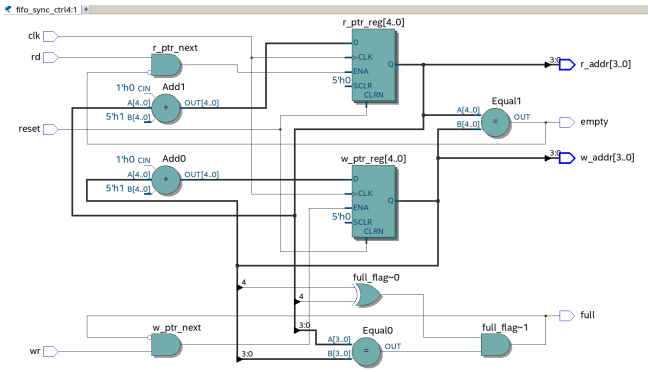


Fonte: Elaborada pela autora

3.1.2 RTL Viewer

No RTL, figura 10, é possível visualizar que os registradores `w_ptr_reg` e `r_ptr_reg` armazenam os ponteiros de escrita e leitura, respectivamente. Esses registradores são ampliados para *W+1* bits para incluir um bit adicional, utilizado para determinar a condição de `full` e `empty`.

Figura 10: RTL Viewer do FIFO Controller - Enlarged Bin Counter



Fonte: Elaborada pela autora

3.1.3 Elementos Lógicos e Frequência Máxima para 16 Endereços

A seguir, os resultados das operações feitas no Quartus II para a arquitetura `enlarged_bin_arch` com 16 endereços de armazenamento.

Elementos Lógicos Na figura 11, documento o número de elementos lógicos para a arquitetura e número de endereços em questão.

Figura 11: Elementos Lógicos do FIFO Controller - Enlarged Bin Counter (16 Endereços)

Total logic elements	17
Total registers	10
Total pins	14

Fonte: Elaborada pela autora

Timing Analysis Na tabela 1, documento a frequência máxima obtida com o circuito para a arquitetura e número de endereços em questão.

Tabela 1: Desempenho do FIFO Controller - Enlarged Bin Counter (16 Endereços)

Frequência	Condição	**Temperatura**
Slow	125 ℃	372.16 MHz
Slow	-40 ℃	434.03 MHz
Fast	-40 ℃	825.08 MHz

Fonte: Elaborada pela autora

3.1.4 Elementos Lógicos e Frequência Máxima para 1024 Endereços

A seguir, os resultados das operações feitas no Quartus II para a arquitetura `enlarged_bin_arch` com 1024 endereços de armazenamento.

Elementos Lógicos Na figura 12, documento o número de elementos lógicos para a arquitetura e número de endereços em questão.

Figura 12: Elementos Lógicos do FIFO Controller - Enlarged Bin Counter (1024 Endereços)

Total logic elements	33
Total registers	22
Total pins	26

Fonte: Elaborada pela autora

Timing Analysis Na tabela 2, documento a frequência máxima obtida com o circuito para a arquitetura e número de endereços em questão.

Tabela 2: Desempenho do FIFO Controller - Enlarged Bin Counter (1024 Endereços)

Condição	Temperatura	Frequência
Slow	125℃	285.96 MHz
Slow	-40℃	331.90 MHz
Fast	-40℃	637.35 MHz

Fonte: Elaborada pela autora

3.2 Lookahead Binary Counter

A arquitetura `lookahead_bin_arch` utiliza um contador binário como versão da seção 3.1, porém com uma lógica de *lookahead*, que antecipa o próximo valor dos ponteiros da FIFO. Os ponteiros são incrementados de acordo com a operação de escrita ou leitura.

A lógica de *lookahead* calcula os valores sucessores dos ponteiros:

```
-- Valores sucessores dos ponteiros
w_ptr_succ <= w_ptr_reg + 1;
r_ptr_succ <= r_ptr_reg + 1;
```

Em seguida, avalia as operações de leitura e escrita. O próximo estado dos ponteiros e das flags é determinado com base na operação solicitada e nas condições da FIFO:

```
-- Lógica de próximo estado
wr_op <= wr & rd;
process(w_ptr_reg, w_ptr_succ, r_ptr_reg, r_ptr_succ,
        wr_op, empty_reg, full_reg)
begin
   w_ptr_next <= w_ptr_reg;
   r_ptr_next <= r_ptr_reg;
   full_next <= full_reg;
   empty_next <= empty_reg;

   case wr_op is
      when "00" => -- Sem operação
      when "10" => -- Escrita
         if (full_reg /= '1') then
            w_ptr_next <= w_ptr_succ;
            empty_next <= '0';
            if (w_ptr_succ = r_ptr_reg) then full_next <= '1'; end if;
         end if;
```

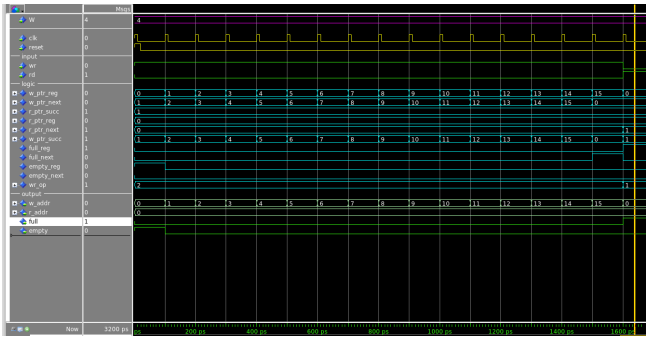
```
19        when "01" => -- Leitura
20           if (empty_reg /= '1') then
21              r_ptr_next <= r_ptr_succ;
22              full_next <= '0';
23              if (r_ptr_succ = w_ptr_reg) then empty_next <= '1'; end if;
24           end if;
25        when others => -- Leitura/Escrita
26           w_ptr_next <= w_ptr_succ;
27           r_ptr_next <= r_ptr_succ;
28     end case;
29  end process;
```

3.2.1 Simulação Funcional

Nesta seção, é demonstrada a simulação funcional do controlador FIFO utilizando a arquitetura `lookahed_bin_arch`. Diferente da arquitetura anterior, essa não utiliza um bit extra para armazenar o estado da FIFO (cheia ou vazia), mas utiliza sinais adicionais para essa finalidade. Continua com um contador binário, incrementando de um em um, como na simulação realizada anteriormente na subseção 3.1.1. Entre os sinais extras, destaco que *wr_op*, a concatenação dos sinais de entrada `wr` e `rd`, está dentro de um *case* para verificar as opções de escrita e leitura, isso é visível no trecho de código anterior.

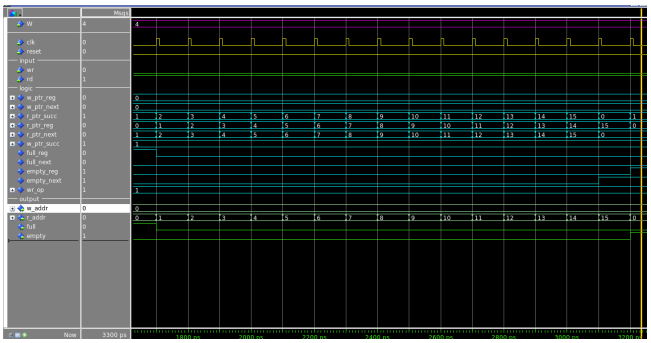
A saída `full` é ativada "1"quando a FIFO está cheia, conforme a figura 13 e a saída `empty` é ativada "1"quando a FIFO está vazia, conforme a figura 14, o que indica o correto funcionamento do controlador.

Figura 13: Simulação Funcional do FIFO Controller - Lookahead Bin Counter (FIFO Cheia)



Fonte: Elaborada pela autora

Figura 14: Simulação Funcional do FIFO Controller - Lookahead Bin Counter (FIFO Vazia)

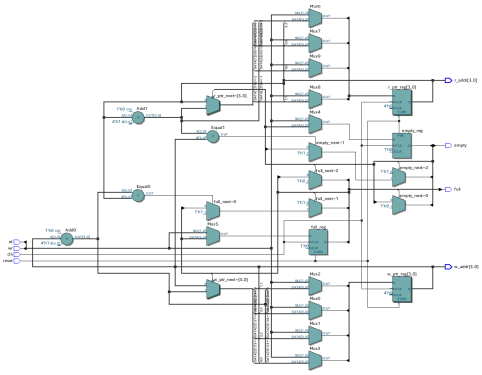


Fonte: Elaborada pela autora

3.2.2 RTL Viewer

No RTL, figura 15, é possível visualizar que essa arquitetura exigiu multiplexadores em seu funcionamento e outros blocos de registradores.

Figura 15: RTL Viewer do FIFO Controller - Lookahead Bin Counter



Fonte: Elaborada pela autora

3.2.3 Elementos Lógicos e Frequência Máxima para 16 Endereços

A seguir, os resultados das operações feitas no Quartus II para a arquitetura `lookahed_bin_arch` com 16 endereços de armazenamento.

Elementos Lógicos Na figura 16, documento o número de elementos lógicos para a arquitetura e número de endereços em questão.

Figura 16: Elementos Lógicos do FIFO Controller - Lookahead Bin Counter (16 Endereços)

Total logic elements	19
Total registers	10
Total pins	14

Fonte: Elaborada pela autora

Frequência Máxima Na tabela 3, documento a frequência máxima obtida com o circuito para a arquitetura e número de endereços em questão. Analisando os resultados obtidos anteriormente para a arquitetura `enlarged_bin_arch` também com 16 endereços (tabela 1) , não houveram grandes mudanças em relação à primeira arquitetura na questão de frequência.

Tabela 3: Desempenho do FIFO Controller - Lookahed Bin Counter (16 Endereços)

Condição	Temperatura	Frequência
Slow	125 ℃	397.77 MHz
Slow	-40 ℃	465.98 MHz
Fast	-40 ℃	892.06 MHz

Fonte: Elaborada pela autora

3.2.4 Elementos Lógicos e Frequência Máxima para 1024 Endereços

A seguir, os resultados das operações feitas no Quartus II para a arquitetura `lookahed_bin_arch` com 1024 endereços de armazenamento.

Elementos Lógicos Na figura 17, documento o número de elementos lógicos para a arquitetura e número de endereços em questão.

Figura 17: Elementos Lógicos do FIFO Controller - Lookahed Bin Counter (1024 Endereços)

Total logic elements	39
Total registers	22
Total pins	26

Fonte: Elaborada pela autora

Frequência Máxima Na tabela 4, documento a frequência máxima obtida com o circuito para a arquitetura e número de endereços em questão.

Tabela 4: Desempenho do FIFO Controller - Lookahed Bin Counter (1024 Endereços)

Condição	Temperatura	Frequência
Slow	125 ℃	227.62 MHz
Slow	-40 ℃	328.08 MHz
Fast	-40 ℃	628.93 MHz

Fonte: Elaborada pela autora

3.3 Lookahead LFSR Counter

A arquitetura `lookahead_LFSR_arch` utiliza um contador baseado em *Linear Feedback Shift Register* (LFSR) para gerenciar os ponteiros da FIFO. Esta abordagem representa uma alternativa muito mais eficiente em comparação ao contador binário padrão e emprega uma lógica de previsão similar à da arquitetura 3.2, anteriormente descrita.

Na abordagem *lookahead*, os valores sucessores dos ponteiros são calculados utilizando o LFSR. O LFSR gera um novo valor baseado no estado atual dos ponteiros, como ilustrado no código abaixo na versão para 4 bits (16 endereços):

```
1 -- Valores sucessores dos ponteiros com LFSR (4 bits)
2 w_ptr_succ <= (w_ptr_reg(1) xor w_ptr_reg(0)) & w_ptr_reg(3 downto 1);
3 r_ptr_succ <= (r_ptr_reg(1) xor r_ptr_reg(0)) & r_ptr_reg(3 downto 1);
```

De acordo com a documentação técnica (AMD, 1996), para montar um contador LFSR com outra quantidade de bits, é necessário observar com atenção em qual posição adicionar a porta XOR. Para 10 bits (1024 endereços), a análise é feita com o código da seguinte maneira:

```
1 -- Valores sucessores dos ponteiros com LFSR (10 bits)
2 w_ptr_succ <= (w_ptr_reg(3) xor w_ptr_reg(0)) & w_ptr_reg(3 downto 1);
3 r_ptr_succ <= (r_ptr_reg(3) xor r_ptr_reg(0)) & r_ptr_reg(3 downto 1);
```

A lógica de previsão é responsável por determinar o próximo estado dos ponteiros e das flags, considerando a operação solicitada e as condições da FIFO, de maneira semelhante à arquitetura 3.2.

3.3.1 Simulação Funcional

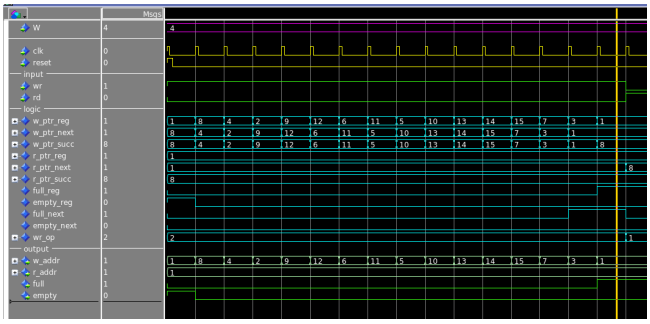
O processo de simulação segue o mesmo padrão das arquiteturas anteriores: iniciar preenchendo completamente as posições de armazenamento e, em seguida, esvaziar a FIFO.

É importante destacar que, ao contrário das duas arquiteturas anteriores, esta utiliza um contador baseado em *Linear Feedback Shift Register* (LFSR). A natureza pseudoaleatória do LFSR é evidente na simulação, refletindo-se na ordem não linear dos ponteiros de escrita.

Semelhante à arquitetura anterior, a lógica *lookahead* requer a implementação de sinais adicionais, que podem ser observados nesta simulação.

Primeiramente, a simulação do controlador com a FIFO preenchida, conforme figura 18.

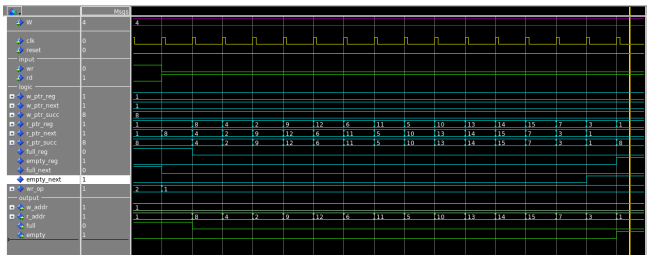
Figura 18: Simulação Funcional do FIFO Controller - Lookahead LFSR Counter (FIFO Cheia)



Fonte: Elaborada pela autora

Em seguida, apresento a simulação do controlador ao esvaziar os dados (é feita a leitura), conforme mostrado na Figura 19. Note que, apesar das posições pseudoaleatórias dos ponteiros de leitura, os primeiros dados removidos ainda são os primeiros que foram armazenados. Esse comportamento está de acordo com o padrão FIFO, que garante que os dados sejam lidos na ordem em que foram inseridos.

Figura 19: Simulação Funcional do FIFO Controller - Lookahead LFSR Counter (FIFO Vazia)

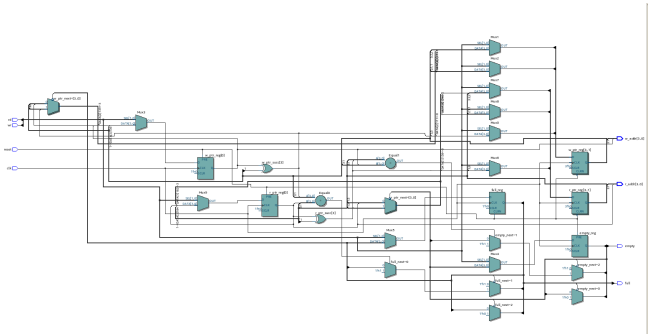


Fonte: Elaborada pela autora

3.3.2 RTL Viewer

No RTL, figura 20, é possível verificar a estrutura da arquitetura em questão.

Figura 20: RTL Viewer do FIFO Controller - Lookahead LFSR Counter



Fonte: Elaborada pela autora

3.3.3 Elementos Lógicos e Frequência Máxima para 16 Endereços

A seguir, os resultados das operações feitas no Quartus II para a arquitetura `lookahed_LFSR_arch` com 16 endereços de armazenamento.

Elementos Lógicos Na figura 21, documento o número de elementos lógicos para a arquitetura e número de endereços em questão.

Figura 21: Elementos Lógicos do FIFO Controller - Lookahead LFSR Counter (16 Endereços)

Total logic elements	19
Total registers	10
Total pins	14

Fonte: Elaborada pela autora

Frequência Máxima Na tabela 5, documento a frequência máxima obtida com o circuito para a arquitetura e número de endereços em questão.

Tabela 5: Desempenho do FIFO Controller - Lookahed LFSR Counter (16 Endereços)

Condição	Temperatura	Frequência
Slow	125℃	500.50 MHz
Slow	-40℃	597.73 MHz
Fast	-40℃	1103.75 MHz

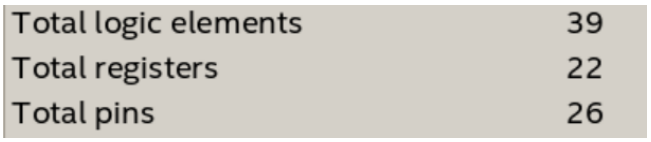
Fonte: Elaborada pela autora

3.3.4 Elementos Lógicos e Frequência Máxima para 1024 Endereços

A seguir, os resultados das operações feitas no Quartus II para a arquitetura `lookahed_bin_arch` com 1024 endereços de armazenamento.

Elementos Lógicos Na figura 22, documento o número de elementos lógicos para a arquitetura e número de endereços em questão.

Figura 22: Elementos Lógicos do FIFO Controller - Lookahed LFSR Counter (1024 Endereços)



Fonte: Elaborada pela autora

Frequência Máxima Na tabela 6, documento a frequência máxima obtida com o circuito para a arquitetura e número de endereços em questão.

Tabela 6: Desempenho do FIFO Controller - Lookahed LFSR Counter (1024 Endereços)

Condição	Temperatura	Frequência
Slow	125℃	407.00 MHz
Slow	-40℃	465.33 MHz
Fast	-40℃	889.68 MHz

Fonte: Elaborada pela autora

4 FIFO Completa

Para a construção de uma FIFO completa, foi necessário integrar os circuitos Register File e o controlador de FIFO como componentes do projeto. Destaco que ambos os circuitos foram testados individualmente antes de sua integração para formar a FIFO, seguindo os princípios de um projeto hierárquico, facilitando a identificação de possíveis erros.

As funcionalidades individuais desses componentes levam à geração da FIFO. O Register file é responsável por armazenar temporariamente os dados até que sejam lidos ou escritos de maneira organizada, ou seja, atua como a memória da FIFO. Já o controlador de FIFO é responsável por atualizar os ponteiros de leitura e escrita e assegurar que as operações de leitura e escrita sejam realizadas de forma sincronizada, evitando sobreposições e garantindo a integridade dos dados.

Na sequência, serão apresentados os mesmos passos utilizados anteriormente: análise RTL, contagem de elementos lógicos, análise da frequência máxima, simulação funcional e testes com diferentes quantidades de endereços da FIFO. E para as versões do controlador, vistas na seção 3, serão destacadas suas individualidades aqui na FIFO completa também.

4.1 Simulação Funcional

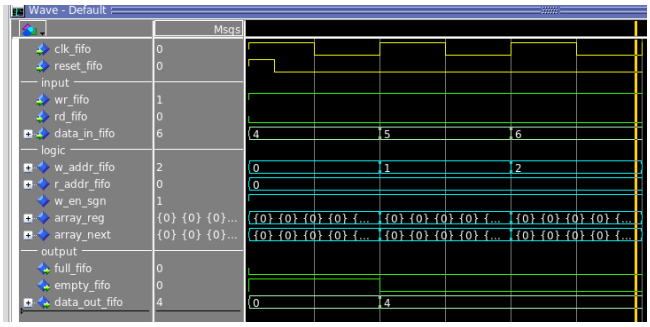
Nesta parte, estão as diferenças entre o contador binário e o LFSR durante a simulação. Foi observado a diferença na localização dos dados na memória, visualizando o armazenamento linear nas posições do contador binário e o armazenamento pseudoaleatório do contador LFSR.

A simulação foi realizada com 4 bits, ou seja, 16 endereços, demonstrando como os sinais de FULL e EMPTY são alterados durante as operações de escrita e leitura.

4.1.1 Com Contador Binário

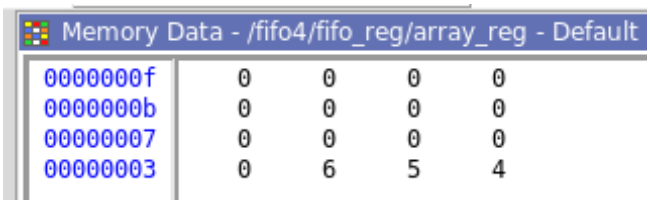
Inicialmente, na figura 23, são escritos os valores "4", "5"e "6". O contador binário armazena na ordem crescente, das posições de 0 até N, sendo N o número total de posições. Isso é visível na *Memory List*, figura 24, que demonstra essas posições sendo armazenadas na memória.

Figura 23: Simulação Funcional da FIFO - Armazenamento Incompleto - Binary Counter Controller



Fonte: Elaborada pela autora

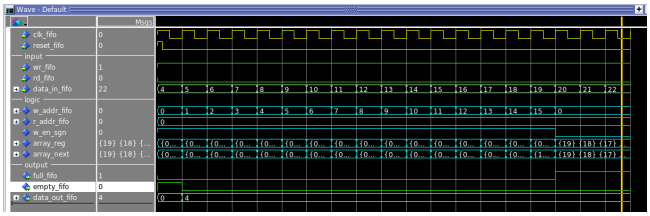
Figura 24: Visualização da Memória da FIFO - Armazenamento Incompleto - Binary Counter Controller



Fonte: Elaborada pela autora

Após isso, preenchendo todas as posições da FIFO (figura 25), podemos visualizar novamente na *Memory List* que os valores foram armazenados linearmente, conforme figura 26.

Figura 25: Simulação Funcional da FIFO - Armazenamento Completo - Binary Counter Controller



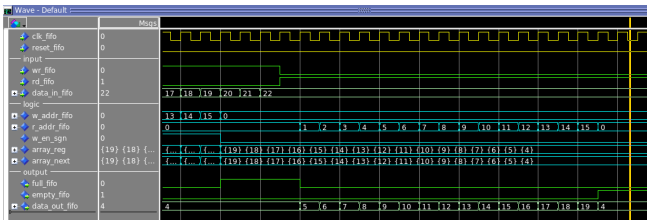
Fonte: Elaborada pela autora

Figura 26: Visualização da Memória da FIFO - Armazenamento Completo - Binary Counter Controller

Fonte: Elaborada pela autora

Por fim, na figura 27 removendo todos os dados e verificando que a saída `Empty_fifo` agora é '1', indicando que a FIFO está vazia. Note também o comportamento FIFO (First In First Out), onde os primeiros dados inseridos foram os primeiros removidos.

Figura 27: Simulação Funcional da FIFO - Armazenamento Vazio - Binary Counter Controller

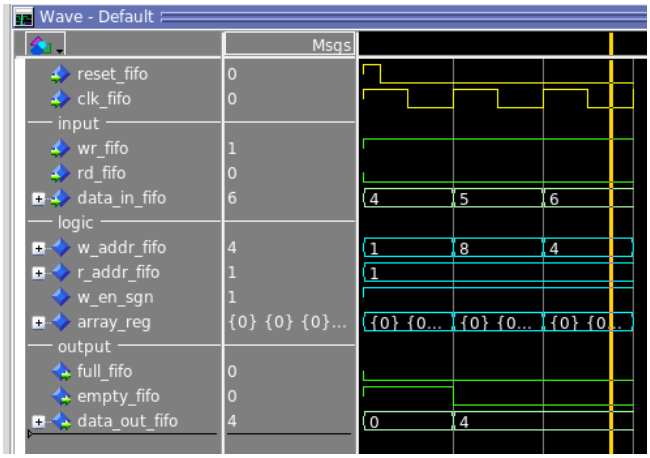


Fonte: Elaborada pela autora

4.1.2 Com Contador LFSR

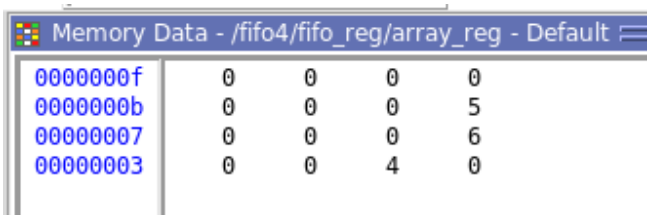
Inicialmente, na figura 28, são escritos os valores "4", "5"e "6". O contador LFSR armazena esses valores em uma ordem pseudoaleatória. Isso é visível na *Memory List* apresentada na figura 29, que mostra essas posições sendo armazenadas na memória.

Figura 28: Simulação Funcional da FIFO - Armazenamento Incompleto - Lookahead LFSR Controller



Fonte: Elaborada pela autora

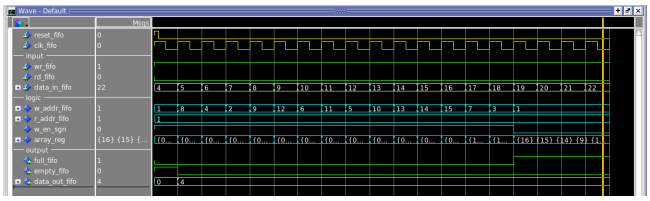
Figura 29: Visualização da Memória da FIFO - Armazenamento Incompleto - Lookahead LFSR Controller



Fonte: Elaborada pela autora

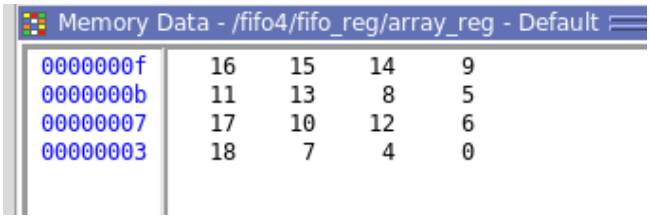
Após preencher todas as posições da FIFO (figura 30), podemos visualizar na *Memory List* que os valores foram armazenados sem seguir uma sequência linear, conforme mostrado na figura 31.

Figura 30: Simulação Funcional da FIFO - Armazenamento Completo - Lookahead LFSR Controller



Fonte: Elaborada pela autora

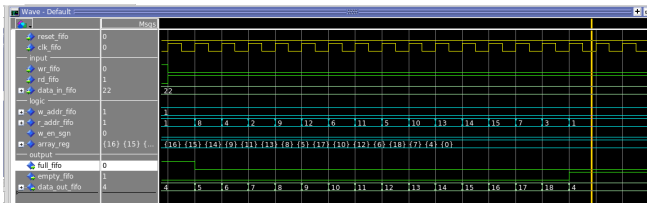
Figura 31: Visualização da Memória da FIFO - Armazenamento Completo - Lookahead LFSR Controller



Fonte: Elaborada pela autora

Por fim, na figura 32, ao remover todos os dados, verificamos que a saída `Empty_fifo` agora é '1', indicando que a FIFO está vazia. Note também o comportamento FIFO (First In First Out), onde os primeiros dados inseridos são os primeiros removidos.

Figura 32: Simulação Funcional da FIFO - Armazenamento Vazio - Lookahead LFSR Controller

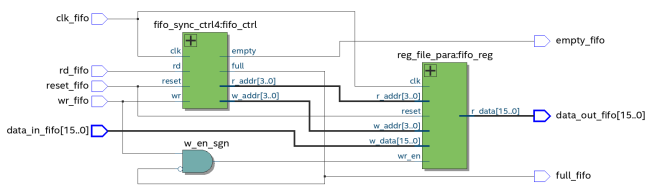


Fonte: Elaborada pela autora

4.2 RTL Viewer

No RTL, figura 33, é possível visualizar uma construção um pouco diferente do que já foi visto até então na disciplina. Os blocos em verde representam os componentes, o Register File e o controlador de FIFO.

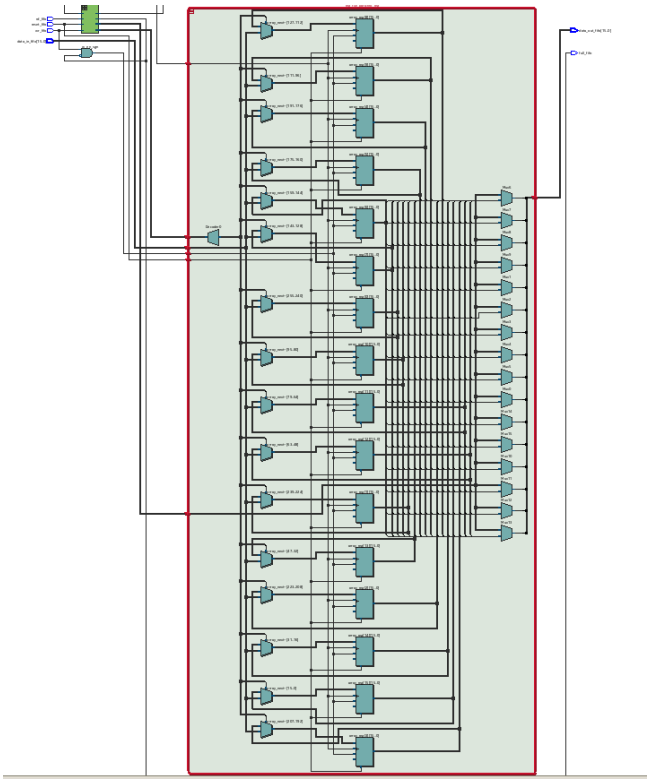
Figura 33: RTL Viewer da FIFO - Visão Geral



Fonte: Elaborada pela autora

Um detalhe interessante é que, quando forem feitas as análises da FIFO completa com diferentes arquiteturas do controlador, não será possível obter grandes mudanças visíveis neste RTL de imediato. Será necessário clicar no ícone *"+"* para expandir esse componente e, assim, verificar sua implementação interna. Isso foi feito com o Register File apenas para registro, conforme mostrado na figura 34.

Figura 34: RTL Viewer da FIFO - Visão do Register File



Fonte: Elaborada pela autora

4.3 Elementos Lógicos e Frequência Máxima com controladores diferentes na FIFO

A seguir, estão documentados o número de elementos lógicos e frequência máxima para as diferentes arquiteturas de controladores.

4.3.1 Enlarged Bin Counter

A seguir, os resultados das operações feitas no Quartus II para a arquitetura `enlarged_bin_arch` com 16 e 1024 endereços de armazenamento para o desenvolvimento da FIFO.

Elementos Lógicos da FIFO - Enlarged Bin Counter (16 Endereços) Na figura 35, documento o número de elementos lógicos para a arquitetura em questão com 16 endereços.

Figura 35: Elementos Lógicos do FIFO Controller - Enlarged Bin Counter (16 Endereços)

Total logic elements	462
Total registers	266
Total pins	38

Fonte: Elaborada pela autora

Desempenho da FIFO - Enlarged Bin Counter (16 Endereços) Na tabela 7, documento a frequência máxima obtida com o circuito para a arquitetura em questão com 16 endereços.

Tabela 7: Desempenho da FIFO - Enlarged Bin Counter (16 Endereços)

Condição	Temperatura	Frequência
Slow	100℃	184.13 MHz
Slow	-40℃	189.86 MHz
Fast	-40℃	332.12 MHz

Fonte: Elaborada pela autora

Elementos Lógicos da FIFO - Enlarged Bin Counter (1024 Endereços) Na figura 36, documento o número de elementos lógicos para a arquitetura em questão com 1024 endereços.

Figura 36: Elementos Lógicos do FIFO Controller - Enlarged Bin Counter (1024 Endereços)

Total logic elements	28,399
Total registers	16406
Total pins	38

Fonte: Elaborada pela autora

Desempenho da FIFO - Enlarged Bin Counter (1024 Endereços) Na tabela 8, documento a frequência máxima obtida com o circuito para a arquitetura em questão com 1024 endereços.

Tabela 8: Desempenho da FIFO - Enlarged Bin Counter (1024 Endereços)

Condição	Temperatura	Frequência
Slow	100℃	112.74 MHz
Slow	-40℃	117.16 MHz
Fast	-40℃	201. MHz

Fonte: Elaborada pela autora

4.3.2 Lookahed Bin Counter

A seguir, os resultados das operações feitas no Quartus II para a arquitetura `lookahed_bin_arch` com 16 e 1024 endereços de armazenamento para o desenvolvimento da FIFO.

Elementos Lógicos da FIFO - Lookahead Bin Counter (16 Endereços) Na figura 37, documento o número de elementos lógicos para a arquitetura em questão com 16 endereços.

Figura 37: Elementos Lógicos do FIFO Controller - Lookahead Bin Counter (16 Endereços)

Total logic elements	463
Total registers	266
Total pins	38

Fonte: Elaborada pela autora

Desempenho da FIFO - Lookahead Bin Counter (16 Endereços) Na tabela 10, documento a frequência máxima obtida com o circuito para a arquitetura em questão com 16 endereços.

Tabela 9: Desempenho da FIFO - Lookahed Bin Counter (16 Endereços)

Condição	Temperatura	Frequência
Slow	100℃	248.39 MHz
Slow	-40℃	255.56 MHz
Fast	-40℃	452.90 MHz

Fonte: Elaborada pela autora

Elementos Lógicos da FIFO - Lookahead Bin Counter (1024 Endereços) Na figura 38, documento o número de elementos lógicos para a arquitetura em questão com 1024 endereços.

Figura 38: Elementos Lógicos do FIFO Controller - Lookahead Bin Counter (1024 Endereços)

Total logic elements	28,406
Total registers	16406
Total pins	38

Fonte: Elaborada pela autora

Desempenho da FIFO - Lookahead Bin Counter (1024 Endereços) Na tabela 10, documento a frequência máxima obtida com o circuito para a arquitetura em questão com 1024 endereços.

Tabela 10: Desempenho da FIFO - Lookahed Bin Counter (1024 Endereços)

Condição	Temperatura	Frequência
Slow	100℃	128.34 MHz
Slow	-40℃	134.30 MHz
Fast	-40℃	226.3 MHz

Fonte: Elaborada pela autora

4.3.3 Lookahed LFSR Counter

A seguir, os resultados das operações feitas no Quartus II para a arquitetura `lookahed_LFSR_arch` com 16 e 1024 endereços de armazenamento para o desenvolvimento da FIFO.

Elementos Lógicos da FIFO - Lookahead LFSR Counter (16 Endereços) Na figura 39, documento o número de elementos lógicos para a arquitetura em questão com 16 endereços.

Figura 39: Elementos Lógicos do FIFO Controller - Lookahead LFSR Counter (16 Endereços)

Total logic elements	464
Total registers	266
Total pins	38

Fonte: Elaborada pela autora

Desempenho da FIFO - Lookahead LFSR Counter (16 Endereços) Na tabela 11, documento a frequência máxima obtida com o circuito para a arquitetura em questão com 16 endereços.

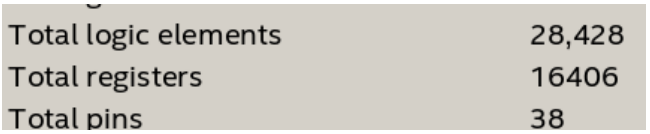
Tabela 11: Desempenho da FIFO - Lookahed LFSR Counter (16 Endereços)

Condição	Temperatura	Frequência
Slow	100℃	236.97 MHz
Slow	-40℃	244.56 MHz
Fast	-40℃	439.37 MHz

Fonte: Elaborada pela autora

Elementos Lógicos da FIFO - Lookahead LFSR Counter (1024 Endereços) Na figura 40, documento o número de elementos lógicos para a arquitetura em questão com 1024 endereços.

Figura 40: Desempenho da FIFO - Lookahed LFSR Counter (1024 Endereços)



Fonte: Elaborada pela autora

Desempenho da FIFO - Lookahead LFSR Counter (1024 Endereços) Na tabela 12, documento a frequência máxima obtida com o circuito para a arquitetura em questão com 1024 endereços.

Tabela 12: Desempenho da FIFO - Lookahed LFSR Counter (1024 Endereços)

Condição	Temperatura	Frequência
Slow	100℃	139.65 MHz
Slow	-40℃	144.30 MHz
Fast	-40℃	242.54 MHz

Fonte: Elaborada pela autora

5 Conclusão

Portanto, ao considerar a implementação de um contador no projeto, é fundamental avaliar se a utilização de um contador LFSR pode oferecer vantagens em termos de eficiência em comparação com um contador binário. Os resultados mostraram que, tanto no circuito isolado do controlador quanto na FIFO completa, o contador LFSR demonstrou um desempenho superior ao do contador binário.

Embora o contador binário seja frequentemente utilizado devido à sua lógica simples e familiaridade, a adoção do contador LFSR pode ser mais vantajosa. O contador binário, por ser mais simples, é o padrão quando se pensa em contadores. No entanto, os benefícios em eficiência e desempenho do contador LFSR justificam a escolha desta alternativa menos convencional.

Adicionalmente, a experiência adquirida com a estrutura de dados FIFO (First In, First Out) e a construção de um projeto hierárquico proporcionaram uma compreensão mais profunda dos princípios de implementação. O processo envolveu a criação e teste de componentes individuais antes de

integrá-los ao circuito principal, o que se revelou fundamental para a obtenção de uma solução eficiente e funcional.

Além disso, houve a utilização de uma nova ferramenta do *Quartus II* para a análise da memória, a *Memory List*, permitiu uma visualização do comportamento memória durante a simulação funcional.

Referências

AMD. *Linear Feedback Shift Register*. 1996. AMD Technical Documentation. Disponível em: <https://docs.amd.com/v/u/en-US/xapp052>. Acesso em: 25 Jul 2024.