



# ***Barramento Avalon***

Felipe Portavales Goldstein – RA023772

portavales@gmail.com

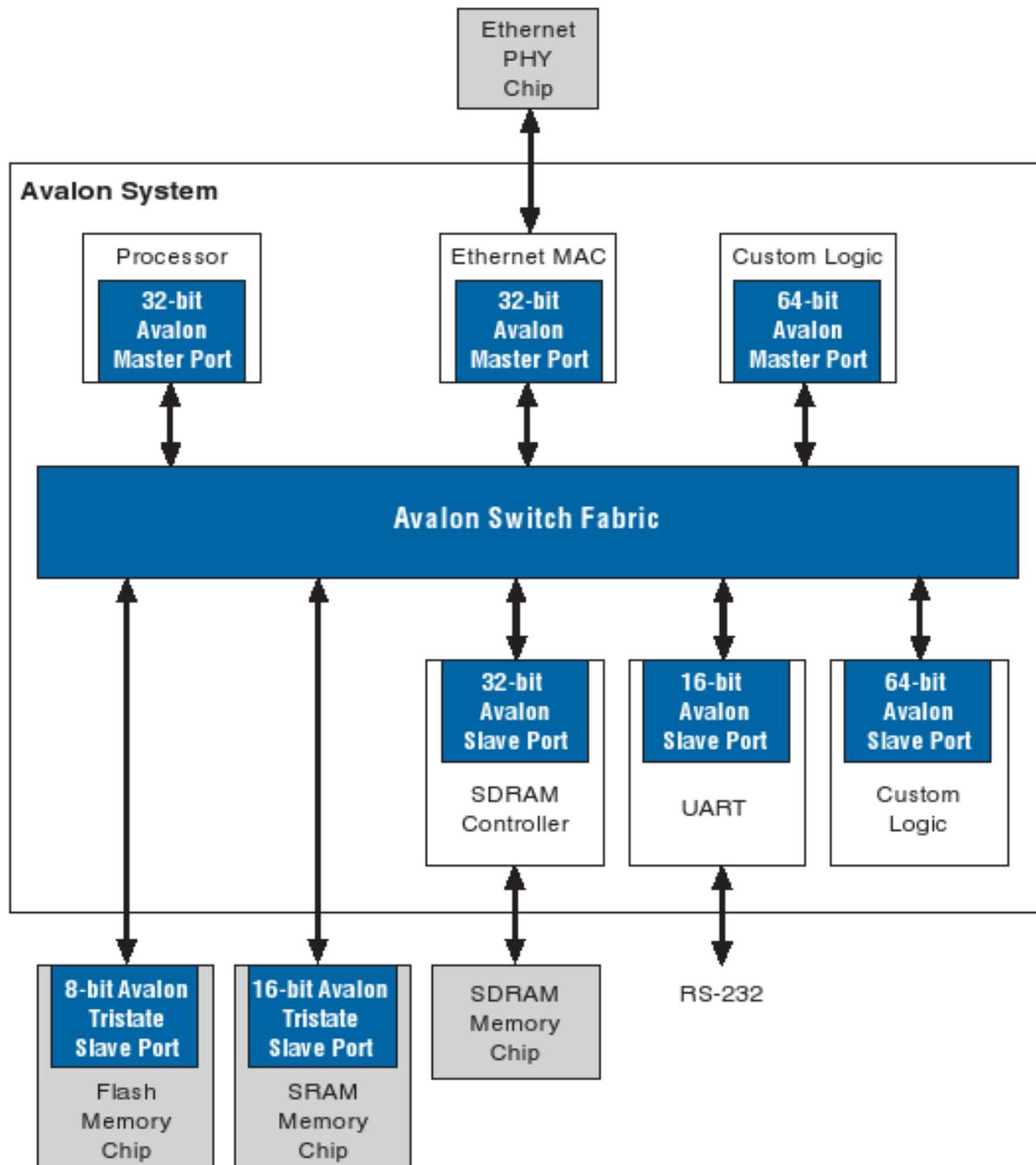
## **Design Goals**

- **Simplicidade** - Protocolo fácil de entender e implementar
- **Pouca utilização de recursos** – Economia de elementos lógicos da FPGA
- **Operação Síncrona** – Fácil integração com outros módulos e evita problemas de temporização
- **Flexibilidade** – Opera em diferentes larguras de dados e modos de transferência

## Características

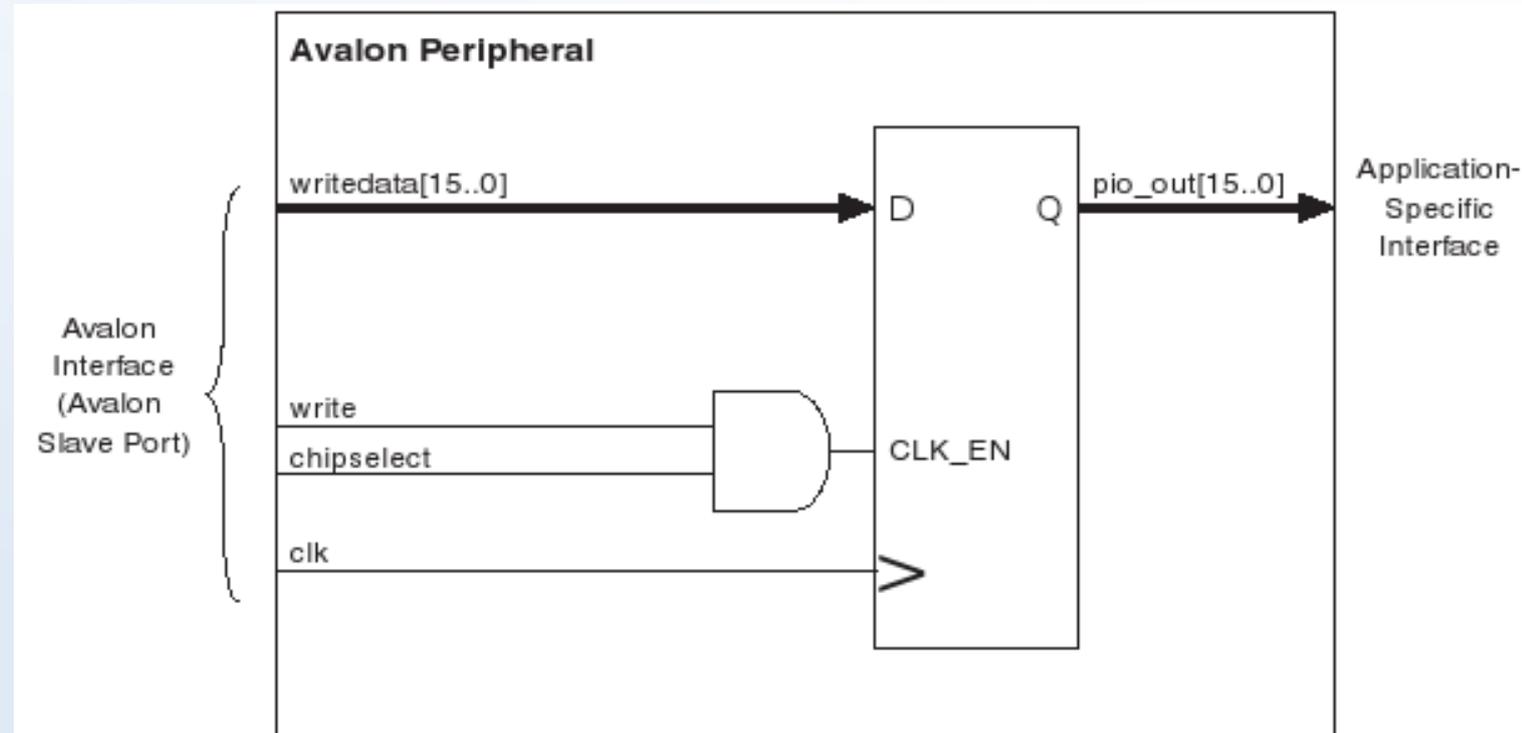
- Endereço, Dado e Controle em linhas separadas
  - Simplifica o projeto do periférico. Todos disponíveis no mesmo ciclo.
- Largura de dados de até 128-bits
  - Dimensionamento dinâmico de largura: Automaticamente manipula a transferência de dados entre dispositivos com portas de tamanhos diferentes.
- Múltiplos Mestres
  - Múltiplos periféricos mestres podem estar ligados ao mesmo barramento e a lógica de arbítrio é gerada automaticamente.
- Decodificação de Endereço
  - A decodificação de endereço é feita pelo barramento e o periférico só precisa verificar o sinal **Chip Select**.
- Alta Performance
  - Capaz de transferir um dado por clock

# Barramento Avalon



## Interface com o barramento

- Exemplo simples
  - Sinais básicos para leitura do barramento
    - **writedata[15..0]**
    - **write**
    - **chipsselect**
    - **clk**



## Transferências (escravo)

- **Sinais**

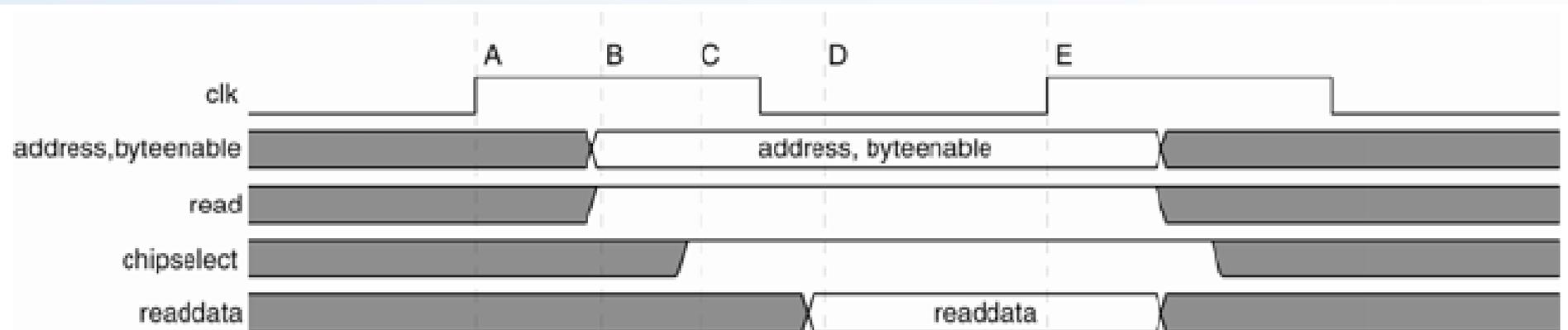
- **address** [ 32 bits ]
  - endereço a ser acessado
- **readdata & writedata** [1 a 128 bits ]
  - dados associados com a transferência
- **chipselect** [1 bit]
  - indica o periférico a ser acessado
- **read & write** [1 bit]
  - Indica se a transferência é uma leitura ou uma escrita
- **byteenable & writebyteenable** [4bits]
  - Indica qual byte de *writedata* deve ser escrito
- **begintransfer** [1bit]
  - Usado para indicar que uma nova transferência foi iniciada (sua interpretação é específica da implementação)

# Transferências (escravo)

- **Leitura básica**
  - **Termina em 1 ciclo**
  - **Throughput máximo**
    - Uma transferência por ciclo
  - ***readdata* com dado correto deve estar disponível na subida de clock seguinte**

## Transferências (escravo)

- A) Primeiro ciclo começa na borda de subida do clock (*clk*)
- B) *address* e *read* estabilizam
- C) O barramento decodifica o endereço e *chipselect* estabiliza.
- D) O periférico escravo disponibiliza os dados válidos em *readdata*
- E) O barramento captura os dados em *readdata* na subida do próximo clock. O próximo ciclo começa aqui e pode iniciar uma nova transferência.



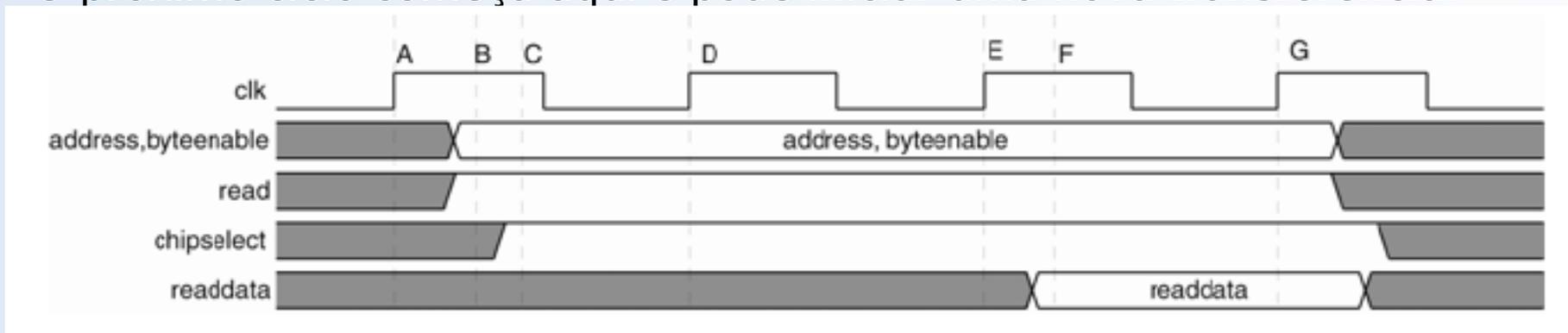
# Transferências (escravo)

- **Leitura com *wait states* fixos**
  - **Somente na transferência de leitura**
  - **Pre-determinado em tempo de projeto**
  - **Termina em  $n$  ciclos ( $n$  *wait states*)**
  - **Throughput é afetado**
    - Throughput máximo com 1 *wait state* é uma transferência a cada 2 ciclos
  - ***readdata* com dado correto deve estar disponível na subida do *next* clock seguinte**



## Transferências (escravo)

- A) Primeiro ciclo começa na borda de subida do clock (*clk*)
- B) *address* e *read* estabilizam
- C) O barramento decodifica o endereço e *chipsselect* estabiliza.
- D) Subida do clock marca o fim do primeiro e ciclo de *wait-state*. Módulo síncrono escravo captura os sinais *address*, *read* e *chipsselect*.
- E) Subida do clock marca o fim do segundo ciclo de *wait-state*.
- F) O módulo escravo disponibiliza os dados válidos em *readdata*, antes do fim do terceiro ciclo de clock.
- G) O barramento captura os dados em *readdata* na subida do próximo clock. O próximo ciclo começa aqui e pode iniciar uma nova transferência.



# Transferências (escravo)

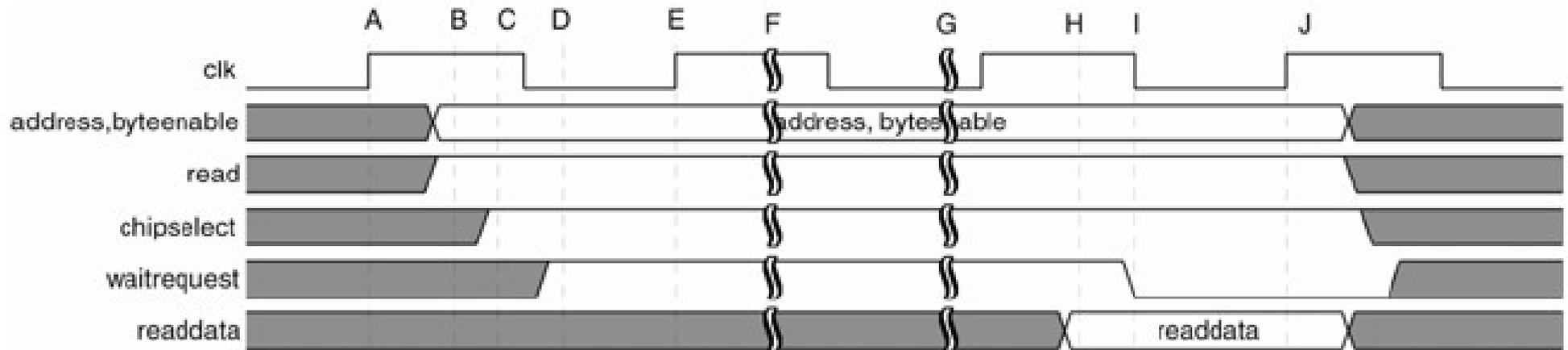
- **Leitura com *wait states* variáveis**
  - Permite fazer o barramento esperar quantos ciclos forem necessários para disponibilizar os dados em *readdata*
  - Utiliza-se o sinal ***waitrequest***
  - Escravo faz ***waitrequest* = 1** e o barramento espera até que em alguma subida de clock o sinal *waitrequest* esteja = 2
  - O mestre fica “travado” enquanto o *waitrequest* estiver = 1

# Transferências (escravo)

- A) Primeiro ciclo começa na borda de subida do clock (*clk*)
- B) *address* e *read* estabilizam
- C) O barramento decodifica o endereço e *chipselect* estabiliza.
- D) Sinal *waitrequest* deve estabilizar antes do fim do primeiro ciclo.
- E) O barramento detecta o pedido de espera pelo *waitrequest = 1* e *readdata* ainda não é lido pelo barramento
- F) e G) Enquanto *waitrequest = 1* passam-se um número indefinido de clocks
- H) Escravo disponibiliza dados válidos em *readdata*.
- I) Escravo faz *waitrequest = 0*
- J) O barramento captura os dados em *readdata* na subida do próximo clock. A transferência acabou aqui e neste ciclo pode-se iniciar uma nova transferência.

(VEJA FIGURA NO SLIDE SEGUINTE)

## Transferências (escravo)



# Transferências (escravo)

- **Leitura com *setup-time***

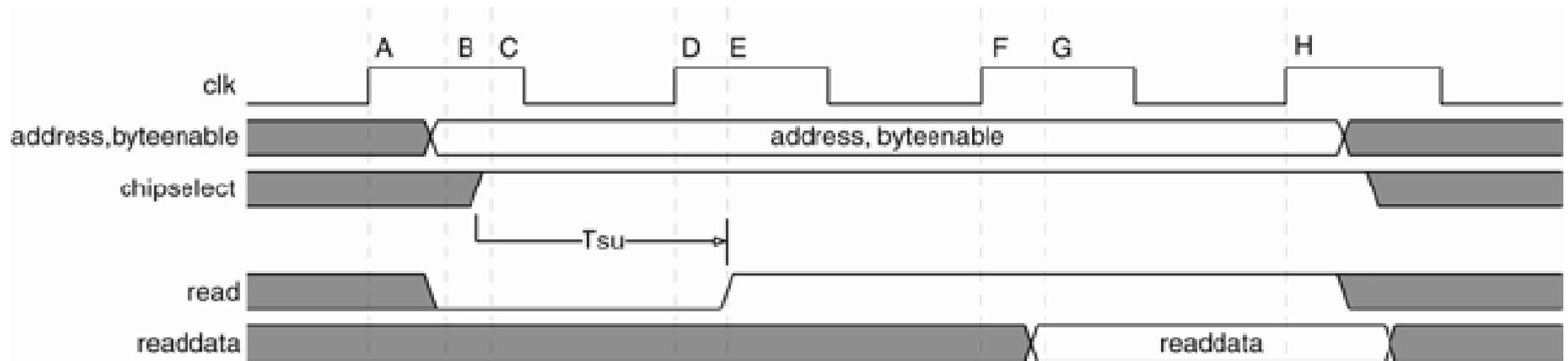
- Geralmente usado em periféricos assíncronos fora do chip (ligados pelos pinos de I/O da FPGA)
- Necessidade de que os sinais *address* e *chipselect* fique estáveis por um período determinado de tempo antes que o sinal *read* “suba”.
- Pode ser combinado com *waitstates*

# Transferências (escravo)

- A) Primeiro ciclo começa na borda de subida do clock (*clk*). O primeiro e único ciclo de *setup-time* começa aqui.
- B) *address* e *byteenable* estabilizam mas *read* ainda não é ativado
- C) O barramento decodifica o endereço e *chipselect* estabiliza.
- D) Subida do clock marca o fim do ciclo de *setup-time* (*Tsu*) e o início do ciclo de *wait-state*.
- E) O barramento ativa o sinal *read*
- F) Subida do clock marca o fim do ciclo de *wait-state*.
- G) O módulo escravo disponibiliza os dados válidos em *readdata*
- H) O barramento captura os dados em *readdata* na subida do próximo clock. O próximo ciclo começa aqui e pode iniciar uma nova transferência.

(VEJA FIGURA NO SLIDE SEGUINTE)

## Transferências (escravo)

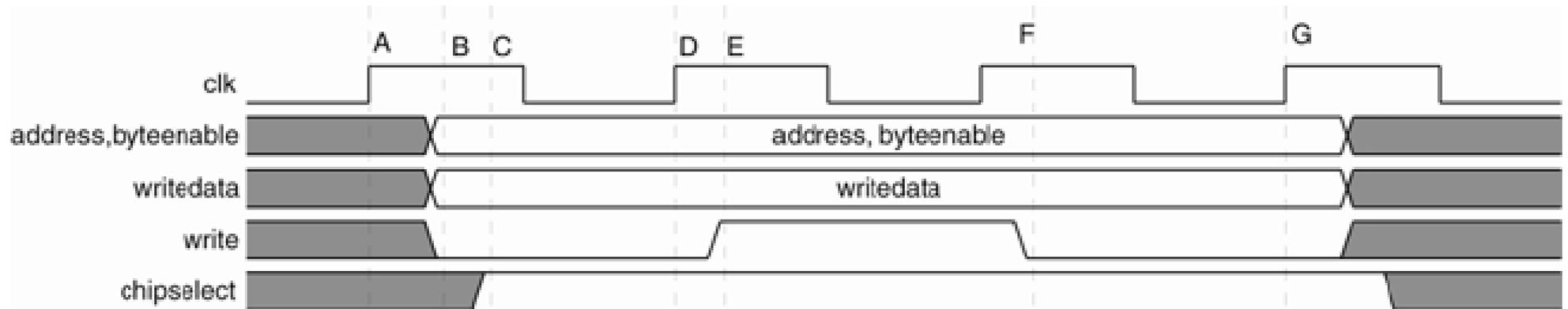


# **Transferências (escravo)**

- **Escrita se dá de modo análogo à leitura, trocando-se o sinal *read* por *write* e *readdata* por *writedata***

## Transferências (escravo)

- **Escrita com *setup-time* e *hold-time***
  - Exemplo abaixo: 1 ciclo de setup-time e 1 ciclo de hold-time



# Transferências (mestre)

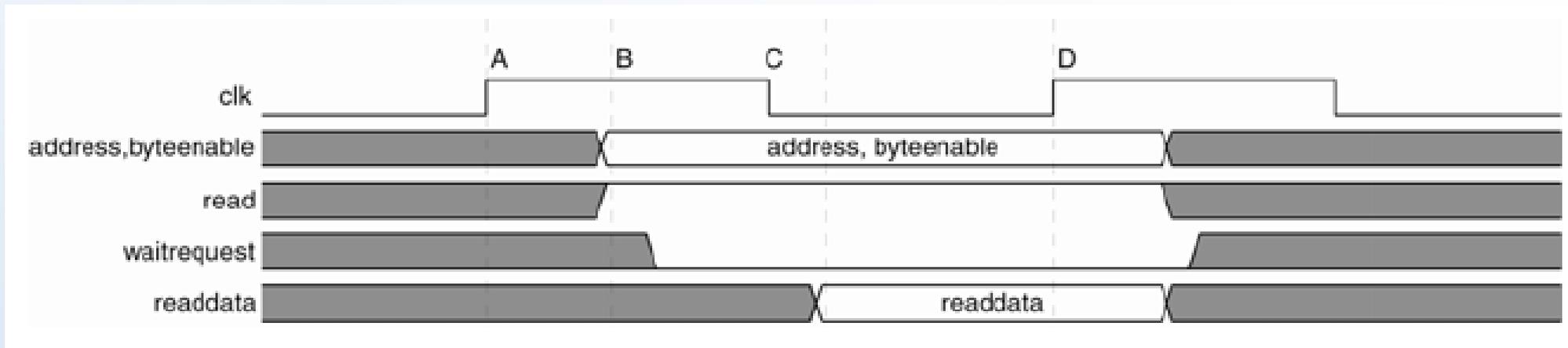
- O sinal *waitrequest* é uma entrada e deve ser obedecido sempre!
- No começo de cada transferência o mestre muda os sinais apropriados e espera até que o barramento faça o sinal *waitrequest* = 0
- Por definição todas as transferências no mestre usam o sinal *waitrequest*, portanto toda transferência no mestre é feita em um número variável de *wait-states*.
- *Setup* e *Hold-time* são transformados pelo barramento para sinalizar o *waitrequest* de forma apropriada para o mestre.

# Transferências (mestre)

- O sinal *address* representa o endereço em bytes.
- Endereços devem ser alinhados pelo tamanho da palavra de dados. Para um tamanho de dados de 32 bits, o barramento ignora os 2 últimos bits do endereço.
- Para acessar um byte específico, pode ser usado o sinal *byteenable*.
- O tamanho das portas *readdata* e *writedata* devem ser 8, 16, 32, 64 ou 128 bits e para as duas portas, deve ser igual

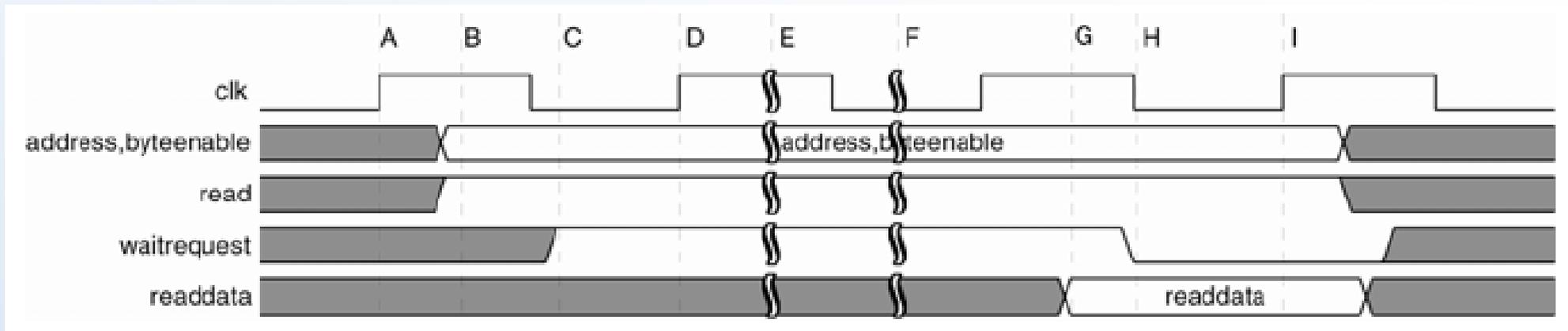
## Transferências (mestre)

- **No mestre a leitura é semelhante ao escravo**
  - Leitura sem wait-states



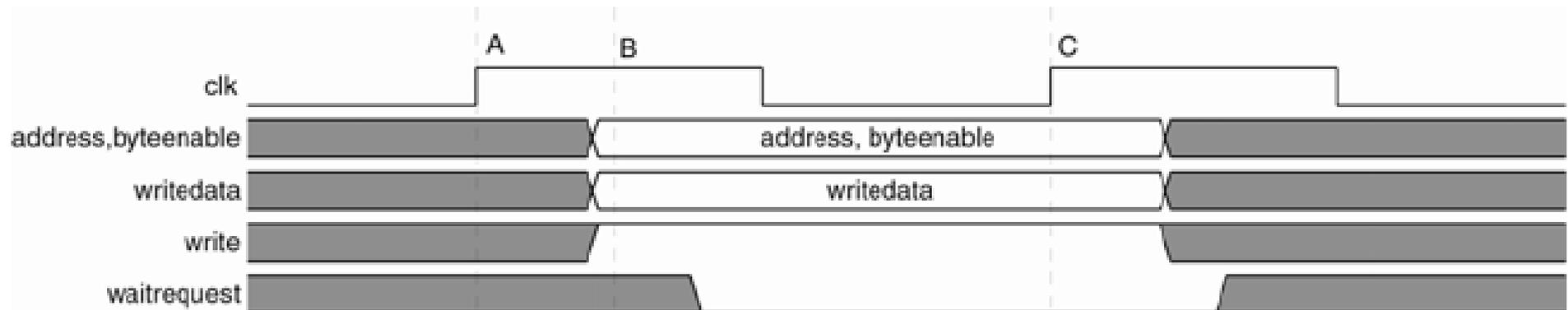
## Transferências (mestre)

- No mestre a leitura é semelhante ao escravo
  - Leitura com *wait-states*



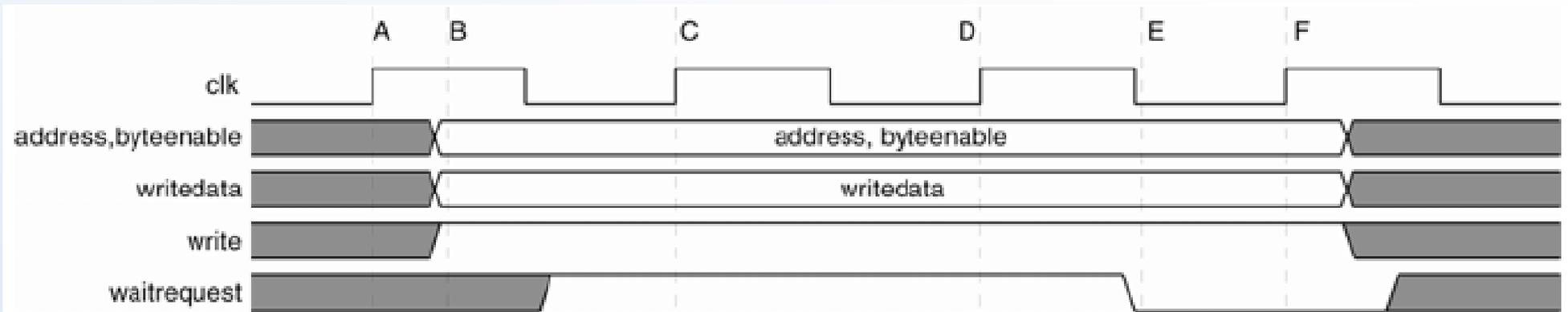
## Transferências (mestre)

- Escrita sem wait-states



## Transferências (mestre)

- Escrita com wait-states

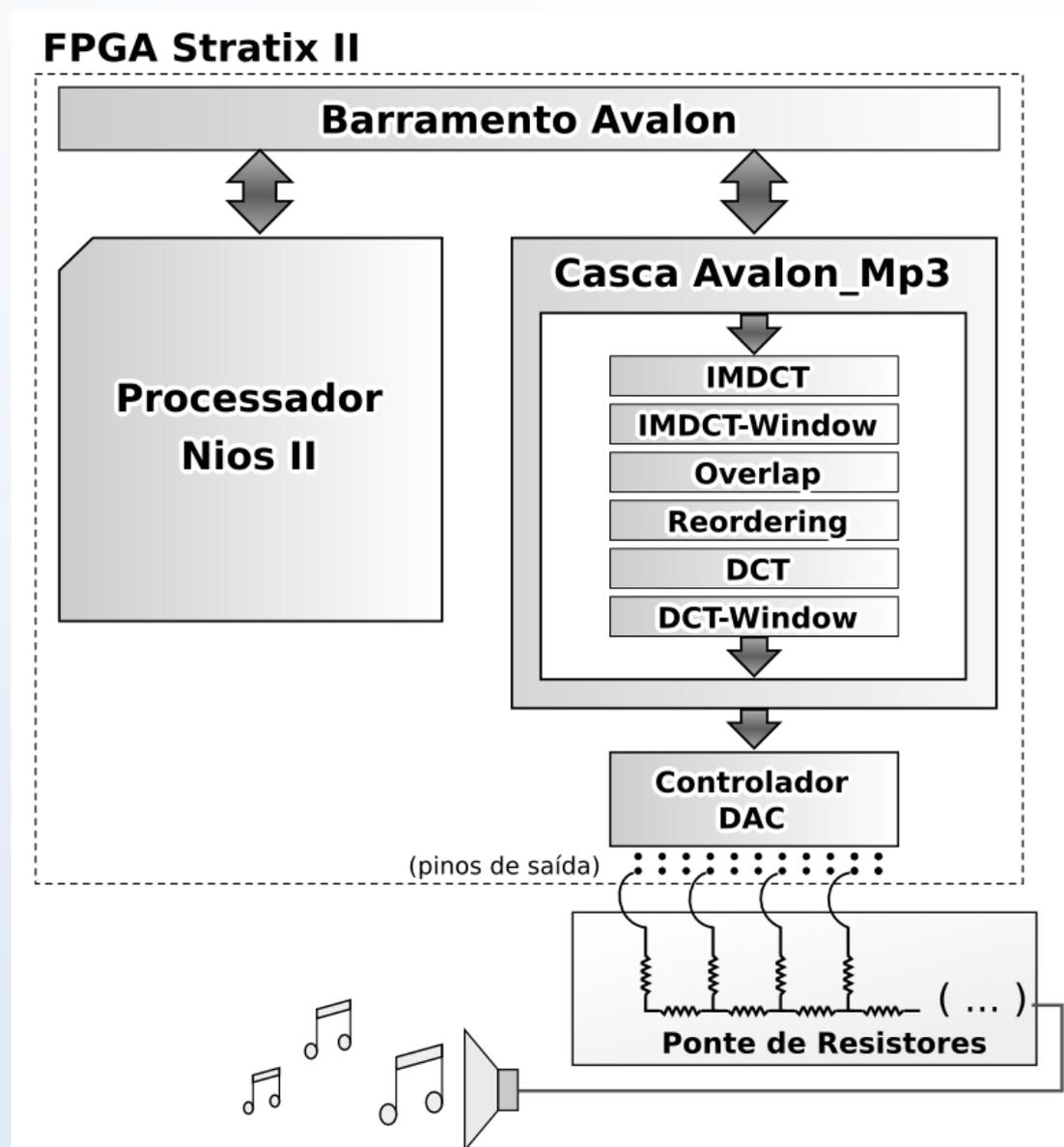


## **Caso de Uso – um Decodificador de MP3**

- Parte do processo de decodificação é feito em hardware dedicado, implementado como um periférico do Nios 2, interligado pelo barramento Avalon.
- Um software rodando no Nios faz a leitura da stream MP3, realiza uma parte da decodificação, copia os dados parcialmente decodificados para o periférico que vai terminar de decodificar e enviar para um conversor DAC.
- O hardware dedicado é responsável por realizar a parte mais “pesada” do processo de decodificação.
- A transferência dos dados para o periférico é feita como se fosse uma cópia de memória:
  - `int* endereco_periferico = 0x0A7809;`  
`(*endereco_periferico) = sample_mp3_parcialmente_decodificado;`

# Barramento Avalon

## Caso de Uso - um Decodificador de MP3



```

module avalon_mp3(clk, reset_n, address, read, readdata, write, writedata, chipselect);
    input clk;
    input reset_n;
    input [2:0] address;
    input read;
    input write;
    input [31:0] writedata; //output data to avalon bus
    input chipselect;
    output [31:0] readdata; // input data from avalon bus
    reg [31:0] readdata;
    wire in_valid;
    wire out_requested;

    wire[15:0] in_data; //input data from mp3 decoder module
    reg out_valid;
    reg in_request;
    reg[31:0] out_data; //output data to mp3 decoder module

    always @(posedge(clk))
        begin :prc_avalon
            if (!reset_n) begin
                in_request <= 1'd0;
                out_valid <= 1'd0;
                out_data <= 32'd0;
            end
            else begin
                in_request <= 1'd0;
                out_valid <= 1'd0;
                if (chipselect) begin
                    if (read) begin
                        case (address)
                            3'd0: begin // Can software write data to Module ?
                                readdata <= {31'b0, out_requested};
                            end
                            3'd1: begin // Can software read PCM data from Module ?
                                readdata <= {31'b0, in_valid};
                            end
                            3'd2: begin // Read PCM data from Module
                                readdata <= {{ 16 {in_data[15]}}, in_data};
                                in_request <= 1'd1;
                            end
                        endcase
                    end
                    if (write) begin
                        out_data <= writedata;
                        out_valid <= 1'd1;
                    end
                end
            end
        end
        hw_decode decode0(.in_request(out_requested), .in_valid(out_valid), .in_data_value(out_data),
            .out_requested(in_request), .out_valid(in_valid), .out_data(in_data), .RSTN(reset_n), .CLK(clk));
endmodule

```

# Barramento Avalon

## Configuração do Avalon no SOPC builder

The screenshot shows the 'Component Editor - avalon\_mp3' window with the 'Interfaces' tab selected. The configuration is for an 'avalon slave "avalon\_slave\_0" (1 of 1)'. The 'Name' field is 'avalon\_slave\_0' and the 'Type' is 'avalon slave'. Under 'Avalon Slave Settings', 'Slave addressing' is set to 'Registers (use native bus sizing)', 'Minimum Arbitration Shares' is 1, and 'Can receive stderr/stdout' is 'No'. Under 'Avalon Slave Timing', 'Setup' is 0, 'Read Wait' is 1, 'Write Wait' is 1, and 'Hold' is 0, with 'Units' set to 'cycles'. Under 'Pipelined Transfers', 'Read Latency' is 0 and 'Max Pending Read Transactions' is 1. The 'Assert chipselect through read latency' checkbox is unchecked. At the bottom, there are buttons for 'Add Interface' and 'Remove Interfaces With No Signals'. A status bar at the bottom indicates 'Component "avalon\_mp3" is ok.' and navigation buttons for '< Prev', 'Next >', and 'Finish...'.

Component Editor - avalon\_mp3

File Templates

Introduction HDL Files Signals Interfaces SW Files Component Wizard

▶ About Interfaces

▼ avalon slave "avalon\_slave\_0" (1 of 1)

Name:

Type:

▼ Avalon Slave Settings

Slave addressing:

Minimum Arbitration Shares:

Can receive stderr/stdout:

▼ Avalon Slave Timing

Setup:  Read Wait:  Hold:  Units:

Write Wait:

Pipelined Transfers

Read Latency:  Max Pending Read Transactions:

Assert chipselect through read latency

Read Waveforms

OK Component "avalon\_mp3" is ok.

< Prev Next > Finish...

# Barramento Avalon

## Configuração do Avalon no SOPC builder

The screenshot shows the 'Component Editor - avalon\_mp3' window with the 'Interfaces' tab selected. The window displays two waveform diagrams: 'Read Waveforms' and 'Write Waveforms'. Both diagrams show signals for clock (clk), address (A0), data (DO), chipselect, and waitrequest. The 'Read Waveforms' diagram shows the read\_n signal and readdata signal. The 'Write Waveforms' diagram shows the write\_n signal and writedata signal. The timing granularity is noted as 'System Clock cycles'. At the bottom of the window, there are buttons for 'Add Interface' and 'Remove Interfaces With No Signals'. A status bar at the bottom indicates 'Component "avalon\_mp3" is ok.' and navigation buttons for '< Prev', 'Next >', and 'Finish...'.

Component Editor - avalon\_mp3

File Templates

Introduction HDL Files Signals **Interfaces** SW Files Component Wizard

**Read Waveforms**

clk  
address A0  
read\_n  
chipselect  
readdata DO  
waitrequest

**Write Waveforms**

clk  
address A0  
writedata DO  
write\_n  
chipselect  
waitrequest

Timing granularity is System Clock cycles.

Add Interface Remove Interfaces With No Signals

OK Component "avalon\_mp3" is ok.

< Prev Next > Finish...

# Barramento Avalon

## Configuração do Avalon no SOPC builder

The screenshot shows the 'Component Editor - avalon\_mp3' window. The 'Interfaces' tab is selected, and the 'Avalon Slave Timing' section is expanded. The configuration parameters are as follows:

- Setup: 0
- Read Wait: 1
- Write Wait: 1
- Hold: 0
- Units: cycles
- Pipelined Transfers: Read Latency: 2, Max Pending Read Transactions: 1
- Assert chipselect through read latency

The 'Read Waveforms' section displays a timing diagram with signals: clk, address (A0), read\_n, chipselect, readdata (D0), and waitrequest. The 'Write Waveforms' section shows a partial timing diagram for clk and address (A0).

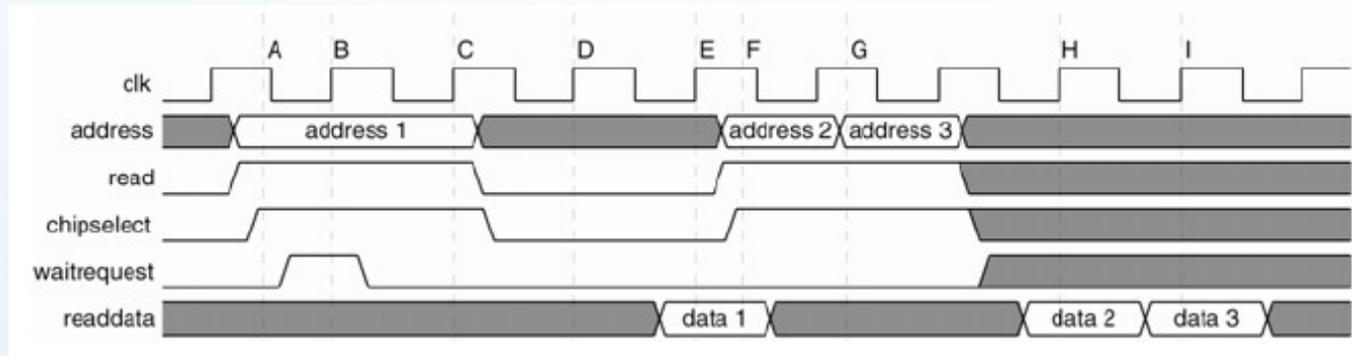
Buttons at the bottom of the configuration area are 'Add Interface' and 'Remove Interfaces With No Signals'.

A status bar at the bottom indicates: **OK** Component "avalon\_mp3" is ok.

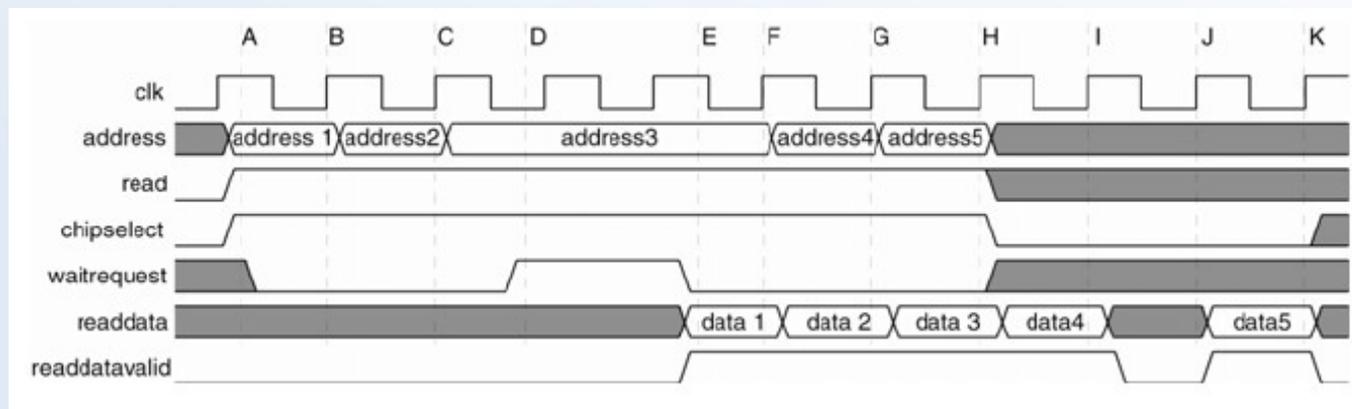
Navigation buttons at the very bottom are '< Prev', 'Next >', and 'Finish...'.

## Outras funcionalidades avançadas

- Leitura em modo pipe-line com latência fixa (escravo)

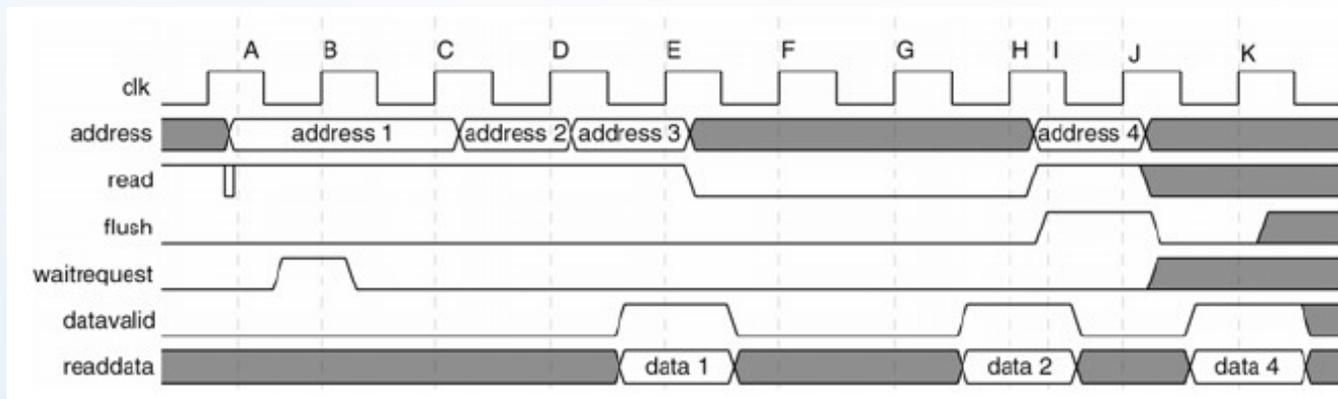


- Leitura em modo pipe-line com latência variável (escravo)



## Outras funcionalidades avançadas

- Leitura em modo pipe-line (mestre)



## **FIM**

Felipe Portavales Goldstein – RA023772

portavales@gmail.com