RESUMO

O crescente aumento da necessidade de se colocar sistemas complexos inteiros dentro de um único chip para atender a demanda de criação de dispositivos cada vez menores, com mais funcionalidades e que precisam ser desenvolvidos cada vez mais rápido, torna necessário o uso de novas metodologias e técnicas de desenvolvimento e validação de sistemas. Barramentos são os elementos que interligam os dispositivos de um sistema. Para se aumentar a eficiência e rapidez no desenvolvimento de sistemas simulados, existe a necessidade do desenvolvimento de mecanismos que facilitem a criação, o uso e o teste de barramentos. Esse trabalho propõe um *framework*, modelado em alto nível (TLM) e baseado na linguagem SystemC, para auxiliar a criação de simuladores de barramentos. Esse trabalho descreve, detalhadamente, todas as classes e interfaces que compõem o *framework* proposto. Quatro barramentos, AMBA, Avalon, Wishbone e Coreconnect, foram estudados e são descritos nesse documento. Para dois dos barramentos estudados, AMBA-AHB e Avalon, foram desenvolvidos simuladores baseados no *framework* proposto. Como os simuladores para os barramentos AHB e Avalon são completamente funcionais e executáveis, esse trabalho também descreve, demonstra e analisa os resultados de experimentos executados com ambos os barramentos.

ABSTRACT

The system-on-chip era is creating new challenges to the system design. There is an increasing demand for smaller electronic devices with more features and reduced time to market. To face these new challenges is necessary to introduce new methodologies and development techniques. Buses are important elements for connecting devices in a complex system. To increase the efficiency and speed of systems development, it is important to introduce new mechanisms to help the creation and tests of buses. This document presents a framework based on SystemC language and implemented using the transaction level modeling (TLM). The framework goal is to help designers to create bus simulators. This document describes in detail all framework classes and interfaces. Four buses, Wishbone, Coreconnect, AMBA and Avalon are described along the text. Two specific buses, AMBA-AHB and Avalon, were fully implemented and have executable simulators. Tests were performed using these simulators, the test results and analysis are described in the end of this document.

ÍNDICE

Lista de Figuras

Lista de Tabelas

1 Introdução

O crescente aumento da necessidade de se colocar sistemas complexos inteiros dentro de um único *chip* para atender a demanda de criação de dispositivos cada vez menores, com mais funcionalidades e que precisam ser desenvolvidos cada vez mais rápido, torna necessário o uso de novas metodologias e técnicas de desenvolvimento e validação de sistemas. A modelagem de alto nível é uma opção para atender esses requisitos.

Com a modelagem em alto nível é possível se fazer a validação e viabilização de um projeto a nível sistêmico deixando-se os detalhes de lado. Esses detalhes podem ser trabalhados depois do sistema validado,.

Juntamente com a modelagem de alto nível é interessante a geração de protótipos executáveis para se ter um modelo funcional do sistema já na fase inicial do projeto. Com esse modelo funcional e executável do sistema é possível que se comece o desenvolvimento e testes do software que será executado no futuro hardware. Dessa maneira, é possível identificar e corrigir os problemas, tanto de hardware como de software, nas fases iniciais. Essa técnica reduz os custos, reduz a necessidade de retrabalhos e o tempo total de desenvolvimento. SystemC é uma boa alternativa para atender essas necessidades.

Com SystemC é possível se fazer modelagem em nível de transação e gerar simuladores executáveis de um sistema. Além do mais, tem a vantagem de ser desenvolvido e ter a mesma sintaxe do C++, que é amplamente conhecido, facilitando assim a sua aprendizagem e utilização.

Para os sistemas complexos desenvolvidos atualmente, na maioria da vezes, é necessário a presença de pelo menos um processador. ArchC é boa alternativa para se descrever e simular processadores.

Com a linguagem de descrição de arquitetura ArchC é possível gerar simuladores executáveis de processadores que são componentes importantes na prototipação e simulação de sistemas complexos.

Barramentos são os elementos que interligam os dispositivos de um sistema. Da mesma maneira, para se aumentar a eficiência e rapidez no desenvolvimento de sistemas simulados, existe a necessidade do desenvolvimento de mecanismos que facilitem a criação de barramentos. Esse trabalho propõe um *framework,* baseado na linguagem SystemC, para auxiliar a criação de simuladores de barramentos.

Nesse trabalho é feito um estudo dos barramentos AMBA, Avalon, Wishbone e Coreconnect que serviram como modelos base para o desenvolvimento das classes e interfaces genéricas que compõem o *framework* de simulação de barramentos. Desses, dois barramentos, o AMBA e o Avalon, foram mais profundamente estudados e descritos. O AMBA foi escolhido por ser um barramento complexo e, provavelmente, o mais usado atualmente devido a ampla utilização dos processadores ARM em sistemas dedicados. Avalon é um barramento simples, escolhido para se garantir que o *framework* seja maleável o suficiente para se enquadrar na definição de barramentos simples e complexos.

Pelas mesmas razões descritas acima, também foram desenvolvidos os simuladores para os barramentos AMBA-AHB e Avalon. Todas as classes e interfaces usadas nos simuladores de ambos os barramentos são descritas, detalhadamente, nesse trabalho.

Como os simuladores dos barramentos AHB e Avalon são completamente funcionais e executáveis, esse trabalho também descreve, demonstra e analisa os resultados de experimentos executados com ambos os barramentos.

Essa dissertação está organizada da seguinte forma: no capítulo 2 são levantados os conceitos básicos, SystemC, TLM e ArchC, que foram fundamentais para o desenvolvimento de todo o trabalho. No capítulo 3 são descritas as especificações dos barramentos AMBA, Avalon, Wishbone e Coreconnect. No capítulo 4 são mostradas as implementações de dois barramentos, o AMBA-AHB e o Avalon. No capítulo 5, são apresentados resultados e análises de simulações dos barramentos AHB e Avalon. Finalmente, no capítulo 6, são apresentadas conclusões e sugestões de trabalhos futuros.

2 Conceitos Básicos

Esse capitulo cobre os conceitos básicos que serão utilizados no decorrer do texto, a biblioteca de classes SystemC, o nível de abstração TLM e a linguagem de descrição de arquitetura ArchC.

A biblioteca de classes de simulação SystemC foi um elemento essencial para o desenvolvimento dos modelos de simulação de barramentos porque fornece os elementos básicos de simulação utilizados por esses modelos.

Com o uso de TLM é possível aumentar as velocidades de simulação substituindo os modelos e interfaces com precisão de pino por modelos ao nível de transação.

A linguagem de descrição de arquitetura ArchC descreve arquiteturas de processadores que são mestres típicos de barramentos. Os modelos de simulação de barramentos desenvolvidos integram-se aos modelos de processadores descritos pelo ArchC.

2.1 SystemC

SystemC [1] é uma biblioteca de classes em C++ desenvolvida para auxiliar a criação de modelos de software usados para definir arquiteturas de hardware. Essa biblioteca, juntamente com C++, pode ser usada para criar modelos em nível de transação dos sistemas desejados, para fazer simulações de validação e para otimização.

A biblioteca de classes SystemC implementa os modelos necessários para a arquitetura de um sistema incluindo temporização, concorrência, comportamento reativo e suporta modelagem em nível de transação.

SystemC possui diversas estruturas tais como: módulos, processos, portas, sinais, *clocks*, etc, que facilitam a descrição de hardware, software e interfaces em C++. Um módulo em SystemC é um *container* que pode ter outros módulos, processos e portas. Um processo descreve uma funcionalidade dentro de um módulo. Portas são usadas para a interligação entre módulos e podem ser unidirecionais ou bidirecionais. O sinal é o responsável pela transmissão da informação da porta fonte à porta destino as quais ele está conectado. *Clock* em SystemC é um tipo especial de sinal usado para sincronização e temporização durante a simulação.

A biblioteca do SystemC possui uma ampla variedade de classes e estruturas para contemplar uma grande variedade de sistemas de simulação. As seções seguintes descrevem as classes e estruturas mais importantes que foram utilizadas para o desenvolvimento dos modelos de simulação de barramentos.

2.1.1 Módulos

Módulos são os blocos básicos de implementação dentro do SystemC. Os módulos permitem que sistemas complexos sejam divididos em pequenos blocos mais simples e os detalhes internos de implementação e representação de dados de cada bloco sejam escondidos de outros módulos.

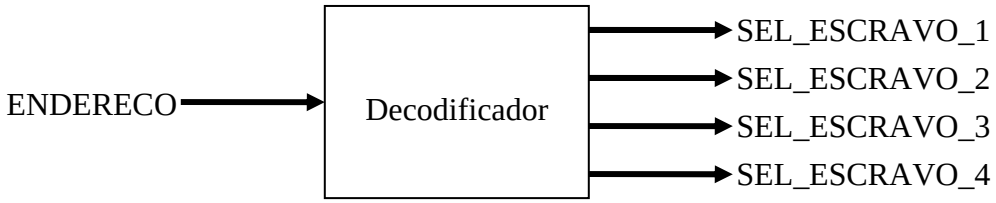

Figura 1 – Diagrama de um módulo do SystemC

Um módulo pode conter elementos tais como portas, sinais internos, dados internos, outros módulos, processos e construtores. A representação em SystemC do módulo visto na Figura 1 é como mostrado no Algoritmo 1 onde cada porta do diagrama tem um elemento que a descreve em SystemC dentro da declaração do módulo. No caso de *ENDERECO* por exemplo, existe a declaração *sc_in<int> ENDERECO* indicando que *ENDERECO* é uma porta de entrada do tipo inteiro no módulo *Decodificador*.

```
SC_MODULE(Decodificador)
{
        sc_in<int> ENDERECO;
        sc_out<bool> SEL_ESCRAVO_1;
        sc_out<bool> SEL_ESCRAVO_2;
        sc_out<bool> SEL_ESCRAVO_3;
        sc_out<bool> SEL_ESCRAVO_4;
        ...
}
```

Algoritmo 1 – Declaração de um módulo em SystemC

2.1.2 Processos

Processos podem ser definidos como as unidades básicas de execução dentro do SystemC e são chamados para simular o comportamento de um dispositivo ou sistema. Existem três tipos de processos no SystemC, métodos, *threads* e *clocked threads.* Cada um dos tipos tem um comportamento e uma aplicação diferente dentro de um sistema.

O processo do tipo *método* comporta-se simplesmente como uma função que é chamada pelo núcleo do simulador quando um sinal ao qual o método é sensível é modificado. O método executa e deve retornar o controle de execução ao núcleo do simulador. Seguindo com o exemplo do decodificador de endereços da Figura 1 que pode ter um método sensível a *ENDERECO* para fazer a decodificação e ajustar os valores das portas de saída do módulo *Decodificador* como mostrado no Algoritmo 2.

```
SC_MODULE(Decodificador)
{
        sc_in<int> ENDERECO;
        sc_out<bool> SEL_ESCRAVO_1;
        sc_out<bool> SEL_ESCRAVO_2;
        sc_out<bool> SEL_ESCRAVO_3;
        sc_out<bool> SEL_ESCRAVO_4;

        void Decodifica() // Método
        {
                // Ajusta os valores de SEL_ESCRAVO_x dependendo do valor
                // de ENDERECO
                SEL_ESCRAVO_1 = (ENDERECO >= 0x0000 && ENDERECO < 0x1000);
                SEL_ESCRAVO_2 = (ENDERECO >= 0x1000 && ENDERECO < 0x2000);
                SEL_ESCRAVO_3 = (ENDERECO >= 0x2000 && ENDERECO < 0x3000);
                SEL_ESCRAVO_4 = (ENDERECO >= 0x3000 && ENDERECO < 0x4000);
        }

        SC_CTOR(Decodificador)
        {
                SC_METHOD(Decodifica); // Declaração do método.
                sensitive(ENDERECO); // Indicação de que é sensível a ENDERECO.
        }
}
```

Algoritmo 2 – Método do SystemC

Quando o valor de *ENDERECO* é alterado, o SystemC automaticamente invoca o método Decodificador::Decodifica() que deve executar e retornar imediatamente.

Os processos *thread* são executados dentro de um *loop* infinito e podem ser suspensos e reativados. Podem conter chamadas *wait()* que suspendem a execução até que um evento ao qual

o processo é sensível ocorra. O processo do tipo *thread* é o processo mais genérico e pode ser usado para modelar quase tudo.

Clocked thread é um tipo especial do processo *thread* sendo somente sensível à variação do sinal de *clock*. *Clocked threads* podem ser usadas para criar máquinas de estados onde cada estado é formado por um grupo de instruções separadas por chamadas wait() e, a cada ciclo de clock, a máquina de estado muda para o próximo estado. O Algoritmo 3 mostra um exemplo de utilização de clocked thread.

```
SC_MODULE(Barramento)
{
        sc_in_clock<int> clock;

        ...

        void ExecutaTransferencias() // Clocked Thread
        {
                while (true) // Loop infinito.
                {
                        wait(); // espera a variação de clock.

                        // Executa uma transferência
                        Arbitro.Arbitrate();
                        endereço = Mestre.Endereço();
                        Escravo = Decodificador.Decodifica(endereço);
                        dado = Mestre.Dado();
                        Escravo.Escreve(dado);
                }
        }

        ...

        SC_CTOR(Barramento)
        {
                // Declaração do método Clocked Thread sensível ao clock.
                SC_CTHREAD(ExecutaTransferencias, clock.pos());
        }
}
```

Algoritmo 3 – Processo do tipo *Clocked Thread* do SystemC

No pseudocódigo do Algoritmo 3 o módulo *Barramento* possui um processo *Clocked Thread* que executa uma transferência do barramento a cada ciclo de *clock*.

2.1.3 Portas e Sinais

Portas de um módulo são as interfaces externas por onde trafegam informações de e para os módulos. Os sinais representam as conexões entre as portas dos módulos permitindo que os módulos se comuniquem entre si.

As portas podem ser de três tipos distintos dependendo do sentido que a informação trafega. Portas de entrada *sc_in* transferem dados externos para dentro do módulo. As portas de saída *sc_out* transferem dados do módulo para outros módulos através dos sinais. As portas de entrada e saída *sc_inout* são bidirecionais transferindo dados de dentro para fora e de fora para dentro do módulo.

2.2 TLM

A era do SoC criou a necessidade do desenvolvimento de novas metodologias de *design* para lidar com o crescente aumento de complexidade e com necessidade do desenvolvimento cada vez mais rápido de novos sistemas. TLM (*Transaction Level Modeling*) é uma metodologia que foi inserida para fazer frente aos desafios da era do *system-on-chip*.

Na modelagem no nível de transação toda a complexidade e detalhes dos *designs* com precisão de pino são substituídos por definições de interfaces e chamadas de funções. TLM descreve um sistema como uma série de transações de escrita e leitura, simplificando os esforços de modelagem e aumentando as velocidades de simulação. Detalhes desnecessários de comunicação e processamento são deixados de lado na fase de validação em nível sistêmico e esses detalhes, depois do sistema validado, podem ser trabalhados.

Os modelos de simulação de barramento foram desenvolvidos usando-se modelagem no nível de transação como pode ser visto na seção 4 (*Implementação dos Modelos*).

2.3 ArchC

ArchC [2] é uma linguagem de descrição de arquitetura baseada em SystemC. Com essa linguagem é possível descrever a arquitetura de um processador e hierarquia de memórias seguindo a sintaxe similar a do SystemC. O principal objetivo do ArchC é prover informações suficientes para se verificar e validar uma nova arquitetura gerando automaticamente ferramentas de software tais como montadores, simuladores e interfaces de verificação.

A descrição de uma arquitetura em ArchC é dividida em duas partes: *Instruction Set Architecture* (AC_ISA) e *Architecture Resources* (AC_ARCH). Para o AC_ISA, o usuário provê

informações sobre as instruções do processador tais como formato, tamanho e nome combinados com as informações necessárias para se decodificar e o comportamento de cada uma delas. O AC_ARCH contem informações sobre dispositivos de armazenamento, estruturas de *pipeline*, etc. Baseado nas informações acima, o ArchC gera um simulador comportamental descrito em SystemC para a referida arquitetura.

ArchC 2.0 especifica uma interface TLM chamada *ac_inout_if* para comunicação dos modelos de processadores com elementos externos. Essa interface basicamente define métodos *read* para leitura de dados externos e métodos *write* para envio de dados para elementos externos ao processador.

É através da interface *ac_inout_if* que um modelo de processador do ArchC conecta-se aos modelos de barramentos desenvolvidos. Para compatibilizar a conexão desses elementos é necessário a criação de módulos adaptadores que de um lado implementam a interface *ac_inout_if* e, do outro lado, implementam os requisitos necessários de um mestre de barramento. Os módulos adaptadores para os barramentos AHB e Avalon são descritos em seções posteriores desse documento.

Esse capítulo descreveu conceitos básicos de modelagem que serão usados nas descrições dos barramentos desse trabalho. O próximo capítulo lista os barramentos estudados e suas características.

3 Barramentos

Para o desenvolvimento dos modelos de software para simulação de barramentos foram estudados diferentes barramentos. Os barramentos estudados, AMBA, Avalon, Coreconnect e Wishbone, estão descritos nas próximas seções.

3.1 AMBA

As especificações AMBA (*Advanced Microcontroller Bus Architecture*) definem padrões de barramento para comunicação de alto desempenho. Estas especificações descrevem quatro tipos distintos de barramentos: *Advanced eXtensible Interface* (AXI) [13], *Advance High-Performance Bus* (AHB), *Advanced System Bus* (ASB) [12] e *Advanced Peripheral Bus* (APB) [14].

A especificação AXI, que define um protocolo baseado em *bursts*, é apropriada para interconexões de altíssima velocidade e desempenho.

AHB é utilizada para comunicação de alto desempenho com operação em altas freqüências de *clock*. Ideal para atuar como um barramento *backbone*. AHB pode ser conectado eficientemente a processadores, memórias e interfaces de memórias externas.

ASB é utilizada para conexão de módulos de alto desempenho e é uma alternativa ao AHB quando os requisitos de maior desempenho do AHB não são necessários. ASB também se conecta a processadores, memórias e interfaces de memórias externas.

A especificação APB é utilizada para conexão de periféricos de baixa capacidade. APB é otimizado para baixo consumo de energia e tem interface de baixa complexidade.

Um sistema AMBA, conforme visto na Figura 2, tipicamente é formado por um barramento *backbone* (AHB ou ASB) conectado à memória externa, CPU e dispositivos DMA. Este barramento provê alta capacidade de taxa de transferência de dados entre os elementos que estão envolvidos na maioria das transferências de dados no sistema. Ao barramento APB de baixa capacidade de transferência estão conectados os demais periféricos pertencentes ao sistema. A interligação entre o *backbone* e o barramento APB é feito através de uma ponte APB. Este dispositivo ponte, do ponto de vista do *backbone*, é apenas um dispositivo escravo.

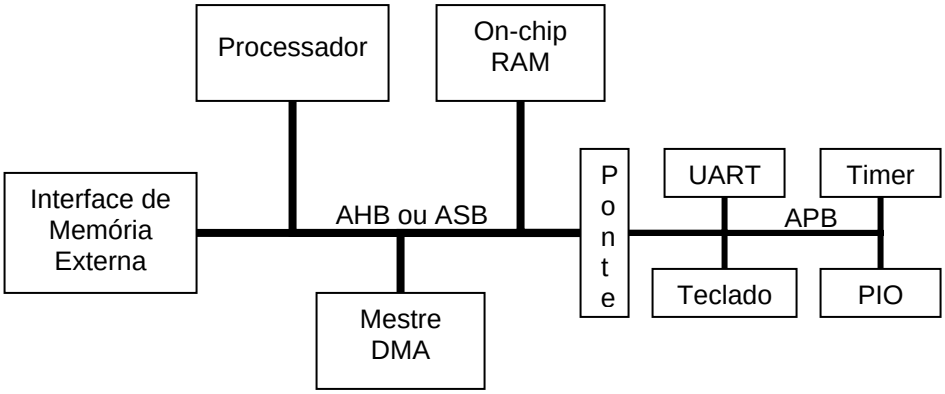

Figura 2 – Sistema AMBA típico

As seções a seguir descrevem, em detalhes, três tipos de barramentos do padrão AMBA, AHB, ASB e APB. O padrão AXI, lançado recentemente, não faz parte do escopo desse trabalho.

3.1.1 AMBA AHB

AMBA AHB foi desenvolvido para atender requisitos de alto desempenho suportando múltiplos mestres e altas taxas de transferência de dados. Um sistema AHB típico contem os seguintes componentes: Um ou mais dispositivos mestres, pelo menos um dispositivo escravo, um árbitro e um decodificador de endereços.

Um mestre do barramento é capaz de iniciar operações de escrita e leitura fornecendo o endereço do dispositivo escravo e os sinais de controle necessários. Somente um mestre pode estar ativo no barramento em um determinado tempo.

Um escravo é um elemento reativo, que apenas responde a operações de leitura e escrita dentro de sua faixa de endereço. O escravo também sinaliza para o mestre o resultado de uma operação, que pode ser sucesso, falha ou indicação de que o mestre precisa esperar para completar a operação desejada.

O árbitro é quem garante que apenas um mestre está ativo no barramento. O árbitro do AHB pode ser implementado usando-se qualquer algoritmo, tais como *highest priority* ou *fair access*, dependendo dos requisitos desejados.

O decodificador de endereços é utilizado pelo barramento em todas as transferências para selecionar o escravo relativo àquele endereço.

O protocolo do barramento AHB é feito para ser usado com um multiplexador central de interconexões. Nesse esquema todos os mestres do barramento, independentemente se são ou não os donos do barramento, geram seus sinais de saída tais como endereço e sinais de controle. O árbitro determina qual mestre é o dono do barramento e propaga os sinais desse mestre e somente desse mestre para todos os escravos. O decodificador de endereços decodifica o endereço fornecido pelo mestre ativo e seleciona os sinais do escravo envolvido na transferência.

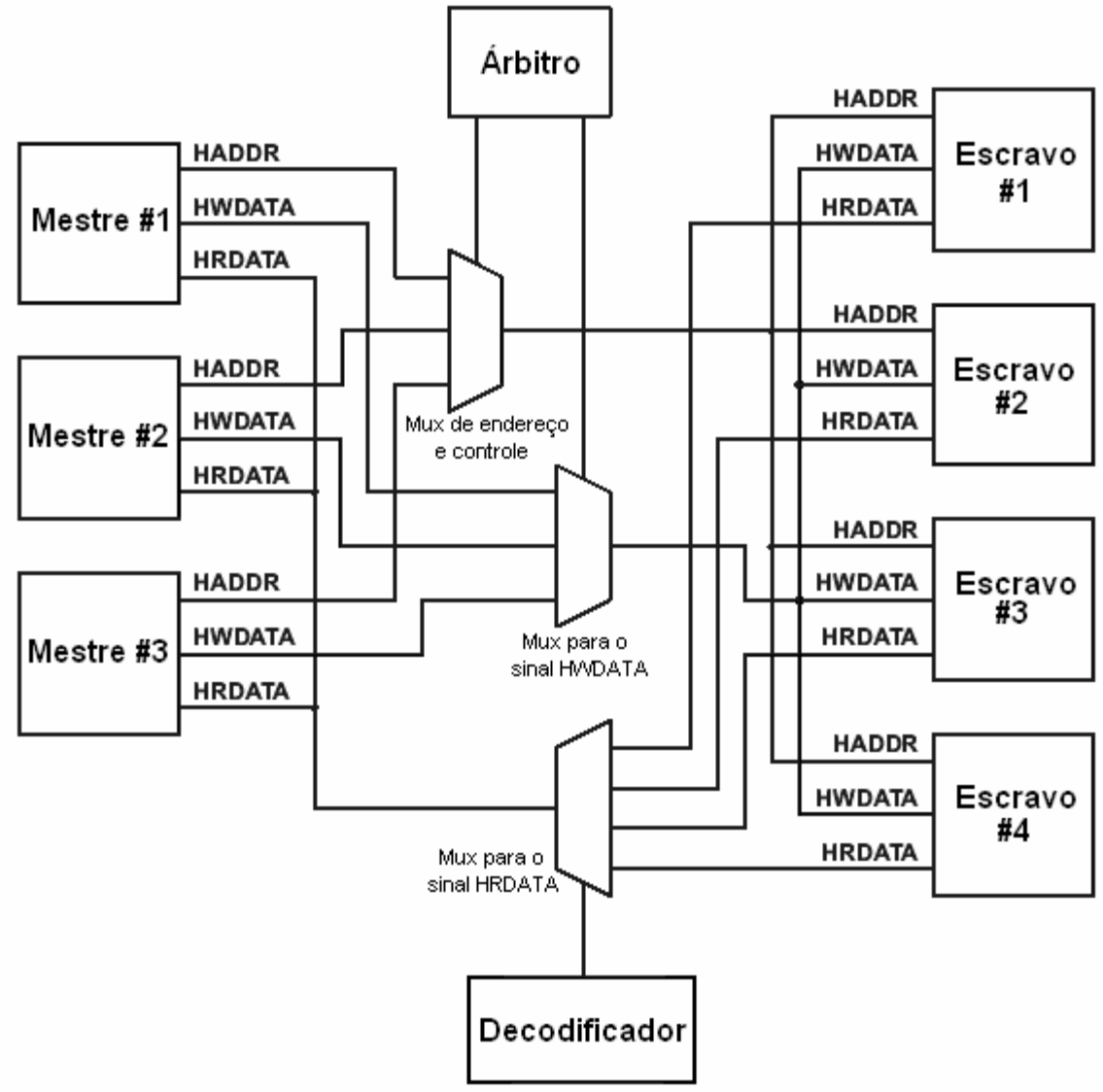

Figura 3 – Multiplexador de interconexões do AHB

Antes de iniciar uma transferência de dados, um mestre precisa requisitar o barramento e esperar a indicação de que ele é o dono do barramento para poder iniciar a transferência. A

transferência começa com o mestre provendo o endereço do escravo e os sinais de controle que indicam se é uma operação de leitura ou escrita, o número de bytes e se é uma transferência simples ou uma operação em *burst*.

Cada transferência é formada por um ciclo onde são fornecidos o endereço e os sinais de controle e por um ou mais ciclos de dados. O endereço não pode ser estendido e os escravos devem armazenar esse valor durante o primeiro ciclo da transferência. Os dados podem ser estendidos pelo escravo que, se necessário, pode fazer o sinal HREADY igual à zero para forçar a inserção de ciclos de espera enquanto não tiver dados a fornecer ou não pode fazer a leitura dos dados disponíveis.

Durante uma transferência, o escravo também indica o estado da operação através do sinal HRESP que pode ser OKAY, ERROR, RETRY ou SPLIT. OKAY indica que o progresso da transferência está normal e pode ser completada com sucesso. ERROR indica que algum erro aconteceu e a transferência não foi e não pode ser efetuada. RETRY e SPLIT indicam que a transferência não pode ser efetuada imediatamente, mas o mestre deve continuar tentando completar a operação nos próximos ciclos.

3.1.1.1 Transferência simples

Uma transferência de dados no AHB sempre é formada por duas fases distintas. A fase de endereço que é composta por um único ciclo de *clock* e a fase de dados que pode se prolongar por vários ciclos dependendo do sinal HREADY que é controlado pelo escravo envolvido na transferência.

Na primeira fase da transferência, o mestre fornece o endereço e os sinais de controle. No próximo ciclo de *clock*, onde começa a fase de dados, o escravo captura o endereço e os sinais de controle e logo em seguida sinaliza com a resposta apropriada. A resposta do escravo será capturada pelo mestre no início do próximo ciclo.

A fase de endereço de qualquer transferência ocorre durante a fase de dados da transferência anterior. Essa sobreposição das fases de endereço e dados de transferências distintas é fundamental para se garantir operação em *pipeline* e se conseguir alto desempenho com altas taxas de transferência ao mesmo tempo que reserva tempo adequado para o escravo durante a operação.

A Figura 4 mostra uma transferência simples na qual o escravo está pronto para capturar ou fornecer dados e não insere nenhum ciclo de espera, fazendo com que a fase de dados também seja composta por um único ciclo de *clock*.

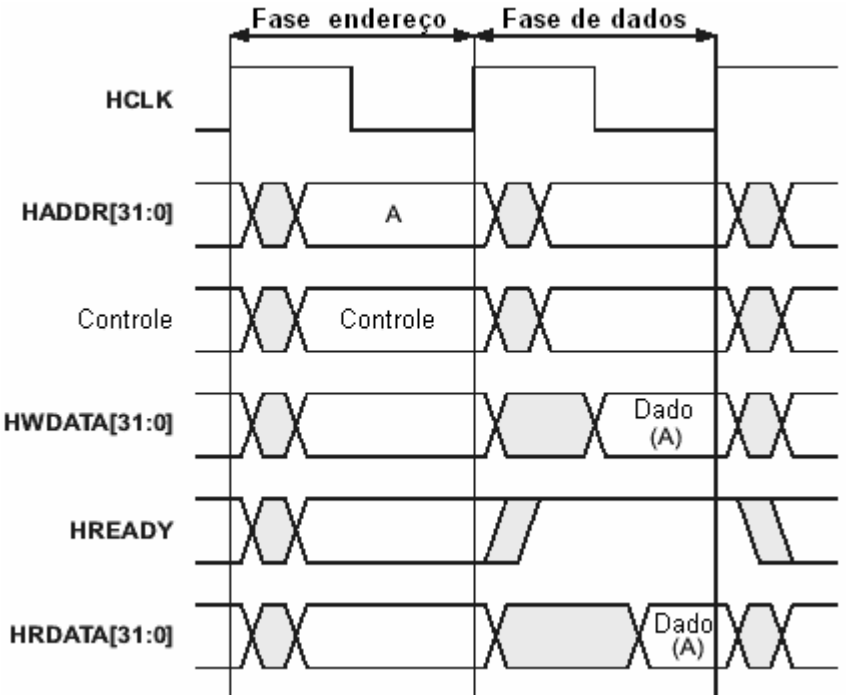

Figura 4 – Uma transferência simples no AHB sem ciclos de espera

3.1.1.2 Transferência em *burst*

AHB suporta variados tipos de transferências em *burst*, incluindo *bursts* incrementais e *wrapping bursts.*

Bursts incrementais acessam posições seqüenciais e o endereço de cada transferência dentro do mesmo *burst* é apenas um incremento do endereço da transferência anterior.

Em uma transferência *wrapping burst*, se o endereço inicial da transferência não estiver alinhado com o número total de bytes, o endereço será cortado quando o limite de alinhamento for atingido.

3.1.1.3 Sinais de controle

Além do tipo de transferência e dados sobre operação em *burst*, cada transferência tem outros sinais de controle que fornecem informações adicionais sobre a operação em execução. Esses sinais de controle devem seguir a mesma temporização que o endereço mas devem permanecer constantes durante toda a transferência.

O sinal HWRITE indica a direção da transferência. Quando igual à zero, indica uma operação de leitura em que o escravo deve fornecer dados relativos ao endereço indicado pelo mestre. E quando igual a um, indica uma operação de escrita em que o escravo deve capturar os dados fornecidos pelo mestre.

O sinal de controle HSIZE indica o tamanho da transferência, como mostrado na Tabela 1.

HSIZE	Número de bits
0.0.0	8 bits
0.0.1	16 bits
0.1.0	32 bits
0.1.1	64 bits
1.0.0	128 bits
1.0.1	256 bits
1.1.0	512 bits
1.1.1	1024 bits

Tabela 1 – Valores possíveis para HSIZE

O sinal de proteção HPROT transporta informações adicionais sobre acesso ao barramento e deve ser usado por módulos que desejam implementar algum nível de proteção. Esse sinal indica se a transferência é para leitura de instruções de um programa em execução ou acesso a dados juntamente com a indicação de modo, que pode ser privilegiado ou modo usuário.

3.1.1.4 Decodificador de endereços

O decodificador de endereços no AMBA é usado para fazer a decodificação de endereços de forma centralizada, que promove a portabilidade de periféricos fazendo-os independentes do mapa de memória do sistema.

O decodificador de endereços é responsável pela geração do sinal HSELx, um para cara escravo do barramento, definindo qual o escravo selecionado para a transferência. Um escravo somente deve amostrar o endereço e os sinais de controle quando HSELx e HREADY estiverem ambos ativos.

O espaço de endereçamento mínino que pode ser alocado para um escravo é 1KB e os mestres são proibidos de fazer transferências incrementais maiores que o limite de 1 KB, isso assegura que nunca uma transferência em *burst* vai passar pelos limites de endereços dos escravos.

No caso em que o mapa de memória não está completamente preenchido, um escravo adicional deve ser implementado para fornecer a resposta no caso de acesso a uma área de memória não utilizada. Tipicamente, esse escravo adicional é implementado como parte do decodificador de endereços.

A Figura 5 mostra o diagrama de interface do decodificador AHB.

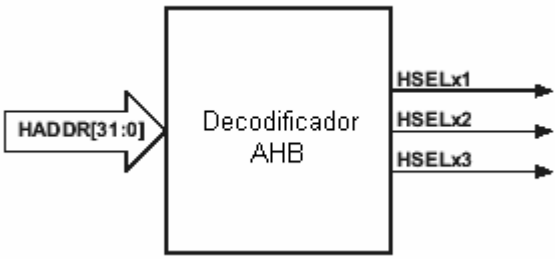

Figura 5 – Diagrama da interface do decodificador de endereços do AHB

3.1.1.5 Árbitro AHB

A função do árbitro no AMBA é controlar qual mestre terá acesso ao barramento. Um mestre, quando deseja fazer uma transferência, ativa o sinal HBUSREQ e o árbitro, baseado em

um algoritmo de priorização, define qual o mestre será o dono do barramento naquele ciclo. O mestre em questão terá o seu sinal de controle HGRANT ativado.

Os detalhes de funcionamento do algoritmo de priorização do árbitro não são especificados pelo AMBA, sendo aceitável que se use quaisquer sinais, do AMBA ou não, para o funcionamento do árbitro, dependendo das necessidades da aplicação.

A Figura 6 mostra o diagrama de interface do árbitro AHB.

Figura 6 – Diagrama da interface do árbitro AHB

O sinal HBUSREQx é usado por um mestre para requisitar acesso ao barramento. Cada mestre no barramento tem seu sinal HBUSREQx, sendo possível até dezesseis mestres.

O sinal HLOCKx deve ser acionado ao mesmo tempo que HBUSREQx, é usado para indicar que o mestre deseja executar transferências indivisíveis e o árbitro não deve dar acesso ao barramento a outro mestre depois que as transferências começarem.

HGRANTx é gerado pelo árbitro e indica qual o mestre é dono do barramento num determinado ciclo. O árbitro também indica qual o mestre dono do barramento gerando o sinal HMASTER[3:0] que contem o número do mestre em questão.

O árbitro pode indicar que a transferência atual é uma transferência indivisível acionando o sinal HMASTLOCK.

HSPLIT[15:0] é usado por escravos com capacidade de executar transações fracionadas e indica qual mestre pode completar uma operação iniciada e interrompida anteriormente. Essa informação é necessária para o árbitro dar acesso ao barramento para o mestre correto que tem uma operação pendente e que pode ser concluída.

3.1.1.6 Mestre AHB

O mestre do barramento AHB possui a mais complexa interface do sistema. Tipicamente, novos projetos devem tentar reusar implementações já testadas para se evitar problemas com os diversos detalhes de implementação da interface do mestre AHB. A Figura 7 mostra o diagrama de interface do mestre AHB.

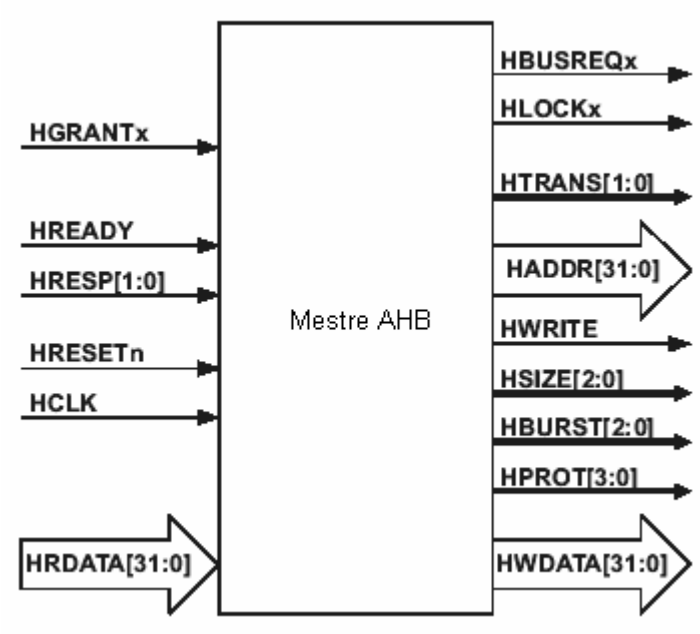

Figura 7 – Diagrama da interface do mestre AHB

3.1.1.7 Escravo AHB

Um escravo no AHB é passivo e apenas responde às transferências iniciadas pelos mestres no sistema. O escravo usa o sinal HSELx originado pelo decodificador de endereços para determinar quando deve responder a uma transferência através dos sinais HREADY e HRESP. A Figura 8 mostra o diagrama de interface de um escravo AHB.

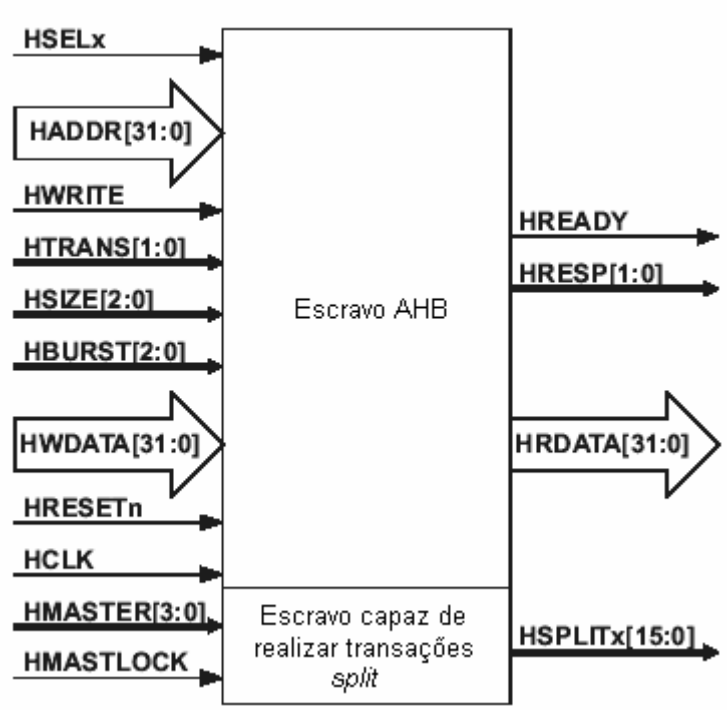

Figura 8 – Diagrama da interface do escravo AHB

Um escravo no AHB pode ser capaz ou não de executar transações *split*. Transações *split* são transações que aumentam a utilização do barramento porque separam as fases de endereço e dados de uma transferência.

Quando uma transferência é iniciada, o escravo pode decidir se será ou não uma transação *split* dependendo do tempo necessário para preparar os dados da resposta. Se a preparação da resposta para a transação levar vários ciclos, o escravo responde imediatamente com o sinal HRESP igual à SPLIT. Mais tarde quando o processamento dos dados estiver pronto, o escravo

sinaliza para o árbitro através do sinal HSPLITx que o mestre que iniciou a transação *split* pode receber o barramento para finalizar a transferência iniciada anteriormente.

3.1.1.8 Sinais do barramento AHB

A Tabela 2 mostra todos os sinais do barramento AHB com as respectivas descrições.

Sinal	Fonte do sinal	Descrição
HCLK	*Clock*	Sinal de clock do sistema usado para temporização.
HRESETn	Sinal global de *reset*	Sinal global de *reset* do sistema.
HADDR	Mestre	Barramento de endereços de 32 *bits*.
HTRANS	Mestre	Indica o tipo da transferência que está sendo executada. Pode ser *NONSEQUENTIAL*, *SEQUENTIAL*, *IDLE* ou *BUSY*.
HWRITE	Mestre	Quando ativo indica uma operação de escrita. Quando desativo indica uma operação de leitura.
HSIZE	Mestre	Indica o tamanho da transferência. Tipicamente pode ser 8-*bits*, 16-*bits* ou 32-*bits*, mas o protocolo permite transferências de até 1024 *bits*.
HBURST	Mestre	Indica se a transferência faz parte de um *burst*.
HPROT	Mestre	Sinal de indicação de proteção. É usado por módulos que usam níveis de proteção nas transferências.
HWDATA	Mestre	É usado para transferir dados do mestre para o escravo em operações de escrita.
HSELx	Decodificador de endereços	Sinal de seleção de escravo.
HRDATA	Escravo	É usado para transferir dados do escravo para o mestre em operações de leitura.
HREADY	Escravo	Quando ativado indica que a transferência foi finalizada no barramento. É usado pelos escravos para extender uma transferência.
HRESP	Escravo	É o sinal que provê informações adicionais sobre o resultado da transferência. Pode ser *OKAY*, *ERROR*, *RETRY* ou *SPLIT*.
HBUSREQx	Mestre	Sinal usado pelos mestres para requisitar o barramento.
HLOCKx	Mestre	Usado pelos mestres para indicar uma transferência indivisível.
HGRANTx	Árbitro	Sinal gerado pelo árbitro indicando que o mestre x é o dono do barramento e pode iniciar uma transferência.

HMASTER	Árbitro	Indica qual o mestre que está usando o barramento. Auxilia os escravos nas transferências *split*.
HMASTLOCK	Árbitro	Indica que o mestre dono do barramento está executando uma seqüência indivisível de transferências.
HSPLITx	Escravo	É usado pelos escravos para indicar ao árbitro que um mestre pode reiniciar uma transferência interrompida por uma transação *split*.

Tabela 2 – Sinais do barramento AMBA AHB

3.1.2 AMBA ASB

ASB (*Advanced System Bus*) é a primeira geração de barramentos AMBA. ASB está um nível acima do barramento APB e especifica um protocolo de alto desempenho que inclui transferências em *bursts*, transferências em *pipeline* e suporte a múltiplos mestres de barramento.

A seqüência básica de operação do barramento ASB é a seguinte: Primeiro o árbitro determina qual é o mestre que será o dono do barramento. Depois disso o mestre dono do barramento inicia a transferência. Durante a transferência, o decodificador de endereços usa os *bits* de mais alta ordem do endereço para determinar o escravo. O escravo fornece a resposta do estado da transferência e a transação é finalizada.

Os elementos típicos de um barramento ASB são os mestres, os escravos, o decodificador de endereços e o árbitro. Esses elementos são descritos, juntamente com os tipos de transferência e os sinais do barramento ASB, nas seções a seguir.

3.1.2.1 Tipos de transferência

Existem três tipos de transferência especificadas pelo protocolo do barramento ASB: transferência seqüencial, não seqüencial e transferência somente de endereço. O tipo de transferência é indicado pelo sinal BTRAN.

A transferência não seqüencial é usada para se fazer uma transferência simples ou para indicar a primeira transferência de um *burst*.

Uma transferência seqüencial ocorre quando o endereço atual da transferência está relacionado ao endereço da transferência anterior. Se uma transferência seqüencial vem logo depois de uma não seqüencial ou depois de outra seqüencial, o endereço atual pode ser calculado usando-se o tamanho e o endereço da transferência anterior.

Transferência somente de endereço indica que nenhum dado será requisitado e nenhum escravo estará envolvido na transação. Esse tipo de transferência pode ser usado de três maneiras: para o mestre dono do barramento inserir ciclos *idle* na transação, para um mestre verificar um endereço que será usado na próxima transferência e para inserir um ciclo de intervalo entre a troca de mestres (quando o árbitro passa o controle do barramento para outro mestre).

3.1.2.2 Decodificador de endereços

No barramento ASB, a decodificação de endereços é feita por um decodificador central que facilita a portabilidade dos periféricos fazendo-os independentes do mapa de memória do sistema e simplifica o projeto dos escravos do barramento. O diagrama de interface do decodificador de endereços ASB pode ser visto na Figura 9.

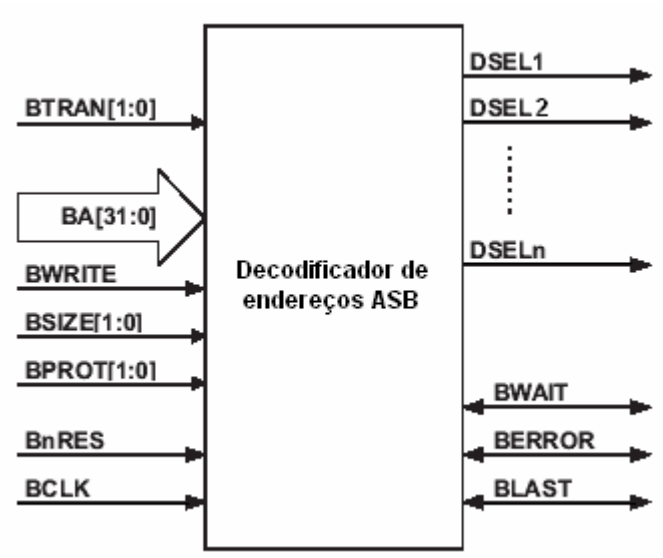

Figura 9 – Diagrama de interface do decodificador de endereços ASB

O decodificador de endereços do barramento ASB gera um sinal de seleção *DSELx* para cada escravo do barramento.

O decodificador também é usado para outras funções no barramento. Atua como uma unidade de proteção gerando uma resposta de erro para um mestre que esteja tentando fazer uma transferência para um endereço inválido ou protegido. Também é responsável por fornecer a resposta de uma transferência somente de endereço onde nenhum escravo está envolvido.

3.1.2.3 Árbitro ASB

A função do árbitro no ASP é determinar qual mestre terá acesso ao barramento. Cada mestre tem dois sinais, AREQx e AGNTx, que estão conectados com o árbitro e, em cada ciclo, o árbitro usa esses sinais e um algoritmo de priorização para decidir a qual dos mestres o barramento será disponibilizado. A Figura 10 mostra o diagrama de interface do árbitro ASB.

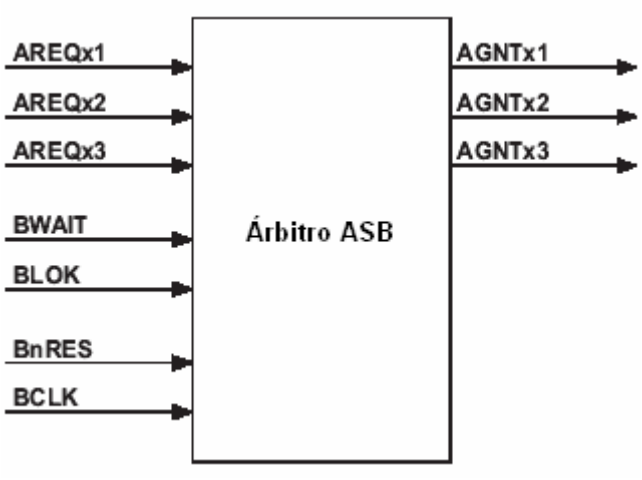

Figura 10 – Diagrama de interface do árbitro ASB

Os detalhes do algoritmo de priorização não são especificados pelo AMBA e podem ser definidos dependendo dos requisitos de cada aplicação.

3.1.2.4 Escravo ASB

Um escravo no barramento ASB é um elemento passivo e apenas responde a transações iniciadas pelos mestres do barramento. O escravo usa o sinal de seleção DSELx gerado pelo decodificador de endereços para determinar quando deve ou não responder a uma transferência. Todos os outros sinais usados pelo escravo são gerados pelo mestre envolvido na transação. A Figura 11 mostra o diagrama de interface do escravo ASB.

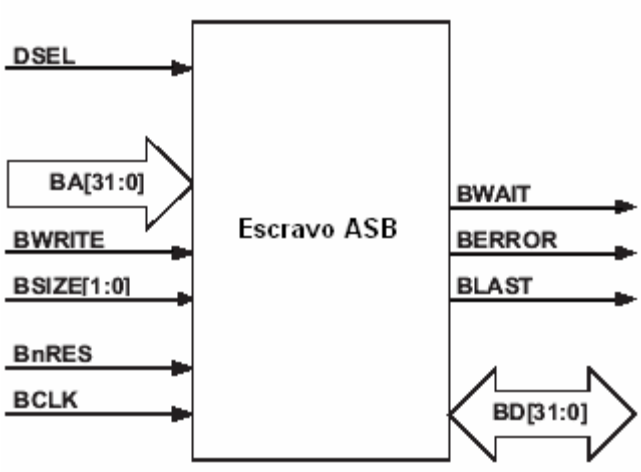

Figura 11 – Diagrama de interface do escravo ASB

3.1.2.5 Mestre ASB

O mestre do barramento ASB tem a interface mais complexa dentro do sistema. Tipicamente se usam projetos prontos de mestres para evitar a complexidade de implementação da interface do mestre. O digrama de interface do mestre do barramento ASB é mostrado na Figura 12.

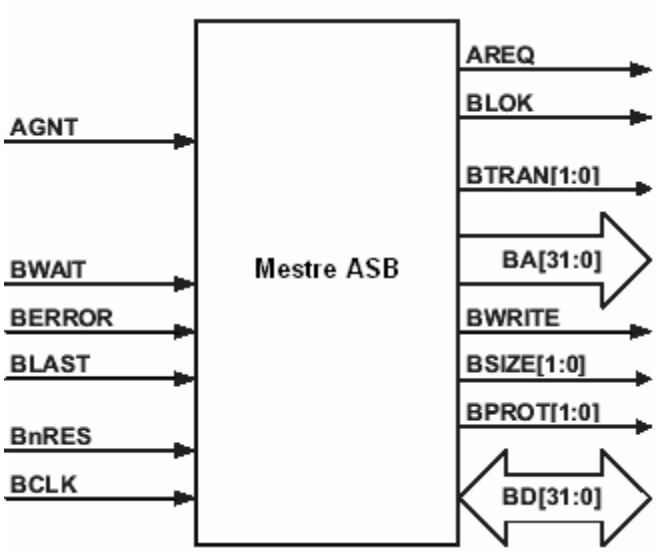

Figura 12 – Diagrama de interface do mestre ASB

Antes de iniciar uma transação, o mestre deve ativar o sinal AREQx e ficar monitorando o sinal AGNTx. Quando o sinal AGNTx está ativo, indica que o mestre é o dono do barramento e pode executar a transferência. Durante a transação o mestre aciona os sinais de saída da sua interface para controlar a transferência e monitora os sinais gerados pelo escravo, BWAIT, BERROR e BLAST, para verificar o progresso.

3.1.2.6 Sinais do barramento ASB

A Tabela 3 mostra todos os sinais do barramento ASB com as respectivas descrições.

Sinal	Descrição
AGNTx	Sinal que vai do árbitro para o mestre x indicando que ele vai ser o dono do barramento quando BWAIT for desativado.
AREQx	Sinal do mestre x para o árbitro requisitando o barramento.
BA	Barramento de endereços do sistema. É controlado pelo mestre dono do barramento.
BCLK	Sinal de *clock* do sistema.
DB	Barramento de dados bidirecional. É controlado pelo mestre durante

	operações de escrita e pelo escravo em operações de leitura.
BERROR	Sinal usado pelo escravo envolvido em uma transferência para indicar erro.
BLAST	Sinal usado pelo escravo envolvido em uma transação para indicar que essa transferência deve ser a última da seqüência de *burst*.
BLOK	Esse sinal indica que a transferência corrente e a próxima fazem parte de uma transação indivisível e que o mestre não deve perder o controle do barramento.
BnRES	Sinal global de *reset* do sistema.
BPROT	Sinal de indicação de proteção. É usado por módulos que usam níveis de proteção nas transferências.
BSIZE	Indica o tamanho da transferência. Pode ser 8-*bits*, 16-*bits* ou 32-*bits.*
BTRAN	Indica o tipo da próxima transação, pode ser *ADDRESS-ONLY*, *NONSEQUENTIAL* ou *SEQUENTIAL.* É controlado pelo mestre dono do barramento.
BWAIT	Esse sinal é controlado pelo escravo envolvido na transferência. Indica que a transferência não pode ser completada nesse ciclo.
BWRITE	Indica o tipo de transferência. Quando ativado indica uma operação de escrita e quando desativado indica uma operação de leitura.
DSELx	Sinal gerado pelo decodificador de endereços para selecionar o escravo correto em uma transação.

Tabela 3 – Sinais do barramento AMBA ASB

3.1.3 AMBA APB

APB (*Advanced Peripheral Bus*) faz parte da família de protocolos AMBA. A especificação APB provê uma interface de baixo custo que é otimizada para o mínimo consumo de energia com interface de baixa complexidade. APB conecta-se a quaisquer periféricos de

baixa taxa de transferência que não requerem a alto desempenho fornecida por um barramento com transferências em *pipeline*.

Todos os sinais do barramento APB somente são amostrados na borda de subida do *clock* e cada transferência demora pelo menos dois ciclos de *clock*.

O barramento APB pode conectar-se aos barramentos de alto desempenho, AXI, AHB e ASB, através de pontes AXI-APB, AHB-APB e ASB-APB respectivamente.

3.1.3.1 Transferência simples

A Figura 13 mostra o diagrama de uma transferência simples no barramento APB.

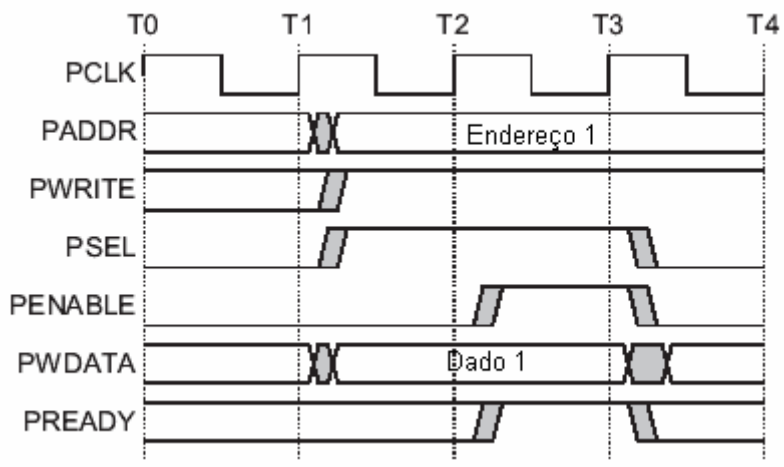

Figura 13 – Transferência simples no barramento APB

A transferência inicia com o endereço (PADDR), com o dado (PWDATA), com o sinal de indicação de escrita (PWRITE ativo) e com o sinal de seleção de escravo (PSELx) sendo ativados depois da borda se subida do *clock*. Esse primeiro ciclo de clock (T1) é chamado de fase de *setup*. No próximo ciclo (T2) o sinal PENABLE é ativado indicando que a fase de acesso está iniciando. Nessa fase os outros sinais anteriormente ajustados devem permanecer constantes. O sinal PENABLE é desativado no final da transferência. O sinal PREADYx indica que o escravo capturou o dado no ciclo de acesso e a transferência pode ser finalizada.

3.1.3.2 Sinais do barramento APB

A Tabela 4 mostra todos os sinais do barramento APB com as respectivas descrições.

Sinal	Fonte do sinal	Descrição
PCLK	*Clock*	*Clock*. Todas as transferências são executadas na borda de subida do *clock*.
PRESETn	Externo	Sinal de *reset* geral do sistema.
PADDR	Ponte APB	Sinal de endereço. Pode ser de até 32 *bits*.
PSELx	Ponte APB	Existe um sinal PSELx para cada escravo no barramento APB. Esse sinal indica qual o escravo referente a uma transferência.
PENABLE	Ponte APB	Esse sinal indica o segundo e os subseqüentes ciclos de uma transferência.
PWRITE	Ponte APB	Indica a direção da transferência. Quando ativo é indicação de uma operação de escrita. Quando não ativo, indica uma operação de leitura.
PWDATA	Ponte APB	Dados de uma operação de escrita. Pode ser de até 32 *bits*.
PREADYx	Escravo APB	Os escravos usam esse sinal para extender uma transferência quando não estão prontos para continuar com a operação.
PRDATA	Escravo APB	Dados de uma operação de leitura que são fornecidos pelos escravos. Pode ser de até 32 *bits*.
PSLVERR	Escravo APB	Esse sinal é controlado pelos escravos e indica que ocorreu falha na transferência.

Tabela 4 – Sinais do barramento AMBA APB

3.2 AVALON

O barramento Avalon [15] é uma arquitetura simples, feito para conectar processadores e periféricos. A interface do Avalon especifica portas para conexão entre mestres e escravos juntamente com a temporização necessária para esses elementos se comunicarem.

Uma transferência de dados no Avalon pode ser constituída por um único *byte*, por meia palavra de 16 *bits* ou uma palavra de 32 *bits* e logo depois que a transferência é finalizada, no próximo ciclo de *clock*, o barramento já está disponível para executar a próxima transferência. O barramento Avalon também suporta periféricos com latência, transferências em *streaming* e

múltiplos mestres, sendo possível transferir várias unidades de dados em uma única transação no barramento.

Os mestres e os escravos no barramento Avalon interagem baseados numa técnica chamada de arbitragem pelo lado do escravo, isto é, o árbitro define qual mestre terá acesso a um escravo quando múltiplos mestres tentam acessar o mesmo escravo. Nessa técnica, os detalhes de arbitragem ficam escondidos dentro do barramento e cada mestre trabalha como se fosse o único dentro do barramento e múltiplos mestres podem executar suas transações simultaneamente desde que eles não tentem acessar o mesmo escravo no mesmo ciclo.

Avalon é uma arquitetura de barramento *on-chip* formado por lógica e recursos de roteamento dentro de um PLD. A interface de periféricos é síncrona ao *clock* do barramento portanto, esquemas assíncronos complexos de *handshaking* e *ack* não são necessários. No barramento são usados portas separadas e dedicadas para dados, endereço e sinais de controle, facilitando assim o *design* dos periféricos.

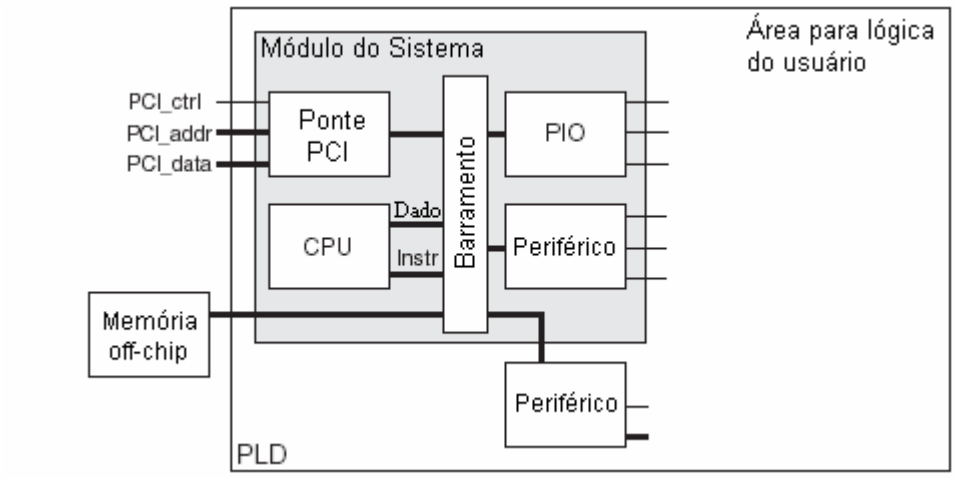

Figura 14 – Arquitetura do barramento Avalon

As seções seguintes descrevem os elementos da arquitetura do barramento Avalon.

3.2.1 Módulo do barramento

O módulo do barramento do Avalon mostrado na Figura 14 é o *backbone* do sistema. Ele é o caminho principal de comunicação entre os periféricos. O módulo do barramento Avalon é a

soma de todos os sinais de controle, do barramento de dados, do barramento de endereços e da lógica de arbitragem que juntos conectam os periféricos formando o sistema como um todo. O módulo do barramento Avalon implementa uma arquitetura de barramento configurável que é modificada para adaptar-se as necessidades de interconexão dos periféricos.

3.2.2 Mestre Avalon

Um mestre no Avalon é o periférico que pode iniciar transferências no barramento e contem pelo menos uma porta mestre que é conectada ao módulo do barramento Avalon.

Os sinais da interface do mestre Avalon são descritos na Tabela 5.

Sinal	Tamanho	Descrição
Clk	1	*Clock*. Todas as transferências são síncronas em relação ao *clock*.
Reset	1	Sinal de *reset* geral do sistema.
address	1 - 32	Linhas de endereço do barramento.
byteenable	0, 2, 4	Sinal de seleção dos bytes da palavra sendo transferida.
Read	1	Quando ativo, indica uma operação de leitura.
readdata	8, 16, 32	Linhas de dados do barramento usadas em operações de leitura.
Write	1	Quando ativo, indica uma operação de escrita.
writedata	8, 16, 32	Linhas de dados do barramento usadas em operações de escrita.
waitrequest	1	Quando ativo, força o mestre a esperar antes de completar a transferência.
Irq	1	Quanto ativo indica que um ou mais escravos fizeram pedido de interrupção.
irqnumber	6	Prioridade da interrupção do escravo que gerou o pedido de interrupção.
endofpackage	1	Sinal usado para transferências em *streaming*. Pode ser usado para indicar *final de pacote* para o mestre durante a transferência. A implementação do tratamento desse sinal é dependente dos periféricos.
readdatavalid	1	Sinal usado para transferências de leitura com latência. Indica que o dado presente em *readdata* é válido.
Flush	1	Sinal usado para transferências de leitura com latência. O mestre pode acionar esse sinal para

		interromper qualquer transferência latente que esteja pendente.

Tabela 5 – Sinais da interface do mestre no barramento Avalon

3.2.3 Escravo Avalon

Um escravo no barramento Avalon é um periférico que apenas responde a transferências feitas pelo barramento, não pode iniciar transferências. Um escravo do barramento contem pelo menos uma porta escravo que é conectada ao módulo do barramento Avalon.

Os sinais que chegam aos escravos sempre são o resultado de uma transferência iniciada por algum mestre, mas esse sinais não vêem diretamente do mestre. Os sinais que chegam aos escravos são o resultado da lógica de funcionamento e arbitragem do barramento.

Os sinais da interface do escravo Avalon são mostrados com as respectivas descrições na Tabela 6.

Sinal	Tamanho	Descrição
Clk	1	*Clock*. Todas as transferências são síncronas em relação ao *clock*.
Reset	1	Sinal de *reset* geral do sistema.
chipselect	1	Sinal de seleção de escravo. Um escravo deve ignorar todos os outros sinais enquanto chipselect estiver desativado.
address	1 - 32	Linhas de endereço do barramento.
begintransfer	1	Acionado durante o primeiro ciclo de cada nova transferência iniciada barramento
byteenable	0, 2, 4	Sinal de seleção dos bytes da palavra sendo transferida.
read	1	Quando ativo, indica uma operação de leitura.
readdata	1 – 32	Linhas de dados do barramento usadas em operações de leitura.
write	1	Quando ativo, indica uma operação de escrita.
writedata	1 – 32	Linhas de dados do barramento usadas em operações de escrita.
readdatavalid	1	Usada somente por escravos com latência variável. É ativado pelo escravo quando o valor de *readdata* é válido.
waitrequest	1	Usado pelo escravo para congelar uma transferência quando não estiver pronto para responder imediatamente.

readyfordata	1	Sinal usado para transferências em *streaming*. Indica que o escravo pode receber dados.
dataavailable	1	Sinal usado para transferências em *streaming*. Indica que o escravo tem dados disponíveis para leitura.
endofpackage	1	Sinal usado para transferências em *streaming*. Pode ser usado para indicar *final de pacote* para o mestre durante a transferência. A implementação do tratamento desse sinal é dependente dos periféricos.
irq	1	Sinal de requisição de interrupção. O escravo ativa *irq* quando desejar ser atendido por um mestre.
resetrequest	1	Sinal de *reset*. Pode ser usado pelo periférico para reiniciar todo o sistema.

Tabela 6 – Sinais da interface do escravo no barramento Avalon

3.2.4 Tipos de transferência

No barramento Avalon existem dois tipos de transferência, a elementar e a transferência *streaming*. Esses tipos de transferência vistas pelo lado do mestre e pelo lado do escravo são descritas nas seções seguintes.

3.2.4.1 Transferência elementar do ponto de vista do escravo

A transferência elementar é iniciada pelo barramento e é transferida uma unidade de informação, sem latência e em um único ciclo de *clock*. A Figura 15 mostra uma transferência de leitura elementar.

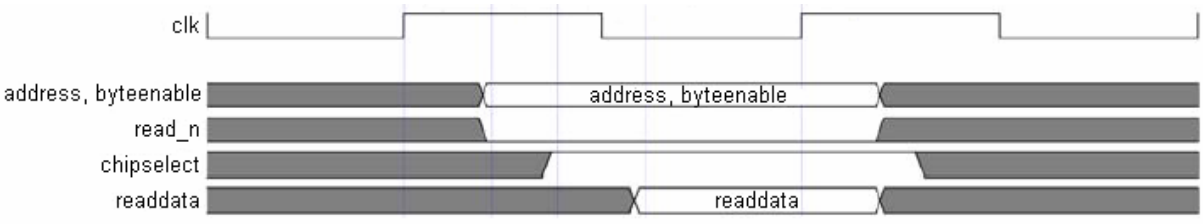

Figura 15 – Transferência elementar no barramento Avalon

Na primeira borda de subida do *clock*, o barramento fornece os sinais *address*, *byteenable* e *read_n* para o escravo. O barramento decodifica o endereço internamente e ativa o *chipselect* do escravo relativo ao endereço fornecido pelo mestre. Depois que *chipselect* é ativado, o escravo fornece o dado através do sinal *readdata* o mais rápido possível. Finalmente, o barramento captura o valor de *readdata* na próxima borda de subida do *clock*. Outra transferência pode ser iniciada nesse mesmo ciclo de *clock*.

3.2.4.2 Transferência elementar do ponto de vista do mestre

Existe uma regra fundamental para o mestre realizar transferências no barramento Avalon: O mestre deve ativar todos os sinais para iniciar a transferência e então, deve somente esperar até que o sinal *waitrequest* seja desativado.

A transferência elementar é iniciada pelo mestre do barramento e é transferida uma unidade de informação, sem latência e em um único ciclo de *clock*. A Figura 16 mostra uma transferência de leitura elementar pelo ponto de vista do mestre.

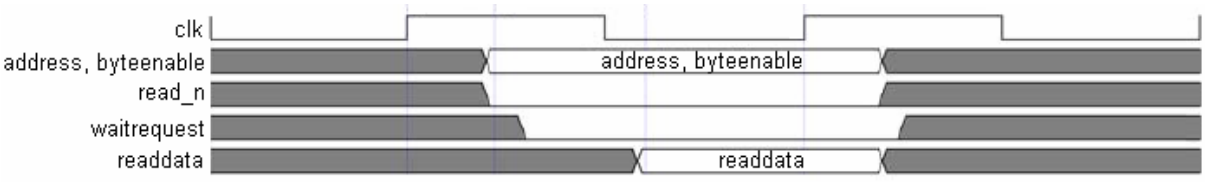

Figura 16 – Transferência elementar no barramento Avalon

A transferência é iniciada na borda de subida do *clock*. Imediatamente depois da primeira borda de subida do *clock* o mestre ativa os sinais *address*, *byteenable* e *read_n*. Se o barramento não puder fornecer o valor de *readdata* dentro do mesmo ciclo, então ele ativa o sinal *waitrequest* e o mestre é obrigado a esperar. Depois que *waitrequest* é desativado, o mestre pode capturar o valor de *readdata* na próxima borda de subida do *clock*.

3.2.4.3 Transferência *streaming* do ponto de vista do escravo

Três sinais da interface do escravo Avalon, *readyfordata, dataavailable e endofpackage*, são usados para o controle de transferências *streaming*. O escravo indica que está pronto para receber dados ativando o sinal *readyfordata* e se está pronto para fornecer dados em operações de leitura ativa o sinal *dataavailable.* O uso do sinal *endofpackage* não é especificado pelo Avalon. *endofpackage* é transmitido do escravo ao mestre e seu uso é dependente dos periféricos.

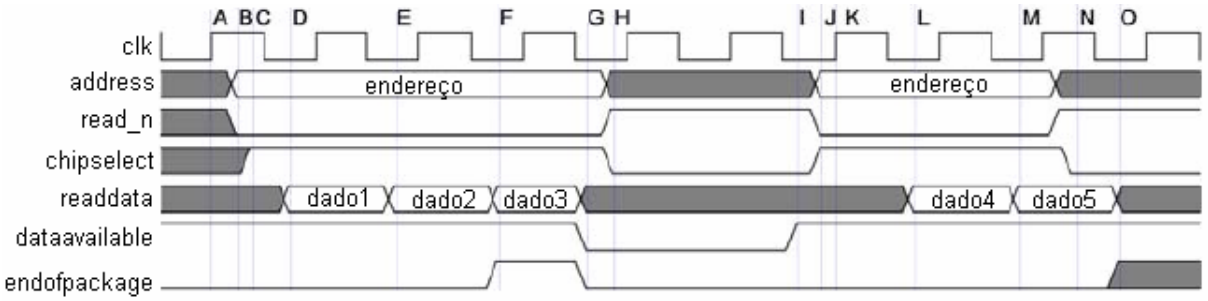

Figura 17 – Transferência *streaming* no barramento Avalon

Os passos da transferência de leitura *streaming* da Figura 17 são os seguintes:

(A) A transferência é iniciada na primeira borda de subida do *clock.*

(B) Logo em seguida o barramento ajusta os sinais *address* e *read_n.*

(C) O endereço é decodificado e *chipselect* é ativado.

(D) O escravo fornece um valor válido para *readdata* antes da próxima borda de subida do *clock*.

(E) Para cada ciclo que *chipselect* e *read_n* estão ativos, o escravo fornece um dado válido para *readdata*.

(F) O escravo pode ativar *endofpackage* em qualquer momento. No exemplo, o escravo ativou *endofpackage* durante o fornecimento do dado3.

(G) Logo depois de ativar *endofpackage* o escravo desativou o sinal *dataavailable* forçando o barramento a adiar a transferência.

(H) O barramento desativa *chipselect, read_n* e *address* em resposta à mudança do sinal *dataavailable*.

(I) Algum tempo depois o escravo reativa *dataavailable.*

(J) Em resposta o barramento reativa os sinal *chipselect*, *read_n* e *address.*

(K) Uma nova transferência *streaming* é iniciada na borda de subida do *clock*.

(L-M) O escravo fornece dados válidos para *readdata* antes da borda de subida do *clock* enquanto *chipselect* e *read_n* estiverem ativos.

(N) O barramento desativa *chipselect* e *read_n* indicando que a transferência *streaming* acabou.

(O) No exemplo, o sinal *dataavailable* permanece ativo indicando que uma nova transferência pode ser iniciada no próximo ciclo.

3.2.4.4 Transferência *streaming* do ponto de vista do mestre

Na interface do mestre existe somente um sinal, chamado *endofpackage*, relacionado apenas à transferências *streaming*. Todos os outros são usados para ambos os tipos de transferência.

O sinal *endofpackage* é passado do escravo para o mestre em cada ciclo de uma transferência. A interpretação desse sinal é dependente do *design* dos dispositivos. Por exemplo, *endofpackage* pode ser usado como um delimitador de pacotes informando o mestre quando inicia e quando termina a transferência de uma seqüência longa.

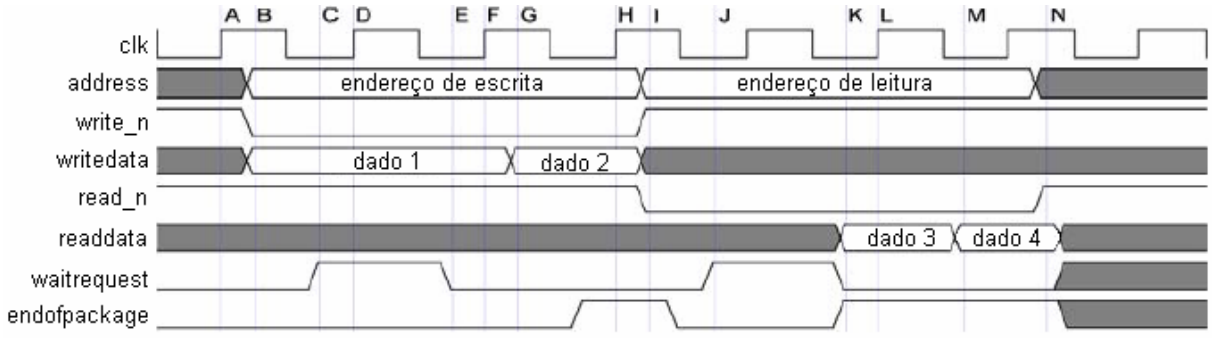

Figura 18 – Transferência *streaming* no barramento Avalon

Os passos da transferência *streaming* de escrita seguida pela transferência *streaming* de leitura da Figura 18 são os seguintes:

(A) A transferência é iniciada na primeira borda de subida do *clock.*

(B) O mestre ativa *address, write_n* e *writedata.*

(C) O barramento ativa *waitrequest* forçando o mestre a esperar.

(D) *waitrequest* está ativo, então o mestre mantém os sinais *address, write_n* e *writedata* constantes.

(E) O barramento desativa *waitrequest.*

(F) O barramento captura *waitrequest* na borda de subida do *clock.*

(G) O mestre mantém *address* e *write_n* e ajusta outro valor para *writedata.*

(H) O mestre captura *endofpackage* e desativa *address, write_n* e *writedata.*

(I) Imediatamente o mestre inicia uma transferência de leitura ativando *read_n* e ajustando um valor válido para *address.*

(J) O barramento ativa *waitrequest* indicando que não pode retornar dados na próxima borda de subida do *clock.*

(K) O barramento desativa *waitrequest* e apresenta um valor válido para *readdata*.

(L) O mestre captura o valor de *readdata* na próxima borda de subida do *clock.*

(M) O mestre mantém *read_n* ativado *address* com um valor válido. Então o barramento fornece outro valor através de *readdata*.

(N) O mestre desativa *read_n* e a transferência é finalizada.

3.3 WISHBONE

Wishbone [16] é uma arquitetura mestre/escravo onde as interfaces mestres iniciam transações de trocas de dados com as interfaces escravos. Os mestres e escravos comunicam-se através de uma interface de interconexões chamada INTERCON. Essa interface de interconexões pode ser descrita como um conjunto de circuitos que permitem que os mestres se comuniquem com os escravos. A figura 19 mostra a interface INTERCON.

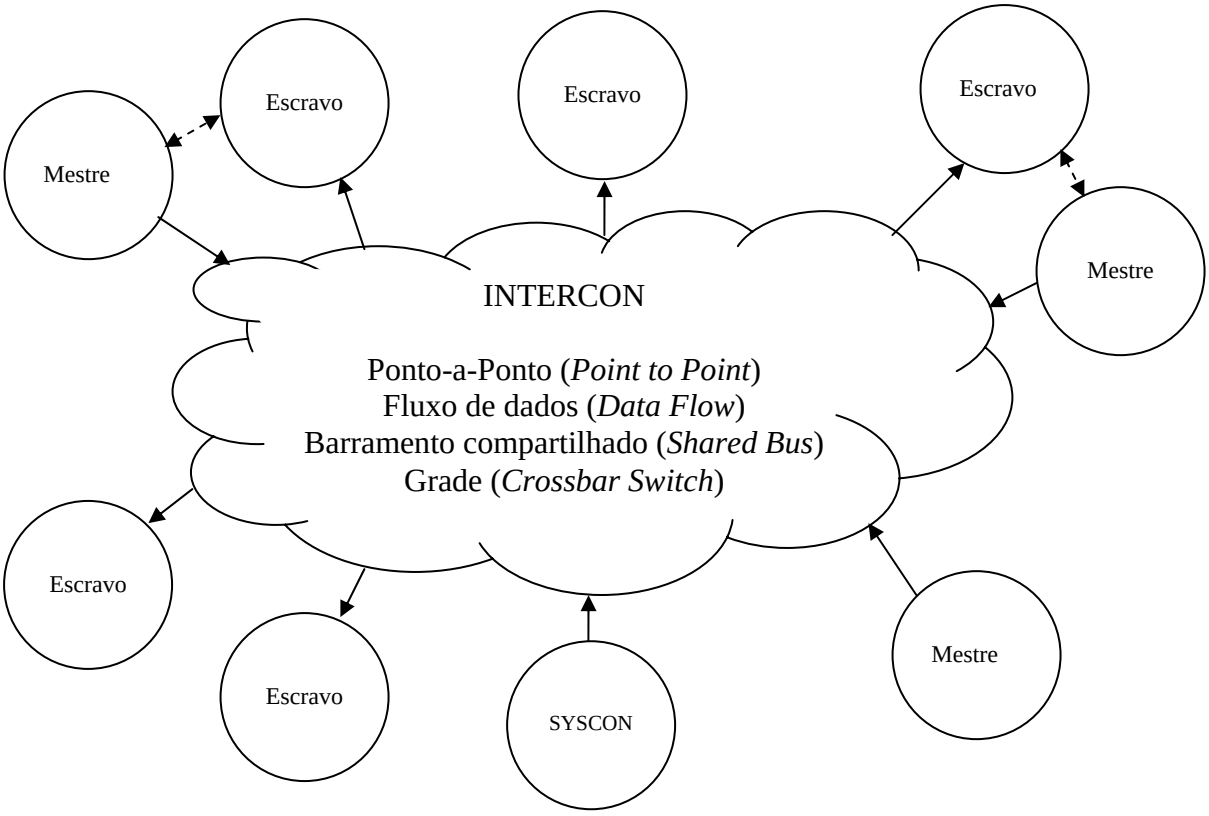

Figura 19 – Barramento WISHBONE

O INTERCON é uma interface variável de interconexões onde existem maneiras diferentes de se conectar mestres e escravos para suprir as necessidades de cada aplicação. Por exemplo, um par mestre-escravo pode conectar-se em topologias ponto-a-ponto, fluxo de dados, barramento compartilhado ou grade de interconexões. As quatro topologias definidas pela especificação do Wishbone são descritas nas próximas seções.

3.3.1 Interconexão ponto a ponto

A interconexão ponto a ponto é a maneira mais simples de se conectar um mestre a um escravo. Nesse esquema, como mostrado na figura 20, a interface de um mestre conecta-se diretamente a interface de um escravo.

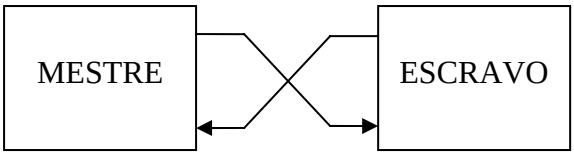

Figura 20 – Interconexão ponto a ponto no Wishbone

3.3.2 Interconexão de fluxo de dados

Interconexão de fluxo de dados é usado quando os dados precisam ser processados de maneira seqüencial. Como pode ser visto na figura 21, cada módulo na arquitetura de fluxo de dados contem uma interface mestre e uma interface escravo e os dados fluem de um módulo a outro através dessas interfaces.

Figura 21 – Interconexão de fluxo de dados no Wishbone

3.3.3 Interconexão por barramento compartilhado

Esse tipo de interconexão é útil quando se deseja conectar dois ou mais mestres com um ou mais escravos. Nessa topologia os mestres iniciam transações com os escravos e o árbitro do barramento determina qual o mestre que será dono do barramento em um determinado ciclo. A figura 22 mostra o diagrama de interconexões por barramento compartilhado.

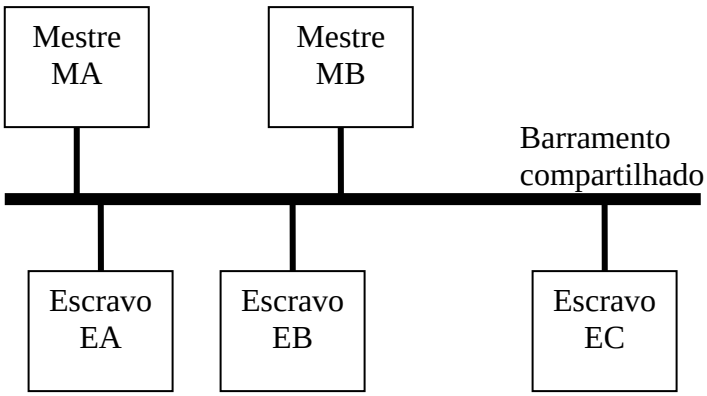

Figura 22 – Barramento compartilhado no Wishbone

3.3.4 Grade de interconexões

Grade de interconexões é usada para se conectar dois ou mais mestres com dois ou mais escravos. Nessa topologia é possível que mais de um mestre esteja realizando transações no mesmo ciclo mas com escravo diferentes. Quando dois ou mais mestres tentam acessar o mesmo escravo em um mesmo ciclo, o árbitro determina qual mestre terá acesso àquele escravo. A figura 23 mostra o diagrama de interconexões por grade de interconexões.

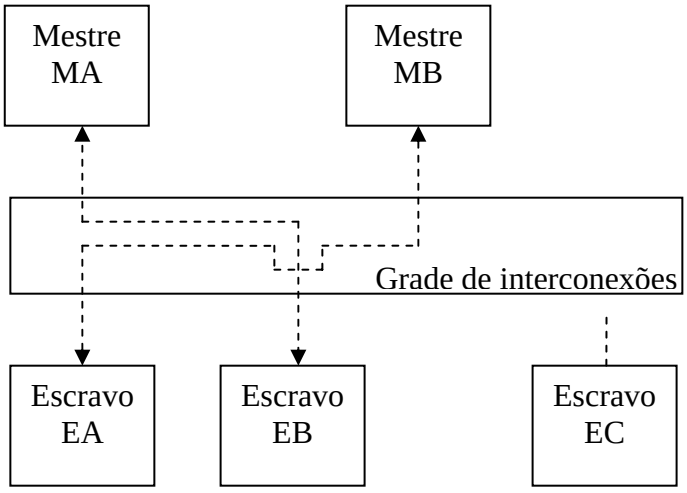

Figura 23 – Grade de interconexões no Wishbone

A taxas de transferências de dados na topologia de grade de interconexões são maiores que na topologia de barramento compartilhado por ser possível realizar transferências em paralelo para diferentes pares mestre-escravo.

3.4 CORECONNECT

A arquitetura de barramento Coreconnect é um padrão de interconexão criado pela IBM para a conexão de múltiplos elementos. A especificação do Coreconnect é composta por três barramentos, PLB [17] (*Processor Local Bus*), OPB [22] (*On-chip Pheripheral Bus*) e DCR [23] (*Device Control Register*). Os três barramentos Coreconnect podem ser vistos na figura 24 e são descritos nas seções seguintes.

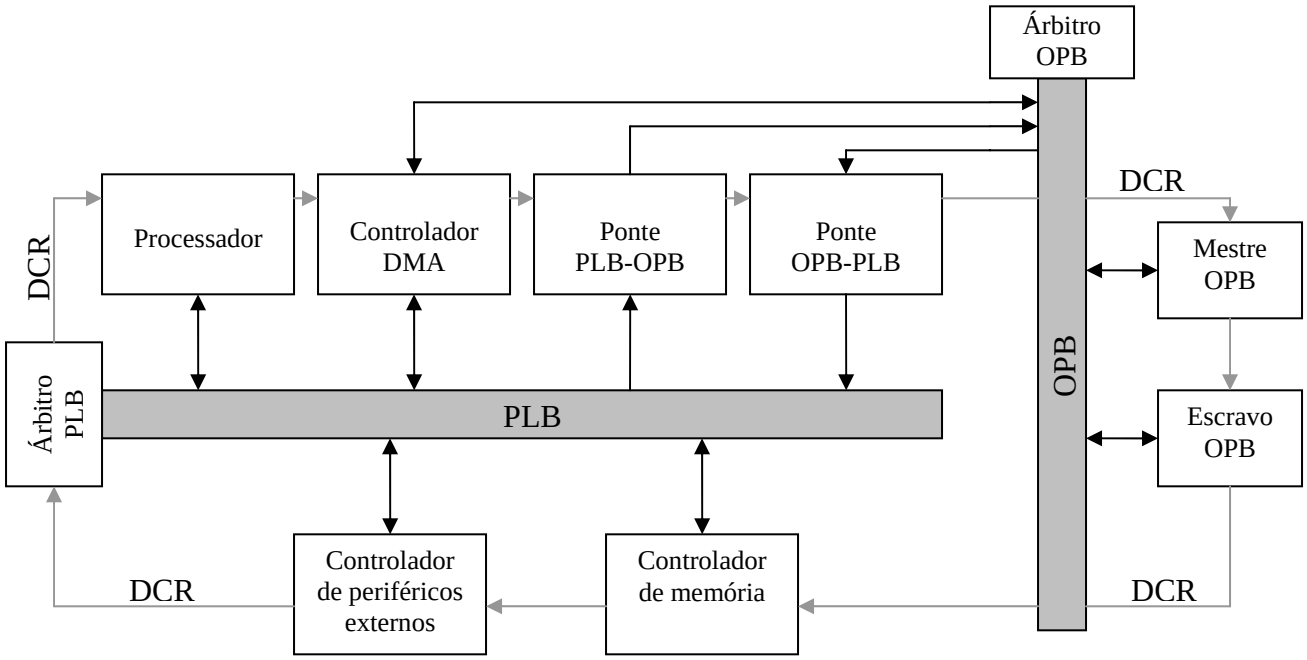

Figura 24 – Barramentos Coreconnect

3.4.1 PLB – Processor Local Bus

PLB é um barramento síncrono de 64 bits de endereços e 128 bits de dados que suporta transferências de escrita e leitura entre dispositivos mestres e escravos que implementam as interfaces do barramento.

Cada mestre é conectado ao barramento através de sinais separados de endereço, dados de escrita, dados para leitura e sinais de controle. Os escravos são conectados ao barramento através de sinais compartilhados e desacoplados para endereço, dados para leitura, dados de escrita e sinais de controle.

O controle de acesso ao barramento PLB é feito por um mecanismo central de arbitragem. O mecanismo de arbitragem é flexível e suporta vários tipos de algoritmos para as mais diversas aplicações.

A conexão entre os barramentos PLB e OPB é feita através de pontes que permitem a transferência de dados entre mestres PLB e escravos OPB.

A arquitetura do barramento PLB suporta até dezesseis mestres e qualquer quantidade de dispositivos escravos, entretanto, o número de mestres e escravos conectados ao barramento afeta diretamente a performance do sistema como um todo.

3.4.2 OPB – On-chip Pheripheral Bus

O barramento OPB é síncrono e projetado para a conexão de dispositivos de baixa performance e baixo consumo de energia não sendo indicado para ser conectado ao processador central de um sistema.

Um mestre do barramento OPB pode fazer transferências de dados com escravos do barramento PLB através de pontes OPB-PLB. A ponte PLB-OPB é um escravo no barramento PLB e ao mesmo tempo mestre no barramento OPB permitindo transferências entre mestres PLB e escravos OPB. Do outro lado, a ponte OPB-PLB é um escravo no barramento OPB e ao mesmo tempo é mestre no barramento PLB permitindo transferências entre mestres OPB e escravos PLB.

3.4.3 DCR – Device Control Register

O DCR é um barramento síncrono usado para transferências de dados entre um mestre DCR, tipicamente os registradores de propósitos gerais de uma CPU, e escravos DCR. O barramento DCR permite a remoção de registradores de configuração do mapa de endereçamento de memória, reduz a carga, aumenta a taxa de transferência do barramento PLB e reduz a latência dos registradores.

As transações do barramento DCR podem ser usadas para controlar a configuração de periféricos, como controladores de memória, controladores DMA, árbitros, pontes, etc. Ou seja, o barramento DCR é usado primeiramente para acessar registradores de estado e controle presentes em dispositivos mestres e escravos nos barramento PLB e OPB.

4 Implementação dos Modelos

Os modelos dos barramentos foram implementados usando-se TLM e as classes do SystemC como base. A estratégia de projeto foi agrupar as funcionalidades e características que os barramentos estudados têm em comum em classes bases e, a partir dessas classes bases criadas, derivar classes especializadas de cada barramento para atender os requisitos de funcionamento específicos.

A especificação AMBA descreve quatro barramentos distintos, *Advanced eXtensible Interface* (AXI), *Advanced High Performance Bus* (AHB), *Advanced System Bus* (ASB) e *Advanced Peripheral Bus* (APB) que, do ponto de vista de implementação, são considerados implementações distintas. Este trabalho contempla a implementação do AHB e deixa os outros barramentos, AXI, ASB e APB, para trabalhos futuros.

O outro barramento implementado e que pode ser visto nas seções seguintes foi o Avalon. Os barramentos Wishbone e Coreconnect são deixados para trabalhos futuros.

Em TLM, as definições das interfaces que conectam os elementos de um sistema são mais importantes do que os detalhes de implementação dessas interfaces. As seções seguintes descrevem, em detalhes, como são as interfaces do trabalho desenvolvido para se conectar os diferentes módulos de um barramento através do SystemC.

4.1 Interface do mestre

Foi definida uma classe chamada *bus_if* que é a classe base que contem os elementos comuns de interface para um mestre de barramento. Essa classe é uma especialização da classe *sc_interface* do SystemC. Todos os métodos são abstratos e devem ser implementados pelo barramento. É através dessa interface que um mestre do barramento conecta-se ao barramento. A Figura 25 mostra o diagrama da classe *bus_if*.

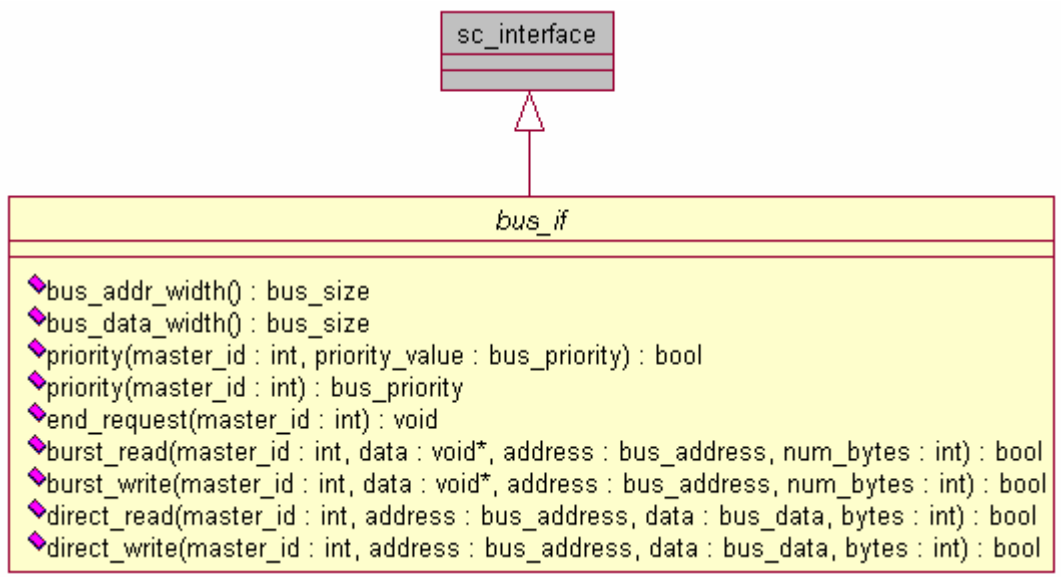

Figura 25 – Diagrama da classe bus_if

Descrição dos métodos:

- **bus_size bus_addr_width():** Retorna o tamanho do barramento de endereços suportado pelo barramento.
- **bus_size bus_data_width():** Retorna o tamanho do barramento de dados suportado pelo barramento.
- **bool priority(int master_id, bus_priority priority_value):** Ajusta o valor da prioridade do mestre que é usada pelo árbitro.
- **bus_priority priority(int master_id):** Retorna o valor da prioridade configurada para o mestre.
- **void end_request(int master_id):** Sinaliza para o barramento que o mestre não está requisitando o barramento. A implementação desse método é dependente da especificação do barramento.
- **bool burst_read(int master_id, void* data, bus_address address, int num_bytes):** Executa uma transferência de leitura em *burst*. A implementação desse método é dependente da especificação do barramento.
- **bool burst_write(int master_id, void* data, bus_address address, int num_bytes):** Executa uma transferência de escrita em *burst*. A implementação desse método é dependente da especificação do barramento.

- **bool direct_read(int master_id, bus_address address, bus_data data, int bytes):** Lê diretamente n bytes do escravo referente ao endereço *address*. Esta é uma interface para depuração do sistema.
- **bool direct_write(int master_id, bus_address address, bus_data data, int bytes):** Escreve diretamente n bytes no escravo referente ao endereço *address*. Esta é uma interface para depuração do sistema.

Os métodos definidos pela interface *bus_if* podem ser descritos como sendo serviços, implementados pelo barramento, que estão disponíveis para os mestres de barramento através das portas mestre.

4.1.1 Interface do mestre AHB

A classe *ahb_bus_if* é a classe que contem os elementos específicos de interface para um mestre de barramento AHB. Essa classe é uma especialização da classe *bus_if*, onde todos os métodos são abstratos e devem ser implementados pelo barramento AHB. É através dessa interface que um mestre do barramento conecta-se ao barramento AHB. A Figura 26 mostra o diagrama da classe *ahb_bus_if*.

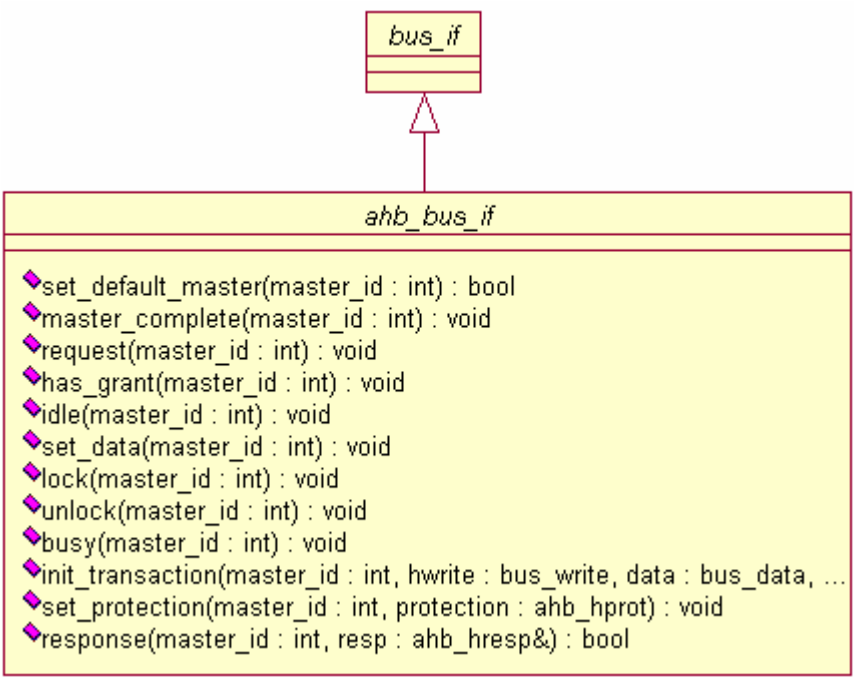

Figura 26 – Diagrama da classe ahb_bus_if

Descrição dos métodos:

- **bool set_default_master(int master_id):** Indica ao barramento e ao árbitro qual o mestre *default*. Na especificação AHB é necessário que se tenha um mestre que será dono do barramento quando todos os outros não estão requisitando o barramento e/ou todos os mestres estão bloqueados esperando a continuação de transações *split*.
- **void master_complete(int master_id):** Indica ao barramento AHB que o mestre completou a execução de um ciclo. Para um mestre AHB é mandatório chamar esse método em cada ciclo de execução.
- **void request(int master_id):** Para fazer uma transferência o mestre deve ser o dono do barramento. Esse método é usado pelo mestre para requisitar o barramento e é equivalente ao sinal HBUSREQx.
- **bool has_grant(int master_id):** Usado pelo mestre para verificar se ele é o dono do barramento para poder executar uma transferência. É equivalente ao sinal HGRANTx. O mestre pode iniciar a transferência somente depois que esse método juntamente com *response*() retornarem verdadeiro.
- **void idle(int master_id):** O mestre pode inserir transferências *idle* usando esse método. O método *idle*() também é usado para abortar uma transferência em *burst* quando o

próximo endereço já foi escrito no barramento. Esse método é equivalente a fazer o sinal HTRANS igual à IDLE.

- **void set_data(int master_id):** Esse método pode ser definido como sendo a fase de dados de uma transferência e só deve ser chamado depois que o método *response*() retornar verdadeiro. Em uma operação de escrita indica que o mestre forneceu um dado válido para ser transferido para o escravo.
- **void lock(int master_id):** Esse método indica que o mestre pretende realizar uma transferência sem ser interrompido mesmo que algum outro mestre de maior prioridade esteja requisitando o barramento. Esse método é equivalente a ativar o sinal HLOCKx.
- **void unlock(int master_id):** Esse método é equivalente a desativar o sinal HLOCKx. Indica que o barramento pode ser liberado para algum mestre de maior prioridade que esteja requisitando o barramento.
- **void busy(int master_id):** O mestre dono do barramento pode introduzir um ciclo BUSY quando está ocupado e não é capaz de fornecer ou processar dados no próximo ciclo. Esse método é equivalente a fazer o sinal HTRANS igual à BUSY.
- **void init_transaction(int master_id, bus_write hwrite, bus_data data, bus_address address, ahb_hburst burst_mode, bus_size hsize, bool unaligned_access = false, int master_domain_info = 0):** Depois de requisitar e receber o barramento o mestre deve iniciar a transação.
- **void set_protection(int master_id, ahb_hprot protection):** Ajusta as informações de proteção para a próxima transferência. É equivalente a ajustar o sinal HPROT. Se esse método não for chamado, o valor padrão de proteção é usado (AHB_DATA | AHB_PRIVILEGED). O valor de proteção é opcional e pode não ser levado em conta pelos escravos.
- **bool response(int master_id, ahb_hresp& resp):** O estado do barramento e do escravo deve ser verificado por esse método. O valor de retorno desse método é equivalente ao sinal HREADY e o parâmetro de retorno *resp* é equivalente ao sinal HRESP ambos retornados pelo escravo envolvido na transferência.

O fragmento de código no Algoritmo 4 exemplifica como é a implementação de um mestre do barramento AHB para realizar uma transferência simples, sem ciclos de espera, utilizando os serviços fornecidos pela interface *ahb_bus_if*:

```
...
m_port.request(); // Requisita o barramento

// Inicia a transação
m_port.init_transaction(WR, data, address, SINGLE, SIZE_32);

// Espera até ser o dono do barramento
while ((!m_port.has_grant()) || (!m_port.response(resp))) wait();

// É o dono do barramento, pode proseguir com a operação
m_port.set_data();

// Espera um ciclo pela resposta
wait();

// Verifica a resposta do escravo
bool hready = m_port.response(resp);

// Verifica se o escravo está pronto
if (hready)
{
   if (resp == OKAY) // Ok. A transação foi executada com sucesso.
   {
        m_port.end_request(); // Fim da transação. Faz HBUSREQ = 0.
   }
}
...
```

Algoritmo 4 – Fragmento de código de um mestre AHB para uma transferência simples

Primeiramente, antes de fazer uma transferência, o mestre deve requisitar o barramento através do método *request*. O método *init_transaction* externaliza os sinais relevantes para a transferência. Depois disso, antes de prosseguir, é necessário que o mestre monitore se ele é o dono do barramento através da chamada *has_grant*. Enquanto não for o dono, deve esperar.

O método *set_data* transfere os dados para escravo através do barramento e deve ser chamado somente quando o mestre for o dono do barramento. Depois de feita a transferência o mestre deve verificar a resposta do barramento/escravo através da chamada *response*.

4.1.2 Interface do mestre Avalon

A classe *avalon_bus_if* é a classe que contem os elementos específicos de interface para um mestre de barramento Avalon. Essa classe é uma especialização da classe *bus_if*, todos os métodos são abstratos e devem ser implementados pelo barramento Avalon. É através dessa

interface que um mestre do barramento conecta-se ao barramento Avalon. A Figura 27 mostra o diagrama da classe *avalon_bus_if*.

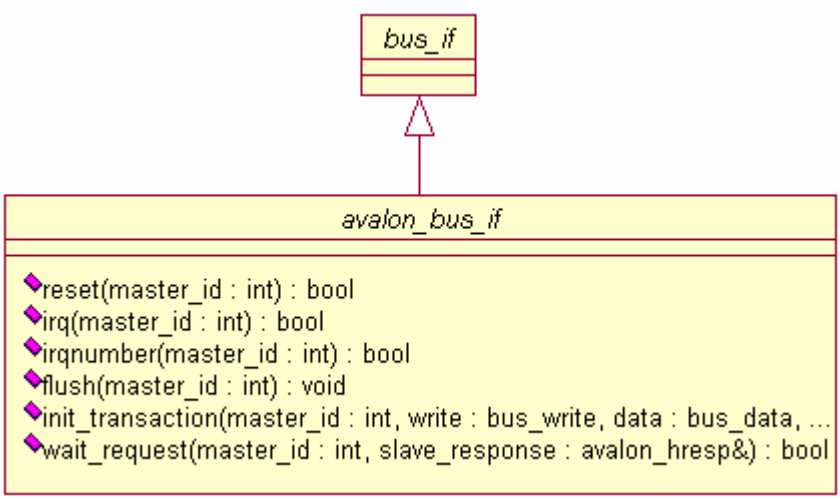

Figura 27 – Diagrama da classe avalon_bus_if

Descrição dos métodos:

- **bool reset(int master_id):** Esse método representa o sinal global de reinicialização do sistema. Sua implementação depende dos requisitos do mestre. É equivalente ao sinal RESET.
- **bool irq(int master_id):** Esse método é chamado pelo barramento quando o mestre precisa atender a um escravo. É equivalente ao sinal IRQ (*interrupt request*).
- **int irqnumber(int master_id):** Para tratar corretamente uma chamada ao método irq() o mestre deve verificar qual o número da interrupção. Esse método representa o sinal IRQNUMBER.
- **void flush(int master_id):** Esse método é usado para transferências de leitura com latência. O mestre pode cancelar qualquer operação de leitura com latência que estiver pendente chamando esse método. É equivalente ao sinal FLUSH.
- **void init_transaction(int master_id, bus_write write, bus_data data, bus_address address, avalon_byteenable byteenable):** Depois de requisitar e receber o barramento o mestre deve iniciar a transação.
- **bool wait_request(int master_id, avalon_hresp& slave_response):** O estado do barramento e do escravo deve ser verificado por esse método. O valor de retorno desse

método é equivalente ao sinal WAITREQUEST e o parâmetro de retorno *slave_response* representa os sinais READYFORDATA, DATAAVAILABLE e ENDOFPACKAGE retornados pelo escravo envolvido na transferência. Quando esse método retorna verdadeiro, o mestre é obrigado a esperar antes de prosseguir com a transferência.

O fragmento de código no Algoritmo 5 exemplifica como é a implementação de um mestre do barramento Avalon para realizar uma transferência simples utilizando os serviços fornecidos pela interface *avalon_bus_if*:

```
...
m_port.init_transaction(RD, data, address, 0xF); // 0xF=1111b - Lê todos os bytes da palavra de 32 bits.

// Espera enquanto o sinal waitrequest estiver ativo
while (m_port.wait_request(slave_response))
{
   wait();
}

// Verifica a resposta do escravo
if (slave_response == DATA_AVAILABLE)
{
   m_port.end_request(); // Ok. Transação executada.
}
...
```

Algoritmo 5 – Fragmento de código de um mestre Avalon para uma transferência simples

O mestre Avalon, para iniciar uma transferência, deve chamar o método *init_transaction* passando os dados relevantes da transferência. Depois disso deve monitorar o sinal *waitrequest* que é a indicação de que a transferência foi finalizada. Depois de finalizada a transferência, o mestre pode verificar o estado reportado pelo escravo através do parâmetro *slave_response*.

4.2 Portas Mestre

A classe *bus_master_port* é classe base para as classes que implementam as portas mestre que são usadas pelos mestres para se conectarem ao barramento. A Figura 28 mostra o diagrama das portas mestre.

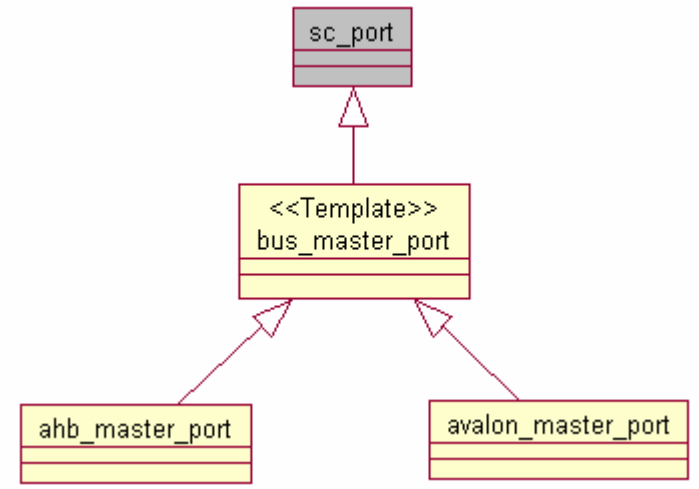

Figura 28 – Diagrama das classes de portas mestre

Os métodos implementados pela classe *bus_master_port* são descritos na seção *bus_if*.

Os métodos implementados pela classe *ahb_master_port* são descritos na seção ahb_*bus_if*.

Os métodos implementados pela classe *avalon_master_port* são descritos na seção avalon_*bus_if*.

4.3 Interface do escravo

A classe *bus_slave_if* é a classe base que contem os elementos comuns de interface para um escravo de barramento. Essa classe é uma especialização da classe *sc_interface* do SystemC. Todos os métodos são abstratos e devem ser implementados pelos escravos. É através dessa interface que os escravos conectam-se ao barramento. A Figura 29 mostra o diagrama da classe *bus_slave_if*.

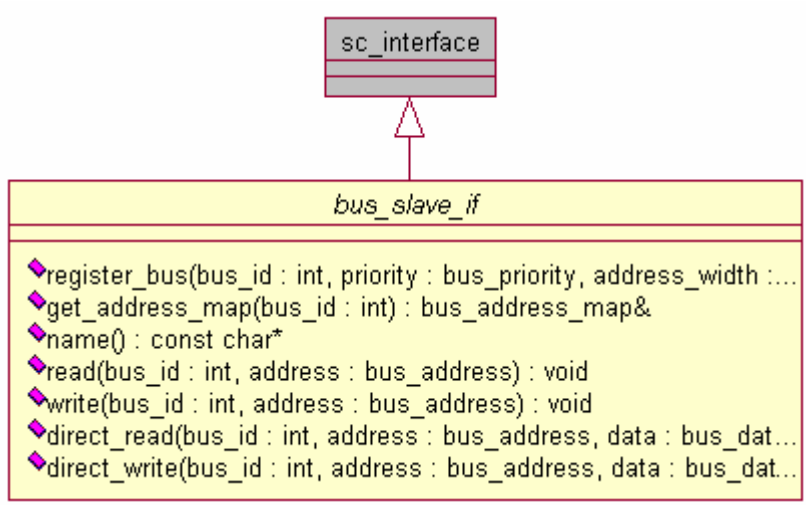

Figura 29 – Diagrama da classe bus_slave_if

Descrição dos métodos:

- **int register_bus(int bus_id, bus_priority priority, bus_size address_width, bus_size data_width):** No final da fase de elaboração da simulação o barramento chama o método *register_bus*() de todos os escravos conectados para transmitir informações de configuração. O escravo deve retornar seu identificador que é único no sistema.
- **const bus_address_map& get_address_map(int bus_id):** Esse método é chamado pelo barramento para fazer o mapeamento das faixas de endereços utilizadas pelos escravos. Esses valores são utilizados pelo decodificador de endereço para definir qual o escravo referente a um dado endereço.
- void **read(int bus_id, bus_address address):** Esse método sinaliza para o escravo que a transação desse ciclo é de leitura e o endereço é dado pelo parâmetro *address*. O barramento garante que o endereço dado está dentro da faixa de endereços do escravo em questão.
- **void write(int bus_id, bus_address address):** Esse método sinaliza para o escravo que a transação desse ciclo é de escrita e o endereço é dado pelo parâmetro *address*. O barramento garante que o endereço dado está dentro da faixa de endereços do escravo em questão.
- **bool direct_read(int bus_id, bus_address address, bus_data data, int bytes):** Método para acesso direto de leitura do escravo pelo mestre. Esse método é usado somente para

depuração do sistema. Esse método não consome ciclos de barramento e é executado com tempo de simulação igual a zero.

- **bool direct_write(int bus_id, bus_address address, bus_data data, int bytes):** Método para acesso direto de escrita no escravo pelo mestre. Esse método é usado somente para depuração do sistema. Esse método não consome ciclos de barramento e é executado com tempo de simulação igual a zero.

Os métodos definidos pela interface *bus_slave_if* podem ser descritos como sendo serviços, implementados pelos escravos, que estão disponíveis para o barramento através das portas escravo.

4.3.1 Interface do escravo AHB

A classe *ahb_slave_if* é a classe que contem os elementos específicos de interface para um escravo do barramento AHB. Essa classe é uma especialização da classe *bus_slave_if*. Todos os métodos são abstratos e devem ser implementados pelos escravos AHB. É através dessa interface que o barramento AHB conecta-se aos escravos. A Figura 30 mostra o diagrama da classe *ahb_slave_if*.

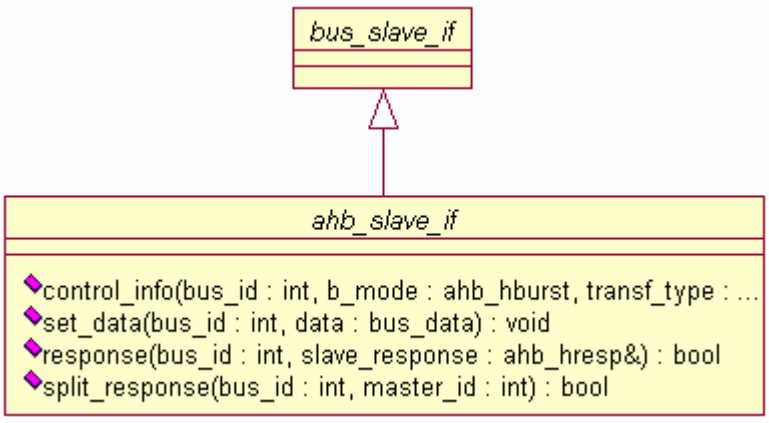

Figura 30 – Diagrama da classe ahb_slave_if

Descrição dos métodos:

- **void control_info(int bus_id, ahb_hburst burst_mode, ahb_htrans transf_type, ahb_hprot protection, int master_id, bool locked_transfer, bus_size transfer_size):** Esse método passa ao escravo as informações de controle da transferência corrente.
- **void set_data(int bus_id, bus_data data):** Esse método informa o endereço do buffer onde o escravo deve ler ou escrever. Para uma operação de escrita, o buffer estará previamente inicializado com os dados que serão repassados ao escravo. Esse método é chamado uma vez por transferência de dados.
- **bool response(int bus_id, ahb_hresp& slave_response):** Esse método consulta o estado do escravo a respeito da operação de escrita ou leitura do último ciclo. O valor de retorno representa o sinal HREADY que quando igual a falso indica que o escravo não está pronto para completar a transação. O valor de retorno *slave_response* representa o sinal HRESP e os valores possíveis são OKAY, RETRY, ERROR ou SPLIT.
- **bool split_response(int bus_id, int master_id):** O barramento usa esse método para verificar o estado de uma transferência que foi interrompida por ser uma transação *split*. O retorno igual a verdadeiro indica que a transação pode continuar. Esse método só deve ser implementado por escravos que suportam esse tipo de transação.

O fragmento de código no Algoritmo 6 exemplifica como é a implementação de um escravo do barramento AHB para realizar uma transferência simples, sem ciclos de espera, onde o barramento utiliza os serviços fornecidos pela interface *ahb_slave_if*:

```
...
void AHB_Slave::set_data(int bus_id, bus_data data)
{
    // Calcula o endereço interno.
    bus_address internal_address = m_current_address - m_base_addr;

    // Calcula o número de bytes da transferência
    int transfer_bytes = BUSSIZE_TO_BYTES(m_current_transfer_size);

    if (m_current_operation == WR) // É uma operação de escrita
    {
      // Armazena e/ou processa o dado
      memcpy(m_pdata + internal_address, data, transfer_bytes);
    }
    else // RD - É uma operação de leitura
    {
      // Transfere o dado relativo ao endereço fornecido para data
      memcpy(data, m_pdata + internal_address, transfer_bytes);
    }
```

```
    // Sinaliza a resposta ao barramento
    m_current_hready_status = true;
    m_current_response_status = OKAY;
}
...
```

Algoritmo 6 – Fragmento de código de um escravo AHB para uma transferência simples

A chamada do método *set_data* pelo barramento é a última fase de uma transferência vista pelo lado do escravo AHB. É na implementação desse método que o dado é tratado e a resposta que será enviada ao barramento/mestre deve ser ajustada. A implementação desse método depende da natureza do dispositivo escravo.

4.3.2 Interface do escravo Avalon

A classe *avalon_slave_if* é a classe que contem os elementos específicos de interface para um escravo do barramento Avalon. Essa classe é uma especialização da classe *bus_slave_if*. Todos os métodos são abstratos e devem ser implementados pelos escravos Avalon. É através dessa interface que o barramento Avalon conecta-se aos escravos. A Figura 31 mostra o diagrama da classe *avalon_slave_if*.

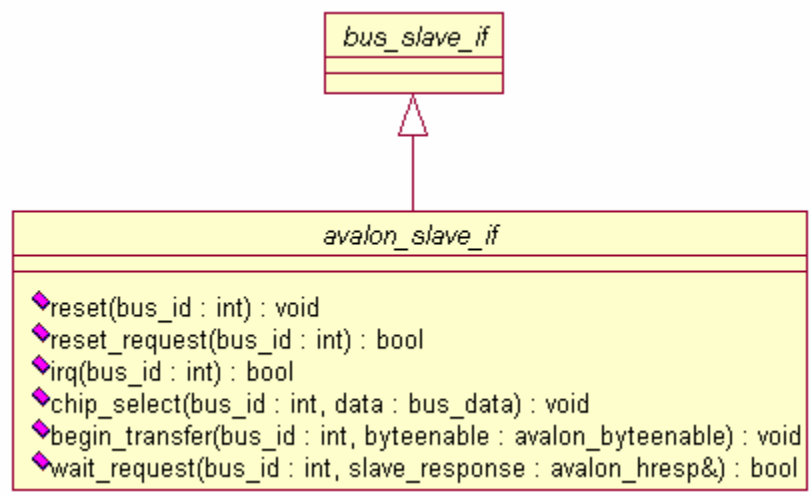

Figura 31 – Diagrama da classe avalon_slave_if

Descrição dos métodos:

- **void reset(int bus_id):** Esse método representa o sinal global de reinicialização do sistema. Sua implementação depende dos requisitos do escravo. É equivalente ao sinal RESET.
- **bool reset_request(int bus_id):** Representa o sinal RESETREQUEST e permite que um escravo reinicialize o sistema inteiro.
- **bool irq(int bus_id):** Representa o sinal IRQ, é chamado pelo barramento para verificar se o escravo precisa ser atendido por algum mestre.
- **void chip_select(int bus_id, bus_data data):** Esse método representa o sinal CHIPSELECT. É nesse método que a operação final de escrita ou leitura é implementada pelo escravo.
- **void begin_transfer(int bus_id, avalon_byteenable byteenable):** Esse método fornece ao escravo os parâmetros de controle da transferência. Representa o sinal BEGINTRANSFER de um escravo Avalon.
- **bool wait_request(int bus_id, avalon_hresp& slave_response):** Esse método consulta o estado do escravo a respeito da operação de escrita ou leitura do último ciclo. O valor de retorno representa o sinal WAITREQUEST que quando igual a verdadeiro indica que o escravo não está pronto para completar a transação. O valor de retorno *slave_response* representa os sinais READYFORDATA, DATAAVAILABLE e ENDOFPACKAGE retornados pelo escravo envolvido na transferência.

O fragmento de código no Algoritmo 7 exemplifica como é a implementação de um escravo do barramento Avalon para realizar uma transferência simples, sem ciclos de espera, onde o barramento utiliza os serviços fornecidos pela interface *avalon_slave_if*:

```
...
void Avalon_Slave::chip_select(int bus_id, bus_data data)
{
    // Calcula o endereço interno.
    bus_address internal_address = m_current_address - m_base_addr;
    unsigned char* pData = (unsigned char *) data;

    // Avalon faz transferências de 32 bits. byteenable deve ser verificado
    int transfer_bytes = 4;

    // Verifica o tipo de transferência
    if (m_current_operation == WR) // Operação de escrita
    {
         //  Recebe/processa o dado "data" levando em conta o sinal byteenable
         if (m_current_byteenable & 0x08) // 1000b
              m_pdata[internal_address + 3] = pData[3];

         if (m_current_byteenable & 0x04) // 0100b
               m_pdata[internal_address + 2] = pData[2];

         if (m_current_byteenable & 0x02) // 0010b
              m_pdata[internal_address + 1] = pData[1];

         if (m_current_byteenable & 0x01) // 0001b
              m_pdata[internal_address + 0] = pData[0];
    }
    else //  RD - Operação de leitura
    {
      // Transfere o dado para "data" levando em conta o sinal byteenable
      if (m_current_byteenable & 0x08) // 1000b
                pData[3] = m_pdata[internal_address + 3];

      if (m_current_byteenable & 0x04) // 0100b
                pData[2] = m_pdata[internal_address + 2];

      if (m_current_byteenable & 0x02) // 0010b
                pData[1] = m_pdata[internal_address + 1];

      if (m_current_byteenable & 0x01) // 0001b
                pData[0] = m_pdata[internal_address + 0];
    }

    // Sinaliza a resposta ao barramento
    m_current_response_status = READY_FOR_DATA | DATA_AVAILABLE;
    m_current_waitrequest_status = false;
}
...
```

Algoritmo 7 – Fragmento de código de um escravo Avalon para uma transferência simples

A chamada do método c*hip_select* pelo barramento é a última fase de uma transferência vista pelo lado do escravo Avalon. É na implementação desse método que o dado é tratado e a resposta que será enviada ao barramento/mestre deve ser ajustada. A implementação desse método depende da natureza do dispositivo escravo.

4.4 Portas escravo

A classe *bus_slave_port* é classe base para as classes que implementam as portas escravo que são usadas pelos barramentos para se conectarem aos escravos. O diagrama da classe *bus_slave_port* é mostrado na Figura 32.

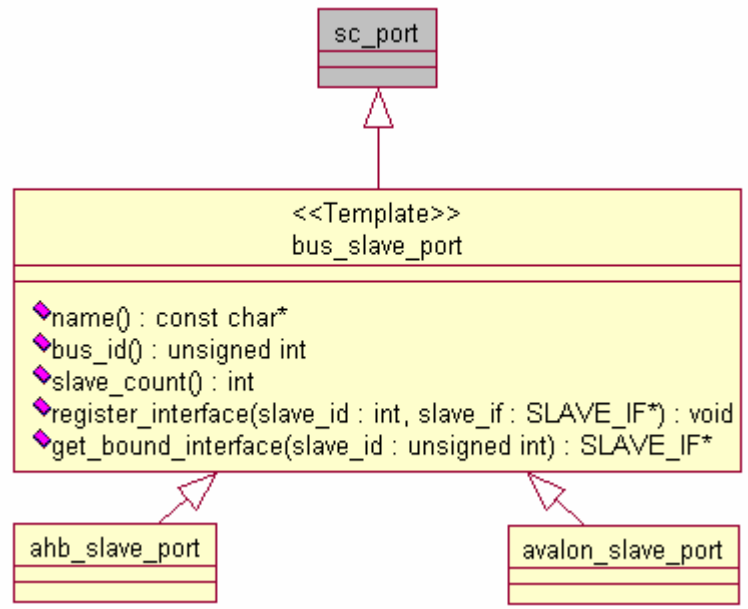

Figura 32 – Diagrama das classes de portas escravo

Descrição dos métodos:

- **const char* name():** Retorna o nome do *sc_port* configurado no SystemC.
- **unsigned int bus_id():** Retorna o ID do barramento que instanciou a porta.
- **int slave_count():** Retorna o número de interfaces de escravos que estão conectadas nessa porta.
- **void register_interface(int slave_id, SLAVE_IF* slave_if):** Esse método é chamado quando uma interface de escravo conecta-se a porta.
- **SLAVE_IF* get_bound_interface(unsigned int slave_id):** Retorna a referência da interface do escravo referente a *slave_id*.

As classes ahb_slave_port e avalon_slave_port são as classes instanciáveis de portas escravo para os barramentos AHB e Avalon respectivamente.

4.5 Interface do decodificador de endereços

A classe *bus_decoder_if* contem os elementos de interface para o decodificador de endereços do barramento. Essa classe é uma especialização da classe *sc_interface* definida pelo SystemC. Todos os métodos são abstratos e devem ser implementados pelos decodificadores de endereços. É através dessa interface que o barramento conecta-se a instância do decodificador. A Figura 33 mostra o diagrama da classe *bus_decoder_if*.

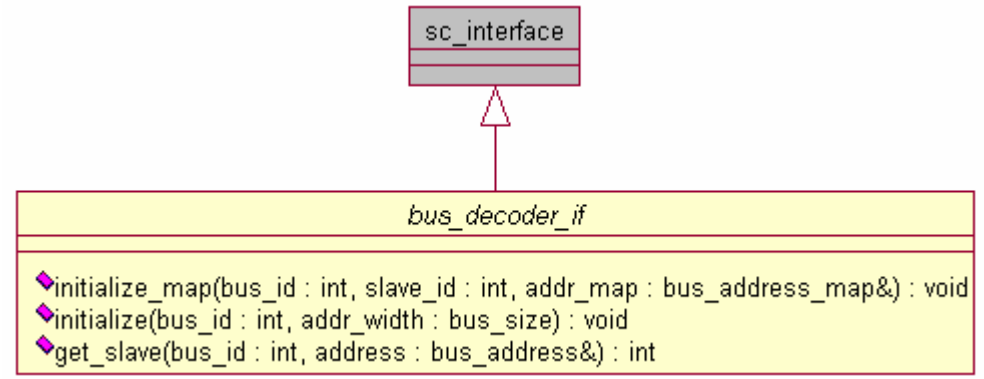

Figura 33 – Diagrama da classe bus_decoder_if

Descrição dos métodos:

- **void initialize_map(int bus_id, int slave_id, const bus_address_map& addr_map):** Esse método é chamado uma vez para cada escravo conectado ao barramento e é utilizado para informar o decodificador de endereços sobre as faixas de endereços utilizados por cada um dos escravo.
- **void initialize(int bus_id, bus_size addr_width):** O método *initialize* é chamado pelo barramento na fase de elaboração da simulação para informar o decodificador de endereços o tamanho do barramento de endereços.
- **int get_slave(int bus_id, bus_address& address):** Esse é o método responsável pela decodificação de endereços e é chamado pelo barramento sempre que um endereço

precise ser decodificado. Retorna o ID do escravo correspondente ao endereço *address* passado como parâmetro.

Os métodos definidos pela interface *bus_decoder_if* podem ser descritos como sendo serviços, implementados pelos decodificadores de endereços, que estão disponíveis para o barramento através da porta (*sc_port* do SystemC) de conexão do decodificador de endereços.

O decodificador de endereços possui a interface mais simples do sistema e, para os barramentos estudados, não é necessário fazer nenhuma especialização da interface porque a especificação da interface base *bus_decoder_if* já atende a todos os requisitos de ambos os barramentos, AHB e Avalon.

4.6 Decodificador de endereços

A classe *bus_default_decoder* é uma implementação básica de um decodificador de endereços. *bus_default_decoder* implementa todos os métodos definidos pela interface *bus_decoder_if*. O diagrama da classe *bus_default_decoder* é mostrado na Figura 34.

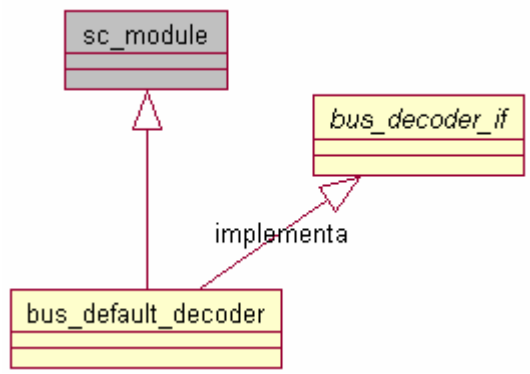

Figura 34 – Diagrama de classe do decodificador de endereços

O fragmento de código no Algoritmo 8 exemplifica como é a implementação do decodificador de endereços do qual o barramento utiliza os serviços fornecidos pela interface *bus_decoder_if* para descobrir qual é o escravo relativo a uma transferência em execução.

```
...
int Bus_default_decoder::get_slave(int bus_id, bus_address& address)
{
   // percorre a lista de escravos verificando as faixas de endereços
   for (int slave = 0; slave < address_map_list_count; slave++)
   {
      // Pega o mapa de endereços do escravo
      bus_address_map* slave_addr_map = address_map_list[slave].address_map;

      // Verifica se o endereço está dentro da faixa de endereços do escravo
      if ((slave_addr_map->size() > 0)&&
         (address >= (*slave_addr_map)[0].normal_start_addr)&&
         (address <= (*slave_addr_map)[0].normal_end_addr))
      {
         // Se o endereço está dentro da faixa, retorna o ID do escravo
         return address_map_list[slave].slave_id;
      }
   }

   // Endereço inválido.
   return 0;
}
...
```

Algoritmo 8 – Fragmento de código do decodificador de endereços

O decodificador de endereços é informado sobre o mapa de endereçamento de todos os escravos conectados ao barramento na fase de configuração da simulação através de chamadas ao método *initialize_map.*

Durante a simulação, o decodificador de endereços é invocado através do método *get_slave* para retornar qual o escravo relativo a um endereço. O código do Algoritmo 8 somente percorre a lista de escravos verificando em qual escravo o endereço dado se encaixa.

4.7 Interface do árbitro

A classe *bus_arbiter_if* contem os elementos de interface para o árbitro do barramento. Essa classe é uma especialização da classe *sc_interface* definida pelo SystemC. Todos os métodos são abstratos e devem ser implementados pelos árbitros. É através dessa interface que o barramento conecta-se a instância do árbitro. A Figura 35 mostra o diagrama da classe *bus_arbiter_if*.

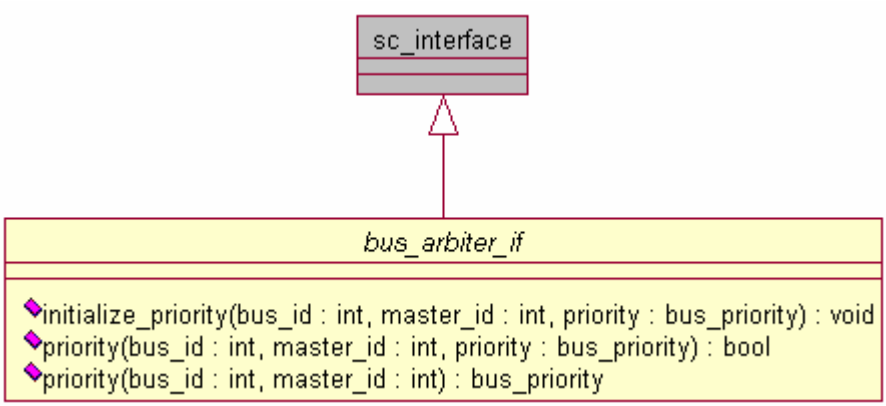

Figura 35 – Diagrama da classe bus_arbiter_if

Descrição dos métodos:

- **void initialize_priority(int bus_id, int master_id, bus_priority priority):** Depois da fase de elaboração, para cada mestre conectado, o barramento chama esse método para informar o árbitro a prioridade inicial de arbitragem.
- **bool priority(int bus_id, int master_id, bus_priority priority):** Esse método é usado para mudar a prioridade de um mestre durante a simulação.
- **bus_priority priority(int bus_id, int master_id):** Esse método informa a prioridade corrente de um mestre.

Os métodos definidos pela interface *bus_arbiter_if* podem ser descritos como sendo serviços, implementados pelos árbitros, que estão disponíveis para o barramento através da porta (*sc_port* do SystemC) de conexão do árbitro.

4.7.1 Interface do árbitro AHB

A classe *ahb_arbiter_if* é a classe que contem os elementos específicos de interface para um árbitro do barramento AHB. Essa classe é uma especialização da classe *bus_arbiter_if*. Todos os métodos são abstratos e devem ser implementados pelos árbitros AHB. É através dessa interface que o barramento AHB conecta-se ao árbitro. A Figura 36 mostra o diagrama da classe *ahb_arbiter_if*.

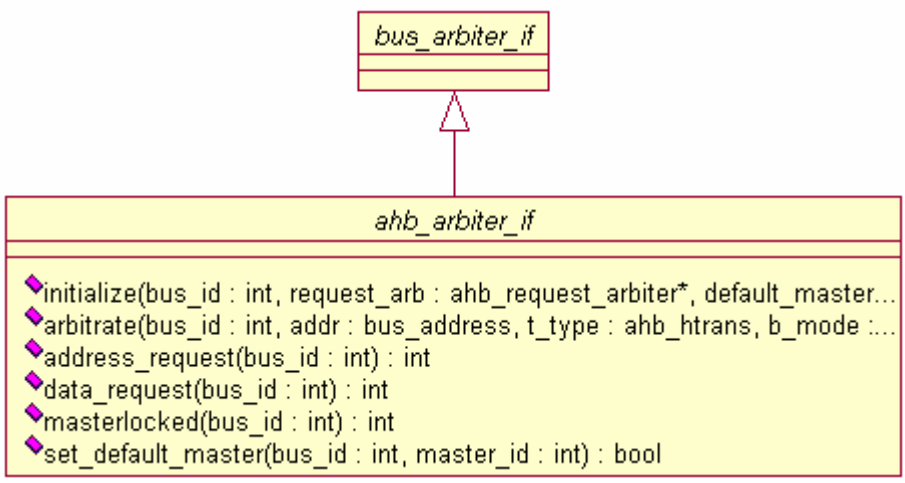

Figura 36 – Diagrama da classe ahb_arbiter_if

Descrição dos métodos:

- **void initialize(int bus_id, ahb_request_arbiter* request_arb, int default_master_id):** Depois da fase de elaboração da simulação, o barramento chama esse método para informar ao árbitro o endereço da estrutura que contem os sinais internos para troca de informação de arbitragem que ocorre entre o barramento e árbitro durante a execução.
- **bool arbitrate(int bus_id, bus_address address, ahb_htrans transfer_type, ahb_hburst burst_mode, ahb_hresp status_response, bool ready_response):** Método que faz a arbitragem durante a simulação.
- **int address_request(int bus_id):** Método que indica qual o mestre que é dono do barramento de endereços no ciclo corrente.
- **int data_request(int bus_id):** Método que indica qual o mestre que é dono do barramento de endereços no ciclo corrente.
- **int masterlocked(int bus_id):** Método que indica qual o ID do mestre que está executando uma transferência indivisível. Retorna zero se nenhum mestre estiver executando esse tipo de transferência.
- **bool set_default_master(int bus_id, int master_id):** Esse método informa o árbitro qual é o mestre *default* que é dono do barramento quando nenhum outro mestre está requisitando o barramento.

4.7.2 Interface do árbitro Avalon

A classe *avalon_arbiter_if* é a classe que contem os elementos específicos de interface para um árbitro do barramento Avalon. Essa classe é uma especialização da classe *bus_arbiter_if*. Todos os métodos são abstratos e devem ser implementados pelos árbitros Avalon. É através dessa interface que o barramento Avalon conecta-se ao árbitro. A Figura 37 mostra o diagrama da classe *avalon_arbiter_if*.

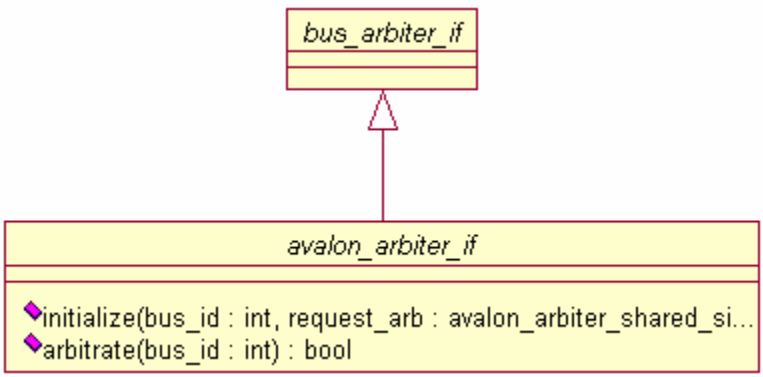

Figura 37 – Diagrama da classe avalon_arbiter_if

Descrição dos métodos:

- **void initialize(int bus_id, avalon_arbiter_shared_sig* request_arb, Bus_decoder_if* decoder_if):** Depois da fase de elaboração da simulação, o barramento chama esse método para informar ao árbitro o endereço da estrutura que contem os sinais internos para troca de informação de arbitragem que ocorre entre o barramento e árbitro durante a execução.
- **bool arbitrate(int bus_id):** Método que faz a arbitragem durante a simulação.

4.8 Árbitro AHB

A classe *ahb_default_arbiter* é uma implementação de um árbitro para o barramento AHB. *ahb_default_arbiter* implementa todos os métodos definidos pela interface *ahb_arbiter_if*. O diagrama da classe *ahb_default_arbiter* é mostrado na Figura 38.

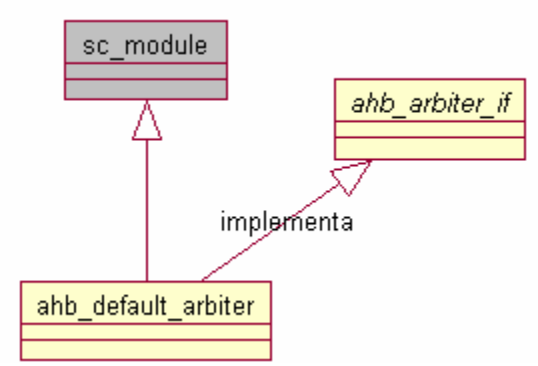

Figura 38 – Diagrama do árbitro do barramento AHB

O fragmento de código no Algoritmo 9 exemplifica como é a implementação de um árbitro AHB do qual o barramento utiliza os serviços fornecidos pela interface *ahb_arbiter_if* para fazer a arbitragem. Esse algoritmo de arbitragem leva em conta se um mestre está executando uma transferência indivisível e ou transferência em *burst* e não as interrompe. Nos outros casos, sempre o mestre de maior prioridade receberá o barramento.

```
...
bool AHB_default_arbiter::arbitrate(int bus_id, bus_address address,
     ahb_htrans transfer_type, ahb_hburst burst_mode, ahb_hresp
     status_response, bool ready_response)
{
   if ((m_masterlocked == m_data_owner) && (m_request_arb[m_data_owner].lock))
   {
     // O dono atual do barramento está executando uma transferência "locked".
     // Então, não faz nada e retorna.
     return true;
   }

   // Para a implementação corrente do árbitro AHB, quando um mestre
   // está executando uma transferência em burst não é interrompido.
   if ((transfer_type != IDLE)&&(burst_mode != SINGLE))
   {
     // Então, não faz nada e retorna.
     return true;
   }

   m_data_owner = m_address_owner;

   // Procura pelo mestre de maior prioridade que está requisitando o barramento.
   bus_priority current_priority = (bus_priority) -1;
   int master = m_default_master_id;
   for (int k = 0; k < master_count; k++)
   {
      if ((m_request_arb[master_list[k]].request == true) &&
          (m_request_arb[master_list[k]].priority < current_priority))
      {
          current_priority = m_request_arb[master_list[k]].priority;
          master = master_list[k];
      }

      m_request_arb[master_list[k]].grant = false;
```

```
    }

    // Atualiza os sinais que indicam o dono do barramento.
    m_address_owner = master;
    m_request_arb[master].grant = true;

    return true;
}
...
```

Algoritmo 9 – Fragmento de código de um árbitro AHB

O árbitro AHB deve decidir quem será o dono do barramento baseado nas prioridades dos mestres. A implementação do Algoritmo 9 percorre a lista de mestres verificando quais estão requisitando o barramento. Quando mais de um mestre está requisitando o controle do barramento, o de maior prioridade ganha acesso e o de menor prioridade espera até o próximo ciclo para ter a chance de concorrer novamente.

4.9 Árbitro Avalon

A classe *avalon_default_arbiter* é uma implementação de um árbitro para o barramento Avalon. *avalon_default_arbiter* implementa todos os métodos definidos pela interface *avalon_arbiter_if*. O diagrama da classe *avalon_default_arbiter* é mostrado na Figura 39.

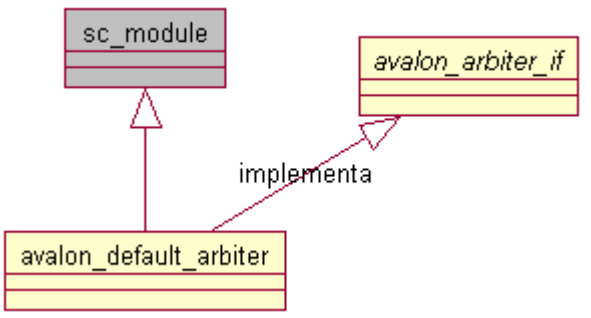

Figura 39 – Diagrama do árbitro do barramento Avalon

O fragmento de código no Algoritmo 10 exemplifica como é a implementação de um árbitro Avalon do qual o barramento utiliza os serviços fornecidos pela interface *avalon_arbiter_if* para fazer a arbitragem.

```
...
bool Avalon_default_arbiter::arbitrate(int bus_id)
{
    int slave_alocation[NUM_SLAVES];

    memset(slave_alocation, 0, sizeof(slave_alocation));

    // Percorre todos os mestres para verificar quem está requisitando o barramento.
    for (int k = 0; k < master_count; k++)
    {
            m_request_arb[master_list[k]].grant = false;

            if (m_request_arb[master_list[k]].request == true) // O mestre k está requisitando
            {
                // Define qual o escravo sendo requisitado pelo mestre
                bus_address addr = (bus_address) *(m_request_arb[master_list[k]].m_address);
                int slave_id = m_decoder_if->get_slave(bus_id, addr);

                if (slave_id > 0) // somente para escravos/endereços válidos
                {
                    // esse é o primeiro mestre que requisita esse escravo
                    if (slave_alocation[slave_id] == 0)
                    {
                        slave_alocation[slave_id] = master_list[k]; // Reserva o escravo para esse mestre.
                        m_request_arb[master_list[k]].grant = true; // O mestre recebe acesso temporário ao escravo.
                    }
                    else // Tem mais de um mestre requisitando o mesmo escravo,
                    {    // Então faz a arbitragem pela maior prioridade
                        bus_priority priority1 = m_request_arb[slave_alocation[slave_id]].priority; //  prioridade de A
                        bus_priority priority2 = m_request_arb[master_list[k]].priority; // pega prioridade do mestre B

                        if (priority2 < priority1) // Se a prioridade do segundo mestre é maior, o escravo vai para ele.
                        {
                            // O primeiro mestre perde o escravo.
                            m_request_arb[slave_alocation[slave_id]].grant = false;
                            slave_alocation[slave_id] = master_list[k]; // O segundo mestre ganha o escravo.
                            m_request_arb[master_list[k]].grant = true; // O mestre recebe acesso ao escravo.
                        }
                    }
                }
                else
                {
                    /* Se o mestre está tentando acessar um endereço inválido (não mapeado),
                       o árbitro deve dar acesso ao mestre para ser possível ao barramento
                       retornar erro ao mestre. Esse comportamento previne deadlock quando
                       um mestre tenta acessa um endereço inválido. */
                    m_request_arb[master_list[k]].grant = true;
                }
            }
    }

    return true;
}
...
```

Algoritmo 10 – Fragmento de código de um árbitro Avalon

O árbitro Avalon faz arbitragem pelo lado do escravo, ou seja, somente é necessário fazer a arbitragem por prioridade quando mais de um mestre requisita o mesmo escravo.

O código do Algoritmo 10 percorre a lista de mestres para preencher a lista de alocação de escravos. Quando um mestre tenta alocar um escravo que já está alocado, o árbitro decide quem será o dono verificando quem é o de maior prioridade.

4.10 Interface do monitor

O monitor do barramento é um elemento opcional que pode ou não existir e estar conectado ao barramento em uma simulação. O monitor é um elemento passivo no sistema, não interfere no funcionamento e apenas monitora os sinais do barramento. Implementações do monitor geralmente são usadas para se gerar *logs* dos sinais durante uma simulação para posterior análise.

A classe *bus_monitor_if* contem os elementos de interface para o monitor do barramento. Essa classe é uma especialização da classe *sc_interface* definida pelo SystemC. Todos os métodos são abstratos e devem ser implementados pelo barramento. É através dessa interface que o monitor conecta-se ao barramento. A Figura 40 mostra o diagrama da classe *bus_monitor_if*.

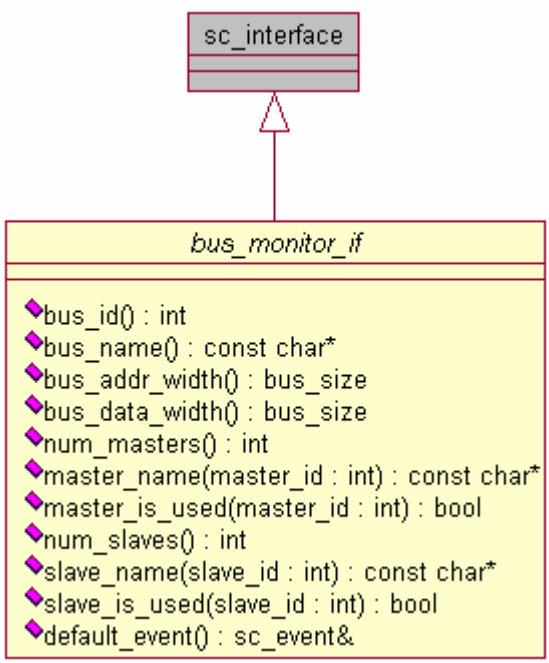

Figura 40 – Diagrama da classe bus_monitor_if

Descrição dos métodos:

- **int bus_id():** Retorna o ID do barramento onde o monitor está conectado.
- **const char* bus_name():** Retorna o nome do barramento onde o monitor está conectado.
- **bus_size bus_addr_width():** Retorna o tamanho do barramento de endereços.
- **bus_size bus_data_width():** Retorna o tamanho do barramento de dados.

- **int num_masters():** Retorna o número de mestres que estão conectados ao barramento.
- **const char* master_name(int master_id):** Retorna o nome do mestre referente ao identificador *master_id*.
- **bool master_is_used(int master_id):** Retorna verdadeiro se o mestre referente ao identificador *master_id* está conectado ao barramento.
- **int num_slaves():** Retorna o número de escravos que estão conectados ao barramento.
- **const char* slave_name(int slave_id):** Retorna o nome do escravo referente ao identificador *slave_id*.
- **bool slave_is_used(int slave_id):** Retorna verdadeiro se o escravo referente ao identificador *slave_id* está conectado ao barramento.
- **const sc_event& default_event():** Os módulos monitores devem ser sensíveis ao evento retornado por esse método. Os eventos são disparados pelo barramento quando os sinais estão estáveis e podem ser capturados pelo monitor.

Os métodos definidos pela interface *bus_monitor_if* podem ser descritos como sendo serviços de verificação do estado dos sinais do barramento em um dado ciclo, implementados pelo próprio barramento, que estão disponíveis para os monitores através da porta (*sc_port* do SystemC) de conexão do monitor.

4.10.1 Interface do monitor AHB

A classe *ahb_monitor_if* é a classe que contem os elementos específicos de interface para um monitor do barramento AHB. Essa classe é uma especialização da classe *bus_monitor_if*. Todos os métodos são abstratos e devem ser implementados pelo barramento AHB. É através dessa interface que o monitor conecta-se ao barramento AHB. A Figura 41 mostra o diagrama da classe *ahb_monitor_if*.

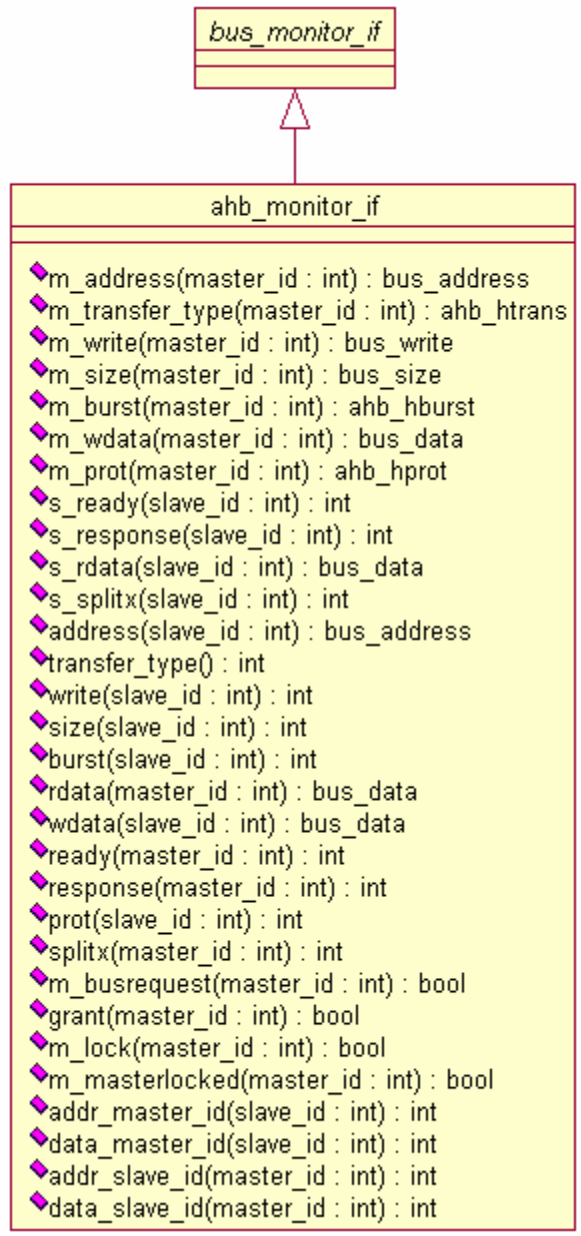

Figura 41 – Diagrama da classe ahb_monitor_if

Descrição dos métodos:

- **bus_address m_address(int master_id):** Retorna o endereço (sinal HADDR) sinalizado pelo mestre referente a *master_id*.
- **ahb_htrans m_transfer_type(int master_id):** Retorna o tipo de transferência (sinal HTRANS) sinalizado pelo mestre referente a *master_id*.

- **bus_write m_write(int master_id):** Retorna o tipo da operação (Leitura ou Escrita) (sinal HWRITE) sinalizado pelo mestre referente a *master_id*.
- **bus_size m_size(int master_id):** Retorna o tamanho da transferência (sinal HSIZE) sinalizado pelo mestre referente a *master_id*.
- **ahb_hburst m_burst(int master_id):** Retorna o modo da transferência (sinal HBURST) sinalizado pelo mestre referente a *master_id*.
- **const bus_data m_wdata(int master_id):** Retorna o dado a ser transferido (sinal HWDATA) em uma operação de escrita sinalizado pelo mestre referente a *master_id*.
- **ahb_hprot m_prot(int master_id):** Retorna o modo de proteção (sinal HPROT) sinalizado pelo mestre referente a *master_id*.
- **int s_ready(int slave_id):** Retorna a resposta (sinal HREADY) sinalizada pelo escravo referente a *slave_id*.
- **int s_response(int slave_id):** Retorna a resposta (sinal HRESP) sinalizada pelo escravo referente a *slave_id*.
- **const bus_data s_rdata(int slave_id):** Retorna o dado (sinal HRDATA) sinalizado numa operação de leitura pelo escravo referente a *slave_id*.
- **int s_splitx(int slave_id):** Retorna a indicação de que um mestre pode prosseguir uma operação *split* (sinal HSPLIT) sinalizada pelo escravo referente a *slave_id.*
- **bus_address address(int slave_id):** Retorna o endereço (sinal HADDR) fornecido ao escravo referente a *slave_id.*
- **int transfer_type(int slave_id):** Retorna o tipo de transferência (sinal HTRANS) fornecido ao escravo referente a *slave_id*.
- **int write(int slave_id):** Retorna a direção da transferência (Escrita ou Leitura) (sinal HWRITE) fornecida ao escravo referente a slave_id.
- **int size(int slave_id):** Retorna o tamanho da transferência (sinal HSIZE) fornecido ao escravo referente a *slave_id.*
- **int burst(int slave_id):** Retorna o modo de transferência (sinal HBURST) fornecido ao escravo referente a *slave_id.*
- **const bus_data rdata(int master_id):** Retorna o dado a ser recebido (sinal HRDATA) pelo mestre *master_id* em uma operação de leitura.

- **const bus_data wdata(int slave_id):** Retorna o dado (sinal HWDATA) fornecido em uma operação de escrita ao escravo referente a *slave_id*.
- **int ready(int master_id):** Retorna a resposta do escravo a ser recebido (sinal HREADY) pelo mestre *master_id*.
- **int response(int master_id):** Retorna a resposta do escravo a ser recebido (sinal HRESP) pelo mestre *master_id*.
- **int prot(int slave_id):** Retorna a informação de proteção (sinal HPROT) fornecida ao escravo referente a *slave_id*.
- **int splitx(int master_id):** Retorna a indicação de que o mestre *master_id* pode continuar ou não a execução de uma operação *split*.
- **bool m_busrequest(int master_id):** Retorna verdadeiro se o mestre *master_id* está requisitando (sinal HBUSREQ) o barramento.
- **bool grant(int master_id):** Retorna verdadeiro se o mestre *master_id* é dono (sinal HGRANT) do barramento.
- **bool m_lock(int master_id):** Retorna verdadeiro se o mestre *master_id* pretende realizar uma operação de transferência indivisível (sinal HLOCK).
- **bool m_masterlocked(int master_id):** Retorna verdadeiro se o mestre *master_id* está realizando transferências indivisíveis (sinal HMASTLOCK).
- **int addr_master_id(int slave_id):** Retorna o ID do mestre que está realizando uma transferência (fase de endereço) relacionada ao escravo *slave_id*.
- **int data_master_id(int slave_id):** Retorna o ID do mestre que está realizando uma transferência (fase de dados) relacionada ao escravo *slave_id*.
- **int addr_slave_id(int master_id):** Retorna o ID do escravo que está envolvido na transferência realizada pelo mestre *master_id* durante a fase de endereço.
- **int data_slave_id(int master_id):** Retorna o ID do escravo que está envolvido na transferência realizada pelo mestre *master_id* durante a fase de dados.

4.10.2 Interface do monitor Avalon

A classe *avalon_monitor_if* é a classe que contem os elementos específicos de interface para um monitor do barramento Avalon. Essa classe é uma especialização da classe

bus_monitor_if. Todos os métodos são abstratos e devem ser implementados pelo barramento Avalon. É através dessa interface que o monitor conecta-se ao barramento Avalon. A Figura 42 mostra o diagrama da classe *avalon_monitor_if*.

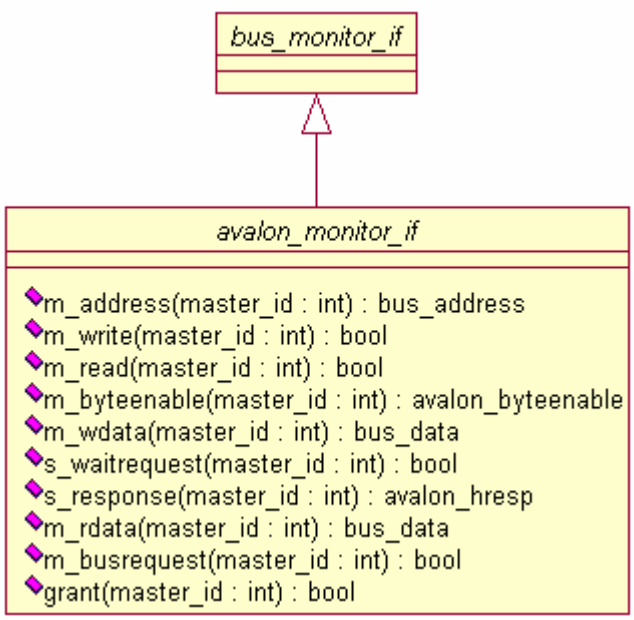

Figura 42 – Diagrama da classe avalon_monitor_if

Descrição dos métodos:

- **bus_address m_address(int master_id):** Retorna o endereço (sinal *address*) sinalizado pelo mestre referente a *master_id*.
- **bool m_write(int master_id):** Retorna verdadeiro (sinal *write*) se o mestre referente a *master_id* está executando uma operação de escrita.
- **bool m_read(int master_id):** Retorna verdadeiro (sinal *read*) se o mestre referente a *master_id* está executando uma operação de leitura.
- **avalon_byteenable m_byteenable(int master_id):** Retorna a indicação de quais bytes da palavra (sinal *byteenable*) são relevantes na transação.
- **const bus_data m_wdata(int master_id):** Retorna o dado a ser transferido (sinal *writedata*) em uma operação de escrita sinalizado pelo mestre referente a *master_id*.
- **bool s_waitrequest(int master_id):** Retorna verdadeiro se mestre deve esperar (sinal *waitrequest*) para completar uma transação.

- **avalon_hresp s_response(int master_id):** Retorna a resposta (sinais *readyfordata*, *dataavailable* e *endofpackage*) sinalizada pelo escravo referente a *slave_id*.
- **const bus_data m_rdata(int master_id):** Retorna o dado a ser recebido (sinal *readdata*) pelo mestre *master_id* em uma operação de leitura.
- **const bool m_busrequest(int master_id):** Retorna verdadeiro se o mestre *master_id* está executando uma transação (o retorno desse método é o resultado de uma operação *ou* entre os sinais *read* e *write*). Esse sinal é um sinal de controle interno entre o barramento Avalon e o árbitro.
- **bool grant(int master_id):** Retorna verdadeiro se o mestre pode executar uma transação que foi iniciada anteriormente. Esse sinal é um sinal de controle interno entre o barramento Avalon e o árbitro.

4.11 Diagrama do barramento

Para a implementação da lógica de funcionamento básico de um barramento foi definida a classe abstrata *bus* que contem os elementos genéricos de funcionamento de um barramento simulado. Essa classe é uma especialização da classe *sc_module* do SystemC e implementa as interfaces *bus_if* e *bus_monitor_if*. Todas as implementações de barramentos devem herdar dessa classe. O diagrama da classe *bus* é mostrado na Figura 43.

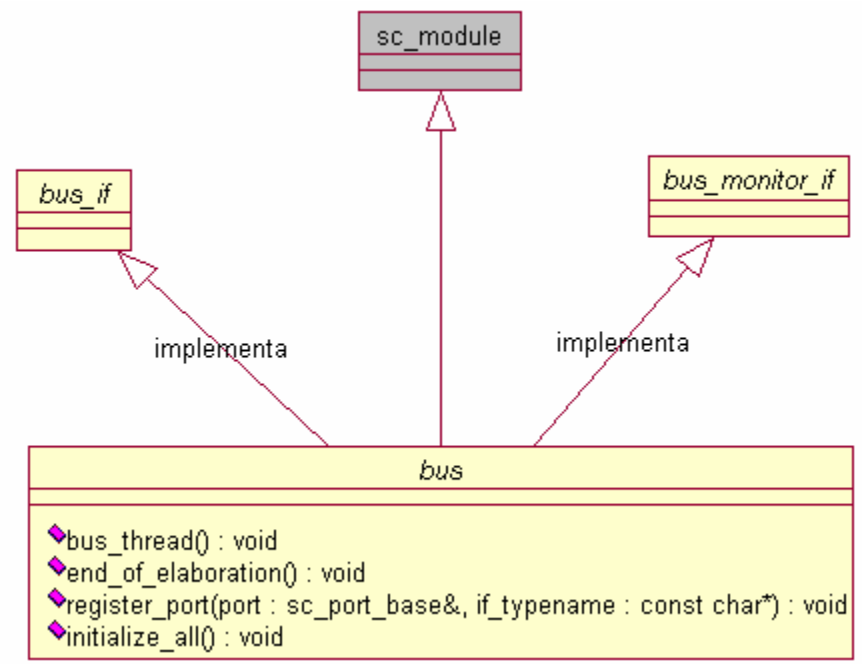

Figura 43 – Diagrama da classe bus

Descrição dos métodos:

- **void bus_thread():** Esse é um método abstrato e deve ser implementado pelas classes filhas da classe *bus*. *bus_thread* é uma *thread* do SystemC sensível ao sinal *clock*. A lógica de funcionamento do barramento é definida nesse método.
- **void end_of_elaboration():** Esse método é automaticamente chamado pelo SystemC instantes antes de se iniciar a simulação, ao final da fase de elaboração.
- **void register_port(sc_port_base& port, const char* if_typename):** Esse método é automaticamente chamado pelo SystemC quando uma porta mestre (*sc_port* que implementa *bus_if*) é conectada ao barramento.
- **void initialize_all():** Método utilizado para se fazer todas as inicializações necessárias antes de iniciar a simulação.

4.11.1 Diagrama do barramento AHB

Para o barramento AHB foi definida a classe *ahb_bus* que implementa a lógica de funcionamento de um barramento AHB simulado. Essa classe é uma especialização da classe *bus*

descrita anteriormente e implementa as interfaces *ahb_bus_if* e *abh_monitor_if*. O diagrama da classe *ahb_bus* é mostrado na Figura 44.

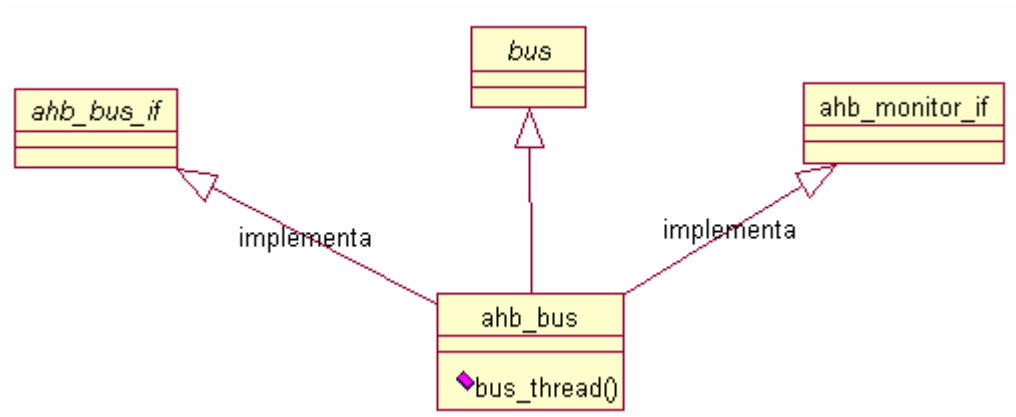

Figura 44 – Diagrama da classe ahb_bus

Descrição dos métodos:

- **void bus_thread():** Esse método implementa a lógica de funcionamento do barramento AHB.

A descrição dos métodos definidos pelas interfaces *ahb_bus_if* e *ahb_monitor_if* que são implementados pela classe *ahb_bus* estão descritos nas respectivas seções.

4.11.2 Diagrama do barramento Avalon

Para o barramento Avalon foi definida a classe *avalon_bus* que implementa a lógica de funcionamento de um barramento Avalon simulado. Essa classe é uma especialização da classe *bus* e implementa as interfaces *avalon_bus_if* e *abh_monitor_if*. O diagrama da classe *avalon_bus* é mostrado na Figura 45.

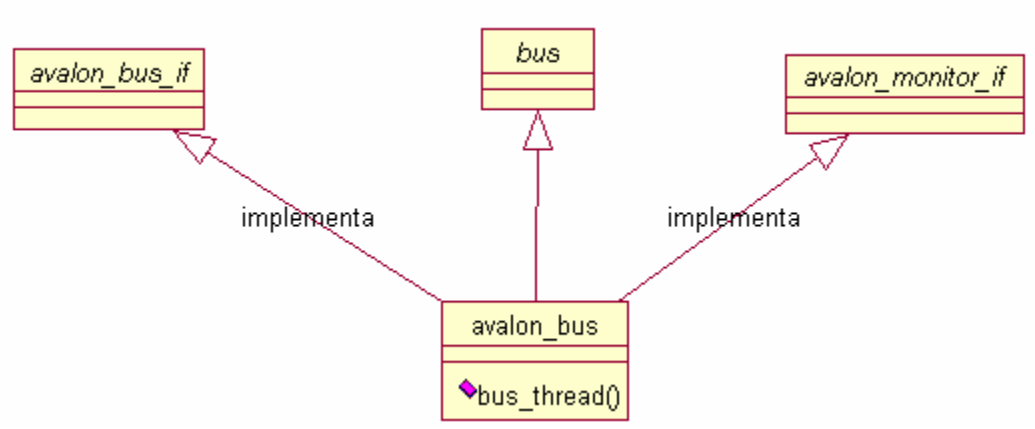

Figura 45 – Diagrama da classe avalon_bus

Descrição dos métodos:

- **void bus_thread():** Esse método implementa a lógica de funcionamento do barramento Avalon.

A descrição dos métodos definidos pelas interfaces *avalon_bus_if* e *avalon_monitor_if* que são implementados pela classe *avalon_bus* estão descritos nas respectivas seções.

4.12 Diagrama geral do barramento AHB

A Figura 46 mostra o diagrama completo das classes desenvolvidas para contemplar o funcionamento de um barramento AHB simulado. As classes AHB_MASTER e AHB_SLAVE indicam os pontos onde novas implementações de mestres e escravos devem ser inseridas.

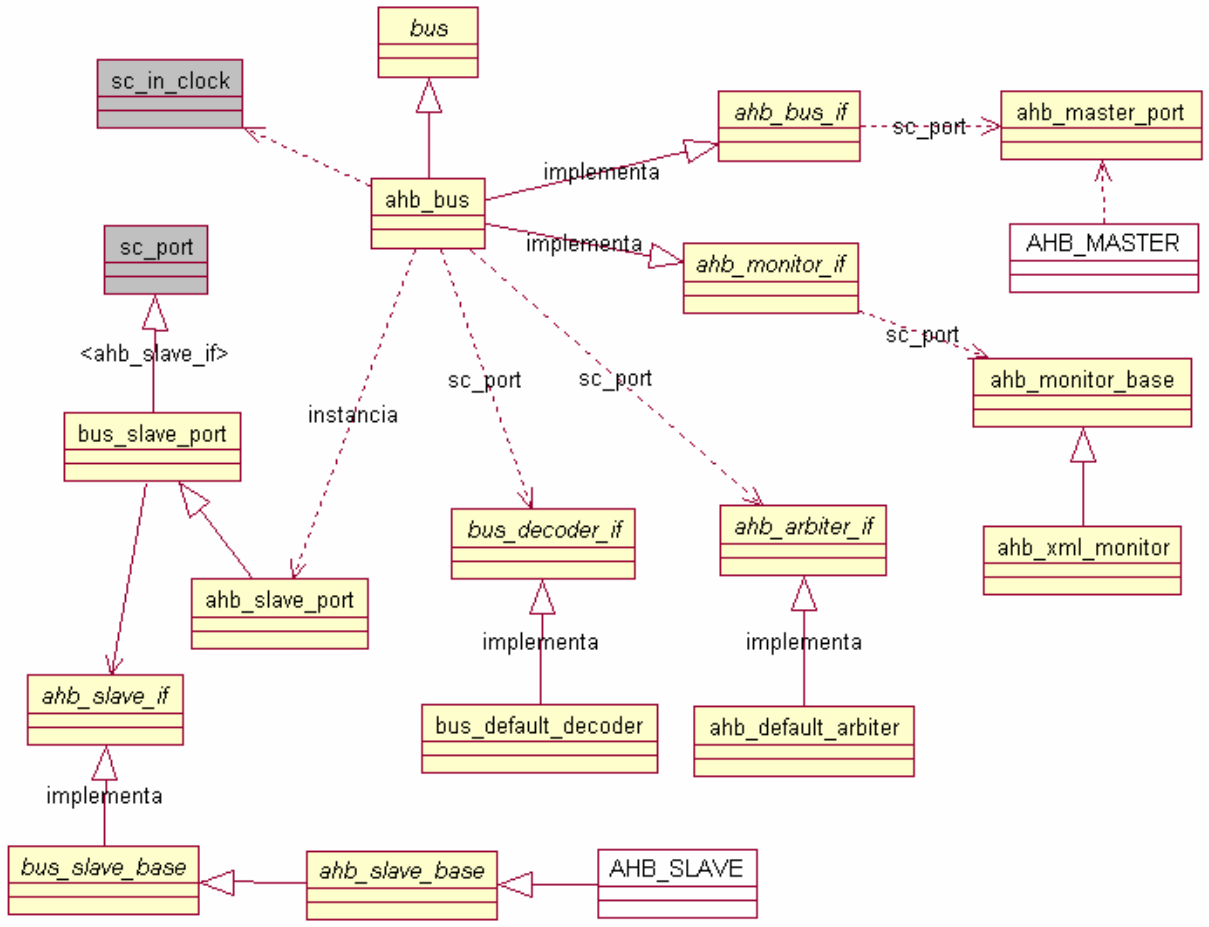

Figura 46 – Diagrama de classes do barramento AHB

4.13 Diagrama geral do barramento Avalon

A Figura 47 mostra o diagrama completo das classes desenvolvidas para contemplar o funcionamento de um barramento Avalon simulado. As classes AVALON_MASTER e AVALON_SLAVE indicam os pontos onde novas implementações de mestres e escravos devem ser inseridas.

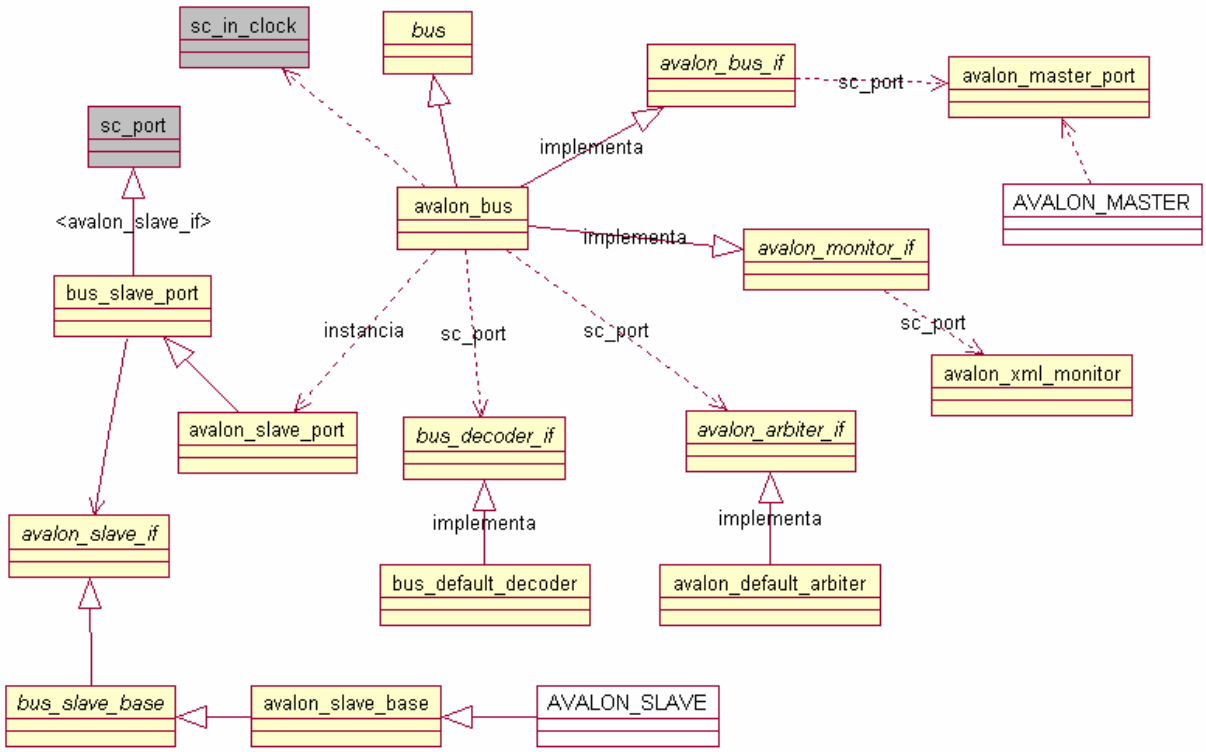

Figura 47 – Diagrama de classes do barramento Avalon

4.14 Integração Bus x ArchC

Para a integração dos barramentos desenvolvidos e o ArchC é necessário a implementação de classes de adaptação que, de um lado consigam se conectar aos modelos de simulação do ArchC e, por outro lado, consigam se conectar ao barramento desejado.

Para a conexão com os modelos do ArchC é necessário que as classes adaptadoras implementem a interface *ac_inout_if* definida pelo ArchC e para a integração com o barramento é necessário que a classes adaptadoras implementem as características de um mestre de barramento. A Figura 48 mostra graficamente como é a integração dos barramentos com o ArchC.

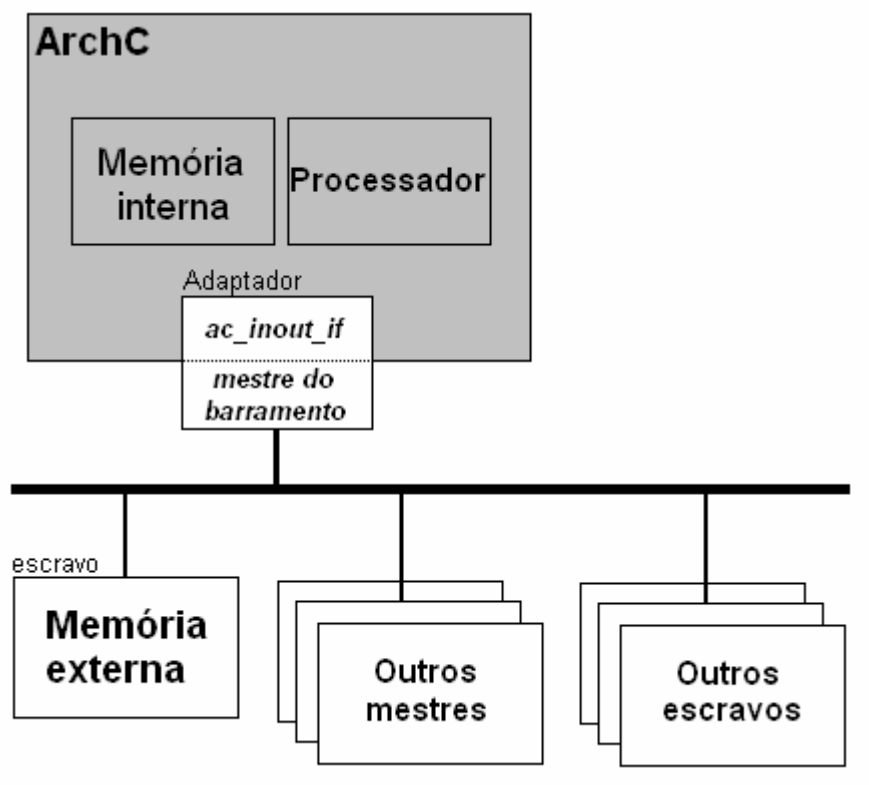

Figura 48 – Integração do barramento com ArchC

Com relação ao endereçamento, são definidas duas faixas de endereços. A primeira faixa, reservada para a memória interna, inicia no endereço 00000000h e vai até *tamanho_da_memória_interna* – 1. A segunda faixa, onde estão todos os outros periféricos conectados ao barramento, inicia no endereço *tamanho_da_memória_interna* e vai até FFFFFFFFh (para o caso de 32 bits de endereçamento).

4.14.1 Implementação das classes adaptadoras

Para a integração do barramento AHB com o ArchC foi definida a classe adaptadora *ac_inout_amba_master_wrapper* que implementa os métodos da interface *ac_inout_if* e atua como mestre do barramento AHB. *ac_inout_amba_master_wrapper* possui uma porta do tipo *ahb_master_port* que se conecta ao barramento AHB.

Para a integração do barramento Avalon com o ArchC foi definida a classe adaptadora *ac_inout_avalon_master_wrapper* que implementa os métodos da interface *ac_inout_if* e atua

como mestre do barramento Avalon. *ac_inout_avalon_master_wrapper* possui uma porta do tipo *avalon_master_port* que se conecta ao barramento Avalon.

A classe base *ac_inout_bus_master_wrapper* foi criada para reunir os elementos comuns de todas as classes adaptadoras. A Figura 49 mostra o diagrama das classes adaptadoras.

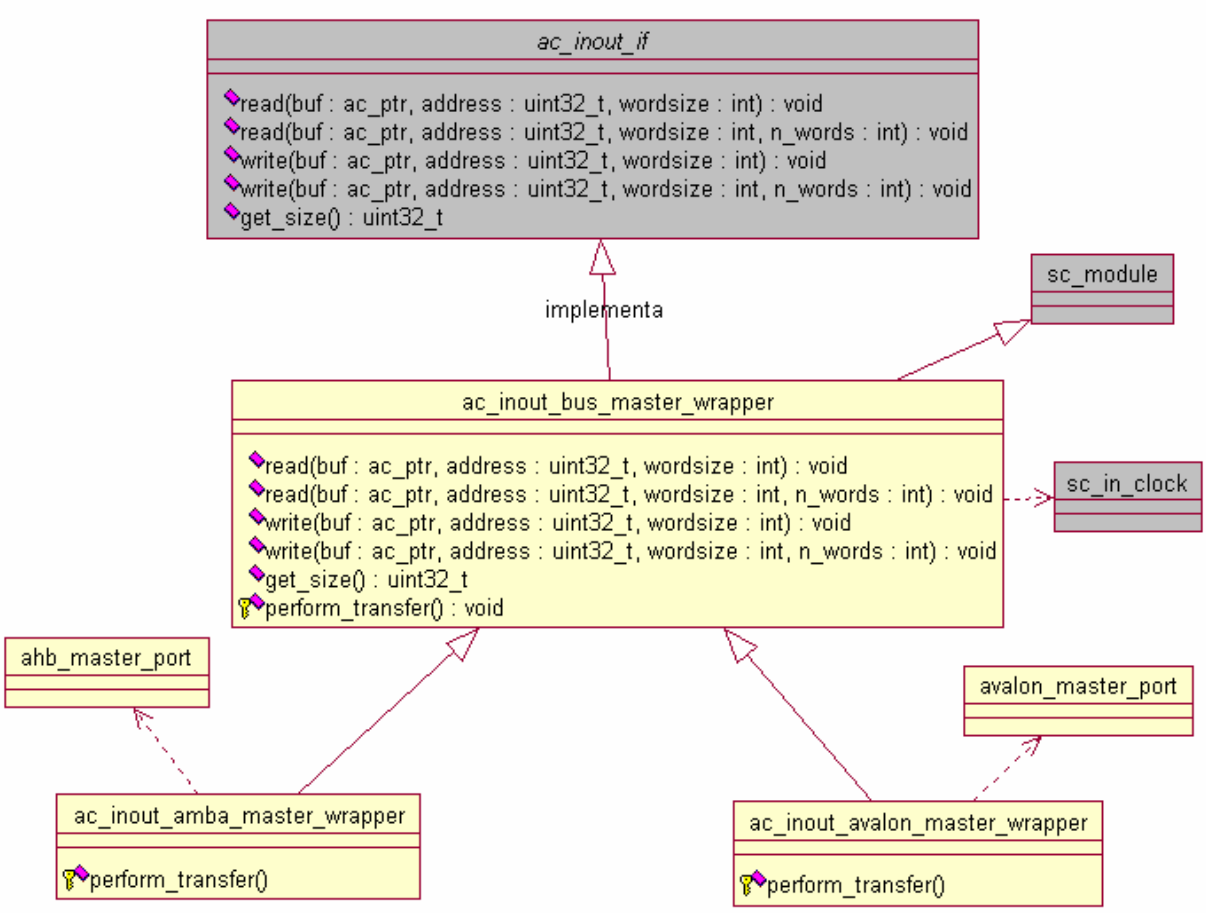

Figura 49 – Diagrama de classes para integração Avalon/AMBA com ArchC

Descrição dos métodos:

- **void read(ac_ptr buf, uint32_t address, int wordsize):** Definido pela interface *ac_inout_if* do ArchC. Usado para ler uma palavra.
- **void read(ac_ptr buf, uint32_t address, int wordsize, int n_words):** Definido pela interface *ac_inout_if* do ArchC. Usado para ler múltiplas palavras.
- **void write(ac_ptr buf, uint32_t address, int wordsize):** Definido pela interface *ac_inout_if* do ArchC. Usado para escrever uma palavra.

- **void write(ac_ptr buf, uint32_t address, int wordsize, int n_words):** Definido pela interface *ac_inout_if*. Usado para escrever múltiplas palavras.
- **uint32_t get_size():** Definido pela interface *ac_inout_if* do ArchC.
- **void perform_transfer():** Esse método é uma *thread* do SystemC sensível ao *clock*. É nesse método que a lógica do mestre do barramento deve ser implementada.

O fragmento de código no Algoritmo 11 exemplifica como seria, em termos de implementação, a definição da arquitetura de simulação mostrada na Figura 48. Esse fragmento de código usa um barramento Avalon, um processador MIPS e para conectar esse dois elementos é usado um elemento adaptador.

```
...

/* Instanciação do clock que será usado
   pelo barramento e periféricos */
sc_clock clk("clk", 20, 0.5, true);

/* Instanciação do decodificador de endereços */
Bus_default_decoder Decoder("Decoder");

/* Instância do árbitro Avalon */
Avalon_default_arbiter Arbiter("Arbiter");

/* Instância do barramento Avalon */
Avalon_Bus Bus("Bus_AVALON", 1);

/* Instância da memória externa (escravo #1)
   Endereço de início: 500000h
   Tamanho: 100 KB */
Avalon_memory_device ExternalMemory("ExternalMemory", 1, 5*1024*1024, 1024 * 100);

/* Instância de um display (escravo #2)
   Endereço de início: FFFFFF00h
   Tamanho: 1 KB */
Avalon_display_device Display1("Display1", 2, 0xFFFFFF00);

/* Instância do adaptador, que será usado para conectar o
   processador ao barramento (Mestre #1) */
ac_inout_avalon_master_wrapper ac_bus_wrapper("Processor1_MasterPort_AVALON", 1);

/* Agora é necessário conectar os elementos entre sí */
Bus.hclk(clk); // Conexão do barramento ao clock.
ac_bus_wrapper.hclk(clk); // Conexão do adaptador ao clock.
Bus.arbiter_port(Arbiter); // Conexão do árbitro ao barramento.
Bus.decoder_port(Decoder); // Conexão do decodificador ao barramento.
Bus.slave_port(ExternalMemory); // Conexão da memória ao barramento.
Bus.slave_port(Display1); // Conexão do display ao barramento.
ac_bus_wrapper.m_port(Bus); // Conexão do barramento à porta do adaptador

/* Instância do processador MIPS.
   Aqui também é passada a referência do adaptador
   que será usada pelo MIPS para acessar o barramento.
*/
mips1 mips1_proc1("mips1", ac_bus_wrapper);
mips1_proc1.init(ac1, av1);
```

```
/* Inicio da simulação */
sc_start(-1);
...
```

Algoritmo 11 – Fragmento de código para integração do barramento com modelos do ArchC

Primeiramente são instanciados os elementos que farão parte do sistema de simulação, como por exemplo, *clock*, *decoder*, árbitro, etc. Depois os elementos são conectados entre si para formar a arquitetura de simulação e, finalmente, a simulação é iniciada.

5 Experimentos

Nessa seção são descritos os experimentos que foram feitos para validar as implementações dos barramentos AMBA-AHB e Avalon.

O principal objetivo dos experimentos é verificar a corretude das implementações dos barramentos. No primeiro experimento executa-se um programa de *benchmark* do ArchC e os resultados com e sem os barramentos são comparados. No segundo experimento exercita-se os algoritmos de arbitragem de ambos os barramentos implementados.

5.1 Primeiro experimento

Um dos objetivos desse experimento é verificar a corretude das implementações dos barramentos AHB e Avalon. Outro objetivo é medir as diferenças de desempenho de um processador do ArchC executando no modo *standalone* e executando conectado aos dispositivos escravos através de um barramento.

A arquitetura de simulação montada para esse experimento é como vista na Figura 50. A arquitetura é formada por um processador MIPS e uma memória externa conectados ao barramento.

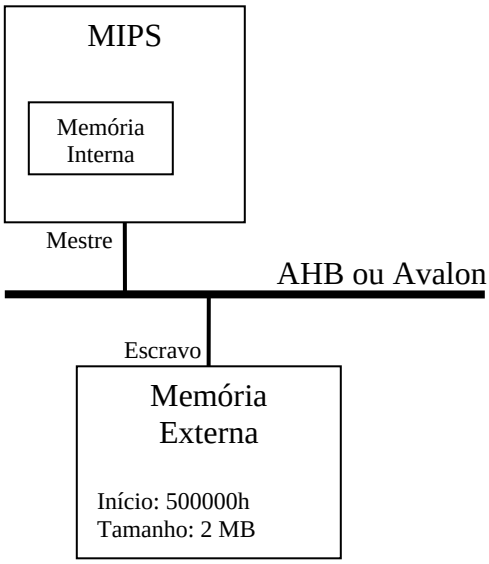

Figura 50 – Arquitetura do experimento

O código C do Algoritmo 12 mostra o programa *Quick Sort* (*Benchmark* do ArchC [21]) que foi usado para a execução do experimento.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define UNLIMIT
#define MAXARRAY 10000

struct myStringStruct
{
        char qstring[128];
};

int compare(const void *elem1, const void *elem2)
{
        return strcmp((*((struct myStringStruct *)elem1)).qstring,
                                (*((struct myStringStruct *)elem2)).qstring);
}

int main(int argc, char *argv[])
{
        struct myStringStruct array[MAXARRAY];
        FILE *fp;
        int i,count = 0;

        if (argc < 2)
        {
                fprintf(stderr,"Usage: qsort_small <file>\n");
                exit(-1);
        }
        else
        {
                fp = fopen(argv[1], "r");

                while((fscanf(fp, "%s", &array[count].qstring) == 1) && (count < MAXARRAY))
                {
                        count++;
                }
        }

        printf("\nSorting %d elements.\n\n",count);

        qsort(array,count,sizeof(struct myStringStruct),compare);

        for(i = 0; i < count; i++)
        {
                printf("%s\n", array[i].qstring);
        }

        return 0;
}
```

Algoritmo 12 – Código Quick Sort benchmark do ArchC para execução do experimento

Esse experimento consistiu em executar o binário gerado a partir do código do Algoritmo 12 para o processador MIPS. Para a primeira execução a alocação do vetor *array* é feita na

memória interna do ArchC, ou seja, a memória externa e o barramento não fazem parte da simulação.

Para a segunda e terceira execução é necessário uma pequena mudança no algoritmo para que o vetor *array* esteja alocado na memória externa. Essa mudança é mostrada no Algoritmo 13. Para a segunda execução foi usado o barramento AHB e para a terceira execução foi usado o barramento Avalon.

```
/* array aponta para o início do endereço da memória externa */
struct myStringStruct* array = (struct myStringStruct *) 0x500000;
```

Algoritmo 13 – Código Quick Sort benchmark do ArchC modificado

A corretude das implementações foi verificada comparando-se os valores de saída (lista de *strings* ordenada) das três execuções. As implementações de ambos os barramentos podem ser consideradas corretas porque os valores de saída das respectivas simulações foram idênticos aos valores de saída da primeira execução.

Esse experimento também foi usado para medir as diferenças de desempenho de simulação entre os três cenários descritos. Essas diferenças podem ser vista na Figura 51. Os valores são dados em *K Instruções por segundo* e foram fornecidos pelo ArchC no final de cada simulação.

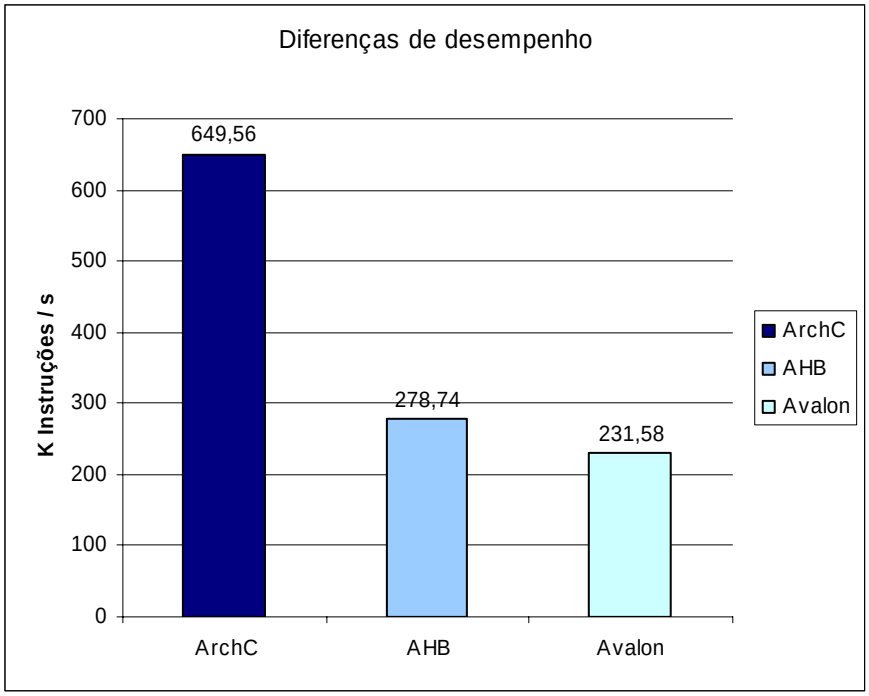

Figura 51 – Gráfico de desempenho

No primeiro cenário, onde o barramento e a memória externa não estão envolvidos (*standalone*), atingiu-se 649,56 K instruções por segundo. Esse valor é mais de duas vezes maior que os dos outros cenários e isso se deve ao fato de os barramentos serem executados com precisão de ciclo e no primeiro cenário todos os acessos são chamadas diretas de função.

No gráfico também se pode observar a diferença de desempenho entre os barramentos. Para o AHB atingiu-se 278,74 K instruções por segundo enquanto que para o Avalon atingiu-se apenas 231,58 K instruções por segundo.

5.2 Segundo experimento

O objetivo desse experimento é verificar o comportamento dos algoritmos de arbitragem para os barramentos AHB e Avalon, descritos no Algoritmo 9 e Algoritmo 10 respectivamente.

Primeiramente é descrito a arquitetura de simulação e, em seguida, são descritas as seis etapas do experimento. A primeira etapa contem quinze escravos, a segunda contem dez escravos, a terceira contem cinco escravos, a quarta contem dois escravos, quinta etapa contem

apenas um escravo e a sexta e última etapa contem um escravo mas os algoritmos de arbitragem são trocados de prioridade simples (*highest priority*) por algoritmos *round-robin*. Ao final é feita uma análise dos resultados obtidos.

A arquitetura de simulação montada para esse experimento é como vista na Figura 52. São dez processadores MIPS e quinze *displays* escravos conectados ao barramento.

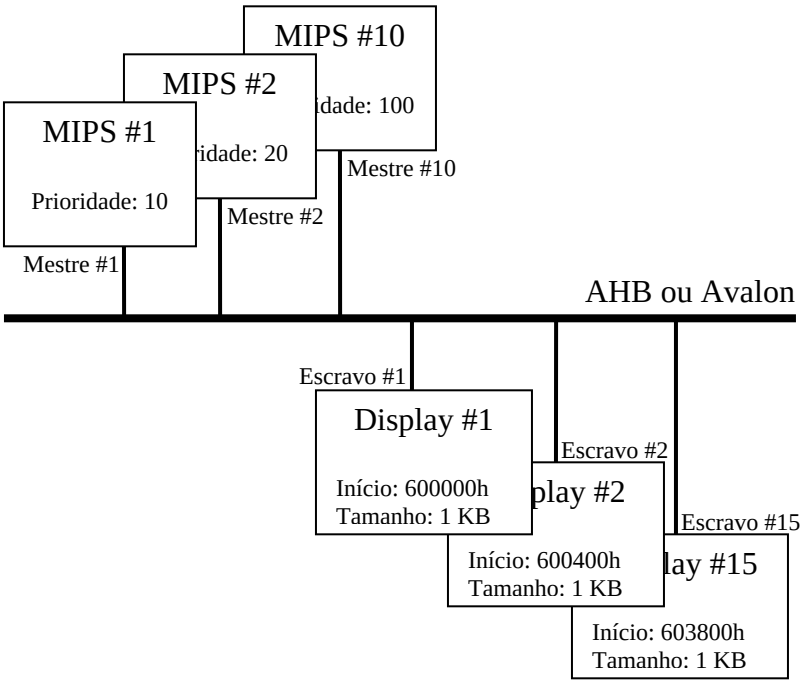

Figura 52 – Arquitetura do experimento

Os dez mestres do barramento possuem prioridades diferentes, sendo que o MIPS #1 é o de maior prioridade. A tabela 7 mostra as prioridades dos mestres do barramento.

Mestre	Prioridade de acesso
MIPS #1	10
MIPS #2	20
MIPS #3	30
MIPS #4	40
MIPS #5	50
MIPS #6	60
MIPS #7	70
MIPS #8	80

MIPS #9	90
MIPS #10	100

Tabela 7 – Prioridades dos mestres

O mapa de endereçamento dos escravos do barramento é mostrado na Tabela 8. Esse experimento, como dito anteriormente, é dividido em etapas e dependendo da etapa, um ou mais escravos podem fazer parte da simulação.

Dispositivo	Endereço inicial	Tamanho
Memória interna do MIPS	00000000h	500000h (5 MB)
Display #1	00600000h	400h (1 KB)
Display #2	00600400h	400h (1 KB)
Display #3	00600800h	400h (1 KB)
Display #4	00600C00h	400h (1 KB)
Display #5	00601000h	400h (1 KB)
Display #6	00601400h	400h (1 KB)
Display #7	00601800h	400h (1 KB)
Display #8	00601C00h	400h (1 KB)
Display #9	00602000h	400h (1 KB)
Display #10	00602400h	400h (1 KB)
Display #11	00602800h	400h (1 KB)
Display #12	00602C00h	400h (1 KB)
Display #13	00603000h	400h (1 KB)
Display #14	00603400h	400h (1 KB)
Display #15	00603800h	400h (1 KB)

Tabela 8 – Mapa de endereços dos escravos

O segmento de código do Algoritmo 14 mostra parcialmente o programa usado para a execução do experimento. Esse programa, depois de compilado, é executado em todos os dez processadores MIPS simultaneamente durante a simulação.

```
...
#define DISPLAY_COUNT  1 /* Número de displays */

int main(int argc, char* argv[])
{
        int display;

        /* Variável para a contagem do loop de execução */
        int loop_count = 0;

        /* Pega o tempo inicial da execução */
        time_t initial_time = time(0);

        while (time(0) - initial_time < 3 * 60) /* Executa por 3 minutos */
```

```
        {
             for (display = 0; display < DISPLAY_COUNT; display++)
             {
                  WriteToDisplay(display); /* Acessa o escravo # */

                  loop_count++; /* Incrementa o contador de loop */
             }
        }

        /* Imprime o resultado */
        printf("Processor: %i, loop_count: %i\n", processor, loop_count);

        return 0;
}
...
```

Algoritmo 14 – Fragmento de código para execução do experimento

A definição *DISPLAY_COUNT* determina quantos escravos serão acessados pelo programa em execução.

O algoritmo basicamente permanece em um laço de execução por 3 minutos e, em cada iteração, acessa todos os displays escravo incrementado o contador *loop_count*. Ao final da simulação, o valor resultante de *loop_count* é impresso.

O valor de *loop_count* indica quantas vezes o processador conseguiu acessar os escravos no intervalo de três minutos.

Cada acesso de um mestre a um escravo é uma operação de escrita de uma palavra de 32 bits (transferência simples).

Primeira etapa:

Para essa simulação foram usados os quinze *displays (#define DISPLAY_COUNT 15)*. Essa etapa mostra o que acontece quando existem mais escravos que mestres no barramento.

Foram executadas cinco simulações para cada barramento e o valor de *loop_count* para cada processador é a média das cinco simulações. Para cada barramento foi gerado um gráfico que mostra a relação entre os processadores e o número de acessos aos escravos. O número de acesso aos escravos varia de acordo com a prioridade de cada mestre (processador) conectado ao barramento.

O gráfico resultante para o barramento AHB pode ser visto na Figura 53. Nesse gráfico, observa-se que a quantidade de acessos começa a cair efetivamente a partir do sexto mestre.

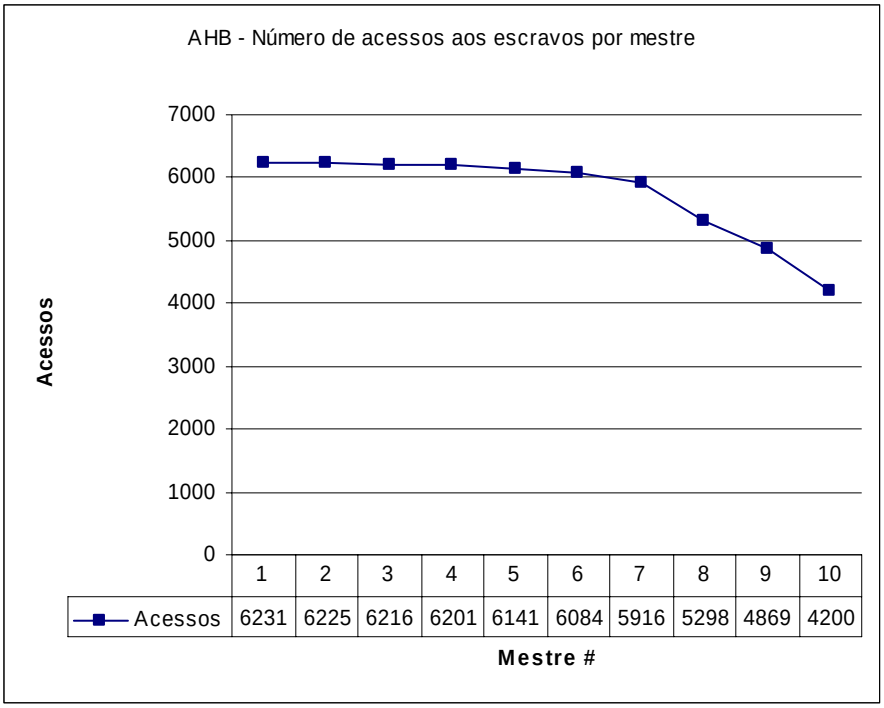

Figura 53 – Resultado da primeira etapa da simulação para o AHB

O gráfico resultante para o barramento Avalon pode ser visto na Figura 54. Nesse gráfico observa-se que a quantidade de acessos entre o mestre de maior prioridade e o de menor prioridade sofre uma pequena variação.

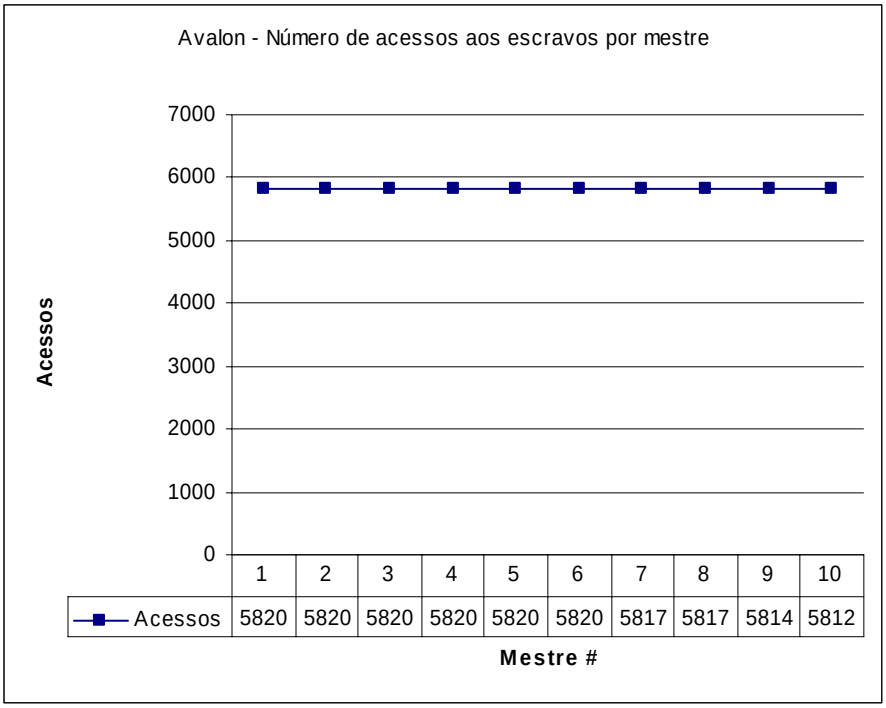

Figura 54 – Resultado da primeira etapa da simulação para o Avalon

Todos os dados obtidos das cinco simulações dessa etapa para cada barramento são mostrados no ANEXO I.

Segunda etapa:

Para essa simulação foram usados dez *displays (#define DISPLAY_COUNT 10)*. Essa etapa mostra o que acontece quando existe a mesma quantidade de escravos e mestres no barramento.

Foram executadas cinco simulações para cada barramento e o valor de *loop_count* para cada processador é a média das cinco simulações. Para cada barramento foi gerado um gráfico que mostra a relação entre os processadores e o número de acessos aos escravos. O número de acesso aos escravos varia de acordo com a prioridade de cada mestre (processador) conectado ao barramento.

O gráfico resultante para o barramento AHB pode ser visto na Figura 55. Nesse gráfico, assim como na etapa anterior para barramento AHB, observa-se que a quantidade de acessos começa a cair efetivamente a partir do sexto mestre.

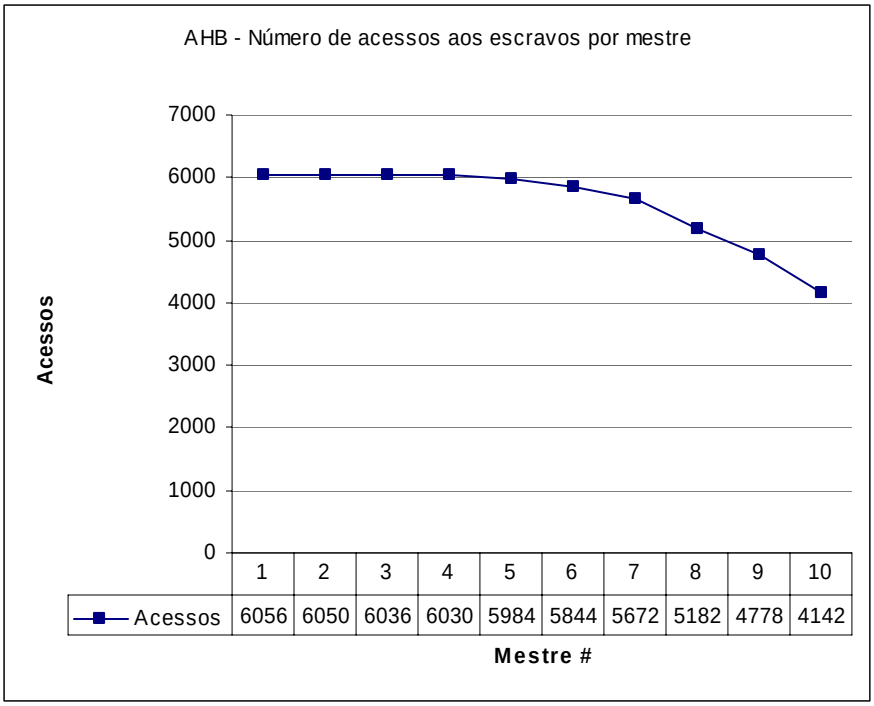

Figura 55 – Resultado da segunda etapa da simulação para o AHB

O gráfico resultante para o barramento Avalon pode ser visto na Figura 56. Nesse gráfico, assim como na etapa anterior para barramento Avalon, observa-se que a quantidade de acessos entre o mestre de maior prioridade e o de menor prioridade sofre uma pequena variação.

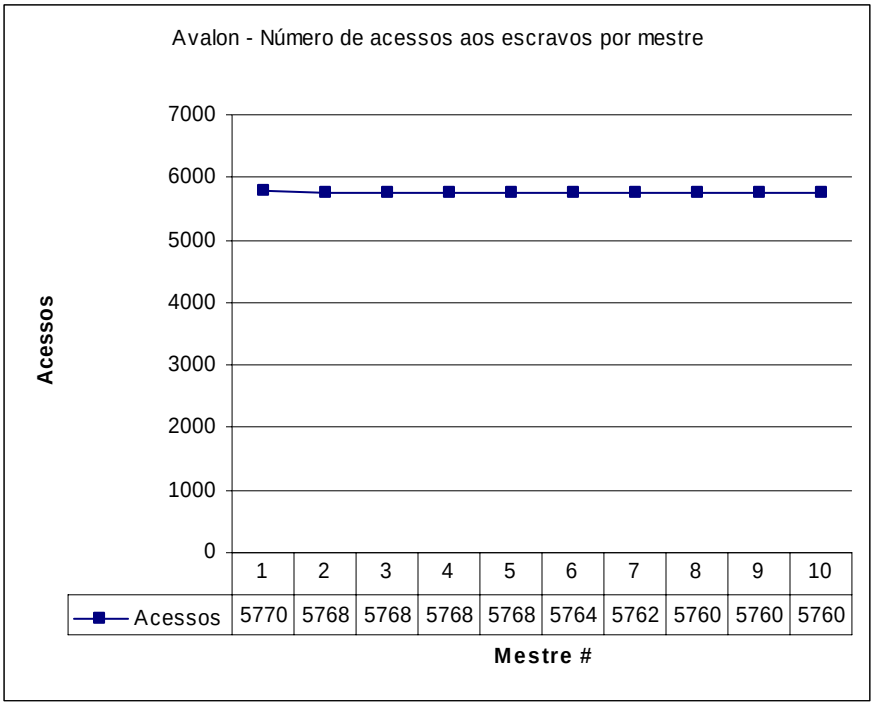

Figura 56 – Resultado da segunda etapa da simulação para o Avalon

Todos os dados obtidos das cinco simulações dessa etapa para cada barramento são mostrados no ANEXO I.

Terceira etapa:

Para essa simulação foram usados cinco *displays (#define DISPLAY_COUNT 5)*. Essa etapa mostra o que acontece quando existem menos escravos que mestres no barramento.

Foram executadas cinco simulações para cada barramento e o valor de *loop_count* para cada processador é a média das cinco simulações. Para cada barramento foi gerado um gráfico que mostra a relação entre os processadores e o número de acessos aos escravos. O número de acesso aos escravos varia de acordo com a prioridade de cada mestre (processador) conectado ao barramento.

O gráfico resultante para o barramento AHB pode ser visto na Figura 57. Nesse gráfico, assim como nas etapas anteriores para barramento AHB, observa-se que a quantidade de acessos começa a cair efetivamente a partir do sexto mestre.

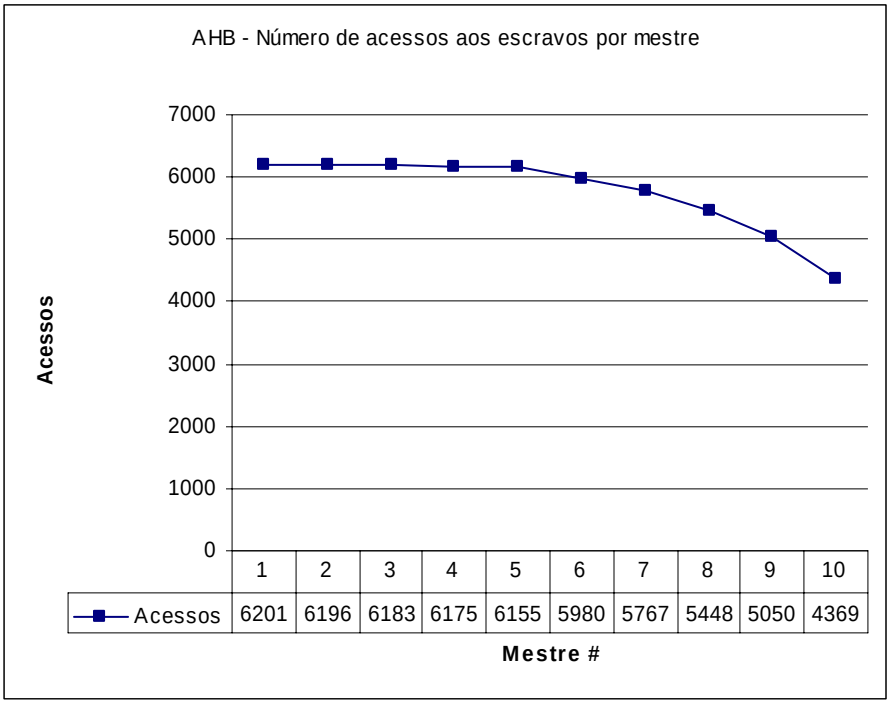

Figura 57 – Resultado da terceira etapa da simulação para o AHB

O gráfico resultante para o barramento Avalon pode ser visto na Figura 58. Nesse gráfico, assim como nas etapas anteriores para barramento Avalon, observa-se que a quantidade de acessos entre o mestre de maior prioridade e o de menor prioridade sofre uma pequena variação.

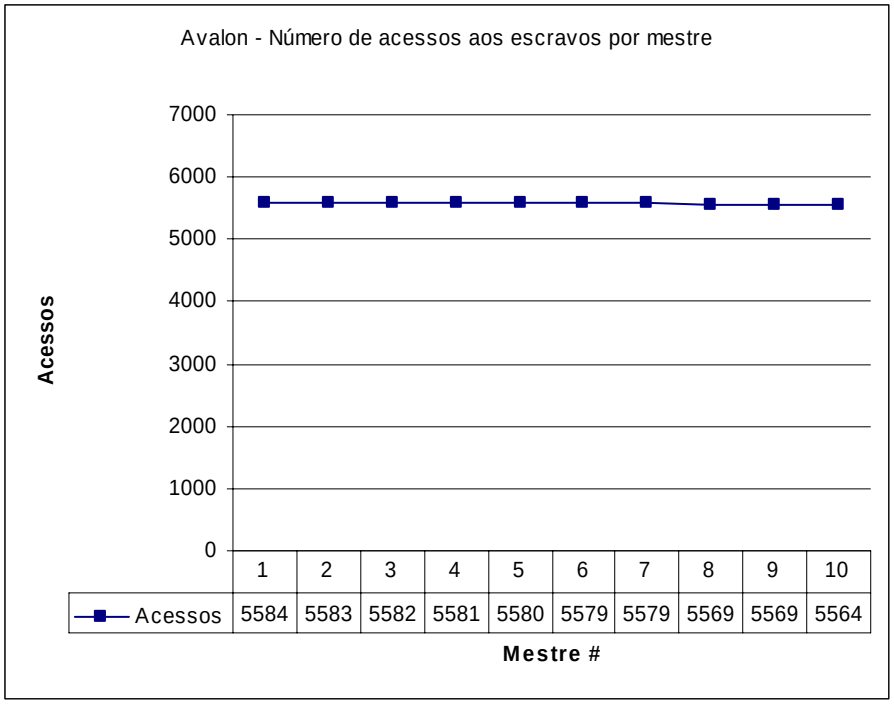

Figura 58 – Resultado da terceira etapa da simulação para o Avalon

Todos os dados obtidos das cinco simulações dessa etapa para cada barramento são mostrados no ANEXO I.

Quarta etapa:

Para essa simulação foram usados dois *displays (#define DISPLAY_COUNT 2)*. Essa etapa mostra o que acontece quando existem dois escravo e muitos mestres no barramento.

Foram executadas cinco simulações para cada barramento e o valor de *loop_count* para cada processador é a média das cinco simulações. Para cada barramento foi gerado um gráfico que mostra a relação entre os processadores e o número de acessos aos escravos. O número de acesso aos escravos varia de acordo com a prioridade de cada mestre (processador) conectado ao barramento.

O gráfico resultante para o barramento AHB pode ser visto na Figura 59. Nesse gráfico, assim como nas etapas anteriores para barramento AHB, observa-se que a quantidade de acessos começa a cair efetivamente a partir do sexto mestre.

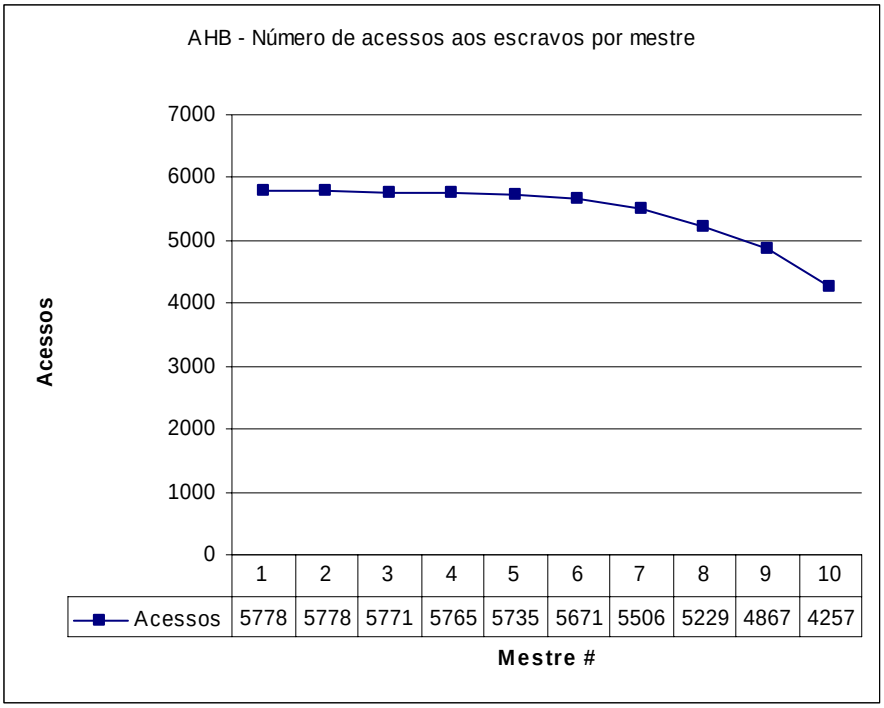

Figura 59 – Resultado da quarta etapa da simulação para o AHB

O gráfico resultante para o barramento Avalon pode ser visto na Figura 60. Nesse gráfico, diferentemente das etapas anteriores para barramento Avalon, observa-se que a quantidade de acessos entre o mestre de maior prioridade e o de menor prioridade sofre uma variação de maior proporção.

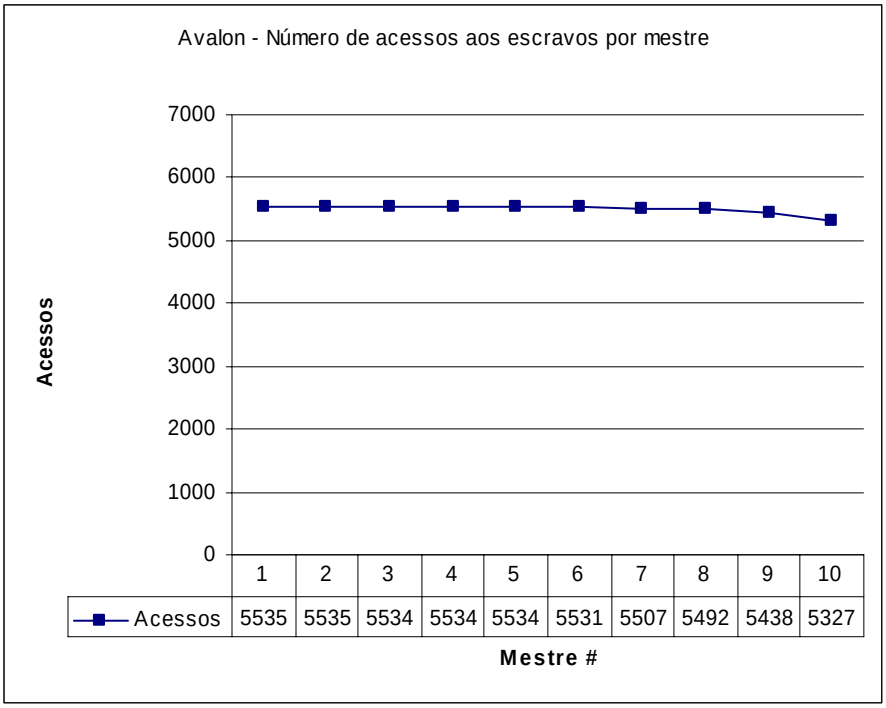

Figura 60 – Resultado da quarta etapa da simulação para o Avalon

Todos os dados obtidos das cinco simulações dessa etapa para cada barramento são mostrados no ANEXO I.

Quinta etapa:

Para essa simulação foi usado apenas um *display (#define DISPLAY_COUNT 1)*. Essa etapa mostra o que acontece quando existe apenas um escravo e muitos mestres no barramento.

Foram executadas cinco simulações para cada barramento e o valor de *loop_count* para cada processador é a média das cinco simulações. Para cada barramento foi gerado um gráfico que mostra a relação entre os processadores e o número de acessos ao escravo. O número de acesso aos escravos varia de acordo com a prioridade de cada mestre (processador) conectado ao barramento.

O gráfico resultante para o barramento AHB pode ser visto na Figura 61. Nesse gráfico observa-se que o comportamento é praticamente o mesmo das etapas anteriores para barramento AHB.

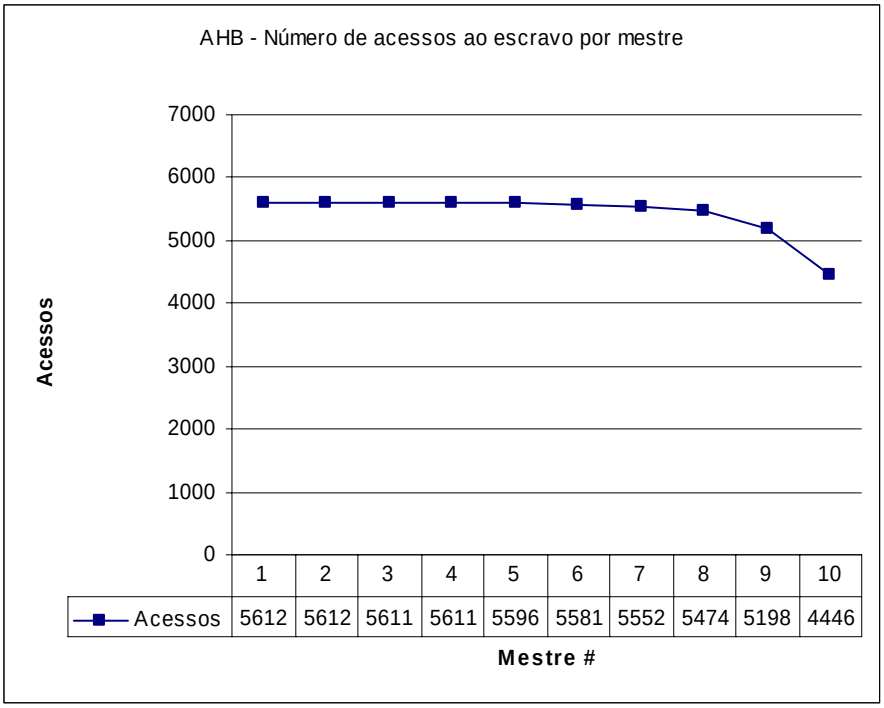

Figura 61 – Resultado da quinta etapa da simulação para o AHB

O gráfico resultante para o barramento Avalon pode ser visto na Figura 62. Nesse gráfico observa-se que a queda no número de acessos é muito grande a partir do quinto mestre.

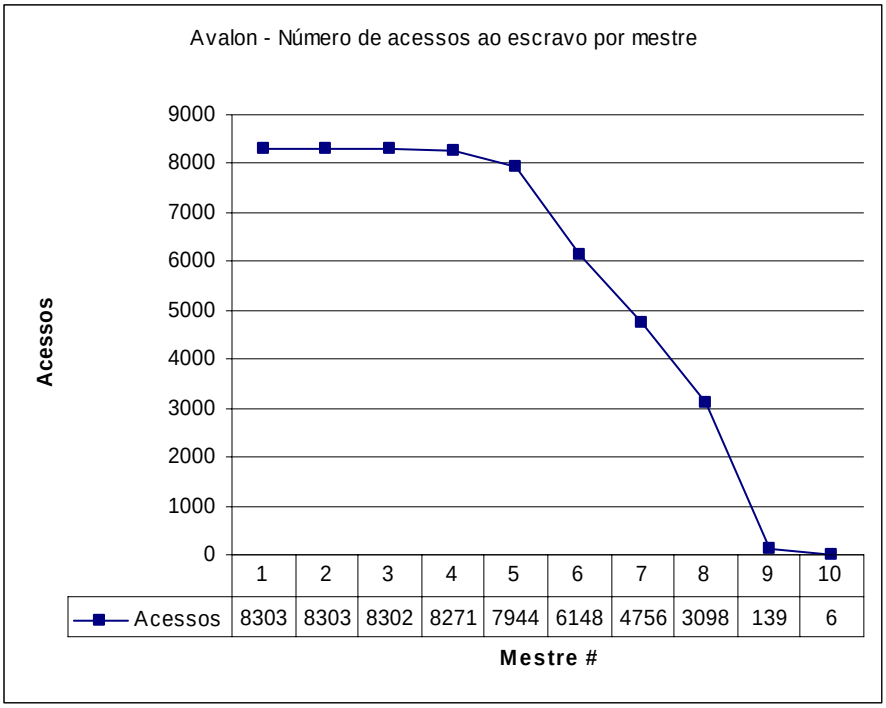

Figura 62 – Resultado da quinta etapa da simulação para o Avalon

Todos os dados obtidos das cinco simulações dessa etapa para cada barramento são mostrados no ANEXO I.

Sexta etapa:

Para essa simulação, assim como na etapa anterior, foi usado apenas um *display (#define DISPLAY_COUNT 1)*, mas os algoritmos de arbitragem foram substituídos por algoritmos *round-robin*.

Foram executadas cinco simulações para cada barramento e o valor de *loop_count* para cada processador é a média das cinco simulações. Para cada barramento foi gerado um gráfico que mostra a relação entre os processadores e o número de acessos ao escravo.

O gráfico resultante para o barramento AHB pode ser visto na Figura 63. Nesse gráfico observa-se que o comportamento é diferente das etapas anteriores, devido ao uso de um árbitro *round-robin*, todos os mestres AHB tiveram praticamente a mesma quantidade de acessos ao escravo.

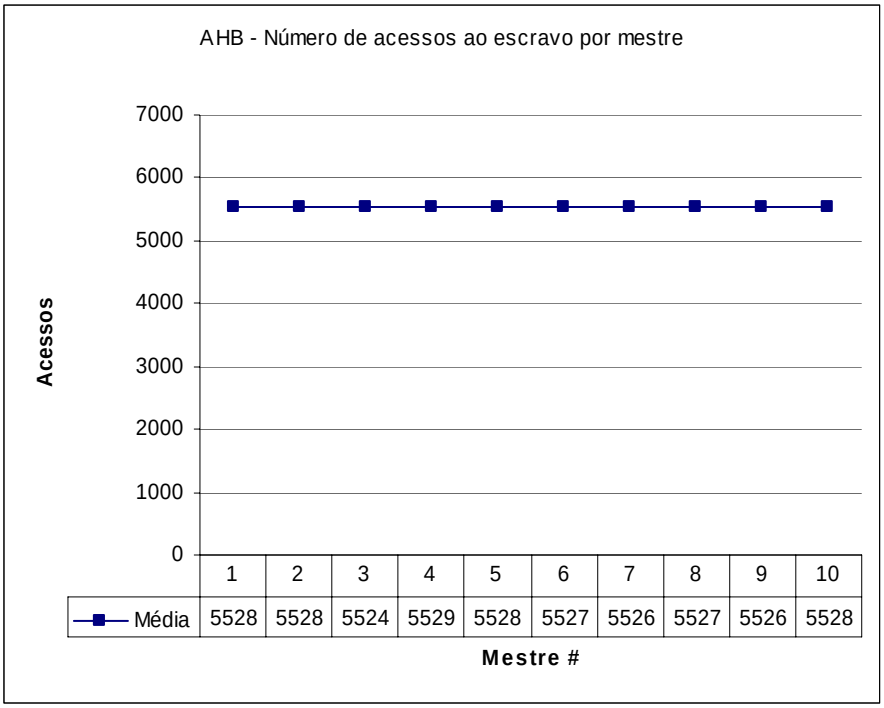

Figura 63 – Resultado da sexta etapa da simulação para o AHB

O gráfico resultante para o barramento Avalon pode ser visto na Figura 64. Nesse gráfico observa-se que o comportamento é diferente da etapa anterior, devido ao uso de um árbitro *round-robin*, todos os mestres Avalon tiveram praticamente a mesma quantidade de acessos ao escravo.

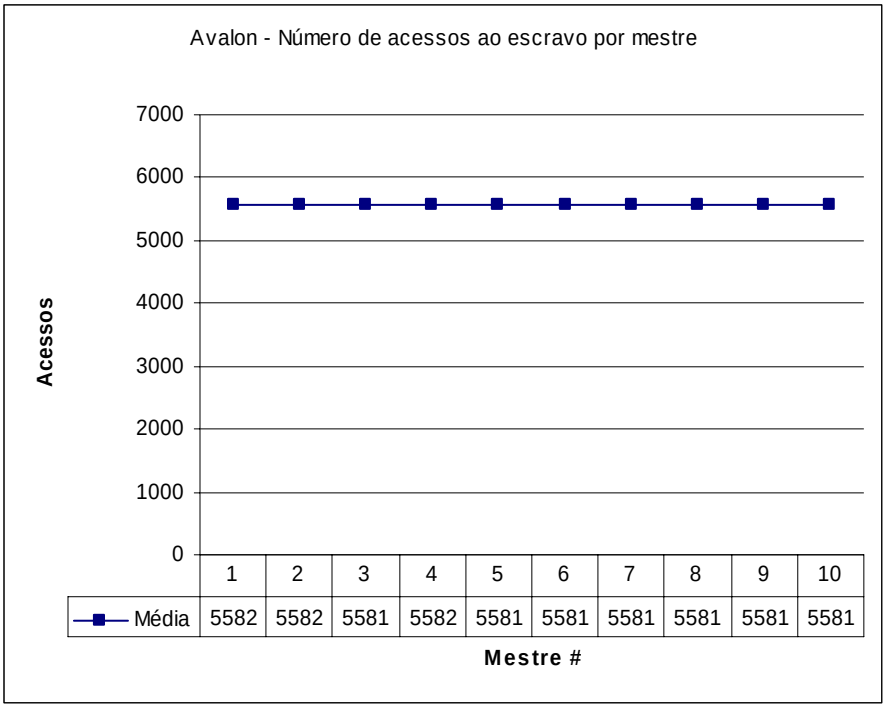

Figura 64 – Resultado da sexta etapa da simulação para o Avalon

Todos os dados obtidos das cinco simulações dessa etapa para cada barramento são mostrados no ANEXO I.

Análise dos resultados:

Para o barramento AHB observar-se que o comportamento foi praticamente o mesmo nas cinco primeiras etapas de simulação. Ou seja, o número de acessos aos escravos executados por cada mestre independe da quantidade de escravos. Isso é explicado pelo fato de que, no barramento AHB, os mestres concorrem pelo barramento e não por cada escravo individualmente. O número de acessos de cada mestre varia apenas em função da prioridade desse mestre.

Para o barramento Avalon pode-se observar que o número de acessos de cada mestre depende da relação entre a quantidade de escravos e mestres (Mestres / Escravos). Isso é explicado pelo fato de que, no barramento Avalon, os mestres concorrem por cada escravo individualmente (arbitragem pelo lado do escravo) e muitas transferências são executadas

simultaneamente em um mesmo ciclo. A prioridade de cada mestre só é relevante quando acontecem muitos conflitos durante as transferências.

O algoritmo de arbitragem usado para o barramento Avalon mostrou-se muito mais ineficiente e injusto que o usado para o barramento AHB, como pode ser visto no gráfico da Figura 62. Os mestres #9 e #10 no barramento Avalon quase não tiveram acesso ao escravo durante a quinta etapa da simulação.

O comportamento na sexta etapa, onde foram usados árbitros *round-robin*, foi praticamente o mesmo para ambos os barramentos. Todos os mestres tiveram praticamente o mesmo número de acessos ao escravo.

6 Conclusão

Inicialmente, para o desenvolvimento desse trabalho, foi feito um estudo aprofundado da linguagem SystemC e sua aplicabilidade na modelagem em TLM.

Depois, quatro barramentos foram detalhadamente estudados para se definir e agrupar os elementos comuns a todos eles. Em seguida, foi desenvolvido um *framework* de barramentos. Esse *framework* é um conjunto de classes e interfaces em C++ que contem os elementos genéricos presentes na maioria dos barramentos existentes. O *framework* criado, através do reuso e especialização de suas classes, facilita a implementação de novos barramentos para os mais diversos fins como, por exemplo, para estudos de desempenho.

Com a utilização do *framework* foram desenvolvidos dois dos barramentos estudados, o AMBA-AHB e o Avalon. Finalmente, foram executadas algumas simulações para se verificar a corretude das implementações. Esses dois barramentos podem ser usados futuramente para se fazer simulações de sistemas onde se deseja tê-los como os elementos de interconexão entre dispositivos mestres e escravos. A principal característica conseguida através dessa metodologia é que os barramentos se tornam intercambeáveis por suportarem uma interface comum.

Para trabalhos futuros existe a necessidade de implementação de novos barramentos como Coreconnect e Wishbone, dentre outros. Depois de desenvolvidos alguns novos barramentos pode-se fazer um estudo de comparação de desempenho entre todos os barramentos implementados, em diversas configurações de arquitetura, para se verificar a aplicabilidade e eficiência de cada um nessas arquiteturas.

Outra possibilidade para trabalhos futuros é a implementação de diversos algoritmos de arbitragem para serem usados nos barramentos desenvolvidos. Com diversos algoritmos de arbitragem disponíveis pode-se fazer um estudo comparativo.

Existe a necessidade da criação de uma biblioteca de periféricos. Esses novos periféricos aumentariam as possibilidades de arquiteturas de simulação.

7 Referências Bibliográficas

[1] Synopsys, Inc. SystemC, Versão 2.0.1; 2002. User's Guide.

[2] ArchC Team. The ArchC Architecture Description Language, v1.1.0; August 2004. Reference Manual.

[3] OCP International Partnership. A SystemC OCP Transaction Level Comunication Channel, version 1.5; October 20, 2004. Document.

[4] OCP International Partnership. Open Core Protocol Specification. Release 2.1; 2005. Specification

[5] Daniel D. Gajski; Sanjiv Narayan. Interfacing Incompatible Protocols using Interface Process Generation. 1995. White Paper.

[6] Jijay K. Madisetti. Rapid Digital System Prototyping: Current Practice, Future Challenges. 1996. White Paper.

[7] M. Caldari; M. Conti; M. Coppola; L. Pieralisi; C. Turchetti. Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0. 2003. White Paper.

[8] W. Cesário; A. Baghdadi; L. Gauthier; D. Lyonnard; G. Nicolescu; Y. Paviot; S. Yoo; A.A. Jerraya; M. Diaz-Nava. Component-Based Design Approach for Multicore SoCs. 2002; White Paper.

[9] Anssi Haverinen; Maxime Leclercq; Norman Weyrich; Drew Wingard. SystemC based SoC Communication Modeling for the OCP Protocol, version 1.0; October 14, 2002; White Paper.

[10] Kanishka Lahiri; Anand Raghunathan; Sujit Dey. Efficient Exploration of the SoC Communication Architecture Design Space. White Paper.

[11] ARM. AMBA AHB Cycle Level Interface Specification, Design Methodology Infrastructure and Tools, version 1.1.0; July 15, 2003. Specification.

[12] ARM. AMBA Specification, Rev 2.0; May 1999. Specification.

[13] ARM. AMBA AXI Protocol, v1.0; March 19, 2004. Specification

[14] ARM. AMBA 3 APB Protocol, v1.0; August 2004. Specification

[15] Altera. Avalon Interface Specification, version 2.4; January 2004. Reference Manual.

[16] Wishbone System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Revision B.3; September 7, 2002. Specification.

[17] IBM. Coreconnect. 128-bit Processor Local Bus. Version 4.6; July 2004. Architecture Specifications.

[18] Dongwan Shin; Lukai Cai; Andreas Gerslauer; Rainer Dömer; Daniel D. Gajski. System-on-Chip Transaction-Level Modeling Style Guide. July, 2004; Technical Report CECS-TR-04-24.

[19] Adam Rose; Stuart Swan; John Pierce; Jean-Michel Fernadez. Transaction Level Modeling in SystemC. White Paper.

[20] Lukai Cai; Daniel Gajski. Transaction Level Modeling in System Level Design. CECS Technical Report 03-10, Mar 28, 2003.

[21] Computer Systems Laboratory (LSC); IC-Unicamp; www.archc.org; ArchC 2.0.

[22] IBM. Coreconnect. On-Chip Peripheral Bus. Architecture Specifications. Version 2.1; April 2001. Specification.

[23] IBM. Coreconnect. Device Control Register Bus. Architecture Specifications. Version 3.5; January 27, 2006. Specification.

ANEXO I – Tabelas dos dados do segundo experimento

Essa seção contem todos os dados obtidos pelas simulações feitas no segundo experimento.

A tabela 9 mostra os dados obtidos na primeira etapa (10 mestres x 15 escravos) para o barramento AHB.

AHB						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	6495	6210	6255	6015	6180	6231
2	6480	6210	6255	6015	6165	6225
3	6480	6195	6240	6000	6165	6216
4	6465	6180	6225	5985	6150	6201
5	6405	6120	6165	5925	6090	6141
6	6300	6030	6075	6015	6000	6084
7	6120	5865	5910	5850	5835	5916
8	5475	5250	5295	5250	5220	5298
9	5040	4815	4860	4830	4800	4869
10	4335	4155	4185	4185	4140	4200

Tabela 9 – Dados da primeira etapa do segundo experimento - AHB

A tabela 10 mostra os dados obtidos na primeira etapa (10 mestres x 15 escravos) para o barramento Avalon.

Avalon						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	5715	5880	5850	5790	5865	5820
2	5715	5880	5850	5790	5865	5820
3	5715	5880	5850	5790	5865	5820
4	5715	5880	5850	5790	5865	5820
5	5715	5880	5850	5790	5865	5820
6	5715	5880	5850	5790	5865	5820
7	5715	5880	5850	5790	5850	5817
8	5715	5880	5850	5790	5850	5817
9	5715	5865	5850	5790	5850	5814
10	5715	5865	5850	5780	5850	5812

Tabela 10 – Dados da primeira etapa do segundo experimento - Avalon

A tabela 11 mostra os dados obtidos na segunda etapa (10 mestres x 10 escravos) para o barramento AHB.

AHB						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	6110	6010	6070	6050	6040	6056
2	6110	6000	6060	6050	6030	6050
3	6090	5990	6050	6030	6020	6036
4	6090	5980	6040	6030	6010	6030
5	6040	5940	5990	5980	5970	5984
6	5900	5800	5850	5840	5830	5844
7	5720	5630	5680	5670	5660	5672
8	5230	5140	5190	5180	5170	5182
9	4820	4740	4780	4780	4770	4778
10	4180	4110	4150	4140	4130	4142

Tabela 11 – Dados da segunda etapa do segundo experimento - AHB

A tabela 12 mostra os dados obtidos na segunda etapa (10 mestres x 10 escravos) para o barramento Avalon.

Avalon						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	5710	5770	5690	5840	5840	5770
2	5710	5770	5690	5840	5830	5768
3	5710	5770	5690	5840	5830	5768
4	5710	5770	5690	5840	5830	5768
5	5710	5770	5690	5840	5830	5768
6	5710	5760	5690	5830	5830	5764
7	5710	5760	5680	5830	5830	5762
8	5700	5760	5680	5830	5830	5760
9	5700	5760	5680	5830	5830	5760
10	5700	5760	5680	5830	5830	5760

Tabela 12 – Dados da segunda etapa do segundo experimento - Avalon

A tabela 13 mostra os dados obtidos na terceira etapa (10 mestres x 5 escravos) para o barramento AHB.

AHB						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	6390	6195	6150	6135	6135	6201
2	6385	6190	6145	6130	6130	6196
3	6370	6175	6135	6115	6115	6183
4	6360	6170	6125	6105	6110	6175
5	6340	6150	6105	6085	6090	6155
6	6165	5975	5930	5910	5915	5980
7	5945	5760	5720	5700	5705	5767
8	5615	5440	5400	5385	5390	5448
9	5205	5045	5005	4990	4995	5050
10	4500	4365	4330	4320	4320	4369

Tabela 13 – Dados da terceira etapa do segundo experimento - AHB

A tabela 14 mostra os dados obtidos na terceira etapa (10 mestres x 5 escravos) para o barramento Avalon.

Avalon						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	5630	5720	5345	5635	5590	5584
2	5630	5720	5345	5635	5585	5583
3	5630	5715	5345	5635	5585	5582
4	5630	5715	5340	5635	5585	5581
5	5630	5715	5340	5630	5585	5580
6	5625	5715	5340	5630	5585	5579
7	5625	5715	5340	5630	5585	5579
8	5615	5705	5330	5620	5575	5569
9	5615	5705	5330	5620	5575	5569
10	5610	5700	5325	5615	5570	5564

Tabela 14 – Dados da terceira etapa do segundo experimento - Avalon

A tabela 15 mostra os dados obtidos na quarta etapa (10 mestres x 2 escravos) para o barramento AHB.

AHB						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	6120	5606	5766	5706	5690	5778
2	6120	5606	5766	5706	5690	5778
3	6112	5600	5760	5700	5684	5771
4	6106	5594	5754	5694	5678	5765
5	6074	5566	5724	5664	5648	5735
6	6046	5494	5650	5590	5574	5671
7	5874	5332	5484	5428	5412	5506
8	5588	5060	5208	5152	5138	5229
9	5208	4706	4846	4794	4780	4867
10	4562	4116	4238	4190	4178	4257

Tabela 15 – Dados da quarta etapa do segundo experimento - AHB

A tabela 16 mostra os dados obtidos na quarta etapa (10 mestres x 2 escravos) para o barramento Avalon.

Avalon						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	5664	5506	5524	5482	5500	5535
2	5664	5506	5524	5482	5500	5535
3	5664	5504	5522	5480	5500	5534
4	5664	5504	5522	5480	5500	5534
5	5662	5504	5522	5480	5500	5534
6	5660	5502	5520	5478	5496	5531
7	5636	5478	5496	5454	5472	5507
8	5620	5464	5480	5438	5458	5492
9	5566	5408	5426	5384	5404	5438
10	5450	5298	5316	5276	5296	5327

Tabela 16 – Dados da quarta etapa do segundo experimento - Avalon

A tabela 17 mostra os dados obtidos na quinta etapa (10 mestres x 1 escravo) para o barramento AHB.

AHB						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	5603	5642	5536	5648	5630	5612
2	5603	5642	5536	5647	5630	5612
3	5603	5641	5536	5647	5630	5611
4	5602	5641	5535	5647	5630	5611
5	5587	5626	5521	5632	5615	5596
6	5572	5611	5505	5617	5599	5581
7	5543	5582	5477	5587	5570	5552
8	5465	5503	5399	5509	5492	5474
9	5192	5225	5128	5231	5216	5198
10	4441	4471	4382	4474	4462	4446

Tabela 17 – Dados da quinta etapa do segundo experimento - AHB

A tabela 18 mostra os dados obtidos na quinta etapa (10 mestres x 1 escravo) para o barramento Avalon.

Avalon						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	8524	8245	8252	8296	8198	8303
2	8524	8245	8252	8296	8198	8303
3	8524	8244	8252	8295	8197	8302
4	8491	8214	8221	8264	8167	8271
5	8152	7890	7896	7938	7844	7944
6	6307	6105	6110	6146	6074	6148
7	4879	4723	4727	4754	4695	4756
8	3171	3078	3080	3093	3066	3098
9	140	139	139	140	139	139
10	6	6	6	6	6	6

Tabela 18 – Dados da quinta etapa do segundo experimento - Avalon

A tabela 19 mostra os dados obtidos na sexta etapa (10 mestres x 1 escravo) para o barramento AHB com árbitro *round-robin*.

AHB						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	5647	5519	5493	5475	5507	5528
2	5648	5519	5493	5475	5507	5528
3	5643	5515	5489	5470	5503	5524
4	5648	5520	5494	5476	5508	5529
5	5646	5519	5493	5474	5507	5528
6	5646	5518	5492	5473	5506	5527
7	5644	5517	5492	5473	5505	5526
8	5646	5518	5492	5474	5506	5527
9	5644	5517	5491	5473	5505	5526
10	5646	5519	5493	5474	5507	5528

Tabela 19 – Dados da sexta etapa do segundo experimento - AHB

A tabela 20 mostra os dados obtidos na quinta etapa (10 mestres x 1 escravo) para o barramento Avalon com árbitro *round-robin.*

Avalon						
Processador	Execução 1	Execução 2	Execução 3	Execução 4	Execução 5	Média
1	5704	5552	5614	5515	5523	5582
2	5704	5552	5614	5515	5523	5582
3	5704	5551	5614	5514	5522	5581
4	5704	5552	5614	5515	5523	5582
5	5704	5551	5614	5514	5522	5581
6	5703	5551	5614	5514	5522	5581
7	5703	5551	5613	5514	5522	5581
8	5703	5551	5613	5514	5522	5581
9	5703	5551	5614	5514	5522	5581
10	5703	5551	5613	5514	5522	5581

Tabela 20 – Dados da sexta etapa do segundo experimento - Avalon