PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE ENGENHARIA
ENGENHARIA DE COMPUTAÇÃO

INTEGRAÇÃO DE UM MÓDULO DE COMPRESSÃO DE DADOS EM UM SOC ZYNQ-7000

TADEU DE SOUSA MARCHESE

TRABALHO DE CONCLUSÃO DO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

Orientador: Prof. FERNANDO GEHM MORAES

Porto Alegre
2017

TADEU DE SOUSA MARCHESE

INTEGRAÇÃO DE UM MÓDULO DE COMPRESSÃO DE DADOS EM UM SOC ZYNQ-7000

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Engenheiro de Computação da Faculdade de Engenharia da Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. DR. FERNANDO GEHM MORAES

Porto Alegre
2017

INTEGRAÇÃO DE UM MÓDULO DE COMPRESSÃO DE DADOS EM UM SOC ZYNQ-7000

RESUMO

A otimização da área de armazenamento é um tema altamente abordado em sistemas embarcados, pois permite reduzir os requisitos de tamanho de memória externa, além da redução do consumo de energia do sistema. Existem diversos algoritmos de compressão de dados em hardware, onde o princípio adotado é a remoção de informação redundante. Dentre os algoritmos de compressão, há os algoritmos sem perdas, que permitem recuperar a informação original; e os com perdas, que atingem uma maior taxa de compressão, porém o processo não é reversível.

O objetivo deste trabalho é desenvolver um projeto de hardware que intercepte as requisições de escrita e leitura de um processador embarcado à memória, realizando respectivamente a compressão e a descompressão dos dados. Através deste processo, a utilização de memória pelo sistema embarcado é reduzida.

Para desenvolvimento deste trabalho foi adotado um compressor de dados open source (XMatch), disponível no repositório OpenCores. Este IP foi integrado de forma eficiente com o processador embarcado e com a controladora de memória, através de um *wrapper* que realiza a interface entre as transações do barramento AXI4 e o protocolo do XMatch. O projeto de hardware foi validado com sucesso no ambiente Vivado, da Xilinx.

O projeto aqui apresentado permitiu um aprofundamento em diversos temas abordados durante o curso de Engenharia de Computação, como projeto de sistemas embarcados, técnicas de compressão de dados, e utilização de barramentos padronizados (AXI4). Outra contribuição, não menos relevante do trabalho, é a descrição do método de integração de hardware e software. Este trabalho é relevante para a indústria de sistemas eletrônicos, pois introduz um processo que pode reduzir os custos dos projetos de hardware.

Palavras-chave: Sistemas embarcados; FPGA; Compressão de dados; Otimização de área de armazenamento.

INTEGRATION OF A DATA COMPRESSION MODULE IN A ZYNQ-7000 SOC

ABSTRACT

Data storage optimization is a relevant topic in embedded systems design because, in addition to reducing the requirements of external memory size, this enables to decrease the system energy consumption. There are several in-hardware data compression algorithms available in the industry, them principle is the removal of redundant information. Among the compression algorithms, there are lossless algorithms, which allow recovering the original information; and lossy ones, which reach a higher compression rate, at the cost of being irreversible.

The goal of this work is to develop a hardware design that intercepts the writing and reading requests from an embedded processor to the memory, executing the compression and the decompression of the data respectively. Through this process, the amount of memory used by the embedded system is reduced.

In order to develop this work, we adopted the open source data compressor XMatch, available in the OpenCores repository. It was efficiently integrated with the embedded processor and the memory controller by means of a wrapper, which implements the interface between AXI4 bus transactions and the XMatch protocol. The hardware design was successfully validated in the Xilinx's Vivado environment.

The design herein presented allowed increasing the knowledge on several topics that were covered during the Computer Engineering course, such as embedded systems design, data compression techniques, and the use of standard buses (AXI4). Another relevant contribution is the description of the hardware and software integration method. This work is important to the electronics industry since it introduces a process that can reduce the costs of hardware designs.

Keywords: Embedded systems; FPGA; Data compression; Data storage optimization.

LISTA DE FIGURAS

LISTA DE ACRÔNIMOS

AMBA	*Advanced Microcontroller Bus Architecture*
AXI	*Advanced eXtensible Interface*
MIG	*Memory Interface Generator*
PS7	*Processing System*
DDR	*Double Data Rate*
PS	*Processing System*
PL	*Programming Logic*
CPU	*Central Processing Unit*
SDRAM	*Synchronous Dynamic Random-Access Memory*
NVM	*Non-Volatile Memory*
BRAM	*Block Random Access Memory*
OS	*Operating System*
EDA	*Electronic Design Automation*
IP	*Intellectual Property*
FPGA	*Field-Programmable Gate Array*
RTL	*Register-Transfer Level*
HDL	*Hardware Description Language*
VHDL	*Very High Speed Integrated Circuits*
VLSI	*Very-Large-Scale Integration*
FSM	*Finite-State Machine*
LUT	*Lookup Table*
FF	*Flip-Flop*

SUMÁRIO

1. INTRODUÇÃO

Com o crescimento populacional e a difusão tecnológica, os recursos de armazenamento de dados precisam ser usados de forma eficiente. Métodos de compressão podem ser utilizados para otimizar a área de armazenamento e reduzir o consumo de energia em sistemas computacionais, e em particular sistemas embarcados [2].

O surgimento de novas tecnologias de memórias trazem novos desafios e oportunidades acadêmicas [3]. Em sistemas embarcados a otimização de área de armazenamento é um tema abordado com frequência [4]. Neste contexto, o presente trabalho propõe a integração de um módulo de hardware dedicado entre o processador e a controladora de memória. Este módulo será responsável pela compressão e descompressão dos dados que transitam pelo barramento do SOC.

Existem diversos algoritmos de compressão conhecidos, os quais produzem diferentes resultados. Entretanto, a sua maioria se baseia no mesmo princípio, remoção de informação redundante [5]. Como o foco do trabalho é na integração e não no desenvolvimento de um hardware compressor, foi realizada uma busca por módulos de compressão de código aberto e uma breve análise sobre os métodos de compressão empregados.

O projeto de hardware digital é um processo complexo, requerendo métodos bem definidos de projeto. Para o projeto de sistemas eletrônicos, diversos métodos podem ser empregados, tais como *full custom*, *standard cells*, *gate arrays* e FPGAs [6]. Neste trabalho optamos por desenvolver o protótipo utilizando FPGAs, o que nos permite reduzir o tempo de projeto e produção. Outra vantagem dos dispositivos FPGAs é que estes são hoje SOCs, com processadores ARM e lógica programável. O protocolo de comunicação utilizado entre o processador ARM e a lógica programável é o AMBA AXI4 [7]. Assim, o presente projeto de TCC permite, além de aprofundar os conhecimentos em desenvolvimento de módulos de hardware digital, o estudo e compreensão da operação do barramento AMBA AXI4.

1.1 Motivação

A motivação principal deste trabalho é aprofundar os conhecimentos adquiridos ao longo do curso nas áreas de arquitetura de computadores, linguagem de descrição de hardware, prototipagem de sistemas embarcados. Em particular, o presente trabalho visa exercitar o fluxo de desenvolvimento de hardware digital a fim de integrar um IP em um SOC ARM utilizando o protocolo de comunicação AMBA AXI4.

1.2 Objetivos

O objetivo estratégico deste TCC é integrar um módulo compressor de dados no barramento de dados e endereços do processador a fim de reduzir a quantidade de memória necessária em um

sistema embarcado. Para atingir este objetivo estratégico, os seguintes objetivos específicos devem ser atendidos:

1. Exercitar o fluxo de desenvolvimento de hardware digital.
2. Compreender o funcionamento de uma ferramenta de Electronic Design Automation (EDA), como Vivado.
3. Estudo do protocolo de comunicação AMBA AXI4, utilizado amplamente pela indústria, e integração de um IP a este barramento.
4. Breve estudo sobre métodos de compressão de dados.
5. Desenvolvimento do hardware necessário para integrar o IP de compressão de dados ao barramento AMBA AXI4[1].

O presente trabalho se dispõe a cumprir os objetivos acima especificados por meio da proposta de arquitetura conceitual apresentada na figura 1.1.

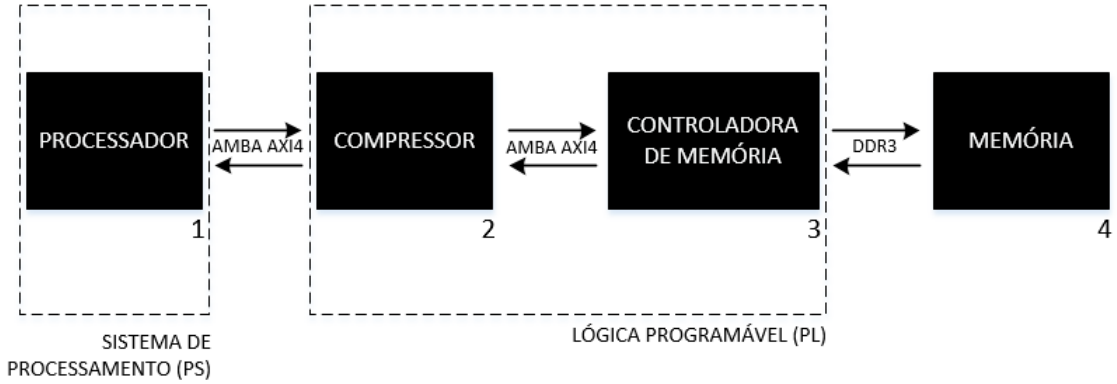

Figura 1.1 – Arquitetura conceitual proposta.

1. O processador consiste em uma unidade de hardware embarcada no SOC e deve ser capaz de executar um programa de teste a fim de gerar acessos à memória (4) através de instruções de leitura e escrita convencionais. As operações de leitura e escrita são realizadas no nível de hardware por meio do protocolo AMBA AXI4.
2. O compressor consiste em um módulo de hardware descrito em linguagem de descrição de hardware a fim de receber as operações do processador (1) via protocolo AMBA AXI4, modificá-las de acordo com o algoritmo de compressão e então enviá-las para a controladora de memória (3), também via protocolo AMBA AXI4. Este deve ser um IP de código aberto.

[1]Neste trabalho utilizaremos como sinônimos "*protocolo de comunicação AMBA AXI4*" e "*barramento AMBA AXI4*"

3. A controladora de memória se comunica com o compressor (2) através do protocolo AMBA AXI4. Esta consiste em um IP desenvolvido pela Xilinx para abstrair os protocolos de memória (4), como por exemplo DDR3.

4. A memória consiste em um componente de hardware da plataforma de desenvolvimento conectado ao dispositivo FPGA.

1.3 Organização do documento

A presente proposta de TCC está organizada em 5 Capítulos, como segue:

- **Capítulo 1** - Descreve o tema do presente trabalho, assim como a motivação e objetivos.
- **Capítulo 2** - Apresenta os fundamentos para o desenvolvimento para este projeto de TCC, incluindo: (*i*) a arquitetura da plataforma de desenvolvimento e o FPGA Zynq-7000; (*ii*) os modos de operação do protocolo de comunicação AMBA AXI4; (*iii*) o IP de compressão de dados utilizado no projeto, e uma breve explicação dos métodos de compressão por ele empregados.
- **Capítulo 3** - Apresenta inicialmente a operação detalhada do módulo de compressão de dados. Após esta descrição, apresenta-se a principal contribuição deste TCC, que é o desenvolvimento do módulo que integra o compressor de dados ao barramento AXI4.
- **Capítulo 4** - Apresenta a validação da proposta apresentando de forma incremental 3 projetos no ambiente Vivado (Xilinx): (*i*) projeto de referência que realiza a comunicação com o barramento AXI4; (*ii*) projeto que integra o barramento AXI4 ao IP de compressão, projeto este descrito em detalhes no Capítulo 3; (*iii*) projeto completo do hardware, com dados da síntese do hardware.
- **Capítulo 5** - Finaliza este TCC, apresentando as conclusões do trabalho realizado, e trabalhos futuros a serem desenvolvidos.

2. FUNDAMENTOS PARA O DESENVOLVIMENTO DO TCC

Este Capítulo apresenta os fundamentos para o desenvolvimento deste projeto de TCC. A Seção 2.1 apresenta a arquitetura da plataforma de desenvolvimento e o FPGA Zynq-7000. A Seção 2.2 apresenta os modos de operação do protocolo de comunicação AMBA AXI4. A Seção 2.3 apresenta o IP de compressão de dados utilizado no projeto, e uma breve explicação dos métodos de compressão por ele empregados.

2.1 Arquitetura base

Para desenvolvimento da arquitetura conceitual apresentada na Figura 1.1 o presente trabalho necessita de um SOC que integre unidades de processamento e lógica programável. A Seção 2.1.1 descreve a plataforma de desenvolvimento, focando nos periféricos disponíveis na mesma. Na sequência, Seção 2.1.2, descreve-se o dispositivo FPGA com recursos para processamento (ARM Cortex A9) e lógica programável.

2.1.1 Plataforma de Desenvolvimento Xilinx ZC706

O projeto proposto será desenvolvido em uma plataforma de desenvolvimento da Xilinx, modelo ZC706 [8], conforme mostra o diagrama de blocos da figura 2.1. O kit de desenvolvimento desta placa supre o requisito de configuração de hardware para o desenvolvimento do projeto proposto, pois possui um SOC Zynq-7000 que integra unidades de processamento com lógica programável, duas memórias DDR3, sendo uma *on-board* constituída por um conjunto de 4x256 MB SDRAMs, modelo MT41J256M8HX-15E, e a outra *off-board* DDR3 SODIMM, modelo MT8JTF12864HZ-1G6G1, ambas fabricadas pela empresa Micron.

Esta plataforma de desenvolvimento também possui outros componentes que podem ser utilizados caso necessário.

2.1.2 FPGA Zynq-7000

O FPGA Zynq-7000 é um SOC, e está organizado em duas partes que se comunicam entre si: Sistema de Processamento (PS) e Lógica programável (PL) como pode ser observado na figura 2.2.

O PS integra dois processadores físicos ARM® Cortex™-A9 MPCore, como podemos observar na figura 2.3. Estes processadores executam os programas de teste que realizam os acessos à memória.

As duas memórias DDR3, disponíveis pela plataforma de desenvolvimento Zynq-7000, estão posicionadas de forma diferente na arquitetura do SOC. A memória *on-board* está conectada no PS e é acessada diretamente pelo *hard-memory controller* embarcado no SOC, não sendo possível a inserção de qualquer módulo de lógica programável neste barramento. A memória *off-board* está

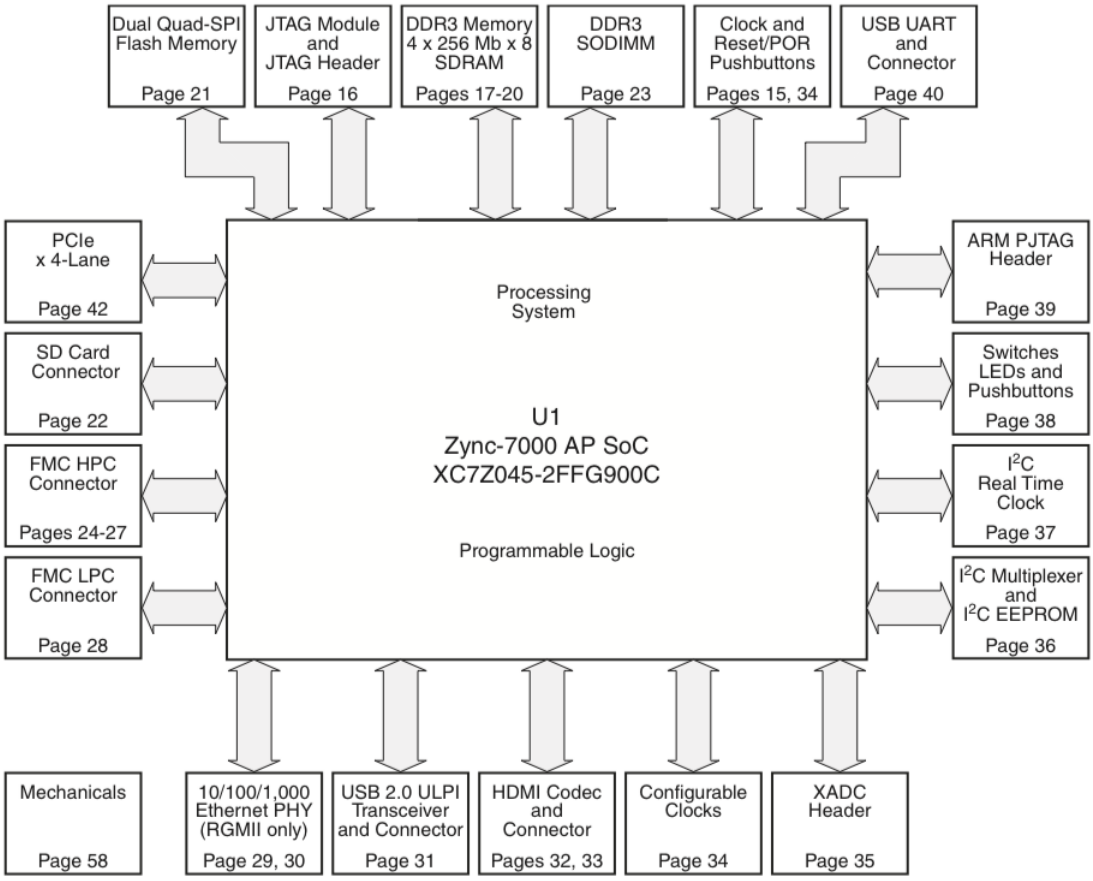

Figura 2.1 – Diagrama de blocos da plataforma de desenvolvimento ZC706 [8].

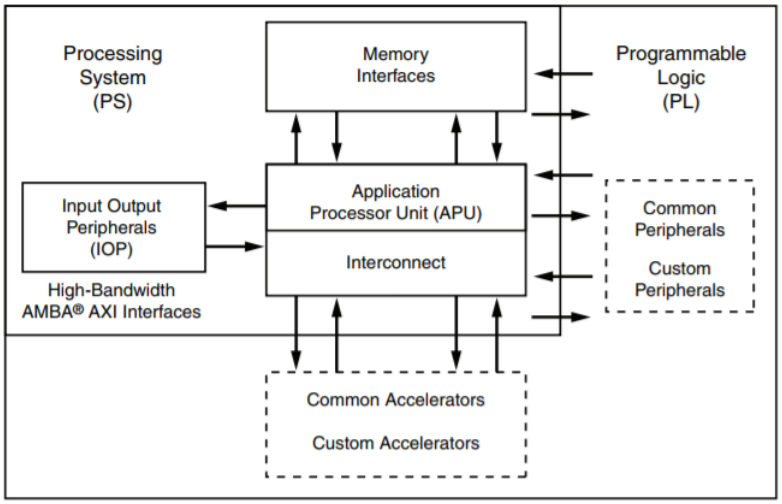

Figura 2.2 – Organização do SoC Zynq-7000 [9].

conectada na PL, o que nos permite a inserção de lógica programável de forma a interceptar as requisições do processador e se comunicar com a controladora de memória.

Neste trabalho utilizamos uma controladora de memória desenvolvida pela Xilinx em lógica programável, denominada MIG (Memory Interface Generator) [10]. Sendo assim, todos os acessos à memória *off-board* são realizados através desta controladora de memória, que recebe as requisições

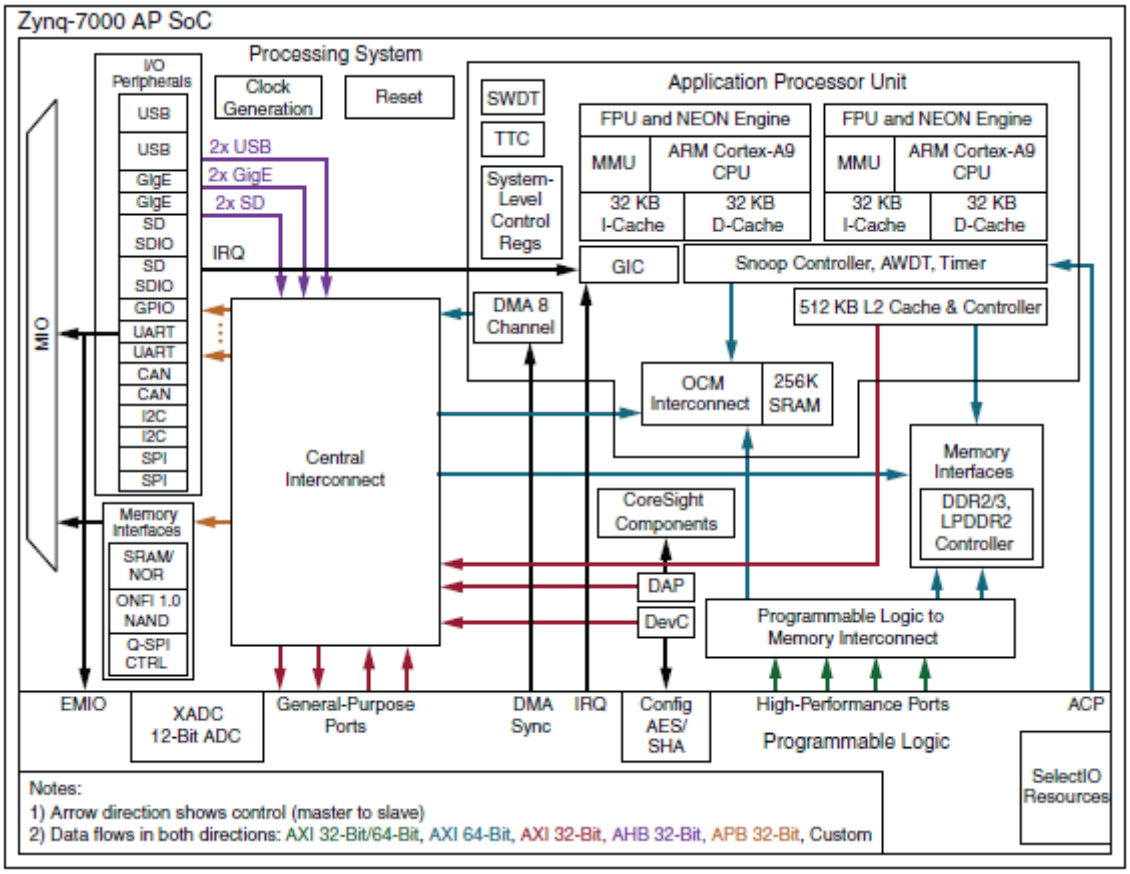

Figura 2.3 – Diagrama de blocos do SoC Zynq-7000 [9].

do processador seguindo o protocolo AXI4 e se comunica com a memória *off-board* seguindo o protocolo DDR3. Baseado no tipo de operação a controladora de memória realiza as seguintes tarefas:

- Para operações de leitura:
    1. Recebe a operação do processador ARM Cortex-A9;
    2. Lê um bloco de dados da memória DDR3 *off-board*;
    3. Retorna ao processador o dado solicitado.
- Para operações de escrita:
    1. Recebe a operação do processador ARM Cortex-A9;
    2. Escreve o bloco na memória *off-board*;
    3. Envia confirmação de escrita para o processador.

A figura 2.4 apresenta o diagrama de blocos de um projeto de referência no ambiente Vivado, o qual possui uma unidade de processamento e uma instância do IP da controladora de memória. A comunicação entre os módulos ocorre como segue:

- O processador se conecta à memória *on-board* através do barramento DDR, dado que este controlador de memória está embarcado no próprio dispositivo.
- O processador está conectado à controladora de memória por meio de um bloco chamado AXI Interconnect, que é responsável por gerenciar as conexões entre IPs que seguem o protocolo AMBA AXI4.
- A comunicação com a memória *off-board*, barramento DDR3, se dá através da controladora de memória (MIG). É neste barramento, entre o processador e a controladora de memória que devemos interceptar e modificar os acessos.

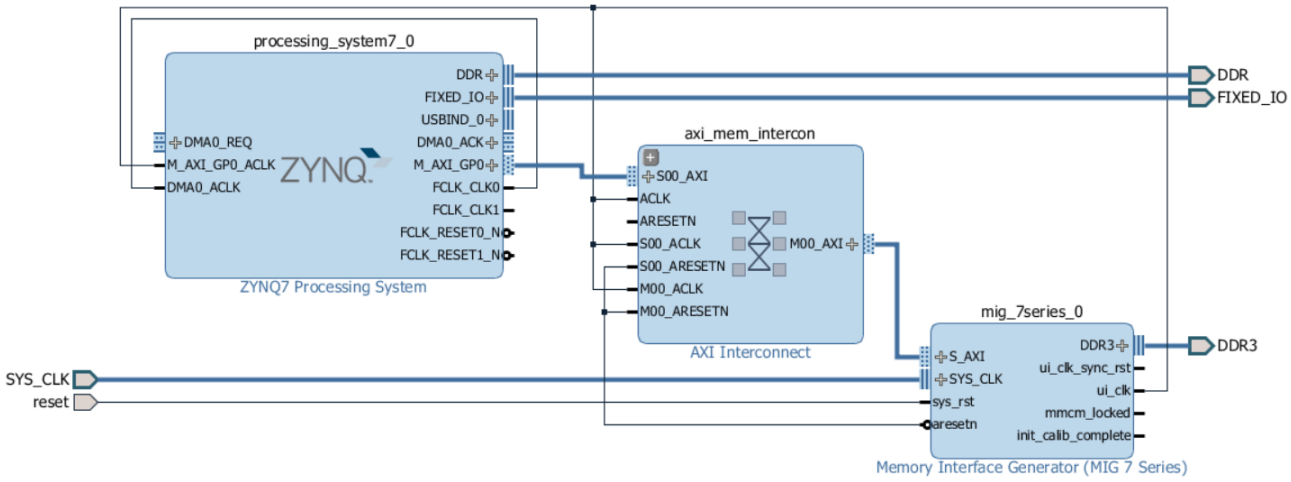

Figura 2.4 – Diagrama de blocos da arquitetura base.

2.2 Protocolo AMBA AXI4

O protocolo AMBA AXI4, proprietário da empresa ARM, tem por objetivo padronizar a comunicação de IPs em um SoC. Existem três protocolos de comunicação adotados pelo AXI4, conforme descrição abaixo:

- **AXI4-Memory Mapped Interface** - Este tipo de comunicação utiliza endereços mapeados no espaço de endereçamento do processador. Este protocolo é adotado neste trabalho, pois a proposta é receber no compressor de dados as transações efetuadas pelo processador em um endereço de memória, processá-las e então efetuar uma nova transação para a controladora de memória.
- **AXI4-Stream** - Protocolo adotado para aplicações em que o foco é a transmissão de dados em rajada, e não em endereços específicos.
- **AXI4-Lite** - Protocolo normalmente utilizado para controle e escritas únicas em registradores, não suportando transações em rajada.

A comunicação se inicia sempre pelo módulo denominado de mestre (*master*), e posteriormente o escravo (*slave*) responde de acordo com a configuração de resposta especificada pelo *master*.

2.2.1 AXI4-Memory Mapped Interface

Este protocolo é utilizado quando as transações são realizadas em um endereço específico. Devido ao suporte a operações de leitura e escrita simultâneas, esta interface implementa diversos barramentos independentes. Estes estão organizados em barramentos de dados e controle.

Barramentos de dados

Existem 5 barramentos de dados independentes, cada um com seu conjunto de sinais de controle.

- **AWADDR** – barramento que contém o endereço de escrita;
- **WDATA** – barramento com o dado a ser escrito no endereço de memória;
- **BRESP** – barramento para sinal de confirmação de escrita;
- **ARADDR** – barramento que contém o endereço de leitura;
- **RDATA** – barramento contendo o dado do endereço de memória de leitura.

Sinais de controle

Os sinais de controle são implementados para cada um dos barramentos de dados para sincronizar a troca de informações e garantir a correta comunicação dos módulos de hardware. Como podemos observar na figura 2.5 existem dois sinais de controle principais, VALID e READY.

- **VALID** - Sinaliza que o barramento de dados contém um dado válido e que pode ser lido enquanto este for ativo alto;
- **READY** - Sinaliza que a leitura do barramento já foi efetuada.

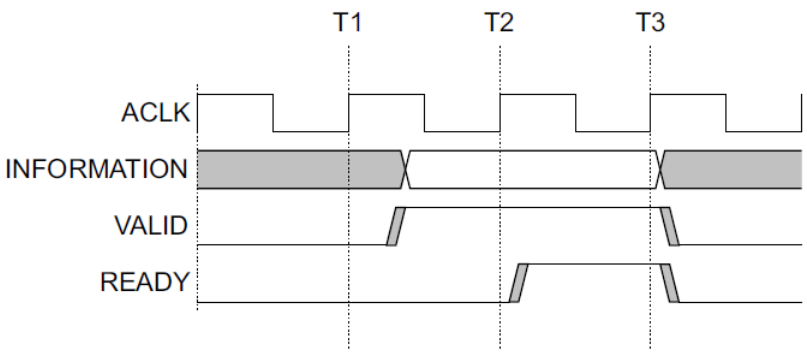

Figura 2.5 – Sinais para controle do barramento [7].

Cada barramento de dados possui o seu conjunto de sinais de controle, descritos abaixo:

- **AWVALID e AWREADY** – controle do barramento de endereço de escrita AWADDR;
- **WVALID e WREADY** – controle do barramento de dados de escrita WDATA;
- **BVALID e BREADY** – controle do barramento de resposta de escrita BRESP;
- **ARVALID e ARREADY** – controle do barramento de endereço de leitura ARADDR;
- **RVALID e RREADY** – controle do barramento de dados de leitura RDATA.

A interface AXI4-Memory Mapped suporta acessos de leitura e escrita simultâneos. Os barramentos de leitura e escrita operam de forma independente devido aos seus sinais de controle e múltiplos barramentos.

Operações de Leitura

As operações de leitura são iniciadas pelo componente mestre, o qual define as configurações da transação seguinte. Deve-se especificar o endereço de leitura, número de operações e ordenamento do conjunto de operações. O escravo então responde com os dados requisitados. A figura 2.6 mostra o fluxo de controle e de dados na comunicação entre os componentes mestre e escravo.

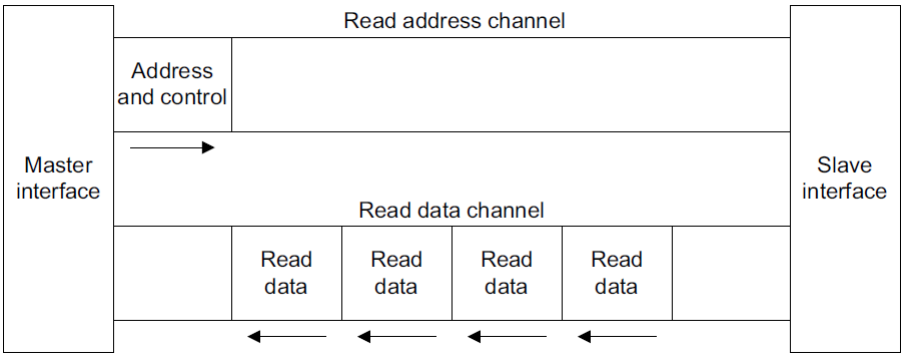

Figura 2.6 – Fluxo de comunicação mestre e escravo para uma operação de leitura [7].

Operações de Escrita

As operações de escrita são iniciadas pelo componente mestre, que define as configurações da transação seguinte. Deve-se especificar o endereço de escrita, número de operações, ordenamento do conjunto de operações e ainda os dados que devem ser escritos. O escravo então responde se a operação ocorreu de forma correta. A figura 2.7 mostra o fluxo de controle e de dados na comunicação entre os componentes mestre e escravo.

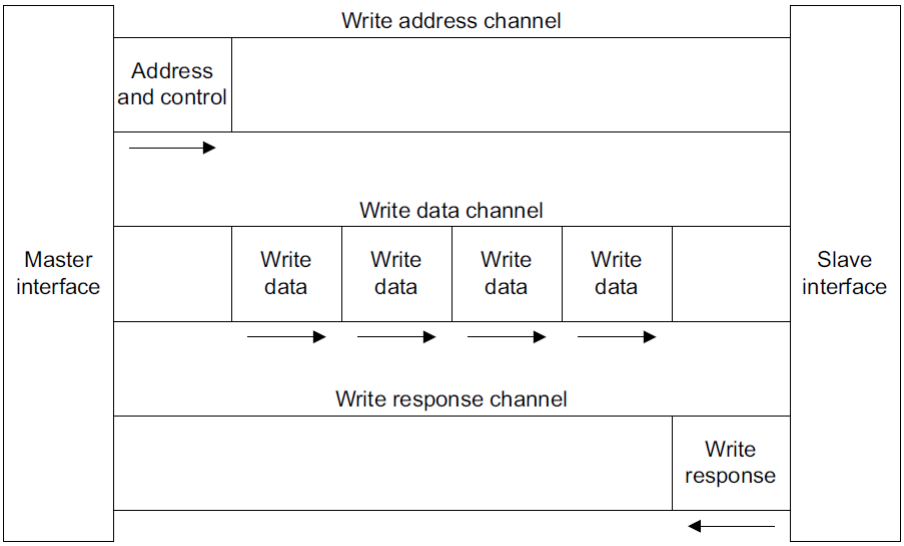

Figura 2.7 – Fluxo de comunicação mestre e escravo para uma operação de escrita [7].

Transações em Rajada

Uma das principais mudanças em relação às versões anteriores do protocolo AMBA AXI4 é a possibilidade de especificar apenas o endereço inicial e configurar uma rajada de transações [11]. Como pode ser observado na figura 2.8, a transação do exemplo foi configurada como uma rajada de 4 acessos e endereço inicial A11. Sendo assim, as transações foram realizadas em sequência sob o endereço A11, A12, A13 e A14. Desta forma não é necessário especificar o endereço de cada operação, o que reduz a complexidade e carga adicional na comunicação entre os módulos de hardware.

Figura 2.8 – Exemplo de duas rajadas de 4 acessos em sequência [11].

Para configurar uma transação em rajada é necessário modificar os seguintes sinais:

- **ARLEN e AWLEN** – Configuram a quantidade de acessos em rajada que serão realizados para leitura e escrita, seguindo a equação $AWLEN + 1$;
- **ARSIZE e AWSIZE** – Determinam o tamanho de cada palavra de uma operação em rajada para leitura e escrita, seguindo a equação $2^{AWSIZE+3}$;
- **ARBURST e AWBURST** – Configuram o ordenamento das operações em rajada. O ordenamento dos dados em rajado podem ser configurados de três formas: (*i*) fixa, todos os dados em um mesmo endereço; (*ii*) incremental, cada dado em um endereço diferente com incremento de

1 byte e (*III*) de forma alinhada, onde a equação $(INT(AWADDR/ARSIZE)) * ARSIZE$ é utilizada para encontrar o endereço alinhado. O modo incremental é utilizado para a maioria das aplicações.

Transações não ordenadas

As transações não precisam necessariamente seguir uma ordem específica. Neste caso, um sinal de identificação (ID) é utilizado para identificar as respostas das transações solicitadas. Como mostra a figura 2.9, as transações são iniciadas nos diferentes endereços e as suas respostas podem chegar no barramento de dados fora de ordem.

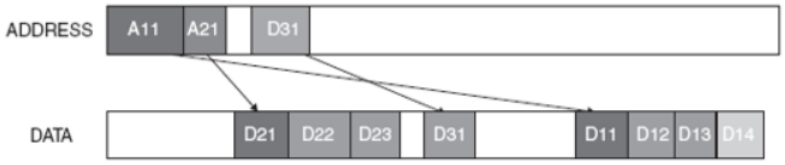

Figura 2.9 – Exemplo de transações não ordenadas [11].

2.3 Compressão de dados

Existem diversos algoritmos de compressão disponíveis na literatura. Eles produzem diferentes resultados, porém a sua maioria baseia-se no mesmo princípio, remoção de informação redundante [5].

São chamados de algoritmos sem perdas os algoritmos de compressão cujo produto é reversível, ou seja, o processo de descompressão produz o mesmo dado que fora comprimido. Algoritmos com perdas são os algoritmos de compressão cujo processo reverso não produz o mesmo dado que foi comprimido. Estes costumam ter uma capacidade de compressão maior se comparados aos algoritmos sem perda de dados.

Métodos de compressão são largamente utilizados em armazenamento de texto, transmissão de áudio, vídeo, entre outros. Abaixo listamos diferentes métodos disponíveis para compressão.

- **Técnicas básicas** - Muitos métodos de compressão foram originados antes mesmo do advento dos computadores. Braille, código morse, também métodos irreversíveis, como remoção de espaços ou caracteres, entre outros.
- **Métodos estatísticos** - Consideram a frequência de aparição dos dados para a definição dos símbolos.
- **Método de dicionário** - Relaciona os dados e o conjunto de dados à entradas de um dicionário. Um dado em um dicionário pode referenciar o índice de outra entrada e assim sucessivamente.
- **Compressão de imagem** - Uma variedade de métodos de compressão são combinados e aplicados em imagens. Os algoritmos podem relacionar também as diferenças entre pontos

adjacentes de uma imagem e dicionários. Um dos métodos mais eficientes envolve a sub-quantização das imagens.

- **Método de transformação wavelet** - Utiliza aproximação a funções matemáticas e transformadas.
- **Compressão de vídeo** - Métodos de compressão de imagem são utilizados também em vídeos. As diferenças e similaridades entre os quadros do vídeo também podem ser utilizadas para compressão.
- **Compressão de áudio** - A frequência de amostragem do sinal pode ser reduzida para a compressão. Dependendo do método e configurações adotadas as perdas podem ser mínimas.

Como o foco deste trabalho é na integração e não no desenvolvimento de um hardware compressor, foi realizada uma busca por módulos de compressão de código aberto e uma breve análise sobre os métodos de compressão nestes empregados.

2.3.1 Eficiência de compressão

Relacionando a saída do compressor aos dados de entrada é possível medir a eficiência do método de compressão em economia de espaço de armazenamento. Neste trabalho utilizamos a Equação 2.1 para a análise dos resultados de compressão.

$$\text{Economia em armazenamento (\%)} = (1 - \frac{\text{Dados comprimidos}}{\text{Dados descomprimidos}}) * 100 \qquad (2.1)$$

2.3.2 IPs de compressão

Para o desenvolvimento deste trabalho avaliamos três alternativas: (*i*) desenvolver o nosso próprio compressor; (*ii*) utilizar um compressor com perdas e (*iii*) utilizar um compressor sem perdas.

A primeira alternativa se mostrou viável em relação ao tempo de projeto, pois poderíamos implementar um algoritmo simples com o intuito de apenas provar o conceito. Os resultados seriam limitados ao algoritmo utilizado, além de não precisar entender um código de terceiro para realizar a integração ao barramento AMBA AXI4.

Quanto à segunda alternativa, encontramos através da plataforma *OpenCores* o IP *JPEG Hardware Compressor* [12], que foi desenvolvido por Victor Lopez Lorenzo, em 2005. O IP limitaria muito a solução para um tipo de arquivo e um tamanho específico de imagem (352x288), além de ser um algoritmo com perdas.

A terceira opção, e que se mostrou mais viável, foi a utilização do IP XMatch [1], também encontrado no *OpenCores*. Este IP implementa um algoritmo de compressão e descompressão sem perdas e possui vasta documentação além de publicações de resultados [13] [14]. Este IP foi desenvolvido pelo Dr. Jose Nunez-Yanez da Faculdade de Engenharia pertencente à Universidade de Bristol e se baseia na utilização de dicionários para remoção de informação redundante.

Dentre os fatores decisivos para esta escolha está a documentação disponível, algoritmo de compressão sem perdas eficiente, de código aberto e também o seu suporte a palavras de 32 bits, que é a mesma largura do barramento de dados utilizada pelo processador ARM Cortex-A9.

XMatch

Este IP foi projetado para trabalhar com blocos de dados de tamanho entre 8 bytes e 4 gigabytes. A sua arquitetura, apresentada na Figura 2.10 foi projetada para tornar independente os processos de compressão e descompressão, permitindo assim uma comunicação *full-duplex*, ou seja, compressão e descompressão simultâneas. Este compressor utiliza um dicionário de dados de 1024 bytes para encontrar redundância de informação. Cada entrada do dicionário é de 4 bytes.

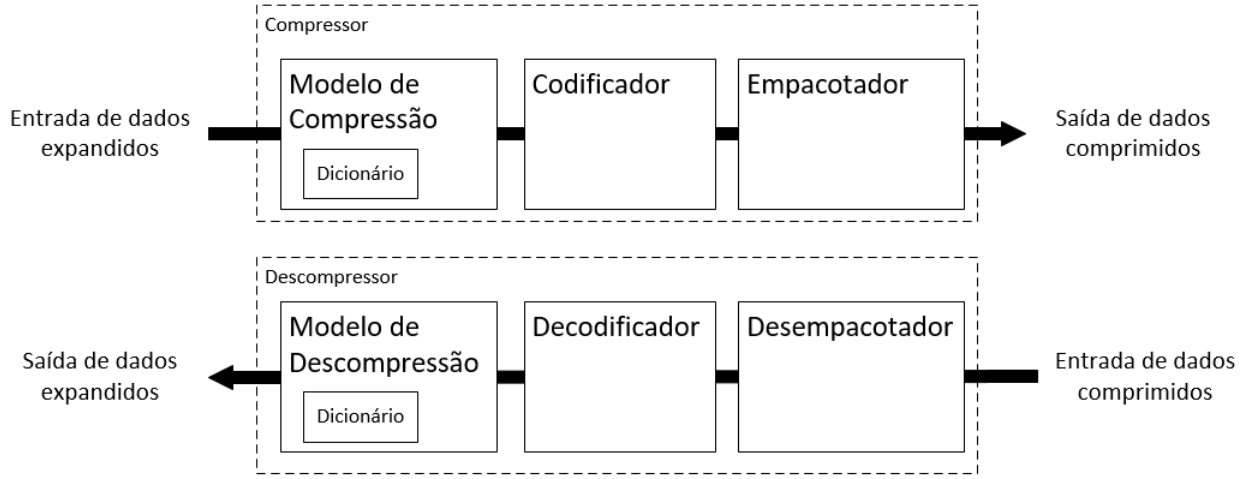

Figura 2.10 – Arquitetura do IP XMatch. [14].

Para codificar os bytes redundantes o IP XMatch utiliza a técnica de Huffman [15], esta emprega probabilidade de ocorrência dos símbolos de um conjunto para a determinar códigos de tamanho variável para os símbolos e então os seus posicionamentos em uma árvore binária completa [15]. A técnica de codificação RLI (*Run-Length-Internal*) é utilizada para comprimir caracteres repetidos em sequência. O objetivo do codificador RLI neste projeto é codificar repetições de localizações do dicionário. O IP ainda utiliza a técnica conhecida como PBC (*phased binary coding*) [16] para codificação da localização da parte redundante em um dicionário.

Na busca por documentação e conteúdo acadêmico relacionado a compressão de dados e ao IP XMatch foi encontrado também uma implementação que utiliza o XMatch em uma plataforma Zynq-7000 para transferir dados da OCM (*on-chip memory*) para a memória DDR via DMA (*direct memory access*). Esta implementação apresenta resultados positivos em relação a redução do consumo de energia ocasionado pela adoção do uso de compressão de dados [17].

3. PROJETO DE INTEGRAÇÃO XMATCH-AXI4

O presente Capítulo divide-se em duas seções. A Seção 3.1 detalha o IP XMatch, apresentando sua interface externa, a configuração dos modos de operação e exemplos de compressão e descompressão. A Seção 3.2 detalha o desenvolvimento do hardware que integra o IP XMatch ao barramento AXI4, correspondendo à principal contribuição deste projeto de TCC.

3.1 Descrição do IP XMatch

A Figura 3.1 apresenta a interface externa do IP XMatch. A interface externa deste IP se divide em três partes: (*i*) interface com um processador, responsável pela configuração do IP (*CPU port*); (*ii*) interface de compressão (*Compression Channel*); (*iii*) interface de descompressão (*Decompression Channel*).

Sinais do barramento de configuração (*CPU port*): os sinais desta interface são de uso comum para o compressor e para o descompressor, sendo assim redirecionado para as interfaces internas.

- **CS** (Entrada) - Seletor do modo de configuração;
- **RW** (Entrada) - Configura banco de registradores do IP para escrita ou leitura;
- **CLEAR** (Entrada) - Sinal de *reset*, para que o IP volte ao seu estado inicial;
- **CLK** (Entrada) - Sinal de *clock* do sistema;
- **ADDRESS** (Entrada) - Barramento para endereçamento do banco de registradores;
- **CONTROL** (Entrada/Saída)- Barramento utilizado para escrever ou ler do banco de registradores, sendo os 16 bits mais significativos destinados ao descompressor e os 16 bits menos significativos destinados ao compressor;
- **INTERRUPT_REQUEST** (Saída) - Interrupção utilizada para notificar algum erro no processo de compressão ou descompressão;
- **INTERRUPT_ACKNOWLEDGE** (Entrada) - Resposta à interrupção referente a erros no processo de compressão ou descompressão.

Existem 4 barramentos de dados, todos eles de 32 bits:

- **U_DATAIN** (Entrada) - Dados expandidos para compressão;
- **C_DATAOUT** (Saída) - Dados comprimidos, resultado do processo de compressão;
- **C_DATAIN** (Entrada) - Dados comprimidos para descompressão;
- **U_DATAOUT** (Saída) - Dados expandidos, resultado do processo de descompressão.

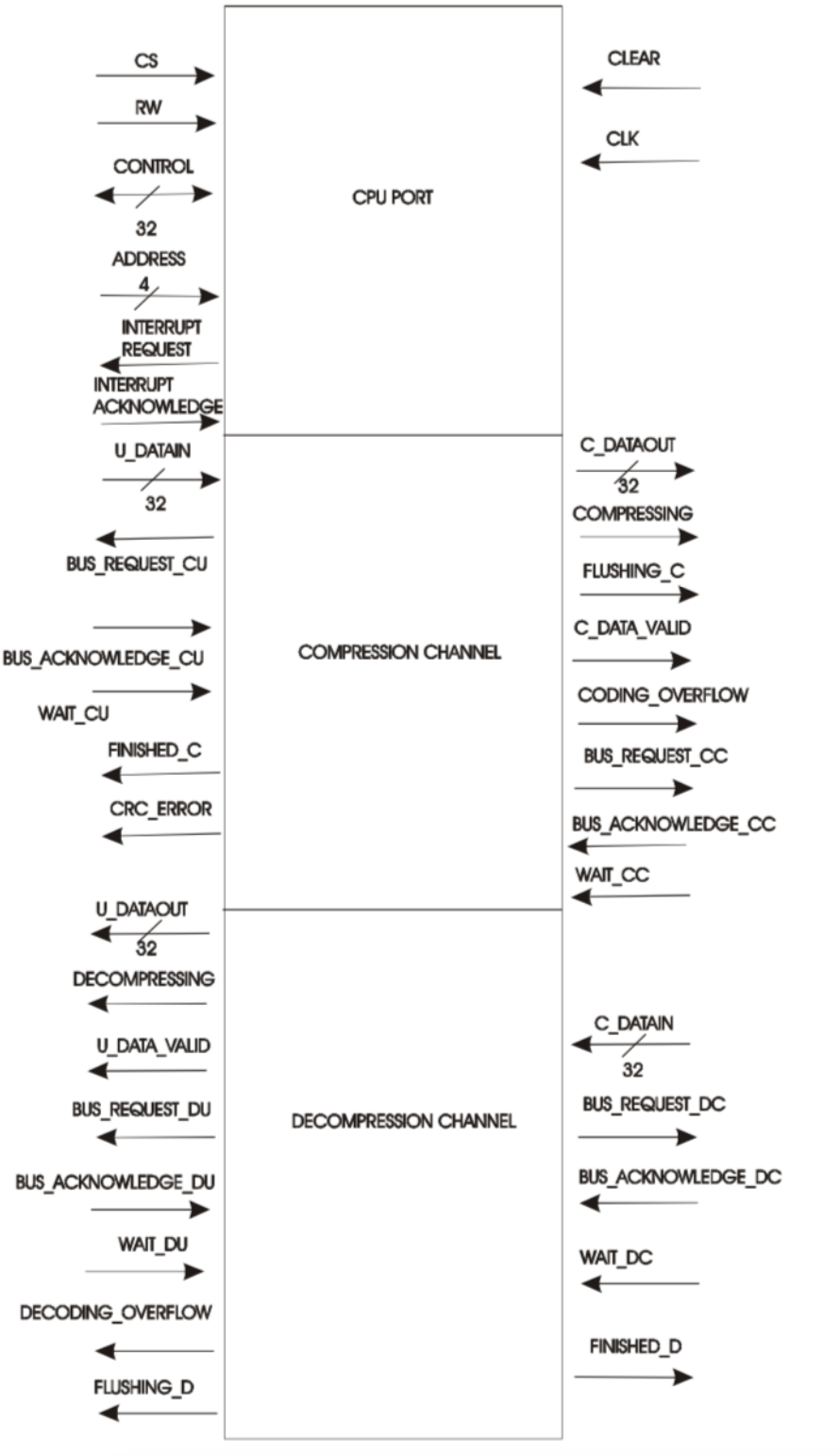

Figura 3.1 – Interface do IP de XMatch [1].

O acesso ao barramento de dados segue um protocolo de sincronização simples. O mesmo é formado por sinais de requisição (*request*) e reconhecimento (*acknowledgement*), conforme mostrado na figura 3.2. O IP XMatch solicita acesso aos barramentos para envio ou recebimento de dados atribuindo ao sinal de *request* nível baixo, e então recebe os dados 1 ciclo após o *acknowledgement*

assumir nível baixo.

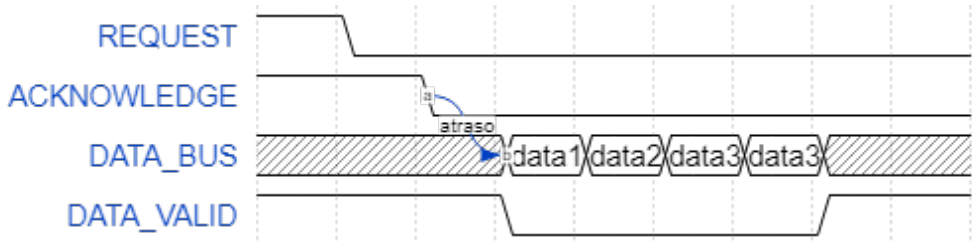

Figura 3.2 – Sinais *request*, *acknowledge* e *valid* do protocolo XMatch.

Sinais de requisição (*request*) e reconhecimento (*acknowledge*) do canal de compressão:

- **BUS_REQUEST_CU** (Saída) - requisita os dados para comprimir;
- **BUS_ACKNOWLEDGE_CU** (Entrada) - notifica que os dados serão disponibilizados após 1 ciclo de *clock*;
- **BUS_REQUEST_CC** (Saída) - requisita o acesso ao barramento para escrever os dados comprimidos;
- **BUS_ACKNOWLEDGE_CC** (Entrada) - avisa o IP que os dados podem ser escritos no barramento.

Sinais de requisição (*request*) e reconhecimento (*acknowledge*) do canal de descompressão:

- **BUS_REQUEST_DU** (Saída) - requisita os dados para descomprimir;
- **BUS_ACKNOWLEDGE_DU** (Entrada) - notifica que os dados serão disponibilizados após 1 ciclo de *clock*;
- **BUS_REQUEST_DC** (Saída) - requisita o acesso ao barramento para escrever os dados descomprimidos;
- **BUS_ACKNOWLEDGE_DC** (Entrada) - avisa o IP que os dados podem ser escritos no barramento.

O IP possui ainda um conjunto de sinais de controle do canal de compressão que informam o andamento do processo de compressão:

- **COMPRESSING** (Saída) - executando a compressão dos dados;
- **FINISHED_C** (Saída) - compressão finalizada;
- **FLUSHING_C** (Saída) - propagando a saída para o barramento adequado e limpando registradores internos;
- **C_DATA_VALID** (Saída) - sinaliza a presença de dados validos na saída do compressor.

Sinais de controle do canal de descompressão:

- **DECOMPRESSING** (Saída) - executando a descompressão dos dados;
- **FINISHED_D** (Saída) - descompressão finalizada;
- **FLUSHING_D** (Saída) - propagando a saída para o barramento adequado e limpando registradores internos;
- **U_DATA_VALID** (Saída) - sinaliza a presença de dados validos na saída do descompressor.

3.1.1 Configuração dos modos de operação

O sinal CS (*chip-select*) habilita o modo de configuração do IP. A configuração dos modos de operação se dá através de um banco de registradores. O barramento de endereço (*address*) é utilizado para selecionar o registrador requerido, enquanto o barramento de controle (*control*) é utilizado para leitura ou escrita deste registrador. O modo de acesso, para leitura ou escrita ainda é controlado pelo sinal RW (*read or write*), já que o barramento *control* é do tipo entrada e saída.

Endereços dos registradores para configuração do canal de compressão:

- **1100** - Ativa e desativa o canal de compressão;
- **1101** - Número de bytes do bloco descomprimido;
- **1110** - Número de bytes do bloco comprimido;
- **1111** - Código de verificação e correção de erros CRC para a compressão;
- **0000** - Informações do canal de compressão.

Os dois primeiros bits em nível alto são utilizados para endereçar registradores do compressor.

Endereços dos registradores para configuração do canal de descompressão:

- **1000** - Ativa e desativa o canal de descompressão;
- **1001** - Configura o número de bytes do barramento descompressão;
- **1010** - Reservado;
- **1011** - Código de verificação e correção de erros CRC para a descompressão;
- **0001** - Informações do canal de descompressão.

O primeiro bit em nível alto e o segundo em nível baixo endereça os registradores do descompressor.

3.1.2 Exemplo de Compressão

O processo de compressão pode ser dividido em três etapas: (*i*) configuração; (*ii*) entrada de dados; (*iii*) saída de dados.

Conforme pode se observar na figura 3.3 a primeira etapa consiste na configuração de dois registradores de endereço 01 e 00. O primeiro registrador recebe por meio do barramento de controle o tamanho do bloco de dados que deve ser comprimido, em bytes. Neste exemplo utilizamos 4 palavras de 32 bits, o equivalente a 16 bytes, ou 10 em base hexadecimal. O segundo registrador recebe o comando codificado de partida do compressor.

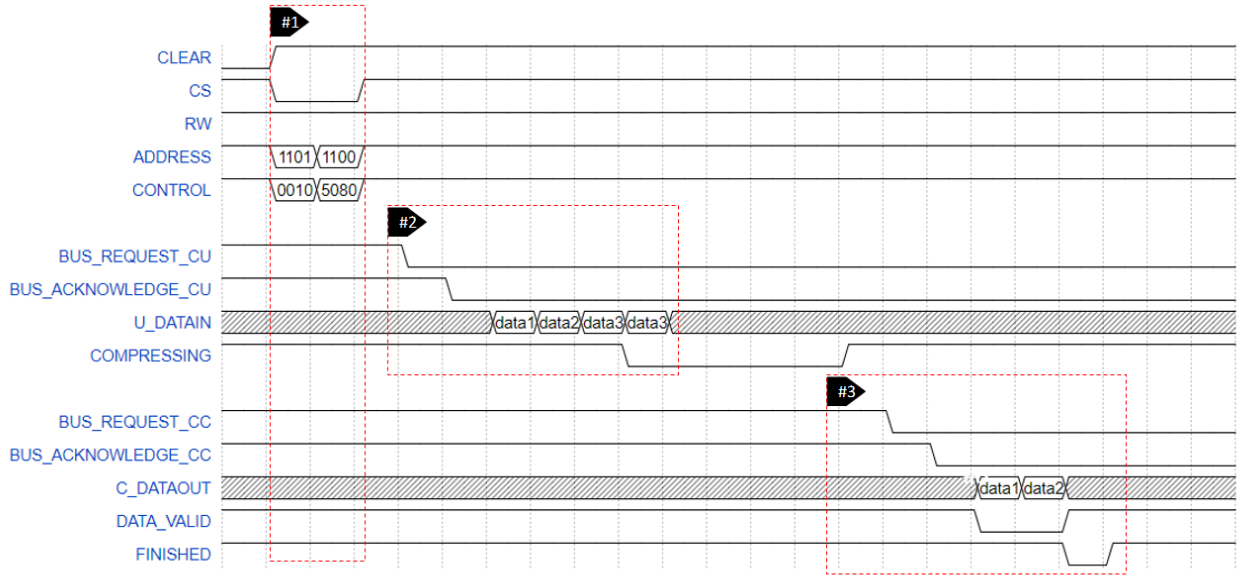

Figura 3.3 – Forma de onda para exemplo de compressão utilizando XMatch.

Ainda na etapa de configuração, o comando de partida do compressor é codificado em 16 bits, de modo que o bit 12 em nível lógico alto seleciona o modo de compressão, o bit 14 em nível lógico alto é responsável por dar início ao processo enquanto os demais bits são utilizados para a definição de um valor de limite *threshold* e ainda ativar o modo de teste. Este valor de *threshold* é utilizado internamente pelo algoritmo para configuração de atrasos e limites aceitáveis de espera.

Após o processo de configuração, se inicia a etapa de entrada de dados (marcador #2 na figura 3.3). Nesta etapa o IP notifica que está pronto para receber os dados atribuindo nível lógico baixo ao sinal *BUS_REQUEST_CU*. Os dados, que serão processados, devem ser disponibilizados, no barramento *U_DATAIN*, 1 ciclo após o sinal *BUS_ACKNOWLEDGE_CU* ir para nível lógico baixo. O sinal *COMPRESSING* informa que o IP está processando os dados.

A etapa de saída de dados (marcador #3 na Figura) se inicia quando o IP finaliza a compressão. Nesta etapa o IP requisita acesso ao barramento *C_DATAOUT* por meio do sinal *BUS_REQUEST_CC*. Após o recebimento do sinal *BUS_ACKNOWLEDGE_CC* o dado é disponibilizado no barramento. O sinal *DATA_VALID* é utilizado para informar que existe um dado válido no barramento, enquanto o sinal *FINISHED* informa o fim da operação de compressão.

3.1.3 Exemplo de Descompressão

O processo de descompressão pode ser dividido em três etapas: (*i*) configuração; (*ii*) entrada de dados; (*iii*) saída de dados.

Conforme pode se observar na figura a primeira etapa (marcador #1 na Figura) consiste na configuração de dois registradores de endereço 01 e 00. O primeiro registrador recebe por meio do barramento de controle o tamanho expandido do bloco de dados que deve ser descomprimido, em bytes. Neste exemplo utilizamos 4 palavras de 32 bits, o equivalente a 16 bytes, ou 10 em base hexadecimal. O segundo registrador recebe o comando codificado de partida do descompressor.

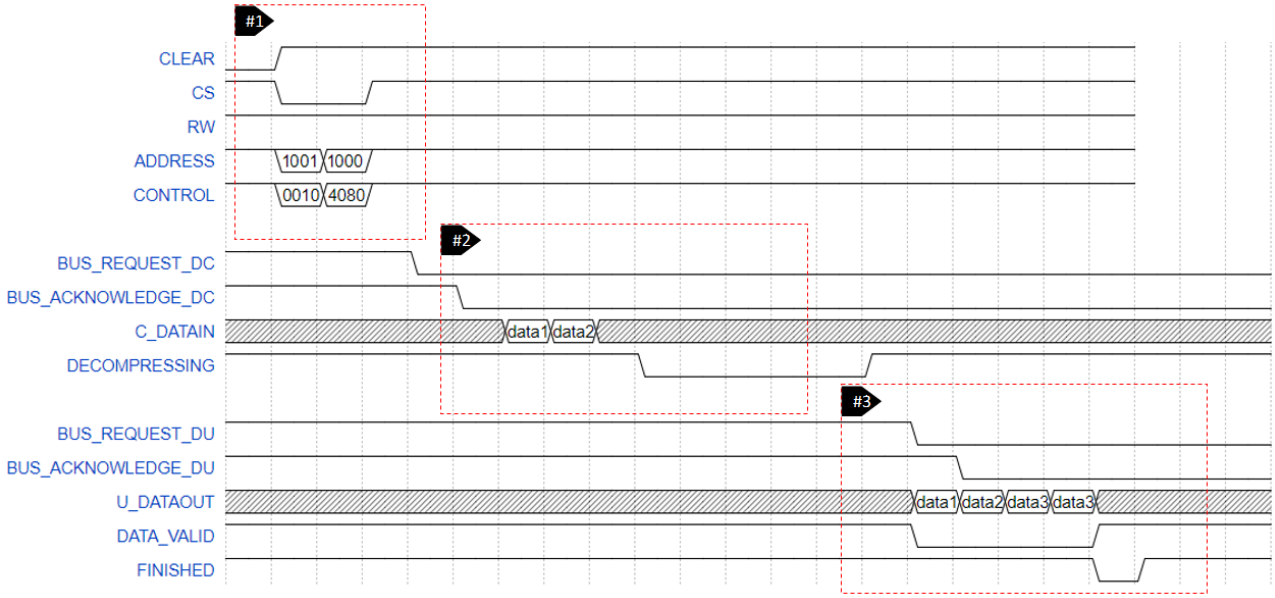

Figura 3.4 – Forma de onda para exemplo de descompressão utilizando XMatch.

Ainda na etapa de configuração, o comando de partida do compressor é codificado em 16 bits, de modo que o bit 12 em nível lógico baixo seleciona o modo de compressão, o bit 14 em nível lógico alto é responsável por dar início ao processo, enquanto os demais bits são utilizados para a definição de um valor de limite *threshold* e ainda ativar o modo de teste. Este valor de *threshold* é utilizado internamente pelo algoritmo para configuração de atrasos e limites aceitáveis de espera.

Após o processo de configuração, se inicia a etapa de entrada de dados (marcador #2 na Figura). Nesta etapa o IP notifica que está pronto para receber os dados atribuindo nível lógico baixo ao sinal *BUS_REQUEST_DC*. Os dados que serão processados devem ser disponibilizados no barramento *C_DATAIN* 1 ciclo após o sinal *BUS_ACKNOWLEDGE_DC*. O sinal *DECOMPRESSING* informa que o IP está processando os dados internamente.

A etapa de saída de dados (marcador #3 na Figura) se inicia quando o IP finaliza a descompressão. Nesta etapa o IP requisita acesso ao barramento *U_DATAOUT* por meio do sinal *BUS_REQUEST_DU*. Após o recebimento do sinal *BUS_ACKNOWLEDGE_DU* o dado é disponibilizado no barramento. O sinal *DATA_VALID* é utilizado para informar que existe um dado válido no barramento, enquanto o sinal *FINISHED* informa o fim da operação de descompressão.

3.1.4 Análise de Desempenho do IP

Para análise de eficiência e performance do algoritmo de compressão XMatch utilizamos um software, desenvolvido pelos autores do XMatch [14], que implementa este mesmo método com tamanho de bloco e tamanho de dicionário configuráveis por parâmetro. Enquanto o tamanho do bloco varia de 256 bytes a 256 kilobytes, o dicionário foi fixado em 1024 bytes, que corresponde ao mesmo tamanho implementado em lógica programável.

As bases de dados Canterbury e Calgary foram utilizadas para esta análise devido a sua adoção na indústria e os tipos de dados que as compõem serem representativos de dados reais [18]. Ambas as bases de dados Canterbury e Calgary tiveram todos os seus dados concatenados para esta execução.

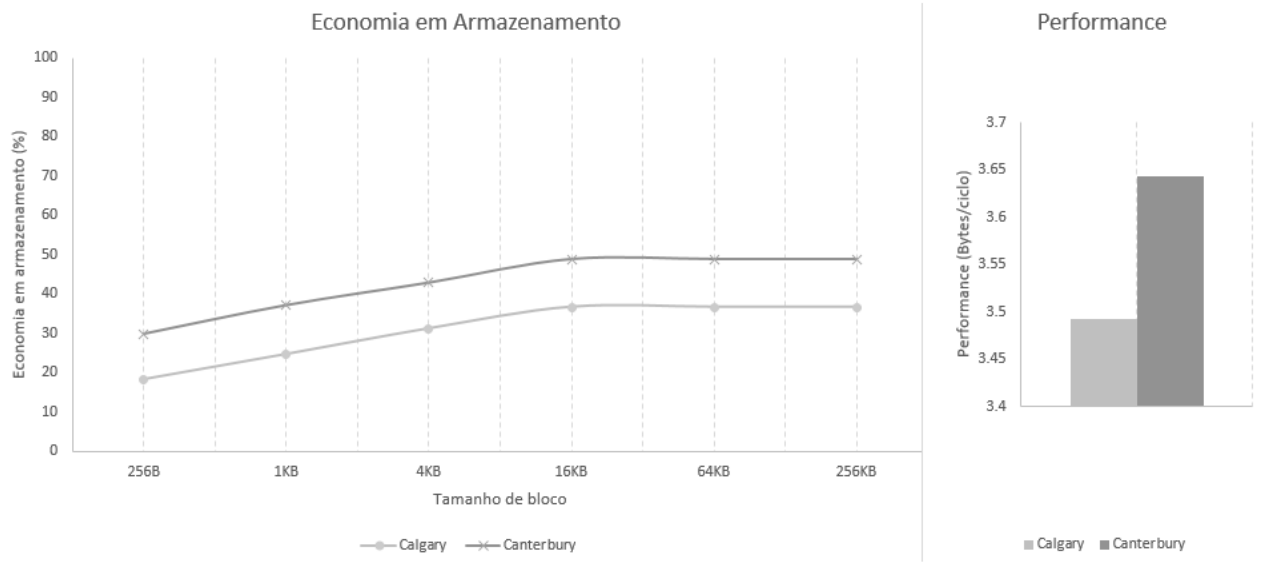

Figura 3.5 – Eficiência e performance do método de compressão.

Como podemos observar na figura 3.5, a medida em que aumentamos o tamanho do bloco de dados conseguimos uma maior economia em armazenamento, a qual pode alcançar até 50% em relação ao dado original. O desempenho varia de acordo com as bases de dados utilizadas, em torno de 3.5 bytes por ciclo de relógio (*clock*).

3.2 Arquitetura do invólucro (*wrapper*)

O IP XMatch implementa um protocolo próprio de comunicação em suas interfaces externas. A fim de integrá-lo ao barramento AMBA é necessário a implementação de interfaces que sigam o protocolo AXI4. A implementação destas interfaces AXI4 foram desenvolvidas como uma forma de *wrapper*, que envolve o IP implementando o protocolo externo adequado para comunicação com os demais componentes sem alterações em seu comportamento interno. Neste trabalho nomeamos a entidade do *wrapper* de **AXI Core** (figura 3.6).

O controle do IP XMatch é realizado por um conjunto de seis máquinas de estado, 3 para o compressor e 3 para o descompressor.

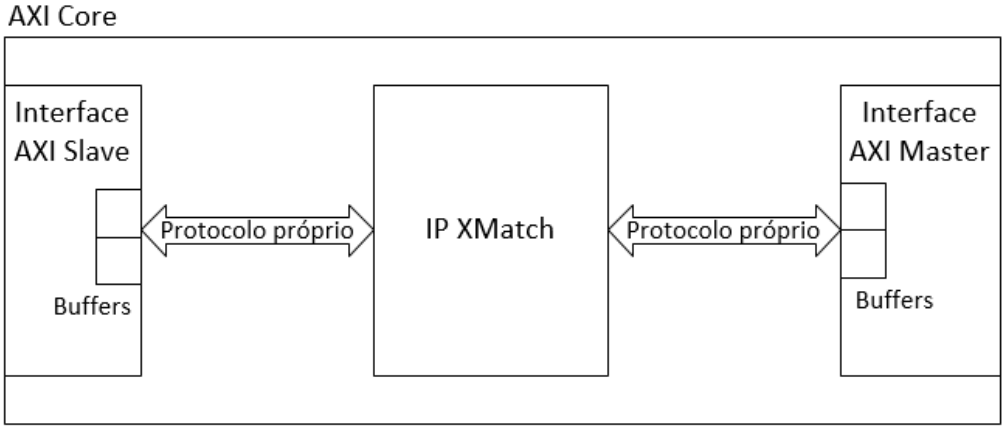

Figura 3.6 – Diagrama arquitetura do *wrapper*.

Quatro *buffers* ainda são utilizados para armazenar os dados de entrada e saída do compressor e descompressor:

- *write_slave_buffer* - Responsável por armazenar os dados provenientes do processador no processo de escrita, que serão enviados ao compressor.
- *write_master_buffer* - Responsável por armazenar os dados provenientes do compressor no processo de escrita, que serão enviados a memória.
- *read_master_buffer* - Responsável por armazenar os dados provenientes da controladora de memória, no processo de leitura.
- *read_slave_buffer* - Responsável por armazenar os dados provenientes do descompressor no processo de leitura, que serão enviados ao processador.

3.2.1 Processo de Compressão

O processador inicia o processo de compressão através de requisições de escrita em memória. Estas requisições são transferidas para o barramento AXI4. O processador implementa a interface *master* enquanto o *wrapper* implementa a interface *slave*. Para tratar e responder a estas requisições, três máquinas de estados (FSM) controlam o processo de compressão. A figura 3.7 mostra o diagrama das três FSMs responsáveis pelo processo de compressão de dados:

- **FSM 1** - AXI *Slave*, realiza a interface com o processador (PS7);
- **FSM 2** - XMatch, realiza a interface com o IP compressor (XMatch);
- **FSM 3** - AXI *Master*, realiza a interface com a controladora de memória (MIG).

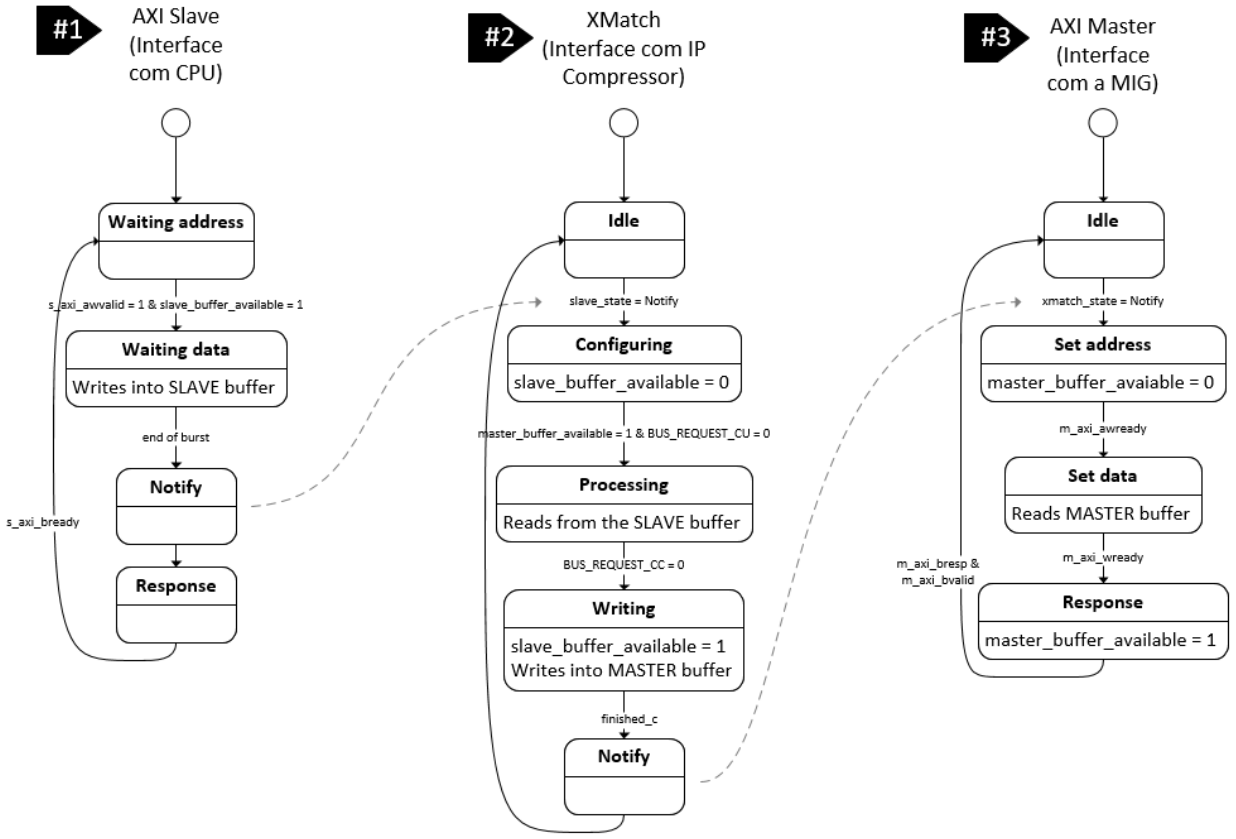

Figura 3.7 – Máquinas de estados do *wrapper* compressor.

AXI *Slave* - FSM 1

Esta FSM é constituída de 4 estados: (*i*) *Waiting_address*; (*ii*) *Waiting_data*; (*ii*) *Notify* e (*iv*) *Response*. A mesma é responsável pelo controle da interface de comunicação com o processador (PS7). A comunicação é toda realizada seguindo o protocolo AMBA AXI4. Primeiramente o endereço é atribuído e em seguida os dados são enviados, utilizando em ambos barramentos os seus sinais de controle, *valid* e *ready*.

O processo de escrita se inicia quando o sinal *s_axi_awvalid* assume nível lógico alto, notificando um endereço válido no barramento de endereço (*s_axi_awaddr*).

A FSM 1 ficará no estado inicial e o sinal *s_axi_awready* ficara em nível lógico baixo até que o buffer de dados *write_slave_buffer*[1] esteja disponível. Somente quando as condições para o início do envio de dados forem atendidas, a FSM1 assumirá o estado *Waiting_data*.

Como a interface AXI possui uma sequência de sinais e temporização diferente do IP XMatch, é necessário a utilização de um buffer intermediário entre o AXI e o XMatch, o *write_slave_buffer*. O *write_slave_buffer* é implementado utilizando um banco de registradores, no qual a FSM 1 escreve os dados, e a FSM 2 os consome.

No estado *Waiting_data*, os dados que devem ser escritos na memória são transmitidos e copiados

[1]Buffer responsável por armazenar os dados provenientes do processador (PS7), que serão enviados ao compressor.

no *write_slave_buffer* sempre que o sinal *s_axi_wvalid* estiver em nível alto. O sinal *s_axi_wready* notifica que o dado foi capturado e então o ponteiro de escrita do *write_slave_buffer* é incrementado para o armazenamento do próximo dado.

O sinal *s_axi_wlast* informa o fim de uma operação em rajada e o início do estado *Notify* (*III*). Este estado destrava a FSM 2, que se encontrava em estado de espera.

Em seguida, o estado *Response* é responsável por enviar uma resposta ao processador informando que a comunicação para esta operação está encerrada. Após o encerramento da operação, a FSM 1 assume o estado *Waiting_address* novamente.

Neste ponto a FSM 1 retorna ao seu estado inicial, aguardando um novo endereço para o recebimento de novas operações. A FSM 2 iniciará o controle do processo de compressão, enquanto a FSM 3 encontra-se em estado de espera.

XMatch - FSM 2

Esta FSM é constituída de 5 estados: (*i*) *Idle*; (*ii*) *Configuring*; (*iii*) *Processing*; (*iv*) *Writing* e (*v*) *Notify*. A mesma é responsável pelo controle da interface de comunicação com o IP XMatch. A comunicação é toda realizada seguindo um protocolo proprietário (sessão 3.1).

O processo de compressão se inicia quando a FSM 1 assume o estado *Notify*. Neste ponto, o *write_slave_buffer* contém o bloco de dados a ser comprimido, e a FSM 2 pode iniciar o processo de compressão.

A primeira etapa da compressão é a configuração do IP no modo de operação adequado. Esta etapa é realizada durante o estado *Configuring*, conforme descrito da sessão 3.1. Nesta etapa, o sinal *CLEAR* deve receber nível lógico alto enquanto os sinais *CS*, *RW*, *ADDRESS* e *CONTROL* são utilizados para efetuar a configuração do IP no modo de compressão via banco de registradores.

Após a configuração o IP está pronto para receber o bloco de dados. No estado *Processing*, os dados são copiados do *write_slave_buffer* para o barramento *U_DATAIN*.

No fim do processo de compressão, que se dá no estado *Processing*, o IP requisita acesso ao barramento *BUS_REQUEST_CC* para escrever o resultado.

A FSM 2 assume o estado *Writing* para escrever o resultado da compressão no buffer de dados *write_master_buffer*[2], deixando o *write_slave_buffer* disponível para o armazenamento dos dados da próxima operação.

O estado *Notify* notifica a FSM 3 de que o dado comprimido já está pronto para ser escrito na memória.

AXI *Master* - FSM 3

Esta FSM é constituída de 4 estados: (*i*) *Idle*; (*ii*) *Set_address*; (*iii*) *Set_data* e (*iv*) *Response*. A mesma é responsável pelo controle da interface de comunicação com a controladora de memória

[2]Buffer responsável por armazenar os dados provenientes do compressor, que serão escritos através da controladora de memória (MIG).

(MIG). A comunicação é toda realizada seguindo o protocolo AMBA AXI4. Primeiramente, o endereço é atribuído, e em seguida os dados são enviados, ambos barramentos com seus conjuntos de sinais de controle, *valid* e *ready*.

O processo de escrita do dado comprimido na memória se inicia quando a FSM 2 assume o estado *Notify*, o que faz com que a FSM 3 assuma o estado *Set_address*.

No estado *Set_address* o endereço de memória inicial, para a escrita do bloco, é informado à controladora de memória (MIG), através do sinal *m_axi_awaddr*. O sinal *m_axi_awvalid* informa que o endereço do barramento *m_axi_awaddr* é válido.

A configuração de endereço só se finaliza com o recebimento do sinal *m_axi_awready*, que informa a captura do endereço válido e faz com que a FSM 3 assuma o próximo estado (*Set_data*).

O estado *Set_data* é responsável por escrever no barramento *m_axi_wdata* o dado comprimido. Neste estágio o *write_master_buffer* já contém o bloco de dados comprimido, bastando então realizar a cópia do banco de registradores *write_master_buffer* para o barramento *m_axi_wdata*. O sinal *m_axi_wvalid* deve assumir nível lógico 1 sempre que houver dado válido no barramento *m_axi_wdata*. Além disso, o ponteiro de leitura do *write_master_buffer* é incrementado sempre que o sinal *m_axi_wready* assume nível lógico alto.

Quando o ponteiro de leitura chega ao fim do *write_master_buffer* a FSM 3 assume o estado *Response*. Este estado é responsável por receber a resposta da MIG, através do sinal *m_axi_bresp*, sendo que *m_axi_bvalid* notifica quando há uma resposta válida, e *m_axi_bready* notifica a captura da resposta. Após a captura da resposta esta máquina de estados volta a assumir o estado inicial *Idle*.

3.2.2 Processo de Descompressão

O processador inicia o processo de descompressão através de requisições de leitura em memória. Estas requisições são transferidas para o barramento AXI4. O processador implementa a interface *master* enquanto o *wrapper* implementa a interface *slave*. Para tratar e responder a estas requisições, três máquinas de estados (FSM) controlam o processo de descompressão. A figura 3.8 mostra o diagrama das três FSMs responsáveis pelo processo de descompressão de dados:

- **FSM 1** - AXI *Slave*, realiza a interface com o processador (PS7);
- **FSM 2** - XMatch, realiza a interface com o IP compressor (XMatch);
- **FSM 3** - AXI *Master*, realiza a interface com a controladora de memória (MIG).

AXI *Slave* - FSM 1

Esta FSM é constituída de 5 estados: (*i*) *Waiting_address*; (*ii*) *Waiting_data*; (*ii*) *Waiting_decompression*; (*iv*) *Response* e (*v*) *Done*. A mesma é responsável pelo controle da interface de comunicação com o processador (PS7). A comunicação é toda realizada seguindo o protocolo

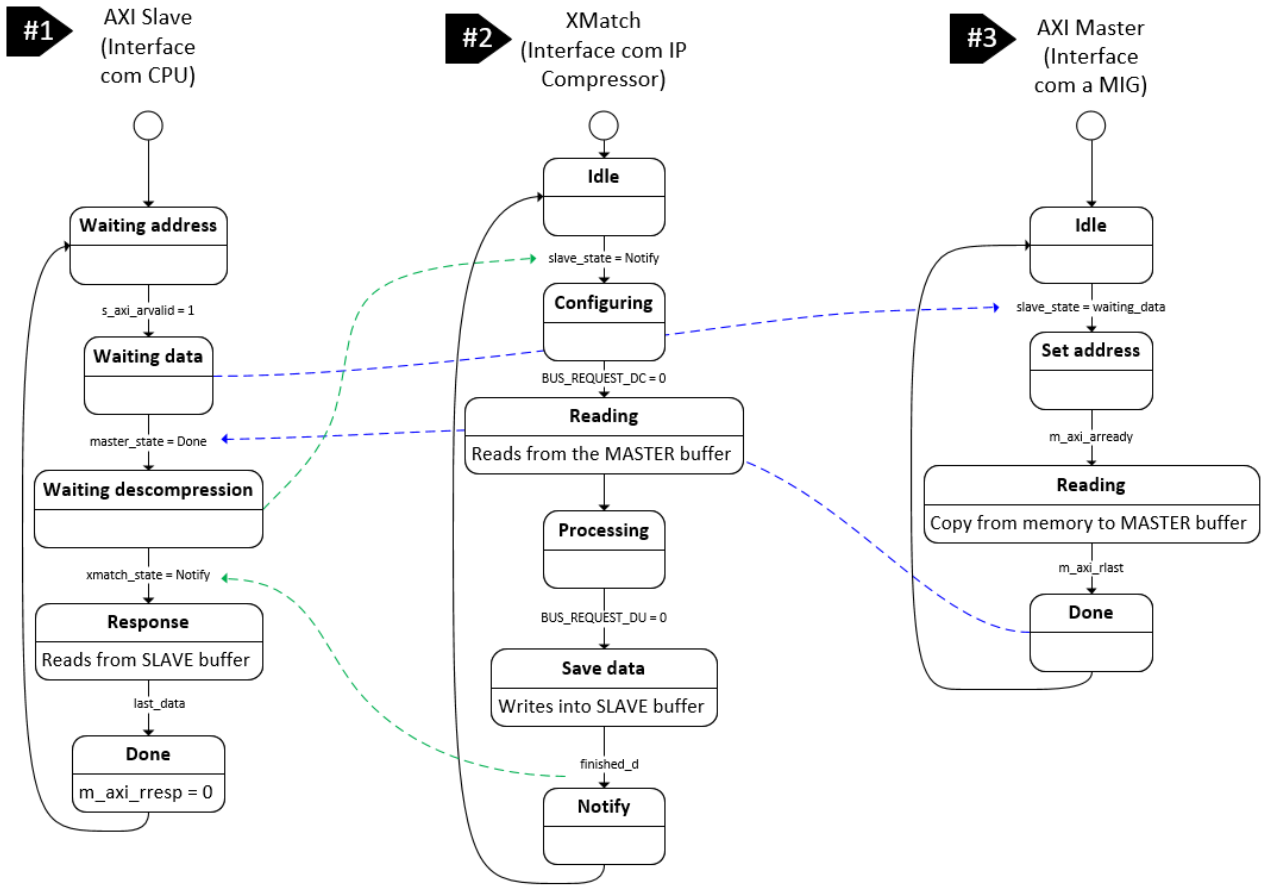

Figura 3.8 – Máquinas de estados do *wrapper* descompressor.

AMBA AXI4. Primeiramente o endereço é atribuído e em seguida os dados são enviados, ambos barramentos com seus conjuntos de sinais de controle, *valid* e *ready*.

O processo de leitura se inicia quando o sinal *s_axi_arvalid* assume nível lógico alto, notificando um endereço válido no barramento de endereço (*s_axi_araddr*). Esta alteração de valor faz com que a FSM 1 mude do estado *Waiting_address* para o estado *Waiting_data*.

O estado *Waiting_data* desbloqueia a FSM 3 para a realização da leitura dos dados comprimidos da memória (MIG). Os dados lidos pela FSM 3 são copiados para um buffer (*read_master_buffer*).

Uma vez que a leitura chegou ao fim, os dados devem ser descomprimidos. Então, a FSM 1 transiciona para o estado *Waiting_decompression*. Neste estado, a FSM 2 processa os dados comprimidos lendo-os do *read_master_buffer* e escrevendo a saída em um outro buffer (*read_slave_buffer*). A FSM 1 transiciona para o estado *Response* quando o processo de descompressão terminar, e os dados descomprimidos estiverem no *read_slave_buffer*.

No estado *Response*, a FSM 1 é responsável por copiar os dados descomprimidos do buffer *read_slave_buffer* para a interface AXI, a fim de retornar os dados requisitados pelo processador (PS7).

XMatch - FSM 2

Esta FSM é constituída de 6 estados: (*i*) *Idle*; (*ii*) *Configuring*; (*iii*) *Reading*; (*iv*) *Processing*; (*v*) *Save_data* e (*vi*) *Notify*. A mesma é responsável pelo controle da interface de comunicação com o IP XMatch. A comunicação é realizada seguindo um protocolo proprietário (sessão 3.1).

O processo de descompressão se inicia quando a FSM 1 assume o estado *Waiting_descompression*. Neste ponto, o *read_master_buffer* contém o bloco de dados a ser descomprimido e a FSM 2 pode iniciar o processo de compressão.

A primeira etapa da descompressão é a configuração do IP no modo de operação adequado. Esta etapa é realizada durante o estado *Configuring*, conforme descrito na sessão 3.1. Nesta etapa o sinal *CLEAR* deve receber nível lógico alto enquanto os sinais *CS*, *RW*, *ADDRESS* e *CONTROL* são utilizados para efetuar a configuração do IP no modo de descompressão via banco de registradores.

Após a configuração, o IP está pronto para receber o bloco de dados comprimido. No estado *Reading* os dados são copiados do *read_master_buffer* para o barramento *C_DATAIN*.

No fim do processo de descompressão, que se dá no estado *Processing*, o IP requisita acesso ao barramento *BUS_REQUEST_DU* para escrever o resultado.

A FSM 2 assume o estado *Save_data* para escrever o resultado da descompressão no buffer de dados *read_slave_buffer*[3].

O estado *Notify* (*V*) notifica a FSM 1 de que o dado descomprimido já está pronto para ser retornado ao processador (PS7).

AXI *Master* - FSM 3

Esta FSM é constituída de 4 estados: (*i*) *Idle*; (*ii*) *Set_address*; (*iii*) *Reading* e (*iv*) *Done*. A mesma é responsável pelo controle da interface de comunicação com a controladora de memória (MIG). A comunicação é realizada seguindo o protocolo AMBA AXI4. Primeiramente, o endereço é atribuído, e em seguida os dados são enviados, ambos barramentos com seus conjuntos de sinais de controle, *valid* e *ready*.

A FSM 3 inicia a leitura dos dados comprimidos na memória quando a FSM 1 assume o estado *Waiting_data*. Primeiramente, no estado *Set_address*, é necessário especificar um endereço de leitura através dos sinais *m_axi_araddr* e *m_axi_arvalid*. Quando a MIG responde o reconhecimento ao endereço especificado, através do sinal *m_axi_arready*, a FSM 3 assume o estado *Reading*.

O estado *Reading* é responsável por copiar os dados comprimidos, que são recebidos da controladora de memória (MIG), para o *read_master_buffer*[4]. Após a sinalização do último dado (sinal *m_axi_rlast*) a FSM 3 assume o estado *Done* e em seguida volta para o estado *Idle*.

[3]Buffer responsável por armazenar os dados provenientes do descompressor, que serão retornados ao processador (PS7).

[4]Buffer responsável por armazenar os dados provenientes da controladora de memória (MIG), que serão enviados ao descompressor.

3.2.3 Gerência dos Endereços de Memória

Cada bloco de dados que é enviado para escrita em modo rajada pode ocupar uma região de tamanho diferente na memória do sistema. Implementamos uma tabela de mapeamento, que opera de forma similar a um sistema de gerência de memória, a fim de suportar operações de escrita e leitura de blocos de tamanho variável.

Esta tabela de mapeamento relaciona os índices dos blocos escritos com os endereços de memória físico onde eles se encontram. Além disso, esta tabela também armazena o espaço ocupado por cada bloco. Desta forma ao final de cada escrita registramos internamente um novo bloco a ser endereçado na tabela de mapeamento e o tamanho por ele ocupado.

O processo de leitura consulta a tabela de mapeamento para descobrir em qual endereço de memória se situa o bloco requisitado. O processo de escrita consulta a tabela de mapeamento para descobrir qual o próximo endereço de memória disponível para o próximo bloco.

Assim, evitamos a ocorrência de espaços vazios entre os blocos, ao contrário de uma abordagem que utiliza tamanhos fixos para os blocos. É importante observar que uso desta tabela, ou outro método similar, é requisito para se obter redução do consumo de memória.

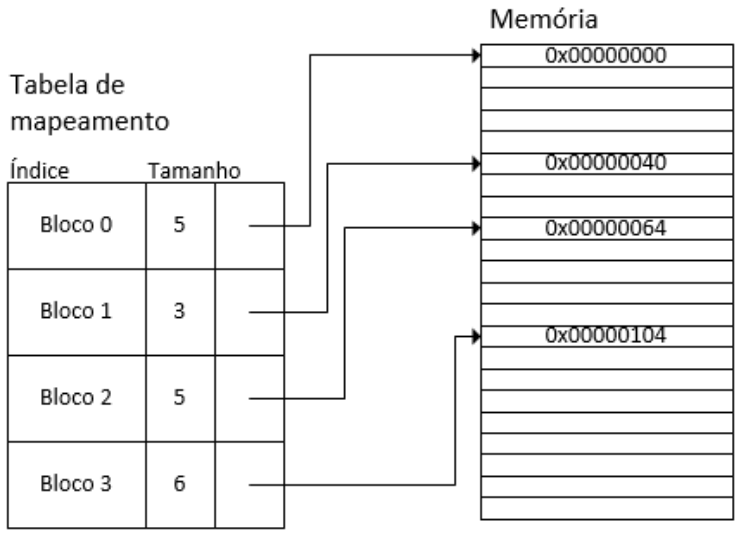

Figura 3.9 – Tabela de mapeamento contendo endereço e tamanho dos blocos.

3.2.4 Limitações da Atual Implementação

Existem duas limitações principais na atual implementação, que não foram resolvidas devido ao tempo limitado de desenvolvimento do trabalho: (*i*) limite do tamanho do bloco nas operações em rajada, e (*ii*) fragmentação externa.

Limite do tamanho do bloco nas operações em rajada

No decorrer do desenvolvimento do trabalho nos deparamos com uma limitação no modo de rajada do protocolo AXI4. Este permite apenas rajadas de tamanho 16, ou seja, se o barramento de dados suporta 32 bits, seríamos limitados a blocos de apenas 64 bytes (equação 3.1).

$$\text{Limite do tamanho da rajada} = \frac{32 \text{ bits} * 16}{8 \text{ bits}} = 64 \text{ bytes} \tag{3.1}$$

Esta limitação pode ser resolvida dividindo as operações maiores em uma sequência de rajadas de tamanho 16, até completar o bloco de tamanho desejado. O tamanho desejado para o bloco poderia ser informado como o primeiro dado a ser transmitido no barramento de dados, e o bloco poderia ser armazenado todo em um *buffer* maior. Desta forma seria possível atingir níveis de compressão maiores, conforme apresentado na sessão 3.1.4.

Fragmentação externa

Outra limitação existente na implementação atual é a fragmentação externa entre os blocos, que ocorre com o particionamento dinâmico, isto é, quando reescrevemos os dados em regiões que já haviam sido alocadas anteriormente com um tamanho diferente.

Existem algoritmos que foram desenvolvidos para solucionar este problema de fragmentação externa [19]. Estes podem mover os blocos para próximos uns dos outros e também escolher a melhor região para alocar os blocos, como: *First-fit* (primeiro que couber); *Best-fit* (melhor que couber) e *Worst-fit* (pior que couber).

4. VALIDAÇÃO DO PROJETO DE INTEGRAÇÃO

Este Capítulo apresenta a validação da proposta, de forma incremental, em 3 projetos desenvolvidos no ambiente Vivado (Xilinx): (*i*) projeto de referência que realiza a comunicação com o barramento AXI4; (*ii*) projeto que integra o barramento AXI4 ao IP de compressão, projeto este descrito em detalhes no Capítulo 3; (*iii*) projeto completo do hardware, com dados da síntese do hardware.

4.1 Projeto de referência com IPs *AXI_MASTER* e *AXI_SLAVE*

A fim de validar o funcionamento do protocolo AXI criou-se um ambiente base, contendo apenas 2 IPs (Figura 4.1): *axi_master* e *axi_slave*. Estes dois IPs são fornecidos pela ferramenta Vivado e implementam o protocolo AXI4 para fins de testes e simulação. Ambos IPs serviram de base para a implementação das interfaces do *wrapper*.

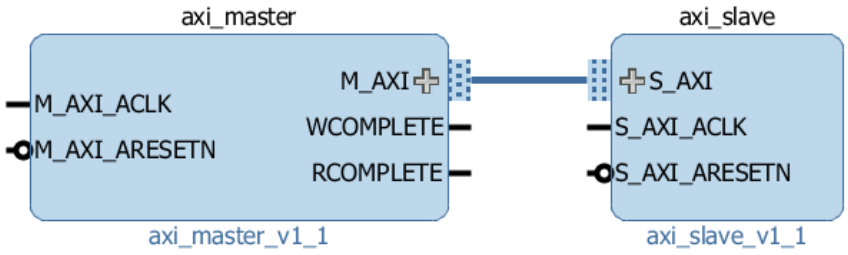

Figura 4.1 – Diagrama de blocos do ambiente base.

A reprodução deste ambiente de simulação pode ser feita através do script TCL apresentado no anexo A.1.

4.1.1 *AXI_MASTER*

Este IP é responsável por iniciar as operações, sejam elas de leitura ou escrita. Ele pode ser configurado através dos parâmetros listados a seguir:

- **C_BURST_LEN** - Configura o tamanho das operações em rajada;
- **C_TRANSACTIONS_NUM** - Configura o número total de operações a ser realizado.

A FSM deste IP implementa quatro estados para auxiliar nos testes de simulação e interpretação das formas de onda:

1. **INIT_COUNTER** - Este estado é responsável por inserir um atraso antes de iniciar as operações;

2. **INIT_WRITE** - Estado em que as operações de escrita são realizadas;
3. **INIT_READ** - Estado em que as operações de leitura são realizadas;
4. **INIT_COMPARE** - Neste estado uma comparação entre o dado lido e o dado esperado é realizada para verificar se houve algum erro no processo.

4.1.2 *AXI_SLAVE*

Este IP é responsável por receber as operações, sejam elas de leitura ou escrita. Ele foi modificado para salvar os dados de operações de escrita em um banco de registradores e também recuperá-los do banco de registradores em operações de leitura.

4.1.3 Execução de Operação de Escrita

Na execução da figura 4.2 podemos observar uma requisição de escrita, por parte do IP *master*.

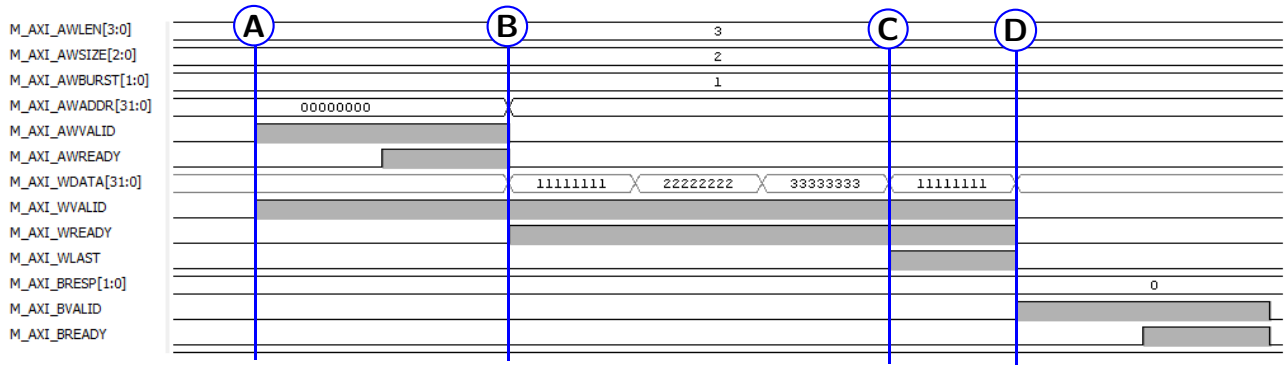

Figura 4.2 – Escrita AXI em ambiente base.

- **(A)** - O sinal *m_axi_awburst* configura a escrita como uma rajada sequencial; *m_axi_awlen* define a quantidade de palavras, neste caso são 4 palavras; enquanto o sinal *m_axi_awsize* configura o tamanho de cada palavra, conforme descrito na sessão 2.2.1, neste caso são 32 bits por palavra; ainda na etapa de configuração o sinal *m_axi_awaddr* define o endereço inicial desta operação de escrita;
- **(B)** - Após a configuração do endereço podemos observar os 4 dados em sequência no barramento *m_axi_wdata*; *m_axi_wvalid* informa que o barramento de dados contém dado válido; o sinal *m_axi_wready* informa que o dado foi capturado corretamente;
- **(C)** - Neste ponto o sinal *m_axi_wlast* informa o último dado da operação em rajada;
- **(D)** - A resposta é recebida através do sinal *m_axi_bresp*; *m_axi_bvalid* informa que o barramento de resposta tem dado válido; o sinal *m_axi_bready* informa que a resposta foi capturada.

4.1.4 Execução de operação de leitura

Na execução da figura 4.3 podemos observar uma requisição de leitura, por parte do IP *master*.

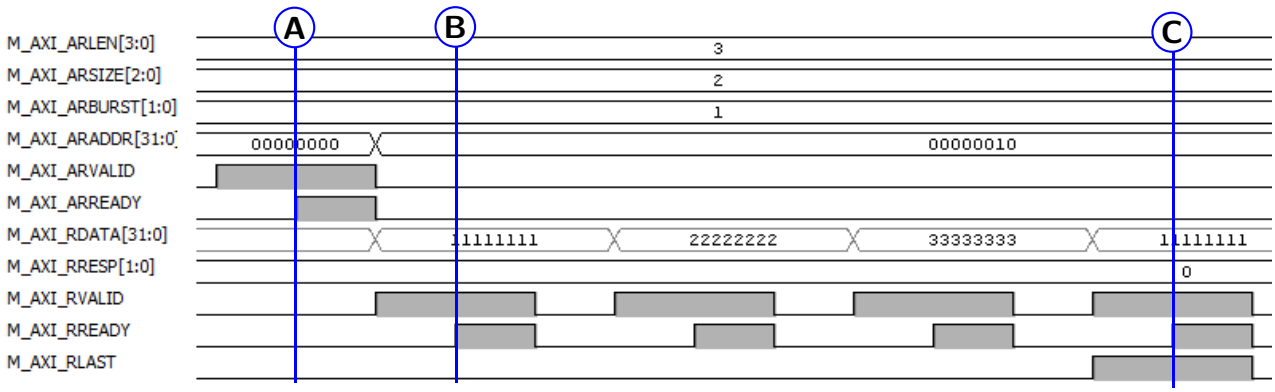

Figura 4.3 – Leitura AXI em ambiente base.

- **(A)** - O sinal *m_axi_arburst* configura a escrita como uma rajada sequencial; *m_axi_arlen* define a quantidade de palavras, neste caso são 4 palavras, enquanto o sinal *m_axi_arsize* configura o tamanho de cada palavra, conforme descrito na sessão 2.2.1, neste caso são 32 bits por palavra; ainda na etapa de configuração o sinal *m_axi_araddr* define o endereço inicial desta operação de leitura;
- **(B)** - Após a configuração do endereço podemos observar os 4 dados em sequência no barramento *m_axi_rdata*; *m_axi_rvalid* informa que o barramento de dados contém dado válido; o sinal *m_axi_rready* informa que o dado foi capturado corretamente;
- **(C)** - O sinal *m_axi_rlast* informa o último dado da operação em rajada; a resposta é recebida através do sinal *m_axi_rresp*, este sinal não possui o controle *valid* e *ready*.

4.1.5 Comparação dos dados

No exemplo de execução da figura 4.4 executamos duas operações de escrita seguidas de leitura e comparação. Como podemos observar, o contador de discrepâncias (*read_mismatch*), que compara o dado lido com o esperado, está zerado (quarto sinal de baixo para cima), mostrando o correto funcionamento do sistema.

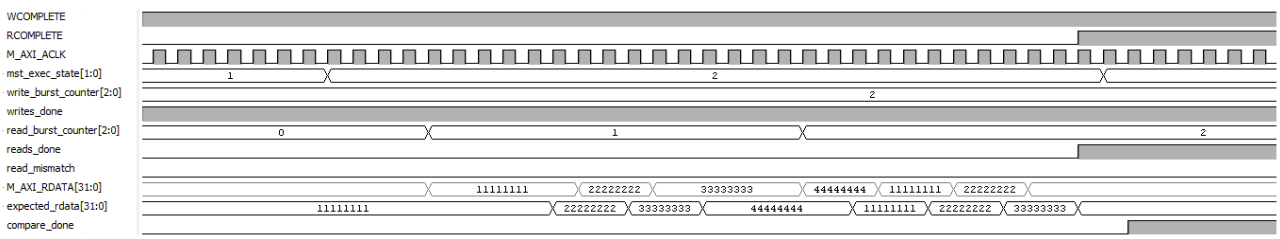

Figura 4.4 – Sinais de auxílio aos testes de simulação em ambiente base.

4.2 Projeto de integração dos IPs AXI com o XMatch

A fim de validar a implementação do *wrapper*, este IP foi inserido no ambiente base de forma que toda a comunicação entre os dois IPs deste ambiente, *axi_master* e *axi_slave*, seja realizada através do *wrapper* (figura 4.5). A implementação das interfaces externas do *wrapper* se basearam no comportamento destes dois IPs, que são fornecidos pela ferramenta Vivado e implementam o protocolo AXI4 para fins de testes e simulação.

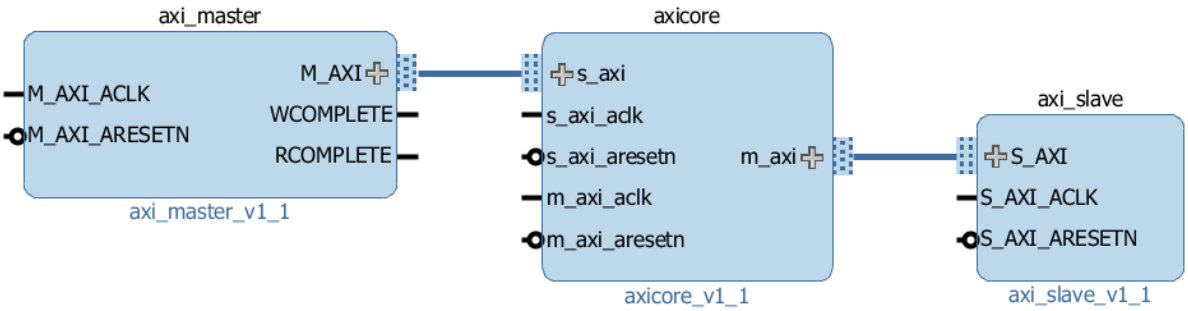

Figura 4.5 – Diagrama de blocos do ambiente com o IP XMatch.

A reprodução deste ambiente de simulação pode ser feita através do script TCL apresentada no anexo A.2

4.2.1 Alterações realizadas no IP *axi_master*

Este IP é responsável por iniciar as operações, sejam elas de leitura ou de escrita. Originalmente o modo *AXI4-Memory Mapped Interface* (2.2.1) endereça a memória byte a byte. Foi realizada uma alteração neste modo de endereçamento para endereçar os índices dos blocos de dados ao invés de endereços de memória (sessão 3.2.3). Com esta alteração é possível gerenciar a área ocupada por cada bloco de dados, já que estes são armazenados em formato comprimido na memória.

4.2.2 Alterações realizadas no IP XMatch

Este IP utiliza muitos sinais internos, implementando um algoritmo de compressão e descompressão complexo. Apesar da proposta deste trabalho de conclusão não contemplar alterações internas no IP XMatch, enfrentamos dificuldade na utilização deste IP que requereram mudanças internas do mesmo.

(*i*) Sinais do processo de descompressão

Ao decorrer do trabalho percebemos que o comportamento de descompressão não se apresentou conforme a documentação disponível. Os sinais de *BUS_REQUEST* e *BUS_ACKNOWLEDGE* não

estavam sendo utilizados para a saída de dados. Ainda neste mesmo contexto, o sinal que notifica dados válidos na saída do descompressor *D_U_DATA_VALID* não estava funcionando de maneira correta, o que dificultou o gerenciamento por parte da máquina de estados do processo de leitura e descompressão.

Para solucionar estes problemas foi necessário compreender o funcionamento interno do IP, e expor o sinal interno *READY* da entidade *OUTPUT_BUFFER* para *D_U_DATA_VALID*. Com este sinal é possível alterar o estado da FSM 2, apresentada na sessão 3.2.2, de *Processing* para *Save_data*.

(*ii*) Memórias internas

O algoritmo implementado pelo XMatch faz uso de cinco memórias internas. São duas memórias para o compressor e três para o descompressor. Estas memórias foram implementadas utilizando uma versão da biblioteca da Xilinx (*XilinxCoreLib*) que não é mais compatível com o Vivado.

A solução que avaliamos mais adequada foi criar a nossa própria memória, implementando a mesma interface para garantir compatibilidade com o XMatch. O código abaixo apresenta como modelamos estas memórias.

```
entity simulation_mem is
        port (
        addra : in std_logic_vector(7 downto 0);
        addrb : in std_logic_vector(7 downto 0);
        clka  : in std_logic;
        clkb  : in std_logic;
        dina  : in std_logic_vector(31 downto 0);
        doutb : out std_logic_vector(31 downto 0);
        enb   : in std_logic;
        wea   : in std_logic);
end simulation_mem;

architecture arch_mem_256 of simulation_mem is

type MEM_256 is array (0 to 255) of std_logic_vector(31 downto 0);
signal mem : MEM_256;

begin

process (clka) -- write process
begin
    if (rising_edge(clka)) then
        if wea = '1' then
            mem(to_integer(unsigned(addra))) <= dina;
        end if;
    end if;
end process;

process (clkb) -- read process
begin
    if (rising_edge(clkb)) then
        if enb = '1' then
            doutb <= mem(to_integer(unsigned(addrb)));
        end if;
    end if;
end process;

end arch_mem_256;
```

O recomendado para prototipação em FPGAs da Xilinx é utilizar BRAMs (Block Random Access Memory) ao invés de registradores para grandes áreas de memória a fim de preservar a área disponível

da FPGA para lógica programável. Conforme projetado, a partir do código HDL desenvolvido (sessão 4.3), a ferramenta Vivado inferiu BRAMs na síntese física, ao invés de utilizar os registradores da FPGA. Desta forma, este mesmo código pode ser utilizado para simulação e prototipação. Outra vantagem de se utilizar esta modelagem é a visibilidade do conteúdo destas memórias. Quando se utiliza um IP, como originalmente feito, a memória se comporta como uma "caixa-preta", sem opção de visualização do seu conteúdo interno.

4.2.3 Execução de operação de escrita

Na execução da Figura 4.6 podemos observar uma requisição de escrita, por parte do IP *master*. Agora o fluxo de escrita inclui o IP XMatch para compressão dos dados.

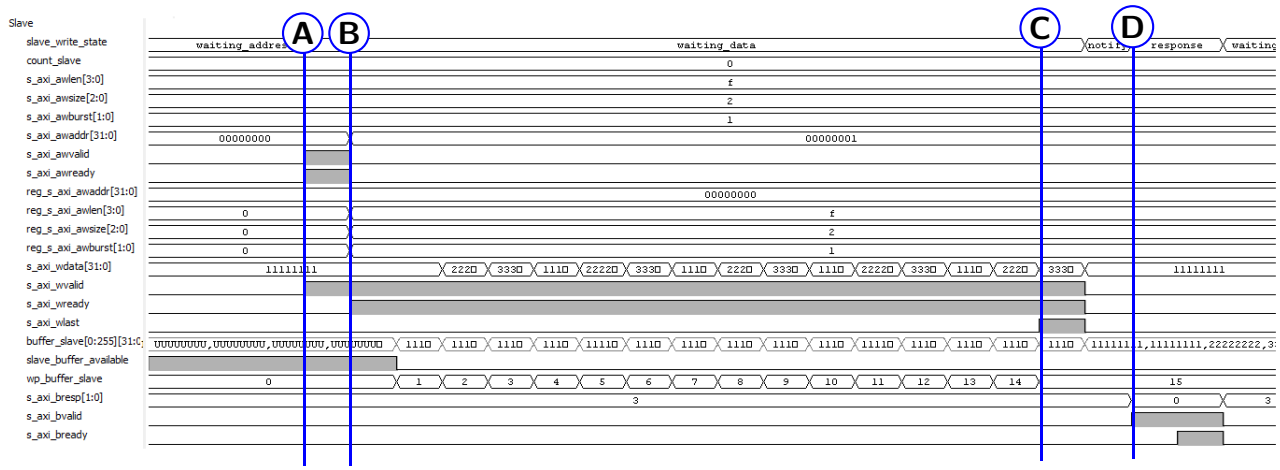

Figura 4.6 – Inicio de escrita AXI em ambiente com o IP XMatch.

- **(A)** - O sinal *m_axi_awburst* configura a escrita como uma rajada sequencial; *m_axi_awlen* define a quantidade de palavras, neste caso são 16 palavras; enquanto o sinal *m_axi_awsize* configura o tamanho de cada palavra, conforme descrito na sessão 2.2.1, neste caso são 32 bits por palavra; ainda na etapa de configuração, através do sinal *m_axi_awaddr*, o primeiro bloco é endereçado para esta operação de escrita;

- **(B)** - Após a configuração do endereço podemos observar a mudança de estado através do sinal *slave_write_state*, que transita do estado *waiting_address* para *waiting_data*. Os 4 dados são escritos em sequência no barramento *m_axi_wdata*; *m_axi_wvalid* informa que o barramento de dados contém dado válido; o sinal *m_axi_wready* informa que o dado foi capturado corretamente;

- **(C)** - Neste ponto o sinal *m_axi_wlast* informa o último dado da operação em rajada; através do sinal *wp_buffer_slave* é possível observar o incremento no índice de escrita do buffer *buffer_slave*, a cada dado válido que é capturado;

- **(D)** - A resposta é enviada ao processador (PS7) através do sinal *m_axi_bresp*; *m_axi_bvalid* informa que o barramento de resposta tem dado válido; o sinal *m_axi_bready* informa que a resposta foi capturada. A resposta da operação de escrita é enviada ao processador após o preenchimento do buffer *wp_buffer_slave*, deixando-o disponível para demais operações.

A figura 4.7 mostra o início da configuração do IP XMatch para compressão dos dados.

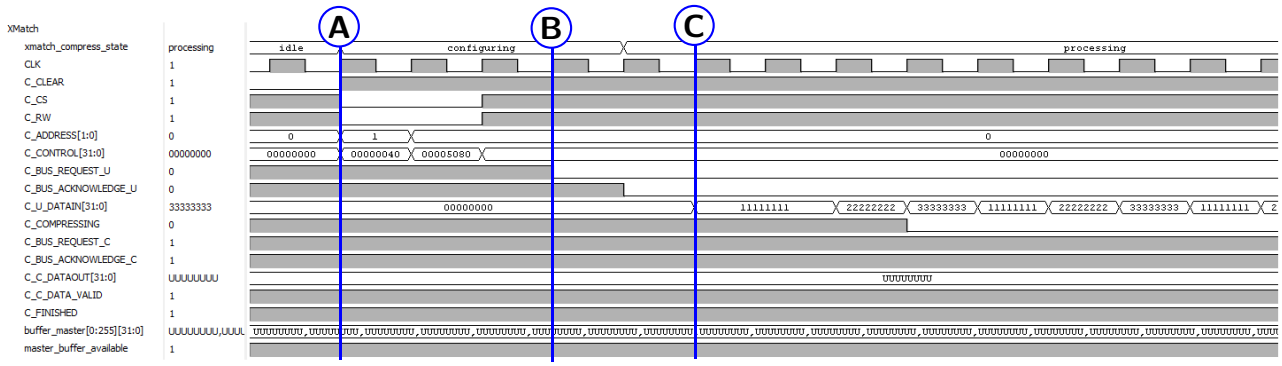

Figura 4.7 – Iniciando a compressão dos dados.

- **(A)** - O IP XMatch é configurado em modo compressão através de dois registradores; o registrador de endereço 01 recebe o valor 0x40, que configura o compressor para receber um bloco de 64 bytes; o registrador 00 recebe o valor 0x5080, que dá início ao processo de compressão conforme sessão 3.1;

- **(B)** - Neste ponto o XMatch solicita os dados expandidos para compressão através do sinal *C_BUS_REQUEST_U*; após responder a requisição, com o sinal *C_BUS_ACKNOWLEDGE_U*, a máquina de estados que controla este IP transiciona do estado *configuring* para *processing*;

- **(C)** - Os dados são enviados em sequência para o compressor através do barramento de dados *C_U_DATAIN* e o processamento dos mesmos se inicia, como podemos observar através do sinal *C_COMPRESSING*.

A figura 4.8 apresenta a conclusão do processo de compressão dos dados.

- **(A)** - Após o término do processo de compressão, sinalizado pelo sinal *C_COMPRESSING*, o IP XMatch solicita acesso ao barramento *C_C_DATAOUT*, através do sinal *C_BUS_REQUEST_C*; em seguida esta requisição é atendida (sinal *C_BUS_ACKNOWLEDGE_C*) e então ocorre a transição do estado *processing* para o estado *writing*, como pode-se observar através do sinal *xmatch_compress_state*;

- **(B)** - Os dados comprimidos são disponibilizados no barramento *C_C_DATAOUT* e o sinal *C_C_DATA_VALID* informa o período em que este barramento possui dados válidos; Neste ponto os dados são copiados em sequência para um banco de registradores (*buffer_master*) e o sinal *master_buffer_available* informa que este não está mais disponível;

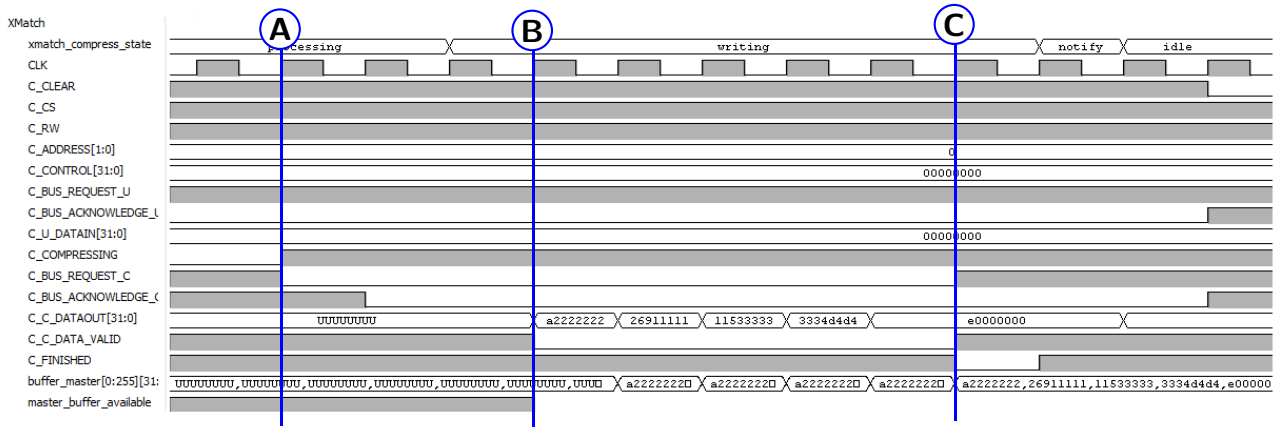

Figura 4.8 – Compressão realizada pelo IP XMatch.

- **(C)** - O sinal *C_FINISHED* informa o final da saída de dados enquanto o sinal *C_C_DATA_VALID* informa que não há mais dado válido no barramento *C_C_DATAOUT*; a máquina de estados então transiciona do estado *Writing* para *Notify*, desta forma a outra máquina de estados, que controla a interface com a controladora de memória, é notificada de que os dados estão disponíveis para escrita.

Na execução da figura 4.9 podemos observar o dado comprimido sendo enviado através do protocolo AXI para a controladora de memória (MIG).

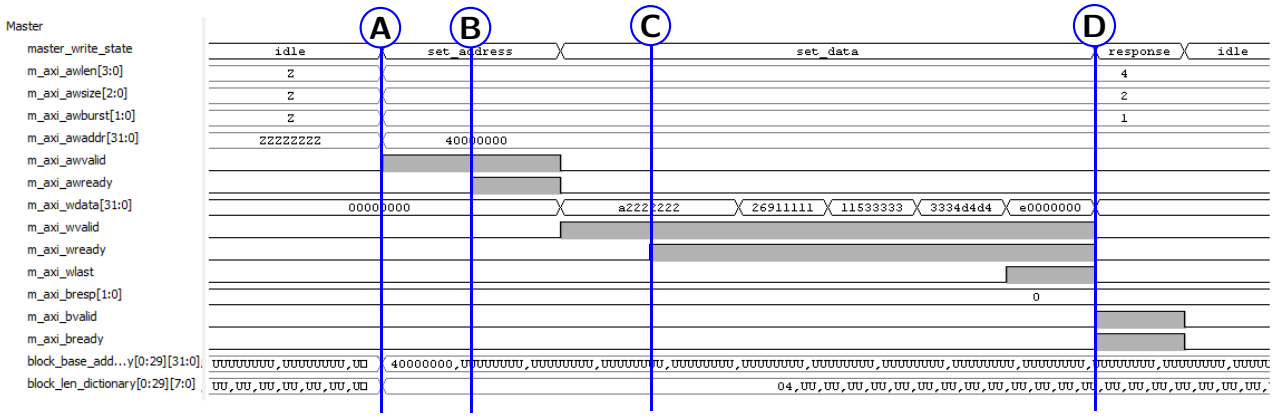

Figura 4.9 – Escrita dos dados comprimidos na memória.

- **(A)** - O endereço de escrita é configurado para a primeira posição de memória através do sinal *m_axi_awaddr*, endereço 0x40000000; o sinal *m_axi_awvalid* informa que o barramento *m_axi_awaddr* possui dado válido;
- **(B)** - O sinal *m_axi_awready* informa que a controladora de memória (MIG) efetuou a leitura do endereço;
- **(C)** - Neste ponto os dados começam a ser enviados em sequência através do barramento *m_axi_wdata*; o sinal *m_axi_wvalid* informa que este barramento possui dados válidos enquanto o sinal *m_axi_wready* informa que a leitura foi efetuada por parte da MIG;

- **(D)** - Após o último dado, informado pelo sinal *m_axi_wlast*, a MIG informa que a operação terminou com sucesso através do sinal *m_axi_bresp*; a leitura da resposta é informada através do sinal *m_axi_bready*.

Ao final da operação de escrita, devido ao processo de compressão, a memória ocupada pelas 16 palavras iniciais (64 bytes) é de 5 palavras (20 bytes). Como o IP XMatch implementa um algoritmo de compressão sem perdas esta informação pode ser revertida ao dado original através do processo de descompressão.

4.2.4 Execução de operação de leitura

Na execução da figura 4.10 podemos observar uma requisição de escrita, por parte do IP *master*, utilizando o IP XMatch para descompressão dos dados. Ao contrário da escrita, a leitura precisa esperar a descompressão finalizar para responder com os dados expandidos ao processador.

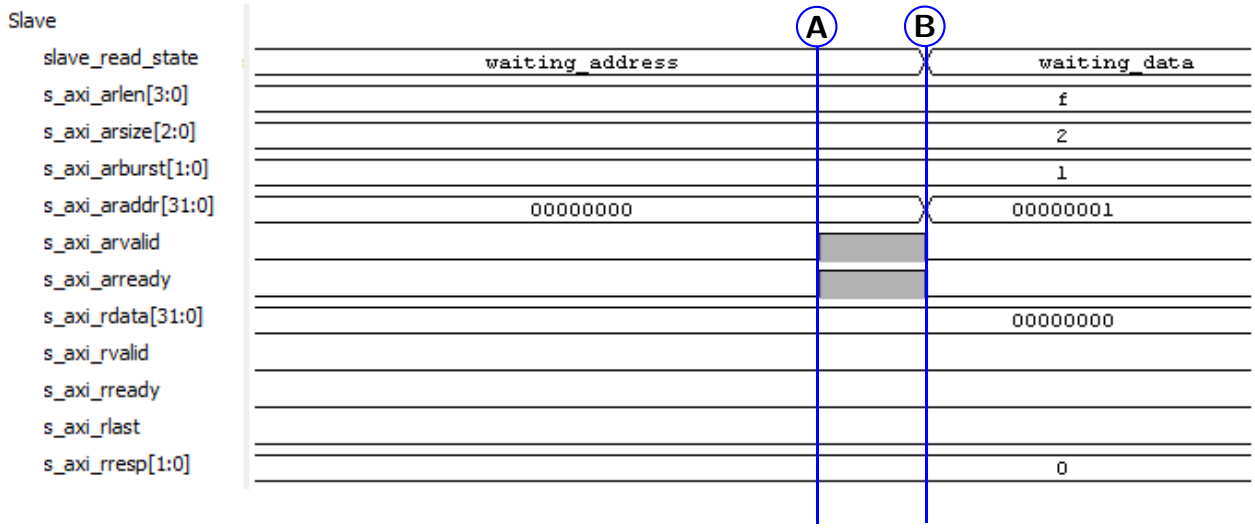

Figura 4.10 – Inicio de leitura AXI em ambiente com o IP XMatch.

- **(A)** - O sinal *m_axi_arburst* configura a leitura como uma rajada sequencial; *m_axi_arlen* define a quantidade de palavras, neste caso são 16 palavras; enquanto o sinal *m_axi_arsize* configura o tamanho de cada palavra, conforme descrito na sessão 2.2.1, neste caso são 32 bits por palavra; ainda na etapa de configuração, através do sinal *m_axi_araddr*, o primeiro bloco é endereçado para esta operação de leitura;

- **(B)** - Neste ponto os dados comprimidos devem ser lidos da memória e em seguida processados pelo descompressor.

Na execução da figura 4.11 podemos observar a requisição de leitura à controladora de memória (MIG), que é realizada para requisitar os dados em formato comprimido.

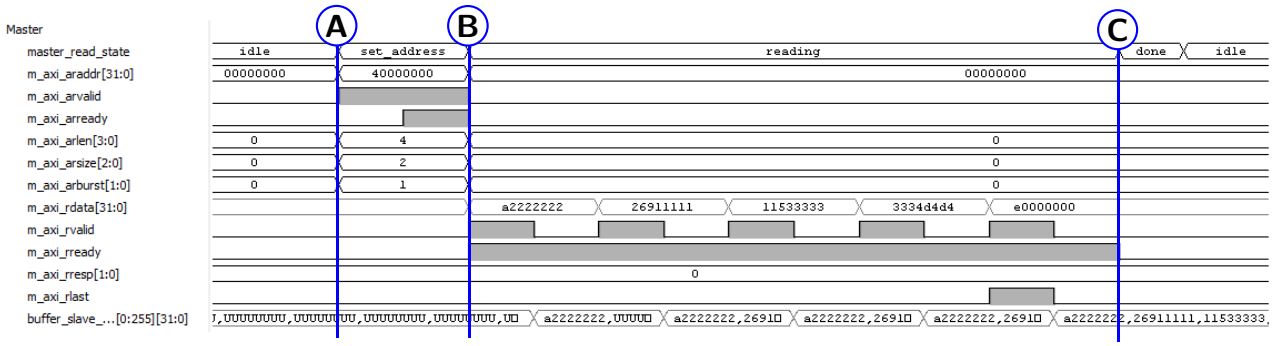

Figura 4.11 – Leitura dos dados comprimidos.

- **(A)** - Endereço de leitura configurado, através do sinal *m_axi_araddr*, para a posição de memória em que o bloco foi escrito (0x40000000); o sinal *m_axi_arvalid* informa que o endereço do barramento *m_axi_araddr* é valido enquanto o sinal *m_axi_arready* informa a leitura deste barramento por parte da controladora de memória (MIG);

- **(B)** - Neste ponto se inicia a transferência dos dados, provenientes da memória, através do barramento *m_axi_rdata* para o *buffer_slave*; o sinal *m_axi_rvalid* informa a existência de dado válido no barramento *m_axi_rdata* enquanto o sinal *m_axi_rready* informa a leitura deste barramento;

- **(C)** - Fim da operação de leitura, todos os dados já foram transferidos para o *buffer_slave* e podem ser utilizados pelo descompressor; o estado *Done*, observado no sinal *master_read_state* informa o compressor de que os dados estão disponíveis.

Na execução da figura 4.12 o IP XMatch inicia o processo de descompressão.

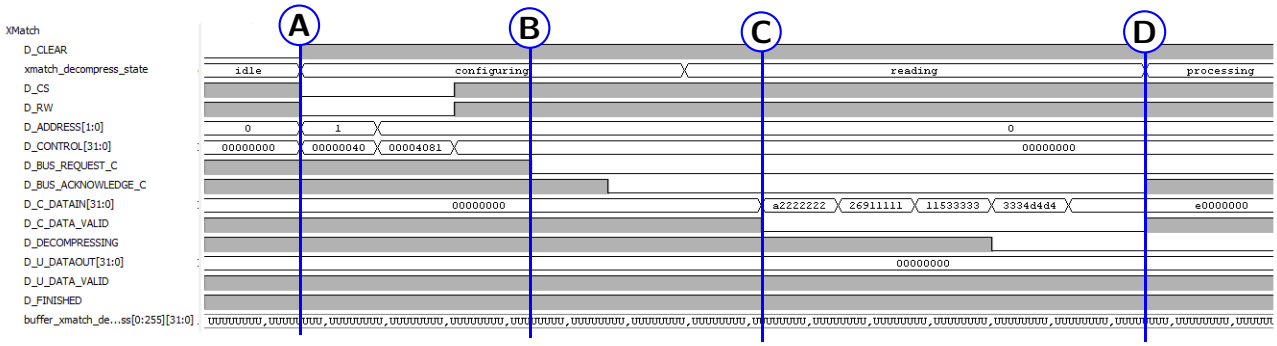

Figura 4.12 – Iniciando a descompressão dos dados.

- **(A)** - O IP XMatch transiciona de um estado ocioso (*Idle*) para o estado de configuração em modo de descompressão; o registrador de endereço 01 recebe o valor 0x40, que configura o compressor para receber um bloco de 64 bytes; o registrador 00 recebe o valor 0x4080, que da início ao processo de descompressão conforme sessão 3.1;

- **(B)** - Neste ponto o XMatch solicita os dados em formato comprimido para descompressão, através do sinal *D_BUS_REQUEST_C*; após responder à requisição, com o sinal *D_BUS_ACKNOWLEDGE_C*, a máquina de estados que controla este IP transiciona do estado *Configuring* para *Reading*;

- **(C)** - Os dados são enviados em sequência para o descompressor através do barramento de dados *D_C_DATAIN* e o processamento dos mesmos se inicia, como podemos observar através do sinal *D_DECOMPRESSING*; durante o envio dos dados o sinal *D_C_DATA_VALID* foi utilizado para informar a presença de dado válido no barramento de entrada *D_C_DATAIN*;

- **(D)** - Ao fim do envio de dados o sinal *D_C_DATA_VALID* volta ao nível lógico alto.

Na execução da figura 4.13 o IP XMatch finaliza o processo de descompressão.

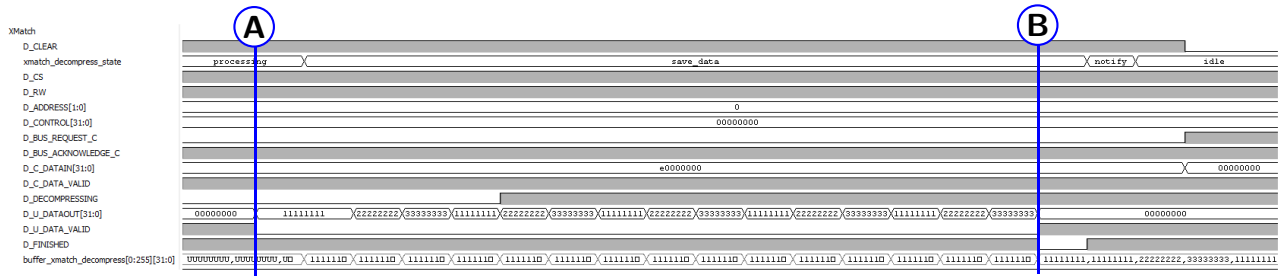

Figura 4.13 – Descompressão realizada pelo IP XMatch.

- **(A)** - O sinal *D_U_DATAVALID* informa a presença de dados válidos no barramento de saída *D_U_DATAOUT*; estes dados são copiados para o banco de registradores *buffer_xmatch_decompressor*;

- **(B)** - Ao fim da transferência de dados o sinal *D_U_DATAVALID* volta ao nível lógico alto e a máquina de estados assume o estado *Notify*, que notifica a máquina de estados responsável pela interface com o processador de que os dados estão prontos para a resposta.

Na execução da figura 4.14 os dados descomprimidos são enviados ao processador (PS7) e a operação de leitura é encerrada.

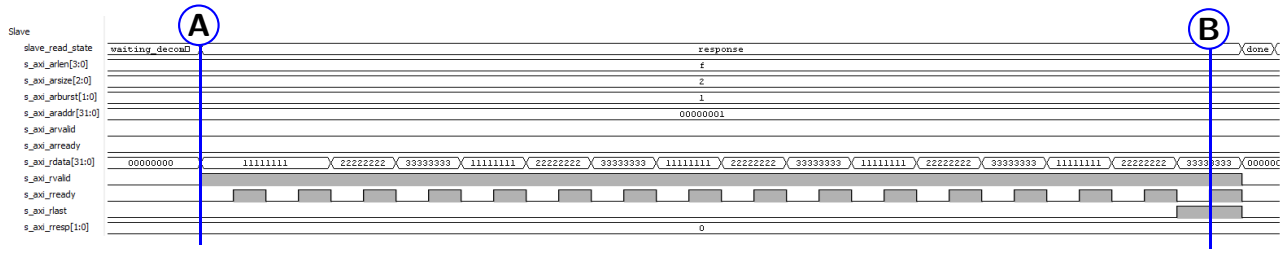

Figura 4.14 – Resposta ao processador com dados descomprimidos.

- **(A)** - Os dados são transferidos do banco de registradores *buffer_xmatch_decompressor* para o processador (PS7), através do barramento *s_axi_rdata*; o sinal *s_axi_rvalid* informa a presença de dado válido no barramento *s_axi_rdata* enquanto o sinal *s_axi_rready* informa a leitura por parte do processador;
- **(B)** - Sinal *s_axi_rlast* informa o último dado que deverá ser respondido ao processador nesta operação através do barramento *s_axi_rdata*; após a transferência a máquina de estados assume o estado *Done*, e após *Waiting_address*, onde estará pronta para a próxima operação.

4.2.5 Comparação dos dados

Na execução da figura 4.15 podemos observar que o contador de discrepâncias (*read_mismatch*), que compara os dados lidos com os esperados está zerado, demostrando o correto funcionamento do sistema.

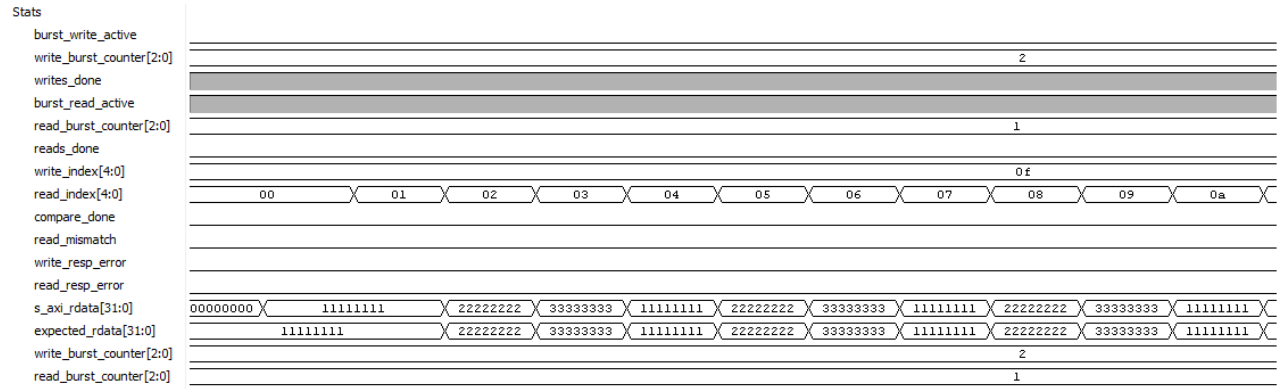

Figura 4.15 – Sinais de auxílio aos testes de simulação em ambiente com XMatch.

4.3 Projeto de hardware

Iniciamos o projeto de hardware com a criação de um script TCL (anexo A.3), que instancia o processador ARM Cortex-A9 (PS7) no lugar do IP *axi_master*, e a controladora de memória DDR (MIG) no lugar do IP *axi_slave*. O nosso IP, que implementa a interface AMBA AXI4 para o XMatch, aqui chamado de *axicore*, foi posicionado entre o processador e a controladora de memória, de forma a interceptar todas as operações de leitura e escrita, como podemos observar no diagrama de blocos da figura 4.16.

Devido ao tempo limitado deste trabalho de conclusão não foi possível concluir a parte de prototipação. Entretanto, foi possível extrair resultados de área que podem ser observados na figura 4.17. Estes resultados mostram dois fatos significativos: (*i*) o consumo de recursos lógicos (LUTs e FFs) é baixo, correspondendo a 9.63% e 5.04% dos recursos disponíveis, respectivamente; (*ii*) a inferência de memória foi correta, com o uso de 5 BRAMs, mostrando que não é necessária a alteração do código desenvolvido para fins de síntese no FPGA.

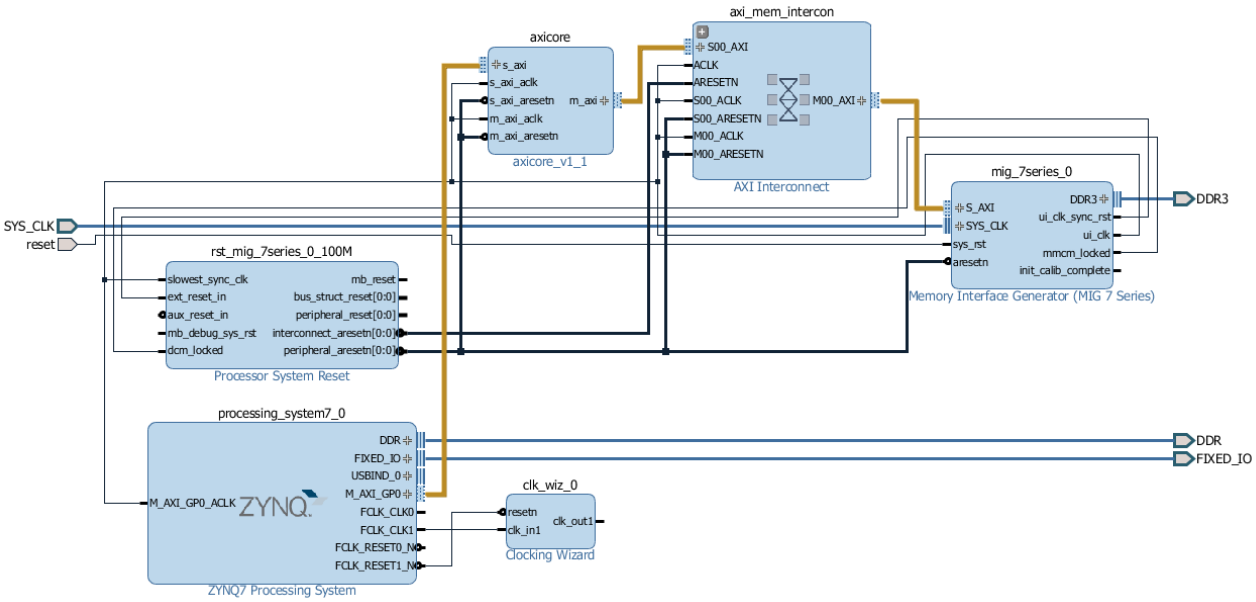

Figura 4.16 – Diagrama de blocos do projeto de hardware.

Resource	Utilization	Available	Utilization %
LUT	21042	218600	9.63
LUTRAM	2757	70400	3.92
FF	22049	437200	5.04
BRAM	5	545	0.92
IO	117	362	32.32
MMCM	2	8	25.00
PLL	1	8	12.50

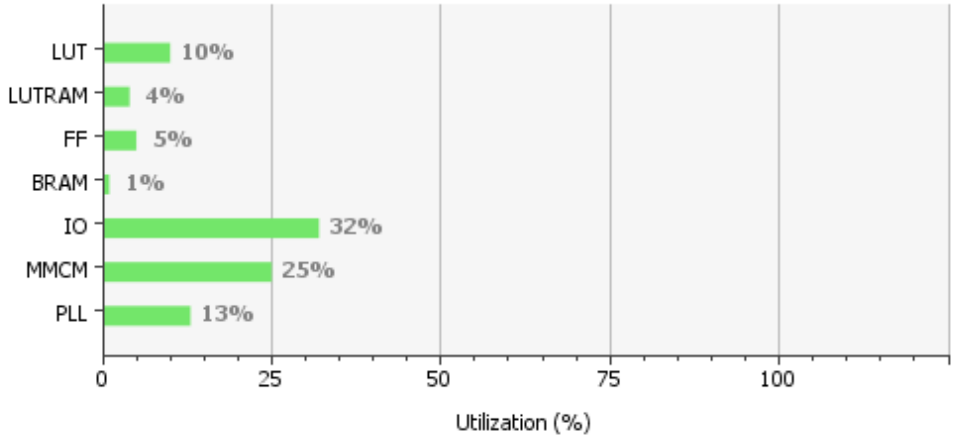

Figura 4.17 – Utilização dos recursos de hardware disponíveis na FPGA.

O algoritmo implementado pelo XMatch faz uso de cinco memórias internas, conforme apresentado nas figuras 4.17 e 4.18. O compressor utiliza duas memórias: (*i*) memória para compor o dicionário de codificação e (*ii*) memória para entrada de dados. O descompressor utiliza internamente três memórias: (*i*) memória para compor o dicionário de decodificação; (*ii*) memória auxiliar e (*iii*) memória para saída de dados. Os *buffers* que mantém os dados de entrada e saída do compressor

ainda fazem uso de banco de registradores (Sessão 3.2).

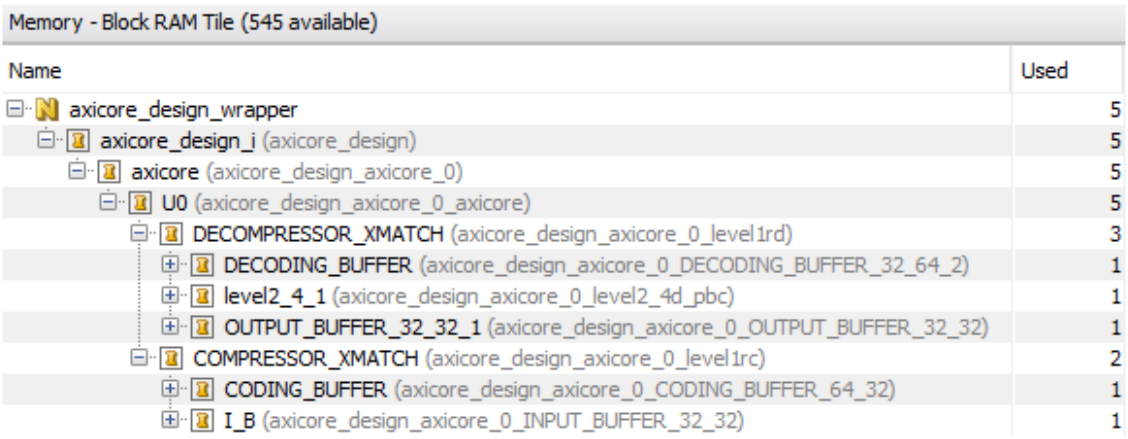

Figura 4.18 – BRAMs inferidas durante a síntese física pela ferramenta Vivado.

A Figura 4.19 apresenta a planta-baixa do projeto após a síntese física. As cores destacadas na planta-baixa apresentam a área ocupada por cada módulo.

- A cor laranja representa a área ocupada pelo processador, este é um processador físico (ARM Cortex-A9) e encontra-se fora da área de lógica programável.
- A cor vermelha representa a área utilizada pelo IP axicore (*wrapper*), este IP contém toda a lógica de controle envolvida no processo de leitura e escrita, além de bancos de registradores para armazenamento da tabela de mapeamento de endereços, *buffers* de entrada e saída das interfaces com o IP XMatch.
- A cor verde representa a área utilizado pela parte do XMatch responsável pela descompressão. Podemos ainda observar pequenos retângulos azuis, que representam as BRAMs utilizadas internamente.
- A cor amarela representa a área utilizada pela parte do XMatch responsável pela compressão. Podemos observar também pequenos retângulos azuis, que representam as BRAMs utilizadas internamente.
- A cor rosa representa a controladora de memória (MIG), esta ocupa a maior parte da área utilizada na FPGA.

Para conclusão do projeto de hardware deve-se ainda realizar a simulação com atrasos, e posteriormente testar o funcionamento na placa de prototipação.

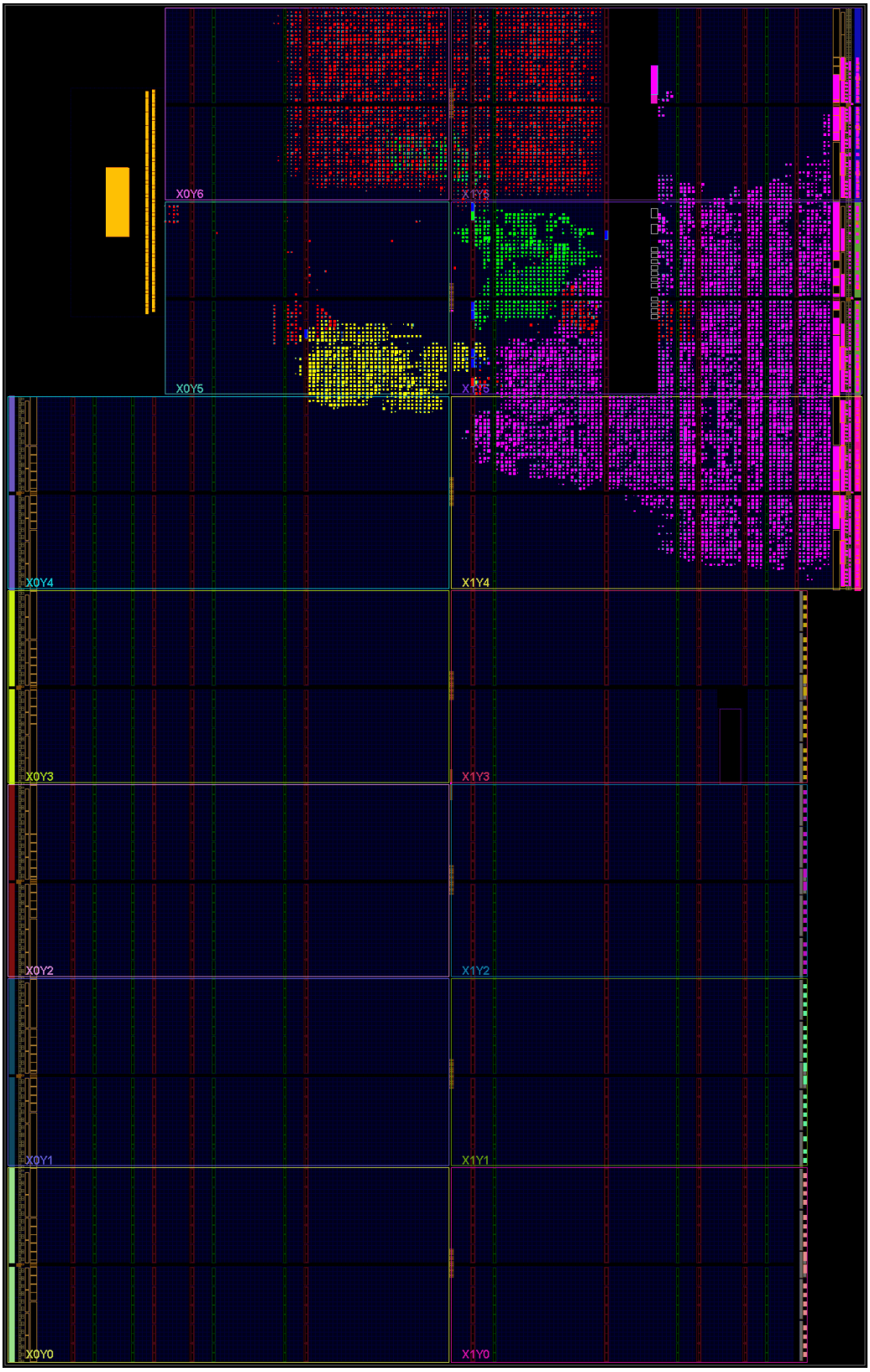

Figura 4.19 – Planta-baixa do projeto após a síntese física.

5. CONCLUSÕES E TRABALHOS FUTUROS

O presente trabalho de conclusão apresenta o projeto de desenvolvimento de um sistema embarcado, capaz de interceptar requisições de acesso à memória oriundas de um processador, comprimindo seus dados para redução dos requisitos de memória. A compressão de dados é realizada por um IP de terceiros. Todavia, destaca-se que a utilização de IPs pré-validados e documentados não significa uma integração direta. Problemas de incompatibilidades entre diferentes módulos de hardware tornam necessário o seu entendimento, além de alterações no funcionamento interno. Este foi um fator de grande dispêndio de energia neste trabalho de conclusão.

Apesar do tempo restrito de desenvolvimento deste projeto foi possível criar um ambiente de simulação para validá-lo, e ainda dar início ao processo de síntese física. Este trabalho contribuiu para aprofundar os conhecimentos de projeto de sistemas digitais, barramentos padronizados de larga utilização na indústria (AMBA AXI4), e aprendizado de novos conceitos, como compressão de dados em hardware.

Além destas contribuições, o presente trabalho também promoveu o aprendizado do ambiente de desenvolvimento de sistemas compostos por processador e módulos de hardware descritos em VHDL (Vivado). O conjunto de scripts TCL, disponível nos anexos deste documento, poderão servir como ponto de partida para projetos futuros que desejem integrar hardware e software.

Como extensão do projeto aqui apresentado sugere-se a finalização da etapa de síntese física e validação no hardware. Sugere-se também a implementação da solução proposta para eliminar a limitação de tamanho de bloco de compressão. Sugere-se ainda como trabalhos futuros o estudo e integração de outros IPs. Estes podem integrar a solução atual, implementando outros algoritmos de compressão, como por exemplo, com perdas, para áudio, imagens e vídeos, ou então estender a solução atual para diferentes aplicações, como criptografia de dados em memória, já que segurança da informação é um tema altamente abordado em sistemas embarcados.

Referências Bibliográficas

[1] J. L. Nunez. Xmatchpro lossless data compressor. 2006. Capturado em Agosto 2017.

[2] L. Benini; A. Macii; E. Macii; M. Poncino. Selective Instruction Compression for Memory Energy Reduction in Embedded Systems. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pages 206–211, 1999.

[3] Intel Corporation; Micron Technology Inc. Intel and Micron Produce Breakthrough Memory Technology. 2015. Capturado em: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology, Agosto 2017.

[4] P. R. Panda; F. Catthoor; N. D. Dutt; K. Danckaert; E. Brockmeyer; C. Kulkarni; A. Vandercappelle; P. G. Kjeldsberg. Data and Memory Optimization Techniques for Embedded Systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2):149–206.

[5] D. Salomon. *Data Compression: The Complete Reference*. Springer New York, 2004.

[6] N. A. Sherwani. *Algorithms for VLSI Physical Design Automation*. Springer US, 2007.

[7] Xilinx Inc. AXI Reference Guide. 2011. Capturado em Agosto 2017.

[8] Xilinx Inc. ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC User Guide. 2016. Capturado em Agosto 2017.

[9] Xilinx Inc. Zynq-7000 All Programmable SoC Data Sheet: Overview. 2017. Capturado em Agosto 2017.

[10] Xilinx Inc. Zynq-7000 AP SoC and 7 Series Devices Memory Interface Solutions v4.2. 2017. Capturado em Agosto 2017.

[11] S. Pasricha; N. Dutt. *On-Chip Communication Architectures: System on Chip Interconnect*. Elsevier Science, 2010.

[12] V. L. Lorenzo. JPEG Hardware Compressor. 2005. Capturado em Agosto 2017.

[13] J. L. Nunez; C. Feregrino; S. Bateman; S. Jones. The X-MatchLITE FPGA-based Data Compressor. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, pages 255–, New York, NY, USA, 1999. ACM.

[14] J. L. Nunez and S. Jones. Lossless data compression programmable hardware for high-speed data networks. In *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.*, pages 290–293, Dec 2002.

[15] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[16] T. C. Bell; J. G. Cleary; I. H. Witten. *Text Compression*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[17] P. Sun; J. L. Nunez. Optimizing Memory Power in Hybrid ARM-FPGA Chips With Lossless Data Compression. In *Proceedings of the FPGA World Conference 2014*, FPGAWorld '14, pages 2:1–2:7, New York, NY, USA, 2014. ACM.

[18] R. Arnold; T. C. Bell. A corpus for the evaluation of lossless compression algorithms. In *Data Compression Conference, 1997. DCC '97. Proceedings*, pages 201–210, Mar 1997.

[19] S. Debasis. *Classic Data Structures, 2nd ed.* Prentice-Hall Of India Pvt. Limited, 2008.

A. APÊNDICE: SCRIPTS TCL

Este apêndice apresenta os *scripts* TCL desenvolvidos para a criação dos ambientes de validação, descritos no Capítulo 4.

A.1 TCL projeto de referência AXI

```
source ./create_package_axi_master.tcl
source ./create_package_axi_slave.tcl

file delete -force base_simulation_project

create_project base_simulation_project base_simulation_project -part
 ↪ xc7z045ffg900-2 -force
set_property board_part xilinx.com:zc706:part0:1.0 [current_project]
import_files -force -norecurse
update_compile_order -fileset sim_1
set_property ip_repo_paths ip_repo [current_fileset]
update_ip_catalog

update_ip_catalog -add_ip ip_repo/pucrs_user_axi_master.zip -repo_path
 ↪ ip_repo
update_ip_catalog -add_ip ip_repo/pucrs_user_axi_slave.zip -repo_path
 ↪ ip_repo
create_bd_design "base_design"

startgroup
create_bd_cell -type ip -vlnv pucrs:user:axi_master axi_master
endgroup
startgroup
create_bd_cell -type ip -vlnv pucrs:user:axi_slave axi_slave
endgroup
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:sim_clk_gen:1.0 sim_clk_gen_0
endgroup

connect_bd_net [get_bd_pins sim_clk_gen_0/clk] [get_bd_pins
 ↪ axi_slave/S_AXI_ACLK]
connect_bd_net [get_bd_pins sim_clk_gen_0/clk] [get_bd_pins
 ↪ axi_master/M_AXI_ACLK]
connect_bd_net [get_bd_pins sim_clk_gen_0/sync_rst] [get_bd_pins
 ↪ axi_master/M_AXI_ARESETN]
connect_bd_net [get_bd_pins sim_clk_gen_0/sync_rst] [get_bd_pins
 ↪ axi_slave/S_AXI_ARESETN]
connect_bd_intf_net [get_bd_intf_pins axi_master/M_AXI] [get_bd_intf_pins
 ↪ axi_slave/S_AXI]
regenerate_bd_layout
regenerate_bd_layout -routing
```

```
assign_bd_address [get_bd_addr_segs {axi_slave/S_AXI/reg0 }]
set_property range 1G [get_bd_addr_segs
 ↪ {axi_master/M_AXI/SEG_axi_slave_reg0}]

validate_bd_design

make_wrapper -files [get_files
 ↪  base_simulation_project/base_simulation_project.srcs/sources_1/bd/base_design/b
 ↪ -top
add_files -norecurse
 ↪  base_simulation_project/base_simulation_project.srcs/sources_1/bd/base_design/h
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
```

A.2 TCL projeto de integração AXI com XMatch

```
source ./create_package_axi_master.tcl
source ./create_package_axi_slave.tcl
source ./create_package_axicore.tcl

file delete -force axicore_simulation_project

create_project axicore_simulation_project axicore_simulation_project
 ↪ -part xc7z045ffg900-2 -force
set_property board_part xilinx.com:zc706:part0:1.0 [current_project]
add_files -fileset sim_1 -norecurse ../../src/axicore_design_wrapper_tb.v
import_files -force -norecurse
update_compile_order -fileset sim_1
set_property ip_repo_paths ip_repo [current_fileset]
update_ip_catalog

file delete -force ip_repo/pucrs_user_axi_master
file delete -force ip_repo/pucrs_user_axi_slave
file delete -force ip_repo/pucrs_user_axicore

update_ip_catalog -add_ip ip_repo/pucrs_user_axi_master.zip -repo_path
 ↪  ip_repo
update_ip_catalog -add_ip ip_repo/pucrs_user_axi_slave.zip -repo_path
 ↪  ip_repo
update_ip_catalog -add_ip ip_repo/pucrs_user_axicore.zip -repo_path
 ↪  ip_repo
create_bd_design "axicore_design"

startgroup
create_bd_cell -type ip -vlnv pucrs:user:axi_master axi_master
endgroup
startgroup
create_bd_cell -type ip -vlnv pucrs:user:axi_slave axi_slave
endgroup
startgroup
```

```
create_bd_cell -type ip -vlnv pucrs:user:axicore axicore
endgroup

startgroup
create_bd_port -dir I -type clk ACLK
set_property CONFIG.FREQ_HZ 100000000 [ get_bd_ports /ACLK]
endgroup
create_bd_port -dir I -type rst ARESETN
connect_bd_net [get_bd_ports /ACLK] [get_bd_pins /axi_master/M_AXI_ACLK]
connect_bd_net [get_bd_ports /ARESETN] [get_bd_pins
 ↪  /axi_master/M_AXI_ARESETN]
connect_bd_net [get_bd_ports /ACLK] [get_bd_pins /axi_slave/S_AXI_ACLK]
connect_bd_net [get_bd_ports /ARESETN] [get_bd_pins
 ↪  /axi_slave/S_AXI_ARESETN]

connect_bd_net -net [get_bd_net /aclk_1] [get_bd_ports /ACLK]
 ↪  [get_bd_pins /axicore/s_axi_aclk]
connect_bd_net -net [get_bd_net /aresetn_1] [get_bd_ports /ARESETN]
 ↪  [get_bd_pins /axicore/s_axi_aresetn]
connect_bd_net -net [get_bd_net /aclk_1] [get_bd_ports /ACLK]
 ↪  [get_bd_pins /axicore/m_axi_aclk]
connect_bd_net -net [get_bd_net /aresetn_1] [get_bd_ports /ARESETN]
 ↪  [get_bd_pins /axicore/m_axi_aresetn]

connect_bd_intf_net [get_bd_intf_pins /axi_master/M_AXI]
 ↪  [get_bd_intf_pins /axicore/s_axi]
connect_bd_intf_net [get_bd_intf_pins /axicore/m_axi] [get_bd_intf_pins
 ↪  /axi_slave/S_AXI]

regenerate_bd_layout
assign_bd_address
set_property range 1G [get_bd_addr_segs
 ↪  {axi_master/M_AXI/SEG_axicore_reg0}]
set_property offset 0x00000000 [get_bd_addr_segs
 ↪  {axi_master/M_AXI/SEG_axicore_reg0}]
set_property range 1G [get_bd_addr_segs
 ↪  {axicore/m_axi/SEG_axi_slave_reg0}]
set_property offset 0x00000000 [get_bd_addr_segs
 ↪  {axicore/m_axi/SEG_axi_slave_reg0}]

validate_bd_design
save_bd_design

make_wrapper -files [get_files
 ↪  axicore_simulation_project/axicore_simulation_project.srcs/sources_1/bd/axicore
 ↪  -top

update_compile_order -fileset sources_1
set_property runtime {} [get_filesets sim_1]
launch_xsim -simset sim_1 -mode behavioral
```

```
open_wave_config {axicore_simulation_project_behav.wcfg}
run 40 us
```

A.3 TCL projeto de hardware

```
file delete -force axicore_prototype_project

source ./create_package_axicore.tcl

create_project axicore_prototype_project axicore_prototype_project -part
 ↪  xc7z045ffg900-2 -force
set_property board_part xilinx.com:zc706:part0:1.0 [current_project]
add_files -fileset sim_1 -norecurse ../../src/axicore_design_wrapper_tb.v
import_files -force -norecurse
update_compile_order -fileset sim_1
set_property ip_repo_paths ip_repo [current_fileset]
update_ip_catalog

file delete -force ip_repo/pucrs_user_axicore

update_ip_catalog -add_ip ip_repo/pucrs_user_axicore.zip -repo_path
 ↪  ip_repo
create_bd_design "axicore_design"

# Interface ports
set DDR [ create_bd_intf_port -mode Master -vlnv
 ↪  xilinx.com:interface:ddrx_rtl:1.0 DDR ]
set FIXED_IO [ create_bd_intf_port -mode Master -vlnv
 ↪  xilinx.com:display_processing_system7:fixedio_rtl:1.0 FIXED_IO ]
set DDR3 [ create_bd_intf_port -mode Master -vlnv
 ↪  xilinx.com:interface:ddrx_rtl:1.0 DDR3 ]
set SYS_CLK [ create_bd_intf_port -mode Slave -vlnv
 ↪  xilinx.com:interface:diff_clock_rtl:1.0 SYS_CLK ]

# clk and rst
set clk_wiz_0 [ create_bd_cell -type ip -vlnv xilinx.com:ip:clk_wiz:5.3
 ↪  clk_wiz_0 ]
set_property -dict [ list CONFIG.CLKOUT1_REQUESTED_OUT_FREQ {12.29}
 ↪  CONFIG.RESET_TYPE {ACTIVE_LOW} CONFIG.USE_LOCKED {false}  ]
 ↪  $clk_wiz_0
set rst_mig_7series_0_100M [ create_bd_cell -type ip -vlnv
 ↪  xilinx.com:ip:proc_sys_reset:5.0 rst_mig_7series_0_100M ]
set reset [ create_bd_port -dir I -type rst reset ]
set_property -dict [ list CONFIG.POLARITY {ACTIVE_HIGH}  ] $reset

# ips
set processing_system7_0 [ create_bd_cell -type ip -vlnv
 ↪  xilinx.com:ip:processing_system7:5.5 processing_system7_0 ]
```

```
set_property -dict [ list CONFIG.PCW_EN_CLK1_PORT {1}
 ↪  CONFIG.PCW_EN_RST1_PORT {1} CONFIG.PCW_FPGA0_PERIPHERAL_FREQMHZ {100}
 ↪  CONFIG.PCW_FPGA1_PERIPHERAL_FREQMHZ {200}
 ↪  CONFIG.PCW_TTC0_PERIPHERAL_ENABLE {0} CONFIG.PCW_USE_DMA0 {0}
 ↪  CONFIG.PCW_USE_S_AXI_HP0 {0} CONFIG.PCW_APU_PERIPHERAL_FREQMHZ {400}
 ↪  CONFIG.PCW_APU_CLK_RATIO_ENABLE {4:2:1} CONFIG.preset {ZC706*}  ]
 ↪  $processing_system7_0
set mig_7series_0 [ create_bd_cell -type ip -vlnv
 ↪  xilinx.com:ip:mig_7series:4.0 mig_7series_0 ]
set str_mig_folder [get_property IP_DIR [ get_ips [ get_property
 ↪  CONFIG.Component_Name $mig_7series_0 ] ] ]
file copy mig_b.prj ${str_mig_folder}/mig_b.prj
set_property -dict [ list CONFIG.BOARD_MIG_PARAM {Custom}
 ↪  CONFIG.MIG_DONT_TOUCH_PARAM {Custom} CONFIG.RESET_BOARD_INTERFACE
 ↪ {reset} CONFIG.XML_INPUT_FILE {mig_b.prj}  ] $mig_7series_0

# general connections
set axi_mem_intercon [ create_bd_cell -type ip -vlnv
 ↪  xilinx.com:ip:axi_interconnect:2.1 axi_mem_intercon ]
set_property -dict [ list CONFIG.NUM_MI {1}  ] $axi_mem_intercon
connect_bd_net -net mig_7series_0_mmcm_locked [get_bd_pins
 ↪  mig_7series_0/mmcm_locked] [get_bd_pins
 ↪  rst_mig_7series_0_100M/dcm_locked]
connect_bd_net -net mig_7series_0_ui_clk [get_bd_pins
 ↪  axi_mem_intercon/ACLK] [get_bd_pins axi_mem_intercon/M00_ACLK]
 ↪  [get_bd_pins axi_mem_intercon/S00_ACLK] [get_bd_pins
 ↪  mig_7series_0/ui_clk] [get_bd_pins
 ↪  processing_system7_0/M_AXI_GP0_ACLK] [get_bd_pins
 ↪  rst_mig_7series_0_100M/slowest_sync_clk]
connect_bd_net -net mig_7series_0_ui_clk_sync_rst [get_bd_pins
 ↪  mig_7series_0/ui_clk_sync_rst] [get_bd_pins
 ↪  rst_mig_7series_0_100M/ext_reset_in]
connect_bd_net -net processing_system7_0_FCLK_CLK1 [get_bd_pins
 ↪  clk_wiz_0/clk_in1] [get_bd_pins processing_system7_0/FCLK_CLK1]
connect_bd_net -net processing_system7_0_FCLK_RESET1_N [get_bd_pins
 ↪  clk_wiz_0/resetn] [get_bd_pins processing_system7_0/FCLK_RESET1_N]
connect_bd_net -net reset_1 [get_bd_ports reset] [get_bd_pins
 ↪  mig_7series_0/sys_rst]
connect_bd_net -net rst_mig_7series_0_100M_interconnect_aresetn
 ↪  [get_bd_pins axi_mem_intercon/ARESETN] [get_bd_pins
 ↪  rst_mig_7series_0_100M/interconnect_aresetn]
connect_bd_net -net rst_mig_7series_0_100M_peripheral_aresetn
 ↪  [get_bd_pins axi_mem_intercon/M00_ARESETN] [get_bd_pins
 ↪  axi_mem_intercon/S00_ARESETN] [get_bd_pins mig_7series_0/aresetn]
 ↪  [get_bd_pins rst_mig_7series_0_100M/peripheral_aresetn]
connect_bd_intf_net [get_bd_intf_ports DDR] [get_bd_intf_pins
 ↪  processing_system7_0/DDR]
connect_bd_intf_net [get_bd_intf_ports DDR3] [get_bd_intf_pins
 ↪  mig_7series_0/DDR3]
connect_bd_intf_net [get_bd_intf_pins processing_system7_0/M_AXI_GP0]
 ↪ -boundary_type upper [get_bd_intf_pins axi_mem_intercon/S00_AXI]
```

```
connect_bd_intf_net -boundary_type upper [get_bd_intf_pins
 ↪  axi_mem_intercon/M00_AXI] [get_bd_intf_pins mig_7series_0/S_AXI]
connect_bd_intf_net [get_bd_intf_ports SYS_CLK] [get_bd_intf_pins
 ↪  mig_7series_0/SYS_CLK]
connect_bd_intf_net [get_bd_intf_ports FIXED_IO] [get_bd_intf_pins
 ↪  processing_system7_0/FIXED_IO]
regenerate_bd_layout

# Create address segments
create_bd_addr_seg -range 0x40000000 -offset 0x40000000
 ↪  [get_bd_addr_spaces processing_system7_0/Data] [get_bd_addr_segs
 ↪  mig_7series_0/memmap/memaddr] SEG_mig_7series_0_memaddr

# Add custom IP
set axicore_ip [ create_bd_cell -type ip -vlnv pucrs:user:axicore axicore
 ↪  ]
delete_bd_objs [get_bd_intf_nets processing_system7_0_M_AXI_GP0]
connect_bd_intf_net [get_bd_intf_pins axicore/m_axi] -boundary_type upper
 ↪  [get_bd_intf_pins axi_mem_intercon/S00_AXI]
connect_bd_intf_net [get_bd_intf_pins axicore/s_axi] [get_bd_intf_pins
 ↪  processing_system7_0/M_AXI_GP0]
connect_bd_net [get_bd_pins axicore/s_axi_aclk] [get_bd_pins
 ↪  axi_mem_intercon/S00_ACLK]
connect_bd_net [get_bd_pins axicore/s_axi_aresetn] [get_bd_pins
 ↪  axi_mem_intercon/S00_ARESETN]
connect_bd_net [get_bd_pins axicore/m_axi_aclk] [get_bd_pins
 ↪  axi_mem_intercon/M00_ACLK]
connect_bd_net [get_bd_pins axicore/m_axi_aresetn] [get_bd_pins
 ↪  axi_mem_intercon/M00_ARESETN]

# assign custom address segments
delete_bd_objs [get_bd_addr_segs
 ↪  processing_system7_0/Data/SEG_mig_7series_0_memaddr]
delete_bd_objs [get_bd_addr_segs -excluded
 ↪  axicore/m_axi/SEG_mig_7series_0_memaddr]
assign_bd_address [get_bd_addr_segs {axicore/s_axi/reg0 }]
assign_bd_address [get_bd_addr_segs {mig_7series_0/memmap/memaddr }]
set_property offset 0x40000000 [get_bd_addr_segs
 ↪  {axicore/m_axi/SEG_mig_7series_0_memaddr}]

regenerate_bd_layout
regenerate_bd_layout -routing
validate_bd_design
save_bd_design

generate_target all [get_files
 ↪  axicore_prototype_project/axicore_prototype_project.srcs/sources_1/bd/axicore_c
make_wrapper -files [get_files
 ↪  axicore_prototype_project/axicore_prototype_project.srcs/sources_1/bd/axicore_c
 ↪  -top
```

```
add_files -norecurse
 ↪  axicore_prototype_project/axicore_prototype_project.srcs/sources_1/bd/axicore_c
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
```