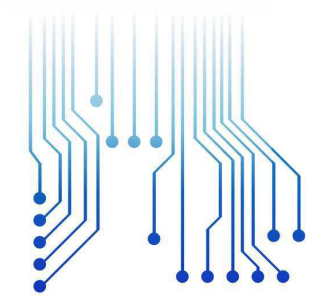


CURSO DE ENGENHARIA ELÉTRICA

Universidade Federal
de Campina Grande

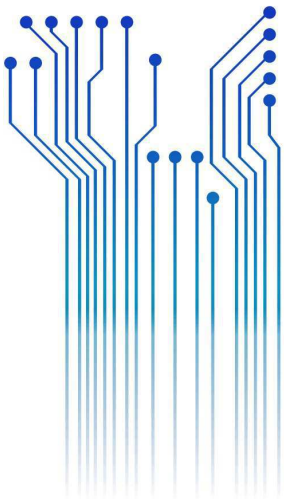
JOSÉ WEMERSON FARIAS LIMA

Centro de Engenharia
Elétrica e Informática

RELATÓRIO DE ESTÁGIO SUPERVISIONADO

LABORATÓRIO EMBEDDED

Departamento de
Engenharia Elétrica



CAMPINA GRANDE

DEZEMBRO DE 2019

JOSÉ WEMERSON FARIAS LIMA

LABORATÓRIO EMBEDDED

RELATÓRIO DE ESTÁGIO SUPERVISIONADO submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Área de concentração: Microeletrônica

Orientador:
Professor Orientador

CAMPINA GRANDE
DEZEMBRO DE 2019

JOSÉ WEMERSON FARIAS LIMA

LABORATÓRIO EMBEDDED

RELATÓRIO DE ESTÁGIO SUPERVISIONADO submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Área de concentração: Microeletrônica

Aprovado em: / /

Professor Avaliador
Universidade Federal de Campina Grande
Avaliador

Professor Orientador
Orientador, UFCG

CAMPINA GRANDE
DEZEMBRO DE 2019

Dedico este trabalho à Deus por toda a força e orientação que me concedeu ao longo desse trabalho.

AGRADECIMENTOS

Primeiramente, agradeço a Deus pelo dom da vida. Em seguida, aos meus pais, José Lima de Oliveira e Ana Lúcia Apolinário Farias Lima, por todo o apoio e suporte que me deram nessa longa jornada até aqui e por sempre acreditarem que eu seria capaz.

Ao meu irmão, Lucas, agradeço por ter me feito evoluir pessoal e profissionalmente.

Agradeço aos mitos que conheco, grupo de estudantes, que sempre estiveram presentes ao longo dessa caminhada, em várias noites de estudos e momentos de descontração. Agradeço também por todas as risadas que foram dadas, por todo apoio e incentivo que nos demos ao longo dessa jornada.

Agradeço aos grupos que participei ao longo da graduação, PET e X-MEN Lab, que me trouxeram engradecimento profissional e pessoal.

Aos meus professores, por todos os ensinamentos e oportunidades que me foram dadas, em especial ao professor Marcos Morais, que me orientou nessa reta final e sempre acreditou no meu potencial, e ao Professor Gutemberg Gonçalves dos Santos Júnior que sempre esteve próximo para sanar as dificuldades.

"Tudo o que fizeres, façam de todo o coração, como para o Senhor, e não para os homens, sabendo que receberão do Senhor a recompensa da herança. É a Cristo, o Senhor, que estão servindo."

Colossenses 3:23-24

RESUMO

Este trabalho apresenta o relatório das atividades realizadas pelo aluno José Wemerson Farias Lima durante o Estágio Supervisionado no Laboratório EMBEDDED localizado no Departamento de Engenharia Elétrica (DEE) da Universidade Federal de Campina Grande (UFCG), sob a orientação do Professor Marcos Ricardo Alcântara Morais e a supervisão do Professor Gutemberg Gonçalves dos Santos Júnior . O estágio consistiu em realizar o estudo de uma CPU RISC-V e uma GPU, posteriormente fazer a conexão entre a CPU e GPU, e por fim, realizar testes da implementação.

Palavras-chave: CPU, GPU. testes.

ABSTRACT

This work presents a report of the activities developed by the student José Wemerson Farias Lima during the Supervised Internship at the EMBEDDED Lab located in the Department of Electrical Engineering (DEE) of the Federal University of Campina Grande (UFCG), under the guidance of Professor Marcos Ricardo Alcântara Morais and supervision of Professor Gutemberg Gonçalves dos Santos Júnior . The stage consisted of studying a RISC-V CPU and a GPU, then making the connection between the CPU and GPU, and finally, perform implementation tests.

Keywords: CPU, GPU, tests .

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Este relatório apresenta as atividades realizadas pelo aluno José Wemerson Farias Lima durante o Estágio Supervisionado no Laboratório Embedded, sob a orientação Professor Marcos Ricardo Alcântara Morais e a supervisão do Professor Gutemberg Gonçalves dos Santos Júnior. O estágio foi realizado entre o período de 16 de Setembro até 04 de Dezembro de 2019, com uma carga horária de 22 horas semanais, somando 251 horas totais.

As unidades centrais de processamento, ou CPU, como é mais conhecida , é uma parte muito importante em qualquer dispositivo digital, sendo o principal processador central. Os usos da CPU são muitos e incluem cálculos, execução de programas, etc. A CPU funciona como parte de um ecossistema mais amplo e diversificado, que inclui RAM (Random Access Memory) e outras partes do computador. A RAM envia muitas instruções para a CPU - que decodifica as instruções e depois as processa e entrega a saída com base.

Mas muitas vezes a CPU não tem como realizar as operações, pois também realiza operações de controle, e vem a pergunta: qual importância da GPU? A pergunta que muitos compradores de smartphones e tablets geralmente não pensam, mas que é muito importante para o desempenho do dispositivo. A GPU, ou unidade de processamento gráfico, é um circuito especializado que se concentra em realizar massivamente cálculos para a geração de imagens para exibição de um dispositivo. Todo dispositivo móvel moderno possui uma forma de GPU para ajudar na geração de imagens e gráficos de computador e é uma parte essencial de qualquer dispositivo móvel moderno. Sem ele, jogos de alto desempenho e elementos elaborados da interface do usuário não seriam possíveis sem sobrecarregar seriamente a CPU e a bateria do dispositivo. A GPU também é muito melhor no processamento de grandes porções de dados em paralelo que uma CPU e permite que a CPU trabalhe menos para produzir gráficos mais detalhados.

Este trabalho está organizado da seguinte forma: a seguir nesta seção, será comentado os objetivos deste trabalho e a metodologia aplicada; na seção 2 encontra-se uma fundamentação teórica acerca do tema; na seção 3 os procedimentos para realização dos testes, como foram realizados e os resultados obtidos com eles; e, finalmente nas seções 4 e 5, as considerações finais e referências utilizadas.

1.1 Objetivos

1.1.1 Objetivo Geral

Fazer a integração por meio de um *bridge* (ponte) entre uma CPU e GPU, escrever e ler da memória de ambos, realizar cálculos mátemáticos com a GPU.

1.1.2 Objetivos Específicos

- Revisão bibliográfico do PULPino;
- Revisão bibliográfica de GPU;
- Integrar CPU e GPU por meio de um barramento.

1.2 Metodologia

A metodologia empregada neste trabalho envolveu a realização de pesquisa e atualização bibliográfica sobre o tema proposto. Inicialmente foram realizados com o *Core* do PULPino, em seguida foram feitos testes com a GPU MEAOW e por fim, foi desenvolvida integração entre esses dois.

2 Fundamentação Teórica

2.1 CPU

Para começar, CPU significa Central Processing Unit. Este é um nome robusto para o que é, em última análise, uma robusta peça de tecnologia, pelo menos em termos do trabalho que realiza. Para ter certeza, a CPU não é a única unidade de processamento em um computador. Há muitos. No entanto, é a unidade central de processamento e, portanto, a mais importante.

Uma boa analogia para a CPU é o cérebro. O cérebro não é apenas o centro dos nervos do corpo, pois não realiza todas as decisões no corpo. A medula espinhal também é um centro de nervos, assim como muitas outras partes do corpo. Todos esses são centros de processamento. No entanto, como o cérebro é a central de processamento, direcionando o maior número de órgãos e processos - também é o mais importante. O cérebro é a CPU do corpo. A CPU é o cérebro do computador.

Os usos da CPU são muitos, cálculos, execução de programas etc. Uma CPU funciona como parte de um ecossistema mais amplo e diversificado, que inclui RAM (Random Access Memory) e outras partes do computador. A RAM envia muitas instruções para a CPU - que decodifica como instruções e depois como processo e entrega à saída com base. As CPUs sofreram muitas melhorias desde que foram fabricadas. Mesmo assim, em essência, todas as CPUs têm as mesmas funções básicas, que são as que as tornam tão importantes.

Toda CPU tem quatro funções principais:

- Fetch;
- Decode;
- Execute;
- Store.

A função **Fetch** é tão simples quanto a recepção de uma instrução ou conjunto de instruções pela CPU. Essas instruções virão da RAM e estão na forma de uma série de números binários (1s e 0s), que é o como os sistemas de *hadware* digitais interagem entre si e para executar processos. Cada instrução que chega à CPU é apenas uma parte de um todo. É um pequeno bloco de construção de uma operação maior. Isso significa que a CPU não recebe apenas instruções às cegas; ele também precisa saber qual instrução seguirá a atual. Para fazer isso, ele possui um contador de programa que monitora os endereços de RAM de onde vêm as instruções.

Se a instrução vier do endereço 1, a CPU saberá que a próxima instrução para esse programa específico deve vir do endereço 2. As instruções são armazenadas em um registro conhecido como Registro de Instruções. Uma vez feito isso, o contador do programa adicionará um para fazer referência ao endereço da próxima instrução.

Uma vez que o computador tenha obtido uma instrução com êxito e armazenada no registro de instruções, pretende-se decodificar essa instrução. Para fazer isso, passa a instrução para um circuito especial conhecido como decodificador de instruções. O decodificador de instruções pega a instrução e a decodifica em um conjunto de sinais, que serão enviados para diferentes partes da CPU em que podem ser acionadas.

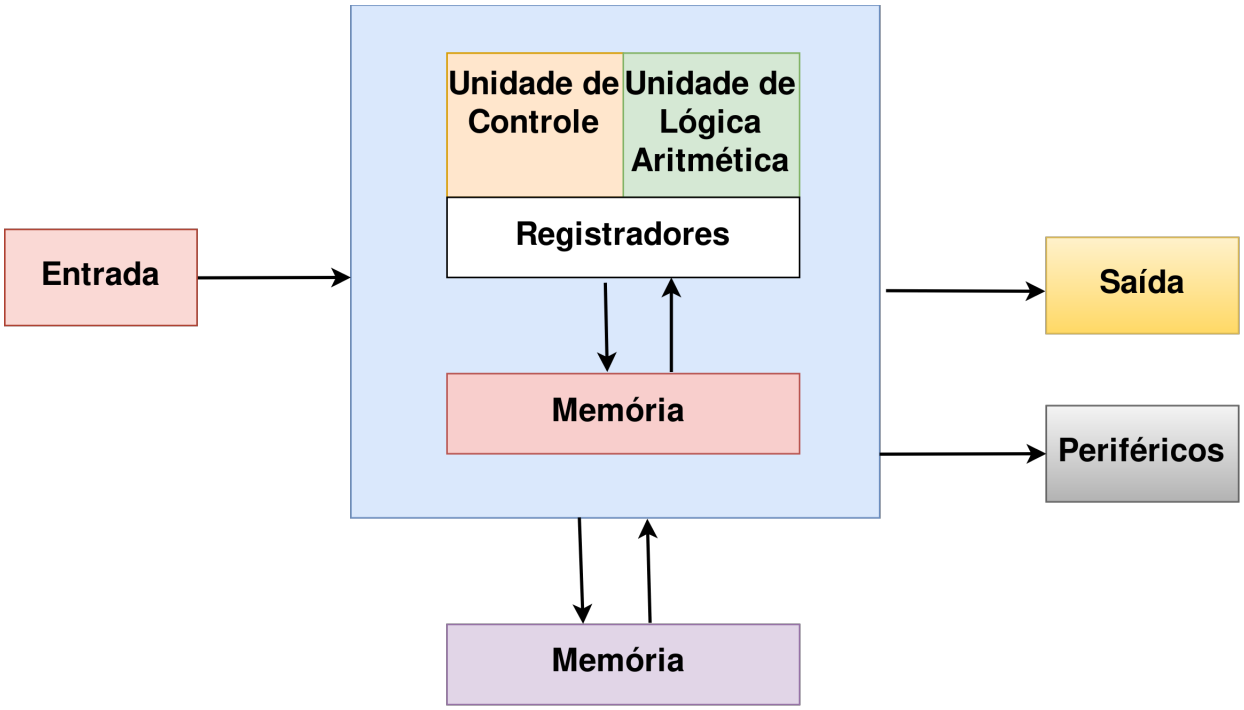
As instruções são recebidas pela CPU, armazenadas no registro de instruções e enviadas ao decodificador de instruções, onde foram decodificadas. Agora elas são enviadas para as diferentes partes da CPU em que são executados. Uma vez executadas as instruções, haverá

saídas armazenadas no registro da CPU, onde outras instruções posteriores poderão referenciá-las. Se um resultado não precisar ser armazenado no registro da CPU ou não tiver atingido seu objetivo no registro, ele será enviado à RAM ou ao disco rígido para armazenamento ou a qualquer um dos muitos dispositivos de saída no computador, como o monitor, os alto-falantes, etc.

Por um lado, a CPU envia as saídas para a RAM e o disco rígido, onde estão armazenadas. Por outro lado, a própria CPU é um dispositivo de armazenamento. Ele é capaz de armazenar dados em qualquer um de seus vários registradores, mesmo que por pouco tempo. Observe, no entanto, que a CPU não é um dispositivo de armazenamento permanente. Quaisquer dados na CPU provavelmente serão perdidos quando a energia do computador for perdida. Todos os processos em execução quando a energia foi desligada também serão interrompidos imediatamente. Os registradores na CPU estão lá com o objetivo principal de manter os dados necessários nos programas e processos atualmente em execução. Assim que os dados são concluídos, eles são excluídos ou enviados para outro local.

A CPU pode executar várias tarefas, alternando entre essas tarefas fazendo parecer que elas estão acontecendo ao mesmo tempo. No entanto, até a CPU mais poderosa tem suas limitações. É por isso que os fabricantes decidiram começar a colocar várias CPUs menores, chamadas núcleos (*cores*), em uma única CPU. Uma CPU com duas CPUs é conhecida como *dual-core*, uma com 4 é conhecida como *quad-core* e assim por diante. Essas CPUs tornam possível realizar várias tarefas, uma vez que cada CPU pode lidar com uma única tarefa de cada vez. Isso torna os computadores muito mais poderosos e também aumenta o desempenho do processador.

Figura 1 – Estrutura básica de uma CPU.



Fonte: O Próprio autor.

2.2 PULPino

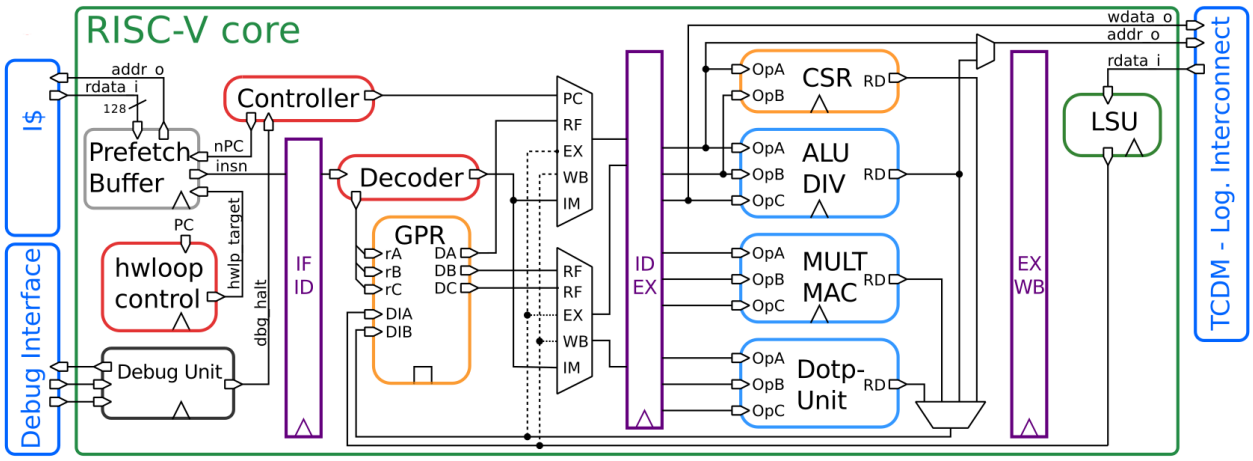
PULPino é o primeiro lançamento *open source* da plataforma de processamento *Parallel Ultra Low Power* (PULP), criada pela Universidade de Bologna e a ETH de Zürich em 2013 com objetivo de explorar novas e eficientes arquiteturas para processamento de baixo consumo de energia.

Este é um SoC de um núcleo e reutiliza a maioria dos componentes do PULP e portas separadas para as RAMs de dados e instruções. Inclui uma ROM de *boot* que contém informações de inicialização que pode carregar um programa via SPI por um dispositivo externo de flash. O SoC utiliza a interface AXI4 como principal interconexão com uma ponte para a interface APB para os periféricos GPIO, UART, I2C, SPI Master, SoC Control, Timer e Event Unit. Ambas as interfaces utilizam um barramento com canais de dados de 32 bits.

O RISC-V é um núcleo com 4 estágios de pipeline e possui um IPC (Instruções por ciclo) próximo a 1, com suporte completo para o conjunto de instruções de inteiro (RV32I), instruções compactadas (RV32C) e extensão do conjunto de instruções de multiplicação (RV32M). Ele pode ser configurado para ter uma extensão de conjunto de instruções de ponto flutuante de precisão única (RV32F). Ele implementa várias extensões ISA, como: loops de hardware, instruções de carregamento e armazenamento pós-incrementais, instruções de manipulação de bits, operações MAC, suporte a operações de ponto fixo, instruções de empacotamento SIMD e produto escalar. Ele foi projetado para aumentar a eficiência energética em aplicações de processamento de sinal com consumo de energia muito baixo.

Quando o núcleo está ocioso, a plataforma pode ser colocada em um modo de baixo consumo de energia, onde apenas uma unidade de evento simples está ativa e todo o resto é ativado pelo *clock* e consome energia mínima (*leakage*). Uma unidade de evento especializada ativa o núcleo caso haja um evento um chegue uma interrupção.

Figura 2 – Diagrama de bloco do *Core* do RISC-V com quatro estágios de *pipeline*.



Fonte: Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices (Publicação IEEE).

2.3 GPU - MIAOW

MIAOW (Acelerador Integrado de Vários Núcleos de Waterdeep / Wisconsin) é uma GPU de código aberto criada pelo Grupo de Pesquisa Vertical da Universidade de Wisconsin-Madison, liderada pelo professor Karu Sankaralingam. Baseado no ISA Southern Islands divulgado publicamente pela AMD, a GPU MIAOW implementa uma unidade de computação adequada para executar análises e experimentações de arquitetura com cargas de trabalho da GPGPU. Além do Verilog HDL que compõe a unidade de computação, o MIAOW também inclui um conjunto de testes de unidade e referências para testes de regressão. Deve-se enfatizar que o MIAOW representa a unidade de computação de uma GPU, por isso não possui a lógica auxiliar necessária para produzir saída gráfica real nem possui lógica para conectá-la a uma interface de memória específica ou barramento do sistema.

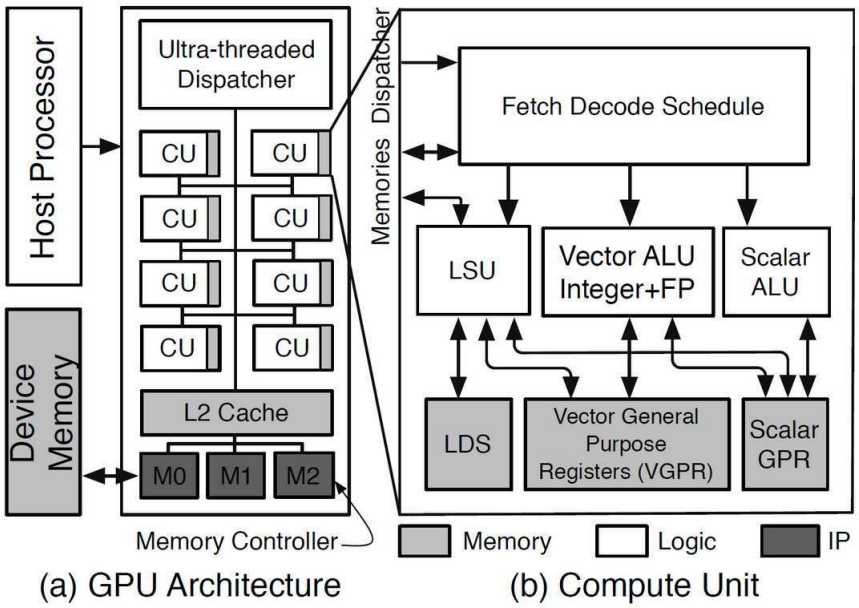
A unidade de computação é composta pelos seguintes componentes lógicos, cada um executando uma etapa na operação das instruções de execução. Há um grau significativo de complexidade em cada componente que dificulta a modificação e a extensão. As páginas a seguir tentarão explicar em detalhes como cada módulo executa suas tarefas e como essas tarefas interagem com o restante do pipeline. Observe que o guia não tentará fornecer um arquivo por arquivo, passo a passo do projeto. No entanto, destacará os módulos importantes e quaisquer pontos técnicos que merecem atenção.

- Fetch: o *fetch* recupera instruções da memória de instruções e também atua como a interface entre a unidade de computação e o mundo externo.
- Wavepool: O wavepool rastreia o progresso da execução das instruções para até 40 frentes de onda em execução na unidade de computação. Cada unidade de computação é projetada para acompanhar 40 frentes de onda em execução simultânea nela. O módulo responsável por acompanhar o estado dessas frentes de onda *wavepool*. Cada frente de onda que é enfileirada na unidade de computação carrega certas informações. Várias delas indicam qual subconjunto do armazenamento da unidade de computação foi alocado para a frente de onda. Outros são simplesmente mecanismos internos de rastreamento de estados.
- Decode: O módulo *decode* recebe uma instrução fornecida pelo *wavepool* e determina quais recursos são necessários para sua execução.
- Issue: A unidade *issue* determina quando os recursos exigidos por uma instrução estão disponíveis e a execução pode prosseguir. É facilmente a parte mais complicada do MIAOW e, a menos que se tenha uma necessidade explícita de modificar o sistema de agendamento.
- Exec: O *exec* é usado para gerar uma máscara de execução que indica quais dos 64 threads em uma frente de onda devem realmente estar em execução.
- SALU: A ULA escalar é usada para executar operações aritméticas para o que necessita instruções de ramificação.
- SIMD: O SIMD é uma ULA de vetor inteiro. Existem quatro deles em uma configuração completa da unidade de computação.
- SIMF: O SIMF é uma ULA de vetor de ponto flutuante. Existem quatro deles em uma configuração completa da unidade de computação.
- SGPR: O arquivo de registro de uso geral escalar é usado para operações aritméticas escalares. Também é onde os dados constantes compartilhados entre vários *threads* são armazenados.
- VGPR: O arquivo de registro de uso geral de vetor é usado para operações aritméticas de vetor.

- RFA: O registrador arbitrário foi projetado para mediar o acesso à única porta de gravação do VGPR. Ele prioriza o acesso à LSU, pois as operações de memória tendem a ser de alta latência e, portanto, cumpri-las o mais rápido possível permite o progresso em qualquer frente de onda que estivesse parada nelas.
- LSU: O LSU lida com solicitações de memória. A Unidade de *Load/Store* é responsável por atender às solicitações de memória emitidas pelas formas de onda. Seu design é semi-flexível, pois pode ser facilmente modificado para acomodar diferentes larguras de barramento, mas essas alterações requerem modificação do código e não são tão simples quanto definir um parâmetro. A LSU é dividida em dois componentes principais com um componente auxiliar para apoiar suas operações. Os módulos principais são o lsu_opcode_decoder e o lsu_op_manager enquanto o componente de suporte é o lsu_addr_calculator. Quando as solicitações de memória entram no LSU, elas são enviadas primeiro ao lsu_opcode_decoder, que extrai das informações da solicitação, como que tipo de instrução, o número de operações de memória em que a solicitação se decompõe e se são necessárias várias operações nos arquivos de registro. Essas informações são transmitidas ao lsu_op_manager, que executa a solicitação real, acompanhando o progresso e notificando a unidade de problema após a conclusão da operação.

Uma instanciação completa da MIAOW deve ser composta por todos os componentes necessários para criar um acelerador de computação altamente paralelo. Isso inclui não apenas as UCs (Unidades de Computação) e os blocos que as controlam, mas também o controlador de memória que faz a mediação entre as UCs e a memória do dispositivo. Também existem caches L1 dedicados para dados e instruções escalares e um cache L2 unificado. A estrutura por diagrama de blocos da GPU MIAOW é ilustrado na Figura 3.

Figura 3 – Estrutura de blocos da GPU MIAOW.

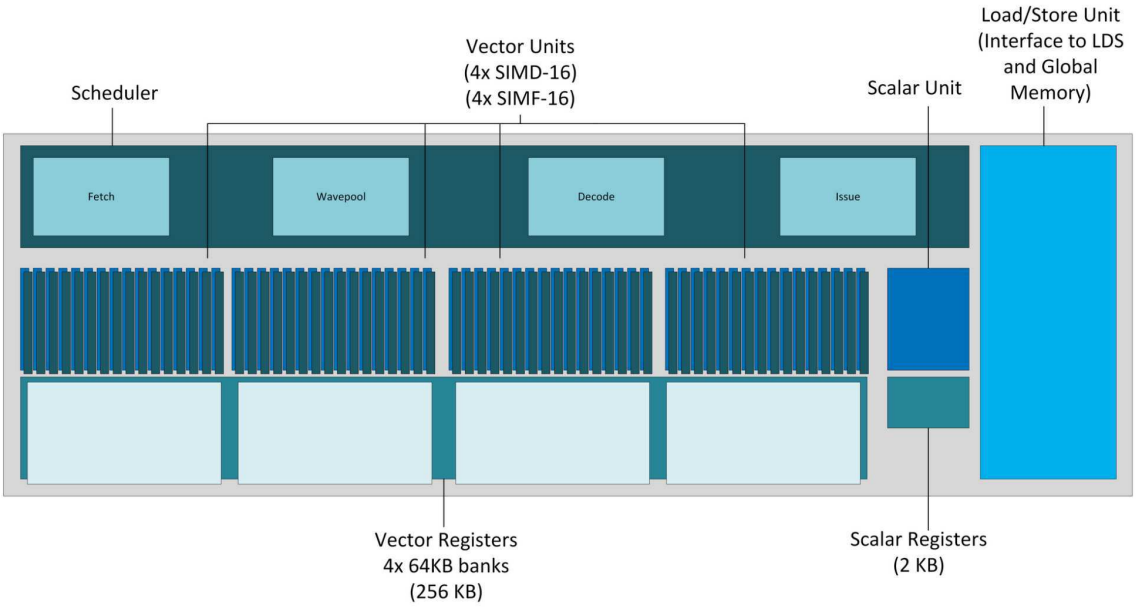


Fonte: Documentação da MIAOW - https://github.com/VerticalResearchGroup/miaow/wiki/Architecture-Overview

De uma perspectiva de alto nível, a GPU MIAOW possui uma implementação bastante fiel da unidade de computação da arquitetura GCN (*Graphics Core Next*), conforme mostrado nos dois diagramas apresentados nas Figuras 4 e 5. Observando que as principais diferenças

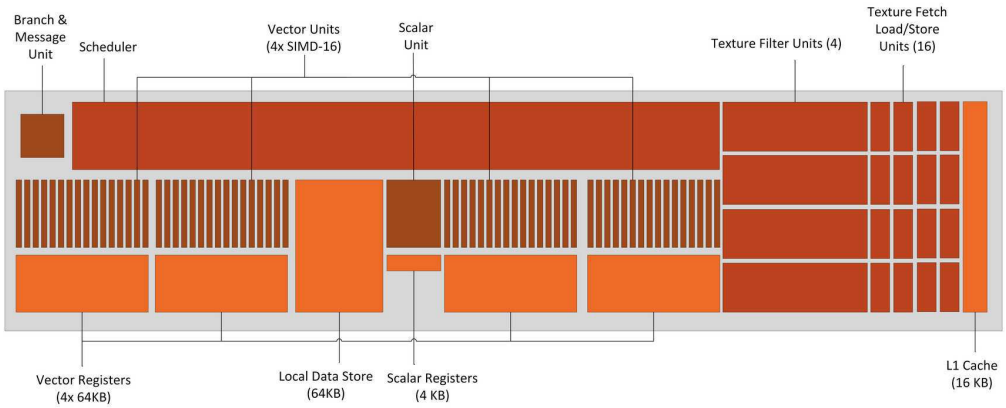
estão na organização da memória, que é altamente específica para o processo tecnológico usado na fabricação, e na funcionalidade relacionada a gráficos que não faziam parte dos objetivos de design originais da MIAOW, mas que poderiam ser adicionados.

Figura 4 – Módulos da Unidade de Computação da MIAOW.



Fonte: Documentação da MIAOW - https://github.com/VerticalResearchGroup/miaow/wiki/Architecture-Overview

Figura 5 – Módulo da Unidade de Computação de Kaveri, adaptado da apresentação da AMD em Hot Chips 26.



Fonte: Documentação da MIAOW - https://github.com/VerticalResearchGroup/miaow/wiki/Architecture-Overview

A UC é composta pelos módulos necessários para executar operações aritméticas escalares e vetoriais. A unidade **fetch** atua como a interface entre a UC e o **dispatcher**, recebendo as informações necessárias para executar uma frente de onda de dados quando uma é agendada para a UC. O **wavepool** atua como uma fila para todas as instruções que foram

buscadas. É capaz de acompanhar e suportar até 40 frentes de onda diferentes ao mesmo tempo. A unidade de decodificação lida com decodificação de instruções e também com instruções de 64 bits. Ele também determina qual ALU executará a instrução e executará a conversão de endereços para os registros. A unidade de emissão rastreia todas as instruções de voo e os recursos que estão usando, resolvendo dependências entre elas. Ele garante que todos os recursos necessários para uma instrução estejam disponíveis antes de permitir sua execução.

A UC suporta operações vetoriais e escalares. Em uma instanciação completa da UC, as ULAs vetoriais são organizadas em bancos de dezesseis, com quatro bancos para entradas inteiras e de ponto flutuante. Isso fornece 64 ULAs para suportar a execução simultânea de todos os 64 threads em uma frente de onda, assumindo que não haja conflitos de outros recursos.

Os recursos de memória na UC são organizados em registros, um bloco local de registro e uma interface para a memória do dispositivo. Os registradores são separados em arquivos de registro dedicados para operações escalares e vetoriais, com as portas do arquivo de registro vetorial adequadamente ampliadas para suportar o acesso simultâneo de um banco inteiro de ULAs vetoriais.

Vários **dispatches** foram desenvolvidos para testar a MIAOW. O original era uma combinação dos códigos Verilog e C que atuavam como **testbenchs** para executar testes de unidade e benchmarks maiores. Outro distribuidor de software foi implementado como um programa C incorporado para o processador Microblaze na FPGA Virtex7. Um **dispatcher** de hardware também foi implementado para a MIAOW. Ele monitora os recursos alocados e os grupos de trabalho que a CPU solicitou que fosse executada. Quando uma UC com recursos suficientes estiver disponível, ela dividirá o grupo de trabalho em frentes de onda e entregará cada frente de onda à UC.

A unidade **fetch** do MIAOW busca uma única instrução de cada vez, enquanto a documentação da GCN especifica uma busca de 16 ou 32 instruções. Especulamos que isso se deve a um alinhamento da linha de cache. A taxa de emissão de uma instrução por ciclo foi projetada para corresponder à da largura de banda do **fetch**. Aumentá-lo exigiria portas de leitura adicionais para os arquivos de registro por razões óbvias, já que o bloqueio das leituras do arquivo de registro nega efetivamente qualquer vantagem que possa ser obtida com a emissão de várias instruções em um único ciclo.

Mesmo sendo uma boa unidade de processamento de dados, a GPU MIAOW possui duas limitações principais em sua implementação, que precisariam ser resolvidas para colocar o MIAOW em produção como uma placa de vídeo real. A primeiro limitação tem impacto no seu uso para fins de GPGPU. Especificamente, a interface de memória do MIAOW ainda está em um estado de fluxo. Durante o trabalho para ativar o MIAOW em um FPGA, observou-se que a interface de memória existente era muito ampla e profunda e que tentar abstraí-lo teria resultado em uma sobrecarga significativa. Foi tomada a decisão de refazer o pipeline da unidade de computação, ou seja, na unidade **issue**, para permitir uma interface de memória mais prática. Este trabalho ainda não foi concluído.

A segunda limitação diz respeito ao uso do MIAOW como uma unidade gráfica. Como o MIAOW foi implementado para experimentar cargas de trabalho da GPGPU, faltam algumas instruções e funcionalidades que eram exclusivamente para gráficos, como suporte de textura e similares. Ele também não possui nada para produzir gráficos de verdade. O conjunto de instruções suportadas precisaria ser expandido e a lógica auxiliar para saída gráfica também precisaria ser implementada, embora o último seja mais um problema de projeto elétrico do que lógica digital.

3 Atividades Desenvolvidas

A primeira atividade desenvolvida foi usar os arquivos disponíveis no GitHub do grupo de pesquisa que desenvolveu a GPU MIAOU, e seguir as instruções inciais para colocar a GPU para funcionar. As ferramentas utilizadas tanto pelos autores da MIAOW quanto pelo estagiário foram da Synopsys e no Sistema Operacional Linux Centos 7.

Os autores da MIOAW forneceram as seguintes informações:

- Clonar o repositório da MIAOW para o diretório local
- Ir até a pasta de *testbench* cd $TOP_DIR/src/verilog/tb
- E inserir o comando via terminal **make compile**

Mas alguns problemas foram detectados imediatamente. Vários arquivos Verilog estavam faltando para serem compilados, um deles era a memória. Dessa forma, a GPU não tem como salvar as instruções e os dados a serem processados. Para sanar o problema, tentou-se entrar em contato com a equipe de desenvolvimento da MIAOW mas sem êxito. Foi decidido fazer o RTL da memória, que possui a macroarquitetura ilustrada na Figura 6.

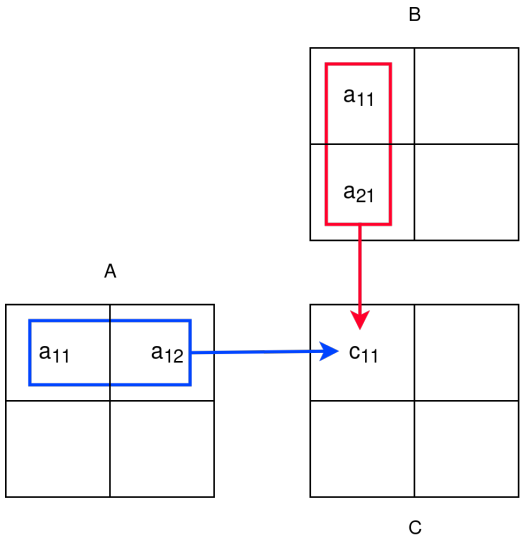
Figura 6 – Macroarquitetura da memória da GPU.

Fonte: O Próprio autor.

A microarquitura dessa memória se divide em duas partes: uma para a parte procedural, a qual irá realizar as operações de leitura e escrita, e a segunda parte referente ao controle da memória, responsável por decidir em que momento cada ação deve ser executada. Já com a nova memória em funcionamento, foi adicionada ao escopo do projeto e feito uma nova rodada de exucação para simulação dos arquivos Verilog.

Os primeiros testes executados foram os mesmos já disponíveis no *testbench* dos desenvolvedores da MIAOW. Mas são apenas somas simples e que utilizam poucos *threads*. Para utilizar uma maior capacidade da GPU, foram feitas multiplicações de matrizes. Antes de ver a implementação, é necessário entender como funciona a multiplicação de matrizes da forma tradicional. Sejam duas matrizes, A e B. Suponha que A seja uma matriz $n \times m$, o que significa que possui n linhas e m colunas. Suponha também que B seja uma matriz $m \times w$. O resultado da multiplicação A * B (que é diferente de B * A) é uma matriz $n \times w$, que chamamos de M. Ou seja, o número de linhas na matriz resultante é igual ao número de linhas da primeira matriz A e o número de colunas da segunda matriz B.

Figura 7 – Forma de multiplicar matrizes.

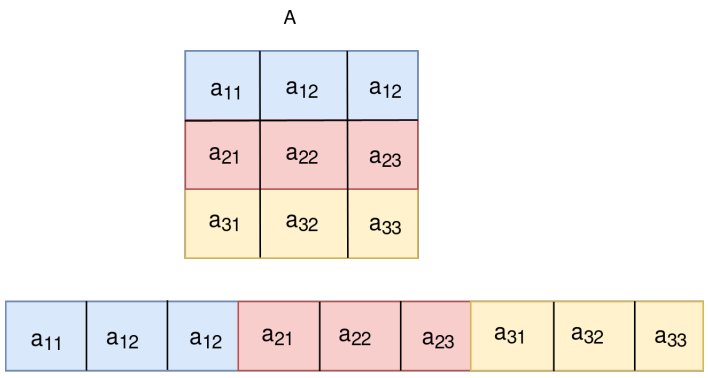


Fonte: O Próprio autor.

A partir da Figura 7, fica mais claro agora porque a multiplicação matriz-matriz é um bom exemplo para computação paralela. Temos que calcular todos os elementos em C, e cada um deles é independente dos outros, para que possamos paralelizar com eficiência. Veremos diferentes maneiras de conseguir isso. O objetivo é adicionar novos conceitos ao longo deste relatório, terminando com um kernel 2D, que usa memória compartilhada para otimizar operações com eficiência.

Para realizar as operações matriciais, as matrizes serão interpretadas como 1D. Isso pode parecer um pouco confuso, mas o problema está na própria linguagem de programação. O padrão no qual a maioria das GPUs são desenvolvidas precisa saber o número de colunas antes de compilar o programa. Portanto, é impossível alterá-lo ou configurá-lo no meio do código.

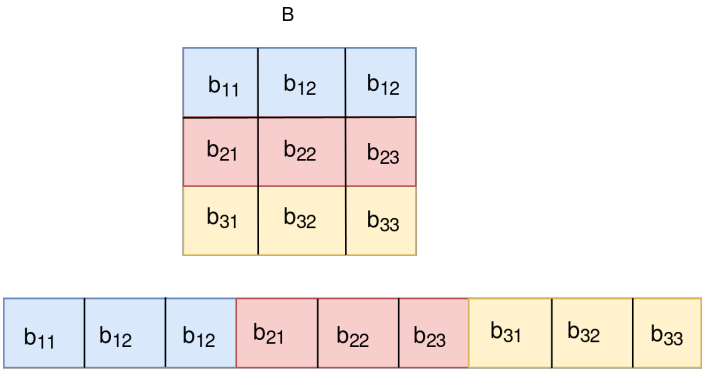
Figura 8 – Forma de interpretação de uma matrix 2D para 1D (matriz A).



Fonte: O Próprio autor.

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31}$$

Figura 9 – Forma de interpretação de uma matrix 2D para 1D (matriz B).

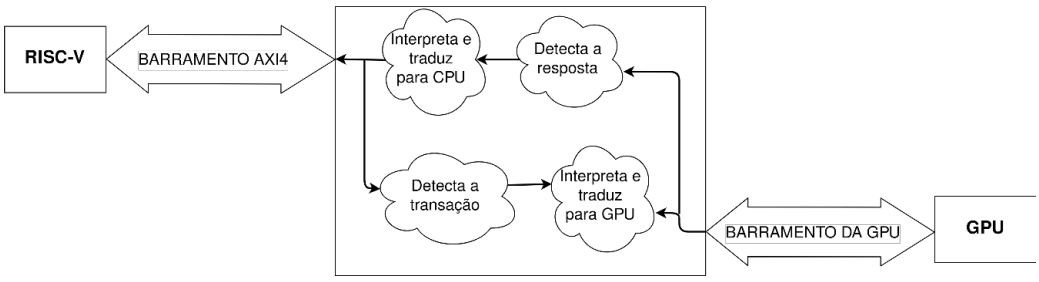


Fonte: O Próprio autor.

$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22} + a_{13} * b_{32}$$
$$c_{13} = a_{11} * b_{13} + a_{12} * b_{23} + a_{13} * b_{33}$$
$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21} + a_{23} * b_{31}$$
$$c_{21} = a_{21} * b_{12} + a_{22} * b_{22} + a_{23} * b_{32}$$
$$c_{21} = a_{21} * b_{13} + a_{22} * b_{23} + a_{23} * b_{33}$$
$$c_{31} = a_{31} * b_{11} + a_{32} * b_{21} + a_{33} * b_{31}$$
$$c_{31} = a_{31} * b_{12} + a_{32} * b_{22} + a_{33} * b_{32}$$
$$c_{31} = a_{31} * b_{13} + a_{32} * b_{23} + a_{33} * b_{33}$$

Para utilizar o barramento AXI4 que o PULPino faz uso, foi necessário projetar uma *brigde* para fazer a relação entre as requisições do processador e as respostas da GPU MIAOW. O diagrama de bloco dessa *bridge* é ilustrado na Figura 10.

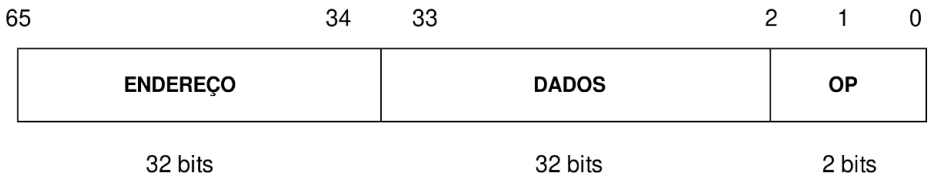
Figura 10 – Diagrama de blocos da *bridge* entre o barramento AXI4 e o barramento da GPU.



Fonte: O Próprio autor.

Dessa forma, há muita operações que podem ser executadas de forma paralela. Os desenvolvedores da MIAOW já deixaram disponível código assembly e hexadecimal para ser carregado ao processador. Posteriormente, o processador envia por meio do AXI4 os dados,

Figura 11 – Composição da palavra do barramento AXI4.



endereço e o bit de leitura ou escrita. São 32 bits de dados, 32 bits de endereço, 1 bit para identificar se é operação de leitura ou escrita e um bit de habilitação de processo.

4 CONCLUSÃO

O período decorrido ao longo do estágio foi consideravelmente proveitoso para o crescimento pessoal e profissional do estagiário, uma vez que, por muitas vezes, foi necessária a comunicação com colegas de diferentes setores para melhor entendimento do problema a ser resolvido.

Dessa forma, a imersão em um problema que ainda não tinha sido resolvido, com prazos a serem cumpridos, consequências de decisões equivocadas, o que acabou gerando um maior desempenho por parte do aluno. Com isso, cumpriu-se o objetivo acadêmico e pessoal que era o de experimental o funcionamento de um estágio voltado diretamente para a área de microeletrônica.

Por fim, durante toda a extensão dessas atividades, foi necessário o emprego de várias habilidades comportamentais, denominadas *soft skills*. Essas habilidades envolvem capacidades interpessoais, e seu emprego é altamente desejado no ambiente de trabalho. Comportamentos de interesse geralmente envolvem proatividade, criatividade, gentileza, interesse e empenho ao se realizar atividades. Dessa maneira, todas as atividades empregadas durante o período de estágio tiveram, como embasamento, o emprego destes valores para que o ambiente de trabalho se mantivesse sempre saudável. A partir do retorno de colegas de trabalho, foi possível então aprender como se portar de maneira mais agradável e produtiva no ambiente de trabalho, assim como descobrir pontos a serem trabalhados e aperfeiçoados.

Referências

Michael Gautschi; Pasquale Davide Schiavone; ,"IEEE Transactions on Very Large Scale Integration (VLSI) Systems", Volume: 25 , Issue: 10 , Outubro de 2017

Nicky LaMarco, "The Importance of a Computer CPU". Disponível em: https://smallbusiness.chron.com/importance-computer-cpu-54856.html

QuantStart, "Matrix-Matrix Multiplication on the GPU with Nvidia CUDA". Disponível em: https://www.quantstart.com/articles/Matrix-Matrix-Multiplication-on-the-GPU-with-Nvidia-CUDA

Vertical Research Group. Repositório GitHub: https://github.com/VerticalResearchGroup/miaow

Documentação do PULPino. Repositório GitHub: https://github.com/pulp-platform/pulpino

Apêndices

APÊNDICE A – Código em SystemVerilog da Memória da GPU

```
1
2  module memx(
3      dft_tm_i,
4      clk_i,
5      memx_rd_i,
6      memx_wr_i,
7      memx_addr_i,
8      memx_wdt_i,
9      memx_busy_o,
10     memx_rdt_o,
11     memx_wok_o,
12     ram_ADDR,
13     ram_READ,
14     ram_PROG,
15     ram_DIN,
16     ram_DOUT,
17     ram_READY
18 );
19 // ====================================================
20 // Parameters
21 // ====================================================
22 parameter ADDR_WIDTH = 6;
23 parameter DATA_WIDTH = 16;
24 // ====================================================
25 // Interface
26 // ====================================================
27
28 input  logic                  dft_tm_i;
29 input  logic                  clk_i;
30 input  logic                  memx_rd_i;
31 input  logic                  memx_wr_i;
32 input  logic [ADDR_WIDTH-1:0] memx_addr_i;
33 input  logic [DATA_WIDTH-1:0] memx_wdt_i;
34 output logic                  memx_busy_o;
35 output logic [DATA_WIDTH-1:0] memx_rdt_o;
36 output logic                  memx_wok_o;
37
38 // ====================================================
39 // RAM memory signals
40 // ====================================================
41
42 output logic                  ram_READ;
43 output logic                  ram_PROG;
44 output logic [ADDR_WIDTH-1:0] ram_ADDR;
45 output logic [DATA_WIDTH-1:0] ram_DIN;
46 input  logic [DATA_WIDTH-1:0] ram_DOUT;
```

```
47 input  logic                    ram_READY;
48
49 // ================================================
50 // Internal signals
51 // ================================================
52
53 logic                          enb_cg;
54 logic                          rstn;
55
56
57
58 // ================================================
59 // SUB-BLOCK memx_CTRL
60 // ================================================
61 memx_ctrl memx_ctrl(
62     .dft_tm_i          (dft_tm_i),
63     .clk_i             (clk_i),
64     .memx_rd_i         (memx_rd_i),
65     .memx_wr_i         (memx_wr_i),
66     .memx_busy_o       (memx_busy_o),
67     .memx_wok_o        (memx_wok_o),
68     .enb_cg_o          (enb_cg),
69     .rstn_o            (rstn),
70     .ram_READ          (ram_READ),
71     .ram_PROG          (ram_PROG),
72     .ram_READY         (ram_READY)
73 );
74 // ================================================
75 // SUB-BLOCK memx_PROC
76 // ================================================
77 memx_proc #(
78              .ADDR_WIDTH       (ADDR_WIDTH),
79              .DATA_WIDTH       (DATA_WIDTH)
80     )memx_proc(
81     .clk_i             (clk_i),
82     .memx_addr_i       (memx_addr_i),
83     .memx_wdt_i        (memx_wdt_i),
84     .memx_rdt_o        (memx_rdt_o),
85     .enb_wr_i          (memx_wr_i),
86     .enb_cg_i          (enb_cg),
87     .rstn_i            (rstn),
88     .ram_ADDR          (ram_ADDR),
89     .ram_DIN           (ram_DIN),
90     .ram_DOUT          (ram_DOUT)
91 );
92
93 endmodule
94
95
96
97 endmodule
```

```
1
2  module memx_proc
3    #(parameter
4      ADDR_WIDTH = 6,
5      DATA_WIDTH = 32
6    )
7    (
8      clk_i,
9      memx_addr_i,
10     memx_wdt_i,
11     memx_rdt_o,
12     enb_wr_i,
13     wr_ok_o,
14     enb_cg_i,
15     rstn_i,
16     ram_ADDR,
17     ram_DIN,
18     ram_DOUT
19 );
20
21 // ================================================
22 // Interface
23 // ================================================
24 input  logic                  clk_i;
25 input  logic [ADDR_WIDTH-1:0] memx_addr_i;
26 input  logic [DATA_WIDTH-1:0] memx_wdt_i;
27 input  logic                  enb_wr_i;
28 input  logic                  enb_cg_i;
29 input  logic                  rstn_i;
30
31 output logic [DATA_WIDTH-1:0] memx_rdt_o;
32 output logic                  wr_ok_o;
33
34 // ================================================
35 // RAM memory signals
36 // ================================================
37
38 input  logic [DATA_WIDTH-1:0] ram_DOUT;
39 output logic [ADDR_WIDTH-1:0] ram_ADDR;
40 output logic [DATA_WIDTH-1:0] ram_DIN;
41
42 // ================================================
43 // Registers
44 // ================================================
45
46 always_ff @(posedge clk_i or negedge rstn_i) begin : proc_addr
47   if(~rstn_i)
48     ram_ADDR <= 0;
49   else begin
50     if(enb_cg_i) begin
```

```
51        ram_ADDR <= memx_addr_i;
52      end else begin
53        ram_ADDR <= ram_ADDR;
54      end
55    end
56  end
57
58  always_ff @(posedge clk_i or negedge rstn_i) begin : proc_data_i
59    if(~rstn_i)
60      ram_DIN <= 0;
61    else begin
62      if(enb_cg_i && enb_wr_i) begin
63        ram_DIN <= memx_wdt_i;
64      end else begin
65        ram_DIN <= ram_DIN;
66      end
67    end
68  end
69
70  always_ff @(posedge clk_i or negedge rstn_i) begin : proc_rdt_o
71    if(~rstn_i) begin
72      memx_rdt_o <= 0;
73    end else begin
74      if()
75      memx_rdt_o <= ;
76    end
77  end
78
79  // ================================================
80  // Output logic
81  // ================================================
82
83  assign wr_ok_o    = (ram_DOUT == ram_DIN) ?1'b1 :1'b0;
84
85  endmodule
```

```
1
2  module memx_ctrl(
3          dft_tm_i,
4      clk_i,
5      memx_rd_i,
6      memx_wr_i,
7      memx_busy_o,
8      memx_wok_o,
9      wr_ok_i,
10     enb_cg_o,
11     rstn_o,
12     ram_READ,
13     ram_PROG,
14     ram_READY
```

```
15 );
16 // ======================================================
17 // Interface
18 // ======================================================
19 input  logic     dft_tm_i;         //DFT signal
20 input  logic     clk_i;            //CLOCK
21 input  logic     memx_rd_i;        //READ enable input
22 input  logic     memx_wr_i;        //WRITE enable input
23 input  logic     wr_ok_i;          //WRITE successful signal
24
25 output logic     enb_cg_o;         //clock gate enable
26 output logic     rstn_o;           //reset signal
27 output logic     memx_busy_o;      //busy signal
28 output logic     memx_wok_o;       //writing done signal
29
30 // ======================================================
31 // RAM memory signals
32 // ======================================================
33
34 output logic     ram_READ;         //READ enable for memory
35 output logic     ram_PROG;         //WRITE enable for memory
36 input  logic     ram_READY;        //Ready signal from memory
37
38 // ======================================================
39 // List of states
40 // ======================================================
41
42 parameter IDLE     = 3'b000;
43 parameter WAIT     = 3'b010;
44 parameter WRITING  = 3'b110;
45 parameter READING  = 3'b011;
46 parameter WR_DONE  = 3'b001;
47 parameter WR_CHECK = 3'b111;
48
49 // ======================================================
50 // Internal signals
51 // ======================================================
52
53 logic  [2:0]     state;
54 logic  [2:0]     next_state;
55 logic            enb_cg_w;
56 logic            rstn_b_w;
57
58 // ======================================================
59 // Main logic
60 // ======================================================
61
62 assign enb_cg_w   =  memx_rd_i |  memx_wr_i;
63 assign rstn_b_w   = (dft_tm_i == 1'b0)? enb_cg_w : 1'b1;
64 assign rstn_o     = rstn_b_w;
65
66 // ======================================================
```

```
67 // Controller -> FSM
68 // ======================================================
69
70 always_ff @(posedge clk_i or negedge rstn_b_w) begin
71     if( ~rstn_b_w )
72         state <= IDLE;
73     else begin
74         if(enb_cg_w)
75                         state <= next_state;
76                 else
77                         state <= state;
78         end
79 end
80
81 always_comb begin
82 ext_state = IDLE;
83 ase(state)
84  IDLE:      begin
85             if(enb_cg_w && ram_READY)
86                 next_state = WAIT;
87             else
88                 next_state = IDLE;
89         end
90
91  WAIT:      begin
92                         if(~ram_READY) begin
93                                 if(memx_rd_i)
94                                         next_state = READING;
95                                 else if (memx_wr_i)
96                                         next_state = WRITING;
97                         end else
98                                         next_state = WAIT;
99                         end
100
101  READING:       begin
102                         if(ram_READY) begin
103                                 if(memx_wr_i)
104                                         next_state = WR_DONE;
105                         else
106                                 next_state = IDLE;
107                 end
108                 else
109                         next_state = READING;
110         end
111
112  WRITING:  begin
113         if(ram_READY)
114                         next_state = WR_CHECK;
115                 else
116                     next_state = WRITING;
117         end
118
```

```
119  WR_CHECK: begin
120                          if(~ram_READY)
121                                  next_state = READING;
122                          else
123                                  next_state = WR_CHECK;
124                          end
125      endcase
126  end
127
128  always_comb begin
129          memx_busy_o     = 1'b0;
130          memx_wok_o      = 1'b0;
131          ram_PROG        = 1'b0;
132          ram_READ        = 1'b0;
133          enb_cg_o        = 1'b0;
134
135      case(state)
136        IDLE:      begin
137                          enb_cg_o      = enb_cg_w;
138                          end
139
140        WAIT:      begin
141                          memx_busy_o   = 1'b1;
142                          if (memx_rd_i)
143                                  ram_READ = 1'b1;
144                          else if (memx_wr_i)
145                                  ram_PROG = 1'b1;
146                          end
147
148            WRITING:  begin
149                                  memx_busy_o = 1'b1;
150                                  ram_PROG    = 1'b1;
151                                  end
152
153            READING:  begin
154                                  memx_busy_o = 1'b1;
155                                  ram_READ    = 1'b1;
156                                  end
157
158            WR_CHECK: begin
159                                  memx_busy_o = 1'b1;
160                                  ram_READ    = 1'b1;
161                                  end
162
163            WR_DONE:  begin
164                                  if(wr_ok_i)
165                                          memx_wok_o   = 1'b1;
166                                  else
167                                          memx_wok_o   = 1'b0;
168                          end
169      endcase
170  end
```

```
171
172 endmodule
```