UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

THALES BAIERLE TABORDA

Implementação de Acelerador com Arquitetura Multi-Núcleos ACQuA

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Gabriel Luca Nazar

Porto Alegre
2017

AGRADECIMENTOS

Aos meus pais, pelo apoio incondicional, o carinho e o incentivo. São sem dúvida os grandes pilares sobre os quais se sustenta tudo que construí até hoje. Sem os mesmos nenhuma etapa dessa jornada teria sido possível.

Aos meus professores e à universidade, em especial ao Instituto de Informática, pela excelência em ensino oferecida aos seus discentes.

Ao Professor Dr. Gabriel Nazar, cuja orientação na elaboração desse trabalho foi inestimável e também pelo grande papel que representou na escolha dos meus interesses dentro da computação.

Aos amigos pelo suporte despendido de muitas formas ao longo da graduação. Seja nas madrugadas realizando trabalhos ou nas madrugadas com cerveja e videogame.

À Tuti, pela presença em 87% do tempo de elaboração desse trabalho.

E finalmente, à Paula, por sempre acreditar em mim, mesmo quando eu não o fazia.

"If I have seen further it is only by standing on the shoulders of giants." (Newton, Isaac)

RESUMO

Uma das técnicas mais proeminentes para se obter desempenho computacional atualmente é mediante o uso de paralelismo, executar múltiplas tarefas simultaneamente em unidades de processamento distintas. Amplamente utilizada nos processadores multi-núcleo recentes, tais técnicas dependem, em sua grande maioria, do envolvimento direto de um desenvolvedor com mecanismos de sincronização, controle de seções críticas e a correta manipulação dos dados, o que pode aumentar os níveis de esforço e tempo necessários para produzir software e garantir seu perfeito funcionamento.

Esse trabalho tem como objetivo, portanto, a descrição e implementação de um acelerador com a arquitetura ACQuA (Active Call Queue Architecture), que visa a exploração do paralelismo inerente a linguagens funcionais de maneira transparente ao programador, eliminando a necessidade de explicitar paralelismo e estruturas de sincronização. O acelerador desenvolvido se utiliza de estruturas e mecanismos presentes em hardware buscando minimizar overheads de despacho, comunicação e sincronização de chamadas de função independentes servindo, também, como instrumento de análise adicional da arquitetura proposta.

Palavras-chave: ACQuA, Paralelismo, Linguagens Funcionais, Acelerador de Hardware, Multi-Núcleo.

Development of a Hardware Accelerator of the ACQuA Multi-Core Architecture

ABSTRACT

One of the most prominent techniques to increase processing power is parallelism, to execute multiple tasks simultaneously in distinct processing units. Widely used in recent multi-core processors, such techniques depend mostly on a developer's direct engagement to synchronization mechanisms, critical section control and correct manipulation of data, this can increase the amount of time needed to produce software and guarantee its perfect working condition.

This work's focus, therefore, is to develop and implement a hardware accelerator of the ACQuA architecture (Active Call Queue Architecture), a novel architecture that aims to explore the inherent parallelism present in pure functional languages in a transparent manner, eliminating the need to explicit complex synchronization structures. The developed accelerator benefits from hardware structures and mechanisms to facilitate dispatch, communication and synchronization of independent function calls in order to minimize the overhead of these operations and also serves as a further analysis instrument to the proposed architecture.

Keywords: ACQuA, Parallelism, Functional Languages, Hardware Accelerator, Multi-Core.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACQuA	Active Call Queue Architecture
API	Application Program Interface
FPGA	Field Programmable Gate Array
ISA	Instruction Set Architecture
PU	Processing Unit
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

SUMÁRIO

1 INTRODUÇÃO

1.1 Motivação

Nos processadores de propósito geral de hoje em dia, contamos com arquiteturas multi-núcleos para melhorar o desempenho realizando tarefas em paralelo. Softwares devem ser, portanto, escritos com esse paralelismo em mente, dificultando o processo de desenvolvimento e o trabalho do programador, visto que a tarefa de paralelização automática é extremamente não-trivial em linguagens imperativas devido a possíveis *side-effects*.

Side-effects ocorrem quando uma função ou trecho de código altera uma variável (seja uma variável global, estática, uma escrita em disco ou em um display) e são bastante frequentes em linguagens imperativas, que são caracterizadas por terem uma noção implícita de estado que é modificada durante o código por atribuições, *loops*, comandos condicionais dentre outras estruturas que fazem com que um programa tenha uma característica de sequência entre os comandos, permitindo controle preciso e determinístico sobre o seu estado. Linguagens puramente funcionais, por outro lado, evitam, ou eliminam completamente os *side-effects*, o estado da execução é carregado de forma explícita nas aplicações que necessitam do mesmo e *loops* são realizados por recursão em vez de estruturas do tipo *while-loop* ou *for-loop*, fazendo com que boa parte das linguagens desse tipo tenha uma forte semelhança com uma notação matemática (HUDAK, 1989). Essas características permitem que funções em linguagens funcionais possam, dessa forma, serem executadas em paralelo com outras chamadas de função (HAMMOND 2011).

Esse é o paralelismo que ACQuA, arquitetura proposta em projeto conduzido no Instituto de Informática da Universidade Federal do Rio Grande do Sul, busca acelerar, executando chamadas de função em diferentes unidades de processamento sem que um programador necessite explicitar quais trechos do programa são executados em paralelo (TANUS; NAZAR; MOREIRA, 2017). Facilitando, dessa forma, tanto o ganho em desempenho quanto a clareza do código, sem a necessidade de construções de sincronização (como semáforos e variáveis *mutex*) que muitas vezes dificultam a análise de códigos que exploram paralelismo. Esses benefícios fazem da arquitetura algo altamente desejável, visto que um processador eventualmente pode conter dezenas de núcleos e a obtenção de paralelismo e sincronização podem se tornar tarefas quase impossíveis de serem executadas manualmente.

Nesse trabalho foi implementado em VHDL um acelerador de hardware que se utiliza da arquitetura em questão. O código foi desenvolvido de forma parametrizável, tendo como

objetivo avaliar de forma precisa a área e desempenho que podem ser obtidos usando ACQuA. A parametrização da arquitetura permite a exploração de espaço de projeto em diferentes eixos e uma avaliação mais profunda da sua escalabilidade à medida que os parâmetros são modificados. Oferece-se uma análise da variação de um dos parâmetros mais notáveis, que é o número de núcleos de processamento do acelerador, e como sua variação impacta o desempenho, área e frequência de operação do hardware desenvolvido.

É descrita também nesse texto a estrutura geral de cada módulo do acelerador e as diferentes unidades criadas com o propósito de possibilitar seu funcionamento. As operações executadas por cada unidade são discutidas em detalhe e como elas são integradas no fluxo de execução das funções em ACQuA com o objetivo de obter aumento em desempenho.

O trabalho desenvolvido em (TANUS; NAZAR; MOREIRA, 2017) provê, além da descrição do funcionamento teórico de ACQuA, um compilador de uma linguagem funcional pura para uma linguagem intermediária ($ACQuA_{IR}$) que é similar a uma *ISA* (Instruction Set Architecture) comum, mas contém os comandos necessários para o modelo de execução ACQuA. Um simulador para $ACQuA_{IR}$ também foi desenvolvido em (TANUS; NAZAR; MOREIRA, 2017) e o mesmo foi utilizado como instrumento de validação da arquitetura. Um número estimado de ciclos foi atribuído no simulador para cada instrução e operação que não necessariamente refletem números reais da execução em um circuito, diferentemente do resultado obtido e discutido no trabalho de implementação aqui descrito. Tanus, Nazar e Moreira (2017) não estabelecem, também, todas estruturas de hardware necessárias para o funcionamento da arquitetura de maneira que os módulos aqui demonstrados e seu respectivo funcionamento são decisões de projeto feitas pelo autor. Por fim, é importante ressaltar que as unidades de processamento neste trabalho não executam diretamente $ACQuA_{IR}$, tendo sido feita uma tradução manual do código $ACQuA_{IR}$ equivalente para uma versão binária na *ISA* aqui utilizada, mais especificamente a *ISA* RISC-V.

1.2 Estrutura do Trabalho

O capítulo 2 contém uma discussão dos trabalhos relacionados. No capítulo 3 é dada uma descrição da arquitetura ACQuA e o modelo de seu funcionamento. No capítulo 4 é discutida a implementação do acelerador de hardware com a arquitetura proposta. O capítulo 5 contém os resultados experimentais do trabalho realizado, servindo como métricas para a viabilidade da arquitetura. No capítulo 6 são apresentadas as conclusões obtidas desse trabalho.

2 TRABALHOS RELACIONADOS

Em busca de elevada performance e eficiência energética, foco tem sido direcionado a aceleradores de hardware para realizar determinadas operações de propósito específico. Em (CASPER, OLUKOTUN, 2014) um acelerador em FPGA para aumentar o desempenho de operações em bancos de dados é descrito, obtendo um aumento considerável na *throughput* de sistemas desse tipo. (CHEN et al., 2016) propõe *Eyeriss* uma arquitetura reconfigurável para obter alto nível de paralelismo e baixo consumo energético em manipulação de redes neurais convolucionais densas. Um acelerador para aplicações em genômica chamado *Darwin* é proposto em (TURAKHIA et al, 2017). Operando simultaneamente em seis FPGAs, aplicações são até 125 vezes mais rápidas que software estado-da-arte da área.

As arquiteturas supracitadas que evidenciam o interesse e sucesso de aceleradores de hardware, são, porém, soluções de propósito específico, sua utilização é restrita a um grupo de funções implementadas em sua concepção. Aceleradores de propósito geral também existem, tal como em (NICKOLLS et al., 2008) onde CUDA é proposta, uma linguagem de programação para realizar aplicações paralelamente utilizando unidades de processamento gráfico compatíveis da NVIDIA, diferentemente do acelerador proposto ao longo desse trabalho, que se utiliza de uma linguagem funcional pura para descrever as aplicações a serem executadas.

Outros exemplos de aceleradores de hardware que se valem de arquiteturas diferentes das usuais podem ser citados, como em (GHICA, SMITH, SINGH, 2011) que traduz um subconjunto de funções recursivas para a descrição de um circuito que pode ser sintetizado para FPGA, de maneira a reduzir, quando comparado a processadores de propósito geral, o número de ciclos necessários para computar um resultado. Explorando paralelismo espacial e temporal das recursões em circuito é possível explorar *trade-offs* em área e tempo, escolhendo a que melhor se adequa à aplicação. (ETSION et al, 2010) é outro exemplo de arquitetura que busca explorar paralelismo em tarefas utilizando múltiplos núcleos como unidades funcionais, de maneira semelhante à ACQuA. O acelerador difere do aqui descrito por se valer de um modelo de programação adicional para linguagens imperativas e não utilizar linguagens funcionais.

Um acelerador de hardware que utiliza linguagem funcional é (NAYLOR, RUNCIMAN, 2012), que busca obter ganho em desempenho efetuando mais de uma redução semântica em um único ciclo. Funcionando também sobre uma plataforma de hardware reconfigurável, a solução proposta difere de ACQuA por explorar um grão mais fino de paralelismo, o das operações básicas das linguagens funcionais. O paralelismo explorado por (NAYLOR, RUNCIMAN, 2012) poderia ser integrado com o paralelismo em nível de funções aqui explorado, criando uma arquitetura com múltiplos níveis de paralelismo.

Para processadores de arquitetura multi-núcleo, existem APIs que procuram facilitar a obtenção de paralelismo tais como (OPENMP, 1997-2015) e (OPEN MPI, 2004) que permitem a execução paralela em trechos de código determinados pelo programador. Essas soluções, porém, necessitam da descrição direta dos mecanismos de sincronização para garantir o correto funcionamento do software desenvolvido e também familiarização com as construções de linguagem que explicitam o paralelismo e/ou diferentes linguagens de programação, dificultando, assim, o seu uso e aumentando o tempo de desenvolvimento e validação do software. Com o interesse de mitigar esse problema e ainda se baseando em software, (AUBREY-JONES, FISCHER, 2015) desenvolve um método de gerar código C++ com diretivas de Open MPI a partir de código escrito em uma linguagem funcional de alto nível, simplificando o trabalho do programador em obter paralelismo, cobrindo assim um dos objetivos de ACQuA, mas ainda se restringindo a arquiteturas multi-núcleo tradicionais.

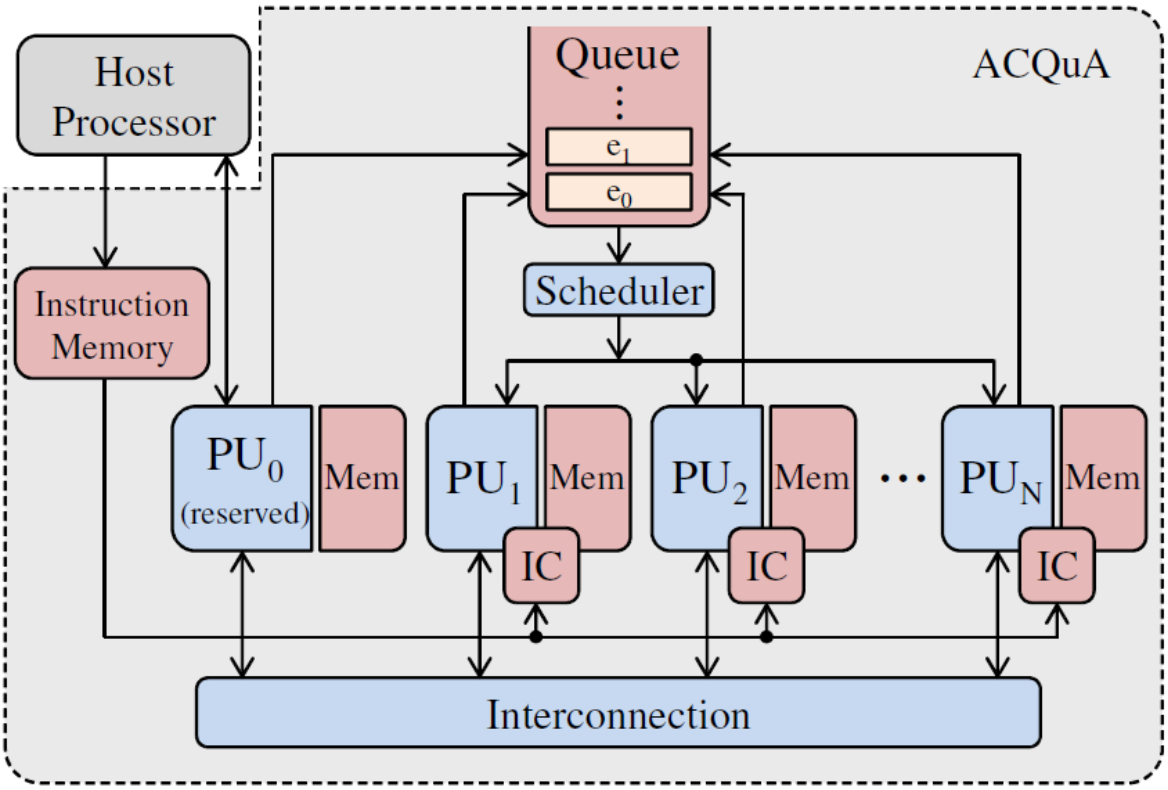
Esse trabalho descreve, portanto, a implementação de um acelerador de hardware de propósito geral que independe de diretivas explícitas de paralelismo de maneira a facilitar a produção de software e explorar execução paralela no mesmo de maneira transparente ao programador, uma vez que as poucas estruturas de sincronização necessárias são inseridas pelo compilador para a arquitetura ACQuA. O acelerador proposto busca, dessa maneira, atingir bom desempenho computacional para uma ampla família de aplicações utilizando-se do paralelismo inerente das linguagens funcionais.

3 ACQUA

3.1 Visão Geral

O acelerador ACQuA conta com alguns componentes de hardware que facilitam a execução das chamadas de função em paralelo. Os principais componentes são as múltiplas unidades de processamento e uma fila dedicada para as chamadas de função que são distribuídas pelas diferentes unidades por um *scheduler* dedicado, também em hardware. Uma visão geral da arquitetura pode ser vista na Figura 3.1.1.

Figura 3.1.1 – Visão Geral ACQuA



Fonte: Tanus, Nazar, Moreira (2017, p. 3)

Nota-se também a presença de uma interconexão entre as unidades de processamento e a presença de uma cache de instruções em cada unidade, além das suas memórias individualmente associadas. A fila, as unidades de processamento e a interconexão são componentes síncronos, i.e., trabalham na mesma frequência. A unidade de processamento reservada é responsável por iniciar o acelerador e retornar valores resultantes ao processador *host*. Deve, portanto, contar com capacidades de comunicação desnecessárias às demais unidades. O espaço de endereçamento é global de maneira a facilitar as transferências de dados entre as diferentes unidades de processamento.

3.2 Modelo de Funcionamento

ACQuA busca obter paralelismo através da paralelização de chamadas de função, i.e., toda função chamada em qualquer das unidades de processamento pode ser executada em outra das unidades de processamento presentes. É importante destacar que a unidade chamadora não espera o valor de retorno da função, a não ser que esse seja imediatamente necessário para a sua continuação e é assim que a arquitetura ganha em desempenho.

A execução de uma função em ACQuA conta, portanto, com uma série de estruturas de hardware e de dados que coordenam seu ciclo de vida, da sua criação ao seu fim, ou seja, seu eventual retorno e encaminhamento desse valor de retorno para a função chamadora. O hardware deve ser capaz de transferir parâmetros, resultados, endereços de função, endereços de retorno, dentre uma série de informações necessárias entre seus diferentes núcleos de processamento.

Ao ser criada uma função, uma estrutura de dados chamada *call record* é gerada que irá conter seus parâmetros e outros valores que serão detalhados posteriormente. Quando todos os parâmetros estão disponíveis dentro dessa estrutura, sua execução está pronta para ser iniciada. Nesse momento, o endereço do *call record* e o endereço de retorno da função são encapsulados em uma nova estrutura chamada de *queue entry*.

Uma *queue entry* é encaminhada (e nesse momento a unidade chamadora é livre para continuar sua execução) para uma estrutura de hardware externa à unidade de processamento que se encarrega de encaminhá-la para outro núcleo disponível no momento. Se nenhum estiver disponível a *queue entry* fica em espera. Estabelece-se um sistema de *First-In-First-Out* para a distribuição de *queue entries*. Quando um núcleo livre é encontrado, a estrutura de dados é enviada para o núcleo em questão que inicia a preparação necessária para a execução da função.

Primeiramente é criada uma nova estrutura de dados chamada *execution record*, que é a estrutura que acompanha a função até o fim de sua existência. Nela são armazenados alguns dados que são importantes para a correta execução da função que serão detalhados em seção posterior. De posse da *queue entry* e criado o *execution record* a cópia dos parâmetros da função, i.e., do *call record*, da memória da unidade chamadora para a memória interna do núcleo é o passo seguinte.

Com seus parâmetros disponíveis, a função está pronta para iniciar sua execução e, se necessário, realizar outras chamadas de função. Quando um valor de retorno de uma chamada de função ainda pendente é necessário para a continuação da execução, essa deve ser suspensa

até que o valor esteja disponível. Nesse caso o *execution record* é armazenado ainda dentro do núcleo para o resumo da sua execução quando todos valores pendentes estiverem disponíveis. Nesse ponto a unidade está liberada para receber novos *queue entries* ou resumir a execução de funções que já estão prontas para executar dentro do próprio núcleo (essas ficam armazenadas em uma fila interna). Executar funções pendentes é a operação que tem prioridade, i.e., quando uma unidade fica em estado ocioso, terminar funções mais antigas é considerado mais importante que começar funções novas.

Quando uma função atinge seu fim, o valor de retorno é encaminhado à unidade chamadora, que deve tratar de inserir a função pendente na fila interna de funções como pronta para executar, se esse for o último valor de retorno necessário. Caso contrário, essa operação apenas decrementa o contador de chamadas pendentes e a função continua em espera.

3.3 Exemplo

Nessa seção consideraremos a execução de Fibonacci recursivo definido em uma linguagem funcional que tem o código descrito na Figura 3.2.1:

Figura 3.3.1 – Código Fibonacci Funcional

```
fun fib ( x ) = if x <= 2 then 1
                else fib ( x - 1 ) + fib ( x - 2 )
```

Fonte: Tanus, Nazar, Moreira (2017, p.8).

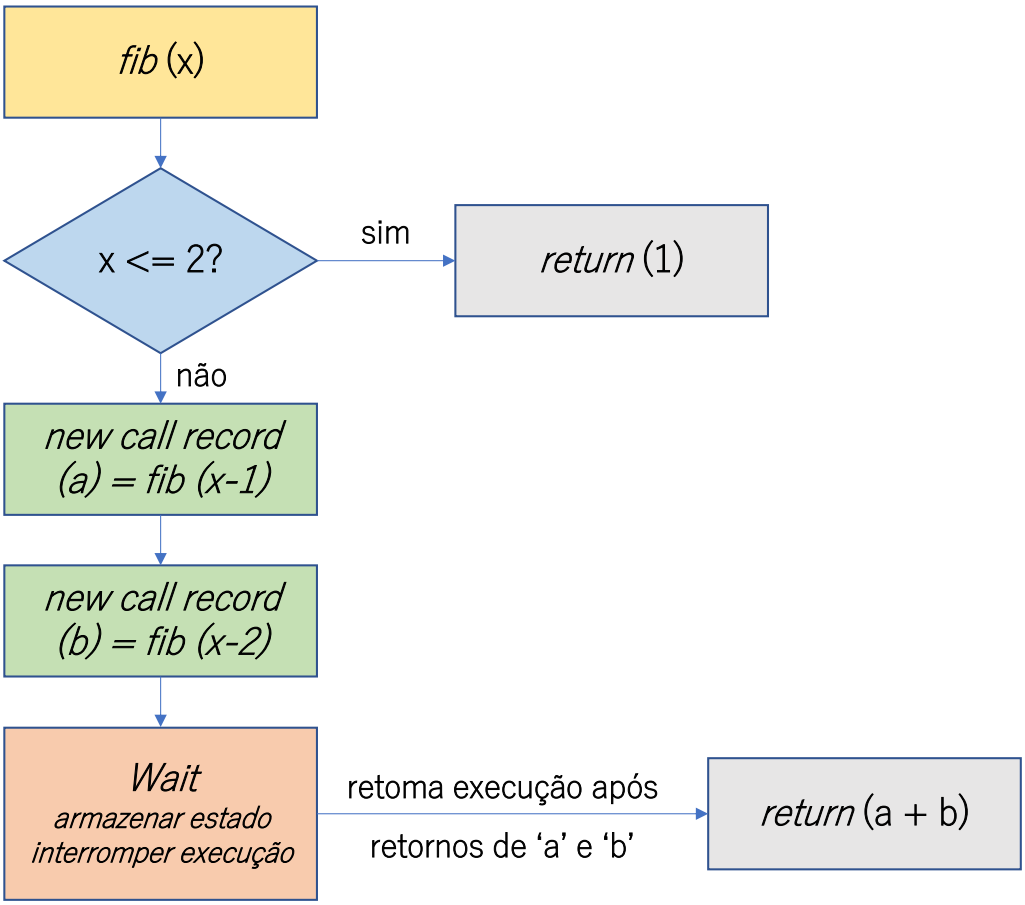
Para facilitar a compreensão do código e como funciona sua operação em ACQuA, foi criado o fluxograma da Figura 3.3.2, com a inserção das estruturas necessárias da arquitetura.

O primeiro passo, como em boa parte das funções recursivas, é checar a condição de término, que causa o fim da função e seu respectivo ramo da árvore de recursão quando a comparação tem resultado afirmativo. Caso contrário o que ocorre são duas chamadas de função (que em ACQuA correspondem à criação de *call records*). Quando os *call records* têm seus parâmetros computados e inseridos em sua estrutura, no caso '*x – 1'* para o primeiro e '*x – 2*' para o segundo, os mesmos são ativos e duas *queue entries* são geradas com os endereços de retorno apontados por '*(a)*' e '*(b)*'. A execução das funções chamadas será realizada em outras unidades de processamento.

Nesse ponto consta a diretiva *wait*, que é inserida automaticamente pelo compilador ACQuA e tem o importante papel de sinalizar para o suporte arquitetural que a função só deve continuar quando todos as chamadas de função pendentes estiverem concluídas, i.e., todos os

valores de retorno estão disponíveis. Lembrando que chamadas de função não são bloqueantes, essa diretiva se faz necessária para garantir o respeito às dependências de dados. O que ocorre é o armazenamento do *execution record* e a liberação da unidade de processamento para realizar outras atividades.

Figura 3.3.2 – Fluxograma Fibonacci em ACQuA



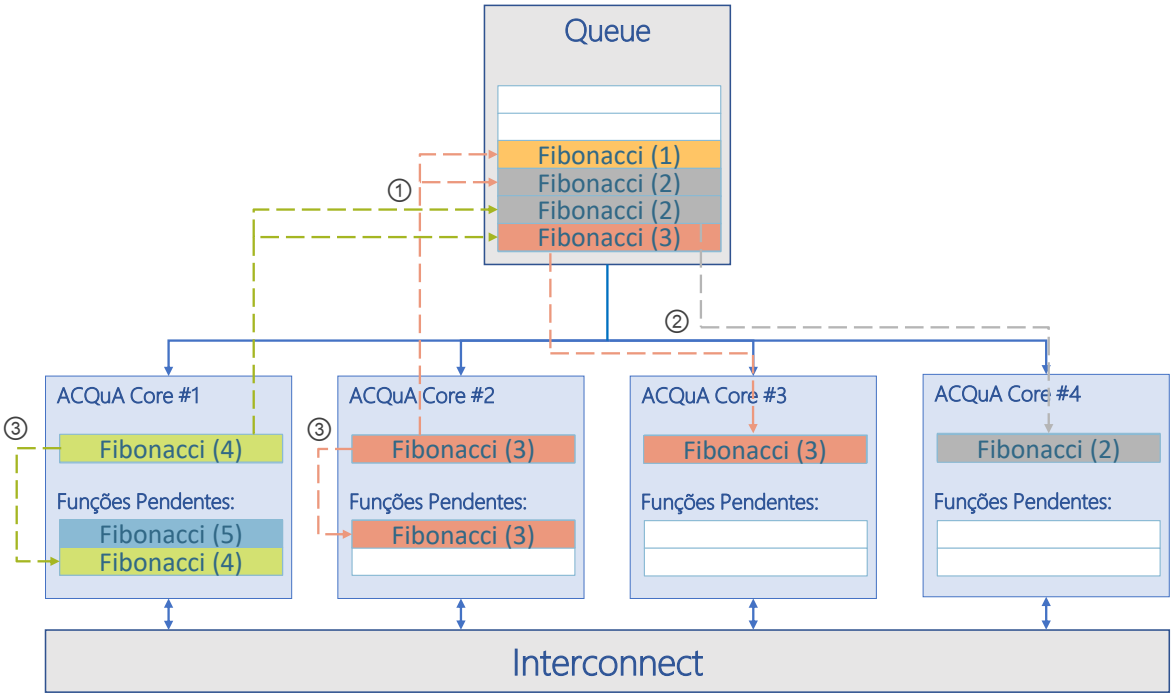
Fonte: o autor

Quando todos os valores estiverem disponíveis e a função estiver apta a resumir sua execução, ela o faz no ponto após o *wait*, que soma '*a*' e '*b*' e retorna esse valor à função chamadora, possivelmente possibilitando a sua continuação. Um exemplo da execução de Fibonacci em uma visão alto nível do acelerador pode ser consultado na Figura 3.3.3.

Em ① Fibonacci de 4 e 3 geram duas chamadas de função cada um e as mesmas são enviadas para a *queue* global. Em ② duas dessas chamadas de função são encaminhadas para os núcleos ociosos presentes no acelerador. Em ③ Fibonacci de 4 e 3, incapazes de continuar sua execução sem os valores de retorno das chamadas realizadas em ① são consideradas

pendentes (diretiva *wait*) e sua execução será retomada apenas quando os valores estiverem disponíveis. Nesse ponto os núcleos #1 e #2 são livres para executar novas funções.

Figura 3.3.3 – Exemplo Fibonacci em ACQuA Alto Nível



Fonte: o autor

É fornecida, dessa forma, uma noção do fluxo de operações necessárias para a execução de funções em paralelo na arquitetura e como é explorado tal paralelismo. A seguir serão discutidas as estruturas de dados utilizadas para possibilitar o funcionamento da mesma.

3.4 Estruturas de Dados

Para tornar possível a execução de funções em paralelo, a arquitetura conta com estruturas de dados bem definidas que são usadas pelo suporte de hardware para transferir parâmetros, endereços e valores de retorno, endereços de funções dentre outros dados que são importantes para a comunicação e sincronização entre unidades de processamento. Essas estruturas de dados são descritas a seguir.

3.4.1 Call Record

Um *call record* é a estrutura criada na memória de dados da unidade de processamento quando a mesma realiza uma chamada de função. É composta pelos seguintes campos:

<fn_addr, bindings, n_available, n_missing>

O campo *fn_addr* contém o endereço da função a ser executada, *bindings* contém os parâmetros da função, *n_available* contém o número de parâmetros disponíveis no Call Record e *n_missing* contém o número de parâmetros faltantes no Call Record da função. Uma função está pronta para ser executada quando seu *n_missing* é nulo, i.e., todos os parâmetros estão disponíveis.

3.4.2 Queue Entry

Uma *queue entry* é composta dos seguintes campos:

<cr_addr, ret_addr, isMap, mapParam>

Na qual *cr_addr* representa o endereço do *call record* associado a essa chamada de função, assim como *ret_addr* é seu endereço de retorno. Os campos *isMap* e *mapParam* são usados em manipulação de listas, nas quais um mesmo *call record* pode ser reutilizado por múltiplas chamadas de função, reduzindo o overhead de comunicação gerado pelos mesmos. O suporte a listas e funções do tipo *map* previstas em (TANUS, NAZAR, MOREIRA, 2017) são, nesse momento consideradas adições futuras à arquitetura.

3.4.3 Execution Record

Execution record é estrutura que acompanha uma função durante seu ciclo de vida, alguns de seus dados são uma combinação das estruturas anteriores:

<ret_addr, callCount, exeCtxt, env>

O campo *ret_addr* é copiado da *queue entry*, *callCount* é incrementado sempre que a função em execução realiza mais chamadas de função e decrementado sempre que a função recebe um valor de retorno, mantendo dessa forma uma contagem das chamadas de função pendentes. *exeCtxt* contém os registradores e contexto da unidade de processamento e *env* é a memória de dados usada pela função.

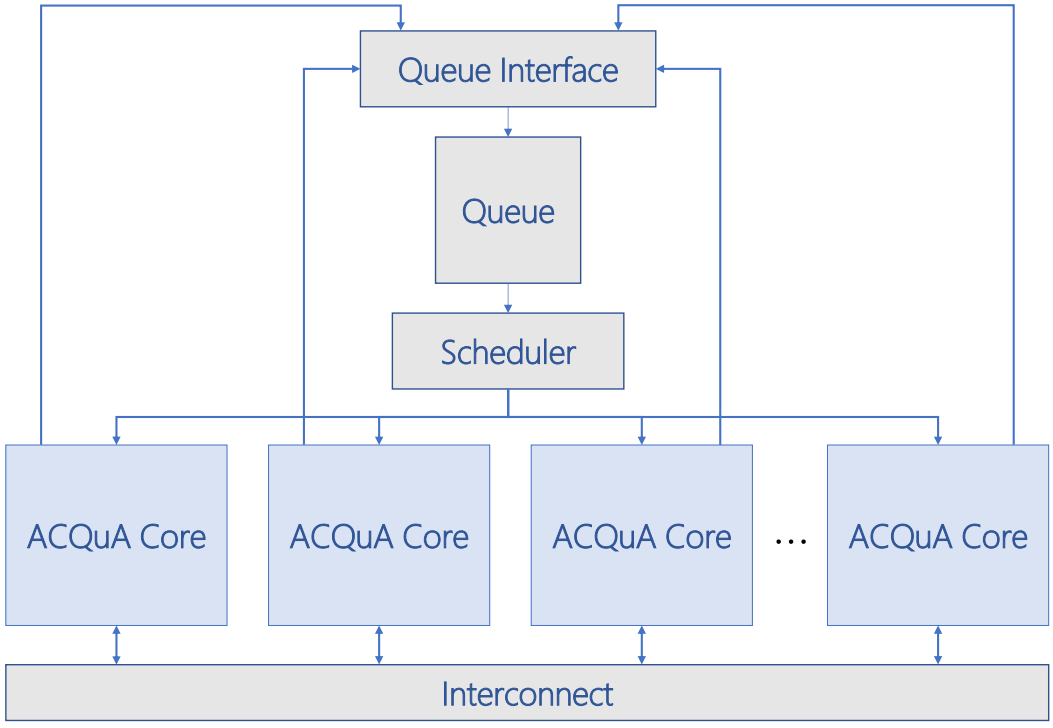
4 IMPLEMENTAÇÃO

Neste capítulo, será apresentada a implementação de ACQuA conduzida neste trabalho. Inicialmente será apresentada uma visão geral da arquitetura, seguida de uma descrição detalhada dos módulos que a compõem e das operações e diretivas executadas por cada módulo. Por fim, será descrito o fluxo de execução de uma chamada de função e do retorno de uma função, utilizando os módulos e diretivas previamente apresentados.

4.1 Visão Geral

De maneira análoga à definição da arquitetura, o acelerador desenvolvido tem as estruturas de hardware demonstradas na Figura 4.1.1.

Figura 4.1.1 – Visão Geral da Implementação



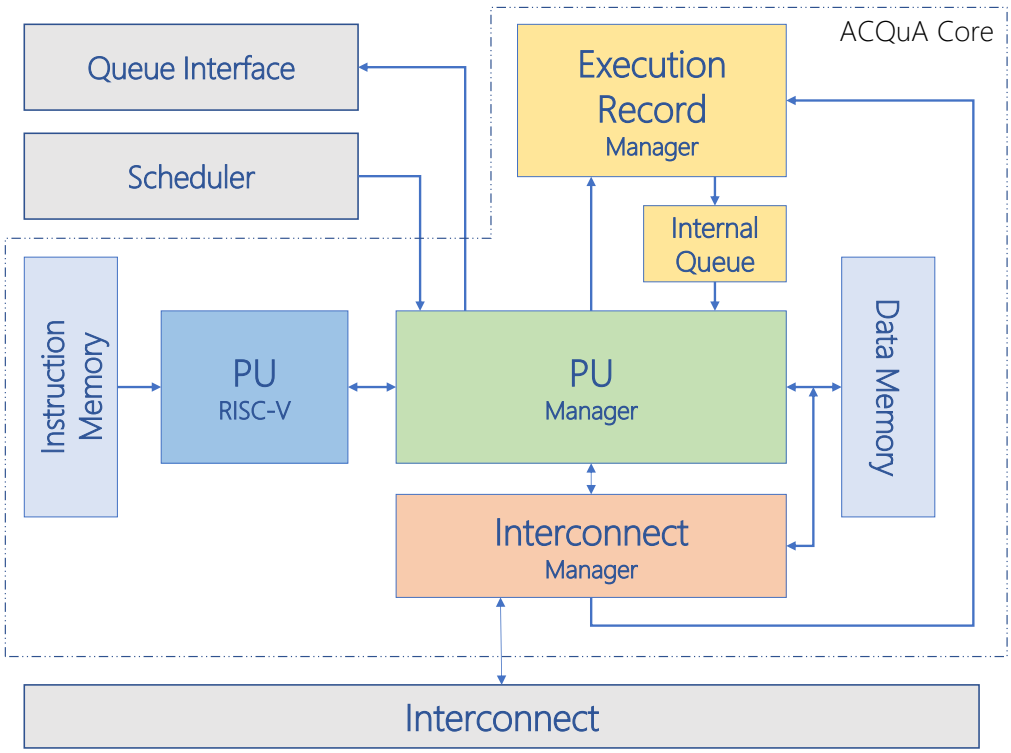
Fonte: o autor

Para o escopo desse trabalho foram omitidas a PU reservada e a interface com um processador *host*, visto que, inicialmente, o acelerador desenvolvido será utilizado apenas em simulação. Todos os núcleos ACQuA contam também com uma memória de instruções contendo as funções necessárias substituindo a cache de instruções e a memória de dados foi movida para dentro do núcleo. Uma *queue interface* foi adicionada com o objetivo de coordenar requisições de escrita simultâneas de múltiplas fontes na *queue* global. O código completo do acelerador compreende cerca de duas mil linhas de código descrito em VHDL, validado e

implementado utilizando as ferramentas de simulação e síntese da ferramenta da Xilinx, ISE 14.7 e foi feito de maneira totalmente parametrizável para qualquer tamanho de memórias e quantidade de núcleos.

O núcleo ACQuA foi modularizado de forma a separar as diferentes tarefas necessárias para o funcionamento da arquitetura. Seus respectivos módulos são detalhados a seguir na Figura 4.1.2.

Figura 4.1.2 – Núcleo ACQuA



Fonte: o autor

4.2 Processing Unit

A unidade de processamento desenvolvida para esse trabalho é um processador multiciclo da *ISA* de código aberto *RISC-V* (*RISC Five)*, uma arquitetura desenvolvida na *University of California, Berkeley*, com o propósito inicial de ser uma *ISA* de uso acadêmico e de pesquisa. Atualmente tem como objetivo tornar-se um padrão também para uso de implementações na indústria, sob o controle da *RISC-V Foundation* (WATERMAN et. al. 2016). O processador RISC-V foi totalmente desenvolvido no contexto deste trabalho, inicialmente com a intenção de incluir instruções específica para o funcionamento de ACQuA.

Porém, como será discutido a seguir, estas não foram necessárias e o núcleo permaneceu somente com as instruções já previstas na *ISA* padrão.

Tabela 4.2.1 – Instruções do Processador RISC-V

Aritméticas e Lógicas	Aritméticas e Lógicas (Imediato)	Load/Store	Controle de Fluxo
ADD	LUI	LB	JAL
SUB	AUIPC	LH	JALR
SLL	ADDI	LW	BEQ
SLT	SLTI	LBU	BNE
SLTU	SLTIU	LHU	BLT
XOR	XORI	SB	BGE
SRL	ORI	SH	BLTU
SRA	ANDI	SW	BGEU
OR	SLLI		
AND	SRLI		
	SRAI		

Fonte: Waterman et. al. (2016 p.54)

Para esse trabalho foi escolhida uma variação do conjunto de instruções definidas no *RV32I Base Instruction Set* cujas instruções disponíveis podem ser consultadas na Tabela 4.2.1. Foram omitidas as instruções do tipo FENCE usadas para controle de threads, visto que as mesmas não são necessárias nesse trabalho, bem como as do tipo CSSR, usadas para ler e escrever em registradores de sistema. A comunicação com o PU Manager, estrutura que coordena os fluxos de execução de um núcleo ACQuA, é realizada por meio de instruções do tipo *load* e *store* para endereços de memória pré-definidos. A unidade de processamento, portanto, não necessita de modificações em sua *ISA* para ser utilizada no acelerador, facilitando a sua substituição por outra unidade (que pode até contar com uma *ISA* diferente), conforme a necessidade da aplicação.

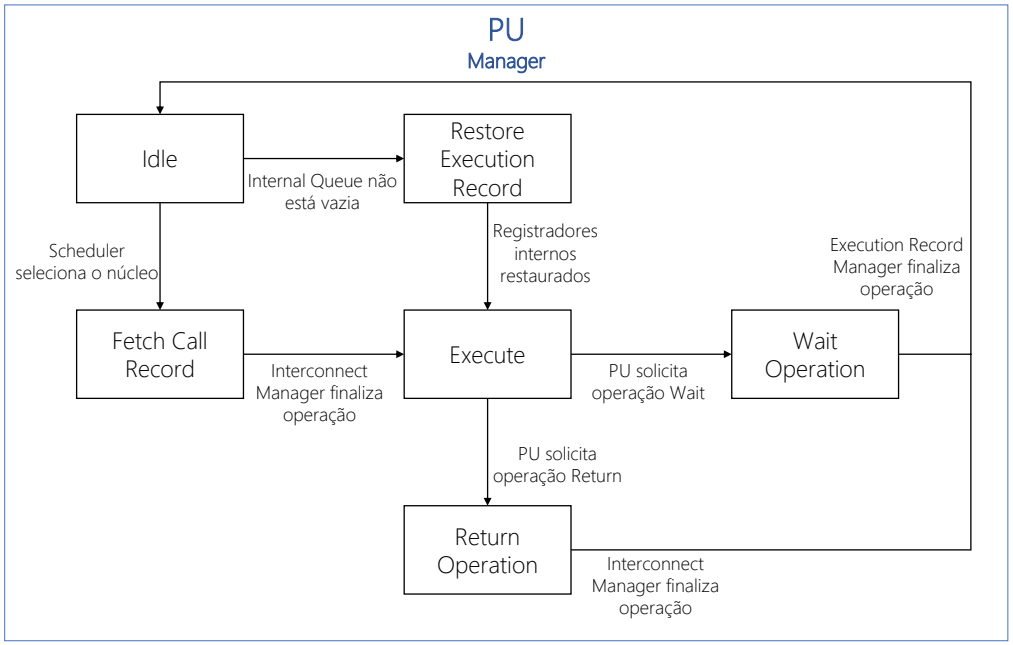
4.3 Processing Unit Manager

Essa unidade tem como papel realizar a interface com os diversos módulos contidos em um núcleo ACQuA, bem como com a *queue* global, local e com o *scheduler*. Além disso é a principal responsável por controlar o fluxo de execução das funções mediante o uso de uma máquina de estados finita, simplificadamente detalhada na Figura 4.3.1.

No diagrama constam as operações de recebimento de uma *queue entry* proveniente do *scheduler,* leitura da *queue* local, busca de *call record,* execução e as operações de *wait* e *return.* A unidade ainda é responsável por pausar o funcionamento da PU quando operações de

interconexão forem necessárias (como por exemplo enviar um *call record* para outro núcleo ou receber um valor de retorno) já que as mesmas fazem uso da memória de dados, tratando também de multiplexar a entrada da memória entre a unidade de processamento e o Interconnect Manager conforme a necessidade. Cada estado do diagrama e suas respectivas transições são brevemente discutidos a seguir.

Figura 4.3.1 – PU Manager – Diagrama de Operação



Fonte: o autor

- Idle

  Um núcleo sai do estado Idle quando é recebido um sinal de seleção do *scheduler*, que ocorre quando uma chamada de função é enviada para a *queue* por outro dos núcleos do acelerador. Uma outra possibilidade para o núcleo sair desse estado é uma entrada nova ser gerada na *queue* interna, significando que alguma das execuções pendentes recebeu todos seus valores de retorno e está pronta para resumir sua execução. Nota-se novamente que, na presença das duas entradas, uma proveniente do *scheduler* e outra da *queue* interna, é essa última que tem prioridade;

- Fetch Call Record

  Primeiro passo a ser realizado uma vez que uma nova função é designada para o núcleo pelo *scheduler*. O PU Manager se comunica com o Interconnect Manager para solicitar o *call record* da função que deve ser copiado do seu endereço de

origem para a memória de dados do núcleo. Ao fim dessa operação, a função, de posse dos seus parâmetros, está pronta para começar sua execução;

- Execute

  Nesse estado a unidade de processamento está executando uma função e existem quatro diferentes possibilidades que são tratadas nesse estado. A primeira é quando ocorre um pedido proveniente da interconexão, no qual um outro núcleo solicita um *call record* ou deseja enviar um valor de retorno para esse núcleo. Nesse caso a unidade de processamento tem sua execução temporariamente suspensa enquanto o Interconnect Manager utiliza a memória e realiza o tratamento da requisição. Após o término dessa operação, a execução continua.

  A segunda situação ocorre quando a unidade de processamento realiza uma chamada de função utilizando-se das diretivas: Call Record Address Write, Return Address Write, Call e Call Status Read, nessa ordem. Essas diretivas estão contidas na Tabela 4.3.1 e são tratadas em uma máquina de estados finita diferente da principal, específica para a manipulação dos registradores internos e comunicação com a *queue* global. O processo de chamada de uma função é visto e exemplificado em mais detalhes posteriormente nessa seção. As operações restantes são Return e Wait, contidas no diagrama e detalhadas a seguir;

- Restore Execution Record

  Consiste no estado que restaura os registradores da arquitetura para a execução da função pendente. Ao fim dessa operação, a unidade de processamento é habilitada para a execução;

- Return Operation

  Ocorre quando uma função atinge seu fim. A última instrução realizada é a diretiva Return, que, quando recebida pelo PU Manager, inicia uma operação de interconexão, solicitando ao Interconnect Manager o envio do valor de retorno para o endereço de retorno do núcleo apropriado. Após o fim dessa operação o núcleo retorna para o estado ocioso;

- Wait Operation

  Quando uma função necessita de resultados provenientes de outras chamadas de função é iniciado um fluxo de operações que tem como objetivo verificar se as

funções chamadas já retornaram seus valores mediante o uso de um Call Count Read e, em caso negativo, quando ainda houver funções pendentes, a unidade de processamento deve salvar seu contexto e se utilizar das diretivas da arquitetura Resume Address Write e Wait para suspender sua execução. Esse fluxo também é descrito em mais detalhes posteriormente nessa seção.

As diretivas disponíveis que operam sobre os registradores internos da arquitetura são as contidas e detalhadas na Tabela 4.3.1. Suas aplicações foram supracitadas na descrição dos estados de um PU Manager e os casos de uso, bem como sua ordem estão descritos nas Figuras 4.3.2, que demonstra o fluxo de uma chamada de função em diretivas ACQuA e 4.3.3 que demonstra uma operação Wait. Essas diretivas são instruções que devem ser automaticamente inseridas pelo compilador ACQuA de maneira a garantir o respeito das dependências de dados e do correto encaminhamento de chamadas de função para a *queue global*. O acionamento de qualquer uma destas diretivas é feito através de instruções de acesso à memória em endereços previamente estabelecidos, não sendo necessárias instruções específicas para estes propósitos, conforme mencionado anteriormente.

Tabela 4.3.1 – Diretivas ACQuA

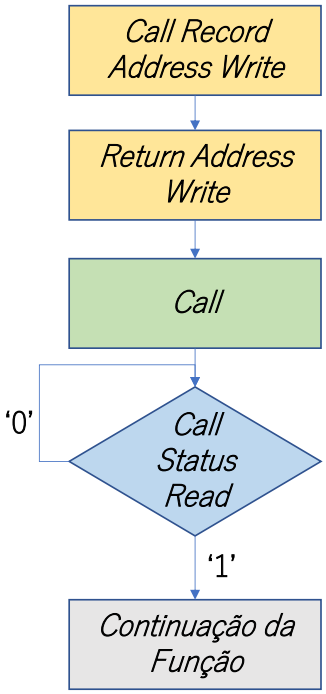
Diretiva	*Load/Store*	*Dado*	*Descrição*
Wait	Store	-	Para a execução se *call count* diferente de 0
Return	Store	Valor de Retorno	Retorna o dado para seu respectivo endereço/núcleo de retorno
Call	Store	-	Gera uma *queue entry* composta pelos registradores Call Record Address e Return Address e encaminha a mesma para a *queue* global
Call Count Read	Load	-	Lê o *call count* atual
Call Status Read	Load	-	Lê o estado da solicitação de envio da *queue entry* para a *queue* global (1 = *queue entry* enviada, 0 = *queue entry* pendente)
Return Address Write	Store	Endereço de Retorno	Escreve no registrador de endereço de retorno de chamada de função (Return Address)
Call Record Address Write	Store	Endereço de Call Record	Escreve no registrador de endereço de *call* record de chamada de função (Call Record Address)
Resume Address Write	Store	Endereço de Continuação da Função	Escreve no registrador que aponta o endereço na memória de instruções onde uma função retoma sua execução após o retorno de um *wait* (Resume Address)

Fonte: o autor

Em uma chamda de função o primeiro passo consiste em escrever o endereço do *call record* em um dos registradores da arquitetura, valendo-se da diretiva Call Record Address

Write. De forma análoga é registrado também o endereço de retorno da função com Return Address Write. Nesse ponto já se tem o necessário para gerar uma *queue entry* encapsulando ambos os endereços e enviá-la para a *queue* global, o que é feito com a diretiva Call. A operação seguinte é a leitura do registrador de estado da chamada, que tem valor ‘1’ quando a *queue entry* já foi armazenada na *queue* global e ‘0’ caso o contrário. Enquanto Call Status Read retornar um valor diferente de ‘1’, é necessário que a unidade de processamento continue executando essa operação para garantir o correto envio da chamada para a *queue* global.

Figura 4.3.2 – Fluxograma de Chamada de Função – Operações ACQuA



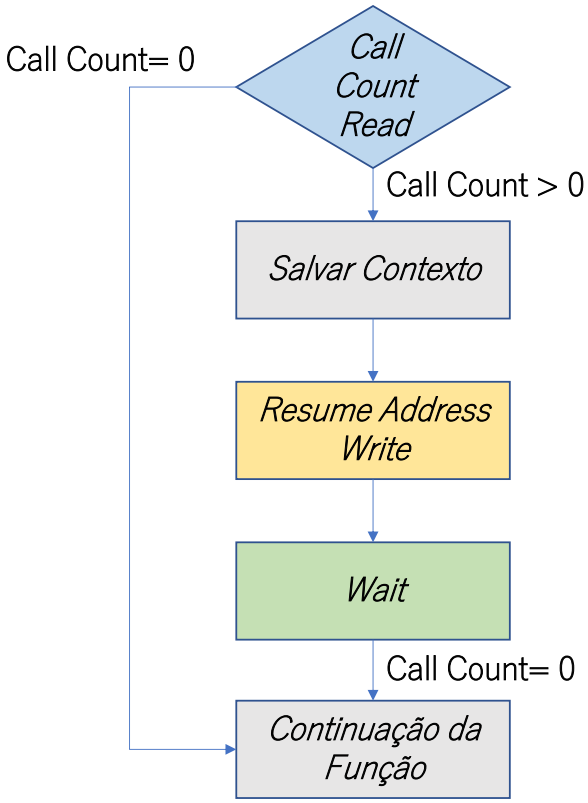
Fonte: o autor

Diretivas ACQuA também são utilizadas quando a função necessita de valores de retorno proveniente das funções chamadas para prosseguir sua execução. O fluxo de operações está contido na Figura 4.3.3.

Primeiramente é realizado um Call Count Read, que fornece à unidade de processamento a contagem de chamadas de função que ainda não retornaram. Se ainda houver resultados pendentes, i.e., call count é maior do que zero, o processador salva o seu contexto na memória de dados e executa um Resume Address Write, que armazena em um dos registradores da arquitetura o endereço no qual a função irá continuar a sua execução, i. e., o program counter, quando todos valores de retorno estiverem disponíveis. Finalmente é executado um Wait, que salva o execution record da função atual e para sua execução. É importante mencionar que as

eventuais condições de corrida que poderiam acontecer, como por exemplo o recebimento de um valor de retorno e decremento do call count enquanto o processador se encontra em meio ao processo descrito, o que poderia, no pior caso, causar que uma função ficasse para sempre esperando um valor de retorno, foram devidamente tratadas ao longo do código.

Figura 4.3.3 – Fluxograma de Operação Wait – Operações ACQuA



Fonte: o autor

Ao final de uma função ACQuA sempre há, também, uma operação de '*Return'*, que, como previamente discutido, informa ao PU Manager qual valor deve ser retornado para a unidade chamadora encaminhado pela interconexão.

Com isso estabelecem-se as diretivas que devem estar contidas após a compilação em código binário ACQuA, bem como sua respectiva ordem. O PU Manager recebendo, então, essas diretivas, tem o papel de encaminhar cada passo das requisições para as outras unidades do núcleo, detalhadas a seguir, conforme o necessário.

4.4 Execution Record Manager

O Execution Record Manager tem como principal objetivo manter um registro das funções que realizaram a operação *wait* e aguardam valores de retorno para continuar sua

execução. Possui uma tabela que contém os valores necessários para a atualização do *call count* das funções em espera, bem como sua eventual reativação, i. e., envio para a *queue* local quando a função já possuir todos seus valores de retorno. Uma entrada da tabela tem o seguinte formato:

<*Resume_Address, Return_Address, End_Address, Begin_Address, Call_Count, Valid*>

- Resume_Address – Endereço da memória de instruções no qual a função continuará sua execução (valor a ser atribuído ao *program counter* da unidade de processamento);
- Return_Address – Endereço de retorno da função;
- End_Address, Begin_Address – Primeira e última posições de memória usadas pela função enquanto executava. Ao receber um valor de retorno, o Interconnect Manager, além de armazenar o valor, envia também o endereço de retorno ao Execution Record Manager, que trata de percorrer as posições válidas da tabela procurando pela entrada que possua o respectivo endereço contido no intervalo formado por Begin_Address e End_Address. Quando encontrada, a mesma possui seu *call count* decrementado. Ao atingir um *call count* igual a zero, a entrada da tabela é eliminada e uma nova entrada é gerada na *queue* interna com os valores necessários para que a função continue sua execução, visto que agora todos seus valores de retorno já estão disponíveis;
- Call_Count – Quantidade de valores de retorno ainda necessários para que a função possa continuar sua execução;
- Valid – Indica se a entrada da tabela é válida.

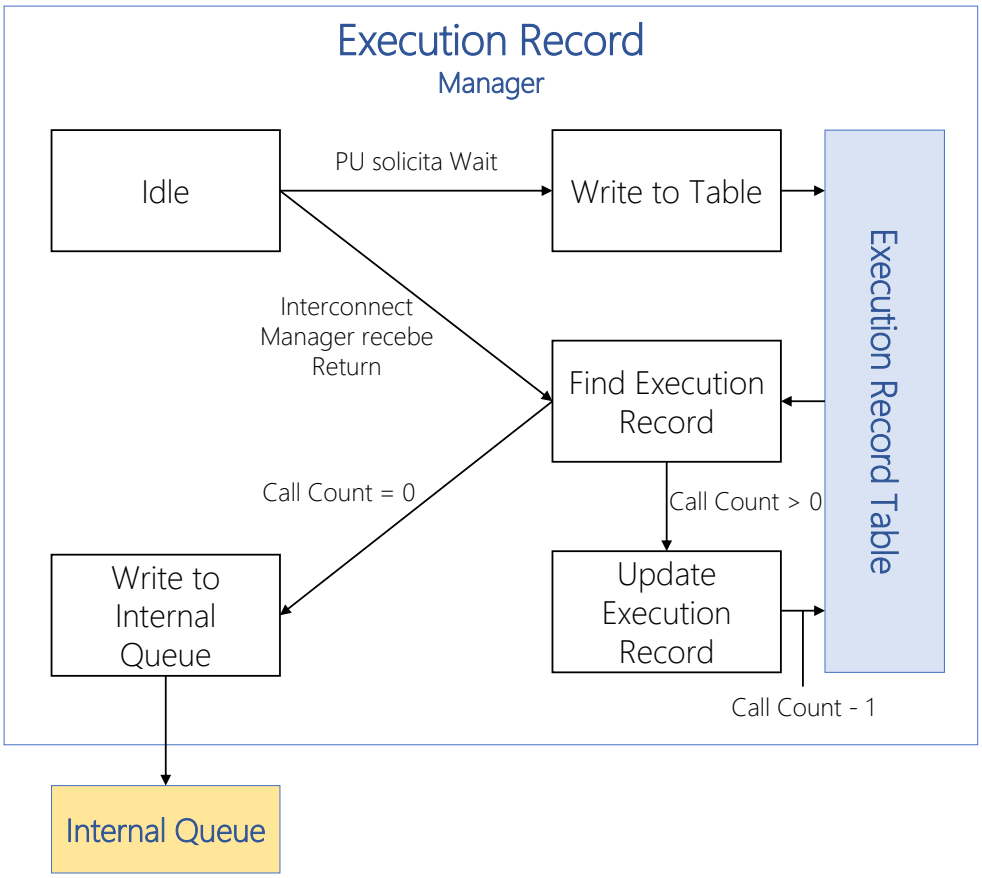
O funcionamento do Execution Record Manager é exemplificado pelo diagrama da Figura 4.4.1.

Escritas na tabela são realizadas mediante o uso da operação *wait*, que criará uma nova entrada na tabela na primeira posição não válida (Valid = 0). A cada valor de retorno obtido, o Interconnect Manager armazena o mesmo e envia o endereço de memória no qual foi realizada a escrita para o Execution Record Manager, que buscará na tabela a entrada correspondente para atualizar o *call count*.

Ao atingir o valor zero a entrada é transferida da Execution Record Table para a *queue* local, caso contrário o *call count* é decrementado e a entrada da tabela é atualizada. O valor de retorno pode também ser destinado à função atualmente em execução na unidade de

processamento. Nesse caso o PU Manager é notificado pelo Execution Record Manager e seu *call count* é decrementado.

Figura 4.4.1 – Execution Record Manager – Diagrama de Operação



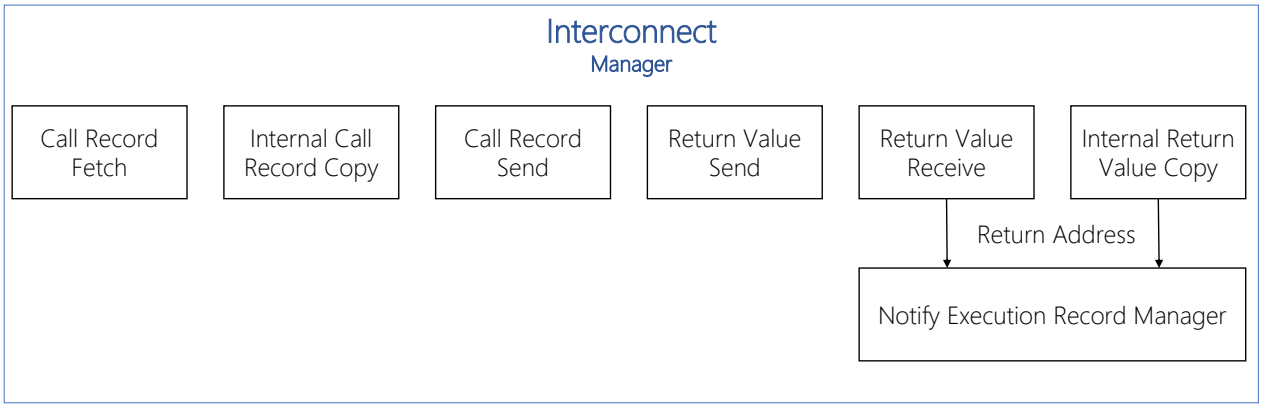
Fonte: o autor

4.5 Interconnect Manager

O Interconnect Manager é responsável por seis situações diferentes de transmissão/recepção de dados, sendo quatro casos entre diferentes núcleos e dois casos de cópias internas, quando o endereço de destino e origem estão no mesmo núcleo. A operações executadas estão descritas na Figura 4.5.1.

Cada situação compreende um número elevado de estados de uma máquina de estados finita que realiza o *handshake* necessário com o núcleo transmissor/receptor, solicita acesso à memória junto ao PU Manager e notifica o Execution Record Manager quando a operação se trata de um recebimento/cópia interna de valor de retorno.

Figura 4.5.1 – Interconnect Manager – Diagrama de Operação



Fonte: o autor

A explicação detalhada de cada caso é feita a seguir:

- Call Record Fetch

  Primeira operação de interconexão que uma chamada de função realiza, é solicitada ao Interconnect Manager pelo PU Manager. É enviada uma solicitação de *call record* ao núcleo chamador passando-se o endereço do mesmo (obtido da *queue entry*). Os dados recebidos pela interconexão são salvos na memória de dados local;

- Call Record Send

  Operação realizada quando o Interconnect Manager recebe uma solicitação de *call record*. É enviada ao PU Manager do núcleo uma solicitação de uso de memória, e, quando o mesmo envia de volta a permissão para usar a memória, é realizada a leitura dos dados do *call record* e seu envio pela interconexão é realizado;

- Return Value Send

  Última operação de interconexão que uma chamada de função realiza, é solicitada ao Interconnect Manager pelo PU Manager. É enviada uma solicitação de envio de valor de retorno para o núcleo chamador passando-se o endereço do mesmo (obtido do *execution record*) e o valor de retorno (obtido mediante a operação ACQuA Return);

- Return Value Receive

  Operação realizada quando o Interconnect Manager recebe uma solicitação de envio de valor de retorno. É enviada ao PU Manager do núcleo uma solicitação de uso de memória e, quando o mesmo envia de volta a permissão para usar a memória, o valor de retorno enviado é salvo no endereço solicitado. Notifica-se, também, o Execution

Record Manager, para decrementar o *call count* ou inserir a função na *queue* interna quando apta a executar (*call count* é zero);

- Internal Return Value Copy

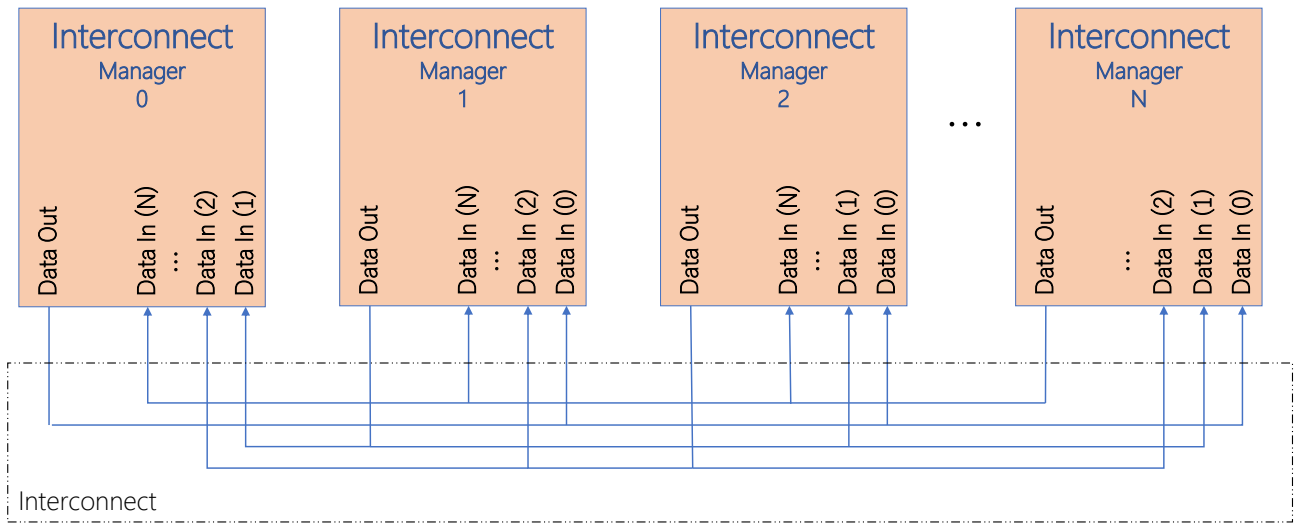
  É a combinação das operações de ‘Return Value Send’ e ‘Return Value Receive’. Ocorre quando o endereço de retorno se encontra na memória de dados do núcleo que solicita a operação de retorno, i.e., a unidade chamadora é a mesma unidade da execução. A cópia é realizada dentro do próprio núcleo e a interconexão não é utilizada. Nesse caso o Execution Record Manager também é notificado para que seja decrementado o *call count* ou realizada a inserção na *queue* interna;

- Internal Call Record Copy

  É a combinação das operações de ‘Call Record Send’ e ‘Call Record Receive’, ocorre quando o endereço do *call record* se encontra na memória de dados do núcleo que realizará a execução, i.e., o núcleo chamador é o mesmo que executará a função. A cópia é realizada dentro do próprio núcleo e a interconexão não é utilizada.

O modelo da interconexão de dados com outros núcleos é do tipo *crossbar*, que foi escolhido por ser o mais simples de implementar e contar com menor latência. Sua estrutura pode ser consultada na Figura 4.5.2. Foram omitidos os sinais de controle.

Figura 4.5.2 – Interconnect Manager – Modelo de Interconexão

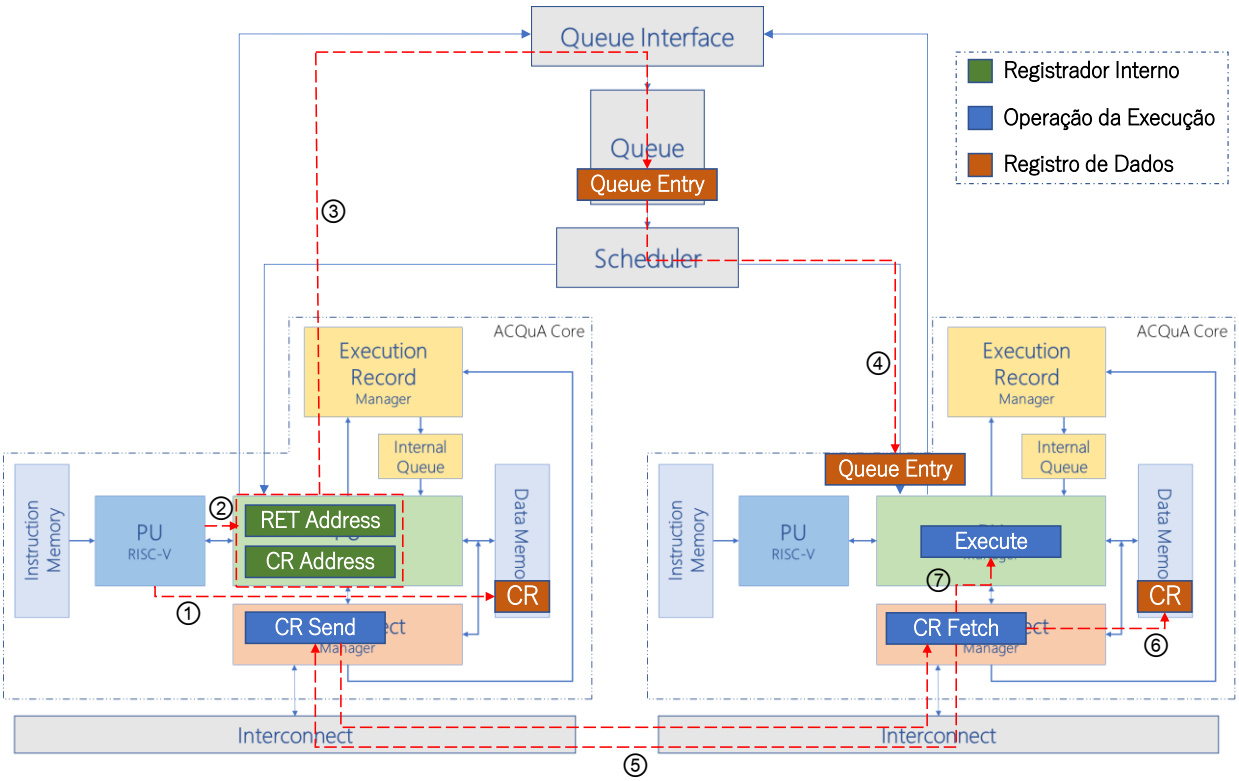


Fonte: o autor

4.6 Fluxo de uma Chamada de Função

Nessa seção é detalhado o processo de uma chamada de função passo a passo dentro do acelerador. Na Figura 4.6.1 estão demonstradas e numeradas as etapas que são descritas a seguir:

Figura 4.6.1 – Fluxo de uma Chamada de Função



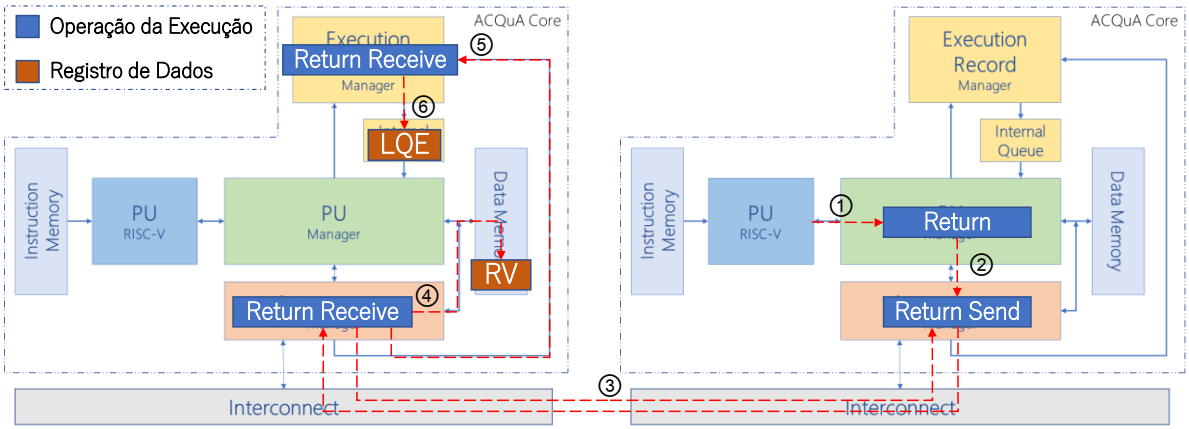
Fonte: o autor

1) Ao atingir uma chamada de função, a unidade de processamento gera um call record para a mesma em sua memória de dados (lembrando que um *call record* contém o endereço da memória de instruções da função a ser executada, o número de parâmetros disponíveis e faltantes da função, bem como os valores dos parâmetros);
2) Usando as diretivas Call Record Address Write e Return Address Write a unidade de processamento escreve nos registradores internos do PU Manager respectivamente o endereço do *call record* e o endereço no qual o valor de retorno deve ser armazenado em sua memória de dados;
3) A unidade de processamento executa, então, a diretiva Call. O PU Manager encapsula os registradores Call Record Address e Return Address atualizados no passo 2 em uma *queue entry*, que é então enviada para a queue global;
4) O *scheduler* encaminha a *queue entry* gerada para um núcleo disponível;

5) De posse da *queue entry*, o PU Manager solicita ao Interconnect Manager a busca do Call Record da função (Call Record Fetch). O núcleo chamador interrompe a execução da unidade de processamento (se o mesmo estiver atualmente executando uma função) e envia o *call record* (Call Record Send);
6) O *call record* é armazenado na memória de dados;
7) O PU Manager prepara o ambiente de execução e a função começa a execução.

4.7 Fluxo de um Retorno de Função

Figura 4.7.1 – Fluxo de um Retorno de Função



Fonte: o autor

Nessa seção descreve-se o processo que ocorre ao final de uma função, quando o seu valor de retorno deve ser propriamente enviado à unidade chamadora. Na Figura 4.7.1 estão demonstradas e numeradas as etapas que são descritas a seguir:

1) A unidade de processamento utiliza a diretiva Return, fornecendo ao PU Manager o valor de retorno e sinalizando ao mesmo que esse valor deve ser encaminhado para o núcleo chamador;
2) O PU Manager solicita ao Interconnect Manager que envie o valor de retorno;
3) O Interconnect Manager envia para o Interconnect Manager do núcleo chamador uma solicitação de envio de valor de retorno (Return Send). O núcleo chamador interrompe a execução da sua unidade de processamento (se o mesmo estiver atualmente executando uma função) e começa a operação de recebimento de valor de retorno (Return Receive);
4) O Interconnect Manager salva na memória de dados o valor de retorno recebido;

5) O Interconnect Manager notifica ao Execution Record Manager que um valor de retorno foi recebido e fornece o endereço onde o mesmo foi armazenado;

6) De posse do endereço do retorno recebido, o Execution Record Manager procura em sua tabela a qual função pendente o mesmo pertence e decrementa seu *call count*. Caso o mesmo tenha o valor zero, significando que a função pendente está pronta para executar, uma entrada é gerada na *queue* local do núcleo (Local Queue Entry).

5 RESULTADOS EXPERIMENTAIS

Essa seção analisa a escalabilidade da arquitetura em termos de área, frequência de operação e tempo de execução (*speedup*), e sua viabilidade para aplicações reais. Simulações foram realizadas para aceleradores contendo 2, 4, 8 e 16 núcleos ACQuA, tendo como plataforma alvo o FPGA Kintex-7 XC7K160T-2FBG676 da fabricante Xilinx. Os dados aqui fornecidos para área e frequência são obtidos da ferramenta Xilinx ISE 14.7 enquanto que o tempo de execução é proveniente do Xilinx ISE Simulator (ISim). O código desenvolvido e os *test benches* utilizados podem ser encontrados em https://github.com/tbtaborda/acqua-accelerator.

5.1 Área

Na Tabela 5.1.1 há o tamanho definido para as memórias utilizadas. A *queue* interna, global e a tabela de *execution records* têm largura variável conforme o número de núcleos utilizados, pois há variação no tamanho do endereço das memórias de dados, visto que o endereçamento é global.

Tabela 5.1.1 – Tamanho de Memórias

Memória	Largura (2 núcleos)	Largura (4 núcleos)	Largura (8 núcleos	Largura (16 núcleos)	Profundidade
Instruction Memory	32				65536
Data Memory	32				65536
Queue	34	36	38	40	1024
Internal Queue	65	66	67	68	1024
Execution Record Table	78	79	80	81	256

Fonte: o autor

Na Tabela 5.1.2 consta o número de componentes utilizados pelos módulos cujos tamanhos independem da quantidade de núcleos presentes no acelerador.

Tabela 5.1.2 – Área – Módulos de Tamanho Constante

Módulo	Quantidade de Componentes		
	Slice Registers	LUTs	BRAM
PU	1201	1600	0
PU_Manager	222	413	0
Execution Record Manager	292	1977	2

Fonte: o autor

A Tabela 5.1.3 contém a área ocupada pelo Interconnect Manager, a área total do núcleo e a área total do acelerador, para configurações de 2, 4, 8 e 16 núcleos.

Tabela 5.1.3 – Área – Módulos de Tamanho Variável

Módulo	Núcleos	Quantidade de Componentes		
		Slice Registers	LUTs	BRAM
Interconnect Manager	2	191	496	0
ACQuA Core		1920	4692	124
ACQuA Accelerator		4017	9379	248
Interconnect Manager	4	198	560	0
ACQuA Core		1930	4741	124
ACQuA Accelerator		7922	18956	495
Interconnect Manager	8	211	715	0
ACQuA Core		1943	4899	124
ACQuA Accelerator		15780	39248	990
Interconnect Manager	16	236	973	0
ACQuA Core		1980	5188	124
ACQuA Accelerator		31946	82355	1978

Fonte: o autor

Nota-se, portanto, que a área aumenta de maneira quase linear com o número de núcleos, o que é um resultado bom e dentro do esperado, significando que a escalabilidade da arquitetura e da implementação não é inviável quanto ao custo em área.

5.2 Frequência

As frequências de operação dos diferentes módulos estão descritas na Tabela 5.2.1, obtidas para a versão de 16 núcleos da arquitetura. Não há grande variação na frequência causada por mudanças no número de núcleos, a versão de 2 núcleos do acelerador tem frequência 134.628 MHz, um acréscimo de 3.3% em relação à frequência do acelerador de 16 núcleos.

Tabela 5.2.1 – Frequência de Operação

Módulo	Frequência
PU	211.376 MHz
PU_Manager	314.921 MHz
Execution Record Manager	251.375 MHz
Interconnect Manager	372.024 MHz
ACQuA Core	137.013 MHz
ACQuA Accelerator	130.322 MHz

Fonte: o autor

5.3 Tempo e Speedup

Na Tabela 5.3.1 consta o tempo necessário para cada configuração executar a função Fibonacci descrito na seção 2 com ‘x = 8’ e ‘x = 10’ em simulação.

Tabela 5.3.1 – Tempo de Execução

Número de Núcleos	Tempo (Fibonacci 8)	Tempo (Fibonacci 10)
2	53135 ns	172915 ns
4	27415 ns	77935 ns
8	15395 ns	37095 ns
16	12135 ns	23205 ns

Fonte: o autor

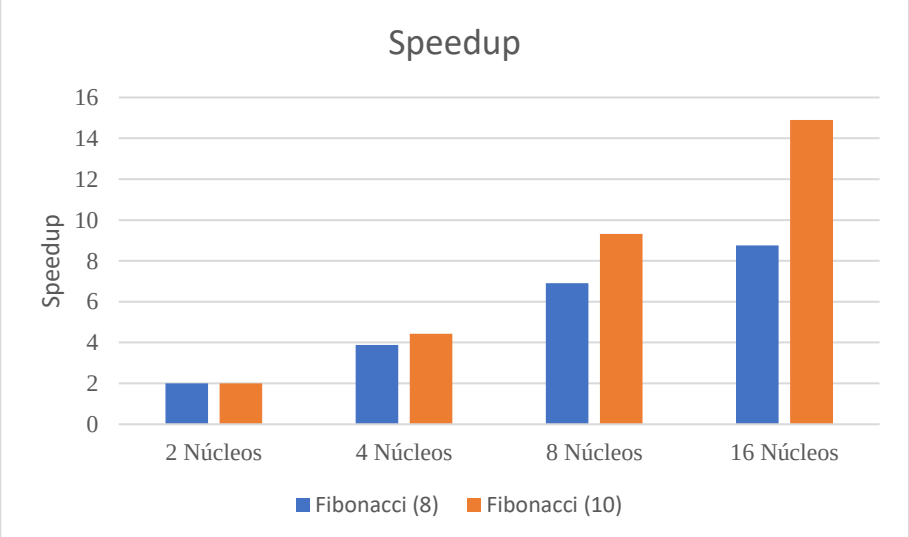
A Figura 5.3.2 contém o resultado obtido para o cálculo de *speedup* das funções simuladas. Atentar-se para o fato de que o acelerador desenvolvido, no momento, é funcional para número de núcleos a partir de 2. De maneira a normalizar o cálculo à noção que normalmente se espera de *speedup*, ou seja, tendo o tempo sequencial de processamento como referência, a versão de 2 núcleos do acelerador teve seu *speedup* normalizado para o valor 2, i.e., o cálculo do *speedup* é feito mediante a fórmula da Figura 5.3.1, tal que N é o número de núcleos.

Figura 5.3.1 – Fórmula Speedup

$$Speedup(N) = \frac{Tempo\ (2)}{Tempo(N)} * 2$$

Fonte: o autor

Figura 5.3.2 – Gráfico Speedup



Fonte: o autor

Nesse caso observa-se um *speedup* também quase linear (N para N) com o número de núcleos (e melhor que o linear em alguns casos). Fibonacci de 8 para o acelerador de 16 núcleos obteve *speedup* quase idêntico ao de 8 núcleos por se tratar de um curto tempo de execução e boa parte do mesmo ser gasto em operações de interconexão em ambos aceleradores.

É notável que, em todas configurações, aumentando-se o tamanho da entrada de 8 para 10, aumenta-se também o *speedup*. Significando que, quanto mais paralelismo é oferecido, melhor a arquitetura se comporta. O acelerador é capaz de enfileirar mais chamadas de função na *queue* global e escaloná-las para núcleos ociosos. Essas operações são muito rápidas, a escrita na *queue* pode ser realizada em dois ciclos, a operação de *scheduling* demora três ciclos quando há um núcleo imediatamente disponível (caso contrário espera-se até que um núcleo esteja novamente ocioso). As operações de retorno e atualização dos *execution records*, ao contrário, é mais demorada pois envolve a busca em uma tabela, atualização de *call counts* e eventual escrita na *queue* interna. Essas operações acontecem, porém, sem interromper a execução da função no núcleo. É provável, portanto, que para Fibonacci com entrada 10, mais operações desse tipo ocorram enquanto a unidade de processamento executa uma função, quando comparado a Fibonacci com entrada 8, no qual as mesmas ocorrem com o núcleo ocioso.

Outro resultado notável é que o *speedup* de N é maior do que o próprio valor de N quando N = 4 e N = 8. Isso ocorre porque, com mais núcleos, o número médio de funções pendentes por núcleo é reduzido, i. e., ficam melhor distribuídas no acelerador e, consequentemente, as tabelas de *execution records* contém menos entradas, reduzindo o tempo necessário para busca nas mesmas. Aumentando, dessa forma, o desempenho acima do limite esperado de N. Destaca-se também que os resultados aqui obtidos estão de acordo com os observados em (TANUS, NAZAR, MOREIRA, 2017), mediante o uso de um simulador mais abstrato da arquitetura.

6 CONCLUSÕES

Nesse trabalho foram apresentadas uma breve descrição da arquitetura ACQuA, seu funcionamento básico e a motivação por trás de sua criação. Foi demonstrada, também, a implementação de um acelerador de hardware usando essa arquitetura e a estrutura interna dos módulos criados para viabilizar seu funcionamento. O acelerador desenvolvido serve de importante mecanismo de validação da arquitetura, facilitando a obtenção de resultados e dados de simulação, bem como a sua área, frequência de operação e tempo total de execução.

De posse dos resultados experimentais apresentados, é possível afirmar que, para aplicações suficientemente paralelizáveis (como cálculo de Fibonacci) o desempenho do acelerador aumenta linearmente com o número de núcleos, embora melhorias em estruturas de hardware possam ser propostas para melhorar o desempenho. Quanto à escalabilidade, o aumento em área também é linear em relação ao número de núcleos. A arquitetura é, portanto, viável para essas aplicações e seu uso é benéfico em termos de ganho de desempenho.

Trabalhos futuros podem ser realizados de forma a melhorar o desempenho e a facilidade de uso do acelerador aqui proposto, tais como:

- Automatizar geração de código

  O código de Fibonacci executado nesse trabalho foi desenvolvido em código Assembly para RISC-V e as operações de sincronização ACQuA foram manualmente inseridas. É interessante, portanto, que o processo de geração de código seja automatizado. Já existe um compilador para código de uma linguagem funcional para instruções imperativas em uma linguagem intermediária (TANUS, NAZAR, MOREIRA, 2017). É necessário introduzir um tradutor desta linguagem intermediária para a representação binária final do RISC-V utilizada nesse trabalho;

- Suporte completo à arquitetura ACQuA

  Nesse trabalho não foram implementadas funções de alta ordem, suporte às funções do tipo *map* e suporte a listas. Essas funcionalidades estão previstas na arquitetura e sua inclusão pode afetar positivamente o desempenho do acelerador;

- Melhorias nas estruturas de hardware

  Algumas estruturas de hardware utilizadas podem ser revistas de maneira a melhorar o desempenho geral da arquitetura. Exemplo disso é o Interconnect Manager, que atualmente possui entrada sequencial e saída combinacional, sua conversão em puramente sequencial pode afetar positivamente a frequência, diminuindo o caminho crítico do acelerador. O Execution Record Manager também pode ser melhorado. A busca que ocorre em sua tabela sempre que é recebido um valor de retorno pode se tornar um gargalo para um elevado número de entradas. Isso pode ser mitigado com o uso de uma busca do tipo *hash* que restrinja a seção a ser buscada dentro da tabela mediante o envio de alguns *bits* adicionais em conjunto com o endereço de retorno de uma função.

REFERÊNCIAS

AUBREY-JONES, T.; FISCHER, B.; **Synthesizing MPI Implementations from Functional Data-Parallel Programs**. International Journal of Parallel Programming archive. v. 44. n. 3. p, 552-573. Junho 2016.

CASPER, J.; OLUKOTIN.; K. **Hardware acceleration of database operations**. In: FPGA'14 Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays, 2014. p. 151-160.

CHEN, Y.; KRISHNA, T.; EMER, J. S.; SZE, V. **Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks**. In: Solid-State Circuits Conference (ISSCC), 2016 IEEE International, 2016.

ETSION, Y.; CABARCAS, F.; RICO, A.; RAMIREZ, A.; BADIA, R. M.; AYGUADE, E.; LABARTA, J.; VALERO, M. **Task Superscalar: An Out-of-Order Task Pipeline**. In: MICRO '43 Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. p. 89-100. 2010.

GABRIEL, E.; FAGG, G. E.; BOSILCA, G.; ANGSKUN, T.; DONGARRA, J. J.; SQUYRES, J. M.; SAHAY, V.; KAMBADUR, P.; BARRET, B.; LUMSDAINE, A.; CASTAIN. R. H.; DANIEL, D. J.; GRAHAM, R. L.; WOODWALL, T. S. **Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation**. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, Setembro 2004.

GHICA, D. R.; SMITH, A.; SINGH, S. **Geometry of Synthesis IV**, Compiling Affine Recursion into Static Hardware. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. p. 221-233. 2011.

HAMMOND, K. **Why parallel functional programming matters: Panel statement**. In: Reliable Software Technologies-Ada-Europe 2011. Springer. p. 201–205.

HUDAK, P. **Conception, evolution, and application of functional programming languages**. ACM Computing Surveys (CSUR). v. 21. n. 3. p. 359-411. Setembro 1989.

NAYLOR, M.; RUNCIMAN, C. **The Reduceron reconfigured and re-evaluated.** Journal of Functional Programming archive. v. 22. n. 4-5. p. 574-613. Agosto 2012.

NIKKOLS, J.; BUCK, I.; GARLAND, M.; SKADRON, K. **Scalable Parallel Programming with CUDA**. Queue - GPU Computing. v. 6 n. 2. p. 40-53. Março/Abril 2008.
OpenMP Application Programming Interface. Disponível em: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. Acesso em 09/07/2017.

TANUS, F.; NAZAR G.; MOREIRA, A. **Parallelization of Function Applications with the ACQuA Architecture.** Relatório técnico, 2017.

TURAKHIA, Y.; ZHENG, K. J.; BEJERANO, G.; DALLY, W. J. **Darwin: A Hardware-acceleration Framework for Genomic Sequence Alignment**. Disponível em: https://doi.org/10.1101/092171. Acesso em 09/07/2017.

WATERMAN, E.; ASANOVIC, K. **The RISC-V Instruction Set Manual**, **Volume I: User-Level ISA, Document Version 2.2**. RISC-V Foundation, May 2017. Disponível em: https://riscv.org/specifications/. Acesso em: 09/07/2017.