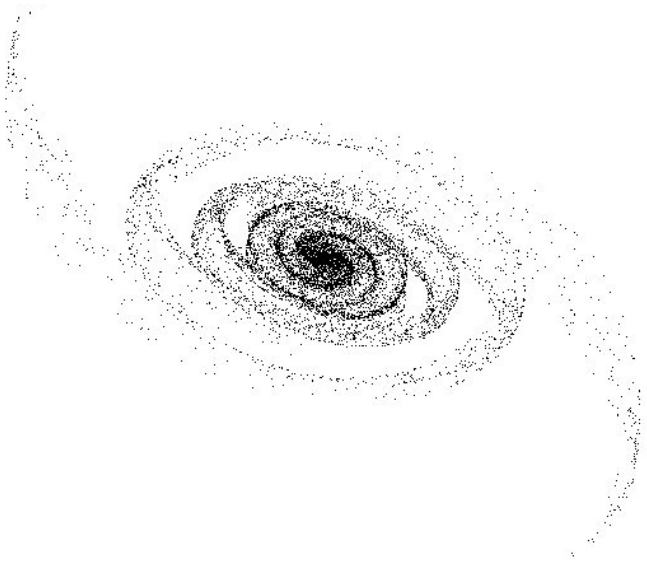
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA
SERVIÇO DE DOCUMENTAÇÃO E PUBLICAÇÕES

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA
Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores



Sistema de Multiprocessamento para Simulação de *N*-corpos em FPGA

Ricardo Joel Martins Pereira
(Licenciado em Engenharia Eletrónica e Telecomunicações e de Computadores)

Trabalho Final de Mestrado para Obtenção do Grau de Mestre em Engenharia de Electrónica e Telecomunicações

Orientador:
Professor Mário Pereira Véstias

Júri:
Presidente: Professor Fernando Manuel Ascenso Fortes
Arguente: Professor Horácio Cláudio de Campos Neto

Dezembro de 2013

Agradecimentos

Gostaria de agradecer ao meu orientador, Dr. Mário Véstias, por me ter encaminhado neste projeto, discutindo e dando conselhos quanto à sua forma, e ter gentilmente disponibilizado kits de desenvolvimento para a implementação do mesmo.

Gostaria de agradecer também ao Dr. António Serrador e ao Eng. Duarte Carona, membros do Grupo de Investigação em Electrónica e Sistemas de Telecomunicações, por me terem permitido trabalhar no laboratório de Investigação e Desenvolvimento e terem cedido material utilizado na realização do projeto. Agradeço ainda aos meus colegas Fábio Cardoso, Filipe Palhinha e Ana Gonçalves por me terem acompanhado no trabalho e pela troca de ideias que realizámos.

Agradeço aos meus pais, Emília e Fernando, pelo carinho e suporte que me deram. Sem o seu esforço e trabalho não me seria possível alcançar e terminar esta etapa. Foram eles que tornaram isto possível. Agradeço ao meu irmão, Fábio, por ser um companheiro em todo o meu percurso de vida. Agradeço à minha namorada, Bruna, por me ter acompanhado e incentivado na realização neste trabalho.

Resumo

Este projeto tem como objetivo o desenvolvimento de uma arquitetura de multiprocessamento dedicada à simulação de sistemas baseados na interação entre *N*-corpos. O sistema é desenvolvido numa placa de desenvolvimento com *FPGA*, tendo por base módulos lógicos dedicados ao cálculo das expressões base da simulação.

Escolhido como problema exemplo a evolução de um sistema de interação gravítica de *N*-corpos, são analisados os algoritmos de obtenção do valor das forças envolvidas e algoritmos de integração numérica, tendo em vista obter uma descrição do peso computacional e precisão dos mesmos.

São analisados os recursos presentes numa *FPGA* de forma a perceber a sua influência na implementação dos circuitos lógicos. São também analisados tipos de representações numéricas para entender qual melhor se adapta ao problema em causa.

Numa segunda fase é analisado o compromisso entre a precisão numérica e a área dos núcleos a desenvolver. Esta análise permite, dada uma *FPGA* com uma determinada quantidade de recursos internos, obter uma estimativa da quantidade de núcleos de processamento realizáveis.

Numa terceira e última fase, é criada uma arquitetura de multiprocessamento com módulos dedicados, onde é executado o algoritmo. Uma análise do sistema completo permite obter uma descrição dos recursos utilizados pelo sistema, bem como da taxa de cálculo final.

O sistema desenvolvido é modular e configurável em vários parâmetros internos, entre os quais o formato numérico utilizado, obtendo-se um sistema adaptável às necessidades do utilizador. É utilizada a memória interna da *FPGA* para o armazenamento dos dados de simulação. A interface com o exterior é uma porta série, que é uma interface genérica conhecida com bastantes aplicações de utilização já desenvolvidas.

É atingida uma capacidade de cálculo média de 5.3 *GFLOPS*, mas esta pode ser aumentada utilizando uma *FPGA* de maior capacidade que a usada no desenvolvimento deste trabalho, uma Virtex-4 XC4VSX35. O sistema é também facilmente portável para famílias mais recentes de *FPGA*, possibilitando taxas de cálculo ainda superiores. Para uma *FPGA* XC7VX980T, da família Virtex-7, a capacidade de cálculo estimada é de 49,5 *GFLOPS*.

Palavras-chave

Simulação de N-corpos, núcleo gravitacional, unidade de cálculo com *pipeline*, *FPGA*

Abstract

This project aims to develop a multiprocessing architecture dedicated to simulate *N*-body interaction based systems. The system is developed on a FPGA development board, having as base logic modules dedicated to the calculus of the base expressions of the simulation.

Chosen as example problem the evolution of a gravitational interaction *N*-body system, algorithms for obtaining the value of involved forces and numerical integration algorithms are analyzed, having in mind obtaining a computational weight and accuracy description of those.

The resources present in a FPGA are analyzed as a way to understand their influence on logic circuits implementation. Numeric representation types are also analyzed to understand which one best fits the problem at hand.

On a second phase the compromise between numerical precision and the area of the cores to develop is analyzed. This analysis allows to, given a *FPGA* with a certain amount of internal resources, obtain an estimate of the amount of processing cores realizable.

In a third and final phase, a dedicated module multiprocessing architecture, in which the algorithm is executed, is created. An analysis of the complete system allows for a description of used resources, as well as of the final calculation rate.

The developed system is modular and configurable in various internal parameters, including the numeric format used, resulting in a system adaptable to user needs. Having in sight to obtain a system-on-a-chip, the internal memory of the FPGA is used to store the simulation data. The interface with the outside is a serial port, which is a known generic interface with many use applications already developed.

An average calculation rate of 5.3 GFLOPS is achieved, but this can be increased using a FPGA with a higher capacity that the one used in the development of this work, a XC4VSX35 Virtex-4. The system is also easily portable to newer FPGA families, allowing calculation rates even higher. For a XC7VX980T FPGA, from the Virtex-7 family, the estimated calculation rate is 49.5 GFLOPS.

Keywords

N-body simulation, gravitational kernel, pipelined calculus unit, FPGA

Índice

Índice de Figuras

Índice de Tabelas

Lista de Siglas

ASCII – *American Standard Code for Information Interchange*

ASIC – *Application Specific Integrated Circuit*

BRAM – *Block RAM*

CLB – *Configurable Logic Block*

DCM – *Digital Clock Manager*

DSP – *Digital Signal Processor*

FFT – *Fast Fourier Transform*

FIR – *Finite Impulse Response*

FPGA – *Field Programmable Gate Array*

FPU – *Floating-Point Unit*

GFLOPS – *Giga-Floating-point Operations Per Second*

GPU – *Graphics Processing Unit*

GRAPE – *Gravity Pipe*

IEEE – *Institute of Electrical and Electronics Engineers*

IIR – *Infinite Impulse Response*

IOB – *Input Output Block*

IP – *Intellectual Property*

IPCore – *Intellectual Property Core*

ISE – *Integrated software environment*

LED – *Light-Emitting Diode*

LUT – *Lookup Table*

NaN – *Not a Number*

NAIF – *Navigation and Ancillary Information Facility*

NASA – *National Aeronautics And Space Administration*

OPB – *On-chip Peripheral Bus*

PCI – *Peripheral Component Interconnect*

PISO – *Parallel Input Serial Output*

PROGRAPE – *Programmable Gravity Pipe*

P&R – *Place and Route*

RAM – *Random-Access Memory*

SIPO – *Serial Input Parallel Output*

TANOR – *Tool for Accelerating N-body Simulations on Reconfigurable Platform*

UART – *Universal Asynchronous Receiver/Transmitter*

VHDL – *VHSIC Hardware Description Language*

VHSIC – *Very High Speed Integrated Circuits*

Lista de Símbolos

Símbolos comuns

$\overrightarrow{e_x}$ - Versor da coordenada *x*

$\overrightarrow{e_y}$ - Versor da coordenada *y*

$\overrightarrow{e_z}$ - Versor da coordenada *z*

$floor()$ - Função de arredondamento para o número inteiro mais próximo inferior.

N - Número de corpos

N_{steps} - Número de passos da simulação

n - Índice do passo de simulação

$\mathcal{O}(x)$ - Termo de ordem *x*

x_x - Abcissa (coordenada *x*) do corpo de índice *x*

y_x - Ordenada (coordenada *y*) do corpo de índice *x*

z_x - Cota (coordenada *z*) do corpo de índice *x*

Δt - Intervalo temporal entre passos de simulação

Grandezas físicas

$\vec{a}$ - Vetor aceleração, segunda derivada em ordem ao tempo do vetor posição

$\overrightarrow{a_{g_{x,y}}}$ - Aceleração provocada pela força da gravidade aplicada pelo corpo de índice *y* no corpo de índice *x*

$\vec{b}$ - Vetor arrancada, terceira derivada em ordem ao tempo do vetor posição

$\vec{F}$ - Vetor força

$\overrightarrow{F_{g_x}}$ - Força gravítica total aplicada no corpo de índice *x*

$\overrightarrow{F_{g_{x,y}}}$ - Força gravítica aplicada no corpo de índice *x* pelo corpo de índice *y*

G - Constante de gravitação universal ($6.6742101 \times 10^{-11}\ m^3 kg^{-1} s^{-2}$)

m - Massa

m_x - Massa do corpo de índice *x*

$\vec{r}$ - Vector posição

t - Tempo

t_x - Instante temporal de índice *x*

$\vec{v}$ - Vetor velocidade, primeira derivada em ordem ao tempo do vetor posição

Integração numérica

a_{xy} - Coeficiente *a* de índices *x* e *y* do método Runge-Kutta

b_x - Coeficiente *b* de índice *x* do método Runge-Kutta

c_x - Coeficiente *c* de índice *x* do método Runge-Kutta

g_x - Valor da função genérica *g* no instante *x*

$g(t)$ - Função genérica dependente do tempo

$g'(t)$ - Primeira derivada em ordem ao tempo da função genérica $g(t)$

$g''(t)$ - Segunda derivada em ordem ao tempo da função genérica $g(t)$

k_x - Parcela de índice *x* do método Runge-Kutta

N_g - Número de pontos da grelha espacial

Representações numéricas

$b(x)$ - Bit de índice *x*

exp - Expoente da representação de vírgula flutuante

exp_{eff} - Expoente efetivo do número de vírgula flutuante

exp_l - Dimensão do expoente

$exp(x)$ - Bit de índice *x* do expoente

$frac_l$ - Dimensão da parte fracionária

$frac(x)$ - Bit de índice *x* da parte fracionária

int_l - Dimensão da parte inteira

$int(x)$ - Bit de índice *x* da parte inteira

$mant_l$ - Dimensão da mantissa

$mant(x)$ - Bit de índice *x* da mantissa

S - Bit sinal

Sistema

L_{dados} - Largura em bits dos dados da linha de deslocamento de dados

L_{linha} - Número de estágios da linha de deslocamento de dados

N_{cores} - Número de unidades de processamento

$N_{flip\ flops_{linha}}$ - Número de *flip flops* da linha de deslocamento de dados

$N_{LUTS_{linha}}$ - Número de *LUT* da linha de deslocamento de dados

N_{slices} - Número de *slices* disponíveis na *FPGA*

$Taxa_{calcúlo}$ - Número de operações de vírgula flutuante por segundo

1 Introdução

Na área da física existem vários fenómenos cuja evolução ao longo do tempo pode ser obtida pela soma das contribuições de cada uma das partes dos sistemas abordados. Estes sistemas são constituídos por vários elementos de características semelhantes, sendo a interação entre estes determinada por forças da Natureza, como a força gravítica ou a força elétrica. Neste conjunto de sistemas estão incluídos sistemas diversos, aglomerados de corpos celestes, como o nosso sistema solar ou qualquer galáxia do universo, ou ainda as moléculas que constituem o nosso corpo, por exemplo. No primeiro caso, a evolução temporal do estado dos corpos celestes é regida pela atração gravítica entre os mesmos, no segundo caso, a mesma evolução é regida por interações como a van de Waals, modelada pela equação de Lennard-Jones [1].

Dada a natureza de cada um dos fenómenos apresentados, e apesar das leis de interação ponto a ponto serem definidas por equações conhecidas, não é possível obter analiticamente expressões matemáticas que descrevam o comportamento destes sistemas quando o número de elementos é superior a dois. Desta forma, o comportamento destes sistemas é inferido por simulação computacional, recorrendo a métodos de integração numérica.

Na simulação de um sistema deste género é definido um intervalo temporal entre iterações, ao fim do qual são calculadas as forças de interação entre os corpos. Estas forças são depois utilizadas para obter as acelerações dos mesmos. As futuras posições dos corpos são calculadas tendo por base as posições atuais, velocidades ou ainda as acelerações, por meio de integração. Estes novos dados são utilizados como base de cálculo da iteração seguinte, sendo o resultado de cada iteração dependente do resultado da iteração anterior.

A simulação computacional permite teoricamente obter o resultado pretendido, mas surgem algumas questões quanto à implementação da mesma. Uma primeira questão levanta-se com a definição do intervalo temporal entre iterações. Embora um intervalo temporal reduzido permita diminuir o erro provocado pela imprecisão das representações numéricas utilizadas, também implica um maior tempo de simulação. Por outro lado, o elevado número de corpos de alguns destes sistemas [2] contribui também para um aumento do tempo total de simulação. Para fazer face ao elevado peso computacional deste tipo de simulação, são frequentemente utilizadas arquiteturas de multiprocessamento. Por fim, a representação numérica utilizada também contribui de forma importante para o resultado final, pois os erros derivados da precisão numérica podem acumular-se a cada passo da simulação.

Existem várias opções de suporte físico para este tipo de simulação. Os computadores genéricos são uma opção acessível, mas a sua arquitetura não está otimizada para o problema em causa. No outro extremo, estão soluções de *hardware* concebidas especificamente para realizar este tipo de simulação [3], constituídos por conjuntos de módulos de hardware baseados em *Application Specific Integrated Circuits* (*ASIC*s) que trabalham segundo uma lógica de multiprocessamento, possuindo um desempenho bastante superior à solução anterior, mas os custos económicos derivados do processo de desenvolvimento e produção de *ASIC* são elevados. Existe ainda um tipo de solução intermédia, que é o uso de *Field Programmable Gate Arrays* (*FPGA*s), que são dispositivos de *hardware* programáveis. Embora não apresentem o desempenho dos *ASIC*, as *FPGA* podem ser programadas para que as suas células de *hardware* digital constituam módulos de processamento dedicado, que podem estar também inseridos em esquemas de multiprocessamento. As próprias *FPGA* podem também ser integradas em aglomerados de processamento.

Este trabalho enquadra-se no problema apresentado como uma solução de *hardware* dedicado em *FPGA*. O sistema de processamento desenvolvido utiliza um algoritmo de cálculo da força gravítica num sistema de *N*-corpos e um algoritmo de integração numérica para se obter um simulador da evolução de sistemas de *N*-corpos. A utilização de uma *FPGA* permite o desenvolvimento do sistema num ambiente que possibilita uma depuração de erros mais fácil e sem custos de produção de *hardware* associados aos *ASIC*. O sistema utilizará apenas a memória interna da FPGA. A evolução da tecnologia de fabrico das *FPGA* tem resultado num aumento da capacidade de memória internas das mesmas, permitindo o armazenamento de maiores volumes de dados.

A utilização de *hardware* dedicado, quer por meio de *ASIC* ou por meio de *FPGA*, introduz ainda novas questões quanto à implementação de uma solução. Visto o número de células lógicas destes circuitos ser limitada, existe um compromisso entre a precisão de cálculo e a percentagem de área utilizada destes circuitos. Representações de vírgula flutuante com grande número de casas decimais originam unidades lógicas de grande dimensão, diminuindo o número de elementos de processamento paralelo concebíveis. Representações com reduzido número de casas decimais geram unidades lógicas de menor dimensão, mas os erros de cálculo assumem uma maior expressão no resultado final. Representações de vírgula fixa teriam uma melhor relação entre o número de bits da palavra numérica e a área utilizada, mas este tipo de representação tem uma menor gama dinâmica, não se adequando a cálculos em que os vários valores assumem ordens de grandeza bastante diferentes. Um estudo da relação área/precisão numérica será realizado para obter informação quanto ao número de unidades de processamento realizáveis em FPGA, fator importante na taxa de cálculo total e tempo de processamento das simulações.

1.1 Objetivos

O objetivo deste trabalho é o projeto e desenvolvimento de um sistema integrado de cálculo das acelerações resultantes das forças gravíticas e utilização destes valores num algoritmo de integração numérica, por forma a obter a evolução de um sistema de *N*-corpos.

A integração de ambos os passos de obtenção da evolução do estado do sistema de *N*-corpos numa *FPGA* tem por objetivo eliminar tempos de comunicação durante a simulação, como aconteceria se apenas uma parte do algoritmo fosse implementado na *FPGA* e o restante implementado num computador genérico.

A solução adotada será configurável em termos de número de unidades de processamento e formato numérico utilizado, adaptando-se assim às necessidades de precisão da aplicação final. A arquitetura criada será modular, permitindo a sua adaptação a outros cenários de simulação e algoritmos de cálculo de forças em sistemas de *N*-corpos.

1.2 Organização da tese

Este relatório de projeto final de mestrado segue a seguinte estrutura de capítulos:

- Capítulo 2 - Estado-da-arte – Capítulo em que são apresentados alguns trabalhos desenvolvidos na área temática deste projeto;
- Capítulo 3 - Enquadramento teórico e preparação – Neste capítulo são apresentados os temas base do desenvolvimento do projeto proposto, como algoritmos de cálculo da interação gravítica entre *N*-corpos, algoritmos de integração numérica, estrutura interna de *FPGA* e constituição de representações numéricas. É realizada uma implementação em *software* dos algoritmos e são definidas características do sistema em *hardware*;
- Capítulo 4 - Módulo de cálculo da aceleração em *hardware* – Neste capítulo é desenvolvido o módulo principal do sistema de simulação, tendo em conta as limitações de recursos existentes e a relação entre área utilizada e taxa de cálculo final;
- Capítulo 5 - Sistema de simulação de *N*-corpos – Neste capítulo são abordados os módulos do sistema desenvolvido e a arquitetura segundo a qual estes são utilizados;
- Capítulo 6 - Implementação e Resultados – Neste capítulo são apresentados resultados da implementação do sistema quanto a área ocupada e taxas de cálculo;
- Capítulo 7 – Conclusões – Capítulo final com considerações sobre o trabalho realizado.

2 Estado-da-arte

Existem várias arquiteturas possíveis para simulação de sistemas de N-corpos. No conjunto de arquiteturas utilizadas estão incluídas as seguintes:

- aglomerados computacionais com computadores genéricos ou super-computadores;
- sistemas baseados em processadores gráficos, *GPU* (*Graphics Processing Unit*);
- aglomerados de processadores gráficos;
- sistemas de hardware dedicado baseados em *ASIC* ou *FPGA* com comunicação com computador genérico;
- aglomerados computacionais de componentes de hardware dedicado com comunicação com computador genérico;
- sistemas mistos, em que é utilizada aceleração de hardware mas parte do cálculo é efetuado num computadores genérico;
- sistemas mistos com vários componentes de *hardware* dedicado.

Aglomerados de unidades de processamento, de computadores genéricos, *FPGA* ou ambos, têm diferentes características. Os aglomerados de computadores genéricos são comuns, mas a arquitetura de um processador não dedicado não permite explorar ao máximo as possibilidades de paralelização do problema, mesmo quando utilizados vários processadores [26].

Em sistemas mistos, é possível aprofundar os mecanismos de paralelismo por meio de estruturas de pipeline ou criação de várias unidades de processamento paralelas. Estas soluções incluem o sistema *PROGRAPE* (*Programmable Gravity Pipe*), que é uma versão programável do sistema *GRAPE* (*Gravity Pipe*), pois utiliza *FPGA* ao invés de *ASIC*. O sistema *GRAPE* consiste numa placa de processamento em que são incluídos vários módulos de *hardware* dedicado com estruturas de *pipeline*. A placa possui uma interface *PCI* (*Peripheral Component Interconnect*) que possibilita a sua inclusão num computador genérico. As tarefas de complexidade $\mathcal{O}(N^2)$ são executadas no sistema de *hardware* dedicado, sendo as restantes, de complexidade $\mathcal{O}(N)$, executadas no computador genérico. No entanto estes sistemas apresentam tempos de espera nas comunicações de dados entre o computador e o *hardware* dedicado.

Uma versão recente do sistema *GRAPE* é o *GRAPE-8* [3]. A placa de processamento deste sistema inclui dois *chips GRAPE-8* e uma *FPGA* Altera Arria GX. Os *chips GRAPE-8* contêm as unidades de cálculo em *pipeline* e a *FPGA* age como controlador e interface dos *chips* com o conector *PCI-Express*. Cada *chip* integra 48 unidades de *pipeline*, funcionando a uma frequência de 250 MHz e executando 40 operações de vírgula flutuante por ciclo de relógio, num total de 480 *GFLOPS* por *chip* e 960 *GFLOP* por placa, para um consumo de 46 W. É

utilizada uma representação de vírgula fixa com 32 bits para as posições, 64 bits para a acumulação e vírgula flutuante com 9 bits de mantissa para as restantes operações. As operações são realizadas segundo a estrutura presente na Figura 2.1.

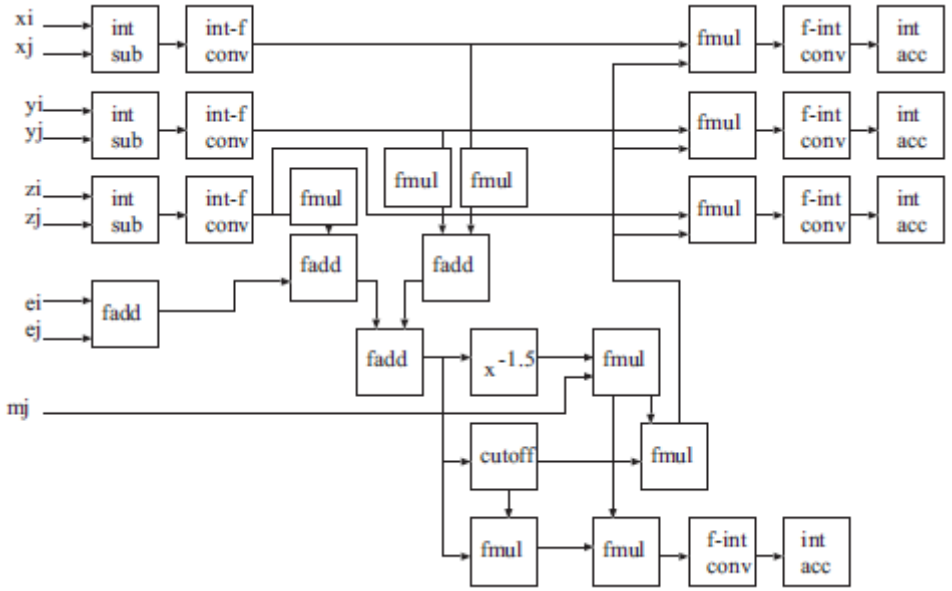


Figura 2.1 - Estrutura de cálculo com pipeline do GRAPE-8.

Sistemas baseados em aglomerados de *FPGA* são caracterizados por ligação paralelas entre as várias unidades do sistema, em que cada unidade possui memória local, ao invés do acesso a uma zona de memória global, existindo comunicação entre os vários módulos do sistema para a gestão da memória.

Um sistema computacional constituído unicamente por *FPGA Xilinx Virtex-II Pro XC2VP100*, baseado numa lógica de memória distribuída e ligações configuráveis entre nós organizados em níveis hierárquicos foi concebidos em outro projeto [4]. A programação do sistema é constituída por 4 fases, uma fase inicial em que é definido um algoritmo de simulação em linguagem C/C++, uma segunda fase em que este algoritmo é dividido em tarefas elementares que possam ser paralelizadas, uma terceira fase em que estas tarefas são implementadas em processadores embebidos nas *FPGA* e uma fase final em que parte ou a totalidade das tarefas são executadas por estruturas de *hardware* dedicado nas *FPGA*, ao invés de serem utilizados processadores embebidos. Uma primeira versão aplicação deste sistema foi realizada para realizar simulações de gases nobres, simulações para as quais só é necessário calcular as forças de van de Waals presentes no sistema. O sistema utiliza duas *FPGA*, uma que realiza todos os cálculos de forças, a acumulação das mesmas e atualiza as posições dos átomos, e outra que realiza a comunicação com o computador hospedeiro, tal como representado na Figura 2.2 [4].

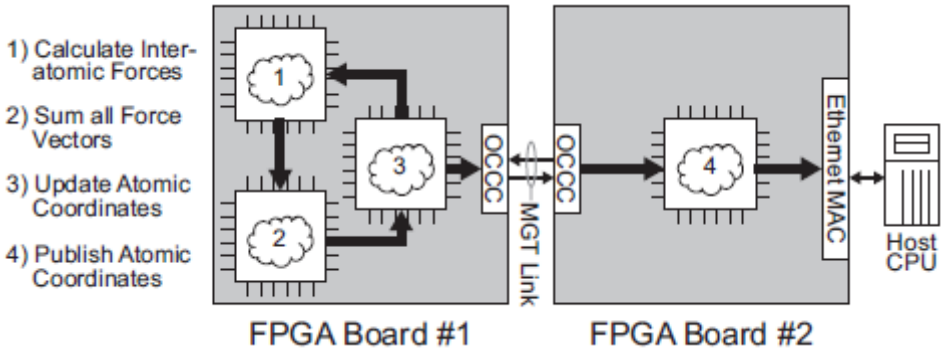


Figura 2.2 - Implementação da aplicação de simulação de gases nobres.

A ferramenta TANOR (*Tool for Accelerating N-body Simulations on Reconfigurable Platform*) [5] permite a descrição de um algoritmo em MATLAB, sendo este depois traduzido na configuração de uma *FPGA* por meio de um processo automático. Esta ferramenta tem foco no problema da simulação de sistemas com *N*-corpos, podendo restrições quanto à área, precisão, latência e consumo ser alteradas numa interface gráfica.

O algoritmo realizado em MATLAB é analisado para serem removidas redundâncias e são extraídas duas partes, uma a realizar em *FPGA* e outra para ser executada no computador que serve de hospedeiro. Para a programação da *FPGA* são analisadas três principais vertentes: a geração de núcleos de cálculo paralelo com *pipeline*, o controlo de fluxo de dados e a interface com o hospedeiro, realizada por *PCI-Express*. No hospedeiro a comunicação de dados é controlada por meio de interrupções. Por exemplo, quando a simulação é concluída é gerada uma interrupção pelo *hardware* de simulação.

O sistema gerado por esta ferramenta suporta representações de vírgula flutuante personalizadas. Operações tais como a soma e subtração, multiplicação, divisão e raiz quadrada são suportadas com a utilização de *IP Cores* da Xilinx (*LogiCore IP Floating Point*), bem como funções cuja implementação seja realizada com tabelas.

O sistema exemplo foi implementado em duas *FPGA* Virtex2Pro-100 da Xilinx. Para a simulação de interações gravíticas foi atingido um desempenho de 22.2 *GFLOPS* para uma representação numérica com 8 bits de expoente e 16 de mantissa, 56 operações por unidade de cálculo e 3 unidades de cálculo por *FPGA*. Para a precisão numérica utilizada foi obtida uma precisão da ordem de $1x10^{-3}$. O número de *slices* gastos por unidade de cálculo ultrapassa os 15000.

Segue-se a apresentação de sistemas de simulação em *FPGA* criadas segundo processo típico de projeto. Um simulador em *FPGA* da evolução de um sistema de *N*-corpos foi realizado numa *FPGA* Virtex-II XC-2V2000 [1]. Este sistema tem por objetivo o processamento de interações num sistema de dinâmica de moléculas, calculando o potencial de Lennard-Jones e uma soma modificada das forças elétricas. O sistema inclui uma interface de porta série, um processador

embebido *Microblaze* e um módulo de processamento dedicado, todos ligados entre si por meio de um barramento *OPB* (*On-chip Peripheral Bus*), segundo a arquitetura da Figura 2.3 [1]. O processador tem a função de controlar a porta série e comunicação com o hospedeiro, para além de transmitir os dados para a estrutura de cálculo dedicado. Foi utilizada a representação de vírgula fixa, com 7 bits para a parte inteira e 25 para a parte fracionária das coordenadas e 2 bits para a parte inteira e 29 para a parte fracionária das cargas das moléculas. As forças são calculadas indexando a uma tabela e utilizando interpolação para obter os valores não disponíveis na mesma.

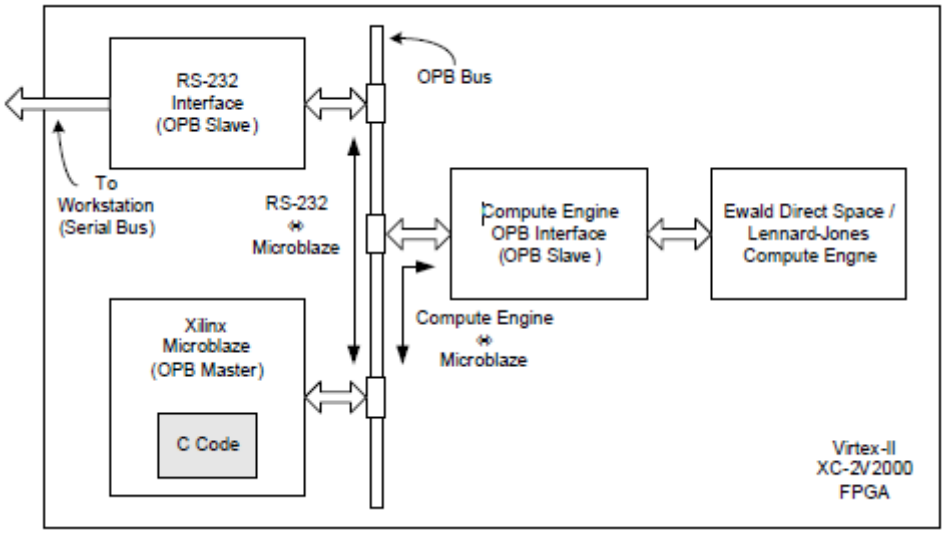


Figura 2.3 - Arquitetura do processador com processador Microblaze e módulo de cálculo dedicado.

Para o módulo de cálculo da soma das forças elétricas é obtida uma frequência de 82.2 MHz e são obtidos 80 MHz para o módulo de cálculo da interação de Lennard-Jones. O módulo de cálculo das interações de Lennard-Jones utiliza 24 blocos de BRAM de 18 kbits, 52 multiplicadores e 9258 *slices*. Por sua vez o módulo das forças elétricas utiliza 24 blocos de BRAM, 55 multiplicadores e 8648 *slices.*

3 Enquadramento teórico e preparação

Neste capítulo são apresentadas algumas temáticas que fazem parte do assunto em estudo do trabalho, bem como uma realização prática dos algoritmos em *software*, que serve de preparação para a implementação em *hardware*. As características base desta implementação são também referidas.

Na secção 3.1 são apresentados alguns métodos de cálculo das forças gravíticas num sistema de *N*-corpos. Em seguida, em 3.2, são apresentados vários métodos de integração numérica. Em 3.3 é apresentada a estrutura interna de uma *FPGA* e quais os tipos de recursos disponíveis na mesma. Finalmente, em 3.4 são apresentadas as representações numéricas de vírgula fixa e vírgula flutuante, tendo em foco a representação de vírgula flutuante, que é a utilizada neste trabalho.

Os primeiros dois temas abordados, cálculo de forças gravíticas e método de integração numérica, permitem definir como serão obtidas as novas posições dos corpos em cada iteração da simulação. Dado um intervalo temporal Δt entre passos consecutivos da simulação, o algoritmo genérico de simulação é o seguinte:

- cálculo das forças gravíticas entre corpos e obtenção da aceleração de cada corpo, $\vec{a}(t)$, a partir das forças calculadas;
- cálculo da velocidade seguinte de cada corpo, $\vec{v}(t + \Delta t)$, a partir da velocidade anterior, $\vec{v}(t)$, e da aceleração;
- cálculo da posição seguinte de cada corpo, $\vec{r}(t + \Delta t)$, a partir da posição anterior, $\vec{r}(t)$, e dos dados de velocidade e aceleração.

Este algoritmo é repetido tantas vezes quanto necessário para realizar o número de passos de simulação pretendidos.

A sequência dos passos e forma dos mesmos depende do método de integração numérica utilizado. Por exemplo, a posição seguinte pode ser calculada utilizando diretamente ou não a aceleração. Também o número de cálculos intermédios pode variar, podendo, por exemplo, serem calculadas várias velocidades intermédias no passo de simulação. Diferentes métodos de integração numérica necessitam de passos de iniciação dos valores utilizados. Outros ainda utilizam dados referentes a vários passos de simulação anteriores. Existem também várias formas de cálculo das interações gravíticas entre corpos, a partir das quais são obtidas as acelerações dos mesmos. O método de simulação implementado depende do algoritmo de cálculo de força gravítica e método de integração numérica escolhidos.

As secções que abordam a estrutura interna de uma *FPGA* e tipos de representação numérica têm como função uma melhor compreensão dos recursos disponíveis e como estes são utilizados pelo sistema de simulação.

Na secção 3.5 são apresentados resultados práticos e a estrutura de implementação do algoritmo de simulação, sendo estas realizadas em MATLAB e C. Para estas realizações foram escolhidos para o cálculo das forças gravíticas o método direto e o método de Barnes-Hut, apresentados nas secções 3.1.1 e 3.1.2, e o método *Verlet* de velocidade para a integração numérica, conforme a secção 3.2.2. Estas implementações servem como uma primeira abordagem aos algoritmos antes da implementação em *hardware*.

3.1 Algoritmos de cálculo da interação gravítica entre *N*-corpos

Existem vários algoritmos de cálculo das interações gravíticas num sistema de *N*-corpos. O método direto é o método base de cálculo com a aplicação direta da equação da força gravítica a todos os pares de corpos, sendo os outros métodos tentativas de reduzir o peso computacional do método base. Métodos baseados em árvore, tal como o proposto por Barnes e Hut, ou métodos baseados em redes de partículas reduzem o número de cálculos realizados mas o erro final obtido é maior. Existem ainda métodos mistos, que procuram ser soluções intermédias em termos de erro e processamento. Um exemplo é a aplicação do método direto para distâncias curtas entre corpos e do método de rede de partículas quando as distâncias são maiores.

3.1.1 Algoritmo de cálculo direto

Este método baseia-se na obtenção do valor de todas as interações gravíticas entre os corpos. Embora este método permita um cálculo mais preciso do valor da força aplicada a cada corpo em relação a outros métodos, a sua complexidade é $\mathcal{O}(N^2)$, tornando-se um método com um elevado peso computacional para um grande número de corpos.

Para um dado corpo de índice i, a força gravítica total aplicada ao corpo é dada pelo somatório das interações ponto-a-ponto entre os corpos:

$$\overrightarrow{F_{g_i}} = G\sum_{i\neq j} m_j m_i \frac{\left(x_j - x_i\right)\overrightarrow{e_x} + \left(y_j - y_i\right)\overrightarrow{e_y} + \left(z_j - z_i\right)\overrightarrow{e_z}}{\left(\left(x_j - x_i\right)^2 + \left(y_j - y_i\right)^2 + \left(z_j - z_i\right)^2\right)^{\frac{3}{2}}}, \tag{3.1}$$

em que a força aplicada por um dado corpo 2 em outro corpo 1 é dada por

$$\overrightarrow{F_{g_{1,2}}} = Gm_2m_1\frac{(x_2 - x_1)\overrightarrow{e_x} + (y_2 - y_1)\overrightarrow{e_y} + (z_2 - z_1)\overrightarrow{e_z}}{((x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2)^{\frac{3}{2}}}. \tag{3.2}$$

Na expressão (3.2) pode ser inserido um fator de suavização ϵ, que permite obter menores valores de força quando os corpos se aproximam, resultando em

$$\overrightarrow{F_{g_{1,2}}} = Gm_2m_1\frac{(x_2-x_1)\overrightarrow{e_x}+(y_2-y_1)\overrightarrow{e_y}+(z_2-z_1)\overrightarrow{e_z}}{((x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2+\epsilon^2)^{\frac{3}{2}}}. \quad (3.3)$$

Para o cálculo das posições são necessários os valores das acelerações dos corpos que podem ser obtidas a partir da conhecida relação entre a força e a aceleração

$$\vec{F} = m\vec{a}, \quad (3.4)$$

assim, o valor da aceleração provocada pelo corpo 2 no corpo 1 é obtido a partir de (3.2) e (3.4):

$$\overrightarrow{a_{g_{1,2}}} = \frac{\overrightarrow{F_{g_{1,2}}}}{m_1} = Gm_2\frac{(x_2-x_1)\overrightarrow{e_x}+(y_2-y_1)\overrightarrow{e_y}+(z_2-z_1)\overrightarrow{e_z}}{((x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2)^{\frac{3}{2}}}. \quad (3.5)$$

De igual forma pode ser calculado o valor da aceleração total num corpo de índice *i*:

$$\overrightarrow{a_{g_i}} = G\sum_{i\neq j} m_j\frac{(x_j-x_i)\overrightarrow{e_x}+(y_j-y_i)\overrightarrow{e_y}+(z_j-z_i)\overrightarrow{e_z}}{\left((x_j-x_i)^2+(y_j-y_i)^2+(z_j-z_i)^2\right)^{\frac{3}{2}}}, \quad (3.6)$$

O valor obtido para a aceleração pode então ser utilizado, juntamente com outros dados como a posição e a velocidade atuais, para obter a nova posição através de integração numérica, que descreveremos mais à frente.

A expressão (3.5) pode ainda ser transformada na expressão seguinte:

$$\overrightarrow{a_{g_{1,2}}} = Gm_2\frac{(x_2-x_1)\overrightarrow{e_x}+(y_2-y_1)\overrightarrow{e_y}+(z_2-z_1)\overrightarrow{e_z}}{((x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2)} \times\frac{1}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}}. \quad (3.7)$$

Este foi o método de cálculo de forças gravíticas implementado em *hardware*. A sua utilização foi vantajosa no desenvolvimento deste trabalho, sendo abordada nos capítulos 3.5 e 4.

3.1.2 Algoritmo com árvore de Barnes e Hut

Sendo um pequeno erro no cálculo da aceleração tolerável para sistemas de corpos sem colisões [2], pode ser utilizada uma estrutura em árvore para otimizar o número de cálculos necessários para a obtenção do valor da força. Ao calcular-se a força aplicada em um corpo, em vez de se obter o valor da interação entre esse corpo e todos os restantes, é obtida a interação entre o corpo e nós da árvore, em que cada nó representa um ou mais corpos de uma região do espaço.

Um tipo de árvore que pode ser criado é a árvore octal [6] [7]. É definida uma região limitada do espaço que contém todos os corpos. Este espaço tridimensional é dividido em 8 octantes e cada uma destas regiões é também dividida desta forma enquanto nela estiver contido mais do que um corpo. A cada região corresponde um corpo representativo de todos os corpos inseridos nesta região, e todos os corpos estão inseridos numa estrutura em árvore.

Para calcular a força aplicada em um corpo, a árvore é percorrida a partir do topo e analisado o rácio entre o volume representado por cada nó e a distância ao corpo cuja força aplicada está a ser calculada. Se o valor deste rácio for inferior a um limiar pré-definido, é realizado o cálculo da força entre o corpo e o nó. Caso contrário desce-se um nível na árvore e o procedimento é repetido para os nós de nível inferior, até chegar às folhas, que representam apenas um corpo. A força total será o somatório das forças calculadas.

A complexidade deste algoritmo é $\mathcal{O}(Nlog(N))$, sendo que o número total de cálculos é definido pelo valor do limiar utilizado para determinar quando se deve descer de nível na árvore. Se for tomado um valor de limiar nulo este método degenera no método direto, mas com o peso computacional adicional de construção da árvore. Este método permite obter uma importante otimização para elevados números de corpos e foi primeiro definido por Barnes e Hut [8].

Embora este método apresente vantagens do ponto de vista de número de operações de vírgula flutuante em relação ao método direto, a sua implementação numa estrutura de multiprocessamento implica a paralelização dos algoritmos de manipulação da estrutura em árvore, razão pela qual não foi implementado em *hardware*. No entanto foi realizado um estudo comparativo dos dois métodos no capítulo 3.5.

3.1.3 Algoritmo de rede de partículas

Na aplicação deste método [9] é construída uma grelha tridimensional. Os corpos são percorridos um a um e, no caso do cálculo de forças gravíticas, as suas massas são adicionadas uma a uma ao ponto da grelha mais próximo do corpo.

Obtida a distribuição de massa no espaço é aplicada uma *Fast Fourier Transform* (FFT) à mesma para obter a solução da equação de Poisson para o potencial gravítico [7]. Obtido o valor da força gravítica na grelha este é interpolado para os corpos.

Se o número de pontos da grelha for menor que o número de corpos, este algoritmo é de complexidade $\mathcal{O}\left(N_g log\left(N_g\right)\right)$, em que N_g corresponde ao número de pontos da grelha. Este método apresenta uma precisão reduzida em zonas com elevada concentração de corpos. Para resolver este problema podem ser utilizados métodos mistos, que aplicam o método direto para as interações de curta distância, ou métodos com rede variável no espaço.

3.2 Métodos de integração numérica

Neste capítulo são apresentados alguns métodos numéricos para a resolução de equações diferenciais ordinárias. Os métodos apresentados diferem na sua ordem, sendo esta determinada pela ordem do erro global do integrador. A ordem do erro global é obtida a partir da ordem da potência de Δt associada à parcela do erro global.

3.2.1 Método de Euler

O método de *Euler* [10] é um método de integração numérica de primeira ordem.

Dado o problema de condição inicial

$$g'(t) = f\big(t, g(t)\big), g(t_0) = g_0, \tag{3.8}$$

e considerando um intervalo temporal igual a Δt tal que $t_n = t_0 + n\Delta t$, o método de Euler resulta em

$$g(t + \Delta t) = g(t) + g'(t)\Delta t. \tag{3.9}$$

Se se particularizar para o cálculo da posição em função da velocidade e aceleração obtém-se:

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \vec{a}(t)\Delta t, \tag{3.10}$$

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t. \tag{3.11}$$

Se a ordem da expansão de Taylor de $y(t)$ for aumentada obtém-se

$$g(t + \Delta t) = g(t) + g'(t)\Delta t + \frac{g''(t)\Delta t^2}{2} + \mathcal{O}(\Delta t^3), \tag{3.12}$$

de onde se pode concluir que o erro de truncatura local, correspondente à diferença entre a expansão de Taylor de maior ordem e a utilizada no método de integração, é de segunda ordem.

O limite do erro global, erro resultante da acumulação dos erros de truncatura locais, pode ser obtido analiticamente. Este erro assume um valor de primeira ordem.

3.2.2 Método de Verlet

O algoritmo de Verlet [11] é um método de integração numérica de segunda ordem comum em simulações de dinâmica de moléculas. São utilizadas duas séries de Taylor de terceira ordem para obter as posições no instante posterior e anterior da simulação:

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \frac{\vec{a}(t)\Delta t^2}{2} + \frac{\vec{b}(t)\Delta t^3}{6} + \mathcal{O}(\Delta t^4), \quad (3.13)$$

$$\vec{r}(t-\Delta t) = \vec{r}(t) - \vec{v}(t)\Delta t + \frac{\vec{a}(t)\Delta t^2}{2} - \frac{\vec{b}(t)\Delta t^3}{6} + \mathcal{O}(\Delta t^4), \quad (3.14)$$

sendo $\vec{b}(t)$ a terceira derivada da posição em ordem ao tempo e $O(\Delta t^4)$ o termo de erro local de truncatura de quarta ordem da posição.

Das expressões (3.13) e (3.14) obtém-se

$$\vec{r}(t+\Delta t) = 2\vec{r}(t) - \vec{r}(t-\Delta t) + \vec{a}(t)\Delta t^2 + \mathcal{O}(\Delta t^4), \quad (3.15)$$

Sendo que esta expressão nos permite obter a próxima posição com base apenas nas posições atual e anterior e aceleração atual, pois os termos da primeira e terceira derivada da posição em ordem ao tempo são eliminados.

No entanto, a aplicação deste método introduz algumas questões. Uma delas é o facto de ser necessária não só a posição atual como a posição anterior para se determinar a posição seguinte. Outra é o facto de a velocidade não ser gerada diretamente pelo método de integração. Expressões auxiliares podem ser usadas para resolver estas duas questões, mas introduzindo um maior erro no cálculo.

Uma variante do método de *Verlet*, o *Verlet* de velocidade, permite obter a posição seguinte sem conhecimento da posição anterior, para além de a velocidade ser calculada no corpo do mesmo.

Este algoritmo baseia-se na seguinte sequência de cálculos para obter a posição após o intervalo de tempo definido $\vec{x}(t+\Delta t)$:

- cálculo da velocidade a meio intervalo:

  $$\vec{v}\left(t+\frac{1}{2}\Delta t\right) = \vec{v}(t) + \frac{1}{2}\vec{a}(t)\Delta t; \quad (3.16)$$

- cálculo da posição seguinte:

  $$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}\left(t+\frac{1}{2}\Delta t\right)\Delta t; \quad (3.17)$$

- cálculo da aceleração no intervalo de tempo seguinte $\vec{a}(t+\Delta t)$ utilizando a posição seguinte;
- cálculo da velocidade seguinte:

$$\vec{v}(t+\Delta t) = \vec{v}\left(t+\frac{1}{2}\Delta t\right) + \frac{1}{2}\vec{a}(t+\Delta t)\Delta t; \qquad (3.18)$$

Se a aceleração não depender diretamente da velocidade, como é o caso, o cálculo pode ser simplificado pela eliminação da velocidade a meio intervalo. A sequência de cálculos passa a ser a seguinte:

- cálculo da posição seguinte:

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \frac{1}{2}\vec{a}(t)\Delta t^2; \qquad (3.19)$$

- cálculo da aceleração no intervalo de tempo seguinte $\vec{a}(t+\Delta t)$ utilizando a posição seguinte;
- cálculo da velocidade seguinte:

$$\vec{v}(t+\Delta t) = \vec{v}(t) + \frac{1}{2}\big(\vec{a}(t) + \vec{a}(t+\Delta t)\big)\Delta t; \qquad (3.20)$$

O erro global de truncatura deste integrador é de segunda ordem, determinando a ordem deste integrador. Este foi o método de integração escolhido para o desenvolvimento deste trabalho, pois apresenta uma melhoria em termos de erro quando comparado com o método de Euler, mas mantendo uma complexidade reduzida.

3.2.3 Métodos Runge-Kutta

Esta família de métodos de integração numérica [12] [13] [14] tem por base o cálculo da primeira derivada da variável a integrar em vários momentos intermédios do passo de integração, ao invés da adição de derivadas de maior ordem no cálculo.

Existem vários métodos de Runge-Kutta. Por exemplo, os métodos explícitos de Runge-Kutta são definidos por:

$$g(t+\Delta t) = g(t) + \Delta t \sum_{i=1}^{s} b_i k_i, \qquad (3.21)$$

onde

$$k_i = f\left(t + c_i\Delta t, g(t) + \Delta t \sum_{j=1}^{i-1} a_{ij} k_j\right) \qquad (3.22)$$

e

$$c_1 = 0, \qquad c_i = \sum_{j=1}^{i-1} a_{ij}\,, \qquad i = 2,3,\ldots,s. \qquad (3.23)$$

Os coeficientes de um método Runge-Kutta explícitos podem ser apresentados segundo uma tabela de Butcher:

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & & \ddots & & \\ c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \dots & b_{s-1} & b_s \end{array} . \qquad (3.24)$$

O método clássico e mais conhecido desta família é o método de quarta ordem com tabela de Butcher dada por

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} , \qquad (3.25)$$

ou seja:

$$g(t+\Delta t) = g(t) + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4), \qquad (3.26)$$

$$k_1 = f\big(t, g(t)\big), \qquad (3.27)$$

$$k_2 = f\left(t + \frac{\Delta t}{2}, g(t) + \frac{\Delta t}{2}k_1\right), \qquad (3.28)$$

$$k_3 = f\left(t + \frac{\Delta t}{2}, g(t) + \frac{\Delta t}{2}k_2\right), \qquad (3.29)$$

$$k_4 = f(t + \Delta t, g(t) + \Delta t k_3). \qquad (3.30)$$

Particularizando para o problema de simulação em estudo, para o cálculo de cada uma das parcelas k_i é necessário calcular um valor de velocidade e para cada um destes valores de velocidade um valor de aceleração, pois a posição dos corpos varia para cada uma das parcelas. Isto implica um número de cálculos bastante superior ao necessário para os métodos anteriores, pelo que este método não foi utilizado. No entanto, este método teria a vantagem de possuir um erro local de truncatura de quinta ordem e um erro global de truncatura de quarta ordem.

3.3 Estrutura interna de uma *FPGA*

Este capítulo apresenta a estrutura interna de uma *FPGA*. Para a análise desta estrutura toma-se como exemplo a tecnologia do fabricante Xilinx [15], sendo os nomes utilizados referentes a componentes das *FPGA* deste mesmo fabricante. Outros fabricantes, como a Altera [16], usam uma estrutura interna semelhante com os blocos de função idêntica aos da Xilinx, mas estes blocos tomam outros nomes.

3.3.1 Blocos internos de uma *FPGA*

Os blocos internos de uma *FPGA* apresentam-se distribuídos numa estrutura modular. Uma *FPGA* tem como principais componentes internos:

- Blocos lógicos configuráveis, *Configurable Logic Blocks* (*CLB*s) – estes blocos são constituídos por *slices*, sendo que estes contêm a lógica combinatória e registos, ou *flip flops*, necessários para a construção dos circuitos lógicos pretendidos.
- Blocos de entrada e saída, *Input Output Block* (*IOB*s) – Estes blocos realizam a ligação com o exterior da *FPGA*. As entradas/saídas da *FPGA* encontram-se divididas por bancos, sendo que cada banco pode suportar um *standard* de níveis lógicos diferente.
- Linhas de interligação – estas linham ligam os diversos *CLB* entre si e aos *IOB*. Estas linhas estão otimizadas para introduzirem atrasos de propagação baixos nos sinais aplicados às mesmas, ou, em alguns casos, otimizadas de forma a comportarem-se como linhas de distribuição com reduzidas diferenças de atraso entre as várias saídas das mesmas, tal como no caso das linhas de distribuição de relógio;

As *FPGA* contêm ainda outros blocos internos que permitem melhorar a sua funcionalidade, nomeadamente:

- Blocos de *RAM* interna, *Block RAM* (*BRAM*) – Uma *FPGA* contém blocos de *hardware* dedicado para a implementação de memória de acesso aleatório, *Random-Access Memory* (*RAM*). Nas *FPGA* recentes estes blocos têm capacidades de 9, 18 ou 36 kbits, e são configuráveis quanto à dimensão e ao número das palavras armazenadas;
- *Digital Clock Managers* (*DCM*s) – Estes blocos são utilizados para realizar divisão ou multiplicação da frequência de um sinal de relógio, e alteração da fase do mesmo;
- Blocos de *hardware* dedicados a operações aritméticas – Também estão presentes na estrutura interna de uma *FPGA* blocos dedicados à realização de operações específicas, como multiplicadores ou *slices* de processamento de sinal.

3.3.2 *XtremeDSP slices*

Os blocos de *hardware* dedicados permitem realizar as operações associadas usando uma menor área que utilizando blocos lógicos genéricos, para além de ser possível obter maiores frequências de funcionamento. Na família Virtex-4 são incluídos *slices* dedicados ao processamento, os *XtremeDSP slices*, modelo *DSP48* [17].

Estes blocos incluem um multiplicador 18x18 e um somador/subtrator de 48 bits. O somador/subtrator pode ser usado como acumulador. Estão ainda disponíveis níveis de *pipeline* opcionais, que permitem aumentar a frequência máxima de funcionamento, bem como ligações internas entre *XtremeDSP slices* que permitem o transporte de dados entre estes e associações de forma a processar operandos de maior dimensão. A arquitetura destes blocos é a presente na Figura 3.1 [17]. Este bloco dispensa a utilização de células lógicas para a realização do acumulador, visto este estar incluindo no próprio bloco.

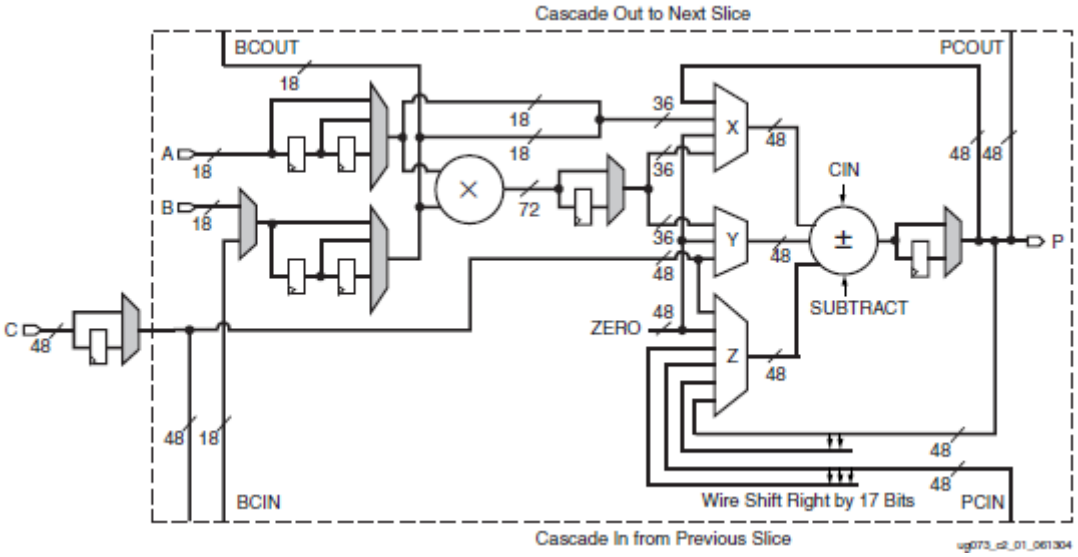


Figura 3.1 - Arquitetura do slice DSP48.

Os multiplicadores estão associados com blocos de memória interna, denominados nesta família de blocos de *SelectRAM*. Associando estas duas entidades a um acumulador realizado com blocos lógicos é possível realizar circuitos de processadores de sinal digital, ou *DSP*s (*Digital Signal Processors*). Estas células podem ser utilizadas, por exemplo, no projeto de filtros digitais *FIR* (*Finite Impulse Response*) e *IIR* (*Infinite Impulse Response*), de módulos aritméticos, etc.

Em outras famílias de *FPGA*, existem diferenças nos blocos internos dos *XtremeDSP slices* tais como pré-somadores, unidades com multiplicadores de 25x18 bits ou ainda capazes de realizar operações lógicas e alterações na estrutura de *pipeline*. Em *FPGA* mais antigas estes blocos podem não existir, estando, por exemplo, apenas disponíveis os multiplicadores.

3.3.3 Diferenças entre *CLB* e *slices* para diversas famílias de *FPGA*

As várias famílias de *FPGA* da Xilinx apresentam algumas diferenças na construção dos seus blocos de lógica configurável, sendo que estas influenciam a quantidade de blocos necessária para a realização de um dado circuito lógico.

Na família Virtex-4, por exemplo, cada *CLB* é constituído por 4 *slices*, como mostrado na Figura 3.2 [18].

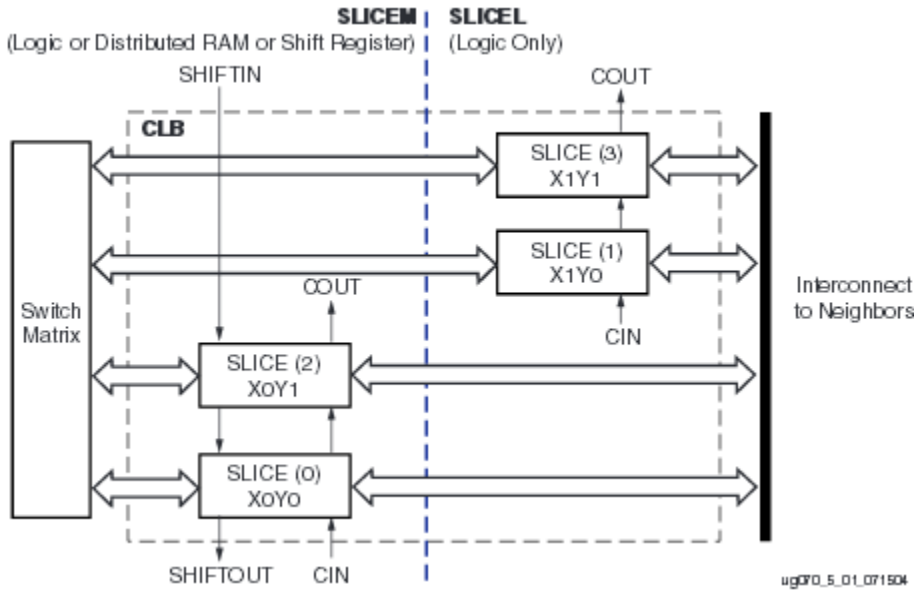


Figura 3.2 - Constituição de um CLB de uma FPGA Virtex-4.

Estes *slices* estão agrupados segundo dois tipos, *SLICEM* e *SLICEL*. Cada *slice* possui dois blocos de função lógica ou *LUT*s (*Look-Up Tables*), que são usados para implementar qualquer função lógica de uma saída e até 4 entradas e memórias ROM de 16x1 bit, dois elementos de armazenamento que podem ser utilizados como *flip flops* ou *latches*, multiplexadores, lógica de propagação de *carry* e portas aritméticas. A lógica de multiplexagem permite combinar os recursos das várias *LUT*. As *LUT* dos *slices* do tipo *SLICEM* podem ser ainda usadas como *RAM*s de 16x1 bit ou registos de deslocamento de 16 ou menos bits.

Em outra família de *FPGA*, a família Virtex-II, os *CLB* têm uma constituição semelhantes aos das Virtex-4, mas todas as *LUT* podem ser utilizadas como *RAM*s de 16x1 bits e registos de deslocamento de 16 bits [19], comportando-se como as *LUT* dos *slices* do tipo *SLICEM* das Virtex-4.

A partir da família Virtex-5 a organização dos *CLB* foi alterada, passando cada *CLB* a conter apenas dois *slices*, como mostrado na Figura 3.3 [20]. No entanto, a constituição dos *slices* foi também alterada, passando estes a conter 4 *LUT* e 4 *flip flops*. Logo, o número de elementos de função lógica e de armazenamento de um *CLB* manteve-se.

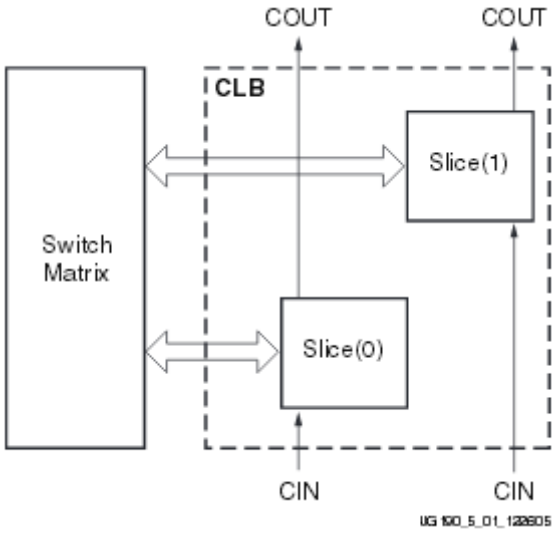


Figura 3.3 - Constituição de um CLB de uma FPGA Virtex-5.

Outra alteração foi realizada a nível das unidades lógicas. As *LUT* passaram de *LUT* com 4 entradas e uma saída, *LUT*4, para *LUT* de 6 entradas e 2 saídas, *LUT*6, que podem ser usadas como blocos de função de 6 entrada e uma saída ou dois blocos de função com 5 entradas e 2 saídas, desde que existam entradas partilhadas suficientes. Caso estas *LUT* pertençam a um SLICEM, podem ainda ser utilizadas como memórias de 64x1 bit e registos de deslocamento de 32 bits.

Na Tabela 3.1 pode-se observar alguns dados sobre a estrutura interna dos *CLB* de algumas famílias de *FPGA* [21] [22] [23]. Os dados da tabela não se aplicam para os modelos XC4VFX12 e XC4VFX20 da família Virtex-4, cuja percentagem de SLICEM não é exatamente 50%, embora o erro seja da ordem de 0,2-0,3%, nem para o modelo XC6SLX4 da Spantan-6, pois este modelo não possui *slices* do tipo SLICEL. Na família Spartan 6 existe outro tipo de *slice*, SLICEX, que não possuem linhas de propagação de *carry* e cujos recursos de multiplexagem são menores.

	Tipo de LUT	Número de *slices* por CLB	Número de LUTs por *slice*	Número de *flip flops* por *slice*	Percentagem de SLICEM
Virtex-4	LUT4	4	2	2	50%
Virtex-5	LUT6	2	4	4	25-45%
Spartan-6	LUT6	2	4	8	23-25%
Virtex-6	LUT6	2	4	8	28-41%
Artix-7	LUT6	2	4	8	29-34%
Kintex-7	LUT6	2	4	8	31-37%
Virtex-7	LUT6	2	4	8	28-43%

Tabela 3.1 - Diferenças entre estrutura dos CLB para diferentes famílias de FPGA.

3.4 Representações numéricas

Neste capítulo são apresentadas as representações de vírgula fixa e vírgula flutuante, duas opções para a implementação de números com parte fracionária, como os utilizados neste trabalho.

3.4.1 Vírgula fixa

A representação de vírgula fixa é bastante semelhante à representação usual de inteiros, mas uma vírgula virtual é definida e os bits posicionados à direita desta são utilizados para definir a parte fracionária do número [24]. Pode ser incluído ou não um bit sinal. Na Figura 3.4 observa-se um exemplo desta representação.

Figura 3.4 – Exemplo de formato de número de vírgula fixa com sinal.

É também definido um fator de escala associado ao número, que serve de fator multiplicativo, de forma a obter números de valor absoluto maior ou menor. Este fator não é incluído na representação binária do número, tendo o utilizador de realizar o controlo deste fator. Também é possível eliminar a parte inteira, trabalhando com número entre 0 e 1, que serão multiplicados pelo fator. Podem ser utilizadas várias dimensões distintas para a parte inteira e fracionária ao mesmo tempo, desde que sejam respeitadas algumas regras na realização das operações aritméticas, em particular o alinhamento do ponto na operação soma e subtração.

O valor de um número de vírgula fixa com sinal é dado por:

$$(-1)^S \times \frac{1}{2^{frac_l}} \times \left[\sum_{n=1}^{int_l+frac_l} b(n) \times 2^{n-1} \right] \times fator. \qquad (3.31)$$

As operações de vírgula fixa são realizadas da mesma forma que as correspondentes de números inteiros, no entanto, conforme a operação, é necessário compensar os fatores de multiplicação ou colocar os números no mesmo formato. Na soma, por exemplo, é necessário transformar os números para que tenham as mesmas dimensões da parte inteira e da parte fracionária, para além de terem de ser usados os mesmos fatores multiplicativos. Para alinhar os formatos utilizam-se deslocamentos de bits. O resultado terá o mesmo formato dos operandos. Na multiplicação, os operandos podem ter formatos e fatores diferentes, não sendo necessário alinhar as partes inteiras e decimais dos operandos, obtendo-se à saída um número com número de bits da parte inteira correspondente à soma do número de bits das partes inteiras dos operandos, e número de bits da parte fracionária correspondente também à soma dos números de bits correspondentes dos operandos. O fator multiplicativo final é o produto dos fatores multiplicativos.

3.4.2 Vírgula flutuante

Na representação de vírgula flutuante um número é composto por um bit sinal, *S*, um conjunto de bits que determinam as casas decimais do mesmo, a mantissa, e um conjunto de bits que age como expoente de um fator multiplicativo, Figura 3.5. O formato de vírgula flutuante é definido pela norma *IEEE 754* [25] do *IEEE* (*Institute of Electrical and Electronics Engineers*).

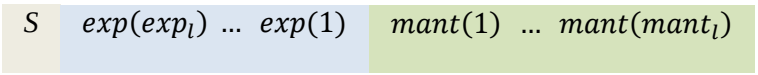


Figura 3.5 - Formato de número de vírgula flutuante.

A mantissa tem o formato

$$1, bits\ da\ mantissa, \tag{3.32}$$

ou seja, o '1' está escondido, não entrando na representação binária do número mas fazendo parte do valor do mesmo.

Dado este formato, o valor de um número normalizado de vírgula flutuante de base binária é dado por [26]:

$$(-1)^S \times \left[1 + \sum_{n=1}^{mant_l} mant(n) \times 2^{-n}\right] \times 2^{exp_{eff}}. \tag{3.33}$$

O fator multiplicativo permite obter valores bastante altos e valores bastante baixos, resultando numa grande gama dinâmica, principal vantagem deste formato. O valor de expoente presente na representação binária, exp, não é diretamente utilizado no cálculo do valor final do número representado. O valor de expoente efetivo do número, exp_{eff}, é dado por

$$exp_{eff} = exp - \left(\frac{2^{exp_l}}{2} - 1\right), \tag{3.34}$$

ou seja, ao expoente apresentado na representação binária de um número de vírgula flutuante é subtraído um *bias*, que age como um desnível que permite obter expoentes negativos. Isto possibilita dividir a gama dinâmica em valores de vírgula flutuante superiores e inferiores à unidade.

As operações aritméticas realizadas sobre este tipo de número são normalmente realizadas por módulos de cálculo dedicados, tais como as Unidades de Vírgula Flutuante, *Floating-Point Units* (*FPU*s), pois não podem ser realizadas das mesma forma que as operações de inteiros. A soma, por exemplo, requere determinar qual o maior e menor dos operados e colocá-los com o mesmo expoente antes da realização da operação de soma inteira entre as mantissas. A multiplicação é mais simples, dada a forma da representação numérica os expoentes podem ser somados e as mantissas multiplicadas. As operações de vírgula flutuante requerem mais recursos computacionais que as operações com vírgula fixa.

Estão ainda contempladas neste formato algumas representações de valores especiais. Estas representações incluem valores como ±∞ ou ±0 ou *NaN* (*Not a Number*). Existem também outros números, os números desnormalizados, que permitem obter valores absolutos mais baixos que os possíveis com a expressão (3.33). A Tabela 3.2 mostra as características que diferenciam estas representações especiais dos restantes números de vírgula flutuante [27].

Valor	bit sinal	expoente	mantissa
-0	1	0	0
+0	0	0	0
Núm. desnormalizado	0/1	0	≠0
-∞	1	Tudo a '1'	0
+∞	0	Tudo a '1'	0
NaN	0/1	Tudo a '1'	≠0

Tabela 3.2 - Representações de valores especiais em vírgula flutuante.

Os números desnormalizados, caracterizados por terem um expoente nulo e mantissa não nula, diferenciam-se dos números normalizados pelo facto de o '1' escondido da mantissa se tornar um '0'. Isto, aliado ao facto de o valor de expoente ser o menor possível, 1, é que permite obter números de valor absoluto mais baixo. O valor de um número desnormalizado é dado por

$$(-1)^S \times \left[\sum_{n=1}^{mant_l} mant(n) \times 2^{-n}\right] \times 2^{1-\left(\frac{2^{exp_l}}{2}-1\right)}, \qquad (3.35)$$

sendo que esta expressão pode ser transformada em

$$(-1)^S \times \left[\sum_{n=1}^{mant_l} mant(n) \times 2^{-n}\right] \times 2^{\frac{2^{exp_l}}{2}+2}. \qquad (3.36)$$

Na norma *IEEE 754* estão definidos os dois formatos de vírgula flutuante mais utilizados em sistemas computacionais: o formato de precisão simples e o formato de precisão dupla. O formato de precisão simples tem 32 bits, num total de 8 bits para o expoente e 23 bits para a mantissa. O formato de precisão dupla tem 64 bits, 11 para o expoente e 53 para a mantissa.

Em *FPGA* é ainda possível utilizar outros tipos de representações de vírgula flutuante, alterando a dimensão do expoente e da mantissa, pois as células lógicas das *FPGA* podem ser programadas para realizar operadores aritméticos personalizados. Em aplicações em que não seja necessária uma grande precisão do cálculo, o número de bits da mantissa pode ser reduzido, traduzindo-se numa menor utilização de recursos e maior frequência de funcionamento. A situação inversa também é possível, quando a aplicação justificar um aumento da precisão de cálculo à custa de uma maior utilização dos recursos disponíveis. A dimensão do expoente também pode ser adaptada para ajustar a gama dinâmica dos valores às necessidades da aplicação, influenciando também a área utilizada e a frequência de funcionamento.

3.5 Simulação do algoritmo integrado em MATLAB e C

Nesta secção é apresentado o algoritmo de simulação da evolução de um sistema de *N*-corpos, e realizados testes do mesmo. Para a realização do algoritmo foi escolhido o método de integração numérica *Velocity Verlet*. Foi escolhido este método, pois, comparando com os outros métodos apresentados, permite uma redução do erro face ao algoritmo de *Euler*, sem um grande aumento de complexidade e tem uma complexidade menor que a do método de *Runge-Kutta*. São considerados dois métodos de cálculo da aceleração, o método direto e o método de Barnes e Hut, para se realizar uma comparação dos mesmos.

O algoritmo usa a representação numérica de vírgula flutuante, pois esta tem uma maior gama dinâmica que a representação de vírgula fixa, adaptando-se melhor ao problema em causa, dadas as diferenças de ordem de grandeza dos valores processados. Por exemplo, a constante de gravitação universal *G* assume o valor 6.674210×10^{-11} $m^3kg^{-1}s^{-2}$ e a massa do Sol é cerca de $1{,}989 \times 10^{30}$ kg.

Foi realizada uma versão em *software* do algoritmo integrado de cálculo da aceleração e integração numérica, primeiro em linguagem MATLAB, e depois em C. Esta versão serve de referência de comparação do tempo de execução do algoritmo.

Aos passos que compõem o algoritmo *Verlet* de velocidade, secção 3.2.2, foi adicionado um passo de cálculo inicial de $\vec{a}(t)$, para permitir o cálculo de $\vec{r}(t+\Delta t)$ na primeira iteração da simulação, visto considerar-se que as posições e velocidades iniciais dos corpos já são conhecidas. O algoritmo é o apresentado em seguida:

- passo 1 – cálculo da aceleração inicial, $\vec{a}(t)$;
- passo 2 – cálculo da posição seguinte de todos os corpos, $\vec{r}(t+\Delta t)$, segundo a equação (3.19):

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \frac{1}{2}\vec{a}(t)\Delta t^2;$$

- passo 3 – cálculo da aceleração de todos os corpos no intervalo de tempo seguinte, $\vec{a}(t+\Delta t)$, utilizando $\vec{r}(t+\Delta t)$;
- passo 4 – cálculo da velocidade seguinte $\vec{v}(t+\Delta t)$, segundo a equação (3.20):

$$\vec{v}(t+\Delta t) = \vec{v}(t) + \frac{1}{2}\big(\vec{a}(t) + \vec{a}(t+\Delta t)\big)\Delta t.$$

Cada iteração da simulação corresponde a uma sequência dos passos 2, 3 e 4.

Foram criadas duas versões deste algoritmo. Uma versão com cálculo direto das acelerações dos corpos, realizada utilizando as funções *acceleration_DirectG()* e *DirectG_VelVerlet()*, e uma versão que utiliza o método de Barnes e Hut para o cálculo das acelerações, que utiliza as funções *acceleration_BarnesHut()* e *BarnesHut_VelVerlet()*.

A função *acceleration_DirectG()* devolve um vetor com as acelerações dos corpos a partir de um vetor com posições e um vetor com massas. A função *DirectG_VelVerlet()* recebe o número de iterações da simulação, vetores com a informação de posições, velocidades e acelerações iniciais, para além de um vetor com as massas e o valor do intervalo temporal entre iterações da simulação. Nesta função podem ser identificados os passos 2, 3 e 4 do algoritmo:

- passo 2:

```
for i=1:n_body
      positions(:,i)=positions(:,i)+...
            velocities(:,i)*tstep+...
            0.5*accelerations(:,i)*tstep^2;
end
```

- passo 3:

```
for i=1:n_body
      pi=positions(:,i);

      for j=1:n_body
            if(i~=j)
                  d=norm(positions(:,j)-pi);
            accelerations_aux(:,i)=accelerations_aux(:,i)+...
                        G*Masses(j)*(positions(:,j)-pi)/d^3;
            end
      end
end
```

- passo 4:

```
for i=1:n_body
      velocities(:,i)=velocities(:,i)+...
      0.5*(accelerations(:,i)+...
                        accelerations_aux(:,i))*tstep;
end
```

Na versão do algoritmo que utiliza o método de Barnes e Hut, que inclui as funções *acceleration_BarnesHut()* e *BarnesHut_VelVerlet()*, o cálculo direto das acelerações é substituído pela construção da árvore e posterior utilização no cálculo da aceleração de cada corpo, como demonstrado no seguinte extrato de código:

```
node=build_tree(positions,Masses);

for i=1:n_body
      pi=positions(:,i);

      accelerations_aux(:,i)=walk_tree(node,pi,threshold);
end
```

A função *build_tree()* chama a função *build_first_node()*, passando como parâmetros as posições e as massas dos corpos. É também passada com parâmetro uma posição central correspondente à origem dos eixos do referencial, no entanto esta é genérica e pode ser utilizada com diferentes posições centrais. Visto esta versão do algoritmo de Barnes e Hut ter sido desenhada para um espaço com apenas 2 dimensões, são criadas 4 vetores vazios de corpos e massas, sendo que estes correspondem aos nós que representam cada um dos quatros quadrantes do espaço bidimensional. Uma variável com a dimensão total da área em que os corpos estão contidos é iniciada com valores nulos das dimensões em *x* e *y*.

Em seguida são analisadas uma a uma as posições dos corpos. Sempre que o módulo de uma das componentes da distância de um dos corpos à posição central ultrapassa o valor de uma das componentes da área total, esta última é atualizada. Os corpos e suas massas são inseridos nos vetores dos quadrantes correspondentes conforme as posições assumidas pelos mesmos corpos.

É criada uma estrutura que alberga os nós-filho. Cada um dos 4 vetores com corpos é percorrido. Caso o vetor esteja vazio não é necessário criar um nó-filho para aquele quadrante. Caso apenas exista um corpo no vetor, o nó-filho será uma folha da árvore, tendo a dimensão da área representada por esse nó o valor 0. Caso existam vários corpos no vetor é obtido o ponto central da área representada pelo nó-filho e chamada a função *build_node()*, passando como parâmetros a dimensão da área, o seu ponto central, os corpos pertencentes à mesma área e as suas massas. Este comportamento é descrito pelo seguinte excerto de código MATLAB:

```
for i=1:4
    if(size(subs(i).pos,2)~=0)
      if(size(subs(i).pos,2)==1)
          sub_nodes_number=sub_nodes_number+1;
          sub_nodes(sub_nodes_number)=struct('AreaDim',0,'Position',...
                          subs(i).pos,'Mass',subs(i).mas,'SubNodes',[]);
      else
          switch(i)
            case 1
                  sub_central_pos=central_pos+[sub_area_dim(1)/2
                                               sub_area_dim(2)/2];
            case 2
                  sub_central_pos=central_pos+[-sub_area_dim(1)/2
                                               sub_area_dim(2)/2];
            case 3
                  sub_central_pos=central_pos+[-sub_area_dim(1)/2
                                               -sub_area_dim(2)/2];
            case 4
                  sub_central_pos=central_pos+[sub_area_dim(1)/2
                                               -sub_area_dim(2)/2];
          end

          sub_nodes_number=sub_nodes_number+1;
          sub_nodes(sub_nodes_number)=build_node(sub_area_dim,...
                               sub_central_pos,subs(i).pos,subs(i).mas);
      end
    end
end
```

Terminado o ciclo é testado se a raiz da árvore tem apenas um nó-filho. Se esta possibilidade se provar verdadeira, a raiz da árvore passa a ser o seu nó-filho, de forma a eliminar um nível desnecessário da árvore. Caso existam vários nós-filho, a posição do nó-pai é uma média ponderada das posições dos filhos em função das suas massas. É criada a estrutura que representa o nó, contendo o valor da área representada, a sua posição e massa e os nós-filho. O valor da área representada é obtido pelo cálculo do quadrado da norma da diagonal da área.

A função *build_node()* tem o mesmo objetivo da função *build_first_node()*, a criação da estrutura que representa o nó e distribuição dos corpos pelos nós-filho de forma a que estes sejam também processados, mas não possui o ciclo inicial de determinação da área retangular total ocupada pelos corpos. Esta função é utilizada de forma recursiva até cada corpo do sistema estar colocado numa folha da árvore.

A função *walk_tree()* permite o cálculo das acelerações dos corpos. Para cada corpo a árvore é percorrida começando pela raiz. É obtida a posição do nó e retirada a sua distância ao quadrado em relação à posição do corpo para o qual está a ser calculada a aceleração. O valor da diagonal ao quadrado da área representada, guardada em *node.Area_dim* é dividida pela distância ao quadrado. Se este rácio for maior que o valor de *threshold* recebido pela função, então a aceleração é calculada como sendo a soma das acelerações obtidas pela chamada recursiva desta função para os nós-filho. Caso contrário é utilizada a expressão (3.7) para realizar o cálculo da aceleração provocada pelo nó no corpo. Esta expressão permite aproveitar para o cálculo o valor já calculado da distância ao quadrado. O seguinte extrato código da função realiza o procedimento indicado:

```
d2=sum((node.Position-position).^2);
if(d2~=0)
      if(node.AreaDim/d2>threshold)
         nsub=size(node.SubNodes,2);
         acceleration=zeros(size(position));
         for i=1:nsub
               acceleration=acceleration+...
                  walk_tree(node.SubNodes(i),position,threshold);
         end
      else
         d=sqrt(d2);
         acceleration=G*node.Mass*(node.Position-position)/d^3;
      end
else
      acceleration=zeros(size(position));
end
```

Para testar o desempenho dos algoritmos desenvolvidos foi criado um cenário de simulação com 100 iterações para 100 corpos gerados aleatoriamente. Estas simulações foram realizadas num computador genérico com um processador Intel(R) Core(TM) i7-3632QM com 8 processadores lógicos a funcionar a uma frequência de 2.2GHz. Visto não terem sido utilizadas técnicas de paralelização do código desenvolvido este apenas utiliza um processador lógico.

Foi obtido um tempo de execução de 1,815 segundos para o método direto e 62,707 segundos para o método Barnes-Hut com *threshold* nulo. Ao utilizar-se um *threshold* nulo o cálculo da aceleração é sempre realizado com as folhas da árvore, resultando num número de cálculos de aceleração igual aos realizados no método direto. A diferença entre os dois valores de tempo de simulação, 60,892 segundos, corresponde então ao tempo de construção e utilização da árvore. O tempo extra de simulação derivado da construção e utilização da árvore tem um impacto menor no tempo de simulação total quando são utilizados valores maiores de *threshold* e o sistema possui um maior número de corpos. Nestas situações são realizados mais cálculos com os nós da árvore, tornando-se este método mais vantajoso que o direto, pois o tempo de simulação é menor. Isto será demonstrado posteriormente neste capítulo.

Foi ainda criado um cenário de simulação com 1000 corpos, para o qual se obteve um tempo de processamento para o algoritmo de cálculo direto de 183,884 segundos, o que confirma a complexidade $\mathcal{O}(N^2)$ deste algoritmo, pois o número de corpos foi aumentado 10 vezes e o tempo de simulação aumentou aproximadamente 100 vezes.

Foi realizada uma análise do desempenho do algoritmo de Barnes e Hut, cujos resultados são apresentados na Figura 3.6 e na Figura 3.7. Na Figura 3.6 podemos verificar uma diminuição esperada do tempo de execução com o aumento do valor de *threshold*, que se explica por menores tempos de percurso da árvore.

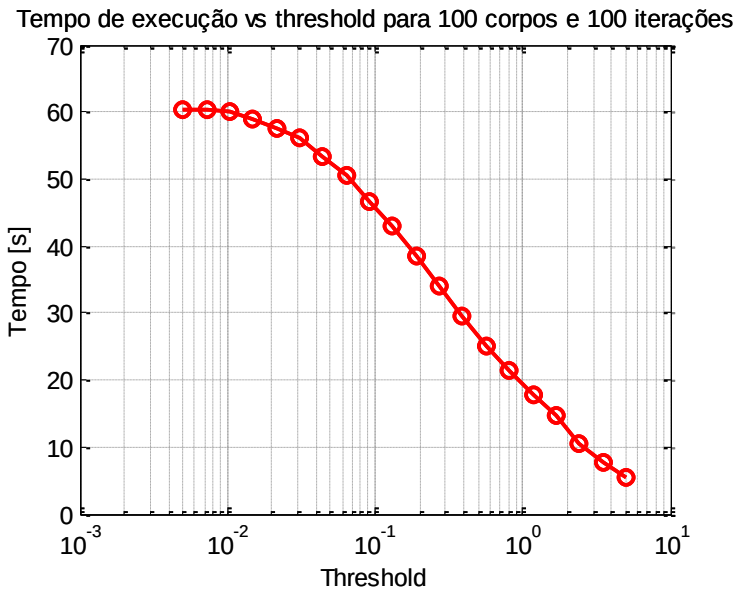


Figura 3.6 - Evolução do tempo de execução do algoritmo Barnes-Hut com o valor de threshold.

Na Figura 3.7 observamos um aumento esperado do erro relativo com o aumento do valor de *threshold*. Estes valores de erro foram obtidos comparando o resultado deste método com o do método direto. A partir destas figuras podemos verificar que é possível variar obter uma redução do tempo de execução do algoritmo mantendo um valor de erro baixo.

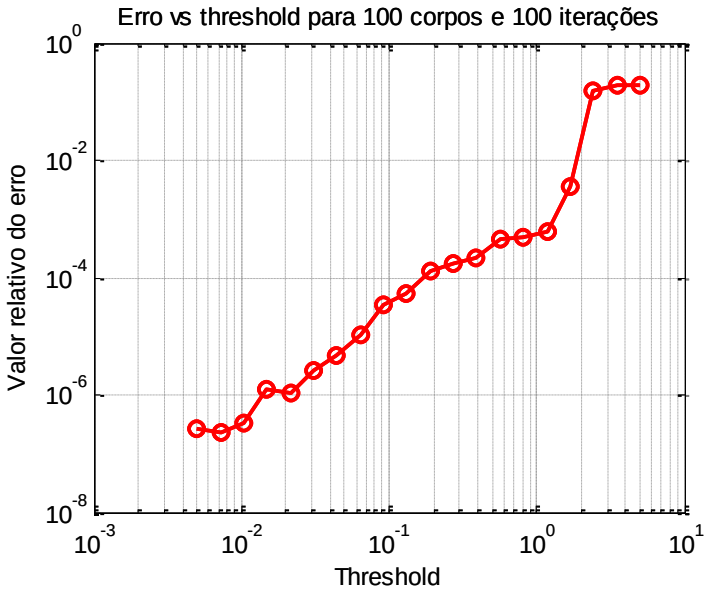


Figura 3.7 - Evolução do erro relativo resultante da aplicação do método Barnes-Hut em função do valor de threshold utilizado.

Uma outra versão destes algoritmos foi criada, mas em linguagem C e com o auxílio da plataforma de desenvolvimento *Visual Studio*. A necessidade de criação desta nova versão surge do facto de os algoritmos escritos na linguagem MATLAB terem um desempenho abaixo do nível ótimo, visto serem interpretados em tempo de execução. Por outro lado, quando passamos vetores como parâmetros de funções, não é utilizado o conceito de ponteiro, sendo a memória correspondente aos vetores copiada para o *stack* da função. Isto diminui o desempenho e aumenta o consumo de memória, principalmente em funções recursivas, como as utilizadas no método de Barnes e Hut.

Enquanto o algoritmo direto foi simplesmente traduzido, a forma do algoritmo Barnes-Hut foi otimizada, com inserção dos corpos na árvore um a um, ao invés de passar como parâmetro da função de construção dos nós-filho o conjunto de todos os corpos pertencentes a cada um deles. Esta nova versão deste último algoritmo foi desenvolvida para suportar num espaço a três dimensões.

Na Figura 3.8 podemos verificar os tempos de simulação obtidos para os algoritmos a 3 dimensões em C. Estes tempos de simulação são também referentes a conjuntos de 100 iterações. O valor de *threshold* utilizado para o algoritmo Barnes-Hut foi 0,25. Este valor permite uma redução do tempo de execução de cerca de 40 % mantendo o erro relativo em 0,015 %.

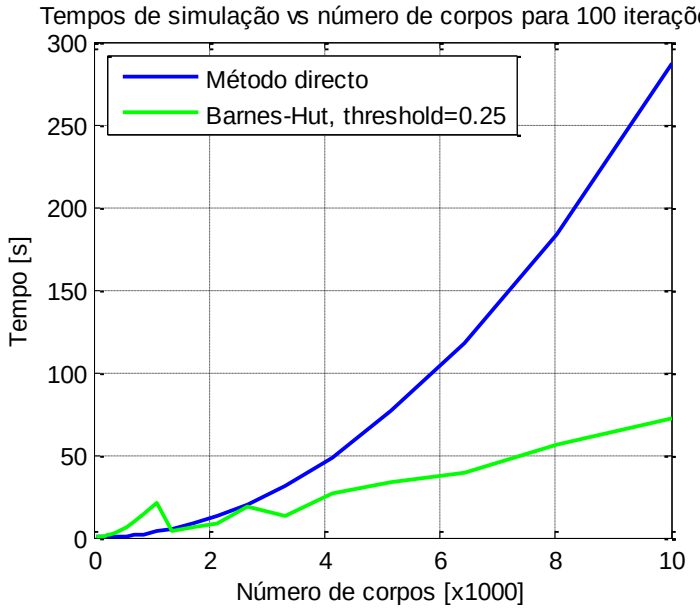


Figura 3.8 - Comparação entre os tempos de simulação do método direto e do método de Barnes e Hut.

Para 100 corpos, os valores apresentados são menores que os obtidos anteriormente em linguagem MATLAB a 2 dimensões, 1,815 segundos para o método direto e cerca de 10 segundos para método Barnes-Hut com o mesmo valor de *threshold*, tempo de execução observável na Figura 3.6. Isto demonstra a vantagem da utilização da linguagem C e da otimização do algoritmo de Barnes e Hut. A partir de cerca de 1300 corpos torna-se vantajosa a utilização do método de Barnes e Hut.

Todos os resultados apresentados foram obtidos utilizando números de vírgula flutuante de precisão dupla. Foi ainda realizada uma comparação do resultado da simulação do sistema de *N*-corpos utilizando esta precisão e a precisão simples. Apenas foi comparado o resultado para o algoritmo de cálculo direto das forças gravíticas. Como dados de simulação foram utilizadas as posições e velocidades de alguns corpos do Sistema Solar, disponíveis ao público na página da *Navigation and Ancillary Information Facility* (*NAIF*) da *National Aeronautics And Space Administration* (*NASA*). Na simulação foi utilizado um intervalo temporal, *tstep*, de 100 segundos e um período total de 365 dias. Comparando as posições finais dos corpos usando precisão simples com as posições finais usando precisão dupla, obtemos um erro médio relativo de 6.0686×10^{-5} para a precisão simples, usando a precisão dupla como referência. É esperado um aumento do erro de simulação com a diminuição da precisão.

3.6 Caracterização do sistema de *hardware*

O sistema é descrito em *VHDL* (*VHSIC Hardware Description Language*) e desenvolvido no ambiente de projeto *Integrated Software Environment* (*ISE*) *Project Navigator* da Xilinx. Foram utilizados *Intellectual Property Cores* (*IPCore*s) da Xilinx, disponíveis no *ISE*, para implementar os módulos que realizam as operações de vírgula flutuante realizadas neste módulo e no resto do simulador. Estes módulos têm estruturas de *pipeline*. Este tipo de estruturas permite obter um resultado a cada ciclo do relógio, resultando em uma taxa de cálculo superior. É utilizada a representação numérica de vírgula flutuante, em conformidade com a implementação em *software*.

Também foi estudada a utilização dos módulos de vírgula flutuante da *RPL VFloat Library* [31] como uma alternativa aos módulos da Xilinx. Esta biblioteca disponibiliza vários componentes de vírgula flutuante de interesse para este trabalho, tais como somadores e multiplicadores. No entanto, devido a um erro detetado num componente base dos somadores e subtratores, e ao facto de as áreas e frequências de funcionamento dos módulos obtidas não terem uma clara vantagem em relação às obtidas com os módulos da Xilinx, os operadores aritméticos desta biblioteca não foram utilizados, tendo sido apenas adaptada e utilizada a estrutura do acumulador, que permite a realização de uma operação a cada ciclo de relógio.

O sistema tem suporte físico numa *FPGA*, tendo sido disponibilizados kits de desenvolvimento com *FPGA* para o desenvolvimento deste trabalho. Os kits de desenvolvimento considerados para a realização deste trabalho foram os kits ML403 e ML402 [28] da Xilinx.

A placa de desenvolvimento ML403 inclui uma Virtex-4 XC4VFX12 [29]. As *FPGA* da família FX das Virtex-4 contêm um *PowerPC*, razão pela qual a quantidade de *flip flops* e *LUT* disponível neste tipo de *FPGA* é menor. A placa de desenvolvimento ML402 inclui uma Virtex-4 XC4VSX35 [29]. As *FPGA* da família SX estão otimizadas para processamento de sinal, incluindo um maior número de *XtremeDSP Slices*, que contêm um multiplicador de 18 bits, como referido na secção 3.3.2.

Os recursos disponíveis em cada uma das *FPGA* consideradas são apresentados na Tabela 3.3. Os valores de *flip flops* e *LUT* disponíveis foram obtidos a partir do *ISE*. A quantidade e tipo dos recursos disponíveis tiveram influência na estrutura do sistema, em especial na escolha dos módulos aritméticos usados.

Dev. Kit	FPGA	Slices	Flip Flops	LUTs	DSP Slices
ML403	XC4VFX12	5472	10944	10944	32
ML402	XC4VSX35	15360	30720	30720	192

Tabela 3.3 - Placas de desenvolvimento, FPGA correspondentes e respectivos recursos disponíveis.

4 Módulo de cálculo da aceleração em *hardware*

Nesta secção é apresentado o processo de projeto do módulo de *hardware* dedicado ao cálculo da aceleração. Este é o componente com maior impacto na área ocupada pelo sistema de simulação, visto o cálculo da aceleração ser paralelizado, como explicado no capítulo 5. Dado isto, a análise de desempenho em termos de área ocupada e capacidade de cálculo é usada para definir os módulos aritméticos utilizados em todo o sistema.

Para a realizar o cálculo da aceleração provocada pela força gravítica aplicada a cada corpo, foi desenvolvido um módulo dedicado ao cálculo do valor numérico tomado pela expressão (3.7):

$$\overrightarrow{a_{g_{1,2}}} = Gm_2 \frac{(x_2 - x_1)\overrightarrow{e_x} + (y_2 - y_1)\overrightarrow{e_y} + (z_2 - z_1)\overrightarrow{e_z}}{((x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2)}$$

$$\times \frac{1}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}}.$$

Esta forma da expressão permite o cálculo da aceleração sem recorrer ao cálculo de potências com expoente diferente de 2, o que possibilita a sua implementação com bibliotecas de módulo de *hardware* de vírgula flutuante adaptável, que normalmente não incluem módulos dedicados ao cálculo de potências.

Para concretizar esta estrutura são necessários 3 módulos de subtração, 2 de adição, 8 de multiplicação, 1 módulo de raiz quadrada e 1 módulo de divisão. O grafo de fluxo de dados pode ser observado na Figura 4.1. Os valores v_1 e v_2 correspondem a G e m_2.

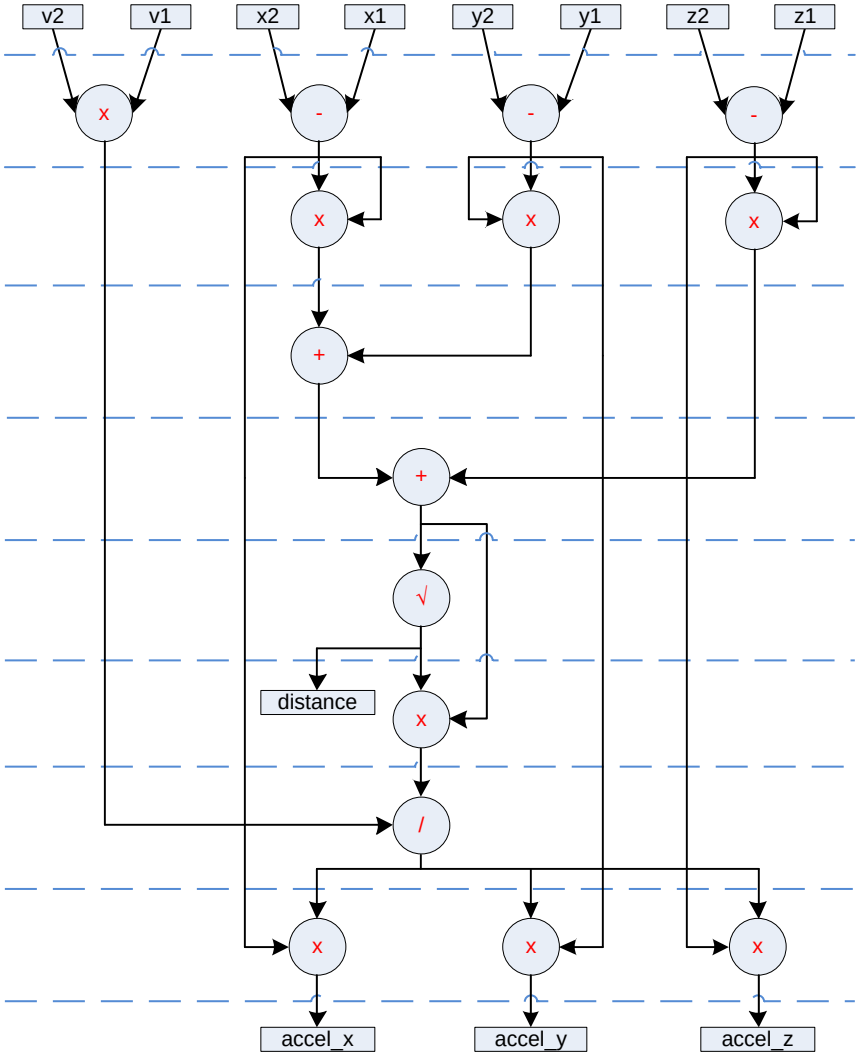


Figura 4.1 - Esquema de operações do cálculo da aceleração.

Para a correta implementação duma estrutura com *pipeline* que represente este esquema de operações é necessário compensar os atrasos das operações nos casos em que os dados à entrada de uma operação não tenham atrasos iguais. Também é adicionada uma linha de registos que propaga da entrada até à saída um sinal que é colocado a '1' no início da operação. Tendo esta linha de registos o mesmo atraso que as operações aritméticas realizadas, será apresentado o valor lógico '1' à saída quando o cálculo estiver terminado.

4.1 Estudo da ocupação de recursos utilizando precisão dupla (64 bits)

Numa primeira fase, foi escolhida a dimensão de 64 bits para os valores numéricos em causa, conforme o formato de precisão dupla apresentado na secção 3.4.2. Para reduzir os recursos utilizados não foram utilizados sinais de controlo dos módulos, de exceção ou de *handshaking*.

A Tabela 4.1 e a Tabela 4.2 mostram os recursos utilizados por algumas configurações dos módulos aritméticos testadas. Os resultados apresentados dizem respeito à síntese dos módulos.

		Slices	LUTs	Atraso máx. [ns]
Latência nula	somador	632	1142	28,2
	subtractor	632	1142	28,2
	multiplicador	1740	3278	24,2
	divisor	1602	3180	249
	raíz quadrada	980	1827	183

Tabela 4.1 - Recursos utilizados pelos módulos aritméticos combinatórios de vírgula flutuante de 64 bits.

Operadores de 64 bits			Slices	Flip Flops	LUTs	Ciclos de latência	Máx freq. [MHz]
Latência máxima	1 ciclo	somador	826	1152	1213	14	265
		subtractor	826	1152	1214	14	265
		multiplicador	1393	2454	2310	9	169
		raíz quadrada	1843	3284	1896	57	201
		divisor	3418	5982	3303	57	203
	Ciclos máximos	raíz quadrada	283	386	435	57	201
		divisor	252	372	396	57	170

Tabela 4.2 - Recursos utilizados pelos módulos aritméticos de vírgula flutuante de 64 bits.

Na Tabela 4.1 são visíveis os recursos utilizados ao escolher-se uma latência nula para os módulos, ou seja, uma implementação combinatória dos mesmos. A quantidade de recursos utilizada é elevada. O multiplicador utiliza cerca de 32% dos *slices* disponíveis na *FPGA* XC4VFX12 e 11% dos disponíveis na XC4VSX35. Os atrasos são também elevados, correspondendo o atraso do divisor a uma frequência máxima de operação de 3,93 MHz, se fossem colocados registos à entrada e à saída do módulo.

Na Tabela 4.2 são observáveis os recursos utilizados pelos mesmos módulos, mas utilizando a latência máxima possível. O número de ciclos de latência máximos é elevado, principalmente no caso da divisão e raiz quadrada, pois os atrasos dos módulos combinatórios são também elevados. Como esperado o número de *LUT* utilizadas em geral não varia muito em relação à realização combinatória, mas a elevada utilização de *flip flops* provoca um aumento do número

de *slices* utilizados. O multiplicador surge como uma exceção a este aumento de área, já que a implementação em *pipeline* utiliza menos recursos que a combinatória.

Também estão presentes os recursos utilizados pela raiz quadrada e divisor com uma implementação iterativa, necessitando a raiz quadrada de 54 ciclos para ser concluída, e a divisão de 55 ciclos. Os recursos utilizados baixam bastante, mas este tipo de módulo não pode ser utilizado numa estrutura de *pipeline*, pois não permite a obtenção de um novo resultado a cada ciclo de relógio. A utilização de módulos cíclicos implicaria a paralisação da estrutura de *pipeline* durante os ciclos necessários para concluir as operações, reduzindo bastante a taxa de cálculo.

Na Tabela 4.3 e na Tabela 4.4 é apresentado um cálculo dos recursos utilizados do módulo de cálculo da aceleração, utilizando uma solução combinatória e uma solução com *pipeline*, respetivamente, tendo por base o número de operações de cada tipo e os recursos utilizados individualmente pelo módulos. Na Tabela 4.5 são visíveis os recursos utilizados indicados pela ferramenta de síntese.

		Slices	LUTs	Num. Oper.	Total Sli.	Total LUTs
Latência nula	sub64	632	1142	3	1896	3426
	add64	632	1142	2	1264	2284
	mul64	1740	3278	8	13920	26224
	div64	1602	3180	1	1602	3180
	sqrt64	980	1827	1	980	1827
				Total	19662	36941

Tabela 4.3 - Estimativa do total de recursos utilizados para implementar o cálculo da aceleração de forma combinatória a 64 bits.

		Slices	Flip Flops	LUTs	Num. Oper.	Total Sli.	Total FFs	Total LUTs
Latência máx.	sub64	826	1152	1214	3	2478	3456	3642
	add64	826	1152	1213	2	1652	2304	2426
	mul64	1393	2454	2310	8	11144	19632	18480
	div64	3418	5982	3303	1	3418	5982	3303
	sqrt64	1843	3284	1896	1	1843	3284	1896
					Total	20535	34658	29747

Tabela 4.4 - Estimativa do total de recursos utilizados para implementar o cálculo da aceleração com pipeline a 64 bits.

	Latência	Slices	Flip Flops	LUTs	Mul. 18x18	Máx. Freq [MHz]	Atraso máx. [ns]
64 bits	0	19662	NA	36941	0	-	581
	Máx.	22160	35610	32832	0	169	-

Tabela 4.5 - Recursos utilizados pelas soluções analisadas de 64 bits.

Verifica-se que no caso combinatório a estimativa corresponde exatamente ao resultado da síntese. No caso com *pipeline* existe alguma diferença entre os resultados. No entanto, deve ser tido em conta o valor indicado na Tabela 4.5 inclui a utilização das linhas de compensação de atraso das operações.

Se as linhas de atraso fossem implementadas utilizando *flip flops* seria utilizada uma quantidade bastante superior de *flip flops* que a indicada. No caso das 3 maiores linhas de atraso, que se prolongam desde a primeira linha de subtrações até às últimas multiplicações, e que compensam o atraso de 2 multiplicações, 2 somas, 1 raiz quadrada e 1 divisão, e dado terem uma largura de 64 bits, obtém-se um número total de *flip flops* igual a:

$$3 \times \big((2 \times 9 + 2 \times 14 + 2 \times 57) \times 64\big) = 30720,$$

sendo que a este número deve ser somado o valor necessário para criar as restantes 3 linhas de atraso. A linha de atraso que se estende desde a primeira multiplicação até à divisão tem a particularidade de depender da diferença de ciclos de latência entre as subtrações e a multiplicação. Esta linha de atraso necessitaria de

$$\big((14 - 9) + 2 \times 9 + 2 \times 14 + 57\big) \times 64 = 6912$$

flip flops para ser implementada. As outras duas linhas, uma que compensa o atraso da soma de $(x_2 - x_1)^2$ com $(y_2 - y_1)^2$, e a que compensa o atraso da raiz quadrada fazem um total de

$$896 + 3648 = 4544$$

flip flops. Finalmente, o valor total de *flip flops* necessários para implementar as linhas de atraso seria igual a

$$30720 + 6912 + 4544 = 42176.$$

No entanto, visto as *LUT* presentes nas Virtex-4 poderem ser utilizadas com registos de deslocamento de 16 bits ou dimensão mais reduzida, seria apenas necessário um número superior a

$$\frac{42176}{16} = 2636$$

LUT para realizar as compensações de atraso, visto estes atrasos não serem todos múltiplos de 16. Isto explica a diferença entre a estimativa e o resultado da síntese no número de *LUT* gastas,

$$32832 - 29747 = 3085,$$

e também a diferença no número de *flip flops*, pois estes dois tipos de recursos estarão a ser utilizados para implementar as linhas de atraso.

Verifica-se que nenhuma das *FPGA* disponíveis tem capacidade para alojar um módulo de cálculo de 64 bits. Por esta razão foi decidido reduzir a dimensão das variáveis para 32 bits.

4.2 Estudo da ocupação de recursos utilizando precisão simples (32 bits)

Em seguida foi estudado o uso de uma dimensão de 32 bits para os valores numéricos, de acordo com o formato de precisão simples apresentado na secção 3.4.2.

À semelhança da Tabela 4.2, a Tabela 4.6 mostra os recursos utilizados pelos módulos aritméticos de 32 bits. Verifica-se uma descida substancial dos recursos utilizados por cada módulo e um aumento da frequência máxima de operação, como descrito na Tabela 4.7. A descida de utilização de recursos mais importante é a dos multiplicadores, visto esta ser a operação aritmética mais utilizada e visto passarem a utilizar 28% dos recursos anteriores.

Operadores de 32 bits			Slices	Flip Flops	LUTs	Ciclos de latência	Máx freq. [MHz]
Latência máxima	1 ciclo	somador	401	581	577	13	352
		subtractor	400	581	578	13	352
		multiplicador	398	701	640	8	255
		raíz quadrada	464	808	509	28	268
		divisor	767	1352	799	28	268
	Ciclos máximos	raíz quadrada	134	200	210	28	268
		divisor	146	209	239	28	258

Tabela 4.6 - Recursos utilizados pelos módulos aritméticos de vírgula flutuante de 32 bits.

			Slices	Flip Flops	LUTs	Ciclos de latência	Máx freq.
Latência máxima	1 ciclo	somador	0,49	0,50	0,48	0,93	1,33
		subtractor	0,48	0,50	0,48	0,93	1,33
		multiplicador	0,29	0,29	0,28	0,89	1,51
		raíz quadrada	0,25	0,25	0,27	0,49	1,33
		divisor	0,22	0,23	0,24	0,49	1,32
	Ciclos máximos	raíz quadrada	0,47	0,52	0,48	0,49	1,33
		divisor	0,58	0,56	0,60	0,49	1,52

Tabela 4.7 - Razão entre os recursos, latência e frequência máxima de funcionamento dos módulos de 32 bits e os módulos de 64 bits.

Também para o caso de 32 bits foi realizada uma estimativa dos recursos utilizados pela versão final do cálculo da expressão, apresentada na Tabela 4.8. Verifica-se que a maior parte dos recursos é utilizada pelas multiplicações, seguindo-se das subtrações e somas, apesar de serem os módulos de menores dimensões, devido ao facto de a expressão só incluir uma divisão e uma raiz quadrada. Tal como anteriormente, os valores obtidos com a síntese do circuito, Tabela 4.10 são superiores aos da estimativa, devido à existência das linhas de atraso.

		Slices	Flip Flops	LUTs	Num. Oper.	Total Sli.	Total FFs	Total LUTs
Latência máx.	sub32	400	581	578	3	1200	1743	1734
	add32	401	581	577	2	802	1162	1154
	mul32	398	701	640	8	3184	5608	5120
	div32	767	1352	799	1	767	1352	799
	sqrt32	464	808	509	1	464	808	509
					Total	6417	10673	9316

Tabela 4.8 - Estimativa do total de recursos utilizados para implementar o cálculo da aceleração com pipeline a 32 bits.

Para esta dimensão das variáveis foi ainda estudada a possibilidade da utilização dos multiplicadores de 18 bits disponíveis nos módulos *XtremeDSP Slice*, 3.3.2. A utilização dos multiplicadores possibilita uma redução das *LUT* e *flip flops* utilizados, como mostrado na Tabela 4.9. Para a obtenção destes resultados foi selecionada a opção *Full Usage* para o número de multiplicadores a utilizar na interface de configuração dos *IPCore*s.

Operadores de 32 bits			Slices	Flip Flops	LUTs	Mult. 18x18	Ciclos de latência	Máx freq. [MHz]
Lat. máx.	1 ciclo	somador	319	475	336	4	16	330
		subtractor	319	475	337	4	16	330
		multiplicador	178	274	145	4	10	361

Tabela 4.9 - Recursos utilizados pelos módulos aritméticos de vírgula flutuante de 32 bits recorrendo aos multiplicadores de 18 bits.

A Tabela 4.10 mostra os recursos utilizados pela solução com módulos de máxima latência, seguida da solução com máxima latência e multiplicadores de 18 bits nas multiplicações e a solução com máxima latência e multiplicadores de 18 bits tanto nas multiplicações como nas subtrações e nas somas. Observa-se um ligeiro aumento da frequência máxima de operação na opção com multiplicadores de 18 bits, visto o desempenho em frequência das multiplicações melhorar com o uso dos mesmos, como observado através da análise da Tabela 4.6 e Tabela 4.9. As operações que limitam a frequência de funcionamento são agora a divisão e a raiz quadrada.

	Latência	Slices	Flip Flops	LUTs	Mul. 18x18	Máx. Freq [MHz]
32bits	Máx.	6972	11177	10349	0	255
	Máx.	5212	7761	6389	32	268
	Máx.	4805	7231	5184	52	268

Tabela 4.10 - Recursos utilizados pelas soluções analisadas de 32 bits.

Visto a força gravítica total aplicada a um corpo depender do somatório das forças geradas nesse corpo por todos os outros do conjunto, e em consequência, a aceleração total do corpo também depender do conjunto de um conjunto de contribuições, é útil utilizar um módulo

acumulador logo após cada uma 3 saídas do módulo que correspondem às componentes em x, y e z da aceleração.

De forma a implementar este acumulador foi utilizado um módulo somador, que inclui 4 multiplicadores de 18 bits, e foi realizada uma realimentação do mesmo, estando a saída ligada a uma das suas entradas por meio de uma fila de registos.

No entanto, surge uma questão relacionada com a validade dos dados à entrada dos acumuladores. Se 2 ou mais corpos se aproximarem de tal forma que a sua distância não possa ser representada pela representação numérica utilizada, assumindo o valor zero, o denominador da expressão da aceleração tomará o valor zero, resultando num valor de aceleração infinito. Com a aplicação da integração numérica, a posição dos corpos afetados passaria a ser inválida. No ciclo seguinte todas as posições passariam a ser inválidas, pois no numerador da expressão seriam realizadas subtrações com infinito.

Para prevenir esta situação, foi incluído um teste à distância entre os dois corpos, com o intuito de detetar quando a distância é zero. É testado se o valor corresponde ao zero positivo ou zero negativo. Neste caso, o sinal que indica ao acumulador quando tem dados para processar à entrada, resultante do sinal que indica que o cálculo da expressão foi terminado, é colocado a zero, evitando a acumulação.

É também incluído um contador que regista o número de operações aritméticas. Caso o sinal de início de operação e o sinal de fim de operação, após atraso correspondente ao atraso dos acumuladores, sejam iguais, o valor da contagem é mantido, pois ou não é iniciada nem terminada nenhuma operação, ou é iniciada uma operação e terminada outra. Caso apenas o sinal de início de operação esteja ativo o contador é incrementado. Caso apenas esteja ativo o sinal de fim de operação o contador é decrementado. O número máximo de operações realizadas corresponde ao número de níveis da estrutura de *pipeline*. O módulo *Gravi_core* integra o módulo de cálculo da expressão, *calc*, os acumuladores, o teste à distância entre corpos e a contagem de operações.

Para estudar o impacto do tipo de otimização utilizado no processo de síntese na área ocupada, este módulo foi sintetizado de duas formas: utilizando otimização de velocidade e otimização de área. Os resultados obtidos são apresentados na Tabela 4.11. Os recursos utilizados não variam muito, pois a síntese dos *IPCores* de vírgula flutuante não é afetada pelo tipo de otimização escolhida. O número de multiplicadores de 18 bits utilizados é 64, pois cada um dos 3 acumuladores acrescenta 4 multiplicadores.

	Método de optimização	Slices	Flip Flops	LUTs	Mul. 18x18	Máx. Freq [MHz]
Gravi_core 32bits	Velocidade	6049	9087	6423	64	268
	Área	5967	8988	6429	64	268

Tabela 4.11 - Recursos utilizados com otimização de velocidade e área para módulo de 32 bits.

Todavia, visto o objetivo deste trabalho ser a realização de um sistema de multiprocessamento, e dado o facto de o número de *slices* utilizados por apenas um módulo de cálculo da aceleração ser 42% do número total presente na *FPGA* disponível com maior capacidade, a XC4VSX35, foi decidido reduzir mais uma vez a dimensão da representação numérica utilizada. A possibilidade de utilização da *FPGA* XC4VFX12 foi descartada.

4.3 Estudo da ocupação de recursos utilizando 24 bits

A nova representação numérica usa 7 bits para o expoente e 16 bits para a mantissa, num total de 24 bits. A dimensão da mantissa foi escolhida de forma a permitir que as multiplicações da mesma fossem realizadas utilizando apenas um multiplicador de 18 bits.

Numa primeira fase esta nova dimensão foi apenas aplicada ao módulo *calc*, não sendo alterada a precisão dos acumuladores. De forma a manter a interface deste módulo com a dimensão anterior, foram introduzidos módulos de conversão de formato.

A área utilizada pelos novos operadores aritméticos é mostrada na Tabela 4.12. Na Tabela 4.13 observa-se a razão entre os recursos utilizados pelos módulos de 24 bits e os módulos de 32 bits utilizados na versão anterior de *Gravi_core*. Pode-se verificar uma redução geral da área utilizada, principalmente nos operadores de multiplicação, divisão e raiz quadrada. Na soma a redução do número de *slices* não é tão significativa, existindo mesmo um aumento do número de *LUT*, pois a opção para utilizar multiplicadores de 18 bits já não está disponível para o tipo de representação numérica agora usada. Verifica-se que os módulos de conversão de formato ocupam uma área bastante reduzida.

	Operadores de 24 bits	Slices	Flip Flops	LUTs	Mult. 18x18	Ciclos de latência	Máx freq. [MHz]
Latência máxima	somador	305	422	420	0	12	270
	subtractor	305	422	421	0	12	270
	multiplicador	90	125	87	1	6	328
	raíz quadrada	274	461	304	0	21	268
	divisor	419	736	442	0	21	343
	conv. 32 para 24	55	77	73	0	3	332
	conv 24 para 32	34	52	33	0	2	610

Tabela 4.12 - Recursos utilizados pelos módulos aritméticos de vírgula flutuante de 24 bits.

		Slices	Flip Flops	LUTs	Mult. 18x18	Ciclos de latência	Máx freq.
Latência máxima	somador	0,96	0,89	1,25	0,00	0,75	0,82
	subtractor	0,96	0,89	1,25	0,00	0,75	0,82
	multiplicador	0,51	0,46	0,60	0,25	0,60	0,91
	raíz quadrada	0,59	0,57	0,60	-	0,75	1,00
	divisor	0,55	0,54	0,55	-	0,75	1,28

Tabela 4.13 - Rácio entre os recursos, latência e frequência máxima de funcionamento dos módulos de 24 bits e os módulos de 32 bits.

A Tabela 4.14 apresenta os resultados da síntese do módulo *Gravi_core*. Comparando esta nova versão com a versão totalmente realizada com uma precisão de 32 bits, verifica-se uma redução de 15% dos recursos utilizados, ocupando este módulo um total de 33% dos *slices* da *FPGA*. Este valor continua a ser elevado, principalmente devido ao facto de não incluir a área ocupada pelos integradores numéricos. Mais uma vez não é observável uma diferença considerável entre as áreas utilizadas com otimização de velocidade e área no processo de síntese.

	Método de optimização	Slices	Flip Flops	LUTs	Mul. 18x18	Máx. Freq [MHz]
Gravi_core 24/32bits	Velocidade	5135	7378	6170	20	259
	Área	5049	7279	6170	20	268

Tabela 4.14 - Recursos utilizados pelo módulo Gravi_core com optimização de velocidade e optimização de área.

No entanto, a área ocupada pelo módulo pode ser reduzida ao utilizar apenas 24 bits também nos acumuladores. Por outro lado, dada a forma como o acumulador foi implementado, é necessário esperar que o resultado da soma à saída do acumulador percorra toda a linha de registos de realimentação para se poder realizar uma nova operação, resultando numa espera com número de ciclos de relógio igual ao número de ciclos de latência da soma. A frequência de cálculo efetiva será dada pela frequência de operação a dividir pelo número de ciclos de espera por operação.

De forma a analisar qual o valor de número de ciclos de latência do somador deste acumulador que permite o melhor desempenho, foi realizado o estudo presente na Tabela 4.15. Após a divisão da frequência máxima de operação pelo número de ciclos de latência, observa-se que a frequência efetiva de cálculo diminui com o aumento da latência do somador. O valor indicado de frequência de operação do somador combinatório foi obtido colocando uma linha de registos nas entradas e uma linha de registos na saída do mesmo. No caso do somador com um ciclo de latência foi colocada uma linha de registos nas entradas.

Análise de somador com formato (7,16)		
Ciclos de latência	Máx. freq. [MHz]	Máx. freq./Num. ciclos [MHz]
0	43,2	-
1	44,4	44,4
2	86,7	43,4
3	127	42,3
4	168	42,0
5	174	34,8
6	204	34,0
7	204	29,1
8	233	29,1
9	211	23,4
10	211	21,1
11	270	24,5
12	270	22,5

Tabela 4.15 - Análise de recursos utilizados e frequência de utilização para somador com 7 bits de expoente e 16 bits de mantissa (resultados de síntese).

Dado este resultado foi definido que o número de ciclos de latência a utilizar seria um. Todavia é necessário poder reiniciar o valor à saída dos acumuladores, para evitar que o resultado de uma iteração da simulação seja afetado pelo resultado da anterior. Visto não ser possível gerar o circuito de soma com um sinal de reiniciação de saída, só estando disponível um sinal de reiniciação dos sinais controlo de fluxo e exceção do componente, foi utilizado o somador combinatório, sendo colocado um registo com sinal de reiniciação à saída do mesmo.

Como a frequência de utilização do circuito está limitada pelo acumulador a um valor de 43.2 MHz, não se deve optar por utilizar o número de ciclos de latência máximo nos outros operadores aritméticos, pois esta situação leva à utilização de um número de *flip flops* superior ao necessário. A latência dos outros operadores aritméticos foi reduzida de forma a reduzir os recursos utilizados por cada um e as frequências de funcionamento mas evitando que estes se tornassem em novos limitadores da frequência máxima de operação.

Os valores dos novos números de ciclos de latência e os recursos utilizados por cada componente do módulo *calc* são apresentados na Tabela 4.16. Os somadores utilizados neste módulo são diferentes dos utilizados no acumulador, pois têm um ciclo de latência. Esta opção foi tomada para evitar a soma dos atrasos da lógica combinatória de operações executadas em sequência. Na Tabela 4.17 podemos verificar que a redução da latência dos operadores aritméticos permite uma diminuição do número de *slices* utilizados para 57-78% do número utilizado com operadores com latência máxima. Esta redução é devida ao menor número de *flip flops* pretendido, sendo o número de *LUT* semelhante ao anterior. A frequência máxima de operação também desce, conforme esperado.

Operadores de 24 bits	Slices	Flip Flops	LUTs	Mult. 18x18	Ciclos de latência	Máx freq. [MHz]
somador	209	24	363	0	1	44
subtractor	237	68	408	0	2	87
multiplicador	43	24	79	1	1	94
raíz quadrada	177	107	282	0	4	67
divisor	239	127	437	0	4	57

Tabela 4.16 - Recursos utilizados pelos módulos aritméticos de vírgula flutuante de 24 bits com latência ajustada.

	Slices	Flip Flops	LUTs	Ciclos de latência	Máx freq.
somador	0,69	0,06	0,86	0,08	0,16
subtractor	0,78	0,16	0,97	0,17	0,32
multiplicador	0,48	0,19	0,91	0,17	0,29
raíz quadrada	0,65	0,23	0,93	0,19	0,25
divisor	0,57	0,17	0,99	0,19	0,17

Tabela 4.17 - Rácio entre os recursos, latência e frequência máxima de funcionamento dos módulos de 24 bits com latência máxima e os módulos de 24 bits com latência ajustada.

Visto todas as operações aritméticas serem realizadas a 24 bits, já não é necessário incluir conversores de formato no módulo *Gravi_core*. Na Tabela 4.18 observa-se o resultado da síntese deste módulo quanto a recursos e frequência máxima de operação. A frequência máxima de operação é a indicada Tabela 4.15 para o somador combinatório, como pretendido. Quanto à área ocupada, o número de *slices* é apenas 56% do obtido anteriormente, Tabela 4.14, pois o número de *flip flops* é reduzido para 16% do número anterior, como resultado da redução da latência do operador. Visto o número de *slices* utilizados corresponder a 19% dos presentes na *FPGA* utilizada torna-se viável criar um protótipo de uma arquitetura de multiprocessamento na mesma. Uma outra redução da precisão numérica permitiria um maior número de núcleos de processamento, mas o erro presente no resultado da simulação aumentaria.

	Método de optimização	Slices	Flip Flops	LUTs	Mul. 18x18	Máx. Freq [MHz]
Gravi_core 24 bits	Velocidade	2855	1204	4750	8	43

Tabela 4.18 - Recursos utilizados pelo módulo Gravi_core com operadores de 24 bits com latência ajustada.

4.4 Alteração da precisão para 25 bits e análise de recursos utilizados.

A redução da dimensão da mantissa foi realizada tendo em vista uma redução substancial dos recursos utilizados pelos multiplicadores. No entanto, não foram analisadas as consequências da diminuição da dimensão do expoente.

Visto o expoente de um número de vírgula flutuante determinar a gama de valores que pode ser representada, uma redução da sua dimensão implica uma diminuição do valor máximo representável. Numa simulação em que as variáveis representam grandezas como massas de corpos celestes, é importante ter em consideração o alcance da representação numérica utilizada. A expressão (3.33) pode ser utilizada para obter limites do valor absoluto máximo e mínimo representável para um dado formato de vírgula flutuante.

No cálculo do valor máximo representável, pode-se simplificar a expressão (3.33) de forma a obter-se uma expressão aproximada deste valor que seja função apenas da dimensão do expoente. Visto o maior valor representável pela mantissa ser dado por

$$1 + \sum_{n=1}^{mant_l} mant(n) \times 2^{-n}, \tag{4.1}$$

em que $mant_l$ representa o número de bits da mantissa e $mant(n)$ o enésimo bit da mesma, e visto

$$\lim_{mant_l \to \infty} \left[1 + \sum_{n=1}^{mant_l} mant(n) \times 2^{-n} \right] = 2, \tag{4.2}$$

pode-se afirmar que o máximo valor representável pela mantissa é aproximadamente 2. Esta aproximação é tanto melhor quanto maior o número de bits da mantissa.

Com base nas expressões (3.33), (3.34) e (4.2), e subtraindo uma unidade ao valor expoente máximo, pois o maior expoente possível corresponde a $+\infty$ ou $-\infty$, obtem-se o maior valor representável

$$2 \times 2^{\left(2^{exp_l} - 2 - \left(\frac{2^{exp_l}}{2} - 1\right)\right)} = 2^{\frac{2^{exp_l}}{2}} = 2^{2^{(exp_l - 1)}}, \tag{4.3}$$

sendo exp_l a dimensão em bits do expoente.

Visto os componentes aritméticos utilizados neste trabalho não suportarem números desnormalizados [30], a influência destes no valor absoluto mínimo representável pode ser desprezada. Sendo que estes números têm expoente nulo, tal como o zero positivo e o zero negativo, este valor de expoente não pode ser utilizado para o cálculo do valor absoluto mínimo

representável. Por outro lado, o menor valor que a mantissa de um número normalizado pode ter é 1, no caso em que todos os seus bits são nulos. Com base nestas afirmações pode-se obter o valor absoluto mínimo representável:

$$1 \times 2^{\left(1-\left(\frac{2^{exp_l}}{2}-1\right)\right)} = 2^{-\frac{2^{exp_l}}{2}+2} = 2^{-2^{(exp_l-1)}+2}. \quad (4.4)$$

Na Tabela 4.19 são observáveis os valores máximos e mínimos representáveis para vários valores de número de bits do expoente, incluindo o número de bits usado em precisão simples, 8, e o número de bits utilizado em precisão dupla, 11. Com base nestes valores foi concluído que a utilização de 7 bits não se adequa às necessidades da simulação a realizar. Por exemplo, o valor da massa do Sol é de $1{,}9891 \times 10^{30}$ kg, e o valor da massa da Terra $5{,}9722 \times 10^{24}$ kg.

Núm. de bits do expoente	Valor máximo representável	Valor mín. abs. representável
4	256	0,015625
5	65536	6,10E-05
6	4,29E+09	9,31E-10
7	1,84E+19	2,17E-19
8	3,40E+38	1,18E-38
9	1,16E+91	3,45E-77
10	1,34E+168	2,98E-154
11	1,80E+308	2.23E-308

Tabela 4.19 - Valores máximos e mínimos absolutos representáveis em função do número de bits do expoente.

Outra alteração foi realizada ao componente divisor. De forma a simplificar a lógica presente no módulo *Gravi_core*, e aproveitando o facto de poder ser incluído um sinal de indicação de divisão por zero no divisor, foi eliminado o teste à distância realizado no módulo *Gravi_core*, passando este teste a ser realizado internamente no divisor. Um porto de saída de indicação de exceção foi adicionado ao módulo *calc*. Para tal, foi acrescentada uma linha de atraso na saída correspondente do divisor para compensar o atraso das últimas três multiplicações. Com esta alteração foi possível eliminar o ciclo intermédio de teste da distância em *Gravi_core*.

Alterado o número de bits da representação numérica utilizada, foram obtidos os resultados de síntese presentes na Tabela 4.20. Comparativamente com os resultados da Tabela 4.16, não se observam alterações substanciais na área ocupada por cada operador, apesar da inclusão do sinal de exceção no divisor.

Operadores de 25 bits	Slices	Flip Flops	LUTs	Mult. 18x18	Ciclos de latência	Máx freq. [MHz]
somador	221	25	375	0	1	44
subtractor	245	70	421	0	2	86
multiplicador	44	25	83	1	1	93
raíz quadrada	178	110	284	0	4	67
divisor	245	131	446	0	4	57

Tabela 4.20 - Recursos utilizados pelos módulos aritméticos de vírgula flutuante de 25 bits.

Na Tabela 4.21 são apresentados os recursos gastos pela versão de 25 bits do módulo *Gravi_core*. Também não se observam grandes variações da quantidade de recursos utilizados em relação à versão anterior, Tabela 4.18. A diminuição do número de *flip flops* deve-se à eliminação do ciclo intermédio de teste à distância.

	Método de optimização	Slices	Flip Flops	LUTs	Mul. 18x18	Máx. Freq [MHz]
Gravi_core 25 bits	Velocidade	2894	1074	4856	8	43

Tabela 4.21 - Recursos utilizados pelo módulo Gravi_core com operadores de 25 bits.

4.5 Utilização da estrutura do acumulador da RPL Vfloat Library

A estrutura apresentada do acumulador em 4.3 tem o ponto negativo de resultar numa limitação da frequência máxima de funcionamento do sistema de cálculo. Uma possível abordagem para resolver o problema seria introduzir um ou mais acumuladores em paralelo, realizando as operações de soma alternadamente em cada acumulador. Seriam também necessários vários somadores que agrupassem as saídas dos acumuladores de forma a se obter o resultado total da acumulação. Isto permitiria utilizar acumuladores com maior latência, mas tem o inconveniente de resultar numa utilização de recursos maior.

A *RPL Vfloat Library* inclui um módulo acumulador de vírgula flutuante que permite receber um operando por ciclo de relógio sem a limitação de ter apenas um ciclo de latência [31]. A estrutura consiste num módulo somador com um módulo de normalização e um *buffer* à saída. Os valores à entrada do somador são controladas por uma máquina de estados, podendo ser provenientes da entrada do módulo ou do *buffer*, como mostrado na Figura 5.5 [31].

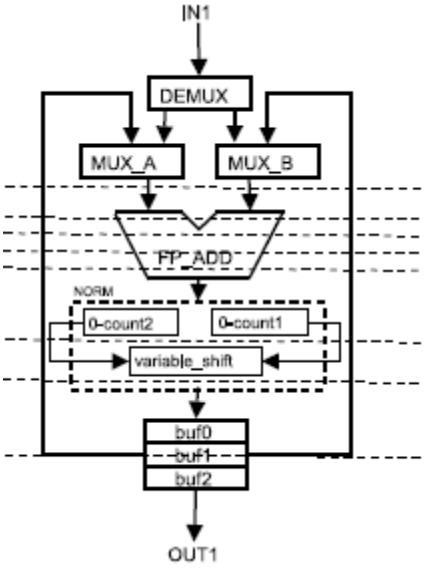


Figura 4.2 - Estrutura do acumulador da RPL Vfloat Library.

Porém, visto o somador disponível na biblioteca ter uma latência fixa, o módulo está limitado na sua frequência máxima de funcionamento, não sendo possível aumentar a mesma à custa da utilização de um maior número de *flip flops*. Visto isto, a estrutura foi alterada de forma a utilizar o módulo somador da Xilinx e o bloco de normalização foi removido. Com esta alteração a estrutura passa a ter uma latência configurável, desde que o contador interno de operações seja também alterado de forma a ter capacidade para registar um número de operações máximo igual à latência escolhida do somador.

4.6 Definição dos módulos aritméticos utilizados

Dada a possibilidade de configurar a latência de todos os módulos, foi analisada a relação frequência de funcionamento/recursos ocupados. Na Tabela 4.22 e na Tabela 4.23 são apresentados os resultados de área ocupada, frequência de funcionamento e relação entre estas para o multiplicador e o subtrator respetivamente. São apresentados os resultados após a fase de *Place and Route* pois estes melhor caracterizam o sistema final. Os resultados para o divisor e a raiz quadrada podem ser consultados em anexo. Estes módulos, e o somador, representam a maior parte do número das operações de vírgula flutuante, podendo a relação frequência de funcionamento/área ocupada do sistema ser analisada apenas tendo em conta estes módulos. Os resultados do somador são idênticos aos obtidos para o subtrator, razão pela qual não são apresentados. Observa-se uma tendência de subida da relação frequência de funcionamento/recursos ocupados para os módulos de soma e subtração. No multiplicador não se observa nenhuma tendência deste género. Com estes dados pode-se concluir que este fator de desempenho tende a subir com o número de ciclos de latência, pelo que será mais vantajoso utilizar latências elevadas nos módulos aritméticos do sistema.

Análise do multiplicador com formato (8,16) após P&R						
Ciclos de latência	Slices	Flip flops	LUTs	MUL18x18	Máx. freq. [MHz]	Máx freq/ Slices
0	54	-	98	1	90	3,20
1	44	25	84	1	93	2,12
2	46	40	81	1	173	3,76
3	76	75	106	1	204	2,68
4	78	82	107	1	255	3,27
5	96	114	92	1	306	3,19
6	108	130	98	1	313	2,90

Tabela 4.22 - Evolução da utilização de recursos e frequência de funcionamento do multiplicador.

Análise do subtrator com formato (8,16) após P&R					
Ciclos de latência	Slices	Flip flops	Total LUTs	Máx. freq. [MHz]	Máx freq/ Slices
0	207	-	405	47	0,40
1	197	25	387	48	0,24
2	219	70	427	83	0,38
3	244	116	444	126	0,52
4	241	159	437	151	0,63
5	260	186	455	158	0,61
6	258	226	442	207	0,80
7	262	244	444	201	0,77
8	280	277	464	219	0,78
9	283	305	468	234	0,83
10	301	327	469	234	0,78
11	312	360	458	303	0,97
12	361	429	466	235	0,65

Tabela 4.23 - Evolução da utilização de recursos e frequência de funcionamento do subtrator.

Foi definida uma frequência de 150 MHz para o funcionamento do sistema. Não tendo sido possível obter esta frequência com os módulos de latência mínima, a latência dos mesmos foi aumentada. Os valores finais de número de ciclos de latência são os mostrados em seguida:

- subtrator/somador: 6 ciclos de latência;
- multiplicador: 4 ciclos de latência;
- divisor: 11 ciclos de latência;
- raiz quadrada: 10 ciclos de latência.

Para estes valores é obtida a informação de recursos utilizados presente na Tabela 4.24.

Resultados após *P&R*	Método de optimização	Slices	Flip Flops	Total LUTs	Mul. 18x18	Máx. Freq [MHz]
Gravi_core 25 bits	Velocidade	4533	3793	6161	8	162

Tabela 4.24 - Utilização de recursos do módulo Gravi_core com frequência de funcionmanto melhorada.

5 Sistema de simulação de *N*-corpos

O sistema de simulação de *N*-corpos em *hardware* dedicado é implementado pelo módulo *nbody_processor*. Este corresponde a uma implementação em *hardware* do algoritmo apresentado no capítulo 3.5. Para a realização do algoritmo foram escolhidos o método de cálculo direto das interações entre os corpos e o método de integração numérica *Velocity Verlet.* O algoritmo direto de cálculo das interações gravíticas entre corpos foi escolhido pois é facilmente paralelizável, e dada a sua simplicidade é bom ponto de partida para uma implementação em *hardware*. A paralelização desta componente do algoritmo permite uma redução significativa do tempo de simulação, visto esta componente do algoritmo ter esta ter uma complexidade de $\mathcal{O}(N^2)$.

Neste capítulo é apresentada a arquitetura do sistema realizado, bem como os componentes que o constituem.

5.1 Arquitetura do sistema

O módulo *nbody_processor* contém as memórias ondem são armazenados os dados de simulação, o módulo gerador dos coeficientes de integração, conversores de formato que permitem realizar a tradução dos dados de vírgula flutuante entre *UARTcontrol*, apresentado na secção 6.2, e os elementos internos deste módulo, unidades de cálculo integrado da aceleração e integração numérica, e linhas de deslocamento de dados que transportam os dados das unidades de cálculo para as memórias.

Estão presentes 13 memórias no sistema, numeradas de 0 a 12:

- *pos_x_mem*, *pos_y_mem* e *pos_z_mem* – memórias que guardam as posições dos corpos;
- *vel_x_mem*, *vel_y_mem*, *vel_z_mem* – memórias que guardam as velocidades dos corpos;
- *acc_x_mem0*, *acc_y_mem0* e *acc_z_mem0* – primeiro conjunto de memórias que guardam as acelerações dos corpos;
- *acc_x_mem1*, *acc_y_mem1* e *acc_z_mem1* – segundo conjunto de memórias que guardam as acelerações dos corpos;
- *mass_mem* – memória que guarda as massas dos corpos.

Na Figura 5.1 podemos observar os vários módulos do sistema e de que forma os dados de simulação são transmitidos entre os mesmos. Esta estrutura é uma tradução para *hardware* do algoritmo realizado no capítulo 3.5, realizando o cálculo direto das acelerações.

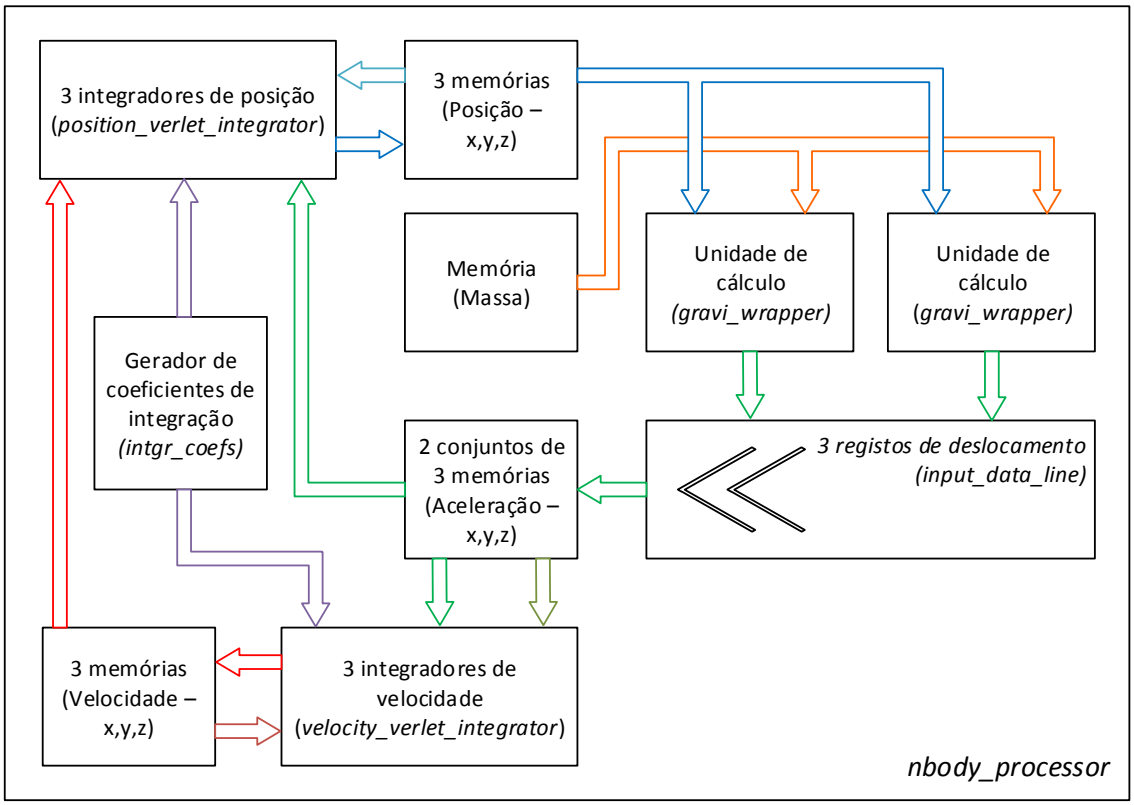


Figura 5.1 - Arquitetura e fluxo de dados do simulador.

A presença de dois conjuntos de memórias de armazenamento de acelerações deve-se ao facto de na integração das velocidades dos corpos serem utilizadas as acelerações atuais e anteriores dos corpos, sendo necessário guardar dados de aceleração respetiva ao passo atual e ao passo anterior da simulação. Quando um passo de simulação é terminado, ao invés de copiar a informação de aceleração atual para as memórias com as acelerações anteriores, a função dos conjuntos de memória é simplesmente trocada, por meio de um sinal binário interno, *mem_selector*, que controla a multiplexagem à entrada e à saída destas memórias. Quando este sinal assume o valor '0', o primeiro conjunto de memórias guarda as acelerações anteriores e o segundo conjunto as atuais, sendo a escrita de dados realizada no segundo conjunto. Quando o sinal toma o valor '1', a funcionalidade das memórias é invertida e a escrita de dados é efetuada no primeiro conjunto. Em cada passo de simulação as acelerações atuais tornam-se as anteriores e as restantes são descartadas. Este procedimento é repetido em todos os passos da simulação.

Os módulos de integração numérica de cada uma das coordenadas da posição, *x*, *y* e *z*, recebem os dados das memórias com os dados de posição, velocidade e aceleração anterior para realizarem o cálculo das novas posições, que são guardadas nas memórias de posição, eliminando os dados anteriores. De forma semelhante, os integradores de velocidade recebem os dados da velocidade e aceleração anteriores, bem como da aceleração atual, sendo as novas velocidades colocadas nas respetivas memórias, também com eliminação dos dados anteriores.

Nos passos do processo de integração numérica são utilizados valores auxiliares dependentes do valor do passo de simulação, *tstep*. Estes são gerados por um módulo dedicado cujas saídas estão ligadas aos integradores.

O cálculo das acelerações é realizado utilizando os dados de posição e massa dos corpos. Os portos de entrada de cada unidade estão ligados a sinais comuns a todas as unidades, como os sinais *data_x*, *data_y*, *data_z* e *data_v*, que transportam os dados de posição e massa dos corpos. Estas unidades de processamentos são geradas utilizando a instrução *generate* do *VHDL*. Isto permite alterar facilmente o número de unidades de processamento alterando a constante *NCORES* presente em *properties.vhd*, secção 6.1. Por outro lado, cada um possui um endereço único, usado como identificação quando são carregados os dados dos corpos para os quais são calculadas as acelerações. Os dados à saída destes módulos são transportados por linhas de deslocamento de dados que os introduzem uma nova aceleração nas memórias correspondentes em cada ciclo de relógio em que estão ativas.

O módulo *nbody_processor* é controlado por uma máquina de estados. Na Figura 5.2 é apresentado um fluxograma simplificado. O estado *data_transfer* é o estado de reiniciação do sistema.

Após a reiniciação do sistema, é necessário realizar o cálculo das acelerações de todos os corpos para se proceder à integração das posições dos mesmos, visto só estarem presentes no sistema dados de posição e velocidade dos corpos. Este cálculo corresponde a um conjunto dos estados *load_core_bodies*, *wait_load_start_sim*, *simulate*, *wait_sim_complete*, *load_data_lines*, *shift_data_lines*, *test_calc_complete* e *end_step* após o estado *prep_coefs*. Esta informação não foi inserida na Figura 5.2 pois trata-se de um caso particular da evolução dos estados e a sua inclusão dificultaria a compreensão do normal funcionamento da máquina de estados.

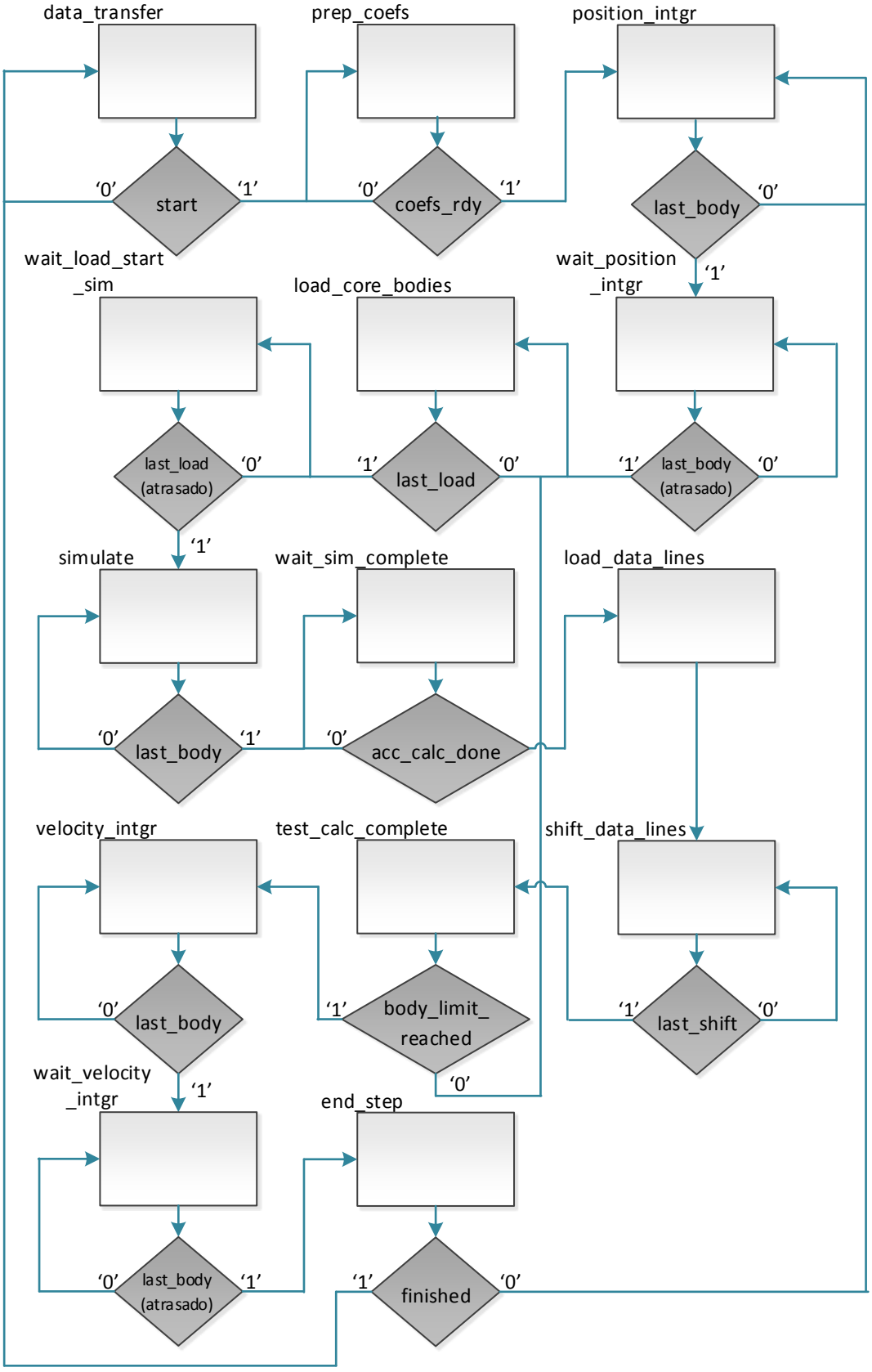


Figura 5.2 - Lógica de mudança de estado do módulo nbody_processor.

O estado *data_transfer* destina-se à transferência de dados entre as memórias e o exterior. Neste estado as memórias são acedidas apenas pelo módulo *UARTcontrol*, como mostrado na Figura 5.3. O sinal *mem_mode* representa o modo de operação das memórias. Apenas neste estado o modo de operação é o de transferência de dados, nos restantes é o modo de simulação. Dada a diferença entre os formatos utilizados, é necessária a introdução de conversores nas linhas de dados. A introdução destes conversores implica outras alterações no sistema, pois estes incluem atrasos nas linhas de dados. Na escrita, é necessário introduzir um número de ciclos de atraso igual ao da conversão na linha de endereço e sinal de escrita, de forma a garantir a correspondência entre o endereço e os dados e acionar o sinal de escrita da memória no momento correto. Na leitura não é necessária a introdução de atraso na linha de endereço. Para determinar o momento de recolha dos dados pedidos, a saída de leitura das memórias presente em *UARTcontrol* é colocada à entrada de um registo de deslocamento com atraso igual à soma do atraso da conversão de formato e do atraso local na leitura da memória. O sinal resultante tem o mesmo atraso que os dados e indica que os mesmos estão disponíveis.

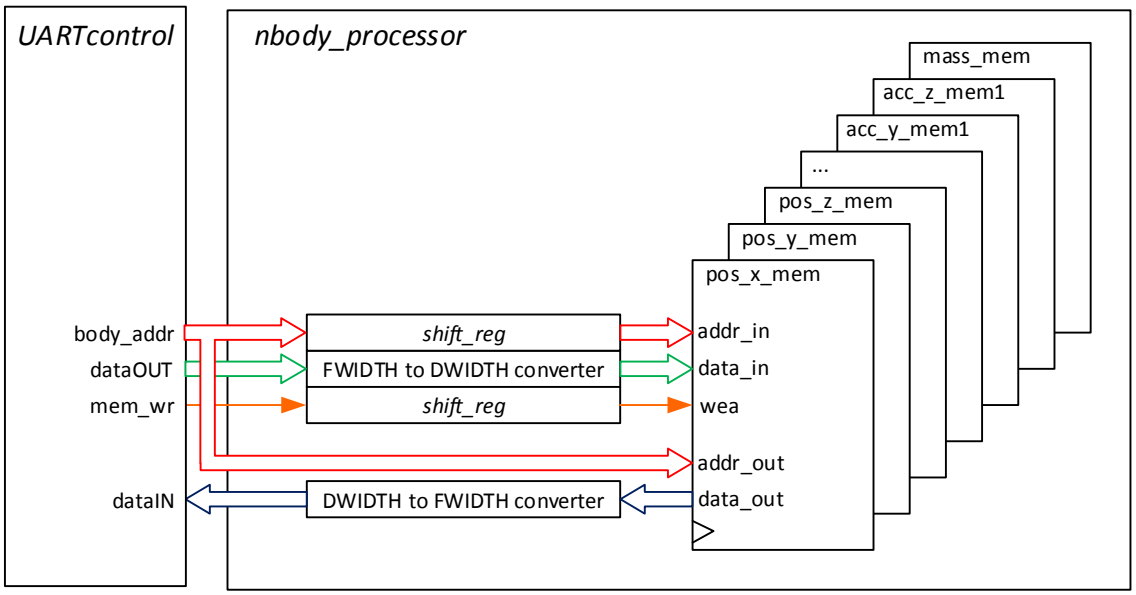


Figura 5.3 - Esquema simplificado do acesso às memórias pelo módulo UARTcontrol.

Na Figura 5.3 não são apresentados os detalhes de multiplexagem das linhas. O sinal de escrita nas memórias não é comum, caso contrário todas as memórias seriam escritas simultaneamente. Recorrendo às saídas de *UARTcontrol* que indicam qual o tipo de dados e coordenada da transferência, e tendo em atenção o valor de *mem_selector*, é possível determinar qual a memória de destino dos dados e ativar apenas o sinal de escrita desta, sendo todos os outros colocados a '0'. Nos sinais de endereço de escrita e sinal de escrita é introduzido um atraso igual ao atraso de conversão dos dados. Na leitura das memórias também é selecionada a saída de dados da memória pretendida. Esta multiplexagem é feita antes do conversor e tendo por

base os sinais de secção de memória atrasados um ciclo, compensando o ciclo de atraso resultante da leitura da memória.

No estado *data_transfer* é reiniciado o contador do número de passos da simulação, *sim_steps*. Quando é recebida a mensagem de início de simulação, a saída respetiva de *UARTcontrol* toma o valor '1', ativando a entrada *start* e provocando a mudança de estado.

O estado seguinte, *prep_coefs*, serve de estado de preparação dos coeficientes de integração numérica. Visto o valor de *tstep* poder ter sido alterado, é necessário recalcular os coeficientes de integração numérica. O sinal de sinalização de início de operação do módulo *intgr_coefs* é colocado a '1'. Quando as operações aritméticas estão completas, percorrendo o valor '1' a linha de atraso interna do módulo, a saída de indicação de operação completa é ativada. O sinal *coefs_rdy* está ligado a esta saída. Uma vez estando o cálculo completo, o estado pode mudar para o seguinte.

Os coeficientes são gerados apenas uma vez por simulação. Por esta razão, e visto *tstep* ser guardado em registo, as saídas deste módulo apresentam valores constantes durante a simulação, não sendo necessário realizar compensação dos atrasos das operações nos módulos que utilizam estes valores.

Os estados *position_intgr* e *wait_position_intgr* são os estados de integração numérica das posições. No estado *position_intgr* as posições, velocidades e acelerações anteriores de todos os corpos são obtidos das memórias e colocados à entrada dos módulos de integração da posição, do tipo *position_verlet_integrator*, como mostrado na Figura 5.4. Ainda durante este estado, se o número de corpos for superior à latência do integrador, as novas posições são escritas em memória.

O endereço dos dados à entrada sofre um atraso igual ao do integrador. Quando os dados da nova posição surgem na saída do integrador, o endereço correspondente está disponível à saída da linha de atraso. O mesmo se passa para o sinal de início de operação, *start_pos_intgr*, e sinal de escrita nas memórias. O sinal *start_pos_intgr* está ativo durante o estado *position_intgr*, percorrendo a linha de atraso interior do integrador de posição. Quando a operação é concluída, o valor do sinal anterior apresenta-se na saída de conclusão de operação, sendo utilizado como sinal de escrita nas memórias. A leitura e escrita paralela é possível visto as memórias possuírem um porto de leitura e um porto de escrita separados e os endereços de leitura e escrita serem diferentes.

O estado *position_intgr* é terminado quando lidas as informações do último corpo presente em memória. O estado seguinte, *wait_position_intgr*, garante que as posições dos corpos não são acedidas nos estados posteriores antes que a nova posição do último corpo seja colocada na memória.

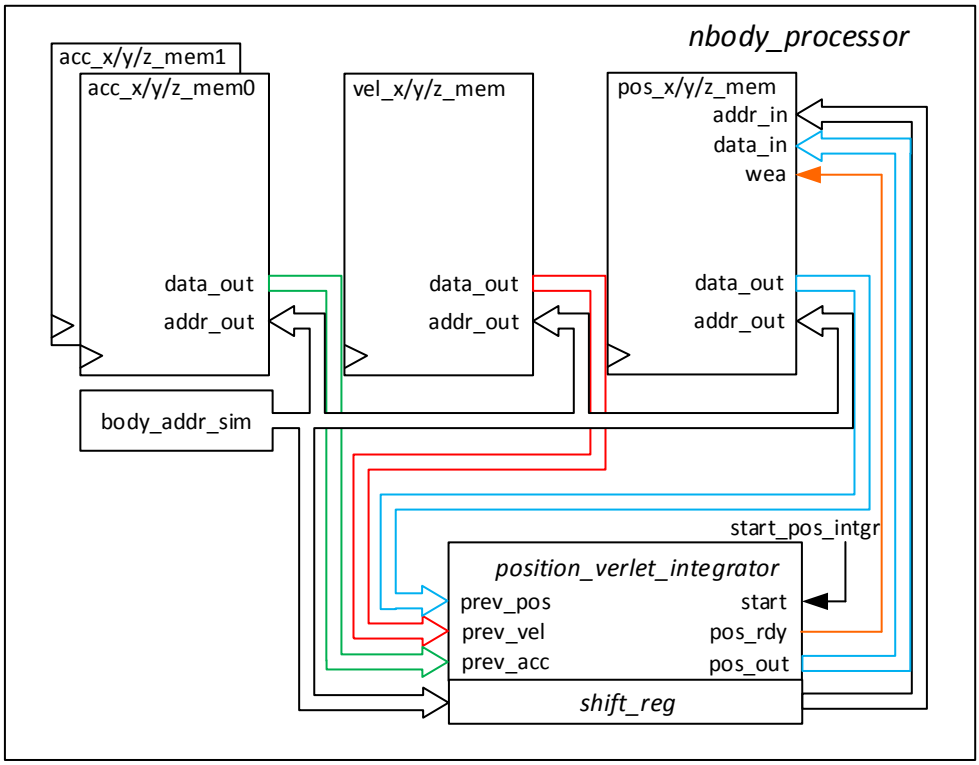


Figura 5.4 - Fluxo de dados na integração numérica das posições.

No estado *load_core_bodies* são carregadas as posições dos corpos para os quais as acelerações devem ser calculadas. Quando o número de corpos é superior ao número de unidades de processamento as acelerações serão calculadas para apenas uma parte do conjunto de corpos do sistema, pois a cada unidade de processamento é atribuído um corpo. No carregamento das posições é utilizado o esquema de ligações apresentado na Figura 5.5.

Os endereços de corpo a carregar e de unidade de processamento são incrementados a cada ciclo de relógio, para que o próximo núcleo de processamento receba os dados de posição do corpo seguinte. Dado o atraso na leitura das memórias, o endereço das unidades de processamento utilizado é igualmente atrasado. É este sinal que é comparado com o valor *NCORES* subtraído de uma unidade, para determinar se o núcleo de processamento atual é o último. O sinal *body_addr_load* é também comparado com o registo *nbodies* do módulo *UARTcontrol*, para determinar se o corpo atual é o último. Caso algumas das condições seja verdadeira o estado muda para *wait_load_start_sim*. À semelhança do estado *wait_position_intgr* na integração das posições dos corpos, este estado garante a consistência dos dados das unidades de processamento. Neste estado são ainda reiniciados os acumuladores internos das unidades de processamento, preparando estas estruturas para o cálculo das acelerações.

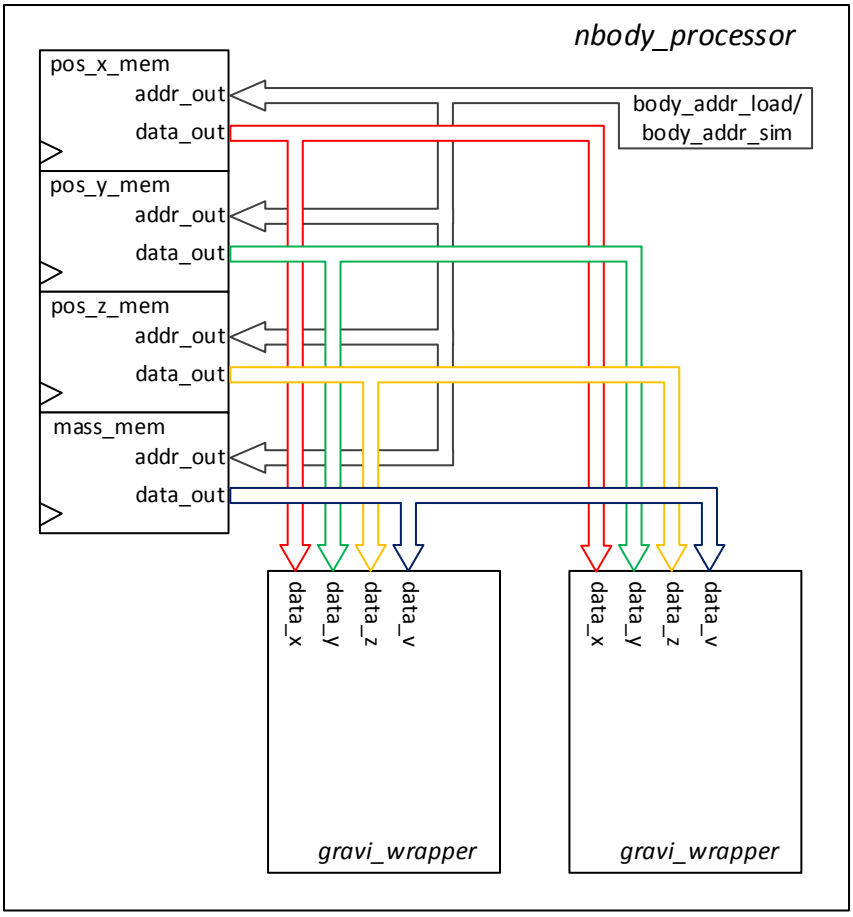


Figura 5.5 - Esquema de ligações do carregamento de posições e massas dos corpos nas unidades de processamento.

No estado *simulate* os dados das posições e massas de todos os corpos são colocados nas entradas *data_x*, *data_y*, *data_z*, e *data_v* das unidades de processamento, segundo o esquema de ligações presente na Figura 5.5. O sinal de iniciação de operação dos módulos *Gravi_core* de cada unidade de processamento é ativado, iniciando o cálculo. Quando são carregados os dados do último corpo a simular, o sinal *last_body* assume o valor '1', provocando a mudança de estado no ciclo de relógio seguinte.

Em seguida, no estado *wait_sim_complete*, o simulador aguarda o fim das operações de cálculo. Visto a entrada de dados ser simultânea em todas as unidades, e consequentemente as operações acabarem ao mesmo tempo, para determinar se o cálculo está completo basta testar a saída de indicação de conclusão de operação do primeiro módulo.

Segue-se o processo de deslocamento dos resultados do cálculo da aceleração para as memórias. O estado *load_data_lines* dura apenas um ciclo de relógio e tem por objetivo o carregamento dos dados resultantes deste cálculo nas linhas de deslocamento para as memórias. Cada uma destas linhas de deslocamento transporta os dados de uma coordenada diferente. O sinal *load_lines_sig*, ligado ao porto *load* das 3 linhas de deslocamento de dados presentes no sistema, é ativado para efetuar a operação de carregamento. O endereço *body_addr_wr*, utilizado para indexar na escrita às memórias de aceleração, é carregado com o endereço

referente ao valor de endereço do corpo carregado na primeira unidade de cálculo, pois a saída das linhas está colocada antes do primeiro estágio das mesmas, e visto este está ligado às saídas da primeira unidade de processamento. A Figura 5.6 exemplifica a estrutura utilizada.

No estado *shift_data_lines* as acelerações calculadas são carregadas uma a uma nas memórias, também segundo o esquema da Figura 5.6. Este estado termina quando o endereço de escrita atinge o valor referente ao último corpo guardado nas unidades de cálculo.

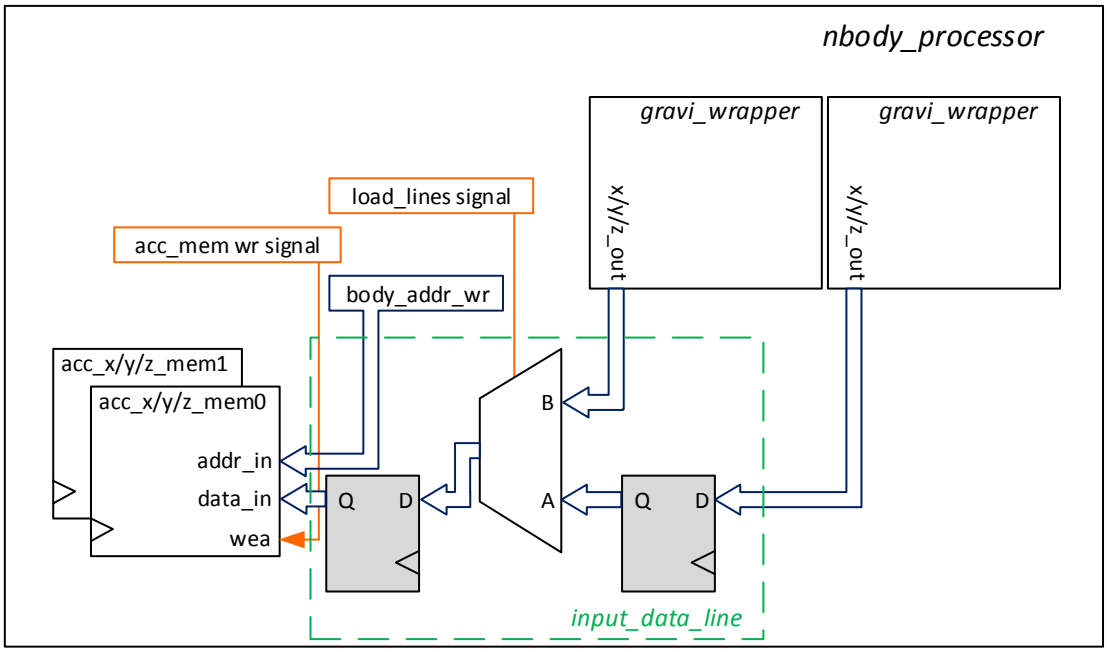


Figura 5.6 - Fluxo de dados das unidades de processamento para as memórias de aceleração.

No estado *test_calc_complete* é verificado que todas as acelerações dos corpos foram calculadas. Caso existam corpos por processar no passo de simulação presente, o estado seguinte será *load_core_bodies*, e serão carregados dados dos restantes corpos, tendo por limite o número de unidades de processamento. Caso contrário, é realizado o cálculo das novas velocidades dos corpos.

Nos estados *velocity_intgr* e *wait_velocity_intgr* o sistema tem um comportamento semelhante aos estados position_*intgr* e *wait_position_intgr*, mas são atualizadas as velocidades dos corpos. Tal como mostrado na Figura 5.7, são utilizadas as acelerações atuais e anteriores, sendo acedidas todas as 6 memórias com dados de aceleração, bem como as memórias que armazenam as velocidades dos corpos.

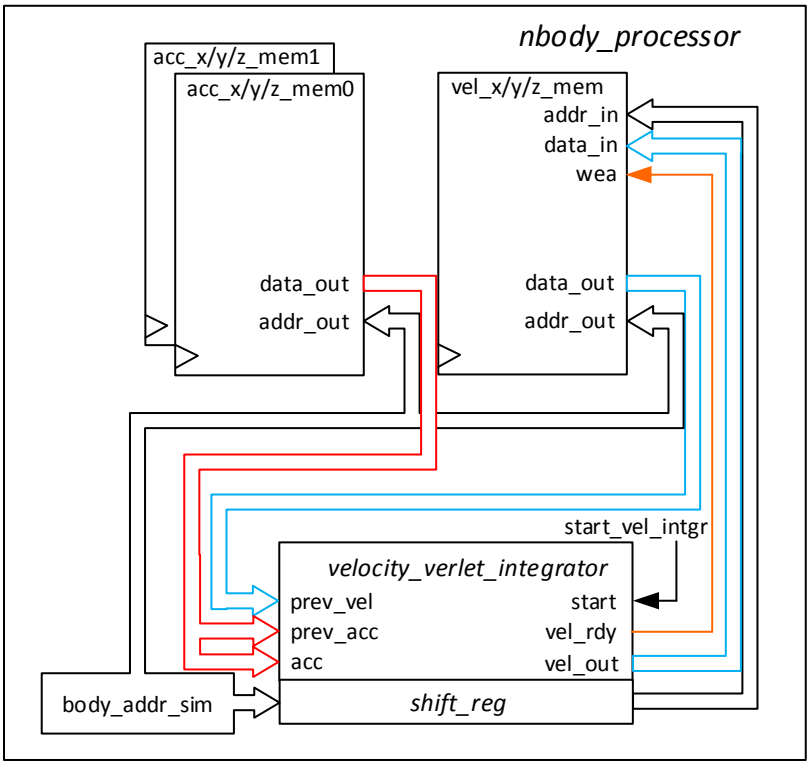


Figura 5.7 - Fluxo de dados na integração numérica das velocidades.

No estado *end_step* é testado se a simulação já cumpriu o número de passos desejados. Se for o caso, o próximo estado será *data_transfer*, para permitir a extração dos dados de simulação pelo utilizador do sistema. Caso contrário segue-se o estado *position_intgr*, dando-se início a um novo passo de simulação. O contador de passos é incrementado, exceto quando este estado se refere ao cálculo das acelerações iniciais dos corpos. É também neste estado que as memórias das acelerações são invertidas. Os endereços referentes ao primeiro corpo carregado numa unidade de processamento são reiniciados.

5.2 Integração numérica

O método de integração numérica *Velocity Verlet*, apresentado em 3.2.2, possui na sua variante simplificada, um passo de cálculo da posição seguinte, $\vec{r}(t+\Delta t)$, e um passo de cálculo da velocidade seguinte, $\vec{v}(t+\Delta t)$.

Foram desenvolvidos dois módulos de cálculo dedicado, um da nova posição, 5.2.1, e um da nova velocidade 5.2.2. Cada um deste módulos foi concebido segundo uma estrutura de *pipeline*, utilizando os módulos aritmético definidos na secção 4.6 e com a inserção de linhas de atraso para compensar a latência das operações aritméticas.

Estes módulos, à semelhança do módulo de cálculo da aceleração, possuem cada um uma entrada de iniciação, cujo valor é propagado ao longo de linhas de atraso. Introduzindo o valor lógico '1' nesta entrada, este valor será apresentado nas saídas *pos_rdy* e *vel_rdy*.

5.2.1 Módulo de cálculo da posição seguinte

O módulo *position_verlet_integrator* implementa a expressão (3.19)

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \frac{1}{2}\vec{a}(t)\Delta t^2,$$

contendo 2 multiplicações e 2 somas, segundo o esquema de operações da Figura 5.8.

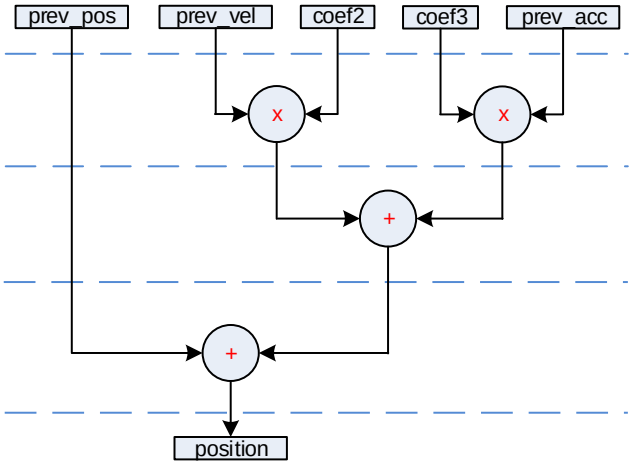


Figura 5.8 - Esquema de operações do módulo position_verlet_integrator.

A utilização de recursos deste módulo é apresentada na Tabela 5.1. O número de *slices* ocupados corresponde a 5% dos disponíveis na *FPGA*.

Resultados após *P&R*	Método de optimização	Slices	Flip Flops	Total LUTs	Mul. 18x18	Máx Freq. [MHz]
position_verlet_integrator 25 bits	Velocidade	836	734	1150	2	188

Tabela 5.1 - Utilização de recursos do módulo position_verlet_integrator.

5.2.2 Módulo de cálculo da velocidade seguinte

O módulo *position_verlet_integrator* implementa a expressão (3.20)

$$\vec{v}(t+\Delta t) = \vec{v}(t) + \frac{1}{2}\big(\vec{a}(t) + \vec{a}(t+\Delta t)\big)\Delta t,$$

contendo um total de 2 multiplicações e uma soma, segundo o esquema de operações apresentado na Figura 5.9.

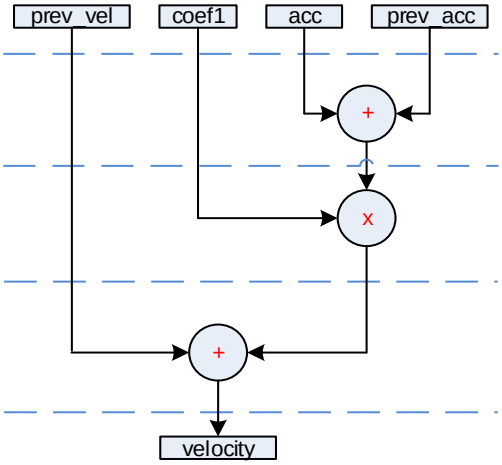


Figura 5.9 - Esquema de operações do módulo velocity_verlet_integrator.

A utilização de recursos deste módulo é a indicada na Tabela 5.2. O número de *slices* utilizados corresponde a 4% dos totais disponíveis na *FPGA*.

Resultados após *P&R*	Método de optimização	Slices	Flip Flops	Total LUTs	Mul. 18x18	Máx Freq. [MHz]
velocity_verlet_integrator 25 bits	Velocidade	766	652	1043	1	184

Tabela 5.2 - Utilização de recursos do módulo velocity_verlet_integrator.

5.3 Unidade de processamento

Este módulo corresponde à unidade de processamento do sistema realizado e é descrito por *gravi_wrapper*. Cada unidade inclui um módulo de cálculo da aceleração gravítica e serve de interface deste módulo. Este módulo realiza o processamento da aceleração de um corpo de cada vez.

Cada uma destas unidades possui 4 linhas de entrada de dados, *data_x*, *data_y*, *data_z* e *data_v*, correspondentes às entradas de dados do módulo *Gravi_core*, ou seja, recebem as coordenadas *x*, *y* e *z* e a massa de todos os corpos envolvidos na simulação. A posição anterior é a utilizada no cálculo da aceleração, pelo que é recebida nas entradas *data_x*, *data_y* e *data_z*.

Num sistema em que várias unidades de processamento estão ligadas a barramentos comuns de dados, é necessário distinguir qual o destino dos dados de posição do corpo para o qual é calculada a aceleração. Deste modo, cada unidade de cálculo é distinguida por um endereço definido por um *generic* do módulo *VHDL*. Uma linha comum com o endereço do módulo alvo dos dados transmitidos é comparada internamente em cada unidade com o seu próprio endereço, de forma a obter um sinal, *cs*, com o nível lógico '1' quando os endereços são iguais.

Caso os sinais *load_data* e *cs* assumam ambos o valor '1', os dados em *data_x*, *data_y* e *data_z* são guardados em *x1_reg*, *y1_reg* e *z1_reg*, respetivamente. Assim a posição do corpo para o qual é calculada a aceleração é guardada. Estes registos estão ligados às entradas *x1*, *y1* e *z1* do módulo *Gravi_core*, respetivamente.

Os recursos utilizados por cada unidade de processamento estão descritos na Tabela 5.3. Estes correspondem à soma dos recursos utilizados pelo módulo de cálculo da aceleração, lógica combinatória e registos deste módulo.

Resultados após *P&R*	Método de optimização	Slices	Flip Flops	Total LUTs	Mul. 18x18	Máx. Freq [MHz]
gravi_wrapper	Velocidade	4627	3969	6164	8	166

Tabela 5.3 - Recursos utilizados pelo módulo Gravi_wrapper.

5.4 Módulo de geração dos coeficientes de integração

O módulo *intgr_coefs* gera os coeficientes necessários à integração numérica. Os coeficientes indicados têm os seguintes valores, retirados das expressões (3.19) e (3.20):

- coeficiente 1 – $tstep/2$,
- coeficiente 2 – $tstep$,
- coeficiente 3 – $tstep^2/2$,

sendo *tstep* o valor do passo de simulação. Visto *tstep* ser representado por um registo do sistema, apenas os coeficientes 1 e 3 são gerados neste módulo, sendo *tstep* uma entrada do mesmo. Os operadores aritméticos são os mesmos que os utilizados em 4.6. Para a geração do coeficiente 1 é realizada a multiplicação de *tstep* pela representação de vírgula flutuante do valor 1/2. Para a geração do terceiro coeficiente, é realizada uma multiplicação do coeficiente 1 por *tstep*.

Visto os operadores aritméticos utilizados possuírem uma latência entre a entrada de operandos e saída dos resultados, este módulo possui uma entrada de indicação de operação, alimentando esta uma linha de atraso de dimensão igual à latência da operação de geração do coeficiente 3, que tem latência igual à soma da latência de duas multiplicações. Uma saída de indicação de operação completa, ligada à saída da linha de atraso, permite determinar quando os coeficientes estão prontos.

A utilização de recursos deste módulo está presente na Tabela 5.4. O número de *slices* gasto corresponde apenas a 0,9 % do número total presente na *FPGA* utilizada.

Resultados após *P&R*	Método de optimização	Slices	Flip Flops	Total LUTs	Mul. 18x18	Máx Freq. [MHz]
intgr_coefs 25 bits	Velocidade	144	171	203	2	272

Tabela 5.4 - Recursos utilizados pelo módulo intgr_coefs.

5.5 Conversores de formato

Os dados de simulação, posições, velocidades, acelerações, massas e dimensão do passo de simulação, são recebidos por porta série. Por outro lado, a representação numérica utilizada neste sistema não pertence ao conjunto de representações usuais em computadores genéricos, a precisão simples ou precisão dupla, definidas em 3.4.2.

Visto isto, é necessário converter o formato das representações numéricas na entrada e saída de dados. São utilizados conversores realizados com *IPCore*s da Xilinx, tais como os utilizados em 4.3. Estão presentes 3 conversores no sistema: um conversor de entrada que converte o valor de *tstep*, um conversor de entrada que converte os dados a escrever nas memórias, e um conversor de saída que converte os dados a ler das memórias.

O sistema realizado foi configurado para uma dimensão de 32 bits no exterior do sistema. A Tabela 5.5 mostra os recursos utilizados por estes módulos. É observável que o consumo de recursos é bastante baixo. Ao conversor de 25 bits para 32 bits foi adicionada uma linha de registos à entrada, para isolar a lógica combinatória à entrada deste módulo do resto do circuito, pois esta lógica provocava uma diminuição da frequência máxima de funcionamento.

Conversor de formato	Slices	Flip Flops	Total LUTs	Ciclos de latência	Máx Freq. [MHz]
float32 to float(8,16)	50	75	73	3	368
float(8,16) to float32	53	78	36	3	487

Tabela 5.5 - Recursos utilizados por conversores entre formato de 25bits e formato de 32 bits.

5.6 Linhas de deslocamento de dados para as memórias

Para conduzir os dados provenientes dos vários núcleos de processamento para as memórias, foram criadas linhas de deslocamentos de dados. Estas linhas comportam-se como conjuntos de registos de deslocamento com opção de carregamento paralelo de dados.

Na Figura 5.10 pode-se observar uma representação desta estrutura. Os dados presentes nos registos são deslocados todos os ciclos de relógio. Quando o sinal *load* está ativo os registos são carregados com os dados presentes no sinal *load_lines*, do tipo *data_vector* definido na secção 6.1.

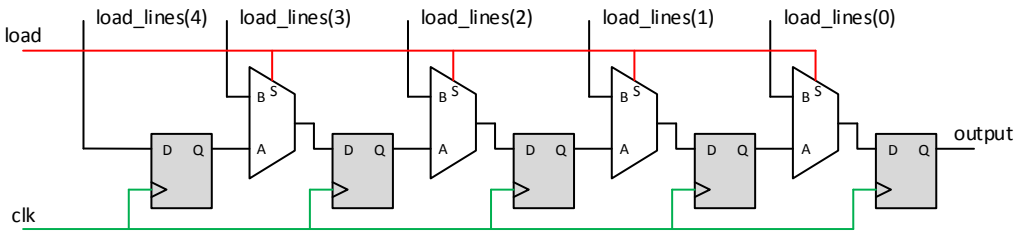


Figura 5.10 - Linha de deslocamento de dados de dimensão 5.

Visto a estrutura permitir o carregamento paralelo de dados, os registos de deslocamento não podem ser realizados com *LUT*, sendo realizados com *flip flops*. Cada multiplexador é realizado com uma *LUT*. Deste modo, o número teórico total de *LUT* gastas é dado por

$$N_{LUTS_{linha}} = (L_{linha} - 1) \times L_{dados}, \tag{5.1}$$

em que L_{linha} corresponde à dimensão da linha de dados e L_{dados} à sua largura. O número teórico de *flip flops* gastos é dado por:

$$N_{flip\ flops_{linha}} = L_{linha} \times L_{dados}. \tag{5.2}$$

Após uma análise de resultados após *Place and Route* para várias dimensões de linha e dimensões da representação numérica dos dados, foi concluído que o número de *slices* utilizados corresponde aproximadamente ao número de *flip flops* total. No entanto o número de *flip flops* cresce em número aproximadamente igual à largura, resultando em:

$$N_{flip\ flops_{linha}} \cong (L_{linha} + 1) \times L_{dados}. \tag{5.3}$$

Por seu lado, o número de *LUT* mantém-se. Por exemplo, para uma dimensão da linha igual a 5, são obtidas 100 *LUT*, 151 *flip flops* e 146 *slices*.

6 Implementação e Resultados

O módulo de nível mais alto hierárquico é *nbody_system*. Neste módulo são apenas realizadas as ligações entre os seus 4 componentes internos: o módulo *UARTcomponent*, que serve de módulo *UART*, o módulo *UARTcontrol*, apresentado na secção 6.2, responsável pela interpretação dos dados recebidos e coordenação do envio de dados, o módulo *nbody_processor*, que incluí toda a lógica responsável pela realização da simulação, e um *DCM*, que permite alterar a frequência de relógio dos outros módulos. O módulo *nbody_system* tem 3 entradas, a entrada de sinal da porta série, sinal de reiniciação do sistema e o sinal de relógio proveniente da placa de desenvolvimento, e uma saída, a saída de dados da porta série.

6.1 Ficheiro de propriedades do sistema

O ficheiro *properties.vhd* serve de ficheiro de configuração do sistema. Neste ficheiro é definido o pacote *properties*, sendo este importado pela maioria dos módulos que fazem parte do sistema desenvolvido neste trabalho de projeto. Este pacote inclui três tipos de dados importantes para o sistema: constantes de configuração, tipos utilizados em vários módulos e como ligação entre módulos, e descrições de componentes base do sistema, como os operadores aritméticos e linhas de atraso.

As constantes incluídas neste ficheiro permitem configurar o sistema. Estão incluídas as seguintes constantes:

- número de bits do expoente da representação numérica, *EXPONENT_BITS*;
- número de bits da mantissa da representação numérica, *MANTISSA_BITS*;
- dimensão total da representação numérica, obtida a partir das duas constantes anteriores, *DWIDTH*;
- número máximo de corpos a simular, *MAX_NBODIES*, parâmetro importante para definir a dimensão das memórias utilizadas;
- logaritmo de base 2 do número máximo de corpos, *LOG2_NBODIES*, que permite definir a dimensão dos sinais de endereço das memórias (arredondado para o inteiro mais próximo igual ou superior);
- número de núcleos de processamento, *NCORES*, que define o número de unidades de processamento;
- número de ciclos de latência das várias operações aritméticas, que incluem soma, subtração, multiplicação, divisão, raiz quadrada e soma realizada no acumulador, permitindo estes valores definir as dimensões das linhas de atraso utilizadas no sistema;

- dimensão e número de *bytes* dos valores de recebidos pela porta série, *VAR_LENGTH* e *VAR_BYTES*;
- dimensão dos dados de vírgula flutuante do sistema recebidas por porta série, *FWIDTH*;
- número de ciclos de latência dos conversores de formato entre os valores de vírgula flutuante recebidos pela porta série e o formato de dados usado internamente no sistema;
- representação numérica da constante de gravitação universal *G*;
- representação numérica do valor 1/2, valor usado na geração de coeficientes de integração;
- valor da frequência do sinal de relógio utilizado no sistema, *SYSTEM_CLK*;
- valor do débito símbolo da porta série, *BAUD_RATE*;
- valor de divisão do relógio do sistema para obter o relógio do módulo *UART*, *UART_BAUD_DIVIDE*.

Em seguida são definidas duas enumerações utilizadas no sistema, *axis* e *dataType*. O tipo *axis* possui os valores *x*, *y* e *z*, e destina-se a ser utilizado para distinguir a que coordenada espacial se referem alguns dados do sistema. O tipo *dataType* tem o mesmo papel, mas distingue o tipo de dados: *position*, *velocity*, *acceleration*, *mass* e *none*. São também definidos dois tipos, *data_vector* e *addr_vector*, *arrays* de dados e de endereços, respetivamente, que são utilizados para realizar as ligações das memórias.

Finalmente estão definidas as descrições de entradas, saídas e parâmetros de configuração de vários componentes base do sistema, como o componente utilizado para implementar as linhas de atraso, os operadores aritméticos, o componente utilizado como memória e os conversores de formato. Os componentes que usam *IPCore*s têm cada um ficheiro *vhd* que serve de camada de abstração entre o nome do operador no sistema e o nome utilizado no *IPCore*, pois estes módulos incluem os *IPCore*s no seu interior. As descrições presentes em *properties.vhd* referem-se aos ficheiros *vhd*. Desta forma o *IPCore* utilizado em todo o sistema pode ser alterado trocando o componente referido no ficheiro *vhd*.

Os valores de configuração são definidos como constantes presentes num pacote de forma a obter um sistema reconfigurável mas com uma síntese otimizada para cada configuração. No entanto, isto implica a síntese, tradução, mapeamento, posicionamento e roteamento e geração da programação da *FPGA* de cada vez que uma das constantes do sistema é alterada. Os *IPCore*s incluídos no sistema também têm de ser gerados de acordo com as especificações deste ficheiro, em particular as características das representações numéricas.

6.2 Módulo de controlo do *UART*

Este módulo, *UARTcontrol*, realiza a interface entre o sistema de processamento e o *Universal Asynchronous Receiver/Transmitter* (*UART*), módulo controlador do meio de comunicação utilizado, a porta série presente no kit ML402. O componente de controlo da porta série utilizado, *UARTcomponent*, foi desenvolvido no âmbito da unidade curricular de Projecto de Sistemas Digitais, pelo docente da mesma, orientador deste trabalho final. O débito símbolo da porta série utilizado foi de 38400 baud.

O controlador da porta série apresenta uma entrada de dados de 8 bits, referida neste módulo como *uartIN*, uma saída de dados de 8 bits, referida neste módulo como *uartOUT*, e sinais de controlo. Estes sinais incluem o sinal *RDA* (*Read Data Available*) que indica quando estão disponíveis novos dados à saída do *UART*, o sinal *TBE* (*Transfer Bus Empty*), que indica quando podem ser colocados novos dados no registo de envio, o sinal *RD* que efetua a leitura de dados, e o sinal *WR* que efetua a escrita de dados.

Foram incluídos em *UARTControl* dois módulos de conversão entre sequência de bits e palavra binária a receber ou transmitir. Estes módulos são o *SIPO*, *Serial Input Parallel Output*, responsável pela receção dos valores com a dimensão *VAR_LENGTH* e número de *bytes VAR_BYTES*, dimensões definidas em *properties.vhd* e referidas na secção 6.1, e o *PISO*, *Parallel Input Serial Output*, responsável pela transmissão para o exterior dos valores com as mesmas dimensões. Estes componentes foram concebidos com o objetivo do modularizar e simplificar a receção e transmissão de dados, comunicando diretamente com o *UART*, sua saída e entrada de dados e sinais de controlo.

6.2.1 Módulo de entrada em série e saída paralela - *SIPO*

Este módulo procede à leitura automática de um número pré-definido de *bytes* do *UART*. Na Figura 6.1 é apresentada a evolução da máquina de estados do mesmo. Após a deteção de um valor lógico ‘1’ na entrada de início de operação durante o estado *idle*, o módulo aguarda a receção de dados pelo *UART* no estado *wait_data*. Quando são disponibilizados dados à saída do *UART*, o estado muda para *read_data*, sendo lidos os mesmos dados e colocados na primeira posição do registo interno de deslocamento, deslocando todos os outros para a esquerda. O índice referente aos dados é também decrementado, pois os *bytes* são recebidos começando pelo de maior peso. No estado seguinte, *comp_index*, é testado se já foram recebidos todos os dados, em função do valor de índice do *byte*. Se sim, o próximo estado é *data_output*, e é sinalizada a conclusão da receção de dados. Se não, segue-se o estado *wait_data* para aguardar a chegada de um novo *byte*.

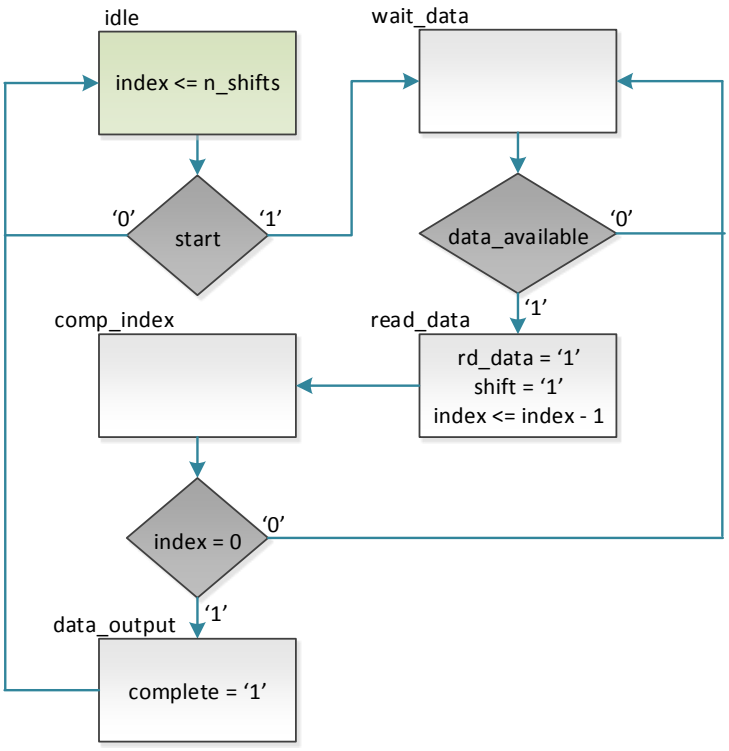


Figura 6.1 - Fluxograma da máquina de estados do módulo SIPO.

6.2.2 Módulo de entrada paralela e saída em série - *PISO*

Este módulo procede à transmissão automática pelo *UART* de um valor com número pré-definido de *bytes*. Na Figura 6.2 é apresentada a evolução da máquina de estados deste componente. Após a deteção de um valor lógico '1' na entrada de início de operação no estado *idle*, o módulo carrega o valor na entrada paralela no registo de deslocamento interno. A escrita do *byte* mais à esquerda do registo interno é efetuada no estado *write_data* quando o registo de transmissão do módulo *UART* mostra-se disponível. No estado *wait_write* é mantido o sinal de escrita enquanto o envio da informação é realizado. Isto é necessário para garantir o funcionamento do módulo *UART* usado. Segue-se o estado *dec*_index, no qual o índice referente aos dados é decrementado e os dados do registo interno são deslocados para a esquerda. No estado seguinte, *comp_index*, é testado se já foram enviados todos os dados, em função do valor de índice do *byte*. Se sim, o próximo estado é *sending_complete*, e é sinalizada a conclusão da receção de dados. Se não, segue-se o estado *write_data* para enviar o *byte* seguinte.

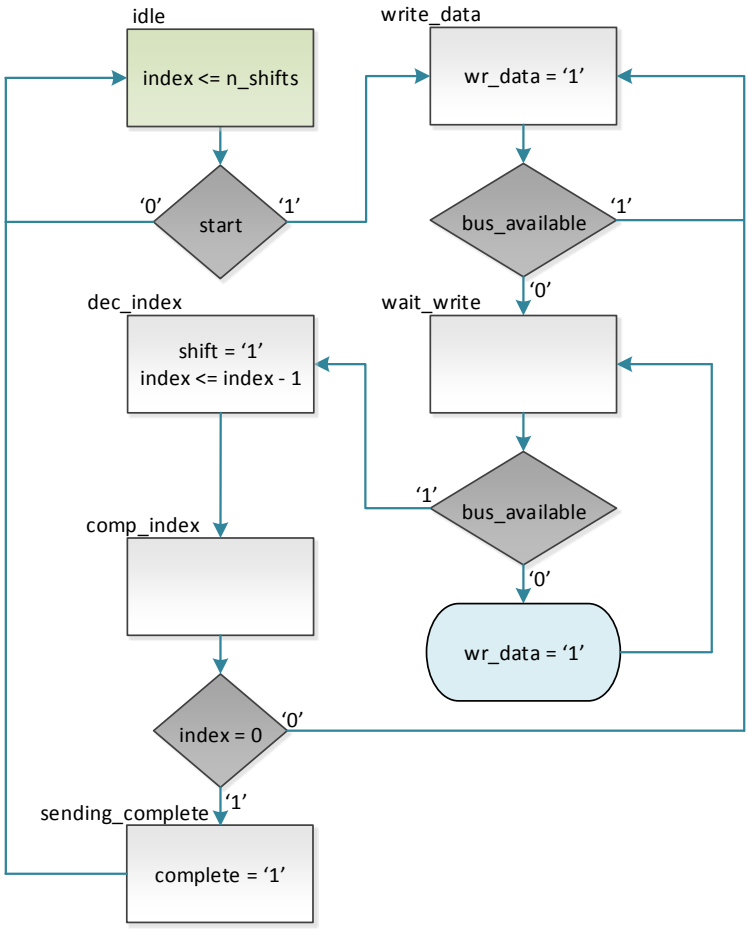


Figura 6.2 - Fluxograma da máquina de estados do módulo PISO.

6.2.3 Protocolo de comunicação e máquina de estados

Foi definido um pequeno protocolo de comunicação entre o sistema de processamento desenvolvido e o exterior. Cada comando indica o tipo de dados que será transmitido ou recebido nos próximos *bytes* ou serve de ordem de início de simulação ou indicação de fim de simulação. Desta forma é possível aceder à memória interna do sistema e alterar ou obter os dados da mesma. Para dar início a uma simulação após a reiniciação do sistema é necessário definir os seguintes dados:

- posições de todos os corpos;
- velocidades de todos os corpos;
- massas de todos os corpos;
- valor do intervalo temporal entre passos da simulação;
- número de corpos total;
- número de passos da simulação.

Se for pretendido obter dados de simulação periodicamente, por exemplo a cada 1000 iterações, podem ser realizadas várias simulações de 1000 iterações em série. Os dados necessários para o arranque da simulação são introduzidos apenas na preparação da primeira, visto nas seguintes todos os dados necessários estarem já no interior do sistema. No final de cada simulação podem ser lidos os dados pretendidos e dado início a uma nova simulação.

Este módulo é controlado por uma máquina de estados. Da Figura 6.3 à Figura 6.10 pode-se observar os fluxogramas desta máquina de estados. Não é apresentado apenas um fluxograma de forma a simplificar a leitura dos esquemas e dado o número de sinais envolvidos.

O estado *idle* é o estado de espera do módulo, Figura 6.3. Neste estado o módulo aguarda o fim da simulação ou a receção de um comando do exterior. Caso a simulação termine, estando a entrada *finished* ativa, o próximo estado será *end_sim*. Caso contrário a evolução do estado será determinada pela receção de um *byte* pela porta série, facto indicado pelo valor de *RDA*. Todos os estados, após terminadas as operações correspondentes e caso a simulação não tenha terminado, são sucedidos por este estado, de forma a simplificar a estrutura da máquina de estados.

Se existirem dados disponíveis à saída do *UART*, é testado se estes dados correspondem a um comando ou um carácter. Esta distinção é realizada tendo em conta o valor do bit de maior peso do *byte* recebido. Visto os caracteres da tabela *ASCII* (*American Standard Code for Information Interchange*) estarem limitados ao valor 127, 0x7F em hexadecimal, o bit de maior peso assume sempre o valor '0' para os caracteres *ASCII*. Esta propriedade foi aproveitada para permitir a receção de caracteres e comandos, estabelecendo-se que os comandos são caracterizados por possuírem o bit de maior peso com o valor '1'. Caso seja recebido um carácter, o estado seguinte será *char_receive*. Caso contrário, o estado é escolhido em função do protocolo definido na Tabela 6.1.

Comando	uartOUT								Estado correspondente
	cmd	estado				tipo de dados			
Início de simulação	1	1	0	0	0	-	-	-	*start_sim*
Número de corpos	1	0	1	0	0	1	0	0	*var_receive*
Número de passos	1	0	1	0	0	0	1	0	*var_receive*
Valor de *tstep*	1	0	1	0	0	0	0	1	*var_receive*
Escrita de posições	1	0	0	1	0	1	0	0	*data_receive*
Leitura de posições	1	0	0	1	1	1	0	0	*data_send*
Escrita de velocidades	1	0	0	1	0	0	1	0	*data_receive*
Leitura de velocidades	1	0	0	1	1	0	1	0	*data_send*
Leitura de acelerações	1	0	0	1	1	0	0	1	*data_send*
Escrita de massas	1	0	0	1	0	0	0	0	*data_receive*

Tabela 6.1 - Formato dos vários comandos e estados correspondentes.

Na Tabela 6.1 não estão presentes um estado de escrita de acelerações ou leitura de massas. Isto deve-se ao facto de as acelerações serem calculadas pelo sistema e as massas não serem alteradas na simulação, não sendo necessário lê-las. A codificação dos comandos foi realizada tenho em conta a otimização da lógica do sistema. Ao tipo do estado seguinte corresponde um conjunto de apenas 3 bits, em que apenas um bit está ativo para cada tipo de estado, sendo que os estados *data_send* e *data_receive* são distinguidos por um bit extra. O tipo de variável interna ou dados a receber/enviar é distinguido por um conjunto de 3 bits, em que também apenas um se encontra ativo para cada tipo.

De forma a preparar o módulo *UART* para a receção de novos dados, é necessário ler os dados à saída do mesmo. Caso o *byte* recebido se trate de um comando, o sinal *rd_command* é ativado. Visto o sinal *RD* corresponder ao ou lógico dos sinal *rd_command*, *rd_echo*, e *rd_shift*, isto provoca a leitura dos dados. De forma semelhante, também o sinal *WR* corresponde ao ou lógico dos sinais *wr_aux* e *wr_shift*.

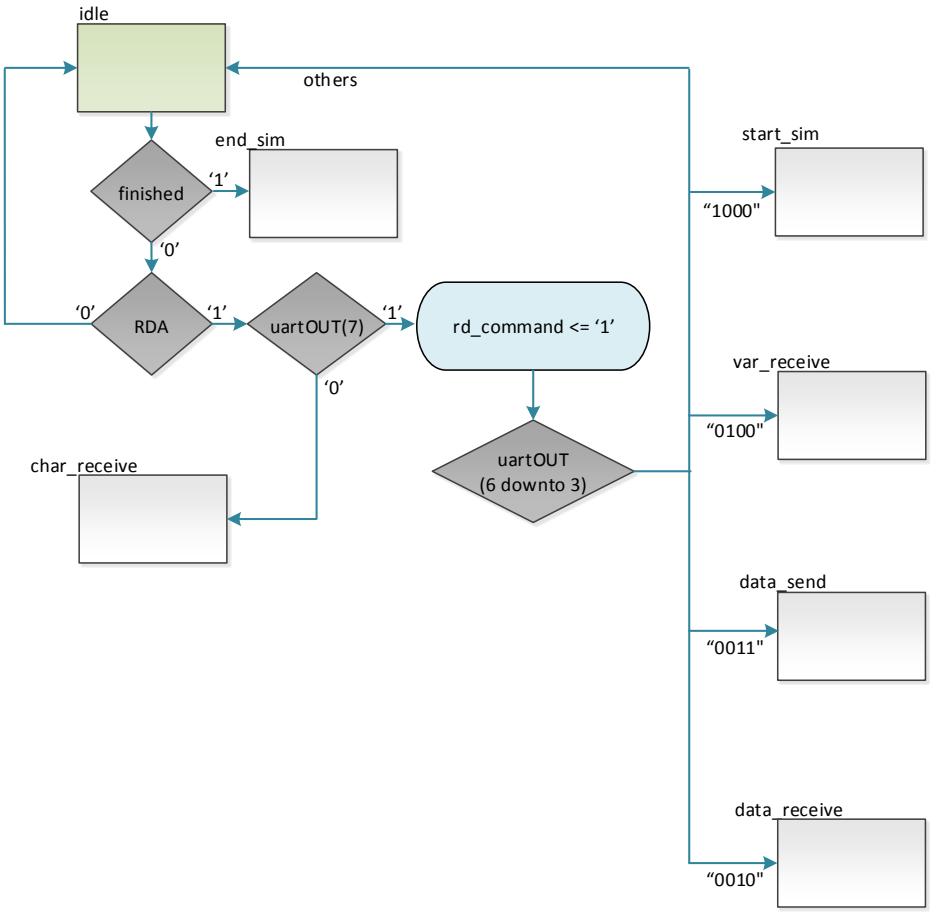


Figura 6.3 - Fluxograma do estado idle.

O estado *char_receive*, Figura 6.4, foi desenvolvido de forma a permitir o eco de caracteres recebidos pela porta série, servindo de forma de teste da funcionalidade da mesma. Para tal, é necessário escrever na entrada do módulo *UART* a informação presente na sua saída, mas tal só

pode ser realizado quando o registo de envio está disponível, ou seja, quando *TBE* está ativo. Se for esse o caso, é efetuada a leitura e escrita do *byte* recebido. É efetuado um redireccionamento dos dados de *uartOUT* para *uartIN*.

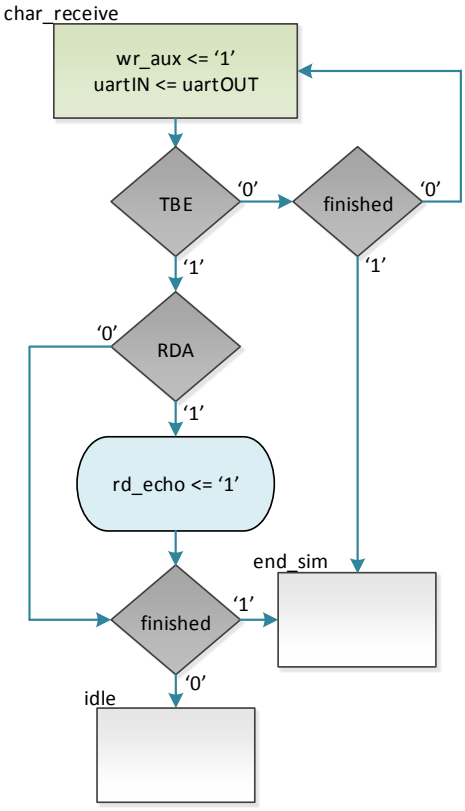


Figura 6.4 - Fluxograma do estado char_receive.

O estado *start_sim* destina-se a iniciar o processo de simulação. Para tal, o sinal *start* é colocado a '1'. Este estado tem a duração de apenas um ciclo de relógio.

O estado *var_receive* permite a receção dos valores inteiros de controlo e do valor de *tstep*. Todos estes valores têm dimensão igual a *VAR_LENGTH*, estando este valor definido no pacote *properties*, secção 6.1. Neste estado o sinal *data_changed* assume o valor '1'. Este sinal indica que a configuração ou dados da simulação foram alterados.

Na receção é utilizado o módulo *SIPO*. Enquanto a recolha dos *bytes* do valor a receber não estiver concluída, o sinal *start_shift_in*, ligado à entrada *start* do módulo *SIPO*, é colocado com o valor lógico '1'. Isto é uma otimização da iniciação do módulo *SIPO*. Visto este módulo só estar sensível à sua entrada de iniciação durante o estado inicial, o sinal *start_shift_in* pode permanecer ativo enquanto a operação de receção de dados decorre neste estado. Sendo assim, o valor deste sinal apenas depende do estado e do sinal *shift_in_done*. Caso este sinal apenas estivesse ativo no primeiro ciclo deste estado, seria necessária mais lógica para definir o valor deste sinal.

Quando a operação de deslocamento de dados é concluída, o sinal *shift_done* toma o valor '1', pois resulta do ou lógico dos sinais *shift_in_done* e *shift_out_done*, e o sinal *load_var* é ativado. Este último sinal comanda o carregamento dos dados recebidos nos registos correspondentes a *nbodies*, *num_steps* ou *tstep*. O sinal *variable_cur* indica qual o registo a afetar. O valor deste sinal é definido conforme a descodificação realizada segundo a Tabela 6.1 no momento de receção do comando.

O comportamento descrito para este estado é observável na Figura 6.5.

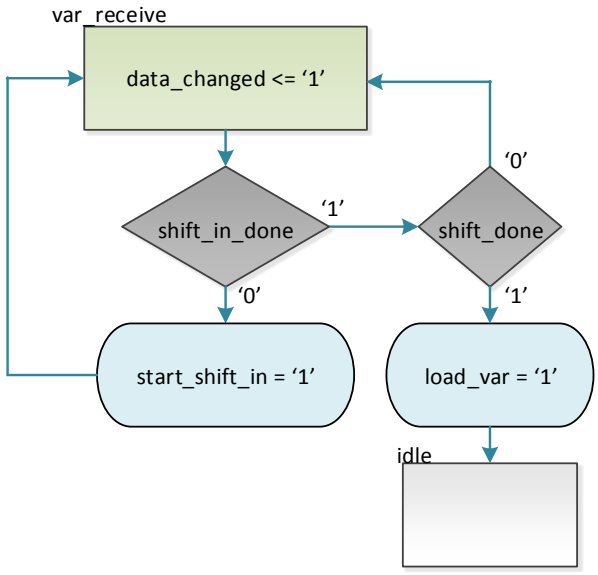


Figura 6.5 - Fluxograma do estado var_receive.

O estado *data_receive* destina-se à receção de sequências de dados, ou seja, as posições, velocidades e massas dos corpos. Um fluxograma com as saídas deste estado pode ser observado na Figura 6.6 e um fluxograma com a evolução de estado pode ser observado na Figura 6.7. O comportamento do sistema neste estado foi dividido em duas figuras de forma a simplificar a compreensão do mesmo.

Tal como no estado *var_receive*, o sinal *data_changed* assume o valor '1' e a transferência de dados é realizada utilizando o módulo *SIPO*, sendo o sinal de iniciação de operação deste módulo tratado da mesma forma.

Os dados são recebidos partindo do corpo de menor índice. No caso das massas são recebidos os valores das massas de todos os corpos, aumentando o índice do corpo conforme estes são recebidos. Quando este índice atinge o valor de *nbodies_dec_reg*, resultante da subtração de uma unidade a *nbodies*, o corpo atual é o último e o índice é reiniciado. No caso das posições e velocidades, são realizadas 3 sequências de transmissão, a primeira com os dados correspondentes à coordenada *x*, a segunda com os dados da coordenada *y* e a terceira com os dados da coordenada *z*. O sinal *last_data_cycle* indica se a sequência atual de dados é a última.

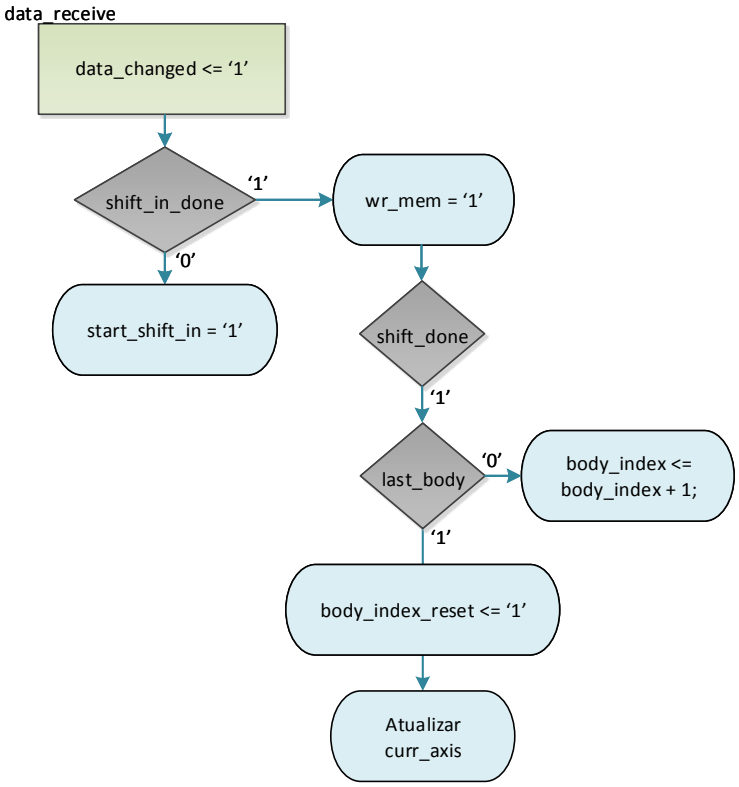


Figura 6.6 - Fluxograma do estado data_receive (apenas saídas).

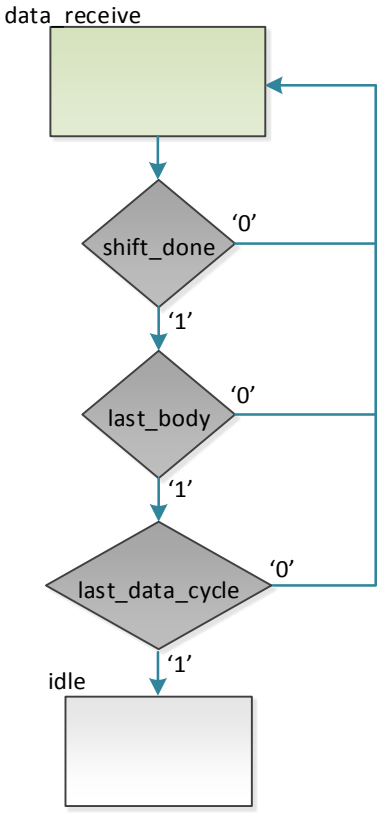


Figura 6.7 - Fluxograma do estado data_receive (lógica de mudança de estado).

O módulo *UARTcontrol* possui duas saídas que indicam o tipo de dados e a coordenada correspondente dos dados a transferir entre este e as memórias, *data_type* e *ax*. A estes portos estão ligados aos sinais que indicam o tipo de dados e coordenada respetivamente. Utilizando estes sinais é possível determinar qual a memória a que correspondem os dados transferidos.

O estado *data_send* permite o envio dos dados presentes nas memórias do sistema para o exterior. A Figura 6.8 mostra parte do comportamento do sistema neste estado, no que diz respeito às saídas do mesmo, pois foram utilizados estados auxiliares para simplificar a definição do comportamento deste estado. Não é apresentada uma figura com a evolução de estado visto esta ser igual à do estado *data_receive*, presente na Figura 6.7. Os recursos de mudança de estado são partilhados com os do estado *data_receive*.

A Figura 6.8 apresenta uma sequência de ações semelhantes às da Figura 6.6, pois o código de atualização das saídas apresentadas em seguida é partilhado, à exceção da atualização de *uartIN*.

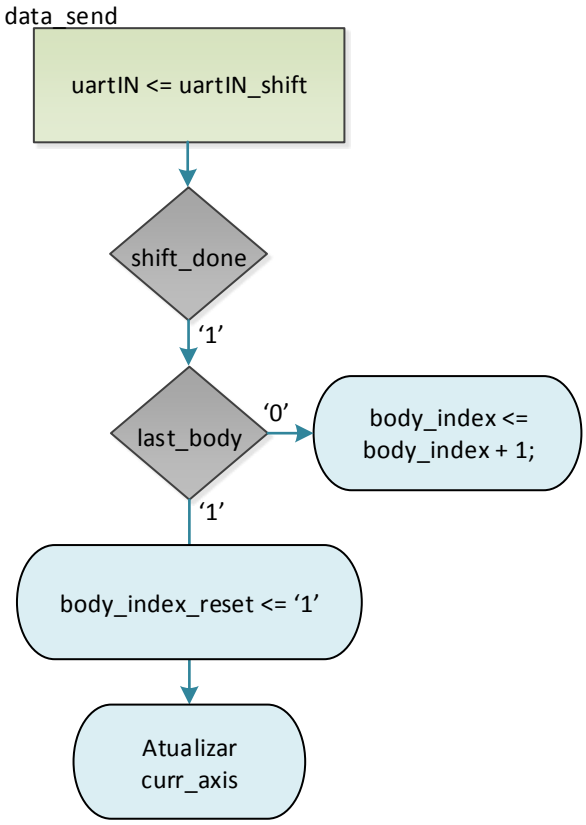


Figura 6.8 - Fluxograma do estado data_send (atualização do índice do corpo e coordenada).

A lógica dos estados auxiliares do estado *data_send* é apresentada na Figura 6.9. O subestado *ds_idle* serve de estado de espera pelo comando de transmissão de dados para o exterior. Quando o estado muda para *data_send*, o subestado passa a ser *mem_rd_state*, estado no qual é iniciada a leitura do valor pretendido das memórias do sistema. Segue-se outro estado de espera, *wait_transfer*, que aguarda que a transmissão dos dados esteja completa, evento assinalado por *shift_out_done* tomar o valor '1'. Este comportamento assume uma forma simples pois o módulo *PISO* está encarregue de realizar a transferência dos *bytes* para o *UART*, e visto o sinal *mem_data_rdy*, que indica quando os dados lidos das memórias estão disponíveis, estar ligado diretamente à entrada de início de operação do módulo *PISO*.

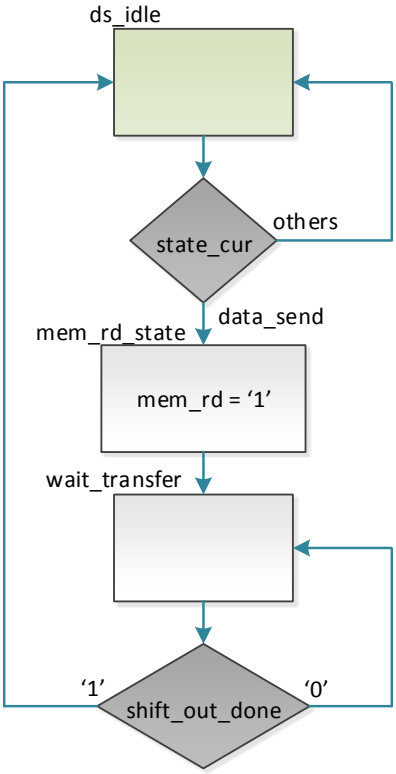


Figura 6.9 - Fluxograma dos estados auxiliares do estado data_send.

O estado *end_sim* tem por objetivo o envio de uma mensagem de sinalização de fim de simulação ao sistema exterior. No entanto, esta mensagem só pode ser enviada se o registo de entrada de dados do *UART* estiver disponível, ou seja, o sinal *TBE* ativo. Enquanto este sinal permanece inativo o estado não é alterado. Quando o sinal *TBE* é ativado, é realizada a escrita do valor "11111111" na entrada no *UART*, sendo este o valor enviado pela porta série. Neste caso o estado seguinte é o estado *idle*. O comportamento descrito é apresentado na Figura 6.10.

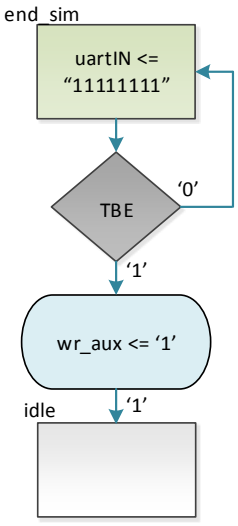


Figura 6.10 - Fluxograma do estado end_sim.

Na Tabela 6.2 são apresentados os recursos utilizados pelos módulos *UARTcomponent* e *UARTcontrol* após *Place and Route*.

Módulo	Método de optimização	Slices	Flip Flops	Total LUTs	Máx Freq. [MHz]
UART	Velocidade	55	60	63	350
UARTControl	Velocidade	261	255	197	258

Tabela 6.2 - Recursos utilizados pelo módulo UARTcomponent e pelo módulo UARTcontrol.

6.3 Resultados

O resultado de área final do sistema implementado é apresentado na Tabela 6.3. Este resultado foi obtido com um conjunto de 2 unidades de processamento *gravi_wrapper*. Foi adicionado um *DCM* de forma a transformar a frequência de relógio de 100 MHz disponível na ML402 na frequência de 150 MHz utilizada. O número de corpos máximo suportado pelo sistema foi escolhido foi 4095. O número de *slices* utilizados corresponde a 82% dos disponíveis na *FPGA* XC4VSX35.

Resultados após *P&R para* nbody_system	Recursos utilizados	Percentagem de utilização
Slices	12619	82%
Flip Flops	13018	42%
Total LUTs	20579	66%
Mul. 18x18	27	14%
BRAMs	91	47%
DCMs	1	12%
Máx. Freq [MHz]	150,670	

Tabela 6.3 - Recursos utilizados pelo sistema e frequência de funcionamento.

Tendo por base os resultados apresentados na Tabela 5.3 e na Tabela 6.3, é possível estabelecer uma relação entre a área total disponível e a capacidade de cálculo do sistema. Visto cada unidade *gravi_wrapper* consumir 4627 *slices* e o sistema realizado com duas destas unidades utilizar 12823 *slices*, é possível afirmar que o número de *slices* utilizados pelo resto do sistema, que não varia significativamente com o número de unidades de processamento, é cerca de:

$$12619 - 2 \times 4627 = 3365\ slices.$$

Dada uma *FPGA* com tecnologia semelhante à utilizada, e definindo N_{slices} como o número total de *slices* disponíveis na mesma, obtém-se o número de unidades de processamento, N_{cores}:

$$N_{cores} = floor\left(\frac{N_{slices} \times 0{,}85 - 3365}{4627}\right), \qquad (6.1)$$

em que *floor* é uma função de arredondamento para o número inteiro mais próximo inferior.

O número de *XtremeDSP slice* presentes na *FPGA* também é um fator limitativo no número de unidades de processamento realizáveis. Visto cada unidade *gravi_wrapper* possuir 8 multiplicadores, 11 dos 27 utilizados pelo sistema pertencem a outros módulos. Outro limite do número de unidade de processamento pode ser obtido por:

$$N_{cores} = floor\left(\frac{N_{XtremeDSP} - 11}{8}\right). \qquad (6.2)$$

O número final de unidades de processamento será igual ao menor destes dois limites.

Dada a arquitetura do sistema, tendo em conta o número de operações aritméticas da integração de posição, integração de velocidade e cálculo da aceleração, e a frequência de funcionamento do sistema, obtêm-se também as seguintes taxas máximas de cálculo:

- integração de posição: 1 *GFLOPS*;
- integração de velocidade: 0,75 *GFLOPS*;
- cálculo da aceleração: 5,4 *GFLOPS*.

Foi ainda realizado um estudo do tempo de simulação no sistema dedicado em função do número de corpos para um total de 100 iterações. Os resultados foram comparados com os obtidos para a implementação em C no capítulo 3.5. Na Figura 6.11 é apresentado o resultado desta comparação. Os tempos de simulação atingidos com o sistema desenvolvido são inferiores aos anteriormente obtidos.

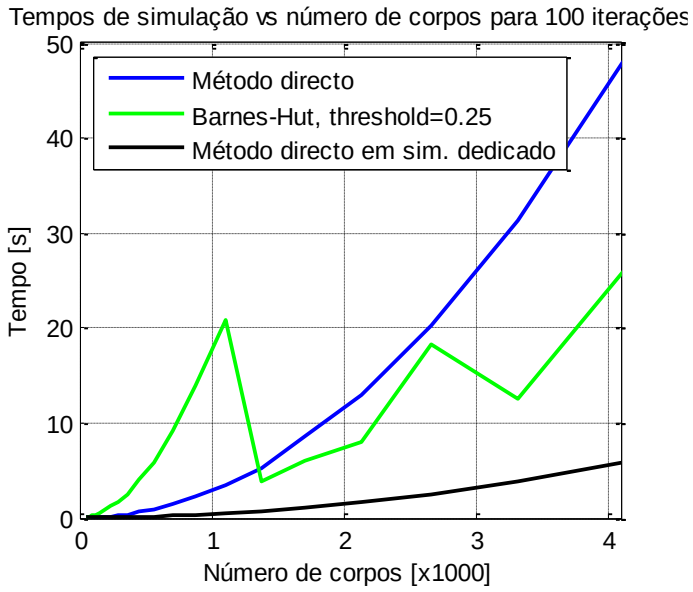


Figura 6.11 - Comparação entre os tempos de simulação do método direto e método de Barnes e Hut em linguagem C e tempo de simulação com o sistema de hardware desenvolvido.

Com estes novos resultados pode ainda ser obtida uma estimativa da taxa média de cálculo do sistema dividindo o número total de operações realizadas pelo tempo total de simulação. Para o número máximo de corpos, 4095, 100 iterações e números de operações aritméticas do cálculo da aceleração e integração numérica, e incluindo o cálculo das acelerações iniciais, o número total de operações é

$$101 \times 4095^2 \times 18 + 100 \times 4095 \times (4 + 3) \cong 3{,}05 \times 10^{10}$$

operações. Dividindo pelo tempo de simulação correspondente, 5,746 segundos, a taxa de cálculo média toma o valor de aproximadamente 5,3 *GFLOPS*, que se aproxima bastante da taxa máxima, demonstrando a eficiência temporal do sistema. Porém, este valor é dependente do número de corpos do sistema, tomando valores inferiores para sistemas com menor número de

corpos, devido ao facto de os ciclos adicionais das máquinas de estados e da realimentação dos acumuladores terem um maior peso para números inferiores de corpos. Por exemplo, para cerca de 700 corpos o valor obtido é de apenas 1,31 *GFLOPS*.

Numa *FPGA* da mesma família mas com área superior, a taxa de cálculo máxima da aceleração depende do número de unidades de processamento, e é dada por

$$Taxa_{cálculo} = N_{cores} \times 18 \times 150M, \qquad (6.3)$$

crescendo linearmente com o número de unidades de processamento. Por exemplo, para uma *FPGA* XC4VFX140, com mais de 63 mil *slices* genéricos e 192 *XtremeDSP slices* [29], seria possível implementar um total de 10 unidades de cálculo, resultando numa taxa máxima de cálculo de 27 *GFLOPS*.

Obtendo os resultados da implementação do sistema numa *FPGA* da família Virtex-7, pode ser realizada uma estimativa das taxas de cálculo possíveis em *FPGA* mais recentes. Utilizando as mesmas latências para os operadores aritméticos, 5 unidades de cálculo no sistema e um número máximo de corpos igual a 4095, como o anteriormente usado, é atingida uma frequência de funcionamento de 250 MHz e a utilização de *slices* é apenas 7% dos disponíveis numa XC7VX980T, 10960 de 153000 *slices*. A utilização de *XtremeDSP slices* é ainda menor, 1% dos totais, 51 de 3600 *slices*. Com base nestes valores, e de acordo com as expressões anteriores, podemos estimar um máximo de 11 unidades de processamento e uma taxa de cálculo máxima de 49,5 *GFLOPS*.

A precisão do cálculo foi também analisada, continuando o estudo efetuado no capítulo 3.5. Para um período total de 365 dias o erro relativo final das posições resultante é bastante superior no sistema desenvolvido, que utiliza mantissas com 16 bits, quando comparado com o erro obtido para uma simulação realizada a 32 bits, conforme mostrado na Tabela 6.4. Embora este erro possa em parte ser compensado com a utilização de um menor intervalo temporal entre passos de simulação, isto implica um maior tempo de processamento.

Dimensão da mantissa	Intervalo temporal [s]	Erro médio relativo
23 bits	100	6,07E-05
16 bits	100	6,21E-01
	10	2,76E-01
	5	1,42E-03

Tabela 6.4 - Erro médio relativo das posições finais para diferentes dimensões de mantissa e intervalos de simulação.

7 Conclusões e Trabalho Futuro

Na construção do sistema realizado neste trabalho de projeto tomou-se a opção de apenas paralelizar o cálculo da aceleração provocada pela força gravítica, dado o facto de esta ser a componente do algoritmo que tem maior peso computacional, $\mathcal{O}(N^2)$. Uma maior paralelização da integração numérica das posições e das velocidades também seria possível, mas seria ocupada uma maior quantidade de recursos sem obter uma melhoria significativa da taxa de cálculo final. Para um elevado número de corpos o peso da integração numérica utilizada no algoritmo com cálculo direto da força gravítica é desprezável.

A capacidade de cálculo do sistema desenvolvido foi drasticamente limitada pelo número de unidades de processamento realizáveis, dados os limites de área impostos pela *FPGA* utilizada. A frequência de funcionamento de 150 MHz foi escolhida tendo em conta os recursos disponíveis. Seria possível realizar uma unidade de cálculo com frequência superior, por exemplo 200 MHz, mas ao incluir apenas uma unidade no sistema a taxa de cálculo iria baixar. Numa *FPGA* com maior quantidade de recursos seria possível obter uma taxa de cálculo maior. Outra opção seria a utilização de *FPGA* mais recentes, as da série 7 da Xilinx por exemplo, que, para além de possuírem maior número de recursos lógicos [32], apresentam recursos com maior potencialidade [22], como as *LUT6* e os *DSP48E1*, referidos em 0 e 0. Isto permitiria também realizar um sistema de simulação com precisão de 32 bits, para se obter uma maior precisão do cálculo. Por outro lado, a tecnologia utilizada nas *FPGA* mais recentes permite obter frequências de funcionamento superiores para a mesma área utilizada, permitindo uma contribuição ainda maior de cada unidade de processamento para a taxa de cálculo final.

A estrutura apresentada é modular e facilmente configurável por meio da alteração dos parâmetros do sistema, embora sempre que a configuração de latências e largura de bits dos dados seja alterada os *IP Core* afetados tenham de ser gerados de novo. Dada a arquitetura utilizada, seria possível alterar o tipo de força calculada sem grandes modificações no sistema. A utilização da representação de vírgula flutuante permite processar valores com ordens de grandeza bastantes diferentes, tais como as massas de corpos celestes ou a carga elétrica de partículas subatómicas. Também seria possível alterar o tipo de integração numérica, embora esta operação exija um esforço maior.

Uma solução que utilizasse o método de Barnes e Hut, poderia ter sido também analisada, visto o mesmo algoritmo ter sido estudado neste trabalho. Com o estudo realizado seria possível criar uma arquitetura dedicada à construção da árvore e utilizar os módulos de cálculo criados para efetuar a integração numérica e o cálculo da aceleração. No entanto não seria possível realizar em tempo útil outro sistema de simulação. Outra questão seria a paralelização da construção e

leitura da árvore e resolução dos problemas de acesso à memória resultantes da utilização de uma árvore comum. Não seria possível replicar a estrutura em memória da árvore por cada unidade de processamento pois isso esgotaria os recursos de memória interna da *FPGA* utilizada.

O método de comunicação com o meio exterior tem um débito inferior ao desejado para as transferências de dados dos elementos dos corpos pois este não foi um ponto foco do trabalho. Para uma simulação com um elevado número de corpos e um elevado número de passos de simulação, o tempo de comunicação de dados não é relevante, pois só é realizada uma transferência no início e outra no fim da simulação. No kit de desenvolvimento utilizado estava também disponível uma interface *Ethernet*, mas a utilização desta implicaria um tempo adicional de configuração da mesma, para além de um maior consumo de recursos, ao passo que a interface de porta série é de utilização quase imediata e traduz-se num gasto mínimo de recursos.

Num trabalho futuro seria de interesse explorar mais a fundo outros algoritmos de cálculo da força gravítica e se possível apresentar estruturas de *hardware* dedicado para os mesmos. O estudo duma arquitetura baseada num microcontrolador com periféricos de *hardware* dedicados às operações de vírgula flutuante específicas do problema também seria uma possibilidade a ponderar. A utilização de um kit de desenvolvimento com outras possibilidades de comunicação como o PCI Express seria também um ponto a considerar.

Bibliografia

[1] D. Chui, An FPGA Implementation of the Ewald Direct Space and Lennard-Jones Compute Engines, Toronto, 2005.

[2] T. Naab, N-body Simulations and Galaxy Formation, Munich, 2006.

[3] J. Makino e H. Daisaka, GRAPE-8 — An Accelerator for Gravitational N-body Simulation with 20.5Gflops/W Performance, Salt Lake City, 2012.

[4] A. Patel, C. A. Madill, M. Saldaña, C. Comis, R. Pomès e P. Chow, A Scalable FPGA-based Multiprocessor, Toronto, 2006.

[5] J. S. Kim, P. Mangalagiri, K. Irick, M. Kandemir, V. Narayanan, K. Sobti, L. Deng, C. Chakrabarti, N. Pitsianis e X. Sun, TANOR: A tool for accelerating N-body simulations on reconfigurable platform, State College, 2007.

[6] S.-G. SEO, Comparison of Parallel Solutions to N-Body Problems, 1999.

[7] E. Bertschinger, Simulations of structure formation of the universe, Massachusetts, 1998.

[8] J. Barnes e P. Hut, A hierarchical O(N log N) force-calculation algorithm, Princeton, 1986.

[9] S. Lee, An FPGA Implementation of the Smooth Particle Mesh Ewald Reciprocal Sum Compute Engine (RSCE), Toronto, 2005.

[10] T. Lakoba, "Simple Euler method and its modifications," [Online]. Available: http://www.cems.uvm.edu/~tlakoba/math337/notes_1.pdf. [Acedido em Agosto 2013].

[11] F. Ercolessi, "The Verlet algorithm," Outubro 1997. [Online]. Available: http://www.fisica.uniud.it/~ercolessi/md/md/node21.html. [Acedido em Agosto 2013].

[12] K. Valle e A. Vera, "Métodos numéricos de Euler e Runge-Kutta," 2012. [Online]. Available: http://www.mat.ufmg.br/~espec/Monografias_Noturna/Monografia_KarineNayara.pdf. [Acedido em Agosto 2013].

[13] F. Boesch, "Integration by Example - Euler vs Verlet vs Runge-Kutta," Agosto 2010. [Online]. Available: http://codeflow.org/entries/2010/aug/28/integration-by-example-euler-vs-verlet-vs-runge-kutta/#euler-integration. [Acedido em Agosto 2013].

[14] A. Araújo e M. Vicente, "Capítulo 6 Métodos numéricos para problemas diferenciais ordinários," [Online]. Available: http://www.mat.uc.pt/~alma/aulas/anem/sebenta/cap6.pdf. [Acedido em Agosto 2013].

[15] Xilinx, "What is a FPGA?," [Online]. Available: http://www.xilinx.com/fpga/index.htm. [Acedido em Setembro 2013].

[16] Altera, "FPGAs," [Online]. Available: http://www.altera.com/products/fpga.html. [Acedido em Setembro 2013].

[17] Xilinx, "XtremeDSP for Virtex-4 FPGAs," 2008. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug073.pdf. [Acedido em Setembro 2013].

[18] Xilinx, "Virtex-4 FPGA User Guide," 2008. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug070.pdf. [Acedido em Agosto 2013].

[19] Xilinx, "Virtex-II Platform FPGAs: Complete Data Sheet," Novembro 2007. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf. [Acedido em Setembro 2013].

[20] Xilinx, "Virtex-5 FPGA User Guide," 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug190.pdf. [Acedido em Agosto 2013].

[21] Xilinx, "Virtex-6 FPGA Configurable Logic Block User Guide," 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug364.pdf. [Acedido em Agosto 2013].

[22] Xilinx, "Spartan-6 FPGA Configurable Logic Block User Guide," 2010. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug384.pdf. [Acedido em Agosto 2013].

[23] Xilinx, "7 Series FPGAs Configurable Logic Block User Guide," 2013. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf. [Acedido em Agosto 2013].

[24] G. Ivancescu, "Fixed Point Arithmetic and Tricks," Dezembro 2007. [Online]. Available: http://x86asm.net/articles/fixed-point-arithmetic-and-tricks/. [Acedido em Setembro

2013].

[25] IEEE, 754-2008 - IEEE Standard for Floating-Point Arithmetic, 2008.

[26] A. Proença, "Arquitectura de Computadores," 2000. [Online]. Available: http://gec.di.uminho.pt/discip/TextoAC/anexoB.html. [Acedido em Setembro 2013].

[27] A. Araújo, "Projecto de Sistemas Digitais - Aritmética em Vírgula Flutuante: Algoritmos e Arquitecturas," Novembro 2004. [Online]. Available: http://paginas.fe.up.pt/~aja/PSD2004_05/slides/VFL.pdf. [Acedido em Setembro 2013].

[28] Xilinx, "ML401/ML402/ML403 Evaluation Platform," Maio 2006. [Online]. Available: ML401/ML402/ML403 Evaluation Platform. [Acedido em Setembro 2013].

[29] Xilinx, "Virtex-4 Family Overview," 2010. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf. [Acedido em Agosto 2013].

[30] Xilinx, "LogiCORE IP Floating-Point Operator v5.0," 2011. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf. [Acedido em Agosto 2013].

[31] X. Wang e M. Lesser, "VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware," Setembro 2010. [Online]. Available: http://delivery.acm.org/10.1145/1840000/1839486/a16-wang.pdf?ip=193.137.129.117&id=1839486&acc=ACTIVE%20SERVICE&key=C2716FEBFA981EF13FBF88496CDDB76007025F6048F85584&CFID=379760757&CFTOKEN=24726960&__acm__=1384707165_04199964f1926b201b0931a77be856f9. [Acedido em Setembro 2013].

[32] Xilinx, "7 Series FPGAs Overview," Julho 2013. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. [Acedido em Novembro 2013].

[33] Xilinx, "7 Series DSP48E1 Slice User Guide," 2013. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf. [Acedido em Setembro 2013].

[34] Xilinx, "Spartan-3A FPGA Family: Data Sheet," 2010. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds529.pdf. [Acedido em Setembro 2013].

[35] Xilinx, “Spartan-6 FPGA DSP48A1 Slice User Guide,” 2009. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug389.pdf. [Acedido em Setembro 2013].

[36] Xilinx, “Virtex-5 FPGA XtremeDSP Design Considerations,” 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug193.pdf. [Acedido em Setembro 2013].

[37] Xilinx, “Virtex-6 FPGA DSP48E1 Slice User Guide,” 2011. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug369.pdf. [Acedido em Setembro 2013].

[38] Xilinx, “XtremeDSP DSP48A for Spartan-3A DSP FPGAs,” 2008. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug431.pdf. [Acedido em Setembro 2013].

[39] M. Leeser, “VFLOAT: The Northeastern Variable precision FLOATing point library,” [Online]. Available: http://www.coe.neu.edu/Research/rcl/projects/floatingpoint/index.html. [Acedido em Setembro 2013].

[40] L. Zhuo, G. Morris e V. Prasanna, Designing Scalable FPGA-Based Reduction Circuits Using Pipelined, Los Angeles, 2005.

Anexos

- Resultados de implementação do divisor com formato (8,16)

Análise do divisor com formato (8,16) após P&R					
Ciclos de latência	Slices	Flip flops	Total LUTs	Máx. freq. [MHz]	Máx freq/ Slices
0	230	-	453	18,813	0,086
1	220	25	432	19,405	0,088
2	218	56	426	19,728	0,090
3	242	94	443	41,259	0,170
4	258	129	450	52,274	0,203
5	270	164	451	72,474	0,268
6	281	199	452	90,695	0,323
7	294	222	468	90,728	0,309
8	309	277	456	114,969	0,372
9	318	293	472	110,546	0,348
10	318	293	472	114,064	0,359
11	341	388	453	169,52	0,497
12	353	405	469	162,022	0,459
13	354	405	470	170,271	0,481
14	355	413	473	170,387	0,480
15	355	413	473	170,387	0,480
16	355	413	473	170,387	0,480
17	355	413	473	170,387	0,480
18	355	413	473	170,387	0,480
19	367	430	474	170,648	0,465
20	393	461	473	164,989	0,420
21	484	741	458	317,46	0,656

Tabela A.1 - Evolução com a latência da utilização de recursos e frequência de funcionamento do divisor.

- Resultados de implementação do divisor com formato (8,16)

Análise da raiz quadrada com formato (8,16) após P&R					
Ciclos de latência	Slices	Flip flops	Total LUTs	Máx. freq. [MHz]	Máx freq/ Slices
0	167	-	322	22,201	0,143
1	158	25	304	23,35	0,148
2	167	53	303	23,823	0,143
3	181	84	303	44,932	0,248
4	187	110	315	63,605	0,340
5	193	138	315	80,083	0,415
6	203	164	315	97,895	0,482
7	207	182	315	125,36	0,606
8	215	184	330	121,286	0,564
9	214	184	332	122,399	0,572
10	236	248	321	179,63	0,761
11	239	250	336	176,274	0,738
12	238	250	338	178,795	0,751
13	238	250	338	180,083	0,757
14	238	250	338	180,083	0,757
15	238	250	338	180,083	0,757
16	238	250	338	180,083	0,757
17	238	250	338	180,083	0,757
18	238	250	338	181,818	0,764
19	296	434	328	328,839	1,111
20	302	442	337	333,556	1,104
21	317	463	338	341,647	1,078

Tabela A.2 - Evolução com a latência da utilização de recursos e frequência de funcionamento da raiz quadrada.