UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

VICENTE SILVA CRUZ

Expansão da Arquitetura de Conjunto de Instruções MIPS para Suporte à Robótica

Trabalho de Graduação.

Prof. Dr. Philippe Olivier Alexandre Navaux

Prof. Dr. Henrique Cota de Freitas, Pontifícia Universidade Católica de Minas Gerais (PUC-Minas)

Porto Alegre, novembro de 2009.

AGRADECIMENTOS

Se eu fosse agradecer a todas as pessoas que fizeram parte da minha vida acadêmica, iria ocupar boa parte deste trabalho. Entretanto, existem agradecimentos que são indispensáveis de serem feitos, os quais escrevo com enorme prazer.

Quero agradecer primeiramente a Deus, pois sem Ele nada disso seria possível. Tenho muito a agradecer por tamanha conquista. Todos os obstáculos e desafios encontrados só vieram a me fortalecer.

Outra pessoa que foi essencial nessa caminhada, e que eu jamais poderia deixar de agradecer, é minha mãe Cleide Teresinha da Silva Cruz. Posso dizer tranquilamente que se não fosse por ela, sequer estaria escrevendo esse material. Foi graças a ela, graças a todo o suporte que eu tive dela, tanto para ingressar nesta universidade, quanto para me acompanhar na caminhada, que hoje posso concluir o curso.

Gostaria de agradecer também ao meu pai, Juarez Lima Cruz, por todo apoio, ensinamento e desabafo nas horas de dificuldade, e também por compreender as minhas ausências nos últimos tempos.

Um agradecimento muito sincero também vai ao professor Dr. Philippe Navaux, por ter me acolhido ao Grupo de Processamento Paralelo e Distribuído e dado a chance de realizar minhas pesquisas e principalmente por sua orientação neste trabalho. Também agradeço ao professor Dr. Henrique Freitas pela orientação e auxílio nas minhas pesquisas, e ao Msc. Marco Antônio, por suas ajudas, conselhos e idéias.

Também agradeço ao meu irmão Filipe Silva Cruz, pelo apoio no decorrer do curso. À Camilla Costa Rossato, pelo seu auxílio, incentivo e consolo durante vários momentos críticos tanto do curso quanto pessoais. Ao Rogério Boff pelas horas e horas de momentos filosóficos nas conversas, troca de idéias e experiências. E ao meu primo Diego pelo companheirismo.

Por fim, agradeço aos meus amigos e colegas de faculdade, em especial ao Thiago Presa, Guilherme Macedo, Marcos Crippa e Felipe Flores, pela amizade e parceiria nos trabalhos realizados no decorrer do curso. E aos meus amigos e companheiros da vida, especialmente o Fabrício e o Igor, também deixo um “muito obrigado”.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ADL	Architecture Description Language
ALU	Arithmetic and Logic Unit
ARM	Advanced RISC Machine
ATM	Asyncronous Transfer Mode
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CNC	Computer Numerical Control
CPLD	Complex Programmable Logic Device
EX	Execution Stage
EPROM	Erasable Programmable Read Only Memory
FP	Floating Point
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
Func	Function
ID	Instruction Decode Stage
IF	Instruction Fetch Stage
IMM	Immediate
IO	Input-Output
IOB	Input-Output Buffer
ISA	Instruction Set Architecture
iRMX	Real-time Multitasking eXecutive
LUT	Lookup Table
MEM	Memory Access Stage
MIPS	Microprocessor with Interlocked Pipeline Stages
MSB	Most Significant Bit
Opcode	Operation Code
OOO	Out-Of-Order

PDA	Personal Digital Assistant
PLA	Programmable Logic Array
SDL	System Description Language
Shamt	Shift Amount
RCCL	Robot Control C Library
RISC	Reduced Instruction Set Computer
RD	Register Destiny
RS	Register Source
RT	Register Target
RVP	Robotics Vector Processor
PC	Program Counter
PUMA	Programmable Universal Machine for Assembly
TLM	Transaction Level Modeling
VAX	Virtual Address eXtension
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WB	Write Back Stage

LISTA DE FIGURAS

RESUMO

Arquitetura de computadores é uma área que tem se desenvolvido muito nos últimos anos, e as pesquisas são cada vez mais crescentes. Os avanços tecnológicos atuais nos permitem processar grandes quantidades de dados em pouco tempo, e também auxiliam diversas áreas do conhecimento, como a robótica. Este trabalho tem por objetivo propor a extensão da arquitetura de conjunto de instruções do processador de propósitos gerais MIPS através da inclusão de instruções que auxiliam nos cálculos necessários ao movimento de robôs.

Para atingir esse objetivo fez-se um estudo na área da robótica para verificar os tipos de robôs existentes, seguido da análise matemática dos movimentos realizados por esses robôs, e da elaboração das novas instruções. A inclusão das operações robóticas no conjunto de instruções foi feita em duas etapas: a primeira envolveu a modificação e simulação do novo *ISA* no nível de arquitetura, ou seja, com a abstração dos detalhes físicos de aumento de área e velocidade, e a segunda, o desenvolvimento no nível de hardware para a obtenção desses valores físicos. A primeira etapa teve o objetivo de avaliar o desempenho de velocidade do novo *ISA* em relação ao original, obtidos através da simulação de uma aplicação que emula o movimento de um braço robótico. Uma vez que se constatou um ganho significativo de desempenho de velocidade com esta inclusão, a próxima etapa focou na geração e avaliação dos custos físicos pelas modificações da organização do processador para que fosse possível incluir essas instruções. Esses resultados foram obtidos através da sintetização da descrição do processador, na linguagem VHDL, em *FPGA*. Apesar de se obter um aumento significativo da área, a implementação desse processador é viável devido ao aumento da frequência de operação e alto ganho de desempenho de velocidade.

Palavras-Chave: arquitetura de computadores, arquiteturas específicas, computação reconfigurável, robótica.

Instruction Set Architecture Expansion of the MIPS Processor for Robotics Support

ABSTRACT

The computer architecture field has been improving a lot in the later years, and its research is increasing even more. Its current technological advances allow us to process a big amount of data in a short time, and it also helps in others knowledge fields, like robotics. The objective of this work is to propose an extension of the MIPS general purpose processor Instruction Set Architecture through the inclusion of instructions that helps on the needed calculations for the robots moving.

To reach this objective we studied the robotics field to check which robot types exists, followed by a mathematical analysis of their movements, and new instructions elaboration. The instruction set inclusion of the robotics operations was made in two steps. The first, it involved the modification and simulation of the new ISA at the architecture level, that is, with the physics details abstraction of area increasing and speed, and the second, the development at hardware level to obtain these physics values. The first step had the objective to compare the new ISA speed performance with the older, checked through the simulation of an application that emulates the movement of a robotic arm. Once we noted a significative speed performance gain with the inclusion, the next step focused on the physics costs generation and evaluation of the processors organization modification needed to insert these instructions. These results were obtained through the processor VHDL description synthetization in FPGA. Even with a significant increase in the area, we noticed that the implementation of this processor is valid, because of the frequency increase and it's high velocity performance.

Keywords: computer architecture, specific architectures, reconfigurable computer, robotics.

1 INTRODUÇÃO

Os processadores vêm aumentando cada vez mais seu poder de processamento. Tarefas que em anos passados levariam horas ou dias, hoje são executados em minutos ou até mesmo, segundos. Em paralelo, devido à mudança tecnológica da válvula para o silício, o seu custo de fabricação e aquisição diminuiu bastante (Rosen, 1969). Hoje em dia, graças às reduções consideráveis da área ocupada e da energia necessária para operá-los, os computadores evoluíram para vários modelos, e conseguem ocupar diversos ambientes (Ceruzzi, 2003). Pode-se encontrar computadores em casa através dos modelos *desktops*, ou dispor da mobilidade proporcionada pelos *notebooks*. Atualmente é possível ter as funcionalidades de um computador até nos telefones celulares ou em *PDAs* (*Personal Digital Assistant* – Assistente Digital Pessoal). Essa nova geração possui um poder de processamento muito maior que os antigos *mainframes*.

Outra área tecnológica que vem acompanhando e se beneficiando diretamente da evolução computacional é a robótica (Rosheim, 1994). Com as reduções de área e custo, é possível criar *chips* controladores que permitam aos robôs executarem tarefas que antes não eram possíveis. Porém, junto com a possibilidade de criação desses robôs, surgem também alguns desafios.

Em situações nocivas ao homem, nas quais ele deve desempenhar tarefas críticas, os robôs vêm em grande auxílio. Ambientes inatingíveis tais como o planeta Marte (Huntsberger, 2000), ou de difícil acesso e de exploração bastante perigosa (Albers, 1999), podem ser alcançados, pois os robôs aparecem como excelentes substitutos. Embora grande parte das ordens recebidas por essas máquinas seja feita remotamente, muitas vezes elas devem tomar suas próprias decisões de forma rápida. A decisão de qual caminho ela deve tomar para continuar a exploração, quando o sinal de controle remoto se perde, é um bom exemplo.

Um ambiente onde os robôs também estão bastante presentes é nas linhas de montagem e manufatura das fábricas. Eles são responsáveis por substituir o homem na execução de tarefas pesadas e repetitivas. No ramo industrial automobilístico, a General Motors foi pioneira no uso de robôs para a confecção de automóveis (Nof, 1999), e no setor metalúrgico, a Gerdau também ganha destaque com a automação de processos para a confecção de aço (Ferreira, 2003).

Os robôs não servem apenas como substituto ao homem em ambientes hostis, pesados ou repetitivos. Eles vêm desempenhando um importante papel na medicina e demais áreas da saúde. Através da precisão obtida por essas máquinas, as cirurgias são realizadas com o mínimo de cortes, agindo diretamente no ponto desejado, facilitando a regeneração dos pacientes (Taylor, 2003). Alguns robôs também são capazes de ajudar na compreensão da fisiologia humana (Dario, 1996). Entretanto, talvez a maior

contribuição para a área médica, tanto da computação e robótica, junto com a comunicação de dados, seja a possibilidade de realizar cirurgias a longa distância (Marescaux, 2002). Através de conexões *ATM* (*Asyncronous Transfer Mode* – Modo de Transferência Assíncrona) dedicadas, médicos cirurgiões que estão em um país são capazes de realizar uma cirurgia de alta precisão em pacientes localizados em outro.

1.1 Problemas

Uma vez que muitas situações envolvem vidas humanas, diversas variáveis devem ser analisadas cautelosamente durante o projeto de um robô, e o desempenho é uma delas. As aplicações exigem respostas imediatas, e é imprescindível investir na velocidade de processamento. Entretanto, as operações necessárias para realizar movimentos robóticos durante operações críticas exigem uma latência bastante significativa que deve ser levada em consideração. Em robôs que possuem apenas um controlador central, essa latência tende a engargalar o fluxo de execução das tarefas que devem ser executadas.

1.2 Motivação

Para ajudar a ganhar desempenho e tempo nas tomadas de decisões e realização de movimentos robóticos, um processador pode ser perfeitamente empregado como co-processador aritmético do controlador central. Dessa forma, uma vez que haverá um balanceamento de carga de processamento, o controlador disponibilizará de mais tempo para realizar outras atividades. Conforme o exemplo do robô explorador, a chance de decidir pelo melhor caminho para continuar a exploração é bastante alta, e com mais tempo disponível, a *CPU* poderá executar algoritmos de inteligência artificial com maior eficiência.

Embora o uso de circuitos dedicados aumente o desempenho de robôs em tarefas específicas, optou-se pelo uso do processador de propósitos gerais MIPS. Essa escolha foi feita porque assim é possível programar a máquina para desempenhar atividades diferentes. Dessa forma, uma vez que o robô finalize uma tarefa, é possível reconfigurá-lo, e usá-lo em uma nova aplicação, aumentando o uso dos recursos disponíveis.

1.3 Objetivos e Metas

Este trabalho propõe o uso do MIPS como um co-processador aritmético de um *chip* controlador de robôs, e visa expandir sua arquitetura de conjunto de instruções com a inclusão de instruções que auxiliem nos cálculo necessários à movimentação destes. O objetivo dessas instruções é obter maior desempenho na realização dos cálculos necessários para a movimentação robótica. Com isso, tem-se por meta mostrar que o desempenho do co-processador aritmético que possui essas instruções é superior ao de um processador que não as possui.

1.4 Organização da Monografia

A seguir, segue uma descrição dos conteúdos abordados em cada seção. A seção 2 apresenta uma introdução aos processadores através de tópicos que descrevem suas funcionalidades e técnicas para ganho de desempenho, concluindo com os trabalhos correlatos. A seção 3 mostra os estudos realizados na área da robótica, mostrando os

tipos de robôs existentes. Na seção 4 é feita uma análise matemática que descreve as operações responsáveis pela movimentação destes robôs. A seção 5 é vista a definição das instruções que executam os cálculos necessários para realizar essas movimentações, além de apresenta a Linguagem de Descrição de Arquiteturas (*Architecture Description Language – ADL*) *ArchC* (Rigo, 2004) e sua importância para a elaboração de conjunto de instruções. Ainda nesta seção, verificamos o impacto no desempenho que as novas instruções causaram no processador. Na seção 6 é analisado o custo organizacional do processador para se incluir as novas instruções, tais como a frequência do processador e aumento de área. Finalmente, na seção 7 são apresentados as conclusões e trabalhos futuros.

2 INTRODUÇÃO AOS PROCESSADORES

Para que seja possível acompanhar o desenvolvimento do trabalho, é importante explicar alguns tópicos sobre arquiteturas de computadores. Serão abordados aqui os conceitos introdutórios sobre os processadores, assim como definição e técnicas. A seguir são mostrados os trabalhos correlatos.

2.1 Processadores

Os processadores costumam ser o núcleo de controle de diversos sistemas, e por essa característica também são conhecidos por Unidade Central de Processamento (*Central Processing Unit – CPU*). Eles ficaram popularmente conhecidos devido ao seu uso nos computadores pessoais, mas diversas são as aplicações nas quais um processador pode atuar. Uma área cada vez mais crescente na qual as *CPU*s executam uma tarefa crucial é em ambientes de sistemas embarcados (Carro, 2003). Esta área se ramifica em várias outras como a indústria automotiva, telefones celulares, assistentes pessoais digitais (*Personal Digital Assistant - PDA*) e a própria robótica (Bräunl, 2006). Tais exemplos são apenas alguns ambientes de atuação dos processadores.

A seguir será dada uma breve explicação do funcionamento dos processadores, assim como técnicas empregadas aos mesmos para se obter um melhor rendimento. Para obter informações mais detalhadas a respeito, consulte Patterson (2005).

2.1.1 Introdução

Todo processador é conceitualmente dividido em duas partes: arquitetura e organização. A arquitetura é o que define seu conjunto de instruções (*Instruction Set Architecture – ISA*), ou seja, todas as instruções que ele é capaz de executar. Já a organização é a forma como o processador é implementado tal que ele consiga executar suas instruções. No que diz respeito à funcionalidade, ela se divide basicamente em três atividades principais: a busca de uma instrução na memória, decodificação desta instrução, e sua execução (Hwang, 1984). A definição do *ISA* e a forma de execução de cada uma das atividades principais é o que diferencia um processador do outro.

2.1.2 Unidade de controle e caminho de dados

Da mesma forma, o processador também é dividido fisicamente em duas partes: unidade de controle e caminho de dados. Essa divisão é comumente usada em projetos de circuitos digitais extensos e complexos para obter maior controle e gerenciamento da aplicação (Carro, 2001). Como o próprio nome diz, é responsabilidade da unidade de controle a gerência da execução das tarefas do *chip*. Esta unidade é controlada por uma máquina de estados finita (*Finite State Machine – FSM*) do tipo Mealy que, dado o

estado atual e os sinais de entrada, determina o próximo estado. Através deste novo estado, são gerados sinais de controle que indicam e habilitam as instruções que devem ser realizadas naquele momento no caminho de dados.

Por sua vez, o caminho de dados é a parte do processador responsável pela execução das instruções propriamente ditas. Ela normalmente é composta de registradores, multiplexadores, codificadores e decodificadores, e unidades lógicas e aritméticas (*Arithmetic and Logic Unit – ALU*), que realizam suas respectivas tarefas à medida que os sinais de controle são recebidos.

2.1.3 O processador MIPS

Uma vez que o processador usado neste trabalho é o MIPS, será explicada, resumidamente, as suas características. É importante explicar suas funcionalidades, tais como a busca, decodificação e execução das instruções, para melhor compreender a técnica *pipeline* descrita a seguir.

O MIPS (*Microprocessor with Interlocked Pipeline Stages* – Microprocessador com Estágios Pipeline Integrado) é um processador RISC (*Reduced Instruction Set Computer* – Computador com Conjunto de Instruções Reduzidos) de 32 bits do modelo *Harvard*, ou seja, com memória de instruções e memória de dados separados. Ele possui uma quantidade menor de instruções no seu *ISA* que um processador CISC (*Complex Instruction Set Computer* – Computador com Conjunto de Instruções Complexas). Como o próprio nome diz, os processadores CISC são capazes de executar operações complexas em uma única instrução, o que reduz consideravelmente a sua velocidade. O propósito dos processadores RISC é oferecer maior desempenho através da elaboração de um conjunto de instruções simples, que não exijam muito tempo para serem executadas. Assim, uma instrução complexa CISC, de alta latência, é realizada por várias instruções RISC de alta velocidade.

No MIPS as instruções são acessadas de quatro em quatro endereços. Isso acontece porque a memória é endereçada a *byte*, e uma vez que uma *word* contém 32 bits, é necessário buscar 4 *bytes* da memória para obter uma instrução ou dado. O MIPS usa a semântica *Big Endian*, ou seja, os primeiros *bytes* buscados da memória são os mais significativos da *word*. Por exemplo, se for referenciado o endereço de memória 4 para buscar uma instrução, então serão buscados os *bytes* das posições 4 (contendo os bits 31 até 24), 5 (bits 23 até 16), 6 (bits 15 até 8) e 7 (com os bits menos significativos de 7 até 0) respectivamente, sendo o endereço 8 o inicio da próxima instrução.

2.1.3.1 Formato das Instruções

Existem três tipos de instruções no MIPS. As instruções do tipo Registrador (*tipo – R*) realizam operações aritméticas e lógicas com dados buscados do banco de registradores, salvando no mesmo a resposta da operação. Conforme consta na Figura 2.1 (A), esse tipo de instrução possui 6 campos. O código do campo *Opcode* (*Operation Code* – Código da Operação), de 6 bits, informa que a instrução é do *tipo – R*, e o campo *Func*, também de 6 bits, determina qual é a operação a ser realizada. O campo *RS* (*Register Source* – Registrador de Origem) informa qual é o registrador a ser lido do banco de registradores para obter o primeiro operando da operação. De forma semelhante, o campo *RT* (*Register Target* – Registrador de Destino) informa qual é o registrador a ser lido do banco de registradores para obter o segundo operando. *RD* (*Register Destiny* – Registrador de Destino), por sua vez, informa o registrador que

armazenará o resultado da operação realizada por *RS* e *RT*. Por fim, o campo *Shamt* (*Shift Amount* – Quantidade de Deslocamento) informa quantos bits devem ser deslocados, para esquerda ou para direita, nas instruções de deslocamento. Todos esses demais campos possuem 5 bits.

Instruções do tipo Imediato (*tipo – I*) são utilizadas de várias formas. As operações aritméticas e lógicas são semelhantes às do *tipo – R*. A diferença é que o segundo operando vem no campo *IMM* (*Immediate* – Imediato), de 16 bits, junto da instrução buscada da memória, e o registrador responsável por armazenar o resultado da operação é *RT*. Esse tipo de instrução também é usado para executar desvios condicionais. Nesse caso, o campo *Opcode* informa a comparação a ser feita entre *RS* e *RT*. Se a comparação for verdadeira, realiza-se o desvio para o endereço indicado no campo *IMM*. Outro tipo de operação realizada pelas instruções do *tipo – I* são as instruções de carga ou armazenamento de palavra na memória de dados, tais como *Load Word* e *Store Word*. Neste tipo de operação, o registrador indicado por *RT* irá salvar a palavra buscada da memória, no caso de uma instrução de *Load*, ou será usado como origem, em instruções de *Store*, para armazenar a palavra na memória. O endereço da memória a ser acessado é obtido pela soma do valor contido em *RS* com o endereço do campo *IMM*. Por exemplo: se no campo *IMM* possuir o valor 50, e se 32 for o dado armazenado no registrador *RS*, o endereço da memória de dados a ser acessado é 82. A Figura 2.1 (B) mostra o formato desse tipo de instrução.

As instruções do ultimo formato, tipo – *Jump* (*tipo – J*), são as instruções de desvio incondicional. O campo *Opcode* informa o código da operação de desvio a ser tomado, e o campo *Destiny Address* de 26 bits contém o endereço de destino para o qual o programa será desviado. Desvio e retorno de sub-rotinas e instruções de saltos em geral se enquadram nesse tipo. O formato desse tipo de instrução pode ser visto na Figura 2.1 (C).

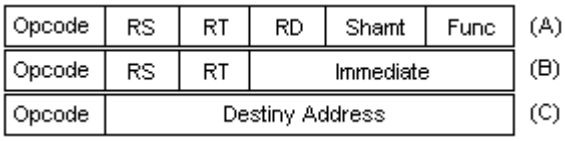

Figura 2.1: Formato das instruções *tipo – R* (A), *tipo – I* (B) e *tipo – J* (C).

2.1.3.2 Execução das operações – o caminho de dados do MIPS

No MIPS original (isto é, sem otimizações de desempenho), o caminho de dados é dividido em cinco partes, com cada parte executada em um ciclo de *clock*, mas nem todas as instruções passam pelas cinco etapas para serem concluídas. A primeira parte é conhecida como Busca da Instrução (*Instruction Fetch Stage – IF*), na qual o contador de programa (*program counter – PC*) envia para a memória de instruções o endereço da instrução a ser buscada. Em paralelo, o *PC* é incrementado para buscar a instrução seguinte da memória.

Na segunda etapa, chamada de Decodificação da Instrução (*Instruction Decode Stage – ID*) o registrador de instruções recebe a instrução que estava na memória no endereço apontado por *PC* e a envia para unidade de controle para ser decodificada. Em paralelo, buscam-se os possíveis operandos do banco de registradores para adiantar a etapa de execução, caso a instrução a ser executada seja do *tipo – R*. É nessa etapa também que se calcula o endereço de destino a ser recebido pelo *PC*, se a instrução buscada da memória for de desvio condicional ou incondicional.

O estágio de Execução (*Execution Stage – EX*) é o terceiro. Aqui, depois de decodificada a instrução, a unidade de controle envia para *ALU* a operação aritmética ou lógica a ser executada com os operandos buscados do banco de registradores no estágio anterior para o caso da instrução decodificada ser do *tipo – R*. Se a instrução for do *tipo – I*, a *ALU* executará a operação com um dos operandos buscado do banco de registradores e com a constante que veio nos 16 bits menos significativos da instrução. Ou ainda calculará o endereço da memória de dados a ser acessado para buscar, ou armazenar um dado. A instrução também pode ser de desvio, e nesse caso, se for um desvio condicional, a *ALU* executará a operação de comparação com os registradores *RS* e *RT*. Se a comparação for verdadeira, então ela enviará um sinal informando que o desvio deverá ser tomado, e o *PC* receberá o endereço de destino calculado no estágio *ID*. Por fim, se o desvio for incondicional, não será feita nenhuma comparação, e o *PC* receberá o novo endereço da instrução a ser buscada, realizando o desvio.

A quarta parte, conhecida por Acesso à Memória (*Memory Access Stage – MEM*) consiste em armazenar o resultado da operação lógica ou aritmética no banco de registradores, no caso de uma instrução tipo – R. Se for uma instrução tipo – I, então o resultado da operação aritmética imediata pode ser armazenado no banco de registradores, ou usado para referenciar o endereço da memória de dados na qual um dado será armazenado o dado, ou buscado.

Finalmente, a última parte, conhecida por Escrita de Volta (*Write Back Stage - WB*), tem o objetivo de armazenar no banco de registradores o dado buscado da memória de dados. A instrução *Load Word* é uma das poucas instruções que utilizam completamente as cinco partes do caminho de dados.

2.1.3.3 Pipeline

O MIPS é bastante conhecido no setor industrial. Entretanto, ele possui mais destaque no meio acadêmico por ser usado didaticamente para explicar a funcionalidade dos processadores. Talvez uma das maiores contribuições do MIPS no ramo acadêmico seja a explicação da técnica *pipeline* para obter otimização de velocidade e de recursos físicos. Apesar de ser uma técnica relativamente antiga, o *pipeline* ainda é muito usado nos processadores atuais, pois pode garantir um aumento de desempenho de até *N* vezes, onde *N* é o número de estágios do *pipeline*, conforme descrito a seguir.

Diferente da execução das instruções sem otimização, no *pipeline* todas as instruções, independente do tipo, passam pelas cinco etapas. A idéia é fazer com que todas as etapas estejam executando uma tarefa, ou seja, nenhuma parte do processador deve ficar ociosa. Para mostrar a técnica, suponha a “aplicação exemplo” a seguir: sejam cinco instruções *I1*, *I2*, *I3*, *I4* e *I5* armazenadas nas posições de memória 0, 4, 8, 12 e 16 respectivamente. Na primeira etapa, *PC* aponta para a posição 0, buscando a instrução *I1*. Na segunda etapa, enquanto a instrução *I1* é enviada para a unidade de controle para ser decodificada, *PC* já está referenciando a posição de memória 4, buscando a instrução *I2*. Na terceira etapa, a instrução *I1* é executada, *I2* é decodificada, e *PC* já referencia a posição 8, buscando a instrução *I3*. A quarta etapa segue a mesma lógica, ou seja, *I1* está na etapa de acesso à memória de dados que será acessada se a instrução *I1* for uma instrução *Load Word* ou *Store Word*, *I2* é executada, *I3* é decodificada e *I4* é buscada da memória na posição 12. Finalmente, na última etapa *I1* armazena o resultado da instrução no banco de registradores, *I2* passa pelo quarto estágio, *I3* é executada, *I4* é decodificada e *I5* é buscada da posição 16 da memória de instruções. Nesta última etapa o *pipeline* está cheio, pois todos os recursos estão sendo

utilizados. Somente agora é que se obteve o máximo desempenho da técnica, pois cada ciclo de *clock* finaliza uma instrução, ao invés de esperar os cinco ciclos de execução da instrução para buscar a próxima.

É importante lembrar que o *pipeline* não diminui o tempo de execução das instruções, pois todas elas passam pelos cinco estágios antes de concluírem. Além disso, o tempo que se espera para preenchê-lo é de, no mínimo, cinco ciclos, além do fato dele não estar sempre cheio, como será visto mais adiante. As Figuras 2.2 e 2.3 mostram o tempo de execução das instruções sem e com o *pipeline*, respectivamente. Percebe-se o ganho de desempenho ao observar que enquanto se levam 15 ciclos para executar *I1*, *I2* e *I3* no processador normal, com a técnica, todas as instruções do exemplo são executadas em apenas 9 ciclos.

Figura 2.2: Tempo de execução sem *pipeline* (Adaptado de Patterson, 2005)

Figura 2.3: Tempo de execução com *pipeline* (Adaptado de Patterson, 2005)

Para implementar a técnica, colocam-se registradores intermediários entre as etapas de execução. Assim, à medida que a instrução percorre o *pipeline*, as informações necessárias para sua execução se preservam. Considerando a "aplicação exemplo" acima, depois de decodificar a instrução *I1*, a informação do registrador no qual será salvo o resultado de uma instrução aritmética é perdida quando a unidade de controle decodifica a instrução *I2* no ciclo seguinte. A Figura 2.4 mostra a organização do MIPS com os registradores intermediários.

Entretanto, devido às características da técnica, alguns problemas acabam emergindo. Tais problemas são conhecidos como *hazards*. Um desses *hazards* é o *hazard* de dados, e ocorre quando a instrução a ser executada depende da resposta da instrução anterior. Uma vez que a resposta de cada instrução é entregue somente no quinto estágio, a dependência por essa resposta faz instrução seguinte ficar parada por dois ciclos. Para resolver essa situação, desenvolveu-se uma técnica chamada adiantamento de dados (*data forwarding*), na qual os dados são passados para as instruções seguintes no instante em que eles são gerados. A Figura 2.5 mostra como a resposta da primeira instrução é adiantada para a segunda e a terceira.

Embora a solução resolva boa parte dos casos, ainda assim existem situações em que a instrução deve esperar pelo dado sem fazer nada. Em situações como essa, inserem-se bolhas (*stalls*) no *pipeline*, indicando que ele ficará parado. Ao injetar uma bolha no *pipeline*, o fluxo de instruções que a segue ficarão paradas durante um ciclo de *clock*, ou

seja, se houvessem cinco bolhas consecutivas, as instruções posteriores congelariam por cinco ciclos.

Vamos supor que a instrução *I3* da "aplicação exemplo" anterior seja *Load Word* e *I4* seja uma subtração entre um operando buscado do banco de registradores e o dado que veio da memória em *I3*. Como o dado de *I3* só é obtido no quarto estágio, o processador deve injetar uma bolha no *pipeline* no próximo ciclo. Logo após, o dado é liberado, e o fluxo segue normalmente, com *I4* executando sua instrução. Essa situação é mostrada na Figura 2.6.

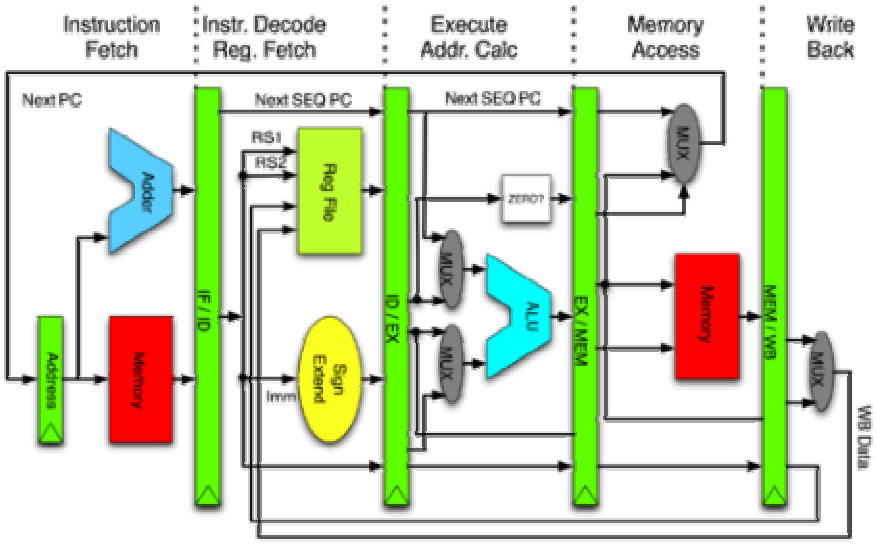

Figura 2.4: MIPS com os registradores intermediários (Patterson, 2005)

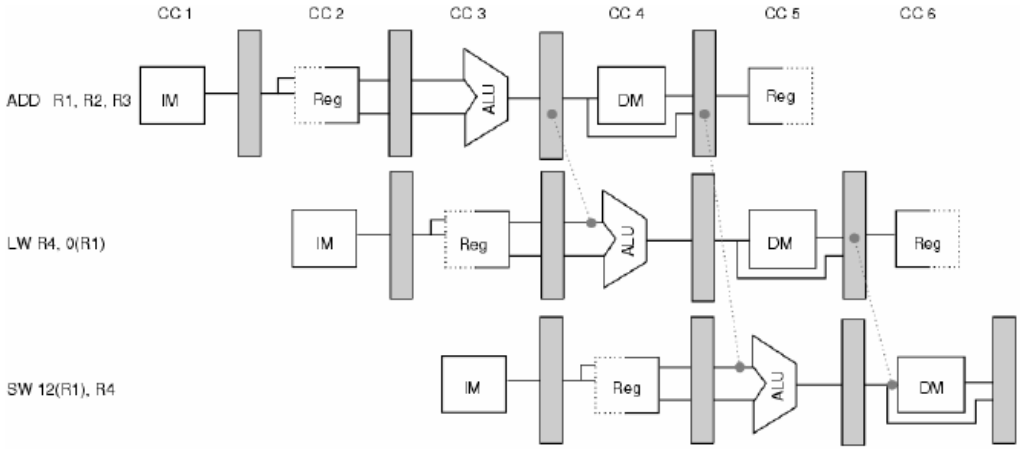

Figura 2.5: Uso do adiantamento dos dados (Patterson, 2005)

Apesar do *hazard* de dados eventualmente provocar algumas bolhas, mesmo com o uso do *forwarding*, o principal problema da técnica *pipeline* está nas instruções de desvio condicional, ou, como é conhecido, *hazard* de controle ou *hazard* de desvio. Uma vez que as instruções são buscadas em endereços consecutivos, pode acontecer de uma instrução ocasionar em um desvio no fluxo de execução, fazendo com que as instruções já contidas no *pipeline* sejam descartadas.

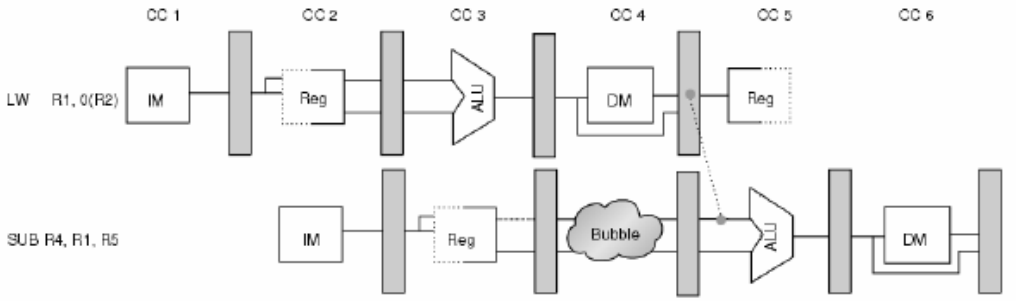

Figura 2.6: Geração de bolha no *pipeline* (Patterson, 2005)

Vamos supor que a instrução *I1* da "aplicação exemplo" seja um desvio condicional. Embora o endereço de destino do desvio seja calculado no estágio *ID*, é somente no estágio *EX* que a decisão é tomada. Nesse ponto, *I2* está sendo decodificada, e *I3* está sendo buscada da memória. Se o desvio não for tomado em *I1*, então o pipeline continua sua execução normal, com *I2* e *I3*. Entretanto, se o desvio for tomado, então as instruções *I2* e *I3* devem ser descartadas, fazendo com que o *pipeline* fique vazio.

O mecanismo de limpeza do *pipeline* também é conhecido por *flush*, e é o principal responsável pela perda de desempenho dos processadores. No caso do MIPS, a perda de desempenho não é tão impactante, pois a profundidade do *pipeline* é de 5 estágios, e se perdem apenas duas instruções. Entretanto, a perda é muito significativa em alguns processadores. Esse é o caso do Intel® Pentium® 4, que possuem um *pipeline* de 20 estágios com a tecnologia *Hyper-Pipelined* (Intel, 2001).

Embora não seja possível evitar totalmente os *hazards* de desvio, existem algumas técnicas que ajudam a evitá-las. Uma opção consiste em assumir que o desvio nunca será tomado, seguindo o processamento com o *pipeline* cheio nas situações em que não se realiza o desvio, e realizando o *flush* quando este acontece. Este é um exemplo de previsão de desvio estático, e não garante um bom desempenho devido à alta taxa de erros na previsão. As técnicas de previsão dinâmica de desvio são mais elaboradas, e garantem uma taxa de 99% de acerto. Os algoritmos de *Scoreboard* e Tomasulo (Hennessy, 2007) são exemplos de técnicas de previsão dinâmica.

2.2 Trabalhos Correlatos

Já existem pesquisas em processadores para robótica há um bom tempo. Clássicos sistemas dedicados para robótica, assim como uma arquitetura completa para aplicações com robôs são alguns exemplos de aplicações.

Em Kossman (1987), Kossman e Malowany apresentam um sistemas de controle de robôs multi-processados para RCCL dentro do iRMX, onde se descreve um projeto de extensão da Biblioteca C para Controle de Robôs (*Robot Control C Library - RCCL*) (Hayvard 1986) da arquitetura VAX para Unix para um sistema Intel multi-processado rodando o Sistema Operacional de Tempo Real iRMX para controlar o robô *Microbo Ecureuil*.

Já em Fujita (1997), Fujita e Kageyama propõem uma arquitetura aberta para entretenimento robótico, na qual se fornece uma interface para os diversos componentes de um robô tais como sensores e ativadores, mecanismos para obter informações de funções dos componentes e suas configurações, e uma arquitetura em camadas para adaptação de hardware, serviços de sistema e aplicações para o desenvolvimento de hardware e software.

Sadayappan (1989) descreve um processador vetorial robótico (RVP – *Robotics Vector Processor*) cujo propósito é explorar o paralelismo das operações realizadas com matrizes e vetores comuns da cinemática e da computação dinâmica necessárias aos controles de tempo real. Para isso, são usados três processadores de ponto-flutuante de 32 bits fortemente sincronizados, o que garante alto poder computacional.

A proposta desse trabalho não visa realizar o movimento do robô propriamente dito, mas sim em incluir instruções no processador que auxiliem no cálculo das transformações espaciais que rotacionam e transladam um ponto no espaço. Com esses resultados em mãos, o processador responsável pelos movimentos do robô poderá movimentar seus membros para o ponto de destino calculado.

3 TIPOS DE ROBÔS E SEUS FUNCIONAMENTOS

3.1 Introdução

Para determinar as operações a serem incluídas no conjunto de instruções do processador MIPS, realizou-se um estudo sobre robótica que incluiu a história dos robôs, os componentes centrais necessários à sua confecção, e os tipos principais robóticos que são usados de acordo com a aplicação. Cada um desses assuntos é visto a seguir.

3.2 História dos Robôs

A palavra robô foi usada pela primeira vez pelo novelista tcheco Karel Capek e significa trabalhador ou servo. Um robô possui várias definições dependendo do modelo, mas a mais genérica diz que é um manipulador multifuncional e reprogramável projetado para mover materiais, componentes, ferramentas ou dispositivos especializados através de variados movimentos programados para desempenhar uma variedade de tarefas (Lee, 2005).

A idéia de criar humanóides artificiais e dispositivos mecânicos sempre foi viva durante a história das civilizações. Há 4 séculos AC, na Grécia, o matemático grego Archytas de Tarentum desenvolveu o pássaro mecânico operado a vapor chamado "*The Pigeon*" (Huffman, 2007). Entre os anos 10 e 70 DC, outro matemático grego chamado Heron de Alexandria criou diversos dispositivos configurados por usuário e descreveu máquinas energizadas por pressão de ar, vapor e água (O'Connor, 2006). Embora hajam outras incidências de criações mecânicas automatizadas importantes, como o relógio astronômico de Su Song (Liu, 2002), os primeiros robôs humanóides programáveis foram inventados pelo matemático árabe Al-Jazari no ano de 1206. O propósito destes robôs era entreter os convidados do rei nas festas nobres, e apareciam dentro de um barco em um lago artificial tocando, cada um deles, um instrumento musical (Sharkey, 2006). Leonardo da Vinci também contribuiu para a história da robótica. Seus estudos em anatomia o deram uma compreensão bastante ampla sobre a movimentação humana, e fez com ele projetasse detalhadamente um cavaleiro mecânico conhecido como "O Robô de Leonardo", em 1495. Esse robô era capaz de sentar, cruzar os braços, mover a cabeça e a mandíbula (Taddei, 2007). Em 1738, diversos equipamentos autômatos foram construídos por Jacques de Vaucanson, os quais incluem um pato mecânico capaz de bater as asas, erguer o pescoço e engolir comida (Wood, 2003).

No entanto, foi no Japão que iniciou o desenvolvimento dos robôs considerados modernos. Hisashige Tanaka, ainda na sua juventude entre 1815 e 1830, desenvolveu uma série de equipamentos autômatos complexos, conhecidos como bonecos *karakuri*.

Tais bonecos eram capazes de servir chá, desenhar um ideograma japonês (*kanji*) e até mesmo disparar flechas contra um alvo (Hornyak, 2006). Porém, o primeiro robô japonês efetivo foi construído pelo biólogo Makoto Nishimura em 1928, e se chamava *Gakutensoku*, palavra japonesa que significa "prendendo com as leis da natureza" (Yomiuri, 2008). Já em 1948, os primeiros robôs eletrônicos foram criados por William Grey Walter, se chamavam Elmer e Elsier e tinham o formato de uma tartaruga. Eles podiam sentir a luz e interagir com objetos, usando esses estímulos para navegar (Grey, 1950).

Por fim, o primeiro robô industrial, e considerado oficialmente moderno, foi desenvolvido por George Devol no Instituto Neurológico de Burden, em Bristol, no ano de 1954 (Nof, 1999). Num primeiro instante, a máquina foi nomeada como *Universal Automation* (Automação Universal), devido a sua capacidade de ser programável, e posteriormente teve seu nome abreviado para *Unimate*. O robô ficou bastante conhecido por ter sido empregado nas linhas de montagem *General Motors*, a empresa pioneira no uso de robôs em suas fábricas. Sua principal tarefa consistia em erguer peças de metais quentes de uma máquina de fundição para empilhá-las. A Figura 3.1 mostra o *Unimate* interagindo com as máquinas de fundição. Por consequência, George Devol fundou a *Unimation*, a primeira fábrica de robôs. Esta empresa foi responsável por construir, em 1978, a evolução do *Unimate*: o *PUMA* (*Programmable Universal Machine for Assembly* – Máquina Universal Programável para Montagem), também para a *General Motors* (Nof, 1999). A Figura 3.2 mostra o esquemático do *PUMA* modelo 562.

Desde então, a evolução robótica se tornou acentuada. Diversas empresas começaram a adotar robôs em suas linhas de produção e montagem, cada um atendendo uma necessidade específica. Assim, com um novo mercado de trabalho em ascensão, as universidades passaram a adotar disciplinas de robótica nos cursos de graduação das engenharias mecânica, elétrica e computação. Logo em seguida, devido aos constantes avanços na área e à crescente quantidade de informações, a Corporação Elétrica Yaskawa deu origem ao curso específico de Engenharia Mecatrônica, em 1969, com o objetivo de atender a demanda do mercado. O curso envolve conhecimentos de engenharia mecânica, elétrica, e computação. Mais além, as universidades passaram a oferecer esse curso, o que permitiu que as pesquisas e desenvolvimento se intensificassem.

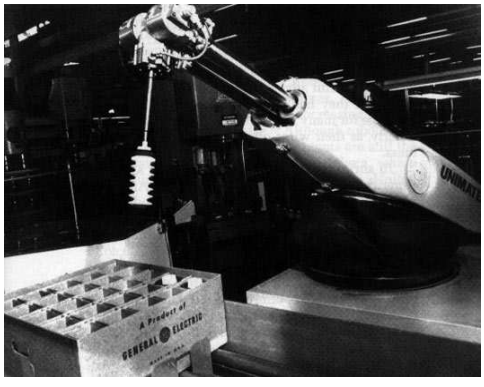

Figura 3.1: *Unimate* erguendo um pedaço de metal da máquina de fundição.

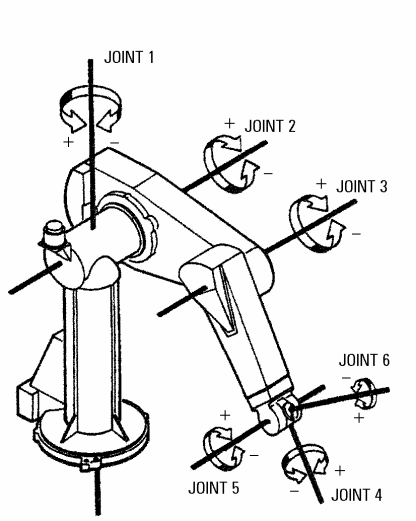

Figura 3.2: *PUMA* modelo 562 (Juang 1998).

Nos dias atuais, a robótica atua em diversos setores da humanidade, auxiliando o homem de desde a indústria automotiva, passando pela medicina, e indo até aos departamentos espaciais. Diversos são os modelos existentes, de forma que normalmente é possível empregar um equipamento autômato que satisfaça uma determinada necessidade. Para obter mais detalhes sobre a história e evolução da robótica e seu emprego no setor industrial, consulte Nof (1999).

3.3 Componentes

Para que seja possível construir e movimentar um robô, devem-se empregar diversos componentes. Além dos membros responsáveis pelo suporte do robô, existem peças que interligam, movimentam e rotacionam os membros. Também existem peças responsáveis por erguer e sustentar objetos. Por fim, os controladores centrais são responsáveis por oferecer inteligência ao robô, gerenciando seu funcionamento. Não é o objetivo deste trabalho abordar exaustivamente todas as peças necessárias à sua construção, de modo que será visto a seguir apenas tópicos dos componentes centrais. Para obter mais detalhes, consulte McKerrow (1995) e Craig (2004).

3.3.1 Bases de sustentação

Os robôs se sustentam em dois tipos de bases: fixas e móveis. Robôs com bases fixas desempenham suas tarefas sem se locomover, sendo bastante usados nas indústrias de manufatura. O *Unimate* é um exemplo de robô de base fixa. Já os robôs com base móvel são capazes de se locomover usando rodas ou pernas. A figura 3.3 mostra o Azimo (Honda, 2000), um robô em formato humanóide construído pela Honda capaz de chutar uma bola de futebol, subir escadas e até mesmo correr.

Figura 3.3: O robô Azimo (Honda, 2000).

3.3.2 Eixos e juntas

Semelhantes às vigas usadas na construção civil, ou aos ossos dos seres vertebrados, os eixos formam os membros principais do robô e oferecem sustentação. As juntas, como mostrado na figura 3.4, interligam os eixos e oferecem mobilidade.

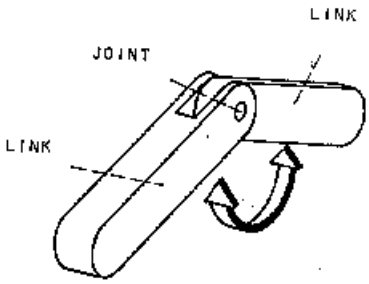

Figura 3.4: Juntas oferecendo mobilidade (Franchin, 2007).

Existem dois tipos de juntas, cada uma responsável por oferecer um tipo de mobilidade: juntas deslizantes e juntas de rotação. A junta deslizante permite que os eixos se estendam ou comprimam, oferecendo movimentos lineares. As juntas de rotação, por sua vez, são responsáveis por oferecer movimentos de rotação aos eixos. É comum que se combine as três juntas de rotação permitindo ao robô uma movimentação em um espaço esférico. As figuras 3.5, 3.6 e 3.7 mostram essas juntas.

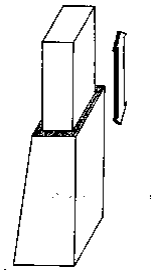

Figura 3.5: Junta deslizante (Franchin, 2007).

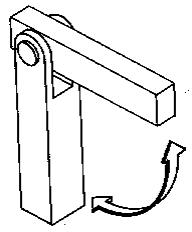

Figura 3.6: Junta de rotação (Franchin, 2007).

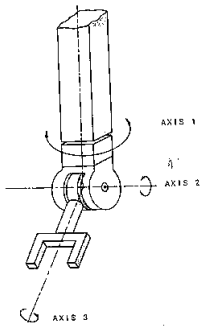

Figura 3.7: Combinação de três juntas de rotação (Franchin, 2007).

3.3.3 Sensores

Da mesma forma que os humanos usam os sentidos para perceber o ambiente ao redor de si, os robôs também são capazes de receber estímulos através de sensores. Dependendo do estímulo recebido por um determinado sensor, a máquina reage de uma forma. Entretanto, frequentemente são utilizados mais sensores no robô do que os cinco sentidos do ser humano. Em algumas aplicações é importante que se perceba a quantia de calor em um local. Já em outras, a quantidade de radioatividade de uma sala. Ainda existem atividades onde é preciso medir movimentos que são muito pequenos, ou muito rápidos ao olho humano. Exemplos de sensores robóticos incluem visão, força e aproximação.

Alguns exemplos de sensores robóticos incluem visão, força e aproximação. Os sensores visuais são muito utilizados em aplicações onde se devem coletar objetos de um lugar aleatório para movê-los a outro. Já os sensores de força são eficazes quando o robô manipula objetos sensíveis à pressão, como o encaixe de peças. Nesse ambiente, deve-se torcer uma válvula o suficiente para encaixá-la em outro compartimento, sem quebrá-lo. Por fim, como o próprio nome diz, os sensores de aproximação avisam a máquina quando ele está se aproximando de um objeto, ou o contrário. Esse tipo de sensor é muito usado em missões de exploração de terrenos abertos, para desviar de obstáculos ou coletar objetos.

3.3.4 Atuadores

Percebe-se que a implementação de robôs segue bastante próximo ao organismo humano, e em relação à aplicação de força e movimentação, o mecanismo é bastante parecido. Enquanto o homem utiliza os músculos para aplicar sua força e movimentação, os robôs possuem dispositivos eletromecânicos, chamados atuadores, para exercer o mesmo papel. Atuando em auxílio com as juntas, os robôs são capazes de se movimentarem e erguer objetos pesados. Atuadores pneumáticos, hidráulicos e elétricos são alguns exemplos bastante usados. Os dois primeiros são usados em

aplicações muito complexas, com alto controle estratégico, e último, em linhas de montagem onde é preciso alta velocidade de produção.

3.3.5 Controlador central

Finalmente, o cérebro dos dispositivos robóticos é representado por um processador conhecido por controlador central. Os tipos de processadores usados como controladores centrais são semelhantes às CPU usadas nos computadores convencionais (*Desktops* e *Notebooks*) e se comportam da mesma forma, ou seja, buscam instruções da memória, decodificam e executam. A diferença está no fato desses processadores serem desenvolvidos para ambientes embarcados, ou seja, devem ser capazes de armazenar um programa e processar dados em um ambiente com poucos recursos. Outra diferença está na forma de processamento das *I/O* (*Input-Output*, Entrada-Saída): o controlador processa os dados recebidos pelos sensores (a interface de entrada), e baseado no estado do robô e do programa carregado na memória, gera a ação a ser desempenhada pelos atuadores (a resposta de saída). Os processadores ARM (*Advanced RISC Machine* – Máquinas RISC Avançadas) são exemplos de uso para controladores (Wang, 2008).

3.4 Tipos de Robôs

Os robôs são classificados de acordo com suas características, dividindo-se em robôs cartesianos, cilíndricos e esféricos. Cada um deles possui movimentos e aplicações distintas, as quais são descritas a seguir. Para obter mais informações a respeito de tipos robótico, consulte McKerrow (1995) e Franchin (2007)

3.4.1 Robôs cartesianos

Esse tipo também é conhecido como robô linear devido a sua capacidade de realizar movimentos de extensão e retração ao longo dos eixos cartesianos X, Y e Z. O robô cartesiano é mostrado na Figura 3.8.

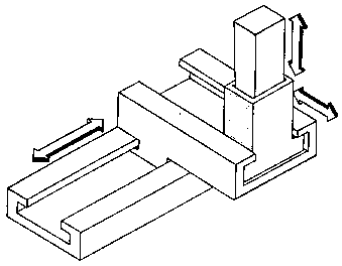

Figura 3.8: Robô cartesiano (Franchin, 2007)

A principal aplicabilidade deste robô está na automação de máquinas que são operadas por comandos abstratos. Essas máquinas são conhecidas por máquina de controle numérica computacional (*Computer Numerical Control Machine – CNC Machine*) (Reintjes 1991) e realizam operações sobre objetos que exigem alta precisão, como a queima do silício na confecção de processadores (T4, 2009) (Patterson 2005).

3.4.2 Robôs cilíndricos

Assim como os robôs cartesianos, os robôs cilíndricos são capazes de realizar movimentos de extensões e retração. Entretanto, a principal diferença (a que dá o nome a esse tipo de robô) está no fato dele conseguir fazer movimentos de rotação em torno

do próprio eixo, provendo movimentação em um espaço cilíndrico. A Figura 3.9 mostra um robô cilíndrico.

Esses robôs não possuem um grau de precisão tão elevado quanto os robôs cartesianos, e seu controle é mais complexo devido à rotação da base e aos momentos de inércia para se posicionar nos diferentes pontos que sua área de trabalho permite alcançar. Suas aplicações estão voltadas extensivamente a pesquisas de medicamentos, que incluem desenvolvimento de drogas, toxicologia e classificação do DNA (T4, 2009).

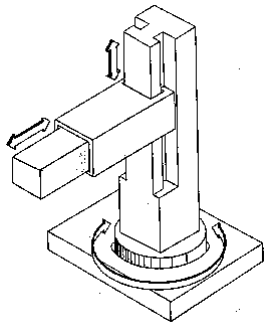

Figura 3.9: Robô cilíndrico, (Franchin, 2007).

3.4.3 Robôs esféricos

Por fim, os últimos tipos básicos de robôs são os esféricos ou polares. Eles também realizam movimentos de extensão e retração, mas são capazes de realizar rotações na base, assim como os cilíndricos, e longitudinais, permitindo a eles ter um alcance de trabalho em um espaço esférico.

Embora seja possível efetuar rotações em dois eixos, o controle desse robô é ainda mais complexo do que os cilíndricos, além de ter uma precisão menor. A Figura 3.10 mostra o robô esférico.

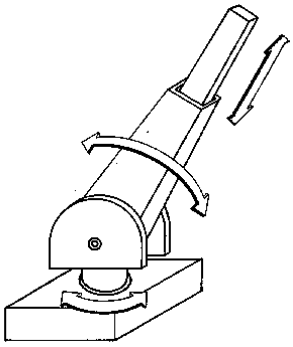

Figura 3.10: Robô esférico, (Franchin, 2007)

Devido a sua capacidade de alcançar áreas inclinadas, sua principal área de atuação está focada na indústria de manufatura de carros e soldagem (T4, 2009).

4 OPERAÇÕES MATEMÁTICAS PARA ROBÓTICA

4.1 Introdução

Para que um robô possa desempenhar suas ações, é necessário modelar matematicamente seus movimentos. Dessa forma, serão apresentados tópicos introdutórios em matemática robótica, assim como algumas definições, para melhor compreender as funções que descrevem a movimentação de robôs. Para obter um conhecimento mais profundo, assim como as deduções e as provas formais das definições apresentadas, consulte Lages (2007) e McKerrow (1995).

4.2 Definições

Seja $\boldsymbol{R^3}$ o conjunto de todos os elementos pertencentes ao espaço 3D. Seja um sistema de coordenadas *{A}*, com eixos X_A, Y_A e Z_A, representando um espaço em $\boldsymbol{R^3}$ conforme a figura 4.1.

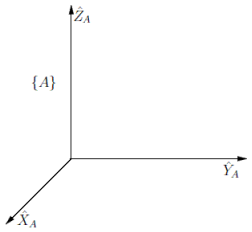

Figura 4.1: Sistema de coordenadas *{A}* em $\boldsymbol{R^3}$ (Lages, 2007).

Se *P* é um ponto nesse sistema de coordenadas, então ${}^{A}P$ é o vetor associado a esse ponto. O vetor possui componentes p_x, p_y, p_z dados como a projeção desse vetor nos eixos de *{A}* como mostra a figura 4.2. A figura 4.3 mostra a representação matricial de ${}^{A}P$.

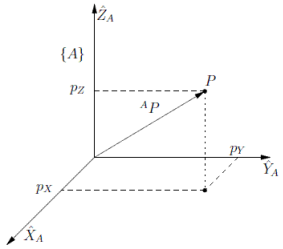

Figura 4.2: O ponto *P*, o vetor ${}^{A}P$ associado a *P*, e a projeção de ${}^{A}P$ nos eixos de *{A}* (Lages, 2007).

$$ {}^{A}P = \begin{bmatrix} p_X \\ p_Y \\ p_Z \end{bmatrix} $$

Figura 4.3: Representação matricial do vetor ${}^{A}P$ (Lages, 2007).

O vetor é um conceito abstrato para um objeto genérico, e da forma como está definido, não possui orientação nenhuma. A orientação de um objeto representa o quanto ele está rotacionado em relação ao sistema de coordenadas ao qual ele pertence. Para definir essa orientação, é preciso associar ao objeto um novo sistema de coordenadas *{B}*, com eixos X_B, Y_B e Z_B. Dessa forma, a orientação do objeto, descrita pela relação dos versores (vetores que tem módulo igual a 1) dos eixos de *{B}* em relação a *{A}*, é denotada como ${}^{A}X_B$, ${}^{A}Y_B$ e ${}^{A}Z_B$. Para simplificar a notação, define-se essa relação como ${}^{A}R_B = [{}^{A}X_B \mid {}^{A}Y_B \mid {}^{A}Z_B]$, onde ${}^{A}R_B$ é a matriz de rotação de *{B}* em relação à *{A}*. A figura 4.4 mostra graficamente essa relação. Se o objeto for um braço robótico, seria interessante determinar sua orientação em relação à origem ao qual o robô está posicionado para decidir a próxima ação a ser tomada. Essa ação pode ser o recolhimento do braço para reposicioná-lo e efetuar um novo movimento.

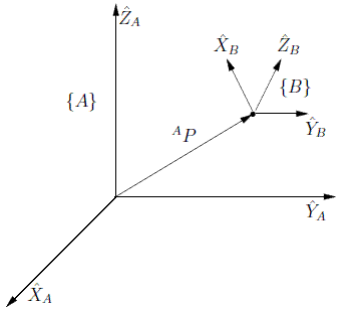

Figura 4.4: Definição da orientação de um objeto (Adaptado de Lages, 2007).

A projeção dos eixos X_B, Y_B e Z_B em *{A}* é obtida pelo produto escalar de cada versor de *{B}* pelos versores de *{A}*, como mostra a figura 4.5. Por fim, a matriz ${}^{A}R_B$ é o resultado da orientação desse objeto (figura 4.6).

$$ {}^{A}\hat{X}_B = \begin{bmatrix} \hat{X}_B \cdot \hat{X}_A \\ \hat{X}_B \cdot \hat{Y}_A \\ \hat{X}_B \cdot \hat{Z}_A \end{bmatrix} $$

$$ {}^{A}\hat{Y}_B = \begin{bmatrix} \hat{Y}_B \cdot \hat{X}_A \\ \hat{Y}_B \cdot \hat{Y}_A \\ \hat{Y}_B \cdot \hat{Z}_A \end{bmatrix} $$

$$ {}^{A}\hat{Z}_B = \begin{bmatrix} \hat{Z}_B \cdot \hat{X}_A \\ \hat{Z}_B \cdot \hat{Y}_A \\ \hat{Z}_B \cdot \hat{Z}_A \end{bmatrix} $$

Figura 4.5: Cálculo da projeção dos versores de *{B}* em *{A}* (Lages, 2007).

$$ {}^{A}R_B = \begin{bmatrix} \hat{X}_B \cdot \hat{X}_A & \hat{Y}_B \cdot \hat{X}_A & \hat{Z}_B \cdot \hat{X}_A \\ \hat{X}_B \cdot \hat{Y}_A & \hat{Y}_B \cdot \hat{Y}_A & \hat{Z}_B \cdot \hat{Y}_A \\ \hat{X}_B \cdot \hat{Z}_A & \hat{Y}_B \cdot \hat{Z}_A & \hat{Z}_B \cdot \hat{Z}_A \end{bmatrix} $$

Figura 4.6: A matriz orientação de um objeto (Lages, 2007).

Dessa forma, se chamarmos o vetor ${}^{A}P$ de ${}^{A}P_{Borg}$ e o considerarmos como o vetor responsável por apontar para a origem do novo sistema *{B}*, então, junto com a matriz

de rotação ${}^{A}R_{B}$, é possível descrever completamente a orientação do objeto em relação à origem por $\{B\} = \{{}^{A}R_{B}, {}^{A}P_{Borg}\}$.

4.3 Mapeamento de Coordenadas

Os robôs são construídos com diversos componentes, cada um representado por um sistema de coordenadas específico. Sendo assim, é importante obter a relação de um componente pertencente a um sistema de coordenadas qualquer até o sistema de origem. Essa relação é obtida pelo mapeamento de coordenadas. Se o controlador central instruir o robô a mover um componente, é através do mapeamento que ele pode realizar os cálculos necessários para movê-lo. Por exemplo, supondo que o pescoço de um robô seja representado por um sistema de coordenadas *{B}*, e a base seja representado pelo sistema de coordenadas *{A}*. A cabeça desse robô é um objeto que está associado ao pescoço, ou seja, é um objeto pertencente ao sistema *{B}*. Então, com o mapeamento da cabeça do robô até *{A}*, é possível gerar os cálculos necessários para movimentá-la.

Existem duas formas de mapeamento: o mapeamento por translação e o mapeamento por rotação. No mapeamento por translação, é assumido que dois sistemas distintos *{A}* e *{B}* estão com a mesma orientação. Se existe um ponto *P* representado em *{B}*, e um vetor ${}^{B}P$ associado a ele cujas coordenadas são conhecidas, então o mapeamento desse ponto em relação a *{A}* é dado pela soma do vetor ${}^{A}P_{Borg}$ com o vetor ${}^{B}P$, ou seja: ${}^{A}P = {}^{A}P_{Borg} + {}^{B}P$. A figura 4.7 mostra o mapeamento por translação.

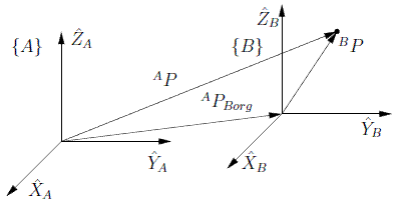

Figura 4.7: Mapeamento por translação (Lages, 2007).

O mapeamento por rotação assume que *{A}* e *{B}* têm origem no mesmo ponto. Novamente, se *P* pertence à {B}, e ${}^{B}P$ é o vetor associado a *P*, então o mapeamento por rotação é dado pela multiplicação da matriz de rotação ${}^{A}R_{B}$ por ${}^{B}P$, ou seja: ${}^{A}P = {}^{A}R_{B}\,{}^{B}P$. O roteamento pode ser visto na figura 4.8

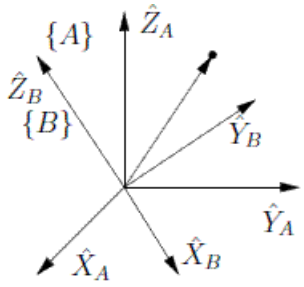

Figura 4.8: Mapeamento por rotação (Lages, 2007).

Embora seja possível efetuar a maioria dos movimentos robóticos com esses dois mapeamentos, diversas operações exigem a composição de mapeamentos, de forma que o robô possa esticar e girar um membro específico. Para descrever o mapeamento composto, será usado um sistema de coordenadas auxiliar *{C}* que tem o mesmo alinhamento que *{A}* e o mesmo ponto de origem de *{B}*. Da mesma forma como é feito o mapeamento por translação de *{A}* para *{B}*, se ${}^{C}P$ é o vetor associado a um ponto em

{C}, então ${}^{A}P = {}^{A}P_{Corg} + {}^{C}P$, onde ${}^{A}P_{Corg}$ é o vetor em *{A}* responsável por indicar a origem de *{C}*. Seguindo o mesmo raciocínio do mapeamento por rotação, ${}^{C}P = {}^{C}R_B \, {}^{B}P$. Como *{B}* e *{C}* possuem a mesma origem, então ${}^{A}P_{Corg}$ é idêntico a ${}^{A}P_{Borg}$. De maneira semelhante, se *{A}* e *{C}* possuem a mesma orientação, então ${}^{C}R_B$ também é idêntico a ${}^{A}R_B$. Dessa forma, realizando substituição direta, se conclui que a composição de mapeamentos entre *{A}* e *{B}* é dada por ${}^{A}P = {}^{A}P_{Borg} + {}^{A}R_B \, {}^{B}P$. A figura 4.9 mostra a composição realizada por essa expressão.

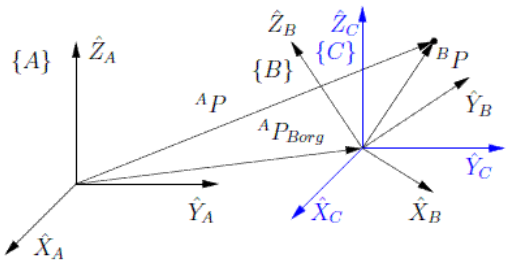

Figura 4.9: Composição de mapeamentos (Lages, 2007).

4.4 Transformações Homogêneas

Uma vez determinados os mapeamentos entre sistemas de coordenadas, é possível fazer com que um robô estique e encolha um membro, e também girá-lo. Porém, como a expressão de composição mostra, tal operação é bastante trabalhosa porque envolve duas operações matriciais: soma e multiplicação. Dessa forma, com a utilização das transformadas homogêneas, é possível realizar os mapeamentos por translação e rotação através de uma única operação, minimizando a latência.

A matriz de operação é dividida em quatro partes, cada uma responsável por realizar um tipo de operação, como mostra a figura 4.10 (A). O novo vetor ${}^{A}P$ é obtido através da multiplicação do operador de transformação homogênea pelo vetor ${}^{B}P$. Essa expressão pode ser vista pela figura 4.10 (B) mostra o formato da expressão de transformação homogênea. A operação de rotação é realizada pela matriz do canto superior esquerdo, e é representada por ${}^{A}R_B$. Já a operação de translação é executada pelo vetor ${}^{A}P_{Borg}$. Embora seja possível realizar operações de perspectiva e escala, elas não são aplicadas à robótica porque são mais voltadas para aplicações gráficas computacionais, e assim, possuem os valores nulos 0_{1x3} e 1 para perspectiva e escala, respectivamente. Por fim, a figura 4.10 (B) também mostra a representação dos vetores ${}^{A}P$ e ${}^{B}P$ em coordenadas homogêneas.

$$\text{(A)} \left[\begin{array}{c|c} \text{rotação} & \text{translação} \\ \hline \text{perspectiva} & \text{escala} \end{array}\right]$$

$$\text{(B)} \left[\begin{array}{c} {}^{A}P \\ \hline 1 \end{array}\right] = \left[\begin{array}{c|c} {}^{A}R_B & {}^{A}P_{Borg} \\ \hline 0_{1\times 3} & 1 \end{array}\right] \left[\begin{array}{c} {}^{B}P \\ \hline 1 \end{array}\right]$$

Figura 4.10: (A) Semântica da Transformação Homogênea. (B) Representação da operação em transformações homogêneas (Adaptado de Lages, 2007).

4.4.1 Operação de translação

Seja ${}^{A}P_1$ o vetor associado a um ponto no sistema de coordenadas *{A}*, e ${}^{A}Q$, o vetor de translação com componentes q_{1x}, q_{1y}, q_{1z}. O novo vetor ${}^{A}P_2$ é obtido pela soma

pontual dos componentes de ${}^{A}P_1$ com os componentes de ${}^{A}Q$, ou seja: $p_{2x} = p_{1x} + q_{1x}$, $p_{2y} = p_{1y} + q_{1y}$ e $p_{2z} = p_{1z} + q_{1z}$. A figura 4.11 mostra a aplicação da operação de translação.

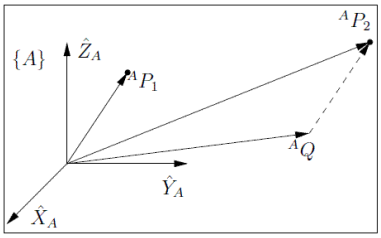

Figura 4.11: Operação de translação, (Lages, 2007).

A equação usada nessa operação é dada por ${}^{A}P_2 = D(q)^{A} * {}^{A}P_1$, onde o componente $D(q)^{A}$ representa a matriz de transformação homogênea de translação. A Figura 4.12 mostra essa matriz. Os componentes q_x, q_y *e* q_z representam as unidades de translação nas direções X_A, Y_A *e* Z_A respectivamente.

$$D(q) = \begin{bmatrix} 1 & 0 & 0 & q_x \\ 0 & 1 & 0 & q_y \\ 0 & 0 & 1 & q_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 4.12: Matriz da transformação homogênea de translação, (Lages, 2007).

Como é possível perceber ao analisar a matriz, a multiplicação matricial anula diversos elementos, de modo que a translação é obtida pela soma pontual dos vetores ${}^{A}P_1$ e ${}^{A}Q$. Uma vez que é possível estender e retrair um ponto no espaço, um *chip* controlador de robôs que execute essa instrução poderá esticar e encolher os eixos dos robôs descritos na seção 3.4.1.

4.4.2 Operações de rotação

A operação de rotação é melhor compreendida quando visualizada na Figura 4.13. Entretanto, ao invés de realizar a rotação de θ em torno de um vetor ${}^{A}Q$ qualquer, é bastante comum realizar a rotação em torno dos eixos de *{A}*. A Figura 4.14 mostra a operação de rotação em torno do eixo X_A. Com esta representação, rotaciona-se α graus em torno do eixo X_A, β graus em torno de Y_A, e γ graus em Z_A. As matrizes de transformações homogêneas $R_X(\alpha)$, $R_Y(\beta)$ *e* $R_Z(\gamma)$ de rotação nos eixos X_A, Y_A e Z_A, respectivamente, são mostradas na Figuras 4.15, 4.16 e 4.17. De forma semelhante à equação matricial de translação, as equações que geram o novo ponto ${}^{A}P_2$ são dadas por ${}^{A}P_2 = R_X(\alpha) * {}^{A}P_1$, ${}^{A}P_2 = R_Y(\beta) * {}^{A}P_1$, e ${}^{A}P_2 = R_Z(\gamma) * {}^{A}P_1$.

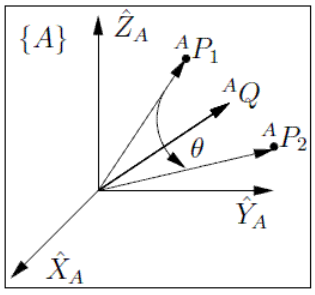

Figura 4.13: Operação de rotação, (Lages, 2007).

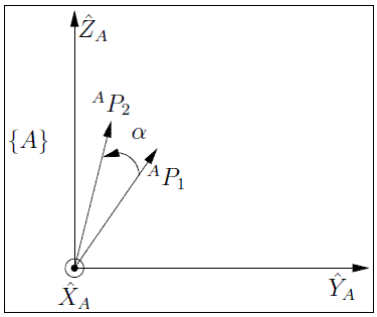

Figura 4.14: Operação de rotação no eixo X_A, (Lages, 2007).

$$R_X(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\operatorname{sen}\alpha & 0 \\ 0 & \operatorname{sen}\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 4.15: Matriz de transformação homogênea de rotação em X_A (Lages, 2007).

$$R_Y(\beta) = \begin{bmatrix} \cos\beta & 0 & \operatorname{sen}\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\operatorname{sen}\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 4.16: Matriz de transformação homogênea de rotação em Y_A (Lages, 2007).

$$R_Z(\gamma) = \begin{bmatrix} \cos\gamma & -\operatorname{sen}\gamma & 0 & 0 \\ \operatorname{sen}\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 4.17: Matriz de transformação homogênea de rotação em Z_A (Lages, 2007).

Tais operações são responsáveis por realizar os movimentos de rotação característicos dos robôs cilíndricos e esféricos. Logo, o *ISA* do processador responsável pelo controle desses robôs deve ser capaz de realizar essas instruções.

5 PROPOSTA E EXPANSÃO DO NOVO CONJUNTO DE INSTRUÇÕES

5.1 Introdução

Os capítulos 3 e 4 forneceram a base necessária para determinar as instruções que serão incluídas no MIPS. Através delas, os robôs poderão aumentar o desempenho na realização das suas atividades porque cada instrução tem o objetivo de fornecer as coordenadas de destino para onde o robô deverá mover um eixo. Porém, os cálculos realizados para determinar essas coordenadas são bastante pesados, pois envolvem operações vetoriais e matriciais complexas. Logo, a inserção das operações que obtém essas novas coordenadas no *ISA* do MIPS tenderá a minimizar a latência da geração dos resultados. Isso será possível porque através da execução de uma dessas instruções será possível obter o mesmo resultado que se conseguiria ao realizar diversos cálculos vetoriais e matriciais separados.

A um primeiro momento, deseja-se verificar o impacto de desempenho apenas no nível de conjunto de instruções, abstraindo a implementação das operações em hardware. Neste capítulo são apresentadas as novas instruções e o ambiente de desenvolvimento utilizado para sua implementação. Em seguida, é mostrada uma aplicação na qual as instruções são usadas, com o objetivo de validar o novo conjunto de instruções, seguida da conclusão dos resultados.

5.2 Definição das Novas Instruções

Através da análise dos modelos robóticos apresentados na seção 3.4, é possível notar os tipos de movimentos realizados por cada um. Embora apenas os robôs cilíndricos e esféricos sejam capazes de girar alguns membros, todos eles são capazes de expandir e contrair um determinado eixo. Por sua vez, as transformações homogêneas estudadas na seção 4.4 descrevem as operações necessárias para que essas máquinas movimentem-se de acordo com os seus modelos.

A instrução de translação é simples, e segue a mesma lógica apresentada na subseção 4.4.1. Como é possível notar, ela é realizada através da soma de dois operandos. O primeiro operando representa as coordenadas iniciais de um ponto, e o segundo, o vetor de translação. O resultado dessa soma contém as coordenadas do ponto transladado.

As instruções de rotação sobre X, sobre Y e sobre Z merecem mais atenção. Na seção 4.4.2, nota-se o uso do cálculo do seno e cosseno de um ângulo nessas operações. Dessa forma, torna-se necessário incluir essas duas novas instruções como funções

auxiliares, e tanto o seno quanto o cosseno possuem apenas um operando, o ângulo a ser calculado. Assim, finalmente é possível definir as instruções de rotação, sendo que a definição é a mesma para os três tipos. Para isso, são necessários três operandos: o primeiro é semelhante à instrução de translação, ou seja, representa as coordenadas iniciais de um ponto. Já segundo e o terceiro representam o seno e o cosseno de um ângulo, respectivamente. Como resposta dessa instrução, são obtidas as coordenadas do novo ponto rotacionado em X, ou em Y, ou em Z. Sendo assim, as novas instruções que serão incluídas no *ISA* do MIPS são:

- **Translação;**
- **Rotação em X**;
- **Rotação em Y**;
- **Rotação em Z**;
- **Seno**;
- **Cosseno**.

5.3 Desenvolvimento do Novo Conjunto de Instruções

Quando o objetivo é incluir novas instruções no *ISA* de um processador para verificar o desempenho de um novo conjunto de instruções, podem-se abstrair certas etapas do desenvolvimento do *chip* em um primeiro momento. Isso é possível através do uso de uma ferramenta que permite a implementação somente do conjunto de instruções, sem levar em consideração o custo de hardware.

5.3.1 A linguagem de descrição de arquiteturas *ArchC*

Com a utilização da linguagem de descrição de arquiteturas (*Architecture Description Language – ADL*) ArchC (Rigo, 2004), é possível descrever o conjunto de instruções de um processador de forma a se obter resultados como: o tempo de execução de um programa, a quantidade de acessos à memória e ao banco de registradores, e o total de instruções efetuadas. A linguagem *ArchC* é baseada na linguagem de descrição de sistemas (*System Descritpion Language – SDL*) *SystemC* (Ghosh, 2001). Ambas linguagem visam modelar sistemas a nível de transação (*Transaction-Level Modeling – TLM*), ou seja, se preocupam apenas com a sua funcionalidade. Os detalhes de implementação, assim como o desenvolvimento dos protocolos de comunicação entre as entidades envolvidas no ambiente, fazem parte de uma etapa separada do projeto. O *SystemC* é um conjunto de classes e macros que estende a linguagem C++, logo, as regras de sintaxe são as mesmas. Como o *ArchC* é baseado no *SystemC*, essa idéia permanece, de forma que programadores habituados à linguagem C não encontrarão muitas dificuldades para se adaptarem a ela.

A definição de um processador em *ArchC* envolve três etapas. A primeira etapa especifica detalhes da arquitetura, tais como tamanho da palavra, quantidade de registradores no banco de registradores e tamanho da memória. Se a técnica *pipeline* for incluída, então as etapas e os registradores responsáveis por separar cada uma delas, também são definidos aqui. A segunda etapa consiste na definição do conjunto de instruções, informando-se os tipos e formatos das operações do processador, junto com os mnemônicos e os operandos. Por fim, a terceira etapa especifica as funcionalidades

do processador, definindo o comportamento de todas as instruções que foram definidas na etapa anterior.

5.3.2 Implementação das novas instruções

Para desenvolver as novas instruções para o processador MIPS, foi utilizado o modelo MIPS R3000, especificado em ArchC (2007). Esse modelo descreve o MIPS com os cinco estágios de *pipeline* explicados na subseção 2.1.3.3, e serão apresentadas, aqui, as alterações realizadas nesse processador para permitir a inclusão das novas instruções. Para obter maiores informações sobre a implementação do MIPS R3000, assim como detalhes mais aprofundados à descrição de arquiteturas, consulte ArchC (2007).

A descrição original do MIPS R3000 realiza apenas operações sobre valores inteiros. Entretanto, uma vez que as instruções propostas são aplicadas sobre números pertencentes ao conjunto dos números reais, também foi preciso incluir instruções de ponto flutuante (*Floating Point – FP*) no seu *ISA*. Dessa forma foram adicionadas as seguintes instruções ao R3000 original: **soma**, **subtração**, **multiplicação e divisão em *FP***, **conversão de números inteiros para *FP*** e **movimentação de um valor em *FP*** de um registrador para outro. Porém, para compreender melhor o processamento dos números em ponto flutuante, é importante estudar o padrão IEEE – 754, que descreve o formato e representação desse tipo de números para computadores.

5.3.2.1 Padrão IEEE – 754

Os números em *FP* são representados segundo o padrão IEEE – 754 (Patterson, 2005), que regula sobre o formato e o uso desses números para computadores de 32 e 64 bits, embora o novo padrão IEEE – 754/2008 já determine as regras para 128 bits. Entretanto, como as instruções foram elaboradas com precisão simples, utilizou-se apenas a regra do padrão para números de 32 bits. Dos 32 bits que compõem uma palavra no MIPS (de 31 até 0), o bit 31 representa o sinal, os bits no intervalo de 30 até 23 representam o expoente, e os outros 23 bits (do 22 até 0) fazem parte da mantissa, ou fração. É importante explicar que o expoente sofre um deslocamento negativo de $(2^{e-1}) - 1$, onde e é a quantidade de bits usados para representar o expoente. Nesse caso, como e vale 8, o expoente é subtraído por 127. Isso acontece porque o expoente deve ser positivo, para ser possível gerar números grandes, ou negativo, para gerar números pequenos. Porém, a operação de comparação de dois números em *FP* acaba sendo difícil pelo fato dos números com sinal serem representados em complemento de dois. Para facilitar essa operação, é retirado o deslocamento do expoente, somando-o com 127. Isso faz com que ele seja representado sem sinal, permitindo que a comparação seja feita sem dificuldades.

Além da representação de valores positivos e negativos para o expoente, existem outros detalhes que devem ser levados em consideração. O bit mais significativo do número não é armazenado, mas quando for preciso operar com valores em *FP*, deve-se considerar o seguinte: quando o valor do expoente é maior que zero e menor que $2^e - 1$, sem considerar o deslocamento, diz-se que o número está normalizado, e o bit mais significativo (*Most Significant Bit – MSB*) é setado para 1 na hora da operação, mas quando o expoente é zero, e a mantissa é diferente de zero, então o número não está normalizado e o *MSB* é setado em 0. O valor zero é representado quando todos os bits do expoente e da mantissa forem zero, independente do bit de sinal ser zero ou um, ou seja, existem os valores –0 e +0. O infinito é representado quando todos os bits do

expoente forem 1, e o bit de sinal indica se o infinito é positivo ou negativo. Em qualquer outra situação, é assumido que a palavra não é um número. A figura 5.1 mostra o formato de números em *FP*.

Figura 5.1: Formato de números em *FP* de 32 bits pelo padrão IEEE – 754.

5.3.2.2 R3000 – Descrição da arquitetura

A primeira parte do projeto visou a expansão da arquitetura do MIPS R3000, já implementada em ArchC (2007), de forma que seja possível dar suporte às novas instruções. Nesta etapa, definiu-se o tamanho da palavra, de 32 bits, o tamanho da memória, de 64 Kbytes, o banco de registradores com 32 registradores de propósitos gerais, e mais dois, para armazenamento do resultado das operações de multiplicação e divisão inteiras, o banco de registradores para armazenamento de valores em *FP* com 10 registradores. Também foram definidos os cinco estágios do *pipeline*: IF, ID, EX, MEM e WB, os registradores delimitadores desses estágios: IF_ID, ID_EX, EX_MEM e MEM_WB, e a ordem de armazenamento das palavras na memória do tipo *big endian*. A figura 5.2 mostra a descrição arquitetural do R3000.

```
AC_ARCH(submips_extended_pipeline){
        ac_mem          DM:64K;
        ac_regbank      RB:34;
        ac_regbank      RB_float:10;
        ac_wordsize     32;
        ac_format F_IF_ID =     "%npc:32";
        ac_format F_ID_EX =     "%npc:32
                                 %data1:32 %data2:32
                                 %imm:32:s
                                 %fp_instr:1
                                 %rs:5 %rt:5 %rd:5
                                 %shamt:5
                                 %mult_write:1 %regwrite:1
                                 %memread:1 %memwrite:1
                                 %array1_1:32 %array1_2:32 %array1_3:32
                                 %array2_1:32 %array2_2:32 %array2_3:32
                                 %array_instr:1 %rotation:1";
        ac_format F_EX_MEM =    "%alures:32
                                 %wdata:32
                                 %rdest:5 %rdesthalf:5
                                 %fp_instr:1
                                 %mult_write:1 %regwrite:1
                                 %memread:1 %memwrite:1
                                 %array_resp1:32 %array_resp2:32 %array_resp3:32
                                 %array_instr:1 %rotation:1";
        ac_format F_MEM_WB =    "%wbdata:32
                                 %halfdata:32
                                 %rdest:5 %rdesthalf:5
                                 %fp_instr:1
                                 %mult_write:1 %regwrite:1
                                 %array_wb1:32 %array_wb2:32 %array_wb3:32
                                 %array_instr:1 %rotation:1";

        ac_reg<F_IF_ID> IF_ID;
        ac_reg<F_ID_EX> ID_EX;
        ac_reg<F_EX_MEM> EX_MEM;
        ac_reg<F_MEM_WB> MEM_WB;

        ac_stage IF,ID,EX,MEM,WB;

        ARCH_CTOR(submips_extended_pipeline){
                ac_isa("submips_extended_pipeline_isa.ac");
                set_endian("big");
        };
};
```

Figura 5.2: Descrição arquitetural do R3000.

Os dados do tipo *ac_format*: F_IF_ID, F_ID_EX, F_EX_MEM e F_MEM_WB, são os formatos dos registradores do *pipeline*. Como a figura 5.2 mostra, para criar um campo no registrador, é preciso dar um nome a ele, seguido do seu tamanho em bits. Por exemplo, na linha 19 dessa figura, o valor *%wdata:32* define campo *wdata* de 32 bits, na especificação de F_EX_MEM. Embora o primeiro registrador não tenha sofrido alterações, todos os outros foram modificados. O registrador IF_ID armazena a instrução recém buscada da memória. Já o registrador ID_EX foi estendido de forma a armazenar a leitura de múltiplos dados dos bancos de registradores, informar se a instrução executada naquele estágio é vetorial ou não, se haverá múltipla escrita nos bancos de registradores, e se a instrução é de rotação. Esse registrador possui tamanho de 379 bits. O registrador EX_MEM também foi estendido para armazenar o resultado de uma operação vetorial e manter salvas as informações de escrita no banco de registradores. Seu tamanho é de 177 bits. Por fim, o registrador MEM_WB possui quase os mesmos dados do registrador EX_MEM, mas seu tamanho é de 175 bits. O comportamento do processador para efetuar múltiplas leituras e escritas nos bancos de registradores é explicado mais adiante.

5.3.2.3 Definição dos mnemônicos e formado das instruções

Nesta etapa são definidos os tipos, formatos e mnemônicos das instruções. Os formatos das instruções originais do MIPS continuam as mesmas, ou seja, soma e subtração, junto com as demais operações aritméticas e lógicas, são do tipo – R. Operações aritméticas imediatas e desvios condicionais, são do tipo – I, e desvios incondicionais, do tipo – J. Como as novas instruções também têm o objetivo de realizar operações aritméticas, todas foram definidas como tipo – R. A figura 5.3 mostra a primeira parte do arquivo, contendo a definição dos tipos de instruções, a associação de cada mnemônico das instruções ao seu tipo, a identificação dos registradores dentro dos bancos de registradores, tanto o banco normal quando o de *FP*.

```
1  AC_ISA(submips_extended_pipeline){
2          ac_format Type_R = "%op:6 %rs:5 %rt:5 %rd:5 %shamt:5 %func:6";
3          ac_format Type_I = "%op:6 %rs:5 %rt:5 %imm:16:s";
4          ac_format Type_J = "%op:6 %addr:26";
5
6          //###### Definição das novas instruções ######
7          ac_instr<Type_R> sin, cos, trs, rtx, rty, rtz;
8          ac_instr<Type_R> adds, subs, muls, divs, movf;
9          //##############################################
10         ac_instr<Type_R> add, sub;
11         ac_instr<Type_R> mul, div, mfhi, mflo;
12
13         ac_instr<Type_I> addi, bne, lw, sw;
14         ac_instr<Type_I> itof;
15         ac_instr<Type_I> sll;
16         ac_instr<Type_I> nop;
17         ac_instr<Type_J> j, jal;
18         ac_instr<Type_R> jr, jalr;
19
20         ac_asm_map reg {
21                 "$"[0..31] = [0..31];
22                 "$zero" = 0;
23                 "$at" = 1;
24                 "$v"[0..1] = [2..3];
25                 "$a"[0..3] = [4..7];
26                 "$t"[0..7] = [8..15];
27                 "$s"[0..7] = [16..23];
28                 "$t"[8..9] = [24..25];
29                 "$kt"[0..1] = [26..27];
30                 "$gp" = 28;
31                 "$sp" = 29;
32                 "$fp" = 30;
33                 "$ra" = 31;
34         }
35         //##### O Banco de Registradores de Ponto Flutuante #####
36         ac_asm_map regf {
37                 "$f"[0..9] = [0..9];
38         }
```

Figura 5.3: Tipos e mnemônicos das instruções, e bancos de registradores do R3000.

Os mnemônicos *sin*, *cos*, *trs*, *rtx*, *rty* e *rtz* representam as instruções seno, cosseno, translação, rotação em X, rotação em Y e rotação em Z, respectivamente. As instruções de soma, subtração, multiplicação e divisão em *FP* de precisão simples e movimentação de valores em *FP* são descritos pelos mnemônicos *adds*, *subs*, *muls*, *divs* e *movf*, nessa ordem. É através desses, e dos outros mnemônicos, que se especificaram as novas instruções. Em seguida, informou-se a origem dos operandos, se vem do banco de registradores ou se o valor é imediato. Por fim, definiu-se a representação do endereço dos registradores de origem e destino. A figura 5.4 mostra essas especificações e completa, junto com a figura 5.3, a definição das novas instruções.

```
40          ISA_CTOR(submips_extended_pipeline){
41                  ...
42                  //soma em FP precisao simples
43                  adds.set_asm("adds %regf, %regf, %regf", rd, rs, rt);
44                  adds.set_decoder(op=0x00, func=0x06);
45
46                  //subtracao em FP precisao simples
47                  subs.set_asm("subs %regf, %regf, %regf", rd, rs, rt);
48                  subs.set_decoder(op=0x00, func=0x07);
49
50                  //multiplicacao em FP precisao simples
51                  muls.set_asm("muls %regf, %regf, %regf", rd, rs, rt);
52                  muls.set_decoder(op=0x00, func=0x08);
53
54                  //divisao em FP precisao simples
55                  divs.set_asm("divs %regf, %regf, %regf", rd, rs, rt);
56                  divs.set_decoder(op=0x00, func=0x09);
57
58                  //Converte um valor do tipo inteiro PF
59                  //Move o valor do Banco de Registradores Inteiro para PF
60                  itof.set_asm("itof %regf, %reg", rt, rs);
61                  itof.set_decoder(op=0x09);
62
63                  //mover um dado de um reg. PF pra outro reg. PF
64                  movf.set_asm("movf %regf, %regf", rt, rs);
65                  movf.set_decoder(op=0x00, func=0x0C);
66
67                  //Cálculo do seno
68                  sin.set_asm("sin %regf, %regf", rt, rs);
69                  sin.set_decoder(op=0x00, func=0x0D);
70
71                  //Cálculo do coseno
72                  cos.set_asm("cos %regf, %regf", rt, rs);
73                  cos.set_decoder(op=0x00, func=0x0E);
74
75                  //Cálculo da translação
76                  trs.set_asm("trs %regf, %regf, %regf", rd, rs, rt);
77                  trs.set decoder(op=0x00, func=0x02);
78
79                  //Cálculo da Rotação em X
80                  rtx.set_asm("rtx %regf, %regf, %regf", rd, rs, rt);
81                  rtx.set_decoder(op=0x00, func=0x03);
82
83                  //Cálculo da Rotação em Y
84                  rty.set_asm("rty %regf, %regf, %regf", rd, rs, rt);
85                  rty.set_decoder(op=0x00, func=0x04);
86
87                  //Cálculo da Rotação em Z
88                  rtz.set_asm("rtz %regf, %regf, %regf", rd, rs, rt);
89                  rtz.set_decoder(op=0x00, func=0x05);
90          };
91  };
```

Figura 5.4: Mnemônicos, operandos, *Opcode* e *Func* das novas instruções do R3000.

Por exemplo, a linha 43 define que o mnemônico *adds*, possui três valores. O primeiro valor informa que o resultado dessa operação será escrito no endereço *rd* do banco de registradores *FP*, o segundo e o terceiro diz que os operandos são lidos dos endereços *rs* e *rt* desse mesmo banco, respectivamente. Por fim, a linha 44 diz que o valor do *Opcode* é 00, e o valor de *Func* é 06, ambos em hexadecimal.

5.3.2.4 Definição do comportamento das novas instruções

A ultima etapa da definição do R3000 consiste em informar o comportamento das instruções, ou seja, a forma como elas realizam suas operações. A figuras 5.5 mostra o comportamento da instrução *adds* realizadas no estágio *EX* do pipeline. Também é interessante notar a propagação dos sinais de controle do registrador ID_EX para o registrador EX_MEM, da linha 1839 à linha 1846, simulando o funcionamento do processador real.

```
1818 //! Instruction adds behavior method.
1819 void ac_behavior(adds)
1820 {
1821     float op1, op2, resp;
1822
1823     switch (stage) {
1824     ...
1825     case EX:
1826         memcpy(&op1, &ex_value1, 4);
1827         memcpy(&op2, &ex_value2, 4);
1828
1829         //###### Comportamento da função "adds" ######
1830         resp = op1 + op2;
1831         //##########################################
1832
1833         memcpy(&EX_MEM.alures, &resp, 4);
1834         printf("ADDS: %2.1f + %2.1f = %2.1f\n",op1, op2, resp);
1835
1836         EX_MEM.fp_instr = ID_EX.fp_instr;
1837         EX_MEM.array_instr = ID_EX.array_instr;
1838         EX_MEM.rotation = ID_EX.rotation;
1839         EX_MEM.mult_write = ID_EX.mult_write;
1840         EX_MEM.regwrite = ID_EX.regwrite;
1841         EX_MEM.memread = ID_EX.memread;
1842         EX_MEM.memwrite = ID_EX.memwrite;
1843         EX_MEM.rdest = ID_EX.rd;
1844         break;
1845     ...
1846     }
1847 }
```

Figura 5.5: Comportamento da instrução *adds*.

Para simular o funcionamento, são usadas três variáveis auxiliares, que armazenam temporariamente o valor dos operandos e a resposta. Após o cálculo, a resposta é propagada para o registrador EX_MEM. De forma bastante semelhante, as figuras 5.6, 5.7 e 5.8 mostram o comportamento das instruções *subs*, *muls* e *divs*, respectivamente. A instrução *itof* é implementada de forma trivial. Ela busca um valor inteiro do banco de registradores de propósitos gerais, aplica um *typecast* de inteiro para *FP* no estágio EX, e armazena o novo valor no banco de registradores *FP*. Essa função é mostrada na figura 5.9. Por fim, a figura 5.10 mostra a instrução *movf*, que simplesmente repassa o valor do operando ao registrador EX_MEM.

```
1849 //! Instruction subs behavior method.
1850 void ac_behavior(subs)
1851 {
1852     float op1, op2, resp;
1853 
1854     switch (stage) {
1855     ...
1856     case EX:
1857         memcpy(&op1, &ex_value1, 4);
1858         memcpy(&op2, &ex_value2, 4);
1859 
1860         //###### Comportamento da função "subs" ######
1861         resp = op1 - op2;
1862         //##############################################
1863 
1864         memcpy(&EX_MEM.alures, &resp, 4);
1865         printf("SUBS: %2.1f - %2.1f = %2.1f\n",op1, op2, resp);
1866 
1867         EX_MEM.fp_instr = ID_EX.fp_instr;
1868         EX_MEM.array_instr = ID_EX.array_instr;
1869         EX_MEM.rotation = ID_EX.rotation;
1870         EX_MEM.mult_write = ID_EX.mult_write;
1871         EX_MEM.regwrite = ID_EX.regwrite;
1872         EX_MEM.memread = ID_EX.memread;
1873         EX_MEM.memwrite = ID_EX.memwrite;
1874         EX_MEM.rdest = ID_EX.rd;
1875         break;
1876     ...
1877     }
1878 }
```

Figura 5.6: Comportamento da instrução *subs*.

```
1880 //! Instruction muls behavior method.
1881 void ac_behavior(muls)
1882 {
1883     float op1, op2, resp;
1884 
1885     switch (stage) {
1886     ...
1887     case EX:
1888         memcpy(&op1, &ex_value1, 4);
1889         memcpy(&op2, &ex_value2, 4);
1890 
1891         //###### Comportamento da função "muls" ######
1892         resp = op1*op2;
1893         //##############################################
1894 
1895         memcpy(&EX_MEM.alures, &resp, 4);
1896         printf("MULS: %2.1f * %2.1f = %2.1f\n",op1, op2, resp);
1897 
1898         EX_MEM.fp_instr = ID_EX.fp_instr;
1899         EX_MEM.array_instr = ID_EX.array_instr;
1900         EX_MEM.rotation = ID_EX.rotation;
1901         EX_MEM.mult_write = ID_EX.mult_write;
1902         EX_MEM.regwrite = ID_EX.regwrite;
1903         EX_MEM.memread = ID_EX.memread;
1904         EX_MEM.memwrite = ID_EX.memwrite;
1905         EX_MEM.rdest = ID_EX.rd;
1906         break;
1907     ...
1908     }
1909 }
```

Figura 5.7: Comportamento da instrução *muls*.

```
1911 //! Instruction divs behavior method.
1912 void ac_behavior(divs)
1913 {
1914     float op1, op2, resp;
1915 
1916     switch (stage) {
1917     ...
1918     case EX:
1919         memcpy(&op1, &ex_value1, 4);
1920         memcpy(&op2, &ex_value2, 4);
1921 
1922         //###### Comportamento da função "divs" ######
1923         resp = op1/op2;
1924         //##############################################
1925 
1926         memcpy(&EX_MEM.alures, &resp, 4);
1927         printf("DIVS: %2.1f / %2.1f = %2.1f\n",op1, op2, resp);
1928 
1929         EX_MEM.fp_instr = ID_EX.fp_instr;
1930         EX_MEM.array_instr = ID_EX.array_instr;
1931         EX_MEM.rotation = ID_EX.rotation;
1932         EX_MEM.mult_write = ID_EX.mult_write;
1933         EX_MEM.regwrite = ID_EX.regwrite;
1934         EX_MEM.memread = ID_EX.memread;
1935         EX_MEM.memwrite = ID_EX.memwrite;
1936         EX_MEM.rdest = ID_EX.rd;
1937         break;
1938     ...
1939     }
1940 }
```

Figura 5.8: Comportamento da instrução *divs*.

```
1942 //! Instruction itof behavior method.
1943 void ac_behavior(itof)
1944 {
1945     float temp;
1946 
1947     switch (stage) {
1948     ...
1949     case EX:
1950         //###### Comportamento da função "itof" ######
1951         temp = (float) ex_value1;
1952         //##############################################
1953 
1954         memcpy(&EX_MEM.alures, &temp, 4);
1955         printf("ITOF: %x to %2.0f\n",ex_value1, temp);
1956 
1957         EX_MEM.fp_instr = ID_EX.fp_instr;
1958         EX_MEM.array_instr = ID_EX.array_instr;
1959         EX_MEM.rotation = ID_EX.rotation;
1960         EX_MEM.mult_write = ID_EX.mult_write;
1961         EX_MEM.regwrite = ID_EX.regwrite;
1962         EX_MEM.memread = ID_EX.memread;
1963         EX_MEM.memwrite = ID_EX.memwrite;
1964         EX_MEM.rdest = ID_EX.rt;
1965         break;
1966     ...
1967     }
1968 }
```

Figura 5.9: Comportamento da instrução *itof*.

```
2011 //! Instruction itof behavior method.
2012 void ac_behavior(movf)
2013 {
2014     switch (stage) {
2015     ...
2016     case EX:
2017         //###### Comportamento da função "movf" ######
2018         memcpy(&EX_MEM.alures,&ex_value1,4);
2019         //##########################################
2020 
2021         EX_MEM.fp_instr = ID_EX.fp_instr;
2022         EX_MEM.array_instr = ID_EX.array_instr;
2023         EX_MEM.rotation = ID_EX.rotation;
2024         EX_MEM.mult_write = ID_EX.mult_write;
2025         EX_MEM.regwrite = ID_EX.regwrite;
2026         EX_MEM.memread = ID_EX.memread;
2027         EX_MEM.memwrite = ID_EX.memwrite;
2028         EX_MEM.rdest = ID_EX.rt;
2029         break;
2030     ...
2031     }
2032 }
```

Figura 5.10: Comportamento da instrução *movf*.

Os comportamentos das instruções anteriores finalizam a definição do *ISA* do R3000 original. A partir de agora, serão definidos os comportamentos das instruções vetoriais e matriciais propostas na seção 5.2.

Para implementar o comportamento das instruções seno e cosseno, utilizou-se a biblioteca *math.h* disponível nos compiladores C e C++. Assim, basta chamar essas funções para obter a resposta dessas instruções. De forma semelhante às operações anteriores, criaram-se duas variáveis auxiliares para armazenar o valor de entrada e saída dessas funções. O valor da entrada é dado em radianos, e a resposta armazena o seno ou cosseno desse valor. Por fim, depois dos cálculos, a resposta é passada para o registrador *pipeline* EX_MEM. As figuras 5.11 e 5.12 mostram o comportamento das instruções *sin* e *cos*.

```
2314 //! Instruction sin behavior method.
2315 void ac_behavior(sin)
2316 {
2317     float temp1, temp2;
2318 
2319     switch (stage) {
2320     ...
2321     case EX:
2322         memcpy(&temp1,&ex_value1,4);
2323         temp2 = sin(temp1);
2324         memcpy(&EX_MEM.alures,&temp2,4);
2325         printf("SIN: ex_value1 = %x, resp = %4.3f\n",ex_value1,temp2);
2326 
2327         EX_MEM.fp_instr = ID_EX.fp_instr;
2328         EX_MEM.array_instr = ID_EX.array_instr;
2329         EX_MEM.rotation = ID_EX.rotation;
2330         EX_MEM.mult_write = ID_EX.mult_write;
2331         EX_MEM.regwrite = ID_EX.regwrite;
2332         EX_MEM.memread = ID_EX.memread;
2333         EX_MEM.memwrite = ID_EX.memwrite;
2334         EX_MEM.rdest = ID_EX.rt;
2335         break;
2336     ...
2337     }
2338 };
```

Figura 5.11: Comportamento da instrução *sin*.

```
2340 //! Instruction cos behavior method.
2341 void ac_behavior(cos)
2342 {
2343     float temp1, temp2;
2344
2345     switch (stage) {
2346     ...
2347     case EX:
2348         memcpy(&temp1,&ex_value1,4);
2349         temp2 = cos(temp1);
2350         memcpy(&EX_MEM.alures,&temp2,4);
2351         printf("COS: ex_value1 = %x, resp = %4.3f\n",ex_value1,temp2);
2352
2353         EX_MEM.fp_instr = ID_EX.fp_instr;
2354         EX_MEM.array_instr = ID_EX.array_instr;
2355         EX_MEM.rotation = ID_EX.rotation;
2356         EX_MEM.mult_write = ID_EX.mult_write;
2357         EX_MEM.regwrite = ID_EX.regwrite;
2358         EX_MEM.memread = ID_EX.memread;
2359         EX_MEM.memwrite = ID_EX.memwrite;
2360         EX_MEM.rdest = ID_EX.rt;
2361         break;
2362     ...
2363     }
2364 };
```

Figura 5.12: Comportamento da instrução *cos*.

As descrições de *trs*, *rtx*, *rty* e *rtz* são mais complexas porque efetuam cálculos com múltiplos dados. Logo, é necessário alterar o comportamento padrão de leitura, escrita e execução de cálculos da arquitetura MIPS. Para representar um ponto no sistema de coordenadas $\boldsymbol{R^3}$, precisa-se do valor da coordenada X, da coordenada Y e da coordenada Z. Para ler e escrever esses valores do banco de registradores, assume-se que elas estarão guardadas em três registradores consecutivos, que guardarão os valores de X, Y e Z, nessa ordem. Esses três valores são lidos ou escritos ao mesmo tempo. Assim, se o registrador *rs* possuir a coordenada X, então o registrador *rs* + 1 tem a coordenada Y, e o *rs* + 2, a coordenada Z. Por exemplo: deseja-se executar a instrução *trs rd rs rt*, onde *rd* = 10, *rs* = 15 e *rt* = 20. As novas coordenadas X, Y e Z são obtidas pela soma das coordenadas X, Y e Z do ponto inicial, armazenadas nos registrador 15, 16 e 17, com as coordenada X, Y e Z do vetor de deslocamento, armazenadas em 20, 21 e 22. Os resultados são salvos nos registradores 10, 11 e 12. A operação *trs* é vista na figura 5.13.

```
2424  //! Instruction trs behavior method
2425  void ac_behavior(trs)
2426  {
2427      /* ponto = [op0 op1 op2]
2428       *
2429       * matriz de translacao
2430       * |1 0 0 x|
2431       * |0 1 0 y|
2432       * |0 0 1 z|
2433       * |0 0 0 1|
2434      **/
2435
2436      float op0, op1, op2;
2437      float op3, op4, op5;
2438      float op6, op7, op8;
2439
2440      switch (stage) {
2441      ...
2442      case EX:
2443          memcpy(&op0,&array_ex_value1[0],4);
2444          memcpy(&op1,&array_ex_value1[1],4);
2445          memcpy(&op2,&array_ex_value1[2],4);
2446
2447          memcpy(&op3,&array_ex_value2[0],4);
2448          memcpy(&op4,&array_ex_value2[1],4);
2449          memcpy(&op5,&array_ex_value2[2],4);
2450
2451          op6 = op0 + op3; //novoP1 = ponto[op0] + matriz[x]
2452          op7 = op1 + op4; //novoP2 = ponto[op1] + matriz[y]
2453          op8 = op2 + op5; //novoP3 = ponto[op2] + matriz[z]
2454
2455          memcpy(&EX_MEM.array_resp1,&op6,4);
2456          memcpy(&EX_MEM.array_resp2,&op7,4);
2457          memcpy(&EX_MEM.array_resp3,&op8,4);
2458
2459          printf("TRS: resp = %4.3f, %4.3f, %4.3f\n",op6,op7,op8);
2460
2461          EX_MEM.fp_instr = ID_EX.fp_instr;
2462          EX_MEM.array_instr = ID_EX.array_instr;
2463          EX_MEM.rotation = ID_EX.rotation;
2464          EX_MEM.mult_write = ID_EX.mult_write;
2465          EX_MEM.regwrite = ID_EX.regwrite;
2466          EX_MEM.memread = ID_EX.memread;
2467          EX_MEM.memwrite = ID_EX.memwrite;
2468          EX_MEM.rdest = ID_EX.rd;
2469          break;
2470      ...
2471      }
2472  };
```

Figura 5.13: Comportamento da instrução *trs*.

Embora as operações de rotação também necessitem de múltiplas leituras e escritas do banco de registradores, elas diferem da instrução *trs*. Neste novo ambiente, o registrador *rd* é usado como origem e destino ao mesmo tempo, ou seja, as coordenadas do ponto inicial X, Y e Z são lidas de *rd*, *rd* + 1 e *rd* + 2, e o novo ponto é guardado nesses mesmos registradores. Os valores do seno e cosseno, necessários às instruções de rotação, não estão em registradores consecutivos necessariamente, então eles são lidos dos registradores *rs* e *rt*, respectivamente. O comportamento das instruções rtx, rty e rtz pode ser visto nas figuras 5.14, 5.15 e 5.16.

```
2474 //! Instruction rtx behavior method
2475 void ac_behavior(rtx)
2476 {
2477     /* Ponto = [temp0 temp1 temp2]
2478      *
2479      * | 1    0    0   0|    |temp0| :Matriz de Transformacao
2480      * | 0   cos  -sen 0| x  |temp1| :do ponto em torno do eixo X
2481      * | 0   sen  cos  0|    |temp2|
2482      * | 0    0    0   1|    |  1  |
2483     **/
2484
2485     float op0, op1, op2;        //coordenadas do ponto a ser transformado
2486     float op3, op4;             //op3 = seno, op4 = cosseno
2487     float temp0, temp1, temp2;  //guardam temporariamente o ponto
2488
2489     switch (stage) {
2490     ...
2491     case EX:
2492         memcpy(&temp0,&array_ex_value1[0],4);
2493         memcpy(&temp1,&array_ex_value1[1],4);
2494         memcpy(&temp2,&array_ex_value1[2],4);
2495
2496         memcpy(&op3,&ex_value1,4);
2497         memcpy(&op4,&ex_value2,4);
2498
2499         printf("ponto = %4.3f, %4.3f, %4.3f\n",temp0, temp1, temp2);
2500         printf("seno = %4.3f, cosseno = %4.3f\n",op3, op4);
2501
2502         op0 = temp0;                //op0 <- ponto[temp0]
2503         op1 = temp1*op4 - temp1*op3; //op1 <- ponto[temp1]*cosseno - ponto[temp1]*seno
2504         op2 = temp2*op3 + temp2*op4; //op2 <- ponto[temp2]*seno + ponto[temp2]*cosseno
2505
2506         memcpy(&EX_MEM.array_resp1,&op0,4);
2507         memcpy(&EX_MEM.array_resp2,&op1,4);
2508         memcpy(&EX_MEM.array_resp3,&op2,4);
2509
2510         printf("RTX: resp = %4.3f, %4.3f, %4.3f\n",op0,op1,op2);
2511
2512         EX_MEM.fp_instr = ID_EX.fp_instr;
2513         EX_MEM.array_instr = ID_EX.array_instr;
2514         EX_MEM.rotation = ID_EX.rotation;
2515         EX_MEM.mult_write = ID_EX.mult_write;
2516         EX_MEM.regwrite = ID_EX.regwrite;
2517         EX_MEM.memread = ID_EX.memread;
2518         EX_MEM.memwrite = ID_EX.memwrite;
2519         EX_MEM.rdest = ID_EX.rd;
2520         break;
2521     ...
2522     }
2523 };
```

Figura 5.14: Comportamento da instrução *rtx*.

```
2525 //! Instruction rty behavior method
2526 void ac_behavior(rty)
2527 {
2528     /* Ponto = [temp0 temp1 temp2]
2529      *
2530      * | cos  0  sen  0|    |temp0| :Matriz de Transformacao
2531      * |  0   1   0   0| x  |temp1|  do ponto em torno do eixo Y
2532      * | -sen 0  cos  0|    |temp2|
2533      * |  0   0   0   1|    |  1  |
2534     **/
2535
2536     float op0, op1, op2;        //coordenadas do ponto a ser transformado
2537     float op3, op4;             //op3 = seno, op4 = cosseno
2538     float temp0, temp1, temp2;  //guardam temporariamente o ponto
2539
2540     switch (stage) {
2541     ...
2542     case EX:
2543         memcpy(&temp0,&array_ex_value1[0],4);
2544         memcpy(&temp1,&array_ex_value1[1],4);
2545         memcpy(&temp2,&array_ex_value1[2],4);
2546
2547         memcpy(&op3,&ex_value1,4);
2548         memcpy(&op4,&ex_value2,4);
2549
2550         printf("ponto = %4.3f, %4.3f, %4.3f\n",temp0, temp1, temp2);
2551         printf("seno = %4.3f, cosseno = %4.3f\n",op3, op4);
2552
2553         op0 = temp0*op4 + temp0*op3; //novoP1 <- ponto[temp0]*cosseno + ponto[temp2]*seno
2554         op1 = temp1;                 //novoP2 <- ponto[temp1]
2555         op2 = temp2*op4 - temp2*op3; //novoP1 <- ponto[temp2]*cosseno - ponto[temp0]*seno
2556
2557         memcpy(&EX_MEM.array_resp1,&op0,4);
2558         memcpy(&EX_MEM.array_resp2,&op1,4);
2559         memcpy(&EX_MEM.array_resp3,&op2,4);
2560
2561         printf("RTY: resp = %4.3f, %4.3f, %4.3f\n",op0,op1,op2);
2562
2563         EX_MEM.fp_instr = ID_EX.fp_instr;
2564         EX_MEM.array_instr = ID_EX.array_instr;
2565         EX_MEM.rotation = ID_EX.rotation;
2566         EX_MEM.mult_write = ID_EX.mult_write;
2567         EX_MEM.regwrite = ID_EX.regwrite;
2568         EX_MEM.memread = ID_EX.memread;
2569         EX_MEM.memwrite = ID_EX.memwrite;
2570         EX_MEM.rdest = ID_EX.rd;
2571         break;
2572     ...
2573     }
2574 };
```

Figura 5.15: Comportamento da instrução *rty*.

```
2576 //! Instruction rtz behavior method
2577 void ac_behavior(rtz)
2578 {
2579     /* Ponto = [temp0 temp1 temp2]
2580      *
2581      * | cos -sen  0  0|    |temp0| :Matriz de Transformacao
2582      * | sen  cos  0  0| x  |temp1| :do ponto em torno do eixo Z
2583      * |  0    0   1  0|    |temp2|
2584      * |  0    0   0  1|    |  1  | :Operacao que sera realizada...
2585     **/
2586
2587     float op0, op1, op2;        //coordenadas do ponto a ser transformado
2588     float op3, op4;             //op3 = seno, op4 = cosseno
2589     float temp0, temp1, temp2;  //guardam temporariamente o ponto
2590
2591     switch (stage) {
2592     ...
2593     case EX:
2594         memcpy(&temp0,&array_ex_value1[0],4);
2595         memcpy(&temp1,&array_ex_value1[1],4);
2596         memcpy(&temp2,&array_ex_value1[2],4);
2597
2598         memcpy(&op3,&ex_value1,4);
2599         memcpy(&op4,&ex_value2,4);
2600
2601         printf("ponto = %4.3f, %4.3f, %4.3f\n",temp0, temp1, temp2);
2602         printf("seno = %4.3f, cosseno = %4.3f\n",op3, op4);
2603
2604         op0 = temp0*op4 - temp0*op3; //novoP1 <- ponto[temp0]*cosseno - ponto[temp1]*seno
2605         op1 = temp1*op3 + temp1*op4; //novoP2 <- ponto[temp0]*seno + ponto[temp1]*cosseno
2606         op2 = temp2;                 //novoP1 <- ponto[temp2]
2607
2608         memcpy(&EX_MEM.array_resp1,&op0,4);
2609         memcpy(&EX_MEM.array_resp2,&op1,4);
2610         memcpy(&EX_MEM.array_resp3,&op2,4);
2611
2612         printf("RTZ: resp = %4.3f, %4.3f, %4.3f\n",op0,op1,op2);
2613
2614         EX_MEM.fp_instr = ID_EX.fp_instr;
2615         EX_MEM.array_instr = ID_EX.array_instr;
2616         EX_MEM.rotation = ID_EX.rotation;
2617         EX_MEM.mult_write = ID_EX.mult_write;
2618         EX_MEM.regwrite = ID_EX.regwrite;
2619         EX_MEM.memread = ID_EX.memread;
2620         EX_MEM.memwrite = ID_EX.memwrite;
2621         EX_MEM.rdest = ID_EX.rd;
2622         break;
2623     ...
2624     }
2625 };
```

Figura 5.16: Comportamento da instrução *rtz*.

O objetivo desse capítulo não é desenvolver a estrutura física do R3000. No entanto, para ter uma visão mais clara das alterações que devem ser feitas na sua organização, de forma a dar suporte às novas instruções, a figura 5.17 mostra parte das modificações necessárias no caminho de dados deste processador. Tais modificações são vistas no capítulo 7.

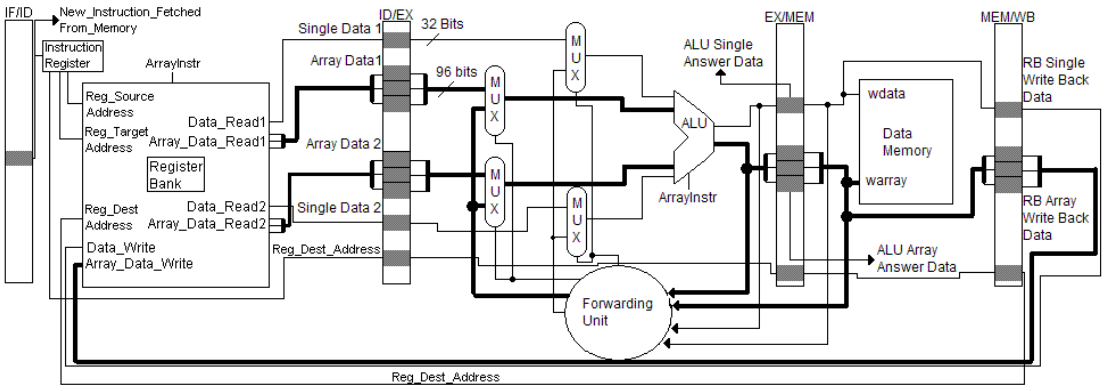

Figura 5.17: Modificações do caminho de dados do MIPS R3000.

6 VALIDAÇÃO DO NOVO CONJUNTO DE INSTRUÇÕES

6.1 Introdução

Uma vez que a implementação do novo conjunto de instruções foi finalizada, é necessário compará-lo ao *ISA* original para verificar a eficiência de um em relação ao outro. O objetivo é que o novo *ISA* possua maior desempenho de velocidade na execução de tarefas. Isso é medido através de um ambiente de simulação no qual é executada uma aplicação. Na simulação mede-se o tempo que cada *ISA* gastou na execução da aplicação, a quantidade de instruções realizadas por cada uma, a quantidade de acessos à memória, e aos bancos de registradores. É importante lembrar que o *ArchC* é uma linguagem de descrição de arquiteturas do tipo *TLM* e não leva em consideração os custos físicos oriundos à implementação do *ISA*, ou seja, os resultados obtidos nesse capítulo não são definitivos e podem mudar.

6.2 Aplicação

Para obter as informações de desempenho do novo *ISA*, foi desenvolvida uma aplicação que simula o comportamento de um robô esférico. Como esse tipo de robô é empregado nas linhas de montagem automotiva, é comum que ele estique e rotacione seus eixos para encaixar uma nova peça no automóvel. Nessa aplicação, o braço do robô iniciará no ponto com coordenadas (3, 4, 5) do espaço $\boldsymbol{R^3}$ e aplicará a instrução de translação para esticá-lo três unidades na direção dos três eixos cartesianos. Em seguida, o braço será girado por um valor de 2 radianos sobre o eixo X, e finalmente rotacionará 3 radianos em torno do eixo Z.

Como o *ISA* original não possui as instruções de seno e cosseno, a responsabilidade de calculá-las é delegada à aplicação. Esse cálculo é bastante extenso porque efetua a expansão em séries de Taylor e McLaurin (Anton, 2007) para as duas funções. As figuras 6.1 (A) e 6.1 (B) mostram os somatórios das respectivas séries. Somado a isso, os cálculos de translação e rotação também são implementados na aplicação. Logo, o tamanho da aplicação do *ISA* original é significativamente maior que a aplicação do novo *ISA*.

Uma vez que não existe um compilador que gere código para essas arquiteturas, foi preciso desenvolver uma aplicação para o *ISA* original e outra para o novo *ISA*. No original, os cálculos do seno, cosseno, translação e rotações são feitos na própria aplicação, enquanto que o *ISA* novo aplica as novas instruçõe para realizar as mesmas operações. As figuras 6.2 e 6.3 mostram a aplicação para o R3000 original e para o novo, respectivamente.

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} \quad \text{(A)} \qquad \cos x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} \quad \text{(B)}$$

Figura 6.1 (A): Série do Seno. (B): Série do Cosseno (Adaptado de Anton, 2007).

```
main:
        addi $t0, $0, 3 #
        itof $f0, $t0   #
        addi $t0, $0, 4 #
        itof $f1, $t0   #
        addi $t0, $0, 5 #
        itof $f2, $t0   # seta o ponto inicial do robo em [3 4 5]
translacao:
        addi $t0, $0, 3
        itof $f3, $t0           #px_matriz
        addi $t0, $0, 3
        itof $f4, $t0           #py_matriz
        addi $t0, $0, 3
        itof $f5, $t0           #pz_matriz
        adds $f7, $f0, $f3      #
        adds $f8, $f1, $f4      #
        adds $f9, $f2, $f5      #efetua a translacao
fim_translacao:
        #pontos modificados estao em $f7, $f8, $f9 (px, py, pz)
rotacaoX:
        addi $t0, $0, 2 #passa o angulo por parametro para a funcao seno
        itof $f0, $t0
        jal seno
        nop
        movf $f6, $f3   #salva o seno temporariamente
        addi $t0, $0, 2 #passa o angulo por parametro para a funcao cosseno
        itof $f0, $t0
        jal cosseno
        nop
        movf $f1, $f3
        movf $f0, $f6   #guarda o seno no lugar correto
                        #Seno esta em $f0 e Cosseno, em $f1
        movf $f2, $f7           #"calcula" o npx

        muls $f6, $f8, $f1      # aux = $f8 * $f1 -> aux = py*cos
        movf $f3, $f6           # $f3 = aux
        muls $f6, $f8, $f0      # aux = $f8 * $f0 -> aux = py*sen
        subs $f3, $f3, $f6      # $f3 = $f6 - aux # npy = py*cos - py*sen

        muls $f6, $f9, $f0      # aux = $f9 * $f0 -> aux = pz*sen
        movf $f4, $f6           # $f4 = aux
        muls $f6, $f9, $f1      # aux = $f9 * $f1 -> aux = pz*cos
        adds $f4, $f4, $f6      # $f4 = $f7 + aux -> npz = pz*sen + pz*cos
fim_rotx:                       # pontos estao em $f2, $f3, $f4
rotacaoZ:
        movf $f7, $f2
        movf $f8, $f3
        movf $f9, $f4

        addi $t0, $0, 3 #angulo
        itof $f0, $t0
        jal seno
        nop
        movf $f6, $f3   #salva o seno temporariamente
        addi $t0, $0, 3 #passa o angulo por parametro para a funcao cosseno
        itof $f0, $t0
        jal cosseno
        nop
        movf $f1, $f3
        movf $f0, $f6   #restaura o seno no lugar correto

        muls $f6, $f7, $f1      # aux = $f7 * $f1 -> aux = px*cos
        movf $f2, $f6           # $f5 = aux
        muls $f6, $f7, $f0      # aux = $f7 * $f0 -> aux = px*sen
        subs $f2, $f2, $f6      # $f5 = $f5 - aux -> npx = px*cos - px*sen

        muls $f6, $f8, $f0      # aux = $f8 * $f0 -> aux = py*sen
        movf $f3, $f6           # $f6 = aux
        muls $f6, $f8, $f1      # aux = $f8 * $f1 -> aux = py*cos
        adds $f3, $f3, $f6      # $f6 = $f6 + aux -> npy = py*sen + py*cos
        movf $f4, $f9           # npz = pz
fim_rotacaoZ:
        j fim_aplicacao
        nop

```

Figura 6.2 (A): Aplicação para o R3000 original.

```
seno:
sin_main:
        itof $f3, $0                # Soma = 0
#       $f0 <- angulo (angulo = X)  # $f0 = x. A variavel cujo seno sera calculado.
#       addi $t0, $0, 4             # $s0 = n. O iterador do somatório
#       itof $f0, $t0               # $s1 = i. Iterador auxiliar para calcular o numerador,
                                    #          ou seja, será usado p/calcular
                                    #          ((-1)^i)*(x^(2*i+1))/((2*i+1)!),
#       $s2 <- tam_expansao         #          o n-ésimo termo da expansão.
        addi $s2, $0, 8             # $f1 = numerador. O n-ésimo termo da expansao.
                                    # $s2 = tam_expansao. O tamanho da expansão, ou seja,
                                    #                     informa quantos termos será
                                    #                     gerado na expansao.

        addi $s0, $0, 0             #for (n=0; n<=tam_expansao; n++)
sin_loop:
        bne $s0, $s2, sin_iteracao
        nop
        j sin_fim
        nop
sin_iteracao:
        addi $t2, $0, 1
        itof $f1, $t2               # numerador <- 1
        addi $s1, $s0, 0            # i <- n
        sll $t1, $s1, 1             # $t1 <- 2*i+1 (valor cujo angulo sera elevado à potencia
        addi $t1, $t1, 1            #              (e apos dividido pelo fatorial)

        addi $t0, $0, 0             # for (cont_pow=0; cont_pow<= 2*i+1; cont_pow++)
sin_calc_pow:                       #     numerador *= numerador;
        bne $t1, $t0, sin_it_calc_pow
        nop
        j sin_fim_calc_pow
        nop
sin_it_calc_pow:
        muls $f1, $f1, $f0          #
        addi $t0, $t0, 1            #
        j sin_calc_pow              #
        nop
sin_fim_calc_pow:                   # Verifica se nao faz divisao por 0
        bne $t1, $0, sin_init_div_num_fat # (na primeira iteracao, 2*i = 0)
        nop
        j sin_fim_div_num_fat
        nop
sin_init_div_num_fat:
        addi $t2, $0, 1             #for (fat=2*i; fat>1; fat--)
sin_div_num_fat:                    #  numerador /= fat
        bne $t1, $t2, sin_it_div_num_fat
        nop
        j sin_fim_div_num_fat       #
        nop
sin_it_div_num_fat:
        itof $f2, $t1               #
        divs $f1, $f1, $f2          # !! Divisao por 0 na primeira iteracao,
        sub $t1, $t1, $t2           # !! vai entrar em loop infinito: $t1 <- -1
        j sin_div_num_fat           #
        nop
sin_fim_div_num_fat:                # verifica o sinal do n-esimo termo da serie:
        addi $t0, $0, 2             # i.e. (-1)^i.
        div $s1, $t0                # if (i%2==0) //sinal positivo
        mfhi $t1                    #   soma = soma + numerador
        bne $t1, $0, sin_sub_sum    # else
        nop
        adds $f3, $f3, $f1          #   soma = soma - numerador
        j sin_fim_iteracao
        nop
sin_sub_sum:
        subs $f3, $f3, $f1

sin_fim_iteracao:
        addi $s0, $s0, 1
        j sin_loop
        nop
sin_fim:
        jr $ra
        nop
```

Figura 6.2 (B): Expansão em série de Taylor e McLauren da função Seno.

```
152 cosseno:
153 cos_main:
154         itof $f3, $0                # Soma = 0
155 #       $f0 <- angulo (angulo = X)  # $f0 = x -> A variavel cujo cosseno sera calculado.
156 #       addi $t0, $0, 4             # $s0 = n -> O iterador do somatório
157 #       itof $f0, $t0               # $s1 = i -> Iterador auxiliar no calculo do numerador,
158                                     #            ou seja, será usado p/calcular
159                                     #            ((-1)^i)*(x^(2*i))/((2*i)!)
160                                     #            que é o n-ésimo termo da expansão.
161                                     #
162 #       $s2 <- tam_expansao         #
163         addi $s2, $0, 8             # $f1 = numerador ->  O n-ésimo termo da expansao.
164                                     # $s2 = tam_expansao. O tamanho da expansão, ou seja,
165                                     #                     informa quantos termos serão
166                                     #                     gerados na expansao.
167         addi $s0, $0, 0             # for (n=0; n<=tam_expansao; n++)
168 cos_loop:                           #
169         bne $s0, $s2, cos_iteracao  #
170         nop
171         j cos_fim
172         nop
173 cos_iteracao:
174         addi $t2, $0, 1
175         itof $f1, $t2               # numerador <- 1
176         addi $s1, $s0, 0            # i <- n
177         sll $t1, $s1, 1             # $t1 <- 2*i (o valor cujo angulo sera elevado à potencia
178                                     #           (e apos dividido pelo fatorial)
179
180         addi $t0, $0, 0             # for (cont_pow=0; cont_pow<= 2*i; cont_pow++)
181 cos_calc_pow:                       #     numerador *= numerador;
182         bne $t1, $t0, cos_it_calc_pow
183         nop
184         j cos_fim_calc_pow
185         nop
186 cos_it_calc_pow:                    #
187         muls $f1, $f1, $f0          #
188         addi $t0, $t0, 1            #
189         j cos_calc_pow              #
190         nop
191 cos_fim_calc_pow:                   # Verifica se nao faz divisao por 0
192         bne $t1, $0, cos_init_div_num_fat # (na primeira iteracao, 2*i = 0)
193         nop
194         j cos_fim_div_num_fat
195         nop
196 cos_init_div_num_fat:
197         addi $t2, $0, 1             # for (fat=2*i; fat>1; fat--)
198 cos_div_num_fat:                    #     numerador /= fat
199         bne $t1, $t2, cos_it_div_num_fat
200         nop
201         j cos_fim_div_num_fat       #
202         nop
203 cos_it_div_num_fat:                 #
204         itof $f2, $t1               #
205         divs $f1, $f1, $f2          # !! Se for divisao por 0 na primeira iteracao,
206         sub $t1, $t1, $t2           # !! vai entrar em loop infinito: $t1 <- -1
207         j cos_div_num_fat           #
208         nop
209 cos_fim_div_num_fat:                # verifica o sinal do n-esimo termo da serie:
210         addi $t0, $0, 2             # i.e. (-1)^i.
211         div $s1, $t0                # if (i%2==0) //sinal positivo
212         mfhi $t1                    #   soma = soma + numerador
213         bne $t1, $0, cos_sub_sum    # else
214         nop                         #   soma = soma - numerador
215         adds $f3, $f3, $f1          #
216         j cos_fim_iteracao
217         nop
218 cos_sub_sum:
219         subs $f3, $f3, $f1
220
221 cos_fim_iteracao:
222         addi $s0, $s0, 1
223         j cos_loop
224         nop
225 cos_fim:
226         jr $ra
227         nop
228
229 fim_aplicacao:
230         nop
```

Figura 6.2 (C): Expansão em série de Taylor e McLauren da função Coseno.

```
1  main:
2          addi $t0, $0, 3 #
3          itof $f0, $t0   #
4          addi $t0, $0, 4 #
5          itof $f1, $t0   #
6          addi $t0, $0, 5 #
7          itof $f2, $t0   # ponto = [3 4 5]
8
9  translacao:
10         addi $t0, $0, 3 #
11         itof $f3, $t0   #
12         addi $t0, $0, 3 #
13         itof $f4, $t0   #
14         addi $t0, $0, 3 #
15         itof $f5, $t0   # matriz deslocamento [3 3 3]
16         trs  $f7, $f0, $f3
17 fim_translacao:
18
19 rotacaoX:
20         addi $t0, $0, 2 #passa o angulo por parametro para a funcao seno
21         itof $f2, $t0
22         sin  $f0, $f2
23         cos  $f1, $f2
24         rtx  $f7, $f0, $f1
25 fim_rotacaoX:
26
27 rotacaoZ:
28         addi $t0, $0, 3 #passa o angulo por parametro para a funcao seno
29         itof $f2, $t0
30         sin  $f0, $f2
31         cos  $f1, $f2
32         rtz  $f7, $f0, $f1
33 fim_rotacaoZ:
34         nop
```

Figura 6.3: Aplicação para o novo R3000.

6.3 Simulação e Resultados

Para gerar os resultados de desempenho do novo conjunto de instruções, primeiramente executou-se a aplicação do movimento robótico do *ISA* original, e em seguida rodou-se a aplicação do *ISA* estendido. A simulação do *ArchC* não considera as alterações físicas do processador, ou seja, resultados como aumento de área, consumo de potência e a nova velocidade do *clock* decorrente das alterações são ignoradas. Nesse ambiente, o simulador determina que o processador executará sua aplicação com velocidade de 50MHz, independente da complexidade das operações que ele realiza. Sabe-se que a frequência do processador é determinada pela operação de maior latência. Por exemplo, se a instrução mais custosa de um processador for multiplicação em ponto flutuante, que leva hipoteticamente 50ns para ser executada, então a frequência dele será de 20MHz. Porém, se nesse mesmo processador for incluída a função seno, com *overhead* de 100ns, então sua nova velocidade será de 10MHz. Essa regra é ignorada na simulação do *ArchC*, logo, por maior que seja a latência da instrução mais complexa, a velocidade do processador é estipulada sempre em 50MHz. Abaixo, seguem os resultados obtidos na simulação. As figuras 6.4, 6.5, 6.6 e 6.7 mostram o tempo de execução do R3000 original e do R3000 novo, o total de instruções realizadas por cada um, a quantidade de acessos à memória e aos bancos de registradores de inteiros e de FP.

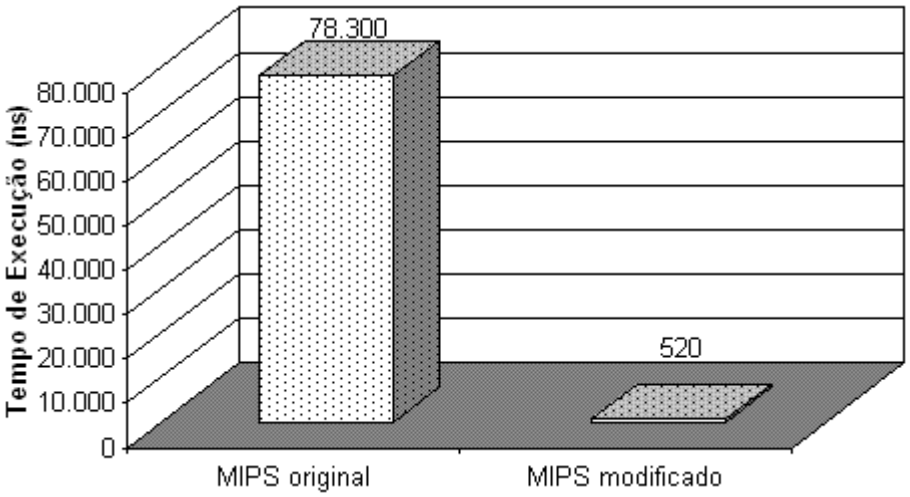

Figura 6.4: Tempo de execução.

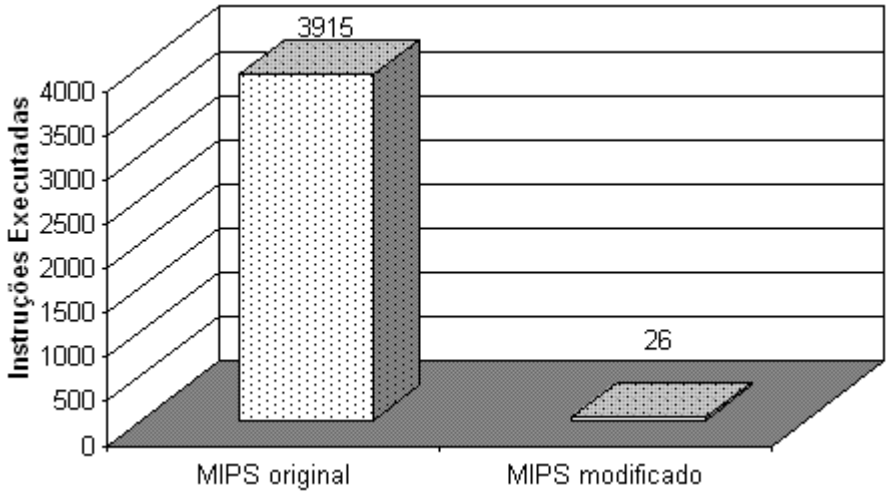

Figura 6.5: Total de instruções executadas.

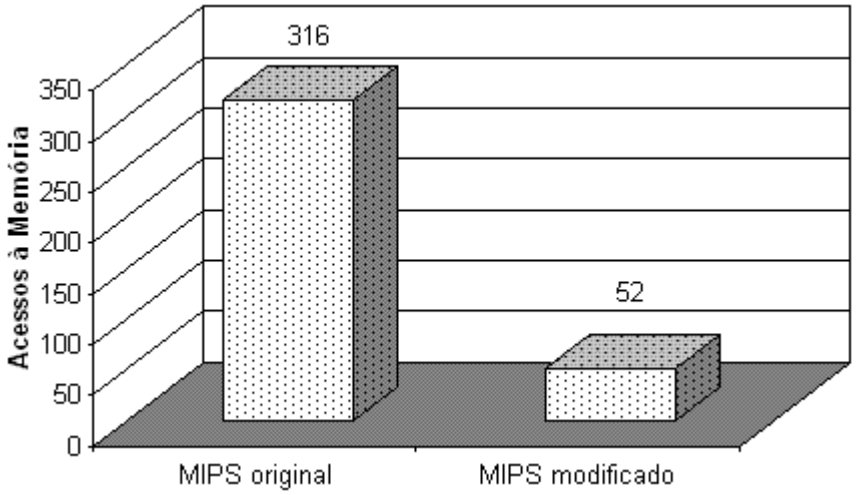

Figura 6.6: Quantidade de acessos à memória.

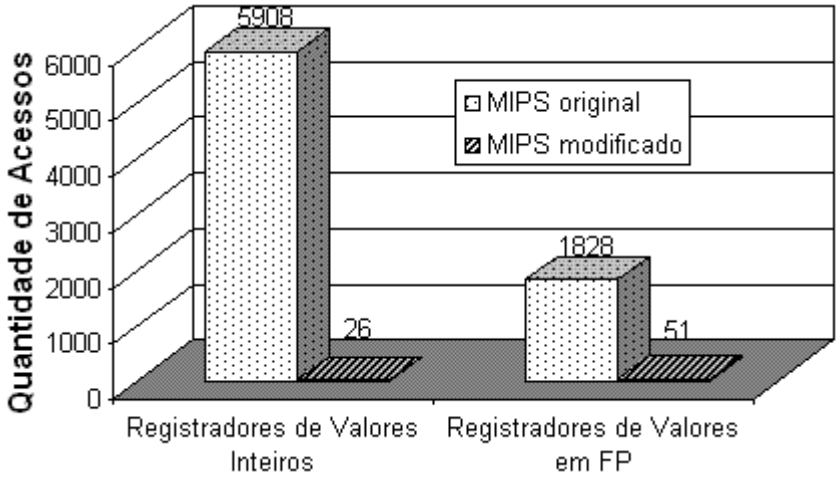

Figura 6.7: Quantidade de acessos ao banco de registradores.

A execução da aplicação no novo conjunto de instruções foi quase 150 vezes mais rápido que o original, com igual proporção de instruções executadas, conforme se previu na seção 6.2. Enquanto o *ISA* original acessou a memória 316 vezes, o novo acessou apenas 52, ou seja, uma quantidade 6 vezes menos. Por fim, o total de acesso aos registradores inteiros reduziu aproximadamente 3 vezes, embora os registradores de *FP* tiveram 25 acessos a mais.

6.4 Conclusões

O ganho de desempenho da nova arquitetura é excepcionalmente maior, o que significa que a inclusão das novas instruções no *ISA* para aplicá-la na área da robótica vale a pena. O gargalo da aplicação no R3000 original se encontra no cálculo do seno e cosseno porque é preciso executar diversas instruções aritméticas e lógicas na execução da série de Taylor e McLauren, enquanto que a resposta é obtida com apenas uma instrução no novo ISA. Além disso, realizam-se 8 operações para efetuar uma operação de rotação, enquanto é necessário realizar apenas uma com o conjunto de instruções estendido.

Os resultados obtidos aqui não são definitivos, pois, como dito na seção 6.3, o custo físico para obter esse desempenho não foi levado em consideração. Com a inclusão de novas unidades aritméticas, responsáveis por executar as novas instruções, a área e o consumo de potência do processador aumentará. Somado a isso, a principal consequência de estender o conjunto de instruções está na determinação da nova velocidade do processador, porque instruções complexas demoram mais tempo para completarem, e essa latência influencia diretamente no período do *clock* do processador. Toda essa análise é feita a seguir.

7 IMPLEMENTAÇÃO FÍSICA

7.1 Introdução

Os resultados obtidos no capítulo anterior mostram que o desempenho de velocidade do R3000 aumenta significativamente com a inserção das novas instruções, entretanto eles ainda não são conclusivos porque não consideram os custos das mesmas. Tal conclusão só é obtida com a implementação física do processador, porque é nessa etapa que são inseridas as unidades de execução, registradores e barramentos responsáveis pela realização das novas operações. Através da análise dos custos de tamanho de área, consumo de potência e a nova velocidade do processador, é possível concluir se a implementação desse *ISA* é interessante ou não.

Entretanto, a confecção de um processador é bastante custosa porque envolve todo um processo de limpeza e purificação do silício, geração dos discos que contém os processadores, a queima do mesmo para a geração do *layout*, além de toda infra-estrutura necessária à sua confecção (Patterson, 2005). Assim, com a utilização de computação reconfigurável, é possível simular um processador físico através de ferramentas que permitem descrevê-lo e sintetizá-lo em *FPGA* (*Field Programmable Gate Array*). A sintetização gera os resultados físicos necessários para concluir efetivamente a viabilidade da proposta do novo *ISA*.

Esse capítulo apresentará o conceito de computação reconfigurável, o dispositivo físico configurável *FPGA* e a sua importância para a sintetização de um processador. Depois, mostra-se o PLASMA (Rhoads, 2001), um processador que tem o mesmo *ISA* do MIPS R3000, seguido da modificação da sua descrição para adaptá-lo às novas instruções. Por fim, mostram-se os resultados obtidos com a sintetização do PLASMA em *FPGA*.

7.2 Computação Reconfigurável

A computação reconfigurável é um conceito computacional que tem por objetivo utilizar os benefícios do hardware e do software convencional (Martins, 2003). A vantagem do hardware convencional está no fato de desempenhar tarefas de forma bastante rápida. Aplicações que exigem respostas rápidas, como sistemas de tempo-real, normalmente são implementadas em hardware. Contudo, o hardware é bastante limitado por ser estático, e normalmente ele é usado em apenas uma aplicação. O software não sofre esse defeito porque é configurável, e por isso é capaz de resolver diversos problemas, mas peca pelo desempenho. Durante a execução do software, o processador precisa buscar a instrução da memória, decodificar, executar e salvar os resultados das operações, um *overhead* que não existe no hardware. Através da computação

reconfigurável, os ambientes de desenvolvimente de aplicações buscam a maleabilidade do software somado à velocidade do hardware almejando, dessa forma, realizar diversas aplicações com bom desempenho.

7.2.1 FPGA

Embora a computação reconfigurável se aplique em diversas áreas da computação, a área dos dispositivos reconfiguráveis é a que mais utiliza esse conceito. Dentre esses dispositivos existem as *EPROM* (*Erasable Programmable Read Only Memory*), *PLA* (*Programmable Logic Array*), *CPLD* (*Complex Programable Logic Device*) e o *FPGA*. Esse último é o mais utilizado em computação reconfigurável.

O *FPGA* é um dispositivo composto por blocos lógicos configuráveis (*CLB – Configurable Logic Block*), Buffers de Entrada e Saída (*IOB – Input Output Buffers*) e uma rede de interconexão responsável por fornecer comunicação entre os elementos. Os *CLB* são as unidades responsáveis pela criação de elementos de execução do *FPGA*, e se organizam para realizar a tarefa que foi definida na configuração. Sua arquitetura é composta por *LUT*s (*Lookup Tables*), que efetuam funções combinacionais, multiplexadores, portas de entrada e saída, e flip-flops atuando no papel de registradores nas entradas dessas portas. Quando a aplicação é complexa os *CLB* são configurados para se comportar como componentes mais elaborados, como uma *ALU*. A rede de interconexão conecta as *CLB* em uma topologia em forma de matriz, e nas bordas desta, estão os *IOB* responsáveis por fazer a comunicação dos *CLB* com o mundo externo. Em um ambiente no qual o *FPGA* é configurado como um processador, a memória está localizada no lado externo da matriz, logo, os *IOB* são responsáveis por receber e enviar dados para essa memória. A figura 7.1 mostra a arquitetura de um *FPGA*.

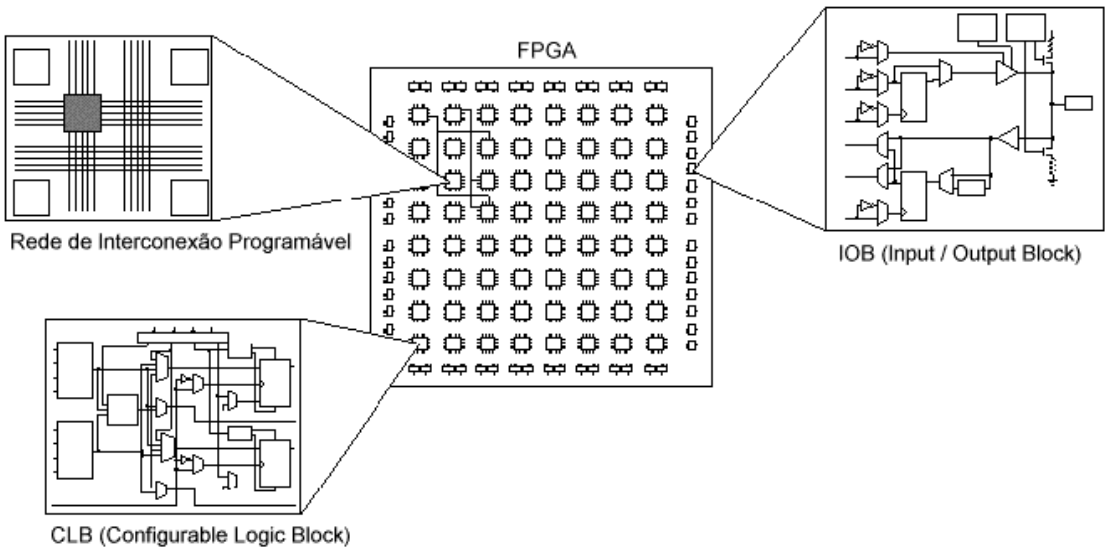

Figura 7.1: Arquitetura de um FPGA (Martins, 2003).

É através da sintetização, ou seja, do mapeamento da descrição do R3000 em *FPGA* que se obtém os resultados físicos desejados. A quantidade de área ocupada pelo novo processador é dada principalmente pelo total de *LUT*s, *flip-flops* e *IOB* usados. Também são determinados a nova frequência, e o consumo de potência do processador.

Existem diversos ambientes de desenvolvimento que permitem a implementação de projetos em computação reconfigurável e sistemas digitais, dentre as podemos citar o *ISE*, da Xilinx® (Xilinx, 2009), e o *Quartus II*, da Altera® (Altera, 2009). Para ambas

ferramentas existem versões empresariais e acadêmicas, sendo a última opção, gratuita. Para implementar a extensão do MIPS R3000, foi utilizada a versão acadêmica *ISE Webpack*, da Xilinx®, limitada a projetos pequenos, com sintetização configurada para o dispositivo *FPGA* XC5VLX50 da família Virtex5.

7.3 O Processador PLASMA

O PLASMA (Rhoads, 2001) é um processador de código aberto e possui o mesmo conjunto de instruções do *MIPS R3000*. Seu desenvolvimento foi realizado usando a linguagem de descrição de hardware *VHDL* (*VHSIC Hardware Description Language*, onde *VHSIC* é o acrônimo de *Very High Speed Integrated Circuit*, ou circuitos integrados de velocidade muito alta). O PLASMA foi projetado para usar dois tipos de *pipelines*: um de profundidade 4, e outro, de 5, logo, dependendo da aplicação, pode-se optar entre um ou o outro. Embora possua o caminho de dados diferente do MIPS R3000, o fato dele possuir o mesmo *ISA* justifica seu uso no projeto, porque o objetivo é validar as novas instruções. As figuras 7.2, 7.3, 7.4, 7.5 e 7.6 mostram as unidades de parte caminho de dados acessadas em cada um dos estágios do *pipeline* de profundidade 5 durante a execução de uma instrução *tipo – R ou tipo – I*.

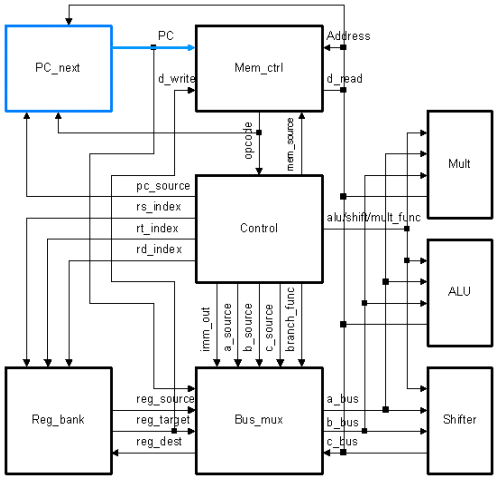

Figura 7.2: Estágio 1 do *pipeline* do PLASMA.

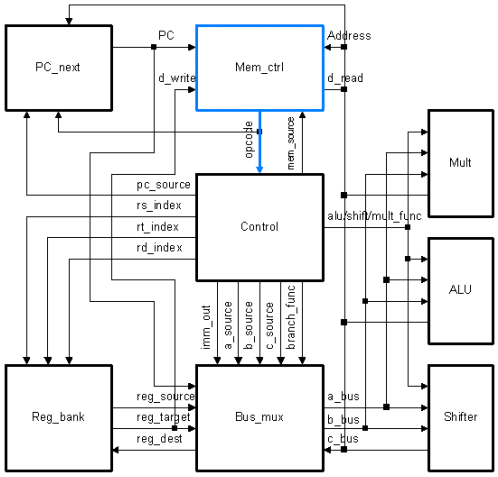

Figura 7.3: Estágio 2 do *pipeline* do PLASMA.

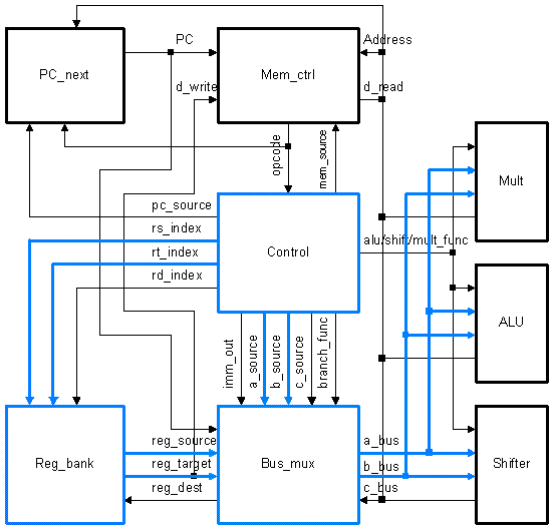

Figura 7.4: Estágio 3 do *pipeline* do PLASMA.

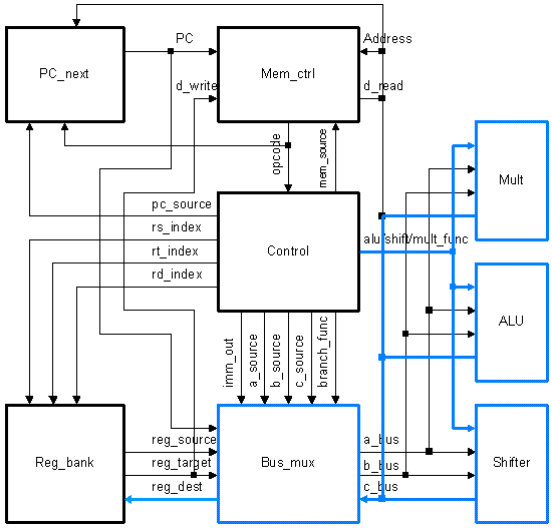

Figura 7.5: Estágio 4 do *pipeline* do PLASMA.

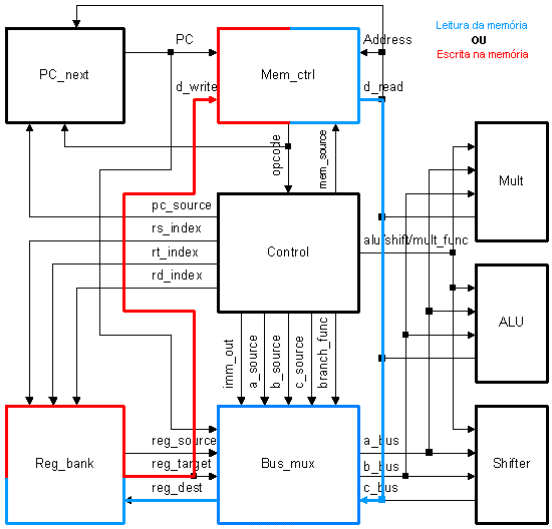

Figura 7.6: Estágio 5 do *pipeline* do PLASMA.

No primeiro estágio, o endereço do PC é enviado à para fazer a leitura da próxima instrução, e é incrementado. No segundo, a instrução é buscada da memória. O terceiro estágio efetua diversas funções: a instrução é decodificada, os sinais de habilitação são ativados, os dados são lidos do banco de registradores e encaminhados para as respectivas unidades de execução. Na quarta etapa, a instrução é executada e o resultado é encaminhado ao barramento, para ser armazenado no banco de registradores, ou escrito na memória. Finalmente, no último estágio o dado é salvo no banco de registradores, ou lido/escrito na memória.

7.4 Implementação do Novo Caminho de Dados

Para adaptar o caminho de dados do processador PLASMA às novas instruções, foi necessário efetuar diversas mudanças. De forma semelhante à descrição do MIPS R3000 em *ArchC*, o PLASMA original também não possui uma unidade de *FP*, e esse foi o ponto de partida do desenvolvimento das alterações. Assim, foram criadas as unidades de conversão de valores inteiros para *FP*, soma, subtração, multiplicação e divisão. Em seguida, o banco de registradores foi modificado para permitir a leitura e escrita de múltiplos dados em paralelo, seguida das alterações dos barramentos por onde eles trafegam. Logo após, foram feitas modificações no *pipeline* e na unidade de *forwarding*. Por fim, incluiu-se a unidade CORDIC (Andraka, 1998), para o calculo do seno e cosseno, a unidade de translação e de rotação. Finalizada a implementação, o novo PLASMA foi sintetizado para obter os resultados físicos de aumento de área e nova velocidade, terminando com a simulação comportamental da mesma aplicação realizada na seção 6.2. No entanto, dessa vez foi levado em conta a nova frequência do processador para determinar o novo tempo de execução.

7.4.1 Inclusão das Unidades de Ponto Flutuante

Diferente da implementação da instrução de conversão de valores inteiro para *FP* em *ArchC*, foi preciso analisar os bits que representam o valor inteiro para convertê-lo em *FP*. Para fazer a conversão, primeiramente são realizados testes para verificar se o valor não é zero ou negativo. Se a primeira opção for verdadeira, então o algoritmo é finalizado, pois o valor já está no formato *FP*. Se a segunda opção for verdadeira, então o bit de sinal é armazenado temporariamente, o valor é transformando em positivo, a conversão é efetuada, e o bit de sinal é concatenado ao novo dado no final do algoritmo, porque a representação de números em *FP* é no formato sinal-magnitude. Para aplicar a conversão, percorre-se o *array* de bits a partir do *MSB* até encontrar o primeiro bit em 1. A posição desse bit no *array* é o expoente, e o restante, a fração. Assim, concatena-se o os oito bits do expoente com a fração. A figura 7.7 mostra o algoritmo de conversão de valores inteiros em *FP*.

Para desenvolver as operações de soma, subtração, multiplicação e divisão em *FP*, utilizaram-se os núcleos de Propriedades Intelectuais (*Intelectual Properties Cores – IP Cores*) disponibilizadas no *ISE*. Porém, essas funções exigem mais de um ciclo de *clock* para serem efetuadas. Dependendo da operação, os *IP Cores* permitem a escolha da quantidade de ciclos necessários para efetuar o cálculo. Se for escolhida a opção de executar as operações em um único ciclo, então essas instruções terão uma latência muito grande, afetando negativamente a determinação da nova frequência do processador. Por outro lado, ao usar a quantidade máxima de ciclos permitida pelo respectivo *IP Core*, a velocidade do processador será alta, mas a execução dessas operações causará um gargalo, porque o fluxo de execução do *pipeline* congelará devido

à espera pelo completamento dessa instrução. Dessa forma, com o objetivo de obter o melhor custo – benefício entre a latência da operação e o total de ciclos, optou-se por utilizar aproximadamente a metade dos ciclos disponíveis pelos *IP Cores* para uma dada instrução. Dessa forma, todas as operações aritméticas de *FP* utilizam seis ciclos, à exceção da divisão que usa 10. Logo após, integraram-se esses núcleos aritméticos ao caminho de dados, ligando-o aos sinais de controle gerados pela unidade de controle. As figuras 7.8, 7.9, 7.10 e 7.11 apresentam os componentes de conversão de inteiro para *FP*, dos *IP Cores* da soma e subtração, multiplicação, e divisão em FP, respectivamente. Logo após, a figura 7.12 mostra a organização do PLASMA com as novas unidades incluídas.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mlite_pack.all;

-- A module that converts an integer value to 32 bits floating point value
-- that compliances with IEEE 754 standard

entity integer_fp is
   port(reset, enable : in std_logic;
        integer_data_in : in std_logic_vector(31 downto 0);
        fp_data_out : out std_logic_vector(31 downto 0));
end entity;

architecture behavior of integer_fp is
   signal exponent : std_logic_vector(7 downto 0);
   signal mantisse : std_logic_vector(22 downto 0);
begin
   -- process that put the value in scientific notation and gets the exponent
   normalizing_process : process (reset, enable, integer_data_in)
      variable aux : std_logic_vector(31 downto 0);
      variable backup_integer_data_in : std_logic_vector(31 downto 0);
      variable aux_negated : integer;
      variable int_exponent : integer range 0 to 32;
      variable sign : std_logic;
   begin
      if reset = '1' or enable = '0' then
         exponent <= ZERO(7 downto 0);
         mantisse <= ZERO(22 downto 0);
         fp_data_out <= ZERO;
         aux := integer_data_in;
         backup_integer_data_in := integer_data_in;
         int_exponent := 32; -- helps on exponent computation
      elsif enable = '1' then
         exponent <= ZERO(7 downto 0);
         mantisse <= ZERO(22 downto 0);
         int_exponent := 32;
         -- if the number is negative, converts to positive with 2-complement
         -- and set the sign indicating that's a negative number
         if integer_data_in(31) = '1' then
            sign := '1'; -- number is negative
            -- converts from std_logic_vector to integer
            aux_negated := to_integer(unsigned(not integer_data_in));
            aux_negated := aux_negated + 1;
            aux := std_logic_vector(to_unsigned(aux_negated,32));
         else
            sign := '0'; -- number is positive
            aux := integer_data_in;
         end if;
         -- saves the value to helps on computation of mantisse
         backup_integer_data_in := aux;

         -- if the operand is not zero (if it is, the algorithm is finished...)
         if integer_data_in /= ZERO then
            -- computes the exponent
            -- int_exponent is the index of 'aux' and is used to fetch one
            -- position after (from the end(31) to beginning(0)) it finds the first '1'
            -- of the aux array
            -- ex: aux = 00000000000000000000000000011001 (integer 25)
            -- int_exponent starts with 32 and, after loop, will be 4.
            -- it's used to calculate the mantisse, too.
            -- This value in FP is 01000001110010000000000000000000 (FP 25.0)
            for I in 0 to 31 loop
               int_exponent := int_exponent - 1;
               if aux(31) = '0' then
                  aux := aux(30 downto 0)  & '0';
               else exit;
               end if;
            end loop;
            fp_data_out <= sign &
                           -- concatenate the exponent (with bias)
                           -- obs: converts from integer to std_logic_vector
                           std_logic_vector(to_signed(127+int_exponent,8)) &
                           -- concatenate the mantisse
                           backup_integer_data_in(int_exponent-1 downto 0) &
                           ZERO(22 downto int_exponent);
         end if; -- integer_data_in /= ZERO
      end if;
   end process;

end architecture;
```

Figura 7.7: Algoritmo de conversão de inteiro para *FP*.

```
396   component integer_fp
397      port(clk in std_logic;
398           reset, enable : in std_logic;
399           integer_data_in : in std_logic_vector(31 downto 0);
400           fp_data_out : out std_logic_vector(31 downto 0));
401   end component;
```

Figura 7.8: Unidade de conversão de valores inteiros em *FP*.

```
403    component add_sub_fpu
404       port(
405             a: IN std_logic_VECTOR(31 downto 0);
406             b: IN std_logic_VECTOR(31 downto 0);
407             operation: IN std_logic_VECTOR(5 downto 0);
408             operation_nd: IN std_logic;
409             clk: IN std_logic;
410             sclr: IN std_logic;
411             result: OUT std_logic_VECTOR(31 downto 0);
412             rdy: OUT std_logic);
413    end component;
```

Figura 7.9: Unidade de soma e subtração em *FP*.

```
415    component mult_fpu
416       port(
417             a: IN std_logic_VECTOR(31 downto 0);
418             b: IN std_logic_VECTOR(31 downto 0);
419             operation_nd: IN std_logic;
420             clk: IN std_logic;
421             sclr: IN std_logic;
422             result: OUT std_logic_VECTOR(31 downto 0);
423             rdy: OUT std_logic);
424    end component;
```

Figura 7.10: Unidade de multiplicação em *FP*.

```
426    component div_fpu
427       port(
428             a: IN std_logic_VECTOR(31 downto 0);
429             b: IN std_logic_VECTOR(31 downto 0);
430             operation_nd: IN std_logic;
431             clk: IN std_logic;
432             sclr: IN std_logic;
433             result: OUT std_logic_VECTOR(31 downto 0);
434             rdy: OUT std_logic);
435    end component;
```

Figura 7.11: Unidade de divisão em *FP*.

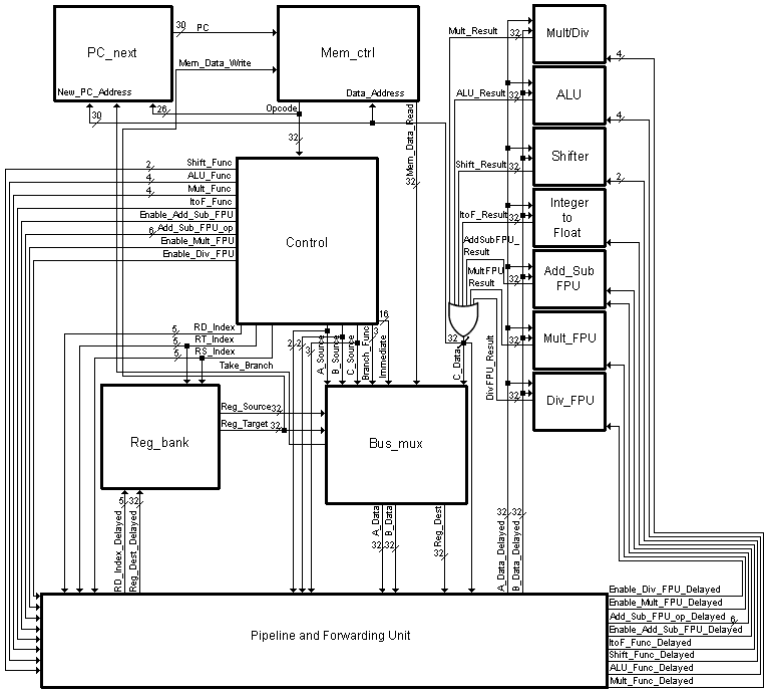

Figura 7.12: Organização do PLASMA com as unidades de *FP*.

7.4.2 Inclusão das Novas Instruções

Uma vez que se atingiram as premissas necessárias à implementação das novas instruções, finalmente foi possível incluí-las no caminho de dados do PLASMA. Assim, primeiramente modificou-se o banco de registradores para permitir a leitura e escrita de mais de um valor em paralelo. As figuras 7.13 e 7.14 mostram as modificações no esquema lógico do banco de registradores, e a declaração desse componente em VHDL.

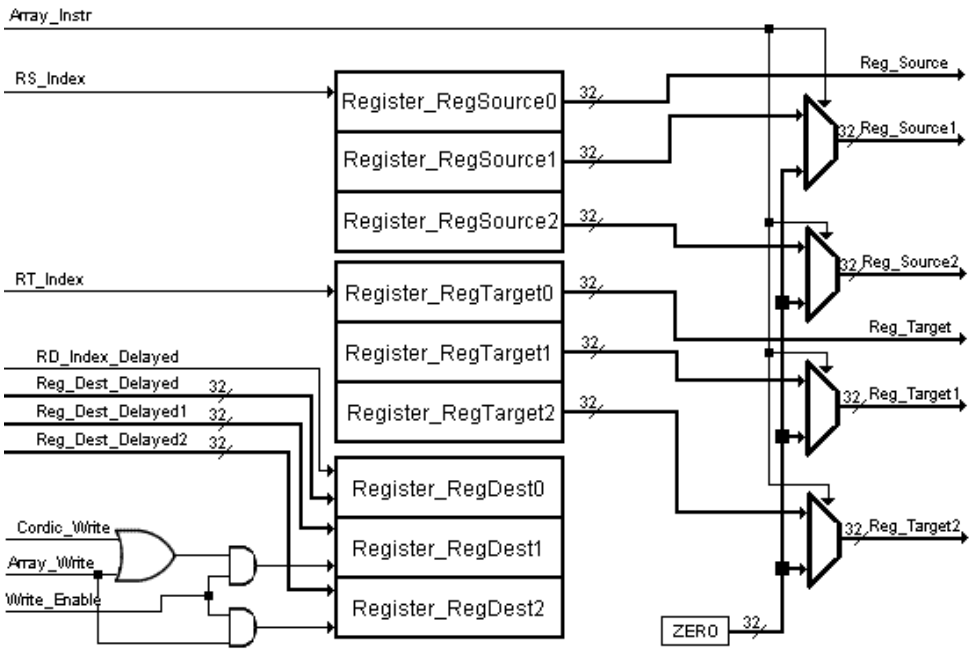

Figura 7.13: Nova lógica para múltiplas leituras e escritas no banco de registradores.

```
393     component reg_bank_tri_access
394        generic(memory_type : string := "RAM16X1D_TRIACCESS");
395        port(clk             : in  std_logic;
396          reset_in       : in  std_logic;
397          pause          : in  std_logic;
398          ------------------------------------------------
399          cordic_write   : in  std_logic;
400          array_instr    : in  std_logic;
401          array_instr_write    : in  std_logic;
402          ------------------------------------------------
403          rs_index       : in  std_logic_vector(5 downto 0);
404          rt_index       : in  std_logic_vector(5 downto 0);
405          rd_index       : in  std_logic_vector(5 downto 0);
406
407          reg_source_out : out std_logic_vector(31 downto 0);
408          ------------------------------------------------
409          reg_source_out1 : out std_logic_vector(31 downto 0);
410          reg_source_out2 : out std_logic_vector(31 downto 0);
411          ------------------------------------------------
412
413          reg_target_out : out std_logic_vector(31 downto 0);
414          ------------------------------------------------
415          reg_target_out1 : out std_logic_vector(31 downto 0);
416          reg_target_out2 : out std_logic_vector(31 downto 0);
417          ------------------------------------------------
418
419          reg_dest_new   : in  std_logic_vector(31 downto 0);
420          ------------------------------------------------
421          reg_dest_new1   : in  std_logic_vector(31 downto 0);
422          reg_dest_new2   : in  std_logic_vector(31 downto 0);
423          ------------------------------------------------
424          intr_enable    : out std_logic);
425     end component;
```

Figura 7.14: Unidade do novo banco de registradores.

A inclusão da unidade de cálculo do seno/cosseno seguiu a mesma lógica de inclusão das operações aritméticas de *FP*, ou seja, através da inserção do *IP Core* responsável por essa função. Entretanto, diferente da expansão em série de Taylor e McLauren do seno e do cosseno usado na aplicação da seção 6.2, o *IP Core* implementa o algoritmo de CORDIC, muito usado para efetuar operações trigonométricas em ambientes digitais. A declaração do componente seno/cosseno é mostrada na figura 7.15.

```
643     -- components for sine/cosine calculation
644     ------------------------------------------------------
645     component sinecosine
646        port(phase_in : in std_logic_vector(31 downto 0);
647             clk, reset, enable : in std_logic;
648             sinecosine_rdy : out std_logic;
649             error : out std_logic;
650             pause_cordic : out std_logic;
651             sine : out std_logic_vector(31 downto 0);
652             cosine : out std_logic_vector(31 downto 0));
653     end component;
```

Figura 7.15: Unidade Seno – Cosseno.

A alteração no caminho de dados do PLASMA que permitiu a inclusão da instrução de translação se deu através da adição de mais dois *IP Cores* de soma e subtração em *FP*. Dessa forma, com essas três unidades, é possível efetuar as três somas ou subtrações da operação de translação em paralelo. Os dois novos *IP Cores* demoram seis ciclos de *clock* para efetuarem a operação. Por fim, a unidade de rotação possui como entradas as coordenadas do ponto a ser transformado, junto com o seno e o cosseno do ângulo de rotação. Ela é composta por quatro unidades de multiplicação, duas unidades de soma e subtração em *FP*, e multiplexadores que selecionam os valores a serem multiplicados e somados para gerarem a resposta. A figura 7.16 mostra o esquemático da unidade de rotação.

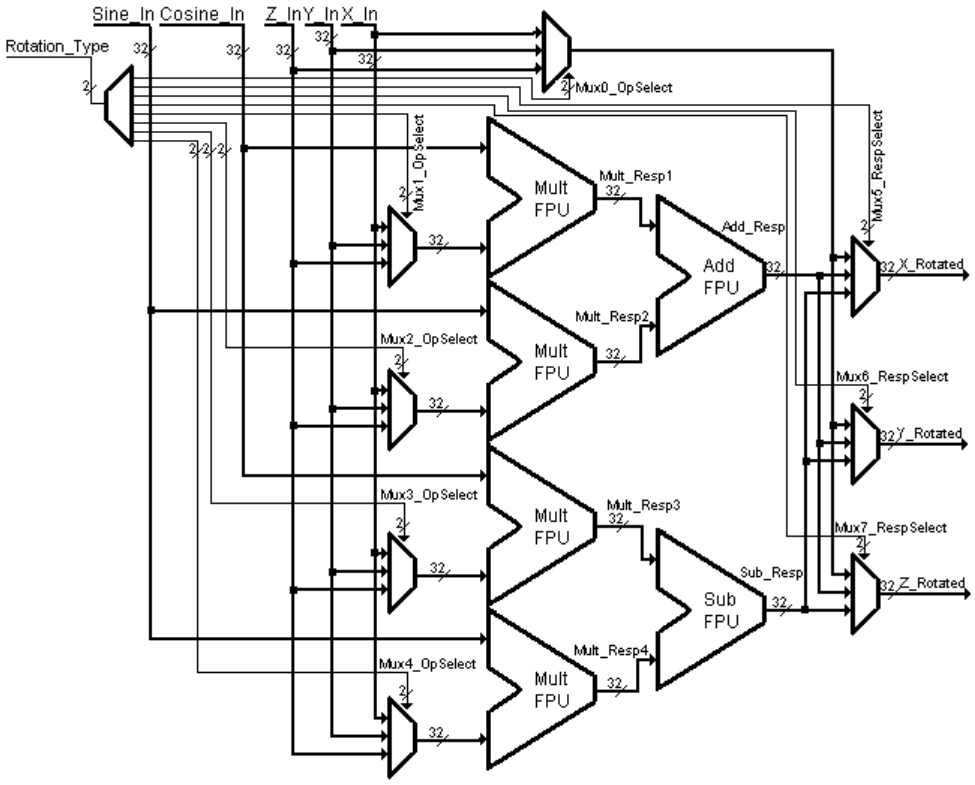

Figura 7.16: Lógica da unidade de rotação.

Supondo que o ponto a ser transformado possui coordenadas p_x, p_y, p_z, e o seno e cosseno de um dado ângulo são representados pelos sinais *sine_in* e *cosine_in*. Se a instrução a ser realizada for rotação em X, então o sinal *Mux0_OpSelect* deixa passar o valor p_x, e o sinal *Mux1_OpSelect* seleciona p_z para ser multiplicado com *cosine_in*, gerando a resposta *Mult_Resp1*. O sinal *Mux2_OpSelect* seleciona p_y para ser multiplicado com *sine_in* e obtém como resposta *Mult_Resp2*. Em paralelo, os sinais *Mux3_OpSelect* e *Mux4_OpSelect* multiplexam p_x e p_z para serem multiplicados por *cosine_in e sine_in*, resultando nos valores *Mult_Resp3* e *Mult_Resp4*, respectivamente. Após a etapa de multiplicação, é feita a soma de *Mult_Resp1* com *Mult_Resp2* que geram como resposta *Add_Resp*, e a subtração de *Mult_Resp3* com *Mult_Resp4*, obtendo *Sub_Resp*. Por fim, os sinais *Mux5_RespSelect*, *Mux6_RespSelect* e *Mux7_RespSelect* multiplexam p_x, *Add_Resp* e *Sub_Resp*, colocando as novas coordenadas do ponto rotacionado nas portas de saída da unidade de rotação. A declaração do componente da unidade de rotação é mostrada na figura 7.17.

```
680     component rotation_unit
681        port(clk, reset : in std_logic;
682             enable : in std_logic;
683             rotation_op : in std_logic_vector(1 downto 0);
684             sine_in : in std_logic_vector(31 downto 0);
685             cosine_in : in std_logic_vector(31 downto 0);
686             x_in : in std_logic_vector(31 downto 0);
687             y_in : in std_logic_vector(31 downto 0);
688             z_in : in std_logic_vector(31 downto 0);
689             x_out : out std_logic_vector(31 downto 0);
690             y_out : out std_logic_vector(31 downto 0);
691             z_out : out std_logic_vector(31 downto 0);
692             pause_rotation : out std_logic;
693             rotation_rdy : out std_logic);
694     end component;
```

Figura 7.17: Unidade de rotação.

Dessa forma, com as inserções das unidades responsáveis pelas operações das novas instruções, junto das adaptações do *pipeline* e da unidade de *forwarding*, as alterações do PLASMA foram finalizadas. A figura 7.18 mostra o caminho de dados final do novo processador.

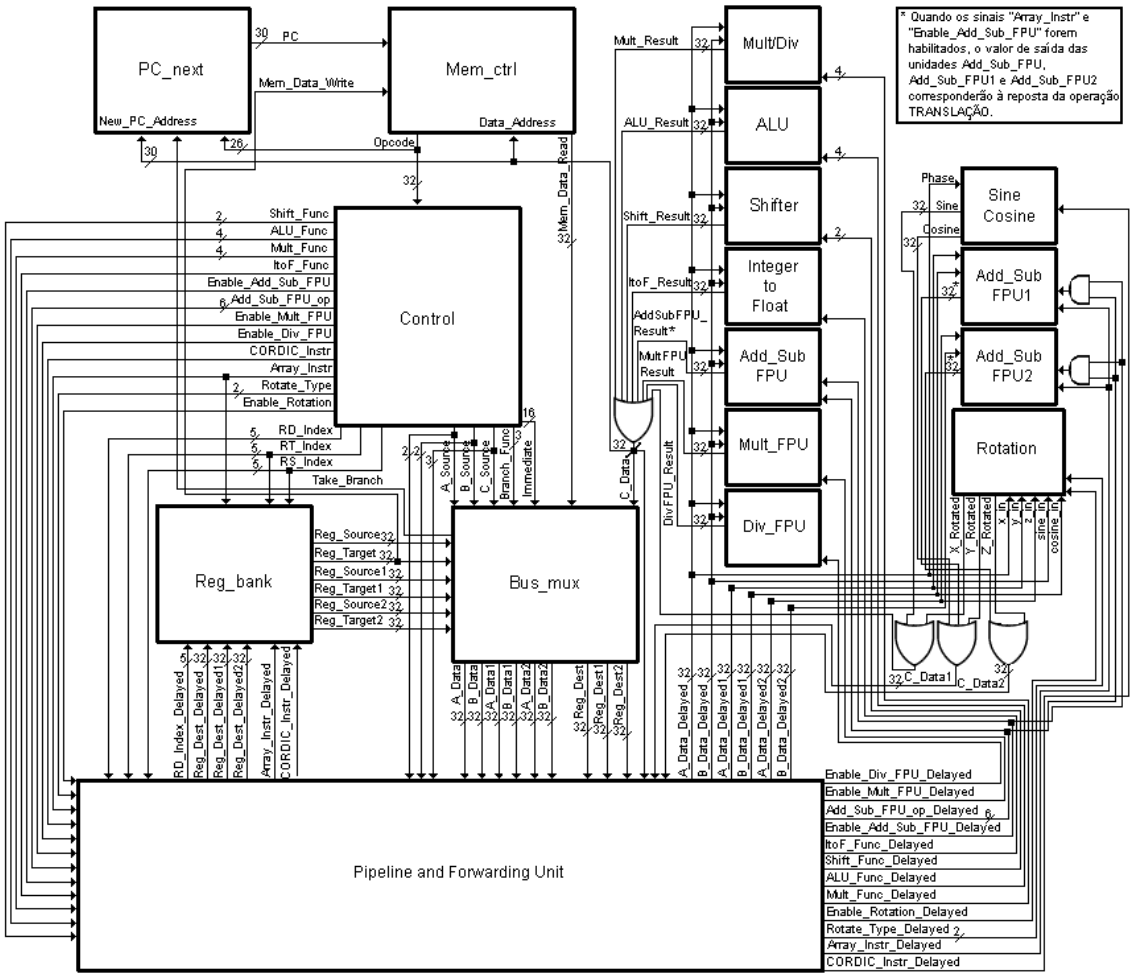

Figura 7.18: Caminho de dados final do novo PLASMA.

7.5 Resultados

Uma vez que se tem a implementação física do processador, é possível verificar o verdadeiro custo de inclusão das novas instruções. Para isso, foram medidos novos tempos de execução para a mesma aplicação da seção 6.2 para o PLASMA com o *ISA* original, e a mesma aplicação para o *ISA* novo. Como não existe um compilador responsável pela compilação e montagem da aplicação para o novo processador, todo processo de montagem do programa e determinação dos endereços de destinos para desvios condicionais e incondicionais foi realizado manualmente. Para instanciar a memória e carregar a aplicação, foram utilizados quatro componentes *IP Core Block RAM Single Port* de 8 bits de largura de palavra e 1024 posições. Dessa forma, a palavra do MIPS é dividida em quatro partes, sendo que os *MSB* da palavra, de 31 até 24, são armazenados na *Block RAM* 1, e os bits de 23 a 16, na *Block RAM* 2. Os intervalos de bits, 15 até 8, e 7 até 0, são armazenados nas *Block RAMs* 3 e 4 respectivamente. Esse esquema é mostrado na figura 7.19.

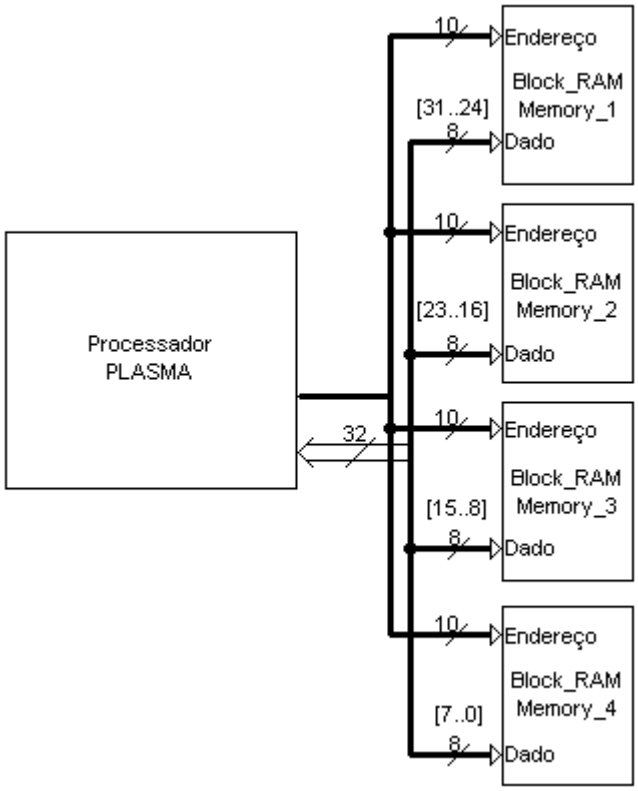

Figura 7.19: Arquitetura de acesso às memórias.

Para gerar os novos resultados, primeiramente foram carregados os binários da aplicação para o PLASMA original nas respectivas memórias, seguido da sintetização desse processador. Com isso foram obtidos os valores referentes à área ocupada e frequência. Em seguida seguiram-se os mesmos passos na geração dos resultados para o novo PLASMA. Dessa forma foi possível analisar o aumento de área de um *chip* em relação ao outro, e também comparar suas velocidade. Finalmente, uma vez que se obteve a frequência dos processadores, foi possível efetuar a simulação comportamental (*Behavioral Simulation*) com a execução da aplicação para dois processadores. A quantidade de pares de *LUTs-FF*, *slice LUTS*, *IO Buffers* e *slice registers*, mostrando o total de área ocupada, são mostrados nas figuras 7.20, 7.21, 7.22 e 7.23,

respectivamente. Já os valores de velocidade, e novos tempos de execução baseados na verdadeira frequência são vistos nas figuras 7.25, 7.26, nessa ordem.

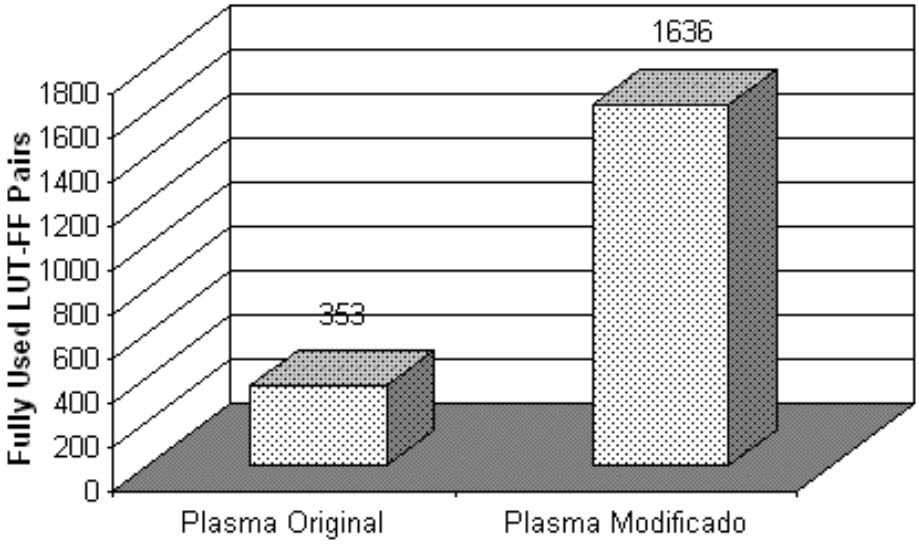

Figura 7.20: Quantidade de *Fully Used LUTs-FF Pairs*.

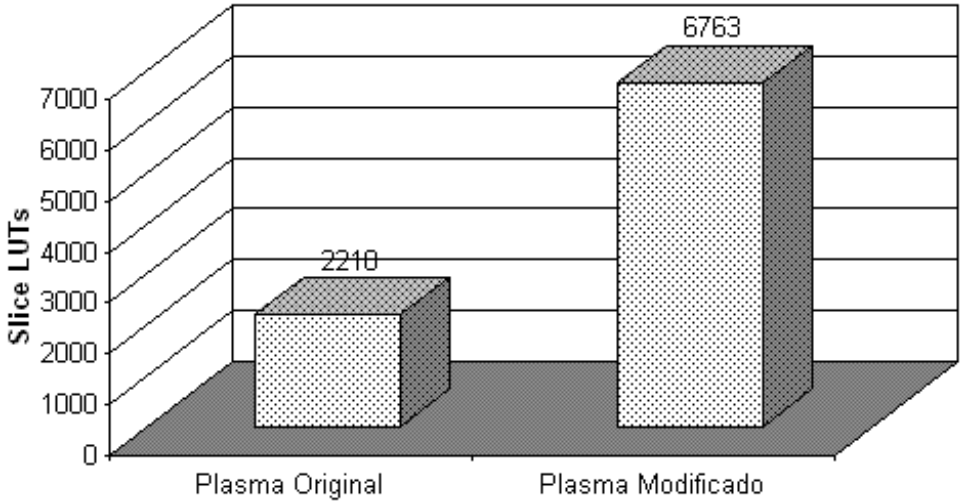

Figura 7.21: Quantidade de *Slice LUTs*.

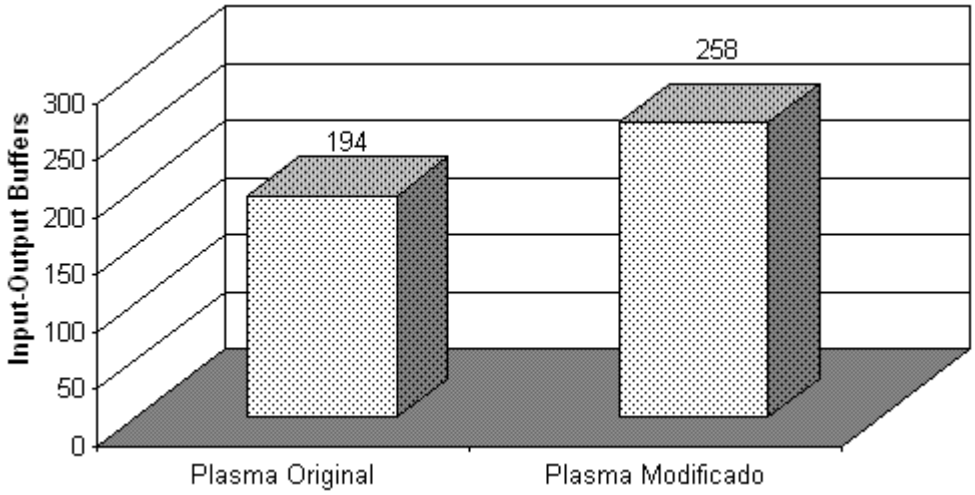

Figura 7.22: Quantidade de *IO Buffers*.

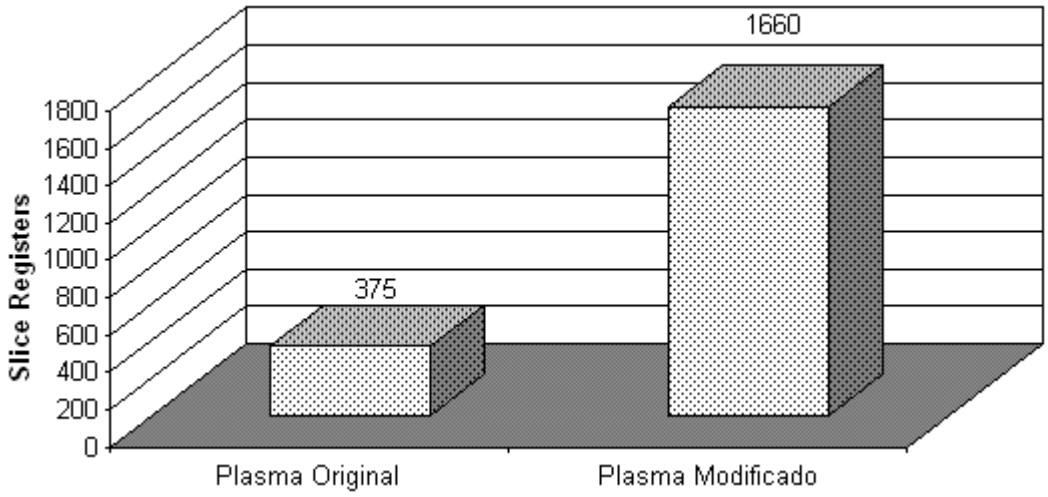

Figura 7.23: Quantidade de *Slice Registers*.

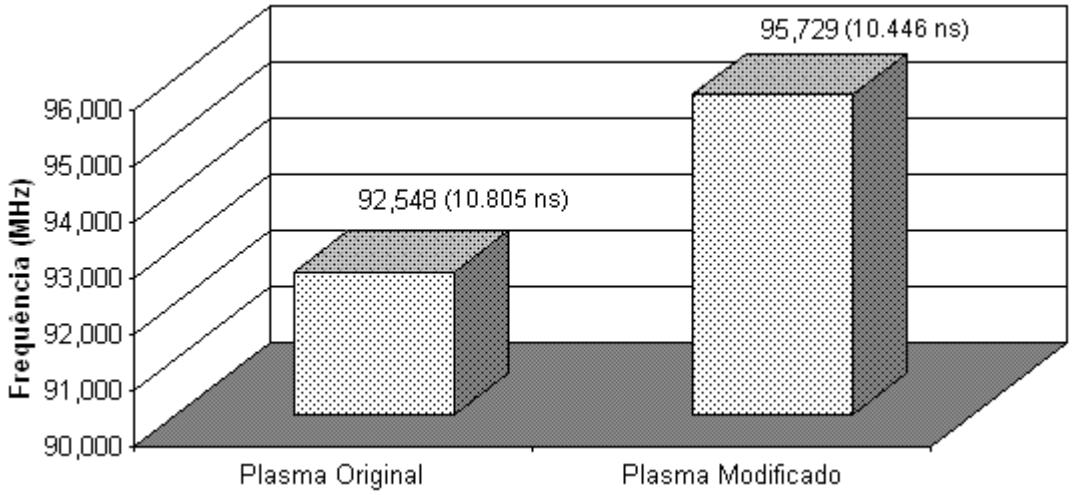

Figura 7.24: Frequência.

Houve um aumento de aproximadamente 3,6 vezes no total de *fully used LUT-FF pairs* do novo PLASMA em relação ao original, acompanhado de um incremento aproximado de 2,06 vezes no uso de *slice LUTs*. Embora o novo PLASMA tenha usado uma quantidade quase 0,33 vezes a mais de *IO Buffers*, a quantidade de *slice registers* usada por ele é aproximadamente 3,4 vezes maior do que a quantia original. É interessante perceber que, mesmo com a inclusão de novas instruções complexas ao *ISA* do PLASMA, a velocidade do novo processador foi quase 3,5% maior, quando o esperado era que diminuísse. A instrução mais complexa do novo *ISA* é o cálculo do seno e cosseno pelo algoritmo de CORDIC, logo, sua latência determinaria a velocidade do novo processador. Isso não aconteceu porque a execução dessa operação foi dividida em diversos ciclos (25 ao todo), e cada iteração é pequena o suficiente para não afetar o período do *clock*. Assim, a frequência permaneceu elevada. O que determinou a velocidade dos processadores foi o atraso na propagação de um sinal dentro da unidade de *pipeline*, gerada automaticamente na etapa de roteamento durante a sintetização do processador pelo *ISE Webpack*. Esse atraso é de 10.805ns para o PLASMA original, e 10.446ns para o novo PLASMA, o que explica o fato da velocidade do segundo ser maior do que o primeiro. Tendo os verdadeiros valores de frequência, foi possível efetuar uma nova simulação da aplicação para obter os verdadeiros tempos de execução.

A figura 7.25 compara o tempo de execução da simulação em *ArchC* da seção 6.3 com o tempo obtido pela nova simulação, para os dois processadores.

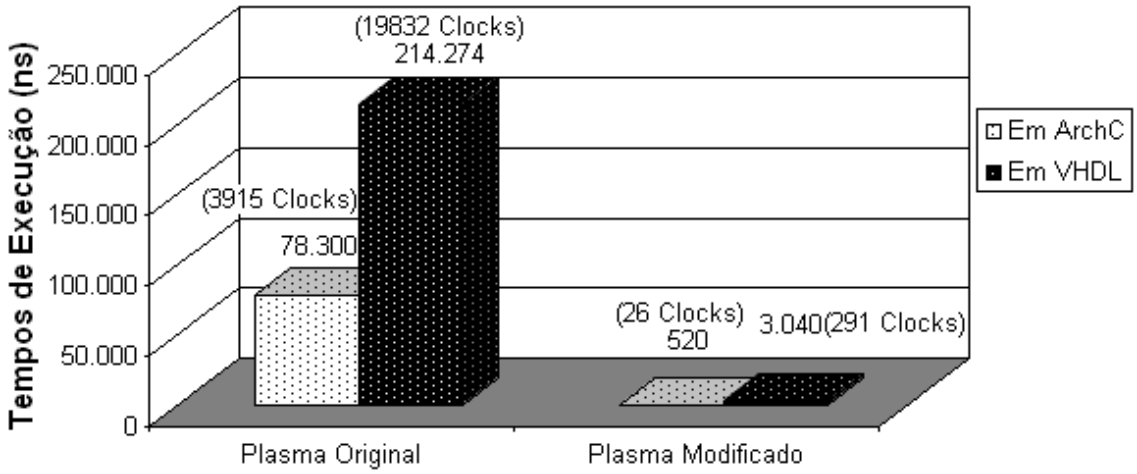

Figura 7.25: Novos tempos de execução (número de *clocks* aproximado).

O tempo de execução real do PLASMA original em relação à simulação realizada em *ArchC* aumentou mais de 1,5 vezes, enquanto que a novo tempo de simulação do PLASMA modificado foi 4,8 vezes maior comparada à simulação anterior. A frequência de operação real dos dois processadores é superior à frequência de simulação em *ArchC*, ou seja, 92,54 MHz do original e 95,72 MHz do novo contra 50 MHz. Entretanto, a maioria das instruções em *FP* nos processadores descritos em VHDL demora pelo menos seis ciclo de *clock* para serem executadas, enquanto todas elas demoram apenas um ciclo, na simulação em *ArchC*. Enquanto que no PLASMA original, a aplicação foi executada em 19832 ciclos contra os 3915 ciclos da simulação em *ArchC* para o mesmo *ISA*, o novo PLASMA completou a simulação com 265 ciclos a mais que a execução da aplicação do novo *ISA* em *ArchC*, o que explica o aumento do novo tempo de execução. Ainda assim, o PLASMA modificado executou a simulação 70 vezes mais rápido que o PLASMA original, confirmando o grande ganho de desempenho de velocidade concluído na seção 6.4.

7.6 Conclusões

Apesar da área ter aumentado significativamente, não houve perda de velocidade do processador novo em relação ao original. Além disso, o tempo de execução da aplicação no PLASMA modificado foi bastante inferior ao tempo de execução do PLASMA normal. Uma vez que o objetivo da inclusão das novas instruções no PLASMA é obter maior desempenho no tempo de execução, necessitou-se incluir diversas unidades idênticas para permitir o processamento paralelo dos dados, e por consequência, o aumento da vazão. O uso de três unidades de soma e subtração em *FP* para executar a instrução de translação exemplifica essa idéia. Essa característica explica o grande aumento de área ocupada pelo novo *chip*, logo, o ganho de desempenho obtido através dessas modificações viabiliza o seu uso.

No entanto, mesmo efetuando as operações em *FP* com a metade dos ciclos de *clock* permitidos pelos *IP Cores*, a operação de cálculo do seno e do cosseno é realizada em 25 ciclos. Isso significa que essa instrução é um gargalo para o processador, e seu uso repetitivo implicará em perda de desempenho, visto que o *pipeline* permanece paralisado durante esse tempo. A solução para esse problema é permitir que as outras

unidades lógicas e aritméticas processem outras instruções em paralelo através da criação de uma unidade de despacho de instruções para a unidade de execução correspondente. Assim, o *pipeline* seguiria o fluxo normal sem paralisar, e as instruções seriam executadas em paralelo e fora da ordem (*Out-Of-Order – OOO*) dos comandos do programa, transformando o processador em superescalar. Além disso, o fato de realizar a simulação em um ambiente ideal não permitiu a obtenção dos valores de consumo de potência. Somente com a execução da aplicação em um ambiente real, na qual se leva em conta o tempo de propagação do sinal entre duas unidades digitais, será possível gerar tais resultados. A nova simulação, a geração dos resultados de consumo de potência, assim como a criação de uma unidade de despacho para execução de instruções em paralelo, serão efetuada em trabalhos futuros.

8 CONCLUSÕES

Conforme se acompanhou durante esse trabalho, o processo de projeto e desenvolvimento de processadores não é trivial. Para incluir novas instruções em um processador de propósitos gerais, de modo a auxiliar nos cálculos oriundos ao movimento de robôs, foi preciso seguir um estudo teórico sobre os processadores de propósitos gerais, seguido dos tipos de robôs existentes e as operações matemáticas que descrevem a movimentação dos mesmos. Com isso, definiu-se que as novas instruções a serem incluídas no processador são: translação, rotação em X, rotação em Y, rotação em Z, seno e cosseno.

O propósito do projeto foi analisar o impacto de desempenho das novas instruções ao *ISA* do MIPS. Através da simulação de uma aplicação que emula o movimento de um braço mecânico no processador original e modificado, constatou-se que o segundo rodou a aplicação em 520 ns, ou seja, quase 150 vezes mais rápido que o primeiro, com igual proporção no total de instruções executadas, mostrando um ganho de desempenho de velocidade bastante significativo. Somado a isso, a quantidade de acessos à memória também foi aproximadamente 6 vezes menor, acompanhado do decréscimo de 3 vezes no acessos ao banco de registradores inteiros. No entanto, o banco de registradores de *FP* do novo *ISA* foi acessado 51 vezes, ou seja, 25 a mais que o original. Através desses resultados se conclui que é viável incluir essas instruções, visto que em quase todas as métricas, o novo *ISA* se mostrou bastante superior.

A última etapa do projeto envolveu a implementação física e análise dos custos das novas instruções. Através da sintetização da descrição do processador antigo e do novo, constatou-se que houve um grande aumento da área do segundo em relação ao primeiro. A quantidade de *fully used LUT-FF pairs* e *Slice LUTs* aumentou 3,6 vezes e 2,06 vezes, respectivamente. Esses valores foram acompanhados pelo acréscimo de 0,33 vezes de *IO Buffers* e 3,4 vezes no uso de *Slice Registers*. Tais dados são explicados pela inclusão de diversas unidades de execução idênticas, com o fim de obter o processamento paralelo dos operandos vetoriais e, consequentemente, o aumento da vazão. Isso pôde ser constatado pelo alto ganho de desempenho na velocidade obtidos com a nova execução da aplicação pelo processador modificado: 3040 ns contra os 214274 ns do original, ou seja, quase 70 vezes mais rápido. A frequência de operação da nova implementação também influenciou no tempo de execução, pois sua velocidade se mostrou superior à antiga: 95,72 MHz contra 92,54 MHz. Dessa, forma, mesmo com o grande aumento da área física, as novas instruções causam um grande impacto no ganho de desempenho de velocidade do processador MIPS R3000, confirmando a conclusão anterior. Assim, a inclusão das instruções robóticas no conjunto de instruções do processador é viável.

Entretanto, a instrução de cálculo do seno e do cosseno é um gargalo para o processador porque o *pipeline* fica paralisado por 25 ciclos. Esse problema é resolvido com o aproveitamento das demais unidades lógicas e aritméticas, que ficam ociosas durante esse tempo, através da inclusão de uma unidade de despacho de instruções, que escalona uma instrução para a respectiva unidade de execução, aumentando a utilização dos recursos do processador. Isso permitiria a execução paralela e fora de ordem dos comandos do programa. Além disso, mesmo concluindo, através dos resultados obtidos nesse trabalho, que a implementação do novo conjunto de instruções do MIPS R3000 para uso na área da robótica é importante, não se obteve o resultado de consumo de potência. Pode-se obter tal valor através da sintetização e execução da aplicação do novo *chip* em um ambiente real. Essa nova simulação, junto com a obtenção dos novos resultados, e a inclusão da unidade de despacho, serão efetuadas como trabalhos futuros.

REFERÊNCIAS

ALBERS, S.; KURSAWE, K.; SCHUIERER, S. **Exploring unknown environments with obstacles**. In Proc. of the 10th Symposium on Discrete Algorithms, 1999.

ALTERA®. **Quartus II Web Edition Software**. Última versão em 2009. Disponível em: < http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index .html>. Acesso em: jan. 2008.

ANDRAKA, R. **A Survey of CORDIC Algorithms for FPGA Based Computers**. Sixth International Symposium on Field Programmable Gate Arrays. February 1998. pp. 191-200.

ANTON, H.; BIVENS I. C.; DAVIS, S. **Cálculo**, volume 2, 8ª Edição. Editora Bookman. 2007.

ARCHC. **The ArchC Architecture Description Language**. fev. 2006. Disponível em: < http://www.archc.org/>. Acesso em: jan. 2007.

BRÄUNL, T. **Embedded Robotics**: Mobile Robot Design and Applications with Embedded Systems. Springer – Verlag, 2ª edição. 2006.

CARRO, L. **Projeto e Prototipação de Sistemas Digitais**. Editora da Universidade - UFRGS. 2001.

CARRO, L; WAGNER, F. R. **Sistemas Computacionais Embarcados**. SBC-JAI, Campinas. 2003.

CERUZZI, P. E. **A History of Modern Computing**, 2ª Edição. MIT Press, 2003.

CRAIG, J. J. **Introduction to Robotics**: Mechanics and Control, 3ª Edição. Editora Prentice Hall. 2004.

DAILY YOMIURI ONLINE **Japan's first-ever robot, version 2.0**. may 15. 2008. Disponível em: <http://www.yomiuri.co.jp/dy/features/science/20080515TDY20001. htm>. Acesso em: ago. 2009.

DARIO, P.; et al. **Robotics for medical applications**. IEEE Robotics & Automation Magazine , vol. 3, no. 3, 1996, pp. 44 - 56.

FERREIRA, M. R.; et al. **Automação de Processos da Planta de Produtos Carboquímicos da Gerdau Açominas**. Seminário de Automação de Processos da ABM, vol. 7, no. 5, Oct. 2003, pp. 208 - 217.

FRANCHIN, M. N. **Departamento de Engenharia Elétrica – UNESP**. Fev. 2007. Disponível em: < http://www.dee.feb.unesp.br/~marcelo/robotica/Robot3.htm>. Acesso em: set. 2009.

FUJITA, M.; KAGEYAMA, K. **An Open Architecture for Robot Entertainment**. In Proceedings of the First International Conference on Autonomous Agents. 1997. pp. 435 – 442.

GHOSH, A.; et al. **System modeling with SystemC**. International Conference on ASIC. 2001. pp. 18 – 20.

GREY, W. W. **An Imitation of Life**. Revista Scientific American. May 1950. pp. 42-45.

HUNTSBERGER, T. L.; RODRIGUEZ, G.; SCHENKER, P. S. **Robotics Challenges for Robotic and Human Mars Exploration**. Proc. ROBOTICS Albuquerque, NM, Mar. 2000.

HAYWARD, V.; PAUL, R. P. **Robot Manipulator Control Under UNIX – RCCL**: A Robot Control C Library. International Journal of Robotics Research, Vol. 5, Num. 4. 1986. pp. 94 – 111.

HENNESSY, J.L.; PATTERSON, D. A. **Computer architecture:** A Quantitative Approach, 4ª Edição. Editora Elsevier. 2007.

HONDA. **Azimo**: The Honda Humanoid Robot. 2000. Disponível em: <http://world.honda.com/ASIMO/>. Acesso em: ago. 2009.

HORNYAK, T. N. **Loving the Machine**: The Art and Science of Japanese Robots. Kodansha International. New York 2006.

HUFFMAN, C. **Archytas of Tarentum**. 2007. Disponível em: < http://plato.stanford.edu/entries/archytas/>. Acesso em: ago. 2009.

HWANG, K.; BRIGGS, F. A. **Computer Architecture and Parallel Processing**. Editora McGraw-Hill. 1984.

INTEL® PENTIUM® 4 PROCESSOR FAMILY. Desktop Performance and Optimization for Intel® Pentium® 4 Processor, Feb. 2001. Disponível em: <http://www.intel.com/design/pentium4/papers/249438.htm>. Acesso em: agosto 2009.

JUANG, J. G. **Collision Avoidance using Potential Fields**. International Journal of Industrial Robot. MCB Uni Press. vol. 25. 1998. pp. 408 – 415.

KOSSMAN, D.; MALOWANY, A. **Multi-Processor Robot Control System for RCCL Under IRMX**. In Proceedings of IEEE Conference on Robotics and Automation. 1987. pp. 1298 – 1306.

LAGES, W. F. **Escola de Engenharia – Grupo de controle, automação e robótica – UFRGS**. Mar. 2007. Disponível em: <http://www.ece.ufrgs.br/~fetter/eng04479/spatial.pdf>. Acesso em: set. 2009.

LEE, D. G. **Axiomatic Design and Fabrication of Composite Structures**. Oxford University Press. 2005.

LIU, H. **"The Water Mill" and Northern Song Imperial Patronage of Art, Commerce, and Science**. The Art Bulletin. Volume 84. Number 4. 2002, pp. 566 - 595.

MARESCAUX, J.; et al. **Transcontinental robot-assisted remote telesurgery**: feasibility and potential applications. Ann Surg, 2002, pp. 87 - 92.

MARTINS, C. A. P. S.; et al. **Computação Reconfigurável**: Conceitos, Tendências e Aplicações. XXII Jornada de Atualização em Informática (JAI), SBC 2003, Vol. 2. pp.339-388.

MCKERROW, P. J. **Introduction to Robotics**. Editora Addison-Welsey. 1995.

NOF, S. Y. **Handbook of Industrial Robotics**, 2ª Edição. Editora Wiley, 1999.

O'CONNOR, J. J.; ROBERTSON, E. F. **Heron Biography**. 2006. Disponível em: < http://www-history.mcs.st-andrews.ac.uk/history/Biographies/Heron.html>. Acesso em: ago. 2009.

PATTERSON, D. A.; HENNESSY, J. L. **Organização e Projeto de Computadores**: A Interface Hardware/Software, 3ª Edição. Editora Campus. 2005.

REINTJES, J. F. **Numerical Control**: Making a New Technology. Oxford University Press. 1991.

RHOADS, S. **Plasma – Most MIPS I(TM) opcodes.** 2001. Disponível em: < http://www.opencores.org>. Acesso em: jan. 2008.

RIGO, S.; et al. **ArchC**: A SystemC-Based Architecture Description Language. International Symposium on Computer Architecture and High Performance Processing. October 2004. pp.66 – 73.

ROSEN, S. **Electronic Computers**: A Historical Survey. ACM Computing Surveys (CSUR), v.1 n.1, mar. 1969. p.7-36.

ROSHEIM, M. E. **Robot Evolution**: The Development of Anthrobotics. Wiley-Interscience, 1994.

SADAYAPPAN, P.; et al. **A Restructurable VLSI Robotics Vector Processor Architecture for Real – Time Control**. IEEE Transactions on Robotics and Automation, 5ª Edição, October 1989, pp. 583 – 599.

SHARKEY, N. **A 13th Century Programmable Robot**. 2006. Disponível em: < http://www.shef.ac.uk/marcoms/eview/articles58/robot.html>. Acesso em: ago. 2009.

T4. **Technology Subjects Support Service**. Disponível em: < http://www.t4.ie/Professional_Development/RD8_Technology/Robotics/Applied%20Control%20%20Systems.ppt>. Acesso em: ago. 2009.

TAYLOR, R. H.; STOIANOVICI, D. **Medical Robotics in Computer Integrated Surgery**. IEEE Transactions on Robotics and Automation , vol. 19, no. 5, Oct. 2003, pp. 765 - 781.

TADDEI, M. **I Robot di Leonardo (La meccanica e nuovi automi nei codici svelati)**. Editora Leonardo3. 2007.

WANG, X.; et al. **New fault tolerant robotic central controller for space robot system based on ARM processor**. IEEE International Conference on Industrial Technology. apr. 2008. pp.1 - 5.

WOOD, G. **Living Dolls**: A Magical History Of The Quest For Mechanical Life. Editora Faber and Faber. 2003.

XILINX®. **ISE Webpack**. Última versão em 2009. Disponível em: < http://toolbox.xilinx.com>. Acesso em: jan. 2008.