TMS320C6000 — uma visão detalhada

Anderson de Rezende Rocha
André Atanasio Maranhão Almeida
Wylber Polonini

UNICAMP – Universidade Estadual de Campinas
Caixa Postal 6176, 13083-970, Campinas, SP, Brasil
{anderson.rocha, andre.atanasio, wylber.polonini}@ic.unicamp.br

30 de abril de 2005

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este trabalho apresenta a série de processadores digitais de sinais TMS320C6000 da *Texas Instruments*. Está série é parte da família TMS320 e é dividida em três grupos básicos de dispositivos: TMS320C62x (’C62x) e TMS320C64x que são DSPs de ponto fixo e o TMS320C67x (’C67x) que são DSPs de ponto flutuante.

Os TMSs ’C62x, ’C64x e ’C67x são compatíveis entre si tanto em código quanto em características de arquitetura. A família TMS320 é construída sobre a arquitetura *VelociTI*^{TM} (Seção 4). A arquitetura *VelociTI* é uma arquitetura de alto desempenho e instruções muito longas (VLIW) desenvolvida pela *Texas Instruments*. Estas características fazem destes dispositivos uma excelente escolha para aplicações de *multicanais* e *multifunções* por oferecer um rápido desenvolvimento de aplicações de alta performance através de um aumento crescente de paralelismo. Neste trabalho, focamos os dispositivos ’C62x e ’C67x. A série ’C64x é uma extensão do ’C62x.

Para um melhor entendimento, o trabalho está organizado como segue. O capítulo 2 apresenta as principais características da série TMS320C6000 servindo como referência rápida. O capítulo 3 apresenta um pequeno histórico da série TMS320C6000. No capítulo 4 são apresentados em detalhes a arquitetura da CPU e o conjunto de instruções enquanto o capítulo 5 mostra os principais periféricos disponíveis para esta série. As características do paralelismo, restrições de recursos e *pipeline* aparecem em destaque no capítulo 6. A organização de memória aparece no capítulo 7 e finalmente, as principais aplicações e uma pequena análise de mercado são mostradas no capítulo 8 juntamente com alguns aspectos de *benchmark* relacionados à série TMS320.

Capítulo 2

Características gerais

Este capítulo apresenta as principais características presentes nos DSPs da família TMS320. É uma visão geral destes dispositivos. Os conceitos e características apresentados são aprofundados nos capítulos posteriores do trabalho.

A família TMS320 consiste de dispositivos de ponto fixo de 16 e 32 *bits* e de ponto flutuante. Toda a família é constituída de DSPs que são dispositovos para análise digital de sinais. Os DSPs possuem tanto a flexibilidade operacional dos controladores de alto desempenho quanto a capacidade numérica de um *array* de processadores. Algumas das características gerais mais importantes destes dispositivos são:

- Conjunto de instruções muito flexível
- Flexibilidade operacional inerente
- Alta performance
- Arquitetura paralela
- Custo/benefício

O projetista e/ou desenvolvedor pode tirar muito proveito do conjunto de instruções dado que ele é muito flexível permitindo o desenvolvimento de códigos robustos e específicos. A arquitetura *VelociTI* e as VLIW garantem a alta performance e o paralelismo. Finalmente, o custo/benefício é propício ao desenvolvimento de um grande número de aplicações.

O TMS320C6000 possui uma performance superior a 2000 milhões de instruções por segundo (MIPS) rodando a 250 MHz.

Os dispositivos 'C6211, 'C6701 e 'C6202 operam nas freqüências de 150, 167, 200 e 250 MHz com tempos de ciclo respectivos de 6.67, 6, 5 e 4 nanossegundos. Todos estes DSPs executam até 8 instruções a cada ciclo. O cerne da CPU consiste de 32 registradores de propósito geral com palavras de 32 *bits* de comprimento e 8 unidades funcionais a saber:

- Dois multiplicadores
- Seis ULAs

Os dispositivos 'C62x/C67x possuem um eficiente compilador C, um *debugger windows*, um otimizador de código e um emulador de *hardware* de modo a facilitar em muito o desenvolvimento de aplicativos.

As principais características destes dispositivos são:

- Instruções ao estilo RISC (*RISC like*).
- O empacotamento de instruções (*instruction packing*) fornece a equivalência de tamanhos de código para oito instruções executadas em paralelo ou serialmente além de reduzir o consumo de energia, a busca de instruções (*fetches* de programa) e o tamanho do código.
- Todas as instruções podem executar condicionalmente (através de registradores de condição) diminuindo os custos de *branches* e aumentando o paralelismo.

- A execução do código é feita como programada, isto é, em unidades funcionais independentes.
- Suporte a dados de 8/16/32 *bits*. Isto permite um eficiente suporte aos mais variados tipos de memória disponíveis no mercado.
- Opção de realização de operações aritméticas de *40-bits* adicionando precisão extra a aplicações intensivas.
- Operações de saturação e normalização que simplificam muitas operações aritméticas.
- Suporte a manipulação de campos de *bits* tais como extração, *set*, *clear*, contagem de *bits* entre outros.

Adicionalmente, a série 'C67x (ponto flutuante) apresenta as seguintes características:

- Pico de 1336 MIPS a 167 MHz.
- Pico de 1 GFlops a 167 MHz para operações de precisão simples.
- Pico de 250 MFlops a 167 MHz para operações de precisão dupla.
- Pico de 688 MFlops a 167 MHz para operações de mulplicação e acumulação de resultados.
- Suporte em *hardware* para operações de ponto flutuante com precisão simples (*32-bits*) e precisão dupla (*64-bits*).
- Multiplicação de 32x32 bits com resultados de 32 ou 64 *bits*.

O TMS320C6000 é compatível com uma grande variedade de periféricos e memórias:

- Possui uma memória embutida *on-chip* razoavelmente grande permitindo a execução rápida de muitos algoritmos.
- A interface de memória externa de 32 *bits* suporta memórias do tipo SDRAM, SBSRAM, SRAM e outras memórias assíncronas.
- Possui uma interface *host* (*host port*) permitindo a comunicação com outras memórias, dispositivos e/ou *hosts*.
- Controlador DMA multicanal.
- Portas seriais multicanais.
- Temporizadores (*timers*) multicanais.

Um quadro-resumo comparativo entre as características de freqüência de *clock*, consumo de energia, preço e outros pontos releventes para os componentes da família TMS320C6000 é apresentado na Tabela 2.1. *Observação*: as instruções de MACs referem-se a instruções de multiplicação e acumulação de resultados. Estas operações são muito comuns, por exemplo, em comunicação de dados onde várias *Transformadas de Fourier* podem ser necessárias na conversão entre sinais.

Parâmetro	**'C62x**	**'C64x**	**'C67x**
MHz	150-300	300-1000	100-225
MIPS, MFLOPS	1200-2400 MIPS	2400-8000 MIPS	600-1350 MFLOPS
MACs 16-*bits*	300-600	1200-2400	200-550
MACs 8-*bits*	300-600	2400-8000 200-500	

Instruções especiais	Transmissão de dados de voz multicanal e processamento de imagens	Aceleração de vídeo, áudio	Operações de PF no formato IEEE com precisão simples e dupla
Consumo energia	0.9-2.1 watts	0.25-1.06 watts	0.5-1.4 watts
Preço (10000un.)	$ 8.55-$102	$19.95-$199.00	$13.50-$111.00
Periféricos	McBSP, DMA (4/16 canais), X-Bus, Timers, EMIF, HPI	McBSP, DMA (64 canais), Timers Viterbi, Co-processador Turbo	McASP, McBSP, EMIF, DMA (4/16 canais)
Tecnologia CMOS de gravação	0.13, 0.18 e 0.26 μm	0.13 e 0.18 μm	0.13, 0.18 e 0.26 μm
Packaging	256-352 BGA, 18-27mm	532 BGA, 23mm	256-352 BGA, 27-35mm

Tabela 2.1: A família TMS320C6000 – características relevantes

Nota. BGA refere-se a *ball grid array* ou quantidade de pinos no *array* do processador em questão. A Figura 2.1 apresenta o TMS320C6201 com BGA de 352 pinos.

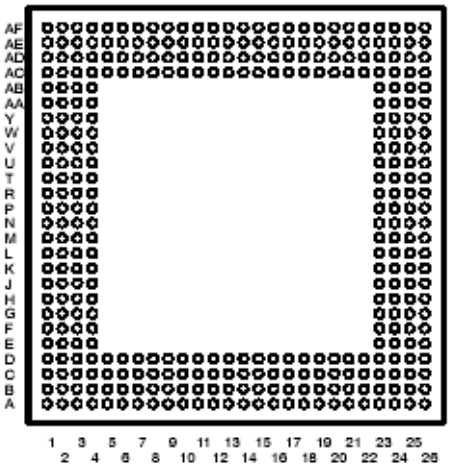

Figura 2.1: TMS320C6201 com tecnologia de *packaging* de 352 BGA

Capítulo 3

Aspectos históricos

Em 1982, a *Texas Instruments* introduziu o TMS32010 — o primeiro DSP na família TMS320. O produto fez tanto sucesso que antes de um ano, a revista *Eletronic Products* o premiava como “produto do ano”. O TMS32010 tornou-se o modelo para as futuras gerações TMS320. O TMS32010 foi fabricado com a tecnologia 2.7 *micron* com porta lógicas NMOS tendo aproximadamente 55000 transistores. Havia 60 instruções disponíveis. A dissipação de energia era em torno de 950 miliwatts. O produto era oferecido por $500 dólares.

Atualmente, a família TMS320 consiste em três plataformas de suporte: TMS320C2000, TMS320C5000 e TMS320C6000. Dentro da plataforma TMS320C6000 ainda temos três gerações: TMS320C62x, TMS320C64x e TMS320C67x.

A família ’C2000 foi desenvolvida priorizando a otimização do controle (*control optimized*) para permitir respostas a interações mais rápidas sendo propícia a aplicações de mecatrônica. Divide-de em ’C20x e ’C24x. Já a família ’C5000 tem otimização para baixo consumo de energia (*power efficient performance*) sendo propícia a aplicação de telecomunicações em geral. Finalmente, a família ’C6000 tem otimização para alta performance (*high performance*) para aplicações médicas, processamento de imagens entre outros.

De forma geral, a evolução da família ocorreu como apresentado na Figura 3.1.

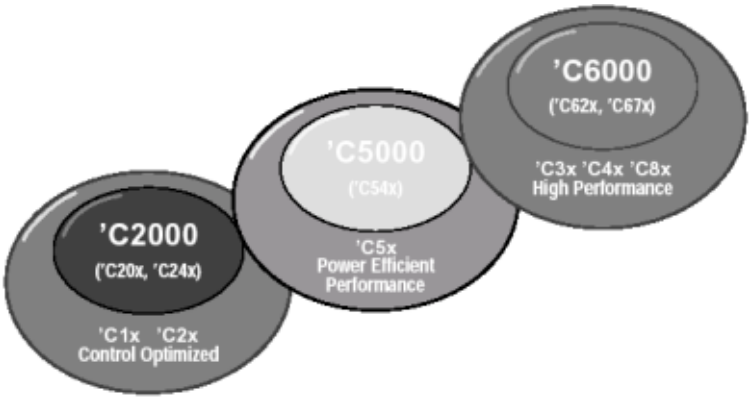

Figura 3.1: Evolução da família TMS320 de DSPs

Capítulo 4

Arquitetura da CPU e o conjunto de instruções

Este capítulo apresenta as principais características da arquitetura *VelociTI* e VLIW presente na família de DSPs TMS320. Em seguida, são apresentados os diagramas de blocos da CPU, caminhos de dados, registradores de propósito geral, unidades funcionais, registradores de controle, modos de endereçamento e interrupções.

4.1 VelociTI e VLIW

A arquitetura *VelociTI* torna os dispositivos da série 'C6000 os primeiros DSPs comerciais a usar uma melhoria da tradicional arquitetura VLIW para atingir alta performance através do aumento do nível de paralelismo.

A arquitetura VLIW tradicional consiste de múltiplas unidades de execução trabalhando em paralelo o que permite executar múltiplas instruções durante um ciclo de *clock* a exemplo da técnica superescalar. No entanto, no caso do VLIW, o compilador garante a inexistência de dependências entre as instruções distribuídas além de garantir a existência de recursos de *hardware* suficientes para executá-las. Desta forma, o paralelismo é a chave para a alta performance.

VelociTI é uma arquitetura altamente determinística, com poucas restrições sobre *como* ou *quando* as instruções serão buscadas (*fetched*), executadas ou armazendas. Esta flexibilidade é a chave para os altos níveis de eficiência alcançados pelo compilador da série 'C6000.

4.2 Diagrama de blocos

A grosso modo, os processadores da série 'C62x e 'C67x possuem três partes principais: *CPU* (núcleo), *periféricos* e *memória*. As oito unidades divididas em dois conjuntos de unidades funcionais básicas operam em paralelo. Estas duas divisões comunicam-se através de um caminho de dados cruzado (*crosspath*) entre os dois bancos de registradores. Cada banco possui 16 registradores de propósito geral (RPGs). O paralelismo é definido em tempo de compilação. Não há checagem de dependência de dados em *hardware* durante o tempo de execução. Os *fetches* de programa são de 256 *bits* de largura, isto é, a cada ciclo 8 instruções são buscadas na memória de programa.

A Figura 4.1 apresenta a CPU da família 'C62x/C67x. Estes dispositivos são desenvolvidos com uma memória de dados e de programa *on-chip* que, em alguns dispositivos, podem ser configuradas como memórias *cache*. Os periféricos incluem controladores para acessos diretos à memória (DMA), lógica de desligamento (*power-down logic*), interfaces para memórias externas (EMIF), portas seriais, barramento de expansão (*expansion bus*) ou *host-port* e temporizadores (*timers*).

4.3 A unidade central de processamento

A CPU apresentada na Figura 4.1 é comum a todos os dispositivos da série 'C62x/C67x. Esta CPU contém:

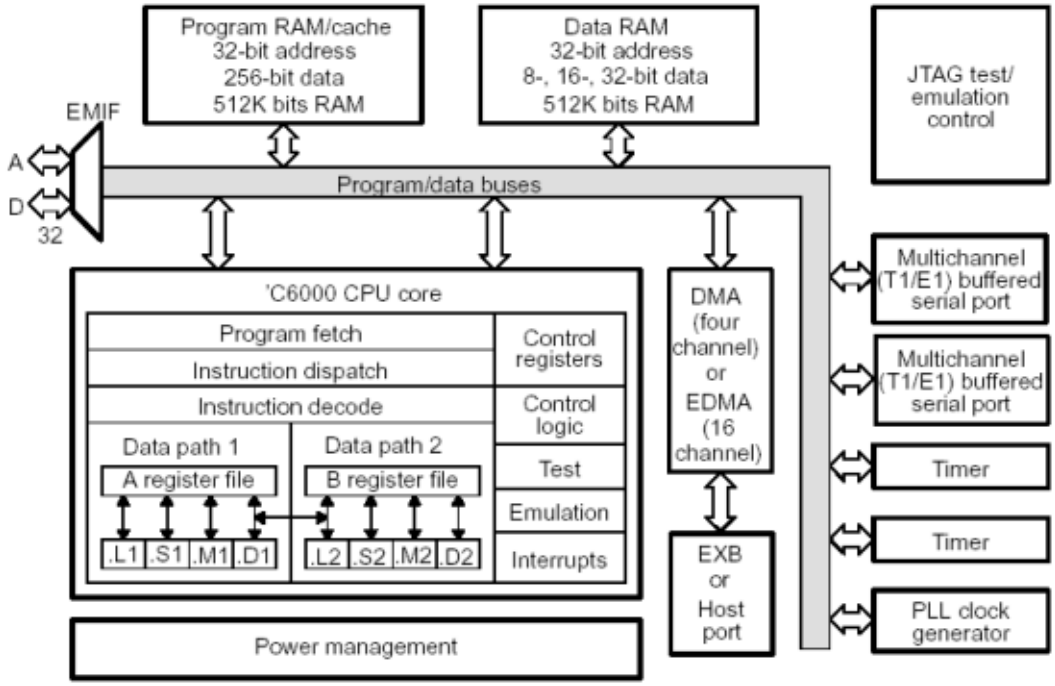

Figura 4.1: Diagrama de blocos da família TMS320C62x/C67x

- Unidade de busca de instruções (*program fetch*).
- Uma unidade de despacho de instruções.
- Uma unidade de decodificação de instruções.
- 32 registradores de 32 *bits* de propósito geral.
- Dois caminhos de dados (*datapath*), cada um com 4 unidades funcionais.
- Registradores de controle.
- Lógica de controle.
- Teste, emulação e lógica de interrupção.

Como dito, a CPU tem dois caminhos de dados (A e B) em que ocorre o processamento. Cada caminho de dados tem 4 unidades funcionais (.L, .S, .M e .D) e um banco de registradores (*register file*) contendo 16 registradores de 32 *bits*. As unidades funcionais executam operações *lógicas* (.L), *deslocamento* (.S), *multiplicação* (.M) e *endereçamento de dados* (.D). Todas as instruções, exceto as de *load* e *store* operam sobre registradores. As unidades de endereçamento de dados (.D1 e .D2) são responsáveis por todas as transferências de dados entre os registradores e a memória.

As quatro unidades funcionais de um caminho de dados tem um barramento de dados simples ligado aos registradores do outro lado. Desta forma, as unidades funcionais do caminho de dados A podem trocar informações com registradores do banco de registradores B. Os acessos cruzados a registradores $A \rightarrow B$ e $B \rightarrow A$ suportam apenas uma operação de leitura e escrita por ciclo de *clock*.

Cada unidade funcional disponível é controlada por instruções de 32 *bits*. A busca, despacho e decodificação de instruções podem entregar até 8 instruções por ciclo de *clock*. Os registradores de controle provêem os métodos para configurar e controlar os vários aspectos do processador. Os registradores de controle podem ser acessados via caminho de dados B.

O processamento VLIW começa quando um pacote (*packet*) de 256 *bits* de uma busca de instrução (*fetch*) é feito na memória principal.

4.4 Caminho de dados

A Figura 4.2 apresenta o caminho de dados da série ’C62x e a Figura 4.3 apresenta o caminho de dados da série ’C67x. Os caminhos de dados consistem de:

- Dois bancos de registradores de propósito geral (RPGs).
- Oito unidades funcionais (.L1, .L2, .S1, ,.S2, .M1, .M2, .D1 e .D2).
- Dois caminhos de dados para *loads* (LD1 e LD2).
- Dois caminhos de dados *stores* (ST1 e ST2).
- Dois caminhos de dados cruzados (1X e 2X) para comunicação entre os dois bancos de registradores.
- Dois caminhos de dados para endereços (*data address*) (DA1 e DA2).

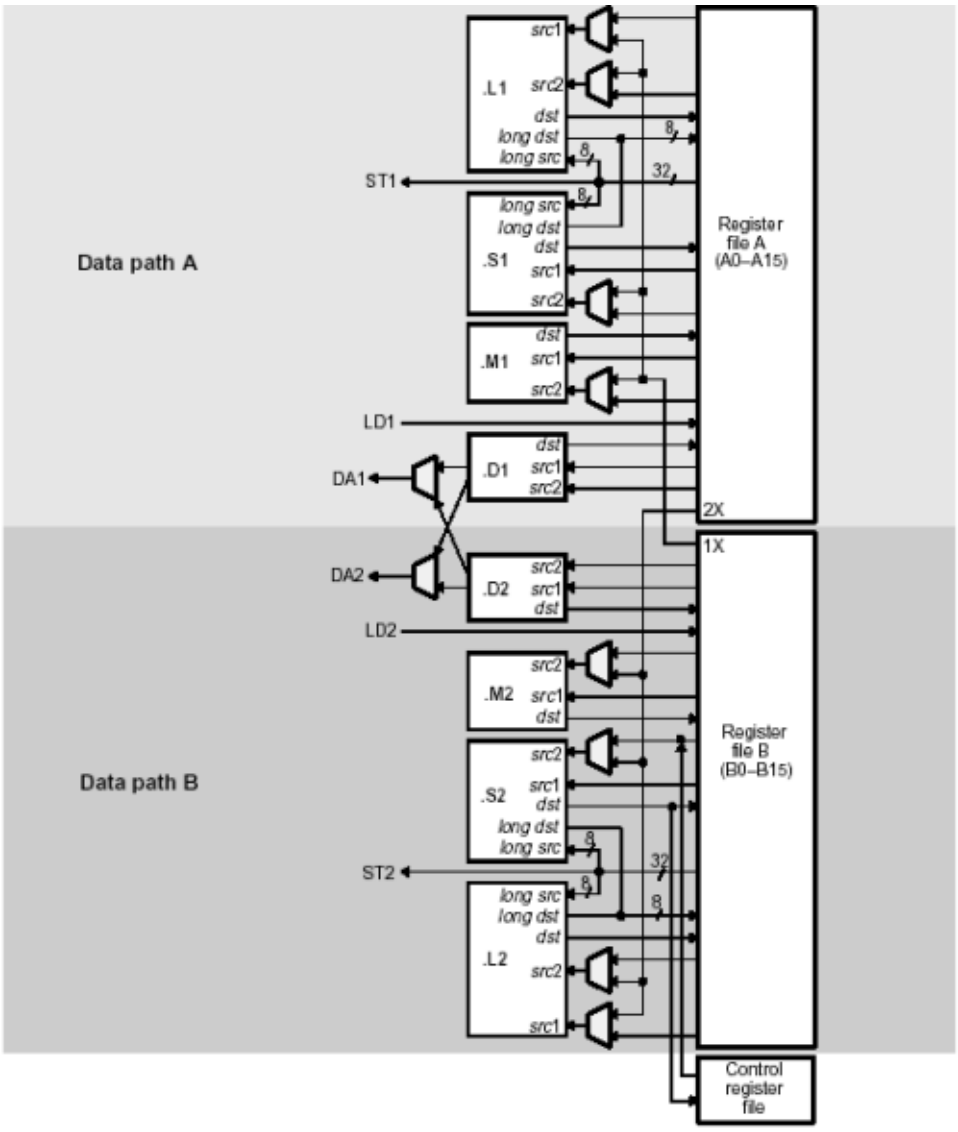

Figura 4.2: Caminho de dados do TMS320C62x

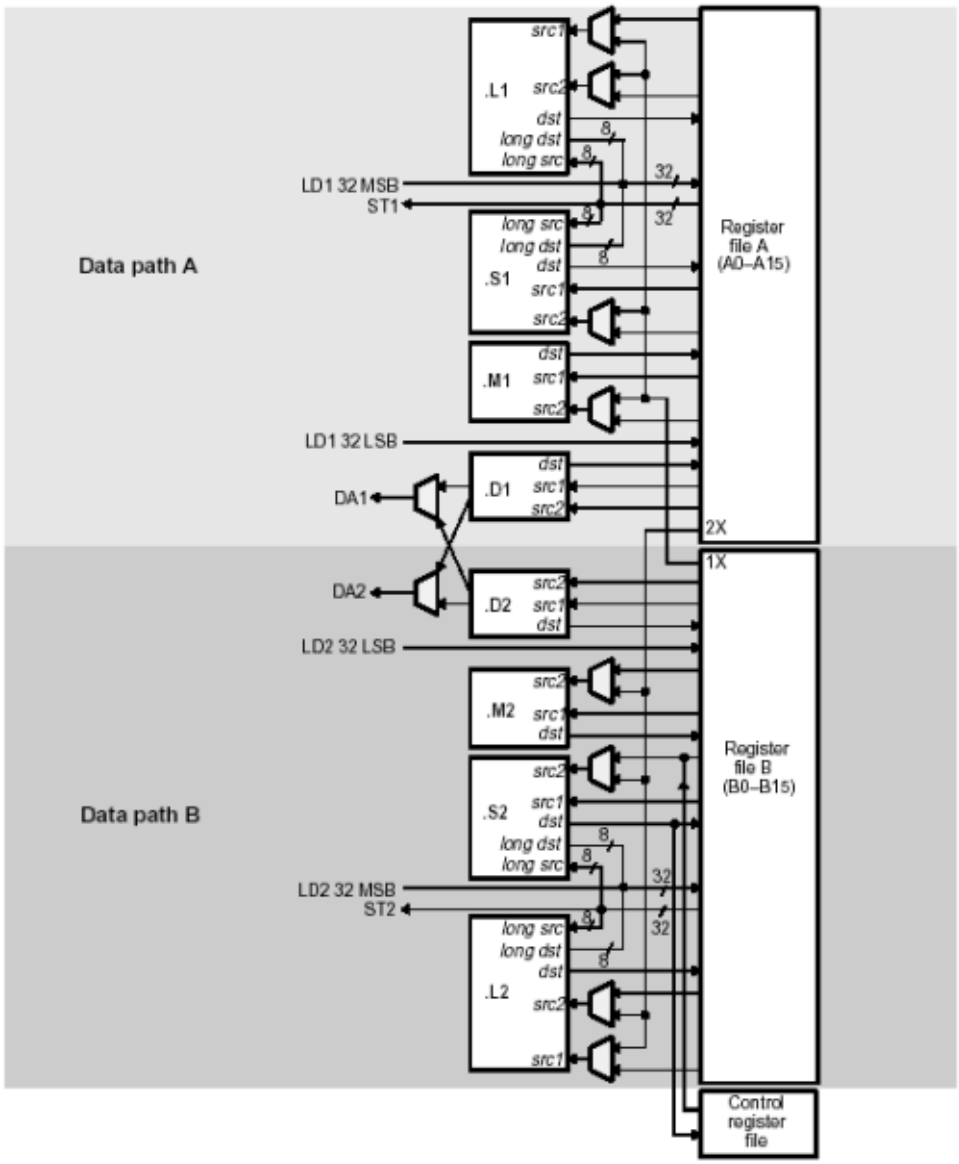

Figura 4.3: Caminho de dados do TMS320C67x

Caminho de dados de load e store

Há dois caminhos de dados de 32 *bits* para carregar dados a partir da memória para o banco de registradores: LD1 para o banco de registradores A e LD2 para o banco de registradores B. A série 'C67x tem um segundo caminho de dados para ambos os bancos (A e B), que permite a instrução LDDW (*load double word*) carregar simultaneamente dois registradores dentro do lado A e dois registradores dentro do lado B. Há também dois caminhos de 32 *bits*, ST1 e ST2, para armazenar valores na memória a partir de cada banco de registradores. Os caminhos de dados de *store* são compartilhados com os caminhos de leituras longas (*long read path*) de .S e .L.

4.5 Registradores de propósito geral

Os dois bancos de registradores disponíveis A e B contém cada um 16 registradores de 32 *bits*. Estes registradores podem ser usados para armazenar dados ou para ponteiros de endereços de dados. Os registradores A1, A2, B0, B1 e B2 podem ser usados como registradores de condição. Os registradores A4-A7 a B4-B7

podem ser usados para *endereçamento circular*.

Estes bancos de registradores suportam operações de 32 e de 40 *bits* em ponto fixo. Os dados de 32 *bits* podem estar em quaisquer registradores. Os dados de 40 *bits* estão contidos em dois registradores. Os 32 *bits* mais significativos do dado são colocados em um registrador par e os 8 *bits* restantes são colocados nos 8 *bits* menos significativos do próximo registrador acima do atual (sempre ímpar). A série 'C67x também utiliza esta abordagem para valores em ponto flutuante de precisão dupla.

4.6 As unidades funcionais

As oito unidades funcionais no caminho de dados dos dispositivos 'C62x/C67x podem ser divididas em dois grupos de quatro; cada unidade funcional em um caminho de dados é muito parecida com sua similar no caminho de dados correspondente. A maioria das linhas de dados na CPU suporta operandos de 32 *bits*, e algumas suportam operandos longos de 40 *bits*.

Cada unidade funcional tem sua própria porta de escrita de 32 *bits* no banco de registradores. Todas as unidades terminadas em 1 (L1, por exemplo) escrevem no banco de registradores A e todas as unidades terminadas em 2 escrevem no banco de registradores B. Cada unidade funcional possui portas de leitura de 32 *bits* (src1 e src2) para os operandos fonte. Além disso, quatro unidades funcionais (.L1, .L2, .S1 e .S2) possuem uma porta extra de 8 *bits* para escritas de 40 *bits* bem como para operandos de 40 *bits*. Devido ao fato de cada unidade funcional possuir sua própria porta de escrita no banco de registradores, todas as oito unidades funcionais podem ser usadas em paralelo (no mesmo ciclo de *clock*).

A Tabela 4.1 apresenta algumas características adicionais das unidades funcionais disponíveis na série TMS320C6000 de DSPs.

Unidade Funcional	**Operações de ponto fixo**	**Operações de ponto flutuante**
.L (.L1 e .L2)	Operações aritméticas de 32 e 40 *bits*, contagem de *bits* mais a esquerda para 32 e 40 *bits*, normalização para 32 e 40 *bits*, operações lógicas de 32 *bits*.	Operações aritméticas, operações de conversão $DP \rightarrow SP$, $INT \rightarrow DP$, $INT \rightarrow SP$
.S (.S1 e .S2)	Operações aritméticas de 32 *bits*, deslocamentos de 32/40 *bits*, manipulação de campos de *bits*, operações lógicas de 32 *bits*, Desvios (*branches*), geração de constantes, transferências de e para os registradores de controle (apenas .S2).	Comparações recíprocas, operações de raiz quadrada recíprocas, cálculo de valores absolutos, conversão $SP \rightarrow DP$
.M (.M1 e .M2)	Multiplicações de 16x16 *bits*.	Multiplicações de 32x32 *bits*, multiplicação de ponto flutuante
.D (.D1 e .D2)	Adição, subtração de 32 *bits*, cálculo de endereços lineares e circulares. *Loads* e *stores* com uma constante de 5 *bits* de *offset* e *loads* e *stores* com uma constante de 15 *bits* de *offset* (apenas .D2).	*Load* palavras duplas com uma constante de 5 *bits* de *offset*

Tabela 4.1: Unidades funcionais e operações realizadas

4.7 Registradores de controle

A unidade .S2 pode ler dos e escrever nos registradores de controle. Cada registrador de controle é acessado através da instrução MVC.

A Tabela 4.2 apresenta uma relação dos registradores de controle.

Abreviação	Nome e Descrição
AMR	Modo de endereçamento. Especifica o uso do modo de enderaçamento circular ou linear.
CSR	Controle de *status*. Contém o *bit* global de interrupção habilitado, *bits* de controle da *cache* entre outros.
IFR	*Flag* de interrupção. Mostra o *status* das interrupções.
ISR	Permite setar interrupções pendentes manualmente.
ICR	Permite limpar interrupções pendentes manualmente.
IER	Permite habilitar/desabilitar interrupções individuais manualmente.
ISTP	Aponta para o início da tabela de interrupções.
IRP	Contém o endereço a ser usado para retornar de uma interrupção mascarável.
NRP	Contém o endereço de retorno de uma interrupção não mascarável.
PCE1	Contador de programa, fase E1 do *pipeline*. Contém o endereço do pacote de *fetch* que contém o pacote de execução no estágio E1 do *pipeline*.

Tabela 4.2: Registradores de controle disponíveis

4.8 Modos de endereçamento

Os modos de endereçamento nos modelos 'C62x e 'C67x podem ser linear, circular utilizando BK0 e circular utilizando BK1. BK0 e BK1 são dois campos de 5 *bits* presentes no registrador de modos de endereçamento (AMR). Funcionam como identificadores para o cálculo dos tamanhos dos blocos a serem utilizados pelo endereçamento circular. O tamanho de um bloco (em *bytes*) é dado pela fórmula 2^{N+1} onde N é o valor de 5 *bits* presente em BK0 ou BK1. Exemplo: **000000** corresponde a um bloco de tamanho 2 e **000011** corresponde a um bloco de tamanho 16.

Todos os registradores podem fazer endereçamento linear. Contudo, apenas oito podem fazer endereçamento circular: A4-A7 (usados pela unidade D1) e B4-B7 (usados pela unidade D2). Nenhuma outra funcional unidade pode fazer endereçamento circular. As instruções **LDB/LDH/LDW, STB/STH/STW, ADDAB/ADDAH/ADDAW** e **SUBAB/SUBAH/SUBAW** usam o registrador AMR para determinar quais os tipos de cálculos de endereço devem ser feitos.

Endereçamento linear

Para instruções de LD/ST (*load/store*), o *offsetR* ou a *constante* utilizada simplesmente são deslocados para a esquerda por 2, 1 ou 0 (palavra, meia palavra, *byte*) seguidos do cálculo de uma adição ou subtração em relação ao registrador base (dependendo da operação especificada).

Para instruções de ADDA/SUBA (adição/subtração utilizando modo de endereçamento), o operando fonte 1 (*src1*) ou a constante utilizada como parâmetro são deslocados para a esquerda por 2, 1 ou 0 (palavra, meia palavra, *byte*) antes do cálculo ser efetuado.

Endereçamento circular

Como dito, o endereçamento circular é feito com base nos campos BK0 e BK1 do registrador de controle do modo de endereçamento (AMR).

Para instruções de LD/ST (*load/store*), o *offsetR* ou a *constante* utilizada simplesmente são deslocados para a esquerda por 2, 1 ou 0 (palavra, meia palavra, *byte*) seguidos do cálculo de uma adição ou subtração

em relação ao registrador base (dependendo da operação especificada). A diferença é que o resultado ficará no intervalo especificado pelo bloco de BK0 ou BK1.

Para facilitar o entendimento, considere o exemplo a seguir. Seja a instrução LDW (*load word*) executada com o registrador A4 no modo de endereçamento circular utilizando BK0 = 4. Desta forma, o tamanho do *buffer* circular criado é de 32 *bytes*, 16 meias palavras ou 8 palavras. No caso, o valor presente no registrador de modo de endereçamento (AMR) é **0004 0001h**.

```
LDW .D1 *++A4[9], A1

Antes de LDW:
   A4        = 0000 0100h
   A1        = xxxx xxxxh
   mem[104h] = 1234 5678h

Depois de 1 ciclo
   A4        = 0000 0104h
   A1        = xxxx xxxxh
   mem[104h] = 1234 5678h

Depois de 5 ciclos
   A4        = 0000 0104h
   A1        = 1234 5678h
   mem[104h] = 1234 5678h
```

Nota: 9h palavras são 24h *bytes*. 24h *bytes* possuem 4 *bytes* além do limite de 32 *bytes* (20h) mapeados no intervalo limitado pelo endereço de A4 e o tamanho do *buffer* (100h-11Fh). Assim, o resultado da soma de endereços é feita modularmente para poder ser mapeada neste intervalo (124h - 20h = 104h).

Sintaxe da geração de endereços para *load/store*

Como visto, o TMS320C6000 tem uma arquitetura *load/store*. Isto significa que a única forma de acessar a memória é com as instruções de *load* e *store*. A Tabela 4.3 apresenta os modos de endereçamento indireto para as instruções de *load/store*.

Endereçamento	**End. sem modificações**	**Preincremento ou predecremento**	**Posincremento ou posdecremento**
Indireto por registrador	*R	*++R e *–R	*R++ e *R–
Relativo por registrador	*+R[ucte5] *-R[ucte5]	*++R[ucte5] *–R[ucte5]	*R++[ucte5] *R–[ucte5]
Base + índice	*+R[offsetR] *-R[offsetR]	*++R[offsetR] *–R[offsetR]	*R++[offsetR] *R–[offsetR]

Tabela 4.3: Modos de endereçamento disponíveis

4.9 Interrupções

A CPU 'C62x/C67x tem 14 interrupções. São elas: **RESET'**, **NMI** *(nonmaskable interrupt)* e os sinais da CPU **INT4-INT15**. Geralmente, as interrupções **RESET'** e **NMI** são diretamente conectadas aos pinos dos dispositivos. As principais características do serviço de interrupção são:

- O pino **IACK** da CPU é usado como reconhecimento de uma requisição de interrupção.
- Os pinos **INUM0-INUM3** indicam que o vetor de interrupções está sendo utilizado.

- Os vetores de interrupção são relocáveis. Isto quer dizer que eles podem ser mapeados em partes diferentes da memória.

4.10 Conjunto de instruções

A Figura 4.4 apresenta as instruções de ponto flutuante e a Figura 4.5 apresenta as instruções de ponto fixo mapeadas em suas respectivas unidades funcionais.

.L Unit	.M Unit	.S Unit	.D Unit
ADDDP	MPYDP	ABSDP	ADDAD
ADDSP	MPYI	ABSSP	LDDW
DPINT	MPYID	CMPEQDP	
DPSP	MPYSP	CMPEQSP	
INTDP		CMPGTDP	
INTDPU		CMPGTSP	
INTSP		CMPLTDP	
INTSPU		CMPLTSP	
SPINT		RCPDP	
SPTRUNC		RCPSP	
SUBDP		RSQRDP	
SUBSP		RSQRSP	
		SPDP	

Figura 4.4: Instruções de ponto flutuante disponíveis

.L Unit	.M Unit	.S Unit		.D Unit	
ABS	MPY	ADD	SET	ADD	STB (15-bit offset)‡
ADD	MPYU	ADDK	SHL	ADDAB	STH (15-bit offset)‡
ADDU	MPYUS	ADD2	SHR	ADDAH	STW (15-bit offset)‡
AND	MPYSU	AND	SHRU	ADDAW	SUB
CMPEQ	MPYH	B disp	SHRL	LDB	SUBAB
CMPGT	MPYHU	B IRP†	SUB	LDBU	SUBAH
CMPGTU	MPYHUS	B NRP†	SUBU	LDH	SUBAW
CMPLT	MPYHSU	B reg	SUB2	LDHU	ZERO
CMPLTU	MPYHL	CLR	XOR	LDW	
LMBD	MPYHLU	EXT	ZERO	LDB (15-bit offset)‡	
MV	MPYHULS	EXTU		LDBU (15-bit offset)‡	
NEG	MPYHSLU	MV		LDH (15-bit offset)‡	
NORM	MPYLH	MVC†		LDHU (15-bit offset)‡	
NOT	MPYLHU	MVK		LDW (15-bit offset)‡	
OR	MPYLUHS	MVKH		MV	
SADD	MPYLSHU	MVKLH		STB	
SAT	SMPY	NEG		STH	
SSUB	SMPYHL	NOT		STW	
SUB	SMPYLH	OR			
SUBU	SMPYH				
SUBC					
XOR					
ZERO					

† S2 only
‡ D2 only

Figura 4.5: Instruções de ponto fixo disponíveis

A partir das Figuras 4.4 e 4.5 apresentadas, podemos definir os tipos de instruções disponíveis na família ’C62x/C67x.

A tabela 4.4 apresenta as principais instruções disponíveis no modelo ’C62x (ponto fixo).

Instrução	**Tipo**	**Descrição**
ABS	Ciclo único	Valor absoluto de um número.
ADD(U)	Ciclo único	Adição com ou sem sinal.
ADDAB, ADDAH, ADDAW	Ciclo único	Adição inteira utilizando modo de endereçamento.
ADDK	Ciclo único	Adição inteira utilizando uma contante de 16 *bits*.

ADD2	Ciclo único	Duas adições inteiras de 16 *bits* sobre as metades superiores e inferiores de um registrador.
AND	Ciclo único	**E** lógico.
B(R)	*Branch*	*Branch* utilizando despacho ou registrador.
B IRP	*Branch*	*Branch* utilizando o registrador de retorno de interrupção (IRP).
B NRP	*Branch*	*Branch* utilizando registrador de retorno de uma interrupção não mascarável (NMI Return Pointer).
CLR	Ciclo único	Limpeza (*clear*) de um campo de *bits*.
CMPEQ	Ciclo único	Comparação de igualdade.
CMPGT(U)	Ciclo único	Comparação de maior ou igual.
CMPLT(U)	Ciclo único	Comparação de menor ou igual.
EXT	Ciclo único	Extração de campos de *bits* e extensão de sinal.
EXTU	Ciclo único	Extração de campos ce *bits* e extensão do sinal 0.
IDLE	NOP	Nenhuma operação (*No Operation*) com multiciclos. Esta operação executa um número infinito de NOPs até ser terminada por uma interrupção da CPU. Quando presente em um dos *slots* de atraso de um *branch*, pode ser interrompida caso o desvio se realize.
LDB(U), LDH(U), LDW	*Load*	*Load* a partir da memória com um *offset* constante de 5 *bits* ou 15 *bits*. Além disso, a instrução pode utilizar um registrador como *offset*.
LMBD	Ciclo único	Detecção do *bit* mais a esquerda.
MPY(U, US, SU)	Multiplicação 16x16	Multiplicação dos 16 LSBs (*Least Significant Bits*) de um registrador pelos 16 LSBs de outro registrador.
MPYHL(U), MPYHULS, MPYHSLU	Multiplicação 16x16	Multiplicação dos 16 MSBs (*Most Significant Bits*) de um registrador pelos 16 MSBs de outro registrador. A multiplicação também pode ser 16 MSBs x 16 LSBs ou 16 LSBs x 16 MSBs.
MV	Ciclo único	Movimentação de conteúdo de registradores. Esta é uma pseudo-instrução.
MVC	Ciclo único	Movimentação de conteúdo entre o banco de registradores de controle e o banco de registradores de propósito geral.
MVK	Ciclo único	Movimentação de uma constante de 16 *bits* com sinal para um registrador seguido de extensão do sinal.

MVKH, MVKLH	Ciclo único	Movimentação de uma constante de 16 *bits* para os 16 MSBs de um registrador.
MVKL	Ciclo único	Extensão de sinal de uma constante de 16 *bits* seguida de um armazenamento em um registrador destino. Esta é uma pseudo-instrução.
NEG	Ciclo único	Negativo de um número. Esta é uma pseudo-instrução.
NOP	NOP	Nenhuma operação (*No Operation*). Vale observar que pode-se utilizar um argumento numérico indicando o número de NOPs a serem realizados. *Exemplo*: NOP 5 é o mesmo que 5 NOPs justapostos.
NORM	Ciclo único	Normalização de inteiros.
NOT	Ciclo único	**NÃO** lógico. Esta é uma pseudo-instrução.
OR	Ciclo único	**OU** lógico.
SADD	Ciclo único	Adição inteira com detecção de *overflow*.
SAT	Ciclo único	Conversão de um inteiro de 40 *bits* para um inteiro de 32 *bits*. Caso o valor de 40 *bits* seja maior que a maior representação possível com 32 *bits*, um *overflow* é produzido.
SET	Ciclo único	Liga um campo de *bits*.
SHL	Ciclo único	Deslocamento aritmético-lógico à esquerda.
SHR	Ciclo único	Deslocamento aritmético-lógico à direita.
SHRU	Ciclo único	Deslocamento lógico à esquerda.
SMPY(HL, LH, H)	Ciclo único (16x16)	Multiplicação de inteiros com deslocamento à esquerda e detecção de *overflow*.
SSHL	Ciclo único	Deslocamento à esquerda com detecção de *overflow*.
SSUB	Ciclo único	Subtração de inteiros com detecção de *overflow*.
STB, STH, STW	*Store*	*Store* na memória com um *offset* constante de 5 ou 15 *bits*. Além disso, a instrução pode utilizar um registrador como *offset*.
SUB(U)	Ciclo único	Subtração com ou sem sinal e sem detecção de *overflow*.
SUBAB, SUBAH, SUBAW	Ciclo único	Subtração de inteiros utilizando modo de endereçamento.
SUBC	Ciclo único	Subtração inteira condicional e deslocamento – utilizado para divisão.
SUB2	Ciclo único	Duas subtrações inteiras de 16 *bits* sobre as metades superiores e inferiores de um registrador.

XOR	Ciclo único	**OU-Exclusivo** lógico.
ZERO	Ciclo único	Zera um registrador. Esta é uma pseudo-instrução.

Tabela 4.4: Instruções de ponto fixo e sua classificação

A Tabela 4.5 apresenta as principais instruções disponíveis no modelo 'C67x (ponto flutuante). OBS: DP significa precisão dupla e SP significa precisão simples.

Instrução	**Tipo**	**Descrição**
ABSDP	2 ciclos DP	Valor absoluto de um número em precisão dupla.
ABSSP	Ciclo único	Valor absoluto de um número em precisão simples.
ADDAD	Ciclo único	Adição inteira usando modo de endereçamento com palavra dupla (*doubleword*).
ADDDP	ADDDP/SUBDP	Adição de com precisão dupla.
ADDSP	4 ciclos	Adição de com precisão simples.
CMPEQDP	Comparação DP	Comparação de igualdade com precisão dupla.
CMQEQSP	Ciclo único	Comparação de igualdade com precisão simples.
CMPGTDP	Comparação DP	Comparação de maior ou igual com precisão dupla.
CNPGTSP	Ciclo único	Comparação de maior ou igual com precisão simples.
CMPLTDP	Comparação DP	Comparação de menor ou igual com precisão dupla.
CMPLTSP	Ciclo único	Comparação de menor ou igual com precisão simples.
DPINT	4 ciclos	Conversão de um valor com precisão dupla para um valor inteiro.
DPSP	4 ciclos	Conversão de um valor com precisão dupla para um valor com precisão simples.
PTRUNC	4 ciclos	Conversão de um valor com precisão dupla para um valor inteiro com truncamento.
INTDP(U)	INTDP	Conversão de um inteiro para um valor com precisão dupla.
INTSP(U)	4 ciclos	Conversão de um inteiro para um valor com precisão simples.
LDDW	*Load*	*Load*. *Load* de uma palavra dupla a partir da memória com um *offset* constante ou a partir do conteúdo de um registrador como *offset*.
MPYDP	MPYDP	Multiplicação com precisão dupla.
MPYI	MPYI	Multiplicação de 32 *bits* com resultado de 32 *bits*.

MPYID	MPYID	Multiplicação de 32 *bits* com resultado de 64 *bits*.
MPYSP	4 ciclos	Multiplicação com precisão simples.
RCPDP	2 ciclos DP	Aproximação recíproca com precisão dupla. Esta instrução aplica o algoritmo de Newton-Rhapson e é utilizada para extender a precisão da mantissa.
RCPSP	Ciclo único	Aproximação recíproca com precisão simples.
RSQRDP	2 ciclos DP	Aproximação recíproca da raiz quadrada com precisão dupla.
RSQRSP	Ciclo único	Aproximação recíproca da raiz quadrada com precisão simples.
SPDP	2 ciclos DP	Conversão de um valor com precisão simples para um valor com precisão dupla.
SPINT	4 ciclos	Conversão de um valor com precisão simples para inteiro.
SPTRUNC	4 ciclos	Conversão de um valor com precisão simples para inteiro com truncamento.
SUBDP	ADDDP/SUBDP	Subtração com precisão dupla.
SUBSP	4 ciclos	Subtração com precisão simples.

Tabela 4.5: Instruções de ponto flutuante e sua classificação

A Tabela 4.6 apresenta os símbolos utilizados na definição das instruções. Estas instruções são adicionais ao conjunto presente no modelos de ponto fixo.

As Figuras 4.6 e 4.7 apresentam os mapeamentos dos *opcodes*.

4.10.1 Slots de atraso

A execução de instruções de ponto fixo ou de ponto flutuante podem ser definidas em termos de *slots* de atraso (*delay*). O número de *delay slots* é equivalente ao número de ciclos necessários após a leitura dos operandos fonte para que o resultado esteja disponível para leitura.

Todas as instruções que são comuns na série 'C62x/C67x tem a latência 1 na unidade funcional. Isto significa que uma nova instrução pode ser iniciada na unidade funcional a cada ciclo de *clock*.

Ponto fixo

Para uma instrução do tipo simples (*single-cycle type*) (tal como ADD), os operandos fonte lidos no ciclo i produzem um resultado que pode ser lido no ciclo $i + 1$. Para uma instrução do tipo 2 ciclos (*2-cycle type*) (tal como MPY), os operandos fontes lidos no ciclo i produzem o resultado no ciclo $i + 2$. A Figura 4.8 apresenta os *slots* de atraso para cada tipo de instrução de ponto fixo.

Símbolo	Significado
baseR	Endereço do registrador base
creg	Campo de 3 *bits* especificando um registrador de condição
cst	Constante
csta	Constante a
cstb	Constante b
dst	Destino
h	*Bit* MVK ou MVKH
ld/st	campo do opcode de *load/store*
mode	Modo de endereçamento utilizado
offsetR	Registrador de offset
op	Campo opcode indicando uma instrução única
p	Execução parelela
r	*Bit* LDDW
rsv	Reservado
s	Seleciona o lado A ou B para o destino
src2	Fonte 1
src1	Fonte 2
ucstn	Constante de **n bits** sem sinal
x	Use caminho de dados cruzado (*crosspath*) para o fonte 2 (*src2*)
y	Seleciona a unidade funcional .D1 ou .D2
z	Teste de igualdade com zero.

Tabela 4.6: Símbolos utilizados na definição das instruções

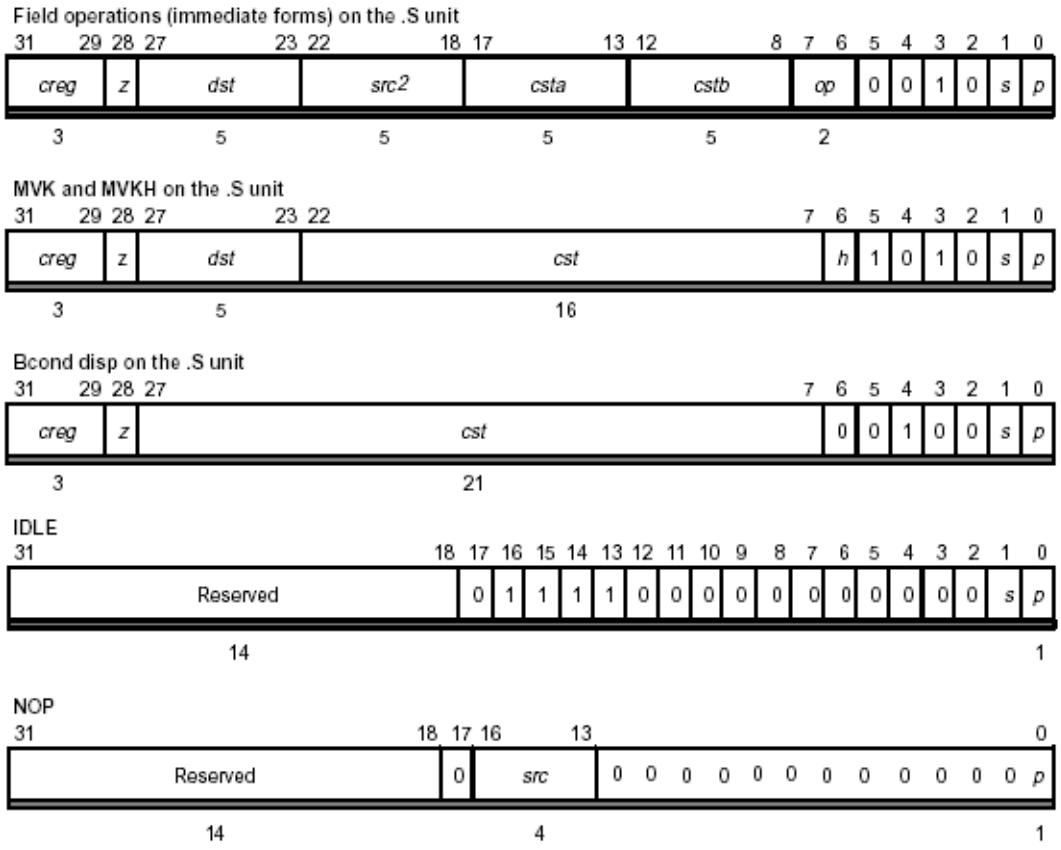

Figura 4.6: Mapeamentos dos *opcodes*

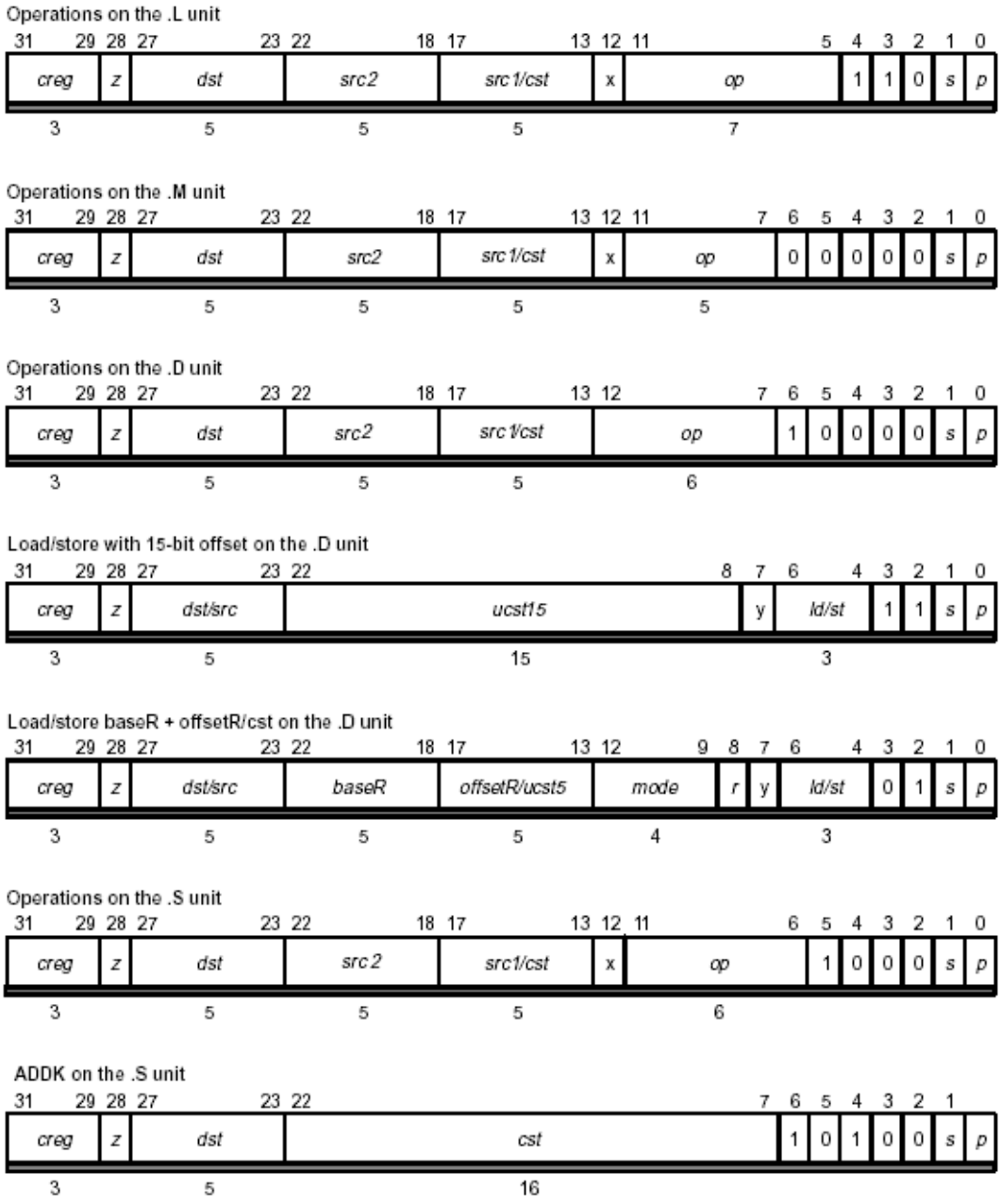

Figura 4.7: Mapeamentos dos *opcodes* – continuação

Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles†	Write Cycles†	Branch Taken†
NOP (no operation)	0	1			
Store	0	1	i	i	
Single cycle	0	1	i	i	
Multiply (16 × 16)	1	1	i	i + 1	
Load	4	1	i	i, i + 4§	
Branch	5	1	i‡		i + 5

† Cycle i is in the E1 pipeline phase.
‡ The branch to label, branch to IRP, and branch to NRP instructions instruction does not read any registers.
§ The write on cycle i + 4 uses a separate write port from other .D unit instructions.

Figura 4.8: *Slots* de atraso para as instruções de ponto fixo

Ponto flutuante

Para uma instrução do tipo simples (*single-cycle type*) (tal como CMPGT2), os operandos fonte lidos no ciclo i produzem um resultado que pode ser lido no ciclo $i+1$. Para uma instrução do tipo 2 ciclos (*2-cycle type*) (tal como AVGU4), os operandos fontes lidos no ciclo i produzem o resultado no ciclo $i+2$. Para uma instrução do tipo 4 ciclos (*4-cycle type*) (tal como DOTP2), os operandos fontes lidos no ciclo i produzem o resultado no ciclo $i+4$. A Figura 4.9 apresenta os *slots* de atraso para cada tipo de instrução de ponto flutuante.

Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles†	Write Cycles†
Single cycle	0	1	i	i
2-cycle DP	1	1	i	i, i + 1
4-cycle	3	1	i	i + 3
INTDP	4	1	i	i + 3, i + 4
Load	4	1	i	i, i + 4‡
DP compare	1	2	i, i + 1	1 + 1
ADDDP/SUBDP	6	2	i, i + 1	i + 5, i + 6
MPYI	8	4	i, i + 1, 1 + 2, i + 3	i + 8
MPYID	9	4	i, i + 1, 1 + 2, i + 3	i + 8, i + 9
MPYDP	9	4	i, i + 1, 1 + 2, i + 3	i + 8, i + 9

† Cycle i is in the E1 pipeline phase.
‡ A write on cycle i + 4 uses a separate write port from other instructions on the .D unit.

Figura 4.9: *Slots* de atraso para as instruções de ponto flutuante

4.10.2 As instruções individualmente

A descrição sobre o funcionamento de cada instrução, opcodes, comportamento entre outros detalhes podem ser encontrados em [Texas Instruments, 2004a].

Para termos uma idéia de como é a sintaxe das instruções, a seguir são apresentadas algumas delas.

- **ABS**. Valor absoluto de um número. **Sintaxe**: ABS (.unit) src2, dst. Onde *.unit* pode ser as unidades .L1 ou .L2, *src2* é o operando fonte e *dst* é o destinho. **Tipo**: ciclo único.
- **ADD(U)**. Adição inteira cou ou sem sinal. **Sintaxe**: ADD (.unit) src1, src2, dst. ou ADDU (.L1 ou .L2) src1, src2, dst ou ADD (.D1 ou .D2) src2, src1, dst. Onde *.unit* pode ser as unidades .L1,.L2, .S1 ou .S2, *src1, src2* são os operandos fontes e *dst* é o destinho. **Tipo**: ciclo único.
- **B**. *Branch* utilizando despacho (*label*). **Sintaxe**: B (.unit) label. Onde *.unit* pode ser as unidades .S1 ou .S2. **Tipo**: *branch*.

Capítulo 5

Periféricos

Adicionalmente ao *chip* de memória, o TMS320C62x e TMS320C67x contém periféricos para comunicação com memórias fora do *chip*, coprocessadores, outros processadores e dispositivos seriais. Estes periféricos são brevemente descritos nas seções a seguir. É importante ressaltar que cada dispositivo da série C6000 tem apenas alguns destes periféricos. Para mais informações sobre periféricos na família TMS320 consulte [Texas Instruments, 2004d].

5.1 DMA

Os acessos diretos a memória (DMA – Direct Memory Access) transferem dados entre regiões na memória sem intervenção da CPU. O DMA permite o movimento de dados a partir de e para a memória interna, periféricos internos e/ou dispositivos externos trabalhando em *background* à operação da CPU. O DMA tem 4 canais independentes e programáveis. Um quinto canal auxiliar permite requisições de serviço a partir das interfaces HPI (ver seção 5.3) ou XB (ver seção 5.4). DMA está implementado nos dispositivos 'C6201, 'C6202 e 'C6701. As características de DMA implementadas permitem:

- **Operação em *background***: o DMA opera independentemente da CPU.
- **Alto *throughput***: elementos podem ser transferidos na taxa de *clock* da CPU.
- **4 canais**: DMA pode transferir até quatro blocos de dados independentes.
- **Canal auxiliar**: este canal permite que uma porta de *host* (HPI ou XB) faça requisições no espaço de memória da CPU.
- **Operação de divisão (*split*)**: um canal pode ser usado simultaneamente para fazer uma receber e uma transmitir dados para ou a partir de dois periféricos e a memória. Desta forma, este canal atua como dois DMAs.
- **Transferência multi-quadros**: cada transferência de blocos consiste de múltiplos quadros de tamanhos programável.
- **Prioridade programável**: cada canal tem uma prioridade programável em relação à CPU.
- **Geração programável de endereços**: cada fonte e destino de um canal de DMA endereça registradores que podem ter índices configuráveis para cada transferência de leitura ou escrita. O endereço pode permanecer constante, incrementado, decrementado ou ser ajustado para um valor qualquer. Isto é muito utilizado em ordenação multicanal [Texas Instruments, 2004d].
- **Intervalo de 32 *bits* completo**: DMA pode acessar qualquer região no mapeamento de memória seja ela memória de dados e/ou de programa *on-chip*, periféricos *on-chip* e a interface de memória externa (EMIF).
- **Largura de transferência programável**: cada canal pode ser configurado para transmitir bytes, meias palavras (halfwords) ou palavras inteiras.

- **Autoinicialização**: uma vez que um canal tenha terminado de transferir um bloco de dados, ele pode reiniciar a si mesmo e começar a transferir outro bloco de dados.
- **Sincronização de eventos**: cada leitura, escrita ou transferência de blocos pode ser iniciada a partir de eventos selecionados (sincronização por evento).
- **Geração de interrupções**: ao completar a transferência de um bloco de dados ou mesmo sob várias condições de erro cada canal de DMA pode enviar uma interrupção à CPU.

5.2 EDMA

Presente nos dispositivos 'C6211 o controlador de DMA melhorado (EDMA – *Enhanced Direct Memory Access*), tal o DMA normal, controla a transferência de dados entre regiões da memória sem a intervenção da CPU. A diferença aparece no número de canais independentes: 16.

5.3 HPI

Presente nos dispositivos 'C6201, 'C6211 e 'C6701, a interface de *host port* (HPI) é uma porta paralela de 16 *bits* de largura em que um *host* (outro DSP, processador entre outras opções) pode acessar diretamente o espaço de memória da CPU. O *host* funciona como um mestre (*master*) para a interface facilitando o acesso. O *host* e a CPU podem trocar informações via memória interna ou externa podendo também acessar diretamente os periféricos mapeados na memória.

5.4 XB

Presente nos dispositivos 'C6202 o barramento de expansão (XP – *Expansion Bus*), serve como uma substituição para a interface HPI e como um complemento para a interface de memória externa (EMIF). Com um segundo barramento para dispositivos de I/O, a sobrecarga sobre a interface EMIF será reduzida e a vazão (*throughput*) dos dados pode ser aumentado.

5.5 EMIF

Presente em todos os dispositivos, a interface de memória externa (EMIF – *External Memory Interface*) suporta uma grande quantidade de dispositivos externos permitindo espaço adicional de memória além do que é fornecido *on-chip*. Os tipos de memória suportados são: SRAM, SBSRAM, DRAM, SRRAM além de dispositivos SRAMs e ROMs assíncronos.

5.6 BCL

Presente em todos os dispositivos, o BCL é a lógica de configuração de *boot* (BCL – *Boot Configuration Logic*). O BCL provê uma grande variedade de configurações de *boot* para inicialização dos dispositivos. Estas configurações determinam que ações o 'C62x/C67x deve executar após a reinicialização do dispositivo. As configurações de *boot* (setadas por pinos externos) determinam:

- O mapeamento de memória do dispositivo.
- O tipo de memória externa.
- O processo de *boot* utilizado para inicializar a memória no endereço zero antes que a CPU tenha permissão de execução entre outras ações.

5.7 McBSP

Presente em todos os dispositivos, a porta serial multicanal bufferizada (McBSP – *Multichannel buffered serial port*) é baseada na interface de porta serial padrão encontrada nas plataformas TMS320C2000 e ’C5000. Estas interfaces padrão provêm:

- Comunicação *full-duplex*.
- Registradores de dados duplamente *bufferizados* permitindo um fluxo de dados contínuo.
- *Framing* e *clocking* para recepção e transmissão independentes.

Por sua vez, o McBSP implementa características adicionais tais como:

- Transmissão e recepção multicanal de até 128 canais.
- Maior liberdade na escolha no tamanho dos elementos a serem transmitidos. Os tamanhos em de um elemento, em *bits*, podem ser de 8, 12, 16, 20, 24 ou 32.
- Polaridade programável tanto para sincronização de quadros quanto para *clock* de dados.

5.8 Timers

Os dispositivos da família ’C62x/C67x têm dois temporizadores (*timers*) de propósito geral de 32 *bits* que podem ser usados para:

- Eventos de tempo.
- Eventos de contagem.
- Geração de pulsos.
- Interrupção da CPU.
- Enviar eventos de sincronização para o controlador DMA.

Com um *clock* interno, um temporizador pode sinalizar um conversor A/D externo para iniciar uma conversão ou ele pode dar um sinal para o controlador DMA começar uma determinada transferência.

Capítulo 6

Características do paralelismo

Este capítulo apresenta as principais característica do paralelismo na família de DSPs TMS320. É mostrado a forma de realização de operações em paralelo, as operações condicionais, restrições no acesso a recursos de *hardware* por instruções concorrentes, o modo de operação do *pipeline* e finalmente, algumas considerações sobre previsão de desvio..

6.1 Operações em paralelo

Como já apresentado, oito instruções são buscadas a cada *fetch* de programa. Isto constitui o que é chamado de pacote de *fecth* (*fetch packet*). O formato básico de um pacote de *fetch* é mostrado na Figura 6.1. Como cada palavra disponível tem 32 *bits*, o tamanho de um pacote de *fetch* é de 256 *bits*.

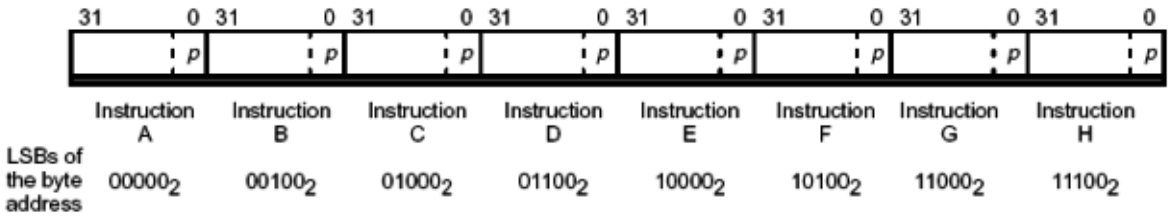

Figura 6.1: Formato básico de um pacote de *fetch*

A execução de instruções individuais é parcialmente controlada por um *bit* em cada instrução, o *bit p* (*p-bit*). O *bit p* (bit 0 ou menos significativo de cada instrução) determina se uma instrução executa em paralelo com outra instrução. Se o *bit p* da instrução i está setado com 1 então a instrução i executa em paralelo (no mesmo ciclo) com a instrução $i + 1$. Cada instrução executando em paralelo constitui um pacote de execução (*execution packet*). Um pacote de execução tem, no máximo, 8 instruções e todas devem utilizar unidades funcionais diferentes.

Um pacote de execução não pode ultrapassar 8 palavras. Desta forma, o último *bit p* de um pacote de *fetch* deve ser setado para 0. Há três tipos de configuração possíveis para os *bits p* em um pacote de *fetch*. Estas três configurações resultam nas seguintes seqüências de execução para oito instruções:

- Totalmente serial
- Totalmente paralelo
- Parcialmente serial

6.1.1 Totalmente serial

A Figura 6.2 apresenta o padrão de *bits p* para um pacote de *fetch* totalmente serial. A execução das instruções é apresentada na Tabela 6.1.

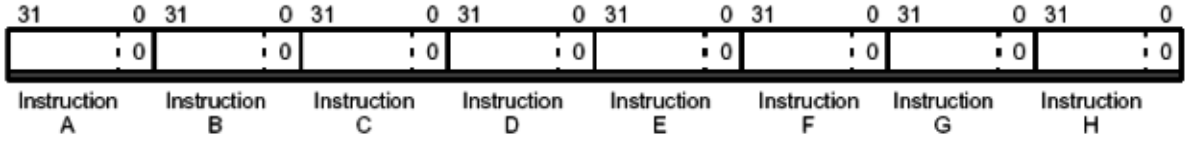

Figura 6.2: Padrão do *bit p* de um pacote de *fetch* totalmente serial

Ciclo / Pacote de execução	Instruções
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H

Tabela 6.1: Todas as oito instruções executando serialmente

6.1.2 Totalmente paralelo

A Figura 6.3 apresenta o padrão de *bits p* para um pacote de *fetch* totalmente paralelo. A execução das instruções é apresentada na Tabela 6.2.

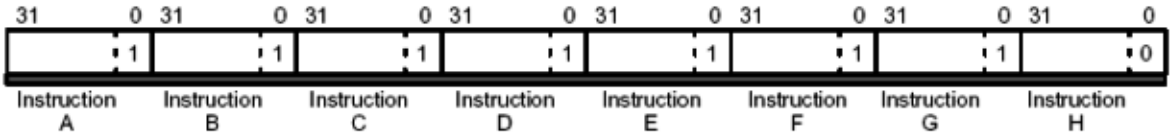

Figura 6.3: Padrão do *bit p* de um pacote de *fetch* totalmente paralelo

6.1.3 Parcialmente serial

A Figura 6.4 apresenta o padrão de *bits p* para um pacote de *fetch* parcialmente serial. A execução das instruções é apresentada na Tabela 6.3.

6.1.4 Exemplo de código paralelo

Os caracteres || significam que uma instrução é executada em paralelo com a instrução anterior. O código para a Figura 6.4 poderia ser representado como segue:

```
    instrução A

    instrução B

    instrução C
|| instrução D
|| instrução E

    instrução F
|| instrução G
```

Ciclo / Pacote de execução	Instruções
1	A B C D E F G H

Tabela 6.2: Todas as oito instruções executando em paralelo

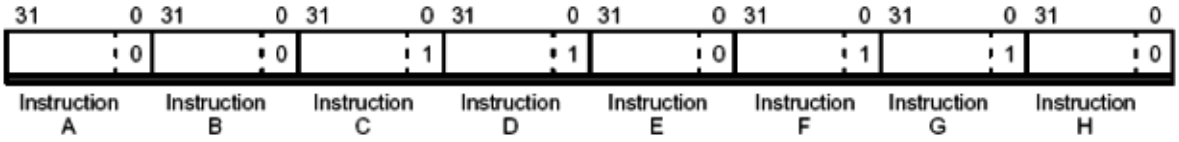

Figura 6.4: Padrão do *bit p* de um pacote de *fetch* parcialmente serial

```
|| instrução H
```

6.1.5 *Branches* no meio de um pacote de execução

Caso um *branch* apareça no meio de um pacote de execução, todas as instruções anteriores a este *branch* são ignoradas. No exemplo da seção 6.1.4, se um *branch* apontando para o endereço de D acontece então apenas D e E são executadas para o pacote de execução C, D, E. As instruções A e B também são ignoradas pois estão em pacotes de execução anteriores ao alvo do *branch*.

6.2 Operações condidionais

Todas as instruções *podem* ser condicionais. A condição é controlada pelos 3 *bits* (*creg*) do *opcode* e um bit (*z*) que especifica um teste para zero ou não zero. Os 4 *bits* mais significativos de todos os *opcodes* são *creg* e *z*. O registrador de condição especificado é testado no estágio E1 do *pipeline* para todas as instruções. Para mais informações do *pipeline* veja a seção 6.4.

Instruções condicionais são representadas no código usando-se colchetes, [ ], limitando o registrador de condição desejado. Caso a execução seja não condicional (padrão) então os três *bits* de *creg* e o *bit z* são 0. O seguinte pacote tem duas instruções de ADD executadas em paralelo. O primeiro ADD é condicional sobre B0 sendo não-zero. O segundo ADD é condicional sobre B0 sendo zero. O caracter "!" indica o inverso da condição.

```
   [B0]  ADD .L1 A1, A2, A3
|| [!B0] ADD .L2 B1, B2, B3
```

As instruções acima são mutuamente exclusivas. Isto significa que apenas uma das duas irá executar. Quando duas instruções compartilham o mesmo recurso elas não podem ser executadas em paralelo mesmo que sejam mutuamente exclusivas.

6.3 Restrições de recursos

Duas instruções dentro de um mesmo pacote de execução podem usar os mesmos recursos. No entanto, duas instruções não podem escrever o mesmo registrador no mesmo ciclo. As subseções a seguir apresentam como cada instrução pode usar cada um dos recursos possíveis.

6.3.1 Restrições sobre instruções usando a mesma unidade funcional

Duas instruções no mesmo pacote de instrução não podem utilizar a mesma unidade funcional.

O seguinte pacote de execução é inválido:

```
   ADD .S1 A0, A1, A2; \ .S1 é usado por ambas instruções
|| SHR .S1 A3, 15, A4; /
```

Ciclo / Pacote de execução	Instruções
1	A
2	B
3	C D E
4	F G H

Tabela 6.3: Parte das instruções executando em serial e parte em paralelo

O seguinte pacote de execução é válido:

```
   ADD .L1 A0, A1, A2; \ duas unidades funcionais diferentes
|| SHR .S1 A3, 15, A4; / são utilizadas
```

6.3.2 Restrições sobre os caminhos cruzados (*crosspaths*) (1X e 2X)

Uma unidade (.S, .L ou .M) por caminho de dados, por pacote de execução, pode ler um operando fonte do banco de registradores oposto via caminho de dados 1X e 2X.

O seguinte pacote de execução é inválido:

```
   ADD.L1X A0, B1, A1; \ caminho 1X é usado por ambas instruções
|| MPY.M1X A4, B4, A5; /
```

O seguinte pacote de execução é válido:

```
   ADD.L1X A0, B1, A1; \ instruções utilizam 1X e 2X
|| MPY.M2X B4, A4, B2; /
```

6.3.3 Restrições sobre *loads* e *stores*

Loads e *stores* podem usar ponteiros para um banco de registradores enquanto estão carregando de ou armazenando no outro banco de registradores. Duas instruções de *load/store* usando um *destino/fonte* a partir do mesmo bando de registradores não pode ser colocado no mesmo pacote de execução. O registrador de endereço precisa estar do mesmo lado em que a unidade .D utilizada estiver.

O seguinte pacote de execução é inválido:

```
   LDW.D1 *A0, A1; \ .D2 precisa usar o registrador de endereço
|| LDW.D2 *A2, B2; / do banco de registradores B
```

O seguinte pacote de execução é válido:

```
   LDW.D1 *A0, A1; \ registradores de endereço localizaodos
|| LDW.D2 *B0, B2; / no banco de registradores correto
```

Dois *loads* e/ou *stores* carregando para e/ou armazendo a partir do mesmo banco de registradores não podem ser colocados no mesmo pacote de execução.

O seguinte pacote de execução é inválido:

```
   LDW.D1 *A4,  A5; \ carregando para e armazendo a partir do mesmo
|| STW.D2  A6, *B4; / banco de registradores
```

O seguinte pacote de execução é válido:

```
   LDW.D1 *A4,  B5; \ carregando para, e armazendo a partir
|| STW.D2  A6, *B4; / de diferentes bancos de registradores

   LDW.D1 *A0,  B2; \ carregando para bancos de regstradores
|| LDW.D2 *B0,  A1; / diferentes
```

6.3.4 Restrições sobre dados de 40 *bits*

Devido ao fato de as unidades .S e .L compartilharem uma porta de leitura de um registrador para operandos de 40 *bits* e uma porta de escrita para um registrador para resultados de 40 *bits*, apenas um resultado de 40 *bits* pode ser colocado em um banco de registradores por pacote de execução.

O seguinte pacote de execução é inválido:

```
   ADD.L1 A5:A4, A1, A3:A2; \ duas escritas de 40-bits no banco de
|| SHL.S1    A8, A9, A7:A6; / registradores A
```

O seguinte pacote de execução é válido:

```
   ADD.L1 A5:A4, A1, A3:A2; \ uma escrita de 40-bits em cada banco
|| SHL.S2    B8, B9, B7:B6; / de registradores
```

6.3.5 Restrições sobre leituras de registradores

Mais que 4 leituras no mesmo registrador não podem ocorrer no mesmo ciclo. Registradores condicionais não são incluídos nesta contagem.

A seguinte seqüência de códigos é inválida:

```
   MPY .M1 A1, A1, A4; cinco leituras do registrador A1
|| ADD .L1 A1, A1, A5
|| SUB .D1 A1, A2, A3

   MPY .M1  A1, A1, A4; cinco leituras do registrador A1
|| ADD .L1  A1, A1, A5
|| SUB .D2x A1, B2, B3
```

A seguinte seqüência de códigos é válida:

```
        MPY .M1 A1, A1, A4; apenas 4 leituras de A1
|| [A1] ADD .L1 A0, A1, A5;
||      SUB .D1 A1, A2, A3;
```

6.3.6 Restrições sobre escritas em registradores

Duas instruções não podem escrever no mesmo registrador ao mesmo tempo. Duas instruções com o mesmo destino podem ser agendadas em paraleo desde que elas não escrevam no registrador no mesmo ciclo. Por exemplo, uma instrução MPY instanciada no ciclo *i* seguida de um ADD no ciclo *i + 1* não podem escrever no mesmo registrador dado que ambas as instruções escrevem o resultado no ciclo *i + 1*. Desta forma, a seguinte seqüência de código é inválida:

```
MPY .M1 A0, A1, A2
ADD .L1 A4, A5, A2
```

No entanto, a seguinte seqüência é válida:

```
   MPY .M1 A0, A1, A2
|| ADD .L1 A4, A5, A2
```

6.4 Pipeline

Esta seção apresenta o funcionamento geral do *pipeline* da série TMS320. Cada estágio do *pipeline* é mostrado com exemplos claros com os vários tipos de instrução da série. Isto tende facilitar o entendimento do *pipeline* diante das várias configurações possíveis a um determinado conjunto de instruções sendo executado.

6.4.1 Visão geral do funcionamento do *pipeline*

As fases do *pipeline* são agrupadas em três estágios:

- Busca de instrução (*fetch*)
- Decodificação (*decode*)
- Execução (*execute*)

Todas as instruções no conjunto de instruções das famílias C62x e C67x seguem o seguinte fluxo de estágios de *pipeline*: busca (*fetch*), decodificação (*decode*) e execução (*execute*). O estágio de busca de instrução possui quatro fases e a de decodificação possui duas fases. Todas as instruções, independentemente de seu tipo, passam por todas as fases do estágio de busca e de decodificação. No estágio de execução é que há uma diferenciação entre as instruções, há número variável de fases e tarefas diferentes depedendo do tipo de instrução. Os estágios do *pipeline* da família C62x são mostrados na Figura 6.5. A diferença nos estágios do *pipeline* da família C67x está na execução, onde no C67x é mais longo que no C62x.

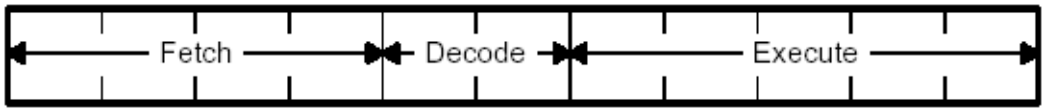

Figura 6.5: Estágios do *pipeline* da família C62x.

Busca de instrução (*fetch*)

As fases de busca de instrução são:

- **PG:** Cálculo de endereço da instrução (*program address generate*)
- **PS:** Envio de endereço da instrução para a memória (*program address send*)
- **PW:** Espera por conclusão de acesso à memória (*program access ready wait*)
- **PR:** Recebimento do pacote de busca (*fetch packet* – FP) de instruções (*program fetch packet received*)

Os dispositivos das famílias C62x e C67x usam o pacote de busca (*fetch packet*) com oito instruções. Todas as oito instruções seguem o processamento da busca juntas, ao longo das fases PG, PS, PW e PR. A Figura 6.6 mostra as fases da busca na ordem em que são executadas. A Figura 6.7 é um diagrama funcional com o fluxo de instruções ao longo das fases da busca. Durante a fase PG o endereço de programa é gerado na CPU. Na fase PS o endereço de programa é enviado para a memória. Na fase PW a leitura de memória é realizada. Finalmente, na fase PR o pacote de busca (FP) é recebido pela CPU. A Figura 6.8 exibe pacotes de busca fluindo ao longo das fases do estágio de busca do *pipeline*. Na Figura 6.8 o primeiro pacote de busca (em PR) é dividido em quatro pacotes de execução (*execute packets* – EP), e o segundo e terceiro (em PW e PS) contêm dois pacotes de execução cada. O últmo pacote de busca (em PG) contém um único pacote de execução de oito instruções.

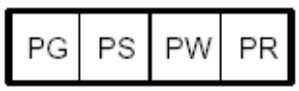

Figura 6.6: Ordem de execução das fases no estágio de busca de instrução.

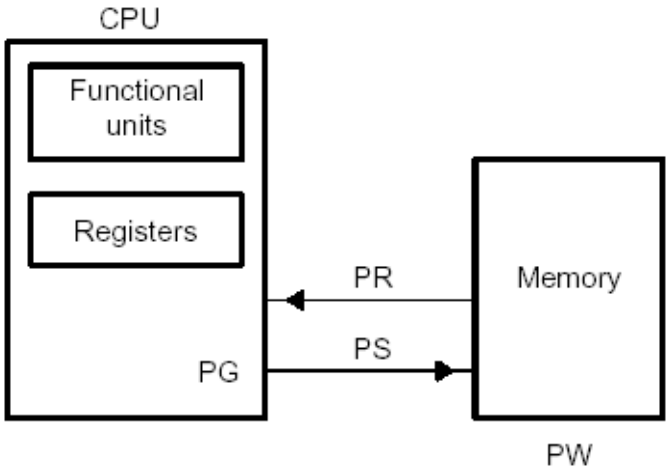

Figura 6.7: Diagrama funcional com o fluxo de instruções ao longo das fases da busca.

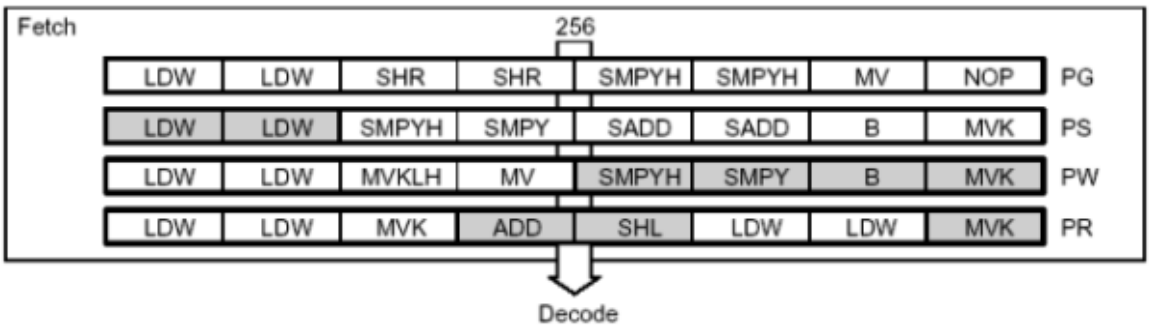

Figura 6.8: Fluxo de pacotes de busca ao longo das fases da busca.

Decodificação (*decode*)

As fases de decodificação do *pipeline* são:

- **DP:** Despacho de instrução (*instruction dispatch*)
- **DC:** Decodificação de instrução (*instruction decode*)

Na fase DP do *pipeline* os pacotes de busca são divididos em pacotes de execução. Pacotes de execução consistem de conjuntos de uma a oito instruções que podem executar em paralelo. Durante a fase DP as instruções em um pacote de execução são associadas às unidades funcionais apropriadas. Na fase DC os registradores fonte, registradores destino e caminhos associados são decodificados para a execução das instruções nas unidades funcionais.

A Figura 6.9 mostra as fases de decodificação na ordem em que são executadas. A Figura 6.10 mostra um pacote de busca que contém dois pacotes de execução, ilustrando como eles são processados durante o estágio de decodificação do *pipeline*. As últimas seis instruções do pacote de busca são paralelas e formam um pacote de execução. Este pacote de execução está na fase de despacho do estágio de decodificação. As setas indicam cada instrução associada a uma unidade funcional para execução durante o mesmo ciclo. A instrução **NOP** no oitavo *slot* do FP não é despachada para uma unidade funcional porque não há execução associada a ela.

Os primeiros dois *slots* do pacote de busca, que aparecem sombreados, representam um pacote de execução de duas instruções paralelas que foram despachadas no ciclo anterior. Este pacote de execução contém duas instruções **MPY** que estão agora em fase de decodificação. Não há instruções decodificadas para as unidades funcionais .L, .S e .D na situação ilustrada.

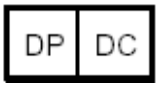

Figura 6.9: Fluxo de pacotes de busca ao longo das fases da decodificação.

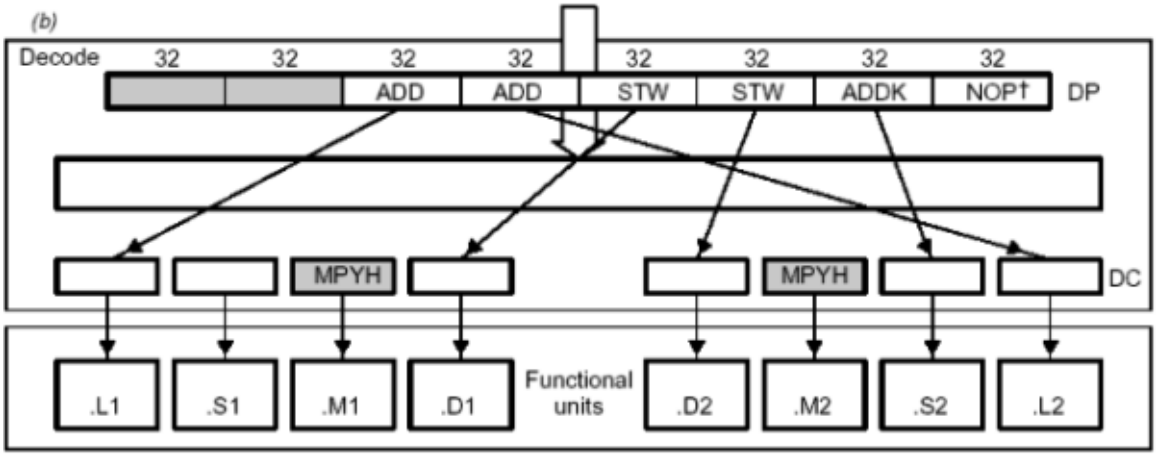

Figura 6.10: Processamento de pacotes nas fases de decodificação.

Execução (*execute*)

O estágio de execução do *pipeline* em ponto fixo (nos dispositivos da família C62x) é subdividido em cinco fases (E1 – E5). Já o estágio de execução do *pipeline* em ponto flutuante (nos dispositivos da família C67x) é subdividido em dez fases (E1 – E10). Os tipos diferentes de instruções requerem diferente número de fases de execução para serem concluídas. A execução dos diferentes tipos de instruções no *pipeline* é descrito na Seção 6.4.2, Execução no *pipeline* de tipos de instrução. A Figura 6.11 mostra as fases de execução no *pipeline* de ponto fixo e a Figura 6.12 mostra as fases de execução no *pipeline* de ponto flutuante. A Figura 6.13 mostra a parte do diagrama de bloco funcional no qual ocorre o estágio de execução.

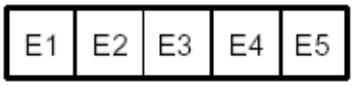

Figura 6.11: Ordem de execução das fases no estágio de execução dos C62x.

Resumo do funcionamento do *pipeline*

Na Figura 6.14 pode-se obter uma visão geral das fases e da ordem em que elas ocorrem no *pipeline* dos dispositivos da família C62x. Já Figura 6.15 pode-se visualizar as fases no *pipeline* dos C67x.

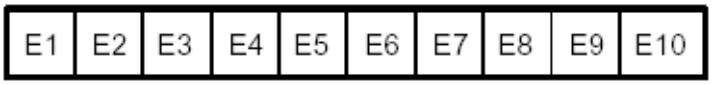

Figura 6.12: Ordem de execução das fases no estágio de execução dos C67x.

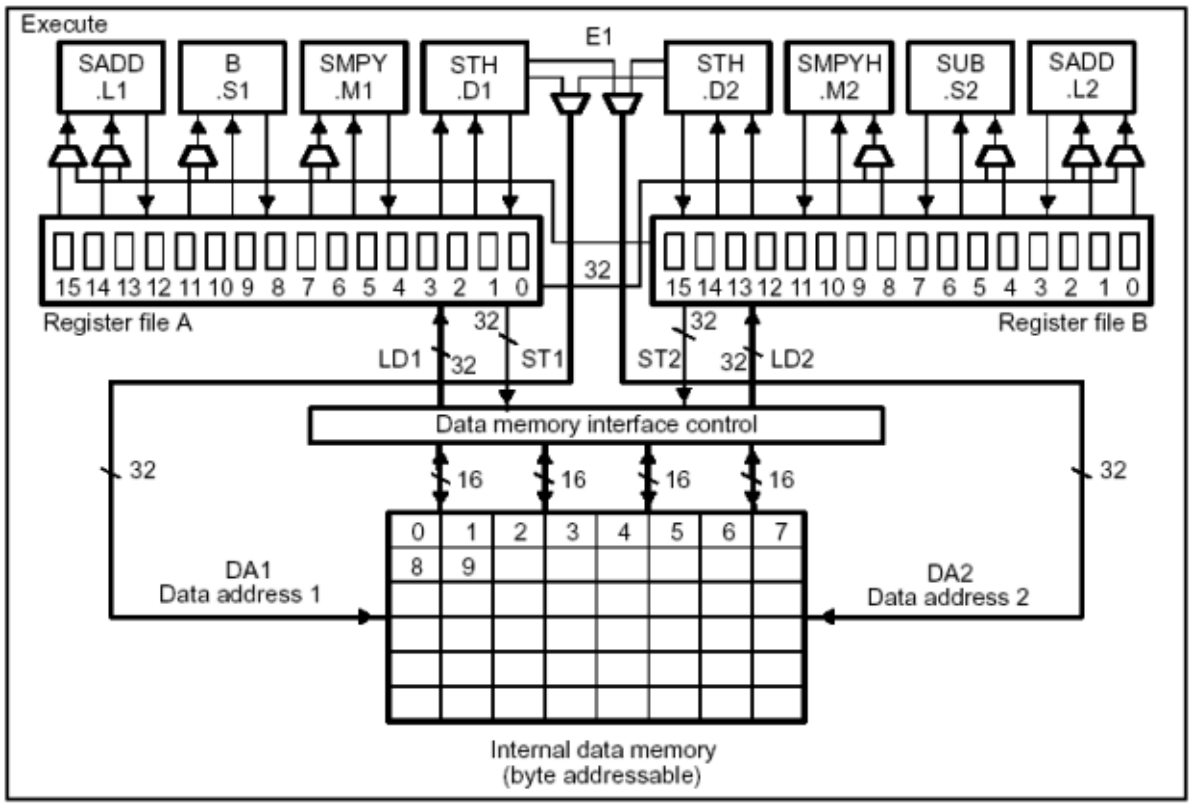

Figura 6.13: Diagrama de bloco funcional do estágio de execução.

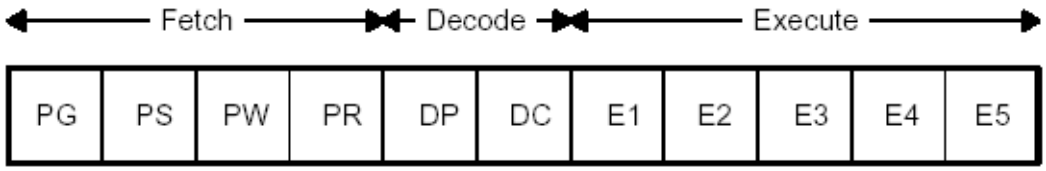

Figura 6.14: Todas as fases do *pipeline* nos dispositivos da família C62x.

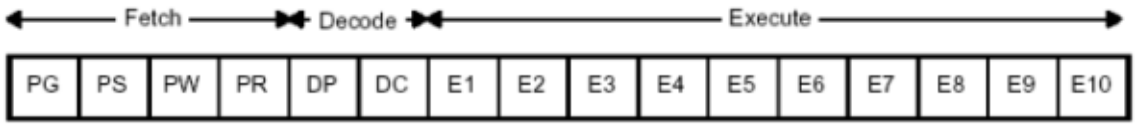

Figura 6.15: Todas as fases do *pipeline* nos dispositivos da família C67x.

A Figura 6.16 mostra um exemplo do fluxo do *pipeline*. Neste exemplo têm-se pacotes de busca contendo oito instruções paralelas, ou seja, em cada pacote encontra-se apenas um pacote de execução. O exemplo exposto aplica-se aos dispositivos da família C62x, mas podemos extendê-lo aos C67x se considerar os pacotes de busca com 10 fases de execução.

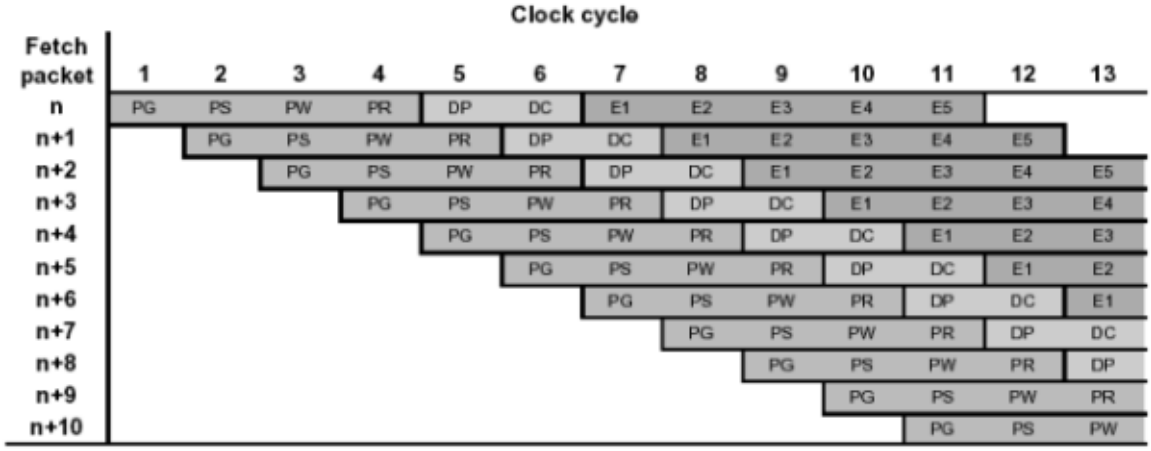

Figura 6.16: Funcionamento do *pipeline* com ponto fixo

Resume-se o funcionamento do *pipeline* de ponto fixo na Tabela 6.4.

Estágio	**Fase**	**Símbolo**	**Durante esta fase**	**Tipo de instrução concluída**
Busca	Cálculo de endereço	**PG**	O endereço do pacote de busca é determinado.	
	Envio de endereço à memória	**PS**	O endereço do pacore de busca é enviado para a memória.	
	Acesso à memória	**PW**	É realizado um acesso a memória de programa.	
	Recebimento das instruções.	**PR**	O pacote de busca encontra-se na CPU.	
Decodificação	Despacho	**DP**	O próximo pacote de execução no pacote de busca é determinado e enviado para a unidade funcional apropriada para ser decodificado.	
	Decodificação	**DC**	Instruções são decodificadas nas unidades funcionais.	

Execução	Execução 1	**E1**	Para todos os tipos de instrução, as condições para as instruções são avaliadas e os operandos são lidos. Para instruções *load* e *store*, o cálculo de endereço é realizado e as modificações de endereço são escritas no banco de registradores. Para instruções *branch*, o pacote de busca na fase PG é afetado. Para instruções de ciclo único, os resultados são escritos no banco de registradores.	**Ciclo único**
	Execução 2	**E2**	Para instruções *load*, o endereço é enviado para a memória. Para instruções *store*, o endereço e os dados são enviados para a memória. Nas instruções de ciclo único onde há controle de *overflow*, é definido o *bit* SAT do registrador de controle de estado (CSR) caso tenha ocorrido um *overflow*. Para instruções de multiplicação 16 x 16, os resultados são escritos no banco de registradores.	***Multiply 16 x 16***
	Execução 3	**E3**	São realizados acessos a memória de dados. Nas instruções de multiplicação com controle de *overflow*, o *bit* SAT no registrador de controle de estado (CSR) é definido caso tenha ocorrido o *overflow*.	***Store***
	Execução 4	**E4**	Para instruções *load*, o dado é entregue a CPU.	
	Execução 5	**E5**	Para instruções *load*, o dado é escrito no registrador.	***Load***

Tabela 6.4: Resumo das fases do *pipeline* com ponto fixo

Resume-se na Tabela 6.5 o funcionamento do *pipeline* de ponto flutuante.

Estágio	**Fase**	**Símbolo**	**Durante esta fase**	**Tipo de instrução concluída**
Busca	Cálculo de endereço	**PG**	O endereço do pacote de busca é definido.	
	Envio de endereço à memória	**PS**	O endereço é enviado à memória.	
	Acesso à memória	**PW**	O acesso à memória é realizado.	
	Recebimento das instruções	**PR**	O pacote de busca é entregue a CPU.	
Decodificação	Despacho	**DP**	O próximo pacote de execução do pacote de busca corrente é determinado e enviado para a unidade funcional apropriada para ser decodificado.	

	Decodificação	**DC**	As instruções são decodificadas nas unidades funcionais.	
Execução	Execução 1	**E1**	Para todas os tipos de instrução, as condições para as instruções são avaliadas e os operandos são lidos. Para instruções de *load* e *store*, são realizados os cálculos de endereço, as modificações de endereços e são escritos no banco de registradores. Para instruções *branch*, altera o pacote de busca na fase PG, caso necessário. Para instruções de ciclo único, os resultados são escritos no banco de registradores. Para instruções de comparação, adição, subtração e multiplicação com precisão dupla, os 32 *bits* de mais baixa ordem são lidos. Para todas as outras instruções, os fontes são lidos. Para as instruções de duplo ciclo com dupla precisão, os 32 *bits* de mais baixa ordem do resultado são escritos no banco de registradores.	**Ciclo único**
	Execução 2	**E2**	Para todas as instruções, o endereço é enviado para a memória. Para a instrução *store*, o endereço e os dados são enviados para a memória. Instruções de ciclo único com controle de *overflow*, é definido o *bit* SAT do registrador de controle de estado (CSR) caso tenha ocorrido um *overflow*. Para instruções de multiplicação, de duplo ciclo com precisão dupla e de comparação com precisão dupla, os resultados são escritos no banco de registradores. Para instruções de comparação, adição e subtração com precisão dupla, os 32 *bits* de mais alta ordem dos fontes são lidos. Para a instrução de multiplicação com precisão dupla, os 32 *bits* de mais baixa ordem de *src1* e os 32 bits de mais alta ordem de *src2* são lidos. Para as instruções MPYI e MPYID, os fontes são lidos.	**Multiplicação 16 x 16, duplo ciclo com precisão dupla, comparação de duplo ciclo**
	Execução 3	**E3**	Acessos à memória de dados são realizados. Nas instruções de multiplicação com controle de *overflow*, o *bit* SAT do CSR é definido caso tenha ocorrido *overflow*. Para a instrução MPYDP, os 32 *bits* de mais alta ordem de *src1* e os 32 *bits* de mais baixa ordem de *src2* são lidos. Para as instruções MPYI e MPYID, os fontes são lidos.	***Store***

	Execução 4	**E4**	Para instruções *load*, o dado chega a CPU. Para instruções MPYI e MPYID, os fontes são lidos. Para a instrução MPYDP, os 32 *bits* de mais alta ordem dos fontes são lidos. Para instruções de quatro ciclos, os resultados são escritos no banco de registradores. Para a instrução INTDP, os 32 *bits* de mais baixa ordem do resultado são escritos no banco de registradores.	**4-ciclos**
	Execução 5	**E5**	Para instruções *load*, o dado é escrito no banco de registradores. Para a instrução INTDP, os 32 bits de mais alta ordem do resultado são escritos no banco de registradores.	***Load,*** **INTDP**
	Execução 6	**E6**	Para instruções de adição e subtração com precisão dupla, os 32 *bits* de mais baixa ordem do resultado são escritos no banco de registradores.	
	Execução 7	**E7**	Para instruções de adição e subtração com precisão dupla, os 32 *bits* de mais alta ordem do resultado são escritos no banco de registradores.	**ADDDP, SUBDP**
	Execução 8	**E8**	Nada é lido ou escrito	
	Execução 9	**E9**	Para a instrução MPYI, o resultado é escrito no banco de registradores. Para as instruções MPYDP e MPYID, os 32 *bits* de mais baixa ordem do resultado são escritos no banco de registradores.	**MPYI**
	Execução 10	**E10**	Para as instruções MPYDP e MPYID, os 32 *bits* de mais alta ordem do resultado são escritos no banco de registradores.	**MPYDP, MPYID**

Tabela 6.5: Resumo das fases do *pipeline* com ponto flutuante

Um diagrama de bloco funcional com todos os estágios pode ser visto na Figura 6.17.

O funcionamento do *pipeline* é baseado nos ciclos da CPU. Um ciclo de CPU é o período durante o qual um particular pacote de execução está em uma particular fase do *pipeline*. O limite do ciclo de CPU sempre ocorre no limite do ciclo do clock.

Na fase DC da Figura 6.17, uma caixa está vazia porque havia um **NOP** no pacote de busca em DC e não é necessário alocar uma unidade funcional para um **NOP**. Finalmente, a figura mostra seis unidades funcionais processando código durante o mesmo ciclo de um *pipeline*.

Os registradores usados pelas instruções em E1 estão sombreados na Figura 6.17. Os multiplexadores usados para a entrada dos operandos nas unidades funcionais também estão sombreados na figura. Os caminhos cruzados em negrito são usados pelas instruções **MPY**.

A maioria das instruções nos dispositivos das famílias C62x e C67x são instrução de ciclo único, o que significa que elas possuem apenas uma fase de execução (E1).

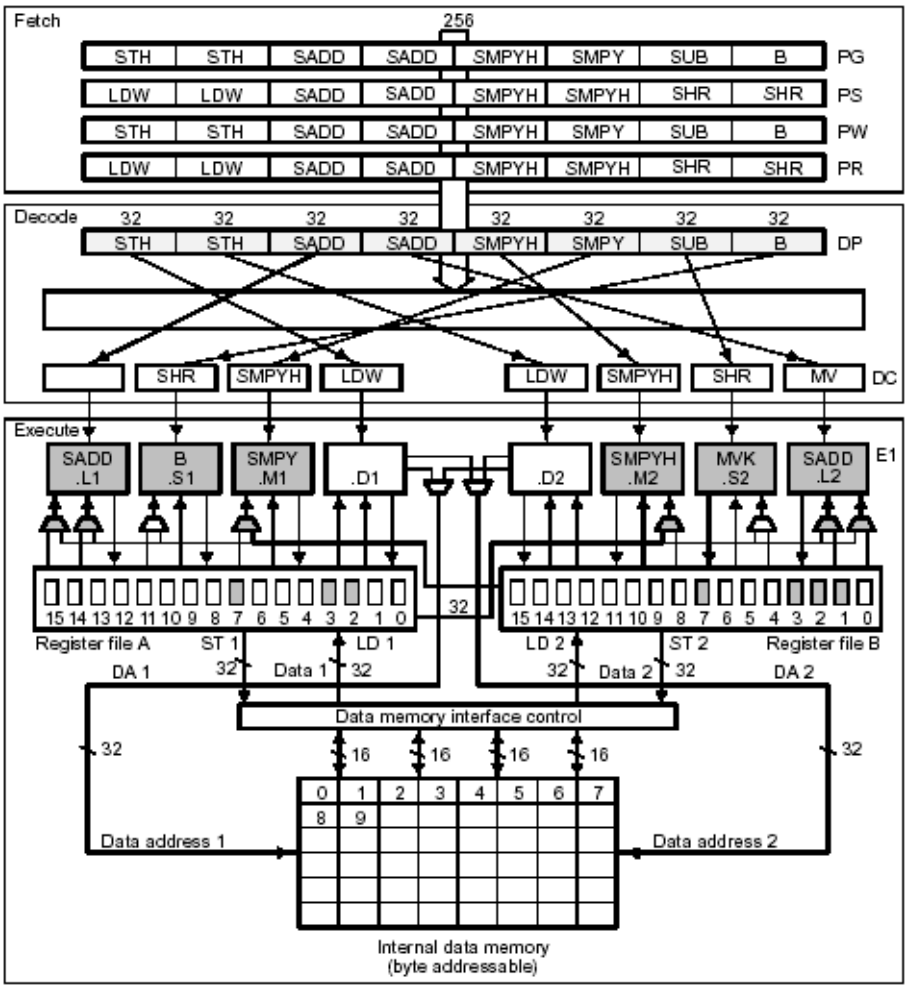

Figura 6.17: Diagrama de bloco funcional.

6.4.2 Execução no *pipeline* de tipos de instrução

A execução de instruções pode ser definida em termos de *delay slots*. Um *delay slot* é um ciclo de CPU que ocorre depois da primeira fase de execução (E1) de uma instrução. Os resultados de instruções com *delay slots* não estão disponíveis até o fim do último *delay slot*. Por exemplo, uma instrução de multiplicação tem um *delay slot*, que significa que um ciclo de CPU deve ocorrer antes que os resultados da multiplicação estejam disponíveis para ser usado pela instrução subsequente.

Se uma instrução possui latência de unidade funcional multiciclo, a unidade funcional é bloqueado por determinado número de ciclos. Se uma instrução for despachada para uma unidade funcional que esteja bloqueada, resultados serão indefinidos. Se uma instrução com latência possui uma condição que é avaliada como falsa durante E1, ele irá bloquear a unidade funcional pelos ciclos subseqüentes.

As instruções nos dispositivos da família TMS320C62x podem ser categorizadas em seis tipos de instruções, do ponto de vista do *pipeline*. Cinco destes são apresentados na Tabela 6.6 na página 45 (**NOP** não é incluído na tabela), a qual é um mapeamento das operações que ocorrem em cada fase do estágio de execução para os diferentes tipos de instruções. Os *delay slots* associados a cada tipo de instrução são listados na penúltima linha da tabela e a latência da unidade funcional na última.

	Ciclo único	**Multiplicação 16 x 16**	***Store***	***Load***	***Branch***
E1	Computa resultado e escreve no registrador	Lê operandos e inicia coputação	Calcula endereço	Calcula endereço	Código alvo em PG
E2		Obtém resultado e escreve no registrador	Envia endereço e dado para a memória	Envia endereço para a memória	
E3			Acesso a memória	Acesso a memória	
E4				Envia dado para a CPU	
E5				Escreve dado no registrador	
	0	1	0	4	5
	1	1	1	1	1

Tabela 6.6: Fases do estágio de execução por tipo de instrução

Já nos C67x as instruções podem ser categorizadas em 14 tipos de instruções, do ponto de vista do *pipeline*. Cinco destes são apresentados na Tabela 6.6 e o restante nas Tabelas 6.7 e 6.8.

	2-ciclo DP	**4-ciclo**	**INTDP**	**Comparação DP**
E1	Calcula os *bits* de mais baixa ordem do resultado e escreve no banco de registradores	Lê fontes e começa cálculo	Lê fontes e começa cálculo	Lê os *bits* de mais baixa ordem e começa cálculo.
E2	Calcula os *bits* de mais alta ordem do resultado e escreve no banco de registradores	Continua cálculo	Continua cálculo	Lê os *bits* de mais alta ordem dos fontes, finaliza cálculo e escreve o resultado no registrador
E3		Continua cálculo	Continua cálculo	
E4		Completa o cálculo e escreve o resultado no registrador	Continua cálculo e escreve os *bits* de mais baixa ordem do resultado no registrador	
E5			Completa cálculo e escreve os *bits* de mais alta ordem do resultado no registrador	
E6				
E7				
E8				
E9				
E10				
	1	3	4	1
	1	1	1	1

Tabela 6.7: Fases do estágio de execução por tipo de instrução no C67x

	ADDDP/SUBDP	MPYI	MPYID	MPYDP
E1	Lê os *bits* de mais baixa ordem dos fontes e começa cálculo	Lê os fontes e começa cálculo	Lê os fontes e começa cálculo	Lê os *bits* de mais baixa ordem dos fontes e começa cálculo
E2	Lê os *bits* de mais alta ordem e continua cálculo	Lê fontes e continua cálculo	Lê fontes e continua cálculo	Lê os *bits* de mais baixa ordem de *src1* e os de mais alta ordem de *src2* e continua cálculo
E3	Continua cálculo	Lê fontes e continua cálculo	Lê fontes e continua cálculo	Lê os *bits* de mais baixa ordem de *src2* e os de mais alta ordem de *src1* e continua cálculo
E4	Continua cálculo	Lê fontes e continua cálculo	Lê fontes e continua cálculo	Lê os *bits* de mais alta ordem dos fontes e continua cálculo
E5	Continua cálculo	Continua cálculo	Continua cálculo	Continua cálculo
E6	Calcula os *bits* de mais baixa ordem do resultado e escreve no registrador	Continua cálculo	Continua cálculo	Continua cálculo
E7	Calcula os *bits* de mais alta ordem do resultado e escreve no registrador	Continua cálculo	Continua cálculo	Continua cálculo
E8		Continua cálculo	Continua cálculo	Continua cálculo
E9		Conclui cálculo e escreve o resultado no registrador	Continua cálculo e escreve os *bits* de mais baixa ordem no registrador	Continua cálculo e escreve os *bits* de mais baixa ordem no registrador
E10			Conclui cálculo e escreve os *bits* de mais alta ordem no registrador	Conclui cálculo e escreve os *bits* de mais alta ordem no registrador
	6	8	9	9
	1	1	1	1

Tabela 6.8: Fases do estágio de execução por tipo de instrução no C67x

Uma instrução dos seguintes tipos executando no ciclo i possui as seguintes restrições:

Comparação DP Nenhuma outra instrução pode usar a unidade funcional nos ciclos i e $i + 1$.

ADDDP/SUBDP Nenhuma outra instrução pode usar a unidade funcional nos ciclos i e $i + 1$.

MPYI Nenhuma outra instrução pode usar a unidade funcional nos ciclos i, $i + 1$, $i + 2$ e $i + 3$.

MPYID Nenhuma outra instrução pode usar a unidade funcional nos ciclos i, $i + 1$, $i + 2$ e $i + 3$.

MPYDP Nenhuma outra instrução pode usar a unidade funcional nos ciclos i, $i+1$, $i+2$ e $i+3$.

Se um caminho cruzado é usado para ler um fonte em uma instrução com latência de unidade funcional multiciclo, é necessário assegurar que nenhuma outra instrução que usa o mesmo caminho cruzado está executando no mesmo lado.

Outra restrição existe pelo fato das instruções possuírem variável número de *delay slots* e necessitarem ler e escrever nas entradas por variáveis números de ciclos (varia de acordo com a instrução). Um restrição de leitura ou escrita ocorre quando quando duas instruções na mesma unidade funcional tentam ler ou escrever, respectivamente, no banco de registradores nos ciclos subseqüentes. Maiores detalhes a respeito das restrições podem ser vistos na Seção 6.4.17.

6.4.3 Instruções de ciclo único

As instruções de ciclo único completam sua execução durante a fase E1 do *pipeline*. Nesta categoria encontram-se, basicamente, as instruções aritméticas e lógicas mais simples em ponto fixo.

A Figura 6.18 mostra o diagrama de execução das instruções de ciclo único. Os operandos são lidos, a operação é realizada e os resultados são escritos no registrador, tudo durante E1. As instruções de ciclo único não possuem *delay slots*.

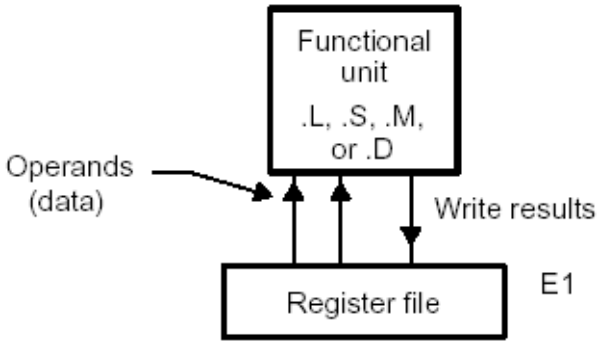

Figura 6.18: Diagrama de execução das instruções de ciclo único.

6.4.4 Multiplicação 16 x 16

As instruções de multiplicação 16 x 16 usam as fases E1 e E2 do *pipeline* para completar suas operações. Elas efetuam a multiplicação de dois valores de 16 *bits*, gerando um resultado de 32 *bits*.

A Figura 6.19 mostra as operações que ocorrem na multiplicação 16 x 16. Na fase E1 os operandos são lidos e começa a multiplicação. Na fase E2 a multiplicação termina e o resultado é escrito no registrador de destino. As instruções de multiplicação possuem um *delay slot*.

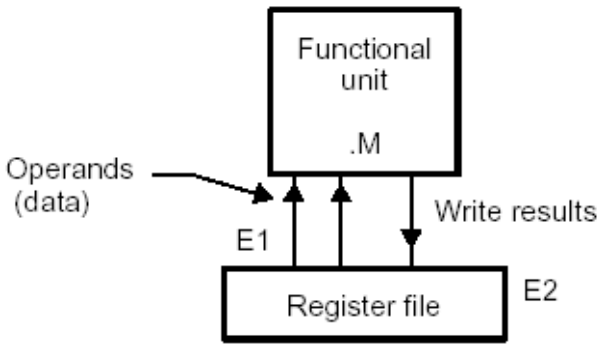

Figura 6.19: Diagrama de execução das instruções de multiplicação.

6.4.5 Instruções *store*

As instruções *store* requerem as fases E1, E2 e E3 para completar suas operações. A Figura 6.20 mostra as operações que ocorrem nas fases do *pipeline* para as instruções *store*. Na fase E1, o endereço é calculado. Na fase E2, os dados e o endereço de destino é enviado para a memória de dados. Na fase E3, a escrita em memória é realizada. A modificação de endereço é realizada em E1. Mesmo as instruções *store* concluindo sua execução na fase E3 do *pipeline*, elas não possuem *delay slots*.

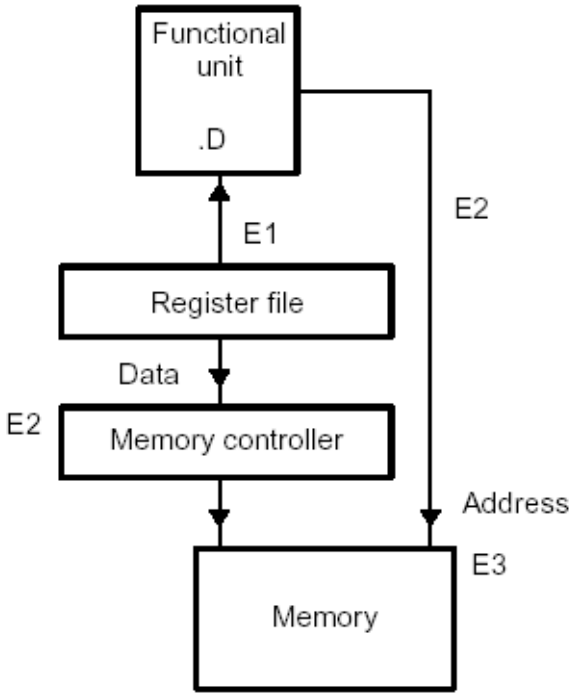

Figura 6.20: Diagrama de execução das instruções de *store*.

Quando você executa um *load* e um *store* na mesma posição de memória, estas regras são aplicadas ($i = ciclo$):

- Quando um *load* é executado antes de um *store*, o valor antigo é carregado e o novo é armazenado.
- Quando um *store* é executado antes que um *load*, o novo valor é armazenado e o novo valor é carregado.
- Quando as instruções são executadas em paralelo, o valor antigo é carregado primeiro e então o novo valor é armazenado, mas ambos ocorrem na mesma fase.

6.4.6 Instruções *load*

Carregamento de dados requer cinco fases de execução do *pipeline* para completar suas operações. A Figura 6.21 mostra as operações que ocorrem nas fases do *pipeline* para um *load*. Na fase E1, o ponteiro para o endereço dos dados é modificado no registrador. Na fase E2, o endereço dos dados é enviado para a memória de dados. Na fase E3, é realizada a leitura de memória. Na fase E4, os dados são recebidos pela CPU. Finalmente, na fase E5, os dados são armazenados no registrador. Por causa do fato dos dados só serem escritos no registrador na última fase, instruções *load* possuem quatro *delay slots*. Por causa dos ponteiros de resultado serem escritos no registrador em E1, não há *delay slots* associados com modificação de endereço.

No seguinte código, ponteiros de resultados são escritos no registrador A4 na primeira fase de execução do *pipeline* e os dados são escritos no registrador A3 na quinta fase de execução.

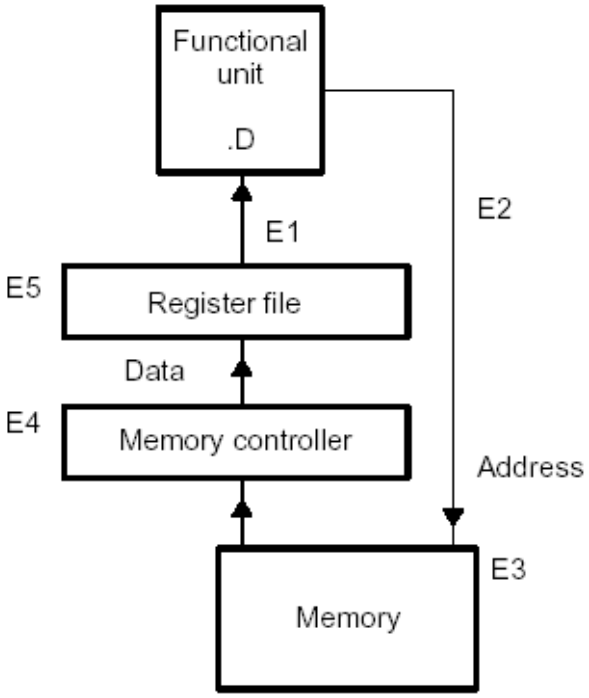

Figura 6.21: Diagrama de execução das instruções de *load*.

```
LDW .Dw  *A4++, A3
```

Pelo fato de um *store* necessitar de três fases de execução para escrever um valor na memória e um *load* necessitar de três fases de execução para ler da memória, um *load* seguindo um *store* acessa o valor colocado em memória pelo *store*, no ciclo que segue o da conclusão do *store*. Isto explica o fato do *store* ser considerado como tendo nenhum *delay slot*.

6.4.7 Instruções *branch*

Apesar do *branch* tomar apenas uma fase de execução, há cinco *delay slots* entre a execução do branch e a execução do código alvo. A Figura 6.22 mostra as fases do *pipeline* usadas pela instrução *branch* e o código alvo do *branch*. Os *delay slots* estão sombreados.

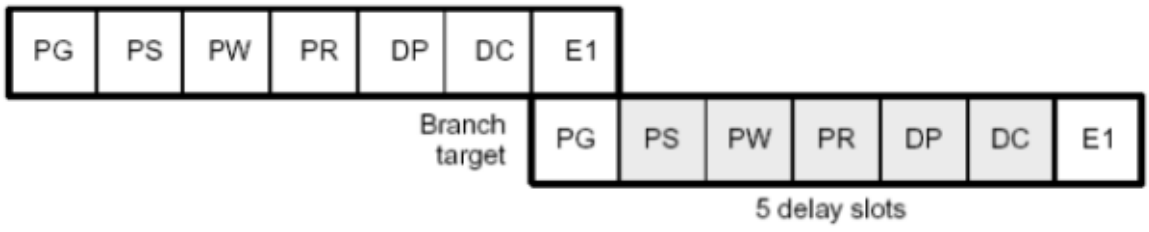

Figura 6.22: Todas as fases do *pipeline* usadas pelas instruções de *branch*.

A Figura 6.23 mostra um diagrama de bloco de execução do *branch*. Se um *branch* está na fase E1 do *pipeline* (na unidade .S2 da figura), o alvo do *branch* está no pacote de busca que está em PG durante o mesmo ciclo (sombreado na figura). Pelo fato do alvo do *branch* ter que esperar até alcançar a fase E1, o *branch* possui cinco *delay slots*.

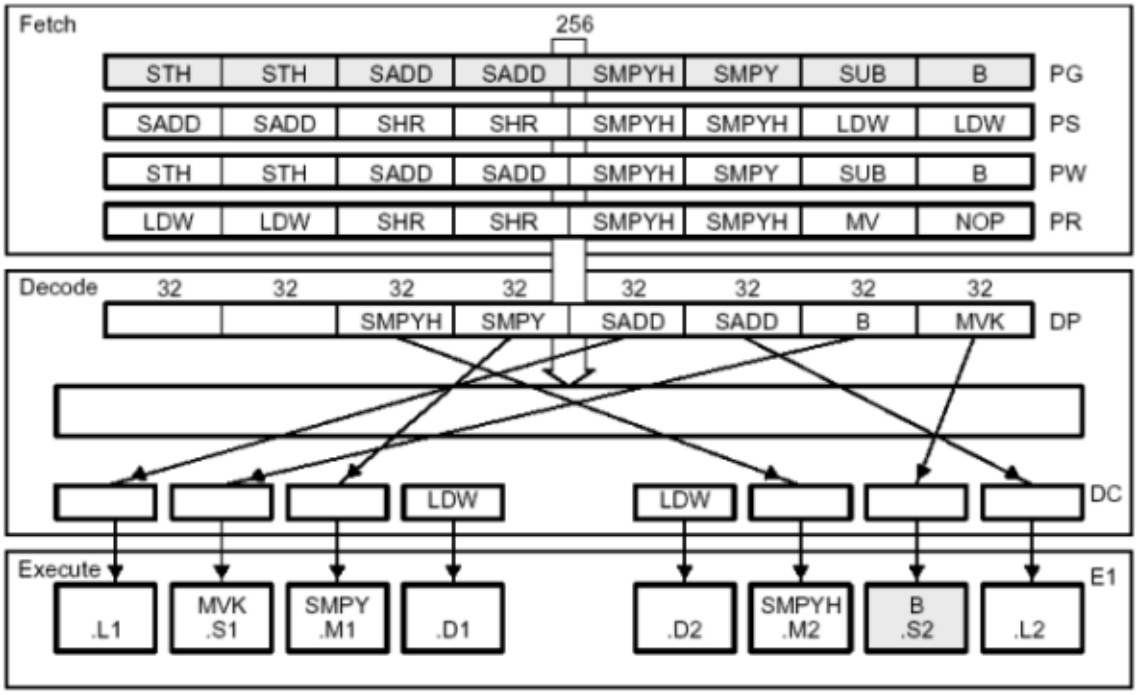

Figura 6.23: Diagrama de execução das instruções de *branch*.

6.4.8 Instruções de duplo ciclo com precisão dupla

As instruções de duplo ciclo com dupla precisão utilizam as fases E1 e E2 do *pipeline* para completar suas operações. Abaixo são listadas intruções que se enquandram neste tipo:

- ABSDP
- RCPDP
- RSQDP
- SPDP

Na fase E1 são lidos 32 *bits* de cada fonte, os de mais baixa ordem de *src1* e os de mais alta ordem em *src2*. Na fase E2 são lidos os *bits* restantes. As instruções deste tipo são executadas na unidade .S. Na Tabela 6.9 é possível ver, em detalhes, o que acontece em cada fase.

	E1	**E2**
Lê	*src1_l, src2_h*	*src2_l, src1_h*
Escreve	*dst_l*	*dst_h*
Unidade	.S	

Tabela 6.9: Execução de instruções de duplo ciclo com precisão dupla.

6.4.9 Instruções 4-ciclos

As instruções 4-ciclos usam as fases E1 – E4 do *pipeline* para completar suas operações. Abaixo são listadas intruções que se enquandram neste tipo:

- ADDSP
- DPINT
- DPSP
- DPTRUNC
- INTSP
- MPYSP
- SPINT
- SPTRUNC
- SUBSP

Os fontes são lidos na fase E1 e os resultados são escritos na fase E4. As instruções 4-ciclos são executadas nas unidades .M e .L. Na Tabela 6.10 é possível ver, em detalhes, o que acontece em cada fase.

	E1	**E2**	**E3**	**E4**
Lê	*src1, src2*			
Escreve				*dst*
Unidade	.L ou .M			

Tabela 6.10: Execução de instruções 4-ciclos.

6.4.10 Instrução INTDP

A instrução INTDP usa as fases E1 – E5 do *pipeline* para completar suas operações. Na fase E1, *src2* é lido, os 32 *bits* de mais baixa ordem do resultado é escrito na fase E4 e o restante em E5. Esta instrução é executada na unidade .L. Na Tabela 6.11 é possível ver, em detalhes, o que acontece em cada fase.

	E1	**E2**	**E3**	**E4**	**E5**
Lê	*src2*				
Escreve				*dst_l*	*dst_h*
Unidade	.L				

Tabela 6.11: Execução da instrução INTDP.

6.4.11 Instruções de comparação com precisão dupla

As instruções de comparação com dupla precisão usam as fases E1 e E2 do *pipeline* para completar suas operações. Os 32 *bits* de mais baixa ordem dos fontes são lidos em E1, o restante em E2 e o resultado é escrito em E2. Segue a lista das instruções:

- CMPEQDP
- CMPLTDP
- CMPGTDP

As instruções de comparação com precisão dupla são executadas na unidade .S. A latência da unidade funcional para estas instruções é 2. Na Tabela 6.12 é possível ver em detalhes o que acontece em cada fase.

	E1	E2
Lê	*src1_l, src2_l*	*src1_h, src2_h*
Escreve		*dst*
Unidade	.S	

Tabela 6.12: Execução das instruções de comparação em dupla precisão.

6.4.12 Instruções ADDDP e SUBDP

As instruções ADDDP E SUBDP usam as fases E1 – E7 do *pipeline* para completar suas operações. Nas fases E1 e E2 os fontes são lidos e nas fases E6 e E7 os resultados são escritos. Estas instruções são executadas na unidade .L e latência para estas instruções é 2. Na Tabela 6.13 é possível ver, em detalhes, o que acontece em cada fase.

	E1	E2	E3	E4	E5	E6	E7
Lê	*src1_l, src2_l*	*src1_h, src2_h*					
Escreve						*dst_l*	*dst_h*
Unidade	.L						

Tabela 6.13: Execução das instruções ADDDP e SUBDP.

6.4.13 Instrução MPYI

A instrução MPYI usa as fases E1 – E9 do pipeline para completar suas operações. Os fontes são lidos nas fases E1 – E4 e o resultado é escrito na fase E9. Esta instrução é executada na unidade .M e sua latência é de 4. Na Tabela 6.14 é possível ver em detalhes o que acontece em cada fase.

	E1	E2	E3	E4	E5	E6	E7	E8	E9
Lê	*src1, src2*	*src1, src2*	*src1, src2*	*src1, src2*					
Escreve									*dst*
Unidade	.M								

Tabela 6.14: Execução da instrução MPYI.

6.4.14 Instrução MPYID

A instrução MPYID usa as fases E1 – E10 do pipeline para completar suas operações. Os fontes são lidos nas fases E1 – E4 e o resultado é escrito nas fases E9 e E10. Esta instrução é executada na unidade .M e sua latência é de 4. Na Tabela 6.15 é possível ver em detalhes o que acontece em cada fase.

	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
Lê	*src1, src2*	*src1, src2*	*src1, src2*	*src1, src2*						
Escreve									*dst_l*	*dst_h*
Unidade	.M									

Tabela 6.15: Execução da instrução MPYID.

6.4.15 Instrução MPYDP

A instrução MPYDP usa as fases E1 – E10 do pipeline para completar suas operações. Os fontes são lidos nas fases E1 – E4 e o resultado é escrito nas fases E9 e E10. Esta instrução é executada na unidade .M e sua latência é de 4. Na Tabela 6.16 é possível ver em detalhes o que acontece em cada fase.

	E1	E2	E3	E4	E5–E8	E9	E10
Lê	*src1_l, src2_l*	*src1_l, src2_l*	*src1_h, src2_h*	*src1_h, src2_h*			
Escreve						*dst_l*	*dst_h*
Unidade	.M						

Tabela 6.16: Execução da instrução MPYDP.

6.4.16 Considerações de desempenho

Um pacote de busca (FP) é um agrupamento de oito instruções. Cada FP pode ser dividido em até oito pacotes de execução (EPs). Cada EP contém instruções que executam em paralelo. Cada instrução executa em uma unidade funcional independente. O efeito no *pipeline* de combinações de EPs que incluem número variável de instruções paralelas, ou apenas única instrução que executa serialmente com outro código, é considerado aqui.

Em geral, o número de pacotes de execução em um único FP define o fluxo das instruções através do *pipeline*. Outro fator determinante são os tipos das instruções no EP. Cada tipo de instrução tem um número fixo de ciclos de execução que determina quando as operações desta instrução são concluídas.

Finalmente, o efeito do sistema de memória em operação no *pipeline* é considerado. O acesso a memória de programa e a de dados é discutido, com os congelamentos de memória.

Funcionamento do *pipeline* com múltiplos pacotes de execução em um pacote de busca

Novamente referenciando a Figura 6.16 na página 40, o funcionamento do *pipeline* é mostrado com oito instruções em todo pacote de busca e estas formam um único pacote de execução. Na Figura 6.24, entretanto, é exibido o funcionamento do *pipeline* com um pacote de busca que contém múltiplos pacotes de execução. O código para a situação ilustrada na Figura 6.24 deve ter o seguinte aspecto:

```
   instruction A;  EP k       FP n
|| instruction B;

   instruction C;  EP k + 1   FP n
|| instruction D;
|| instruction E;

    instruction F;  EP k + 2   FP n
|| instruction G;
|| instruction H;

   instruction I;  EP k + 3   FP n + 1
|| instruction J;
|| instruction K;
|| instruction L;
|| instruction M;
|| instruction N;
|| instruction O;
|| instruction P;

...
```

Na Figura 6.24 o pacote de busca n contém três pacotes de execução e é seguido por seis pacotes de busca (n + 1 até n + 6), cada um destes contendo apenas um pacote de execução. O primeiro pacote de busca (n) segue através das fases do estágio de busca durante os ciclos de 1 a 4. Durante estes ciclos, uma fase de busca é iniciada, em cada ciclo, para cada pacote de busca que segue.

No quinto ciclo (a fase de despacho) a CPU avalia e detecta que há mais que um pacote de execução no pacote de busca n. Isto faz com que o *pipeline* congele, o qual permite que a fase DP inicie para os

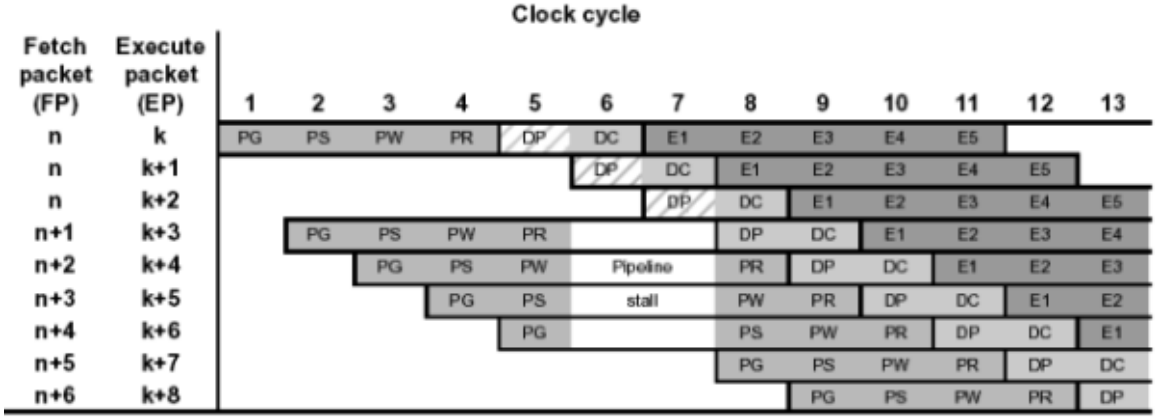

Figura 6.24: Funcionamento do *pipeline*: pacotes de busca com diferentes números de pacotes de execução.

pacotes de execução k + 1 e K + 2 nos ciclos 6 e 7. Uma vez que o pacote de execução k + 2 está pronto para mover-se para a fase DC (ciclo 8), o congelamento do *pipeline* é encerrado e o fluxo do *pipeline* é reestabelecido até que um novo congelamento torne-se necessário.

NOPs multiciclo

A instrução **NOP** tem um operando opcional, *count*, que permite emitir uma instrução única para **NOPs** multiciclo. Um **NOP** 2, por exemplo, preenche *delay slots* extras para as instruções no pacote de execução e para todos pacotes de execução anteriores.

A Figura 6.25 mostra um NOP em um pacote de execução (em paralelo) com outro código. O resultado do **LD**, **ADD** e **MPY** estará disponível durante o ciclo apropriado para cada instrução. Neste exemplo o **NOP** não tem efeito no pacote de execução.

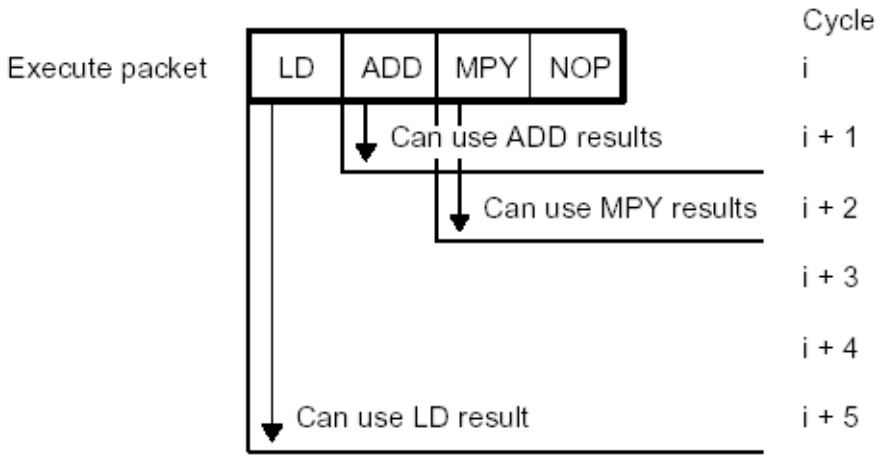

Figura 6.25: NOP em um pacote de execução.

A Figura 6.26 mostra a substituição do **NOP** de ciclo único por um **NOP** multiciclo (**NOP** 5) no mesmo pacote de execução. o **NOP** 5 fará com que as instruções que estão executando em paralelo (no mesmo pacote de execução) só disponibilizem os seus resultados juntamente com o encerramento do **NOP** multiciclo. Sendo assim, os resultados do **LD**, **ADD** e **MPY** não podem ser usados por alguma outra instrução até que o período do **NOP** 5 tenha concluído.

A Figura 6.27 mostra como um **NOP** multiciclo pode ser afetado por um *branch*. Se os *delay slots* de um *branch* finaliza enquanto um *NOP* multiciclo ainda está sendo despachado, o *branch* sobrepõe o **NOP** multiciclo e o alvo do *branch* começa a execução cinco *delay slots* depois do *branch* ter sido emitido.

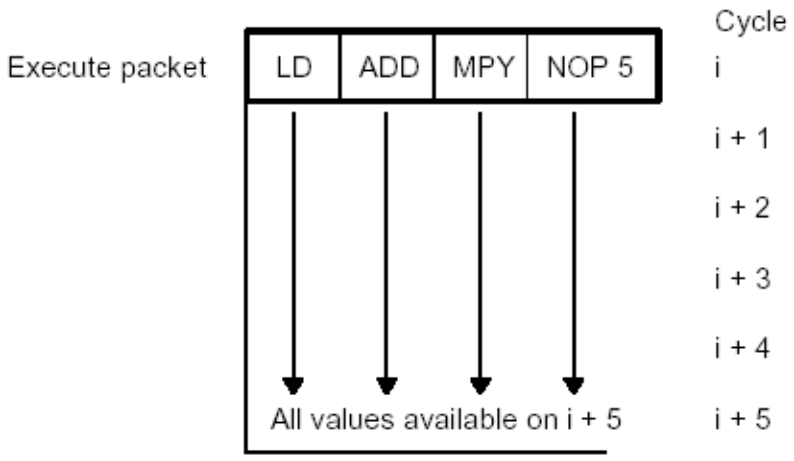

Figura 6.26: NOP multiciclo em um pacote de execução.

Em um caso, o pacote de execução 1 (EP1) não tem um *branch.* O **NOP** 5 em EP6 irá forçar a CPU esperar até o ciclo 11 para executar EP7.

Em outro caso, EP1 possui um *branch.* Os *delay slots* do *branch* coincidem com os ciclos 2 até o 6. Uma vez que o código alvo atinja E1 no ciclo 7, ele executa.

Considerações acerca da memória

Os dispositivos da família C62x e C67x possuem uma configuração de memória típica de um DSP, com memória de programa em um espaço físico e memória de dados em outro espaço físico. Carregamento de dados e buscas de instruções possuem as mesmas operações no *pipeline*, eles apenas usam diferentes fases para completar suas operações. Em ambos o acesso a memória é dividido em múltiplas fases, como pode ser visto na Figura 6.28.

Para entender os acessos a memória, compare os carremento de dados e busca/despacho de instrução. A comparação é válida porque o carregamento de dados e busca de instrução operam em mémorias internas da mesma velocidade nos dispositivos da família C62x e C67x e executam os mesmos tipos de operações (listados na Tabela 6.17). A Tabela 6.17 exibe como é realizada a busca de instrução em comparação com o carregamento de dados.

Operação	**Fase no acesso a memória de programa**	**Fase no acesso a memória de dados**
Calcula endereço	PG	E1
Envia endereço para a memória	PS	E2
Leitura da memória	PW	E3
Memória de programa: CPU recebe pacote de busca. Memória de dados: CPU recebe os dados.	PR	E4
Memória de programa: envia as instruções para as unidades funcionais. Memória de dados: envia dados para o registrador.	DP	E5

Tabela 6.17: Acessos a memória de programa X acessos a memória de dados

Dependendo do tipo de memória e do tempo requerido para completar um acesso, o *pipeline* deve congelar para assegurar apropriada sincronia de dados e instruções. Isto é discutido na Seção 6.4.16, Congelamento de Memória.

No caso onde acessos múltiplos são feitos a uma memória de única entrada, o *pipeline* irá congelar para permitir que os acessos extra ocorram.

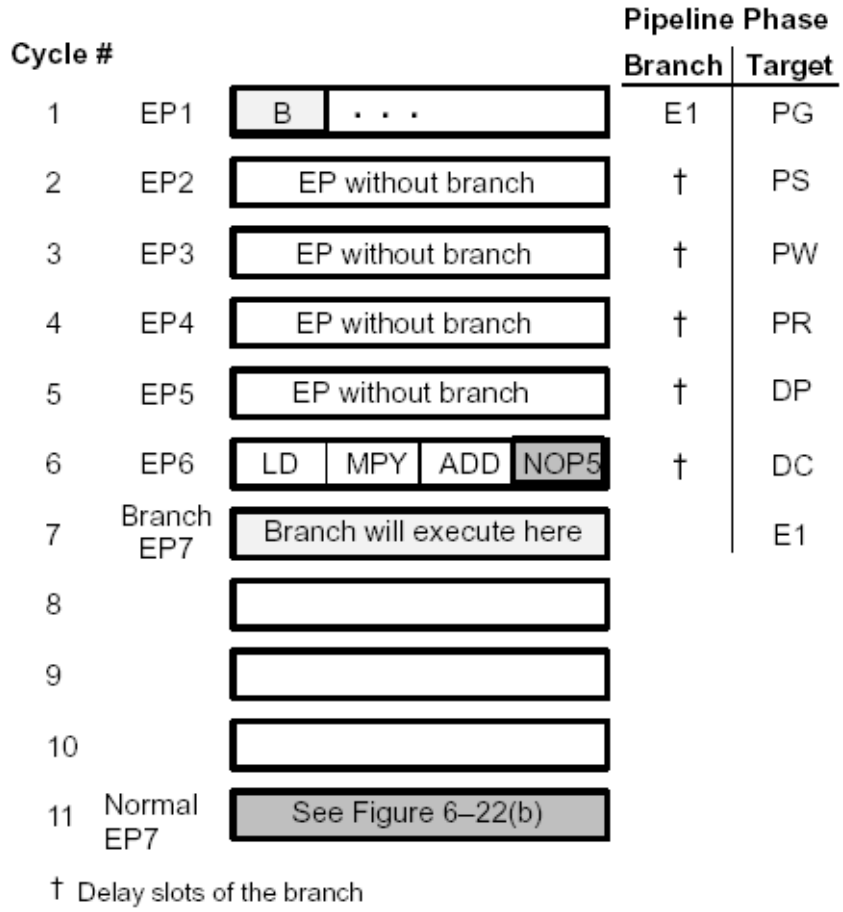

Figura 6.27: *Branchs* e **NOPs** multiciclo.

Congelamento de memória

Um congelamento de memória ocorre quando a memória não está pronta para responder a um acesso da CPU. Este acesso ocorre durante a fase PW para um acesso a memória de programa e durante a fase E3 para um acesso a memória de dados. O congelamento de memória faz com que todas as fases do *pipeline* alonguem além de um único ciclo de *clock*, levando a execução a tomar ciclos de *clock* adicionais para finalizar. Os resultados da execução do programa são identicos tanto com um congelamento quanto não. A Figura 6.29 ilustra esta situação.

***Hits* no banco de memória**

A maioria dos dispositivos C62x e C67x usam um esquema de banco de memória intercalado, como ilustrado na Figura 6.30. Os dispositivos C6211 e C6711 utilizam um esquema de memória cache em dois níveis. Nos dispositivos da família C67x há 8 bancos de memória. Assim, o exemplo ilustrado não pertence a estes dispositivos. Cada número no diagrama representa um endereço de *byte*. Uma instrução de carregamento de *byte* (**LDB**) de endereço 0 carrega o *byte* 0 do banco 0. Um *load halfword* (**LDH**) de endereço 0 carrega o valor de comprimento de meia palavra nos *bytes* 0 e 1, os quais estão também no banco 0. Um **LDW** do endereço 0 carrega do *byte* 0 ao 3 dos bancos 0 e 1.

Pelo fato de cada um destes bancos ser de memória de única entrada, apenas um acesso é permitido a cada banco por ciclo. Dois acessos a um único banco em um dado ciclo resulta em um congelamento de memória que pára todo o funcionamento do *pipeline* por um ciclo, enquanto o segundo valor é lido da memória. Duas operações em memória por ciclo são permitidas sem congelamento desde que não acessem o mesmo banco.

Considere o código abaixo. Pelo fato de ambos os *loads* estarem tentando acessar o mesmo banco no

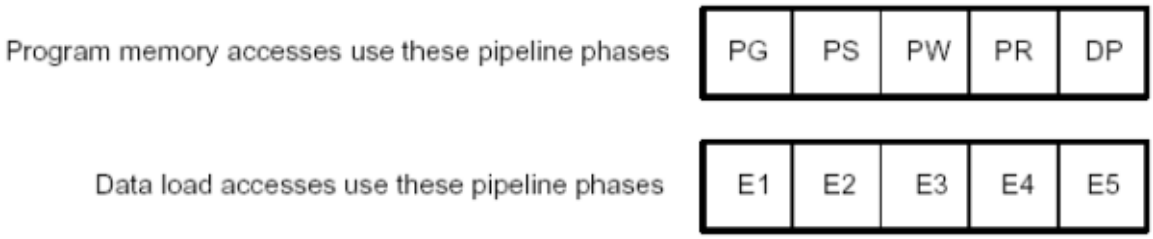

Figura 6.28: Fases do *pipeline* usadas durante o acesso a memória.

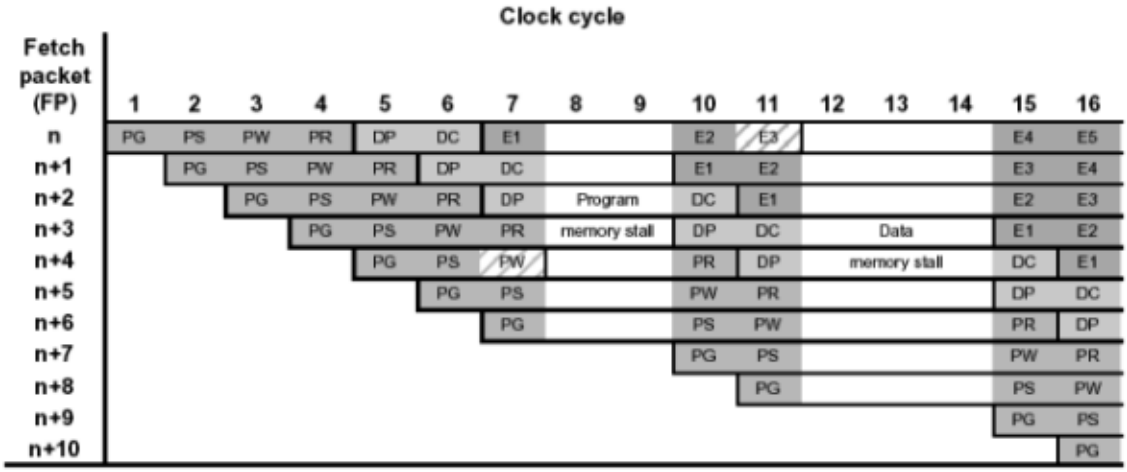

Figura 6.29: Congelamento das memórias de programa e dados.

mesmo tempo, um *load* deve esperar. O primeiro **LDW** acessa o banco 0 no ciclo $i + 2$ (na fase E3) e o segundo **LDW** acessa o banco 0 no ciclo $i + 3$ (no fase E3). Veja a Tabela 6.18 para identificação de ciclos e fases. A fase E4 para ambas as instruções **LDW** está no ciclo $i + 4$. Para eliminar esta fase extra, os *loads* devem acessar dados de diferentes bancos.

```
   LDW .D1 *A4++, A5;   load 1, endereço A4 está no banco 0.
|| LDW .D2 *B4++, B5;   load 2, endereço B4 está no banco 0.
```

	i	**i + 1**	**i + 2**	**i + 3**	**i + 4**	**i + 5**
LDW .D1	E1	E2	E3	–	E4	E5
LDW .D2	E1	E2	–	E3	E4	E5

Tabela 6.18: Loads no pipeline

Para dispositivos que possuem mais que um espaço de memória (veja a Figura 6.31), um acesso ao banco 0 em um espaço não interfere um acesso ao banco 0 em outro espaço de memória, e não ocorre congelamento do *pipeline*.

As memórias internas da família C62x e C67x varia de dispositivo para dispositivo. Veja o TMS320C6000 Periferals Reference Guide para determinar os espaços de memória em seu dispositivo em particular.

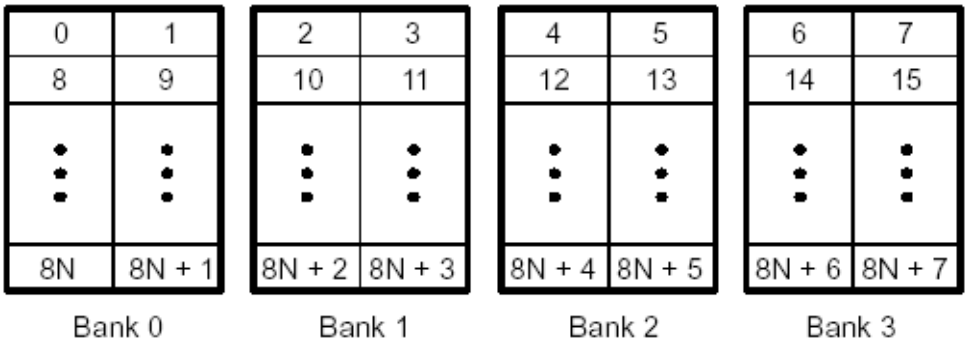

Figura 6.30: Banco de memória intercalado.

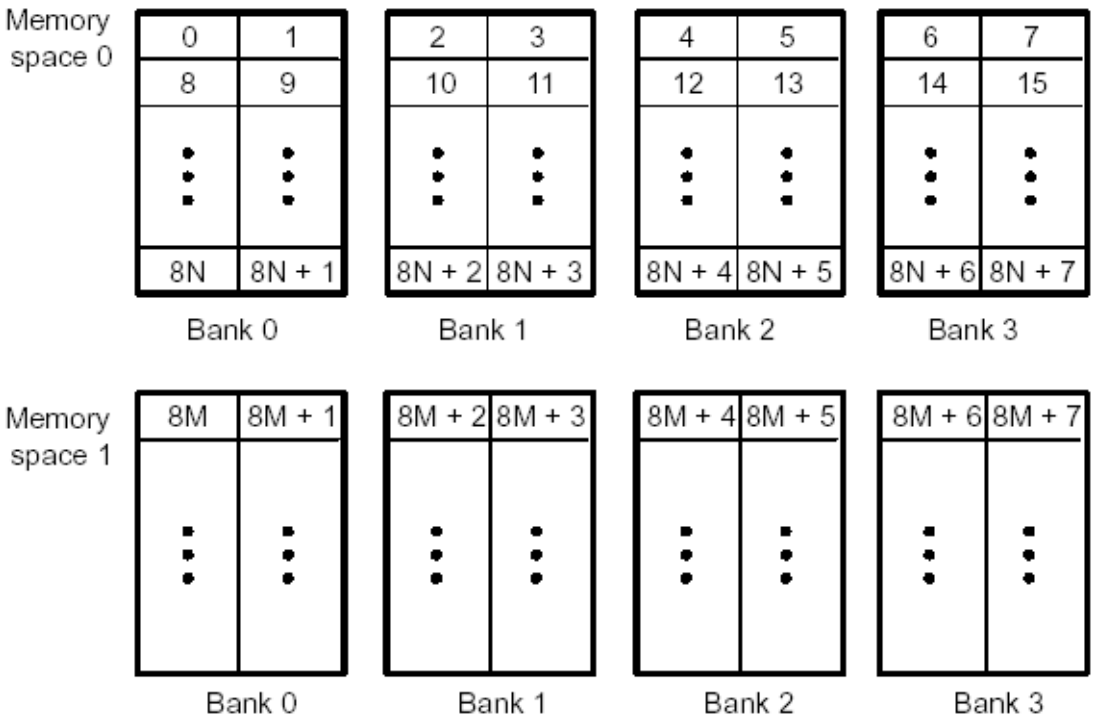

Figura 6.31: Banco de memória intercalada com dois espaços de memória.

6.4.17 Restrições nas unidades funcionais

Caso seja feita uma tentativa de otimização das instruções no pipeline, considere as instruções que são executadas em cada unidade. Fontes e destinos são lidos e escritos diferentemente para cada instrução. É possível realizar grandes melhorias na otimização se for considerado o que acontece durante as fases de execução das instruções que utilizam a mesma unidade funcional em cada pacote de execução.

Nas seções que seguem é possível encontrar maiores informações sobre estas restrições e as possibilidades de otimização.

Restrições na unidade .S

A Figura 6.32 mostra as restrições de instrução para instruções de ciclo único executando na unidade .S.

	Instruction Execution	
Cycle	1	2
Single-cycle	RW	

Instruction Type	Subsequent Same-Unit Instruction Executable
Single-cycle	✓
DP compare	✓
2-cycle DP	✓
Branch	✓

Instruction Type	Same Side, Different Unit, Both Using Cross Path Executable
Single-cycle	✓
Load	✓
Store	✓
INTDP	✓
ADDDP/SUBDP	✓
16 × 16 multiply	✓
4-cycle	✓
MPYI	✓
MPYID	✓
MPYDP	✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- W — Destinations written for the instruction
- ✓ — Next instruction can enter E1 during cycle

Figura 6.32: Restrições de instrução para instrução de ciclo único na unidade .S.

A Figura 6.33 mostra as restrições de instrução para instruções de comparação com precisão dupla executando na unidade .S.

	Instruction Execution		
Cycle	1	2	3
DP compare	R	RW	

Instruction Type	Subsequent Same-Unit Instruction Executable	
Single-cycle	Xrw	✓
DP compare	Xr	✓
2-cycle DP	Xrw	✓
Branch†	Xr	✓

Instruction Type	Same Side, Different Unit, Both Using Cross Path Executable	
Single-cycle	Xr	✓
Load	Xr	✓
Store	Xr	✓
INTDP	Xr	✓
ADDDP/SUBDP	Xr	✓
16 × 16 multiply	Xr	✓
4-cycle	Xr	✓
MPYI	Xr	✓
MPYID	Xr	✓
MPYDP	Xr	✓

Legend: ■ E1 phase of the single-cyle instruction
R Sources read for the instruction
W Destinations written for the instruction
✓ Next instruction can enter E1 during cycle
Xr Next instruction cannot enter E1 during cycle–read/decode constraint
Xrw Next instruction cannot enter E1 during cycle–read/decode/write constraint
† The branch on register instruction is the only branch instruction that reads a general-purpose register

Figura 6.33: Restrições de instrução para instrução de comparação com precisão dupla na unidade .S.

A Figura 6.34 mostra as restrições de instrução para instruções de duplo ciclo com precisão dupla executando na unidade .S.

	Instruction Execution		
Cycle	1	2	3
2-cycle	RW	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**		
Single-cycle		Xw	✓
DP compare		✓	✓
2-cycle DP		Xw	✓
Branch		✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**		
Single cycle		✓	✓
Load		✓	✓
Store		✓	✓
INTDP		✓	✓
ADDDP/SUBDP		✓	✓
16 × 16 multiply		✓	✓
4-cycle		✓	✓
MPYI		✓	✓
MPYID		✓	✓
MPYDP		✓	✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- W — Destinations written for the instruction
- ✓ — Next instruction can enter E1 during cycle
- Xw — Next instruction cannot enter E1 during cycle–write constraint

Figura 6.34: Restrições de instrução para instrução de duplo ciclo com precisão dupla na unidade .S.

A Figura 6.35 mostra as restrições de instrução para instruções de *branch* executando na unidade .S.

	Instruction Execution							
Cycle	1	2	3	4	5	6	7	8
Branch†	R							
Instruction Type	**Subsequent Same-Unit Instruction Executable**							
Single-cycle		✓	✓	✓	✓	✓	✓	✓
DP compare		✓	✓	✓	✓	✓	✓	✓
2-cycle DP		✓	✓	✓	✓	✓	✓	✓
Branch		✓	✓	✓	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**							
Single-cycle		✓	✓	✓	✓	✓	✓	✓
Load		✓	✓	✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓	✓	✓
INTDP		✓	✓	✓	✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓	✓	✓	✓
16 × 16 multiply		✓	✓	✓	✓	✓	✓	✓
4-cycle		✓	✓	✓	✓	✓	✓	✓
MPYI		✓	✓	✓	✓	✓	✓	✓
MPYID		✓	✓	✓	✓	✓	✓	✓
MPYDP		✓	✓	✓	✓	✓	✓	✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- ✓ — Next instruction can enter E1 during cycle
- † — The branch on register instruction is the only branch instruction that reads a general-purpose register

Figura 6.35: Restrições de instrução para instrução de *branch* na unidade .S.

Restrições na unidade .M

A Figura 6.36 mostra as restrições de instrução para instruções de multiplicação 16 x 16 executando na unidade .M.

	Instruction Execution		
Cycle	1	2	3
16 X 16 multiply	R	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**		
16 X 16 multiply		✓	✓
4-cycle		✓	✓
MPYI		✓	✓
MPYID		✓	✓
MPYDP		✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**		
Single-cycle		✓	✓
Load		✓	✓
Store		✓	✓
DP compare		✓	✓
2-cycle DP		✓	✓
Branch		✓	✓
4-cycle		✓	✓
INTDP		✓	✓
ADDDP/SUBDP		✓	✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- W — Destinations written for the instruction
- ✓ — Next instruction can enter E1 during cycle

Figura 6.36: Restrições de instrução para instrução de multiplicação 16 x 16 na unidade .M.

A Figura 6.37 mostra as restrições de instrução para instruções 4-ciclos executando na unidade .M.

	Instruction Execution				
Cycle	1	2	3	4	5
4-cycle	R			W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**				
16 × 16 multiply		✓	Xw	✓	✓
4-cycle		✓	✓	✓	✓
MPYI		✓	✓	✓	✓
MPYID		✓	✓	✓	✓
MPYDP		✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**				
Single-cycle		✓	✓	✓	✓
Load		✓	✓	✓	✓
Store		✓	✓	✓	✓
DP compare		✓	✓	✓	✓
2-cycle DP		✓	✓	✓	✓
Branch		✓	✓	✓	✓
4-cycle		✓	✓	✓	✓
INTDP		✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- W — Destinations written for the instruction
- ✓ — Next instruction can enter E1 during cycle
- Xw — Next instruction cannot enter E1 during cycle–write constraint

Figura 6.37: Restrições de instrução para instrução 4-ciclos na unidade .M.

A Figura 6.38 mostra as restrições de instrução para a instrução MPYI executando na unidade .M.

	Instruction Execution									
Cycle	1	2	3	4	5	6	7	8	9	10
MPYI	R	R	R	R					W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**									
16 × 16 multiply		Xr	Xr	Xr	✓	✓	✓	Xw	✓	✓
4-cycle		Xr	Xr	Xr	Xu	Xw	Xu	✓	✓	✓
MPYI		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
MPYID		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
MPYDP		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**									
Single-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
Load		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
Store		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
DP compare		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
2-cycle DP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
Branch		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
4-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
INTDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
ADDDP/SUBDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- W — Destinations written for the instruction
- ✓ — Next instruction can enter E1 during cycle
- Xr — Next instruction cannot enter E1 during cycle–read/decode constraint
- Xw — Next instruction cannot enter E1 during cycle–write constraint
- Xu — Next instruction cannot enter E1 during cycle–other resource conflict

Figura 6.38: Restrições de instrução para instrução MPYI na unidade .M.

A Figura 6.39 mostra as restrições de instrução para a instrução MPYID executando na unidade .M.

	Instruction Execution										
Cycle	1	2	3	4	5	6	7	8	9	10	11
MPYID	R	R	R	R					W	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**										
16 × 16 multiply		Xr	Xr	Xr	✓	✓	✓	Xw	Xw	✓	✓
4-cycle		Xr	Xr	Xr	Xu	Xw	Xw	✓	✓	✓	✓
MPYI		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
MPYID		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
MPYDP		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**										
Single-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Load		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Store		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
DP compare		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
2-cycle DP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Branch		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
4-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
INTDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
ADDDP/SUBDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- W — Destinations written for the instruction
- ✓ — Next instruction can enter E1 during cycle
- Xr — Next instruction cannot enter E1 during cycle–read/decode constraint
- Xw — Next instruction cannot enter E1 during cycle–write constraint
- Xu — Next instruction cannot enter E1 during cycle–other resource conflict

Figura 6.39: Restrições de instrução para instrução MPYID na unidade .M.

A Figura 6.40 mostra as restrições de instrução para a instrução MPYDP executando na unidade .M.

	Instruction Execution										
Cycle	1	2	3	4	5	6	7	8	9	10	11
MPYDP	R	R	R	R					W	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**										
16 × 16 multiply		Xr	Xr	Xr	✓	✓	✓	Xw	Xw	✓	✓
4-cycle		Xr	Xr	Xr	Xu	Xw	Xw	✓	✓	✓	✓
MPYI		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓	✓
MPYID		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓	✓
MPYDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**										
Single-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Load		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Store		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
DP compare		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
2-cycle DP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Branch		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
4-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
INTDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
ADDDP/SUBDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- W — Destinations written for the instruction
- ✓ — Next instruction can enter E1 during cycle
- Xr — Next instruction cannot enter E1 during cycle–read/decode constraint
- Xw — Next instruction cannot enter E1 during cycle–write constraint
- Xu — Next instruction cannot enter E1 during cycle–other resource conflict

Figura 6.40: Restrições de instrução para instrução MPYDP na unidade .M.

Restrições na unidade .L

A Figura 6.41 mostra as restrições de instrução para instruções de ciclo único executando na unidade .L.

	Instruction Execution	
Cycle	1	2
Single-cycle	RW	

Instruction Type	Subsequent Same-Unit Instruction Executable
Single-cycle	✓
4-cycle	✓
INTDP	✓
ADDDP/SUBDP	✓

Instruction Type	Same Side, Different Unit, Both Using Cross Path Executable
Single-cycle	✓
DP compare	✓
2-cycle DP	✓
4-cycle	✓
Load	✓
Store	✓
Branch	✓
16 × 16 multiply	✓
MPYI	✓
MPYID	✓
MPYDP	✓

Legend: E1 phase of the single-cyle instruction
R Sources read for the instruction
W Destinations written for the instruction
✓ Next instruction can enter E1 during cycle

Figura 6.41: Restrições de instrução para instruções de ciclo único na unidade .L.

A Figura 6.42 mostra as restrições de instrução para instruções 4-ciclos executando na unidade .L.

	Instruction Execution				
Cycle	1	2	3	4	5
4-cycle	R			W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**				
Single-cycle		✓	✓	Xw	✓
4-cycle		✓	✓	✓	✓
INTDP		✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**				
Single-cycle		✓	✓	✓	✓
DP compare		✓	✓	✓	✓
2-cycle DP		✓	✓	✓	✓
4-cycle		✓	✓	✓	✓
Load		✓	✓	✓	✓
Store		✓	✓	✓	✓
Branch		✓	✓	✓	✓
16 × 16 multiply		✓	✓	✓	✓
MPYI		✓	✓	✓	✓
MPYID		✓	✓	✓	✓
MPYDP		✓	✓	✓	✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- W — Destinations written for the instruction
- ✓ — Next instruction can enter E1 during cycle
- Xw — Next instruction cannot enter E1 during cycle–write constraint

Figura 6.42: Restrições de instrução para instruções 4-ciclos na unidade .L.

A Figura 6.43 mostra as restrições de instrução para a instrução INTDP executando na unidade .L.

	Instruction Execution					
Cycle	1	2	3	4	5	6
INTDP	R			W	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**					
Single-cycle		✓	✓	Xw	Xw	✓
4-cycle		Xw	✓	✓	✓	✓
INTDP		Xw	✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**					
Single-cycle		✓	✓	✓	✓	✓
DP compare		✓	✓	✓	✓	✓
2-cycle DP		✓	✓	✓	✓	✓
4-cycle		✓	✓	✓	✓	✓
Load		✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓
Branch		✓	✓	✓	✓	✓
16 × 16 multiply		✓	✓	✓	✓	✓
MPYI		✓	✓	✓	✓	✓
MPYID		✓	✓	✓	✓	✓
MPYDP		✓	✓	✓	✓	✓

Legend: E1 phase of the single-cyle instruction
R Sources read for the instruction
W Destinations written for the instruction
✓ Next instruction can enter E1 during cycle
Xw Next instruction cannot enter E1 during cycle–write constraint

Figura 6.43: Restrições de instrução para a instrução INTDP na unidade .L.

A Figura 6.44 mostra as restrições de instrução para instruções ADDDP/SUBDP executando na unidade .L.

	Instruction Execution							
Cycle	1	2	3	4	5	6	7	8
ADDDP/SUBDP	R	R				W	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**							
Single-cycle		Xr	✓	✓	✓	Xw	Xw	✓
4-cycle		Xr	Xw	Xw	✓	✓	✓	✓
INTDP		Xrw	Xw	Xw	✓	✓	✓	✓
ADDDP/SUBDP		Xr	✓	✓	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**							
Single-cycle		Xr	✓	✓	✓	✓	✓	✓
DP compare		Xr	✓	✓	✓	✓	✓	✓
2-cycle DP		Xr	✓	✓	✓	✓	✓	✓
4-cycle		Xr	✓	✓	✓	✓	✓	✓
Load		Xr	✓	✓	✓	✓	✓	✓
Store		Xr	✓	✓	✓	✓	✓	✓
Branch		Xr	✓	✓	✓	✓	✓	✓
16 × 16 multiply		Xr	✓	✓	✓	✓	✓	✓
MPYI		Xr	✓	✓	✓	✓	✓	✓
MPYID		Xr	✓	✓	✓	✓	✓	✓
MPYDP		Xr	✓	✓	✓	✓	✓	✓

Legend:
- ■ E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle
- Xr Next instruction cannot enter E1 during cycle–read/decode constraint
- Xw Next instruction cannot enter E1 during cycle–write constraint
- Xrw Next instruction cannot enter E1 during cycle–read/decode/write constraint

Figura 6.44: Restrições de instrução para as instruções ADDDP/SUBDP na unidade .L.

Restrições na unidade .D

A Figura 6.45 mostra as restrições de instrução para a instrução *load* executando na unidade .D.

	Instruction Execution					
Cycle	1	2	3	4	5	6
Load	RW				W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**					
Single-cycle		✓	✓	✓	✓	✓
Load		✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**					
16 × 16 multiply		✓	✓	✓	✓	✓
MPYI		✓	✓	✓	✓	✓
MPYID		✓	✓	✓	✓	✓
MPYDP		✓	✓	✓	✓	✓
Single-cycle		✓	✓	✓	✓	✓
DP compare		✓	✓	✓	✓	✓
2-cycle DP		✓	✓	✓	✓	✓
Branch		✓	✓	✓	✓	✓
4-cycle		✓	✓	✓	✓	✓
INTDP		✓	✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓	✓

Legend: E1 phase of the single-cyle instruction
R Sources read for the instruction
W Destinations written for the instruction
✓ Next instruction can enter E1 during cycle

Figura 6.45: Restrições de instrução para a instrução *load* na unidade .D.

A Figura 6.46 mostra as restrições de instrução para a instrução *store* executando na unidade .D.

	Instruction Execution			
Cycle	1	2	3	4
Store	RW			
Instruction Type	**Subsequent Same-Unit Instruction Executable**			
Single-cycle		✓	✓	✓
Load		✓	✓	✓
Store		✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**			
16 × 16 multiply		✓	✓	✓
MPYI		✓	✓	✓
MPYID		✓	✓	✓
MPYDP		✓	✓	✓
Single-cycle		✓	✓	✓
DP compare		✓	✓	✓
2-cycle DP		✓	✓	✓
Branch		✓	✓	✓
4-cycle		✓	✓	✓
INTDP		✓	✓	✓
ADDDP/SUBDP		✓	✓	✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- W — Destinations written for the instruction
- ✓ — Next instruction can enter E1 during cycle

Figura 6.46: Restrições de instrução para a instrução *store* na unidade .D.

A Figura 6.47 mostra as restrições de instrução para as instruções de ciclo único executando na unidade .D.

	Instruction Execution	
Cycle	1	2
Single-cycle	RW	
Instruction Type	**Subsequent Same-Unit Instruction Executable**	
Single-cycle		✓
Load		✓
Store		✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**	
16 × 16 multiply		✓
MPYI		✓
MPYID		✓
MPYDP		✓
Single-cycle		✓
DP compare		✓
2-cycle DP		✓
Branch		✓
4-cycle		✓
INTDP		✓
ADDDP/SUBDP		✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- W — Destinations written for the instruction
- ✓ — Next instruction can enter E1 during cycle

Figura 6.47: Restrições de instrução para as instruções de ciclo único na unidade .D.

A Figura 6.48 mostra as restrições de instrução para a instrução LDDW executando na unidade .D.

	Instruction Execution					
Cycle	1	2	3	4	5	6
LDDW	RW				W	

Instruction Type	Subsequent Same-Unit Instruction Executable				
Instruction with long result		✓	✓	✓	Xw	✓

Legend:
- E1 phase of the single-cyle instruction
- R — Sources read for the instruction
- W — Destinations written for the instruction
- ✓ — Next instruction can enter E1 during cycle
- Xw — Next instruction cannot enter E1 during cycle–write constraint

Figura 6.48: Restrições de instrução para a instrução LDDW na unidade .D.

6.5 Previsão de desvio

De acordo com [Talla et al., 2000], a série ’C62x não possui previsão de desvio. Com relação às séries ’C64x e ’C67x, não conseguimos nenhuma referência a respeito.

Capítulo 7

Memória

Este capítulo apresenta uma visão geral da organização de memória na família de DSPs TMS320.

7.1 Visão geral

A família TMS320C6000 apresenta uma arquitetura de memória composta por dois níveis de caches e uma memória externa. O cache nível 1 é dividido em cache de programa (L1P) e cache de dados (L1D). O nível 2 de cache pode ser configurado e dividido em memória L2 SRAM (memória interna) e L2 *cache*, que é usada para fazer cache da memória externa. Esta é acessada através de uma interface conhecida como EMIF (*External Memory InterFace* - Interface de memória externa). De modo geral essa é a hierarquia de memória dos processadores da família 6000.

Neste trabalho, serão mostrados os componentes do sistema de memória, suas características e como eles estão organizados. Além disso, será mostrado como é o funcionamento da hierarquia de memória, ou seja, como sãio tratados hits e misses. Finalmente, serão abordadas questões relacionadas à manutenção da coerência entre os vários níveis de memória.

O estudo será concentrado nos dispositovos 64x, 621x e 627x1. Estes dispositovos resumem as características da família TMS320C6000.

7.2 Componentes

A Figura 7.1é uma esquema de bloco mostrando os principais elementos que compõem a memória dos dispositivos da família 6000. L1 *Program cache controller* (Controlador do cache de programa L1): gerencia requisições de acesso a memória de programa feitos pela CPU. L1 *data cache controller* (Controlador do cache de dados L1): gerencia requisições de acesso a memória de programa feitos pela CPU. L2 *cache controller* (Controlador do cache L2): gerencia as operações de memória envolvendo o cache de nível 2. Além disso interage com o DMA/EDMA.

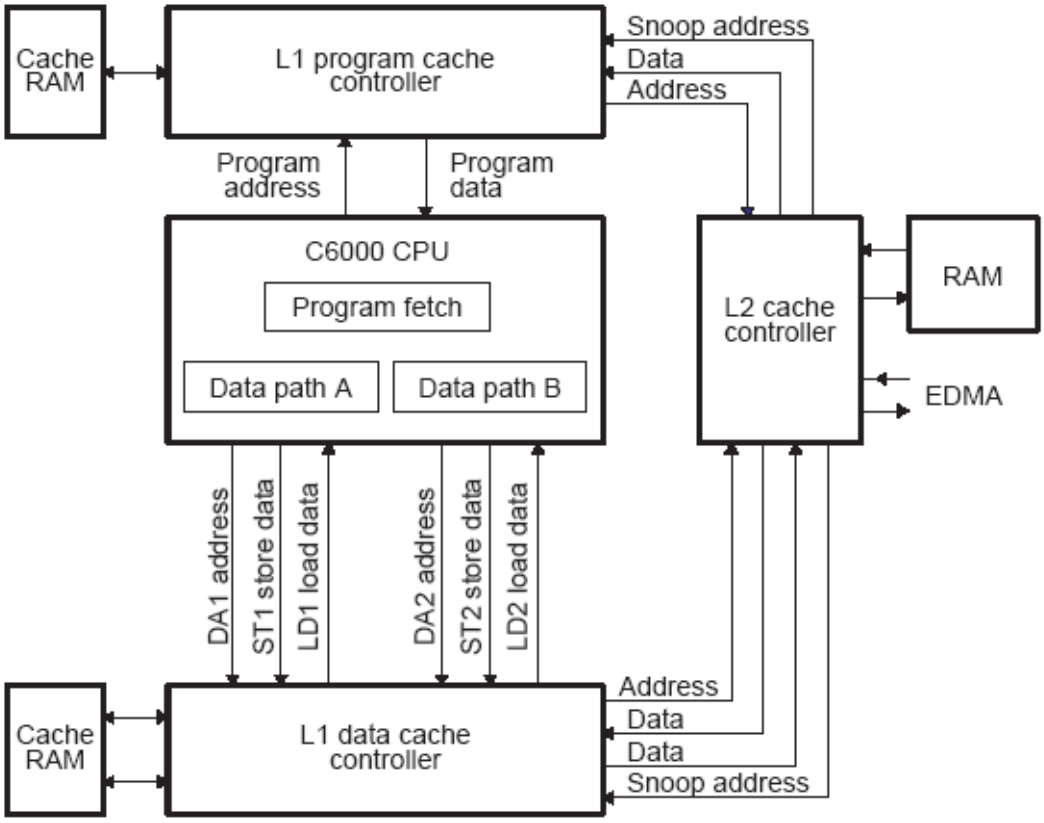

Figura 7.1: Diagrama de blocos - 621x/671x/64x

7.3 Arquitetura

A Figura 7.2 mostra a hierarquia de memória para os processadores 64x enquanto a Figura 7.3 mostra a hierarquia dos processadores 621x/627x. Os dispositvos 64x possuem 16Kbyte L1D e L1P enquanto os dispositivos 621x/627x possuem 4Kbytes. A tamanho da memória de nível 2 (L2 *cache* + L2 SRAM) é 64Kbytes para os 621x/627x e de 1024 Kbytes para os 64x.

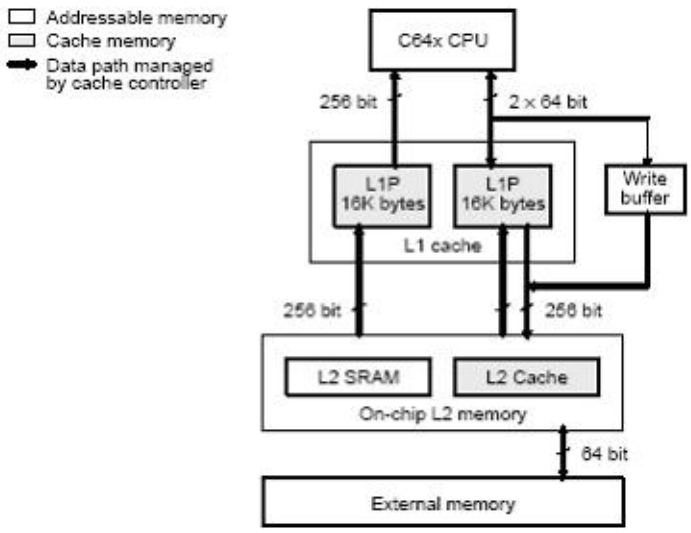

Figura 7.2: Arquitetura de memória - 64x

A seguir será feita uma breve descrição desses níveis de memória:

7.3.1 L1P

Sempre que a CPU acessa uma instrução na memória, ela é trazida para o L1P. O L1P é um *cache* somente de instruções. De modo geral, os L1P tem um comportamento similar para os dispositivos 621x/627x e 64x. Eles se diferem principalmente pelo tamanho, o tamanho da linha. A Figura 7.4 mostra as características desses dispositivos.

O L1P é composto pela memória cache e pela lógica de controle, mostrados na Figura 7.5

O endereço, de 32 bits, é dividido em três partes. Os 5 bits menos significativos (6 para os dispositivos 621x/671x) indicam o deslocamento (*offset*) dentro de uma linha. Os bits de 5-13 (6-11) indicam para qual linha o endereço deve ser mapeado. Os bits 14-31 (12-31) são usados como *tag* para diferenciar endereços que devem ser mapeados para a mesma linha do cache (o mapeamento é direto). O bit V (bit de validade) indica se o dado numa determinada linha é válido. A lógica de controle verifica o *tag* e o bit V, para determinar se o endereço é o requisitado pela CPU. Caso seja o endereço solicitado, os dados são fornecidos para a CPU em um ciclo de *clock*, sem haver congelamento da CPU (*stall*). Caso os dado não esteja no L1P, o dado é solicitado ao próximo nível de memória na hierarquia, o L2 (*cache* de nível 2). Como a CPU não escreve no L1P (pois, ele só contém instruções), não é preciso considerar a escrita nesses tipo de *cache*.

7.3.2 L1D

Quando a CPU faz um acesso a dados na memória, eles são alocados no L1D. Os dispositivos 621x, 671x e 64x apresentam funcionamento de forma semelhante, diferindo apenas no tamanho do cache e no tamanho da linha. A Figura 7.6 mostra as características do L1D.

A Figura 7.7 mostra a arquitetura do L1D, que é composto pela memória cache e pela lógica de controle.

O endereço de memória, de 32 bits é divido em três partes. Os 6 (5 para os DSPs 621x/671x) bits menos significativos indicam o deslocamento de uma palavra dentro de uma linha. Os bits 6-12 (5-10) indicam a linha do *cache* onde o dado deve ser mapeado. Os bites 13-31 (11-31) são usados como *tag*. Além disso, ainda existem o bit de validade, V, o bit LRU. O bit LRU é usado para implementar a política *Last Recent Used*. Quando a CPU lê um dado da memória, a lógica de controle verifica se o dado já está no *cache*. Como o *cache* é *two-way* associativo, o endereço pode ser mapeado tanto para o banco 1 como para o banco 0. Logo, a lógica de controle verifica em ambos os bancos se o dado está presente. Caso haja um *hit* em um dos bancos, o dado é fornecido para a CPU em um ciclo de *clock* e não há congelamento da CPU. O bit LRU é setado para o valor do outro banco, de forma que este passe a ser o último a ser usado . Se

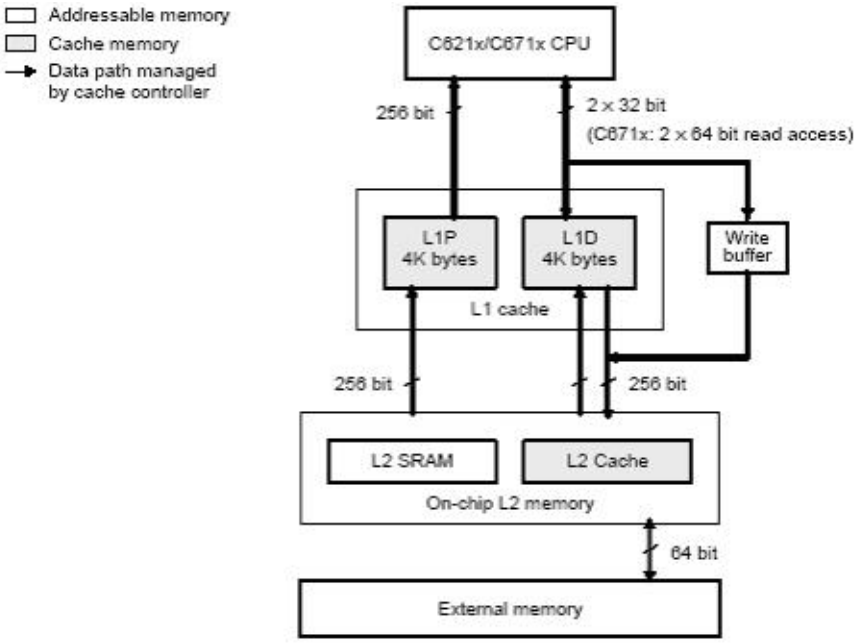

Figura 7.3: Arquitetura de memória – 621x/671x

houver *miss* em ambos os bancos, a requisição é passada para o L2. Quando o dado for obtido, ele será inserido no banco determinado pelo bit LRU. O L1D é um *cache read-allocate*, o que significa que uma linha só é alocada na leitura. Na ocorrência de um *write miss*, o dado é escrito diretamente na memória de nível inferior (mais distante da CPU), contornando o cache L1D. A escrita é feita através de um buffer de escrita (*write buffer*). Se houver um *write hit*, o dado é escrito no *cache*, mas os dados não são passados imediatamente para a memória inferior, ou seja, o cache é do tipo *write-back*. Para marcar as linhas que tiveram os dados modifificados por uma escrita, é usado um bit especial conhecido como *dirty bit* (D). Se a CPU precisar carregar uma nova linha (*read miss*) e o *dirty bit* dessa linha indicar que ela foi modificada por uma operação de escrita (D=1), então é necessário gravar a linha na memória inferior (L2) antes de carregar os dados. Se a linha não tiver sido modificada (D=0), então os dados da linha são sobrescrito.

7.3.3 L2

O segundo nível de *cache* é inserido para reduzir ainda mais o número de acessos a memória principal. O segundo nível de *cache* geralmente é introduzido quando há uma grande diferença de tamanho e velocidade

Characteristic	C621x/C671x DSP	C64x DSP
Organization	Direct-mapped	Direct-mapped
Protocol	Read Allocate	Read Allocate
CPU access time	1 cycle	1 cycle
Capacity	4 Kbytes	16 Kbytes
Line size	64 bytes	32 bytes
Single miss stall	5 cycles	8 cycles
Miss pipelining	No	Yes

Figura 7.4: Características do L1P

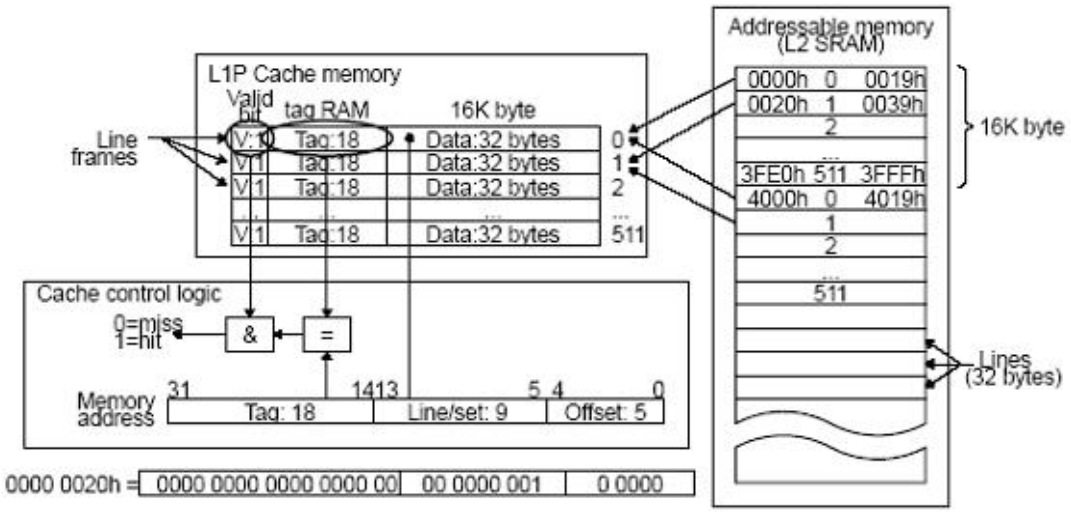

Figura 7.5: Cache L1P

entre o *cache* de nível 1 e a memória principal. O *cache* de nível 2 opera de forma semelhante a um *cache* de nível 1. Ele difere do primeiro por ser maior e mais lento. O segundo nível de memória da família TMS320C6000 pode ser dividido em duas partes: L2 SRAM e L2 *cache*. A primeira é um tipo de memória interna que pode ser endereçada. A segunda funciona como *cache* e é usada para fazer *cache* apenas da memória externa. Diferente dos *cache* de nível 1, o de nível 2 é do tipo *read and write allocate*, ou seja, uma nova linha pode ser alocada tanto na leitura quanto na escrita. Quando a CPU requisita um endereço que causa um *miss* em L1 (L1D ou L1P) e também em L2, o endereço correspondente deve ser buscado na memória mais externa e alocado em L2. A política de substituição é a LRU. Se a linha estiver suja (D=1), então o dados são escritos na memória externa antes de fazer a busca . Uma vez que o dado está em L2, ele é enviado para L1, que o armazena e o envia para a CPU. Quando um dado é passado da CPU para o cache L2 através do *writer buffer* e L2 detecta que ocorreu um miss, então ele busca a linha correspondente na memória principal, modifica-a com o dado da CPU e a aloca no cache L2. Se a linha a onde o dado estiver suja, ela precisa ser gravada na memória principal antes de buscar o dado da memória principal.

7.4 Interface de memória externa

A interface de memória externa (EMIF *External Memory InterFace*) conecta a CPU a uma memória externa. A EMIF possibilita o uso tanto memórias síncronas como SDRAM (*synchronous dynamic RAM*) e SBSRAM (*synchronous burst static* RAM) como de memória assíncronas. Além disso, a EMIF também prove capacidade de leitura de bytes (8 bits) e meia palavra (16 bit), permitindo o uso de memória de baixo custo. Essas memórias são usadas principalmente para guardar informações de boot. Entre elas pode-se destacar (flash, EEPROM, EPROM e PROM). O uso da EMIF possibilita aumentar a memória disponível para CPU. No entanto, essas memórias são muito mais lentas que as memórias internas (L1 e L2).

7.5 Coerência de cache

Uma aspecto fundamental que não deve ser esquecido quando se está trabalhando com um sistema de memória baseado em *cache* é a coerência entre os diferentes níveis de memória. Um protocolo deve ser seguido para que dispositivos como periféricos ou a própria CPU não acessem dados desatualizados. Na Figura 7.8, temos um exemplo de incoerência de *cache*. Suponha que a CPU acesse uma locação de memória que é alocada no *cache* (1). Num momento posterior, uma periférico escreve dados para a mesma locação (2), na memória principal. Uma vez que o endereço de memória é mantido no cache, um novo acesso da CPU resulta na leitura do dado desatualizado armazenado no *cache* (3). O processo descrito resultou em uma incoerência entre as informações contida no cache e na memória.

Characteristic	C621x/C671x DSP	C64x DSP
Organization	2-way set-associative	2-way set-associative
Protocol	Read Allocate, Write-back	Read Allocate, Write-back
CPU access time	1 cycle	1 cycle
Capacity	4 Kbytes	16 Kbytes
Line size	32 bytes	64 bytes
Single read miss stall (L2 SRAM)	4 cycles	6 cycles
Single read miss stall (L2 Cache)	4 cycles	8 cycles
Miss pipelining	No	Yes
Multiple consecutive misses (L2 SRAM)	4 cycles	4 + 2 × M cycles
Multiple consecutive misses (L2 Cache)	4 cycles	6 + 2 × M cycles
Write miss	Passed through 4 × 32-bit write buffer. Only stalls when full.	Passed through 4 × 64-bit write buffer. Only stalls when full.

Figura 7.6: Características do L1D

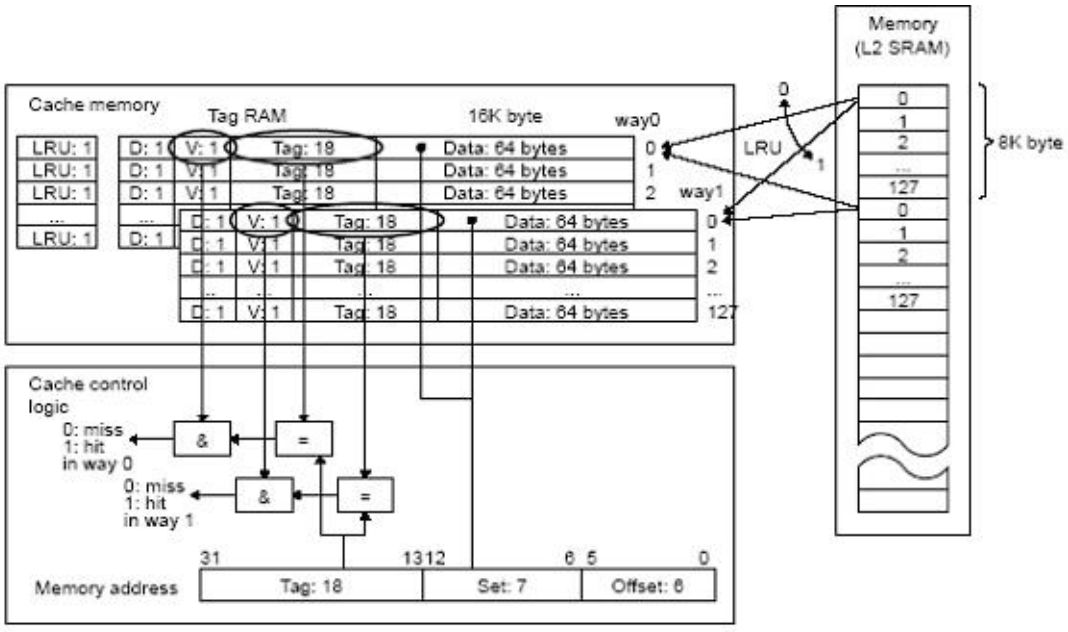

Figura 7.7: Cache L1D

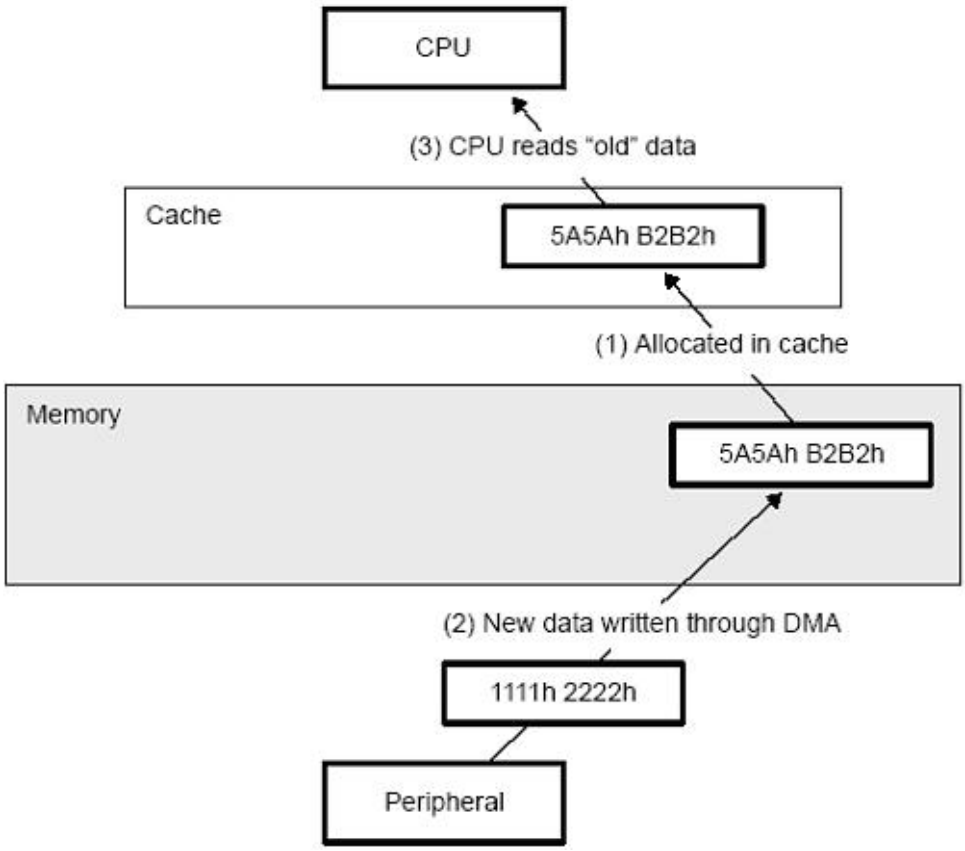

Figura 7.8: Incoerência

O controlador de cache dos dispositivos C621x/C671x e 64x automaticamente mantém o coerência para acessos da CPU e DMA (EDMA) a memória interna. A coerência é garantida pelo uso de um protocolo baseado em comandos snoop, que consiste de uma memória nível inferior verificar se os dados estão armazenados na memória de nível superior. Caso o dado esteja alocado na memória de nível superior, tipicamente um *writeback* ou *invalidate* ocorre. Quando CPU e outros dispositivos (DMA e EDMA) acessam dados na memória externa, não há nenhuma garantia, por parte do sistema de memória, que a coerência será mantida. Logo é responsabilidade de quem usa a memória garantir a coerência dos dados. Para tal, existe uma biblioteca, a *Chip Support Library*, (CSL) que prove um conjunto de funções que permitem ao usuário controlar o *cache*. Essas funções basicamente realizam um *writeback-invalidate* ou, alternativamente, somente para dispositivos C64x, uma operação *invalidate*, que é mais rápida.

Capítulo 8

Aplicações, análise de mercado e benchmark

A seguir, a seção 8.1 apresentada as principais aplicações para a família TMS320C6000. A seção 8.2 mostra a configuração atual do mercado de DSPs. Na seção 8.3 são mostrados as principais ferramentas disponíveis atualmente para a família TMS320C6000. A seção 8.4 apresenta alguns aspectos de *benchmark* relacionados à família TMS320.

8.1 Aplicações

De forma geral, a família TMS320 de DSPs é aplicável para a maioria dos problemas de processamento digital de sinais tais como codificação e compressão de voz (*vocoding*), filtragem, correção de erros entre outros. Além disso, esta família de DSPs é capaz de suportar aplicações complexas que necessitem de múltiplas operações sendo feitas simultaneamente. Outras aplicações importantes são mostradas na Tabela 8.1.

Automotivo	**Consumidor**	**Controle**
Telefones celulares	Rádios e TVs digitais	Impressoras a laser
Posicionamento global	Brinquedos educativos	Controle robótico
Navegação	Pagers	Controle de drives de disco
Comandos de voz	Detectores de radar	
Fins gerais	**Imagens e gráficos**	**Indústrias**
Filtragem adaptativa	Computação 3D	Controle numérico
Convolução	Processamento homomórfico	Acesso seguro
Correlação	Compressão/transm. de imagens	
Trasformações de Fourier	Reconhecimento de padrões	
Transformações de Hilbert	Visão robótica	
Medicina	**Militares**	**Telecomunicações**
Equipamentos de diagnóstico	Processamento imagens	Cancelamento de eco
Monitoramento fetal	Guia de mísseis	Encriptação de dados
Equipamentos de ultrasom	Comunicações seguras	Controle de estações base
Monitoramento remoto de pacientes	Processamento de sonares	Equalizadores

Tabela 8.1: Aplicações da família TMS320 de DSPs

8.2 Participação de mercado

Os quatro principais fabricantes no mercado de DSPs são *Texas Instruments*, *Lucent*, *Analog Devices* e *Motorola*. Estas companhias compartilham cerca de 90% do mercado como mostrado na Figura 8.1.

Um resumo dos pontos fortes e fracos de cada um dos principais líderes do mercado de DSPs é apresentado na Tabela 8.2.

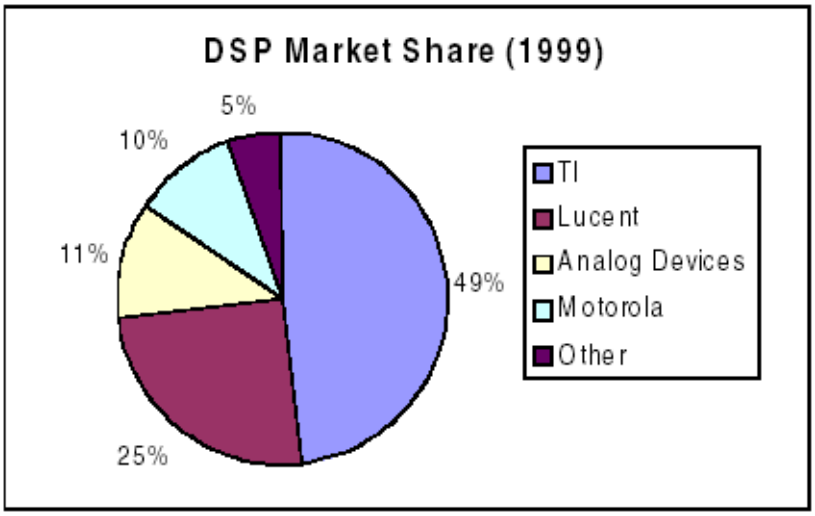

Figura 8.1: Distribuição do mercado de DSPs

Companhia	**Pontos Fortes**	**Pontos Fracos**
Texas Instruments	Líder de mercado, foco em DSPs, extenso grupo de P&D.	O foco em DSPs a deixa fraca em segmentos de outros tecnologias.
Lucent/Agere Systems	Parceria com a Motorola	Lentidão para colocar produtos no mercado.
Analog Devices	Parceria com a Intel	Foco em muitos dispositivos a deixa sem uma política mais consistente no mercado de DSPs.
Motorola	Parceria com a *Lucent/Agere*	Muitos de seus clientes potenciais são competidores de mercado.

Tabela 8.2: Principais fabricantes de DSPS: pontos fortes e pontos fracos

A *Texas Instruments* ao possuir uma grande fatia do mercado de DSPs (50%), consegue ter recursos suficientes para continuar seu já bem conhecida política de pesquisa e desenvolvimento. Isto resulta em gerações de DSPs que resolvem grande parte dos problemas práticos da atualidade como é o caso da família TMS320C6000.

A figura 8.2 apresenta as principais perspectivas para a família TMS320.

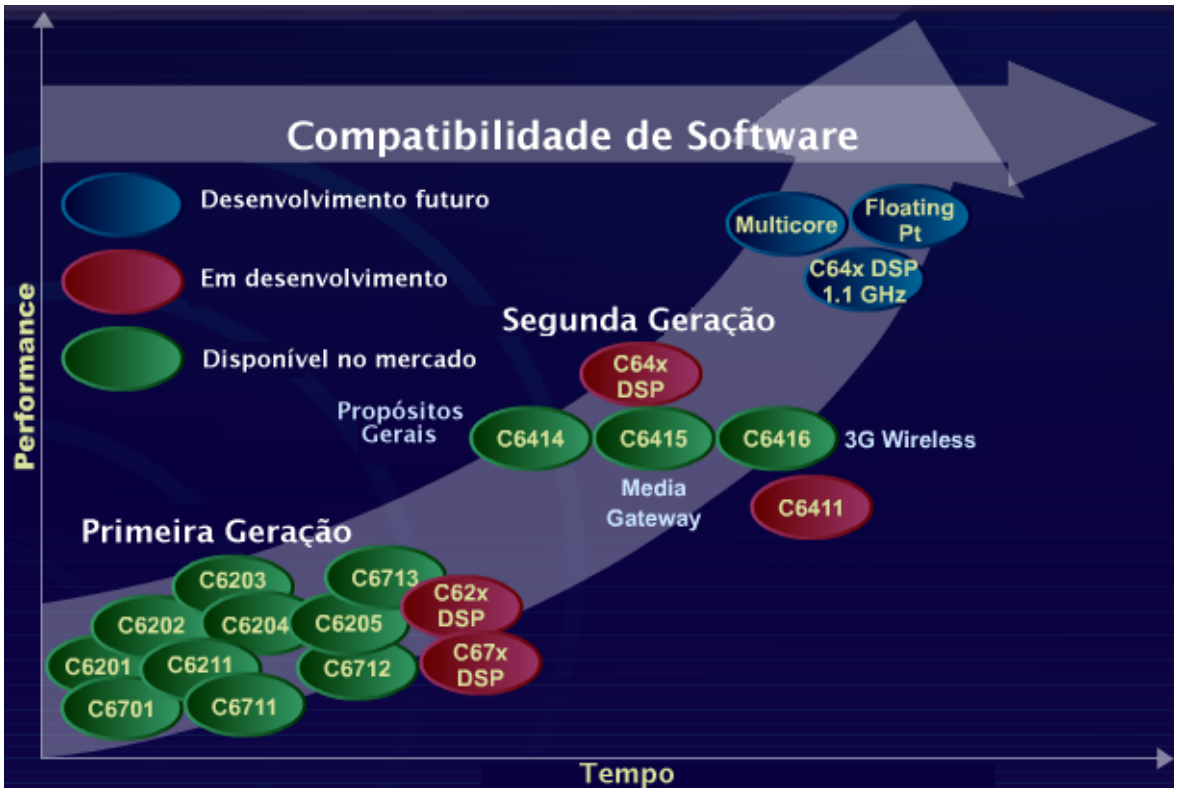

Figura 8.2: Perspectivas para a família de DSPs TMS320

Em um aparente acompanhamento das tendências de mercado, a *Texas* parece preocupada em disponibilizar as mais diversas soluções aos seus clientes. Com os DSPs TMS de segunda geração vários problemas da área de telefonia 3G passam a ser resolvidos. Continuando um processo dinâmico, vários dispositivos estão sendo estudados como possíveis soluções para a TV digital, VoIP (voz sobre IP) entre outros problemas para um futuro próximo. Isto seria a terceira geração de TMSs da *Texas Instruments*.

8.3 Softwares disponíveis

A família TMS320C6000 conta com um conjunto completo de desenvolvimento tanto para *PC* quanto para *Sun Workstations*:

- Compilador C
- Otimizador *assembler*
- *Assembler*
- *Linker*
- Ferramentas de avaliação
  - Interface *Windows*^{TM} de *debugger*
  - Simulador
  - Emulador de *hardware*.

8.3.1 Descrição das ferramentas

O desenvolvimento de aplicações para o TMS320C6000 é baseado no avançado compilador C e no revolucionário otimizador utilizado. O compilador e o otimizador eliminam a necessidade de um um conhecimento extensivo da arquitetura DSP durante o processo de desenvolvimento. Através do desenvolvimento de código C independente e altamente estruturado, o tempo de desenvolvimento é bastante reduzido ao mesmo tempo em que é mantido a performance inerente relacionada à arquitetura VLIW.

A interface gráfica para para *debugging* é bastante intuitiva. Com esta interface, é possível acompanhar os estados da pilha de execução, memória, registradores e também visualiar estruturas como menus e interfaces gráficas desenvolvidas. A ferramente oferece um *Profiler* responsável por indicar possíveis otimizações em tempo de desenvolvimento.

A Figura 8.3 apresenta o *compilador/profiler* disponível em um dos kits de desenvolvimento.

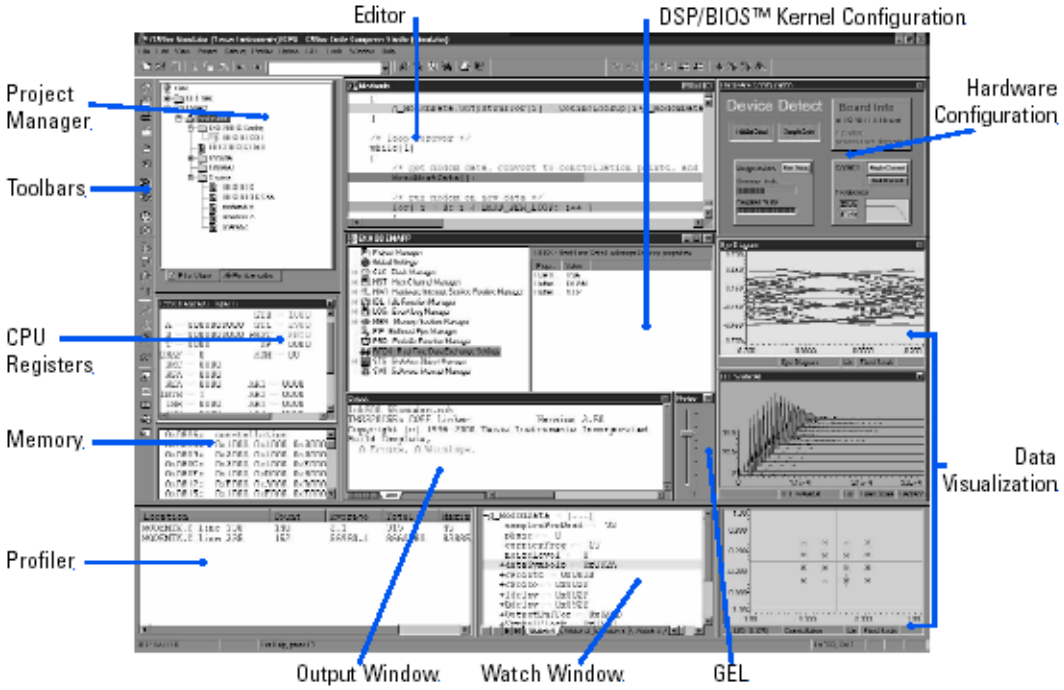

Figura 8.3: Interface do compilador/profiler do TMS320

8.3.2 Suporte

A *Texas Instruments* oferece em seu *site* `www.ti.com`, completo suporte sobre a família TMS320C6000. São disponibilizados documentos técnicos, manuais, análises de *benchmarks*, artigos entre outros.

8.3.3 *Kit* de desenvolvimento

O *kit* de desenvolvimento completo pode ser encontrado no *site* . Os preços variam de acordo com as necessidades do comprador. Um conjunto básico com um DSP TMS320C6713, compilador C (não otimizado), *debugger*, simulador e *linker* pode ser encontrado (em junho de 2004) por $395 dólares.

Existem *kits* específicos a vários tipos de desenvolvimento tais como para aplicações em rede, processamento de imagens, mecânica e controle em indústricas entre outros.

8.3.4 Parceiros

Devida a sua grande participação de mercado, a *Texas* conta com um grande número de parceiros. Estes parceiros são responsáveis pelo desenvolvimento de aplicativos com as mais diversas características. Abaixo estão listados *alguns* parceiros e os produtos desenvolvidos por eles.

- **Ariel Corp**. Produtos para telefonia.
- **Cheops GmbH & Co**. Processamento de imagens médicas.
- **D.SignT**. Integração de serviços. Processamento de sinais.
- **Eonic Systems**. Sistemas de tempo real.
- **White Montain**. Ferramentas de emulação e aplicações multiplataforma.

8.4 Benchmarks

Como fonte dos *benchmarks* aqui apresentados temos *Berkeley Design Technology, Inc.* (BDTI - http://www.bdti.com) e *Embedded Microprocessor Benchmark Consortium* (EEMBC - http://www.eembc.org). Estas são duas organizações independentes que trabalham com analise de desempenho de dispositivos.

Na Tabela 8.3 apresentamos um quadro comparativo, produzido pela EEMBC, demonstrando a posição do dispositivo TMS320C6203 (300 MHz) perante alguns de seus concorrentes em aplicações na área de telecomunicações. Neste quadro podemos ver que os processadores da *Texas* possuem pontuação melhores que os concorrentes. Infelizmente não conseguimos *benchmarks* com os processadores possuindo o mesmo *clock*, assim como com otimizações.

Processador - *clock*	**AD21065L – 60**	**MPC603e – 300**	**TMS320C6203 – 300**	**TMS320C6203 – 300**	**TMPR4927ATB – 200**
$Tele_{mark}$	1.1	5.8	68.5	6.8	2.9
Tipo de certificação	OOTB	OOTB	Otimizado	OOTB	OOTB
Certificação	07/11/2000	20/02/2001	22/03/2000	22/03/2000	12/07/2002
Observações			*Assembly* otimizado		
Tipo de dado nativo	32	32	16	16	64/32
Arquitetura	DSP	RISC	DSP VLIW	DSP VLIW	RISC
Cache L1 de inst. (KB)	0	16	0	0	32
Cache L1 de dados (KB)	0	16	0	0	32
Largura de barramento externo	32	64	32	32	64
Clock de memória (mhz)	60	66,7	300	300	100

Tabela 8.3: Quadro comparativo com diversos dispositivos aplicados a área de telecomunicações.

As informações exibidas a seguir foram extraídas do artigo [Williston et al., 2003]. Neste artigo são mostradas uma série de análises a respeito de diversas famílias de dispositivos, vamos nos concentrar aqui nos da família ‘C64x da *Texas*, nos da família 21535 da *Analog*, e nos da família PXA2xx da *Intel*. Dentre os parâmetros que serão analisados encontra-se velocidade, eficiência em consumo, eficiência em utilização da memória e custo. As informações estão dispostas em um gráfico radar e cada um dos parâmetros está simbolizado em um ícone, da seguinte forma:

Velocidade representado por uma onça, é a pontuação BDTI do processador.

Eficiência em utilização da memória representado por um disquete, é o inverso do uso de memória.

Acessibilidade representado por notas (dinheiro), é o inverso do preço por 10.000 unidades.

Eficiência em consumo representado pela bateria,

Pontuações maiores representam melhor desempenho em todos os casos. Para todas as métricas é utilizada uma escala linear.

Na Figura 8.4 temos o desempenho do TMS320C5409 de 120 MHz, um processador intermediário da família ‘C54x da *Texas Instruments*. Ele será tomado como referência para todas as análises por se tratar de um processador popular entre os DSPs. Nos gráficos haverá sempre um intervalo de valores para cada parâmetro, isso ocorre devido as variações entre os membros da família.

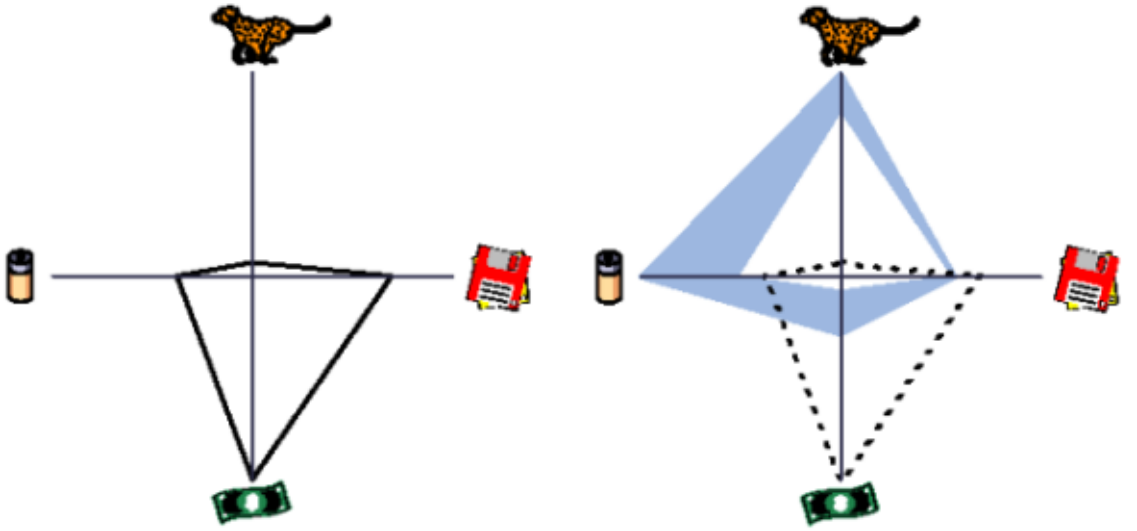

Figura 8.4: Desempenho do TMS320C5409, e ‘C64x

Na Figura 8.4 é mostrado o gráfico a respeito do TMS320C64x (2000). Estes dispositivos são uma evolução da família de dispositivos ‘C62x, uma família de DSPs de ponto fixo da *Texas* lançada em 1997. Os dispositivos ‘C64x estão baseados em uma arquitetura VLIW que permite a execução de até 8 instruções em paralelo. Os dispositivos desta família podem atingir velocidade de até 600 MHz, porém estes dispositivos têm como desvantagem para os outros aqui apresentados um *pipeline* longo (11 fases contra um máximo de 8) e latência de unidade funcional para muitas de suas instruções. Eles possuem dois níveis de memória, o primeiro é interno ao processador (sempre funciona como cache) e parte do segundo pode ser configurado como cache.

Na Figura 8.5 é mostrado o gráfico a respeito do ADSP-21535. Este dispositivo contém dois caminhos de dados de ponto fixo (2 unidades MAC, 2 ALUs, e um deslocador) e duas unidades de cálculo de endereço.

Ele pode usar emissão múltipla de instruções para executar uma instrução aritmética e duas instruções de movimentação de dados em um único ciclo. Suas instruções são mais complexas que as do ‘C64x. Também possui dois níveis de memória.

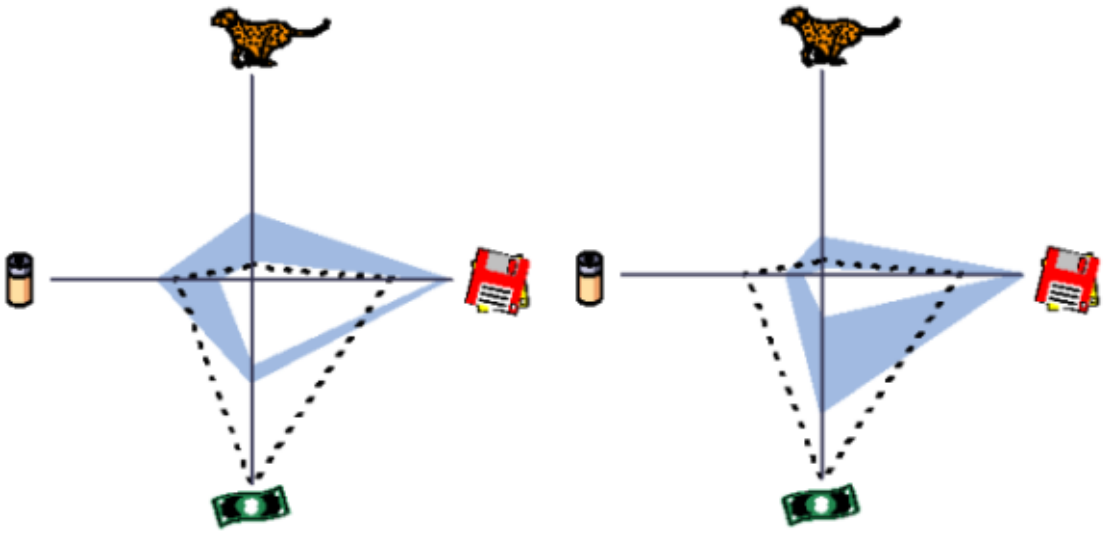

Figura 8.5: Desempenho do *Analog* ADSP-21535 e *Intel* PXA2xx

Na Figura 8.5 é mostrado o gráfico a respeito do *Intel* PXA2xx. Estes dispositivos possuem um *pipeline* em sete estágios e é compatível com uma série de processadores anteriores da *Intel*.

O que pode ser concluído com todas estas informações é que a *Texas* prioriza velocidade e efeciência em consumo. Já os outros fabricantes estão preocupados com uma utilização mais eficiente da memória e com os custos mais baixos.

Na Figura 8.6 é mostrado uma série de gráficos comparativos do ’C67x com alguns dos concorrentes. Em todos estes casos *Transformada de Fourier* em 256 pontos. No primeiro gráfico, temos o tempo de execução (em microsegundos), quanto menor melhor. No segundo, o custo por tempo de execução (em $/microsegundos), quanto menor melhor. No terceiro, o consumo de energia (watts/microsegundos), quanto menor melhor; e temos a seguintes configurações: ADSP 895 mWatt para 100 MHz e 1,8 V, TMS 685 mWatt para 200 MHz e 1,2 V, SH 4 285 mWatt para 200 MHz e 1,5 V. No quarto, temos o uso total da memória (em Bytes). Esta análise pode ser encontrada em [BDTI – Berkeley Design Technology, Inc., 2003].

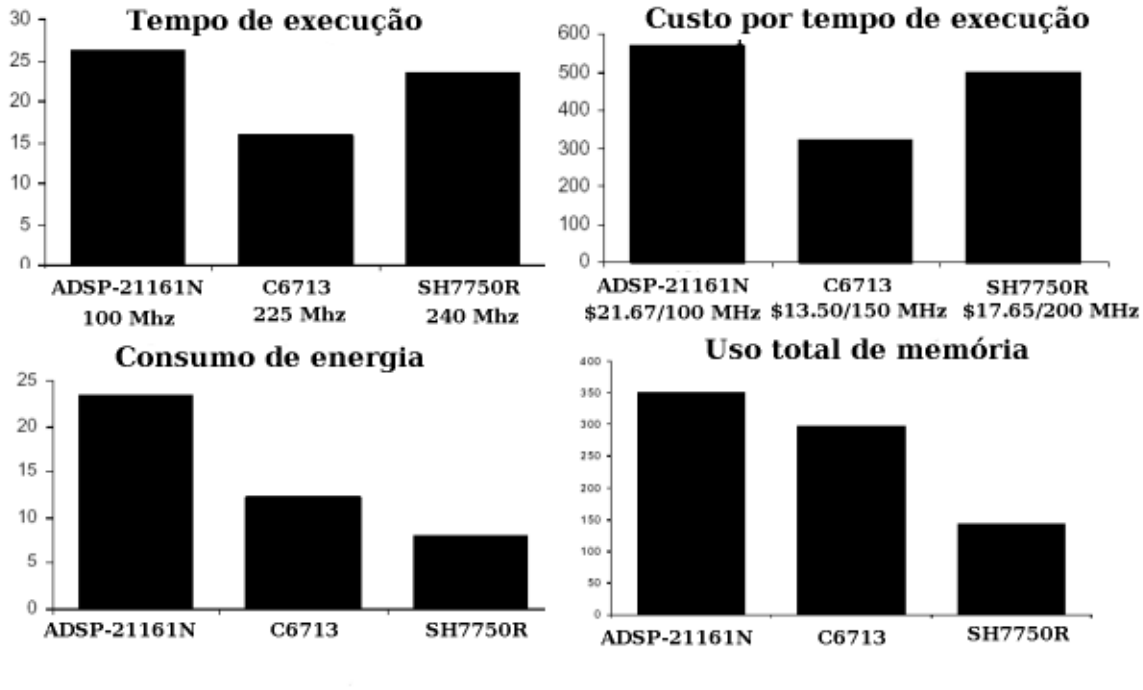

Figura 8.6: Desempenho do ‘C67x perante os concorrentes

Referências Bibliográficas

[Dunn, 2000] Dunn, D. (2000). Ti raises bar with new dsp families. Electronic Buyers News, pag 34, 28/02/2000.

[BDTI – Berkeley Design Technology, Inc., 2003] BDTI – Berkeley Design Technology, Inc. (2003). Texas instruments TMS320C67x. Disponível em `http://www.ece.utexas.edu`.

[Texas Instruments, 2004a] Texas Instruments (2004a). Tms320c6000 – cpu and instruction set reference guide. Disponível em `http://dspvillage.ti.com/docs/`.

[Texas Instruments, 2004b] Texas Instruments (2004b). Tms320c6000 – enhanced direct memory access (edma) controller reference guide. Disponível em `http://dspvillage.ti.com/docs/`.

[Texas Instruments, 2004c] Texas Instruments (2004c). Tms320c6000 – manual update sheet. Disponível em `http://dspvillage.ti.com/docs/`.

[Texas Instruments, 2004d] Texas Instruments (2004d). Tms320c6000 – peripherals overview reference guide. Disponível em `http://dspvillage.ti.com/docs/`.

[Texas Instruments, 2004e] Texas Instruments (2004e). Tms320c6000 – power-down logic and modes reference guide. Disponível em `http://dspvillage.ti.com/docs/`.

[Texas Instruments, 2004f] Texas Instruments (2004f). Tms320c6000 – programmer's guide. Disponível em `http://dspvillage.ti.com/docs/`.

[Texas Instruments, 2004g] Texas Instruments (2004g). Tms320c6000 – technical overview. Disponível em `http://dspvillage.ti.com/docs/`.

[Texas Instruments, 2004h] Texas Instruments (2004h). Tms320c6000 – two-level internal memory. Disponível em `http://dspvillage.ti.com/docs/`.

[Texas Instruments, 2004i] Texas Instruments (2004i). Tms320c6000 – user's guide. Disponível em `http://dspvillage.ti.com/docs/`.

[Texas Instruments, 2004j] Texas Instruments (2004j). Tms320c6201 – fixed point digital signal processor. Disponível em `http://dspvillage.ti.com/docs/`.

[Texas Instruments, 2004k] Texas Instruments (2004k). Tms320c62x/c67x – power consumption summary. Disponível em `http://dspvillage.ti.com/docs/`.

[Talla et al., 2000] Talla, D., John, L. K., Lapinskii, V., and Evans, B. L. (2000). Evaluating signal processing and multimedia applications on simd, vliw and superscalar architectures. Disponível em `http://dspvillage.ti.com/docs/`.

[Williston et al., 2003] Williston, K., Tsai, M., and Bier, J. (2003). Dsp benchmark results for the latest processors. Disponível em `http://www.techonline.com`.

[Xu et al., 2001] Xu, D., Zhong, L. C., and Dunn, T. (2001). Opportunities for texas instruments in the digital signal processing market. Disponível em `www.forwardconcepts.com`.

[]